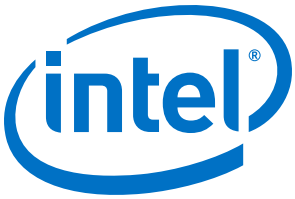


Intel® Arria® 10 Avalon-ST Interface for PCIe* Solutions User Guide

UG-01145_avst
2017.05.15

Last updated for Intel® Quartus® Prime Design Suite: 17.0


Subscribe


Send Feedback

Contents

1 Datasheet

1.1 Arria 10 Avalon-ST Interface for PCIe Datasheet

Intel® Arria® 10 FPGAs include a configurable, hardened protocol stack for PCI Express® that is compliant with *PCI Express Base Specification 3.0*. The Hard IP for PCI Express using the Avalon® Streaming (Avalon-ST) interface is the most flexible variant. However, this variant requires a thorough understanding of the PCIe® Protocol.

Figure 1. Arria 10 PCIe Variant with Avalon-ST Interface

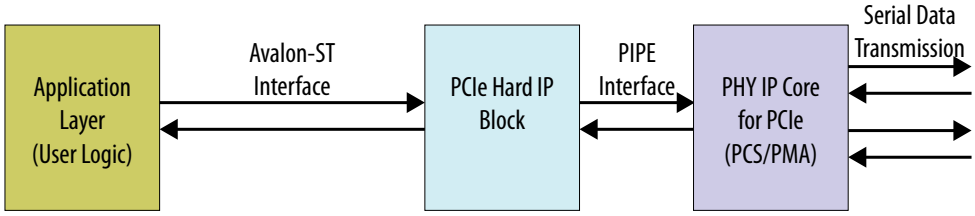


Table 1. PCI Express Data Throughput

The following table shows the aggregate bandwidth of a PCI Express link for Gen1, Gen2, and Gen3 for 1, 2, 4, and 8 lanes. This table provides bandwidths for a single transmit (TX) or receive (RX) channel. The numbers double for duplex operation. The protocol specifies 2.5 giga-transfers per second for Gen1, 5.0 giga-transfers per second for Gen2, and 8.0 giga-transfers per second for Gen3. Gen1 and Gen2 use 8B/10B encoding which introduces a 20% overhead. In contrast, Gen3 uses 128b/130b encoding which reduces the data throughput lost to encoding to about 1.5%.

	Link Width			
	×1	×2	×4	×8
PCI Express Gen1 (2.5 Gbps)	2	4	8	16
PCI Express Gen2 (5.0 Gbps)	4	8	16	32
PCI Express Gen3 (8.0 Gbps)	7.87	15.75	31.51	63

Refer to the *AN 456: PCI Express High Performance Reference Design* for more information about calculating bandwidth for the hard IP implementation of PCI Express in many Intel FPGAs, including the Arria 10 Hard IP for PCI Express IP core.

Devices

Related Links

- Arria 10 Avalon-ST Interface for PCIe Solutions User Guide Archive on page 232

- Introduction to Intel FPGA IP Cores
  Provides general information about all Intel FPGA IP cores, including parameterizing, generating, upgrading, and simulating IP cores.
- Creating Version-Independent IP and Qsys Simulation Scripts
  Create simulation scripts that do not require manual updates for software or IP version upgrades.
- Project Management Best Practices
  Guidelines for efficient management and portability of your project and IP files.
- PCI Express High Performance Reference Design
  For a design example demonstrating DMA performance that you can download to an Intel Development Kit.
- PCI Express Base Specification 3.0

1.1.1 Arria 10 Features

New features in the Quartus® Prime 17.0 software release:

- Added optional soft DFE controller IP to improve bit error rate (BER) margin. This option is available on the **PHY** tab of the parameter editor. The default for this option is off because it is typically not required. Short reflective links may benefit from this soft DFE controller IP. This parameter is available only for Gen3 configurations.

The Arria 10 Hard IP for PCI Express supports the following features:

- Complete protocol stack including the Transaction, Data Link, and Physical Layers implemented as hard IP.
- Support for ×1, ×2, ×4, and ×8 configurations with Gen1, Gen2, or Gen3 lane rates for Root Ports and Native Endpoints.
- Dedicated 16 KB receive buffer.
- Optional support for Configuration via Protocol (CvP) using the PCIe link allowing the I/O and core bitstreams to be stored separately.
- Qsys example designs demonstrating parameterization, design modules, and connectivity.
- Extended credit allocation settings to better optimize the RX buffer space based on application type.
- Support for multiple packets per cycle with the 256-bit Avalon-ST interface.
- Optional end-to-end cyclic redundancy code (ECRC) generation and checking and advanced error reporting (AER) for high reliability applications.
- Easy to use:
  - Flexible configuration.
  - Substantial on-chip resource savings and guaranteed timing closure.
  - No license requirement.
  - Example designs to get started.

Table 2. Feature Comparison for all Hard IP for PCI Express IP Cores

The table compares the features for three variants of the Hard IP for PCI Express IP Core. An SR-IOV variant is also available, but not included because it is very specialized product. Consult the *Arria 10 Avalon-ST Interface with SR-IOV PCIe Solutions User Guide* for features of this IP core.

Feature	Avalon-ST Interface	Avalon-MM Interface	Avalon-MM DMA
IP Core License	Free	Free	Free
Native Endpoint	Supported	Supported	Supported
Root port	Supported	Supported	Not Supported
Gen1	×1, ×2, ×4, ×8	×1, ×2, ×4, ×8	Not Supported
Gen2	×1, ×2, ×4, ×8	×1, ×2, ×4, ×8	×4, ×8
Gen3	×1, ×2, ×4, ×8	×1, ×2, ×4	×2, ×4, ×8
64-bit Application Layer interface	Supported	Supported	Not supported
128-bit Application Layer interface	Supported	Supported	Supported
256-bit Application Layer interface	Supported	Not Supported	Supported
Maximum payload size	128, 256, 512, 1024, 2048 bytes	128, 256 bytes	128, 256 bytes
Number of tags supported for non-posted requests	32, 64, 128, 256	8 for 64-bit interface 16 for 128-bit interface	16 or 256
Automatically handle out-of-order completions (transparent to the Application Layer)	Not supported	Supported	Not Supported
Automatically handle requests that cross 4 KB address boundary (transparent to the Application Layer)	Not supported	Supported	Supported
Polarity Inversion of PIPE interface signals	Supported	Supported	Supported
Number of MSI requests	1, 2, 4, 8, 16, or 32	1, 2, 4, 8, 16, or 32	1, 2, 4, 8, 16, or 32
MSI-X	Supported	Supported	Supported
Legacy interrupts	Supported	Supported	Supported
Expansion ROM	Supported	Not supported	Not supported
PCIe bifurcation	Not supported	Not supported	Not supported

Table 3. TLP Support Comparison for all Hard IP for PCI Express IP Cores

The table compares the TLP types that the variants of the Hard IP for PCI Express IP Cores can transmit. Each entry indicates whether this TLP type is supported (for transmit) by Endpoints (EP), Root Ports (RP), or both (EP/RP).

Transaction Layer Packet type (TLP) (transmit support)	Avalon-ST Interface	Avalon-MM Interface	Avalon-MM DMA
Memory Read Request (`Mrd`)	EP/RP	EP/RP	EP
Memory Read Lock Request (`MRdLk`)	EP/RP		EP
Memory Write Request (`MWr`)	EP/RP	EP/RP	EP
I/O Read Request (`IORd`)	EP/RP	EP/RP	
I/O Write Request (`IOWr`)	EP/RP	EP/RP	
Config Type 0 Read Request (`CfgRd0`)	RP	RP	

continued...

Transaction Layer Packet type (TLP) (transmit support)	Avalon-ST Interface	Avalon-MM Interface	Avalon-MM DMA
Config Type 0 Write Request (CfgWr0)	RP	RP	
Config Type 1 Read Request (CfgRd1)	RP	RP	
Config Type 1 Write Request (CfgWr1)	RP	RP	
Message Request (Msg)	EP/RP	EP/RP	
Message Request with Data (MsgD)	EP/RP	EP/RP	
Completion (Cpl)	EP/RP	EP/RP	EP
Completion with Data (CplD)	EP/RP		EP
Completion-Locked (CplLk)	EP/RP		
Completion Lock with Data (CplDLk)	EP/RP		
Fetch and Add AtomicOp Request (FetchAdd)	EP		

The *Arria 10 Avalon-ST Interface for PCIe Solutions User Guide* explains how to use this IP core and not the PCI Express protocol. Although there is inevitable overlap between these two purposes, use this document only in conjunction with an understanding of the *PCI Express Base Specification*.

Note: This release provides separate user guides for the different variants. The Related Information provides links to all versions.

Related Links

- Arria 10 Avalon-MM DMA Interface for PCIe Solutions User Guide
  For the Avalon-MM interface and DMA functionality.
- Arria 10 Avalon-MM Interface for PCIe Solutions User Guide
  For the Avalon-MM interface with no DMA.
- Arria 10 Avalon-ST Interface with SR-IOV PCIe Solutions User Guide
  For the Avalon-ST interface with Single Root I/O Virtualization (SR-IOV).

1.2 Release Information

Table 4. Hard IP for PCI Express Release Information

Item	Description
Version	17.0
Release Date	May 2017
Ordering Codes	No ordering code is required
Product IDs	There are no encrypted files for the Arria 10 Hard IP for PCI Express. The Product ID and Vendor ID are not required because this IP core does not require a license.
Vendor ID	

Intel verifies that the current version of the Quartus Prime software compiles the previous version of each IP core, if this IP core was included in the previous release. Intel reports any exceptions to this verification in the *Intel IP Release Notes* or clarifies them in the Quartus Prime IP Update tool. Intel does not verify compilation with IP core versions older than the previous release.

Related Links

- Errata for the Arria 10 Hard IP for PCI Express IP Core in the Knowledge Base
- Intel FPGA IP Release Notes
  Provides release notes for the current and past versions Intel FPGA IP cores.

1.3 Device Family Support

The following terms define device support levels for Intel® FPGA IP cores:

- **Advance support**—the IP core is available for simulation and compilation for this device family. Timing models include initial engineering estimates of delays based on early post-layout information. The timing models are subject to change as silicon testing improves the correlation between the actual silicon and the timing models. You can use this IP core for system architecture and resource utilization studies, simulation, pinout, system latency assessments, basic timing assessments (pipeline budgeting), and I/O transfer strategy (data-path width, burst depth, I/O standards tradeoffs).
- **Preliminary support**—the IP core is verified with preliminary timing models for this device family. The IP core meets all functional requirements, but might still be undergoing timing analysis for the device family. It can be used in production designs with caution.
- **Final support**—the IP core is verified with final timing models for this device family. The IP core meets all functional and timing requirements for the device family and can be used in production designs.

Table 5. Device Family Support

Device Family	Support Level
Arria 10	Final.
Other device families	Refer to the *Intel's PCI Express IP Solutions* web page for support information on other device families.

Related Links

PCI Express Solutions Web Page

1.4 Configurations

The Arria 10 Hard IP for PCI Express includes a full hard IP implementation of the PCI Express stack including the following layers:

- Physical (PHY), including:
  - Physical Media Attachment (PMA)
  - Physical Coding Sublayer (PCS)
- Media Access Control (MAC)
- Data Link Layer (DL)
- Transaction Layer (TL)

The Hard IP supports all memory, I/O, configuration, and message transactions. It is optimized for Intel devices. The Application Layer interface is also optimized to achieve maximum effective throughput. You can customize the Hard IP to meet your design requirements.

Figure 2. PCI Express Application with a Single Root Port and Endpoint

The following figure shows a PCI Express link between two Arria 10 FPGAs.

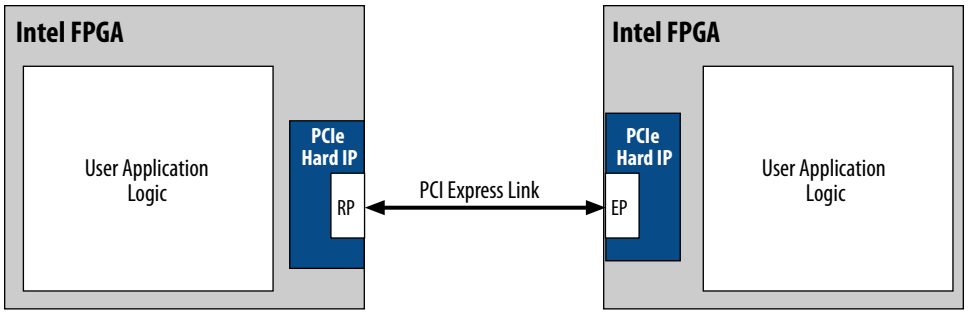
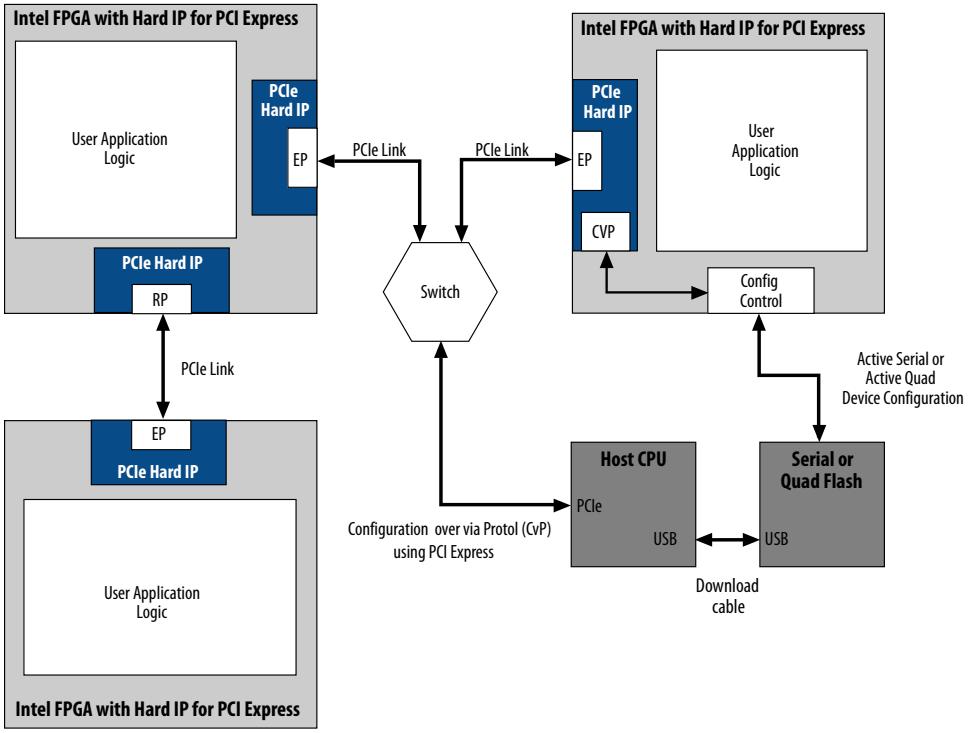


Figure 3. PCI Express Application Using Configuration via Protocol

The Arria 10 design below includes the following components:

- A Root Port that connects directly to a second FPGA that includes an Endpoint.
- Two Endpoints that connect to a PCIe switch.
- A host CPU that implements CvP using the PCI Express link connects through the switch.



Related Links

Arria 10 CvP Initialization and Partial Reconfiguration over PCI Express User Guide

1.5 Debug Features

Debug features allow observation and control of the Hard IP for faster debugging of system-level problems.

Related Links

Debugging on page 222

1.6 IP Core Verification

To ensure compliance with the PCI Express specification, Intel performs extensive verification. The simulation environment uses multiple testbenches that consist of industry-standard bus functional models (BFMs) driving the PCI Express link interface. Intel performs the following tests in the simulation environment:

- Directed and pseudorandom stimuli test the Application Layer interface, Configuration Space, and all types and sizes of TLPs
- Error injection tests inject errors in the link, TLPs, and Data Link Layer Packets (DLLPs), and check for the proper responses
- PCI-SIG® Compliance Checklist tests that specifically test the items in the checklist
- Random tests that test a wide range of traffic patterns

Intel provides example designs that you can leverage to test your PCBs and complete compliance base board testing (CBB testing) at PCI-SIG, upon request.

1.6.1 Compatibility Testing Environment

Intel has performed significant hardware testing to ensure a reliable solution. In addition, Intel internally tests every release with motherboards and PCI Express switches from a variety of manufacturers. All PCI-SIG compliance tests are run with each IP core release.

1.7 Performance and Resource Utilization

Because the PCIe protocol stack is implemented in hardened logic, it uses no core device resources (no ALMs and no embedded memory).

Related Links

- Fitter Resources Reports
  For numerous reports on hardware resources such as Differential I/O,PLLs, RAM usage, and GXB RX and TX channels.
- Running the Fitter
  For information on Fitter constraints.

1.8 Recommended Speed Grades

Recommended speed grades are pending characterization of production Arria 10 devices.

Table 6. Arria 10 Recommended Speed Grades for All Link Widths and Application Layer Clock Frequencies

Intel recommends setting the Quartus Prime Analysis & Synthesis Settings **Optimization Technique** to **Speed** when the Application Layer clock frequency is 250 MHz. For information about optimizing synthesis, refer to *Setting Up and Running Analysis and Synthesis* in Quartus II Help. For more information about how to effect the **Optimization Technique** settings, refer to *Area and Timing Optimization* in volume 2 of the *Quartus Prime Handbook*.

Link Rate	Link Width	Interface Width	Application Clock Frequency (MHz)	Recommended Speed Grades
Gen1	x1	64 bits	62.5[1],125	–1, –2 , –3
	x2	64 bits	125	–1, –2, –3
	x4	64 bits	125	–1, –2, –3
	x8	64 bits	250	–1, –2
	x8	128 Bits	125	–1, –2, –3
Gen2	x1	64 bits	125	–1, –2, –3
	x2	64 bits	125	–1, –2, –3
	x4	64 bits	250	–1, –2
	x4	128 bits	125	–1, –2, –3
	x8	128 bits	250	–1, –2
	x8	256 bits	125	–1, –2, –3
Gen3	x1	64 bits	125	–1, –2, –3
	x2	64 bits	250	–1, –2
	x2	128 bits	125	–1, –2, –3
	x4	128 bits	250	–1, –2
	x4	256 bits	125	–1, –2, –3
	x8	256 bits	250	–1, –2

Related Links

- Intel FPGA Software Installation and Licensing
  Provides comprehensive information for installing and licensing Intel FPGA software.
- Running Synthesis
  For settings that affect timing closure.

1 This is a power-saving mode of operation

1.9 Creating a Design for PCI Express

Select the PCIe variant that best meets your design requirements.

- Is your design an Endpoint or Root Port?
- What Generation do you intend to implement?
- What link width do you intend to implement?
- What bandwidth does your application require?
- Does your design require Configuration via Protocol (CvP)?

1. Select parameters for that variant.
2. For Arria 10 devices, you can use the new **Example Design** tab of the component GUI to generate a design that you specify. Then, you can simulate this example and also download it to an Arria 10 FPGA Development Kit. Refer to the *Arria 10 PCI Express IP Core Quick Start Guide* for details.
3. For all devices, you can simulate using an Intel-provided example design. All static PCI Express example designs are available under `<install_dir>/ip/altera/altera_pcie/altera_pcie_<dev>_ed/example_design/<dev>`. Alternatively, generate an example design that matches your parameter settings, or create a simulation model and use your own custom or third-party BFM. The Qsys Generate menu generates simulation models. Intel supports ModelSim* - Intel FPGA Edition for all IP. The PCIe cores support the Aldec RivieraPro, Cadence NCsim, Mentor Graphics ModelSim, and Synopsys* VCS and VCS-MX simulators.

   The Intel testbench and Root Port or Endpoint BFM provide a simple method to do basic testing of the Application Layer logic that interfaces to the variation. However, the testbench and Root Port BFM are not intended to be a substitute for a full verification environment. To thoroughly test your application, Intel suggests that you obtain commercially available PCI Express verification IP and tools, or do your own extensive hardware testing, or both.
4. Compile your design using the Quartus Prime software. If the versions of your design and the Quartus Prime software you are running do not match, regenerate your PCIe design.
5. Download your design to an Intel development board or your own PCB. Click on the *All Development Kits* link below for a list of Intel's development boards.
6. Test the hardware. You can use Intel's SignalTap® Logic Analyzer or a third-party protocol analyzer to observe behavior.
7. Substitute your Application Layer logic for the Application Layer logic in Intel's testbench. Then repeat Steps 3–6. In Intel's testbenches, the PCIe core is typically called the DUT (device under test). The Application Layer logic is typically called APPS.

Related Links

- Arria 10 Parameter Settings on page 36
- Getting Started with the Arria 10 Hard IP for PCI Express on page 24
  For a design example that illustrates a chaining DMA application.
- Quick Start Guide on page 17
- All Development Kits
- Intel Wiki PCI Express

For complete design examples and help creating new projects and specific functions, such as MSI or MSI-X related to PCI Express. Intel Applications engineers regularly update content and add new design examples. These examples help designers like you get more out of the Intel PCI Express IP core and may decrease your time-to-market. The design examples of the Intel Wiki page provide useful guidance for developing your own design. However, the content of the Intel Wiki is not guaranteed by Intel.

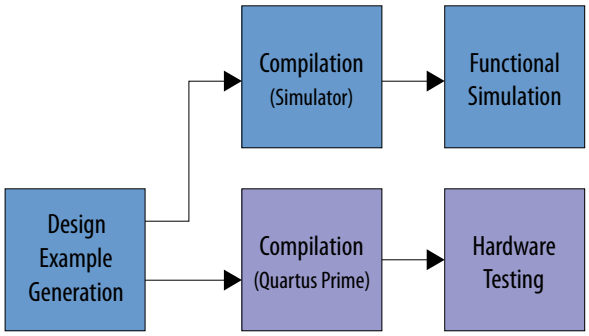
2 Quick Start Guide

The Intel Arria 10 Hard IP for PCI Express* IP core includes a programmed I/O (PIO) design example to help you understand usage. The PIO example transfers memory from a host processor to a target device. It is appropriate for low-bandwidth applications. The design example includes an Avalon-ST to Avalon-MM Bridge. This component translates the TLPs received on the PCIe* link to Avalon-MM memory reads and writes to the on-chip memory.

This design example automatically creates the files necessary to simulate and compile in the Quartus Prime software. You can download the compiled design to the Arria 10 GX FPGA Development Kit. The design examples cover a wide range of parameters. However, the automatically generated design examples do not cover all possible parameterizations of the PCIe IP Core. If you select an unsupported parameter set, generations fails and provides an error message.

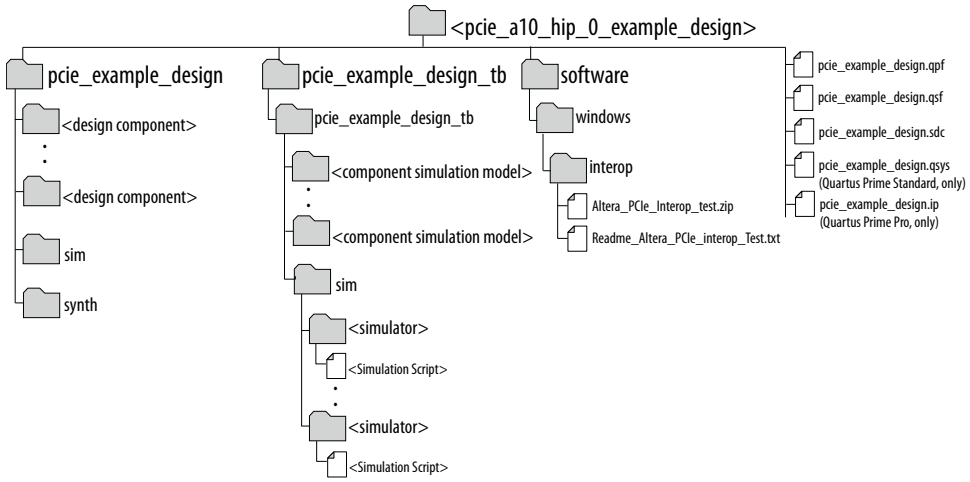
In addition, many static design examples for simulation are only available in the `<install_dir>/ip/altera/altera_pcie/altera_pcie_a10_ed/example_design/a10` directory.

Figure 4. Development Steps for the Design Example



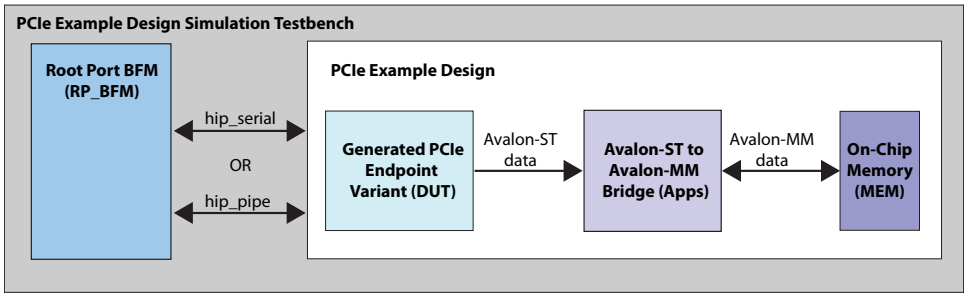
2.1 Directory Structure

Figure 5. Directory Structure for the Generated Design Example



2.2 Design Components

Figure 6. Block Diagram for the Qsys PIO Design Example Simulation Testbench

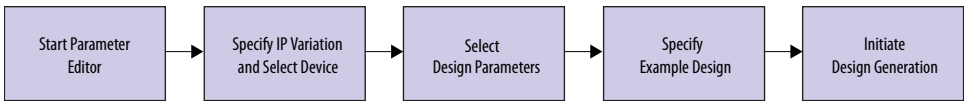


Related Links

Arria 10 Development Kit Conduit Interface on page 105
The Arria 10 Development Kit conduit interface signals are optional signals that allow you to connect your design to the Arria 10 FPGA Development Kit.

2.3 Generating the Design

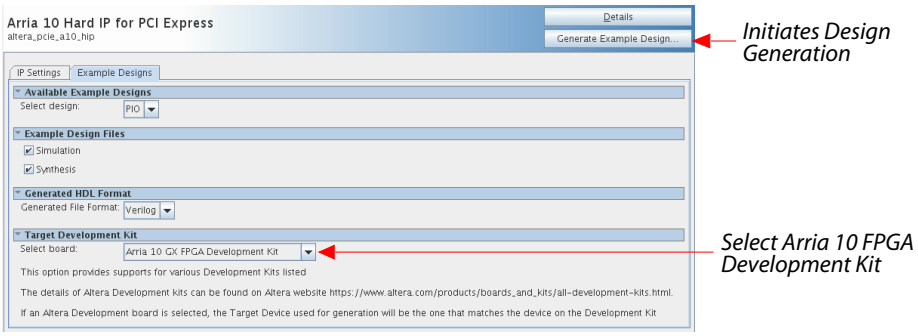
Figure 7. Procedure



Follow these steps to generate the design from the IP Parameter Editor:

1. In the IP Catalog (**Tools ➤ IP Catalog**) locate and select the **Arria 10 Hard IP for PCI Express**.
2. Starting with the Quartus Prime Pro 16.1 software, the **New IP Variation** dialog box appears.
3. Specify a top-level name and the folder for your custom IP variation, and the target device. Click **OK**
4. On the **IP Settings** tabs, specify the parameters for your IP variation.
5. On the **Example Designs** tab, the **PIO** design is available for your IP variation.

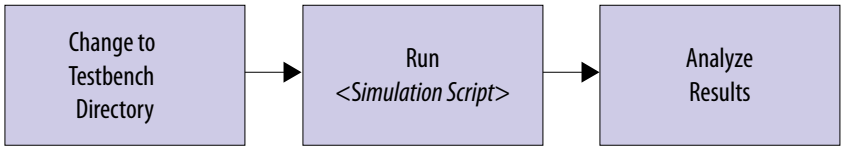
Figure 8. Example Design Tab



6. For **Example Design Files**, select the **Simulation** and **Synthesis** options.
7. For **Generated HDL Format**, only Verilog is available.
8. For **Target Development Kit** select the **Arria 10 FPGA Development Kit** option.
9. Click the **Generate Example Design** button. The software generates all files necessary to run simulations and hardware tests on the **Arria 10 FPGA Development Kit**.Click **Close** when generation completes.
10. Click **Finish**.
11. The prompt, **Recent changes have not been generated. Generate now?**, allows you to create files for simulation and synthesis. Click **No** to continue to simulate the design example you just generated.

2.4 Simulating the Design

Figure 9. Procedure



1. Change to the testbench simulation directory.
2. Run the simulation script for the simulator of your choice. Refer to the table below.
3. Analyze the results.

Table 7. Steps to Run Simulation

Simulator	Working Directory	Instructions
ModelSim	*<example_design>*/pcie_example_design_tb/pcie_example_design_tb/sim/mentor/	1. do msim_setup.tcl 2. ld_debug 3. run -all 4. A successful simulation ends with the following message, "Simulation stopped due to successful completion!"
VCS*	<example_design>/pcie_example_design_tb/pcie_example_design_tb/sim/synopsys/vcs	1. sh vcs_setup.sh USER_DEFINED_SIM_OPTIONS="" 2. A successful simulation ends with the following message, "Simulation stopped due to successful completion!"
Cadence*	*<example_design>*/pcie_example_design_tb/pcie_example_design_tb/sim/cadence	1. sh ncsim_setup.sh USER_DEFINED_SIM_OPTIONS="" 2. A successful simulation ends with the following message, "Simulation stopped due to successful completion!"

Figure 10. Partial Transcript from Successful Avalon-ST PIO Simulation Testbench

```
# INFO:           60504 ns            New Link Speed: 8.0GT/s
# INFO:           60576 ns   RP PCI Express Link Control Register (0040):
# INFO:           60576 ns       Common Clock Config: System Reference Clock Used
# INFO:           61640 ns   RP PCI Express Link Capabilities Register (01606483):
# INFO:           61640 ns        Maximum Link Width: x8
# INFO:           61640 ns      Supported Link Speed: 8.0GT/s or 5.0GT/s or 2.5GT/s
# INFO:           61640 ns                 L0s Entry: Supported
# INFO:           61640 ns                 L1  Entry: Not Supported
# INFO:           61640 ns           L0s Exit Latency: 2 us to 4 us
# INFO:           61640 ns           L1  Exit Latency: Less Than 1 us
# INFO:           61640 ns                Port Number: 01
# INFO:           61768 ns   RP PCI Express Device Control Register (5010):
# INFO:           61768 ns   Error Reporting Enables: 0
# INFO:           61768 ns          Relaxed Ordering: Enabled
# INFO:           61768 ns               Max Payload: 128 Bytes
# INFO:           61768 ns              Extended Tag: Disabled
# INFO:           61768 ns           Max Read Request: 4KBytes
# INFO:           61768 ns   RP PCI Express Device Status Register (0000):
# INFO:           62096 ns Configuring Bus 000, Device 000, Function 00
# INFO:           62096 ns   RP Read Only Configuration Registers:
# INFO:           62096 ns                 Vendor ID: 1172
# INFO:           62096 ns                 Device ID: E001
# INFO:           62096 ns               Revision ID: 01
# INFO:           62096 ns                Class Code: FF0000
# INFO:           62096 ns             Interrupt Pin: INTA# used
# INFO:           62784 ns BAR Address Assignments:
# INFO:           62784 ns BAR    Size       Assigned Address  Type
# INFO:           62784 ns BAR0   Disabled
# INFO:           62784 ns BAR1   Disabled
# INFO:           62784 ns ExpROM Disabled
# INFO:           66680 ns Completed configuration of Endpoint BARs.
# INFO:           67728 ns TASK:downstream_loop
# INFO:           68584 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:           69448 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:           70296 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:           71160 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:           72008 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:           72864 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:           73720 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:           74568 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:           75432 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:           76280 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
# SUCCESS: Simulation stopped due to successful completion!
```

Related Links

AN-811: Using the Avery BFM for PCI Express Gen3x16 Simulation on Intel Stratix 10 Devices

2.5 Compiling and Testing the Design in Hardware

Figure 11. Procedure

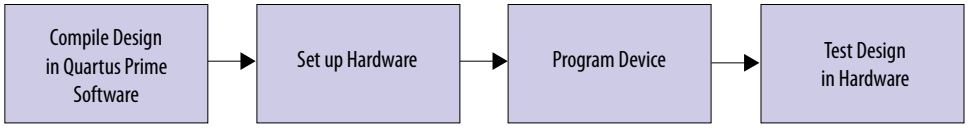
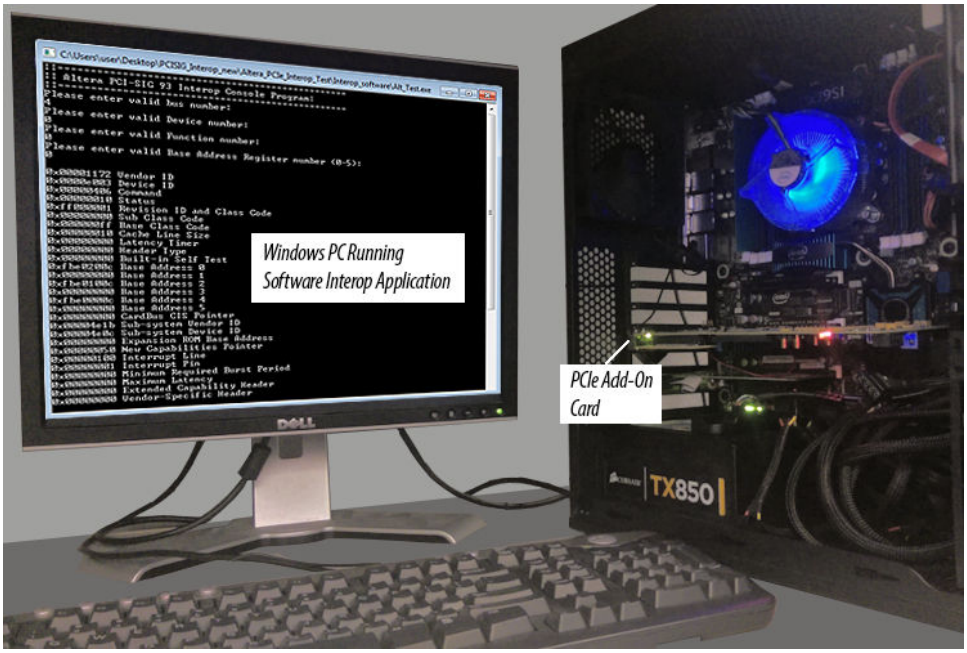


Figure 12. Software Application to Test the PCI Express Design Example on the Arria 10 GX FPGA Development Kit

A software application running on a Windows PC performs the same hardware test for all of the PCI Express Design Examples.



The software application to test the PCI Express Design Example on the Arria 10 GX FPGA Development Kit is available on both 32- and 64-bit Windows platforms. This program performs the following tasks:

1. Prints the Configuration Space, lane rate, and lane width.
2. Writes 0x00000000 to the specified BAR at offset 0x00000000 to initialize the memory and read it back.
3. Writes 0xABCD1234 at offset 0x00000000 of the specified BAR. Reads it back and compares.

If successful, the test program displays the message 'PASSED'

Follow these steps to compile the design example in the Quartus Prime software:

1. Launch the Quartus Prime software and open <*example_design*>`pcie_example_design.qpf`.
2. On the **Processing** > menu, select **Start Compilation**).

   The timing constraints for the design example and the design components are automatically loaded during compilation.

Follow these steps to test the design example in hardware:

1. In the <*example_design*>`/software/windows/interop` directory, unzip `Altera_PCIe_Interop_Test.zip`.

   Note: You can also refer to `readme_Altera_PCIe_interop_Test.txt` file in this same directory for instructions on running the hardware test.
2. Install the Intel FPGA Windows Demo Driver for PCIe on the Windows host machine, using **altera_pcie_win_driver.inf**.

   Note: If you modified the default Vendor ID or Device ID specified in the component GUI, you must also modify them in **altera_pcie_win_driver.inf**.

   a. In the <*example_design*> directory, launch the Quartus Prime software and compile the design (**Processing** > **Start Compilation**).

   b. Connect the development board to the host computer.

   c. Configure the FPGA on the development board using the generated `.sof` file (**Tools** > **Programmer**).

   d. Open the Windows Device Manager and scan for hardware changes.

   e. Select the Intel FPGA listed as an unknown PCI device and point to the appropriate 32- or 64-bit driver (**altera_pice_win_driver.inf**) in the **Windows_driver** directory.

   f. After the driver loads successfully, a new device named **Altera PCI API Device** appears in the Windows Device Manager.

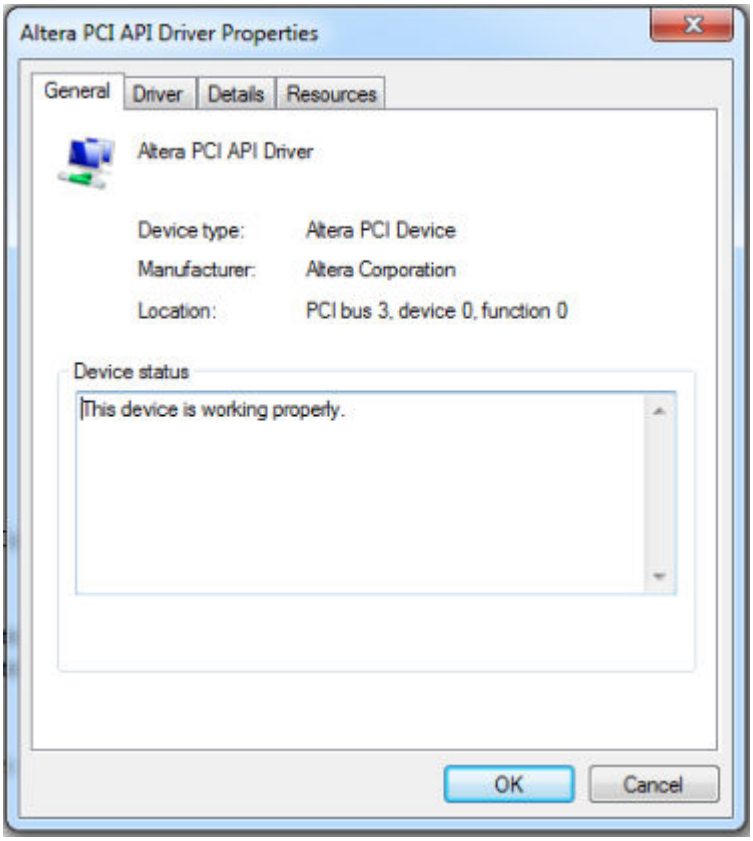
   g. Determine the bus, device, and function number for the **Altera PCI API Device** listed in the Windows Device Manager.

      i. Expand the tab, **Altera PCI API Driver** under the devices.

      ii. Right click on **Altera PCI API Device** and select **Properties**.

      iii. Note the bus, device, and function number for the device. The following figure shows one example.

Figure 13. Determining the Bus, Device, and Function Number for New PCIe Device



3. In the <*example_desing*>`/software/windows/interop/Altera_PCIe_Interop_Test/Interop_software` directory, click `Alt_Test.exe`.
4. When prompted, type the bus, device, and function numbers and select the BAR number (0-5) you specified when parameterizing the IP core.

   Note: The bus, device, and function numbers for your hardware setup may be different.
5. The test displays the message, PASSED, if the test is successful.

Related Links

- Arria 10 Development Kit Conduit Interface on page 105
  The Arria 10 Development Kit conduit interface signals are optional signals that allow you to connect your design to the Arria 10 FPGA Development Kit.
- Arria 10 GX FPGA Development Kit

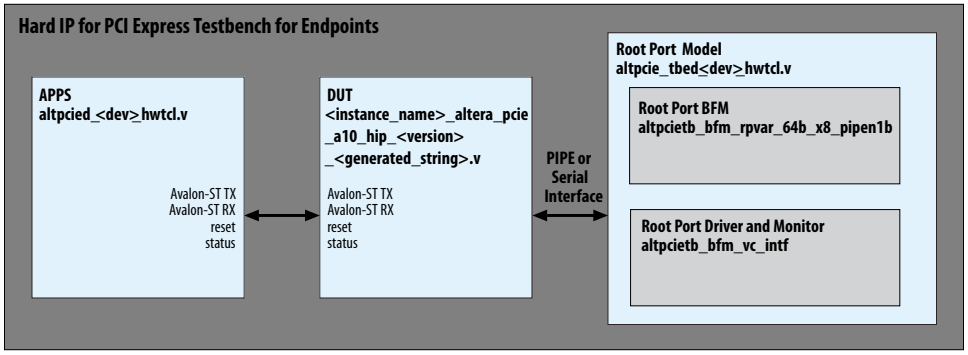
3 Getting Started with the Arria 10 Hard IP for PCI Express

This Gen1 x8 Endpoint design example illustrates a chaining DMA application. It provides instructions to help you quickly customize, simulate, and compile the Arria 10 Hard IP for PCI Express IP Core.This design examples creates the files required for simulation and synthesis, but does not generate all the files necessary to download the design to hardware. The *Quick Start Guide* described in the previous chapter does include all files necessary to download your design to the Arria 10 GX FPGA Development Kit

When you install the Quartus Prime software you also install the IP Library. This installation includes design examples for Hard IP for PCI Express under the `<install_dir>/ip/altera/altera_pcie/` directory.

After you install the Quartus Prime software, you can copy the design examples from the `<install_dir>/ip/altera/altera_pcie/altera_pcie_a10_ed/example_design/a10` directory. This walkthrough uses the Gen1 ×8 Endpoint, **ep_g1x8.qsys**. The following figure illustrates the top-level modules of the testbench in which the DUT, a Gen1 Endpoint, connects to a chaining DMA engine, labeled APPS in the following figure, and a Root Port model. The simulation can use the parallel PHY Interface for PCI Express (PIPE) or serial interface.

Figure 14. Testbench for an Endpoint



Note: The Quartus Prime software automatically creates a simulation log file, `altpcie_monitor_<dev>_dlhip_tlp_file_log.log`, in your simulation directory. Refer to *Understanding Simulation Log File Generation* for details.

Intel provides example designs to help you get started with the Arria 10 Hard IP for PCI Express IP Core. You can use example designs as a starting point for your own design. The example designs include scripts to compile and simulate the Arria 10 Hard IP for PCI Express IP Core. This example design provides a simple method to perform basic testing of the Application Layer logic that interfaces to the Hard IP for PCI Express.

For a detailed explanation of this example design, refer to the *Testbench and Design Example* chapter. If you choose the parameters specified in this chapter, you can run all of the tests included in *Testbench and Design Example* chapter.

For more information about Qsys, refer to *System Design with Qsys* in the *Quartus Prime Handbook*. For more information about the Qsys GUI, refer to *About Qsys* in Quartus Prime Help.

Related Links

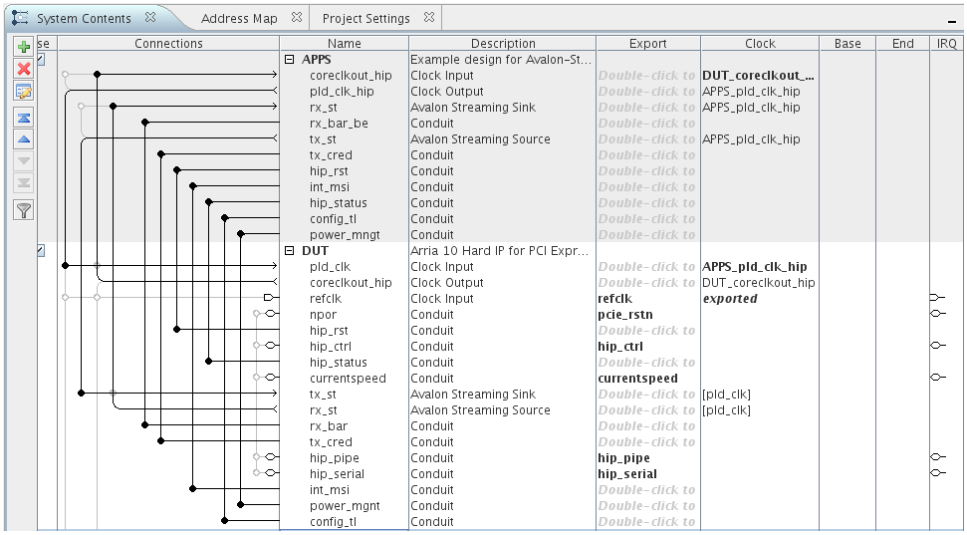
- Testbench and Design Example on page 174
- Understanding Simulation Log File Generation on page 27

3.1 Qsys Design Flow

Copy the **ep_g1x8.qsys** design example from the `<install_dir>/ip/altera/altera_pcie/altera_pcie/altera_pcie_a10_ed/example_designs/a10` to your working directory.

The following figure illustrates this Qsys system.

Figure 15. Complete Gen1 ×8 Endpoint (DUT) Connected to Example Design (APPS)



The example design includes the following components:

- DUT—This is Gen1 ×8 Endpoint. For your own design, you can select the data rate, number of lanes, and either Endpoint or Root Port mode.
- APPS—This Root Port BFM configures the DUT and drives read and write TLPs to test DUT functionality. An Endpoint BFM is available if your PCI Express design implements a Root Port.

3.1.1 Generating the Testbench

1. On the **Generate** menu, select **Generate Testbench System**. Specify the parameters listed in the following table.

Table 8. Parameters to Specify on the Generation Tab in Qsys

Parameter	Value
Testbench System	
Create testbench Qsys system	**Standard, BFMs for standard Qsys interfaces**
Create testbench simulation model	**Verilog**
Allow mixed-language simulation	Turn this option off
Output Directory	
Clear output directories for selected generation targets	Turn this option off
Testbench	<*working_dir*>/ep_g1x8_tb/

2. Click the **Generate** button at the bottom of the Generation tab to create the testbench.
   This testbench assumes that you are running the DMA application that the example design available in the installation directory creates. Otherwise, the testbench tests will probably fail unless your own testbench has equivalent functionality.

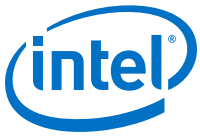
   Note: Arria 10 devices do not support the **Create timing and resource estimates for third-party EDA synthesis tools** option on the **Generate ➤ Generate HDL** menu. You can select this menu item, but generation fails.

3.1.2 Simulating the Example Design

1. Start your simulation tool. This example uses the ModelSim® software.
2. From the ModelSim transcript window, in the testbench directory, <*working_dir*>/ep_g1x8_tb/ep_g1x8_tb/sim/mentor, type the following commands:
   a. `do msim_setup.tcl`
   b. `ld_debug` (This command compiles all design files and elaborates the top-level design without any optimization.)
   c. `run -all`

The simulation includes the following stages:

- Link training
- Configuration
- DMA reads and writes
- Root Port to Endpoint memory reads and writes



Disabling Scrambling for Gen1 and Gen2 to Interpret TLPs at the PIPE Interface

1. Go to <*project_directory*/ep_g1x8_tb/ep_g1x8_tb/altera_pcie_a10_tbed_140/sim/.
2. Open **altpcietb_bfm_top_rp.v.**
3. Locate the assignment for `test_in[2:1]`. Set `test_in[2] = 1` and `test_in[1] = 0`. Changing `test_in[2] = 1` disables data scrambling on the PIPE interface.
4. Save **altpcietb_bfm_top_rp.v**.

3.1.3 Generating Synthesis Files

1. On the **Generate** menu, select **Generate HDL**.
2. For **Create HDL design files for synthesis**, select **Verilog**.

   You can leave the default settings for all other items.
3. Click **Generate** to generate files for synthesis.
4. Click **Finish** when the generation completes.

Related Links

What assignments do I need for a PCIe Gen1, Gen2 or Gen3 design that targets an Arria 10 ES2, ES3 or production device?

Starting with the Quartus Prime Software Release 17.0, these assignments are automatically included in the design. You do not have to add them.

3.1.4 Understanding the Files Generated

Table 9. Overview of Qsys Generation Output Files

Directory	Description
<*testbench_dir*>/<*variant_name*>/synth	Includes the top-level HDL file for the Hard IP for PCI Express.
<*testbench_dir*>/<*variant_name*>/sim/<*cad_vendor*>	Includes the HDL source files and scripts for the simulation testbench.

For a more detailed listing of the directories and files the Quartus Prime software generates, refer to *Files Generated for Intel IP Cores* in *Compiling the Design in the Qsys Design Flow*.

3.1.5 Understanding Simulation Log File Generation

Starting with the Quartus II 14.0 software release, simulation automatically creates a log file, `altpcie_monitor_<dev>_dlhip_tlp_file_log.log` in your simulation directory.

Table 10. Sample Simulation Log File Entries

Time	TLP Type	Payload (Bytes)	TLP Header
17989 RX	CfgRd0	0004	04000001_0000000F_01080008
17989 RX	MRd	0000	00000000_00000000_01080000
18021 RX	CfgRd0	0004	04000001_0000010F_0108002C
18053 RX	CfgRd0	0004	04000001_0000030F_0108003C
18085 RX	MRd	0000	00000000_00000000_0108000C

3.1.6 Understanding Physical Placement of the PCIe IP Core

For more information about physical placement of the PCIe blocks, refer to the links below. Contact your Intel sales representative for detailed information about channel and PLL usage.

Related Links

- Channel Placement and fPLL Usage for the Gen1 and Gen2 Data Rates on page 52
  For channel placement of x1, x2, x4, and x8 configurations.
- Channel Placement and fPLL and ATX PLL Usage for the Gen3 Data Rate on page 54
  For channel placement of x1, x2, x4, and x8 configurations.

3.1.7 Adding Virtual Pin Assignment to the Quartus II Settings File (.qsf)

To compile successfully you must add a virtual pin assignment statement for the PIPE interface to your `.qsf` file. The PIPE interface is useful for debugging, but is not a top-level interface of the IP core.

1. Browse to the synthesis directory that includes the `.qsf` for your project, `<project_dir>/ep_g1x8/`
2. Open `ep_g1x8.qsf`.
3. Add the following assignment statement:

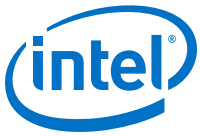
   ```
   set_instance_assignment -name VIRTUAL_PIN ON -to hip_pipe_*
   ```

4. Save the `.qsf` file.

3.1.8 Compiling the Design in the Qsys Design Flow

To compile the Qsys design example in the Quartus Prime software, you must create a Quartus Prime project and add your Qsys files to that project.

1. Before compiling, you can optionally turn on two parameters in the testbench. The first parameter specifies pin assignments that match those for the Intel Development Kit board I/Os. The second parameter enables the Compliance Base Board (CBB) logic on the development board. In the Gen1 x8 example design, complete the following steps if you want to enable these parameters:



a. Right-click the **APPS** component and select **Edit**.
b. Turn on **Enable FPGA Dev kit board I/Os**.
c. Turn on **Enable FPGA Dev kit board CBB logic**.
d. Click Finish.
e. On the Generate menu, select **Generate Testbench System** and then click **Generate**.
f. On the Generate menu, select **Generate HDL** and then click **Generate**. (You can use the same parameters that are specified in *Generating the Testbench* earlier in this chapter).

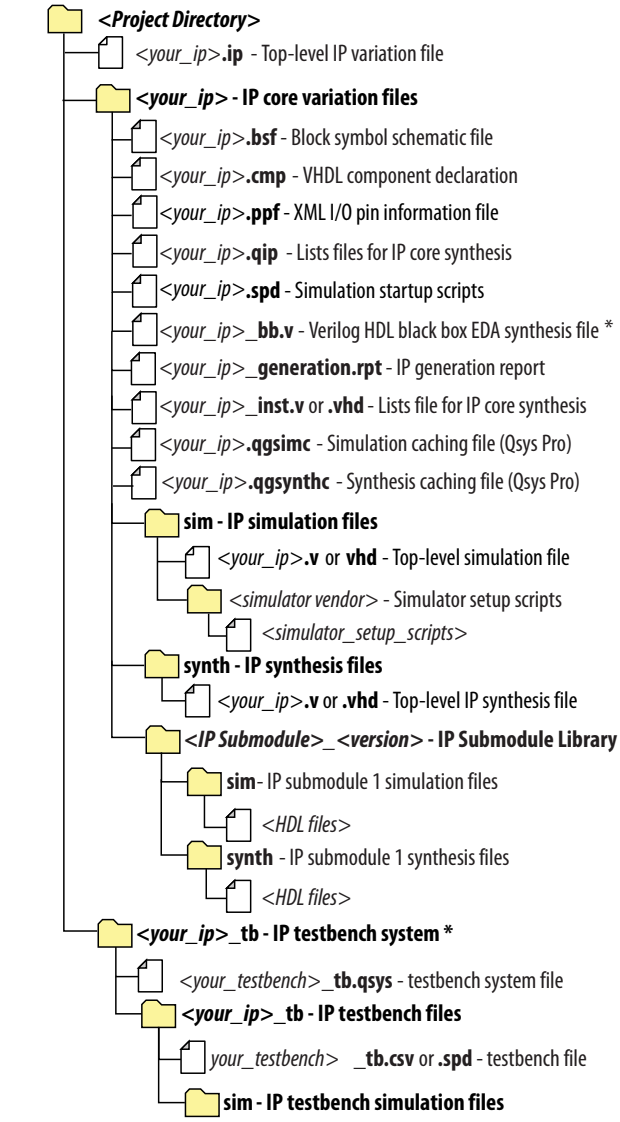
2. In the Quartus Prime software, click the **New Project Wizard** icon.
3. Click **Next** in the **New Project Wizard: Introduction** (The introduction does not appear if you previously turned it off.)
4. On the **Directory, Name, Top-Level Entity** page, enter the following information:
   a. The working directory shown is correct. You do not have to change it.
   b. For the project name, click the browse button browse to the synthesis directory that includes your Qsys project, `<working_dir>/ep_g1x8/synth` and click **Choose**. If the top-level design entity and Qsys system names are identical, theQuartus Prime software treats the Qsys system as the top-level design entity.
   c. For **What is the name of this project**, select your variant name `ep_g1x8`. Then click **Open**. If the top-level design entity and Qsys system names are identical, the Quartus Prime software treats the Qsys system as the top-level design entity.
   d. For **Project Type** select **Empty project**.
5. Click **Next** to display the **Add Files** page.
6. Complete the following steps to add the Quartus Prime IP File (**.qip**)to the project:
   a. Click the **browse** button. The **Select File** dialog box appears.
   b. Browse up one level to `<working_dir>/ep_g1x8/` button.
   c. In the **Files of type** list, select **IP Variation Files (*.qip)**.
   d. Click `ep_g1x8.qip` and then click **Open**.
   e. On the **Add Files** page, click **Add**.
7. Click **Next** to display the **Device** page.
8. On the **Family & Device Settings** page, choose the following target device family and options:
   a. In the **Family** list, select Arria 10 (GX/SX/GT).
   b. In the **Devices** list, select Arria 10 **All**.
   c. In the **Devices** list, select **All**.
   d. In the **Available devices** list, select the appropriate device. For Arria 10 GX FPGA Development Kit, select **10AX115S2F45I1SG**.
9. Click **Next** to close this page and display the **EDA Tool Settings** page.

10. From the **Simulation** list, select **ModelSim®**. From the **Format** list, select the HDL language you intend to use for simulation.
11. Click **Next** to display the **Summary** page.
12. Check the **Summary** page to ensure that you have entered all the information correctly.
13. Click **Finish** to create the Quartus Prime project.
14. Before compiling, you must assign I/O standards to the pins of the device. Refer to *Making Pin Assignments to Assign I/O Standard to Serial Data Pins* for instructions.
15. You must connect the `pin_perst` reset signal to the corresponding nPERST pin of the device. Refer to the definition of `pin_perst` in the *Reset, Status, and Link Training Signals* section for more information.
16. Next, set the value of the `test_in` bus to a value that is compatible for hardware testing. In Qsys design example provided, `test_in` is a top-level port.
    a. Comment out the `test_in` port in the top-level Verilog generated file.
    b. Add the following declaration, `wire[31:0] test_in`, to the same top-level Verilog file.
    c. Assign `hip_ctrl_test_in = 32'h188`.
    d. Connect `test_in` to `hip_ctrl_test_in`.

    Refer to the definition of `test_in` in the *Test Signals* section for more information about the bits of the `test_in` bus.
17. To compile your design using the Quartus Prime software, on the Processing menu, click **Start Compilation**. The Quartus Prime software then performs all the steps necessary to compile your design.

Files Generated for Intel IP Cores

Figure 16. IP Core Generated Files



* If supported and enabled for your IP core variation.

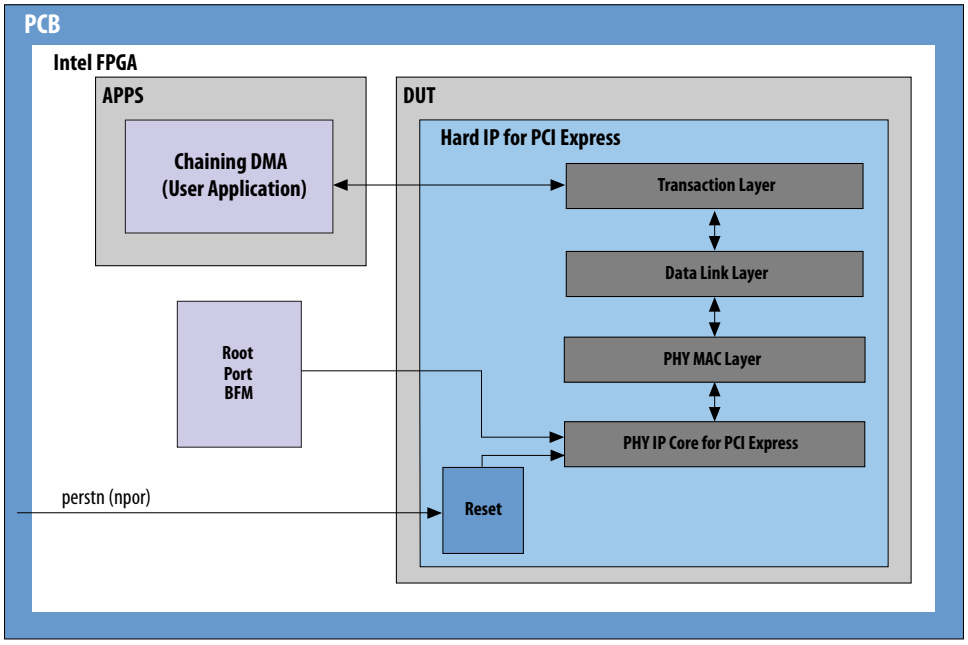
Related Links

- Making Pin Assignments to Assign I/O Standard to Serial Data Pins on page 165
  Before running Quartus Prime compilation, use the **Pin Planner** to assign I/O standards to the pins of the device.
- Test Signals on page 105
- Reset, Status, and Link Training Signals on page 82
- Generating the Testbench on page 25
- Simulating the Example Design on page 26

3.1.9 Modifying the Example Design

To use this example design as the basis of your own design, replace the Chaining DMA Example shown in the following figure with your own Application Layer design. Then modify the Root Port BFM driver to generate the transactions needed to test your Application Layer.

Figure 17. Testbench for PCI Express



3.1.10 Using the IP Catalog To Generate Your Arria 10 Hard IP for PCI Express as a Separate Component

You can also instantiate the Arria 10 Hard IP for PCI Express IP Core as a separate component for integration into your project.

You can use the Quartus Prime IP Catalog and IP Parameter Editor to select, customize, and generate files representing your custom IP variation. The IP Catalog (**Tools ➤ IP Catalog**) automatically displays IP cores available for your target device. Double-click any IP core name to launch the parameter editor and generate files representing your IP variation.

For more information about the customizing and generating IP Cores refer to *Specifying IP Core Parameters and Options* in *Introduction to Intel FPGA IP Cores*. For more information about upgrading older IP cores to the current release, refer to *Upgrading Outdated IP Cores* in *Introduction to Intel FPGA IP Cores*.

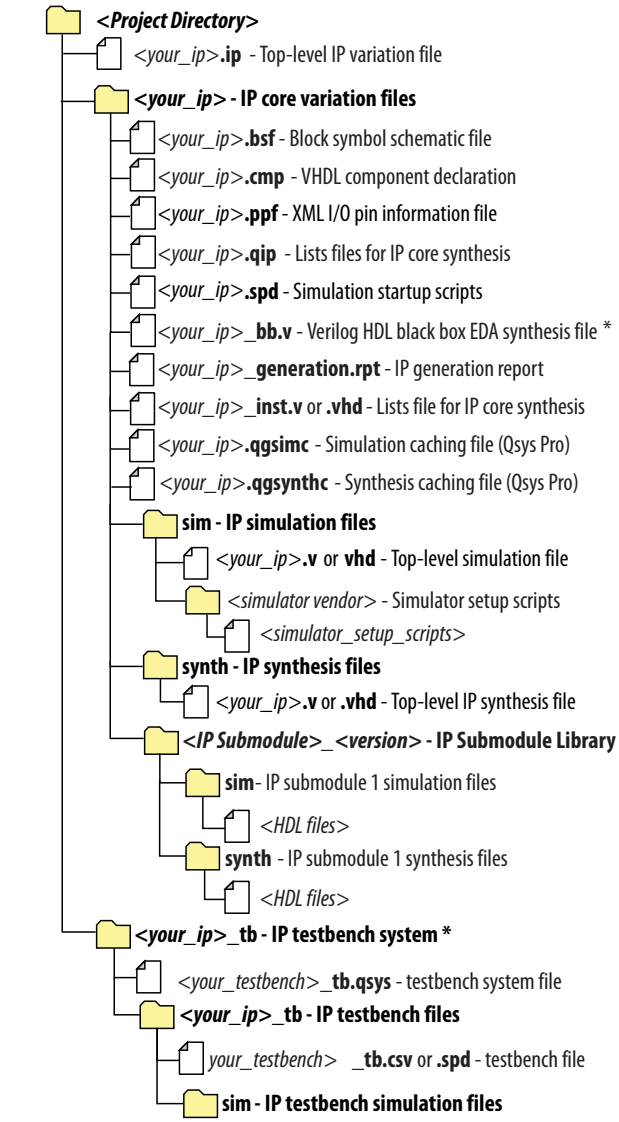
Related Links

Qsys Design Flow on page 25

3.1.11 IP Core Generation Output (Quartus Prime Pro Edition)

The Quartus Prime software generates the following output file structure for individual IP cores that are not part of a Qsys Pro system.

Figure 18. Individual IP Core Generation Output (Quartus Prime Pro Edition)



* If supported and enabled for your IP core variation.

Table 11. Files Generated for IP Cores

File Name	Description
<*your_ip*>.ip	Top-level IP variation file that contains the parameterization of an IP core in your project. If the IP variation is part of a Qsys Pro system, the parameter editor also generates a .qsys file.
<*your_ip*>.cmp	The VHDL Component Declaration (.cmp) file is a text file that contains local generic and port definitions that you use in VHDL design files.
<*your_ip*>_generation.rpt	IP or Qsys Pro generation log file. Displays a summary of the messages during IP generation.
<*your_ip*>.qgsimc (Qsys Pro systems only)	Simulation caching file that compares the .qsys and .ip files with the current parameterization of the Qsys Pro system and IP core. This comparison determines if Qsys Pro can skip regeneration of the HDL.
<*your_ip*>.qgsynth (Qsys Pro systems only)	Synthesis caching file that compares the .qsys and .ip files with the current parameterization of the Qsys Pro system and IP core. This comparison determines if Qsys Pro can skip regeneration of the HDL.
<*your_ip*>.qip	Contains all information to integrate and compile the IP component.
<*your_ip*>.csv	Contains information about the upgrade status of the IP component.
<*your_ip*>.bsf	A symbol representation of the IP variation for use in Block Diagram Files (.bdf).
<*your_ip*>.spd	Required input file for ip-make-simscript to generate simulation scripts for supported simulators. The .spd file contains a list of files you generate for simulation, along with information about memories that you initialize.
<*your_ip*>.ppf	The Pin Planner File (.ppf) stores the port and node assignments for IP components you create for use with the Pin Planner.
<*your_ip*>_bb.v	Use the Verilog blackbox (_bb.v) file as an empty module declaration for use as a blackbox.
<*your_ip*>_inst.v or _inst.vhd	HDL example instantiation template. Copy and paste the contents of this file into your HDL file to instantiate the IP variation.
<*your_ip*>.regmap	If the IP contains register information, the Quartus Prime software generates the .regmap file. The .regmap file describes the register map information of master and slave interfaces. This file complements the .sopcinfo file by providing more detailed register information about the system. This file enables register display views and user customizable statistics in System Console.
<*your_ip*>.svd	Allows HPS System Debug tools to view the register maps of peripherals that connect to HPS within a Qsys Pro system. During synthesis, the Quartus Prime software stores the .svd files for slave interface visible to the System Console masters in the .sof file in the debug session. System Console reads this section, which Qsys Pro queries for register map information. For system slaves, Qsys Pro accesses the registers by name.
<*your_ip*>.v <*your_ip*>.vhd	HDL files that instantiate each submodule or child IP core for synthesis or simulation.
mentor/	Contains a script msim_setup.tcl to set up and run a simulation.
aldec/	Contains a Riviera*-PRO script rivierapro_setup.tcl to setup and run a simulation.
/synopsys/vcs /synopsys/vcsmx	Contains a shell script vcs_setup.sh to set up and run a VCS* simulation. Contains a shell script vcsmx_setup.sh and synopsys_sim.setup file to set up and run a VCS MX* simulation.

continued...

File Name	Description
/cadence	Contains a shell script ncsim_setup.sh and other setup files to set up and run an NCSIM simulation.
/submodules	Contains HDL files for the IP core submodule.
<*IP submodule*>/	For each generated IP submodule directory, Qsys Pro generates /synth and /sim sub-directories.

