Arria V GZ Hard IP for PCI Express

User Guide

101 Innovation Drive
San Jose, CA 95134
www.altera.com

UG-01127-1.1

Document last updated for Altera Complete Design Suite version: 13.0
Document publication date: May 2013

Feedback Subscribe

Contents

1. Arria V GZ Datasheet

This document describes both the Altera® Hard IP for PCI Express® and Avalon®-MM Arria V GZ Hard IP for PCI Express MegaCore functions. PCI Express is a high-performance interconnect protocol for use in a variety of applications including network adapters, storage area networks, embedded controllers, graphic accelerator boards, and audio-video products. The PCI Express protocol is software backwards-compatible with the earlier PCI and PCI-X protocols, but is significantly different from its predecessors. It is a packet-based, serial, point-to-point interconnect between two devices. The performance is scalable based on the number of lanes and the generation that is implemented. Altera offers a configurable hard IP block in Arria V GZ devices for both Endpoints and Root Ports that is compliant with *PCI Express Base Specification 3.0*. Using a configurable hard IP block, rather than programmable logic, saves significant FPGA resources. The hard IP block is available in ×1, ×2, ×4, and ×8 configurations. Table 1–1 shows the aggregate bandwidth of a PCI Express link for Gen1, Gen2, and Gen3 for 1, 2, 4, and 8 lanes. The protocol specifies 2.5 giga-transfers per second for Gen1, 5 giga-transfers per second for Gen2, and 8.0 giga-transfers per second for Gen3. Table 1–1 provides bandwidths for a single transmit (TX) or receive (RX) channel, so that the numbers in Table 1–1 double for duplex operation. 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 less than 1%.

Table 1–1. PCI Express Data Throughput

	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 *PCI Express High Performance Reference Design* for more information about calculating bandwidth for the hard IP implementation of PCI Express in many Altera FPGAs.

Features

The Arria V GZ Hard IP for PCI Express and the Avalon-MM Arria V GZ Hard IP for PCI Express IP cores support the following key features:

- Complete protocol stack including the Transaction, Data Link, and Physical Layers implemented as hard IP.
- Feature rich:
  - Support for ×1, ×2, ×4, and ×8 configurations with Gen1, Gen2, or Gen3 lane

rates for Root Ports and Endpoints

- Dedicated 16 KByte receive buffer
- Dedicated hard reset controller
- Optional support for Configuration via Protocol (CvP) using the PCIe link allowing the I/O and core bitstreams to be stored separately.
- Qsys support using the Avalon Streaming (Avalon-ST) or Avalon Memory-Mapped (Avalon-MM) interface.
- 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:
  - Easy parameterization.
  - Substantial on-chip resource savings and guaranteed timing closure.
  - Easy adoption with no license requirement.
  - Example designs to get started.
- New features in the 13.0 release:
  - Support for Configuration Space Bypass Mode, allowing you to design a custom Configuration Space and support multiple functions
  - 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 embeds a DMA engine with a 256-bit datapath. Refer to the *Avalon-MM 256-Bit Hard IP for PCI Express User Guide* for more information.
  - Support for 64-bit address in the Avalon-MM Hard IP for PCI Express IP Core, making address translation unnecessary
  - Support for Gen3 PIPE simulation

The Arria V GZ Hard IP for PCI Express offers different features for variants that use the Avalon-ST and Avalon-MM interfaces to the Application Layer. The Avalon-MM 256-Bit Hard IP for PCI Express IP core includes Read DMA and Write DMA modules. These modules combined with a Descriptor Controller IP Core available in the Qsys Component Library to implement a high performance DMA engine. Consequently, a host processor can program the DMA engine to transfer data between a Root Port or Root Complex and the Application Layer Avalon-MM slave components. For both the Avalon-MM Hard IP for PCI Express and Avalon-MM 256-Bit Hard IP for PCI Express IP Cores, a PCI Express to Avalon-MM bridge translates the PCI Express read, write, and completion Transaction Layer Packets (TLPs) into standard Avalon-MM read and write commands typically used by master and slave interfaces. This PCI Express to Avalon-MM bridge also translates Avalon-MM read, write and read data commands to PCI Express read, write and completion TLPs. Table 1–2 lists the features available for the variants.

Table 1–2. Hard IP for PCI Express Features (Part 1 of 2)

Feature	Avalon-ST Interface	Avalon-MM Interface	Avalon-MM 256-Bit Interface (1)
MegaCore License	Free	Free	Free
Native Endpoint	Supported	Supported	Supported
Legacy Endpoint (2)	Supported	Not Supported	Not 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 (3)	×1, ×2, ×4, ×8	×1, ×2, ×4	×4, ×8
MegaWizard Plug-In Manager design flow	Supported	Not supported	Not supported
Qsys design flow	Supported	Supported	Supported
64-bit Application Layer interface	Supported	Supported	Not supported
128-bit Application Layer interface	Supported	Supported	Not supported
256-bit Application Layer interface	Supported	Supported	Supported
Transaction Layer Packet type (TLP) (3)	■ Memory Read Request ■ Memory Read Request-Locked ■ Memory Write Request ■ I/O Read Request ■ I/O Write Request ■ Configuration Read Request (Root Port) ■ Configuration Write Request (Root Port) ■ Message Request ■ Message Request with Data Payload ■ Completion without Data ■ Completion with Data ■ Completion for Locked Read without Data	■ Memory Read Request ■ Memory Write Request ■ I/O Read Request ■ I/O Write Request ■ Configuration Read Request (Root Port) ■ Configuration Write Request (Root Port) ■ Message Request ■ Message Request with Data Payload ■ Completion without Data ■ Completion with Data ■ Memory Read Request (single dword) ■ Memory Write Request (single dword)	■ Memory Read Request ■ Memory Write Request ■ Message Request ■ Message Request with Data Payload ■ Completion without Data ■ Completion with Data
Payload size	128–2048 bytes	128–256 bytes	128, 256, 512 bytes
Number of tags supported for non-posted requests	32 or 64	8	16

Table 1–2. Hard IP for PCI Express Features (Part 2 of 2)

Feature	Avalon-ST Interface	Avalon-MM Interface	Avalon-MM 256-Bit Interface (1)
62.5 MHz clock	Supported	Supported	Not Supported
Out-of-order completions (transparent to the Application Layer)	Not supported	Supported	Supported
Requests that cross 4 KByte address boundary (transparent to the Application Layer)	Not supported	Supported	Supported
Polarity Inversion of PIPE interface signals	Supported	Supported	Supported
ECRC forwarding on RX and TX	Supported	Not supported	Not supported
Number of MSI requests	16	1, 2, 4, 8, or 16	1, 2, 4, 8, or 16
MSI-X	Supported	Supported	Supported
Multiple MSI, MSI-X, and INTx	Not Supported	Supported	Supported
Legacy interrupts	Supported	Supported	Supported
Expansion ROM	Supported	Not supported	Not supported

Notes to Table 1–2

(1) Refer to the *Avalon-MM 256-Bit Hard IP for PCI Express User Guide* for more information.

(2) Not recommended for new designs.

(3) Refer to Appendix A, Transaction Layer Packet (TLP) Header Formats for the layout of TLP headers.

This document describes both the Arria V GZ Hard IP for PCI Express which includes an Avalon-ST interface to the Application Layer and the Avalon-MM Arria V GZ Hard IP for PCI Express IP Cores. In the interest of brevity, the remainder of this document frequently does not list both product names.

The purpose of the *Arria V GZ Hard IP for PCI Express User Guide* is to explain how to use the Arria V GZ Hard IP for PCI Express and not to explain the PCI Express protocol. Although there is inevitable overlap between these two purposes, this document should be used in conjunction with an understanding of the *PCI Express Base Specification 3.0*

Release Information

Table 1–3 provides information about this release of the PCI Express Compiler.

Table 1–3. PCI Express Compiler Release Information

Item	Description
Version	13.0
Release Date	May 2013
Ordering Codes	No ordering code is required

Table 1–3. PCI Express Compiler Release Information

Item	Description
Product IDs	There are no encrypted files for the Arria V GZ 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	

Device Family Support

Table 1–4 shows the level of support offered by the Arria V GZ Hard IP for PCI Express.

Table 1–4. Device Family Support

Device Family	Support
Arria V GZ	Final. The IP core is verified with final timing models. The IP core meets all functional and timing requirements for the device family and can be used in production designs.
Other device families	Refer to the following user guides for other device families: ■ *IP Compiler for PCI Express User Guide* ■ *Arria V Hard IP for PCI Express User Guide* ■ *Stratix V Hard IP for PCI Express User Guide* ■ *Cyclone V Hard IP for PCI Express User Guide*

Configurations

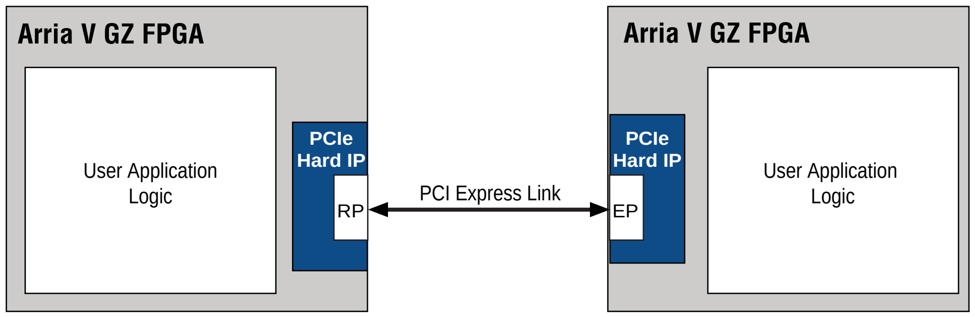
The Arria V GZ Hard IP for PCI Express includes a full hard IP implementation of the PCI Express stack including the following layers:

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

Optimized for Altera devices, the Arria V GZ Hard IP for PCI Express supports all memory, I/O, configuration, and message transactions. It has a highly optimized Application Layer interface to achieve maximum effective throughput. You can customize the Hard IP to meet your design requirements using either the MegaWizard Plug-In Manager or the Qsys design flow. When configured as an Endpoint, the Avalon-MM Arria V GZ Hard IP for PCI Express supports memory read and write requests and completions with or without data. In Root Port mode, the core also supports configuration reads and writes, message transactions, legacy interrupts, and single dword reads and writes.

Figure 1–1 shows a PCI Express link between two Arria V GZ FPGAs. One is configured as a Root Port and the other as an Endpoint.

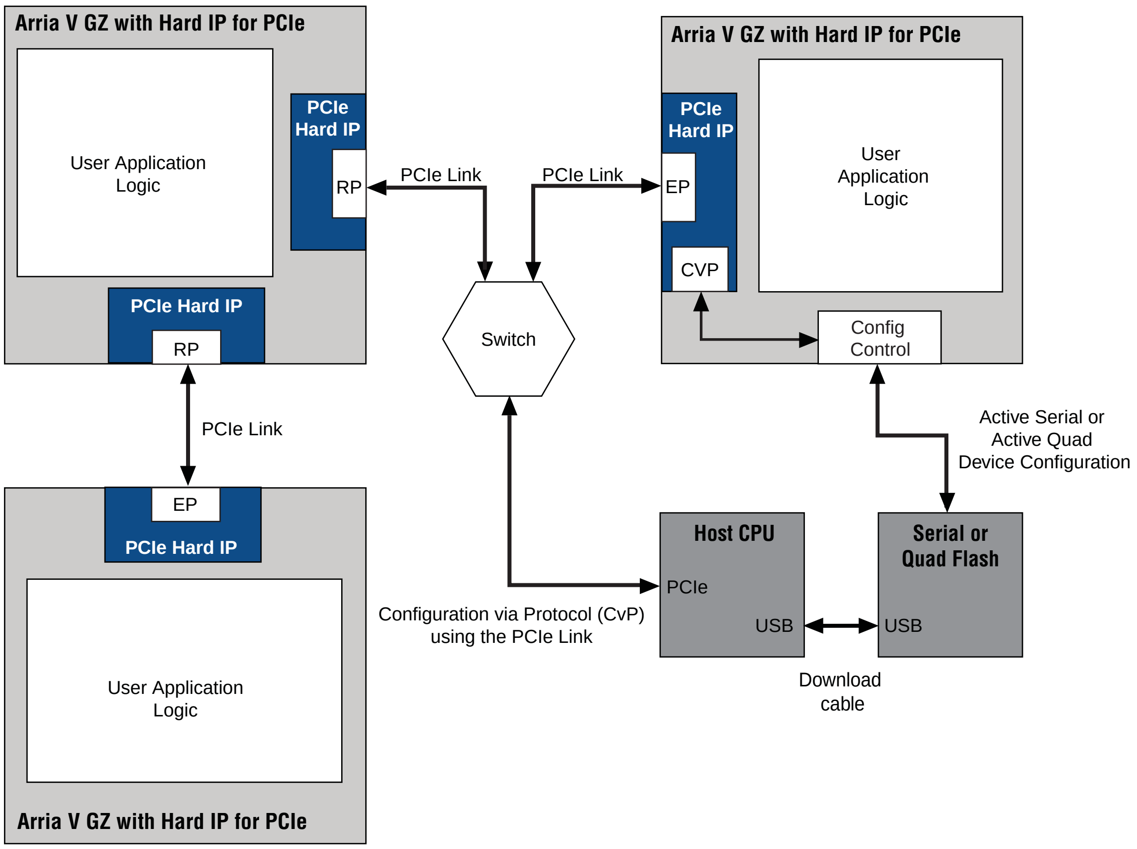
Figure 1–1. PCI Express Application with a Single Root Port and Endpoint



illustrates a Arria V GZ design that 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. For more information about configuration over a PCI Express link, refer to "Configuration via Protocol (CvP)" on page 12–1.

Figure 1–2. PCI Express Application Including Arria V GZ using Configuration via Protocol



Debug Features

The Arria V GZ Hard IP for PCI Express includes debug features that allow observation and control of the Hard IP for faster debugging of system-level problems. For more information about debugging refer to Chapter 18, Debugging.

IP Core Verification

To ensure compliance with the PCI Express specification, Altera performs extensive validation of the Arria V GZ Hard IP Core for PCI Express.

The simulation environment uses multiple testbenches that consist of industry-standard bus functional models (BFMs) driving the PCI Express link interface.

Altera performs the following tests in the simulation environment:

- Directed and pseudo random stimuli are applied to test the Application Layer interface, Configuration Space, and all types and sizes of TLPs.
- Error injection tests that 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

Compatibility Testing Environment

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

Performance and Resource Utilization

Because the IP core is implemented in hardened logic, it uses less than 1% of Arria V GZ resources.

The Avalon-MM Arria V GZ Hard IP for PCI Express implements the Avalon-MM bridge in soft logic. Table 1–5 shows the typical expected device resource utilization for selected configurations of the Avalon-MM Arria V GZ Hard IP for PCI Express using the current version of the Quartus II software targeting a Arria V GZ (**5AGZME5K2F40C3**) device. With the exception of M20K memory blocks, the numbers of ALMs and logic registers in the following tables are rounded up to the nearest 50. Resource utilization numbers reflect changes to the resource utilization reporting starting in the Quartus II software v12.1 release 28 nm device families and upcoming device families.

For information about Quartus II resource utilization reporting, refer to *Fitter Resources Reports* in the Quartus II Help.

Table 1–5. Performance and Resource Utilization Avalon-MM Hard IP for PCI Express

Data Rate or Interface Width	ALMs	Memory M20K	Logic Registers
Avalon-MM Bridge			
Gen1 ×4	1100	17	1500
Gen2 ×8	1900	25	2900
Avalon-MM Interface– Gen3 x8 256-Bit DMA			
64	1100	14	1650
128	1750	19	2600
Avalon-MM Interface–Burst Capable Completer Only			
64	650	8	1000
128	1400	12	2400
Avalon-MM–Completer Only Single DWord			
64	250	0	350

Soft calibration of the transceiver module requires additional logic. The amount of logic required depends upon the configuration.

Recommended Speed Grades

Table 1–6 and Table 1–7 list the recommended speed grades for the supported interface widths, link widths, and Application Layer clock frequencies when using the Avalon-ST or Avalon-MM interfaces, respectively. When the Application Layer clock frequency is 250 MHz, Altera recommends setting the Quartus II Analysis & Synthesis Settings **Optimization Technique** to **Speed**.

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 II Handbook*.

Table 1–6. Arria V GZ Recommended Speed Grades for All Avalon-ST Widths and Frequencies

Link Rate	Link Width	Interface Width	Application Clock Frequency (MHz)	Recommended Speed Grades
Gen1	×1	64 bits	62.5 (1),125	–1, –2, –3, –4
	×2	64 bits	125	–1, –2, –3, –4
	×4	64 bits	125	–1, –2, –3, –4
	×8	64 bits	250	–1, –2, –3 (2)
	×8	128 Bits	125	–1, –2, –3, –4
Gen2	×1	64 bits	62.5 (1), 125	–1, –2, –3, –4
	×2	64 bits	125	–1, –2, –3, –4
	×4	64 bits	250	–1, –2, –3 (2)
	×4	128 bits	125	–1, –2, –3, –4
	×8	128 bits	250	–1, –2, –3 (2)
	×8	256 bits	125	–1, –2, –3, –4
Gen3	×1	64 bits	125	–1, –2, –3, –4
	×2	64 bits	250	–1, –2, –3, –4
	×2	128 bits	125	–1, –2, –3, –4
	×4	128 bits	250	–1, –2, –3 (2)
	×4	256 bits	125	–1, –2, –3,–4
	×8	256 bits	250	–1, –2, –3 (2)

Notes to Table 1–6:

(1) This is a power-saving mode of operation

(2) The -4 speed grade is also possible for this configuration; however, it requires significant effort by the end user to close timing.

Table 1–7 lists the recommended speed grades for the Avalon-MM interface.

Table 1–7. Arria V GZ Recommended Speed Grades for All Avalon-MM Widths and Frequencies

Lane Rate	Link Width	Interface Width	Application Clock Frequency (MHz)	Recommended Speed Grades
Gen1	×1	64 bits	62.5 (1), 125	–1, –2, –3, –4
	×2	64 bits	125	–1, –2, –3, –4
	×4	64 bits	125	–1, –2, –3, –4
	×8	64 bits	250	–1, –2, –3 (2)
	×8	128 Bits	125	–1, –2, –3, –4
Gen2	×1	64 bits	62.5, 125	–1, –2, –3, –4
	×2	64 bits	125	–1, –2, –3 (2)
	×4	128 bits	125	–1, –2, –3, –4
	×8	128 bits	250	–1, –2, –3 (2)
Gen3	×1	64 bits	125	–1, –2, –3, –4
	×2	64 bits	250	–1, –2, –3 (2)
	×2	128 bits	125	–1, –2, –3 (2)
	×4	128 bits	250	–1, –2, –3 (2)
	×8	256 bits	250	–1, –2, –3 (2)

Notes to Table 1–7:

(1) This is a power-saving mode of operation.

(2) The -4 speed grade is also possible for this configuration; however, it requires significant effort by the end user to close timing.

For details on installation, refer to the *Altera Software Installation and Licensing Manual*.

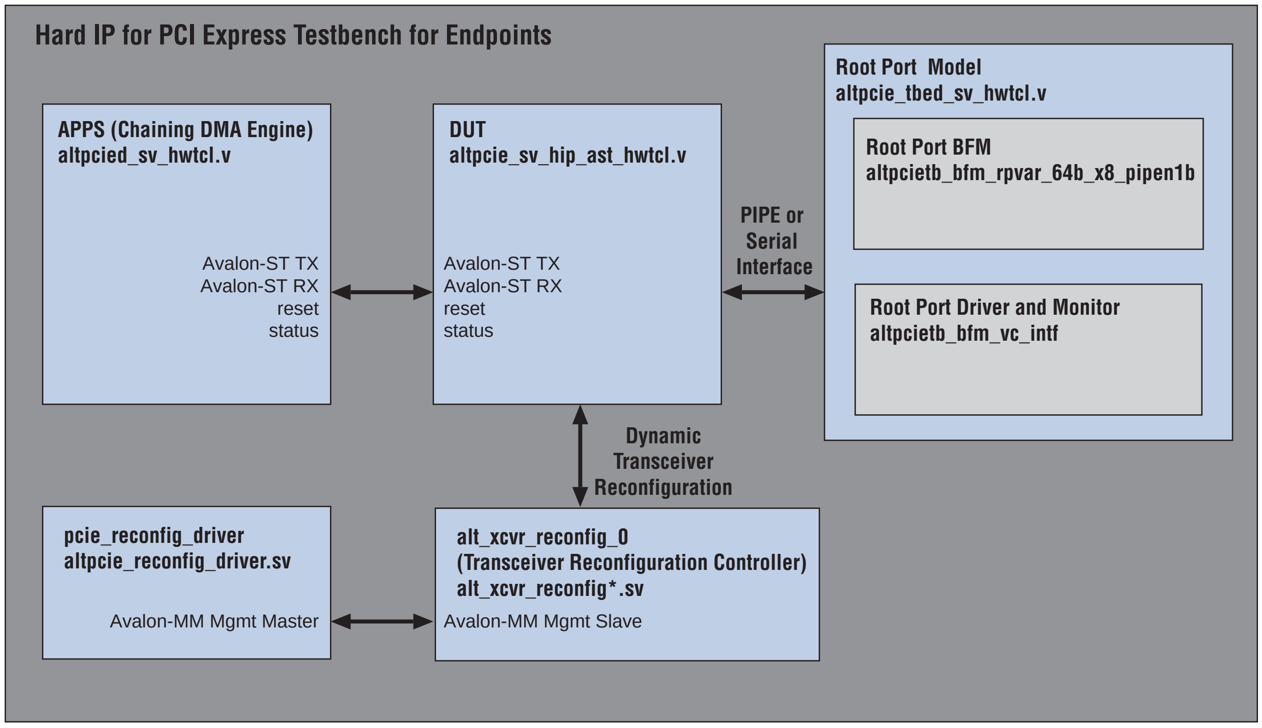
2. Getting Started with the Arria V GZ Hard IP for PCI Express

This section provides step-by-step instructions to help you quickly customize, simulate, and compile the Arria V GZ Hard IP for PCI Express using either the MegaWizard Plug-In Manager or Qsys design flow. When you install the Quartus II software you also install the IP Library. This installation includes design examples for Hard IP for PCI Express in <*install_dir*>**/ip/altera/altera_pcie/altera_pcie_hip_ast_ed/example_design/**<*device*> directory.

☞ If you have an existing Arria V GZ 12.1 or older design, you must regenerate it in 13.0 before compiling with the 13.0 version of the Quartus II software.

After you install the Quartus II software for 13.0, you can copy the design examples from the <*install_dir*>**/ip/altera/altera_pcie/altera_pcie_hip_ast_ed/example_design/**<*device*> directory. This walkthrough uses the Gen1 ×4 Endpoint. Figure 2–1 illustrates the top-level modules of the testbench in which the DUT, a Gen1 ×8 Endpoint, connects to a chaining DMA engine, labeled APPS in Figure 2–1, and a Root Port model. The Transceiver Reconfiguration Controller dynamically reconfigures analog settings to optimize signal quality of the serial interface. The pcie_reconfig_driver drives the Transceiver Reconfiguration Controller. The simulation can use the parallel PHY Interface for PCI Express (PIPE) or serial interface.

Figure 2–1. Testbench for an Endpoint



For a detailed explanation of this example design, refer to Chapter 18, Testbench and Design Example. If you choose the parameters specified in this chapter, you can run all of the tests included in Chapter 18.

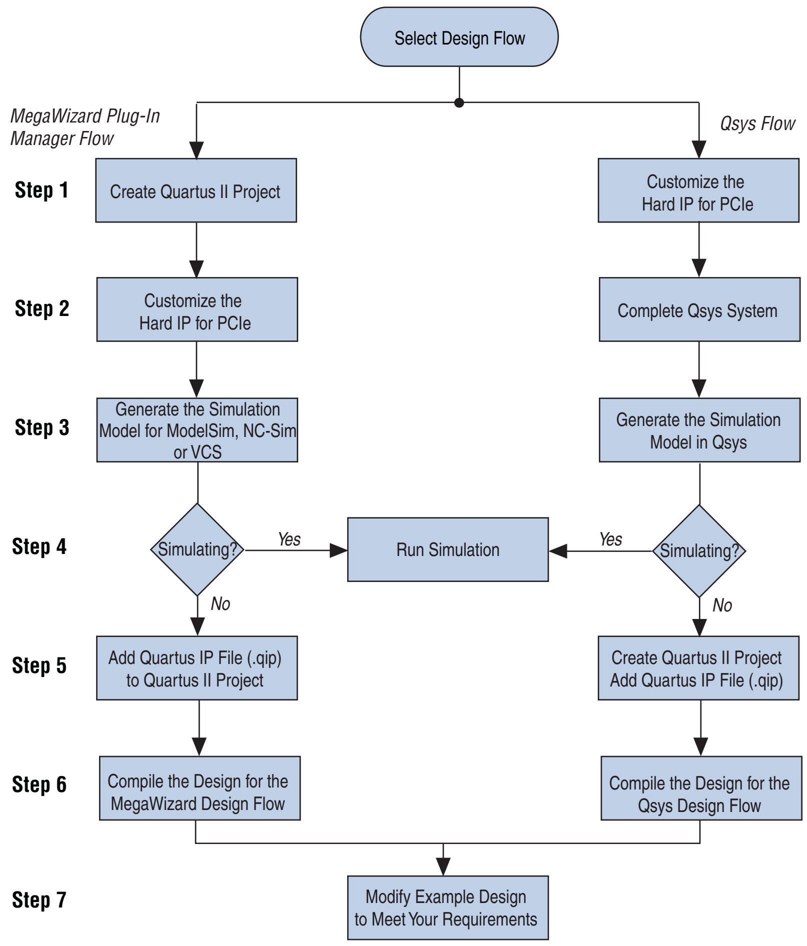
The Arria V GZ Hard IP for PCI Express offers exactly the same feature set in both the MegaWizard and Qsys design flows. Consequently, your choice of design flow depends on whether you want to integrate the Arria V GZHard IP for PCI Express using RTL instantiation or using Qsys, which is a system integration tool available in the Quartus II software.

For more information about Qsys, refer to *System Design with Qsys* in the *Quartus II Handbook*.

For more information about the Qsys GUI, refer to *About Qsys* in Quartus II Help.

Figure 2–2 illustrates the steps necessary to customize the Arria V GZ Hard IP for PCI Express and run the example design.

Figure 2–2. MegaWizard Plug-In Manager and Qsys Design Flows


Select Design Flow
MegaWizard Plug-In Manager Flow
Qsys Flow
Step 1
Create Quartus II Project
Customize the Hard IP for PCIe
Step 2
Customize the Hard IP for PCIe
Complete Qsys System
Step 3
Generate the Simulation Model for ModelSim, NC-Sim or VCS
Generate the Simulation Model in Qsys
Step 4
Simulating?
Yes
Run Simulation
Yes
Simulating?
No
No
Step 5
Add Quartus IP File (.qip) to Quartus II Project
Create Quartus II Project Add Quartus IP File (.qip)
Step 6
Compile the Design for the MegaWizard Design Flow
Compile the Design for the Qsys Design Flow
Step 7
Modify Example Design to Meet Your Requirements

MegaWizard Plug-In Manager Design Flow

This section guides you through the steps necessary to customize the Arria V GZ Hard IP for PCI Express and run the example testbench, starting with the creation of a Quartus II project.

Follow these steps to copy the example design files and create a Quartus II project.

1. Choose **Programs > Altera > Quartus II** <*version*> (Windows Start menu) to run the Quartus II software.
2. On the Quartus II File menu, click **New,** then **New Quartus II Project**, then **OK**.
3. Click **Next** in the **New Project Wizard: Introduction** (The introduction does not display if you previously turned it off.)
4. On the **Directory, Name, Top-Level Entity** page, enter the following information:
   a. The working directory for your project. This design example uses <*working_dir*>**/example_design**
   b. The name of the project. This design example uses **pcie_de_gen1_x8_ast128**.

   ☞ The Quartus II software specifies a top-level design entity that has the same name as the project automatically. Do not change this name.
5. Click **Next** to display the **Add Files** page.
6. Click **Yes**, if prompted, to create a new directory.
7. Click **Next** to display the **Family & Device Settings** page.
8. On the **Family & Device Settings** page, choose the following target device family and options:
   a. In the **Family** list, select Arria V GZ
   b. In the **Devices** list, select All
   c. In the **Available devices** list, select**5AGZME5K2F40C3**.
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 II project.

Customizing the Endpoint in the MegaWizard Plug-In Manager Design Flow

This section guides you through the process of customizing the Endpoint in the MegaWizard Plug-In Manager design flow. It specifies the same options that are chosen in Chapter 18, Testbench and Design Example.

Follow these steps to customize your variant in the MegaWizard Plug-In Manager:

1. On the Tools menu, click **MegaWizard Plug-In Manager**. The MegaWizard Plug-In Manager appears.
2. Select **Create a new custom megafunction variation** and click **Next**.
3. In **Which device family will you be using?** Select the **Arria V GZ** device family.
4. Expand the **Interfaces** directory under **Installed Plug-Ins** by clicking the + icon left of the directory name, expand **PCI Express**, then click **Arria V GZ Hard IP for PCI Express** <*version_number*>
5. Select the output file type for your design. This walkthrough supports VHDL and Verilog HDL. For this example, select **Verilog HDL**.
6. Specify a variation name for output files <*working_dir*>/**example_design/** <*variation name*>. For this walkthrough, specify <*working_dir*>**/example_design/gen1_x8.**
7. Click **Next** to open the parameter editor for the **Arria V GZ Hard IP for PCI Express**.
8. Specify the **System Settings** values listed inTable 2–1.

Table 2–1. System Settings Parameters

Parameter	Value
Number of Lanes	x8
Lane Rate	Gen 1 (2.5 Gbps)
Port type	Native endpoint
Application Layer interface	Avalon-ST 64-bit
RX buffer credit allocation - performance for received requests	Low
Reference clock frequency	100 MHz
Use 62.5 MHz Application Layer clock for ×1	Leave this option off
Use deprecated RX Avalon-ST data byte enable port (rx_st_be)	Leave this option off
Enable configuration via the PCIe link	Leave this option off
Enable byte parity ports on Avalon-ST interface	Leave this option off
Multiple packets per cycle	Leave this option off
Enable configuration via the PCIe link	Leave this option off
Enable Hard IP reconfiguration	Leave this option off

9. .Specify the **Base Address Register and Expansion ROM** settings listed Table 2–2.

Table 2–2. Base Address Register and Expansion ROM Settings

BAR Number	TYPE	Size
0	64-bit Prefetchable Memory	256 MBytes - 28 bits
1	Disable this BAR	N/A
2	32-bit Non-Prefetchable Memory	1 KByte - 10 bits
3	Disable this BAR	N/A
4	Disable this BAR	N/A
5	Disable this BAR	N/A
Expansion ROM	Disabled	—

10. Under the **Base and Limit Registers** heading, disable both the **Input/Output** and **Prefetchable memory** options. (These options are for Root Ports.)
11. Specify the **Device Identification Registers** settings listed in the center column of Table 2–3. The right-hand column of this table lists the value assigned to Altera devices. You must use the Altera values to run the reference design described in *AN 456 PCI Express High Performance Reference Design*. Be sure to use your company's values for your final product.

Table 2–3. Device Identification Registers

Register Name	Value	Altera Value
Vendor ID	`0x00000000`	0x00001172
Device ID	`0x00000001`	0x0000E001
Revision ID	`0x00000001`	0x00000001
Class Code	`0x00000000`	0x00FF0000
Subsystem Vendor ID	`0x00000000`	0x00001172
Subsystem Device ID	`0x00000000`	0x0000E001

12. Specify the **Device** settings listed in Table 2–4.

Table 2–4. Device Settings

Parameter	Value
Maximum payload size	**256 bytes**
Number of tags supported	**32**
Completion timeout range	**ABCD**
Implement completion timeout disable	**On**

13. On the **Error Reporting** tab, leave all options off.
14. Specify the **Link** settings listed in Table 2–5.

Table 2–5. Link Tab

Parameter	Value
Link port number	**1**
Data link layer active reporting	**Off**
Surprise down reporting	**Off**
Slot clock configuration	**On**

15. On the **MSI Capabilities** tab, for **MSI messages requested**, select **4**.
16. On the **MSI-X Capabilities** tab, turn I**mplement MSI-X** off.
17. On the **Slot Capabilities** tab, leave the **Slot register** turned off.

18. Table 2–6 lists the **Power Management** parameters.

Table 2–6. Power Management Parameters

Parameter	Value
Endpoint L0s acceptable latency	Maximum of 64 ns
Endpoint L1 acceptable latency	Maximum of 1 µs

19. On the **PHY Characteristics** tab, specify the settings in Table 2–7.

Table 2–7. PHY Characteristics

Parameter	Value
Gen2 transmit deemphasis	6dB
Use ATX PLL	Off

20. Click **Finish**. The Generation dialog box appears.
21. Turn on **Generate Example Design** to generate the Endpoint, testbench, and supporting files.
22. Click **Exit**.
23. Click **Yes** if you are prompted to add the Quartus II IP File (**.qip**) to the project.

    The **.qip** is a file generated by the parameter editor contains all of the necessary assignments and information required to process the IP core in the Quartus II compiler. Generally, a single **.qip** file is generated for each IP core.

Understanding the Files Generated

Table 2–8 provides an overview of directories and files generated.

Table 2–8. Qsys Generation Output Files

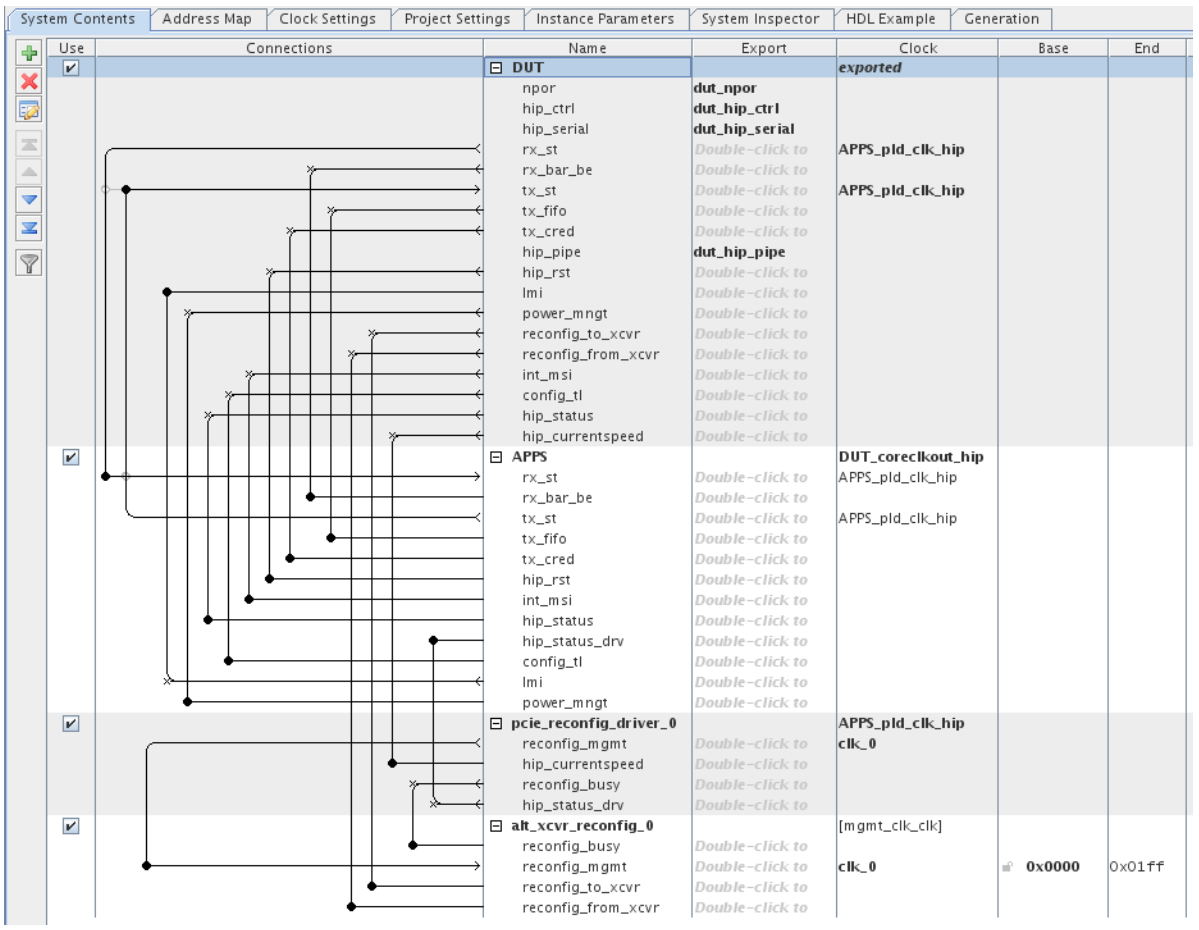
Directory	Description
<working_dir>/<variant_name>/	Includes the files for synthesis
<working_dir>/*<variant_name>***_sim/ altera_pcie_***<device>***_hip_ast**	Includes the simulation files.
<working_dir>/*<variant_name>***_example_design/ altera_pcie_***<device>***_hip_ast**	Includes a Qsys testbench that connects the Endpoint to a chaining DMA engine, Transceiver Reconfiguration Controller, and driver for the Transceiver Reconfiguration Controller.

Follow these steps to generate the chaining DMA testbench from the Qsys system design example.

1. On the Quartus II File menu, click **Open**.
2. Navigate to the Qsys system in the **altera_pcie_**<*device*>**_hip_ast** subdirectory.

3. Click **pcie_de_gen1_x8_ast128.qsys** to bring up the Qsys design. Figure 2–3 illustrates this Qsys system.

Figure 2–3. Qsys System Connecting the Endpoint Variant and Chaining DMA Testbench



Use	Connections	Name	Export	Clock	Base	End
✓		⊟ DUT		*exported*		
		npor	dut_npor			
		hip_ctrl	dut_hip_ctrl			
		hip_serial	dut_hip_serial			
		rx_st	Double-click to	APPS_pld_clk_hip		
		rx_bar_be	Double-click to			
		tx_st	Double-click to	APPS_pld_clk_hip		
		tx_fifo	Double-click to			
		tx_cred	Double-click to			
		hip_pipe	dut_hip_pipe			
		hip_rst	Double-click to			
		lmi	Double-click to			
		power_mngt	Double-click to			
		reconfig_to_xcvr	Double-click to			
		reconfig_from_xcvr	Double-click to			
		int_msi	Double-click to			
		config_tl	Double-click to			
		hip_status	Double-click to			
		hip_currentspeed	Double-click to			
✓		⊟ APPS		DUT_coreclkout_hip		
		rx_st	Double-click to	APPS_pld_clk_hip		
		rx_bar_be	Double-click to			
		tx_st	Double-click to	APPS_pld_clk_hip		
		tx_fifo	Double-click to			
		tx_cred	Double-click to			
		hip_rst	Double-click to			
		int_msi	Double-click to			
		hip_status	Double-click to			
		hip_status_drv	Double-click to			
		config_tl	Double-click to			
		lmi	Double-click to			
		power_mngt	Double-click to			
✓		⊟ pcie_reconfig_driver_0		APPS_pld_clk_hip		
		reconfig_mgmt	Double-click to	clk_0		
		hip_currentspeed	Double-click to			
		reconfig_busy	Double-click to			
		hip_status_drv	Double-click to			
✓		⊟ alt_xcvr_reconfig_0		[mgmt_clk_clk]		
		reconfig_busy	Double-click to			
		reconfig_mgmt	Double-click to	clk_0	0x0000	0x01ff
		reconfig_to_xcvr	Double-click to			
		reconfig_from_xcvr	Double-click to			

4. To display the parameters of the **APPS** component shown in Figure 2–3, click on it and then select **Edit** from the right-mouse menu. illustrates this component. Note that the values for the following parameters match those set in the DUT component:
   - **Targeted Device Family**
   - **Lanes**
   - **Lane Rate**
   - **Application Clock Rate**
   - **Port**
   - **Application interface**
   - **Tags supported**
   - **Maximum payload size**
   - **Number of Functions**

   ☞ You can use this Qsys APPS component to test any Endpoint variant with compatible values for these parameters.

5. To close the **APPS** component, click the **X** in the upper right-hand corner of the parameter editor.

Go to "Simulating the Example Design" on page 2–11 for instructions on system simulation.

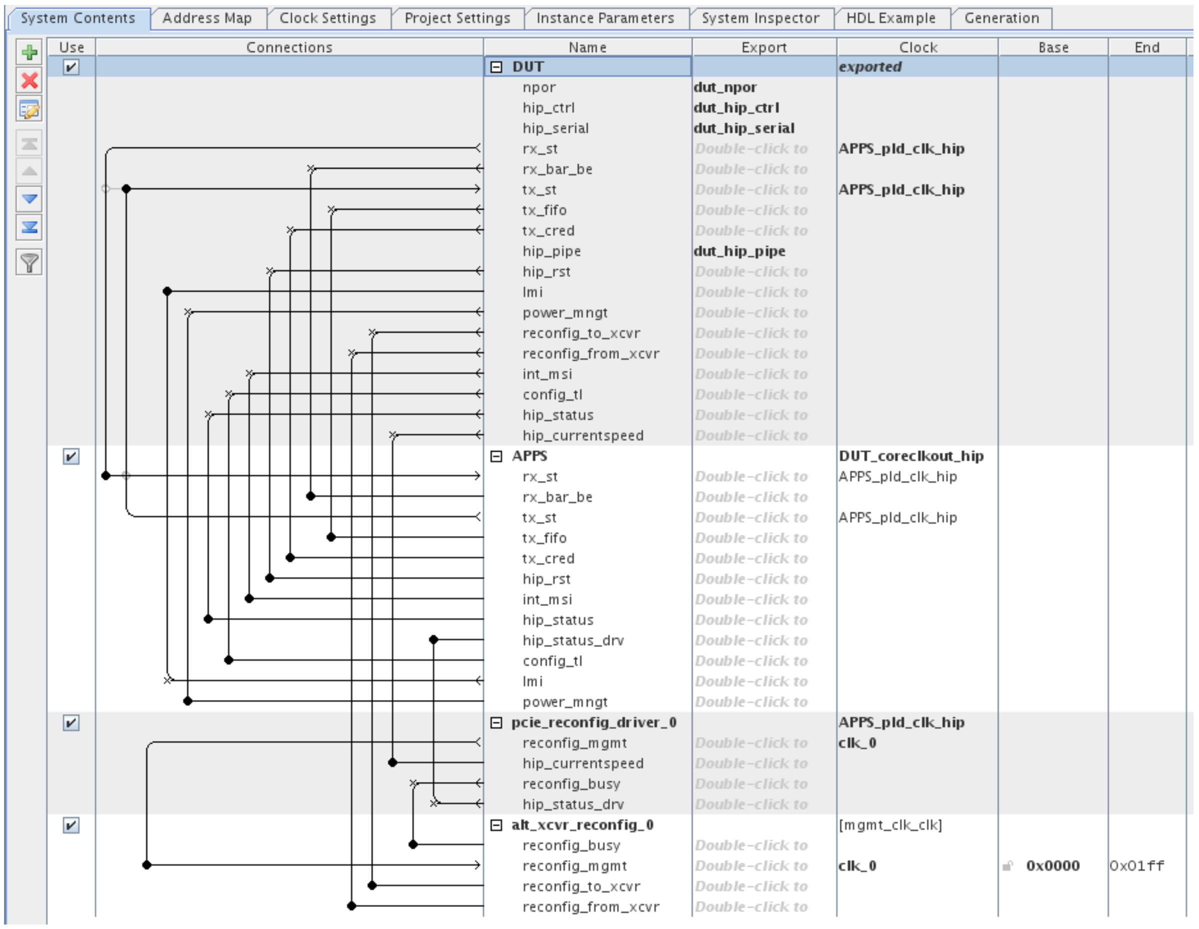
Qsys Design Flow

This section guides you through the steps necessary to customize the Arria V GZ Hard IP for PCI Express and run the example testbench in Qsys. Reviewing the Qsys Example Design for PCIe

For this example, copy the Gen1 ×8 Endpoint example design from installation directory: <*install_dir*>**/ip/altera/altera_pcie/altera_pcie_hip_ast_ed/example_design /**<*device*> directory to a working directory.

Figure 2–4 illustrates this Qsys system.

Figure 2–4. Complete Gen1 ×4 Endpoint (DUT) Connected to Example Design (APPS)



Use	Connections	Name	Export	Clock	Base	End
✓		⊟ DUT		*exported*		
		npor	dut_npor			
		hip_ctrl	dut_hip_ctrl			
		hip_serial	dut_hip_serial			
		rx_st	Double-click to	APPS_pld_clk_hip		
		rx_bar_be	Double-click to			
		tx_st	Double-click to	APPS_pld_clk_hip		
		tx_fifo	Double-click to			
		tx_cred	Double-click to			
		hip_pipe	dut_hip_pipe			
		hip_rst	Double-click to			
		lmi	Double-click to			
		power_mngt	Double-click to			
		reconfig_to_xcvr	Double-click to			
		reconfig_from_xcvr	Double-click to			
		int_msi	Double-click to			
		config_tl	Double-click to			
		hip_status	Double-click to			
		hip_currentspeed	Double-click to			
✓		⊟ APPS		DUT_coreclkout_hip		
		rx_st	Double-click to	APPS_pld_clk_hip		
		rx_bar_be	Double-click to			
		tx_st	Double-click to	APPS_pld_clk_hip		
		tx_fifo	Double-click to			
		tx_cred	Double-click to			
		hip_rst	Double-click to			
		int_msi	Double-click to			
		hip_status	Double-click to			
		hip_status_drv	Double-click to			
		config_tl	Double-click to			
		lmi	Double-click to			
		power_mngt	Double-click to			
✓		⊟ pcie_reconfig_driver_0		APPS_pld_clk_hip		
		reconfig_mgmt	Double-click to	clk_0		
		hip_currentspeed	Double-click to			
		reconfig_busy	Double-click to			
		hip_status_drv	Double-click to			
✓		⊟ alt_xcvr_reconfig_0		[mgmt_clk_clk]		
		reconfig_busy	Double-click to			
		reconfig_mgmt	Double-click to	clk_0	0x0000	0x01ff
		reconfig_to_xcvr	Double-click to			
		reconfig_from_xcvr	Double-click to			

The example design includes the following four 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.
- pcie_reconfig_driver_0—This Avalon-MM master drives the Transceiver Reconfiguration Controller. The pcie_reconfig_driver_0 is implemented in clear text that you can modify if your design requires different reconfiguration functions. After you generate your Qsys system, the Verilog HDL for this component is available as: *<working_dir>*/*<variant_name>***/testbench/** *<variant_name>***_tb/simulation/submodules/altpcie_reconfig_driver.sv**.

- Transceiver Reconfiguration Controller—The Transceiver Reconfiguration Controller dynamically reconfigures analog settings to improve signal quality. For Gen1 and Gen2 data rates, the Transceiver Reconfiguration Controller must perform offset cancellation and PLL calibration. For the Gen3 data rate the Transceiver Reconfiguration Controller must also perform adaptive equalization (AEQ).

Generating the Testbench

Follow these steps to generate the chaining DMA testbench:

1. On the Qsys **Generation** tab, specify the parameters listed in Table 2–9.

Table 2–9. Parameters to Specify on the Generation Tab in Qsys

Parameter	Value
Simulation	
Create simulation model	**None.** (This option generates a simulation model you can include in your own custom testbench.)
Create testbench Qsys system	**Standard, BFMs for standard Avalon interfaces**
Create testbench simulation model	**Verilog**
Synthesis	
Create HDL design files for synthesis	Turn this option on
Create block symbol file (.bsf)	Turn this option on
Output Directory	
Path	**pcie_qsys/pcie_de_gen1_x8_ast128**
Simulation	Leave this option blank
Testbench (1)	**pcie_qsys/pcie_de_gen1_x8_ast128/testbench**
Synthesis (2)	**pcie_qsys/pcie_de_gen1_x8_ast128/synthesis**

Note to Table 2–9:

(1) Qsys automatically creates this path by appending **testbench** to the output directory/.

(2) Qsys automatically creates this path by appending **synthesis** to the output directory/.

2. Click the **Generate** button at the bottom of the **Generation** tab to create the chaining DMA testbench.

Understanding the Files Generated

Table 2–10 provides an overview of the files and directories Qsys generates.

Table 2–10. Qsys Generation Output Files (Part 1 of 2)

Directory	Description
<testbench_dir>/<variant_name>/ **synthesis**	includes the top-level HDL file for the Hard I for PCI Express and the **.qip** file that lists all of the necessary assignments and information required to process the IP core in the Quartus II compiler. Generally, a single **.qip** file is generated for each IP core.
<testbench_dir>/<variant_name>/ **synthesis/submodules**	Includes the HDL files necessary for Quartus II synthesis.

Table 2–10. Qsys Generation Output Files (Part 2 of 2)

Directory	Description
<testbench_dir>/<variant_name>/ **testbench/**	Includes testbench subdirectories for the Aldec, Cadence and Mentor simulation tools with the required libraries and simulation scripts.
<testbench_dir>/<variant_name>/ **testbench/***<cad_vendor>*	Includes the HDL source files and scripts for the simulation testbench.

Simulating the Example Design

Follow these steps to compile the testbench for simulation and run the chaining DMA testbench.

1. Start your simulation tool. This example uses the ModelSim® software.
2. From the ModelSim transcript window, in the testbench directory (**./example_design/altera_pcie_***<device>***_hip_ast/***<variant>***/testbench/mentor**) type the following commands:

   a. `do msim_setup.tcl` ↵

   b. h ↵ (This is the ModelSim help command.)

   c. ld_debug ↵ (This command compiles all design files and elaborates the top-level design without any optimization.)

   d. run -all ↵

Example 2–1 shows a partial transcript from a successful simulation. As this transcript illustrates, the simulation includes the following stages:

- Link training
- Configuration
- DMA reads and writes

- Root Port to Endpoint memory reads and writes

Example 2–1. Excerpts from Transcript of Successful Simulation Run

```
Time: 56000  Instance: top_chaining_testbench.ep.epmap.pll_250mhz_to_500mhz.
# Time: 0  Instance:
pcie_de_gen1_x8_ast128_tb.dut_pcie_tb.genblk1.genblk1.altpcietb_bfm_top_rp.rp.rp.nl00O
0i.Arria V GZii_pll.pll1
#  Note : Arria V GZ II PLL locked to incoming clock
# Time: 25000000  Instance:
pcie_de_gen1_x8_ast128_tb.dut_pcie_tb.genblk1.genblk1.altpcietb_bfm_top_rp.rp.rp.nl00O
0i.Arria V GZii_pll.pll1
# INFO:       464 ns Completed initial configuration of Root Port.
# INFO:       3661 ns  RP LTSSM State: DETECT.ACTIVE
# INFO:       3693 ns  RP LTSSM State: POLLING.ACTIVE
# INFO:       3905 ns  EP LTSSM State: DETECT.ACTIVE
# INFO:       4065 ns  EP LTSSM State: POLLING.ACTIVE
# INFO:       6369 ns  EP LTSSM State: POLLING.CONFIG
# INFO:       6461 ns  RP LTSSM State: POLLING.CONFIG
# INFO:      7741 ns  RP LTSSM State: CONFIG.LINKWIDTH.START
# INFO:      7969 ns  EP LTSSM State: CONFIG.LINKWIDTH.START
# INFO:      8353 ns  EP LTSSM State: CONFIG.LINKWIDTH.ACCEPT
# INFO:            8781 ns  RP LTSSM State: CONFIG.LINKWIDTH.ACCEPT
# INFO:      9537 ns  EP LTSSM State: CONFIG.LANENUM.WAIT
# INFO:      9857 ns  EP LTSSM State: CONFIG.LANENUM.ACCEPT
# INFO:      9933 ns  RP LTSSM State: CONFIG.LANENUM.ACCEPT
# INFO:      10189 ns  RP LTSSM State: CONFIG.COMPLETE
# INFO:      10689 ns  EP LTSSM State: CONFIG.COMPLETE
# INFO:      12109 ns  RP LTSSM State: CONFIG.IDLE
# INFO:      13697 ns  EP LTSSM State: CONFIG.IDLE
# INFO:       13889 ns  EP LTSSM State: L0
# INFO:       13981 ns   RP LTSSM State: L0
# INFO:     17800 ns Configuring Bus 001, Device 001, Function 00
# INFO:     17800 ns  EP Read Only Configuration Registers:
# INFO:       17800 ns          Vendor ID: 1172
# INFO:       17800 ns          Device ID: E001
# INFO:       17800 ns         Revision ID: 01
# INFO:       17800 ns         Class Code: FF0000
# INFO:      17800 ns     Subsystem Vendor ID: 1172
# INFO:      17800 ns         Subsystem ID: E001
# INFO:      17800 ns        Interrupt Pin: INTA# used
# INFO:       17800 ns
# INFO:      20040 ns  PCI MSI Capability Register:
# INFO:      20040 ns  64-Bit Address Capable: Supported
# INFO:      20040 ns     Messages Requested:  4
# INFO:       20040 ns
#INFO:      31208 ns  EP PCI Express Link Status Register (1081):
# INFO:      31208 ns    Negotiated Link Width: x8
# INFO:     31208 ns     Slot Clock Config: System Reference Clock Used
# INFO:      33481 ns RP LTSSM State: RECOVERY.RCVRLOCK
# INFO:      34321 ns EP LTSSM State: RECOVERY.RCVRLOCK
# INFO:      34961 ns EP LTSSM State: RECOVERY.RCVRCFG
# INFO:      35161 ns RP LTSSM State: RECOVERY.RCVRCFG
# INFO:      36377 ns RP LTSSM State: RECOVERY.IDLE
# INFO:      37457 ns EP LTSSM State: RECOVERY.IDLE
# INFO:       37649 ns  EP LTSSM State: L0
# INFO:       37737 ns  RP LTSSM State: L0
# INFO:      39944 ns      Current Link Speed: 2.5GT/s
# INFO:     58904 ns Completed configuration of Endpoint BARs.
# INFO:       61288 ns ---------
# INFO:      61288 ns TASK:chained_dma_test
# INFO:           61288 ns    DMA: Read
```

Example 2–1. Excerpts from Transcript of Successful Simulation Run (continued)

```
# INFO:       8973 ns RP LTSSM State: CONFIG.LANENUM.WAIT

# INFO:       61288 ns ---------
# INFO:       61288 ns TASK:dma_rd_test
# INFO:       61288 ns ---------
# INFO:       61288 ns TASK:dma_set_rd_desc_data
# INFO:       61288 ns ---------
# INFO:       61288 ns TASK:dma_set_msi READ
# INFO:       61288 ns Message Signaled Interrupt Configuration
# INFO:       61288 ns  msi_address (RC memory)= 0x07F0
# INFO:       63512 ns  msi_control_register = 0x0084
# INFO:       72440 ns  msi_expected = 0xB0FC
# INFO:       72440 ns  msi_capabilities address = 0x0050
# INFO:       72440 ns  multi_message_enable = 0x0002
# INFO:       72440 ns  msi_number = 0000
# INFO:       72440 ns  msi_traffic_class = 0000
# INFO:       72440 ns ---------
# INFO:            72440 ns TASK:dma_set_header READ
# INFO:       72440 ns Writing Descriptor header
# INFO:       72480 ns data content of the DT header
# INFO:       72480 ns
# INFO:       72480 ns Shared Memory Data Display:
# INFO:       72480 ns Address  Data
# INFO:       72480 ns -------  ----
# INFO:       72480 ns 00000900 00000003 00000000 00000900 CAFEFADE
# INFO:       72480 ns ---------
# INFO:       72480 ns TASK:dma_set_rclast
# INFO:       72480 ns  Start READ DMA : RC issues MWr (RCLast=0002)
# INFO:       72496 ns ---------
# INFO:    72509 ns TASK:msi_poll  Polling MSI Address:07F0---> Data:FADE......
# INFO:            72693 ns TASK:rcmem_poll  Polling RC Address0000090C   current data
(0000FADE)  expected data (00000002)
# INFO:            80693 ns TASK:rcmem_poll  Polling RC Address0000090C   current data
(00000000)  expected data (00000002)
# INFO:     84749 ns TASK:msi_poll  Received DMA Read MSI(0000) : B0FC
# INFO:            84893 ns TASK:rcmem_poll  Polling RC Address0000090C   current data
(00000002)  expected data (00000002)
# INFO:     84893 ns TASK:rcmem_poll ---> Received Expected Data (00000002)
# INFO:            84901 ns ---------
# INFO:   84901 ns Completed DMA Read # INFO:   84901 ns TASK:chained_dma_test
# INFO:       84901 ns   DMA: Write
# INFO:       84901 ns ---------
# INFO:       84901 ns TASK:dma_wr_test
# INFO:       84901 ns   DMA: Write
# INFO:       84901 ns ---------
# INFO:       84901 ns TASK:dma_set_wr_desc_data
# INFO:       84901 ns ---------
# INFO:       84901 ns TASK:dma_set_msi WRITE
# INFO:       84901 ns Message Signaled Interrupt Configuration
# INFO:       84901 ns  msi_address (RC memory)= 0x07F0
# INFO:       87109 ns  msi_control_register = 0x00A5
# INFO:       96005 ns  msi_expected = 0xB0FD
# INFO:            96005 ns   msi_capabilities address = 0x0050
```

Example 2-1Excerpts from Transcript of Successful Simulation Run (continued)

```
# INFO:      96005 ns  multi_message_enable = 0x0002
# INFO:       96005 ns  msi_number = 0001
# INFO:       96005 ns  msi_traffic_class = 0000
# INFO:       96005 ns ---------
# INFO:       96005 ns TASK:dma_set_header WRITE
# INFO:       96005 ns Writing Descriptor header
# INFO:       96045 ns data content of the DT header
# INFO:       96045 ns
# INFO:       96045 ns Shared Memory Data Display:
# INFO:       96045 ns Address  Data
# INFO:       96045 ns -------  ----
# INFO:     96045 ns 00000800 10100003 00000000 00000800 CAFEFADE
# INFO:       96045 ns ---------
# INFO:       96045 ns TASK:dma_set_rclast
# INFO:     96045 ns  Start WRITE DMA : RC issues MWr (RCLast=0002)
# INFO:       96061 ns ---------
# INFO:    96073 ns TASK:msi_poll  Polling MSI Address:07F0---> Data:FADE......
# INFO:           96257 ns TASK:rcmem_poll  Polling RC Address0000080C   current data
(0000FADE)  expected data (00000002)
# INFO:          101457 ns TASK:rcmem_poll  Polling RC Address0000080C   current data
(00000000)  expected data (00000002)
# INFO:    105177 ns TASK:msi_poll  Received DMA Write MSI(0000) : B0FD
# INFO:          105257 ns TASK:rcmem_poll  Polling RC Address0000080C   current data
(00000002)  expected data (00000002)
# INFO:    105257 ns TASK:rcmem_poll ---> Received Expected Data (00000002)
# INFO:      105265 ns ---------
# INFO:      105265 ns Completed DMA Write
# INFO:      105265 ns ---------
# INFO:      105265 ns TASK:check_dma_data
# INFO:     105265 ns  Passed : 0644 identical dwords.
# INFO:      105265 ns ---------
# INFO:      105265 ns TASK:downstream_loop
# INFO:   107897 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:   110409 ns Passed: 0008 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:   113033 ns Passed: 0012 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:   115665 ns Passed: 0016 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:   118305 ns Passed: 0020 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:   120937 ns Passed: 0024 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:   123577 ns Passed: 0028 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:   126241 ns Passed: 0032 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:   128897 ns Passed: 0036 same bytes in BFM mem addr 0x00000040 and 0x00000840
# INFO:   131545 ns Passed: 0040 same bytes in BFM mem addr 0x00000040 and 0x00000840
# SUCCESS: Simulation stopped due to successful completion!
```

Understanding Channel Placement Guidelines

For more information about transceiver clocking and channel placement refer to "Transceiver Clocking and Channel Placement Guidelines" in *Transceiver Configurations in Arria V GZ Devices*.

Compiling the Design in the MegaWizard Plug-In Manager Design Flow

Before compiling the complete example design in the Quartus II software, you must add the example design files that you generated in Qsys to your Quartus II project. The Quartus II IP File (**.qip**) lists all files necessary to compile the project.

Follow these steps to add the Quartus II IP File (**.qip**) to the project:

1. On the Project menu, select **Add/Remove Files in Project**.
2. Click the browse button next the **File name** box and browse to the **gen1_x8_example_design/altera_pcie_sv_hip_ast/pcie_de_gen1_x8_ast128/ synthesis/** directory.
3. In the **Files of type** list, **pcie_de_gen1_x8_ast128.qip** and then click **Open**.
4. On the **Add Files** page, click **Add**, then click **OK**.
5. Add the Synopsys Design Constraints (SDC) shown in Example 2–2, to the top-level design file for your Quartus II project.

Example 2–2. Synopsys Design Constraint

```
create_clock -period “100 MHz” -name {refclk_pci_express} {*refclk_*}
derive_pll_clocks
derive_clock_uncertainty

######################################################################
# PHY IP reconfig controller constraints
# Set reconfig_xcvr clock
# Modify to match the actual clock pin name
# used for this clock, and also changed to have the correct period set
create_clock -period "125 MHz" -name {reconfig_xcvr_clk}
{*reconfig_xcvr_clk*}

######################################################################

# HIP Soft reset controller SDC constraints
set_false_path -to   [get_registers
*altpcie_rs_serdes|fifo_err_sync_r[0]]
set_false_path -from [get_registers *sv_xcvr_pipe_native*] -to
[get_registers *altpcie_rs_serdes|*]

# Hard IP testin pins SDC constraints
set_false_path -from [get_pins -compatibility_mode *hip_ctrl*]
```

6. On the Processing menu, select **Start Compilation**.

Compiling the Design in the Qsys Design Flow

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

Complete the following steps to create your Quartus II project:

1. From the Windows Start Menu, choose **Programs > Altera > Quartus II 13.0** to run the Quartus II software.

2. Click the browse button next to the **File Name** box and browse to the synthesis directory that includes your Qsys project, <*working_dir*>**/gen1_x8_example_design/altera_pcie_sv_hip_ast/pcie_de_gen1_x8_ast128/synthesis**
3. On the Quartus II File menu, click **New,** then **New Quartus II Project**, then **OK**.
4. Click **Next** in the **New Project Wizard: Introduction** (The introduction does not appear if you previously turned it off.)
5. 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 buttons and select your variant name, **pcie_de_gen1_x8_ast128,** then click **Open.**↵

   ☞ If the top-level design entity and Qsys system names are identical, the Quartus II software treats the Qsys system as the top-level design entity.

6. Click **Next** to display the **Add Files** page.
7. Complete the following steps to add the Quartus II IP File (**.qip**) to the project:
   a. Click the **browse** button. The **Select File** dialog box appears.
   b. In the **Files of type** list, select **IP Variation Files (*.qip)**.
   c. Click **pcie_de_gen1_x8_ast128.qip** and then click **Open**.
   d. On the **Add Files** page, click **Add**, then click **OK**.
8. Click **Next** to display the **Device** page.
9. On the **Family & Device Settings** page, choose the following target device family and options:
   a. In the **Family** list, select Arria V GZ
   b. In the **Devices** list, select All
   c. In the **Available devices** list, select**5AGZME5K2F40C3**.
10. Click **Next** to close this page and display the **EDA Tool Settings** page.
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 II project.

14. Add the Synopsys Design Constraint (SDC) shown inExample 2–3, to the top-level design file for your Quartus II project.

Example 2–3. Synopsys Design Constraint

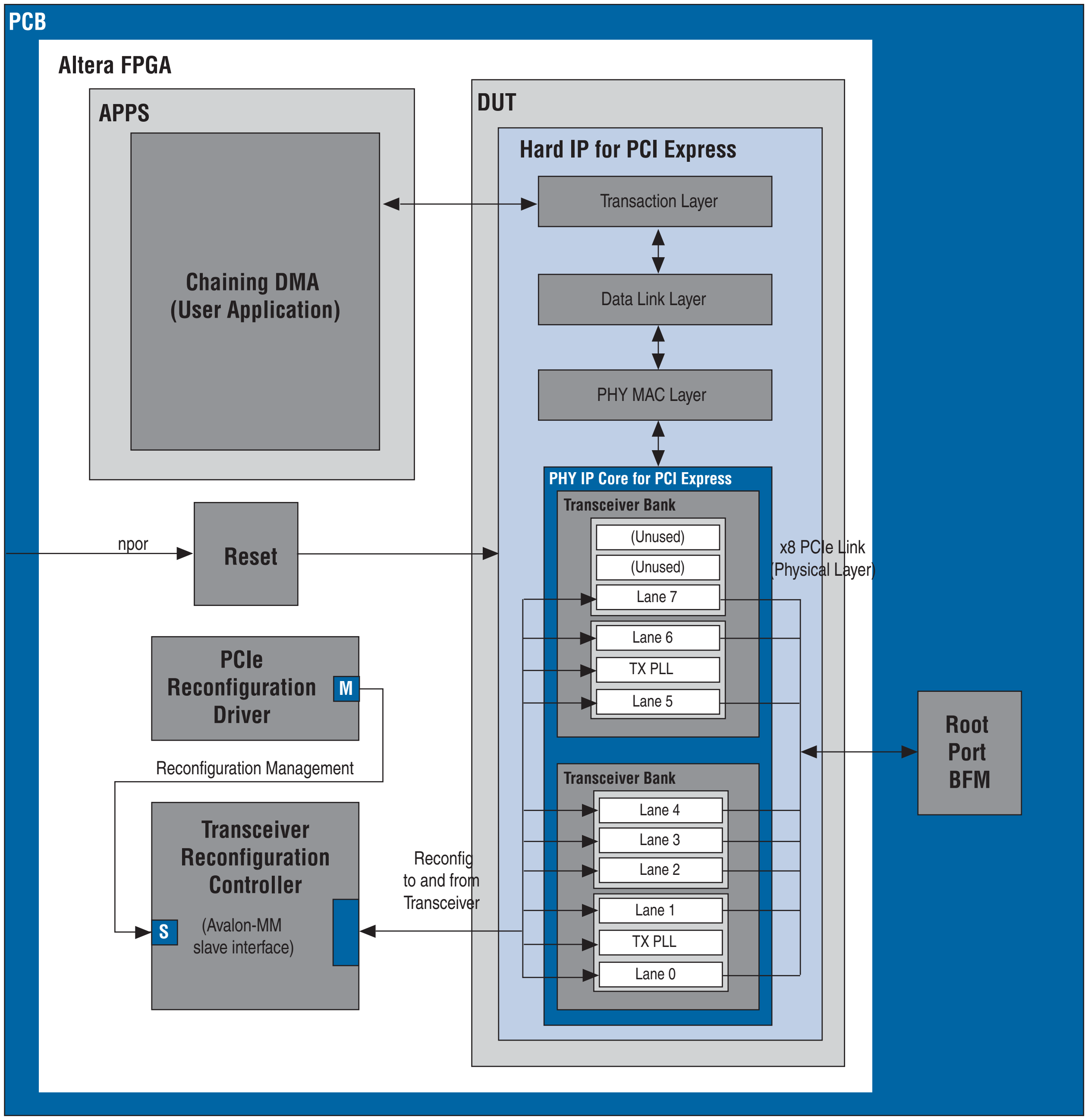
```
create_clock -period “100 MHz” -name {refclk_pci_express} {*refclk_*}
derive_pll_clocks
derive_clock_uncertainty

########################################################################
# PHY IP reconfig controller constraints
# Set reconfig_xcvr clock
# Modify to match the actual clock pin name
# used for this clock, and also changed to have the correct period set
create_clock -period "125 MHz" -name {reconfig_xcvr_clk}
{*reconfig_xcvr_clk*}

########################################################################

# HIP Soft reset controller SDC constraints
set_false_path -to   [get_registers
*altpcie_rs_serdes|fifo_err_sync_r[0]]
set_false_path -from [get_registers *sv_xcvr_pipe_native*] -to
[get_registers *altpcie_rs_serdes|*]

# Hard IP testin pins SDC constraints
set_false_path -from [get_pins -compatibilitly_mode *hip_ctrl*]
```

15. To compile your design using the Quartus II software, on the Processing menu, click **Start Compilation**. The Quartus II software then performs all the steps necessary to compile your design.

Modifying the Example Design

To use this example design as the basis of your own design, replace the Chaining DMA Example shown in Figure 2–5 with your own Application Layer design. Then modify the Root Port BFM driver to generate the transactions needed to test your Application Layer.

Figure 2–5. Testbench for PCI Express


PCB
Altera FPGA
APPS
Chaining DMA (User Application)
DUT
Hard IP for PCI Express
Transaction Layer
Data Link Layer
PHY MAC Layer
PHY IP Core for PCI Express
Transceiver Bank
(Unused)
(Unused)
Lane 7
Lane 6
TX PLL
Lane 5
Transceiver Bank
Lane 4
Lane 3
Lane 2
Lane 1
TX PLL
Lane 0
x8 PCIe Link (Physical Layer)
Root Port BFM
npor
Reset
PCIe Reconfiguration Driver
M
Reconfiguration Management
Transceiver Reconfiguration Controller
S
(Avalon-MM slave interface)
Reconfig to and from Transceiver

3. Getting Started with the Avalon-MM Arria V GZ Hard IP for PCI Express

This Qsys design example provides detailed step-by-step instructions to generate a Qsys system. When you install the Quartus II software you also install the IP Library. This installation includes design examples for the Avalon-MM Arria V GZ Hard IP for PCI Express in the <*install_dir*>**/ip/altera/altera_pcie/altera_pcie_avgz_hip_avmm/example_designs/** directory.

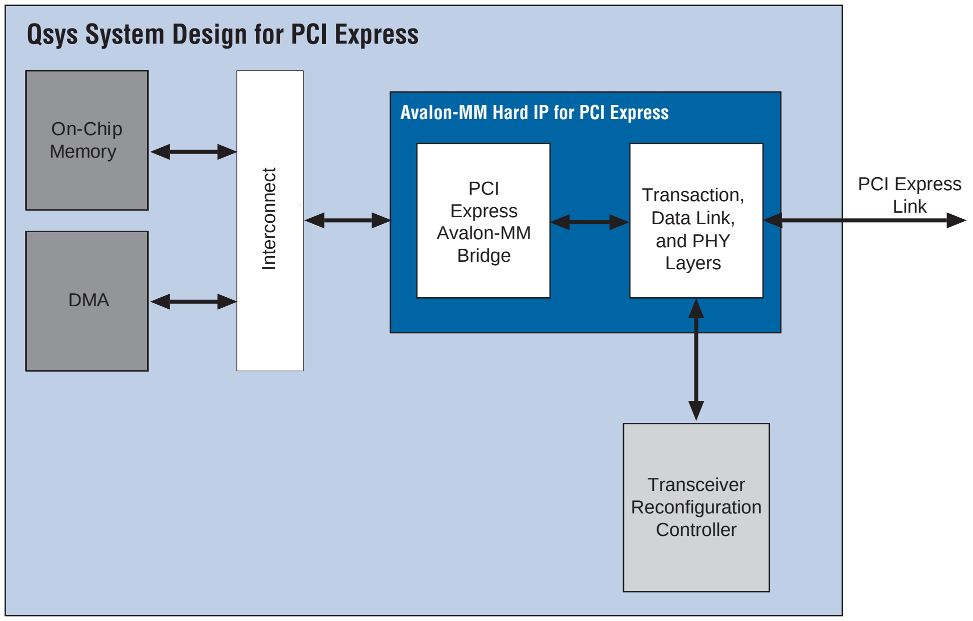
The design examples contain the following components:

- Avalon-MM Arria V GZ Hard IP for PCI Express ×4 IP core
- On-Chip memory
- DMA controller
- Transceiver Reconfiguration Controller

In the Qsys design flow you select the Avalon-MM Arria V GZ Hard IP for PCI Express as a component. This component supports PCI Express ×1, ×2, ×4, or ×8 Endpoint applications with bridging logic to convert PCI Express packets to Avalon-MM transactions and vice versa. The design example included in this chapter illustrates the use of an Endpoint with an embedded transceiver.

Figure 3–1 provides a high-level block diagram of the design example included in this release.

Figure 3–1. Qsys Generated Endpoint


Qsys System Design for PCI Express
On-Chip Memory
DMA
Interconnect
Avalon-MM Hard IP for PCI Express
PCI Express Avalon-MM Bridge
Transaction, Data Link, and PHY Layers
PCI Express Link
Transceiver Reconfiguration Controller

As Figure 3–1 illustrates, the design example transfers data between an on-chip memory buffer located on the Avalon-MM side and a PCI Express memory buffer located on the root complex side. The data transfer uses the DMA component which is programmed by the PCI Express software application running on the Root Complex processor. The example design also includes the Transceiver Reconfiguration Controller which allows you to dynamically reconfigure transceiver settings. This component is necessary for high performance transceiver designs.

Running Qsys

Follow these steps to launch Qsys:

1. Choose **Programs > Altera > Quartus II><*version_number*>** (Windows Start menu) to run the Quartus II software. Alternatively, you can also use the Quartus II Web Edition software.
2. On the Quartus II File menu, click **New.**
3. Select **Qsys System File** and click **OK**. Qsys appears.
4. To establish global settings, click the **Project Settings** tab.
5. Specify the settings in Table 3–1.

Table 3–1. Project Settings

Parameter	Value
Device family	Arria V GZ
Device	5GZME5K2F403
Clock crossing adapter type	Handshake
Limit interconnect pipeline stages to	2
Generation Id	0

Refer to *Creating a System with Qsys* in volume 1 of the *Quartus II Handbook* for more information about how to use Qsys, including information about the Project Settings tab.

For an explanation of each Qsys menu item, refer to *About Qsys* in Quartus II Help.

This example design requires that you specify the same name for the Qsys system as for the top-level project file. However, this naming is not required for your own design. If you want to choose a different name for the system file, you must create a wrapper HDL file that matches the project top level name and instantiate the generated system.

6. To add modules from the **Component Library** tab, under **Interface Protocols** in the **PCI** folder, click the **Avalon-MM Arria V GZ Hard IP for PCI Express** component, then click **+Add**.

Customizing the Arria V GZ Hard IP for PCI Express IP Core

The parameter editor uses bold headings to divide the parameters into separate sections. You can use the scroll bar on the right to view parameters that are not initially visible. Follow these steps to parameterize the Hard IP for PCI Express IP core:

1. Under the **System Settings** heading, specify the settings in Table 3–2.

Table 3–2. System Settings

Parameter	Value
Number of lanes	×4
Lane rate	Gen1 (2.5 Gbps)
Port type	Native endpoint
RX buffer credit allocation – performance for received requests	Low
Reference clock frequency	100 MHz
Use 62.5 MHz application clock	Off
Enable configuration via the PCIe link	Off
ATX PLL	Off

2. Under the **PCI Base Address Registers (Type 0 Configuration Space)** heading, specify the settings in Table 3–3.

Table 3–3. PCI Base Address Registers (Type 0 Configuration Space)

BAR	BAR Type	BAR Size
0	64-bit Prefetchable Memory	0
1	Not used	0
2	32 bit Non-Prefetchable	0
3–5	Not used	0

For existing Qsys Avalon-MM designs created in the Quartus II 12.0 or earlier release, you must re-enable the BARs in 12.1.

For more information about the use of BARs to translate PCI Express addresses to Avalon-MM addresses, refer to "PCI Express-to-Avalon-MM Address Translation for Endpoints for 32-Bit Bridge" on page 7–20. For more information about minimizing BAR sizes, refer to "Minimizing BAR Sizes and the PCIe Address Space" on page 7–21.

3. For the **Device Identification Registers**, specify the values listed in the center column of Table 3–4. The right-hand column of this table lists the value assigned to Altera devices. You must use the Altera values to run the Altera testbench. Be sure to use your company's values for your final product.

Table 3–4. Device Identification Registers (Part 1 of 2)

Parameter	Value	Altera Value
Vendor ID	`0x00000000`	0x00001172
Device ID	`0x00000001`	0x0000E001

Table 3–4. Device Identification Registers (Part 2 of 2)

Parameter	Value	**Altera Value**
Revision ID	`0x00000001`	0x00000001
Class Code	`0x00000000`	0x00FF0000
Subsystem Vendor ID	`0x00000000`	0x00001172
Subsystem Device ID	`0x00000000`	0x0000E001

4. Under the **PCI Express and PCI Capabilities** heading, specify the settings in Table 3–5.

Table 3–5. PCI Express and PCI Capabilities

Parameter	**Value**
Device	
Maximum payload size	**128 Bytes**
Completion timeout range	**ABCD**
Implement completion timeout disable	Turn on this option
Error Reporting	
Advanced error reporting (AER)	Turn off this option
ECRC checking	Turn off this option
ECRC generation	Turn off this option
Link	
Link port number	**1**
Slot clock configuration	Turn on this option
MSI	
Number of MSI messages requested	**4**
MSI-X	
Implement MSI-X	Turn this option off
Power Management	
Endpoint L0s acceptable latency	**Maximum of 64 ns**
Endpoint L1 acceptable latency	**Maximum of 1 us**

5. Under the **Avalon-MM System Settings** heading, specify the settings in Table 3–6.

Table 3–6. Avalon Memory-Mapped System Settings

Parameter	Value
Avalon-MM width	64 bits
Peripheral Mode	Requester/Completer
Single DWord Completer	Off
Control register access (CRA) Avalon-MM Slave port	On
Enable multiple MSI/MSI-X support	Off
Auto Enable PCIe Interrupt (enabled at power-on)	Off

6. Under the **Avalon-MM to PCI Express Address Translation Settings**, specify the settings in Table 3–7.

Table 3–7. Avalon-MM to PCI Express Translation Settings

Parameter	Value
Number of address pages	2
Size of address pages	1 MByte - 20 bits

   Refer to "Avalon-MM-to-PCI Express Address Translation Algorithm for 32-Bit Addressing" on page 7–23 for more information about address translation.

