

UltraScale+ Devices

Integrated Block for PCI Express v1.3

Product Guide

Vivado Design Suite

PG213 (v1.3) September 22, 2020



Table of Contents

Chapter 1: Introduction.....	5
Features.....	5
IP Facts.....	6
Chapter 2: Overview.....	7
Feature Summary.....	9
Applications.....	11
Unsupported Features.....	11
Licensing and Ordering.....	12
Chapter 3: Product Specification.....	13
Standards Compliance.....	13
Resource Utilization.....	13
Minimum Device Requirements.....	13
Available Integrated Blocks for PCI Express.....	14
GT Locations.....	18
Port Descriptions.....	18
Configuration Space.....	82
Chapter 4: Designing with the Core.....	88
Tandem Configuration.....	88
Clocking.....	122
Resets.....	125
AXI4-Stream Interface Description.....	126
Power Management.....	270
Generating Interrupt Requests.....	273
Receive Message Interface.....	278
Configuration Management Interface.....	281
Enabling Loopback Master on Root Port.....	284
Link Training: 2-Lane, 4-Lane, 8-Lane, and 16-Lane Components.....	284
Lane Reversal.....	286

Chapter 5: Design Flow Steps.....	287
Customizing and Generating the Core.....	287
Constraining the Core.....	312
Simulation.....	314
Synthesis and Implementation.....	316
Chapter 6: Example Design.....	317
Overview of the Example Design.....	317
Generating the Core.....	331
Simulating the Example Design.....	333
Synthesizing and Implementing the Example Design.....	335
Chapter 7: Test Bench.....	336
Root Port Model Test Bench for Endpoint.....	336
Endpoint Model Test Bench for Root Port.....	350
Appendix A: Upgrading.....	353
Migrating from UltraScale to UltraScale+ Devices.....	353
Upgrading in the Vivado Design Suite.....	356
Appendix B: GT Locations.....	358
Virtex UltraScale+ Devices Available GT Quads.....	361
Kintex UltraScale+ Devices Available GT Quads.....	375
Zynq UltraScale+ Devices Available GT Quads.....	378
Appendix C: Debugging.....	393
Finding Help on Xilinx.com.....	393
Hardware Debug.....	394
Appendix D: Using the Xilinx Virtual Cable to Debug.....	398
Overview.....	398
Host PC XVC-Server Application.....	399
Host PC XVC-over-PCIe Driver.....	399
XVC-over-PCIe Enabled FPGA Design.....	400
Using the PCIe-XVC-VSEC Example Design.....	406
Appendix E: Additional Resources and Legal Notices.....	415
Xilinx Resources.....	415
Documentation Navigator and Design Hubs.....	415

References.....	415
Revision History.....	417
Please Read: Important Legal Notices.....	421

Introduction

The UltraScale+™ Devices Integrated Block for PCI Express® (PCIe®) solution IP core is a high-bandwidth, scalable, and reliable serial interconnect building block solution for use with UltraScale+ devices. Xilinx offers two PCIe integrated blocks in the UltraScale+ architecture: the PCIE4 integrated block, and the PCIE4C integrated block. The PCIE4 block, which is found in all UltraScale+ devices, supports the PCIe IP. The Virtex® UltraScale+™ devices with high bandwidth memory (HBM) contain a mixture of the PCIE4 blocks and enhanced PCIE4C blocks. The PCIE4 block supports 1-lane, 2-lane, 4-lane, 8-lane, and 16-lane configurations, including Gen1 (2.5 GT/s), Gen2 (5.0 GT/s) and Gen3 (8 GT/s) speeds. It is compliant with [PCI Express Base Specification, rev3.1](#). The PCIE4C block is functionally equivalent to the PCIE4 block, and additionally supports Gen4 (16 GT/s) speed with 1-lane, 2-lane, 4-lane and 8-lane configuration, compatible to PCI Express Base Specification, Revision 4.0. This solution supports the AXI4-Stream interface for the customer user interface.

Note: The PCI Express Base Specification Revision 4.0 compatibility of the PCIE4C block references interoperability with Gen4 compliant devices at Gen4 speeds, with certain unsupported features. PCIE4C blocks will interoperate with each other at Gen4 speeds. To determine the suitability of the PCIE4C block for a specific application, Xilinx recommends that designers review the unsupported features in [Unsupported PCI Express Base Specification 4.0 Features \(PCIE4C\)](#) while evaluating the PCIE4C block for use at Gen4 speeds.

Features

- Designed to comply with the [PCI Express Base Specification, rev3.1](#).
- PCI Express® Endpoint, Legacy Endpoint, or Root Port Modes.
- x1, x2, x4, x8 or x16 link widths with Gen1, Gen2 and Gen3 link speeds for both PCIE4 and PCIE4C blocks.
- x1, x2, x4 and x8 link widths with Gen4 link speeds for the PCIE4C block.
- AXI4-Stream Interface to customer logic.
- Advanced Error Reporting (AER) and End-to-End CRC (ECRC).
- Block RAM used for Transaction buffering.
- One PCI Express virtual channel, and eight traffic classes.
- Up to 4 physical functions and 252 virtual functions.

- 3 x 64-bit, or 6 x 32-bit Base Address Registers (BARs) that are fully configurable.

For a full list of features, see [Feature Summary](#).

IP Facts

LogiCORE™ IP Facts Table	
Core Specifics	
Supported Device Family ¹	UltraScale+
Supported User Interfaces	AXI4-Stream
Resources	Performance and Resource Utilization page²
Provided with Core	
Design Files	Verilog
Example Design	Verilog
Test Bench	Verilog
Constraints File	XDC
Simulation Model	Verilog
Supported S/W Driver ²	N/A
Tested Design Flows ³	
Design Entry	Vivado Design Suite
Simulation	For supported simulators, see the Xilinx Design Tools: Release Notes Guide .
Synthesis	Vivado Synthesis
Support	
Release Notes and Known Issues	Master Answer Record: 65751
All Vivado IP Change Logs	Master Vivado IP Change Logs: 72775
Xilinx Support web page	

Notes:

1. For a complete list of supported devices, see the Vivado IP catalog.
2. The resource utilization data applies to both the PCIE4 and PCIE4C blocks.
3. For the supported versions of the tools, see the [Xilinx Design Tools: Release Notes Guide](#).

Overview

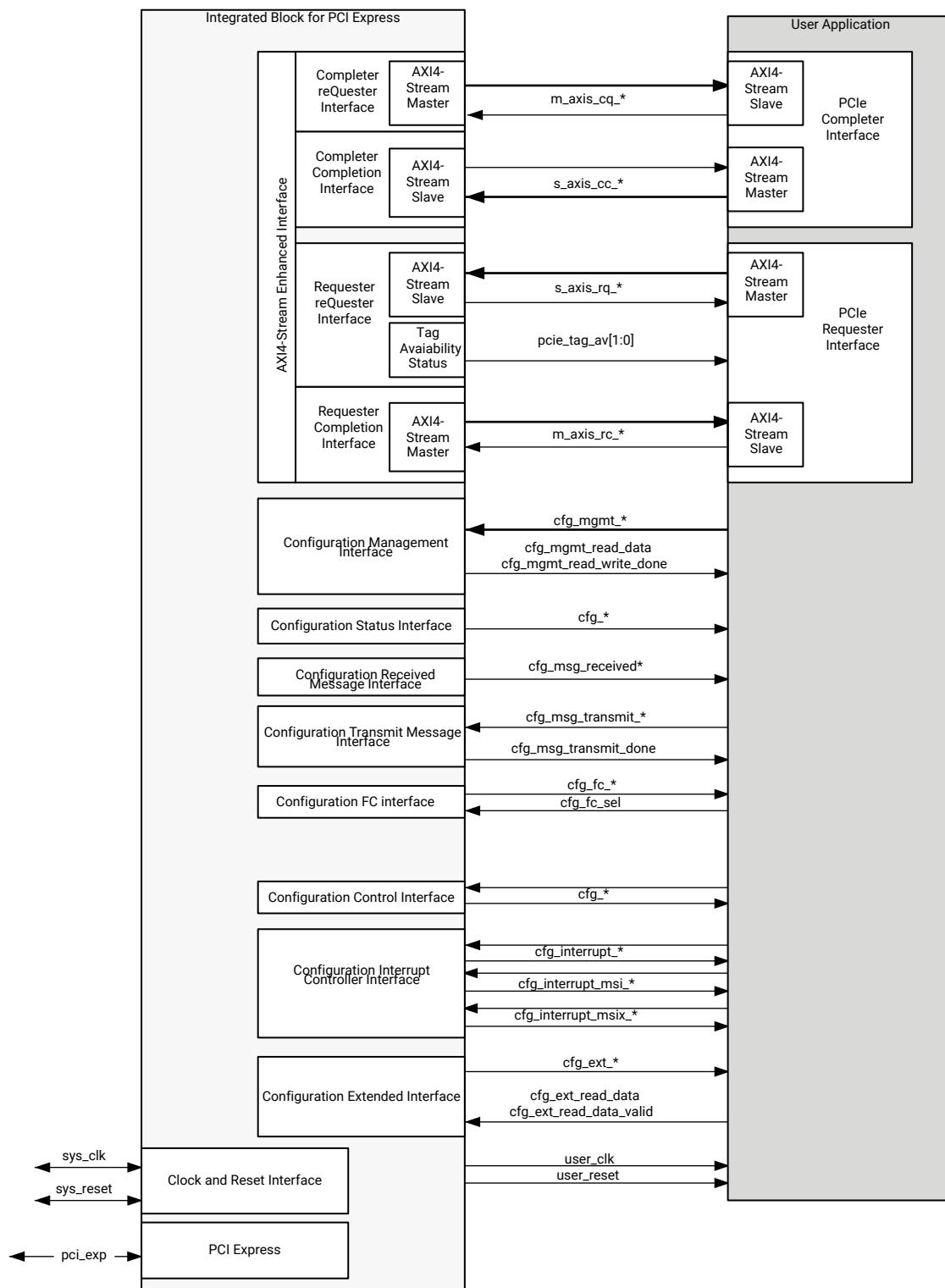
The Integrated Block for PCIe® core is a reliable, high-bandwidth, scalable serial interconnect building block for use with UltraScale+™ devices. The core instantiates the integrated block found in UltraScale+ devices.



IMPORTANT! To implement a design for UltraScale™ devices, refer to the UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide ([PG156](#)) for details.

The following figure shows the interfaces for the core.

Figure 1: Core Interfaces



X16318-030217

Feature Summary

The GTH and GTY transceivers in the PCIE4 and PCIE4C integrated blocks solution support 1-lane, 2-lane, 4-lane, 8-lane, and 16-lane operation, running at 2.5 GT/s (Gen1), 5.0 GT/s (Gen2), and 8.0 GT/s (Gen3) line speeds. Additionally, the PCIE4C integrated block supports 1-lane, 2-lane, 4-lane and 8-lane operation running at 16.0 GT/s (Gen4) line speed. Endpoint and Root Port configurations are supported.

The customer user interface is compliant with the AMBA® AXI4-Stream interface. This interface supports separate Requester, Completion, and Message interfaces. It allows for flexible data alignment and parity checking. Flow control of data is supported in the receive and transmit directions. The transmit direction additionally supports discontinuation of in-progress transactions. Optional back-to-back transactions use straddling to provide greater link bandwidth.

