



Intel® 64 and IA-32 Architectures Optimization Reference Manual

Documentation Changes

Document Number: 355308-051US

October 2026

Notices & Disclaimers

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

No product or component can be absolutely secure.

Your costs and results may vary.

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

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

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

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

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

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

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

Contents

Revision History	4
Preface	5
Nomenclature	5
Summary Tables of Changes	5
Documentation Changes	5

Revision History

Date	Revision	Description
August 2023	048	Initial release of document.
January 2024	049	Q1 Release
April 2024	050	Q2 Release
October 2026	051	Q3 Release

Preface

This document is an update to the optimization recommendations contained in the **Intel® 64 and IA-32 Architectures Optimization Reference Manual**, also known as:

- Software Optimization Manual (SOM)
- Optimization Manual
- Optimization Reference Manual (ORM)

This document is a compilation of device and documentation errata, specification clarifications and changes. It is intended for hardware system manufacturers and software developers of applications, operating systems, or tools.

Nomenclature

Documentation Changes include gross content changes or omissions from the current published specifications . These changes are those beyond typos, capitalization, or basic edits. These will be incorporated in any new release of the specification upon approval.

Summary Tables of Changes

The following table indicates documentation changes which apply to the Intel® 64 and IA-32 Architecture software optimization topics covered by this reference manual.

No.	DOCUMENTATION CHANGES
1	Updates to Chapter 2

Documentation Changes

Changes to the **Intel® 64 and IA-32 Architectures Optimization Reference Manual** volumes follow, and are listed by chapter. Only chapters with changes are included in this document.

1. Updates to Chapter 2

Change bars and **violet** text show changes to Chapter 2 of the Intel® 64 and IA-32 Architectures Optimization Reference Manual: **Intel® 64 and IA-32 Processor Architectures**.

Changes to this chapter:

- Section 2.1
 - Section 2.4.1, “Golden Cove Microarchitecture Overview”: Updated Figure 2-2 and Table 2-3.

CHAPTER 2

INTEL® 64 AND IA-32 PROCESSOR ARCHITECTURES

This chapter overviews features relevant to software optimization for current generations of Intel® 64 and IA-32 processors¹. These features include:

- Microarchitectures that enable executing instructions with high throughput at high clock speeds, a high-speed cache hierarchy, and a high-speed system bus.
- Intel® Hyper-Threading Technology² (Intel® HT Technology) support.
- Intel 64 architecture on Intel 64 processors.
- Single Instruction Multiple Data (SIMD) instruction extensions: MMX™ technology, Streaming SIMD Extensions (Intel® SSE), Streaming SIMD Extensions 2 (Intel® SSE2), Streaming SIMD Extensions 3 (Intel® SSE3), Supplemental Streaming SIMD Extensions 3 (SSSE3), Intel® SSE4.1, and Intel® SSE4.2.
- Intel® Advanced Vector Extensions (Intel® AVX).
- Half-precision floating-point conversion and RDRAND.
- Fused Multiply Add Extensions.
- Intel® Advanced Vector Extensions 2 (Intel® AVX2).
- ADX and RDSEED.
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512).
- Intel® Thread Director.

2.1 6TH GENERATION INTEL® XEON® SCALABLE PROCESSOR FAMILY

2.1.1 THE REDWOOD COVE MICROARCHITECTURE

The Redwood Cove microarchitecture is the successor of the Golden Cove microarchitecture. The Redwood Cove microarchitecture introduces the following enhancements:

- Improvements for larger code footprint workloads.
- Larger instruction cache: 32K→64K.
- Branch Hint x86 architecture extension.
- Code Software Prefetch x86 architecture extension (Granite Rapids only).
- Improved LSD coverage: the IDQ can hold 192 μ ops per logical processor in single-thread mode or 96 μ ops per thread when SMT is active.
- Improved branch prediction and reduced average branch misprediction recovery latency.
- New LD+OP and MOV+OP macro fusions.

-
1. Intel® Atom® processors are covered in [Chapter 4, “Intel Atom® Processor Architectures.”](#)
 2. Intel® HT Technology requires a computer system with an Intel processor supporting hyper-threading and an Intel® HT Technology-enabled chipset, BIOS, and operating system. Performance varies depending on the hardware and software used.

- EXE: 3-cycle Floating Point multiplication.
- Improved SMT performance and efficiency.
- Improved load lock performance.
- New HW data prefetcher to recognize and prefetch the “Array of Pointers” pattern.
- Mid-level-cache size increased to 2MBs for Client.
- Improved Memory Bandwidth.
 - Increased number of outstanding misses (48→64 Deeper MLC miss queues).
 - LLC Page Prefetcher.
- AMX supports FP16 for AI/ML (Granite Rapids only).

2.1.1.1 Branch Hint

If the branch predictor has stored information about a conditional branch, the predictor should use this information to predict the branch. If the predictor does not have stored information, the predictor predicts the branch to be **not-taken**. This is usually, but not always, correct.

Starting with the Redwood Cove microarchitecture, if the predictor has no stored information about a branch, the branch has the Intel® SSE2 branch taken hint (i.e., instruction prefix 3EH),

When the codec decodes the branch, it flips the branch’s prediction from not-taken to taken. It then flushes the pipeline in front of it and steers this pipeline to fetch the **taken path** of the branch.

If the branch is taken when executed, the hint reduces the misprediction penalty by replacing an execution time pipeline flush and re-steer with a decode time of one. Alternatively, if the branch is not-taken, the hint increases the branch misprediction penalty by introducing a new decode time and a new execution time flush and re-steer. Consequently, the hint should not be added to a **not-taken branch** since this increases the program’s run time (see [Section 2.1.1.2](#)).

The hint is only used when the predictor does not have stored information about the branch. To avoid code bloat and reducing the instruction fetch bandwidth, don’t add the hint to a branch in hot code—for example, a branch inside a loop with a high iteration count—because the predictor will likely have stored information about that branch. Ideally, the hint should only be added to infrequently executed branches that are mostly taken, but identifying those branches may be difficult. Compilers are advised to add the hints as part of profile-guided optimization, where the one-sided execution path cannot be laid out as a fall-through. The Redwood Cove microarchitecture introduces new performance monitoring events to guide hint placement (see [Section 2.1.1.3](#)).

Judiciously adding hints can reduce the program’s run time by reducing the branch misprediction penalty. Hints are especially useful for large code footprint workloads where the number of branches exceeds the predictor’s capacity.

Details include:

- The hint only applies to JCCs, not CXZ and LOOP/LOOPCC¹.
- The microarchitecture is obliged to decode the Intel® SSE2 branch not-taken hint (i.e., instruction prefix 2EH), but otherwise ignores it.
 - There’s no need to hint that a branch is not-taken since, by default, it’s predicted not-taken if the predictor doesn’t have stored information about it.
- An instruction has the taken hint if it has at least one instruction prefix 3EH.
 - **Example:** If an instruction has the 3EH and 2EH prefixes, regardless of order, it still has the taken the hint.

1. As specified in section 2.1.1 of the [Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A Chapter 2, “Instruction Format”](#).

2.1.1.2 Profiling Support for Branch Hints

The following performance monitoring events may be used to identify branches that may benefit from adding the previous hint:

- **FRONTEND_RETIRED.MISP_ANT** (Event Code = C6H, Unit Mask = 02H): Always Not-Taken conditional branch instructions retired that were miss-predicted.
- **FRONTEND_RETIRED.ANY_ANT** (Event Code = C6H, Unit Mask = 03H): Always Not-Taken conditional branch instructions retired.
- **BR_INST_RETIRED.COND** (Event Code = C4H, Unit Mask = 11H): conditional branch instructions retired.

Note that collecting Last Branch Records (LBR) when sampling on the last two events is instrumental to determining the polarity of the branch (e.g., was the branch mostly taken or mostly not taken).

2.1.1.3 New Redwood Cove Macro-Fusions

The Redwood Cove microarchitecture supports two new types of instruction macro-fusion: MOV-OP and LOAD-OP. The new macro-fusions are only supported out of the μ op-Cache (not supported on the legacy decode pipeline).

With a new MOV+OP macro-fusion, a register MOV instruction can be fused with following an OP instruction to form the non-destructive source operation form: $c \leftarrow a \text{ op } b$, preserving both source operands.

For example:

Table 2-1. MOV+OP Macro-Fusion Example

Class	Example	Macro-Fused to:
MOV+OP	mov rax, rbx sub rax, rcx	rax = sub rbx, rcx

The two instructions are macro-fused to a single micro-operation cached in the μ op-Cache. The macro-fused operation takes a single slot in IDQ and during allocation, execution, and retirement.

Similarly, with a new LOAD+OP macro-fusion, a LOAD instruction could be macro-fused by following an OP instruction to form a fused load-op micro-operation, which often cannot be encoded within the x86 instruction set.

For example:

Table 2-2. LD+OP Macro-Fusion Example

Class	Example	Macro-Fused to:
LD+OP	mov eax, [mem] sub eax, ebx	eax = sub [mem], ebx ¹

NOTES:

1. Note that only sub reg, [mem] is encodable within x86 instruction set.

Limitations of LOAD+OP Macro-Fusion

The following instruction pair can be macro-fused to the single `DST:= OP [mem], SRC_REG` micro-operation.

(1) MOV DST, [mem]

(2) OP DST, SRC_REG // SRC_REG cannot be immediate and must be different from DST

- LOAD and OP operations must be the same Operand Size 32/64 matching load data size.
- Load cannot be encoded using RIP-Relative form or have Index operand.
- Only integer and packed SIMD/floating point 128-bit vector operations are supported.
- The LOAD must be:
 - An integer load operation without sign-extension (opcode 0x8B).
 - SSE packed load instructions (movaps/pd, movups/pd, movdqa/u).
- No VEX or EVEX forms are supported.

2.1.1.4 Improved Memory BW

The Redwood Cove microarchitecture improves the peak memory bandwidth that a core can consume over the Golden Cove microarchitecture. This significantly increases the depth of the MLC miss queue (48→64 outstanding MLC misses) and introduces new hardware prefetcher algorithms.

2.1.1.5 Array of Pointers Prefetcher

An array of pointers is an array whose elements are pointers. The data associated with each element of the array is usually dynamically allocated elsewhere and can be accessed by dereferencing the pointer pointing to it. For example:

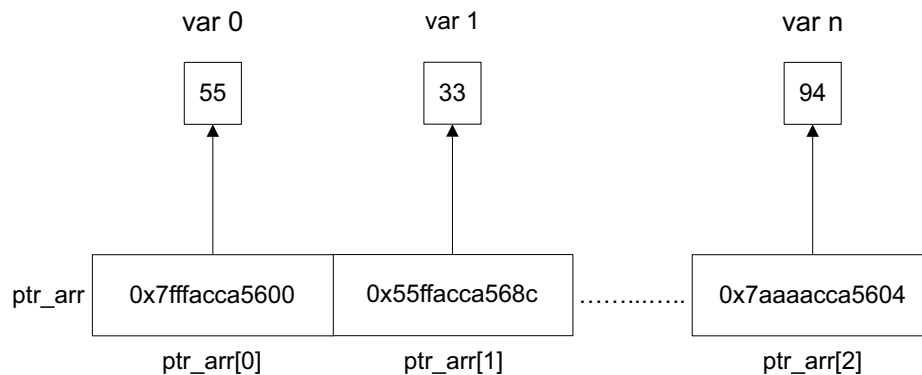


Figure 2-1. Layout of Array of Pointers Prefetchers

The Redwood Cove microarchitecture adds a new Array of Pointers (AOP) hardware prefetcher for detecting and prefetching an array of pointers references into the cache hierarchy. The AOP prefetcher treats the data prefetched for a constant stride load as a pointer and may issue prefetch requests to the memory addresses corresponding to the pointer's value.

Successful detection of the pattern allows AOP Prefetch to bring data to be accessed by the program in advance, saving costly cache misses.

To benefit most from the AOP Prefetching capabilities, we recommend the following:

The array of pointers pattern, in a nutshell, may look like this:

```
for (i=0; i<array_length; i++) {
    (1)mov rax, array[i]
    (2)mov rbx, [rax + offset]; data of rax used as a pointer
}
```

In the Redwood Cove microarchitecture, to benefit from AOP Prefetch, software must ensure that:

- 1st load operation is a regular integer load, naturally aligned to its data size (64-bit or 32-bit in legacy or compatibility mode)
- 2nd load operation depends upon the first load through the base operand and has no index operand.
 - The AOP Prefetcher does not track and would issue a prefetch ignoring the offset. Therefore, it is recommended to use small offsets or avoid using offsets at all.
 - The AOP Prefetcher avoids triggering prefetch if the offset exceeds 32 bytes.

Software that observes any issue with the AOP Prefetcher engine (for example, exceeded memory bandwidth due to redundant prefetch requests) can deactivate it through bit 7 of MSR 0x1A4 (MSR_PREFETCH_CONTROL). The AOP Prefetch is a data-dependent prefetcher. Security implications of data-dependent prefetching are discussed separately at:

<https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/data-dependent-prefetcher.html>

2.1.1.6 LLC Page Prefetcher (LLCPP)

The LLC Page Prefetcher extends the Next-Page-Prefetcher algorithm, which detects a sequence of accesses that are about to reach the end of a page (either going up towards the next page in memory or down towards the previous page in memory) and triggers a linear prefetch to the start of the next predicted page. NPP's intends to prefetch the translation of the next page and initiate prefetching of the first few cache lines from that predicted page.

The LLC Page Prefetch enhances Next-Page Prefetcher in two significant ways:

- It issues prefetches two pages ahead in linear address space.
- The prefetch attempts to bring the entire 4Kb page to the last level cache using opportunistic IDI request slots without the DCU or MLC buffers.

LLCPP is effective when streaming through memory over a buffer greater than 8KB. Smaller buffer access might result in redundant prefetching to the last level cache.

Software that observes any issue with the LLC Page Prefetcher engine (for example, exceeded memory bandwidth due to redundant prefetch requests) can disable LLC Page Prefetcher through bit 6 of MSR 0x1A4 (MSR_PREFETCH_CONTROL).

2.2 THE SAPPHIRE RAPIDS MICROARCHITECTURE

Intel processors based on Sapphire Rapids microarchitecture use Golden Cove cores and support the following additional features:

- Intel® Advanced Matrix Extensions (Intel® AMX) ([Chapter 20](#)).
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512) ([Chapter 18](#)).
- Intel® Data Streaming Accelerator (Intel® DSA)¹.

- Intel® In-Memory Analytics Accelerator (Intel® IAA)¹.
- Intel® Quick Assist Technology (Intel® QAT)([Chapter 21](#))

2.2.1 4TH GENERATION INTEL® XEON® SCALABLE FAMILY OF PROCESSORS

Intel's fourth generation Xeon® Scalable Family of Processors changes from a single-die monolithic design to multi-die Tiles.

