

PCI Express Compiler User Guide



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

Compiler Version: 9.0
Document Date: March 2009

Copyright © 2009 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

Chapter 1. About This Compiler

New Features in PCI Express Compiler Version 9.0	1-1
Release Information	1-2
Device Family Support	1-3
Features	1-3
General Description	1-4
MegaCore Function and Device Families with PCI Express Hard IP	1-4
MegaCore Functions and Device Families with High Speed Serial Interface (HSSI) Transceivers	1-6
External PHY Support	1-7
Debug Features	1-7
Testbench and Example Designs	1-8
BFM	1-8
Root Port Design Example	1-8
Endpoint Design Example	1-9
MegaCore Function Verification	1-9
Simulation Environment	1-10
Compatibility Testing Environment	1-10
Performance and Resource Utilization	1-10
Avalon-ST Interface	1-10
Hard IP Implementation	1-11
Soft IP Implementation	1-12
Descriptor/Data Interface	1-13
Arria GX Devices	1-14
Cyclone III Family	1-14
Stratix II GX Devices	1-15
Stratix III Family	1-15
Stratix IV Family	1-16
Avalon-MM Interface	1-16
Arria GX Devices	1-17
Cyclone III Family	1-17
Stratix II GX Devices	1-18
Stratix III Family	1-18
Stratix IV Family	1-18
Recommended Speed Grades	1-19
Installation and Licensing	1-20
OpenCore Plus Evaluation (Not Required for Hard IP)	1-21
OpenCore Plus Time-Out Behavior (Not Required for Hard IP)	1-22

Chapter 2. Getting Started

Design Flow	2-1
MegaWizard Plug-In Manager Design Flow	2-3
Specify Parameters	2-3
Simulate the Design	2-5
Instantiate the MegaCore Function	2-5
Constrain, Compile, and Program	2-5
SOPC Builder Design Flow	2-6
Specify Parameters	2-6
Complete the SOPC Builder System	2-6

Chapter 3. Parameter Settings

System Settings	3-1
PCI Registers	3-3
Capabilities Parameters	3-5
Buffer Setup	3-8
Power Management	3-10
Avalon-MM Configuration	3-12

Chapter 4. Functional Description

Architecture	4-1
Application Interfaces	4-3
Avalon-ST Application Interface	4-3
Descriptor/Data Application Interface (Soft IP Only)	4-7
Avalon-MM Interface	4-8
Transaction Layer	4-9
Transmit Virtual Channel Arbitration	4-11
Configuration Space	4-11
Transaction Layer Routing Rules	4-12
Receive Buffer Reordering	4-12
Data Link Layer	4-13
Physical Layer	4-15
Physical Layer Architecture	4-16
Lane Initialization and Reversal	4-18
PCI Express Avalon-MM Bridge	4-19
Avalon-MM-to-PCI Express Write Requests	4-21
Avalon-MM-to-PCI Express Upstream Read Requests	4-22
PCI Express-to-Avalon-MM Read Completions	4-22
PCI Express-to-Avalon-MM Downstream Write Requests	4-22
PCI Express-to-Avalon-MM Downstream Read Requests	4-22
Avalon-MM-to-PCI Express Read Completions	4-23
PCI Express-to-Avalon-MM Address Translation	4-23
Avalon-MM-to-PCI Express Address Translation	4-24
Generation of PCI Express Interrupts	4-25
Generation of Avalon-MM Interrupts	4-27
Analyzing Throughput	4-27
Throughput of Posted Writes	4-27
Throughput of Non-Posted Reads	4-30
PCI Express Reconfiguration Block	4-32
Configuration Space Register Content	4-40
PCI Express Avalon-MM Bridge Control Register Content	4-44
Avalon-MM to PCI Express Interrupt Registers	4-45
PCI Express Mailbox Registers	4-46
Avalon-MM-to-PCI Express Address Translation Table	4-47
PCI Express to Avalon-MM Interrupt Status and Enable Registers	4-48
Avalon-MM Mailbox Registers	4-49
Active State Power Management (ASPM)	4-50
Exit Latency	4-50
Acceptable Latency	4-51

Error Handling	4-52
Physical Layer	4-52
Data Link Layer	4-52
Transaction Layer	4-53
ECRC	4-55
ECRC on the Rx Path	4-55
ECRC on the Tx Path	4-56
Error Logging and Reporting	4-56
Data Poisoning	4-56
Supported Message Types	4-57
Stratix GX PCI Express Compatibility	4-59
Clocking	4-59
MegaWizard Plug-In Manager Design Flow Clocking—Hard IP Implementation	4-59
p_clk	4-60
core_clk, core_clk_out	4-61
pld_clk	4-61
MegaWizard Plug-In Manager Design Flow Clocking—Soft IP Implementation	4-61
Generic PIPE PHY Clocking Configuration	4-62
Stratix GX PHY with 100 MHz Reference Clock	4-62
Stratix GX PHY with 125 MHz Reference Clock	4-63
Stratix GX PHY with 156.25 MHz Reference Clock	4-64
Arria GX, Stratix II GX, or Stratix IV PHY ×1 and ×4 100 MHz Reference Clock	4-64
Stratix II GX PHY ×8 with 100 MHz Reference Clock	4-65
SOPC Builder Design Flow Clocking—Soft IP Implementation	4-66
Separate PCI Express and Avalon Clock Domains	4-66
Multiple Core Configurations	4-67

Chapter 5. Signals

Avalon-ST Interface	5-1
64- or 128-Bit Avalon-ST Rx Port	5-6
64- or 128-Bit Avalon-ST Tx Port	5-10
Mapping of Avalon-ST Packets to PCI Express	5-12
Root Port Mode Configuration Requests	5-16
ECRC Forwarding	5-16
Clock Signals—Hard IP Implementation	5-16
Clock Signals—Soft IP Implementation	5-17
Reset Signals	5-17
ECC Error Signals	5-20
PCI Express Interrupts for Endpoints	5-20
PCI Express Interrupts for Root Ports	5-25
Configuration Space Signals—Hard IP Implementation	5-26
Configuration Space Signals—Soft IP Implementation	5-29
LMI Signals—Hard IP Implementation	5-30
LMI Read Operation	5-31
LMI Write Operation	5-32
PCI Express Reconfiguration Block Signals—Hard IP Implementation	5-32
Power Management Signals	5-32
Completion Side Band Signals	5-35

Descriptor/Data Interface	5-36
Receive Operation Interface Signals	5-38
Receive Datapath Interface Signals	5-38
Transaction Examples Using Receive Signals	5-41
Dependencies Between Receive Signals	5-47
Transmit Operation Interface Signals	5-48
Transmit Datapath Interface Signals	5-48
Transaction Examples Using Transmit Signals	5-53
Completion Interface Signals for Descriptor/Data Interface	5-60
Avalon-MM Application Interface	5-61
32-Bit Non-bursting Avalon-MM CRA Slave Signals	5-63
64-Bit Bursting Rx Avalon-MM Master Signals	5-63
64-Bit Bursting Tx Avalon-MM Slave Signals	5-64
Clock Signals	5-65
Reset Signals	5-65
Physical Layer Interface Signals	5-66
Transceiver Control	5-67
Serial Interface Signals	5-68
PIPE Interface Signals	5-68
Test Signals	5-70

Chapter 6. External PHYs

External PHY Support	6-1
16-bit SDR Mode	6-1
16-bit SDR Mode with a Source Synchronous TxClk	6-2
8-bit DDR Mode	6-3
8-bit DDR with a Source Synchronous TxClk	6-4
8-bit SDR Mode	6-5
8-bit SDR with a Source Synchronous TxClk	6-6
16-bit PHY Interface Signals	6-7
8-bit PHY Interface Signals	6-9
Selecting an External PHY	6-10
External PHY Constraint Support	6-11

Chapter 7. Testbench and Design Example

Endpoint Testbench	7-2
Root Port Testbench	7-4
Chaining DMA Design Example	7-6
Design Example BAR/Address Map	7-13
Chaining DMA Control and Status Registers	7-14
Chaining DMA Descriptor Tables	7-16
Test Driver Module	7-18
DMA Write Cycles	7-19
DMA Read Cycles	7-21
Root Port Design Example	7-22
Root Port BFM	7-26
BFM Memory Map	7-28
Configuration Space Bus and Device Numbering	7-28
Configuration of Root Port and Endpoint	7-28
Issuing Read and Write Transactions to the Application Layer	7-33

BFM Procedures and Functions	7-34
BFM Read and Write Procedures	7-34
ebfm_barwr Procedure	7-35
ebfm_barwr_imm Procedure	7-35
ebfm_barrd_wait Procedure	7-36
ebfm_barrd_nowt Procedure	7-36
ebfm_cfgwr_imm_wait Procedure	7-37
ebfm_cfgwr_imm_nowt Procedure	7-38
ebfm_cfgrd_wait Procedure	7-39
ebfm_cfgrd_nowt Procedure	7-39
BFM Performance Counting	7-40
ebfm_start_perf_sample Procedure	7-40
ebfm_disp_perf_sample Procedure	7-41
BFM Configuration Procedures	7-41
ebfm_cfg_rp_ep Procedure	7-42
ebfm_cfg_decode_bar Procedure	7-43
BFM Shared Memory Access Procedures	7-43
Shared Memory Constants	7-43
shmem_write	7-44
shmem_read Function	7-44
shmem_display VHDL Procedure or Verilog HDL Function	7-44
shmem_fill Procedure	7-45
shmem_chk_ok Function	7-45
BFM Log and Message Procedures	7-46
Log Constants	7-46
ebfm_display VHDL Procedure or Verilog HDL Function	7-47
ebfm_log_stop_sim VHDL Procedure or Verilog HDL Function	7-48
ebfm_log_set_suppressed_msg_mask Procedure	7-48
ebfm_log_set_stop_on_msg_mask Procedure	7-48
ebfm_log_open Procedure	7-49
ebfm_log_close Procedure	7-49
VHDL Formatting Functions	7-49
himage (std_logic_vector) Function	7-49
himage (integer) Function	7-50
Verilog HDL Formatting Functions	7-50
himage1	7-50
himage2	7-50
himage4	7-51
himage8	7-51
himage16	7-51
dimage1	7-52
dimage2	7-52
dimage3	7-52
dimage4	7-53
dimage5	7-53
dimage6	7-53
dimage7	7-54
Procedures and Functions Specific to the Chaining DMA Design Example	7-54
chained_dma_test Procedure	7-54
dma_rd_test Procedure	7-55
dma_wr_test Procedure	7-55
dma_set_rd_desc_data Procedure	7-55
dma_set_wr_desc_data Procedure	7-55
dma_set_header Procedure	7-56

rc_mempoll Procedure	7-56
msi_poll Procedure	7-57
dma_set_msi Procedure	7-57
find_mem_bar Procedure	7-58
dma_set_rclast Procedure	7-58
ebfm_display_verb Procedure	7-58

Chapter 8. SOPC Builder Design Example

Create a Quartus II Project	8-2
Run SOPC Builder	8-3
Parameterize the PCI Express MegaCore Function	8-4
Add the Remaining Components to the SOPC Builder System	8-5
Complete the Connections in SOPC Builder	8-5
Generate the SOPC Builder System	8-8
Simulate the SOPC Builder System	8-8
Compile the Design	8-10
Program a Device	8-11
Recommended Top-Down Incremental Compilation Flow	8-11

Appendix A. Transaction Layer Packet Header Formats

Packet Format without Data Payload	A-1
Packet Format with Data Payload	A-4

Appendix B. Test Port Interface Signals

Hard IP Test Signals	B-1
Hard IP Implementation—test_in	B-1
Hard IP Implementation—test_out	B-4
Transaction Layer	B-5
Data Link Layer Outputs	B-8
PHY-MAC Outputs	B-12
PHY PCS layer	B-15
CDC Test Outputs	B-15
Mixed Mode Test Outputs	B-16
Receive FC Test Outputs	B-20
Transmit FC Test Outputs	B-20
Request ID and Error Signal Test Outputs	B-21
Receive DLL Data Test Outputs	B-22
Transmit DLL Data Test Outputs	B-22
Miscellaneous Test Outputs	B-23
Soft IP MegaCore Function	B-23
Soft IP Implementation x1 and x4 test_out	B-24
Soft IP x8 MegaCore Function test_out	B-38
Soft IP Implementation x1, x4, and x8 test_in	B-40

Appendix C. Incremental Compile Module for Descriptor/Data Examples

ICM Features	C-1
ICM Functional Description	C-2
<variation_name>_icm Partition	C-2
ICM Block Diagram	C-3
Rx Datapath	C-5
Tx Datapath	C-5
MSI Datapath	C-5
Sideband Datapath	C-5
ICM Files	C-5
ICM Application-Side Interface	C-6
Rx Ports	C-6
Tx Ports	C-8
Sideband Interface	C-10

Additional Information	About-1
Revision History	About-1
How to Contact Altera	About-5
Typographic Conventions	About-5

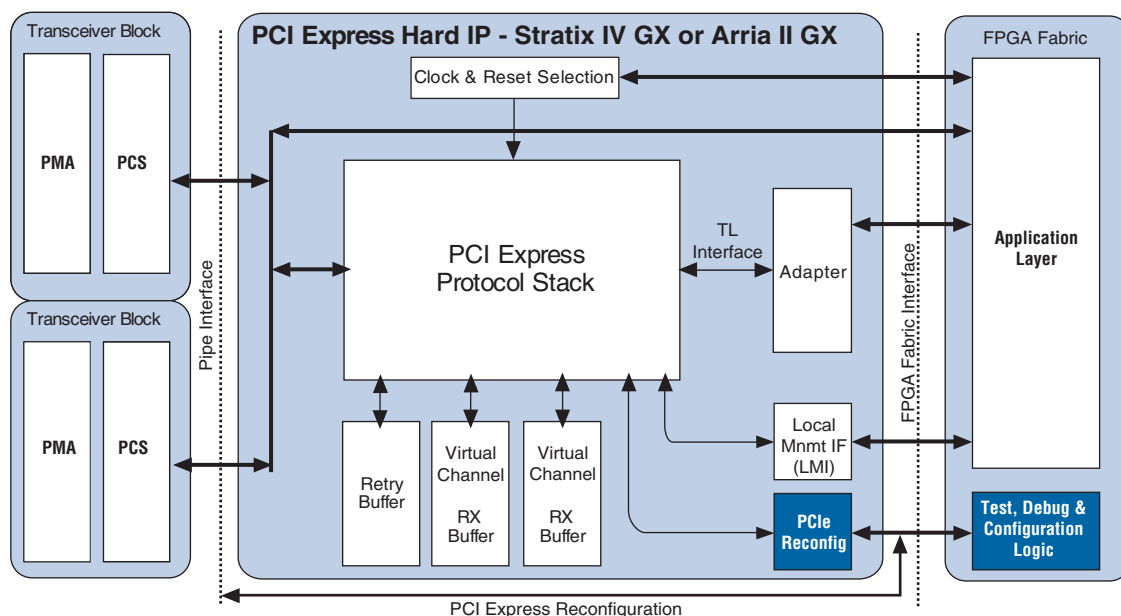
New Features in PCI Express Compiler Version 9.0

The Altera® PCI Express Compiler provides the following new features in the 9.0 release:

- Support for the PCI Express Gen1 protocol in both the hard and soft IP implementations in Arria® II GX devices
- Support for the PCI Express Gen2 protocol in Stratix® IV GX devices
- Support for the Avalon® Memory-Mapped (Avalon-MM) interface for the hard IP implementation in SOPC Builder, in addition to the previously available soft IP implementation
- PCI Express reconfiguration block for the hard IP implementation to dynamically update read-only configuration space registers
- Enhancements to the Altera-provided design example, including default instantiation of the RC slave module and modules to exercise ECRC and PCI Express hard IP dynamic reconfiguration functionality

The Arria II GX device support includes ×1, ×4, and ×8 configurations in the hard IP implementation and ×1 and ×4 configurations in soft IP implementation. With this additional device support, you now have a low-cost, hard IP solution for the PCI Express protocol. With dynamic reconfiguration of the hard IP implementation available, you can change the default values of configuration space read-only registers without powering down your system. [Figure 1–1](#) highlights the new reconfiguration interface.

Figure 1–1. PCI Express Hard IP High-Level Block Diagram (Note 1)



Note to Figure 1–1:

- (1) The Arria II GX implementation has a single virtual channel.

The hard IP implementation includes all of the required and most optional features of the specification for the transaction, data link, and physical layers. If you target a Stratix IV GX device, you can parameterize the hard IP to include one or two virtual channels with either Gen1 or Gen2 protocol support. Depending upon the Stratix IV GX device you choose, two or four instances of the hard IP PCI Express MegaCore function are available.

The Arria II GX device includes a single instance of the hard IP implementation. In either device family, you can configure the PCI Express MegaCore function as either a root port or end point in $\times 1$, $\times 4$, and $\times 8$ modes. You can also include instances of both the soft and hard IP PCI Express MegaCore function in a single device.

Release Information

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

Table 1-1. PCI Express Compiler Release Information

Item	Description
Version	9.0
Release Date	March 2009
Ordering Codes	IP-PCIE/1 IP-PCIE/4 IP-PCIE/8 IP-AGX-PCIE/1 IP-AGX-PCIE/4 No ordering code is required for the hard IP implementation.
Product IDs ■ Hard IP Implementation ■ Soft IP Implementation	FFFF $\times 1$ -00A9 $\times 4$ -00AA $\times 8$ -00AB
Vendor ID ■ Hard IP Implementation ■ Soft IP Implementation	6AF7 6A66

Altera verifies that the current version of the Quartus® II software compiles the previous version of each MegaCore function. Any exceptions to this verification are reported in the *MegaCore IP Library Release Notes and Errata*. Altera does not verify compilation with MegaCore function versions older than one release.

Device Family Support

MegaCore functions provide either full or preliminary support for target Altera device families:

- *Full support* means the MegaCore function meets all functional and timing requirements for the device family and can be used in production designs.
- *Preliminary support* means the MegaCore function meets all functional requirements, but may still be undergoing timing analysis for the device family; it can be used in production designs with caution.

Table 1–2 shows the level of support offered by the PCI Express Compiler to each Altera device family.

Table 1–2. Device Family Support

Device Family	Support
Arria GX	Full
Arria II GX	Preliminary
Cyclone® II	Full
Cyclone III	Full
HardCopy® II	Full
HardCopy III	Preliminary
HardCopy IV E	Preliminary
Stratix® GX	Full
Stratix II	Full
Stratix II GX	Full
Stratix III	Full
Stratix IV	Preliminary
Other device families	No support

Features

Extensive support across multiple device families

- Hard IP implementation
 - *PCI Express Base Specification 1.1 or 2.0* support for Stratix IV GX devices
 - *PCI Express Base Specification 1.1* support for Arria II GX devices
- Soft IP implementation
 - *PCI Express Base Specification 1.0a or 1.1* support for Arria II GX and Stratix IV GX devices
 - Many other device families supported. Refer to Table 1–2

- Feature rich
 - Support for $\times 1$, $\times 2$, $\times 4$, and $\times 8$ configurations ($\times 2$ lane operation is supported by down configuring an initial $\times 4$ link)
 - Optional end-to-end cyclic redundancy code (ECRC) generation and checking and advanced error reporting (AER) for high reliability applications
 - Extensive maximum payload support:
 - Stratix IV GX—Up to 2 KBytes (128, 256, 512, 1,024, or 2,045 bytes)
 - Arria II GX—Up to 512 bytes (128, 256, 512 bytes)
- Easy to use
 - Substantial on-chip resource savings and guaranteed timing closing using the PCI Express hard IP implementation
 - Easy adoption with no license requirement for the hard IP implementation
 - Example designs to get started
 - Easy GUI-based configuration
 - SOPC Builder support

General Description

The PCI Express Compiler generates customized PCI Express MegaCore functions you use to design PCI Express root ports or endpoints, including non-transparent bridges, or truly unique designs combining multiple PCI Express components in a single Altera device. The PCI Express MegaCore functions implement all required and most optional features of the PCI Express specification for the transaction, data link, and physical layers.

MegaCore Function and Device Families with PCI Express Hard IP

If you target a Stratix IV GX or Arria II GX device, you can parameterize the MegaCore function to include a full hard PCI Express stack including the following layers:

- Physical (PHY)
- Media Access Control (MAC)
- Physical Coding Sublayer (PCS)
- Physical Media Attachment (PMA)
- Data link
- Transaction

Optimized for Altera devices, the hard IP implementation supports all memory, I/O, configuration, and message transactions. The MegaCore functions have a highly optimized application interface to achieve maximum effective throughput. Because the compiler is parameterizeable, you can customize the MegaCore functions to meet your design requirements.

Table 1-3 lists the configurations that are available for the PCI Express hard IP implementation.

Table 1-3. PCI Express Hard IP Configurations for the PCI Express Compiler in 9.0

Device	Generation	×1	×4	×8
Avalon Streaming (Avalon-ST) Interface using MegaWizard Plug-In Manager Design Flow				
Stratix IV GX	Gen1	yes	yes	yes
	Gen2	yes	yes	yes
Arria II GX	Gen1	yes	yes	yes (1)
	Gen2	no	no	no
Avalon-MM Interface using SOPC Builder Design Flow				
Stratix IV GX	Gen1	yes	yes	no
	Gen2	yes	no	no
Arria II GX	Gen1	yes	yes	no
	Gen2	no	no	no

Note to Table 1-3:

(1) The ×8 support uses a 128-bit bus at 125 MHz.

The PCI Express Compiler allows you to select MegaCore functions that support ×1, ×4, or ×8 operation (Table 1-4 on page 1-7) that are suitable for either root port or endpoint applications. You can use the MegaWizard Plug-In Manager or SOPC Builder to customize the MegaCore function. Figure 1-2 shows a relatively simple application that includes two PCI Express MegaCore functions, one configured as a root port and the other as an endpoint.

Figure 1-2. PCI Express Application with a Single Root Port and Endpoint

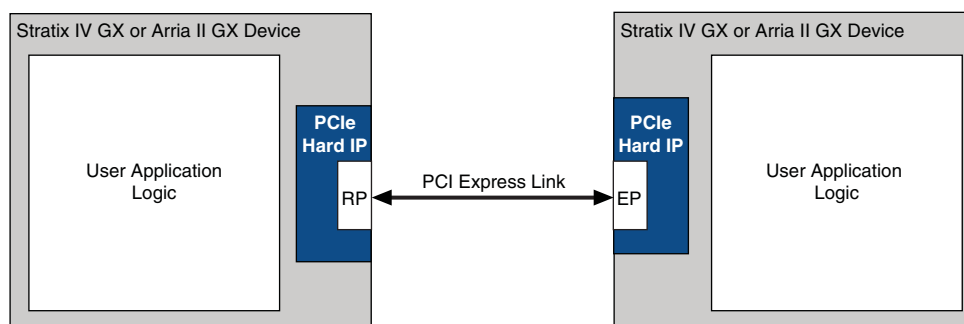
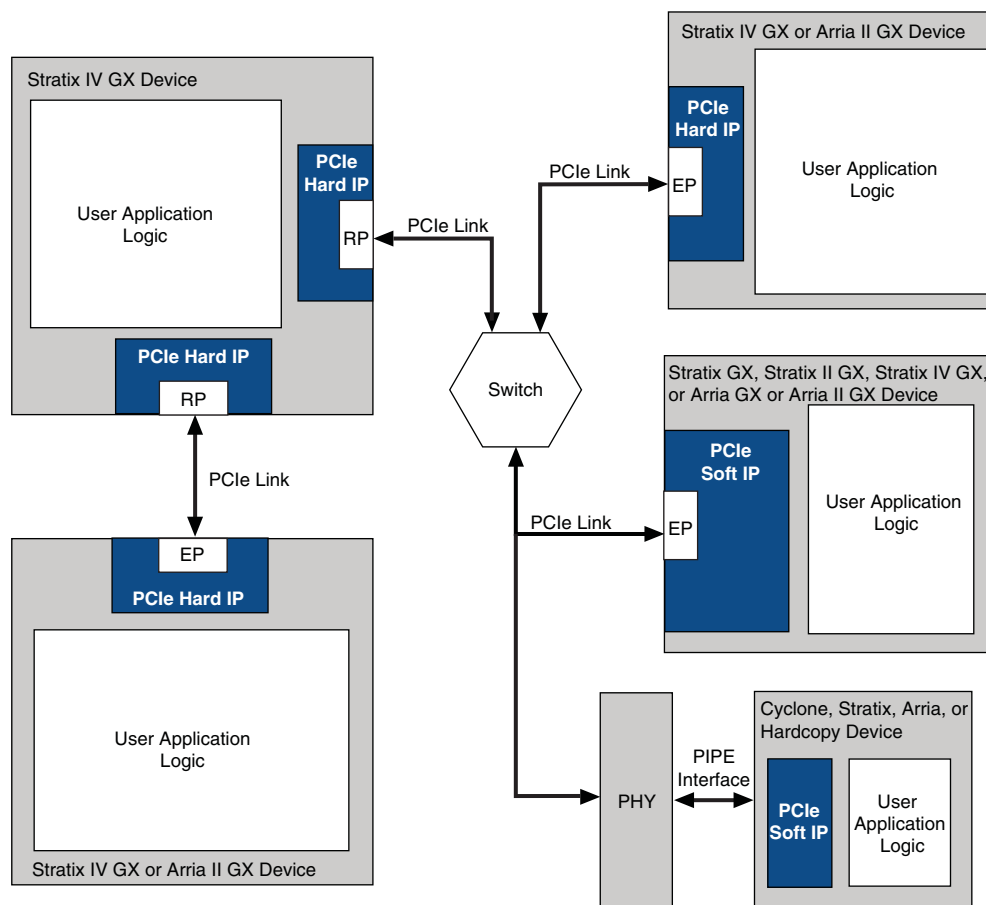


Figure 1-3 illustrates a heterogeneous topology, including a Stratix IV GX or Arria II GX device that includes two PCI Express root ports. One root port connects directly to a second Stratix IV GX or Arria II device. The second root port connects to a switch that multiplexes among three PCI Express endpoints.

Figure 1-3. PCI Express Application Including Stratix IV GX or Arria II GX Devices with Two Root Ports (Note 1)**Note to Figure 1-3:**

(1) Altera does not recommend Stratix family devices for new designs.

MegaCore Functions and Device Families with High Speed Serial Interface (HSSI) Transceivers

If you target the MegaCore function for Arria GX, Arria II GX, Stratix GX, Stratix II GX, or Stratix IV GX devices, you can parameterize the MegaCore function to include a complete PHY layer, including the MAC, PCS, and PMA layers. If you target other device architectures, the PCI Express Compiler generates the MegaCore function with the Intel-designed PIPE interface, making the MegaCore function usable with other PIPE-compliant external PHY devices.

Table 1-4. Single and Multi-Lane Operation in Devices with HSSI Transceivers (Note 1)

Device Family	×1	×4	×8
Stratix IV GX hard IP—Gen1 and Gen2	Yes (2)	Yes (2)	Yes (3)
Stratix IV soft IP—Gen1	Yes	Yes	No
Arria II GX—Gen1 Hard IP Implementation	Yes	Yes	Yes
Arria II GX—Gen1 Soft IP Implementation	Yes	Yes	No
Arria GX	Yes	Yes	No
Stratix II GX	Yes	Yes	Yes
Stratix GX	Yes (4)	Yes (4)	No

Notes to Table 1-4:

- (1) Refer to Table 2-1 on page 2-1 for a list of features available in the different implementations.
- (2) Not available in -4 speed grade. Requires -2 or -3 speed grade.
- (3) Gen2 is only available in the fastest speed grade.
- (4) Altera does not recommend using Stratix GX for new PCI Express designs. Refer to “Stratix GX PCI Express Compatibility” on page 4-59.

The MegaCore functions have a highly optimized application interface to achieve maximum effective throughput. Because the compiler is parameterizeable, you can customize the MegaCore functions to meet your design requirements.

For example, the MegaCore functions can support up to 4 virtual channels for ×1 or ×4 configurations, or up to 2 channels for ×8 configurations in Stratix IV GX devices. You can also customize the payload size, buffer sizes, and configuration space (base address registers support and other registers). Additionally, the PCI Express Compiler supports end-to-end cyclic redundancy code (ECRC) and advanced error reporting for ×1, ×4, and ×8 configurations.

External PHY Support

PCI Express MegaCore functions support a wide range of PHYs, including the TI XIO1100 PHY in 8-bit DDR/SDR mode or 16-bit SDR mode; Philips PX1011A for 8-bit SDR mode, a serial PHY, and a range of custom PHYs using 8-bit/16-bit SDR with or without source synchronous transmit clock modes and 8-bit DDR with or without source synchronous transmit clock modes. You can constrain Tx I/Os by turning on the **Fast Output Enable Register** option in the GUI, or by editing this setting in the Quartus II Settings File (.qsf). This constraint ensures fastest t_{CO} timing.

Debug Features

The PCI Express MegaCore functions also include debug features that allow observation and control of the MegaCore functions. These additional inputs and outputs help with faster debugging of system-level problems.

Testbench and Example Designs

The PCI Express Compiler includes two testbenches that allow you to select either an endpoint or a root port as the device under test (DUT). When the endpoint is the DUT, a basic root port bus functional model (BFM) and a high performance chaining DMA design example provide the stimulus. When the root port is the DUT, it is programmed to implement the same chaining DMA example and an endpoint BFM carries out the specified transactions. The following sections describe these modules.

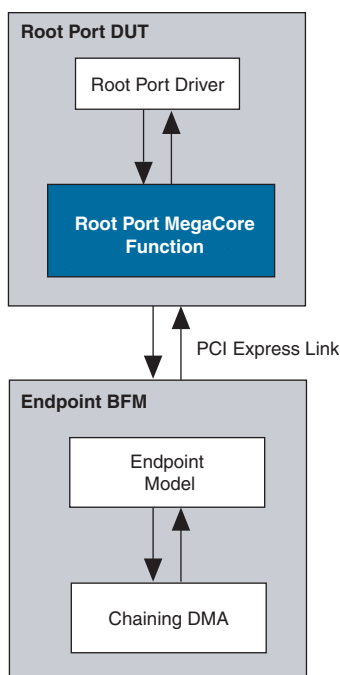
BFM

The basic root port BFM incorporates example driver procedures and an IP functional simulation model of a root port. The endpoint BFM incorporates example driver procedures and an IP functional simulation model of an endpoint.

Root Port Design Example

The root port design example illustrates the application interface to the PCI Express MegaCore function and is delivered as clear-text source-code (Verilog HDL) suitable for simulation. In addition, a synthesizable module containing the MegaCore function is provided for compilation in the Quartus II software. This design example is available when you specify the root port hard IP variant using the MegaWizard Plug-In Manager flow (not the SOPC Builder flow.) and works seamlessly with the Avalon Streaming (Avalon-ST) interface. Figure 1-4 illustrates the root port testbench.

Figure 1-4. Root Port Design Example

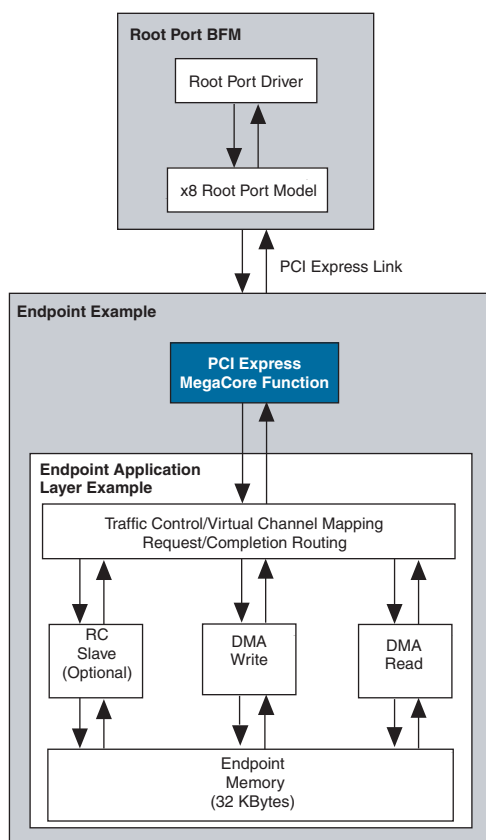


Endpoint Design Example

The endpoint design example illustrates the application interface to the PCI Express MegaCore function and is delivered as clear-text source-code (VHDL and Verilog HDL) suitable for both simulation and synthesis, as well as for OpenCore Plus evaluation of the MegaCore function in hardware. This design example is available when using the MegaWizard Plug-In Manager flow (not the SOPC Builder flow) and works seamlessly with the Avalon Streaming (Avalon-ST) interface or the descriptor/data interface.

Figure 1-5 illustrates the testbench for the chaining DMA example.

Figure 1-5. Testbench for the Chaining DMA Design Example



You can replace the endpoint application layer example shown in Figure 1-5 with your own application layer design and then modify the BFM driver to generate the transactions needed to test your application layer.

MegaCore Function Verification

To ensure PCI Express compliance, Altera performs extensive validation of the PCI Express MegaCore functions. Validation includes both simulation and hardware testing.

Simulation Environment

Altera's verification simulation environment for the PCI Express MegaCore functions uses multiple testbenches consisting of industry-standard BFM's driving the PCI Express link interface. A custom BFM connects to the application-side interface.

Altera performs the following tests in the simulation environment:

- Directed tests that test all types and sizes of transaction layer packets and all bits of the configuration space.
- Error injection tests that inject errors in the link, transaction layer packets, data link layer packets, and check for the proper response from the MegaCore functions.
- PCI-SIG Compliance Checklist tests that specifically test the items in the checklist.
- Random tests that test a wide range of traffic patterns across one or more virtual channels.

Compatibility Testing Environment

Altera has performed significant hardware testing of the PCI Express MegaCore functions to ensure a reliable solution. The MegaCore functions have been tested at various PCI-SIG PCI Express Compliance Workshops in 2005, 2006, and 2007 with Stratix II GX and Arria GX devices and various external PHYs, and they have passed all PCI-SIG gold tests and interoperability tests with a wide selection of motherboards and test equipment. In addition, Altera internally tests every release with motherboards and switch chips from a variety of manufacturers. All PCI-SIG compliance tests are also run with each MegaCore function release.

Performance and Resource Utilization

This section provides separate size and performance numbers for device families using the MegaWizard Plug-In Manager design flow and for device families using the SOPC Builder design flow.

Avalon-ST Interface

The Avalon-ST interface is now available in a hard IP implementation for Stratix IV GX and Arria II GX devices. This section tabulates the typical expected performance and resource utilization for both the hard IP and soft IP implementations for various parameters when using the MegaWizard Plug-In Manager. The OpenCore Plus evaluation feature is disabled and the following parameter settings are used:

- On the **Buffer Setup** page, for $\times 1$, $\times 4$, and $\times 8$ configurations:
 - **Maximum payload size** set to **256 Bytes** unless specified otherwise.
 - **Desired performance for received requests** and **Desired performance for completions** both set to **Medium** unless otherwise specified.
- On the **Capabilities** page, the number of **Tags supported** set to **16** for all configurations unless specified otherwise.

For a complete description of all parameters, refer to [Chapter 3, Parameter Settings](#).

Performance and resource utilization tables provide results for both the hard IP and soft IP implementations of the Avalon-ST interface. Tables appear for the following device families:

- Stratix IV
- Arria GX
- Cyclone III
- Stratix II
- Stratix II GX
- Stratix III

Hard IP Implementation

The hard IP implementation is available in Stratix IV GX and Arria II GX devices.

Stratix IV Family

Table 1-5 shows the resource utilization of the Stratix IV (EP4SGX290FH29C2X) hard IP implementation using either the Avalon-ST or Avalon-MM interface with a maximum payload of 256 bytes and 32 tags for the Avalon-ST interface and 16 tags for the Avalon-MM interface.

Table 1-5. Performance and Resource Utilization, Avalon-ST Interface and Avalon-MM–Hard IP Implementation, Stratix IV GX Devices

Parameters			Size		
×1/ ×4	Internal Clock (MHz)	Virtual Channel	Combinational ALUTs	Dedicated Registers	Memory Blocks M9K
Avalon-ST Interface–MegaWizard Plug-In Manager Design Flow					
×1	125	1	100	100	0
×1	125	2	100	100	0
×4	125	1	200	200	0
×4	125	2	200	200	0
×8	250	1	200	200	0
×8	250	2	200	200	0
Avalon-MM Interface–SOPC Builder Design Flow (1)					
×1	125	1	4300	3500	17
×4	125	1	4200	3400	17
Avalon-MM Interface–SOPC Builder Design Flow - Completer Only					
×1	125	1	4200	3100	14
×4	125	1	3800	2800	14
Note to Table 1-5:					
(1) The transaction layer of the Avalon-MM implementation is implemented in programmable logic to improve latency.					

Soft IP Implementation

The following sections show the resource utilization for the soft IP implementation.

Arria GX Devices

Table 1–6 shows the typical expected performance and resource utilization of Arria GX (EP1AGX60DF780C6) devices for different parameters with a maximum payload of 256 bytes using the Quartus II software, version 9.0.

Table 1–6. Performance and Resource Utilization, Avalon-ST Interface–Arria GX Devices (Note 1)

Parameters			Size			
×1/ ×4	Internal Clock (MHz)	Virtual Channel	Combinational ALUTs	Logic Registers	Memory Blocks	
					M512	M4K
×1	125	1	5800	4100	2	11
×1	125	2	7600	5300	3	17
×4	125	1	7500	5100	6	17
×4	125	2	9400	6400	7	27

Note to Table 1–6:

(1) This configuration only supports Gen1.

Stratix II GX Devices

Table 1–7 shows the typical expected performance and resource utilization of Stratix II and Stratix II GX (EP2SGX130GF1508C3) devices for a maximum payload of 256 bytes for devices with different parameters, using the Quartus II software, version 9.0.

Table 1–7. Performance and Resource Utilization, Avalon-ST Interface - Stratix II and Stratix II GX Devices

Parameters			Size			
×1/ ×4 ×8	Internal Clock (MHz)	Virtual Channels	Combinational ALUTs	Logic Registers	Memory Blocks	
					M512	M4K
×1	125	1	5600	4000	2	13
×1	125	2	7300	5100	3	17
×4	125	1	7200	4900	6	15
×4	125	2	8900	6100	7	27
×8	250	1	6900	6000	10	17
×8	250	2	8100	7000	9	24

Stratix III Family

Table 1–8 shows the typical expected performance and resource utilization of Stratix III (EP3SL200F1152C2) devices for a maximum payload of 256 bytes with different parameters, using the Quartus II software, version 9.0.

Table 1–8. Performance and Resource Utilization, Avalon-ST Interface - Stratix III Family

Parameters			Size			
×1/ ×4	Internal Clock (MHz)	Virtual Channels	Combinational ALUTs	Logic Registers	M9K Memory Blocks	M144K Memory Blocks
×1	125	1	5500	4500	5	0
×1	125	2	7000	5900	9	0
×1 (1)	62.5	1	6000	4800	5	0
×1 (2)	62.5	2	7200	6000	11	1
×4	125	1	7200	5500	9	0
×4	125	2	8500	6600	15	0

Note to Table 1–8:

- (1) C4 device used.
- (2) C3 device used.

Stratix IV Family

Table 1–8 shows the typical expected performance and resource utilization of Stratix IV GX (EP3SGX290FH29C2X) devices for a maximum payload of 256 bytes with different parameters, using the Quartus II software, version 9.0.

Table 1–9. Performance and Resource Utilization, Avalon-ST Interface - Stratix IV Family

Parameters			Size		
×1/ ×4	Internal Clock (MHz)	Virtual Channels	Combinational ALUTs	Logic Registers	M9K Memory Blocks
×1	125	1	5600	4500	5
×1	125	2	7000	5700	10
×4	125	1	7200	5500	7
×4	125	2	8600	6600	14

Descriptor/Data Interface

This section tabulates the typical expected performance and resource utilization of the listed device families for various parameters when using the MegaWizard Plug-In Manager design flow using the descriptor/data interface, with the OpenCore Plus evaluation feature disabled and the following parameter settings:

- On the **Buffer Setup** page, for ×1, ×4, and ×8 configurations:
 - **Maximum payload size** set to **256 Bytes** unless specified otherwise.
 - **Desired performance for received requests** and **Desired performance for completions** both set to **Medium** unless specified otherwise.
- On the **Capabilities** page, the number of **Tags supported** set to **16** for all configurations unless specified otherwise.

Size and performance tables appear here for the following device families:

- Arria GX
- Cyclone III
- Stratix II
- Stratix II GX
- Stratix III
- Stratix IV

Arria GX Devices

Table 1–10 shows the typical expected performance and resource utilization of Arria GX (EP1AGX60DF780C6) devices for a maximum payload of 256 bytes with different parameters, using the Quartus II software, version 9.0.

Table 1–10. Performance and Resource Utilization, Descriptor/Data Interface - Arria GX Devices

Parameters			Size			
×1/ ×4	Internal Clock (MHz)	Virtual Channels	Combinational ALUTs	Logic Registers	Memory Blocks	
					M512	M4K
×1	125	1	5600	3700	1	9
×1	125	2	6900	4600	2	13
×4	125	1	7000	4500	5	13
×4	125	2	8200	5400	6	33

Cyclone III Family

Table 1–11 shows the typical expected performance and resource utilization of Cyclone III (EP3C80F780C6) devices for different parameters, using the Quartus II software, version 9.0.

Table 1–11. Performance and Resource Utilization, Descriptor/Data Interface - Cyclone III Family

Parameters			Size		
×1/ ×4	Internal Clock (MHz)	Virtual Channels	Logic Elements	Dedicated Registers	M9K Memory Blocks
×1	125	1	8800	3600	6
×1	125	2	10800	4400	9
×1 (1)	62.5	1	8700	3800	11
×1	62.5	2	10400	4600	14
×4	125	1	11300	4500	12
×4	125	2	13100	5500	17

Note to Table 1–11:

- (1) **Max payload** set to 128 bytes, the number of **Tags supported** set to 4, and **Desired performance for received requests** and **Desired performance for completions** both set to Low.

Stratix II GX Devices

Table 1-12 shows the typical expected performance and resource utilization of the Stratix II and Stratix II GX (EP2SGX130GF1508C3) devices for a maximum payload of 256 bytes with different parameters, using the Quartus II software, version 9.0.

Table 1-12. Performance and Resource Utilization, Descriptor/Data Interface - Stratix II and Stratix II GX Devices

Parameters			Size			
×1/ ×4	Internal Clock (MHz)	Virtual Channels	Combinational ALUTs	Logic Registers	Memory Blocks	
					M512	M4K
×1	125	1	5300	3500	1	9
×1	125	2	6600	4400	2	13
×4	125	1	6800	4500	5	13
×4	125	2	8000	5300	6	19
×8	250	1	6700	5600	8	42
×8	250	2	7500	6300	8	16

Stratix III Family

Table 1-13 shows the typical expected performance and resource utilization of Stratix III (EP3SL200F1152C2) devices for a maximum payload of 256 bytes with different parameters, using the Quartus II software, version 9.0.

Table 1-13. Performance and Resource Utilization, Descriptor/Data Interface - Stratix III Family

Parameters			Size		
×1/ ×4	Internal Clock (MHz)	Virtual Channels	Combinational ALUTs	Dedicated Registers	M9K Memory Blocks
×1	125	1	5300	3600	6
×1	125	2	6300	4400	8
×1 (1)	62.5	1	5600	3900	8
×1 (2)	62.5	2	6400	4700	11
×4	125	1	6900	4600	14
×4	125	2	7900	5400	18
Notes to Table 1-13: (1) C4 device used. (2) C3 device used.					

Stratix IV Family

Table 1–13 shows the typical expected performance and resource utilization of Stratix IV (EP4SGX290FH29C2X) devices for a maximum payload of 256 bytes with different parameters, using the Quartus II software, version 9.0.

Table 1–14. Performance and Resource Utilization, Descriptor/Data Interface - Stratix IV Family

Parameters			Size		
×1/ ×4	Internal Clock (MHz)	Virtual Channels	Combinational ALUTs	Dedicated Registers	M9K Memory Blocks
×1	125	1	5200	3600	5
×1	125	2	6300	4400	8
×4	125	1	6900	4600	9
×4	125	2	7900	5400	12

Avalon-MM Interface

The Avalon-MM interface is available in a hard IP implementation for Stratix IV GX and Arria II GX devices. Refer to Table 1–5 on page 1–11 for resource utilization of the hard IP implementation. This section tabulates the typical expected performance and resource utilization for the soft IP implementation of the listed device families for various parameters when using the SOPC Builder design flow to create a design with an Avalon-MM interface and the following parameter settings:

- On the **Buffer Setup** page, for ×1, ×4 configurations:
 - **Maximum payload size** set to **256 Bytes** unless specified otherwise
 - **Desired performance for received requests** and **Desired performance for completions** set to **Medium** unless specified otherwise
- 16 Tags

Size and performance tables appear here for the following device families:

- Arria GX
- Cyclone III
- Stratix II
- Stratix II GX
- Stratix III and Stratix IV

Arria GX Devices

Table 1-15 shows the typical expected performance and resource utilization of Arria GX (EP1AGX60CF780C6) devices for a maximum payload of 256 bytes with different parameters, using the Quartus II software, version 9.0.

Table 1-15. Performance and Resource Utilization, Avalon-MM Interface - Arria GX Devices
(Note 1)

Parameters			Size			
×1/ ×4	Internal Clock (MHz)	Virtual Channels	Combinational ALUTs	Logic Registers	Memory Blocks	
					M512	M4K
×1	125	1	7300	5000	3	31
×4	125	1	8900	5900	7	35

Note to Table 1-13:
(1) These numbers are preliminary.

It may be difficult to achieve 125 MHz frequency in complex designs that target the Arria GX device. Altera recommends the following strategies to achieve timing:

- Use separate clock domains for the Avalon-MM and PCI Express modules
- Set the Quartus II Analysis & Synthesis Settings **Optimization Technique** to **Speed**
- Add non-bursting pipeline bridges to the Avalon-MM master ports
- Use Quartus II seed sweeping methodology

Cyclone III Family

Table 1-16 shows the typical expected performance and resource utilization of Cyclone III (EP3C80F780C6) devices for a maximum payload of 256 bytes with different parameters, using the Quartus II software, version 9.0.

Table 1-16. Performance and Resource Utilization, Avalon-MM Interface - Cyclone III Family

Parameters		Size		
×1/ ×4	Internal Clock (MHz)	Logic Elements	Dedicated Registers	M9K Memory Blocks
×1	125	11300	4500	27
×1 (1)	62.5	11300	4800	34
×4	125	13800	5400	31

Note to Table 1-16:

(1) Maximum payload of 128 bytes. C8 device used.

Stratix II GX Devices

Table 1-17 shows the typical expected performance and resource utilization of Stratix II and Stratix II GX (EP2SGX130GF1508C3) devices for a maximum payload of 256 bytes with different parameters, using the Quartus II software, version 9.0.

Table 1-17. Performance and Resource Utilization, Avalon-MM Interface - Stratix II GX Devices

Parameters		Size			
×1/ ×4	Internal Clock (MHz)	Combinational ALUTs	Dedicated Registers	Memory Blocks	
				M512	M4K
×1	125	6900	5000	3	34
×4	125	8760	5900	7	38

Stratix III Family

Table 1-18 shows the typical expected performance and resource utilization of Stratix III (EP3SL200F1152C2) devices for a maximum payload of 256 bytes with different parameters, using the Quartus II software, version 9.0.

Table 1-18. Performance and Resource Utilization, Avalon-MM Interface - Stratix III Family

Parameters		Size		
×1/ ×4	Internal Clock (MHz)	Combinational ALUTs	Dedicated Registers	M9K Memory Blocks
×1	125	7200	5400	17
×1 (1)	62.5	7500	5500	22
×4	125	8500	6100	19

Note to Table 1-8:

(1) C4 device used.

Stratix IV Family

Table 1-18 shows the typical expected performance and resource utilization of Stratix IV (EP4SGX290FH29C2X) devices for a maximum payload of 256 bytes with different parameters, using the Quartus II software, version 9.0.

Table 1-19. Performance and Resource Utilization, Avalon-MM Interface - Stratix IV Family

Parameters		Size		
×1/ ×4	Internal Clock (MHz)	Combinational ALUTs	Dedicated Registers	M9K Memory Blocks
×1	125	7000	5100	19
×4	125	8400	6000	19

Recommended Speed Grades

Table 1-20 shows the recommended speed grades for each device family for the supported link widths and internal clock frequencies. When the internal clock frequency is 125 MHz or 250 MHz, Altera recommends setting the Quartus II Analysis & Synthesis Settings **Optimization Technique** to **Speed**.


 Refer to “Setting Up and Running Analysis and Synthesis” in Quartus II Help and *Area and Timing Optimization* in volume 2 of the *Quartus II Handbook* for more information about how to effect this setting.

Table 1–20. Recommended Device Family Speed Grades (Part 1 of 2)

Device Family	Link Width	Internal Clock Frequency (MHz)	Recommended Speed Grades
Avalon-ST or Descriptor/Data Interface Soft IP Implementation			
Arria GX	×1, ×4	125	–6
Cyclone II, Cyclone III	×1, ×4	125	–6 (1)
	×1	62.5	–6, –7, –8 (1)
Stratix GX	×1, ×4	125	–5 (1), (2)
	×1	62.5	–5, –6 (1), (2)
Stratix II	×1, ×4	125	–3, –4, –5
	×1	62.5	–3, –4, –5 (1)
Stratix II GX	×1, ×4	125	–3, –4, –5 (1)
	×8	250	–3 (1) (3)
Stratix III	×1, ×4	125	–2, –3, –4
	×1	62.5	–2, –3, –4
Stratix IV E Gen1	×1	62.5	all speed grades
	×1, ×4	125	all speed grades
Stratix IV GX Gen1	×1	62.5	all speed grades
	×4	125	all speed grades
Avalon-ST Hard IP Implementation			
Arria II GX Gen1 with ECC Support (4)	×1	125	–4, –5, –6
	×4	125	–4, –5, –6
	×8	125	–4, –5, –6
Stratix IV GX Gen1 with ECC Support (4)	×1	125	–2, –3, –4
	×4	125	–2, –3, –4
	×8	250	–2, –3, –4 (5)
Stratix IV GX Gen2 with ECC Support (4)	×1	125	–2, –3 (5)
	×4	250	–2, –3 (5)
Stratix IV GX Gen2 without ECC Support	×8	500	–2
Avalon-MM Interface			
Arria GX	×1, ×4	125	–6
Arria II GX	×1, ×4	125	–4, –5, –6
Cyclone II, Cyclone III	×1, ×4	125	–6
	×1	62.5	–6, –7, –8 (6)
Stratix GX	×1, ×4	125	–5 (1), (2)
	×1	62.5	–5, –6, (2)

Table 1-20. Recommended Device Family Speed Grades (Part 2 of 2)

Device Family	Link Width	Internal Clock Frequency (MHz)	Recommended Speed Grades
Stratix II	×1, ×4	125	-3, -4, -5 (1)
	×1	62.5	-3, -4, -5
Stratix II GX	×1, ×4	125	-3, -4, -5 (1)
Stratix III	×1, ×4	125	-2, -3, -4
	×1	62.5	-2, -3, -4
Stratix IV GX	×1, ×4	125	-2, -3, -4

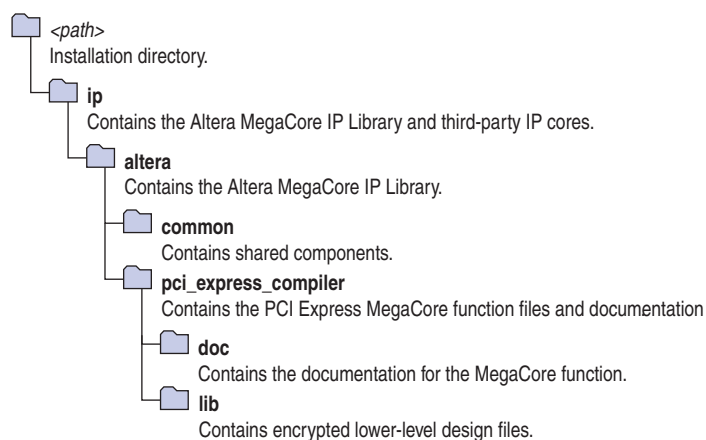
Notes to Table 1-20:

- (1) You must turn on the following Physical Synthesis settings in the Quartus II Fitter Settings to achieve timing closure for these speed grades and variations: **Perform physical synthesis for combinational logic**, **Perform register duplication**, and **Perform register retiming**. In addition, can use the Quartus II Design Space Explorer or Quartus II seed sweeping methodology. Refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 1 of the *Quartus II Development Software Handbook* for more information about how to set these options.
- (2) Altera does not recommend using Stratix GX for new PCI Express designs. For more information, refer to “Stratix GX PCI Express Compatibility” on page 4-59.
- (3) Altera recommends disabling the OpenCore Plus feature for the ×8 configuration.
- (4) The Rx Buffer and Retry Buffer ECC options are only available in the hard IP implementation.
- (5) Final results pending characterization by Altera for -2, -3, and -4. Refer the **.fit.rpt** file generated by the Quartus II software.
- (6) Altera recommends the External PHY 16-bit SDR or 8-bit SDR modes in the -8 speed grade.

Installation and Licensing

The PCI Express Compiler is installed with the Altera Complete Design Suite. You do not need a separate license to use the PCI Express hard IP implementation.

Figure 1-6 shows the directory structure after you install the PCI Express Compiler, where *<path>* is the installation directory. The default installation directory in Windows is **c:\altera\90**; in Linux it is **/opt/altera90**.

Figure 1-6. Directory Structure

You can use Altera's free OpenCore Plus evaluation feature to evaluate the MegaCore function in simulation and in hardware before you purchase a license. You need to purchase a license for the MegaCore function only after you are satisfied with its functionality and performance, and you are ready to take your design to production.

After you purchase a license for the PCI Express MegaCore function, you can request a license file from the Altera website at www.altera.com/licensing and install it on your computer. When you request a license file, Altera emails you a **license.dat** file. If you do not have internet access, contact your local Altera representative.



For details on installation and licensing, refer to the *Quartus II Installation & Licensing for Windows and Linux Workstations*.

OpenCore Plus Evaluation (Not Required for Hard IP)

With Altera's free OpenCore Plus evaluation feature, you can perform the following actions:

- Simulate the behavior of a megafunction (Altera MegaCore function or AMPPSM megafunction) in your system
- Verify the functionality of your design, as well as evaluate its size and speed quickly and easily
- Generate time-limited device programming files for designs that include megafunctions
- Program a device and verify your design in hardware

OpenCore Plus hardware evaluation is not applicable to the hard IP implementation of the PCI Express Compiler. You may use the hard IP implementation of this MegaCore function without a separate license.

OpenCore Plus Time-Out Behavior (Not Required for Hard IP)

OpenCore Plus hardware evaluation supports the following two operation modes:

- *Untethered*—the design runs for a limited time.
- *Tethered*—requires a connection between your board and the host computer. If tethered mode is supported by all megafunctions in a design, the device can operate for a longer time or indefinitely.

All megafunctions in a device time out simultaneously when the most restrictive evaluation time is reached. If your design includes more than one megafunction, a specific megafunction's time-out behavior may be masked by the time-out behavior of the other megafunctions.

For MegaCore functions, the untethered timeout is one hour; the tethered timeout value is indefinite. Your design stops working after the hardware evaluation time expires. During time-out the Link Training and Status State Machine (LTSSM) is held in the reset state.



For more information about OpenCore Plus hardware evaluation, refer to *AN 320: OpenCore Plus Evaluation of Megafunctions*.

Design Flow

You can parameterize and instantiate the PCI Express MegaCore function using either the MegaWizard Plug-In Manager or SOPC Builder design flow. The hard IP implementation is available for Stratix IV GX and Arria II GX devices in both the MegaWizard Plug-In Manager flow with the Avalon-ST interface and the SOPC Builder design flow with the Avalon-MM interface. The MegaWizard Plug-In Manager flow is used for designs with the Avalon-ST interface and the SOPC Builder design flow is used for designs with the Avalon-MM interface. The hard IP implementation uses significantly fewer resources than the soft IP implementation.



You can alternatively use the IP Advisor to start your PCI Express design with the Avalon-ST interface. On the Quartus II Tools menu, point to **Advisors**, and then click **IP Advisor**. The IP Advisor guides you through a series of recommendations for selecting, parameterizing, evaluating and instantiating a PCI Express MegaCore function in your design. The Advisor then guides you through compilation of your project.

Table 2–1 outlines the differences between the features offered by the two design flows.

Table 2–1. PCI Express MegaCore Function Features (Part 1 of 2)

Feature	Hard IP Implementation		Soft IP Implementation	
	MegaWizard Plug-In Manager Flow	SOPC Builder Design Flow	MegaWizard Plug-In Manager Flow	SOPC Builder Flow
MegaCore License	Free	Free	Required	Required
Root port	Supported	Not supported	Not supported	Not supported
Gen1	×1, ×4, ×8	×1, ×4	Yes	Yes
Gen2	×1, ×4, ×8 (1)	×1 (1)	No	No
Avalon Memory-Mapped (Avalon-MM) Interface	Not supported	Supported	Not supported	Supported
64-Avalon Streaming (Avalon-ST) Interface	Supported	Not supported	Supported	Not supported
128-bit Avalon-ST Interface	Supported	Not supported	Not supported	Not supported
Descriptor/Data Interface (2)	Not supported	Not supported	Supported	Not supported
Legacy Endpoint	Supported	Not supported	Supported	Not supported
IP Advisor	Supported	Not supported	Supported	Not supported
Transaction layer packet type (3)	All	<ul style="list-style-type: none"> Memory read request Memory write request Completion with or without data 	All	<ul style="list-style-type: none"> Memory read request Memory write request Completion with or without data

Table 2–1. PCI Express MegaCore Function Features (Part 2 of 2)

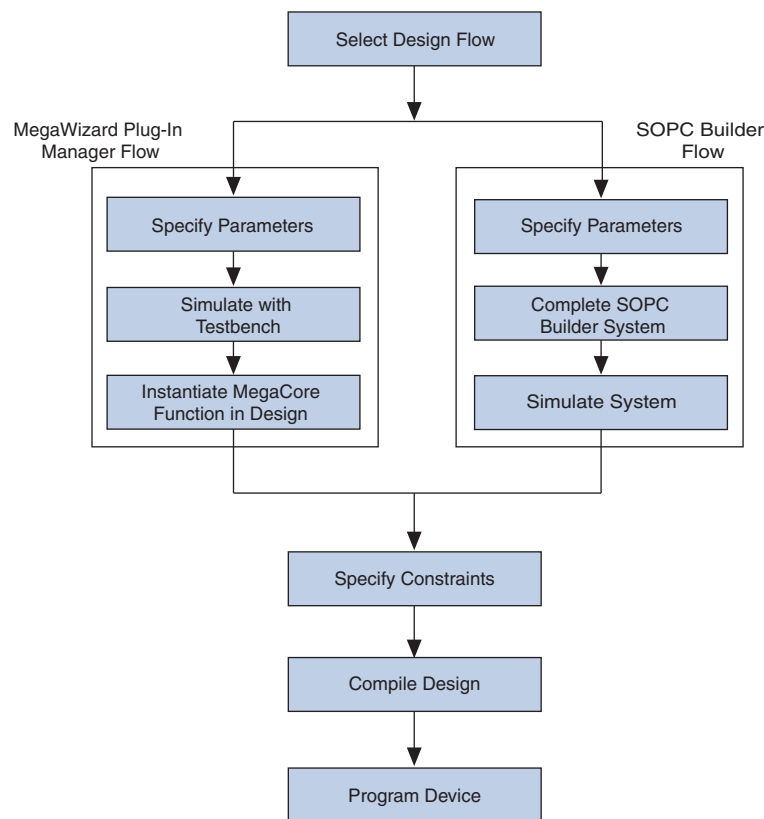
Feature	Hard IP Implementation		Soft IP Implementation	
	MegaWizard Plug-In Manager Flow	SOPC Builder Design Flow	MegaWizard Plug-In Manager Flow	SOPC Builder Flow
Maximum payload size	128 bytes–2 KBytes (Stratix IV GX) 128 bytes–512 bytes (Arria II GX)	128–256 bytes	128 bytes–2 KBytes	128–256 bytes
Number of virtual channels	2 (Stratix IV GX) 1 (Arria II GX)	1	1–4	1
Reordering of out-of-order completions (transparent to the application layer)	Not supported	Supported	Not supported	Supported
Requests that cross 4 KByte address boundary (transparent to the application layer)	Not supported	Supported	Not supported	Supported
Number of tags supported for non-posted requests	32 or 64	16	4–256	16
ECRC forwarding on Rx and Tx	Supported	Not Supported	Not supported	Not supported
MSI-X	Supported	Not Supported	Not supported	Not supported

Notes to Table 2–1:

- (1) Stratix IV GX devices only.
- (2) Not recommended for new designs.
- (3) Refer to [Appendix A, Transaction Layer Packet Header Formats](#) for the layout of TLP headers.

Figure 2–1 shows the stages for creating a system with the PCI Express MegaCore function and the Quartus II software. Each of the stages is described in subsequent sections.

Figure 2-1. PCI Express Compiler Design Flow



MegaWizard Plug-In Manager Design Flow

This section provides a high-level overview showing you how to instantiate the PCI Express Compiler using the MegaWizard Plug-In Manager design flow. This section teaches you how to simulate and compile the design example provided with this MegaCore function.

Specify Parameters

Follow the steps below to specify PCI Express Compiler parameters using the MegaWizard Plug-In Manager flow:

1. Create a Quartus II project using the **New Project Wizard** available from the File menu. When prompted, select a device family that supports your desired PCI Express implementation.
2. Launch MegaWizard Plug-In Manager from the Tools menu and follow the prompts in the MegaWizard Plug-In Manager interface to create a new custom megafunction variation.



You can find **PCI Express Compiler** by expanding **Installed Plug-Ins > Interfaces > PCI Express**.

3. If the **PHY** type is **Stratix IV GX** or **Arria II GX**, select the **PCI Express soft IP** or **PCI Express hard IP** option. For other devices only the **PCI Express soft IP** option is available.
4. Specify parameters on all pages in the **Parameter Settings** tab.

For detailed explanations of the parameters, refer to [Chapter 3, Parameter Settings](#).

On the **PCI Registers** tab, specify the BAR settings shown in [Table 2-2](#) to enable all of the tests in the provided testbench.

Table 2-2. PCI Registers Tab

Port type	Native Endpoint	
BAR	BAR TYPE	BAR Size
1:0	64-bit Prefetchable Memory	256 MBytes - 28 bits
2 or 3 (1)	32-bit Non-Prefetchable Memory	256 MBytes - 21 bits

Note to Table 2-2:

- (1) The DMA controller used for the endpoint chaining DMA design example requires the use of BAR2 or BAR3.

Many other BAR settings allow full testing of the DMA design example. Refer to “[Test Driver Module](#)” on [page 7-18](#) for a description of the BAR settings required for complete testing by the Chaining DMA test driver modules. Refer to [Table 3-4](#) on [page 3-8](#) for a detailed description of the available parameters.

1. On the **EDA** tab, turn on **Generate simulation model** to generate an IP functional simulation model for the MegaCore function. An IP functional simulation model is a cycle-accurate VHDL or Verilog HDL model produced by the Quartus II software.



Use the simulation models only for simulation and not for synthesis or any other purposes. Using these models for synthesis creates a non-functional design.

2. On the **Summary** tab, select the files you want to generate. A gray checkmark indicates a file that is automatically generated. All other files are optional.
3. Click **Finish** to generate the MegaCore function and supporting files.



A report file, *<variation name>.html*, in your project directory lists each file generated and provides a description of its contents.

4. Click **Yes** when you are prompted to add the Quartus II IP File to the project.

The Quartus II IP File (**.qip**) is a file generated by the MegaWizard interface or SOPC Builder that contains all of the necessary assignments and information required to process the core or system in the Quartus II compiler. Generally, a single **.qip** file is generated for each MegaCore function and for each SOPC Builder system. However, some more complex SOPC Builder components generate a separate **.qip** file, so the system **.qip** file references the component **.qip** file.

Simulate the Design

You can simulate the MegaCore function using the IP functional simulation model and the chaining DMA design example that accompany the PCI Express Compiler. [Chapter 7, Testbench and Design Example](#) provides a complete description of the design example. [Chapter 8, SOPC Builder Design Example](#) provides step-by-step instructions to simulate an SOPC Builder design example.



For more information about IP functional simulation models, refer to the [Simulating Altera IP in Third-Party Simulation Tools](#) chapter in volume 3 of the *Quartus II Handbook*.

Instantiate the MegaCore Function

You can now integrate your PCI Express MegaCore function variation in your design, and simulate the system with your custom testbench.

Constrain, Compile, and Program

To constrain, compile, and program your design, perform the following steps:

1. Open your Quartus II project in the Quartus II software.
2. Ensure your preferred timing analyzer is selected. (**Assignments Menu > Timing Analysis Settings.**) Altera recommends that you use the TimeQuest timing analyzer.
3. Source the constraint file that was generated with the variation by typing the following command at the Tcl console command prompt:

```
source <variation_name>.tcl ↵
```

This command adds the necessary logic constraints to your Quartus II project. A `<variation_name>.sdc` file is also generated with the variation to provide timing constraints for the TimeQuest timing analyzer.



The provided `<variation_name>.tcl` and `<variation_name>.sdc` files assume that the PCI Express device I/O pins (clocks and serial or PIPE interface signals) are named the same in your top-level design as they are on the PCI Express MegaCore variation. If you have renamed the ports, you must edit these files and make the same name changes.

4. On the Processing menu, click **Start Compilation** to compile your design.



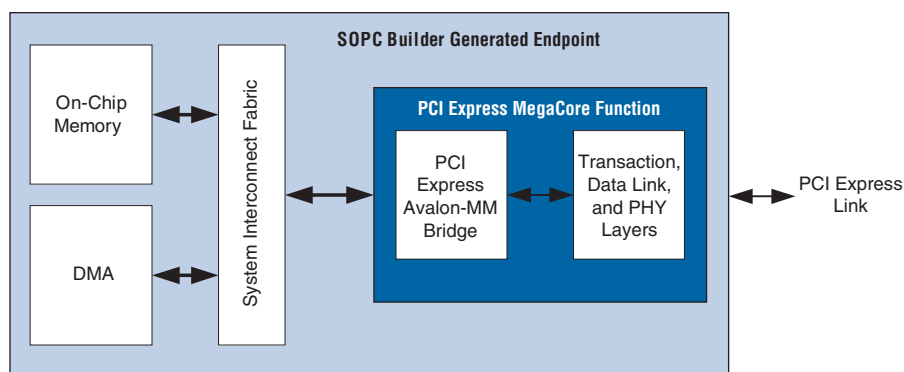
If your design does not initially meet the timing constraints, you can use the Design Space Explorer (Tools menu) in the Quartus II software to find the optimal Fitter settings for your design.

5. After a successful compilation, you can program the target Altera device and verify the design in hardware.

SOPC Builder Design Flow

The SOPC Builder design flow allows you to add the PCI Express MegaCore function directly to a new or existing SOPC Builder system. You can also easily add other available components to quickly create an SOPC Builder system. Figure 2-2 shows an SOPC Builder system that includes an on-chip memory and DMA controller. SOPC Builder automatically generates system interconnect logic and also creates the system simulation environment.

Figure 2-2. SOPC Builder Generated Endpoint



For more information about SOPC Builder, refer to *Volume 4: SOPC Builder* of the *Quartus II Handbook*. For more information about the Quartus II software, refer to Quartus II Help.

Specify Parameters

Follow the steps below to specify the PCI Express MegaCore function parameters using the SOPC Builder flow.

1. Create a new Quartus II project using the **New Project Wizard** available from the File menu.
2. Click **SOPC Builder** from the Tools menu.
3. For a new system, specify the system name and language.
4. Add the **PCI Express Compiler** to your system from the **System Contents** tab. You can find **PCI Express** by expanding **Interface Protocols > PCI > PCI Express**.
5. Specify the required parameters on all pages in the **Parameter Settings** tab. For detailed explanations of the parameters, refer to [Chapter 3, Parameter Settings](#).
6. Click **Finish** to complete the PCI Express MegaCore function and add it to the system.

Complete the SOPC Builder System

To complete the SOPC Builder system, perform the following steps.

1. Add and parameterize any additional components of the system.

For a complete example of an SOPC Builder system that contains the PCI Express MegaCore function, refer to [Chapter 8, SOPC Builder Design Example](#).

2. Connect the components using the SOPC Builder connection panel.



For Arria GX, Arria II GX, Stratix II GX, and Stratix IV GX designs, ensure that you connect the calibration clock (`cal_blk_clk`) to a clock signal with the appropriate frequency range of 10-125 MHz. Additionally the `cal_blk_clk` ports on other components that use transceivers must be connected to the same clock signal.

By default, clock names are not displayed. To display clock names in the **System Contents** tab, click **Filter** to display the **Filters** dialog box, from the Filter list, select **All**.

3. If you intend to simulate your SOPC builder system, turn on the **Simulation** option on the **System Generation** tab to generate a functional simulation model for the system.
4. Click **Generate** to generate the system.



For Stratix IV GX and Arria II GX designs, before compiling your design using the Quartus II software you must instantiate the `ALT_RECONFIG` megafunction and connect it to three pins from your SOPC Builder system that are required to dynamically reconfigure the transceiver. Dynamic reconfiguration is required to compensate for variations due to process, voltage, and temperature. Because these pins are of type `export`, they do not appear in the SOPC Builder GUI; therefore, you must edit the SOPC Builder RTL outside of the SOPC Builder GUI. The three pins are named using the following format by SOPC Builder:

- `reconfig_clk_<PCle_Compiler_instance_name>`
- `reconfig_fromgxb_<PCle_Compiler_instance_name>`
- `reconfig_togxb_<PCle_Compiler_instance_name>`

During system generation, SOPC Builder optionally generates a simulation model and testbench for the entire system which you can use to easily simulate your system in any of Altera's supported simulation tools. SOPC Builder also generates a set of ModelSim Tcl scripts and macros that you can use to compile the testbench, IP Functional simulation models, and plain-text RTL design files that describe your system to the ModelSim simulation software.

This chapter describes the PCI Express Compiler parameters, which you can set using the MegaWizard interface **Parameter Settings** tab.

System Settings

The first page of the MegaWizard interface contains the parameters for the overall system settings. [Table 3–1](#) describes these settings.

Table 3–1. System Settings Parameters (Part 1 of 3)

Parameter	Value	Description
PCIe Core Type	PCI Express hard IP	The hard IP implementation uses embedded dedicated logic to implement the PCI Express protocol stack, including the physical layer, data link layer, and transaction layer.
	PCI Express soft IP	The soft IP implementation uses optimized PLD logic to implement the PCI Express protocol stack, including physical layer, data link layer, and transaction layer.
PCIe System Parameters		
PHY type	Custom	Allows all types of external PHY interfaces (except serial). The number of lanes can be $\times 1$ or $\times 4$. This option is only available for the soft IP implementation.
	Stratix GX	Serial interface where Stratix GX uses the Stratix GX device family's built-in altgxb transceiver. Selecting this PHY allows only serial PHY interface and the Number of Lanes can be Gen1 $\times 1$, $\times 4$, or $\times 8$. Altera does not recommend Stratix GX for new PCI Express designs. For more information refer to “Stratix GX PCI Express Compatibility” on page 4–59 .
	Stratix II GX	Serial interface where Stratix II GX uses the Stratix II GX device family's built-in transceiver. Selecting this PHY allows only a serial PHY interface with the lane configuration set to Gen1 $\times 1$, $\times 4$, or $\times 8$.
	Stratix IV GX	Serial interface where Stratix IV GX uses the Stratix IV GX device family's built-in transceiver to support PCI Express Gen1 $\times 1$, $\times 4$, and $\times 8$ and Gen2 $\times 1$ and $\times 4$.
	Arria GX	Serial interface where Arria GX uses the Arria GX device family's built-in transceiver. Selecting this PHY allows only a serial PHY interface with the lane configuration set to Gen1 $\times 1$ or $\times 4$.
	Arria II GX	Serial interface where Arria II GX uses the Arria IV GX device family's built-in transceiver to support PCI Express Gen1 $\times 1$, $\times 4$, and $\times 8$.
	TI XIO1100	TI XIO1100 uses an 8-bit DDR/SDR with a TxClk or a 16-bit SDR with a transmit clock PHY interface. Both of these options restrict the number of lanes to $\times 1$. This option is only available for the soft IP implementation.
	NXP PX1011A	Philips NPX1011A uses an 8-bit SDR with a TxClk and a PHY interface. This option restricts the number of lanes to $\times 1$. This option is only available for the soft IP implementation.

Table 3–1. System Settings Parameters (Part 2 of 3)

Parameter	Value	Description
PHY interface	8-bit SDR, 16-bit SDR, 16-bit SDR w/TxClk, 8-bit DDR, 8-bit DDR w/TxClk, 8-bit SDR w/TxClk, 8-bit DDR/SDR w/TxClk serial	Selects the specific type of external PHY interface based on the interface datapath width and clocking mode. Refer to Chapter 6, External PHYs for additional detail on specific PHY modes. The external PHY setting only applies to the soft IP implementation.
Configure transceiver block	Enable fast recovery mode, Enable rate match FIFO Starting channel number	Displays a dialog box that allows you to configure the transceiver block for Arria GX and Stratix II GX. When Enable fast recovery mode is turned on, this option includes circuitry for a faster exit from the Rx ASPM L0s state. When turned off, exit from Rx ASPM L0s typically requires invoking link recovery. This feature will be available for Arria II GX and Stratix IV GX devices after these devices are characterized. When the Enable rate match FIFO is turned on, the Rx transceiver uses a rate matching FIFO for synchronization. When turned off, the Rx transceiver bypasses the rate matching FIFO, requiring a common clock configuration. For Arria GX, Arria II GX, Stratix II GX, and Stratix IV GX configuration, you must ensure that the OPPM clock group settings are set. An example of the Oppm clock group settings is provided for you in the <i><variation name>.tcl</i> file that is created with your MegaCore function variation. The settings are <code>GXB_OPPM_CLOCK_GROUP</code> and <code>GXB_OPPM_CLOCK_GROUP_DRIVER</code> . So that each transceiver can be correctly addressed by the ALT2GXB_RECONFIG block in Stratix II GX or the ALTGX_RECONFIG block in Stratix IV GX or Arria II GX, a starting channel number must be specified for the transceiver instance. You can select the starting channel number for each PCI Express MegaCore function in the MegaWizard interface. For more information refer to the Stratix II GX ALT2GXB_RECONFIG Megafunction User Guide chapter in volume 2 of the <i>Stratix II GX Device Handbook</i> or the Transceiver Configuration Guide in volume 3 of the <i>Stratix IV Device Handbook</i> .
Lanes	×1, ×4, ×8	Specifies the maximum number of lanes supported. The ×8 configuration is only supported in the MegaWizard Plug-In Manager flow for Stratix II GX and the hard IP implementations in the Stratix IV GX and Arria II GX devices.
Xcvr ref_clk PHY pclk	100 MHz, 125 MHz, 156.25 MHz	Specifies the frequency of the <code>refclk</code> input clock signal when using the Stratix GX PHY . The Stratix GX PHY can use either a 125- or 156.25 MHz clock directly. If you select 100 MHz, the MegaCore function uses a Stratix GX PLL to create a 125-MHz clock from the 100 MHz input. The Arria GX , Arria II GX , Stratix II GX , and Stratix IV GX PHY settings require a 100 MHz clock. If you use a PIPE interface (and the PHY type is neither Stratix GX , Stratix II GX , Stratix IV GX , Arria GX , nor Arria II GX) the <code>refclk</code> is not required. For Custom and TI X101100 PHYs, the <code>PHY pclk</code> frequency is 125 MHz. For the NXP PX1011A PHY, the <code>pclk</code> value is 250 MHz.

Table 3–1. System Settings Parameters (Part 3 of 3)

Parameter	Value	Description
Application Interface	64-bit Avalon-ST, 128-bit Avalon-ST, Descriptor/Data, Avalon-MM	Specifies the interface between the PCI Express transaction layer and the application layer. When using the MegaWizard Plug-In Manager flow, this parameter can be set to Avalon-ST or Descriptor/Data . Altera recommends the Avalon-ST option for all new designs. Refer to “ Avalon-ST Interface ” on page 1–10 and “ Descriptor/Data Interface ” on page 1–13 for more information. When using the SOPC Builder design flow this parameter is read-only and set to Avalon-MM . 128-bit Avalon-ST is only available when using the hard IP implementation.
Port type	Native Endpoint Legacy Endpoint Root Port	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 SOPC Builder design flow only supports Native Endpoint and the Avalon-MM interface to the user application. The Root Port option is available in the hard IP implementation for Stratix IV GX and Arria II GX devices.
PCI Express version	1.0A, 1.1, 2.0	Selects the PCI Express specification with which the variation is compatible. The PCI Express hard IP implementation supports PCI Express versions 1.1 and 2.0. The PCI Express soft IP implementation supports PCI Express versions 1.0a and 1.1
Application clock	62.5 MHz 125 MHz 250 MHz	Specifies the frequency at which the application interface clock operates. This frequency can only be set to 62.5 MHz or 125 MHz for soft IP Gen1 ×1 external PHY variations. For all other variations this field displays the frequency of operation which is controlled by the number of lanes, application interface width and Max rate setting. Refer to Table 4–1 on page 4–5 for a list of the supported combinations.
Max rate	Gen 1 (2.5 Gbps) Gen 2 (5.0 Gbps)	Specifies the maximum data rate at which the link can operate. The Gen2 rate is only supported in the hard IP implementation in Stratix IV GX devices.
Test out width	0, 9, 64, 128 or 512 bits	Indicates the width of the <code>test_out</code> signal. The following widths are possible: Hard IP <code>test_out</code> width: None, 9 bits, or 64 bits Soft IP ×1 or ×4 <code>test_out</code> width: None, 9 bits, or 512 bits Soft IP ×8 <code>test_out</code> width: None, 9 bits, or 128 bits Refer to Table 5–44 on page 5–70 and Appendix B, Test Port Interface Signals for signal definitions.
PCIe reconfig	Enable/Disable	Enables reconfiguration of the hard IP PCI Express read-only configuration registers. This parameter is only available for the hard IP implementation.

PCI Registers

The ×1 and ×4 MegaCore functions support memory space BARs ranging in size from 128 bytes to the maximum allowed by a 32-bit or 64-bit BAR. The ×8 MegaCore functions support memory space BARs from 4 KBytes to the maximum allowed by a 32-bit or 64-bit BAR.

The ×1 and ×4 MegaCore functions in legacy endpoint mode support I/O space BARs sized from 16 Bytes to 4 KBytes. The ×8 MegaCore function only supports I/O space BARs of 4 KBytes.

The SOPC Builder flow supports the following functionality:

- ×1 and ×4 lane width
- Native endpoint, with no support for:
 - I/O space BAR
 - 32-bit prefetchable memory
- 16 Tags
- 1 Message Signaled Interrupts (MSI)
- 1 virtual channel
- Up to 256 bytes maximum payload

In the SOPC Builder design flow, you can choose to allow SOPC Builder to automatically compute the BAR sizes and Avalon-MM base addresses or to enter the values manually. The Avalon-MM address is the translated base address corresponding to a BAR hit of a received request from PCI Express link. Altera recommends using the **Auto** setting. However, if you decide to enter the address translation entries, then you must avoid a conflict in address assignment when adding other components, making interconnections, and assigning base addresses in SOPC Builder. This process may take a few iterations between SOPC builder address assignment and MegaWizard address assignment to resolve address conflicts.

Table 3–2. PCI Registers (Part 1 of 2)

Parameter	Value	Description
PCI Base Address Registers		
BAR Table (BAR0)	BAR type and size	BAR0 size and type mapping (I/O space ⁽¹⁾ , memory space, prefetchable). BAR0 and BAR1 can be combined to form a 64-bit BAR. ⁽²⁾
BAR Table (BAR1)	BAR type and size	BAR1 size and type mapping (I/O space ⁽¹⁾ , memory space, prefetchable).
BAR Table (BAR2) ⁽³⁾	BAR type and size	BAR2 size and type mapping (I/O space ⁽¹⁾ , memory space, prefetchable). BAR2 and BAR3 can be combined to form a 64-bit BAR. ⁽²⁾
BAR Table (BAR3) ⁽³⁾	BAR type and size	BAR3 size and type mapping (I/O space ⁽¹⁾ , memory space, prefetchable).
BAR Table (BAR4) ⁽³⁾	BAR type and size	BAR4 size and type mapping (I/O space ⁽¹⁾ , memory space, prefetchable). BAR4 and BAR5 can be combined to form a 64-bit BAR. ⁽²⁾
BAR Table (BAR5) ⁽³⁾	BAR type and size	BAR5 size and type mapping (I/O space ⁽¹⁾ , memory space, prefetchable).
BAR Table (EXP-ROM) ⁽⁴⁾	BAR type and size	expansion ROM BAR size and type mapping (I/O space, memory space, prefetchable).
PCIe Read-Only Registers		
Device ID	0x0004	Sets the read-only value of the device ID register.
Subsystem ID ⁽³⁾	0x0004	Sets the read-only value of the subsystem device ID register.
Revision ID	0x01	Sets the read-only value of the revision ID register.

Table 3–2. PCI Registers (Part 2 of 2)

Vendor ID	0x1172	Sets the read-only value of the vendor ID register. This parameter can not be set to 0xFFFF per the PCI Express Specification.
Subsystem vendor ID (3)	0x1172	Sets the read-only value of the subsystem vendor ID register. This parameter can not be set to 0xFFFF per the <i>PCI Express Base Specification 1.1 or 2.0</i> .
Class code	0xFF0000	Sets the read-only value of the class code register.
Base and Limit Registers		
Input/Output (5)	Disable 16-bit I/O addressing 32-bit I/O addressing	Specifies what address widths are supported for the IO base and IO limit registers.
Prefetchable memory (5)	Disable 32-bit I/O addressing 64-bit I/O addressing	Specifies what address widths are supported for the prefetchable memory base register and prefetchable memory limit register.

Notes to Table 3–2:

- (1) A prefetchable 64-bit BAR is supported. A non-prefetchable 64-bit BAR is not supported because in a typical system, the root port configuration register of type 1 sets the maximum non-prefetchable memory window to 32-bits.
- (2) The SOPC Builder flow does not support I/O space for BAR type mapping. I/O space is only supported for legacy endpoint port types.
- (3) Only available for EP designs which require the use of the Header type 0 PCI configuration register.
- (4) The SOPC Builder flow does not support the expansion ROM.
- (5) Only available for RP designs which require the use of the Header type 1 PCI configuration register.

Capabilities Parameters

The **Capabilities** page contains the parameters setting various capability properties of the MegaCore function. These parameters are described in [Table 3–3](#).



The **Capabilities** page that appears in SOPC Builder does not include the **Simulation Mode** and **Summary** tabs.

Table 3–3. Capabilities Parameters (Part 1 of 4)

Parameter	Value	Description
Device Capabilities		
Tags supported	4–256	<p>Indicates the number of tags supported for non-posted requests transmitted by the application layer. The following options are available:</p> <ul style="list-style-type: none"> Hard IP: 32 or 64 tags for ×1, ×4, and ×8 Soft IP: 4–256 tags for ×1 and ×4; 4–32 for ×8 SOPC Builder: 16 tags for ×1 and ×4 <p>The transaction layer tracks all outstanding completions for non-posted requests made by the application. 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 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 as <code>cfg_devcsr[8]</code>.</p>

Table 3–3. Capabilities Parameters (Part 2 of 4)

Parameter	Value	Description
Implement completion timeout disable	On/Off	This option is only selectable for Gen2 root ports and endpoints. The timeout range is selectable. When On , the core supports the completion timeout disable mechanism. In all other modes its value is fixed. For Gen1.1, this option not available; both endpoints and root ports must implement a timeout mechanism. The timeout range is 50 μ s–50 ms. The recommended timeout value is 10 ms. For Gen1 root ports and endpoints, this option is always Off .
Completion timeout range	Ranges A–D	<p>This option is only available for PCI Express version 2.0. It 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, endpoints that issue requests on their own behalf, and PCI Express to PCI/PCI-X bridges that take ownership of requests issued on PCI Express link. For all other functions this field is reserved and must be hardwired to 0x0000b. Four time value ranges are defined:</p> <p>Range A: 50 μs to 10 ms Range B: 10 ms to 250 ms Range C: 250 ms to 4 s Range D: 4 s to 64 s</p> <p>Bits are set according to the table below to show timeout value ranges supported. 0x0000b completion timeout programming is not supported. The function must implement a timeout value in the range 50 s to 50 ms. The following encodings are used to specify the range:</p> <p>0x0001b Range A 0x0010b Range B 0x0011b Ranges A and B 0x0110b Ranges B and C 0x0111b Ranges A, B, and C 0x1110b Ranges B, C and D 0x1111b Ranges A, B, C, and D</p> <p>This setting is not available for PCIe version 1.0. All other values are reserved. Altera recommends that the completion timeout mechanism expire in no less than 10 ms.</p>
Error Reporting		
Implement advanced error reporting	On/Off	Implements the advanced error reporting (AER) capability.
Implement ECRC check	On/Off	Enables ECRC checking capability. Sets the read-only value of the ECRC check capable bit in the advanced error capabilities and control register. This parameter requires you to implement the advanced error reporting capability.
Implement ECRC generation	On/Off	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 implement the advanced error reporting capability.
Implement ECRC forwarding	On/Off	Available for hard IP implementation only. Forward ECRC to the application layer. On the Avalon-ST receive path, the incoming TLP contains the ECRC dword and the $\overline{\text{TD}}$ bit is set if an ECRC exists. On the Avalon-ST transmit path, the TLP from the application must contain the ECRC dword and have the $\overline{\text{TD}}$ bit set.
MSI Capabilities		

Table 3-3. Capabilities Parameters (Part 3 of 4)

Parameter	Value	Description
MSI messages requested	1, 2, 4, 8, 16, 32	Indicates the number of messages the application requests. Sets the value of the multiple message capable field of the message control register. The SOPC Builder design flow supports only 1 MSI.
MSI message 64-bit address capable	On/Off	Indicates whether the MSI capability message control register is 64-bit addressing capable. PCI Express native endpoints always support MSI 64-bit addressing.
Link Capabilities		
Link common clock	On/Off	Indicates if the common reference clock supplied by the system is used as the reference clock for the PHY. This parameter sets the read-only value of the slot clock configuration bit in the link status register.
Data link layer active reporting	On/Off	Turn this option on 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 <code>Hot-Plug Capable</code> field of the <code>Slot Capabilities</code> register), this option must be turned on. For upstream ports and components that do not support this optional capability, turn this option off .
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.
Link port number	0x01	Sets the read-only values of the port number field in the link capabilities register.
Slot Capabilities		
Enable slot capability	On/Off	The slot capability is required for root ports if a slot is implemented on the port. Slot status is recorded in the <code>PCI Express Capabilities</code> register. Only valid for root port variants.
Slot capability register	0x0000000	<p>Defines the characteristics of the slot. You turn this option on by selecting Enable slot capability. The various bits are defined as follows:</p> <p>31 19 18 17 16 15 14 7 6 5 4 3 2 1 0</p> <p>Physical Slot Number</p> <p>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</p>
MSI-X Capabilities		
Implement MSI-X	On/Off	The MSI-X functionality is only available in the hard IP implementation.
MSI-X Table size	10:0	System software reads this field to determine the MSI-X Table size $\langle N \rangle$, which is encoded as $\langle N-1 \rangle$. For example, a returned value of <code>10'b00000000011</code> indicates a table size of 4. This field is read-only.
MSI-X Table Offset	31:3	Points to the base of the MSI-X Table. The lower 3 bits of the Table BIR are set to zero by software to form a 32-bit qword-aligned offset. This field is read-only.

Table 3-3. Capabilities Parameters (Part 4 of 4)

Parameter	Value	Description
MSI-X Table BAR Indicator (BIR)	<5-1>:0	Indicates which one of a function's Base Address registers, located beginning at 0x10 in configuration space, is used to map the MSI-X table into memory space. This field is read-only. Depending on BAR settings, from 2 to BARs are available.
Pending Bit Array (PBA)		
Offset	31:3	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.
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.

Note to Table 3-3:

- (1) Throughout *The 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 or 2.0*. A word is 16 bits, a dword is 32 bits, and a qword is 64 bits.

Buffer Setup

The **Buffer Setup** page contains the parameters for the receive and retry buffers. [Table 3-4](#) describes the parameters you can set on this page.

Table 3-4. Buffer Setup Parameters (Part 1 of 3)

Parameter	Value	Description
Maximum payload size	128 bytes, 256 bytes, 512 bytes, 1 KByte, 2 KBytes	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 and optimizes the MegaCore function for this size payload. The SOPC Builder design flow supports only maximum payload sizes of 128 bytes and 256 bytes. The Arria II GX soft IP implementation only supports payload sizes of 128, 256, and 512 bytes.
Number of virtual channels	1-4	Specifies the number of virtual channels supported. This parameter sets the read-only extended virtual channel count field of port virtual channel capability register 1 and controls how many virtual channel transaction layer interfaces are implemented. The number of virtual channels supported depends upon the configuration, as follows: Hard IP: 1-2 channels for Stratix IV GX devices, 1 channel for Arria II GX devices Soft IP ×1 or ×4: 1-4 channels; Soft IP ×8: 2 channels SOPC Builder: 1 channel
Number of low-priority VCs	None, 2, 3, 4	Specifies the number of virtual channels in the low-priority arbitration group. The virtual channels numbered less than this value are low priority. Virtual channels numbered greater than or equal to this value are high priority. Refer to “Transmit Virtual Channel Arbitration” on page 4-11 for more information. This parameter sets the read-only low-priority extended virtual channel count field of the port virtual channel capability register 1.
Auto configure retry buffer size	On/Off	Controls automatic configuration of the retry buffer based on the maximum payload size. For the hard IP implementation, this is set to On .