4 Arria 10 Parameter Settings

4.1 Parameters

This chapter provides a reference for all the parameters of the Arria 10 Hard IP for PCI Express IP core.

Table 12. System Settings

Parameter	Value	Description
Application Interface Type	**Avalon-ST** **Avalon-MM** **Avalon-MM with DMA** **Avalon-ST with SR-IOV**	Selects the interface to the Application Layer.
Hard IP mode	**Gen3x8, Interface: 256-bit, 250 MHz** **Gen3x4, Interface: 256-bit, 125 MHz** **Gen3x4, Interface: 128-bit, 250 MHz** **Gen3x2, Interface: 128-bit, 125 MHz** **Gen3x2, Interface: 64-bit, 250 MHz** **Gen3x1, Interface: 64-bit, 125 MHz** **Gen2x8, Interface: 256-bit, 125 MHz** **Gen2x8, Interface: 128-bit, 250 MHz** **Gen2x4, Interface: 128-bit, 125 MHzGen2x2, Interface: 64-bit, 125 MHz Gen2x4, Interface: 64-bit, 250 MHz Gen2x1, Interface: 64-bit, 125 MHz Gen1x8, Interface: 128-bit, 125 MHz Gen1x8, Interface: 64-bit, 250 MHz Gen1x4, Interface: 64-bit, 125 MHz Gen1x2, Interface: 64-bit, 125 MHz Gen1x1, Interface: 64-bit, 125 MHz Gen1x1, Interface: 64-bit, 62.5 MHz**	Selects the following elements: • The lane data rate. Gen1, Gen2, and Gen3 are supported • The width of the data interface between the hard IP Transaction Layer and the Application Layer implemented in the FPGA fabric • The Application Layer interface frequency The interface supports only the 256-bit modes.
Port type	**Native Endpoint** **Root Port**	Specifies the port type. The Endpoint stores parameters in the Type 0 Configuration Space. The Root Port stores parameters in the Type 1 Configuration Space. The interface supports only **Native Endpoint** operation.
RX Buffer credit allocation - performance for received requests	**Minimum** **Low** **Balanced** **High** **Maximum**	Determines the allocation of posted header credits, posted data credits, non-posted header credits, completion header credits, and completion data credits in the 16 KB RX buffer. The settings allow you to adjust the credit allocation to optimize your system. The credit allocation for the selected setting displays in the **Message** pane. The **Message** pane dynamically updates the number of credits for Posted, Non-Posted Headers and Data, and Completion Headers and Data as you change this selection. Refer to the *Throughput Optimization* chapter for more information about optimizing your design.

continued...

Parameter	Value	Description
		Refer to the *RX Buffer Allocation Selections Available by Interface Type* below for the availability of these settings by interface type. **Minimum**—configures the minimum PCIe specification allowed for non-posted and posted request credits, leaving most of the RX Buffer space for received completion header and data. Select this option for variations where application logic generates many read requests and only infrequently receives single requests from the PCIe link. **Low**—configures a slightly larger amount of RX Buffer space for non-posted and posted request credits, but still dedicates most of the space for received completion header and data. Select this option for variations where application logic generates many read requests and infrequently receives small bursts of requests from the PCIe link. This option is recommended for typical endpoint applications where most of the PCIe traffic is generated by a DMA engine that is located in the endpoint application layer logic. **Balanced**—configures approximately half the RX Buffer space to received requests and the other half of the RX Buffer space to received completions. Select this option for variations where the received requests and received completions are roughly equal. **High**—configures most of the RX Buffer space for received requests and allocates a slightly larger than minimum amount of space for received completions. Select this option where most of the PCIe requests are generated by the other end of the PCIe link and the local application layer logic only infrequently generates a small burst of read requests. This option is recommended for typical root port applications where most of the PCIe traffic is generated by DMA engines located in the endpoints. **Maximum**—configures the minimum PCIe specification allowed amount of completion space, leaving most of the RX Buffer space for received requests. Select this option when most of the PCIe requests are generated by the other end of the PCIe link and the local application layer logic never or only infrequently generates single read requests. This option is recommended for control and status endpoint applications that don't generate any PCIe requests of their own and only are the target of write and read requests from the root complex.
RX Buffer completion credits	**Header credits, Data credits**	Displays the number of completion credits in the 16 KB RX buffer resulting from the credit allocation parameter. Each header credit is 16 bytes. Each data credit is 20 bytes.

Related Links

PCI Express Base Specification 3.0

4.2 Arria 10 Avalon-ST Settings

Table 13. System Settings for PCI Express

Parameter	Value	Description
Enable Avalon-ST reset output port	**On/Off**	When **On**, the generated reset output port has the same functionality that the `reset_status` port included in the Reset and Link Status interface.
Enable byte parity ports on Avalon-ST interface	**On/Off**	When **On**, the RX and TX datapaths are parity protected. Parity is odd. The Application Layer must provide valid byte parity in the Avalon-ST TX direction. This parameter is only available for the Avalon-ST Arria 10 Hard IP for PCI Express.
Enable multiple packets per cycle for the 256-bit interface	**On/Off**	When **On**, the 256-bit Avalon-ST interface supports the transmission of TLPs starting at any 128-bit address boundary, allowing support for multiple packets in a single cycle. To support multiple packets per cycle, the Avalon-ST interface includes 2 start of packet and end of packet signals for the 256-bit Avalon-ST interfaces. This is not supported for the Avalon-ST with SR-IOV interface.
Enable credit consumed selection port	**On/Off**	When you turn on this option, the core includes the `tx_cons_cred_sel` port. This parameter does not apply to the Avalon-MM interface.
Enable Configuration bypass (CfgBP)	**On/Off**	When **On**, the Arria 10 Hard IP for PCI Express bypasses the Transaction Layer Configuration Space registers included as part of the Hard IP, allowing you to substitute a custom Configuration Space implemented in soft logic. This parameter is not available for the Avalon-MM IP Cores.
Enable local management interface (LMI)	**On/Off**	When **On**, your variant includes the optional LMI interface. This interface is used to log error descriptor information in the TLP header log registers. The LMI interface provides the same access to Configuration Space registers as Configuration TLP requests.

Related Links

- Throughput Optimization on page 161
- PCI Express Base Specification 3.0

4.3 Base Address Register (BAR) and Expansion ROM Settings

The type and size of BARs available depend on port type.

Table 14. BAR Registers

Parameter	Value	Description
Type	**Disabled** **64-bit prefetchable memory** **32-bit non-prefetchable memory** **32-bit prefetchable memory** **I/O address space**	If you select 64-bit prefetchable memory, 2 contiguous BARs are combined to form a 64-bit prefetchable BAR; you must set the higher numbered BAR to **Disabled**. A non-prefetchable 64-bit BAR is not supported because in a typical system, the Root Port Type 1 Configuration Space sets the maximum non-prefetchable memory window to 32 bits. The BARs can also be configured as separate 32-bit memories. Defining memory as prefetchable allows contiguous data to be fetched ahead. Prefetching memory is advantageous when the requestor may require more data from the same

continued...

Parameter	Value	Description
		region than was originally requested. If you specify that a memory is prefetchable, it must have the following 2 attributes: • Reads do not have side effects such as changing the value of the data read • Write merging is allowed The **32-bit prefetchable memory** and **I/O address space** BARs are only available for the **Legacy Endpoint**.
Size	**16 Bytes–8 EB**	Supports the following memory sizes: • 128 bytes–2 GB or 8 EB: **Endpoint** and **Root Port** variants • 6 bytes–4 KB: **Legacy Endpoint** variants
Expansion ROM	**Disabled–16 MB**	Specifies the size of the optional ROM. The expansion ROM is only available for the Avalon-ST interface.

4.4 Base and Limit Registers for Root Ports

Table 15. Base and Limit Registers

The following table describes the `Base` and `Limit` registers which are available in the Type 1 Configuration Space for Root Ports. These registers are used for TLP routing and specify the address ranges assigned to components that are downstream of the Root Port or bridge.

Parameter	Value	Description
Input/Output	**Disabled** **16-bit I/O addressing** **32-bit I/O addressing**	Specifies the address widths for the `IO base` and `IO limit` registers.
Prefetchable memory	**Disabled** **16-bit memory addressing** **32-bit memory addressing**	Specifies the address widths for the `Prefetchable Memory Base` register and `Prefetchable Memory Limit` register.

Note: The Avalon-MM Hard IP for PCI Express Root Port does not filter addresses.

Related Links

PCI to PCI Bridge Architecture Specification

4.5 Device Identification Registers

Table 16. Device ID Registers

The following table lists the default values of the read-only Device ID registers. You can use the parameter editor to change the values of these registers. At run time, you can change the values of these registers using the optional reconfiguration block signals. You can specify Device ID registers for each Physical Function.

Register Name	Default Value	Description
Vendor ID	0x00001172	Sets the read-only value of the `Vendor ID` register. This parameter can not be set to 0xFFFF per the PCI Express Specification. Address offset: 0x000.
Device ID	Custom value	Sets the read-only value of the `Device ID` register. Address offset: 0x000.
Revision ID	Custom value	Sets the read-only value of the `Revision ID` register.

continued...

Register Name	Default Value	Description
		Address offset: 0x008.
Class code	Custom value	Sets the read-only value of the `Class Code` register. Address offset: 0x008.
Subsystem Vendor ID	Custom value	Sets the read-only value of the ` register in the PCI Type 0 Configuration Space. This parameter cannot be set to 0xFFFF per the *PCI Express Base Specification*. This value is assigned by PCI-SIG to the device manufacturer. Address offset: 0x02C.
Subsystem Device ID	Custom value	Sets the read-only value of the `Subsystem Device ID` register in the PCI Type 0 Configuration Space. Address offset: 0x02C

Related Links

PCI Express Base Specification 2.1 or 3.0

4.6 PCI Express and PCI Capabilities Parameters

This group of parameters defines various capability properties of the IP core. Some of these parameters are stored in the PCI Configuration Space - PCI Compatible Configuration Space. The byte offset indicates the parameter address.

4.6.1 PCI Express and PCI Capabilities

Table 17. Capabilities Registers

Parameter	Possible Values	Default Value	Description
Maximum payload size	**128 bytes** **256 bytes** **512 bytes** **1024 bytes** **2048 bytes**	128 bytes	Specifies the maximum payload size supported. This parameter sets the read-only value of the max payload size supported field of the Device Capabilities register (0x084[2:0]). Address: 0x084**.**
Number of Tags supported	**32** **64**	32	Indicates the number of tags supported for non-posted requests transmitted by the Application Layer. This parameter sets the values in the Device Control register (0x088) of the PCI Express capability structure described in Table 9–9 on page 9–5. The Transaction Layer tracks all outstanding completions for non-posted requests made by the Application Layer. This parameter configures the Transaction Layer for the maximum number of **Tags supported** to track. The Application Layer must set the tag values in all non-posted PCI Express headers to be less than this value. Values greater than 32 also set the extended tag field supported bit in the Configuration Space Device Capabilities register. The Application Layer can only use tag numbers greater than 31 if configuration software sets the Extended Tag Field Enable bit of the Device Control register. This bit is available to the Application Layer on the `tl_cfg_ctl` output signal as `cfg_devcsr[8]`.
Completion timeout range	**ABCD** **BCD** **ABC** **AB** **B** **A**	ABCD	Indicates device function support for the optional completion timeout programmability mechanism. This mechanism allows system software to modify the completion timeout value. This field is applicable only to Root Ports and Endpoints that issue requests on their own behalf. Completion timeouts are specified and enabled in the Device Control 2 register (0x0A8) of the *PCI*

continued...

Parameter	Possible Values	Default Value	Description
	None		*Express Capability Structure Version*. For all other functions this field is reserved and must be hardwired to 0x0000b. Four time value ranges are defined: • Range A: 50 us to 10 ms • Range B: 10 ms to 250 ms • Range C: 250 ms to 4 s • Range D: 4 s to 64 s Bits are set to show timeout value ranges supported. The function must implement a timeout value in the range 50 s to 50 ms. The following values specify the range: • None—Completion timeout programming is not supported • 0001 Range A • 0010 Range B • 0011 Ranges A and B • 0110 Ranges B and C • 0111 Ranges A, B, and C • 1110 Ranges B, C and D • 1111 Ranges A, B, C, and D All other values are reserved. Intel recommends that the completion timeout mechanism expire in no less than 10 ms.
Disable completion timeout	**On/Off**	On	Disables the completion timeout mechanism. When **On**, the core supports the completion timeout disable mechanism via the PCI Express `Device Control Register 2`. The Application Layer logic must implement the actual completion timeout mechanism for the required ranges.

4.6.2 Error Reporting

Table 18. Error Reporting

Parameter	Value	Default Value	Description
Enable Advanced Error Reporting (AER)	**On/Off**	Off	When **On**, enables the Advanced Error Reporting (AER) capability.
Enable ECRC checking	**On/Off**	Off	When **On**, enables ECRC checking. Sets the read-only value of the ECRC check capable bit in the `Advanced Error Capabilities and Control Register`. This parameter requires you to enable the AER capability.
Enable ECRC generation	**On/Off**	Off	When **On**, enables ECRC generation capability. Sets the read-only value of the ECRC generation capable bit in the `Advanced Error Capabilities and Control Register`. This parameter requires you to enable the AER capability.
Enable ECRC forwarding on the Avalon-ST interface	**On/Off**	Off	When **On**, enables ECRC forwarding to the Application Layer. On the Avalon-ST RX path, the incoming TLP contains the ECRC dword(1) and the TD bit is set if an ECRC exists. On the transmit the TLP from the Application Layer must contain the ECRC dword and have the TD bit set.
Track RX completion buffer	**On/Off**	Off	When **On**, the core includes the `rxfc_cplbuf_ovf` output status signal to track the RX posted completion buffer overflow status.

continued...

Parameter	Value	Default Value	Description
overflow on the Avalon-ST interface			

Note:
1. Throughout this user guide, the terms word, dword and qword have the same meaning that they have in the *PCI Express Base Specification*. A word is 16 bits, a dword is 32 bits, and a qword is 64 bits.

4.6.3 Link Capabilities

Table 19. Link Capabilities

Parameter	Value	Description
Link port number (Root Port only)	**0x01**	Sets the read-only value of the port number field in the `Link Capabilities` register. This parameter is for Root Ports only. It should not be changed.
Data link layer active reporting (Root Port only)	**On/Off**	Turn **On** this parameter for a Root Port, if the attached Endpoint supports the optional capability of reporting the DL_Active state of the Data Link Control and Management State Machine. For a hot-plug capable Endpoint (as indicated by the `Hot Plug Capable` field of the `Slot Capabilities` register), this parameter must be turned **On**. For Root Port components that do not support this optional capability, turn **Off** this option.
Surprise down reporting (Root Port only)	**On/Off**	When your turn this option **On**, an Endpoint supports the optional capability of detecting and reporting the surprise down error condition. The error condition is read from the Root Port.
Slot clock configuration	**On/Off**	When you turn this option **On**, indicates that the Endpoint or Root Port uses the same physical reference clock that the system provides on the connector. When **Off**, the IP core uses an independent clock regardless of the presence of a reference clock on the connector. This parameter sets the Slot Clock Configuration bit (bit 12) in the `PCI Express Link Status` register.

4.6.4 MSI and MSI-X Capabilities

Table 20. MSI and MSI-X Capabilities

Parameter	Value	Description
MSI messages requested	**1, 2, 4, 8, 16, 32**	Specifies the number of messages the Application Layer can request. Sets the value of the `Multiple Message Capable` field of the `Message Control` register, Address: 0x050[31:16].
MSI-X Capabilities		
Implement MSI-X	**On/Off**	When **On**, adds the MSI-X functionality.
	Bit Range	
Table size	[15:0]	System software reads this field to determine the MSI-X Table size <*n*>, which is encoded as <*n–1*>. For example, a returned value of 2047 indicates a table size of 2048. This field is read-only. Legal range is 0–2047 (2^{16}). Address offset: 0x068[26:16]
Table offset	[31:0]	Points to the base of the MSI-X Table. The lower 3 bits of the table BAR indicator (BIR) are set to zero by software to form a 64-bit qword-aligned offset. This field is read-only.

continued...

Parameter	Value	Description
Table BAR indicator	[2:0]	Specifies which one of a function's BARs, located beginning at 0x10 in Configuration Space, is used to map the MSI-X table into memory space. This field is read-only. Legal range is 0–5.
Pending bit array (PBA) offset	[31:0]	Used as an offset from the address contained in one of the function's Base Address registers to point to the base of the MSI-X PBA. The lower 3 bits of the PBA BIR are set to zero by software to form a 32-bit qword-aligned offset. This field is read-only. [2]
Pending BAR indicator	[2:0]	Specifies the function Base Address registers, located beginning at 0x10 in Configuration Space, that maps the MSI-X PBA into memory space. This field is read-only. Legal range is 0–5.

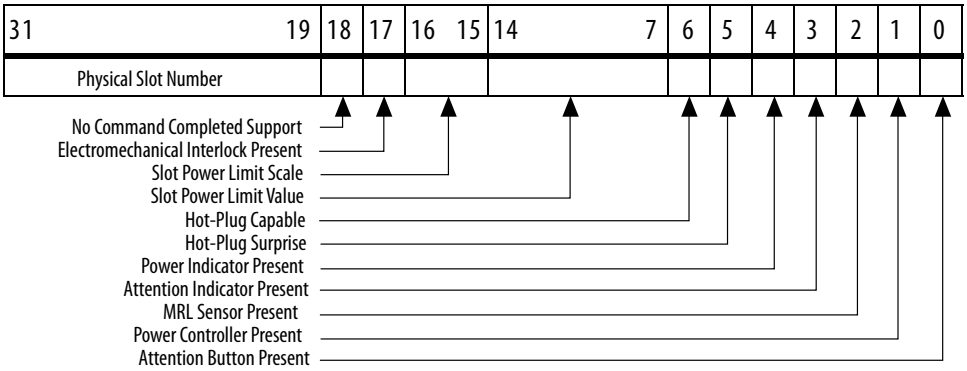
4.6.5 Slot Capabilities

Table 21. Slot Capabilities

Parameter	Value	Description
Use Slot register	**On/Off**	This parameter is only supported in Root Port mode. The slot capability is required for Root Ports if a slot is implemented on the port. Slot status is recorded in the `PCI Express Capabilities` register. Defines the characteristics of the slot. You turn on this option by selecting **Enable slot capability**. Refer to the figure below for bit definitions.
Slot power scale	`0–3`	Specifies the scale used for the **Slot power limit**. The following coefficients are defined: • 0 = 1.0x • 1 = 0.1x • 2 = 0.01x • 3 = 0.001x The default value prior to hardware and firmware initialization is b'00. Writes to this register also cause the port to send the `Set_Slot_Power_Limit` Message. Refer to Section 6.9 of the *PCI Express Base Specification Revision* for more information.
Slot power limit	`0–255`	In combination with the **Slot power scale value**, specifies the upper limit in watts on power supplied by the slot. Refer to Section 7.8.9 of the *PCI Express Base Specification* for more information.
Slot number	`0-8191`	Specifies the slot number.

2 Throughout this user guide, the terms word, dword and qword have the same meaning that they have in the *PCI Express Base Specification*. A word is 16 bits, a dword is 32 bits, and a qword is 64 bits.

Figure 19. Slot Capability



4.6.6 Power Management

Table 22. Power Management Parameters

Parameter	Value	Description
Endpoint L0s acceptable latency	**Maximum of 64 ns** **Maximum of 128 ns** **Maximum of 256 ns** **Maximum of 512 ns** **Maximum of 1 us** **Maximum of 2 us** **Maximum of 4 us** **No limit**	This design parameter specifies the maximum acceptable latency that the device can tolerate to exit the L0s state for any links between the device and the root complex. It sets the read-only value of the Endpoint L0s acceptable latency field of the `Device Capabilities Register` (0x084). This Endpoint does not support the L0s or L1 states. However, in a switched system there may be links connected to switches that have L0s and L1 enabled. This parameter is set to allow system configuration software to read the acceptable latencies for all devices in the system and the exit latencies for each link to determine which links can enable Active State Power Management (ASPM). This setting is disabled for Root Ports. The default value of this parameter is 64 ns. This is the safest setting for most designs.
Endpoint L1 acceptable latency	**Maximum of 1 us** **Maximum of 2 us** **Maximum of 4 us** **Maximum of 8 us** **Maximum of 16 us** **Maximum of 32 us** **No limit**	This value indicates the acceptable latency that an Endpoint can withstand in the transition from the L1 to L0 state. It is an indirect measure of the Endpoint's internal buffering. It sets the read-only value of the Endpoint L1 acceptable latency field of the `Device Capabilities Register`. This Endpoint does not support the L0s or L1 states. However, a switched system may include links connected to switches that have L0s and L1 enabled. This parameter is set to allow system configuration software to read the acceptable latencies for all devices in the system and the exit latencies for each link to determine which links can enable Active State Power Management (ASPM). This setting is disabled for Root Ports. The default value of this parameter is 1 µs. This is the safest setting for most designs.

4.7 Vendor Specific Extended Capability (VSEC)

Table 23. VSEC

Parameter	Value	Description
Vendor Specific Extended Capability (VSEC) ID:	0x00001172	Sets the read-only value of the 16-bit User ID register from the Vendor Specific Extended Capability.
Vendor Specific Extended Capability (VSEC) Revision:	0x00000000	Sets the read-only value of the 4-bit VSEC Revision register from the Vendor Specific Extended Capability.
User Device or Board Type ID register from the Vendor Specific Extended Capability:	0x00000000	Sets the read-only value of the 16-bit Device or Board Type ID register from the Vendor Specific Extended Capability.

4.8 Configuration, Debug, and Extension Options

Table 24. System Settings for PCI Express

Parameter	Value	Description
Enable configuration via Protocol (CvP)	**On/Off**	When **On**, the Quartus Prime software places the Endpoint in the location required for configuration via protocol (CvP). For more information about CvP, click the *Configuration via Protocol (CvP)* link below.
Enable dynamic reconfiguration of PCIe read-only registers	**On/Off**	When **On**, you can use the Hard IP reconfiguration bus to dynamically reconfigure Hard IP read-only registers. For more information refer to *Hard IP Reconfiguration Interface*.
Enable transceiver dynamic reconfiguration	**On/Off**	When on, creates an Avalon-MM slave interface that software can drive to update transceiver registers.
Enable Altera Debug Master Endpoint (ADME)	**On/Off**	When **On**, an embedded Altera Debug Master Endpoint connects internally to the Avalon-MM slave interface for dynamic reconfiguration. The ADME can access the reconfiguration space of the transceiver. It uses JTAG via the System Console to run tests and debug functions.
Enable Arria 10 FPGA Development Kit connection	**On/Off**	When **On**, add control and status conduit interface to the top level variant, to be connected a PCIe Development Kit component.

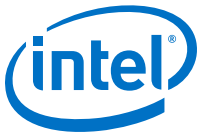
Related Links

Configuration over Protocol (CvP) on page 168

4.9 PHY Characteristics

Table 25. PHY Characteristics

Parameter	Value	Description
Gen2 TX de-emphasis	**3.5dB** **6dB**	Specifies the transmit de-emphasis for Gen2. Intel recommends the following settings: • 3.5dB: Short PCB traces • 6.0dB: Long PCB traces.
Requested equalization far-end TX preset	**Preset0-Preset9**	Specifies the requested TX preset for Phase 2 and 3 far-end transmitter. The default value **Preset8** provides the best signal quality for most designs.
Enable soft DFE controller IP	**On** **Off**	When **On**, the PCIe Hard IP core includes a decision feedback equalization (DFE) soft controller in the FPGA fabric to improve the bit error rate (BER) margin. The default for this option is **Off** because the DFE controller is typically not required. However, short reflective links may benefit from this soft DFE controller IP. This parameter is available only for Gen3 mode. It is not supported when CvP or autonomous modes are enabled.



4.10 Arria 10 Example Designs

Table 26. Example Designs

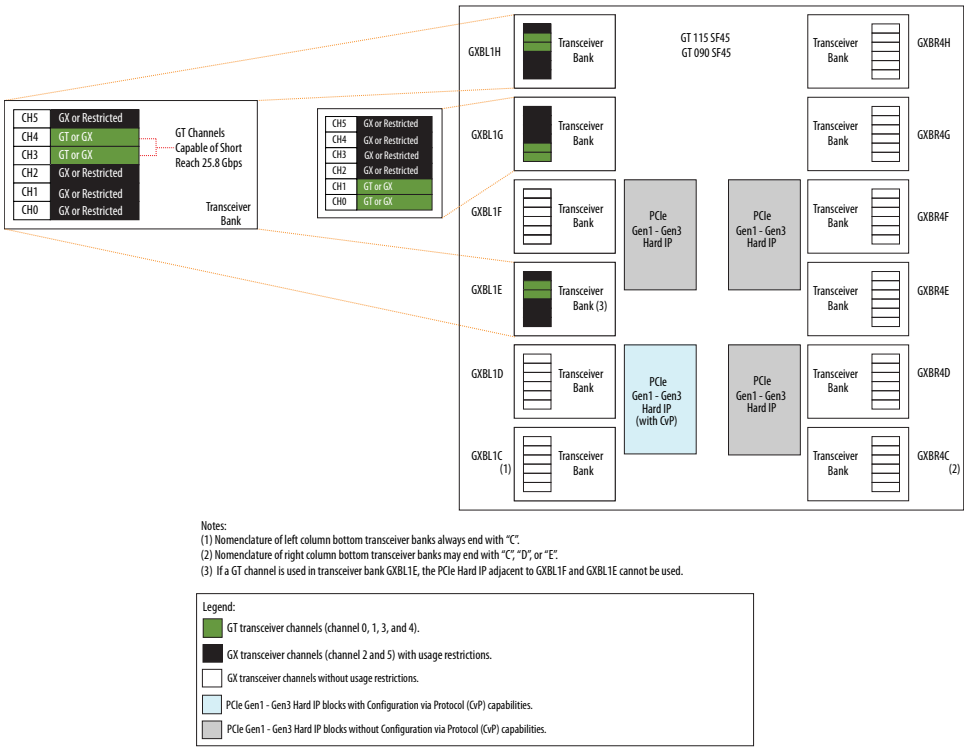
Parameter	Value	Description
Available Example Designs	**DMA** **PIO**	When you select the **DMA** option, the generated example design includes a direct memory access application. This application includes upstream and downstream transactions. When you select the **PIO** option, the generated design includes a target application including only downstream transactions.
Simulation	**On/Off**	When **On**, the generated output includes a simulation model.
Synthesis	**On/Off**	When **On**, the generated output includes a synthesis model.
Generated HDL format	**Verilog**	Only Verilog HDL is supported.
Target Development Kit	**Arria 10 GX FPGA Development Kit** **Arria 10 GX FPGA Development Kit ES2** **None**	Select **Arria 10 FPGA Development Kit** for Arria 10 production devices. Select **Arria 10 FPGA Development Kit ES** for engineering sample (ES) or ES2 devices. Select **None** if you are targeting your own development board.

5 Physical Layout of Hard IP In Arria 10 Devices

Arria 10 devices include 1–4 hard IP blocks for PCI Express. The bottom left hard IP block includes the CvP functionality for flip chip packages. For other package types, the CvP functionality is in the bottom right block.

Note: Arria 10 devices do not support configurations that configure a bottom (left or right) hard IP block with a Gen3 x4 or Gen3 x8 IP core and also configure the top hard IP block on the same side with a Gen3 x1 or Gen3 x2 IP core variation.

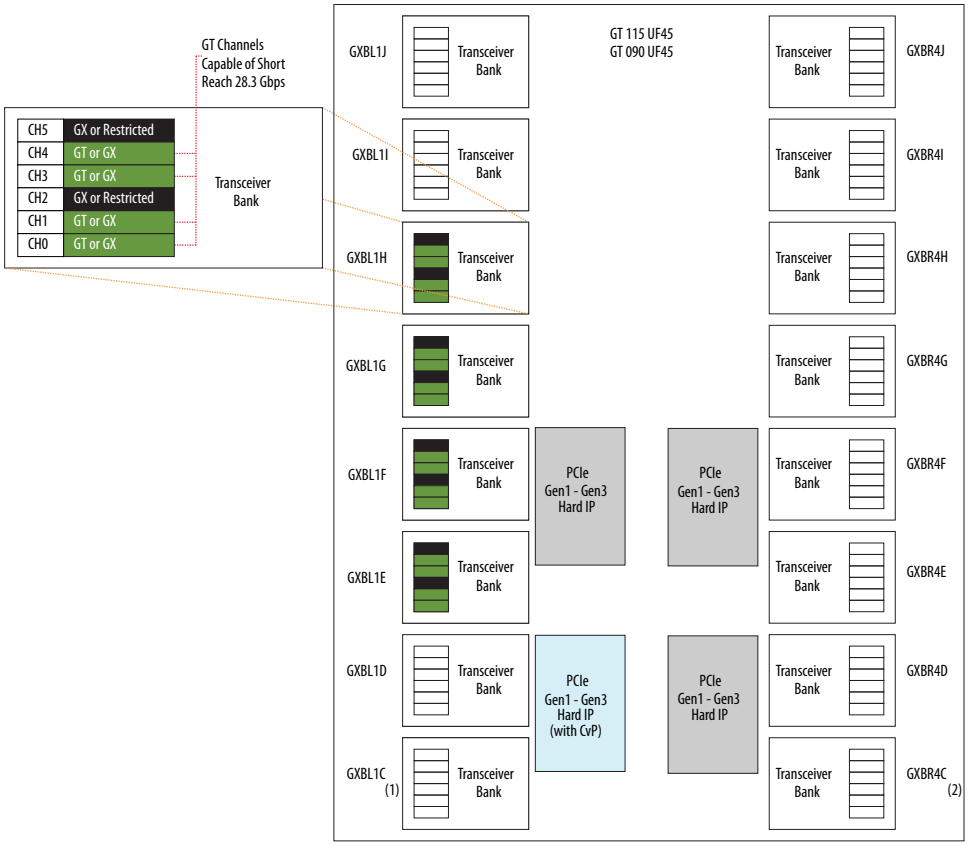
Figure 20. Arria 10 Devices with 72 Transceiver Channels and Four PCIe Hard IP Blocks



Notes:
(1) Nomenclature of left column bottom transceiver banks always end with "C".
(2) Nomenclature of right column bottom transceiver banks may end with "C", "D", or "E".
(3) If a GT channel is used in transceiver bank GXBL1E, the PCIe Hard IP adjacent to GXBL1F and GXBL1E cannot be used.

Legend:
- GT transceiver channels (channel 0, 1, 3, and 4).
- GX transceiver channels (channel 2 and 5) with usage restrictions.
- GX transceiver channels without usage restrictions.
- PCIe Gen1 - Gen3 Hard IP blocks with Configuration via Protocol (CvP) capabilities.
- PCIe Gen1 - Gen3 Hard IP blocks without Configuration via Protocol (CvP) capabilities.

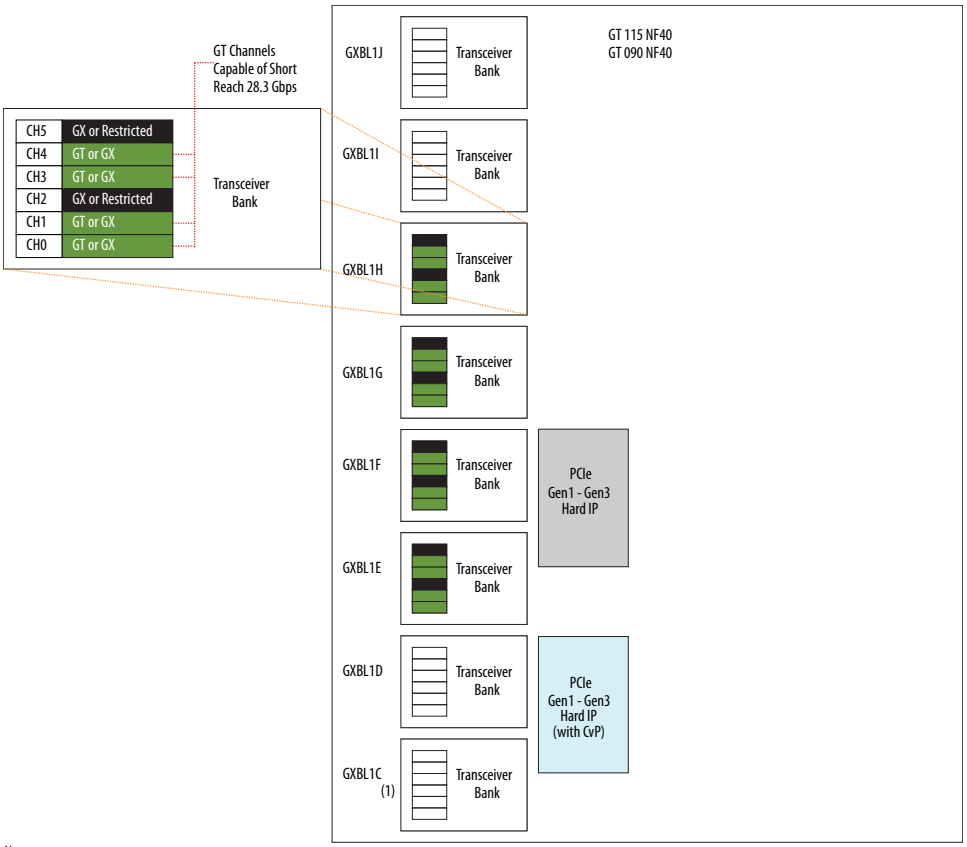
Figure 21. Arria 10 Devices with 96 Transceiver Channels and Four PCIe Hard IP Blocks



Notes:
(1) Nomenclature of left column bottom transceiver banks always ends with "C".
(2) Nomenclature of right column bottom transceiver banks may end with "C", "D", or "E".

Legend:
- GT transceiver channels (channel 0, 1, 3, and 4)
- GX transceiver channels (channel 2 and 5) with usage restrictions.
- GX transceiver channels without usage restrictions.
- PCIe Gen1 - Gen3 Hard IP blocks with Configuration via Protocol (CvP) capabilities.
- PCIe Gen1 - Gen3 Hard IP blocks without Configuration via Protocol (CvP) capabilities.

Figure 22. Arria 10 GT Devices with 48 Transceiver Channels and Two PCIe Hard IP Blocks



Refer to the *Arria 10 Transceiver Layout* in the *Intel FPGA Arria 10 Transceiver PHY User Guide* for comprehensive figures for Arria 10 GT, GX, and SX devices.

Related Links

Intel FPGA Arria 10 Transceiver PHY IP Core User Guide
For information about the transceiver physical (PHY) layer architecture, PLLs, clock networks, and transceiver PHY IP.

5.1 Channel and Pin Placement for the Gen1, Gen2, and Gen3 Data Rates

The following figures illustrate the x1, x2, x4, and x8 channel and pin placements for the Arria 10 Hard IP for PCI Express.

In these figures, channels that are not used for the PCI Express protocol are available for other protocols. Unused channels are shown in gray.

Note: In all configurations, physical channel 4 in the PCS connects to logical channel 0 in the hard IP. You cannot change the channel placements illustrated below.

For the possible values of <txvr_block_N> and <txvr_block_N+1>, refer to the figures that show the physical location of the Hard IP PCIe blocks in the different types of Arria 10 devices, at the start of this chapter. For each HIP block, the transceiver block that is adjacent and extends below the HIP block, is <txvr_block_N>, and the transceiver block that is directly above <txvr_block_N> is <txvr_block_N+1>. For example, in an Arria 10 device with 96 transceiver channels and four PCIe HIP blocks, if your design uses the HIP block that supports CvP, <txvr_block_N> is GXB1C and <txvr_block_N+1> is GXB1D.

Figure 23. Arria 10 Gen1, Gen2, and Gen3 x1 Channel and Pin Placement

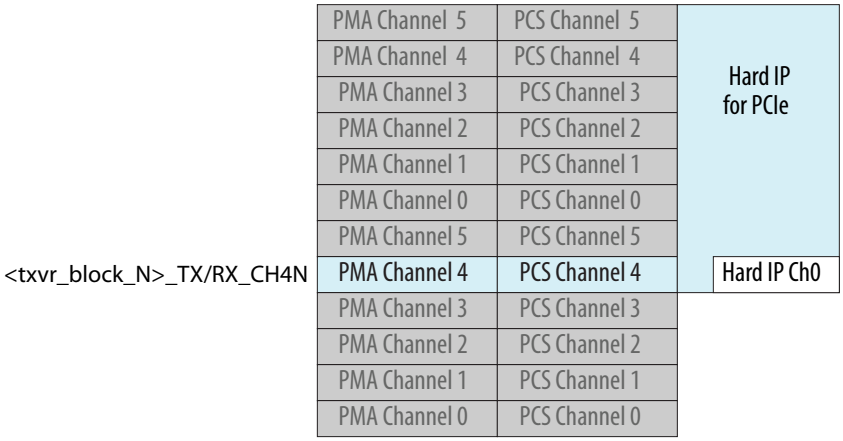


Figure 24. Arria 10 Gen1 Gen2, and Gen3 x2 Channel and Pin Placement

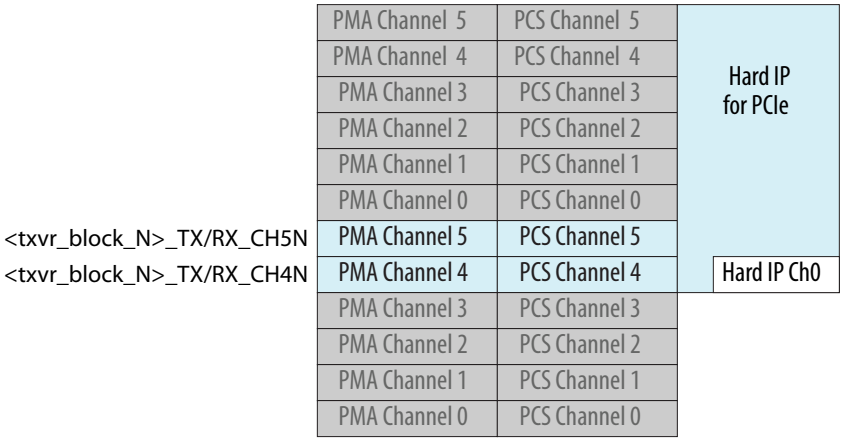


Figure 25. Arria 10 Gen1, Gen2, and Gen3 x4 Channel and Pin Placement

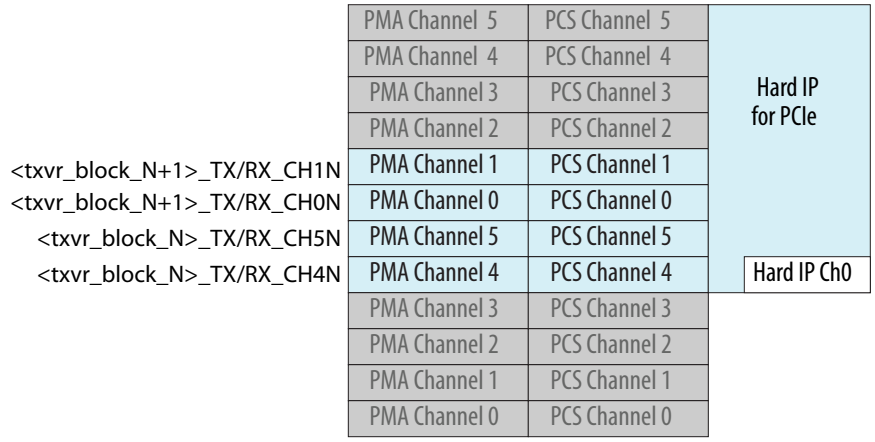
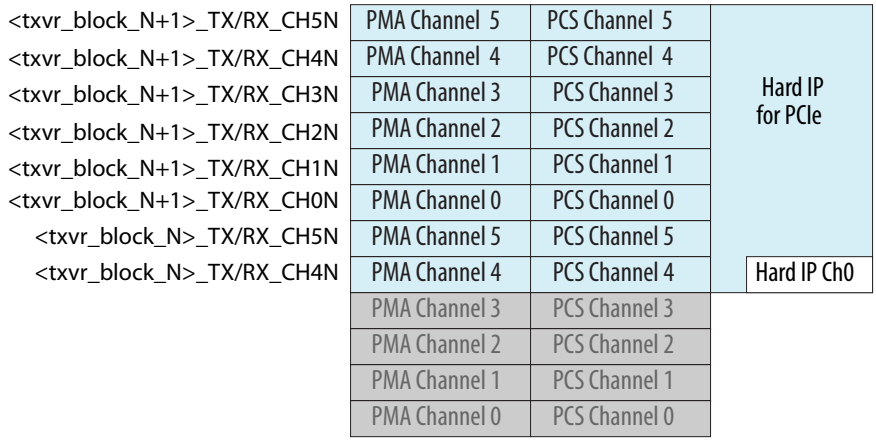


Figure 26. Arria 10 Gen1, Gen2, and Gen3 x8 Channel and Pin Placement



5.2 Channel Placement and fPLL Usage for the Gen1 and Gen2 Data Rates

The following figures illustrate the x1, x2, x4, and x8 channel placement for the Arria 10 Hard IP for PCI Express. In these figures, channels that are not used for the PCI Express protocol are available for other protocols. Unused channels are shown in gray.

Note: In all configurations, physical channel 4 in the PCS connects to logical channel 0 in the hard IP. You cannot change the channel placements illustrated below.

Figure 27. Arria 10 Gen1 and Gen2 x1 Channel Placement

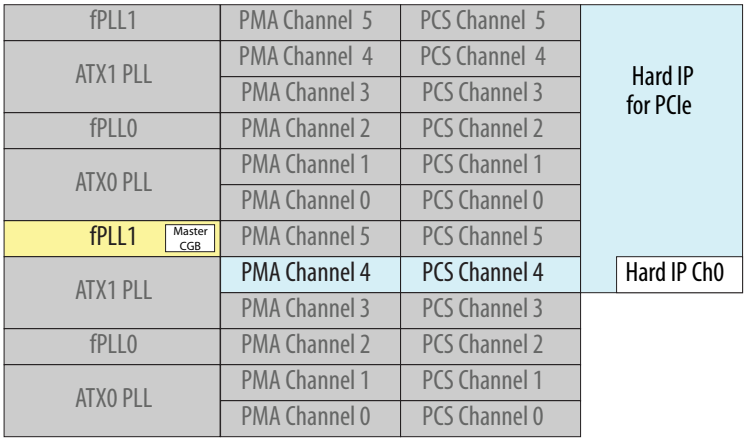


Figure 28. Arria 10 Gen1 and Gen2 x2 Channel Placement

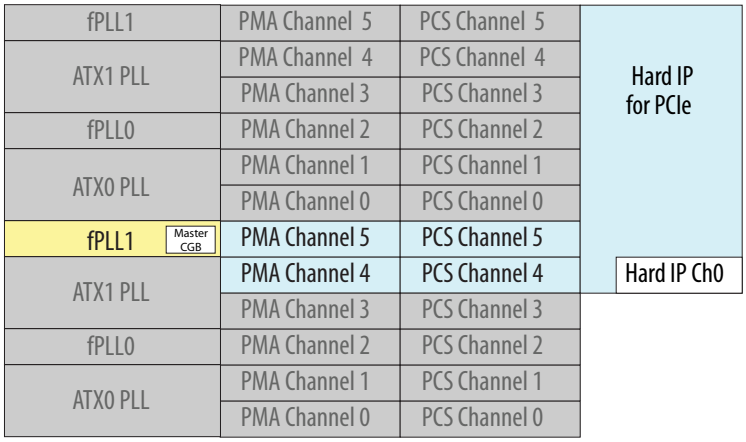


Figure 29. Arria 10 Gen1 and Gen2 x4 Channel Placement

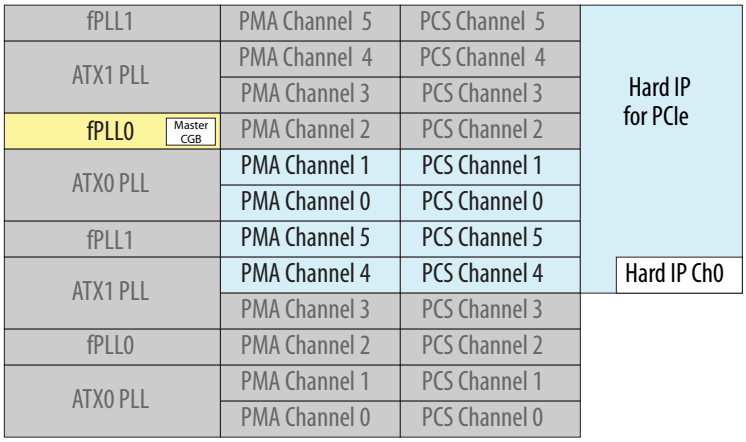
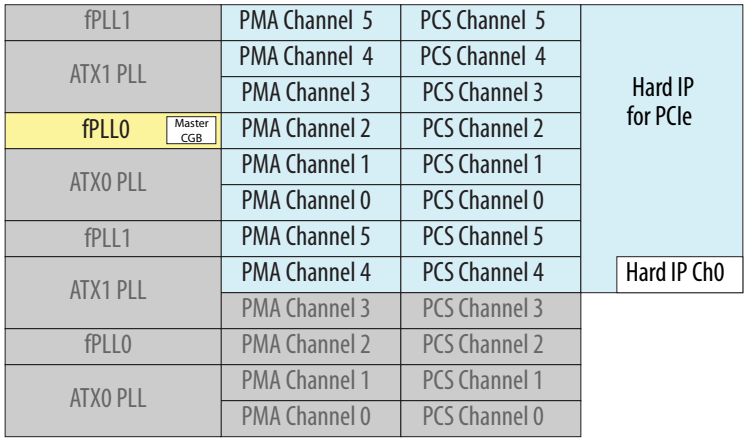


Figure 30. Gen1 and Gen2 x8 Channel Placement



5.3 Channel Placement and fPLL and ATX PLL Usage for the Gen3 Data Rate

The following figures illustrate the x1, x2, x4, and x8 channel placement for the Arria 10 Hard IP for PCI Express.

Gen3 variants must initially train at the Gen1 data rate. Consequently, Gen3 variants require an fPLL to generate the 2.5 and 5.0 Gbps clocks, and an ATX PLL to generate the 8.0 Gbps clock. In these figures, channels that are not used for the PCI Express protocol are available for other protocols. Unused channels are shown in gray.

Note: In all configurations, physical channel 4 in the PCS connects to logical channel 0 in the hard IP. You cannot change the channel placements illustrated below.

Figure 31. Arria 10 Gen3 x1 Channel Placement

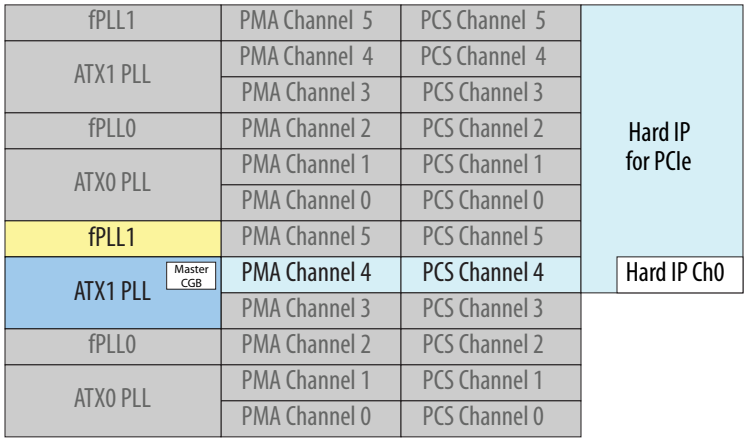


Figure 32. Arria 10 Gen3 x2 Channel Placement

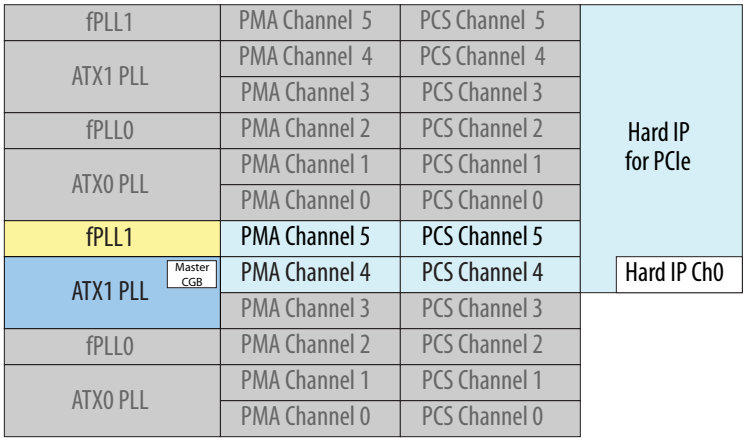


Figure 33. Arria 10 Gen3 x4 Channel Placement

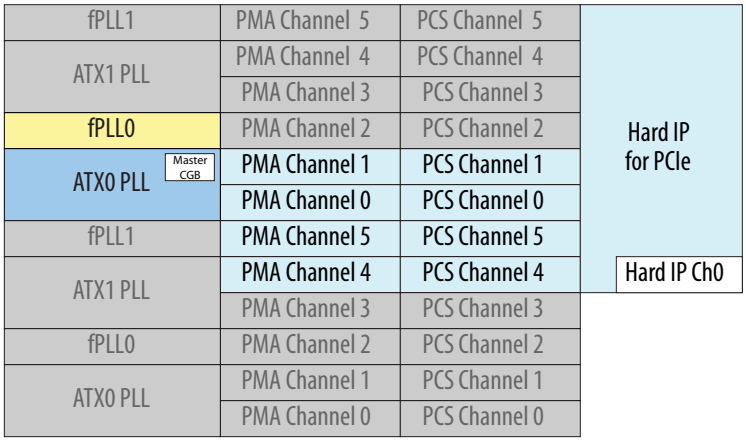
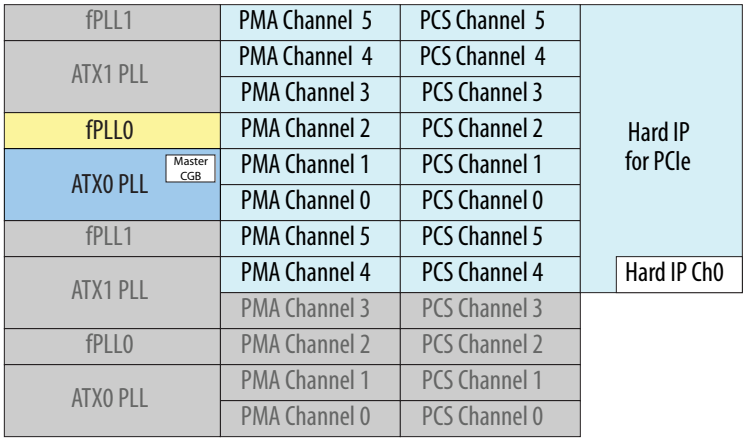


Figure 34. Gen3 x8 Channel Placement



5.4 PCI Express Gen3 Bank Usage Restrictions

Any transceiver channels that share a bank with active PCI Express interfaces that are Gen3 capable have the following restrictions. This includes both Hard IP and Soft IP implementations:

- When VCCR_GXB and VCCT_GXB are set to 1.03 V or 1.12 V, the maximum data rate supported for the non-PCIe channels in those banks is 12.5 Gbps for chip-to-chip applications. These channels cannot be used to drive backplanes or for GT rates.
- When VCCR_GXB and VCCT_GXB are set to 0.95 V, the non-PCIe channels in those banks cannot be used.

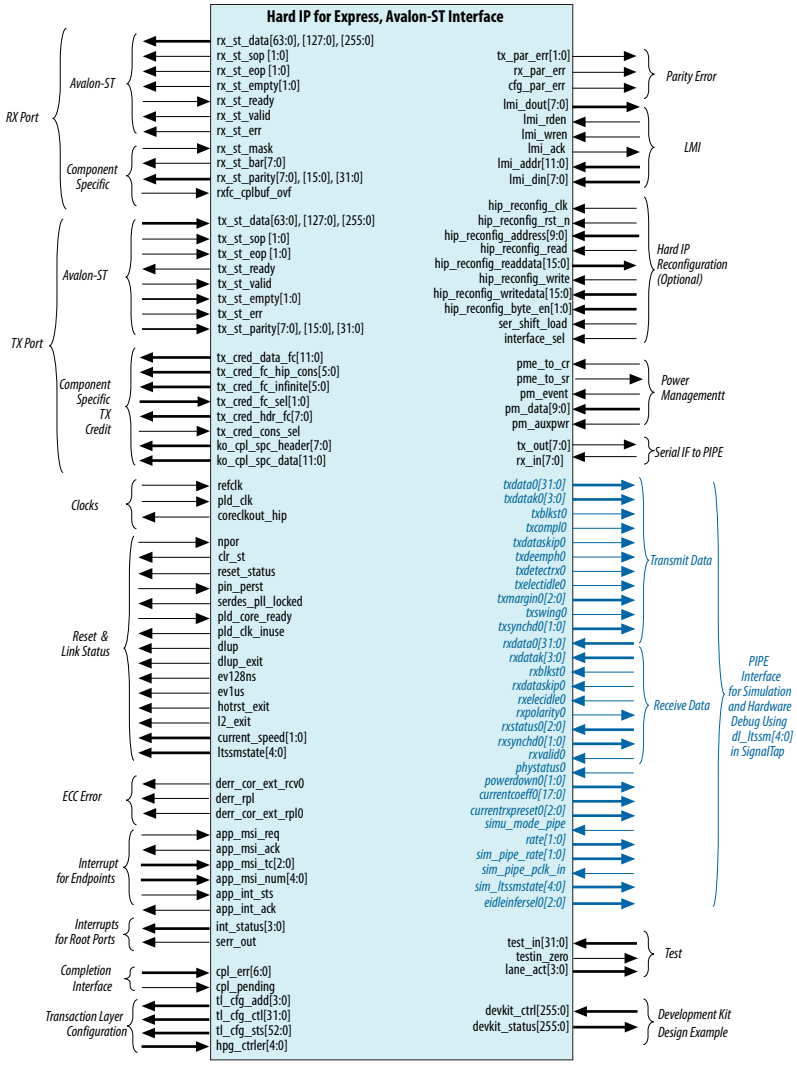
PCI Express interfaces that are only Gen1 or Gen2 capable are not affected.

Status

Affects all Arria 10 ES and production devices. No fix is planned.

6 Interfaces and Signal Descriptions

Figure 35. Avalon-ST Hard IP for PCI Express Top-Level Signals



6.1 Avalon-ST RX Interface

The following table describes the signals that comprise the Avalon-ST RX Datapath. The RX data signal can be 64, 128, or 256 bits.

Table 27. 64-, 128-, or 256-Bit Avalon-ST RX Datapath

Signal	Direction	Description
rx_st_data[<*n*>-1:0]	Output	Receive data bus. Refer to figures following this table for the mapping of the Transaction Layer's TLP information to rx_st_data and examples of the timing of this interface. Note that the position of the first payload dword depends on whether the TLP address is qword aligned. The mapping of message TLPs is the same as the mapping of TLPs with 4-dword headers. When using a 64-bit Avalon-ST bus, the width of rx_st_data is 64. When using a 128-bit Avalon-ST bus, the width of rx_st_data is 128. When using a 256-bit Avalon-ST bus, the width of rx_st_data is 256 bits.
rx_st_sop[1:0]	Output	Indicates that this is the first cycle of the TLP when rx_st_valid is asserted. When using a 256-bit Avalon-ST bus the following correspondences apply: When you turn on **Enable multiple packets per cycle**, • bit 0 indicates that a TLP begins in rx_st_data[127:0] • bit 1 indicates that a TLP begins in rx_st_data[255:128] In single packet per cycle mode, this signal is a single bit which indicates that a TLP begins in this cycle.
rx_st_eop[1:0]	Output	Indicates that this is the last cycle of the TLP when rx_st_valid is asserted. When using a 256-bit Avalon-ST bus the following correspondences apply: When you turn on **Enable multiple packets per cycle**, • bit 0 indicates that a TLP ends in rx_st_data[127:0] • bit 1 indicates that a TLP ends in rx_st_data[255:128] In single packet per cycle mode, this signal is a single bit which indicates that a TLP ends in this cycle.

continued...

Signal	Direction	Description
rx_st_empty[1:0]	Output	Indicates the number of empty qwords in rx_st_data. Not used when rx_st_data is 64 bits. Valid only when rx_st_eop is asserted in 128-bit and 256-bit modes. For 128-bit data, only bit 0 applies; this bit indicates whether the upper qword contains data. For 256-bit data single packet per cycle mode, both bits are used to indicate whether 0-3 upper qwords contain data, resulting in the following encodings for the 128-and 256-bit interfaces: • 128-Bit interface: — rx_st_empty = 0, rx_st_data[127:0]contains valid data — rx_st_empty = 1, rx_st_data[63:0] contains valid data • 256-bit interface: single packet per cycle mode — rx_st_empt y = 0, rx_st_data[255:0] contains valid data — rx_st_empty = 1, rx_st_data[191:0] contains valid data — rx_st_empty = 2, rx_st_data[127:0] contains valid data — rx_st_empty = 3, rx_st_data[63:0] contains valid data • For 256-bit data, when you turn on **Enable multi ple packets per cycle**, the following correspondences apply: — bit 1 applies to the eop occurring in rx_st_data[255:128] — bit 0 applies to the eop occurring in rx_st_data[127:0] • When the TLP ends in the lower 128 bits, the following equations apply: — rx_st_eop[0]=1 & rx_st_empty[0]=0, rx_st_data[127:0] contains valid data — rx_st_eop[0]=1 & rx_st_empty[0]=1, rx_st_data[63:0] contains valid data, rx_st_data[127:64] is empty • When TLP ends in the upper 128bits, the following equations apply: — rx_st_ eop[1]=1 & rx_st_empty[1]=0, rx_st_data[255:128] contains valid data — rx_st_eop[1]=1 & rx_st_empty[1]=1, rx_st_data[191:128] contains valid data, rx_st_data[255:192] is empty
rx_st_ready	Input	Indicates that the Application Layer is ready to accept data. The Application Layer deasserts this signal to throttle the data stream. If rx_st_ready is asserted by the Application Layer on cycle <*n*> , then <*n* + > readyLatency > is a ready cycle, during which the Transaction Layer may assert valid and transfer data. The RX interface supports a readyLatency of 2 cycles.
rx_st_valid	Output	Clocks rx_st_data into the Application Layer. Deasserts within 2 clocks of rx_st_ready deassertion and reasserts within 2 clocks of rx_st_ready assertion if more data is available to send. For 256-bit data, when you turn on **Enable multiple packets per cycle**, bit 0 applies to the entire bus rx_st_data[255:0]. Bit 1 is not used.
rx_st_err[<*n*>-1:0]	Output	Indicates that there is an uncorrectable error correction coding (ECC) error in the internal RX buffer. Active when ECC is enabled. ECC is automatically enabled by the Quartus Prime assembler. ECC corrects single-bit errors and detects double-bit errors on a per byte basis. When an uncorrectable ECC error is detected, rx_st_err is asserted for at least 1 cycle while rx_st_valid is asserted. For 256-bit data, when you turn on **Enable multiple packets per cycle**, bit 0 applies to the entire bus rx_st_data[255:0]. Bit 1 is not used. Intel recommends resetting the Arria 10 Hard IP for PCI Express when an uncorrectable double-bit ECC error is detected.

Attention: If you instantiate this IP core as a separate component from the Quartus Prime IP Catalog, the Message pane reports the following warning messages:

```
pcie_a10.pcie_a10_hip_0.tx.st Interface must have an associated reset
pcie_a10.pcie_a10_hip_0.rx.st Interface must have an associated reset
```

You can safely ignore these warnings because the IP core has a dedicated hard reset pin that is not part of the Avalon-ST TX or RX interface.

Related Links

Avalon Interface Specifications
For information about the Avalon-ST interface protocol.

6.1.1 Avalon-ST RX Component Specific Signals

Table 28. Avalon-ST RX Component Specific Signals

Signal	Direction	Description
rx_st_mask	Input	The Application Layer asserts this signal to tell the Hard IP to stop sending non-posted requests. This signal can be asserted at any time. The total number of non-posted requests that can be transferred to the Application Layer after rx_st_mask is asserted is not more than 10. This signal stalls only non-posted TLPs. All others continue to be forwarded to the Application Layer. The stalled non-posted TLPs are held in the RX buffer until the mask signal is deasserted. They are not be discarded. If used in a Root Port mode, asserting the rx_st_mask signal stops all I/O and MemRd and configuration accesses because these are all non-posted transactions.
rx_st_bar[7:0]	Output	The decoded BAR bits for the TLP. Valid for MRd, MWr, IOWR, and IORD TLPs. Ignored for the completion or message TLPs. Valid during the cycle in which rx_st_sop is asserted. Refer to *64-Bit Avalon-ST rx_st_data<n> Cycle Definitions for 4-Dword Header TLPs with Non-Qword Addresses* and *128-Bit Avalon-ST rx_st_data<n> Cycle Definition for 3-Dword Header TLPs with Qword Aligned Addresses* for the timing of this signal for 64- and 128-bit data, respectively. The following encodings are defined for Endpoints: • Bit 0: BAR 0 • Bit 1: BAR 1 • Bit 2: BAR 2 • Bit 3: BAR 3 • Bit 4: BAR 4 • Bit 5: BAR 5 • Bit 6: Expansion ROM • Bit 7: Reserved The following encodings are defined for Root Ports: • Bit 0: BAR 0 • Bit 1: BAR 1 • Bit 2: Primary Bus number • Bit 3: Secondary Bus number • Bit 4: Secondary Bus number to Subordinate Bus number window • Bit 5: I/O window • Bit 6: Non-Prefetchable window • Bit 7: Prefetchable window

continued...

Signal	Direction	Description
		For multiple packets per cycle, this signal is undefined. If you turn on **Enable multiple packets per cycle**, do not use this signal to identify the address BAR hit.
rx_st_parity[<*n*>-1:0]	Output	The IP core generates byte parity when you turn on **Enable byte parity ports on Avalon-ST interface** on the **System Settings** tab of the parameter editor. Each bit represents odd parity of the associated byte of the rx_st_datarx_st_data bus. For example, bit[0] corresponds to rx_st_data[7:0] rx_st_data[7:0], bit[1] corresponds to rx_st_data[15:8].
rxfc_cplbuf_ovf]	Output	When asserted indicates that the internal RX buffer has overflowed.

For more information about the Avalon-ST protocol, refer to the *Avalon Interface Specifications*.

Related Links

Avalon Interface Specifications
For information about the Avalon-ST interface protocol.

6.1.2 Data Alignment and Timing for the 64-Bit Avalon-ST RX Interface

To facilitate the interface to 64-bit memories, the Arria 10 Hard IP for PCI Express aligns data to the qword or 64 bits by default. Consequently, if the header presents an address that is not qword aligned, the Hard IP block shifts the data within the qword to achieve the correct alignment.

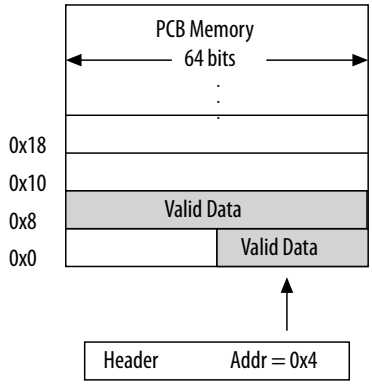
Qword alignment applies to all types of request TLPs with data, including the following TLPs:

- Memory writes
- Configuration writes
- I/O writes

The alignment of the request TLP depends on bit 2 of the request address. For completion TLPs with data, alignment depends on bit 2 of the `lower address` field. This bit is always 0 (aligned to qword boundary) for completion with data TLPs that are for configuration read or I/O read requests.

Figure 36. Qword Alignment

The following figure shows how an address that is not qword aligned, 0x4, is stored in memory. The byte enables only qualify data that is being written. This means that the byte enables are undefined for 0x0–0x3. This example corresponds to *64-Bit Avalon-ST rx_st_data<n> Cycle Definition for 3-Dword Header TLPs with Non-Qword Aligned Address*.



The following table shows the byte ordering for header and data packets.

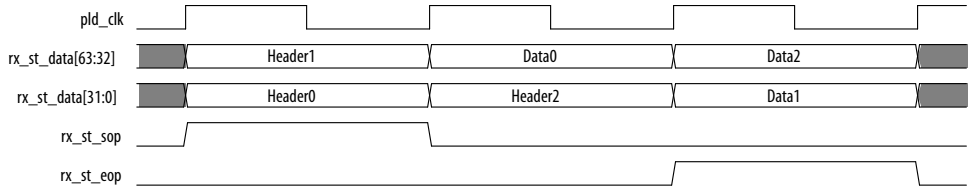
Table 29. Mapping Avalon-ST Packets to PCI Express TLPs

Packet	TLP
Header0	pcie_hdr_byte0, pcie_hdr _byte1, pcie_hdr _byte2, pcie_hdr _byte3
Header1	pcie_hdr _byte4, pcie_hdr _byte5, pcie_hdr byte6, pcie_hdr _byte7
Header2	pcie_hdr _byte8, pcie_hdr _byte9, pcie_hdr _byte10, pcie_hdr _byte11
Header3	pcie_hdr _byte12, pcie_hdr _byte13, header_byte14, pcie_hdr _byte15
Data0	pcie_data_byte3, pcie_data_byte2, pcie_data_byte1, pcie_data_byte0
Data1	pcie_data_byte7, pcie_data_byte6, pcie_data_byte5, pcie_data_byte4
Data2	pcie_data_byte11, pcie_data_byte10, pcie_data_byte9, pcie_data_byte8
Data<*n*>	pcie_data_byte<*4n+3*>, pcie_data_byte<*4n+2*>, pcie_data_byte<*4n+1*>, pcie_data_byte<*n*>

The following figure illustrates the mapping of Avalon-ST RX packets to PCI Express TLPs for a three dword header with non-qword aligned addresses with a 64-bit bus. In this example, the byte address is unaligned and ends with 0x4, causing the first data to correspond to `rx_st_data[63:32]` .