- The server products are scalable from dual-socket to eight-socket configurations.
- The I/O is increased with PCI Express 5.0, DDR5 memory, and Compute Express Link 1.1.
- Packaging includes a multi-die chip with up to 4 tiles. Each tile is a 400mm² SoC, providing compute cores and I/O.
- Each tile contains 15 Golden Cove cores (see [Section 2.4](#)). Its memory controller provides two channels of DDR5, up to eight channels across four tiles, and 28 PCIe 5.0 lanes, up to a maximum of 112 across four tiles.

2.3 THE ALDER LAKE PERFORMANCE HYBRID ARCHITECTURE

The Alder Lake performance hybrid architecture combines two Intel architectures, combining the Golden Cove performant cores and the Gracemont efficient Atom cores onto a single SoC. For details on the Golden Cove microarchitecture, see [Section 2.4](#). For details on the Crestmont microarchitecture, see [Section 4.1](#).

2.3.1 12TH GENERATION INTEL® CORE™ PROCESSORS SUPPORTING PERFORMANCE HYBRID ARCHITECTURE

12th Generation Intel® Core™ processors supporting performance hybrid architecture comprise up to eight Performance cores (P-cores) and eight Efficient cores (E-cores). These processors also include a 3MBs Last Level Cache (LLC) per IDI module, where a module is one P-core or four E-cores. It has a symmetrical ISA and comes in a variety of configurations.

P-cores provide single or limited thread performance, while E-cores help provide improved scaling and multithreaded efficiency. These processors' P-cores can also have Intel Hyper-Threading Technology enabled. When the operating system (OS) decides to schedule all processors, all cores can be active simultaneously.

A key OSV requirement for enabling hybrid is symmetric ISA across different core types in a performance hybrid architecture. In 12th Generation Intel Core processors supporting performance hybrid architecture, ISA is converged to a common baseline between the P-cores and E-cores. To maintain symmetric ISA, the E-cores do not support the following features: Intel AVX-512, Intel AVX-512 FP-16, and Intel® TSX. The E-cores do support Intel AVX2 and Intel AVX-VNNI.

2.3.2 HYBRID SCHEDULING

2.3.2.1 Intel® Thread Director

Intel® Thread Director monitors software in real-time, giving hints to the operating system's scheduler, allowing it to make more intelligent and data-driven decisions on thread scheduling. With Intel Thread Director, the hardware provides runtime feedback to the OS per thread based on various IPC performance characteristics in the form of:

1. Please see the [Intel® DSA Specification](#) and [Intel® DSA User Guide](#).
1. Please see the [Intel® IAA Specification](#).

- Dynamic performance and energy efficiency capabilities of P-cores and E-cores based on power/thermal limits.
- Idling hints when power and thermal are constrained.

Intel Thread Director was first introduced in desktop and mobile variants of the 12th generation Intel Core processor based on Alder Lake performance hybrid architecture.

A processor containing P-cores and E-cores with different performance characteristics challenges the operating system's scheduler. Additionally, different software threads see different performance ratios between the P-cores and E-cores. For example, the performance ratio between the P-cores and E-cores for highly vectorized floating-point code is higher than that for scalar integer code. So, when the operating system must make an optimal scheduling decision, it must be aware of the characteristics of the software threads that are candidates for scheduling. Suppose there are insufficient P-cores and a mix of software threads with different characteristics. In that case, the operating system should schedule those threads that benefit most from the P-cores onto those cores and schedule the others on the E-cores.

Intel Thread Director provides the necessary hint to the operating system about the characteristics of the software thread executing on each of the logical processors. The hint is dynamic and reflects the recent characteristics of the thread, i.e., it may change over time based on the dynamic instruction mix of the thread. The processor also considers microarchitecture factors to define the dynamic software thread characteristics.

Thread specific hardware support is enumerated via the CPUID instruction and enabled by the operating system via writing to configuration MSRs. The Intel Thread Director implementation on processors based on Alder Lake performance hybrid architecture defines four thread classes:

1. Non-vectorized integer or floating-point code.
2. Integer or floating-point vectorized code, excluding Intel® Deep Learning Boost (Intel® DL Boost) code.
3. Intel DL Boost code.
4. Pause (spin-wait) dominated code.

The dynamic code need not be 100% of the class definition. It should be large enough to be considered belonging to that class. Also, dynamic microarchitectural metrics such as consumed memory bandwidth or cache bandwidth may move software threads between classes. Example pseudo-code sequences for the Intel Thread Director classes available on processors based on Alder Lake performance hybrid architecture are provided in [Examples 2-1](#) through [2-4](#).

Intel Thread Director also provides a table in system memory, only accessible to the operating system, that defines the P-core vs. E-core performance ratio per class. This allows the operating system to pick and choose the correct software thread for the correct logical processor.

In addition to the performance ratio between P-cores and E-cores, Intel Thread Director provides the energy efficiency ratio between those cores. The operating system can use this information when it prefers energy savings over maximum performance. For example, a background task such as indexing can be scheduled on the most energy-efficient core since its performance is less critical.

Example 2-1. Class 0 Pseudo-code Snippet

```
while (1)
{
    asm("xor rax, rax;"
        "add rax, 5;"
        "inc rax;"
    );
}
```

Example 2-2. Class 1 Pseudo-code Snippet

```

while (1)
{
    asm("vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"

        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
    );
}

```

Example 2-3. Class 2 Pseudo-code Snippet

```

while (1)
{
    __asm(
        vpdpbud ymm2, ymm0, ymm1
        vpdpbud ymm3, ymm0, ymm1
        vpdpbud ymm4, ymm0, ymm1
        vpdpbud ymm5, ymm0, ymm1
        vpdpbud ymm6, ymm0, ymm1
        vpdpbud ymm7, ymm0, ymm1
        vpdpbud ymm8, ymm0, ymm1
        vpdpbud ymm9, ymm0, ymm1
        vpdpbud ymm10, ymm0, ymm1
        vpdpbud ymm11, ymm0, ymm1
        vpdpbud ymm12, ymm0, ymm1
        vpdpbud ymm13, ymm0, ymm1
    );
}

```

Example 2-4. Class 3 Pseudo-code Snippet

```

while (1)
{
    asm("PAUSE;")
    asm("PAUSE;")
    asm("PAUSE;")
    asm("PAUSE;")
    asm("PAUSE;")
    asm("PAUSE;")
    asm("PAUSE;")
    asm("PAUSE;")
    asm("PAUSE;")
    asm("PAUSE;")
};
}

```

2.3.2.2 Scheduling with Intel® Hyper-Threading Technology-Enabled on Processors Supporting x86 Hybrid Architecture

E-cores are designed to provide better performance than a logical P-core with both hardware sibling hyper-threads busy.

2.3.2.3 Scheduling with a Multi-E-Core Module

E-cores within an idle module help improve performance than E-cores in a busy module.

2.3.2.4 Scheduling Background Threads on x86 Hybrid Architecture

In most scenarios, background threads can leverage the scalability and multithread efficiency of E-cores.

2.3.3 RECOMMENDATIONS FOR APPLICATION DEVELOPERS

The following are recommendations when using processors supporting performance hybrid architecture:

- Stay up to date on updates on operating systems and optimized libraries.
- Software must avoid setting hard affinities on either threads or processes to allow the operating system to provide the optimal core selection for Intel Hybrid.
- Software should replace active spin-waits with lightweight waits, ideally using the new UMWAIT/TPAUSE and older PAUSE instructions. This will allow the scheduler to get better hints on time spinning.
- Software can utilize the Windows Power Throttling information using process and thread information APIs to give the scheduler hints on the Quality of Service (QoS) required for a particular thread or process, improving performance and energy efficiency.
- Leverage Windows frameworks and media APIs for multimedia application development. Windows Media Foundation framework is optimized for hybrid architecture, enabling media applications to run efficiently while preventing glitches.
- The Windows IrqPolicyMachineDefault policy enables Windows to target interrupts to the right core optimally, and more so on hybrid architecture.

For additional recommendations and information on performance hybrid architecture, refer to the white papers on the [Performance Hybrid Architecture page](#).

2.4 THE GOLDEN COVE MICROARCHITECTURE

The Golden Cove microarchitecture is the successor of the Ice Lake microarchitecture. The Golden Cove microarchitecture introduces the following enhancements:

- Wider machine: 5→6 wide allocation, 10→12 execution ports, and 4→8 wide retirement.
- Significant increases in the size of key structures enable deeper OOO execution and expose more instruction-level parallelism.
- Greater capabilities per execution port, e.g., 5th integer ALU execution ports with expanded capability and a new fast floating-point adder.
- Intel® Advanced Matrix Extensions (Intel® AMX)¹: Built-in integrated Tiled Matrix Multiplication / Machine Learning Accelerator.
- Improved branch prediction.
- Improvements for large code footprint workloads, for example, larger branch prediction structures, enhanced code prefetcher, and larger instruction TLB.
- Wider fetch: legacy decode pipeline fetch bandwidth increase to 32B/cycles, 4→6 decoders, increased micro-op cache size, and increased micro-op cache bandwidth.

1. Intel AMX are unavailable on client parts.

- Maximum load bandwidth increased from two loads/cycle to three loads/cycle.
- Larger 4K Pages DTLB, increase of outstanding Page Miss handlers.
- Increase of outstanding misses (16 FB, 32→48 Deeper MLC miss queues).
- Enhanced data prefetchers for increased memory parallelism.
- Mid-level cache size increased to 2MB on server parts; remains 1.25MB on client parts.

2.4.1 GOLDEN COVE MICROARCHITECTURE OVERVIEW

The basic pipeline functionality of the Golden Cove microarchitecture is depicted in [Figure 2-2](#).

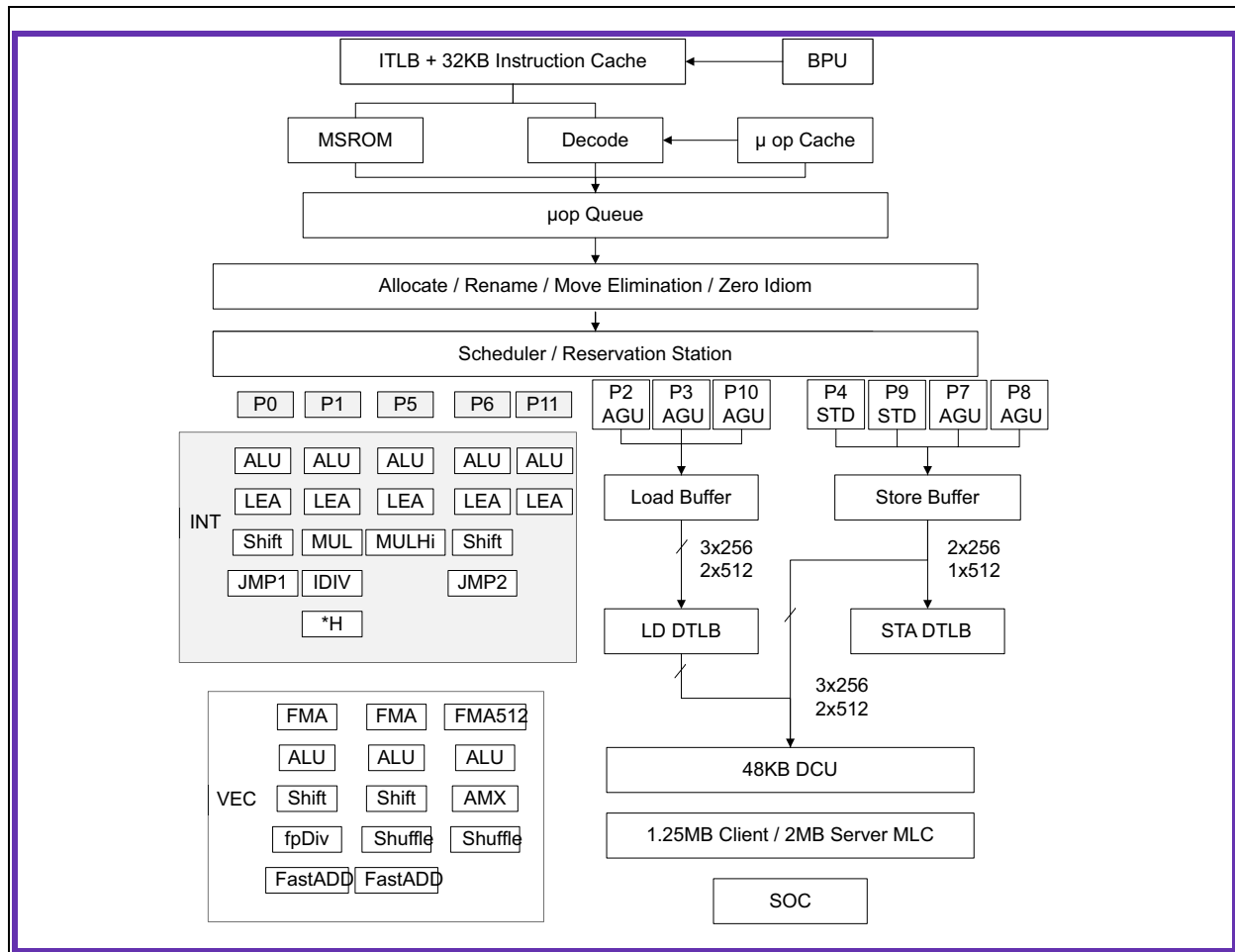


Figure 2-2. Processor Core Pipeline Functionality of the Golden Cove Microarchitecture

The Golden Cove front end is depicted in [Figure 2-3](#). The front end is built to feed the wider and deeper out-of-order core:

- Legacy decode pipeline fetch bandwidth increased from 16 to 32 bytes/cycle.
- The number of decoders increased from four to six, allowing the decode of up to six instructions per cycle.
- The micro-op cache size increased, and its bandwidth increased to deliver up to eight micro-ops per cycle.
- Improved branch prediction.