Table 3-4. Buffer Setup Parameters (Part 2 of 3)

Parameter	Value	Description
Retry buffer size	256 Bytes–16 KBytes (powers of 2)	Sets the size of the retry buffer for storing transmitted PCI Express packets until acknowledged. This option is only available if you do not turn on Auto configure retry buffer size . The hard IP retry buffer is fixed at 4 KBytes for Arria II GX devices and 16 KBytes for Stratix IV GX devices.
Maximum retry packets	4–256 (powers of 2)	Set the maximum number of packets that can be stored in the retry buffer. For the hard IP implementation this parameter is set to 64 .
Desired performance for received requests	Maximum, High, Medium, Low	<p>Low—Provides the minimal amount of space for desired traffic. Select this option when the throughput of the received requests is not critical to the system design. This setting minimizes the device resource utilization.</p> <p>Because the Arria II GX and Stratix IV hard IP have a fixed Rx Buffer size, the choices for this parameter are limited to a subset of these values. For Max payload sizes of 512 bytes or less, the only available value is Maximum. For Max payload sizes of 1 KBytes or 2 KBytes a tradeoff has to be made between how much space is allocated to requests versus completions. At 1 KByte and 2 KByte Max payload sizes, selecting a lower value for this setting forces a higher setting for the Desired performance for received completions.</p> <p>For more information, refer to “Analyzing Throughput” on page 4-27</p>
Desired performance for received completions	Maximum, High, Medium, Low	<p>Specifies how to configure the Rx buffer size and the flow control credits:</p> <p>Maximum—Provides additional space to allow for additional external delays (link side and application side) and still allows full throughput.</p> <p>If you need more buffer space than this parameter supplies, select a larger payload size and this setting. The maximum setting increases the buffer size and slightly increases the number of logic elements (LEs), to support a larger payload size than is used. This is the default setting for the hard IP implementation.</p>

Table 3-4. Buffer Setup Parameters (Part 3 of 3)

Parameter	Value	Description
Desired performance for received completions (continued)	Maximum, High, Medium, Low	<p>Medium—Provides a moderate amount of space for received completions. Select this option when the received completion traffic does not need to use the full link bandwidth, but is expected to occasionally use short bursts of maximum sized payload packets.</p> <p>Low—Provides the minimal amount of space for received completions. Select this option when the throughput of the received completions is not critical to the system design. This is used when your application is never expected to initiate read requests on the PCI Express links. Selecting this option minimizes the device resource utilization.</p> <p>For the hard IP implementation, this parameter is not directly adjustable. The value set is derived from the values of Max payload size and the Desired performance for received requests parameter.</p> <p>For more information, refer to “Analyzing Throughput” on page 4-27.</p>
Rx Buffer Space Allocation (per VC)	Read-Only table	<p>Shows the credits and space allocated for each flow-controllable type, based on the Rx buffer size setting. All virtual channels use the same Rx buffer space allocation.</p> <p>The table does not show non-posted data credits because the MegaCore function always advertises infinite non-posted data credits and automatically has room for the maximum number of dwords of data that can be associated with each non-posted header.</p> <p>The numbers shown for completion headers and completion data indicate how much space is reserved in the Rx buffer for completions. However, infinite completion credits are advertised on the PCI Express link as is required for endpoints. It is up to the application layer to manage the rate of non-posted requests to ensure that the Rx buffer completion space does not overflow. The hard IP Rx buffer is fixed at 16 KBytes for Stratix IV GX devices and 4 KBytes for Arria II GX devices.</p>

Power Management

The **Power Management** page contains the parameters for setting various power management properties of the MegaCore function.



The **Power Management** page in the SOPC Builder flow does not include **Simulation Mode** and **Summary** tabs.

[Table 3-5](#) describes the parameters you can set on this page.

Table 3-5. Power Management Parameters (Part 1 of 2)

Parameter	Value	Description
L0s Active State Power Management (ASPM)		
Idle threshold for L0s entry	256 ns–8,192 ns (in 256 ns increments)	Indicates the idle threshold for L0s entry. This parameter specifies the amount of time the link must be idle before the transmitter transitions to L0s state. The PCI Express specification states that this time should be no more than 7 μ s, but the exact value is implementation-specific. If you select the Arria GX , Arria II GX , Stratix GX , Stratix II GX , or Stratix IV GX PHY, this parameter is disabled and set to its maximum value. If you are using an external PHY, consult the PHY vendor's documentation to determine the correct value for this parameter.

Table 3-5. Power Management Parameters (Part 2 of 2)

Parameter	Value	Description
Endpoint L0s acceptable latency	< 64 ns – > 4 μs	Indicates the acceptable endpoint L0s latency for the device capabilities register. Sets the read-only value of the endpoint L0s acceptable latency field of the device capabilities register. This value should be based on how much latency the application layer can tolerate. This setting is disabled for root ports.
Number of fast training sequences (N_FTS)		
Common clock	Gen1: 0–255 Gen2: 0–255	Indicates the number of fast training sequences needed in common clock mode. The number of fast training sequences required is transmitted to the other end of the link during link initialization and is also used to calculate the L0s exit latency field of the device capabilities register. If you select the Arria GX, Arria II GX, Stratix GX, Stratix II GX or Stratix IV GX PHY, this parameter is disabled and set to its maximum value. If you are using an external PHY, consult the PHY vendor's documentation to determine the correct value for this parameter.
Separate clock	Gen1: 0–255 Gen2: 0–255	Indicates the number of fast training sequences needed in separate clock mode. The number of fast training sequences required is transmitted to the other end of the link during link initialization and is also used to calculate the L0s exit latency field of the device capabilities register. If you select the Arria GX, Arria II GX, Stratix GX, Stratix II GX or Stratix IV GX PHY, this parameter is disabled and set to its maximum value. If you are using an external PHY, consult the PHY vendor's documentation to determine the correct value for this parameter.
Electrical idle exit (EIE) before FTS	3:0	Sets the number of EIE symbols sent before sending the N_FTS sequence. Legal values are 4–8. N_FTS is disabled for Arria II GX and Stratix IV GX devices pending device characterization.
L1s Active State Power Management (ASPM)		
Enable L1 ASPM	On/Off	Sets the L1 active state power management support bit in the link capabilities register. If you select the Arria GX, Arria II GX, Stratix GX, Stratix II GX , or Stratix IV GX PHY, this option is turned off and disabled.
Endpoint L1 acceptable latency	< 1μ – > 64 μs	Indicates the endpoint L1 acceptable latency. Sets the read-only value of the endpoint L1 acceptable latency field of the device capabilities register. This value should be based on the amount of latency the application layer can tolerate. This setting is disabled for root ports.
L1 Exit Latency Common clock	< 1μ – > 64 μs	Indicate the L1 exit latency for the separate clock. Used to calculate the value of the L1 exit latency field of the device capabilities register. If you select the Arria GX, Arria II GX, Stratix GX, Stratix II GX , or Stratix IV GX PHY, this parameter is disabled and set to its maximum value. If you are using an external PHY, consult the PHY vendor's documentation to determine the correct value for this parameter.
L1 Exit Latency Separate clock	< 1μ – > 64 μs	Indicate the L1 exit latency for the common clock. Used to calculate the value of the L1 exit latency field of the device capabilities register. If you select the Arria GX, Arria II GX, Stratix GX, Stratix II GX , or Stratix IV GX PHY, this parameter is disabled and set to its maximum value. If you are using an external PHY, consult the PHY vendor's documentation to determine the correct value for this parameter.

Avalon-MM Configuration

The **Avalon Configuration** page contains parameter settings for the PCI Express Avalon-MM bridge, available only in the SOPC Builder design flow soft IP implementation. [Table 3-6](#) describes the parameters on the **Avalon Configuration** page.

Table 3-6. Avalon Configuration Settings (Part 1 of 2)

Parameter	Value	Description
Avalon Clock Domain	Use PCIe core clock, Use separate clock	<p>Allows you to specify one or two clock domains for your application and the PCI Express MegaCore function. The single clock domain is higher performance because it avoids the clock crossing logic that separate clock domains require.</p> <p>Use PCIe core clock—In this mode, the PCI Express MegaCore function provides a clock output, <code>clk125_out</code>, to be used as the single clock for the PCI Express MegaCore function and the SOPC Builder system.</p> <p>Use separate clock—In this mode, the protocol layers of the PCI Express MegaCore function operate on an internally generated clock. The PCI Express MegaCore function exports <code>clk125_out</code>; however, this clock is not visible to SOPC Builder and cannot drive SOPC Builder components. The Avalon-MM bridge logic of the PCI Express MegaCore function operates on a different clock specified using SOPC Builder.</p>
PCIe Peripheral Mode	Requester/Completer, Completer-Only	<p>Specifies if the PCI Express component is capable of sending requests to the upstream PCI Express devices.</p> <p>Requester/Completer—Enables the PCI Express MegaCore function to send request packets on the PCI Express Tx link as well as receiving request packets on the PCI Express Rx link.</p> <p>Completer-Only—In this mode, the PCI Express MegaCore function can receive requests, but cannot send requests to PCI Express devices. 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. When selecting this option, you should also select Low for the Desired performance for received completions option on the Buffer Setup page to minimize the device resources consumed.</p>
Address translation table configuration	Dynamic translation table, Fixed translation table	<p>Sets Avalon-MM-to-PCI Express address translation scheme to dynamic or fixed.</p> <p>Dynamic translation table—Enables application software to write the address translation table contents using the control register access slave port. On-chip memory stores the table. Requires that the Avalon-MM CRA Port be enabled. Use several address translation table entries to avoid updating a table entry before outstanding requests complete.</p> <p>Fixed translation table—Configures the address translation table contents to hardwired fixed values at the time of system generation.</p>

Table 3-6. Avalon Configuration Settings (Part 2 of 2)

Parameter	Value	Description
Address translation table size		Sets Avalon-MM-to-PCI Express address translation windows and size.
Number of address pages	Up to 512-2ⁿ	Specifies the number of PCI Express base address pages of memory that the bridge can access. This value corresponds to the number of entries in the address translation table. The Avalon address range is segmented into one or more equal-sized pages that are individually mapped to PCI Express addressees. Select the number and size of the address pages. If you select a dynamic translation table , use several address translation table entries to avoid updating a table entry before outstanding requests complete.
Size of address pages	1 MByte-128 MBytes	Specifies the size of each PCI Express memory segment accessible by the bridge. This value is common for all address translation entries.
Fixed Address Translation Table Contents		Specifies the type and PCI Express base addresses of memory that the bridge can access. The upper bits of the Avalon-MM address are replaced with part of a specific entry. The MSBs of the Avalon-MM address, used to index the table, select the entry to use for each request. The values of the lower bits (as specified in the size of address pages parameter) entered in this table are ignored. Those lower bits are replaced by the lower bits of the incoming Avalon-MM addresses.
PCIe base address	32-bit 64-bit	
Type	32-bit Memory 64-bit Memory	
Avalon-MM CRA port	Enable Disable	Allows read/write access to bridge registers from Avalon using a specialized slave port. Disabling this option disallows read/write access to bridge registers.

This section provides a functional description of the following PCI Express Compiler features:

- Architecture
- Analyzing Throughput
- Configuration Space Register Content
- PCI Express Avalon-MM Bridge Control Register Content
- Active State Power Management (ASPM)
- Error Handling
- Stratix GX PCI Express Compatibility
- Clocking
- Multiple Core Configurations

Architecture

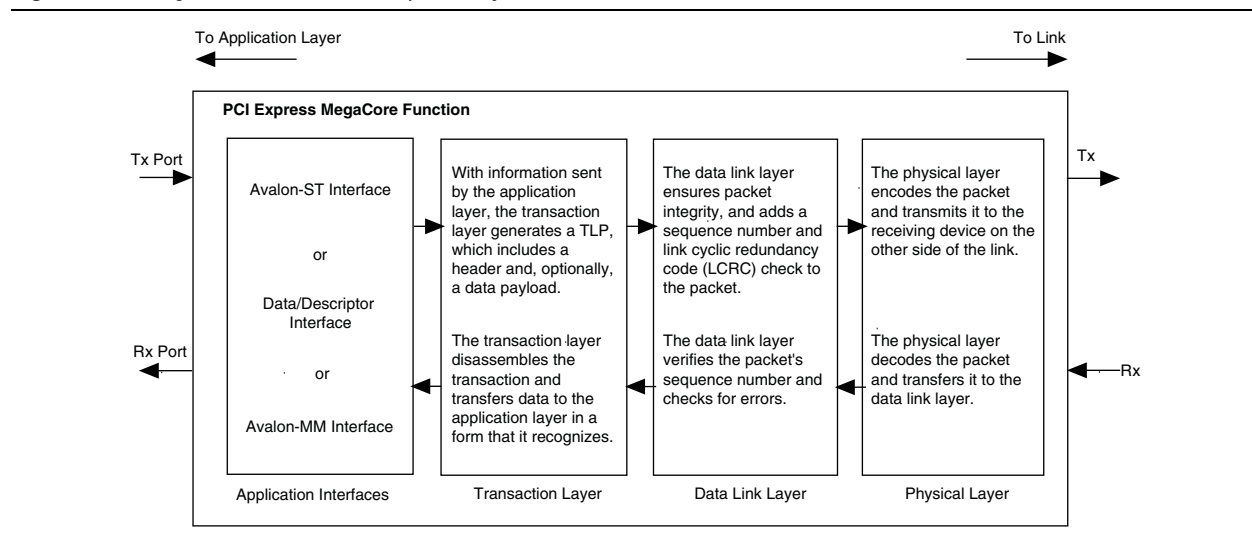
This section describes the following architectural features of the PCI Express Compiler:

- Transaction Layer
- Data Link Layer
- Physical Layer
- PCI Express Avalon-MM Bridge

For the hard IP implementation, you can design a root port using the Avalon-ST interface or end point using the Avalon-ST interface or the Avalon-MM interface. For the soft IP implementation, you can design an end point using the Avalon-ST, Avalon-MM, or the Descriptor/Data interface. All configurations, regardless of application interface, contain a transaction layer, a data link layer, and a PHY layer. [Figure 4–1](#) broadly describes the roles of each layer of the PCI Express MegaCore function.



The PCI Express endpoint resulting from the MegaWizard Plug-In Manager design flow can implement either a native PCI Express endpoint or a legacy endpoint. Altera recommends using native PCI Express endpoints for new applications; they support memory space read and write transactions only. Legacy endpoints provide compatibility with existing applications and can support I/O space read and write transactions. A PCI Express endpoint generated using the SOPC Builder design flow can only implement a native PCI Express endpoint.

Figure 4–1. MegaCore Function PCI Express Layers

PCI Express soft IP endpoints comply with the *PCI Express Base Specification 1.0a, or 1.1*. The PCI Express hard IP endpoint and root port comply with the *PCI Express Base Specification 1.1, or 2.0*. All of these implement all three layers of the specification:

- **Transaction Layer**—The transaction layer contains the configuration space, which manages communication with the application layer: the receive and transmit channels, the receive buffer, and flow control credits. You can choose one of the following two options for the application layer interface from the MegaWizard Plug-In Manager design flow:

- Avalon-ST Interface
- Descriptor/Data Interface

You can choose the Avalon-MM interface from the SOPC Builder flow.

- **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
 - 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 data link layer packets 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.

Application Interfaces

You can generate the PCI Express MegaCore function with one of three types of application interfaces: the Avalon-ST application interface, the descriptor/data application interface, or the Avalon-MM application interface.

Avalon-ST Application Interface

A PCI Express root port or endpoint which result from using the MegaWizard Plug-In Manager to specify the Avalon-ST interface includes a PCI Express Avalon-ST adapter module in addition to the three PCI Express layers: transaction layer, data link layer, and PHY layer (Figure 4-2).

The PCI Express Avalon-ST adapter maps PCI Express transaction layer packets (TLPs) to the user application Rx and Tx busses. Figure 4-2 illustrates this interface.

Figure 4-2. MegaCore Function with PCI Express Avalon-ST Interface Adapter

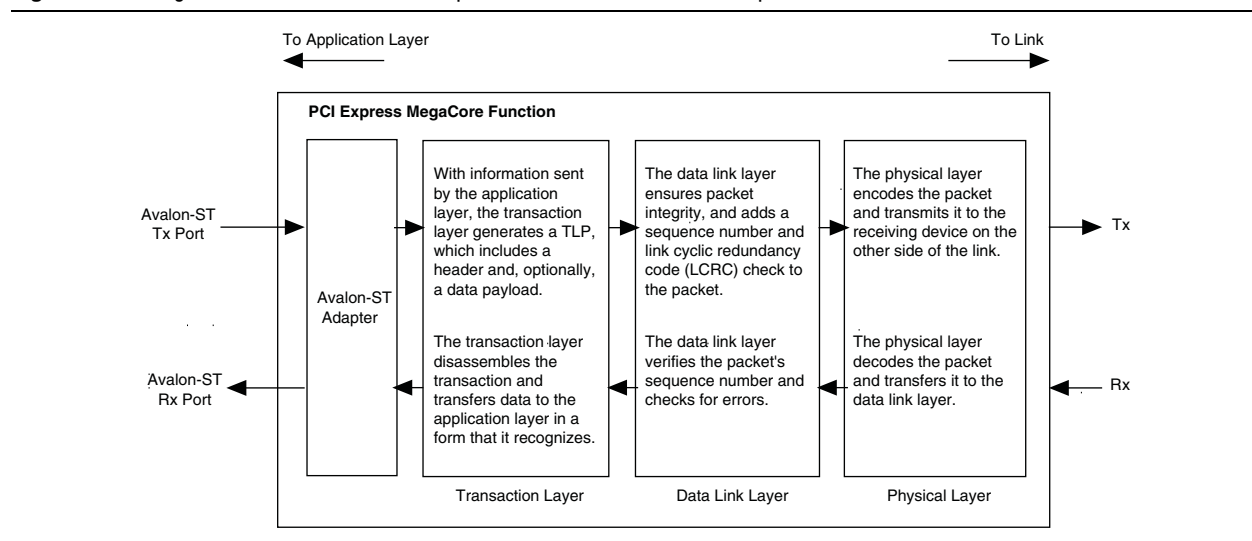


Figure 4-3 and Figure 4-4 illustrate the hard and soft IP implementations of the PCI Express MegaCore function. In both cases the adapter maps the user application Avalon-ST interface to PCI Express TLPs. The hard IP and soft IP implementations differ in the following respects:

- The hard IP implementation includes dedicated clock domain crossing logic between the PHYMAC and data link layers. In the soft IP implementation you can specify one or two clock domains for the MegaCore function.

- The hard IP implementation includes the following interfaces to access the configuration space registers:
 - The LMI interface
 - The Avalon-MM PCIe reconfig bus which can access any read-only configuration space register
 - In root port configuration, 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 RP configuration space registers, and a type 1 configuration TLP is used to access the configuration space registers of downstream nodes, typically endpoints on the other side of the link.

Figure 4–3. PCI Express Hard IP Implementation with Avalon-ST Interface to User Application

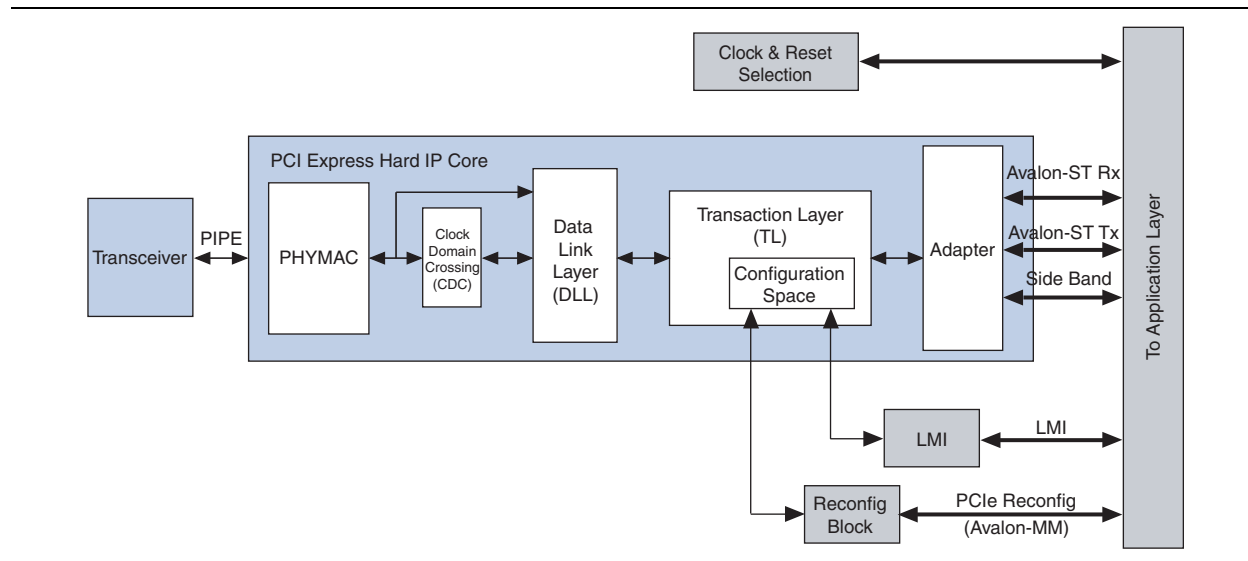


Figure 4–4. PCI Express Soft IP Implementation with Avalon-ST Interface to User Application

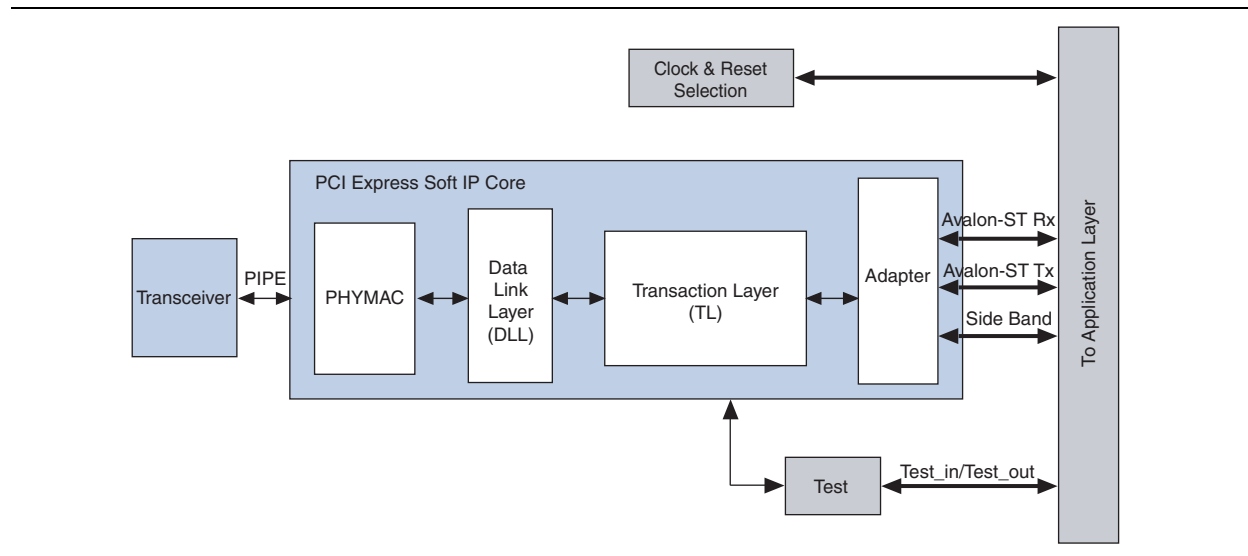


Table 4–1 provides the application clock frequencies for the hard IP and soft IP implementations. As this table indicates, the Avalon-ST interface can be either 64 or 128 bits for the hard IP implementation. For the soft IP implementation, the Avalon-ST interface is 64 bits.

Table 4–1. Application Clock Frequencies

Hard IP Implementation—Stratix IV GX		
Lanes	Gen1	Gen2
x1	125 MHz @ 64 bits	125 MHz @ 64 bits
x4	125 MHz @ 64 bits	250 MHz @ 64 bits or 125 MHz @ 128 bits
x8	250 MHz @ 64 bits or 125 MHz @ 128 bits	250 MHz @ 128 bits
Hard IP Implementation—Arria II GX		
Lanes	Gen1	Gen2
x1	125 MHz @ 64 bits	—
x4	125 MHz @ 64 bits	—
x8	125 MHz @ 128 bits	—
Soft IP Implementation		
Lanes	Gen1	Gen2
x1	62.5 MHz @ 64 bits or 125 MHz @ 64 bits	—
x4	125 MHz @ 64 bits	—
x8	250 MHz @ 64 bits	—

The following sections introduce the functionality of the interfaces shown in Figure 4–3 and Figure 4–4. For more detailed information, refer to “64- or 128-Bit Avalon-ST Rx Port” on page 5–6 and “64- or 128-Bit Avalon-ST Tx Port” on page 5–10.

Rx Datapath

The Rx datapath transports data from the transaction layer to the Avalon-ST interface. A FIFO buffers the Rx data from the transaction layer until the streaming interface accepts it. The adapter autonomously acknowledges all packets it receives from the PCI Express MegaCore function. The `rx_abort` and `rx_retry` signals of the transaction layer interface are not used. Masking of non-posted requests is partially supported. Refer to the description of the `rx_st_mask0` signal for further information about masking.

The Avalon-ST Rx datapath has a latency range of 3 to 6 `p1d_clk` cycles.

Tx Datapath

The Tx datapath transports data from the application's Avalon-ST interface to the transaction layer. A FIFO buffers the Avalon-ST data until the transaction layer accepts it.

If required, TLP ordering should be implemented by the application layer. The Tx datapath provides a Tx credit (`tx_cred`) vector which reflects the number of credits available. Note that for non-posted requests, this vector accounts for credits pending in the Avalon-ST adapter. For completions and posted requests, the `tx_cred` vector reflects the credits available in the transaction layer of the PCI Express MegaCore function. You must account for completion and posted credits which may be pending in the Avalon-ST adapter. You can use the read and write FIFO pointers and the FIFO empty flag to track packets as they are popped from the adaptor FIFO and transferred to the transaction layer.

The Avalon-ST Tx datapath has a latency range of 3 to 6 `pld_clk` cycles.

LMI Interface (Hard IP Only)

The LMI bus provides access to the PCI Express configuration space in the transaction layer. For more LMI details, refer to the [“LMI Signals—Hard IP Implementation” on page 5-29](#).

PCI Express Reconfiguration Block Interface (Hard IP Only)

The PCI Express reconfiguration bus allows you to dynamically change the read-only values stored in the configuration registers. For detailed information refer to the [“PCI Express Reconfiguration Block Signals—Hard IP Implementation” on page 5-31](#).

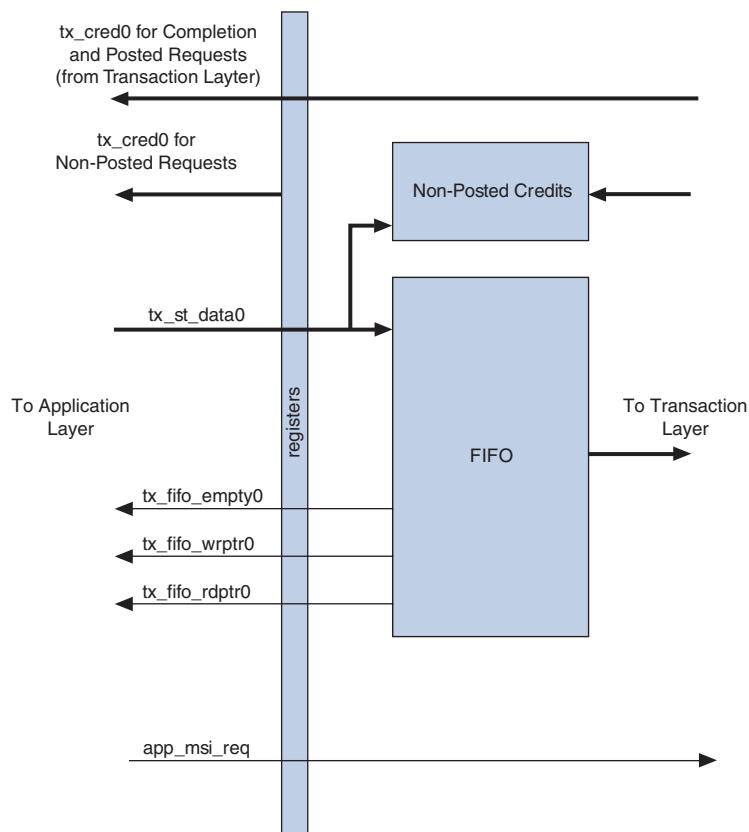
MSI (Message Signal Interrupt) Datapath

The MSI datapath contains the MSI boundary registers for incremental compilation. The interface uses the transaction layer's request-acknowledge handshaking protocol.

You use the Tx FIFO empty flag from the Tx datapath FIFO for Tx/MSI synchronization. When the Tx block application drives a packet to the Avalon-ST adapter, the packet remains in the Tx datapath FIFO as long as the Megacore function throttles this interface. When it is necessary to send an MSI request after a specific Tx packet, you can use the Tx FIFO empty flag to determine when the MegaCore function receives the TX packet.

For example, you may want to send an MSI request only after all Tx packets are issued to the transaction layer. Alternatively, if you cannot interrupt traffic flow to synchronize the MSI, you can use a counter to count 16 writes (the depth of the FIFO) after a Tx packet has been written to the FIFO (or until the FIFO goes empty) to ensure that the transaction layer interface receives the packet before issuing the MSI request. [Figure 4-5](#) illustrates the Avalon-ST Tx and MSI datapaths.

Figure 4–5. Avalon-ST Tx and MSI Datapaths



Incremental Compilation

The MegaCore function with Avalon-ST interface includes a fully registered interface between the user application and the PCI Express transaction layer. For the soft IP implementation, you can use incremental compilation to lock down the placement and routing of the PCI Express MegaCore function with the Avalon-ST interface to preserve placement and timing while changes are made to your application.

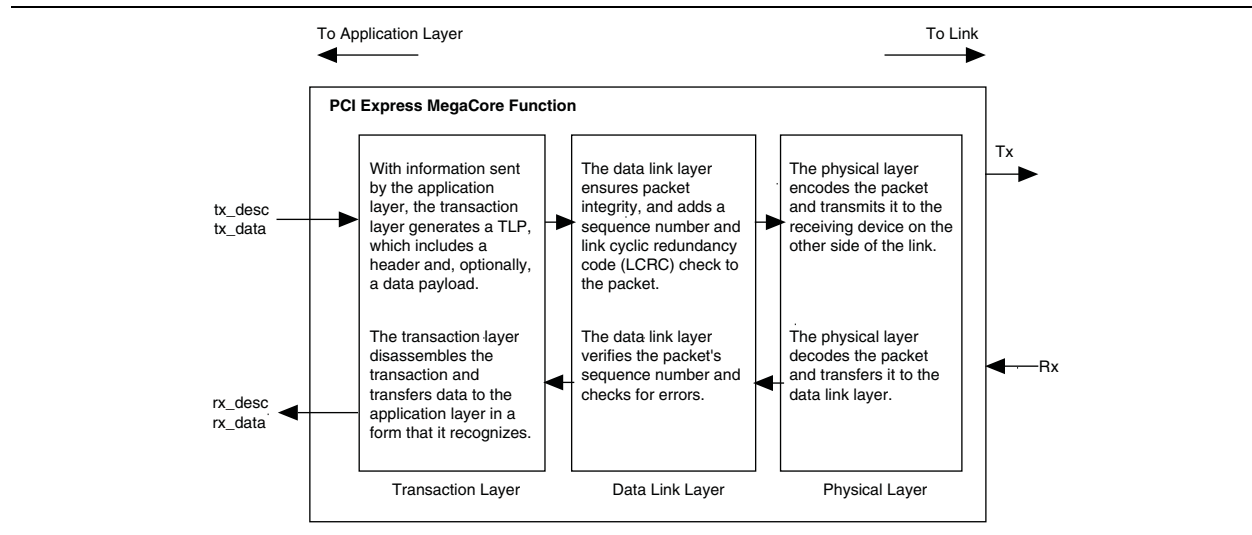


Incremental recompilation is not necessary for the PCI Express hard IP implementation. This implementation is fixed. All signals in the hard IP implementation are fully registered.

Descriptor/Data Application Interface (Soft IP Only)

When you use the MegaWizard Plug-In Manager to generate a PCI Express endpoint with the descriptor/data interface, the MegaWizard interface generates the transaction, data link, and PHY layers as described in the previous sections.

Figure 4–6 illustrates this interface.

Figure 4-6. PCI Express MegaCore Function with Descriptor/Data Interface

This configuration provides the lowest latency interface to the user, because the application connects directly to the transaction layer. The descriptor/data interface consists of an Rx port and Tx port for each virtual channel, as well as an interrupt/MSI interface, and configuration sideband signals.



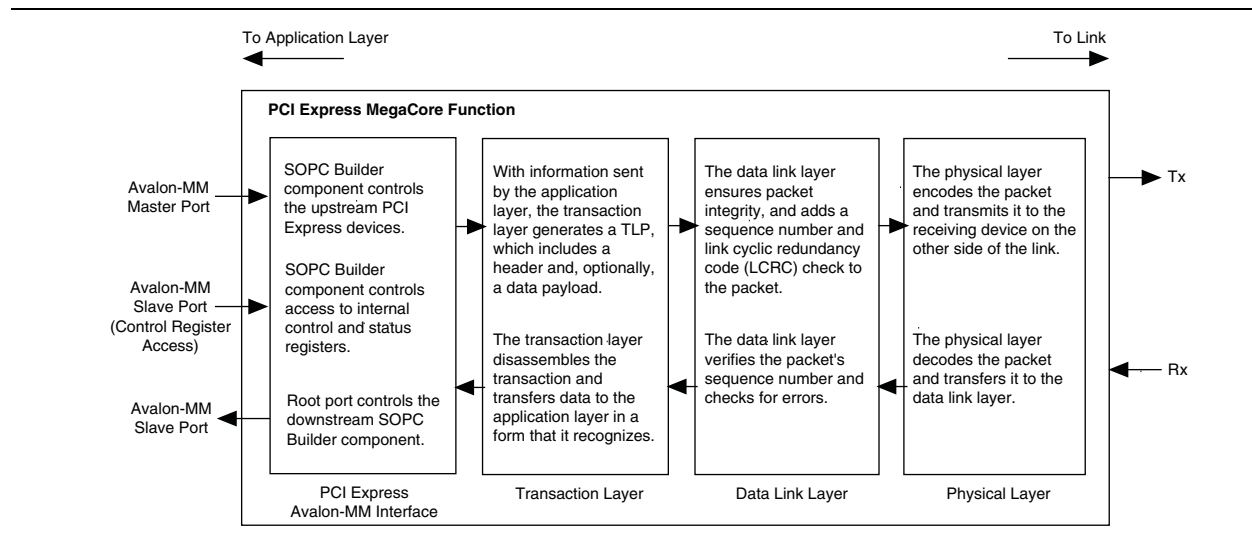
Altera recommends that you use the Avalon-ST interface for timing closure in incremental compilation flows, and compatibility with the hard IP interface.

Rx and Tx ports use a data/descriptor style interface, which presents the application with a descriptor bus containing the TLP header and a separate data bus containing the TLP payload. A single-cycle-turnaround handshaking protocol controls the transfer of data. Refer to [“Descriptor/Data Interface” on page 5-35](#) for more information.

Avalon-MM Interface

The PCI Express endpoint which results from the SOPC Builder flow is now available in the hard IP implementation. The endpoint comprises a PCI Express Avalon-MM bridge that interfaces to hard IP implementation with a soft IP implementation of the transaction layer optimized for the Avalon-MM protocol.

Figure 4-7. PCI Express MegaCore Function with Avalon-MM Interface



The PCI Express Avalon-MM bridge provides an interface between the PCI Express transaction layer and other SOPC Builder components across the system interconnect fabric.

Transaction Layer

The transaction layer sits between the application layer and the data link layer. It generates and receives transaction layer packets. [Figure 4-8](#) illustrates the transaction layer of a component with two initialized virtual channels (VCs). The transaction layer contains three general subblocks: the transmit datapath, the configuration space, and the receive datapath, which are shown with vertical braces in [Figure 4-8](#).



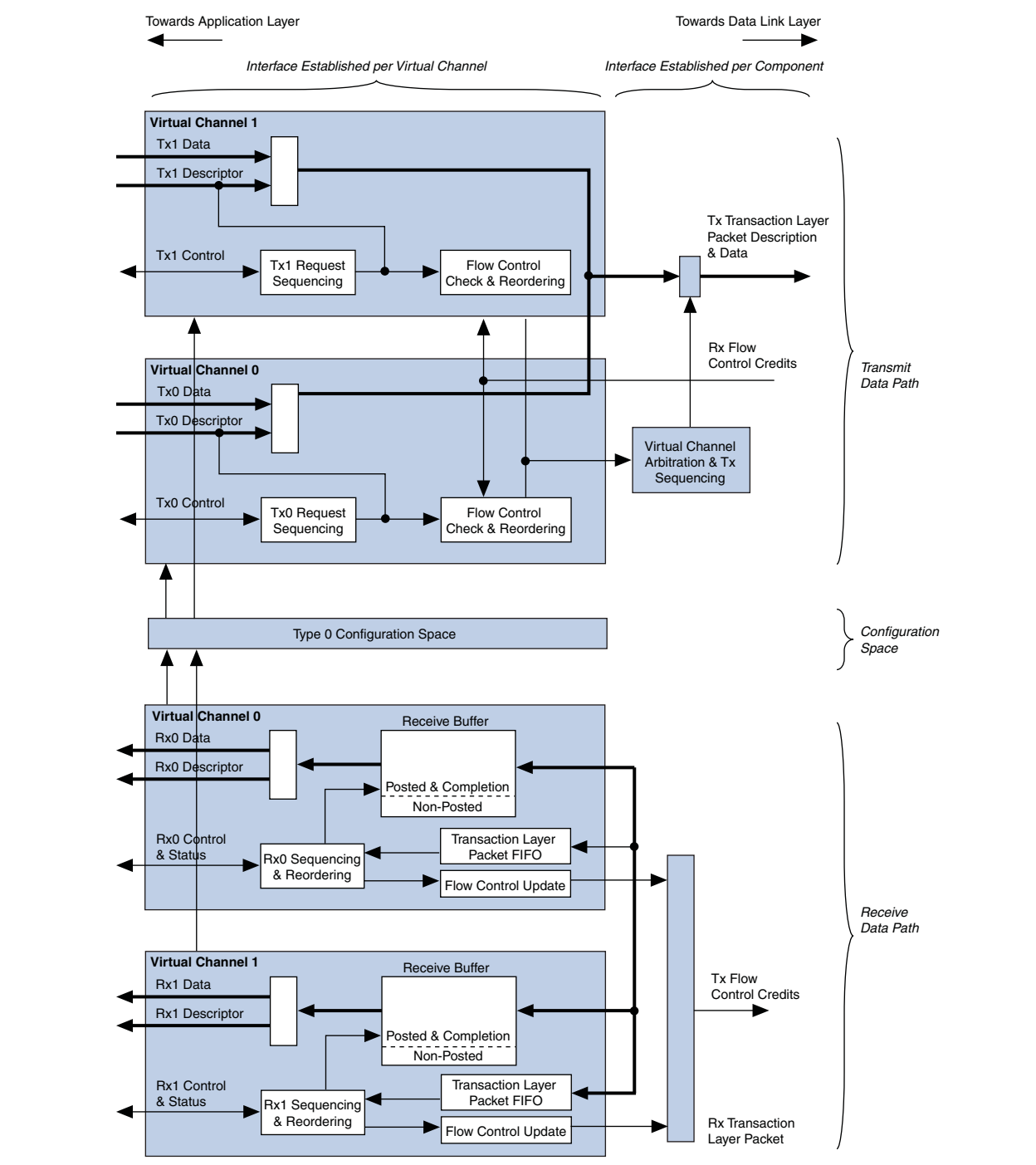
You can parameterize the Stratix IV GX MegaCore function to include one or two virtual channels. The Arria II GX implementation includes a single virtual channel.

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

1. The transaction layer receives a TLP from the data link layer.
2. The configuration space determines whether the transaction layer packet is well formed and directs the packet to the appropriate virtual channel based on traffic class (TC)/virtual channel (VC) mapping.
3. Within each virtual channel, transaction layer packets are stored in a specific part of the receive buffer depending on the type of transaction (posted, non-posted, and completion).
4. The transaction layer packet FIFO block stores the address of the buffered transaction layer packet.

- The receive sequencing and reordering block shuffles the order of waiting transaction layer packets as needed, fetches the address of the priority transaction layer packet from the transaction layer packet FIFO block, and initiates the transfer of the transaction layer packet to the application layer. Receive logic separates the descriptor from the data of the transaction layer packet and transfers the descriptor and data across the receive descriptor bus `rx_desc[135:0]` and receive data bus `rx_data[63:0]`, respectively, to the application layers.

Figure 4-8. Architecture of the Transaction Layer: Dedicated Receive Buffer per Virtual Channel



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

1. The MegaCore function informs the application layer with transmit credit (`tx_cred[21:0]` for the soft IP implementation and `tx_cred[35:0]` for the hard IP implementation) that sufficient flow control credits exist for a particular type of transaction. The application layer may choose to ignore this information.
2. The application layer requests a transaction layer packet transmission. The application layer must provide the PCI Express transaction header on the `tx_desc[127:0]` bus and be prepared to provide the entire data payload on the `tx_data[63:0]` bus in consecutive cycles.
3. The MegaCore function verifies that sufficient flow control credits exist, and acknowledges or postpones the request.
4. The transaction layer packet is forwarded by the application layer. The transaction layer arbitrates among virtual channels, and then forwards the priority transaction layer packet to the data link layer.

Transmit Virtual Channel Arbitration

The PCI Express MegaCore function allows you to divide the virtual channels into high and low priority groups as specified in Chapter 6 of the *PCI Express Base Specification 1.0a, 1.1 or 2.0*.

Arbitration of high-priority virtual channels is implemented using a strict priority arbitration scheme, in which higher numbered virtual channels always have higher priority than lower numbered virtual channels. Low-priority virtual channels use a fixed round robin arbitration scheme.


You can use the settings on the **Buffer Setup** page, accessible from the **Parameter Settings** tab in the MegaWizard interface, to specify the number of virtual channels and the number of virtual channels in the low priority group. Refer to “*Buffer Setup Parameters*” on page 3–8.

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
- Message Signaled Interrupt (MSI) Capability Structure
- PCI Express Capability Structure
- Virtual Channel Capabilities

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 in the direction of the PCI Express link. All such transactions are dependent upon the content of the PCI Express configuration space as described in the *PCI Express Base Specification Revision 1.0a, 1.1 or 2.0*.

 Refer To “Configuration Space Register Content” on page 4–40 or Chapter 7 in the *PCI Express Base Specification 1.0a, 1.1 or 2.0* for the complete content of these registers.

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 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 MegaCore function generates and transmits the completion.
- In root port mode, the application can issue type 0 or type 1 configuration TLPs on the Avalon-ST Tx bus.
 - The type 1 configuration TLPs are sent downstream on the PCI Express link toward the endpoint that matches the completer ID set in the transmit packet. If the bus number of the type 1 configuration TLP matches the Subordinate Bus Number register value in the root port configuration space, the TLP is converted to a type 0 TLP.
 - The type 0 configuration TLPs are only routed to the configuration space of the MegaCore function configured as a root port and are not sent downstream on the PCI Express link.
- The MegaCore function handles supported received message transactions (power management and slot power limit) internally.
- 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.
- 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 MegaCore function can generate and transmit power management, interrupt, and error signaling messages automatically under the control of dedicated signals. Additionally, the MegaCore function can generate MSI requests under the control of the dedicated signals.

Receive Buffer Reordering

The receive datapath implements a receive 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 masks non-posted request transactions made to the application interface so that only posted and completion transactions are presented to the application. Table 4–2 lists the transaction ordering rules.

Table 4-2. Transaction Ordering Rules (Note 1)–(12)

Row Pass Column		Posted Request		Non Posted Request				Completion			
		Memory Write or Message Request		Read Request		I/O or Cfg Write Request		Read Completion		I/O or Cfg Write Completion	
		Spec	Core	Spec	Core	Spec	Core	Spec	Core	Spec	Core
Posted	Memory Write or Message Request	1) N 2) Y/N	1) N 2) N	yes	yes	yes	yes	1) Y/N 2) Y	1) N 2) N	1) Y/N 2) Y	1) No 2) No
	Read Request	No	No	Y/N	1) Yes	Y/N	2) Yes	Y/N	No	Y/N	No
NonPosted	I/O or Configuration Write Request	No	No	Y/N	3) Yes	Y/N	4) Yes	Y/N	No	Y/N	No
	Read Completion	1) No 2) Y/N	1) No 2) No	Yes	Yes	Yes	Yes	1) Y/N 2) No	1) No 2) No	Y/N	No
Completion	I/O or Configuration Write Completion	Y/N	No	Yes	Yes	Yes	Yes	Y/N	No	Y/N	No

Notes to Table 4-2:

- (1) CfgRd0 can pass IORD or MRd.
- (2) CfgWr0 can IORD or MRd.
- (3) CfgRd0 can pass IORD or MRd.
- (4) CfgWr0 can pass IOWr.
- (5) 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.
- (6) 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.
- (7) Endpoints, Switches, and Root Complex may allow Memory Write and Message Requests to pass Completions or be blocked by Completions.
- (8) Memory Write and Message Requests can pass Completions traveling in the PCI Express to PCI directions to avoid deadlock.
- (9) If the Relaxed Ordering attribute is not set, then a Read Completion cannot pass a previously enqueued Memory Write or Message Request.
- (10) If the Relaxed Ordering attribute is set, then a Read Completion is permitted to pass a previously enqueued Memory Write or Message Request.
- (11) Read Completion associated with different Read Requests are allowed to be blocked by or to pass each other.
- (12) Read Completions for Request (same Transaction ID) must return in address order.

Data Link Layer

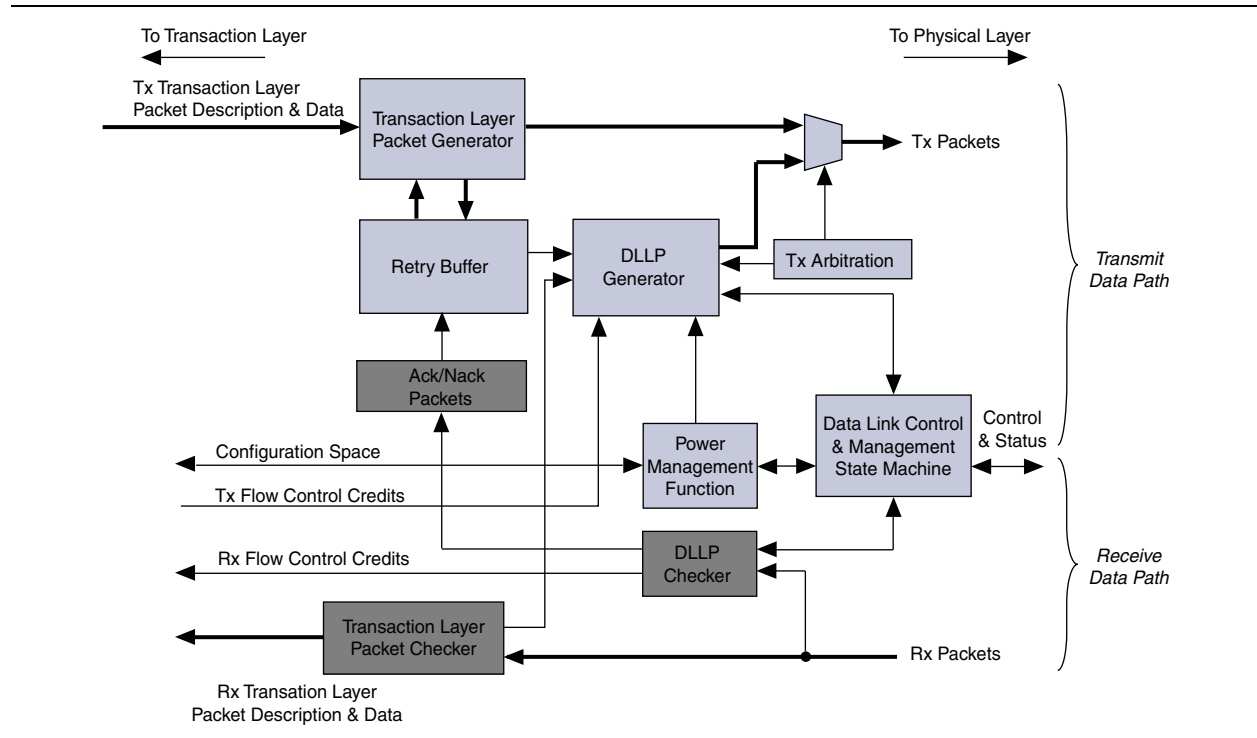
The data link layer is located between the transaction layer and the physical layer. It is responsible for maintaining packet integrity and for communication (by data link layer packet transmission) at the PCI Express link level (as opposed to component communication by transaction layer packet transmission in the interconnect fabric). Specifically, the data link layer is responsible for the following functions:

- Link management through the reception and transmission of data link layer packets, which are used for the following functions:
 - To initialize and update flow control credits for each virtual channel
 - For power management of data link layer packet reception and transmission
 - To transmit and receive ACK/NACK packets

- Data integrity through generation and checking of CRCs for transaction layer packets and data link layer packets
- Transaction layer packet retransmission in case of NAK data link layer packet reception using the retry buffer
- Management of the retry buffer
- Link retraining requests in case of error through the LTSSM of the physical layer

Figure 4-9 illustrates the architecture of the data link layer.

Figure 4-9. Data Link Layer



The data link layer has the following subblocks:

- **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 virtual channel flow control credits and reports status to the configuration space. (Virtual channel 0 is initialized by default, as are additional virtual channels if they have been physically enabled and the software permits them.)
- **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 PM DLLPs.
- **Data Link Layer Packet Generator and Checker**—This block is associated with the data link layer packet's 16-bit CRC and maintains the integrity of transmitted packets.

- Transaction Layer Packet Generator—This block generates transmit packets according to the descriptor and data received from the transaction layer, generating a sequence number and a 32-bit CRC. The packets are also sent to the retry buffer for internal storage. In retry mode, the transaction layer packet generator receives the packets from the retry buffer and generates the CRC for the transmit packet.
- Retry Buffer—The retry buffer stores transaction layer packets 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 data link layer packets and generates the sequence number of transmitted packets.
- Transaction Layer Packet Checker—This block checks the integrity of the received transaction layer packet and generates a request for transmission of an ACK/NAK data link layer packet.
- Tx Arbitration—This block arbitrates transactions, basing priority on the following order:
 1. Initialize FC data link layer packet
 2. ACK/NAK data link layer packet (high priority)
 3. Update FC data link layer packet (high priority)
 4. PM data link layer packet
 5. Retry buffer transaction layer packet
 6. Transaction layer packet
 7. Update FC data link layer packet (low priority)
 8. ACK/NAK FC data link layer packet (low priority)

Physical Layer

The physical layer is located at the lowest level of the MegaCore function. 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 running at 2.5 Gbps for Gen1 implementations and at 2.5/5.0 Gbps for Gen2 implementations. Only the hard IP implementation supports the Gen2 rate.

The physical layer is responsible for the following actions:

- Initializing the link
- Scrambling/descrambling and 8B10B encoding/decoding of 2.5 Gbps (Gen1) or 5.0 Gbps (Gen2) per lane 8B10B
- Serializing and deserializing data

The hard IP implementation includes the following additional functionality:

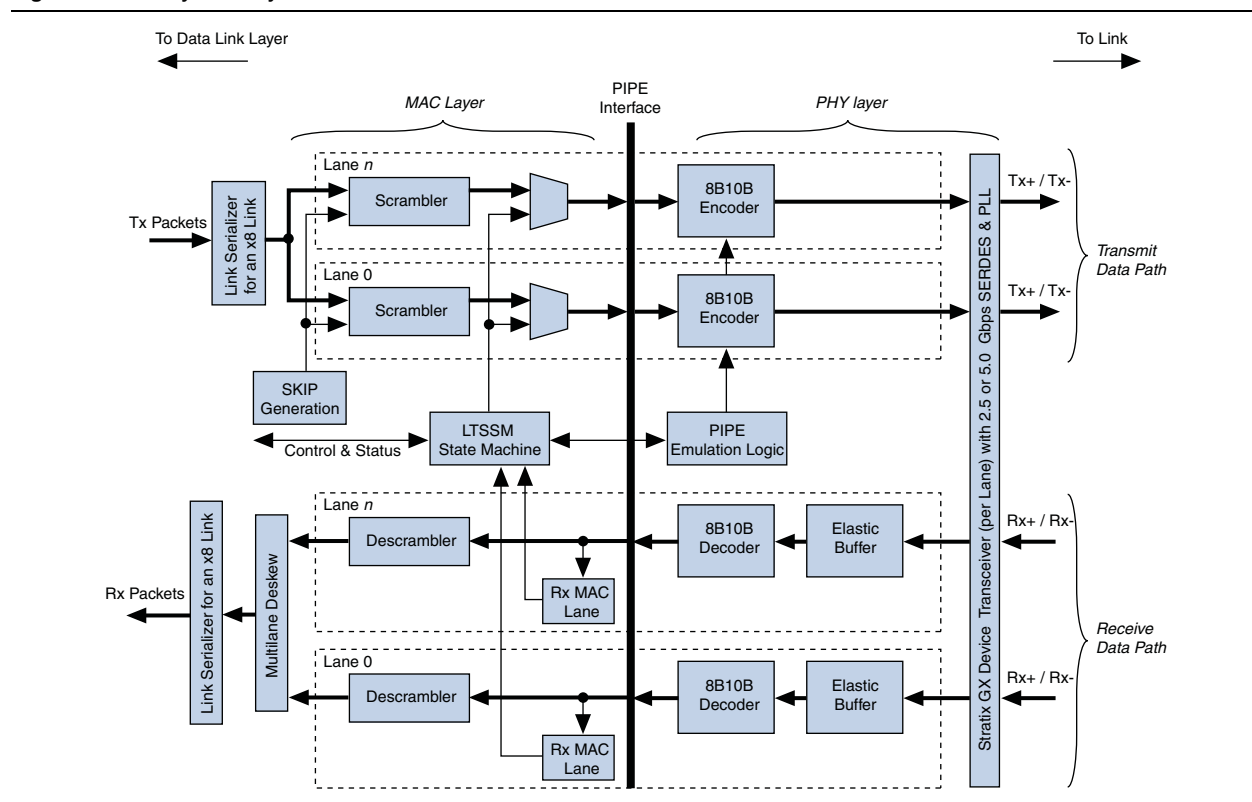
- PIPE 2.0 Interface Gen1/Gen2: 8-bit@250/500 MHz (fixed width, variable clock)
- Auto speed negotiation (Gen2)
- Training sequence transmission and decode

- Hardware autonomous speed control
- Auto lane reversal

Physical Layer Architecture

Figure 4–10 illustrates the physical layer architecture.

Figure 4–10. Physical Layer



The physical layer is subdivided by the PIPE Interface Specification into two layers (bracketed horizontally in Figure 4–10):

- **Media Access Controller (MAC) Layer**—The MAC layer includes the Link Training and Status state machine (LTSSM) and the scrambling/descrambling and multilane deskew functions.
- **PHY Layer**—The PHY layer includes the 8B10B 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 MegaCore function is compliant with the PIPE interface, allowing integration with other PIPE-compliant external PHY devices.

Depending on the parameters you set in the MegaWizard interface, the MegaCore function can automatically instantiate a complete PHY layer when targeting the Arria GX, Stratix GX, Stratix II GX, or Stratix IV GX devices.

The PHYMAC block is divided in four main sub-blocks:

- **MAC Lane**—Both the receive and the transmit path use this block.
 - On the receive side, the block decodes the physical layer packet (PLP) and reports to the LTSSM the type of TS1/TS2 received and the number of TS1s received since the LTSSM entered the current state. The LTSSM also reports the reception of FTS, SKIP and IDL ordered sets and the reception of eight consecutive D0.0 symbols.
 - On the transmit side, the block multiplexes data from the data link layer 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 PLPs reported by each MAC lane sub-block. It also enables the multilane deskew block and the delay required before the Tx alignment sub-block can move to the recovery or low power state. A higher layer can direct this block to move to the recovery, disable, hot reset or low power states through a simple request/acknowledge protocol. This block reports the physical layer status to higher layers.
- **LTSTX (Ordered Set and SKP Generation)**—This sub-block generates the physical layer packet (PLP). It receives control signals from the LTSSM block and generates PLP for each lane of the core. It generates the same PLP 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 SKIP ordered set at every predefined timeslot and interacts with the Tx alignment block to prevent the insertion of a SKIP 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 8-word FIFO for each lane to store symbols. Each symbol includes 8 data bits and 1 control bit. The FTS, COM, and SKP symbols are discarded by the FIFO; the PAD and IDL are replaced by 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 7 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 data link layer.

Lane Initialization and Reversal

Connected PCI Express components need not support the same number of lanes. The $\times 4$ MegaCore function supports initialization and operation with components that have 1, 2, or 4 lanes. The $\times 8$ MegaCore function supports initialization and operation with components that have 1, 4, or 8 lanes. The hard IP implementation includes lane reversal, which permits the logical reversal of lane numbers for the $\times 1$, $\times 4$, and $\times 8$ configurations.

Reversal allows more flexibility in board layout, reducing the number of signals that must cross over each other in the board design.

Table 4-3 summarizes the lane assignments for normal configuration.

Table 4-3. Lane Assignments without Reversal

Lane Number	7	6	5	4	3	2	1	0
$\times 8$ MegaCore function	7	6	5	4	3	2	1	0
$\times 4$ MegaCore function	—	—	—	—	3	2	1	0
$\times 1$ MegaCore function	—	—	—	—	—	—	—	0

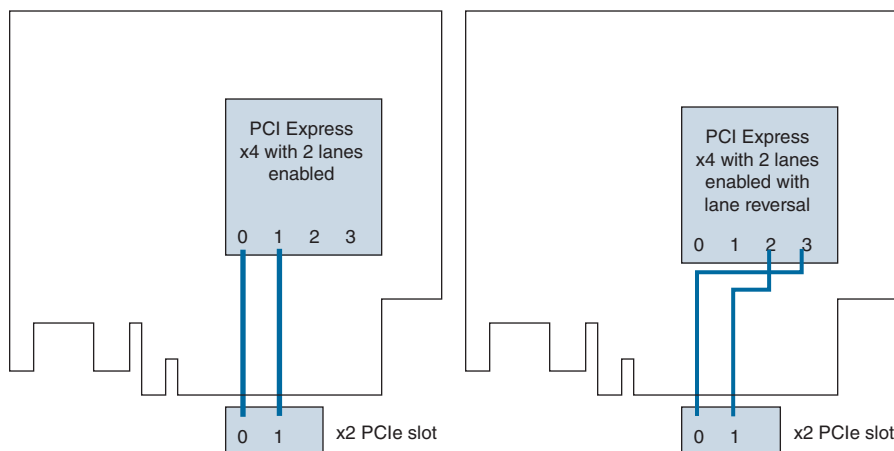
Table 4-4 summarizes the lane assignments with lane reversal.

Table 4-4. Lane Assignments with 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 4-11 illustrates a PCI Express card with a $\times 4$ MegaCore function down-configured to $\times 2$ with normal routing and lane-reversal. You can create $\times 2$ configuration by plugging a MegaCore function that was configured for $\times 8$ or $\times 4$ into a $\times 2$ slot.

Figure 4-11. PCI Express $\times 4$ Down-Configured to $\times 2$

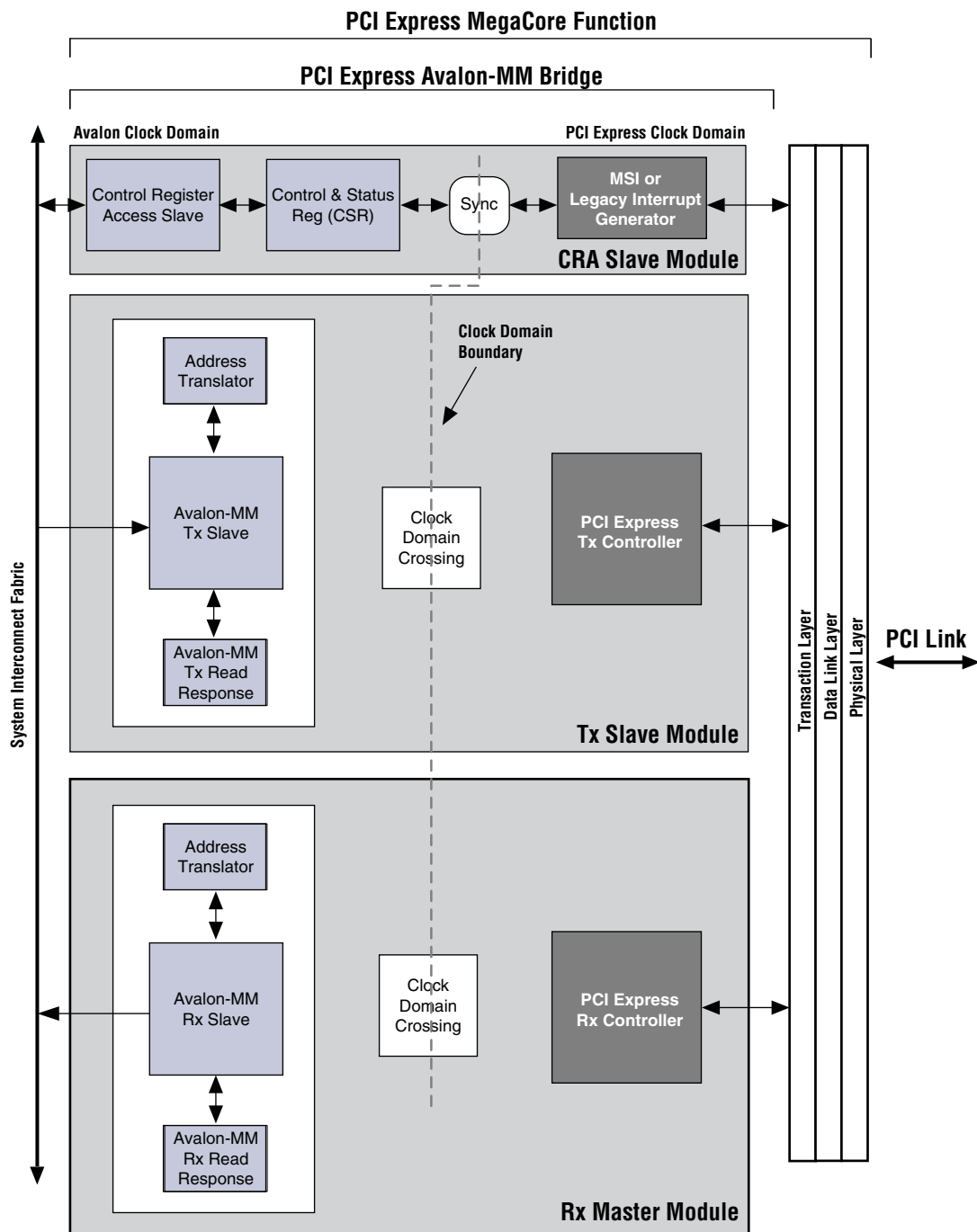


PCI Express Avalon-MM Bridge

The PCI Express Compiler configured using the SOPC Builder design flow uses the PCI Express Compiler's Avalon-MM bridge module to connect the PCI Express link to the system interconnect fabric. The bridge facilitates the design of PCI Express endpoints that include SOPC Builder components.

The full-featured PCI Express Avalon-MM bridge, shown in [Figure 4-12](#), provides three possible Avalon-MM ports: a bursting master, an optional bursting slave, and an optional non-bursting slave. The PCI Express Avalon-MM bridge comprises the following three modules:

- **Tx Slave Module**—This optional 64-bit bursting, Avalon-MM dynamic addressing slave port propagates write requests of up to 4 KBytes in size from the system interconnect fabric to the PCI Express link. It also propagates read requests of up to 512 bytes in size. The bridge translates requests from the interconnect fabric to PCI Express request packets.
- **Rx Master Module**—This 64-bit bursting Avalon-MM master port propagates PCI Express requests, converting them to bursting read or write requests to the system 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 for upstream PCI Express devices and external Avalon-MM masters. Implementations that use MSI or dynamic address alignment require this port.

Figure 4-12. PCI Express Avalon-MM Bridge

The PCI Express Avalon-MM bridge supports the following TLPs:

- Memory write requests
- Received downstream memory read requests of up to 512 bytes in size
- Transmitted upstream memory read requests of up to 256 bytes in size
- Completion



The PCI Express Avalon-MM bridge supports native PCI Express endpoints, but not legacy PCI Express endpoints. Therefore, the bridge does not support I/O space BARs and I/O space requests cannot be generated.

The bridge has the following additional characteristics:

- Vendor-defined incoming messages are discarded
- Completion-to-a-flush request is generated, but not propagated to the system interconnect fabric

Each PCI Express base address register (BAR) in the transaction layer maps to a specific, fixed Avalon-MM address range. Separate BARs can be used to map to various Avalon-MM slaves connected to the Rx Master port.

The following sections describe the modes of operation:

- [Avalon-MM-to-PCI Express Write Requests](#)
- [Avalon-MM-to-PCI Express Upstream Read Requests](#)
- [PCI Express-to-Avalon-MM Read Completions](#)
- [PCI Express-to-Avalon-MM Downstream Write Requests](#)
- [PCI Express-to-Avalon-MM Downstream Read Requests](#)
- [PCI Express-to-Avalon-MM Read Completions](#)
- [Avalon-MM-to-PCI Express Address Translation](#)
- [Generation of PCI Express Interrupts](#)
- [Generation of Avalon-MM Interrupts](#)

Avalon-MM-to-PCI Express Write Requests

The PCI Express Avalon-MM bridge accepts Avalon-MM burst write requests with a burst size of up to 4 KBytes at the Avalon-MM Tx slave interface. It 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 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 enable 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 enable 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 Avalon-MM Tx slave interface can receive read requests with burst sizes of up to 4 KBytes 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 enable 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 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

When the PCI Express Avalon-MM bridge receives PCI Express write requests, it converts them to burst write requests before sending them to the system interconnect fabric. 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 MegaCore function parameterization.

PCI Express-to-Avalon-MM Downstream Read Requests

The PCI Express Avalon-MM bridge sends PCI Express read packets to the system interconnect fabric as burst reads with a maximum burst size of 512 bytes. 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 address translation lookup table values are user configurable. Malformed write packets are dropped, and therefore do not appear on the Avalon-MM interface. Unsupported read requests generate a completer abort response.

For downstream write and read requests, if more than one byte enable is asserted, the byte lanes must be adjacent. As an example, [Table 4–5](#) gives the byte enables for a 32-bit data.

Table 4–5. Valid Byte Enable Configurations (Part 1 of 2)

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

Table 4-5. Valid Byte Enable Configurations (Part 2 of 2)

Byte Enable Value	Description
4'b0100	Write byte 2 only
4'b1000	Write byte 3 only

Avalon-MM-to-PCI Express Read Completions

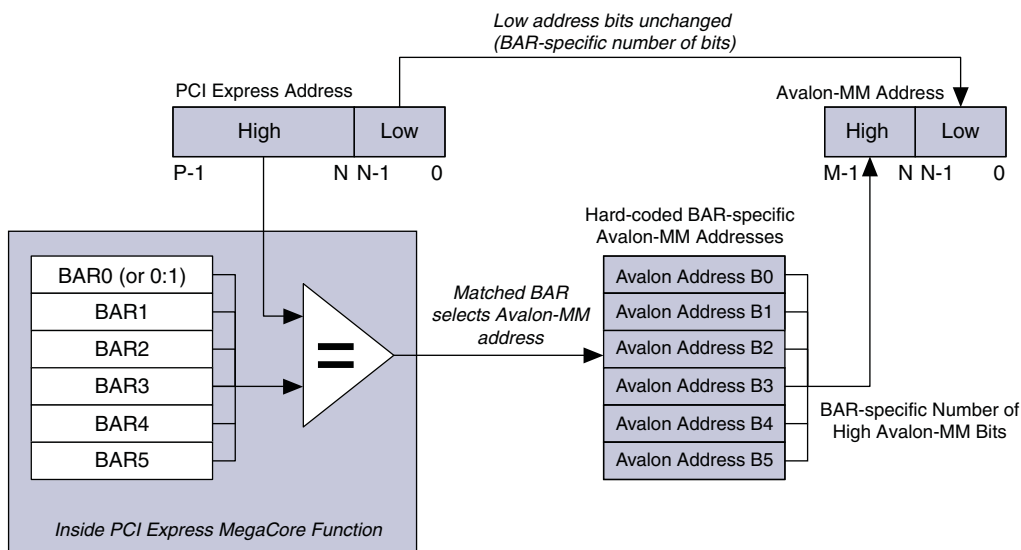
The PCI Express Avalon-MM bridge converts read response data from the external Avalon-MM slave to PCI Express completion packets and sends them to the transaction layer.

A single read request may produce multiple completion packets based on the `MAX_PAYLOAD_SIZE` and the size of the received read request. For example, if the read is 512 bytes but the `MAX_PAYLOAD_SIZE` 128 bytes, the bridge produces four completion packets of 128 bytes each. The bridge does not generate out-of-order completions.

PCI Express-to-Avalon-MM Address Translation

The PCI Express address of a received request packet is translated to the Avalon-MM address before the request is sent to the system interconnect fabric. This address translation proceeds by replacing the MSB bits of the PCI Express address with the value from a specific translation table entry; the LSB bits remain unchanged. The number of MSB bits to replace is calculated from the total memory allocation of all Avalon-MM slaves connected to the Rx Master Module port. Six possible address translation entries in the address translation table are configurable by the user or by SOPC Builder. Each entry corresponds to a PCI Express BAR. The BAR hit information from the request header determines the entry that is used for address translation. Figure 4-13 depicts the PCI Express Avalon-MM bridge address translation process.

Figure 4-13. PCI Express Avalon-MM Bridge Address Translation (Note 1)



Note to Figure 4-13:

(1) *N* is the number of pass-through bits (BAR specific). *M* is the number of Avalon-MM address bits. *P* is the number of PCI Express address bits (64/32)

The Avalon-MM Rx master module port has an 8-byte data path. This 8-byte wide data path therefore means that native address alignment Avalon-MM slaves that are connected to the Rx master module port will have their internal registers at 8-byte intervals in the PCI Express address space. When reading or writing a native address alignment Avalon-MM Slave (such as the SOPC Builder DMA controller core) the PCI Express address should increment by eight bytes to access each successive register in the native address slave.



For more information, refer to the “Native Address Alignment and Dynamic Bus Sizing” section in the *System Interconnect Fabric for Memory-Mapped Interfaces* chapter in volume 4 of the *Quartus II Handbook*.

Avalon-MM-to-PCI Express Address Translation

The Avalon-MM address of a received request on the Tx Slave Module port is translated to the PCI Express address before the request packet is sent to the transaction layer. This address translation process proceeds by replacing the MSB bits of the Avalon-MM address with the value from a specific translation table entry; the LSB bits remain unchanged. The number of MSB bits to be replaced is calculated based on the total address space of the upstream PCI Express devices that the PCI Express MegaCore function can access.

The address translation table contains up to 512 possible address translation entries that you can configure. Each entry corresponds to a base address of the 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 bits to be replaced. In addition, each entry has a 2-bit field, $S_p[1:0]$, that specifies 32-bit or 64-bit PCI Express addressing for the translated address. Refer to [Figure 4-14](#). The most significant bits of the Avalon-MM address are used by the system interconnect fabric to select this slave port, these bits 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 4-14](#) 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:

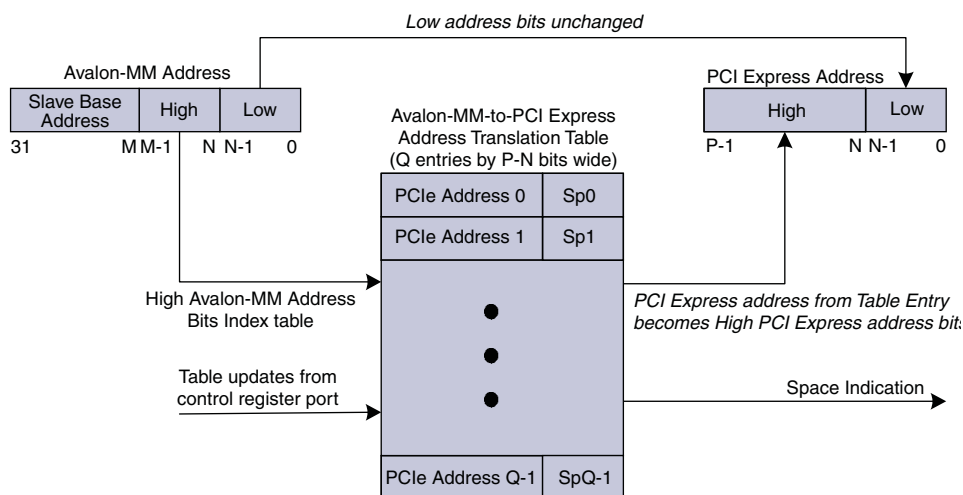
- Bits [31:24] select the Tx slave module port from among other slaves connected to the same master by the system interconnect fabric. The decode is based on the base addresses assigned in the SOPC Builder screen.
- Bits [23:20] select the address translation table entry.

- 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 can be hardwired or dynamically configured at run time. When the MegaCore function is parameterized for dynamic address translation, the address translation table is implemented in memory and can be accessed through the CRA slave module. This access mode is useful 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 control register access slave, refer to the “[Avalon-MM-to-PCI Express Address Translation Table](#)” on page 4-47. Figure 4-14 depicts the Avalon-MM-to-PCI Express address translation process.

Figure 4-14. Avalon-MM-to-PCI Express Address Translation (Note 1) – (Note 5)



Notes to Figure 4-14:

- (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.

Generation of PCI Express Interrupts

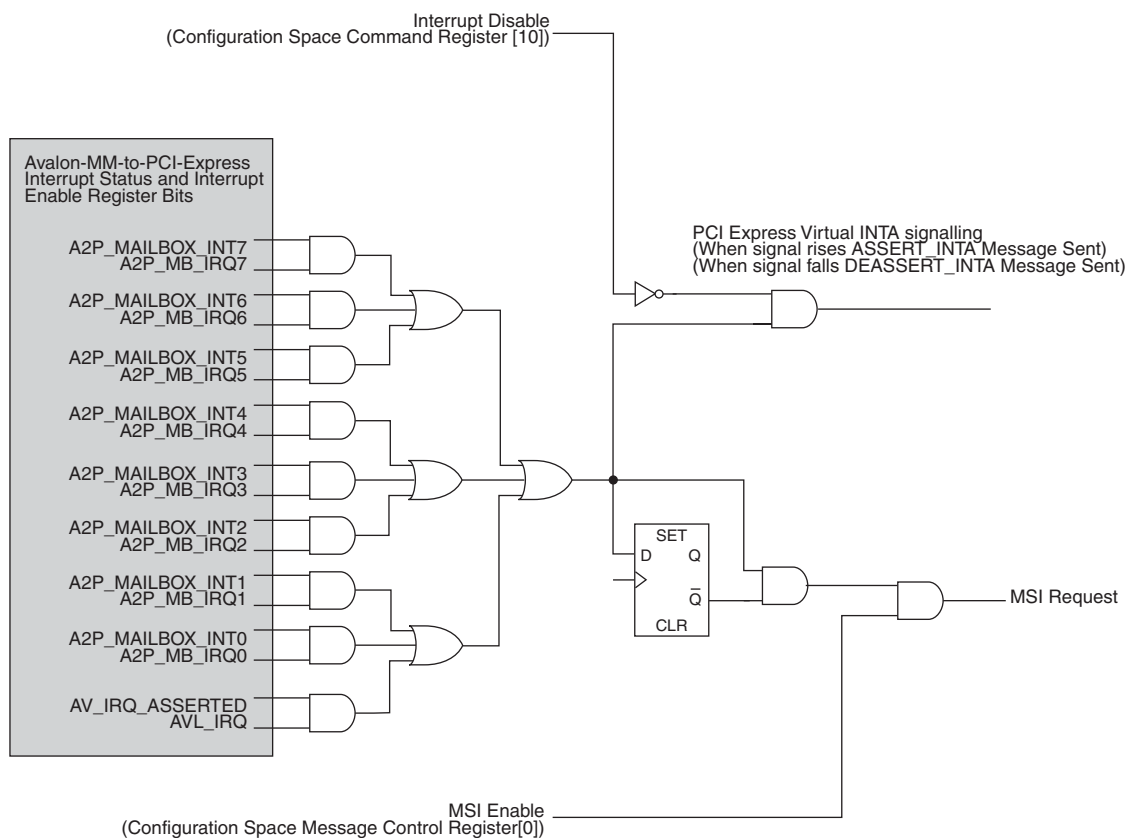
The PCI Express Avalon-MM bridge supports MSI or legacy interrupts. Interrupt support requires instantiation of the CRA slave module where the interrupt registers and control logic are implemented.

The Rx master module port has an Avalon-MM interrupt ($Rxmlrq_i$) input. Assertion of this signal or a PCI Express mailbox register write access sets a bit in the PCI Express interrupt status register and generates a PCI Express interrupt, if enabled. Software can enable the “[Avalon-MM to PCI Express Interrupt Status Register](#)” on page 4-45 by writing to the PCI Express “[Avalon-MM to PCI Express Interrupt Enable Register](#)” on page 4-46 through the CRA slave. When the IRQ input is asserted, the IRQ vector is written to the “[Avalon-MM to PCI Express Interrupt Status Register](#)” on page 4-45, accessible by the CRA slave. 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 “PCI Express to Avalon-MM Interrupt Status Register” on page 4-48 mailbox writes, the status bits should be cleared in the “Avalon-MM to PCI Express Interrupt Status Register” on page 4-45. For interrupts due to the RxmIrq_i signal, the interrupt status should be cleared in the other Avalon peripheral that sourced the interrupt. This sequence prevents interrupts from being lost during interrupt servicing.

Figure 4-15 shows the logic for the entire PCI Express interrupt generation process.

Figure 4-15. PCI Express Avalon-MM 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. The Interrupt Disable bit, which is bit 10 of the Command register (Table 4-11) can be used to disable legacy interrupts. The MSI enable bit, which is bit 0 of the MSI Control Status register () in the MSI capability shown in Table 4-13 on page 4-42, can be used to enable MSI interrupts. Only one type of interrupt can be enabled at a time.

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 (`CraIrq_o`) output. A write access to an Avalon-MM mailbox register sets one of the `P2A_MAILBOX_INT<n>` bits in the “PCI Express to Avalon-MM Interrupt Status Register” on page 4-48 and asserts the `CraIrq_o` output, if enabled. Software can enable the interrupt by writing to the “PCI Express to Avalon-MM Interrupt Enable Register Address: 0x3070” on page 4-49 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 there is no other interrupt status pending.

Analyzing Throughput

Throughput analysis requires that you understand the Flow Control Loop, shown in Figure 4-16 on page 4-28. This section discusses the Flow Control Loop and points that help you improve throughput.

Throughput of Posted Writes

The throughput of posted writes is limited primarily by the Flow Control Update loop shown in Figure 4-16. If the requester of the writes sources the data as quickly as possible, and the completer of the writes 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 4-16 shows the main components of the Flow Control Update loop with two communicating PCI Express ports:

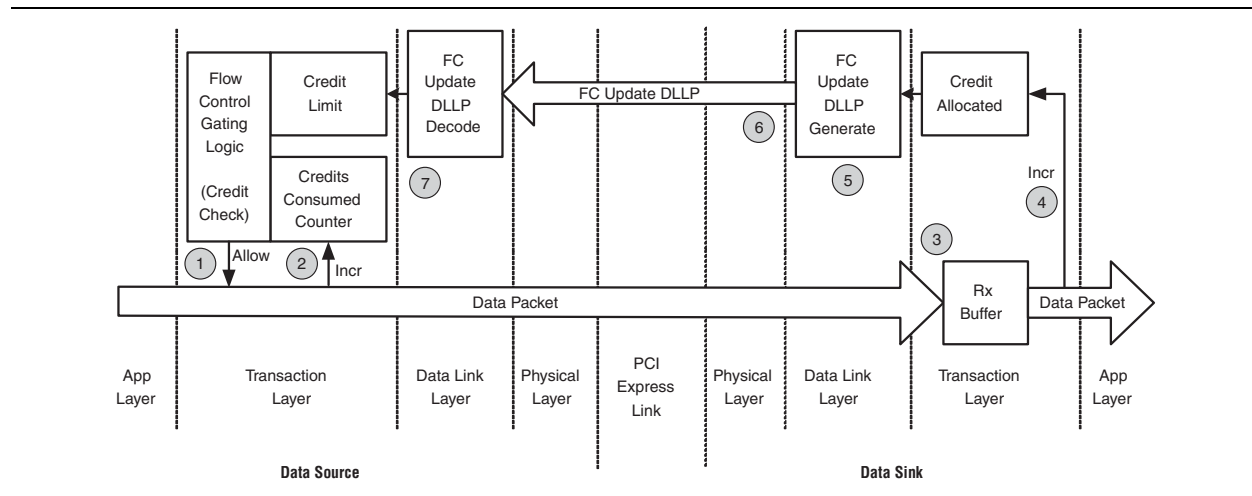
- Write Requester
- Write Completer

As the PCI Express specification describes, 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 4-16. Flow Control Update Loop



The following numbered steps describe each step in the Flow Control Update loop. The corresponding numbers on [Figure 4-16](#) 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 to transmit, 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 transport layer protocols (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 μ s 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 stalled waiting for credits, they now can be transmitted.

To allow the write requester in the above description 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 freeing of the credits from the very first TLP transmitted.

Table 4–6 shows the delay components for the FC Update Loop when the PCI Express MegaCore function is implemented in a Stratix II GX device. The delay components are independent of the packet length. The total delays in the loop increase with packet length.

Table 4–6. FC Update Loop Delay Components For Stratix II GX (Part 1 of 2) (*Note 1*), (*Note 2*)

Delay in Nanoseconds	×8 Function		×4 Function		×1 Function	
	Min	Max	Min	Max	Min	Max
From decrement transmit credit consumed counter to PCI Express Link.	60	68	104	120	272	288
From PCI Express Link until packet is available at Application Layer interface.	124	168	200	248	488	536
From Application Layer draining packet to generation and transmission of Flow Control (FC) Update DLLP on PCI Express Link (assuming no arbitration delay).	60	68	120	136	216	232

Table 4-6. FC Update Loop Delay Components For Stratix II GX (Part 2 of 2) (Note 1), (Note 2)

Delay in Nanoseconds	×8 Function		×4 Function		×1 Function	
	Min	Max	Min	Max	Min	Max
From receipt of FC Update DLLP on the PCI Express Link to updating of transmitter's Credit Limit register.	116	160	184	232	424	472

Note to Table 4-6:

- (1) The numbers for other Gen1 PHYs are similar.
 (2) Gen2 numbers are to be determined.

Based on the above FC Update Loop delays and additional arbitration and packet length delays, Table 4-7 shows the number of flow control credits that must be advertised to cover the delay. The Rx buffer size must support this number of credits to maintain full bandwidth.

Table 4-7. Data Credits Required By Packet Size

Max Packet Size	×8 Function		×4 Function		×1 Function	
	Min	Max	Min	Max	Min	Max
128	64	96	56	80	40	48
256	80	112	80	96	64	64
512	128	160	128	128	96	96
1024	192	256	192	192	192	192
2048	384	384	384	384	384	384

These numbers take into account the device delays at both ends of the PCI Express link. Different devices at the other end of the link could have smaller or larger delays, which affects the minimum number of credits required. In addition, if the application layer cannot drain received packets immediately in all cases, it may be necessary to offer additional credits to cover this delay.

Setting the **Desired performance for received requests** to **High** on the **Buffer Setup** page on the **Parameter Settings** tab in the MegaWizard interface configures the Rx buffer with enough space to meet the above required credits. You can adjust the **Desired performance for received request** up or down from the **High** setting to tailor the Rx buffer size to your delays and required performance.

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 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 PCI Express MegaCore function 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. PCI Express endpoints must offer an infinite number of completion credits. The PCI Express MegaCore function must buffer this data in the Rx buffer until the application can process it. Because the PCI Express MegaCore function is no longer managing the Rx buffer through the flow control mechanism, the application 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 4-27](#). 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.

Assuming there is a PCI Express switch in the path between the read requester and the read completer and assuming typical read completion times for root ports, [Table 4-8](#) shows the estimated completion space required to cover the read transaction’s round trip delay.

Table 4-8. Completion Data Space (in Credit units) to Cover Read Round Trip Delay

Max Packet Size	×8 Function Typical	×4 Function Typical	×1 Function Typical
128	120	96	56
256	144	112	80
512	192	160	128
1024	256	256	192
2048	384	384	384
4096	768	768	768

Note also that the completions can be broken up into multiple completions of smaller packet size. In this scheme, 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 **Desired performance for received completions** to **High** on the **Buffer Setup** page when specifying parameter settings in your MegaCore function configures the Rx buffer with enough space to meet the above requirements. You can adjust the **Desired performance for received completions** 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 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 PCI Express MegaCore function. For the ×8 function, you can specify 32 tags. For the ×1 and ×4 functions, you can specify up to 256 tags, though configuration software can restrict the application to use only 32 tags. In commercial PC systems, 32 tags are typically sufficient to maintain optimal read throughput.

PCI Express Reconfiguration Block

The PCI Express reconfiguration block allows you to dynamically change configuration registers in the hard IP implementation of the PCI Express MegaCore function, if those registers were created using the MegaWizard Plug-In Manager design flow. The PCIE Reconfig module implements this functionality. You access this module using its Avalon-MM slave interface. A single Avalon-MM master can drive multiple PCIE Reconfig slave modules. For a complete description of the signals in this interface, refer to “[PCI Express Reconfiguration Block Signals—Hard IP Implementation](#)” on page 5–31. The PCIE Reconfig module provides access to configuration registers, including configuration space, link configuration, MSI and MSI-X capabilities, power management, and advanced error reporting. These registers are normally read-only at run time.

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

1. Bring down the PCI Express link by asserting the `npor` reset signal.
2. Reprogram HIP configuration registers using the Avalon-MM slave PCIE Reconfig interface.
3. Release the `npor` reset signal.

[Table 4–9](#) lists all of the registers that you can update using the PCI Express reconfiguration block interface.