Note: The Avalon-ST protocol, as defined in *Avalon Interface Specifications*, is big endian, while the Hard IP for PCI Express packs symbols into words in little endian format. Consequently, you cannot use the standard data format adapters available in Qsys.

Figure 37. 64-Bit Avalon-ST rx_st_data<n> Cycle Definition for 3-Dword Header TLPs with Non-Qword Aligned Address



The following figure illustrates the mapping of Avalon-ST RX packets to PCI Express TLPs for a three dword header with qword aligned addresses. Note that the byte enables indicate the first byte of data is not valid and the last dword of data has a single valid byte.

Figure 38. 64-Bit Avalon-ST rx_st_data<n> Cycle Definition for 3-Dword Header TLPs with Qword Aligned Address

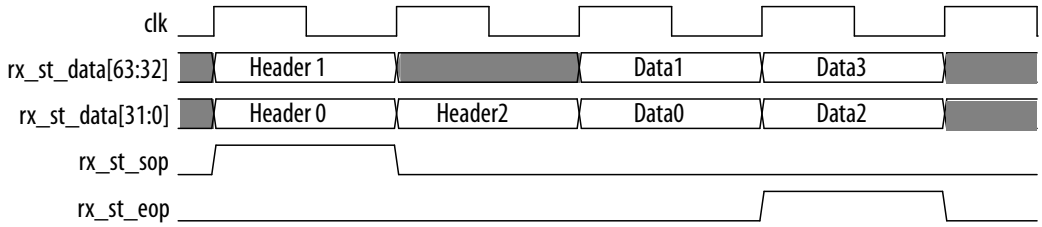


Figure 39. 64-Bit Avalon-ST rx_st_data<n> Cycle Definitions for 4-Dword Header TLPs with Qword Aligned Addresses

The following figure shows the mapping of Avalon-ST RX packets to PCI Express TLPs for TLPs for a four dword header with qword aligned addresses with a 64-bit bus.

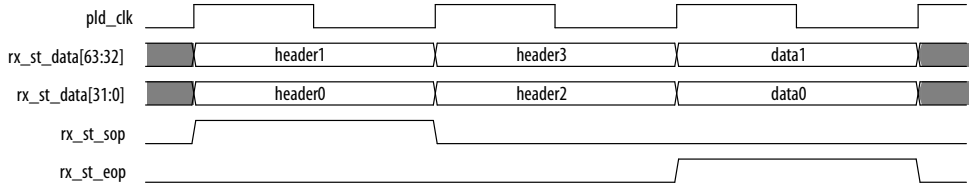


Figure 40. 64-Bit Avalon-ST rx_st_data<n> Cycle Definitions for 4-Dword Header TLPs with Non-Qword Addresses

The following figure shows the mapping of Avalon-ST RX packet to PCI Express TLPs for TLPs for a four dword header with non-qword addresses with a 64-bit bus. Note that the address of the first dword is 0x4. The address of the first enabled byte is 0xC.

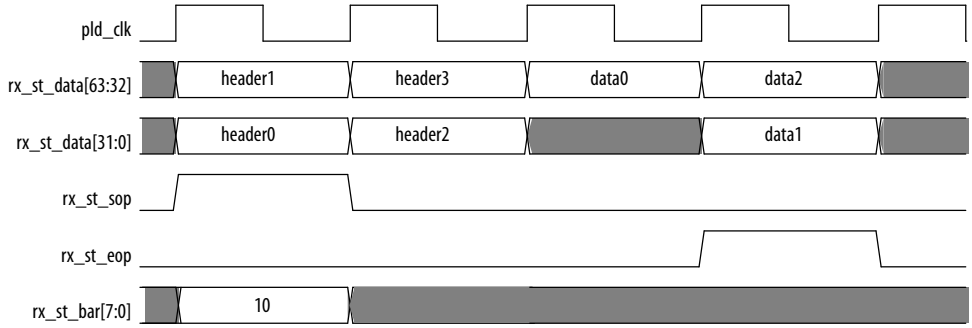


Figure 41. 64-Bit Application Layer Backpressures Transaction Layer

The following figure illustrates the timing of the RX interface when the Application Layer backpressures the Arria 10 Hard IP for PCI Express by deasserting `rx_st_ready`. The `rx_st_valid` signal deasserts within three cycles after `rx_st_ready` is deasserted. In this example, `rx_st_valid` is deasserted in the next cycle. `rx_st_data` is held until the Application Layer is able to accept it.

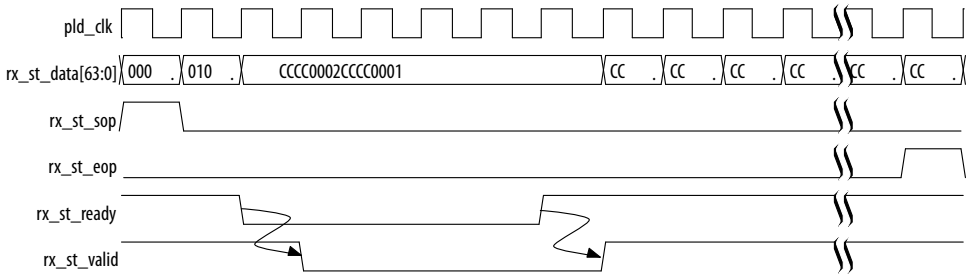
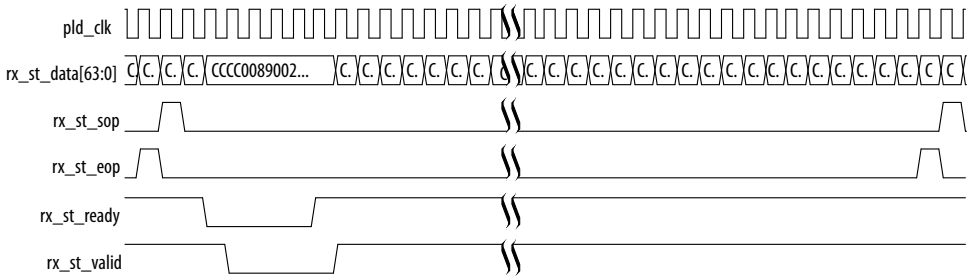


Figure 42. 64-Bit Avalon-ST Interface Back-to-Back Transmission

The following figure illustrates back-to-back transmission on the 64-bit Avalon-ST RX interface with no idle cycles between the assertion of `rx_st_eop` and `rx_st_sop`.



Related Links

Avalon Interface Specifications

For information about the Avalon-ST interface protocol.

6.1.3 Data Alignment and Timing for the 128-Bit Avalon-ST RX Interface

Figure 43. 128-Bit Avalon-ST rx_st_data<n> Cycle Definition for 3-Dword Header TLPs with Qword Aligned Addresses

The following figure shows the mapping of 128-bit Avalon-ST RX packets to PCI Express TLPs for TLPs with a three dword header and qword aligned addresses. The assertion of `rx_st_empty` in a `rx_st_eop` cycle, indicates valid data on the lower 64 bits of `rx_st _data`.

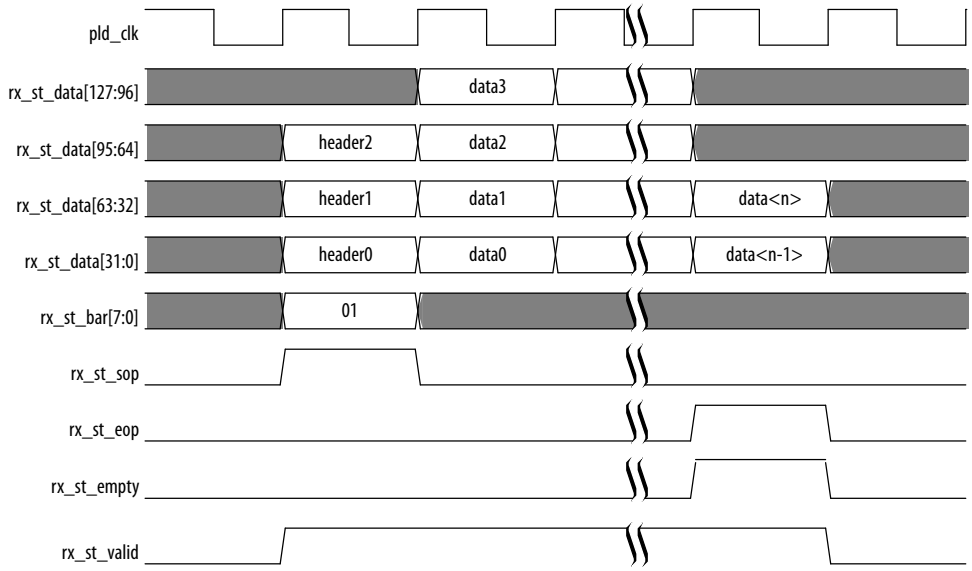


Figure 44. 128-Bit Avalon-ST rx_st_data<n> Cycle Definition for 3-Dword Header TLPs with non-Qword Aligned Addresses

The following figure shows the mapping of 128-bit Avalon-ST RX packets to PCI Express TLPs for TLPs with a 3 dword header and non-qword aligned addresses. In this case, bits[127:96] represent Data0 because address[2] in the TLP header is set. The assertion of `rx_st_empty` in a `rx_st_eop` cycle indicates valid data on the lower 64 bits of `rx_st_data`.

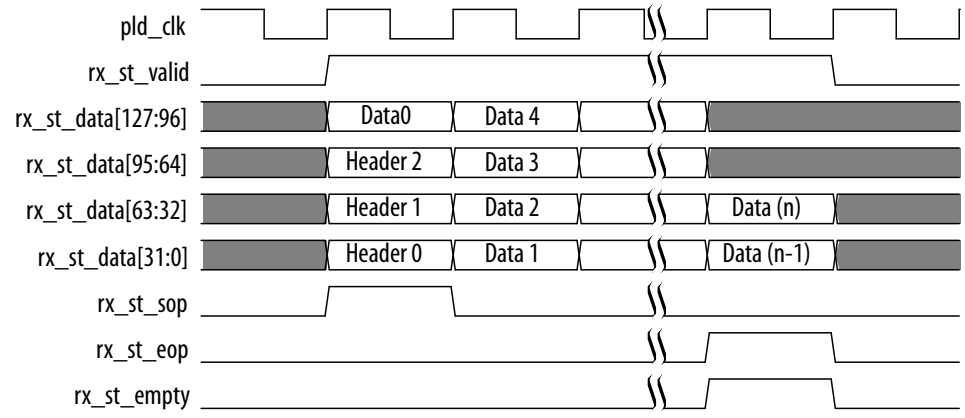


Figure 45. 128-Bit Avalon-ST rx_st_data Cycle Definition for 4-Dword Header TLPs with non-Qword Aligned Addresses

The following figure shows the mapping of 128-bit Avalon-ST RX packets to PCI Express TLPs for a four dword header with non-qword aligned addresses. In this example, `rx_st_empty` is low because the data is valid for all 128 bits in the `rx_st_eop` cycle.

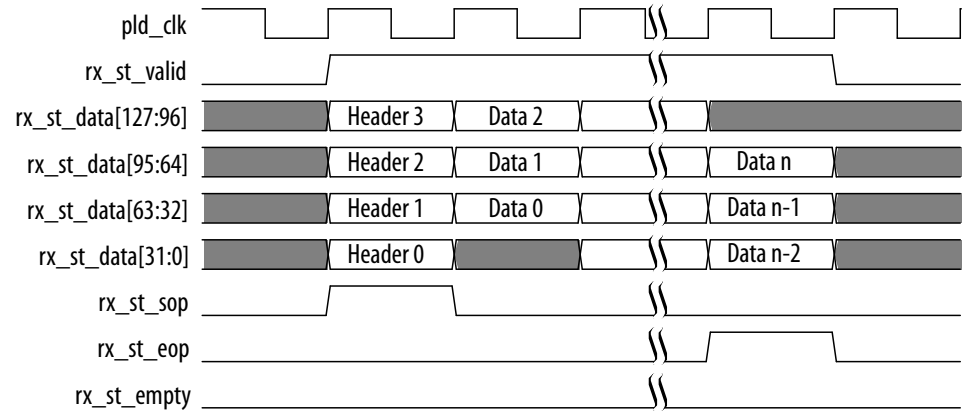


Figure 46. 128-Bit Avalon-ST rx_st_data Cycle Definition for 4-Dword Header TLPs with Qword Aligned Addresses

The following figure shows the mapping of 128-bit Avalon-ST RX packets to PCI Express TLPs for a four dword header with qword aligned addresses. In this example, `rx_st_empty` is low because data is valid for all 128-bits in the `rx_st_eop` cycle.

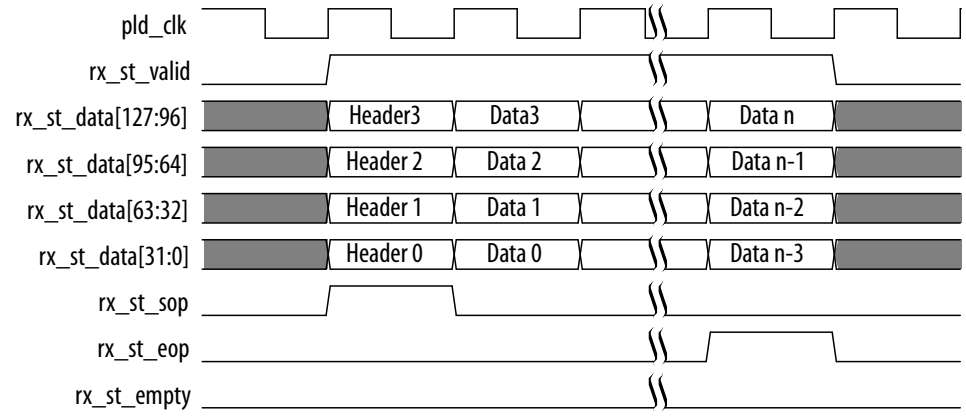
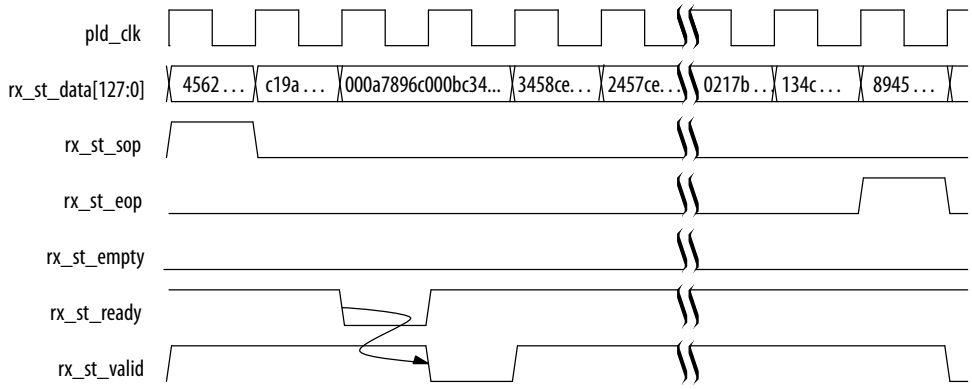


Figure 47. 128-Bit Application Layer Backpressures Hard IP Transaction Layer for RX Transactions

The following figure illustrates the timing of the RX interface when the Application Layer backpressures the Hard IP by deasserting `rx_st_ready`. The `rx_st_valid` signal deasserts within three cycles after `rx_st_ready` is deasserted. In this example, `rx_st_valid` is deasserted in the next cycle. `rx_st_data` is held until the Application Layer is able to accept it.



The following figure illustrates back-to-back transmission on the 128-bit Avalon-ST RX interface with no idle cycles between the assertion of `rx_st_eop` and `rx_st_sop`.

Figure 48. 128-Bit Avalon-ST Interface Back-to-Back Transmission

The following figure illustrates back-to-back transmission on the 128-bit Avalon-ST RX interface with no idle cycles between the assertion of `rx_st_eop` and `rx_st_sop`.

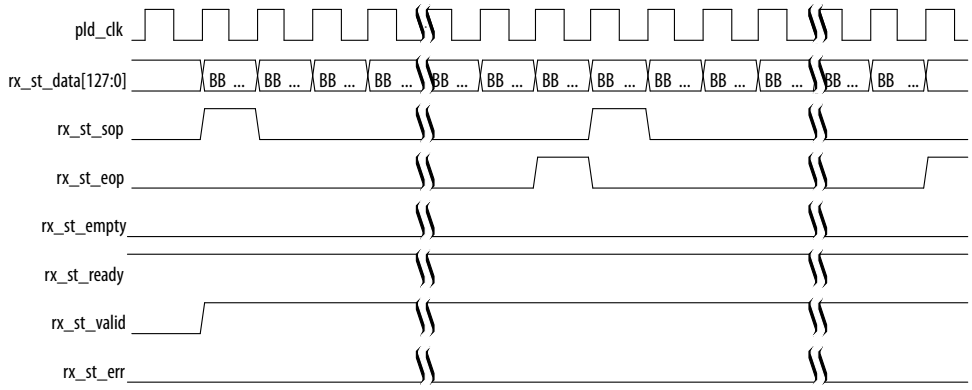
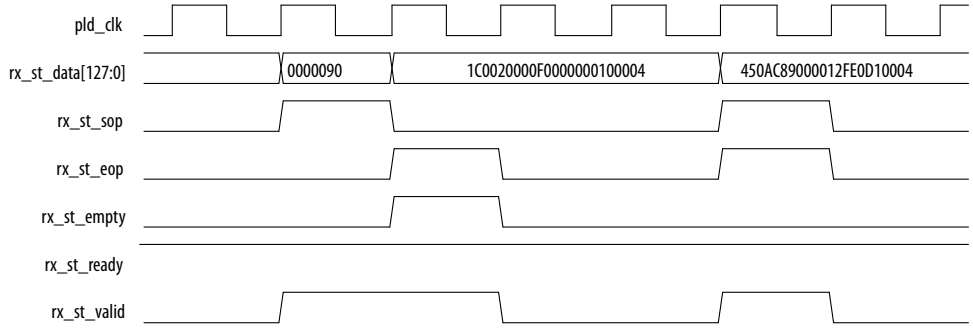


Figure 49. 128-Bit Packet Examples of rx_st_empty and Single-Cycle Packet

The following figure illustrates a two-cycle packet with valid data in the lower qword (`rx_st_data[63:0]`) and a one-cycle packet where the `rx_st_sop` and `rx_st_eop` occur in the same cycle.



6.1.4 Data Alignment and Timing for 256-Bit Avalon-ST RX Interface

Figure 50. Location of Headers and Data for Avalon-ST 256-Bit Interface

The following figure shows the location of headers and data for the 256-bit Avalon-ST packets. This layout of data applies to both the TX and RX buses.

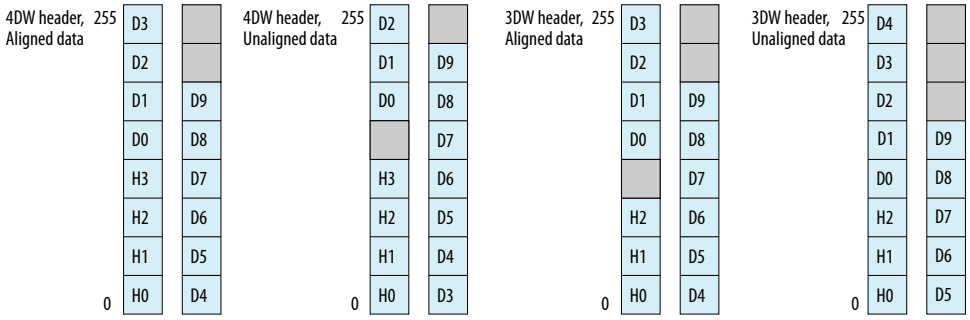
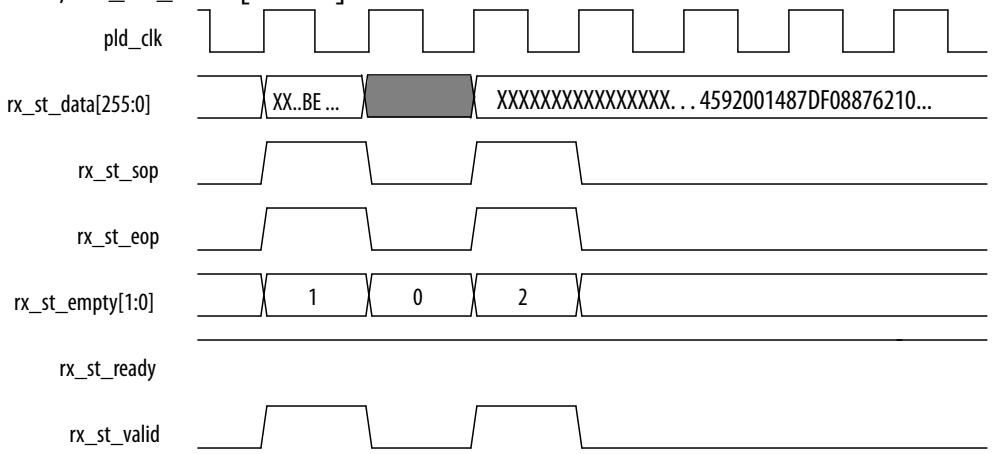


Figure 51. 256-Bit Avalon-ST RX Packets Use of rx_st_empty and Single-Cycle Packets

The following figure illustrates two single-cycle 256-bit packets. The first packet has two empty dwords, `rx_st_data[191:0]` is valid. The second packet has four empty dwords; `rx_st_data[127:0]` is valid.



6.1.5 Tradeoffs to Consider when Enabling Multiple Packets per Cycle

If you enable **Multiple Packets Per Cycle** under the **Systems Settings** heading, a TLP can start on a 128-bit boundary. This mode supports multiple start of packet and end of packet signals in a single cycle when the Avalon-ST interface is 256 bits wide. It reduces the wasted bandwidth for small packets.

A comparison of the largest and smallest packet sizes illustrates this point. Large packets using the full 256 bits achieve the following throughput:

```
256/256*8 = 8 GBytes/sec
```

The smallest PCIe packet, such as a 3-dword memory read, uses 96 bits of the 256-bits bus and achieve the following throughput:

```
96/256*8 = 3 GBytes/sec
```

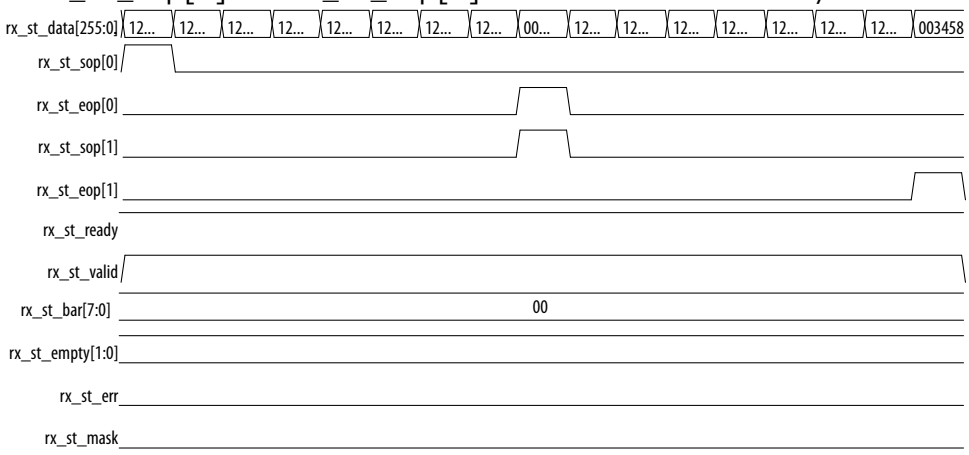
If you enable m**Multiple Packets Per Cycle**, when a TLP ends in the upper 128 bits of the Avalon-ST bus, a new TLP can start in the lower 128 bits Consequently, the bandwidth of small packets doubles:

```
96*2/256*8 = 6 GBytes/sec
```

This mode adds complexity to the Application Layer user decode logic. However, it could result in higher throughput.

Figure 52. 256-Bit Avalon-ST RX Interface with Multiple Packets Per Cycle

The following figure illustrates this mode for a 256-bit Avalon-ST RX interface. In this figure `rx_st_eop[0]` and `rx_st_sop[1]` are asserted in the same cycle.



6.2 Avalon-ST TX Interface

The following table describes the signals that comprise the Avalon-ST TX Datapath. The TX data signal can be 64, 128, or 256 bits.

Table 30. 64-, 128-, or 256-Bit Avalon-ST TX Datapath

Signal	Direction	Description
`tx_st_data[<n>-1:0]`	Input	Data for transmission. Transmit data bus. Refer to the following sections on data alignment for the 64-, 128-, and 256-bit interfaces for the mapping of TLP packets to `tx_st_data` and examples of the timing of this interface. When using a 64-bit Avalon-ST bus, the width of `tx_st_data` is 64. When using a 128-bit Avalon-ST bus, the width of `tx_st_data` is 128 bits. When using a 256-bit Avalon-ST bus, the width of `tx_st_data` is 256 bits. The Application Layer must provide a properly formatted TLP on the TX interface. The mapping of message TLPs is the same as the mapping of Transaction Layer TLPs with 4 dword headers. The number of data cycles must be correct for the length and address fields in the header. Issuing a packet with an incorrect number of data cycles results in the TX interface hanging and becoming unable to accept further requests. <*n*> = 64, 128, or 256.
`tx_st_sop[<n>-1:0]`	Input	Indicates first cycle of a TLP when asserted together with `tx_st_valid`. <*n*> = 1 or 2. When using a 256-bit Avalon-ST bus with **Multiple packets per cycle**, bit 0 indicates that a TLP begins in tx_st_data[127:0], bit 1 indicates that a TLP begins in tx_st_data[255:128].
`tx_st_eop[<n>-1:0]`	Input	Indicates last cycle of a TLP when asserted together with `tx_st_valid`. <*n*> = 1 or 2. When using a 256-bit Avalon-ST bus with **Multiple packets per cycle**, bit 0 indicates that a TLP ends with tx_st_data[127:0], bit 1 indicates that a TLP ends with tx_st_data[255:128].
`tx_st_ready`	Output	Indicates that the Transaction Layer is ready to accept data for transmission. The core deasserts this signal to throttle the data stream. `tx_st_ready` may be asserted during reset. The Application Layer should

continued...

Signal	Direction	Description
		wait at least 2 clock cycles after the reset is released before issuing packets on the Avalon-ST TX interface. The reset_status signal can also be used to monitor when the IP core has come out of reset. If tx_st_ready is asserted by the Transaction Layer on cycle <*n*> , then <*n* + readyLatency> is a ready cycle, during which the Application Layer may assert valid and transfer data. When tx_st_ready, tx_st_valid and tx_st_data are registered (the typical case), Intel recommends a readyLatency of 2 cycles to facilitate timing closure; however, a readyLatency of 1 cycle is possible. If no other delays are added to the read-valid latency, the resulting delay corresponds to a readyLatency of 2.
tx_st_valid	Input	Clocks tx_st_data to the core when tx_st_ready is also asserted. Between tx_st_sop and tx_st_eop, tx_st_valid must not be deasserted in the middle of a TLP except in response to tx_st_ready deassertion. When tx_st_ready deasserts, this signal must deassert within 1 or 2 clock cycles. When tx_st_ready reasserts, and tx_st_data is in mid-TLP, this signal must reassert within 2 cycles. The figure entitled*64-Bit Transaction Layer Backpressures the Application Layer* illustrates the timing of this signal. For 256-bit data, when you turn on **Enable multiple packets per cycle**, the bit 0 applies to the entire bus tx_st_data[255:0]. Bit 1 is not used. To facilitate timing closure, Intel recommends that you register both the tx_st_ready and tx_st_valid signals. If no other delays are added to the ready-valid latency, the resulting delay corresponds to a readyLatency of 2.
tx_st_empty[1:0]	Input	Indicates the number of qwords that are empty during cycles that contain the end of a packet. When asserted, the empty dwords are in the high-order bits. Valid only when tx_st_eop is asserted. Not used when tx_st_data is 64 bits. For 128-bit data, only bit 0 applies and indicates whether the upper qword contains data. For 256-bit data, both bits are used to indicate the number of upper words that contain data, resulting in the following encodings for the 128-and 256-bit interfaces: 128-Bit interface:tx_st_empty = 0, tx_st_data[127:0]contains valid datatx_st_empty = 1, tx_st_data[63:0] contains valid data 256-bit interface:tx_st_empty = 0, tx_st_data[255:0] contains valid datatx_ st_empty = 1, tx_st_data[191:0] contains valid datatx_st_empty = 2, tx_st_data[127:0] contains valid datatx_st_empty = 3, tx_st_data[63:0] contains valid data For 256-bit data, when you turn on **Enable multiple packets per cycle**, the following correspondences apply: • bit 1 applies to the eop occurring in rx_st_data[255:128] • bit 0 applies to the eop occurring in rx_st_data[127:0] When the TLP ends in the lower 128bits, the following equations apply: • tx_st_eop[0]=1 & tx_st_empty[0]=0, tx_st_data[127:0] contains valid data • tx_st_eop[0]=1 & tx_st_empty[0]=1, tx_st_data[63:0] contains valid data, tx_st_data[127:64] is empty When TLP ends in the upper 128bits, the following equations apply: • tx_st_eop[1]=1 & tx_st_empty[1]=0, tx_st_data[255:128] contains valid data • tx_st_eop[1]=1 & tx_st_empty[1]=1, tx_st_data[191:128] contains valid data, tx_st_data[255:192] is empty
tx_st_err	Input	Indicates an error on transmitted TLP. This signal is used to nullify a packet. It should only be applied to posted and completion TLPs with payload. To nullify a packet, assert this signal for 1 cycle after the SOP and

continued...

Signal	Direction	Description
		before the EOP. When a packet is nullified, the following packet should not be transmitted until the next clock cycle. `tx_st_err` is not available for packets that are 1 or 2 cycles long. For 256-bit data, when you turn on **Enable multiple packets per cycle**, bit 0 applies to the entire bus `tx_st_data[255:0]`. Bit 1 is not used. Refer to the figure entitled *128-Bit Avalon-ST tx_st_data Cycle Definition for 3-Dword Header TLP with non-Qword Aligned Address* for a timing diagram that illustrates the use of the error signal. Note that it must be asserted while the valid signal is asserted.
`tx_st_parity[<n>-1:0]`	Output	Byte parity is generated when you turn on **Enable byte parity ports on Avalon ST interface** on the **System Settings** tab of the parameter editor.Each bit represents odd parity of the associated byte of the `tx_st_data` bus. For example, bit[0] corresponds to `tx_st_data[7:0]`, bit[1] corresponds to `tx_st_data[15:8]`, and so on. <*n*> = 8, 16, or 32.
Component Specific Signals		
`tx_cred_data_fc[11:0]`	Output	Data credit limit for the credit type specified by `tx_cred_fc_sel`. Each credit is 16 bytes. There is a latency of two `pld_clk` clocks between a change on `tx_cred_fc_sel` and the corresponding data appearing on `tx_cred_data_fc` and `tx_cred_hdr_fc`.
`tx_cred_fc_hip_cons[5:0]`	Output	Asserted for 1 cycle each time the Hard IP consumes a credit. These credits are from messages that the Hard IP for PCIe generates for the following reasons: • To respond to memory read requests • To send error messages This signal is not asserted when an Application Layer credit is consumed. For optimum performance the Application Layer can track of its own consumed credits. (The hard IP also tracks credits and deasserts `tx_st_ready` if it runs out of credits of any type.) To calculate the total credits consumed, the Application Layer can add its own credits consumed to those consumed by the Hard IP for PCIe. The credit signals are valid after the link is up. The 6 bits of this vector correspond to the following 6 types of credit types: • [5]: posted headers • [4]: posted data • [3]: non-posted header • [2]: non-posted data • [1]: completion header • [0]: completion data During a single cycle, the IP core can consume either a single header credit or both a header and a data credit.
`tx_cred_fc_infinite[5:0]`	Output	When asserted, indicates that the corresponding credit type has infinite credits available and does not need to calculate credit limits. The 6 bits of this vector correspond to the following 6 types of credit types: • [5]: posted headers • [4]: posted data • [3]: non-posted header • [2]: non-posted data • [1]: completion header • [0]: completion data

continued...

Signal	Direction	Description
tx_cred_fc_sel[1:0]	Input	Signal to select between which credit type is displayed on the tx_cred_hdr_fc and tx_cred_data_fc outputs. There is a latency of two pld_clk clocks between a change on tx_cred_fc_sel and the corresponding data appearing on tx_cred_data_fc and tx_cred_hdr_fc. The following encoding are defined: • 2'b00: Output Posted credits • 2'b01: Output Non-Posted credits • 2'b10: Output Completions
tx_cred_hdr_fc[7:0]	Output	Header credit limit for the credit type selected by tx_cred_fc_sel. Each credit is 20 bytes. There is a latency of two pld_clk clocks between a change on tx_cred_fc_sel and the corresponding data appearing on tx_cred_data_fc and tx_cred_hdr_fc.
tx_cred_cons_sel	Input	When 1, the tx_cred_* output signals specify the value of the hard IP internal credits-consumed counter. When 0, the output signals tx_cred_* specify the credit limit value.
ko_cpl_spc_header[7:0]	Output	The Application Layer can use this signal to build circuitry to prevent RX buffer overflow for completion headers. Endpoints must advertise infinite space for completion headers; however, RX buffer space is finite. ko_cpl_spc_header is a static signal that indicates the total number of completion headers that can be stored in the RX buffer.
ko_cpl_spc_data[11:0]	Output	The Application Layer can use this signal to build circuitry to prevent RX buffer overflow for completion data. Endpoints must advertise infinite space for completion data; however, RX buffer space is finite. ko_cpl_spc_data is a static signal that reflects the total number of 16 byte completion data units that can be stored in the completion RX buffer.

Related Links

- Data Alignment and Timing for the 64-Bit Avalon-ST TX Interface on page 73
- Data Alignment and Timing for the 128-Bit Avalon-ST TX Interface on page 76
- Data Alignment and Timing for the 256-Bit Avalon-ST TX Interface on page 79

6.2.1 Avalon-ST Packets to PCI Express TLPs

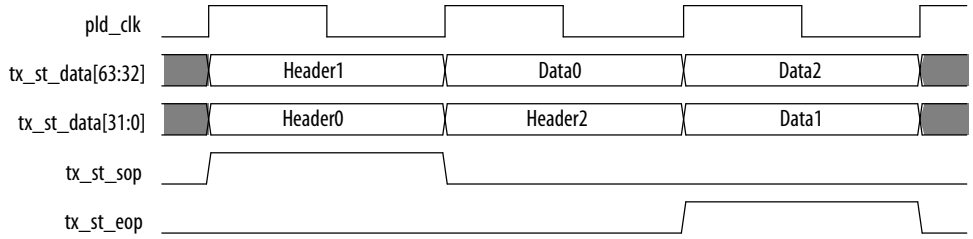
The following figures illustrate the mappings between Avalon-ST packets and PCI Express TLPs. These mappings apply to all types of TLPs, including posted, non-posted, and completion TLPs. Message TLPs use the mappings shown for four dword headers. TLP data is always address-aligned on the Avalon-ST interface whether or not the lower dwords of the header contains a valid address, as may be the case with TLP type (message request with data payload).

For additional information about TLP packet headers, refer to *Section 2.2.1 Common Packet Header Fields* in the *PCI Express Base Specification* .

6.2.2 Data Alignment and Timing for the 64-Bit Avalon-ST TX Interface

The following figure illustrates the mapping between Avalon-ST TX packets and PCI Express TLPs for three dword header TLPs with non-qword aligned addresses on a 64-bit bus.

Figure 53. 64-Bit Avalon-ST tx_st_data Cycle Definition for 3-Dword Header TLP with Non-Qword Aligned Address

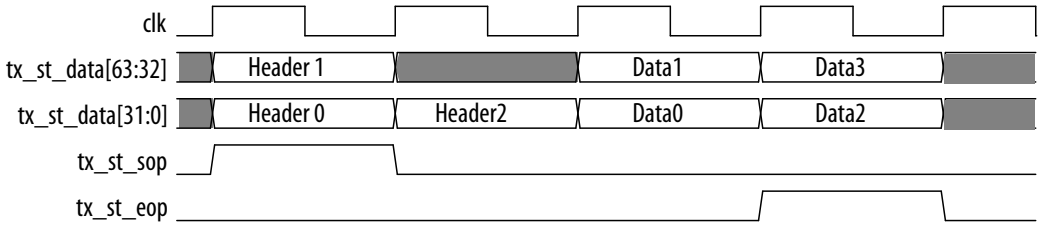


This figure illustrates the storage of non-qword aligned data.) Non-qword aligned address occur when `address[2]` is set. When address[2] is set, `tx_st_data[63:32]`contains `Data0` and `tx_st_data[31:0]` contains dword `header2`. In this figure, the headers are formed by the following bytes:

```
H0 ={pcie_hdr_byte0, pcie_hdr _byte1, pcie_hdr _byte2, pcie_hdr _byte3}
H1 = {pcie_hdr_byte4, pcie_hdr _byte5, header pcie_hdr byte6, pcie_hdr _byte7}
H2 = {pcie_hdr _byte8, pcie_hdr _byte9, pcie_hdr _byte10, pcie_hdr _byte11}
Data0 = {pcie_data_byte3, pcie_data_byte2, pcie_data_byte1, pcie_data_byte0}
Data1 = {pcie_data_byte7, pcie_data_byte6, pcie_data_byte5, pcie_data_byte4}
Data2 = {pcie_data_byte11, pcie_data_byte10, pcie_data_byte9, pcie_data_byte8}
```

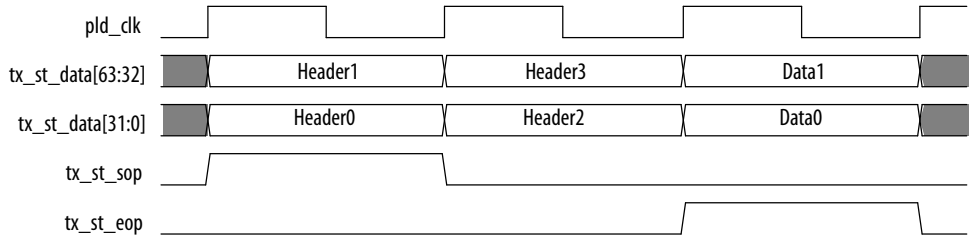
The following figure illustrates the mapping between Avalon-ST TX packets and PCI Express TLPs for three dword header TLPs with qword aligned addresses on a 64-bit bus.

Figure 54. 64-Bit Avalon-ST tx_st_data Cycle Definition for 3-Dword Header TLP with Qword Aligned Address



The following figure illustrates the mapping between Avalon-ST TX packets and PCI Express TLPs for a four dword header with qword aligned addresses on a 64-bit bus

Figure 55. 64-Bit Avalon-ST tx_st_data Cycle Definition for 4-Dword TLP with Qword Aligned Address



In this figure, the headers are formed by the following bytes.

```
H0 = {pcie_hdr_byte0, pcie_hdr _byte1, pcie_hdr _byte2, pcie_hdr _byte3}
H1 = {pcie_hdr _byte4, pcie_hdr _byte5, pcie_hdr byte6, pcie_hdr _byte7}
H2 = {pcie_hdr _byte8, pcie_hdr _byte9, pcie_hdr _byte10, pcie_hdr _byte11}
H3 = pcie_hdr _byte12, pcie_hdr _byte13, header_byte14, pcie_hdr _byte15}, 4
dword header only
Data0 = {pcie_data_byte3, pcie_data_byte2, pcie_data_byte1, pcie_data_byte0}
Data1 = {pcie_data_byte7, pcie_data_byte6, pcie_data_byte5, pcie_data_byte4}
```

Figure 56. 64-Bit Avalon-ST tx_st_data Cycle Definition for TLP 4-Dword Header with Non-Qword Aligned Address

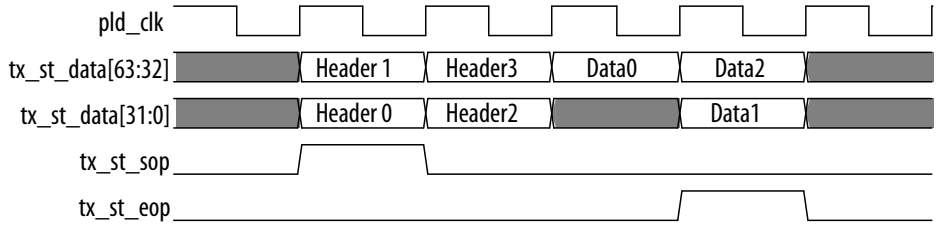


Figure 57. 64-Bit Transaction Layer Backpressures the Application Layer

The following figure illustrates the timing of the TX interface when the Arria 10 Hard IP for PCI Express pauses transmission by the Application Layer by deasserting `tx_st_ready`. Because the `readyLatency` is two cycles, the Application Layer deasserts `tx_st_valid` after two cycles and holds `tx_st_data` until two cycles after `tx_st_ready` is asserted.

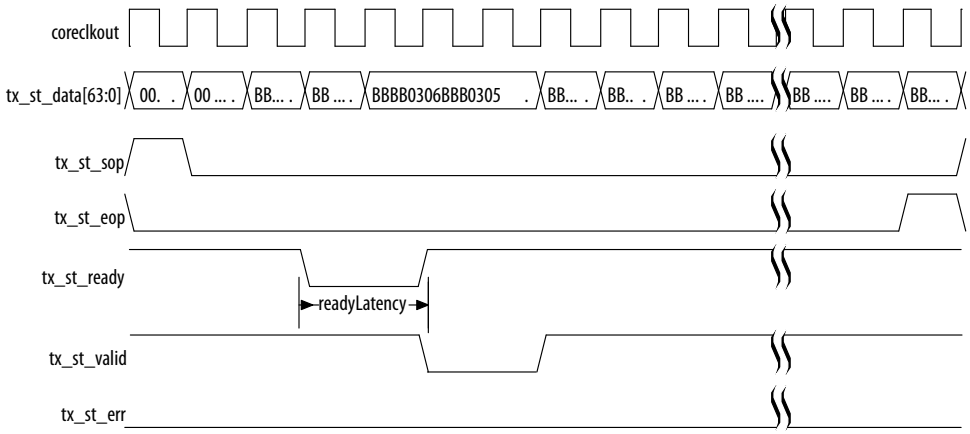
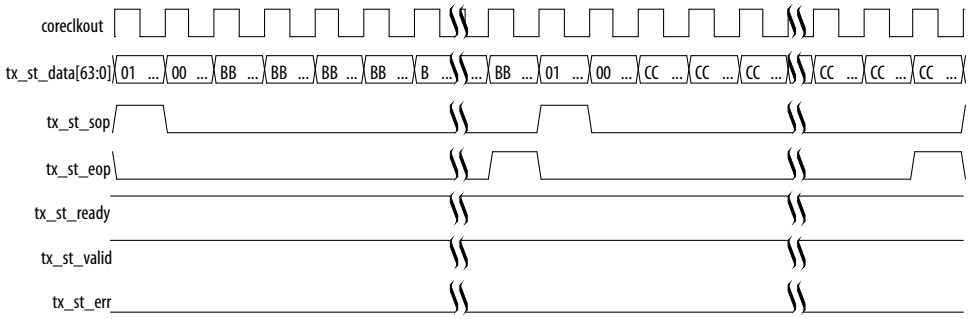


Figure 58. 64-Bit Back-to-Back Transmission on the TX Interface

The following figure illustrates back-to-back transmission of 64-bit packets with no idle cycles between the assertion of `tx_st_eop` and `tx_st_sop`.



6.2.3 Data Alignment and Timing for the 128-Bit Avalon-ST TX Interface

Figure 59. 128-Bit Avalon-ST tx_st_data Cycle Definition for 3-Dword Header TLP with Qword Aligned Address

The following figure shows the mapping of 128-bit Avalon-ST TX packets to PCI Express TLPs for a three dword header with qword aligned addresses. Assertion of `tx_st_empty` in an `rx_st_eop` cycle indicates valid data in the lower 64 bits of `tx_st_data`.

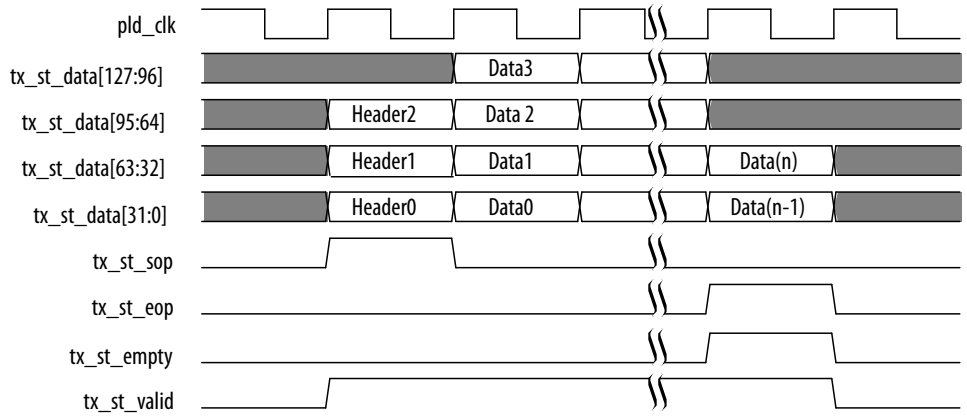


Figure 60. 128-Bit Avalon-ST tx_st_data Cycle Definition for 3-Dword Header TLP with non-Qword Aligned Address

The following figure shows the mapping of 128-bit Avalon-ST TX packets to PCI Express TLPs for a 3 dword header with non-qword aligned addresses. It also shows `tx_st_err` assertion.

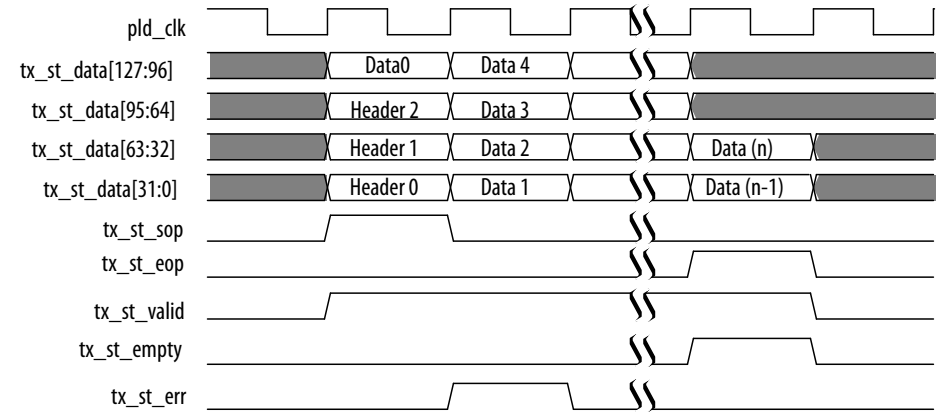


Figure 61. 128-Bit Avalon-ST tx_st_data Cycle Definition for 4-Dword Header TLP with Qword Aligned Address

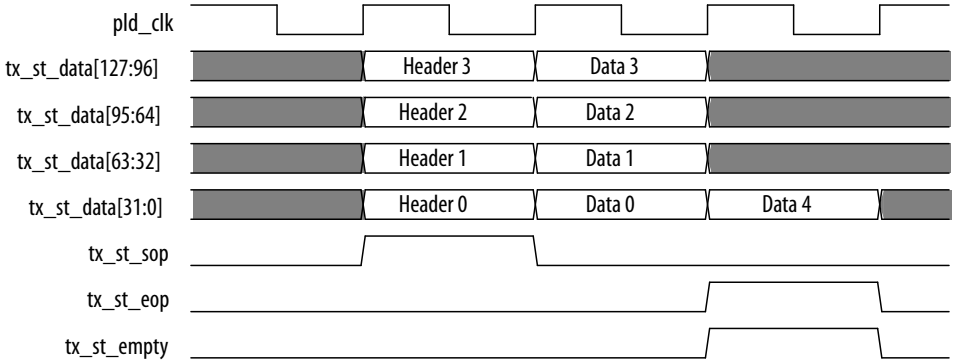


Figure 62. 128-Bit Avalon-ST tx_st_data Cycle Definition for 4-Dword Header TLP with non-Qword Aligned Address

The following figure shows the mapping of 128-bit Avalon-ST TX packets to PCI Express TLPs for a four dword header TLP with non-qword aligned addresses. In this example, `tx_st_empty` is low because the data ends in the upper 64 bits of `tx_st_data`.

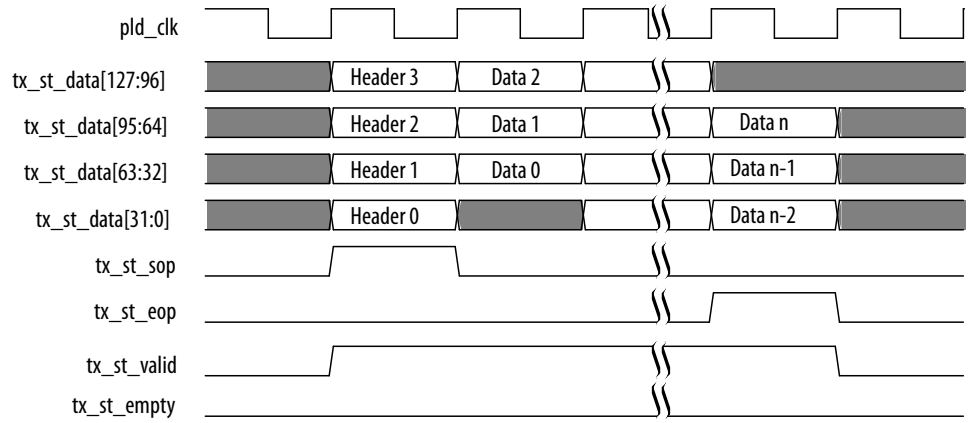


Figure 63. 128-Bit Back-to-Back Transmission on the Avalon-ST TX Interface

The following figure illustrates back-to-back transmission of 128-bit packets with idle dead cycles between the assertion of `tx_st_eop` and `tx_st_sop`.

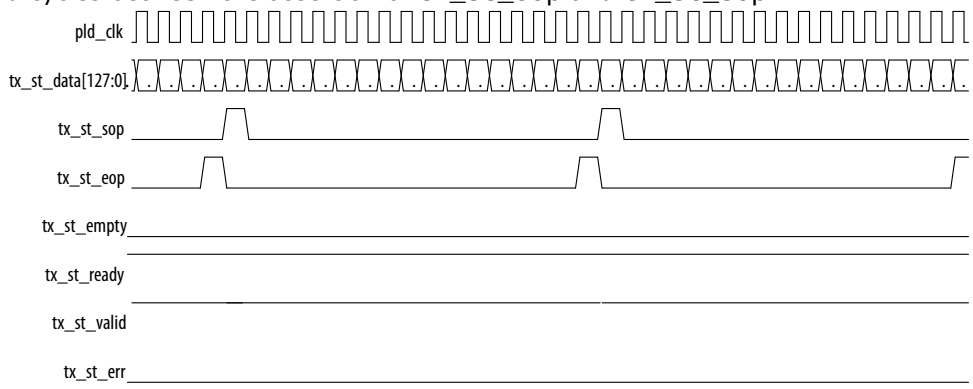
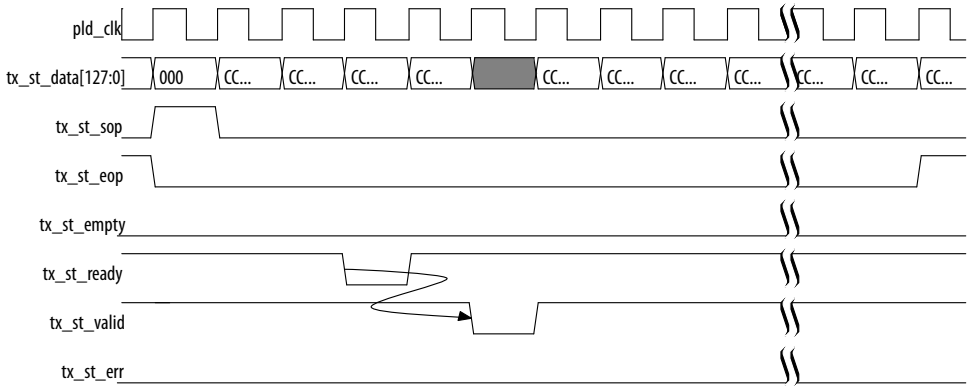


Figure 64. 128-Bit Hard IP Backpressures the Application Layer for TX Transactions

The following figure illustrates the timing of the TX interface when the Arria 10 Hard IP for PCI Express pauses the Application Layer by deasserting `tx_st_ready`. Because the `readyLatency` is two cycles, the Application Layer deasserts `tx_st_valid` after two cycles and holds `tx_st_data` until two cycles after `tx_st_ready` is reasserted



6.2.4 Data Alignment and Timing for the 256-Bit Avalon-ST TX Interface

Refer to Figure 8–16 on page 8–15 layout of headers and data for the 256-bit Avalon-ST packets with qword aligned and qword unaligned addresses.

Single Packet Per Cycle

In single packer per cycle mode, all received TLPs start at the lower 128-bit boundary on a 256-bit Avalon-ST interface. Turn on **Enable Multiple Packets per Cycle** on the System Settings tab of the parameter editor to change multiple packets per cycle.

Single packet per cycle mode requires simpler Application Layer packet decode logic on the TX and RX paths because packets always start in the lower 128-bits of the Avalon-ST interface. Although this mode simplifies the Application Layer logic, failure to use the full 256-bit Avalon-ST may slightly reduce the throughput of a design.

Figure 65. 256-Bit Avalon-ST tx_st_data Cycle Definition for 3-Dword Header TLP with Qword Addresses

The following figure illustrates the layout of header and data for a three dword header on a 256-bit bus with aligned and unaligned data.

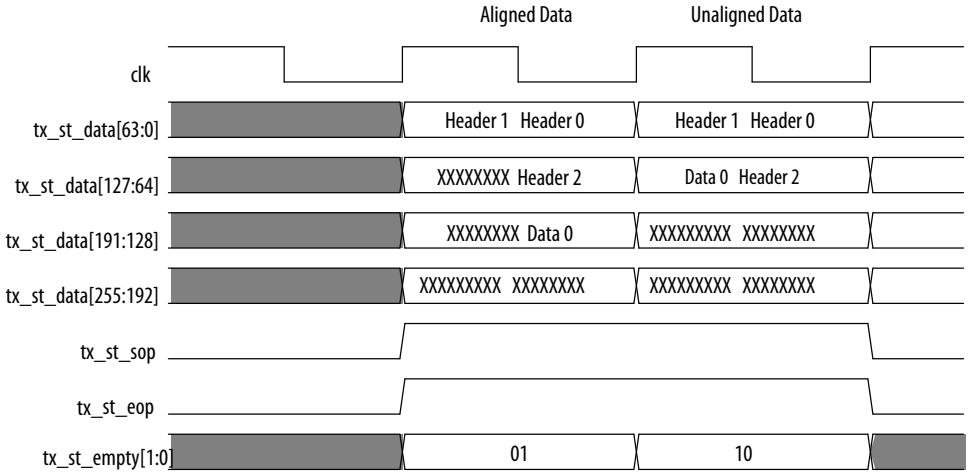
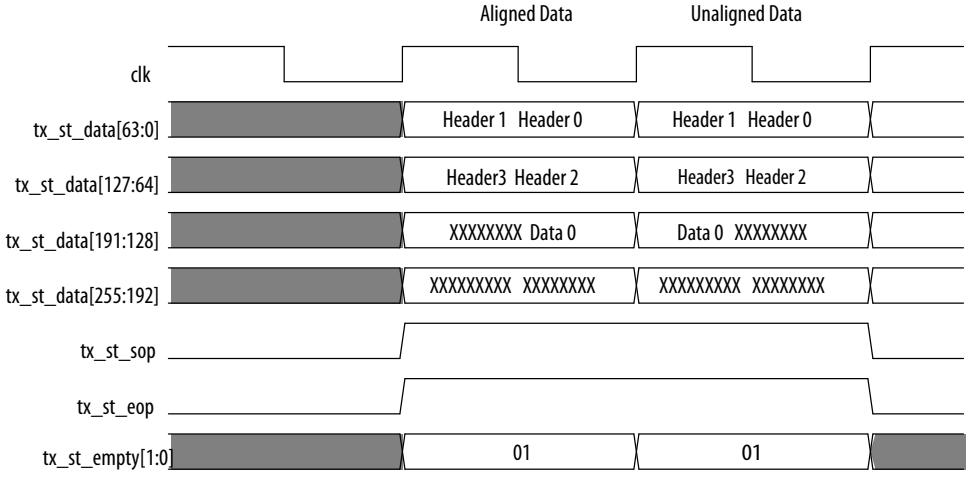


Figure 66. 256-Bit Avalon-ST tx_st_data Cycle Definition for 4-Dword Header TLP with Qword Addresses

The following figure illustrates the layout of header and data for a four dword header on a 256-bit bus with aligned and unaligned data.



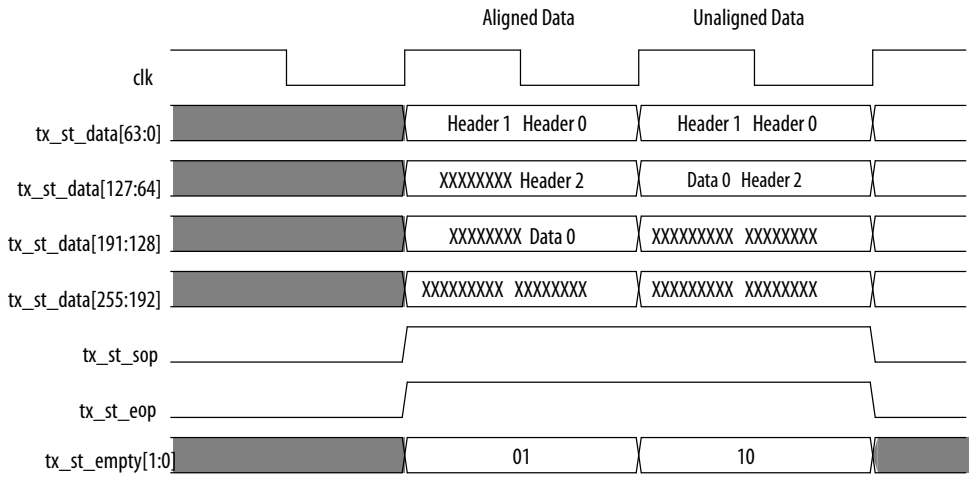
6.2.4.1 Single Packet Per Cycle

In single packer per cycle mode, all received TLPs start at the lower 128-bit boundary on a 256-bit Avalon-ST interface. Turn on **Enable Multiple Packets per Cycle** on the System Settings tab of the parameter editor to change multiple packets per cycle.

Single packet per cycle mode requires simpler Application Layer packet decode logic on the TX and RX paths because packets always start in the lower 128-bits of the Avalon-ST interface. Although this mode simplifies the Application Layer logic, failure to use the full 256-bit Avalon-ST may slightly reduce the throughput of a design.

The following figure illustrates the layout of header and data for a three dword header on a 256-bit bus with aligned and unaligned data.

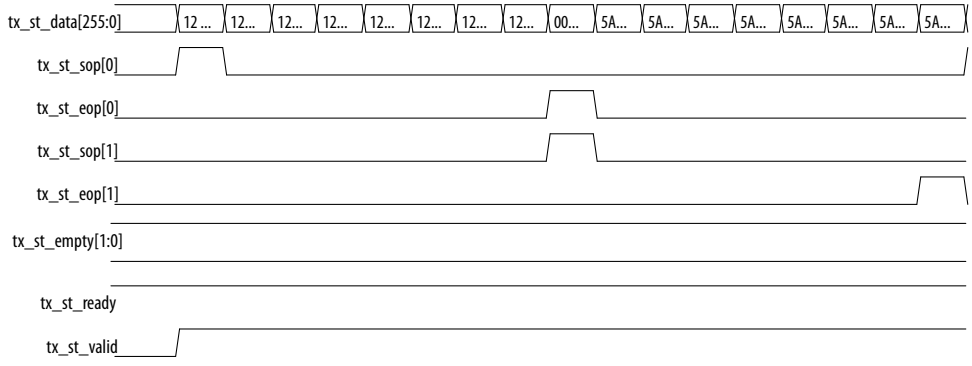
Figure 67. 256-Bit Avalon-ST tx_st_data Cycle Definition for 3-Dword Header TLP with Qword Addresses



6.2.4.2 Multiple Packets per Cycle on the Avalon-ST TX 256-Bit Interface

If you enable **Multiple Packets Per Cycle** under the **Systems Settings** heading, a TLP can start on a 128-bit boundary. This mode supports multiple start of packet and end of packet signals in a single cycle when the Avalon-ST interface is 256 bits wide. The following figure illustrates this mode for a 256-bit Avalon-ST TX interface. In this figure `tx_st_eop[0]` and `tx_st_sop[1]` are asserted in the same cycle. Using this mode increases the complexity of the Application Layer logic but results in higher throughput, depending on the TX traffic. Refer to *Tradeoffs to Consider when Enabling Multiple Packets per Cycle* for an example of the bandwidth when **Multiple Packets Per Cycle** is enabled and disabled.

Figure 68. 256-Bit Avalon-ST TX Interface with Multiple Packets Per Cycle



Related Links

Tradeoffs to Consider when Enabling Multiple Packets per Cycle on page 69

6.2.5 Root Port Mode Configuration Requests

If your Application Layer implements ECRC forwarding, it should not apply ECRC forwarding to Configuration Type 0 packets that it issues on the Avalon-ST interface. There should be no ECRC appended to the TLP, and the TD bit in the TLP header should be set to 0. These packets are processed internally by the Hard IP block and are not transmitted on the PCI Express link.

To ensure proper operation when sending Configuration Type 0 transactions in Root Port mode, the application should wait for the Configuration Type 0 transaction to be transferred to the Hard IP for PCI Express Configuration Space before issuing another packet on the Avalon-ST TX port. You can do this by waiting for the core to respond with a completion on the Avalon-ST RX port before issuing the next Configuration Type 0 transaction.

6.3 Clock Signals

Table 31. Clock Signals

Signal	Direction	Description
refclk	Input	Reference clock for the IP core. It must have the frequency specified under the **System Settings** heading in the parameter editor. This is a dedicated free running input clock to the dedicated REFCLK pin.
pld_clk	Input	Clocks the Application Layer. You can drive this clock with coreclkout_hip. If you drive pld_clk with another clock source, it must be equal to or faster than coreclkout_hip.
coreclkout_hip	Output	This is a fixed frequency clock used by the Data Link and Transaction Layers.

Related Links

Clocks on page 123

6.4 Reset, Status, and Link Training Signals

Refer to *Reset and Clocks* for more information about the reset sequence and a block diagram of the reset logic.

Table 32. Reset Signals

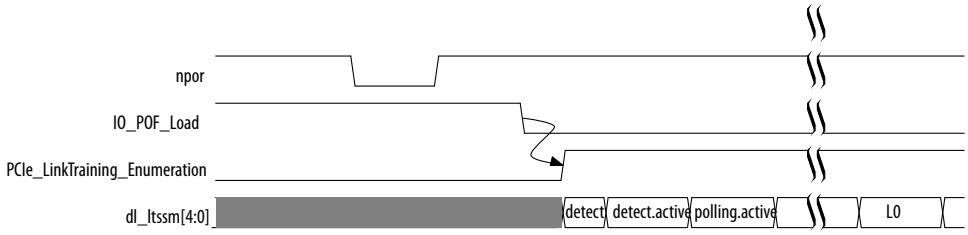
Signal	Direction	Description
npor	Input	Active low reset signal. In the Intel hardware example designs, npor is the OR of pin_perst and local_rstn coming from the software Application Layer. If you do not drive a soft reset signal from the Application Layer, this signal must be derived from pin_perst. You cannot disable this signal. Resets the entire IP Core and transceiver. Asynchronous.

continued...

Signal	Direction	Description
		This signal is *edge*, *not level* sensitive; consequently, you cannot use a low value on this signal to hold custom logic in reset. For more information about the reset controller, refer to *Reset*.
clr_st	Output	This optional reset signal has the same effect as reset_status. You enable this signal by turning **On** the **Enable Avalon-ST reset output port** in the parameter editor.
reset_status	Output	Active high reset status signal. When asserted, this signal indicates that the Hard IP clock is in reset. The reset_status signal is synchronous to the pld_clk clock and is deasserted only when the npor is deasserted and the Hard IP for PCI Express is not in reset (reset_status_hip = 0). You should use reset_status to drive the reset of your application. This reset is used for the Hard IP for PCI Express IP Core with the Avalon-ST interface.
pin_perst	Input	Active low reset from the PCIe reset pin of the device. pin_perst resets the datapath and control registers. Configuration over PCI Express (CvP) requires this signal. For more information about CvP refer to *Arria 10 CvP Initialization and Partial Reconfiguration over PCI Express User Guide*. Arria 10 devices can have up to 4 instances of the Hard IP for PCI Express. Each instance has its own pin_perst signal. *You must connect the* pin_perst *of each Hard IP instance to the corresponding* nPERST *pin of the device.* These pins have the following locations: • NPERSTL0: bottom left Hard IP and CvP blocks • NPERSTL1: top left Hard IP block • NPERSTR0: bottom right Hard IP block • NPERSTR1: top right Hard IP block For example, if you are using the Hard IP instance in the bottom left corner of the device, you must connect pin_perst to NPERSL0. For maximum use of the Arria 10 device, Intel recommends that you use the bottom left Hard IP first. This is the only location that supports CvP over a PCIe link. If your design does not require CvP, you may select other Hard IP blocks. Refer to the *Arria 10 GX, GT, and SX Device Family Pin Connection Guidelines* for more detailed information about these pins.