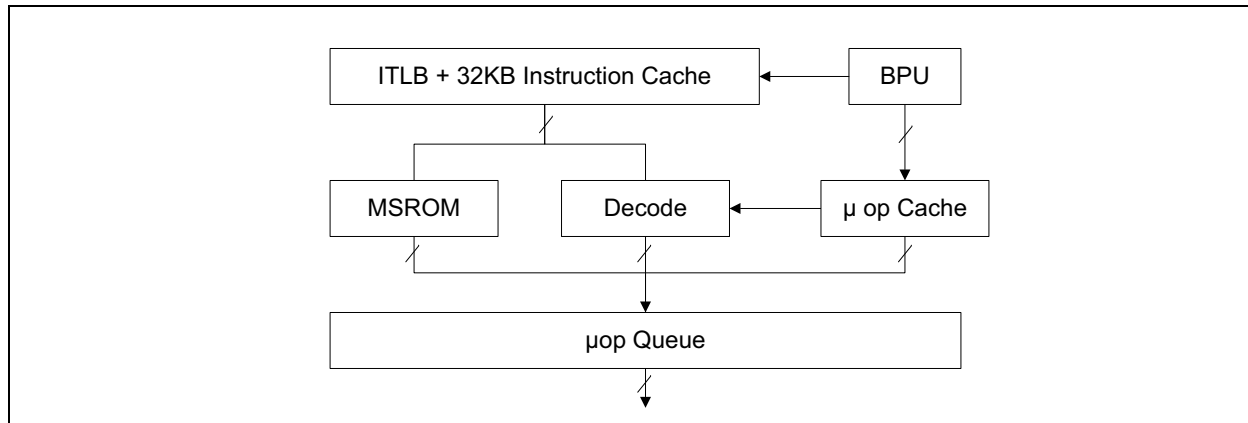


Figure 2-3. Processor Front End of the Golden Cove Microarchitecture

Improvements for large code footprint workloads:

- Double the size of the instruction TLB: 128→256 entries for 4K pages, 16→32 entries for 2M/4M pages.
- Bigger branch prediction structures.
- Enhanced code prefetcher.
- Improved LSD coverage.
- The IDQ can hold 144 μ ops per logical processor in single-thread mode, or 72 μ ops per thread when SMT is active.

Additional improvements include:

- The significant increase in the size of key buffer structures to enable deeper OOO execution and expose more instruction-level parallelism.
- Wider machine:
 - Wider allocation (5→6 μ ops per cycle) and retirement (4→8 μ ops per cycle) width.
 - Increased execution ports (10→12).
 - Greater capabilities per execution port.

[Table 2-3](#) summarizes the OOO engine's capability to dispatch different types of operations to ports.

Table 2-3. Dispatch Port and Execution Stacks of the Golden Cove Microarchitecture

Port 0	Port 1 ¹	Port 2	Port 3	Port 4	Port 5 ²	Port 6	Ports 7, 8	Port 9	Port 10	Port 11
INT ALU LEA INT Shift Jump1	INT ALU ³ LEA INT Mul INT Div	Load	Load	Store Data	INT ALU LEA INT MUL Hi	INT ALU LEA INT Shift Jump2	Store Address	Store Data	Load	INT ALU LEA

Table 2-3. Dispatch Port and Execution Stacks of the Golden Cove Microarchitecture (Contd.)

Port 0	Port 1 ¹	Port 2	Port 3	Port 4	Port 5 ²	Port 6	Ports 7, 8	Port 9	Port 10	Port 11
FMA Vec ALU Vec Shift FP Div	FMA* Fast Adder* Vec ALU* Vec Shift* Shuffle *				FMA** Fast Adder Vec ALU Shuffle					

NOTES:

1. “*” in this table indicates that these features are unavailable for 512-bit vectors.
2. “**” in this table indicates that these features are unavailable for 512-bit vectors in Client parts.
3. The Golden Cove microarchitecture implemented performance improvements requiring constraint of the micro-ops which use *H partial registers (i.e. AH, BH, CH, DH). See [Section 3.5.2.3](#) for more details.

[Table 2-4](#) lists execution units and common representative instructions that rely on these units.

Throughput improvements across the Intel® SSE, Intel AVX, and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-4. Golden Cove Microarchitecture Execution Units and Representative Instructions¹

Execution Unit	# of Unit	Instructions
ALU	5 ²	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2 ³	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blmsk, bzhi, etc.
Vec ALU	2x256-bit 1x512-bit	(v)add, (v)cmp, (v)max, (v)min, (v)sub, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2sl, (v)cvtss2sl
	3x256-bit 2x512-bit	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2x256-bit 1x512-bit	(v)psllv*, (v)psrlv*, vector shift count in imm8
VEC Add (in VEC FMA)	2x256-bit 1x512-bit	(v)add*, (v)cmp*, (v)max*, (v)min*, (v)sub*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpsq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si
VEC Fast Add	2x256-bit 1x512-bit	(v)add*, (v)addsub*, (v)sub*

Table 2-4. Golden Cove Microarchitecture Execution Units and Representative Instructions¹

Execution Unit	# of Unit	Instructions
Shuffle	2x256-bit 1x512-bit	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw (new cross lane shuffle on both ports)
Vec Mul/FMA	2x256-bit (1 or 2)x512-bit	(v)mul*, (v)pmul*, (v)pmadd*
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm
FP Mov	1	(v)movsd/ss, (v)movd gpr
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See [Section 15.16.5](#) on MMX instruction throughput remedy.
2. The Golden Cove microarchitecture implemented performance improvements requiring constraint of the micro-ops which use *H partial registers (i.e. AH, BH, CH, DH). See [Section 3.5.2.3](#) for more details.
3. Ibid.

[Table 2-5](#) describes bypass delay in cycles between producer and consumer operations.

Table 2-5. Bypass Delay Between Producer and Consumer Micro-Ops

FROM [EU/Port/Latency]	TO [EU/PORT/Latency]						
	SIMD/0,1/1	FMA/0,1/4	MUL/0,1/4	Fast Adder/1,5/3	SIMD/5/1,3	SHUF/1,5/1,3	V2I/0/3
SIMD/0,1/1	0	1	1	1	0	0	0
FMA/0,1/4	1	0	1	0	0	0	0
MUL/0,1/4	1	0	1	0	0	0	0
Fast Adder/0,1/3	1	0	1	-1	0	0	0
SIMD/5/1,3	0	1	1	1	0	0	0
SHUF/1,5/1,3	0	0	1	0	0	0	0
V2I/0/3	0	0	1	0	0	0	0
I2V/5/1	0	1	1	0	0	0	0

The attributes relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the μ op. For example:

- “SIMD/0,1/1” applies to a 1-cycle vector SIMD μ op dispatched to either port 0 or port 1.
- “SIMD/5/1,3” applies to either a 1-cycle or 3-cycle non-shuffle μ op dispatched to port 5.
- “V2I/0/3” applies to a three-cycle vector-to-integer μ op dispatched to port 0.

- “I2V/5/1” applies to a one-cycle integer-to-vector μ op dispatched to port 5.
- “Fast Adder/1,5/3” applies to either a three-cycle 256-bit μ op dispatched to port 1 or port 5, or a 512-bit μ op dispatched to port 5. This operation supports two cycles back-to-back between a pair of Fast Adder operations.

A new Fast Adder¹ unit is added as 512-bit on port 5 in the VEC stack and as 256-bit on ports 1 and 5. The Fast Adder performs floating-point ADD/SUB operations in three cycles.

Back-to-back ADD/SUB operations, both executed on the Fast Adder unit, perform the operations in two cycles.

- In 128/256-bit, back-to-back ADD/SUB operations executed on the Fast Adder unit perform the operations in two cycles.
- In 512-bit, back-to-back ADD/SUB operations are executed in two cycles if both operations use the Fast Adder unit on port 5.

The following instructions are executed by the Fast Adder unit:

- (V)ADDSS/SD/PS/PD
- (V)ADDSS/SD/PS/PD
- (V)SUBSS/SD/PS/PD

2.4.1.1 Cache Subsystem and Memory Subsystem

Changes in the cache subsystem and memory subsystem within the Golden Cove microarchitecture include:

- Maximum load bandwidth increased from two to three loads per cycle. Bandwidth of Intel AVX-512 loads, Intel AMX loads, and MMX/x87 loads remain at a maximum of two loads per cycle.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Number of entries for 4K pages in the load DTLB increased from 64 to 96.
- The Page Miss handler can handle up to four D-side page walks in parallel instead of two.
- Increased outstanding DCU and MLC misses.
- Enhanced data prefetchers for increased memory parallelism.
- Partial store forwarding allowing forwarding data from store to load also when only part of the load was covered by the store (in case the load's offset matches the store's offset).

2.4.1.2 Avoiding Destination False Dependency

Some SIMD instructions incur false dependencies on the destination operand. The following instructions are affected:

- VFMULSH, VFMULCPH
- VFCMULSH, VFCMULCPH
- VPERMD, VPERMQ, VPERMPS, VPERMPD
- VRANGE[SS,PS,SD,PD]
- VGETMANTSH, VGETMANTSS, VGETMANTSD
- VGETMANTPS, VGETMANTPD (memory versions only)
- VPMULLQ

1. The Fast Adder unit is unavailable on 512-bit vectors in Client parts.

Recommendation: Use dependency breaking zero idioms on the destination register before the affected instructions to avoid potential slowdown from the false dependency.

Example 2-5. Breaking False Dependency through Zero Idiom

Code with False Dependency Impact	Mitigation: Break False Dependency with Zero Idiom
<pre>vaddps zmm3, zmm4, zmm5 vmovaps [rsi], zmm3 vfmulcph zmm3, zmm2, zmm1 ;False dependency on zmm3. //Will not execute out-of-order until vaddps writes zmm3.</pre>	<pre>vaddps zmm3, zmm4, zmm5 vmovaps [rsi], zmm3 vpxord zmm3, zmm3, zmm3 //Dependency-breaking zero idiom. vfmulcph zmm3, zmm2, zmm1 //Execute out-of-order without waiting for vaddps result.</pre>

2.5 ICE LAKE CLIENT MICROARCHITECTURE

The Ice Lake client microarchitecture introduces the following new features that allow optimizations of applications for performance and power consumption:

- Targeted vector acceleration.
- Crypto acceleration.
- Intel® Software Guard Extensions (Intel® SGX) enhancements.
- Cache line writeback instruction (CLWB).

2.5.1 ICE LAKE CLIENT MICROARCHITECTURE OVERVIEW

The Ice Lake client microarchitecture builds on the successes of the Skylake client microarchitecture. The basic pipeline functionality of the Ice Lake Client microarchitecture is depicted in [Figure 2-4](#).

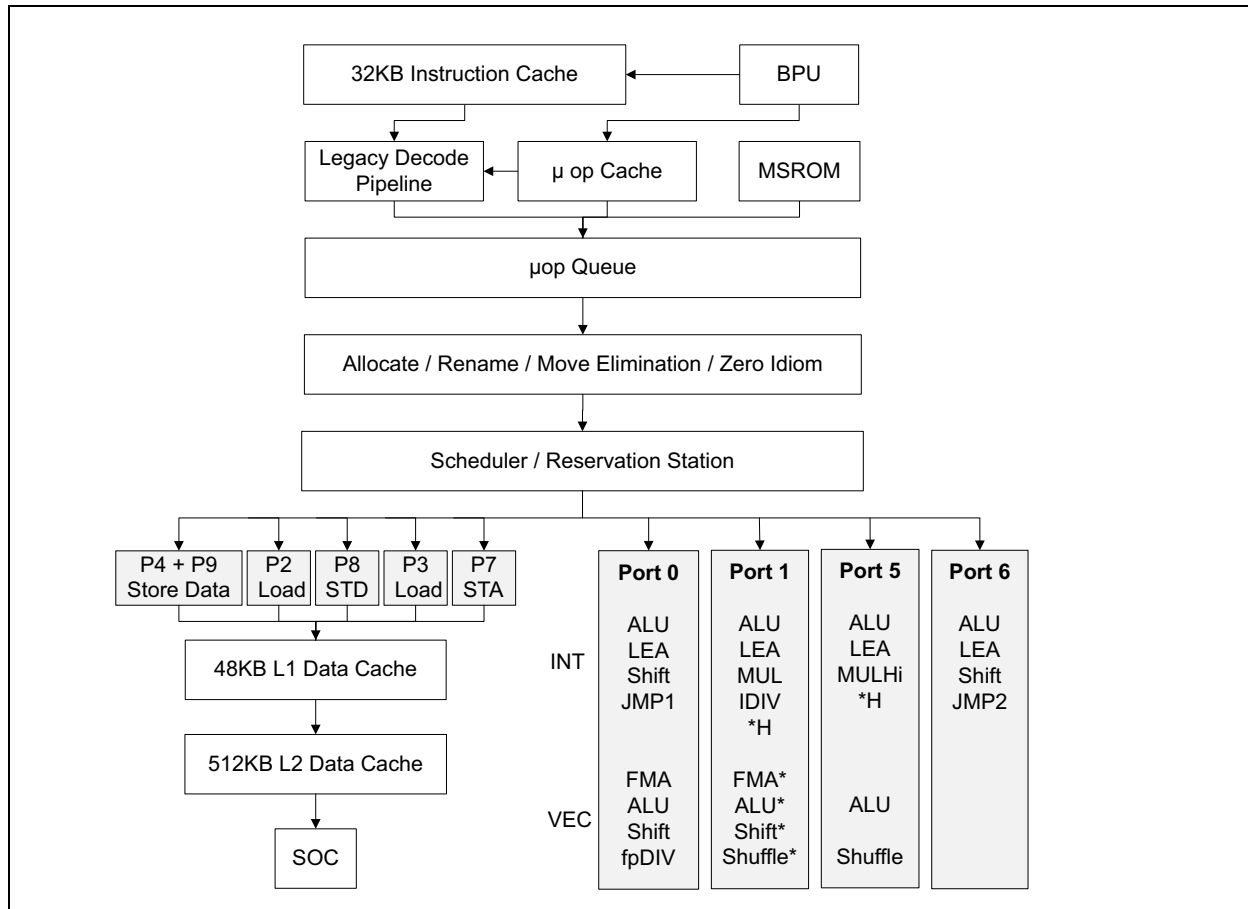


Figure 2-4. Processor Core Pipeline Functionality of the Ice Lake Client Microarchitecture¹

NOTES:

1. "*" in the figure above indicates these features are unavailable for 512-bit vectors.
2. "INT" represents GPR scalar instructions.
3. "VEC" represents floating-point and integer vector instructions.
4. "MULHi" produces the upper 64 bits of the result of an iMul operation that multiplies two 64-bit registers and places the result into two 64-bits registers.
5. The "Shuffle" on port 1 is new, and supports only in-lane shuffles that operate within the same 128-bit sub-vector.
6. The "IDIV" unit on port 1 is new, and performs integer divide operations at a reduced latency.
7. The Golden Cove microarchitecture implemented performance improvements requiring constraint of the micro-ops which use *H partial registers (i.e. AH, BH, CH, DH). See [Section 3.5.2.3](#) for more details.

The Ice Lake client microarchitecture introduced the following features:

- Significant increase in size of key structures enable deeper OOO execution.
- Wider machine: 4 → 5 wide allocation, 8 → 10 execution ports.
- Intel AVX-512 (new for client processors): 512-bit vector operations, 512-bit loads and stores to memory, and 32 new 512-bit registers.
- Greater capabilities per execution port (e.g., SIMD shuffle, LEA), reduced latency Integer Divider.
- 2×BW for AES-NI peak throughput for existing binaries (microarchitectural).