Table 4–9. Dynamically Reconfigurable in the Hard IP Implementation (Part 1 of 8)

Address	Bits	Description	Default Value	Additional Information
0x0	0	When 0, PCIe reconfig mode is enabled. When 1, PCIe reconfig mode is disabled and the original configuration is used.	b'1	—
	1	When 1, drives the reset (<code>npor</code>) to the hard IP implementation of PCI Express MegaCore function.	b'0	
0x001-0x88		Reserved.	—	
0x0089	15:0	Vendor ID.	0x1172	Table 4–11 , Table 4–12
0x008A	15:0	Device ID.	0x0001	Table 4–11 , Table 4–12
	7:0	Revision ID.	0x01	Table 4–11 , Table 4–12
0x008B	15:8	Class code[7:0].	—	Table 4–11 , Table 4–12
0x008C	15:0	Class code[23:8].	—	Table 4–11
0x008D	15:0	Subsystem vendor ID.	0x1172	Table 4–11
0x008E	15:0	Subsystem device ID.	0x0001	Table 4–11
0x008F		Reserved.	—	
0x0090	0	Advanced Error Reporting.	b'0	Table 4–18 Port VC Cap 1
	3:1	Low Priority VC (LPVC).	b'000	
	7:4	VC arbitration capabilities.	b'00001	
	15:8	Reject Snoop Transaction.	b'00000000	Table 4–18 VC Resource Capability register

Table 4–9. Dynamically Reconfigurable in the Hard IP Implementation (Part 2 of 8)

Address	Bits	Description	Default Value	Additional Information
	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 4–17, Device Capability register
	3	Surprise Down error reporting capabilities. (Available in <i>PCI Express Base Specification Revision 1.1</i> 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 4–17, Link Capability register
	4	Data Link Layer active reporting capabilities. (Available in <i>PCI Express Base Specification Revision 1.1</i> 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 4–17, Link Capability register
	5	Extended TAG field supported.	b'0	Table 4–17, 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 μ s. b'101 – Maximum of 2 μ s. b'110 – Maximum of 4 μ s. b'111– No limit.	b'000	Table 4–17, Device Capability register

Table 4–9. Dynamically Reconfigurable in the Hard IP Implementation (Part 3 of 8)

Address	Bits	Description	Default Value	Additional Information
0x0091	11:9	Endpoint L1 acceptable latency. The following encodings are defined: b'000 – Maximum of 1 μ s. b'001 – Maximum of 2 μ s. b'010 – Maximum of 4 μ s. b'011 – Maximum of 8 μ s. b'100 – Maximum of 16 μ s. b'101 – Maximum of 32 μ s. b'110 – Maximum of 64 μ s. b'111 – No limit.	b'000	Table 4–17, 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 4–17, Slot Capability register
	15	Role-Based error reporting. (Available in <i>PCI Express Base Specification Revision 1.1</i> compliant Cores only.) In 1.1 compliant cores, this bit should be set to 1.	b'1	Table 4–19, Correctable Error Mask register
	1:0	Slot Power Limit Scale.	b'00	Table 4–17, Slot Capability register

Table 4–9. Dynamically Reconfigurable in the Hard IP Implementation (Part 4 of 8)

Address	Bits	Description	Default Value	Additional Information
0x0092	7:2	Max Link width.	b'000100	Table 4–17, Link Capability register
	9:8	L0s Active State power management support. L1 Active State power management support.	b'01	Table 4–17, Link Capability register
	15:10	L1 exit latency common clock. L1 exit latency separated clock. The following encodings are defined: b'000 – Less than 1 μ s. b'001 – 1 μ s to less than 2 μ s. b'010 – 2 μ s to less than 4 μ s. b'011 – 4 μ s to less than 8 μ s. b'100 – 8 μ s to less than 16 μ s. b'101 – 16 μ s to less than 32 μ s. b'110 – 32 μ s to 64 μ s. b'111 – More than 64 μ s.	b'000000	Table 4–17, Link Capability register
0x0093		[0]: Attention button implemented on the chassis.	b'0000000	Table 4–17, 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 to 1, a device can be removed from this slot without prior notification.		
	6:0	[6]: Hot-plug capable.	b'000	
	9:7	Reserved.		
	15:10	Slot Power Limit Value.	b'00000000	
0x0094	1:0	Reserved.	—	Table 4–17, Slot Capability register
	2	Electromechanical Interlock present (Available in <i>PCI Express Base Specification Revision 1.1</i> compliant Megacore functions 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	
0x0095	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 4–9. Dynamically Reconfigurable in the Hard IP Implementation (Part 5 of 8)

Address	Bits	Description	Default Value	Additional Information
0x0096	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 4–17, Device Capability register 2
	4	Completion Timeout supported 0: completion timeout disable not supported 1: completion timeout disable supported	b'0	Table 4–17, Device Capability register 2
	7:5	Reserved.	b'0	—
	8	ECRC generate.	b'0	Table 4–19, Advanced Error Capability and Control register
	9	ECRC check.	b'0	Table 4–19, Advanced Error Capability and Control register
	10	No command completed support. (available only in <i>PCI Express Base Specification Revision 1.1</i> compliant Cores)	b'0	Table 4–17, 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 4–13, Message Control register
	14	MSI 32/64-bit addressing mode. b'0: 32 bits only. b'1: 32 or 64 bits	b'1	
0x0096	15	MSI per-bit vector masking (read-only field).	b'0	
0x0097	0	Function supports MSI.	b'1	Table 4–13, Message Control register for MSI
	3:1	Interrupt pin.	b'001	—
	5:4	Reserved.	b'00	
	6	Function supports MSI-X.	b'0	Table 4–13, Message Control register for MSI
	15:7	MSI-X table size	b'0	Table 4–14, MSI-X Capability Structure
0x0098	1:0	Reserved.	—	Table 4–14, MSI-X Capability Structure
	4:2	MSI-X Table BIR.	b'0	
	15:5	MSI-X Table Offset.	b'0	

Table 4–9. Dynamically Reconfigurable in the Hard IP Implementation (Part 6 of 8)

Address	Bits	Description	Default Value	Additional Information
0x0099	15:10	MSI-X PBA Offset.	b'0	—
0x009A	15:0	Reserved.	b'0	
0x009B	15:0	Reserved.	b'0	
0x009C	15:0	Reserved.	b'0	
0x009D	15:0	Reserved.	b'0	
0x009E	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	
0x009F	7:0	Number of NFTS for common clock in Gen2 rate.	b'11111111	Table 4–17, 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 4–17, PCI Express capability register
	15:13	LOs 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 4–17, Link Capability register
	2:0	LOs 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 μ s. b'101 – 1 μ s to less than 2 μ s. b'110 – 2 μ s to 4 μ s. b'111 – More than 4 μ s.	b'110	Table 4–17, Link Capability register
0x00A0	15:3	Reserved.	0x0000	

Table 4–9. Dynamically Reconfigurable in the Hard IP Implementation (Part 7 of 8)

Address	Bits	Description	Default Value	Additional Information
0x00A1		BAR0[31:0].		Table 4–11, Table 4–12
	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.	0xFFFFFFFF	
	15:4	BAR0[15:4].	b'0	
0x00A2	15:0	BAR0[31:16].	b'0	
0x00A3		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	
0x00A4	15:0	BAR1[63:48].	b'0	
0x00A5		BAR2[95:64].	b'0	Table 4–11
	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	
0x00A6	15:0	BAR2[95:80].	b'0	

Table 4–9. Dynamically Reconfigurable in the Hard IP Implementation (Part 8 of 8)

Address	Bits	Description	Default Value	Additional Information
0x00A7		BAR3[127:96].	b'0	Table 4–11
	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	
0x00A8	15:0	BAR3[127:112].	b'0	
0x00A9		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	
0x00AA	15:0	BAR4[159:144].	b'0	
0x00AB		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	
0x00AC	15:0	BAR5[191:176].	b'0	
0x00AD	15:0	Expansion BAR[223:192]: Bar size mask.	b'0	
		Expansion BAR[207:192].	b'0	
0x00AE	15:0	Expansion BAR[223:208].	b'0	
0x00AF	1:0	IO. 00: no IO windows. 01: IO 16 bit. 11: IO 32-bit.	b'0	Table 4–12
	3:2	Prefetchable. 00: not implemented. 01: prefetchable 32. 11: prefetchable 64.	b'0	
	15:4	Reserved.	—	
0xFFFF-00B0		Reserved.	—	—

Configuration Space Register Content

This section describes the PCI Express configuration space registers.



For more information about these registers, refer to Chapter 7 of the *PCI Express Base Specification Revision 1.0a, 1.1 or 2.0* depending on the version you specify on the **System Setting** page of the MegaWizard interface.

Table 4–10 shows the common configuration space header. The following tables provide more details.

Table 4–10. Common Configuration Space Header

Byte Offset	31:24	23:16	15:8	7:0
0x000:0x03C	PCI Header Type 0 configuration registers (refer to Table 4–11 for details.)			
0x000:0x03C	PCI Header Type 1 configuration registers (refer to Table 4–12 for details.)			
0x040h:0x04C	Reserved			
0x050:0x05C	MSI capability structure (refer to Table 4–13 for details.)			
0x060:0x074	Reserved			
0x078:0x07C	Power management capability structure (refer to Table 4–15 for details.)			
0x080:0x0B8	PCI Express capability structure (refer to Table 4–16 for details.)			
0x0B8:0x0FC	Reserved			
0x094:0x0FF	Root port			
0x100:0x16C	Virtual channel capability structure (refer to Table 4–18 for details.)			
0x170:0x17C	Reserved			
0x180:0x1FC	Virtual channel arbitration table			
0x200:0x23C	Port VC0 arbitration table (Reserved)			
0x240:0x27C	Port VC1 arbitration table (Reserved)			
0x280:0x2BC	Port VC2 arbitration table (Reserved)			
0x2C0:0x2FC	Port VC3 arbitration table (Reserved)			
0x300:0x33C	Port VC4 arbitration table (Reserved)			
0x340:0x37C	Port VC5 arbitration table (Reserved)			
0x380:0x3BC	Port VC6 arbitration table (Reserved)			
0x3C0:0x3FC	Port VC7 arbitration table (Reserved)			
0x400:0x7FC	Reserved			
0x800:0x834	Advanced Error Reporting AER (optional)			
0x838:0xFFF	Reserved			

Table 4–11 describes the type 0 configuration settings.

Table 4–11. PCI Header Type 0 Configuration Settings

Byte Offset	31:24	23:16	15:8	7:0
0x0000	Device ID		Vendor ID	
0x004	Status		Command	
0x008	Class Code			Revision ID
0x00C	0x00	Header Type	0x00	Cache Line Size

Table 4-11. PCI Header Type 0 Configuration Settings

Byte Offset	31:24	23:16	15:8	7:0
0x010	Base Address 0			
0x014	Base Address 1			
0x018	Base Address 2			
0x01C	Base Address 3			
0x020	Base Address 4			
0x024	Base Address 5			
0x028	Reserved			
0x02C	Subsystem Device ID		Subsystem Vendor ID	
0x030	Expansion ROM base address			
0x034	Reserved			Capabilities PTR
0x038	Reserved			
0x03C	0x00	0x00	Interrupt Pin	Interrupt Line

Table 4-12 describes the type 1 configuration settings.

Table 4-12. PCI Header Type 1 Configuration Settings

Byte Offset	31:24	23:16	15:8	7:0
0x0000	Device ID		Vendor ID	
0x004	Status		Command	
0x008	Class Code			Revision ID
0x00C	BIST	Header Type	Primary Latency Timer	Cache Line Size
0x010	Base Address 0			
0x014	Base Address 1			
0x018	Secondary Latency Timer	Subordinate Bus Number	Secondary Bus Number	Primary Bus Number
0x01C	Secondary Status		I/O Limit	I/O Base
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 PTR
0x038	Expansion ROM Base Address			
0x03C	Bridge Control		Interrupt Pin	Interrupt Line

Table 4–13 describes the MSI capability structure.

Table 4–13. MSI Capability Structure

Byte Offset	31:24	23:16	15:8	7:0
0x050	Message Control		Next Cap Ptr	Capability ID
0x054	Message Address			
0x058	Message Upper Address			
0x05C	Reserved		Message Data	

Table 4–14 describes the MSI-X capability structure.

Table 4–14. MSI-X Capability Structure

Byte Offset	31:24	23:16	15:8	7:3	2:0
0x68	Message Control		Next Cap Ptr	Capability ID	
0x6C	MSI-X Table Offset				BIR
0x70	Pending Bit Array (PBA) Offset				BIR

Table 4–15 describes the power management capability structure.

Table 4–15. Power Management Capability Structure

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	

Table 4–16 describes the PCI Express capability structure for specification versions 1.0a and 1.1.

Table 4–16. PCI Express Capability Structure Version 1.0a and 1.1 (Note 1)

Byte Offset	31:24	23:16	15:8	7:0
0x080	PCI Express Capabilities Register		Next Cap PTR	Capability ID
0x084	Device capabilities			
0x088	Device Status		Device Control	
0x08C	Link capabilities			
0x090	Link Status		Link Control	
0x094	Slot capabilities			
0x098	Slot Status		Slot Control	
0x09C	Reserved		Root Control	
0x0A0	Root Status			

Note to Table 4–16:

- (1) Reserved and preserved. As per the *PCI Express Base Specification 1.1*, this register is reserved for future RW implementations. Registers are read-only and must return 0 when read. Software must preserve the value read for writes to bits.

Table 4-17 describes the PCI Express capability structure for specification version 2.0.

Table 4-17. PCI Express Capability Structure Version 2.0

Byte Offset	31:16	15:8	7:0
0x080	PCI Express Capabilities Register	Next Cap PTR	PCI Express Cap ID
0x084	Device capabilities		
0x088	Device Status	Device Control	
0x08C	Link capabilities		
0x090	Link Status	Link Control	
0x094	Slot Capabilities		
0x098	Slot Status	Slot Control	
0x09C	Root Capabilities	Root Control	
0x0A0	Root Status		
0x0A4	Device 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 4-17:

- (1) Registers not applicable to a device are reserved.

Table 4-18 describes the virtual channel capability structure.

Table 4-18. Virtual Channel Capability Structure

Byte Offset	31:24	23:16	15:8	7:0
0x100	Next Cap PTR		Vers.	Extended Cap ID
0x104	ReservedP			Port VC Cap 1
0x108	VAT offset	ReservedP		
0x10C	Port VC Status			Port VC control
0x110	PAT offset 0 (31:24)	VC Resource Capability Register (0)		
0x114	VC Resource Control Register (0)			
0x118	VC Resource Status Register (0)			ReservedP
0x11C	PAT offset 1 (31:24)	VC Resource Capability Register (1)		
0x120	VC Resource Control Register (1)			
0x124	VC Resource Status Register (1)			ReservedP
	...			
0x164	PAT offset 7 (31:24)	VC Resource Capability Register (7)		
0x168	VC Resource Control Register (7)			
0x16C	VC Resource Status Register (7)			ReservedP

Table 4–19 describes the PCI Express advanced error reporting extended capability structure.

Table 4–19. PCI Express Advanced Error Reporting Extended Capability Structure

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	

PCI Express Avalon-MM Bridge Control Register 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 space is spread over a 16 KByte region. Each 4 KByte sub-region contains a specific 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 system interconnect fabric — requests from the PCI Express MegaCore function 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 straightforward enforcement by processor software. The four subregions are described Table 4–20:

Table 4–20. 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	Reserved.
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 complete map of PCI Express Avalon-MM bridge registers is shown in [Table 4-21](#):

Table 4-21. PCI Express Avalon-MM Bridge Register Map

Address Range	Register
0x0040	PCI Express Interrupt Status Register
0x0050	PCI Express Interrupt Enable Register
0x0800-0x081F	PCI Express Avalon-MM Bridge Mailbox Registers, Read/Write
0x0900-0x091F	Avalon-MM-to-PCI Express Mailbox Registers, read-only
0x1000-0x1FFF	Avalon-MM-to PCI Express Address Translation Table
0x3060	Avalon-MM Interrupt Status Register
0x3070	Avalon-MM Interrupt Enable Register
0x3A00-0x3A1F	Avalon-MM-to-PCI Express Mailbox Registers, Read/Write
0x3B00-0x3B1F	PCI Express Avalon-MM Bridge Mailbox Registers, read-only

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. These registers can be accessed by other PCI Express root complexes only; however, hardware does not prevent other Avalon-MM masters from accessing them.

[Table 4-22](#) shows the status of all conditions that can cause a PCI Express interrupt to be asserted.

Table 4-22. Avalon-MM to PCI Express Interrupt Status Register

Address: 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
15:14	Reserved	—	—
13:8	AVL_IRQ_INPUT_VECTOR	RO	Avalon-MM interrupt input vector. When the interrupt input <code>RxmIrq_i</code> is asserted, the interrupt vector <code>RxmIrqNum_i</code> is loaded in this register.
7	AVL_IRQ_ASSERTED	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.
6:0	Reserved	—	—

A PCI Express interrupt can be asserted for any of the conditions registered in the PCI Express interrupt status register by setting the corresponding bits in the Avalon-MM-to-PCI Express interrupt enable register (Table 4-23). Either MSI or legacy interrupts can be generated as explained in the section “Generation of PCI Express Interrupts” on page 4-25.

Table 4-23. Avalon-MM to PCI Express Interrupt Enable Register

Address: 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.
[15:8]	Reserved	—	—
[7]	AVL_IRQ	RW	Enables generation of PCI Express interrupts when Rxmirq_i is asserted
[6:0]	Reserved	—	—

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. There are eight mailbox registers available.

The PCI Express-to-Avalon-MM mailbox registers are writable at the addresses shown in Table 4-24. Writing to one of these registers causes the corresponding bit in the Avalon-MM interrupt status register to be set to a one.

Table 4-24. PCI Express-to-Avalon-MM Mailbox Registers, Read/Write

Address Range: 0x800-0x815

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 Table 4-25. The PCI Express root complex should use these addresses to read the mailbox information after being signaled by the corresponding bits in the PCI Express interrupt enable register.

Table 4-25. Avalon-MM-to-PCI Express Mailbox Registers, read-only Address Range: 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 if dynamic translation is enabled.

Each entry in the PCI Express address translation table (Table 4-26) 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 4-26. Avalon-MM-to-PCI Express Address Translation Table Address Range: 0x1000-0x1FFF

Address	Bits	Name	Access	Description
0x1000	[1:0]	A2P_ADDR_SPACE0	RW	Address space indication for entry 0. Refer to Table 4-27 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 4-27 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 number of 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 table entries is greater than 1.

Note to Table 4-26:

- (1) These table entries are repeated for each address specified in the **Number of address pages** parameter (Table 3-6 on page 3-12). 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 [Table 4-27](#).

Table 4-27. 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

PCI Express to Avalon-MM Interrupt Status and Enable Registers

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. These registers can be accessed by a processor local to the system interconnect fabric that processes the Avalon-MM interrupts. These registers must not be accessed by the PCI Express Avalon-MM bridge master ports; however, there is nothing in the hardware that prevents this.

The interrupt status register ([Table 4-28](#)) records the status of all conditions that can cause an Avalon-MM interrupt to be asserted.

Table 4-28. PCI Express to Avalon-MM Interrupt Status Register

Address: 0x3060

Bits	Name	Access	Description
[15:0]	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 interrupt enable register ([Table 4-29](#)).

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. There is typically a single process in either the PCI Express or Avalon-MM domain that is responsible for handling the condition reported by the interrupt.

Table 4-29. PCI Express to Avalon-MM Interrupt Enable Register

Address: 0x3070

Bits	Name	Access	Description
[15:0]	Reserved	—	—
[23:16]	P2A_MB_IRQ	RW	Enables assertion of Avalon-MM interrupt <code>CraIrq_o</code> signal when the specified mailbox is written by the root complex.
[31:24]	Reserved	—	—

Avalon-MM Mailbox Registers

A processor local to the system 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 [Table 4-30](#). When the Avalon-MM processor writes to one of these registers the corresponding bit in the PCI Express interrupt status register is set to 1.

Table 4-30. Avalon-MM-to-PCI Express Mailbox Registers, Read/Write

Address Range: 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 [Table 4-31](#). The Avalon-MM processor reads these registers when the corresponding bit in the Avalon-MM interrupt status register is set to 1.

Table 4-31. PCI Express-to-Avalon-MM Mailbox Registers, Read-Only

Address Range: 0x3800-0x3B1F

Address	Name	Access Mode	Description
0x3B00	P2A_MAILBOX0	R0	PCI Express-to-Avalon-MM mailbox 0.
0x3B04	P2A_MAILBOX1	R0	PCI Express-to-Avalon-MM mailbox 1
0x3B08	P2A_MAILBOX2	R0	PCI Express-to-Avalon-MM mailbox 2
0x3B0C	P2A_MAILBOX3	R0	PCI Express-to-Avalon-MM mailbox 3
0x3B10	P2A_MAILBOX4	R0	PCI Express-to-Avalon-MM mailbox 4
0x3B14	P2A_MAILBOX5	R0	PCI Express-to-Avalon-MM mailbox 5
0x3B18	P2A_MAILBOX6	R0	PCI Express-to-Avalon-MM mailbox 6
0x3B1C	P2A_MAILBOX7	R0	PCI Express-to-Avalon-MM mailbox 7

Active State Power Management (ASPM)

The PCI Express protocol mandates link power conservation, even if a device has not been placed in a low power state by software. ASPM is initiated by software but is subsequently handled by hardware. The MegaCore function automatically shifts to one of two low power states to conserve power:

- **L0s ASPM**—The PCI Express protocol specifies the automatic transition to L0s. In this state, the MegaCore function passes to transmit electrical idle but can maintain an active reception interface because only one component across a link moves to a lower power state. Main power and reference clocks are maintained.



L0s ASPM is not supported when using the Stratix GX internal PHY. It can be optionally enabled when using the Arria GX, Stratix II GX, or Stratix IV GX internal PHY. It is supported for other device families to the extent allowed by the attached external PHY device.

- **L1 ASPM**—Transition to L1 is optional and conserves even more power than L0s. In this state, both sides of a link power down together, so that neither side can send or receive without first transitioning back to L0.



L1 ASPM is not supported when using the Arria GX, Stratix GX, Stratix II GX, or Stratix IV GX internal PHY. It is supported for other device families to the extent allowed by the attached external PHY device.

How quickly a component powers up from a low-power state, and even whether a component has the right to transition to a low power state in the first place, depends on *exit latency* and *acceptable latency*.

Exit Latency

A component's exit latency is defined as the time it takes for the component to awake from a low-power state to L0, and depends on the SERDES PLL synchronization time and the common clock configuration programmed by software. A SERDES generally has one transmit PLL for all lanes and one receive PLL per lane.

- **Transmit PLL**—When transmitting, the transmit PLL must be locked.
- **Receive PLL**—Receive PLLs train on the reference clock. When a lane exits electrical idle, each receive PLL synchronizes on the receive data (clock data recovery operation). If receive data has been generated on the reference clock of the slot, and if each receive PLL trains on the same reference clock, the synchronization time of the receive PLL is lower than if the reference clock is not the same for all slots.

Each component must report in the configuration space if they use the slot's reference clock. Software then programs the common clock register, depending on the reference clock of each component. Software also retrains the link after changing the common clock register value to update each exit latency. [Table 4-32](#) describes the L0s and L1 exit latency. Each component maintains two values for L0s and L1 exit latencies; one for the common clock configuration and the other for the separate clock configuration.

Table 4-32. L0s and L1 Exit Latency

Power State	Description
L0s	<p>L0s exit latency is calculated by the MegaCore function based on the number of fast training sequences specified on the Power Management page of the MegaWizard Plug-In Manager. It is maintained in a configuration space registry. Main power and the reference clock remain present and the PHY should resynchronize quickly for receive data.</p> <p>Resynchronization is performed through fast training order sets, which are sent by the opposite component. A component knows how many sets to send because of the initialization process, at which time the required number of sets is determined through training sequence ordered sets (TS1 and TS2).</p>
L1	<p>L1 exit latency is specified on the Power Management page of the MegaWizard Plug-In Manager. It is maintained in a configuration space registry. Both components across a link must transition to L1 low-power state together. When in L1, a component's PHY is also in P1 low-power state for additional power savings. Main power and the reference clock are still present, but the PHY can shut down all PLLs to save additional power. However, shutting down PLLs causes a longer transition time to L0.</p> <p>L1 exit latency is higher than L0s exit latency. When the transmit PLL is locked, the LTSSM moves to recovery, and back to L0 after both components have correctly negotiated the recovery state. Thus, the exact L1 exit latency depends on the exit latency of each component (the higher value of the two components). All calculations are performed by software; however, each component reports its own L1 exit latency.</p>

Acceptable Latency

The acceptable latency is defined as the maximum latency permitted for a component to transition from a low power state to L0 without compromising system performance. Acceptable latency values depend on a component's internal buffering and are maintained in a configuration space registry. Software compares the link exit latency with the endpoint's acceptable latency to determine whether the component is permitted to use a particular power state.

- For L0s, the opposite component and the exit latency of each component between the root port and endpoint is compared with the endpoint's acceptable latency. For example, for an endpoint connected to a root port, if the root port's L0s exit latency is 1 μ s and the endpoint's L0s acceptable latency is 512 ns, software will probably not enable the entry to L0s for the endpoint.
- For L1, software calculates the L1 exit latency of each link between the endpoint and the root port, and compares the maximum value with the endpoint's acceptable latency. For example, for an endpoint connected to a root port, if the root port's L1 exit latency is 1.5 μ s and the endpoint's L1 exit latency is 4 μ s, and the endpoint acceptable latency is 2 μ s, the exact L1 exit latency of the link is 4 μ s and software will probably not enable the entry to L1.

Some time adjustment may be necessary if one or more switches are located between the endpoint and the root port.



To maximize performance, Altera recommends that you set L0s and L1 acceptable latency values to their minimum values.

Error Handling

Each PCI Express compliant device must implement a basic level of error management and can optionally implement advanced error management. The MegaCore function does both, as described in this section. 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 specifications defines three types of errors, outlined in [Table 4-33](#).

Table 4-33. 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.

Physical Layer

[Table 4-34](#) describes errors detected by the physical layer.

Table 4-34. Errors Detected by the Physical Layer *(Note 1)*

Error	Type	Description
Receive port error	Correctable	<p>This error has the following 3 potential causes:</p> <ul style="list-style-type: none"> Physical coding sublayer error when a lane is in L0 state. The error is reported per lane on <code>rx_status[2:0]</code>: <ul style="list-style-type: none"> 100: 8B10B 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 4-34](#):

(1) Considered optional by the PCI Express specification.

Data Link Layer

[Table 4-35](#) describes errors detected by the data link layer.

Table 4-35. Errors Detected by the Data Link Layer

Error	Type	Description
Bad TLP	Correctable	This error occurs when a link cyclical redundancy check (LCRC) verification fails or when a sequence number error occurs.
Bad DLLP	Correctable	This error occurs when a CRC verification fails.

Table 4-35. Errors Detected by the Data Link Layer

Error	Type	Description
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 AckNak_Seq_Num does not correspond to an unacknowledged TLP.

Transaction Layer

Table 4-36 describes errors detected by the transaction layer.

Table 4-36. Errors Detected by the Transaction Layer (Part 1 of 2)

Error	Type	Description
Poisoned TLP received	Uncorrectable (non-fatal)	<p>This error occurs if a received transaction layer packet has the EP poison bit set.</p> <p>The received TLP is presented on the <code>rx_desc</code> and <code>rx_data</code> buses and the application layer logic must take application appropriate action in response to the poisoned TLP. In PCI Express 1.1, this error is treated as an advisory error.</p>
ECRC check failed (1)	Uncorrectable (non-fatal)	<p>This error is caused by an ECRC check failing despite the fact that the transaction layer packet is not malformed and the LCRC check is valid.</p> <p>The MegaCore function handles this transaction layer packet automatically. If the TLP is a non-posted request, the MegaCore function generates a completion with completer abort status. In all cases the TLP is deleted in the MegaCore function and not presented to the application layer.</p>
Unsupported request	Uncorrectable (non-fatal)	<p>This error occurs whenever a component receives any of the following unsupported requests:</p> <ul style="list-style-type: none"> ■ Completion transaction for which the requester ID does not match the bus/device. ■ Unsupported message. ■ A type 1 configuration request transaction layer packet 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. <p>If the TLP is a non-posted request, the MegaCore function generates a completion with unsupported request status. In all cases the TLP is deleted in the MegaCore function and not presented to the application layer.</p>
Completion timeout	Uncorrectable (non-fatal)	<p>This error occurs when a request originating from the application layer does not generate a corresponding completion transaction layer packet 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 <code>cpl_err[0]</code> signal.</p>

Table 4-36. Errors Detected by the Transaction Layer (Part 2 of 2)

Error	Type	Description
Completer abort (1)	Uncorrectable (non-fatal)	The application layer reports this error using the <code>cpl_err[1]</code> signal when it aborts receipt of a transaction layer packet.
Unexpected completion	Uncorrectable (non-fatal)	<p>This error is caused by an unexpected completion transaction. The MegaCore function handles the following conditions:</p> <ul style="list-style-type: none"> ■ 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. <p>The TLP is not presented to the application layer; the MegaCore function deletes it. The application layer must keep track of invalid completion lengths by keeping track of the request and the length of the completion data.</p>
Receiver overflow (1)	Uncorrectable (fatal)	This error occurs when a component receives a transaction layer packet that violates the FC credits allocated for this type of transaction layer packet. In all cases the MegaCore function 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 within the 200 Ω s limit.
Malformed TLP	Uncorrectable (fatal)	<p>This error is caused by any of the following conditions:</p> <ul style="list-style-type: none"> ■ The data payload of a received transaction layer packet exceeds the maximum payload size. ■ The TD field is asserted but no transaction layer packet digest exists, or a transaction layer packet digest exists but the TD bit of the PCI Express request header packet is not asserted. ■ A transaction layer packet violates a byte enable rule. The MegaCore function checks for this violation, which is considered optional by the PCI Express specifications. ■ A transaction layer packet in which the type and length fields do not correspond with the total length of the transaction layer packet. ■ A transaction layer packet in which the combination of format and type is not specified by the PCI Express specification. ■ A request specifies an address/length combination that causes a memory space access to exceed a 4 KByte boundary. The MegaCore function checks for this violation, which is considered optional by the PCI Express specification. ■ Messages, such as <code>Assert_INTx</code>, power management, error signaling, unlock, and <code>Set_Slot_power_limit</code>, must be transmitted across the default traffic class.
Malformed TLP (cont)	Uncorrectable (fatal)	<ul style="list-style-type: none"> ■ A transaction layer packet that uses an uninitialized virtual channel. <p>The MegaCore function deletes the malformed TLP; it is not presented to the application layer.</p>

Note to Table 4-36:(1) Considered optional by the *PCI Express Base Specification Revision 1.0a, 1.1 or 2.0*.

ECRC

ECRC ensures end-to-end data integrity for systems that require high reliability. You can specify this option on the **Capabilities** page of the MegaWizard Plug-In Manager. The ECRC function includes the ability to check and generate ECRC for all PCI Express MegaCore functions. The hard IP implementation can also forward the TLP with ECRC to the receive port of the application layer. The hard IP implementation transmits a TLP with ECRC from the transmit port of the application layer. When using ECRC forward mode, the ECRC check and generate are done in the application layer.

You must select **Implement advanced error reporting** on the **Capabilities** page of the MegaWizard interface to enable ECRC forwarding, ECRC checking and ECRC generation.

ECRC on the Rx Path

When the ECRC option is turned on, errors are detected when receiving TLPs with bad ECRC. If the ECRC option is turned off, no error detection takes place. If the ECRC forwarding option is turned on, the ECRC value is forwarded to the application layer with the TLP. If ECRC forwarding option is turned off, the ECRC value is not forwarded. [Table 4-37](#) summarizes the Rx ECRC functionality for all possible conditions.

Table 4-37. ECRC Operation on Rx Path

ECRC Forward	ECRC Check Enable ⁽¹⁾	ECRC Status	Error	TLP Forward to Application
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

Notes to Table 4-37:

(1) The **ECRC Check Enable** is in the configuration space advanced error capabilities and control register.

ECRC on the Tx Path

You turn on the **Implement ECRC generation** option on the “**Capabilities Parameters**” on page 3–5. When this option is on, Tx path generates ECRC. If you turn on **Implement ECRC forwarding**, the ECRC value is forwarded with the transaction layer packet. Table 4–38 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 Header Formats**.

Table 4–38. ECRC Generation and Forwarding on Tx Path (Note 1)

ECRC Forward	ECRC Generation Enable (2)	TLP on Application	TLP on Link	Comments
No	No	TD=0, without ECRC	TD=0, without ECRC	ECRC is generated
		TD=1, without ECRC	TD=0, without ECRC	
	Yes	TD=0, without ECRC	TD=1, with ECRC	
		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 4–38:

- (1) All unspecified cases are unsupported and the behavior of the MegaCore function is unknown.
- (2) The ECRC Generation Enable is in the configuration space advanced error capabilities and control register.

Error Logging and Reporting

How the endpoint handles a particular error depends on the configuration registers of the device. Refer to the *PCI Express Base Specification 1.0a, 1.1 or 2.0* for a description of the device signaling and logging for an endpoint.

Data Poisoning

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

- Received poisoned transaction layer packets are sent to the application layer and status bits are automatically updated in the configuration space. In PCI Express 1.1, this is treated as an advisory error.
- Received poisoned configuration write transaction layer packets are not written in the configuration space.
- The configuration space never generates a poisoned transaction layer packet; the error/poisoned bit of the header is always set to 0.

Poisoned transaction layer packets can also set the parity error bits in the PCI configuration space status register. Table 4–39 lists the conditions that cause parity errors.

Table 4-39. Parity Error Conditions

Status Bit	Conditions
Detected parity error (status register bit 15)	Set when any received transaction layer packet is poisoned.
Master data parity error (status register bit 8)	<p>This bit is set when the command register parity enable bit is set and one of the following conditions is true:</p> <ul style="list-style-type: none"> The poisoned bit is set during the transmission of a write request transaction layer packet. The poisoned bit is set on a received completion transaction layer packet.

Poisoned packets received by the MegaCore function are passed to the application layer. Poisoned transmit transaction layer packets are similarly sent to the link.

Supported Message Types

Table 4-40 describes the message types supported by the MegaCore function.

Table 4-40. Supported Message Types (Part 1 of 3) (Note 1)

Message	Root Port	Endpoint	Generated by			Comments
			App Layer	Core	Core (with AL 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 TLPs of type Message Interrupt which triggers the <code>int_status[3:0]</code> signals to the application layer.: ■ <code>int_status[0]</code> : Interrupt signal A ■ <code>int_status[1]</code> : Interrupt signal B ■ <code>int_status[2]</code> : Interrupt signal C ■ <code>int_status[3]</code> : 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 <code>pme_to_cr</code> signal is used to send and acknowledge this message: ■ Root Port: When <code>pme_to_cr</code> asserted, the Root Port sends the <code>PME_turn_off</code> message. ■ Endpoint: This signal is asserted to acknowledge the <code>PME_turn_off</code> message through sending <code>pme_to_ack</code> to the root port.
PME_TO_Ack	Receive	Transmit	No	No	Yes	

Table 4–40. Supported Message Types (Part 2 of 3) (*Note 1*)

Message	Root Port	Endpoint	Generated by			Comments
			App Layer	Core	Core (with AL input)	
Error Signaling Messages						
ERR_COR	Receive	Transmit	No	Yes	No	<p>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:</p> <ul style="list-style-type: none">■ <code>serr_out</code> output signal. When set, indicates to the application layer that an error has been logged in the AER capability structure■ <code>aer_msi_num</code> input signal. When the Implement advanced error reporting option is turned on, you can set <code>aer_msi_num</code> 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	
Hot Plug Messages						
Attention_indicator On	Transmit	Receive	No	Yes	No	<p>As per the recommendations in the <i>PCI Express Base Specification Revision 1.1 or 2.0</i>, these messages are not transmitted to the application layer in the hard IP implementation.</p> <p>For soft IP implementation, following the PCI Express Specification 1.0a, these messages are transmitted to the application layer.</p>
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	

Table 4-40. Supported Message Types (Part 3 of 3) (Note 1)

Message	Root Port	Endpoint	Generated by			Comments
			App Layer	Core	Core (with AL input)	
Attention Button_Pressed(2)	Receive	Transmit	No	No	Yes	

Notes to Table 4-40:

- (1) In the *PCI Express Base Specification Revision 1.1 or 2.0*, this message is no longer mandatory after link training.
- (2) In endpoint mode.

Stratix GX PCI Express Compatibility

During the PCI Express receiver detection sequence, if some other PCI Express devices cannot detect the Stratix GX receiver, the other device remains in the LTSSM detect state. The Stratix GX device remains in the Compliance state, and the link is not initialized because Stratix GX devices do not exhibit the correct receiver impedance characteristics when the receiver input is at electrical idle. Stratix GX devices were designed before the PCI Express specification was developed. Arria GX, Arria II GX, Stratix II GX, and Stratix IV GX devices were designed to meet the PCI Express protocol and do not have this issue. However, Arria GX and Stratix II GX are examples of PCI Express devices that are unable to detect Stratix GX. The resulting design impact is that Stratix GX does not interoperate with some other PCI Express devices.



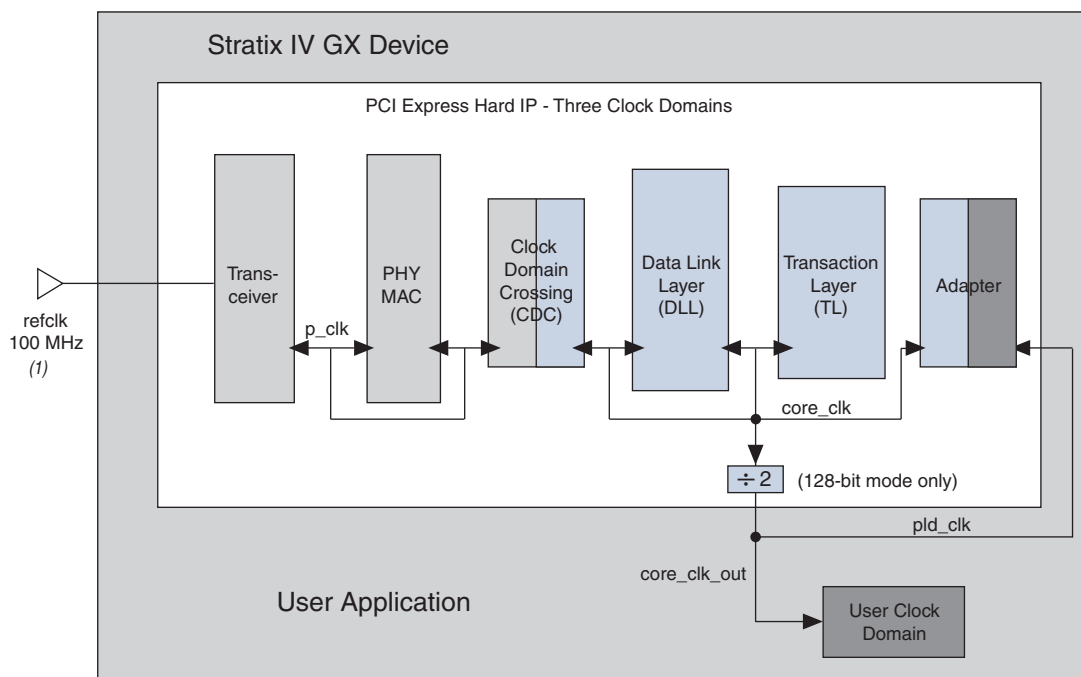
Altera does not recommend using Stratix GX for new designs.

Clocking

Clocking for the PCI Express MegaCore function depends on the choice of design flow. Within the MegaWizard Plug-In Manager design flow, the clock distribution for the hard IP and soft IP implementations is different. The following sections describe the clocking for each MegaCore function variation.

MegaWizard Plug-In Manager Design Flow Clocking—Hard IP Implementation

Figure 4-17 is a block diagram illustrating the clock distribution for this variation.

Figure 4-17. PCI Express MegaCore Function Clock Domains**Notes to Figure 4-17:**

- (1) The 100 MHz `refclk` can only drive the transceiver.
- (2) If the `core_clk_out` frequency is 125 MHz, you can use this clock signal to drive the `cal_blk_clk` signal.

The MegaCore function contains a clock domain crossing (CDC) synchronizer at the interface between the PHYMAC and the DLL layers which allows the data link and transaction layers to run at independent frequencies from the PHY/MAC and provides more flexibility for the user clock interface to the MegaCore function. Depending on system requirements, this additional flexibility can be used to enhance performance by running at a higher frequency for latency optimization or at a lower frequency to save power. As Figure 4-17 indicates, there are three clock domains:

- `p_clk`
- `core_clk`, `core_clk_out`
- `pld_clk`

p_clk

The transceiver derives `p_clk` from the 100 MHz `refclk` signal that you must provide to the Stratix IV GX device. The `p_clk` frequency is 250 MHz for Gen1 systems and 500 MHz for Gen2. The PCI Express specification allows a +/- 300 ppm variation on the clock frequency.

The CDC module implements the asynchronous clock domain crossing between the PHYMAC `p_clk` domain and the data link layer `core_clk` domain.

core_clk, core_clk_out

The `core_clk` signal is derived from `P_CLK`. The `core_clk_out` signal is derived from `core_clk`. Table 4-41 outlines the frequency requirements for `core_clk` and `core_clk_out` to meet PCI Express link bandwidth constraints. An asynchronous FIFO in the adapter decouples the `core_clk` and `p1d_clk` clock domains.

Table 4-41. `core_clk_out` Values for All Parameterizations

Link Width	Max Link Rate	Avalon-ST Width	core_clk	core_clk_out
x1	Gen1	64	125 MHz	125 MHz
x4	Gen1	64	125 MHz	125 MHz
x8	Gen1	64	250 MHz	250 MHz
x8	Gen1	128	250 MHz	125 MHz
x1	Gen2	64	125 MHz	125 MHz
x4	Gen2	64	250 MHz	250 MHz
x4	Gen2	128	250 MHz	125 MHz
x8	Gen2	128	500 MHz	250 MHz

p1d_clk

The application layer and part of the adapter use this clock. Ideally, the `p1d_clk` drives all user logic within the application layer, including other instances of the PCI Express MegaCore function and memory interfaces. The `p1d_clk` input clock pin is typically connected to the `core_clk_out` output clock pin. Alternatively, you can use a separate clock signal to drive the `p1d_clk` input clock pin. If you choose this approach, the clock signal frequency which drives `p1d_clk` must be equal to the `core_clk_out` frequency.

MegaWizard Plug-In Manager Design Flow Clocking—Soft IP Implementation

The soft IP implementation of the PCI Express MegaCore function uses one of several possible clocking configurations, depending on the PHY (external PHY, Arria GX, Arria II GX, Stratix GX, Stratix II GX, or Stratix IV GX) and the reference clock frequency. There are two clock input signals: `refclk` and either `clk125_in` for x1 or x4 variations or `clk250_in` for x8 variations.

The x1 and x4 MegaCore functions also have an output clock, `clk125_out`, that is a 125 MHz transceiver clock. In Stratix GX PHY implementations, `clk125_out` is a 125 MHz version of the transceiver reference clock and must be used to generate `clk125_in`. For external PHY variations `clk125_out` is driven from the `refclk` input. The x8 MegaCore function has an output clock, `clk250_out`, that is the 250 MHz transceiver clock output.

The input clocks are used for the following functions:

- `refclk`—This signal provides the reference clock for the transceiver for Stratix GX PHY implementations. For generic PIPE PHY implementations, `refclk` is driven directly to `clk125_out`.

- `clk125_in`—This signal is the clock for all of the ×1 and ×4 MegaCore function registers, except for a small portion of the receive PCS layer that is clocked by a recovered clock in internal PHY implementations. All synchronous application layer interface signals are synchronous to this clock. `clk125_in` must be 125 MHz and in Stratix GX PHY implementations it must be the exact same frequency as `clk125_out`. In generic PIPE PHY implementations, `clk125_in` must be connected to the `pclk` signal from the PHY.
- `clk250_in` – This signal is the clock for all of the ×8 MegaCore function registers. All synchronous application layer interface signals are synchronous to this clock. `clk250_in` must be 250 MHz and it must be the exact same frequency as `clk250_out`.



Implementing the ×4 MegaCore function in Stratix GX devices uses 4 additional clock resources for the recovered clocks on a per lane basis. The PHY layer elastic buffer uses these clocks.

Generic PIPE PHY Clocking Configuration

When you implement a generic PIPE PHY in the MegaCore function, you must provide a 125 MHz clock on the `clk125_in` input. Typically, the generic PIPE PHY provides the 125 MHz clock across the PIPE interface.

All of the MegaCore function interfaces, including the user application interface and the PIPE interface, are synchronous to the `clk125_in` input. You are not required to use the `refclk` and `clk125_out` signals in this case.

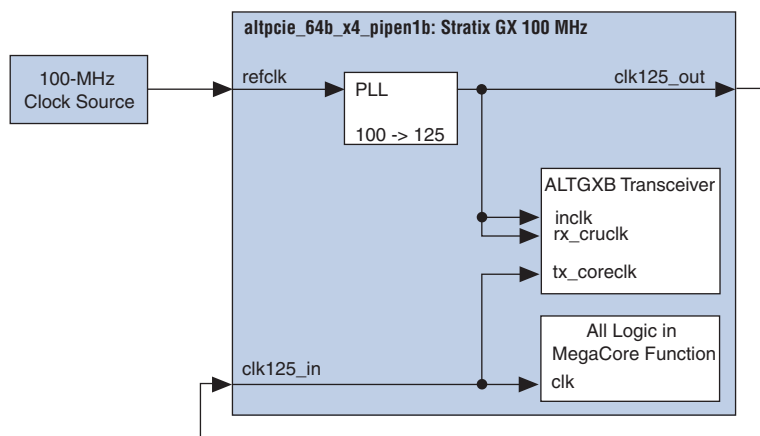
Stratix GX PHY with 100 MHz Reference Clock

If you implement a Stratix GX PHY with a 100 MHz reference clock, you must provide a 100 MHz clock on the `refclk` input. Typically, this clock is the 100 MHz PCI Express reference clock as specified by the Card Electro-Mechanical (CEM) specification.

In this configuration, the 100 MHz `refclk` connects to an enhanced PLL within the MegaCore function to create a 125 MHz clock for use by the Stratix GX transceiver and as the `clk125_out` signal. The 125-MHz clock is provided on the `clk125_out` signal.

You must connect `clk125_out` back to the `clk125_in` input, possibly through a distribution circuit needed in the application. All of the MegaCore function interfaces, including the user application interface and the PIPE interface, are synchronous to the `clk125_in` input. Refer to [Figure 4-18](#).

Figure 4-18. Stratix GX PHY, 100 MHz Reference Clock Configuration (Note 1)



Note to Figure 4-18:

- (1) User and PIPE interface signals are synchronous to `clk125_in`.

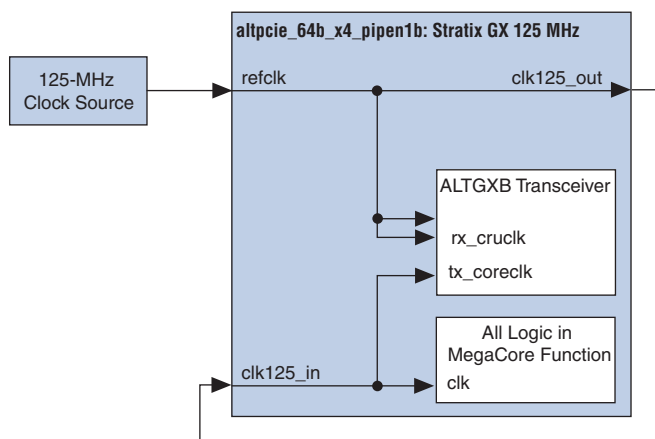
If you want to use other outputs of the enhanced PLL for other purposes or with different phases or frequencies, you should use the 125 MHz reference clock mode and use a 100 to 125 MHz PLL external to the MegaCore function.

Stratix GX PHY with 125 MHz Reference Clock

When implementing the Stratix GX PHY with a 125 MHz reference clock, you must provide a 125 MHz clock on the `refclk` input. The same clock is provided to the `clk125_out` signal with no delay.

You must connect `clk125_out` back to the `clk125_in` input, for example, through a clock distribution circuit needed in the application. All of the MegaCore function's interfaces, including the user application interface and the PIPE interface, are synchronous to the `clk125_in` input. Refer to Figure 4-19.

Figure 4-19. Stratix GX PHY, 125 MHz Reference Clock Configuration (Note 1)



Note to Figure 4-19:

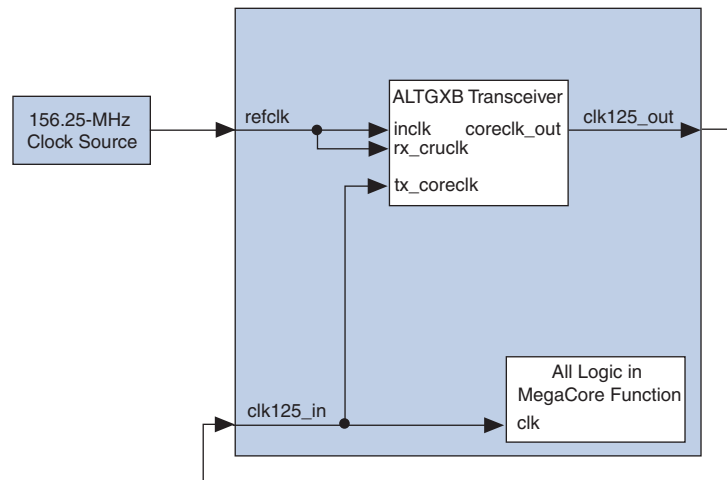
- (1) User and PIPE interface signals are synchronous to `clk125_in`.

Stratix GX PHY with 156.25 MHz Reference Clock

When implementing the Stratix GX PHY with a 156.25 MHz reference clock, you must provide a 156.25 MHz clock on the `refclk` input. The 156.25 MHz clock goes directly to the Stratix GX transceiver. The transceiver's `coreclk_out` output becomes the MegaCore function 125 MHz `clk125_out` output.

You must connect `clk125_out` back to the `clk125_in` input, possibly through a clock distribution circuit needed in the application. All of the MegaCore function interfaces, including the user application interface and the PIPE interface, are synchronous to the `clk125_in` input. Refer to [Figure 4-20](#).

Figure 4-20. Stratix GX PHY with 156.25 MHz Reference Clock Configuration *(Note 1)*



Note to Figure 4-20:

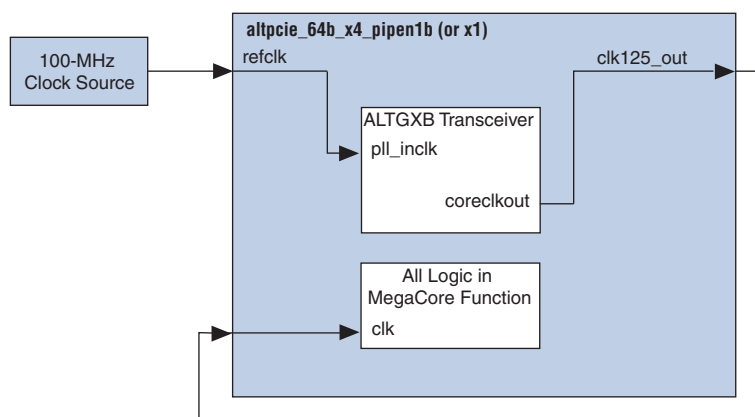
(1) User and PIPE interface signals are synchronous to `clk125_in`.

Arria GX, Stratix II GX, or Stratix IV PHY ×1 and ×4 100 MHz Reference Clock

When implementing the Arria GX, Stratix II GX, or Stratix IV GX PHY in a ×1 or ×4 configuration, the 100 MHz clock is connected directly to the transceiver. The `clk125_out` is driven by the output of the transceiver.

The `clk125_out` must be connected back to the `clk125_in` input, possibly through a clock distribution circuit needed in the specific application. All of the interfaces of the MegaCore function, including the user application interface and the PIPE interface are synchronous to the `clk125_in` input. Refer to [Figure 4-22](#) on page 4-65 for this clocking configuration.

Figure 4-21. Arria GX, Stratix II GX, or Stratix IV PHY $\times 1$ and $\times 4$ with 100 MHz Ref Clock (Note 1)



Note to Figure 4-21:

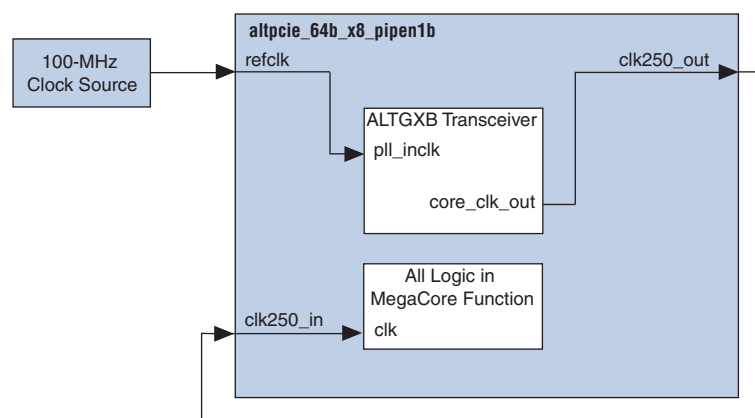
- (1) User and PIPE interface signals are synchronous to `clk125_in`.

Stratix II GX PHY $\times 8$ with 100 MHz Reference Clock

When the Stratix II GX PHY is used in a $\times 8$ configuration, the 100 MHz clock is connected directly to the transceiver. The `clk250_out` is driven by the output of the transceiver.

The `clk250_out` must be connected to the `clk250_in` input, possibly through a clock distribution circuit needed in the specific application. All of the interfaces of the MegaCore function, including the user application interface and the PIPE interface are synchronous to the `clk250_in` input. Refer to Figure 4-22 on page 4-65 for this clocking configuration.

Figure 4-22. Stratix II GX PHY $\times 8$ with 100 MHz Reference Clock 1 on page 4-65



Note to Figure 4-22:

- (1) User and PIPE interface signals are synchronous to `clk250_in`.

SOPC Builder Design Flow Clocking—Soft IP Implementation

When using the PCI Express MegaCore function in the SOPC Builder design flow, the clocking requirements explained in the previous sections remain valid. In addition, the PCI Express MegaCore function supports two clocking modes:

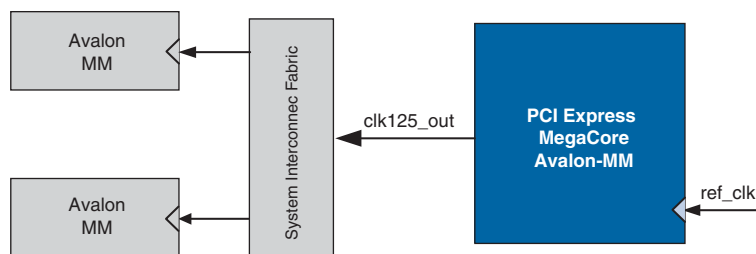
- Separate PCI Express and Avalon clock domains
- Single PCI Express core clock as the system clock for the Avalon-MM clock domain

Separate PCI Express and Avalon Clock Domains

When you turn on the **Use separate clock** option on the **Avalon Configuration** page of the configuration wizard, the system clock source, labeled `ref_clk` in Figure 4-23, is external to the PCI Express MegaCore function. The protocol layers of the MegaCore function are driven by an internal clock that is generated from the reference clock, `ref_clk`. The PCI Express MegaCore function exports a 125 MHz clock, `clk125_out`, which can be used for logic outside the MegaCore function. This clock is not visible to SOPC Builder and therefore cannot drive other Avalon-MM components in the system.

The system interconnect fabric drives the additional input clock, `clk` in Figure 4-23, to the PCI Express MegaCore function. In general, `clk` is the main clock of the SOPC Builder system and originates from an external clock source.

Figure 4-23. SOPC Builder - Separate Clock Domains



Note to Figure 4-23:

- (1) `clk` connects to Avalon-MM global clock, `Av1Clk_L`.

If you turn on the **Use PCIe core clock** option for the Avalon clock domain, you must make appropriate clock assignments for all Avalon-MM components. Figure 4-24 illustrates a system that uses a single clock domain.

Figure 4-24. Connectivity for a PCI Express MegaCore Function with a Single Clock Domain

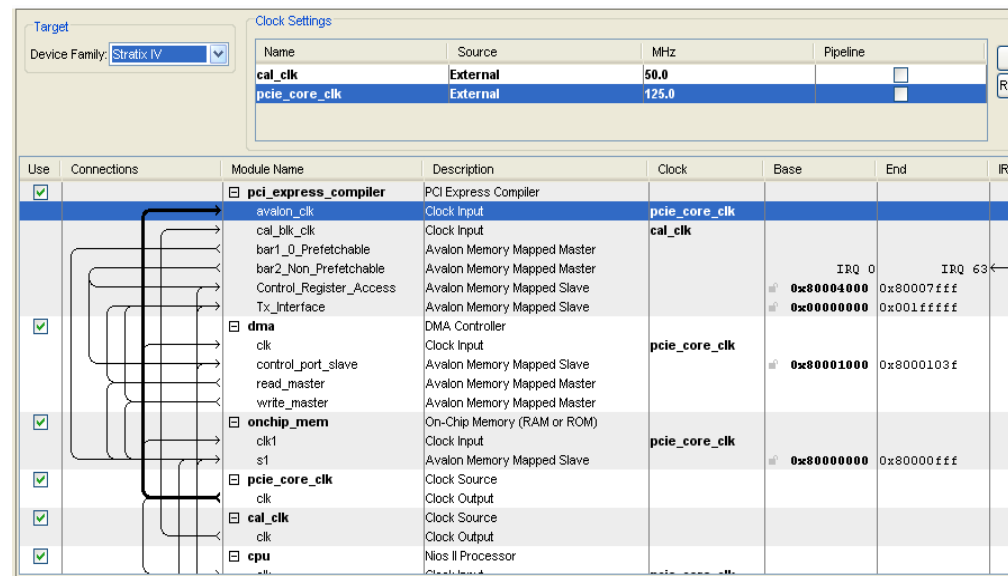


Table 4-42 summarizes the differences between the two Avalon clock modes.

Table 4-42. Selecting the Avalon Clock Domain

Avalon Clock Domain	Description
Use PCIe core clock	In this clocking mode, the PCI Express MegaCore function provides a 125 MHz clock output to be used as a system clock and the MegaCore function protocol layers operate on the same clock. This clock is visible to SOPC Builder and can be selected as the clock source for any Avalon-MM component in the system.
Use separate clock	In this clocking mode, the PCI Express MegaCore function's Avalon-MM logic operates on an external clock source while the MegaCore function protocol layers operate on an internally generated clock.

Multiple Core Configurations

When you instantiate multiple PCI Express MegaCore functions, additional steps are required to create a working design:

- Whether you use the MegaWizard Plug-In Manager or SOPC Builder flow, you must ensure that the `cal_blk_clk` input to each PCI Express MegaCore function is driven by the same calibration clock source.
- When you use SOPC Builder to create a system with multiple PCI Express MegaCore variants, you must click the **Filter** button on the **System Contents** tab to direct SOPC Builder to display the clock connections, rather than using the default view which filters out the clocks.

- After you display clock connections, ensure that `cal_blk_clk` and any other MegaCore function variations in the system that use the transceiver are connected to the `cal_blk_clk` port on the PCI Express MegaCore function variant.
- To ensure that the multiple cores are merged successfully within a single transceiver quadrant, the `gxb_powerdown` signal must be driven with a common signal to all of the MegaCore functions.
- For each variation of PCI Express MegaCore function you should also edit the `<variation_name>_constraints.tcl` and `<variation_name>.sdc` files to change the signal names for constrained signals (clocks and serial or PIPE interface signals) to match the names used in your top level design.
- When using the MegaWizard Plug-In Manager design flow, for each variation of the PCI Express MegaCore function you should also edit the `<variation_name>_constraints.tcl` and `<variation_name>.sdc` files to change the signal names for constrained signals (clocks and serial or PIPE interface signals) to match the names used in your top level design. If you instantiate the same variation twice in your design you must replicate the lines in each of these files for each of the variants. For the SOPC Builder Design flow, each `<variation_name>_constraints.tcl` and `<variation_name>.sdc` file automatically uses the port names at the top level of the SOPC Builder system. You only need to edit these files if you changed the names at a level above the SOPC Builder system.
- You must source the `<variation_name>_constraints.tcl` files for all of the variations that you have used in your design.

This chapter describes the signals that are part of the PCI Express MegaCore function for each of the following primary configurations:

- Signals in the Hard IP Implementation Root Port with Avalon-ST Interface
- Signals in the Hard IP Implementation Endpoint with Avalon-ST Interface
- Signals in the Soft IP Implementation with Avalon-ST Interface
- PCI Express MegaCore Function with Descriptor Data Interface
- PCI Express MegaCore Function with Avalon-MM Interface

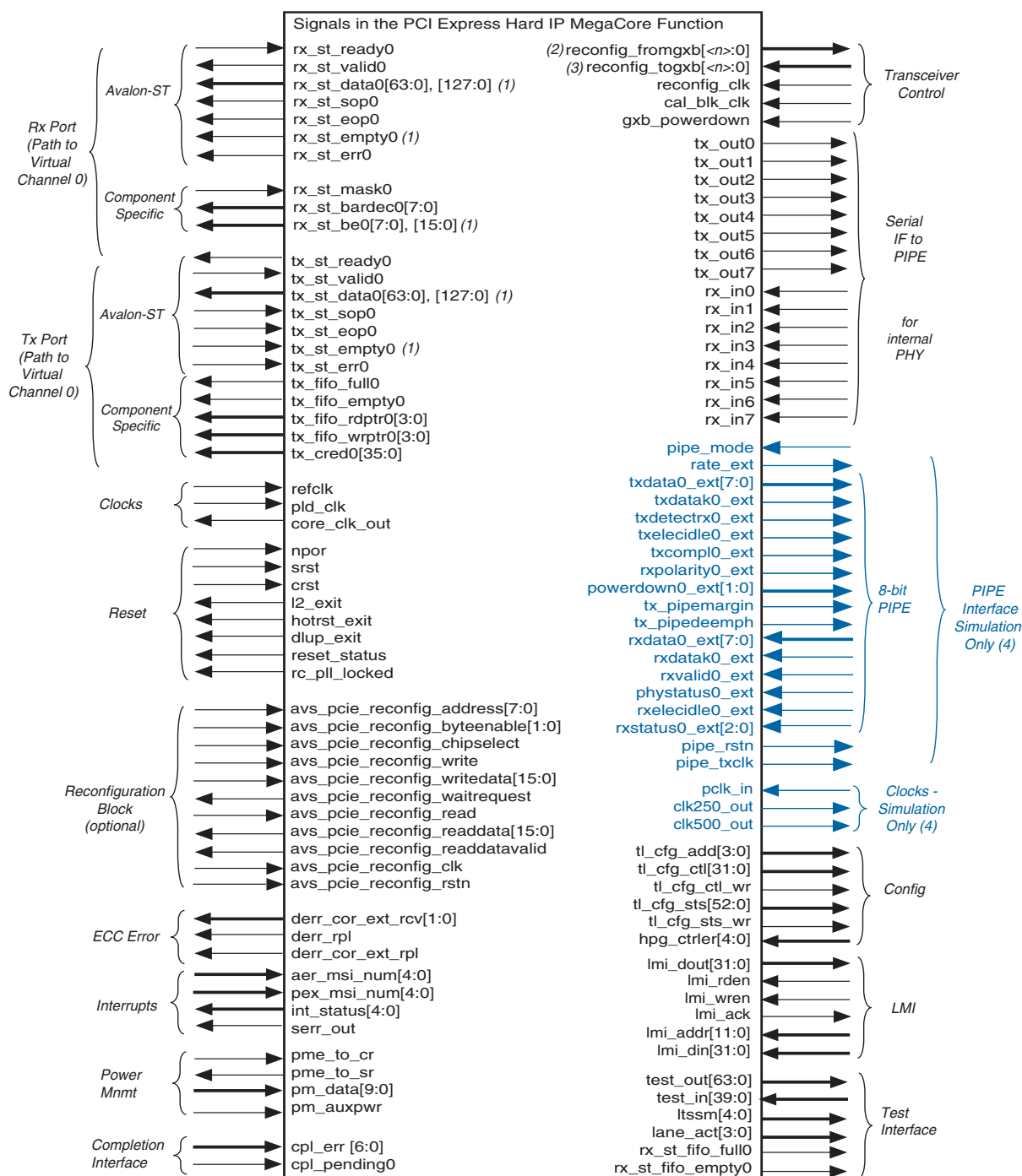
Designs using the Avalon-ST interface in a Stratix IV GX device can take advantage of the hard IP implementation of the MegaCore function which includes a full PCI Express stack with negligible cost in logic elements. You can configure the hard IP implementation as a root port or endpoint. The hard IP implementation is not available for designs using the Avalon-MM interface.



Altera does not recommend the Descriptor/Data interface for new designs.

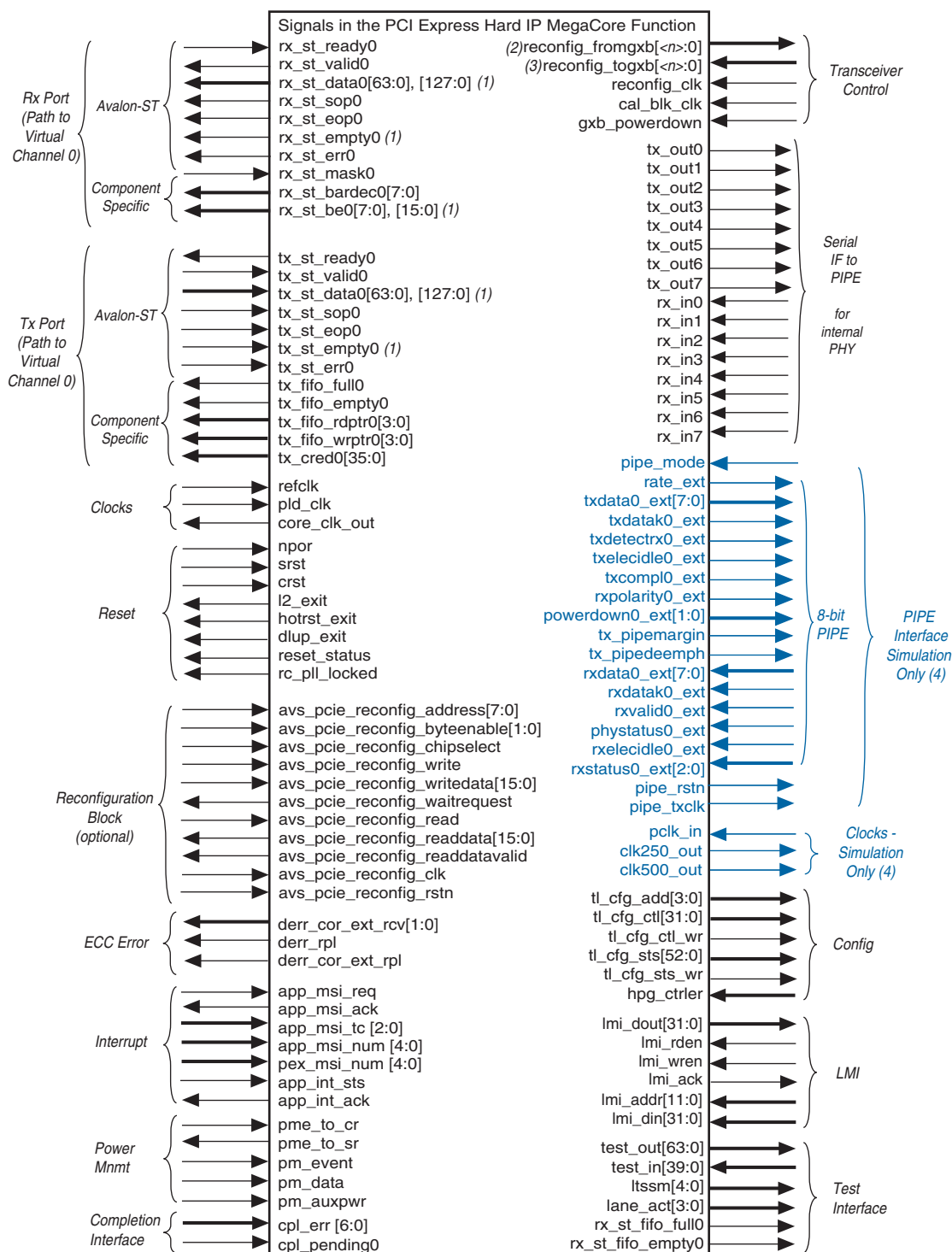
Avalon-ST Interface

The main functional differences between the hard IP and soft IP implementations using an Avalon-ST interface are the configuration and clocking schemes. In addition, the hard IP implementation offers a 128-bit Avalon-ST bus for some configurations. In 128-bit mode, the streaming interface clock, `p1d_clk`, is one-half the frequency of the core clock, `core_clk`, and the streaming data width is 128 bits. In 64-bit mode, the streaming interface clock, `p1d_clk`, is the same frequency as the core clock, `core_clk`, and the streaming data width is 64 bits.

Figure 5–1. Signals in the Hard IP Implementation Root Port with Avalon-ST Interface**Notes to Figure 5–1:**

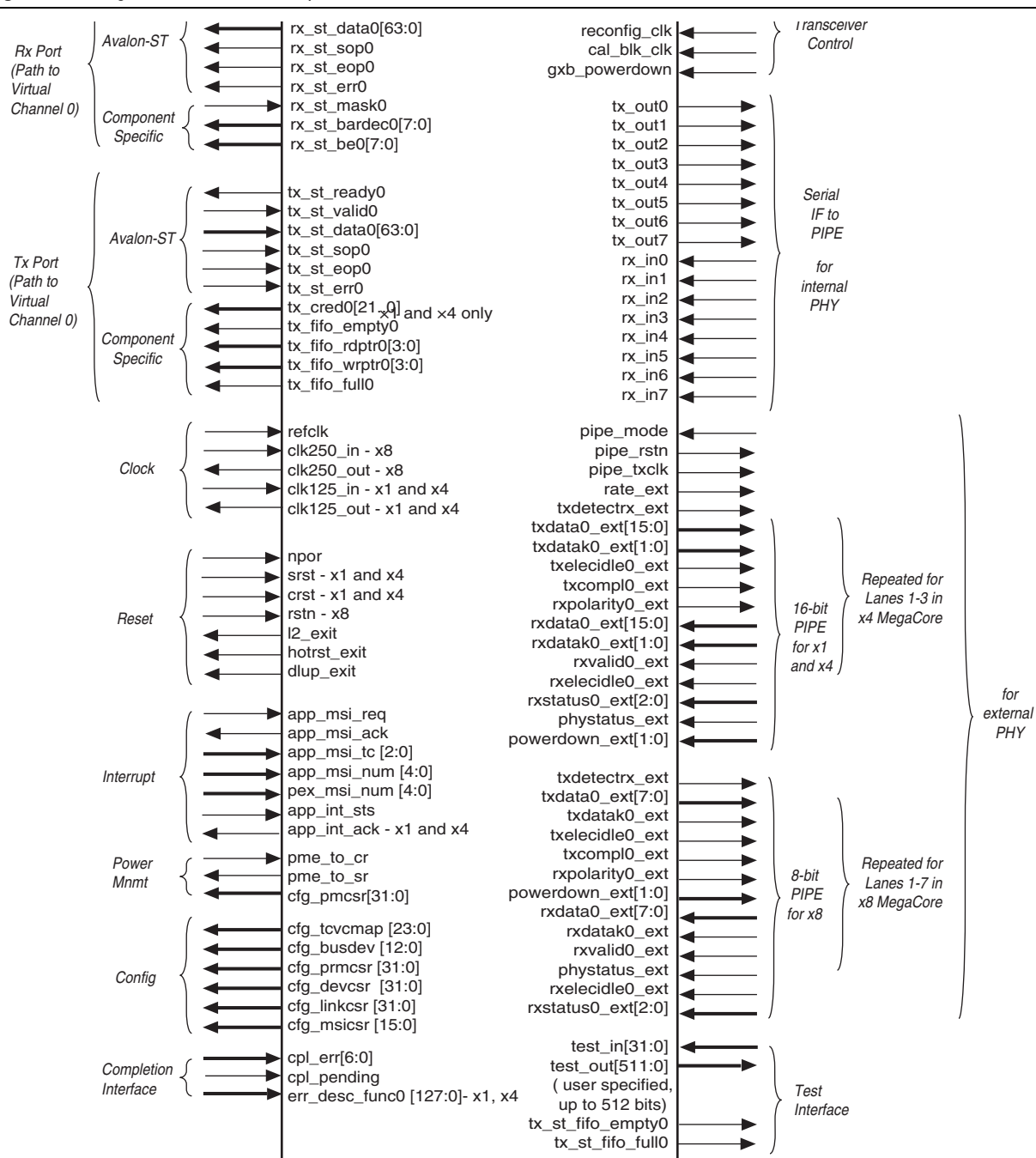
- (1) Applies to 128-bit Avalon-ST only.
- (2) Available in Stratix II GX, Stratix IV GX, Arria GX, and Hardcopy IV GX devices. The `reconfig_fromgxb` is a single wire for Stratix II GX and Arria GX. For Stratix IV GX, $\langle n \rangle = 16$ for $\times 1$ and $\times 4$ MegaCore functions and $\langle n \rangle = 33$ the $\times 8$ MegaCore function.
- (3) Available in Stratix II GX, Stratix IV GX, Arria GX, and Hardcopy IV GX devices. For Stratix II GX and Arria GX `reconfig_togxb`, $\langle n \rangle = 2$. For Stratix IV GX, $\langle n \rangle = 3$.
- (4) Signals in blue are for simulation only.

Figure 5-2. Signals in the Hard IP Implementation Endpoint with Avalon-ST Interface



Notes to Figure 5-2:

- (1) Applies to 128-bit Avalon-ST only.
- (2) Available in Stratix II GX, Stratix IV GX, Arria GX, and Hardcopy IV GX devices. The `reconfig_fromgxb` is a single wire for Stratix II GX and Arria GX. For Stratix IV GX, `<n> = 16` for x1 and x4 MegaCore functions and `<n> = 33` the x8 MegaCore function.
- (3) Available in Stratix II GX, Stratix IV GX, Arria GX, and Hardcopy IV GX devices. For Stratix II GX and Arria GX `reconfig_togxb`, `<n> = 2`. For Stratix IV GX, `<n> = 3`.
- (4) Signals in blue are for simulation only.

Figure 5-3. Signals in the Soft IP Implementation with Avalon-ST Interface**Notes to Figure 5-3:**

- (1) Available in Stratix II GX, Stratix IV GX, Arria GX, and Hardcopy IV GX devices. The `reconfig_fromgxb` is a single wire for Stratix II GX and Arria GX. For Stratix IV GX, $\langle n \rangle = 16$ for x1 and x4 MegaCore functions and $\langle n \rangle = 33$ for the x8 MegaCore function.
- (2) Available in Stratix II GX, Stratix IV GX, Arria GX, and Hardcopy IV GX devices. For Stratix II GX and Arria GX `reconfig_togxb`, $\langle n \rangle = 2$. For Stratix IV GX, $\langle n \rangle = 3$.

Table 5–1 lists the interfaces of both the hard IP and soft IP implementations with links to the subsequent sections that describe each interface.

Table 5–1. Signal Groups in the PCI Express MegaCore Function with Avalon-ST Interface

Signal Group	Hard IP		Soft IP	Description
	End point	Root Port		
Logical				
Avalon-ST Rx	✓	✓	✓	“64- or 128-Bit Avalon-ST Rx Port” on page 5–6
Avalon-ST Tx	✓	✓	✓	“64- or 128-Bit Avalon-ST Tx Port” on page 5–10
Clock	✓	✓	—	“Clock Signals—Hard IP Implementation” on page 5–15
Clock	—	—	✓	“Clock Signals—Soft IP Implementation” on page 5–16
Reset	✓	✓	✓	“Reset Signals” on page 5–16
ECC Error	✓	✓	—	“ECC Error Signals” on page 19
Interrupt	✓	—	✓	“PCI Express Interrupts for Endpoints” on page 5–19
Interrupt and global error	—	✓	—	“PCI Express Interrupts for Root Ports” on page 5–24
Configuration space	✓	✓	—	“Configuration Space Signals—Hard IP Implementation” on page 5–25
Configuration space	—	—	✓	“Configuration Space Signals—Soft IP Implementation” on page 5–28
LMI	✓	✓	—	“LMI Signals—Hard IP Implementation” on page 5–29
PCI Express reconfiguration block	✓	✓	—	“PCI Express Reconfiguration Block Signals—Hard IP Implementation” on page 5–31
Power management	✓	✓	✓	“Power Management Signals” on page 5–31
Completion	✓	✓	✓	“Completion Side Band Signals” on page 5–34
Physical				
Transceiver Control	✓	✓	✓	“Transceiver Control” on page 5–66
Serial	✓	✓	✓	“Serial Interface Signals” on page 5–67
PIPE	(1)	(1)	✓	“PIPE Interface Signals” on page 5–68
Test				
Test	✓	✓		“Test Interface Signals—Hard IP Implementation” on page 5–70
Test	—	—	✓	“Test Interface Signals—Soft IP Implementation” on page 5–72

Note to Table 5–1:

(1) Provided for simulation only

64- or 128-Bit Avalon-ST Rx Port

Table 5–2 describes the signals that comprise the Avalon-ST Rx Datapath.

Table 5–2. 64- or 128-Bit Avalon-ST Rx Datapath

Signal	Width	Dir	Avalon-ST Type	Description
rx_st_ready0	1	1	ready	Indicates that The application is ready to accept data. The application deasserts this signal to throttle the data stream.
rx_st_valid0	1	0	valid	Clocks rx_st_data0 into application. Deasserts within 3 clocks of rx_st_ready0 deassertion and reasserts within 3 clocks of rx_st_ready0 assertion if there is more data to send.
rx_st_data0	64 128	0	data	Receive data bus. Refer to Figure 5–4 through Figure 5–10 for the mapping of the transaction layer's TLP information to rx_st_data. Refer to Figure 5–11 for the timing. Note that the position of the first payload dword depends on whether the TLP address is qword aligned. When using a 64-bit Avalon-ST bus, the width of rx_st_data0 is 64. When using a 128-bit Avalon-ST bus, the width of rx_st_data0 is 128.
rx_st_sop0	1	0	start of packet	Indicates that this is the first cycle of the TLP.
rx_st_eop0	1	0	end of packet	Indicates that this is the last cycle of the TLP.
rx_st_empty0	1	0	empty	Indicates that the TLP ends in the lower 64 bits of rx_st_data. Valid only when rx_st_eop0 is asserted. This signal only applies to 128-bit mode in the hard IP implementation.
rx_st_err0	1	0	error	Indicates that there is an uncorrectable ECC error in the core's internal Rx buffer of the associated VC. When an uncorrectable ECC error is detected, rx_st_err is asserted for at least 1 cycle while rx_st_valid is asserted. If the error occurs before the end of a TLP payload, the packet may be terminated early with an rx_st_eop and with rx_st_valid deasserted on the cycle after the eop. This signal is only active for the hard IP implementations when ECC is enabled.
Component Specific Signals				
rx_st_mask0	1	1	component specific	Application asserts this signal to tell the MegaCore function to stop sending non-posted requests. This signal does not affect non-posted requests that have already been transferred from the transaction layer to the Avalon-ST Adaptor module. This signal can be asserted at any time.
rx_st_bardec0	8	0	component specific	The decoded BAR bits for the TLP. They correspond to the transaction layer's rx_desc[135:128]. They are valid on the 2nd cycle of rx_st_data0.
rx_st_be0	8 16	0	component specific	These are the byte enables corresponding to the transaction layer's rx_be. Valid on any cycle containing payload data. When using 64-bit Avalon-ST bus, the width of rx_st_be is 8. When using 128-bit Avalon-ST bus, the width of rx_st_be is 16.

Table 5-3 shows the byte ordering for header and data packets for Figure 5-4–Figure 5-10. In these figures, the placement of header and data packets is different for aligned and unaligned TLPs.

Table 5-3. 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, pcie_hdr_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<n>, pcie_data_byte<n-1>, pcie_data_byte<n-2>, pcie_data_byte<n-3>

Figure 5-4 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.

Figure 5-4. 64-Bit Avalon-ST rx_st_data0 Cycle Definition for 3-DWord Header TLPs with Non-QWord Aligned Address

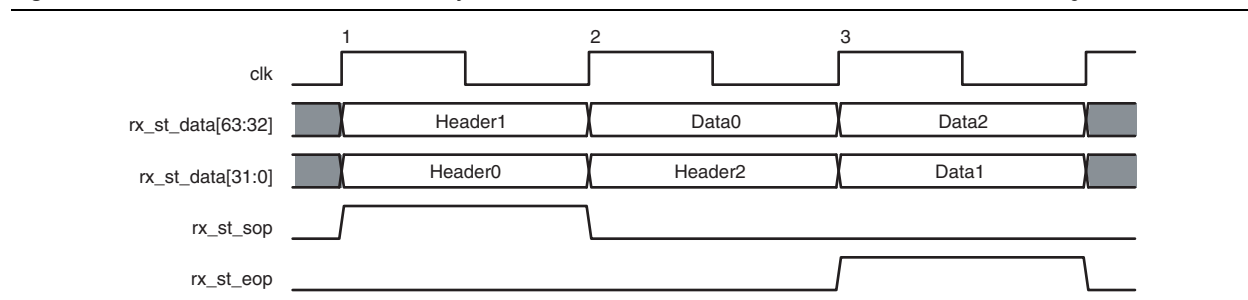


Figure 5-5 shows the mapping of Avalon-ST Rx packets to PCI Express TLPs for TLPs with qword aligned addresses with a 64-bit bus.

Figure 5-5. 64-Bit Avalon-ST rx_st_data0 Cycle Definitions for TLPs with QWord Aligned Addresses

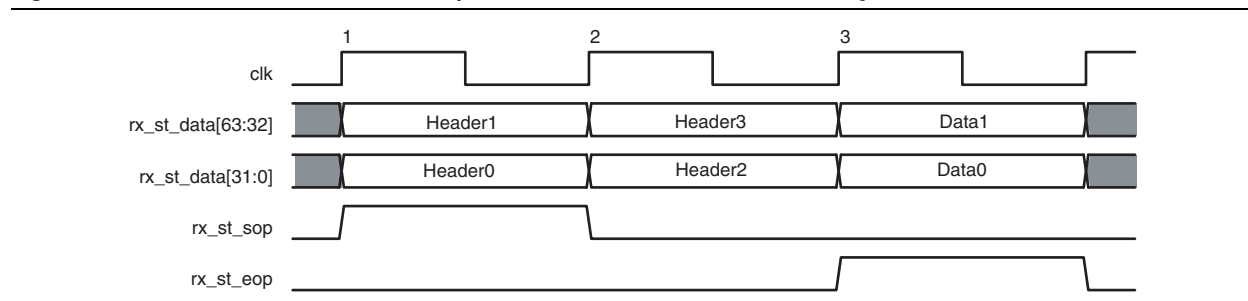


Figure 5-6 shows the mapping of Avalon-ST Rx packet to PCI Express TLPs for TLPs with qword unaligned addresses with a 64-bit bus.

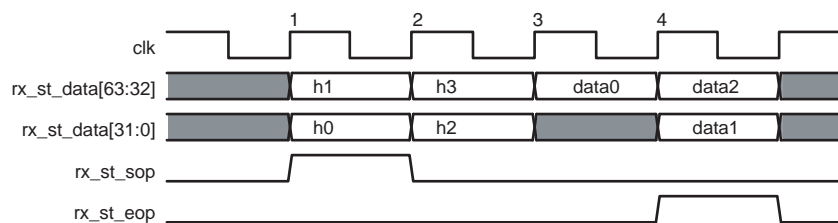
Figure 5-6. 64-Bit Avalon-ST rx_st_data0 Cycle Definitions for 4-DWord Header TLPs with Non-QWord Addresses

Figure 5-7 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.

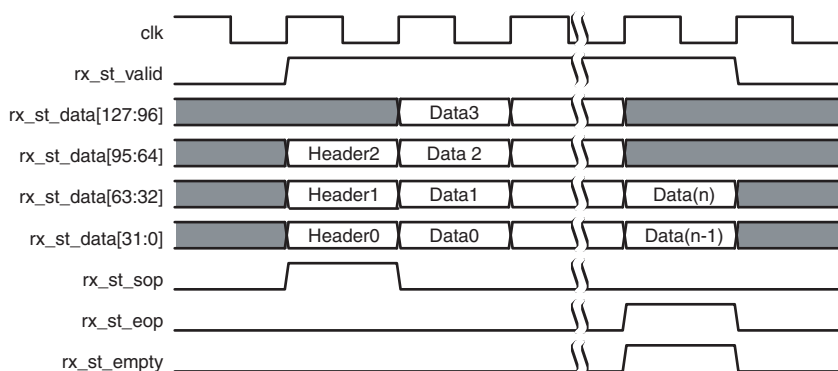
Figure 5-7. 128-Bit Avalon-ST rx_st_data0 Cycle Definition for 3-DWord Header TLPs with QWord Aligned Addresses

Figure 5-8 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.

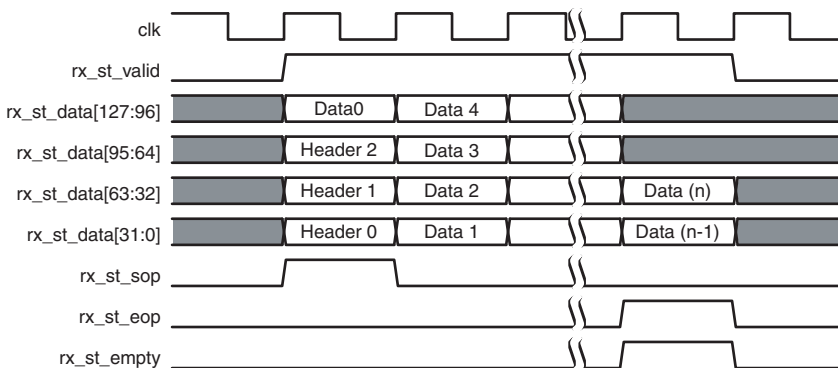
Figure 5-8. 128-Bit Avalon-ST rx_st_data0 Cycle Definition for 3-DWord Header TLPs with non-QWord Aligned Addresses

Figure 5-9 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 ends in the upper 64 bits of rx_st_data.

Figure 5-9. 128-Bit Avalon-ST rx_st_data Cycle Definition for 4-DWord Header TLPs with non-QWord Aligned Addresses

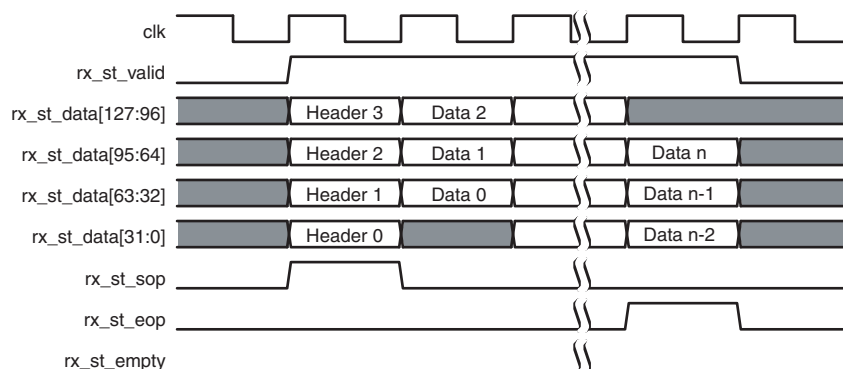
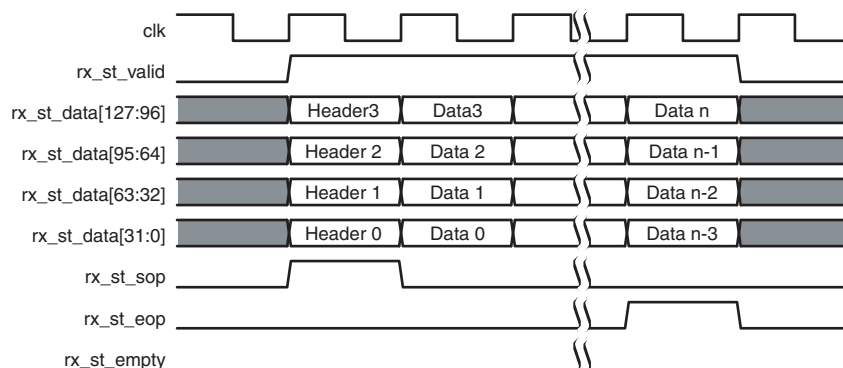


Figure 5-10 shows the mapping of 128-bit Avalon-ST Rx packets to PCI Express TLPs for a four dword header with qword aligned addresses.

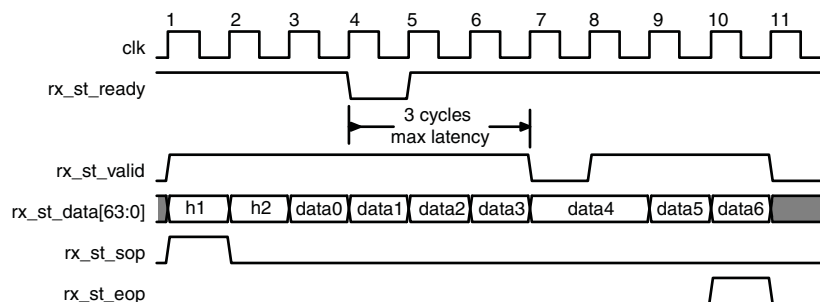
Figure 5-10. 128-Bit Avalon-ST rx_st_data Cycle Definition for 4-DWord Header TLPs with QWord Aligned Addresses



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

Figure 5-11 illustrates the timing of the Avalon-ST Rx interface. On this interface, the core deasserts rx_st_valid in response to the deassertion of rx_st_ready from the application.

Figure 5-11. Avalon-ST Rx Interface Timing



64- or 128-Bit Avalon-ST Tx Port

Table 5–4 describes the signals that comprise the Avalon-ST Tx Datapath.

Table 5–4. 64- or 128-Bit Avalon-ST Tx Datapath (Part 1 of 2)

Signal	Width	Dir	Avalon-ST Type	Description
tx_st_ready0	1	0	ready	Indicates that the PCIe core is ready to accept data for transmission. The core deasserts this signal to throttle the data stream. In the hard IP implementation, tx_st_ready0 may be asserted during reset. The application 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 MegaCore function has come out of reset.
tx_st_valid0	1	1	valid	Clocks tx_st_data0 into the core. Between tx_st_sop0 and tx_st_eop0, must be asserted if tx_st_ready0 is asserted. When tx_st_ready0 deasserts, this signal must deassert within 3 cycles for soft IP implementation and within 2 cycles for hard IP implementation. When tx_st_ready0 reasserts, and tx_st_data0 is in mid-TLP, this signal must reassert within 3 cycles for soft IP and 2 cycles for the hard IP implementation. Refer to Figure 5–19 on page 5–14 for the timing of this signal.
tx_st_data0	64, 128	1	data	Data for transmission. Transmit data bus. Refer to Figure 5–13 through Figure 5–18 for the mapping of TLP packets to tx_st_data0. Refer to Figure 5–19 for the timing of this interface. When using a 64-bit Avalon-ST bus, the width of tx_st_data is 64. When using 128-bit Avalon-ST bus, the width of tx_st_data is 128. The application layer must provide a properly formatted TLP on the Tx interface. 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 unable to accept further requests.
tx_st_sop0	1	1	start of packet	Indicates first cycle of a TLP.
tx_st_eop0	1	1	end of packet	Indicates last cycle of a TLP.
tx_st_empty0	1	1	empty	Indicates that the TLP ends in the lower 64 bits of tx_st_data0. Valid only when tx_st_eop0 is asserted. This signal only applies to 128-bit mode in the hard IP implementation.
tx_st_err0	1	1	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. In the case that a packet is nullified, the following packet should not be transmitted until the next clock cycle. This signal is not available on the x8 Soft IP.

Table 5-4. 64- or 128-Bit Avalon-ST Tx Datapath (Part 2 of 2)

Signal	Width	Dir	Avalon-ST Type	Description
Component Specific Signals				
tx_fifo_full0	1	0	component specific	Indicates that the adapter Tx FIFO is almost full.
tx_fifo_empty0	1	0	component specific	Indicates that the adapter Tx FIFO is empty.
tx_fifo_rdp0[3:0]	4	0	component specific	This is the read pointer for the adaptor Tx FIFO.
tx_fifo_wrp0[3:0]	4	0	component specific	This is the write pointer for the adaptor Tx FIFO.
tx_cred0 (1)	36	0	component specific	<p>This vector contains the available header and data credits for each type of TLP (completion, non-posted, and posted).</p> <p>Non-posted TLP credit fields reflect header and data credits available from the adaptor. These values account for credits pending in the adaptor. The maximum value for the credit fields is 7.</p> <p>Completions and posted TLP credit fields reflect header and data credits available from the transaction layer, and so do not account for credits pending in the adaptor. The maximum value of the header credit fields is 7. Data credit fields are actual counts.</p> <p>Refer to Table 5-5 for the layout of fields in this signal.</p>

Note to Table 5-4:

- (1) For the completion header, posted header, non-posted header, and non-posted data fields, a value of 7 indicates 7 or more available credits.