7. Click **Finish**.
8. To rename the **Arria V GZ Hard IP for PCI Express**, in the **Name** column of the **System Contents** tab, right-click on the component name, select **Rename**, and type `DUT` ↵

☞ Your system is not yet complete, so you can ignore any error messages generated by Qsys at this stage.

☞ Qsys displays the values for **Posted header credit**, **Posted data credit**, **Non-posted header credit**, **Completion header credit**, and **Completion data credit** in the message area. These values are computed based upon the values set for **Maximum payload size** and **Desired performance for received requests**.

Adding the Remaining Components to the Qsys System

This section describes adding the DMA controller and on-chip memory to your system.

1. On the **Component Library** tab, type the following text string in the search box:

   `DMA` ↵

   Qsys filters the component library and shows all components matching the text string you entered.

2. Click **DMA Controller** and then click **+Add**. This component contains read and write master ports and a control port slave.

3. In the **DMA Controller** parameter editor, specify the parameters and conditions listed in the following table.

Table 3–8. DMA Controller Parameters

Parameter	Value
Width of the DMA length register	13
Enable burst transfers	Turn on this option
Maximum burst size	Select **128**
Data transfer FIFO depth	Select **32**
Construct FIFO from registers	Turn off this option
Construct FIFO from embedded memory blocks	Turn on this option
Advanced	
Allowed Transactions	Turn on all options

4. Click **Finish**. The DMA Controller module is added to your Qsys system.
5. On the **Component Library** tab, type the following text string in the search box:

```
On Chip ↵
```

Qsys filters the component library and shows all components matching the text string you entered.

6. Click **On-Chip Memory (RAM or ROM)** and then click **+Add**. Specify the parameters listed in the following table.

Table 3–9. On-Chip Memory Parameters (Part 1 of 2)

Parameter	Value
Memory Type	
Type	Select **RAM (Writeable)**
Dual-port access	Turn off this option
Single clock option	Not applicable
Read During Write Mode	Not applicable
Block type	**Auto**
Size	
Data width	**64**
Total memory size	**4096 Bytes**
Minimize memory block usage (may impact f_{MAX})	Not applicable
Read latency	
Slave s1 latency	**1**
Slave s2 latency	Not applicable
Memory initialization	
Initialize memory content	Turn on this option
Enable non-default initialization file	Turn off this option

Table 3–9. On-Chip Memory Parameters (Part 2 of 2)

Parameter	Value
Enable In-System Memory Content Editor feature D	Turn off this option
Instance ID	Not required

7. Click **Finish**.
8. The On-chip memory component is added to your Qsys system.
9. On the **File** menu, click **Save** and type the file name `ep_g1x4.qsys`. You should save your work frequently as you complete the steps in this walkthrough.
10. On the **Component Library** tab, type the following text string in the search box:

    `recon ↵`

    Qsys filters the component library and shows all components matching the text string you entered.
11. Click **Transceiver Reconfiguration Controller** and then click **+Add**. Specify the parameters listed in Table 3–10.

Table 3–10. Transceiver Reconfiguration Controller Parameters

Parameter	Value
Device family	**Arria V GZ**
Interface Bundles	
Number of reconfiguration interfaces	5
Optional interface grouping	Leave this entry blank
Transceiver Calibration Functions	
Enable offset cancellation	Leave this option on
Enable PLL calibration	Leave this option on
Create optional calibration status ports	Leave this option off
Analog Features	
Enable Analog controls	Turn this option on
Enable EyeQ block	Leave this option off
Enable decision feedback equalizer (DFE) block	Leave this option off
Enable AEQ block	Leave this option off
Reconfiguration Features	
Enable channel/PLL reconfiguration	Leave this option off
Enable PLL reconfiguration support block	Leave this option off

☞ Originally, you set the **Number of reconfiguration interfaces** to `5`. Although you must initially create a separate logical reconfiguration interface for each channel and TX PLL in your design, when the Quartus II software compiles your design, it merges logical channels. After compilation, the design has two reconfiguration interfaces, one for the TX PLL and one for the channels; however, the number of logical channels is still five.

12. Click **Finish**.

13. The Transceiver Reconfiguration Controller is added to your Qsys system.

For more information about the Transceiver Reconfiguration Controller, refer to the *Transceiver Reconfiguration Controller* chapter in the *Altera Transceiver PHY IP Core User Guide*.

Completing the Connections in Qsys

In Qsys, hovering the mouse over the **Connections** column displays the potential connection points between components, represented as dots on connecting wires. A filled dot shows that a connection is made; an open dot shows a potential connection point. Clicking a dot toggles the connection status. If you make a mistake, you can select **Undo** from the Edit menu or type `Ctrl-z`.

By default, Qsys filters some interface types to simplify the image shown on the **System Contents** tab. Complete these steps to display all interface types:

1. Click the **Filter** tool bar button.
2. In the Filter list, select **All interfaces**.
3. Close the **Filters** dialog box.

To complete the design, create the following connections:

1. Connect the pcie_sv_hip_avmm_0 `Rxm_BAR0` Avalon Memory-Mapped Master port to the onchip_memory2_0 `s1` Avalon Memory-Mapped slave port using the following procedure:
   a. Click the `Rxm_BAR0` port, then hover in the **Connections** column to display possible connections.
   b. Click the open dot at the intersection of the `onchip_mem2_0 s1` port and the pci_express_compiler `Rxm_BAR0` to create a connection.
2. Repeat step 1 to make the connections listed in Table 3–11.

Table 3–11. Qsys Connections (Part 1 of 2)

Make Connection From:	To:
DUT `nreset_status` Reset Output	onchip_memory `reset1` Avalon slave port
DUT `nreset_status` Reset Output	dma_0 `reset` Reset Input
DUT `nreset_status` Reset Output	alt_xcvr_reconfig_0 `mgmt_rst_reset` Reset Input
DUT `Rxm_BAR0` Avalon Memory Mapped Master	onchip_memory `s1` Avalon slave port
DUT `Rxm_BAR2` Avalon Memory Mapped Master	DUT `Cra` Avalon Memory Mapped Slave
DUT `Rxm_BAR2` Avalon Memory Mapped Master	dma_0 `control_port_slave` Avalon Memory Mapped Slave
DUT `RxmIrq` Interrupt Receiver	dma_0 `irq` Interrupt Sender
DUT `reconfig_to_xcvr` Conduit	alt_xcvr_reconfig_0 `reconfig_to_xcvr` Conduit
DUT `reconfig_busy` Conduit	alt_xcvr_reconfig_0 `reconfig_busy` Conduit
DUT `reconfig_from_xcvr` Conduit	alt_xcvr_reconfig_0 `reconfig_from_xcvr` Conduit
DUT `Txs` Avalon Memory Mapped Slave	dma_0 `read_master` Avalon Memory Mapped Master

Table 3–11. Qsys Connections (Part 2 of 2)

Make Connection From:	To:
DUT `Txs` Avalon Memory Mapped Slave	dma_0 `write_master` Avalon Memory Mapped Master
onchip_memory `s1` Avalon Memory Mapped Slave	dma_0 `read_master` Avalon Memory Mapped Master
DUT `nreset_status`	onchip_memory `reset1`
DUT `nreset_status`	dma_0 `reset`
DUT `nreset_status`	alt_scvr_reconfig_0 `mgmt_rst_reset`
DUT `nreset_status`	clk0 `clk_reset`

Specifying Clocks and Interrupts

Complete the following steps to connect the clocks and specify interrupts:

1. To connect DUT `coreclkout` to the **onchip_memory** and **dma_0** clock inputs, click in the **Clock** column next to the DUT `coreclkout` clock input. Click **onchip_memory.clk1** and **dma_0.clk**.
2. To connect alt_xcvr_reconfig_0 `mgmt_clk_clk` to clk_0 `clk`, click in the **Clock** column next to the alt_xcvr_reconfig_0 `mgmt_clk_clk` clock input. Click **clk_0.clk**.
3. To specify the interrupt number for DMA interrupt sender, `control_port_slave`, type `0` in the **IRQ** column next to the `irq` port.
4. On the File menu, click **Save.**

Specifying Exported Interfaces

Many interface signals in this Qsys system connect to modules outside the design. Follow these steps to export an interface:

1. Click in the **Export** column.
2. First, accept the default name that appears in the **Export** column. Then, right-click on the name, select **Rename** and type the name shown in Table 3–12.

Table 3–12. Exported Interfaces

Interface Name	Exported Name
DUT `refclk`	`refclk`
DUT `npor`	`npor`
DUT `reconfig_clk_locked`	`pcie_svhip_avmm_0_reconfig_clk_locked`
DUT `hip_serial`	`hip_serial`
DUT `hip_pipe`	`hip_pipe`
DUT `hip_ctrl`	`hip_ctrl`
alt_xcvr_reconfig_0 `reconfig_mgmt`	`alt_xcvr_reconfig_0_reconfig_mgmt`
clk_0 `clk_in`	`xcvr_reconfig_clk`
clk_0 `clk_in_reset`	`xcvr_reconfig_reset`

Specifying Address Assignments

Qsys requires that you resolve the base addresses of all Avalon-MM slave interfaces in the Qsys system. You can either use the auto-assign feature, or specify the base addresses manually. To use the auto-assign feature, on the **System** menu, click **Assign Base Addresses**. In the design example, you assign the base addresses manually.

The Avalon-MM Arria V GZ Hard IP for PCI Express assigns base addresses to each BAR. The maximum supported BAR size is 4 GByte, or 32 bits.

Follow these steps to assign a base address to an Avalon-MM slave interface manually:

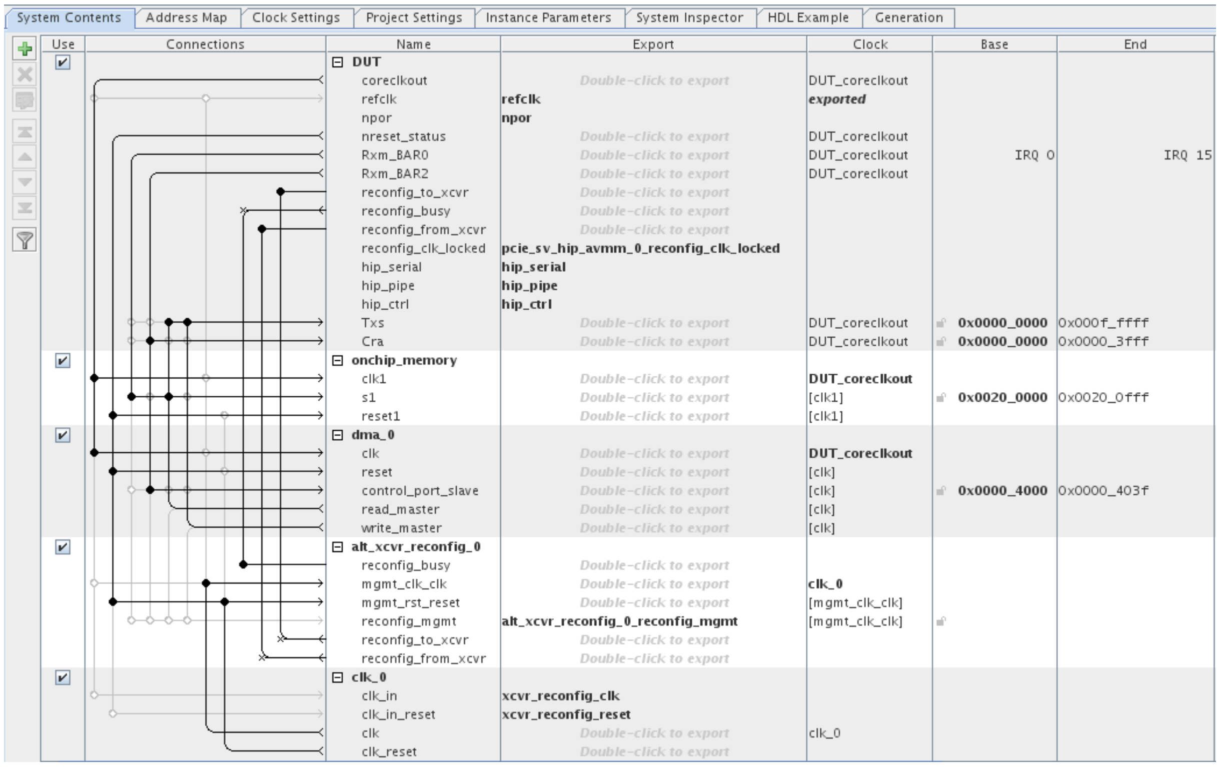
1. In the row for the Avalon-MM slave interface base address you want to specify, click the **Base** column.
2. Type your preferred base address for the interface.
3. Assign the base addresses listed in Table 3–13.

Table 3–13. Base Address Assignments for Avalon-MM Slave Interfaces

Interface Name	Exported Name
DUT `Txs`	`0x00000000`
DUT `Cra`	`0x00000000`
DMA `control_port_slave`	`0x00004000`
onchip_memory_0 `s1`	`0x00200000`

Figure 3–2

Figure 3–2. Complete PCI Express Example Design



Use	Connections	Name	Export	Clock	Base	End
✓		⊟ **DUT**				
		coreclkout	*Double-click to export*	DUT_coreclkout		
		refclk	**refclk**	***exported***		
		npor	**npor**			
		nreset_status	*Double-click to export*	DUT_coreclkout		
		Rxm_BAR0	*Double-click to export*	DUT_coreclkout	IRQ 0	IRQ 15
		Rxm_BAR2	*Double-click to export*	DUT_coreclkout		
		reconfig_to_xcvr	*Double-click to export*			
		reconfig_busy	*Double-click to export*			
		reconfig_from_xcvr	*Double-click to export*			
		reconfig_clk_locked	**pcie_sv_hip_avmm_0_reconfig_clk_locked**			
		hip_serial	**hip_serial**			
		hip_pipe	**hip_pipe**			
		hip_ctrl	**hip_ctrl**			
		Txs	*Double-click to export*	DUT_coreclkout	**0x0000_0000**	0x000f_ffff
		Cra	*Double-click to export*	DUT_coreclkout	**0x0000_0000**	0x0000_3fff
✓		⊟ **onchip_memory**				
		clk1	*Double-click to export*	**DUT_coreclkout**		
		s1	*Double-click to export*	[clk1]	**0x0020_0000**	0x0020_0fff
		reset1	*Double-click to export*	[clk1]		
✓		⊟ **dma_0**				
		clk	*Double-click to export*	**DUT_coreclkout**		
		reset	*Double-click to export*	[clk]		
		control_port_slave	*Double-click to export*	[clk]	**0x0000_4000**	0x0000_403f
		read_master	*Double-click to export*	[clk]		
		write_master	*Double-click to export*	[clk]		
✓		⊟ **alt_xcvr_reconfig_0**				
		reconfig_busy	*Double-click to export*			
		mgmt_clk_clk	*Double-click to export*	**clk_0**		
		mgmt_rst_reset	*Double-click to export*	[mgmt_clk_clk]		
		reconfig_mgmt	**alt_xcvr_reconfig_0_reconfig_mgmt**	[mgmt_clk_clk]		
		reconfig_to_xcvr	*Double-click to export*			
		reconfig_from_xcvr	*Double-click to export*			
✓		⊟ **clk_0**				
		clk_in	**xcvr_reconfig_clk**			
		clk_in_reset	**xcvr_reconfig_reset**			
		clk	*Double-click to export*	clk_0		
		clk_reset	*Double-click to export*			

For this example BAR1:0 is 22 bits or 4 MBytes. This BAR accesses Avalon addresses from 0x00200000– 0x00200FFF. BAR2 is 15 bits or 32 KBytes. BAR2 accesses the DMA control_port_slave at offsets 0x00004000 through 0x0000403F. The pci_express `CRA` slave port is accessible at offsets 0x0000000–0x0003FFF from the programmed BAR2 base address. For more information on optimizing BAR sizes, refer to "Minimizing BAR Sizes and the PCIe Address Space" on page 7–21.

Simulating the Example Design

Follow these steps to generate the files for the testbench and synthesis.

1. On the **Generation** tab, in the **Simulation** section, set the following options:
   a. For **Create simulation model**, select **None**. (This option allows you to create a simulation model for inclusion in your own custom testbench.)
   b. For **Create testbench Qsys system**, select **Standard, BFMs for standard Avalon interfaces**.
   c. For **Create testbench simulation model**, select **Verilog**.
2. In the **Synthesis** section, turn on **Create HDL design files for synthesis**.
3. Click the **Generate** button at the bottom of the tab.
4. After Qsys reports **Generate Completed** in the **Generate** progress box title, click **Close**.
5. On the **File** menu, click **Save**. and type the file name `ep_g1x4.qsys`.

Table 3–14 lists the directories that are generated in your Quartus II project directory.

Table 3–14. Qsys System Generated Directories

Directory	Location
Qsys system	*<project_dir>*/**ep_g1x4**
Testbench	*<project_dir>*/**ep_g1x4/testbench**
Synthesis	*<project_dir>*/**ep_g1x4/synthesis**

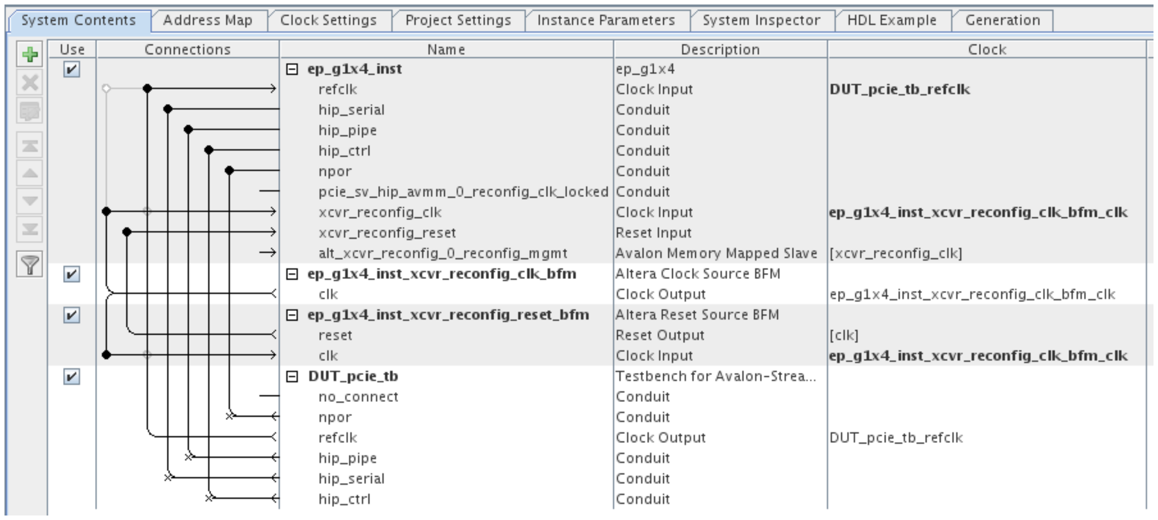
Qsys creates a top-level testbench named *<project_dir>*/**ep_g1x4/testbench/ ep_g1x4_tb.qsys.** This testbench connects an appropriate BFM to each exported interface. Qsys generates the required files and models to simulate your PCI Express system.

The simulation of the design example uses the following components and software:

- The system you created using Qsys
- A testbench created by Qsys in the *<project_dir>*/**ep_g1_x4/testbench** directory. You can view this testbench in Qsys by opening *<project_dir>*/**ep_g1_x4/testbench/ s5_avmm_tb.qsys** which shown in Figure 3–3.
- The ModelSim software

You can also use any other supported third-party simulator to simulate your design.

Figure 3–3. Qsys Testbench for the PCI Example Design



Use	Connections	Name	Description	Clock
✓		ep_g1x4_inst	ep_g1x4	
		refclk	Clock Input	DUT_pcie_tb_refclk
		hip_serial	Conduit	
		hip_pipe	Conduit	
		hip_ctrl	Conduit	
		npor	Conduit	
		pcie_sv_hip_avmm_0_reconfig_clk_locked	Conduit	
		xcvr_reconfig_clk	Clock Input	ep_g1x4_inst_xcvr_reconfig_clk_bfm_clk
		xcvr_reconfig_reset	Reset Input	
		alt_xcvr_reconfig_0_reconfig_mgmt	Avalon Memory Mapped Slave	[xcvr_reconfig_clk]
✓		ep_g1x4_inst_xcvr_reconfig_clk_bfm	Altera Clock Source BFM	
		clk	Clock Output	ep_g1x4_inst_xcvr_reconfig_clk_bfm_clk
✓		ep_g1x4_inst_xcvr_reconfig_reset_bfm	Altera Reset Source BFM	
		reset	Reset Output	[clk]
		clk	Clock Input	ep_g1x4_inst_xcvr_reconfig_clk_bfm_clk
✓		DUT_pcie_tb	Testbench for Avalon-Strea...	
		no_connect	Conduit	
		npor	Conduit	
		refclk	Clock Output	DUT_pcie_tb_refclk
		hip_pipe	Conduit	
		hip_serial	Conduit	
		hip_ctrl	Conduit	

Qsys creates IP functional simulation models for all the system components. The IP functional simulation models are the **.vo** or **.vho** files generated by Qsys in your project directory.

For more information about IP functional simulation models, refer to *Simulating Altera Designs* in volume 3 of the *Quartus II Handbook*.

Complete the following steps to run the Qsys testbench:

1. In a terminal window, change to the *<project_dir>***/ep_g1x4/testbench/mentor** directory.
2. Start the ModelSim simulator.
3. To run the simulation, type the following commands in a terminal window:
   a. `do msim_setup.tcl` ↵
   b. `ld_debug` ↵ (The -debug argument stops optimizations, improving visibility in the ModelSim waveforms.)
   c. `run 140000 ns` ↵

The driver performs the following transactions with status of the transactions displayed in the ModelSim simulation message window:

- Various configuration accesses to the Avalon-MM Arria V GZ Hard IP for PCI Express in your system after the link is initialized
- Setup of the Address Translation Table for requests that are coming from the DMA component
- Setup of the DMA controller to read 512 Bytes of data from the Transaction Layer Direct BFM's shared memory

- Setup of the DMA controller to write the same data back to the Transaction Layer Direct BFM's shared memory
- Data comparison and report of any mismatch

Example 3–1 shows the transcript from a successful simulation run.

Example 3–1. Transcript from ModelSim Simulation of Gen1 x4 Endpoint

```
# 464 ns Completed initial configuration of Root Port.
# INFO:       2657 ns  EP LTSSM State: DETECT.ACTIVE
# INFO:       3661 ns  RP LTSSM State: DETECT.ACTIVE
# INFO:       6049 ns  EP LTSSM State: POLLING.ACTIVE
# INFO:       6909 ns  RP LTSSM State: POLLING.ACTIVE
# INFO:       9037 ns  RP LTSSM State: POLLING.CONFIG
# INFO:       9441 ns  EP LTSSM State: POLLING.CONFIG
# INFO:      10657 ns  EP LTSSM State: CONFIG.LINKWIDTH.START
# INFO:      10829 ns  RP LTSSM State: CONFIG.LINKWIDTH.START
# INFO:      11713 ns  EP LTSSM State: CONFIG.LINKWIDTH.ACCEPT
# INFO:      12253 ns  RP LTSSM State: CONFIG.LINKWIDTH.ACCEPT
# INFO:      12573 ns  RP LTSSM State: CONFIG.LANENUM.WAIT
# INFO:      13505 ns  EP LTSSM State: CONFIG.LANENUM.WAIT
# INFO:      13825 ns  EP LTSSM State: CONFIG.LANENUM.ACCEPT
# INFO:      13853 ns  RP LTSSM State: CONFIG.LANENUM.ACCEPT
# INFO:      14173 ns  RP LTSSM State: CONFIG.COMPLETE
# INFO:      14721 ns  EP LTSSM State: CONFIG.COMPLETE
# INFO:      16001 ns  EP LTSSM State: CONFIG.IDLE
# INFO:      16093 ns  RP LTSSM State: CONFIG.IDLE
# INFO:      16285 ns  RP LTSSM State: L0
# INFO:          16545 ns  EP LTSSM State: L0
# INFO:          19112 ns Configuring Bus 001, Device 001, Function 00
# INFO:          19112 ns   EP Read Only Configuration Registers:
# INFO:       19112 ns           Vendor ID: 0000
# INFO:       19112 ns           Device ID: 0001
# INFO:       19112 ns         Revision ID: 01
# INFO:       19112 ns          Class Code: 000000
# INFO:       19112 ns    Subsystem Vendor ID: 0000
# INFO:       19112 ns         Subsystem ID: 0000
# INFO:          19112 ns              Interrupt Pin: INTA# used
# INFO:      20584 ns  PCI MSI Capability Register:
# INFO:      20584 ns   64-Bit Address Capable: Supported
# INFO:          20584 ns         Messages Requested:  4
#INFO:      28136 ns  EP PCI Express Link Status Register (1041):
# INFO:      28136 ns    Negotiated Link Width: x4
# INFO:     28136 ns     Slot Clock Config: System Reference Clock Used
# INFO:      29685 ns  RP LTSSM State: RECOVERY.RCVRLOCK
# INFO:      30561 ns  EP LTSSM State: RECOVERY.RCVRLOCK
# INFO:      31297 ns  EP LTSSM State: RECOVERY.RCVRCFG
# INFO:      31381 ns  RP LTSSM State: RECOVERY.RCVRCFG
# INFO:      32661 ns  RP LTSSM State: RECOVERY.IDLE
# INFO:      32961 ns  EP LTSSM State: RECOVERY.IDLE
# INFO:       33153 ns  EP LTSSM State: L0
# INFO:       33237 ns  RP LTSSM State: L0
# INFO:          34696 ns        Current Link Speed: 2.5GT/s
INFO:        34696 ns
# INFO:      36168 ns  EP PCI Express Link Control Register (0040):
# INFO:     36168 ns    Common Clock Config: System Reference Clock Used
# INFO:       36168 ns
# INFO:       37960 ns
```

Example 3–1. Transcript from ModelSim Simulation of Gen1 x4 Endpoint (continued)

```
# INFO:      37960 ns  EP PCI Express Capabilities Register (0002):
# INFO:      37960 ns      Capability Version: 2
# INFO:      37960 ns              Port Type: Native Endpoint
# INFO:      37960 ns  EP PCI Express Device Capabilities Register (00008020):
# INFO:      37960 ns    Max Payload Supported: 128 Bytes
# INFO:      37960 ns           Extended Tag: Supported
# INFO:      37960 ns   Acceptable L0s Latency: Less Than 64 ns
# INFO:      37960 ns   Acceptable L1  Latency: Less Than 1 us
# INFO:      37960 ns       Attention Button: Not Present
# INFO:      37960 ns    Attention Indicator: Not Present
# INFO:      37960 ns        Power Indicator: Not Present
# INFO:      37960 ns  EP PCI Express Link Capabilities Register (01406041):
# INFO:      37960 ns      Maximum Link Width: x4
# INFO:      37960 ns    Supported Link Speed: 2.5GT/s
# INFO:      37960 ns              L0s Entry: Not Supported
# INFO:      37960 ns           L1  Entry: Not Supported
# INFO:      37960 ns       L0s Exit Latency: 2 us to 4 us
# INFO:      37960 ns       L1  Exit Latency: Less Than 1 us
# INFO:      37960 ns          Port Number: 01
# INFO:      37960 ns  Surprise Dwn Err Report: Not Supported
# INFO:      37960 ns  DLL Link Active Report: Not Supported
# INFO:      37960 ns
# INFO:      37960 ns   EP PCI Express Device Capabilities 2 Register (0000001F):
# INFO:      37960 ns  Completion Timeout Rnge: ABCD (50us to 64s)
# INFO:      39512 ns
# INFO:      39512 ns  EP PCI Express Device Control Register (0110):
# INFO:      39512 ns  Error Reporting Enables: 0
# INFO:      39512 ns          Relaxed Ordering: Enabled
# INFO:      39512 ns  Error Reporting Enables: 0
# INFO:      39512 ns       Relaxed Ordering: Enabled
# INFO:      39512 ns           Max Payload: 128 Bytes
# INFO:      39512 ns          Extended Tag: Enabled
# INFO:      39512 ns       Max Read Request: 128 Bytes
# INFO:      39512 ns
# INFO:      39512 ns  EP PCI Express Device Status Register (0000):
# INFO:      39512 ns
# INFO:      41016 ns  EP PCI Express Virtual Channel Capability:
# INFO:      41016 ns       Virtual Channel: 1
# INFO:      41016 ns       Low Priority VC: 0
# INFO:      41016 ns
# INFO:      46456 ns
# INFO:      46456 ns BAR Address Assignments:
# INFO:      46456 ns BAR   Size    Assigned Address  Type
# INFO:      46456 ns ---   ----    ----------------
# INFO:      46456 ns BAR1:0  4 MBytes 00000001 00000000 Prefetchable
# INFO:      46456 ns BAR2   32 KBytes      00200000 Non-Prefetchable
# INFO:      46456 ns BAR3   Disabled
# INFO:      46456 ns BAR4   Disabled
# INFO:      46456 ns BAR5   Disabled
# INFO:      46456 ns ExpROM Disabled
INFO:        48408 ns
# INFO:      48408 ns Completed configuration of Endpoint BARs.
# INFO:      50008 ns Starting Target Write/Read Test.
# INFO:      50008 ns  Target BAR = 0
# INFO:      50008 ns  Length = 000512, Start Offset = 000000
# INFO:      54368 ns  Target Write and Read compared okay!
# INFO:      54368 ns Starting DMA Read/Write Test.
```

Example 3–1. Transcript from ModelSim Simulation of Gen1 x4 Endpoint (continued)

```
# INFO:       54368 ns  Setup BAR = 2
# INFO:      54368 ns  Length = 000512, Start Offset = 000000
# INFO:     60609 ns Interrupt Monitor: Interrupt INTA Asserted
# INFO:      60609 ns Clear Interrupt INTA
# INFO:     62225 ns Interrupt Monitor: Interrupt INTA Deasserted
# INFO:      69361 ns MSI recieved!
# INFO:     69361 ns  DMA Read and Write compared okay!
# SUCCESS: Simulation stopped due to successful completion!
# Break at ./..//ep_g1x4_tb/simulation/submodules//altpcietb_bfm_log.v line 78
```

Simulating the Single DWord Design

You can use the same testbench to simulate the **Completer-Only single dword** IP core by changing the settings in the driver file. Complete the following steps for the Verilog HDL design example:

1. In a terminal window, change to the *<project_dir>*/*<variant>***/testbench/** *<variant>***_tb/simulation/submodules** directory.
2. Open **altpcietb_bfm_driver_avmm.v** file your text editor.
3. To enable target memory tests and specify the completer-only single dword variant, specify the following parameters:
   - `parameter RUN_TGT_MEM_TST = 1;`
   - `parameter RUN_DMA_MEM_TST = 0;`
   - `parameter AVALON_MM_LITE = 1;`
4. Change to the *<project_dir>*/*<variant>***/testbench/mentor** directory.
5. Start the ModelSim simulator.
6. To run the simulation, type the following commands in a terminal window:
   a. `do msim_setup.tcl` ↵
   b. `ld_debug` ↵ (The -debug suffix stops optimizations, improving visibility in the ModelSim waveforms.)
   c. `run 140000 ns` ↵

Understanding Channel Placement Guidelines

Arria V GZ transceivers are organized in banks of six channels. The transceiver bank boundaries are important for clocking resources, bonding channels, and fitting. Refer to the channel placement figures following "Serial Interface Signals" on page 8–60 for illustrations of channel placement for ×1, ×2, ×4, and ×8 variants using both CMU and ATX PLLs.

For more information about transceiver clocking and channel placement refer to "Transceiver Clocking and Channel Placement Guidelines" in *Transceiver Configurations in Arria V GZ Devices*.

Adding Synopsis Design Constraints

Before you can compile your design using the Quartus II software, you must add a few Synopsys Design Constraints (SDC) to your project. Complete the following steps to add these constraints:

1. Browse to *<project_dir>*/**ep_g1x4/synthesis/submodules**.
2. Add the constraints shown inExample 3–2 to **altera_pci_express.sdc**.

Example 3–2. Synopsys Design Constraints

```
create_clock -period “100 MHz” -name {refclk_pci_express} {*refclk_*}
create_clock -period "125 MHz" -name {reconfig_xcvr_clk}
{*reconfig_xcvr_clk*}
derive_pll_clocks
derive_clock_uncertainty
```

Because **altera_pci_express.sdc** is overwritten each time you regenerate your design, you should save a copy of this file in an additional directory that the Quartus II software does not overwrite.

Creating a Quartus II Project

You can create a new Quartus II project with the New Project Wizard, which helps you specify the working directory for the project, assign the project name, and designate the name of the top-level design entity. To create a new project follow these steps:

1. On the Quartus II File menu, click **New,** then **New Quartus II Project**, then **OK**.
2. Click **Next** in the **New Project Wizard: Introduction** (The introduction does not appear if you previously turned it off.)
3. On the **Directory, Name, Top-Level Entity** page, enter the following information:
   a. For What is the working directory for this project, browse to *<project_dir>*/**ep_g1x4/synthesis/**
   b. For **What is the name of this project**, select **ep_g1x4** from the **synthesis** directory.
4. Click **Next**.
5. On the **Add Files** page, add *<project_dir>*/**ep_g1x4/synthesis/ep_ge1_x4.qip** to your Quartus II project. This file lists all necessary files for Quartus II compilation, including the **altera_pci_express.sdc** that you just modified.
6. Click **Next** to display the **Family & Device Settings** page.
7. On the **Device** page, choose the following target device family and options:
   a. In the **Family** list, select **Arria V GZ**.
   b. In the **Devices** list, select **All**.
   c. In the **Available devices** list, select **5AGZME5K2F40C3.**
8. Click **Next** to close this page and display the **EDA Tool Settings** page.

9. From the **Simulation** list, select **ModelSim**®. From the **Format** list, select the HDL language you intend to use for simulation.
10. Click **Next** to display the **Summary** page.
11. Check the **Summary** page to ensure that you have entered all the information correctly.

Compiling the Design

Follow these steps to compile your design:

1. On the Quartus II Processing menu, click **Start Compilation**.
2. After compilation, expand the **TimeQuest Timing Analyzer** folder in the Compilation Report. Note whether the timing constraints are achieved in the Compilation Report.

   If your design does not initially meet the timing constraints, you can find the optimal Fitter settings for your design by using the Design Space Explorer. To use the Design Space Explorer, click **Launch Design Space Explorer** on the tools menu.

Programming a Device

After you compile your design, you can program your targeted Altera device and verify your design in hardware.

For more information about programming Altera FPGAs, refer to *Quartus II Programmer*.

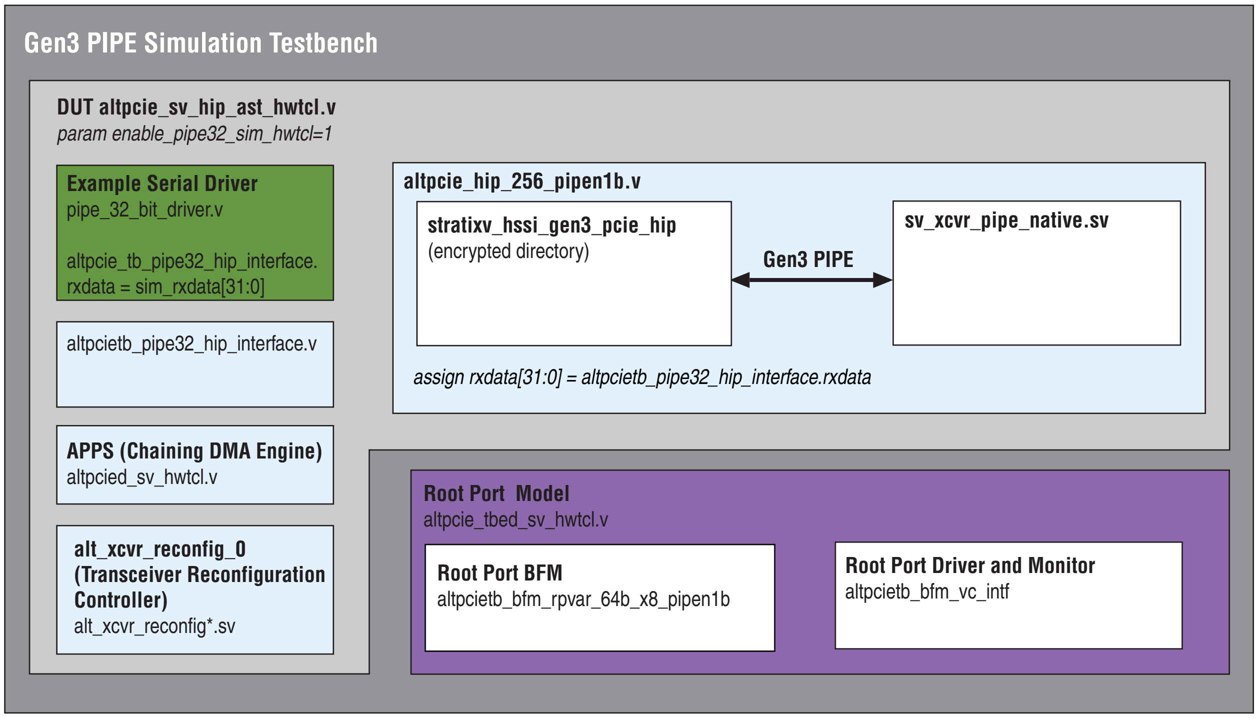
4. Getting Started with the Gen3 PIPE Simulation

This Quartus II software release introduces PIPE simulation for Gen3 Endpoints. Simulation using the PIPE interface is significantly faster than the previously available serial interface.

Altera provides Gen3 example designs in the <*install_dir*>**/ip/altera/altera_pcie/altera_pcie_hip_ast_ed/example_design/**<*device*> directory. After you install the Quartus II software, you can copy the design examples from the <*install_dir*>**/ip/altera/altera_pcie/altera_pcie_hip_ast_ed/example_design/**<*device*> directory.

The Gen3 example designs include a serial driver for the simulation testbench which is shown in green in the following figure. The example designs also include Altera's Root Port BFM, shown in purple. After running Altera's simulation testbench, you can replace these modules with your own driver and BFM to ensure full verification of your Gen3 Endpoint.

Figure 1. Gen3 Example Design



Gen3 PIPE Simulation Testbench
DUT altpcie_sv_hip_ast_hwtcl.v
param enable_pipe32_sim_hwtcl=1
Example Serial Driver
pipe_32_bit_driver.v
altpcie_tb_pipe32_hip_interface.
rxdata = sim_rxdata[31:0]
altpcietb_pipe32_hip_interface.v
APPS (Chaining DMA Engine)
altpcied_sv_hwtcl.v
alt_xcvr_reconfig_0
(Transceiver Reconfiguration Controller)
alt_xcvr_reconfig*.sv
altpcie_hip_256_pipen1b.v
stratixv_hssi_gen3_pcie_hip
(encrypted directory)
Gen3 PIPE
sv_xcvr_pipe_native.sv
assign rxdata[31:0] = altpcietb_pipe32_hip_interface.rxdata
Root Port Model
altpcie_tbed_sv_hwtcl.v
Root Port BFM
altpcietb_bfm_rpvar_64b_x8_pipen1b
Root Port Driver and Monitor
altpcietb_bfm_vc_intf

Generating the Gen3 PIPE Simulation

You can generate and run the simulation testbench for the **pcie_de_gen3_x8_ast256.qsys** design available in the Quartus II 12.1 SP1 installation to become familiar with the Gen3 PIPE simulation.

Follow these steps to generate the Gen3 PIPE simulation:

1. Copy the Qsys design to a working directory. This example uses: *<working_dir>*/**gen3x8.**
2. By default, the top-level Qsys file, **pcie_de_gen3_x8_ast256.qsys**, does not enable the Example Serial Driver for Gen3 PIPE simulation. Complete the following steps to enable the Gen3 PIPE32 simulation:
   a. Open **pcie_de_gen3_x8_ast256.qsys**.
   b. Locate the parameter, `enable_pipe32_sim_hwtcl`, and change its value to 1.
   c. Save **pcie_de_gen3_x8_ast256.qsys**.
3. In Qsys, open **pcie_de_gen3_x8_ast256.qsys**.
4. On the Qsys **Generation** tab, specify the parameters listed in the following table.

Table 1. Parameters to Specify on the Generation Tab in Qsys

Parameter	Value
Simulation	
Create simulation model	**None.** (This option generates a simulation model you can include in your own custom testbench.)
Create testbench Qsys system	**Standard, BFMs for standard Avalon interfaces**
Create testbench simulation model	**Verilog**
Synthesis	
Create HDL design files for synthesis	Turn this option off
Create block symbol file (.bsf)	Turn this option on
Output Directory	
Path	*<working_dir>/***gen3x8/pcie_de_gen3_x8_ast256**
Simulation	Leave this option blank
Testbench	*<working_dir>/***gen3x8/pcie_de_gen3_x8_ast256/testbench**
Synthesis	Leave this option blank

5. Click the **Generate** button at the bottom of the **Generation** tab to generate the chaining DMA testbench.

Enabling the Gen3 PIPE Simulation

By default the testbench does not enable the example driver for the Gen3 PIPE interface. After running the Gen3 PIPE simulation using Altera’s example driver, you should create your own driver, testbench, and BFM for the Gen3 PIPE interface.

The parameter, `enable_pipe32_phyip_ser_driver_hwtcl`, in **pcie_de_gen3_x8_ast256_tb.v** enables Gen3 PIPE simulation. Complete the following steps to enable the Gen3 PIPE simulation:

1. Change to the testbench simulation directory by typing the following command:
   `cd <working_dir>/gen3x8/pcie_de_gen3_x8_ast256/testbench/pcie_de_gen3_x8_ast256_tb/simulation/` ↵
2. Open **pcie_de_gen3_x8_ast256_tb.v**.
3. Locate the parameter, `enable_pipe32_phyip_ser_driver_hwtcl = 0`, and change the 0 to a 1. Save **pcie_de_gen3_x8_ast256_tb.v**.

Simulating the Gen3 x8 Testbench

Follow these steps to compile the testbench for simulation and run the chaining DMA testbench.

1. Start your simulation tool. This example uses the Synopsys® software. Currently, the ModelSim simulator does not support all of the constructs necessary for this simulation.
2. In the Synopsys testbench directory, *<working_dir>***/gen3x8/pcie_de_gen3_x8_ast256/testbench/synopsys/vcs**,
   you must change the `USER_DEFINED_SIM_OPTIONS` variable to run the simulation by completing the following steps:
   a. Open **vcs_setup.sh**.
   b. Change the value of `USER_DEFINED_SIM_OPTIONS to "+vcs"` by deleting the text `+finish+100`.
   c. Save **vcs_setup.sh**.
3. `Type the following command to run the simulation: source vcs_setup.sh` ↵

The following example shows a transcript from a successful simulation.

As this transcript illustrates, the simulation includes the following stages:

- Link training
- Configuration
- DMA reads and writes

Example 4–1. Transcript from Successful VCS Simulation

```
INFO:            1533 ns  EP LTSSM State: DETECT.ACTIVE
INFO:            2445 ns  EP LTSSM State: POLLING.ACTIVE
INFO:            4225 ns  RP LTSSM State: DETECT.ACTIVE
INFO:            5313 ns  RP LTSSM State: POLLING.ACTIVE
INFO:            8065 ns  RP LTSSM State: POLLING.CONFIG
INFO:            8173 ns  EP LTSSM State: POLLING.CONFIG
INFO:            9581 ns  EP LTSSM State: CONFIG.LINKWIDTH.START
INFO:            9665 ns  RP LTSSM State: CONFIG.LINKWIDTH.START
INFO:           10285 ns  EP LTSSM State: CONFIG.LINKWIDTH.ACCEPT
INFO:           10785 ns  RP LTSSM State: CONFIG.LINKWIDTH.ACCEPT
INFO:           11105 ns  RP LTSSM State: CONFIG.LANENUM.WAIT
INFO:           11629 ns  EP LTSSM State: CONFIG.LANENUM.WAIT
INFO:           11949 ns  EP LTSSM State: CONFIG.LANENUM.ACCEPT
```

Example 4–1. Transcript from Successful VCS Simulation (continued)

```
INFO:           12941 ns  EP LTSSM State: CONFIG.COMPLETE
INFO:            1533 ns  EP LTSSM State: DETECT.ACTIVE
INFO:            2445 ns  EP LTSSM State: POLLING.ACTIVE
INFO:            4225 ns  RP LTSSM State: DETECT.ACTIVE
INFO:            5313 ns  RP LTSSM State: POLLING.ACTIVE
INFO:            8065 ns  RP LTSSM State: POLLING.CONFIG
INFO:            8173 ns  EP LTSSM State: POLLING.CONFIG
INFO:            9581 ns  EP LTSSM State: CONFIG.LINKWIDTH.START
INFO:            9665 ns  RP LTSSM State: CONFIG.LINKWIDTH.START
INFO:           10285 ns  EP LTSSM State: CONFIG.LINKWIDTH.ACCEPT
INFO:           10785 ns  RP LTSSM State: CONFIG.LINKWIDTH.ACCEPT
INFO:           11105 ns  RP LTSSM State: CONFIG.LANENUM.WAIT
INFO:           11629 ns  EP LTSSM State: CONFIG.LANENUM.WAIT
INFO:           11949 ns  EP LTSSM State: CONFIG.LANENUM.ACCEPT
INFO:           12065 ns  RP LTSSM State: CONFIG.LANENUM.ACCEPT
INFO:           12385 ns  RP LTSSM State: CONFIG.COMPLETE
INFO:           12941 ns  EP LTSSM State: CONFIG.COMPLETE
INFO:           14433 ns  RP LTSSM State: CONFIG.IDLE
INFO:           15821 ns  EP LTSSM State: CONFIG.IDLE
INFO:           16013 ns  EP LTSSM State: L0
INFO:           16257 ns  RP LTSSM State: L0
INFO:           16520 ns Configuring Bus 000, Device 000, Function 00
INFO:           16520 ns   RP Read Only Configuration Registers:
INFO:           16520 ns                 Vendor ID: 1172
INFO:           16520 ns                 Device ID: E001
INFO:           16520 ns               Revision ID: 01
INFO:           16520 ns                Class Code: FF0000
INFO:           16520 ns             Interrupt Pin: INTA# used
INFO:           16520 ns
INFO:           17200 ns   RP Base Address Registers:
INFO:           17200 ns
INFO:           17200 ns BAR Address Assignments:
INFO:           17200 ns BAR    Size       Assigned Address  Type
INFO:           17200 ns ---    ----       ----------------
INFO:           17200 ns BAR0   Disabled
INFO:           17200 ns BAR1   Disabled
INFO:           17200 ns ExpROM Disabled
INFO:           17200 ns
INFO:           17200 ns    I/O Base and Limit Register: Disable
INFO:           17200 ns    Prefetchable Base and Limit Register: Disable
INFO:           17200 ns
INFO:           17272 ns   PCI MSI Capability Register:
INFO:           17272 ns    64-Bit Address Capable: Supported
INFO:           17272 ns        Messages Requested:  4
INFO:           17272 ns
INFO:           17416 ns    RP PCI Express Slot Capability Register (00040000):
INFO:           17416 ns        Attention Button: Not Present
INFO:           17416 ns        Power Controller: Not Present
INFO:           17416 ns              MRL Sensor: Not Present
INFO:           17416 ns     Attention Indicator: Not Present
INFO:           17416 ns         Power Indicator: Not Present
INFO:           17416 ns       Hot-Plug Surprise: Not Supported
INFO:           17416 ns        Hot-Plug Capable: Not Supported
INFO:           17416 ns         Slot Power Limit Value: 0
INFO:           17416 ns         Slot Power Limit Scale: 0
INFO:           17416 ns           Physical Slot Number: 0
```

Example 4–1. Transcript from Successful VCS Simulation (continued)

```
INFO:          17560 ns   RP PCI Express Link Status Register (1081):
INFO:          17560 ns     Negotiated Link Width: x8
INFO:          17560 ns         Slot Clock Config: System Reference Clock Used
INFO:          18113 ns  RP LTSSM State: RECOVERY.RCVRLOCK
INFO:          18829 ns  EP LTSSM State: RECOVERY.RCVRLOCK
INFO:          19597 ns  EP LTSSM State: RECOVERY.RCVRCFG
INFO:          20513 ns  RP LTSSM State: RECOVERY.RCVRCFG
INFO:          22689 ns  RP LTSSM State: RECOVERY.SPEED
INFO:          23021 ns  EP LTSSM State: RECOVERY.SPEED
INFO:          29560 ns            New Link Speed: 8.0GT/s
INFO:          29632 ns   RP PCI Express Link Control Register (0040):
INFO:          29632 ns       Common Clock Config: System Reference Clock Used
INFO:          30752 ns   RP PCI Express Capabilities Register (0042):
INFO:          30752 ns        Capability Version: 2
INFO:          30752 ns                 Port Type: Root Port
INFO:          30752 ns
INFO:          30752 ns   RP PCI Express Device Capabilities Register (10008003):
INFO:          30752 ns     Max Payload Supported: 1KBytes
INFO:          30752 ns              Extended Tag: Not Supported
INFO:          30752 ns    Acceptable L0s Latency: Less Than 64 ns
INFO:          30752 ns    Acceptable L1  Latency: Less Than 1 us
INFO:          30752 ns          Attention Button: Not Present
INFO:          30752 ns       Attention Indicator: Not Present
INFO:          30752 ns           Power Indicator: Not Present
INFO:          30752 ns   RP PCI Express Link Capabil
INFO:          30752 ns        Maximum Link Width: x8
INFO:          30752 ns      Supported Link Speed: 8.0GT/s or 5.0GT/s or 2.5GT/s
INFO:          30752 ns                 L0s Entry: Supported
INFO:          30752 ns                 L1  Entry: Not Supported
INFO:          30752 ns          L0s Exit Latency: 2 us to 4 us
INFO:          30752 ns          L1  Exit Latency: Less Than 1 us
INFO:          30752 ns               Port Number: 01
INFO:          30752 ns   Surprise Dwn Err Report: Not Supported
INFO:          30752 ns    DLL Link Active Report: Not Supported
INFO:          30752 ns   RP PCI Express Device Capabilities 2 Register (0010001F):
INFO:          30752 ns   Completion Timeout Rnge: ABCD (50us to 64s)
INFO:          30880 ns   RP PCI Express Device Control Register (5030):
INFO:          30880 ns   Error Reporting Enables: 0
INFO:          30880 ns          Relaxed Ordering: Enabled
INFO:          30880 ns               Max Payload: 256 Bytes
INFO:          30880 ns              Extended Tag: Disabled
INFO:          30880 ns          Max Read Request: 4KBytes
INFO:          30880 ns   RP PCI Express Device Status Register (0000):
INFO:          30880 ns
INFO:          30952 ns   RP PCI Express Virtual Channel Capability:
INFO:          30952 ns          Virtual Channel: 1
INFO:          30952 ns          Low Priority VC: 0
INFO:          30952 ns
INFO:          34864 ns Completed configuration of Endpoint BARs.
INFO:          35944 ns TASK:downstream_loop
INFO:         36776 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:         37592 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:    38440 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:    39248 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:    40064 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:    40888 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:    41720 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:    42536 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:    43352 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
```

Replacing the Example Serial Driver

The Verilog HDL code to enable the Example Serial Driver is included in *<working_dir>*/**gen3x8/pcie_de_gen3_x8_ast256**/**testbench/pcie_de_gen3_x8_ast256_tb/simulation/submodules/altpcie_tbed_sv_hwtcl.v**. The parameter, `enable_pipe32_phyip_ser_driver_hwtcl`, enables the Example Serial Driver.

The following example shows the code that instantiates this driver when it is enabled.

Example 1. Instantiating the Example Serial Driver

```
if (enable_pipe32_phyip_ser_driver_hwtcl==1) begin
         assign serdes_rx_serial_data =
{rx_in7,rx_in6,rx_in5,rx_in4,rx_in3,rx_in2,rx_in1,rx_in0 };
         altpcietb_pipe32_driver # (
            .LANES(LANES),
            .gen123_lane_rate_mode((gen123_lane_rate_mode_hwtcl=="Gen3 (8.0
Gbps)")?"gen1_gen2_gen3": (gen123_lane_rate_mode_hwtcl=="Gen2 (5.0
Gbps)")?"gen1_gen2":"gen1"),
            .pll_refclk_freq( "100 MHz")
         ) altpcietb_pipe32_driver (
            .refclk(refclk),
            .npor(npor),
            .serdes_rx_serial_data(serdes_rx_serial_data[LANES-1:0]),
            .serdes_tx_serial_data(serdes_tx_serial_data[LANES-1:0])
         );
      end
```

You can modify this Verilog HDL file to instantiate your own driver.

5. Getting Started with the Configuration Space Bypass Mode Qsys Example Design

This Qsys design example demonstrates Configuration Space Bypass mode for the Arria V GZ Hard IP for PCI Express IP Core. A Root Port BFM provides stimulus to the Endpoint design. The Endpoint bypasses the standard Configuration Space to access the custom Configuration Space and memory of two functions. The Configuration Space Bypass Example Design performs the following functions:

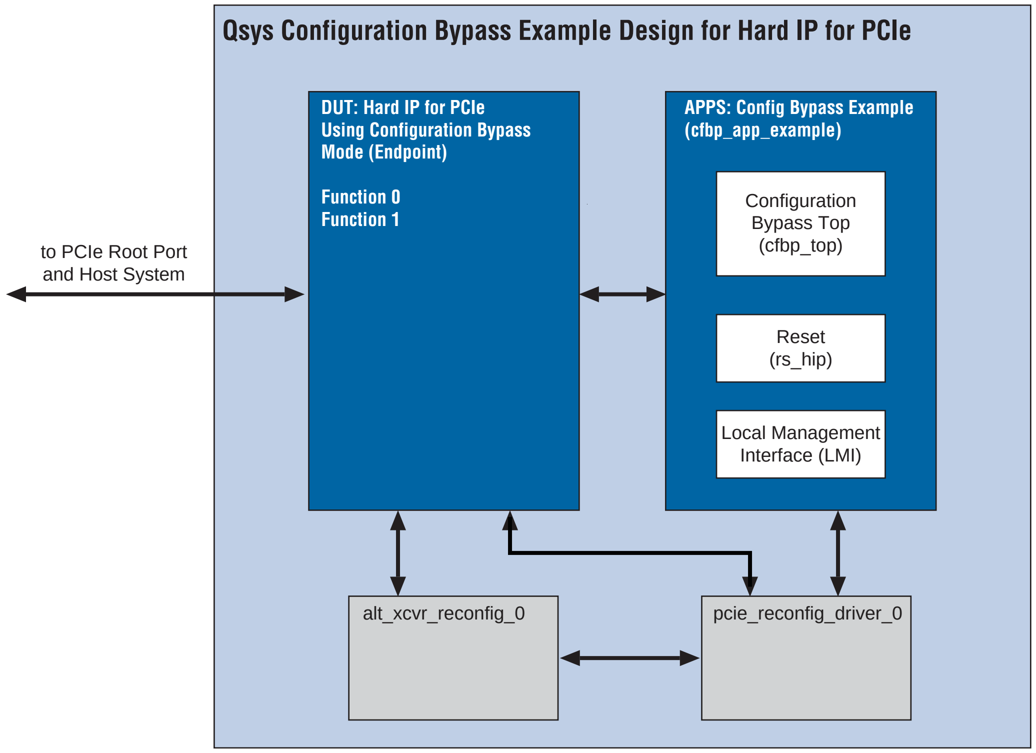
- Accepts Configuration, Memory, and Message TLPs on the Arria V GZ Hard IP for PCI Express RX Avalon-ST interface
- Translates Type 0 Configuration Read and Configuration Write Requests to Avalon-MM read and write requests that target the Configuration Space of either Function 0 or Function 1.
- Responds to invalid Type 0 Configuration Requests with an Unsupported Request (UR) status in a Completion Message.
- Converts single dword Memory Read and Memory Write Requests to access 32-bit registers of the target function using the Avalon-MM interface.
- Maps two contiguous MBytes of memory for the two functions with the first MByte for Function 0 and the second MByte for Function 1.
- Sets up two registers for each function.
- Drops the following invalid Write Requests:
- Memory Write Requests with a payload of more than one dword
- Messages with data
- Returns Completer Abort (CA) status in Completion message for invalid Memory Read Requests such as Memory Read Requests with a payload greater than one dword.
- Returns a Completion Status of Successful Completion for valid Configuration Requests to Function 0 and Function 1.

The following figure illustrates, the components of the Configuration Space Bypass Mode Qsys Example Design. As this figure illustrates the example design includes the following components:

- **DUT**: The Arria V GZ Hard IP for PCI Express. The example turns on the **Enable Configuration Space Bypass** parameter.
- **APPS**: The Configuration Space Bypass application demonstrates Configuration Space Bypass mode.
- **pcie_xcvr_reconfig_0**: The Transceiver Reconfiguration Controller performs offset cancellation to compensate for variations due to process, voltage, and temperature (PVT).

- **pcie_reconfig_driver_0**: The PCIe Reconfig Driver drives the Transceiver Reconfiguration Controller. This driver is a plain text Verilog HDL file that you can modify if necessary to meet your system requirements.

Figure 5–1. Configuration Bypass Mode Qsys Example Design


Qsys Configuration Bypass Example Design for Hard IP for PCIe
DUT: Hard IP for PCIe Using Configuration Bypass Mode (Endpoint)
Function 0
Function 1
APPS: Config Bypass Example (cfbp_app_example)
Configuration Bypass Top (cfbp_top)
Reset (rs_hip)
Local Management Interface (LMI)
to PCIe Root Port and Host System
alt_xcvr_reconfig_0
pcie_reconfig_driver_0

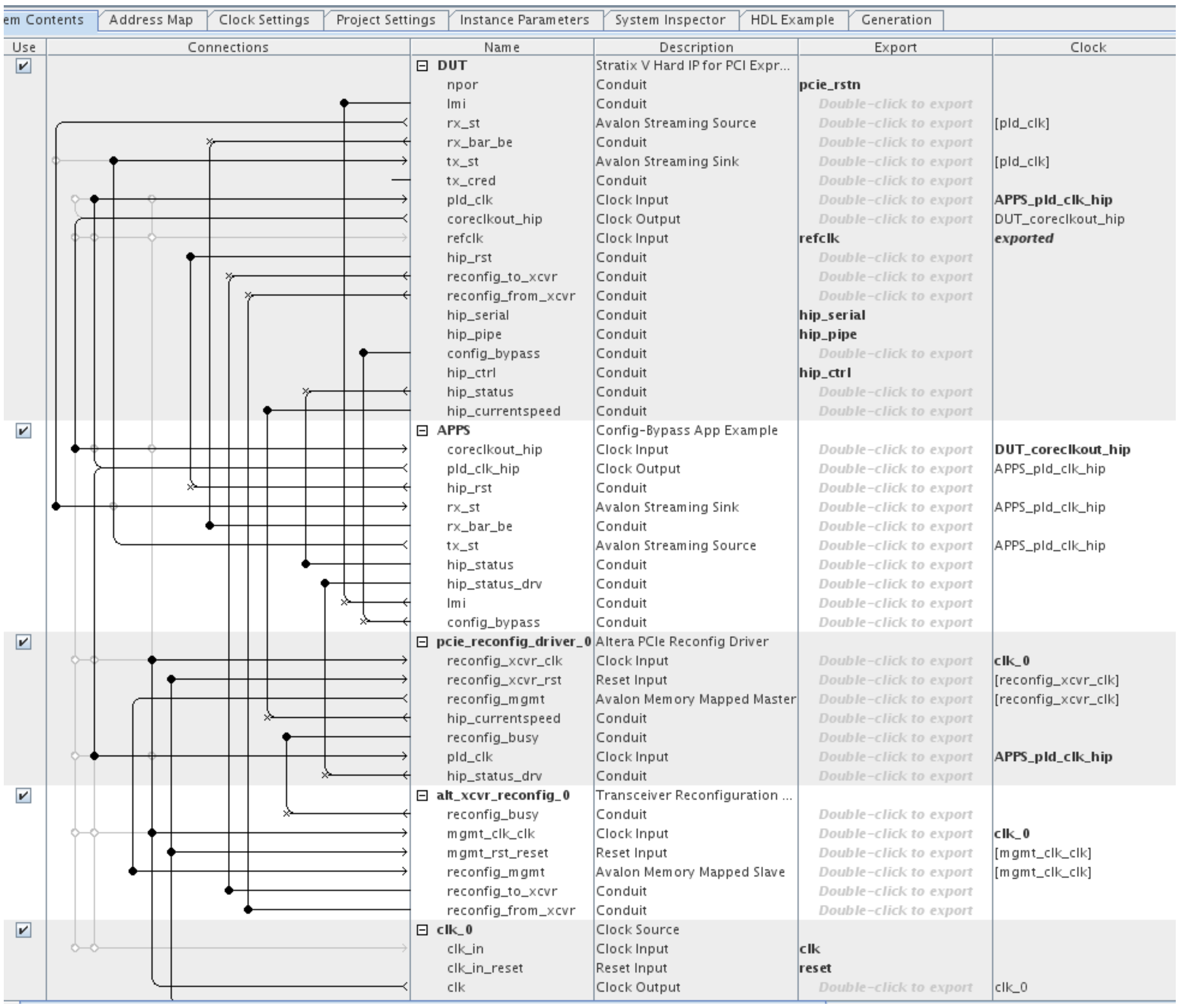
Copying the Configuration Space Bypass Mode Example Design

Follow these steps to copy the Configuration Space Bypass Mode Qsys Example Design to your working directory:

1. Copy the example design, **pcie_cfbp_g2x8_ast256.qsys**, from the installation directory: <*install_dir*>**/ip/altera/altera_pcie/altera_pcie_hip_ast_ed/altera_pcie_cfgbp_ed/qsys_example** to your working directory.
2. Copy the Qsys wrapper file for the Configuration Space Bypass application logic, **altera_pcie_cfgby_ed_hw.tcl**, from the installation directory: <*install_dir*>**/ip/altera/altera_pcie/altera_pcie_hip_ast_ed/altera_pcie_cfgbp_ed/**to your working directory.
3. Rename the **pcie_cfbp_g2x8_ast256.qsys top.qys.** Renaming is necessary because the testbench defines **top.v** as the top-level wrapper. Qsys creates **top.v** from **top.qsys** when you generate the system.
4. Start Qsys by typing `qsys-edit` and open **top.qsys** when prompted by Qsys.

The following figure shows the complete system.

Figure 5–2. Configuration Bypass Qsys System



em Contents | Address Map | Clock Settings | Project Settings | Instance Parameters | System Inspector | HDL Example | Generation

Use	Connections	Name	Description	Export	Clock
✓		⊟ **DUT**	Stratix V Hard IP for PCI Expr...		
		npor	Conduit	**pcie_rstn**	
		lmi	Conduit	Double-click to export	
		rx_st	Avalon Streaming Source	Double-click to export	[pld_clk]
		rx_bar_be	Conduit	Double-click to export	
		tx_st	Avalon Streaming Sink	Double-click to export	[pld_clk]
		tx_cred	Conduit	Double-click to export	
		pld_clk	Clock Input	Double-click to export	**APPS_pld_clk_hip**
		coreclkout_hip	Clock Output	Double-click to export	DUT_coreclkout_hip
		refclk	Clock Input	**refclk**	***exported***
		hip_rst	Conduit	Double-click to export	
		reconfig_to_xcvr	Conduit	Double-click to export	
		reconfig_from_xcvr	Conduit	Double-click to export	
		hip_serial	Conduit	**hip_serial**	
		hip_pipe	Conduit	**hip_pipe**	
		config_bypass	Conduit	Double-click to export	
		hip_ctrl	Conduit	**hip_ctrl**	
		hip_status	Conduit	Double-click to export	
		hip_currentspeed	Conduit	Double-click to export	
✓		⊟ **APPS**	Config-Bypass App Example		
		coreclkout_hip	Clock Input	Double-click to export	**DUT_coreclkout_hip**
		pld_clk_hip	Clock Output	Double-click to export	APPS_pld_clk_hip
		hip_rst	Conduit	Double-click to export	
		rx_st	Avalon Streaming Sink	Double-click to export	APPS_pld_clk_hip
		rx_bar_be	Conduit	Double-click to export	
		tx_st	Avalon Streaming Source	Double-click to export	APPS_pld_clk_hip
		hip_status	Conduit	Double-click to export	
		hip_status_drv	Conduit	Double-click to export	
		lmi	Conduit	Double-click to export	
		config_bypass	Conduit	Double-click to export	
✓		⊟ **pcie_reconfig_driver_0**	Altera PCIe Reconfig Driver		
		reconfig_xcvr_clk	Clock Input	Double-click to export	**clk_0**
		reconfig_xcvr_rst	Reset Input	Double-click to export	[reconfig_xcvr_clk]
		reconfig_mgmt	Avalon Memory Mapped Master	Double-click to export	[reconfig_xcvr_clk]
		hip_currentspeed	Conduit	Double-click to export	
		reconfig_busy	Conduit	Double-click to export	
		pld_clk	Clock Input	Double-click to export	**APPS_pld_clk_hip**
		hip_status_drv	Conduit	Double-click to export	
✓		⊟ **alt_xcvr_reconfig_0**	Transceiver Reconfiguration ...		
		reconfig_busy	Conduit	Double-click to export	
		mgmt_clk_clk	Clock Input	Double-click to export	**clk_0**
		mgmt_rst_reset	Reset Input	Double-click to export	[mgmt_clk_clk]
		reconfig_mgmt	Avalon Memory Mapped Slave	Double-click to export	[mgmt_clk_clk]
		reconfig_to_xcvr	Conduit	Double-click to export	
		reconfig_from_xcvr	Conduit	Double-click to export	
✓		⊟ **clk_0**	Clock Source		
		clk_in	Clock Input	**clk**	
		clk_in_reset	Reset Input	**reset**	
		clk	Clock Output	Double-click to export	clk_0

The following walkthrough does not provide detailed step-by-step instructions to recreate the Qsys system. For step-by-step instructions illustrating how to create designs using Qsys, refer to the Chapter 2, Getting Started with the Arria V GZ Hard IP for PCI Express.

1. Note the following key parameter settings for the Configuration Space Bypass Example Design:
   a. For the DUT, the **Enable Configuration Bypass** parameter is turned on under the **System Settings** banner.
   b. The **Base Address Registers** specify **BAR0** as **1 MByte - 20 bits** of **64-bit prefetchable memory** for each function. In Configuration Space Bypass Mode, the BAR registers inside the Hard IP for PCI Express are not used because the Application Layer implements the Configuration Space for each function.
   c. For testbench compatibility, the **Config-Bypass App Example**, labeled **APPs**, must retain a Device ID of 0xE001 (57345_{10}) and a **Vendor ID** of 0x1172 (4466_{10}).

Generating the Qsys System

On the Qsys **Generation** tab, specify the parameters listed in Table 5–1.

Table 5–1. Parameters to Specify on the Generation Tab in Qsys

Parameter	Value
Simulation	
Create simulation model	**None.** (This option generates a simulation model you can include in your own custom testbench and also allows you to review the HDL code.)
Create testbench Qsys system	**Standard, BFMs for standard Avalon interfaces**
Create testbench simulation model	**Verilog**
Synthesis	
Create HDL design files for synthesis	**Verilog**
Create block symbol file (.bsf)	Enable this option
Output Directory	
Path	**altera_pcie_cfgbp_ed/top**
Simulation	**—**
Testbench	**altera_pcie_cfgbp_ed/top/testbench**
Synthesis	**altera_pcie_cfgbp_ed/top/ synthesis**

1. On the **Generation** tab, click **Generate** to generate the simulation and testbench files.
2. On the **File** menu, click **Save**.

Understanding the Generated Files

The following table describes the files and directories Qsys generates.

Table 5–2. Qsys Generation Output Files

Directory	Description
<testbench_dir>/<variant_name>/ **synthesis**	Includes the top-level HDL file for the Hard IP for PCI Express and the **.qip** file that lists all of the necessary assignments and information required to process the IP core in the Quartus II compiler. Generally, a single **.qip** file is generated for each IP core. These files are used for Quartus II synthesis.
<testbench_dir>/<variant_name>/ **synthesis/submodules**	Includes the HDL files necessary for Quartus II synthesis.
<testbench_dir>/<variant_name>/ **testbench/top_tb/simulation/submodules**	Includes the HDL files necessary for simulation testbench.
<testbench_dir>/<variant_name>/ **testbench/***<cad_vendor>*	Includes the HDL source files and scripts for the simulation testbench.

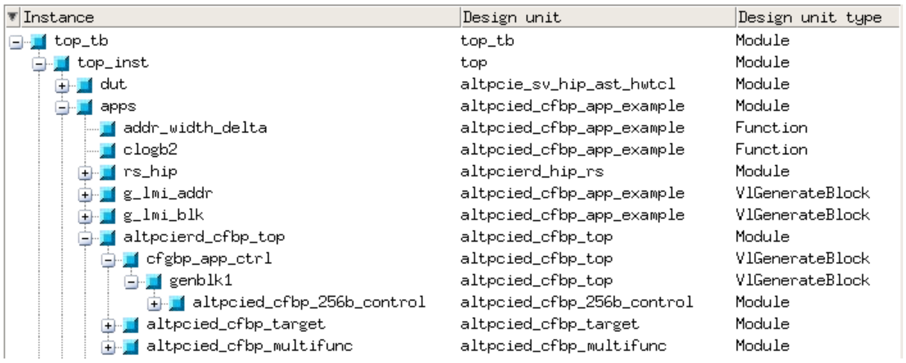
Simulating the Example Design

Follow these steps to simulate the Qsys system using ModelSim:

1. In a terminal window, change to the a**ltera_pcie_cfgbp_ed/top/testbench/mentor** directory.
2. Start the ModelSim simulator by typing `vsim`.
3. To compile the simulation, type the following commands in the terminal window:
   a. `source msim_setup.tcl` ↵ The `msim_setup.tcl` file defines aliases.
   b. `ld_debug` ↵ The `ld_debug` command argument stops optimizations, improving visibility in the ModelSim waveforms.

   The following figure shows the design hierarchy for the Configuration Space Bypass Example Design after compilation.

Figure 5–3. Design Hierarchy for the Configuration Space Bypass Example Design for 256-Bit Avalon-ST Interface



```
Instance                               Design unit                  Design unit type
top_tb                                 top_tb                       Module
  top_inst                             top                          Module
    dut                                altpcie_sv_hip_ast_hwtcl     Module
    apps                               altpcied_cfbp_app_example    Module
      addr_width_delta                 altpcied_cfbp_app_example    Function
      clogb2                           altpcied_cfbp_app_example    Function
      rs_hip                           altpcierd_hip_rs             Module
      g_lmi_addr                       altpcied_cfbp_app_example    VlGenerateBlock
      g_lmi_blk                        altpcied_cfbp_app_example    VlGenerateBlock
      altpcierd_cfbp_top               altpcied_cfbp_top            Module
        cfgbp_app_ctrl                 altpcied_cfbp_top            VlGenerateBlock
          genblk1                      altpcied_cfbp_top            VlGenerateBlock
            altpcied_cfbp_256b_control altpcied_cfbp_256b_control   Module
        altpcied_cfbp_target           altpcied_cfbp_target         Module
        altpcied_cfbp_multifunc        altpcied_cfbp_multifunc      Module
```

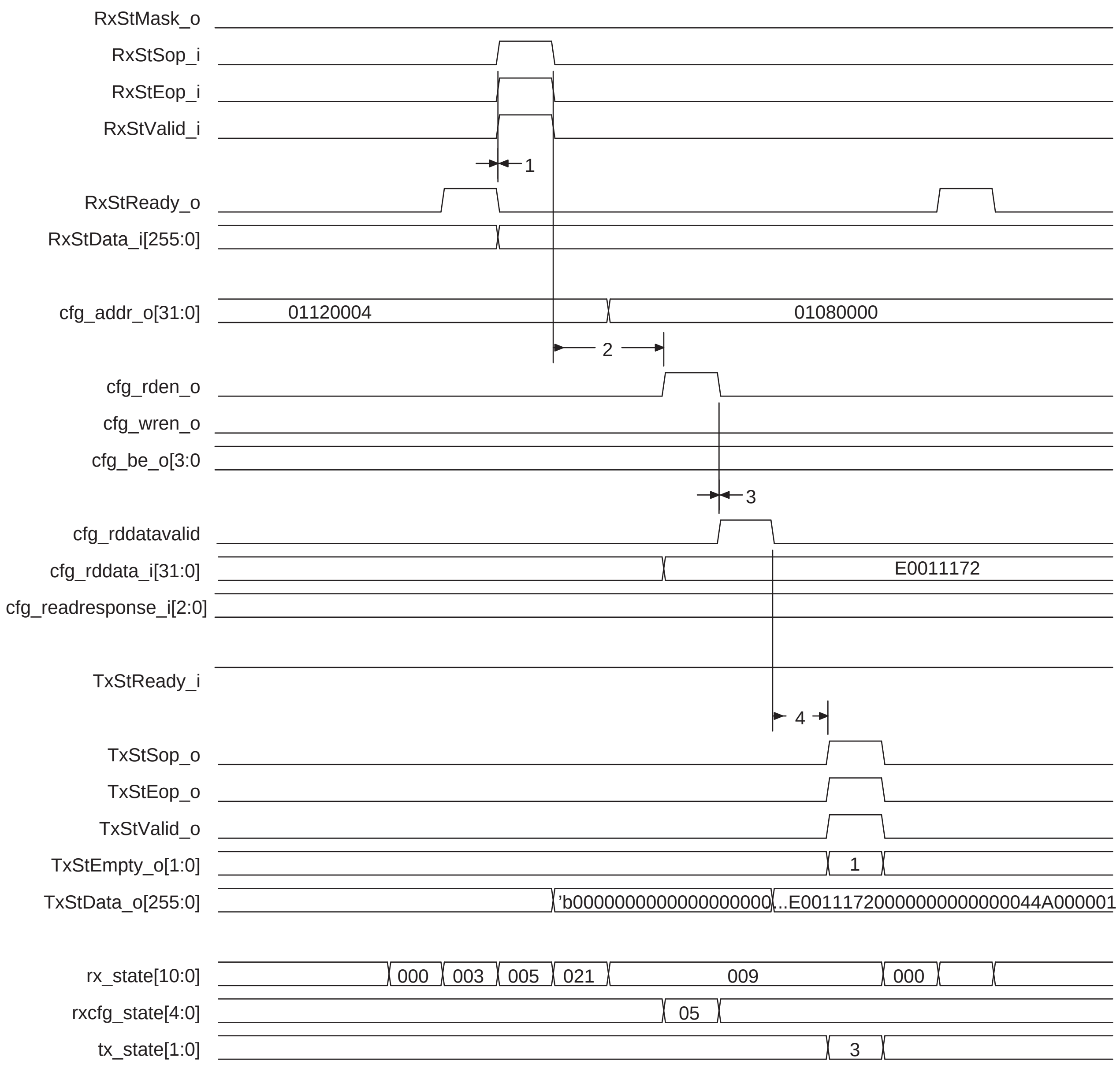
4. To observe the simulation, on the ModelSim View menu, select **wave**. Then add some key interfaces to the wave window. The following four interfaces under the **/top_tb/top_inst/apps/altpcierd_cfbp_top/cfgbp_app_ctrl/genblk1** illustrate the TX and RX interfaces, the current state, and configuration.
   a. *RxSt*
   b. *TxSt*
   c. *Rxm*
   d. *_state*
   e. cfg_*
5. To run the simulation, type the following command: `run -all`↵

☞ By default, the simulation is serial, to simulate using the parallel PIPE interface, you can change the default value of the `serial_sim_hwtcl` parameter from 1 to 0 in **altera_pcie_cfgbp_ed/top/testbench/top_tb/simulation/top_tb.v**. After changing that value, you must recompile the simulation to pick up the new value of the `serial_sim_hwtcl` parameter before running the simulation.

Timing Diagram for Configuration Read to Function 0 for the 256-Bit Avalon-ST Interface

The following timing diagram illustrates a Configuration Read to Function 0 starting at time 60568 ns in the simulation.