- Rep move string acceleration.
- 50% increase in size of the L1 data cache.
- Reduced effective load latency.
- 2×L1 store bandwidth: 1 → 2 stores per cycle.
- Enhanced data prefetchers for increased memory parallelism.
- Larger 2nd level TLB.
- Larger μ op cache.
- Improved branch predictor.
- Large page ITLB size in single thread mode doubled.
- Larger L2 cache.

The Ice Lake client microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of some components including a ring interconnect to multiple slices of L3, processor graphics, integrated memory controller, interconnect fabrics, and more.

2.5.1.1 The Front End

The front end changes in Ice Lake Client microarchitecture included:

- Improved branch predictor.
- Large page ITLB in single thread mode increased from 8 to 16 entries.
- Larger μ op cache.
- The IDQ can hold 70 μ ops per logical processor vs. 64 μ ops per logical processor in previous generations when two sibling logical processors in the same core are active (2×70 vs. 2×64 per core). If only one logical processor is active in the core, the IDQ can hold 70 μ ops vs. 64 μ ops.
- The LSD in the IDQ can detect loops of up to 70 μ ops per logical processor irrespective single thread or multi thread operation.

2.5.1.2 The Out of Order and Execution Engines

The Out of Order and execution engines changes in Ice Lake client microarchitecture include:

- A significant increase in size of reorder buffer, load buffer, store buffer, and reservation stations enable deeper OOO execution and higher cache bandwidth.
- Wider machine: 4 → 5 wide allocation, 8 → 10 execution ports.
- Greater capabilities per execution port (e.g., SIMD shuffle, LEA).
- Reduced latency Integer Divider.
- A new iDIV unit was added, significantly reducing the latency and improving the throughput of integer divide operations.

Table 2-6 summarizes the OOO engine's capability to dispatch different types of operations to ports.

Table 2-6. Dispatch Port and Execution Stacks of the Ice Lake Client Microarchitecture

Port 0	Port 1 ¹	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7	Port 8	Port 9
INT ALU LEA INT Shift Jump1	INT ALU LEA INT Mul INT Div	Load	Load	Store Data	INT ALU LEA INT MUL Hi	INT ALU LEA INT Shift Jump2	Store Address	Store Address	Store Data
FMA Vec ALU Vec Shift FP Div	FMA* Vec ALU* Vec Shift* Vec Shuffle*				Vec ALU Vec Shuffle				

NOTES:

1. “*” in this table indicates these features are unavailable for 512-bit vectors.

Table 2-7 lists execution units and common representative instructions that rely on these units.

Throughput improvements across the Intel SSE, Intel AVX, and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-7. Ice Lake Client Microarchitecture Execution Units and Representative Instructions¹

Execution Unit	# of Unit	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blmsk, bzhi, etc.
Vec ALU	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2	(v)psllv*, (v)psrlv*, vector shift count in imm8
Vec Add	2	(v)addp*, (v)cmpp*, (v)max*, (v)min*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si
Shuffle	2	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw
Vec Mul	2	(v)mul*, (v)pmul*, (v)pmadd*

Table 2-7. Ice Lake Client Microarchitecture Execution Units and Representative Instructions¹

Execution Unit	# of Unit	Instructions
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm
FP Mov	1	(v)movsd/ss, (v)movd gpr
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See [Section 15.16.5](#) on MMX instruction throughput remedy.

[Table 2-8](#) describes bypass delay in cycles between producer and consumer operations.

Table 2-8. Bypass Delay Between Producer and Consumer Micro-ops

FROM [EU/Port/Latency]	TO [EU/PORT/Latency]						
	SIMD/0,1/1	FMA/0,1/4	VIMUL/0,1/4	SIMD/5,1,3	SHUF/5,1,3	V2I/0/3	I2V/5/1
SIMD/0,1/1	0	1	1	0	0	0	NA
FMA/0,1/4	1	0	1	0	0	0	NA
VIMUL/0,1/4	1	0	1	0	0	0	NA
SIMD/5,1,3	0	1	1	0	0	0	NA
SHUF/5,1,3	0	0	1	0	0	0	NA
V2I/0/3	0	0	1	0	0	0	NA
I2V/5/1	0	1	1	0	0	0	NA

The attributes that are relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the μ op. For example:

- “SIMD/0,1/1” applies to 1-cycle vector SIMD μ op dispatched to either port 0 or port 1.
- “SIMD/5,1,3” applies to either a 1-cycle or 3-cycle non-shuffle μ op dispatched to port 5.
- “V2I/0/3” applies to a 3-cycle vector-to-integer μ op dispatched to port 0.
- “I2V/5/1” applies to a 1-cycle integer-to-vector μ op to port 5.

2.5.1.3 Cache and Memory Subsystem

The cache hierarchy changes in Ice Lake Client microarchitecture include:

- 50% increase in size of the L1 data cache.
- 2×L1 store bandwidth: 3 → 4 AGUs, 1 → 2 store data.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Higher cache bandwidth compared to previous generations.
- Larger 2nd level TLB: 1.5K entries → 2K entries.

- Enhanced data prefetchers for increased memory parallelism.
- L2 cache size increased from 256KB to 512KB.
- L2 cache associativity increased from 4 ways to 8 ways.
- Significant reduction in effective load latency.

Table 2-9. Cache Parameters of the Ice Lake Client Microarchitecture

Level	Capacity / Associativity	Line Size (bytes)	Latency ¹ (cycles)	Peak Bandwidth (bytes/cycles)	Sustained Bandwidth (bytes/cycles)	Update Policy
First Level (DCU)	48KB/8	64	5	2×64B loads + 1×64B or 2×32B stores	Same as peak	Writeback
Second Level (MLC)	512KB/8	64	13	64	48	Writeback
Third Level (LLC)	Up to 2MB per core/up to 16 ways	64	xx^2	32	21	Writeback

NOTES:

1. Software-visible latency/bandwidth will vary depending on access patterns and other factors.
2. This number depends on core count.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, shared L2 TLB for 4K and 4MB pages and a dedicated L2 TLB for 1GB pages.

Table 2-10. TLB Parameters of the Ice Lake Client Microarchitecture

Level	Page Size	Entries ST	Per-thread Entries MT Latency	Associativity
Instruction	4KB	128	64	8
Instruction	2MB/4MB	16	8	8
First Level Data (loads)	4KB	64	64 competitively shared	4
First Level Data (loads)	2MB/4MB	32	32 competitively shared	4
First Level Data (loads)	1GB	8	8 competitively shared	8
First Level Data (stores)	Shared for all page sizes	16	16 competitively shared	16

Table 2-10. TLB Parameters of the Ice Lake Client Microarchitecture (Contd.)

Level	Page Size	Entries ST	Per-thread Entries MT Latency	Associativity
Second Level	Shared for all page sizes	2048 ¹	2048 competitively shared	16

NOTES:

1. 4K pages can use all 2048 entries. 2/4MB pages can use 1024 entries (in 8 ways), sharing them with 4K pages. 1GB pages can use the other 1024 entries (in 8 ways), also sharing them with 4K pages.

Paired Stores

Ice Lake Client microarchitecture includes two store pipelines in the core, with the following features:

- Two dedicated AGU for LDs on ports 2 and 3.
- Two dedicated AGU for STAs on ports 7 and 8.
- Two fully featured STA pipelines.
- Two 256-bit wide STD pipelines (Intel AVX-512 store data takes two cycles to write).
- Second senior store pipeline to the DCU via store merging.

Ice Lake Client microarchitecture can write two senior stores to the cache in a single cycle if these two stores can be paired together. That is:

- The stores must be to the same cache line.
- Both stores are of the same memory type, WB or USWC.
- None of the stores cross cache line or page boundary.

To maximize performance from the second store port try to:

- Align store operations whenever possible.
- Place consecutive stores in the same cache line (not necessarily as adjacent instructions).

As seen in [Example 2-6](#), it is important to take into consideration all stores, explicit or not.

Example 2-6. Considering Stores

Stores are Paired Across Loop Iterations	Stores Not Paired Due to Stack Update in Between
Loop: <pre> compute reg ... store [X], reg add X, 4 jmp Loop ; stores from different iterations of the loop can be paired all together because they usually would be same line </pre>	Loop: <pre> call function to compute reg ... store [X], reg add X, 4 jmp Loop ; stores from different iterations of the loop cannot be paired anymore because of the call store to stack ; the call is disturbing pairing </pre>

In some cases it is possible to rearrange the code to achieve store pairing. [Example 2-7](#) provides details.

Example 2-7. Rearranging Code to Achieve Store Pairing

Stores to Different Cache Lines - Not Paired	Unrolling May Solve the Problem
<pre> Loop: ... compute ymm1 ... vmovaps [x], ymm1 ... compute ymm2 ... vmovaps [y], ymm2 add x, 32 add y, 32 jmp Loop //this loop cannot pair any store because of alternating store to different cache lines [x] and [y] </pre>	<pre> Loop: ... compute ymm1 ... vmovaps [x], ymm1 ... compute new ymm1 ... vmovaps [x+32], ymm1 ... compute ymm2 ... vmovaps [y], ymm2 ... compute new ymm2 ... vmovaps [y+32], ymm2 add x, 64 add y, 64 jmp Loop //the loop was unrolled 2 times and stores rearranged to make sure two stores to the same cache line are placed one after another. Now stores to addresses [x] and [x+32] are to the same cache line and could be paired together and executed in same cycle. </pre>

2.5.1.4 Fast Store Forwarding Prediction (FSFP)

This section includes recommendations for effective use of Fast Store Forwarding Prediction (FSFP) introduced in Ice Lake microarchitecture. Extrapolated from previous behavior, FSFP enables the processor to predict that a store will forward data to a younger load and optimize that case. The optimization allows the load to complete using the data of predicted store but without accessing the memory. Only integer loads support FSFP in the Ice Lake microarchitecture.

The Fast Store Forwarding Prediction has limitations. to maximize performance gain on Ice Lake microarchitecture it is recommended to follow these recommendations:

- Only loads and stores without Index (encoded with no SIB byte) are supported. LEA operation can be used to avoid Index register usage during memory address computations.
- Loads and stores using RIP-relative addressing do not support FSFP. We recommend using the LEA operation to pre-compute address to enable FSFP for such cases.
- Loads and stores operating with 16-bit General Purpose Registers (AX/BX/CX/DX and etc) or *H 8-bit registers do not support FSFP optimization. We recommend using **movzx** instruction instead of unsupported registers.

Example 2-8. FSFP Optimization

Slow Version Not Enabling PFSP	Enabling FSFP Using LEA Operation
<pre> Loop: mov r10,[rsi+r8*8] inc qword[r10+r10*8] mov r11,[rsi+r8*8] inc r8 inc qword[r10+r11*8] ... jmp Loop </pre>	<pre> Loop: mov r10,[rsi+r8*8] lea r12,[r10+r10*8] ; using LEA to avoid ;Index register for ;inc below inc qword[r12] mov r11,[rsi+r8*8] inc r8 lea r13,[r10+r11*8] ; another similar case inc qword [r13] jmp Loop </pre>

2.5.1.5 New Instructions

New instructions and architectural changes in Ice Lake Client microarchitecture are listed below. Actual support may be product dependent.

- Crypto acceleration
 - SHA NI for acceleration of SHA1 and SHA256 hash algorithms.
 - Big-Number Arithmetic (IFMA): VPMADD52 - two new instructions for big number multiplication for acceleration of RSA vectorized SW and other Crypto algorithms (Public key) performance.
 - Galois Field New Instructions (GFNI) for acceleration of various encryption algorithms, error correction algorithms, and bit matrix multiplications.
 - Vector AES and Vector Carry-less Multiply (PCLMULQDQ) instructions to accelerate AES and AES-GCM.
- Security Technologies
 - Intel® SGX enhancements to improve usability and applicability: EDMM, multi-package server support, support for VMM memory oversubscription, performance, larger secure memory.
- Sub Page protection for better performance of security VMMs.
- Targeted Acceleration
 - Vector Bit Manipulation Instructions: VBMI1 (permutes, shifts) and VBMI2 (Expand, Compress, Shifts)- used for columnar database access, dictionary based decompression, discrete mathematics, and data-mining routines (bit permutation and bit-matrix-multiplication).
 - VNNI with support for integer 8 and 16 bits data types- CNN/ML/DL acceleration.
 - Bit Algebra (POPCNT, Bit Shuffle).
 - Cache line writeback instruction (CLWB) enables fast cache-line update to memory, while retaining clean copy in cache.
- Platform analysis features for more efficient performance software tuning and debug.
 - AnyThread removal.
 - 2x general counters (up to 8 per-thread).
 - Fixed Counter 3 for issue slots.

- New performance metrics for built-in support for Level 1 Top-Down method (% of Issue slots that are front-end bound, back-end bound, bad speculation, retiring) while leaving the 8 general purpose counters free for software use.

2.5.1.6 Ice Lake Client Microarchitecture Power Management

Processors based on Ice Lake client microarchitecture are the first client processors whose cores may execute at a different frequency from one another. The frequency is selected based on the specific instruction mix; the type, width and number of vector instructions of the program that executes on each core, the ratio between active time and idle time of each core, and other considerations such as how many cores share similar characteristics.

Most of the power management features of Skylake Server Microarchitecture (see [Section 2.6](#)) is applicable to Ice Lake Client microarchitecture as well. The main differences are the following:

- The typical P0n max frequency difference between Intel® Advanced Vector Extensions (Intel® AVX-512) and Intel® Advanced Vector Extensions 2 (Intel® AVX2) on Ice Lake Client microarchitecture is much lower than on Skylake Server microarchitecture. Therefore, the negative impact on overall application performance is much smaller.
- All processors based on Ice Lake Client microarchitecture contain a single 512-bit FMA unit, whereas some of the processors based on Skylake Server microarchitecture contain two such units. Both processors contain two 256-bit FMA units. The power consumed by Ice Lake Client FMA units is the same, whereas on Skylake Server the 512-bit units consume twice as much.

Compute heavy workloads, especially those that span multiple Ice Lake client cores, execute at a lower frequency than P0n, under Intel AVX-512 and under Intel AVX2 instruction sets, due to power limitations. In this scenario, Intel AVX-512 architecture, which requires less dynamic instructions to complete the same task than Intel AVX2 architecture, consumes less power and thus may achieve higher frequency. The net result may be higher performance due to the shorter path length and a bit higher frequency.

There are still some cases where coding to the Intel AVX-512 instruction set yields lower performance than when coding to the Intel AVX2 instruction set. Sometimes it is due to microarchitecture artifacts of longer vectors, in other cases the natural vectors are just not long enough. Most compilers are still maturing their Intel AVX-512 support, and it may take them a few more years to generate optimal code.

The general recommendation in the Skylake Server Power Management section (see [Section 2.6.3](#)) still holds. Developers should code to the Intel AVX-512 instruction set and compare the performance to their Intel AVX2 workload on Ice Lake client microarchitecture, before making the decision to proceed with a complete port.