Table 5-5 illustrates the format of the tx_credit signal.

Table 5-5. Format of the Tx_Credit Signal (Note 1)

35	24	23	21	20	18	17	15	14	3	2	0
Completion Data		Comp Hdr		NPData		NP Hdr		Posted Data		Posted Header	

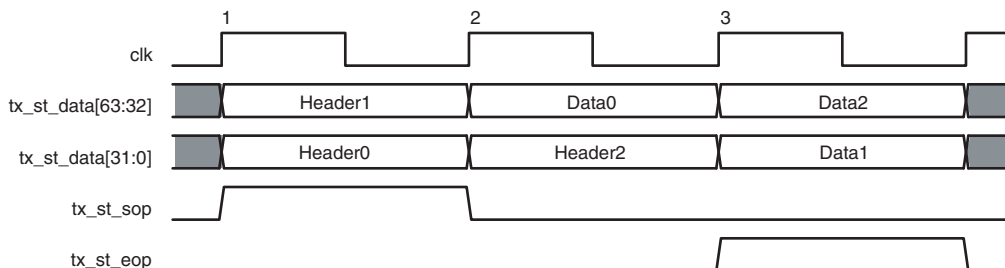
Note to Table 5-5:

- (1) For the completion header, posted header, non-posted header, and non-posted data fields, a value of 7 indicates 7 or more available credits.

Mapping of Avalon-ST Packets to PCI Express

Figure 5-12 illustrates the mapping between Avalon-ST Tx packets and PCI Express TLPs for 3 dword header TLPs with non-qword aligned addresses with a 64-bit bus.

Figure 5-12. 64-Bit Avalon-ST tx_st_data Cycle Definition for 3-DWord Header TLPs with Non-QWord Aligned Addresses

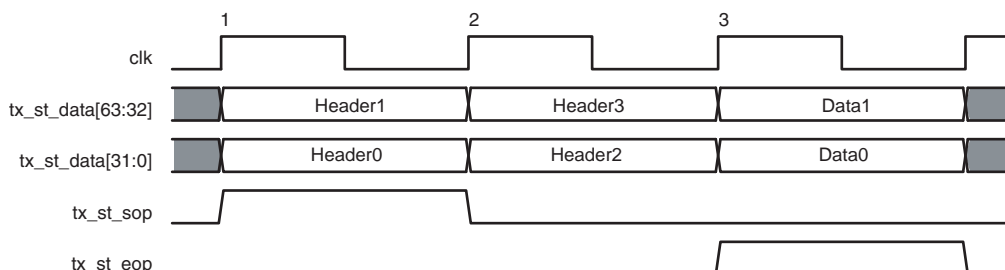


Notes to Figure 5-12:

- (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) 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 5-13 illustrates the mapping between Avalon-ST Tx packets and PCI Express TLPs for a four dword header with qword aligned addresses with a 64-bit bus.

Figure 5-13. 64-Bit Avalon-ST tx_st_data Cycle Definition for TLPs with QWord Aligned Addresses



Notes to Figure 5-13:

- (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, pcie_hdr_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 5-14 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 5-14. 64-Bit Avalon-ST tx_st_data Cycle Definition for TLPs 4-DWord Header with Non-QWord Aligned Addresses

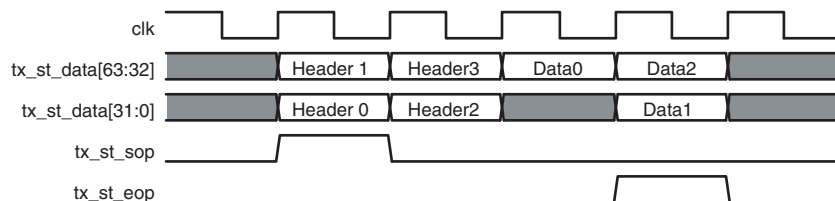


Figure 5-15 shows the mapping of 128-bit Avalon-ST Tx packets to PCI Express TLPs for a three dword header with qword aligned addresses.

Figure 5-15. 128-Bit Avalon-ST tx_st_data Cycle Definition for 3-DWord Header TLPs with QWord Aligned Addresses

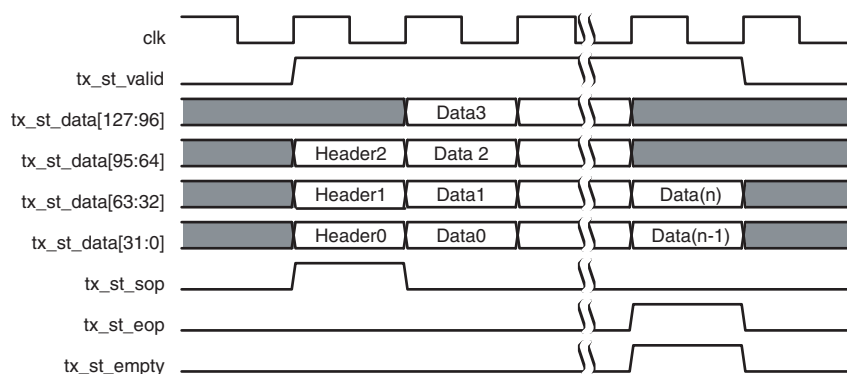


Figure 5-16 shows the mapping of 128-bit Avalon-ST Tx packets to PCI Express TLPs for a 3 dword header with non-qword aligned addresses.

Figure 5-16. 128-Bit Avalon-ST tx_st_data Cycle Definition for 3-DWord Header TLPs with non-QWord Aligned Addresses

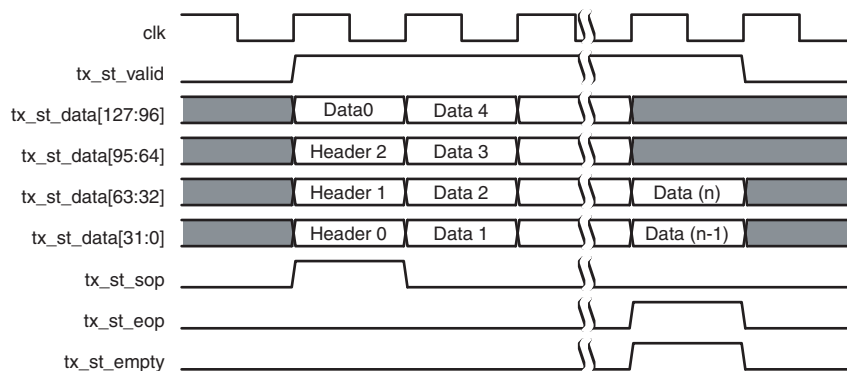


Figure 5-17 shows the mapping of 128-bit Avalon-ST Tx packets to PCI Express TLPs for a four dword header TLP with qword aligned data.

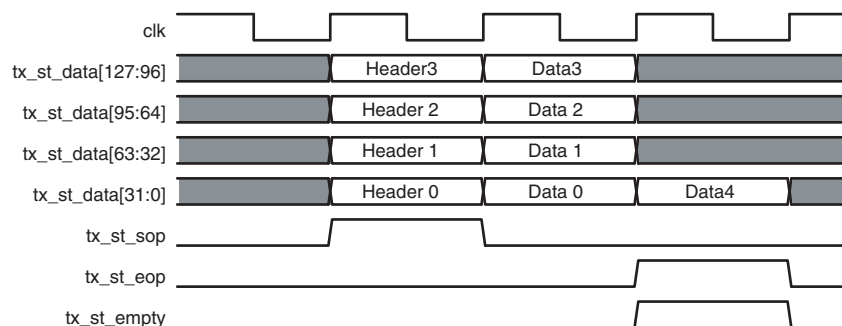
Figure 5-17. 128-Bit Avalon-ST tx_st_data Cycle Definition for 4-DWord Header TLPs with QWord Aligned Addresses

Figure 5-18 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.

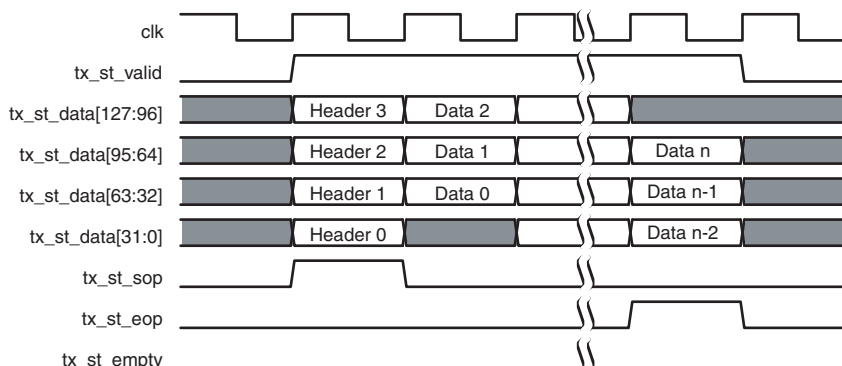
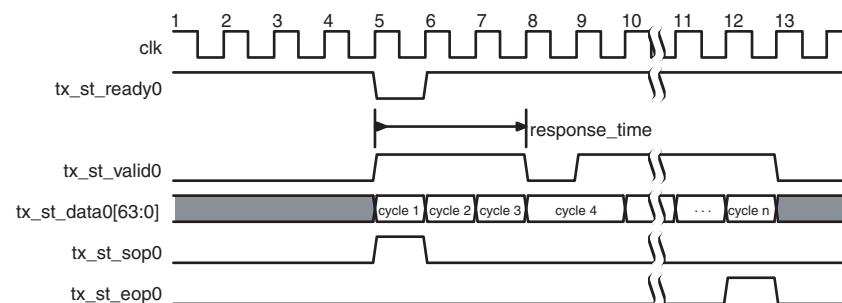
Figure 5-18. 128-Bit Avalon-ST tx_st_data Cycle Definition for 4-DWord Header TLPs with non-QWord Aligned Addresses

Figure 5-19 illustrates the timing of the Avalon-ST Tx interface. The core can deassert tx_st_ready0 to throttle the application which is the source.

Figure 5-19. Avalon-ST Tx Interface Timing**Notes to Figure 5-19:**

- (1) The maximum allowed response time is 3 clock cycles for the soft IP implementation and 2 clock cycles for the hard IP implementation.

Root Port Mode Configuration Requests

To ensure proper operation when sending CFG0 transactions in root port mode, the application should wait for the CFG0 to be transferred to the MegaCore Function's configuration space before issuing another packet on the Avalon-ST Tx port. You can do this by waiting at least 10 clocks from the time the CFG0 SOP is issued on Avalon-ST and then checking for `tx_fifo_empty0==1` before sending the next packet.

If your application implements ECRC forwarding, it should not apply ECRC forwarding to CFG0 packets that it issues on Avalon-ST. 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 internally consumed by the MegaCore function 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 5-5 on page 5-7](#) through [Figure 5-10 on page 5-9](#) illustrate the position of the ECRC data for Rx data. [Figure 5-12 on page 5-12](#) through [Figure 5-18 on page 5-14](#) illustrate the position of ECRC data for Tx data. For packets with no payload data, the ECRC would correspond to Data0 in these figures.

Clock Signals—Hard IP Implementation

[Table 5-6](#) describes the clock signals that comprise the clock interface used in the hard IP implementation.

Table 5-6. Clock Signals Hard IP Implementation

Signal	I/O	Description
refclk	I	Reference clock for the MegaCore function. It must be the frequency specified on the System Settings page accessible from the Parameter Settings tab in the MegaWizard interface.
pld_clk	I	Clocks the application layer and part of the adapter. You must drive this clock from <code>core_clk_out</code> .
core_clk_out	O	This is a fixed frequency clock used by the data link and transaction layers. To meet PCI Express link bandwidth constraints, it has minimum frequency requirements which are outlined in Table 4-41 .
p_clk	I	This is used for simulation only, and is derived from the <code>refclk</code> . It is the PIPE interface clock used for PIPE mode simulation.
clk250_out	O	This is used for simulation only. The testbench uses this to generate <code>p_clk</code> .
clk500_out	O	This is used for simulation only. The testbench uses this to generate <code>p_clk</code> .

Refer to [“Clocking” on page 4-59](#) for a complete description of the clock interface for each PCI Express MegaCore function.

Clock Signals—Soft IP Implementation

Table 5–7. Clock Signals Soft IP Implementation

Signal	I/O	Description
refclk	I	Reference clock for the MegaCore function. It must be the frequency specified on the System Settings page accessible from the Parameter Settings tab in the MegaWizard interface.
clk125_in	I	Input clock for the ×1 and ×4 MegaCore function. All of the MegaCore function I/O signals (except refclk, clk125_out, and npor) are synchronous to this clock signal. This signal must be connected to the clk125_out signal.
clk125_out	O	Output clock for the ×1 and ×4 MegaCore function. 125-MHz clock output derived from the refclk input. This signal is not on the ×8 MegaCore function.
clk250_in	I	Input clock for the ×8 MegaCore function. All of the MegaCore function I/O signals (except refclk, clk250_out, and npor) are synchronous to this clock signal. This signal must be connected to the clk250_out signal.
clk250_out	O	Output from the ×8 MegaCore function. 250 MHz clock output derived from the refclk input. This signal is only on the ×8 MegaCore function.

Reset Signals

Table 5–8 describes the reset signals available in configurations using the Avalon-ST interface or descriptor/data interface.

Table 5–8. Reset Signals (Part 1 of 2)

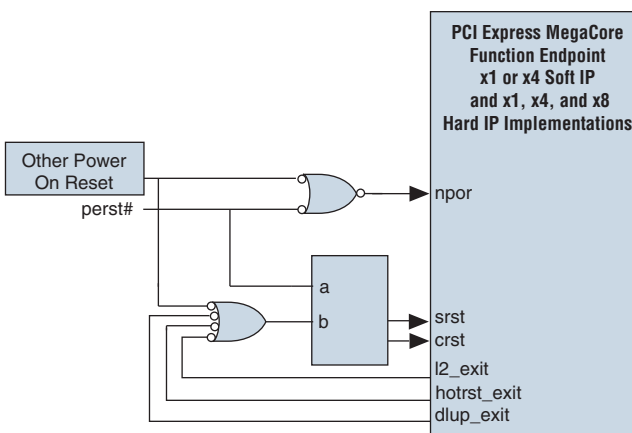
Signal	I/O	Description
rstn	I	Asynchronous reset of configuration space and datapath logic. Active Low. This signal is only available on the ×8 MegaCore function. Used in ×8 soft IP implementation only.
npor	I	Power on reset. This signal is the asynchronous active-low power-on reset signal. This reset signal is used to initialize all configuration space sticky registers, PLL, and SERDES circuitry. In Stratix GX 100 or 156.25 MHz reference clock implementations, clk125_out is held low while npor is asserted.
srst	I	Synchronous datapath reset. This signal is the synchronous reset of the datapath state machines of the MegaCore function. It is active high. This signal is only available on the hard IP and soft IP ×1 and ×4 implementations.
crst	I	Synchronous configuration reset. This signal is the synchronous reset of the nonsticky configuration space registers. It is active high. This signal is only available on the hard IP, and ×1 and ×4 soft IP implementations.
l2_exit	O	L2 exit. The PCI Express specification defines fundamental hot, warm, and cold reset states. A cold reset (assertion of crst and srst for the hard IP implementation and the ×1 and ×4 soft IP implementation, or rstn for ×8 soft IP implementation) must be performed when the LTSSM exits L2 state (signaled by assertion of this signal). This signal is active low and otherwise remains high.
hotrst_exit	O	Hot reset exit. This signal is asserted for 1 clock cycle when the LTSSM exits the hot reset state. It informs the application layer that it is necessary to assert a global reset (crst and srst for the hard IP implementation and the ×1 and ×4 soft IP implementation, or rstn for ×8 soft IP implementation). This signal is active low and otherwise remains high.
dlup_exit	O	This signal is active for one pld_clk cycle when the MegaCore function exits the DLCSM DLUP state. In endpoints, this signal should cause the application to assert a global reset (crst and srst in the hard IP implementation and ×1 and ×4 soft IP implementation, or rstn in ×8 the soft IP implementation). In root ports, this signal should cause the application to assert srst, but not crst. This signal is active low and otherwise remains high.

Table 5-8. Reset Signals (Part 2 of 2)

Signal	I/O	Description
reset_status	0	Reset Status signal. When asserted, this signal indicates that the MegaCore function is in reset. This signal is only available in the hard IP implementation.
rc_pll_locked	0	Indicates that the SERDES receiver PLL is in locked mode with the reference clock. In pipe simulation mode this signal is always asserted.

Figure 5-20 shows the MegaCore function's global reset signals for $\times 1$ and $\times 4$ endpoints in the soft IP implementation and $\times 1$, $\times 4$, and $\times 8$ endpoints in the hard IP implementations

Figure 5-20. Global Reset Signals for $\times 1$ and $\times 4$ Endpoints in the Soft IP and 1, $\times 4$, and $\times 8$ endpoints in the Hard IP Implementations



Notes to Figure 5-20:

- (1) Input a asserts reset for 1,024 cycles.
- (2) Input b asserts reset for 16 cycles

Figure 5-21 shows the MegaCore function's global reset signals for the root port which is available in the hard IP implementation.

Figure 5-21. Global Reset Signals for Root Port Hard IP Implementation

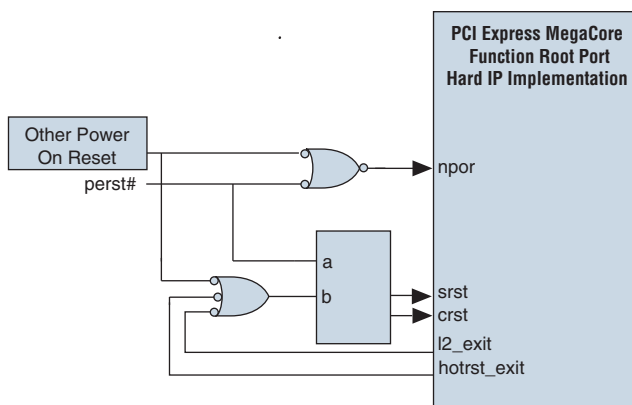


Figure 5–21. Global Reset Signals for Root Port Hard IP Implementation**Notes to Figure 5–21:**

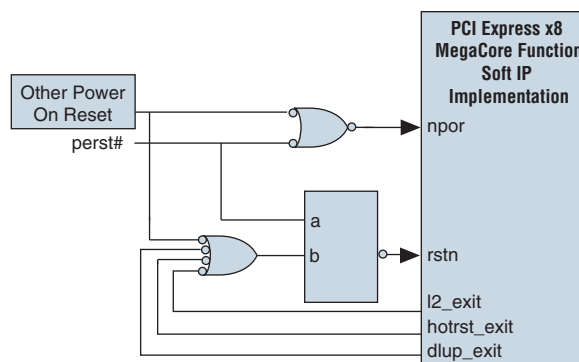
- (1) Input a asserts reset for 1,024 cycles.
- (2) Input b asserts reset for 16 cycles

The hard IP implementation ($\times 1$, $\times 4$ and $\times 8$) or the soft IP implementation ($\times 1$ and $\times 4$) have three reset inputs: `npor`, `srst`, and `crst`. `npor` is used internally for all sticky registers (registers that may not be reset in L2 low power mode or by the fundamental reset). `npor` is typically generated by a logical OR of the power-on-reset generator and the `perst` signal as specified in the PCI Express card electromechanical specification.

The `srst` signal is a synchronous reset of the datapath state machines. The `crst` signal is a synchronous reset of the nonsticky configuration space registers. For endpoints, whenever the `l2_exit`, `hotrst_exit`, `dlup_exit`, or other power-on-reset signals are asserted, `srst` and `crst` should be asserted for one or more cycles for the soft IP implementation and for at least 2 clock cycles for hard IP implementation. For root ports, `srst` should be asserted whenever `l2_exit`, `hotrst_exit`, `dlup_exit`, and power-on-reset signals are asserted. The root port `crst` signal should be asserted whenever `l2_exit`, `hotrst_exit` and other power-on-reset signals are asserted. When the `perst#` signal is asserted, `srst` and `crst` should be asserted for a longer period of time to ensure that the root complex is stable and ready for link training.

The reset block shown in Figure 5–22 is provided as clear text and is not included as part of the MegaCore function to provide some flexibility for implementation-specific methods of generating a reset.

Figure 5–22 shows the $\times 8$ soft IP MegaCore function's global reset signals.

Figure 5–22. Global Reset Signals for $\times 8$ Endpoint Soft IP Implementation**Notes to Figure 5–22:**

- (1) Input a asserts reset for 1,024 cycles.
- (2) Input b asserts reset for 16 cycles.

The PCIe soft IP implementation ($\times 8$) has two reset inputs, `npor` and `rstn`. The `npor` reset is used internally for all sticky registers that may not be reset in L2 low power mode or by the fundamental reset. `npor` is typically generated by a logical OR of the power-on-reset generator and the `perst#` signal as specified in the *PCI Express Card electromechanical Specification*.

The `rstn` signal is an asynchronous reset of the datapath state machines and the nonsticky configuration space registers.

Whenever the `l2_exit`, `hotrst_exit`, `dlup_exit`, or other power-on-reset signals are asserted, `rstn` should be asserted for one or more cycles. When the `perst#` signal is asserted, `rstn` should be asserted for a longer period of time to ensure that the root complex is stable and ready for link training.

The reset block shown in [Figure 5-22](#) is not included as part of the MegaCore function to provide some flexibility for implementation-specific methods of generating a reset.

ECC Error Signals

Table 5-9. ECC Error Signals for Hard IP Implementation

Signal	I/O	Description
<code>derr_cor_ext_rcv[1:0]</code> (1)	0	Indicates a correctable error in the Rx buffer for the corresponding virtual channel.
<code>derr_rpl</code> (1)	0	Indicates an uncorrectable error in the retry buffer.
<code>derr_cor_ext_rpl</code> (1)	0	Indicates a correctable error in the retry buffer.
Note to Table 5-9: (1) This signal applies only when ECC is enabled in some hard IP configurations. Refer to Table 1-20 on page 1-19 for more information.		

PCI Express Interrupts for Endpoints

The PCI Express Compiler provides support for PCI Express legacy interrupts, MSI and MSI-X interrupts when configured in endpoint. See section 6.1 of [PCI Express 2.0 Base Specification](#) for a general description of PCI Express interrupt support for endpoints. MSI-X interrupts are only available in the hard IP implementation endpoint variations. The other interrupts and corresponding signals are available in configurations using the Avalon-ST interface (hard IP or soft IP implementations) or the descriptor/data interface.

Legacy interrupts are signaled on the PCI Express link using message TLPs that are generated internally by the PCI Express MegaCore function. 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. Refer to [Figure 5-27](#) and [Figure 5-28](#).

MSI interrupts are signaled on the PCI Express link using posted write TLPs generated internally by the PCI Express MegaCore function. 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. Refer to [Figure 5-28](#).

The MSI, MSI-X, and legacy interrupts are mutually exclusive. After power up, the MegaCore function starts in INTx mode, after which time software decides whether to switch to MSI mode by programming the `msi_enable` bit of the MSI control register to 1 or to MSI-X mode if you turn on **Implement MSI-X** on the **Capabilities** page of the MegaWizard interface. 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. MSI-X TLPs are generated by the application and sent through the transmit interface. 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. In the hard IP implementation, 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*.

Table 5–10 describes the MegaCore function's interrupt signals for endpoints.

Table 5–10. Interrupt Signals for Endpoints

Signal	I/O	Description
<code>app_msi_req</code>	I	Application MSI request. Assertion causes an MSI posted write TLP to be generated based on the MSI configuration register values and the <code>app_msi_tc</code> and <code>app_msi_num</code> input ports.
<code>app_msi_ack</code>	O	Application MSI acknowledge. This signal is sent by the MegaCore function to acknowledge the application's request for an MSI interrupt.
<code>app_msi_tc[2:0]</code>	I	Application 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).
<code>app_msi_num[4:0]</code>	I	Application MSI offset number. This signal is used by the application to indicate the offset between the base message data and the MSI to send.
<code>cfg_msicsr[15:0]</code>	O	Configuration MSI control status register. This bus provides MSI software control. Refer to Table 5–11 and Table 5–12 for more information.
<code>pex_msi_num[4:0]</code>	I	Power management MSI number. This signal is used by power management and/or hot plug to determine the offset between the base message interrupt number and the message interrupt number to send through MSI.
<code>app_int_sts</code>	I	Controls legacy interrupts. Assertion of <code>app_int_sts</code> causes an Assert_INTA message TLP to be generated and sent upstream. Deassertion of <code>app_int_sts</code> causes a Deassert_INTA message TLP to be generated and sent upstream.
<code>app_int_ack</code>	O	This signal is the acknowledge for <code>app_int_sts</code> . 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 <code>app_int_sts</code> signal or when the Deassert_INTA message TLP has been transmitted in response to the deassertion of the <code>app_int_sts</code> signal. It is only included on the Avalon-ST interface for the hard IP implementation and the x1 and x4 soft IP implementation. Refer to Figure 5–27 for timing information.

Table 5–11 shows the layout of the Configuration MSI Control Status Register.

Table 5–11. Configuration MSI Control Status Register

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

Table 5–12 outlines the use of the various fields of the Configuration MSI Control Status Register.

Table 5–12. Configuration MSI Control Status Register Field Descriptions

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 <code>Mask_Bits</code> and <code>Pending_Bits</code> registers defined in the <i>PCI Local Bus Specification, Rev. 3.0</i> . Per vector masking can be implemented using application layer registers as shown in Figure 5–24.
[7]	64-bit address capability	64-bit address capable <ul style="list-style-type: none"> 1: function capable of sending a 64-bit message address 0: function not capable of sending a 64-bit message address
[6:4]	multiple message enable	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 <ul style="list-style-type: none"> 000: 1 MSI allocated 001: 2 MSI allocated 010: 4 MSI allocated 011: 8 MSI allocated 100: 16 MSI allocated 101: 32 MSI allocated 110: Reserved 111: Reserved
[3:1]	multiple message capable	Multiple message capable: This field is read by system software to determine the number of requested MSI messages. <ul style="list-style-type: none"> 000: 1 MSI requested 001: 2 MSI requested 010: 4 MSI requested 011: 8 MSI requested 100: 16 MSI requested 101: 32 MSI requested 110: Reserved
[0]	MSI Enable	If set to 0, this component is not permitted to use MSI.

Figure 5–23 illustrates the architecture of the MSI handler block.

Figure 5–23. MSI Handler Block

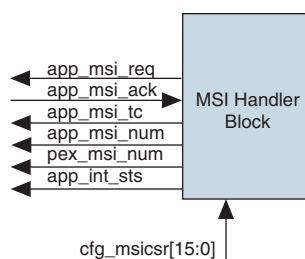
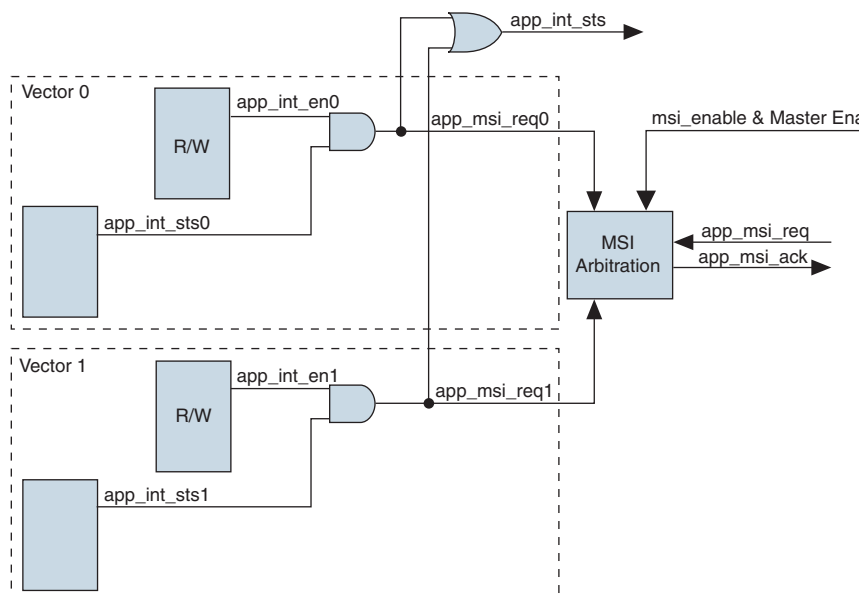


Figure 5-24 illustrates a possible implementation of the MSI handler block with a per vector enable bit. A global application interrupt enable can also be implemented instead of this per vector MSI.

Figure 5-24. 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 5-25, the endpoint requests eight MSIs but is only allocated two. In this case, the application layer must be designed to use only two allocated messages.

Figure 5-25. MSI Request Example

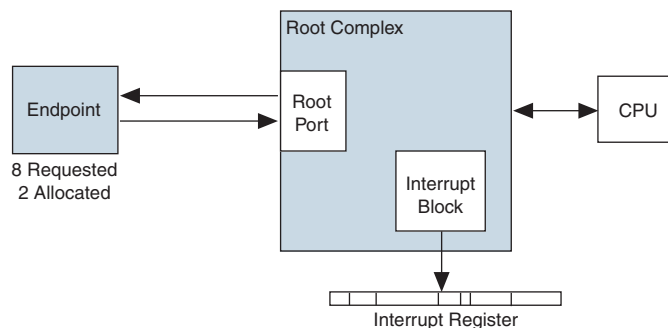
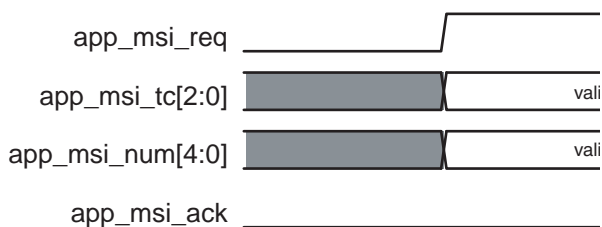


Figure 5-26 illustrates the interactions among MSI interrupt signals for the root port in Figure 5-25. The minimum latency possible between app_msi_req and app_msi_ack is one clock cycle.

Figure 5-26. MSI Interrupt Signals Waveform



Notes to Figure 5-26:

- (1) For variants using the Avalon-ST interface, `app_msi_req` can extend beyond `app_msi_ack` before deasserting. For descriptor/data variants, `app_msi_req` must deassert on the cycle following `app_msi_ack`.

Figure 5-27 illustrates interrupt timing for the legacy interface. In this figure the assertion of `app_int_ack` indicates that the `Assert_INTA` message TLP has been sent.

Figure 5-27. Legacy Interrupt Assertion

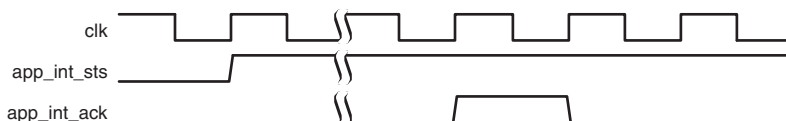


Figure 5-28 illustrates the timing for deassertion of legacy interrupts. The assertion of `app_int_ack` indicates that the `Deassert_INTA` message TLP has been sent.

Figure 5-28. Legacy Interrupt Deassertion

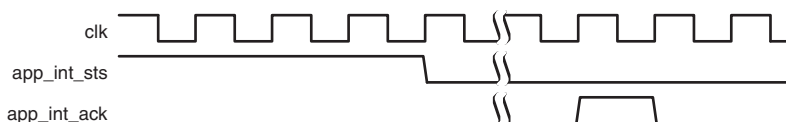


Table 5-13 describes 3 example implementations; one in which all 32 MSI messages are allocated and 2 in which only 4 are allocated.

Table 5-13. 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	29:0	1:0	2:0

MSI generated for hot plug, power management events, and system errors always use TC0. MSI 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.

PCI Express Interrupts for Root Ports

In root port mode, the PCI Express MegaCore function 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:

- If AER is enabled. When the AER option is turned on, 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 port and you must set it to a fixed value. 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 advanced error reporting extended capability structure. It is located at offset 0x830 of the configuration space registers.

Table 5-14 describes the signals available to a root port to handle interrupts.

Table 5-14. Interrupt Signals for Root Ports

Signal	I/O	Description
<code>int_status[3:0]</code>	I	These signals drive legacy interrupts to the application layer using a TLP of type Message Interrupt as follows: <ul style="list-style-type: none"> ■ <code>int_status[0]</code>: interrupt signal A ■ <code>int_status[1]</code>: interrupt signal B ■ <code>int_status[2]</code>: interrupt signal C ■ <code>int_status[3]</code>: interrupt signal D
<code>aer_msi_num[4:0]</code>	I	Advanced error reporting (AER) MSI number. This signal is used by AER to determine the offset between the base message data and the MSI to send. This signal is only available for root port mode.
<code>pex_msi_num[4:0]</code>	I	Power management MSI number. This signal is used by power management and/or hot plug to determine the offset between the base message interrupt number and the message interrupt number to send through MSI.
<code>serr_out</code>	O	System Error: This signal only applies to hard IP root port designs that report each system error detected by the MegaCore function, assuming the proper enabling bits are asserted in the root control register and the device control register. If enabled, <code>serr_out</code> is asserted for a single clock cycle when a system error occurs. System errors are described in the <i>PCI Express Base Specification 1.1 or 2.0</i> . in the root control register.

Configuration Space Signals—Hard IP Implementation

The configuration space signals provide access to the control and status information available in the configuration space registers. This interface is synchronous to `core_clk`. To access the configuration space from the application layer, you must synchronize to the application layer clock. Table 5-15 describes the configuration space interface and hot plug signals that are available in the hard IP implementation. Refer to Chapter 6 *PCI Express Base Specification Revision 2.0* for more information about the hot plug signals.

Table 5-15. Configuration Space Signals (Hard IP Implementation) (Part 1 of 2)

Signal	Width	Dir	Description
<code>tl_cfg_addr</code>	4	0	Address of the register that has been updated. The information updates every 8 <code>core_clks</code> along with <code>tl_cfg_ctl</code> .
<code>tl_cfg_ctl</code>	32	0	Data for the entire register that has been updated. Updates every 8 <code>core_clk</code> cycles.
<code>tl_cfg_ctl_wr</code>	1	0	Write signal. This signal toggles when <code>tl_cfg_ctl</code> has been updated (every 8 <code>core_clk</code> cycles). The toggle edge marks where the <code>tl_cfg_ctl</code> data changes. You can use this edge as a reference for determining when the data is safe to sample.
<code>tl_cfg_sts</code>	53	0	Config status bits. This information updates every 8 <code>core_clk</code> cycles. The <code>cfg_sts</code> group consists of (from MSB to LSB): <code>tl_cfg_sts[52:49]</code> = <code>cfg_devcsr[19:16]</code> error detection signal as follows: [correctable error reporting, enable, non-fatal error reporting enable, fatal error reporting enable, unsupported request reporting enable] <code>tl_cfg_sts[48]</code> = <code>cfg_slotcsr[24]</code> Data link layer state changed <code>tl_cfg_sts[47]</code> = <code>cfg_slotcsr[20]</code> Command completed <code>tl_cfg_sts[46:31]</code> = <code>cfg_linkcsr[31:16]</code> Link status bits <code>tl_cfg_sts[30]</code> = <code>cfg_link2csr[16]</code> Current de-emphasis level <code>tl_cfg_sts[29:25]</code> = <code>cfg_prmcsr[31:27]</code> 5 primary command status error bits <code>tl_cfg_sts[24]</code> = <code>cfg_prmcsr[24]</code> 6th primary command status error bit <code>tl_cfg_sts[23:6]</code> = <code>cfg_rootcsr[25:8]</code> PME bits <code>tl_cfg_sts[5:1]</code> = <code>cfg_seccsr[31:27]</code> 5 secondary command status error bits <code>tl_cfg_sts[0]</code> = <code>cfg_seccsr[4]</code> 6th secondary command status error bit

Table 5–15. Configuration Space Signals (Hard IP Implementation) (Part 2 of 2)

Signal	Width	Dir	Description
tl_cfg_sts_wr	1	0	Write signal. This signal toggles when tl_cfg_sts has been updated (every 8 core_clk cycles). The toggle marks the edge where tl_cfg_sts data changes. You can use this edge as a reference for determining when the data is safe to sample.
hpg_ctrler	5	1	<p>hpg_ctrler[0]: Attention button pressed: This signal is asserted when the attention button is pressed. If no attention button exists on the component, this bit should be hardwired to 0. You can use this signal to control an LED. When the button is pushed, the signal is asserted, and the LED blinks. If the button is pushed again within 5 seconds, the signal is deasserted and the LED stops blinking.</p> <p>In endpoint mode, the MegaCore function detects the rising edge of this signal and automatically generates the corresponding hot plug message. All other bits of this signal must be set to 0.</p> <p>hpg_ctrler[1]: Presence detect change: This signal is asserted when a presence detect change is detected. This signal is level sensitive.</p> <p>hpg_ctrler[2]: Manually-operated retention latch (MRL) sensor changed: This signal is asserted when an MRL sensor detects a change, indicating that a component is being removed from the fabric and should be powered down. Both the rising and falling edge of this signal are detected in order to set the corresponding hot plug status register bit. When this bit is set, a hot plug MSI or INT is generated, if enabled. This signal is level sensitive.</p> <p>hpg_ctrler[3]: Power fault detected: This signal is asserted when a power fault is detected on this slot. This signal is edge sensitive.</p> <p>hpg_ctrler[4]: Power controller status: This signal reports the status of the power controller. It is used to set the command complete register when the power controller status is equal to the power controller control signal. This signal is level sensitive.</p>

The tl_cfg_ctl signal is a multiplexed bus that contains the contents of configuration space registers as shown in Table 5–15. Information stored in the configuration space is accessed in round robin order where tl_cfg_addr indicates which register is being accessed. Table 5–16 shows the layout of configuration information that is multiplexed on tl_cfg_ctl.

Table 5–16. Multiplexed Configuration Register Information Available on tl_cfg_ctl (Note 1)

Address	31:24	23:16	15:8	7:0
0	cfg_devcsr[15:0]		cfg_dev2csr[15:0]	
1	cfg_slotcsr[31:16]		cfg_slotcsr[15:0]	
2	cfg_linkscr[15:0]		cfg_link2csr[15:0]	
3	8'h00	cfg_prmcsr[15:0]		cfg_rootcsr[7:0]
4	cfg_seccsr[15:0]		cfg_secbus[7:0]	cfg_subbus[7:0]
5	12'h000		cfg_io_bas[19:0]	
6	12'h000		cfg_io_lim[19:0]	
7	cfg_np_bas[11:0]	cfg_np_lim[11:0]		
8	cfg_pr_bas[31:0]			
9	20'h00000			cfg_pr_bas[43:32]
A	cfg_pr_lim[31:0]			

Table 5-16. Multiplexed Configuration Register Information Available on tl_cfg_ctl (Note 1)

Address	31:24	23:16	15:8	7:0
B	20'h00000			cfg_pr_lim[43:32]
C	cfg_pmcsr[31:0]			
D	cfg_msixcsr[15:0]		cfg_msicsr[15:0]	
E	8'h00	cfg_tcvcmmap[23:0]		
F	16'h0000		3'b000	cfg_busdev[12:0]
Note to Table 5-16: (1) Items in blue are only available for root ports.				

Table 5-17 describes the configuration space registers referred to in Table 5-15 and Table 5-16.

Table 5-17. Configuration Space Register Descriptions (Part 1 of 2)

Register	Width	Dir	Description
cfg_devcsr, cfg_dev2csr	32	0	cfg_devcsr[31:16] is device control and cfg_devcsr[15:0] is status of the PCI Express capability register.
cfg_slotcsr	16	0	cfg_slotcsr[31:16] is the slot control and cfg_slotcsr[15:0] is the slot status of the PCI Express capability register. This register is only available in root port mode.
cfg_linkcsr, cfg_link2csr	32	0	cfg_linkcsr[31:16] is the link status and cfg_linkcsr[15:0] is the link control of the PCI Express capability register. cfg_link2csr[31:16] is the second link status and cfg_link2csr[15:0] is the second control of the PCI Express capability register. For Gen1 variants, the link bandwidth notification bit is always set to 0. For Gen2 variants, this bit is set to 1.
cfg_prmcsr	32	0	Base/Primary control and status register for the PCI configuration space.
cfg_rootcsr	8	0	Root control and status register of the PCI-Express capability. This register is only available in root port mode.
cfg_seccsr	16	0	Secondary bus control and status register of the PCI-Express capability. This register is only available in root port mode.
cfg_secbus	8	0	Secondary bus number. Available in root port mode.
cfg_subbus	8	0	Subordinate bus number. Available in root port mode.
cfg_io_bas	20	0	IO base windows of the Type1 configuration space. This register is only available in root port mode.
cfg_io_lim	20	0	IO limit windows of the Type1 configuration space. This register is only available in root port mode.
cfg_np_bas	12	0	Non-prefetchable base windows of the Type1 configuration space. This register is only available in root port mode.
cfg_np_lim	12	0	Non-prefetchable limit windows of the Type1 configuration space. This register is only available in root port mode.
cfg_pr_bas	44	0	Prefetchable base windows of the Type1 configuration space. This register is only available in root port mode.
cfg_pr_lim	12	0	Prefetchable limit windows of the Type1 configuration space. Available in root port mode.

Table 5-17. Configuration Space Register Descriptions (Part 2 of 2)

Register	Width	Dir	Description
cfg_pmcsr	32	0	cfg_pmcsr[31:16] is power management control and cfg_pmcsr[15:0] the power management status register. This register is only available in root port mode.
cfg_msixcsr	16	0	MSI-X control and status register. Duplicated for each function implementing MSI-X.
cfg_msicsr	16	0	MSI control and status register. Duplicated for each function implementing MSI.
cfg_tcvcmap	24	0	Configuration traffic class/virtual channel mapping. The application layer uses this signal to generate a transaction layer packet mapped to the appropriate virtual 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_busdev	13	0	Bus/device number captured by or programmed in the core.

Figure 5-29 illustrates the timing of the tl_cfg_ctl interface.

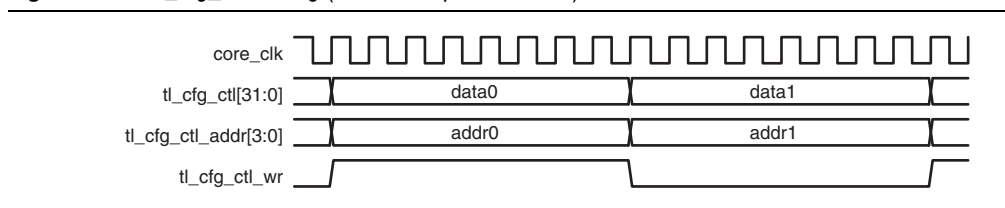
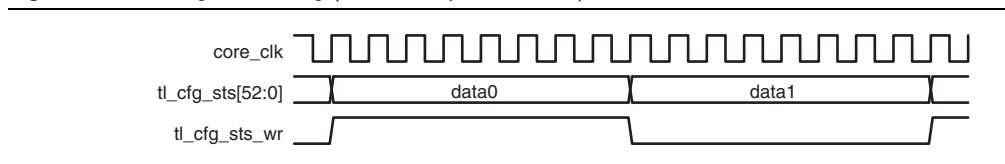
Figure 5-29. tl_cfg_ctl Timing (Hard IP Implementation)

Figure 5-30 illustrates the timing of the tl_cfg_sts interface.

Figure 5-30. tl_cfg_sts Timing (Hard IP Implementation)

Configuration Space Signals—Soft IP Implementation

The signals in Table 5-18 reflect the current values of several configuration space registers that the application layer may need to access. These signals are available in configurations using the Avalon-ST interface (soft IP implementation) or the descriptor/data Interface.

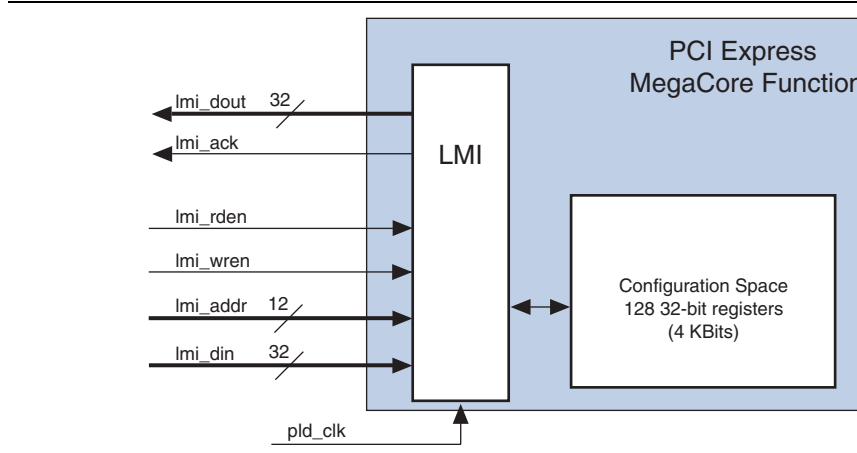
Table 5-18. Configuration Space Signals (Soft IP Implementation)

Signal	I/O	Description
cfg_tcvcmap[23:0]	0	Configuration traffic class/virtual channel mapping: The application layer uses this signal to generate a transaction layer packet mapped to the appropriate virtual 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_busdev[12:0]	0	Configuration bus device: This signal generates a transaction ID for each transaction layer packet, and indicates the bus and device number of the MegaCore function. Because the MegaCore function only implements one function, the function number of the transaction ID must be set to 000b. cfg_busdev[12:5]: Bus number. cfg_busdev[4:0]: Device number.
cfg_prmcsr[31:0]	0	Configuration primary control status register. The content of this register controls the PCI status.
cfg_devcsr[31:0]	0	Configuration dev control status register. Refer to the <i>PCI Express Base Specification</i> for details.
cfg_linkcsr[31:0]	0	Configuration link control status register. Refer to the <i>PCI Express Base Specification</i> for details.

LMI Signals—Hard IP Implementation

The LMI signals provide access to the PCI Express configuration space in the transaction layer of the MegaCore function. You can use these signals to read and write the configuration space registers. [Figure 5-31](#) illustrates the LMI interface. These signals are only available in the hard IP implementation.

Figure 5-31. Local Management Interface



The LMI interface is synchronized to `p1d_clk` and runs at frequencies up to 250 MHz. The LMI address is the same as the PCIe 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.

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 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 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.

Table 5-19 describes the signals that comprise the LMI interface.

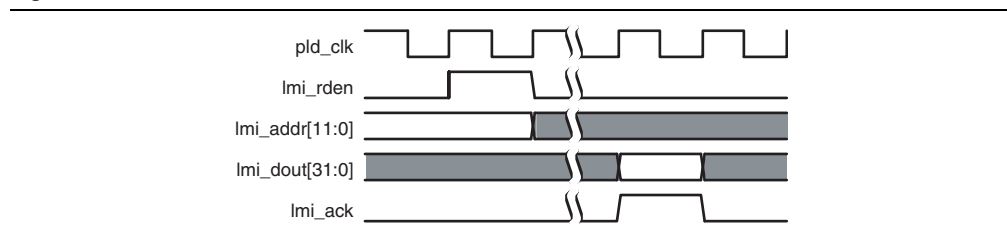
Table 5-19. LMI Interface

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

LMI Read Operation

Figure 5-32 illustrates the read operation. The read data remains available until the next local read or system reset.

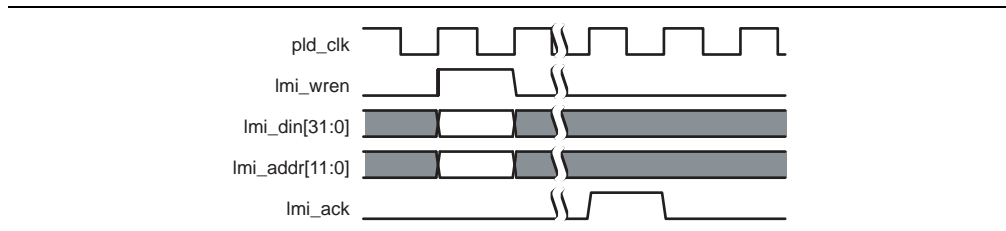
Figure 5-32. LMI Read



LMI Write Operation

Figure 5–33 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 5–33. LMI Write



PCI Express Reconfiguration Block Signals—Hard IP Implementation

The PCI Express reconfiguration block interface is implemented using an Avalon-MM slave interface with a 8-bit address and 16-bit data. For a detailed description of the Avalon-MM protocol, refer to the *Avalon Memory-Mapped Interfaces* chapter in the *Avalon Interface Specifications*.

Signal	I/O	Description
avs_pcie_reconfig_address[7:0]	I	A 8-bit address.
avs_pcie_reconfig_byteenable[1:0]	I	Byte enables, currently unused.
avs_pcie_reconfig_chipselect	I	Chipselect.
avs_pcie_reconfig_write	I	Write signal.
avs_pcie_reconfig_writedata[15:0]	I	16-bit write data bus.
avs_pcie_reconfig_waitrequest	O	Asserted when unable to respond to a read or write request. When asserted, the control signals to the slave remain constant. waitrequest can be asserted during idle cycles. An Avalon-MM master may initiate a transaction when waitrequest is asserted.
avs_pcie_reconfig_read	I	Read signal.
avs_pcie_reconfig_readdata[15:0]	O	16-bit read data bus.
avs_pcie_reconfig_readdatavalid	O	Read data valid signal.
avs_pcie_reconfig_clk	I	The Avalon-MM clock.
avs_pcie_reconfig_rstn	I	Active-low Avalon-MM reset.

Power Management Signals

Table 5–20 shows the MegaCore function’s power management signals. These signals are available in configurations using the Avalon-ST interface or Descriptor/Data interface.

Table 5–20. Power Management Signals

Signal	I/O	Description
pme_to_cr	I	<p>Power management turn off control register.</p> <p>Root port—When this signal is asserted, the root port sends the <code>PME_turn_off</code> message.</p> <p>Endpoint—This signal is asserted to acknowledge the <code>PME_turn_off</code> message by sending <code>pme_to_ack</code> to the root port.</p>
pme_to_sr	O	<p>Power management turn off status register.</p> <p>Root port—This signal is asserted for 1 clock cycle when the root port receives the <code>pme_turn_off</code> acknowledge message.</p> <p>Endpoint—This signal is asserted when the endpoint receives the <code>PME_turn_off</code> message from the root port. For the soft IP implementation, it is asserted until <code>pme_to_cr</code> is asserted. For the hard IP implementation, it is asserted for one cycle.</p>
cfg_pmcsr[31:0]	O	<p>Power management capabilities register. This register is read-only and provides information related to power management for a specific function. Refer to Table 5–21 and Table 5–22 for additional information. This signal only exists in soft IP implementation. In the hard IP implementation, this information is accessed through the configuration interface. Refer to “Configuration Space Signals—Hard IP Implementation” on page 5–25.</p>
pm_event	I	<p>Power Management Event. This signal is only available in the hard IP End Point implementation.</p> <p>Endpoint—initiates a <code>power_management_event</code> message (PM_PME) that is sent to the root port. If the MegaCore function is in a low power state, the link exists from the low-power state to send the message. This signal is positive edge-sensitive.</p>
pm_data[9:0]	I	<p>Power Management Data. This signal is only available in the hard IP implementation.</p> <p>This bus indicates power consumption of the component. This bus can only be implemented if all three bits of <code>AUX_power</code> (part of the Power Management Capabilities structure) are set to 0. This bus includes the following bits:</p> <ul style="list-style-type: none"> ■ <code>pm_data[9:2]</code>: Data Register: This register is used to maintain a value associated with the power consumed by the component. (Refer to the example below) ■ <code>pm_data[1:0]</code>: Data Scale: This register is used to maintain the scale used to find the power consumed by a particular component and can include the following values: <ul style="list-style-type: none"> b'00: unknown b'01: 0.1 × b'10: 0.01 × b'11: 0.001 × <p>For example, the two registers might have the following values:</p> <ul style="list-style-type: none"> ■ <code>pm_data[9:2]</code>: b'1110010 = 114 ■ <code>pm_data[1:0]</code>: b'10, which encodes a factor of .01 <p>To find the maximum power consumed by this component, multiply the data value by the data Scale (114 X .01 = 1.14). 1.14 watts is the maximum power allocated to this component in the power state selected by the <code>data_select</code> field.</p>
pm_auxpwr	I	<p>Power Management Auxiliary Power: This signal is only available in the hard IP implementation. This signal must be used to inform the MegaCore function that an auxiliary power has been detected.</p>

Table 5–21 shows the layout of the Power Management Capabilities register.

Table 5–21. Power Management Capabilities Register

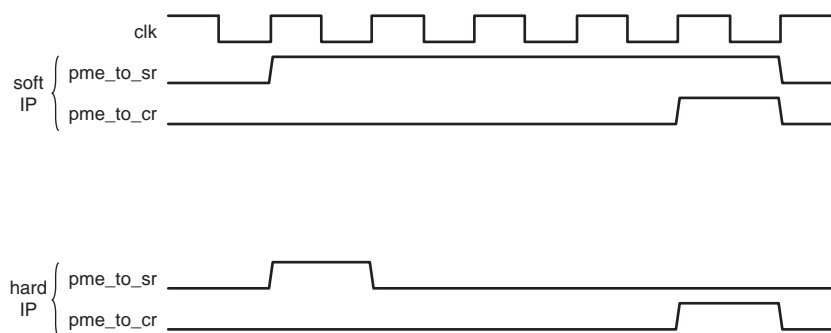
31	24	22	16	15	14	13	12	9	8	7	2	1	0
data register		rsvd		PME_status	data_scale		data_select		PME_EN	rsvd		PM_state	

Table 5–22 outlines the use of the various fields of the Power Management Capabilities register.

Table 5–22. Power Management Capabilities Register Field Descriptions

Bits	Field	Description
[31:24]	Data register	This field indicates in which power states a function can assert the PME# message.
[22:16]	reserved	—
[15]	PME_status	When this signal is set to 1, it 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#
[7:2]	reserved	—
[1:0]	PM_state	Specifies the power management state of the operating condition being described. Defined encodings are: <ul style="list-style-type: none"> ■ 2b'00 D0 ■ 2b'01 D1 ■ 2b'10 D2 ■ 2b'11 D A device returns 2b'11 in this field and Aux or PME Aux in the type register to specify the <i>D3-Cold PM</i> state. An encoding of 2b'11 along with any other type register value specifies the <i>D3-Hot</i> state.

Figure 5–34 illustrates the behavior of pme_to_sr and pme_to_cr in an endpoint. First, the MegaCore function receives the PME_turn_off message. Then, the application tries to send the PME_to_ack message to the root port.

Figure 5–34. pme_to_sr and pme_to_cr in an Endpoint MegaCore function

Completion Side Band Signals

Table 5–23 describes the signals that comprise the completion side band for the Avalon-ST interface.

Table 5–23. Completion Signals for Avalon-ST

Signal	I/O	Description
cpl_err[6:0]	I	<p>Completion error. This signal reports completion errors to the configuration space. When an error occurs, the appropriate signal is asserted for one cycle.</p> <ul style="list-style-type: none"> ■ 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 and the error is correctable. The MegaCore function 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 and the error is not correctable. The MegaCore function automatically generates a non-advisory error message that is sent to the root complex. ■ cpl_err[2]: Completer abort error. This signal must be asserted when a target block cannot process a non-posted request. In this case, the target block generates and sends a completion packet with completer abort (CA) status to the requestor and then asserts this error signal to the MegaCore function. The block automatically generates the error message and sends it to the root complex. ■ cpl_err[3]: Unexpected completion error. This signal must be asserted when a master block detects an unexpected completion transaction, for example, if no completion resource is waiting for a specific packet. It is used in both the soft IP implementation Avalon-ST interface and the hard IP implementation. ■ cpl_err[4]: Unsupported request error for posted TLP. Used in soft IP implementation Avalon-ST interface and the hard IP implementation. ■ cpl_err[5]: Unsupported request error for non-posted TLP. Used in soft IP implementation Avalon-ST interface and the hard IP implementation.

Table 5-23. Completion Signals for Avalon-ST

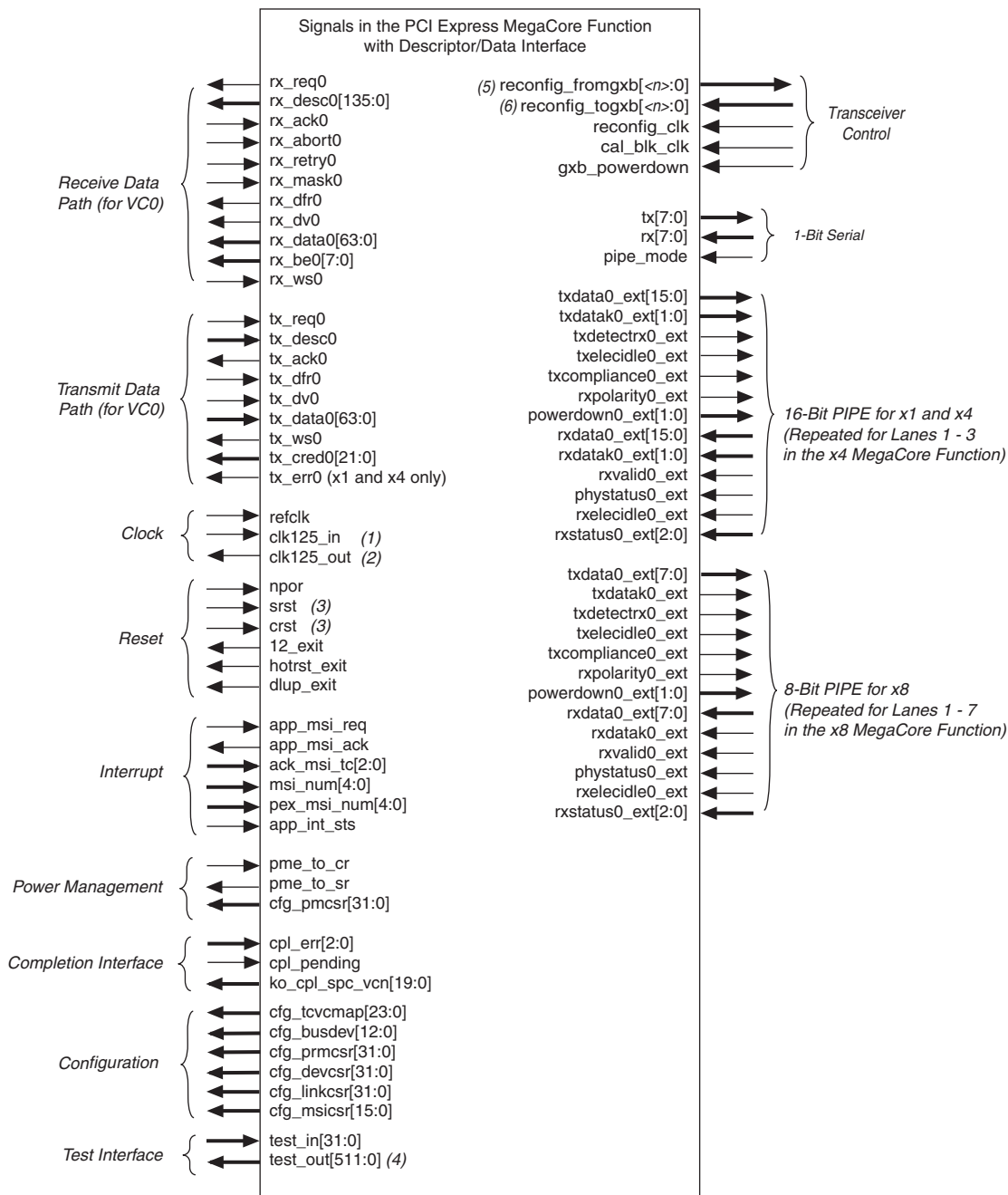
Signal	I/O	Description
		<ul style="list-style-type: none"> ■ <code>cpl_err[6]</code>: Log header. When asserted, logs <code>err_desc_func0</code> header. Used in soft IP implementation Avalon-ST interface and the hard IP implementation. For the $\times 8$ soft IP, only bits [3:1] are used. $\times 1$, $\times 4$ use all bits.
<code>err_desc_func0</code> [127:0]	I	TLP Header corresponding to a <code>cpl_err</code> . Logged by the MegaCore function when <code>cpl_err[6]</code> is asserted. This signal is only available for the $\times 1$ and $\times 4$ soft IP implementation. In the hard IP implementation, this information can be written to the AER header log register through the LMI interface.
<code>cpl_pending</code>	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. If this signal is asserted and low power mode is requested, the MegaCore function waits for the deassertion of this signal before transitioning into low-power state.

Descriptor/Data Interface

Figure 5-35 shows all the signals for PCI Express MegaCore function using the descriptor/data interface. In this figure, the transmit and receive signals apply to each implemented virtual channel, while configuration and global signals are common to all virtual channels on a link.



Altera recommends choosing the Avalon-ST interface for all new designs.

Figure 5-35. PCI Express MegaCore Function with Descriptor Data Interface**Notes to Figure 5-35:**

- (1) clk125_in replaced with clk250_in for x8 MegaCore function
- (2) clk125_out replaced with clk250_out for x8 MegaCore function
- (3) srst and crst removed for x8 MegaCore function
- (4) test_out [511:0] replaced with test_out [127:0] for x8 MegaCore function
- (5) Available in Stratix II GX, Stratix IV GX, Arria GX, and Hardcopy IV GX devices. The reconfig_fromgxb is a single wire for Stratix II GX and Arria GX. For Stratix IV GX, <n> = 16 for x1 and x4 MegaCore functions and <n> = 33 for the x8 MegaCore function.
- (6) Available in Stratix II GX, Stratix IV GX, Arria GX, and Hardcopy IV GX devices. For Stratix II GX and Arria GX reconfig_togxb, <n> = 2. For Stratix IV GX, <n> = 3.

Table 5–24 lists the interfaces for this MegaCore with links to the sections that describe each interface.

Table 5–24. Signal Groups in the PCI Express MegaCore Function using the Descriptor/Data Interface

Signal Group	Description
Logical	
Descriptor Rx	“Receive Operation Interface Signals” on page 5–37
Descriptor Tx	“Transmit Operation Interface Signals” on page 5–47
Clock	“Clock Signals Soft IP Implementation” on page 5–16
Reset	“Reset Signals” on page 5–16
Interrupt	“PCI Express Interrupts for Endpoints” on page 5–19
Configuration space	“Configuration Space Signals—Soft IP Implementation” on page 5–28
Power management	“PCI Express Reconfiguration Block Signals—Hard IP Implementation” on page 5–31
Completion	“Completion Interface Signals for Descriptor/Data Interface” on page 5–59
Physical	
Transceiver Control	“Transceiver Control” on page 5–66
Serial	“Serial Interface Signals” on page 5–67
Pipe	“PIPE Interface Signals” on page 5–68
Test	
Test	“Test Signals” on page 5–69

Receive Operation Interface Signals

The receive interface, like the transmit interface, is based on two independent buses: one for the descriptor phase (`rx_desc[135:0]`) and one for the data phase (`rx_data[63:0]`). Every transaction includes a descriptor. A descriptor is a standard transaction layer packet header as defined by the *PCI Express Base Specification 1.0a, 1.1 or 2.0* with two exceptions. Bits 126 and 127 indicate the transaction layer packet group and bits 135:128 describe BAR and address decoding information (refer to `rx_desc[135:0]` in Table 5–25 for details).

Receive Datapath Interface Signals

Receive datapath signals can be divided into the following two groups:

- Descriptor phase signals
- Data phase signals



In the following tables, transmit interface signal names with a 0 suffix are for virtual channel 0. If the MegaCore function implements additional virtual channels, there are an additional set of signals with the virtual channel number.

Table 5–25 describes the standard Rx descriptor phase signals.

Table 5–25. Rx Descriptor Phase Signals (Part 1 of 2)

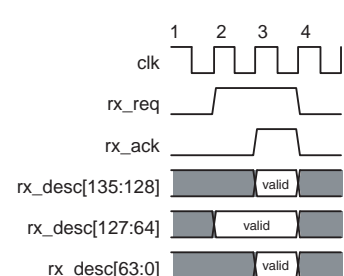
Signal	I/O	Description
<code>rx_req<n></code> (1),(2)	0	Receive request. This signal is asserted by the MegaCore function to request a packet transfer to the application interface. It is asserted when the first 2 DWORDS of a transaction layer packet header are valid. This signal is asserted for a minimum of 2 clock cycles; <code>rx_abort</code> , <code>rx_retry</code> , and <code>rx_ack</code> cannot be asserted at the same time as this signal. The complete descriptor is valid on the second clock cycle that this signal is asserted.
<code>rx_desc<n></code> [135:0] (1),(2)	0	<p>Receive descriptor bus. Bits [125:0] have the same meaning as a standard transaction layer packet header as defined by the <i>PCI Express Base Specification Revision 1.0a, 1.1 or 2.0</i>. Byte 0 of the header occupies bits [127:120] of the <code>rx_desc</code> bus, byte 1 of the header occupies bits [119:112], and so on, with byte 15 in bits [7:0]. Refer to Appendix A, Transaction Layer Packet Header Formats for the header formats.</p> <p>For bits [135:128] (descriptor and BAR decoding), refer to Table 5–26. Completion transactions received by an endpoint do not have any bits asserted and must be routed to the master block in the application layer.</p> <p><code>rx_desc</code> [127:64] begins transmission on the same clock cycle that <code>rx_req</code> is asserted, allowing precoding and arbitration to begin as quickly as possible. The other bits of <code>rx_desc</code> are not valid until the following clock cycle as shown in the following figure.</p>  <p>Bit 126 of the descriptor indicates the type of transaction layer packet in transit:</p> <ul style="list-style-type: none"> ■ <code>rx_desc</code> [126] when set to 0: transaction layer packet without data ■ <code>rx_desc</code> [126] when set to 1: transaction layer packet with data
<code>rx_ack<n></code> (1),(2)	1	Receive acknowledge. This signal is asserted for 1 clock cycle when the application interface acknowledges the descriptor phase and starts the data phase, if any. The <code>rx_req</code> signal is deasserted on the following clock cycle and the <code>rx_desc</code> is ready for the next transmission. <code>rx_ack</code> is independent of <code>rx_dv</code> and <code>rx_data</code> . It cannot be used to backpressure <code>rx_data</code> . You can use <code>rx_ws</code> to insert wait states.
<code>rx_abort<n></code> (1),(2)	1	Receive abort. This signal is asserted by the application interface if the application cannot accept the requested descriptor. In this case, the descriptor is removed from the receive buffer space, flow control credits are updated, and, if necessary, the application layer generates a completion transaction with unsupported request (UR) status on the transmit side.
<code>rx_retry<n></code> (1),(2)	1	Receive retry. The application interface asserts this signal if it is not able to accept a non-posted request. In this case, the application layer must assert <code>rx_mask<n></code> along with <code>rx_retry<n></code> so that only posted and completion transactions are presented on the receive interface for the duration of <code>rx_mask<n></code> .

Table 5-25. Rx Descriptor Phase Signals (Part 2 of 2)

Signal	I/O	Description
<code>rx_mask<n></code> (1),(2)	I	Receive mask (non-posted requests). This signal is used to mask all non-posted request transactions made to the application interface to present only posted and completion transactions. This signal must be asserted with <code>rx_retry<n></code> and deasserted when the MegaCore function can once again accept non-posted requests.

Notes to Table 5-25:

- (1) where `<n>` is the virtual channel number; For `×1` and `×4`, `n` can be 0 - 3
- (2) For `×8`, `<n>` can be 0 or 1

The MegaCore function generates the eight MSBs of this signal with BAR decoding information. Refer to Table 5-26.

Table 5-26. `rx_desc[135:128]`: Descriptor and BAR Decoding (Note 1)

Bit	Type 0 Component
128	= 1: BAR 0 decoded
129	= 1: BAR 1 decoded
130	= 1: BAR 2 decoded
131	= 1: BAR 3 decoded
132	= 1: BAR 4 decoded
133	= 1: BAR 5 decoded
134	= 1: Expansion ROM decoded
135	Reserved

Note to Table 5-26:

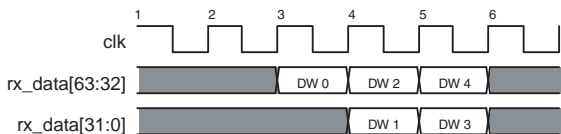
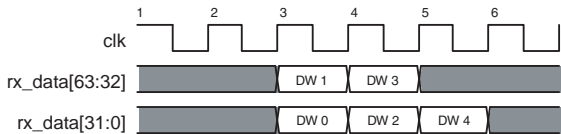
- (1) Only one bit of [135:128] is asserted at a time.

Table 5-27 describes the data phase signals.

Table 5-27. Rx Data Phase Signals (Part 1 of 2)

Signal	I/O	Description
<code>rx_dfr<n></code> (1),(2)	O	Receive data phase framing. This signal is asserted on the same or subsequent clock cycle as <code>rx_req</code> to request a data phase (assuming a data phase is needed). It is deasserted on the clock cycle preceding the last data phase to signal to the application layer the end of the data phase. The application layer does not need to implement a data phase counter.
<code>rx_dv<n></code> (1),(2)	O	Receive data valid. This signal is asserted by the MegaCore function to signify that <code>rx_data[63:0]</code> contains data.

Table 5-27. Rx Data Phase Signals (Part 2 of 2)

Signal	I/O	Description
<code>rx_data<n>[63:0]</code> (1),(2)	0	<p>Receive data bus. This bus transfers data from the link to the application layer. It is 2 DWORDS wide and is naturally aligned with the address in one of two ways, depending on bit 2 of <code>rx_desc</code>.</p> <ul style="list-style-type: none"> ■ <code>rx_desc[2]</code> (64-bit address) when 0: The first DWORD is located on <code>rx_data[31:0]</code>. ■ <code>rx_desc[34]</code> (32-bit address) when 0: The first DWORD is located on bits <code>rx_data[31:0]</code>. ■ <code>rx_desc[2]</code> (64-bit address) when 1: The first DWORD is located on bits <code>rx_data[63:32]</code>. ■ <code>rx_desc[34]</code> (32-bit address) when 1: The first DWORD is located on bits <code>rx_data[63:32]</code>. <p>This natural alignment allows you to connect <code>rx_data[63:0]</code> directly to a 64-bit datapath aligned on a QW address (in the little endian convention).</p> <p>Bit 2 is set to 1 (5 DWORD transaction)</p> <p>Figure 5-36.</p>  <p>Bit 2 is set to 0 (5 DWORD transaction)</p> <p>Figure 5-37.</p> 
<code>rx_be<n>[7:0]</code> (1),(2)	0	<p>Receive byte enable. These signals qualify data on <code>rx_data[63:0]</code>. Each bit of the signal indicates whether the corresponding byte of data on <code>rx_data[63:0]</code> is valid. These signals are not available in the x8 MegaCore function.</p>
<code>rx_ws<n></code> (1),(2)	1	<p>Receive wait states. With this signal, the application layer can insert wait states to throttle data transfer.</p>
<code>tx_cred0[21:0]</code>	0	<p>Refer to Table 5-32 for a description of the bits in this bus.</p>

Notes to Table 5-27:

- (1) where `<n>` is the virtual channel number; For x1 and x4, `n` can be 0 - 3
(2) For x8, `<n>` can be 0 or 1

Transaction Examples Using Receive Signals

This section provides the following additional examples that illustrate how transaction signals interact:

- Transaction without Data Payload LRetried transaction and masked non-posted transactions
- Transaction Aborted
- Transaction with Data Payload
- Transaction with Data Payload and Wait States

Transaction without Data Payload

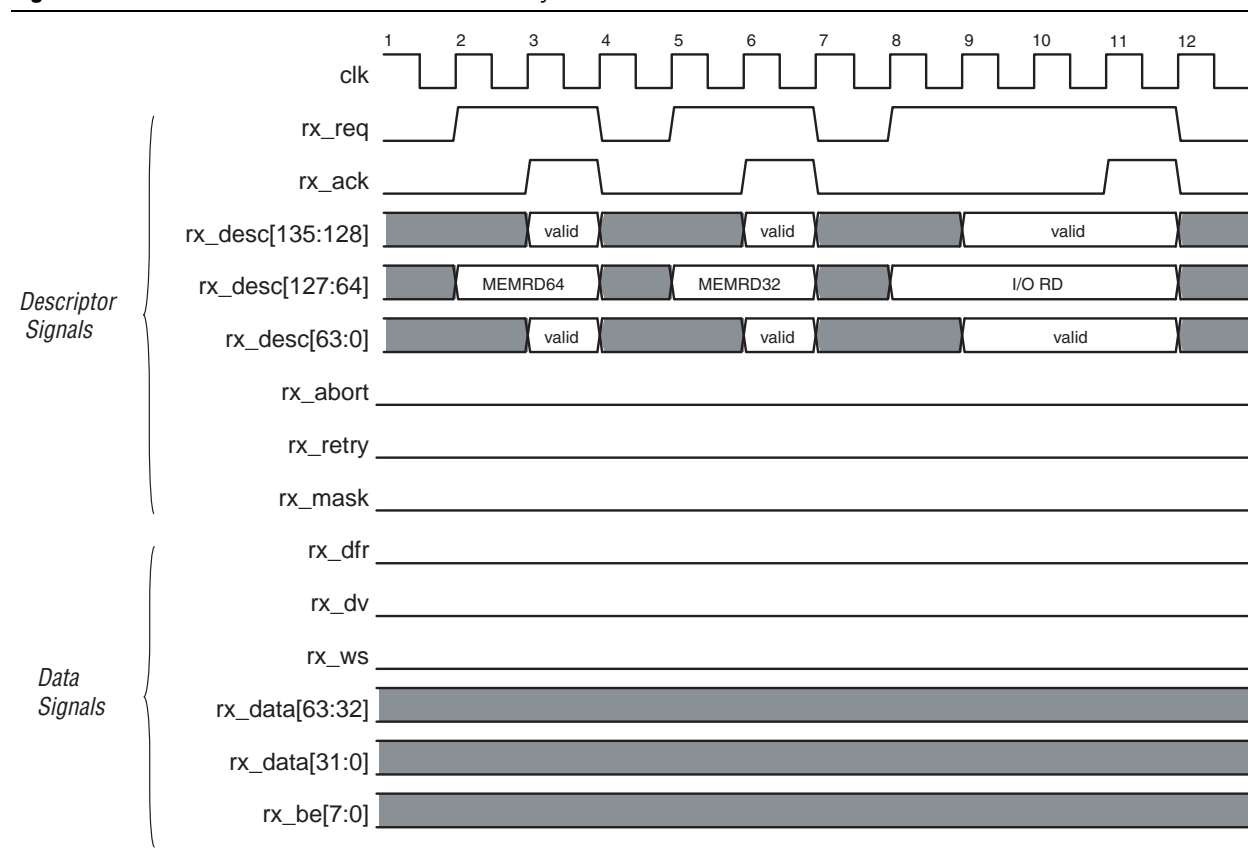
In Figure 5-38, the MegaCore function receives three consecutive transactions, none of which have data payloads:

- Memory read request (64-bit addressing mode)
- Memory read request (32-bit addressing mode)
- I/O read request

In clock cycles 4, 7, and 12, the MegaCore function updates flow control credits after each transaction layer packet has either been acknowledged or aborted. When necessary, the MegaCore function generates flow control DLLPs to advertise flow control credit levels.

The I/O read request initiated at clock cycle 8 is not acknowledged until clock cycle 11 with assertion of `rx_ack`. The relatively late acknowledgment could be due to possible congestion.

Figure 5-38. Rx Three Transactions without Data Payloads Waveform



Retried Transaction and Masked Non-Posted Transactions

When the application layer can no longer accept non-posted requests, one of two things happen: either the application layer requests the packet be resent or it asserts `rx_mask`. For the duration of `rx_mask`, the MegaCore function masks all non-posted transactions and reprioritizes waiting transactions in favor of posted and completion transactions. When the application layer can once again accept non-posted transactions, `rx_mask` is deasserted and priority is given to all non-posted transactions that have accumulated in the receive buffer.

Each virtual channel has a dedicated datapath and associated buffers and no ordering relationships exist between virtual channels. While one virtual channel may be temporarily blocked, data flow continues across other virtual channels without impact. Within a virtual channel, reordering is mandatory only for non-posted transactions to prevent deadlock. Reordering is not implemented in the following cases:

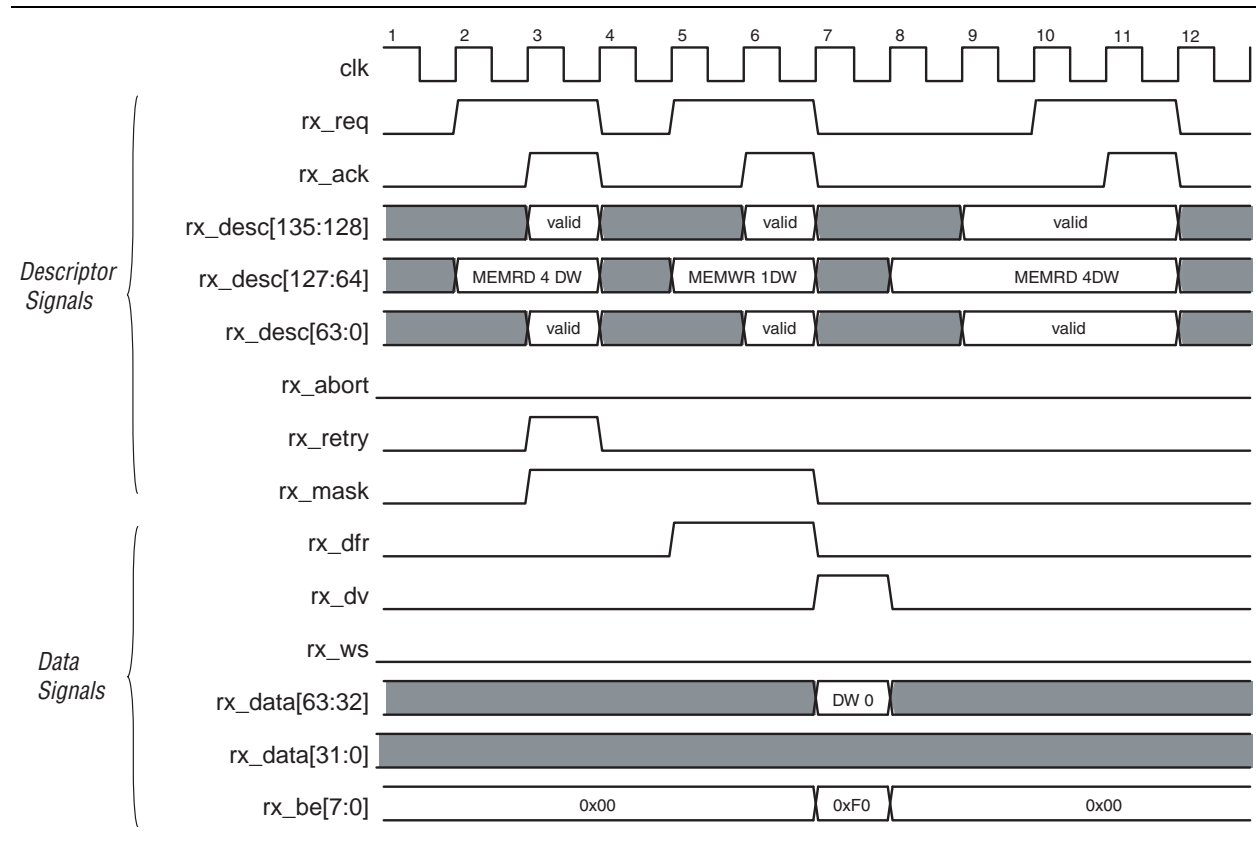
- Between traffic classes mapped in the same virtual channel
- Between posted and completion transactions
- Between transactions of the same type regardless of the relaxed-ordering bit of the transaction layer packet

In [Figure 5-39](#), the MegaCore function receives a memory read request transaction of 4 DWORDS that it cannot immediately accept. A second transaction (memory write transaction of one DWORD) is waiting in the receive buffer. Bit 2 of `rx_data[63:0]` for the memory write request is set to 1.

In clock cycle three, transmission of non-posted transactions is not permitted for as long as `rx_mask` is asserted.

Flow control credits are updated only after a transaction layer packet has been extracted from the receive buffer and both the descriptor phase and data phase (if any) have ended. This update happens in clock cycles 8 and 12 in [Figure 5-39](#).

Figure 5-39. Rx Retried Transaction and Masked Non-Posted Transaction Waveform

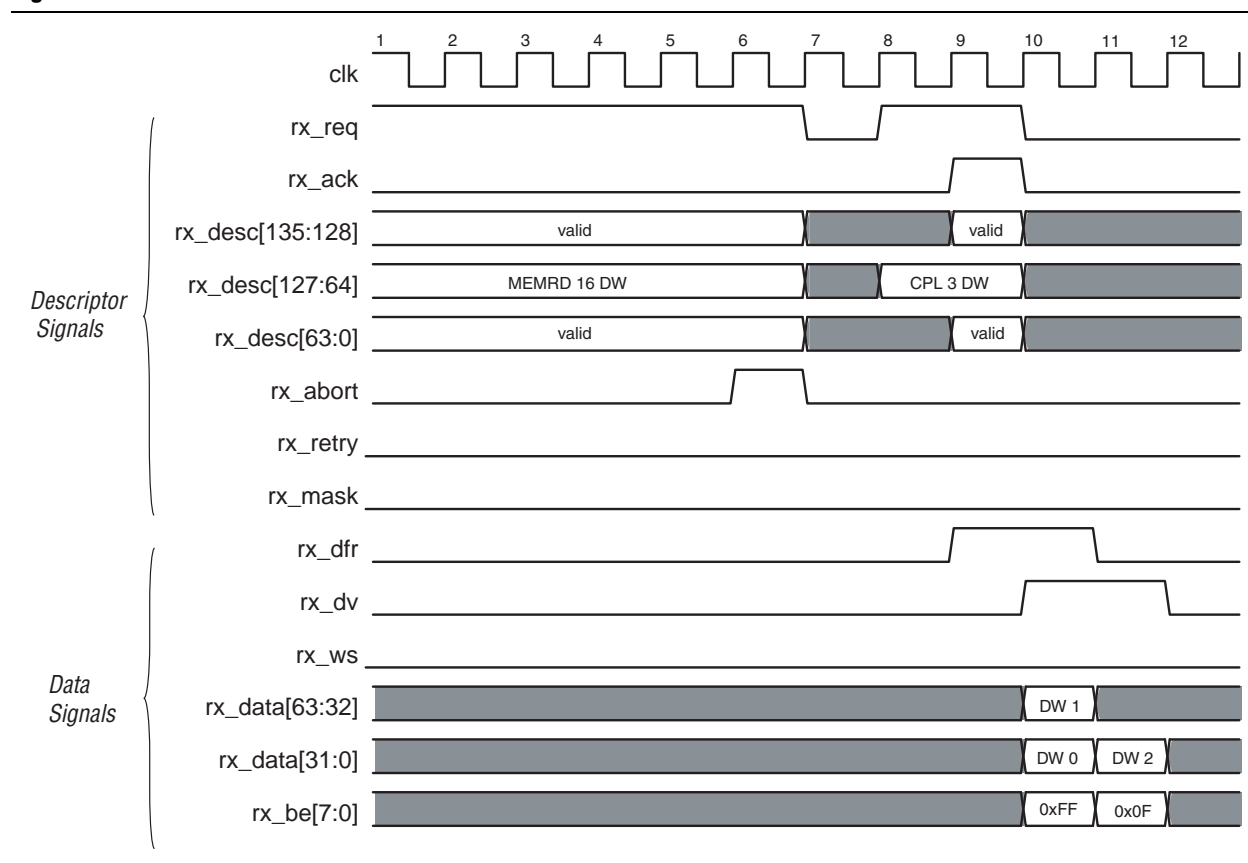


Transaction Aborted

In Figure 5-40, a memory read of 16 DWORDS is sent to the application layer. Having determined it will never be able to accept the transaction layer packet, the application layer discards it by asserting `rx_abort`. An alternative design might implement logic whereby all transaction layer packets are accepted and, after verification, potentially rejected by the application layer. An advantage of asserting `rx_abort` is that transaction layer packets with data payloads can be discarded in one clock cycle.

Having aborted the first transaction layer packet, the MegaCore function can transmit the second, a three DWORD completion in this case. The MegaCore function does not treat the aborted transaction layer packet as an error and updates flow control credits as if the transaction were acknowledged. In this case, the application layer is responsible for generating and transmitting a completion with completer abort status and to signal a completer abort event to the MegaCore function configuration space through assertion of `cpl_err`.

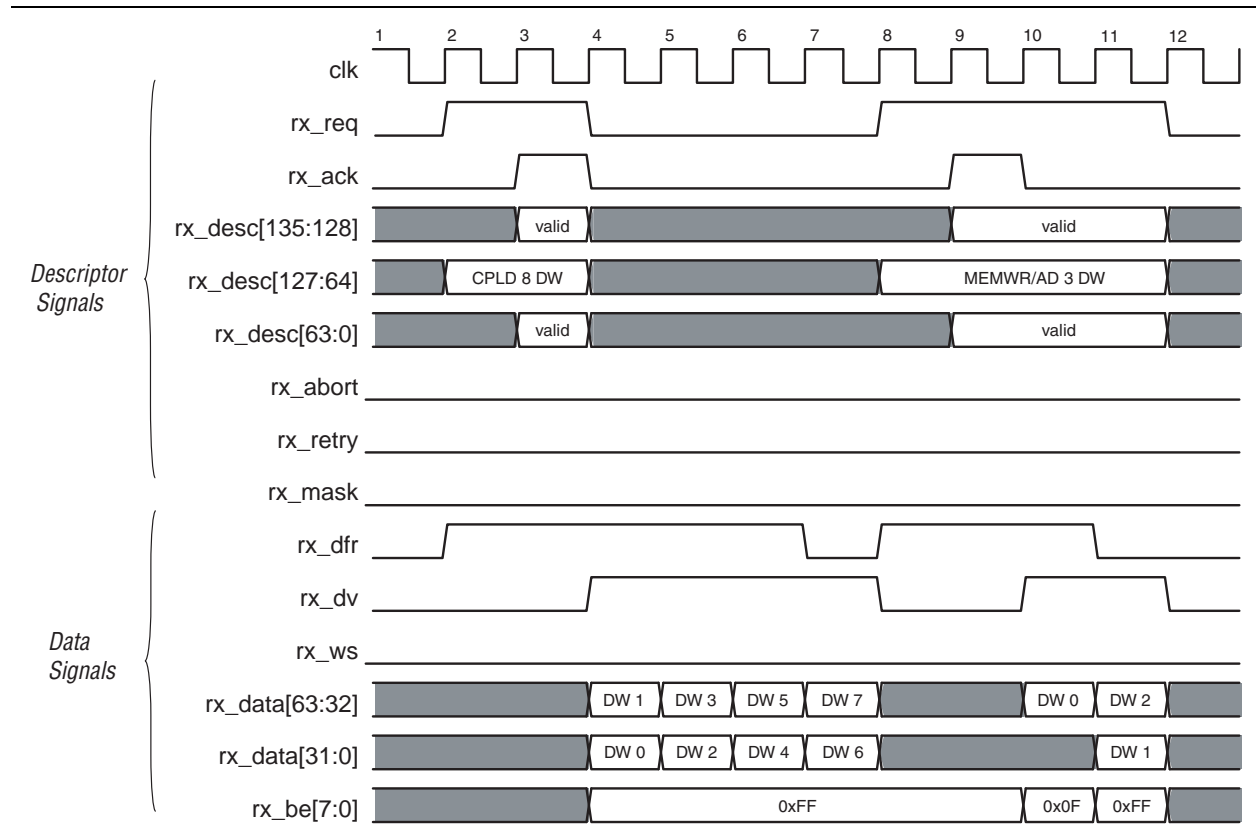
In clock cycle 6, `rx_abort` is asserted and transmission of the next transaction begins on clock cycle number.

Figure 5-40. Rx Aborted Transaction Waveform**Transaction with Data Payload**

In [Figure 5-41](#), the MegaCore function receives a completion transaction of eight DWORDS and a second memory write request of three DWORDS. Bit 2 of `rx_data[63:0]` is set to 0 for the completion transaction and to 1 for the memory write request transaction.

Normally, `rx_dfr` is asserted on the same or following clock cycle as `rx_req`; however, in this case the signal is already asserted until clock cycle 7 to signal the end of transmission of the first transaction. It is immediately reasserted on clock cycle eight to request a data phase for the second transaction.