Figure 5–4. Configuration Read to Function 0



The preceding timing diagram illustrates the following sequence of events:

1. The Application Layer indicates it is ready to receive requests by asserting `RxSTReady_o`. The RX Avalon-ST interface initiates a Configuration Read, asserting its `RxStSop_i` and `RxStValid_i` signals.
2. At the falling edge of `RxStSop_i`, the Avalon-MM master interface asserts `cfg_rden_o` and specifies the address on `cfg_addr_o[31:0]`.

3. The Function 0 Avalon-MM slave interface asserts `cfg_rddavalid_i` and drives the data on `cfg_rddata_i[31:0]`.
4. On the falling edge of `cfg_rddavalid_i`, the TX interface asserts `TxStSop_o` and `TxStValid_o` and drives the data of `TxStData_o[255:0]`. This is the Completion Request to the host corresponding to its Configuration Read Request.

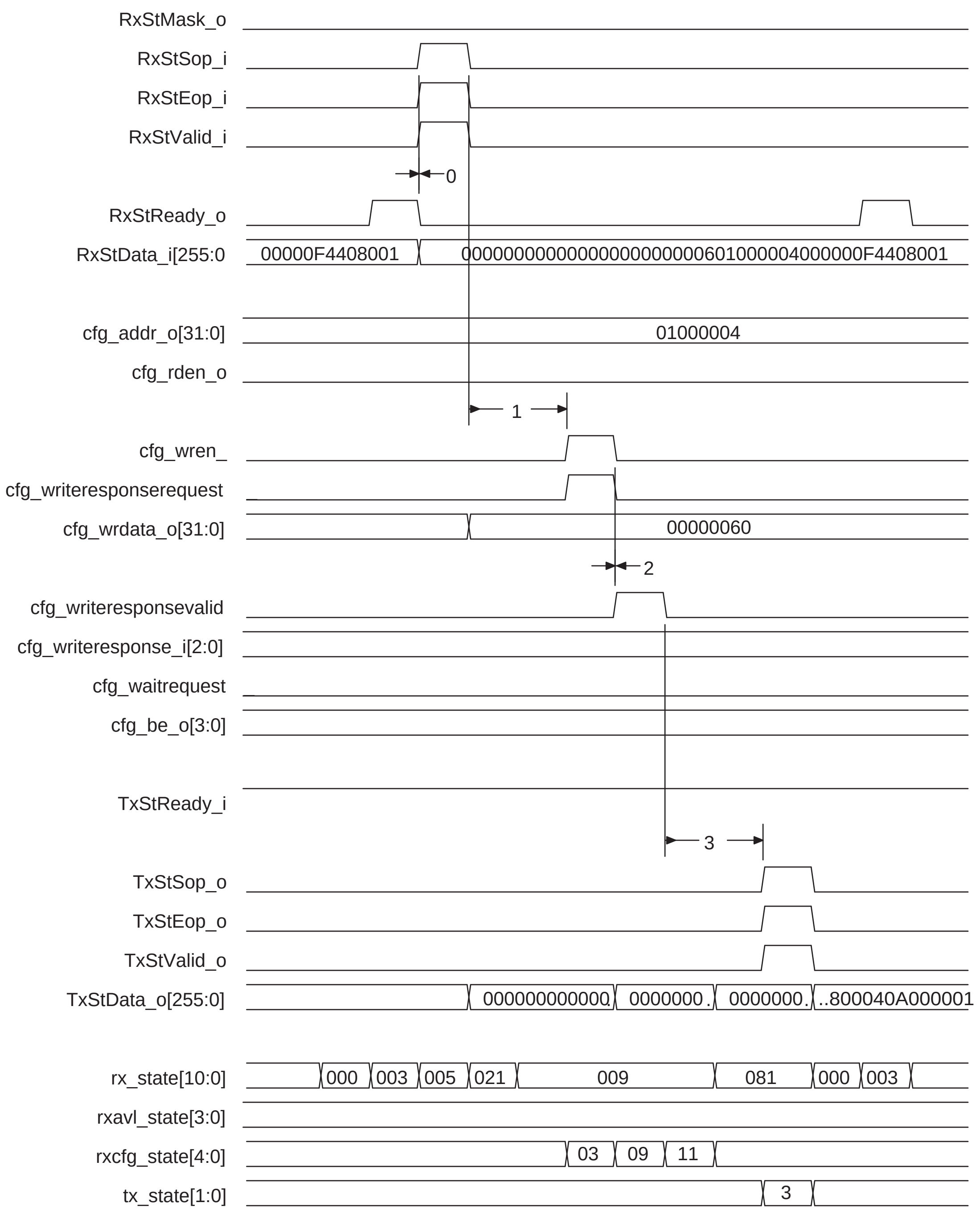
Timing Diagram for Configuration Write to Function 0 for the 256-Bit Avalon-ST Interface

The following timing diagram illustrates a configuration write to Function 0 starting at time 61859 ns in the simulation.

The timing diagram illustrates the following sequence of events:

1. The Application Layer indicates it is ready to receive requests by asserting `RxSTReady_o`. The RX Avalon-ST interface initiates a Configuration Write, asserting its `RxStSop_i` and `RxStValid_i` signals.
2. At the falling edge of `RxStSop_i`, the Avalon-MM master interface asserts `cfg_wren_o` and specifies the data on `cfg_wrdata_o[31:0]`. The Master interface also assert `cfg_writeresponserequest_o`, to request completion status from Function 0.
3. On the falling edge of `cfg_writeresponserequest_o`, Function 0 asserts `cfg_writeresponsevalid_i`.
4. On the falling edge of `cfg_writeresponsevalid_i`, the TX interface asserts `TxStSop_o` and `TxStValid_o` and drives the completion data on `TxStData_o[255:0]`.

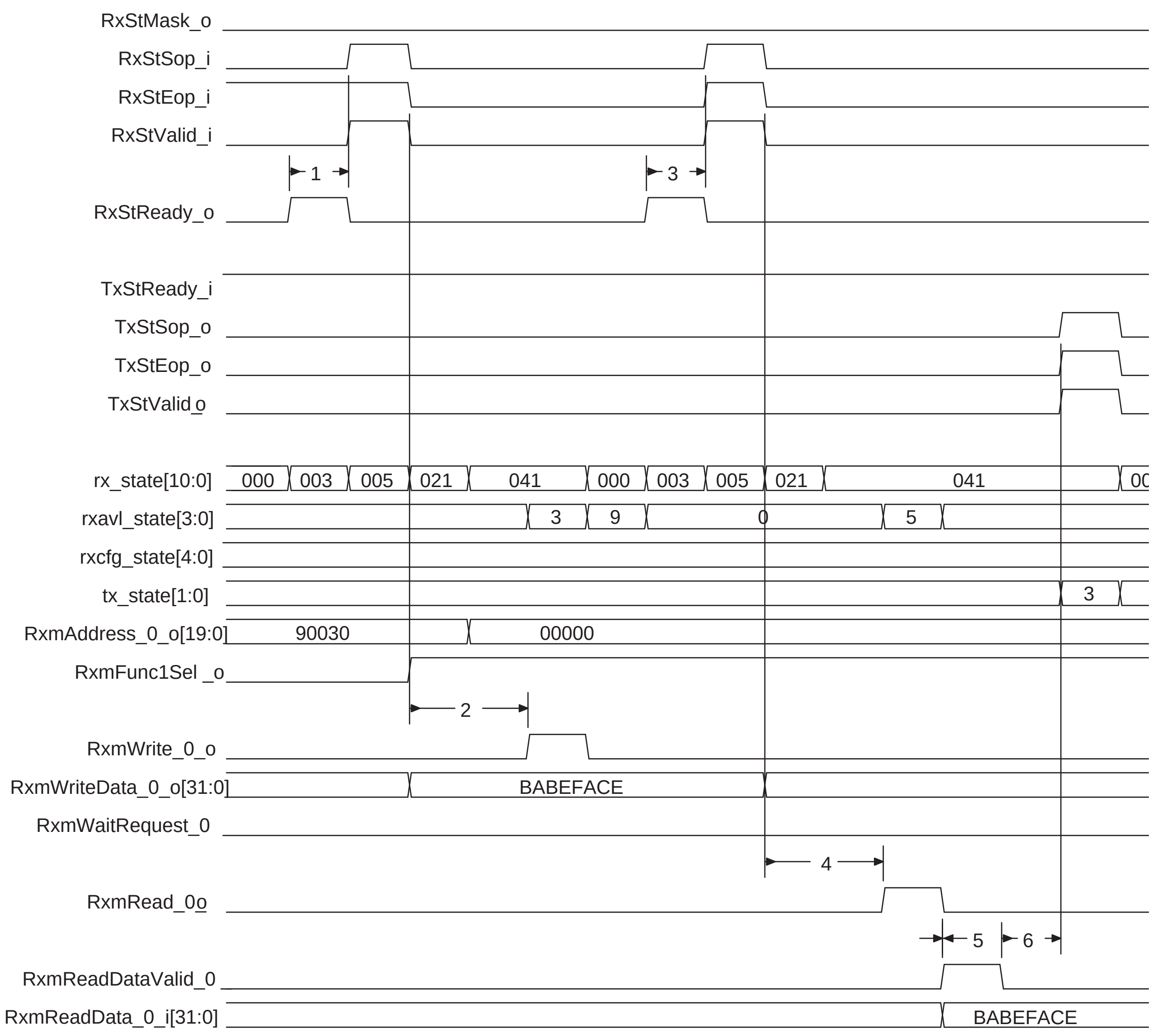
Figure 5–5. Configuration Write to Function 0



Timing Diagram for Memory Write and Read of Function 1256-Bit Avalon-ST Interface

The following timing diagram illustrates memory to Function 1 which occurs in the simulation starting at time 99102 ns.

Figure 5–6. Timing for Memory Write and Read of Function 1



The timing diagram illustrates the following sequence of events:

1. The Application Layer indicates it is ready to receive requests by asserting `RxSTReady_o`. The RX Avalon-ST interface initiates a Memory Write to Function 1, asserting its `RxStSop_i` and `RxStValid_i` signals.
2. At the falling edge of `RxStSop_i`, `RxmFunc1Sel_o` is asserted and the write data is driven on `RxmWriteData_0_o[31:0]`. The Memory Write to Function 1 completes when the data is written.
3. The Application Layer indicates it is ready to receive requests by asserting `RxSTReady_o`. The RX Avalon-ST interface initiates a Memory Read to Function 1, asserting its `RxStSop_i` and `RxStValid_i` signals.

4. After the falling edge of RxStSop_i, the RX Avalon-MM master interface asserts RxmRead_0_o to Function 1.
5. At the falling edge of RxmRead_0_o, Function 1 asserts RxmReadDataValid_0 and drives the data on RxmReadData_0_i[31:0].
6. The host receives the completion data when TxStValid_o, TxStSop_o, and TxStEop_o are asserted.

Partial Transcript for Configuration Space Bypass Simulation

The driver performs the following transactions with status of the transactions displayed in the ModelSim simulation message window:

- Various configuration reads and writes to the Avalon-MM Arria V GZ Hard IP for PCI Express in your system after the link is initialized
- Register writes, reads and compares to both functions
- Burst memory writes, reads, and compares to both functions

Example 5–1 shows the transcript from a successful simulation run.

Example 5–1. Transcript from ModelSim Simulation of Gen1 x4 Endpoint

```
INFO:             464 ns Completed initial configuration of Root Port.
# 495000: INFO: top_tb.top_inst_reset_bfm.reset_deassert: Reset deasserted
# INFO:            3657 ns  RP LTSSM State: DETECT.ACTIVE
# INFO:            4425 ns  RP LTSSM State: POLLING.ACTIVE
# INFO:           17257 ns  RP LTSSM State: DETECT.QUIET
# INFO:           20473 ns  RP LTSSM State: DETECT.ACTIVE
# INFO:           21193 ns  RP LTSSM State: POLLING.ACTIVE
# INFO:           29909 ns  EP LTSSM State: DETECT.ACTIVE
# INFO:           30949 ns  EP LTSSM State: POLLING.ACTIVE
# INFO:           33957 ns  EP LTSSM State: POLLING.CONFIG
# INFO:           34025 ns  RP LTSSM State: DETECT.QUIET
# INFO:           37241 ns  RP LTSSM State: DETECT.ACTIVE
# INFO:           37961 ns  RP LTSSM State: POLLING.ACTIVE
# INFO:           39945 ns  RP LTSSM State: POLLING.CONFIG
# INFO:           41033 ns  RP LTSSM State: CONFIG.LINKWIDTH.START
# INFO:           41445 ns  EP LTSSM State: CONFIG.LINKWIDTH.START
# INFO:           41765 ns  EP LTSSM State: CONFIG.LINKWIDTH.ACCEPT
# INFO:           42057 ns  RP LTSSM State: CONFIG.LINKWIDTH.ACCEPT
# INFO:           42249 ns  RP LTSSM State: CONFIG.LANENUM.WAIT
# INFO:           42789 ns  EP LTSSM State: CONFIG.LANENUM.WAIT
# INFO:           43033 ns  RP LTSSM State: CONFIG.LANENUM.ACCEPT
# INFO:           43109 ns  EP LTSSM State: CONFIG.LANENUM.ACCEPT
# INFO:           43225 ns  RP LTSSM State: CONFIG.COMPLETE
# INFO:           43685 ns  EP LTSSM State: CONFIG.COMPLETE
# INFO:           44953 ns  RP LTSSM State: CONFIG.IDLE
# INFO:           47941 ns EP LTSSM State: CONFIG.IDLE
# INFO:           48089 ns  RP LTSSM State: L0
# INFO:           48133 ns  EP LTSSM State: L0
# INFO:           48226 ns Configuring Bus 000, Device 000, Function 00
# INFO:           48226 ns  RP Read Only Configuration Registers:
# INFO:           48226 ns          Vendor ID: 1556
# INFO:           48226 ns          Device ID: 5555
# INFO:           48226 ns        Revision ID: 00
# INFO:           48226 ns                Class Code: 040000
```

Example 5–1. Transcript from ModelSim Simulation of Gen1 x4 Endpoint (continued)

```
# INFO:      48706 ns     ECRC Check Capable: Supported
# INFO:      48706 ns  ECRC Generation Capable: Supported
# INFO:            48738 ns    RP PCI Express Slot Capability
# INFO:      48738 ns     Power Controller: Not Present
# INFO:      48738 ns         MRL Sensor: Not Present
# INFO:      48738 ns   Attention Indicator: Not Present
# INFO:      48738 ns      Power Indicator: Not Present
# INFO:      48738 ns    Hot-Plug Surprise: Not Supported
# INFO:      48738 ns     Hot-Plug Capable: Not Supported
# INFO:      48738 ns      Slot Power Limit Value: 0
# INFO:      48738 ns      Slot Power Limit Scale: 0
# INFO:      48738 ns          Physical Slot Number: 0
# INFO:      48738 ns Activity_toggle flag is set
# INFO:      48802 ns  RP PCI Express Link Status Register (0081):
# INFO:      48802 ns  RP PCI Express Max Link Speed (0002):
# INFO:      48802 ns  RP PCI Express Current Link Speed (0001):
# INFO:      48802 ns   Negotiated Link Width: x8
# INFO:      48802 ns     Slot Clock Config: Local Clock Used
# INFO:      48834 ns     Current Link Speed: 2.5GT/s
# INFO:      48889 ns  RP LTSSM State: RECOVERY.RCVRLOCK
# INFO:      49669 ns  EP LTSSM State: RECOVERY.RCVRLOCK
# INFO:      50501 ns  EP LTSSM State: RECOVERY.RCVRCFG
# INFO:      51209 ns  RP LTSSM State: RECOVERY.RCVRCFG
# INFO:      53337 ns  RP LTSSM State: RECOVERY.SPEED
# INFO:      53669 ns  EP LTSSM State: RECOVERY.SPEED
# INFO:      54721 ns  RP LTSSM State: RECOVERY.RCVRLOCK
# INFO: 54746 ns Wait for Link to enter L0 after negotiated to the expected speed of EP
#            Target Link Speed (0002):
# INFO:      55235 ns  EP LTSSM State: RECOVERY.RCVRLOCK
# INFO:      56299 ns  EP LTSSM State: RECOVERY.RCVRCFG
# INFO:      57163 ns  RP LTSSM State: RECOVERY.RCVRCFG
# INFO:      57707 ns  RP LTSSM State: RECOVERY.IDLE
# INFO:      57979 ns  EP LTSSM State: RECOVERY.IDLE
# INFO:       58035 ns  RP LTSSM State: L0
# INFO:       58075 ns  EP LTSSM State: L0
# INFO:            58090 ns            New Link Speed: 5.0GT/s
# INFO:      58106 ns  RP PCI Express Link Control Register (0000):
# INFO:            58106 ns       Common Clock Config: Local Clock
# . . .
# INFO: 70602 ns Completed configuration of Endpoint BARs.
# INFO: 70602 ns TASK:my_test Setup
# INFO: 70602 ns TASK:my_test Write to 32bit register at addr = 0x0 with wdata=0xBABEFACE
# INFO: 70610 ns TASK:my_test Read from 32bit register at addr 0x00000000
# INFO: 71298 ns TASK:my_test Register compare matches!
# INFO: 71298 ns TASK:my_test Write to 32bit register at 0x00000004 Actual 0x12345678
# INFO: 71306 ns TASK:my_test => 1.22 Read from 32bit register at addr = 0x00000004
# INFO: 71994 ns TASK:my_test => 1.23  Register compare matches!
# . . .
# INFO 71994 ns TASK:my_test => 2.11 Fill write memory with QWORD_INC pattern
# INFO: 71994 ns TASK:my_test Memory write burst at addr=0x00 with wdata=0x10203040
# INFO: 72002 ns TASK:my_test => 2.21 Memory Read burst
# INFO: 72690 ns TASK:my_test Memory write burst at addr=0x04 with wdata=0x10203040
# INFO: 72698 ns TASK:my_test Memory Read burst
# INFO: 73354 ns TASK:my_test Memory write burst at addr=0x08 with wdata=0x10203040
# INFO: 73362 ns TASK:my_test => 2.21 Memory Read burst
# INFO: 74178 ns TASK:my_test Memory write burst at addr=0x0C with wdata=0x10203040
# . . . .
```

Example 5–1. Transcript from ModelSim Simulation of Gen1 x4 Endpoint (continued)

```
# INFO:   88154 ns Enumerate EP function = 0x01
# INFO:   88154 ns cfgbp_enum_config_space => Setup config space for func = 00000001
# INFO:  88154 ns  Config Read
# INFO:   88946 ns CfgRD at addr =0x00000000 returns data = 0xE0011172
# INFO:   88946 ns Set Bus_Master and Memory_Space_Enable bit in Command register00000001
# INFO:   88946 ns Read Modified WRite to config register =0x00000004 in func = 0x00000001
# INFO:   88946 ns Read config reg
# INFO:   89738 ns Original config read data = 00000000
# INFO:   89738 ns Config write with data = 00000006
# INFO:   91338 ns After cfg_rd_modified_wr, config_data = 0x00000006
# INFO:   92938 ns CfgRD at BAR0 (addr =0x00000010) returns data = 0xFFF0000C
# INFO:   94530 ns CfgRD at addr =0x00000010 returns data = 0x8000000C
# INFO:   97658 ns BAR Address Assignments:
# INFO:   97658 ns BAR    Size       Assigned Address  Type
# INFO:   97658 ns BAR1:0  1 MBytes 00000001 00000000 Prefetchable
# INFO:   97658 ns BAR2  Disabled
# INFO:   97658 ns BAR3  Disabled
# INFO:   97658 ns BAR4  Disabled
# INFO:   97658 ns BAR5  Disabled
# INFO:   97658 ns ExpROM Disabled

# INFO:   98794 ns Completed configuration of Endpoint BARs.
# INFO:   98794 ns TASK:my_test => Setup
# INFO:   98794 ns TASK:my_test Write to 32bit register at 0x000000 with wdata=0xBABEFACE
# INFO:   98802 ns TASK:my_test => 1.12 Read from 32bit register at addr = 0x00000000
# INFO:   9490 ns TASK:my_test => 1.13 Register compare matches!
# . . .
INFO:     115370 ns TASK:my_test => 2.21 Memory Read burst
# SUCCESS: Simulation stopped due to successful completion!
# Break in Function ebfm_log_stop_sim at
./..//top_tb/simulation/submodules//altpcietb_bfm_log.v line 78
```

6. Parameter Settings

This chapter describes the parameters which you can set using the MegaWizard Plug-In Manager or Qsys design flow to instantiate the following IP Cores:

- Arria V GZ Hard IP for PCI Express
- Avalon-MM Arria V GZ Hard IP for PCI Express

In the following tables, hexadecimal addresses in green are links to additional information in the *Chapter 9, Register Descriptions*.

System Settings

The first group of settings defines the overall system. Table 6–1 describes these settings.

Table 6–1. System Settings for PCI Express (Part 1 of 4)

Parameter	Value	Description
Number of Lanes	×1, ×2, ×4, ×8	Specifies the maximum number of lanes supported.
Lane Rate	Gen1 (2.5 Gbps) (1) Gen2 (2.5/5.0 Gbps) Gen3 (2.5/5.0/8.0 Gbps)	Specifies the maximum data rate at which the link can operate. Gen3 is only supported for Arria V GZ production devices.
Port type	Native Endpoint Root Port Legacy Endpoint	Specifies the port type. Altera recommends **Native Endpoint** for all new Endpoint designs. Select **Legacy Endpoint** only when you require I/O transaction support for compatibility. The **Legacy Endpoint** is not available for the Avalon-MM Arria V GZ Hard IP for PCI Express. The Endpoint stores parameters in the Type 0 Configuration Space which is outlined in Table 9–2 on page 9–2. The Root Port stores parameters in the Type 1 Configuration Space which is outlined in Table 9–3 on page 9–2.
PCI Express Base Specification version	2.1 3.0	Select either the 2.1 or 3.0 specification.
Application interface	64-bit Avalon-ST 128-bit Avalon-ST 256-bit Avalon-ST	Specifies the interface between the PCI Express Transaction Layer and the Application Layer. Refer to Table 10–3 on page 10–6 for a comprehensive list of available link width, interface width, and frequency combinations. Refer to "Avalon Memory-Mapped System Settings" on page 6–12 to set the width of the Application Layer interface for the Avalon-MM Arria V GZ Hard IP for PCI Express. This parameter does not apply to the Avalon-MM IP Cores.

Table 6–1. System Settings for PCI Express (Part 2 of 4)

Parameter	Value	Description
RX Buffer credit allocation - performance for received requests	**Minimum** **Low** **Medium**	Determines the allocation of posted header credits, posted data credits, non-posted header credits, completion header credits, and completion data credits in the 16 KByte RX buffer. The 5 settings allow you to adjust the credit allocation to optimize your system. The credit allocation for the selected setting displays in the message pane. Refer to Chapter 14, Flow Control, for more information about optimizing performance. The Flow Control chapter explains how the **RX credit allocation** and the **Maximum payload size** that you choose affect the allocation of flow control credits. You can set the **Maximum payload size** parameter on the **Device** tab. The **Message** window of the GUI dynamically updates the number of credits for Posted, Non-Posted Headers and Data, and Completion Headers and Data as you change this selection. ■ **Minimum**–This setting 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**–This setting 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**–This setting allocates 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 **Maximum**	■ **High**–This setting 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**–This setting 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.

Table 6–1. System Settings for PCI Express (Part 3 of 4)

Parameter	Value	Description
Reference clock frequency	**100 MHz** **125 MHz**	The *PCI Express Base Specification 3.0* requires a 100 MHz ±300 ppm reference clock. The 125 MHz reference clock is provided as a convenience for systems that include a 125 MHz clock source. For more information about Gen3 operation, refer to "4.3.8 Refclk Specifications for 8.0 GT/s" in the specification. For Gen3, Altera recommends using a common reference clock (0 ppm) because when using separate reference clocks (non 0 ppm), the PCS occasionally must insert SKP symbols, potentially causes the PCIe link to go to recovery. Arria V GZ PCIe Hard IP in Gen1 or Gen2 modes are not affected by this issue. Systems using the common reference clock (0 ppm) are not affected by this issue. The primary repercussion of this is a slight decrease in bandwidth. On Gen3 x8 systems, this bandwidth impact is negligible. If non 0 ppm mode is required, so that separate reference clocks are being used, please contact Altera for further information and guidance.
Use 62.5 MHz application clock	**On/Off**	This mode is only available only for Gen1 ×1.
Use deprecated RX Avalon-ST data byte enable port (rx_st_be)	**On/Off**	This parameter is only available for the Avalon-ST Arria V GZ Hard IP for PCI Express.
Enable byte parity ports on Avalon-ST interface	**On/Off**	When **On**, the RX and TX datapaths are parity protected. Parity is odd. This parameter is not available for the Avalon-MM IP Cores.
Multiple packets per cycle	**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 feature is only supported for Gen3 ×8. This parameter is not available for the Avalon-MM IP Cores. For more information refer to "Multiple Packets per Cycle" on page 8–27.
Enable configuration via the PCIe link	**On/Off**	When **On**, the Quartus II software places the Endpoint in the location required for configuration via protocol (CvP). For more information about CvP, refer to "Configuration via Protocol (CvP)" on page 12–1. CvP is not supported for Gen3 variants.
Use credit consumed selection port tx_cons_cred_sel	**On/Off**	
Enable Configuration bypass	**On/Off**	When **On**, the Arria V GZ 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.

Table 6–1. System Settings for PCI Express (Part 4 of 4)

Parameter	Value	Description
Enable Hard IP Reconfiguration	**On/Off**	When enabled, 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" on page 17–1. This parameter is not available for the Avalon-MM IP Cores.

Note to Table 6–1:

(1) The Gen1 and Gen2 simulation models support pipe and serial simulation. The Gen3 simulation model supports serial simulation only with Phase 2 and Phase 3 Equalization bypassed.

Base Address Register (BAR) and Expansion ROM Settings

Table 6–2 lists the PCI BAR and Expansion ROM register settings. As this table indicates, the type and size of BARs available depend on port type. For more information about how the Avalon-MM Bridge uses the BARs, refer to "PCI Express-to-Avalon-MM Address Translation for Endpoints for 32-Bit Bridge" on page 7–20. The

Table 6–2. 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. The **I/O address space** BAR is only available for the **Legacy Endpoint**.
Size	**16 Bytes–8 EBytes**	Supports the following memory sizes: ■ 128 bytes–2 GBytes or 8 EBytes: **Endpoint** and **Root Port** variants ■ 6 bytes–4 KBytes: **Legacy Endpoint** variants
Expansion ROM	**Disabled–16 MBytes**	Specifies the size of the optional ROM. The expansion ROM is not available for the Avalon-MM Arria V GZ Hard IP for PCI Express.

Base and Limit Registers for Root Ports

Table 6–3 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.

The Avalon-MM Hard IP for PCI Express Root Port does not filter addresses; consequently, it does not provide the `Base` and `Limit` register parameters.

For more information, refer to the *PCI-to-PCI Bridge Architecture Specification*.

Table 6–3. Base and Limit Registers

Parameter	Value	Description
Input/Output	**Disable** **16-bit I/O addressing** **32-bit I/O addressing**	Specifies the address widths for the `IO base` and `IO limit` registers.
Prefetchable memory	**Disable** **32-bit I/O addressing** **64-bit I/O addressing**	Specifies the address widths for the `Prefetchable Memory Base` register and `Prefetchable Memory Limit` register.

Device Identification Registers

Table 6–4 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. For more information, refer to "Hard IP Reconfiguration Interface" on page 8–49.

Table 6–4. Device ID Registers

Register Name/ Offset Address	Range	Default Value	Description
Vendor ID	16 bits	0x0000	Sets the read-only value of the `Vendor ID` register. This parameter can not be set to 0xFFFF per the PCI Express Specification. Address: 0x000.
Device ID	16 bits	0x0001	Sets the read-only value of the `Device ID` register. Address: 0x000.
Revision ID	8 bits	0x01	Sets the read-only value of the `Revision ID` register. Address: 0x008.
Class code	24 bits	0x000000	Sets the read-only value of the `Class Code` register. Address: 0x008.
Subsystem vendor ID	16 bits	0x0000	Sets the read-only value of the `Subsystem Vendor ID` register in the PCI Type 0 Configuration Space. This parameter cannot be set to 0xFFFF per the *PCI Express Base Specification 2.1* or *3.0*. Address: 0x02C.
Subsystem Device ID	16 bits	0x0000	Sets the read-only value of the `Subsystem Device ID` register in the PCI Type 0 Configuration Space. Address: 0x02C

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. The byte offset within the PCI Configuration Space indicates the parameter address. Table 6–5 lists these parameters.

Table 6–5. Capabilities Registers (Part 1 of 2)

Parameter	Possible Values	Default Value	Description
Device Capabilities			
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**.**
Tags supported	**32 64**	32 - Avalon-ST	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 9–8 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 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]`. The Avalon-MM Arria V GZ Hard IP for PCI Express always supports 8 tags. You do not need to configure this parameter.

Table 6–5. Capabilities Registers (Part 2 of 2)

Parameter	Possible Values	Default Value	Description
Completion timeout range	**ABCD** **BCD** **ABC** **AB** **B** **A** **None**	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 Express Capability Structure Version 2.0 described in Table 9–8 on page 9–5. 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 are used to 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. Altera recommends that the completion timeout mechanism expire in no less than 10 ms.
Implement completion timeout disable 0x0A8	**On/Off**	On	For Endpoints using PCI Express version 2.0, this option must be **On**. The timeout range is selectable. 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.

Error Reporting

Table 6–6 lists the error reporting and ECRC parameter registers.

Table 6–6. Error Reporting 0x800–0x834

Parameter	Value	Default Value	Description
Advanced error reporting (AER)	**On/Off**	Off	When **On**, enables the Advanced Error Reporting (AER) capability.
ECRC check	**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.
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.
ECRC forwarding	**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. This parameter is not available for the Avalon-MM IP Cores.
Track Receive Completion Buffer Overflow	**On/Off**	Off	When **On**, the core includes the `rxfx_cplbuf_ovf` output status signal to track the RX posted completion buffer overflow status. This signal is not available for the Avalon-MM IP Cores.

Note to Table 6–6:

(1) Throughout *the Arria V GZ Hard IP for PCI Express User Guide*, the terms word, dword and qword have the same meaning that they have in the *PCI Express Base Specification Revision 2.1 or 3.0*. A word is 16 bits, a dword is 32 bits, and a qword is 64 bits.

Link Capabilities

Table 6–7 lists the Link Capabilities parameter registers.

Table 6–7. Link Capabilities 0x090 (Part 1 of 2)

Parameter	Value	Description
Link port number	**0x01**	Sets the read-only value of the port number field in the `Link Capabilities Register`.
Data link layer active reporting	**On/Off**	Turn **On** this parameter for a downstream port, if the component supports the optional capability of reporting the DL_Active state of the Data Link Control and Management State Machine. For a hot-plug capable downstream port (as indicated by the `Hot-Plug Capable` field of the `Slot Capabilities` register), this parameter must be turned **On**. For upstream ports and components that do not support this optional capability, turn **Off** this option. This parameter is only supported for the Arria V GZ Hard IP for PCI Express in Root Port mode.

Table 6–7. Link Capabilities 0x090 (Part 2 of 2)

Parameter	Value	Description
Surprise down reporting	**On/Off**	When this option is **On**, a downstream port supports the optional capability of detecting and reporting the surprise down error condition. This parameter is only supported for the Arria V GZ Hard IP for PCI Express in Root Port mode.
Slot clock configuration	**On/Off**	When **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.

MSI and MSI-X Capabilities

Table 6–8 lists the MSI and MSI-X Capabilities parameter registers.

Table 6–8. MSI and MSI-X Capabilities 0x050–0x05C, 0x068–0x06C

Parameter	Value	Description
MSI messages requested	**1, 2, 4, 8, 16**	Specifies the number of messages the Application Layer can request. Sets the value of the `Multiple Message Capable` field of the `Message Control` register, 0x050[31:16].
MSI-X Capabilities (0x068–0x06C)		
Implement MSI-X	**On/Off**	When **On**, enables the MSI-X functionality.
	Bit Range	
MSI-X Table size	[10:0]	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^{11}). Address: 0x068[26:16]
MSI-X 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 32-bit qword-aligned offset [(1)]. This field is read-only.
MSI-X 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.
MSI-X 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.
MSI-X Pending Bit Array – BAR Indicator	[2:0]	Indicates which of a function's Base Address registers, located beginning at 0x10 in Configuration Space, is used to map the function's MSI-X PBA into memory space. This field is read-only. Legal range is 0–5.

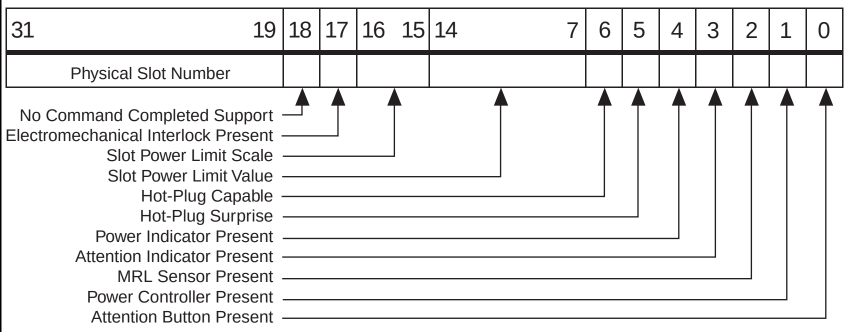
Note to Table 6–5:

(1) Throughout *The Arria V GZ Hard IP for PCI Express User Guide*, the terms word, dword and qword have the same meaning that they have in the *PCI Express Base Specification Revision 1.0a, 1.1, 2.0 or 2.1*. A word is 16 bits, a dword is 32 bits, and a qword is 64 bits.

Slot Capabilities

Table 6–9 lists the Slot Capabilities parameter registers.

Table 6–9. Slot Capabilities 0x094

Parameter	Value	Description
Slot register	**On/Off**	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`. This parameter is only supported for the Arria V GZ Hard IP for PCI Express in Root Port mode.
Slot capability register	—	Defines the characteristics of the slot. You turn on this option by selecting **Enable slot capability**. The various bits are defined as follows:  31 19 18 17 16 15 14 7 6 5 4 3 2 1 0 Physical Slot Number No Command Completed Support Electromechanical Interlock Present Slot Power Limit Scale Slot Power Limit Value Hot-Plug Capable Hot-Plug Surprise Power Indicator Present Attention Indicator Present MRL Sensor Present Power Controller Present Attention Button Present
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 2.1* 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 Revision 2.1* for more information.
Slot number	`0-8191`	Specifies the slot number.

Power Management

Table 6–10 describes the Power Management parameter registers.

Table 6–10. Power Management Parameters

Parameter	Value	Description
Endpoint L0s acceptable latency	**Maximum of 64 ns** **Maximum of 128 n** **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**)**. The Arria V GZ Hard IP for PCI Express and Avalon-MM Arria V GZ Hard IP for PCI Express do 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`. The Arria V GZ Hard IP for PCI Express and Avalon-MM Arria V GZ Hard IP for PCI Express do 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 1 µs. This is the safest setting for most designs.

PHY Characteristics

Table 6–11 lists the PHY characteristics. These parameters is only available for the Arria V GZ Hard IP for PCI Express IP Core.

Table 6–11. PHY Characteristics (Part 1 of 2)

Parameter	Value	Description
Gen2 transmit deemphasis	**3.5dB** **6dB**	Specifies the transmit deemphasis for Gen2.Altera recommends the following settings: ■ 3.5dB: Short PCB traces ■ 6.0dB: Long PCB traces.

Table 6–11. PHY Characteristics (Part 2 of 2)

Parameter	Value	Description
Use ATX PLL	**On/Off**	When enabled, the Hard IP for PCI Express uses the ATX PLL instead of the CMU PLL. For other configurations, using the ATX PLL instead of the CMU PLL reduces the number of transceiver channels that are necessary. This option requires the use of the soft reset controller and does not support the CvP flow. For more information about channel placement, refer to "Serial Interface Signals" on page 8–60.
Enable Common Clock Configuration	**On/Off**	When you turn this option on, the Application Layer and Transaction Layer use a common clock. Using a common clock reduces datapath latency because synchronizers are not necessary.

Avalon Memory-Mapped System Settings

Table 6–12 lists the Avalon-MM system parameter registers. These parameters are available for the Avalon-MM Hard IP for PCI Express IP Core.

Table 6–12. Avalon Memory-Mapped System Settings (Part 1 of 2)

Parameter	Value	Description
Avalon-MM width	**64-bit** **128-bit**	Specifies the interface width between the PCI Express Transaction Layer and the Application Layer.
Avalon-MM address width	**32-bit** **64-bit**	Specifies the address width for Avalon-MM RX master ports that access Avalon-MM slaves in the Avalon address domain. When you select 32-bit addresses, the PCI Express Avalon-MM Bridge performs address translation. When you specify 64-bits addresses, no address translation is performed in either direction. The destination address specified is forwarded to the Avalon-MM interface without any changes. You can limit the number of address bits used by Avalon-MM slave (TXS) components to the actual size required by specifying the address size in the Avalon-MM slave component GUI.
Peripheral Mode	**Requester/Completer, Completer-Only**	Specifies whether the Avalon-MM Arria V GZ Hard IP for PCI Express is capable of sending requests to the upstream PCI Express devices, and whether the incoming requests are pipelined. **Requester/Completer**—In this mode, the Hard IP can send request packets on the PCI Express TX link and receive request packets on the PCI Express RX link. **Completer-Only**—In this mode, the Hard IP can receive requests, but cannot initiate upstream requests. However, it can transmit completion packets on the PCI Express TX link. This mode removes the Avalon-MM TX slave port and thereby reduces logic utilization.
Single dword completer	**On/Off**	This is a non-pipelined version of **Completer-Only** mode. At any time, only a single request can be outstanding. **Single dword completer** uses fewer resources than **Completer-Only. This** variant is targeted for systems that require simple read and write register accesses from a host CPU. If you select this option, the width of the data for RXM BAR masters is always 32 bits, regardless of the **Avalon-MM width**.

Table 6–12. Avalon Memory-Mapped System Settings (Part 2 of 2)

Parameter	Value	Description
Control Register Access (CRA) Avalon slave port	**On/Off**	Allows read and write access to bridge registers from the interconnect fabric using a specialized slave port. This option is required for **Requester/Completer** variants and optional for **Completer-Only** variants. Enabling this option allows read and write access to bridge registers, except in the Completer-Only single dword variations.
Enable multiple MSI/MSI-X support	**On/Off**	When you turn this option **On**, the core includes top-level MSI and MSI-X interfaces that you can use to implement a Customer Interrupt Handler for MSI and MSI-X interrupts. For more information about the Custom Interrupt Handler, refer to Interrupts for End Points Using the Avalon-MM Interface with Multiple MSI/MSI-X Support.
Auto Enable PCIe Interrupt (enabled at power-on)	**On/Off**	Turning on this option enables the Avalon-MM Arria V GZ Hard IP for PCI Express interrupt register at power-up. Turning off this option disables the interrupt register at power-up. The setting does not affect run-time configuration of the interrupt enable register.

Avalon to PCIe Address Translation Settings

Table 6–13 lists the Avalon-MM PCI Express address translation parameter registers.

Starting in version 13.0 of the Quartus II software, the Avalon-MM Hard IP for PCI Express supports 64-bit addressing. If you select 64-bit addressing, no address translation is necessary.

Table 6–13. Avalon Memory-Mapped System Settings

Parameter	Value	Description
Number of address pages	**1,2,4,8,16,32,64, 128,256,512**	Specifies the number of pages required to translate Avalon-MM addresses to PCI Express addresses before a request packet is sent to the Transaction Layer. Each of the 512 possible entries corresponds to a base address of the PCI Express memory segment of a specific size.
Size of address pages	**4 KByte –4 GBytes**	Specifies the size of each memory segment. Each memory segment must be the same size. Refer to Avalon-MM-to-PCI Express Address Translation Algorithm for 32-Bit Addressing for more information about address translation.

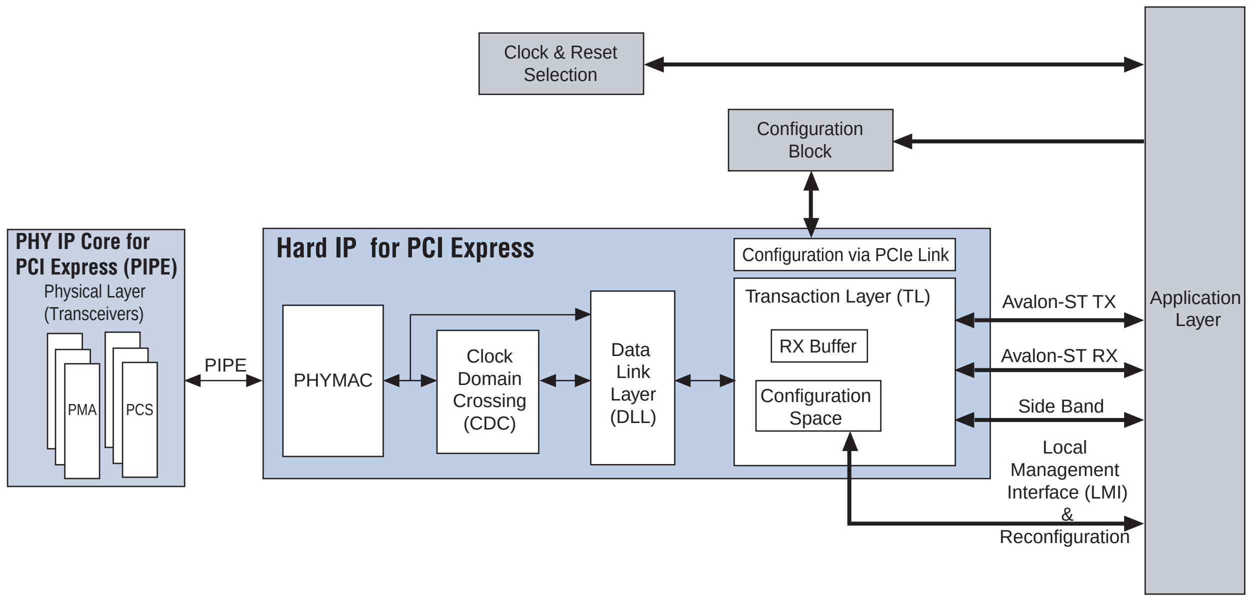
7. IP Core Architecture

The Arria V GZ Hard IP for PCI Express implements the complete PCI Express protocol stack as defined in the *PCI Express Base Specification 3.0.* 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.

Figure 7–1 provides a high-level block diagram of the Arria V GZ Hard IP for PCI Express.

Figure 7–1. Arria V GZ Hard IP for PCI Express with Avalon-ST Interface



As the following illustrates, an Avalon-ST interface provides access to the Application Layer which can be either 64, 128, or 256 bits. Table 7–1 provides the Application Layer clock frequencies.

Table 7–1. Application Layer Clock Frequencies

Lanes	Gen1	Gen2	Gen3
×1	125 MHz @ 64 bits or 62.5 MHz @ 64 bits	125 MHz @ 64 bits 62.5 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:

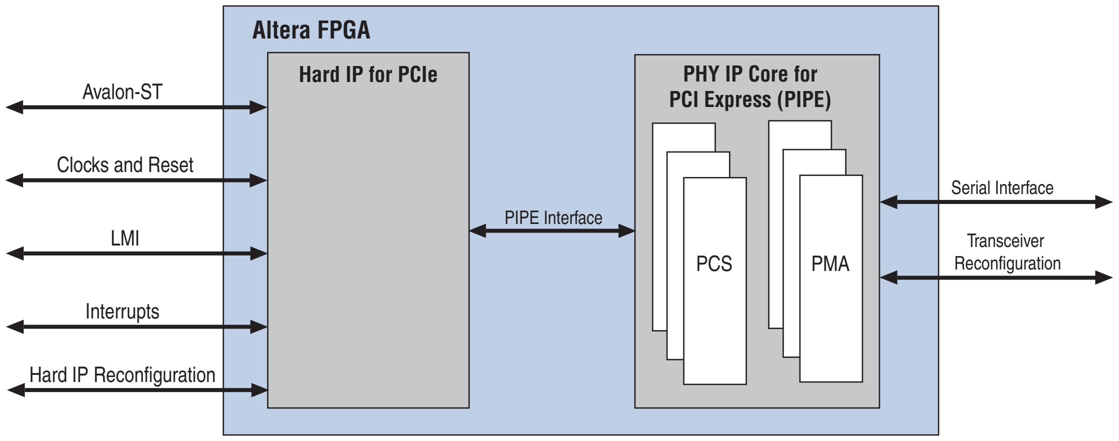
- The LMI interface
- The Avalon-MM PCIe reconfiguration interface, which can access any read-only Configuration Space Register
- In Root Port mode, you can also access the Configuration Space Registers with a Configuration Type TLP using the Avalon-ST interface. A Type 0 Configuration TLP is used to 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.

Key Interfaces

If you select the Arria V GZ Hard IP for PCI Express, your design includes an Avalon-ST interface to the Application Layer. If you select the Avalon-MM Arria V GZ Hard IP for PCI Express, your design includes an Avalon-MM interface to the Application Layer. The following sections introduce the interfaces shown in Figure 7–2.

Figure 7–2. Key Interfaces in the Arria V GZ Hard IP for PCI Express



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. For more information about the RX datapath, refer to "Avalon-ST RX Interface" on page 8–4.

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. for more information about the signals in this interface., refer to "Avalon-ST TX Interface" on page 8–17Avalon-MM Interface

In Qsys, the Arria V GZ Hard IP for PCI Express is available with either an Avalon-ST interface or an Avalon-MM interface to the Application Layer. When you select the Avalon-MM Arria V GZ Hard IP for PCI Express, an Avalon-MM bridge module connects the PCI Express link to the system interconnect fabric. If you are not familiar with the PCI Express protocol, variants using the Avalon-MM interface may be easier to understand. A PCI Express to Avalon-MM bridge translates the PCI Express read, write and completion TLPs into standard Avalon-MM read and write commands typically used by master and slave interfaces. The PCI Express to Avalon-MM bridge also translates Avalon-MM read, write and read data commands to PCI Express read, write and completion TLPs.

Clocks and Reset

The *PCI Express Base Specification* requires an input reference clock, which is called `refclk` in this design. Although the *PCI Express Base Specification* stipulates that the frequency of this clock be 100 MHz, the Hard IP also accepts a 125 MHz reference clock as a convenience. You can specify the frequency of your input reference clock using the parameter editor under the **System Settings** heading.

The *PCI Express Base Specification* also requires a system configuration time of 100 ms. To meet this specification, the Arria V GZ Hard IP for PCI Express includes an embedded hard reset controller. This reset controller exits the reset state after the I/O ring of the device is initialized. For more information about clocks and reset, refer to "Reset and Clocks" on page 10–1, "Clock Signals" on page 8–28, and "Reset Signals, Status, and Link Training Signals" on page 8–28.

Local Management Interface (LMI Interface)

The LMI bus provides access to the PCI Express Configuration Space in the Transaction Layer. For information about the LMI interface, refer to "LMI Signals" on page 8–47.

Hard IP Reconfiguration

The PCI Express reconfiguration bus allows you to dynamically change the `read-only` values stored in the Configuration Registers. For detailed information, refer to "Hard IP Reconfiguration Interface" on page 8–49.

Transceiver Reconfiguration

The transceiver reconfiguration interface allows you to dynamically reconfigure the values of analog settings in the PMA block of the transceiver. Dynamic reconfiguration is necessary to compensate for process variations. The Altera Transceiver Reconfiguration Controller IP core provides access to these analog settings. For more information about the transceiver reconfiguration interface, refer to "Transceiver PHY IP Reconfiguration" on page 17–8.

Interrupts

The Arria V GZ Hard IP for PCI Express offers three interrupt mechanisms:

- Message Signaled Interrupts (MSI)— MSI uses the Transaction Layer's request-acknowledge handshaking protocol to implement interrupts. The MSI Capability structure is stored in the Configuration Space and is programmable using Configuration Space accesses.
- MSI-X—The Transaction Layer generates MSI-X messages which are single dword memory writes. In contrast to the MSI capability structure, which contains all of the control and status information for the interrupt vectors, the MSI-X Capability structure points to an MSI-X table structure and MSI-X PBA structure which are stored in memory.
- Legacy interrupts—The `app_int_sts` input port controls legacy interrupt generation. When `app_int_sts` is asserted, the Hard IP generates an Assert_INT<*n*> message TLP.

For more detailed information about interrupts, refer to "Interrupt Signals for Endpoints" on page 8–32.

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 and Gen2 simulation models support pipe and serial simulation. The Gen3 simulation model supports serial simulation only with equalization bypassed.

Transaction Layer

The Transaction Layer is located between the Application Layer and the Data Link Layer. It generates and receives Transaction Layer Packets. Figure 7–3 illustrates the Transaction Layer. The Transaction Layer includes three sub-blocks: the TX datapath, the Configuration Space, and the RX datapath, which are shown in Figure 7–3 on page 7–6.

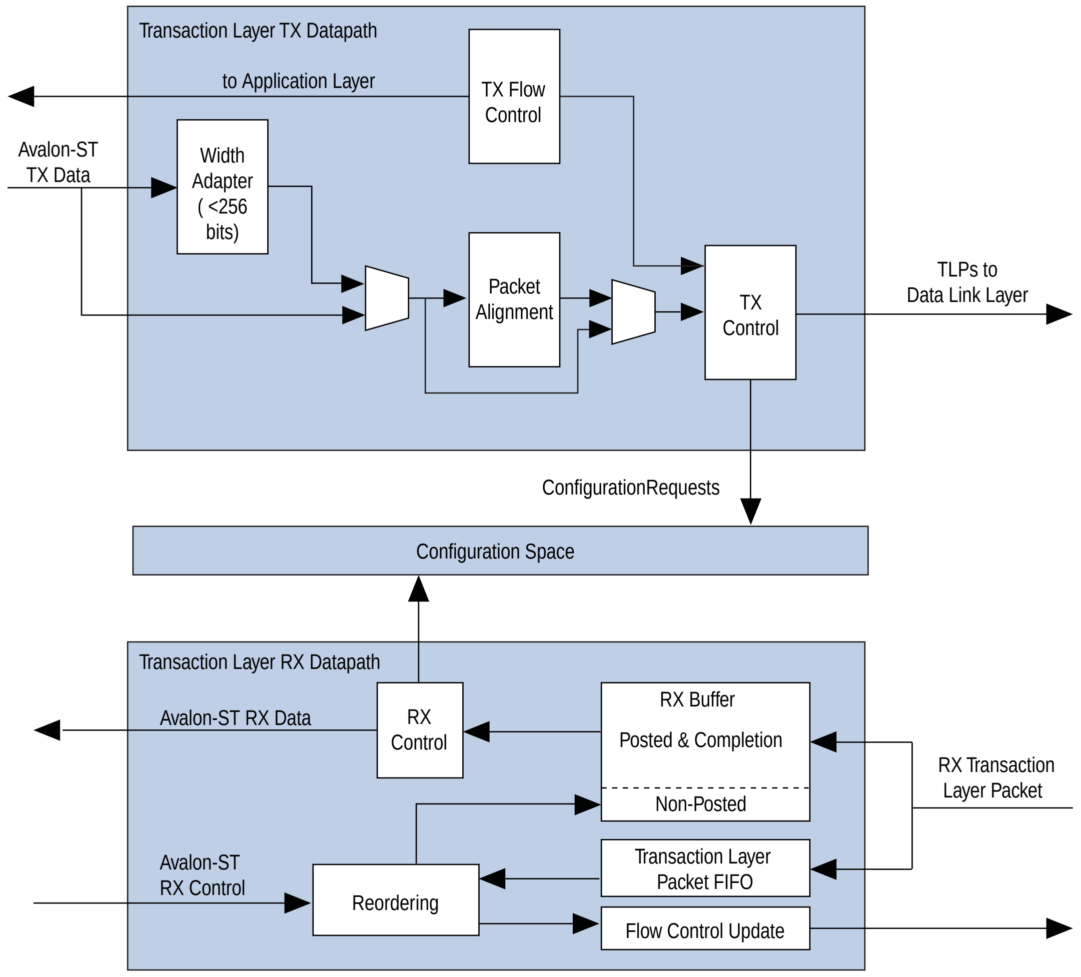
Tracing a transaction through the RX datapath includes the following steps:

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 7–3. Architecture of the Transaction Layer: Dedicated Receive Buffer



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 Revision 2.1*.

Refer To "Configuration Space Register Content" on page 9–1 or Chapter 7 in the *PCI Express Base Specification 2.1* for the complete content of these registers

Configuration Space Bypass Mode

When you select **Enable Configuration Space Bypass** under the **System Settings** heading of the parameter editor, the Arria V GZ 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 as illustrated in Figure 7–4. If you implement Configuration Space Bypass mode, the Configuration Shadow Extension Bus is not available. In Configuration Space Bypass mode, all received Type 0 configuration writes and reads are forwarded to the Avalon-ST interface.

In Configuration Space Bypass mode, you must also implement all of the TLP BAR matching and completion tag checking in soft logic.

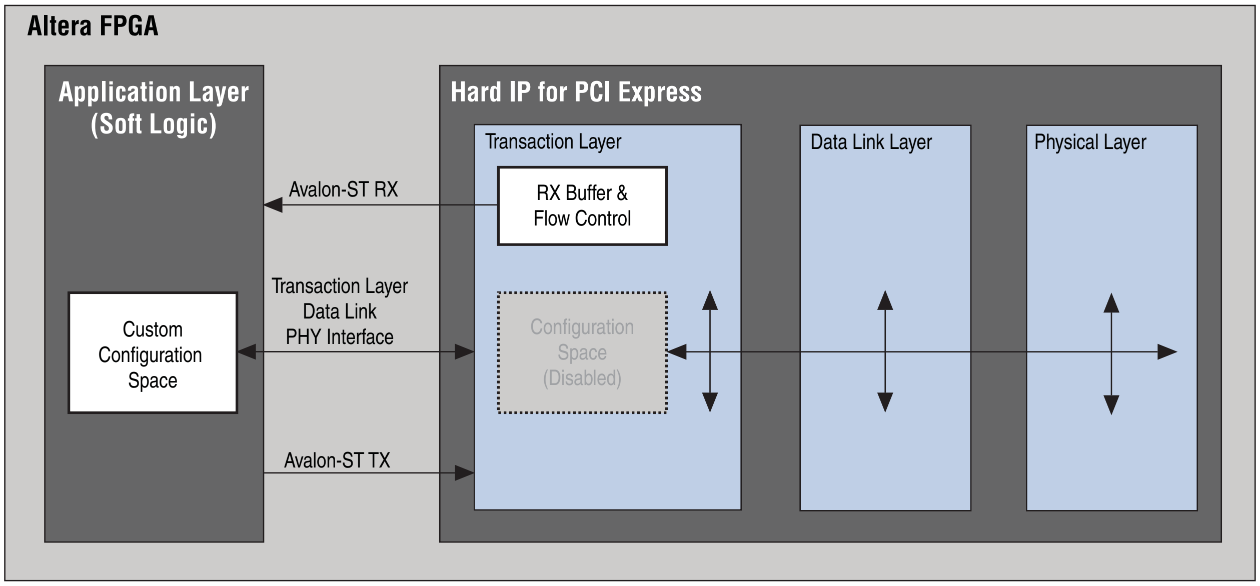
If you enable Configuration Space Bypass mode, you can implement the following features in soft logic:

- Resizable BARs
- Latency Tolerance Reporting
- Multicast
- Dynamic Power Allocation
- Alternative Routing-ID Interpretation (ARI)
- Single Root I/O Virtualization (SR-IOV)

- Multi-functions

The RX Buffer, Flow Control, DL and PHY layers from the Arria V GZ Hard IP for PCI Express are retained in the Hard IP.

Figure 7–4. Configuration Space Bypass Mode



In Configuration Space Bypass Mode, the hard IP passes all well-formed TLPs to the Application Layer using the Avalon-ST RX interface. The hard IP detects and drops malformed TLPs. Application Layer logic must detect and handle Unsupported Requests and Unexpected Completions. Application Layer logic must also generate all completions and messages and transmit them using the Avalon-ST TX interface. Refer to "Configuration Space Bypass Mode Input Signals" on page 8–42 and "Configuration Space Bypass Mode Output Signals" on page 8–44 for descriptions of these signals.

In Configuration Space Bypass Mode, the Power Management, MSI and legacy interrupts, Completion Errors, and Configuration Interfaces are disabled inside the hard IP. You must implement these features in the Application Layer. You can use the LMI bus in Configuration Space Bypass mode to log the TLP header of the first error in the AER registers.

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. Refer to "Errors Detected by the Transaction Layer" on page 15–3 for the malformed TLPs the Transaction Layer detects. 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.

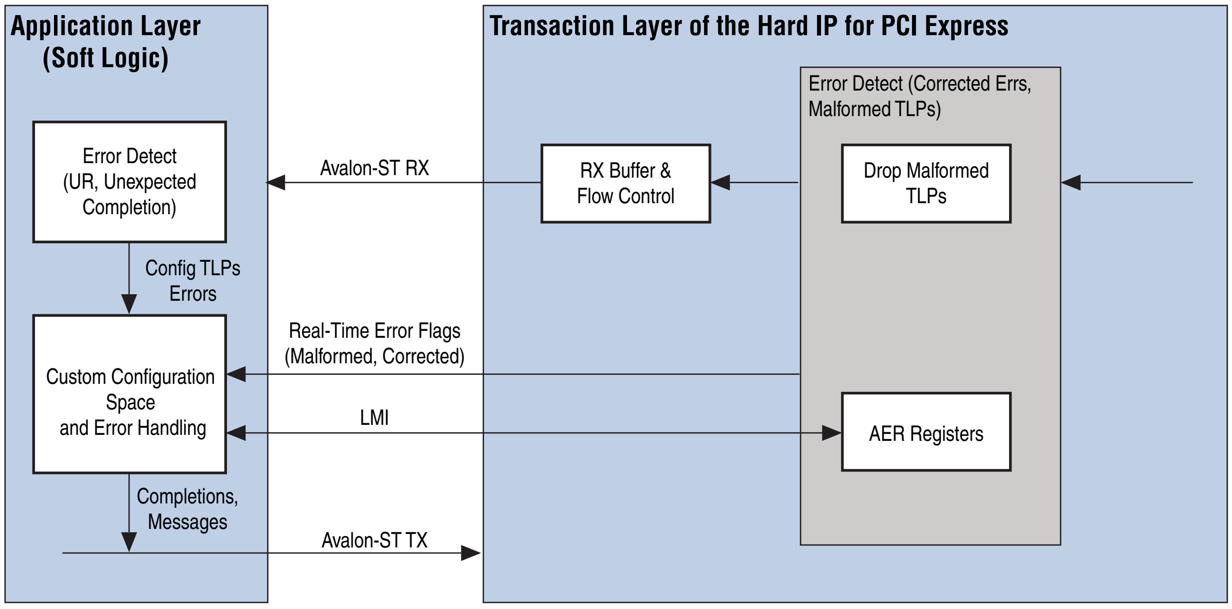
Because the AER header log is not available in Configuration Space Bypass Mode, the Application Layer must implement logic to read the AER header log using the LMI interface. You may need to arbitrate between Configuration Space Requests to the AER registers of the Hard IP for PCI Express and Configuration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

Altera 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 7–5 shows the division of error checking between the Transaction Layer of the hard IP for PCI Express and the Application Layer. Note that the real-time error flags assert for one `pld_clk` as the errors are detected by the Transaction Layer.

Figure 7–5. Error Handing in Configuration Space Bypass Mode


Application Layer (Soft Logic)
Error Detect (UR, Unexpected Completion)
Config TLPs Errors
Custom Configuration Space and Error Handling
Completions, Messages
Avalon-ST RX
Real-Time Error Flags (Malformed, Corrected)
LMI
Avalon-ST TX
Transaction Layer of the Hard IP for PCI Express
RX Buffer & Flow Control
Error Detect (Corrected Errs, Malformed TLPs)
Drop Malformed TLPs
AER Registers

The following 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 `rx_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. Refer to Table 12–1 on page 12–3 and Table 12–2 on page 12–3 for additional information.
- 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 `Uncorrectable Error Severity` error reporting bits appropriately so that the errors are logged appropriately internal to the Arria V GZ 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 V GZ 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.

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.

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 (as opposed to component communication by TLP transmission in the interconnect fabric).

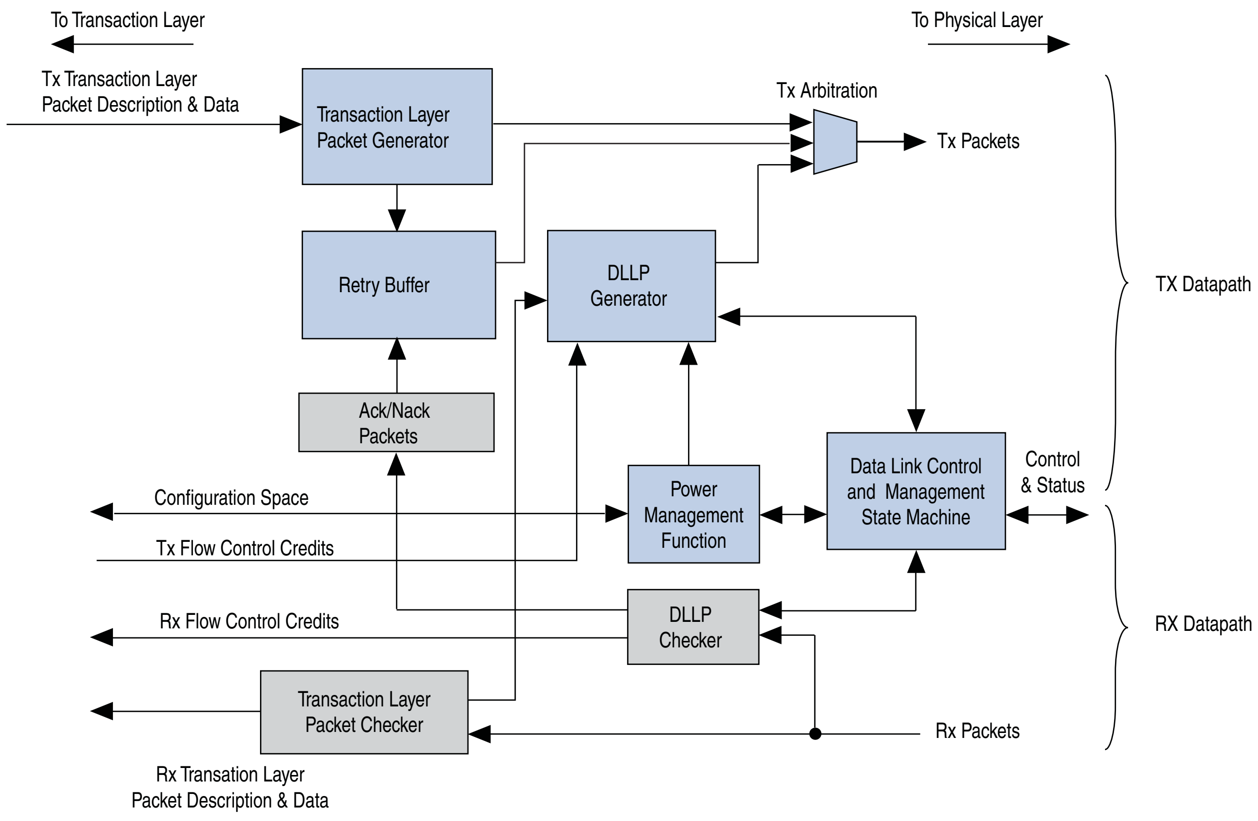
The DLL implements the following functions:

- Link management through the reception and transmission of DLL packets (DLLP), which are used for the following functions:
  - For power management of DLLP reception and transmission
  - To transmit and receive `ACK/NACK` packets
- Data integrity through generation and checking of CRCs for TLPs and DLLPs

- TLP retransmission in case of NAK DLLP reception using the retry 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 7–6 illustrates the architecture of the DLL.

Figure 7–6. Data Link Layer


To Transaction Layer
To Physical Layer
Tx Transaction Layer Packet Description & Data
Transaction Layer Packet Generator
Tx Arbitration
Tx Packets
Retry Buffer
DLLP Generator
TX Datapath
Ack/Nack Packets
Configuration Space
Power Management Function
Data Link Control and Management State Machine
Control & Status
Tx Flow Control Credits
Rx Flow Control Credits
DLLP Checker
RX Datapath
Transaction Layer Packet Checker
Rx Packets
Rx Transation Layer Packet Description & Data

The DLL has the following sub-blocks:

- Data Link Control and Management State Machine—This state machine is synchronized with the Physical Layer's LTSSM state machine and is also connected to the Configuration Space Registers. It initializes the link and flow control credits and reports status to the Configuration Space.
- 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.
- 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, generating a sequence number and a 32-bit 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. For 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:

  a. Initialize FC Data Link Layer packet

  b. ACK/NAK DLLP (high priority)

  c. Update FC DLLP (high priority)

  d. PM DLLP

  e. Retry buffer TLP

  f. TLP

  g. Update FC DLLP (low priority)

  h. ACK/NAK FC DLLP (low priority)

Physical Layer

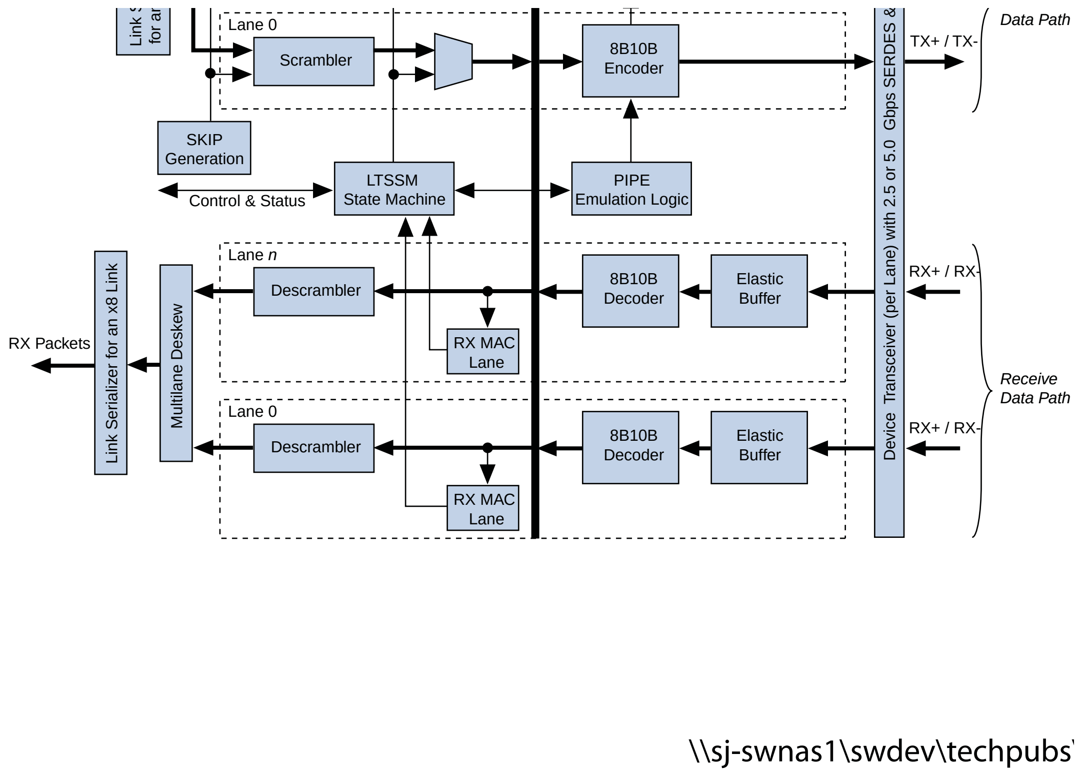
The Physical Layer is the lowest level of the Arria V GZ Hard IP for PCI Express. It is the layer closest to the link. It encodes and transmits packets across a link and accepts and decodes received packets. The Physical Layer connects to the link through a high-speed SERDES interface running at 2.5 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 Gen 3 implementations.

The Physical Layer is responsible for the following actions:

- Initializing the link
- Scrambling/descrambling and 8B/10B encoding/decoding of 2.5 Gbps (Gen1), 5.0 Gbps (Gen2), or 128b/130b encoding/decoding of 8.0 Gbps (Gen3) per lane
- Serializing and deserializing data
- 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

Figure 7–7 illustrates the Physical Layer architecture.

Figure 7–7. Physical Layer



The Physical Layer is subdivided by the PIPE Interface Specification into two layers (bracketed horizontally in Figure 7–7):

- Media Access Controller (MAC) Layer—The MAC layer includes the LTSSM and the scrambling/descrambling and multilane deskew functions.
- PHY Layer—The PHY layer includes the 8B/10B and 128b/130b encode/decode functions, elastic buffering, and serialization/deserialization functions.

The Physical Layer integrates both digital and analog elements. Intel designed the PIPE interface to separate the MAC from the PHY. The Arria V GZ Hard IP for PCI Express compiles with the PIPE interface specification.

The PHYMAC block is divided in 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 LTSTX sub-block. 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 what is received and transmitted 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 number of initialized lanes and the 64-bit data path.

  The multilane deskew implements an eight-word FIFO for each lane to store symbols. Each symbol includes eight data bits, one disparity bit, and one control bit. The FIFO discards the FTS, COM, and SKP symbols and replaces PAD and IDL with D0.0 data. When all eight FIFOs contain data, a read can occur.

  When the multilane lane deskew block is first enabled, each FIFO begins writing after the first COM is detected. If all lanes have not detected a COM symbol after seven clock cycles, they are reset and the resynchronization process restarts, or else the RX alignment function recreates a 64-bit data word which is sent to the DLL.

32-Bit PCI Express Avalon-MM Bridge

In Qsys, the Arria V GZ Hard IP for PCI Express is available with either an Avalon-ST or an Avalon-MM interface to the Application Layer. When you select the Avalon-MM Arria V GZ Hard IP for PCI Express, an Avalon-MM bridge module connects the PCI Express link to the interconnect fabric. The bridge facilitates the design of Endpoints and Root Ports that include Qsys components.

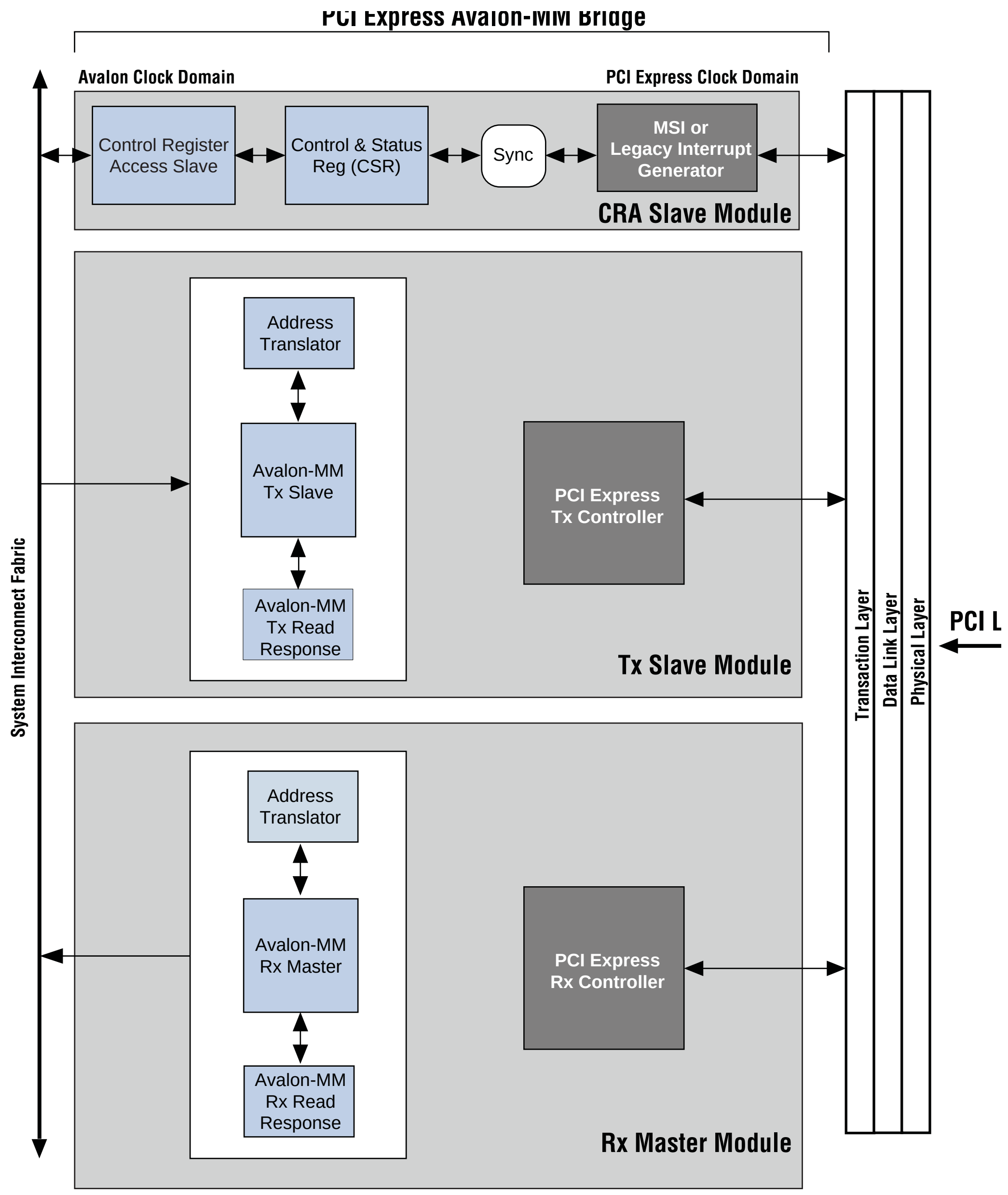
The full-featured Avalon-MM bridge provides three possible Avalon-MM ports: a bursting master, an optional bursting slave, and an optional non-bursting slave. The Avalon-MM bridge comprises the following three modules:

- TX Slave Module—This optional 64- or 128-bit bursting, Avalon-MM dynamic addressing slave port propagates read and write requests of up to 4 KBytes in size from the interconnect fabric to the PCI Express link. The bridge translates requests from the interconnect fabric to PCI Express request packets.
- RX Master Module—This 64- or 128-bit bursting Avalon-MM master port propagates PCI Express requests, converting them to bursting read or write requests to the interconnect fabric.

- Control Register Access (CRA) Slave Module—This optional, 32-bit Avalon-MM dynamic addressing slave port provides access to internal control and status registers from upstream PCI Express devices and external Avalon-MM masters. Implementations that use MSI or dynamic address translation require this port. The CRA port supports single dword read and write requests. It does not support bursting.

When you select the **Single dword completer** in the GUI for the Avalon-MM Hard IP for PCI Express, Qsys substitutes a unpipelined, 32-bit RX master port for the 64- or 128-bit full-featured RX master port. For more information about the 32-bit RX master refer to "Avalon-MM RX Master Block" on page 7–26. Figure 7–8 shows the block diagram of a full-featured PCI Express Avalon-MM bridge.

Figure 7–8. PCI Express Avalon-MM Bridge


PCI Express Avalon-MM Bridge
Avalon Clock Domain
PCI Express Clock Domain
Control Register Access Slave
Control & Status Reg (CSR)
Sync
MSI or Legacy Interrupt Generator
CRA Slave Module
Address Translator
Avalon-MM Tx Slave
Avalon-MM Tx Read Response
PCI Express Tx Controller
Tx Slave Module
Address Translator
Avalon-MM Rx Master
Avalon-MM Rx Read Response
PCI Express Rx Controller
Rx Master Module
System Interconnect Fabric
Transaction Layer
Data Link Layer
Physical Layer
PCI L

The bridge has the following additional characteristics:

- Type 0 and Type 1 vendor-defined incoming messages are discarded
- Completion-to-a-flush request is generated, but not propagated to the interconnect fabric

For End Points, each PCI Express base address register (BAR) in the Transaction Layer maps to a specific, fixed Avalon-MM address range. You can use separate BARs to map to various Avalon-MM slaves connected to the RX Master port. In contrast to Endpoints, Root Ports do not perform any BAR matching and forwards the address to a single RX Avalon-MM master port.

Avalon-MM Bridge TLPs

The PCI Express to Avalon-MM bridge translates the PCI Express read, write, and completion Transaction Layer Packets (TLPs) into standard Avalon-MM read and write commands typically used by master and slave interfaces. This PCI Express to Avalon-MM bridge also translates Avalon-MM read, write and read data commands to PCI Express read, write and completion TLPs. The following topics describe the Avalon-MM bridges translations.

Avalon-MM-to-PCI Express Write Requests

The Avalon-MM bridge accepts Avalon-MM burst write requests with a burst size of up to 512 Bytes at the Avalon-MM TX slave interface. The Avalon-MM bridge converts the write requests to one or more PCI Express write packets with 32– or 64-bit addresses based on the address translation configuration, the request address, and the maximum payload size.