2.6 SKYLAKE SERVER MICROARCHITECTURE

The Intel® Xeon® Processor scalable processors based on the Skylake microarchitecture can be identified using CPUID's DisplayFamily_DisplayModel signature, which can be found in Table 2-1 of [CHAPTER 2 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4](#).

The Skylake Server microarchitecture introduces the following new features¹ that allow you to optimize your application for performance and power consumption.

- A new core based on the Skylake Server microarchitecture with process improvements based on the Kaby Lake microarchitecture.
- Intel AVX-512 support.
- More cores per socket (max 28 vs. max 22).
- 6 memory channels per socket in Skylake microarchitecture vs. 4 in the Broadwell microarchitecture.

1. Some features may not be available on all products.

- Bigger L2 cache, smaller non inclusive L3 cache.
- Intel® Optane™ support.
- Intel® Omni-Path Architecture (Intel® OPA).
- Sub-NUMA Clustering (SNC) support.

The gray rectangles in [Figure 2-5](#) represent features different in Skylake Server microarchitecture compared to Skylake microarchitecture for client;

- A 1MB L2 cache.
- An additional Intel AVX-512 FMA unit on port 5 which is available on some parts.

In this figure:

- INTEGER represents GPR scalar instructions.
- VEC represents floating point and integer vector instructions.
- SLOW LEA represents a lea with two registers and displacement, all other lea versions considered as FAST LEA.
- BRANCH1 is the primary branch and more capable than BRANCH2.

Since port 0 and port 1 are 256-bits wide, Intel AVX-512 operations that will be dispatched to port 0 will execute on port 0 and port 1; however, other operations such as **LEA** can still execute on port 1 in parallel. See the red block in [Figure 2-9](#) for the fusion of ports 0 and 1.

Notice that, unlike Skylake microarchitecture for client, the Skylake Server microarchitecture has its front end loop stream detector (LSD) disabled.

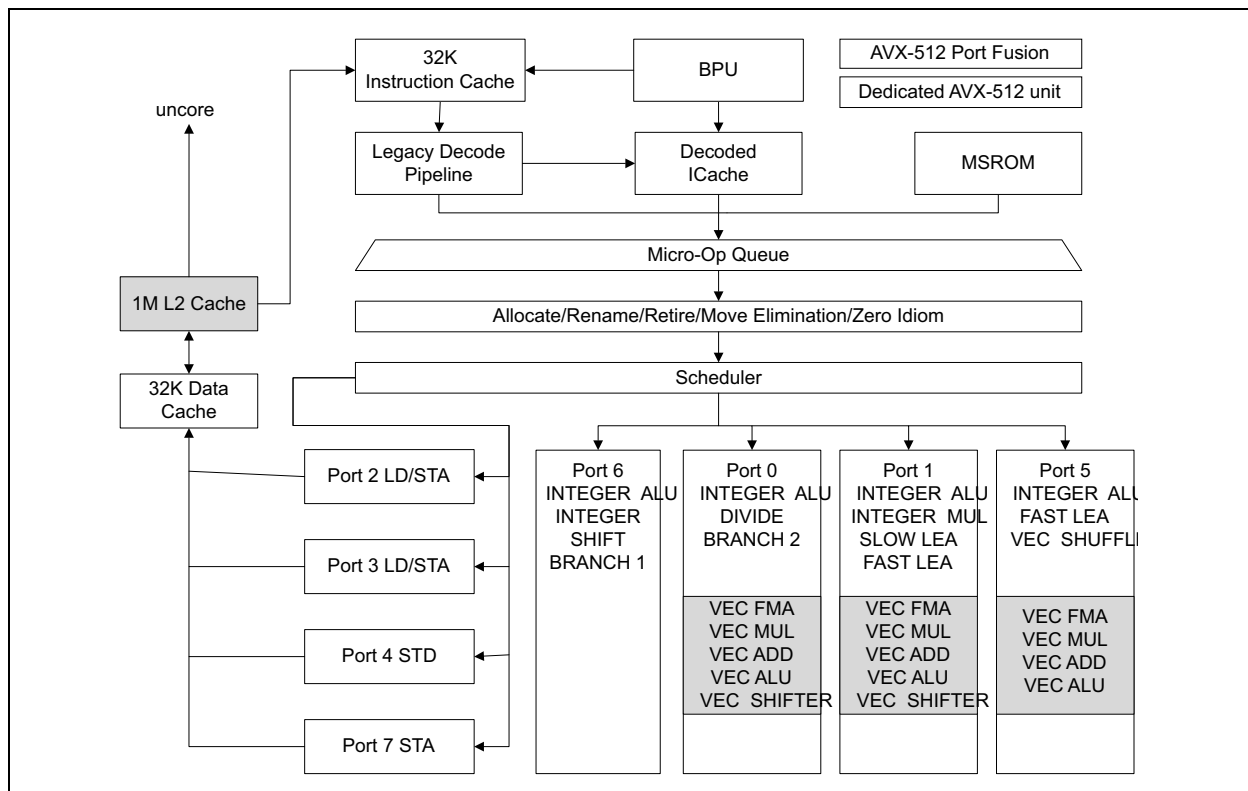


Figure 2-5. Processor Core Pipeline Functionality of the Skylake Server Microarchitecture

2.6.1 SKYLAKE SERVER MICROARCHITECTURE CACHE

Intel Xeon scalable processors based on Skylake server microarchitecture has significant changes in core and uncore architecture to improve performance and scalability of several components compared with the previous generation of the Intel Xeon processors based on the Broadwell microarchitecture.

2.6.1.1 Larger Mid-Level Cache

Skylake server microarchitecture implements a mid-level (L2) cache of 1 MB capacity with a minimum load-to-use latency of 14 cycles. The mid-level cache capacity is four times larger than the capacity in previous Intel Xeon processor family implementations. The line size of the mid-level cache is 64B and it is 16-way associative. The mid-level cache is private to each core.

Software that has been optimized to place data in mid-level cache may have to be revised to take advantage of the larger mid-level cache available in Skylake server microarchitecture.

2.6.1.2 Non-Inclusive Last Level Cache

The last level cache (LLC) in Skylake is a non-inclusive, distributed, shared cache. The size of each of the banks of last level cache has shrunk to 1.375 MBs per bank. Because of the non-inclusive nature of the last level cache, blocks that are present in the mid-level cache of one of the cores may not have a copy resident in a bank of last level cache. Based on the access pattern, size of the code and data accessed, and sharing behavior between cores for a cache block, the last level cache may appear as a victim cache of the mid-level cache and the aggregate cache capacity per core may appear to be a combination of the private mid-level cache per core and a portion of the last level cache.

2.6.1.3 Skylake Server Microarchitecture Cache Recommendations

A high-level comparison between Skylake server microarchitecture cache and the previous generation Broadwell microarchitecture cache is available in the table below.

Table 2-11. Cache Comparison Between Skylake Microarchitecture and Broadwell Microarchitecture

Cache level	Category	Broadwell Microarchitecture	Skylake Server Microarchitecture
L1 Data Cache Unit (DCU)	Size [KB]	32	32
	Latency [cycles]	4-6	4-6
	Max bandwidth [bytes/cycles]	96	192
	Sustained bandwidth [bytes/cycles]	93	133
	Associativity [ways]	8	8

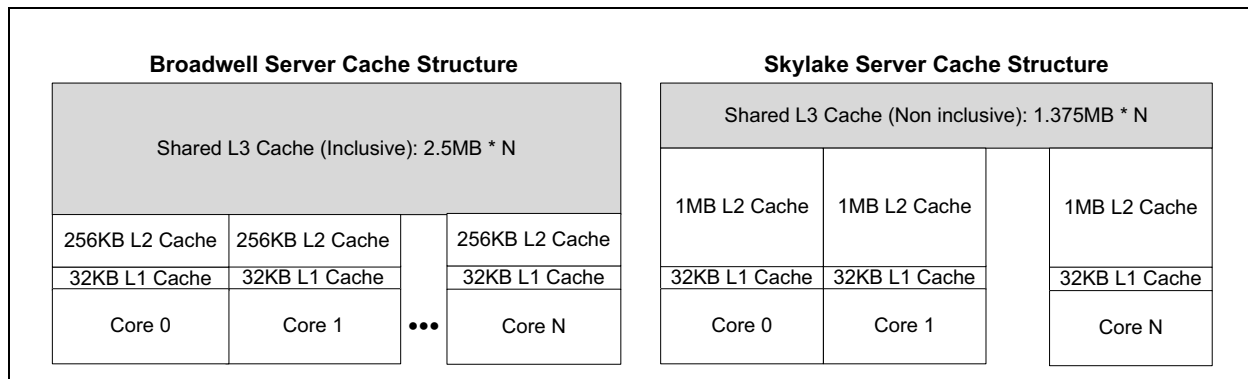
Table 2-11. Cache Comparison Between Skylake Microarchitecture and Broadwell Microarchitecture (Contd.)

Cache level	Category	Broadwell Microarchitecture	Skylake Server Microarchitecture
L2 Mid-level Cache (MLC)	Size [KB]	256	1024 (1MB)
	Latency [cycles]	12	14
	Max bandwidth [bytes/cycles]	32	64
	Sustained bandwidth [bytes/cycles]	25	52
	Associativity [ways]	8	16
L3 Last-level Cache (LLC)	Size [MB]	Up to 2.5 per core	up to 1.375 ¹ per core
	Latency [cycles]	50-60	50-70
	Max bandwidth [bytes/cycles]	16	32
	Sustained bandwidth [bytes/cycles]	14	15

NOTES:

1. Some Skylake server parts have some cores disabled and hence have more than 1.375 MBs per core of L3 cache.

The figure below shows how Skylake server microarchitecture shifts the memory balance from shared-distributed with high latency, to private-local with low latency.

**Figure 2-6. Broadwell Microarchitecture and Skylake Server Microarchitecture Cache Structures**

The potential performance benefit from the cache changes is high, but software will need to adapt its memory tiling strategy to be optimal for the new cache sizes.

Recommendation: Rebalance application shared and private data sizes to match the smaller, non-inclusive L3 cache, and larger L2 cache.

Choice of cache blocking should be based on application bandwidth requirements and changes from one application to another. Having four times the L2 cache size and twice the L2 cache bandwidth compared to the previous generation Broadwell microarchitecture enables some applications to block to L2 instead of L1 and thereby improves performance.

Recommendation: Consider blocking to L2 on Skylake Server microarchitecture if L2 can sustain the application's bandwidth requirements.

The change from inclusive last level cache to non-inclusive means that the capacity of mid-level and last level cache can now be added together. Programs that determine cache capacity per core at run time should now use a combination of mid-level cache size and last level cache size per core to estimate the effective cache size per core. Using just the last level cache size per core may result in non-optimal use of available on-chip cache; see [Section 2.6.2](#) for details.

Recommendation: In case of no data sharing, applications should consider cache capacity per core as L2 and L3 cache sizes and not only L3 cache size.

2.6.2 NON-TEMPORAL STORES ON SKYLAKE SERVER MICROARCHITECTURE

Because of the change in the size of each bank of last level cache on Skylake server microarchitecture, if an application, library, or driver only considers the last level cache to determine the size of on-chip cache-per-core, it may see a reduction with Skylake server microarchitecture and may use non-temporal store with smaller blocks of memory writes. Since non-temporal stores evict cache lines back to memory, this may result in an increase in subsequent cache misses and memory bandwidth demands on Skylake Server microarchitecture, compared to the previous Intel Xeon processor family.

Also, because of a change in the handling of accesses resulting from non-temporal stores by Skylake Server microarchitecture, the resources within each core remain busy for a longer duration compared to similar accesses on the previous Intel Xeon processor family. As a result, if a series of such instructions are executed, there is a potential that the processor may run out of resources and stall, thus limiting the memory write bandwidth from each core.

The increase in cache misses due to overuse of non-temporal stores and the limit on the memory write bandwidth per core for non-temporal stores may result in reduced performance for some applications.

To avoid the performance condition described above with Skylake server microarchitecture, include mid-level cache capacity per core in addition to the last level cache per core for applications, libraries, or drivers that determine the on-chip cache available with each core. Doing so optimizes the available on-chip cache capacity on Skylake server microarchitecture as intended, with its non-inclusive last level cache implementation.

2.6.3 SKYLAKE SERVER POWER MANAGEMENT

This section describes the interaction of Skylake Server's Power Management and its Vector ISA.

Skylake Server microarchitecture dynamically selects the frequency at which each of its cores executes. The selected frequency depends on the instruction mix; the type, width, and number of vector instructions that execute over a given period of time. The processor also takes into account the number of cores that share similar characteristics.

Intel® Xeon® processors based on Broadwell microarchitecture work similarly, but to a lesser extent since they only support 256-bit vector instructions. Skylake Server microarchitecture supports Intel® AVX-512 instructions, which can potentially draw more current and more power than Intel® AVX2 instructions.

The processor dynamically adjusts its maximum frequency to higher or lower levels as necessary, therefore a program might be limited to different maximum frequencies during its execution.

[Table 2-12](#) includes information about the maximum Intel® Turbo Boost technology core frequency for each type of instruction executed. The maximum frequency (P0n) is an array of frequencies which depend on the number of cores within the category. The more cores belonging to a category at any given time, the lower the maximum frequency.

Table 2-12. Maximum Intel® Turbo Boost Technology Core Frequency Levels

Level	Category	Frequency Level	Max Frequency (P0n)	Instruction Types
0	Intel® AVX2 light instructions	Highest	Max	Scalar, AVX128, SSE, Intel® AVX2 w/o FP or INT MUL/FMA
1	Intel® AVX2 heavy instructions + Intel® AVX-512 light instructions	Medium	Max Intel® AVX2	Intel® AVX2 FP + INT MUL/FMA, Intel® AVX-512 without FP or INT MUL/FMA
2	Intel® AVX-512 heavy instructions	Lowest	Max Intel® AVX-512	Intel® AVX-512 FP + INT MUL/FMA

For per SKU max frequency details (reference figure 1-15), refer to the [Intel® Xeon® Scalable Processor Family Technical Resources page](#).

[Figure 2-7](#) is an example for core frequency range in a given system where each core frequency is determined independently based on the demand of the workload.