Figure 69. Reset and Link Training Timing Relationships

The following figure illustrates the timing relationship between npor and the LTSSM L0 state.



Note: To meet the 100 ms system configuration time, you must use the fast passive parallel configuration scheme with CvP and a 32-bit data width (FPP x32) or use the Arria 10 Hard IP for PCI Express in autonomous mode.

Table 33. Status and Link Training Signals

Signal	Direction	Description
serdes_pll_locked	Output	When asserted, indicates that the PLL that generates the coreclkout_hip clock signal is locked. In pipe simulation mode this signal is always asserted.
pld_core_ready	Input	When asserted, indicates that the Application Layer is ready for operation and is providing a stable clock to the pld_clk input. If the coreclkout_hip Hard IP output clock is sourcing the pld_clk Hard IP input, this input can be connected to the serdes_pll_locked output.
pld_clk_inuse	Output	When asserted, indicates that the Hard IP Transaction Layer is using the pld_clk as its clock and is ready for operation with the Application Layer. For reliable operation, hold the Application Layer in reset until pld_clk_inuse is asserted.
dlup	Output	When asserted, indicates that the Hard IP block is in the Data Link Control and Management State Machine (DLCMSM) DL_Up state.
dlup_exit	Output	This signal is asserted low for one pld_clk cycle when the IP core exits the DLCMSM DL_Up state, indicating that the Data Link Layer has lost communication with the other end of the PCIe link and left the Up state. When this pulse is asserted, the Application Layer should generate an internal reset signal that is asserted for at least 32 cycles.
ev128ns	Output	Asserted every 128 ns to create a time base aligned activity.
ev1us	Output	Asserted every 1µs to create a time base aligned activity.
hotrst_exit	Output	Hot reset exit. This signal is asserted for 1 clock cycle when the LTSSM exits the hot reset state. This signal should cause the Application Layer to be reset. This signal is active low. When this pulse is asserted, the Application Layer should generate an internal reset signal that is asserted for at least 32 cycles.
l2_exit	Output	L2 exit. This signal is active low and otherwise remains high. It is asserted for one cycle (changing value from 1 to 0 and back to 1) after the LTSSM transitions from l2.idle to detect. When this pulse is asserted, the Application Layer should generate an internal reset signal that is asserted for at least 32 cycles.
lane_act[3:0]	Output	Lane Active Mode: This signal indicates the number of lanes that configured during link training. The following encodings are defined: • 4'b0001: 1 lane • 4'b0010: 2 lanes • 4'b0100: 4 lanes • 4'b1000: 8 lanes
currentspeed[1:0]	Output	Indicates the current speed of the PCIe link. The following encodings are defined: • 2b'00: Undefined • 2b'01: Gen1 • 2b'10: Gen2 • 2b'11: Gen3
ltssmstate[4:0]	Output	LTSSM state: The LTSSM state machine encoding defines the following states: • 00000: Detect.Quiet • 00001: Detect.Active • 00010: Polling.Active • 00011: Polling.Compliance • 00100: Polling.Configuration • 00101: Polling.Speed • 00110: config.Linkwidthstart

continued...

Signal	Direction	Description
		• 00111: Config.Linkaccept • 01000: Config.Lanenumaccept • 01001: Config.Lanenumwait • 01010: Config.Complete • 01011: Config.Idle • 01100: Recovery.Rcvlock • 01101: Recovery.Rcvconfig • 01110: Recovery.Idle • 01111: L0 • 10000: Disable • 10001: Loopback.Entry • 10010: Loopback.Active • 10011: Loopback.Exit • 10100: Hot.Reset • 10101: LOs • 11001: L2.transmit.Wake • 11010: Speed.Recovery • 11011: Recovery.Equalization, Phase 0 • 11100: Recovery.Equalization, Phase 1 • 11101: Recovery.Equalization, Phase 2 • 11110: Recovery.Equalization, Phase 3

Related Links

- PCI Express Card Electromechanical Specification 2.0
- Arria 10 GX, GT, and SX Device Family Pin Connection Guidelines
  For information about connecting pins on the PCB including required resistor values and voltages.

6.5 ECRC Forwarding

On the Avalon-ST interface, the ECRC field follows the same alignment rules as payload data. For packets with payload, the ECRC is appended to the data as an extra dword of payload. For packets without payload, the ECRC field follows the address alignment as if it were a one dword payload. The position of the ECRC data for data depends on the address alignment. For packets with no payload data, the ECRC position corresponds to the position of `Data0`.

6.6 Error Signals

The following table describes the ECC error signals. These signals are all valid for one clock cycle. They are synchronous to `coreclkout_hip`.

ECC for the RX and retry buffers is implemented with MRAM. These error signals are flags. If a specific location of MRAM has errors, as long as that data is in the ECC decoder, the flag indicates the error.

When a correctable ECC error occurs, the Arria 10 Hard IP for PCI Express recovers without any loss of information. No Application Layer intervention is required. In the case of uncorrectable ECC error, Intel recommends that you reset the core.

Table 34. Error Signals

Signal	I/O	Description
derr_cor_ext_rcv0	Output	Indicates a corrected error in the RX buffer. This signal is for debug only. It is not valid until the RX buffer is filled with data. This is a pulse, not a level, signal. Internally, the pulse is generated with the 500 MHz clock. A pulse extender extends the signal so that the FPGA fabric running at 250 MHz can capture it. Because the error was corrected by the IP core, no Application Layer intervention is required. (1)
derr_rpl	Output	Indicates an uncorrectable error in the retry buffer. This signal is for debug only. (1)
derr_cor_ext_rpl0	Output	Indicates a corrected ECC error in the retry buffer. This signal is for debug only. Because the error was corrected by the IP core, no Application Layer intervention is required. (1)

Notes:
1. Debug signals are not rigorously verified and should only be used to observe behavior. Debug signals should not be used to drive logic custom logic.

Related Links

Avalon-ST RX Interface on page 58

6.7 Interrupts for Endpoints

Refer to *Interrupts* for detailed information about all interrupt mechanisms.

Table 35. Interrupt Signals for Endpoints

Signal	Direction	Description
app_msi_req	Input	Application Layer MSI request. Assertion causes an MSI posted write TLP to be generated based on the MSI configuration register values and the app_msi_tc and app_msi_num input ports.
app_msi_ack	Output	Application Layer MSI acknowledge. This signal acknowledges the Application Layer's request for an MSI interrupt.
app_msi_tc[2:0]	Input	Application Layer MSI traffic class. This signal indicates the traffic class used to send the MSI (unlike INTX interrupts, any traffic class can be used to send MSIs).
app_msi_num[4:0]	Input	MSI number of the Application Layer. This signal provides the low order message data bits to be sent in the message data field of MSI messages requested by app_msi_req. Only bits that are enabled by the MSI Message Control register apply.
app_int_sts	Input	Controls legacy interrupts. Assertion of app_int_sts causes an Assert_INTA message TLP to be generated and sent upstream. Deassertion of app_int_sts causes a Deassert_INTA message TLP to be generated and sent upstream.
app_int_ack	Output	This signal is the acknowledge for app_int_sts. It is asserted for at least one cycle either when either of the following events occur: • The Assert_INTA message TLP has been transmitted in response to the assertion of the app_int_sts. • The Deassert_INTA message TLP has been transmitted in response to the deassertion of the app_int_sts signal.

6.8 Interrupts for Root Ports

Table 36. Interrupt Signals for Root Ports

Signal	Direction	Description
int_status[3:0]	Output	These signals drive legacy interrupts to the Application Layer as follows: • int_status[0]: interrupt signal A • int_status[1]: interrupt signal B • int_status[2]: interrupt signal C • int_status[3]: interrupt signal D
serr_out	Output	System Error: This signal only applies to Root Port designs that report each system error detected, assuming the proper enabling bits are asserted in the Root Control and Device Control registers. If enabled, serr_out is asserted for a single clock cycle when a system error occurs. System errors are described in the *PCI Express Base Specification 2.1 or 3.0* in the Root Control register.

Related Links

PCI Express Base Specification 3.0

6.9 Completion Side Band Signals

The following table describes the signals that comprise the completion side band signals for the Avalon-ST interface. The Arria 10 Hard IP for PCI Express provides a completion error interface that the Application Layer can use to report errors, such as programming model errors. When the Application Layer detects an error, it can assert the appropriate cpl_err bit to indicate what kind of error to log. If separate requests result in two errors, both are logged. The Hard IP sets the appropriate status bits for the errors in the Configuration Space, and automatically sends error messages in accordance with the *PCI Express Base Specification*. Note that the Application Layer is responsible for sending the completion with the appropriate completion status value for non-posted requests. Refer to *Error Handling* for information on errors that are automatically detected and handled by the Hard IP.

For a description of the completion rules, the completion header format, and completion status field values, refer to Section 2.2.9 of the *PCI Express Base Specification*.

Table 37. Completion Signals for the Avalon-ST Interface

Signal	Direction	Description
cpl_err[6:0]	Input	Completion error. This signal reports completion errors to the Configuration Space. When an error occurs, the appropriate signal is asserted for one cycle. • cpl_err[0]: Completion timeout error with recovery. This signal should be asserted when a master-like interface has performed a non-posted request that never receives a corresponding completion transaction after the 50 ms timeout period when the error is correctable. The Hard IP automatically generates an advisory error message that is sent to the Root Complex. • cpl_err[1]: Completion timeout error without recovery. This signal should be asserted when a master-like interface has performed a non-posted request that never receives a corresponding completion transaction after the 50 ms time-out period when the error is not correctable. The Hard IP automatically generates a non-advisory error message that is sent to the Root Complex. • cpl_err[2]: Completer abort error. The Application Layer asserts this signal to respond to a non-posted request with a Completer Abort (CA) completion. The Application Layer generates and sends a completion packet with Completer Abort (CA) status to the requestor and then asserts this error signal to the Hard IP. The Hard IP automatically sets the error status bits in the Configuration Space register and sends error messages in accordance with the *PCI Express Base Specification*. • cpl_err[3]: Unexpected completion error. This signal must be asserted when an Application Layer master block detects an unexpected completion transaction. Many cases of unexpected completions are detected and reported internally by the Transaction Layer. For a list of these cases, refer to *Transaction Layer Errors*. • cpl_err[4]: Unsupported Request (UR) error for posted TLP. The Application Layer asserts this signal to treat a posted request as an Unsupported Request. The Hard IP automatically sets the error status bits in the Configuration Space register and sends error messages in accordance with the *PCI Express Base Specification*. Many cases of Unsupported Requests are detected and reported internally by the Transaction Layer. For a list of these cases, refer to *Transaction Layer Errors*. • cpl_err[5]: Unsupported Request error for non-posted TLP. The Application Layer asserts this signal to respond to a non-posted request with an Request (UR) completion. In this case, the Application Layer sends a completion packet with the Unsupported Request status back to the requestor, and asserts this error signal. The Hard IP automatically sets the error status bits in the Configuration Space Register and sends error messages in accordance with the *PCI Express Base Specification*. Many cases of Unsupported Requests are detected and reported internally by the Transaction Layer. For a list of these cases, refer to *Transaction Layer Errors*. • cpl_err[6]: Log header. If header logging is required, this bit must be set in the every cycle in which any of cpl_err[2], cpl_err[3], cpl_err[4], or cpl_err[5]is set. The Application Layer presents the header to the Hard IP by writing the following values to the following 4 registers using LMI before asserting cpl_err[6]:. The Application Layer presents the header to the Hard IP by writing the following values to the following 4 registers using LMI before asserting cpl_err[6]: — lmi_addr: 12'h81C, lmi_din: err_desc_func0[127:96] — lmi_addr: 12'h820, lmi_din: err_desc_func0[95:64] — lmi_addr: 12'h824, lmi_din: err_desc_func0[63:32] — lmi_addr: 12'h828, lmi_din: err_desc_func0[31:0]
cpl_pending	Input	Completion pending. The Application Layer must assert this signal when a master block is waiting for completion, for example, when a Non-Posted Request is pending. The state of this input is reflected by the Transactions Pending bit of the Device Status Register as defined in Section 7.8.5 of the *PCI Express Base Specification*.

Related Links

- Transaction Layer Errors on page 134
- PCI Express Base Specification Rev 3.0

6.10 Parity Signals

Parity protection provides some data protection in systems that do not use ECRC checking. Parity is odd. This option is not available for the Avalon-MM Arria 10 Hard IP for PCI Express.

On the RX datapath, parity is computed on the incoming TLP prior to the LCRC check in the Data Link Layer. Up to 32 parity bits are propagated to the Application Layer along with the RX Avalon-ST data. The RX datapath also propagates up to 32 parity bits to the Transaction Layer for Configuration TLPs. On the TX datapath, parity generated in the Application Layer is checked in Transaction Layer and the Data Link Layer.

The following table lists the signals that indicate parity errors. When an error is detected, parity error signals are asserted for one cycle.

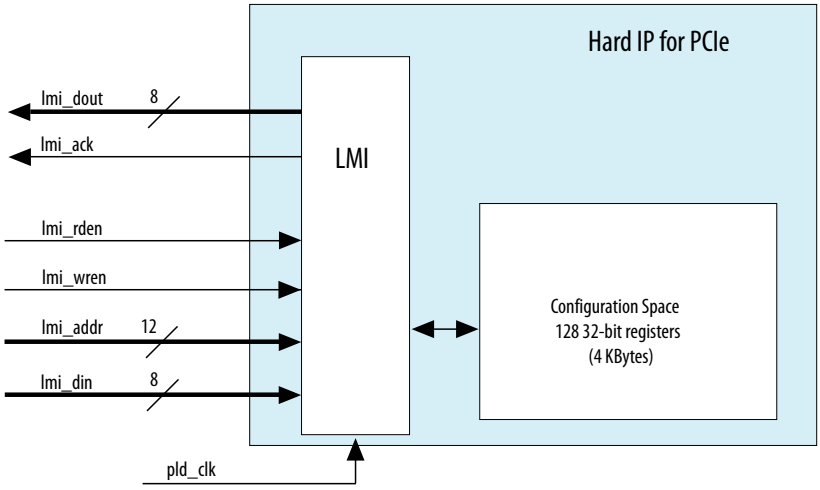
Table 38. Parity Signals

Signal Name	Direction	Description
`tx_par_err[1:0]`	Output	When asserted for a single cycle, indicates a parity error during TX TLP transmission. These errors are logged in the VSEC register. The following encodings are defined: • 2'b10: A parity error was detected by the TX Transaction Layer. The TLP is nullified and logged as an uncorrectable internal error in the VSEC registers. For more information, refer to *Uncorrectable Internal Error Status Register*. • 2'b01: Some time later, the parity error is detected by the TX Data Link Layer which drives 2'b01 to indicate the error. Intel recommends resetting the Arria 10 Hard IP for PCI Express when this error is detected. Contact Intel if resetting becomes unworkable. Note that not all simulation models assert the Transaction Layer error bit in conjunction with the Data Link Layer error bit.
`rx_par_err`	Output	When asserted for a single cycle, indicates that a parity error was detected in a TLP at the input of the RX buffer. This error is logged as an uncorrectable internal error in the VSEC registers. For more information, refer to *Uncorrectable Internal Error Status Register*. If this error occurs, you must reset the Hard IP if this error occurs because parity errors can leave the Hard IP in an unknown state.
`cfg_par_err`	Output	When asserted for a single cycle, indicates that a parity error was detected in a TLP that was routed to internal Configuration Space or to the Configuration Space Shadow Extension Bus. This error is logged as an uncorrectable internal error in the VSEC registers. For more information, refer to *Uncorrectable Internal Error Status Register*. If this error occurs, you must reset the core because parity errors can put the Hard IP in an unknown state.

6.11 LMI Signals

LMI interface is used to write log error descriptor information in the TLP header log registers. The LMI access to other registers is intended for debugging, not normal operation.

Figure 70. Local Management Interface



The LMI interface is synchronized to `pld_clk` and runs at frequencies up to 250 MHz. The LMI address is the same as the Configuration Space address. The LMI interface provides the same access to Configuration Space registers as Configuration TLP requests. Register bits have the same attributes, (read only, read/write, and so on) for accesses from the LMI interface and from Configuration TLP requests. The 32-bit read and write data is driven, LSB to MSB over 4 consecutive cycles.

Note: You can also use the Configuration Space signals to read Configuration Space registers. For more information, refer to *Transaction Layer Configuration Space Signals*.

When a LMI write has a timing conflict with configuration TLP access, the configuration TLP accesses have higher priority. LMI writes are held and executed when configuration TLP accesses are no longer pending. An acknowledge signal is sent back to the Application Layer when the execution is complete.

All LMI reads are also held and executed when no configuration TLP requests are pending. The LMI interface supports two operations: local read and local write. The timing for these operations complies with the Avalon-MM protocol described in the *Avalon Interface Specifications*. LMI reads can be issued at any time to obtain the contents of any Configuration Space register. LMI write operations are not recommended for use during normal operation. The Configuration Space registers are written by requests received from the PCI Express link and there may be unintended consequences of conflicting updates from the link and the LMI interface. LMI Write operations are provided for AER header logging, and debugging purposes only.

- In Root Port mode, do not access the Configuration Space using TLPs and the LMI bus simultaneously.

Table 39. LMI Interface

Signal	Direction	Description
lmi_dout[7:0]	Output	Data outputs. Data is driven from LSB, [7:0], to MSB,[31:24]. The LSB coincides with lmi_ack.
lmi_rden	Input	Read enable input.
lmi_wren	Input	Write enable input.
lmi_ack	Output	Write execution done/read data valid.
lmi_addr[11:0]	Input	Address inputs, [1:0] not used.
lmi_din[7:0]	Input	Data inputs. Data is driven from LSB, [7:0], to MSB,[31:24]. The LSB coincides with lim_wren.

Figure 71. LMI Read

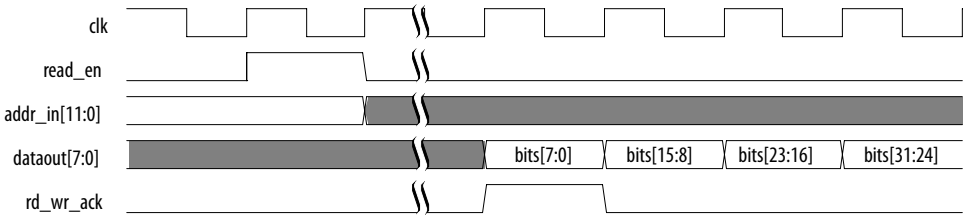
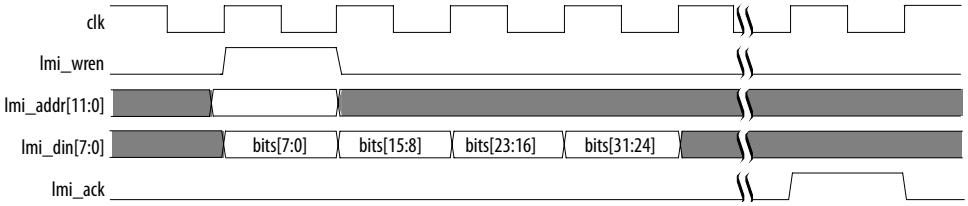


Figure 72. LMI Write

Only writable configuration bits are overwritten by this operation. Read-only bits are not affected. LMI write operations are not recommended for use during normal operation with the exception of AER header logging.



Related Links

Avalon Interface Specifications

For information about the Avalon-MM interfaces to implement read and write interfaces for master and slave components.

6.12 Transaction Layer Configuration Space Signals

Table 40. Configuration Space Signals

These signals are not available if Configuration Space Bypass mode is enabled.

Signal	Direction	Description
tl_cfg_add[3:0]	Output	Address of the register that has been updated. This signal is an index indicating which Configuration Space register information is being driven onto tl_cfg_ctl. The indexing is defined in *Multiplexed Configuration Register Information Available on tl_cfg_ctl*. The index increments every 8 pld_clk cycles
tl_cfg_ctl[31:0]	Output	The tl_cfg_ctl signal is multiplexed and contains the contents of the Configuration Space registers. The indexing is defined in *Multiplexed Configuration Register Information Available on tl_cfg_ctl*.
tl_cfg_sts[52:0]	Output	Configuration status bits. This information updates every pld_clk cycle. The following table provides detailed descriptions of the status bits.
hpg_ctrler[4:0]	Input	The hpg_ctrler signals are only available in Root Port mode and when the Slot capability register is enabled. Refer to the Slot register and Slot capability register parameters in Table 6–9 on page 6–10. For Endpoint variations the hpg_ctrler input should be hardwired to 0s. The bits have the following meanings:
	Input	• [0]: Attention button pressed. This signal should be asserted when the attention button is pressed. If no attention button exists for the slot, this bit should be hardwired to 0, and the Attention Button Present bit (bit[0]) in the Slot capability register parameter is set to 0.
	Input	• [1]: Presence detect. This signal should be asserted when a presence detect circuit detects a presence detect change in the slot.
	Input	• [2]: Manually-operated retention latch (MRL) sensor changed. This signal should be asserted when an MRL sensor indicates that the MRL is Open. If an MRL Sensor does not exist for the slot, this bit should be hardwired to 0, and the MRL Sensor Present bit (bit[2]) in the Slot capability register parameter is set to 0.
	Input	• [3]: Power fault detected. This signal should be asserted when the power controller detects a power fault for this slot. If this slot has no power controller, this bit should be hardwired to 0, and the Power Controller Present bit (bit[1]) in the Slot capability register parameter is set to 0.
	Input	• [4]: Power controller status. This signal is used to set the command completed bit of the Slot Status register. Power controller status is equal to the power controller control signal. If this slot has no power controller, this bit should be hardwired to 0 and the Power Controller Present bit (bit[1]) in the Slot capability register is set to 0.

Table 41. Mapping Between tl_cfg_sts and Configuration Space Registers

tl_cfg_sts	Configuration Space Register	Description
[52:49]	Device Status Register[3:0]	Records the following errors: • Bit 3: unsupported request detected • Bit 2: fatal error detected • Bit 1: non-fatal error detected • Bit 0: correctable error detected
[48]	Slot Status Register[8]	Data Link Layer state changed

continued...

tl_cfg_sts	Configuration Space Register	Description
[47]	Slot Status Register[4]	Command completed. (The hot plug controller completed a command.)
[46:31]	Link Status Register[15:0]	Records the following link status information: • Bit 15: link autonomous bandwidth status • Bit 14: link bandwidth management status • Bit 13: Data Link Layer link active - This bit is only available for Root Ports. It is always 0 for Endpoints. • Bit 12: Slot clock configuration • Bit 11: Link Training • Bit 10: Undefined • Bits[9:4]: Negotiated Link Width • Bits[3:0] Link Speed
[30]	Link Status 2 Register[0]	Current de-emphasis level.
[29:25]	Status Register[15:11]	Records the following 5 primary command status errors: • Bit 15: detected parity error • Bit 14: signaled system error • Bit 13: received master abort • Bit 12: received target abort • Bit 11: signaled target abort
[24]	Secondary Status Register[8]	Master data parity error
[23:6]	Root Status Register[17:0]	Records the following PME status information: • Bit 17: PME pending • Bit 16: PME status • Bits[15:0]: PME request ID[15:0]
[5:1]	Secondary Status Register[15:11]	Records the following 5 secondary command status errors: • Bit 15: detected parity error • Bit 14: received system error • Bit 13: received master abort • Bit 12: received target abort • Bit 11: signaled target abort
[0]	Secondary Status Register[8]	Master Data Parity Error

6.12.1 Configuration Space Register Access Timing

The signals of the `tl_cfg_*` interface include multi-cycle paths. Depending on the parameterization, the `tl_cfg_add` and `tl_cfg_ctl` signals update every four or eight `coreclkout_hip` cycles.

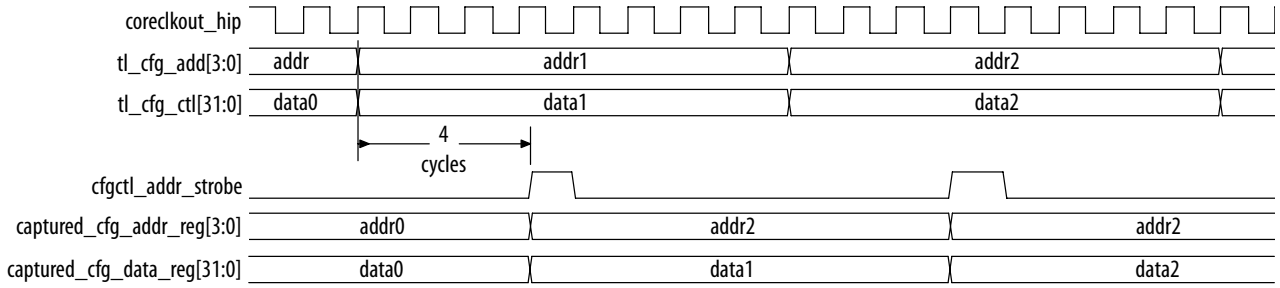
To ensure correct values are captured, your Application RTL must include code to force sampling to the middle of this window. The RTL shown below detects the change of address. A new strobe signal, `cfgctl_addr_strobe`, forces sampling to the middle of the window.

```
// detect the address transition
      always @(posedge coreclkout_hip)
      begin
      // detect address change
      cfgctl_addr_change <= cfg_addr_reg[3:0] != tl_cfg_add[3:0];
      // delay two clocks and use as strobe to sample the input 32-bit data
      cfgctl_addr_change2 <= cfgctl_addr_change;
```

```
      cfgctl_addr_strobe <= cfgctl_addr_change2;
      end
      // captured cfg ctl addr/data bus with the strobe
      always @(posedge coreclkout_hip)
      if(cfgctl_addr_strobe)
      begin
      captured_cfg_addr_reg[3:0] <= tl_cfg_add[3:0];
      captured_cfg_data_reg[31:0] <= tl_cfg_ctl[31:0];
      end
```

Note: Before Quartus Prime version 16.0.1, the multi-cycle paths did not include proper timing constraints. If you use this interface, you must upgrade to 16.0.1 or later to ensure proper sampling of the `tl_cfg_ctl` bus.

Figure 73. Sample tl_cfg_ctl in the Middle of Eight-Cycle Window



6.12.2 Configuration Space Register Access

The `tl_cfg_ctl` signal is a multiplexed bus that contains the contents of Configuration Space registers as shown in the figure below. Information stored in the Configuration Space is accessed in round robin order where `tl_cfg_add` indicates which register is being accessed. The following table shows the layout of configuration information that is multiplexed on `tl_cfg_ctl`.

Figure 74. Multiplexed Configuration Register Information Available on tl_cfg_ctl

Fields in blue are available only for Root Ports.

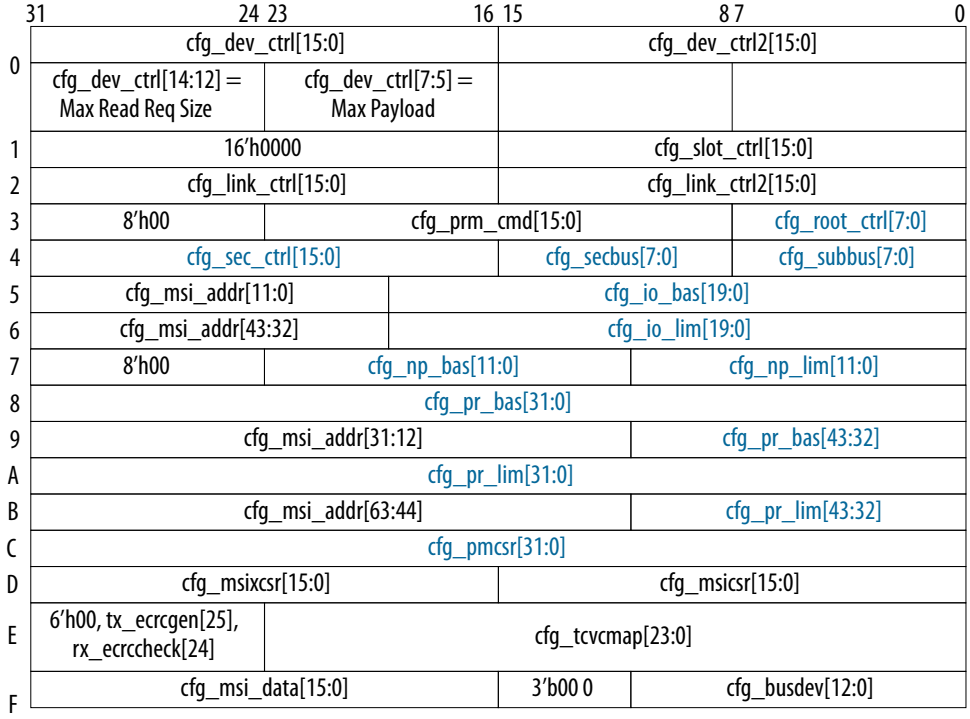


Table 42. Configuration Space Register Descriptions

Register	Width	Direction	Description
cfg_dev_ctrl	16	Output	cfg_devctrl[15:0] is Device Control for the PCI Express capability structure.
cfg_dev_ctrl2	16	Output	cfg_dev2ctrl[15:0] is Device Control 2 for the PCI Express capability structure.
cfg_slot_ctrl	16	Output	cfg_slot_ctrl[15:0] is the Slot Status of the PCI Express capability structure. This register is only available in Root Port mode.
cfg_link_ctrl	16	Output	cfg_link_ctrl[15:0]is the primary Link Control of the PCI Express capability structure. For Gen2 or Gen3 operation, you must write a 1'b1 to the Retrain Link bit (Bit[5] of the cfg_link_ctrl) of the Root Port to initiate retraining to a higher data rate after the initial link training to Gen1 L0 state. Retraining directs the Link Training and Status State Machine (LTSSM) to the Recovery state. Retraining to a higher data rate is not automatic for the Arria 10 Hard IP for PCI Express IP Core even if both devices on the link are capable of a higher data rate.
cfg_link_ctrl2	16	Output	cfg_link_ctrl2[31:16] is the secondary Link Control register of the PCI Express capability structure for Gen2 operation.

continued...

Register	Width	Direction	Description
			When `tl_cfg_addr=4'b0010`, `tl_cfg_ctl` returns the primary and secondary Link Control registers, `{ {cfg_link_ctrl[15:0], cfg_link_ctrl2[15:0]}`. The primary Link Status register contents are available on `tl_cfg_sts[46:31]`. For Gen1 variants, the link bandwidth notification bit is always set to 0. For Gen2 variants, this bit is set to 1.
`cfg_prm_cmd`	16	Output	Base/Primary Command register for the PCI Configuration Space.
`cfg_root_ctrl`	8	Output	Root control and status register of the PCI Express capability. This register is only available in Root Port mode.
`cfg_sec_ctrl`	16	Output	Secondary bus Control and Status register of the PCI Express capability. This register is available only in Root Port mode.
`cfg_secbus`	8	Output	Secondary bus number. This register is available only in Root Port mode.
`cfg_subbus`	8	Output	Subordinate bus number. This register is available only in Root Port mode.
`cfg_msi_addr`	64	Output	`cfg_msi_add[63:32]` is the message signaled interrupt (MSI) upper message address. `cfg_msi_add[31:0]` is the MSI message address.
`cfg_io_bas`	20	Output	The upper 20 bits of the I/O limit registers of the Type1 Configuration Space. This register is only available in Root Port mode.
`cfg_io_lim`	20	Output	The upper 20 bits of the IO limit registers of the Type1 Configuration Space. This register is only available in Root Port mode.
`cfg_np_bas`	12	Output	The upper 12 bits of the memory base register of the Type1 Configuration Space. This register is only available in Root Port mode.
`cfg_np_lim`	12	Output	The upper 12 bits of the memory limit register of the Type1 Configuration Space. This register is only available in Root Port mode.
`cfg_pr_bas`	44	Output	The upper 44 bits of the prefetchable base registers of the Type1 Configuration Space. This register is only available in Root Port mode.
`cfg_pr_lim`	44	Output	The upper 44 bits of the prefetchable limit registers of the Type1 Configuration Space. Available in Root Port mode.
`cfg_pmcsr`	32	Output	`cfg_pmcsr[31:16]` is Power Management Control and `cfg_pmcsr[15:0]`is the Power Management Status register.
`cfg_msixcsr`	16	Output	MSI-X message control.
`cfg_msicsr`	16	Output	MSI message control. Refer to the following table for the fields of this register.

continued...

Register	Width	Direction	Description
cfg_tcvcmap	24	Output	Configuration traffic class (TC)/virtual channel (VC) mapping. The Application Layer uses this signal to generate a TLP mapped to the appropriate channel based on the traffic class of the packet. • cfg_tcvcmap[2:0]: Mapping for TC0 (always 0). • cfg_tcvcmap[5:3]: Mapping for TC1. • cfg_tcvcmap[8:6]: Mapping for TC2. • cfg_tcvcmap[11:9]: Mapping for TC3. • cfg_tcvcmap[14:12]: Mapping for TC4. • cfg_tcvcmap[17:15]: Mapping for TC5. • cfg_tcvcmap[20:18]: Mapping for TC6. • cfg_tcvcmap[23:21]: Mapping for TC7.
cfg_msi_data	16	Output	cfg_msi_data[15:0] is message data for MSI.
cfg_busdev	13	Output	Bus/Device Number captured by or programmed in the Hard IP.

Figure 75. Configuration MSI Control Status Register

Field and Bit Map					
15 9	8	7	6 4	3 1	0
reserved	mask capability	64-bit address capability	multiple message enable	multiple message capable	MSI enable

Table 43. Configuration MSI Control Status Register Field Descriptions

Bit(s)	Field	Description
[15:9]	Reserved	N/A
[8]	mask capability	Per-vector masking capable. This bit is hardwired to 0 because the function does not support the optional MSI per-vector masking using the Mask_Bits and Pending_Bits registers defined in the *PCI Local Bus Specification*. Per-vector masking can be implemented using Application Layer registers.
[7]	64-bit address capability	64-bit address capable. • 1: function capable of sending a 64-bit message address • 0: function not capable of sending a 64-bit message address
[6:4]	multiple message enable	This field indicates permitted values for MSI signals. For example, if "100" is written to this field 16 MSI signals are allocated. • 3'b000: 1 MSI allocated • 3'b001: 2 MSI allocated • 3'b010: 4 MSI allocated • 3'b011: 8 MSI allocated • 3'b100: 16 MSI allocated • 3'b101: 32 MSI allocated • 3'b110: Reserved • 3'b111: Reserved
[3:1]	multiple message capable	This field is read by system software to determine the number of requested MSI messages. • 3'b000: 1 MSI requested • 3'b001: 2 MSI requested • 3'b010: 4 MSI requested • 3'b011: 8 MSI requested

continued...

Bit(s)	Field	Description
		• 3'b100: 16 MSI requested • 3'b101: 32 MSI requested • 3'b110: Reserved
[0]	MSI Enable	If set to 0, this component is not permitted to use MSI.

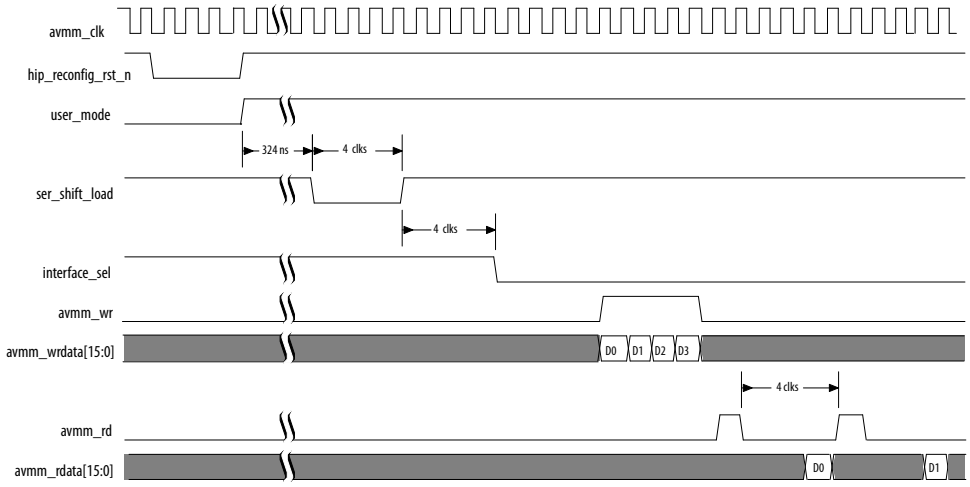
6.13 Hard IP Reconfiguration Interface

The Hard IP reconfiguration interface is an Avalon-MM slave interface with a 10-bit address and 16-bit data bus. You can use this bus to dynamically modify the value of configuration registers that are read-only at run time. To ensure proper system operation, reset or repeat device enumeration of the PCI Express link after changing the value of read-only configuration registers of the Hard IP.

Table 44. Hard IP Reconfiguration Signals

Signal	Direction	Description
hip_reconfig_clk	Input	Reconfiguration clock. The frequency range for this clock is 100–125 MHz.
hip_reconfig_rst_n	Input	Active-low Avalon-MM reset. Resets all of the dynamic reconfiguration registers to their default values as described in *Hard IP Reconfiguration Registers*.
hip_reconfig_address[9:0]	Input	The 10-bit reconfiguration address.
hip_reconfig_read	Input	Read signal. This interface is not pipelined. You must wait for the return of the hip_reconfig_readdata[15:0] from the current read before starting another read operation.
hip_reconfig_readdata[15:0]	Output	16-bit read data. hip_reconfig_readdata[15:0] is valid on the third cycle after the assertion of hip_reconfig_read.
hip_reconfig_write	Input	Write signal.
hip_reconfig_writedata[15:0]	Input	16-bit write model.
hip_reconfig_byte_en[1:0]	Input	Byte enables, currently unused.
ser_shift_load	Input	You must toggle this signal once after changing to user mode before the first access to read-only registers. This signal should remain asserted for a minimum of 324 ns after switching to user mode.
interface_sel	Input	A selector which must be asserted when performing dynamic reconfiguration. Drive this signal low 4 clock cycles after the release of ser_shif t_load.

Figure 76. Hard IP Reconfiguration Bus Timing of Read-Only Registers



For a detailed description of the Avalon-MM protocol, refer to the *Avalon Memory Mapped Interfaces* chapter in the *Avalon Interface Specifications*.

Related Links

Avalon Interface Specifications
For information about the Avalon-MM interfaces to implement read and write interfaces for master and slave components.

6.14 Power Management Signals

Table 45. Power Management Signals

Signal	Direction	Description
pme_to_cr	Input	Power management turn off control register. Root Port—When this signal is asserted, the Root Port sends the PME_turn_off message. Endpoint—This signal is asserted to acknowledge the PME_turn_off message by sending pme_to_ack to the Root Port.
pme_to_sr	Output	Power management turn off status register. Root Port—This signal is asserted for 1 clock cycle when the Root Port receives the pme_turn_off acknowledge message. Endpoint—This signal is asserted for 1 cycle when the Endpoint receives the PME_turn_off message from the Root Port.
pm_event	Input	Power Management Event. This signal is only available for Endpoints.

continued...

Signal	Direction	Description
		The Endpoint initiates a a power_management_event message (PM_PME) that is sent to the Root Port. If the Hard IP is in a low power state, the link exits from the low-power state to send the message. This signal is positive edge-sensitive.
pm_data[9:0]	Input	Power Management Data. This bus indicates power consumption of the component. This bus can only be implemented if all three bits of AUX_power (part of the Power Management Capabilities structure) are set to 0. This bus includes the following bits: • pm_data[9:2]: Data Register: This register maintains a value associated with the power consumed by the component. (Refer to the example below) • pm_data[1:0]: Data Scale: This register maintains the scale used to find the power consumed by a particular component and can include the following values: • 2b'00: unknown • 2b'01: 0.1 × • 2b'10: 0.01 × • 2b'11: 0.001 × For example, the two registers might have the following values: • pm_data[9:2]: b'1110010 = 114 • pm_data[1:0]: b'10, which encodes a factor of 0.01 To find the maximum power consumed by this component, multiply the data value by the data Scale (114 × .01 = 1.14). 1.14 watts is the maximum power allocated to this component in the power state selected by the data_select field.
pm_auxpwr	Input	Power Management Auxiliary Power: This signal can be tied to 0 because the L2 power state is not supported.

Figure 77. Layout of Power Management Capabilities Register

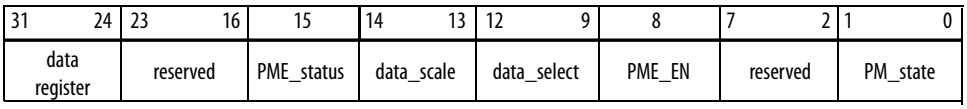


Table 46. Power Management Capabilities Register Field Descriptions

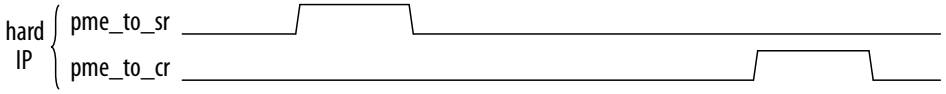
Bits	Field	Description
[31:24]	Data register	This field indicates in which power states a function can assert the PME# message.
[23:16]	reserved	—
[15]	PME_status	When set to 1, indicates that the function would normally assert the PME# message independently of the state of the PME_en bit.
[14:13]	data_scale	This field indicates the scaling factor when interpreting the value retrieved from the data register. This field is read-only.
[12:9]	data_select	This field indicates which data should be reported through the data register and the data_scale field.

continued...

Bits	Field	Description
[8]	PME_EN	1: indicates that the function can assert PME#0: indicates that the function cannot assert PME#
[7:2]	reserved	—
[1:0]	PM_state	Specifies the power management state of the operating condition being described. The following encodings are defined: • 2b'00 D0 • 2b'01 D1 • 2b'10 D2 • 2b'11 D3 A device returns 2b'11 in this field and Aux or PME Aux in the type register to specify the *D3-Cold PM* state. An encoding of 2b'11 along with any other type register value specifies the *D3-Hot* state.

Figure 78. pme_to_sr and pme_to_cr in an Endpoint IP core

The following figure illustrates the behavior of pme_to_sr and pme_to_cr in an Endpoint. First, the Hard IP receives the PME_turn_off message which causes pme_to_sr to assert. Then, the Application Layer sends the PME_to_ack message to the Root Port by asserting pme_to_cr.



6.15 Physical Layer Interface Signals

Intel provides an integrated solution with the Transaction, Data Link and Physical Layers. The IP Parameter Editor generates a SERDES variation file, <*variation*>_serdes.v or **.vhd** , in addition to the Hard IP variation file, <*variation*>.v or .vhd. The SERDES entity is included in the library files for PCI Express.

6.15.1 Serial Data Signals

This differential, serial interface is the physical link between a Root Port and an Endpoint.

The PCIe IP Core supports 1, 2, 4, or 8 lanes. Each lane includes a TX and RX differential pair. Data is striped across all available lanes.

Table 47. 1-Bit Interface Signals

The following table shows the signals for the x8 IP core.

Signal	Direction	Description
tx_out[7:0]	Output	Transmit output. These signals are the serial outputs of lanes 7–0.
rx_in[7:0]	Input	Receive input. These signals are the serial inputs of lanes 7–0.

Refer to *Pin-out Files for Intel Devices* for pin-out tables for all Intel devices in **.pdf**, **.txt**, and **.xls** formats.

Transceiver channels are arranged in groups of six. For GX devices, the lowest six channels on the left side of the device are labeled GXB_L0, the next group is GXB_L1, and so on. Channels on the right side of the device are labeled GXB_R0, GXB_R1, and

so on. Be sure to connect the Hard IP for PCI Express on the left side of the device to appropriate channels on the left side of the device, as specified in the *Pin-out Files for Intel Devices*.

Related Links

- Physical Layout of Hard IP In Arria 10 Devices on page 48
  Arria 10 devices include 1–4 hard IP blocks for PCI Express.
- Pin-out Files for Intel Devices

6.15.2 PIPE Interface Signals

These PIPE signals are available for Gen1, Gen2, and Gen3 variants so that you can simulate using either the serial or the PIPE interface. Simulation is much faster using the PIPE interface because the PIPE simulation bypasses the SERDES model . By default, the PIPE interface is 8 bits for Gen1 and Gen2 and 32 bits for Gen3. You can use the PIPE interface for simulation even though your actual design includes a serial interface to the internal transceivers. However, it is not possible to use the Hard IP PIPE interface in hardware, including probing these signals using SignalTap® II Embedded Logic Analyzer.

Note: The Intel Root Port BFM bypasses Gen3 Phase 2 and Phase 3 Equalization. However, Gen3 variants can perform Phase 2 and Phase 3 equalization if instructed by a third-party BFM.

In the following table, signals that include lane number 0 also exist for lanes 1-7. For Gen1 and Gen2 operation, Gen3 outputs can be left floating.

Table 48. PIPE Interface Signals

Signal	Direction	Description
txdata0[31:0]	Output	Transmit data <*n*>. This bus transmits data on lane <*n*>.
txdatak0[3:0]	Output	Transmit data control <*n*>. This signal serves as the control bit for txdata <*n*>. Bit 0 corresponds to the lowest-order byte of txdata, and so on. A value of 0 indicates a data byte. A value of 1 indicates a control byte. For Gen1 and Gen2 only.
txblkst0	Output	For Gen3 operation, indicates the start of a block in the transmit direction.
txcompl0	Output	Transmit compliance <*n*>. This signal forces the running disparity to negative in Compliance Mode (negative COM character).
txdataskip0	Output	For Gen3 operation. Allows the MAC to instruct the TX interface to ignore the TX data interface for one clock cycle. The following encodings are defined: • 1'b0: TX data is invalid • 1'b1: TX data is valid
txdeemph0	Output	Transmit de-emphasis selection. The Arria 10 Hard IP for PCI Express sets the value for this signal based on the indication received from the other end of the link during the Training Sequences (TS). You do not need to change this value.
txdetectrx0	Output	Transmit detect receive <*n*>. This signal tells the PHY layer to start a receive detection operation or to begin loopback.
txelecidle0	Output	Transmit electrical idle <*n*>. This signal forces the TX output to electrical idle.

continued...

Signal	Direction	Description
txswing	Output	When asserted, indicates full swing for the transmitter voltage. When deasserted indicates half swing.
txmargin[2:0]	Output	Transmit V_{OD} margin selection. The value for this signal is based on the value from the Link Control 2 Register. Available for simulation only.
txsynchd0[1:0]	Output	For Gen3 operation, specifies the transmit block type. The following encodings are defined: • 2'b01: Ordered Set Block • 2'b10: Data Block Designs that do not support Gen3 can let this signal float.
rxdata0[31:0]	Input	Receive data <*n*>. This bus receives data on lane <*n*>.
rxdatak[3:0]	Input	Receive data >*n*>. This bus receives data on lane <*n*>. Bit 0 corresponds to the lowest-order byte of rxdata, and so on. A value of 0 indicates a data byte. A value of 1 indicates a control byte. For Gen1 and Gen2 only.
rxblkst0	Input	For Gen3 operation, indicates the start of a block in the receive direction.
rxdataskip0	Output	For Gen3 operation. Allows the PCS to instruct the RX interface to ignore the RX data interface for one clock cycle. The following encodings are defined: • 1'b0: RX data is invalid • 1'b1: RX data is valid
rxelecidle0	Input	Receive electrical idle <*n*>. When asserted, indicates detection of an electrical idle.
rxpolarity0	Output	Receive polarity <*n*>. This signal instructs the PHY layer to invert the polarity of the 8B/10B receiver decoding block.
rxstatus0[2:0]	Input	Receive status <*n*>. This signal encodes receive status, including error codes for the receive data stream and receiver detection.
rxsynchd0[1:0]	Input	For Gen3 operation, specifies the receive block type. The following encodings are defined: • 2'b01: Ordered Set Block • 2'b10: Data Block Designs that do not support Gen3 can ground this signal.
rxvalid0	Input	Receive valid <*n*>. This signal indicates symbol lock and valid data on rxdata<*n*> and rxdatak <*n*>.
phystatus0	Input	PHY status <*n*>. This signal communicates completion of several PHY requests.
powerdown0[1:0]	Output	Power down <*n*>. This signal requests the PHY to change its power state to the specified state (P0, P0s, P1, or P2).
currentcoeff0[17:0]	Output	For Gen3, specifies the coefficients to be used by the transmitter. The 18 bits specify the following coefficients: • [5:0]: C_{-1} • [11:6]: C_0 • [17:12]: C_{+1}
currentrxpreset0[2:0]	Output	For Gen3 designs, specifies the current preset.
simu_mode_pipe	Input	When set to 1, the PIPE interface is in simulation mode.
sim_pipe_rate[1:0]	Output	The 2-bit encodings have the following meanings: • 2'b00: Gen1 rate (2.5 Gbps) • 2'b01: Gen2 rate (5.0 Gbps) • 2'b10: Gen3 rate (8.0 Gbps)

continued...

Signal	Direction	Description
rate[1:0]	Output	The 2-bit encodings have the following meanings: • 2'b00: Gen1 rate (2.5 Gbps) • 2'b01: Gen2 rate (5.0 Gbps) • 2'b1X: Gen3 rate (8.0 Gbps)
sim_pipe_pclk_in	Input	This clock is used for PIPE simulation only, and is derived from the refclk. It is the PIPE interface clock used for PIPE mode simulation.
sim_pipe_ltssmstate0[4:0]	Input and Output	LTSSM state: The LTSSM state machine encoding defines the following states: • 5'b00000: Detect.Quiet • 5'b00001: Detect.Active • 5'b00010: Polling.Active • 5'b 00011: Polling.Compliance • 5'b 00100: Polling.Configuration • 5'b00101: Polling.Speed • 5'b00110: config.LinkwidthsStart • 5'b 00111: Config.Linkaccept • 5'b 01000: Config.Lanenumaccept • 5'b01001: Config.Lanenumwait • 5'b01010: Config.Complete • 5'b 01011: Config.Idle • 5'b01100: Recovery.Rcvlock • 5'b01101: Recovery.Rcvconfig • 5'b01110: Recovery.Idle • 5'b 01111: L0 • 5'b10000: Disable • 5'b10001: Loopback.Entry • 5'b10010: Loopback.Active • 5'b10011: Loopback.Exit • 5'b10100: Hot.Reset • 5'b10101: L0s • 5'b11001: L2.transmit.Wake • 5'b11010: Recovery.Speed • 5'b11011: Recovery.Equalization, Phase 0 • 5'b11100: Recovery.Equalization, Phase 1 • 5'b11101: Recovery.Equalization, Phase 2 • 5'b11110: Recovery.Equalization, Phase 3 • 5'b11111: Recovery.Equalization, Done
rxfreqlocked0	Input	When asserted indicates that the pclk_in used for PIPE simulation is valid.
eidleinfersel0[2:0]	Output	Electrical idle entry inference mechanism selection. The following encodings are defined: • 3'b0xx: Electrical Idle Inference not required in current LTSSM state • 3'b100: Absence of COM/SKP Ordered Set in the 128 us window for Gen1 or Gen2 • 3'b101: Absence of TS1/TS2 Ordered Set in a 1280 UI interval for Gen1 or Gen2 • 3'b110: Absence of Electrical Idle Exit in 2000 UI interval for Gen1 and 16000 UI interval for Gen2 • 3'b111: Absence of Electrical idle exit in 128 us window for Gen1

6.15.3 Test Signals

Table 49. Test Interface Signals

The `test_in` bus provides run-time control and monitoring of the internal state of the IP core.

Signal	Direction	Description
`test_in[31:0]`	Input	The bits of the `test_in` bus have the following definitions. Set this bus to 0x00000188. • [0]: Simulation mode. This signal can be set to 1 to accelerate initialization by reducing the value of many initialization counters. • [1]: Reserved. Must be set to 1'b0. • [2]: Descramble mode disable. This signal must be set to 1 during initialization in order to disable data scrambling. You can use this bit in simulation for Gen1 and Gen2 Endpoints and Root Ports to observe descrambled data on the link. Descrambled data cannot be used in open systems because the link partner typically scrambles the data. • [4:3]: Reserved. Must be set to 2'b01. • [5]: Compliance test mode. Set this bit to 1'b0. Setting this bit to 1'b1 prevents the LTSSM from entering compliance mode. Toggling this bit controls the entry and exit from the compliance state, enabling the transmission of Gen1, Gen2 and Gen3 compliance patterns. • [6]: Forces entry to compliance mode when a timeout is reached in the polling.active state and not all lanes have detected their exit condition. • [7]: Disable low power state negotiation. Intel recommends setting this bit. • [8]: Set this bit to 1'b1. • [31:9]: Reserved. Set to all 0s.
`testin_zero`	Output	When asserted, indicates accelerated initialization for simulation is active.
`lane_act[3:0]`	Output	Lane Active Mode: This signal indicates the number of lanes that configured during link training. The following encodings are defined: • 4'b0001: 1 lane • 4'b0010: 2 lanes • 4'b:0100: 4 lanes • 4'b:1000: 8 lanes

6.15.4 Arria 10 Development Kit Conduit Interface

The Arria 10 Development Kit conduit interface signals are optional signals that allow you to connect your design to the Arria 10 FPGA Development Kit. Enable this interface by selecting **Enable Arria 10 FPGA Development Kit connection** on the **Configuration, Debug, and Extension Options** tab of the component GUI. The `devkit_status` output port includes signals useful for debugging.

Table 50.

Signal Name	Direction	Description
`devkit_status[255:0]`	Output	The `devkit_status[255:0]` bus comprises the following status signals : • `devkit_status[1:0]: current_speed` • `devkit_status[2]: derr_cor_ext_rcv` • `devkit_status[3]: derr_cor_ext_rpl` • `devkit_status[4]: derr_err` • `devkit_status[5]: rx_par_err` • `devkit_status[7:6]: tx_par_err` • `devkit_status[8]: cfg_par_err` • `devkit_status[9]: dlup`

continued...

Signal Name	Direction	Description
		• devkit_status[10]: dlup_exit • devkit_status[11]: ev128ns • devkit_status[12]: ev1us • devkit_status[13]: hotrst_exit • devkit_status[17:14]: int_status[3:0] • devkit_status[18]: l2_exit • devkit_status[22:19]: lane_act[3:0] • devkit_status[27:23]: ltssmstate[4:0] • devkit_status[35:28]: ko_cpl_spc_header[7:0] • devkit_status[47:36]: ko_cpl_spc_data[11:0] • devkit_status[48]: rxfc_cplbuf_ovf • devkit_status[49]: reset_status • devkit_status[255:50]: Reserved
devkit_ctrl[255:0]	Input	The devkit_ctrl[255:0] bus comprises the following status signals. You can optionally connect these pins to an on-board switch for PCI-SIG compliance testing, such as bypass compliance testing. • devkit_ctrl[0]:test_in[0] is typically set to 1'b0 • devkit_ctrl[4:1]:test_in[4:1] is typically set to 4'b0100 • devkit_ctrl[6:5]:test_in[6:5] is typically set to 2'b01 • devkit_ctrl[31:7]:test_in[31:7] is typically set to 25'h3 • devkit_ctrl[63:32]:is typically set to 32'b0 • devkit_ctrl[255:64]:is typically set to 192'b0

7 Registers

7.1 Correspondence between Configuration Space Registers and the PCIe Specification

Table 51. Correspondence between Configuration Space Capability Structures and PCIe Base Specification Description

For the Type 0 and Type 1 Configuration Space Headers, the first line of each entry lists Type 0 values and the second line lists Type 1 values when the values differ.

Byte Address	Hard IP Configuration Space Register	Corresponding Section in PCIe Specification
0x000:0x03C	PCI Header Type 0 Configuration Registers	Type 0 Configuration Space Header
0x000:0x03C	PCI Header Type 1 Configuration Registers	Type 1 Configuration Space Header
0x040:0x04C	Reserved	N/A
0x050:0x05C	MSI Capability Structure	MSI Capability Structure
0x068:0x070	MSI-X Capability Structure	MSI-X Capability Structure
0x070:0x074	Reserved	N/A
0x078:0x07C	Power Management Capability Structure	PCI Power Management Capability Structure
0x080:0x0B8	PCI Express Capability Structure	PCI Express Capability Structure
0x0B8:0x0FC	Reserved	N/A
0x094:0x0FF	Root Port	N/A
0x100:0x16C	Virtual Channel Capability Structure (Reserved)	Virtual Channel Capability
0x170:0x17C	Reserved	N/A
0x180:0x1FC	Virtual channel arbitration table (Reserved)	VC Arbitration Table
0x200:0x23C	Port VC0 arbitration table (Reserved)	Port Arbitration Table
0x240:0x27C	Port VC1 arbitration table (Reserved)	Port Arbitration Table
0x280:0x2BC	Port VC2 arbitration table (Reserved)	Port Arbitration Table
0x2C0:0x2FC	Port VC3 arbitration table (Reserved)	Port Arbitration Table
0x300:0x33C	Port VC4 arbitration table (Reserved)	Port Arbitration Table
0x340:0x37C	Port VC5 arbitration table (Reserved)	Port Arbitration Table
0x380:0x3BC	Port VC6 arbitration table (Reserved)	Port Arbitration Table
0x3C0:0x3FC	Port VC7 arbitration table (Reserved)	Port Arbitration Table
0x400:0x7FC	Reserved	PCIe spec corresponding section name
0x800:0x834	Advanced Error Reporting AER (optional)	Advanced Error Reporting Capability
0x838:0xFFF	Reserved	N/A

continued...

Byte Address	Hard IP Configuration Space Register	Corresponding Section in PCIe Specification
Overview of Configuration Space Register Fields		
0x000	Device ID, Vendor ID	Type 0 Configuration Space Header Type 1 Configuration Space Header
0x004	Status, Command	Type 0 Configuration Space Header Type 1 Configuration Space Header
0x008	Class Code, Revision ID	Type 0 Configuration Space Header Type 1 Configuration Space Header
0x00C	BIST, Header Type, Primary Latency Timer, Cache Line Size	Type 0 Configuration Space Header Type 1 Configuration Space Header
0x010	Base Address 0	Base Address Registers
0x014	Base Address 1	Base Address Registers
0x018	Base Address 2 Secondary Latency Timer, Subordinate Bus Number, Secondary Bus Number, Primary Bus Number	Base Address Registers Secondary Latency Timer, Type 1 Configuration Space Header, Primary Bus Number
0x01C	Base Address 3 Secondary Status, I/O Limit, I/O Base	Base Address Registers Secondary Status Register ,Type 1 Configuration Space Header
0x020	Base Address 4 Memory Limit, Memory Base	Base Address Registers Type 1 Configuration Space Header
0x024	Base Address 5 Prefetchable Memory Limit, Prefetchable Memory Base	Base Address Registers Prefetchable Memory Limit, Prefetchable Memory Base
0x028	Reserved Prefetchable Base Upper 32 Bits	N/A Type 1 Configuration Space Header
0x02C	Subsystem ID, Subsystem Vendor ID Prefetchable Limit Upper 32 Bits	Type 0 Configuration Space Header Type 1 Configuration Space Header
0x030	Expansion ROM base address I/O Limit Upper 16 Bits, I/O Base Upper 16 Bits	Type 0 Configuration Space Header Type 1 Configuration Space Header
0x034	Reserved, Capabilities PTR	Type 0 Configuration Space Header Type 1 Configuration Space Header
0x038	Reserved Expansion ROM Base Address	N/A Type 1 Configuration Space Header
0x03C	Interrupt Pin, Interrupt Line Bridge Control, Interrupt Pin, Interrupt Line	Type 0 Configuration Space Header Type 1 Configuration Space Header
0x050	MSI-Message Control Next Cap Ptr Capability ID	MSI and MSI-X Capability Structures
0x054	Message Address	MSI and MSI-X Capability Structures
0x058	Message Upper Address	MSI and MSI-X Capability Structures
0x05C	Reserved Message Data	MSI and MSI-X Capability Structures
0x068	MSI-X Message Control Next Cap Ptr Capability ID	MSI and MSI-X Capability Structures
0x06C	MSI-X Table Offset BIR	MSI and MSI-X Capability Structures
0x070	Pending Bit Array (PBA) Offset BIR	MSI and MSI-X Capability Structures

continued...

Byte Address	Hard IP Configuration Space Register	Corresponding Section in PCIe Specification
0x078	Capabilities Register Next Cap PTR Cap ID	PCI Power Management Capability Structure
0x07C	Data PM Control/Status Bridge Extensions Power Management Status & Control	PCI Power Management Capability Structure
0x800	PCI Express Enhanced Capability Header	Advanced Error Reporting Enhanced Capability Header
0x804	Uncorrectable Error Status Register	Uncorrectable Error Status Register
0x808	Uncorrectable Error Mask Register	Uncorrectable Error Mask Register
0x80C	Uncorrectable Error Severity Register	Uncorrectable Error Severity Register
0x810	Correctable Error Status Register	Correctable Error Status Register
0x814	Correctable Error Mask Register	Correctable Error Mask Register
0x818	Advanced Error Capabilities and Control Register	Advanced Error Capabilities and Control Register
0x81C	Header Log Register	Header Log Register
0x82C	Root Error Command	Root Error Command Register
0x830	Root Error Status	Root Error Status Register
0x834	Error Source Identification Register Correctable Error Source ID Register	Error Source Identification Register

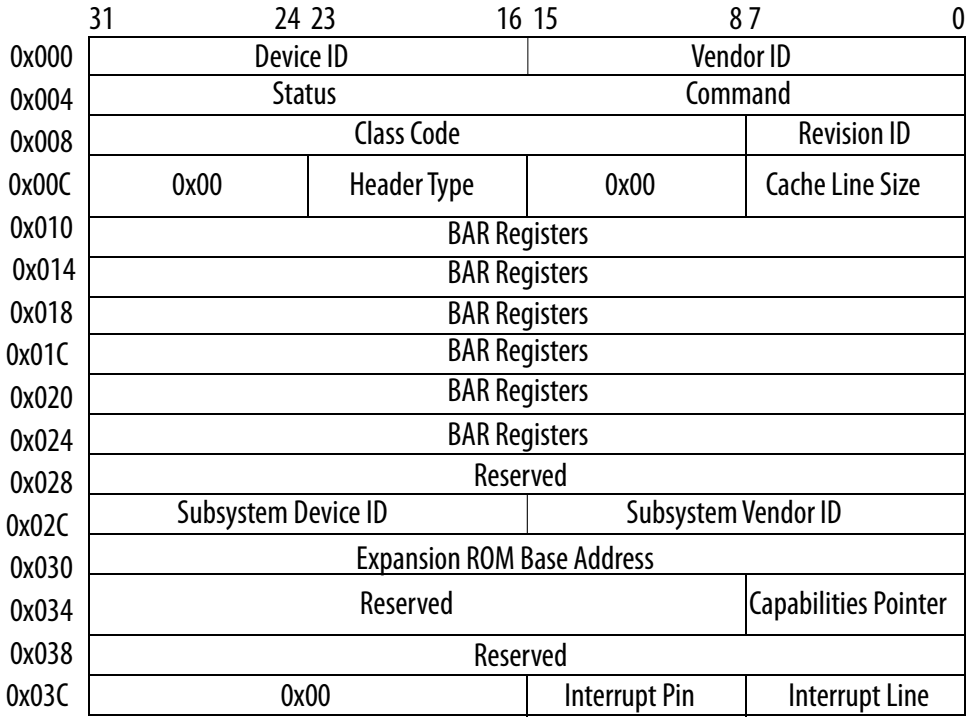
Related Links

PCI Express Base Specification 3.0

7.2 Type 0 Configuration Space Registers

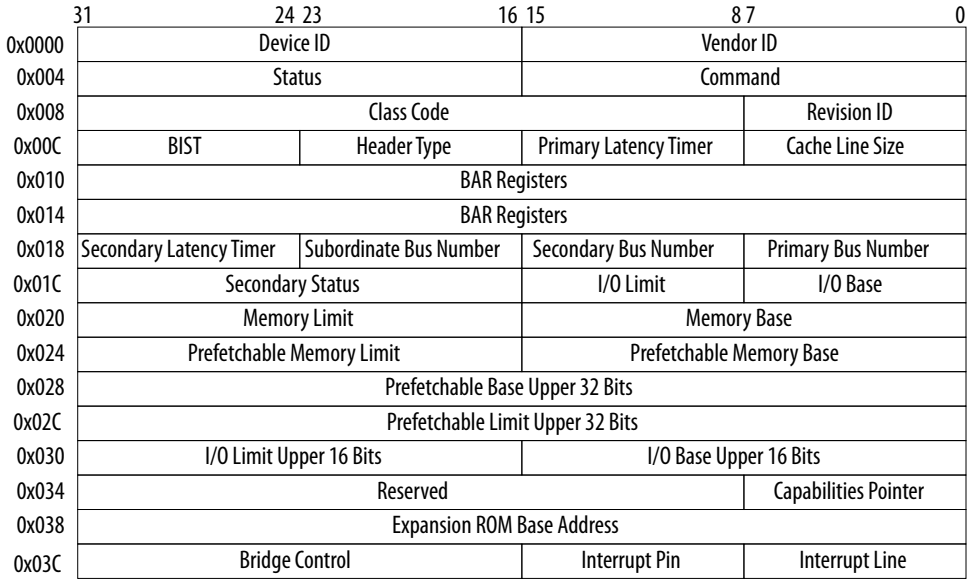
Figure 79. Type 0 Configuration Space Registers - Byte Address Offsets and Layout

Endpoints store configuration data in the Type 0 Configuration Space. The Correspondence between Configuration Space Registers and the PCIe Specification on page 107 lists the appropriate section of the *PCI Express Base Specification* that describes these registers.