The Avalon-MM write requests can start on any address in the range defined in the PCI Express address table parameters. The bridge splits incoming burst writes that cross a 4 KByte boundary into at least two separate PCI Express packets. The bridge also considers the root complex requirement for maximum payload on the PCI Express side by further segmenting the packets if needed.

The bridge requires Avalon-MM write requests with a burst count of greater than one to adhere to the following byte enable rules:

- The Avalon-MM byte enables must be asserted in the first qword of the burst.
- All subsequent byte enables must be asserted until the deasserting byte enable.
- The Avalon-MM byte enables may deassert, but only in the last qword of the burst.

☞ To improve PCI Express throughput, Altera recommends using an Avalon-MM burst master without any byte-enable restrictions.

Avalon-MM-to-PCI Express Upstream Read Requests

The PCI Express Avalon-MM bridge converts read requests from the system interconnect fabric to PCI Express read requests with 32-bit or 64-bit addresses based on the address translation configuration, the request address, and the maximum read size.

The Avalon-MM TX slave interface of a PCI Express Avalon-MM bridge can receive read requests with burst sizes of up to 512 bytes sent to any address. However, the bridge limits read requests sent to the PCI Express link to a maximum of 256 bytes. Additionally, the bridge must prevent each PCI Express read request packet from crossing a 4 KByte address boundary. Therefore, the bridge may split an Avalon-MM read request into multiple PCI Express read packets based on the address and the size of the read request.

For Avalon-MM read requests with a burst count greater than one, all byte enables must be asserted. There are no restrictions on byte enables for Avalon-MM read requests with a burst count of one. An invalid Avalon-MM request can adversely affect system functionality, resulting in a completion with the abort status set. An example of an invalid request is one with an incorrect address.

PCI Express-to-Avalon-MM Read Completions

The PCI Express Avalon-MM bridge returns read completion packets to the initiating Avalon-MM master in the issuing order. The bridge supports multiple and out-of-order completion packets.

PCI Express-to-Avalon-MM Downstream Write Requests

The PCI Express Avalon-MM bridge receives PCI Express write requests, it converts them to burst write requests before sending them to the interconnect fabric. For Endpoints, the bridge translates the PCI Express address to the Avalon-MM address space based on the BAR hit information and on address translation table values configured during the IP core parameterization. For Root Ports, all requests are forwarded to a single RX Avalon-MM master that drives them to the interconnect fabric. Malformed write packets are dropped, and therefore do not appear on the Avalon-MM interface.

For downstream write and read requests, if more than one byte enable is asserted, the byte lanes must be adjacent. In addition, the byte enables must be aligned to the size of the read or write request.

As an example, Table 7–2 lists the byte enables for 32-bit data.

Table 7–2. Valid Byte Enable Configurations

Byte Enable Value	Description
4’b1111	Write full 32 bits
4’b0011	Write the lower 2 bytes
4’b1100	Write the upper 2 bytes
4’b0001	Write byte 0 only
4’b0010	Write byte 1 only
4’b0100	Write byte 2 only
4’b1000	Write byte 3 only

In burst mode, the Arria V GZ Hard IP for PCI Express supports only byte enable values that correspond to a contiguous data burst. For the 32-bit data width example, valid values in the first data phase are 4’b1111, 4’b1110, 4’b1100, and 4’b1000, and valid values in the final data phase of the burst are 4’b1111, 4’b0111, 4’b0011, and 4’b0001. Intermediate data phases in the burst can only have byte enable value 4’b1111.

PCI Express-to-Avalon-MM Downstream Read Requests

The PCI Express Avalon-MM bridge sends PCI Express read packets to the interconnect fabric as burst reads with a maximum burst size of 512 bytes. For Endpoints, the bridge converts the PCI Express address to the Avalon-MM address space based on the BAR hit information and address translation lookup table values. The RX Avalon-MM master port drives the received address to the fabric. You can set up the Address Translation Table Configuration in the GUI. Unsupported read requests generate a completer abort response. For more information about optimizing BAR addresses, refer to Minimizing BAR Sizes and the PCIe Address Space.

Avalon-MM-to-PCI Express Read Completions

The PCI Express Avalon-MM bridge converts read response data from Application Layer Avalon-MM slaves to PCI Express completion packets and sends them to the Transaction Layer.

A single read request may produce multiple completion packets based on the **Maximum payload size** and the size of the received read request. For example, if the read is 512 bytes but the **Maximum payload size** 128 bytes, the bridge produces four completion packets of 128 bytes each. The bridge does not generate out-of-order completions. You can specify the **Maximum payload size** parameter on the **Device** tab under the **PCI Express/PCI Capabilities** heading in the GUI. Refer to “PCI Express and PCI Capabilities Parameters” on page 6–6.

PCI Express-to-Avalon-MM Address Translation for Endpoints for 32-Bit Bridge

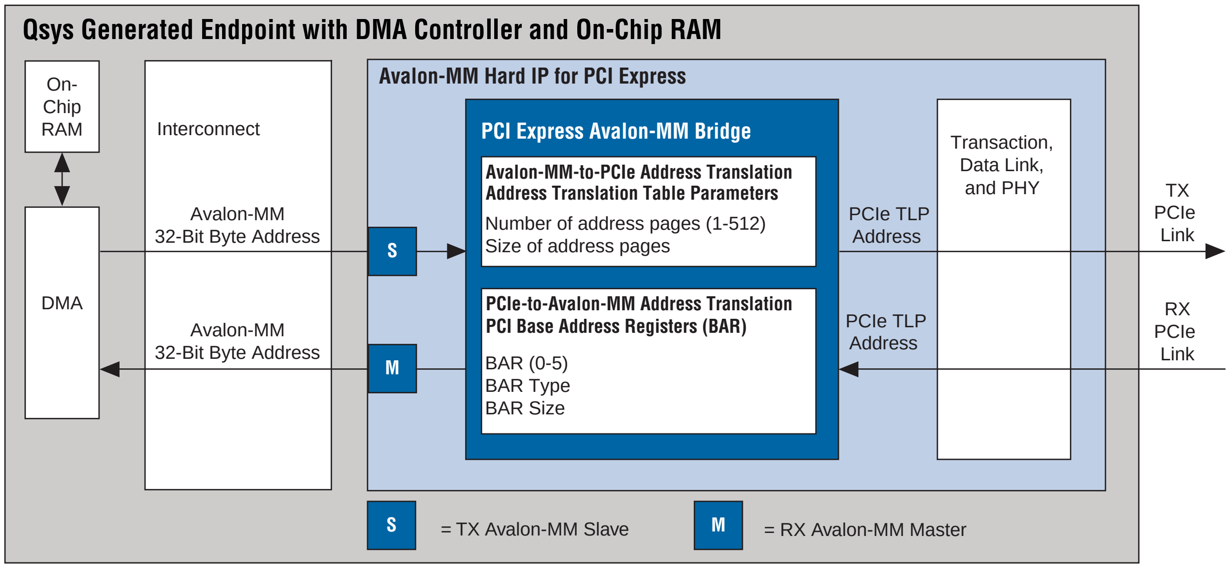
The PCI Express Avalon-MM Bridge translates the system-level physical addresses, typically up to 64 bits, to the significantly smaller addresses required by the Application Layer’s Avalon-MM slave components.

☞ Starting with the 13.0 version of the Quartus II software, the PCI Express-to-Avalon-MM bridge supports both 32- and 64-bit addresses. If you select 64-bit addressing the bridge does not perform address translation. It drives the addresses specified to the interconnect fabric. You can limit the number of address bits used by Avalon-MM slave components to the actual size required by specifying the address size in the Avalon-MM slave component GUI.

You can specify up to six BARs for address translation when you customize your Hard IP for PCI Express as described in “Base Address Register (BAR) and Expansion ROM Settings” on page 6–4. When 32-bit addresses are specified, the PCI Express Avalon-MM Bridge also translates Application Layer addresses to system-level physical addresses as described in “Avalon-MM-to-PCI Express Address Translation Algorithm for 32-Bit Addressing” on page 7–23.

Figure 7–9 provides a high-level view of address translation in both directions.

Figure 7–9. Address Translation in TX and RX Directions For Endpoints


Qsys Generated Endpoint with DMA Controller and On-Chip RAM
On-Chip RAM
Interconnect
DMA
Avalon-MM 32-Bit Byte Address
Avalon-MM 32-Bit Byte Address
Avalon-MM Hard IP for PCI Express
PCI Express Avalon-MM Bridge
Avalon-MM-to-PCIe Address Translation
Address Translation Table Parameters
Number of address pages (1-512)
Size of address pages
PCIe-to-Avalon-MM Address Translation
PCI Base Address Registers (BAR)
BAR (0-5)
BAR Type
BAR Size
S
M
PCIe TLP Address
PCIe TLP Address
Transaction, Data Link, and PHY
TX PCIe Link
RX PCIe Link
S = TX Avalon-MM Slave
M = RX Avalon-MM Master

When configured as a Root Port, a single RX Avalon-MM master forwards all RX TLPs to the Qsys interconnect.

The Avalon-MM RX master module port has an 8-byte datapath in 64-bit mode and a 16-byte datapath in 128-bit mode. The Qsys interconnect fabric manages mismatched port widths transparently.

As Memory Request TLPs are received from the PCIe link, the most significant bits are used in the BAR matching as described in the PCI specifications. The least significant bits not used in the BAR match process are passed unchanged as the Avalon-MM address for that BAR's RX Master port.

For example, consider the following configuration specified using the Base Address Registers in the GUI.

1. BAR1:0 is a **64-bit prefetchable memory** that is **4KBytes -12 bits**
2. System software programs BAR1:0 to have a base address of 0x00001234 56789000
3. A TLP received with address 0x00001234 56789870
4. The upper 52 bits (0x0000123456789) are used in the BAR matching process, so this request matches.
5. The lower 12 bits, 0x870, are passed through as the Avalon address on the Rxm_BAR0 Avalon-MM Master port. The BAR matching software replaces the upper 20 bits of the address with the Avalon-MM base address.

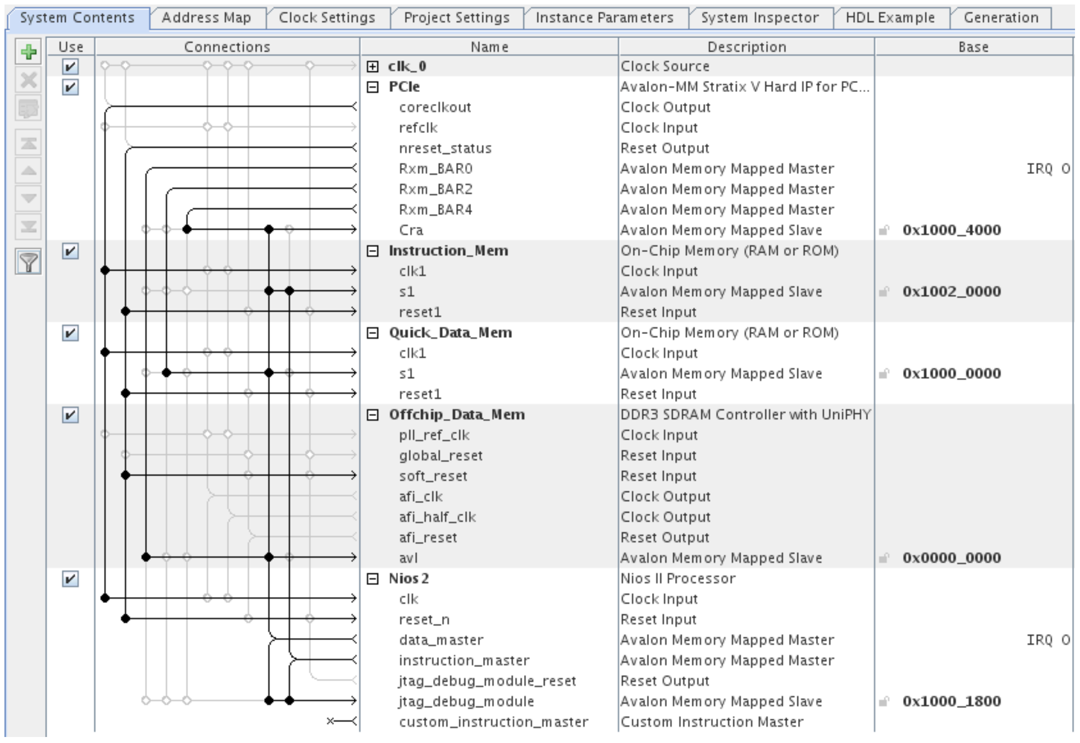
Minimizing BAR Sizes and the PCIe Address Space

For designs that include multiple BARs, you may need to modify the base address assignments auto-assigned by Qsys in order to minimize the address space that the BARs consume. For example, consider a Qsys system with the following components:

- **Offchip_Data_Mem DDR3** (SDRAM Controller with UniPHY) controlling 256 MBytes of memory—Qsys auto-assigned a base address of 0x00000000
- **Quick_Data_Mem** (On-Chip Memory (RAM or ROM)) of 4 KBytes—Qsys auto-assigned a base address of 0x10000000
- **Instruction_Mem** (On-Chip Memory (RAM or ROM)) of 64 KBytes—Qsys auto-assigned a base address of 0x10020000
- **PCIe** (Avalon-MM Arria V GZ Hard IP for PCI Express)
  - **Cra** (Avalon-MM Slave)—auto assigned base address of 0x10004000
  - **Rxm_BAR0** connects to **Offchip_Data_Mem DDR3 avl**
  - **Rxm_BAR2** connects to **Quick_Data_Mem s1**
  - **Rxm_BAR4** connects to PCIe. **Cra Avalon-MM Slave**
- **Nios2** (Nios® II Processor)
  - **data_master** connects to **PCIe Cra**, **Offchip_Data_Mem DDR3 avl**, **Quick_Data_Mem s1**, **Instruction_Mem s1**, **Nios2 jtag_debug_module**
  - **instruction_master** connects to **Instruction_Mem s1**

Figure 7–10 illustrates this Qsys system. (Figure 7–10 uses a filter to hide the Conduit interfaces that are not relevant in this discussion.)

Figure 7–10. Qsys System for PCI Express with Poor Address Space Utilization

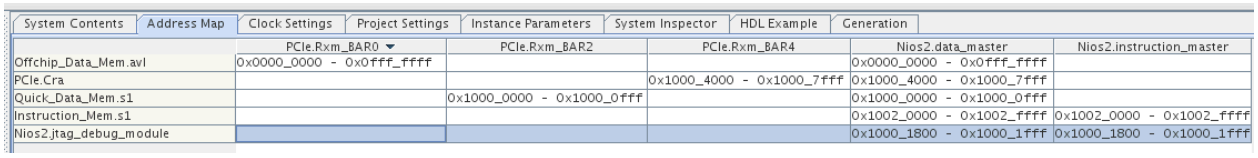


System Contents | Address Map | Clock Settings | Project Settings | Instance Parameters | System Inspector | HDL Example | Generation

Use	Connections	Name	Description	Base
✓		⊞ **clk_0**	Clock Source	
✓		⊟ **PCIe**	Avalon-MM Stratix V Hard IP for PC...	
		coreclkout	Clock Output	
		refclk	Clock Input	
		nreset_status	Reset Output	
		Rxm_BAR0	Avalon Memory Mapped Master	IRQ 0
		Rxm_BAR2	Avalon Memory Mapped Master	
		Rxm_BAR4	Avalon Memory Mapped Master	
		Cra	Avalon Memory Mapped Slave	**0x1000_4000**
✓		⊟ **Instruction_Mem**	On-Chip Memory (RAM or ROM)	
		clk1	Clock Input	
		s1	Avalon Memory Mapped Slave	**0x1002_0000**
		reset1	Reset Input	
✓		⊟ **Quick_Data_Mem**	On-Chip Memory (RAM or ROM)	
		clk1	Clock Input	
		s1	Avalon Memory Mapped Slave	**0x1000_0000**
		reset1	Reset Input	
✓		⊟ **Offchip_Data_Mem**	DDR3 SDRAM Controller with UniPHY	
		pll_ref_clk	Clock Input	
		global_reset	Reset Input	
		soft_reset	Reset Input	
		afi_clk	Clock Output	
		afi_half_clk	Clock Output	
		afi_reset	Reset Output	
		avl	Avalon Memory Mapped Slave	**0x0000_0000**
✓		⊟ **Nios2**	Nios II Processor	
		clk	Clock Input	
		reset_n	Reset Input	
		data_master	Avalon Memory Mapped Master	IRQ 0
		instruction_master	Avalon Memory Mapped Master	
		jtag_debug_module_reset	Reset Output	
		jtag_debug_module	Avalon Memory Mapped Slave	**0x1000_1800**
		custom_instruction_master	Custom Instruction Master	

Figure 7–11 illustrates the address map for this system.

Figure 7–11. Poor Address Map



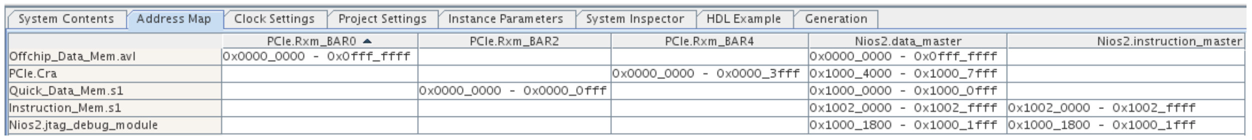
	PCIe.Rxm_BAR0	PCIe.Rxm_BAR2	PCIe.Rxm_BAR4	Nios2.data_master	Nios2.instruction_master
Offchip_Data_Mem.avl	0x0000_0000 - 0x0fff_ffff			0x0000_0000 - 0x0fff_ffff	
PCIe.Cra			0x1000_4000 - 0x1000_7fff	0x1000_4000 - 0x1000_7fff	
Quick_Data_Mem.s1		0x1000_0000 - 0x1000_0fff		0x1000_0000 - 0x1000_0fff	
Instruction_Mem.s1				0x1002_0000 - 0x1002_ffff	0x1002_0000 - 0x1002_ffff
Nios2.jtag_debug_module				0x1000_1800 - 0x1000_1fff	0x1000_1800 - 0x1000_1fff

The auto-assigned base addresses result in the following three large BARs:

- BAR0 is 28 bits. This is the optimal size because it addresses the **Offchip_Data_Mem** which requires 28 address bits.
- BAR2 is 29 bits. BAR2 addresses the **Quick_Data_Mem** which is 4 KBytes;. It should only require 12 address bits; however, it is consuming 512 MBytes of address space.
- BAR4 is also 29 bits. BAR4 address **PCIe Cra** which is 16 KBytes. It should only require 14 address bits; however, it is also consuming 512 MBytes of address space.

This design is consuming 1.25GB of PCIe address space when only 276 MBytes are actually required. The solution is to edit the address map to place the base address of each BAR at 0x0000_0000. Figure 7–12 illustrates the optimized address map.

Figure 7–12. Optimized Address Map

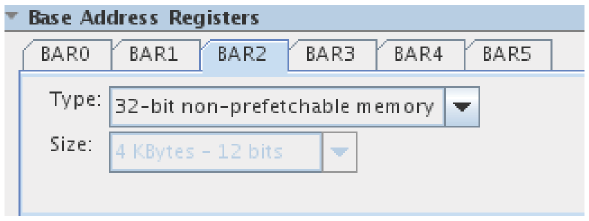


	PCIe.Rxm_BAR0	PCIe.Rxm_BAR2	PCIe.Rxm_BAR4	Nios2.data_master	Nios2.instruction_master
Offchip_Data_Mem.avl	0x0000_0000 - 0x0fff_ffff			0x0000_0000 - 0x0fff_ffff	
PCIe.Cra			0x0000_0000 - 0x0000_3fff	0x1000_4000 - 0x1000_7fff	
Quick_Data_Mem.s1		0x0000_0000 - 0x0000_0fff		0x1000_0000 - 0x1000_0fff	
Instruction_Mem.s1				0x1002_0000 - 0x1002_ffff	0x1002_0000 - 0x1002_ffff
Nios2.jtag_debug_module				0x1000_1800 - 0x1000_1fff	0x1000_1800 - 0x1000_1fff

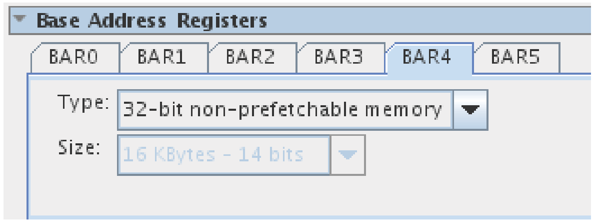
For more information about changing Qsys addresses using the Qsys address map, refer to Address Map Tab (Qsys) in Quartus II Help.

Figure 7–13 shows the number of address bits required when the smaller memories accessed by BAR2 and BAR4 have a base address of 0x0000_0000.

Figure 7–13. Reduced Address Bits for BAR2 and BAR4



Type: 32-bit non-prefetchable memory
Size: 4 KBytes - 12 bits



Type: 32-bit non-prefetchable memory
Size: 16 KBytes - 14 bits

For cases where the BAR Avalon-MM RX master port connects to more than one Avalon-MM slave, assign the base addresses of the slaves sequentially and place the slaves in the smallest power-of-two-sized address space possible. Doing so minimizes the system address space used by the BAR.

Avalon-MM-to-PCI Express Address Translation Algorithm for 32-Bit Addressing

Starting with the 13.0 version of the Quartus II software, the PCI Express-to-Avalon-MM bridge supports both 32- and 64-bit addresses. If you select 64-bit addressing the bridge does not perform address translation.

When you specify 32-bit addresses, the Avalon-MM address of a received request on the TX Avalon-MM slave port is translated to the PCI Express address before the request packet is sent to the Transaction Layer. You can specify up to 512 address pages and sizes ranging from 4 KByte to 4 GBytes when you customize your Avalon-MM Arria V GZ Hard IP for PCI Express as described in "Avalon to PCIe Address Translation Settings" on page 6–13. This address translation process proceeds by replacing the MSB of the Avalon-MM address with the value from a specific translation table entry; the LSB remains unchanged. The number of MSBs to be replaced is calculated based on the total address space of the upstream PCI Express devices that the Avalon-MM Hard IP for PCI Express can access. The number of MSB bits is defined by the difference between the maximum number of bits required to represent the address space supported by the upstream PCI Express device minus the number of bits required to represent the **Size of address pages** which are the LSB pass-through bits (*N*). The **Size of address pages** (*N*) is applied to all entries in the translation table.

Each of the 512 possible entries corresponds to the base address of a PCI Express memory segment of a specific size. The segment size of each entry must be identical. The total size of all the memory segments is used to determine the number of address MSB to be replaced. In addition, each entry has a 2-bit field, `Sp[1:0]`, that specifies 32-bit or 64-bit PCI Express addressing for the translated address. Refer to Figure 7–14 on page 7–24. The most significant bits of the Avalon-MM address are used by the interconnect fabric to select the slave port and are not available to the slave. The next most significant bits of the Avalon-MM address index the address translation entry to be used for the translation process of MSB replacement.

For example, if the core is configured with an address translation table with the following attributes:

- **Number of Address Pages—16**
- **Size of Address Pages—1 MByte**
- **PCI Express Address Size—64 bits**

then the values in Figure 7–14 are:

- *N* = 20 (due to the 1 MByte page size)
- *Q* = 16 (number of pages)
- *M* = 24 (20 + 4 bit page selection)
- *P* = 64

In this case, the Avalon address is interpreted as follows:

- Bits [31:24] select the TX slave module port from among other slaves connected to the same master by the system interconnect fabric. The decode is based on the base addresses assigned in Qsys.

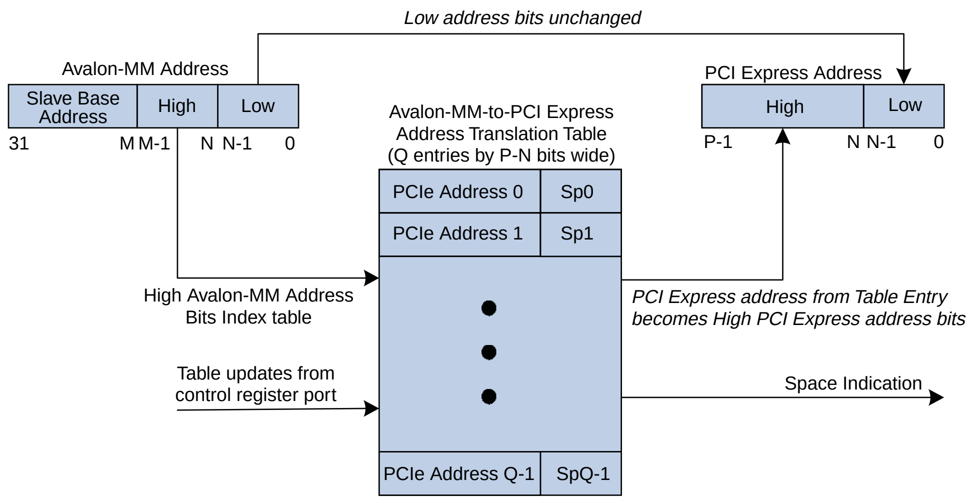
- Bits [23:20] select the address translation table entry.
- Bits [63:20] of the address translation table entry become PCI Express address bits [63:20].
- Bits [19:0] are passed through and become PCI Express address bits [19:0].

The address translation table is dynamically configured at run time. The address translation table is implemented in memory and can be accessed through the CRA slave module. Dynamic configuration is optimal in a typical PCI Express system where address allocation occurs after BIOS initialization.

For more information about how to access the dynamic address translation table through the CRA slave, refer to the "Avalon-MM-to-PCI Express Address Translation Table 0x1000–0x1FFF" on page 9–15.

Figure 7–14 depicts the Avalon-MM-to-PCI Express address translation process.

Figure 7–14. Avalon-MM-to-PCI Express Address Translation (1), (2), (3), (4), (5)


Low address bits unchanged
Avalon-MM Address
Slave Base Address
High
Low
31
M M-1
N N-1
0
PCI Express Address
High
Low
P-1
N N-1
0
Avalon-MM-to-PCI Express Address Translation Table (Q entries by P-N bits wide)
PCIe Address 0
Sp0
PCIe Address 1
Sp1
PCIe Address Q-1
SpQ-1
High Avalon-MM Address Bits Index table
PCI Express address from Table Entry becomes High PCI Express address bits
Table updates from control register port
Space Indication

Notes to Figure 7–14:

(1) *N* is the number of pass-through bits.
(2) *M* is the number of Avalon-MM address bits.
(3) *P* is the number of PCI Express address bits.
(4) *Q* is the number of translation table entries.
(5) `Sp[1:0]` is the space indication for each entry.

Completer Only Single Dword Endpoint

The completer only single dword endpoint is intended for applications that use the PCI Express protocol to perform simple read and write register accesses from a host CPU. The completer only single dword endpoint is a hard IP implementation available for Qsys systems, and includes an Avalon-MM interface to the Application Layer. The Avalon-MM interface connection in this variation is 32 bits wide. This endpoint is not pipelined; at any time a single request can be outstanding.

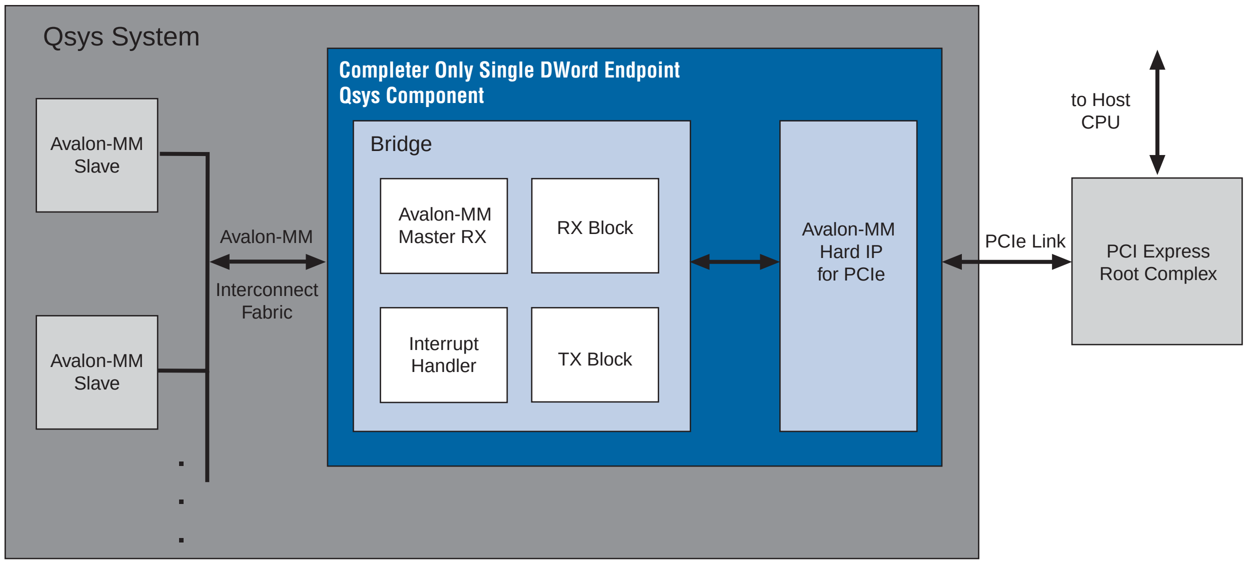
The completer-only single dword endpoint supports the following requests:

- Read and write requests of a single dword (32 bits) from the Root Complex

- Completion with Completer Abort status generation for other types of non-posted requests
- INTX or MSI support with one Avalon-MM interrupt source

Figure 7–15 shows a Qsys system that includes a completer-only single dword endpoint.

Figure 7–15. Qsys Design Including Completer Only Single Dword Endpoint for PCI Express



As Figure 7–15 illustrates, the completer-only single dword endpoint connects to a PCI Express root complex. A bridge component includes the Arria V GZ Hard IP for PCI Express TX and RX blocks, an Avalon-MM RX master, and an interrupt handler. The bridge connects to the FPGA fabric using an Avalon-MM interface. The following sections provide an overview of each block in the bridge.

RX Block

The RX Block control logic interfaces to the hard IP block to process requests from the root complex. It supports memory reads and writes of a single dword. It generates a completion with Completer Abort (CA) status for read requests greater than four bytes and discards all write data without further action for write requests greater than four bytes.

The RX block passes header information to the Avalon-MM master, which generates the corresponding transaction to the Avalon-MM interface. The bridge accepts no additional requests while a request is being processed. While processing a read request, the RX block deasserts the `ready` signal until the TX block sends the corresponding completion packet to the hard IP block. While processing a write request, the RX block sends the request to the Avalon-MM interconnect fabric before accepting the next request.

Avalon-MM RX Master Block

The 32-bit Avalon-MM master connects to the Avalon-MM interconnect fabric. It drives read and write requests to the connected Avalon-MM slaves, performing the required address translation. The RX master supports all legal combinations of byte enables for both read and write requests.

For more information about legal combinations of byte enables, refer to *Chapter 3, Avalon Memory-Mapped Interfaces* in the *Avalon Interface Specifications.*

TX Block

The TX block sends completion information to the Avalon-MM Hard IP for PCI Express which sends this information to the root complex. The TX completion block generates a completion packet with Completer Abort (CA) status and no completion data for unsupported requests. The TX completion block also supports the zero-length read (flush) command.

Interrupt Handler Block

The interrupt handler implements both INTX and MSI interrupts. The `msi_enable` bit in the configuration register specifies the interrupt type. The `msi_enable_bit` is part of the MSI message control portion in the MSI Capability structure. It is bit[16] of address 0x050 in the Configuration Space registers. If the `msi_enable` bit is on, an MSI request is sent to the Arria V GZ Hard IP for PCI Express when received, otherwise INTX is signaled. The interrupt handler block supports a single interrupt source, so that software may assume the source. You can disable interrupts by leaving the interrupt signal unconnected in the IRQ column of Qsys.

When the MSI registers in the Configuration Space of the Completer Only Single Dword Arria V GZ Hard IP for PCI Express are updated, there is a delay before this information is propagated to the Bridge module shown in Figure 7–15. You must allow time for the Bridge module to update the MSI register information. Normally, setting up MSI registers occurs during enumeration process. Under normal operation, initialization of the MSI registers should occur substantially before any interrupt is generated. However, failure to wait until the update completes may result in any of the following behaviors:

- Sending a legacy interrupt instead of an MSI interrupt
- Sending an MSI interrupt instead of a legacy interrupt
- Loss of an interrupt request

According to the *PCI Express Base Specification*, if `MSI_enable`=0 and the `Disable Legacy Interrupt bit`=1 in the Configuration Space `Command` register (0x004), the Hard IP should not send legacy interrupt messages when an interrupt is generated.

8. IP Core Interfaces

This chapter describes the signals that are part of the Arria V GZ Hard IP for PCI Express. It describes the top-level signals in the following variants:

- Avalon-ST Hard IP for PCI Express Top-Level Signals
- Avalon-MM Hard IP for PCI Express Top-Level Signals

Variants using the Avalon-ST interface are available in both the MegaWizard Plug-In Manager and the Qsys design flows. Variants using the Avalon-MM interface are only available in the Qsys design flow. Variants using the Avalon-MM interface may be easier to understand because a PCI Express to Avalon-MM bridge translates the PCI Express read, write and completion TLPs into standard Avalon-MM read and write commands. The Avalon-MM reads and write commands are the same as those used by master and slave interfaces to access memories and registers. Consequently, you do not need a detailed understanding of the PCI Express TLPs to use the Avalon-MM variants. Refer to "Hard IP for PCI Express Features" on page 1–3 to learn about the difference in the features available for the Avalon-ST and Avalon-MM interfaces.

The Arria V GZ Hard IP for PCI Express with the Avalon-ST interface offers exactly the same features in the MegaWizard and Qsys design flows. Your decision about which design flow to use depends on whether you want to integrate using RTL instantiation or Qsys. The Qsys system integration tool automatically generates the interconnect logic between the IP components in your system, saving time and effort. Refer to "MegaWizard Plug-In Manager Design Flow" on page 2–3 and "Qsys Design Flow" on page 2–8 for a description of the steps involved in the two design flows.

Figure 8–1 illustrates the top-level signals in the Arria V GZ Hard IP for PCI Express using the Avalon-ST interface.

Figure 8–1. Avalon-ST Hard IP for PCI Express Top-Level Signals

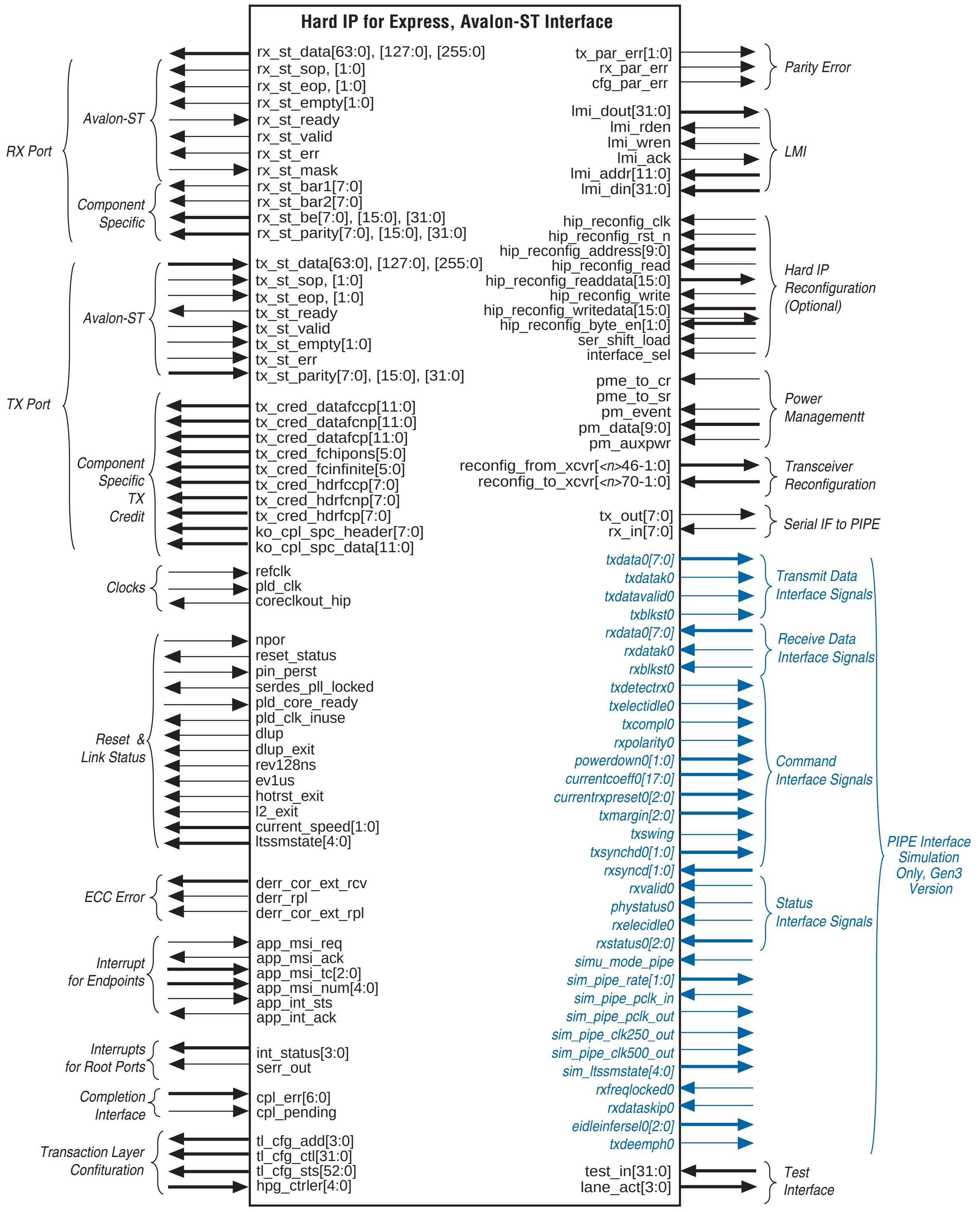


Table 8–1 lists the interfaces with links to the subsequent sections that describe each interface. The signals are described in the order in which they are shown in Figure 8–1.

Table 8–1. Signal Groups in the Arria V GZ Hard IP for PCI Express

Signal Group	Description
Logical	
Avalon-ST RX	"Avalon-ST RX Interface" on page 8–4
Avalon-ST TX	"Avalon-ST TX Interface" on page 8–17
Clocks	"Clock Signals" on page 8–28
Reset and status	"Reset Signals, Status, and Link Training Signals" on page 8–28
ECC error	"Error Signals" on page 8–31
Interrupt for Endpoints	"Interrupts for Endpoints" on page 8–32
Interrupt for Root Ports	"Interrupts for Root Ports" on page 8–34
Completion	"Completion Side Band Signals" on page 8–34
Configuration space	"Transaction Layer Configuration Space Signals" on page 8–36
Configuration Space Bypass mode	"Configuration Space Bypass Mode Interface Signals" on page 8–42
Parity Error	"Parity Signals" on page 8–46
LMI	"LMI Signals" on page 8–47
Hard IP reconfiguration block	"Hard IP Reconfiguration Interface" on page 8–49
Power management	"Power Management Signals" on page 8–51
Physical	
Transceiver control	"Transceiver Reconfiguration" on page 8–59
Serial	"Serial Interface Signals" on page 8–60
PIPE (1)	"PIPE Interface Signals" on page 8–66
Test	
Test	"Test Signals" on page 8–69

Note to Table 8–1:

(1) Provided for simulation only

Avalon-ST RX Interface

Table 8–2 describes the signals that comprise the Avalon-ST RX Datapath. The RX data signal can be 64, 128, or 256 bits.

Table 8–2. 64-, 128-, or 256-Bit Avalon-ST RX Datapath (Part 1 of 4)

Signal	Width	Dir	Avalon-ST Type	Description
`rx_st_data`	64, 128, 256	O	`data`	Receive data bus. Refer to Figure 8–3 through Figure 8–17 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, 2	O	`start of packet`	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, 2	O	`end of packet`	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.

Table 8–2. 64-, 128-, or 256-Bit Avalon-ST RX Datapath (Part 2 of 4)

Signal	Width	Dir	Avalon-ST Type	Description
`rx_st_empty[1:0]`	1, 2	O	`empty`	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_empty` = 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 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: ■ `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`	1	I	`ready`	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.

Table 8–2. 64-, 128-, or 256-Bit Avalon-ST RX Datapath (Part 3 of 4)

Signal	Width	Dir	Avalon-ST Type	Description
`rx_st_valid`	1, 2	O	`valid`	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`	1, 2	O	`error`	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 II 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. Altera recommends resetting the Arria V GZ Hard IP for PCI Express when an uncorrectable (double-bit) ECC error is detected.
Component Specific Signals				
`rx_st_mask`	1	I	`component specific`	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.
`rx_st_bardec1` `rx_st_bardec2`	8	O	`component specific`	The decoded BAR bits for the TLP. Valid for `MRd`, `MWr`, `IOWR`, and `IORD` TLPs; ignored for the completion or message TLPs. `rx_st_bardec1` is valid on the first cycle of `rx_st_data` for TLPs that begin in the lower 2 qwords of `rx_st_data`([127:0]). When using a 256-bit Avalon-ST bus with **Multiple packets per cycle**, `rx_st_bardec2` is valid on the first cycle of `rx_st_data` for TLPs that begin in the upper 2 qwords of `rx_st_data`([255:128]). Figure 8–6 and Figure 8–9 illustrate the timing of this signal for 64- and 128-bit data, respectively.

Table 8–2. 64-, 128-, or 256-Bit Avalon-ST RX Datapath (Part 4 of 4)

Signal	Width	Dir	Avalon-ST Type	Description

`rx_st_bardec1`
`rx_st_bardec2`
(continued)
				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
				■

`rx_st_be`
(deprecated)
	8,
	16, 32
		O	component specific	Byte enables corresponding to the `rx_st_data`. The byte enable signals only apply to PCI Express Memory Write and I/O Write TLP payload fields. When using 64-bit Avalon-ST bus, the width of `rx_st_be` is 8 bits. When using 128-bit Avalon-ST bus, the width of `rx_st_be` is 16 bits. When using a 256-bit Avalon-ST bus, the width of `rx_st_be` is 32 bits. This signal is optional. You can derive the same information by decoding the `FBE` and `LBE` fields in the TLP header. The byte enable bits correspond to data bytes as follows:
				`rx_st_data[63:56] = rx_st_be[7]`
				`rx_st_data[55:48] = rx_st_be[6]`
				`rx_st_data[47:40] = rx_st_be[5]`
				`rx_st_data[39:32] = rx_st_be[4]`
				`rx_st_data[31:24] = rx_st_be[3]`
				`rx_st_data[23:16] = rx_st_be[2]`
				`rx_st_data[15:8] = rx_st_be[1]`
				`rx_st_data[7:0] = rx_st_be[0]`
				This signal is deprecated.

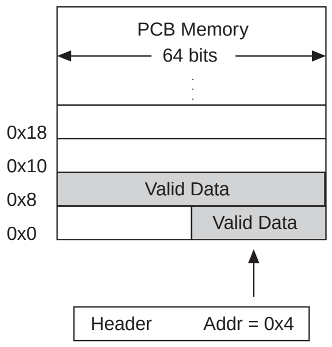
`rx_st_parity`	8, 16, 32	O	component specific	Byte parity is generated when you turn on **Enable byte parity ports on Avalon-ST interface** on the **System Settings** tab of the GUI. Each bit represents odd parity of the associated byte of the `rx_st_data` bus. For example, bit[0] corresponds to `rx_st_data[7:0]`, bit[1] corresponds to `rx_st_data[15:8]`, and so on.
`rxfc_cplbuf_ovf`	1	O	component specific	When asserted indicates that the RX buffer has overflowed.

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

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

To facilitate the interface to 64-bit memories, the Arria V GZ 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. Figure 8–2 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 Figure 8–3 on page 8–9. Qword alignment applies to all types of request TLPs with data, including memory writes, configuration writes, and 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 8–2. Qword Alignment



Refer to Appendix A, Transaction Layer Packet (TLP) Header Formats for the formats of all TLPs.

Table 8–3 shows the byte ordering for header and data packets for Figure 8–3 through Figure 8–12. Refer to Appendix A, Transaction Layer Packet (TLP) Header Formats for a layout of each byte of the TLP headers.

Table 8–3. 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*>

Figure 8–3 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]`.

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 8–3. 64-Bit Avalon-ST rx_st_data<*n*> Cycle Definition for 3-Dword Header TLPs with Non-Qword Aligned Address

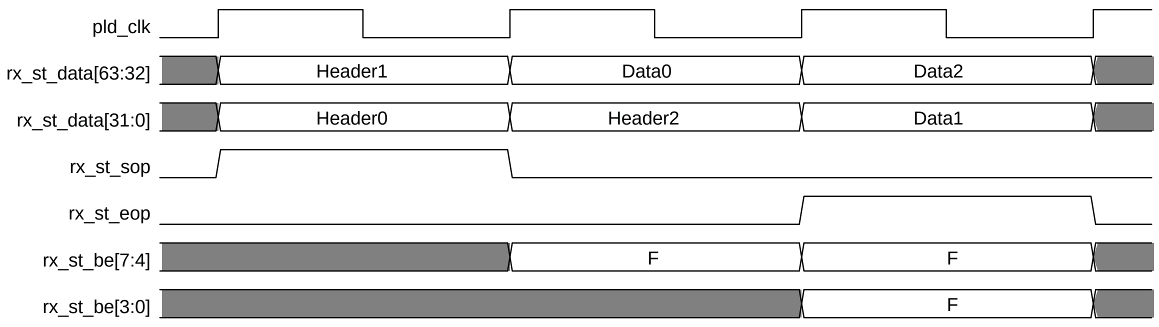
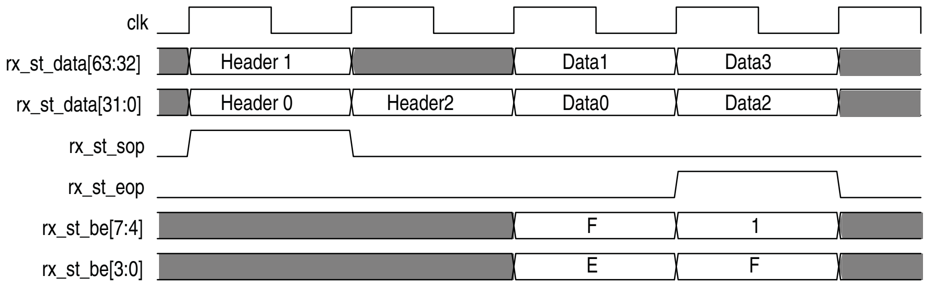


Figure 8–4 illustrates the mapping of Avalon-ST RX packets to PCI Express TLPs for a three dword header with qword aligned addresses. Note that the byte enables indicate the first byte of data is not valid and the last dword of data has a single valid byte.

Figure 8–4. 64-Bit Avalon-ST rx_st_data<n> Cycle Definition for 3-Dword Header TLPs with Qword Aligned Address (1)



Note to Figure 8–4:

(1) `rx_st_be[7:4]` corresponds to `rx_st_data[63:32]`. `rx_st_be[3:0]` corresponds to `rx_st_data[31:0]`

Figure 8–5 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 8–5. 64-Bit Avalon-ST rx_st_data<*n*> Cycle Definitions for 4-Dword Header TLPs with Qword Aligned Addresses

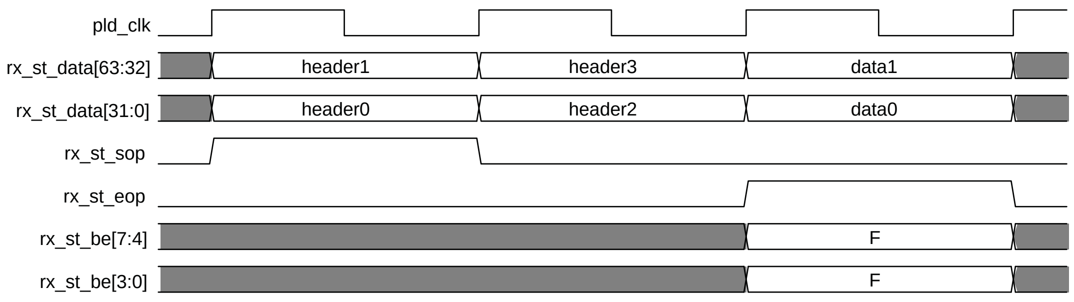
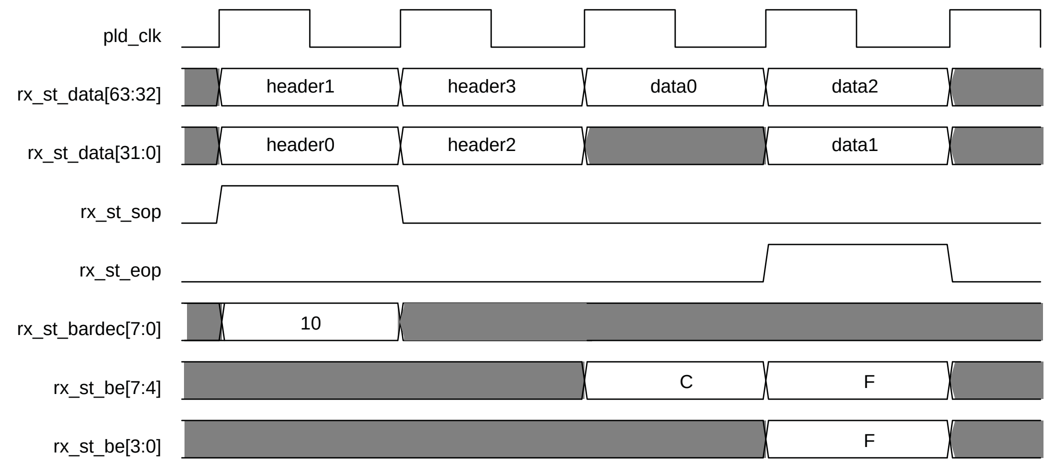


Figure 8–6 shows the mapping of Avalon-ST RX packet to PCI Express TLPs for TLPs for a four dword header with non-qword addresses with a 64-bit bus. Note that the address of the first dword is 0x4. The address of the first enabled byte is 0xC. This example shows one valid word in the first dword, as indicated by the `rx_st_be` signal.

Figure 8–6. 64-Bit Avalon-ST rx_st_data<*n*> Cycle Definitions for 4-Dword Header TLPs with Non-Qword Addresses (1)



Note to Figure 8–6:

(1) `rx_st_be[7:4]` corresponds to `rx_st_data[63:32]`. `rx_st_be[3:0]` corresponds to `rx_st_data[31:0]`.

Figure 8–7 illustrates the timing of the RX interface when the Application Layer backpressures the Arria V GZ 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 8–7. 64-Bit Application Layer Backpressures Transaction Layer

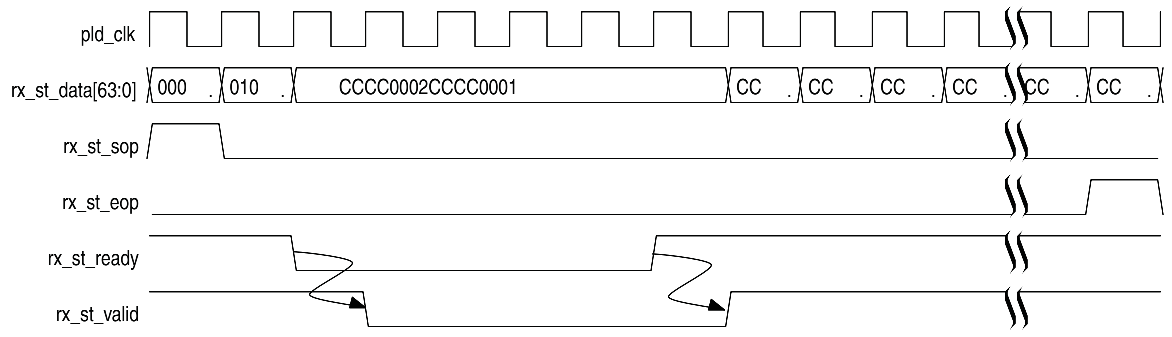
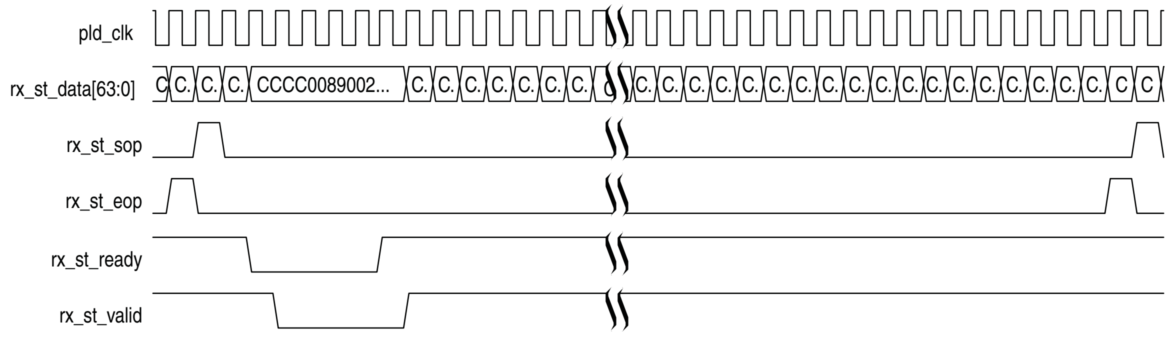


Figure 8–8 illustrates back-to-back transmission on the 64-bit Avalon-ST RX interface with no idle cycles between the assertion of `rx_st_eop` and `rx_st_sop`.

Figure 8–8. 64-Bit Avalon-ST Interface Back-to-Back Transmission



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

Figure 8–9 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 8–9. 128-Bit Avalon-ST rx_st_data<*n*> Cycle Definition for 3-Dword Header TLPs with Qword Aligned Addresses

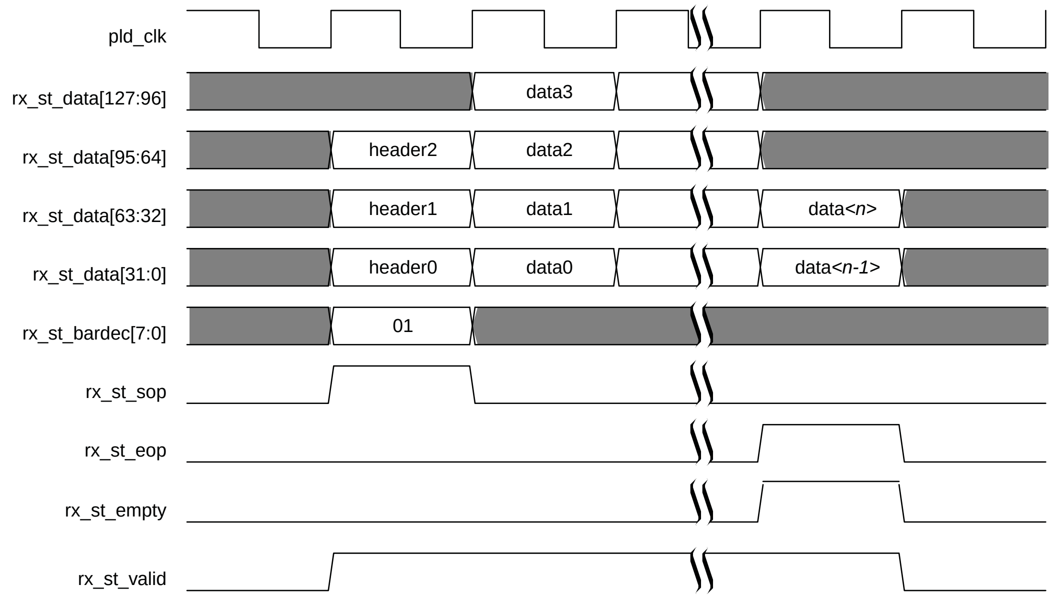


Figure 8–10 shows the mapping of 128-bit Avalon-ST RX packets to PCI Express TLPs for TLPs with a 3 dword header and non-qword aligned addresses. In this case, bits[127:96] represent Data0 because address[2] in the TLP header is set. The assertion of `rx_st_empty` in a `rx_st_eop` cycle indicates valid data on the lower 64 bits of `rx_st_data`.

Figure 8–10. 128-Bit Avalon-ST rx_st_data<*n*> Cycle Definition for 3-Dword Header TLPs with non-Qword Aligned Addresses

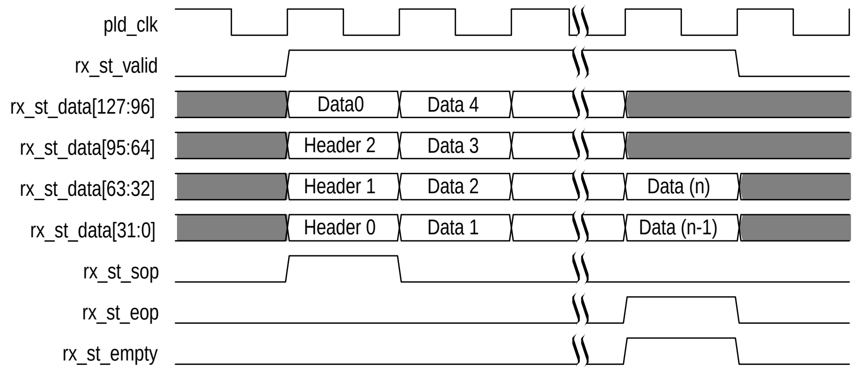


Figure 8–11 shows the mapping of 128-bit Avalon-ST RX packets to PCI Express TLPs for a four dword header with non-qword aligned addresses. In this example, `rx_st_empty` is low because the data is valid for all 128 bits in the `rx_st_eop` cycle.

Figure 8–11. 128-Bit Avalon-ST rx_st_data Cycle Definition for 4-Dword Header TLPs with non-Qword Aligned Addresses

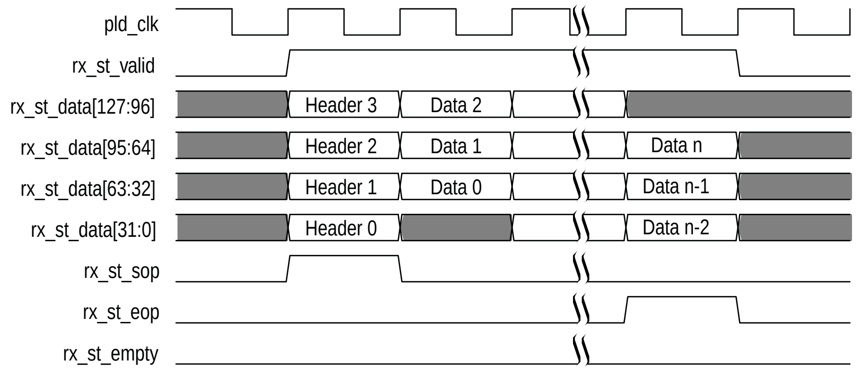


Figure 8–12 shows the mapping of 128-bit Avalon-ST RX packets to PCI Express TLPs for a four dword header with qword aligned addresses. In this example, `rx_st_empty` is low because data is valid for all 128-bits in the `rx_st_eop` cycle.

Figure 8–12. 128-Bit Avalon-ST rx_st_data Cycle Definition for 4-Dword Header TLPs with Qword Aligned Addresses

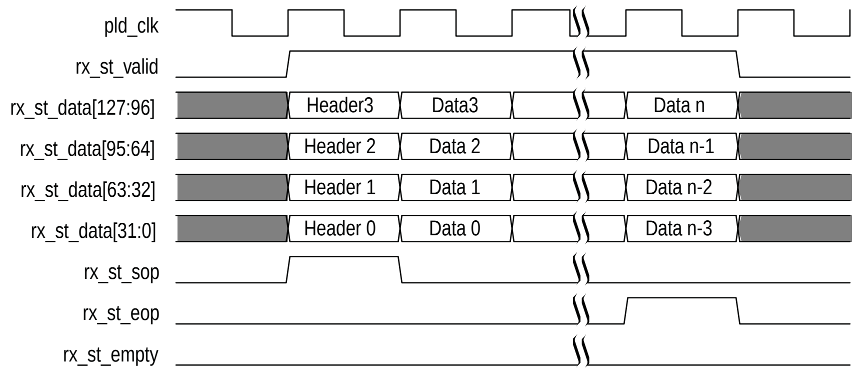


Figure 8–13 illustrates the timing of the RX interface when the Application Layer backpressures the Hard IP by deasserting `rx_st_ready`. The `rx_st_valid` signal deasserts within three cycles after `rx_st_ready` is deasserted. In this example, `rx_st_valid` is deasserted in the next cycle. `rx_st_data` is held until the Application Layer is able to accept it.

Figure 8–13. 128-Bit Application Layer Backpressures Hard IP Transaction Layer for RX Transactions

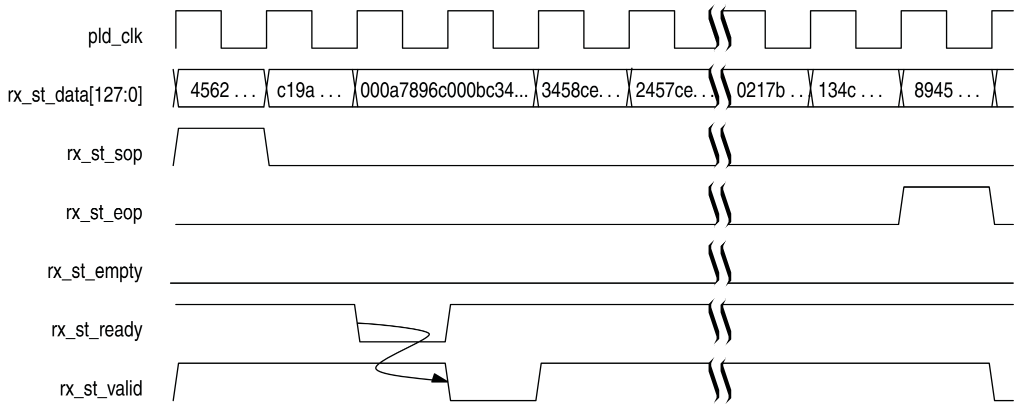


Figure 8–14 illustrates back-to-back transmission on the 128-bit Avalon-ST RX interface with no idle cycles between the assertion of `rx_st_eop` and `rx_st_sop`.

Figure 8–14. 128-Bit Avalon-ST Interface Back-to-Back Transmission

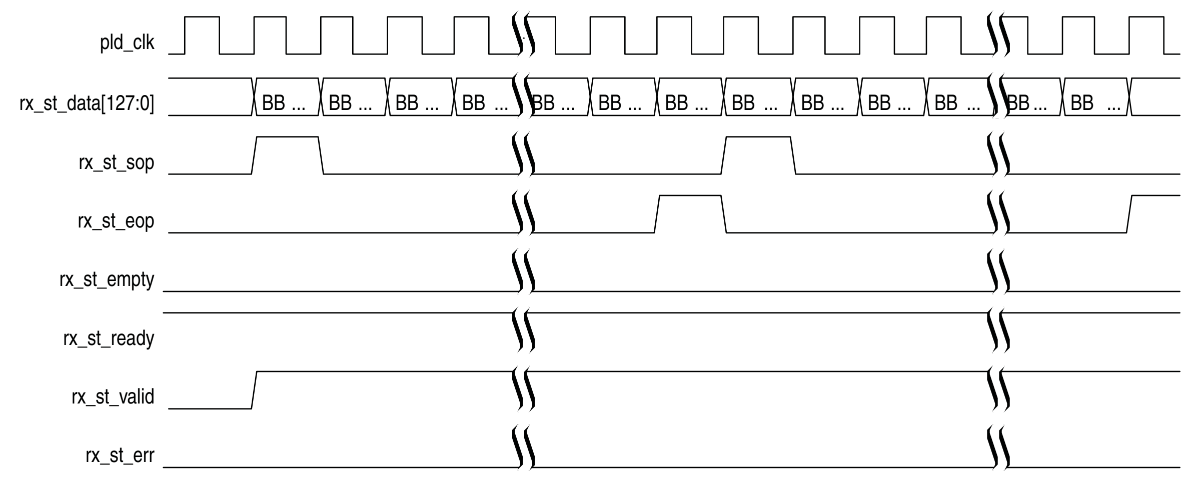
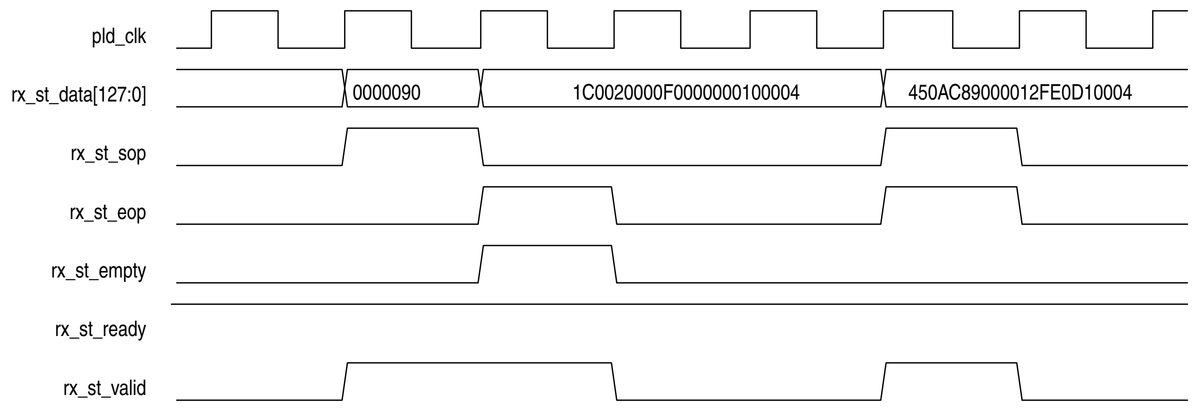


Figure 8–15 illustrates a two-cycle packet with valid data in the lower qword (`rx_st_data[63:0]`) and a one-cycle packet where the `rx_st_sop` and `rx_st_eop` occur in the same cycle.

Figure 8–15. 128-Bit Packet Examples of rx_st_empty and Single-Cycle Packet



For a complete description of the TLP packet header formats, refer to Appendix A, Transaction Layer Packet (TLP) Header Formats.

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 packer 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. However, the Application Layer must still track Completion Credits to avoid RX buffer overflow. To track Completion Credits, use the following signals to monitor the completion space available and to ensure enough space is available before transmitting Non-Posted requests.

- `ko_cpl_spc_header`
- `ko_cpl_spc_data`

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

Figure 8–16 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 8–16. Location of Headers and Data for Avalon-ST 256-Bit Interface

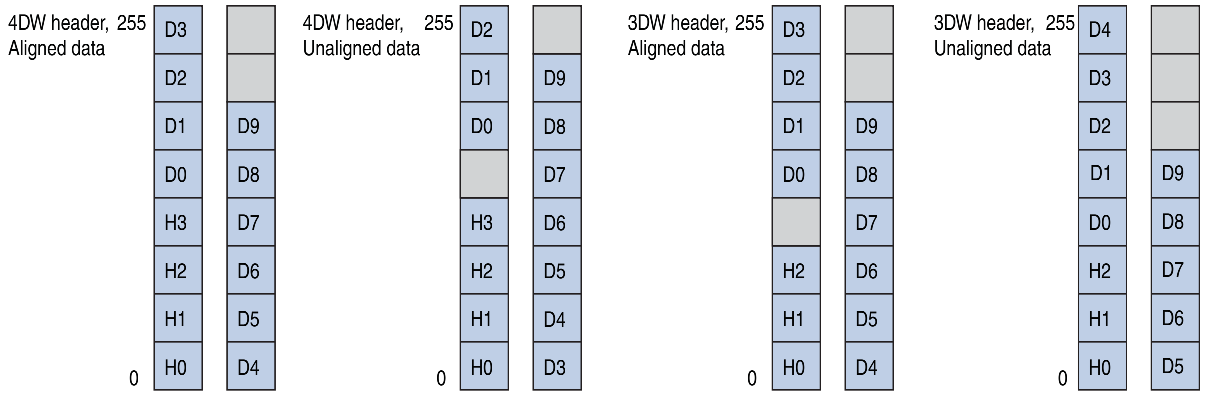
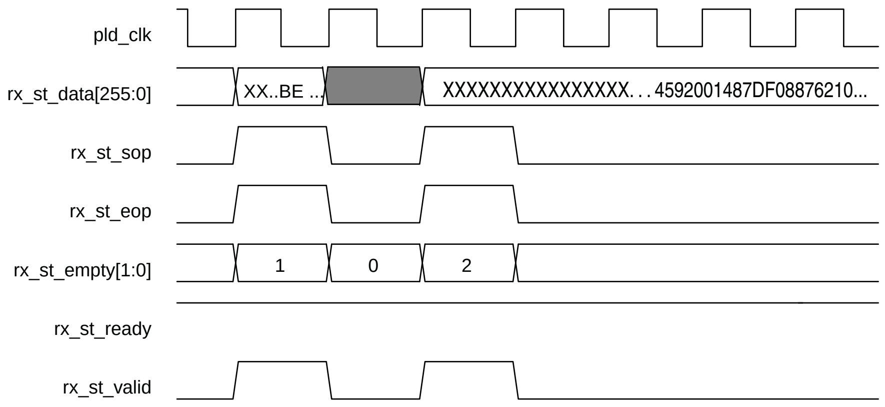


Figure 8–17 illustrates two single-cycle 256-bit packets. The first packet has two empty qword, `rx_st_data[191:0]` is valid. The second packet has two empty dwords; `rx_st_data[127:0]` is valid.

Figure 8–17. 256-Bit Avalon-ST RX Packets Use of rx_st_empty and Single-Cycle Packets

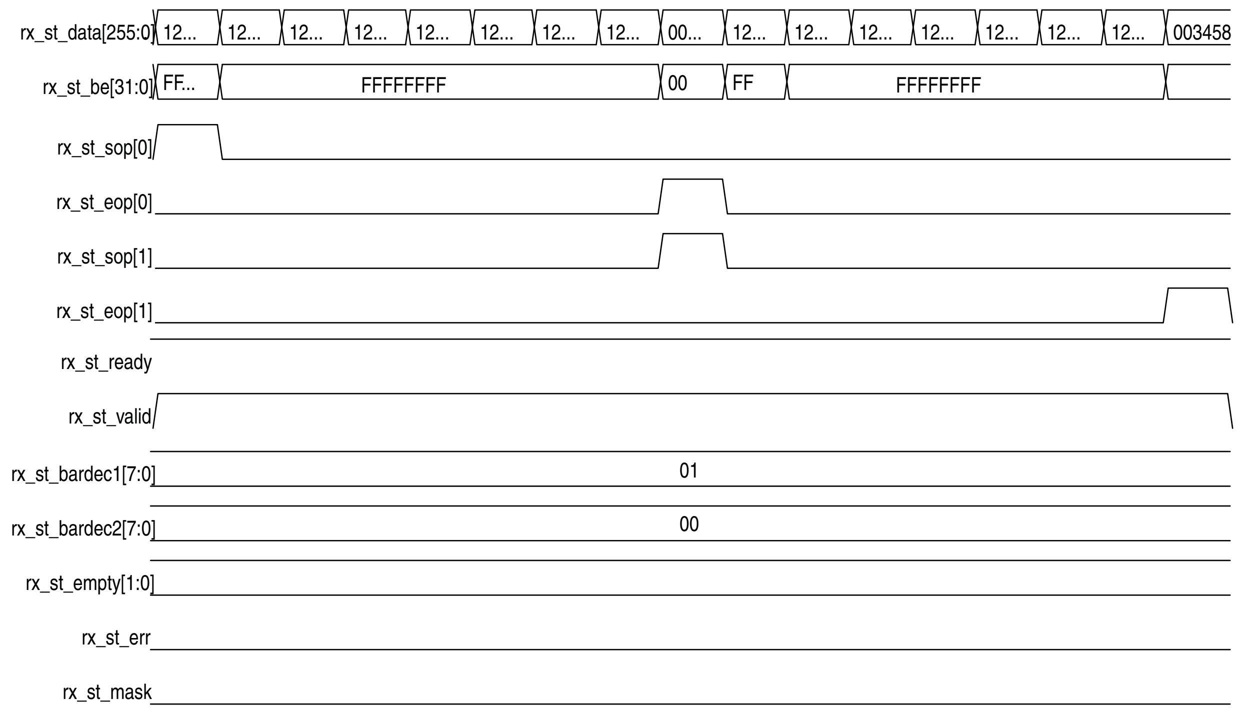


Multiple Packets per Cycle (256-Bit Interface Only)

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 when a TLP ends in the upper 128-bits of the Avalon-ST interface because a new TLP can start in the lower 128-bit Avalon-ST interface. This mode adds complexity to the Application Layer user decode logic. However, it could result in higher throughput.

Figure 8–18 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.

Figure 8–18. 256-Bit Avalon-ST RX Interface with Multiple Packets Per Cycle



Avalon-ST TX Interface

Table 8–4 describes the signals that comprise the Avalon-ST TX Datapath. The TX data signal can be 64, 128, or 256 bits.

Table 8–4. 64-, 128-, or 256-Bit Avalon-ST TX Datapath (Part 1 of 4)

Signal	Width	Dir	Avalon-ST Type	Description
`tx_st_data`	64, 128, 256	I	`data`	Data for transmission. Transmit data bus. Refer to Figure 8–19 through Figure 8–18 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.
`tx_st_sop`	1, 2	I	`start of packet`	Indicates first cycle of a TLP when asserted together with `tx_st_valid`. 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`	1, 2	I	`end of packet`	Indicates last cycle of a TLP when asserted together with `tx_st_valid`. 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` *(1)*	1	O	`ready`	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 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), Altera 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.

Table 8–4. 64-, 128-, or 256-Bit Avalon-ST TX Datapath (Part 2 of 4)

Signal	Width	Dir	Avalon-ST Type	Description
`tx_st_valid` *(1)*	1	I	`valid`	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. Refer to Figure 8–22 on page 8–22 for 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, Altera 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]`	2	I	`empty`	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 data `tx_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 data `tx_st_empty` = 1, `tx_st_data[191:0]` contains valid data `tx_st_empty` = 2, `tx_st_data[127:0]` contains valid data `tx_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

Table 8–4. 64-, 128-, or 256-Bit Avalon-ST TX Datapath (Part 3 of 4)

Signal	Width	Dir	Avalon-ST Type	Description
`tx_st_empty[1:0]` `(continued)`	2	I	`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`	1	I	`error`	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 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 Figure 8–25 on page 8–23 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`	8, 16, 32	O	`component specific`	Byte parity is generated when you turn on **Enable byte parity ports on Avalon-ST interface** on the **System Settings** tab of the GUI. 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.
Component Specific Signals				
`tx_cred_datafccp`	12	O	`component specific`	Data credit limit for the received FC completions. Each credit is 16 bytes.
`tx_cred_datafcnp`	12	O	`component specific`	Data credit limit for the non-posted requests. Each credit is 16 bytes.
`tx_cred_datafcp`	12	O	`component specific`	Data credit limit for the FC posted writes. Each credit is 16 bytes.
`tx_cred_fchipcons`	6	O	`component specific`	Asserted for 1 cycle each time the Hard IP consumes a credit. 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.

Table 8–4. 64-, 128-, or 256-Bit Avalon-ST TX Datapath (Part 4 of 4)

Signal	Width	Dir	Avalon-ST Type	Description
`tx_cred_fc_infinite`	6	O	`component specific`	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
`tx_cred_hdrfccp`	8	O	`component specific`	Header credit limit for the FC completions. Each credit is 20 bytes.
`tx_cred_hdrfcnp`	8	O	`component specific`	Header limit for the non-posted requests. Each credit is 20 bytes.
`tx_cred_hdrfcp`	8	O	`component specific`	Header credit limit for the FC posted writes. Each credit is 20 bytes.
`ko_cpl_spc_header`	8	O	`component specific`	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`	12	O	`component specific`	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.

Note to Table 8–4:

(1) To be Avalon-ST compliant, your Application Layer must have a `readyLatency` of 1 or 2 cycles.

Avalon-ST Packets to PCI Express TLPs

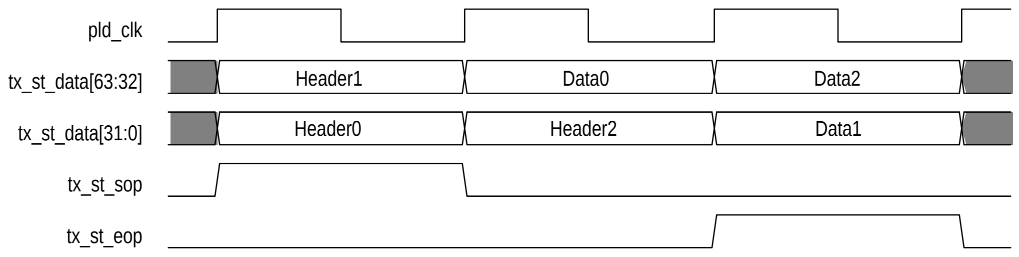
The following figures illustrate the mappings between Avalon-ST packets and PCI Express TLPs. These mappings apply to all types of TLPs, including posted, non-posted, and completion TLPs. Message TLPs use the mappings shown for four dword headers. TLP data is always address-aligned on the Avalon-ST interface 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 Appendix A, Transaction Layer Packet (TLP) Header Formats and *Section 2.2.1 Common Packet Header Fields* in the *PCI Express Base Specification 2.1*.