Detailed features of the core are:

- PCIE4 and PCIE4C blocks are compliant with [PCI Express Base Specification, rev3.1](#).
- PCI Express Endpoint, Legacy Endpoint or Root Port Modes
- Both PCIE4 and PCIE4C blocks: x1, x2, x4, x8 or x16 link widths with Gen1, Gen2 and Gen3 link speeds
- PCIE4C block: x1, x2, x4 and x8 link widths with Gen4 link speeds
- AXI4 Streaming Interface to customer logic
 - Configurable 64-bit/128-bit/256-bit/512-bit data path widths
 - Four Independent Request/Completion Streams
- Parity protection on internal logic data paths and data interfaces
- Advanced Error Reporting (AER) and End-to-End CRC (ECRC)
- One PCI Express virtual channel, eight traffic classes
- Supports multiple functions and Single-Root I/O Virtualization
 - Up to 4 physical functions
 - Up to 252 virtual functions
- Built-in lane reversal and receiver lane-to-lane de-skew
- 3 x 64-bit, or 6 x 32-bit Base Address Registers (BARs) that are fully configurable
 - Expansion ROM BAR supported
- Maximum Payload Size: 128, 256, 512, and 1024 bytes

- All Interrupt types are supported:
 - INTx
 - 32 multi-vector MSI capability
 - MSI-X capability with up to 2048 vectors with optional, built-in MSI-X vector tables
- Built-in Initiator Read Request/Completion Tag Manager
 - Up to 256 outstanding Initiator Read Request Transactions supported
- Dynamic Reconfiguration Port (DRP) port supported
- Features that enable high performance applications:
 - AXI4 Streaming Transaction Layer Packets (TLP) Straddle on Requester Completion Interface
 - Up to 256 RX Completion Header Credits and 32 kilobyte RX Completion Payload Space
 - Relaxed Transaction Ordering in the Receive Data Path
 - Address Translation Services (ATS) Messaging
 - Atomic Operation Transactions Support
 - TLP Processing Hints (TPH)
- Several ease of use and configurability features are supported:
 - BAR and ID based filtering of Received Transactions
 - ASPM Optionality
 - Configuration Extend Interface
 - AXI4 Streaming Interfaces Address Align Mode
 - Configuration over PCI Express media configuration access port (MCAP), and 100 ms power on to configuration (Support in the IP core is planned for a future release)
 - Debug and Diagnostics Interface

Related Information

[Unsupported Features](#)

Applications

The core architecture enables a broad range of computing and communications target applications, emphasizing performance, cost, scalability, feature extensibility and mission-critical reliability. Typical applications include:

- Data communications networks
 - Telecommunications networks
 - Broadband wired and wireless applications
 - Network interface cards
 - Chip-to-chip and backplane interface cards
 - Server add-in cards for various applications
-

Unsupported Features

Unsupported PCI Express Base Specification 3.1 Features

The PCI Express Base Specification 3.1 has many optional features. Some of the features which are not supported are listed below:

- Does not implement the Address Translation Service, but allows its implementation in external soft logic.
- Switch ports.
- Resizable BAR extended capability.
- ID-based TLP ordering.
- The PCI Express secure IP model does not support simulation of the DRP interface.
- Tandom PROM simulation is not supported; the simulation model of the hard block does not support it.

Unsupported PCI Express Base Specification 4.0 Features (PCIE4C)

The following PCI Express Base Specification 4.0 Version 1.0 features are not supported by PCIE4C (UltraScale+ HBM):

- Tag Scaling (10b Tag)
- Feature DLLP, Flow Control Scaling
- Retimer Present Bits
- Lane Margining
- Polling Compliance States, and Changes to Compliance Patterns
- Link Extension Devices (Retimers)



IMPORTANT! While the above limitations limit the ability to be 4.0 compliant, interoperation at 16.0 GT/s speeds is possible and will work with the majority of 4.0 devices. Work with the vendor of the connected PCIe device to ensure that these limitations pose no issues.

Licensing and Ordering

The UltraScale+ Devices Integrated Block for PCI Express core is provided at no additional cost with the Vivado Design Suite under the terms of the [Xilinx End User License](#). Information about this and other Xilinx® LogiCORE™ IP modules is available at the [Xilinx Intellectual Property](#) page. For information about pricing and availability of other Xilinx LogiCORE IP modules and tools, contact your local [Xilinx sales representative](#).

Product Specification

Standards Compliance

The UltraScale+ Devices Integrated Block for PCI Express solution is compatible with industry-standard application form factors such as the PCI Express® Card Electromechanical (CEM) v3.0 and the PCIe® Industrial Computer Manufacturers Group (PICMG) 3.4 specifications.

Resource Utilization

For full details about performance and resource use, visit the [Performance and Resource Use web page](#).

Note: The resource utilization data at the above link applies to both the PCIE4 and PCIE4C blocks.

Minimum Device Requirements

Xilinx® offers two PCIe® integrated blocks in the UltraScale+™ architecture. The PCIE4 block is found in all UltraScale+ devices that support the PCIe IP. Virtex UltraScale+ devices with high bandwidth memory (HBM) contain a mixture of the PCIE4 blocks as well as enhanced PCIE4C blocks. The PCIE4C integrated blocks are functionally equivalent to PCIE4 and also support PCIe Gen3 x16 in the -2LV speed grade.

The minimum device requirements for the core are as follows:

Table 1: Minimum Device Requirements for PCIE4

Capability Link Speed	Capability Link Widths	Supported Speed Grades
Gen1/Gen2	x16	-1, -1L, -1LV, -2, -2L, -2LV, -3

Table 1: Minimum Device Requirements for PCIE4 (cont'd)

Capability Link Speed	Capability Link Widths	Supported Speed Grades
Gen3	x16	-1, -1L, -2, -2L, -3
	x8 NL ¹	-1, -1L, -1LV, -2, -2L, -2LV, -3
	x8 LL ²	-1, -1L, -2, -2L, -3

Notes:

1. x8 NL = Gen3 x8 Normal Latency (core_clock = 250 MHz).
2. x8 LL = Gen3 x8 Low Latency (core_clock = 500 MHz).

Table 2: Minimum Device Requirements for PCIE4C

Capability Link Speed	Capability Link Widths	Supported Speed Grades
Gen1/Gen2	x16	-1, -2, -2L, -2LV, -3
Gen3	x16	-1, -2, -2L, -2LV, -3
	x8 NL	-1, -2, -2L, -2LV, -3
	x8 LL	-1, -2, -2L, -3
Gen4	x8	-2, -2L, -3

Available Integrated Blocks for PCI Express

The following tables list the integrated blocks for PCI Express® available for use in devices containing multiple integrated blocks. In some cases, not all integrated blocks can be used due to lack of bonded GTH and GTY transceiver sites adjacent to the integrated block.

Table 3: Available Integrated Blocks for PCI Express - Virtex® UltraScale+™

Device Selection		PCI Express Block Location									
Device	Package	X0Y0	X0Y1	X0Y2	X0Y3	X0Y5	X1Y0	X1Y1	X1Y2	X1Y4	
XCVU3P	FFVC1517		Yes				Yes				
XCVU5P	FLVA2104		Yes		Yes		Yes		Yes		
	FLVB2104		Yes		Yes		Yes		Yes		
	FLVC2104		Yes		Yes		Yes		Yes		
XCVU7P	FLVA2104		Yes		Yes		Yes		Yes		
	FLVB2104		Yes		Yes		Yes		Yes		
	FLVC2104		Yes		Yes		Yes		Yes		
XCVU9P	FLGA2104		Yes		Yes				Yes	Yes	
	FLGB2104		Yes		Yes				Yes	Yes	
	FLGC2104		Yes		Yes	Yes	Yes		Yes	Yes	
	FLGA2577		Yes		Yes	Yes	Yes		Yes	Yes	

Table 3: Available Integrated Blocks for PCI Express - Virtex® UltraScale+™ (cont'd)

Device Selection		PCI Express Block Location								
Device	Package	X0Y0	X0Y1	X0Y2	X0Y3	X0Y5	X1Y0	X1Y1	X1Y2	X1Y4
XQVU7P	FLRA2104		Yes		Yes		Yes		Yes	
	FLRB2104		Yes		Yes		Yes		Yes	
XCVU11P	FLGA2577	Yes	Yes	Yes						
	FLGB2104	Yes	Yes	Yes						
	FLGC2104	Yes	Yes	Yes						
	FLGF1924	Yes	Yes	Yes						
XQVU11P	FLRC2104	Yes	Yes	Yes						
XCVU13P	FHGA2104		Yes	Yes						
	FHGB2014		Yes	Yes	Yes					
	FHGC2104	Yes	Yes	Yes	Yes					
	FLGA2577	Yes	Yes	Yes	Yes					
	FIGD2104		Yes	Yes	Yes					
	FSGA2577	Yes	Yes	Yes	Yes					
XCVU29P	FIGD2104	Yes								
	FSGA2577	Yes								
XCVU31P ¹	FSVH1924	Yes ³	Yes ³				Yes ³	Yes ³		
XCVU33P ¹	FSVH2104	Yes ³	Yes ³				Yes ³	Yes ³		
XCVU35P ²	FSVH2892	Yes ^{3,4}	Yes ³				Yes ³	Yes ³		
	FSVH2104	Yes ^{3,4}	Yes ³				Yes ³	Yes ³		
XCVU37P ²	FSVH2892	Yes ^{3,4}	Yes ^{3,4}				Yes ³	Yes ³		

Notes:

1. These devices contain only PCIE4C blocks.
2. These devices contain both PCIE4 and PCIE4C blocks.
3. PCIE4C block location.
4. PCIE4 block location.

Table 4: Available Integrated Blocks for PCI Express - Zynq® UltraScale+™

Device Selection		PCI Express Block Location						
Device	Package	X0Y0	X0Y1	X0Y2	X0Y3	X1Y0	X1Y1	X1Y2
XCZU11EG	FFVC1760			Yes	Yes	Yes	Yes	
	FFVB1517					Yes	Yes	
	FFVC1156					Yes	Yes	
	FFVF1517					Yes	Yes	
XCZU17EG	FFVC1760			Yes	Yes	Yes	Yes	Yes
	FFVE1924					Yes	Yes	Yes
	FFVB1517					Yes	Yes	
	FFVD1760			Yes	Yes	Yes	Yes	Yes

Table 4: Available Integrated Blocks for PCI Express - Zynq[®] UltraScale+™ (cont'd)

Device Selection		PCI Express Block Location						
Device	Package	X0Y0	X0Y1	X0Y2	X0Y3	X1Y0	X1Y1	X1Y2
XCZU19EG	FFVC1760			Yes	Yes	Yes	Yes	Yes
	FFVE1924					Yes	Yes	Yes
	FFVB1517					Yes	Yes	Yes
	FFVD1760			Yes	Yes	Yes	Yes	Yes
XQZU19EG	FFRB1517					Yes	Yes	Yes
	FFRC1760			Yes	Yes	Yes	Yes	Yes
XCZU4EV	FBVB900	Yes	Yes					
	SFVC784	Yes	Yes					
XCZU5EV	FBVB900	Yes	Yes					
	SFVC784	Yes	Yes					
XCZU7EV	FBVB900	Yes	Yes					
	FFVC1156	Yes	Yes					
	FFVF1517	Yes	Yes					
XCZU4CG	FBVB900	Yes	Yes					
	SFVC784	Yes	Yes					
XCZU5CG	FBVB900	Yes	Yes					
	SFVC784	Yes	Yes					
XCZU7CG	FBVB900	Yes	Yes					
	FFVC1156	Yes	Yes					
	FFVF1517	Yes	Yes					
XCZU4EG	FBVB900	Yes	Yes					
	SFVC784	Yes	Yes					
XCZU5EG	FBVB900	Yes	Yes					
	SFVC784	Yes	Yes					
XCZU7EG	FBVB900	Yes	Yes					
	FFVC1156	Yes	Yes					
	FFVF1517	Yes	Yes					
XCZU21DR	FFVD1156	Yes	Yes					
XCZU25DR	FFVE1156	Yes						
	FFVG1517	Yes						
	FSVE1156	Yes						
	FSVG1517	Yes						
XCZU28DR	FFVE1156	Yes	Yes					
	FFVG1517	Yes	Yes					
	FSVE1156	Yes	Yes					
	FSVG1517	Yes	Yes					
XCZU29DR	FFVF1760	Yes	Yes					
	FSVF1760	Yes	Yes					

Table 4: Available Integrated Blocks for PCI Express - Zynq® UltraScale+™ (cont'd)

Device Selection		PCI Express Block Location						
Device	Package	X0Y0	X0Y1	X0Y2	X0Y3	X1Y0	X1Y1	X1Y2
XCZU27DR	FFVE1156	Yes	Yes					
	FFVG1517	Yes	Yes					
	FSVE1156	Yes	Yes					
	FSVG1517	Yes	Yes					
XAZU4EV	SFVC784	Yes	Yes					
XAZU5EV	SFVC784	Yes	Yes					
XQZU5EV	FFRB900	Yes	Yes					
	SFRC784	Yes	Yes					
XQZU7EV	FFRB900	Yes	Yes					
	FFRC1156	Yes	Yes					
XQZU21DR	FFRD1156	Yes	Yes					
XQZU28DR	FFRE1156	Yes	Yes					

Table 5: Available Integrated Blocks for PCI Express - Kintex® UltraScale+™

Device Selection		PCI Express Block Location						
Device	Package	X0Y0	X0Y2	X0Y3	X1Y0	X1Y1	X1Y2	
XCKU11P	FFVE1517		Yes	Yes	Yes	Yes		
	FFVA1156		Yes	Yes	Yes	Yes		
	FFVD900				Yes	Yes		
XCKU15P	FFVE1517		Yes	Yes	Yes	Yes	Yes	
	FFVA1156		Yes	Yes	Yes	Yes	Yes	
	FFVA1760		Yes	Yes	Yes	Yes	Yes	
	FFVE1760		Yes	Yes	Yes	Yes	Yes	
XCKU3P	FFVA676	Yes						
	FFVB676	Yes						
	FFVD900	Yes						
	SFVB784	Yes						
XCKU5P	FFVA676	Yes						
	FFVB676	Yes						
	FFVD900	Yes						
	SFVB784	Yes						

GT Locations

[Appendix B: GT Locations](#) provides a list of GTs locations available for this IP core and lists some key recommendations that should be considered when selecting the GT location.