7.3 Type 1 Configuration Space Registers

Figure 80. Type 1 Configuration Space Registers (Root Ports)



7.4 PCI Express Capability Structures

The layout of the most basic Capability Structures are provided below. Refer to the *PCI Express Base Specification* for more information about these registers.

Figure 81. MSI Capability Structure

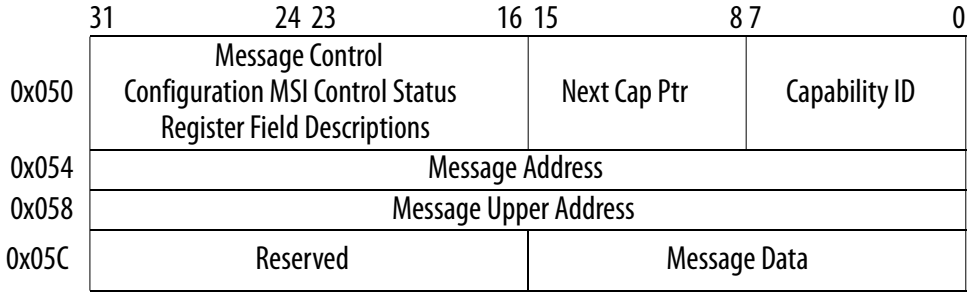


Figure 82. MSI-X Capability Structure

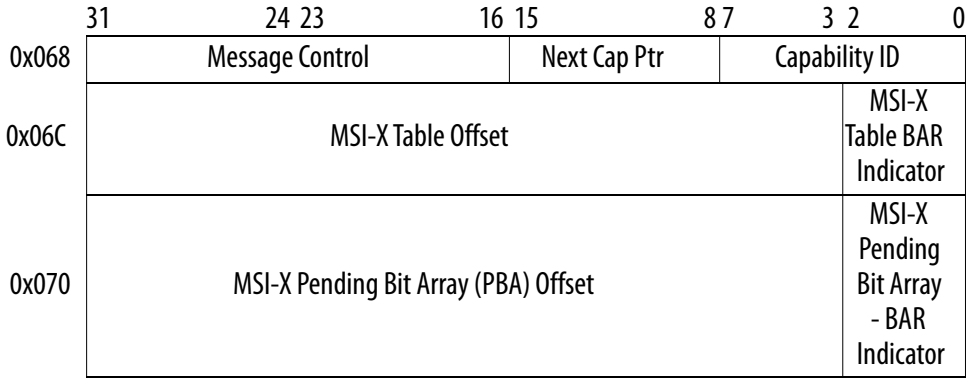


Figure 83. Power Management Capability Structure - Byte Address Offsets and Layout

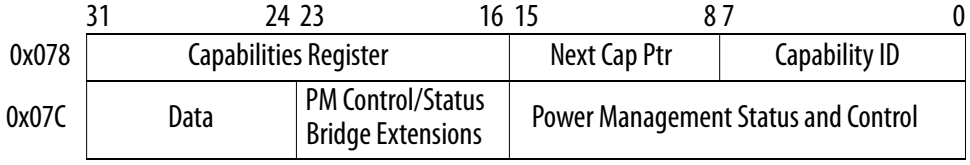


Figure 84. PCI Express AER Extended Capability Structure

Byte Offset	31:24	23:16	15:8	7:0
0x800	PCI Express Enhanced Capability Register			
0x804	Uncorrectable Error Status Register			
0x808	Uncorrectable Error Mask Register			
0x80C	Uncorrectable Error Severity Register			
0x810	Correctable Error Status Register			
0x814	Correctable Error Mask Register			
0x818	Advanced Error Capabilities and Control Register			
0x81C	Header Log Register			
0x82C	Root Error Command Register			
0x830	Root Error Status Register			
0x834	Error Source Identification Register		Correctable Error Source Identification Register	

Figure 85. PCI Express Capability Structure - Byte Address Offsets and Layout

In the following table showing the PCI Express Capability Structure, registers that are not applicable to a device are reserved.

	31 24	23 16	15 8	7 0
0x080	PCI Express Capabilities Register		Next Cap Pointer	PCI Express Capabilities ID
0x084	Device Capabilities			
0x088	Device Status		Device Control	
0x08C	Link Capabilities			
0x090	Link Status		Link Control	
0x094	Slot Capabilities			
0x098	Slot Status		Slot Control	
0x09C	Root Capabilities		Root Control	
0x0A0	Root Status			
0x0A4	Device Compatibilities 2			
0x0A8	Device Status 2		Device Control 2	
0x0AC	Link Capabilities 2			
0x0B0	Link Status 2		Link Control 2	
0x0B4	Slot Capabilities 2			
0x0B8	Slot Status 2		Slot Control 2	

Related Links

- PCI Express Base Specification 3.0
- PCI Local Bus Specification

7.5 Intel-Defined VSEC Registers

Figure 86. VSEC Registers

This extended capability structure supports Configuration via Protocol (CvP) programming and detailed internal error reporting.

	31 20	19 16	15 8	7 0
0x200	Next Capability Offset	Version	Intel-Defined VSEC Capability Header	
0x204	VSEC Length	VSEC Revision	VSEC ID Intel-Defined, Vendor-Specific Header	
0x208	Intel Marker			
0x20C	JTAG Silicon ID DW0 JTAG Silicon ID			
0x210	JTAG Silicon ID DW1 JTAG Silicon ID			
0x214	JTAG Silicon ID DW2 JTAG Silicon ID			
0x218	JTAG Silicon ID DW3 JTAG Silicon ID			
0x21C	CvP Status		User Device or Board Type ID	
0x220	CvP Mode Control			
0x224	CvP Data2 Register			
0x228	CvP Data Register			
0x22C	CvP Programming Control Register			
0x230	Reserved			
0x234	Uncorrectable Internal Error Status Register			
0x238	Uncorrectable Internal Error Mask Register			
0x23C	Correctable Internal Error Status Register			
0x240	Correctable Internal Error Mask Register			

Table 52. Intel-Defined VSEC Capability Register, 0x200

The Intel-Defined Vendor Specific Extended Capability. This extended capability structure supports Configuration via Protocol (CvP) programming and detailed internal error reporting.

Bits	Register Description	Value	Access
[15:0]	PCI Express Extended Capability ID. Intel-defined value for VSEC Capability ID.	0x000B	RO
[19:16]	Version. Intel-defined value for VSEC version.	0x1	RO
[31:20]	Next Capability Offset. Starting address of the next Capability Structure implemented, if any.	Variable	RO

Table 53. Intel-Defined Vendor Specific Header

You can specify these values when you instantiate the Hard IP. These registers are read-only at run-time.

Bits	Register Description	Value	Access
[15:0]	VSEC ID. A user configurable VSEC ID.	User entered	RO
[19:16]	VSEC Revision. A user configurable VSEC revision.	Variable	RO
[31:20]	VSEC Length. Total length of this structure in bytes.	0x044	RO

Table 54. Intel Marker Register

Bits	Register Description	Value	Access
[31:0]	Intel Marker. This read only register is an additional marker. If you use the standard Intel Programmer software to configure the device with CvP, this marker provides a value that the programming software reads to ensure that it is operating with the correct VSEC.	A Device Value	RO

Table 55. JTAG Silicon ID Register

Bits	Register Description	Value	Access
[127:96]	JTAG Silicon ID DW3	Application Specific	RO
[95:64]	JTAG Silicon ID DW2	Application Specific	RO
[63:32]	JTAG Silicon ID DW1	Application Specific	RO
[31:0]	JTAG Silicon ID DW0. This is the JTAG Silicon ID that CvP programming software reads to determine that the correct SRAM object file (**.sof**) is being used.	Application Specific	RO

Table 56. User Device or Board Type ID Register

Bits	Register Description	Value	Access
[15:0]	Configurable device or board type ID to specify to CvP the correct **.sof**.	Variable	RO

7.6 CvP Registers

Table 57. CvP Status

The CvP Status register allows software to monitor the CvP status signals.

Bits	Register Description	Reset Value	Access
[31:26]	Reserved	0x00	RO
[25]	PLD_CORE_READY. From FPGA fabric. This status bit is provided for debug.	Variable	RO
[24]	PLD_CLK_IN_USE. From clock switch module to fabric. This status bit is provided for debug.	Variable	RO
[23]	CVP_CONFIG_DONE. Indicates that the FPGA control block has completed the device configuration via CvP and there were no errors.	Variable	RO

continued...

Bits	Register Description	Reset Value	Access
[22]	Reserved	Variable	RO
[21]	USERMODE. Indicates if the configurable FPGA fabric is in user mode.	Variable	RO
[20]	CVP_EN. Indicates if the FPGA control block has enabled CvP mode.	Variable	RO
[19]	CVP_CONFIG_ERROR. Reflects the value of this signal from the FPGA control block, checked by software to determine if there was an error during configuration.	Variable	RO
[18]	CVP_CONFIG_READY. Reflects the value of this signal from the FPGA control block, checked by software during programming algorithm.	Variable	RO
[17:0]	Reserved	Variable	RO

Table 58. CvP Mode Control

The CvP Mode Control register provides global control of the CvP operation.

Bits	Register Description	Reset Value	Access
[31:16]	Reserved.	0x0000	RO
[15:8]	CVP_NUMCLKS. This is the number of clocks to send for every CvP data write. Set this field to one of the values below depending on your configuration image: • 0x01 for uncompressed and unencrypted images • 0x04 for uncompressed and encrypted images • 0x08 for all compressed images	0x00	RW
[7:3]	Reserved.	0x0	RO
[2]	CVP_FULLCONFIG. Request that the FPGA control block reconfigure the entire FPGA including the Arria 10 Hard IP for PCI Express, bring the PCIe link down.	1'b0	RW
[1]	HIP_CLK_SEL. Selects between PMA and fabric clock when USER_MODE = 1 and PLD_CORE_READY = 1. The following encodings are defined: • 1: Selects internal clock from PMA which is required for CVP_MODE. • 0: Selects the clock from soft logic fabric. This setting should only be used when the fabric is configured in USER_MODE with a configuration file that connects the correct clock. To ensure that there is no clock switching during CvP, you should only change this value when the Hard IP for PCI Express has been idle for 10 µs and wait 10 µs after changing this value before resuming activity.	1'b0	RW
[0]	CVP_MODE. Controls whether the IP core is in CVP_MODE or normal mode. The following encodings are defined: • 1:CVP_MODE is active. Signals to the FPGA control block active and all TLPs are routed to the Configuration Space. This CVP_MODE cannot be enabled if CVP_EN = 0. • 0: The IP core is in normal mode and TLPs are routed to the FPGA fabric.	1'b0	RW

Table 59. CvP Data Registers

The following table defines the CvP Data registers. For 64-bit data, the optional CvP Data2 stores the upper 32 bits of data. Programming software should write the configuration data to these registers. If you Every write to these register sets the data output to the FPGA control block and generates <*n*> clock cycles to the FPGA control block as specified by the CVP_NUM_CLKS field in the CvP Mode Control register. Software must ensure that all bytes in the memory write dword are enabled. You can access this register using configuration

writes, alternatively, when in CvP mode, these registers can also be written by a memory write to any address defined by a memory space BAR for this device. Using memory writes should allow for higher throughput than configuration writes.

Bits	Register Description	Reset Value	Access
[31:0]	Upper 32 bits of configuration data to be transferred to the FPGA control block to configure the device. You can choose 32- or 64-bit data.	0x00000000	RW
[31:0]	Lower 32 bits of configuration data to be transferred to the FPGA control block to configure the device.	0x00000000	RW

Table 60. CvP Programming Control Register

This register is written by the programming software to control CvP programming.

Bits	Register Description	Reset Value	Access
[31:2]	Reserved.	0x0000	RO
[1]	START_XFER. Sets the CvP output to the FPGA control block indicating the start of a transfer.	1'b0	RW
[0]	CVP_CONFIG. When asserted, instructs that the FPGA control block begin a transfer via CvP.	1'b0	RW

7.7 Uncorrectable Internal Error Mask Register

Table 61. Uncorrectable Internal Error Mask Register

The `Uncorrectable Internal Error Mask` register controls which errors are forwarded as internal uncorrectable errors. With the exception of the configuration error detected in CvP mode, all of the errors are severe and may place the device or PCIe link in an inconsistent state. The configuration error detected in CvP mode may be correctable depending on the design of the programming software. The access code *RWS* stands for Read Write Sticky meaning the value is retained after a soft reset of the IP core.

Bits	Register Description	Reset Value	Access
[31:12]	Reserved.	1b'0	RO
[11]	Mask for RX buffer posted and completion overflow error.	1b'0	RWS
[10]	Reserved	1b'1	RO
[9]	Mask for parity error detected on Configuration Space to TX bus interface.	1b'1	RWS
[8]	Mask for parity error detected on the TX to Configuration Space bus interface.	1b'1	RWS
[7]	Mask for parity error detected at TX Transaction Layer error.	1b'1	RWS
[6]	Reserved	1b'1	RO
[5]	Mask for configuration errors detected in CvP mode.	1b'0	RWS
[4]	Mask for data parity errors detected during TX Data Link LCRC generation.	1b'1	RWS
[3]	Mask for data parity errors detected on the RX to Configuration Space Bus interface.	1b'1	RWS
[2]	Mask for data parity error detected at the input to the RX Buffer.	1b'1	RWS
[1]	Mask for the retry buffer uncorrectable ECC error.	1b'1	RWS
[0]	Mask for the RX buffer uncorrectable ECC error.	1b'1	RWS

7.8 Uncorrectable Internal Error Status Register

Table 62. Uncorrectable Internal Error Status Register

This register reports the status of the internally checked errors that are uncorrectable. When specific errors are enabled by the `Uncorrectable Internal Error Mask` register, they are handled as Uncorrectable Internal Errors as defined in the *PCI Express Base Specification 3.0*. This register is for debug only. It should only be used to observe behavior, not to drive custom logic. The access code RW1CS represents Read Write 1 to Clear Sticky.

Bits	Register Description	Reset Value	Access
[31:12]	Reserved.	0	RO
[11]	When set, indicates an RX buffer overflow condition in a posted request or Completion	0	RW1CS
[10]	Reserved.	0	RO
[9]	When set, indicates a parity error was detected on the Configuration Space to TX bus interface	0	RW1CS
[8]	When set, indicates a parity error was detected on the TX to Configuration Space bus interface	0	RW1CS
[7]	When set, indicates a parity error was detected in a TX TLP and the TLP is not sent.	0	RW1CS
[6]	When set, indicates that the Application Layer has detected an uncorrectable internal error.	0	RW1CS
[5]	When set, indicates a configuration error has been detected in CvP mode which is reported as uncorrectable. This bit is set whenever a `CVP_CONFIG_ERROR` rises while in `CVP_MODE`.	0	RW1CS
[4]	When set, indicates a parity error was detected by the TX Data Link Layer.	0	RW1CS
[3]	When set, indicates a parity error has been detected on the RX to Configuration Space bus interface.	0	RW1CS
[2]	When set, indicates a parity error was detected at input to the RX Buffer.	0	RW1CS
[1]	When set, indicates a retry buffer uncorrectable ECC error.	0	RW1CS
[0]	When set, indicates a RX buffer uncorrectable ECC error.	0	RW1CS

Related Links

PCI Express Base Specification 3.0

7.9 Correctable Internal Error Mask Register

Table 63. Correctable Internal Error Mask Register

The `Correctable Internal Error Mask` register controls which errors are forwarded as Internal Correctable Errors. This register is for debug only.

Bits	Register Description	Reset Value	Access
[31:8]	Reserved.	0	RO
[7]	Reserved.	1	RO
[6]	Mask for Corrected Internal Error reported by the Application Layer.	1	RWS
[5]	Mask for configuration error detected in CvP mode.	1	RWS

continued...

Bits	Register Description	Reset Value	Access
[4:2]	Reserved.	0	RO
[1]	Mask for retry buffer correctable ECC error.	1	RWS
[0]	Mask for RX Buffer correctable ECC error.	1	RWS

7.10 Correctable Internal Error Status Register

Table 64. Correctable Internal Error Status Register

The `Correctable Internal Error Status` register reports the status of the internally checked errors that are correctable. When these specific errors are enabled by the `Correctable Internal Error Mask` register, they are forwarded as Correctable Internal Errors as defined in the *PCI Express Base Specification 3.0*. This register is for debug only. Only use this register to observe behavior, not to drive logic custom logic.

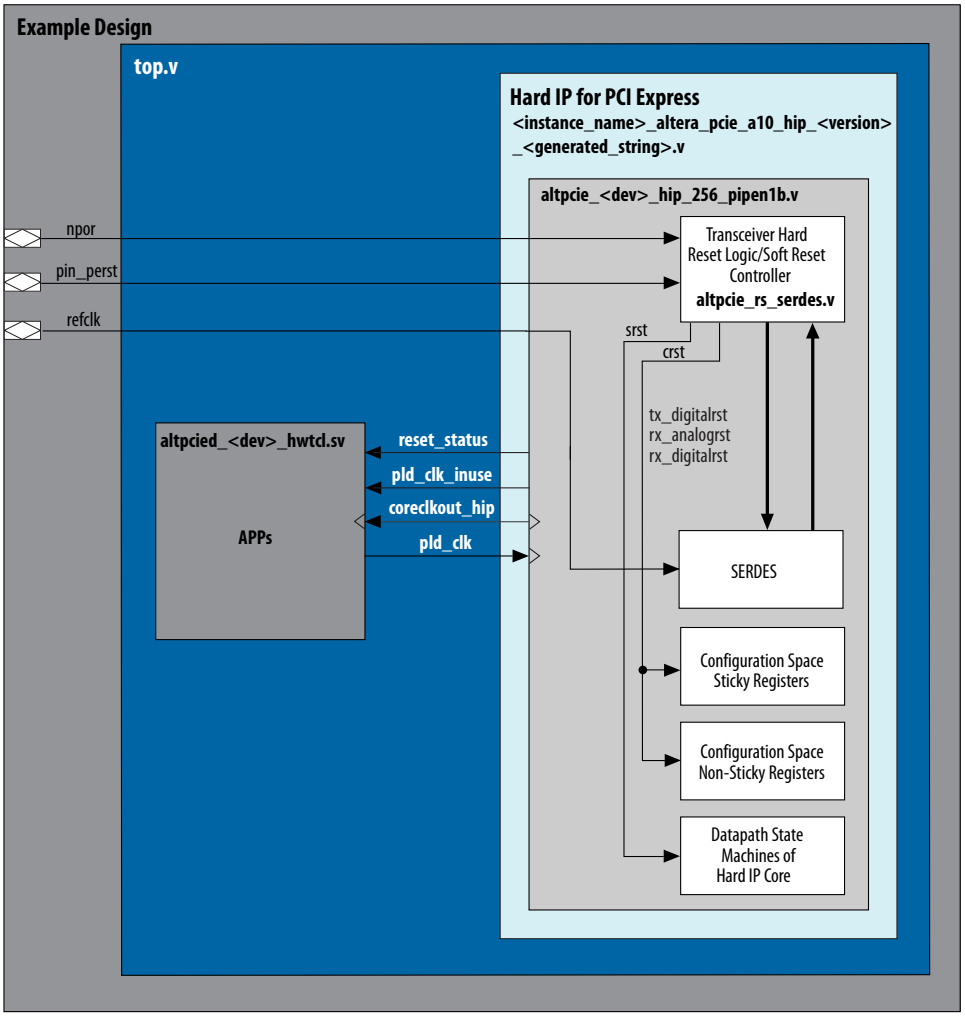
Bits	Register Description	Reset Value	Access
[31:7]	Reserved.	0	RO
[6]	Corrected Internal Error reported by the Application Layer.	0	RW1CS
[5]	When set, indicates a configuration error has been detected in CvP mode which is reported as correctable. This bit is set whenever a `CVP_CONFIG_ERROR` occurs while in `CVP_MODE`.	0	RW1CS
[4:2]	Reserved.	0	RO
[1]	When set, the retry buffer correctable ECC error status indicates an error.	0	RW1CS
[0]	When set, the RX buffer correctable ECC error status indicates an error.	0	RW1CS

Related Links

PCI Express Base Specification 3.0

8 Arria 10 Reset and Clocks

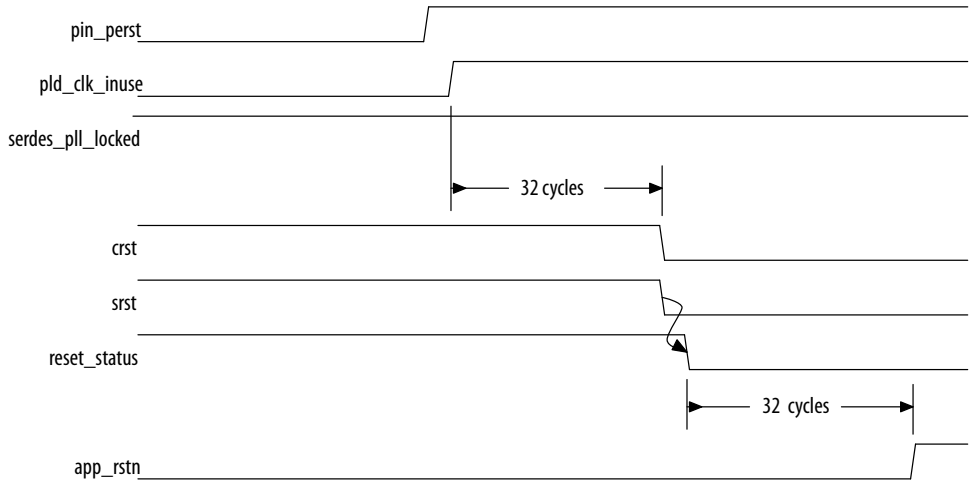
Figure 87. Reset Controller in Arria 10 Devices



8.1 Reset Sequence for Hard IP for PCI Express IP Core and Application Layer

Figure 88. Hard IP for PCI Express and Application Logic Reset Sequence

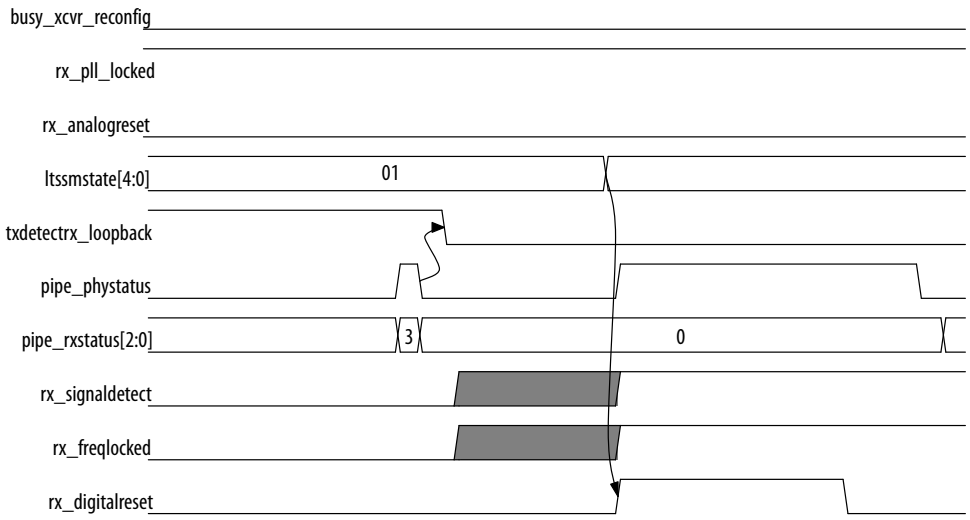
Your Application Layer can instantiate a module similar to the one in this figure to generate `app_rstn`, which resets the Application Layer logic.



This reset sequence includes the following steps:

1. After `pin_perst` or `npor` is released, the Hard IP reset controller waits for `pld_clk_inuse` to be asserted.
2. `csrt` and `srst` are released 32 cycles after `pld_clk_inuse` is asserted.
3. The Hard IP for PCI Express deasserts the `reset_status` output to the Application Layer.
4. The `altpcied_<device>v_hwtcl.sv` deasserts `app_rstn` 32 `pld_clk`cycles after `reset_status` is released.

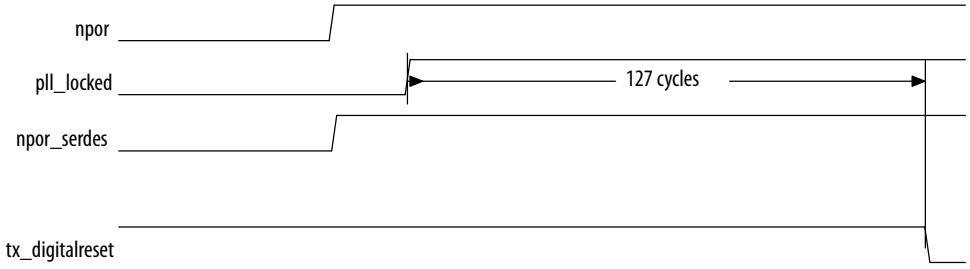
Figure 89. RX Transceiver Reset Sequence



The RX transceiver reset sequence includes the following steps:

1. After `rx_pll_locked` is asserted, the LTSSM state machine transitions from the Detect.Quiet to the Detect.Active state.
2. When the `pipe_phystatus` pulse is asserted and `pipe_rxstatus[2:0] = 3`, the receiver detect operation has completed.
3. The LTSSM state machine transitions from the Detect.Active state to the Polling.Active state.
4. The Hard IP for PCI Express asserts `rx_digitalreset`. The `rx_digitalreset` signal is deasserted after `rx_signaldetect` is stable for a minimum of 3 ms.

Figure 90. TX Transceiver Reset Sequence



The TX transceiver reset sequence includes the following steps:

1. After `npor` is deasserted, the IP core deasserts the `npor_serdes` input to the TX transceiver.
2. The SERDES reset controller waits for `pll_locked` to be stable for a minimum of 127 `pld_clk` cycles before deasserting `tx_digitalreset`.

For descriptions of the available reset signals, refer to *Reset Signals, Status, and Link Training Signals*.

Related Links

Reset, Status, and Link Training Signals on page 82

8.2 Clocks

The Hard IP contains a clock domain crossing (CDC) synchronizer at the interface between the PHY/MAC and the DLL layers. The synchronizer allows the Data Link and Transaction Layers to run at frequencies independent of the PHY/MAC. The CDC synchronizer provides more flexibility for the user clock interface. Depending on parameters you specify, the core selects the appropriate `coreclkout_hip`. You can use these parameters to enhance performance by running at a higher frequency for latency optimization or at a lower frequency to save power.

In accordance with the *PCI Express Base Specification*, you must provide a 100 MHz reference clock that is connected directly to the transceiver.

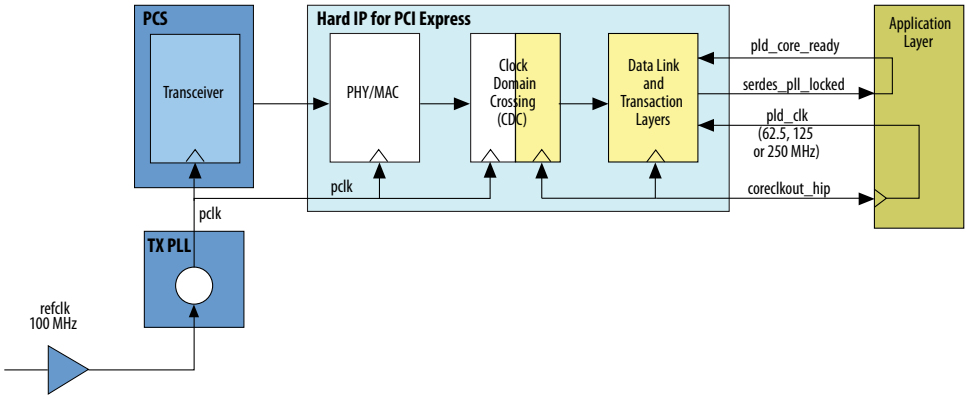
Related Links

PCI Express Base Specification 3.0

8.2.1 Clock Domains

Figure 91. Clock Domains and Clock Generation for the Application Layer

The following illustrates the clock domains when using `coreclkout_hip` to drive the Application Layer and the `pld_clk` of the IP core. The Intel-provided example design connects `coreclkout_hip` to the `pld_clk`. However, this connection is not mandatory.



As this figure indicates, the IP core includes the following clock domains:

8.2.1.1 coreclkout_hip

Table 65. Application Layer Clock Frequency for All Combinations of Link Width, Data Rate and Application Layer Interface Widths

The `coreclkout_hip` signal is derived from `pclk`. The following table lists frequencies for `coreclkout_hip`, which are a function of the link width, data rate, and the width of the Application Layer to Transaction Layer interface. The frequencies and widths specified in this table are maintained throughout operation. If the link

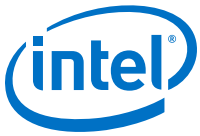
downtrains to a lesser link width or changes to a different maximum link rate, it maintains the frequencies it was originally configured for as specified in this table. (The Hard IP throttles the interface to achieve a lower throughput.)

Link Width	Maximum Link Rate	Avalon Interface Width	coreclkout_hip
×1	Gen1	64	62.5 MHz[3]
×1	Gen1	64	125 MHz
×2	Gen1	64	125 MHz
×4	Gen1	64	125 MHz
×8	Gen1	64	250 MHz
×8	Gen1	128	125 MHz
×1	Gen2	64	125 MHz
×2	Gen2	64	125 MHz
×4	Gen2	64	250 MHz
×4	Gen2	128	125 MHz
×8	Gen2	128	250 MHz
×8	Gen2	256	125 MHz
×1	Gen3	64	125 MHz
×2	Gen3	64	125 MHz
×2	Gen3	128	125 MHz
×2	Gen3	64	250 MHz
×4	Gen3	128	250 MHz
×4	Gen3	256	125 MHz
×8	Gen3	256	250 MHz

8.2.1.2 pld_clk

`coreclkout_hip` can drive the Application Layer clock along with the `pld_clk` input to the IP core. The `pld_clk` can optionally be sourced by a different clock than `coreclkout_hip`. The `pld_clk` minimum frequency cannot be lower than the `coreclkout_hip` frequency. Based on specific Application Layer constraints, a PLL can be used to derive the desired frequency.

3 This mode saves power



8.2.2 Clock Summary

Table 66. Clock Summary

Name	Frequency	Clock Domain
coreclkout_hip	62.5, 125 or 250 MHz	Avalon-ST interface between the Transaction and Application Layers.
pld_clk	62.5, 125, or 250 MHz	Application and Transaction Layers.
refclk	100 MHz	SERDES (transceiver). Dedicated free running input clock to the SERDES block.
hip_reconfig_clk		Avalon-MM interface for Hard IP dynamic reconfiguration interface which you can use to change the value of read-only configuration registers at run-time. This interface is optional. It is not required for Arria 10 devices.

9 Interrupts

9.1 Interrupts for Endpoints

The Arria 10 Hard IP for PCI Express provides support for PCI Express MSI, MSI-X, and legacy interrupts when configured in Endpoint mode. The MSI, MSI-X, and legacy interrupts are *mutually exclusive.* After power up, the Hard IP block starts in legacy interrupt mode. Then, software decides whether to switch to MSI or MSI-X mode. To switch to MSI mode, software programs the `msi_enable` bit of the `MSI Message Control Register` to 1, (bit[16] of 0x050). You enable MSI-X mode, by turning on **Implement MSI-X** under the **PCI Express/PCI Capabilities** tab using the parameter editor. If you turn on the **Implement MSI-X** option, you should implement the MSI-X table structures at the memory space pointed to by the BARs.

Note:

Refer to section 6.1 of *PCI Express Base Specification* for a general description of PCI Express interrupt support for Endpoints.

Related Links

PCI Express Base Specification 3.0

9.1.1 MSI and Legacy Interrupts

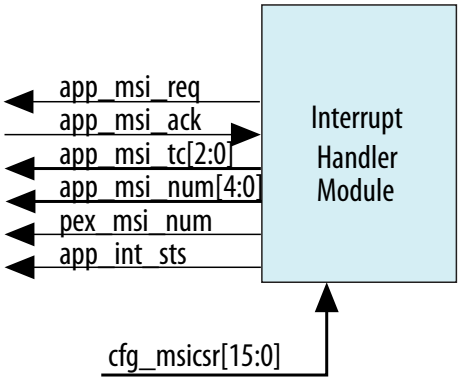
The IP core generates single dword Memory Write TLPs to signal MSI interrupts on the PCI Express link. The Application Layer Interrupt Handler Module `app_msi_req` output port controls MSI interrupt generation. When asserted, it causes an MSI posted Memory Write TLP to be generated. The IP core constructs the TLP using information from the following sources:

- The MSI Capability registers
- The traffic class (`app_msi_tc`)
- The message data specified by `app_msi_num`

To enable MSI interrupts, the Application Layer must first set the `MSI enable` bit. Then, it must disable legacy interrupts by setting the `Interrupt Disable`, bit 10 of the `Command` register.

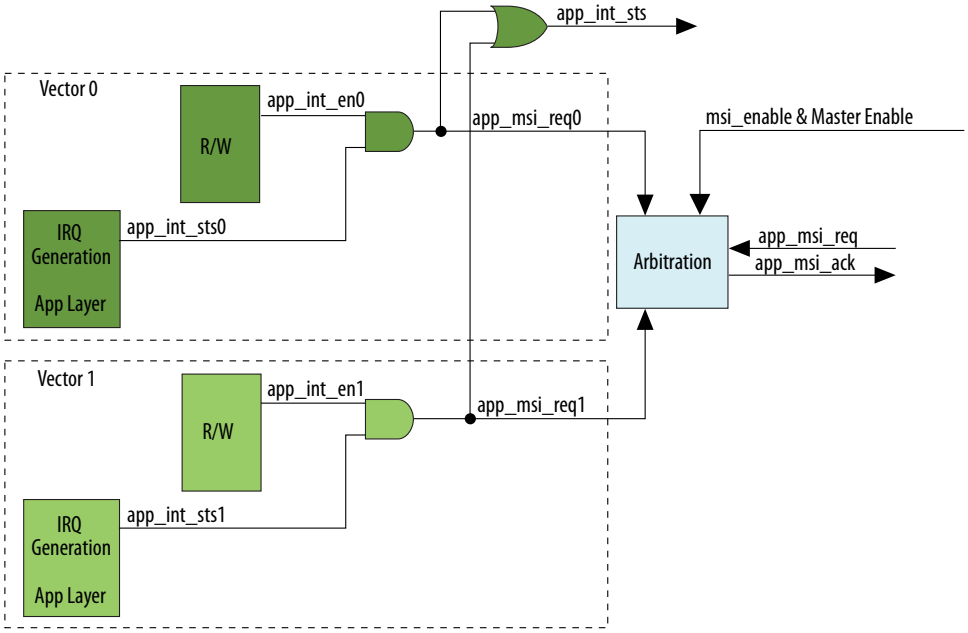
The Application Layer Interrupt Handler Module also generates legacy interrupts. The `app_int_sts` signal controls legacy interrupt assertion and deassertion.

Figure 92. Interrput Handler Module in the Application Layer



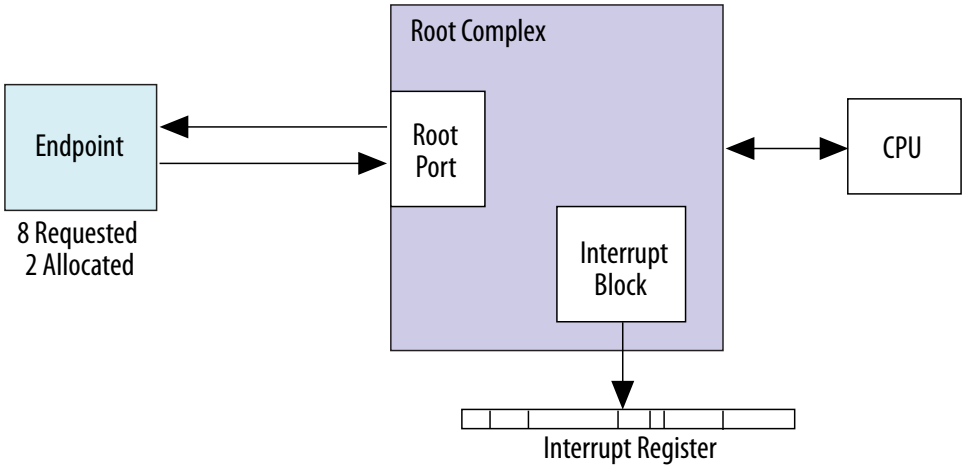
The following figure illustrates a possible implementation of the Interrupt Handler Module with a per vector enable bit. Alternatively, the Application Layer could implement a global interrupt enable instead of this per vector MSI.

Figure 93. Example Implementation of the Interrupt Handler Block



There are 32 possible MSI messages. The number of messages requested by a particular component does not necessarily correspond to the number of messages allocated. For example, in the following figure, the Endpoint requests eight MSIs but is only allocated two. In this case, you must design the Application Layer to use only two allocated messages.

Figure 94. MSI Request Example



The following table describes three example implementations. The first example allocates all 32 MSI messages. The second and third examples only allocate 4 interrupts.

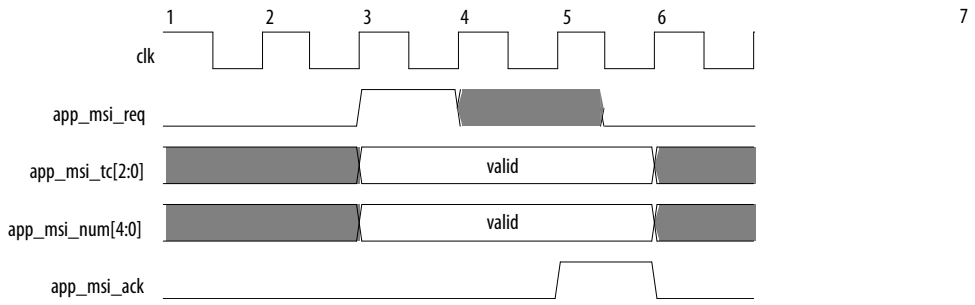
Table 67. MSI Messages Requested, Allocated, and Mapped

MSI	Allocated		
	32	4	4
System Error	31	3	3
Hot Plug and Power Management Event	30	2	3
Application Layer	29:0	1:0	2:0

MSI interrupts generated for Hot Plug, Power Management Events, and System Errors always use Traffic Class 0. MSI interrupts generated by the Application Layer can use any Traffic Class. For example, a DMA that generates an MSI at the end of a transmission can use the same traffic control as was used to transfer data.

The following figure illustrates the interactions among MSI interrupt signals for the Root Port. The minimum latency possible between `app_msi_req` and `app_msi_ack` is one clock cycle. In this timing diagram `app_msi_req` can extend beyond `app_msi_ack` before deasserting. However, `app_msi_req` must be deasserted before or within the same clock as `app_msi_ack` is deasserted to avoid inferring a new interrupt.

Figure 95. MSI Interrupt Signals Timing



Related Links

Correspondence between Configuration Space Registers and the PCIe Specification on page 107

9.1.2 MSI-X

You can enable MSI-X interrupts by turning on **Implement MSI-X** under the **PCI Express/PCI Capabilities** heading using the parameter editor. If you turn on the **Implement MSI-X** option, you should implement the MSI-X table structures at the memory space pointed to by the BARs as part of your Application Layer.

The Application Layer transmits MSI-X interrupts on the Avalon®-ST TX interface. MSI-X interrupts are single dword Memory Write TLPs. Consequently, the `Last DW Byte Enable` in the TLP header must be set to 4b'0000. MSI-X TLPs should be sent only when enabled by the `MSI-X enable` and the function mask bits in the `Message Control` for the MSI-X Configuration register. These bits are available on the `tl_cfg_ctl` output bus.

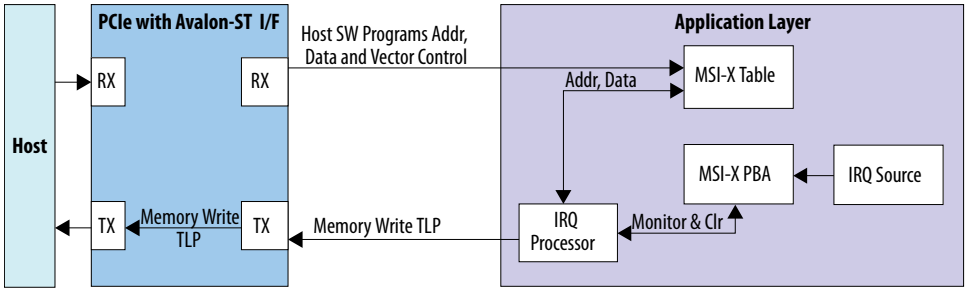
Related Links

- PCI Local Bus Specification
- PCI Express Base Specification 3.0

9.1.3 Implementing MSI-X Interrupts

Section 6.8.2 of the *PCI Local Bus Specification* describes the MSI-X capability and table structures. The MSI-X capability structure points to the MSI-X Table structure and MSI-X Pending Bit Array (PBA) registers. The BIOS sets up the starting address offsets and BAR associated with the pointer to the starting address of the MSI-X Table and PBA registers.

Figure 96. MSI-X Interrupt Components



1. Host software sets up the MSI-X interrupts in the Application Layer by completing the following steps:
   a. Host software reads the `Message Control` register at 0x050 register to determine the MSI-X Table size. The number of table entries is the *<value read> + 1*.

      The maximum table size is 2048 entries. Each 16-byte entry is divided in 4 fields as shown in the figure below. The MSI-X table can reside in any BAR. The base address of the MSI-X table must be aligned to a 4 KB boundary.
   b. The host sets up the MSI-X table. It programs MSI-X address, data, and masks bits for each entry as shown in the figure below.

Figure 97. Format of MSI-X Table

DWORD 3	DWORD 2	DWORD 1	DWORD 0		Host Byte Addresses
Vector Control	Message Data	Message Upper Address	Message Address	Entry 0	Base
Vector Control	Message Data	Message Upper Address	Message Address	Entry 1	Base + 1 × 16
Vector Control	Message Data	Message Upper Address	Message Address	Entry 2	Base + 2 × 16
⋮	⋮	⋮	⋮	⋮	⋮
Vector Control	Message Data	Message Upper Address	Message Address	Entry (*N* - 1)	Base + (*N* - 1) × 16

   c. The host calculates the address of the <n^{th}> entry using the following formula:

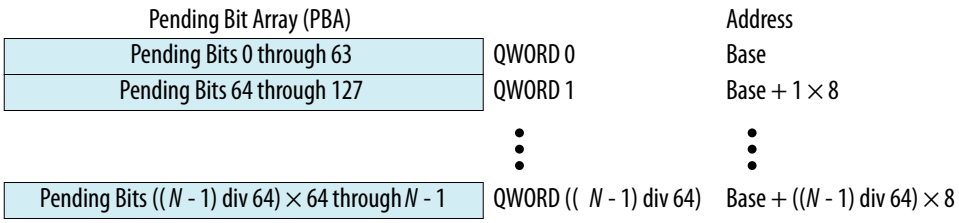
```
nth_address = base address[BAR] + 16<n>
```

2. When Application Layer has an interrupt, it drives an interrupt request to the IRQ Source module.
3. The IRQ Source sets appropriate bit in the MSI-X PBA table.

   The PBA can use qword or dword accesses. For qword accesses, the IRQ Source calculates the address of the <m^{th}> bit using the following formulas:

```
qword address = <PBA base addr> + 8(floor(<m>/64))
qword bit = <m> mod 64
```

Figure 98. MSI-X PBA Table



4. The IRQ Processor reads the entry in the MSI-X table.
   a. If the interrupt is masked by the `Vector_Control` field of the MSI-X table, the interrupt remains in the pending state.
   b. If the interrupt is not masked, IRQ Processor sends Memory Write Request to the TX slave interface. It uses the address and data from the MSI-X table. If **`Message Upper Address`** = 0, the IRQ Processor creates a three-dword header. If the **`Message Upper Address`** > 0, it creates a 4-dword header.
5. The host interrupt service routine detects the TLP as an interrupt and services it.

Related Links

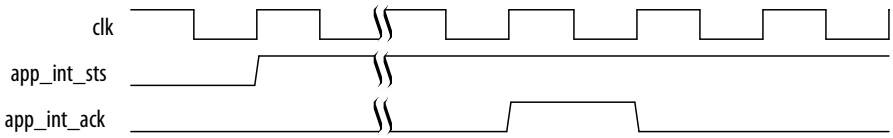
- Floor and ceiling functions
- PCI Local Bus Specification, Rev. 3.0

9.1.4 Legacy Interrupts

Legacy interrupts mimic the original PCI level-sensitive interrupts using *virtual wire* messages. The Arria 10signals legacy interrupts on the PCIe link using Message TLPs. The term, INTx, refers collectively to the four legacy interrupts, INTA#, INTB#, INTC# and INTD#. The Arria 10 asserts `app_int_sts` to cause an `Assert_INTx` Message TLP to be generated and sent upstream. Deassertion of `app_int_sts` causes a `Deassert_INTx` Message TLP to be generated and sent upstream. To use legacy interrupts, you must clear the `Interrupt Disable` bit, which is bit 10 of the `Command` register. Then, turn off the `MSI Enable` bit.

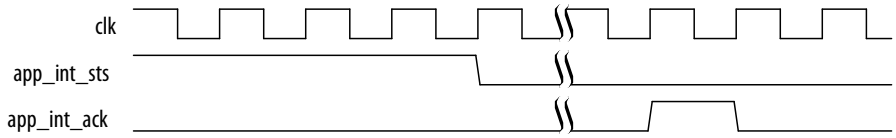
The following figures illustrates interrupt timing for the legacy interface. The legacy interrupt handler asserts `app_int_sts` to instruct the Hard IP for PCI Express to send a `Assert_INTx` message TLP.

Figure 99. Legacy Interrupt Assertion



The following figure illustrates the timing for deassertion of legacy interrupts. The legacy interrupt handler asserts `app_int_sts` causing the Hard IP for PCI Express to send a `Deassert_INTx` message.

Figure 100. Legacy Interrupt Deassertion



Related Links

- Correspondence between Configuration Space Registers and the PCIe Specification on page 107
- target-title
  Delete element if none

9.2 Interrupts for Root Ports

In Root Port mode, the Arria 10 Hard IP for PCI Express receives interrupts through two different mechanisms:

- MSI—Root Ports receive MSI interrupts through the Avalon-ST RX Memory Write TLP. This is a memory mapped mechanism.
- Legacy—Legacy interrupts are translated into Message Interrupt TLPs and sent to the Application Layer using the `int_status` pins.

Normally, the Root Port services rather than sends interrupts; however, in two circumstances the Root Port can send an interrupt to itself to record error conditions:

- When the AER option is enabled, the `aer_msi_num[4:0]` signal indicates which MSI is being sent to the root complex when an error is logged in the AER Capability structure. This mechanism is an alternative to using the `serr_out` signal. The `aer_msi_n um[4:0]` is only used for Root Ports and you must set it to a constant value. It cannot toggle during operation.
- If the Root Port detects a Power Management Event, the `pex_msi_num[4:0]` signal is used by Power Management or Hot Plug to determine the offset between the base message interrupt number and the message interrupt number to send through MSI. The user must set `pex_msi_num[4:0]`to a fixed value.

The `Root Error Status` register reports the status of error messages. The `Root Error Status` register is part of the PCI Express AER Extended Capability structure. It is located at offset 0x830 of the Configuration Space registers.

10 Error Handling

Each PCI Express compliant device must implement a basic level of error management and can optionally implement advanced error management. The IP core implements both basic and advanced error reporting. Error handling for a Root Port is more complex than that of an Endpoint.

Table 68. Error Classification

The *PCI Express Base Specification* defines three types of errors, outlined in the following table.

Type	Responsible Agent	Description
Correctable	Hardware	While correctable errors may affect system performance, data integrity is maintained.
Uncorrectable, non-fatal	Device software	Uncorrectable, non-fatal errors are defined as errors in which data is lost, but system integrity is maintained. For example, the fabric may lose a particular TLP, but it still works without problems.
Uncorrectable, fatal	System software	Errors generated by a loss of data and system failure are considered uncorrectable and fatal. Software must determine how to handle such errors: whether to reset the link or implement other means to minimize the problem.

Related Links

PCI Express Base Specification 3.0

10.1 Physical Layer Errors

Table 69. Errors Detected by the Physical Layer

The following table describes errors detected by the Physical Layer. Physical Layer error reporting is optional in the *PCI Express Base Specification*.

Error	Type	Description
Receive port error	Correctable	This error has the following 3 potential causes: • Physical coding sublayer error when a lane is in L0 state. These errors are reported to the Hard IP block via the per lane PIPE interface input receive status signals, `rxstatus<lane_number>[2:0]` using the following encodings: — 3'b100: 8B/10B Decode Error — 3'b101: Elastic Buffer Overflow — 3'b110: Elastic Buffer Underflow — 3'b111: Disparity Error • Deskew error caused by overflow of the multilane deskew FIFO. • Control symbol received in wrong lane.

10.2 Data Link Layer Errors

Table 70. Errors Detected by the Data Link Layer

Error	Type	Description
Bad TLP	Correctable	This error occurs when a LCRC verification fails or when a sequence number error occurs.
Bad DLLP	Correctable	This error occurs when a CRC verification fails.
Replay timer	Correctable	This error occurs when the replay timer times out.
Replay num rollover	Correctable	This error occurs when the replay number rolls over.
Data Link Layer protocol	Uncorrectable(fatal)	This error occurs when a sequence number specified by the Ack/Nak block in the Data Link Layer (`AckNak_Seq_Num`) does not correspond to an unacknowledged TLP.

10.3 Transaction Layer Errors

Table 71. Errors Detected by the Transaction Layer

Error	Type	Description
Poisoned TLP received	Uncorrectable (non-fatal)	This error occurs if a received Transaction Layer packet has the EP poison bit set. The received TLP is passed to the Application Layer and the Application Layer logic must take appropriate action in response to the poisoned TLP. Refer to "2.7.2.2 Rules for Use of Data Poisoning" in the *PCI Express Base Specification* for more information about poisoned TLPs.
ECRC check failed *(1)*	Uncorrectable (non-fatal)	This error is caused by an ECRC check failing despite the fact that the TLP is not malformed and the LCRC check is valid. The Hard IP block handles this TLP automatically. If the TLP is a non-posted request, the Hard IP block generates a completion with completer abort status. In all cases the TLP is deleted in the Hard IP block and not presented to the Application Layer.
Unsupported Request for Endpoints	Uncorrectable (non-fatal)	This error occurs whenever a component receives any of the following Unsupported Requests: • Type 0 Configuration Requests for a non-existing function. • Completion transaction for which the Requester ID does not match the bus, device and function number. • Unsupported message. • A Type 1 Configuration Request TLP for the TLP from the PCIe link. • A locked memory read (MEMRDLK) on native Endpoint. • A locked completion transaction. • A 64-bit memory transaction in which the 32 MSBs of an address are set to 0. • A memory or I/O transaction for which there is no BAR match. • A memory transaction when the Memory Space Enable bit (bit [1] of the PCI Command register at Configuration Space offset 0x4) is set to 0. • A poisoned configuration write request (`CfgWr0`) In all cases the TLP is deleted in the Hard IP block and not presented to the Application Layer. If the TLP is a non-posted request, the Hard IP block generates a completion with Unsupported Request status.

continued...

Error	Type	Description
Unsupported Requests for Root Port	Uncorrectable (fatal)	This error occurs whenever a component receives an Unsupported Request including: • Unsupported message • A Type 0 Configuration Request TLP • A 64-bit memory transaction which the 32 MSBs of an address are set to 0. • A memory transaction that does not match the address range defined by the Base and Limit Address registers
Completion timeout	Uncorrectable (non-fatal)	This error occurs when a request originating from the Application Layer does not generate a corresponding completion TLP within the established time. It is the responsibility of the Application Layer logic to provide the completion timeout mechanism. The completion timeout should be reported from the Transaction Layer using the `cpl_err[0]` signal.
Completer abort *(1)*	Uncorrectable (non-fatal)	The Application Layer reports this error using the `cpl_err[2]`signal when it aborts receipt of a TLP.
Unexpected completion	Uncorrectable (non-fatal)	This error is caused by an unexpected completion transaction. The Hard IP block handles the following conditions: • The Requester ID in the completion packet does not match the Configured ID of the Endpoint. • The completion packet has an invalid tag number. (Typically, the tag used in the completion packet exceeds the number of tags specified.) • The completion packet has a tag that does not match an outstanding request. • The completion packet for a request that was to I/O or Configuration Space has a length greater than 1 dword. • The completion status is Configuration Retry Status (CRS) in response to a request that was not to Configuration Space. In all of the above cases, the TLP is not presented to the Application Layer; the Hard IP block deletes it. The Application Layer can detect and report other unexpected completion conditions using the `cpl_err[2]` signal. For example, the Application Layer can report cases where the total length of the received successful completions do not match the original read request length.
Receiver overflow *(1)*	Uncorrectable (fatal)	This error occurs when a component receives a TLP that violates the FC credits allocated for this type of TLP. In all cases the hard IP block deletes the TLP and it is not presented to the Application Layer.
Flow control protocol error (FCPE) *(1)*	Uncorrectable (fatal)	This error occurs when a component does not receive update flow control credits with the 200 µs limit.
Malformed TLP	Uncorrectable (fatal)	This error is caused by any of the following conditions: • The data payload of a received TLP exceeds the maximum payload size. • The TD field is asserted but no TLP digest exists, or a TLP digest exists but the TD bit of the PCI Express request header packet is not asserted. • A TLP violates a byte enable rule. The Hard IP block checks for this violation, which is considered optional by the PCI Express specifications. • A TLP in which the `type` and `length` fields do not correspond with the total length of the TLP.

continued...

Error	Type	Description
		• A TLP in which the combination of format and type is not specified by the PCI Express specification. • A request specifies an address/length combination that causes a memory space access to exceed a 4 KB boundary. The Hard IP block checks for this violation, which is considered optional by the PCI Express specification. • Messages, such as Assert_INTX, Power Management, Error Signaling, Unlock, and Set Power Slot Limit, must be transmitted across the default traffic class. The Hard IP block deletes the malformed TLP; it is not presented to the Application Layer.
Note: 1. Considered optional by the *PCI Express Base Specification Revision*.		

10.4 Error Reporting and Data Poisoning

How the Endpoint handles a particular error depends on the configuration registers of the device.

Refer to the *PCI Express Base Specification 3.0* for a description of the device signaling and logging for an Endpoint.

The Hard IP block implements data poisoning, a mechanism for indicating that the data associated with a transaction is corrupted. Poisoned TLPs have the error/poisoned bit of the header set to 1 and observe the following rules:

- Received poisoned TLPs are sent to the Application Layer and status bits are automatically updated in the Configuration Space.
- Received poisoned Configuration Write TLPs are not written in the Configuration Space.
- The Configuration Space never generates a poisoned TLP; the error/poisoned bit of the header is always set to 0.

Poisoned TLPs can also set the parity error bits in the PCI Configuration Space Status register.

Table 72. Parity Error Conditions

Status Bit	Conditions
Detected parity error (status register bit 15)	Set when any received TLP is poisoned.
Master data parity error (status register bit 8)	This bit is set when the command register parity enable bit is set and one of the following conditions is true: • The poisoned bit is set during the transmission of a Write Request TLP. • The poisoned bit is set on a received completion TLP.

Poisoned packets received by the Hard IP block are passed to the Application Layer. Poisoned transmit TLPs are similarly sent to the link.

Related Links

PCI Express Base Specification 3.0

10.5 Uncorrectable and Correctable Error Status Bits

The following section is reprinted with the permission of PCI-SIG. Copyright 2010 PCI-SIG.

Figure 101. Uncorrectable Error Status Register

The default value of all the bits of this register is 0. An error status bit that is set indicates that the error condition it represents has been detected. Software may clear the error status by writing a 1 to the appropriate bit.

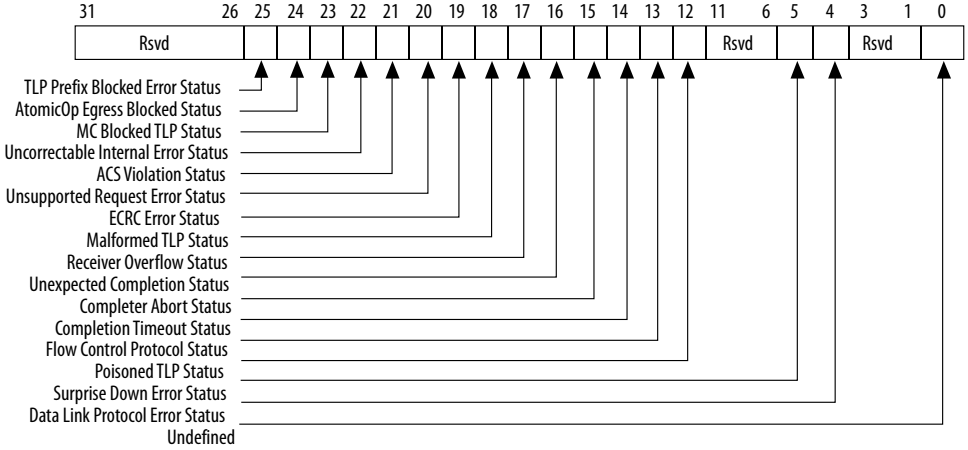
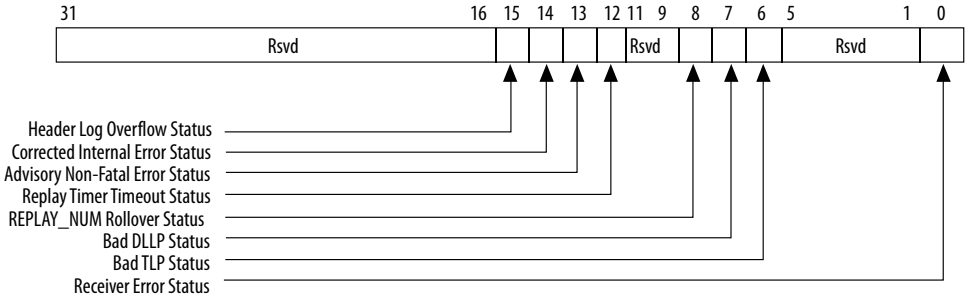


Figure 102. Correctable Error Status Register

The default value of all the bits of this register is 0. An error status bit that is set indicates that the error condition it represents has been detected. Software may clear the error status by writing a 1 to the appropriate bit.



11 IP Core Architecture

The Arria 10 Hard IP for PCI Express implements the complete PCI Express protocol stack as defined in the *PCI Express Base Specification.* The protocol stack includes the following layers:

- *Transaction Layer*—The Transaction Layer contains the Configuration Space, which manages communication with the Application Layer, the RX and TX channels, the RX buffer, and flow control credits.
- *Data Link Layer*—The Data Link Layer, located between the Physical Layer and the Transaction Layer, manages packet transmission and maintains data integrity at the link level. Specifically, the Data Link Layer performs the following tasks:
  - Manages transmission and reception of Data Link Layer Packets (DLLPs)
  - Generates all transmission cyclical redundancy code (CRC) values and checks all CRCs during reception
  - Manages the retry buffer and retry mechanism according to received ACK/NAK Data Link Layer packets
  - Initializes the flow control mechanism for DLLPs and routes flow control credits to and from the Transaction Layer
- *Physical Layer*—The Physical Layer initializes the speed, lane numbering, and lane width of the PCI Express link according to packets received from the link and directives received from higher layers.

The following figure provides a high-level block diagram.

Figure 103. Arria 10 Hard IP for PCI Express Using the Avalon-ST Interface

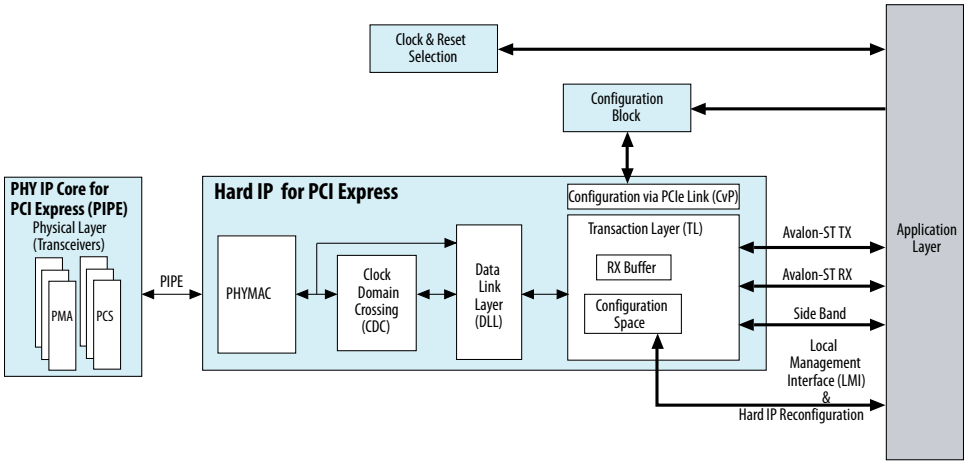


Table 73. Application Layer Clock Frequencies

Lanes	Gen1	Gen2	Gen3
×1	125 MHz @ 64 bits or 62.5 MHz @ 64 bits	125 MHz @ 64 bits	125 MHz @64 bits
×2	125 MHz @ 64 bits	125 MHz @ 128 bits	250 MHz @ 64 bits or 125 MHz @ 128 bits
×4	125 MHz @ 64 bits	250 MHz @ 64 bits or 125 MHz @ 128 bits	250 MHz @ 128 bits or 125 MHz @ 256 bits
×8	250 MHz @ 64 bits or 125 MHz @ 128 bits	250 MHz @ 128 bits or 125 MHz @ 256 bits	250 MHz @ 256 bits

The following interfaces provide access to the Application Layer's Configuration Space Registers:

- The LMI interface
- The Avalon-MM PCIe reconfiguration interface, which can access any read-only Configuration Space Register
- In Root Port mode, you can also access the Configuration Space Registers with a Configuration TLP using the Avalon-ST interface. A Type 0 Configuration TLP is used to access the Root Port configuration Space Registers, and a Type 1 Configuration TLP is used to access the Configuration Space Registers of downstream components, typically Endpoints on the other side of the link.

The Hard IP includes dedicated clock domain crossing logic (CDC) between the PHYMAC and Data Link Layers.

Related Links

PCI Express Base Specification 3.0

11.1 Top-Level Interfaces

11.1.1 Avalon-ST Interface

An Avalon-ST interface connects the Application Layer and the Transaction Layer. This is a point-to-point, streaming interface designed for high throughput applications. The Avalon-ST interface includes the RX and TX datapaths.

For more information about the Avalon-ST interface, including timing diagrams, refer to the *Avalon Interface Specifications*.

RX Datapath

The RX datapath transports data from the Transaction Layer to the Application Layer's Avalon-ST interface. Masking of non-posted requests is partially supported. Refer to the description of the `rx_st_mask` signal for further information about masking.

TX Datapath

The TX datapath transports data from the Application Layer's Avalon-ST interface to the Transaction Layer. The Hard IP provides credit information to the Application Layer for posted headers, posted data, non-posted headers, non-posted data, completion headers and completion data.

The Application Layer may track credits consumed and use the credit limit information to calculate the number of credits available. However, to enforce the PCI Express Flow Control (FC) protocol, the Hard IP also checks the available credits before sending a request to the link, and if the Application Layer violates the available credits for a TLP it transmits, the Hard IP blocks that TLP and all future TLPs until credits become available. By tracking the credit consumed information and calculating the credits available, the Application Layer can optimize performance by selecting for transmission only the TLPs that have credits available.

Related Links

- Avalon-ST RX Interface on page 58
- Avalon-ST TX Interface on page 70
- Avalon Interface Specifications
  For information about the Avalon-ST interface protocol.

11.1.2 Clocks and Reset

The *PCI Express Base Specification* requires an input reference clock, which is called `refclk` in this design. The *PCI Express Base Specification* stipulates that the frequency of this clock be 100 MHz.

The *PCI Express Base Specification* also requires a system configuration time of 100 ms. To meet this specification, IP core includes an embedded hard reset controller. This reset controller exits the reset state after the periphery of the device is initialized.

Related Links

- Clock Signals on page 82
- Reset, Status, and Link Training Signals on page 82

11.1.3 Local Management Interface (LMI Interface)

The LMI bus provides access to the PCI Express Configuration Space in the Transaction Layer.

Related Links

LMI Signals on page 90

11.1.4 Hard IP Reconfiguration

The PCI Express reconfiguration bus allows you to dynamically change the `read-only` values stored in the Configuration Registers.

Related Links

Hard IP Reconfiguration Interface on page 98

11.1.5 Interrupts

The Hard IP for PCI Express offers the following interrupt mechanisms:

- Message Signaled Interrupts (MSI)— MSI uses the TLP single dword memory writes to to implement interrupts. This interrupt mechanism conserves pins because it does not use separate wires for interrupts. In addition, the single dword provides flexibility in data presented in the interrupt message. The MSI Capability structure is stored in the Configuration Space and is programmed using Configuration Space accesses.
- MSI-X—The Transaction Layer generates MSI-X messages which are single dword memory writes. The MSI-X Capability structure points to an MSI-X table structure and MSI-X PBA structure which are stored in memory. This scheme is in contrast to the MSI capability structure, which contains all of the control and status information for the interrupt vectors.
- Legacy interrupts—The `app_int_sts` port controls legacy interrupt generation. When `app_int_sts` is asserted, the Hard IP generates an Assert_INT<*n*> message TLP.

Related Links

- Interrupts for Endpoints on page 86
- Interrupts for Root Ports on page 87

11.1.6 PIPE

The PIPE interface implements the Intel-designed PIPE interface specification. You can use this parallel interface to speed simulation; however, you cannot use the PIPE interface in actual hardware.

- The Gen1, Gen2, and Gen3 simulation models support PIPE and serial simulation.
- For Gen3, the Intel BFM bypasses Gen3 Phase 2 and Phase 3 Equalization. However, Gen3 variants can perform Phase 2 and Phase 3 equalization if instructed by a third-party BFM.