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

Figure 8–19 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 8–2 on page 8–8 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`.

Figure 8–19. 64-Bit Avalon-ST tx_st_data Cycle Definition for 3-Dword Header TLP with Non-Qword Aligned Address

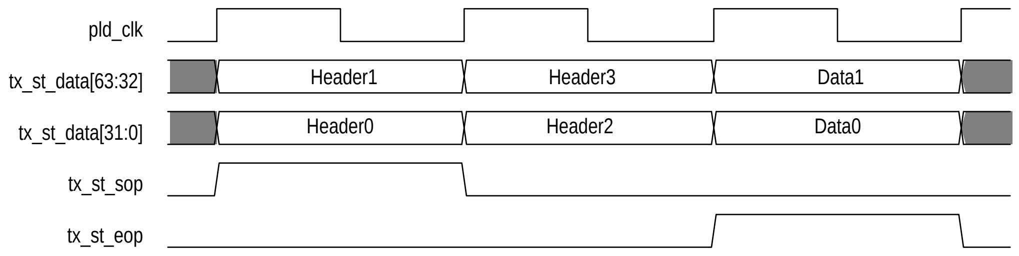


Notes to Figure 8–19:

(1) Header0 ={pcie_hdr_byte0, pcie_hdr _byte1, pcie_hdr _byte2, pcie_hdr _byte3}
(2) Header1 = {pcie_hdr_byte4, pcie_hdr _byte5, header pcie_hdr byte6, pcie_hdr _byte7}
(3) Header2 = {pcie_hdr _byte8, pcie_hdr _byte9, pcie_hdr _byte10, pcie_hdr _byte11}
(4) Data0 = {pcie_data_byte3, pcie_data_byte2, pcie_data_byte1, pcie_data_byte0}
(5) Data1 = {pcie_data_byte7, pcie_data_byte6, pcie_data_byte5, pcie_data_byte4}
(6) Data2 = {pcie_data_byte11, pcie_data_byte10, pcie_data_byte9, pcie_data_byte8}

Figure 8–20 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 8–20. 64-Bit Avalon-ST tx_st_data Cycle Definition for 4-Dword TLP with Qword Aligned Address



Notes to Figure 8–20:

(1) Header0 = {pcie_hdr_byte0, pcie_hdr _byte1, pcie_hdr _byte2, pcie_hdr _byte3}
(2) Header1 = {pcie_hdr _byte4, pcie_hdr _byte5, pcie_hdr byte6, pcie_hdr _byte7}
(3) Header2 = {pcie_hdr _byte8, pcie_hdr _byte9, pcie_hdr _byte10, pcie_hdr _byte11}
(4) Header3 = pcie_hdr _byte12, pcie_hdr _byte13, header_byte14, pcie_hdr _byte15}, 4 dword header only
(5) Data0 = {pcie_data_byte3, pcie_data_byte2, pcie_data_byte1, pcie_data_byte0}
(6) Data1 = {pcie_data_byte7, pcie_data_byte6, pcie_data_byte5, pcie_data_byte4}

Figure 8–21 illustrates the mapping between Avalon-ST TX packets and PCI Express TLPs for four dword header with non-qword aligned addresses with a 64-bit bus.

Figure 8–21. 64-Bit Avalon-ST tx_st_data Cycle Definition for TLP 4-Dword Header with Non-Qword Aligned Address

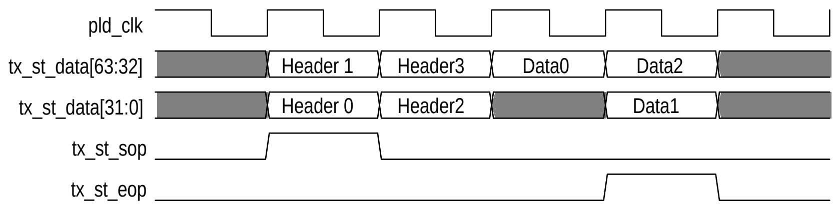


Figure 8–22 illustrates the timing of the TX interface when the Arria V GZ Hard IP for PCI Express backpressures the Application Layer by deasserting `tx_st_ready`. Because the `readyLatency` is two cycles, the Application Layer deasserts `tx_st_valid` after two cycles and holds `tx_st_data` until two cycles after `tx_st_ready` is asserted.

Figure 8–22. 64-Bit Transaction Layer Backpressures the Application Layer

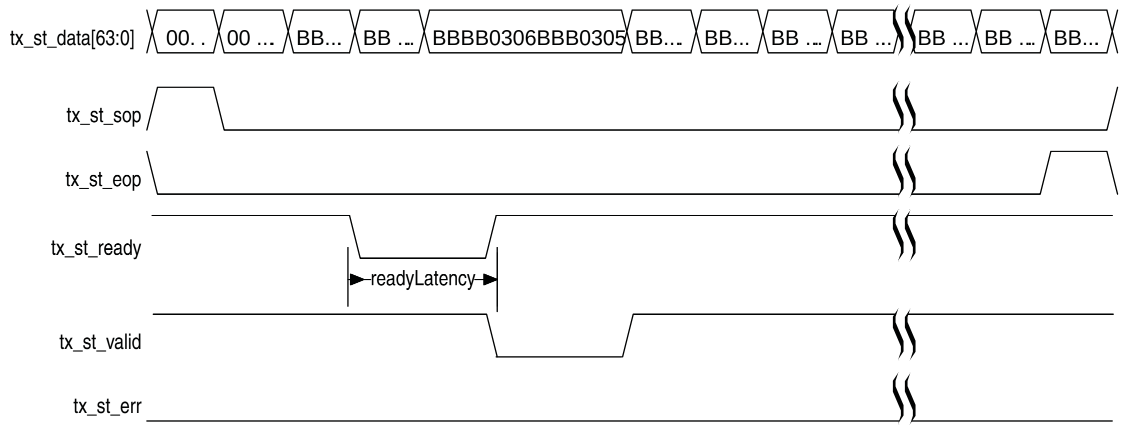
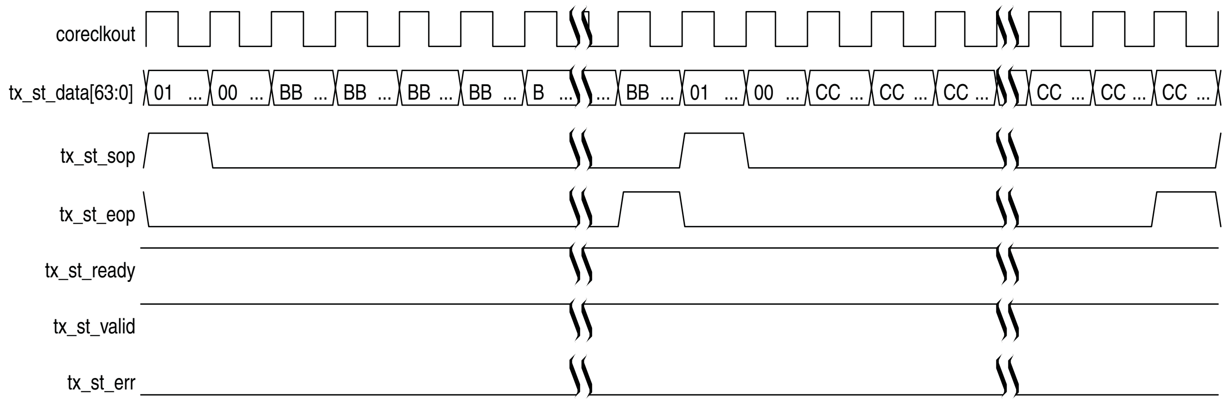


Figure 8–23 illustrates back-to-back transmission of 64-bit packets with no idle cycles between the assertion of `tx_st_eop` and `tx_st_sop`.

Figure 8–23. 64-Bit Back-to-Back Transmission on the TX Interface



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

Figure 8–24 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 8–24. 128-Bit Avalon-ST tx_st_data Cycle Definition for 3-Dword Header TLP with Qword Aligned Address

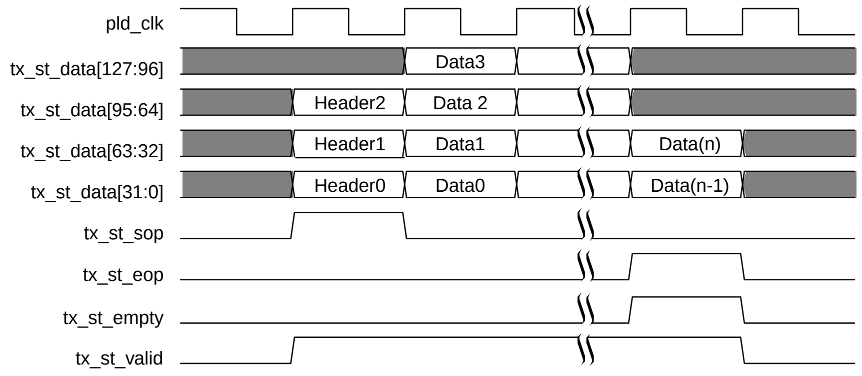


Figure 8–25 shows the mapping of 128-bit Avalon-ST TX packets to PCI Express TLPs for a 3 dword header with non-qword aligned addresses. It also shows `tx_st_err` assertion.

Figure 8–25. 128-Bit Avalon-ST tx_st_data Cycle Definition for 3-Dword Header TLP with non-Qword Aligned Address

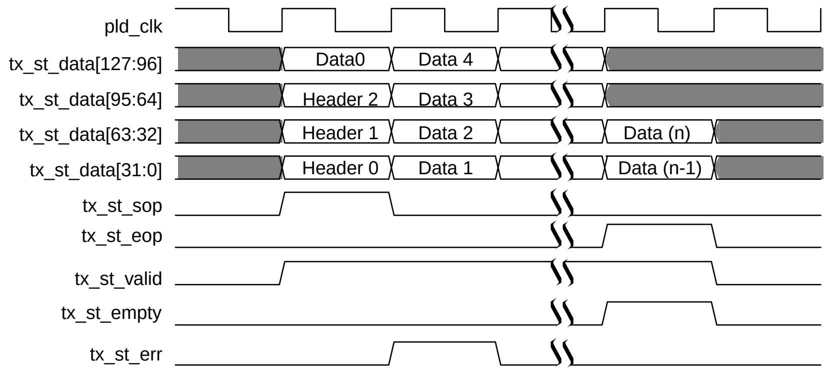


Figure 8–26 shows the mapping of 128-bit Avalon-ST TX packets to PCI Express TLPs for a four dword header TLP with qword aligned data.

Figure 8–26. 128-Bit Avalon-ST tx_st_data Cycle Definition for 4-Dword Header TLP with Qword Aligned Address

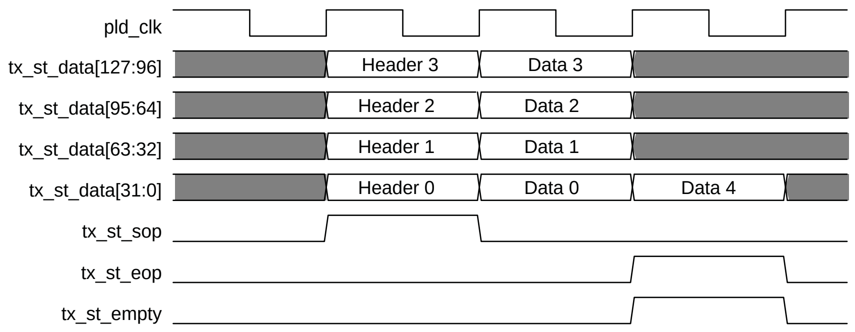


Figure 8–27 shows the mapping of 128-bit Avalon-ST TX packets to PCI Express TLPs for a four dword header TLP with non-qword aligned addresses. In this example, `tx_st_empty` is low because the data ends in the upper 64 bits of `tx_st_data`.

Figure 8–27. 128-Bit Avalon-ST tx_st_data Cycle Definition for 4-Dword Header TLP with non-Qword Aligned Address

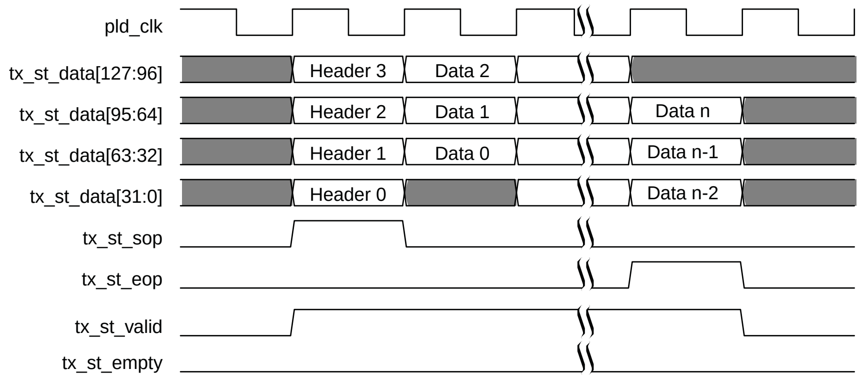


Figure 8–28 illustrates back-to-back transmission of 128-bit packets with idle dead cycles between the assertion of `tx_st_eop` and `tx_st_sop`.

Figure 8–28. 128-Bit Back-to-Back Transmission on the Avalon-ST TX Interface

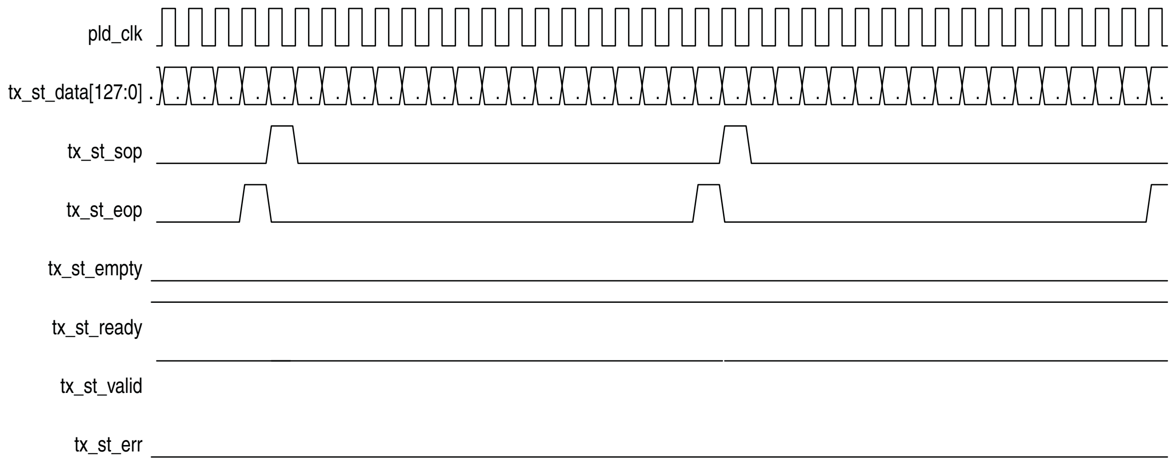
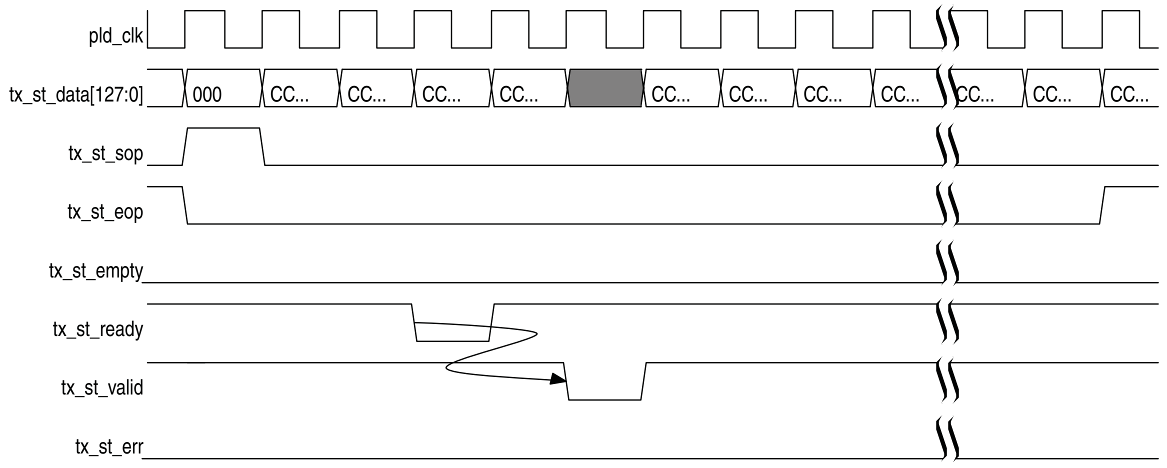


Figure 8–29 illustrates the timing of the TX interface when the Arria V GZ Hard IP for PCI Express backpressures the Application Layer by deasserting `tx_st_ready`. Because the `readyLatency` is two cycles, the Application Layer deasserts `tx_st_valid` after two cycles and holds `tx_st_data` until two cycles after `tx_st_ready` is reasserted

Figure 8–29. 128-Bit Hard IP Backpressures the Application Layer for TX Transactions



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.

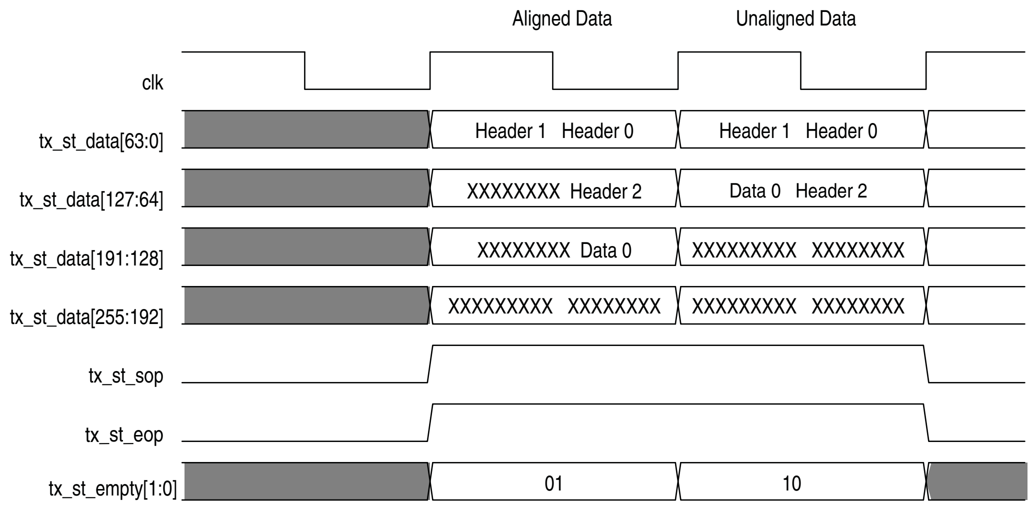
Single Packet Per Cycle

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 8–30 illustrates the layout of header and data for a three dword header on a 256-bit bus with aligned and unaligned data.

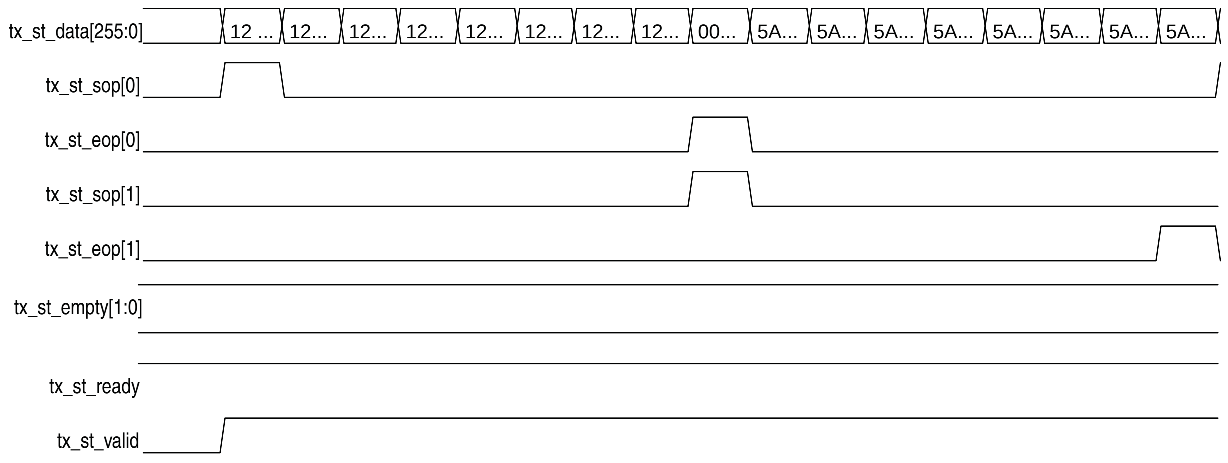
Figure 8–30. 256-Bit Avalon-ST tx_st_data Cycle Definition for 3-Dword Header TLP with Qword Addresses



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. Figure 8–31 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.

Figure 8–31. 256-Bit Avalon-ST TX Interface with Multiple Packets Per Cycle



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.

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. Depending on the address alignment, Figure 8–5 on page 8–9 through Figure 8–12 on page 8–13 illustrate the position of the ECRC data for RX data. Figure 8–19 on page 8–21 through Figure 8–27 on page 8–24 illustrate the position of ECRC data for TX data. For packets with no payload data, the ECRC position corresponds to the position of Data0 in these figures.

Clock Signals

Table 8–5 describes the clock signals that comprise the clock interface.

Table 8–5. Clock Signals Hard IP Implementation (1)

Signal	I/O	Description
`refclk`	I	Reference clock for the Arria V GZ Hard IP for PCI Express. It must have the frequency specified under the **System Settings** heading in the parameter editor. If your design meets the following criteria: ■ It enables CvP ■ Includes an additional transceiver PHY connected to the same Transceiver Reconfiguration Controller then you must connect `refclk` to the `mgmt_clk_clk` signal of the Transceiver Reconfiguration Controller and the additional transceiver PHY. In addition, if your design includes more than one Transceiver Reconfiguration Controller on the same side of the FPGA, they all must share the `mgmt_clk_clk` signal.
`pld_clk`	I	Clocks the Application Layer. You must drive this clock from `coreclkout_hip`. This signal is not used for the Avalon-MM Hard IP for PCI Express IP Core.
`coreclkout`	O	This is a fixed frequency clock used by the Data Link and Transaction Layers. To meet PCI Express link bandwidth constraints, this clock has minimum frequency requirements as listed in Table 10–3 on page 10–6.

Note to Table 8–5:

(1) Figure 10–5 on page 10–5 illustrates these clock signals.

Refer to Chapter 10, Reset and Clocks for a complete description of the clock interface.

Reset Signals, Status, and Link Training Signals

Table 8–6 describes the reset signals. Refer to Chapter 10, Reset and Clocks for more information about the reset sequence and a block diagram of the reset logic. The following table describes reset signals used in all IP Cores for PCI Express.

Table 8–6. Reset Signals (Part 1 of 2)

Signal	I/O	Description
`npor`	I	Active low reset signal. In the Altera 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. Asynchronous. Resets the entire Arria V GZ Hard IP for PCI Express IP Core and transceiver. In systems that use the hard reset controller, 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 hard and soft reset controllers, refer to Reset.

Table 8–6. Reset Signals (Part 2 of 2)

Signal	I/O	Description
`reset_status`	O	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.
`pin_perst`	I	Active low reset from the PCIe reset pin of the device. This reset signal is an input to the embedded reset controller as illustrated in Figure 10–1 on page 10–2. It resets the datapath and control registers. This signal is required for Configuration via Protocol (CvP). For more information about CvP refer to "Configuration via Protocol (CvP)" on page 12–1. Arria V GZ devices have up to 4 instances of the Hard IP for PCI Express. Each instance has its own `pin_perst` signal. Every Arria V GZ device has 4 `nPERST` pins, even devices with fewer than 4 instances of the Hard IP for PCI Express. *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 V GZ device, Altera recommends that you use the bottom left Hard IP first. This is the only location that supports CvP over a PCIe link. Refer to the appropriate Arria V GZ device pinout for correct pin assignment for more detailed information about these pins. The *PCI Express Card Electromechanical Specification 2.0* specifies this pin to require 3.3 V. You can drive this 3.3V signal to the `nPERST*` even if the V_{CCIO} of the bank is not 3.3V if the following 2 conditions are met: ■ The input signal meets the V_{IH} and V_{IL} specification for LVTTL. ■ The input signal meets the overshoot specification for 100°C operation as specified by the "Maximum Allowed Overshoot and Undershoot Voltage" section in volume 3 of the *Arria V Device Handbook*. Refer to Figure 8–32 on page 8–31 for a timing diagram illustrating the use of this signal.

The following table describes additional signals related to the reset function for the Arria V GZ Hard IP for PCI Express IP Core that uses the Avalon-ST interface, including the `ltsssm_state[4:0]` bus that indicates the current link training state.

Table 8–7. Status and Link Training Signals (Part 1 of 3)

Signal	I/O	Description
`serdes_pll_locked`	O	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`	I	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`	O	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.

Table 8–7. Status and Link Training Signals (Part 2 of 3)

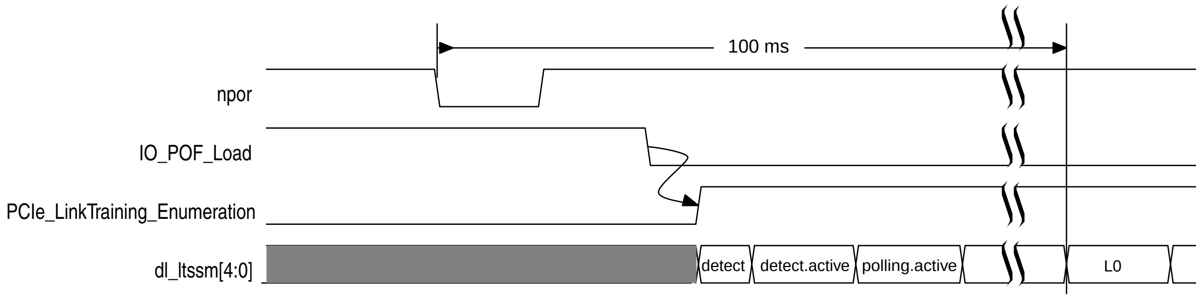
Signal	I/O	Description
`dlup`	O	When asserted, indicates that the Hard IP block is in the Data Link Control and Management State Machine (DLCMSM) DL_Up state.
`dlup_exit`	O	This signal is asserted low for one `pld_clk` cycle when the IP core exits the DLCMSM DL_Up state, indicating that the Data Link 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`	O	Asserted every 128 ns to create a time base aligned activity.
`ev1us`	O	Asserted every 1 µs to create a time base aligned activity.
`hotrst_exit`	O	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`	O	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.
`currentspeed[1:0]`	O	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]`	O	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 ■ 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

Table 8–7. Status and Link Training Signals (Part 3 of 3)

Signal	I/O	Description
	O	■ 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

Figure 8–32 illustrates the timing relationship between `npor` and the LTSSM L0 state.

Figure 8–32. 100 ms Requirement (1)



Note to Figure 8–32:

(1) The ability of Gen2- and Gen3-capable designs to begin link initialization and ultimately to reach L0 before the FPGA is configured is pending device characterization.

Error Signals

Table 8–8 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 V GZ Hard IP for PCI Express recovers without any loss of information. No Application Layer intervention is required. In the case of uncorrectable ECC error, Altera recommend that you reset the core.

Table 8–8. Error Signals for Hard IP Implementation (1)

Signal	I/O	Description
`derr_cor_ext_rcv`	O	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. (2)
`derr_rpl`	O	Indicates an uncorrectable error in the retry buffer. This signal is for debug only. (2)
`derr_cor_ext_rpl`	O	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. (2)
`rxfc_cplbuf_ovf`	O	The optional status signal is available when you turn on **Track Receive Completion Buffer Overflow** in the GUI. Because the RX buffer completion space advertises infinite credits for Endpoints, you can use this status bit as an additional check to complement the soft logic that tracks space in the completion buffer.

Note to Table 8–8:

(1) The Avalon-ST `rx_st_err` described in Table 8–2 on page 8–4 indicates an uncorrectable error in the RX buffer.

(2) 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.

Interrupts for Endpoints

Table 8–9 describes the IP core's interrupt signals for Endpoints. Refer to Chapter 13, Interrupts for detailed information about all interrupt mechanisms.

Table 8–9. Interrupt Signals for Endpoints (Part 1 of 2)

Signal	I/O	Description
`app_msi_req`	I	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`	O	Application Layer MSI acknowledge. This signal acknowledges the Application Layer's request for an MSI interrupt.
`app_msi_tc[2:0]`	I	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]`	I	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. Refer to Table 8–18 on page 8–41 for more information.

Table 8–9. Interrupt Signals for Endpoints (Part 2 of 2)

Signal	I/O	Description
`app_int_sts`	I	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`	O	This signal is the acknowledge for `app_int_sts`. This signal is asserted for at least one cycle either when the `Assert_INTA` message TLP has been transmitted in response to the assertion of the `app_int_sts` signal or when the `Deassert_INTA` message TLP has been transmitted in response to the deassertion of the `app_int_sts` signal. Refer to Figure 13–5 on page 13–4 and Figure 13–6 on page 13–5 for timing information.

Interrupts for Endpoints when Multiple MSI/MSI-X Support Is Enabled

Table 8–10 describes the IP core's exported interrupt signals when you turn on **Enable multiple MSI/MSI-X support** under the **Avalon-MM System Settings** banner in the GUI. Refer to Chapter 13, Interrupts for detailed information about all interrupt mechanisms.

Table 8–10. Exported Interrupt Signals for Endpoints when Multiple MSIMSI-X Support is Enabled

Signal	I/O	Description
`msi_intf[81:0]`	O	This bus provides the following MSI address, data, and enabled signals: ■ `msi_intf[81]`: Master enable ■ `msi_intf[80}`: MSI enable ■ `msi_intf[79:64]`: MSI data ■ `msi_intf[63:0]`: MSI address
`msix_intf[79:0]`	O	This bus provides the following MSI address, data, and enabled signals: ■ `msix_intf[79:64]`: MSI-X control register ■ `msix_intf[63:32}`: MSI-X PBA Offset/BIR ■ `msix_intf[31:0]`: MSI-X PBA Table Offset/BIR
`Intx_inf[1]`	I	IntxReq_i. Legacy interrupt request.
`Intx_inf[0]`	O	IntxAck_o. Legacy interrupt acknowledge.

Interrupts for Root Ports

Table 8–11 describes the signals available to a Root Port to handle interrupts.

Table 8–11. Interrupt Signals for Root Ports

Signal	I/O	Description
`int_status[3:0]`	O	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`	O	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` register and the `Device Control` register. 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 2.0* or *3.0* in the `Root Control` register.

Completion Side Band Signals

Table 8–12 describes the signals that comprise the completion side band signals for the Avalon-ST interface. The Arria V GZ 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 Chapter 15, 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, Rev. 2.1*.

Table 8–12. Completion Signals for the Avalon-ST Interface (Part 1 of 2)

Signal	I/O	Description
`cpl_err[6:0]`	I	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.

Table 8–12. Completion Signals for the Avalon-ST Interface (Part 2 of 2)

Signal	I/O	Description
		■ `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, Rev. 2.1.* ■ `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” on page 15–3. ■ `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” on page 15–3.
	I	■ `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” on page 15–3. ■ `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]`: ■ 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]` Refer to the “LMI Signals” on page 8–47 for more information about LMI signalling.
`cpl_pending`	I	Completion pending. The Application Layer must assert this signal when a master block is waiting for completion, for example, when a transaction is pending.

Transaction Layer Configuration Space Signals

Table 8–13 describes the Transaction Layer Configuration Space signals.

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

Table 8–13. Configuration Space Signals (Hard IP Implementation)

Signal	Dir	Description
`tl_cfg_add[3:0]`	O	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 Table 8–15 on page 8–38. The index increments on every `pld_clk`.
`tl_cfg_ctl[31:0]`	O	The `tl_cfg_ctl` signal is multiplexed and contains the contents of the Configuration Space registers. The information presented on this bus depends on the `tl_cfg_add` index according to Table 8–15 on page 8–38.
`tl_cfg_sts[52:0]`	O	Configuration status bits. This information updates every `pld_clk` cycle. Refer to Table 8–14 for a detailed description of the status bits.
`hpg_ctrler[4:0]`	I	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:
	I	■ [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.
	I	■ [1]: Presence detect. This signal should be asserted when a presence detect circuit detects a presence detect change in the slot.
	I	■ [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.
	I	■ [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.
	I	■ [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 8–14 describes the bits of the bits of the `tl_cfg_sts` bus.

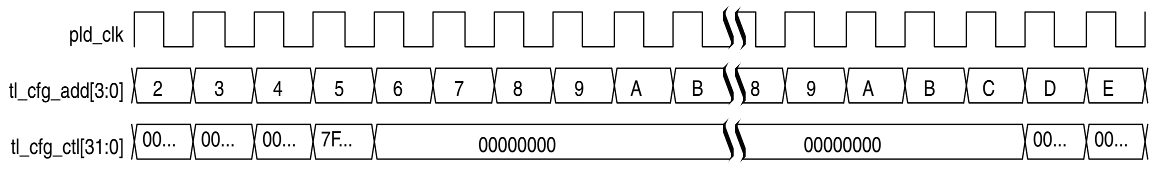
Table 8–14. Mapping Between tl_cfg_sts and Configuration Space Registers

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
[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 ■ 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: signalled 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: signalled target abort
[0]	Secondary Status Register[8]	Master Data Parity Error

Configuration Space Register Access Timing

Figure 8–33 shows typical traffic on the `tl_cfg_ctl` bus. The `tl_cfg_add` index increments on the rising edge of `pld_clk` specifying which Configuration Space register information is being driven onto `tl_cfg_ctl`.

Figure 8–33. tl_cfg_ctl Timing



Configuration Space Register Access

The `tl_cfg_ctl` signal is a multiplexed bus that contains the contents of Configuration Space registers as shown in Table 8–13. Information stored in the Configuration Space is accessed in round robin order where `tl_cfg_add` indicates which register is being accessed. Table 8–15 shows the layout of configuration information that is multiplexed on `tl_cfg_ctl`.

Table 8–15. Multiplexed Configuration Register Information Available on tl_cfg_ctl (1)

Address	31:24	23:16	15:8	7:0
0	cfg_dev_ctrl[15:0] cfg_dev_ctrl[14:12]= Max Read Req Size (2)	cfg_dev_ctrl[7:5]= Max Payload (2)	cfg_dev_ctrl2[15:0]	
1	16'h0000		cfg_slot_ctrl[15:0]	
2	cfg_link_ctrl[15:0]		cfg_link_ctrl2[15:0]	
3	8'h00	cfg_prm_cmd[15:0]		cfg_root_ctrl[7:0]
4	cfg_sec_ctrl[15:0]		cfg_secbus[7:0]	cfg_subbus[7:0]
5	cfg_msi_addr[11:0]	cfg_io_bas[19:0]		
6	cfg_msi_addr[43:32]	cfg_io_lim[19:0]		
7	8h'00	cfg_np_bas[11:0]	cfg_np_lim[11:0]	
8	cfg_pr_bas[31:0]			
9	cfg_msi_addr[31:12]		cfg_pr_bas[43:32]	
A	cfg_pr_lim[31:0]			
B	cfg_msi_addr[63:44]		cfg_pr_lim[43:32]	
C	cfg_pmcsr[31:0]			
D	cfg_msixcsr[15:0]		cfg_msicsr[15:0]	
E	6'h00, tx_ecrcgen[25] (3), rx_ecrccheck[24]	cfg_tcvcmap[23:0]		
F	cfg_msi_data[15:0]		3'b000	cfg_busdev[12:0]

Notes to Table 8–15:

(1) Items in blue are only available for Root Ports.

(2) This field is encoded as specified in Section 7.8.4 of the *PCI Express Base Specification*. (3'b000–3'b101 correspond to 128–4096 bytes).

(3) `rx_ecrccheck` and `tx_ecrcgen` are bit s 24 and 25 of `tl_cfg_ctl`, respectively. (Other bit specifications in this table indicate the bit location within the Configuration Space register.)

Table 8–16 describes the Configuration Space registers referred to in Table 8–13 and Table 8–15.

Table 8–16. Configuration Space Register Descriptions (Part 1 of 3)

Register	Width	Dir	Description	Register Reference
`cfg_dev_ctrl`	16	O	`cfg_devctrl[15:0]` is Device Control for the PCI Express capability structure.	Table 9–7 on page 9–4
`cfg_dev_ctrl2`	16	O	`cfg_dev2ctrl[15:0]` is device control 2 for the PCI Express capability structure.	Table 9–8 on page 9–5
`cfg_slot_ctrl`	16	O	`cfg_slot_ctrl[15:0]`is the Slot Status of the PCI Express capability structure. This register is only available in Root Port mode.	Table 9–7 on page 9–4 Table 9–8 on page 9–5
`cfg_link_ctrl`	16	O	`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 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 LTSSM to the Recovery state. Retraining to a higher data rate is not automatic for the Arria V GZ Hard IP for PCI Express IP Core even if both devices on the link are capable of a higher data rate.	Table 9–7 on page 9–4
`cfg_link_ctrl2`	16	O	`cfg_link_ctrl2[31:16]` is the secondary Link Control register of the PCI Express capability structure for Gen2 operation. When `tl_cfg_addr=2`, `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 is 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.	Table 9–8 on page 9–5 0x0B0 (Gen2, only)
`cfg_prm_cmd`	16	O	Base/Primary Control and Status register for the PCI Configuration Space.	Table 9–2 on page 9–2 0x004 (Type 0) Table 9–3 on page 9–2 0x004 (Type 1)
`cfg_root_ctrl`	8	O	Root control and status register of the PCI-Express capability. This register is only available in Root Port mode.	Table 9–7 on page 9–4 Table 9–8 on page 9–5
`cfg_sec_ctrl`	16	O	Secondary bus Control and Status register of the PCI-Express capability. This register is only available in Root Port mode.	Table 9–3 on page 9–2 0x01C
`cfg_secbus`	8	O	Secondary bus number. Available in Root Port mode.	Table 9–3 on page 9–2 0x018
`cfg_subbus`	8	O	Subordinate bus number. Available in Root Port mode.	Table 9–3 on page 9–2 0x018

Table 8–16. Configuration Space Register Descriptions (Part 2 of 3)

Register	Width	Dir	Description	Register Reference
`cfg_msi_addr`	64	O	`cfg_msi_add[63:32]` is the MSI upper message address. `cfg_msi_add[31:0]` is the MSI message address.	Table 9–4 on page 9–3 0x050
`cfg_io_bas`	20	O	The upper 20 bits of the IO limit registers of the Type1 Configuration Space. This register is only available in Root Port mode.	Table 9–3 on page 9–2 0x01C
`cfg_io_lim`	20	O	The upper 20 bits of the IO limit registers of the Type1 Configuration Space. This register is only available in Root Port mode.	Table 9–8 on page 9–5 0x01C
`cfg_np_bas`	12	O	The upper 12 bits of the memory base register of the Type1 Configuration Space. This register is only available in Root Port mode.	Table 6–2 on page 6–4
`cfg_np_lim`	12	O	The upper 12 bits of the memory limit register of the Type1 Configuration Space. This register is only available in Root Port mode.	Table 6–2 on page 6–4
`cfg_pr_bas`	44	O	The upper 44 bits of the prefetchable base registers of the Type1 Configuration Space. This register is only available in Root Port mode.	Table 9–3 on page 9–2 0x024 and Table 6–2 on page 6–4 **prefetchable memory**
`cfg_pr_lim`	44	O	The upper 44 bits of the prefetchable limit registers of the Type1 Configuration Space. Available in Root Port mode.	Table 9–3 on page 9–2 0x024 and Table 6–2 on page 6–4 **prefetchable memory**
`cfg_pmcsr`	32	O	`cfg_pmcsr[31:16]` is Power Management Control and `cfg_pmcsr[15:0]`is the Power Management Status register.	Table 9–6 on page 9–4 0x07C
`cfg_msixcsr`	16	O	MSI-X message control.	Table 9–5 on page 9–3 0x068
`cfg_msicsr`	16	O	MSI message control. Refer to Table 8–18 for the fields of this register.	Table 9–4 on page 9–3 0x050

Table 8–16. Configuration Space Register Descriptions (Part 3 of 3)

Register	Width	Dir	Description	Register Reference
`cfg_tcvcmap`	24	O	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	O	`cfg_msi_data[15:0]` is message data for MSI.	Table 9–4 on page 9–3 0x050
`cfg_busdev`	13	O	Bus/Device Number captured by or programmed in the Hard IP.	Table A–5 on page A–2 0x08

Table 8–18 shows the layout of the Configuration MSI Control Status register.

Table 8–17. Configuration MSI Control Status Register

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

Table 8–18 describes the use of the various fields of the Configuration MSI Control and Status Register.

Table 8–18. Configuration MSI Control Status Register Field Descriptions (Part 1 of 2)

Bit(s)	Field	Description
[15:9]	`reserved`	—
[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, Rev. 3.0*. 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

Table 8–18. Configuration MSI Control Status Register Field Descriptions (Part 2 of 2)

Bit(s)	Field	Description
[6:4]	`multiple message enable`	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`	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 ■ 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.

Configuration Space Bypass Mode Interface Signals

In Configuration Space Bypass mode, the soft Configuration Space exchanges control and status information with the Arria V GZ Hard IP for PCI Express's Transaction, Data Link and PHY Layers. Refer to "Configuration Space Bypass Mode" on page 7–7 for information about the division of the Configuration Space functionality in bypass mode.

Table 8–19 describes the input signals which are driven from the Application Layer's soft Configuration Space to the Transaction Layer's hard Configuration Space. In Qsys, the configuration space bypass input signals have the prefix *bypass_in_cfgbp_*. In the MegaWizard Plug-In Manager design flow, these signals have the prefix *cfgbp_*. All signals are synchronous to pld_clk.

Table 8–19. Configuration Space Bypass Mode Input Signals (Part 1 of 2)

Signal Name	Dir	Description
`link2csr[12:0]`	I	Bits[12:0] of the `link2csr` register from the soft Configuration Space.
`comclk_reg`	I	Common clock configuration bit of the `Link Control Register` in the soft Configuration Space.
`extsy_reg`	I	Extended synchronization bit of the `Link Control Register` in the soft Configuration Space.
`max_pload[2:0]`	I	`MAX_PAYLOAD_SIZE` field of the `Device Control Register` in the soft Configuration Space.

Table 8–19. Configuration Space Bypass Mode Input Signals (Part 2 of 2)

Signal Name	Dir	Description
`tx_ecrcgen`	I	`ECRC Generation Enable` which is bit 6 of the `Advanced Error Capabilities and Control` Register (0x18) in the soft Configuration Space.
`rx_ecrchk`	I	`ECRC Check Enable` which is Bit 8 of the `Advanced Error Capabilities and Control` Register (0x18) in the soft Configuration Space.
`secbus[7]`	I	MSB of the `Secondary Bus Number Register` in the soft Configuration Space.
`secbus[6:0]`	I	Low-order 7 bits of `Secondary Bus Number Register` in the soft Configuration Space.
`linkcsr_bit0`	I	`Active State Power Management (ASPM) Control` which is Bit 0 of the `Link Control` Register in the soft Configuration Space. ASPM is not supported.
`tx_req_pm`	I	Assert this signal to request that the TX Data Link Layer send a Power Management Data Link Layer Packet of type `tx_typ_pm_pld`. Deasserts when `tx_ack_pm_pld` is asserted.
`tx_typ_pm[2:0]`	I	Lowest 3 bits of the type field for the Power Management Data Link Layer Packet being requested by `tx_req_pm_pld`.
`req_phypm[3:0]`	I	Directs the LTSSM to low power mode. ■ `req_phypm[3]`: L2 request ■ `req_phypm[2]`: L1 request ■ `req_phypm[1]`: L0s request ■ `req_phypm[0]`: exit any low power state to L0
`req_phycfg[3:0]`	I	Configuration Space transition request bus: ■ `req_phycfg[3]`: Link retrain request. Retrain the link by writing 1b’1 in the retrain link bit in the `Link Control` `Register`. ■ `req_phycfg[2]`: Recovery link request. Directs the LTSSM to the Recovery state. ■ `req_phycfg[1]`: Hot Reset request. Directs the LTSSM to the Hot Reset state. ■ `req_phycfg[0]`: Disable Link request. Directs the LTSSM to the Disable state.
`vc0_tcmap_pld[7:1]`	I	Each bit corresponds to a Traffic Class. Bits[7:1] correspond to Traffic Class 7–Traffic Class 1. When a bit is set to 1, indicates that the corresponding Traffic Class maps to VC0. Traffic Class 0 always maps to Virtual Channel 0.
`inh_dllp`	I	When asserted, all TLP and DLLP transmission requests are stalled at the TX Data Link to PHY interface, except for NAK and Power Management DLLPs.
`inh_tx_tlp`	I	When asserted, all TLP transmission requests are stalled at the TX Transaction Layer to Data Link Layer interface. The Application Layer should assert this signal when requesting a low-power state.
`req_wake`	I	When asserted, requests the LTSSM to exit from a low-power state.
`link3_ctl[1]`	I	`Link Control 3` Register bit[1]: `Link Equalization Request Interrupt Enable`. When set to 1, enables the generation of an interrupt to indicate that the `Link Equalization Request` bit has been set.
`link3_ctl[0]`	I	`Link Control 3` Register bit [0]. `Perform Equalization`. When set to 1, the downstream port must perform equalization.

Table 8–20 describes the Configuration Space Bypass signals that are driven from the hard Configuration Space in the Transaction Layer to the soft Configuration Space implemented in the Application Layer. In Qsys, the Configuration Space Bypass output signals have the prefix *bypass_out_cfgbp_*. In the MegaWizard Plug-In Manager design flow, these signals have the prefix *cfgbp_*. All signals are synchronous to pld_clk.

Table 8–20. Configuration Space Bypass Mode Output Signals (Part 1 of 3)

Signal Name	Dir	Description
`lane_err[7:0]`	O	When a bit is set to 1, indicates a lane error that is reported in the `Lane Error Status Register` of the Secondary PCI Express Extended Capability structure. Bit 0 corresponds to lane 0.
`link_equiz_req`	O	Reported as Bit 5 in the `Link Status 2 Register`.
`equiz_complete`	O	Reported as Bit 1 in the `Link Status 2 Register`.
`phase_3_successful`	O	Reported as Bit 4 in the `Link Status 2 Register`.
`phase_2_successful`	O	Reported as Bit 3 in the `Link Status 2 Register`.
`phase_1_successful`	O	Reported as Bit 2 in the `Link Status 2 Register`.
`current_deemph`	O	Current de-emphasis setting reported in the `Link Status 2 Register`.
`current_speed[1:0]`	O	Current link speed as reported in the `Link Status Register`. The following encodings are defined: ■ 2'b01: Gen1 ■ 2'b10: Gen2 ■ 2'b11: Gen3
`link_up`	O	When asserted indicates that the link is up and has exited the Configuration.Idle state.
`link_train`	O	Reported as Bit 10 of the `Link Status Register`. When asserted, indicates that the link is training.
`l0state`	O	When asserted, indicates that the LTSSM is in the L0 state.
`l0sstate`	O	When asserted, indicates that the LTSSM is in L0 or L0s state.
`rx_val_pm[0]`	O	When asserted, indicates the Configuration Space has received a Power Management DLLP.
`rx_typ_pm[2:0]`	O	Signals the type of received PM DLLP. Has the following values: ■ 000b: PM_Enter_L1 ■ 001b: PM_Enter_L2L3 ■ 011b: PM_AS_Request_L1 ■ 100b: PM_Request_Ack
`tx_ack_pm`	O	Pulse. Ack from TX Data Link in response to Power Management DLLP request `tx_req_pm_pld`.
`ack_phypm[1:0]`	O	Acknowledge of transition to and from the low-power state when operations have been completed. The acknowledge bit mapping is the same as the request bit mapping. ■ Bit 0: `rxelecidle` deasserted or Training Sequence 1 (TS1) received, or `rxvalid` deasserted. Used for L1 exit. ■ Bit 1: RX Electrical Idle Ordered Set detected or TS1 received.

Table 8–20. Configuration Space Bypass Mode Output Signals (Part 2 of 3)

Signal Name	Dir	Description
`vc_status[0]`	O	When asserted, indicates that VC0 credits are initialized. When asserted, the `VC Negotiation Pending` bit of the `VC Resource Status Register` can be cleared indicating that negotiation is complete.
`rxfc_max`	O	When asserted, indicates the Transaction Layer has no high priority FC updates to send. High priority updates occur under the following conditions: 1. The FC update timer expires because no new credits have become available in the timeout period since last update. 2. The source has used all header credits from the last FC update and header credits have freed up. 3. The last FC update did not give the source enough data credits to send a maximum payload TLP and data credits have since freed up. Detects when TLP processing is complete for entry into low power state.
`txfc_max`	O	When asserted, indicates that Transaction Layer has enough credits to send maximum payload TLPs of all types. Used for entry into low power state.
`txbuf_emp`	O	When asserted, indicates there are no Application Layer TLPs pending for transmission in the Transaction Layer.
`rpbuf_emp`	O	When asserted, indicates the replay buffer contains no TLPs.
`dll_req`	O	When asserted, indicates that the Data Link Layer TX path has a pending request. Used to enable L0s entry or other low-power state. This is a level sensitive signal.
`link_auto_bdw_status`	O	When asserted, indicates that the LTSSM detected an autonomous bandwidth change which is reported in the `Link Status Register`.
`link_bdw_mng_status`	O	When asserted, indicates that the LTSSM detected a non-autonomous bandwidth change. Reported in the `Link Status Register`.
`rst_tx_margin_field`	O	When asserted, indicates that the Application Layer should reset the `Transmit Margin` field of the `Link Control 2 Register` in the Application Layer's soft Configuration Space.
`rst_enter_comp_bit`	O	When asserted, indicates that the Application Layer should reset the `Enter Compliance` bit in the `Link Control 2 Register` in the Application Layer's soft Configuration Space.
`rx_st_ecrcerr[3:0]`	O	Indicates which quad word on `rx_st_data` contains a TLP with an ECRC error. Only asserted during the cycle that in which the start of packet is asserted. Only valid only if internal ECRC error checking is enabled.
`err_uncorr_internal`	O	When asserted, indicates an uncorrectable internal error was detected. This is a real-time active high pulse.
`err_corr_internal`	O	When asserted, indicates a corrected internal error (ECC) detected. This is a real-time active high pulse.
`err_tlrcvovf`	O	When asserted, indicates a receiver overflow error. This is a real-time active high pulse.
`txfc_err`	O	When asserted, indicates a TX Flow Control Protocol error. This is a real-time active high pulse.
`err_tlmalf`	O	When asserted, indicates a malformed TLP was detected and dropped. This is a real-time active high pulse.
`err_surpdwn_dll`	O	When asserted, indicates a surprise down error occurred. This is a real-time active high pulse.

Table 8–20. Configuration Space Bypass Mode Output Signals (Part 3 of 3)

Signal Name	Dir	Description
`err_dllrcv`	O	When asserted, indicates a Data Link Protocol Error. This is a real-time active high pulse.
`err_dll_repnum`	O	When asserted, indicates a Replay_NUM rollover. This is a real-time active high pulse.
`err_dllreptim`	O	When asserted, indicates a Replay Timer Timeout. This is a real-time active high pulse.
`err_dllp_baddllp`	O	When asserted, indicates a bad DLLP was detected. This is a real-time active high pulse.
`err_dll_badtlp`	O	When asserted, indicates a bad TLP was detected. This is a real-time active high pulse.
`err_phy_tng`	O	When asserted, indicates a Link Training Error. This is a real-time active high pulse.
`err_phy_rcv`	O	When asserted, indicates a Receiver Error. This is a real-time active high pulse.
`root_err_reg_sts`	O	When asserted, indicates a a bit in the Root Error Status Register is set. The Application Layer can read this register using the LMI. This bit clears when read.
`corr_err_reg_sts`	O	When set to 1, indicates that a bit in Correctable Error Status Register is set. The Application Layer can read this register using the LMI. This bit clears when read.
`unc_err_reg_sts`	O	When set to 1, indicates that a bit in Uncorrectable Error Status Register is set. The Application Layer can read this register using the LMI. This bit clears when read.

Parity Signals

You enable parity checking by selecting **Enable byte parity ports on the Avalon-ST interface** under the **System Settings** heading of the parameter editor. Parity is odd. This option is not available for the Avalon-MM Arria V GZ Hard IP for PCI Express. Parity protection provides some data protection in systems that do not use ECRC checking.

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.

Table 8–21 lists the signals that indicate parity errors. When an error is detected, parity error signals are asserted for one cycle.

Table 8–21. Parity Signals

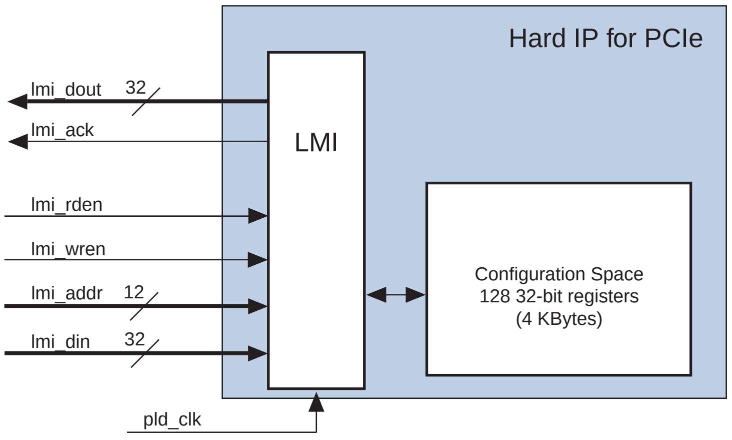
Signal Name	Direction	Description
`tx_par_err[1:0]`	O	When asserted for a single cycle, indicates a parity error during TX TLP transmission. These errors are logged in the VSEC register. The following encodings are defined: ■ 2'b10: A parity error was detected by the TX Transaction Layer. The TLP is nullified and logged as an uncorrectable internal error in the VSEC registers. For more information, refer to "Uncorrectable Internal Error Status Register" on page 9–9. ■ 2'b01: Some time later, the parity error is detected by the TX Data Link Layer. which drives 2'b01 to indicate the error. Altera recommends resetting the Arria V GZ Hard IP for PCI Express when this error is detected. Contact Altera 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`	O	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" on page 9–9. 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`	O	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" on page 9–9. If this error occurs, you must reset the core because parity errors can put the Hard IP in an unknown state.
`cfg_par_err`	O	Indicates that a parity error in a TLP routed to the internal Configuration Space or to the Configuration Space Shadow Extension Bus. This error is also logged in the Vendor Specific Extended Capability internal error register. You must reset the Hard IP if this event occurs.

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 8–34 illustrates the LMI interface.

Figure 8–34. 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 read and write data are always 32 bits. 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.

You can also use the Configuration Space signals to read Configuration Space registers. For more information, refer to "Transaction Layer Configuration Space Signals" on page 8–36.

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.

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

Table 8–22 describes the signals that comprise the LMI interface.

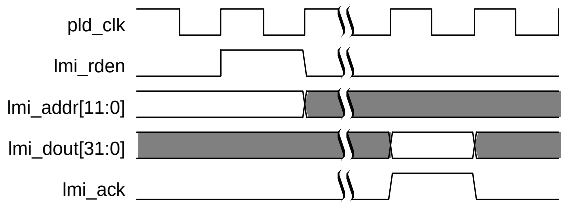
Table 8–22. LMI Interface

Signal	Width	Dir	Description
`lmi_dout`	32	O	Data outputs
`lmi_rden`	1	I	Read enable input
`lmi_wren`	1	I	Write enable input
`lmi_ack`	1	O	Write execution done/read data valid
`lmi_addr`	12	I	Address inputs, [1:0] not used
`lmi_din`	32	I	Data inputs

LMI Read Operation

Figure 8–35 illustrates the read operation.

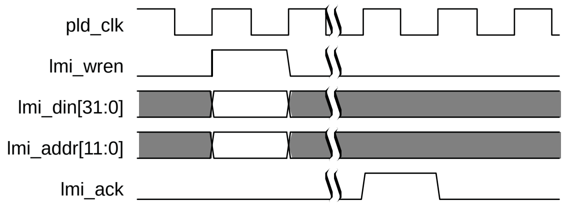
Figure 8–35. LMI Read



LMI Write Operation

Figure 8–36 illustrates the LMI write. Only writeable 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.

Figure 8–36. LMI Write



Hard IP Reconfiguration Interface

The Hard IP reconfiguration interface is consists of an Avalon-MM slave interface with a 10-bit address and 16-bit data. You can use this bus dynamically modify the value of configuration registers that are read-only at run time. To ensure proper system operation, Altera recommends that you reset or repeat device enumeration of the PCI Express link after changing the value of read-only configuration registers of the Hard IP. For a description of the registers available via this interface refer to Chapter 17, Hard IP Reconfiguration and Transceiver Reconfiguration.

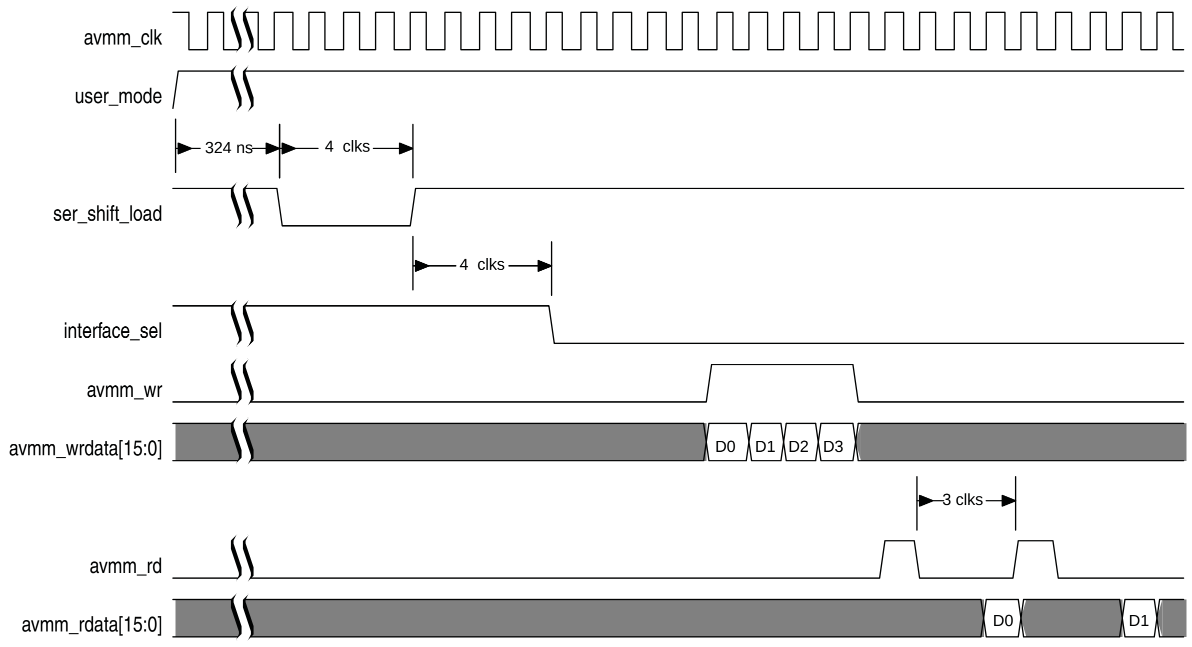
Table 8–25 describes the Hard IP reconfiguration signals.

Table 8–23. Hard IP Reconfiguration Signals

Signal	I/O	Description
`hip_reconfig_clk`	I	Reconfiguration clock. The frequency range for this clock is 50–125 MHz.
`hip_reconfig_rst_n`	I	Active-low Avalon-MM reset. Resets all of the dynamic reconfiguration registers to their default values as described in Table 17–1 on page 17–1.
`hip_reconfig_address[9:0]`	I	The 10-bit reconfiguration address.
`hip_reconfig_read`	I	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]`	O	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`	I	Write signal.
`hip_reconfig_writedata[15:0]`	I	16-bit write model.
`hip_reconfig_byte_en[1:0]`	I	Byte enables, currently unused.
`ser_shift_load`	I	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`	I	A selector which must be asserted when performing dynamic reconfiguration. Drive this signal low 4 clock cycles after the release of `ser_shift_load`.

Figure 8–37 shows the timing of writes and reads on the Hard IP reconfiguration bus.

Figure 8–37. 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*.

Table 8–24. Reconfiguration Block Signals

Signal	I/O	Description
`hip_reconfig_clk`	I	Reconfiguration clock for the Hard IP implementation. This clock should not exceed 70MHz.
`hip_reconfig_rst_n`	I	Active-low Avalon-MM reset. Resets all of the dynamic reconfiguration registers to their default values as described in Table 17–1 on page 17–1.
`hip_reconfig_address[9:0]`	I	A 10-bit address.
`hip_reconfig_read`	I	Read signal.
`hip_reconfig_readdata[15:0]`	O	16-bit read data bus.
`hip_reconfig_write`	I	Write signal.
`hip_reconfig_writedata[15:0]`	I	16-bit write data bus.
`hip_reconfig_byte_en[1:0]`	I	Byte enables.
`ser_shift_load`	O	A pulse on this signal duplicates the global configuration registers to a space the you can update using the Hard IP reconfiguration signals.
`interface_sel`	I	Chipselect.

Power Management Signals

Table 8–25 describes the power management signals.

Table 8–25. Power Management Signals

Signal	I/O	Description
`pme_to_cr`	I	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`	O	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`	I	Power Management Event. This signal is only available for Endpoints. 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 exists from the low-power state to send the message. This signal is positive edge-sensitive.

Table 8–25. Power Management Signals

Signal	I/O	Description
pm_data[9:0]	I	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	I	Power Management Auxiliary Power: This signal can be tied to 0 because the L2 power state is not supported.

Table 8–26 shows the layout of the Power Management Capabilities register.

Table 8–26. Power Management Capabilities Register

31 24	22 16	15	14 13	12 9	8	7 2	1 0
data register	rsvd	PME_status	data_scale	data_select	PME_EN	rsvd	PM_state

Table 8–27 describes the use of the various fields of the Power Management Capabilities register.

Table 8–27. Power Management Capabilities Register Field Descriptions (Part 1 of 2)

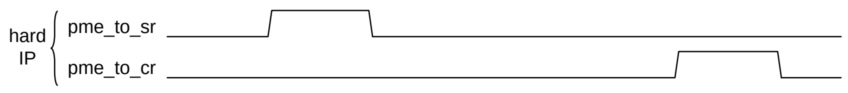
Bits	Field	Description
[31:24]	Data register	This field indicates in which power states a function can assert the PME# message.
[22:16]	reserved	—
[15]	PME_status	When set to 1, indicates that the function would normally assert the PME# message independently of the state of the PME_en bit.
[14:13]	data_scale	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.
[8]	PME_EN	1: indicates that the function can assert PME# 0: indicates that the function cannot assert PME#

Table 8–27. Power Management Capabilities Register Field Descriptions (Part 2 of 2)

Bits	Field	Description
[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 8–38 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`.

Figure 8–38. pme_to_sr and pme_to_cr in an Endpoint IP core



Avalon-MM Hard IP for PCI Express Top-Level Signals

Figure 8–39 illustrates the signals of the full-featured Arria V GZ Hard IP for PCI Express using the Avalon-MM interface available in the Qsys design flow.

In Figure 8–39, signals listed for `rxm_bar0` are also exist for `rxm_bar1` through `rxm_bar5` when those BARs are enabled in the parameter editor.

Figure 8–39. Signals in the Qsys Avalon-MM Arria V GZ Hard IP for PCI Express

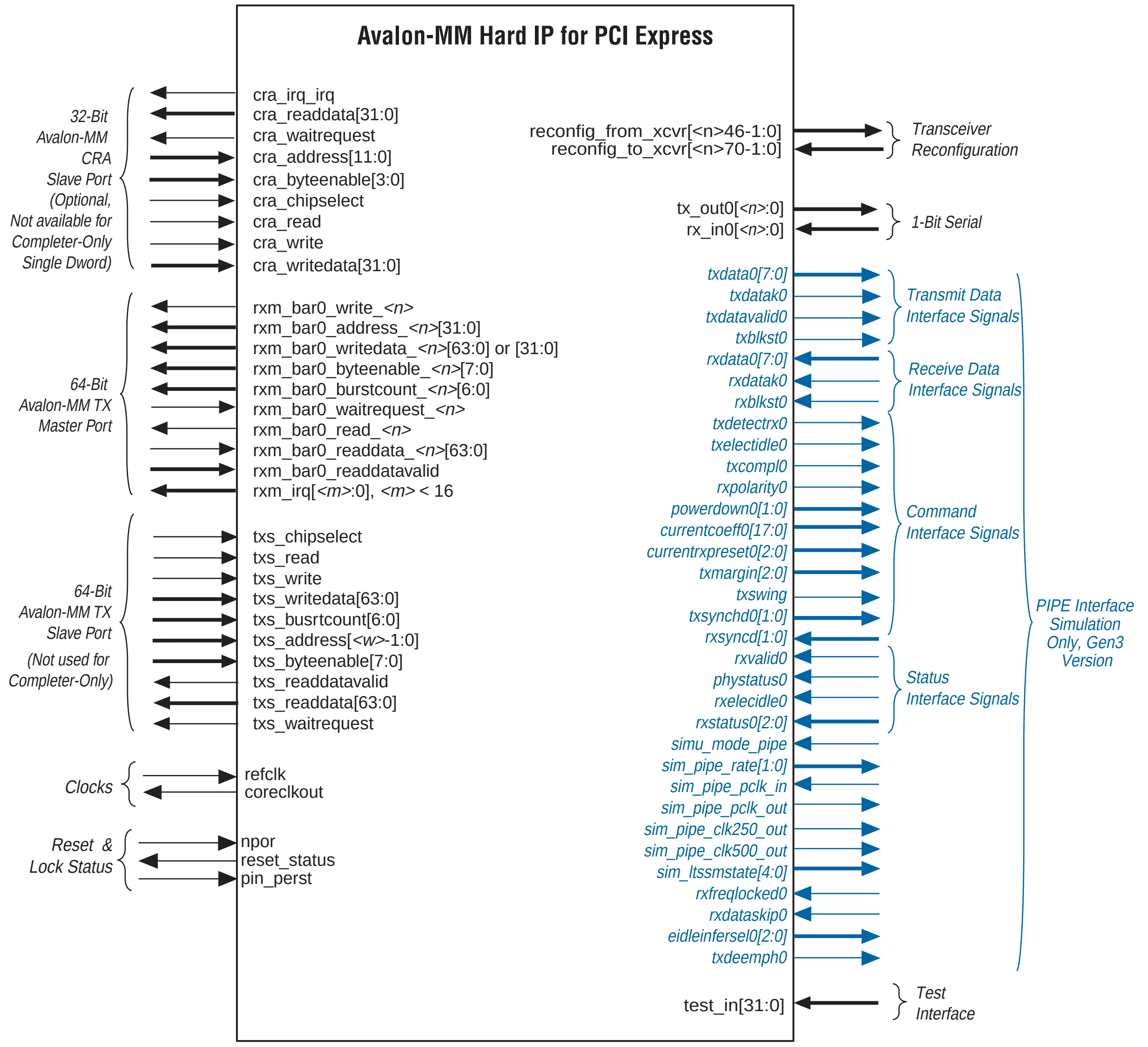


Table 8–28 lists the interfaces of the Avalon-MM Arria V GZ Hard IP for PCI Express with links to the sections that describe them.

Table 8–28. Signal Groups in the Avalon-MM Arria V GZ Hard IP for PCI Express Variants

Signal Group	Full Featured	Completer Only Single DWord	Description
Logical			
Avalon-MM CRA Slave	✓	—	"32-Bit Non-Bursting Avalon-MM Control Register Access (CRA) Slave Signals" on page 8–55
Avalon-MM RX Master	✓	✓	"RX Avalon-MM Master Signals" on page 8–56
Avalon-MM TX Slave	✓	—	"64-Bit Bursting TX Avalon-MM Slave Signals" on page 8–57
Clock	✓	✓	"Clock Signals" on page 8–28
Reset and Status	✓	✓	"Reset Signals, Status, and Link Training Signals" on page 8–28
Multiple MSI/MSI-X Interrupt Support	✓	—	"Interrupts for Endpoints when Multiple MSI/MSI-X Support Is Enabled" on page 8–33
Physical and Test			
Transceiver Control	✓	✓	"Transceiver Reconfiguration" on page 8–59
Serial	✓	✓	"Serial Interface Signals" on page 8–60
Pipe	✓	✓	"PIPE Interface Signals" on page 8–66
Test	✓	✓	"Test Signals" on page 8–69

Variations with Avalon-MM interface implement the Avalon-MM protocol described in the *Avalon Interface Specifications.* Refer to this specification for information about the Avalon-MM protocol, including timing diagrams.

32-Bit Non-Bursting Avalon-MM Control Register Access (CRA) Slave Signals

The optional CRA port for the full-featured IP core allows upstream PCI Express devices and external Avalon-MM masters to access internal control and status registers.

Table 8–29 describes the CRA slave signals.

Table 8–29. Avalon-MM CRA Slave Interface Signals (Part 1 of 2)

Signal Name	I/O	Type	Description
`cra_irq_irq`	O	Irq	Interrupt request. A port request for an Avalon-MM interrupt.
`cra_readdata[31:0]`	O	Readdata	Read data lines
`cra_waitrequest`	O	Waitrequest	Wait request to hold off more requests
`cra_address[11:0]`	I	Address	An address space of 16,384 bytes is allocated for the control registers. Avalon-MM slave addresses provide address resolution down to the width of the slave data bus. Because all addresses are byte addresses, this address logically goes down to bit 2. Bits 1 and 0 are 0.
`cra_byteenable[3:0]`	I	Byteenable	Byte enable
`cra_chipselect`	I	Chipselect	Chip select signal to this slave

Table 8–29. Avalon-MM CRA Slave Interface Signals (Part 2 of 2)

Signal Name	I/O	Type	Description
`cra_read`	I	Read	Read enable
`cra_write`	I	Write	Write request
`cra_writedata[31:0]`	I	Writedata	Write data

RX Avalon-MM Master Signals

This Avalon-MM master port propagates PCI Express requests to the Qsys interconnect fabric. For the full-feature IP core it propagates requests as bursting reads or writes. A separate Avalon-MM master port corresponds to each BAR. Signals that include lane number 0 also exist for BAR1–BAR5 when additional BARs are enabled. Table 8–30 lists the RX Master interface ports.

Table 8–30. Avalon-MM RX Master Interface Signals

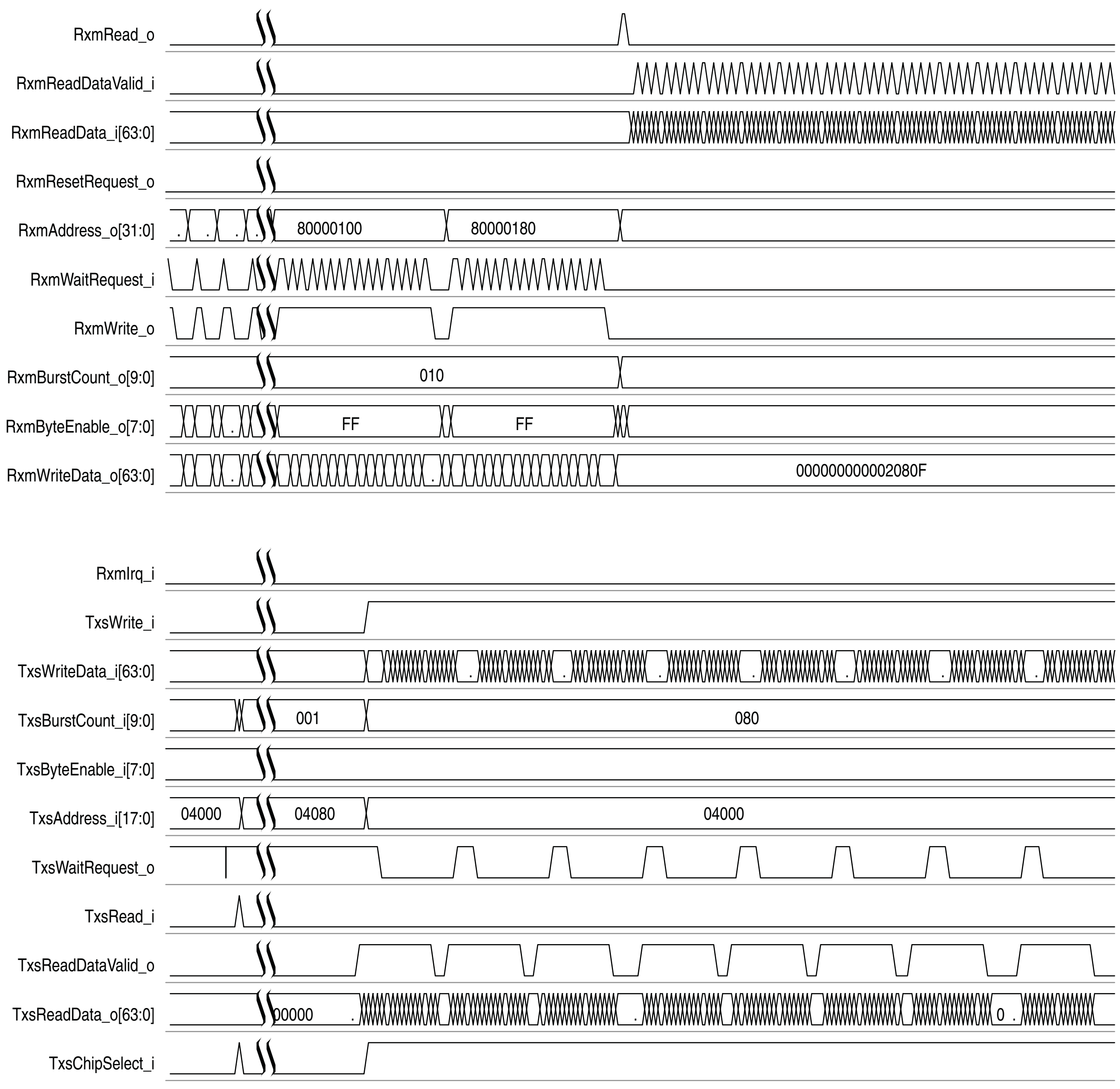
Signal Name	I/O	Description
`rxm_bar0_write_<n>` (1)	O	Asserted by the core to request a write to an Avalon-MM slave.
`rxm_bar0_address_<n>[31:0]`	O	The address of the Avalon-MM slave being accessed.
`rxm_bar0_writedata_<n>[<w>-1:0]`	O	RX data being written to slave. <*w*> = 64 for the full-featured IP core. <*w*> = 32 for the completer-only IP core.
`rxm_bar0_byteenable_<n>[<w>-1:0]`	O	Byte enable for write data.
`rxm_bar0_burstcount_<n>[6:0]`	O	The burst count, measured in qwords, of the RX write or read request. The width indicates the maximum data that can be requested. The maximum data in a burst is 512 bytes.
`rxm_bar0_waitrequest_<n>`	I	Asserted by the external Avalon-MM slave to hold data transfer.
`rxm_bar0_read_<n>`	O	Asserted by the core to request a read.
`rxm_bar0_readdata_<n>[<w>-1:0]`	I	Read data returned from Avalon-MM slave in response to a read request. This data is sent to the IP core through the TX interface. <*w*> = 64 for the full-featured IP core. <*w*> = 32 for the completer-only IP core.
`rxm_bar0_readdatavalid_<n>`	I	Asserted by the system interconnect fabric to indicate that the read data on is valid.
`rxm_irq_<n>[<m>:0]`	I	Indicates an interrupt request asserted from the system interconnect fabric. This signal is only available when the CRA port is enabled. Qsys-generated variations have as many as 16 individual interrupt signals (<*m*> ≤ 15).

Note to Table 8–30:

(1) <n> represents the BAR number for all signals. The core supports up to 6 BARs.

Figure 8–40 illustrates the RX master port propagating requests to the Application Layer and also shows simultaneous, DMA read and write activity

Figure 8–40. Simultaneous DMA Read, DMA Write, and Target Access



64-Bit Bursting TX Avalon-MM Slave Signals

This optional Avalon-MM bursting slave port propagates requests from the interconnect fabric to the full-featured Avalon-MM Arria V GZ Hard IP for PCI Express. Requests from the interconnect fabric are translated into PCI Express request packets. Incoming requests can be up to 512 bytes. For better performance, Altera recommends using smaller read request size (a maximum of 512 bytes).

Table 8–31 lists the TX slave interface signals.