Port Descriptions

This section provides detailed port descriptions for the following interfaces:

- AXI4-Stream Core Interfaces
- Other Core Interfaces

AXI4-Stream Core Interfaces

64/128/256-Bit Interfaces

In addition to status and control interfaces, the core has four required AXI4-Stream interfaces used to transfer and receive transactions, which are described in this section.

Related Information

[512-bit Interfaces](#)

Completer Request Interface

The Completer Request (CQ) interface are the ports through which all received requests from the link are delivered to the user application. The following table defines the ports in the CQ interface of the core. In the Width column, DW denotes the configured data bus width (64, 128, or 256 bits).

Table 6: Completer Request Interface Port Descriptions

Port	Direction	Width	Description
m_axis_cq_tdata	Output	DW	Transmit Data from the CQ Interface. Only the lower 128 bits are used when the interface width is 128 bits, and only the lower 64 bits are used when the interface width is 64 bits. Bits [255:128] are set permanently to 0 by the core when the interface width is configured as 128 bits, and bits [255:64] are set permanently to 0 when the interface width is configured as 64 bits.

Table 6: Completer Request Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
m_axis_cq_tuser	Output	88	<p>CQ User Data.</p> <p>This set of signals contains sideband information for the transaction layer packets (TLP) being transferred. These signals are valid when m_axis_cq_tvalid is High.</p> <p>Table 7: Sideband Signal Descriptions in m_axis_cq_tuser describes the individual signals in this set.</p>
m_axis_cq_tlast	Output	1	<p>TLAST indication for CQ Data.</p> <p>The core asserts this signal in the last beat of a packet to indicate the end of the packet. When a TLP is transferred in a single beat, the core sets this signal in the first beat of the transfer.</p>
m_axis_cq_tkeep	Output	DW/32	<p>TKEEP indication for CQ Data.</p> <p>The assertion of bit <i>i</i> of this bus during a transfer indicates to the user application that Dword <i>i</i> of the m_axis_cq_tdata bus contains valid data. The core sets this bit to 1 contiguously for all Dwords starting from the first Dword of the descriptor to the last Dword of the payload. Thus, m_axis_cq_tdata is set to all 1s in all beats of a packet, except in the final beat when the total size of the packet is not a multiple of the width of the data bus (in both Dwells). This is true for both Dword-aligned and address-aligned modes of payload transfer.</p> <p>Bits [7:4] of this bus are set permanently to 0 by the core when the interface width is configured as 128 bits, and bits [7:2] are set permanently to 0 when the interface width is configured as 64 bits.</p>
m_axis_cq_tvalid	Output	1	<p>CQ Data Valid.</p> <p>The core asserts this output whenever it is driving valid data on the m_axis_cq_tdata bus. The core keeps the valid signal asserted during the transfer of a packet. The user application can pace the data transfer using the m_axis_cq_tready signal.</p>
m_axis_cq_tready	Input	1	<p>CQ Data Ready.</p> <p>Activation of this signal by the user logic indicates to the core that the user application is ready to accept data. Data is transferred across the interface when both m_axis_cq_tvalid and m_axis_cq_tready are asserted in the same cycle.</p> <p>If the user application deasserts the ready signal when m_axis_cq_tvalid is High, the core maintains the data on the bus and keeps the valid signal asserted until the user application has asserted the ready signal.</p>

Table 6: Completer Request Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
pcie_cq_np_req	Input	2	<p>This input is used by the user application to request the delivery of a Non-Posted request. The core implements a credit-based flow control mechanism to control the delivery of Non-Posted requests across the interface, without blocking Posted TLPs.</p> <p>This input to the core controls an internal credit count. The credit count is updated in each clock cycle based on the setting of pcie_cq_np_req[1:0] as follows:</p> <ul style="list-style-type: none"> • 00: No change • 01: Increment by 1 • 10 or 11: Reserved (bit [1] only applicable in 512-bit interface) <p>The credit count is decremented on the delivery of each Non-Posted request across the interface. The core temporarily stops delivering Non-Posted requests to the user logic when the credit count is zero. It continues to deliver any Posted TLPs received from the link even when the delivery of Non-Posted requests has been paused.</p> <p>The user application can either set pcie_cq_np_req[1:0] in each cycle based on the status of its Non-Posted request receive buffer, or can set it to 11 permanently if it does not need to exercise selective backpressure on Non-Posted requests.</p> <p>The setting of pcie_cq_np_req[1:0] does not need to be aligned with the packet transfers on the completer request interface.</p>
pcie_cq_np_req_count	Output	6	<p>This output provides the current value of the credit count maintained by the core for delivery of Non-Posted requests to the user logic. The core delivers a Non-Posted request across the completer request interface only when this credit count is non-zero. This counter saturates at a maximum limit of 32.</p> <p>Because of internal pipeline delays, there can be several cycles of delay between the user application providing credit on the pcie_cq_np_req[1:0] inputs and the PCIe core updating the pcie_cq_np_req_count output in response.</p> <p>This count resets on user_reset and de-assertion of user_lnk_up.</p>

Table 7: Sideband Signal Descriptions in m_axis_cq_tuser

Bit Index	Name	Width	Description
3:0	first_be[3:0]	4	<p>Byte enables for the first Dword of the payload.</p> <p>This field reflects the setting of the First_BE bits in the Transaction-Layer header of the TLP. For Memory Reads and I/O Reads, these four bits indicate the valid bytes to be read in the first Dword. For Memory Writes and I/O Writes, these bits indicate the valid bytes in the first Dword of the payload. For Atomic Operations and Messages with a payload, these bits are set to all 1s.</p> <p>This field is valid in the first beat of a packet, that is, when sop and m_axis_cq_tvalid are both High.</p>

Table 7: Sideband Signal Descriptions in m_axis_cq_tuser (cont'd)

Bit Index	Name	Width	Description
7:4	last_be[3:0]	4	<p>Byte enables for the last Dword.</p> <p>This field reflects the setting of the Last_BE bits in the Transaction-Layer header of the TLP. For Memory Reads, these four bits indicate the valid bytes to be read in the last Dword of the block of data. For Memory Writes, these bits indicate the valid bytes in the ending Dword of the payload. For Atomic Operations and Messages with a payload, these bits are set to all 1s. For Memory Reads and Writes of one DW transfers and zero length transfers, these bits should be 0s.</p> <p>This field is valid in the first beat of a packet, that is, when sop and m_axis_cq_tvalid are both High.</p>
39:8	byte_en[31:0]	32	<p>The user logic can optionally use these byte enable bits to determine the valid bytes in the payload of a packet being transferred. The assertion of bit <i>i</i> of this bus during a transfer indicates that byte <i>i</i> of the m_axis_cq_tdata bus contains a valid payload byte. This bit is not asserted for descriptor bytes.</p> <p>Although the byte enables can be generated by user logic from information in the request descriptor (address and length) as well as the settings of the first_be and last_be signals, you can use these signals directly instead of generating them from other interface signals.</p> <p>When the payload size is more than two Dwords (eight bytes), the one bit on this bus for the payload is always contiguous. When the payload size is two Dwords or less, the one bit can be non-contiguous.</p> <p>For the special case of a zero-length memory write transaction defined by the PCI Express specifications, the byte_en bits are all 0s when the associated one-DW payload is being transferred.</p> <p>Bits [31:16] of this bus are set permanently to 0 by the core when the interface width is configured as 128 bits, and bits [31:8] are set permanently to 0 when the interface width is configured as 64 bits.</p>
40	sop	1	<p>Start of packet.</p> <p>This signal is asserted by the core in the first beat of a packet to indicate the start of the packet. Using this signal is optional.</p>
41	discontinue	1	<p>This signal is asserted by the core in the last beat of a TLP, if it has detected an uncorrectable error while reading the TLP payload from its internal FIFO memory. The user application must discard the entire TLP when such an error is signaled by the core.</p> <p>This signal is never asserted when the TLP has no payload. It is asserted only in a cycle when m_axis_cq_tlast is High.</p> <p>When the core is configured as an Endpoint, the error is also reported by the core to the Root Complex to which it is attached, using Advanced Error Reporting (AER).</p>
42	tph_present	1	<p>This bit indicates the presence of a Transaction Processing Hint (TPH) in the request TLP being delivered across the interface. This bit is valid when sop and m_axis_cq_tvalid are both High.</p>
44:43	tph_type[1:0]	2	<p>When a TPH is present in the request TLP, these two bits provide the value of the PH[1:0] field associated with the hint. These bits are valid when sop and m_axis_cq_tvalid are both High.</p>

Table 7: Sideband Signal Descriptions in m_axis_cq_tuser (cont'd)

Bit Index	Name	Width	Description
52:45	tph_st_tag[7:0]	8	When a TPH is present in the request TLP, this output provides the 8-bit Steering Tag associated with the hint. These bits are valid when sop and m_axis_cq_tvalid are both High.
84:53	parity	32	Bit i provides the odd parity computed for byte i of m_axis_cq_tdata. Only the lower 16 bits are used when the interface width is 128 bits, and only the lower 8 bits are used when the interface width is 64 bits. Bits [31:16] are set permanently to 0 by the core when the interface width is configured as 128 bits, and bits [31:8] are set permanently to 0 when the interface width is configured as 64 bits.
87:85	Reserved	3	Reserved for 512-bit interfaces.

Completer Completion Interface

The Completer Completion (CC) interface are the ports through which completions generated by the user application responses to the completer requests are transmitted. You can process all Non-Posted transactions as split transactions. That is, the CC interface can continue to accept new requests on the requester completion interface while sending a completion for a request. The following table defines the ports in the CC interface of the core. In the Width column, DW denotes the configured data bus width (64, 128, or 256 bits).

Table 8: Completer Completion Interface Port Descriptions

Port	Direction	Width	Description
s_axis_cc_tdata	Input	DW	Completer Completion Data bus. Completion data from the user application to the core. Only the lower 128 bits are used when the interface width is 128 bits, and only the lower 64 bits are used when the interface width is 64 bits.
s_axis_cc_tuser	Input	33	Completer Completion User Data. This set of signals contain sideband information for the TLP being transferred. These signals are valid when s_axis_cc_tvalid is High. The following tables describe the individual signals in this set.
s_axis_cc_tlast	Input	1	TLAST indication for Completer Completion Data. The user application must assert this signal in the last cycle of a packet to indicate the end of the packet. When the TLP is transferred in a single beat, the user application must set this bit in the first cycle of the transfer.

Table 8: Completer Completion Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
s_axis_cc_tkeep	Input	DW/32	TKEEP indication for Completer Completion Data. The assertion of bit i of this bus during a transfer indicates to the core that Dword i of the s_axis_cc_tdata bus contains valid data. Set this bit to 1 contiguously for all Dwords starting from the first Dword of the descriptor to the last Dword of the payload. Thus, s_axis_cc_tdata must be set to all 1s in all beats of a packet, except in the final beat when the total size of the packet is not a multiple of the width of the data bus (both in Dwords). This is true for both Dword-aligned and address-aligned modes of payload transfer. Bits [7:4] of this bus are not used by the core when the interface width is configured as 128 bits, and bits [7:2] are not used when the interface width is configured as 64 bits.
s_axis_cc_tvalid	Input	1	Completer Completion Data Valid. The user application must assert this output whenever it is driving valid data on the s_axis_cc_tdata bus. The user application must keep the valid signal asserted during the transfer of a packet. The core paces the data transfer using the s_axis_cc_tready signal.
s_axis_cc_tready	Output	4	Completer Completion Data Ready. Activation of this signal by the core indicates that it is ready to accept data. Data is transferred across the interface when both s_axis_cc_tvalid and s_axis_cc_tready are asserted in the same cycle. If the core deasserts the ready signal when the valid signal is High, the user application must maintain the data on the bus and keep the valid signal asserted until the core has asserted the ready signal.