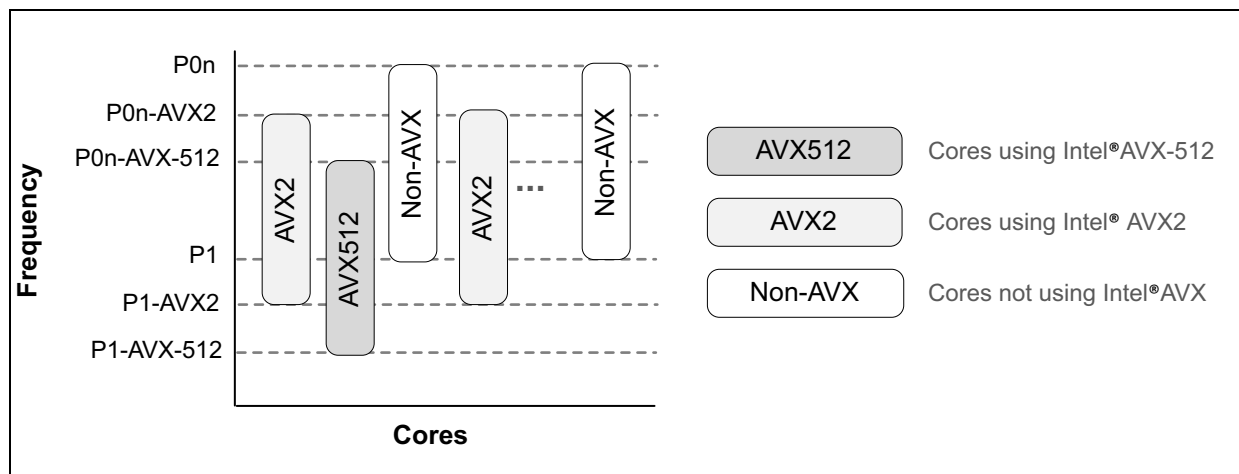


Figure 2-7. Mixed Workloads

The following performance monitoring events can be used to determine how many cycles were spent in each of the three frequency levels.

- `CORE_POWER.LVL0_TURBO_LICENSE`: Core cycles where the core was running in a manner where the maximum frequency was P0n.
- `CORE_POWER.LVL1_TURBO_LICENSE`: Core cycles where the core was running in a manner where the maximum frequency was P0n-AVX2.
- `CORE_POWER.LVL2_TURBO_LICENSE`: Core cycles where the core was running in a manner where the maximum frequency was P0n-AVX-512.

When the core requests a higher license level than its current one, it takes the PCU up to 500 micro-seconds to grant the new license. Until then the core operates at a lower peak capability. During this time period the PCU evaluates how many cores are executing at the new license level and adjusts their frequency as necessary, potentially lowering the frequency. Cores that execute at other license levels are not affected.

A timer of approximately 2ms is applied before going back to a higher frequency level. Any condition that would have requested a new license resets the timer.

NOTES

A license transition request may occur when executing instructions on a mis-speculated path.

A large enough mix of Intel AVX-512 light instructions and Intel AVX2 heavy instructions drives the core to request License 2, despite the fact that they usually map to License 1. The same is true for Intel AVX2 light instructions and Intel SSE heavy instructions that may drive the core to License 1 rather than License 0. For example, The Intel® Xeon® Platinum 8180 processor moves from license 1 to license 2 when executing a mix of 110 Intel AVX-512 light instructions and 20 256-bit heavy instructions over a window of 65 cycles.

Some workloads do not cause the processor to reach its maximum frequency as these workloads are bound by other factors. For example, the LINPACK benchmark is power limited and does not reach the processor's maximum frequency. The following graph shows how frequency degrades as vector width grows, but, despite the frequency drop, performance improves. The data for this graph was collected on an Intel Xeon Platinum 8180 processor.

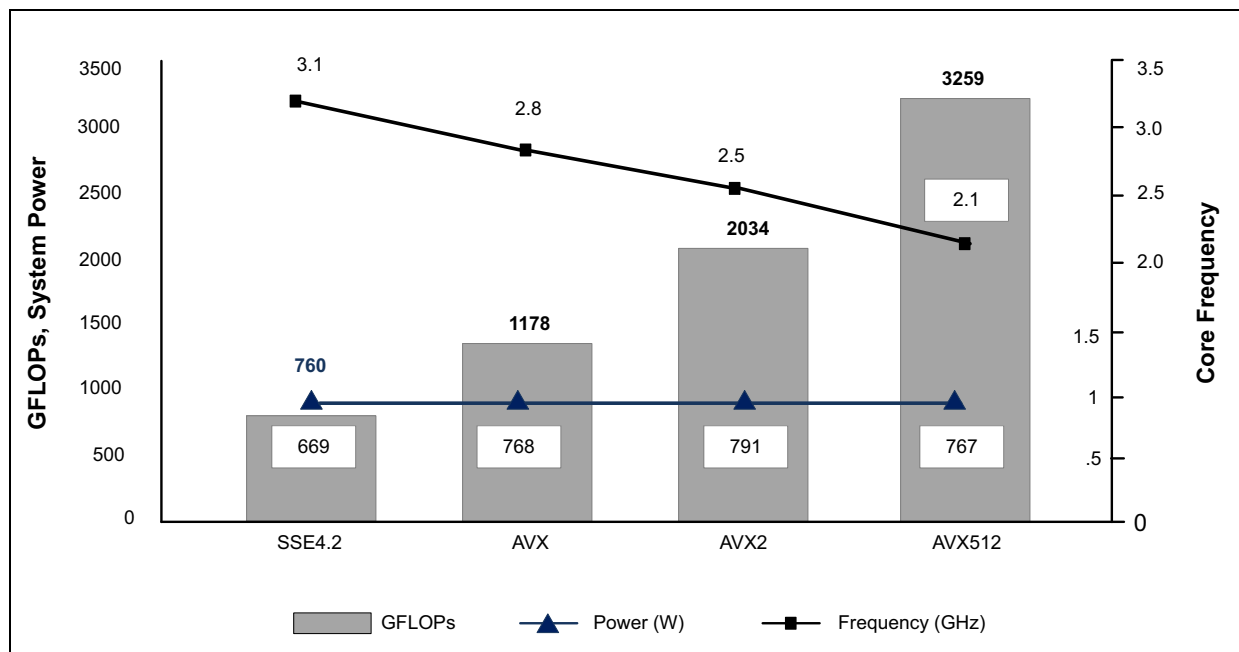


Figure 2-8. LINPACK Performance

Workloads that execute Intel AVX-512 instructions as a large proportion of their whole instruction count can gain performance compared to Intel AVX2 instructions, even though they may operate at a lower frequency. For example, maximum frequency bound Deep Learning workloads that target Intel AVX-512 heavy instructions at a very high percentage can gain 1.3x-1.5x performance improvement vs. the same workload built to target Intel AVX2 (both operating on Skylake Server microarchitecture).

It is not always easy to predict whether a program's performance will improve from building it to target Intel AVX-512 instructions. Programs that enjoy high performance gains from the use of xmm or ymm registers may expect performance improvement by moving to the use of zmm registers. However, some programs that use zmm registers may not gain as much, or may even lose performance. It is recommended to try multiple build options and measure the performance of the program.

Recommendation: To identify the optimal compiler options to use, build the application with each of the following set of options and choose the set that provides the best performance.

- `-xCORE-AVX2 -mtune=skylake-avx512` (Linux* and macOS*)
`/QxCORE-AVX2 /tune=skylake-avx512` (Windows*)
- `-xCORE-AVX512 -qopt-zmm-usage=low` (Linux* and macOS*)
`/QxCORE-AVX512 /Qopt-zmm-usage:low` (Windows*)
- `-xCORE-AVX512 -qopt-zmm-usage=high` (Linux* and macOS*)
`/QxCORE-AVX512 /Qopt-zmm-usage:high` (Windows*)

See [Section 17.26](#) for more information about these options.

The GCC Compiler has the option `-mprefer-vector-width=none|128|256|512` to control vector width preference. While `-march=skylake-avx512` is designed to provide the best performance for the Skylake Server microarchitecture some programs can benefit from different vector width preferences. To identify the optimal compiler options to use, build the application with each of the following set of options and choose the set that provides the best performance. `-mprefer-vector-width=256` is the default for `skylake-avx512`.

- `-march=skylake -mtune=skylake-avx512`
- `-march=skylake-avx512`
- `-march=skylake-avx512 -mprefer-vector-width=512`

Clang/LLVM is currently implementing the option `-mprefer-vector-width=none|128|256|512`, similar to GCC. To identify the optimal compiler options to use, build the application with each of the following set of options and choose the set that provides the best performance.

- `-march=skylake -mtune=skylake-avx512`
- `-march=skylake-avx512` (plus `-mprefer-vector-width=256`, if available)
- `-march=skylake-avx512` (plus `-mprefer-vector-width=512`, if available)

2.7 SKYLAKE CLIENT MICROARCHITECTURE

The Skylake client microarchitecture builds on the successes of the Haswell and Broadwell microarchitectures. The basic pipeline functionality of the Skylake client microarchitecture is depicted in [Figure 2-9](#).

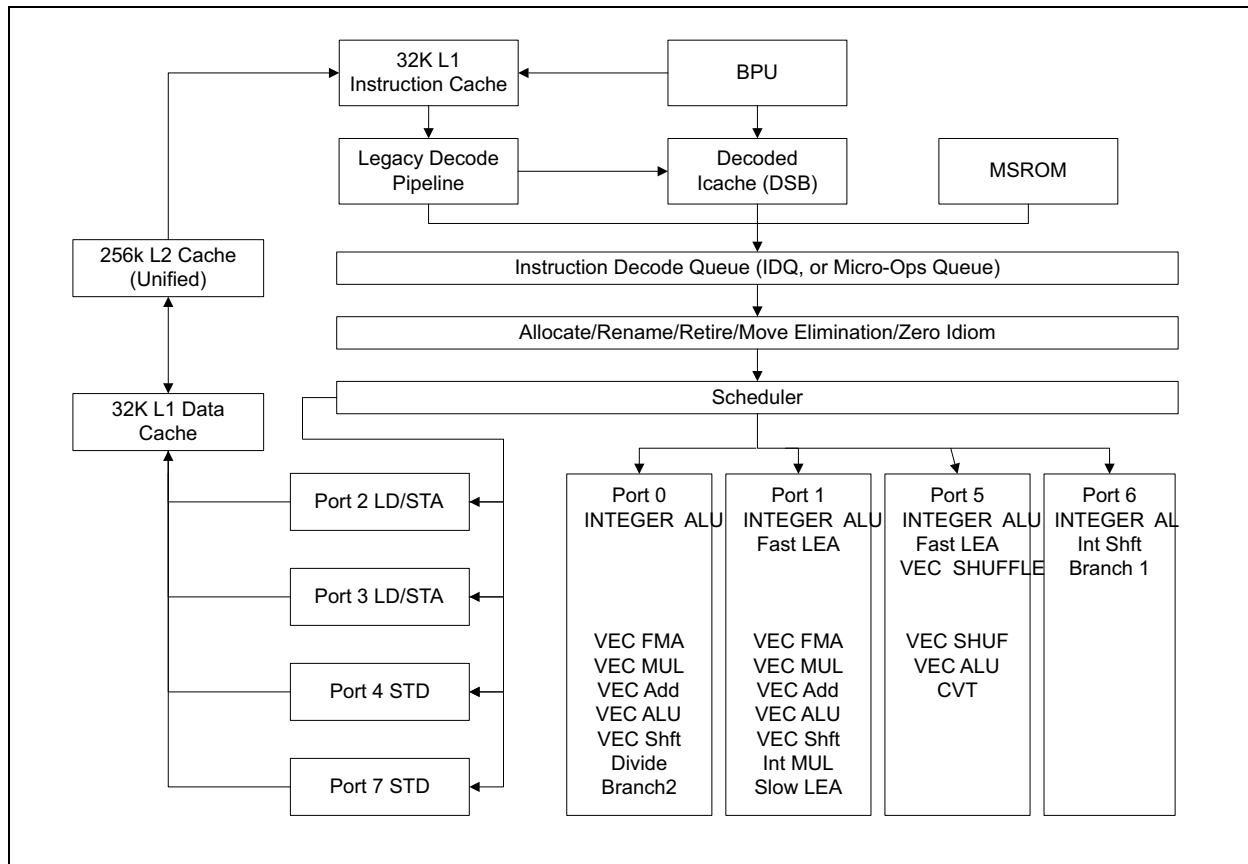


Figure 2-9. CPU Core Pipeline Functionality of the Skylake Client Microarchitecture

The Skylake Client microarchitecture offers the following enhancements:

- Larger internal buffers to enable deeper OOO execution and higher cache bandwidth.
- Improved front end throughput.
- Improved branch predictor.
- Improved divider throughput and latency.
- Lower power consumption.
- Improved SMT performance with Hyper-Threading Technology.
- Balanced floating-point ADD, MUL, FMA throughput and latency.

The microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of a number of components including a ring interconnect to multiple slices of L3 cache (an off-die L4 is optional), processor graphics, integrated memory controller, interconnect fabrics, etc.

2.7.1 THE FRONT END

The front end in the Skylake Client microarchitecture provides the following improvements over previous generation microarchitectures:

- Legacy Decode Pipeline delivery of 5 μ ops per cycle to the IDQ compared to 4 μ ops in previous generations.
- The DSB delivers 6 μ ops per cycle to the IDQ compared to 4 μ ops in previous generations.
- The IDQ can hold 64 μ ops per logical processor vs. 28 μ ops per logical processor in previous generations when two sibling logical processors in the same core are active (2x64 vs. 2x28 per core). If only one logical processor is active in the core, the IDQ can hold 64 μ ops (64 vs. 56 μ ops in ST operation).
- The LSD in the IDQ can detect loops up to 64 μ ops per logical processor irrespective ST or SMT operation.
- Improved Branch Predictor.

2.7.2 THE OUT-OF-ORDER EXECUTION ENGINE

The Out of Order and execution engine changes in Skylake Client microarchitecture include:

- Larger buffers enable deeper OOO execution compared to previous generations.
- Improved throughput and latency for divide/sqrt and approximate reciprocals.
- Identical latency and throughput for all operations running on FMA units.
- Longer pause latency enables better power efficiency and better SMT performance resource utilization.

[Table 2-13](#) summarizes the OOO engine's capability to dispatch different types of operations to various ports.

Table 2-13. Dispatch Port and Execution Stacks of the Skylake Client Microarchitecture

Port 0	Port 1	Port 2, 3	Port 4	Port 5	Port 6	Port 7
ALU, Vec ALU	ALU, Fast LEA, Vec ALU	LD STA	STD	ALU, Fast LEA, Vec ALU,	ALU, Shft,	STA
Vec Shft, Vec Add,	Vec Shft, Vec Add,			Vec Shuffle,	Branch1	
Vec Mul, FMA,	Vec Mul, FMA					
DIV,	Slow Int					
Branch2	Slow LEA					

[Table 2-14](#) lists execution units and common representative instructions that rely on these units. Throughput improvements across the SSE, AVX and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-14. Skylake Client Microarchitecture Execution Units and Representative Instructions¹

Execution Unit	# of Unit	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*

Table 2-14. Skylake Client Microarchitecture Execution Units and Representative Instructions¹

Execution Unit	# of Unit	Instructions
SHFT	2	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blmsk, bzhi, etc
Vec ALU	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2	(v)psllv*, (v)psrlv*, vector shift count in imm8
Vec Add	2	(v)addp*, (v)cmpp*, (v)max*, (v)min*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmqeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtsi2sd
Shuffle	1	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw
Vec Mul	2	(v)mul*, (v)pmul*, (v)pmadd*,
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm,
FP Mov	1	(v)movsd/ss, (v)movd gpr,
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See [Section 15.16.5](#) on MMX instruction throughput remedy.