Related Links

PIPE Interface Signals on page 102

11.2 Transaction Layer

The Transaction Layer is located between the Application Layer and the Data Link Layer. It generates and receives Transaction Layer Packets. The following illustrates the Transaction Layer. The Transaction Layer includes three sub-blocks: the TX datapath, Configuration Space, and RX datapath.

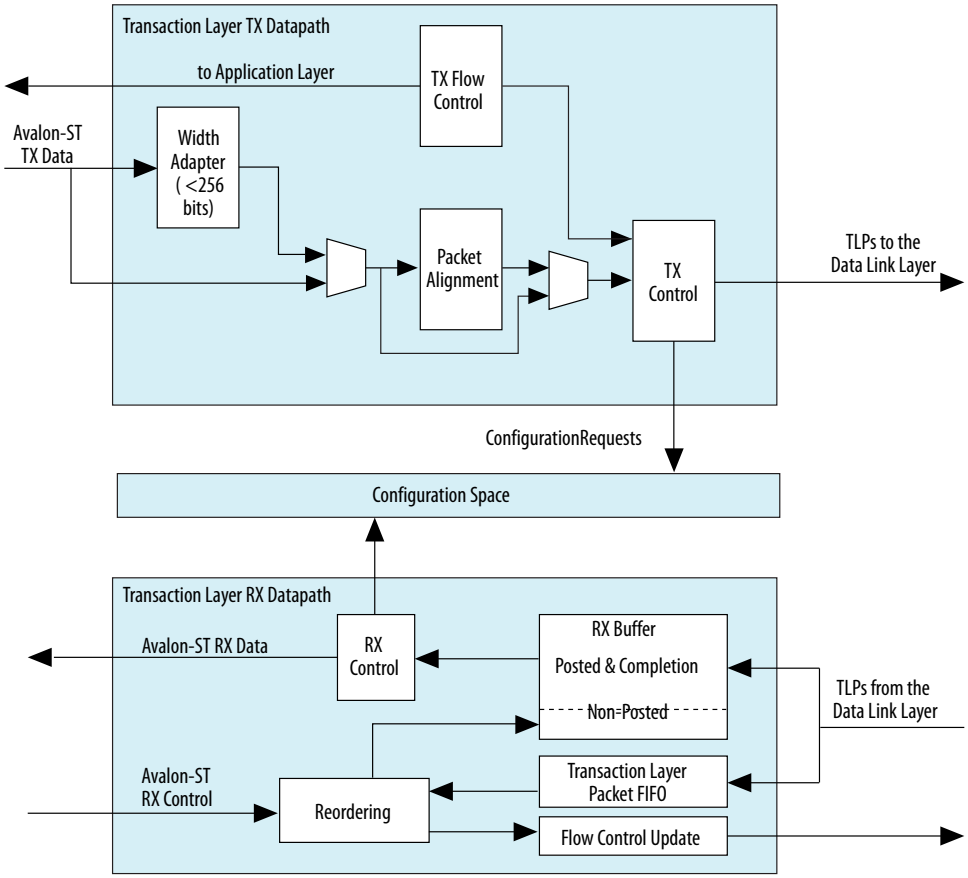
Tracing a transaction through the RX datapath includes the following steps:

1. The Transaction Layer receives a TLP from the Data Link Layer.
2. The Configuration Space determines whether the TLP is well formed and directs the packet based on traffic class (TC).
3. TLPs are stored in a specific part of the RX buffer depending on the type of transaction (posted, non-posted, and completion).
4. The TLP FIFO block stores the address of the buffered TLP.
5. The receive reordering block reorders the queue of TLPs as needed, fetches the address of the highest priority TLP from the TLP FIFO block, and initiates the transfer of the TLP to the Application Layer.
6. When ECRC generation and forwarding are enabled, the Transaction Layer forwards the ECRC dword to the Application Layer.

Tracing a transaction through the TX datapath involves the following steps:

1. The Transaction Layer informs the Application Layer that sufficient flow control credits exist for a particular type of transaction using the TX credit signals. The Application Layer may choose to ignore this information.
2. The Application Layer requests permission to transmit a TLP. The Application Layer must provide the transaction and must be prepared to provide the entire data payload in consecutive cycles.
3. The Transaction Layer verifies that sufficient flow control credits exist and acknowledges or postpones the request.
4. The Transaction Layer forwards the TLP to the Data Link Layer.

Figure 104. Architecture of the Transaction Layer: Dedicated Receive Buffer



11.2.1 Configuration Space

The Configuration Space implements the following configuration registers and associated functions:

- Header Type 0 Configuration Space for Endpoints
- Header Type 1 Configuration Space for Root Ports
- PCI Power Management Capability Structure
- Virtual Channel Capability Structure
- Message Signaled Interrupt (MSI) Capability Structure
- Message Signaled Interrupt–X (MSI–X) Capability Structure
- PCI Express Capability Structure
- Advanced Error Reporting (AER) Capability Structure
- Vendor Specific Extended Capability (VSEC)

The Configuration Space also generates all messages (PME#, INT, error, slot power limit), MSI requests, and completion packets from configuration requests that flow in the direction of the root complex, except slot power limit messages, which are

generated by a downstream port. All such transactions are dependent upon the content of the PCI Express Configuration Space as described in the *PCI Express Base Specification*.

Related Links

- Type 0 Configuration Space Registers on page 110
- Type 1 Configuration Space Registers on page 111
- PCI Express Base Specification 3.0

11.2.2.1 Error Checking and Handling in Configuration Space Bypass Mode

In Configuration Space Bypass mode, the Application Layer receives all TLPs that are not malformed. The Transaction Layer detects and drops malformed TLPs. The Transaction Layer also detects Internal Errors and Corrected Errors. Real-time error status signals report Internal Errors and Correctable Errors to the Application Layer. The Transaction Layer also records these errors in the AER registers. You can access the AER registers using the LMI interface.

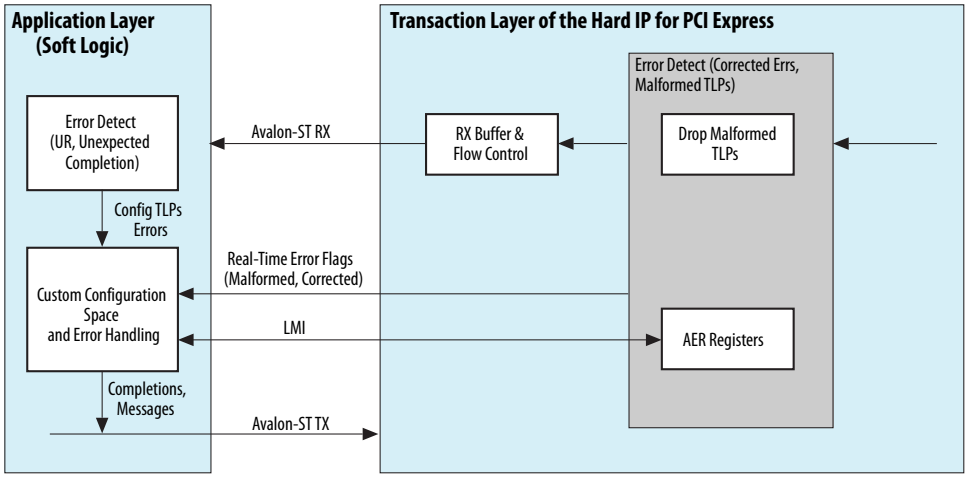
Because the AER header log is not available in Configuration Space Bypass Mode, the Application Layer must implement logic to read the AER header log using the LMI interface. You may need to arbitrate between Configuration Space Requests to the AER registers of the Hard IP for PCI Express and Configuration Space Requests to your own Configuration Space. Or, you can avoid arbitration logic by deasserting the `ready` signal until each LMI access completes.

Note: Intel does not support the use of the LMI interface to read and write the other registers in function0 of the Hard IP for PCI Express Configuration Space. You must create your own function0 in your application logic.

In Configuration Space Bypass mode, the Transaction Layer disables checks for Unsupported Requests and Unexpected Completions. The Application Layer must implement these checks. The Transaction Layer also disables error Messages and completion generation, which the Application Layer must implement.

Figure 105. Error Handing in Configuration Space Bypass Mode

This figure shows the division of error checking between the Transaction Layer of the hard IP for PCI Express and the Application Layer. The real-time error flags assert for one `pld_clk` as the errors are detected by the Transaction Layer.



This list summarizes the behavior of the Transaction Layer error handling in Configuration Space Bypass Mode:

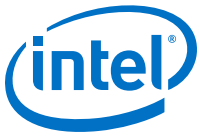
- The Translation Layer discards malformed TLPs. The `err_tlmalf` output signal is asserted to indicate this error. The Transaction Layer also logs this error in the `Uncorrectable Error Status`, `AER Header Log`, and `First Error Pointer Registers`. The Transaction Layer's definition of malformed TLPs is same in normal and Configuration Space Bypass modes.
- Unsupported Requests are not recognized by the Transaction Layer. The Application Layer must identify unsupported requests.
- Unexpected completions are not recognized by the Transaction Layer. The Application Layer must identify unexpected completions.
- You can use the Transaction Layer's ECRC checker in Configuration Space Bypass mode. If you enable ECRC checking with the `r x_ecrcchk_pld` input signal and the Transaction Layer detects an ECRC error, the Transaction Layer asserts the `rx_st_ecrcerr` output signal with the TLP on the Avalon-ST RX interface. The Application Layer must handle the error. If ECRC generation is enabled, the core generates ECRC and appends it to the end of the TX TLP from the Application Layer.
- The Transaction Layer sends poisoned TLPs on the Avalon-ST RX interface for completions and error handling by the Application Layer. These errors are not logged in the Configuration Space error registers.
- The Transaction Layer discards TLPs that violate RX credit limits. The Transaction Layers signals this error by asserting the `err_tlrcvovf` output signal and logging it in the `Uncorrectable Error Status`, `AER Header Log`, and `First Error Pointer Registers`.
- The Transaction Layer indicates Data Link and internal errors with the real-time error output signals `cfgbp_err_*`. These errors are also logged in the `Uncorrectable Error Status`, `AER Header Log`, and `First Error Pointer Registers`.

The Transaction Layer uses error flags to signal the Application Layer with real-time error status output signals. The Application Layer can monitor these flags to determine when the Transaction Layer has detected a Malformed TLP, Corrected Error, or internal error. In addition, the Application Layer can read the Transaction Layer's AER information such as AER Header Log and First Error Pointer Registers using the LMI bus.

- Real-time error signals are routed to the Application Layer using the error status output signals listed in the "Configuration Space Bypass Mode Output Signals" on page 8–44.
- Two sideband signals uncorr_err_reg_sts and corr_err_reg_sts indicate that an error has been logged in the Uncorrectable Error Status or Correctable Error Status Register. The Application Layer can read these Uncorrectable or Correctable Error Status Registers, AER Header Log, and First Error Pointers using the LMI bus to retrieve information. The uncorr_err_reg_sts and corr_err_reg_sts signals remain asserted until the Application Layer clears the corresponding status register. Proper logging requires that the Application Layer set the appropriate Configuration Space registers in the Transaction Layer using the LMI bus. The Application Layer must set the Uncorrectable and Correctable Error Mask and Uncorrec table Error Severity error reporting bits appropriately so that the errors are logged appropriately internal to the Arria 10 Hard IP for PCI Express. The settings of the Uncorrectable and Correctable Error Mask, and Uncorrectable Error Severity error reporting bits do not affect the real-time error output signals. The Application Layer must also log these errors in the soft Configuration Space and send error Messages.
- For more information about error handling, refer to the *PCI Express Base Specification*, Revision 2.0 or 3.0.
- The sideband signal root_err_reg_sts indicates that an error is logged in the Root Error Status Register. The Application Layer can read the Root Error Status Register and the Error Source Identification Register using the LMI bus to retrieve information about the errors. The root_err_reg_sts signal remains asserted until the Application Layer clears the corresponding status register using the LMI bus. The Application Layer must set the Uncorrectable and Correctable Error Mask, Uncorrectable Error Severity, and Device Control Register error reporting bits appropriately so that the errors are logged appropriately in the Arria 10 Hard IP for PCI Express IP Core. The settings of the Uncorrectable and Correctable Error Mask, Uncorrectable Error Severity, and Device Control Register error reporting bits do not affect the real-time error output signals. The Application Layer must also log these errors in the soft Configuration Space and send error Messages.

Related Links

PCI Express Base Specification 3.0



11.2.2.2 Protocol Extensions Supported

The Transaction Layer supports the following protocol extensions:

- TLP Processing Hints (TPH)—Supports both a Requester and Completer. The Application Layer should implement the TPH Requester Capabilities Structure using the soft logic in the Application Layer Extended Configuration Space. The Transaction Layer supports both Protocol Hint (PH) bits and Steering Tags (ST). The Transaction Layer does not support the optional Extended TPH TLP prefix.
- Atomic Operations—Supports both Requester and Completer. The RX buffer supports two, four, or eight non-posted data credits depending on the performance level you selected for the **RX buffer credit allocation—performance for received requests** under the **System Settings** heading of the parameter editor. The Transaction Layer also supports Atomic Operation Egress Blocking to prevent forwarding of AtomicOp Requests to components that should not receive them.
- ID-Based Ordering (IDO)—The Transaction Layer supports ID-Based Ordering to permit certain ordering restrictions to be relaxed to improve performance. However, the Transaction Layer does reorder the TLPs. On the RX side, ID-Based reordering should be implemented in soft logic. On the TX side, the Application Layer should set the IDO bit, which is bit 8 the `Device Control Register 2`, in the TLPs that it generates.

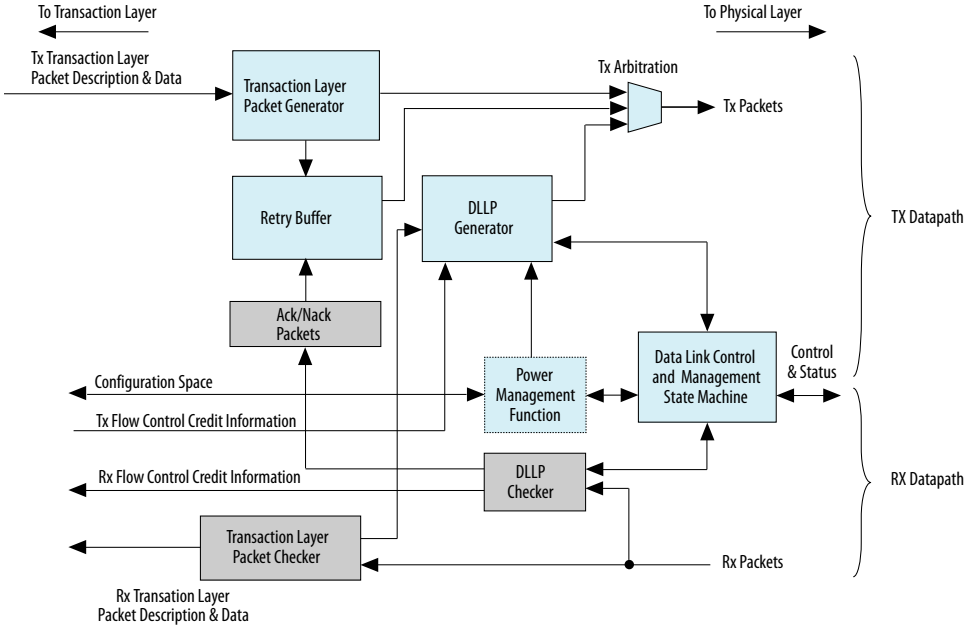
11.3 Data Link Layer

The Data Link Layer is located between the Transaction Layer and the Physical Layer. It maintains packet integrity and communicates (by DLL packet transmission) at the PCI Express link level.

The DLL implements the following functions:

- Link management through the reception and transmission of DLL packets (DLLP), which are used for the following functions:
  - Power management of DLLP reception and transmission
  - To transmit and receive ACK/NAK packets
  - Data integrity through generation and checking of CRCs for TLPs and DLLPs
  - TLP retransmission in case of NAK DLLP reception or replay timeout, using the retry (replay) buffer
  - Management of the retry buffer
  - Link retraining requests in case of error through the Link Training and Status State Machine (LTSSM) of the Physical Layer

Figure 106. Data Link Layer



The DLL has the following sub-blocks:

- Data Link Control and Management State Machine—This state machine connects to both the Physical Layer's LTSSM state machine and the Transaction Layer. It initializes the link and flow control credits and reports status to the Transaction Layer.
- Power Management—This function handles the handshake to enter low power mode. Such a transition is based on register values in the Configuration Space and received Power Management (PM) DLLPs. None of Arria 10 Hard IP for PCIe IP core variants support low power modes.
- Data Link Layer Packet Generator and Checker—This block is associated with the DLLP's 16-bit CRC and maintains the integrity of transmitted packets.
- Transaction Layer Packet Generator—This block generates transmit packets, including a sequence number and a 32-bit Link CRC (LCRC). The packets are also sent to the retry buffer for internal storage. In retry mode, the TLP generator receives the packets from the retry buffer and generates the CRC for the transmit packet.
- Retry Buffer—The retry buffer stores TLPs and retransmits all unacknowledged packets in the case of NAK DLLP reception. In case of ACK DLLP reception, the retry buffer discards all acknowledged packets.

- ACK/NAK Packets—The ACK/NAK block handles ACK/NAK DLLPs and generates the sequence number of transmitted packets.
- Transaction Layer Packet Checker—This block checks the integrity of the received TLP and generates a request for transmission of an ACK/NAK DLLP.
- TX Arbitration—This block arbitrates transactions, prioritizing in the following order:
  - Initialize FC Data Link Layer packet
  - ACK/NAK DLLP (high priority)
  - Update FC DLLP (high priority)
  - PM DLLP
  - Retry buffer TLP
  - TLP
  - Update FC DLLP (low priority)
  - ACK/NAK FC DLLP (low priority)

11.4 Physical Layer

The Physical Layer is the lowest level of the PCI Express protocol stack. It is the layer closest to the serial link. It encodes and transmits packets across a link and accepts and decodes received packets. The Physical Layer connects to the link through a high-speed SERDES interface running at 2.5 Gbps for Gen1 implementations, at 2.5 or 5.0 Gbps for Gen2 implementations, and at 2.5, 5.0 or 8.0 Gbps for Gen3 implementations.

The Physical Layer is responsible for the following actions:

- Training the link
- Scrambling/descrambling and 8B/10B encoding/decoding for 2.5 Gbps (Gen1), 5.0 Gbps (Gen2), or 128b/130b encoding/decoding of 8.0 Gbps (Gen3) per lane
- Serializing and deserializing data
- Equalization (Gen3)
- Operating the PIPE 3.0 Interface
- Implementing auto speed negotiation (Gen2 and Gen3)
- Transmitting and decoding the training sequence
- Providing hardware autonomous speed control
- Implementing auto lane reversal

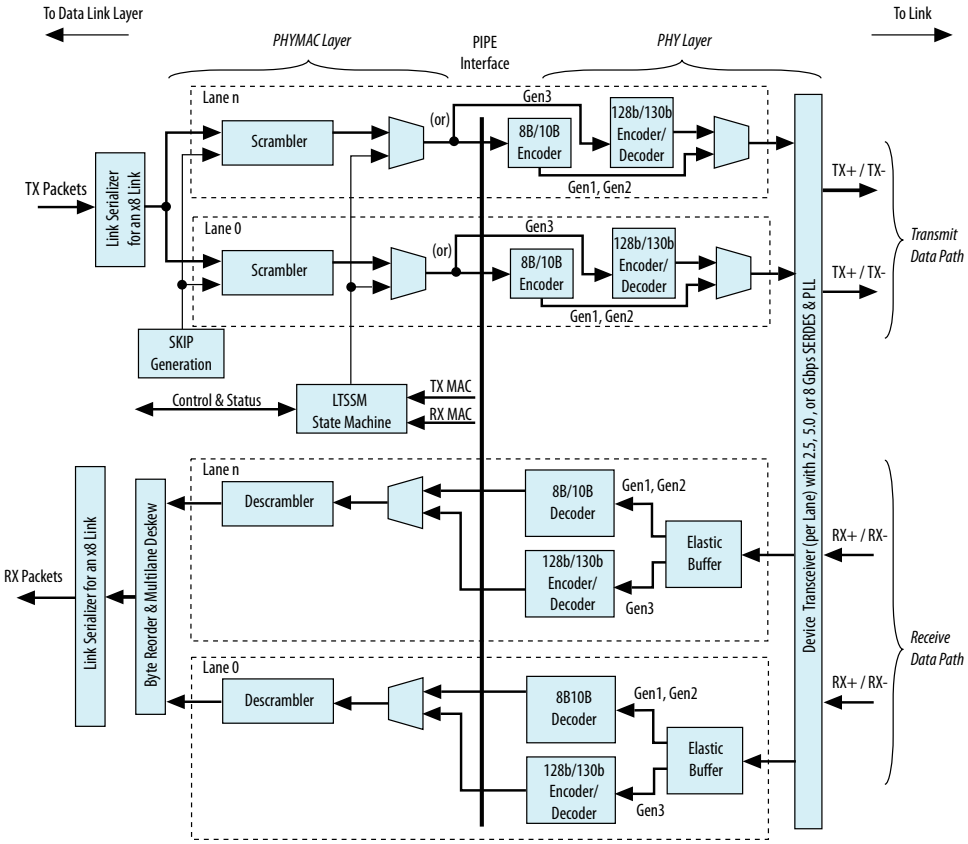
The Physical Layer is subdivided by the PIPE Interface Specification into two layers (bracketed horizontally in above figure):

- PHYMAC—The MAC layer includes the LTSSM and the scrambling/descrambling. byte reordering, and multilane deskew functions.
- PHY Layer—The PHY layer includes the 8B/10B encode and decode functions for Gen1 and Gen2. It includes 128b/130b encode and decode functions for Gen3. The PHY also includes elastic buffering and serialization/deserialization functions.

The Physical Layer integrates both digital and analog elements. Intel designed the PIPE interface to separate the PHYMAC from the PHY. The Arria 10 Hard IP for PCI Express complies with the PIPE interface specification.

Note: The internal PIPE interface is visible for simulation. It is not available for debugging in hardware using a logic analyzer such as Signal Tap. If you try to connect Signal Tap to this interface you will not be able to compile your design.

Figure 107. Physical Layer Architecture



The PHYMAC block comprises four main sub-blocks:

- MAC Lane—Both the RX and the TX path use this block.
  - On the RX side, the block decodes the Physical Layer packet and reports to the LTSSM the type and number of TS1/TS2 ordered sets received.
  - On the TX side, the block multiplexes data from the DLL and the Ordered Set and SKP sub-block (LTSTX). It also adds lane specific information, including the lane number and the force PAD value when the LTSSM disables the lane during initialization.
- LTSSM—This block implements the LTSSM and logic that tracks TX and RX training sequences on each lane.
- For transmission, it interacts with each MAC lane sub-block and with the LTSTX sub-block by asserting both global and per-lane control bits to generate specific Physical Layer packets.
  - On the receive path, it receives the Physical Layer packets reported by each MAC lane sub-block. It also enables the multilane deskew block. This block reports the Physical Layer status to higher layers.
  - LTSTX (Ordered Set and SKP Generation)—This sub-block generates the Physical Layer packet. It receives control signals from the LTSSM block and generates Physical Layer packet for each lane. It generates the same Physical Layer Packet for all lanes and PAD symbols for the link or lane number in the corresponding TS1/TS2 fields. The block also handles the receiver detection operation to the PCS sub-layer by asserting predefined PIPE signals and waiting for the result. It also generates a SKP Ordered Set at every predefined timeslot and interacts with the TX alignment block to prevent the insertion of a SKP Ordered Set in the middle of packet.
  - Deskew—This sub-block performs the multilane deskew function and the RX alignment between the initialized lanes and the datapath. The multilane deskew implements an eight-word FIFO buffer for each lane to store symbols. Each symbol includes eight data bits, one disparity bit, and one control bit.

12 Transaction Layer Protocol (TLP) Details

12.1 Supported Message Types

12.1.1 INTX Messages

Table 74. INTX Messages

Message	Root Port	Endpoint	Generated by			Comments
			App Layer	Core	Core (with App Layer input)	
Assert_INTA	Receive	Transmit	No	Yes	No	For Root Port, legacy interrupts are translated into message interrupt TLPs which triggers the `int_status[3:0]` signals to the Application Layer. • `int_status[0]`: Interrupt signal A • `int_status[1]`: Interrupt signal B • `int_status[2]`: Interrupt signal C • `int_status[3]`: Interrupt signal D
Assert_INTB	Receive	Transmit	No	No	No	
Assert_INTC	Receive	Transmit	No	No	No	
Assert_INTD	Receive	Transmit	No	No	No	
Deassert_INTA	Receive	Transmit	No	Yes	No	
Deassert_INTB	Receive	Transmit	No	No	No	
Deassert_INTC	Receive	Transmit	No	No	No	
Deassert_INTD	Receive	Transmit	No	No	No	

12.1.2 Power Management Messages

Table 75. Power Management Messages

Message	Root Port	Endpoint	Generated by			Comments
			App Layer	Core	Core (with App Layer input)	
PM_Active_State_Nak	TX	RX	No	Yes	No	—
PM_PME	RX	TX	No	No	Yes	—
PME_Turn_Off	TX	RX	No	No	Yes	The `pme_to_cr` signal sends and acknowledges this message: • Root Port: When `pme_to_cr` is asserted, the Root Port sends the `PME_turn_off` message. • Endpoint: When `PME_to_cr` is asserted, the Endpoint acknowledges the `PME_turn_off` message by sending a `pme_to_ack` message to the Root Port.
PME_TO_Ack	RX	TX	No	No	Yes	—

12.1.3 Error Signaling Messages

Table 76. Error Signaling Messages

Message	Root Port	Endpoint	Generated by			Comments
			App Layer	Core	Core (with App Layer input)	
ERR_COR	RX	TX	No	Yes	No	In addition to detecting errors, a Root Port also gathers and manages errors sent by downstream components through the ERR_COR, ERR_NONFATAL, AND ERR_FATAL Error Messages. In Root Port mode, there are two mechanisms to report an error event to the Application Layer: • `serr_out` output signal. When set, indicates to the Application Layer that an error has been logged in the AER capability structure • `aer_msi_num` input signal. When the **Implement advanced error reporting** option is turned on, you can set `aer_msi_num` to indicate which MSI is being sent to the root complex when an error is logged in the AER Capability structure.
ERR_NONFATAL	RX	TX	No	Yes	No	—
ERR_FATAL	RX	TX	No	Yes	No	—

12.1.4 Locked Transaction Message

Table 77. Locked Transaction Message

Message	Root Port	Endpoint	Generated by			Comments
			App Layer	Core	Core (with App Layer input)	
Unlock Message	Transmit	Receive	Yes	No	No	

12.1.5 Slot Power Limit Message

The *PCI Express Base Specification Revision* states that this message is not mandatory after link training.

Table 78. Slot Power Message

Message	Root Port	Endpoint	Generated by			Comments
			App Layer	Core	Core (with App Layer input)	
Set Slot Power Limit	Transmit	Receive	No	Yes	No	In Root Port mode, through software.

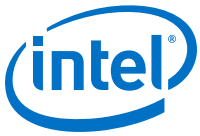
Related Links

PCI Express Base Specification Revision 3.0

12.1.6 Vendor-Defined Messages

Table 79. Vendor-Defined Message

Message	Root Port	Endpoint	Generated by			Comments
			App Layer	Core	Core (with App Layer input)	
Vendor Defined Type 0	Transmit Receive	Transmit Receive	Yes	No	No	
Vendor Defined Type 1	Transmit Receive	Transmit Receive	Yes	No	No	



12.1.7 Hot Plug Messages

Table 80. Locked Transaction Message

Message	Root Port	Endpoint	Generated by			Comments
			App Layer	Core	Core (with App Layer input)	
Attention_indicator On	Transmit	Receive	No	Yes	No	Per the recommendations in the *PCI Express Base Specification Revision* , these messages are not transmitted to the Application Layer.
Attention_Indicator Blink	Transmit	Receive	No	Yes	No	
Attention_indicator Off	Transmit	Receive	No	Yes	No	
Power_Indicator On	Transmit	Receive	No	Yes	No	
Power_Indicator Blink	Transmit	Receive	No	Yes	No	
Power_Indicator Off	Transmit	Receive	No	Yes	No	
Attention Button_Pressed (Endpoint only)	Receive	Transmit	No	No	Yes	N/A

Related Links

PCI Express Base Specification Revision 3.0

12.2 Transaction Layer Routing Rules

Transactions adhere to the following routing rules:

- In the receive direction (from the PCI Express link), memory and I/O requests that match the defined base address register (BAR) contents and vendor-defined messages with or without data route to the receive interface. The Application Layer logic processes the requests and generates the read completions, if needed.
- In Endpoint mode, received Type 0 Configuration requests from the PCI Express upstream port route to the internal Configuration Space and the Arria 10 Hard IP for PCI Express generates and transmits the completion.
- The Hard IP handles supported received message transactions (Power Management and Slot Power Limit) internally. The Endpoint also supports the Unlock and Type 1 Messages. The Root Port supports Interrupt, Type 1, and error Messages.
- Vendor-defined Type 0 and Type 1 Message TLPs are passed to the Application Layer.
- The Transaction Layer treats all other received transactions (including memory or I/O requests that do not match a defined BAR) as Unsupported Requests. The Transaction Layer sets the appropriate error bits and transmits a completion, if needed. These Unsupported Requests are not made visible to the Application Layer; the header and data are dropped.

- For memory read and write request with addresses below 4 GB, requestors must use the 32-bit format. The Transaction Layer interprets requests using the 64-bit format for addresses below 4 GB as an Unsupported Request and does not send them to the Application Layer. If Error Messaging is enabled, an error Message TLP is sent to the Root Port. Refer to *Transaction Layer Errors* for a comprehensive list of TLPs the Hard IP does not forward to the Application Layer.
- The Transaction Layer sends all memory and I/O requests, as well as completions generated by the Application Layer and passed to the transmit interface, to the PCI Express link.
- The Hard IP can generate and transmit power management, interrupt, and error signaling messages automatically under the control of dedicated signals. Additionally, it can generate MSI requests under the control of the dedicated signals.
- In Root Port mode, the Application Layer can issue Type 0 or Type 1 Configuration TLPs on the Avalon-ST TX bus.
- The Type 0 Configuration TLPs are only routed to the Configuration Space of the Hard IP and are not sent downstream on the PCI Express link.
- The Type 1 Configuration TLPs are sent downstream on the PCI Express link. If the bus number of the Type 1 Configuration TLP matches the Secondary Bus Number register value in the Root Port Configuration Space, the TLP is converted to a Type 0 TLP.
- For more information about routing rules in Root Port mode, refer to *Section 7.3.3 Configuration Request Routing Rules* in the *PCI Express Base Specification* .

Related Links

- Transaction Layer Errors on page 134
- PCI Express Base Specification Revision 3.0

12.3 Receive Buffer Reordering

The PCI, PCI-X and PCI Express protocols include ordering rules for concurrent TLPs. Ordering rules are necessary for the following reasons:

- To guarantee that TLPs complete in the intended order
- To avoid deadlock
- To maintain computability with ordering used on legacy buses
- To maximize performance and throughput by minimizing read latencies and managing read/write ordering
- To avoid race conditions in systems that include legacy PCI buses by guaranteeing that reads to an address do not complete before an earlier write to the same address

PCI uses a strongly-ordered model with some exceptions to avoid potential deadlock conditions. PCI-X added a relaxed ordering (RO) bit in the TLP header. It is bit 5 of byte 2 in the TLP header, or the high-order bit of the `attributes` field in the TLP header. If this bit is set, relaxed ordering is permitted. If software can guarantee that no dependencies exist between pending transactions, you can safely set the relaxed ordering bit.

The following table summarizes the ordering rules from the PCI specification. In this table, the entries have the following meanings:

- Columns represent the first transaction issued.
- Rows represent the next transaction.
- At each intersection, the implicit question is: should this row packet be allowed to pass the column packet? The following three answers are possible:
  - Yes: the second transaction must be allowed to pass the first to avoid deadlock.
  - Y/N: There are no requirements. A device may allow the second transaction to pass the first.
  - No: The second transaction must not be allowed to pass the first.

The following transaction ordering rules apply to the table below.

- A Memory Write or Message Request with the Relaxed Ordering Attribute bit clear (b'0) must not pass any other Memory Write or Message Request.
- A Memory Write or Message Request with the Relaxed Ordering Attribute bit set (b'1) is permitted to pass any other Memory Write or Message Request.
- Endpoints, Switches, and Root Complex may allow Memory Write and Message Requests to pass Completions or be blocked by Completions.
- Memory Write and Message Requests can pass Completions traveling in the PCI Express to PCI directions to avoid deadlock.
- If the Relaxed Ordering attribute is not set, then a Read Completion cannot pass a previously enqueued Memory Write or Message Request.
- If the Relaxed Ordering attribute is set, then a Read Completion is permitted to pass a previously enqueued Memory Write or Message Request.
- Read Completion associated with different Read Requests are allowed to be blocked by or to pass each other.
- Read Completions for Request (same Transaction ID) must return in address order.
- Non-posted requests cannot pass other non-posted requests.
- `CfgRd0`CfgRd0 can pass `IORd` or `MRd`.
- `CfgWr0` can `IORd` or `MRd`.
- `CfgRd0` can pass `IORd` or `MRd`.
- `CfrWr0` can pass `IOWr`.

Table 81. Transaction Ordering Rules

Can the Row Pass the Column?		Posted Req		Non Posted Req				Completion	
		Memory Write or Message Req		Read Request		I/O or Cfg Write Req			
		Spec	Hard IP	Spec	Hard IP	Spec	Hard IP	Spec	Hard IP
P	Posted Req	No Y/N	No No	Yes	Yes	Yes	Yes	Y/N Yes	No No
NP	Read Req	No	No	Y/N	No	Y/N	No	Y/N	No

continued...

Can the Row Pass the Column?		Posted Req		Non Posted Req				Completion	
		Memory Write or Message Req		Read Request		I/O or Cfg Write Req			
	Non-Posted Req with data	No	No	Y/N	No	Y/N	No	Y/N	No
Cmpl	Cmpl	No Y/N	No No	Yes	Yes	Yes	Yes	Y/N No	No No
	I/O or Configuration Write Cmpl	Y/N	No	Yes	Yes	Yes	Yes	Y/N	No

As the table above indicates, the RX datapath implements an RX buffer reordering function that allows Posted and Completion transactions to pass Non-Posted transactions (as allowed by PCI Express ordering rules) when the Application Layer is unable to accept additional Non-Posted transactions.

The Application Layer dynamically enables the RX buffer reordering by asserting the `rx_mask` signal. The `rx_mask` signal blocks non-posted Req transactions made to the Application Layer interface so that only posted and completion transactions are presented to the Application Layer.

Note: MSI requests are conveyed in exactly the same manner as PCI Express memory write requests and are indistinguishable from them in terms of flow control, ordering, and data integrity.

Related Links

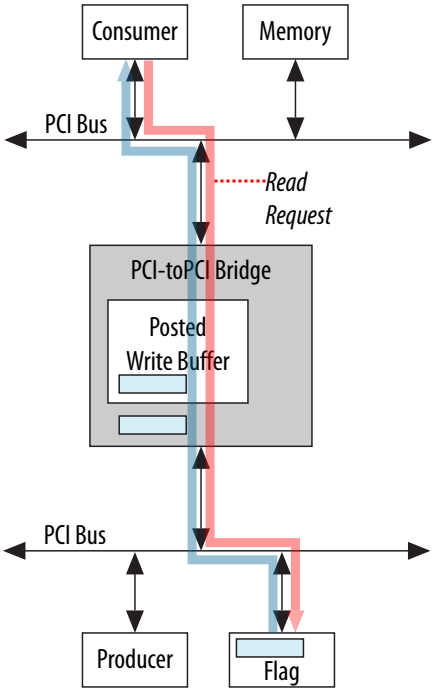
PCI Express Base Specification Revision 3.0

12.3.1 Using Relaxed Ordering

Transactions from unrelated threads are unlikely to have data dependencies. Consequently, you may be able to use relaxed ordering to improve system performance. The drawback is that only some transactions can be optimized for performance. Complete the following steps to decide whether to enable relaxed ordering in your design:

1. Create a system diagram showing all PCI Express and legacy devices.
2. Analyze the relationships between the components in your design to identify the following hazards:
   a. Race conditions: A race condition exists if a read to a location can occur before a previous write to that location completes. The following figure shows a data producer and data consumer on opposite sides of a PCI-to-PCI bridge. The producer writes data to the memory through a PCI-to-PCI bridge. The consumer must read a flag to confirm the producer has written the new data into the memory before reading the data. However, because the PCI-to-PCI bridge includes a write buffer, the flag may indicate that it is safe to read data while the actual data remains in the PCI-to-PCI bridge posted write buffer.

Figure 108. Design Including Legacy PCI Buses Requiring Strong Ordering

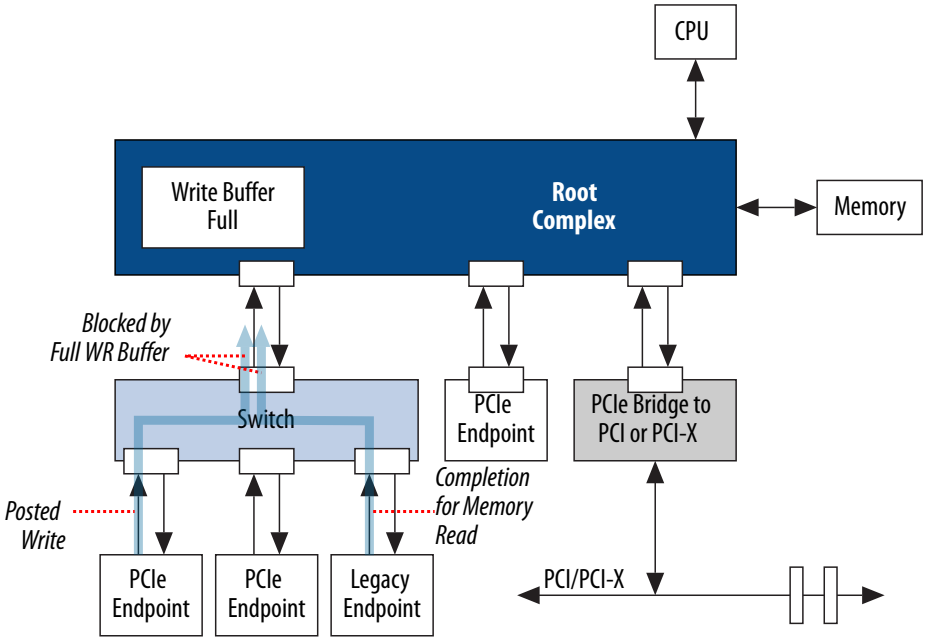


b. A shared memory architecture where more than one thread accesses the same locations in memory.

If either of these conditions exists, relaxed ordering will lead to incorrect results.

3. If your analysis determines that relaxed ordering does not lead to possible race conditions or read or write hazards, you can enable relaxed ordering by setting the RO bit in the TLP header.
4. The following figure shows two PCIe Endpoints and Legacy Endpoint connected to a switch. The three PCIe Endpoints are not likely to have data dependencies. Consequently, it would be safe to set the relaxed ordering bit for devices connected to the switch. In this system, if relax ordering is not enabled, a memory read to the legacy Endpoint is blocked. The legacy Endpoint read is blocked because an earlier posted write cannot be completed as the write buffer is full. .

Figure 109. PCI Express Design Using Relaxed Ordering



5. If your analysis indicates that you can enable relaxed ordering, simulate your system with and without relaxed ordering enabled. Compare the results and performance.
6. If relaxed ordering improves performance without introducing errors, you can enable it in your system.

13 Throughput Optimization

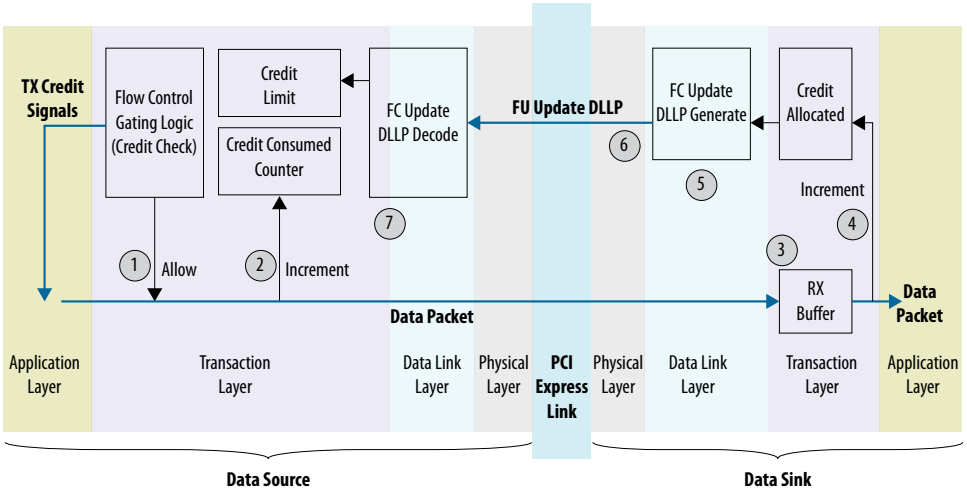
The *PCI Express Base Specification* defines a flow control mechanism to ensure efficient transfer of TLPs.

Each transmitter, the write requester in this case, maintains a `credit limit` register and a `credits consumed` register. The `credit limit` register is the sum of all credits received by the receiver, the write completer in this case. The `credit limit` register is initialized during the flow control initialization phase of link initialization and then updated during operation by Flow Control (FC) Update DLLPs. The `credits consumed` register is the sum of all credits consumed by packets transmitted. Separate `credit limit` and `credits consumed` registers exist for each of the six types of Flow Control:

- Posted Headers
- Posted Data
- Non-Posted Headers
- Non-Posted Data
- Completion Headers
- Completion Data

Each receiver also maintains a `credit allocated` counter which is initialized to the total available space in the RX buffer (for the specific Flow Control class) and then incremented as packets are pulled out of the RX buffer by the Application Layer. The value of this register is sent as the FC Update DLLP value.

Figure 110. Flow Control Update Loop



The following numbered steps describe each step in the Flow Control Update loop. The corresponding numbers in the figure show the general area to which they correspond.

1. When the Application Layer has a packet to transmit, the number of credits required is calculated. If the current value of the credit limit minus credits consumed is greater than or equal to the required credits, then the packet can be transmitted immediately. However, if the credit limit minus credits consumed is less than the required credits, then the packet must be held until the credit limit is increased to a sufficient value by an FC Update DLLP. This check is performed separately for the header and data credits; a single packet consumes only a single header credit.
2. After the packet is selected for transmission the `credits consumed` register is incremented by the number of credits consumed by this packet. This increment happens for both the header and data `credit consumed` registers.
3. The packet is received at the other end of the link and placed in the RX buffer.
4. At some point the packet is read out of the RX buffer by the Application Layer. After the entire packet is read out of the RX buffer, the `credit allocated` register can be incremented by the number of credits the packet has used. There are separate `credit allocated` registers for the header and data credits.
5. The value in the `credit allocated` register is used to create an FC Update DLLP.
6. After an FC Update DLLP is created, it arbitrates for access to the PCI Express link. The FC Update DLLPs are typically scheduled with a low priority; consequently, a continuous stream of Application Layer TLPs or other DLLPs (such as ACKs) can delay the FC Update DLLP for a long time. To prevent starving the attached transmitter, FC Update DLLPs are raised to a high priority under the following three circumstances:
   a. When the last sent `credit allocated` counter minus the amount of received data is less than MAX_PAYLOAD and the current `credit allocated` counter is greater than the last sent credit counter. Essentially, this means the data sink knows the data source has less than a full MAX_PAYLOAD worth of credits, and therefore is starving.
   b. When an internal timer expires from the time the last FC Update DLLP was sent, which is configured to 30 µs to meet the *PCI Express Base Specification* for resending FC Update DLLPs.
   c. When the `credit allocated` counter minus the last sent `credit allocated` counter is greater than or equal to 25% of the total credits available in the RX buffer, then the FC Update DLLP request is raised to high priority.

      After arbitrating, the FC Update DLLP that won the arbitration to be the next item is transmitted. In the worst case, the FC Update DLLP may need to wait for a maximum sized TLP that is currently being transmitted to complete before it can be sent.
7. The original write requester receives the FC Update DLLP. The `credit limit` value is updated. If packets are stalled waiting for credits, they can now be transmitted.

Note: You must keep track of the credits consumed by the Application Layer.

13.1 Throughput of Posted Writes

The throughput of posted writes is limited primarily by the Flow Control Update loop as shown in Figure 110 on page 161. If the write requester sources the data as quickly as possible, and the completer consumes the data as quickly as possible, then the Flow Control Update loop may be the biggest determining factor in write throughput, after the actual bandwidth of the link.

The figure below shows the main components of the Flow Control Update loop with two communicating PCI Express ports:

- Write Requester
- Write Completer

To allow the write requester to transmit packets continuously, the `credit allocated` and the `credit limit` counters must be initialized with sufficient credits to allow multiple TLPs to be transmitted while waiting for the FC Update DLLP that corresponds to the freeing of credits from the very first TLP transmitted.

You can use the **RX Buffer space allocation - Desired performance for received requests** to configure the RX buffer with enough space to meet the credit requirements of your system.

Related Links

PCI Express Base Specification 3.0

13.2 Throughput of Non-Posted Reads

To support a high throughput for read data, you must analyze the overall delay from the time the Application Layer issues the read request until all of the completion data is returned. The Application Layer must be able to issue enough read requests, and the read completer must be capable of processing these read requests quickly enough (or at least offering enough non-posted header credits) to cover this delay.

However, much of the delay encountered in this loop is well outside the IP core and is very difficult to estimate. PCI Express switches can be inserted in this loop, which makes determining a bound on the delay more difficult.

Nevertheless, maintaining maximum throughput of completion data packets is important. Endpoints must offer an infinite number of completion credits. Endpoints must buffer this data in the RX buffer until the Application Layer can process it. Because the Endpoint is no longer managing the RX buffer for Completions through the flow control mechanism, the Application Layer must manage the RX buffer by the rate at which it issues read requests.

To determine the appropriate settings for the amount of space to reserve for completions in the RX buffer, you must make an assumption about the length of time until read completions are returned. This assumption can be estimated in terms of an additional delay, beyond the FC Update Loop Delay, as discussed in the section *Throughput of Posted Writes*. The paths for the read requests and the completions are not exactly the same as those for the posted writes and FC Updates in the PCI Express logic. However, the delay differences are probably small compared with the inaccuracy in the estimate of the external read to completion delays.

With multiple completions, the number of available credits for completion headers must be larger than the completion data space divided by the maximum packet size. Instead, the credit space for headers must be the completion data space (in bytes) divided by 64, because this is the smallest possible read completion boundary. Setting the **RX Buffer space allocation—Desired performance for received completions** to **High** under the **System Settings** heading when specifying parameter settings configures the RX buffer with enough space to meet this requirement. You can adjust this setting up or down from the **High** setting to tailor the RX buffer size to your delays and required performance.

You can also control the maximum amount of outstanding read request data. This amount is limited by the number of header tag values that can be issued by the Application Layer and by the maximum read request size that can be issued. The number of header tag values that can be in use is also limited by the IP core. You can specify 32 or 64 tags though configuration software to restrict the Application Layer to use only 32 tags. In commercial PC systems, 32 tags are usually sufficient to maintain optimal read throughput.

14 Design Implementation

Completing your design includes additional steps to specify analog properties, pin assignments, and timing constraints.

14.1 Making Pin Assignments to Assign I/O Standard to Serial Data Pins

Before running Quartus Prime compilation, use the **Pin Planner** to assign I/O standards to the pins of the device.

1. On the Quartus Prime **Assignments** menu, select **Pin Planner**. The **Pin Planner** appears.
2. In the **Node Name** column, locate the PCIe serial data pins.
3. In the **I/O Standard** column, double-click the right-hand corner of the box to bring up a list of available I/O standards.
4. Select the appropriate standard from the following table.

Table 82. I/O Standards for HSSI Pins

Pin Type	I/O Standard
HSSI REFCLK	**Current Mode Logic (CML)**, **HCSL**
HSSI RX	**Current Mode Logic (CML)**
HSSI TX	**High Speed Differential I/O**

The Quartus Prime software adds instance assignments to your Quartus Prime Settings File (`*.qsf`). The assignment is in the form `set_instance_assignment -name IO_STANDARD <"IO_STANDARD_NAME"> -to <signal_name>`. The `*.qsf` is in your synthesis directory.

Related Links

Arria 10 GX, GT, and SX Device Family Pin Connection Guidelines
For information about connecting pins on the PCB including required resistor values and voltages.

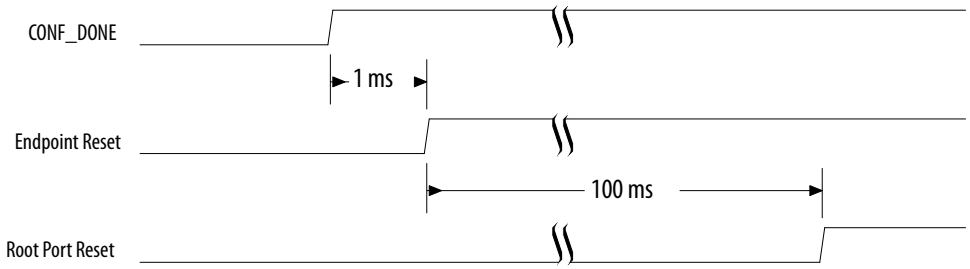
14.2 Recommended Reset Sequence to Avoid Link Training Issues

Successful link training can only occur after the FPGA is configured. Designs using CvP for configuration initially load the I/O ring and periphery image. Arria 10 devices include a Nios II Hard Calibration IP core that automatically calibrates transceivers to optimize signal quality after CvP completes and before entering user mode. Link training occurs after calibration. Refer to *Reset Sequence for Hard IP for PCI Express IP Core and Application Layer* for a description of the key signals that reset, control

dynamic reconfiguration, and link training. Intel recommends separate control of reset signals for the Endpoint and Root Port. Successful reset sequence includes the following steps:

1. Wait until the FPGA is configured as indicated by the assertion of CONFIG_DONE from the FPGA block controller.
2. Wait 1 ms after the assertion of CONFIG_DONE, then deassert the Endpoint reset.
3. Wait approximately 100 ms, then deassert the Root Port reset.
4. Deassert the reset output to the Application Layer.

Figure 111. Recommended Reset Sequence



Related Links

Intel FPGA Arria 10 Transceiver PHY IP Core User Guide
For information about requirements for the CLKUSR pin used during automatic calibration.

14.3 Creating a Signal Tap II Debug File to Match Your Design Hierarchy

For Arria 10 devices, the Quartus Prime Standard Edition software generates two files, `build_stp.tcl` and `<ip_core_name>.xml`. You can use these files to generate a Signal Tap II file with probe points matching your design hierarchy.

The Quartus Prime software stores these files in the `<IP core directory>/synth/debug/stp/` directory.

Synthesize your design using the Quartus Prime software.

1. To open the Tcl console, click **View ➤ Utility Windows ➤ Tcl Console**.
2. Type the following command in the Tcl console:
   `source <IP core directory>/synth/debug/stp/build_stp.tcl`
3. To generate the STP file, type the following command:
   `main -stp_file <output stp file name>.stp -xml_file <input xml_file name>.xml -mode build`
4. To add this Signal Tap II file (**.stp**) to your project, select **Project ➤ Add/Remove Files in Project**. Then, compile your design.
5. To program the FPGA, click **Tools ➤ Programmer**.

6. To start the Signal Tap II Logic Analyzer, click **Quartus Prime ➤ Tools ➤ Signal Tap II Logic Analyzer**.

   The software generation script may not assign the Signal Tap II acquisition clock in `<output stp file name>.stp`. Consequently, the Quartus Prime software automatically creates a clock pin called `auto_stp_external_clock`. You may need to manually substitute the appropriate clock signal as the Signal Tap II sampling clock for each STP instance.

7. Recompile your design.
8. To observe the state of your IP core, click **Run Analysis**.

   You may see signals or Signal Tap II instances that are red, indicating they are not available in your design. In most cases, you can safely ignore these signals and instances. They are present because software generates wider buses and some instances that your design does not include.

14.4 SDC Timing Constraints

Your top-level Synopsys Design Constraints file (`.sdc`) must include the following timing constraint macro for the Arria 10 Hard IP for PCIe IP core.

Example 1. SDC Timing Constraints Required for the Arria 10 Hard IP for PCIe and Design Example

```
# Constraints required for the Arria 10 Hard IP for PCI Express
# derive_pll_clock is used to calculate all clock derived
# from PCIe refclk. It must be applied once across all
# of the SDC files used in a project
derive_pll_clocks -create_base_clocks
```

You should only include this constraint in one location across all of the SDC files in your project. Differences between Fitter timing analysis and TimeQuest timing analysis arise if these constraints are applied multiple times.

Related Links

What assignments do I need for a PCIe Gen1, Gen2 or Gen3 design that targets an Arria 10 ES2, ES3 or production device?

Starting with the Quartus Prime Software Release 17.0, these assignments are automatically included in the design. You do not have to add them.

15 Optional Features

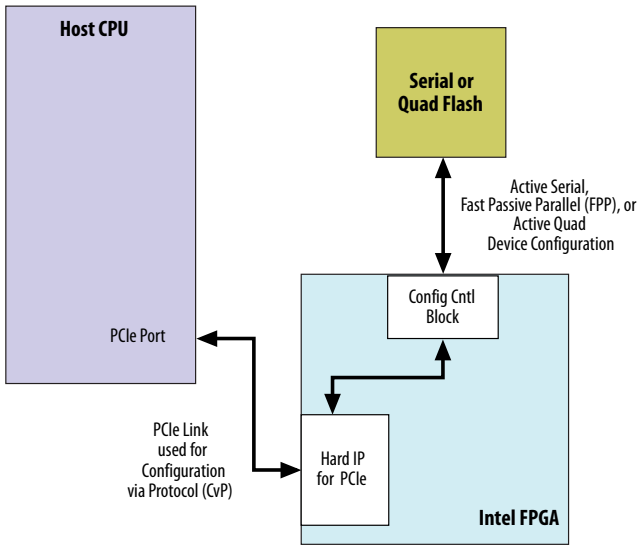
15.1 Configuration over Protocol (CvP)

The Hard IP for PCI Express architecture has an option to configure the FPGA and initialize the PCI Express link. In prior devices, a single Program Object File (**.pof**) programmed the I/O ring and FPGA fabric before the PCIe link training and enumeration began. The **.pof** file is divided into two parts:

- The I/O bitstream contains the data to program the I/O ring, the Hard IP for PCI Express, and other elements that are considered part of the periphery image.
- The core bitstream contains the data to program the FPGA fabric.

When you select the CvP design flow, the I/O ring and PCI Express link are programmed first, allowing the PCI Express link to reach the L0 state and begin operation independently, before the rest of the core is programmed. After the PCI Express link is established, it can be used to program the rest of the device. The following figure shows the blocks that implement CvP.

Figure 112. CvP in Arria 10 Devices



CvP has the following advantages:

- Provides a simpler software model for configuration. A smart host can use the PCIe protocol and the application topology to initialize and update the FPGA fabric.
- Improves security for the proprietary core bitstream.
- Reduces system costs by reducing the size of the flash device to store the **.pof**.
- May reduce system size because a single CvP link can be used to configure multiple FPGAs.

Related Links

Arria 10 CvP Initialization and Partial Reconfiguration over PCI Express User Guide

15.2 Autonomous Mode

Autonomous mode allows the PCIe IP core to operate before the device enters user mode, while the core is being configured.

Intel's FPGA devices always receive the configuration bits for the periphery image first, then for the core image. After the core image configures, the device enters user mode. In autonomous mode, the hard IP for PCI Express begins operation when the periphery configuration completes, before it enters user mode.

In autonomous mode, after completing link training, the Hard IP for PCI Express responds to Configuration Requests from the host with a Configuration Request Retry Status (CRRS). Autonomous mode is when you must meet the 100 ms PCIe wake-up time.

The hard IP for PCIe responds with CRRS under the following conditions:

- Before the core fabric is programmed when you enable autonomous mode.
- Before the core fabric is programmed when you enable initialization of the core fabric using the PCIe link.

All PCIe IP cores on a device can operate in autonomous mode. However, only the bottom Hard IP for PCI Express on either side can satisfy the 100 ms PCIe wake up time requirement. Tansceiver calibration begins with the bottom PCIe IP core on each side of the device. Consequently, this IP core has a faster wake up time.

Arria V, Cyclone V, Stratix V, and Arria 10 devices are the first to offer autonomous mode. In earlier devices, the PCI Express Hard IP Core exits reset only after full FPGA configuration.

Related Links

- Enabling Autonomous Mode on page 169
  These steps specify autonomous mode in the Quartus Prime software.
- Enabling CvP Initialization on page 170
  These steps enable CvP initialization mode in the Quartus Prime software.

15.2.1 Enabling Autonomous Mode

These steps specify autonomous mode in the Quartus Prime software.

1. On the Quartus Prime Assignments menu, select **Device ➤ Device and Pin Options**.
2. Under **Category ➤ General** turn on **Enable autonomous PCIe HIP mode**. The **Enable autonomous PCIe HIP mode** option has an effect if your design has the following two characteristics:
   - You are using the Flash device or Ethernet controller, instead of the PCIe link to load the core image.
   - You have not turned on **Enable Configuration via the PCIe link** in the Hard IP for PCI Express GUI.

15.2.2 Enabling CvP Initialization

These steps enable CvP initialization mode in the Quartus Prime software.

1. On the Assignments menu select **Device ➤ Device and Pin Options.**
2. Under **Category**, select **CvP Settings**.
3. For **Configuration via Protocol**, select **Core initialization** from the drop-down menu.

15.3 ECRC

ECRC ensures end-to-end data integrity for systems that require high reliability. You can specify this option under the **Error Reporting** heading. The ECRC function includes the ability to check and generate ECRC. In addition, the ECRC function can forward the TLP with ECRC to the RX port of the Application Layer. When using ECRC forwarding mode, the ECRC check and generation are performed in the Application Layer.

You must turn on **Advanced error reporting (AER)**, **ECRC checking**, and **ECRC generation** under the **PCI Express/PCI Capabilities** heading using the parameter editor to enable this functionality.

For more information about error handling, refer to *Error Signaling and Logging* in Section 6.2 of the *PCI Express Base Specification*.

Related Links

PCI Express Base Specification 3.0

15.3.1 ECRC on the RX Path

When the **ECRC generation** option is turned on, errors are detected when receiving TLPs with a bad ECRC. If the **ECRC generation** option is turned off, no error detection occurs. If the **ECRC forwarding** option is turned on, the ECRC value is forwarded to the Application Layer with the TLP. If the **ECRC forwarding** option is turned off, the ECRC value is not forwarded.

Table 83. ECRC Operation on RX Path

ECRC Forwarding	ECRC Check Enable [4]	ECRC Status	Error	TLP Forward to Application Layer
No	No	none	No	Forwarded
		good	No	Forwarded without its ECRC
		bad	No	Forwarded without its ECRC
	Yes	none	No	Forwarded
		good	No	Forwarded without its ECRC
		bad	Yes	Not forwarded
Yes	No	none	No	Forwarded
		good	No	Forwarded with its ECRC
		bad	No	Forwarded with its ECRC
	Yes	none	No	Forwarded
		good	No	Forwarded with its ECRC
		bad	Yes	Not forwarded

15.3.2 ECRC on the TX Path

When the **ECRC generation** option is on, the TX path generates ECRC. If you turn on **ECRC forwarding**, the ECRC value is forwarded with the TLP. The following table summarizes the TX ECRC generation and forwarding. All unspecified cases are unsupported and the behavior of the Hard IP is unknown.In this table, if TD is 1, the TLP includes an ECRC. TD is the TL digest bit of the TL packet.

Table 84. ECRC Generation and Forwarding on TX Path

All unspecified cases are unsupported and the behavior of the Hard IP is unknown.

ECRC Forwarding	ECRC Generation Enable [5]	TLP on Application	TLP on Link	Comments
No	No	TD=0, without ECRC	TD=0, without ECRC	
		TD=1, without ECRC	TD=0, without ECRC	
	Yes	TD=0, without ECRC	TD=1, with ECRC	ECRC is generated
		TD=1, without ECRC	TD=1, with ECRC	
Yes	No	TD=0, without ECRC	TD=0, without ECRC	Core forwards the ECRC

continued...

4 The `ECRC Check Enable` field is in the `Configuration Space Advanced Error Capabilities and Control Register`.

5 The `ECRC Generation Enable` field is in the `Configuration Space Advanced Error Capabilities and Control Register`.

ECRC Forwarding	ECRC Generation Enable [5]	TLP on Application	TLP on Link	Comments
		TD=1, with ECRC	TD=1, with ECRC	
	Yes	TD=0, without ECRC	TD=0, without ECRC	
		TD=1, with ECRC	TD=1, with ECRC	

5 The ECRC Generation Enable field is in the Configuration Space Advanced Error Capabilities and Control Register.

16 Hard IP Reconfiguration

The Arria 10 Hard IP for PCI Express reconfiguration block allows you to dynamically change the value of configuration registers that are *read-only*. You access this block using its Avalon-MM slave interface. You must enable this optional functionality by turning on **Enable Hard IP Reconfiguration** in the parameter editor. For a complete description of the signals in this interface, refer to *Hard IP Reconfiguration Interface*.

The Hard IP reconfiguration block provides access to *read-only* configuration registers, including Configuration Space, Link Configuration, MSI and MSI-X capabilities, Power Management, and Advanced Error Reporting (AER). This interface does not support simulation.

The procedure to dynamically reprogram these registers includes the following three steps:

1. Bring down the PCI Express link by asserting the `hip_reconfig_rst_n` reset signal, if the link is already up. (Reconfiguration can occur before the link has been established.)
2. Reprogram configuration registers using the Avalon-MM slave Hard IP reconfiguration interface.
3. Release the `npor` reset signal.

Note: You can use the LMI interface to change the values of configuration registers that are *read/write* at run time. For more information about the LMI interface, refer to *LMI Signals*.

Contact your Intel representative for descriptions of the read-only, reconfigurable registers.

Related Links

LMI Signals on page 90

17 Testbench and Design Example

This chapter introduces the Root Port or Endpoint design example including a testbench, BFM, and a test driver module. You can create this design example for using design flows described in *Getting Started with the Arria 10 Hard IP for PCI Express* .

When configured as an Endpoint variation, the testbench instantiates a design example and a Root Port BFM, which provides the following functions:

- A configuration routine that sets up all the basic configuration registers in the Endpoint. This configuration allows the Endpoint application to be the target and initiator of PCI Express transactions.
- A Verilog HDL procedure interface to initiate PCI Express transactions to the Endpoint.

The testbench uses a test driver module, **altpcietb_bfm_driver_chaining** to exercise the chaining DMA of the design example. The test driver module displays information from the Endpoint Configuration Space registers, so that you can correlate to the parameters you specified using the parameter editor.

When configured as a Root Port, the testbench instantiates a Root Port design example and an Endpoint model, which provides the following functions:

- A configuration routine that sets up all the basic configuration registers in the Root Port and the Endpoint BFM. This configuration allows the Endpoint application to be the target and initiator of PCI Express transactions.
- A Verilog HDL procedure interface to initiate PCI Express transactions to the Endpoint BFM.

This testbench simulates a single Endpoint or Root Port DUT.

The testbench uses a test driver module, **altpcietb_bfm_driver_rp**, to exercise the target memory and DMA channel in the Endpoint BFM. The test driver module displays information from the Root Port Configuration Space registers, so that you can correlate to the parameters you specified using the parameter editor. The Endpoint model consists of an Endpoint variation combined with the chaining DMA application described above.

Note: The Intel testbench and Root Port or Endpoint BFM provide a simple method to do basic testing of the Application Layer logic that interfaces to the variation. This BFM allows you to create and run simple task stimuli with configurable parameters to exercise basic functionality of the Intel example design. The testbench and Root Port BFM are not intended to be a substitute for a full verification environment. Corner cases and certain traffic profile stimuli are not covered. Refer to the items listed below for further details. To ensure the best verification coverage possible, Intel suggests strongly that you obtain commercially available PCI Express verification IP and tools, or do your own extensive hardware testing or both.

Your Application Layer design may need to handle at least the following scenarios that are not possible to create with the Intel testbench and the Root Port BFM:

- It is unable to generate or receive Vendor Defined Messages. Some systems generate Vendor Defined Messages and the Application Layer must be designed to process them. The Hard IP block passes these messages on to the Application Layer which, in most cases should ignore them.
- It can only handle received read requests that are less than or equal to the currently set **Maximum payload size** option specified under **PCI Express/PCI Capabilities** heading under the **Device** tab using the parameter editor. Many systems are capable of handling larger read requests that are then returned in multiple completions.
- It always returns a single completion for every read request. Some systems split completions on every 64-byte address boundary.
- It always returns completions in the same order the read requests were issued. Some systems generate the completions out-of-order.
- It is unable to generate zero-length read requests that some systems generate as flush requests following some write transactions. The Application Layer must be capable of generating the completions to the zero length read requests.
- It uses fixed credit allocation.
- It does not support parity.
- It does not support multi-function designs which are available when using Configuration Space Bypass mode.
- It does not support Single Root I/O Virtualization (SR-IOV).
- It does not support multiple physical functions and virtual functions available when you select the SR-IOV variant.