Table 9: Sideband Signal Descriptions in s_axis_cc_tuser

Bit Index	Name	Width	Description
0	discontinue	1	<p>This signal can be asserted by the user application during a transfer if it has detected an error (such as an uncorrectable ECC error while reading the payload from memory) in the data being transferred and needs to abort the packet. The core nullifies the corresponding TLP on the link to avoid data corruption.</p> <p>The user application can assert this signal during any cycle during the transfer. It can either choose to terminate the packet prematurely in the cycle where the error was signaled, or can continue until all bytes of the payload are delivered to the core. In the latter case, the core treats the error as sticky for the following beats of the packet, even if the user application deasserts the discontinue signal before the end of the packet.</p> <p>The discontinue signal can be asserted only when s_axis_cc_tvalid is High. The core samples this signal only when s_axis_cc_tready is High. Thus, when asserted, it should not be deasserted until s_axis_cc_tready is High.</p> <p>When the core is configured as an Endpoint, this error is also reported by the core to the Root Complex to which it is attached, using AER.</p>

Table 9: Sideband Signal Descriptions in s_axis_cc_tuser (cont'd)

Bit Index	Name	Width	Description
32:1	Parity	32	<p>Odd parity for the 256-bit data.</p> <p>When parity checking is enabled in the core, user logic must set bit i of this bus to the odd parity computed for byte i of s_axis_cc_tdata. Only the lower 16 bits are used when the interface width is 128 bits, and only the lower 8 bits are used when the interface width is 64 bits.</p> <p>When an interface parity error is detected, it is recorded as an uncorrectable internal error and the packet is discarded. According to the Base Spec 6.2.9, an uncorrectable internal error is an error that occurs within a component that results in improper operation of the component. The only method of recovering from an uncorrectable internal error is a reset or hardware replacement.</p> <p>The parity bits can be permanently tied to 0 if parity check is not enabled in the core.</p>

Requester Request Interface

The Requester Request (RQ) interface consists of the ports through which the user application generates requests to remote PCIe® devices. The following table defines the ports in the RQ interface of the core. In the Width column, DW denotes the configured data bus width (64, 128, or 256 bits).

Table 10: Requester Request Interface Port Descriptions

Port	Direction	Width	Description
s_axis_rq_tdata	Input	DW	<p>Requester reQuest Data bus.</p> <p>This input contains the requester-side request data from the user application to the core. Only the lower 128 bits are used when the interface width is 128 bits, and only the lower 64 bits are used when the interface width is 64 bits.</p>
s_axis_rq_tuser	Input	62	<p>Requester reQuest User Data.</p> <p>This set of signals contains sideband information for the TLP being transferred. These signals are valid when s_axis_rq_tvalid is High.</p> <p>The following tables describes the individual signals in this set.</p>
s_axis_rq_tlast	Input	1	<p>TLAST Indication for Requester reQuest Data.</p> <p>The user application must assert this signal in the last cycle of a TLP to indicate the end of the packet. When the TLP is transferred in a single beat, the user application must set this bit in the first cycle of the transfer.</p>

Table 10: Requester Request Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
s_axis_rq_tkeep	Input	DW/32	<p>TKEEP Indication for Requester reQuest Data. The assertion of bit i of this bus during a transfer indicates to the core that Dword i of the s_axis_rq_tdata bus contains valid data. The user application must set this bit to 1 contiguously for all Dwords, starting from the first Dword of the descriptor to the last Dword of the payload. Thus, s_axis_rq_tkeep must be set to all 1s in all beats of a packet, except in the final beat when the total size of the packet is not a multiple of the width of the data bus (in both Dwords). This is true for both Dword-aligned and address-aligned modes of payload transfer.</p> <p>Bits [7:4] of this bus are not used by the core when the interface width is configured as 128 bits, and bits [7:2] are not used when the interface width is configured as 64 bits.</p>
s_axis_rq_tvalid	Input	1	<p>Requester reQuest Data Valid. The user application must assert this output whenever it is driving valid data on the s_axis_rq_tdata bus. The user application must keep the valid signal asserted during the transfer of a packet. The core paces the data transfer using the s_axis_rq_tready signal.</p>
s_axis_rq_tready	Output	4	<p>Requester reQuest Data Ready. Activation of this signal by the core indicates that it is ready to accept data. Data is transferred across the interface when both s_axis_rq_tvalid and s_axis_rq_tready are asserted in the same cycle.</p> <p>If the core deasserts the ready signal when the valid signal is High, the user application must maintain the data on the bus and keep the valid signal asserted until the core has asserted the ready signal.</p> <p>You can assign all 4 bits to 1 or 0.</p>
pcie_rq_seq_num0	Output	6	<p>Requester reQuest TLP transmit sequence number. You can optionally use this output to track the progress of the request in the core transmit pipeline. To use this feature, provide a sequence number for each request on the seq_num[3:0] bus. The core outputs this sequence number on the pcie_rq_seq_num0[3:0] output when the request TLP has reached a point in the pipeline where a Completion TLP from the user application cannot pass it. This mechanism enables you to maintain ordering between Completions sent to the CC interface of the core and Posted requests sent to the requester request interface. Data on the pcie_rq_seq_num0[3:0] output is valid when pcie_rq_seq_num_vld0 is High.</p>
pcie_rq_seq_num_vld0	Output	1	<p>Requester reQuest TLP transmit sequence number valid. This output is asserted by the core for one cycle when it has placed valid data on pcie_rq_seq_num0[3:0].</p>

Table 10: Requester Request Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
pcie_rq_tag0 and pcie_rq_tag1	Output	8	<p>Requester reQuest Non-Posted tag.</p> <p>When tag management for Non-Posted requests is performed by the core (AXISTEN_IF_ENABLE_CLIENT_TAG is 0), this output is used by the core to communicate the allocated tag for each Non-Posted request received.</p> <p>The tag value on this bus is valid for one cycle when pcie_rq_tag_vld0 is High. You must copy this tag and use it to associate the completion data with the pending request.</p> <p>There can be a delay of several cycles between the transfer of the request on the s_axis_rq_tdata bus and the assertion of pcie_rq_tag_vld0 by the core to provide the allocated tag for the request. Meanwhile, the user application can continue to send new requests. The tags for requests are communicated on this bus in FIFO order, so the user application can easily associate the tag value with the request it transferred.</p>
pcie_rq_tag_vld0 and pcie_rq_tag_vld1	Output	1	<p>Requester reQuest Non-Posted tag valid.</p> <p>The core asserts this output for one cycle when it has allocated a tag to an incoming Non-Posted request from the requester request interface and placed it on the pcie_rq_tag0 output.</p>

Table 11: Sideband Signal Descriptions in s_axis_rq_tuser

Bit Index	Name	Width	Description
3:0	first_be[3:0]	4	<p>Byte enables for the first Dword.</p> <p>This field must be set based on the desired value of the First_BE bits in the Transaction-Layer header of the request TLP. For Memory Reads, I/O Reads, and Configuration Reads, these four bits indicate the valid bytes to be read in the first Dword. For Memory Writes, I/O Writes, and Configuration Writes, these bits indicate the valid bytes in the first Dword of the payload.</p> <p>The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High.</p>
7:4	last_be[3:0]	4	<p>Byte enables for the last Dword.</p> <p>This field must be set based on the desired value of the Last_BE bits in the Transaction-Layer header of the TLP. For Memory Reads of two Dwords or more, these four bits indicate the valid bytes to be read in the last Dword of the block of data. For Memory Reads and Writes of one DW transfers and zero length transfers, these bits should be 0s. For Memory Writes of two Dwords or more, these bits indicate the valid bytes in the last Dword of the payload.</p> <p>The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High.</p>

Table 11: Sideband Signal Descriptions in s_axis_rq_tuser (cont'd)

Bit Index	Name	Width	Description
10:8	addr_offset[2:0]	3	<p>When the address-aligned mode is in use on this interface, the user application must provide the byte lane number where the payload data begins on the data bus, modulo 4, on this sideband bus. This enables the core to determine the alignment of the data block being transferred.</p> <p>The core samples this field in the first beat of a packet, when <code>s_axis_rq_tvalid</code> and <code>s_axis_rq_tready</code> are both High.</p> <p>When the requester request interface is configured in the Dword-alignment mode, this field must always be set to 0.</p> <p>In Root Port configuration, Configuration Packets must always be aligned to DWO, and therefore for this type of packets, this field must be set to 0 in both alignment modes.</p>
11	discontinue	1	<p>This signal can be asserted by the user application during a transfer if it has detected an error in the data being transferred and needs to abort the packet. The core nullifies the corresponding TLP on the link to avoid data corruption.</p> <p>You can assert this signal in any cycle during the transfer. You can either choose to terminate the packet prematurely in the cycle where the error was signaled, or continue until all bytes of the payload are delivered to the core. In the latter case, the core treats the error as sticky for the following beats of the packet, even if the user application deasserts the discontinue signal before the end of the packet.</p> <p>The discontinue signal can be asserted only when <code>s_axis_rq_tvalid</code> is High. The core samples this signal only when <code>s_axis_rq_tready</code> is High. Thus, when asserted, it should not be deasserted until <code>s_axis_rq_tready</code> is High.</p> <p>Discontinue is not supported for Non-Posted TLPs. The user logic can assert this signal in any cycle except the first cycle during the transfer.</p> <p>When the core is configured as an Endpoint, this error is also reported by the core to the Root Complex to which it is attached, using Advanced Error Reporting (AER).</p>
12	tph_present	1	<p>This bit indicates the presence of a Transaction Processing Hint (TPH) in the request TLP being delivered across the interface. The core samples this field in the first beat of a packet, when <code>s_axis_rq_tvalid</code> and <code>s_axis_rq_tready</code> are both High.</p> <p>This bit must be permanently tied to 0 if the TPH capability is not in use.</p>
14:13	tph_type[1:0]	2	<p>When a TPH is present in the request TLP, these two bits provide the value of the PH[1:0] field associated with the hint. The core samples this field in the first beat of a packet, when <code>s_axis_rq_tvalid</code> and <code>s_axis_rq_tready</code> are both High.</p> <p>These bits can be set to any value if <code>tph_present</code> is set to 0.</p>
15	tph_indirect_tag_en	1	<p>When this bit is set, the core uses the lower bits of <code>tph_st_tag[7:0]</code> as an index into its Steering Tag Table, and inserts the tag from this location in the transmitted request TLP.</p> <p>When this bit is 0, the core uses the value on <code>tph_st_tag[7:0]</code> directly as the Steering Tag.</p> <p>The core samples this bit in the first beat of a packet, when <code>s_axis_rq_tvalid</code> and <code>s_axis_rq_tready</code> are both High.</p> <p>This bit can be set to any value if <code>tph_present</code> is set to 0.</p>

Table 11: Sideband Signal Descriptions in s_axis_rq_tuser (cont'd)

Bit Index	Name	Width	Description
23:16	tph_st_tag[7:0]	8	When a TPH is present in the request TLP, this output provides the 8-bit Steering Tag associated with the hint. The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High. These bits can be set to any value if tph_present is set to 0.
27:24	seq_num[3:0]	4	You can optionally supply a 4-bit sequence number in this field to keep track of the progress of the request in the core transmit pipeline. The core outputs this sequence number on its pcie_rq_seq_num[3:0] output when the request TLP has progressed to a point in the pipeline where a Completion TLP is not able to pass it. The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High. This input can be hardwired to 0 when the user application is not monitoring the pcie_rq_seq_num[3:0] output of the core.
59:28	parity	32	Odd parity for the 256-bit data. When parity checking is enabled in the core, the user logic must set bit i of this bus to the odd parity computed for byte i of s_axis_rq_tdata. Only the lower 16 bits are used when the interface width is 128 bits, and only the lower 8 bits are used when the interface width is 64 bits. When an interface parity error is detected, it is recorded as an uncorrectable internal error and the packet is discarded. According to the Base Spec 6.2.9 (<i>PCI-SIG Specifications</i> (http://www.pcisig.com/specifications))), an uncorrectable internal error is an error that occurs within a component that results in improper operation of the component. The only method of recovering from an uncorrectable internal error is a reset or hardware replacement. The parity bits can be permanently tied to 0 if parity check is not enabled in the core.
61:60	seq_num[5:4]	2	Extension of seq_num as in [27:24].

Requester Completion Interface

The Requester Completion (RC) interface are the ports through which the completions received from the link in response to your requests are presented to the user application. The following table defines the ports in the RC interface of the core. In the Width column, DW denotes the configured data bus width (64, 128, or 256 bits).

Table 12: Requester Completion Interface Port Descriptions

Port	Direction	Width	Description
m_axis_rc_tdata	Output	DW	Requester Completion Data bus. Transmit data from the core requester completion interface to the user application. Only the lower 128 bits are used when the interface width is 128 bits, and only the lower 64 bits are used when the interface width is 64 bits. Bits [255:128] are set permanently to 0 by the core when the interface width is configured as 128 bits, and bits [255:64] are set permanently to 0 when the interface width is configured as 64 bits.