Figure 5-41. Rx Transaction with a Data Payload Waveform

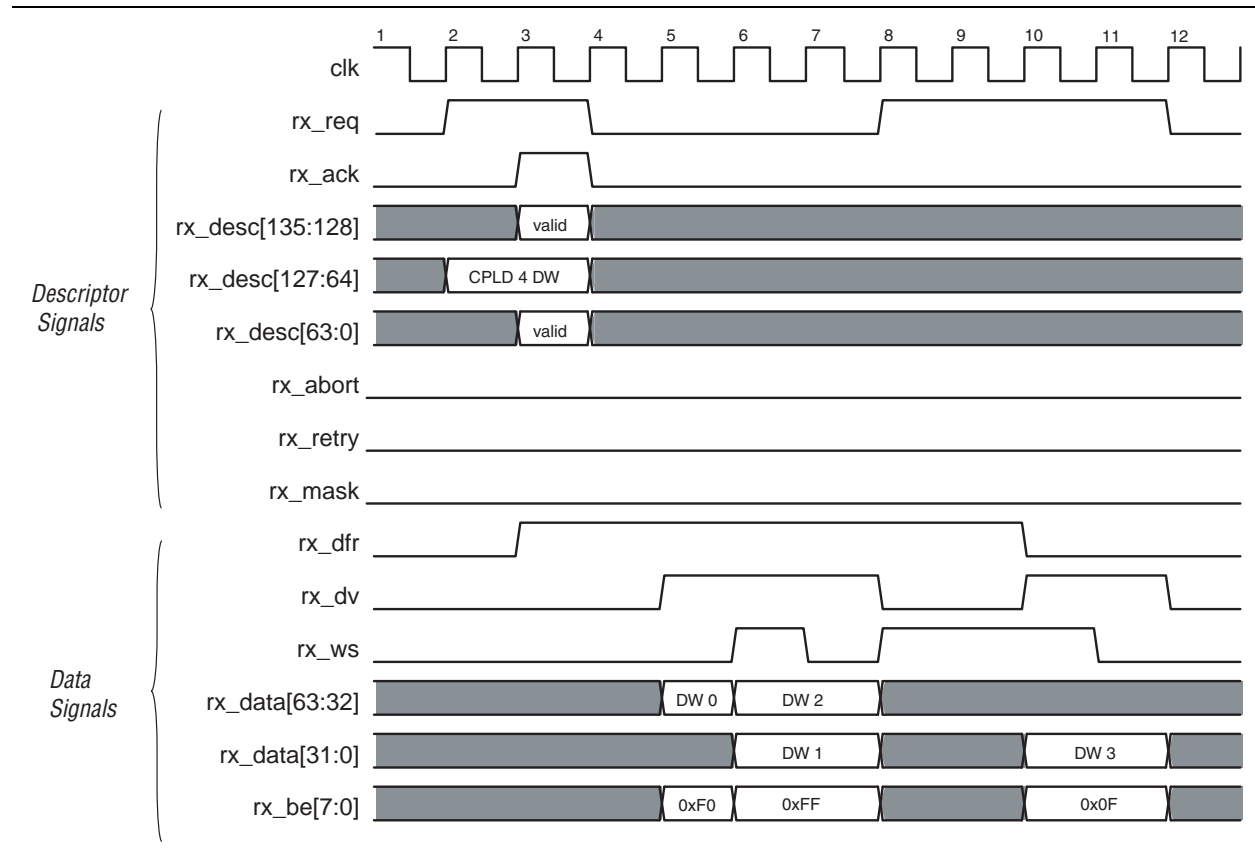


Transaction with Data Payload and Wait States

The application layer can assert `rx_ws` without restrictions. In [Figure 5-42](#), the MegaCore function receives a completion transaction of four DWORDS. Bit 2 of `rx_data[63:0]` is set to 1. Both the application layer and the MegaCore function insert wait states. Normally `rx_data[63:0]` would contain data in clock cycle 4, but the MegaCore function has inserted a wait state by deasserting `rx_dv`.

In clock cycle 11, data transmission does not resume until both of the following conditions are met:

- The MegaCore function asserts `rx_dv` at clock cycle 10, thereby ending a MegaCore function-induced wait state.
- The application layer deasserts `rx_ws` at clock cycle 11, thereby ending an application interface-induced wait state.

Figure 5-42. Rx Transaction with a Data Payload and Wait States Waveform

Dependencies Between Receive Signals

Table 5-28 describes the minimum and maximum latency values in clock cycles between various receive signals.

Table 5-28. Rx Minimum and Maximum Latency Values in Clock Cycles Between Receive Signals

Signal 1	Signal 2	Min	Typical	Max	Notes
<code>rx_req</code>	<code>rx_ack</code>	1	1	N	—
<code>rx_req</code>	<code>rx_dfr</code>	0	0	0	Always asserted on the same clock cycle if a data payload is present, except when a previous data transfer is still in progress. Refer to Figure 5-41 on page 5-45 .
<code>rx_req</code>	<code>rx_dv</code>	1	1-2	N	Assuming data is sent.
<code>rx_retry</code>	<code>rx_req</code>	1	2	N	<code>rx_req</code> refers to the next transaction request.

Transmit Operation Interface Signals

The transmit interface is established per initialized virtual channel and is based on two independent buses, one for the descriptor phase (`tx_desc [127:0]`) and one for the data phase (`tx_data [63:0]`). Every transaction includes a descriptor. A descriptor is a standard transaction layer packet header as defined by the *PCI Express Base Specification 1.0a, 1.1 or 2.0* with the exception of bits 126 and 127, which indicate the transaction layer packet group as described in the following section. Only transaction layer packets with a normal data payload include one or more data phases.

Transmit Datapath Interface Signals

The MegaCore function assumes that transaction layer packets sent by the application layer are well-formed; the MegaCore function does not detect malformed transaction layer packets sent by the application layer.

Transmit datapath signals can be divided into the following two groups:

- Descriptor phase signals
- Data phase signals



In the following tables, transmit interface signal names suffixed with 0 are for virtual channel 0. If the MegaCore function implements additional virtual channels, there are an additional set of signals suffixed with the virtual channel number.

Table 5-29 describes the standard Tx descriptor phase signals.

Table 5-29. Standard Tx Descriptor Phase Signals (Part 1 of 2)

Signal	I/O	Description
<code>tx_req<n></code> (1), (2)	I	Transmit request. This signal must be asserted for each request. It is always asserted with the <code>tx_desc [127:0]</code> and must remain asserted until <code>tx_ack</code> is asserted. This signal does not need to be deasserted between back-to-back descriptor packets.
<code>tx_desc<n>[127:0]</code> (1), (2)	I	<p>Transmit descriptor bus. The transmit descriptor bus, bits [127:0] of a transaction, can include a 3 or 4 DWORDS PCI Express transaction header. Bits have the same meaning as a standard transaction layer packet header as defined by the <i>PCI Express Base Specification Revision 1.0a, 1.1 or 2.0</i>. Byte 0 of the header occupies bits [127:120] of the <code>tx_desc</code> bus, byte 1 of the header occupies bits [119:112], and so on, with byte 15 in bits [7:0]. Refer to Appendix A, Transaction Layer Packet Header Formats for the header formats.</p> <p>The following bits have special significance:</p> <ul style="list-style-type: none"> ■ <code>tx_desc [2]</code> or <code>tx_desc [34]</code> indicate the alignment of data on <code>tx_data</code>. ■ <code>tx_desc [2]</code> (64-bit address) when 0: The first DWORD is located on <code>tx_data [31:0]</code>. ■ <code>tx_desc [34]</code> (32-bit address) when 0: The first DWORD is located on bits <code>tx_data [31:0]</code>. ■ <code>tx_desc [2]</code> (64-bit address) when 1: The first DWORD is located on bits <code>tx_data [63:32]</code>. ■ <code>tx_desc [34]</code> (32-bit address) when 1: The first DWORD is located on bits <code>tx_data [63:32]</code>.

Table 5-29. Standard Tx Descriptor Phase Signals (Part 2 of 2)

Signal	I/O	Description
<code>tx_desc<n>[127:0]</code> (1), (2) (cont.)	I	<p>Bit 126 of the descriptor indicates the type of transaction layer packet in transit:</p> <ul style="list-style-type: none"> ■ <code>tx_desc[126]</code> when 0: transaction layer packet without data ■ <code>tx_desc[126]</code> when 1: transaction layer packet with data <p>The following list provides a few examples of bit fields on this bus:</p> <ul style="list-style-type: none"> ■ <code>tx_desc[105:96] : length[9:0]</code> ■ <code>tx_desc[126:125] : fmt[1:0]</code> ■ <code>tx_desc[126:120] : type[4:0]</code>
<code>tx_ack<n></code> (1), (2)	0	Transmit acknowledge. This signal is asserted for one clock cycle when the MegaCore function acknowledges the descriptor phase requested by the application through the <code>tx_req</code> signal. On the following clock cycle, a new descriptor can be requested for transmission through the <code>tx_req</code> signal (kept asserted) and the <code>tx_desc</code> .

Notes to Table 5-29:

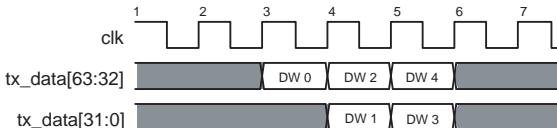
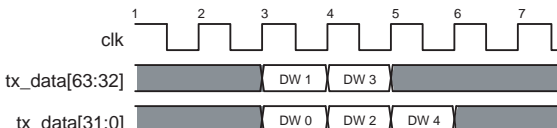
- (1) where `<n>` is the virtual channel number; For `x1` and `x4`, `n` can be 0 - 3
(2) For `x8`, `<n>` can be 0 or 1

Table 5-30 describes the standard Tx data phase signals.

Table 5-30. Standard Tx Data Phase Signals (Part 1 of 2)

Signal	I/O	Description
<code>tx_dfr<n></code> (1), (2)	I	Transmit data phase framing. This signal is asserted on the same clock cycle as <code>tx_req</code> to request a data phase (assuming a data phase is needed). This signal must be kept asserted until the clock cycle preceding the last data phase.
<code>tx_dv<n></code> (1), (2)	I	<p>Transmit data valid. This signal is asserted by the user application interface to signify that the <code>tx_data[63:0]</code> signal is valid. This signal must be asserted on the clock cycle following assertion of <code>tx_dfr</code> until the last data phase of transmission. The MegaCore function accepts data only when this signal is asserted and as long as <code>tx_ws</code> is not asserted.</p> <p>The application interface can rely on the fact that the first data phase never occurs before a descriptor phase is acknowledged (through assertion of <code>tx_ack</code>). However, the first data phase can coincide with assertion of <code>tx_ack</code> if the transaction layer packet header is only 3 DWORDS.</p>
<code>tx_ws<n></code> (1), (2)	0	<p>Transmit wait states. The MegaCore function uses this signal to insert wait states that prevent data loss. This signal might be used in the following circumstances:</p> <ul style="list-style-type: none"> ■ To give a DLLP transmission priority. ■ To give a high-priority virtual channel or the retry buffer transmission priority when the link is initialized with fewer lanes than are permitted by the link. <p>If the MegaCore function is not ready to acknowledge a descriptor phase (through assertion of <code>tx_ack</code>), it will automatically assert <code>tx_ws</code> to throttle transmission. When <code>tx_dv</code> is not asserted, <code>tx_ws</code> should be ignored.</p>

Table 5-30. Standard Tx Data Phase Signals (Part 2 of 2)

Signal	I/O	Description
tx_data<n>[63:0] (1), (2)	I	<p>Transmit data bus. This signal transfers data from the application interface to the link. It is 2 DWORDS wide and is naturally aligned with the address in one of two ways, depending on bit 2 of the transaction layer packet address, which is located on bit 2 or 34 of the tx_desc (depending on the 3 or 4 DWORDS transaction layer packet header bit 125 of the tx_desc signal).</p> <ul style="list-style-type: none"> tx_desc[2] (64-bit address) when 0: The first DWORD is located on tx_data[31:0]. tx_desc[34] (32-bit address) when 0: The first DWORD is located on bits tx_data[31:0]. tx_desc[2] (64-bit address) when 1: The first DWORD is located on bits tx_data[63:32]. tx_desc[34] (32-bit address) when 1: The first DWORD is located on bits tx_data[63:32]. <p>This natural alignment allows you to connect the tx_data[63:0] directly to a 64-bit datapath aligned on a QWORD address (in the little endian convention).</p> <p>Figure 5-43. Bit 2 is set to 1 (5 DWORDS transaction)</p>  <p>Figure 5-44. Bit 2 is set to 0 (5 DWORDS transaction)</p>  <p>The application layer must provide a properly formatted TLP on the Tx Data interface. The number of data cycles must be correct for the length and address fields in the header. Issuing a packet with an incorrect number data cycles will result in the Tx interface hanging and unable to accept further requests.</p>

Notes to Table 5-30:

- (1) where <n> is the virtual channel number; For x1 and x4, n can be 0 - 3
- (2) For x8, <n> can be 0 or 1

Table 5-31 describes the advanced data phase signals.

Table 5–31. Advanced Tx Data Phase Signals

Signal	I/O	Description
tx_cred<n>[65:0] (1),(2)	0	<p>Transmit credit. This signal controls the transmission of transaction layer packets of a particular type by the application layer based on the number of flow control credits available. This signal is optional because the MegaCore function always checks for sufficient credits before acknowledging a request. However, by checking available credits with this signal, the application can improve system performance by dividing a large transaction layer packet into smaller transaction layer packets based on available credits or arbitrating among different types of transaction layer packets by sending a particular transaction layer packet across a virtual channel that advertises available credits. Refer to Table 5–32 for the bit detail.</p> <p>Once a transaction layer packet is acknowledged by the MegaCore function, the corresponding flow control credits are consumed and this signal is updated 1 clock cycle after assertion of tx_ack.</p> <p>For a component that has received infinite credits at initialization, each field of this signal is set to its highest potential value.</p> <p>For the ×1 and ×4 MegaCore functions this signal is 22 bits wide with some encoding of the available credits to facilitate the application layer check of available credits. Refer to Table 5–29 for details.</p> <p>In the ×8 MegaCore function this signal is 66 bits wide and provides the exact number of available credits for each flow control type. Refer to Table 5–33 for details.</p>
tx_err<n> (1)	1	<p>Transmit error. This signal is used to discard or nullify a transaction layer packet, and is asserted for one clock cycle during a data phase. The MegaCore function automatically commits the event to memory and waits for the end of the data phase.</p> <p>Upon assertion of tx_err, the application interface should stop transaction layer packet transmission by deasserting tx_dfr and tx_dv.</p> <p>This signal only applies to transaction layer packets sent to the link (as opposed to transaction layer packets sent to the configuration space). If unused, this signal can be tied to zero. This signal is not available in the ×8 MegaCore function.</p>

Notes to Table 5–31:

- (1) where <n> is the virtual channel number; For ×1 and ×4, n can be 0 - 3
 (2) For ×8, <n> can be 0 or 1

[Table 5–32](#) shows the bit information for tx_cred0[21:0] for the ×1 and ×4 MegaCore functions.

Table 5–32. tx_cred0[21:0] Bits for the ×1 and ×4 MegaCore Functions (Part 1 of 2)

Bits	Value	Description
[0]	<ul style="list-style-type: none"> 0: No credits available 1: Sufficient credit available for at least 1 transaction layer packet 	Posted header.
[9:1]	<ul style="list-style-type: none"> 0: No credits available 1-256: number of credits available 257-511: reserved 	Posted data: 9 bits permit advertisement of 256 credits, which corresponds to 4 KBytes, the maximum payload size.
[10]	<ul style="list-style-type: none"> 0: No credits available 1: Sufficient credit available for at least 1 transaction layer packet 	Non-Posted header.

Table 5-32. tx_cred0[21:0] Bits for the x1 and x4 MegaCore Functions (Part 2 of 2)

Bits	Value	Description
[11]	<ul style="list-style-type: none"> 0: No credits available 1: Sufficient credit available for at least 1 transaction layer packet 	Non-Posted data.
[12]	<ul style="list-style-type: none"> 0: No credits available 1: Sufficient credit available for at least 1 transaction layer packet 	Completion header.
[21:13]	9 bits permit advertisement of 256 credits, which corresponds to 4 KBytes, the maximum payload size.	Completion data, posted data.

Table 5-33 shows the bit information for tx_cred<n>[65:0] for the x8 MegaCore functions.

Table 5-33. tx_cred[65:0] Bits for x8 MegaCore Function

Bits	Value	Description
tx_cred[7:0]	<ul style="list-style-type: none"> 0-127: Number of credits available >127: No credits available 	Posted header. Ignore this field if the value of posted header credits, tx_cred[60], is set to 1.
tx_cred[19:8]	<ul style="list-style-type: none"> 0-2047: Number of credits available >2047: No credits available 	Posted data. Ignore this field if the value of posted data credits, tx_cred[61], is set to 1.
tx_cred[27:20]	<ul style="list-style-type: none"> 0-127: Number of credits available >127: No credits available 	Non-posted header. Ignore this field if value of non-posted header credits, tx_cred[62], is set to 1.
tx_cred[39:28]	<ul style="list-style-type: none"> 0-2047: Number of credits available >2047: No credits available 	Non-posted data. Ignore this field if value of non-posted data credits, tx_cred[63], is set to 1.
tx_cred[47:40]	<ul style="list-style-type: none"> 0-127: Number of credits available >127: No credits available 	Completion header. Ignore this field if value of CPL header credits, tx_cred[64], is set to 1.
tx_cred[59:48]	<ul style="list-style-type: none"> 0-2047: Number of credits available >2047: No credits available 	Completion data. Ignore this field if value of CPL data credits, tx_cred[65], is set to 1.
tx_cred[60]	<ul style="list-style-type: none"> 0: Posted header credits are not infinite 1: Posted header credits are infinite 	Posted header credits are infinite when set to 1.
tx_cred[61]	<ul style="list-style-type: none"> 0: Posted data credits are not infinite 1: Posted data credits are infinite 	Posted data credits are infinite when set to 1.
tx_cred[62]	<ul style="list-style-type: none"> 0: Non-Posted header credits are not infinite 1: Non-Posted header credits are infinite 	Non-posted header credits are infinite when set to 1.
tx_cred[63]	<ul style="list-style-type: none"> 0: Non-posted data credits are not infinite 1: Non-posted data credits are infinite 	Non-posted data credits are infinite when set to 1.
tx_cred[64]	<ul style="list-style-type: none"> 0: Completion credits are not infinite 1: Completion credits are infinite 	Completion header credits are infinite when set to 1.
tx_cred[65]	<ul style="list-style-type: none"> 0: Completion data credits are not infinite 1: Completion data credits are infinite 	Completion data credits are infinite when set to 1.

Transaction Examples Using Transmit Signals

This section provides the following examples that illustrate how transaction signals interact:

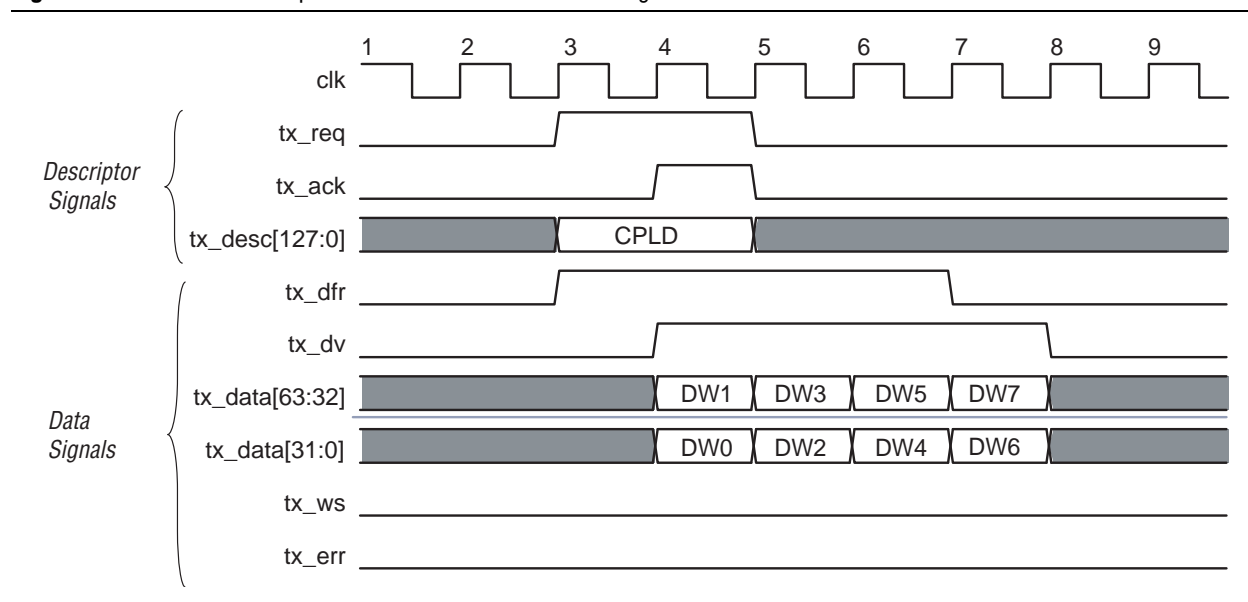
- Ideal Case Transmission
- Transaction Layer Not Ready to Accept Packet
- Possible Wait State Insertion
- Transmit Request Can Remain Asserted Between Transaction Layer Packets
- Priority Given Elsewhere
- Transmit Request Can Remain Asserted Between Transaction Layer Packets
- Multiple Wait States Throttle Data Transmission
- Error Asserted and Transmission Is Nullified

Ideal Case Transmission

In the ideal case, the descriptor and data transfer are independent of each other, and can even happen simultaneously. Refer to [Figure 5-45](#). The MegaCore function transmits a completion transaction of eight dwords. Address bit 2 is set to 0.

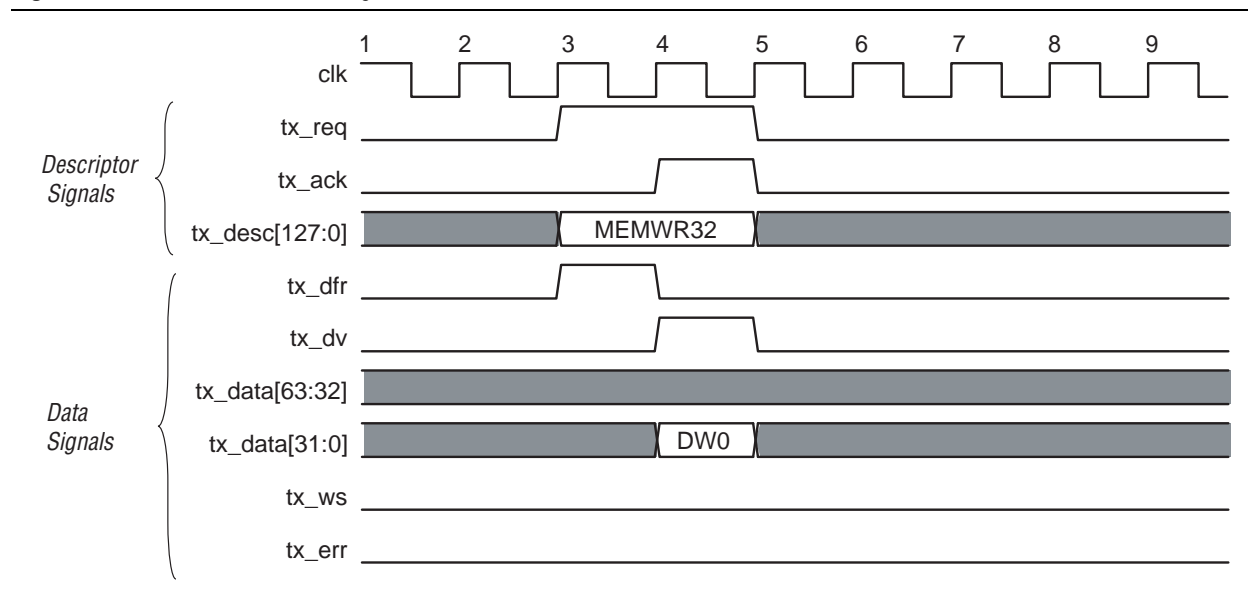
In clock cycle 4, the first data phase is acknowledged at the same time as transfer of the descriptor.

Figure 5-45. Tx 64-Bit Completion with Data Transaction of Eight DWORD Waveform



[Figure 5-46](#) shows the MegaCore function transmitting a memory write of one DWORD.

Figure 5-46. Tx Transfer for A Single DWORD Write



Transaction Layer Not Ready to Accept Packet

In this example, the application transmits a 64-bit memory read transaction of six DWORDs. Address bit 2 is set to 0. Refer to [Figure 5-47](#).

Data transmission cannot begin if the MegaCore function's transaction layer state machine is still busy transmitting the previous packet, as is the case in this example.

Figure 5-47. Tx State Machine Is Busy with the Preceding Transaction Layer Packet Waveform

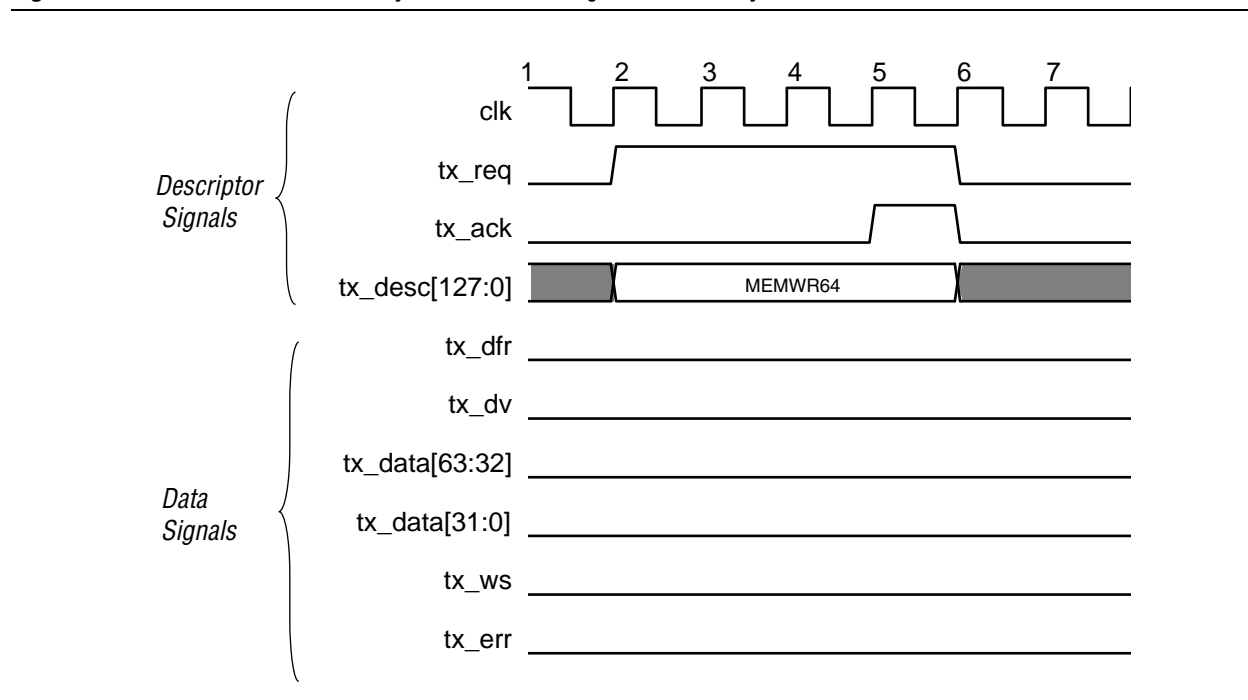
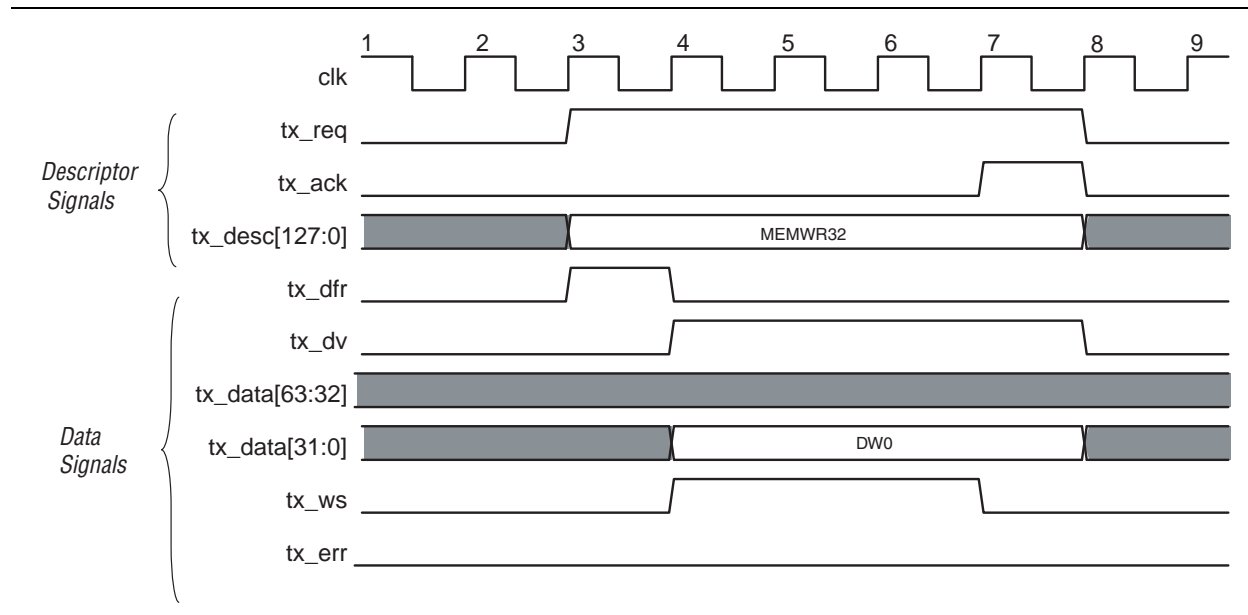


Figure 5-48 shows that the application layer must wait to receive an acknowledge before write data can be transferred. Prior to the start of a transaction (for example, `tx_req` being asserted), note that the `tx_ws` signal is set low for the $\times 1$ and $\times 4$ configurations and is set high for the $\times 8$ configuration.

Figure 5-48. Tx Transaction Layer Not Ready to Accept Packet



Possible Wait State Insertion

If the MegaCore function is not initialized with the maximum potential lanes, data transfer is necessarily hindered. Refer to Figure 5-50. The application transmits a 32-bit memory write transaction of 8 dwords. Address bit 2 is set to 0.

In clock cycle three, data transfer can begin immediately as long as the transfer buffer is not full.

In clock cycle five, once the buffer is full and the MegaCore function implements wait states to throttle transmission; four clock cycles are required per transfer instead of one because the MegaCore function is not configured with the maximum possible number of lanes implemented.

Figure 5-49 shows how the transaction layer extends the a data phase by asserting the wait state signal.

Figure 5-49. Tx Transfer with Wait State Inserted for a Single DWORD Write

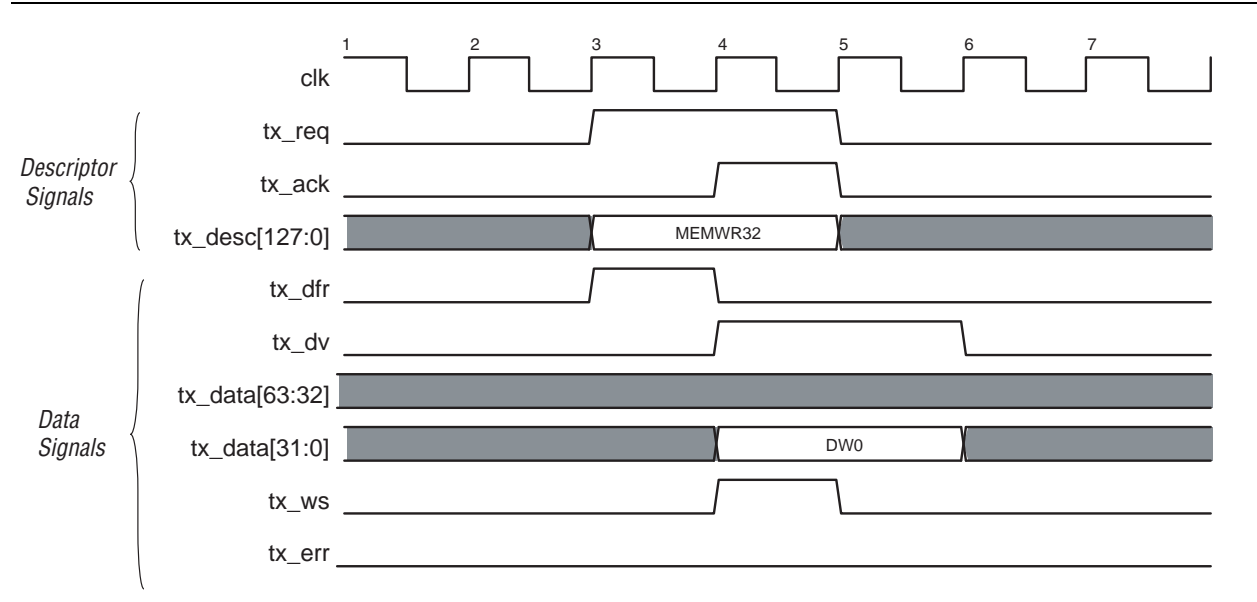
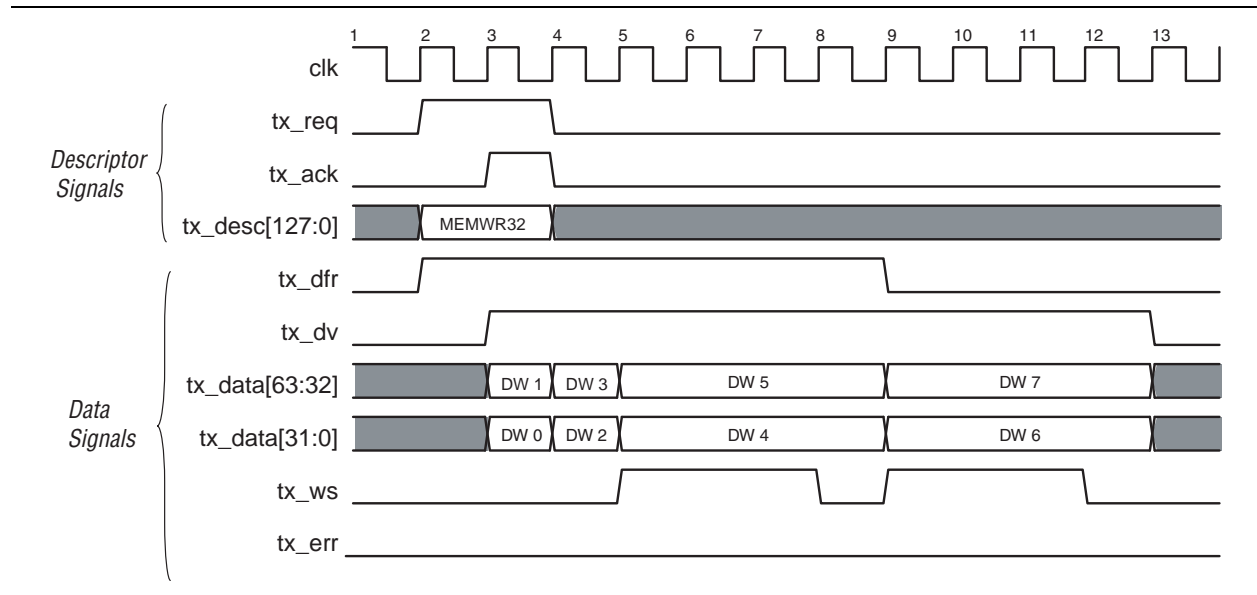


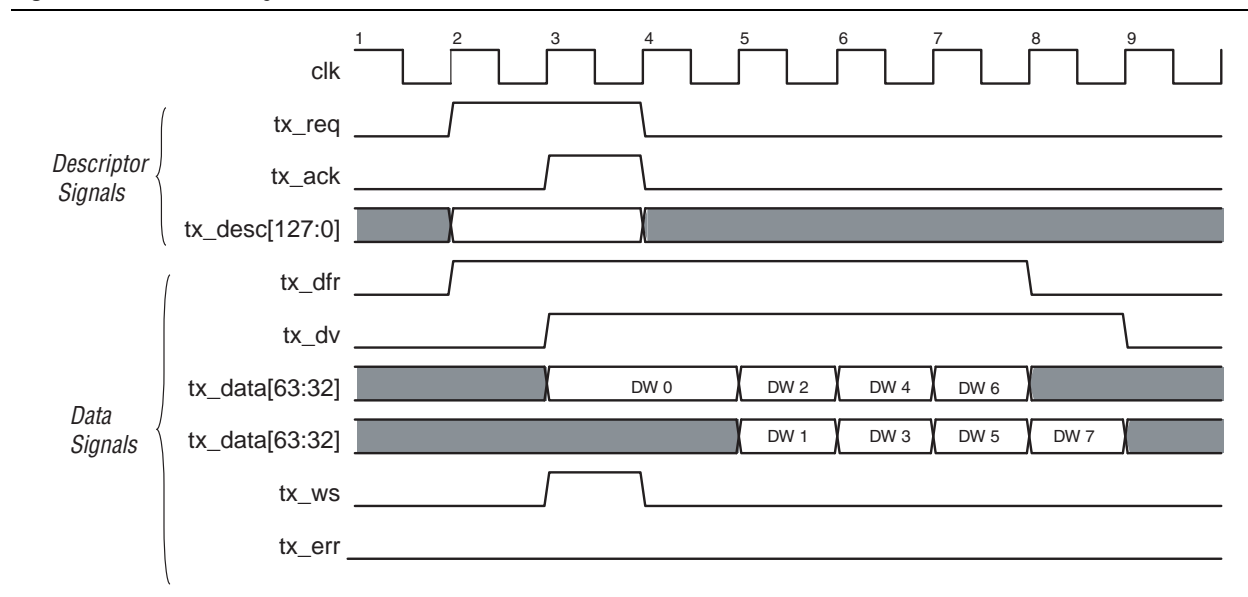
Figure 5-50. Tx Signal Activity When MegaCore Function Has Fewer than Maximum Potential Lanes Waveform



Transaction Layer Inserts Wait States because of Four Dword Header

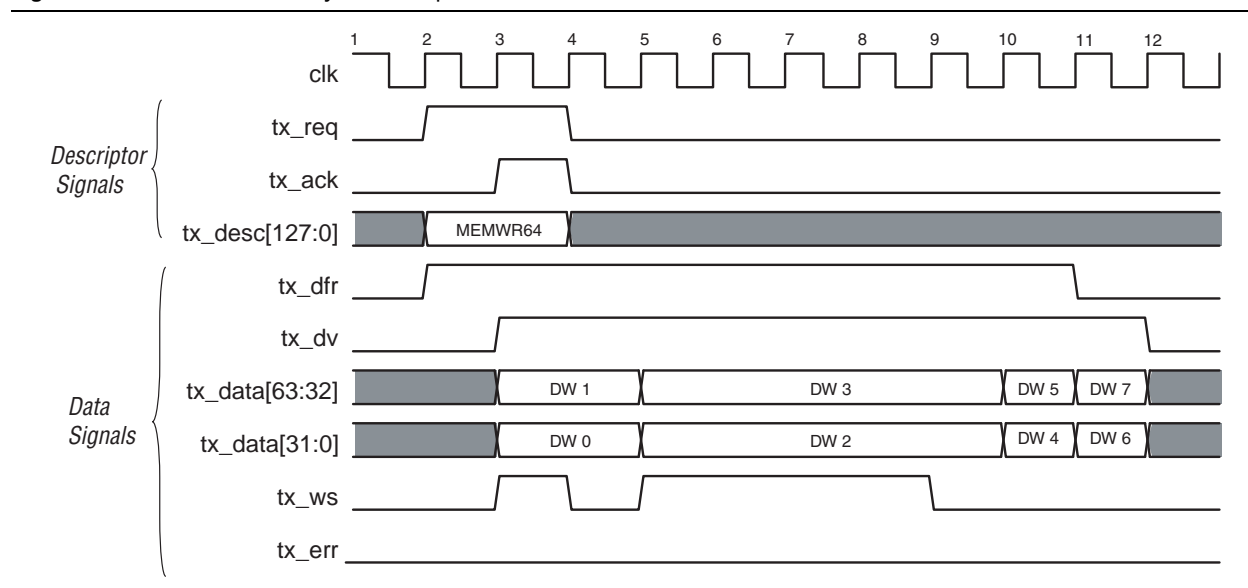
In this example, the application transmits a 64-bit memory write transaction. Address bit 2 is set to 1. Refer to [Figure 5-51](#). No wait states are inserted during the first two data phases because the MegaCore function implements a small buffer to give maximum performance during transmission of back-to-back transaction layer packets.

In clock cycle 3, the MegaCore function inserts a wait state because the memory write 64-bit transaction layer packet request has a 4-DWORD header. In this case, `tx_dv` could have been sent one clock cycle later.

Figure 5-51. Tx Inserting Wait States because of 4-DWORD Header Waveform**Priority Given Elsewhere**

In this example, the application transmits a 64-bit memory write transaction of 8 DWORDS. Address bit 2 is set to 0. The transmit path has a 3-deep, 64-bit buffer to handle back-to-back transaction layer packets as fast as possible, and it accepts the `tx_desc` and first `tx_data` without delay. Refer to [Figure 5-52](#).

In clock cycle five, the MegaCore function asserts `tx_ws` a second time to throttle the flow of data because priority was not given immediately to this virtual channel. Priority was given to either a pending data link layer packet, a configuration completion, or another virtual channel. The `tx_err` is not available in the $\times 8$ MegaCore function.

Figure 5-52. Tx 64-Bit Memory Write Request Waveform

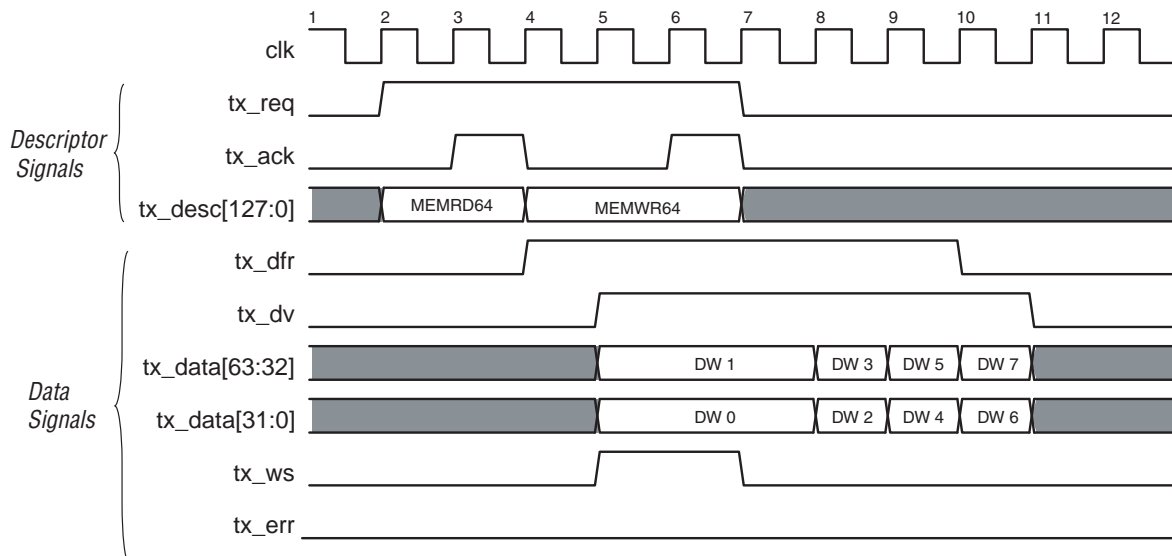
Transmit Request Can Remain Asserted Between Transaction Layer Packets

In this example, the application transmits a 64-bit memory read transaction followed by a 64-bit memory write transaction. Address bit 2 is set to 0. Refer to [Figure 5-53](#).

In clock cycle four, `tx_req` is not deasserted between transaction layer packets.

In clock cycle five, the second transaction layer packet is not immediately acknowledged because of additional overhead associated with a 64-bit address, such as a separate number and an LCRC. This situation leads to an extra clock cycle between two consecutive transaction layer packets.

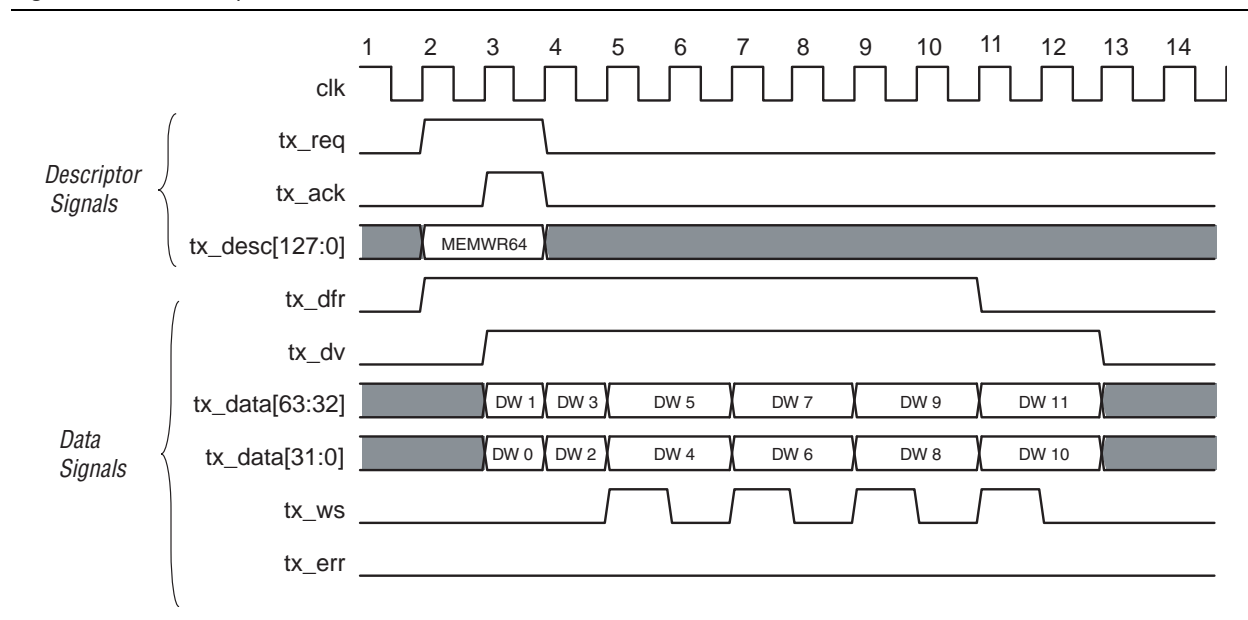
Figure 5-53. Tx 64-Bit Memory Read Request Waveform



Multiple Wait States Throttle Data Transmission

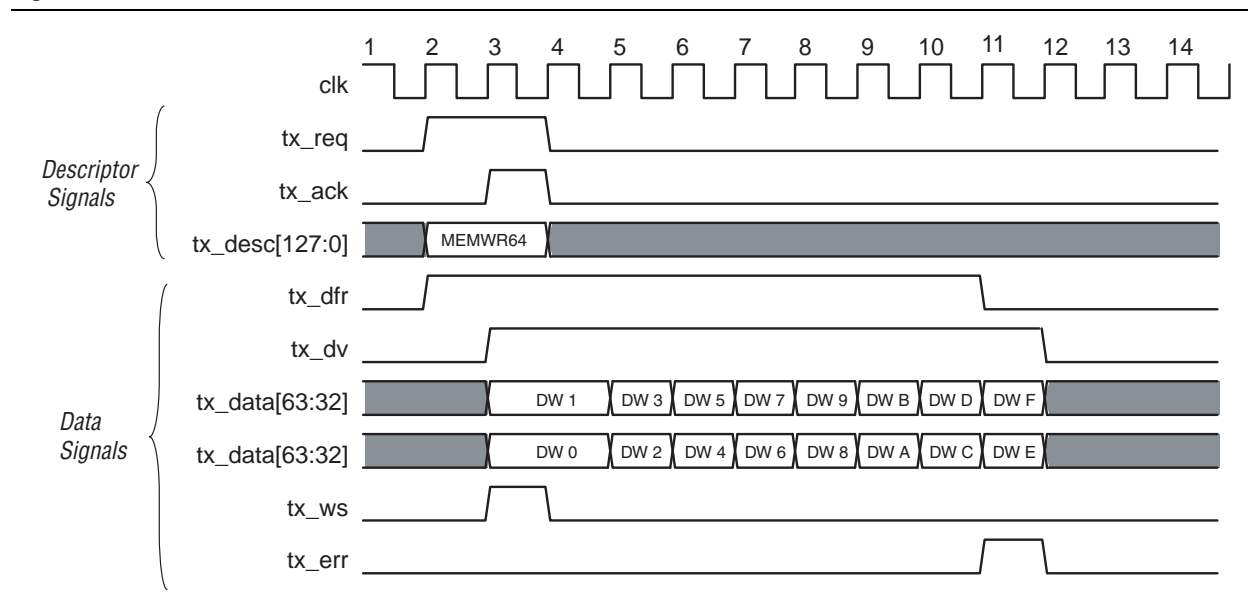
In this example, the application transmits a 32-bit memory write transaction. Address bit 2 is set to 0. Refer to [Figure 5-54](#). No wait states are inserted during the first two data phases because the MegaCore function implements a small buffer to give maximum performance during transmission of back-to-back transaction layer packets.

In clock cycles 5, 7, 9, and 11, the MegaCore function inserts wait states to throttle the flow of transmission.

Figure 5-54. Tx Multiple Wait States that Throttle Data Transmission Waveform**Error Asserted and Transmission Is Nullified**

In this example, the application transmits a 64-bit memory write transaction of 14 DWORDS. Address bit 2 is set to 0. Refer to [Figure 5-55](#).

In clock cycle 12, `tx_err` is asserted which nullifies transmission of the transaction layer packet on the link. Nullified packets have the LCRC inverted from the calculated value and use the end bad packet (EDB) control character instead of the normal END control character.

Figure 5-55. Tx Error Assertion Waveform

Completion Interface Signals for Descriptor/Data Interface

Table 5-34 describes the signals that comprise the completion interface for the descriptor/data interface.

Table 5-34. Completion Interface Signals

Signal	I/O	Description
<code>cpl_err[2:0]</code>	I	<p>Completion error. This signal reports completion errors to the configuration space by pulsing for one cycle. The three types of errors that the application layer must report are:</p> <ul style="list-style-type: none"> ■ <code>cpl_err[0]</code>: Completion timeout error. This signal must 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. The MegaCore function automatically generates an error message that is sent to the root complex. ■ <code>cpl_err[1]</code>: Completer abort error. This signal must be asserted when a target block cannot process a non-posted request. In this case, the target block generates and sends a completion packet with completer abort (CA) status to the requestor and then asserts this error signal to the MegaCore function. The block automatically generates the error message and sends it to the root complex. ■ <code>cpl_err[2]</code>: Unexpected completion error. This signal must be asserted when a master block detects an unexpected completion transaction, for example, if no completion resource is waiting for a specific packet. <p>For $\times 1$ and $\times 4$ the wrapper output is a 7-bit signal with the following format: <code>{3'h0, cpl_err[2:0], 1'b0}</code></p>
<code>cpl_pending</code>	I	<p>Completion pending. The application layer must assert this signal when a master block is waiting for completion, for example, when a transaction is pending. If this signal is asserted and low power mode is requested, the MegaCore function waits for the deassertion of this signal before transitioning into low-power state.</p>

Table 5-34. Completion Interface Signals

Signal	I/O	Description
ko_cpl_spc_vc<n>[19:0] (1)	0	<p>This static signal reflects the amount of Rx buffer space reserved for completion headers and data. It provides the same information as is shown in the Rx buffer space allocation table of the MegaWizard interface Buffer Setup page (refer to “Buffer Setup” on page 3-8). The bit field assignments for this signal are:</p> <ul style="list-style-type: none"> ■ ko_cpl_spc_vc<n>[7:0]: Number of completion headers that can be stored in the Rx buffer. ■ ko_cpl_spc_vc<n>[19:8]: Number of 16-byte completion data segments that can be stored in the Rx buffer. <p>The application layer logic is responsible for making sure that the completion buffer space does not overflow. It needs to limit the number and size of non-posted requests outstanding to ensure this. (2)</p>

Notes to Table 5-34:

- (1) where <n> is 0 - 3 for the x1 and x4 cores, and 0 - 1 for the x8 core
- (2) Receive Buffer size consideration: The receive buffer size is variable for the PCIe soft IP variations and fixed to 16 KByte per VC for the hard IP variation. The Rx Buffer size is set to accommodate optimum throughput of the PCIe link. The receive buffer collects all incoming TLPs from the PCIe link which consists of posted or non-posted TLPs. When configured as an endpoint, the PCI Express credit advertising mechanism prevents the Rx Buffer from overflowing for all TLP packets except incoming completion TLP packets because the endpoint variation advertises infinite credits for completion, per the *PCI Express Base Specification Revision 1.1 or 2.0*.

Therefore for endpoint variations, there could be some rare TLP completion sequences which could lead to a Rx Buffer overflow. For example, a sequence of 3 dword completion TLP using a qword aligned address would require 6 dwords of elapsed time to be written in the Rx buffer: 3 dwords for the TLP header, 1 dword for the TLP data, plus 2 dwords of PHY MAC and data link layer overhead. When using the Avalon-ST 128-bit interface, reading this TLP from the Rx Buffer requires 8 dwords of elapsed time. Therefore, theoretically, if such completion TLPs are sent back-to-back, without any gap introduced by DLLP, update FC or a skip character, the Rx Buffer will overflow because the read frequency does not offset the write frequency. This is certainly an extreme case and in practicalities such a sequence has a very low probability of occurring. However, to ensure that the Rx buffer never overflows with completion TLPs, Altera recommended building a circuit in the application layer which arbitrates the upstream memory read request TLP based on the available space in the completion buffer.

Avalon-MM Application Interface

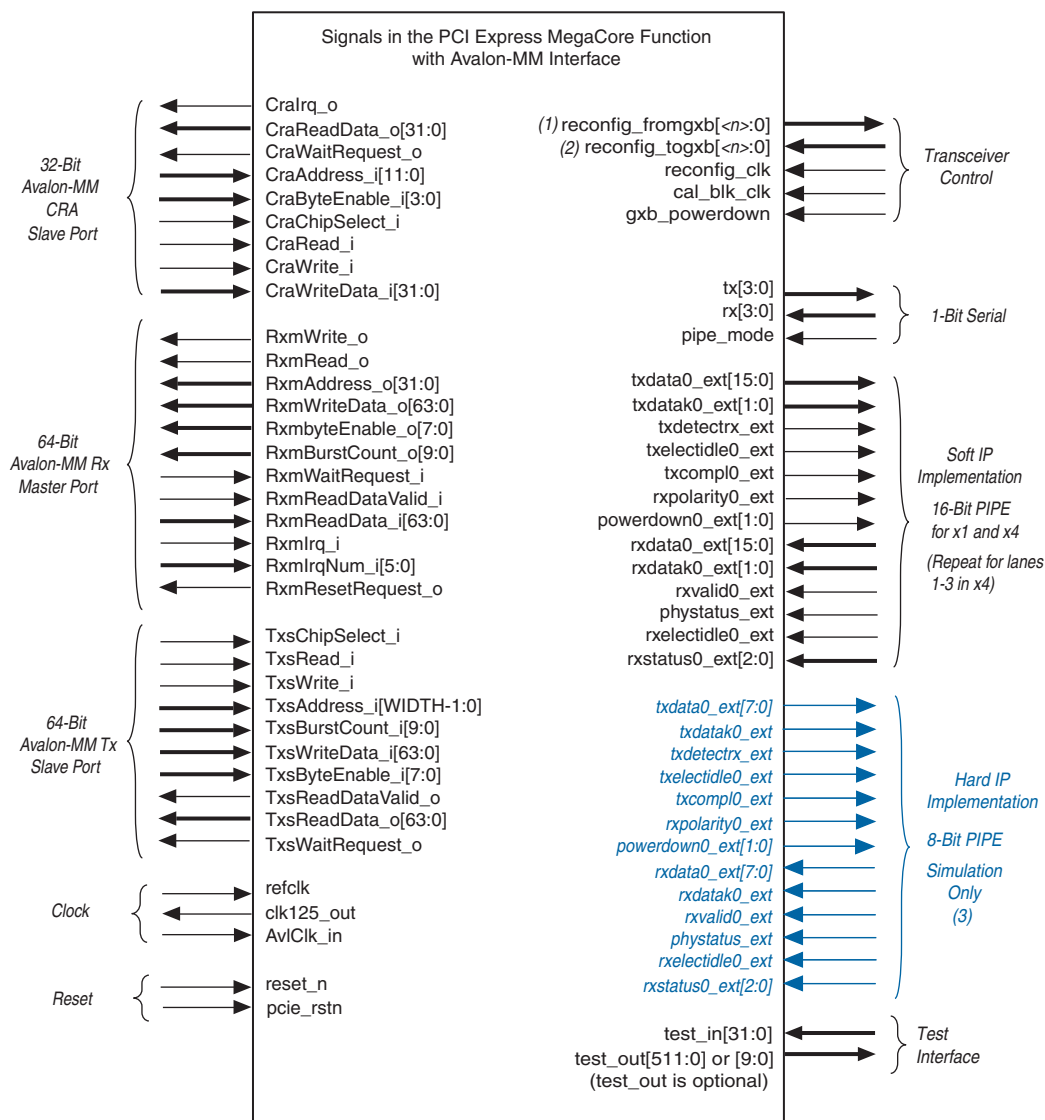
You can choose either the soft or hard IP implementation of PCI Express MegaCore function when using the SOPC Builder design flow. The hard IP implementation is available as an endpoint.

The following Avalon-MM busses are only available when using the PCI Express Compiler MegaCore function in the SOPC Builder design flow:

- 64-Bit Bursting Rx Avalon-MM Master Signals
- 64-Bit Bursting Tx Avalon-MM Slave Signals
- 32-Bit Non-bursting Avalon-MM CRA Slave Signals

Figure 5-56 shows all the signals of a full-featured PCI Express MegaCore function when used in the SOPC Builder design flow. Your parameterization may not include some of the ports. The Avalon-MM signals are shown on the left side of this figure.

Figure 5-56. PCI Express MegaCore Function with Avalon-MM Interface



Notes to Figure 5-56:

- (1) Available in Stratix II GX, Stratix IV GX, Arria GX, and Hardcopy IV GX devices. The reconfig_fromgxb is a single wire for Stratix II GX and Arria GX. For Stratix IV GX, <n> = 16 for x1 and x4 MegaCore functions and <n> = 33 the x8 MegaCore function.
- (2) Available in Stratix II GX, Stratix IV GX, Arria GX, and Hardcopy IV GX devices. For Stratix II GX and Arria GX reconfig_togxb, <n> = 2. For Stratix IV GX, <n> = 3.
- (3) Signals in blue are for simulation only.

Table 5–35 lists the interfaces for this MegaCore function with links to the sections that describe each.

Table 5–35. Signal Groups in the PCI Express Variant—Avalon-MM Interface

Signal Group	Description
Logical	
Avalon-MM CRA Slave	“32-Bit Non-bursting Avalon-MM CRA Slave Signals” on page 5–62
Avalon-MM Tx Master	“64-Bit Bursting Rx Avalon-MM Master Signals” on page 5–63
Avalon-MM Tx	“64-Bit Bursting Tx Avalon-MM Slave Signals” on page 5–63
Clock	“Clock Signals” on page 5–64
Reset	“Reset Signals” on page 5–64
Physical and Test	
Transceiver Control	“Transceiver Control” on page 5–66
Serial	“Serial Interface Signals” on page 5–67
Pipe	“PIPE Interface Signals” on page 5–68
Test	“Test Signals” on page 5–69

32-Bit Non-bursting Avalon-MM CRA Slave Signals

This optional port allows upstream PCI Express devices and external Avalon-MM masters to access internal control and status registers.

Table 5–36 describes the CRA slave ports.

Table 5–36. Avalon-MM CRA Slave Interface Signals

Signal	I/O	Type	Description
CraIrq_o	0	irq	Interrupt request. A port request for an Avalon-MM interrupt.
CraReadData_o[31:0]	0	Readdata	Read data lines
CraWaitRequest_o	0	Waitrequest	Wait request to hold off more requests
CraAddress_i[11:0]	1	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.
CraByteEnable_i[3:0]	1	Byteenable	Byte enable
CraChipSelect_i	1	Chipselect	Chip select signal to this slave
CraRead_i	1	Read	Read enable
CraWrite_i	1	Write	Write request
CraWriteData_i[31:0]	1	Writedata	Write data

64-Bit Bursting Rx Avalon-MM Master Signals

This Avalon-MM master port propagates PCI Express requests as bursting reads or writes to the system interconnect fabric. Table 5-37 lists the 64-bit Rx Master interface ports.

Table 5-37. Avalon-MM Rx Master Interface Signals

Signal	I/O	Description
RxmWrite_o	0	Asserted by the core to request a write to an Avalon-MM slave.
RxmRead_o	0	Asserted by the core to request a read.
RxmAddress_o[31:0]	0	The address of the Avalon-MM slave being accessed.
RxmWriteData_o[63:0]	0	Rx data being written to slave.
RxmByteEnable_o[7:0]	0	Byte enable for RxmWriteData_o.
RxmBurstCount_o[9:0]	0	The burst count, measured in QWORDS, of the Rx write or read request. The width indicates the maximum data, up to 4 KBytes, that can be requested.
RxmWaitRequest_i	1	Asserted by the external Avalon-MM slave to hold data transfer.
RxmReadData_i[63:0]	1	Read data returned from Avalon-MM slave in response to a read request. This data is sent to the core through the Tx interface.
RxmReadDataValid_i	1	Asserted by the system interconnect fabric to indicate that the read data on RxmReadData_i bus is valid.
RxmIrq_i	1	Indicates an interrupt request asserted from the system interconnect fabric. This signal is only available when the control register access port is enabled.
RxmIrqNum_i[5:0]	1	Indicates the ID of the interrupt request being asserted by RxmIrq_i. This signal is only available when the control register access port is enabled.

64-Bit Bursting Tx Avalon-MM Slave Signals

This optional Avalon-MM bursting slave port propagates requests from the system interconnect fabric to the PCI Express MegaCore function. Requests from the system interconnect fabric are translated into PCI Express request packets. Incoming write requests can be up to 4 KBytes in size; incoming read requests can be up to 512 bytes in size.

Table 5-38 lists the Tx slave interface ports.

Table 5-38. Avalon-MM Tx Slave Interface Signals (Part 1 of 2)

Signal	I/O	Description
TxsReadData_o[63:0]	0	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.
TxsReadDataValid_o	0	Asserted by the bridge to indicate that TxsReadData_o is valid.
TxsWaitRequest_o	0	Asserted by the bridge to hold off write data when running out of buffer space.
TxsChipSelect_i	1	The system interconnect fabric asserts this signal to select the Tx slave port.

Table 5-38. Avalon-MM Tx Slave Interface Signals (Part 2 of 2)

Signal	I/O	Description
TxsAddress_i[TXS_ADDR_WIDTH-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 TXS_ADDR_WIDTH value is determined when the system is created.
TxsBurstCount_i[9:0]	I	Asserted by the system interconnect fabric indicating the amount of data requested. This count is limited to 4 KBytes, the maximum data payload supported by the PCI Express protocol.
TxsByteEnable_i[7:0]	I	Write byte enable for TxWriteData_i bus.
TxsRead_i	I	Read request asserted by the system interconnect fabric to request a read.
TxsWrite_i	I	Read request asserted by the system interconnect fabric to request a write.
TxsWriteData_i[63:0]	I	Write data sent by the external Avalon-MM master to the Tx slave port.

Clock Signals

Table 5-39 describes the clock signals for the PCI Express MegaCore function generated in SOPC Builder.

Table 5-39. Avalon-MM Clock Signals

Signal	I/O	Description
refclk	I	An external clock source. When you turn on the Use separate clock option on the Avalon Configuration page, the PCI Express protocol layers are driven by an internal clock that is generated from refclk.
clk125_out	O	This clock is exported by the PCI Express MegaCore function. It can be used for logic outside of the MegaCore function. It is not visible to SOPC Builder and cannot be used to drive other Avalon-MM components in the system.
AvlClk_L	I	Avalon-MM global clock. AvlClk_L connects to clk which is the main clock source of the SOPC Builder system. clk is user-specified. It can be generated on the PCB or derived from other logic in the system.

Refer to “SOPC Builder Design Flow Clocking—Soft IP Implementation” on page 4-66 for a complete explanation of the clocking scheme.

Reset Signals

Figure 5-57 shows the PCI Express SOPC Builder reset scheme. When generated in the SOPC Builder design flow, the PCI Express MegaCore uses two reset inputs:

- Pcie_rstn
- reset_n

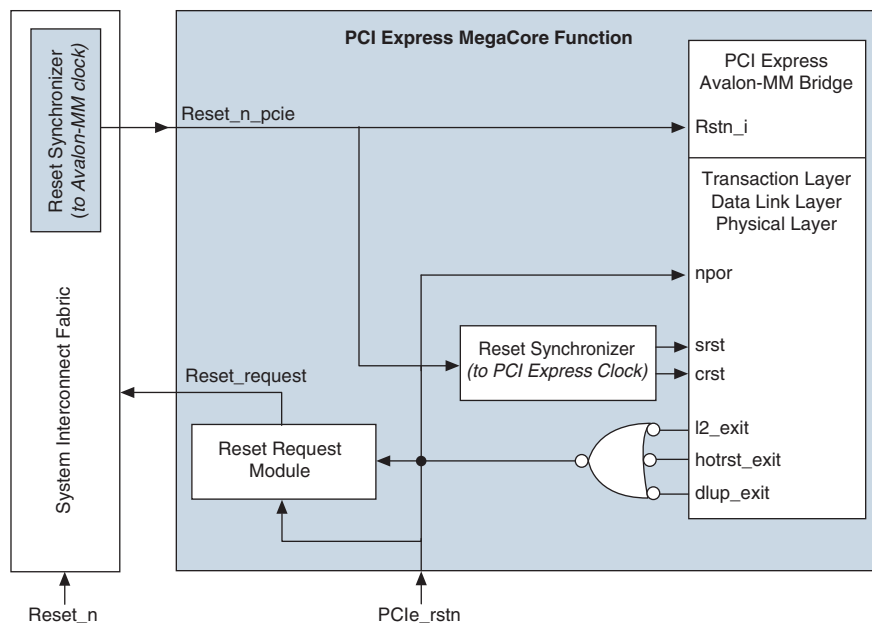
Pcie_rstn directly resets all sticky PCI Express MegaCore function configuration registers through the npor input. Sticky registers are those registers that fail to reset in L2 low power mode or upon a fundamental reset.

Pcie_rstn also resets the rest of the PCI Express MegaCore function, but only after the following synchronization process:

1. When Pcie_rstn asserts, the reset request module asserts reset_request, synchronized to the Avalon-MM clock, to the SOPC reset block.
2. The SOPC reset block sends a reset pulse, Reset_n_pcie, synchronized to the Avalon-MM clock, to the PCI Express Compiler MegaCore function.
3. The reset synchronizer resynchronizes Reset_n_pcie to the PCI Express clock to reset the PCI Express Avalon-MM bridge as well as the three PCI Express layers with srst and crst.
4. The reset_request signal deasserts after Reset_n_pcie asserts.

The system-wide reset, reset_n, resets all PCI Express MegaCore function circuitry not affected by Pcie_rstn. However, the SOPC Reset Block first intercepts the asynchronous reset_n, synchronizes it to the Avalon-MM clock, and sends a reset pulse, Reset_n_pcie to the PCI Express Compiler MegaCore function. The Reset Synchronizer resynchronizes Reset_n_pcie to the PCI Express clock to reset the PCI Express Avalon-MM bridge as well as the three PCI Express layers with srst and crst.

Figure 5-57. PCI Express SOPC Builder Reset Diagram



Note to figure

- (1) The system-wide reset, reset_n, indirectly resets all PCI Express MegaCore function circuitry not affected by Pcie_rstn using the Reset_n_pcie signal and the Reset Synchronizer module.

Physical Layer Interface Signals

This section describes the global PHY support signals which are only present on Arria GX, Arria II GX, Stratix II GX, or Stratix IV GX devices that use an integrated PHY. When selecting an integrated PHY, the MegaWizard Plug-In Manager generates a SERDES variation file, *<variation>_serdes.<v or vhd>*, in addition of the MegaCore function variation file, *<variation>.<v or vhd>*.



You can edit the SERDES variation file using the ALTGXB or ALT2GXB megafunction wizard respectively for the Arria GX, Arria II GX, Stratix II GX, or Stratix IV GX device families.

Transceiver Control

Table 5-40 describes the transceiver support signals.

Table 5-40. Transceiver Control Signals

Signal	I/O	Description
cal_blk_clk	I	The cal_blk_clk input signal is connected to the transceiver calibration block clock (cal_blk_clk) input. All instances of transceivers in the same device must have their cal_blk_clk inputs connected to the same signal because there is only one calibration block per device. This input should be connected to a clock operating as recommended by the <i>The Stratix II GX Transceiver User Guide</i> or the <i>Stratix IV Transceiver Architecture</i> or the <i>Arria II GX Transceiver User Guide</i> .
gxb_powerdown	I	The gxb_powerdown signal connects to the transceiver calibration block gxb_powerdown input. This input should be connected as recommended by the <i>Stratix II GX Device Handbook</i> or volume 2 of the <i>Stratix IV Device Handbook</i> . When the calibration clock is not used, this input must be tied to ground.
reconfig_fromgxb[16:0] (Stratix IV GX x1 and x4)	0	These are the transceiver dynamic reconfiguration signals. Transceiver dynamic reconfiguration is not typically required for PCI Express designs in Stratix II GX or Arria GX devices. These signals may be used for cases in which the PCI Express instance shares a transceiver quad with another protocol that supports dynamic reconfiguration. They may also be used in cases where the transceiver analog controls (V_{OD} , Pre-emphasis, and Manual Equalization) need to be modified to compensate for extended PCI Express interconnects such as cables. In these cases, these signals must be connected as described in the <i>Stratix II GX Device Handbook</i> ; otherwise, when unused, the reconfig_clk signal should be tied low, reconfig_togxb tied to b'010 and reconfig_fromgxb left open.
reconfig_fromgxb[33:0] (Stratix IV GX x8)	0	
reconfig_fromgxb (Stratix II GX, Arria GX)	I	
reconfig_togxb[3:0] (Stratix IV GX)	I	
reconfig_togxb[2:0] (Stratix II GX, Arria GX)	I	
reconfig_clk		For Stratix IV GX devices dynamic reconfiguration is required for PCI Express designs to compensate for variations due to process, voltage and temperature. You must connect the ALTGX_RECONFIG instance to the ALTGX instances with receiver channels, in your design using these signals. The maximum frequency of reconfig_clk is 50 MHz.

These input signals from the user application connect directly to the transceiver instance.

Table 5-41. Transceiver Control Signal Use

Signal	Arria GX	Arria II GX	Stratix II GX	Stratix IV GX
cal_blk_clk	Yes	Yes	Yes	Yes
reconfig_clk	Non-functional	Yes	Yes	Yes
reconfig_togxb	Non-functional	Yes	Yes	Yes
reconfig_fromgxb	Non-functional	Yes	Yes	Yes



For more information refer to the *Stratix II GX ALT2GXB_RECONFIG Megafunction User Guide* or the *Transceiver Configuration Guide* in volume 3 of the *Stratix IV Device Handbook*, as appropriate.

The following sections describe signals for the three possible types of physical interfaces (1-bit, 20-bit, or PIPE). Refer to [Figure 5-1 on page 5-2](#), [Figure 5-2 on page 5-3](#), [Figure 5-3 on page 5-4](#), [Figure 5-35 on page 5-36](#), and [Figure 5-56 on page 5-61](#) for pinout diagrams of all of the PCI Express MegaCore function variants.

Serial Interface Signals

[Table 5-42](#) describes the serial interface signals. These signals are available if you use the Arria GX PHY, Arria II GX PHY, Stratix GX PHY, Stratix II GX PHY, or the Stratix IV GX PHY.

Table 5-42. 1-Bit Interface Signals

Signal	I/O	Description
tx_out<0:7> (1)	0	Transmit input. These signals are the serial outputs of lane 0-7.
rx_in<0:7> (1)	1	Receive input. These signals are the serial inputs of lane 0-7.
pipe_mode	1	pipe_mode selects whether the MegaCore function uses the PIPE interface or the 1-bit interface. Setting pipe_mode to a 1 selects the PIPE interface, setting it to 0 selects the 1-bit interface. When simulating, you can set this signal to indicate which interface is used for the simulation. When compiling your design for an Altera device, set this signal to 0.

Note to Table 5-42:

(1) The ×1 MegaCore function only has lane 0. The ×4 MegaCore function only has lanes 0-3.

For the soft IP implementation of the ×1 MegaCore function any channel of any transceiver block can be assigned for the serial input and output signals. For the hard IP implementation of the ×1 MegaCore function the serial input and output signals must use channel 0 of the Master Transceiver Block associated with that hard IP block.

For the ×4 MegaCore Function the serial inputs (rx_in[0-3]) and serial outputs (tx_out[0-3]) must be assigned to the pins associated with the like-number channels of the transceiver block. The signals rx_in[0]/tx_out[0] must be assigned to the pins associated with channel 0 of the transceiver block, rx_in[1]/tx_out[1] must be assigned to the pins associated with channel 1 of the transceiver block, and so on. Additionally, the ×4 hard IP implementation must use the four channels of the Master Transceiver Block associated with that hard IP block.

For the ×8 MegaCore Function the serial inputs ($rx_in[0-3]$) and serial outputs ($tx_out[0-3]$) must be assigned to the pins associated with the like-number channels of the Master Transceiver Block. The signals $rx_in[0]/tx_out[0]$ must be assigned to the pins associated with channel 0 of the Master Transceiver Block, $rx_in[1]/tx_out[1]$ must be assigned to the pins associated with channel 1 of the Master Transceiver Block, and so on. The serial inputs ($rx_in[4-7]$) and serial outputs ($tx_out[4-7]$) must be assigned in order to the pins associated with channels 0-3 of the Slave Transceiver Block. The signals $rx_in[4]/tx_out[4]$ must be assigned to the pins associated with channel 0 of the Slave Transceiver Block, $rx_in[5]/tx_out[5]$ must be assigned to the pins associated with channel 1 of the Slave Transceiver Block, and so on.



Refer to Volume 2 of the *Arria GX Device Handbook*, Volume 2 of *Arria II GX Device Handbook*, the *Stratix GX Transceiver User Guide*, the *Stratix II GX Transceiver User Guide*, or Volume 2 of the *Stratix IV Device Handbook*, for more information about the transceiver blocks.

PIPE Interface Signals

The ×1 and ×4 soft IP implementation of the MegaCore function is compliant with the 16-bit version of the PIPE interface, enabling use of an external PHY. The ×8 soft IP implementation of the MegaCore function is compliant with the 8-bit version of the PIPE interface. These signals are available even when you select the **Stratix IV GX**, **Arria GX**, **Stratix GX**, or **Stratix II GX** PHY so that you can simulate using both the one-bit and the PIPE interface. Typically, simulation is much faster using the PIPE interface. For hard IP implementations, the 8-bit PIPE interface is also available for simulation purposes. However, it is not possible to use the hard IP PIPE interface in an actual device. Table 5-43 describes the PIPE interface signals used for a standard 16-bit SDR or 8-bit SDR interface. These interfaces are used for simulation of the PIPE interface for variations using an internal transceiver. In Table 5-43, signals that include lane number 0 also exist for lanes 1-7, as marked in the table. Refer to Chapter 6, External PHYs for descriptions of the slightly modified PIPE interface signalling for use with specific external PHYs. The modifications include DDR signalling and source synchronous clocking in the Tx direction.

Table 5-43. PIPE Interface Signals (Part 1 of 3)

Signal	I/O	Description
$txdata<n>_ext[15:0]$ (1)	0	Transmit data <n> (2 symbols on lane <n>). This bus transmits data on lane <n>. The first transmitted symbol is $txdata_ext[7:0]$ and the second transmitted symbol is $txdata0_ext[15:8]$. For the 8-bit PIPE mode only $txdata<n>_ext[7:0]$ is available.
$txdatak<n>_ext[1:0]$ (1)	0	Transmit data control <n> (2 symbols on lane <n>). This signal serves as the control bit for $txdata<n>_ext$; $txdatak<n>_ext[0]$ for the first transmitted symbol and $txdatak<n>_ext[1]$ for the second (8B10B encoding). For 8-bit PIPE mode only the single bit signal $txdatak<n>_ext$ is available.
$txdetectrx<n>_ext$ (1)	0	Transmit detect receive <n>. This signal is used to tell the PHY layer to start a receive detection operation or to begin loopback.
$txelecidle<n>_ext$ (1)	0	Transmit electrical idle 0. This signal forces the transmit output to electrical idle.
$txcompl<n>_ext$ (1)	0	Transmit compliance 0. This signal forces the running disparity to negative in compliance mode (negative COM character).

Table 5-43. PIPE Interface Signals (Part 2 of 3)

Signal	I/O	Description
rxpolarity<n>_ext (1)	0	Receive polarity 0. This signal instructs the PHY layer to do a polarity inversion on the 8B10B receiver decoding block.
powerdown<n>_ext [1:0] (1)	0	Power down 0. This signal requests the PHY to change its power state to the specified state (P0, P0s, P1, or P2).
tx_pipemargin	0	This option creates a tx_pipemargin [2:0] input port which is available only in PCI Express (PIPE) Gen2 (5 Gbps) mode. It selects the transmitter VOD level. The following encodings are used: <ul style="list-style-type: none"> 3'b000: Normal operating range 3'b001: Full swing: 800–1200 mV, Half swing: 400–700 mV 3'b010: To be determined 3'b011: To be determined 3'b100: Full swing: 200–400 mV, Half swing: 100–200 mV 3'b101: Full swing: 200–400 mV, Half swing: 100–200 mV 3'b110: Full swing: 200–400 mV, Half swing: 100–200 mV 3'b111: Full swing: 200–400 mV, Half swing: 100–200 mV
tx_pipedeeemph	0	This option creates a tx_pipedeeemph input port which is available only in PCI Express (PIPE) Gen2 (5 Gbps) mode. It selects the transmitter de-emphasis: <ul style="list-style-type: none"> 1'b0: -6 dB 1'b1: -3.5 dB
rxdata<n>_ext [15:0] (1)	I	Receive data <n> (2 symbols on lane <n>). This bus receives data on lane <n>. The first received symbol is rxdata<n>_ext [7:0] and the second is rxdata<n>_ext [15:8]. For the 8 Bit PIPE mode only rxdata<n>_ext [7:0] is available.
rxdatak<n>_ext [1:0] (1)	I	Receive data control <n> (2 symbols on lane <n>). This signal is used for separating control and data symbols. The first symbol received is aligned with rxdatak<n>_ext [0] and the second symbol received is aligned with rxdatak<n>_ext [1]. For the 8 Bit PIPE mode only the single bit signal rxdatak<n>_ext is available.
rxvalid<n>_ext (1)	I	Receive valid <n>. This symbol indicates symbol lock and valid data on rxdata<n>_ext and rxdatak<n>_ext.
phystatus<n>_ext (1)	I	PHY status <n>. This signal is used to communicate completion of several PHY requests.
rxlecidle<n>_ext (1)	I	Receive electrical idle 0. This signal forces the receive output to electrical idle.
rxstatus<n>_ext [2:0] (1)	I	Receive status 0: This signal encodes receive status and error codes for the receive data stream and receiver detection.
pipe_rstn	0	Asynchronous reset to external PHY. This signal is tied high and expects a pull-down resistor on the board. During FPGA configuration, the pull-down resistor resets the PHY and after that the FPGA drives the PHY out of reset. This signal is only on MegaCore functions configured for the external PHY.
pipe_txclk	0	Transmit datapath clock to external PHY. This clock is derived from refclk and it provides the source synchronous clock for the transmit data of the PHY.
rate_ext	0	When asserted, indicates the interface is operating at the 5.0 Gbps rate. This signal is available for simulation purposes only in the hard IP implementation.

Table 5-43. PIPE Interface Signals (Part 3 of 3)

Signal	I/O	Description
pipe_mode	I	

Note to Table 5-43:

(1) where <n> is the lane number ranging from 0-7

Test Signals

The test_in and test_out busses provide run-time control and monitoring of the internal state of the MegaCore functions. Table 5-44 describes the test signals for the hard IP implementation. Appendix B, [Test Port Interface Signals](#) for a complete description of the individual bits in these busses.



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 signals have not been rigorously verified and will not function as documented in some corner cases.

Table 5-44. Test Interface Signals—Hard IP Implementation (Part 1 of 2)

Signal	I/O	Description
test_in[39:0] (hard IP)	I	The test_in bus provides runtime control for specific MegaCore function features as well as an error injection capability. It also contains controls for selecting signal groups to be monitored on test_out. Refer to Appendix B, Test Port Interface Signals for a complete description. For normal operation, this bus can be driven to all 0's.
test_out[64:0] or [8:0]	O	The test_out bus provides extensive monitoring of the internal state of the MegaCore function. You can select which signal groups to monitor using test_in[15:8]. Refer to Appendix B, Test Port Interface Signals for a complete description. When you choose the 9-bit test_out bus width, a subset of the 64-bit test bus is brought out as follows: <ul style="list-style-type: none"> bits [8:5] = test_out [28:25] bits [4:0] = test_out [4:0]

Table 5-44. Test Interface Signals—Hard IP Implementation (Part 2 of 2)

Signal	I/O	Description
ltssm[4:0]	0	<p>LTSSM state: LTSSM state encoding:</p> <ul style="list-style-type: none"> ■ 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: I/Os ■ 10110: L1.entry ■ 10111: L1.idle ■ 11000: L2.idle ■ 11001: L2.transmit.wake
lane_act[3:0]	0	Indicates the number of initialized lanes using a decimal value.

Table 5–45 describes the test signals for the soft IP implementation.

Table 5–45. Test Interface Signals—Soft IP Implementation

Signal	I/O	Description
test_in[31:0]	I	The test_in bus provides runtime control for specific MegaCore function features as well as error injection capability. Refer to Appendix B, Test Port Interface Signals for a complete description. For normal operation, this bus can be driven to all 0's.
test_out[511:0] or [8:0] for ×1 or ×4 test_out[127:0] or [8:0] for ×8		<p>The test_out bus provides extensive monitoring of the internal state of the MegaCore function. Refer to Appendix , Soft IP ×8 MegaCore Function test_out for a complete description of the ×8 MegaCore function debug signals. Refer to Appendix , Soft IP MegaCore Function for a complete description of the ×4 MegaCore function debug signals.</p> <p>When you choose the 9-bit test_out bus width, a subset of the test_out signals are brought out as follows:</p> <ul style="list-style-type: none"> bits[8:5] = lane_act debug signals. These signals corresponds to test_out[91:88] on the ×8 MegaCore function and test_out[411:408] on the ×4/×1 MegaCore function. bits[4:0] = ltssm_r debug signal. These signals corresponds to test_out[4:0] on the ×8 MegaCore function and test_out[324:320] on the ×4/×1 MegaCore function.

External PHY Support

This chapter discusses external PHY support, which includes the external PHYs and interface modes shown in [Table 6–1](#). The external PHY is not applicable to the hard IP implementation.

Table 6–1. External PHY Interface Modes


PHY Interface Mode	Clock Frequency	Notes
16-bit SDR	125 MHz	In this the generic 16-bit PIPE interface, both the <code>Tx</code> and <code>Rx</code> data are clocked by the <code>refclk</code> input which is the <code>pclk</code> from the PHY.
16-bit SDR mode (with source synchronous transmit clock)	125 MHz	This enhancement to the generic PIPE interface adds a <code>TxC1k</code> to clock the <code>TxData</code> source synchronously to the external PHY. The TIXIO1100 PHY uses this mode.
8-bit DDR	125 MHz	This double data rate version saves I/O pins without increasing the clock frequency. It uses a single <code>refclk</code> input (which is the <code>pclk</code> from the PHY) for clocking data in both directions.
8-bit DDR mode (with 8-bit DDR source synchronous transmit clock)	125 MHz	This double data rate version saves I/O pins without increasing the clock frequency. A <code>TxC1k</code> clocks the data source synchronously in the transmit direction.
8-bit DDR/SDR mode (with 8-bit DDR source synchronous transmit clock)	125 MHz	This is the same mode as 8-bit DDR mode except the control signals <code>rxlecidle</code> , <code>rxstatus</code> , <code>phystatus</code> , and <code>rxvalid</code> are latched using the SDR I/O register rather than the DDR I/O register. The TIXIO1100 PHY uses this mode.
8-bit SDR	250 MHz	This is the generic 8-bit PIPE interface. Both the <code>Tx</code> and <code>Rx</code> data are clocked by the <code>refclk</code> input which is the <code>pclk</code> from the PHY. The NXP PX1011A PHY uses this mode.
8-bit SDR mode (with Source Synchronous Transmit Clock)	250 MHz	This enhancement to the generic PIPE interface adds a <code>TxC1k</code> to clock the <code>TxData</code> source synchronously to the external PHY.

When an external PHY is selected, additional logic required to connect directly to the external PHY is included in the `<variation name>` module or entity.

The user logic must instantiate this module or entity in the design. The implementation details for each of these modes are discussed in the following sections.

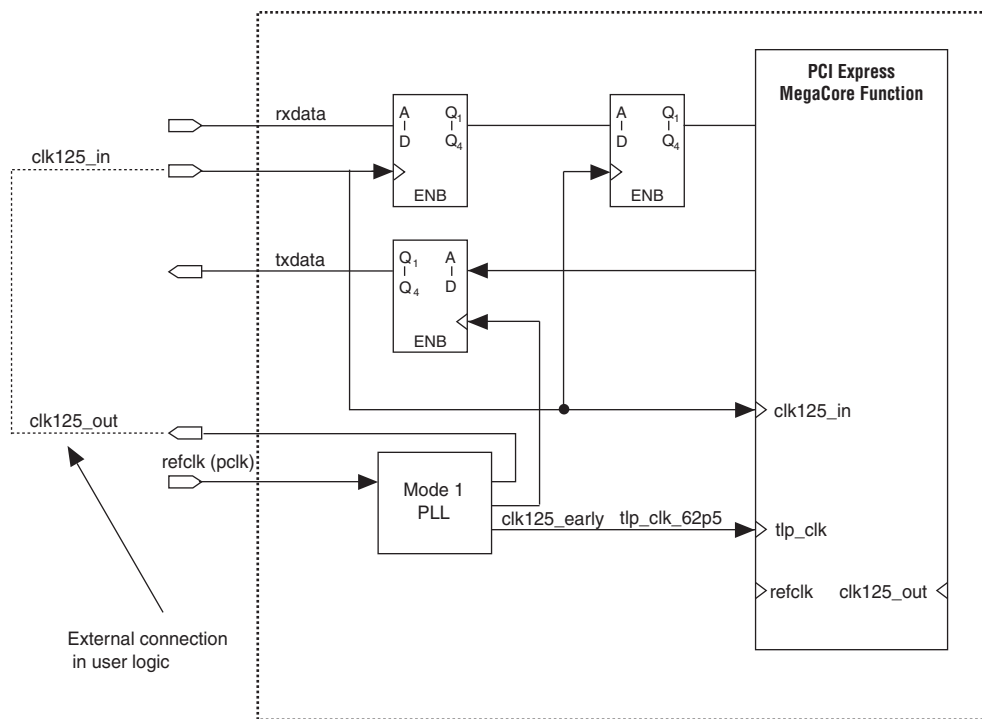
16-bit SDR Mode

The implementation of this 16-bit SDR mode PHY support is shown in [Figure 6–1](#) and is included in the file `<variation name>.v` or `<variation name>.vhd` and includes a PLL. The PLL inclock is driven by `refclk` and has the following outputs:

 The `refclk` is the same as `pclk`, the parallel clock provided by the external PHY. This document uses the terms `refclk` and `pclk` interchangeably.

- `clk125_out` is a 125 MHz output that has the same phase-offset as `refclk`. The `clk125_out` must drive the `clk125_in` input in the user logic as shown in the [Figure 6-1](#). The `clk125_in` is used to capture the incoming receive data and also is used to drive the `clk125_in` input of the MegaCore function.
- `clk125_early` is a 125 MHz output that is phase shifted. This phase-shifted output clocks the output registers of the transmit data. Based on your board delays, you may need to adjust the phase-shift of this output. To alter the phase shift, copy the PLL source file referenced in your variation file from the `<path>/ip/PCI Express Compiler/lib` directory, where `<path>` is the directory in which you installed the PCI Express Compiler, to your project directory. Then use the MegaWizard Plug In Manager in the Quartus II software to edit the PLL source file to set the required phase shift. Then add the modified PLL source file to your Quartus II project.
- `tlp_clk62p5` is a 62.5 MHz output that drives the `tlp_clk` input of the MegaCore function when the MegaCore internal clock frequency is 62.5 MHz.

Figure 6-1. 16-bit SDR Mode - 125 MHz without Transmit Clock



16-bit SDR Mode with a Source Synchronous TxClk

The implementation of the 16-bit SDR mode with a source synchronous `TxC1k` is shown in [Figure 6-2](#) and is included in the file `<variation name>.v` or `<variation name>.vhd`. In this mode the following clocking scheme is used:

- `refclk` is used as the `clk125_in` for the core
- `refclk` clocks a single data rate register for the incoming receive data

- This is the only external PHY mode that does not require a PLL. However, if the slow `tlp_clk` feature is used with this PIPE interface mode, then a PLL is required to create the slow `tlp_clk`. In the case of the slow `tlp_clk`, the circuit is similar to the one shown previously in [Figure 6-1](#), the 16-bit SDR, but with `TxC1k` output added.