A significant portion of the Intel SSE, Intel AVX and general-purpose instructions also have latency improvements. Appendix C lists the specific details. Software-visible latency exposure of an instruction sometimes may include additional contributions that depend on the relationship between micro-ops flows of the producer instruction and the micro-op flows of the ensuing consumer instruction. For example, a two- μ op instruction like VPMULLD may experience two cumulative bypass delays of 1 cycle each from each of the two micro-ops of VPMULLD.

[Table 2-15](#) describes the bypass delay in cycles between a producer μ op and the consumer μ op. The left-most column lists a variety of situations characteristic of the producer micro-op. The top row lists a variety of situations characteristic of the consumer micro-op.

Table 2-15. Bypass Delay Between Producer and Consumer Micro-ops

	SIMD/0,1/1	FMA/0,1/4	VIMUL/0,1/4	SIMD/5/1,3	SHUF/5/1,3	V2I/0/3	I2V/5/1
SIMD/0,1/1	0	1	1	0	0	0	NA
FMA/0,1/4	1	0	1	0	0	0	NA
VIMUL/0,1/4	1	0	1	0	0	0	NA
SIMD/5/1,3	0	1	1	0	0	0	NA
SHUF/5/1,3	0	0	1	0	0	0	NA

Table 2-15. Bypass Delay Between Producer and Consumer Micro-ops (Contd.)

	SIMD/0,1/1	FMA/0,1/4	VIMUL/0,1/4	SIMD/5/1,3	SHUF/5/1,3	V2I/0/3	I2V/5/1
V2I/0/3	NA	NA	NA	NA	NA	NA	NA
I2V/5/1	0	0	1	0	0	0	NA

The attributes that are relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the μ op. For example:

- “SIMD/0,1/1” applies to 1-cycle vector SIMD μ op dispatched to either port 0 or port 1.
- “VIMUL/0,1/4” applies to 4-cycle vector integer multiply μ op dispatched to either port 0 or port 1.
- “SIMD/5/1,3” applies to either 1-cycle or 3-cycle non-shuffle μ op dispatched to port 5.

2.7.3 CACHE AND MEMORY SUBSYSTEM

The cache hierarchy of the Skylake Client microarchitecture has the following enhancements:

- Higher Cache bandwidth compared to previous generations.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Processor can do two page walks in parallel compared to one in Haswell microarchitecture and earlier generations.
- Page split load penalty down from 100 cycles in previous generation to 5 cycles.
- L3 write bandwidth increased from 4 cycles per line in previous generation to 2 per line.
- Support for the CLFLUSHOPT instruction to flush cache lines and manage memory ordering of flushed data using SFENCE.
- Reduced performance penalty for a software prefetch that specifies a NULL pointer.
- L2 associativity changed from 8 ways to 4 ways.

Table 2-16. Cache Parameters of the Skylake Client Microarchitecture

Level	Capacity / Associativity	Line Size (bytes)	Fastest Latency ¹	Peak Bandwidth (bytes/cyc)	Sustained Bandwidth (bytes/cyc)	Update Policy
First Level Data	32 KB/ 8	64	4 cycle	96 (2x32B Load + 1*32B Store)	~81	Writeback
Instruction	32 KB/8	64	N/A	N/A	N/A	N/A
Second Level	256KB/4	64	12 cycle	64	~29	Writeback
Third Level (Shared L3)	Up to 2MB per core/Up to 16 ways	64	44	32	~18	Writeback

NOTES:

1. Software-visible latency will vary depending on access patterns and other factors.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, plus unified TLB for L2. The partition column of [Table 2-17](#) indicates the resource sharing policy when Hyper-Threading Technology is active.

Table 2-17. TLB Parameters of the Skylake Client Microarchitecture

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	8 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2/4MB pages	1536	12	fixed
Second Level	1GB	16	4	fixed

2.7.4 PAUSE LATENCY IN SKYLAKE CLIENT MICROARCHITECTURE

The PAUSE instruction is typically used with software threads executing on two logical processors located in the same processor core, waiting for a lock to be released. Such short wait loops tend to last between tens and a few hundreds of cycles, so performance-wise it is better to wait while occupying the CPU than yielding to the OS. When the wait loop is expected to last for thousands of cycles or more, it is preferable to yield to the operating system by calling an OS synchronization API function, such as `WaitForSingleObject` on Windows* OS or `futex` on Linux.

The PAUSE instruction is intended to:

- Temporarily provide the sibling logical processor (ready to make forward progress exiting the spin loop) with competitively shared hardware resources. The competitively-shared microarchitectural resources that the sibling logical processor can utilize in the Skylake Client microarchitecture are listed below.
 - Front end slots in the Decode ICache, LSD and IDQ.
 - Execution slots in the RS.
- Save power consumed by the processor core compared with executing equivalent spin loop instruction sequence in the following configurations.
 - One logical processor is inactive (e.g., entering a C-state).
 - Both logical processors in the same core execute the PAUSE instruction.
 - HT is disabled (e.g. using BIOS options).

The latency of the PAUSE instruction in prior generation microarchitectures is about 10 cycles, whereas in Skylake Client microarchitecture it has been extended to as many as 140 cycles.

The increased latency (allowing more effective utilization of competitively-shared microarchitectural resources to the logical processor ready to make forward progress) has a small positive performance impact of 1-2% on highly threaded applications. It is expected to have negligible impact on less threaded applications if forward progress is not blocked executing a fixed number of looped PAUSE instructions. There's also a small power benefit in 2-core and 4-core systems.

As the PAUSE latency has been increased significantly, workloads that are sensitive to PAUSE latency will suffer some performance loss.

The following is an example of how to use the PAUSE instruction with a dynamic loop iteration count.

Notice that in the Skylake Client microarchitecture the RDTSC instruction counts at the machine's guaranteed P1 frequency independently of the current processor clock (see the INVARIANT TSC property), and therefore, when running in Intel® Turbo-Boost-enabled mode, the delay will remain constant, but the number of instructions that could have been executed will change.

Use Poll Delay function in your lock to wait a given amount of guaranteed P1 frequency cycles, specified in the “clocks” variable.

Example 2-9. Dynamic Pause Loop Example

```
#include <x86intrin.h>
#include <stdint.h>

/* A useful predicate for dealing with timestamps that may wrap.
Is a before b? Since the timestamps may wrap, this is asking whether it's
shorter to go clockwise from a to b around the clock-face, or anti-clockwise.
Times where going clockwise is less distance than going anti-clockwise
are in the future, others are in the past. e.g. a = MAX-1, b = MAX+1 (=0),
then a > b (true) does not mean a reached b; whereas signed(a) = -2,
signed(b) = 0 captures the actual difference */

static inline bool before(uint64_t a, uint64_t b)
{
    return ((int64_t)b - (int64_t)a) > 0;
}

void pollDelay(uint32_t clocks)
{
    uint64_t endTime = _rdtsc() + clocks;

    for (; before(_rdtsc(), endTime); )
        _mm_pause();
}
```

For contended spinlocks of the form shown in the baseline example below, we recommend an exponential back off when the lock is found to be busy, as shown in the improved example, to avoid significant performance degradation that can be caused by conflicts between threads in the machine. This is more important as we increase the number of threads in the machine and make changes to the architecture that might aggravate these conflict conditions. In multi-socket Intel server processors with shared memory, conflicts across threads take much longer to resolve as the number of threads contending for the same lock increases. The exponential back off is designed to avoid these conflicts between the threads thus avoiding the potential performance degradation. Note that in the example below,

the number of PAUSE instructions are increased by a factor of 2 until some MAX_BACKOFF is reached which is subject to tuning.

Example 2-10. Contended Locks with Increasing Back-off Example

```

/*****/
/*Baseline Version */
/*****/

// atomic {if (lock == free) then change lock state to busy}
while (cmpxchg(lock, free, busy) == fail)
{
    while (lock == busy)
    {
        __asm__ ("pause");
    }
}

/*****/
/*Improved Version */
/*****/

int mask = 1;
int const max = 64; //MAX_BACKOFF
while (cmpxchg(lock, free, busy) == fail)
{
    while (lock == busy)
    {
        for (int i=mask; i; --i){
            __asm__ ("pause");
        }
        mask = mask < max ? mask<<1 : max;
    }
}

```

2.8 INTEL® 64 AND IA-32 INSTRUCTION BEST PRACTICES

2.8.1 NON-PRIVILEGED INSTRUCTION SERIALIZATION

Software may be required to serialize the execution pipeline of the current processor for various reasons in parallel execution environments. Several defined instructions, privileged and non-privileged, are classified as serializing instructions.

The concept of serializing instructions was introduced into the IA-32 architecture with the Pentium processor to support parallel instruction execution. Serializing instructions have no meaning for the Intel486 and earlier processors that do not implement parallel instruction execution. It is important to note that executing serializing instructions on P6 and more recent processor families constrains speculative execution because the results of speculatively executed instructions are discarded.

These instructions ensure the processor completes all modifications to flags, registers, and memory by previous instructions and drains all buffered writes to memory before the next instruction is fetched and executed. The non-privileged serialization instructions are:

- SERIALIZE
- CPUID
- IRET
- RSM

The SERIALIZE instruction was introduced in the Sapphire Rapids and Alder Lake platforms as a purpose-specific method of providing serialization to supersede the current typical usages such as CPUID.(EAX=0H). For example, CPUID usage for serialization has issues such that registers [EAX, EBX, ECX, EDX] are modified and, when executed on top of a VMM, will always incur the latency of a VM exit/VM entry round trip. SERIALIZE does not modify registers, arithmetic flags, or memory and does not incur a VM exit. The **SERIALIZE** instruction is enumerated via **CPUID.(EAX=07H,ECX=0):EDX[14]=1**, and software must verify support before usage.

Software that uses CPUID for serialization is recommended to use **Leaf 0 [CPUID.(EAX=0H)] CPUID**. CPUID leaves have variable performance with Leaf 0 providing the lowest latency when executed natively.

2.8.2 INTEL® HYPER-THREADING TECHNOLOGY (INTEL® HT TECHNOLOGY)

Intel® Hyper-Threading Technology (Intel® HT Technology) enables software to take advantage of task-level, or thread-level parallelism by providing multiple logical processors within a physical processor package, or within each processor core in a physical processor package. In its first implementation in the Intel® Xeon® processor, Intel HT Technology makes a single physical processor (or a processor core) appear as two or more logical processors.

Most Intel Architecture processor families support Intel HT Technology with two logical processors in each processor core, or in a physical processor in early implementations. The rest of this section describes features of the early implementation of Intel HT Technology. Most of the descriptions also apply to later implementations supporting two logical processors. The microarchitecture sections in this chapter provide additional details to individual microarchitecture and enhancements to Intel HT Technology.

The two logical processors each have a complete set of architectural registers while sharing one single physical processor's resources. By maintaining the architecture state of two processors, an Intel HT Technology-capable processor looks like two processors to software, including operating system and application code.

By sharing resources needed for peak demands between two logical processors, Intel HT Technology is well suited for multiprocessor systems to provide an additional performance boost in throughput when compared to traditional MP systems.

[Figure 2-10](#) shows a typical bus-based symmetric multiprocessor (SMP) based on processors supporting Intel HT Technology. Each logical processor can execute a software thread, allowing a maximum of two software threads to execute simultaneously on one physical processor. The two software threads execute simultaneously, meaning that in the same clock cycle an “add” operation from logical processor 0 and another “add” operation and load from logical processor 1 can be executed simultaneously by the execution engine.

In the first implementation of Intel HT Technology, the physical execution resources are shared and the architecture state is duplicated for each logical processor. This minimizes the die area cost of implementing Intel HT Technology while still achieving performance gains for multithreaded applications or multitasking workloads.

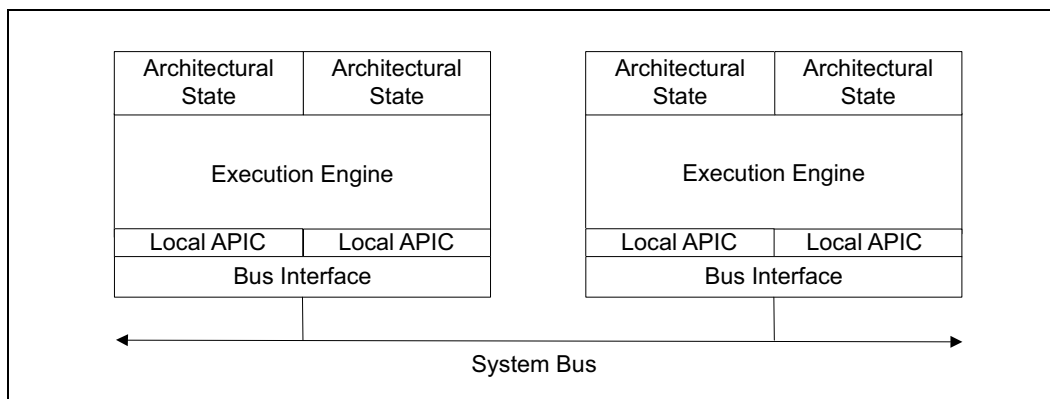


Figure 2-10. Intel® Hyper-Threading Technology on an SMP System

The performance potential due to Intel HT Technology is due to:

- The fact that operating systems and user programs can schedule processes or threads to execute simultaneously on the logical processors in each physical processor.
- The ability to use on-chip execution resources at a higher level than when only a single thread is consuming the execution resources; higher level of resource utilization can lead to higher system throughput.

2.8.2.1 Processor Resources and Intel® HT Technology

Most microarchitecture resources in a physical processor are shared between the logical processors. Only a few small data structures were replicated for each logical processor. This section describes how resources are shared, partitioned or replicated.

Replicated Resources

The architectural state is replicated for each logical processor. The architecture state consists of registers that are used by the operating system and application code to control program behavior and store data for computations. This state includes the eight general-purpose registers, the control registers, machine state registers, debug registers, and others. There are a few exceptions, most notably the memory type range registers (MTRRs) and the performance monitoring resources. For a complete list of the architecture state and exceptions, see the [Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C, & 3D](#).

Other resources such as instruction pointers and register renaming tables were replicated to simultaneously track execution and state changes of the two logical processors. The return stack predictor is replicated to improve branch prediction of return instructions.