Table 12: Requester Completion Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
m_axis_rc_tuser	Output	75	<p>Requester Completion User Data.</p> <p>This set of signals contains sideband information for the TLP being transferred. These signals are valid when m_axis_rc_tvalid is High.</p> <p>The following table describes the individual signals in this set.</p>
m_axis_rc_tlast	Output	1	<p>TLAST indication for Requester Completion Data.</p> <p>The core asserts this signal in the last beat of a packet to indicate the end of the packet. When a TLP is transferred in a single beat, the core sets this bit in the first beat of the transfer. This output is used only when the straddle option is disabled. When the straddle option is enabled (for the 256-bit interface), the core sets this output permanently to 0.</p>
m_axis_rc_tkeep	Output	DW/32	<p>TKEEP indication for Requester Completion Data.</p> <p>The assertion of bit i of this bus during a transfer indicates that Dword i of the m_axis_rc_tdata bus contains valid data. The core sets this bit to 1 contiguously for all Dwords starting from the first Dword of the descriptor to the last Dword of the payload. Thus, m_axis_rc_tkeep sets to 1s in all beats of a packet, except in the final beat when the total size of the packet is not a multiple of the width of the data bus (both in Dwords). This is true for both Dword-aligned and address-aligned modes of payload transfer.</p> <p>Bits [7:4] of this bus are set permanently to 0 by the core when the interface width is configured as 128 bits, and bits [7:2] are set permanently to 0 when the interface width is configured as 64 bits.</p> <p>These outputs are permanently set to all 1s when the interface width is 256 bits and the straddle option is enabled. The user logic must use the signals in m_axis_rc_tuser in that case to determine the start and end of Completion TLPs transferred over the interface.</p>
m_axis_rc_tvalid	Output	1	<p>Requester Completion Data Valid.</p> <p>The core asserts this output whenever it is driving valid data on the m_axis_rc_tdata bus. The core keeps the valid signal asserted during the transfer of a packet. The user application can pace the data transfer using the m_axis_rc_tready signal.</p>
m_axis_rc_tready	Input	1	<p>Requester Completion Data Ready.</p> <p>Activation of this signal by the user logic indicates to the core that the user application is ready to accept data. Data is transferred across the interface when both m_axis_rc_tvalid and m_axis_rc_tready are asserted in the same cycle.</p> <p>If the user application deasserts the ready signal when the valid signal is High, the core maintains the data on the bus and keeps the valid signal asserted until the user application has asserted the ready signal.</p>

Table 13: Sideband Signal Descriptions in m_axis_rc_tuser

Bit Index	Name	Width	Description
31:0	byte_en	32	<p>The user logic can optionally use these byte enable bits to determine the valid bytes in the payload of a packet being transferred. The assertion of bit i of this bus during a transfer indicates that byte i of the <code>m_axis_rc_tdata</code> bus contains a valid payload byte. This bit is not asserted for descriptor bytes.</p> <p>Although the byte enables can be generated by user logic from information in the request descriptor (address and length), the logic has the option to use these signals directly instead of generating them from other interface signals. The 1 bit in this bus for the payload of a TLP is always contiguous.</p> <p>Bits [31:16] of this bus are set permanently to 0 by the core when the interface width is configured as 128 bits, and bits [31:8] are set permanently to 0 when the interface width is configured as 64 bits. The byte enable bit is also set on completions received in response to zero length memory read requests.</p>
32	is_sof_0	1	<p>Start of a first Completion TLP.</p> <p>For 64-bit and 128-bit interfaces, and for the 256-bit interface with no straddling, <code>is_sof_0</code> is asserted by the core in the first beat of a packet to indicate the start of the TLP. On these interfaces, only a single TLP can be started in a data beat, and <code>is_sof_1</code> is permanently set to 0. Use of this signal is optional when the straddle option is not enabled.</p> <p>When the interface width is 256 bits and the straddle option is enabled, the core can straddle two Completion TLPs in the same beat. In this case, the Completion TLPs are not formatted as AXI4-Stream packets. The assertion of <code>is_sof_0</code> indicates a Completion TLP starting in the beat. The first byte of this Completion TLP is in byte lane 0 if the previous TLP ended before this beat, or in byte lane 16 if the previous TLP continues in this beat.</p>
33	is_sof_1	1	<p>This signal is used when the interface width is 256 bits and the straddle option is enabled, when the core can straddle two Completion TLPs in the same beat. The output is permanently set to 0 in all other cases.</p> <p>The assertion of <code>is_sof_1</code> indicates a second Completion TLP starting in the beat, with its first byte in byte lane 16. The core starts a second TLP at byte position 16 only if the previous TLP ended in one of the byte positions 0-15 in the same beat; that is, only if <code>is_eof_0[0]</code> is also set in the same beat.</p>
37:34	is_eof_0[3:0]	4	<p>End of a first Completion TLP and the offset of its last Dword.</p> <p>These outputs are used only when the interface width is 256 bits and the straddle option is enabled.</p> <p>The assertion of the bit <code>is_eof_0[0]</code> indicates the end of a first Completion TLP in the current beat. When this bit is set, the bits <code>is_eof_0[3:1]</code> provide the offset of the last Dword of this TLP.</p>

Table 13: Sideband Signal Descriptions in m_axis_rc_tuser (cont'd)

Bit Index	Name	Width	Description
41:38	is_eof_1[3:0]	4	<p>End of a second Completion TLP and the offset of its last Dword.</p> <p>These outputs are used only when the interface width is 256 bits and the straddle option is enabled. The core can then straddle two Completion TLPs in the same beat. These outputs are reserved in all other cases.</p> <p>The assertion of is_eof_1[0] indicates a second TLP ending in the same beat. When bit 0 of is_eof_1 is set, bits [3:1] provide the offset of the last Dword of the TLP ending in this beat. Because the second TLP can only end at a byte position in the range 27-31, is_eof_1[3:1] can only take one of two values (6 or 7).</p> <p>The offset for the last byte of the second TLP can be determined from the starting address and length of the TLP, or from the byte enable signals byte_en[31:0].</p> <p>If is_eof_1[0] is High, the signals is_eof_0[0] and is_sof_1 are also High in the same beat.</p>
42	discontinue	1	<p>This signal is asserted by the core in the last beat of a TLP, if it has detected an uncorrectable error while reading the TLP payload from its internal FIFO memory. The user application must discard the entire TLP when such an error is signaled by the core.</p> <p>This signal is never asserted when the TLP has no payload. It is asserted only in the last beat of the payload transfer; that is, when is_eof_0[0] is High.</p> <p>When the straddle option is enabled, the core does not start a second TLP if it has asserted discontinue in a beat.</p> <p>When the core is configured as an Endpoint, the error is also reported by the core to the Root Complex to which it is attached, using Advanced Error Reporting (AER).</p>
74:43	parity	32	<p>Odd parity for the 256-bit transmit data.</p> <p>Bit i provides the odd parity computed for byte i of m_axis_rc_tdata. Only the lower 16 bits are used when the interface width is 128 bits, and only the lower 8 bits are used when the interface width is 64 bits. Bits [31:16] are set permanently to 0 by the core when the interface width is configured as 128 bits, and bits [31:8] are set permanently to 0 when the interface width is configured as 64 bits.</p>

512-bit Interfaces

This section provides the description for ports associated with the user interfaces of the core. When you select 512-bit interface, review the Pblock constraints in the Xilinx top XDC file of the example design. They are required to keep the soft 512-bit AXI4-Stream logic near the PCIe integrated block to improve the timing.

Related Information

[64/128/256-Bit Interfaces](#)

Completer Request Interface

Table 14: Completer Request Interface Port Descriptions (512-bit Interface)

Name	Width	Direction	Description
m_axis_cq_tdata	512	Output	Transmit data from the PCIe completer request interface to the user application.
m_axis_cq_tuser	183	Output	This is a set of signals containing sideband information for the TLP being transferred. These signals are valid when m_axis_cq_tvalid is High. The individual signals in this set are described in the following table.
m_axis_cq_tlast	1	Output	The core asserts this signal in the last beat of a packet to indicate the end of the packet. When a TLP is transferred in a single beat, the core sets this bit in the first beat of the transfer. This output is used only when the straddle option is disabled. When the straddle option is enabled, the core sets this output permanently to 0.
m_axis_cq_tkeep	16	Output	<p>The assertion of bit <i>i</i> of this bus during a transfer indicates to the user logic that Dword <i>i</i> of the m_axis_cq_tdata bus contains valid data. The core sets this bit to 1 contiguously for all Dwords starting from the first Dword of the descriptor to the last Dword of the payload. Thus, m_axis_cq_tdata is set to all 1s in all beats of a packet, except in the final beat when the total size of the packet is not a multiple of the width of the data bus (both in Dwords). This is true for both Dword-aligned and 128b address-aligned modes of payload transfer.</p> <p>The tkeep bits are valid only when straddle is not enabled on the CQ interface. When straddle is enabled, the tkeep bits are permanently set to all 1s in all beats. The user logic must use the is_sop/is_eop signals in the m_axis_cq_tuser bus in that case to determine the start and end of TLPs transferred over the interface.</p>
m_axis_cq_tvalid	1	Output	The core asserts this output whenever it is driving valid data on the m_axis_cq_tdata bus. The core keeps the valid signal asserted during the transfer of a packet. The user application can pace the data transfer using the m_axis_cq_tready signal.
m_axis_cq_tready	1	Input	Activation of this signal by the user logic indicates to the PCIe core that the user logic is ready to accept data. Data is transferred across the interface when both m_axis_cq_tvalid and m_axis_cq_tready are asserted in the same cycle. If the user logic deasserts the ready signal when m_axis_cq_tvalid is High, the core maintains the data on the bus and keeps the valid signal asserted until the user logic has asserted the ready signal.

Table 14: Completer Request Interface Port Descriptions (512-bit Interface) (cont'd)

Name	Width	Direction	Description
pcie_cq_np_req	2	Input	<p>This input is used by the user application to request the delivery of a Non-Posted request. The core implements a credit-based flow control mechanism to control the delivery of Non-Posted requests across the interface, without blocking Posted TLPs.</p> <p>This input to the core controls an internal credit count. The credit count is updated in each clock cycle based on the setting of pcie_cq_np_req[1:0] as follows:</p> <ul style="list-style-type: none"> • 00: No change • 01: Increment by 1 • 10 or 11: Increment by 2 <p>The credit count is decremented on the delivery of each Non-Posted request across the interface. The core temporarily stops delivering Non-Posted requests to the user logic when the credit count is zero. It continues to deliver any Posted TLPs received from the link even when the delivery of Non-Posted requests has been paused.</p> <p>The user application can either set pcie_cq_np_req[1:0] in each cycle based on the status of its Non-Posted request receive buffer, or can set it to 11 permanently if it does not need to exercise selective backpressure on Non-Posted requests.</p> <p>The setting of pcie_cq_np_req[1:0] does not need to be aligned with the packet transfers on the completer request interface.</p>
pcie_cq_np_req_count	6	Output	<p>This output provides the current value of the credit count maintained by the core for delivery of Non-Posted requests to the user logic. The core delivers a Non-Posted request across the completer request interface only when this credit count is non-zero. This counter saturates at a maximum limit of 32.</p> <p>Because of internal pipeline delays, there can be several cycles of delay between the user application providing credit on the pcie_cq_np_req[1:0] inputs and the PCIe core updating the pcie_cq_np_req_count output in response.</p> <p>This count resets on user_reset and de-assertion of user_lnk_up.</p>

Table 15: Sideband Signals in m_axis_cq_tuser (512-bit Interface)

Bit Index	Name	Width	Description
7:0	first_be[7:0]	8	<p>Byte enables for the first Dword of the payload. first_be[3:0] reflects the setting of the First Byte Enable bits in the Transaction-Layer header of the first TLP in this beat; and first_be[7:4] reflects the setting of the First Byte Enable bits in the Transaction-Layer header of the second TLP in this beat. For Memory Reads and I/O Reads, the 4 bits indicate the valid bytes to be read in the first Dword. For Memory Writes and I/O Writes, these bits indicate the valid bytes in the first Dword of the payload. For Atomic Operations and Messages with a payload, these bits are set to all 1s.</p> <p>Bits [7:4] of first_be are valid only when straddle is enabled on the CQ interface. When straddle is disabled, these bits are permanently set to 0s.</p> <p>This field is valid in the first beat of a packet. first_be[3:0] is valid when m_axis_cq_tvalid and is_sop[0] are both asserted High. first_be[7:4] is valid when m_axis_cq_tvalid and is_sop[1] are both asserted High.</p>
15:8	last_be[7:0]	8	<p>Byte enables for the last Dword of the payload. last_be[3:0] reflects the setting of the Last Byte Enable bits in the Transaction-Layer header of the first TLP in this beat; and last_be[7:4] reflects the setting of the Last Byte Enable bits in the Transaction-Layer header of the second TLP in this beat. For Memory Reads, the 4 bits indicate the valid bytes to be read in the last Dword of the block of data. For Memory Writes, these bits indicate the valid bytes in the ending Dword of the payload. For Memory Reads and Writes of one DW transfers and zero length transfers, these bits should be 0s. For Atomic Operations and Messages with a payload, these bits are set to all 1s.</p> <p>Bits [7:4] of last_be are valid only when straddle is enabled on the CQ interface. When straddle is disabled, these bits are permanently set to 0s.</p> <p>This field is valid in the first beat of a packet. last_be[3:0] is valid when m_axis_cq_tvalid and is_eop[0] are both asserted High. last_be[7:4] is valid when m_axis_cq_tvalid and is_eop[1] are both asserted High.</p>
79:16	byte_en[63:0]	64	<p>The user logic can optionally use these byte enable bits to determine the valid bytes in the payload of a packet being transferred. The assertion of bit <i>i</i> of this bus during a transfer indicates to the user logic that byte <i>i</i> of the m_axis_cq_tdata bus contains a valid payload byte. This bit is not asserted for descriptor bytes.</p> <p>Although the byte enables can be generated by user logic from information in the request descriptor (address and length), as well as the settings of the first_be and last_be signals, the user logic has the option of using these signals directly instead of generating them from other interface signals.</p> <p>When the payload size is more than 2 Dwords (8 bytes), the first bits on this bus for the payload are always contiguous. When the payload size is 2 Dwords or less, the first bits might be non-contiguous.</p> <p>For the special case of a zero-length memory write transaction defined by the PCI Express Specifications, the byte_en bits are all 0 when the associated 1 Dword payload is being transferred.</p>