[illegible]

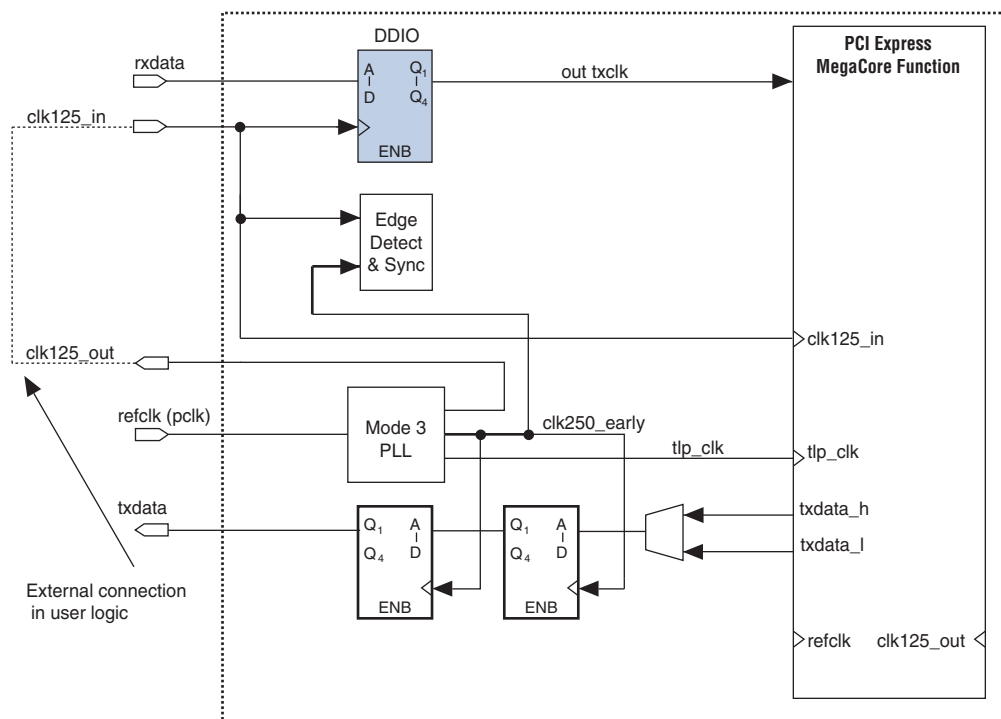
- A zero delay copy of the 125 MHz `refclk`. The zero delay PLL output is used as the `clk125_in` for the core and clocks a double data rate register for the incoming receive data.
- A 250 MHz early output. This is multiplied from the 125 MHz `refclk` is early in relation to the `refclk`. Use the 250 MHz early clock PLL output to clock an 8-bit SDR transmit data output register. A 250 MHz single data rate register is used for the 125 MHz DDR output because this allows the use of the SDR output registers in the Cyclone II I/O block. The early clock is required to meet the required clock to out times for the common `refclk` for the PHY. You may need to adjust the phase shift for your specific PHY and board delays. To alter the phase shift, copy

the PLL source file referenced in your variation file from the `<path>/ip/PCI Express Compiler/lib` directory, where `<path>` is the directory in which you installed the PCI Express Compiler, to your project directory. Then use the MegaWizard Plug In Manager to edit the PLL source file to set the required phase shift. Then add the modified PLL source file to your Quartus II project.

- An optional 62.5 MHz TLP Slow clock is provided for ×1 implementations.

An edge detect circuit detects the relationships between the 125 MHz clock and the 250 MHz rising edge to properly sequence the 16-bit data into the 8-bit output register.

Figure 6-3. 8-Bit DDR Mode without Transmit Clock



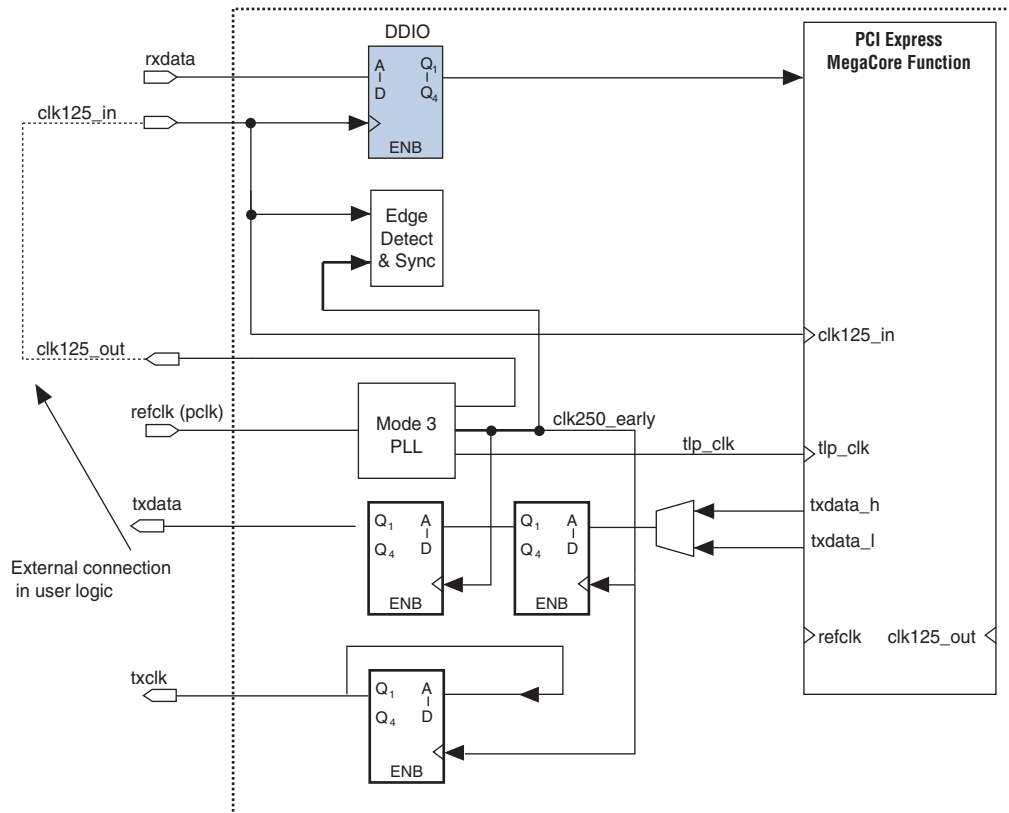
8-bit DDR with a Source Synchronous TxClk

Figure 6-4 shows the implementation of the 8-bit DDR mode with a source synchronous transmit clock (TxClk). It is included in the file `<variation name>.v` or `<variation name>.vhd` and includes a PLL. `refclk` (pclk from the external PHY) drives the PLL inclock. The PLL inclock has the following outputs:

- A zero delay copy of the 125 MHz `refclk` used as the `clk125_in` for the MegaCore function and also to clock DDR input registers for the Rx data and status signals.
- A 250 MHz early clock. This PLL output clocks an 8-bit SDR transmit data output register. It is multiplied from the 125 MHz `refclk` and is early in relation to the `refclk`. A 250 MHz single data rate register for the 125 MHz DDR output allows you to use the SDR output registers in the Cyclone II I/O block.
- An optional 62.5 MHz TLP Slow clock is provided for ×1 implementations.

An edge detect circuit detects the relationships between the 125 MHz clock and the 250 MHz rising edge to properly sequence the 16-bit data into the 8-bit output register.

Figure 6-4. 8-bit DDR Mode with a Source Synchronous Transmit Clock



8-bit SDR Mode

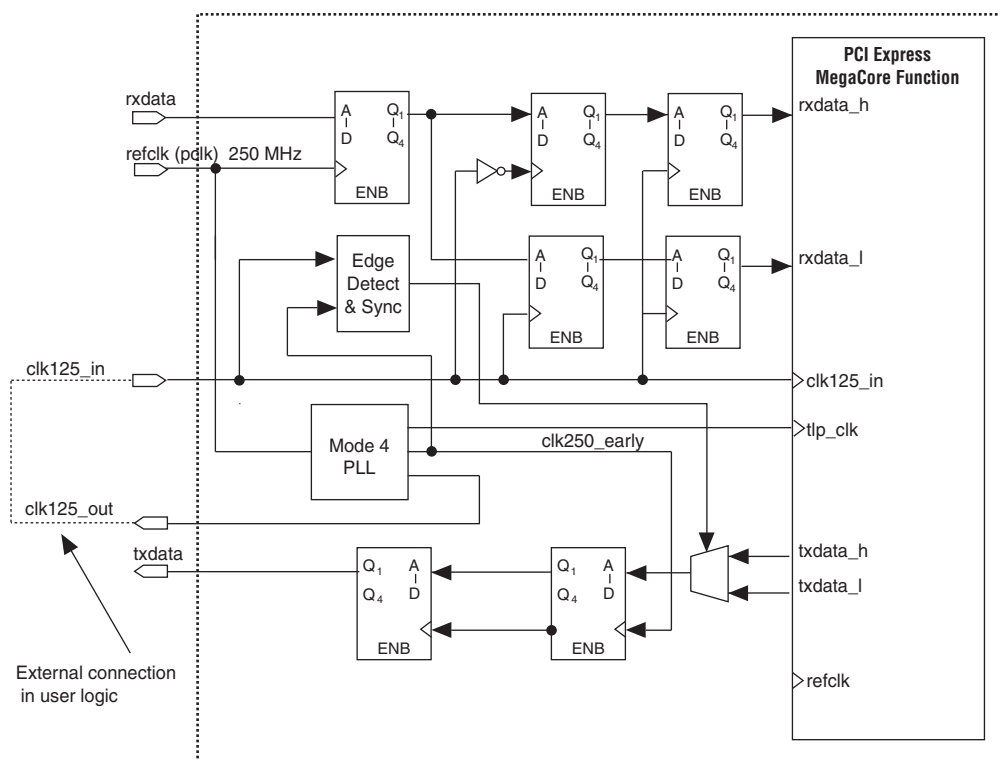
Figure 6-5 illustrates the implementation of the 8-bit SDR mode. This mode is included in the file `<variation name>.v` or `<variation name>.vhd` and includes a PLL. `refclk` (`pclk` from the external PHY) drives the PLL inclock. The PLL has the following outputs:

- A 125 MHz output derived from the 250 MHz `refclk` used as the `clk125_in` for the core and also to transition the incoming 8-bit data into a 16-bit register for the rest of the logic.
- A 250 MHz early output that is skewed early in relation to the `refclk` that is used to clock an 8-bit SDR transmit data output register. The early clock PLL output clocks the transmit data output register. The early clock is required to meet the specified clock-to-out times for the common clock. You may need to adjust the phase shift for your specific PHY and board delays. To alter the phase shift, copy the PLL source file referenced in your variation file from the `<path>/ip/PCI Express Compiler/lib` directory, where `<path>` is the directory in which you installed the PCI Express Compiler, to your project directory. Then use the MegaWizard Plug-In Manager in the Quartus II software to edit the PLL source file to set the required phase shift. Then add the modified PLL source file to your Quartus II project.

- An optional 62.5 MHz TLP Slow clock is provided for ×1 implementations.

An edge detect circuit detects the relationships between the 125 MHz clock and the 250 MHz rising edge to properly sequence the 16-bit data into the 8-bit output register.

Figure 6-5. 8-bit SDR Mode - 250 MHz



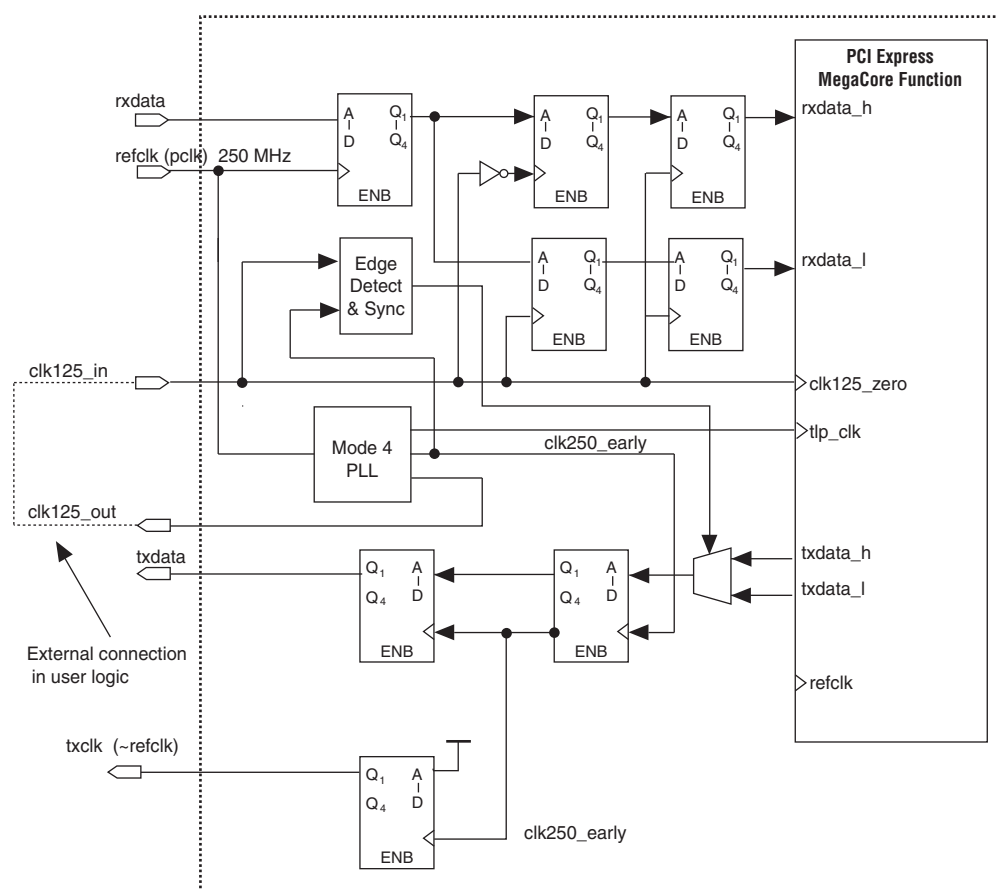
8-bit SDR with a Source Synchronous TxClk

Figure 6-6 illustrates the implementation of the 16-bit SDR mode with a source synchronous TxClk. It is included in the file *<variation name>.v* or *<variation name>.vhd* and includes a PLL. `refclk` (`pclk` from the external PHY) drives the PLL inlock. The PLL has the following outputs:

- A 125 MHz output derived from the 250 MHz `refclk`. This 125 MHz PLL output is used as the `clk125_in` for the MegaCore function.
- A 250 MHz early output that is skewed early in relation to the `refclk` the 250 MHz early clock PLL output clocks an 8-bit SDR transmit data output register.
- An optional 62.5 MHz TLP Slow clock is provided for ×1 implementations.

An edge detect circuit detects the relationships between the 125 MHz clock and the 250 MHz rising edge to properly sequence the 16-bit data into the 8-bit output register.

Figure 6-6. 8-bit SDR Mode with 250 MHz Source Synchronous Transmit Clock



16-bit PHY Interface Signals

Table 6-2 summarizes the external I/O signals for the 16-bit PIPE interface modes. Depending on the number of lanes selected and whether the PHY mode has a TxClk, some of the signals may not be available as noted.

Table 6-2. 16-bit PHY Interface Signals (Part 1 of 3)

Signal Name	Direction	Description	Availability
pcie_rstn	I	PCI Express reset signal, active low.	Always
phystatus_ext	I	PIPE interface phystatus signal. Signals the completion of the requested operation	Always
powerdown_ext [1:0]	O	PIPE interface powerdown signal. Used to request that the PHY enter the specified power state.	Always
refclk	I	Input clock connected to the PIPE interface pclk signal from the PHY. 125 MHz clock that clocks all of the status and data signals.	Always
pipe_txclk	O	Source synchronous transmit clock signal for clocking Tx Data and Control signals going to the PHY.	Only in modes that have the TxClk
rxdata0_ext [15:0]	I	Pipe interface lane 0 Rx data signals, carries the parallel received data.	Always

Table 6-2. 16-bit PHY Interface Signals (Part 2 of 3)

Signal Name	Direction	Description	Availability
rxdatak0_ext[1:0]	I	Pipe interface lane 0 Rx data K-character flags.	Always
rxelecidle0_ext	I	Pipe interface lane 0 Rx electrical idle indication.	Always
rxpolarity0_ext	0	Pipe interface lane 0 Rx polarity inversion control.	Always
rxstatus0_ext[1:0]	I	Pipe interface lane 0 Rx status flags.	Always
rxvalid0_ext	I	Pipe interface lane 0 Rx valid indication.	Always
txcomp10_ext	0	Pipe interface lane 0 Tx compliance control.	Always
txdata0_ext[15:0]	0	Pipe interface lane 0 Tx data signals, carries the parallel transmit data.	Always
txdatak0_ext[1:0]	0	Pipe interface lane 0 Tx data K-character flags.	Always
txelecidle0_ext	0	Pipe interface lane 0 Tx electrical Idle Control.	Always
rxdata1_ext[15:0]	I	Pipe interface lane 1 Rx data signals, carries the parallel received data.	Only in x4
rxdatak1_ext[1:0]	I	Pipe interface lane 1 Rx data K-character flags.	Only in x4
rxelecidle1_ext	I	Pipe interface lane 1 Rx electrical idle indication.	Only in x4
rxpolarity1_ext	0	Pipe interface lane 1 Rx polarity inversion control.	Only in x4
rxstatus1_ext[1:0]	I	Pipe interface lane 1 Rx status flags.	Only in x4
rxvalid1_ext	I	Pipe interface lane 1 Rx valid indication.	Only in x4
txcomp11_ext	0	Pipe interface lane 1 Tx compliance control.	Only in x4
txdata1_ext[15:0]	0	Pipe interface lane 1 Tx data signals, carries the parallel transmit data.	Only in x4
txdatak1_ext[1:0]	0	Pipe interface lane 1 Tx data K-character flags.	Only in x4
txelecidle1_ext	0	Pipe interface lane 1 Tx electrical idle control.	Only in x4
rxdata2_ext[15:0]	I	Pipe interface lane 2 Rx data signals, carries the parallel received data.	Only in x4
rxdatak2_ext[1:0]	I	Pipe interface lane 2 Rx data K-character flags.	Only in x4
rxelecidle2_ext	I	Pipe interface lane 2 Rx electrical idle indication.	Only in x4
rxpolarity2_ext	0	Pipe interface lane 2 Rx polarity inversion control.	Only in x4
rxstatus2_ext[1:0]	I	Pipe interface lane 2 Rx status flags.	Only in x4
rxvalid2_ext	I	Pipe interface lane 2 Rx valid indication.	Only in x4
txcomp12_ext	0	Pipe interface lane 2 Tx compliance control.	Only in x4
txdata2_ext[15:0]	0	Pipe interface lane 2 Tx data signals, carries the parallel transmit data.	Only in x4
txdatak2_ext[1:0]	0	Pipe interface lane 2 Tx data K-character flags.	Only in x4
txelecidle2_ext	0	Pipe interface lane 2 Tx electrical idle control.	Only in x4
rxdata3_ext[15:0]	I	Pipe interface lane 3 Rx data signals, carries the parallel received data.	Only in x4
rxdatak3_ext[1:0]	I	Pipe interface lane 3 Rx data K-character flags.	Only in x4
rxelecidle3_ext	I	Pipe interface lane 3 Rx electrical idle indication.	Only in x4
rxpolarity3_ext	0	Pipe interface lane 3 Rx polarity inversion control.	Only in x4
rxstatus3_ext[1:0]	I	Pipe interface lane 3 Rx status flags.	Only in x4
rxvalid3_ext	I	Pipe interface lane 3 Rx valid indication.	Only in x4

Table 6-2. 16-bit PHY Interface Signals (Part 3 of 3)

Signal Name	Direction	Description	Availability
txcompl3_ext	0	Pipe interface lane 3 Tx compliance control.	Only in x4
txdata3_ext [15:0]	0	Pipe interface lane 3 Tx data signals, carries the parallel transmit data.	Only in x4
txdatak3_ext [1:0]	0	Pipe interface lane 3 Tx data K-character flags.	Only in x4
txelecidle3_ext	0	Pipe interface lane 3 Tx electrical Idle Control.	Only in x4

8-bit PHY Interface Signals

Table 6-3 summarizes the external I/O signals for the 8-bit PIPE interface modes. Depending on the number of lanes selected and whether the PHY mode has a TxClk, some of the signals may not be available as noted.

Table 6-3. 8-bit PHY Interface Signals (Part 1 of 2)

Signal Name	Direction	Description	Availability
pcie_rstn	I	PCI Express reset signal, active low.	Always
phystatus_ext	I	PIPE interface phystatus signal. Signals the completion of the requested operation.	Always
powerdown_ext [1:0]	0	PIPE interface powerdown signal, Used to request that the PHY enter the specified power state.	Always
refclk	I	Input clock connected to the PIPE interface pclk signal from the PHY. Clocks all of the status and data signals. Depending on whether this is an SDR or DDR interface this clock will be either 250 MHz or 125 MHz.	Always
pipe_txclk	0	Source synchronous transmit clock signal for clocking Tx data and control signals going to the PHY.	Only in modes that have the TxClk
rxdata0_ext [7:0]	I	Pipe interface lane 0 Rx data signals, carries the parallel received data.	Always
rxdatak0_ext	I	Pipe interface lane 0 Rx data K-character flag.	Always
rxelecidle0_ext	I	Pipe interface lane 0 Rx electrical idle indication.	Always
rxpolarity0_ext	0	Pipe interface lane 0 Rx polarity inversion control.	Always
rxstatus0_ext [1:0]	I	Pipe interface lane 0 Rx status flags.	Always
rxvalid0_ext	I	Pipe interface lane 0 Rx valid indication.	Always
txcompl0_ext	0	Pipe interface lane 0 Tx compliance control.	Always
txdata0_ext [7:0]	0	Pipe interface lane 0 Tx data signals, carries the parallel transmit data.	Always
txdatak0_ext	0	Pipe interface lane 0 Tx data K-character flag.	Always
txelecidle0_ext	0	Pipe interface lane 0 Tx electrical idle control.	Always
rxdata1_ext [7:0]	I	Pipe interface lane 1 Rx data signals, carries the parallel received data.	Only in x4
rxdatak1_ext	I	Pipe interface lane 1 Rx data K-character flag.	Only in x4
rxelecidle1_ext	I	Pipe interface lane 1 Rx electrical idle indication.	Only in x4
rxpolarity1_ext	0	Pipe interface lane 1 Rx polarity inversion control.	Only in x4
rxstatus1_ext [1:0]	I	Pipe interface lane 1 Rx status flags.	Only in x4

Table 6-3. 8-bit PHY Interface Signals (Part 2 of 2)

Signal Name	Direction	Description	Availability
rxvalid1_ext	I	Pipe interface lane 1 Rx valid indication.	Only in x4
txcompl1_ext	0	Pipe interface lane 1 Tx compliance control.	Only in x4
txdata1_ext[7:0]	0	Pipe interface lane 1 Tx data signals, carries the parallel transmit data.	Only in x4
txdatak1_ext	0	Pipe interface lane 1 Tx data K-character flag.	Only in x4
txelecidle1_ext	0	Pipe interface lane 1 Tx electrical idle control.	Only in x4
rxdata2_ext[7:0]	I	Pipe interface lane 2 Rx data signals, carries the parallel received data.	Only in x4
rxdatak2_ext	I	Pipe interface lane 2 Rx data K-character flag.	Only in x4
rxelecidle2_ext	I	Pipe interface lane 2 Rx electrical idle indication.	Only in x4
rxpolarity2_ext	0	Pipe interface lane 2 Rx polarity inversion control.	Only in x4
rxstatus2_ext[1:0]	I	Pipe interface lane 2 Rx status flags.	Only in x4
rxvalid2_ext	I	Pipe interface lane 2 Rx valid indication.	Only in x4
txcompl2_ext	0	Pipe interface lane 2 Tx compliance control.	Only in x4
txdata2_ext[7:0]	0	Pipe interface lane 2 Tx data signals, carries the parallel transmit data.	Only in x4
txdatak2_ext	0	Pipe interface lane 2 Tx data K-character flag.	Only in x4
txelecidle2_ext	0	Pipe interface lane 2 Tx electrical idle control.	Only in x4
rxdata3_ext[7:0]	I	Pipe interface lane 3 Rx data signals, carries the parallel received data.	Only in x4
rxdatak3_ext	I	Pipe interface lane 3 Rx data K-character flag.	Only in x4
rxelecidle3_ext	I	Pipe interface lane 3 Rx electrical idle indication.	Only in x4
rxpolarity3_ext	0	Pipe interface lane 3 Rx polarity inversion control.	Only in x4
rxstatus3_ext[1:0]	I	Pipe interface lane 3 Rx status flags.	Only in x4
rxvalid3_ext	I	Pipe interface lane 3 Rx valid indication.	Only in x4
txcompl3_ext	0	Pipe interface lane 3 Tx compliance control.	Only in x4
txdata3_ext[7:0]	0	Pipe interface lane 3 Tx data signals, carries the parallel transmit data.	Only in x4
txdatak3_ext	0	Pipe interface lane 3 Tx data K-character flag.	Only in x4
txelecidle3_ext	0	Pipe interface lane 3 Tx electrical idle control.	Only in x4

Selecting an External PHY

You can select an external PHY and set the appropriate options in the MegaWizard Plug-In Manager flow or in the SOPC Builder flow, but the available options may differ. The following description uses the MegaWizard Plug-In Manager flow.

You can select one of the following PHY options on the MegaWizard interface System Settings page:

- Select the specific PHY.
- Select the type of interface to the PHY by selecting **Custom** in the **PHY type** list. Several PHYs have multiple interface modes.

Table 6-4 summarizes the PHY support matrix. For every supported PHY type and interface, the table lists the allowed lane widths.

Table 6-4. External PHY Support Matrix

PHY Type	Allowed Interfaces and Lanes							
	16-bit SDR (pclk only)	16-bit SDR (w/TxCk)	8-bit DDR (pclk only)	8-bit DDR (w/TxCk)	8-bit DDR/SDR (w/TxCk)	8-bit SDR (pclk only)	8-bit SDR (w/TxCk)	Serial Interface
Arria GX	-	-	-	-	-	-	-	×1, ×4
Stratix GX	-	-	-	-	-	-	-	×1, ×4
Stratix II GX	-	-	-	-	-	-	-	×1, ×4, ×8
Stratix IV GX	-	-	-	-	-	-	-	×1, ×4, ×8
TI XIO1100	-	×1	-	-	×1	-	-	-
NXP PX1011A	-	-	-	-	-	-	×1	-
Custom	×1, ×4	×1, ×4	×1, ×4	×1, ×4	-	×1, ×4	×1, ×4	-

The TI XIO1100 device has some additional control signals that need to be driven by your design. These can be statically pulled high or low in the board design, unless additional flexibility is needed by your design and you want to drive them from the Altera device. These signals are shown in the following list:

- P1_SLEEP must be pulled low. The PCI Express MegaCore function requires the refclk (RX_CLK from the XIO1100) to remain active while in the P1 powerdown state.
- DDR_EN must be pulled high if your variation of the PCI Express MegaCore function uses the 8-bit DDR (w/TxCk) mode. It must be pulled low if the 16-bit SDR (w/TxCk) mode is used.
- CLK_SEL must be set correctly based on the reference clock provided to the XIO1100. Consult the XIO1100 data sheet for specific recommendations.

External PHY Constraint Support

The PCI Express Compiler supports various location and timing constraints. When you parameterize and generate your MegaCore function, the Quartus II software creates a Tcl file that runs when you compile your design. The Tcl file incorporates the following constraints that you specify when you parameterize and generate during parameterization.

- refclk (pclk from the PHY) frequency constraint (125 MHz or 250 MHz)
- Setup and hold constraints for the input signals
- Clock-to-out constraints for the output signals
- I/O interface standard

Altera also provides an SDC file with the same constraints. The TimeQuest timing analyzer uses the SDC file.



You may need to modify the timing constraints to take into account the specific constraints of your external PHY and your board design.



To meet timing for the external PHY in the Cyclone III family, you must avoid using dual-purpose V_{REF} pins.

If you are using an external PHY with a design that targets a Stratix GX device, you must modify the PLL instance required by some external PHYs to function correctly in the Stratix GX. If you are using the Stratix GX internal PHY this modification is not necessary.

To modify the PLL instance, follow these steps:

1. Copy the PLL source file referenced in your variation file from the `<path>/ip/PCI Express Compiler/lib` directory, where `<path>` is the directory in which you installed the PCI Express Compiler, to your project directory.
2. Use the MegaWizard Plug In Manager to edit the PLL to specify the PLL uses the Stratix GX device.
3. Add the modified PLL source file to your Quartus II project.

This chapter introduces the root port or endpoint design example including a testbench, BFM, and a test driver module. When you create a PCI Express function variation using the MegaWizard Plug-In Manager flow as described in [Chapter 2, Getting Started](#), the PCI Express compiler generates a design example and testbench customized to your variation. This design example is not generated when using the SOPC Builder flow.

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 VHDL/Verilog HDL procedure interface to initiate PCI Express transactions to the endpoint.

The testbench uses a test driver module, `altpcieth_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 in the MegaWizard Plug-In Manager.

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, `altpcieth_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 in the MegaWizard Plug-In Manager interface. The endpoint model consists of an endpoint variation combined with the chaining DMA application described above.

PCI Express link monitoring and error injection capabilities are limited to those provided by the MegaCore function's `test_in` and `test_out` signals. The following sections describe the testbench, the design example, and root port BFM in detail.

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.

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 MegaCore function passes these messages on to the application layer which in most cases should ignore them, but in all cases using the descriptor/data interface must issue an `rx_ack` to clear the message from the Rx buffer.
- It can only handle received read requests that are less than or equal to the currently set **Maximum payload size** option (**Buffer Setup** page of the MegaWizard Plug-In Manager). 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.

The chaining DMA design example provided with the MegaCore function handles all of the above behaviors, even though the provided testbench cannot test them.

Additionally PCI Express link monitoring and error injection capabilities are limited to those provided by the MegaCore function's `test_in` and `test_out` signals. The testbench and root port BFM do not NAK any transactions.

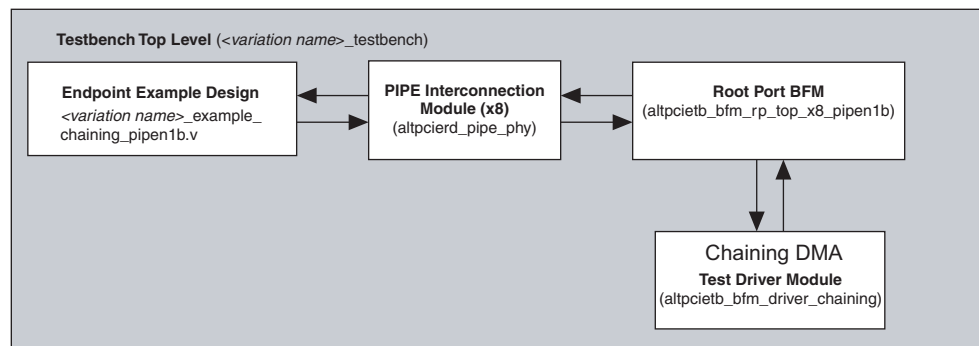
Endpoint Testbench

The testbench is provided in the subdirectory `<variation_name>_examples/chaining_dma/testbench` in your project directory. The testbench top level is named `<variation_name>_chaining_testbench`.

This testbench simulates up to an $\times 8$ 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 7-1 presents a high level view of the testbench.

Figure 7-1. Testbench Top-Level Module for Endpoint Designs



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

- **<variation name>_example_chaining_pipen1b**—This is the example endpoint design that includes your variation of the MegaCore function variation. For more information about this module, refer to “[Chaining DMA Design Example](#)” on page 7-6.
- **altpcieth_bfm_rp_top_x8_pipen1b**—This is the root port PCI Express BFM. For detailed information about this module, refer to “[Root Port BFM](#)” on page 7-26.
- **altpcieth_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.
- **altpcieth_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 7-22.

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.

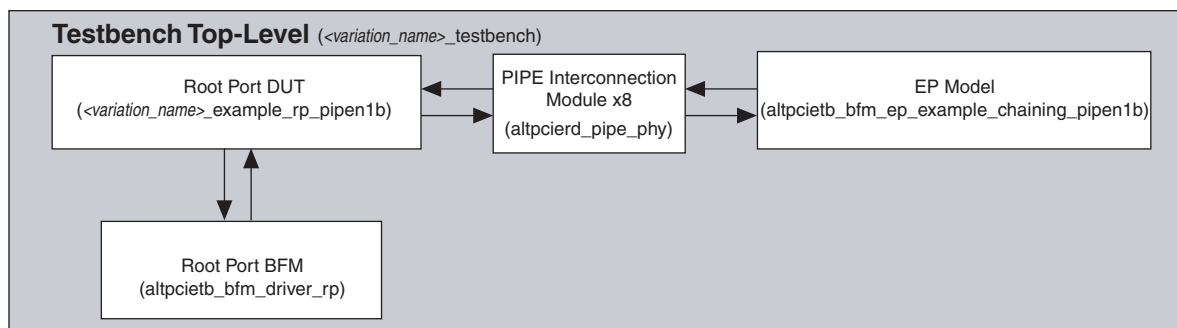
The testbench has several VHDL generics/Verilog HDL parameters that control the overall operation of the testbench. These generics are described in [Table 7-1](#).

Table 7-1. Testbench VHDL Generics /Verilog HDL Parameters

Generic/Parameter	Allowed Values	Default Value	Description
PIPE_MODE_SIM	0 or 1	1	Selects the PIPE interface (PIPE_MODE_SIM=1) or the serial interface (PIPE_MODE_SIM= 0) for the simulation. The PIPE interface typically simulates much faster than the serial interface. If the variation name file only implements the PIPE interface, then setting PIPE_MODE_SIM to 0 has no effect and the PIPE interface is always used.
NUM_CONNECTED_LANES	1,2,4,8	8	Controls how many lanes are interconnected by the testbench. Setting this generic value to a lower number simulates the endpoint operating on a narrower PCI Express interface than the maximum. If your variation only implements the ×1 MegaCore function, then this setting has no effect and only one lane is used.
FAST_COUNTERS	0 or 1	1	Setting this parameter to a 1 speeds up simulation by making many of the timing counters in the PCI Express MegaCore function operate faster than specified in the PCI Express specification. This parameter should usually be set to 1, but can be set to 0 if there is a need to simulate the true time-out values.

Root Port Testbench

The root port testbench is provided in the subdirectory *<variation_name>_examples/root_port/testbench* in your project directory. The top-level testbench is named *<variation_name>_rp_testbench*. Figure 7-2 presents a high level view of the testbench.

Figure 7-2. Testbench Top-Level Module for Root Port Designs

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:

- *<variation name>_example_rp_pipen1b*—This is the example root port design that includes your variation of the MegaCore function. For more information about this module, refer to “[Root Port Design Example](#)” on page 7-22.

- **altpcieth_bfm_ep_example_chaining_pipen1b**—This is the endpoint PCI Express model. The EP BFM consists of a Gen2 ×8 MegaCore function endpoint connected to the chaining DMA design example described in the section [“Chaining DMA Design Example” on page 7-6](#). [Table 7-2](#) shows the parameterization of the Gen2 ×8 MegaCore function endpoint.

Table 7-2. Gen2 ×8 MegaCore Function Endpoint Parameterization

Parameter	Value
Lanes	8
Port Type	Native Endpoint
Max rate	Gen2
BAR Type	BAR1:0—64-bit Prefetchable Memory, 256 MBytes–28 bits Bar 2:—32-Bit Non-Prefetchable, 256 KBytes–18 bits
Device ID	0xABCD
Vendor ID	0x1172
Tags supported	32
MSI messages requested	4
Error Reporting	Implement ECRC check, Implement ECRC generations Implement ECRC generate and forward
Maximum payload size	128 bytes
Number of virtual channels	1

- **altpcieth_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.
- **altpcieth_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 7-18](#).

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.

The testbench has several Verilog HDL parameters that control the overall operation of the testbench. These parameters are described in [Table 7-3](#).

Table 7-3. Testbench Verilog HDL Parameters for the Root Port Testbench

Parameter	Allowed Values	Default Value	Description
PIPE_MODE_SIM	0 or 1	1	Selects the PIPE interface (PIPE_MODE_SIM=1) or the serial interface (PIPE_MODE_SIM= 0) for the simulation. The PIPE interface typically simulates much faster than the serial interface. If the variation name file only implements the PIPE interface, then setting PIPE_MODE_SIM to 0 has no effect and the PIPE interface is always used.
NUM_CONNECTED_LANES	1,2,4,8	8	Controls how many lanes are interconnected by the testbench. Setting this generic value to a lower number simulates the endpoint operating on a narrower PCI Express interface than the maximum. If your variation only implements the ×1 MegaCore function, then this setting has no effect and only one lane is used.
FAST_COUNTERS	0 or 1	1	Setting this parameter to a 1 speeds up simulation by making many of the timing counters in the PCI Express MegaCore function operate faster than specified in the PCI Express specification. This parameter should usually be set to 1, but can be set to 0 if there is a need to simulate the true time-out values.

Chaining DMA Design Example

This design example shows how to use the MegaWizard Plug-In Manager flow 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 testbench generated by the PCI Express Compiler, along with this design example, provides a BFM driver module in Verilog HDL or VHDL 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 MegaCore function variation
- An application layer design example

When using the MegaWizard Plug-In manager flow, both components are automatically generated along with a testbench. All of the components are generated in the language (Verilog HDL or VHDL) that you selected for the variation file.



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 **MSI messages requested** parameter on the **Capabilities** page to at least 2.

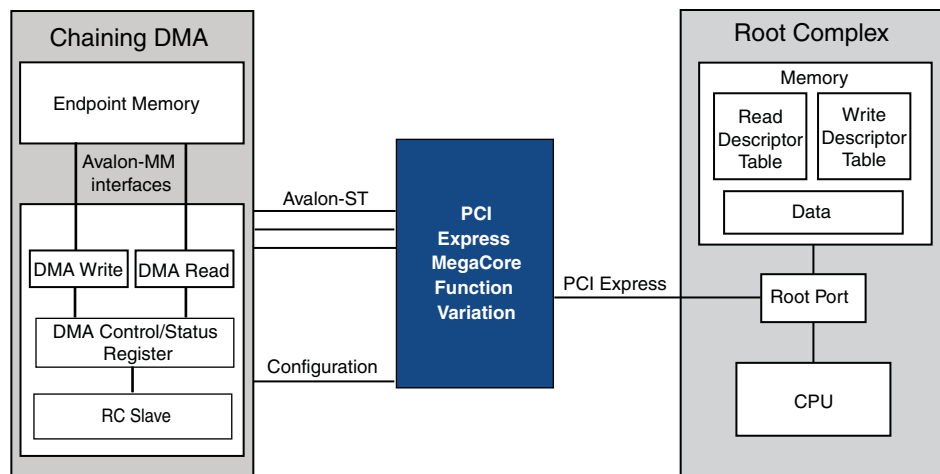
The chaining DMA design example uses an architecture capable of transferring a large amount of fragmented memory without accessing the DMA registers for every memory block. For each block of memory to be transferred, the chaining DMA design example uses a descriptor table containing the following information:

- 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 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 7–3 shows a block diagram of the design example connected to an external RC CPU.

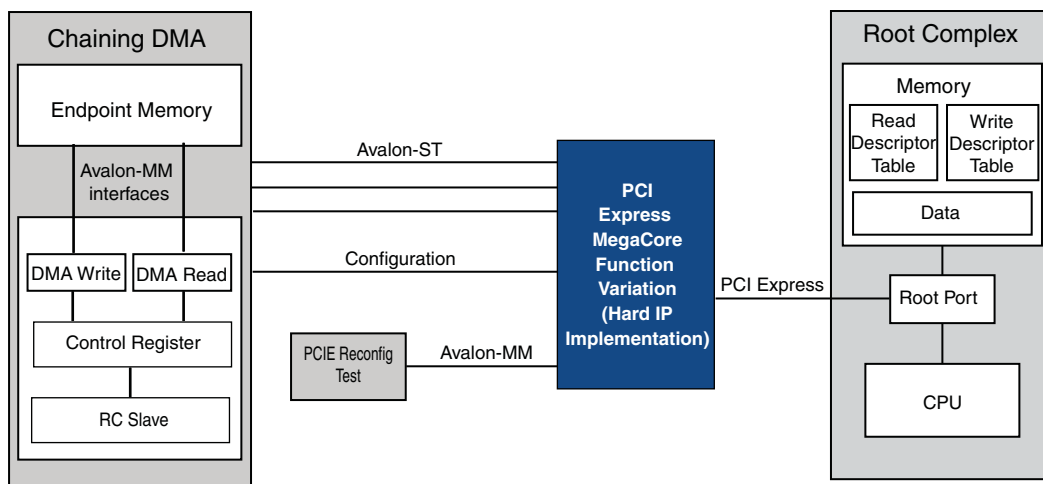
Figure 7–3. Top-Level Chaining DMA Example for Simulation



The block diagram contains the following elements:

- Endpoint DMA write and read requester modules.
- The chaining DMA design example connects to the Avalon-ST interface of the PCI Express MegaCore function when in Avalon-ST mode, or to the ICM when in descriptor/data mode. (Refer to [Appendix C, Incremental Compile Module for Descriptor/Data Examples](#)). The connections consist of the following interfaces:
 - The Avalon-ST Rx receives TLP header and data information from the PCI Express MegaCore function
 - The Avalon-ST Tx transmits TLP header and data information to the PCI Express MegaCore function
 - The Avalon-ST MSI port requests MSI interrupts from the PCI Express MegaCore function
 - 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 design example exercises the optional ECRC module when targeting the hard IP implementation and using a variation with both **Implement advanced error reporting** and **ECRC forwarding** set to **On** in the **“Capabilities Parameters”** on [page 3–5](#).
- The design example exercises the optional PCI Express reconfiguration block when targeting the hard IP implementation created using the MegaWizard Plug-In manager if you selected **PCIe Reconfig** on the **System Settings** page. [Figure 7–4](#) illustrates this test environment.

Figure 7–4. Top-Level Chaining DMA Example for Simulation—Hard IP Implementation with PCIe Reconfig Block



The example endpoint design application layer accomplishes the following objectives:

- Shows you how to interface to the PCI Express MegaCore function in Avalon-ST mode, or in descriptor/data mode through the ICM. Refer to [Appendix C, Incremental Compile Module for Descriptor/Data Examples](#).
- 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 MegaCore function to check the ECRC dword from the Avalon-ST Rx path and to generate the ECRC for the Avalon-ST Tx path.
- If the PCI Express reconfiguration block functionality is enabled, provides a test that increments the Vendor ID register to demonstrate this functionality.

You can use the example endpoint design in the testbench simulation and compile a complete design for an Altera device. All of the modules necessary to implement the design example with the variation file are contained in one of the following files, based on the language you use:

```
<variation name>_examples/chaining_dma/example_chaining.vhd
or
<variation name>_examples/chaining_dma/example_chaining.v
```

These files are created in the project directory when files are generated.

The following modules are included in the design example and located in the subdirectory *<variation name>_example/chaining_dma*:

- *<variation name>_example_pipen1b*—This module is the top level of the example endpoint design that you use for simulation. This module is contained in the following files produced by the MegaWizard interface:

```
<variation name>_example_chaining_top.vhd, and
<variation name>_example_chaining_top.v
```

This module provides both PIPE and serial interfaces for the simulation environment. This module has two debug ports named *test_out_icm* (which is either the *test_out_icm* signal from the Incremental Compile Module in descriptor/data example designs or the *test_out* signal from the MegaCore function in Avalon-ST example designs) and *test_in*. Refer to [“Test Interface Signals—Hard IP Implementation” on page 5-70](#) which allow you to monitor and control internal states of the MegaCore function.

For synthesis, the top level module is *<variation name>_example_chaining_top*. This module instantiates the module *<variation name>_example_pipen1b* 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*—The MegaWizard interface creates this variation name module when it generates files based on the parameters that you set. For simulation purposes, the IP functional simulation model produced by the MegaWizard interface is used. The IP functional simulation model is either the *<variation name>.vho* or *<variation name>.vo* file. The Quartus II software uses the associated *<variation name>.vhd* or *<variation name>.v* file during compilation. For information on producing a functional simulation model, see the [Chapter 2, Getting Started](#).

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 and VHDL, and reflect each hierarchical level:

- **altpcierrd_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.
- **altpcierrd_cdma_ast_rx**, **altpcierrd_cdma_ast_rx_64**, **altpcierrd_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 MegaCore function to the descriptor/data interface used by the chaining DMA submodules. **altpcierrd_cdma_ast_rx** is used with the descriptor/data MegaCore function (through the ICM). **altpcierrd_cdma_ast_rx_64** is used with the 64-bit Avalon-ST MegaCore function. **altpcierrd_cdma_ast_rx_128** is used with the 128-bit Avalon-ST MegaCore function.
- **altpcierrd_cdma_ast_tx**, **altpcierrd_cdma_ast_tx_64**, **altpcierrd_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 MegaCore function. **altpcierrd_cdma_ast_tx** is used with the descriptor/data MegaCore function (through the ICM). **altpcierrd_cdma_ast_tx_64** is used with the 64-bit Avalon-ST MegaCore function. **altpcierrd_cdma_ast_tx_128** is used with the 128-bit Avalon-ST MegaCore function.
- **altpcierrd_cdma_ast_msi**—This module converts MSI requests from the chaining DMA submodules into Avalon-ST streaming data. This module is only used with the descriptor/data MegaCore function (through the ICM).

- **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_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_prog_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 MegaCore function. **altpcierd_read_dma_requester_128** is used with the 128-bit Avalon-ST MegaCore function.
- **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 MegaCore function. **altpcierd_write_dma_requester_128** is used with the 128-bit Avalon-ST MegaCore function.
- **altpcierd_cpld_rx_buffer**—This module monitors the available space of the RX Buffer; It prevents RX Buffer overflow by arbitrating memory read request issued by the application.

- **altpcierrd_cdma_ecrc_check_64, altpcierrd_cdma_ecrc_check_128**—This module checks for and flags PCI Express ECRC errors on TLPs as they are received on the Avalon-ST interface of the chaining DMA.
altpcierrd_cdma_ecrc_check_64 is used with the 64-bit Avalon-ST MegaCore function. **altpcierrd_cdma_ecrc_check_128** is used with the 128-bit Avalon-ST MegaCore function.
- **altpcierrd_cdma_rx_ecrc_64.v, altpcierrd_cdma_rx_ecrc_64_altrcr.v, altpcierrd_cdma_rx_ecrc_64.vo**—These modules contain the CRC32 checking Megafunction used in the **altpcierrd_ecrc_check_64** module. The **.v** files are used for synthesis. The **.vo** file is used for simulation.
- **altpcierrd_cdma_ecrc_gen**—This module generates PCI Express ECRC and appends it to the end of the TLPs transmitted on the Avalon-ST Tx interface of the chaining DMA. This module instantiates the **altpcierrd_cdma_gen_ctl_64, altpcierrd_cdma_gen_ctl_128, and altpcierrd_cdma_gen_datapath** modules.
- **altpcierrd_cdma_ecrc_gen_ctl_64, altpcierrd_cdma_ecrc_gen_ctl_128**—This module controls the data stream going to the **altpcierrd_cdma_tx_ecrc** module for ECRC calculation, and generates controls for the main datapath (**altpcierrd_cdma_ecrc_gen_datapath**).
- **altpcierrd_cdma_ecrc_gen_datapath**—This module routes the Avalon-ST data through a delay pipe before sending it across the Avalon-ST interface to the MegaCore function to ensure the ECRC is available when the end of the TLP is transmitted across the Avalon-ST interface.
- **altpcierrd_cdma_ecrc_gen_calc**—This module instantiates the Tx ECRC megafunctions.
- **altpcierrd_cdma_tx_ecrc_64.v, altpcierrd_cdma_tx_ecrc_64_altrcr.v, altpcierrd_cdma_tx_ecrc_64.vo**—These modules contain the CRC32 generation megafunction used in the **altpcierrd_ecrc_gen** module. The **.v** files are used for synthesis. The **.vo** file is used for simulation.
- **altpcierrd_tx_ecrc_data_fifo, altpcierrd_tx_ecrc_ctl_fifo, altpcierrd_tx_ecrc_fifo**—These are FIFOs that are used in the ECRC generator modules in **altpcierrd_cdma_ecrc_gen**.
- **altpcierrd_pcie_reconfig**—This module is instantiated when the **PCIE reconfig** option on the **System Settings** page is turned on. It consists of a Avalon-MM master which drives the PCIE reconfig Avalon-MM slave of the device under test. The module performs the following sequence using the Avalon-MM interface prior to any PCI Express configuration sequence:
 - a. Turns on PCIE reconfig mode and resets the reconfiguration circuitry in the hard IP implementation by writing 0x2 to PCIE reconfig address 0x0 and asserting the reset signal, **npor**.
 - b. Releases reset of the hard IP reconfiguration circuitry by writing 0x0 to PCIE reconfig address 0x0.
 - c. Reads the PCIE vendor ID register at PCIE reconfig address 0x89.
 - d. Increments the vendor ID register by one and writes it back to PCIE reconfig address 0x89.
 - e. Removes the hard IP reconfiguration circuitry and SERDES from the reset state by deasserting **npor**.

- **altpcierr_cplerr_lmi**—This module transfers the `err_desc_func0` from the application to the PCE Express hard IP using the LMI interface. It also retimes the `cpl_err` bits from the application to the hard IP. This module is only used with the hard IP implementation of the MegaCore function.
- **altpcierr_tl_cfg_sample**—This module demultiplexes the configuration space signals from the `tl_cfg_ctl` bus from the hard IP and synchronizes this information, along with the `tl_cfg_sts` bus to the user clock (`pld_clk`) domain. This module is only used with the hard IP implementation.

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 7-4 shows the mapping.

Table 7-4. 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 <code>rc_slave</code> module to bypass the chaining DMA.
32-bit BAR2 32-bit BAR3 64-bit BAR3:2	Maps to DMA Read and DMA write control and status registers, a minimum of 256 bytes.
32-bit BAR4 32-bit BAR5 64-bit BAR5:4	Maps to 32 KByte target memory block. Use the <code>rc_slave</code> 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 7-5 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 7-5. Chaining DMA Control Register Definitions

Addr (1)	Register Name	31	24	23	16	15	0
0x0	DMA Wr Cntl DW0	Control Field (refer to Table 7–6)				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 7–6)				No 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 7–5: (1) This is the endpoint byte address offset from BAR2 or BAR3.							

Table 7-6 describes the control fields of the of the DMA read and DMA write control registers.

Table 7-6. 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 7-10 describes the descriptor table.
[24:20]	MSI Number	When your RC reads the MSI capabilities of the endpoint, these register bits map to the PCI Express back-end MSI signals <code>app_msi_num</code> [4:0]. If there is more than one MSI, the default mapping if all the MSIs are available, is: <ul style="list-style-type: none"> ■ MSI 0 = Read ■ MSI 1 = Write
[30:28]	MSI Traffic Class	When the RC application software reads the MSI capabilities of the endpoint, this value is assigned by default to MSI traffic class 0. These register bits map to the PCI Express back-end signal <code>app_msi_tc</code> [2:0].

Table 7-7 defines the DMA status registers. These registers are read only.

Table 7-7. Chaining DMA Status Register Definitions

Addr (1)	Register Name	31	24	23	16	15	0
0x20	DMA Wr Status Hi	For field definitions refer to Table 7–8					
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 7–9					
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 7–7:							
(1) This is the endpoint byte address offset from BAR2 or BAR3.							

Table 7-8 describes the fields of the DMA write status register. All of these fields are read only.

Table 7-8. Fields in the DMA Write Status High Register

Bit	Field	Description
[31:28]	CDMA version	Identifies the version of the chaining DMA example design.
[27:26]	Core type	Identifies the core interface. The following encodings are defined: <ul style="list-style-type: none"> 01 Descriptor/Data Interface 10 Avalon-ST soft IP implementation 00 Other
[25:24]	Reserved	—
[23:21]	Max payload size	The following encodings are defined: <ul style="list-style-type: none"> 001 128 bytes 001 256 bytes 010 512 bytes 011 1024 bytes 100 2048 bytes
[20:17]	Reserved	—
16	Write DMA descriptor FIFO empty	Indicates that there are no more descriptors pending in the write DMA.
[15:0]	Write DMA EPLAST	Indicates the number of the last descriptor completed by the write DMA.

Table 7-9 describes the fields in the DMA read status high register. All of these fields are read only.

Table 7-9. Fields in the DMA Read Status High Register

Bit	Field	Description
[31:25]	Board number	Indicates to the software application which board is being used. The following encodings are defined: <ul style="list-style-type: none"> ■ 0 Altera Stratix II GX ×1 ■ 1 Altera Stratix II GX ×4 ■ 2 Altera Stratix II GX ×8 ■ 3 Cyclone II ×1 ■ 4 Arria GX ×1 ■ 5 Arria GX ×4 ■ 6 Custom PHY ×1 ■ 7 Custom PHY ×4
24	Reserved	—
[23:21]	Max Read Request Size	The following encodings are defined: <ul style="list-style-type: none"> ■ 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: <ul style="list-style-type: none"> ■ 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.

Chaining DMA Descriptor Tables

Table 7-10 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 $\langle n \rangle$ 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 7-10. 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 <code>EPLAST_ENA</code> 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 7-11 shows the layout of the descriptor fields following the descriptor header.

Table 7-11. Chaining DMA Descriptor Format Map

31	22	21	16	15	0
Reserved		Control Fields (refer to Table 7–12)		DMA Length	
Endpoint Address					
RC Address Upper DWORD					
RC Address Lower DWORD					

Table 7-12. Chaining DMA Descriptor Format Map (Control Fields)

21	18	17	16
Reserved		EPLAST_ENA	MSI

Each descriptor provides the hardware information on one DMA transfer. Table 7-13 describes each descriptor field.

Table 7-13. 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 bytes 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 $\langle 0 - n \rangle$. (Refer to Table 7-10.)
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 generated by the MegaWizard interface during the generate step 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.

For an endpoint VHDL version of this file, see:

`<variation_name>_examples/chaining_dma/testbench/
altpcieth_bfm_driver_chaining.vhd`

For an endpoint Verilog HDL file, see:

`<variation_name>_examples/chaining_dma/testbench/
altpcieth_bfm_driver_chaining.v`

For a root port Verilog HDL file, see:

`<variation_name>_examples/rootport/testbench/altpcieth_bfm_driver_rp.v`

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 **altpcieth_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 **altpcieth_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 7-6) 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 7-17 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 7-6) 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 7-17 after completing the data transfer for the first and last descriptors.
 - b. The chaining DMA issues an MSI when the last descriptor has completed.
 - c. The data written back to BFM is checked against the data that was read from the BFM.
 - d. 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 7-14, Table 7-15, and Table 7-16).

Table 7-14. Write Descriptor 0

	Offset in BFM Shared Memory	Value	Description
DW0	0x810	82	Transfer length in DWORDS and control bits as described in Table 7-6 on page 7-14
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 7-15. 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 7-18
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 7-16. Write Descriptor 2

	Offset in BFM Shared Memory	Value	Description
DW0	0x830	644	Transfer length in DWORDS and control bits as described in Table 7-6 on page 7-14
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 7-17](#)), into the DMA write register module.

Table 7-17. 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 7-5 on page 7-14
DW1	0x4	0	BFM shared memory descriptor table upper address value

Table 7-17. DMA Control Register Setup for DMA Write

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 7-18, Table 7-19, and Table 7-20).

Table 7-18. 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 7-18
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 7-19. 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 7-18
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 0xB BBBB_0001	Data content in the BFM shared memory from address: 0x10900

Table 7-20. Read Descriptor 2 (Part 1 of 2)

	Offset in BFM Shared Memory	Value	Description
DW0	0x930	644	Transfer length in DWORDS and control bits as described in on page 7-18
DW1	0x934	0	Endpoint address value

Table 7-20. Read Descriptor 2 (Part 2 of 2)

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 7-21) into the DMA read register module.

Table 7-21. 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 7-5 on page 7-14
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.

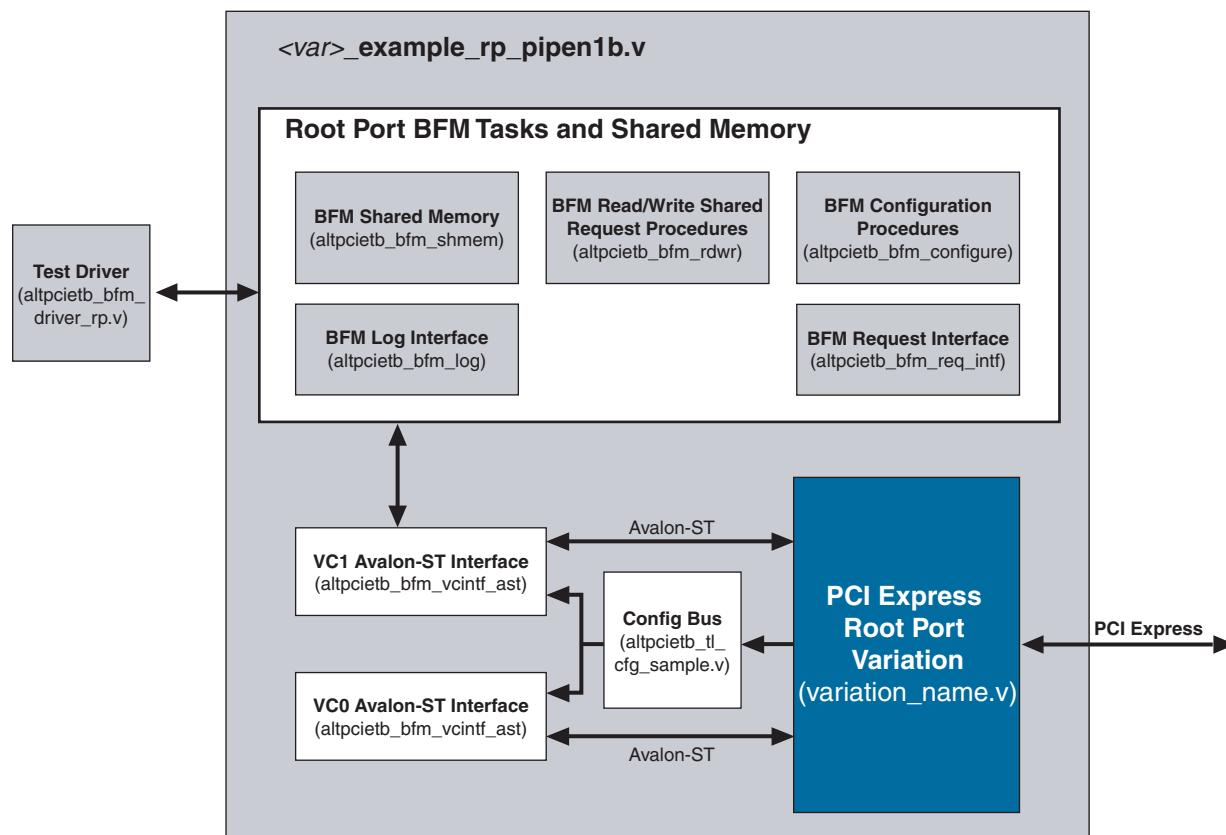
3. Waits for the DMA read completion by polling the BFM share memory location 0x90c, where the DMA read engine is updating the value of the number of completed descriptors. Calls the procedures `rcmem_poll` and `msi_poll` to determine when the DMA read transfers have completed.

Root Port Design Example

The design example includes the following primary components:

- PCI Express MegaCore function root port variation (`<variation_name>.v`).
- VC0:1 Avalon-ST Interfaces (`altpcieth_bfm_vc_intf_ast`)—handles the transfer of PCI Express requests and completions to and from the PCI Express MegaCore function 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 `altpcieth_bfm_vc_intf_ast`, the root port memory space, and simulation functions such as displaying messages and stopping simulation.
- Test Driver (`altpcieth_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 7-18 for a detailed description.

Figure 7-5.



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 `<variation_name>_example_rp_pipen1b.v`. This file is created in the `<variation_name>_examples/root_port` subdirectory of your project when the PCI Express MegaCore function variant is generated.

The MegaWizard interface creates the variation files in the top-level directory of your project, including the following files:

- `<variation_name>.v`—the top level file of the PCI Express MegaCore function variation. The file instantiates the SERDES and PIPE interfaces, and the parameterized core, `<variation_name>_core.v`.
- `<variation_name>_serdes.v`—contains the SERDES.
- `<variation_name>_core.v`—used in synthesizing `<variation_name>.v`.
- `<variation_name>_core.vo`—used in simulating `<variation_name>.v`.

The following modules are generated for the design example in the subdirectory `<variation_name>_examples/root_port`:

- `<variation_name>_example_rp_pipen1b.v`—the top-level of the root port design example that you use for simulation. This module instantiates the root port PCI Express MegaCore function variation, `<variation_name>.v`, and the root port application `altpcieth_bfm_vc_intf_ast`. 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 MegaCore function) and `test_in` which allows you to monitor and control internal states of the PCI Express MegaCore function variation. (Refer to “Test Signals” on page 5–70.)
- `<variation_name>_example_rp_top.v`—the top level of the root port example design that you use for synthesis. The file instantiates `<variation_name>_example_rp_pipen1b.v`. Note, however, that the synthesized design only contains the PCI Express variant, and not the application layer, `altpcieth_bfm_vc_intf_ast`. Instead, the application is replaced with dummy signals in order to preserve the variant's application interface. This module is provided so that you can compile the variation in the Quartus II software.
- `altpcieth_bfm_vc_intf_ast.v`—a wrapper module which instantiates either `altpcieth_vc_intf_ast_64` or `altpcieth_vc_intf_ast_128` based on the type of Avalon-ST interface that is generated. It also instantiates the ECRC modules `altpcieth_cdma_ecrc_check` and `altpcieth_cdma_ecrc_gen` which are used when ECRC forwarding is enabled.
- `altpcieth_vc_intf_ast_64.v` and `altpcieth_vc_intf_ast_128.v`—provide the interface between the PCI Express variant and the root port BFM tasks. They provide the same function as the `altpcieth_vc_intf.v` module, transmitting PCI Express requests and handling completions. Refer to the “Root Port BFM” on page 7–26 for a full description of this function. This version uses Avalon-ST signalling with either a 64- or 128-bit data bus to the PCI Express MegaCore function variation. There is one VC interface per virtual channel.
- `altpcieth_bfm_vc_intf_ast_common.v`—contains tasks called by `altpcieth_vc_intf_ast_64.v` and `altpcieth_vc_intf_ast_128.v`
- `altpcieth_cdma_ecrc_check.v`—checks and removes the ECRC from TLPs received on the Avalon-ST interface of the PCI Express MegaCore function variation. Contains the following submodules:
`altpcieth_cdma_ecrc_check_64.v`, `altpcieth_rx_ecrc_64.v`, `altpcieth_rx_ecrc_64.vo`,
`altpcieth_rx_ecrc_64_altrc.v`, `altpcieth_rx_ecrc_128.v`, `altpcieth_rx_ecrc_128.vo`,
`altpcieth_rx_ecrc_128_altrc.v`. Refer to the “Chaining DMA Design Example” on page 7–6 for a description of these submodules

- **altpcierrd_cdma_ecrc_gen.v**—generates and appends ECRC to the TLPs transmitted on the Avalon-ST interface of the PCI Express variant. Contains the following submodules:

altpcierrd_cdma_ecrc_gen_calc.v, altpcierrd_cdma_ecrc_gen_ctl_64.v, altpcierrd_cdma_ecrc_gen_ctl_128.v, altpcierrd_cdma_ecrc_gen_datapath.v, altpcierrd_tx_ecrc_64.v, altpcierrd_tx_ecrc_64.vo, altpcierrd_tx_ecrc_64_altrcr.v, altpcierrd_tx_ecrc_128.v, altpcierrd_tx_ecrc_128.vo, altpcierrd_tx_ecrc_128_altrcr.v, altpcierrd_tx_ecrc_ctl_fifo.v, altpcierrd_tx_ecrc_data_fifo.v, altpcierrd_tx_ecrc_fifo.v Refer to the “Chaining DMA Design Example” on page 7-6 for a description of these submodules.
- **altpcierrd_tl_cfg_sample.v**—accesses configuration space signals from the variant. Refer to the “Chaining DMA Design Example” on page 7-6 for a description of this module.

Files in subdirectory <variation_name>_example/common/testbench:

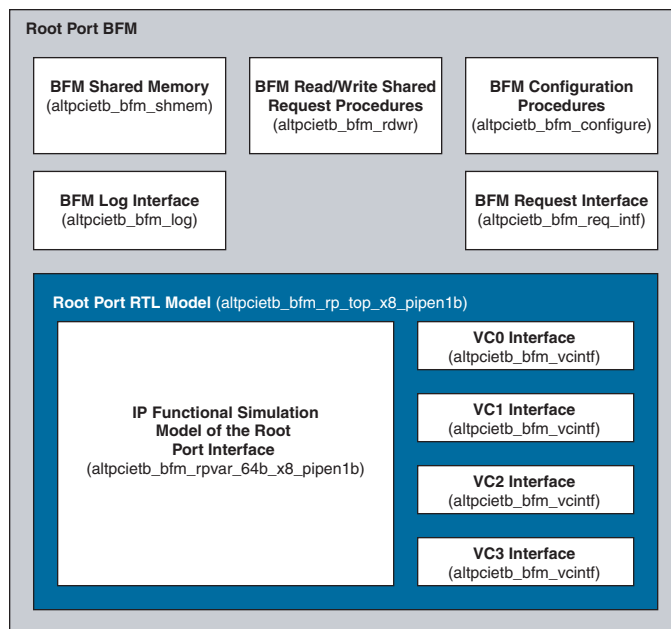
- **altpciemb_bfm_ep_example_chaining_pipen1b.vo**—the simulation model for the chaining DMA endpoint.
- **altpciemb_bfm_shmem.v, altpciemb_bfm_shmem_common.v**—root port memory space. Refer to the “Root Port BFM” on page 7-26 for a full description of this module
- **altpciemb_bfm_rdwr.v**— requests PCI Express read and writes. Refer to the “Root Port BFM” on page 7-26 for a full description of this module.
- **altpciemb_bfm_configure.v**— configures PCI Express configuration space registers in the root port and endpoint. Refer to the “Root Port BFM” on page 7-26 for a full description of this module
- **altpciemb_bfm_log.v, and altpciemb_bfm_log_common.v**—displays and logs simulation messages. Refer to the “Root Port BFM” on page 7-26 for a full description of this module.
- **altpciemb_bfm_req_intf.v, and altpciemb_bfm_req_intf_common.v**—includes tasks used to manage requests from altpciemb_bfm_rdwr to altpciemb_vc_intf_ast. Refer to the “Root Port BFM” on page 7-26 for a full description of this module.
- **altpciemb_bfm_constants.v**—contains global constants used by the root port BFM.
- **altpciemb_ltssm_mon.v**—displays LTSSM state transitions.
- **altpciemb_pipe_phy.v, altpciemb_pipe_xtx2yry.v, and altpciemb_phasefifo.v**—used to simulate the PHY and support circuitry.
- **altpciemb_pll_100_125.v, altpciemb_pll_100_250.v, altpciemb_pll_125_250.v, altpciemb_pll_phy0.v, altpciemb_pll_phy1_62p5.v, altpciemb_pll_phy2.v, altpciemb_pll_phy3_62p5.v, altpciemb_pll_phy4_62p5.v, altpciemb_pll_phy5_62p5.v**—PLLs used for simulation. The type of PHY interface selected for the variant determines which PLL is used.
- **altpciemb_4sgx_alt_reconfig.v**—transceiver reconfiguration module used for simulation.
- **altpciemb_rst_clk.v**— generates PCI Express reset and reference clock.

Root Port BFM

The basic root port BFM provides a VHDL procedure-based or 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 7-6 provides an overview of the root port BFM.

Figure 7-6. Root Port BFM



The functionality of each of the modules included in Figure 7-6 is explained below.

- BFM shared memory (`altpcieth_bfm_shmem` VHDL package or 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 7-43.

- BFM Read/Write Request Procedures/Functions (**altpcieth_bfm_rdwr** VHDL package or Verilog HDL include file)—This package provides the basic BFM procedure calls for PCI Express read and write requests. For details on these procedures, see “BFM Read and Write Procedures” on page 7-34.
- BFM Configuration Procedures/Functions (**altpcieth_bfm_configure** VHDL package or Verilog HDL include file)—These procedures and 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 7-41.
- BFM Log Interface (**altpcieth_bfm_log** VHDL package or Verilog HDL include file)—The BFM log interface 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 7-46.
- BFM Request Interface (**altpcieth_bfm_req_intf** VHDL package or Verilog HDL include file)—This interface provides the low-level interface between the **altpcieth_bfm_rdwr** and **altpcieth_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.
- The root port BFM included with the PCI Express Compiler is designed to test just one PCI Express MegaCore function at a time. When using the SOPC Builder design flow, in order to simulate correctly, you should comment out all but one of the PCI Express Compiler testbench modules, named *<variation_name>_testbench*, in the SOPC Builder generated system file. These modules are instantiated near the end of the system file. You can select which one to use for any given simulation run.
- Root Port RTL Model (**altpcieth_bfm_rp_top_x8_pipen1b** VHDL entity or Verilog HDL Module)—This is the Register Transfer Level (RTL) portion of the model. This model takes the requests from the above modules and handles them at an RTL level to interface to the PCI Express link. You do not need to access this module directly to adapt the testbench to test your endpoint application.
- VC0:3 Interfaces (**altpcieth_bfm_vc_intf**)—These interface modules handle the VC-specific interfaces on 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.
- Root port interface model(**altpcieth_bfm_rpvar_64b_x8_pipen1b**)—This is an IP functional simulation model of a version of the MegaCore function specially modified to support root port operation. Its application layer interface is very similar to the application layer interface of the MegaCore function used for endpoint mode.

All of the files for the BFM are generated by the MegaWizard interface in the `<variation name>_examples/common/testbench` directory.

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 7-7 on page 7-31](#) – [Figure 7-9 on page 7-33](#).

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 part of `altpcieth_bfm_configure`.



Configuration procedures and functions are in the VHDL package file `altpcieth_bfm_configure.vhd` or in the Verilog HDL include file `altpcieth_bfm_configure.v` that uses the `altpcieth_bfm_configure_common.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 7-9 on page 7-33](#) 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 7-8 on page 7-32](#).

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 7-7 on page 7-31](#).

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.

- Based on the above BAR assignments, the root port configuration space address windows are assigned to encompass the valid BAR address ranges.
- 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 7-22](#) shows how those offsets are used.

Table 7-22. 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

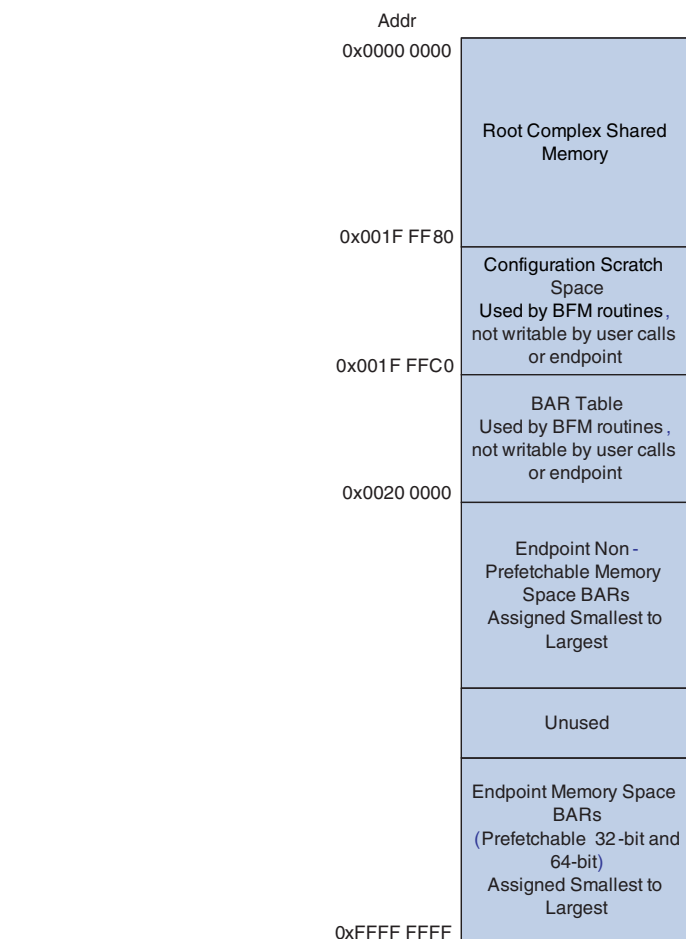
Table 7-22. BAR Table Structure

Offset (Bytes)	Description
+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 virtual channel capability or advanced error reporting capability.

Besides the `ebfm_cfg_rp_ep` procedure in `altpcieth_bfm_configure`, routines to read and write endpoint configuration space registers directly are available in the `altpcieth_bfm_rdwr` VHDL package or 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 7-7](#).

Figure 7-7. Memory Space Layout—4 GByte Limit



If `addr_map_4GB_limit` is 0, the resulting memory space map is shown in [Figure 7-8](#).

Figure 7-8. Memory Space Layout—No Limit

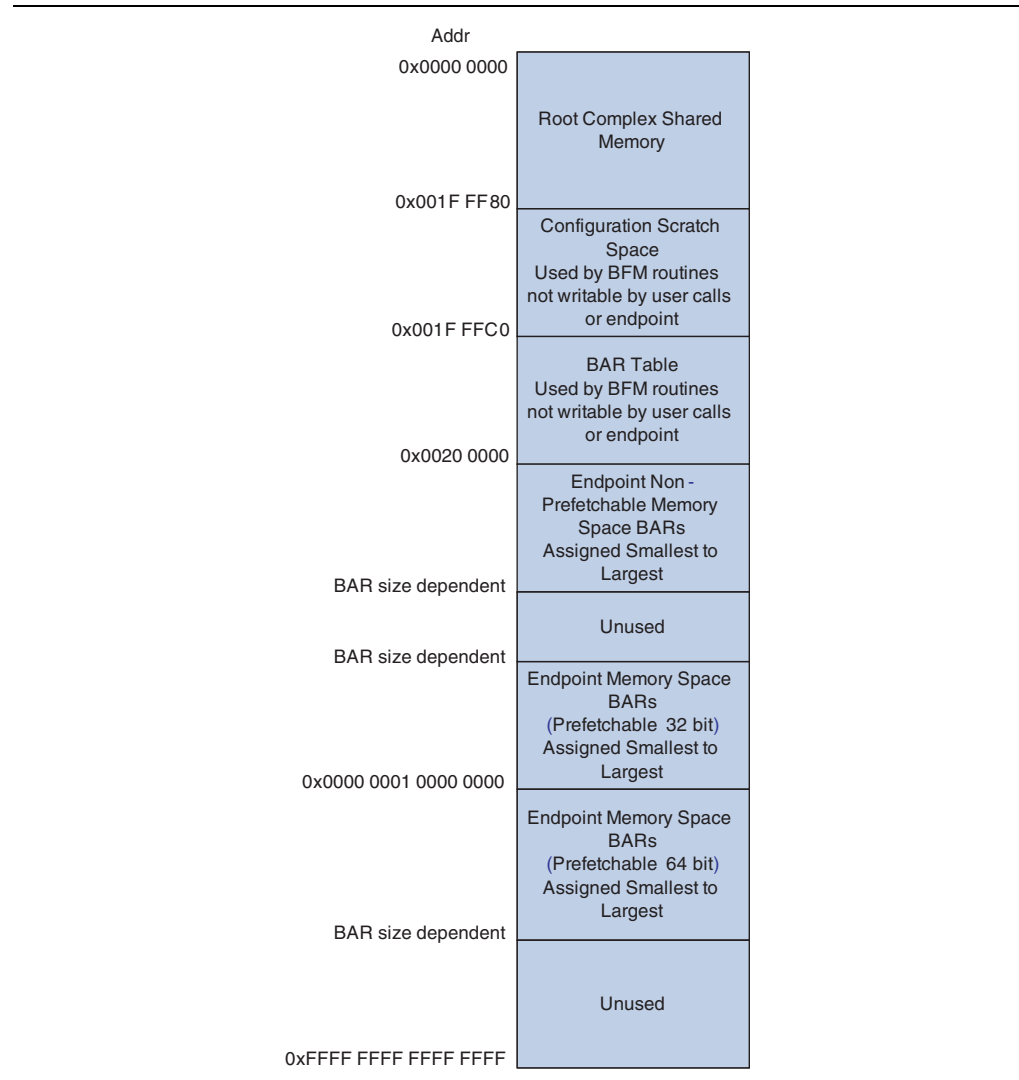
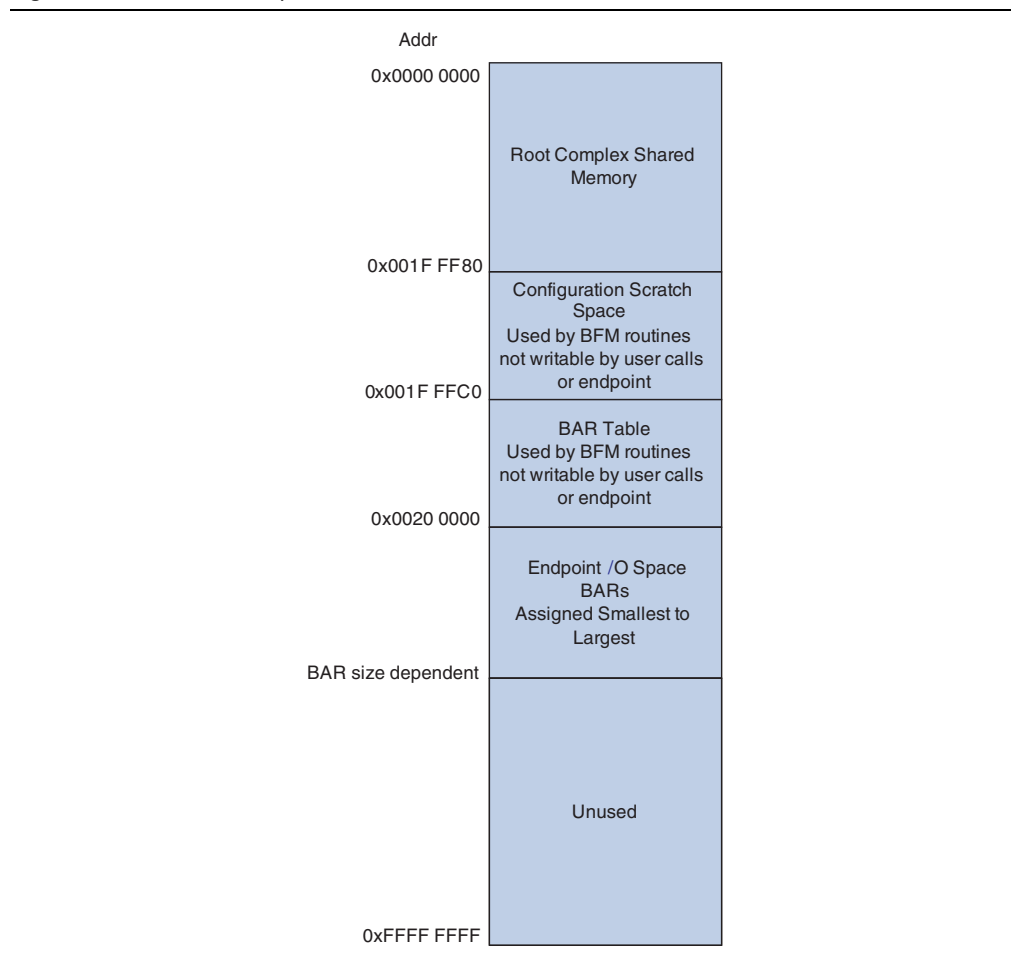


Figure 7-9 shows the I/O address space.

Figure 7-9. I/O Address Space



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 `altpcieth_bfm_rdwr`. The procedures and functions listed below are available in the VHDL package file

`altpcieth_bfm_rdwr.vhd` or in the Verilog HDL include file `altpcieth_bfm_rdwr.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_barrrd_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_barrrd_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 7-28.) 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 7-34.

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.

This section describes both VHDL procedures and functions and Verilog HDL functions and tasks where applicable. Although most VHDL procedure are implemented as Verilog HDL tasks, some VHDL procedures are implemented as Verilog HDL functions rather than Verilog HDL tasks to allow these functions to be called by other Verilog HDL functions. Unless explicitly specified otherwise, all procedures in the following sections also are implemented as Verilog HDL tasks.



You can see some underlying Verilog HDL procedures and functions that are called by other procedures that normally are hidden in the VHDL package. You should not call these undocumented procedures.

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 VHDL package `altpcieth_bfm_rdwr.vhd` or in the Verilog HDL include file `altpcieth_bfm_rdwr.v`. These procedures and functions support issuing memory and configuration transactions on the PCI Express link.

All VHDL arguments are subtype `natural` and are input-only unless specified otherwise. All Verilog HDL arguments are type `integer` and are input-only unless specified otherwise.

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 7-23. `ebfm_barwr` Procedure

Location	altpcieth_bfm_rdwr.v or altpcieth_bfm_rdwr.vhd	
Syntax	<code>ebfm_barwr(bar_table, bar_num, pcie_offset, lcladdr, byte_len, tclass)</code>	
Arguments	<code>bar_table</code>	Address of the endpoint <code>bar_table</code> structure in BFM shared memory. The <code>bar_table</code> 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.
	<code>bar_num</code>	Number of the BAR used with <code>pcie_offset</code> to determine PCI Express address.
	<code>pcie_offset</code>	Address offset from the BAR base.
	<code>lcladdr</code>	BFM shared memory address of the data to be written.
	<code>byte_len</code>	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.
	<code>tclass</code>	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 7-24. `ebfm_barwr_imm` Procedure

Location	altpcieth_bfm_rdwr.v or altpcieth_bfm_rdwr.vhd	
Syntax	<code>ebfm_barwr_imm(bar_table, bar_num, pcie_offset, imm_data, byte_len, tclass)</code>	
Arguments	<code>bar_table</code>	Address of the endpoint <code>bar_table</code> structure in BFM shared memory. The <code>bar_table</code> 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.
	<code>bar_num</code>	Number of the BAR used with <code>pcie_offset</code> to determine PCI Express address.
	<code>pcie_offset</code>	Address offset from the BAR base.
	<code>imm_data</code>	Data to be written. In VHDL, this argument is a <code>std_logic_vector(31 downto 0)</code> . In Verilog HDL, this argument is <code>reg [31:0]</code> . 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
	<code>byte_len</code>	Length of the data to be written in bytes. Maximum length is 4 bytes.
	<code>tclass</code>	Traffic class to be used for the PCI Express transaction.

ebfm_barrrd_wait Procedure

The `ebfm_barrrd_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 7-25. `ebfm_barrrd_wait` Procedure

Location	altpcieth_bfm_rdwr.v or altpcieth_bfm_rdwr.vhd	
Syntax	<code>ebfm_barrrd_wait(bar_table, bar_num, pcie_offset, lcladdr, byte_len, tclass)</code>	
Arguments	<code>bar_table</code>	Address of the endpoint <code>bar_table</code> structure in BFM shared memory. The <code>bar_table</code> 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.
	<code>bar_num</code>	Number of the BAR used with <code>pcie_offset</code> to determine PCI Express address.
	<code>pcie_offset</code>	Address offset from the BAR base.
	<code>lcladdr</code>	BFM shared memory address where the read data is stored.
	<code>byte_len</code>	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.
	<code>tclass</code>	Traffic class used for the PCI Express transaction.

ebfm_barrrd_nowt Procedure

The `ebfm_barrrd_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 7-26. `ebfm_barrrd_nowt` Procedure

Location	altpcieth_bfm_rdwr.v or altpcieth_bfm_rdwr.vhd	
Syntax	<code>ebfm_barrrd_nowt(bar_table, bar_num, pcie_offset, lcladdr, byte_len, tclass)</code>	
Arguments	<code>bar_table</code>	Address of the endpoint <code>bar_table</code> structure in BFM shared memory.
	<code>bar_num</code>	Number of the BAR used with <code>pcie_offset</code> to determine PCI Express address.
	<code>pcie_offset</code>	Address offset from the BAR base.
	<code>lcladdr</code>	BFM shared memory address where the read data is stored.
	<code>byte_len</code>	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.
	<code>tclass</code>	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 7-27. `ebfm_cfgwr_imm_wait` Procedure

Location	altpcieth_bfm_rdwr.v or altpcieth_bfm_rdwr.vhd											
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. In VHDL, this argument is a std_logic_vector(31 downto 0). In Verilog HDL, this argument is reg [31:0]. In both languages, the bits written depend on the length: <table><thead><tr><th>Length</th><th>Bits Written</th></tr></thead><tbody><tr><td>4</td><td>31 downto 0</td></tr><tr><td>3</td><td>23 downto 0</td></tr><tr><td>2</td><td>5 downto 0</td></tr><tr><td>1</td><td>7 downto 0</td></tr></tbody></table>	Length	Bits Written	4	31 downto 0	3	23 downto 0	2	5 downto 0	1	7 downto 0
	Length	Bits Written										
	4	31 downto 0										
3	23 downto 0											
2	5 downto 0											
1	7 downto 0											
compl_status	In VHDL, this argument is a std_logic_vector(2 downto 0) and is set by the procedure on return. In Verilog HDL, this argument is reg [2:0]. In both languages, this argument is the completion status as specified in the PCI Express specification: <table><thead><tr><th>Compl_Status</th><th>Definition</th></tr></thead><tbody><tr><td>000</td><td>SC— Successful completion</td></tr><tr><td>001</td><td>UR— Unsupported Request</td></tr><tr><td>010</td><td>CRS — Configuration Request Retry Status</td></tr><tr><td>100</td><td>CA — Completer Abort</td></tr></tbody></table>	Compl_Status	Definition	000	SC— Successful completion	001	UR— Unsupported Request	010	CRS — Configuration Request Retry Status	100	CA — Completer Abort	
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 7–28. `ebfm_cfgwr_imm_nowt` Procedure

Location	altpcietb_bfm_rdwr.v or altpcietb_bfm_rdwr.vhd										
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 In VHDL, this argument is a std_logic_vector(31 downto 0). In Verilog HDL, this argument is reg [31:0]. In both languages, the bits written depend on the length: <table><tr><td>Length</td><td>Bits Written</td></tr><tr><td>4</td><td>[31:0]</td></tr><tr><td>3</td><td>[23:0]</td></tr><tr><td>2</td><td>[15:0]</td></tr><tr><td>1</td><td>[7:0]</td></tr></table>	Length	Bits Written	4	[31:0]	3	[23:0]	2	[15:0]	1
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 7-29. `ebfm_cfgrd_wait` Procedure

Location	altpcieth_bfm_rdwr.v or altpcieth_bfm_rdwr.vhd											
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	<p>Completion status for the configuration transaction.</p> <p>In VHDL, this argument is a <code>std_logic_vector(2 downto 0)</code> and is set by the procedure on return.</p> <p>In Verilog HDL, this argument is <code>reg [2:0]</code>.</p> <p>In both languages, this is the completion status as specified in the PCI Express specification:</p> <table><thead><tr><th>Compl_Status</th><th>Definition</th></tr></thead><tbody><tr><td>000</td><td>SC— Successful completion</td></tr><tr><td>001</td><td>UR— Unsupported Request</td></tr><tr><td>010</td><td>CRS — Configuration Request Retry Status</td></tr><tr><td>100</td><td>CA — Completer Abort</td></tr></tbody></table>		Compl_Status	Definition	000	SC— Successful completion	001	UR— Unsupported Request	010	CRS — Configuration Request Retry Status	100
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 7-30. `ebfm_cfgrd_nowt` Procedure

Location	altpcieth_bfm_rdwr.v or altpcieth_bfm_rdwr.vhd	
Syntax	<code>ebfm_cfgrd_nowt(bus_num, dev_num, fnc_num, regb_ad, regb_ln, lcladdr)</code>	
Arguments	<code>bus_num</code>	PCI Express bus number of the target device.
	<code>dev_num</code>	PCI Express device number of the target device.
	<code>fnc_num</code>	Function number in the target device to be accessed.
	<code>regb_ad</code>	Byte-specific address of the register to be written.
	<code>regb_ln</code>	Length, in bytes, of the data written. Maximum length is four bytes. The <code>regb_ln</code> and <code>regb_ad</code> arguments cannot cross a DWORD boundary.
	<code>lcladdr</code>	BFM shared memory address where the read data should be placed.

BFM Performance Counting

This section describes BFM routines that allow you to access performance data. The root port BFM maintains a set of performance counters for the packets being transmitted and received by the root port. Counters exist for each of the following packets:

- Transmitted packets
- Transmitted qwords of payload data (A full qword is counted even if not all bytes are enabled.)
- Received packets
- Received qwords of payload data (A full qword is counted even if not all bytes are enabled.)

These counters count continuously from the start of the simulation. The procedure `ebfm_start_perf_sample` resets all of the counters to zero.

The `ebfm_disp_perf_sample` procedure displays scaled versions of these counters to the standard output. These values are displayed as a sum across all of the virtual channels. The `ebfm_disp_perf_sample` also resets the counters to zero, which effectively starts the next performance sample.

Typically, a performance measurement routine calls `ebfm_start_perf_sample` at the time when performance analysis should begin. Then `ebfm_disp_perf_sample` can be called at the end of the performance analysis time giving the aggregate numbers for the entire performance analysis time. Alternatively, `ebfm_disp_perf_sample` could be called multiple times during the performance analysis window to give a more precise view of the performance. The aggregate performance numbers would need to be calculated by post-processing of the simulator standard output.

The following procedures are available in the VHDL package `altpcieth_bfm_rdwr.vhd` or in the Verilog HDL include file `altpcieth_bfm_rdwr.v`.

All VHDL arguments are subtype `natural` and are input-only unless specified otherwise. All Verilog HDL arguments are type `integer` and are input-only unless specified otherwise.

ebfm_start_perf_sample Procedure

This procedure simply resets the performance counters. The procedure waits until the next root port BFM clock edge to ensure the counters are synchronously reset. Calling this routine effectively starts a performance sampling window.

Table 7-31. `ebfm_start_perf_sample` Procedure

Location	<code>altpcieth_bfm_rdwr.v</code> or <code>altpcieth_bfm_rdwr.vhd</code>
Syntax	<code>reg_intf_start_perf_sample</code>
Arguments	none

ebfm_disp_perf_sample Procedure

This procedure displays performance information to the standard output. The procedure will also reset the performance counters on the next root port BFM clock edge. Calling this routine effectively starts a new performance sampling window. No performance count information is lost from one sample window to the next.