Table 8–31. Avalon-MM TX Slave Interface Signals

Signal Name	I/O	Description
`txs_chipselect`	I	The system interconnect fabric asserts this signal to select the TX slave port.
`txs_read`	I	Read request asserted by the system interconnect fabric to request a read.
`txs_Write`	I	Write request asserted by the system interconnect fabric to request a write. The Avalon-MM Arria V GZ Hard IP for PCI Express requires that the Avalon-MM master assert this signal continuously from the first data phase through the final data phase of the burst. The Avalon-MM master Application Layer must guarantee the data can be passed to the interconnect fabric with no pauses. This behavior is most easily implemented with a store and forward buffer in the Avalon-MM master.
`txs_writedata[63:0]`	I	Write data sent by the external Avalon-MM master to the TX slave port.
`txs_burstcount[6:0]`	I	Asserted by the system interconnect fabric indicating the amount of data requested. The count unit is the amount of data that is transferred in a single cycle, that is, the width of the bus. The burst count is limited to 512 bytes.
`txs_address[<w>-1:0]`	I	Address of the read or write request from the external Avalon-MM master. This address translates to 64-bit or 32-bit PCI Express addresses based on the translation table. The <*w*> value is determined when the system is created.
`txs_byteenable[<w>:0]`	I	Write byte enable for data. A burst must be continuous. Therefore all intermediate data phases of a burst must have a byte enable value of 0xFF. The first and final data phases of a burst can have other valid values.
`txs_readdatavalid`	O	Asserted by the bridge to indicate that read data is valid.
`txs_readdata[63:0]`	O	The bridge returns the read data on this bus when the RX read completions for the read have been received and stored in the internal buffer.
`txs_waitrequest`	O	Asserted by the bridge to hold off write data when running out of buffer space. If this signal is asserted during an operation, the master should maintain the `txs_Read` signal (or `txs_Write` signal and `txs_WriteData`) stable until after `txs_WaitRequest` is deasserted.

Clocks

Refer to "Clock Signals" on page 8–28.

MSI and MSI-X Interfaces

The IP Core supports both MSI and MSI-X interrupts when both are enabled in the GUI. The Application Layer uses the information from this interface to send MSI and MSI-X to the Root Port via the Tx Slave Control Interface

Table 8–32. CRA Slave Interface

Signal Name	I/O	Description
`MsiIntf[81:0]`	Output	This bus provides the following MSI address, data, and enabled signals: ■ `msi_intf[81]`: Master enable ■ `msi_intf[80}`: MSI enable ■ `msi_intf[79:64]`: MSI data ■ `msi_intf[63:0]`: MSI address
`MsixIntfc_o[15:0]`	Output	Message Control Register for MSI-X as defined *Section 6.8.2* of the *PCI Local Bus Specification, Rev. 3.0*.

Physical Layer Interface Signals

For V-Series devices, Altera provides an integrated solution with the Transaction, Data Link and Physical Layers. The MegaWizard Plug-In Manager generates a SERDES variation file, <*variation*>**_serdes.<v** or **vhd** >, in addition of the Hard IP variation file, <*variation*>.<**v** or **vhd**>. For Arria V GZ devices the SERDES entity is included in the library files for PCI Express.

Transceiver Reconfiguration

Dynamic reconfiguration compensates for variations due to process, voltage and temperature (PVT). Among the analog settings that you can reconfigure are: $\mathbf{V}_{OD}$, pre-emphasis, and equalization.

You can use the Altera Transceiver Reconfiguration Controller to dynamically reconfigure analog settings in Arria V GZ devices. For more information about instantiating the Altera Transceiver Reconfiguration Controller IP core refer to Chapter 17, Hard IP Reconfiguration and Transceiver Reconfiguration.

Table 8–33 describes the transceiver support signals. In Table 8–33, <*n*> is the number of interfaces required.

Table 8–33. Transceiver Control Signals

Signal Name	I/O	Description
`reconfig_from_xcvr[(<n>46)-1:0]`	O	Reconfiguration signals to the Transceiver Reconfiguration Controller.
`reconfig_to_xcvr[(<n>70)-1:0]`	I	Reconfiguration signals from the Transceiver Reconfiguration Controller.

Table 8–34 shows the number of logical reconfiguration and physical interfaces required for various configurations. The Quartus II fitter merges logical interfaces so that there are fewer physical interfaces in the hardware. Typically, one logical interface is required for each channel and one for each PLL. The ×8 variants require an extra channel for PCS clock routing and control.

Table 8–34. Number of Logical and Physical Reconfiguration Interfaces

Variant	Logical Interfaces
Gen1 and Gen2 ×1	2
Gen1 and Gen2 ×2	3
Gen1 and Gen2 ×4	5
Gen1 and Gen2 ×8	10
Gen3 ×1	3
Gen3 ×2	4
Gen3 ×4	6
Gen3 ×8	11

For more information about the refer to the "Transceiver Reconfiguration Controller" chapter in the *Altera Transceiver PHY IP Core User Guide*.

The following sections describe signals for the serial or parallel PIPE interlaces. The PIPE interface is only available for simulation.

Serial Interface Signals

Table 8–35 describes the serial interface signals.

Table 8–35. 1-Bit Interface Signals

Signal	I/O	Description
tx_out[7:0] (1)	O	Transmit input. These signals are the serial outputs of lanes 7–0.
rx_in[7:0] (1)	I	Receive input. These signals are the serial inputs of lanes 7–0.

Note to Table 8–35:

(1) The ×1 IP core only has lane 0. The ×2 IP core only has lanes 1–0. The ×4 IP core only has lanes 3–0.

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

Channel Placement for Gen1 and Gen2 Using CMU PLL

Figure 8–41 shows the channel placement for Gen1 and Gen2 ×1 and ×4 variants when you select the CMU PLL.

Figure 8–41. Channel Placement Gen1 and Gen2 x1 and x4 Variants

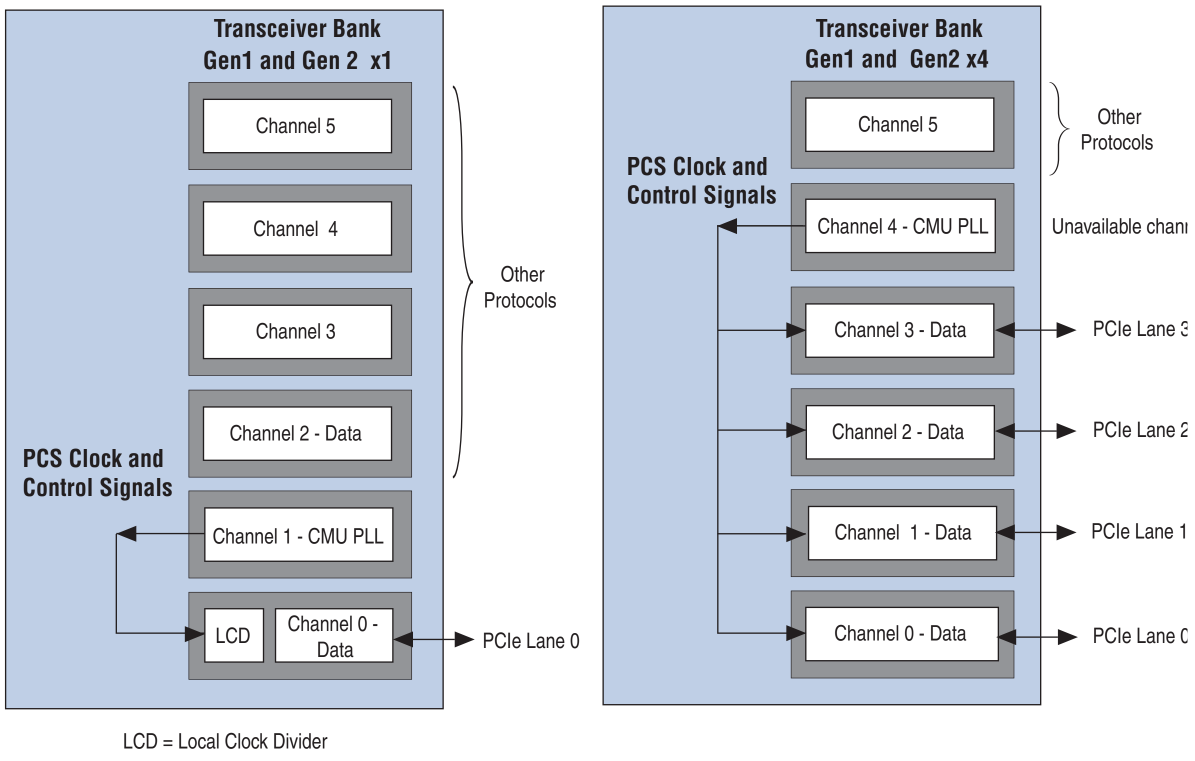


Figure 8–42 shows the channel placement for Gen1 and Gen2 ×8 variants when you select the CMU PLL.

Figure 8–42. Channel Placement Gen1 and Gen2 ×8 Variants Using CMU PLL

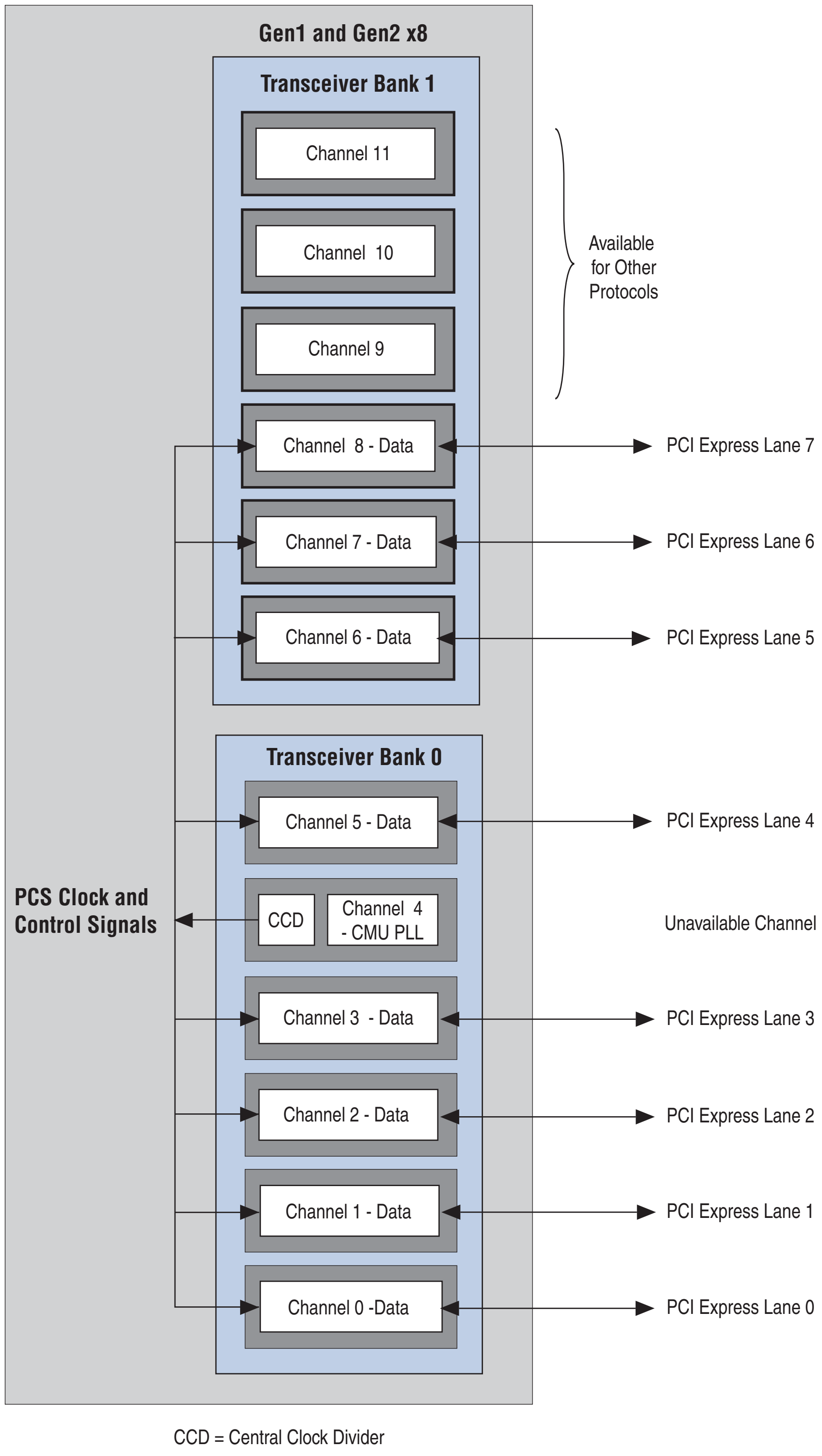
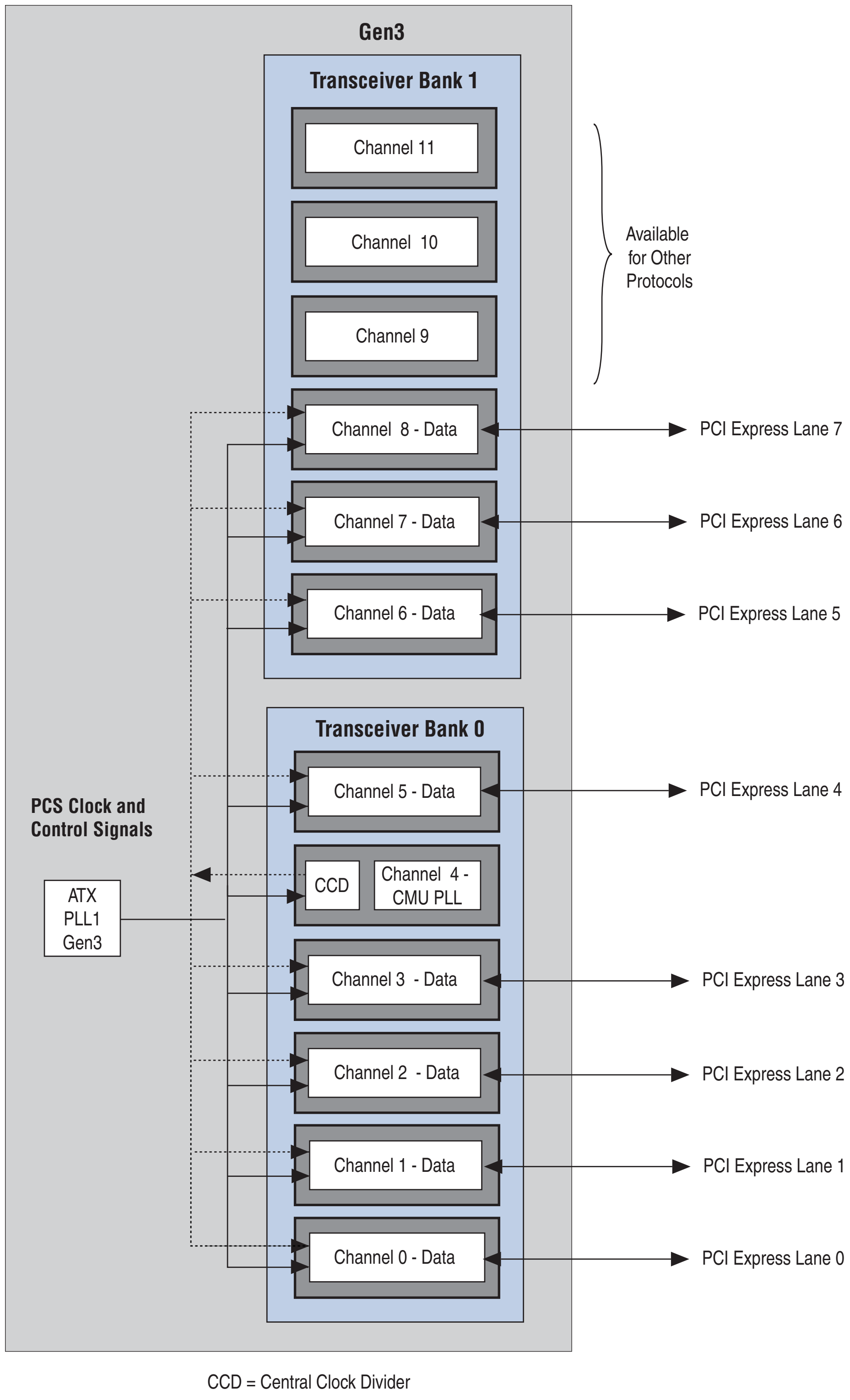


Figure 8–43 shows the channel placement for Gen3 ×8 variants. This variant requires two PLLs to facilitate rate switching between Gen1, Gen2, and Gen3.

Figure 8–43. Channel Placement Gen3 ×8 Variants Using ATX PLL



Channel Placement for Gen1 and Gen2 Using ATX PLL

Selecting the ATX PLL has the following advantages over the CMU PLL:

- The ATX PLL saves one channel in Gen1 and Gen2 ×1 and ×4 configurations.
- The ATX PLL has better jitter performance than the CMU PLL.

Figure 8–44 illustrates the channel placement for Gen1 and Gen2 ×1 and ×4 variants when you select the ATX PLL.

Figure 8–44. Channel Placement Gen1 and Gen2 Using ATX PLL

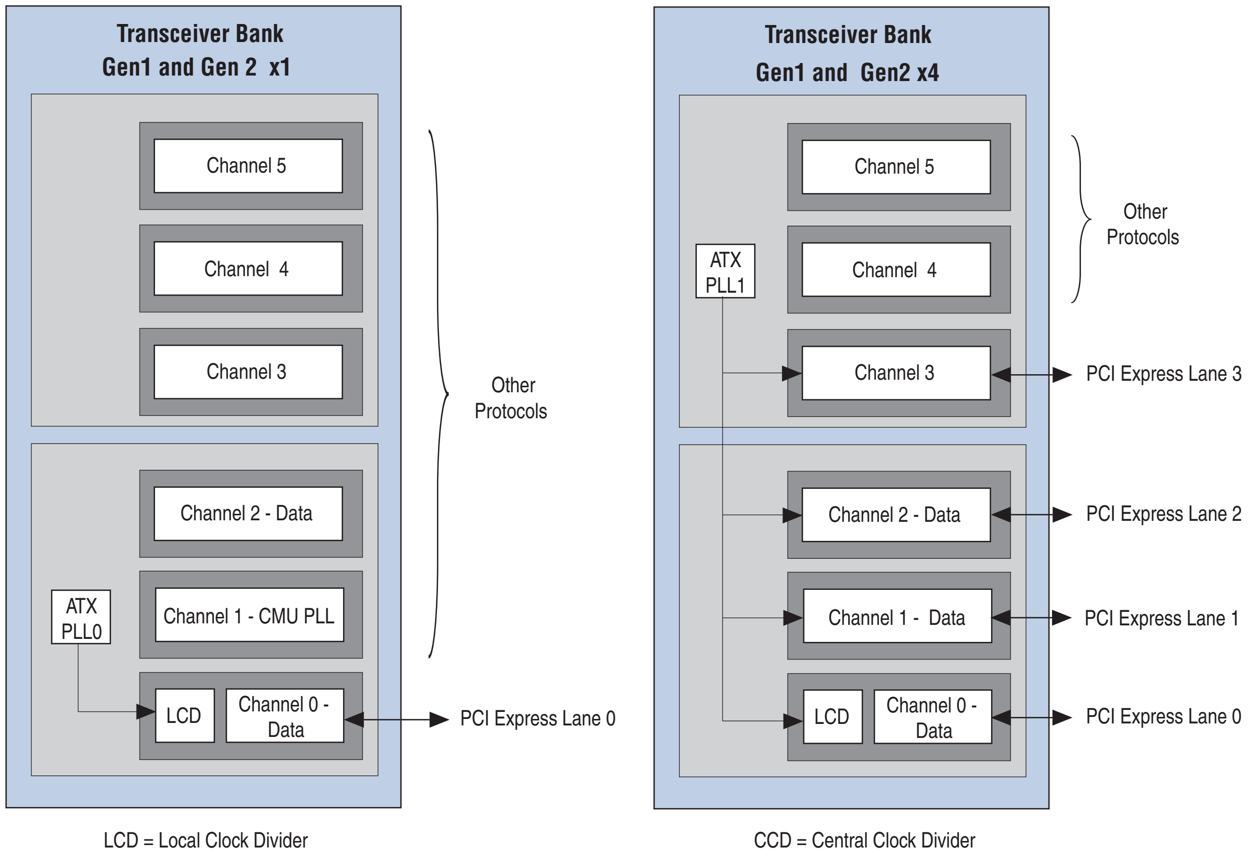
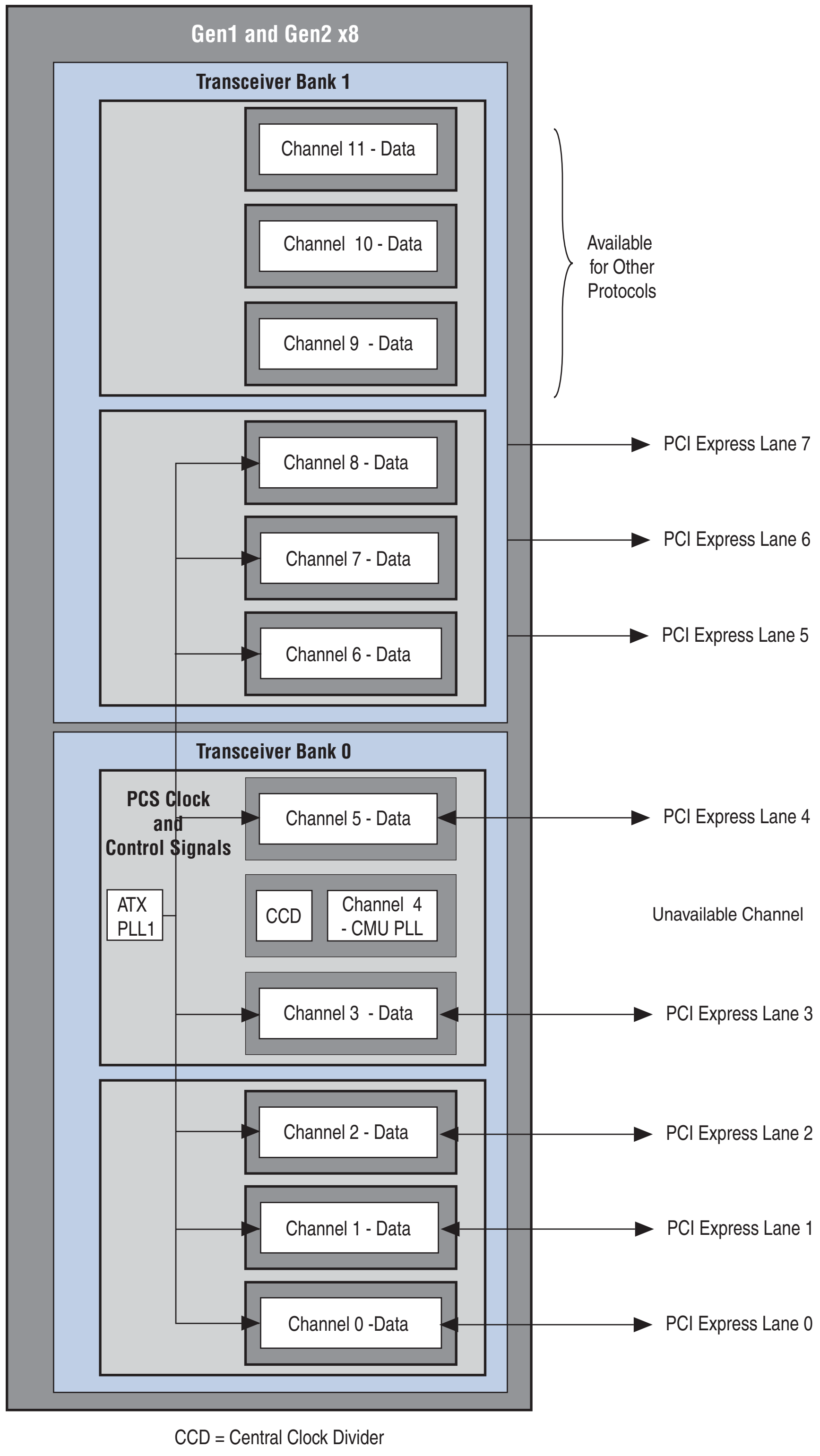


Figure 8–45 illustrates channel placement for Gen1 and Gen2 ×8 when you select the ATX PLL.

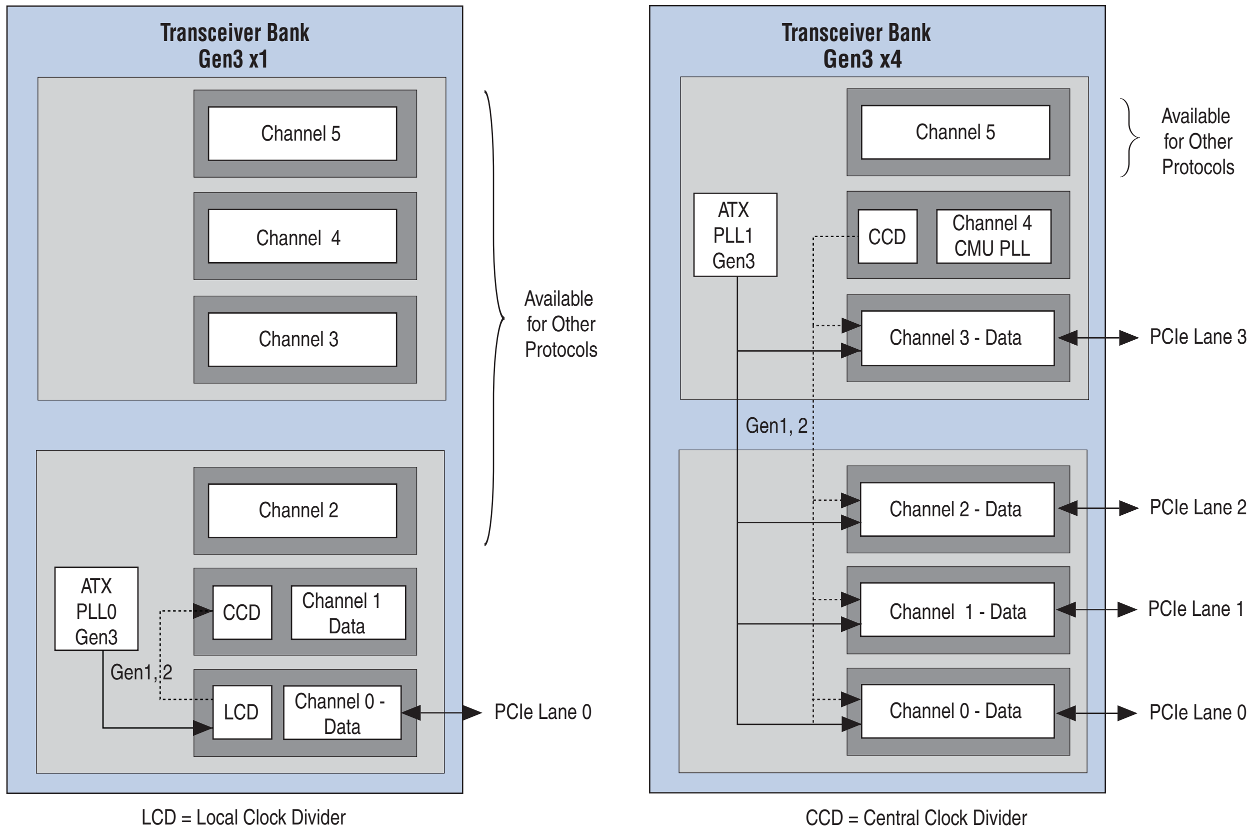
Figure 8–45. Channel Placement Gen1 and Gen2 ×8 Using ATX PLL



Channel Placement for Gen3 Using Both CMU and ATX PLLs

Figure 8–46 illustrates channel placement for Gen3 ×1 and ×4 variants using the ATX PLL.

Figure 8–46. Channel Placement Gen3 ×1 and ×4



PIPE Interface Signals

These PIPE signals are available for Gen1, Gen2, and Gen 3 variants so that you can simulate using either the one-bit or the PIPE interface. Simulation is much faster using the PIPE interface. For Gen1 and Gen2 variants, the PIPE interface is 8 bits. For Gen3 variants, the PIPE interface is 32 bits. 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.

☞ The Root Port BFM bypasses Gne3 Phase 2 and Phase 3 Equalization. You must adjust your third-party Root Port BFM to terminate Equalization after Phase 0 and Phase 1 complete.

In Table 8–36, signals that include lane number 0 also exist for lanes 1-7. In Qsys, the signals that are part of the PIPE interface have the prefix, *hip_pipe*. The signals which are included to simulate the PIPE interface have the prefix, *hip_pipe_sim_pipe*.

Table 8–36. PIPE Interface Signals (Part 1 of 3)

Signal	I/O	Description
`txdata0[7:0]`	O	Transmit data <*n*> (2 symbols on lane <*n*>). This bus transmits data on lane <*n*>.
`txdatak0` (1)	O	Transmit data control <*n*>. This signal serves as the control bit for `txdata<`*n*`>`.
`txdatavalid0`	O	When asserted, the `txdata0[7:0]` is valid.
`txblkst0`	O	For Gen3 operation, indicates the start of a block.
`rxdata0[7:0]` (1) (2)	I	Receive data <*n*> (2 symbols on lane <*n*>). This bus receives data on lane <*n*>.
`rxdatak0` (1) (2)	I	Receive data <*n*>. This bus receives data on lane <*n*>.
`rxblkst0`	I	For Gen3 operation, indicates the start of a block.
`txdetectrx0` (1)	O	Transmit detect receive <*n*>. This signal tells the PHY layer to start a receive detection operation or to begin loopback.
`txelecidle` (1)	O	Transmit electrical idle <*n*>. This signal forces the TX output to electrical idle.
`txcompl0` (1)	O	Transmit compliance <*n*>. This signal forces the running disparity to negative in compliance mode (negative COM character).
`rxpolarity0` (1)	O	Receive polarity <*n*>. This signal instructs the PHY layer to invert the polarity of the 8B/10B receiver decoding block.
`powerdown0[1:0]` (1)	O	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]`	O	For Gen3, selects the transmitter de-emphasis. The 18 bits specify the following coefficients: ■ [5:0]: C_{-1} ■ [11:6]: C_0 ■ [17:12]: C_{+1} In Gen3 capable designs, the TX deemphasis for Gen2 data rates is always -6 dB. The TX deemphasis for Gen1 data rate is always -3.5 dB.
`currentrxpreset0[2:0]`	O	For Gen3 designs, specifies the current preset.
`tx_margin[2:0]`	O	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.
`txswing`	O	When asserted, indicates full swing for the transmitter voltage. When deasserted indicates half swing.
`txsynchd0[1:0]`	O	For Gen3 operation, specifies the block type. The following encodings are defined: ■ 2'b01: Ordered Set Block ■ 2'b10: Data Block

Table 8–36. PIPE Interface Signals (Part 2 of 3)

Signal	I/O	Description
`rxsynchd0[1:0]`	I	For Gen3 operation, specifies the block type. The following encodings are defined: ■ 2'b01: Ordered Set Block ■ 2'b10: Data Block
`rxvalid0` (1) (2)	I	Receive valid <*n*>. This symbol indicates symbol lock and valid data on `rxdata<n>` and `rxdatak<n>`.
`phystatus0` (1) (2)	I	PHY status <*n*>. This signal communicates completion of several PHY requests.
`rxelecidle0` (1) (2)	I	Receive electrical idle <*n*>. When asserted, indicates detection of an electrical idle.
`rxstatus0[2:0]` (1) (2)	I	Receive status <*n*>. This signal encodes receive status and error codes for the receive data stream and receiver detection.
`simu_mode_pipe`	O	When set to 1, the PIPE interface is in simulation mode.
`sim_pipe_rate[1:0]`	O	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`	I	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_pclk_out`		TX datapath clock to the BFM PHY. `pclk_out` is derived from refclk and provides the source synchronous clock for TX data from the PHY.
`sim_pipe_clk250_out`		Used to generate `pclk`.
`sim_pipe_clk500_out`		Used to generate `pclk`.
`sim_pipe_ltssmstate0[4:0]`		LTSSM state: The LTSSM state machine encoding defines the following states: ■ 5’b00000: Detect.Quiet ■ 5’b 00001: 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

Table 8–36. PIPE Interface Signals (Part 3 of 3)

Signal	I/O	Description
`sim_pipe_ltssmstate0[4:0]` (continued)		■ 5’b10001: Loopback.Entry ■ 5’b10010: Loopback.Active ■ 5’b10011: Loopback.Exit ■ 5’b10100: Hot.Reset ■ 5’b10101: LOs ■ 5’b11001: L2.transmit.Wake ■ 5’b11010: Speed.Recovery ■ 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` (1) (2)	I	When asserted indicates that the `pclk_in` used for PIPE simulation is valid.
`rxdataskip0`	O	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
`eidleinfersel0[2:0]`	O	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 the in 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
`tx_deemph0`	O	Transmit de-emphasis selection. The Arria V GZ 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.
`testin_zero`	O	When asserted, indicates accelerated initialization for simulation is active.

Notes to Table 8–36:

(1) Signals that include lane number 0 also exist for lanes 1-7.

(2) These signals are for simulation only. For Quartus II software compilation, these pipe signals can be left floating.

Test Signals

The `test_in` and `test_out` buses provide run-time control and monitoring of the internal state of the Arria V GZ Hard IP for PCI Express. Table 8–37 describes the test signals.

CAUTION Altera recommends that you use the test_out and test_in signals for debug or non-critical status monitoring purposes such as LED displays of PCIe link status. They should not be used for design function purposes. Use of these signals will make it more difficult to close timing on the design. The test signals have not been rigorously verified and will not function as documented in some corner cases. The debug signals provided on test_out are not available in the current release.

Table 8–37 describes the test_in bus signals. In Qsys these signals have the prefix, *hip_ctrl_*.

Table 8–37. Test Interface Signals (1), (2)

Signal	I/O	Description
test_in[31:0]	I	The bits of the test_in bus have the following definitions: ■ [0]: Simulation mode. This signal can be set to 1 to accelerate initialization by reducing the value of many initialization counters. ■ [4:1]: Reserved. Must be set to 4'b0100. ■ [5]: Compliance test mode. Disable/force compliance mode. When set, 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. ■ [31:6]–Reserved. Must be set to 26'h2.
lane_act[3:0]	O	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

Notes to Table 8–37:

(1) All signals are per lane.

(2) Refer to "PIPE Interface Signals" on page 8–67 for definitions of the PIPE interface signals.

Making Pin Assignments

Before running Quartus II compilation, use the **Pin Planner** to assign I/O standards to the pins of the device. Complete the following steps to bring up the **Pin Planner** and assign the 1.5-V pseudo-current mode logic (PCML) I/O standard to the serial data input and output pins:

1. On the Quartus II **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 **1.5-V PCML I/O** standard.

☞ The Arria V GZ Hard IP for PCI Express IP Core automatically assigns other required PMA analog settings, including 100 ohm internal termination.

9. Register Descriptions

Configuration Space Register Content

This table shows the PCI Compatible Configuration Space address map. It provides links to additional tables that provide more detail.

To facilitate finding additional information about these PCI and PCI Express registers, the following tables provide the name of the corresponding section in the *PCI Express Base Specification Revision 3.0.*

Table 9–1. PCI Configuration Space

Byte Offset	Register Set
PCI Compatible Configuration Space	
0x000:0x03C	PCI Type 0 Compatible Configuration Space Header (Refer to Table 9–2 for details.)
0x000:0x03C	PCI Type 1 Compatible Configuration Space Header (Refer to Table 9–3 for details.)
0x040:0x04C	Reserved.
0x050:0x05C	MSI Capability Structure, (Refer to Table 9–4 for details.)
0x060:0x064	Reserved
0x068:0x070	MSI-X Capability Structure, (Refer to Table 9–5 for details.)
0x070:0x074	Reserved
0x078:0x07C	Power Management Capability Structure (Refer to Table 9–6 for details.)
0x080:0x0BC	PCI Express Capability Structure (Refer to Table 9–8 for details.)
0x0C0:0x0C4	Reserved
PCI Express Extended Configuration Space	
0x100:0x16C	Virtual Channel Capability Structure
0x170:0x1FC	Reserved
0x200:0x240	Vendor Specific Extended Capability Structure (Refer to Altera-Defined Vendor Specific Extended Capability (VSEC) for details.)
0x300:0x318	Secondary PCI Express Extended Capability Structure (for Gen3 operation)
0x31C:7FC	Reserved
0x800:0x834	Advanced error reporting (AER) (optional)
0x838:0x8FF	Reserved

For comprehensive information about these registers, refer to Chapter 7 of the *PCI Express Base Specification Revision 3.0.*

The following table describes the Type 0 Configuration settings.

In the following tables, the names of fields that are defined by parameters in the parameter editor are links to the description of that parameter. These links appear as green text.

Table 9–2. PCI Type 0 Configuration Space Header (Endpoints), Rev3.0 Spec: Type 0 Configuration Space Header

Byte Offset	31:24	23:16	15:8	7:0
0x000	Device ID Device ID		Vendor ID Vendor ID	
0x004	Status		Command	
0x008	Class code Class code			Revision IDRevision ID
0x00C	0x00	Header Type (Port type)	0x00	Cache Line Size
0x010	Base Address Register (BAR) and Expansion ROM Settings			
0x014	Base Address Register (BAR) and Expansion ROM Settings			
0x018	Base Address Register (BAR) and Expansion ROM Settings			
0x01C	Base Address Register (BAR) and Expansion ROM Settings			
0x020	Base Address Register (BAR) and Expansion ROM Settings			
0x024	Base Address Register (BAR) and Expansion ROM Settings			
0x028	Reserved			
0x02C	Subsystem Device IDSubsystem Device ID		Subsystem vendor ID Subsystem vendor ID	
0x030	Expansion ROM base address			
0x034	Reserved			Capabilities Pointer
0x038	Reserved			
0x03C	0x00	0x00	Interrupt Pin	Interrupt Line

Note to Table 9–2:

(1) Refer to Table 9–39 on page 9–22 for a comprehensive list of correspondences between the Configuration Space registers and the *PCI Express Base Specification 3.0*.

Table 9–3 describes the Type 1 Configuration settings

Table 9–3. PCI Type 1 Configuration Space Header (Root Ports) Rev3.0 Spec: Type 1 Configuration Space Header (Part 1 of 2)

Byte Offset	31:24	23:16	15:8	7:0
0x0000	Device ID Device ID		Vendor IDVendor ID	
0x004	Status		Command	
0x008	Class code Class code			Revision ID Revision ID
0x00C	BIST	Header Type	Primary Latency Timer	Cache Line Size
0x010	BAR Registers BAR Registers			
0x014	BAR Registers BAR Registers			
0x018	Secondary Latency Timer	Subordinate Bus Number	Secondary Bus Number	Primary Bus Number
0x01C	Secondary Status		I/O Limit	I/O Base

Table 9–3. PCI Type 1 Configuration Space Header (Root Ports) Rev3.0 Spec: Type 1 Configuration Space Header (Part 2 of 2)

Byte Offset	31:24	23:16	15:8	7:0
0x020	Memory Limit		Memory Base	
0x024	Prefetchable Memory Limit		Prefetchable Memory Base	
0x028	Prefetchable Base Upper 32 Bits			
0x02C	Prefetchable Limit Upper 32 Bits			
0x030	I/O Limit Upper 16 Bits		I/O Base Upper 16 Bits	
0x034	Reserved			Capabilities Pointer
0x038	Expansion ROM Base Address			
0x03C	Bridge Control		Interrupt Pin	Interrupt Line

Note to Table 9–3:

(1) Refer to Table 9–39 on page 9–22 for a comprehensive list of correspondences between the Configuration Space registers and the *PCI Express Base Specification 2.0.*

Table 9–4 describes the MSI Capability structure.

Table 9–4. MSI Capability Structure, Rev3.0 Spec: MSI Capability Structures

Byte Offset (1)	31:24	23:16	15:8	7:0
0x050	Message Control Configuration MSI Control Status Register Field Descriptions		Next Cap Ptr	Capability ID
0x054	Message Address			
0x058	Message Upper Address			
0x05C	Reserved		Message Data	

Notes to Table 9–4:

(1) Specifies the byte offset within Arria V GZ Hard IP for PCI Express IP core's address space.

(2) Refer to Table 9–39 on page 9–22 for a comprehensive list of correspondences between the Configuration Space registers and the *PCI Express Base Specification 2.0.*

Table 9–5 describes the MSI-X Capability structure.

Table 9–5. MSI-X Capability Structure, Rev3.0 Spec: MSI-X Capability Structures

Byte Offset	31:24	23:16	15:8	7:3	2:0
0x068	Message Control		Next Cap Ptr	Capability ID	
0x06C	MSI-C Table Offset MSI-X Table Offset				MSI-X Table BAR Indicator MSI-X Table BAR Indicator

Table 9–5. MSI-X Capability Structure, Rev3.0 Spec: MSI-X Capability Structures

Byte Offset	31:24	23:16	15:8	7:3	2:0
0x070	`MSI-X Pending Bit Array (PBA) Offset` `MSI-X Pending Bit Array (PBA) Offset`				MSI-X Pending Bit Array - BAR Indicator MSI-X Pending Bit Array – BAR Indicator

Note to Table 9–5:

(1) Refer to Table 9–39 on page 9–22 for a comprehensive list of correspondences between the Configuration Space registers and the *PCI Express Base Specification 2.0.*

Table 9–6 describes the Power Management Capability structure.

Table 9–6. Power Management Capability Structure, Rev3.0 Spec

Byte Offset	31:24	23:16	15:8	7:0
0x078	Capabilities Register		Next Cap PTR	Cap ID
0x07C	Data	PM Control/Status Bridge Extensions	Power Management Status & Control	

Note to Table 9–6:

(1) Refer to Table 9–39 on page 9–22 for a comprehensive list of correspondences between the Configuration Space registers and the *PCI Express Base Specification 2.0.*

Table 9–7 describes the PCI Express AER Extended Capability structure.

Table 9–7. PCI Express AER Capability Structure, Rev3.0 Spec: AER Capability

Byte Offset	31:24	23:16	15:8	7:0
0x800	PCI Express Enhanced Capability Header			
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			
0x830	Root Error Status			
0x834	Error Source Identification Register		Correctable Error Source ID Register	

Note to Table 9–7:

(1) Refer to Table 9–39 on page 9–22 for a comprehensive list of correspondences between the Configuration Space registers and the *PCI Express Base Specification 2.0.*

Table 9–8 describes the PCI Express Capability Structure.

Table 9–8. PCIe Capability Structure 3.0, Rev3.0

Byte Offset	31:16	15:8	7:0
0x080	PCI Express Capabilities Register	Next Cap Pointer	PCI Express Cap ID
0x084	Device CapabilitiesDevice Capabilities		
0x088	Device Status	Device Control	
0x08C	Link Capabilities Link Capabilities		
0x090	Link Status	Link Control	
0x094	Slot Capabilities Slot Capabilities		
0x098	Slot Status	Slot Control	
0x09C	Root Capabilities	Root Control	
0x0A0	Root Status		
0x0A4	Device Capabilities 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	

Note to Table 9–8:

(1) Registers not applicable to a device are reserved.

(2) Refer to Table 9–39 on page 9–22 for a comprehensive list of correspondences between the Configuration Space registers and the *PCI Express Base Specification 2.0.*

Altera-Defined Vendor Specific Extended Capability (VSEC)

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

Table 9–9. Altera-Defined Vendor Specific Capability Structure (Part 1 of 2)

Byte Offset	Register Name			
	31:20	19:16	15:8	7:0
0x200	`Next Capability Offset`	`Version`	`Altera-Defined VSEC Capability Header Altera-Defined VSEC Capability Header`	
0x204	`VSEC Length`	`VSEC Rev`	`VSEC ID Altera-Defined VSEC Specific Header Altera-Defined Vendor Specific Header`	
0x208	`Altera Marker Altera 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`			

Table 9–9. Altera-Defined Vendor Specific Capability Structure (Part 2 of 2)

Byte Offset	Register Name			
	31:20	19:16	15:8	7:0
0x21C	CVP Status CvP Status		User Device or Board Type IDUser Device or Board Type ID	
0x220	CVP Mode Control CvP Mode Control			
0x228	CVP Data Register CvP Data Register			
0x22C	CVP Data Programming Control Register CvP Programming Control Register			
0x230	Reserved			
0x234	Uncorrectable Internal Error Status Register Uncorrectable Internal Error Status Register			
0x238	Uncorrectable Internal Error Mask Register Uncorrectable Internal Error Mask Register			
0x23C	Correctable Internal Error Status Register Correctable Internal Error Status Register			
0x240	Correctable Internal Error Mask RegisterCorrectable Internal Error Mask Register			

The following table defines the fields of the `Vendor Specific Extended Capability Header` register.

Table 9–10. Altera-Defined VSEC Capability Header

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

The following table defines the fields of the `Altera-Defined Vendor Specific` register. You can specify these fields when you instantiate the Hard IP; they are read-only at run-time.

Table 9–11. Altera-Defined Vendor Specific Header

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

The following table defines the `Altera Marker` register.

Table 9–12. Altera Marker

Bits	Register Description	Value	Access
[31:0]	`Altera Marker`. This read only register is an additional marker. If you use the standard Altera 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

The following table defines the `JTAG Silicon ID` registers.

Table 9–13. JTAG Silicon ID

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

The following table defines the `User Device or Board Type ID` register.

Table 9–14. User Device or Board Type ID

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

The following table defines the fields of the `CvP Status` register. This register allows software to monitor the CvP status signals.

Table 9–15. CvP Status

Bits	Register Description	Reset Value	Access
[15:10]	Reserved.	0x00	RO
[9]	`PLD_CORE_READY`. From FPGA fabric. This status bit is provided for debug.	Variable	RO
[8]	`PLD_CLK_IN_USE`. From clock switch module to fabric. This status bit is provided for debug.	Variable	RO
[7]	`CVP_CONFIG_DONE`. Indicates that the FPGA control block has completed the device configuration via CvP and there were no errors.	Variable	RO
[6]	`CVP_HF_RATE_SEL`. Indicates if the FPGA control block interface to the Arria V GZ hard IP for PCI Express is operating half the normal frequency–62.5MHz, instead of full rate of 125MHz	Variable	RO
[5]	`USERMODE`. Indicates if the configurable FPGA fabric is in user mode.	Variable	RO
[4]	`CVP_EN`. Indicates if the FPGA control block has enabled CvP mode.	Variable	RO
[3]	`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
[2]	CVP_CONFIG_READY – reflects the value of this signal from the FPGA control block, checked by software during programming algorithm	Variable	RO
[1]	Reserved.	—	—
[0]	Reserved.	—	—

The following table defines the fields of the CvP Mode Control register which provides global control of the CvP operation.

Refer to *Configuration via Protocol (CvP) Implementation in Altera FPGAs User Guide* for more information about using CvP.

Table 9–16. CvP Mode Control

Bits	Register Description	Reset Value	Access
[31:16]	Reserved.	0x0000	RO
[15:8]	`CVP_NUMCLKS`. Specifies the number of CvP clock cycles required for every CvP data register write. Valid values are 0x00–0x3F, where 0x00 corresponds to 64 cycles, and 0x01-0x3F corresponds to 1 to 63 clock cycles. The upper bits are not used, but are included in this field because they belong to the same byte enable.	0x00	RW
[7:4]	Reserved.	0x0	RO
[2]	`CVP_FULLCONFIG`. Request that the FPGA control block reconfigure the entire FPGA including the Arria V GZ 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 HIP 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 route to the FPGA fabric.	1’b0	RW

The following table defines the `CvP Data` register. Programming software should write the configuration data to this register. Every write to this 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, this register 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.

Table 9–17. CvP Data Register

Bits	Register Description	Reset Value	Access
[31:0]	Configuration data to be transferred to the FPGA control block to configure the device.	0x00000000	RW

The following table defines the `CvP Programming Control` register. This register is written by the programming software to control CvP programming.

Refer to *Configuration via Protocol (CvP) Implementation in Altera FPGAs User Guide* for more information about using CvP.

Table 9–18. CvP Programming Control Register

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

The following table defines the fields of the `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 logic custom logic.

Table 9–19. Uncorrectable Internal Error Status Register

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

The following table defines the `Uncorrectable Internal Error Mask` register. This 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.

Table 9–20. Uncorrectable Internal Error Mask Register

Bits	Register Description	Reset Value	Access
[31:12]	Reserved.	1b'0	RO
[11]	Mask for RX buffer posted and completion overflow error.	1b'1	RWS
[10]	Reserved	1b'0	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'0	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

The following table defines the `Correctable Internal Error Status` register. This 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. It should only be used to observe behavior, not to drive logic custom logic.

Table 9–21. Correctable Internal Error Status Register

Bits	Register Description	Reset Value	Access
[31:6]	Reserved.	0	RO
[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

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

Table 9–22. Correctable Internal Error Mask Register

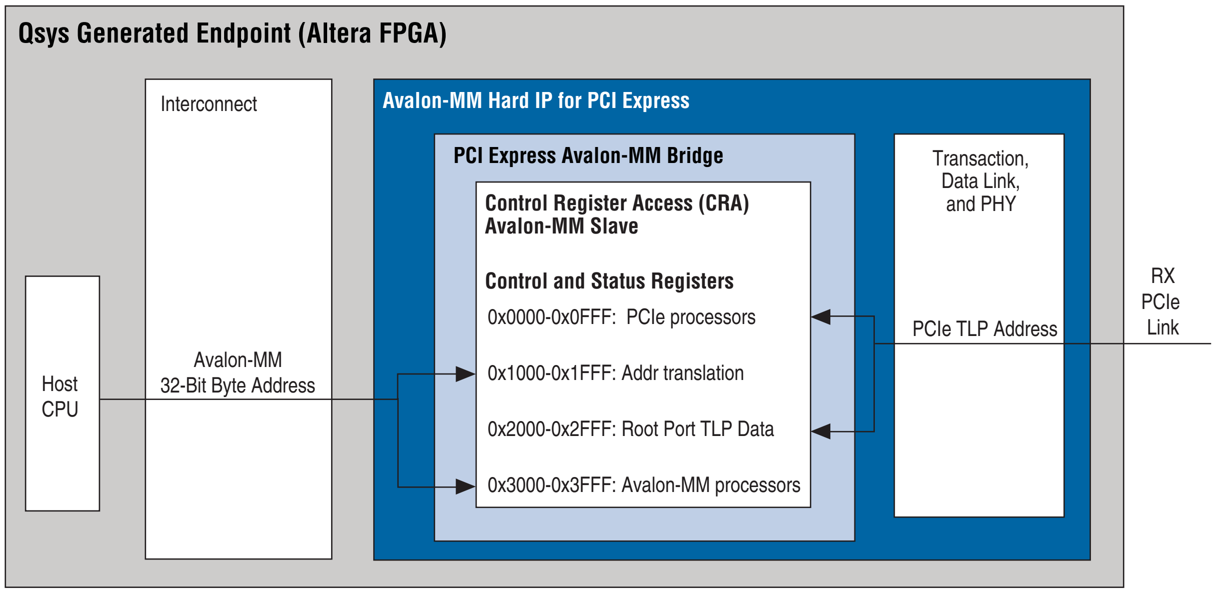
Bits	Register Description	Reset Value	Access
[31:7]	Reserved.	0	RO
[6]	Mask for Corrected Internal Error reported by the Application Layer.	1	RWS
[5]	Mask for configuration error detected in CvP mode.	0	RWS
[4:2]	Reserved.	0	RO
[1]	Mask for retry buffer correctable ECC error.	1	RWS
[0]	Mask for RX Buffer correctable ECC error.	1	RWS

PCI Express Avalon-MM Bridge Control Register Access Content

Control and status registers in the PCI Express Avalon-MM bridge are implemented in the CRA slave module. The control registers are accessible through the Avalon-MM slave port of the CRA slave module. This module is optional; however, you must include it to access the registers.

The control and status register address space is 16 KBytes. Each 4-KByte sub-region contains a set of functions, which may be specific to accesses from the PCI Express Root Complex only, from Avalon-MM processors only, or from both types of processors. Because all accesses come across the interconnect fabric—requests from the Avalon-MM Arria V GZ Hard IP for PCI Express are routed through the interconnect fabric—hardware does not enforce restrictions to limit individual processor access to specific regions. However, the regions are designed to enable straight-forward enforcement by processor software. Figure 9–1 illustrates accesses to the Avalon-MM control and status registers from the Host CPU and PCI Express link.

Figure 9–1. Accesses to the Avalon-MM Bridge Control and Status Registers


Qsys Generated Endpoint (Altera FPGA)
Interconnect
Avalon-MM
32-Bit Byte Address
Host
CPU
Avalon-MM Hard IP for PCI Express
PCI Express Avalon-MM Bridge
Control Register Access (CRA)
Avalon-MM Slave
Control and Status Registers
0x0000-0x0FFF: PCIe processors
0x1000-0x1FFF: Addr translation
0x2000-0x2FFF: Root Port TLP Data
0x3000-0x3FFF: Avalon-MM processors
Transaction,
Data Link,
and PHY
PCIe TLP Address
RX
PCIe
Link

The following table describes the four subregions.

Table 9–23. Avalon-MM Control and Status Register Address Spaces

Address Range	Address Space Usage
0x0000-0x0FFF	Registers typically intended for access by PCI Express processors only. This includes PCI Express interrupt enable controls, write access to the PCI Express Avalon-MM bridge mailbox registers, and read access to Avalon-MM-to-PCI Express mailbox registers.
0x1000-0x1FFF	Avalon-MM-to-PCI Express address translation tables. Depending on the system design these may be accessed by PCI Express processors, Avalon-MM processors, or both.
0x2000-0x2FFF	Root Port request registers. An embedded processor, such as the Nios II processor, programs these registers to send the data to send Configuration TLPs, I/O TLPs, single dword Memory Reads and Write request, and receive interrupts from an Endpoint.
0x3000-0x3FFF	Registers typically intended for access by Avalon-MM processors only. These include Avalon-MM interrupt enable controls, write access to the Avalon-MM-to-PCI Express mailbox registers, and read access to PCI Express Avalon-MM bridge mailbox registers.

The data returned for a read issued to any undefined address in this range is unpredictable.

The following table lists the complete address map for the PCI Express Avalon-MM bridge registers.

In the following table the text in green are links to the detailed register description

Table 9–24. PCI Express Avalon-MM Bridge Register Map

Address Range	Register
0x0040	Avalon-MM to PCI Express Interrupt Status Register 0x0040
0x0050	Avalon-MM to PCI Express Interrupt Enable Register 0x0050
0x0060	Avalon-MM Interrupt Vector Register 0x0060
0x0800–0x081F	PCI Express-to-Avalon-MM Mailbox Registers 0x0800–0x081F
0x0900–x091F	Avalon-MM-to-PCI Express Mailbox Registers 0x0900–0x091F
0x1000–0x1FFF	Avalon-MM-to-PCI Express Address Translation Table 0x1000–0x1FFF
0x2000–0x2FFF	Root Port TLP Data Registers 0x2000–0x2FFF
0x3060	Avalon-MM Interrupt Status Registers for Root Ports 0x3060
0x3060	PCI Express to Avalon-MM Interrupt Status Register for Endpoints 0x3060
0x3070	INT-X Interrupt Enable Register for Root Ports 0x3070
0x3070	INT-X Interrupt Enable Register for Endpoints 0x3070
0x3B00-0x3B1F	PCI Express to Avalon-MM Mailbox Registers 0x3B00–0x3B1F
0x3A00-0x3A1F	Avalon-MM to PCI Express Mailbox Registers 0x3A00–0x3A1F

Avalon-MM to PCI Express Interrupt Registers

The registers in this section contain status of various signals in the PCI Express Avalon-MM bridge logic and allow PCI Express interrupts to be asserted when enabled. Only Root Complexes should access these registers; however, hardware does not prevent other Avalon-MM masters from accessing them.

The following table shows the status of all conditions that can cause a PCI Express interrupt to be asserted.

Table 9–25. Avalon-MM to PCI Express Interrupt Status Register **0x0040**

Bit	Name	Access	Description
[31:24]	Reserved	—	—
[23]	`A2P_MAILBOX_INT7`	RW1C	1 when the A2P_MAILBOX7 is written to
[22]	`A2P_MAILBOX_INT6`	RW1C	1 when the A2P_MAILBOX6 is written to
[21]	`A2P_MAILBOX_INT5`	RW1C	1 when the A2P_MAILBOX5 is written to
[20]	`A2P_MAILBOX_INT4`	RW1C	1 when the A2P_MAILBOX4 is written to
[19]	`A2P_MAILBOX_INT3`	RW1C	1 when the A2P_MAILBOX3 is written to
[18]	`A2P_MAILBOX_INT2`	RW1C	1 when the A2P_MAILBOX2 is written to
[17]	`A2P_MAILBOX_INT1`	RW1C	1 when the A2P_MAILBOX1 is written to
[16]	`A2P_MAILBOX_INT0`	RW1C	1 when the A2P_MAILBOX0 is written to
[15:0]	`AVL_IRQ_ASSERTED[15:0]`	RO	Current value of the Avalon-MM interrupt (IRQ) input ports to the Avalon-MM RX master port: ■ 0 – Avalon-MM IRQ is not being signaled. ■ 1 – Avalon-MM IRQ is being signaled. A Qsys-generated IP Compiler for PCI Express has as many as 16 distinct IRQ input ports. Each `AVL_IRQ_ASSERTED[]` bit reflects the value on the corresponding IRQ input port.

A PCI Express interrupt can be asserted for any of the conditions registered in the `Avalon-MM to` PCI Express `Interrupt Status` register by setting the corresponding bits in the Avalon-MM-to-PCI Express `Interrupt Enable` register (Table 9–26). Either MSI or legacy interrupts can be generated as explained in the section "Enabling MSI or Legacy Interrupts" on page 13–7.

The following table describes the `Avalon-MM to PCI Express Interrupt Enable` register.

Table 9–26. Avalon-MM to PCI Express Interrupt Enable Register **0x0050**

Bits	Name	Access	Description
[31:24]	Reserved	—	—
[23:16]	`A2P_MB_IRQ`	RW	Enables generation of PCI Express interrupts when a specified mailbox is written to by an external Avalon-MM master.
[15:0]	`AVL_IRQ[15:0]`	RX	Enables generation of PCI Express interrupts when a specified Avalon-MM interrupt signal is asserted. Your Qsys system may have as many as 16 individual input interrupt signals.

The following table describes the `Avalon-MM Interrupt Vector` register.

Table 9–27. Avalon-MM Interrupt Vector Register **0x0060**

Bits	Name	Access	Description
[31:5]	Reserved	—	—
[4:0]	`AVALON_IRQ_VECTOR`	RO	Stores the interrupt vector of the system interconnect fabric. The host software should read this register after being interrupted and determine the servicing priority.

PCI Express Mailbox Registers

The PCI Express Root Complex typically requires write access to a set of PCI Express-to-Avalon-MM mailbox registers and read-only access to a set of Avalon-MM-to-PCI Express mailbox registers. Eight mailbox registers are available.

The `PCI Express-to-Avalon-MM Mailbox` registers are writable at the addresses shown in the following table. Writing to one of these registers causes the corresponding bit in the `Avalon-MM Interrupt Status` register to be set to a one.

Table 9–28. PCI Express-to-Avalon-MM Mailbox Registers **0x0800–0x081F**

Address	Name	Access	Description
0x0800	`P2A_MAILBOX0`	RW	PCI Express-to-Avalon-MM Mailbox 0
0x0804	`P2A_MAILBOX1`	RW	PCI Express-to-Avalon-MM Mailbox 1
0x0808	`P2A_MAILBOX2`	RW	PCI Express-to-Avalon-MM Mailbox 2
0x080C	`P2A_MAILBOX3`	RW	PCI Express-to-Avalon-MM Mailbox 3
0x0810	`P2A_MAILBOX4`	RW	PCI Express-to-Avalon-MM Mailbox 4
0x0814	`P2A_MAILBOX5`	RW	PCI Express-to-Avalon-MM Mailbox 5
0x0818	`P2A_MAILBOX6`	RW	PCI Express-to-Avalon-MM Mailbox 6
0x081C	`P2A_MAILBOX7`	RW	PCI Express-to-Avalon-MM Mailbox 7

The `Avalon-MM-to-PCI Express Mailbox` registers are read at the addresses shown in the following table. The PCI Express Root Complex should use these addresses to read the mailbox information after being signaled by the corresponding bits in the `Avalon-MM to PCI Express Interrupt Status` register.

Table 9–29. Avalon-MM-to-PCI Express Mailbox Registers **0x0900–0x091F**

Address	Name	Access	Description
0x0900	`A2P_MAILBOX0`	RO	Avalon-MM-to-PCI Express Mailbox 0
0x0904	`A2P_MAILBOX1`	RO	Avalon-MM-to-PCI Express Mailbox 1
0x0908	`A2P_MAILBOX2`	RO	Avalon-MM-to-PCI Express Mailbox 2
0x090C	`A2P_MAILBOX3`	RO	Avalon-MM-to-PCI Express Mailbox 3
0x0910	`A2P_MAILBOX4`	RO	Avalon-MM-to-PCI Express Mailbox 4
0x0914	`A2P_MAILBOX5`	RO	Avalon-MM-to-PCI Express Mailbox 5
0x0918	`A2P_MAILBOX6`	RO	Avalon-MM-to-PCI Express Mailbox 6
0x091C	`A2P_MAILBOX7`	RO	Avalon-MM-to-PCI Express Mailbox 7

Avalon-MM-to-PCI Express Address Translation Table

The Avalon-MM-to-PCI Express address translation table is writable using the CRA slave port. Each entry in the PCI Express address translation table is 8 bytes wide, regardless of the value in the current PCI Express address width parameter. Therefore, register addresses are always the same width, regardless of PCI Express address width.

Table 9–30. Avalon-MM-to-PCI Express Address Translation Table **0x1000–0x1FFF**

Address	Bits	Name	Access	Description
0x1000	[1:0]	`A2P_ADDR_SPACE0`	RW	Address space indication for entry 0. Refer to Table 9–31 for the definition of these bits.
	[31:2]	`A2P_ADDR_MAP_LO0`	RW	Lower bits of Avalon-MM-to-PCI Express address map entry 0.
0x1004	[31:0]	`A2P_ADDR_MAP_HI0`	RW	Upper bits of Avalon-MM-to-PCI Express address map entry 0.
0x1008	[1:0]	`A2P_ADDR_SPACE1`	RW	Address space indication for entry 1. Refer to Table 9–31 for the definition of these bits.
	[31:2]	`A2P_ADDR_MAP_LO1`	RW	Lower bits of Avalon-MM-to-PCI Express address map entry 1. This entry is only implemented if the number of address translation table entries is greater than 1.
0x100C	[31:0]	`A2P_ADDR_MAP_HI1`	RW	Upper bits of Avalon-MM-to-PCI Express address map entry 1. This entry is only implemented if the number of address translation table entries is greater than 1.

Note to Table 9–30:

(1) These table entries are repeated for each address specified in the **Number of address pages** parameter. If **Number of address pages** is set to the maximum of 512, 0x1FF8 contains A2P_ADDR_MAP_LO511 and 0x1FFC contains A2P_ADDR_MAP_HI511.

The format of the address space field (`A2P_ADDR_SPACEn`) of the address translation table entries is shown in the previous table.

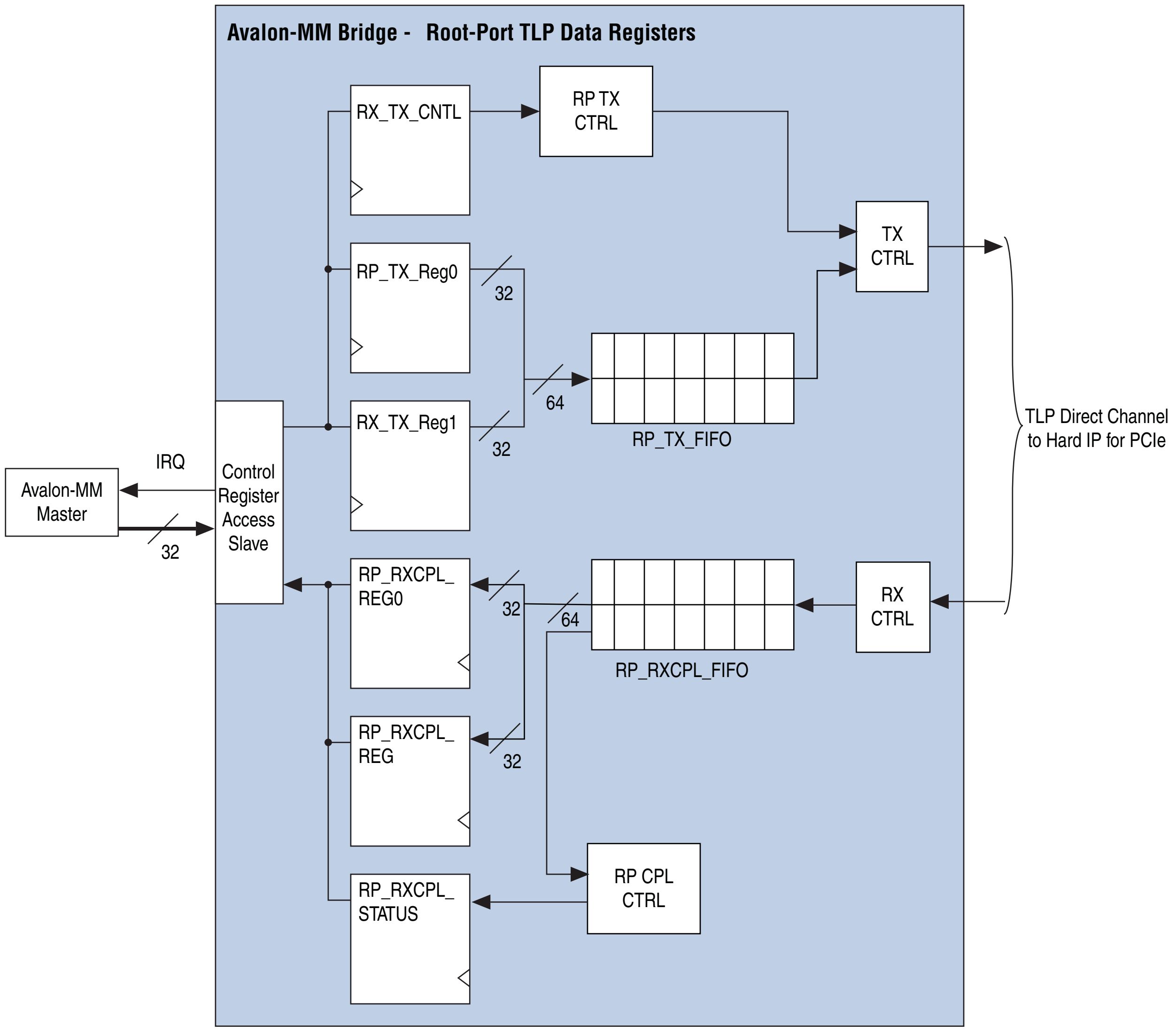
Table 9–31. PCI Express Avalon-MM Bridge Address Space Bit Encodings

Value (Bits 1:0)	Indication
00	Memory Space, 32-bit PCI Express address. 32-bit header is generated. Address bits 63:32 of the translation table entries are ignored.
01	Memory space, 64-bit PCI Express address. 64-bit address header is generated.
10	Reserved.
11	Reserved.

Root Port TLP Data Registers

The TLP data registers provide a mechanism for the Application Layer to specify data that the Root Port uses to construct Configuration TLPs, I/O TLPs, and single dword Memory Reads and Write requests. The Root Port then drives the TLPs on the TLP Direct Channel to access the Configuration Space, I/O space, or Endpoint memory. Figure 9–2 illustrates these registers.

Figure 9–2. Root Port TLP Data Registers



☞ The high performance TLPs implemented by Avalon-MM ports in the Avalon-MM Bridge are also available for Root Ports. For more information about these TLPs, refer to Avalon-MM Bridge TLPs. The following table describes the Root Port TLP data registers.

Table 9–32. Root Port TLP Data Registers **0x2000–0x2FFF**

Root-Port Request Registers				Address Range: 0x2800-0x2018
Address	**Bits**	**Name**	**Access**	**Description**
0x2000	[31:0]	`RP_TX_REG0`	W	Lower 32 bits of the TX TLP.
0x2004	[31:0]	`RP_TX_REG1`	W	Upper 32 bits of the TX TLP.
0x2008	[31:2]	Reserved	—	—
	[1]	`RP_TX_CNTRL.EOP`	W	Write 1’b1 to specify the of end a packet. Writing this bit frees the corresponding entry in the FIFO.
	[0]	`RP_TX_CNTRL.SOP`	W	Write 1’b1 to specify the start of a packet.
0x2010	[31:16]	Reserved	—	—
	[15:8]	`RP_RXCPL_STATUS`	R	Specifies the number of words in the RX completion FIFO that contain valid data.
	[7:2]	Reserved	—	—
	[1]	`RP_RXCPL_STATUS.EOP`	R	When 1’b1, indicates that the data for a Completion TLP is ready to be read by the Application Layer. The Application Layer must poll this bit to determine when a Completion TLP is available.
	[0]	`RP_RXCPL_STATUS.SOP`	R	When 1’b1, indicates that the final data for a Completion TLP is ready to be read by the Application Layer. The Application Layer must poll this bit to determine when the final data for a Completion TLP is available.
0x2014	[31:0]	`RP_RXCPL_REG1`	RC	Lower 32 bits of a Completion TLP. Reading frees this entry in the FIFO.
0x2018	[31:0]	`RP_RXCPL_REG1`	RC	Upper 32 bits of a Completion TLP. Reading frees this entry in the FIFO.

Programming Model for Avalon-MM Root Port

The Application Layer writes the Root Port TLP TX Data registers with TLP formatted data for Configuration Read and Write Requests, I/O Read and Write Requests, or single dword Memory Read and Write Requests. Software should check the Root Port `Link Status` register (offset 0x92) to ensure the Data Link Layer Link `Active` bit is set to 1'b1 before issuing a Configuration requests to downstream ports.

The Application Layer data must be in the appropriate TLP format with the data payload aligned to the TLP address. Aligning the payload data to the TLP address may result in the payload data being either aligned or unaligned to the qword. Figure 9–1 illustrates three dword TLPs with data that is aligned and unaligned to the qword.

Figure 9–1. Layout of Data with 3 DWord Headers

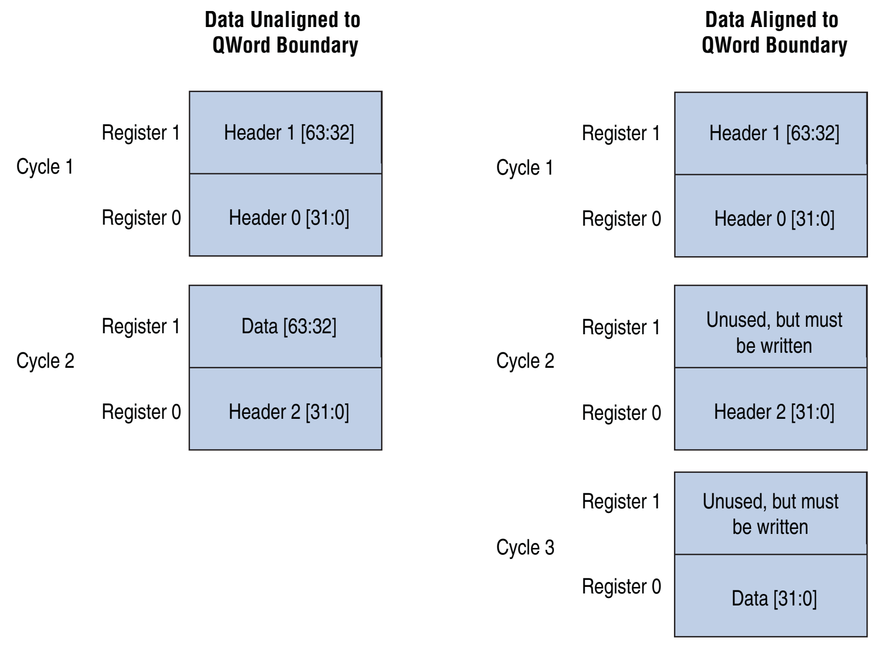
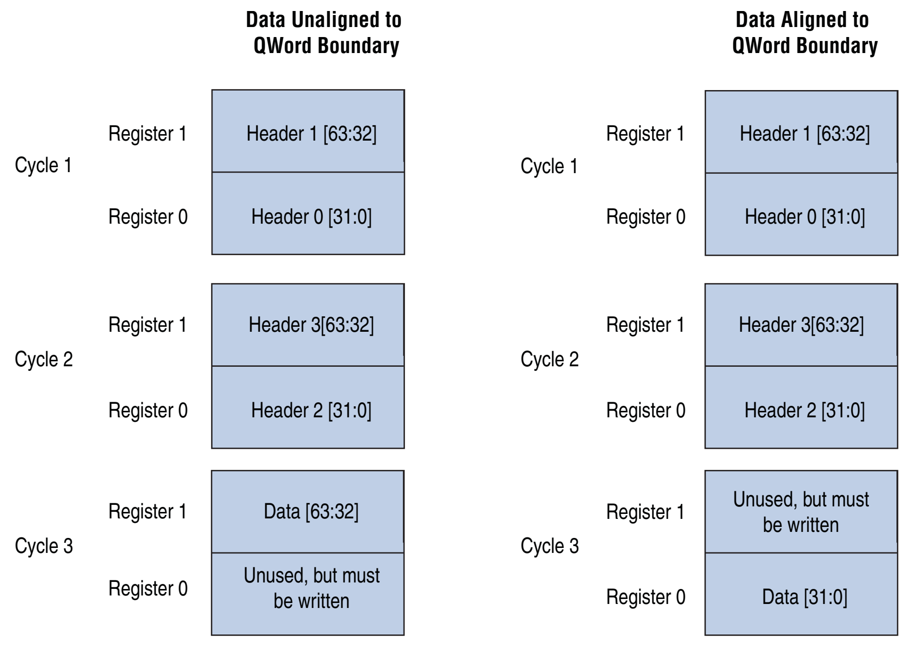


Figure 9–1 illustrates four dword TLPs with data that is aligned and unaligned to the qword.

Figure 9–2. Layout of Data with 4 DWord Headers



The TX TLP programming model scales with the data width. The Application Layer performs the same writes for both the 64- and 128-bit interfaces. The Application Layer can only have one outstanding non-posted request at a time. The Application Layer must use tags 16–31 to identify non-posted requests.

Sending a Write TLP

The Application Layer performs the following sequence of Avalon-MM accesses to the CRA slave port to send a Memory Write Request:

1. Write the first 32 bits of the TX TLP to `RP_TX_REG0`.
2. Write the next 32 bits of the TX TLP to `RP_TX_REG1`.
3. Write the `RP_TX_CNTRL.SOP` to 1′b1 to push the first two dwords of the TLP into the Root Port TX FIFO.
4. Repeat Steps 1 and 2. The second write to `RP_TX_REG1` is required, even for three dword TLPs with aligned data.
5. If the packet is complete write `RP_TX_CNTRL` to 2′b10 o indicate the end of the packet. If the packet is not complete write 2′b00 to `RP_TX_CNTRL`.
6. Repeat this sequence to program a complete TLP.

When the programming of the TX TLP is complete, the Avalon-MM Bridge schedules the TLP with higher priority than TX TLPs coming from the TX slave port.

Receiving a Completion TLP

The Completion TLPs associated with the Non-Posted TX requests are stored in the RP_RX_CPL FIFO buffer and subsequently loaded into RP_RXCPL registers. The Application Layer performs the following sequence to retrieve the TLP.

1. Polls the `RP_RXCPL_STATUS.SOP` to determine when it is set to 1′b1.
2. When `RP_RXCPL_STATUS.SOP` = 1′b′1, reads `RP_RXCPL_REG0` and `RP_RXCPL_REG1` to retrieve dword 0 and dword 1 of the Completion TLP.
3. Read the `RP_RXCPL_STATUS.EOP`.
   a. If `RP_RXCPL_STATUS.EOP =` 1′b0, read `RP_RXCPL_REG0` and `RP_RXCPL_REG1` to retrieve dword 2 and dword 3 of the Completion TLP, then repeat step 3.
   b. If `RP_RXCPL_STATUS.EOP =` 1′b1, read `RP_RXCPL_REG0` and `RP_RXCPL_REG1` to retrieve final dwords of TLP.

PCI Express to Avalon-MM Interrupt Status and Enable Registers for Root Ports

The Root Port supports MSI, MSI-X and legacy (INTx) interrupts. MSI and MSI-X interrupts are memory writes from the Endpoint to the Root Port. MSI and MSI-X requests are forwarded to the interconnect without asserting CraIrq_o.

The following table describes the `Interrupt Status` register for Root Ports. Refer to Table 9–35 for the definition of the `Interrupt Status` register for Endpoints.

Table 9–33. Avalon-MM Interrupt Status Registers for Root Ports (Part 1 of 2) **0x3060**

Bits	Name	Access Mode	Description
[31:5]	Reserved	—	—
[4]	`RPRX_CPL_RECEIVED`	RW1C	Set to 1’b1 when the Root Port has received a Completion TLP for an outstanding Non-Posted request from the TLP Direct channel.