Table 15: Sideband Signals in m_axis_cq_tuser (512-bit Interface) (cont'd)

Bit Index	Name	Width	Description
81:80	is_sop[1:0]	2	<p>Signals the start of a new TLP in this beat. These outputs are set in the first beat of a TLP. When straddle is disabled, only is_sop[0] is valid and is_sop[1] is permanently set to 0. When straddle is enabled, the settings are as follows:</p> <ul style="list-style-type: none"> • 00: No new TLP starting in this beat. • 01: A single new TLP starts in this beat. Its start position is indicated by is_sop0_ptr[1:0]. • 11: Two new TLPs are starting in this beat. is_sop0_ptr[1:0] provides the start position of the first TLP and is_sop1_ptr[1:0] provides the start position of the second TLP. • 10: Reserved. <p>Use of this signal is optional for the user logic when the straddle option is disabled, because a new TLP always starts in the beat following tlast assertion.</p>
83:82	is_sop0_ptr[1:0]	2	<p>Indicates the position of the first byte of the first TLP starting in this beat:</p> <ul style="list-style-type: none"> • 00: Byte lane 0 • 10: Byte lane 32 • 01, 11: Reserved <p>This field is valid only when the straddle option is enabled on the CQ interface. Otherwise, it is set to 0 permanently, as a TLP can only start in byte lane 0.</p>
85:84	is_sop1_ptr[1:0]	2	<p>Indicates the position of the first byte of the second TLP starting in this beat:</p> <ul style="list-style-type: none"> • 10: Byte lane 32 • 00, 01, 11: Reserved. <p>This output is used only when the straddle option is enabled on the CQ interface. The core can then straddle two TLPs in the same beat. The output is permanently set to 0 when straddle is disabled.</p>
87:86	is_eop[1:0]	2	<p>Indicates that a TLP is ending in this beat. These outputs are set in the final beat of a TLP. When straddle is disabled, only is_eop[0] is valid and is_eop[1] is permanently set to 0. When straddle is enabled, the settings are as follows:</p> <ul style="list-style-type: none"> • 00: No TLPs ending in this beat. • 01: A single TLP is ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of this TLP. • 11: Two TLPs are ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of the first TLP and is_eop1_ptr[3:0] provides the offset of the last Dword of the second TLP. • 10: Reserved. <p>The use of this signal is optional for the user logic when the straddle option is not enabled, because tlast is asserted in the final beat of a TLP.</p>
91:88	is_eop0_ptr[3:0]	4	Offset of the last Dword of the first TLP ending in this beat. This output is valid when is_eop[0] is asserted.
95:92	is_eop1_ptr[3:0]	4	Offset of the last Dword of the second TLP ending in this beat. This output is valid when is_eop[1] is asserted. The output is permanently set to 0 when straddle is disabled.

Table 15: Sideband Signals in m_axis_cq_tuser (512-bit Interface) (cont'd)

Bit Index	Name	Width	Description
96	discontinue	1	<p>This signal is asserted by the core in the last beat of a TLP, if it has detected an uncorrectable error while reading the TLP payload from its internal FIFO memory. The user application must discard the entire TLP when such an error is signaled by the core.</p> <p>This signal is never asserted when the TLP has no payload. It is asserted only in the last beat of the payload transfer, that is when is_eop[0] is High.</p> <p>When the straddle option is enabled, the core does not start a second TLP if it has asserted discontinue in a beat.</p> <p>When the core is configured as an Endpoint, the error is also reported by the core to the Root Complex it is attached to, using Advanced Error Reporting (AER).</p>
98:97	tph_present[1:0]	2	<p>These bits indicate the presence of a Transaction Processing Hint (TPH) in the request TLP being delivered across the interface.</p> <ul style="list-style-type: none"> tph_present[0] indicates the presence of a hint in the first TLP of this beat. tph_present[0] is valid when m_axis_cq_tvalid and is_sop[0] are both High. tph_present[1] indicates the presence of a hint in the second TLP of this beat. tph_present[1] is valid when m_axis_cq_tvalid and is_sop[1] are both High.
102:99	tph_type[3:0]	4	<p>When a TPH is present in the request TLP, these two bits provide the value of the PH[1:0] field associated with the hint.</p> <ul style="list-style-type: none"> tph_type[1:0] provides the TPH type associated with the first TLP of this beat. tph_type[1:0] is valid when m_axis_cq_tvalid and is_sop[0] are both High. tph_type[3:2] provides the TPH type associated with the second TLP of this beat. tph_type[3:2] is valid when m_axis_cq_tvalid and is_sop[1] are both High.
118:103	tph_st_tag[15:0]	8	<p>When a TPH is present in the request TLP, this output provides the 8-bit Steering Tag associated with the hint.</p> <ul style="list-style-type: none"> tph_st_tag[7:0] provides the Steering Tag associated with the first TLP of this beat. tph_st_tag[7:0] is valid when m_axis_cq_tvalid and is_sop[0] are both High. tph_st_tag[15:8] provides the Steering Tag associated with the second TLP of this beat. tph_st_atg[15:8] is valid when m_axis_cq_tvalid and is_sop[1] are both High.
182:119	parity	64	Odd parity for the 512-bit transmit data. Bit <i>i</i> provides the odd parity computed for byte <i>i</i> of m_axis_cq_tdata.

Completer Completion Interface

Table 16: Completer Completion Interface Port Descriptions (512-bit Interface)

Name	Width	Direction	Description
s_axis_cc_tdata	512	Input	Completion data from the user application to the PCIe core.

Table 16: Completer Completion Interface Port Descriptions (512-bit Interface) (cont'd)

Name	Width	Direction	Description
s_axis_cc_tuser	81	Input	<p>This is a set of signals containing sideband information for the TLP being transferred. These signals are valid when s_axis_cc_tvalid is High.</p> <p>The individual signals in this set are described in the following table.</p>
s_axis_cc_tlast	1	Input	<p>The user application must assert this signal in the last cycle of a packet to indicate the end of the packet. When the TLP is transferred in a single beat, the user application must set this bit in the first cycle of the transfer.</p> <p>This input is used by the core only when the straddle option is disabled. When the straddle option is enabled, the core ignores the setting of this input, using instead the is_sop/is_eop signals in the s_axis_cc_tuser bus to determine the start and end of TLPs.</p>
s_axis_cc_tkeep	16	Input	<p>The assertion of bit <i>i</i> of this bus during a transfer indicates to the core that Dword <i>i</i> of the s_axis_cc_tdata bus contains valid data. The user logic must set this bit to 1 contiguously for all Dworls starting from the first Dword of the descriptor to the last Dword of the payload. Thus, s_axis_cc_tdata must be set to all 1s in all beats of a packet, except in the final beat when the total size of the packet is not a multiple of the width of the data bus (both in Dworls). This is true for both Dword-aligned and 128b address-aligned modes of payload transfer.</p> <p>The keep bits are valid only when straddle is not enabled on the CC interface. When straddle is enabled, the core ignores the setting of these bits when receiving data across the interface. The user logic must set the is_sop/is_eop signals in the s_axis_cc_tuser bus in that case to signal the start and end of TLPs transferred over the interface.</p>
s_axis_cc_tvalid	1	Input	<p>The user application must assert this output whenever it is driving valid data on the s_axis_cc_tdata bus. The user application must keep the valid signal asserted during the transfer of a packet. The core paces the data transfer using the s_axis_cc_tready signal.</p>
s_axis_cc_tready	4	Output	<p>Activation of this signal by the PCIe core indicates that it is ready to accept data. Data is transferred across the interface when both s_axis_cc_tvalid and s_axis_cc_tready are asserted in the same cycle.</p> <p>If the core deasserts the ready signal when the valid signal is High, the user logic must maintain the data on the bus and keep the valid signal asserted until the core has asserted the ready signal.</p> <p>With this output port, each bit indicates the same value, so the user logic can use any of the bit.</p>

Table 17: Sideband Signals in s_axis_cc_tuser

Bit Index	Name	Width	Description
1:0	is_sop[1:0]	2	<p>Signals the start of a new TLP in this beat. These outputs are set in the first beat of a TLP. When straddle is disabled, only is_sop[0] is valid. When straddle is enabled, the settings are as follows:</p> <ul style="list-style-type: none"> • 00: No new TLP starting in this beat. • 01: A single new TLP starts in this beat. Its start position is indicated by is_sop0_ptr[1:0]. • 11: Two new TLPs are starting in this beat. is_sop0_ptr[1:0] provides the start position of the first TLP and is_sop1_ptr[1:0] provides the start position of the second TLP. • 10: Reserved. <p>This field is used by the core only when the straddle option is enabled. When straddle is disabled, the core uses tlast to determine the first beat of an incoming TLP.</p>
3:2	is_sop0_ptr[1:0]	2	<p>Indicates the position of the first byte of the first TLP starting in this beat:</p> <ul style="list-style-type: none"> • 00: Byte lane 0 • 10: Byte lane 32 • 01, 11: Reserved <p>This field is used by the core only when the straddle option is enabled. When straddle is disabled, the user logic must always start a TLP in byte lane 0.</p>
5:4	is_sop1_ptr[1:0]	2	<p>Indicates the position of the first byte of the second TLP starting in this beat:</p> <ul style="list-style-type: none"> • 10: Byte lane 32 • 00, 01, 11: Reserved. <p>This input is used only when the straddle option is enabled on the CC interface. The user can then straddle two TLPs in the same beat.</p>
7:6	is_eop[1:0]	2	<p>Signals that a TLP is ending in this beat. These outputs are set in the final beat of a TLP. When straddle is disabled, only is_eop[0] is valid. When straddle is enabled, the settings are as follows:</p> <ul style="list-style-type: none"> • 00: No TLPs ending in this beat. • 01: A single TLP is ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of this TLP. • 11: Two TLPs are ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of the first TLP and is_eop1_ptr[3:0] provides the offset of the last Dword of the second TLP. • 10: Reserved. <p>This field is used by the core only when the straddle option is enabled. When straddle is disabled, the core uses tlast and tkeep to determine the ending beat and position of EOP.</p>
11:8	is_eop0_ptr[3:0]	4	<p>Offset of the last Dword of the first TLP ending in this beat. This output is valid when is_eop[0] is asserted.</p> <p>This field is used by the core only when the straddle option is enabled.</p>

Table 17: Sideband Signals in s_axis_cc_tuser (cont'd)

Bit Index	Name	Width	Description
15:12	is_eop1_ptr[3:0]	4	Offset of the last Dword of the second TLP ending in this beat. This output is valid when is_eop[1] is asserted. This field is used by the core only when the straddle option is enabled.
16	discontinue	1	<p>This signal can be asserted by the user application during a transfer if it has detected an error (such as an uncorrectable ECC error while reading the payload from memory) in the data being transferred and needs to abort the packet. The core nullifies the corresponding TLP on the link to avoid data corruption.</p> <p>The user logic can assert this signal in any beat during the transfer except the first beat of the TLP. It can either choose to terminate the packet prematurely in the cycle where the error was signaled, or continue until all bytes of the payload are delivered to the core. In the latter case, the core treats the error as sticky for the following beats of the packet, even if the user logic deasserts the discontinue signal before the end of the packet.</p> <p>The discontinue signal can be asserted only when s_axis_cc_tvalid is High. The core samples this signal only when s_axis_cc_tready is High. Thus, once asserted, it should not be deasserted until s_axis_cc_tready is High.</p> <p>When the straddle option is enabled on the CC interface, the user should not start a new TLP in the same beat when a TLP is ending with discontinue asserted.</p> <p>When the core is configured as an Endpoint, this error is also reported by the core to the Root Complex it is attached to, using Advanced Error Reporting (AER).</p>
80:17	parity	64	<p>Odd parity for the 256-bit data. When parity checking is enabled in the core, user logic must set bit <i>i</i> of this bus to the odd parity computed for byte <i>i</i> of s_axis_cc_tdata.</p> <p>On detection of a parity error, the core nullifies the corresponding TLP on the link and reports it as an Uncorrectable Internal Error.</p> <p>The parity bits can be permanently tied to 0 if parity check is not enabled in the core.</p>

Requester Request Interface

Table 18: Requester Request Interface Port Descriptions (512-bit Interface)

Name	Width	Direction	Description
s_axis_rq_tdata	512	Input	Requester-side request data from the user application to the PCIe core.
s_axis_rq_tuser	137	Input	This is a set of signals containing sideband information for the TLP being transferred. These signals are valid when s_axis_rq_tvalid is High. The individual signals in this set are described in the following table.