Figure 7-10 shows an example of the output from this routine.

Figure 7-10. Output from ebfm_disp_perf_sample Procedure

```
# INFO:      92850 ns PERF: Sample Duration: 5008
ns
# INFO:      92850 ns PERF:      Tx Packets: 33
# INFO:      92850 ns PERF:      Tx Bytes: 8848
# INFO:      92850 ns PERF:      Tx MByte/sec: 1767
# INFO:      92850 ns PERF:      Tx Mbit/sec: 14134
# INFO:      92850 ns PERF:      Rx Packets: 34
# INFO:      92850 ns PERF:      Rx Bytes: 8832
# INFO:      92850 ns PERF:      Rx MByte/sec: 1764
# INFO:      92850 ns PERF:      Rx Mbit/sec: 14109
```

This example is from a VHDL version of the testbench. The Verilog HDL version may have slightly different formatting. Table 7-32 describes the information in Figure 7-10:

Table 7-32. Sample Duration and Tx Packets Description

Label	Description
Sample Duration	The time elapsed since the start of the sampling window, the time when ebfm_start_perf_sample or ebfm_disp_perf_sample was last called.
Tx Packets	Total number of packet headers transmitted by the root port BFM during the sample window.
Tx Bytes	Total number of payload data bytes transmitted by the root port BFM during the sample window. This is the number of QWORDS transferred multiplied by 8. No adjustment is made for partial QWORDS due to packets that do not start or end on a QWORD boundary.
Tx MBytes/sec	Transmitted megabytes per second during the sample window. This is Tx bytes divided by the sample duration.
Tx Mbit/sec	Transmitted megabits per second during the sample window. This is the Tx MBytes/sec multiplied by 8.
Rx Packets	Total number of packet headers received by the root port BFM during the sample window.
Rx Bytes	Total number of payload data bytes received by the root port BFM during the sample window. This is the number of QWORDS transferred multiplied by 8. No adjustment is made for partial QWORDS due to packets that do not start or end on a QWORD boundary.
Rx MByte/sec	Received megabytes per second during the sample window. This is Rx bytes divided by the sample duration.
Rx Mbit/sec	Received megabits per second during the sample window. This is the Rx MByte/sec multiplied by 8.

BFM Configuration Procedures

The following procedures are available in `altpcieth_bfm_configure`. These procedures support configuration of the root port and endpoint configuration space registers.

All VHDL arguments are subtype `natural` and are input-only unless specified otherwise. 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 7-33](#) for a description the arguments for this procedure.

Table 7-33. `ebfm_cfg_rp_ep` Procedure

Location	altpcieth_bfm_configure.v or altpcieth_bfm_configure.vhd	
Syntax	<code>ebfm_cfg_rp_ep(bar_table, ep_bus_num, ep_dev_num, rp_max_rd_req_size, display_ep_config, addr_map_4GB_limit)</code>	
Arguments	<code>bar_table</code>	Address of the endpoint <code>bar_table</code> structure in BFM shared memory. This routine populates the <code>bar_table</code> structure. The <code>bar_table</code> structure stores the size of each BAR and the address values assigned to each BAR. The address of the <code>bar_table</code> structure is passed to all subsequent read and write procedure calls that access an offset from a particular BAR.
	<code>ep_bus_num</code>	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.
	<code>ep_dev_num</code>	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.
	<code>rp_max_rd_req_size</code>	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 <code>MAXIMUM_PAYLOAD_SIZE</code> , 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.
	<code>display_ep_config</code>	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.
	<code>addr_map_4GB_limit</code>	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 7-34. `ebfm_cfg_decode_bar` Procedure

Location	<code>altpcieth_bfm_configure.v</code> or <code>altpcieth_bfm_configure.vhd</code>	
Syntax	<code>ebfm_cfg_decode_bar(bar_table, bar_num, log2_size, is_mem, is_pref, is_64b)</code>	
Arguments	<code>bar_table</code>	Address of the endpoint <code>bar_table</code> structure in BFM shared memory.
	<code>bar_num</code>	BAR number to analyze.
	<code>log2_size</code>	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.
	<code>is_mem</code>	The procedure sets this argument to indicate if the BAR is a memory space BAR (1) or I/O Space BAR (0).
	<code>is_pref</code>	The procedure sets this argument to indicate if the BAR is a prefetchable BAR (1) or non-prefetchable BAR (0).
	<code>is_64b</code>	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 available in the VHDL file `altpcieth_bfm_shmem.vhd` or in the Verilog HDL include file `altpcieth_bfm_shmem.v` that uses the module `altpcieth_bfm_shmem_common.v`, instantiated at the top level of the testbench. These procedures and functions support accessing the BFM shared memory.

All VHDL arguments are subtype `natural` and are input-only unless specified otherwise. All Verilog HDL arguments are type `integer` and are input-only unless specified otherwise.

Shared Memory Constants

The following constants are defined in the BFM shared memory package. They select a data pattern in the `shmem_fill` and `shmem_chk_ok` routines. These shared memory constants are all VHDL subtype `natural` or Verilog HDL type `integer`.

Table 7-35. Constants: VHDL Subtype `NATURAL` or Verilog HDL Type `INTEGER`

Constant	Description
<code>SHMEM_FILL_ZEROS</code>	Specifies a data pattern of all zeros
<code>SHMEM_FILL_BYTE_INC</code>	Specifies a data pattern of incrementing 8-bit bytes (0x00, 0x01, 0x02, etc.)
<code>SHMEM_FILL_WORD_INC</code>	Specifies a data pattern of incrementing 16-bit words (0x0000, 0x0001, 0x0002, etc.)
<code>SHMEM_FILL_DWORD_INC</code>	Specifies a data pattern of incrementing 32-bit dwords (0x00000000, 0x00000001, 0x00000002, etc.)
<code>SHMEM_FILL_QWORD_INC</code>	Specifies a data pattern of incrementing 64-bit qwords (0x0000000000000000, 0x0000000000000001, 0x0000000000000002, etc.)
<code>SHMEM_FILL_ONE</code>	Specifies a data pattern of all ones

shmem_write

The `shmem_write` procedure writes data to the BFM shared memory.

Table 7-36. `shmem_write` VHDL Procedure or Verilog HDL Task

Location	altpcieth_bfm_shmem.v or altpcieth_bfm_shmem.vhd	
Syntax	<code>shmem_write(addr, data, leng)</code>	
Arguments	<code>addr</code>	BFM shared memory starting address for writing data
	<code>data</code>	Data to write to BFM shared memory. In VHDL, this argument is an unconstrained <code>std_logic_vector</code> . This vector must be 8 times the <code>leng</code> length. In Verilog, this parameter is implemented as a 64-bit vector. <code>leng</code> is 1–8 bytes. In both languages, bits 7 downto 0 are written to the location specified by <code>addr</code> ; bits 15 downto 8 are written to the <code>addr+1</code> location, etc.
	<code>leng</code>	Length, in bytes, of data written

shmem_read Function

The `shmem_read` function reads data to the BFM shared memory.

Table 7-37. `shmem_read` Function

Location	altpcieth_bfm_shmem.v or altpcieth_bfm_shmem.vhd	
Syntax	<code>data := shmem_read(addr, leng)</code>	
Arguments	<code>addr</code>	BFM shared memory starting address for reading data
	<code>leng</code>	Length, in bytes, of data read
Return	<code>data</code>	Data read from BFM shared memory. In VHDL, this is an unconstrained <code>std_logic_vector</code> , in which the vector is 8 times the <code>leng</code> length. In Verilog, this parameter is implemented as a 64-bit vector. <code>leng</code> is 1–8 bytes. If <code>leng</code> is less than 8 bytes, only the corresponding least significant bits of the returned data are valid. In both languages, bits 7 downto 0 are read from the location specified by <code>addr</code> ; bits 15 downto 8 are read from the <code>addr+1</code> location, etc.

shmem_display VHDL Procedure or Verilog HDL Function

The `shmem_display` VHDL procedure or Verilog HDL function displays a block of data from the BFM shared memory.

Table 7-38. `shmem_display` VHDL Procedure/ or Verilog Function

Location	altpcieth_bfm_shmem.v or altpcieth_bfm_shmem.vhd	
Syntax	VHDL: <code>shmem_display(addr, leng, word_size, flag_addr, msg_type)</code> Verilog HDL: <code>dummy_return := shmem_display(addr, leng, word_size, flag_addr, msg_type);</code>	
Arguments	<code>addr</code>	BFM shared memory starting address for displaying data.
	<code>leng</code>	Length, in bytes, of data to display.
	<code>word_size</code>	Size of the words to display. Groups individual bytes into words. Valid values are 1, 2, 4, and 8.

Table 7-38. shmem_display VHDL Procedure/ or Verilog Function

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 7-46 for more information about message types. Set to one of the constants defined in Table 7-41 on page 7-46.

shmem_fill Procedure

The shmem_fill procedure fills a block of BFM shared memory with a specified data pattern.

Table 7-39. shmem_fill Procedure

Location	altpcieth_bfm_shmem.v or altpcieth_bfm_shmem.vhd	
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 7-43.
	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 In VHDL. This argument is type std_logic_vector(63 downto 0). In Verilog HDL, this argument is reg [63:0]. In both languages, 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 7-40. shmem_chk_ok Function

Location	altpcieth_bfm_shmem.v or altpcieth_bfm_shmem.vhd	
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 7-43.
	leng	Length, in bytes, of data to check.
	init	In VHDL, this argument is type std_logic_vector(63 downto 0). In Verilog HDL, this argument is reg [63:0]. In both languages, 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 VHDL type Boolean. TRUE—Data pattern compared successfully FALSE—Data pattern did not compare successfully Result in Verilog HDL 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 VHDL package file `altpcieth_bfm_log.vhd` or in the Verilog HDL include file `altpcieth_bfm_log.v` that uses the `altpcieth_bfm_log_common.v` module, instantiated at the top level of the testbench.

These procedures provide support for displaying messages in a common format, suppressing informational messages, and stopping simulation on specific message types.

Log Constants

The following constants are defined in the BFM Log package. They 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 7-41](#).

You can suppress the display of certain message types. The default values determining whether a message type is displayed are defined in [Table 7-41](#). 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 7-41](#) 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 are VHDL subtype `natural` or type `integer` for Verilog HDL.

Table 7-41. Log Messages Using VHDL Constants - Subtype Natural (Part 1 of 2)

Constant (Message Type)	Description	Mask Bit No	Display by Default	Simulation Stops by Default	Message Prefix
<code>EBFM_MSG_DEBUG</code>	Specifies debug messages.	0	No	No	DEBUG :
<code>EBFM_MSG_INFO</code>	Specifies informational messages, such as configuration register values, starting and ending of tests.	1	Yes	No	INFO :
<code>EBFM_MSG_WARNING</code>	Specifies warning messages, such as tests being skipped due to the specific configuration.	2	Yes	No	WARNING :
<code>EBFM_MSG_ERROR_INFO</code>	Specifies additional information for an error. Use this message to display preliminary information before an error message that stops simulation.	3	Yes	No	ERROR :
<code>EBFM_MSG_ERROR_CONTINUE</code>	Specifies a recoverable error that allows simulation to continue. Use this error for data mismatches.	4	Yes	No	ERROR :

Table 7-41. Log Messages Using VHDL Constants - Subtype Natural (Part 2 of 2)

Constant (Message Type)	Description	Mask Bit No	Display by Default	Simulation Stops by Default	Message Prefix
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 VHDL Procedure or 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 7-42. `ebfm_display` Procedure

Location	altpcieth_bfm_log.v or altpcieth_bfm_log.vhd	
Syntax	VHDL: <code>ebfm_display(msg_type, message)</code> Verilog HDL: <code>dummy_return:=ebfm_display(msg_type, message);</code>	
Argument	<code>msg_type</code>	Message type for the message. Should be one of the constants defined in Table 7-41 on page 7-46 .
	<code>message</code>	In VHDL, this argument is VHDL type <code>string</code> and contains the message text to be displayed. In Verilog HDL, 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	<code>always 0</code>	Applies only to the Verilog HDL routine.

ebfm_log_stop_sim VHDL Procedure or Verilog HDL Function

The `ebfm_log_stop_sim` procedure stops the simulation.

Table 7-43. `ebfm_log_stop_sim` Procedure

Location	altpcieth_bfm_log.v or altpcieth_bfm_log.vhd	
Syntax	VHDL: <code>ebfm_log_stop_sim(success)</code> Verilog VHDL: <code>return:=ebfm_log_stop_sim(success);</code>	
Argument	<code>success</code>	When set to a 1, this process stops the simulation with a message indicating successful completion. The message is prefixed with <code>SUCCESS:</code> . Otherwise, this process stops the simulation with a message indicating unsuccessful completion. The message is prefixed with <code>FAILURE:</code> .
Return	Always 0	This value applies only to the Verilog HDL function.

ebfm_log_set_suppressed_msg_mask Procedure

The `ebfm_log_set_suppressed_msg_mask` procedure controls which message types are suppressed.

Table 7-44. `ebfm_log_set_suppressed_msg_mask` Procedure

Location	altpcieth_bfm_log.v or altpcieth_bfm_log.vhd	
Syntax	<code>bfm_log_set_suppressed_msg_mask (msg_mask)</code>	
Argument	<code>msg_mask</code>	In VHDL, this argument is a subtype of <code>std_logic_vector</code> , <code>EBFM_MSG_MASK</code> . This vector has a range from <code>EBFM_MSG_ERROR_CONTINUE</code> downto <code>EBFM_MSG_DEBUG</code> . In Verilog HDL, this argument is <code>reg [EBFM_MSG_ERROR_CONTINUE : EBFM_MSG_DEBUG]</code> . In both languages, a 1 in a specific bit position of the <code>msg_mask</code> causes messages of the type corresponding to the bit position to be suppressed.

ebfm_log_set_stop_on_msg_mask Procedure

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 7-41 on page 7-46](#).

Table 7-45. `ebfm_log_set_stop_on_msg_mask` Procedure

Location	altpcieth_bfm_log.v or altpcieth_bfm_log.vhd	
Syntax	<code>ebfm_log_set_stop_on_msg_mask (msg_mask)</code>	
Argument	<code>msg_mask</code>	In VHDL, this argument is a subtype of <code>std_logic_vector</code> , <code>EBFM_MSG_MASK</code> . This vector has a range from <code>EBFM_MSG_ERROR_CONTINUE</code> downto <code>EBFM_MSG_DEBUG</code> . In Verilog HDL, this argument is <code>reg [EBFM_MSG_ERROR_CONTINUE : EBFM_MSG_DEBUG]</code> . In both languages, a 1 in a specific bit position of the <code>msg_mask</code> causes messages of the type corresponding to the bit position to stop the simulation after the message is displayed.

ebfm_log_open Procedure

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 7-46. `ebfm_log_open` Procedure

Location	altpcieth_bfm_log.v or altpcieth_bfm_log.vhd	
Syntax	<code>ebfm_log_open (fn)</code>	
Argument	<code>fn</code>	This argument is type <code>string</code> and provides the file name of log file to be opened.

ebfm_log_close Procedure

The `ebfm_log_close` procedure closes the log file opened by a previous call to `ebfm_log_open`.

Table 7-47. `ebfm_log_close` Procedure

Location	altpcieth_bfm_log.v or altpcieth_bfm_log.vhd	
Syntax	<code>ebfm_log_close</code>	
Argument	NONE	

VHDL Formatting Functions

The following procedures and functions are available in the VHDL package file **altpcieth_bfm_log.vhd**. This section outlines formatting functions that are only used by VHDL. They take a numeric value and return a string to display the value.

himage (std_logic_vector) Function

The `himage` function is a utility routine that returns a hexadecimal string representation of the `std_logic_vector` argument. The string is the length of the `std_logic_vector` divided by four (rounded up). You can control the length of the string by padding or truncating the argument as needed.

Table 7-48. `himage (std_logic_vector)` Function

Location	altpcieth_bfm_log.vhd	
Syntax	<code>string:= himage (vec)</code>	
Argument	<code>vec</code>	This argument is a <code>std_logic_vector</code> that is converted to a hexadecimal string.
Return	<code>string</code>	Hexadecimal formatted string representation of the argument

himage (integer) Function

The `himage` function is a utility routine that returns a hexadecimal string representation of the integer argument. The string is the length specified by the `hlen` argument.

Table 7-49. `himage (integer) Function`

Location	altpcieth_bfm_log.vhd	
Syntax	<code>string := himage (num, hlen)</code>	
Arguments	<code>num</code>	Argument of type <code>integer</code> that is converted to a hexadecimal string.
	<code>hlen</code>	Length of the returned string. The string is truncated or padded with 0s on the right as needed.
Return	<code>string</code>	Hexadecimal formatted string representation of the argument.

Verilog HDL Formatting Functions

The following procedures and functions are available in the Verilog HDL include file **altpcieth_bfm_log.v** that uses the **altpcieth_bfm_log_common.v** module, instantiated at the top level of the testbench. 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 7-50. `himage1`

Location	altpcieth_bfm_log.v	
syntax	<code>string := himage (vec)</code>	
Argument	<code>vec</code>	Input data type <code>reg</code> with a range of 3:0.
Return range	<code>string</code>	Returns a 1-digit hexadecimal representation of the input argument. Return data is type <code>reg</code> 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 7-51. `himage2`

Location	altpcieth_bfm_log.v	
syntax	<code>string := himage (vec)</code>	
Argument range	<code>vec</code>	Input data type <code>reg</code> with a range of 7:0.
Return range	<code>string</code>	Returns a 2-digit hexadecimal presentation of the input argument, padded with leading 0s, if they are needed. Return data is type <code>reg</code> 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 7-52. himage4

Location	altpcieth_bfm_log.v	
syntax	<code>string:= himage(vec)</code>	
Argument range	<code>vec</code>	Input data type <code>reg</code> 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 <code>reg</code> 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 7-53. himage8

Location	altpcieth_bfm_log.v	
syntax	<code>string:= himage(vec)</code>	
Argument range	<code>vec</code>	Input data type <code>reg</code> with a range of 31:0.
Return range	<code>string</code>	Returns an 8-digit hexadecimal representation of the input argument, padded with leading 0s, if they are needed. Return data is type <code>reg</code> 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 7-54. himage16

Location	altpcieth_bfm_log.v	
syntax	<code>string:= himage(vec)</code>	
Argument range	<code>vec</code>	Input data type <code>reg</code> with a range of 63:0.
Return range	<code>string</code>	Returns a 16-digit hexadecimal representation of the input argument, padded with leading 0s, if they are needed. Return data is type <code>reg</code> with a range of 128:1.

dimage1

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 7-55. dimage1

Location	altpcieth_bfm_log.v	
syntax	<code>string:= dimage(vec)</code>	
Argument range	vec	Input data type <code>reg</code> 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 <code>reg</code> with a range of 8:1. Returns the letter <i>U</i> if the value cannot be represented.

dimage2

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 7-56. dimage2

Location	altpcieth_bfm_log.v	
syntax	<code>string:= dimage(vec)</code>	
Argument range	vec	Input data type <code>reg</code> 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 <code>reg</code> with a range of 16:1. Returns the letter <i>U</i> if the value cannot be represented.

dimage3

This function creates a three-digit hexadecimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Table 7-57. dimage3

Location	altpcieth_bfm_log.v	
syntax	<code>string:= dimage(vec)</code>	
Argument range	vec	Input data type <code>reg</code> 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 <code>reg</code> with a range of 24:1. Returns the letter <i>U</i> if the value cannot be represented.

dimage4

This function creates a four-digit hexadecimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Table 7-58. dimage4

Location	altpcieth_bfm_log.v	
syntax	<code>string:= dimage(vec)</code>	
Argument range	vec	Input data type <code>reg</code> 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 <code>reg</code> with a range of 32:1. Returns the letter <i>U</i> if the value cannot be represented.

dimage5

This function creates a five-digit hexadecimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Table 7-59. dimage5

Location	altpcieth_bfm_log.v	
syntax	<code>string:= dimage(vec)</code>	
Argument range	vec	Input data type <code>reg</code> 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 <code>reg</code> with a range of 40:1. Returns the letter <i>U</i> if the value cannot be represented.

dimage6

This function creates a six-digit hexadecimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Table 7-60. dimage6

Location	altpcieth_bfm_log.v	
syntax	<code>string:= dimage(vec)</code>	
Argument range	vec	Input data type <code>reg</code> 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 <code>reg</code> with a range of 48:1. Returns the letter <i>U</i> if the value cannot be represented.

dimage7

This function creates a seven-digit hexadecimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Table 7-61. `dimage7`

Location	altpcieth_bfm_log.v	
Syntax	<code>string:= dimage(vec)</code>	
Argument range	<code>vec</code>	Input data type <code>reg</code> with a range of 31:0.
Return range	<code>string</code>	Returns a 7-digit decimal representation of the input argument that is padded with leading 0s if necessary. Return data is type <code>reg</code> 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 VHDL entity file **altpcieth_bfm_driver_chaining.vhd** or the Verilog HDL module file **altpcieth_bfm_driver_chaining.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 7-62. `chained_dma_test` Procedure

Location	altpcieth_bfm_driver_chaining.v or altpcieth_bfm_driver_chaining.vhd	
Syntax	<code>chained_dma_test (bar_table, bar_num, direction, use_msi, use_eplast)</code>	
Arguments	<code>bar_table</code>	Address of the endpoint <code>bar_table</code> structure in BFM shared memory.
	<code>bar_num</code>	BAR number to analyze.
	<code>direction</code>	When 0 the direction is read. When 1 the direction is write.
	<code>Use_msi</code>	When set, the root port uses native PCI Express MSI to detect the DMA completion.
	<code>Use_eplast</code>	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 7-63. dma_rd_test Procedure

Location	altpcieth_bfm_driver_chaining.v or altpcieth_bfm_driver_chaining.vhd	
Syntax	<code>dma_rd_test (bar_table, bar_num, use_msi, use_eplast)</code>	
Arguments	<code>bar_table</code>	Address of the endpoint <code>bar_table</code> structure in BFM shared memory.
	<code>bar_num</code>	BAR number to analyze.
	<code>Use_msi</code>	When set, the root port uses native PCI express MSI to detect the DMA completion.
	<code>Use_eplast</code>	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 7-64. dma_wr_test Procedure

Location	altpcieth_bfm_driver_chaining.v or altpcieth_bfm_driver_chaining.vhd	
Syntax	<code>dma_wr_test (bar_table, bar_num, use_msi, use_eplast)</code>	
Arguments	<code>bar_table</code>	Address of the endpoint <code>bar_table</code> structure in BFM shared memory.
	<code>bar_num</code>	BAR number to analyze.
	<code>Use_msi</code>	When set, the root port uses native PCI Express MSI to detect the DMA completion.
	<code>Use_eplast</code>	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 7-65. dma_set_rd_desc_data Procedure

Location	altpcieth_bfm_driver_chaining.v or altpcieth_bfm_driver_chaining.vhd	
Syntax	<code>dma_set_rd_desc_data (bar_table, bar_num)</code>	
Arguments	<code>bar_table</code>	Address of the endpoint <code>bar_table</code> structure in BFM shared memory.
	<code>bar_num</code>	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 7-66. dma_set_wr_desc_data_header Procedure

Location	altpcieth_bfm_driver_chaining.v or altpcieth_bfm_driver_chaining.vhd	
Syntax	<code>dma_set_wr_desc_data_header (bar_table, bar_num)</code>	
Arguments	<code>bar_table</code>	Address of the endpoint <code>bar_table</code> structure in BFM shared memory.
	<code>bar_num</code>	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 7-67. dma_set_header Procedure

Location	altpcieth_bfm_driver_chaining.v or altpcieth_bfm_driver_chaining.vhd	
Syntax	<code>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)</code>	
Arguments	<code>bar_table</code>	Address of the endpoint <code>bar_table</code> structure in BFM shared memory.
	<code>bar_num</code>	BAR number to analyze.
	<code>Descriptor_size</code>	Number of descriptor.
	<code>direction</code>	When 0 the direction is read. When 1 the direction is write.
	<code>Use_msi</code>	When set, the root port uses native PCI Express MSI to detect the DMA completion.
	<code>Use_eplast</code>	When set, the root port uses BFM shared memory polling to detect the DMA completion.
	<code>Bdt_msb</code>	BFM shared memory upper address value.
	<code>Bdt_lsb</code>	BFM shared memory lower address value.
	<code>Msi_number</code>	When <code>use_msi</code> is set, specifies the number of the MSI which is set by the <code>dma_set_msi</code> procedure.
	<code>Msi_traffic_class</code>	When <code>use_msi</code> is set, specifies the MSI traffic class which is set by the <code>dma_set_msi</code> procedure.
	<code>Multi_message_enable</code>	When <code>use_msi</code> is set, specifies the MSI traffic class which is set by the <code>dma_set_msi</code> procedure.

rc_mempoll Procedure

Use the `rc_mempoll` procedure to poll a given DWORD in a given BFM shared memory location.

Table 7-68. rc_mempoll Procedure

Location	altpcieth_bfm_driver_chaining.v or altpcieth_bfm_driver_chaining.vhd	
Syntax	<code>rc_mempoll (rc_addr, rc_data, rc_mask)</code>	
Arguments	<code>rc_addr</code>	Address of the BFM shared memory that is being polled.
	<code>rc_data</code>	Expected data value of the that is being polled.
	<code>rc_mask</code>	Mask that is logically ANDed with the shared memory data before it is compared with <code>rc_data</code> .

msi_poll Procedure

The `msi_poll` procedure tracks MSI completion from the endpoint.

Table 7-69. msi_poll Procedure

Location	altpcieth_bfm_driver_chaining.v or altpcieth_bfm_driver_chaining.vhd	
Syntax	<code>msi_poll(max_number_of_msi,msi_address,msi_expected_dmawr,msi_expected_dmard,dma_write,dma_read)</code>	
Arguments	<code>max_number_of_msi</code>	Specifies the number of MSI interrupts to wait for.
	<code>msi_address</code>	The shared memory location to which the MSI messages will be written.
	<code>msi_expected_dmawr</code>	When <code>dma_write</code> is set, this specifies the expected MSI data value for the write DMA interrupts which is set by the <code>dma_set_msi</code> procedure.
	<code>msi_expected_dmard</code>	When the <code>dma_read</code> is set, this specifies the expected MSI data value for the read DMA interrupts which is set by the <code>dma_set_msi</code> procedure.
	<code>Dma_write</code>	When set, poll for MSI from the DMA write module.
	<code>Dma_read</code>	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 7-70. dma_set_msi Procedure

Location	altpcieth_bfm_driver_chaining.v or altpcieth_bfm_driver_chaining.vhd	
Syntax	<code>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)</code>	
Arguments	<code>bar_table</code>	Address of the endpoint <code>bar_table</code> structure in BFM shared memory.
	<code>bar_num</code>	BAR number to analyze.
	<code>Bus_num</code>	Set configuration bus number.
	<code>dev_num</code>	Set configuration device number.
	<code>Fun_num</code>	Set configuration function number.
	<code>Direction</code>	When 0 the direction is read. When 1 the direction is write.
	<code>msi_address</code>	Specifies the location in shared memory where the MSI message data will be stored.
	<code>msi_data</code>	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.
	<code>Msi_number</code>	Returns the MSI number to be used for these interrupts.
	<code>Msi_traffic_class</code>	Returns the MSI traffic class value.
	<code>Multi_message_enable</code>	Returns the MSI multi message enable status.
	<code>msi_expected</code>	Returns the expected MSI data value, which is <code>msi_data</code> modified by the <code>msi_number</code> chosen.

find_mem_bar Procedure

The `find_mem_bar` procedure locates a BAR which satisfies a given memory space requirement.

Table 7-71. find_mem_bar Procedure

Location	altpcieth_bfm_driver_chaining.v	
Syntax	<code>Find_mem_bar(bar_table, allowed_bars, min_log2_size, sel_bar)</code>	
Arguments	<code>bar_table</code>	Address of the endpoint <code>bar_table</code> structure in BFM shared memory
	<code>allowed_bars</code>	One hot 6 bits BAR selection
	<code>min_log2_size</code>	Number of bit required for the specified address space
	<code>sel_bar</code>	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 7-72. dma_set_rclast Procedure

Location	altpcieth_bfm_driver_chaining.v	
Syntax	<code>Dma_set_rclast(bar_table, setup_bar, dt_direction, dt_rclast)</code>	
Arguments	<code>bar_table</code>	Address of the endpoint <code>bar_table</code> structure in BFM shared memory
	<code>setup_bar</code>	BAR number to use
	<code>dt_direction</code>	When 0 read, When 1 write
	<code>dt_rclast</code>	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 7-73. ebfm_display_verb Procedure

Location	altpcieth_bfm_driver_chaining.v	
Syntax	<code>ebfm_display_verb(msg_type, message)</code>	
Arguments	<code>msg_type</code>	Message type for the message. Should be one of the constants defined in Table 7-41 on page 7-46 .
	<code>message</code>	In VHDL, this argument is VHDL type <code>string</code> and contains the message text to be displayed. In Verilog HDL, 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.

This design example provides detailed step-by-step instructions to generate an SOPC Builder system containing the following components:

- PCI Express ×4 MegaCore function
- On-Chip memory
- DMA controller

In the SOPC Builder design flow you select the PCI Express MegaCore function as a component, which automatically instantiates the PCI Express Compiler's Avalon-MM bridge module. This component supports PCI Express ×1 or ×4 endpoint applications with bridging logic to convert PCI Express packets to Avalon-MM transactions and vice versa. [Figure 8–1](#) shows a PCI Express system that includes three different endpoints created using the SOPC Builder design flow. It shows both the soft and hard IP implementations with one of the soft IP variants using the embedded transceiver and the other using a PIPE interface to an external PHY. The design example included in this chapter illustrates the use of a single hard IP implementation with the embedded transceiver.

Figure 8–1. SOPC Builder Example System with Multiple PCI Express MegaCore Functions

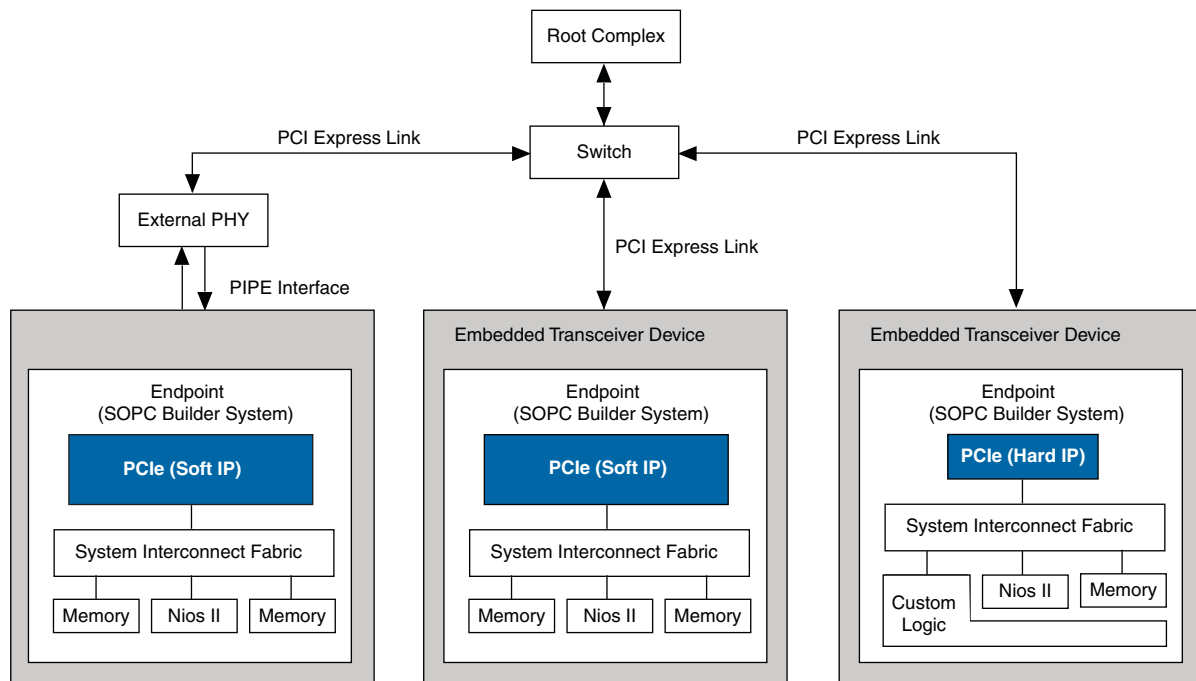
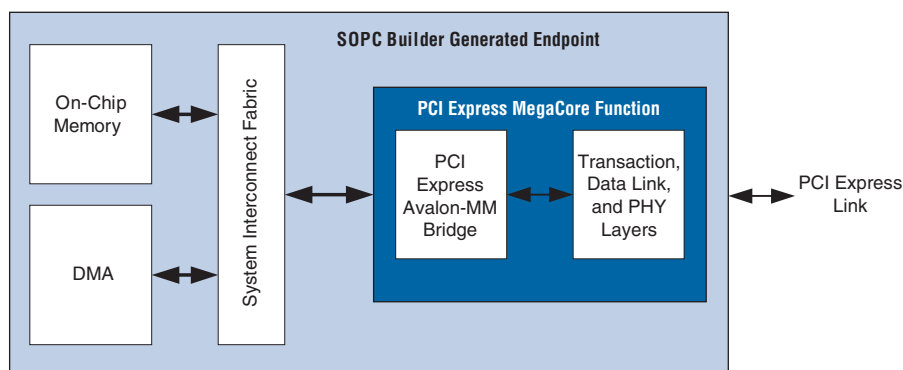


Figure 8–2 shows how SOPC Builder integrates components and the PCI Express MegaCore function using the system interconnect fabric. This design example transfers data between an on-chip memory buffer located on the Avalon-MM side and a PCI Express memory buffer located on the root complex side. The data transfer uses the DMA component which is programmed by the PCI Express software application running on the root complex processor.

Figure 8–2. SOPC Builder Generated Endpoint



This design example uses Verilog HDL. You can substitute VHDL for Verilog HDL.

This design example consists of the following steps:

1. [Create a Quartus II Project](#)
2. [Run SOPC Builder](#)
3. [Parameterize the PCI Express MegaCore Function](#)
4. [Add the Remaining Components to the SOPC Builder System](#)
5. [Complete the Connections in SOPC Builder](#)
6. [Generate the SOPC Builder System](#)
7. [Simulate the SOPC Builder System](#)
8. [Compile the Design](#)

Create a Quartus II Project

You must 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. 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 Project Wizard**.
3. Click **Next** in the **New Project Wizard: Introduction** (the introduction does not display if you turned it off previously).

4. In the **Directory, Name, Top-Level Entity** page, enter the following information:
 - a. Specify the working directory for your project. This design example uses the directory **c:\sopc_pcie**.
 - b. Specify the name of the project. This design example uses **pcie_top**. You must specify the same name for both the project and the top-level design entity.



The Quartus II software specifies a top-level design entity that has the same name as the project automatically. Do not change this name.



Click **Yes**, if prompted, to create a new directory.

5. Click **Next** to display the **Add Files** page.
6. If you have any non-default libraries, add them by following these steps:
 - a. Click **User Libraries**.
 - b. Type **<path>\ip** in the **Library name** box, where **<path>** is the directory in which you installed the PCI Express Compiler.
 - c. Click **Add** to add the path to the Quartus II project.
 - d. Click **OK** to save the library path in the project.
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 **Stratix IV GX**.



This design example creates a design targeting the Stratix IV GX device family. You can also use these procedures for other supported device families.

- b. In the **Target device** box, select **Auto device selected by the Fitter**.
9. Click **Next** to close this page and display the **EDA Tool Settings** page.
10. Click **Next** to display the **Summary** page.
11. Check the **Summary** page to ensure that you have entered all the information correctly.
12. Click **Finish** to complete the Quartus II project.

Run SOPC Builder

To launch the PCI Express MegaWizard interface in SOPC Builder, follow these steps:

1. On the Tools menu, click **SOPC Builder**. SOPC Builder appears.



Refer to Quartus II Help for more information about how to use SOPC Builder.

2. Type **pcie_top** in the **System Name** box, select **Verilog** under **Target HDL**, and click **OK**.



In this example, you are choosing the SOPC Builder-generated system file to be the same as the project's top level file. This is not required for your own design. If you want to choose different name for the system file, you must create a wrapper HDL file of the same name as the project's top level and instantiate the generated system.

3. To build your system by adding modules from the **System Contents** tab, under **Interface Protocols** in the **PCI** folder, double-click the **PCI Express Compiler** component.

Parameterize the PCI Express MegaCore Function

To parameterize the PCI Express MegaCore function in SOPC Builder, follow these steps:

1. On the **System Settings** page, specify the settings in [Table 8-1](#).

Table 8-1. System Settings Page

Parameter	Value
PCIe Core Type	PCI Express hard IP
PHY type	Stratix IV GX
Lanes	×4
PCI Express version	1.1
Test out width	9 bits

2. On the **PCI Registers** page, specify the settings in [Table 8-2](#).

Table 8-2. PCI Registers Page

PCI Base Address Registers (Type 0 Configuration Space)			
BAR	BAR Type	BAR Size	Avalon Base Address
1:0	64-bit Prefetchable Memory	Auto	Auto
2	32-bit Non-Prefetchable Memory	Auto	Auto

3. Click the **Avalon Configuration** page and specify the settings in [Table 8-3](#).

Table 8-3. Avalon Configuration Page

Parameter	Value
Avalon Clock Domain	Use separate clock
PCIe Peripheral Mode	Requester/Completer
Address Translation Table Configuration	Dynamic translation table
Address Translation Table Size	
Number of address pages	2
Size of address pages	1 MByte - 20 bits

For an example of a system that uses the PCI Express core clock for the Avalon clock domain see [Figure 4-24 on page 4-67](#).

4. Click **Finish** to close the MegaWizard interface and return to SOPC Builder.



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

Add the Remaining Components to the SOPC Builder System

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

1. In the **System Contents** tab, double-click **DMA Controller** in the **DMA** subfolder of the **Memories and Memory Controllers** folder. This component contains read and write master ports and a control port slave.
2. In the **DMA Controller** wizard, specify the parameters or conditions listed in [Table 8-4](#).

Table 8-4. DMA Controller Wizard

Parameter	Value
Width of the DMA length register	13
Enable burst transfers	Turn this option on
Maximum burst size	Select 1024
Construct FIFO from embedded memory blocks	Turn this option on

3. Click **Finish**. The DMA Controller module is added to your SOPC Builder system.
4. In the **System Contents** tab, double-click the **On-Chip Memory (RAM or ROM)** in the **On-Chip** subfolder of the **Memory and Memory Controllers** folder. This component contains a slave port.

Table 8-5. On-Chip Memory Wizard

Parameter	Value
Memory type	Select RAM (Writeable)
Block type	Select Auto
Initialize memory content	Turn this option off
Data width	Select 64-bit
Total memory size	Select 4096 Bytes

5. You can leave the rest of the settings at their default values.
6. Click **Finish**. The On-chip Memory component is added to your SOPC Builder system.

Complete the Connections in SOPC Builder

In SOPC Builder, hovering the mouse over the **Connections** column displays the potential connection points between components, represented as dots connecting wires. A filled dot shows that a connection is made; an open dot shows a potential connection point that is not currently connected. Clicking a dot toggles the connection status. To complete this design, create the following connections:

1. Connect the pci_express_compiler bar1_0_Prefetchable Avalon master port to the onchip_mem s1 Avalon slave port using the following procedure:
 - a. Click on the bar1_0_Prefetchable port then hover in the Connections column to display possible connections.
 - b. Click on the open dot at the intersection of the onchip_mem s1 port and the pci_express_compiler bar1_0_Prefetchable to create a connection.
2. Repeat step 1 to make the connections listed in Table 8-6.

Table 8-6. SOPC Builder Connections

Make Connection From:	To:
pci_express_compiler bar2_Non_Prefetchable Avalon master port	dma control_port_slave Avalon slave port
pci_express_compiler bar2_Non_Prefetchable Avalon master port	pci_express_compiler Control_Register_access Avalon slave port
dma_irq Interrupt sender	pci_express_compiler RxmIrq Interrupt Receiver
dma_read_master Avalon master port	onchip_mem s1 Avalon slave port
dma_read_master Avalon master port	pci_express_compiler Tx_Interface Avalon slave port
dma_write_master Avalon master port	onchip_mem s1 Avalon slave port
dma_write_master Avalon master port	pci_express_compiler Tx_Interface Avalon slave port

To complete the system, follow these instructions for clock and address assignments:

1. Under **Clock Settings**, double-click in the **MHz** box, type 125, and press Enter.
2. To add a second external clock, cal_clk, for calibration, follow these steps:
 - a. Under **Clock Settings**, click **Add**. A new clock, clk_1, appears in the **Name** box.
 - b. Double-click clk_1 and type cal_clk, then press Enter.
 - c. To specify the frequency, double-click the **MHz** box and type in the frequency you plan to use. cal_clk can have a frequency range of 10-125 MHz.

By default, clock names are not displayed. To display clock names in the Module Name column and the clocks in the Clock column in the **System Contents** tab, click **Filters** to display the **Filters** dialog box. In the **Filter** list, select **All**.

3. In the Clock column, connect cal_clk following these steps:
 - a. Click in the Clock column next to the cal_blk_clk port. A list of available clock signals appears.
 - b. Click cal_clk from the list of available clocks to connect the calibration clock (cal_blk_clk) of the pci_express_compiler.



All components using transceivers must have their cal_blk_clk connected to the same clock source.

4. To specify the interrupt number for DMA interrupt sender, irq, type a 0 in the IRQ column next to the irq port.

5. In the **Base** column, enter the base addresses in [Table 8-7](#) for all the slaves in your system:

Table 8-7. Base Addresses for Slave Ports

Port	Address
pci_express_compiler_0 Control_Register_Access	0x80004000
pci_express_compiler_0 Tx_Interface	0x00000000
dma_0 control_port_slave	0x80001000
onchip_memory2_0 s1	0x80000000

SOPC Builder generates informational messages indicating the actual PCI BAR settings.

For this example BAR1:0 is sized to 4 KBytes or 12 bits and PCI Express requests that match this BAR, are able to access the Avalon addresses from 0x80000000–0x80000FFF. BAR2 is sized to 32 KBytes or 15 bits and matching PCI Express requests are able to access Avalon addresses from 0x80000000–0x80007FFF. The DMA control_port_slave is accessible at offsets 0x1000 through 0x103F from the programmed BAR2 base address. The pci_express_compiler_0 Control_Register_Access slave port is accessible at offsets 0x4000–0x7FFF from the programmed BAR2 base address. Refer to [“PCI Express-to-Avalon-MM Address Translation” on page 4-23](#) for additional information on this address mapping.

For Avalon-MM accesses directed to the pci_express_compiler_0 Tx_interface port, Avalon-MM address bits 19-0 are passed through to the PCI Express address unchanged because a 1 MByte or 20-bit address page size was selected. Bit 20 is used to select which one of the 2 address translation table entries is used to provide the upper bits of the PCI Express address. Avalon address bits [31:21] are used to select the Tx_interface slave port. Refer to section [“Avalon-MM-to-PCI Express Address Translation Table” on page 4-47](#) for additional information on this address mapping.

[Table 8-6](#) illustrates the required connections.

Figure 8-3. Port Connections

Target
Device Family: **Stratix IV**

Clock Settings

Name	Source	MHz
clk_0	External	125.0
cal_clk	External	125.0

Use	Connections	Module Name	Description	Clock	Base	End	Tags	IRQ
<input checked="" type="checkbox"/>		pcie_compiler_0	PCI Express Compiler					
		avalon_clk	Clock Input	clk_0				
		cal_blk_clk	Clock Input	cal_clk				
		bar1_0_Prefetchable	Avalon Memory Mapped Master					
		bar2_Non_Prefetchable	Avalon Memory Mapped Master					
		Control_Register_Access	Avalon Memory Mapped Slave		0x80004000	0x80007fff		
		Crainq	Interrupt Sender					
		Rxmirq	Interrupt Receiver					
		Tx_Interface	Avalon Memory Mapped Slave		0x00000000	0x001fffff		
<input checked="" type="checkbox"/>		dma_0	DMA Controller					
		clk	Clock Input	clk_0				
		control_port_slave	Avalon Memory Mapped Slave		0x80001000	0x8000103f		
		irq	Interrupt Sender					
		read_master	Avalon Memory Mapped Master					
		write_master	Avalon Memory Mapped Master					
<input checked="" type="checkbox"/>		onchip_memory2_0	On-Chip Memory (RAM or ROM)					
		clk1	Clock Input	clk_0				
		s1	Avalon Memory Mapped Slave		0x80000000	0x80000fff		
<input checked="" type="checkbox"/>		cal_clk	Clock Source					
		clk	Clock Output	cal_clk				
<input checked="" type="checkbox"/>		clk_0	Clock Source					
		clk	Clock Output	clk_0				

Generate the SOPC Builder System

1. In SOPC Builder, click **Next**.
2. In the **System Generation** tab, turn on **Simulation**. Create project simulator files and click **Generate**. After the SOPC Builder generator reports successful system generation, click **Save**. You can now simulate the system using any Altera-supported third party simulator, compile the system in the Quartus II software, and configure an Altera device.

Simulate the SOPC Builder System

SOPC Builder automatically sets up the simulation environment for the generated system. SOPC Builder creates the **pcie_top_sim** subdirectory in your project directory and generates the required files and models to simulate your PCI Express system.

This section of the design example uses the following components:

- The system you created using SOPC Builder
- Simulation scripts created by SOPC Builder in the **c:\sopc_pcie\pcie_top_sim** directory
- The ModelSim-Altera Edition software



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

The PCI Express testbench files are located in the **c:\sopc_pcie\pci_express_compiler_examples\sopc\testbench** directory.

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



For more information about IP functional simulation models, refer to *Simulating Altera IP in Third-Party Simulation Tools* in volume 3 of the *Quartus II Handbook*.

The SOPC Builder-generated top-level file also integrates the simulation modules of the system components and testbenches (if available), including the PCI Express testbench. The Altera-provided PCI Express testbench simulates a single link at a time. You can use this testbench to verify the basic functionality of your PCI Express compiler system. The default configuration of the PCI Express testbench is predefined to run basic PCI Express configuration transactions to the PCI Express device in your SOPC Builder generated system. You can edit the PCI Express testbench **altpcieth_bfm_driver.v** or **altpcieth_bfm_driver.vhd** file to add other PCI Express transactions, such as memory read (MRd) and memory write (MWr).

For more information about the PCI Express BFM, refer to [Chapter 7, Testbench and Design Example](#).

For this design example, perform the following steps:

1. Before simulating the system, if you are running the Verilog HDL design example, edit the **altpcieth_bfm_driver.v** file in the **c:\sopc_pci\pci_express_compiler_examples\sopc\testbench** directory to enable target and DMA tests. Set the following parameters in the file to one:

- `parameter RUN_TGT_MEM_TST = 1;`
- `parameter RUN_DMA_MEM_TST = 1;`

If you are running the VHDL design example, edit the **altpcieth_bfm_driver.vhd** in the **c:\sopc_pci\pci_express_compiler_examples\sopc\testbench** directory to set the following parameters to one.

- `RUN_TGT_MEM_TST : std_logic := '1';`
- `RUN_DMA_MEM_TST : std_logic := '1';`



The target memory and DMA memory tests in the **altpcieth_bfm_driver.v** file enabled by these parameters only work with the SOPC Builder system as specified in this chapter. When designing an application, modify these tests to match your system.

2. Choose **Programs > ModelSim-Altera>6.3g ModelSim** (Windows Start menu) to start the ModelSim-Altera simulator. In the simulator change your working directory to **c:\sopc_pci\pci_top_sim**.
3. To run the script, type the following command at the simulator command prompt:

```
source setup_sim.do
```
4. To generate waveform output for the simulation, type the following command at the simulator command prompt:

```
do wave_presets.do
```



In ModelSim SE 6.3g, design optimization is on by default. Optimization may eliminate design nodes which are referenced in your **wave_presets.do** file. In this case, the **w** alias will fail. You can ignore this failure if you want to run an optimized simulation. However, if you want to see the simulation signals, you can disable the optimized compile by setting `VoptFlow = 0` in your **modelsim.ini** file.

5. To compile all the files and load the design, type the following command at the simulator prompt:

```
s ←
```

6. To simulate the design, type the following command at the simulator prompt:

```
run -all ←
```

The PCI Express Compiler test driver performs the following transactions with display status of the transactions displayed in the ModelSim simulation message window:

- Various configuration accesses to the PCI Express MegaCore function 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 4 KBytes of data from the Root Port BFM's shared memory
- Setup of the DMA controller to write the same 4 KBytes of data back to the Root Port BFM's shared memory
- Data comparison and report of any mismatch

7. Exit the ModelSim tool after it reports successful completion.

Compile the Design

You can use the Quartus II software to compile the system generated by SOPC Builder. Refer to Quartus II Help for more details about compiling your design.

To compile your design:

1. In the Quartus II software, open the project named **pcie_top.qpf** that you created for this design example.
2. On the View menu, point to **Utility Windows**, and then click **Tcl Console**.
3. To source the script that sets the required constraints, type the following command in the Tcl Console window:

```
source pci_express_compiler.tcl ←
```

4. On the Processing menu, click **Start Compilation**.

5. 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.

Program a Device

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

With Altera's free OpenCore Plus evaluation feature, you can evaluate the PCI Express Compiler before you purchase a license. OpenCore Plus evaluation allows you to generate an IP functional simulation model and produce a time-limited programming file.



For more information about IP functional simulation models, see the *Simulating Altera IP in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

Recommended Top-Down Incremental Compilation Flow

When using the incremental compilation flow, Altera recommends that you include a fully registered boundary on your application. By registering signals, you reserve the entire timing budget between the application and PCI Express MegaCore function for routing.



Refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* in volume 1 of the *Quartus II Handbook*.

The following is a suggested top-down incremental compile flow. The instructions cover incremental compilation for both the Avalon-ST and the descriptor/data interfaces.



Altera recommends disabling the OpenCore Plus feature when compiling with this flow. (On the Assignments menu, click **Settings**. Under **Compilation Process Settings**, click **More Settings**. Under **Disable OpenCore Plus hardware evaluation** select **On**.)

1. Open a Quartus II project.
2. On the Processing menu, point to **Start**, and then click **Start Analysis & Synthesis** to run initial logic synthesis on your top-level design. This step displays the design hierarchy in the Project Navigator.

3. Perform one of the following steps:
 - For Avalon-ST designs, in the **Project Navigator**, expand the `<variation_name>_icm` module as follows: `<variation_name>_example_top -> <variation_name>_example_pipen1b:core -> <variation_name>:epmap` and select **Set as Design Partition**.
 - For descriptor/data interface designs, in the **Project Navigator**, expand the `<variation_name>_icm` module as follows: `<variation_name>_example_top -> <variation_name>_example_pipen1b:core -> <variation_name>_icm:icm_epmap`. Right-click `<variation_name>_icm` and select **Set as Design Partition**.
4. On the Assignments menu, click **Design Partitions Window**. The design partition, **Partition_<variation_name>_** or **Partition_<variation_name>_icm** for descriptor/data designs, appears. Under **Netlist Type**, right-click and select **Post-Synthesis**.
5. To turn on incremental compilation, follow these steps:
 - a. On the Assignments menu, click **Settings**.
 - b. In the **Category** list, expand **Compilation Process Settings**.
 - c. Click **Incremental Compilation**.
 - d. Under **Incremental Compilation**, select **Full incremental compilation**.
6. To run a full compilation, on the Processing menu, click **Start Compilation**. Run Design Space Explorer (DSE) if required to achieve timing requirements. (On the Tools menu, click **Launch Design Space Explorer**.)
7. After timing is met, you can preserve the timing of the partition in subsequent compilations by using the following procedure:
 - a. On the Assignments menu, click **Design Partition Window**.
 - b. Under the **Netlist Type** for the **Top** Design Partition, double-click to select **Post-Fit(Strict)**.
 - c. Under **Fitter Preservation** level and double-click to select **Placement And Routing**.



Information for the partition netlist is saved in the **db** folder. Do not delete this folder.

Packet Format without Data Payload

Table A-2 through A-3 show the header format for transaction layer packets without a data payload. When these headers are transferred to and from the MegaCore function as tx_desc and rx_desc, the mapping shown in Table A-1 is used

Table A-1. Header Mapping

Header Byte	tx_desc/rx_desc Bits
Byte 0	127:120
Byte 1	119:112
Byte 2	111:104
Byte 3	103:96
Byte 4	95:88
Byte 5	87:80
Byte 6	79:72
Byte 7	71:64
Byte 8	63:56
Byte 9	55:48
Byte 10	47:40
Byte 11	39:32
Byte 12	31:24
Byte 13	23:16
Byte 14	15:8
Byte 15	7:0

Table A-2. 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	Att r	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, 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-4. 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	Att r		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. 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	Att r	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-6. 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-7. 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-8. Message without Data 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	1	1	0	r	r	r	0	TC			0	0	0	0	TD	EP	0	0	0	0	0	0	0	0	0	0	0	0	0	0
						2	1	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-8:																																
(1) Not supported in Avalon-MM.																																

Table A-9. 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	0	TD	EP	Att r	0	0	Length																			
Byte 4	Completer ID																Status		B	Byte Count																					
Byte 8	Requester ID																Tag								0	Lower Address															
Byte 12	Reserved																																								

Table A-10. 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	Att r	0	0	Length																		
Byte 4	Completer ID																Status				B	Byte Count																			
Byte 8	Requester ID																Tag								0	Lower Address															
Byte 12	Reserved																																								

Packet Format with Data Payload

Table A-11 through A-5 show the content for transaction layer packets with a data payload.

Table A-11. 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	Att r		0	0	Length											
Byte 4	Requester ID																Tag								Last BE				First BE					
Byte 8	Address [31:2]																														0 0			
Byte 12	Reserved																																	

Table A-12. 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	Att r		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-13. 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	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	Bus Number								Device No								0	0	0	0	Ext Reg				Register No				0 0			
Byte 12	Reserved																															

Table A-14. 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-15. 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	Att r	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. 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	Att r	0	0	Length									
Byte 4	Completer ID																Status		B	Byte Count												
Byte 8	Requester ID																Tag						0	Lower Address								
Byte 12	Reserved																															

Table A-17. 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	r	r	r	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																																				

The test port includes `test_out` and `test_in` signals, which add additional observability and control to the PCI Express MegaCore function.

- The output port offers a view of the internal node of the MegaCore function, providing information such as state machine status and error counters for each type of error.
- The input port can be used to configure the MegaCore function in a noncompliant fashion. For example, you can use it to inject errors for automated tests or to add capabilities such as remote boot and force or disable compliance mode.
- The signals can be used for status monitoring. Signals that change slowly are good candidates. The signals `LTSSM` and `DL_UP` are possibilities.



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 signals have not been rigorously verified and will not function as documented in some corner cases.

Hard IP Test Signals

The hard IP implementation of the PCI Express MegaCore function has `test_in` and `test_out` busses that are described in the following sections.

Hard IP Implementation—`test_in`

Table B–1 describes the `test_in` signals for the PCI Express hard IP implementation.

Table B–1. `test_in` Signals Hard IP Implementation (Part 1 of 3)

Signal	Bits	Dynamic (1)	Description
<code>test_in</code>	[0]	no	Simulation mode. This signal can be set to 1 to accelerate MegaCore function initialization by changing many initialization counter values from their PCI Express Specification values to much lower values.
	[1]	no	Loopback master: This signal must be set to 1 in order to direct the Link to loopback (in Master mode). Note that master loopback mode request is masked if hot reset or disable state are required by higher layer or by the opposite component. Static bit (*)
	[2]	no	Descramble mode: This signal must be set to 1 during initialization in order to disable data scrambling. Note that the disable scrambling request is always added in the transmitted TS1/TS2 except if master loopback, hot reset or disable state are required by higher layer or by the link.

Table B-1. Test_in Signals Hard IP Implementation (Part 2 of 3)

Signal	Bits	Dynamic (1)	Description
	[3]	no	<p>FPGA mode: This signal must be set to 1 for an FPGA implementation in order to appear compliant to other PCI Express components. The MegaCore function always detects the maximum number of receivers during the detect state and only goes to compliance state if at least one lane has the correct pattern. Used for masking a check of Electrical Idle Exit when going to Polling.Compliance, (in case of PHY issues).</p> <p>Note: This mode will prevent the LTSSM from moving to compliance state if an exit from electrical idle has never been detected on any lane which has detected a receiver during detect. The FPGA does not support receiver detection and always detects the maximum lane number of the MegaCore function. In this mode, the compliance state is only entered if no lanes have completed the normal exit condition when the timeout condition is reached.</p>
	[4]	no	Remote boot mode. When asserted, this signal disables the BAR check if the link is not initialized and the boot is located behind the component.
	[6:5]	no	<p>Compliance test mode. Disable/force compliance mode:</p> <ul style="list-style-type: none"> ■ bit 0—completely disables compliance mode; never enter compliance mode. ■ bit 1—forces compliance mode. orces entry to compliance mode when timeout is reached in polling.active state (and not all lanes have detected their exit condition).
	[7]	no	Disable low power state negotiation. When asserted, this signal disables all low power state negotiation.
	[15:8]	yes	<p>Test Out select: Selection of the 64 bits of the <code>test_out</code> port.</p> <ul style="list-style-type: none"> ■ Bits 11:8 selects the layer. Refer to Table B-2 for the encoding. ■ Bits 12 selects the VC ■ Bits 15:13 selects the Lane
	[16]	yes	Force TLP LCRC error detection: When asserted, this signal forces the MegaCore function to treat the next received TLP as if it had an LCRC error.
	[18:17]	yes	<p>Force DLLP CRC error detection: This signal forces the MegaCore function to check the next DLLP for a CRC error:</p> <ul style="list-style-type: none"> ■ 00: normal mode ■ 01: ACK/NAK ■ 10: PM ■ 11: FC
	[19]	yes	Force Retry Buffer. When asserted, this signal forces the Retry Buffer to initiate a retry.
	[21:20]	yes	<p>Replace ACK by NAK: This signal replaces an ACK by a NAK with following sequence number:</p> <ul style="list-style-type: none"> ■ 00: normal mode ■ 01: Same sequence number as the ACK ■ 10: Sequence number incriminated ■ 11: Sequence number decremented
	[22]	yes	Inject ECRC error on transmission: When asserted, this signal generates an ECRC error for transmission.
	[23]	yes	Inject LCRC error on transmission: When asserted, this signal generates an LCRC error for transmission.

Table B-1. Test_in Signals Hard IP Implementation (Part 3 of 3)

Signal	Bits	Dynamic (1)	Description
	[25:24]	yes	Inject DLLP CRC error on transmission: Generates a CRC error when transmitting a DLLP: <ul style="list-style-type: none"> 00: Normal 01: ACK error 10: PM error 11: FC error
	[30:26]	yes	Generate wrong value for Update FC. This signal forces an incorrect value when updating FC credits. It does so by adding or removing 1 credit in the credits allocated field when a TLP is extracted from receive buffer and sent to the application layer: <ul style="list-style-type: none"> 00000: normal mode 00001: UFC_P error on header (+1/0) 00010: UFC_P error on data (0/+1) 00011: UFC_P error on header/data (+1/+1) 00100: UFC_NP error on header (+1/0) 00101: UFC_NP error on data (0/+1) 00110: UFC_NP error on header/data (+1/+1) 00111: UFC_CPL error on header (+1/0) 01000: UFC_CPL error on data (0/+1) 01001: UFC_CPL error header/data (+1/+1) 01010: UFC_P error on header (-1/0) 01011: UFC_P error on data (0/-1) 01100: UFC_P error on header/data (-1/-1) 01101: UFC_NP error on header (-1/0) 01110: UFC_NP error on data (0/-1) 01111: UFC_NP error on header/data (-1/-1) 10000: UFC_CPL error on header (-1/0) 10001: UFC_CPL error on data (0/-1) 10010: UFC_CPL error header/data (-1/-1) 10011: UFC_P error on header/data (+1/-1) 10100: UFC_P error on header/data (-1/+1) 10101: UFC_NP error on header/data (+1/-1) 10110: UFC_NP error on header/data (-1/+1) 10111: UFC_CPL error header/data (+1/-1) 11000: UFC_CPL error header/data (-1/+1)
	[31]	no	Compliance Pattern Master. Sets the Compliance Receive bit.
	[39:32]	—	Reserved.

Note to Table B-1:

- (1) Dynamic bits can toggle during run time. A no in this column indicates that the bit should be forced to a constant value during reset and not toggled during run time.

Hard IP Implementation—test_out

The debug signals provided on test_out depends on the setting of test_in[11:8]. [Table B-2](#) provides the encoding for test_in[11:8] with a link to the test_out signals that correspond to this encoding.

Table B-2. test_in[11:8] Encoding

test_in[11:8] Value	Signal Group
4'b0000	“Transaction Layer” on page B-5
4'b0001	“Data Link Layer Outputs” on page B-8
4'b0010	“PHY-MAC Outputs” on page B-12
4'b0011	“PHY PCS layer” on page B-15
4'b0100	“PHY PCS layer” on page B-15
4'b0101	“CDC Test Outputs” on page B-15
4'b0110	“Mixed Mode Test Outputs” on page B-16
4'b0111	“Receive FC Test Outputs” on page B-20
4'b1000	“Transmit FC Test Outputs” on page B-20
4'b1001	“Request ID and Error Signal Test Outputs” on page B-21
4'b1010	“Receive DLL Data Test Outputs” on page B-22
4'b1011	“Miscellaneous Test Outputs” on page B-23

Transaction Layer

Table B-3 describes the transaction layer test output signals for the PCI Express hard IP implementation when the value of `test_in[11:8] = 4'b0000`. `test_in[12]` selects the VC.

Table B-3. Transaction Layer test_out Signals for the Hard IP Implementation when `test_in[11:8] = 4'b0000` (Part 1 of 4)

Bits	Signal Name	Description
Receive path		
[2:0]	rx_fval_tlp rx_hval_tlp rx_dval_tlp	Receive TLP reception state: These signals report the TLP reception sequencing <ul style="list-style-type: none"> bit 0: DW0 and DW1 of the header are valid bit 1: DW2 and DW3 of the header are valid bit 2: The data payload is valid
[10:3]	rx_check_tlp rx_discard_tlp rx_mlf_tlp tlp_err rxfc_ovf rx_ecrcerr_tlp rx_uns_tlp rx_sup_tlp	Receive TLP check state: These signals report the TLP reception sequencing: <ul style="list-style-type: none"> bit 0: Check LCRC bit 1: Indicates an LCRC error or sequence number error bit 2: Indicates a malformed TLP due to a mismatch END / length field bit 3: Indicates a malformed TLP that doesn't conform with formation rules bit 4: Indicates violation of Flow Control (FC) rules bit 5: Indicates a ECRC error (FC credits are updated) bit 6: Indicates reception of an unsupported TLP (FC credits are updated) bit 7: Indicates a TLP routed to the configuration space (FC credits are updated) <p>Note that if bits 1, 2, 3, or 4 are set, the TLP is removed from receive buffer and no FC credits are consumed. If bit 5, 6 or 7 is set, the TLP is routed to the configuration space after being written to the receive buffer and FC credits are updated.</p>
[11]	rx_vc_tlp	Receive TLP VC mapping: This signal reports the VC resource on which the TLP is mapped according to its TC.
[12]	rx_ok_tlp	Receive sequencing valid: This is a sequencing signal pulse. <code>test_out[37:0]</code> are valid only when this signal is asserted.
[15:13]	desc_sm	Receive descriptor state machine: Receive descriptor state machine encoding: <ul style="list-style-type: none"> 000: idle 001: desc0 010: desc1 011: desc2 100: desc_wt others: reserved
[16]	desc_val	Receive bypass mode valid: This signal reports that bypass mode is valid for the current received TLP.
[18:17]	data_sm	Receive data state machine: Receive data state machine encoding: <ul style="list-style-type: none"> 00: idle 01: data_first 10: data_next 11: data_last

Table B-3. Transaction Layer test_out Signals for the Hard IP Implementation when test_in[11:8]= 4'b0000 (Part 2 of 4)

Bits	Signal Name	Description
[19]	req_ro	Receive reordering queue busy: This signal reports that TLPs are currently reordered in the reordering queue (information extracted from the TLP fifo).
[20]	tlp_emp	Receive TLP fifo empty flag: This signal reports that the TLP fifo is empty.
[21]	tlp_val	Receive TLP pending in normal queue: This signal reports that a TLP has been extracted from the TLP fifo, but is still pending for transmission to the application layer.
[25:22]	r2c_ack c2r_ack rxbuf_busy rxfc_updated	Receive VC status: Reports different events related to the selected VC. <ul style="list-style-type: none"> ■ bit 0: TLP sent to the configuration space ■ bit 1: TLP received from configuration space ■ bit 2: Receive buffer not empty ■ bit 3: Receive FC credits updated
Transmit Path		
[26]	tx_req_tlp	Transmit request to DLL: This signal is a global VC request for transmitting TLP to the data link layer.
[27]	tx_ack_tlp	Transmit request acknowledge from DLL: This signal serves as the acknowledge signal for the global request from the Transaction Layer when accepting a TLP from the data link layer.
[28]	tx_dreq_tlp	Transmit data requested from DLL: This is a sequencing signal that makes a request for next data from the transaction layer.
[29]	tx_err_tlp	Transmit nullify TLP request: This signal is asserted by the transaction layer in order to nullify a transmitted TLP.
[31:30]	gnv_vc	Transmit VC arbitration result: This signal reports arbitration results of the TLP that is currently being transmitted.
[32]	tx_ok_tlp	Transmit sequencing valid: This signal, which depends on the number of initialized lanes on the link, is a sequencing signal pulse that enables data transfer from the transaction layer to the data link layer.
[36:33]	txbk_sm	Transmit state machine: Transmit state machine encoding: <ul style="list-style-type: none"> ■ 0000: idle ■ 0001: desc4dw ■ 0010: desc3dw_norm ■ 0011: desc3dw_shft ■ 0100: data_norm ■ 0101: data_shft ■ 0110: data_last ■ 0111: config0 ■ 1000: config1 ■ others: reserved

Table B-3. Transaction Layer test_out Signals for the Hard IP Implementation when test_in[11:8]= 4'b0000 (Part 3 of 4)

Bits	Signal Name	Description
Configuration Space		
[40:37]	lpm_sm	PM state machine: This signal indicates the power management state machine encoding responsible for scheduling the transition to legacy low power: <ul style="list-style-type: none"> 0000: l0_rst 0001: l0 0010: l1_in0 0011: l1_in1 0100: l0_in 0101: l0_in_wt 0110: l2l3_in0 0111: l2l3_in1 1000: l2l3_rdy 1001: aspm_l1_in0 1010: aspm_l1_in1 1011: aspm_l1_out 1100: l1 others: reserved
[41]	pme_sent	PME sent flag: This signal reports that a PM_PME message has been sent by the MegaCore function. Valid in endpoint mode only.
[42]	pme_resent	PME re-sent flag: This signal reports that the MegaCore function has requested to resend a PM_PME message that has timed out due to the latency timer. Valid in endpoint mode only.
[43]	inh_dllp	PM request stop DLLP/TLP transmission: This is a power management function that inhibits DLLP transmission in order to move to low power state.
[47:44]	req_phypm	PM directs LTSSM to low-power: This is a power management function that requests LTSSM to move to low-power state: <ul style="list-style-type: none"> bit 0: exit any low-power state to L0 bit 1: requests transition to L0s bit 2: requests transition to L1 bit 3: requests transition to L2
[49:48]	ack_phypm	LTSSM report PM transition event: This is a power management function that reports that LTSSM has moved to low-power state: <ul style="list-style-type: none"> bit 0: receiver detects low-power exit bit 1: indicates that the transition to low-power state is complete
[50]	pme_status3 & rx_pm_pme	Received PM_PME message discarded: This signal reports that a received PM_PME message has been discarded by the root port because of insufficient storage space.
[51]	link_up	Link Up: This signal reports that the link is up from the LTSSM perspective.
[52]	dl_up	DL Up: This signal reports that the data link is up from the DLCMSM perspective.
[53]	vc_en	VC enable: This signal reports which VCs are enabled by the software. (Note that VC0 is always enabled, thus the VC0 bit is not reported).
[55:54]	vc_status	VC status: This signal report which VC has successfully completed its initialization.

Table B-3. Transaction Layer test_out Signals for the Hard IP Implementation when test_in[11:8]= 4'b0000 (Part 4 of 4)

Bits	Signal Name	Description
[58:56]	req_phycfg	Configuration Space request the LTSSM to specific state: <ul style="list-style-type: none"> ■ bit 0: retrain link through recovery/configuration state. Valid for root port only. ■ bit 1: disable link through recovery/disable state. Valid for root port only. ■ bit 2: hot reset through recovery/hotreset state. Valid for root port only. ■ bit 3: Recover link through recovery state
[59]	retrain_link	From config space; asserted to request link to retrain.
[63:60]	—	Reserved.

Data Link Layer Outputs

Table B-4 describes the data link layer test outputs. These signals are driven on test_out when test_in[11:8] = 4'b0001.

Table B-4. Data Link Layer test_out for the hard IP Implementation when test_in[11:8]=4'b0001 (Part 1 of 4)

Bits	Signal Name	Description
[1:0]	dlcm_sm	DLCM state machine: DLCM state machine encoding: <ul style="list-style-type: none"> ■ 00: dl_inactive ■ 01: dl_init ■ 10: dl_active ■ 11: reserved
[4:2]	fcip_sm	Transmit InitFC state machine: Transmit Init FC state encoding: <ul style="list-style-type: none"> ■ 000: idle ■ 001: prep0 ■ 010: prep1 ■ 011: initfc_p ■ 100: initfc_np ■ 101: initfc_cpl ■ 110: initfc_wt ■ 111: reserved
[7:5]	flagfi1_cpl & flagfi1_np & flagfi1_p	Receive InitFC state machine: Receive Init FC state encoding: <ul style="list-style-type: none"> ■ 000: idle ■ 001: ifc1_p ■ 010: ifc1_np ■ 011: ifc1_cpl ■ 100: ifc2 ■ 111: reserved
[8]	flag_fi1	Flag_fi1: FI1 flag as detailed in the <i>PCI Express Base Specification Revision 1.1</i> and <i>PCI Express Base Specification Revision 1.0a</i>
[9]	flag_fi2	Flag_fi2: FI2 flag as detailed in the <i>PCI Express Base Specification Revision 1.1</i> and <i>PCI Express Base Specification Revision 1.0a</i>

Table B-4. Data Link Layer test_out for the hard IP Implementation when test_in[11:8]=4'b0001 (Part 2 of 4)

Bits	Signal Name	Description
[12:10]	rtry_sm	<ul style="list-style-type: none"> ■ Retry state machine: Retry State Machine encoding: ■ 000: idle ■ 001: rtry_ini ■ 010: rtry_wt0 ■ 011: rtry_wt1 ■ 100: rtry_req ■ 101: rtry_tlp ■ 110: rtry_end ■ 111: reserved
[14:13]	storebuf_sm	Retry buffer storage state machine: Retry Buffer Storage State Machine encoding: <ul style="list-style-type: none"> ■ 00: idle ■ 01: rtry ■ 10: str_tlp ■ 11: reserved
[15]	mem_replay	Retry Buffer running: This signal keeps track of TLPs that have been sent but not yet acknowledged. Note that the replay timer is also running when this bit is set except if a replay is currently being performed.
[16]	mem_rtry	Memorize replay request: This signal indicates that a replay timeout event has occurred or that a NAK DLLP has been received.
[18:17]	replay_num	Replay number counter: This signal counts the number of replays performed by the MegaCore function for a particular TLP as described in the <i>PCI Express Base Specification Revision 1.1</i> and <i>PCI Express Base Specification Revision 1.0a</i> .
[19]	val_nak_r	ACK/NAK DLLP received: This signal reports that an ACK or a NAK DLLP has been received. The res_nak_r, tlp_ack, err_dl and no_rtry signals detail the type of ACK/NAK DLLP received.
[20]	res_nak_r	NAK DLLP parameter: This signal reports that the received ACK/NAK DLLP is NAK.
[21]	tlp_ack	Real ACK DLLP parameter: This signal reports that the received ACK DLLP acknowledges one or several TLPs in the Retry Buffer.
[22]	err_dl	Error ACK/NAK DLLP parameter: This signal reports that the received ACK/NAK DLLP has a sequence number higher than the sequence number of the last transmitted TLP.
[23]	no_rtry	No retry on ACK/NAK DLLP received: This signal reports that the received ACK/NAK DLLP sequence number corresponds to the last acknowledged TLP.
[26:24]	txdl_sm	Transmit TLP State Machine: Transmit TLP state machine encoding: <ul style="list-style-type: none"> ■ 000: idle ■ 001: tlp1 ■ 010: tlp2 ■ 011: tlp3a ■ 100: tlp5a (ECRC only) ■ 101: tlp6a (ECRC only) ■ 111: reserved This signal can be used to inject an LCRC or ECRC error.

Table B-4. Data Link Layer test_out for the hard IP Implementation when test_in[11:8]=4'b0001 (Part 3 of 4)

Bits	Signal Name	Description
[27]	tx3b or tx4 or tx5b	TLP transmitted: This signal is set on the last DW of the packet where the LCRC is added to the packet. This signal can be used to inject an LCRC or ECRC error.
[28]	tx0	DLLP transmitted: This signal is set when a DLLP is sent to the physical layer. This signal can be used to inject a CRC on a DLLP.
[36:29]	gnt	DLL transmit arbitration result: This signal reports the arbitration result between a DLLP and a TLP: <ul style="list-style-type: none"> ■ bit 0: InitFC DLLP ■ bit 1: ACK DLLP (high priority) ■ bit 2: UFC DLLP (high priority) ■ bit 3: PM DLLP ■ bit 4: TXN TLP ■ bit 5: RPL TLP ■ bit 6: UFC DLLP (low priority) ■ bit 7: ACK DLLP (low priority)
[37]	sop	DLL to PHY start of packet: This signal reports that an SDP/STP symbol is in transition to the physical layer.
[38]	eop	DLL to PHY end of packet: This signal reports that an EDB/END symbol is in transition to the Physical Layer. Note that when sop and eop are transmitted together, it indicates that the packet is a DLLP. Otherwise the packet is a TLP.
[39]	eot	DLL to PHY end of transmit: This signal reports that the DLL has finished its previous transmission and enables the physical layer to go to low-power state or to recovery.
[40]	init_lat_timer	Enable ACK latency timer: This signal reports that the ACK latency timer is running.
[41]	req_lat	ACK latency timeout: This signal reports that an ACK/NAK DLLP retransmission has been scheduled due to the ACK latency timer expiring.
[42]	tx_req_nak or tx_snd_nak	ACK/NAK DLLP requested for transmission: This signal reports that an ACK/NAK DLLP is currently requested for transmission.
[43]	tx_res_nak	ACK/NAK DLLP type requested for transmission: This signal reports that type of ACK/NAK DLLP scheduled for transmission: <ul style="list-style-type: none"> ■ 0: ACK ■ 1: NAK
[44]	rx_val_pm	Received PM DLLP: This signal reports that a PM DLLP has been received. The specific type is indicated by rx_vcid_fc: <ul style="list-style-type: none"> ■ 000: PM_Enter_L1 ■ 001: PM_Enter_L23 ■ 011: PM_AS_Request_L1 ■ 100: PM_Request_ACK
[45]	rx_val_fc	Received FC DLLP: This signal reports that a PM DLLP has been received. The type of FC DLLP is indicated by rx_typ_fc and rx_vcid_fc.

Table B-4. Data Link Layer test_out for the hard IP Implementation when test_in[11:8]=4'b0001 (Part 4 of 4)

Bits	Signal Name	Description
[49:46]	rx_typ_fc	Received FC DLLP Type parameter: This signal reports the type of received FC DLLP: <ul style="list-style-type: none"> ■ 0100: InitFC1_P ■ 0101: InitFC1_NP ■ 0110: InitFC1_CPL ■ 1100: InitFC2_P ■ 1101: InitFC2_NP ■ 1110: InitFC2_CPL ■ 1000: UpdateFC_P ■ 1001: UpdateFC_NP ■ 1010: UpdateFC_CPL
[52:50]	rx_vcid_fc	Received FC DLLP VC ID parameter: This signal reports the VC ID of the received FC DLLP: <ul style="list-style-type: none"> ■ 000: VCID 0 ■ 001: VCID 1 ■ ... ■ 111: VCID 7 <p>Note that this signal also indicates the type of PM DLLP received.</p>
[53]	crcinv	Received nullified TLP: This signal indicates that a nullified TLP has been received.
[54]	crcerr	Received TLP with LCRC error: This signal reports that a TLP has been received that contains an LCRC error.
[55]	crcval & eqseq_r	Received valid TLP: This signal reports that a valid TLP has been received that contains the correct sequence number. Such a TLP is transmitted to the application layer.
[56]	crcval & !eqseq_r & infseq_r	Received duplicated TLP: This signal indicates that a TLP has been received that has already been correctly received. Such a TLP is silently discarded.
[57]	crcval & !eqseq_r & !infseq_r	Received erroneous TLP: This signal indicates that a TLP has been received that contains a valid LCRC but a non-sequential sequence number, higher than the current sequence number.
[58]	rx_err_frame	DLL framing error detected: This signal indicates that received data cannot be considered as a DLLP or TLP, in which case a receive port error is generated and link retraining is initiated.
[63:59]	Reserved	—

PHY-MAC Outputs

Table B-5 describes the PHY-MAC output signals. These signals are driven on test_out when test_in[11:8]=b'0010. test_in[15:13] selects the lane.

Table B-5. PHY/MAC test_out hard IP Implementation when test_in[11:8]=b'0010 (Part 1 of 4)

Bits	Signal Name	Description
[4:0]	ltssm_r	LTSSM state: LTSSM state encoding: <ul style="list-style-type: none"> ■ 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: I/Os ■ 10110: L1.entry ■ 10111: L1.idle ■ 11000: L2.idle ■ 11001: L2.transmit.wake
[6:5]	rxl0s_sm	Receive L0s state: Receive L0s state machine <ul style="list-style-type: none"> ■ 00: inact ■ 01: idle ■ 10: fts ■ 11: out.recovery

Table B-5. PHY/MAC test_out hard IP Implementation when test_in[11:8]=b'0010 (Part 2 of 4)

Bits	Signal Name	Description
[9:7]	txl0s_sm	TX L0s state: Transmit L0s state machine <ul style="list-style-type: none"> 000: inact 001: entry 010: idle 011: fts 100: out.I0
[10]	timeout	LTSSM Timeout: This signal serves as a flag that indicates that the LTSSM timeout condition has been reached for the current LTSSM state.
[11]	txos_end	Transmit LTSSM exit condition: This signal serves as a flag that indicates that the LTSSM exit condition for the next state (in order to go to L0) has been completed. If the next state is not reached in a timely manner, it is due to a problem on the receiver.
[12]	tx_ack	Transmit PLP acknowledge: This signal is active for 1 clock cycle when the requested physical layer packet (PLP) has been sent to the link. The type of packet is defined by the tx_ctrl signal. Note that in the 250 MHz version of the MegaCore function, TRANSMIT PLP ACKNOWLEDGE: TX_ACK is asserted 1 clock cycle every 4 clock cycles for D0.0 idle data, FTS, SKP, and IDL OS, and 1 clock cycle every 16 clock cycles for TS1 TS2 OS. For receiver detect, the number of clock cycle is PMA dependent.
[15:13]	tx_ctrl	Transmit PLP type: This signal indicates the type of transmitted PLP: <ul style="list-style-type: none"> 000: Electrical Idle 001: Receiver detect during Electrical Idle 010: TS1 OS 011: TS2 OS 100: D0.0 idle data 101: FTS OS 110: IDL OS 111: Compliance pattern
[16]	txrx_det (selected lane)	Receiver detect result: This signal serves as a per-lane flag that reports the receiver detection result.
[17]	tx_pa (selected lane)	Force PAD on transmitted TS pattern: This is a per-lane internal signal that forces PAD transmission on the link and lane field of the transmitted TS1/TS2 OS. The MegaCore function considers that lanes indicated by this signal should not be initialized during the initialization process.
[18]	rx_ts1 (selected lane)	Received TS1: This signal indicates that a TS1 has been received on the specified lane. This signal is cleared when a new state is reached by the LTSSM state machine.
[19]	rx_ts2 (selected lane)	Received TS2: This signal indicates that a TS1 has been received on the specified lane. This signal is cleared when a new state is reached by the LTSSM state machine.

Table B-5. PHY/MAC test_out hard IP Implementation when test_in[11:8]=b'0010 (Part 3 of 4)

Bits	Signal Name	Description
[20]	rx_8d00 (selected lane)	Received 8 D0.0 symbol: This signal indicates that 8 consecutive Idle data symbols have been received. This signal is meaningful for config.idle and recovery.idle states.
[21]	rx_idl (selected lane)	Received IDL OS: This signal indicates that an IDL OS has been received on a per lane basis.
[22]	rx_linkpad (selected lane)	Received Link Pad TS: This signal indicates that the link field of the received TS1/TS2 is set to PAD for the specified lane.
[23]	rx_lanepad (selected lane)	Received Lane Pad TS: This signal indicates that the lane field of the received TS1/TS2 is set to PAD for the specified lane.
[24]	rx_tsnnum (selected lane)	Received Consecutive Identical TSNumber: This signal reports the number of consecutive identical TS1/TS2 which have been received with exactly the same parameters since entering this state. When the maximum number is reached, this signal restarts from zero. Note that this signal corresponds to the lane configured as logical lane 0 (may vary depending on lane reversal).
[28:25]	lane_act	Lane Active Mode: This signal indicates the number of lanes that have been configured during training: <ul style="list-style-type: none"> 0001: 1 lane 0010: 2 lanes 0100: 4 lanes 1000: 8 lanes
[32:29]	lane_rev	Lane Reversal Mode: This signal indicates whether initialization of the link includes lane reversal: <ul style="list-style-type: none"> 0001: no lane reversal 0010: Lane reversal starting on lane 1 0100: Lane reversal starting on lane 3 1000: Lane reversal starting on lane 7 Note that if lane_act is set to 4'b0010 and this signal is set to 4'b0100, only two lanes are initialized (physical lanes 3 and 2 for logical lanes 0 and 1).
[35:33]	countx (selected lane)	Deskew FIFO count lane x: This signal indicates the number of words in the deskew FIFO for physical lane selected.
[36]	err_deskew (selected lane)	Deskew FIFO error: This signal indicates whether a deskew error (deskew FIFO overflow) has been detected on a particular physical lane. In such a case, the error is considered a receive port error and retraining of the link is initiated.
[37]	directed_speed_change	Directed Speed Change: For LTSSM recovery states, this signal indicates that consecutive TS with the speed_change bit (bit 7 of TSOS 4th symbol) set to 1 have been received.
[38]	successful_speed_negotiation	Successful Speed Negotiation: This signal indicates that speed negotiation before entering the recovery.speed state was successful (the speed change will be implemented).
[39]	changed_speed_recovery	Changed Speed Recovery: This signal indicates that the operating speed has been changed to a mutually negotiated data rate.

Table B-5. PHY/MAC test_out hard IP Implementation when test_in[11:8]=b'0010 (Part 4 of 4)

Bits	Signal Name	Description
[40]	select_deemphasis	Select De-emphasis: If the new operating speed is 5.0 GT/s data rate: <ul style="list-style-type: none"> 6 dB de-emphasis level is selected if select_deemphasis = 0. 3.5 dB de-emphasis level is selected if select_deemphasis = 1.
[41]	idle_to_rlock_transitioned	Idle to Recovery Lock Transition: This signal is asserted when transitioning to Recovery.RcvrLock from Configuration Idle.
[42]	tx_speed_change	Transmit Speed Change: This signal is used to generate the speed change bit (bit 7 of the TSOS 4th symbol). It is asserted when a change to operation speed is requested. The speed change bit can only be set to 1 during Recovery.RcvrLock state.
[63:43]	—	Reserved

PHY PCS layer

describes the PHY PCS test outputs. These signals are driven on test_out when test_in[11:8] = 4'b0011.

^N
Table B-6. PHY PCS test_out Signals (Note 1), (Note 2)

Bits	Signal Name	Description
[7:0]	txdata	Transmit data.
[8]	txdatak	Transmit data control.
[9]	txdetectrx	Transmit detect receive. This signal is used to tell the PHY layer to start a receive detection operation or to begin loopback
[10]	txelecidle	Transmit electrical idle 0. This signal forces the transmit output to electrical idle.
[11]	txcompl	Transmit compliance 0. This signal forces the running disparity to negative in compliance mode (negative COM character).
[12]	rxpolarity	Receive polarity 0. This signal instructs the PHY layer to do a polarity inversion on the 8B10B receiver decoding block.
[14:13]	powerdown	Power down 0. This signal requests the PHY to change its power state to the specified state (P0, P0s, P1, or P2).
[22:15]	rxdata	Receive data.
[23]	rxdatak	Receive data control.
[24]	rxvalid	Receive valid.

Note to Table B-6:

- (1) All signals are per lane.
- (2) Refer to "PIPE Interface Signals" on page 5-68 for a detailed definition of these signals.

CDC Test Outputs

Table B-7 describes the clock domain crossing test outputs. These signals are driven on test_out when test_in[11:8] = 4'b0100.

Table B-7. CDC test_out Signals

Bits	Signal Name	Description
[0]	dll_wrfull	FIFO full from Tx direction
[1]	mac_tx_empty	FIFO empty from Tx direction
[5:2]	dll_wrusedw	Number of data words written in Tx direction
[6]	mac_wrfull	FIFO full from RX direction
[7]	—	Reserved
[8]	dll_rx_empty	FIFO empty on Rx direction
[12:9]	mac_wrusedw	Number of data words written in RX direction
[63:13]	—	Reserved

Mixed Mode Test Outputs

Table B-7 describes the mixed mode test outputs. These signals are driven on test_out when test_in[11:8] = 4'b0101. test_in[15:13] selects the lane to be monitored.

|| **Table B-8.** Mixed Mode test_out Signals (Part 1 of 5)

Bit	Signal Name	Description
[2:0]	Desc_sm	Receive descriptor state machine: Receive descriptor state machine encoding: <ul style="list-style-type: none"> ■ 000: idle ■ 001: desc0 ■ 010: desc1 ■ 011: desc2 ■ 100: desc_wt ■ others: reserved
[4:3]	data_sm	Receive data state machine: Receive data state machine encoding: <ul style="list-style-type: none"> ■ 00: idle ■ 01: data_first ■ 10: data_next ■ 11: data_last
[8:5]	txbk_sm	Transmit state machine: Transmit state machine encoding: <ul style="list-style-type: none"> ■ 0000: idle ■ 0001: desc4dw ■ 0010: desc3dw_norm ■ 0011: desc3dw_shft ■ 0100: data_norm ■ 0101: data_shft ■ 0110: data_last ■ 0111: config0 ■ 1000: config1 ■ others: reserved

Table B-8. Mixed Mode test_out Signals (Part 2 of 5)

Bit	Signal Name	Description
[12:9]	lpm_sm	PM state machine: This signal indicates the Power Management state machine encoding responsible for scheduling the transition to legacy low power: <ul style="list-style-type: none"> 0000: l0_rst 0001: l0 0010: l1_in0 0011: l1_in1 0100: l0_in 0101: l0_in_wt 0110: l2l3_in0 0111: l2l3_in1 1000: l2l3_rdy 1001: aspm_l1_in0 1010: aspm_l1_in1 1011: aspm_l1_out 1100: l1 others: reserved
[13]	link_up	Link up: This signal reports that the link is up from the LTSSM perspective.
[14]	dl_up	DL up: This signal reports that the data link is up from the DLCMSM perspective.
DLL		
[16:15]	dlcm_sm	DLCM state machine: DLCM state machine encoding: <ul style="list-style-type: none"> 00: dl_inactive 01: dl_init 10: dl_active 11: reserved
[19:17]	fcip_sm	Transmit InitFC state machine: Transmit init FC state encoding: <ul style="list-style-type: none"> 000: idle 001: prep0 010: prep1 011: initfc_p 100: initfc_np 101: initfc_cpl 110: initfc_wt 111: reserved

Table B-8. Mixed Mode test_out Signals (Part 3 of 5)

Bit	Signal Name	Description
[22:20]	flagfi1_cpl & flagfi1_np & flagfi1_p	Receive InitFC state machine: Receive Init FC state encoding: <ul style="list-style-type: none"> 000: idle 001: ifc1_p 010: ifc1_np 011: ifc1_cpl 100: ifc2 111: reserved
[25:23]	rtry_sm	Retry state machine: Retry state machine encoding: <ul style="list-style-type: none"> 000: idle 001: rtry_ini 010: rtry_wt0 011: rtry_wt1 100: rtry_req 101: rtry_tlp 110: rtry_end 111: reserved
[27:26]	storebuf_sm	Retry buffer storage state machine: Retry Buffer Storage state machine encoding: <ul style="list-style-type: none"> 00: idle 01: rtry 10: str_tlp 11: reserved
[30:28]	txdl_sm	Transmit TLP state machine: Transmit TLP state machine encoding: <ul style="list-style-type: none"> 000: idle 001: tlp1 010: tlp2 011: tlp3a 100: tlp5a (ECRC only) 101: tlp6a (ECRC only) 111: reserved <p>This signal can be used to inject an LCRC or ECRC error.</p>

Table B-8. Mixed Mode test_out Signals (Part 4 of 5)

Bit	Signal Name	Description
MAC		
[35:31]	ltssm_r	<p>LTSSM state: LTSSM state encoding:</p> <ul style="list-style-type: none"> ■ 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: I/Os ■ 10110: L1.entry ■ 10111: L1.idle ■ 11000: L2.idle ■ 11001: L2.transmit.wake
[37:36]	rxl0s_sm	<p>Receive L0s state: Receive L0s state machine</p> <ul style="list-style-type: none"> ■ 00: inact ■ 01: idle ■ 10: fts ■ 11: out.recovery
[40:38]	txl0s_sm	<p>TX L0s state: Transmit L0s state machine</p> <ul style="list-style-type: none"> ■ 000: inact ■ 001: entry ■ 010: idle ■ 011: fts ■ 100: out.IO

Table B-8. Mixed Mode test_out Signals (Part 5 of 5)

Bit	Signal Name	Description
[48:41]	rxdata (selected lane)	Receive Data: This bus receives data on lane selected.
[49]	rxdata _{ak} (selected lane)	Receive Data Control: This signal is used for separating control and data symbols.
[57:50]	txdata (selected lane)	Transmit Data: This bus transmits data on lane selected.
[58]	txdata _{ak} (selected lane)	Transmit Data Control: This signal serves as the control bit for txdata symbol 8b/10b encoding.
[63:59]	—	Reserved

Receive FC Test Outputs

Table B-9 describes the receive FC test outputs. These signals are driven on test_out when test_in[11:8] = 4'b0110.