Table 9–33. Avalon-MM Interrupt Status Registers for Root Ports (Part 2 of 2) 0x3060

Bits	Name	Access Mode	Description
[3]	`INTD_RECEIVED`	RW1C	The Root Port has received INTD from the Endpoint.
[2]	`INTC_RECEIVED`	RW1C	The Root Port has received INTC from the Endpoint.
[1]	`INTB_RECEIVED`	RW1C	The Root Port has received INTB from the Endpoint.
[0]	`INTA_RECEIVED`	RW1C	The Root Port has received INTA from the Endpoint.

The following table describes fields of the Avalon `Interrupt Enable` register for Root Ports.

Table 9–34. INT-X Interrupt Enable Register for Root Ports 0x3070

Bit	Name	Access Mode	Description
[31:5]	Reserved	—	—
[4]	`RPRX_CPL_RECEIVED`	RW	When set to 1'b1, enables the assertion of `CraIrq_o` when the Root Port Interrupt Status register `RPRX_CPL_RECEIVED` bit indicates it has received a Completion for a Non-Posted request from the TLP Direct channel.
[3]	`INTD_RECEIVED_ENA`	RW	When set to 1'b1, enables the assertion of `CraIrq_o` when the Root Port Interrupt Status register `INTD_RECEIVED` bit indicates it has received INTD.
[2]	`INTC_RECEIVED_ENA`	RW	When set to 1'b1, enables the assertion of `CraIrq_o` when the Root Port Interrupt Status register `INTC_RECEIVED` bit indicates it has received INTC.
[1]	`INTB_RECEIVED_ENA`	RW	When set to 1'b1, enables the assertion of `CraIrq_o` when the Root Port Interrupt Status register `INTB_RECEIVED` bit indicates it has received INTB.
[0]	`INTA_RECEIVED_ENA`	RW	When set to 1'b1, enables the assertion of `CraIrq_o` when the Root Port Interrupt Status register `INTA_RECEIVED` bit indicates it has received INTA.

PCI Express to Avalon-MM Interrupt Status and Enable Registers for Endpoints

The registers in this section contain status of various signals in the PCI Express Avalon-MM bridge logic and allow Avalon interrupts to be asserted when enabled. A processor local to the interconnect fabric that processes the Avalon-MM interrupts can access these registers.

☞ These registers must not be accessed by the PCI Express Avalon-MM bridge master ports; however, there is nothing in the hardware that prevents a PCI Express Avalon-MM bridge master port from accessing these registers.

The following table describes the Interrupt Status register when you configure the core as an Endpoint. It records the status of all conditions that can cause an Avalon-MM interrupt to be asserted. Refer to Table 9–33 for the definition of the Interrupt Status register when you configure the core as a Root Port.

Table 9–35. PCI Express to Avalon-MM Interrupt Status Register for Endpoints **0x3060**

Bits	Name	Access	Description
0	`ERR_PCI_WRITE_ FAILURE`	RW1C	When set to 1, indicates a PCI Express write failure. This bit can also be cleared by writing a 1 to the same bit in the `Avalon-MM to PCI Express Interrupt Status` register.
1	`ERR_PCI_READ_ FAILURE`	RW1C	When set to 1, indicates the failure of a PCI Express read. This bit can also be cleared by writing a 1 to the same bit in the `Avalon-MM to PCI Express Interrupt Status` register.
[15:2]	Reserved	—	—
[16]	`P2A_MAILBOX_INT0`	RW1C	1 when the P2A_MAILBOX0 is written
[17]	`P2A_MAILBOX_INT1`	RW1C	1 when the P2A_MAILBOX1 is written
[18]	`P2A_MAILBOX_INT2`	RW1C	1 when the P2A_MAILBOX2 is written
[19]	`P2A_MAILBOX_INT3`	RW1C	1 when the P2A_MAILBOX3 is written
[20]	`P2A_MAILBOX_INT4`	RW1C	1 when the P2A_MAILBOX4 is written
[21]	`P2A_MAILBOX_INT5`	RW1C	1 when the P2A_MAILBOX5 is written
[22]	`P2A_MAILBOX_INT6`	RW1C	1 when the P2A_MAILBOX6 is written
[23]	`P2A_MAILBOX_INT7`	RW1C	1 when the P2A_MAILBOX7 is written
[31:24]	Reserved	—	—

An Avalon-MM interrupt can be asserted for any of the conditions noted in the `Avalon-MM Interrupt Status` register by setting the corresponding bits in the `PCI Express to Avalon-MM Interrupt Enable` register.

PCI Express interrupts can also be enabled for all of the error conditions described. However, it is likely that only one of the Avalon-MM or PCI Express interrupts can be enabled for any given bit because typically a single process in either the PCI Express or Avalon-MM domain is responsible for handling the condition reported by the interrupt.

Table 9–36. INT-X Interrupt Enable Register for Endpoints **0x3070**

Bits	Name	Access	Description
[31:0]	`PCI Express to Avalon-MM Interrupt Enable`	RW	When set to 1, enables the interrupt for the corresponding bit in the `PCI Express to Avalon-MM Interrupt Status` register to cause the Avalon Interrupt signal (`cra_Irq_o`) to be asserted. Only bits implemented in the `PCI Express to Avalon-MM Interrupt Status` register are implemented in the Enable register. Reserved bits cannot be set to a 1.

Avalon-MM Mailbox Registers

A processor local to the interconnect fabric typically requires write access to a set of `Avalon-MM-to-PCI Express Mailbox` registers and read-only access to a set of `PCI Express-to-Avalon-MM Mailbox` registers. Eight mailbox registers are available.

The `Avalon-MM-to-PCI Express Mailbox` registers are writable at the addresses shown in the following table. When the Avalon-MM processor writes to one of these registers the corresponding bit in the `Avalon-MM to PCI Express Interrupt Status` register is set to 1.

Table 9–37. Avalon-MM to PCI Express Mailbox Registers **0x3A00–0x3A1F**

Address	Name	Access	Description
0x3A00	`A2P_MAILBOX0`	RW	Avalon-MM-to-PCI Express mailbox 0
0x3A04	`A2P_MAILBOX1`	RW	Avalon-MM-to-PCI Express mailbox 1
0x3A08	`A2P _MAILBOX2`	RW	Avalon-MM-to-PCI Express mailbox 2
0x3A0C	`A2P _MAILBOX3`	RW	Avalon-MM-to-PCI Express mailbox 3
0x3A10	`A2P _MAILBOX4`	RW	Avalon-MM-to-PCI Express mailbox 4
0x3A14	`A2P _MAILBOX5`	RW	Avalon-MM-to-PCI Express mailbox 5
0x3A18	`A2P _MAILBOX6`	RW	Avalon-MM-to-PCI Express mailbox 6
0x3A1C	`A2P_MAILBOX7`	RW	Avalon-MM-to-PCI Express mailbox 7

The `PCI Express-to-Avalon-MM Mailbox` registers are read-only at the addresses shown in the following table. The Avalon-MM processor reads these registers when the corresponding bit in the `PCI Express to Avalon-MM Interrupt Status` register is set to 1.

Table 9–38. PCI Express to Avalon-MM Mailbox Registers **0x3B00–0x3B1F**

Address	Name	Access Mode	Description
0x3B00	`P2A_MAILBOX0`	RO	PCI Express-to-Avalon-MM mailbox 0
0x3B04	`P2A_MAILBOX1`	RO	PCI Express-to-Avalon-MM mailbox 1
0x3B08	`P2A_MAILBOX2`	RO	PCI Express-to-Avalon-MM mailbox 2
0x3B0C	`P2A_MAILBOX3`	RO	PCI Express-to-Avalon-MM mailbox 3
0x3B10	`P2A_MAILBOX4`	RO	PCI Express-to-Avalon-MM mailbox 4
0x3B14	`P2A_MAILBOX5`	RO	PCI Express-to-Avalon-MM mailbox 5
0x3B18	`P2A_MAILBOX6`	RO	PCI Express-to-Avalon-MM mailbox 6
0x3B1C	`P2A_MAILBOX7`	RO	PCI Express-to-Avalon-MM mailbox 7

Correspondence between Configuration Space Registers and the PCIe Specification

The following table provides a comprehensive correspondence between the Configuration Space registers and their descriptions in the *PCI Express Base Specification 2.0 and 3.0.*

Table 9–39. Correspondence Configuration Space Registers and PCI Express Base Specification Rev. 2.0 Description

Byte Address	Hard IP Configuration Space Register	Corresponding Section in PCIe Specification
Table 6-1. PCI Configuration Space **PCI Configuration Space**		
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

Table 9–39. Correspondence Configuration Space Registers and PCI Express Base Specification Rev. 2.0 Description

Byte Address	Hard IP Configuration Space Register	Corresponding Section in PCIe Specification
0x040:0x04C	Reserved	
0x050:0x05C	MSI Capability Structure	MSI and MSI-X Capability Structures
0x068:0x070	MSI Capability Structure	MSI and MSI-X Capability Structures
0x070:0x074	Reserved	
0x078:0x07C	Power Management Capability Structure	PCI Power Management Capability Structure
0x080:0x0B8	PCI Express Capability Structure	PCI Express Capability Structure
0x0B8:0x0FC	Reserved	
0x094:0x0FF	Root Port	
0x100:0x16C	Virtual Channel Capability Structure (Reserved)	Virtual Channel Capability
0x170:0x17C	Reserved	
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	
Table 6-2. PCI Type 0 Configuration Space Header (Endpoints), Rev3.0 Spec: Type 0 Configuration Space Header		
0x000	Device ID Vendor ID	Type 0 Configuration Space Header
0x004	Status Command	Type 0 Configuration Space Header
0x008	Class Code Revision ID	Type 0 Configuration Space Header
0x00C	0x00 Header Type 0x00 Cache Line Size	Type 0 Configuration Space Header
0x010	Base Address 0	Base Address Registers (Offset 10h - 24h)
0x014	Base Address 1	Base Address Registers (Offset 10h - 24h)
0x018	Base Address 2	Base Address Registers (Offset 10h - 24h)
0x01C	Base Address 3	Base Address Registers (Offset 10h - 24h)
0x020	Base Address 4	Base Address Registers (Offset 10h - 24h)
0x024	Base Address 5	Base Address Registers (Offset 10h - 24h)
0x028	Reserved	Type 0 Configuration Space Header
0x02C	Subsystem Device ID Subsystem Vendor ID	Type 0 Configuration Space Header
0x030	Expansion ROM base address	Type 0 Configuration Space Header
0x034	Reserved Capabilities PTR	Type 0 Configuration Space Header
0x038	Reserved	Type 0 Configuration Space Header
0x03C	0x00 0x00 Interrupt Pin Interrupt Line	Type 0 Configuration Space Header

Table 9–39. Correspondence Configuration Space Registers and PCI Express Base Specification Rev. 2.0 Description

Byte Address	Hard IP Configuration Space Register	Corresponding Section in PCIe Specification
Table 6-3. PCI Type 1 Configuration Space Header (Root Ports) Rev3.0 Spec: Type 1 Configuration Space Header		
0x000	Device ID Vendor ID	Type 1 Configuration Space Header
0x004	Status Command	Type 1 Configuration Space Header
0x008	Class Code Revision ID	Type 1 Configuration Space Header
0x00C	BIST Header Type Primary Latency Timer Cache Line Size	Type 1 Configuration Space Header
0x010	Base Address 0	Base Address Registers (Offset 10h/14h)
0x014	Base Address 1	Base Address Registers (Offset 10h/14h)
0x018	Secondary Latency Timer Subordinate Bus Number Secondary Bus Number Primary Bus Number	Secondary Latency Timer (Offset 1Bh)/Type 1 Configuration Space Header/ /Primary Bus Number (Offset 18h)
0x01C	Secondary Status I/O Limit I/O Base	Secondary Status Register (Offset 1Eh) / Type 1 Configuration Space Header
0x020	Memory Limit Memory Base	Type 1 Configuration Space Header
0x024	Prefetchable Memory Limit Prefetchable Memory Base	Prefetchable Memory Base/Limit (Offset 24h)
0x028	Prefetchable Base Upper 32 Bits	Type 1 Configuration Space Header
0x02C	Prefetchable Limit Upper 32 Bits	Type 1 Configuration Space Header
0x030	I/O Limit Upper 16 Bits I/O Base Upper 16 Bits	Type 1 Configuration Space Header
0x034	Reserved Capabilities PTR	Type 1 Configuration Space Header
0x038	Expansion ROM Base Address	Type 1 Configuration Space Header
0x03C	Bridge Control Interrupt Pin Interrupt Line	Bridge Control Register (Offset 3Eh)
Table 6-4.MSI Capability Structure, Rev3.0 Spec: MSI Capability Structures		
0x050	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
Table 6-5. MSI-X Capability Structure, Rev3.0 Spec: MSI-X Capability Structures		
0x68	Message Control Next Cap Ptr Capability ID	MSI and MSI-X Capability Structures
0x6C	MSI-X Table Offset BIR	MSI and MSI-X Capability Structures
0x70	Pending Bit Array (PBA) Offset BIR	MSI and MSI-X Capability Structures
Table 6-6. Power Management Capability Structure, Rev3.0 Spec		
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
Table 6-7 PCI Express AER Capability Structure, Rev3.0 Spec: AER Capability		
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

Table 9–39. Correspondence Configuration Space Registers and PCI Express Base Specification Rev. 2.0 Description

Byte Address	**Hard IP Configuration Space Register**	**Corresponding Section in PCIe Specification**
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

10. Reset and Clocks

This chapter covers the functional aspects of the reset and clock circuitry for the Arria V GZ Hard IP for PCI Express. It includes the following sections:

- Reset
- Clocks

For descriptions of the available reset and clock *signals* refer to "Reset Signals, Status, and Link Training Signals" on page 8–28 and "Clock Signals" on page 8–28.

Reset

Arria V GZ includes two reset controllers. One reset controller is implemented in soft logic. A second reset controller is implemented in hard logic. Software selects the appropriate reset controller depending on the configuration you specify. Both reset controllers reset the Arria V GZ Hard IP for PCI Express IP Core and provide sample reset logic in the example design. Figure 10–1 on page 10–2 provides a simplified view of the logic that implements both reset controllers. Table 10–1 summarizes their functionality.

Table 10–1. Use of Hard and Soft Reset Controllers

Reset Controller Used	Description
Hard Reset Controller	`pin_perst` from the input pin of the FPGA resets the Hard IP for PCI Express IP Core. `app_rstn` which resets the Application Layer logic is derived from `reset_status` and `pld_clk_inuse`, which are outputs of the core. This reset controller is supported for Gen 1 and Gen2 production devices.
Soft Reset Controller	Either `pin_perst` from the input pin of the FPGA or `npor` which is derived from `pin_perst` or `local_rstn` can reset the Hard IP for PCI Express IP Core. Application Layer logic generates the optional `local_rstn` signal. `app_rstn` which resets the Application Layer logic is derived from `npor`. This reset controller is supported for Gen3 production devices.

Your Application Layer could instantiate a module similar to **altpcie_rs_hip.v** as shown in Figure 10–1 on page 10–2 to generate `app_rstn` which resets the Application Layer logic.

Figure 10–1. Reset Controller in Arria V GZ Devices

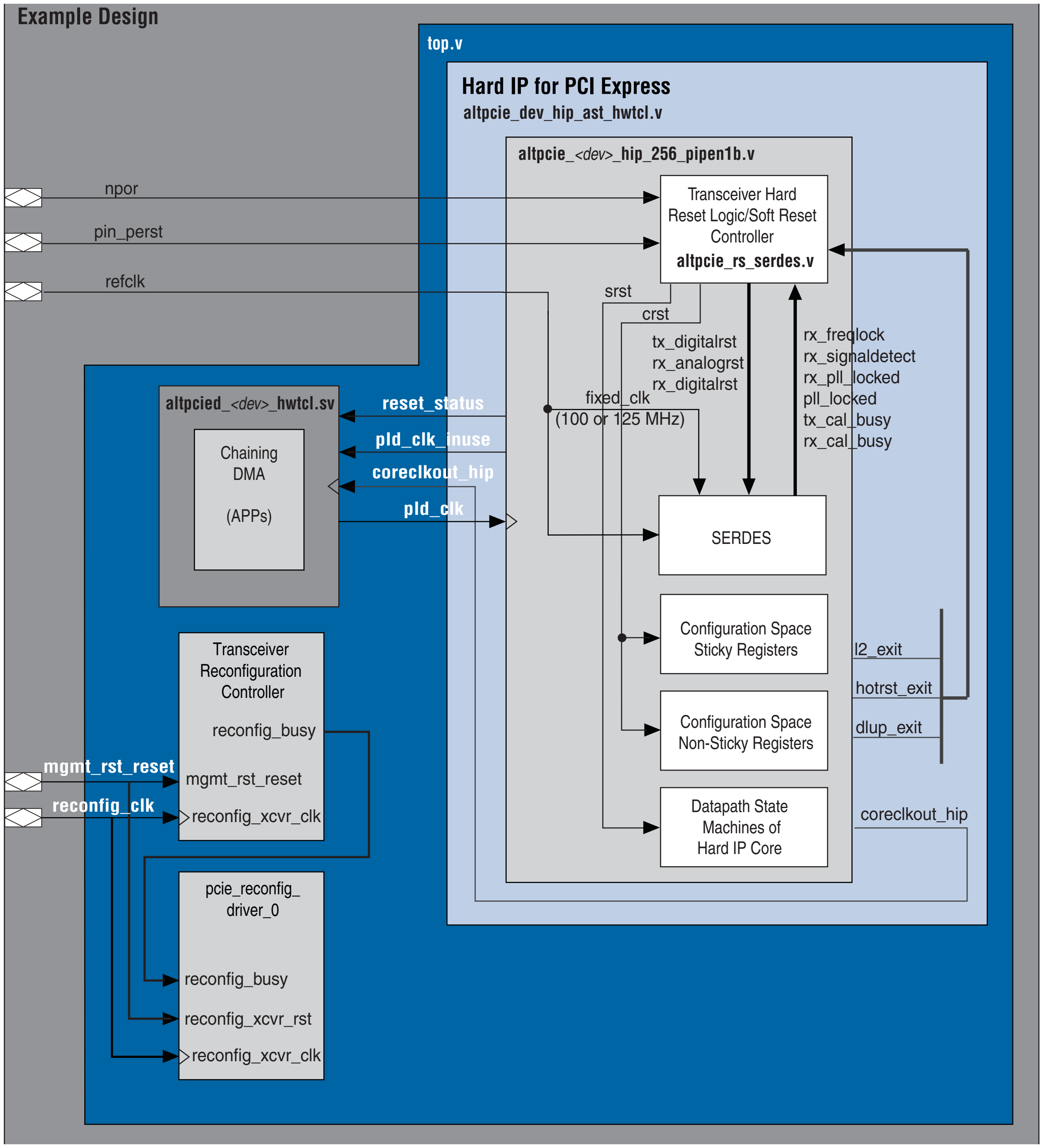
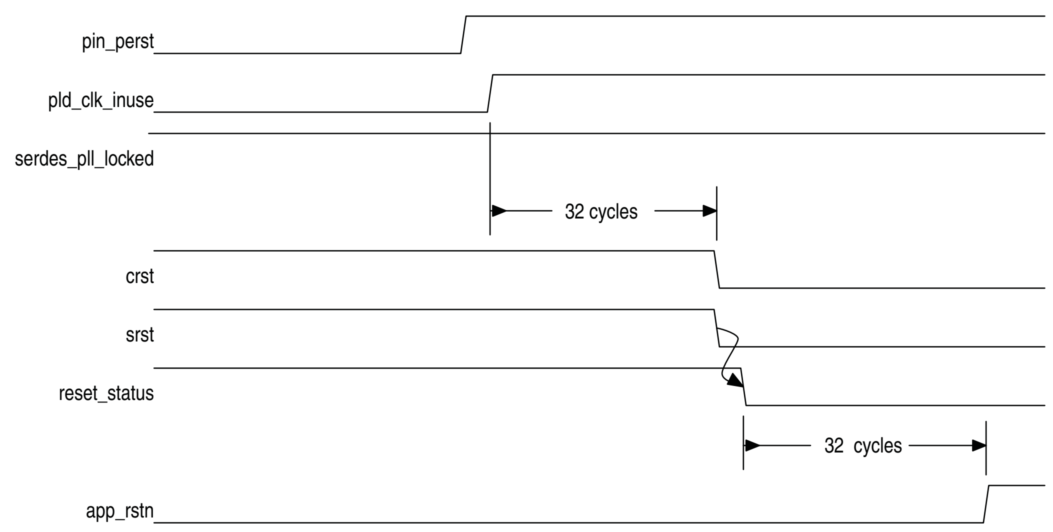


Figure 10–2 illustrates the reset sequence for the Hard IP for PCI Express IP core and the Application Layer logic.

Figure 10–2. Hard IP for PCI Express and Application Logic Reset Sequence

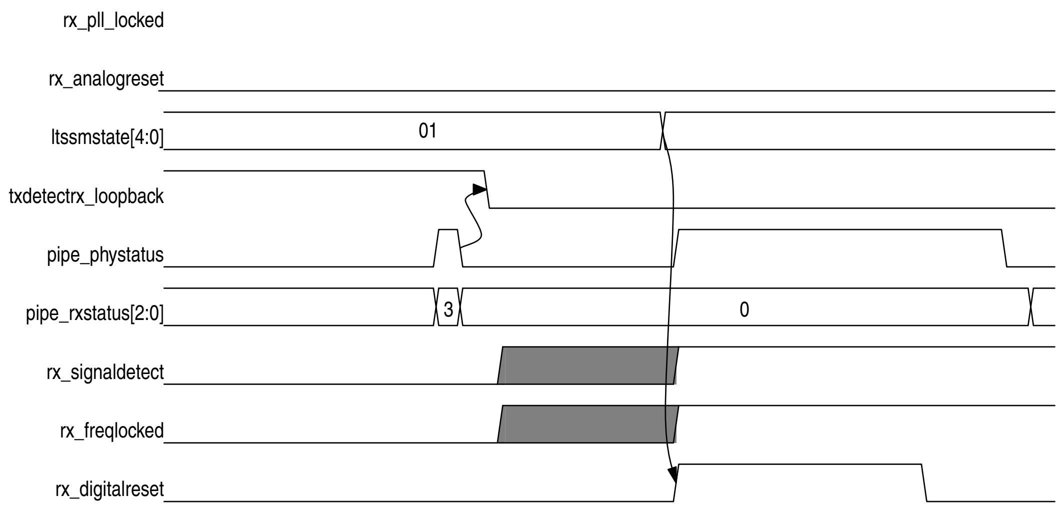


As Figure 10–2 illustrates, this reset sequence includes the following steps:

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 cycles after `reset_status` is released.

Figure 10–3 illustrates the RX transceiver reset sequence.

Figure 10–3. RX Transceiver Reset Sequence

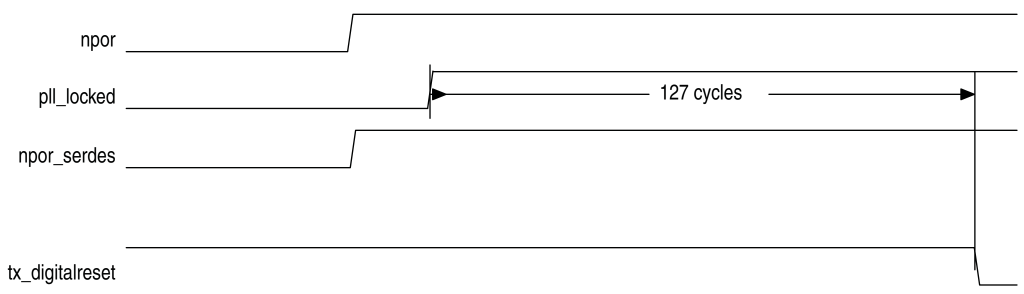


As Figure 10–3 illustrates, the RX transceiver reset includes the following steps:

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 10–4 illustrates the TX transceiver reset sequence.

Figure 10–4. TX Transceiver Reset Sequence



As Figure 10–4 illustrates, the RX transceiver reset includes the following steps:

1. After `npor` is deasserted, the 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 cycles before deasserting `tx_digitalreset.`

Clocks

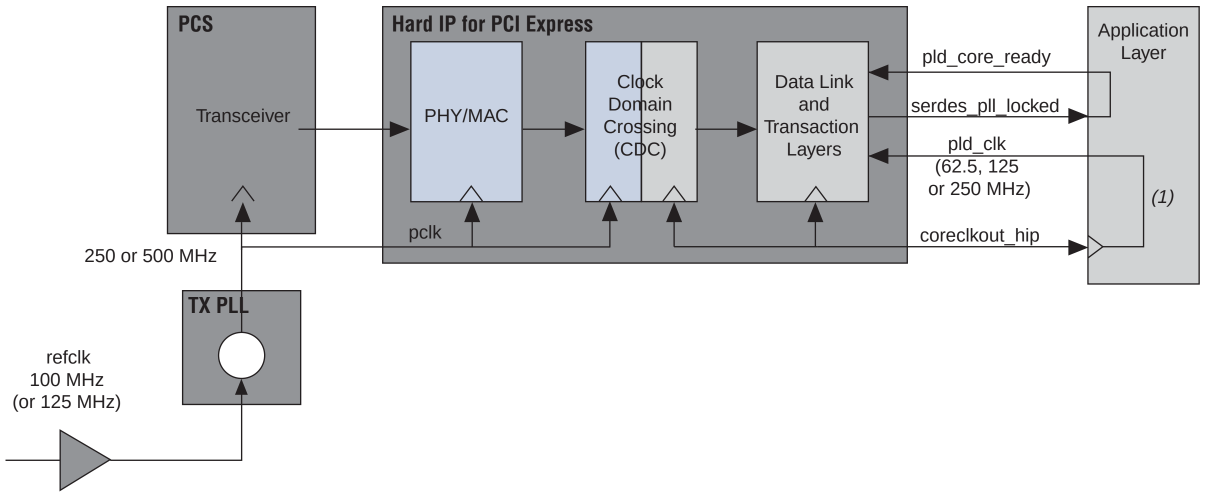
The Hard IP contains a clock domain crossing (CDC) synchronizer at the interface between the PHY/MAC and the DLL layers which allows the Data Link and Transaction Layers to run at frequencies independent of the PHY/MAC and provides more flexibility for the user clock interface. Depending on parameters you specify, the core selects the appropriate `coreclkout_hip` as listed in Table 10–1 on page 10–1. 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 2.1*, you must provide a 100 MHz reference clock that is connected directly to the transceiver. As a convenience, you may also use a 125 MHz input reference clock as input to the TX PLL.

Arria V GZ Hard IP for PCI Express Clock Domains

Figure 10–5 illustrates the clock domains when using coreclkout_hip to drive the Application Layer and the `pld_clk` of the Arria V GZHard IP for PCI Express IP Core.

Figure 10–5. Clock Domains and Clock Generation for the Application Layer



Note to Figure 10–5:

(1) The Example Design connects `coreclkout_hip` to the `pld_clk`. However, this connection is not mandatory.

As Figure 10–5 indicates, the IP core includes three clock domains.

- pclk
- coreclkout
- pld_clk

pclk

The transceiver derives `pclk` from the 100 MHz `refclk` signal that you must provide to the device. The *PCI Express Base Specification 2.1* requires that the `refclk` signal frequency be 100 MHz ±300 PPM; however, as a convenience, you can also use a reference clock that is 125 MHz ±300 PPM.

The transitions between Gen1, Gen2, and Gen3 should be glitchless. `pclk` can be turned off for most of the 1 ms timeout assigned for the PHY to change the clock rate; however, `pclk` should be stable before the 1 ms timeout expires.

Table 10–2 shows the frequency of `pclk` for Gen1, Gen2, and Gen3 variants.

Table 10–2. pclk Clock Frequency

Data Rate	Frequency
Gen1	250 MHz
Gen2	500 MHz
Gen3	250 MHz

The CDC module implements the asynchronous clock domain crossing between the PHY/MAC `pclk` domain and the Data Link Layer `coreclk` domain. The transceiver `pclk` clock is connected directly to the Hard IP for PCI Express and does not connect to the FPGA fabric.

coreclkout

The `coreclkout` signal is derived from `pclk`. Table 10–3 lists frequencies for `coreclkout`, which are a function of the link width, data rate, and the width of the Avalon-ST bus.

The frequencies and widths specified in Table 10–3 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 Table 10–3. (The Hard IP throttles the interface to achieve a lower throughput.)

Table 10–3. coreclkout_hip Values for All Parameterizations

Link Width	Max Link Rate	Avalon Interface Width (1)	coreclkout_hip
×1	Gen1	64	125 MHz
×1	Gen1	64	62.5 MHz (2)
×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	250 MHz
×2	Gen3	128	125 MHz
×4	Gen3	128	250 MHz
×4	Gen3	256	125 MHz
×8	Gen3	256	250 MHz

Notes to Table 10–3:

(1) The 256-bit interface is only available for the Avalon-ST interface.

(2) This mode saves power.

pld_clk

`coreclkout_hip` can drive the Application Layer clock along with the `pld_clk` input to the Arria V GZ Hard IP for PCI Express 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.

For Gen3, Altera recommends using a common reference clock (0 ppm) because when using separate reference clocks (non 0 ppm), the PCS occasionally must insert SKP symbols, potentially causing the PCIe link to go to recovery. Arria V GZ PCIe Hard IP in Gen1 or Gen2 modes are not affected by this issue. Systems using the common reference clock (0 ppm) are not affected by this issue. The primary repercussion of this issue is a slight decrease in bandwidth. On Gen3 x8 systems, this bandwidth impact is negligible. If non 0 ppm mode is required, so that separate reference clocks are used, please contact Altera for further information and guidance.

Additional Clocks

Designs that include the Arria V GZ Hard IP for PCI Express may require the following additional clocks:

hip_reconfig_clk

The frequency range for this clock is 50–125 MHz. This is the clock signal for the Hard IP reconfiguration interface. You can use this interface to change the value of global configuration registers that are read-only at run time. Use of the reconfiguration interface is optional.

reconfig_xcvr_clk

This is a free running clock with a frequency range of 100–125 MHz. This is the clock input to the Transceiver Reconfiguration Controller which performs the transceiver PHY reconfiguration functions required by Gen2 and Gen3 designs. For more information, refer to "Transceiver PHY IP Reconfiguration" on page 17–8.

Clock Summary

Table 10–4 summarizes the clocks for designs that include the Arria V GZ Hard IP for PCI Express IP Core.

Table 10–4. Required Clocks

Name	Frequency	Clock Domain
Clock Used by the Arria V GZ Hard IP for PCI Express IP Core		
`coreclkout_hip`	125, or 250 MHz	Avalon-ST interface between the Transaction and Application Layers.
`pld_clk`	62.5, 125 MHz, or 250 MHz	Application and Transaction Layers.
`refclk`	100 or 125 MHz	SERDES (transceiver). Dedicated free running input clock to the SERDES block.
Other Clocks that May Be Required for PCI Express Designs		
`reconfig_xcvr_clk`	100 –125 MHz	Transceiver Reconfiguration Controller.
`hip_reconfig_clk`	50–125 MHz	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.

11. Transaction Layer Protocol (TLP) Details

This chapter provides detailed information about the Arria V GZ Hard IP for PCI Express TLP handling. It includes the following sections:

- Supported Message Types
- Transaction Layer Routing Rules
- Receive Buffer Reordering

Supported Message Types

Table 11–1 describes the message types supported by the Hard IP.

Table 11–1. Supported Message Types (Part 1 of 3)

Message	Root Port	Endpoint	Generated by			Comments
			App Layer	Core	Core (with App Layer input)	
INTX Mechanism Messages						For Endpoints, only INTA messages are generated.
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	
Power Management Messages						
PM_Active_State_Nak	Transmit	Receive	No	Yes	No	
PM_PME	Receive	Transmit	No	No	Yes	
PME_Turn_Off	Transmit	Receive	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	Receive	Transmit	No	No	Yes	

Table 11–1. Supported Message Types (Part 2 of 3)

Message	Root Port	Endpoint	Generated by			Comments
			App Layer	Core	Core (with App Layer input)	
Error Signaling Messages						
ERR_COR	Receive	Transmit	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	Receive	Transmit	No	Yes	No	
ERR_FATAL	Receive	Transmit	No	Yes	No	
Locked Transaction Message						
Unlock Message	Transmit	Receive	Yes	No	No	
Slot Power Limit Message						
Set Slot Power Limit (1)	Transmit	Receive	No	Yes	No	In Root Port mode, through software. (1)
Vendor-defined Messages						
Vendor Defined Type 0	Transmit Receive	Transmit Receive	Yes	No	No	
Vendor Defined Type 1	Transmit Receive	Transmit Receive	Yes	No	No	

Table 11–1. Supported Message Types (Part 3 of 3)

Message	Root Port	Endpoint	Generated by			Comments
			App Layer	Core	Core (with App Layer input)	
Hot Plug Messages						
Attention_indicator On	Transmit	Receive	No	Yes	No	As per the recommendations in the *PCI Express Base Specification Revision 2.1*, 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 (2)	Receive	Transmit	No	No	Yes	

Notes to Table 11–1:

(1) In the *PCI Express Base Specification Revision 2.1*, this message is no longer mandatory after link training.

(2) In Endpoint mode.

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 V GZ 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 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 is dropped.

- For memory read and write request with addresses below 4 GBytes, requestors must use the 32-bit format. The Transaction Layer interprets requests using the 64-bit format for addresses below 4 GBytes 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 "Errors Detected by the Transaction Layer" on page 15–3 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 on routing rules in Root Port mode, refer to "Section 7.3.3 Configuration Request Routing Rules" in the *PCI Express Base Specification 3.0.*

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 TLP 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 formats shown in Chapter A, Transaction Layer Packet (TLP) Header Formats. If this bit is set, relaxed ordering is permitted. If software can guarantee that no dependencies exist between pending transactions, it is safe to 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.

Table 11–2 lists the transaction ordering rules.

Table 11–2. Transaction Ordering Rules (1)– (9)

Can the Row Pass the Column?		Posted Request		Non Posted Request				Completion	
		Memory Write or Message Request		Read Request		I/O or Cfg Write Request			
		Spec (10)	Hard IP	Spec	Hard IP	Spec	Hard IP	Spec	Hard IP
Posted	Posted Request	No Y/N (12)	No (11) No (12)	Yes	Yes	Yes	Yes	Y/N (11) Yes (12)	No (11) No (12)
NonPosted	Read Request	No	No	Y/N	No (11)	Y/N	No (12)	Y/N	No
NonPosted	Non-Posted Req with data	No	No	Y/N	No (13)	Y/N	No (14)	Y/N	No
Completion	Completion	No (11) Y/N (12)	No (11) No (12)	Yes	Yes	Yes	Yes	Y/N (11) No (12)	No (11) No
Completion	I/O or Configuration Write Completion	Y/N	No	Yes	Yes	Yes	Yes	Y/N	No

Notes to Table 11–2:

(1) 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.

(2) 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.

(3) Endpoints, Switches, and Root Complex may allow Memory Write and Message Requests to pass Completions or be blocked by Completions.

(4) Memory Write and Message Requests can pass Completions traveling in the PCI Express to PCI directions to avoid deadlock.

(5) If the Relaxed Ordering attribute is not set, then a Read Completion cannot pass a previously enqueued Memory Write or Message Request.

(6) If the Relaxed Ordering attribute is set, then a Read Completion is permitted to pass a previously enqueued Memory Write or Message Request.

(7) Read Completion associated with different Read Requests are allowed to be blocked by or to pass each other.

(8) Read Completions for Request (same Transaction ID) must return in address order.

(9) Non-posted requests cannot pass other non-posted requests.

(10) Refers to the *PCI Express Base Specification 3.0.*

(11) `CfgRd0` can pass `IORd` or `MRd`.

(12) `CfgWr0` can `IORd` or `MRd`.

(13) `CfgRd0` can pass `IORd` or `MRd`.

(14) `CfrWr0` can pass `IOWr`.

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 request transactions made to the Application Layer interface so that only posted and completion transactions are presented to the Application Layer.

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.

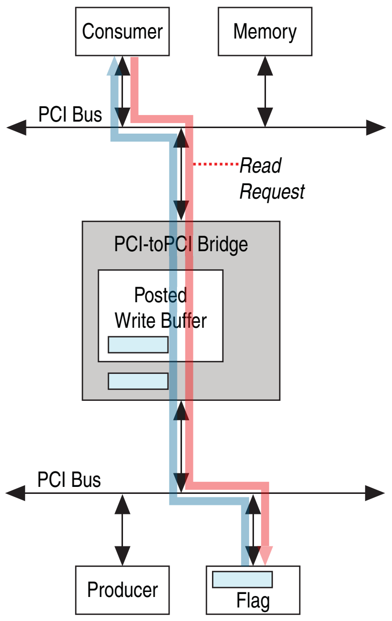
Using Relaxed Ordering

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 consumer must read a flag 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 11–1. Design Including Legacy PCI Buses Requiring Strong Ordering

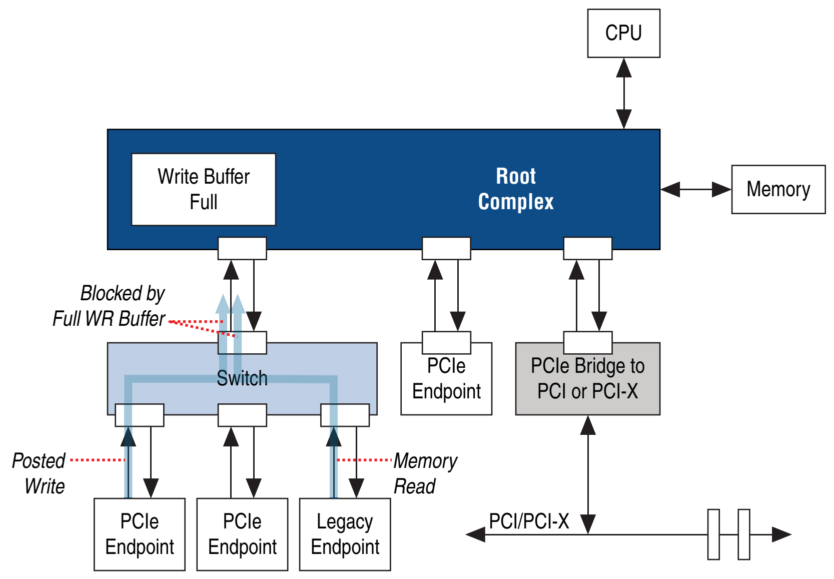


   b. A shared memory architecture where more than one thread accesses the same locations in memory.

      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.

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, failing to enable relaxed ordering blocks a memory read to the Legacy Endpoint because an earlier posted write cannot complete because a write buffer is full.

Figure 11–2. PCI Express Design Using Relaxed Ordering



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

12. Optional Features

This chapter provides information on several additional topics. It includes the following sections:

- Configuration via Protocol (CvP)
- ECRC
- Lane Initialization and Reversal

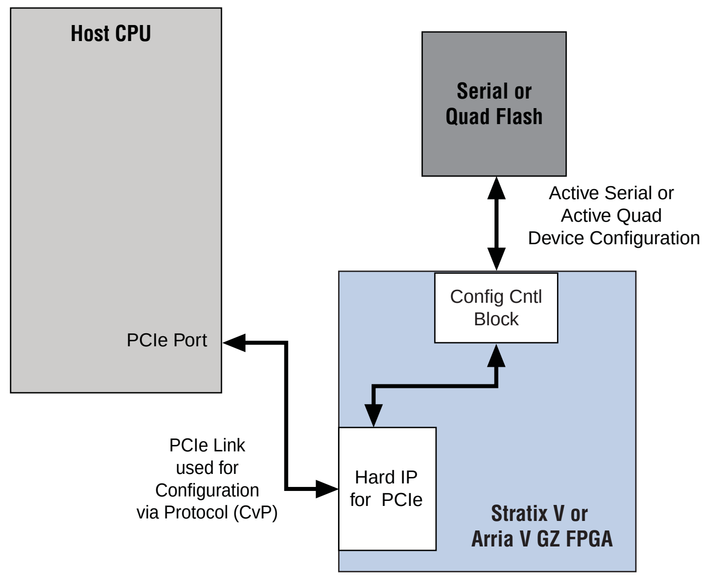
Configuration via Protocol (CvP)

The Arria V GZ architecture introduces has an option for sequencing the processes that configure the FPGA and initializes 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. In Arria V GZ, 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.

In Arria V GZ devices, 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 12–1. CvP in Arria V GZ Devices


Host CPU
PCIe Port
Serial or Quad Flash
Active Serial or Active Quad Device Configuration
Config Cntl Block
Hard IP for PCIe
Stratix V or Arria V GZ FPGA
PCIe Link used for Configuration via Protocol (CvP)

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.
- Enables dynamic core updates without requiring a system power down.
- Improves security for the proprietary core bitstream.
- Reduces system costs by reducing the size of the flash device to store the **.pof**.
- Facilitates hardware acceleration.
- May reduce system size because a single CvP link can be used to configure multiple FPGAs.

CvP is available for Gen1 and Gen2 configurations

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**, **ECRC generation**, and **ECRC forwarding** under the **PCI Express/PCI Capabilities** heading using the parameter editor to enable this functionality.

For more information about error handling, refer to the *Error Signaling and Logging* which is Section 6.2 of the *PCI Express Base Specification, Rev. 2.1*.

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 12–1 summarizes the RX ECRC functionality for all possible conditions.

Table 12–1. ECRC Operation on RX Path

ECRC Forwarding	ECRC Check Enable (1)	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

Note to Table 12–1:

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

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. Table 12–2 summarizes the TX ECRC generation and forwarding. In this table, if `TD` is 1, the TLP includes an ECRC. `TD` is the TL digest bit of the TL packet described in Appendix A, Transaction Layer Packet (TLP) Header Formats.

Table 12–2. ECRC Generation and Forwarding on TX Path (1)

ECRC Forwarding	ECRC Generation Enable (2)	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
		`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	

Notes to Table 12–2:

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

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

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.

The Arria V GZ Hard IP for PCI Express supports lane reversal, which 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 12–3 summarizes the lane assignments for normal configuration.

Table 12–3. 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 IP core	—	—	—	—	—	—	—	0

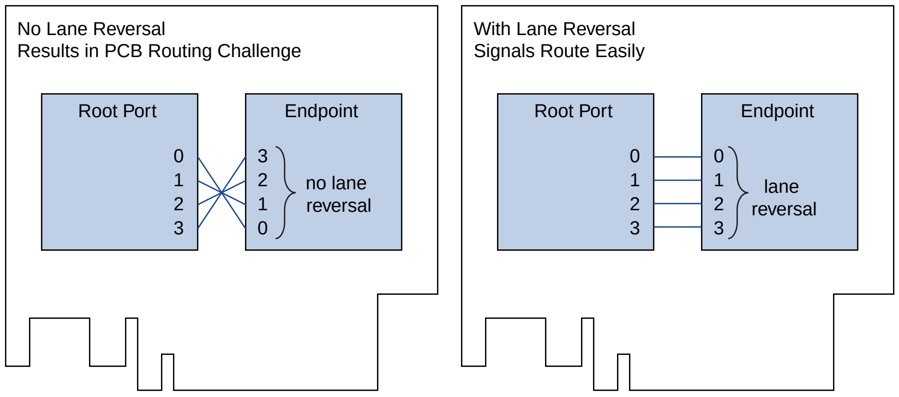
Table 12–4 summarizes the lane assignments with lane reversal.

Table 12–4. 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 assignments	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 12–2 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.

Figure 12–2. Using Lane Reversal to Solve PCB Routing Problems



13. Interrupts

This chapter describes interrupts for the following configurations:

- "Interrupts for Endpoints Using the Avalon-ST Application Interface" on page 13–1
- "Interrupts for Root Ports Using the Avalon-ST Interface to the Application Layer" on page 13–5
- "Interrupts for Endpoints Using the Avalon-MM Interface to the Application Layer" on page 13–5
- "Interrupts for End Points Using the Avalon-MM Interface with Multiple MSI/MSI-X Support" on page 13–7

Refer to "Interrupts for Endpoints" on page 8–32 and "Interrupts for Root Ports Using the Avalon-ST Interface to the Application Layer" on page 13–5 for a description of the interrupt signals.

Interrupts for Endpoints Using the Avalon-ST Application Interface

The Arria V GZ Hard IP for PCI Express provides support for PCI Express legacy interrupts, MSI, and MSI-X 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 INTX mode, after which time software decides whether to switch to MSI mode by programming the `msi_enable` bit of the `MSI message control` register (bit[16] of 0x050) to 1 or to MSI-X mode if you turn 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.

Refer to section 6.1 of *PCI Express 2.1 Base Specification* for a general description of PCI Express interrupt support for Endpoints.

MSI Interrupts

MSI interrupts are signaled on the PCI Express link using a single dword memory write TLPs generated internally by the Arria V GZ Hard IP for PCI Express. The `app_msi_req` input port controls MSI interrupt generation. When the input port asserts `app_msi_req`, it causes a 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. To enable MSI interrupts, software must first set the `MSI enable` bit (Table 8–18 on page 8–41) and then disable legacy interrupts by setting the `Interrupt Disable` which is bit 10 of the `Command` register (Table 9–2 on page 9–2).

Figure 13–1 illustrates the architecture of the MSI handler block.

Figure 13–1. MSI Handler Block

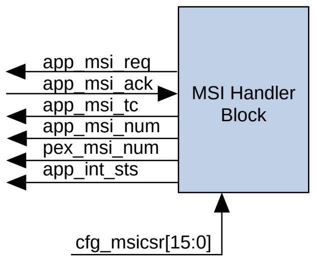
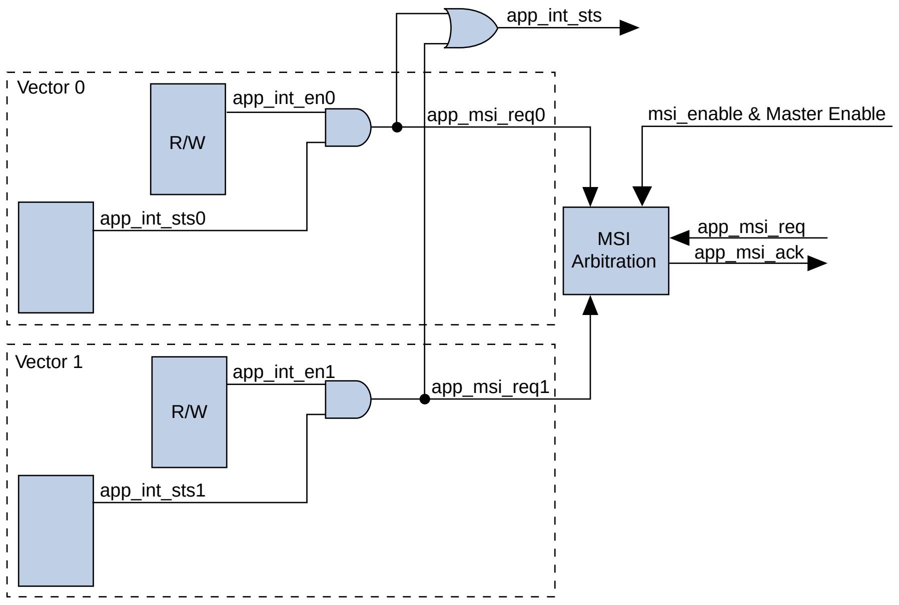


Figure 13–2 illustrates a possible implementation of the MSI handler block with a per vector enable bit. A global Application Layer interrupt enable can also be implemented instead of this per vector MSI.

Figure 13–2. Example Implementation of the MSI 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 Figure 13–3, 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 13–3. MSI Request Example

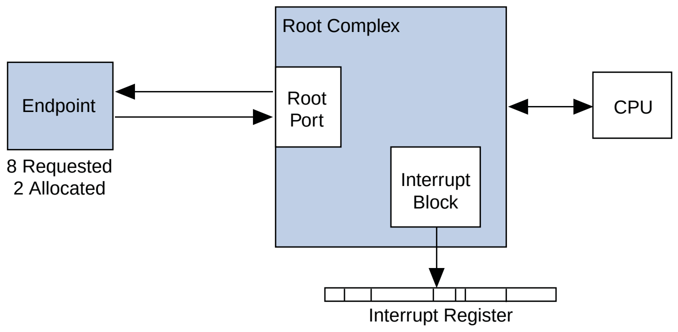


Table 13–1 describes 3 example implementations; 1 in which all 32 MSI messages are allocated and 2 in which only 4 are allocated.

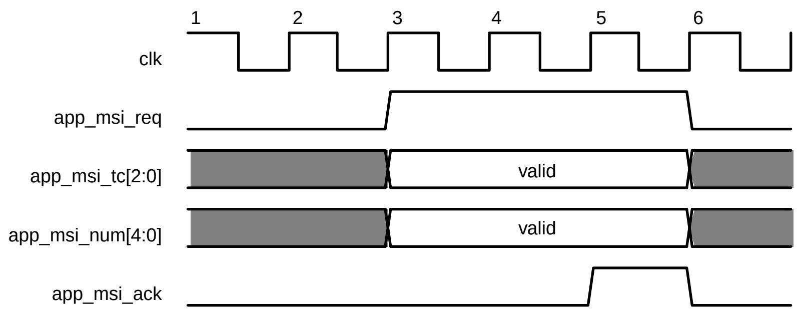
Table 13–1. MSI Messages Requested, Allocated, and Mapped

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 TC0. 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.

Figure 13–4 illustrates the interactions among MSI interrupt signals for the Root Port in Figure 13–3. The minimum latency possible between `app_msi_req` and `app_msi_ack` is one clock cycle.

Figure 13–4. MSI Interrupt Signals Waveform (1)



Note to Figure 13–4:

(1) `app_msi_req` can extend beyond `app_msi_ack` before deasserting. F

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.

MSI-X TLPs are generated by the Application Layer and sent through the TX interface. They are single dword memory writes so that `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 MSI-X Configuration register. These bits are available on the `tl_cfg_ctl` output bus.

For more information about implementing the MSI-X capability structure, refer Section 6.8.2. of the *PCI Local Bus Specification, Revision 3.0*.

Legacy Interrupts

Legacy interrupts are signaled on the PCI Express link using message TLPs that are generated internally by the Arria V GZ Hard IP for PCI Express. The `app_int_sts` input port controls interrupt generation. When the input port asserts `app_int_sts`, it causes an `Assert_INTA` message TLP to be generated and sent upstream. Deassertion of the `app_int_sts` input port causes a `Deassert_INTA` 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 (Table 9–2 on page 9–2). Then, turn off the `MSI Enable` bit (Table 8–18 on page 8–41.)

Figure 13–5 illustrates interrupt timing for the legacy interface. In this figure the assertion of `app_int_ack` instructs the Hard IP for PCI Express to send a `Assert_INTA` message TLP.

Figure 13–5. Legacy Interrupt Assertion

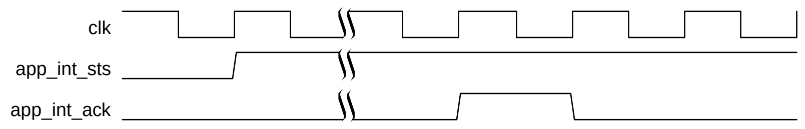
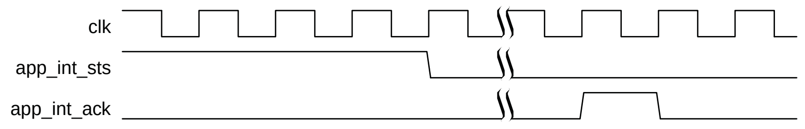


Figure 13–6 illustrates the timing for deassertion of legacy interrupts. The assertion of `app_int_ack` instructs the Hard IP for PCI Express to send a `Deassert_INTA` message.

Figure 13–6. Legacy Interrupt Deassertion



Interrupts for Root Ports Using the Avalon-ST Interface to the Application Layer

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

- MSI—Root Ports receive MSI interrupts through the Avalon-ST RX TLP of type `MWr`. This is a memory mapped mechanism.
- Legacy—Legacy interrupts are translated into TLPs of type `Message Interrupt` which is sent to the Application Layer using the `int_status[3:0]` 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_num[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.

Interrupts for Endpoints Using the Avalon-MM Interface to the Application Layer

The PCI Express Avalon-MM bridge supports MSI or legacy interrupts. The completer only single dword variant includes an interrupt handler that implements both INTX and MSI interrupts. Support requires instantiation of the CRA slave module where the interrupt registers and control logic are implemented.

The PCI Express Avalon-MM bridge supports the Avalon-MM individual requests interrupt scheme: multiple input signals indicate incoming interrupt requests, and software must determine priorities for servicing simultaneous interrupts the Avalon-MM Arria V GZ Hard IP for PCI Express receives.

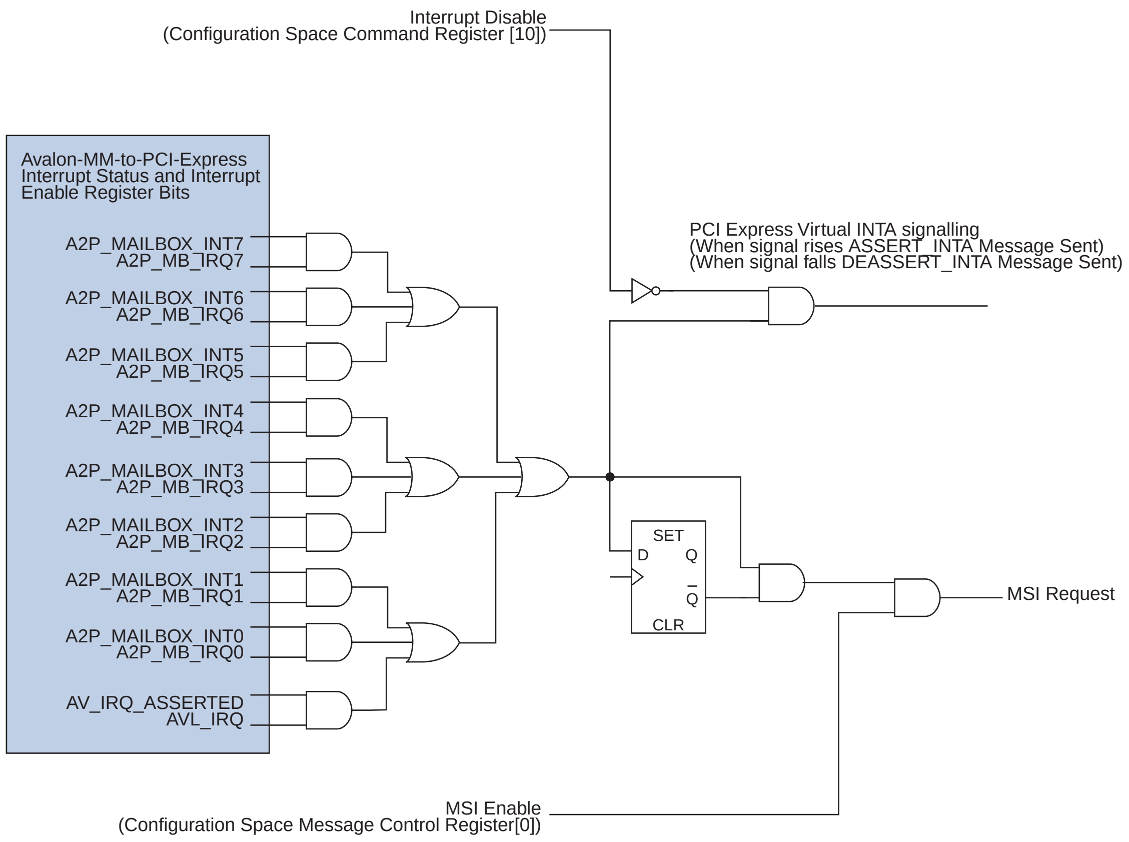
The RX master module port has as many as 16 Avalon-MM interrupt input signals (`RXmirq_irq[<n>:0]`, where <*n*> ≤ 15)). Each interrupt signal indicates a distinct interrupt source. Assertion of any of these signals, or a PCI Express mailbox register write access, sets a bit in the `Avalon-MM to PCI Express Interrupt Status` register. Multiple bits can be set at the same time; software determines priorities for servicing simultaneous incoming interrupt requests. Each set bit in the `Avalon-MM to PCI Express Interrupt Status` interrupt status register generates a PCI Express interrupt, if enabled, when software determines its turn. Software can enable the individual interrupts by writing to the"Avalon-MM to PCI Express Interrupt Enable Register 0x0050" on page 9–13 through the CRA slave.

When any interrupt input signal is asserted, the corresponding bit is written in the "Avalon-MM to PCI Express Interrupt Status Register 0x0040" on page 9–13. Software reads this register and decides priority on servicing requested interrupts.

After servicing the interrupt, software must clear the appropriate serviced interrupt `status` bit and ensure that no other interrupts are pending. For interrupts caused by "Avalon-MM to PCI Express Interrupt Status Register 0x0040" on page 9–13 mailbox writes, the status bits should be cleared in the "Avalon-MM to PCI Express Interrupt Status Register 0x0040" on page 9–13. For interrupts due to the incoming interrupt signals on the Avalon-MM interface, the interrupt status should be cleared in the Avalon-MM component that sourced the interrupt. This sequence prevents interrupt requests from being lost during interrupt servicing.

Figure 13–7 shows the logic for the entire interrupt generation process.

Figure 13–7. Avalon-MM Interrupt Propagation to the PCI Express Link


Interrupt Disable
(Configuration Space Command Register [10])
Avalon-MM-to-PCI-Express Interrupt Status and Interrupt Enable Register Bits
A2P_MAILBOX_INT7
A2P_MB_IRQ7
A2P_MAILBOX_INT6
A2P_MB_IRQ6
A2P_MAILBOX_INT5
A2P_MB_IRQ5
A2P_MAILBOX_INT4
A2P_MB_IRQ4
A2P_MAILBOX_INT3
A2P_MB_IRQ3
A2P_MAILBOX_INT2
A2P_MB_IRQ2
A2P_MAILBOX_INT1
A2P_MB_IRQ1
A2P_MAILBOX_INT0
A2P_MB_IRQ0
AV_IRQ_ASSERTED
AVL_IRQ
PCI Express Virtual INTA signalling
(When signal rises ASSERT_INTA Message Sent)
(When signal falls DEASSERT_INTA Message Sent)
SET
D
Q
Q
CLR
MSI Request
MSI Enable
(Configuration Space Message Control Register[0])

Enabling MSI or Legacy Interrupts

The PCI Express Avalon-MM bridge selects either MSI or legacy interrupts automatically based on the standard interrupt controls in the PCI Express Configuration Space registers. Software can write the `Interrupt Disable` bit, which is bit 10 of the `Command` register (at Configuration Space offset 0x4) to disable legacy interrupts. Software can write the `MSI Enable` bit, which is bit 0 of the `MSI Control Status` register in the MSI capability register (bit 16 at configuration space offset 0x50), to enable MSI interrupts.

Software can only enable one type of interrupt at a time. However, to change the selection of MSI or legacy interrupts during operation, software must ensure that no interrupt request is dropped. Therefore, software must first enable the new selection and then disable the old selection. To set up legacy interrupts, software must first clear the `Interrupt Disable` bit and then clear the `MSI enable` bit. To set up MSI interrupts, software must first set the `MSI enable` bit and then set the `Interrupt Disable` bit.

Generation of Avalon-MM Interrupts

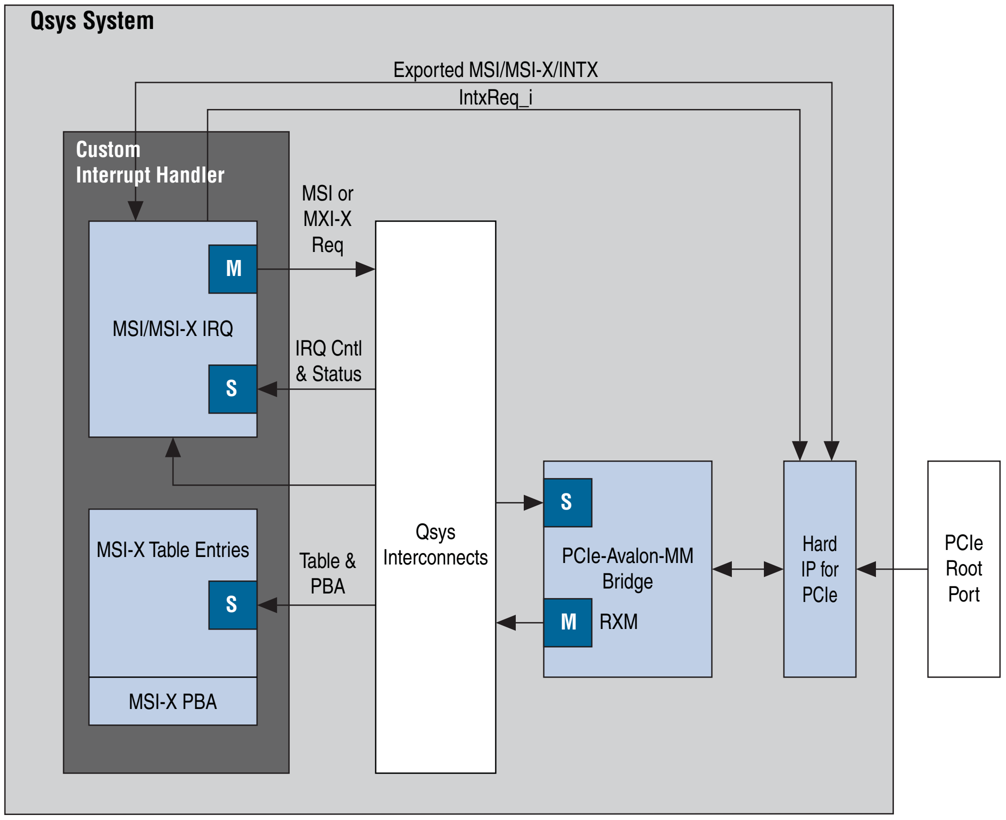
Generation of Avalon-MM interrupts requires the instantiation of the CRA slave module where the interrupt registers and control logic are implemented. The CRA slave port has an Avalon-MM Interrupt output signal, `cra_Irq_irq`. A write access to an Avalon-MM mailbox register sets one of the `P2A_MAILBOX_INT<`*n*`>` bits in the "Avalon-MM to PCI Express Interrupt Status Register 0x0040" on page 9–13 and asserts the `cra_Irq_o` or `cra_Irq_irq` output, if enabled. Software can enable the interrupt by writing to the "INT-X Interrupt Enable Register for Endpoints 0x3070" on page 9–21 through the CRA slave. After servicing the interrupt, software must clear the appropriate serviced interrupt `status` bit in the PCI-Express-to-Avalon-MM `Interrupt Status` register and ensure that no other interrupt is pending.

Interrupts for End Points Using the Avalon-MM Interface with Multiple MSI/MSI-X Support

If you select **Enable multiple MSI/MSI-X support** under the **Avalon-MM System Settings** banner in the GUI, the Hard IP for PCI Express exports the MSI, MSI-X, and INTx interfaces to the Application Layer. The Application Layer must include a Custom Interrupt Handler to send interrupts to the Root Port. You must design this Custom Interrupt Handler. Figure 13–8 provides a an overview of the logic for the Custom Interrupt Handler. The Custom Interrupt Handler should include hardware to perform the following tasks:

- An MSI/MXI-X IRQ Avalon-MM Master port to drive MSI or MSI-X interrupts as memory writes to the PCIe Avalon-MM Bridge.
- A legacy interrupt signal, `IntxReq_i`, to drive legacy interrupts from the MSI/MSI-X IRQ module to the Hard IP for PCI Express.
- An MSI/MSI-X Avalon-MM Slave port to receive interrupt control and status from PCIe Root Port.
- An MSI-X table to store the MSI-X table entries. The PCIe Root Port sets up this table.

Figure 13–8. Block Diagram for Custom Interrupt Handler



Refer to Interrupts for Endpoints for the definitions of MSI, MSI-X and INTx buses.

1. For more information about implementing MSI or MSI-X interrupts, refer to the *PCI Local Bus Specification, Revision 2.3, MSI-X ECN*.

14. Flow Control

Throughput analysis requires that you understand the Flow Control Loop, shown in "Flow Control Update Loop" on page 14–2. This chapter discusses the Flow Control Loop and strategies to improve throughput. It covers the following topics:

- Throughput of Posted Writes
- Throughput of Non-Posted Reads

Throughput of Posted Writes

The throughput of posted writes is limited primarily by the Flow Control Update loop shown in Figure 14–1. 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.

Figure 14–1 shows the main components of the Flow Control Update loop with two communicating PCI Express ports:

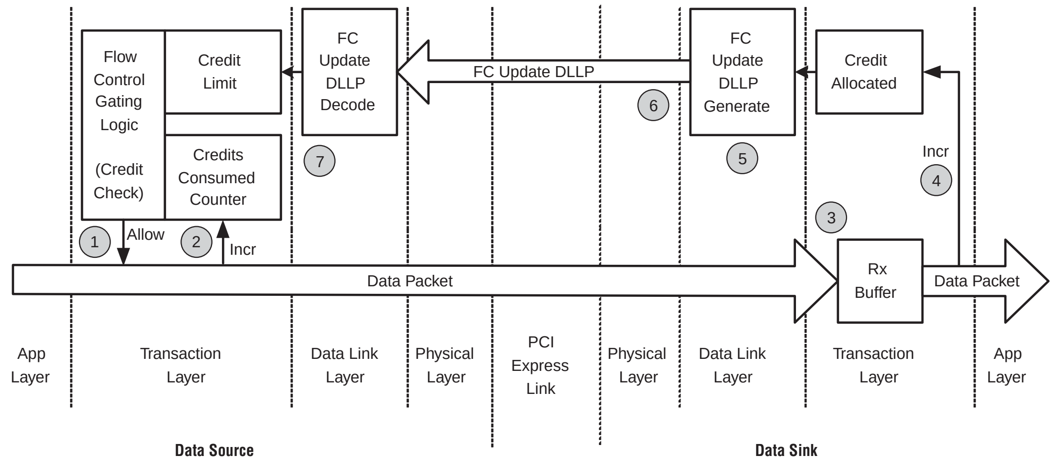
- Write Requester
- Write Completer

As the *PCI Express Base Specification 3.0* describe, 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 issued 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 14–1. Flow Control Update Loop



The following numbered steps describe each step in the Flow Control Update loop. The corresponding numbers on Figure 14–1 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 FC Update DLLP is received back at the original write requester and the `credit limit` value is updated. If packets are stalled waiting for credits, they can now be transmitted.

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.

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 Arria V GZ Hard IP for PCI Express 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 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" on page 14–1. 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 Arria V GZ Hard IP for PCI Express. 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.

15. Error Handling

Each PCI Express compliant device must implement a basic level of error management and can optionally implement advanced error management. The Altera Arria V GZ Hard IP for PCI Express implements both basic and advanced error reporting. Given its position and role within the fabric, error handling for a Root Port is more complex than that of an Endpoint.

The *PCI Express Base Specification 2.1 and 3.0* define three types of errors, outlined in Table 15–1.

Table 15–1. Error Classification

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.

The following sections describe the errors detected by the three layers of the PCI Express protocol and error logging. It includes the following sections:

- Physical Layer Errors
- Data Link Layer Errors
- Transaction Layer Errors
- Error Reporting and Data Poisoning
- Uncorrectable and Correctable Error Status Bits

Physical Layer Errors

Table 15–2 describes errors detected by the Physical Layer.

Table 15–2. Errors Detected by the Physical Layer [(1)]

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: 100: 8B/10B Decode Error 101: Elastic Buffer Overflow 110: Elastic Buffer Underflow 111: Disparity Error ■ Deskew error caused by overflow of the multilane deskew FIFO. ■ Control symbol received in wrong lane.

Note to Table 15–2:

(1) Considered optional by the PCI Express specification.

Data Link Layer Errors

Table 15–3 describes errors detected by the Data Link Layer.

Table 15–3. 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. (Refer to Figure 7–6 on page 7–12.)

Transaction Layer Errors

Table 15–4 describes errors detected by the Transaction Layer.

Table 15–4. Errors Detected by the Transaction Layer (Part 1 of 3)

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 2.1* 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.
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 a Windows address

Table 15–4. Errors Detected by the Transaction Layer (Part 2 of 3)

Error	Type	Description
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. ■ A TLP in which the combination of format and type is not specified by the PCI Express specification.

Table 15–4. Errors Detected by the Transaction Layer (Part 3 of 3)

Error	Type	Description
Malformed TLP (continued)	Uncorrectable (fatal)	■ A request specifies an address/length combination that causes a memory space access to exceed a 4 KByte 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 to Table 15–4:

(1) Considered optional by the *PCI Express Base Specification Revision 2.1*.

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 2.1* 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 15–5 lists the conditions that cause parity errors.

Table 15–5. 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.

Uncorrectable and Correctable Error Status Bits

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

Figure 15–1 illustrates the 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 15–1. Uncorrectable Error Status Register

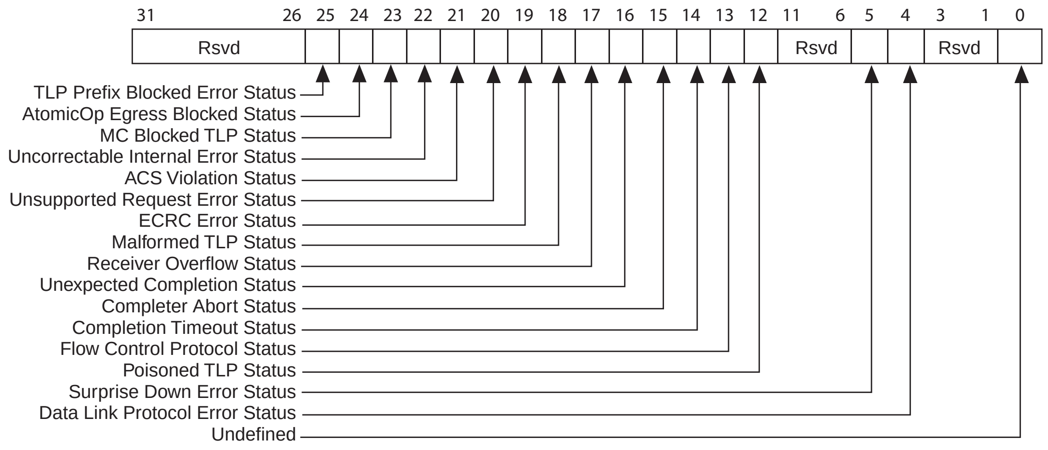
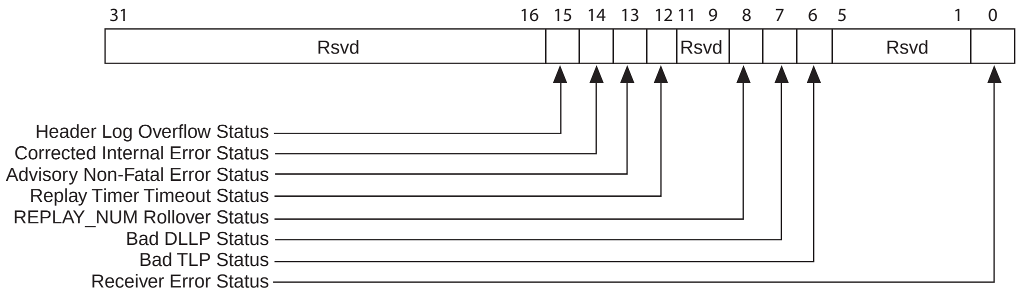

31
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
6
5
4
3
1
0
Rsvd
Rsvd
Rsvd
TLP Prefix Blocked Error Status
AtomicOp Egress Blocked Status
MC Blocked TLP Status
Uncorrectable Internal Error Status
ACS Violation Status
Unsupported Request Error Status
ECRC Error Status
Malformed TLP Status
Receiver Overflow Status
Unexpected Completion Status
Completer Abort Status
Completion Timeout Status
Flow Control Protocol Status
Poisoned TLP Status
Surprise Down Error Status
Data Link Protocol Error Status
Undefined

Figure 15–2 illustrates the 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.0

Figure 15–2. Correctable Error Status Register


31
16
15
14
13
12
11
9
8
7
6
5
1
0
Rsvd
Rsvd
Rsvd
Header Log Overflow Status
Corrected Internal Error Status
Advisory Non-Fatal Error Status
Replay Timer Timeout Status
REPLAY_NUM Rollover Status
Bad DLLP Status
Bad TLP Status
Receiver Error Status

16. SDC Timing Constraints

You must include component-level Synopsys Design Constraints (SDC) timing constraints for the Arria V GZ Hard IP for PCI Express IP Core and system-level constraints for your complete design. The example design that Altera describes in the Testbench and Design Example chapter includes the constraints required for the for Arria V GZ Hard IP for PCI Express IP Core and example design. A single file, *<install_dir>***/ip/altera/altera_pcie/altera_pcie_hip_ast_ed/altpcied_sv.sdc**, includes both the component-level and system-level constraints. Example 16–1 shows **altpcied_sv.sdc**. This **.sdc** file includes constraints for three components:

- Arria V GZ Hard IP for PCI Express IP Core
- Transceiver Reconfiguration Controller IP Core
- Transceiver PHY Reset Controller IP Core

Example 16–1. SDC Timing Constraints Required for the Arria V GZ Hard IP for PCIe and Design Example

```
# Constraints required for the Hard IP for PCI Express
# derive_pll_clock is used to calculate all clock derived from PCIe refclk
# the derive_pll_clocks and derive clock_uncertainty should only be applied
# once across all of the SDC files used in a project

derive_pll_clocks -create_base_clocks
derive_clock_uncertainty

##############################################################################
# PHY IP reconfig controller constraints
# Set reconfig_xcvr clock
# this line will likely need to be modified to match the actual clock pin name
# used for this clock, and also changed to have the correct period set for the actually
# used clock
create_clock -period "125 MHz" -name {reconfig_xcvr_clk} {*reconfig_xcvr_clk*}
######################################################################

# HIP Soft reset controller SDC constraints
set_false_path -to   [get_registers *altpcie_rs_serdes|fifo_err_sync_r[0]]
set_false_path -from [get_registers *sv_xcvr_pipe_native*] -to [get_registers
*altpcie_rs_serdes|*]

# Hard IP testin pins SDC constraints
set_false_path -from [get_pins -compatibility_mode *hip_ctrl*]
```

SDC Constraints for the Hard IP for PCIe

In Example 16–1, you should only apply the first two constraints, to derive PLL clocks and clock uncertainty, once across all of the SDC files in your project. Differences between Fitter timing analysis and TimeQuest timing analysis arise if these constraints are applied more than once.

SDC Constraints for the Example Design

The Transceiver Reconfiguration Controller IP Core is the example design. The **.sdc** file includes constraints for the Transceiver Reconfiguration Controller IP Core. You may need to change the frequency and actual clock pin name to match your design.

The **.sdc** file also specifies some false timing paths for Transceiver Reconfiguration Controller and Transceiver PHY Reset Controller IP Cores. Be sure to include these constraints in your **.sdc** file.

17. Hard IP Reconfiguration and Transceiver Reconfiguration

This chapter describes features of the Arria V GZ Hard IP for PCI Express that you can use to reconfigure the core after power-up. It includes the following sections:

- Hard IP Reconfiguration Interface
- Transceiver PHY IP Reconfiguration

Hard IP Reconfiguration Interface

The Arria V GZ 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. For a complete description of the signals in this interface, refer to "Hard IP Reconfiguration Interface" on page 8–49.

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).

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.

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" on page 8–47.

Table 17–1 lists all of the registers that you can update using the PCI Express reconfiguration block interface.