Table 18: Requester Request Interface Port Descriptions (512-bit Interface) (cont'd)

Name	Width	Direction	Description
s_axis_rq_tlast	1	Input	<p>The user application must assert this signal in the last cycle of a TLP to indicate the end of the packet. When the TLP is transferred in a single beat, the user logic must set this bit in the first cycle of the transfer.</p> <p>This input is used by the core only when the straddle option is disabled. When the straddle option is enabled, the core ignores the setting of this input, using instead the is_sop/is_eop signals in the s_axis_rq_tuser bus to determine the start and end of TLPs.</p>
s_axis_rq_tkeep	16	Input	<p>The assertion of bit <i>i</i> of this bus during a transfer indicates to the core that Dword <i>i</i> of the s_axis_rq_tdata bus contains valid data. The user logic must set this bit to 1 contiguously for all Dwords starting from the first Dword of the descriptor to the last Dword of the payload. Thus, s_axis_rq_tdata must be set to all 1s in all beats of a packet, except in the final beat when the total size of the packet is not a multiple of the width of the data bus (both in Dwords). This is true for both Dword-aligned and 128b address-aligned modes of payload transfer.</p> <p>The tkeep bits are valid only when straddle is not enabled on the RQ interface. When straddle is enabled, the core ignores the setting of these bits when receiving data across the interface. The user logic must set the is_sop/is_eop signals in the s_axis_rq_tuser bus in that case to signal the start and end of TLPs transferred over the interface.</p>
s_axis_rq_tvalid	1	Input	<p>The user application must assert this output whenever it is driving valid data on the s_axis_rq_tdata bus. The user application must keep the valid signal asserted during the transfer of a packet. The core paces the data transfer using the s_axis_rq_tready signal.</p>
s_axis_rq_tready	4	Output	<p>Activation of this signal by the PCIe core indicates that it is ready to accept data. Data is transferred across the interface when both s_axis_rq_tvalid and s_axis_rq_tready are asserted in the same cycle.</p> <p>If the core deasserts the ready signal when the valid signal is High, the user logic must maintain the data on the bus and keep the valid signal asserted until the core has asserted the ready signal.</p> <p>With this output port, each bit indicates the same value, so the user logic can use any of the bit.</p>
pcie_rq_tag_vld0	1	Output	<p>The core asserts this output for one cycle when it has allocated a tag to an incoming Non-Posted request from the requester request interface and placed it on the pcie_rq_tag0 output. The bit is encoded as follows:</p> <ul style="list-style-type: none"> • 0: No tags being provided in this cycle. • 1: A tag is presented on pcie_rq_tag0.
pcie_rq_tag_vld1	1	Output	<p>The core asserts this output for one cycle when it has allocated a tag to an incoming Non-Posted request from the requester request interface and placed it on the pcie_rq_tag1 output. The bit is encoded as follows:</p> <ul style="list-style-type: none"> • 0: No tag is provided on pcie_rq_tag1 in this cycle. • 1: A tag is presented on pcie_rq_tag1.

Table 18: Requester Request Interface Port Descriptions (512-bit Interface) (cont'd)

Name	Width	Direction	Description
pcie_rq_tag0	8	Output	<p>When tag management for Non-Posted requests is performed by the core (Enable Client Tag is unchecked in the IP customization GUI), this output is used by the core to communicate the allocated tag for each Non-Posted request received from the client. The tag value on pcie_rq_tag0 is valid for one cycle when pcie_rq_tag_vld0 is High. The client must copy this tag and use it to associate the completion data with the pending request.</p> <p>There can be a delay of several cycles between the transfer of the request on the s_axis_rq_tdata bus and the assertion of pcie_rq_tag_vld0 by the core to provide the allocated tag for the request. The client can, meanwhile, continue to send new requests. The tags for requests are communicated on this bus in FIFO order. Therefore, the user application must associate the allocated tags with the requests in the order in which the requests were transferred over the interface.</p> <p>When pcie_rq_tag0 and pcie_rq_tag1 are both valid in the same cycle, the value on pcie_rq_tag0 corresponds to the earlier of the two requests transferred over the interface.</p>
pcie_rq_tag1	8	Output	The description of this signal is the same as pcie_rq_tag0, except the tag value on pcie_rq_tag1 is valid for one cycle when pcie_rq_tag_vld1 is asserted.
pcie_rq_seq_num0	6	Output	<p>The user may optionally use this output to keep track of the progress of the request in the core's transmit pipeline. To use this feature, the user application must provide a sequence number for each request on the s_axis_rq_seq_num0[5:0] bus. The core outputs this sequence number on the pcie_rq_seq_num0[5:0] output when the request TLP has progressed to a point in the pipeline where a Completion TLP from the client will not be able to pass it. This mechanism enables the client to maintain ordering between Completions sent to the completer completion interface of the core and Posted requests sent to the requester request interface.</p> <p>Data on the pcie_rq_seq_num0[5:0] output is valid when pcie_rq_seq_num_vld0 is High.</p>
pcie_rq_seq_num1	6	Output	<p>This output is identical in function to that of pcie_rq_seq_num0. It is used to provide a second sequence number in the same cycle when a first sequence number is being presented on pcie_rq_seq_num0.</p> <p>Data on the pcie_rq_seq_num1[5:0] output is valid when pcie_rq_seq_num_vld1 is High.</p>
pcie_rq_seq_num_vld0	1	Output	This output is asserted by the core for one cycle when it has placed valid data on pcie_rq_seq_num0[5:0].
pcie_rq_seq_num_vld1	1	Output	This output is asserted by the core for one cycle when it has placed valid data on pcie_rq_seq_num1[5:0].

Table 19: Sideband Signals in s_axis_rq_tuser (512-bit Interface)

Bit Index	Name	Width	Description
7:0	first_be[7:0]	8	<p>Byte enables for the first Dword. This field must be set based on the desired value of the First_BE bits in the Transaction-Layer header of the request TLP. first_be[3:0] corresponds to the byte enables for the first TLP starting in this beat, and first_be[7:4] corresponds to the byte enables for the second TLP starting in this beat (if present).</p> <p>For Memory Reads, I/O Reads and Configuration Reads, these 4 bits indicate the valid bytes to be read in the first Dword. For Memory Writes, I/O Writes and Configuration Writes, these bits indicate the valid bytes in the first Dword of the payload.</p> <p>The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High.</p>
15:8	last_be[7:0]	8	<p>Byte enables for the last Dword.</p> <p>This field must be set based on the desired value of the Last_BE bits in the Transaction-Layer header of the TLP. last_be[3:0] corresponds to the byte enables for the first TLP starting in this beat, and last_be[7:4] corresponds to the byte enables for the second TLP starting in this beat (if present).</p> <p>For Memory Reads and Writes of one DW transfers and zero length transfers, these bits should be 0s.</p> <p>For Memory Reads of 2 Dwords or more, these 4 bits indicate the valid bytes to be read in the last Dword of the block of data. For Memory Writes of 2 Dwords or more, these bits indicate the valid bytes in the last Dword of the payload.</p> <p>The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High.</p>
19:16	addr_offset[3:0]	4	<p>When 128b the address-aligned mode is in use on this interface, the user application must provide the offset where the payload data begins (in multiples of 4 bytes) on the data bus on this sideband bus. This enables the core to determine the alignment of the data block being transferred.</p> <p>addr_offset[1:0] corresponds to the offset for the first TLP starting in this beat, and addr_offset[3:2] is reserved for future use.</p> <p>The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High.</p> <p>When the requester request interface is configured in the Dword-alignment mode, these bits must always be set to 0.</p>

Table 19: Sideband Signals in s_axis_rq_tuser (512-bit Interface) (cont'd)

Bit Index	Name	Width	Description
21:20	is_sop[1:0]	2	<p>Signals the start of a new TLP in this beat. These outputs are set in the first beat of a TLP. When straddle is disabled, only is_sop[0] is valid. When straddle is enabled, the settings are as follows:</p> <ul style="list-style-type: none"> • 00: No new TLP starting in this beat. • 01: A single new TLP starts in this beat. Its start position is indicated by is_sop0_ptr[1:0]. • 11: Two new TLPs are starting in this beat. is_sop0_ptr[1:0] provides the start position of the first TLP and is_sop1_ptr[1:0] provides the start position of the second TLP. • 10: Reserved. <p>Use of this signal is optional for the user logic when the straddle option is not enabled, because a new TLP always starts in the beat following tlast assertion.</p>
23:22	is_sop0_ptr[1:0]	2	<p>Indicates the position of the first byte of the first TLP starting in this beat:</p> <ul style="list-style-type: none"> • 00: Byte lane 0 • 10: Byte lane 32 • 01, 11: Reserved
25:24	is_sop1_ptr[1:0]	2	<p>Indicates the position of the first byte of the second TLP starting in this beat:</p> <ul style="list-style-type: none"> • 10: Byte lane 32 • 00, 01, 11: Reserved. <p>This output is used only when the straddle option is enabled on the interface.</p>
27:26	is_eop[1:0]	2	<p>Signals that a TLP is ending in this beat. These outputs are set in the final beat of a TLP. When straddle is disabled, only is_eop[0] is valid. When straddle is enabled, the settings are as follows:</p> <ul style="list-style-type: none"> • 00: No TLPs ending in this beat. • 01: A single TLP is ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of this TLP. • 11: Two TLPs are ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of the first TLP and is_eop1_ptr[3:0] provides the offset of the last Dword of the second TLP. • 10: Reserved. <p>Use of this signal is optional for the user logic when the straddle option is not enabled, because tlast is asserted in the final beat of a TLP.</p>
31:28	is_eop0_ptr[3:0]	4	Offset of the last Dword of the first TLP ending in this beat. This output is valid when is_eop[0] is asserted.
35:32	is_eop1_ptr[3:0]	4	Offset of the last Dword of the second TLP ending in this beat. This output is valid when is_eop[1] is asserted.

Table 19: Sideband Signals in s_axis_rq_tuser (512-bit Interface) (cont'd)

Bit Index	Name	Width	Description
36	discontinue	1	<p>This signal can be asserted by the user application during a transfer if it has detected an error in the data being transferred and needs to abort the packet. The core nullifies the corresponding TLP on the link to avoid data corruption.</p> <p>The user logic can assert this signal in any beat of a TLP except the first beat during its transfer. It can either choose to terminate the packet prematurely in the cycle where the error was signaled, or continue until all bytes of the payload are delivered to the core. In the latter case, the core treats the error as sticky for the following beats of the packet, even if the user logic deasserts the discontinue signal before the end of the packet.</p> <p>The discontinue signal can be asserted only when s_axis_rq_tvalid is High. The core samples this signal only when s_axis_rq_tready is High. Thus, once asserted, it should not be deasserted until s_axis_rq_tready is High.</p> <p>When the straddle option is enabled on the RQ interface, the user should not start a new TLP in the same beat when a TLP is ending with discontinue asserted.</p> <p>When the core is configured as an Endpoint, this error is also reported by the core to the Root Complex it is attached to, using Advanced Error Reporting (AER).</p>
38:37	tph_present[1:0]	2	<p>This bit indicates the presence of a Transaction Processing Hint (TPH) in the request TLP being delivered across the interface. The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High.</p> <ul style="list-style-type: none"> • tph_present[0] corresponds to the first TLP starting in this beat. • tph_present[1] corresponds to the second TLP starting in this beat (if present). <p>These inputs must be permanently tied to 0 if the TPH capability is not in use.</p>
42:39	tph_type[3:0]	4	<p>When a TPH is present in the request TLP, these two bits provide the value of the PH[1:0] field associated with the hint. The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High.</p> <ul style="list-style-type: none"> • tph_type[1:0] corresponds to the first TLP starting in this beat. • tph_type[3:2] corresponds to the second TLP starting in this beat (if present). <p>These bits can be set to any value if the corresponding tph_present bit is set to 0.</p>
44:43	tph_indirect_tag_en[1:0]	2	<p>When this bit is set, the core uses the lower bits of the tag presented on tph_st_tag as an index into its Steering Tag Table, and insert the tag from this location in the transmitted request TLP. When this bit is 0, the core uses the value on tph_st_tag directly as the Steering Tag.</p> <ul style="list-style-type: none"> • tph_indirect_tag_en[0] corresponds to the first TLP starting in this beat. • tph_indirect_tag_en[1] corresponds to the second TLP starting in this beat (if present). <p>The core samples this bit in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High.</p> <p>These inputs can be set to any value if the corresponding tph_present bit is set to 0.</p>