Related Links

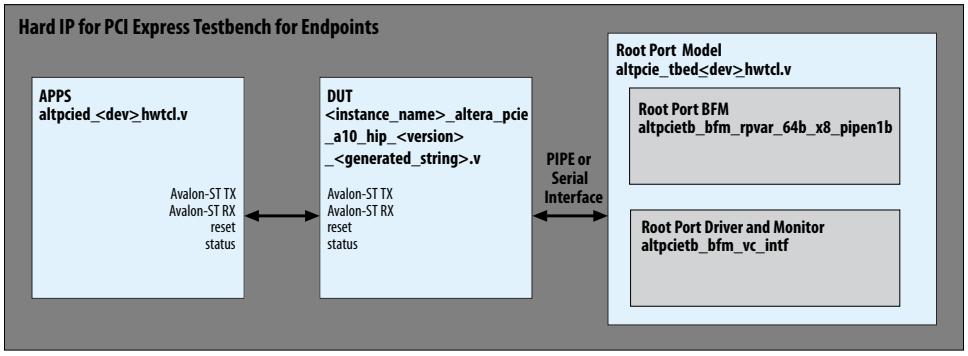
AN-811: Using the Avery BFM for PCI Express Gen3x16 Simulation on Intel Stratix 10 Devices

17.1 Endpoint Testbench

After you install the Quartus Prime software, you can copy any of the example designs from the `<install_dir>/ip/altera/altera_pcie/altera_pcie_a10_ed/example_design/a10` directory.

This testbench simulates up to an ×8 PCI Express link using either the PIPE interfaces of the Root Port and Endpoints or the serial PCI Express interface. The testbench design does not allow more than one PCI Express link to be simulated at a time. The following figure presents a high level view of the design example.

Figure 113. Design Example for Endpoint Designs



The top-level of the testbench instantiates four main modules:

- <qsys_systemname>— This is the example Endpoint design. For more information about this module, refer to *Chaining DMA Design Examples*.
- **altpcietb_bfm_top_rp.v**—This is the Root Port PCI Express BFM. For more information about this module, refer to *Root Port BFM*.
- **altpcietb_pipe_phy**—There are eight instances of this module, one per lane. These modules interconnect the PIPE MAC layer interfaces of the Root Port and the Endpoint. The module mimics the behavior of the PIPE PHY layer to both MAC interfaces.
- **altpcietb_bfm_driver_chaining**—This module drives transactions to the Root Port BFM. This is the module that you modify to vary the transactions sent to the example Endpoint design or your own design. For more information about this module, refer to *Root Port Design Example*.

In addition, the testbench has routines that perform the following tasks:

- Generates the reference clock for the Endpoint at the required frequency.
- Provides a PCI Express reset at start up.

Note: Before running the testbench, you should set the following parameters in `<instantiation_name>_tb/sim/<instantiation_name>_tb.v`:

- `serial_sim_hwtcl`: Set to 1 for serial simulation and 0 for PIPE simulation.
- `enable_pipe32_sim_hwtcl`: Set to 0 for serial simulation and 1 for PIPE simulation.

Related Links

- Quick Start Guide on page 17
- Getting Started with the Arria 10 Hard IP for PCI Express on page 24
  This Gen1 x8 Endpoint design example illustrates a chaining DMA application. It provides instructions to help you quickly customize, simulate, and compile the Arria 10 Hard IP for PCI Express IP Core.

17.2 Root Port Testbench

This testbench simulates up to an ×8 PCI Express link using either the PIPE interfaces of the Root Port and Endpoints or the serial PCI Express interface. The testbench design does not allow more than one PCI Express link to be simulated at a time. The top-level of the testbench instantiates four main modules:

- *<qsys_systemname>*— Name of Root Port This is the example Root Port design. For more information about this module, refer to *Root Port Design Example*.
- **altpcietb_bfm_ep_example_chaining_pipen1b**—This is the Endpoint PCI Express mode described in the section *Chaining DMA Design Examples*.
- **altpcietb_pipe_phy**—There are eight instances of this module, one per lane. These modules connect the PIPE MAC layer interfaces of the Root Port and the Endpoint. The module mimics the behavior of the PIPE PHY layer to both MAC interfaces.
- **altpcietb_bfm_driver_rp**—This module drives transactions to the Root Port BFM. This is the module that you modify to vary the transactions sent to the example Endpoint design or your own design. For more information about this module, see *Test Driver Module*.

The testbench has routines that perform the following tasks:

- Generates the reference clock for the Endpoint at the required frequency.
- Provides a reset at start up.

Note: Before running the testbench, you should set the following parameters:

- `serial_sim_hwtcl`: Set this parameter in ***<instantiation name>*_tb.v** . This parameter controls whether the testbench simulates in PIPE mode or serial mode. When is set to 0, the simulation runs in PIPE mode; when set to 1, it runs in serial mode. Although the `serial_sim_hwtcl` parameter is available in other files, if you set this parameter at the lower level, then it will get overwritten by the **tb.v** level.
- `serial_sim_hwtcl`: Set to 1 for serial simulation and 0 for PIPE simulation.
- `enable_pipe32_sim_hwtcl`: Set to 0 for serial simulation and 1 for PIPE simulation.

17.3 Chaining DMA Design Examples

This design example shows how to create a chaining DMA native Endpoint which supports simultaneous DMA read and write transactions. The write DMA module implements write operations from the Endpoint memory to the root complex (RC) memory. The read DMA implements read operations from the RC memory to the Endpoint memory.

When operating on a hardware platform, the DMA is typically controlled by a software application running on the root complex processor. In simulation, the generated testbench, along with this design example, provides a BFM driver module in Verilog HDL that controls the DMA operations. Because the example relies on no other hardware interface than the PCI Express link, you can use the design example for the initial hardware validation of your system.

The design example includes the following two main components:

- The Root Port variation
- An Application Layer design example

The DUT variant is generated in the language (Verilog HDL or VHDL) that you selected for the variation file. The testbench files are only generated in Verilog HDL in the current release. If you choose to use VHDL for your variant, you must have a mixed-language simulator to run this testbench.

Note: The chaining DMA design example requires setting BAR 2 or BAR 3 to a minimum of 256 bytes. To run the DMA tests using MSI, you must set the **Number of MSI messages requested** parameter under the **PCI Express/PCI Capabilities** page to at least 2.

The chaining DMA design example uses an architecture capable of transferring a large amount of fragmented memory without accessing the DMA registers for every memory block. For each block of memory to be transferred, the chaining DMA design example uses a descriptor table containing the following information:

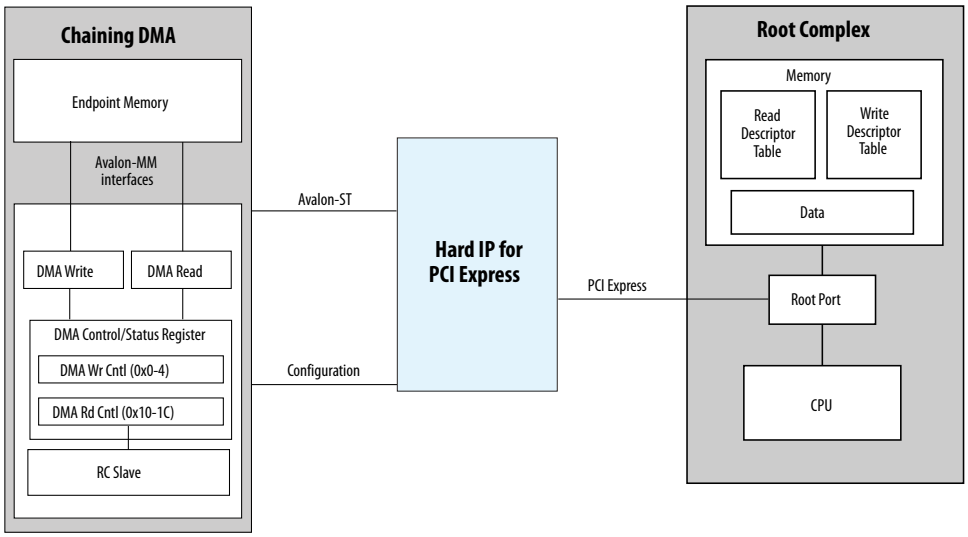
- Length of the transfer
- Address of the source
- Address of the destination
- Control bits to set the handshaking behavior between the software application or BFM driver and the chaining DMA module

Note: The chaining DMA design example only supports dword-aligned accesses. The chaining DMA design example does not support ECRC forwarding.

The BFM driver writes the descriptor tables into BFM shared memory, from which the chaining DMA design engine continuously collects the descriptor tables for DMA read, DMA write, or both. At the beginning of the transfer, the BFM programs the Endpoint chaining DMA control register. The chaining DMA control register indicates the total number of descriptor tables and the BFM shared memory address of the first descriptor table. After programming the chaining DMA control register, the chaining DMA engine continuously fetches descriptors from the BFM shared memory for both DMA reads and DMA writes, and then performs the data transfer for each descriptor.

The following figure shows a block diagram of the design example connected to an external RC CPU. For a description of the DMA write and read registers, *Chaining DMA Control and Status Registers*.

Figure 114. Top-Level Chaining DMA Example for Simulation



The block diagram contains the following elements:

- Endpoint DMA write and read requester modules.
- The chaining DMA design example connects to the Avalon-ST interface of the Arria 10 Hard IP for PCI Express. The connections consist of the following interfaces:
  - The Avalon-ST RX receives TLP header and data information from the Hard IP block
  - The Avalon-ST TX transmits TLP header and data information to the Hard IP block
  - The Avalon-ST MSI port requests MSI interrupts from the Hard IP block
  - The sideband signal bus carries static information such as configuration information
- The descriptor tables of the DMA read and the DMA write are located in the BFM shared memory.
- A RC CPU and associated PCI Express PHY link to the Endpoint design example, using a Root Port and a north/south bridge.

The example Endpoint design Application Layer accomplishes the following objectives:

- Shows you how to interface to the Arria 10 Hard IP for PCI Express using the Avalon-ST protocol.
- Provides a chaining DMA channel that initiates memory read and write transactions on the PCI Express link.
- If the ECRC forwarding functionality is enabled, provides a CRC Compiler IP core to check the ECRC dword from the Avalon-ST RX path and to generate the ECRC for the Avalon-ST TX path.

The following modules are included in the design example and located in the subdirectory <*qsys_systemname*>/testbench/<*qsys_system_name*>_tb/simulation/submodules :

- <*qsys_systemname*> —This module is the top level of the example Endpoint design that you use for simulation.

  This module provides both PIPE and serial interfaces for the simulation environment. This module has a `test_in` debug ports. Refer to *Test Signals* which allow you to monitor and control internal states of the Hard IP.

  For synthesis, the top level module is <*qsys_systemname*>/synthesis/submodules. This module instantiates the top-level module and propagates only a small sub-set of the test ports to the external I/Os. These test ports can be used in your design.
- <*variation name*>**.v** or <*variation name*>**.vhd**— Because Intel provides many sample parameterizations, you may have to edit one of the provided examples to create a simulation that matches your requirements. <*variation name*>**.v** or <*variation name*>**.vhd**— Because Intel provides many sample parameterizations, you may have to edit one of the provided examples to create a simulation that matches your requirements.

The chaining DMA design example hierarchy consists of these components:

- A DMA read and a DMA write module
- An on-chip Endpoint memory (Avalon-MM slave) which uses two Avalon-MM interfaces for each engine

The RC slave module is used primarily for downstream transactions which target the Endpoint on-chip buffer memory. These target memory transactions bypass the DMA engines. In addition, the RC slave module monitors performance and acknowledges incoming message TLPs. Each DMA module consists of these components:

- Control register module—The RC programs the control register (four dwords) to start the DMA.
- Descriptor module—The DMA engine fetches four dword descriptors from BFM shared memory which hosts the chaining DMA descriptor table.
- Requester module—For a given descriptor, the DMA engine performs the memory transfer between Endpoint memory and the BFM shared memory.

The following modules are provided in both Verilog HDL:

- **altpcierd_example_app_chaining**—This top level module contains the logic related to the Avalon-ST interfaces as well as the logic related to the sideband bus. This module is fully register bounded and can be used as an incremental re-compile partition in the Quartus Prime compilation flow.
- **altpcierd_cdma_ast_rx**, **altpcierd_cdma_ast_rx_64**, **altpcierd_cdma_ast_rx_128**—These modules implement the Avalon-ST receive port for the chaining DMA. The Avalon-ST receive port converts the Avalon-ST interface of the IP core to the descriptor/data interface used by the chaining DMA submodules. **altpcierd_cdma_ast_rx** is used with the descriptor/data IP core (through the ICM). **a ltpcierd_cdma_ast_rx_64** is used with the 64-bit Avalon-ST IP core. **altpcierd_cdma_ast_rx_128** is used with the 128-bit Avalon-ST IP core.
- **altpcierd_cdma_ast_tx**, **altpcierd_cdma_ast_tx_64**, **altpcierd_cdma_ast_tx_128**—These modules implement the Avalon-ST transmit port for the chaining DMA. The Avalon-ST transmit port converts the descriptor/data interface of the chaining DMA submodules to the Avalon-ST interface of the IP core. **altpcierd_cdma_ast_tx** is used with the descriptor/data IP core (through the ICM). **altpcierd_cdma_ast_tx_64** is used with the 64-bit Avalon-ST IP core. **altpcierd_cdma_ast_tx_128** is used with the 128-bit Avalon-ST IP core.
- **altpcierd_cdma_ast_msi**—This module converts MSI requests from the chaining DMA submodules into Avalon-ST streaming data.
- **alpcierd_cdma_app_icm**—This module arbitrates PCI Express packets for the modules **altpcierd_dma_dt** (read or write) and **altpcierd_rc_slave**. **alpcierd_cdma_app_icm** instantiates the Endpoint memory used for the DMA read and write transfer.
- **alt pcierd_compliance_test.v**—This module provides the logic to perform CBB via a push button.
- **altpcierd_rc_slave**—This module provides the completer function for all downstream accesses. It instantiates the **altpcierd_rxtx_downstream_intf** and **altpcierd_reg_ access** modules. Downstream requests include programming of chaining DMA control registers, reading of DMA status registers, and direct read and write access to the Endpoint target memory, bypassing the DMA.
- **altpcierd_rx_tx_downstream_intf**—This module processes all downstream read and write requests and handles transmission of completions. Requests addressed to BARs 0, 1, 4, and 5 access the chaining DMA target memory space. Requests addressed to BARs 2 and 3 access the chaining DMA control and status register space using the **altpcierd_reg_access** module.
- **altpcierd_reg_access**—This module provides access to all of the chaining DMA control and status registers (BAR 2 and 3 address space). It provides address decoding for all requests and multiplexing for completion data. All registers are 32-bits wide. Control and status registers include the control registers in the **altpcierd_dma_prg_reg** module, status registers in the **altpcierd_read_dma_requester** and **altpcierd_write_dma_requester** modules, as well as other miscellaneous status registers.
- **altpcierd_dma_dt**—This module arbitrates PCI Express packets issued by the submodules **altpcierd_dma_prg_reg**, **altpcierd_read_dma_requester**, **altpcierd_write_dma_requester** and **altpcierd_dma_descriptor**.

- **altpcierd_dma_prg_reg** —This module contains the chaining DMA control registers which get programmed by the software application or BFM driver.
- **altpcierd_dma_descriptor**—This module retrieves the DMA read or write descriptor from the BFM shared memory, and stores it in a descriptor FIFO. This module issues upstream PCI Express TLPs of type MRd.
- **altpcierd_read_dma_requester**, **altpcierd_read_dma_requester_128**—For each descriptor located in the **altpcierd_descriptor FIFO**, this module transfers data from the BFM shared memory to the Endpoint memory by issuing MRd PCI Express transaction layer packets. **altpcierd_read_dma_requester** is used with the 64-bit Avalon-ST IP core. **altpcierd_read_dma_requester_128** is used with the 128-bit Avalon-ST IP core.
- **altpcie rd_write_dma_requester, altpcierd_write_dma_requester_128**—For each descriptor located in the **altpcierd_descriptor FIFO**, this module transfers data from the Endpoint memory to the BFM shared memory by issuing MWr PCI Express transaction layer packets. **altpcierd_write_dma_requester** is used with the 64-bit Avalon-ST IP core. **altpcierd_write_dma_requester_128** is used with the 128-bit Avalon-ST IP core.ls
- **altpcierd_cpld_rx_buffer**—This modules monitors the available space of the RX Buffer; It prevents RX Buffer overflow by arbitrating memory read request issued by the application.
- **altpcierd_cplerr_lmi**—This module transfers the err_desc_func0 from the application to the Hard IP block using the LMI interface. It also retimes the `cpl_err` bits from the application to the Hard IP block.
- **altpcierd_tl_cfg_sample**—This module demultiplexes the Configuration Space signals from the `tl_cfg_ctl` bus from the Hard IP block and synchronizes this information, along with the `tl_cfg_sts` bus to the user clock (`pld_clk`) domain.

Related Links

- Test Signals on page 105
- Chaining DMA Control and Status Registers on page 183

17.3.1 BAR/Address Map

The design example maps received memory transactions to either the target memory block or the control register block based on which BAR the transaction matches. There are multiple BARs that map to each of these blocks to maximize interoperability with different variation files. The following table shows the mapping.

Table 85. BAR Map

Memory BAR	Mapping
32-bit BAR0 32-bit BAR1 64-bit BAR1:0	Maps to 32 KB target memory block. Use the rc_slave module to bypass the chaining DMA.
32-bit BAR2 32-bit BAR3 64-bit BAR3:2	Maps to DMA Read and DMA write control and status registers, a minimum of 256 bytes.
32-bit BAR4	Maps to 32 KB target memory block. Use the rc_slave module to bypass the chaining DMA.

continued...

Memory BAR	Mapping
32-bit BAR5 64-bit BAR5:4	
Expansion ROM BAR	Not implemented by design example; behavior is unpredictable.
I/O Space BAR (any)	Not implemented by design example; behavior is unpredictable.

17.3.2 Chaining DMA Control and Status Registers

The software application programs the chaining DMA control register located in the Endpoint application. The following table describes the control registers which consists of four dwords for the DMA write and four dwords for the DMA read. The DMA control registers are read/write.

In this table, Addr specifies the Endpoint byte address offset from BAR2 or BAR3.

Table 86. Chaining DMA Control Register Definitions

Addr	Register Name	Bits[31:]24	Bit[23:16]	Bit[15:0]
0x0	DMA Wr Cntl DW0	Control Field		Number of descriptors in descriptor table
0x4	DMA Wr Cntl DW1	Base Address of the Write Descriptor Table (BDT) in the RC Memory–Upper DWORD		
0x8	DMA Wr Cntl DW2	Base Address of the Write Descriptor Table (BDT) in the RC Memory–Lower DWORD		
0xC	DMA Wr Cntl DW3	Reserved	Reserved	RCLAST–Idx of last descriptor to process
0x10	DMA Rd Cntl DW0	Control Field (described in the next table)		Number of descriptors in descriptor table
0x14	DMA Rd Cntl DW1	Base Address of the Read Descriptor Table (BDT) in the RC Memory–Upper DWORD		
0x18	DMA Rd Cntl DW2	Base Address of the Read Descriptor Table (BDT) in the RC Memory–Lower DWORD		
0x1C	DMA Rd Cntl DW3	Reserved	Reserved	RCLAST–Idx of the last descriptor to process

The following table describes the control fields of the of the DMA read and DMA write control registers.

Table 87. Bit Definitions for the Control Field in the DMA Write Control Register and DMA Read Control Register

Bit	Field	Description
16	Reserved	—
17	MSI_ENA	Enables interrupts of all descriptors. When 1, the Endpoint DMA module issues an interrupt using MSI to the RC when each descriptor is completed. Your software application or BFM driver can use this interrupt to monitor the DMA transfer status.
18	EPLAST_ENA	Enables the Endpoint DMA module to write the number of each descriptor back to the EPLAST field in the descriptor table.

continued...

Bit	Field	Description
[24:20]	MSI Number	When your RC reads the MSI capabilities of the Endpoint, these register bits map to the back-end MSI signals app_msi_num [4:0]. If there is more than one MSI, the default mapping if all the MSIs are available, is: • MSI 0 = Read • MSI 1 = Write
[30:28]	MSI Traffic Class	When the RC application software reads the MSI capabilities of the Endpoint, this value is assigned by default to MSI traffic class 0. These register bits map to the back-end signal app_msi_tc[2:0].
31	DT RC Last Sync	When 0, the DMA engine stops transfers when the last descriptor has been executed. When 1, the DMA engine loops infinitely restarting with the first descriptor when the last descriptor is completed. To stop the infinite loop, set this bit to 0.

The following table defines the DMA status registers. These registers are read only. In this table, Addr specifies the Endpoint byte address offset from BAR2 or BAR3.

Table 88. Chaining DMA Status Register Definitions

Addr	Register Name	Bits[31:24]	Bits[23:16]	Bits[15:0]
0x20	DMA Wr Status Hi	For field definitions refer to *Fields in the DMA Write Status High Register* below.		
0x24	DMA Wr Status Lo	Target Mem Address Width	Write DMA Performance Counter. (Clock cycles from time DMA header programmed until last descriptor completes, including time to fetch descriptors.)	
0x28	DMA Rd Status Hi	For field definitions refer to *Fields in the DMA Read Status High Register* below.		
0x2C	DMA Rd Status Lo	Max No. of Tags	Read DMA Performance Counter. The number of clocks from the time the DMA header is programmed until the last descriptor completes, including the time to fetch descriptors.	
0x30	Error Status	Reserved		Error Counter. Number of bad ECRCs detected by the Application Layer. Valid only when ECRC forwarding is enabled.

The following table describes the fields of the DMA write status register. All of these fields are read only.

Table 89. Fields in the DMA Write Status High Register

Bit	Field	Description
[31:28]	CDMA version	Identifies the version of the chaining DMA example design.
[27:24]	Reserved	—
[23:21]	Max payload size	The following encodings are defined: • 001 128 bytes • 001 256 bytes • 010 512 bytes • 011 1024 bytes • 100 2048 bytes

continued...

Bit	Field	Description
[20:17]	Reserved	—
16	`Write DMA descriptor FIFO empty`	Indicates that there are no more descriptors pending in the write DMA.
[15:0]	`Write DMA EPLAS`	Indicates the number of the last descriptor completed by the write DMA. For simultaneous DMA read and write transfers, EPLAST is only supported for the final descriptor in the descriptor table.

The following table describes the fields in the DMA read status high register. All of these fields are read only.

Table 90. Fields in the DMA Read Status High Register

Bit	Field	Description
[31:24]	Reserved	—
[23:21]	Max Read Request Size	The following encodings are defined: • 001 128 bytes • 001 256 bytes • 010 512 bytes • 011 1024 bytes • 100 2048 bytes
[20:17]	Negotiated Link Width	The following encodings are defined: • 4'b0001 ×1 • 4'b0010 ×2 • 4'b0100 ×4 • 4'b1000 ×8
16	Read DMA Descriptor FIFO Empty	Indicates that there are no more descriptors pending in the read DMA.
[15:0]	Read DMA EPLAST	Indicates the number of the last descriptor completed by the read DMA. For simultaneous DMA read and write transfers, EPLAST is only supported for the final descriptor in the descriptor table.

17.3.3 Chaining DMA Descriptor Tables

The following table describes the Chaining DMA descriptor table. This table is stored in the BFM shared memory. It consists of a four-dword descriptor header and a contiguous list of <*n*> four-dword descriptors. The Endpoint chaining DMA application accesses the Chaining DMA descriptor table for two reasons:

- To iteratively retrieve four-dword descriptors to start a DMA
- To send update status to the RP, for example to record the number of descriptors completed to the descriptor header

Each subsequent descriptor consists of a minimum of four dwords of data and corresponds to one DMA transfer. (A dword equals 32 bits.)

Note: The chaining DMA descriptor table should not cross a 4 KB boundary.

Table 91. Chaining DMA Descriptor Table

Byte Address Offset to Base Source	Descriptor Type	Description
0x0	Descriptor Header	Reserved
0x4		Reserved
0x8		Reserved
0xC		EPLAST - when enabled by the EPLAST_ENA bit in the control register or descriptor, this location records the number of the last descriptor completed by the chaining DMA module.
0x10	Descriptor 0	Control fields, DMA length
0x14		Endpoint address
0x18		RC address upper dword
0x1C		RC address lower dword
0x20	Descriptor 1	Control fields, DMA length
0x24		Endpoint address
0x28		RC address upper dword
0x2C		RC address lower dword
. . .		
0x ..0	Descriptor <*n*>	Control fields, DMA length
0x ..4		Endpoint address
0x ..8		RC address upper dword
0x ..C		RC address lower dword

The following table shows the layout of the descriptor fields following the descriptor header.

Table 92. Chaining DMA Descriptor Format Map

Bits[31:22]	Bits[21:16]	Bits[15:0]
Reserved	Control Fields **(**refer to Table 18–9**)**	DMA Length
Endpoint Address		
RC Address Upper DWORD		
RC Address Lower DWORD		

The following table shows the layout of the control fields of the chaining DMA descriptor.

Table 93. Chaining DMA Descriptor Format Map (Control Fields)

Bits[21:18]	Bit[17]	Bit[16]
Reserved	EPLAST_ENA	MSI

Each descriptor provides the hardware information on one DMA transfer. The following table describes each descriptor field.

Table 94. Chaining DMA Descriptor Fields

Descriptor Field	Endpoint Access	RC Access	Description
Endpoint Address	R	R/W	A 32-bit field that specifies the base address of the memory transfer on the Endpoint site.
RC Address Upper DWORD	R	R/W	Specifies the upper base address of the memory transfer on the RC site.
RC Address Lower DWORD	R	R/W	Specifies the lower base address of the memory transfer on the RC site.
DMA Length	R	R/W	Specifies the number of DMA DWORDs to transfer.
EPLAST_ENA	R	R/W	This bit is OR'd with the EPLAST_ENA bit of the control register. When EPLAST_ENA is set, the Endpoint DMA module updates the EPLAST field of the descriptor table with the number of the last completed descriptor, in the form *<0 – n>*. Refer to Chaining DMA Descriptor Tables on page 185 for more information.
MSI_ENA	R	R/W	This bit is OR'd with the MSI bit of the descriptor header. When this bit is set the Endpoint DMA module sends an interrupt when the descriptor is completed.

17.4 Test Driver Module

The BFM driver module, **altpcietb_bfm_driver_chaining.v** is configured to test the chaining DMA example Endpoint design. The BFM driver module configures the Endpoint Configuration Space registers and then tests the example Endpoint chaining DMA channel. This file is stored in the *<working_dir>/testbench/<variation_name>/simulation/submodules* directory.

The BFM test driver module performs the following steps in sequence:

1. Configures the Root Port and Endpoint Configuration Spaces, which the BFM test driver module does by calling the procedure `ebfm_cfg_rp_ep`, which is part of **altpcietb_bfm_configure**.
2. Finds a suitable BAR to access the example Endpoint design Control Register space. Either BARs 2 or 3 must be at least a 256-byte memory BAR to perform the DMA channel test. The `find_mem_bar` procedure in the **altpcietb_bfm_driver_chaining** does this.
3. If a suitable BAR is found in the previous step, the driver performs the following tasks:
   a. DMA read—The driver programs the chaining DMA to read data from the BFM shared memory into the Endpoint memory. The descriptor control fields are specified so that the chaining DMA completes the following steps to indicate transfer completion:
      - The chaining DMA writes the EPLast bit of the *Chaining DMA Descriptor Table* after finishing the data transfer for the first and last descriptors.
      - The chaining DMA issues an MSI when the last descriptor has completed.

   a. DMA write—The driver programs the chaining DMA to write the data from its Endpoint memory back to the BFM shared memory. The descriptor control fields are specified so that the chaining DMA completes the following steps to indicate transfer completion:

- The chaining DMA writes the EPLast bit of the *Chaining DMA Descriptor Table* after completing the data transfer for the first and last descriptors.
- The chaining DMA issues an MSI when the last descriptor has completed.
- The data written back to BFM is checked against the data that was read from the BFM.
- The driver programs the chaining DMA to perform a test that demonstrates downstream access of the chaining DMA Endpoint memory.

Note: Edit this file if you want to add your own custom PCIe transactions. Insert your own custom function after the `find_mem_bar` function. You can use the functions in the *BFM Procedures and Functions* section.

Related Links

- Chaining DMA Descriptor Tables on page 185
- BFM Procedures and Functions on page 202

17.5 DMA Write Cycles

The procedure `dma_wr_test` used for DMA writes uses the following steps:

1. Configures the BFM shared memory. Configuration is accomplished with three descriptor tables described below.

Table 95. Write Descriptor 0

	Offset in BFM in Shared Memory	Value	Description
DW0	0x810	82	Transfer length in dwords and control bits as described in *Bit Definitions for the Control Field in the DMA Write Control Register and DMA Read Control Register*.
DW1	0x814	3	Endpoint address
DW2	0x818	0	BFM shared memory data buffer 0 upper address value
DW3	0x81c	0x1800	BFM shared memory data buffer 1 lower address value
Data Buffer 0	0x1800	Increment by 1 from 0x1515_0001	Data content in the BFM shared memory from address: 0x01800–0x1840

Table 96. Write Descriptor 1

	Offset in BFM Shared Memory	Value	Description
DW0	0x820	1,024	Transfer length in dwords and control bits as described in *Bit Definitions for the Control Field in the DMA Write Control Register and DMA Read Control Register* .
DW1	0x824	0	Endpoint address

continued...

	Offset in BFM Shared Memory	Value	Description
DW2	0x828	0	BFM shared memory data buffer 1 upper address value
DW3	0x82c	0x2800	BFM shared memory data buffer 1 lower address value
Data Buffer 1	0x02800	Increment by 1 from 0x2525_0001	Data content in the BFM shared memory from address: 0x02800

Table 97. Write Descriptor 2

	Offset in BFM Shared Memory	Value	Description
DW0	0x830	644	Transfer length in dwords and control bits as described in *Bit Definitions for the Control Field in the DMA Write Control Register and DMA Read Control Register*.
DW1	0x834	0	Endpoint address
DW2	0x838	0	BFM shared memory data buffer 2 upper address value
DW3	0x83c	0x057A0	BFM shared memory data buffer 2 lower address value
Data Buffer 2	0x057A0	Increment by 1 from 0x3535_0001	Data content in the BFM shared memory from address: 0x057A0

2. Sets up the chaining DMA descriptor header and starts the transfer data from the Endpoint memory to the BFM shared memory. The transfer calls the procedure `dma_set_header` which writes four dwords, DW0:DW3, into the DMA write register module.

Table 98. DMA Control Register Setup for DMA Write

	Offset in DMA Control Register (BAR2)	Value	Description
DW0	0x0	3	Number of descriptors and control bits as described in *Chaining DMA Control Register Definitions*.
DW1	0x4	0	BFM shared memory descriptor table upper address value
DW2	0x8	0x800	BFM shared memory descriptor table lower address value
DW3	0xc	2	Last valid descriptor

After writing the last dword, DW3, of the descriptor header, the DMA write starts the three subsequent data transfers.

3. Waits for the DMA write completion by polling the BFM share memory location 0x80c, where the DMA write engine is updating the value of the number of completed descriptor. Calls the procedures `rcmem_poll` and `msi_poll` to determine when the DMA write transfers have completed.

Related Links

Chaining DMA Control and Status Registers on page 183

17.6 DMA Read Cycles

The procedure `dma_rd_test` used for DMA read uses the following three steps:

1. Configures the BFM shared memory with a call to the procedure `dma_set_rd_desc_data` which sets the following three descriptor tables. .

Table 99. Read Descriptor 0

	Offset in BFM Shared Memory	Value	Description
DW0	0x910	82	Transfer length in dwords and control bits as described in on page 18–15
DW1	0x914	3	Endpoint address value
DW2	0x918	0	BFM shared memory data buffer 0 upper address value
DW3	0x91c	0x8DF0	BFM shared memory data buffer 0 lower address value
Data Buffer 0	0x8DF0	Increment by 1 from 0xAAA0_0001	Data content in the BFM shared memory from address: 0x89F0

Table 100. Read Descriptor 1

	Offset in BFM Shared Memory	Value	Description
DW0	0x920	1,024	Transfer length in dwords and control bits as described in on page 18–15
DW1	0x924	0	Endpoint address value
DW2	0x928	10	BFM shared memory data buffer 1 upper address value
DW3	0x92c	0x10900	BFM shared memory data buffer 1 lower address value
Data Buffer 1	0x10900	Increment by 1 from 0xBBBB_0001	Data content in the BFM shared memory from address: 0x10900

Table 101. Read Descriptor 2

	Offset in BFM Shared Memory	Value	Description
DW0	0x930	644	Transfer length in dwords and control bits as described in on page 18–15
DW1	0x934	0	Endpoint address value
DW2	0x938	0	BFM shared memory upper address value
DW3	0x93c	0x20EF0	BFM shared memory lower address value
Data Buffer 2	0x20EF0	Increment by 1 from 0xCCCC_0001	Data content in the BFM shared memory from address: 0x20EF0

2. Sets up the chaining DMA descriptor header and starts the transfer data from the BFM shared memory to the Endpoint memory by calling the procedure `dma_set_header` which writes four dwords, DW0:DW3 into the DMA read register module.

Table 102. DMA Control Register Setup for DMA Read

	Offset in DMA Control Registers (BAR2)	Value	Description
DW0	0x0	3	Number of descriptors and control bits as described in *Chaining DMA Control Register Definitions*.
DW1	0x14	0	BFM shared memory upper address value
DW2	0x18	0x900	BFM shared memory lower address value
DW3	0x1c	2	Last descriptor written

After writing the last dword of the Descriptor header (DW3), the DMA read starts the three subsequent data transfers.

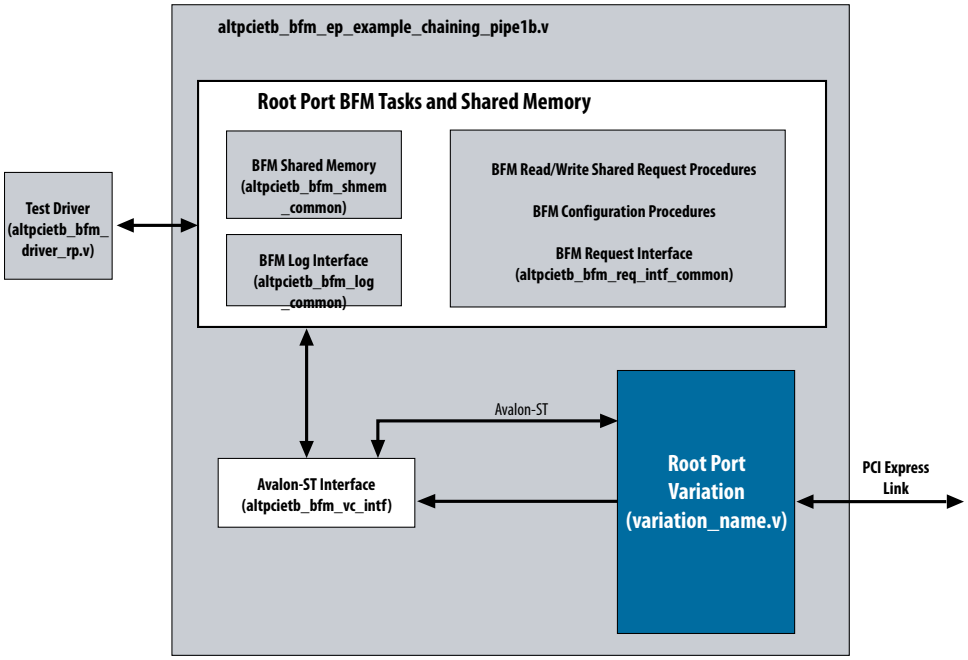
3. Waits for the DMA read completion by polling the BFM shared memory location 0x90c, where the DMA read engine is updating the value of the number of completed descriptors. Calls the procedures `rcmem_poll` and `msi_poll` to determine when the DMA read transfers have completed.

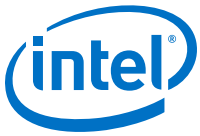
17.7 Root Port Design Example

The design example includes the following primary components:

- Root Port variation (<*qsys_systemname*>.
- Avalon-ST Interfaces (**altpcietb_bfm_vc_intf_ast**)—handles the transfer of TLP requests and completions to and from the Arria 10 Hard IP for PCI Express variation using the Avalon-ST interface.
- Root Port BFM tasks—contains the high-level tasks called by the test driver, low-level tasks that request PCI Express transfers from **altpcietb_bfm_vc_intf_ast**, the Root Port memory space, and simulation functions such as displaying messages and stopping simulation.
- Test Driver (**altpcietb_bfm_driver_rp.v**)—the chaining DMA Endpoint test driver which configures the Root Port and Endpoint for DMA transfer and checks for the successful transfer of data. Refer to the *Test Driver Module*for a detailed description.

Figure 115. Root Port Design Example





You can use the example Root Port design for Verilog HDL simulation. All of the modules necessary to implement the example design with the variation file are contained in **altpcietb_bfm_ep_example_chaining_pipen1b.v**.

The top-level of the testbench instantiates the following key files:

- **altlpcietb_bfm_top_ep.v**— this is the Endpoint BFM. This file also instantiates the SERDES and PIPE interface.
- **altpcietb_pipe_phy.v**—used to simulate the PIPE interface.
- **altp cietb_bfm_ep_example_chaining_pipen1b.v**—the top-level of the Root Port design example that you use for simulation. This module instantiates the Root Port variation, <*variation_name*> **.v**, and the Root Port application **altpcietb_bfm_vc_intf** _<*application_width*> . This module provides both PIPE and serial interfaces for the simulation environment. This module has two debug ports named `test_out_icm_`(which is the `test_out` signal from the Hard IP) and `test_in` which allows you to monitor and control internal states of the Hard IP variation.
- **altpcietb_bfm_vc_intf_ast.v**—a wrapper module which instantiates either **altpcietb_vc_intf_64** or **altpcietb_vc_intf_** <*application_width*> based on the type of Avalon-ST interface that is generated.
- **altpcietb_vc_intf_** _<*application_width*> **.v**—provide the interface between the Arria 10 Hard IP for PCI Express variant and the Root Port BFM tasks. They provide the same function as the **altpcietb_bfm_vc_intf.v** module, transmitting requests and handling completions. Refer to the *Root Port BFM* for a full description of this function. This version uses Avalon-ST signaling with either a 64- or 128-bit data bus interface.
- **altpcierd_tl_cfg_sample.v**—accesses Configuration Space signals from the variant. Refer to the *Chaining DMA Design Examples* for a description of this module.

Files in subdirectory <*qsys_systemname*> **/testbench/simulation/submodules**:

- **altpcietb_bfm_ep_example_chaining_pipen1b.v**—the simulation model for the chaining DMA Endpoint.
- **altpcietb_bfm_driver_rp.v**–this file contains the functions to implement the shared memory space, PCI Express reads and writes, initialize the Configuration Space registers, log and display simulation messages, and define global constants.

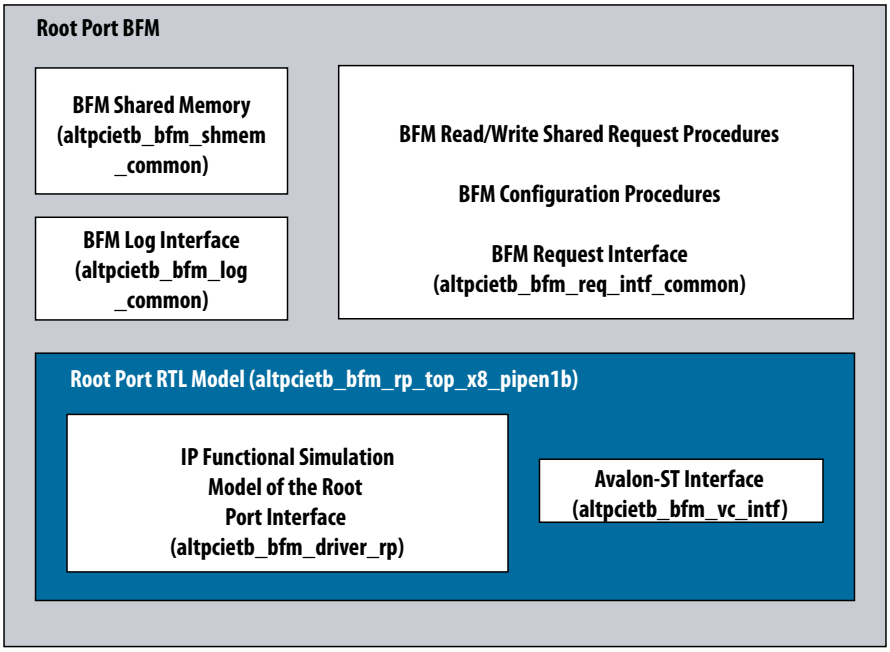
Related Links

- Test Driver Module on page 187
- Chaining DMA Design Examples on page 177

17.8 Root Port BFM

The basic Root Port BFM provides Verilog HDL task-based interface for requesting transactions that are issued to the PCI Express link. The Root Port BFM also handles requests received from the PCI Express link. The following figure provides an overview of the Root Port BFM.

Figure 116. Root Port BFM



The functionality of each of the modules included is explained below.

- BFM shared memory (`altpcietb_bfm_shmem_common` Verilog HDL include file)—The Root Port BFM is based on the BFM memory that is used for the following purposes:
- Storing data received with all completions from the PCI Express link.
- Storing data received with all write transactions received from the PCI Express link.
- Sourcing data for all completions in response to read transactions received from the PCI Express link.
- Sourcing data for most write transactions issued to the PCI Express link. The only exception is certain BFM write procedures that have a four-byte field of write data passed in the call.
- Storing a data structure that contains the sizes of and the values programmed in the BARs of the Endpoint.

A set of procedures is provided to read, write, fill, and check the shared memory from the BFM driver. For details on these procedures, see *BFM Shared Memory Access Procedures*.

- BFM Read/Write Request Functions(`altpcietb_bfm_driver_rp.v`)—These functions provide the basic BFM calls for PCI Express read and write requests. For details on these procedures, refer to *BFM Read and Write Procedures*.
- BFM Configuration Functions(`altpcietb_bfm_driver_rp.v`)—These functions provide the BFM calls to request configuration of the PCI Express link and the Endpoint Configuration Space registers. For details on these procedures and functions, refer to *BFM Configuration Procedures*.
- BFM Log Interface(`altpcietb_bfm_driver_rp.v`)—The BFM log functions provides routines for writing commonly formatted messages to the simulator standard output and optionally to a log file. It also provides controls that stop simulation on errors. For details on these procedures, refer to *BFM Log and Message Procedures*.
- BFM Request Interface(`altpcietb_bfm_driver_rp.v`)—This interface provides the low-level interface between the `altpcietb_bfm_rdwr` and `altpcietb_bfm_configure` procedures or functions and the Root Port RTL Model. This interface stores a write-protected data structure containing the sizes and the values programmed in the BAR registers of the Endpoint, as well as, other critical data used for internal BFM management. You do not need to access these files directly to adapt the testbench to test your Endpoint application.
- Avalon-ST Interfaces (`altpcietb_bfm_vc_intf.v`)—These interface modules handle the Root Port interface model. They take requests from the BFM request interface and generate the required PCI Express transactions. They handle completions received from the PCI Express link and notify the BFM request interface when requests are complete. Additionally, they handle any requests received from the PCI Express link, and store or fetch data from the shared memory before generating the required completions.

Related Links

- Test Signals on page 105
- BFM Shared Memory Access Procedures on page 208

17.8.1 BFM Memory Map

The BFM shared memory is configured to be two MBs. The BFM shared memory is mapped into the first two MBs of I/O space and also the first two MBs of memory space. When the Endpoint application generates an I/O or memory transaction in this range, the BFM reads or writes the shared memory.

17.8.2 Configuration Space Bus and Device Numbering

The Root Port interface is assigned to be device number 0 on internal bus number 0. The Endpoint can be assigned to be any device number on any bus number (greater than 0) through the call to procedure `ebfm_cfg_rp_ep`. The specified bus number is assigned to be the secondary bus in the Root Port Configuration Space.

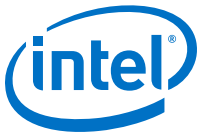
17.8.3 Configuration of Root Port and Endpoint

Before you issue transactions to the Endpoint, you must configure the Root Port and Endpoint Configuration Space registers. To configure these registers, call the procedure `ebfm_cfg_rp_ep`, which is included in **altpcietb_bfm_driver_rp.v**.

The `ebfm_cfg_rp_ep` executes the following steps to initialize the Configuration Space:

1. Sets the Root Port Configuration Space to enable the Root Port to send transactions on the PCI Express link.
2. Sets the Root Port and Endpoint PCI Express Capability Device Control registers as follows:
   a. Disables `Error Reporting` in both the Root Port and Endpoint. BFM does not have error handling capability.
   b. Enables `Relaxed Ordering` in both Root Port and Endpoint.
   c. Enables `Extended Tags` for the Endpoint, if the Endpoint has that capability.
   d. Disables `Phantom Functions`, `Aux Power PM`, and `No Snoop` in both the Root Port and Endpoint.
   e. Sets the `Max Payload Size` to what the Endpoint supports because the Root Port supports the maximum payload size.
   f. Sets the Root Port `Max Read Request Size` to 4 KB because the example Endpoint design supports breaking the read into as many completions as necessary.
   g. Sets the Endpoint `Max Read Request Size` equal to the `Max Payload Size` because the Root Port does not support breaking the read request into multiple completions.
3. Assigns values to all the Endpoint BAR registers. The BAR addresses are assigned by the algorithm outlined below.
   a. I/O BARs are assigned smallest to largest starting just above the ending address of BFM shared memory in I/O space and continuing as needed throughout a full 32-bit I/O space.
   b. The 32-bit non-prefetchable memory BARs are assigned smallest to largest, starting just above the ending address of BFM shared memory in memory space and continuing as needed throughout a full 32-bit memory space.
   c. Assignment of the 32-bit prefetchable and 64-bit prefetchable memory BARS are based on the value of the `addr_map_4GB_limit` input to the `ebfm_cfg_rp_ep`. The default value of the `addr_map_4GB_limit` is `0`.

      If the `addr_map_4GB_limit` input to the `ebfm_cfg_rp_ep` is set to `0`, then the 32-bit prefetchable memory BARs are assigned largest to smallest, starting at the top of 32-bit memory space and continuing as needed down to the ending address of the last 32-bit non-prefetchable BAR.



However, if the addr_map_4GB_limit input is set to 1, the address map is limited to 4 GB, the 32-bit and 64-bit prefetchable memory BARs are assigned largest to smallest, starting at the top of the 32-bit memory space and continuing as needed down to the ending address of the last 32-bit non-prefetchable BAR.

d. If the addr_map_4GB_limit input to the ebfm_cfg_rp_ep is set to 0, then the 64-bit prefetchable memory BARs are assigned smallest to largest starting at the 4 GB address assigning memory ascending above the 4 GB limit throughout the full 64-bit memory space.

   If the addr_map_4 GB_limit input to the ebfm_cfg_rp_ep is set to 1, then the 32-bit and the 64-bit prefetchable memory BARs are assigned largest to smallest starting at the 4 GB address and assigning memory by descending below the 4 GB address to addresses memory as needed down to the ending address of the last 32-bit non-prefetchable BAR.

   The above algorithm cannot always assign values to all BARs when there are a few very large (1 GB or greater) 32-bit BARs. Although assigning addresses to all BARs may be possible, a more complex algorithm would be required to effectively assign these addresses. However, such a configuration is unlikely to be useful in real systems. If the procedure is unable to assign the BARs, it displays an error message and stops the simulation.

4. Based on the above BAR assignments, the Root Port Configuration Space address windows are assigned to encompass the valid BAR address ranges.
5. The Endpoint PCI control register is set to enable master transactions, memory address decoding, and I/O address decoding.

The ebfm_cfg_rp_ep procedure also sets up a bar_table data structure in BFM shared memory that lists the sizes and assigned addresses of all Endpoint BARs. This area of BFM shared memory is write-protected, which means any user write accesses to this area cause a fatal simulation error. This data structure is then used by subsequent BFM procedure calls to generate the full PCI Express addresses for read and write requests to particular offsets from a BAR. This procedure allows the testbench code that accesses the Endpoint Application Layer to be written to use offsets from a BAR and not have to keep track of the specific addresses assigned to the BAR. The following table shows how those offsets are used.

Table 103. BAR Table Structure

Offset (Bytes)	Description
+0	PCI Express address in BAR0
+4	PCI Express address in BAR1
+8	PCI Express address in BAR2
+12	PCI Express address in BAR3
+16	PCI Express address in BAR4
+20	PCI Express address in BAR5
+24	PCI Express address in Expansion ROM BAR
+28	Reserved
+32	BAR0 read back value after being written with all 1's (used to compute size)
+36	BAR1 read back value after being written with all 1's

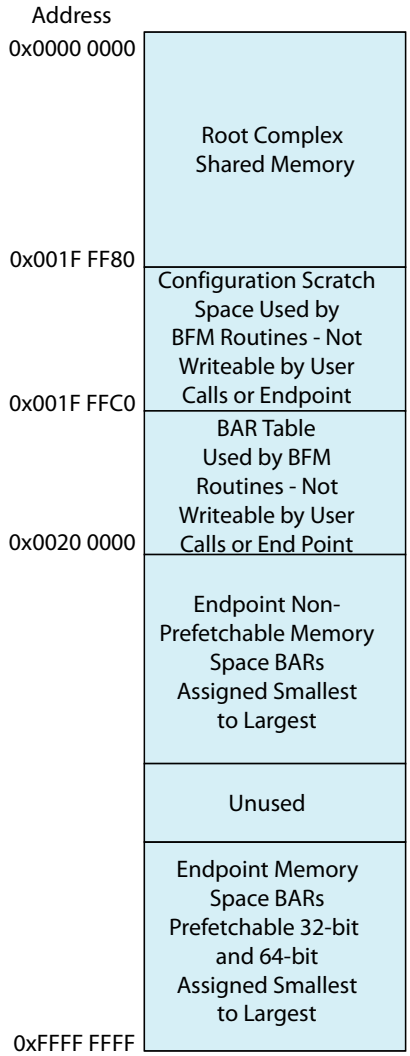
continued...

Offset (Bytes)	Description
+40	BAR2 read back value after being written with all 1's
+44	BAR3 read back value after being written with all 1's
+48	BAR4 read back value after being written with all 1's
+52	BAR5 read back value after being written with all 1's
+56	Expansion ROM BAR read back value after being written with all 1's
+60	Reserved

The configuration routine does not configure any advanced PCI Express capabilities such as the AER capability.

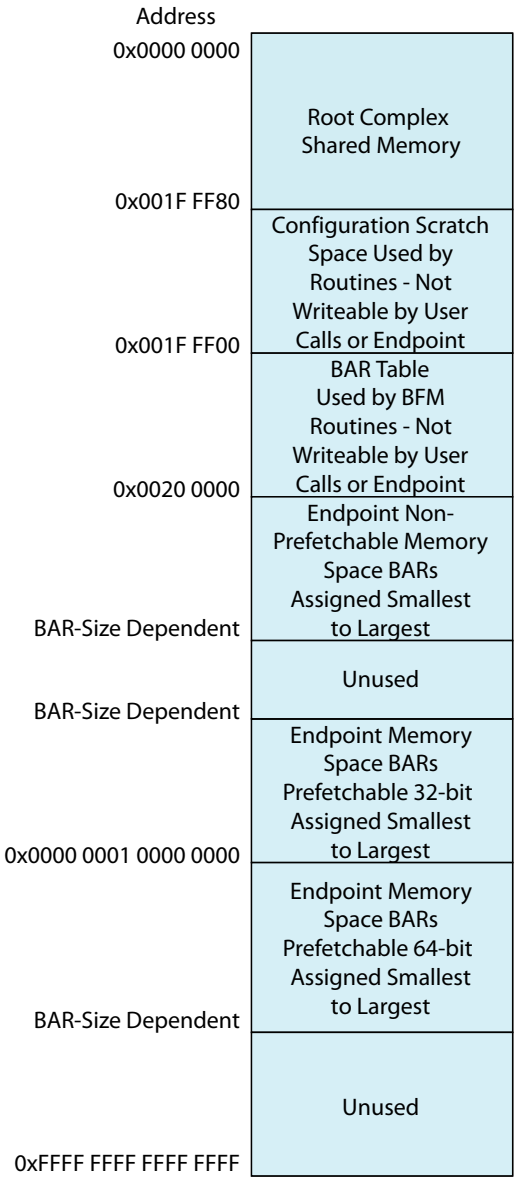
Besides the `ebfm_cfg_rp_ep` procedure in **altpcietb_bfm_driver_rp.v**, routines to read and write Endpoint Configuration Space registers directly are available in the Verilog HDL include file. After the `ebfm_cfg_rp_ep` procedure is run the PCI Express I/O and Memory Spaces have the layout as described in the following three figures. The memory space layout is dependent on the value of the **add r_map_4GB_limit** input parameter. If **addr_map_4GB_limit** is 1 the resulting memory space map is shown in the following figure.

Figure 117. Memory Space Layout—4 GB Limit



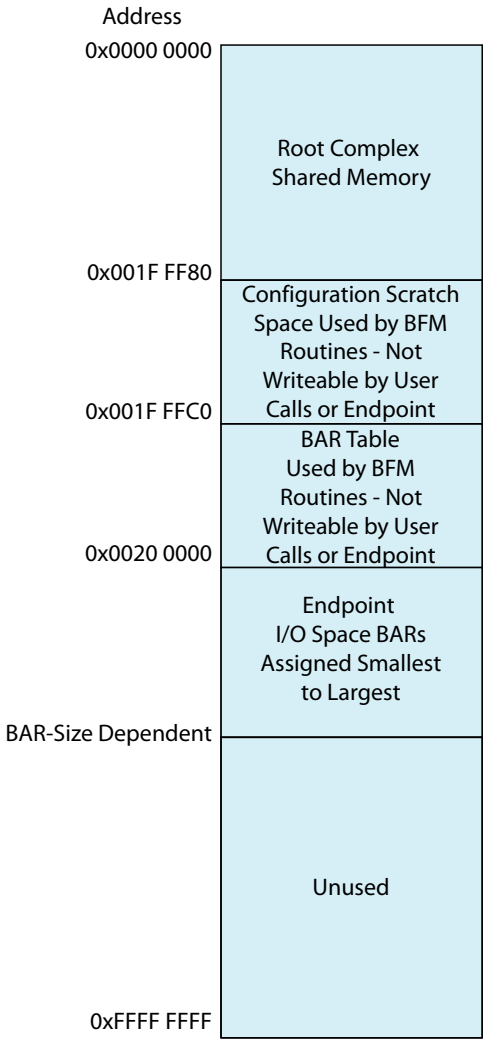
If **addr_map_4GB_limit** is 0, the resulting memory space map is shown in the following figure.

Figure 118. Memory Space Layout—No Limit



The following figure shows the I/O address space.

Figure 119. I/O Address Space



17.8.4 Issuing Read and Write Transactions to the Application Layer

Read and write transactions are issued to the Endpoint Application Layer by calling one of the `ebfm_bar` procedures in **altpcietb_bfm_driver_rp.v**. The procedures and functions listed below are available in the Verilog HDL include file **altpcietb_bfm_driver_rp.v**. The complete list of available procedures and functions is as follows:

- `ebfm_barwr`—writes data from BFM shared memory to an offset from a specific Endpoint BAR. This procedure returns as soon as the request has been passed to the VC interface module for transmission.
- `ebfm_barwr_imm`—writes a maximum of four bytes of immediate data (passed in a procedure call) to an offset from a specific Endpoint BAR. This procedure returns as soon as the request has been passed to the VC interface module for transmission.
- `ebfm_barrd_wait`—reads data from an offset of a specific Endpoint BAR and stores it in BFM shared memory. This procedure blocks waiting for the completion data to be returned before returning control to the caller.
- `ebfm_barrd_nowt`—reads data from an offset of a specific Endpoint BAR and stores it in the BFM shared memory. This procedure returns as soon as the request has been passed to the VC interface module for transmission, allowing subsequent reads to be issued in the interim.

These routines take as parameters a BAR number to access the memory space and the BFM shared memory address of the `bar_table` data structure that was set up by the `ebfm_cfg_rp_ep` procedure. (Refer to *Configuration of Root Port and Endpoint*.) Using these parameters simplifies the BFM test driver routines that access an offset from a specific BAR and eliminates calculating the addresses assigned to the specified BAR.

The Root Port BFM does not support accesses to Endpoint I/O space BARs.

Related Links

Configuration of Root Port and Endpoint on page 196

17.9 BFM Procedures and Functions

The BFM includes procedures, functions, and tasks to drive Endpoint application testing. It also includes procedures to run the chaining DMA design example.

The BFM read and write procedures read and write data among BFM shared memory, Endpoint BARs, and specified configuration registers. The procedures and functions are available in the Verilog HDL. They are in the include file **altpcietb_bfm_driver.v**. These procedures and functions support issuing memory and configuration transactions on the PCI Express link.

17.9.1 ebfm_barwr Procedure

The `ebfm_barwr` procedure writes a block of data from BFM shared memory to an offset from the specified Endpoint BAR. The length can be longer than the configured `MAXIMUM_PAYLOAD_SIZE`; the procedure breaks the request up into multiple transactions as needed. This routine returns as soon as the last transaction has been accepted by the VC interface module.

Location	altpcietb_bfm_rdwr.v	
Syntax	`ebfm_barwr(bar_table, bar_num, pcie_offset, lcladdr, byte_len, tclass)`	
Arguments	`bar_table`	Address of the Endpoint `bar_table` structure in BFM shared memory. The `bar_table` structure stores the address assigned to each BAR so that the driver code does not need to be aware of the actual assigned addresses only the application specific offsets from the BAR.
	`bar_num`	Number of the BAR used with `pcie_offset` to determine PCI Express address.
	`pcie_offset`	Address offset from the BAR base.
	`lcladdr`	BFM shared memory address of the data to be written.
	`byte_len`	Length, in bytes, of the data written. Can be 1 to the minimum of the bytes remaining in the BAR space or BFM shared memory.
	`tclass`	Traffic class used for the PCI Express transaction.

17.9.2 ebfm_barwr_imm Procedure

The `ebfm_barwr_imm` procedure writes up to four bytes of data to an offset from the specified Endpoint BAR.

Location	altpcietb_bfm_driver_rp.v	
Syntax	`ebfm_barwr_imm(bar_table, bar_num, pcie_offset, imm_data, byte_len, tclass)`	
Arguments	`bar_table`	Address of the Endpoint `bar_table` structure in BFM shared memory. The `bar_table` structure stores the address assigned to each BAR so that the driver code does not need to be aware of the actual assigned addresses only the application specific offsets from the BAR.
	`bar_num`	Number of the BAR used with `pcie_offset` to determine PCI Express address.
	`pcie_offset`	Address offset from the BAR base.
	`imm_data`	Data to be written. In Verilog HDL, this argument is `reg [31:0]`.In both languages, the bits written depend on the length as follows: Length Bits Written • 4: 31 downto 0 • 3: 23 downto 0 • 2: 15 downto 0 • 1: 7 downto 0
	`byte_len`	Length of the data to be written in bytes. Maximum length is 4 bytes.
	`tclass`	Traffic class to be used for the PCI Express transaction.

17.9.3 ebfm_barrd_wait Procedure

The `ebfm_barrd_wait` procedure reads a block of data from the offset of the specified Endpoint BAR and stores it in BFM shared memory. The length can be longer than the configured maximum read request size; the procedure breaks the request up into multiple transactions as needed. This procedure waits until all of the completion data is returned and places it in shared memory.

Location	**altpcietb_bfm_driver_rp.v**
Syntax	`ebfm_barrd_wait(bar_table, bar_num, pcie_offset, lcladdr, byte_len, tclass)`

continued...

Arguments	bar_table	Address of the Endpoint bar_table structure in BFM shared memory. The bar_table structure stores the address assigned to each BAR so that the driver code does not need to be aware of the actual assigned addresses only the application specific offsets from the BAR.
	bar_num	Number of the BAR used with pcie_offset to determine PCI Express address.
	pcie_offset	Address offset from the BAR base.
	lcladdr	BFM shared memory address where the read data is stored.
	byte_len	Length, in bytes, of the data to be read. Can be 1 to the minimum of the bytes remaining in the BAR space or BFM shared memory.
	tclass	Traffic class used for the PCI Express transaction.

17.9.4 ebfm_barrd_nowt Procedure

The ebfm_barrd_nowt procedure reads a block of data from the offset of the specified Endpoint BAR and stores the data in BFM shared memory. The length can be longer than the configured maximum read request size; the procedure breaks the request up into multiple transactions as needed. This routine returns as soon as the last read transaction has been accepted by the VC interface module, allowing subsequent reads to be issued immediately.

Location	altpcietb_b fm_driver_rp.v	
Syntax	ebfm_barrd_nowt(bar_table, bar_num, pcie_offset, lcladdr, byte_len, tclass)	
Arguments	bar_table	Address of the Endpoint bar_table structure in BFM shared memory.
	bar_num	Number of the BAR used with pcie_offset to determine PCI Express address.
	pcie_offset	Address offset from the BAR base.
	lcladdr	BFM shared memory address where the read data is stored.
	byte_len	Length, in bytes, of the data to be read. Can be 1 to the minimum of the bytes remaining in the BAR space or BFM shared memory.
	tclass	Traffic Class to be used for the PCI Express transaction.

17.9.5 ebfm_cfgwr_imm_wait Procedure

The ebfm_cfgwr_imm_wait procedure writes up to four bytes of data to the specified configuration register. This procedure waits until the write completion has been returned.

Location	altpcietb_bfm_driver_rp.v	
Syntax	ebfm_cfgwr_imm_wait(bus_num, dev_num, fnc_num, imm_regb_ad, regb_ln, imm_data, compl_status	
Arguments	bus_num	PCI Express bus number of the target device.
	dev_num	PCI Express device number of the target device.
	fnc_num	Function number in the target device to be accessed.
	regb_ad	Byte-specific address of the register to be written.

continued...

Location	altpcietb_bfm_driver_rp.v	
	regb_ln	Length, in bytes, of the data written. Maximum length is four bytes. The regb_ln and the regb_ad arguments cannot cross a DWORD boundary.
	imm_data	Data to be written. This argument is reg [31:0]. The bits written depend on the length: • 4: 31 downto 0 • 3: 23 downto 0 • 2: 15 downto 0 • 1: 7 downto 0
	compl_status	This argument is reg [2:0]. This argument is the completion status as specified in the PCI Express specification. The following encodings are defined: • 3'b000: SC— Successful completion • 3'b001: UR— Unsupported Request • 3'b010: CRS — Configuration Request Retry Status • 3'b100: CA — Completer Abort

17.9.6 ebfm_cfgwr_imm_nowt Procedure

The ebfm_cfgwr_imm_nowt procedure writes up to four bytes of data to the specified configuration register. This procedure returns as soon as the VC interface module accepts the transaction, allowing other writes to be issued in the interim. Use this procedure only when successful completion status is expected.

Location	altpcietb_bfm_driver_rp.v	
Syntax	ebfm_cfgwr_imm_nowt(bus_num, dev_num, fnc_num, imm_regb_adr, regb_len, imm_data)	
Arguments	bus_num	PCI Express bus number of the target device.
	dev_num	PCI Express device number of the target device.
	fnc_num	Function number in the target device to be accessed.
	regb_ad	Byte-specific address of the register to be written.
	regb_ln	Length, in bytes, of the data written. Maximum length is four bytes, The regb_ln the regb_ad arguments cannot cross a DWORD boundary.
	imm_data	Data to be written This argument is reg [31:0]. In both languages, the bits written depend on the length. The following encodes are defined. • 4: [31:0] • 3: [23:0] • 2: [15:0] • 1: [7:0]

17.9.7 ebfm_cfgrd_wait Procedure

The ebfm_cfgrd_wait procedure reads up to four bytes of data from the specified configuration register and stores the data in BFM shared memory. This procedure waits until the read completion has been returned.

Location	altpcietb_bfm_driver_rp.v	
Syntax	`ebfm_cfgrd_wait(bus_num, dev_num, fnc_num, regb_ad, regb_ln, lcladdr, compl_status)`	
Arguments	`bus_num`	PCI Express bus number of the target device.
	`dev_num`	PCI Express device number of the target device.
	`fnc_num`	Function number in the target device to be accessed.
	`regb_ad`	Byte-specific address of the register to be written.
	`regb_ln`	Length, in bytes, of the data read. Maximum length is four bytes. The `regb_ln` and the `regb_ad` arguments cannot cross a DWORD boundary.
	`lcladdr`	BFM shared memory address of where the read data should be placed.
	`compl_status`	Completion status for the configuration transaction. This argument is reg [2:0]. In both languages, this is the completion status as specified in the PCI Express specification. The following encodings are defined. • 3'b000: SC— Successful completion • 3'b001: UR— Unsupported Request • 3'b010: CRS — Configuration Request Retry Status • 3'b100: CA — Completer Abort

17.9.8 ebfm_cfgrd_nowt Procedure

The `ebfm_cfgrd_nowt` procedure reads up to four bytes of data from the specified configuration register and stores the data in the BFM shared memory. This procedure returns as soon as the VC interface module has accepted the transaction, allowing other reads to be issued in the interim. Use this procedure only when successful completion status is expected and a subsequent read or write with a wait can be used to guarantee the completion of this operation.

Location	altpcietb_bfm_driver_rp.v	
Syntax	`ebfm_cfgrd_nowt(bus_num, dev_num, fnc_num, regb_ad, regb_ln, lcladdr)`	
Arguments	`bus_num`	PCI Express bus number of the target device.
	`dev_num`	PCI Express device number of the target device.
	`fnc_num`	Function number in the target device to be accessed.
	`regb_ad`	Byte-specific address of the register to be written.
	`regb_ln`	Length, in bytes, of the data written. Maximum length is four bytes. The `regb_ln` and `regb_ad` arguments cannot cross a DWORD boundary.
	`lcladdr`	BFM shared memory address where the read data should be placed.