Table B-9. Receive FC test_out Signals

Bit	Signal Name	Description
[55:0]	rx_sub	Receive FC credits. The bit definitions for Receive FC Test Outputs are as follows: <ul style="list-style-type: none"> ■ bit [7:0]: Posted header (PH) ■ bit [19:8]: Posted data (PD) ■ bit [27:20]: Non-posted header (NPH) ■ bit [35:28]: Non-posted data (NPD) ■ bit [43:36]: Completion header (CPLH) ■ bit [55:44]: Completion data (CPLD)
[63:56]	—	Reserved

Transmit FC Test Outputs

Table B-10 describes the transmit FC test outputs. These signals are driven on test_out when test_in[11:8] = 4'b0111.

Table B-10. Transmit FC test_out Signals

Bit	Signal Name	Description
[55:0]	tx_sub	<p>Transmit FC credits. The bit definitions for transmit FC test outputs are as follows:</p> <ul style="list-style-type: none"> ■ bit [7:0]: Posted header (PH) ■ bit [19:8]: Posted data (PD) ■ bit [27:20]: Non-posted header (NPH) ■ bit [35:28]: Non-posted data (NPD) ■ bit [43:36]: Completion header (CPLH) ■ bit [55:44]: Completion data (CPLD) <p>The test_out bus reflects the real number of available the credits for the completion header, posted header, non-posted header, and non-posted data fields. The tx_cred port described in Table 5-4 assigns a value of 7 to mean 7 or more available credits.</p>
[63:56]	—	Reserved

Request ID and Error Signal Test Outputs

[Table B-11](#) describes the request ID and error signal test outputs. These signals are selected when test_in[11:8] = 4'b1000.

Table B-11. Request ID and Error test_out Signals

Bit	Signal Name	Description
[44:21]	rx_reqid_tlp	<p>Bit definitions for Request ID and Error Signal Test Outputs are as follows:</p> <p>Receive ReqID. This 24-bit signal reports the requester ID of the completion TLP when rx_hval_tlp and rx_ok_tlp are asserted. The 8 MSBs of this signal also report the type and format of the transaction when rx_fval_tlp and rx_ok_tlp are valid.</p> <p>TRN error. Transaction Layer error. The bits are defined as follows:</p> <ul style="list-style-type: none"> ■ bit 0: poisoned TLP received ■ bit 1: ECRC check failed ■ bit 2: Unsupported request ■ bit 3: Completion timeout ■ bit 4: Completer abort ■ bit 5: Unexpected Completion ■ bit 6: Receiver overflow ■ bit 7: Flow control protocol error ■ bit 8: Malformed TLP <p>DLL error. Data link layer error. The bits are defined as follows:</p> <ul style="list-style-type: none"> ■ bit 0: TLP error ■ bit 1: DLLP error ■ bit 2: Replay timer error ■ bit 3: Replay counter rollover ■ bit 4: DLL protocol error
[20:12]	err_trn	
[11:7]	err_dll	

Table B-11. Request ID and Error test_out Signals

Bit	Signal Name	Description
[6:5]	err_phy	PHY error. Physical Layer error. The bits are defined as follows: <ul style="list-style-type: none"> bit0: Receiver port error bit1: Training error
[4]	err_dllprot	DLL protocol error at Data Link Layer. This signal reports a DLL protocol error.
[3]	err_repnum	Replay counter rollover error at Data Link Layer. This signal indicates that a replay counter has rolled over.
[2]	err_reptim	Replay timer error at Data Link Layer. This signal indicates that a replay timer has timed out.
[1]	rx_err_dllp	Received DLLP with LCRC error. This signal reports that a DLLP has been received that contains an LCRC error.
[0]	rx_err_tlp	Received erroneous TLP conditioned by TLP valid. This signal indicates that an erroneous TLP gated with a TLP valid signal has been received.
[63:45]	—	Reserved

Receive DLL Data Test Outputs

Table B-12 describes the receive data DLL test outputs. These signals are driven on test_out when test_in[11:8] = 4'b1001.

Table B-12. Receive DLL Data test_out Signals

Bit	Signal Name	Description
[63:0]	rxdata_dl	DLL receive data. Receives data from the data link layer. Byte 7 is the first receive byte.

Transmit DLL Data Test Outputs

Table B-13 describes the transmit DLL data test outputs. These signals are driven on test_out when test_in[11:8] = 4'b1010.

Table B-13. Transmit DLL Data test_out Signals

Bit	Signal Name	Description
[63:0]	txdata	DLL transmit data. Transmits data from the data link layer. Byte 7 is the first byte transmitted.

Miscellaneous Test Outputs

Table B-14 describes the miscellaneous test outputs. These signals are driven on test_out when test_in[11:8] = 4'b1011.

Table B-14. Miscellaneous test_out Signals

Bits	Signal Name	Description
[20]	dll_req	DLL request: Reports that the data link layer transmit path has a pending request.
[19:12]	rxdatak_dl	DLL receive data control bits. Receives data control bits to Data Link Layer. Bit 7 is the first received symbol corresponding to Byte 7 (bit [63:56]).
[11]	rxval_dl	DLL receive data valid. Receive valid signal indicates that the corresponding 8 symbols are valid.
[10:3]	txdata	Transmit data control bits. Transmit data control bits from the data link layer. Bit 7 is the first symbol to transmit corresponding to Byte 7, bits [63:56].
[2]	txok	Transmit OK. Enable signal which allows the data link layer to send the next 8 symbols to transmit. This signal is removed when the physical layer inserts a SKIP ordered set or when the link is initialized with fewer lanes than the maximum number possible.
[1]	mst_lpbk	Master loopback. This signal indicates that the MegaCore function is in loopback in master mode.
[0]	cdc_end_operation_timer	CDC End of operation timer. This signal indicates the end of an operation on the CDC.
[63:21]	—	Reserved

Soft IP MegaCore Function

The following sections describe the test_out and test_in busses soft IP implementation of the MegaCore function for the ×1 and ×4 parameterizations.



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 signals have not been rigorously verified and will not function as documented in some corner cases.

Soft IP Implementation x1 and x4 test_out

Table B-15 describes the test_out signals for the x1 and x4 soft IP Avalon-MM, and descriptor/data MegaCore functions.

Table B-15. test_out Signals for the x1 and x4 MegaCore Functions (Part 1 of 14)

Signal	Subblock	Bits	Description
rx_fval_tlp rx_hval_tlp rx_dval_tlp	TRN rxtl	[2:0]	Receive transaction layer packet reception state. These signals report the transaction layer packet reception sequencing. bit 0: DWORD0 and DWORD1 of the header are valid bit 1: DWORD2 and DWORD3 of the header are valid bit 2: The data payload is valid
rx_check_tlp rx_discard_tlp rx_mlf_tlp tlp_err rxfc_ovf rx_ecrcerr_tlp rx_uns_tlp rx_sup_tlp	TRN rxtl	[10:3]	Receive transaction layer packet check state. These signals report the transaction layer packet reception sequencing: <ul style="list-style-type: none"> bit 0: Check LCRC bit 1: Indicates an LCRC error or sequence number error bit 2: Indicates a malformed transaction layer packet due to a mismatch END/length field bit 3: Indicates a malformed transaction layer packet that doesn't conform with formation rules bit 4: Indicates violation of flow control rules bit 5: Indicates a ECRC error (flow control credits are updated) bit 6: Indicates reception of an unsupported transaction layer packet (flow control credits are updated) bit 7: Indicates a transaction layer packet routed to the Configuration space (flow control credits are updated) If bits 1, 2, 3, or 4 are set, the transaction layer packet is removed from the receive buffer and no flow control credits are consumed. If bit 5, 6 or 7 is set, the transaction layer packet is routed to the configuration space after being written to the receive buffer and flow control credits are updated.
rx_vc_tlp	TRN rxtl	[13:1]	Receive transaction layer packet virtual channel mapping. This signal reports the virtual channel resource on which the transaction layer packet is mapped (according to its traffic class).
rx_reqid_tlp	TRN rxtl	[37:14]	Receive ReqID. This 24-bit signal reports the requester ID of the completion transaction layer packet when rx_hval_tlp and rx_ok_tlp are asserted. The 8 MSBs of this signal also report the type and format of the transaction when rx_fval_tlp and rx_ok_tlp are valid.
rx_ok_tlp	TRN rxtl	[38]	Receive sequencing valid. This is a sequencing signal pulse. All previously-described signals (test_out [37:0]) are valid only when this signal is asserted.
tx_req_tlp	TRN txtl	[39]	Transmit request to data link layer. This signal is a global virtual channel request for transmitting transaction layer packet to the data link layer.
tx_ack_tlp	TRN txtl	[40]	Transmit request acknowledge from data link layer. This signal serves as the acknowledge signal for the global request from the transaction layer when accepting a transaction layer packet from the data link layer.

Table B-15. *test_out Signals for the ×1 and ×4 MegaCore Functions (Part 2 of 14)*

Signal	Subblock	Bits	Description
tx_dreq_tlp	TRN txtl	[41]	Transmit data requested from data link layer. This is a sequencing signal that makes a request for next data from the transaction layer.
tx_err_tlp	TRN txtl	[42]	Transmit nullify transaction layer packet request. This signal is asserted by the transaction layer in order to nullify a transmitted transaction layer packet.
gnt_vc	TRN txtl	[50:43]	Transmit virtual channel arbitration result. This signal reports arbitration results of the transaction layer packet that is currently being transmitted.
tx_ok_tlp	TRN txtl	[51]	Transmit sequencing valid. This signal, which depends on the number of initialized lanes on the link, is a sequencing signal pulse that enables data transfer from the transaction layer to the data link layer.
lpm_sm	CFG pmgt	[55:52]	Power management state machine. This signal indicates the power management state machine encoding responsible for scheduling the transition to legacy low power: <ul style="list-style-type: none"> 0000: l0_rst 0001: l0 0010: l1_in0 0011: l1_in1 0100: l0_in 0101: l0_in_wt 0110: l2l3_in0 0111: l2l3_in1 1000: l2l3_rdy others: reserved
pme_sent	CFG pmgt	[56]	PME sent flag. This signal reports that a PM_PME message has been sent by the MegaCore function (endpoint mode only).
pme_resent	CFG pmgt	[57]	PME resent flag. The DWORD is signal reports that the MegaCore function has requested to resend a PM_PME message that has timed out due to the latency timer (endpoint mode only).
inh_dllp	CFG pmgt	[58]	PM request stop DLLP/transaction layer packet transmission. This is a power management function that inhibits DLLP transmission in order to move to low power state.
req_phympm	CFG pmgt	[62:59]	PM directs LTSSM to low-power. This is a power management function that requests LTSSM to move to low-power state: <ul style="list-style-type: none"> bit 0: exit any low-power state to L0 bit 1: requests transition to L0s bit 2: requests transition to L1 bit 3: requests transition to L2
ack_phympm	CFG pmgt	[64:63]	LTSSM report PM transition event. This is a power management function that reports that LTSSM has moved to low-power state: <ul style="list-style-type: none"> bit 0: receiver detects low-power exit bit 1: indicates that the transition to low-power state is complete
pme_status3 rx_pm_pme	CFG pcie	[65]	Received PM_PME message discarded. This signal reports that a received PM_PME message has been discarded by the root port because of insufficient storage space.

Table B-15. *test_out Signals for the ×1 and ×4 MegaCore Functions (Part 3 of 14)*

Signal	Subblock	Bits	Description
link_up	CFG pcie	[66]	Link up. This signal reports that the link is up from the LTSSM perspective.
dl_up	CFG pcie	[67]	DL Up. This signal reports that the data link is up from the data link control and management state machine (DLCMSM) perspective.
vc_en	CFG vcreg	[74:68]	Virtual channel enable. This signal reports which virtual channels are enabled by the software (note that VC0 is always enabled, thus the VC0 bit is not reported).
vc_status	CFG vcreg	[82:75]	Virtual channel status. This signal reports which virtual channel has successfully completed its initialization.
err_phy	CFG errmgt	[84:83]	PHY error. Physical layer error: bit 0: Receiver port error bit 1: reserved
err_dll	CFG errmgt	[89:85]	Data link layer error. Data link layer error: bit 0: Transaction layer packet error bit 1: Data link layerP error bit 2: Replay timer error bit 3: Replay counter rollover bit 4: Data link layer protocol error
err_trn	CFG errmgt	[98:90]	TRN error. Transaction layer error: <ul style="list-style-type: none"> ■ bit 0: Poisoned transaction layer packet received ■ bit 1: ECRC check failed ■ bit 2: Unsupported request ■ bit 3: Completion timeout ■ bit 4: Completer abort ■ bit 5: Unexpected completion ■ bit 6: Receiver overflow ■ bit 7: Flow control protocol error ■ bit 8: Malformed transaction layer packet
r2c_ack c2r_ack rxbuf_busy rxfc_updated	TRN rxvc0	[102:99]	Receive VC0 status. Reports different events related to VC0. <ul style="list-style-type: none"> ■ bit 0: Transaction layer packet sent to the configuration space ■ bit 1: Transaction layer packet received from configuration space ■ bit 2: Receive buffer not empty ■ bit 3: Receive flow control credits updated
r2c_ack c2r_ack rxbuf_busy rxfc_updated	TRN rxvc1	[106:103]	Receive VC1 status. Reports different events related to VC1: <ul style="list-style-type: none"> ■ bit 0: Transaction layer packet sent to the configuration space ■ bit 1: Transaction layer packet received from configuration space ■ bit 2: Receive buffer not empty ■ bit 3: Receive flow control credits updated

Table B-15. *test_out Signals for the ×1 and ×4 MegaCore Functions (Part 4 of 14)*

Signal	Subblock	Bits	Description
r2c_ack c2r_ack rxbuf_busy rxfc_updated	TRN rxvc2	[110:107]	Receive VC2 status. Reports different events related to VC2: <ul style="list-style-type: none"> ■ bit 0: Transaction layer packet sent to the configuration space ■ bit 1: Transaction layer packet received from configuration space ■ bit 2: Receive buffer not empty ■ bit 3: Receive flow control credits updated
r2c_ack c2r_ack rxbuf_busy rxfc_updated	TRN rxvc3	[114:111]	Receive VC3 status. Reports different events related to VC3: <ul style="list-style-type: none"> bit 0: Transaction layer packet sent to the configuration space bit 1: Transaction layer packet received from configuration space bit 2: Receive buffer not empty bit 3: Receive flow control credits updated
Reserved			All consecutive signals between bits 131 and 255 depend on the virtual channel selected by the <code>test_in[31:29]</code> input.
desc_sm	TRN rxvc	[133:131]	Receive descriptor state machine. Receive descriptor state machine encoding: <ul style="list-style-type: none"> ■ 000: idle ■ 001: desc0 ■ 010: desc1 ■ 011: desc2 ■ 100: desc_wt ■ others: reserved
desc_val	TRN rxvc	[134]	Receive bypass mode valid. This signal reports that bypass mode is valid for the current received transaction layer packet.
data_sm	TRN rxvc	[136:135]	Receive data state machine. Receive data state machine encoding: <ul style="list-style-type: none"> ■ 00: idle ■ 01: data_first ■ 10: data_next ■ 11: data_last
req_ro	TRN rxvc	[137]	Receive reordering queue busy. This signal reports that transaction layer packets are currently reordered in the reordering queue (information extracted from the transaction layer packet FIFO).
tlp_emp	TRN rxvc	[138]	Receive transaction layer packet FIFO empty flag. This signal reports that the transaction layer packet FIFO is empty.
tlp_val	TRN rxvc	[139]	Receive transaction layer packet pending in normal queue. This signal reports that a transaction layer packet has been extracted from the transaction layer packet FIFO, but is still pending for transmission to the application layer.

Table B-15. *test_out Signals for the ×1 and ×4 MegaCore Functions (Part 5 of 14)*

Signal	Subblock	Bits	Description
txbk_sm	TRN txvc	[143:140]	Transmit state machine. Transmit state machine encoding: <ul style="list-style-type: none"> 0000: idle 0001: desc4dw 0010: desc3dw_norm 0011: desc3dw_shft 0100: data_norm 0101: data_shft 0110: data_last 0111: config0 1000: config1 others: reserved
rx_sub	TRN rxfc	[199:144]	Receive flow control credits. Receive buffer current credits available: <ul style="list-style-type: none"> bit [7:0]: Posted header (PH) bit [19:8]: Posted data (PD) bit [27:20]: Non-posted header (NPH) bit [35:28]: Non-posted data (NPD) bit [43:36]: Completion header (CPLH) bit [55:44]: Completion data (CPLD) Flow control credits for NPD is limited to 8 bits because more NPD credits than NPH credits is meaningless.
tx_sub	TRN txfc	[255:200]	Transmit flow control credits. Transmit buffer current credits available: <ul style="list-style-type: none"> bit [7:0]: Posted header (PH) bit [19:8]: Posted data (PD) bit [27:20]: Non-posted header (NPH) bit [35:28]: Non-posted data (NPD) bit [43:36]: Completion header (CPLH) bit [55:44]: Completion data (CPLD) The test_out bus reflects the real number of available the credits for the completion header, posted header, non-posted header, and non-posted data fields. The tx_cred port described in Table 5-4 assigns a value of 7 to mean 7 or more available credits. Flow control credits for NPD is limited to 8 bits because more NPD credits than NPH credits is meaningless.
dlcm_sm	DLL dlcm-sm	[257:256]	DLCM state machine. DLCM state machine encoding: <ul style="list-style-type: none"> 00: dl_inactive 01: dl_init 10: dl_active 11: reserved

Table B-15. *test_out Signals for the ×1 and ×4 MegaCore Functions (Part 6 of 14)*

Signal	Subblock	Bits	Description
fcip_sm	DLL dlcmsm	[260:258]	Transmit_InitFC state machine. Transmit_Init flow control state encoding: <ul style="list-style-type: none"> 000: idle 001: prep0 010: prep1 011: initfc_p 100: initfc_np 101: initfc_cpl 110: initfc_wt 111: reserved
rxfc_sm	DLL dlcmsm	[263:261]	Receive_InitFC state machine. Receive_Init flow control state encoding: <ul style="list-style-type: none"> 000: idle 001: ifc1_p 010: ifc1_np 011: ifc1_cpl 100: ifc2 111: reserved
flag_fi1	DLL dlcmsm	[264]	Flag_fi1. FI1 flag as defined in the PCI Express Base Specification Revision 1.0a
flag_fi2	DLL dlcmsm	[265]	Flag_fi2. FI2 flag as defined in the PCI Express Base Specification Revision 1.0a
rxfc_sm	DLL rtry	[268:266]	Retry state machine. Retry State Machine encoding: <ul style="list-style-type: none"> 000: idle 001: rtry_ini 010: rtry_wt0 011: rtry_wt1 100: rtry_req 101: rtry_tlp 110: rtry_end 111: reserved
storebuf_sm	DLL rtry	[270:269]	Retry buffer storage state machine. Retry buffer storage state machine encoding: <ul style="list-style-type: none"> 00: idle 01: rtry 10: str_tlp 11: reserved
mem_replay	DLL rtry	[271]	Retry buffer running. This signal keeps track of transaction layer packets that have been sent but not yet acknowledged. The replay timer is also running when this bit is set except if a replay is currently performed.
mem_rtry	DLL rtry	[272]	Memorize replay request. This signal indicates that a replay time-out event has occurred or that a NAK DLLP has been received.

Table B-15. *test_out Signals for the ×1 and ×4 MegaCore Functions (Part 7 of 14)*

Signal	Subblock	Bits	Description
replay_num	DLL rtry	[274: 273]	Replay number counter. This signal counts the number of replays performed by the MegaCore function for a particular transaction layer packet (as described in the PCI Express Base Specification Revision 1.0a).
val_nak_r	DLL rtry	[275]	ACK/NAK DLLP received. This signal reports that an ACK or a NAK DLLP has been received. The <code>res_nak_r</code> , <code>tlp_ack</code> , <code>err_dl</code> , and <code>no_rtry</code> signals detail the type of ACK/NAK DLLP received.
res_nak_r	DLL rtry	[276]	NAK DLLP parameter. This signal reports that the received ACK/NAK DLLP is NAK.
tlp_ack	DLL rtry	[277]	Real ACK DLLP parameter. This signal reports that the received ACK DLLP acknowledges one or several transaction layer packets in the retry buffer.
err_dl	DLL rtry	[278]	Error ACK/NAK DLLP parameter. This signal reports that the received ACK/NAK DLLP has a sequence number higher than the sequence number of the last transmitted transaction layer packet.
no_rtry	DLL rtry	[279]	No retry on NAK DLLP parameter. This signal reports that the received NAK DLLP sequence number corresponds to the last acknowledged transaction layer packet.
txdl_sm	DLL txdl	[282: 280]	Transmit transaction layer packet state machine. Transmit transaction layer packet state machine encoding: <ul style="list-style-type: none"> 000: idle 001: tlp1 010: tlp2 011: tlp3a 100: tlp5a (ECRC only) 101: tlp6a (ECRC only) 111: reserved This signal can be used to inject an LCRC or ECRC error.
tx3b tx4b tx5b	DLL txdl	[283]	Transaction layer packet transmitted. This signal is set on the last DWORD of the packet where the LCRC is added to the packet. This signal can be used to inject an LCRC or ECRC error.
tx0	DLL txdl	[284]	DLLP transmitted. This signal is set when a DLLP is sent to the physical layer. This signal can be used to inject a CRC on a DLLP.
gnt	DLL txdl	[292: 285]	Data link layer transmit arbitration result. This signal reports the arbitration result between a DLLP and a transaction layer packet: <ul style="list-style-type: none"> bit 0: InitFC DLLP bit 1: ACK DLLP (high priority) bit 2: UFC DLLP (high priority) bit 3: PM DLLP bit 4: TXN transaction layer packet bit 5: RPL transaction layer packet bit 6: UFC DLLP (low priority) bit 7: ACK DLLP (low priority)

Table B-15. *test_out Signals for the ×1 and ×4 MegaCore Functions (Part 8 of 14)*

Signal	Subblock	Bits	Description
sop	DLL txdl	[293]	Data link layer to PHY start of packet. This signal reports that an start of data link layer packet (SDP) or start of transaction layer packet (STP) symbol is in transition to the physical layer.
eop	DLL txdl	[294]	Data link layer to PHY end of packet. This signal reports that an end of a nullified transaction layer packet (EDB) or the end of a transaction layer packet of data link layer packet (END) symbol in transition to the physical layer. When SOP and EOP are transmitted together, it indicates that the packet is a DLLP. Otherwise the packet is a transaction layer packet.
eot	DLL txdl	[295]	Data link layer to PHY end of transmit. This signal reports that the data link layer has finished its previous transmission and enables the physical layer to go to low-power state or to recovery.
init_lat_timer	DLL rxdl	[296]	Enable ACK latency timer. This signal reports that the ACK latency timer is running.
req_lat	DLL rxdl	[297]	ACK latency timeout. This signal reports that an ACK/NAK DLLP retransmission has been scheduled due to the ACK latency timer expiring.
tx_req_nak or tx_snd_nak	DLL rxdl	[298]	ACK/NAK DLLP requested for transmission. This signal reports that an ACK/NAK DLLP is currently requested for transmission.
tx_res_nak	DLL rxdl	[299]	ACK/NAK DLLP type requested for transmission. This signal reports that type of ACK/NAK DLLP scheduled for transmission: ■ 0: ACK ■ 1: NAK
rx_val_pm	DLL rxdl	[300]	Received PM DLLP. This signal reports that a PM DLLP has been received (the specific type is indicated by rx_vcid_fc): 000: PM_Enter_L1 001: PM_Enter_L23 011: PM_AS_Request_L1 100: PM_Request_ACK
rx_val_fc	DLL rxdl	[301]	Received flow control DLLP. This signal reports that a PM DLLP has been received. The type of flow control DLLP is indicated by rx_typ_fc and rx_vcid_fc.
rx_typ_fc	DLL rxdl	[305: 302]	Received flow control DLLP type parameter. This signal reports the type of received flow control DLLP: ■ 0100: InitFC1_P ■ 0101: InitFC1_NP ■ 0110: InitFC1_CPL ■ 1100: InitFC2_P ■ 1101: InitFC2_NP ■ 1110: InitFC2_CPL ■ 1000: UpdateFC_P ■ 1001: UpdateFC_NP ■ 1010: UpdateFC_CPL

Table B-15. *test_out Signals for the ×1 and ×4 MegaCore Functions (Part 9 of 14)*

Signal	Subblock	Bits	Description
rx_vcid_fc	DLL rxdl	[308:306]	Received flow control DLLP virtual channel ID parameter. This signal reports the virtual channel ID of the received flow control DLLP: <ul style="list-style-type: none"> ■ 000: VCID 0 ■ 001: VCID 1 ■ ... ■ 111: VCID 7 This signal also indicates the type of PM DLLP received.
crcinv	DLL rxdl	[309]	Received nullified transaction layer packet. This signal indicates that a nullified transaction layer packet has been received.
crcerr	DLL rxdl	[310]	Received transaction layer packet with LCRC error. This signal reports that a transaction layer packet has been received that contains an LCRC error.
crcval eqseq_r	DLL rxdl	[311]	Received valid transaction layer packet. This signal reports that a valid transaction layer packet has been received that contains the correct sequence number. Such a transaction layer packet is transmitted to the application layer.
crcval !eqseq_r infseq_r	DLL rxdl	[312]	Received duplicated transaction layer packet. This signal indicates that a transaction layer packet has been received that has already been correctly received. Such a transaction layer packet is silently discarded.
crcval !eqseq_r !infseq_r	DLL rxdl	[313]	Received erroneous transaction layer packet. This signal indicates that a transaction layer packet has been received that contains a valid LCRC but a non-sequential sequence number (higher than the current sequence number).
rx_err_frame	DLL dlink	[314]	Data link layer framing error detected. This signal indicates that received data cannot be considered as a DLLP or transaction layer packet, in which case a receive port error is generated and link retraining is initiated.
tlp_count	DLL rtry	[319:315]	Transaction layer packet count in retry buffer. This signal indicates the number of transaction layer packets stored in the retry buffer (saturation limit is 31).

Table B-15. *test_out Signals for the ×1 and ×4 MegaCore Functions (Part 10 of 14)*

Signal	Subblock	Bits	Description
ltssm_r	MAC ltssm	[324: 320]	LTSSM state. LTSSM state encoding: <ul style="list-style-type: none"> ■ 00000: detect.quiet ■ 00001: detect.active ■ 00010: polling.active ■ 00011: polling.compliance ■ 00100: polling.configuration ■ 00101: reserved (polling.speed) ■ 00110: config.linkwidthstart ■ 00111: config.linkaccept ■ 01000: config:lanenumaccept ■ 01001: config.lannumwait ■ 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: L0s (transmit only) ■ 10110: L1.entry ■ 10111: L1.idle ■ 11000: L2.idle ■ 11001: L2.transmit.wake
rxl0s_sm	MAC ltssm	[326: 325]	Receive L0s state. Receive L0s state machine: <ul style="list-style-type: none"> ■ 00: inact ■ 01: idle ■ 10: fts ■ 11: out.recovery
txl0s_sm	MAC ltssm	[329: 327]	Tx L0s state. Transmit L0s state machine: <ul style="list-style-type: none"> ■ 000: inact ■ 001: entry ■ 010: idle ■ 011: fts ■ 100: out.l0
timeout	MAC ltssm	[330]	LTSSM timeout. This signal serves as a flag that indicates that the LTSSM time-out condition has been reached for the current LTSSM state.

Table B-15. *test_out Signals for the ×1 and ×4 MegaCore Functions (Part 11 of 14)*

Signal	Subblock	Bits	Description
txos_end	MAC ltssm	[331]	Transmit LTSSM exit condition. This signal serves as a flag that indicates that the LTSSM exit condition for the next state (to go to L0) has been completed. If the next state is not reached in a timely manner, it is due to a problem on the receiver.
tx_ack	MAC ltssm	[332]	Transmit PLP acknowledge. This signal is active for 1 clock cycle when the requested physical layer packet (PLP) has been sent to the link. The type of packet is defined by the tx_ctrl signal.
tx_ctrl	MAC ltssm	[335: 333]	Transmit PLP type. This signal indicates the type of transmitted PLP: <ul style="list-style-type: none"> ■ 000: Electrical Idle ■ 001: Receiver detect during Electrical Idle ■ 010: TS1 OS ■ 011: TS2 OS ■ 100: D0.0 idle data ■ 101: FTS OS ■ 110: IDL OS ■ 111: Compliance pattern
txrx_det	MAC ltssm	[343: 36]	Receiver detect result. This signal serves as a per lane flag that reports the receiver detection result. The 4 MSBs are always zero.
tx_pad	MAC ltssm	[351: 344]	Force PAD on transmitted TS pattern. This is a per lane internal signal that forces PAD transmission on the link and lane field of the transmitted TS1/TS2 OS. The MegaCore function does not initialize lanes that are driving this pattern. The 4 MSB are always zero.
rx_ts1	MAC ltssm	[359: 352]	Received TS1: This signal indicates that a TS1 has been received on the specified lane. This signal is cleared when a new state is reached by the LTSSM state machine. The 4 MSBs are always zero.
rx_ts2	MAC ltssm	[367: 360]	Received TS2. This signal indicates that a TS1 has been received on the specified lane. This signal is cleared when a the LTSSM state machine reaches a new state. The 4 MSBs are always zero.
rx_8d00	MAC ltssm	[375: 368]	Received 8 D0.0 symbol. This signal indicates that 8 consecutive idle data symbols have been received. This signal is meaningful for config.idle and recovery.idle states. The 4 MSBs are always zero.
rx_idl	MAC ltssm	[383: 376]	Received IDL OS. This signal indicates that an IDL OS has been received on a per lane basis. The 4 MSBs are always zero.
rx_linkpad	MAC ltssm	[391: 384]	Received link pad TS. This signal indicates that the link field of the received TS1/TS2 is set to PAD for the specified lane. The 4 MSBs are always zero.
rx_lanepad	MAC ltssm	[399: 392]	Received lane pad TS. This signal indicates that the lane field of the received TS1/TS2 is set to PAD for the specified lane. The 4 MSBs are always zero.

Table B-15. *test_out Signals for the ×1 and ×4 MegaCore Functions (Part 12 of 14)*

Signal	Subblock	Bits	Description
rx_tsnnum	MAC ltssm	[407: 400]	Received consecutive identical TSNumber. This signal reports the number of consecutive identical TS1/TS2 which have been received with exactly the same parameters since entering this state. When the maximum number is reached, this signal restarts from zero. This signal corresponds to the lane configured as logical lane 0 (may vary depending on lane reversal).
lane_act	MAC ltssm	[411: 408]	Lane active mode. This signal indicates the number of Lanes that have been configured during training: <ul style="list-style-type: none"> ■ 0001: 1 lane ■ 0010: 2 lanes ■ 0100: 4 lanes
lane_rev	MAC ltssm	[415: 412]	Reserved.
count0	MAC deskew	[418: 416]	Deskew FIFO count lane 0. This signal indicates the number of Words in the deskew FIFO for physical lane 0.
count1	MAC deskew	[421: 419]	Deskew FIFO count lane 1. This signal indicates the number of Words in the deskew FIFO for physical lane 1.
count2	MAC deskew	[424: 422]	Deskew FIFO count lane 2. This signal indicates the number of Words in the deskew FIFO for physical lane 2.
count3	MAC deskew	[427: 425]	Deskew FIFO count lane 3. This signal indicates the number of Words in the deskew FIFO for physical lane 3.
Reserved	N/A	[439: 428]	Reserved.
err_deskew	MAC deskew	[447: 440]	Deskew FIFO error. This signal indicates whether a deskew error (deskew FIFO overflow) has been detected on a particular physical lane. In such a case, the error is considered a receive port error and retraining of the link is initiated. The 4 MSBs are hard-wired to zero.
rdusedw0	PCSO	[451: 448]	Elastic buffer counter 0. This signal indicates the number of symbols in the elastic buffer. Monitoring the elastic buffer counter of each lane can highlight the PPM between the receive clock and transmit clock as well as the skew between lanes. Only meaningful when using the Stratix GX PHY interface.

Table B-15. *test_out Signals for the ×1 and ×4 MegaCore Functions (Part 13 of 14)*

Signal	Subblock	Bits	Description
rxstatus0	PCS0	[454:452]	<p>PIPE rxstatus 0. This signal is used to monitor errors detected and reported on a per lane basis. The encoding records 8 different states:</p> <ul style="list-style-type: none"> ■ 000: Receive data OK ■ 001: 1 SKP added ■ 010: 1 SKP removed ■ 011: Receiver detected ■ 100: 8B10B decode error ■ 101: Elastic buffer overflow ■ 110: Elastic buffer underflow ■ 111: Running disparity error <p>Not meaningful when using the external PHY interface.</p>
rxpolarity0	PCS0	[455]	<p>PIPE polarity inversion 0. When asserted, the LTSSM requires the PCS subblock to invert the polarity of the received 10-bit data during training.</p>
rdusedw1	PCS1	[459:456]	<p>Elastic buffer counter 1. This signal indicates the number of symbols in the elastic buffer.</p> <p>Monitoring the elastic buffer counter of each lane can highlight the PPM between the receive clock and transmit clock as well as the skew between lanes. Only meaningful when using the Stratix GX PHY interface.</p>
rxstatus1	PCS1	[462:460]	<p>PIPE rxstatus 1. This signal is used to monitor errors detected and reported on a per lane basis. The encoding records 8 different states:</p> <ul style="list-style-type: none"> ■ 000: Receive data OK ■ 001: 1 SKP added ■ 010: 1 SKP removed ■ 011: Receiver detected ■ 100: 8B10B decode error ■ 101: Elastic buffer overflow ■ 110: Elastic buffer underflow ■ 111: Running disparity error <p>Not meaningful when using the external PHY interface.</p>
rxpolarity1	PCS1	[463]	<p>PIPE polarity inversion 1. When asserted, the LTSSM requires the PCS subblock to invert the polarity of the received 10-bit data during training.</p>
rdusedw2	PCS2	[467:464]	<p>Elastic buffer counter 2. This signal reports the number of symbols in the elastic buffer.</p> <p>Monitoring the elastic buffer counter of each lane can highlight the PPM between the receive clock and transmit clock as well as the skew between lanes. Only meaningful when using the Stratix GX PHY interface.</p>

Table B-15. *test_out* Signals for the $\times 1$ and $\times 4$ MegaCore Functions (Part 14 of 14)

Signal	Subblock	Bits	Description
rxstatus2	PCS2	[470:468]	PIPE rxstatus 2. This signal is used to monitor errors detected and reported on a per lane basis. The encoding records 8 different states: <ul style="list-style-type: none"> 000: receive data OK 001: 1 SKP added 010: 1 SKP removed 011: Receiver detected 100: 8B10B decode error 101: Elastic buffer overflow 110: Elastic buffer underflow 111: Running disparity error Not meaningful when using the external PHY interface.
rxpolarity2	PCS2	[471]	PIPE polarity inversion 2. When asserted, the LTSSM requires the PCS subblock to invert the polarity of the received 10-bit data during training.
rdusedw3	PCS3	[475:472]	Elastic buffer counter 3. This signal reports the number of symbols in the elastic Buffer. Monitoring the elastic buffer counter of each lane can highlight the PPM between the receive clock and transmit clock as well as the skew between lanes. Only meaningful when using the Stratix GX PHY interface.
rxstatus3	PCS3	[478:476]	PIPE rxstatus 3. This signal is used to monitor errors detected and reported on a per lane basis. The encoding records the following 8 different states: <ul style="list-style-type: none"> 000: receive data OK 001: 1 SKP added 010: 1 SKP removed 011: Receiver detected 100: 8B10B decode error 101: Elastic buffer overflow 110: Elastic buffer underflow 111: Running disparity error Not meaningful when using the external PHY interface.
rxpolarity3	PCS3	[479]	PIPE polarity inversion 3. When asserted, the LTSSM requires the PCS subblock to invert the polarity of the received 10-bit data during training.
Reserved	PCS4	[511:480]	Reserved.

You must implement specific logic in order to use the error-injection capabilities of the *test_in* port. For example, to force an LCRC error on the next transmitted transaction layer packet, *test_in*[21] must be asserted for 1 clock cycle when transmit *txdl_sm*, (*test_out*[282:280]) is in a non-idle state.

Soft IP x8 MegaCore Function test_out

Table B-16 describes the test-out signals for the soft IP and descriptor/data x8 MegaCore functions.

Table B-16. test_out Signals for the x8 MegaCore Functions (Part 1 of 3)

Signal	Subblock	Bits	Description
ltssm_r	MAC ltssm	[4:0]	LTSSM state: LTSSM state encoding: <ul style="list-style-type: none"> ■ 00000: detect.quiet ■ 00001: detect.active ■ 00010: polling.active ■ 00011: polling.compliance ■ 00100: polling.configuration ■ 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: L0s ■ 10110: L1.entry ■ 10111: L1.idle ■ 11000: L2.idle ■ 11001: L2 transmit.wake
rxl0s_sm	MAC ltssm	[6:5]	Receive L0s state: Receive L0s state machine <ul style="list-style-type: none"> ■ 00: inact ■ 01: idle ■ 10: fts ■ 11: out.recovery

Table B-16. test_out Signals for the x8 MegaCore Functions (Part 2 of 3)

Signal	Subblock	Bits	Description
txl0s_sm	MAC ltssm	[9:7]	Tx L0s state: Transmit L0s state machine <ul style="list-style-type: none"> ■ 000: <code>inact</code> ■ 001: <code>entry</code> ■ 010: <code>idle</code> ■ 011: <code>fts</code> ■ 100: <code>out.l0</code>
timeout	MAC ltssm	[10]	LTSSM Timeout: This signal is a flag that indicates that the LTSSM timeout condition has been reached for the current LTSSM state.
txos_end	MAC ltssm	[11]	Transmit LTSSM exit condition: This signal is a flag that indicates that the LTSSM exit condition for the next state (in order to go to L0) has been completed. If the next state is not reached in a timely manner, it is due to a problem on the receiver.
tx_ack	MAC ltssm	[12]	Transmit PLP acknowledge: This signal is active for 1 clock cycle when the requested physical layer packet (PLP) has been sent to the link. The type of packet is defined by <code>tx_ctrl</code> .
tx_ctrl	MAC ltssm	[15:13]	Transmit PLP type: The encoding records 8 states: <ul style="list-style-type: none"> ■ 000: Electrical Idle ■ 001: Receiver detect during ■ 010: TS1 OS ■ 011: TS2 OS ■ 100: D0.0 idle data ■ 101: FTS OS ■ 110: IDL OS ■ 111: Compliance pattern
txrx_det	MAC ltssm	[23:16]	Receiver detect result: This signal is a per-lane flag that reports the receiver detection result.
tx_pad	MAC ltssm	[31:24]	Force PAD on transmitted TS pattern: This is a per-lane internal signal that forces PAD transmission on the link and lane field of the transmitted TS1/TS2 OS. The MegaCore function does not initialize lanes that are driving this pattern.
rx_ts1	MAC ltssm	[39:32]	Received TS1: This signal indicates that a TS1 has been received on the specified lane. When the LTSSM state machine reaches a new state, this signal is cleared.
rx_ts2	MAC ltssm	[47:40]	Received TS2: This signal indicates that a TS1 has been received on the specified lane. When the LTSSM state machine reaches a new state, this signal is cleared.
rx_8d00	MAC ltssm	[55:48]	Received 8 D0.0 symbol: This signal indicates that 8 consecutive Idle data symbols have been received. This signal is meaningful for <code>config.idle</code> and <code>recovery.idle</code> states.
rx_id1	MAC ltssm	[63:56]	Received 8 D0.0 symbol: This signal indicates that 8 consecutive Idle data symbols have been received. This signal is meaningful for <code>config.idle</code> and <code>recovery.idle</code> states.

Table B-16. test_out Signals for the x8 MegaCore Functions (Part 3 of 3)

Signal	Subblock	Bits	Description
rx_linkpad	MAC ltssm	[71:64]	Received Link Pad TS: This signal indicates that the Link field of the received TS1/TS2 is set to PAD for the specified lane.
rx_lanepad	MAC ltssm	[79:72]	Received Lane Pad TS: This signal indicates that the Lane field of the received TS1/TS2 is set to PAD for the specified lane.
rx_tsnum	MAC ltssm	[87:80]	Received Consecutive Identical TSNumber: This signal reports the number of consecutive identical TS1/TS2 which have been received with exactly the same parameters since entering this state. When the maximum number is reached, this signal restarts from zero. Note that this signal corresponds to the lane configured as logical lane 0.
lane_act	MAC ltssm	[91:88]	Lane Active Mode: This signal indicates the number of lanes that have been configured during training: <ul style="list-style-type: none"> ■ 0001: 1 lane ■ 0010: 2 lanes ■ 0100: 4 lanes ■ 1000: 8 lanes
lane_rev	MAC ltssm	[95:92]	Reserved.
count0	MAC deskew	[98:96]	Deskew FIFO count lane 0: This signal indicates the number of words in the deskew FIFO for physical lane 0.
count1	MAC deskew	[101:99]	Deskew FIFO count lane 1: This signal indicates the number of Words in the deskew FIFO for physical lane 1.
count2	MAC deskew	[104:102]	Deskew FIFO count lane 2: This signal indicates the number of Words in the deskew FIFO for physical lane 2.
count3	MAC deskew	[107:105]	Deskew FIFO count lane 3: This signal indicates the number of Words in the deskew FIFO for physical lane 3.
count4	MAC deskew	[110:108]	Deskew FIFO count lane 4: This signal indicates the number of Words in the deskew FIFO for physical lane 4.
count5	MAC deskew	[113:111]	Deskew FIFO count lane 5: This signal indicates the number of Words in the deskew FIFO for physical lane 5.
count6	MAC deskew	[116:114]	Deskew FIFO count lane 6: This signal indicates the number of Words in the deskew FIFO for physical lane 6.
count7	MAC deskew	[119:117]	Deskew FIFO count lane 7: This signal indicates the number of Words in the deskew FIFO for physical lane 7.
err_deskew	MAC deskew	[127:120]	Deskew FIFO error: This signal indicates whether a deskew error (deskew FIFO overflow) has been detected on a particular physical Lane. In such a case, the error is considered a Receive Port error and retraining of the Link is initiated.

Soft IP Implementation x1, x4, and x8 test_in

Table B-17 describes test_in signals. The test_in bus is the same for the x1, x4, and x8 MegaCore functions using the soft IP implementation or the descriptor/data interface.

Table B-17. test_in Signals (Part 1 of 4)

Signal	Subblock	Bit	Description
test_sim	MAC Itssm	[0]	Simulation mode. This signal can be set to 1 to accelerate MegaCore function initialization by changing many initialization counter values from their PCI Express Specification values to much lower values. This control is only functional when doing IP functional simulation. It has no effect once the compiled design has been downloaded to a device or in gate-level simulation models.
test_lpbk	MAC Itssm	[1]	Loopback master. This signal can be set to 1 to direct the link to loopback (in master mode). This bit is reserved on the x8 MegaCore function.
test_discr	MAC Itssm	[2]	Descramble mode. This signal can be set to 1 during initialization to disable data scrambling.
test_nonc_phy	MAC Itssm	[3]	Force_rxdet mode. This signal can be set to 1 in cases where the PHY implementation does not support the Rx Detect feature. The MegaCore function always detects the maximum number of receivers during the detect state, and only goes to compliance state if at least one lane has the correct pattern. This signal is forced inside the MegaCore function for Stratix GX PHY implementations.
test_boot	CFGcfcgch k	[4]	Remote boot mode. When asserted, this signal disables the BAR check if the link is not initialized and the boot is located behind the component.
test_compliance	MAC Itssm	[6:5]	Compliance test mode. Disable/force compliance mode: bit 5 completely disables compliance mode. bit 6 forces compliance mode.
test_pwr	CFG PMGT	[7]	Disable low power state negotiation. When asserted, this signal disables all low power state negotiation and entry. This mode can be used when the attached PHY does not support the electrical idle feature used in low-power link states. The MegaCore function will not attempt to place the link in Tx L0s state or L1 state when this bit is asserted. For Stratix GX PHY implementations, this bit is forced to a 1 inside the MegaCore function.

Table B-17. *test_in Signals (Part 2 of 4)*

Signal	Subblock	Bit	Description
test_pcerror	PCS	[13:8]	<p>Lane error injection. Disable/force compliance mode. The first three bits indicate the following modes:</p> <ul style="list-style-type: none"> ■ test_pcerror[2:0]: 000: normal mode ■ test_pcerror[2:0]: 001: inject data error ■ test_pcerror[2:0]: 010: inject disparity error ■ test_pcerror[2:0]: 011: inject different data ■ test_pcerror[2:0]: 100: inject SDP instead of END ■ test_pcerror[2:0]: 101: inject STP instead of END ■ test_pcerror[2:0]: 110: inject END instead of data ■ test_pcerror[2:0]: 111: inject EDB instead of END <p>The last three bits indicate the lane:</p> <ul style="list-style-type: none"> ■ test_pcerror[5:3]: 000: on lane 0 ■ test_pcerror[5:3]: 001: on lane 1 ■ test_pcerror[5:3]: 010: on lane 2 ■ test_pcerror[5:3]: 011: on lane 3
test_rxerrtlp	DLL	[14]	Force transaction layer packet LCRC error detection. When asserted, this signal forces the MegaCore function to treat the next received transaction layer packet as if it had an LCRC error. These bits are reserved on the x8 MegaCore function.
test_rxerrdllp	DLL	[16:15]	<p>Force DLLP CRC error detection. This signal forces the MegaCore function to check the next DLLP for a CRC error:</p> <ul style="list-style-type: none"> ■ 00: normal mode ■ 01: ACK/NAK ■ 10: PM ■ 11: flow control These bits are reserved on the x8 MegaCore function.
test_replay	DLL	[17]	Force retry buffer. When asserted, this signal forces the retry buffer to initiate a retry. These bits are reserved on the x8 MegaCore function.
test_acknak	DLL	[19:18]	<p>Replace ACK by NAK. This signal replaces an ACK by a NAK with following sequence number:</p> <ul style="list-style-type: none"> ■ 00: normal mode ■ 01: Same sequence number as the ACK ■ 10: Sequence number incremented ■ 11: Sequence number decremented <p>If unused, these bits should be hard-wired to 0 to remove unused logic. These bits are reserved on the x8 MegaCore function.</p>
test_ecrcerr	DLL	[20]	Inject ECRC error on transmission. When asserted, this signal generates an ECRC error for transmission.
test_lcrcerr	DLL	[21]	Inject LCRC error on transmission. When asserted, this signal generates an LCRC error for transmission. These bits are reserved on the x8 MegaCore function.

Table B-17. *test_in Signals (Part 3 of 4)*

Signal	Subblock	Bit	Description
test_crcerr	DLL	[23:22]	<p>Inject DLLP CRC error on transmission. Generates a CRC error when transmitting a DLLP:</p> <ul style="list-style-type: none"> ■ 00: normal ■ 01: PM error ■ 10: flow control error ■ 11: ACK error <p>If unused, these bits should be hard-wired to 0 to remove unused logic. These bits are reserved on the x8 MegaCore function.</p>
test_ufcvalue	TRN	[28:24]	<p>Generate wrong value for update flow control. This signal forces an incorrect value when updating flow control credits by adding or removing one credit in the credits allocated field when a transaction layer packet is extracted from the receive buffer and sent to the application layer:</p> <ul style="list-style-type: none"> ■ 00000: normal mode ■ 00001: UFC_P error on header (+1/0) ■ 00010: UFC_P error on data (0/+1) ■ 00011: UFC_P error on header/data (+1/+1) ■ 00100: UFC_NP error on header (+1/0) ■ 00101: UFC_NP error on data (0/+1) ■ 00110: UFC_NP error on header/data (+1/+1) ■ 00111: UFC_CPL error on header (+1/0) ■ 01000: UFC_CPL error on data (0/+1) ■ 01001: UFC_CPL error header/data (+1/+1) ■ 01010: UFC_P error on header (-1/0) ■ 01011: UFC_P error on data (0/-1) ■ 01100: UFC_P error on header/data (-1/-1) ■ 01101: UFC_NP error on header (-1/0) ■ 01110: UFC_NP error on data (0/-1) ■ 01111: UFC_NP error on header/data (-1/-1) ■ 10000: UFC_CPL error on header (-1/0) ■ 10001: UFC_CPL error on data (0/-1) ■ 10010: UFC_CPL error header/data (-1/-1) ■ 10011: UFC_P error on header/data (+1/-1) ■ 10100: UFC_P error on header/data (-1/+1) ■ 10101: UFC_NP error on header/data (+1/-1) ■ 10110: UFC_NP error on header/data (-1/+1) ■ 10111: UFC_CPL error header/data (+1/-1) ■ 11000: UFC_CPL error header/data (-1/+1) <p>If unused, these bits should be hard-wired to 0 to remove unused logic. These bits are reserved on the x8 MegaCore function.</p>

Table B-17. *test_in Signals (Part 4 of 4)*

Signal	Subblock	Bit	Description
test_vcselect	TRN	[31:29]	<p>Virtual channel test selection. This signal indicates which virtual channel is currently considered by the test-out interface (<code>test_out[255:131]</code>). This virtual channel test selection is the select input to a mux that switches a portion of the <code>test_out</code> bus to output debug signals from different virtual channels (VC). For example:</p> <ul style="list-style-type: none"> ■ <code>test_vcselect[31:29]:000:test_out[255:131]</code> describes activity for VC0 ■ <code>test_vcselect[31:29]:001:test_out[255:131]</code> describes activity for VC1 ■ <code>test_vcselect[31:29]:010:test_out[255:131]</code> describes activity for VC2 ... <p>Certain bits of this signal should be set to 0 to remove unused logic:</p> <ul style="list-style-type: none"> ■ 1 virtual channel (or signal completely unused): set all three bits to 000 ■ 2 virtual channels: set the 2 MSBs to 00 ■ 3 or 4 virtual channels: set the MSB to 0. These bits are reserved on the x8 MegaCore function.

Avalon-ST Test Signals

The Avalon-ST test signals carry the status of the Tx adapter FIFO in Avalon-ST implementations for both the hard IP and soft IP MegaCore functions. These signals are for debug purposes only and do not need to be used for normal operation.

Table B-18.

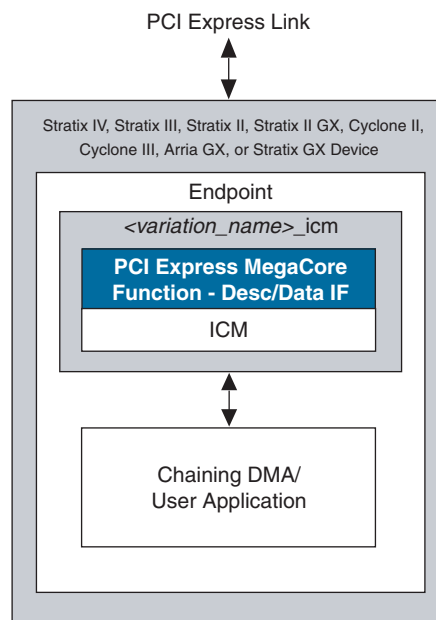
Signal	Width	Dir	Description
<code>rx_st_fifo_full0</code>	1	0	Indicates that the Avalon-ST adapter Rx FIFO is almost full. This signal is used for debug only and not to qualify data.
<code>rx_st_fifo_empty0</code>	1	0	Indicates that the Avalon-ST adapter Rx FIFO is empty. This signal is used for debug only and not to qualify data.

When the Descriptor/Data PCI Express MegaCore function is generated, the example designs are generated with an Incremental Compile Module. This module facilitates timing closure using Quartus II incremental compilation and is provided for backward compatibility only. The ICM facilitates incremental compilation by providing a fully registered interface between the user application and the PCI Express transaction layer. (Refer to [Figure C-1](#)) With the ICM, you can lock down the placement and routing of the PCI Express MegaCore function to preserve timing while changes are made to your application. Altera provides the ICM as clear text to allow its customization if required.



The ICM is provided for backward compatibility only. New designs using the Avalon-ST interface should use the Avalon-ST PCI Express MegaCore instead.

Figure C-1. Design Example with ICM



ICM Features

The ICM provides the following features:

- A fully registered boundary to the application to support design partitioning for incremental compilation
- An Avalon-ST protocol interface for the application at the Rx, Tx, and interrupt (MSI) interfaces for designs using the Avalon-ST interface
- Optional filters and ACK's for PCI Express message packets received from the transaction layer

- Maintains packet ordering between the Tx and MSI Avalon-ST interfaces
- Tx bypassing of non-posted PCI Express packets for deadlock prevention

ICM Functional Description

This section describes details of the ICM within the following topics:

- “<variation_name>_icm Partition”
- “ICM Block Diagram”
- “ICM Files”
- “ICM Application-Side Interface”

<variation_name>_icm Partition

When you generate a PCI Express MegaCore function, the MegaWizard produces module, <variation_name>_icm in the subdirectory <variation_name>_examples\common\incremental_compile_module, as a wrapper file that contains the MegaCore function and the ICM module. (Refer to [Figure C-1](#).) Your application connects to this wrapper file. The wrapper interface resembles the PCI Express MegaCore function interface, but replaces it with an Avalon-ST interface. (Refer to [Table C-1](#).)



The wrapper interface omits some signals from the MegaCore function to maximize circuit optimization across the partition boundary. However, all of the MegaCore function signals are still available on the MegaCore function instance and can be wired to the wrapper interface by editing the <variation_name>_icm file as required.

By setting this wrapper module as a design partition, you can preserve timing of the MegaCore function using the incremental synthesis flow.

[Table C-1](#) describes the <variation_name>_icm interfaces.

Table C-1. <variation_name>_icm Interface Descriptions (Part 1 of 2)

Signal Group	Description
Transmit Datapath	ICM Avalon-ST Tx interface. These signals include tx_stream_valid0, tx_stream_data0, tx_stream_ready0, tx_stream_cred0, and tx_stream_mask0. Refer to Table C-4 on page C-8 for details.
Receive Datapath	ICM interface. These signals include rx_stream_valid0, rx_stream_data0, rx_stream_ready0, and rx_stream_mask0. Refer to Table C-3 on page C-7 for details.
Configuration()	Part of ICM sideband interface. These signals include cfg_busdev_icm, cfg_devcsr_icm, and cfg_linkcsr_icm.
Completion interfaces	Part of ICM sideband interface. These signals include cpl_pending_icm, cpl_err_icm, pex_msi_num_icm, and app_int_sts_icm. Refer to Table C-6 on page C-10 for details.
Interrupt	ICM Avalon-ST MSI interface. These signals include msi_stream_valid0, msi_stream_data0, and msi_stream_ready0. Refer to Table C-5 on page C-9 for details.
Test Interface	Part of ICM sideband signals; includes test_out_icm. Refer to Table C-6 on page C-10 for details.

Table C-1. <variation_name>_icm Interface Descriptions (Part 2 of 2)

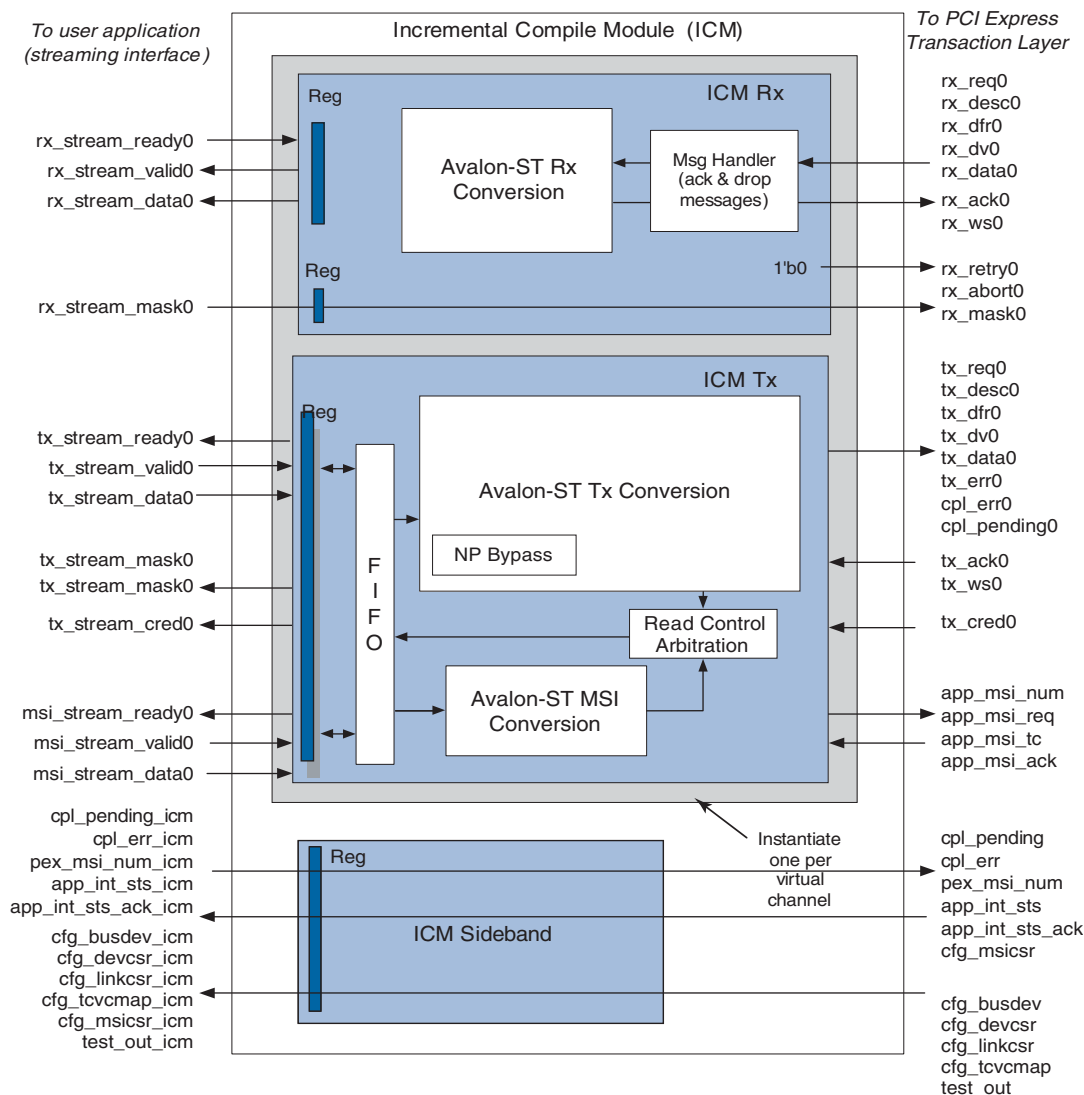
Signal Group	Description
Global Interface	MegaCore function signals; includes refclk, clk125_in, clk125_out, npor, srst, crst, ls_exit, hotrst_exit, and dlup_exit. Refer to Chapter 5, Signals for details.
PIPE Interface	MegaCore function signals; includes tx, rx, pipe_mode, txdata0_ext, txdatak0_ext, txdetectrx0_ext, txelecidle0_ext, txcompliance0_ext, rxpolarity0_ext, powerdown0_ext, rxdata0_ext, rxdatak0_ext, rxvalid0_ext, phystatus0_ext, rxelecidle0_ext, rxstatus0_ext, txdata0_ext, txdatak0_ext, txdetectrx0_ext, txelecidle0_ext, txcompliance0_ext, rxpolarity0_ext, powerdown0_ext, rxdata0_ext, rxdatak0_ext, rxvalid0_ext, phystatus0_ext, rxelecidle0_ext, and rxstatus0_ext. Refer Chapter 5, Signals for details.
Maximum Completion Space Signals	This signal is ko_cpl_spc_vc<n>, and is not available at the <variation_name>_icm ports(.). Instead, this static signal is regenerated for the user in the <variation_name>_example_pipen1b module. This signal is described in “ Completion Interface Signals for Descriptor/Data Interface ” on page 5-59

Note to Table C-1:

- (1) Cfg_tcvcmap is available from the ICM module, but not wired to the <variation_name>_icm ports. Refer to [Table C-6](#) on [page C-10](#) for details.

ICM Block Diagram

[Figure C-2](#) shows the ICM block diagram.

Figure C-2. ICM Block Diagram

The ICM comprises four main sections:

- “Rx Datapath”
- “Tx Datapath”
- “MSI Datapath”
- “Sideband Datapath”

All signals between the PCI Express MegaCore function and the user application are registered by the ICM. The design example implements the ICM interface with one virtual channel. For multiple virtual channels, duplicate the Rx and Tx Avalon-ST interfaces for each virtual channel.

Rx Datapath

The Rx datapath contains the Rx boundary registers (for incremental compile) and a bridge to transport data from the PCI Express MegaCore function interface to the Avalon-ST interface. The bridge autonomously acks all packets received from the PCI Express MegaCore function. For simplicity, the `rx_abort` and `rx_retry` features of the MegaCore function are not used, and `Rx_mask` is loosely supported. (Refer to [Table C-3 on page C-7](#) for further details.) The Rx datapath also provides an optional message-dropping feature that is enabled by default. The feature acknowledges PCI Express message packets from the PCI Express MegaCore function, but does not pass them to the user application. The user can optionally allow messages to pass to the application by setting the **DROP_MESSAGE** parameter in `altpcierrd_icm_rxbridge.v` to 1'b0. The latency through the ICM Rx datapath is approximately four clock cycles.

Tx Datapath

The Tx datapath contains the Tx boundary registers (for incremental compile) and a bridge to transport data from the Avalon-ST interface to the PCI Express MegaCore function interface. A data FIFO buffers the Avalon-ST data from the user application until the PCI Express MegaCore function accepts it. The Tx datapath also implements an NPbypass function for deadlock prevention. When the PCI Express MegaCore function runs out of non-posted (NP) credits, the ICM allows completions and posted requests to bypass NP requests until credits become available. The ICM handles any NP requests pending in the ICM when credits run out and asserts the `tx_mask` signal to the user application to indicate that it should stop sending NP requests. The latency through the ICM Tx datapath is approximately five clock cycles.

MSI Datapath

The MSI datapath contains the MSI boundary registers (for incremental compile) and a bridge to transport data from the Avalon-ST interface to the PCI Express MegaCore function interface. The ICM maintains packet ordering between the Tx and MSI datapaths. In this design example, the MSI interface supports low-bandwidth MSI requests. For example, not more than one MSI request can coincide with a single TX packet. The MSI interface assumes that the MSI function in the PCI Express MegaCore function is enabled. For other applications, you may need to modify this module to include internal buffering, MSI-throttling at the application, and so on.

Sideband Datapath

The sideband interface contains boundary registers for non-timing critical signals such as configuration signals. (Refer to [Table C-6 on page C-10](#) for details.)

ICM Files

This section lists and briefly describes the ICM files. The PCI Express MegaWizard generates all these ICM files placing them in the

`<variation name>_examples\common\incremental_compile_module` folder.

When using the Quartus II software, include the files listed in [Table C-2](#) in your design:

Table C-2. ICM Files

Filename	Description
altpcierrd_icm_top.v or altpcierrd_icm_top.vhd	This is the top-level module for the ICM instance. It contains all of the following modules listed below in column 1.
altpcierrd_icm_rx.v or altpcierrd_icm_rx.vhd	This module contains the ICM RX datapath. It instantiates the altpcierrd_icm_rxbridge and an interface FIFO.
altpcierrd_icm_rxbridge.v or altpcierrd_icm_rxbridge.vhd	This module implements the bridging required to connect the application's interface to the PCI Express transaction layer.
altpcierrd_icm_tx.v or altpcierrd_icm_tx.vhd	This module contains the ICM TX and MSI datapaths. It instantiates the altpcierrd_icm_msibridge , altpcierrd_icm_txbridge_withbypass , and interface FIFOs.
altpcierrd_icm_msibridge.v or altpcierrd_icm_msibridge.vhd	This module implements the bridging required to connect the application's Avalon-ST MSI interface to the PCI Express transaction layer.
altpcierrd_icm_txbridge_withbypass.v or altpcierrd_icm_txbridge_withbypass.vhd	This module instantiates the altpcierrd_icm_txbridge and altpcierrd_icm_tx_pktordering modules.
altpcierrd_icm_txbridge.v or altpcierrd_icm_txbridge.vhd	This module implements the bridging required to connect the application's Avalon-ST Tx interface to the MegaCore function's Tx interface.
altpcierrd_icm_tx_pktordering.v or altpcierrd_icm_tx_pktordering.vhd	This module contains the NP-Bypass function. It instantiates the npbypass FIFO and altpcierrd_icm_npbypassctl .
altpcierrd_icm_npbypassctl.v or altpcierrd_icm_npbypassctl.vhd	This module controls whether a Non-Posted PCI Express request is forwarded to the MegaCore function or held in a bypass FIFO until the MegaCore function has enough credits to accept it. Arbitration is based on the available non-posted header and data credits indicated by the MegaCore function.
altpcierrd_icm_sideband.v or altpcierrd_icm_sideband.vhd	This module implements incremental-compile boundary registers for the non-timing critical sideband signals to and from the MegaCore function.
altpcierrd_icm_fifo.v or altpcierrd_icm_fifo.vhd	This is a MegaWizard-generated RAM-based FIFO.
altpcierrd_icm_fifo_lkhd.v or altpcierrd_icm_fifo_lkhd.vhd	This is a MegaWizard-generated RAM-based look-ahead FIFO.
altpcierrd_icm_defines.v or altpcierrd_icm_defines.vhd	This file contains global <code>define</code> 's used by the Verilog ICM modules.

ICM Application-Side Interface

Tables and timing diagrams in this section describe the following application-side interfaces of the ICM:

- Rx ports
- Tx ports
- MSI port
- Sideband interface

Rx Ports

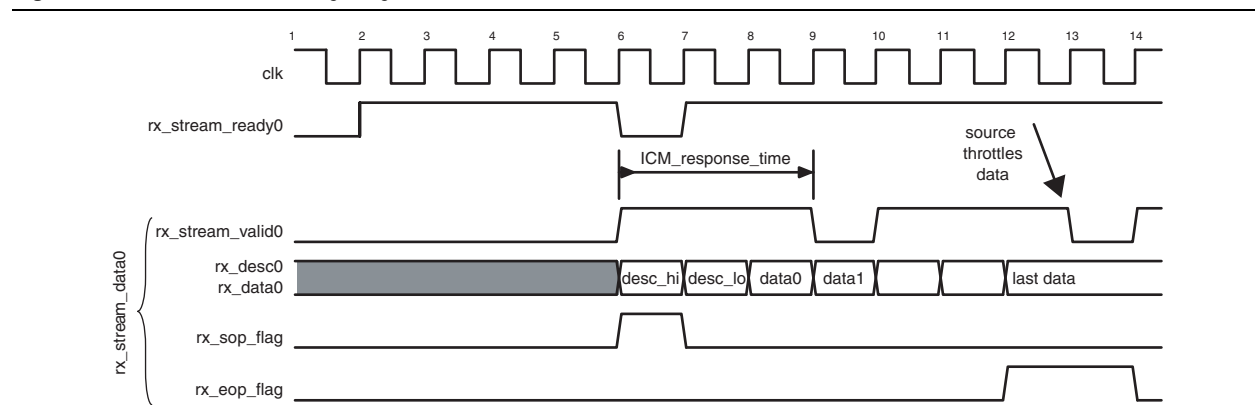
Table C-3 describes the application-side ICM Rx signals.

Table C-3. Application-Side Rx Signals

Signal	Bits	Subsignals	Description
Interface Signals			
rx_st_valid0			Clocks rx_st_data into the application. The application must accept the data when rx_st_valid is high.
rx_st_data0	[81:74]	Byte Enable bits	Byte-enable bits. These are valid on the data (3rd to last) cycles of the packet.
	[73]	rx_sop_flag	When asserted, indicates that this is the first cycle of the packet.
	[72]	rx_eop_flag	When asserted, indicates that this is the last cycle of the packet.
	[71:64]	Bar bits	BAR bits. These are valid on the 2nd cycle of the packet.
	[63:0]	rx_desc/rx_data	Multiplexed rx_desc/rx_data bus 1st cycle – rx_desc0[127:64] 2nd cycle – rx_desc0[63:0] 3rd cycle – rx_data0 (if any) Refer to Table 5-24 on page 5-37 for information on rx_desc0 and rx_data0.
rx_st_ready0			The application asserts this signal to indicate that it can accept more data. The ICM responds 3 cycles later by deasserting rx_st_valid.
Other Rx Interface Signals			
rx_stream_mask0			Application asserts this to tell the MegaCore function to stop sending non-posted requests to the ICM. Note: This does not affect non-posted requests that the MegaCore function already passed to the ICM.

Figure C-3 shows the application-side Rx interface timing diagram.

Figure C-3. Rx Interface Timing Diagram



Tx Ports

Table C-4 describes the application-side Tx signals.

Table C-4. Application-Side Tx Signals

Signal	Bit	Subsignals	Description
Avalon-ST Tx Interface Signals			
tx_st_valid0			Clocks tx_st_data0 into the ICM. The ICM accepts data when tx_st_valid0 is high.
tx_st_data0	63:0	tx_desc/tx_data	Multiplexed tx_desc0/tx_data0 bus. 1st cycle – tx_desc0[127:64] 2nd cycle – tx_desc0[63:0] 3rd cycle – tx_data0 (if any) Refer to for information on Table 5-30 on page 5-48 tx_desc0 and tx_data0.
	71:64		Unused bits
	72	tx_eop_flag	Asserts on the last cycle of the packet
	73	tx_sop_flag	Asserts on the 1st cycle of the packet
	74	tx_err	Same as MegaCore function definition. Refer to Table 5-31 on page 5-50 for more information.
tx_st_ready0			The ICM asserts this signal when it can accept more data. The ICM deasserts this signal to throttle the data. When the ICM deasserts this signal, the user application must also deassert tx_st_valid0 within 3 clk cycles.
Other Tx Interface Signals			
tx_stream_cred0	65:0		Available credits in MegaCore function (credit limit minus credits consumed). This signal corresponds to tx_cred0 from the PCI Express MegaCore function delayed by one system clock cycle. This information can be used by the application to send packets based on available credits. Note that this signal does not account for credits consumed in the ICM. Refer to Table 5-31 on page 5-50 for information on tx_cred0.
tx_stream_mask0			Asserted by ICM to throttle Non-Posted requests from application. When set, application should stop issuing Non-Posted requests in order to prevent head-of-line blocking.

Figure C-4 shows the application-side Tx interface timing diagram.

Figure C-4. Tx Interface Timing Diagram

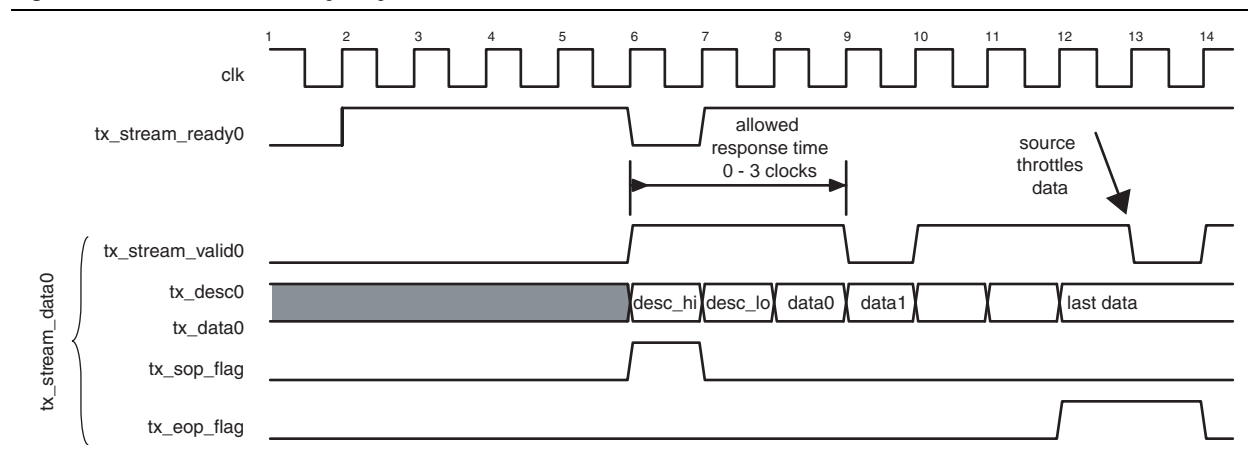


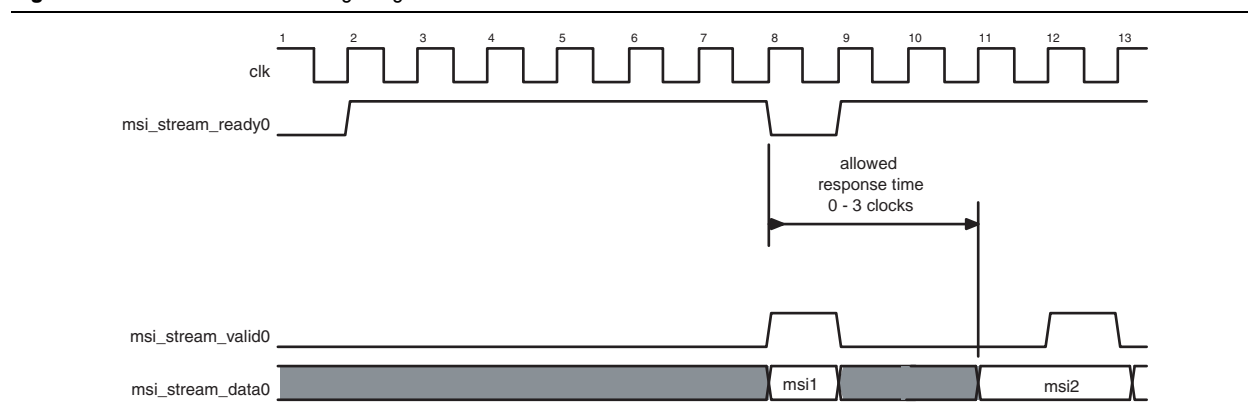
Table C-5 describes the MSI Tx signals.

Table C-5. MSI Tx Signals

Signal	Bit	Subsignals	Description
msi_stream_valid0			Clocks msi_st_data into the ICM.
msi_stream_data0	63:8		msi data.
	7:5		Corresponds to the app_msi_tc signal on the MegaCore function. Refer to Table 5-10 on page 5-20 for more information.
	4:0		Corresponds to the app_msi_num signal on the MegaCore function. Refer to Table 5-10 on page 5-20 for more information.
msi_stream_ready0			The ICM asserts this signal when it can accept more MSI requests. When deasserted, the application must deassert msi_st_valid within 3 CLK cycles.

Figure C-5 shows the application-side MSI interface timing diagram.

Figure C-5. MSI Interface Timing Diagram



Sideband Interface

Table C-6 describes the application-side sideband signals.

Table C-6. Sideband Signals

Signal	Bit	Description
app_int_sts_icm	—	Same as app_int_sts on the MegaCore function interface. ICM delays this signal by one clock. (3)
cfg_busdev_icm	—	Delayed version of cfg_busdev on the MegaCore function interface. (2)
cfg_devcsr_icm	—	Delayed version of cfg_devcsr on the MegaCore function interface. (2)
cfg_linkcsr_icm	—	Delayed version of cfg_linkcsr on MegaCore function interface. ICM delays this signal by one clock. (2)
cfg_tvcmap_icm	—	Delayed version of cfg_tvcmap on MegaCore function interface. (2)
cpl_err_icm	—	Same as cpl_err_icm on MegaCore function interface (1). ICM delays this signal by one clock.
pex_msi_num_icm	—	Same as pex_msi_num on MegaCore function interface (3). ICM delays this signal by one clock.
cpl_pending_icm	—	Same as cpl_pending on MegaCore function interface (1). ICM delays this signal by one clock.
app_int_sts_ack_icm	—	Delayed version of app_int_sts_ack on MegaCore function interface. ICM delays this by one clock. This signal applies to the ×1 and ×4 MegaCore functions only. In ×8, this signal is tied low.
cfg_msicsr_icm	—	Delayed version of cfg_msicsr on the MegaCore function interface. ICM delays this signal by one clock.
test_out_icm	[8:0]	This is a subset of test_out signals from the MegaCore function. Refer to Appendix B for a description of test_out.
	[4:0]	“Itssm_r” debug signal. Delayed version of test_out [4:0] on ×8 MegaCore function interface. Delayed version of test_out [324:320] on ×4/ ×1 MegaCore function interface.
	[8:5]	“lane_act” debug signal. Delayed version of test_out [91:88] on ×8 MegaCore function interface. Delayed version of test_out [411:408] on ×4/ ×1 MegaCore function interface.

Notes to Table C-6:

- (1) Refer to Table 5-34 on page 5-59f for more information.
- (2) Refer to Table 5-18 on page 5-29 for more information.
- (3) Refer to Table 5-10 on page 5-20 for more information.

Revision History

The table below displays the revision history for the chapters in this User Guide.

Date	Version	Changes Made
March 2009	9.0	<ul style="list-style-type: none"> Expanded discussion of “Serial Interface Signals” on page 5–67. Clarified Table 1–20 on page 1–19. All cores support ECC with the exception of Gen2 x8. The internal clock of the x8 core runs at 500 MHz.
March 2009	9.0	<ul style="list-style-type: none"> Added warning about use of <code>test_out</code> and <code>test_in</code> buses. Moved debug signals <code>rx_st_fifo_full0</code> and <code>rx_st_fifo_empty0</code> to the test bus. Documentation for these signals moved from the <i>Signals</i> chapter to Appendix B, Test Port Interface Signals. Added note to Table 4–15 on page 4–42 saying that the PCI Express MegaCore function does not support PME messaging.
February 2009	9.0	<ul style="list-style-type: none"> Updated Table 1–5 on page 1–11 and Table 1–20 on page 1–19. Removed <code>tx_swing</code> signal.
February 2009	9.0	<ul style="list-style-type: none"> Added device support for Arria II GX in both the hard and soft IP implementations. Added preliminary support for HardCopy III and HardCopy IV E. Added support for hard IP endpoints in the SOPC Builder design flow. Added PCI Express reconfiguration block for dynamic reconfiguration of configuration space registers. Updated figures to show this block. Enhanced Chapter 7, Testbench and Design Example to include default instantiation of the RC slave module, tests for ECRC and PCI Express dynamic reconfiguration. Changed Chapter 8, SOPC Builder Design Example to demonstrate use of interrupts. Improved documentation of MSI. Added definitions of DMA read and writes status registers in Chapter 7, Testbench and Design Example. Added the following signals to the hard IP implementation of root port and endpoint using the MegaWizard Plug-In Manager design flow: <code>tx_pipemargin</code>, <code>tx_pipedeeemph</code>, <code>tx_swing</code> (PIPE interface), <code>ltssm[4:0]</code>, and <code>lane_act[3:0]</code> (Test interface). Added recommendation in “Avalon Configuration Settings” on page 3–12 that when the Avalon Configuration selects a dynamic translation table that multiple address translation table entries be employed to avoid updating a table entry before outstanding requests complete. Clarified that ECC support is only available in the hard IP implementation. Updated Figure 4–8 on page 4–10 to show connections between the Type 0 Configuration Space register and all virtual channels.

Date	Version	Changes Made
		<ul style="list-style-type: none"> ■ Made the following corrections to description of Chapter 3, Parameter Settings: <ul style="list-style-type: none"> → The enable rate match FIFO is available for Stratix IV GX → Completion timeout is available for v2.0 → MSI-X Table BAR Indicator (BIR) value can range 1:0–5:0 depending on BAR settings → Changes in “Power Management Parameters” on page 3–10: L0s acceptable latency is $\leq 4 \mu\text{s}$, not $< 4 \mu\text{s}$; L1 acceptable latency is $\leq 64 \mu\text{s}$, not $< 64 \mu\text{s}$, L1 exit latency common clock is $\leq 64 \mu\text{s}$, not $< 64 \mu\text{s}$, L1 exit latency separate clock is $\leq 64 \mu\text{s}$, not $< 64 \mu\text{s}$ → N_FTS controls are disabled for Stratix IV GX pending devices characterization
November 2008	8.1	<ul style="list-style-type: none"> ■ Added new material on root port which is available for the hard IP implementation in Stratix IV GX devices. ■ Changed to full support for Gen2 x8 in the Stratix IV GX device. ■ Added discussion of dynamic reconfiguration of the transceiver for Stratix IV GX devices. Refer to Table 5–40. ■ Updated Resource Usage and Performance numbers for Quartus II 8.1 software ■ Added text explaining where Tx I/Os are constrained. (Chapter 1) ■ Corrected Number of Address Pages in Table 3–6. ■ Revised the Table 4–40 on page 4–57. The following message types Assert_INTB, Assert_INTC, Assert_INTD, Deassert_INTB, Deassert_INTC and Deassert_INTD are not generated by the core. ■ Clarified definition of rx_ack. It cannot be used to backpressure rx_data. ■ Corrected descriptions of cpl_err[4] and cpl_err[5] which were reversed. Added the fact that the cpl_err signals are pulsed for 1 cycle. ■ Corrected 128-bit Rx data layout in Figure 5–7, Figure 5–8, Figure 5–9, Figure 5–10, Figure 5–15, Figure 5–16, and Figure 5–17. ■ Added explanation that for tx_cred port, completion header, posted header, non-posted header and non-posted data fields, a value of 7 indicates 7 or more available credits. ■ Added warning that in the Cyclone III designs using the external PHY must not use the dual-purpose V_{REF} pins. ■ Revised Figure 6–6. For 8.1 txclk goes through a flip flop and is not inverted. ■ Corrected (reversed) positions of the SMI and EPLAST_ENA bits in Table 7–12. ■ Added note that the RC slave module which is by default not instantiated in the Chapter 7, Testbench and Design Example must be instantiated to avoid deadline in designs that interface to a commercial BIOS. ■ Added definitions for test_out in hard IP implementation. ■ Removed description of Training error bit which is not supported in <i>PCI Express Specifications 1.1, 2.0 or 1.0a</i> for endpoints.

Date	Version	Changes Made
May 2008	8.0	<ul style="list-style-type: none"> ■ Added information describing PCI Express hard IP MegaCore function. ■ Moved sections describing signals to separate chapter. ■ Corrected description of <code>cpl_err</code> signal in Table 5–23 on page 5–34 and Table 5–34 on page 5–59 ■ Corrected Figure 8–3 on page 8–8 showing connections for SOPC Builder system. This system no longer requires an interrupt. ■ Improved description of Chapter 7, Testbench and Design Example. Corrected module names and added descriptions of additional modules. ■ Removed descriptions of Type 0 and Type 1 Configuration Read/Write requests because they are not used in the PCI Express endpoint. ■ Added missing signal descriptions for Avalon-ST interface. ■ Completed connections for <code>npwr</code> in Figure 5–20 on page 5–17. ■ Expanded definition of Quartus II .qip file. ■ Added instructions for connecting the calibration clock of the PCI Express Compiler. ■ Updated discussion of clocking for external PHY. ■ Removed simple DMA design example.
October	7.2	<ul style="list-style-type: none"> ■ Added support for Avalon-ST interface in the MegaWizard Plug-In Manager flow. ■ Added single-clock mode in SOPC Builder flow. ■ Re-organized document to put introductory information about the core first and streamline the design examples and moved detailed design example to a separate chapter. ■ Corrected text describing reset for $\times 1$, $\times 4$ and $\times 8$ MegaCore. functions and Figure 5–20 on page 5–17 and Figure 5–21 that illustrate reset. ■ Corrected Timing Diagram: Transaction with a Data Payload.
May 2007	7.1	<ul style="list-style-type: none"> ■ Added support for Arria GX device family. ■ Added SOPC Builder support for $\times 1$ and $\times 4$. ■ Added Incremental Compile Module (ICM).
December 2006	7.0	<ul style="list-style-type: none"> ■ Maintenance release; updated version numbers.
April 2006	2.1.0 rev 2	<ul style="list-style-type: none"> ■ Minor format changes throughout user guide.
May 2007	7.1	<ul style="list-style-type: none"> ■ Added support for Arria GX device family. ■ Added SOPC Builder support for $\times 1$ and $\times 4$. ■ Added Incremental Compile Module (ICM).
December 2006	7.0	<ul style="list-style-type: none"> ■ Added support for Cyclone III device family.
December 2006	6.1	<ul style="list-style-type: none"> ■ Added support Stratix III device family. ■ Updated version and performance information.
April 2006	2.1.0	<ul style="list-style-type: none"> ■ Rearranged content. ■ Updated performance information.
October 2005	2.0.0	<ul style="list-style-type: none"> ■ Added $\times 8$ support. ■ Added device support for Stratix® II GX and Cyclone® II. ■ Updated performance information.

Date	Version	Changes Made
June 2005	1.0.0	■ First release.
May 2007	7.1	■ Added SOPC Builder Design Flow walkthrough. ■ Revised MegaWizard Plug-In Manager Design Flow walkthrough.
December	6.1	■ Updated screen shots and version numbers. ■ Modified text to accommodate new MegaWizard interface. ■ Updated installation diagram. ■ Updated walkthrough to accommodate new MegaWizard interface.
April 2006	2.1.0	■ Updated screen shots and version numbers. ■ Added steps for sourcing Tcl constraint file during compilation to the walkthrough in the section. ■ Moved installation information to release notes.
October 2005	2.0.0	■ Updated screen shots and version numbers.
June 2005	1.0.0	■ First release.
May 2007	7.1	■ Added sections relating to SOPC Builder.
December 2006	6.1	■ Updated screen shots and parameters for new MegaWizard interface. ■ Corrected timing diagrams.
April 2006	2.1.0	■ Added section “Analyzing Throughput” on page 4–27. ■ Updated screen shots and version numbers. ■ Updated System Settings, Capabilities, Buffer Setup, and Power Management Pages and their parameters. ■ Added three waveform diagrams: ■ Transfer for a single write. ■ Transaction layer not ready to accept packet. ■ Transfer with wait state inserted for a single DWORD.
October 2005	2.0.0	■ Updated screen shots and version numbers.
June 2005	1.0.0	■ First release.
May 2007	7.1	■ Made minor edits and corrected formatting.
December 2006	6.1	■ Modified file names to accommodate new project directory structure. ■ Added references for high performance, Chaining DMA Example.
April 2006	2.1.0	■ New chapter Chapter 6, External PHYs added for external PHY support.
May 2007	7.1	■ Added Incremental Compile Module (ICM) section.
December 2006	6.1	■ Added high performance, Chaining DMA Example.
April 2006	2.1.0	■ Updated chapter number to chapter 5. ■ Added section. ■ Added two BFM Read/Write Procedures: ■ <code>ebfm_start_perf_sample</code> Procedure ■ <code>ebfm_disp_perf_sample</code> Procedure
October 2005	2.0.0	■ Updated screen shots and version numbers.
June 2005	1.0.0	■ First release.

Date	Version	Changes Made
April 2006	2.1.0	■ Removed restrictions for x8 ECRC.
June 2005	1.0.0	■ First release.
May 2007	7.1	■ Recovered hidden Content Without Data Payload tables.
October 2005	2.1.0	■ Minor corrections.
June 2005	1.0.0	■ First release.
April	2.1.0	■ Updated ECRC to include ECRC support for x8.
October 2005	1.0.0	■ Updated ECRC noting no support for x8.
June 2005		■ First release.

How to Contact Altera

For the most up-to-date information about Altera® products, see the following table.

Contact	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Altera literature services	Email	literature@altera.com
Non-technical support (General) (Software Licensing)	Email	nacomp@altera.com
	Email	authorization@altera.com






Note:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicates command names, dialog box titles, dialog box options, and other GUI labels. For example, Save As dialog box.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, and software utility names. For example, \qdesigns directory, d: drive, and chiptrip.gdf file.
<i>Italic Type with Initial Capital Letters</i>	Indicates document titles. For example, <i>AN 519: Stratix IV Design Guidelines</i> .
<i>Italic type</i>	Indicates variables. For example, $n + 1$. Variable names are enclosed in angle brackets (< >). For example, <file name> and <project name>. .pot file.
Initial Capital Letters	Indicates keyboard keys and menu names. For example, Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, "Typographic Conventions."

Visual Cue	Meaning
Courier type	<p>Indicates signal, port, register, bit, block, and primitive names. For example, <code>data1</code>, <code>tdi</code>, and <code>input</code>. Active-low signals are denoted by suffix <code>n</code>. For example, <code>resetn</code>.</p> <p>Indicates command line commands and anything that must be typed exactly as it appears. For example, <code>c:\qdesigns\tutorial\chiptrip.gdf</code>.</p> <p>Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword <code>SUBDESIGN</code>), and logic function names (for example, <code>TRI</code>).</p>
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.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
	A warning calls attention to a condition or possible situation that can cause you injury.
	The angled arrow instructs you to press Enter.
	The feet direct you to more information about a particular topic.