Table 17–1. Dynamically Reconfigurable Registers in the Hard IP Implementation (Part 1 of 8)

Address	Bits	Description	Default Value	Additional Information
0x00	0	When 0, PCIe reconfig mode is enabled. When 1, PCIe reconfig mode is disabled and the original read only register values set in the programming file used to configure the device are restored.	b'1	—
0x01-0x88		Reserved.	—	
0x89	15:0	Vendor ID.	0x1172	Table 9–2 on page 9–2, Table 9–3 on page 9–2
0x8A	15:0	Device ID.	0x0001	Table 9–2 on page 9–2, Table 9–3 on page 9–2

Table 17–1. Dynamically Reconfigurable Registers in the Hard IP Implementation (Part 2 of 8)

Address	Bits	Description	Default Value	Additional Information
0x8B	7:0	Revision ID.	0x01	Table 9–2 on page 9–2, Table 9–3 on page 9–2
	15:8	Class code[7:0].	—	Table 9–2 on page 9–2, Table 9–3 on page 9–2
0x8C	15:0	Class code[23:8].	—	Table 9–2 on page 9–2
0x8D	15:0	Subsystem vendor ID.	0x1172	Table 9–2 on page 9–2
0x8E	15:0	Subsystem device ID.	0x0001	Table 9–2 on page 9–2
0x8F		Reserved.	—	
0x90	0	Advanced Error Reporting.	b'0	
	3:1	Low Priority VC (LPVC).0	b'000	
	7:4	VC arbitration capabilities.	b'00001	
	15:8	Reject Snoop Transaction.	b'00000000	
0x91	2:0	Max payload size supported. The following are the defined encodings: 000: 128 bytes max payload size. 001: 256 bytes max payload size. 010: 512 bytes max payload size. 011: 1024 bytes max payload size. 100: 2048 bytes max payload size. 101: 4096 bytes max payload size. 110: Reserved. 111: Reserved.	b'010	Table 9–8 on page 9–5, Device Capability register
	3	Surprise Down error reporting capabilities. (Available in *PCI Express Base Specification Revision 1.1* compliant Cores, only.) `Downstream Port`. This bit must be set to 1 if the component supports the optional capability of detecting and reporting a Surprise Down error condition. `Upstream Port`. For upstream ports and components that do not support this optional capability, this bit must be hardwired to 0.	b'0	Table 9–8 on page 9–5, Link Capability register
	4	Data Link Layer active reporting capabilities. (Available in *P CI Express Base Specification Revision 1.1* compliant Cores, only.) Downstream Port: This bit must be set to 1 if the component supports the optional capability of reporting the DL_Active state of the Data Link Control and Management state machine. Upstream Port: For upstream ports and components that do not support this optional capability, this bit must be hardwired to 0.	b'0	Table 9–8 on page 9–5, Link Capability register

Table 17–1. Dynamically Reconfigurable Registers in the Hard IP Implementation (Part 3 of 8)

Address	Bits	Description	Default Value	Additional Information
	5	Extended TAG field supported.	b'0	Table 9–8 on page 9–5, Device Capability register
	8:6	Endpoint L0s acceptable latency. The following encodings are defined: b'000 – Maximum of 64 ns. b'001 – Maximum of 128 ns. b'010 – Maximum of 256 ns. b'011 – Maximum of 512 ns. b'100 – Maximum of 1 µs. b'101 – Maximum of 2 µs. b'110 – Maximum of 4 µs. b'111– No limit.	b'000	Table 9–8 on page 9–5, Device Capability register
	11:9	Endpoint L1 acceptable latency. The following encodings are defined: b'000 – Maximum of 1 µs. b'001 – Maximum of 2 µs. b'010 – Maximum of 4 µs. b'011 – Maximum of 8 µs. b'100 – Maximum of 16 µs. b'101 – Maximum of 32 µs. b'110 – Maximum of 64 µs. b'111 – No limit.	b'000	Table 9–8 on page 9–5, Device Capability register
	14:12	These bits record the presence or absence of the attention and power indicators. [0]: Attention button present on the device. [1]: Attention indicator present for an endpoint. [2]: Power indicator present for an endpoint.	b'000	Table 9–8 on page 9–5, Slot Capability register
0x91	15	Role-Based error reporting. (Available in *PCI Express Base Specification Revision 1.1* compliant Cores only.)In 1.1 compliant cores, this bit should be set to 1.	b'1	Table 9–7 on page 9–4, Correctable Error Mask register
0x92	1:0	Slot Power Limit Scale.	b'00	Table 9–8 on page 9–5, Slot Capability register

Table 17–1. Dynamically Reconfigurable Registers in the Hard IP Implementation (Part 4 of 8)

Address	Bits	Description	Default Value	Additional Information
0x92	7:2	Max Link width.	b'000100	Table 9–8 on page 9–5, Link Capability register
	9:8	L0s Active State power management support. L1 Active State power management support.	b'01	Table 9–8 on page 9–5, Link Capability register
	15:10	L1 exit latency common clock.	b'000000	Table 9–8 on page 9–5, Link Capability register
		L1 exit latency separated clock. The following encodings are defined: b'000 – Less than 1 µs. b'001 – 1 µs to less than 2 µs. b'010 – 2 µs to less than 4 µs. b'011 – 4 µs to less than 8 µs. b'100 – 8 µs to less than 16 µs. b'101 – 16 µs to less than 32 µs. b'110 – 32 µs to 64 µs. b'111 – More than 64 µs.		
0x93		[0]: Attention button implemented on the chassis.	b'0000000	Table 9–8 on page 9–5, Slot Capability register
		[1]: Power controller present.		
		[2]: Manually Operated Retention Latch (MRL) sensor present.		
		[3]: Attention indicator present for a root port, switch, or bridge.		
		[4]: Power indicator present for a root port, switch, or bridge.		
		[5]: Hot-plug surprise: When this bit set to1, a device can be removed from this slot without prior notification.		
	6:0	[6]: Hot-plug capable.		
	9:7	Reserved.	b'000	
	15:10	Slot Power Limit Value.	b'00000000	
0x94	1:0	Reserved.	—	Table 9–8 on page 9–5, Slot Capability register
	2	Electromechanical Interlock present (Available in *PCI Express Base Specification Revision 1.1* compliant IP cores only.)	b'0	
	15:3	Physical Slot Number (if slot implemented). This signal indicates the physical slot number associated with this port. It must be unique within the fabric.	b'0	
0x95	7:0	NFTS_SEPCLK. The number of fast training sequences for the separate clock.	b'10000000	—
	15:8	NFTS_COMCLK. The number of fast training sequences for the common clock.	b'10000000	

Table 17–1. Dynamically Reconfigurable Registers in the Hard IP Implementation (Part 5 of 8)

Address	Bits	Description	Default Value	Additional Information
0x96	3:0	Completion timeout ranges. The following encodings are defined: b'0001: range A. b'0010: range B. b'0011: range A&B. b'0110: range B&C. b'0111: range A,B&C. b'1110: range B,C&D. b'1111: range A,B,C&D. All other values are reserved.	b'0000	Table 9–8 on page 9–5, Device Capability register 2
	4	Completion Timeout supported 0: completion timeout disable not supported 1: completion timeout disable supported	b'0	Table 9–8 on page 9–5, Device Capability register 2
	7:5	Reserved.	b'0	—
	8	ECRC generate.	b'0	Table 9–7 on page 9–4, Advanced Error Capability and Control register
	9	ECRC check.	b'0	Table 9–7 on page 9–4, Advanced Error Capability and Control register
	10	No command completed support. (available only in *PCI Express Base Specification Revision 1.1* compliant Cores)	b'0	Table 9–8 on page 9–5, Slot Capability register
	13:11	Number of functions MSI capable. b'000: 1 MSI capable. b'001: 2 MSI capable. b'010: 4 MSI capable. b'011: 8 MSI capable. b'100: 16 MSI capable. b'101: 32 MSI capable.	b'010	Table 9–4 on page 9–3, Message Control register
	14	MSI 32/64-bit addressing mode. b'0: 32 bits only. b'1: 32 or 64 bits	b'1	
	15	MSI per-bit vector masking (read-only field).	b'0	
	0	Function supports MSI.	b'1	Table 9–4 on page 9–3, Message Control register for MSI
	3:1	Interrupt pin.	b'001	—
	5:4	Reserved.	b'00	
	6	Function supports MSI-X.	b'0	Table 9–4 on page 9–3, Message Control register for MSI

Table 17–1. Dynamically Reconfigurable Registers in the Hard IP Implementation (Part 6 of 8)

Address	Bits	Description	Default Value	Additional Information
0x97	15:7	MSI-X table size	b'0	Table 9–5 on page 9–3, MSI-X Capability Structure
0x98	1:0	Reserved.	—	
	4:2	MSI-X Table BIR.	b'0	
	15:5	MIS-X Table Offset.	b'0	Table 9–5 on page 9–3, MSI-X Capability Structure
0x99	15:10	MSI-X PBA Offset.	b'0	—
0x9A	15:0	Reserved.	b'0	
0x9B	15:0	Reserved.	b'0	
0x9C	15:0	Reserved.	b'0	
0x9D	15:0	Reserved.	b'0	
0x9E	3:0	Reserved.		
	7:4	Number of EIE symbols before NFTS.	b'0100	
	15:8	Number of NFTS for separate clock in Gen2 rate.	b'11111111	
0x9F	7:0	Number of NFTS for common clock in Gen2 rate.	b'11111111	Table 9–8 on page 9–5, Link Control register 2
	8	Selectable de-emphasis.	b'0	
	12:9	PCIe Capability Version. b'0000: Core is compliant to PCIe Specification 1.0a or 1.1. b'0001: Core is compliant to PCIe Specification 1.0a or 1.1. b'0010: Core is compliant to PCIe Specification 2.0.	b'0010	Table 9–8 on page 9–5, PCI Express capability register
	15:13	L0s exit latency for common clock. Gen1: (*N_FTS* (of separate clock) + 1 (for the SKIPOS)) * 4 * 10 * `UI` (*UI* = 0.4 ns). Gen2: [(*N_FTS2* (of separate clock) + 1 (for the SKIPOS)) * 4 + 8 (max number of received *EIE*)] * 10 * *UI* (*UI* = 0.2 ns).	b'110	Table 9–8 on page 9–5, Link Capability register
0xA0	2:0	L0s exit latency for separate clock. Gen1: (*N_FTS* (of separate clock) + 1 (for the SKIPOS)) * 4 * 10 * *UI* (*UI* = 0.4 ns). Gen2: [(*N_FTS2* (of separate clock) + 1 (for the SKIPOS)) * 4 + 8 (max number of received *EIE*)] * 10 * *UI* (*UI* = 0.2 ns). b'000 – Less than 64 ns. b'001 – 64 ns to less than 128 ns. b'010 – 128 ns to less than 256 ns. b'011 – 256 ns to less than 512 ns. b'100 – 512 ns to less than 1 µs. b'101 – 1 µs to less than 2 µs. b'110 – 2 µs to 4 µs. b'111 – More than 4 µs.	b'110	Table 9–8 on page 9–5, Link Capability register
	15:3	Reserved.	0x0000	

Table 17–1. Dynamically Reconfigurable Registers in the Hard IP Implementation (Part 7 of 8)

Address	Bits	Description	Default Value	Additional Information
0xA1		BAR0[31:0].		Table 9–2 on page 9–2, Table 9–3 on page 9–2,
	0	BAR0[0]: I/O Space.	b'0	
	2:1	BAR0[2:1]: Memory Space. 10: 64-bit address. 00: 32-bit address.	b'10	
	3	BAR0[3]: Prefetchable.	b'1	
		BAR0[31:4]: Bar size mask.	0xFFFFFFF	
	15:4	BAR0[15:4].	b'0	
0xA2	15:0	BAR0[31:16].	b'0	
0xA3		BAR1[63:32].	b'0	
	0	BAR1[32]: I/O Space.	b'0	
	2:1	BAR1[34:33]: Memory Space (see bit settings for BAR0).	b'0	
	3	BAR1[35]: Prefetchable.	b'0	
		BAR1[63:36]: Bar size mask.	b'0	
	15:4	BAR1[47:36].	b'0	
0xA4	15:0	BAR1[63:48].	b'0	
0xA5		BAR2[95:64]:	b'0	Table 9–2 on page 9–2
	0	BAR2[64]: I/O Space.	b'0	
	2:1	BAR2[66:65]: Memory Space (see bit settings for BAR0).	b'0	
	3	BAR2[67]: Prefetchable.	b'0	
		BAR2[95:68]: Bar size mask.	b'0	
	15:4	BAR2[79:68].	b'0	
0xA6	15:0	BAR2[95:80].	b'0	
0xA7		BAR3[127:96].	b'0	Table 9–2 on page 9–2
	0	BAR3[96]: I/O Space.	b'0	
	2:1	BAR3[98:97]: Memory Space (see bit settings for BAR0).	b'0	
	3	BAR3[99]: Prefetchable.	b'0	
		BAR3[127:100]: Bar size mask.	b'0	
	15:4	BAR3[111:100].	b'0	
0xA8	15:0	BAR3[127:112].	b'0	
0xA9		BAR4[159:128].	b'0	
	0	BAR4[128]: I/O Space.	b'0	
	2:1	BAR4[130:129]: Memory Space (see bit settings for BAR0).	b'0	
	3	BAR4[131]: Prefetchable.	b'0	
		BAR4[159:132]: Bar size mask.	b'0	
	15:4	BAR4[143:132].	b'0	

Table 17–1. Dynamically Reconfigurable Registers in the Hard IP Implementation (Part 8 of 8)

Address	Bits	Description	Default Value	Additional Information
0xAA	15:0	BAR4[159:144].	b'0	
0xAB		BAR5[191:160].	b'0	
	0	BAR5[160]: I/O Space.	b'0	
	2:1	BAR5[162:161]: Memory Space (see bit settings for BAR0).	b'0	
	3	BAR5[163]: Prefetchable.	b'0	
		BAR5[191:164]: Bar size mask.	b'0	
	15:4	BAR5[175:164].	b'0	
0xAC	15:0	BAR5[191:176].	b'0	
0xAD	15:0	Expansion BAR[223:192]: Bar size mask.	b'0	
		Expansion BAR[207:192].	b'0	
0xAE	15:0	Expansion BAR[223:208].	b'0	
0xAF	1:0	IO. 00: no IO windows. 01: IO 16 bit. 11: IO 32-bit.	b'0	Table 9–3 on page 9–2
	3:2	Prefetchable. 00: not implemented. 01: prefetchable 32. 11: prefetchable 64.	b'0	
	15:4	Reserved.	—	
B0	5:0	Reserved	—	—
	6	Selectable de-emphasis, operates as specified in the *PCI Express Base Specification* when operating at the 5.0GT/s rate: 1: 3.5 dB 0: -6 dB. This setting has no effect when operating at the 2.5GT/s rate.		
	9:7	Transmit Margin. Directly drives the transceiver `tx_pipemargin` bits.		
0xB1-FF		Reserved.		

Transceiver PHY IP Reconfiguration

As silicon progresses towards smaller process nodes, circuit performance is affected by variations due to process, voltage, and temperature (PVT). Consequently, Gen3 design require offset cancellation and adaptive equalization (AEQ) to ensure correct operation. Altera's Qsys example designs all include Transceiver Reconfiguration Controller and Altera PCIe Reconfig Driver IP Cores that automatically perform these functions during the LTSSM equalization states.

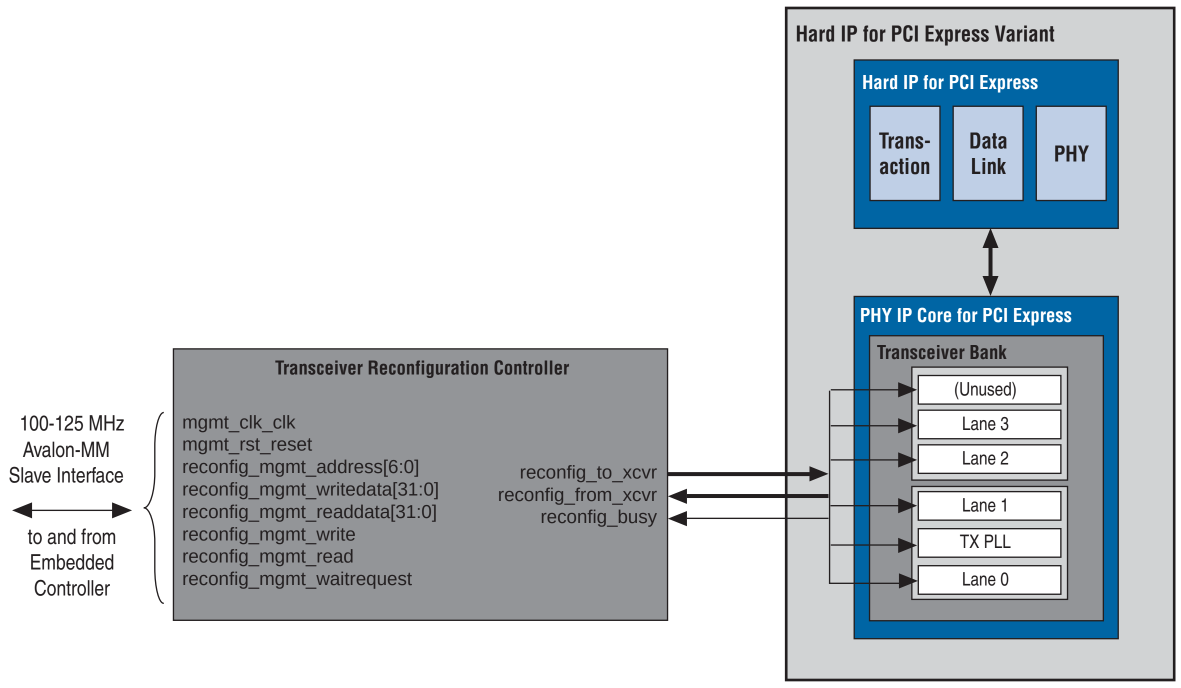
Gen1 and Gen2 do not require any signal integrity functions to operate correctly. However, the Transceiver Reconfiguration Controller IP Core is required to perform channel merging.

Connecting the Transceiver Reconfiguration Controller IP Core

You can instantiate this component using the MegaWizard Plug-In Manager or Qsys. It is available for Arria V GZ devices and can be found in the **Interfaces/Transceiver PHY** category for the MegaWizard design flow. In Qsys, you can find the Transceiver Reconfiguration Controller in the `Interface Protocols/Transceiver PHY` category. When you instantiate your Transceiver Reconfiguration Controller IP core the **Enable offset cancellation block** option is **On** by default. For Gen3 variants, you should also turn on adaptive equalization.

Figure 17–1 shows the connections between the Transceiver Reconfiguration Controller instance and the PHY IP Core for PCI Express instance for a ×4 variant.

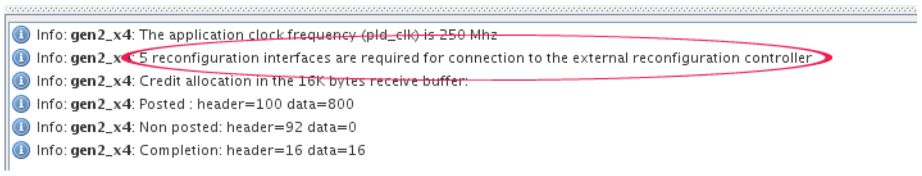
Figure 17–1. Altera Transceiver Reconfiguration Controller Connectivity



As Figure 17–1 illustrates, the `reconfig_to_xcvr[<n>70-1:0]` and `reconfig_from_xcvr[<n>46-1:0]` buses, and the `busy_xcvr_reconfig` connect the Arria V GZ Hard IP for PCI Express and Transceiver Reconfiguration Controller IP Cores. You must provide a 100–125 MHz free-running clock to the `mgmt_clk_clk` clock input of the Transceiver Reconfiguration Controller IP Core.

Initially, the Arria V GZ Hard IP for PCI Express requires a separate reconfiguration interface for each lane and each TX PLL. It reports this number in the message pane of its GUI. You must take note of this number so that you can enter it as a parameter value in the Transceiver Reconfiguration Controller parameter editor. Figure 17–2 illustrates the messages reported for a Gen2 ×4 variant. The variant requires five interfaces: one for each lane and one for the TX PLL.

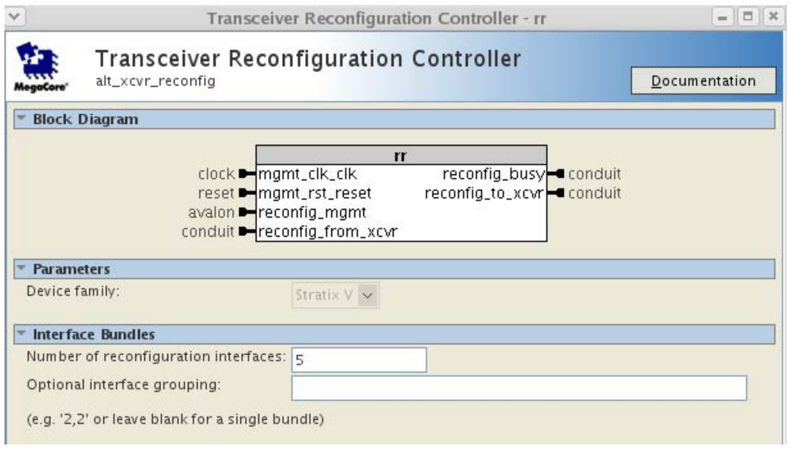
Figure 17–2. Number of External Reconfiguration Controller Interfaces



```
Info: gen2_x4: The application clock frequency (pld_clk) is 250 Mhz
Info: gen2_x4: 5 reconfiguration interfaces are required for connection to the external reconfiguration controller
Info: gen2_x4: Credit allocation in the 16K bytes receive buffer:
Info: gen2_x4: Posted : header=100 data=800
Info: gen2_x4: Non posted: header=92 data=0
Info: gen2_x4: Completion: header=16 data=16
```

When you instantiate the Transceiver Reconfiguration Controller, you must specify the required **Number of reconfiguration interfaces** as Figure 17–3 illustrates.

Figure 17–3. Specifying the Number of Transceiver Interfaces



Device family: Stratix V
Number of reconfiguration interfaces: 5
Optional interface grouping:
(e.g. '2,2' or leave blank for a single bundle)

The Transceiver Reconfiguration Controller includes an **Optional interface grouping** parameter. Arria V GZ devices include six channels in a transceiver bank. For a ×4 variant, no special interface grouping is required because all 4 lanes and the TX PLL fit in one bank.

☞ Although you must initially create a separate logical reconfiguration interface for each lane and TX PLL in your design, when the Quartus II software compiles your design, it reduces the original number of logical interfaces by merging them. Allowing the Quartus II software to merge reconfiguration interfaces gives the Fitter more flexibility in placing transceiver channels.

☞ You cannot use SignalTap to observe the reconfiguration interfaces.

Transceiver Reconfiguration Controller Connectivity for Designs Using CvP

If your design meets the following criteria:

- It enables CvP
- Includes an additional transceiver PHY that is connected to the same Transceiver Reconfiguration Controller as the PCIe Hard IP

then you must connect the PCIe `refclk` signal to the `mgmt_clk_clk` signal of the Transceiver Reconfiguration Controller and the additional transceiver PHY. In addition, if your design includes more than one Transceiver Reconfiguration Controller on the same side of the FPGA, they all must share the `mgmt_clk_clk` signal.

Learning More about Transceiver PHY Reconfiguration

For more information about using the Transceiver Reconfiguration Controller, refer to the "Transceiver Reconfiguration Controller" chapter in the *Altera Transceiver PHY IP Core User Guide* and to *Application Note 645: Dynamic Reconfiguration of PMA Controls in Stratix V Devices*. Although this application note describes dynamic reconfiguration for Stratix V devices, dynamic reconfiguration in Arria V GZ devices operates in the same manner.

18. 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 Chapter 2, Getting Started with the Arria V GZ Hard IP for PCI Express and Chapter 3, Getting Started with the Avalon-MM Arria V GZ 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.

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.

The Altera 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, Altera suggests that you obtain commercially available PCI Express verification IP and tools, or do your own extensive hardware testing or both.

The Gen3 simulation model has the following limitations:

- PIPE simulation is supported using the VCS simulator.
- The Gen3 simulation bypasses Phase 2 and Phase 3 Equalization. You must set your third-party BFM to terminate the equalization process after Phase 0 and Phase 1 complete.

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

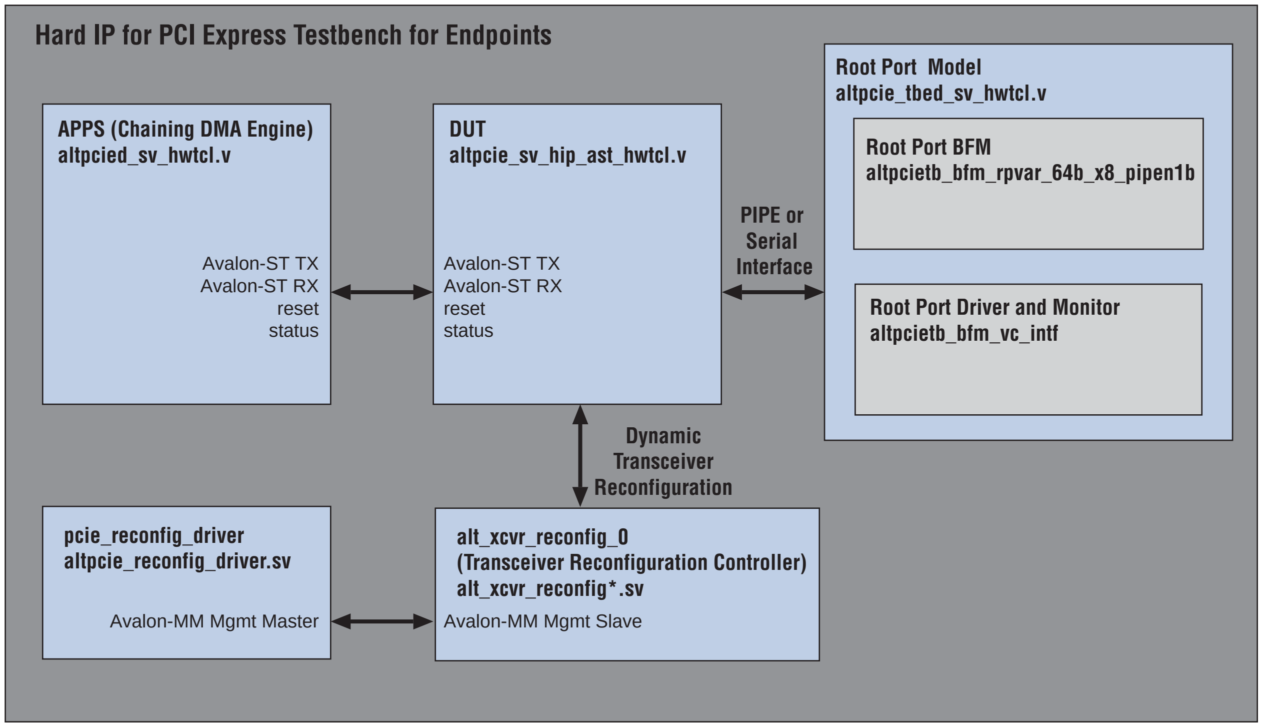
- It is unable to generate or receive Vendor Defined Messages. Some systems generate Vendor Defined Messages and the Application Layer must be designed to process them. The Hard IP block passes these messages on to the Application Layer which, in most cases should ignore them.
- It can only handle received 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 or Single Root I/O Virtualization (RS-IOV).

Endpoint Testbench

After you install the Quartus II software for 11.0, you can copy any of the five example designs from the <*install_dir*>**/ip/altera/altera_pcie/altera_pcie_hip_ast_ed /example_design** directory. You can generate the testbench from the example design as was shown in Chapter 2, Getting Started with the Arria V GZ Hard IP for PCI Express.

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. Figure 18–1 presents a high level view of the design example.

Figure 18–1. 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" on page 18–4.
- **altpcietb_bfm_top_rp.v**—This is the Root Port PCI Express BFM. For more information about this module, refer to"Root Port BFM" on page 18–20.
- **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" on page 18–18.

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.

One parameter, `serial_sim_hwtcl`, in the **altprice_tbed_sv_hwtcl.v** file, 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.

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" on page 18–18.
- **altpcietb_bfm_ep_example_chaining_pipen1b**—This is the Endpoint PCI Express mode described in the section "Chaining DMA Design Examples" on page 18–4.
- **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" on page 18–14.

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.

One parameter, `serial_sim_hwtcl`, in the **altprice_tbed_sv_hwtcl.v** file, controls whether the testbench simulates in PIPE mode or serial mode. When is set to 0, the simulation runs in PIPE mode; otherwise, it runs in serial mode.

Chaining DMA Design Examples

This design examples 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 end point or Root Port 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.

☞ 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:

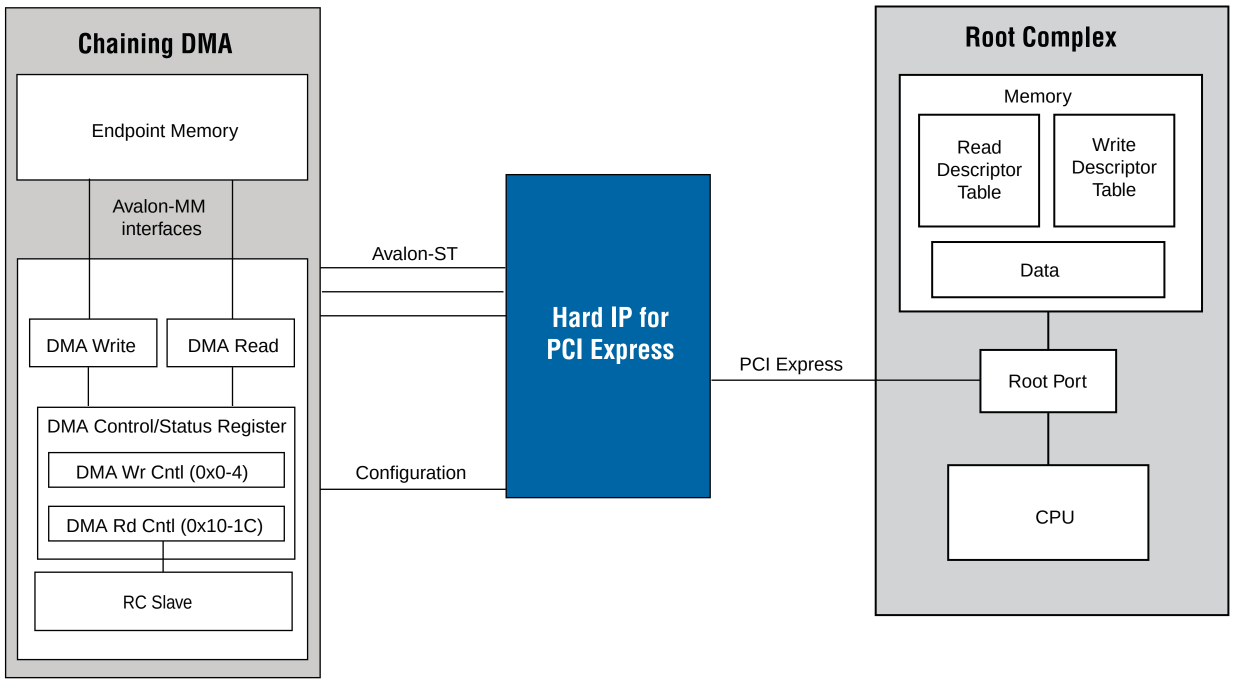
- Length of the transfer
- Address of the source
- Address of the destination
- Control bits to set the handshaking behavior between the software application or BFM driver and the chaining DMA module

☞ The chaining DMA design example only supports dword-aligned accesses. The chaining DMA design example does not support ECRC forwarding for Arria V GZ.

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

Figure 18–2 shows a block diagram of the design example connected to an external RC CPU.

Figure 18–2. Top-Level Chaining DMA Example for Simulation *(Note 1)*



Note to Figure 18–2:

(1) For a description of the DMA write and read registers, refer to Table 18–2 on page 18–10.

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 V GZ 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 V GZ 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_anme>***_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 debug ports named `test_out` and `test_in`. Refer to "Test Signals" on page 8–69 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 Altera provides five 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 II 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). **altpcierd_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.
- **altpcierd_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.
- **altpcierd_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.

Design Example 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. Table 18–1 shows the mapping.

Table 18–1. Design Example BAR Map

Memory BAR	Mapping
32-bit BAR0 32-bit BAR1 64-bit BAR1:0	Maps to 32 KByte 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.

Table 18–1. Design Example BAR Map

32-bit BAR4 32-bit BAR5 64-bit BAR5:4	Maps to 32 KByte target memory block. Use the rc_slave module to bypass the chaining DMA.
Expansion ROM BAR	Not implemented by design example; behavior is unpredictable.
I/O Space BAR (any)	Not implemented by design example; behavior is unpredictable.

Chaining DMA Control and Status Registers

The software application programs the chaining DMA control register located in the Endpoint application. Table 18–2 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.

Table 18–2. Chaining DMA Control Register Definitions *(Note 1)*

Addr *(2)*	Register Name	3124	2316	150
0x0	`DMA Wr Cntl DW0`	Control Field (refer to Table 18–3)		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		RCLAST–Idx of last descriptor to process
0x10	`DMA Rd Cntl DW0`	Control Field (refer to Table 18–3)		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		RCLAST–Idx of the last descriptor to process

Note to Table 18–2:

(1) Refer to Figure 18–2 on page 18–6 for a block diagram of the chaining DMA design example that shows these registers.

(2) This is the Endpoint byte address offset from BAR2 or BAR3.

Table 18–3 describes the control fields of the of the DMA read and DMA write control registers.

Table 18–3. 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. Table 18–7 describes the descriptor table.
[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

Table 18–3. Bit Definitions for the Control Field in the DMA Write Control Register and DMA Read Control Register

Bit	Field	Description
[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.

Table 18–4 defines the DMA status registers. These registers are read only.

Table 18–4. Chaining DMA Status Register Definitions

Addr *(2)*	Register Name	3124	2316	150
0x20	`DMA Wr Status Hi`	For field definitions refer to Table 18–5		
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 Table 18–6		
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.

Note to Table 18–4:

(1) This is the Endpoint byte address offset from BAR2 or BAR3.

Table 18–5 describes the fields of the DMA write status register. All of these fields are read only.

Table 18–5. Fields in the DMA Write Status High Register (Part 1 of 2)

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
[20:17]	Reserved	—

Table 18–5. Fields in the DMA Write Status High Register (Part 2 of 2)

Bit	Field	Description
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.

Table 18–6 describes the fields in the DMA read status high register. All of these fields are read only.

Table 18–6. 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: ■ 0001 ×1 ■ 0010 ×2 ■ 0100 ×4 ■ 1000 ×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.

Chaining DMA Descriptor Tables

Table 18–7 describes the Chaining DMA descriptor table which 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 that the chaining DMA descriptor table should not cross a 4 KByte boundary.

Table 18–7. 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

Table 18–8 shows the layout of the descriptor fields following the descriptor header.

Table 18–8. Chaining DMA Descriptor Format Map

3122	21 16	150
Reserved	Control Fields (refer to Table 18–9)	DMA Length
Endpoint Address		
RC Address Upper DWORD		
RC Address Lower DWORD		

Table 18–9 shows the layout of the control fields of the chaining DMA descriptor.

Table 18–9. Chaining DMA Descriptor Format Map (Control Fields)

2118	17	16
Reserved	`EPLAST_ENA`	`MSI`

Each descriptor provides the hardware information on one DMA transfer. Table 18–10 describes each descriptor field.

Table 18–10. 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 Table 18–7.)
`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.

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:
   - DMA read—The driver programs the chaining DMA to read data from the BFM shared memory into the Endpoint memory. The descriptor control fields (Table 18–3) are specified so that the chaining DMA completes the following steps to indicate transfer completion:

   a. The chaining DMA writes the `EPLast` bit of the "Chaining DMA Descriptor Table" on page 18–13 after finishing the data transfer for the first and last descriptors.

   b. The chaining DMA issues an MSI when the last descriptor has completed.

   - 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 (Table 18–3) are specified so that the chaining DMA completes the following steps to indicate transfer completion:

   c. The chaining DMA writes the `EPLast` bit of the "Chaining DMA Descriptor Table" on page 18–13 after completing the data transfer for the first and last descriptors.

   d. The chaining DMA issues an MSI when the last descriptor has completed.

   e. The data written back to BFM is checked against the data that was read from the BFM.

   f. The driver programs the chaining DMA to perform a test that demonstrates downstream access of the chaining DMA Endpoint memory.

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 (Table 18–11, Table 18–12, and Table 18–13).

Table 18–11. Write Descriptor 0

	Offset in BFM Shared Memory	Value	Description
DW0	0x810	82	Transfer length in dwords and control bits as described in Table 18–3 on page 18–10
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 18–12. Write Descriptor 1

	Offset in BFM Shared Memory	Value	Description
DW0	0x820	1,024	Transfer length in dwords and control bits as described in on page 18–14
DW1	0x824	0	Endpoint address
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 18–13. Write Descriptor 2

	Offset in BFM Shared Memory	Value	Description
DW0	0x830	644	Transfer length in dwords and control bits as described in Table 18–3 on page 18–10
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 (Table 18–14), into the DMA write register module.

Table 18–14. 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 Table 18–2 on page 18–10
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.

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 three descriptor tables (Table 18–15, Table 18–16, and Table 18–17).

Table 18–15. 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 18–14
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 18–16. 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 18–14
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 18–17. 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 18–14
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, (Table 18–18) into the DMA read register module.

Table 18–18. 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 Table 18–2 on page 18–10
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.

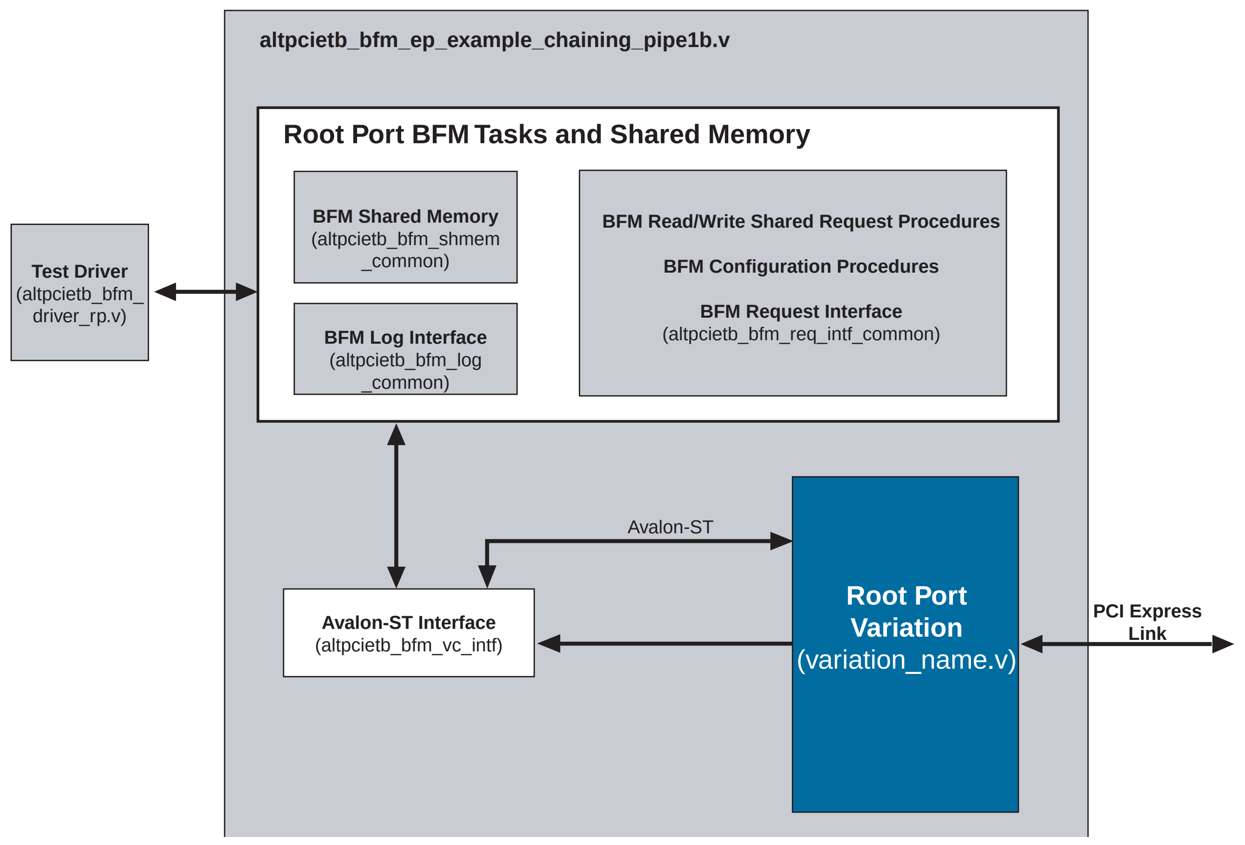
Root Port Design Example

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 V GZ 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" on page 18–14 for a detailed description.

Figure 18–3. Root Port Design Example


altpcietb_bfm_ep_example_chaining_pipe1b.v
Root Port BFM Tasks and Shared Memory
BFM Shared Memory (altpcietb_bfm_shmem _common)
BFM Log Interface (altpcietb_bfm_log _common)
BFM Read/Write Shared Request Procedures
BFM Configuration Procedures
BFM Request Interface (altpcietb_bfm_req_intf_common)
Test Driver (altpcietb_bfm_ driver_rp.v)
Avalon-ST
Avalon-ST Interface (altpcietb_bfm_vc_intf)
Root Port Variation (variation_name.v)
PCI Express Link

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.
- **altpcietb_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. (Refer to "Test Signals" on page 8–69.)

- **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 V GZ 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" on page 18–20 for a full description of this function. This version uses Avalon-ST signalling 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" on page 18–4 for a description of this module.

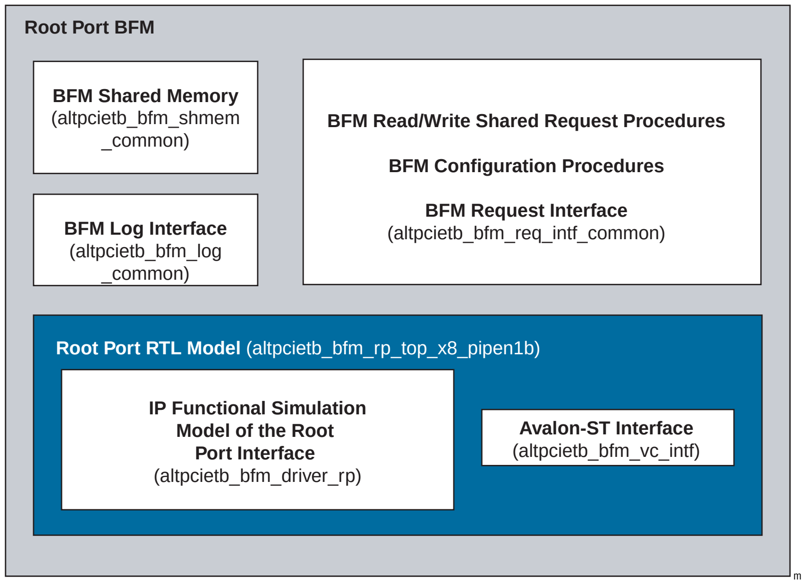
Files in subdirectory <*qsys_systemname*>**/testbench/simulation/submodules**:

- **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.

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. Figure 18–4 provides an overview of the Root Port BFM.

Figure 18–4. Root Port BFM


Root Port BFM
BFM Shared Memory (altpcietb_bfm_shmem _common)
BFM Log Interface (altpcietb_bfm_log _common)
BFM Read/Write Shared Request Procedures
BFM Configuration Procedures
BFM Request Interface (altpcietb_bfm_req_intf_common)
Root Port RTL Model (altpcietb_bfm_rp_top_x8_pipen1b)
IP Functional Simulation Model of the Root Port Interface (altpcietb_bfm_driver_rp)
Avalon-ST Interface (altpcietb_bfm_vc_intf)

The functionality of each of the modules included in Figure 18–4 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" on page 18–35.

- 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, see "BFM Read and Write Procedures" on page 18–28.
- 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, see "BFM Configuration Procedures" on page 18–34.
- 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, see "BFM Log and Message Procedures" on page 18–37.
- 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.

BFM Memory Map

The BFM shared memory is configured to be two MBytes. The BFM shared memory is mapped into the first two MBytes of I/O space and also the first two MBytes 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. For illustrations of the shared memory and I/O address spaces, refer to Figure 18–5 on page 18–25 – Figure 18–7 on page 18–27.

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.

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 KBytes 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. Refer to Figure 18–7 on page 18–27 for more information.
    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 GByte, 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 GByte address assigning memory ascending above the 4 GByte limit throughout the full 64-bit memory space. Refer to Figure 18–6 on page 18–26.

        If the `addr_map_4GB_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 GByte address and assigning memory by descending below the 4 GByte address to addresses memory as needed down to the ending address of the last 32-bit non-prefetchable BAR. Refer to Figure 18–5 on page 18–25.

    The above algorithm cannot always assign values to all BARs when there are a few very large (1 GByte 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. Table 18–19 shows how those offsets are used.

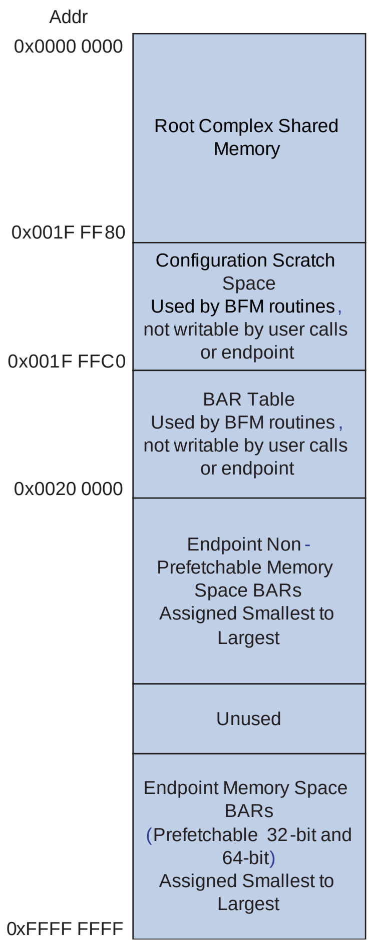
Table 18–19. BAR Table Structure

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
+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 **addr_map_4GB_limit** input parameter. If **addr_map_4GB_limit** is 1 the resulting memory space map is shown in Figure 18–5.

Figure 18–5. Memory Space Layout—4 GByte Limit


Addr
0x0000 0000
Root Complex Shared Memory
0x001F FF80
Configuration Scratch Space Used by BFM routines, not writable by user calls or endpoint
0x001F FFC0
BAR Table Used by BFM routines, not writable by user calls or endpoint
0x0020 0000
Endpoint Non-Prefetchable Memory Space BARs Assigned Smallest to Largest
Unused
Endpoint Memory Space BARs (Prefetchable 32-bit and 64-bit) Assigned Smallest to Largest
0xFFFF FFFF

If **addr_map_4GB_limit** is 0, the resulting memory space map is shown in Figure 18–6.

Figure 18–6. Memory Space Layout—No Limit

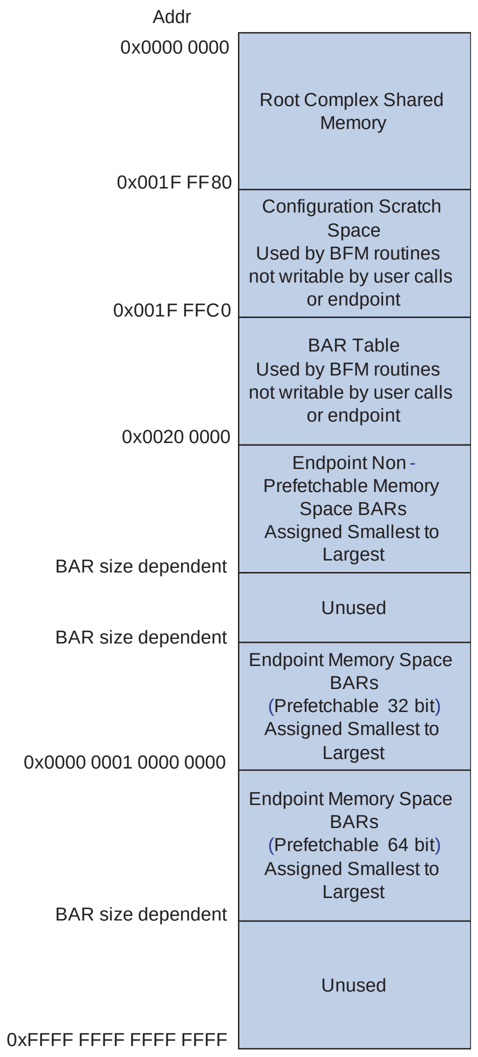
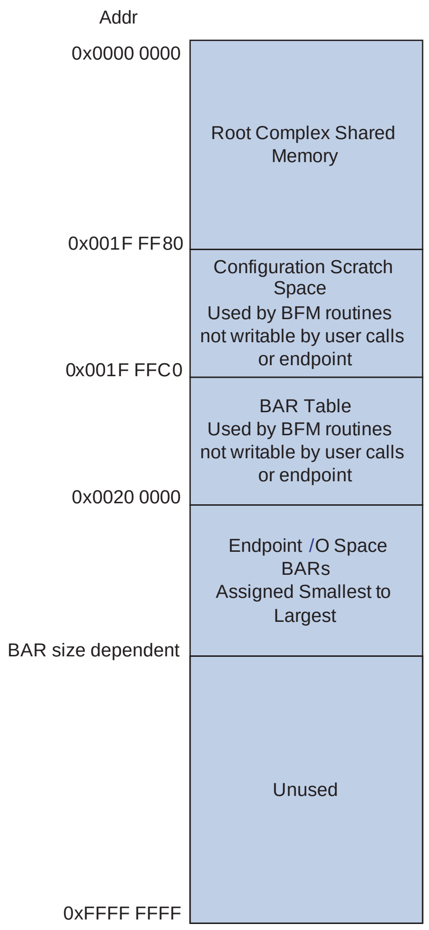

Addr
0x0000 0000
Root Complex Shared Memory
0x001F FF80
Configuration Scratch Space Used by BFM routines not writable by user calls or endpoint
0x001F FFC0
BAR Table Used by BFM routines not writable by user calls or endpoint
0x0020 0000
Endpoint Non-Prefetchable Memory Space BARs Assigned Smallest to Largest
BAR size dependent
Unused
BAR size dependent
Endpoint Memory Space BARs (Prefetchable 32 bit) Assigned Smallest to Largest
0x0000 0001 0000 0000
Endpoint Memory Space BARs (Prefetchable 64 bit) Assigned Smallest to Largest
BAR size dependent
Unused
0xFFFF FFFF FFFF FFFF

Figure 18–7 shows the I/O address space.

Figure 18–7. I/O Address Space


Addr
0x0000 0000
Root Complex Shared Memory
0x001F FF80
Configuration Scratch Space
Used by BFM routines not writable by user calls or endpoint
0x001F FFC0
BAR Table
Used by BFM routines not writable by user calls or endpoint
0x0020 0000
Endpoint /O Space BARs
Assigned Smallest to Largest
BAR size dependent
Unused
0xFFFF FFFF

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" on page 18–22.) 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.

For further details on these procedure calls, refer to the section "BFM Read and Write Procedures" on page 18–28.

BFM Procedures and Functions

This section describes the interface to all of the BFM procedures, functions, and tasks that the BFM driver uses to drive Endpoint application testing.

The last subsection describes procedures that are specific to the chaining DMA design example.

BFM Read and Write Procedures

This section describes the procedures used to read and write data among BFM shared memory, Endpoint BARs, and specified configuration registers.

The following procedures and functions are available in the Verilog HDL include file **altpcietb_bfm_driver.v**. These procedures and functions support issuing memory and configuration transactions on the PCI Express link.

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.

Table 18–20. ebfm_barwr Procedure (Part 1 of 2)

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.

Table 18–20. ebfm_barwr Procedure (Part 2 of 2)

	`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.

ebfm_barwr_imm Procedure

The `ebfm_barwr_imm` procedure writes up to four bytes of data to an offset from the specified Endpoint BAR.

Table 18–21. ebfm_barwr_imm Procedure

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.

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.

Table 18–22. ebfm_barrd_wait Procedure

Location	**altpcietb_bfm_driver_rp.v**	
Syntax	`ebfm_barrd_wait(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 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.

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.

Table 18–23. ebfm_barrd_nowt Procedure

Location	**altpcietb_bfm_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.

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.

Table 18–24. **ebfm_cfgwr_imm_wait Procedure**

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.
	`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: **Length** **Bits Written** 4 31 downto 0 3 23 downto 0 2 5 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: **Compl_Status** **Definition** 000 SC— Successful completion 001 UR— Unsupported Request 010 CRS — Configuration Request Retry Status 100 CA — Completer Abort

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.

Table 18–25. ebfm_cfgwr_imm_nowt Procedure

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: **Length** **Bits Written** 4 [31:0] 3 [23:0] 2 [15:0] 1 [7:0]

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.

Table 18–26. ebfm_cfgrd_wait Procedure

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: **Compl_Status** **Definition** 000 SC— Successful completion 001 UR— Unsupported Request 010 CRS — Configuration Request Retry Status 100 CA — Completer Abort

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.

Table 18–27. ebfm_cfgrd_nowt Procedure

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.

BFM Configuration Procedures

The following 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.

ebfm_cfg_rp_ep Procedure

The `ebfm_cfg_rp_ep` procedure configures the Root Port and Endpoint Configuration Space registers for operation. Refer to Table 18–28 for a description the arguments for this procedure.

Table 18–28. ebfm_cfg_rp_ep Procedure

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 GBytes. Any 64-bit BARs will be assigned below the 4 GByte limit.

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.

Table 18–29. ebfm_cfg_decode_bar Procedure

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.

BFM Shared Memory Access Procedures

The following procedures and functions are in the Verilog HDL include file **altpcietb_bfm_driver.v**. These procedures and functions support accessing the BFM shared memory.

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 18–30. 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

shmem_write

The `shmem_write` procedure writes data to the BFM shared memory.

Table 18–31. shmem_write Verilog HDL Task

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.
	`leng`	Length, in bytes, of data written

shmem_read Function

The `shmem_read` function reads data to the BFM shared memory.

Table 18–32. shmem_read Function

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. 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.

shmem_display Verilog HDL Function

The `shmem_display` Verilog HDL function displays a block of data from the BFM shared memory.

Table 18–33. shmem_display Verilog Function

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 18–38.

shmem_fill Procedure

The `shmem_fill` procedure fills a block of BFM shared memory with a specified data pattern.

Table 18–34. shmem_fill Procedure

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" on page 18–35.
	`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.

shmem_chk_ok Function

The `shmem_chk_ok` function checks a block of BFM shared memory against a specified data pattern.

Table 18–35. shmem_chk_ok Function

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 mis-comparing 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

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 Table 18–36.

You can suppress the display of certain message types. The default values determining whether a message type is displayed are defined in Table 18–36. 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. Table 18–36 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 18–36. 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:`
`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:`

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.

Table 18–37. ebfm_display Procedure

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 18–38.
	`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.

ebfm_log_stop_sim Verilog HDL Function

The `ebfm_log_stop_sim` procedure stops the simulation.

Table 18–38. ebfm_log_stop_sim

Location	**altpcietb_bfm_driver_rp.v**	
Syntax	Verilog VHDL: `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.

ebfm_log_set_suppressed_msg_mask Verilog HDL Function

The `ebfm_log_set_suppressed_msg_mask` procedure controls which message types are suppressed.

Table 18–39. ebfm_log_set_suppressed_msg_mask

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.

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 Table 18–36 on page 18–38.

Table 18–40. ebfm_log_set_stop_on_msg_mask

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.

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.

Table 18–41. ebfm_log_open

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.

ebfm_log_close Verilog HDL Function

The `ebfm_log_close` procedure closes the log file opened by a previous call to `ebfm_log_open`.

Table 18–42. ebfm_log_close Procedure

Location	**altpcietb_bfm_driver_rp.v**
Syntax	`ebfm_log_close`
Argument	NONE

Verilog HDL Formatting Functions

The following procedures and functions are available in the **altpcietb_bfm_driver_rp.v**. This section outlines formatting functions that are only used by Verilog HDL. All these functions take one argument of a specified length and return a vector of a specified length.

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`.

Table 18–43. himage1

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

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`.

Table 18–44. himage2

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

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`.

Table 18–45. himage4

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.

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`.

Table 18–46. himage8

Location	**altpcietb_bfm_driver_rp.v**
syntax	`string:= himage(vec)`

Table 18–46. himage8

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.

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`.

Table 18–47. himage16

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.

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`.

Table 18–48. dimage1

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.

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`.

Table 18–49. dimage2

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.

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`.

Table 18–50. dimage3

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.

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`.

Table 18–51. dimage4

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.

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.`

Table 18–52. dimage5

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.

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`.

Table 18–53. dimage6

Location	**altpcietb_bfm_log.v**
syntax	`string:= dimage(vec)`

Table 18–53. dimage6

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.

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`.

Table 18–54. dimage7

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.

Procedures and Functions Specific to the Chaining DMA Design Example

This section describes procedures that are specific to the chaining DMA design example. These procedures are located in the Verilog HDL module file **altpcietb_bfm_driver_rp.v**.

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

Table 18–55. chained_dma_test Procedure

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.

dma_rd_test Procedure

Use the `dma_rd_test` procedure for DMA reads from the Endpoint memory to the BFM shared memory.

Table 18–56. dma_rd_test Procedure

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.

dma_wr_test Procedure

Use the `dma_wr_test` procedure for DMA writes from the BFM shared memory to the Endpoint memory.

Table 18–57. dma_wr_test Procedure

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.

dma_set_rd_desc_data Procedure

Use the `dma_set_rd_desc_data` procedure to configure the BFM shared memory for the DMA read.

Table 18–58. dma_set_rd_desc_data Procedure

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.

dma_set_wr_desc_data Procedure

Use the `dma_set_wr_desc_data` procedure to configure the BFM shared memory for the DMA write.

Table 18–59. dma_set_wr_desc_data_header Procedure

Location	**altpcietb_bfm_driver_rp.v**
Syntax	`dma_set_wr_desc_data_header (bar_table, bar_num)`

Table 18–59. dma_set_wr_desc_data_header Procedure

Arguments	`bar_table`	Address of the Endpoint `bar_table` structure in BFM shared memory.
	`bar_num`	BAR number to analyze.

dma_set_header Procedure

Use the `dma_set_header` procedure to configure the DMA descriptor table for DMA read or DMA write.

Table 18–60. dma_set_header Procedure

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.

rc_mempoll Procedure

Use the `rc_mempoll` procedure to poll a given dword in a given BFM shared memory location.

Table 18–61. rc_mempoll Procedure

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 `AND`ed with the shared memory data before it is compared with `rc_data`.

msi_poll Procedure

The `msi_poll` procedure tracks MSI completion from the Endpoint.

Table 18–62. msi_poll Procedure

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.
	`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.

dma_set_msi Procedure

The `dma_set_msi` procedure sets PCI Express native MSI for the DMA read or the DMA write.

Table 18–63. dma_set_msi Procedure

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.

find_mem_bar Procedure

The `find_mem_bar` procedure locates a BAR which satisfies a given memory space requirement.

Table 18–64. find_mem_bar Procedure

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
	`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

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).

Table 18–65. dma_set_rclast Procedure

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

ebfm_display_verb Procedure

The `ebfm_display_verb` procedure calls the procedure `ebfm_display` when the global variable `DISPLAY_ALL` is set to 1.

Table 18–66. ebfm_display_verb Procedure

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 Table 18–36 on page 18–38.
	`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.

19. 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.

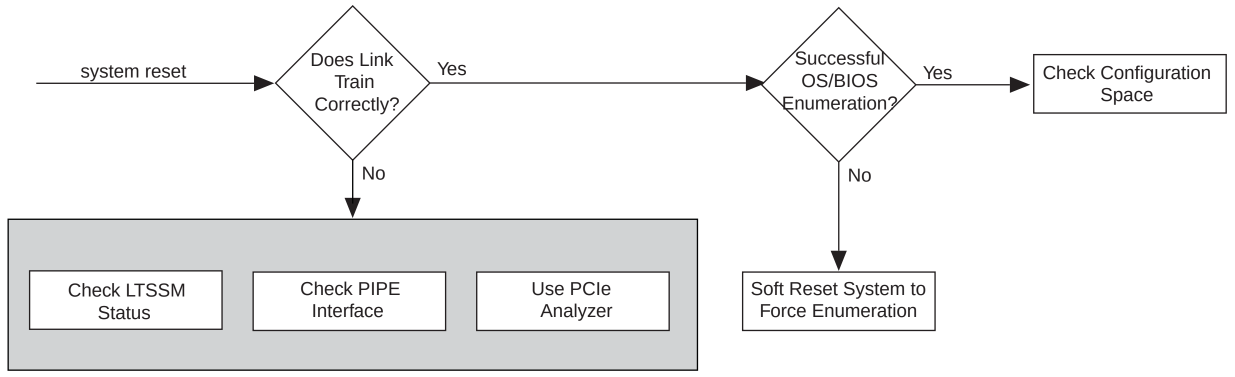
Hardware Bring-Up Issues

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. Altera recommends a systematic approach to diagnosing bring-up issues as illustrated in Figure 19–1.

Figure 19–1. Debugging Link Training Issues


system reset
Does Link Train Correctly?
Yes
No
Successful OS/BIOS Enumeration?
Yes
No
Check Configuration Space
Check LTSSM Status
Check PIPE Interface
Use PCIe Analyzer
Soft Reset System to Force Enumeration

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:

- SignalTap® II Embedded Logic Analyzer
- Third-party PCIe analyzer

You can use SignalTap II Embedded Logic Analyzer to diagnose the LTSSM state transitions that are occurring and the PIPE interface. The `ltssmstate[4:0]` 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 Signals, Status, and Link Training Signals" on page 8–28. 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[4:0]` to determine one of two cases:

- The link training fails before reaching the L0 state. Refer to Table 19–1 for possible causes of the failure to reach L0.
- The link is initially established (L0), but then stalls with `tx_st_ready` deasserted for more than 100 cycles. Refer to Table 19–2 for possible causes.
- Link training fails due to an incorrect reset sequence. Refer to Recommended Reset Sequence to Avoid Link Training Issues for the recommended reset sequence.

Table 19–1. Link Training Fails to Reach L0 (Part 1 of 2)

Possible Causes	Symptoms and Root Causes	Workarounds and Solutions
Link fails the Receiver Detect sequence.	LTSSM toggles between Detect.Quiet(0) and Detect.Active(1) states	Check the following termination settings: ■ The on-chip termination (OCT) must be set to 100 ohm, with 0.1 uF capacitors on the TX pins. ■ Link partner RX pins must also have 100 ohm termination.
Link fails with LTSSM stuck in Detect.Active state (1)	This behavior may be caused by a PMA issue if the host interrupts the Electrical Idle state as indicated by high to low transitions on the RxElecIdle (`rxelecidle`)signal when TxDetectRx=0 (`txdetectrx0`) at PIPE interface. Check if OCT is turned off by a Quartus Settings File (**.qsf**) command. PCIe requires that OCT must be used for proper Receiver Detect with a value of 100 Ohm. You can debug this issue using SignalTap II and oscilloscope.	For Arria V GZ devices, a workaround is implemented in the reset sequence.

Table 19–1. Link Training Fails to Reach L0 (Part 2 of 2)

Possible Causes	Symptoms and Root Causes	Workarounds and Solutions
Link fails with the LTSSM toggling between: Detect.Quiet (0), Detect.Active (1), and Polling.Active (2), or: Detect.Quiet (0), Detect.Active (1), and Polling.Configuration (4)	On the PIPE interface extracted from the `test_out` bus, confirm that the Hard IP for PCI Express IP Core is transmitting valid TS1 in the Polling.Active(2) state or TS1 and TS2 in the Polling.Configuration (4) state on `txdata0`. The Root Port should be sending either the TS1 Ordered Set or a compliance pattern as seen on `rxdata0`. These symptoms indicate that the Root Port did not receive the valid training Ordered Set from Endpoint because the Endpoint transmitted corrupted data on the link. You can debug this issue using SignalTap II. Refer to "PIPE Interface Signals" on page 19–8 for a list of the `test_out` bus signals.	The following are some of the reasons the Endpoint might send corrupted data: ■ Signal integrity issues. Measure the TX eye and check it against the eye opening requirements in the *PCI Express Base Specification, Rev 3.0.* Adjust the transceiver pre-emphasis and equalization settings to open the eye. ■ Bypass the Transceiver Reconfiguration Controller IP Core to see if the link comes up at the expected data rate without this component. If it does, make sure the connection to Transceiver Reconfig Controller IP Core is correct.
Link fails due to unstable `rx_signaldetect`	Confirm that `rx_signaldetect` bus of the active lanes is all 1's. If all active lanes are driving all 1's, the LTSSM state machine toggles between Detect.Quiet(0), Detect.Active(1), and Polling.Active(2) states. You can debug this issue using SignalTap II. Refer to "PIPE Interface Signals" on page 19–8 for a list of the `test_out` bus signals.	This issue may be caused by mismatches between the expected power supply to RX side of the receiver and the actual voltage supplied to the FPGA from your boards. If your PCB drives VCCT/VCCR with 1.0 V, you must apply the following command to both P and N pins of each active channel to override the default setting of 0.85 V. `set_instance_assignment -name XCVR_VCCR_VCCT_VOLTAGE 1_0V –to "pin"` Substitute the pin names from your design for `"pin"`.
Link fails because the LTSSM state machine enters Compliance	Confirm that the LTSSM state machine is in Polling.Compliance(3) using SignalTap II.	Possible causes include the following: ■ Setting `test_in[6]=1` forces entry to Compliance mode when a timeout is reached in the Polling.Active state. ■ Differential pairs are incorrectly connected to the pins of the device. For example, the Endpoint's TX signals are connected to the RX pins and the Endpoint's RX signals are to the TX pins.
Link fails because LTSSM state machine unexpectedly transitions to Recovery	A framing error is detected on the link causing LTSSM to enter the Recovery state.	In simulation, set `test_in[1]=1` to speed up simulation. This solution only solves this problem for simulation. For hardware, customer must set `test_in[1]=0`.

Link Hangs in L0 Due To Deassertion of tx_st_ready

There are many reasons that link may stop transmitting data. Table 19–2 lists some possible causes.

Table 19–2. Link Hangs in L0 (Part 1 of 2)

Possible Causes	Symptoms and Root Causes	Workarounds and Solutions
Avalon-ST signalling 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`. These errors are applicable to both simulation and hardware.	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 signalling of last two TLPs to verify correct `tx_st_sop` and `tx_st_eop` signalling.
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 V GZ Hard IP for PCI Express IP Core credit calculation is out-of-sink 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.

Table 19–2. Link Hangs in L0 (Part 2 of 2)

Possible Causes	Symptoms and Root Causes	Workarounds and Solutions
Malformed TLP is transmitted	Refer to the 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 byte enable signals violate rules for byte enables as specified in the *Avalon Interface Specifications*. ■ 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 End Point in simulation. As a result, the End Point 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.

For more information about link training, refer to the "Link Training and Status State Machine (LTSSM) Descriptions" section of *PCI Express Base Specification 3.0*.

For more information about SignalTap, refer to the *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Recommended Reset Sequence to Avoid Link Training Issues

Successful link training can only occur after the FPGA is configured and the Transceiver Reconfiguration Controller IP Core has dynamically reconfigured SERDES analog settings to optimize signal quality. For designs using CvP, link training occurs after configuration of the I/O ring and Hard IP for PCI Express IP Core. Figure 10–1 on page 10–2 shows the key signals that reset, control dynamic reconfiguration, and link training. Successful reset sequence includes the following steps:

1. Wait until the FPGA is configured as indicated by the assertion of `CONFIG_DONE` from the reconfig block controller.
2. Deassert the `mgmt_rst_reset` input to the Transceiver Reconfiguration Controller IP Core.
3. Wait for `tx_cal_busy` and `rx_cal_busy` SERDES outputs to be deasserted.
4. Deassert `pin_perst` to take the Hard IP for PCIe out of reset. For plug-in cards, the minimum assertion time for `pin_perst` is 100 ms. Embedded systems do not have a minimum assertion time for `pin_perst`.
5. Wait for the `reset_status` output to be deasserted.
6. Deassert the reset output to the Application Layer.

Setting Up Simulation

Changing the simulation parameters reduces simulation time and provides greater visibility. Depending on the variant you are simulating, the following changes may be useful when debugging:

- Use the PIPE Interface for Gen1 and Gen2 Variants
- Reduce Counter Values for Serial Simulations
- Disable the Scrambler for Gen3 Simulations

Use 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**.

Reduce Counter Values for Serial Simulations

You can accelerate simulation by reducing the value of counters whose default values are set for hardware, not simulation.

Complete the following steps to reduce counter values for simulation:

1. Open **<*work_dir*>/<*variant*>/testbench/<*variant*>_tb/simulation/submodules/altpcie_tbed_sv_hwtcl.v**.
2. Search for the string, `test_in`.
3. To reduce the value of several counters, set `test_in[0] = 1`.
4. Save **altpcie_tbed_sv_hwtcl.v**.

Disable the Scrambler for Gen3 Simulations

The 128b/130b 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_sv_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**.

Change between the Hard and Soft Reset Controller

The Hard IP for PCI Express includes both hard and soft reset control logic. By default, Gen1 ES and Gen1 and Gen2 production devices use the Hard Reset Controller. Gen2 and Gen3 ES devices and Gen3 production devices use the soft reset controller. For variants that use the hard reset controller, changing to the soft reset controller provides greater visibility.

Complete the following steps to change to the soft reset controller:

1. Open **<*work_dir*>/<*variant*>/testbench/<*variant*>_tb/simulation/submodules/variant.v**.
2. Search for the string, `hip_hard_reset_hwtcl`.
3. If `hip_hard_reset_hwtcl = 1`, the hard reset controller is active. Set `hip_hard_reset_hwtcl = 0` to change to the soft reset controller.
4. Save **variant.v**.

Using the PIPE Interface

Because the LTSSM signals reflect the behavior of one side of the PCI Express link, you may find it difficult to determine the root cause of the link issue solely by monitoring these signals. Monitoring the PIPE interface signals in addition to the `ltssmstate` bus provides greater visibility.

The PIPE interface is specified by Intel. This interface defines the MAC/PCS functional partitioning and defines the interface signals for these two sublayers. Using the SignalTap logic analyzer to monitor the PIPE interface signals provides more information about the devices that form the link.

During link training and initialization, different pre-defined Physical Layer Packets (PLPs), known as ordered sets are exchanged between the two devices on all lanes. All of these ordered sets have special symbols (K codes) that carry important information to allow two connected devices to exchange capabilities, such as link width, link data rate, lane reversal, lane-to-lane de-skew, and so on. You can track the ordered sets in the link initialization and training on both sides of the link to help you diagnose link issues. You can use SignalTap logic analyzer to determine the behavior.

Table 19–3 lists the PIPE interface signals for a two-lane simulation that you can monitor on the `test_out` bus.

Table 19–3. PIPE Interface Signals (Part 1 of 3)

Signal Name	Lane 0	Lane 1	Description
`reserved[57:0]`	`[159:102]`	`[319:262]`	—
`lanereversalenable`	`[101]`	`[261]`	When asserted, enables lanes reversal. The following encodings are defined: ■ 0: Lanes not reversed ■ 1: Lanes reversed
`eidleinfersel[2:0]`	`[100:98]`	`[260]`	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 the in 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
`txdeemph`	`[97]`	`[257]`	Transmit de-emphasis selection. The Arria V GZ 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).
`txmargin[2:0]`	`[96:94]`	`[256:254]`	Transmit V_{OD} margin selection.

Table 19–3. PIPE Interface Signals (Part 2 of 3)

Signal Name	Lane 0	Lane 1	Description
`rate[1:0]`	`[93:92]`	`[253:252]`	The 2-bit encodings have the following meanings: ■ 2’b01: Gen1 rate (2.5 Gbps) ■ 2’b10: Gen2 rate (5.0 Gbps) ■ 2’b13: Gen3 rate (8.0 Gbps) ■ 2’b00: reserved
`rxstatus0[2:0]`	`[91:89]`	`[251:249]`	Receive status <*n*>. This signal encodes receive status and error codes for the receive data stream and receiver detection. The following encodings are defined: ■ 3’b000: Received data OK. ■ 3’b001: 1 SKP added. ■ 3’b010: 1 SKP removed. ■ 3’b011: Received detected. ■ 3’b100: Both 8B/10B decode error and Receive Disparity error. ■ 3’b101: Elastic Buffer overflow. ■ 3’b110: Elastic Buffer underflow. ■ 3’b111: Reserved.
`rxelecidle0`	`[88]`	`[248]`	Indicates receiver detection of an electrical idle.
`phystatus0`	`[87]`	`[247]`	This signal communicates completion of several PHY requests.
`rxvalid0`	`[86]`	`[246]`	Indicates symbol lock and valid data on rxdata0[31:0] and rxdatak0[3:0]
`rxblkst0`	`[85]`	`[245]`	For Gen3 operation, indicates the start of a block.
`rxsynchd0[1:0]`	`[84:83]`	`[244:243]`	For Gen3 operation, specifies the block type. The following encodings are defined: ■ 2'b01: Ordered Set Block ■ 2'b10: Data Block
`rxdataskip0`	`[82]`	`[242]`	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
`rxdatak0[3:0]`	`[81:78]`	`[241:238]`	These signals show the data and control received by Hard P block from the other device.
`rxdata0[31:0]`	`[77:46]`	`[237:206]`	
`powerdown0[1:0]`	`[45:44]`	`[205:204]`	The 4 encodings of these signals have the following meanings: ■ 2’b00: Phy is transmitting data. ■ 2’b01: PHY is in electrical idle. ■ 2’b10: PHY is in loopback mode. ■ 2’b11: Illegal. Not defined.
`rxpolarity0`	`[43]`	`[203]`	When asserted, the PHY must invert the received data.

Table 19–3. PIPE Interface Signals (Part 3 of 3)

Signal Name	Lane 0	Lane 1	Description
`txcompl0`	[42]	[202]	This signal forces the running disparity to negative in compliance mode (negative COM character).
`txelecidle0`	[41]	[201]	This signal forces the TX output to electrical idle.
`txdetectrx0`	[40]	[200]	This signal tells the PHY layer to start a receive detection operation or to begin loopback.
`txblkst0`	[39]	[199]	For Gen3 operation, indicates the start of a block.
`txsynchd0[1:0]`	[38:37]	[198:197]	For Gen3 operation, specifies the block type. The following encodings are defined: ■ 2'b01: Ordered Set Block ■ 2'b10: Data Block
`txdataskip0`	[36]	[196]	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
`txdatak0[3:0]`	[35:32]	[195:192]	These signals show the data and control being transmitted from the Arria V GZ Hard IP for PCI Express to the other device.
`txdata0[31:0]`	[31:0]	[191:160]	

The *PHY Interface for PCI Express Architecture* specification is available on the Intel website (**www.intel.com**).

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.

BIOS Enumeration Issues

Both FPGA programming (configuration) and the initialization of a PCIe link require time. There is some possibility that Altera 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. To eliminate this issue, you can do a soft reset of the system to retain the FPGA programming while forcing the OS/BIOS to repeat its enumeration.

A. Transaction Layer Packet (TLP) Header Formats

TLP Packet Format without Data Payload

Table A–1 through A–2 show the header format for TLPs without a data payload.

Table A–1. 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																															

Table A–2. 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																															

Table A–3. 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	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

Table A–4. 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

Table A–5. 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																															

Table A–6. 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																															

Table A–7. 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																															

Notes to Table A–7:

(1) Not supported in Avalon-MM.

Table A–8. 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																															

Table A–9. 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											

Table A–9. Completion Locked without Data

Byte 8	Requester ID	Tag	0	Lower Address
Byte 12	Reserved			

TLP Packet Format with Data Payload

Table A–10 through A–4 show the content for TLPs with a data payload.

Table A–10. 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																															

Table A–11. 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

Table A–12. 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																															

Table A–13. 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																															

Table A–14. 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																															

Table A–15. 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																															

Table A–16. 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																															

Additional Information

This chapter provides additional information about the document and Altera.

Revision History

The table below displays the revision history for the chapters in this User Guide.

Date	Version	Changes Made
May 2013	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 embeds a DMA engine with a 256-bit datapath. ■ Added support for Gen3 PIPE simulation. ■ 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 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.
November 2012	1.0	Initial release.

How to Contact Altera

To locate the most up-to-date information about Altera products, refer to the following table.

Contact (1)	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com

Contact [1]	Contact Method	Address
Product literature	Website	www.altera.com/literature
Nontechnical support (general)	Email	nacomp@altera.com
(software licensing)	Email	authorization@altera.com

Note to Table:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicate command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, **\qdesigns** directory, **D:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicate document titles. For example, *Stratix IV Design Guidelines*.
italic type	Indicates variables. For example, *n* + 1. Variable names are enclosed in angle brackets (< >). For example, <*file name*> and <*project name*>**.pof** file.
Initial Capital Letters	Indicate keyboard keys and menu names. For example, the Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections in a document and titles of Quartus II Help topics. For example, "Typographic Conventions."
`Courier type`	Indicates signal, port, register, bit, block, and primitive names. For example, `data1`, `tdi`, and `input`. The suffix n denotes an active-low signal. For example, `resetn`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
↵	An angled arrow instructs you to press the Enter key.
1., 2., 3., and a., b., c., and so on	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■ ■	Bullets indicate a list of items when the sequence of the items is not important.
☞	The hand points to information that requires special attention.
?	The question mark directs you to a software help system with related information.
👣	The feet direct you to another document or website with related information.
🎥	The multimedia icon directs you to a related multimedia presentation.
⚠ CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
⚠ WARNING	A warning calls attention to a condition or possible situation that can cause you injury.

Visual Cue	Meaning
✉	The envelope links to the Email Subscription Management Center page of the Altera website, where you can sign up to receive update notifications for Altera documents.
💬	The feedback icon allows you to submit feedback to Altera about the document. Methods for collecting feedback vary as appropriate for each document.