In addition, a few buffers (for example, the two-entry instruction streaming buffers) were replicated to reduce complexity.

Partitioned Resources

Several buffers are shared by limiting the use of each logical processor to half the entries. These are referred to as partitioned resources. Reasons for this partitioning include:

- Operational fairness.
- Permitting the ability to allow operations from one logical processor to bypass operations of the other logical processor that may have stalled.

For example: a cache miss, a branch misprediction, or instruction dependencies may prevent a logical processor from making forward progress for some number of cycles. The partitioning prevents the stalled logical processor from blocking forward progress.

In general, the buffers for staging instructions between major pipe stages are partitioned. These buffers include μ op queues after the execution trace cache, the queues after the register rename stage, the reorder buffer which stages instructions for retirement, and the load and store buffers.

In the case of load and store buffers, partitioning also provided an easier implementation to maintain memory ordering for each logical processor and detect memory ordering violations.

Shared Resources

Most resources in a physical processor are fully shared to improve the dynamic utilization of the resource, including caches and all the execution units. Some shared resources which are linearly addressed, like the DTLB, include a logical processor ID bit to distinguish whether the entry belongs to one logical processor or the other.

2.8.2.2 Microarchitecture Pipeline and Intel® HT Technology

This section describes the Intel HT Technology microarchitecture and how instructions from the two logical processors are handled between the front end and the back end of the pipeline.

Although instructions originating from two programs or two threads execute simultaneously and not necessarily in program order in the execution core and memory hierarchy, the front end and back end contain several selection points to select between instructions from the two logical processors. All selection points alternate between the two logical processors unless one logical processor cannot make use of a pipeline stage. In this case, the other logical processor has full use of every cycle of the pipeline stage. Reasons why a logical processor may not use a pipeline stage include cache misses, branch mispredictions, and instruction dependencies.

2.8.2.3 Execution Core

The core can dispatch up to six μ ops per cycle, provided the μ ops are ready to execute. Once the μ ops are placed in the queues waiting for execution, there is no distinction between instructions from the two logical processors. The execution core and memory hierarchy is also oblivious to which instructions belong to which logical processor.

After execution, instructions are placed in the re-order buffer. The re-order buffer decouples the execution stage from the retirement stage. The re-order buffer is partitioned so each uses half the entries.

2.8.2.4 Retirement

The retirement logic tracks when instructions from the two logical processors are ready to be retired. It retires the instruction in program order for each logical processor by alternating between the two logical processors. If one logical processor is not ready to retire any instructions, then all retirement bandwidth is dedicated to the other logical processor.

Once stores have retired, the processor must write the store data into the level-one data cache. Selection logic alternates between the two logical processors to commit store data to the cache.

2.8.3 SIMD TECHNOLOGY

SIMD computations (see [Figure 2-11](#)) were introduced to the architecture with MMX technology. MMX technology allows SIMD computations to be performed on packed byte, word, and doubleword integers. The integers are contained in a set of eight 64-bit registers called MMX registers (see [Figure 2-12](#)).

Earlier processors extended the SIMD computation model with the introduction of the Streaming SIMD Extensions (SSE). SSE allows SIMD computations to be performed on operands that contain four packed single-precision floating-point data elements. The operands can be in memory or in a set of eight 128-bit XMM registers (see [Figure 2-12](#)). SSE also extended SIMD computational capability by adding additional 64-bit MMX instructions.

[Figure 2-11](#) shows a typical SIMD computation. Two sets of four packed data elements (X1, X2, X3, and X4, and Y1, Y2, Y3, and Y4) are operated on in parallel, with the same operation being performed on each corresponding pair of data elements (X1 and Y1, X2 and Y2, X3 and Y3, and X4 and Y4). The results of the four parallel computations are sorted as a set of four packed data elements.

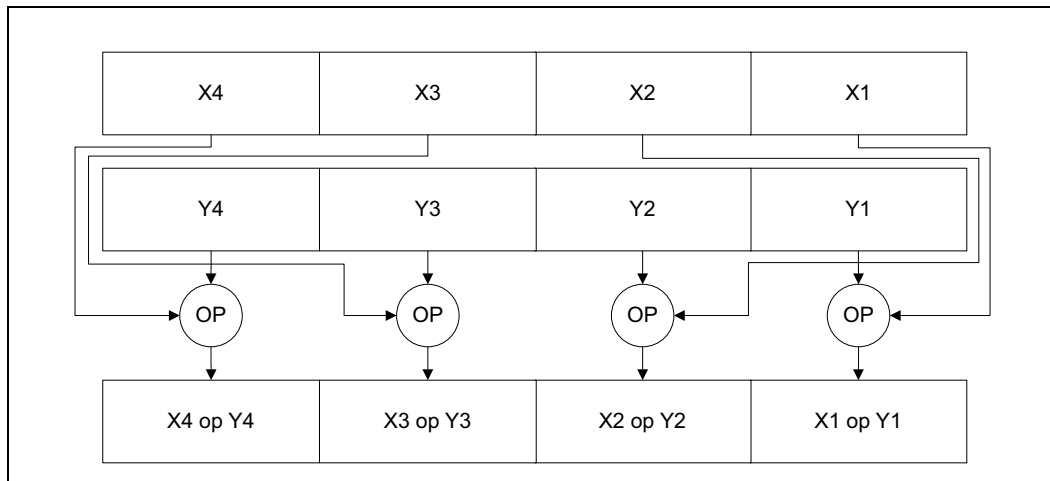


Figure 2-11. Typical SIMD Operations

The Pentium 4 processor further extended the SIMD computation model with the introduction of Streaming SIMD Extensions 2 (SSE2), Streaming SIMD Extensions 3 (SSE3), and Intel Xeon processor 5100 series introduced Supplemental Streaming SIMD Extensions 3 (SSSE3).

SSE2 works with operands in either memory or in the XMM registers. The technology extends SIMD computations to process packed double-precision floating-point data elements and 128-bit packed integers. There are 144 instructions in SSE2 that operate on two packed double-precision floating-point data elements or on 16 packed byte, 8 packed word, 4 doubleword, and 2 quadword integers.

SSE3 enhances x87, SSE and SSE2 by providing 13 instructions that can accelerate application performance in specific areas. These include video processing, complex arithmetics, and thread synchronization. SSE3 complements SSE and SSE2 with instructions that process SIMD data asymmetrically, facilitate horizontal computation, and help avoid loading cache line splits. See [Figure 2-12](#).

SSSE3 provides additional enhancement for SIMD computation with 32 instructions on digital video and signal processing.

SSE4.1, SSE4.2 and AESNI are additional SIMD extensions that provide acceleration for applications in media processing, text/lexical processing, and block encryption/decryption.

The SIMD extensions operates the same way in Intel 64 architecture as in IA-32 architecture, with the following enhancements:

- 128-bit SIMD instructions referencing XMM register can access 16 XMM registers in 64-bit mode.
- Instructions that reference 32-bit general purpose registers can access 16 general purpose registers in 64-bit mode.

64-bit MMX Registers	128-bit XMM Registers
MM7	XMM7
MM6	XMM6
MM5	XMM5
MM4	XMM4
MM3	XMM3
MM2	XMM2
MM1	XMM1
MM0	XMM0

Figure 2-12. SIMD Instruction Register Usage

SIMD improves the performance of 3D graphics, speech recognition, image processing, scientific applications and applications that have the following characteristics:

- Inherently parallel.
- Recurring memory access patterns.
- Localized recurring operations performed on the data.
- Data-independent control flow.

2.8.4 SUMMARY OF SIMD TECHNOLOGIES AND APPLICATION LEVEL EXTENSIONS

SIMD floating-point instructions fully support the IEEE Standard 754 for Binary Floating-Point Arithmetic. They are accessible from all IA-32 execution modes: protected mode, real address mode, and Virtual 8086 mode.

SSE, SSE2, and MMX technologies are architectural extensions. Existing software will continue to run correctly, without modification on Intel microprocessors that incorporate these technologies. Existing software will also run correctly in the presence of applications that incorporate SIMD technologies.

SSE and SSE2 instructions also introduced cacheability and memory ordering instructions that can improve cache usage and application performance.

For more on SSE, SSE2, SSE3 and MMX technologies, see the following chapters in the [Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1](#)

- [Chapter 9, "Programming with Intel® MMX™ Technology."](#)
- [Chapter 10, "Programming with Intel® Streaming SIMD Extensions \(Intel® SSE\)."](#)
- [Chapter 11, "Programming with Intel® Streaming SIMD Extensions 2 \(Intel® SSE2\)."](#)
- [Chapter 12, "Programming with Intel® SSE3, SSSE3, Intel® SSE4, and Intel® AES-NI."](#)
- [Chapter 14, "Programming with Intel® AVX, FMA, and Intel® AVX2."](#)
- [Chapter 15, "Programming with Intel® AVX-512."](#)

2.8.4.1 MMX™ Technology

MMX Technology introduced:

- 64-bit MMX registers.
- Support for SIMD operations on packed byte, word, and doubleword integers.

Recommendation: Integer SIMD code written using MMX instructions should consider more efficient implementations using SSE/Intel AVX instructions.

2.8.4.2 Streaming SIMD Extensions

Streaming SIMD extensions introduced:

- 128-bit XMM registers.
- 128-bit data type with four packed single-precision floating-point operands.
- Data prefetch instructions.
- Non-temporal store instructions and other cacheability and memory ordering instructions.
- Extra 64-bit SIMD integer support.

SSE instructions are useful for 3D geometry, 3D rendering, speech recognition, and video encoding and decoding.

2.8.4.3 Streaming SIMD Extensions 2

Streaming SIMD extensions 2 add the following:

- 128-bit data type with two packed double-precision floating-point operands.
- 128-bit data types for SIMD integer operation on 16-byte, 8-word, 4-doubleword, or 2-quadword integers.
- Support for SIMD arithmetic on 64-bit integer operands.
- Instructions for converting between new and existing data types.
- Extended support for data shuffling.
- Extended support for cacheability and memory ordering operations.

SSE2 instructions are useful for 3D graphics, video decoding/encoding, and encryption.

2.8.4.4 Streaming SIMD Extensions 3

Streaming SIMD extensions 3 add the following:

- SIMD floating-point instructions for asymmetric and horizontal computation.
- A special-purpose 128-bit load instruction to avoid cache line splits.
- An x87 FPU instruction to convert to integer independent of the floating-point control word (FCW).
- Instructions to support thread synchronization.

SSE3 instructions are useful for scientific, video and multi-threaded applications.

2.8.4.5 Supplemental Streaming SIMD Extensions 3

The Supplemental Streaming SIMD Extensions 3 introduces 32 new instructions to accelerate eight types of computations on packed integers. These include:

- 12 instructions that perform horizontal addition or subtraction operations.

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

2.8.5 SSE4.1

SSE4.1 introduces 47 new instructions to accelerate video, imaging and 3D applications. SSE4.1 also improves compiler vectorization and significantly increase support for packed dword computation. These include:

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

2.8.5.1 SSE4.2

SSE4.2 introduces 7 new instructions. These include:

- A 128-bit SIMD integer instruction for comparing 64-bit integer data elements.
- Four string/text processing instructions providing a rich set of primitives, these primitives can accelerate:
 - Basic and advanced string library functions from strlen, strcmp, to strcspn.
 - Delimiter processing, token extraction for lexing of text streams.
 - Parser, schema validation including XML processing.
- A general-purpose instruction for accelerating cyclic redundancy checksum signature calculations.
- A general-purpose instruction for calculating bit count population of integer numbers.

2.8.5.2 AESNI and PCLMULQDQ

AESNI introduces seven new instructions, six of them are primitives for accelerating algorithms based on AES encryption/decryption standard, referred to as AESNI.

The PCLMULQDQ instruction accelerates general-purpose block encryption, which can perform carry-less multiplication for two binary numbers up to 64-bit wide.

Typically, algorithm based on AES standard involve transformation of block data over multiple iterations via several primitives. The AES iteration.

AES encryption involves processing 128-bit input data (plain text) through a finite number of iterative operation, referred to as “AES round”, into a 128-bit encrypted block (ciphertext). Decryption follows the reverse direction of iterative operation using the “equivalent inverse cipher” instead of the “inverse cipher”.

The cryptographic processing at each round involves two input data, one is the “state”, the other is the “round key”. Each round uses a different “round key”. The round keys are derived from the cipher key using a “key schedule” algorithm. The “key schedule” algorithm is independent of the data processing of encryption/decryption, and can be carried out independently from the encryption/decryption phase.

The AES extensions provide two primitives to accelerate AES rounds on encryption, two primitives for AES rounds on decryption using the equivalent inverse cipher, and two instructions to support the AES key expansion procedure.

2.8.5.3 Intel® Advanced Vector Extensions (Intel® AVX)

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

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

Intel AVX instruction set and 256-bit register state management detail are described in [Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D](#). Optimization techniques for Intel AVX are discussed in [Chapter 15, “Programming with Intel® AVX-512.”](#)

2.8.5.4 Half-Precision Floating-Point Conversion (F16C)

VCVTPH2PS and VCVTPS2PH are two instructions supporting half-precision floating-point data type conversion to and from single-precision floating-point data types. These two instruction extends on the same programming model as Intel AVX.

2.8.5.5 RDRAND

The RDRAND instruction retrieves a random number supplied by a cryptographically secure, deterministic random bit generator (DBRG). The DBRG is designed to meet NIST SP 800-90A standard.

2.8.5.6 Fused-Multiply-ADD (FMA) Extensions

FMA extensions enhances Intel AVX with high-throughput, arithmetic capabilities covering fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and

multiply-subtract operations. FMA extensions provide 36 256-bit floating-point instructions to perform computation on 256-bit vectors and additional 128-bit and scalar FMA instructions.

2.8.5.7 Intel® Advanced Vector Extensions 2 (Intel® AVX2)

Intel® AVX2 extends Intel AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. Intel AVX2 instructions follow the same programming model as AVX instructions.

In addition, Intel AVX2 provide enhanced functionalities for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

2.8.5.8 General-Purpose Bit-Processing Instructions

The fourth generation Intel Core processor family introduces a collection of bit processing instructions that operate on the general purpose registers. Most of these instructions uses the VEX-prefix encoding scheme to provide non-destructive source operand syntax.

These instructions are enumerated by three separate feature flags reported by CPUID. For details, see Section 5.1 of [Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1](#) and chapters 3, 4 and 5 of the [Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D](#).

2.8.5.9 RDSEED

The RDSEED instruction retrieves a random number supplied by a cryptographically secure, enhanced deterministic random bit generator Enhanced NRBG). The NRBG is designed to meet the NIST SP 800-90B and NIST SP 800-90C standards.

2.8.5.10 ADCX and ADOX Instructions

The ADCX and ADOX instructions, in conjunction with MULX instruction, enable software to speed up calculations that require large integer numerics.