17.9.9 BFM Configuration Procedures

The BFM configuration procedures are available in **altpcietb_bfm_driver_rp.v**. These procedures support configuration of the Root Port and Endpoint Configuration Space registers.

All Verilog HDL arguments are type `integer` and are input-only unless specified otherwise.

17.9.9.1 ebfm_cfg_rp_ep Procedure

The ebfm_cfg_rp_ep procedure configures the Root Port and Endpoint Configuration Space registers for operation.

Location	altpcietb_bfm_driver_rp.v	
Syntax	ebfm_cfg_rp_ep(bar_table, ep_bus_num, ep_dev_num, rp_max_rd_req_size, display_ep_config, addr_map_4GB_limit)	
Arguments	bar_table	Address of the Endpoint bar_table structure in BFM shared memory. This routine populates the bar_table structure. The bar_table structure stores the size of each BAR and the address values assigned to each BAR. The address of the bar_table structure is passed to all subsequent read and write procedure calls that access an offset from a particular BAR.
	ep_bus_num	PCI Express bus number of the target device. This number can be any value greater than 0. The Root Port uses this as its secondary bus number.
	ep_dev_num	PCI Express device number of the target device. This number can be any value. The Endpoint is automatically assigned this value when it receives its first configuration transaction.
	rp_max_rd_req_size	Maximum read request size in bytes for reads issued by the Root Port. This parameter must be set to the maximum value supported by the Endpoint Application Layer. If the Application Layer only supports reads of the MAXIMUM_PAYLOAD_SIZE, then this can be set to 0 and the read request size will be set to the maximum payload size. Valid values for this argument are 0, 128, 256, 512, 1,024, 2,048 and 4,096.
	display_ep_config	When set to 1 many of the Endpoint Configuration Space registers are displayed after they have been initialized, causing some additional reads of registers that are not normally accessed during the configuration process such as the Device ID and Vendor ID.
	addr_map_4GB_limit	When set to 1 the address map of the simulation system will be limited to 4 GB. Any 64-bit BARs will be assigned below the 4 GB limit.

17.9.9.2 ebfm_cfg_decode_bar Procedure

The ebfm_cfg_decode_bar procedure analyzes the information in the BAR table for the specified BAR and returns details about the BAR attributes.

Location	altpcietb_bfm_driver_rp.v	
Syntax	ebfm_cfg_decode_bar(bar_table, bar_num, log2_size, is_mem, is_pref, is_64b)	
Arguments	bar_table	Address of the Endpoint bar_table structure in BFM shared memory.
	bar_num	BAR number to analyze.
	log2_size	This argument is set by the procedure to the log base 2 of the size of the BAR. If the BAR is not enabled, this argument will be set to 0.
	is_mem	The procedure sets this argument to indicate if the BAR is a memory space BAR (1) or I/O Space BAR (0).
	is_pref	The procedure sets this argument to indicate if the BAR is a prefetchable BAR (1) or non-prefetchable BAR (0).
	is_64b	The procedure sets this argument to indicate if the BAR is a 64-bit BAR (1) or 32-bit BAR (0). This is set to 1 only for the lower numbered BAR of the pair.

17.9.10 BFM Shared Memory Access Procedures

The BFM shared memory access procedures and functions are in the Verilog HDL include file **altpcietb_bfm_driver.v**. These procedures and functions support accessing the BFM shared memory.

17.9.10.1 Shared Memory Constants

The following constants are defined in **altpcietb_bfm_driver.v**. They select a data pattern in the `shmem_fill` and `shmem_chk_ok` routines. These shared memory constants are all Verilog HDL type `integer`.

Table 104. Constants: Verilog HDL Type INTEGER

Constant	Description
`SHMEM_FILL_ZEROS`	Specifies a data pattern of all zeros
`SHMEM_FILL_BYTE_INC`	Specifies a data pattern of incrementing 8-bit bytes (0x00, 0x01, 0x02, etc.)
`SHMEM_FILL_WORD_INC`	Specifies a data pattern of incrementing 16-bit words (0x0000, 0x0001, 0x0002, etc.)
`SHMEM_FILL_DWORD_INC`	Specifies a data pattern of incrementing 32-bit dwords (0x00000000, 0x00000001, 0x00000002, etc.)
`SHMEM_FILL_QWORD_INC`	Specifies a data pattern of incrementing 64-bit qwords (0x0000000000000000, 0x0000000000000001, 0x0000000000000002, etc.)
`SHMEM_FILL_ONE`	Specifies a data pattern of all ones

17.9.10.2 shmem_write

The `shmem_write` procedure writes data to the BFM shared memory.

Location	altpcietb_bfm_driver_rp.v	
Syntax	`shmem_write(addr, data, leng)`	
Arguments	`addr`	BFM shared memory starting address for writing data
	`data`	Data to write to BFM shared memory. This parameter is implemented as a 64-bit vector. `leng` is 1–8 bytes. Bits 7 downto 0 are written to the location specified by `addr`; bits 15 downto 8 are written to the `addr+1` location, etc.
	`length`	Length, in bytes, of data written

17.9.10.3 shmem_read Function

The `shmem_read` function reads data to the BFM shared memory.

Location	altpcietb_bfm_driver_rp.v	
Syntax	`data:= shmem_read(addr, leng)`	
Arguments	`addr`	BFM shared memory starting address for reading data
	`leng`	Length, in bytes, of data read
Return	`data`	Data read from BFM shared memory.

continued...

Location	altpcietb_bfm_driver_rp.v	
		This parameter is implemented as a 64-bit vector. leng is 1- 8 bytes. If leng is less than 8 bytes, only the corresponding least significant bits of the returned data are valid. Bits 7 downto 0 are read from the location specified by addr; bits 15 downto 8 are read from the addr+1 location, etc.

17.9.10.4 shmem_display Verilog HDL Function

The shmem_display Verilog HDL function displays a block of data from the BFM shared memory.

Location	altpcietb_bfm_driver_rp.v	
Syntax	Verilog HDL: dummy_return:=shmem_display(addr, leng, word_size, flag_addr, msg_type);	
Arguments	addr	BFM shared memory starting address for displaying data.
	leng	Length, in bytes, of data to display.
	word_size	Size of the words to display. Groups individual bytes into words. Valid values are 1, 2, 4, and 8.
	flag_addr	Adds a <== flag to the end of the display line containing this address. Useful for marking specific data. Set to a value greater than 2**21 (size of BFM shared memory) to suppress the flag.
	msg_type	Specifies the message type to be displayed at the beginning of each line. See "BFM Log and Message Procedures" on page 18–37 for more information about message types. Set to one of the constants defined in Table 18–36 on page 18–41.

17.9.10.5 shmem_fill Procedure

The shmem_fill procedure fills a block of BFM shared memory with a specified data pattern.

Location	altpcietb_bfm_driver_rp.v	
Syntax	shmem_fill(addr, mode, leng, init)	
Arguments	addr	BFM shared memory starting address for filling data.
	mode	Data pattern used for filling the data. Should be one of the constants defined in section *Shared Memory Constants*.
	leng	Length, in bytes, of data to fill. If the length is not a multiple of the incrementing data pattern width, then the last data pattern is truncated to fit.
	init	Initial data value used for incrementing data pattern modes. This argument is reg [63:0]. The necessary least significant bits are used for the data patterns that are smaller than 64 bits.

Related Links

Shared Memory Constants on page 208

17.9.10.6 shmem_chk_ok Function

The shmem_chk_ok function checks a block of BFM shared memory against a specified data pattern.

Location	altpcietb_bfm_shmem.v	
Syntax	`result:= shmem_chk_ok(addr, mode, leng, init, display_error)`	
Arguments	`addr`	BFM shared memory starting address for checking data.
	`mode`	Data pattern used for checking the data. Should be one of the constants defined in section "Shared Memory Constants" on page 18–35.
	`leng`	Length, in bytes, of data to check.
	`init`	This argument is `reg [63:0]`.The necessary least significant bits are used for the data patterns that are smaller than 64-bits.
	`display_error`	When set to 1, this argument displays the miscomparing data on the simulator standard output.
Return	`Result`	Result is 1-bit. • 1'b1 — Data patterns compared successfully • 1'b0 — Data patterns did not compare successfully

17.9.11 BFM Log and Message Procedures

The following procedures and functions are available in the Verilog HDL include file **altpcietb_bfm_driver_rp.v.**

These procedures provide support for displaying messages in a common format, suppressing informational messages, and stopping simulation on specific message types.

The following constants define the type of message and their values determine whether a message is displayed or simulation is stopped after a specific message. Each displayed message has a specific prefix, based on the message type in the following table.

You can suppress the display of certain message types. The default values determining whether a message type is displayed are defined in the following table. To change the default message display, modify the display default value with a procedure call to `ebfm_log_set_suppressed_msg_mask`.

Certain message types also stop simulation after the message is displayed. The following table shows the default value determining whether a message type stops simulation. You can specify whether simulation stops for particular messages with the procedure `ebfm_log_set_stop_on_msg_mask`.

All of these log message constants type `integer`.

Table 105. Log Messages

Constant (Message Type)	Description	Mask Bit No	Display by Default	Simulation Stops by Default	Message Prefix
`EBFM_MSG_DEBUG`	Specifies debug messages.	0	No	No	`DEBUG:`
`EBFM_MSG_INFO`	Specifies informational messages, such as configuration register values, starting and ending of tests.	1	Yes	No	`INFO:`

continued...

Constant (Message Type)	Description	Mask Bit No	Display by Default	Simulation Stops by Default	Message Prefix
EBFM_MSG_WARNING	Specifies warning messages, such as tests being skipped due to the specific configuration.	2	Yes	No	WARNING:
EBFM_MSG_ERROR_INFO	Specifies additional information for an error. Use this message to display preliminary information before an error message that stops simulation.	3	Yes	No	ERROR:
EBFM_MSG_ERROR_CONTINUE	Specifies a recoverable error that allows simulation to continue. Use this error for data miscompares.	4	Yes	No	ERROR:
EBFM_MSG_ERROR_FATAL	Specifies an error that stops simulation because the error leaves the testbench in a state where further simulation is not possible.	N/A	Yes Cannot suppress	Yes Cannot suppress	FATAL:
EBFM_MSG_ERROR_FATAL_TB_ERR	Used for BFM test driver or Root Port BFM fatal errors. Specifies an error that stops simulation because the error leaves the testbench in a state where further simulation is not possible. Use this error message for errors that occur due to a problem in the BFM test driver module or the Root Port BFM, that are not caused by the Endpoint Application Layer being tested.	N/A	Y Cannot suppress	Y Cannot suppress	FATAL:

17.9.11.1 ebfm_display Verilog HDL Function

The `ebfm_display` procedure or function displays a message of the specified type to the simulation standard output and also the log file if `ebfm_log_open` is called.

A message can be suppressed, simulation can be stopped or both based on the default settings of the message type and the value of the bit mask when each of the procedures listed below is called. You can call one or both of these procedures based on what messages you want displayed and whether or not you want simulation to stop for specific messages.

- When `ebfm_log_set_suppressed_msg_mask` is called, the display of the message might be suppressed based on the value of the bit mask.
- When `ebfm_log_set_stop_on_msg_mask` is called, the simulation can be stopped after the message is displayed, based on the value of the bit mask.

Location	altpcietb_bfm_driver_rp.v	
Syntax	`Verilog HDL: dummy_return:=ebfm_display(msg_type, message);`	
Argument	`msg_type`	Message type for the message. Should be one of the constants defined in Table 18–36 on page 18–41.
	`message`	The message string is limited to a maximum of 100 characters. Also, because Verilog HDL does not allow variable length strings, this routine strips off leading characters of 8'h00 before displaying the message.
Return	`always 0`	Applies only to the Verilog HDL routine.

17.9.11.2 ebfm_log_stop_sim Verilog HDL Function

The `ebfm_log_stop_sim` procedure stops the simulation.

Location	altpcietb_bfm_driver_rp.v	
Syntax	Verilog HDL: `return:=ebfm_log_stop_sim(success);`	
Argument	`success`	When set to a 1, this process stops the simulation with a message indicating successful completion. The message is prefixed with SUCCESS:. Otherwise, this process stops the simulation with a message indicating unsuccessful completion. The message is prefixed with FAILURE:.
Return	Always 0	This value applies only to the Verilog HDL function.

17.9.11.3 ebfm_log_set_suppressed_msg_mask #Verilog HDL Function

The `ebfm_log_set_suppressed_msg_mask` procedure controls which message types are suppressed.

Location	altpcietb_bfm_driver_rp.v	
Syntax	`bfm_log_set_suppressed_msg_mask (msg_mask)`	
Argument	`msg_mask`	This argument is `reg [EBFM_MSG_ERROR_CONTINUE: EBFM_MSG_DEBUG]`. A 1 in a specific bit position of the `msg_mask` causes messages of the type corresponding to the bit position to be suppressed.

17.9.11.4 ebfm_log_set_stop_on_msg_mask Verilog HDL Function

The `ebfm_log_set_stop_on_msg_mask` procedure controls which message types stop simulation. This procedure alters the default behavior of the simulation when errors occur as described in the *BFM Log and Message Procedures*.

Location	altpcietb_bfm_driver_rp.v	
Syntax	`ebfm_log_set_stop_on_msg_mask (msg_mask)`	
Argument	`msg_mask`	This argument is `reg [EBFM_MSG_ERROR_CONTINUE:EBFM_MSG_DEBUG]`. A 1 in a specific bit position of the `msg_mask` causes messages of the type corresponding to the bit position to stop the simulation after the message is displayed.

Related Links

BFM Log and Message Procedures on page 210

17.9.11.5 ebfm_log_open Verilog HDL Function

The `ebfm_log_open` procedure opens a log file of the specified name. All displayed messages are called by `ebfm_display` and are written to this log file as simulator standard output.

Location	altpcietb_bfm_driver_rp.v	
Syntax	`ebfm_log_open (fn)`	
Argument	`fn`	This argument is type `string` and provides the file name of log file to be opened.

17.9.11.6 ebfm_log_close Verilog HDL Function

The `ebfm_log_close` procedure closes the log file opened by a previous call to `ebfm_log_open`.

Location	altpcietb_bfm_driver_rp.v
Syntax	`ebfm_log_close`
Argument	NONE

17.9.12 Verilog HDL Formatting Functions

The Verilog HDL Formatting procedures and functions are available in the **altpcietb_bfm_driver_rp.v**. The formatting functions are only used by Verilog HDL. All these functions take one argument of a specified length and return a vector of a specified length.

17.9.12.1 himage1

This function creates a one-digit hexadecimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_driver_rp.v	
Syntax	`string:= himage(vec)`	
Argument	`vec`	Input data type `reg` with a `range` of 3:0.
Return range	`string`	Returns a 1-digit hexadecimal representation of the input argument. Return data is type `reg` with a `range` of 8:1

17.9.12.2 himage2

This function creates a two-digit hexadecimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_driver_rp.v	
Syntax	`string:= himage(vec)`	
Argument range	`vec`	Input data type `reg` with a `range` of 7:0.
Return range	`string`	Returns a 2-digit hexadecimal presentation of the input argument, padded with leading 0s, if they are needed. Return data is type `reg` with a `range` of 16:1

17.9.12.3 himage4

This function creates a four-digit hexadecimal string representation of the input argument can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_driver_rp.v	
Syntax	string:= himage(vec)	
Argument range	vec	Input data type reg with a range of 15:0.
Return range	Returns a four-digit hexadecimal representation of the input argument, padded with leading 0s, if they are needed. Return data is type reg with a range of 32:1.	

17.9.12.4 himage8

This function creates an 8-digit hexadecimal string representation of the input argument that can be concatenated into a larger message string and passed to ebfm_display.

Location	altpcietb_bfm_driver_rp.v	
Syntax	string:= himage(vec)	
Argument range	vec	Input data type reg with a range of 31:0.
Return range	string	Returns an 8-digit hexadecimal representation of the input argument, padded with leading 0s, if they are needed. Return data is type reg with a range of 64:1.

17.9.12.5 himage16

This function creates a 16-digit hexadecimal string representation of the input argument that can be concatenated into a larger message string and passed to ebfm_display.

Location	altpcietb_bfm_driver_rp.v	
Syntax	string:= himage(vec)	
Argument range	vec	Input data type reg with a range of 63:0.
Return range	string	Returns a 16-digit hexadecimal representation of the input argument, padded with leading 0s, if they are needed. Return data is type reg with a range of 128:1.

17.9.12.6 dimage1

This function creates a one-digit decimal string representation of the input argument that can be concatenated into a larger message string and passed to ebfm_display.

Location	altpcietb_bfm_driver_rp.v	
Syntax	string:= dimage(vec)	
Argument range	vec	Input data type reg with a range of 31:0.
Return range	string	Returns a 1-digit decimal representation of the input argument that is padded with leading 0s if necessary. Return data is type reg with a range of 8:1. Returns the letter *U* if the value cannot be represented.

17.9.12.7 dimage2

This function creates a two-digit decimal string representation of the input argument that can be concatenated into a larger message string and passed to ebfm_display.

Location	altpcietb_bfm_driver_rp.v	
Syntax	string:= dimage(vec)	
Argument range	vec	Input data type reg with a range of 31:0.
Return range	string	Returns a 2-digit decimal representation of the input argument that is padded with leading 0s if necessary. Return data is type reg with a range of 16:1. Returns the letter *U* if the value cannot be represented.

17.9.12.8 dimage3

This function creates a three-digit decimal string representation of the input argument that can be concatenated into a larger message string and passed to ebfm_display.

Location	altpcietb_bfm_driver_rp.v	
Syntax	string:= dimage(vec)	
Argument range	vec	Input data type reg with a range of 31:0.
Return range	string	Returns a 3-digit decimal representation of the input argument that is padded with leading 0s if necessary. Return data is type reg with a range of 24:1. Returns the letter *U* if the value cannot be represented.

17.9.12.9 dimage4

This function creates a four-digit decimal string representation of the input argument that can be concatenated into a larger message string and passed to ebfm_display.

Location	altpcietb_bfm_driver_rp.v	
Syntax	string:= dimage(vec)	
Argument range	vec	Input data type reg with a range of 31:0.
Return range	string	Returns a 4-digit decimal representation of the input argument that is padded with leading 0s if necessary. Return data is type reg with a range of 32:1. Returns the letter *U* if the value cannot be represented.

17.9.12.10 dimage5

This function creates a five-digit decimal string representation of the input argument that can be concatenated into a larger message string and passed to ebfm_display.

Location	altpcietb_bfm_driver_rp.v	
Syntax	string:= dimage(vec)	
Argument range	vec	Input data type reg with a range of 31:0.
Return range	string	Returns a 5-digit decimal representation of the input argument that is padded with leading 0s if necessary. Return data is type reg with a range of 40:1. Returns the letter *U* if the value cannot be represented.

17.9.12.11 dimage6

This function creates a six-digit decimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_log.v	
Syntax	`string:= dimage(vec)`	
Argument range	`vec`	Input data type `reg` with a `range` of 31:0.
Return range	`string`	Returns a 6-digit decimal representation of the input argument that is padded with leading 0s if necessary. Return data is type `reg` with a `range` of 48:1. Returns the letter *U* if the value cannot be represented.

17.9.12.12 dimage7

This function creates a seven-digit decimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_log.v	
Syntax	`string:= dimage(vec)`	
Argument range	`vec`	Input data type `reg` with a `range` of 31:0.
Return range	`string`	Returns a 7-digit decimal representation of the input argument that is padded with leading 0s if necessary. Return data is type `reg` with a `range` of 56:1. Returns the letter <*U*> if the value cannot be represented.

17.9.13 Procedures and Functions Specific to the Chaining DMA Design Example

The procedures specific to the chaining DMA design example are in the Verilog HDL module file **altpcietb_bfm_driver_rp.v**.

17.9.13.1 chained_dma_test Procedure

The `chained_dma_test` procedure is the top-level procedure that runs the chaining DMA read and the chaining DMA write

Location	altpcietb_bfm_driver_rp.v	
Syntax	`chained_dma_test (bar_table, bar_num, direction, use_msi, use_eplast)`	
Arguments	`bar_table`	Address of the Endpoint `bar_table` structure in BFM shared memory.
	`bar_num`	BAR number to analyze.
	`direction`	When 0 the direction is read. When 1 the direction is write.
	`Use_msi`	When set, the Root Port uses native PCI Express MSI to detect the DMA completion.
	`Use_eplast`	When set, the Root Port uses BFM shared memory polling to detect the DMA completion.

17.9.13.2 dma_rd_test Procedure

Use the dma_rd_test procedure for DMA reads from the Endpoint memory to the BFM shared memory.

Location	altpcietb_bfm_driver_rp.v	
Syntax	dma_rd_test (bar_table, bar_num, use_msi, use_eplast)	
Arguments	bar_table	Address of the Endpoint bar_table structure in BFM shared memory.
	bar_num	BAR number to analyze.
	Use_msi	When set, the Root Port uses native PCI express MSI to detect the DMA completion.
	Use_eplast	When set, the Root Port uses BFM shared memory polling to detect the DMA completion.

17.9.13.3 dma_wr_test Procedure

Use the dma_wr_test procedure for DMA writes from the BFM shared memory to the Endpoint memory.

Location	altpcietb_bfm_driver_rp.v	
Syntax	dma_wr_test (bar_table, bar_num, use_msi, use_eplast)	
Arguments	bar_table	Address of the Endpoint bar_table structure in BFM shared memory.
	bar_num	BAR number to analyze.
	Use_msi	When set, the Root Port uses native PCI Express MSI to detect the DMA completion.
	Use_eplast	When set, the Root Port uses BFM shared memory polling to detect the DMA completion.

17.9.13.4 dma_set_rd_desc_data Procedure

Use the dma_set_rd_desc_data procedure to configure the BFM shared memory for the DMA read.

Location	altpcietb_bfm_driver_rp.v	
Syntax	dma_set_rd_desc_data (bar_table, bar_num)	
Arguments	bar_table	Address of the Endpoint bar_table structure in BFM shared memory.
	bar_num	BAR number to analyze.

17.9.13.5 dma_set_wr_desc_data Procedure

Use the dma_set_wr_desc_data procedure to configure the BFM shared memory for the DMA write.

Location	altpcietb_bfm_driver_rp.v	
Syntax	dma_set_wr_desc_data_header (bar_table, bar_num)	
Arguments	bar_table	Address of the Endpoint bar_table structure in BFM shared memory.
	bar_num	BAR number to analyze.

17.9.13.6 dma_set_header Procedure

Use the `dma_set_header` procedure to configure the DMA descriptor table for DMA read or DMA write.

Location	altpcietb_bfm_driver_rp.v	
Syntax	`dma_set_header (bar_table, bar_num, Descriptor_size, direction, Use_msi, Use_eplast, Bdt_msb, Bdt_lab, Msi_number, Msi_traffic_class, Multi_message_enable)`	
Arguments	`bar_table`	Address of the Endpoint `bar_table` structure in BFM shared memory.
	`bar_num`	BAR number to analyze.
	`Descriptor_size`	Number of descriptor.
	`direction`	When 0 the direction is read. When 1 the direction is write.
	`Use_msi`	When set, the Root Port uses native PCI Express MSI to detect the DMA completion.
	`Use_eplast`	When set, the Root Port uses BFM shared memory polling to detect the DMA completion.
	`Bdt_msb`	BFM shared memory upper address value.
	`Bdt_lsb`	BFM shared memory lower address value.
	`Msi_number`	When `use_msi` is set, specifies the number of the MSI which is set by the `dma_set_msi` procedure.
	`Msi_traffic_class`	When `use_msi` is set, specifies the MSI traffic class which is set by the `dma_set_msi` procedure.
	`Multi_message_enable`	When `use_msi` is set, specifies the MSI traffic class which is set by the `dma_set_msi` procedure.

17.9.13.7 rc_mempoll Procedure

Use the `rc_mempoll` procedure to poll a given dword in a given BFM shared memory location.

Location	altpcietb_bfm_driver_rp.v	
Syntax	`rc_mempoll (rc_addr, rc_data, rc_mask)`	
Arguments	`rc_addr`	Address of the BFM shared memory that is being polled.
	`rc_data`	Expected data value of the that is being polled.
	`rc_mask`	Mask that is logically ANDed with the shared memory data before it is compared with `rc_data`.

17.9.13.8 msi_poll Procedure

The `msi_poll` procedure tracks MSI completion from the Endpoint.

Location	altpcietb_bfm_driver_rp.v	
Syntax	`msi_poll(max_number_of_msi,msi_address,msi_expected_dmawr,msi_expected_dmard,dma_write,dma_read)`	
Arguments	`max_number_of_msi`	Specifies the number of MSI interrupts to wait for.

continued...

Location	altpcietb_bfm_driver_rp.v	
	msi_address	The shared memory location to which the MSI messages will be written.
	msi_expected_dmawr	When dma_write is set, this specifies the expected MSI data value for the write DMA interrupts which is set by the dma_set_msi procedure.
	msi_expected_dmard	When the dma_read is set, this specifies the expected MSI data value for the read DMA interrupts which is set by the dma_set_msi procedure.
	Dma_write	When set, poll for MSI from the DMA write module.
	Dma_read	When set, poll for MSI from the DMA read module.

17.9.13.9 dma_set_msi Procedure

The dma_set_msi procedure sets PCI Express native MSI for the DMA read or the DMA write.

Location	altpcietb_bfm_driver_rp.v	
Syntax	dma_set_msi(bar_table, bar_num, bus_num, dev_num, fun_num, direction, msi_address, msi_data, msi_number, msi_traffic_class, multi_message_enable, msi_expected)	
Arguments	bar_table	Address of the Endpoint bar_table structure in BFM shared memory.
	bar_num	BAR number to analyze.
	Bus_num	Set configuration bus number.
	dev_num	Set configuration device number.
	Fun_num	Set configuration function number.
	Direction	When 0 the direction is read. When 1 the direction is write.
	msi_address	Specifies the location in shared memory where the MSI message data will be stored.
	msi_data	The 16-bit message data that will be stored when an MSI message is sent. The lower bits of the message data will be modified with the message number as per the PCI specifications.
	Msi_number	Returns the MSI number to be used for these interrupts.
	Msi_traffic_class	Returns the MSI traffic class value.
	Multi_message_enable	Returns the MSI multi message enable status.
	msi_expected	Returns the expected MSI data value, which is msi_data modified by the msi_number chosen.

17.9.13.10 find_mem_bar Procedure

The find_mem_bar procedure locates a BAR which satisfies a given memory space requirement.

Location	altpcietb_bfm_driver_rp.v	
Syntax	Find_mem_bar(bar_table,allowed_bars,min_log2_size, sel_bar)	
Arguments	bar_table	Address of the Endpoint bar_table structure in BFM shared memory

continued...

Location	altpcietb_bfm_driver_rp.v	
	allowed_bars	One hot 6 bits BAR selection
	min_log2_size	Number of bit required for the specified address space
	sel_bar	BAR number to use

17.9.13.11 dma_set_rclast Procedure

The dma_set_rclast procedure starts the DMA operation by writing to the Endpoint DMA register the value of the last descriptor to process (RCLast).

Location	altpcietb_bfm_driver_rp.v	
Syntax	Dma_set_rclast(bar_table, setup_bar, dt_direction, dt_rclast)	
Arguments	bar_table	Address of the Endpoint bar_table structure in BFM shared memory
	setup_bar	BAR number to use
	dt_direction	When 0 read, When 1 write
	dt_rclast	Last descriptor number

17.9.13.12 ebfm_display_verb Procedure

The ebfm_display_verb procedure calls the procedure ebfm_display when the global variable DISPLAY_ALL is set to 1.

Location	altpcietb_bfm_driver_chaining.v	
Syntax	ebfm_display_verb(msg_type, message)	
Arguments	msg_type	Message type for the message. Should be one of the constants defined in *BFM Log and Message Procedures*.
	message	The message string is limited to a maximum of 100 characters. Also, because Verilog HDL does not allow variable length strings, this routine strips off leading characters of 8'h00 before displaying the message.

Related Links

BFM Log and Message Procedures on page 210

17.10 Setting Up Simulation

Changing the simulation parameters reduces simulation time and provides greater visibility.

17.10.1 Changing Between Serial and PIPE Simulation

By default, the Intel testbench runs a serial simulation. You can change between serial and PIPE simulation by editing the top-level testbench file. For Endpoint designs, the top-level testbench file is *<working_dir>/<instantiation_name>*_tb/*<instantiation_name>*_tb/sim/*<instantiation_name>*_tb.v

The serial_sim_hwtcl and enable_pipe32_sim_hwtcl parameters control serial mode or PIPE simulation mode. To change to PIPE mode, change enable_pipe32_sim_hwtcl to 1'b1 and serial_sim_hwtcl to 1'b0.

Table 106. Controlling Serial and PIPE Simulations

Data Rates	Parameter Settings	
	serial_sim_hwtcl	enable_pipe32_sim_hwtcl
Serial simulation	1	0
PIPE simulation	0	1

17.10.2 Using the PIPE Interface for Gen1 and Gen2 Variants

Running the simulation in PIPE mode reduces simulation time and provides greater visibility.

Complete the following steps to simulate using the PIPE interface:

1. Change to your simulation directory, <*work_dir*>/<*variant*>/testbench/<*variant*>_tb/simulation
2. Open <*variant*>_tb.v.
3. Search for the string, serial_sim_hwtcl. Set the value of this parameter to 0 if it is 1.
4. Save <*variant*>_tb.v.

17.10.3 Viewing the Important PIPE Interface Signals

You can view the most important PIPE interface signals, txdata, txdatak, rxdata, and rxdatak at the following level of the design hierarchy: altpcie_<*device*>_hip_pipen1b|twentynm_hssi_<*gen*>_<*lanes*>_pcie_hip.

17.10.4 Disabling the Scrambler for Gen1 and Gen2 Simulations

The encoding scheme implemented by the scrambler applies a binary polynomial to the data stream to ensure enough data transitions between 0 and 1 to prevent clock drift. The data is decoded at the other end of the link by running the inverse polynomial.

Complete the following steps to disable the scrambler:

1. Open <*work_dir*>/<*variant*>/testbench/<*variant*>_tb/simulation/submodules/altpcie_tbed_<*dev*>_hwtcl.v.
2. Search for the string, test_in.
3. To disable the scrambler, set test_in[2] = 1.
4. Save altpcie_tbed_sv_hwtcl.v.

17.10.5 Disabling 8B/10B Encoding and Decoding for Gen1 and Gen2 Simulations

You can disable 8B/10B encoding and decoding to facilitate debugging.

For Gen1 and Gen2 variants, you can disable 8B/10B encoding and decoding by setting test_in[2] = 1 in **altpcietb_bfm_top_rp.v**.

18 Debugging

As you bring up your PCI Express system, you may face a number of issues related to FPGA configuration, link training, BIOS enumeration, data transfer, and so on. This chapter suggests some strategies to resolve the common issues that occur during hardware bring-up.

18.1 Simulation Fails To Progress Beyond Polling.Active State

If your PIPE simulation cycles between the Detect.Quiet, Detect.Active, and Polling.Active LTSSM states, the PIPE interface width may be incorrect.

Make the changes shown in the following table for the 32-bit PIPE interface.

Table 107. Changes for 32-Bit PIPE Interface

8-Bit PIPE Interface	32-Bit PIPE Interface
`output wire [7:0] pcie_a10_hip_0_hip_pipe_txdata0`	`output wire [31:0] pcie_a10_hip_0_hip_pipe_txdata0`
`input wire [7:0] pcie_a10_hip_0_hip_pipe_rxdata0`	`input wire [31:0] pcie_a10_hip_0_hip_pipe_rxdata0`
`output wire pcie_a10_simulation_inst_pcie_a10_hip_0_hip_pipe_txdatak0`	`output wire [3:0] pcie_a10_simulation_inst_pcie_a10_hip_0_hip_pipe_txdatak0`
`input wire pcie_a10_simulation_inst_pcie_a10_hip_0_hip_pipe_rxdatak0`	`input wire [3:0] pcie_a10_simulation_inst_pcie_a10_hip_0_hip_pipe_rxdatak0`

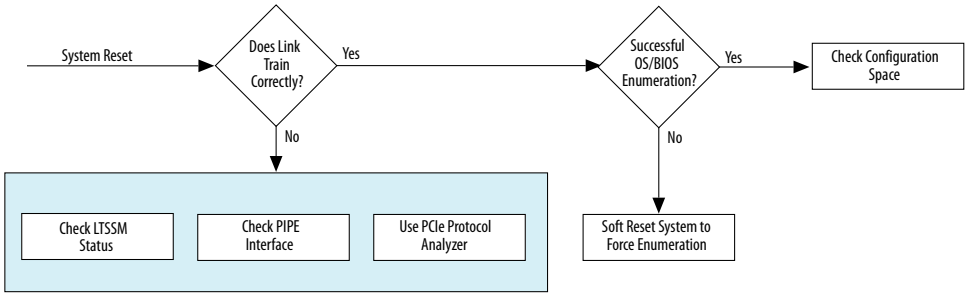
18.2 Hardware Bring-Up Issues

Typically, PCI Express hardware bring-up involves the following steps:

1. System reset
2. Link training
3. BIOS enumeration

The following sections describe how to debug the hardware bring-up flow. Intel recommends a systematic approach to diagnosing bring-up issues as illustrated in the following figure.

Figure 120. Debugging Link Training Issues



18.3 Link Training

The Physical Layer automatically performs link training and initialization without software intervention. This is a well-defined process to configure and initialize the device's Physical Layer and link so that PCIe packets can be transmitted. If you encounter link training issues, viewing the actual data in hardware should help you determine the root cause. You can use the following tools to provide hardware visibility:

- Signal Tap Embedded Logic Analyzer
- Third-party PCIe protocol analyzer

You can use Signal Tap Embedded Logic Analyzer to diagnose the LTSSM state transitions that are occurring on the PIPE interface. The `ltssmstate` bus encodes the status of LTSSM. The LTSSM state machine reflects the Physical Layer's progress through the link training process. For a complete description of the states these signals encode, refer to *Reset, Status, and Link Training Signals*. When link training completes successfully and the link is up, the LTSSM should remain stable in the L0 state. When link issues occur, you can monitor `ltssmstate` to determine the cause.

Related Links

Reset, Status, and Link Training Signals on page 82

18.3.1 Link Hangs in L0 State

There are many reasons that link may stop transmitting data. The following table lists some possible causes.

Table 108. Link Hangs in L0

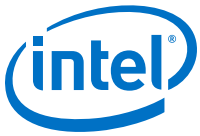
Possible Causes	Symptoms and Root Causes	Workarounds and Solutions
Avalon-ST signaling violates Avalon-ST protocol	Avalon-ST protocol violations include the following errors: • More than one `tx_st_sop` per `tx_st_eop`. • Two or more `tx_st_eop`'s without a corresponding `tx_st_sop`. • `rx_st_valid` is not asserted with `tx_st_sop` or `tx_st_eop`.	Add logic to detect situations where `tx_st_ready` remains deasserted for more than 100 cycles. Set post-triggering conditions to check for the Avalon-ST signaling of last two TLPs to verify correct `tx_st_sop` and `tx_st_eop` signaling.

continued...

Possible Causes	Symptoms and Root Causes	Workarounds and Solutions
	These errors are applicable to both simulation and hardware.	
Incorrect payload size	Determine if the length field of the last TLP transmitted by End Point is greater than the InitFC credit advertised by the link partner. For simulation, refer to the log file and simulation dump. For hardware, use a third-party logic analyzer trace to capture PCIe transactions.	If the payload is greater than the initFC credit advertised, you must either increase the InitFC of the posted request to be greater than the **max payload size** or reduce the payload size of the requested TLP to be less than the InitFC value.
Flow control credit overflows	Determine if the credit field associated with the current TLP type in the `tx_cred` bus is less than the requested credit value. When insufficient credits are available, the core waits for the link partner to release the correct credit type. Sufficient credits may be unavailable if the link partner increments credits more than expected, creating a situation where the Arria 10 Hard IP for PCI Express IP Core credit calculation is out-of-sync with its link partner.	Add logic to detect conditions where the `tx_st_ready` signal remains deasserted for more than 100 cycles. Set post-triggering conditions to check the value of the `tx_cred_*` and `tx_st_*` interfaces. Add a FIFO status signal to determine if the TXFIFO is full.
Malformed TLP is transmitted	Refer to the error log file to find the last good packet transmitted on the link. Correlate this packet with TLP sent on Avalon-ST interface. Determine if the last TLP sent has any of the following errors: • The actual payload sent does not match the length field. • The format and type fields are incorrectly specified. • TD field is asserted, indicating the presence of a TLP digest (ECRC), but the ECRC dword is not present at the end of TLP. • The payload crosses a 4KByte boundary.	Revise the Application Layer logic to correct the error condition.
Insufficient Posted credits released by Root Port	If a Memory Write TLP is transmitted with a payload greater than the **maximum payload size**, the Root Port may release an incorrect posted data credit to the Endpoint in simulation. As a result, the Endpoint does not have enough credits to send additional Memory Write Requests.	Make sure Application Layer sends Memory Write Requests with a payload less than or equal the value specified by the **maximum payload size**.
Missing completion packets or dropped packets	The RX Completion TLP might cause the RX FIFO to overflow. Make sure that the total outstanding read data of all pending Memory Read Requests is smaller than the allocated completion credits in RX buffer.	You must ensure that the data for all outstanding read requests does not exceed the completion credits in the RX buffer.

Related Links

- PIPE Interface Signals on page 102
- Avalon Interface Specifications



For information about the Avalon-ST interface protocol.

- PCI Express Base Specification 3.0

18.4 Use Third-Party PCIe Analyzer

A third-party logic analyzer for PCI Express records the traffic on the physical link and decodes traffic, saving you the trouble of translating the symbols yourself. A third-party logic analyzer can show the two-way traffic at different levels for different requirements. For high-level diagnostics, the analyzer shows the LTSSM flows for devices on both side of the link side-by-side. This display can help you see the link training handshake behavior and identify where the traffic gets stuck. A traffic analyzer can display the contents of packets so that you can verify the contents. For complete details, refer to the third-party documentation.

18.5 BIOS Enumeration Issues

Both FPGA programming (configuration) and the initialization of a PCIe link require time. Potentially, an Intel FPGA including a Hard IP block for PCI Express may not be ready when the OS/BIOS begins enumeration of the device tree. If the FPGA is not fully programmed when the OS/BIOS begins its enumeration, the OS does not include the Hard IP for PCI Express in its device map.

You can use either of the following two methods to eliminate this issue:

- You can perform a soft reset of the system to retain the FPGA programming while forcing the OS/BIOS to repeat its enumeration.
- You can use CvP to program the device.

A Transaction Layer Packet (TLP) Header Formats

The following figures show the header format for TLPs without a data payload.

For more information about the alignment of 3- and 4-dword headers refer to the related links below for *Data Alignment and Timing* for the Avalon-ST TX and RX Interfaces.

Figure 121. Memory Read Request, 32-Bit Addressing

Memory Read Request, 32-Bit Addressing

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	0	0	0	0	0	0	0	TC			0	0	0	0	TD	EP	Attr		0	0	Length									
Byte 4	Requester ID																Tag								Last BE				First BE			
Byte 8	Address[31:2]																														0	0
Byte 12	Reserved																															

Figure 122. Memory Read Request, Locked 32-Bit Addressing

Memory Read Request, Locked 32-Bit Addressing

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	0	0	0	0	0	1	0	TC			0	0	0	0	TD	EP	Attr		0	0	Length									
Byte 4	Requester ID																Tag								Last BE				First BE			
Byte 8	Address[31:2]																														0	0
Byte 12	Reserved																															

Figure 123. Memory Read Request, 64-Bit Addressing

Memory Read Request, 64-Bit Addressing

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	0	0	0	0	0	0	0	TC			0	0	0	0	TD	EP	Attr		0	0	Length									
Byte 4	Requester ID																Tag								Last BE				First BE			
Byte 8	Address[63:32]																															
Byte 12	Address[31:2]																														0	0

Figure 124. Memory Read Request, Locked 64-Bit Addressing

Memory Read Request, Locked 64-Bit Addressing

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	1	0	0	0	0	1	0	TC			0	0	0	0	T	EP	Attr		0	0	Length									
Byte 4	Requester ID																Tag								Last BE				First BE			
Byte 8	Address[63:32]																															
Byte 12	Address[31:2]																														0	0

Figure 125. Configuration Read Request Root Port (Type 1)

Configuration Read Request Root Port (Type 1)

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	TD	EP	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Byte 4	Requester ID																Tag								0	0	0	0	First BE			
Byte 8	Bus Number								Device No					Func			0	0	0	0	Ext Reg				Register No						0	0
Byte 12	Reserved																															

Figure 126. I/O Read Request

I/O Read Request

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	TD	EP	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Byte 4	Requester ID																Tag								0	0	0	0	First BE			
Byte 8	Address[31:2]																														0	0
Byte 12	Reserved																															

Figure 127. Message without Data

Message without Data

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	1	1	0	r 2	r 1	r 0	0	TC			0	0	0	0	TD	EP	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Byte 4	Requester ID																Tag								Message Code							
Byte 8	Vendor defined or all zeros																															
Byte 12	Vendor defined or all zeros																															

Note:

(1) Not supported in Avalon-MM.

Figure 128. Completion without Data

Completion without Data

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	0	0	1	0	1	0	0	TC			0	0	0	0	TD	EP	Attr		0	0	Length									
Byte 4	Completer ID																Status			B	Byte Count											
Byte 8	Requester ID																Tag								0	Lower Address						
Byte 12	Reserved																															

Figure 129. Completion Locked without Data

Completion Locked without Data

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	0	0	1	0	1	1	0	TC				0	0	0	0	TD	EP	Attr		0	0	Length								
Byte 4	Completer ID																Status			B	Byte Count											
Byte 8	Requester ID																Tag								0	Lower Address						
Byte 12	Reserved																															

Related Links

- Data Alignment and Timing for the 64-Bit Avalon-ST RX Interface on page 61
- Data Alignment and Timing for the 128-Bit Avalon-ST RX Interface on page 65
- Data Alignment and Timing for 256-Bit Avalon-ST RX Interface on page 68
- Data Alignment and Timing for the 64-Bit Avalon-ST TX Interface on page 73
- Data Alignment and Timing for the 128-Bit Avalon-ST TX Interface on page 76
- Data Alignment and Timing for the 256-Bit Avalon-ST TX Interface on page 79

A.1 TLP Packet Formats with Data Payload

Figure 130. Memory Write Request, 32-Bit Addressing

Memory Write Request, 32-Bit Addressing

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	1	0	0	0	0	0	0	0	TC				0	0	0	0	TD	EP	Attr		0	0	Length								
Byte 4	Requester ID																Tag								Last BE				First BE			
Byte 8	Address[31:2]																														0	0
Byte 12	Reserved																															

Figure 131. Memory Write Request, 64-Bit Addressing

Memory Write Request, 64-Bit Addressing

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	1	1	0	0	0	0	0	0	TC				0	0	0	0	TD	EP	Attr		0	0	Length								
Byte 4	Requester ID																Tag								Last BE				First BE			
Byte 8	Address[63:32]																															
Byte 12	Address[31:2]																														0	0

Figure 132. Configuration Write Request Root Port (Type 1)

Configuration Write Request Root Port (Type 1)

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	TD	EP	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Byte 4	Requester ID																Tag								0	0	0	0	First BE			
Byte 8	Bus Number								Device No								0	0	0	0	Ext Reg				Register No						0	0
Byte 12	Reserved																															

Figure 133. I/O Write Request

I/O Write Request

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	TD	EP	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Byte 4	Requester ID																Tag								0	0	0	0	First BE			
Byte 8	Address[31:2]																														0	0
Byte 12	Reserved																															

Figure 134. Completion with Data

Completion with Data

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	1	0	0	1	0	1	0	0	TC			0	0	0	0	TD	EP	Attr		0	0	Length									
Byte 4	Completer ID																Status			B	Byte Count											
Byte 8	Requester ID																Tag								0	Lower Address						
Byte 12	Reserved																															

Figure 135. Completion Locked with Data

Completion Locked with Data

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	1	0	0	1	0	1	1	0	TC			0	0	0	0	TD	EP	Attr		0	0	Length									
Byte 4	Completer ID																Status			B	Byte Count											
Byte 8	Requester ID																Tag								0	Lower Address						
Byte 12	Reserved																															

Figure 136. Message with Data

Message with Data

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	1	1	1	0	r2	r1	r0	0	TC			0	0	0	0	TD	EP	0	0	0	0	Length									
Byte 4	Requester ID																Tag								Message Code							
Byte 8	Vendor defined or all zeros for Slot Power Limit																															
Byte 12	Vendor defined or all zeros for Slots Power Limit																															

B Lane Initialization and Reversal

Connected components that include IP blocks for PCI Express need not support the same number of lanes. The ×4 variations support initialization and operation with components that have 1, 2, or 4 lanes. The ×8 variant supports initialization and operation with components that have 1, 2, 4, or 8 lanes.

Lane reversal permits the logical reversal of lane numbers for the ×1, ×2, ×4, and ×8 configurations. Lane reversal allows more flexibility in board layout, reducing the number of signals that must cross over each other when routing the PCB.

Table 109. Lane Assignments without Lane Reversal

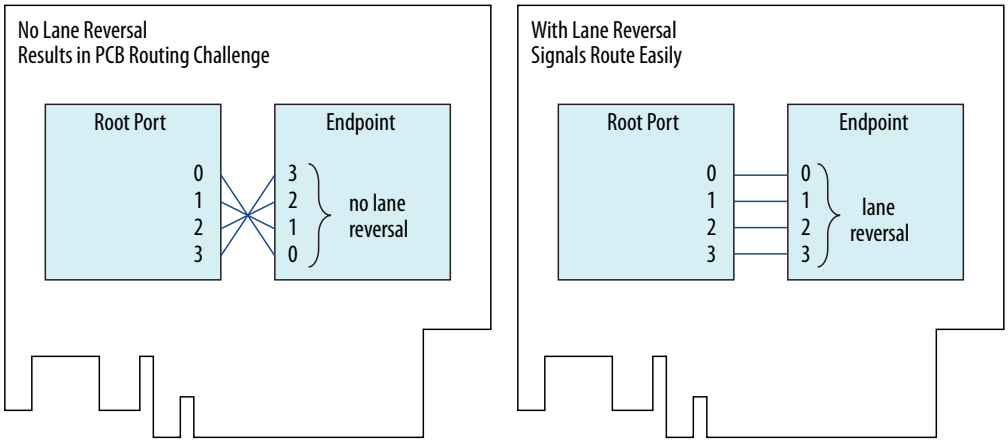
Lane Number	7	6	5	4	3	2	1	0
×8 IP core	7	6	5	4	3	2	1	0
×4 IP core	—	—	—	—	3	2	1	0
—	—	—	—	—	—	—	1	0
×1 IP core	—	—	—	—	—	—	—	0

Table 110. Lane Assignments with Lane Reversal

Core Config	8				4				1				
Slot Size	8	4	2	1	8	4	2	1	8	4	2	1	
Lane pairings	7:0,6:1,5:2, 4:3, 3:4,2:5, 1:6,0:7	3:4,2:5, 1:6,0:7	1:6, 0:7	0:7	7:0,6:1, 5:2,4:3	3:0,2:1, 1:2,0:3	3:0, 2:1	3:0	7:0	3:0	1:0	0:0	

Figure 137. Using Lane Reversal to Solve PCB Routing Problems

The following figure illustrates a PCI Express card with ×4 IP Root Port and a ×4 Endpoint on the top side of the PCB. Connecting the lanes without lane reversal creates routing problems. Using lane reversal solves the problem.



C Arria 10 Avalon-ST Interface for PCIe Solutions User Guide Archive

If an IP core version is not listed, the user guide for the previous IP core version applies.

IP Core Version	User Guide
16.1.1	Arria 10 Avalon-ST Interface for PCIe Solutions User Guide
16.1	Arria 10 Avalon-ST Interface for PCIe Solutions User Guide
16.0	Arria 10 Avalon-ST Interface for PCIe Solutions User Guide
15.1	Arria 10 Avalon-ST Interface for PCIe Solutions User Guide
15.0	Arria 10 Avalon-ST Interface for PCIe Solutions User Guide
14.1	Arria 10 Avalon-ST Interface for PCIe Solutions User Guide

D Revision History

D.1 Revision History for the Avalon-ST Interface

Date	Version	Changes Made
2017.05.26		Made the following changes to the user guide: • Added note that starting with the Quartus Prime Pro Edition Software, version 17.0, the QSF assignments in the following answer *What assignments do I need for a PCIe Gen1, Gen2 or Gen3 design that targets an Arria 10 ES2, ES3 or production device?* are already included in the design.
2017.05.12	17.0	Made the following changes the IP core: • Added option soft DFE Controller IP on the **PHY** tab of the parameter editor to improve BER margin. The default for this option is off because it is typically not required. Short reflective links may benefit from this soft DFE controller IP. This parameter is available only for Gen3 configurations. It is not supported when CvP or autonomous modes are enabled. Made the following changes to the user guide: • Updated *PCI Express Gen3 Bank Usage Restrictions* status. These restrictions affect all Aria 10 ES and production devices. • Added statement that Arria 10 devices do not support the **Create timing and resource estimates for third-party EDA synthesis tools** option on the **Generate ➤ Generate HDL** menu. • Corrected default values for the *Uncorrectable Internal Error Mask Register* and *Correctable Internal Error Mask Register* registers. • Corrected *Feature Comparison for all Hard IP for PCI Express IP Cores* table. Out-of-order Completions are not supported transparently for the Avalon-MM with DMA interface. • Revised discussion of Application Layer Interrupt Handler Module to include legacy interrupt generation. • Corrected minor errors and typos.
2017.03.15	16.1.1	Made the following changes: • Restored *Configuration Space Register Access* topic which was inadvertently removed form previous versions. • Improved definitions of `tx_cred_data_fc[11:0]`, `tx_cred_fc_sel[1:0]` and `tx_cred_fdr_fc[7:0]`. • Added missing signal definition for `tx_cred_fc_sel`. • Added statement that Arria 10 devices do not support the **Create timing and resource estimates for third-party EDA synthesis tools** option on the **Generate ➤ Generate HDL** menu. • Rebranded as Intel.
2016.10.31	16.1	Made the following changes to the IP core: • Changed timing models support to final for most Arria 10 device packages. Exceptions include some military and automotive speed grades with extended temperature ranges. • Added parameter to select the requested preset for Phase2 and Phase3 far-end TX equalization.

continued...

Date	Version	Changes Made
		Made the following changes to the user guide: • Corrected the number of tags supported in the *Feature Comparison for all Hard IP for PCI Express IP Cores* table. • Removed recommendations about connecting `pin_perst`. These recommendations do not apply to Arria 10 devices. • Added PCIe bifurcation to the *Feature Comparison for all Hard IP for PCI Express IP Cores* table. PCI bifurcation is not supported. • Corrected description of `tl_cfg*` bus. Provided sample RTL code to show how sample `tl_cfg_ctl`.Corrected *tl_cfg_ctl Timing* diagram. • Changed the recommended value of `test_in[31:0]` from 0xa8 to 0x188. • Added instructions for turning on autonomous mode in the Quartus Prime software. • Added -3 to recommended speed grades for the 125 MHz interface.
2016.05.02	16.0	Made the following changes: • The PIO Design Examples included in the *Quick Start Guide* now support 64- and 128-bit interfaces to the Application Layer. (The 15.1 release supported only a 256-bit interface to the Application Layer interface.) • The *Quick Start Guide* no longer supports the DMA design example. • Added support for OpenCore Plus IP evaluation in the Quartus Prime Pro Edition software. • Added automatic generation of basic SignalTap Logic Analyzer files to facilitate debugging. • Added figure for TX 3-dword header with qword aligned data. • Added Gen3 x2 128-bit interface with 125 MHz clock to the `coreclkout_hip` *Application Layer Clock Frequency for All Combinations of Link Width, Data Rate and Application Layer Interface Widths* table. • In the Getting Started with the Hard IP for PCI Express chapter, changed the instructions to use specify the 10AX115S2F45I1SG device which is used on the Arria 10 GX FPGA Development Kit - Production (not ES2) Edition. • Added statement that the testbench can only simulate a single Endpoint or Root Port at a time. • Enhanced statements covering the deficiencies of the Intel-provided testbench. • Added simulation support for Gen3 PIPE mode using the ModelSim, VCS, and NCSim simulators. • Added definition for `rxfc_cplbuf_ovf`. • Added **Vendor Specific Extended Capability (VSEC) Revision** and **User Device or Board Type ID register from the Vendor Specific Extended Capability:** to the **VSEC** tab of the component GUI. • Updated figures in *Physical Layout of Hard IP in Arria 10 Devices* to include more detail about transceiver banks and channel restrictions. • Added transceiver bank usage placement restrictions for Gen3 devices. • Removed support for -3 speed grade devices. • Added transceiver bank usage placement restrictions for Gen3 devices. • Added -3 to recommended speed grades with qualifying statement. • Corrected minor errors and typos.
2015.11.30	15.1	Made the following changes: • Added definition for `tx_fifo_empty` signal. • Added figure illustrating data alignment for the TX 3-dword header with qword aligned address. • Added *TLP Support Comparison for all Hard IP for PCI Express IP Cores* in *Datasheet* chapter. • Added new topic on *Autonomous Mode* in which the Hard IP for PCI Express begins operation when the periphery configuration completes.
2015.11.02	15.1	Made the following changes:

continued...

Date	Version	Changes Made
		• Added auto generation of example designs for Endpoints that use the parameters you specify. Generation creates both simulation and hardware testbenches that you can download to the Arria 10 FPGA Development Kit ES2 Edition. This new feature is described in the *Arria 10 Avalon-ST Quick Start Guide* chapter of this user guide. • Added latency between `tx_cred_fc_sel` and `tx_cred_data_fc` and `tx_cred_hdr_fc` to the signal definitions. • Corrected instructions for changing between a serial and PIPE simulation. • Updated definitions of `rxsynchd0[1:0]` and `rxblkst0` to say these signals can be grounded for Gen1 and Gen2 operation. • Improved the definition of `npor`. • Added note saying that the Hard IP for PCI Express supports autonomous mode when CvP is enabled. • In *Transaction Layer Routing Rules*, added Type 1 Message TLPs are also passed to the Application Layer. • Enhanced the definition of `rx_st_mask`. • Added x2 to the *Lane Assignments without Lane Reversal* table. • Removed signal definition for `rx_st_be`. This signal is not supported for Arria 10 devices. • Changed the `app_msi_req` signal to X (don't care) in cycles 4 and 5 of the timing diagram, *MSI Interrupt Signals Timing*. • Removed **Legacy Endpoint** option for **Port type** parameters. The Legacy Endpoint is no longer supported for Arria 10 devices. • Revised discussion on possible conflict between LMI writes and Host writes to the Configuration Space. • Removed *Getting Started with the Configuration Space Bypass Model Qsys Example Design* chapter. This example design is no longer supported. • Removed invalid warning about missing resets when this IP core is instantiated as a separate component from the Quartus Prime IP Catalog. • Corrected *Avalon-ST Hard IP for PCI Express Top-Level Signals* figure and missing signal definitions.
2015.06.05	15.0	Added note in *Physical Layout of Hard IP in Arria 10 Devices* to explain Arria 10 design constraint that requires that if the lower HIP on one side of the device is configured with a Gen3 x4 or Gen3 x8 IP core, and the upper HIP on the same side of the device is also configured with a Gen3 IP core, then the upper HIP must be configured with a x4 or x8 IP core.
2015.05.04	15.0	Made the following changes to the Arria 10 user guide: • Added to description of *Data Link Layer link active* bit. It is only available for Root Ports. It is always 0 for Endpoints. • Corrected link to *Arria 10 Avalon-MM DMA Interface for PCIe Solutions User Guide*. • Added **Enable Altera Debug Master Endpoint (ADME)** parameter to support optional Native PHY register programming with the Altera System Console. • Added information about the custom example designs. This feature is available for this IP core starting in the IP core release 14.1. • • Enhanced descriptions of channel placement, added fPLL placement for Gen1 and Gen2 data rates, and added master CGB location, in *Physical Layout of Hard IP In Arria 10 Devices*. • Added column for Avalon-ST Interface with SR-IOV variations in Feature Comparison for all Hard IP for PCI Express IP Cores table in *Features* section. Moved supported TLPs information to separate table. Updated information in tables. • Removed Migration and TLP Format appendices, and added new appendix *Frequently Asked Questions*. • Corrected LMI Write figure in *LMI Signals*. • Corrected *MSI-X Interrupt Components* figure in *Implementing MSI-X Interrupts*.

continued...

Date	Version	Changes Made
		• Corrected width of `rx_st_sop` and `rx_st_eop` to 1 or two bits. If you turn on **Enable multiple packets per cycle** these signals have two bits; otherwise, they have one bit each. Refer to *Avalon-ST RX Interface*. • Removed non-existent signals `rx_st_bar1` and `rx_st_bar2`. If you turn on **Enable multiple packets per cycle**, the IP core still has only a single `rx_st_bar[7:0]` signal. Do not use this signal if you turn on **Enable multiple packets per cycle**. Refer to i*Avalon-ST RX Component Specific Signals*. • Updated DUT module name in testbench and example design figures. • Reorganized sections in i*Debugging* and *nik1410565029488*. • Updated information in *SDC Timing Constraints* . • Fixed minor errors and typos.
2014.12.15	14.1	Made the following changes to the user guide: • Added simulation log file, `altpcie_monitor_<dev>_dlhip_tlp_file_log.log` in your simulation directory. Generation of the log file requires the following simulation file, `<install_dir>altera/altera_pcie/altera_pcie_<dev>_hip/altpcie_monitor_<dev>_dlhip_sim.sv`, that was not present in earlier releases of the Quartus II software. • Changed device part number for *Getting Started* chapter to **10AX115R2F40I2LG**. • Added statement that the bottom left hard IP block includes the CvP functionality for flip chip packages. For other package types, the CvP functionality is in the bottom right block. • Removed 125 MHz clock as optional `refclk` frequency in Arria 10 devices. Arria 10 devices support an 100 MHz reference clock as specified by the *PCI Express Base Specification, Rev 3.0*. • Corrected bit definitions for `CvP Status` register. • Updated definition of CVP_NUMCLKS in the `CvP Mode Control` register. • Added definitions for `test_in[2]`, `test_in[6]` and `test_in[7]`. • Enhanced instructions *Compiling the Design* to include steps necessary to download to Altera development kits.
2014.08.18	14.0a10	Made the following changes to the Arria 10 Hard IP for PCI Express: • Changed the PIPE interface to 32 bits for all data rates. This change requires you to recompile your 13.1 variant in 14.0. • Made fPLL available as the TX PLL for all data rates. This change allows you to use the ATX PLLs for higher data rate protocols if necessary. Made the following changes to the user guide: • Added statement that the bottom left hard IP block includes the CvP functionality for flip chip packages. For other package types, the CvP functionality is in the bottom right block.
2014.06.30	14.0	Added the following new features to the Arria 10 Hard IP for PCI Express: • Added parameters to enable 256 completion tags with completion tag checking performed in Application Layer. • Added simulation log file, `altpcie_monitor_sv_dlhip_tlp_file_log.log`, that is automatically generated in your simulation directory. To simulation in the Quartus II 14.0 software release, you must regenerate your IP core to create the supporting monitor file the generates `altpcie_monitor_sv_dlhip_tlp_file_log.log`. Refer to *Understanding Simulation Dump File Generation* for details. • Added support for new parameter,**User ID register from the Vendor Specific Extended Capability**, for Endpoints. • Added parameter to create a reset pulse at power-up when the soft reset controller is enabled. • Simulation support for Phase 2 and Phase 3 equalization when requested by third-party BFM.

continued...

Date	Version	Changes Made
		• Increased size of `lmi_addr` to 15 bits. • Changed the directory structure for generated files. Refer to *Files Generated for Intel FPGA IP Cores Targeting Arria 10* for more information. • In the *Getting Started with the Arria 10 Hard IP for PCI Express* chapter, changed the recommended device to 10AX115R2F40I2LG (Advanced). Made the following changes to the user guide: • Added *Next Steps in Creating a Design for PCI Express* to *Datasheet* chapter. • Corrected frequency range for `hip_reconfig_clk`. It should be 100-125 MHz. • Corrected **Maximum payload size** values listed in *Reconfigurable Read-Only Registers* table. The maximum size is 2048 bytes. • Enhanced definition of Device ID and Sub-system Vendor ID to say that these registers are only valid in the Type 0 (Endpoint) Configuration Space. • Changed the default reset controller settings. By default Gen1 devices use the Hard Reset Controller. Gen2 and Gen3 devices use the Soft Reset Controller. • Corrected frequencies of `pclk` in *Reset and Clocks* chapter. • Removed `txdatavalid0` signal from the PIPE interface. This signal is not available. • Removed references to the MegaWizard® Plug-In Manager. In 14.0 the IP Parameter Editor Powered by Qsys has replaced the MegaWizard Plug-In Manager. • Made the following changes to the timing diagram, *Hard IP Reconfiguration Bus Timing of Read-Only Registers*: — Added `hip_reconfig_rst_n`. — Changed timing of `avmm_rdata[15:0]`. Valid data returns 4 cycles after `avmm_rd`. • Added link to a Knowledge Base Solution that shows how to observe the `test_in` bus for debugging. • Removed optional 125 MHz reference clock frequency. This option has not been tested extensively in hardware. • Corrected channel placement diagrams for Gen3 x2 and Gen3 x4. The CMU PLL should be shown in the Channel 4 location. For Gen3 x2, the second data channel is Ch1. For Gen3 x4, the data channels are Ch0 - Ch3. • Corrected figure showing physical placement of PCIe Hard IP modules for Arria V GZ devices. • Added definition for `test_in[6]` and link to Knowledge Base Solution on observing the PIPE interface signals on the `test_out` bus. • Removed references to Gen2 x1 62.5 MHz configuration. This configuration is not supported. • Removed statement that Gen1 and Gen2 designs do not require transceiver reconfiguration. Gen1 and Gen2 designs may require transceiver reconfiguration to improve signal quality. • Removed `reconfig_busy` port from connect between PHY IP Core for PCI Express and the Transceiver Reconfiguration Controller in the *Altera Transceiver Reconfiguration Controller Connectivity* figure. The Transceiver Reconfiguration Controller drives `reconfig_busy` port to the Altera PCIe Reconfig Driver. • Removed soft reset controller `.sdc` constraints from the `<install_dir>/ip/altera/altera_pcie/altera_pcie_hip_ast_ed/altpcied_<dev>.sdc` example. These constraints are now in a separate file in the `synthesis/submodules` directory. • Updated *Power Supply Voltage Requirements* table. • For Arria 10 devices, updated *Physical Placement of the Arria 10 Hard IP for PCIe IP and Channels* to show GT devices instead of GX devices. • For Arria 10 devices, corrected frequency of `hip_reconfig_lck`. I should be 125 MHz.

continued...

Date	Version	Changes Made
2013.12.20	13.1	Made the following changes: • Divided user guide into 3 separate documents by interface type. • Added *Design Implementation* chapter. • In the *Debugging* chapter, removed section explaining how to turn off the scrambler for Gen3 because it does not work. • In the *Debugging* chapter, corrected filename that you must change to reduce counter values in simulation. • In *Getting Started with the Avalon-MM Hard IP for PCI Express* chapter, corrected connects for the Transceiver Reconfiguration Controller IP Core reset signal, alt_xcvr_reconfig_0 `mgmt_rst_reset`. This reset input connects to clk_0 `clk_reset`. • In *Transaction Layer Routing Rules* and *Programming Model for Avalon-MM Root Port* added the fact that Type 0 Configuration Requests sent to the Root Port are not filtered by the device number. Application Layer software must filter out requests for device number greater than 0. • Added illustration showing the location of the Hard IP Cores in the Arria 10 devices. • Added limitation for `rxm_irq_<n>[<m>:0]`when interrupts are received on consecutive cycles. • Corrected description of `cfg_prm_cmr`. It is the Base/Primary Command register for the PCI Configuration Space. • Revised channel placement illustrations.
2013.05.06	13.0	• Added support for Configuration Space Bypass Mode, allowing you to design a custom Configuration Space and support multiple functions • Added preliminary support for a Avalon-MM 256-Bit Hard IP for PCI Express that is capable of running at the Gen3 ×8 data rate. This new IP Core. Refer to the *Avalon-MM* 256-Bit Hard IP for PCI Express User Guide for more information. • Added Gen3 PIPE simulation support. • Added support for 64-bit address in the Avalon-MM Hard IP for PCI Express IP Core, making address translation unnecessary • Added instructions for running the Single Dword variant. • Timing models are now final. • Updated the definition of `refclk` to include constraints when CvP is enabled. • Added section covering clock connectivity for reconfiguration when CvP is enabled. • Corrected access field in Root Port TLP Data registers. • Added Getting Started chapter for Configuration Space Bypass mode. • Added signal and register descriptions for the Gen3 PIPE simulation. • Added 64-bit addressing for the Avalon-MM IP Cores for PCI Express. • Changed descriptions of `rx_st_err[1:0]`, `tx_st_err[1:0]`, `rx_st_valid[1:0]`, and `tx_st_valid[1:0]` buses. Bit 1 is not used. • Corrected definitions of RP_RXCPL_STATUS.SOP and RP_RXCPL_STATUS.EOP bits. SOP is 0x2010, bit[0] and EOP is 0x2010, bit[1]. • Improved explanation of relaxed ordering of transactions and provided examples. • Revised discussion of Transceiver Reconfiguration Controller IP Core. Offset cancellation is not required for Gen1 or Gen2 operation.
2011.07.30	11.01	Corrected typographical errors.
2011.05.06	11.0	First release.