Table 19: Sideband Signals in s_axis_rq_tuser (512-bit Interface) (cont'd)

Bit Index	Name	Width	Description
60:45	tph_st_tag[15:0]	16	<p>When a TPH is present in the request TLP, this output provides the 8-bit Steering Tag associated with the hint. The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High.</p> <ul style="list-style-type: none"> • tph_st_tag[7:0] corresponds to the first TLP starting in this beat. • tph_st_tag[15:8] corresponds to the second TLP starting in this beat (if present). <p>These inputs can be set to any value if the corresponding tph_present bit is set to 0.</p>
66:61	seq_num0[5:0]	6	<p>The user logic can optionally supply a 6-bit sequence number in this field to keep track of the progress of the request in the core's transmit pipeline. The core outputs this sequence number on its pcie_rq_seq_num0 or pcie_rq_seq_num1 output when the request TLP has progressed to a point in the pipeline where a Completion TLP from the user logic is not able to pass it.</p> <p>The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High.</p> <p>This input can be hardwired to 0 when the user logic is not monitoring the pcie_rq_seq_num* outputs of the core.</p>
72:67	seq_num1[5:0]	6	<p>If there is a second TLP starting in the same beat, the user logic can optionally provide a 6-bit sequence number for this TLP on this input. This sequence number is used in the same manner as seq_num0.</p> <p>The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High.</p> <p>This input can be hardwired to 0 when the user logic is not monitoring the pcie_rq_seq_num* outputs of the core.</p>
136:73	parity	64	<p>Odd parity for the 512-bit data. When parity checking is enabled in the core, user logic must set bit <i>i</i> of this bus to the odd parity computed for byte <i>i</i> of s_axis_rq_tdata.</p> <p>On detection of a parity error, the core nullifies the corresponding TLP on the link and reports it as an Uncorrectable Internal Error.</p> <p>These bits can be set to 0 if parity checking is disabled in the core.</p>

Requester Completion Interface

Table 20: Requester Completion Interface Port Descriptions (512-bit Interface)

Name	Width	Direction	Description
m_axis_rc_tdata	512	Output	Transmit data from the PCIe requester completion interface to the user application.
m_axis_rc_tuser	161	Output	This is a set of signals containing sideband information for the TLP being transferred. These signals are valid when m_axis_rc_tvalid is High. The individual signals in this set are described in the following table.

Table 20: Requester Completion Interface Port Descriptions (512-bit Interface) (cont'd)

Name	Width	Direction	Description
m_axis_rc_tlast	1	Output	The core asserts this signal in the last beat of a packet to indicate the end of the packet. When a TLP is transferred in a single beat, the core sets this bit in the first beat of the transfer. This output is used only when the straddle option is disabled. When the straddle option is enabled, the core sets this output permanently to 0.
m_axis_rc_tkeep	16	Output	<p>The assertion of bit <i>i</i> of this bus during a transfer indicates to the user logic that Dword <i>i</i> of the m_axis_rc_tdata bus contains valid data. The core sets this bit to 1 contiguously for all Dwords starting from the first Dword of the descriptor to the last Dword of the payload. Thus, m_axis_rc_tkeep is set to all 1s in all beats of a packet, except in the final beat when the total size of the packet is not a multiple of the width of the data bus (both in Dwords). This is true for both Dword-aligned and address-aligned modes of payload transfer.</p> <p>These outputs are permanently set to all 1s when the straddle option is enabled. The user logic must use the signals in m_axis_rc_tuser in that case to determine the start and end of Completion TLPs transferred over the interface.</p>
m_axis_rc_tvalid	1	Output	The core asserts this output whenever it is driving valid data on the m_axis_rc_tdata bus. The core keeps the valid signal asserted during the transfer of a packet. The user application can pace the data transfer using the m_axis_rc_tready signal.
m_axis_rc_tready	1	Input	<p>Activation of this signal by the user logic indicates to the PCIe core that the user logic is ready to accept data. Data is transferred across the interface when both m_axis_rc_tvalid and m_axis_rc_tready are asserted in the same cycle.</p> <p>If the user logic deasserts the ready signal when the valid signal is High, the core maintains the data on the bus and keep the valid signal asserted until the user logic has asserted the ready signal.</p>

Table 21: Sideband Signals in m_axis_rc_tuser (512-bit Interface)

Bit Index	Name	Width	Description
63:0	byte_en	64	<p>The client logic may optionally use these byte enable bits to determine the valid bytes in the payload of a packet being transferred. The assertion of bit <i>i</i> of this bus during a transfer indicates to the client that byte <i>i</i> of the m_axis_cq_tdata bus contains a valid payload byte. This bit is not asserted for descriptor bytes.</p> <p>Although the byte enables can be generated by client logic from information in the request descriptor (address and length), the client has the option of using these signals directly instead of generating them from other interface signals. The 1 bits in this bus for the payload of a TLP are always contiguous.</p>

Table 21: Sideband Signals in m_axis_rc_tuser (512-bit Interface) (cont'd)

Bit Index	Name	Width	Description
67:64	is_sop[3:0]	4	<p>Signals the start of a new TLP in this beat. These outputs are set in the first beat of a TLP. When straddle is disabled, only is_sop[0] is valid and is_sop[3:1] are permanently set to 0. When straddle is enabled, the settings are as follows:</p> <ul style="list-style-type: none"> • 0000: No new TLP starting in this beat. • 0001: A single new TLP starts in this beat. ts start position is indicated by is_sop0_ptr[1:0]. • 0011: Two new TLPs are starting in this beat. is_sop0_ptr[1:0] provides the start position of the first TLP and is_sop1_ptr[1:0] provides the start position of the second TLP. • 0111: Three new TLPs are starting in this beat. is_sop0_ptr[1:0] provides the start position of the first TLP, is_sop1_ptr[1:0] provides the start position of the second TLP, and is_sop2_ptr[1:0] provides the start position of the third TLP. • 1111: Four new TLPs are starting in this beat. is_sop0_ptr[1:0] provides the start position of the first TLP, is_sop1_ptr[1:0] provides the start position of the second TLP, is_sop2_ptr[1:0] provides the start position of the third TLP, and is_sop3_ptr[1:0] provides the start position of the fourth TLP. • All other settings are reserved. <p>Use of this signal is optional for the client when the straddle option is not enabled, because a new TLP always starts in the beat following m_axis_rc_tlast assertion.</p>
69:68	is_sop0_ptr[1:0]	2	<p>Indicates the position of the first byte of the first TLP starting in this beat:</p> <ul style="list-style-type: none"> • 00: Byte lane 0 • 01: Byte lane 16 • 10: Byte lane 32 • 11: Byte lane 48 <p>This field is valid only when the straddle option is enabled on the RC interface. Otherwise, it is set to 0 permanently, as a TLP can only start in byte lane 0.</p>
71:70	is_sop1_ptr[1:0]	2	<p>Indicates the position of the first byte of the second TLP starting in this beat:</p> <ul style="list-style-type: none"> • 00: Reserved • 01: Byte lane 16 • 10: Byte lane 32 • 11: Byte lane 48 <p>This output is used only when the straddle option is enabled on the RC interface. The output is permanently set to 0 when straddle is disabled.</p>

Table 21: Sideband Signals in m_axis_rc_tuser (512-bit Interface) (cont'd)

Bit Index	Name	Width	Description
73:72	is_sop2_ptr[1:0]	2	<p>Indicates the position of the first byte of the third TLP starting in this beat:</p> <ul style="list-style-type: none"> • 00: Reserved • 01: Reserved • 10: Byte lane 32 • 11: Byte lane 48 <p>This output is used only when the straddle option is enabled on the RC interface. The output is permanently set to 0 when straddle is disabled.</p>
75:74	is_sop3_ptr[1:0]	2	<p>Indicates the position of the first byte of the fourth TLP starting in this beat:</p> <ul style="list-style-type: none"> • 00, 01, 10: Reserved • 11: Byte lane 48 <p>This output is used only when the straddle option is enabled on the RC interface. The output is permanently set to 0 when straddle is disabled.</p>
79:76	is_eop[3:0]	4	<p>Signals that one or more TLPs are ending in this beat only when straddle is enabled. These outputs are set in the final beat of a TLP. The settings are as follows:</p> <ul style="list-style-type: none"> • 0000: No TLPs ending in this beat. • 0001: A single TLP is ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of this TLP. • 0011: Two TLPs are ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of the first TLP and is_eop1_ptr[3:0] provides the offset of the last Dword of the second TLP. • 0111: Three TLPs are ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of the first TLP, is_eop1_ptr[3:0] provides the offset of the last Dword of the second TLP, and is_eop2_ptr[3:0] provides the offset of the last Dword of the third TLP. • 1111: Four TLPs are ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of the first TLP, is_eop1_ptr[3:0] provides the offset of the last Dword of the second TLP, is_eop2_ptr[3:0] provides the offset of the last Dword of the third TLP, and is_eop3_ptr[3:0] provides the offset of the last Dword of the fourth TLP. • All other settings are reserved. <p>When the straddle option is disabled, m_axis_rc_tlast indicates the final beat of a TLP.</p>
83:80	is_eop0_ptr[3:0]	4	<p>Offset of the last Dword of the first TLP ending in this beat. This output is valid when is_eop[0] is asserted.</p> <p>This output is used only when the straddle option is enabled on the RC interface. The output is permanently set to 0 when straddle is disabled.</p>
87:84	is_eop1_ptr[3:0]	4	<p>Offset of the last Dword of the second TLP ending in this beat. This output is valid when is_eop[1] is asserted.</p> <p>This output is used only when the straddle option is enabled on the RC interface. The output is permanently set to 0 when straddle is disabled.</p>

Table 21: Sideband Signals in m_axis_rc_tuser (512-bit Interface) (cont'd)

Bit Index	Name	Width	Description
91:88	is_eop2_ptr[3:0]	4	Offset of the last Dword of the third TLP ending in this beat. This output is valid when is_eop[2] is asserted. This output is used only when the straddle option is enabled on the RC interface. The output is permanently set to 0 when straddle is disabled.
95:92	is_eop3_ptr[3:0]	4	Offset of the last Dword of the fourth TLP ending in this beat. This output is valid when is_eop[3] is asserted. This output is used only when the straddle option is enabled on the RC interface. The output is permanently set to 0 when straddle is disabled.
96	discontinue	1	This signal is asserted by the core in the last beat of a TLP, if it has detected an uncorrectable error while reading the TLP payload from its internal FIFO memory. The client application must discard the entire TLP when such an error is signaled by the core. This signal is never asserted when the TLP has no payload. It is asserted only in the last beat of the payload transfer, that is when is_eop[0] is High. When the straddle option is enabled, the core does not start a new TLP if it has asserted discontinue in a beat. When the core is configured as an Endpoint, the error is also reported by the core to the Root Complex it is attached to, using Advanced Error Reporting (AER).
160:97	parity	64	Odd parity for the 512-bit transmit data. Bit <i>i</i> provides the odd parity computed for byte <i>i</i> of m_axis_cq_tdata.

Other Core Interfaces

The core also provides the interfaces described in this section.

Power Management Interface

The following table defines the ports in the Power Management interface of the core.

Table 22: Power Management Interface Ports

Port	Width	Direction	Description
cfg_pm_aspm_l1_entry_reject	1	Input	Configuration Power Management ASPM L1 Entry Reject: When driven to 1b, Downstream Port rejects transition requests to L1 state.
cfg_pm_aspm_tx_l0s_entry_disable	1	Input	Configuration Power Management ASPM L0s Entry Disable: When driven to 1b, prevents the Port from entering TX L0s.

Configuration Management Interface

The Configuration Management interface is used to read and write to the Configuration Space Registers. The following table defines the ports in the Configuration Management interface of the core.

Table 23: Configuration Management Interface Port Descriptions

Port	Direction	Width	Description
cfg_mgmt_addr	Input	10	Read/Write Address Configuration Space Dword-aligned address.
cfg_mgmt_function_number	Input	8	PCI Function Number Selects the PCI function number for the configuration register read/write.
cfg_mgmt_write	Input	1	Write Enable Asserted for a write operation. Active-High.
cfg_mgmt_write_data	Input	32	Write data Write data is used to configure the Configuration and Management registers.
cfg_mgmt_byte_enable	Input	4	Byte Enable Byte enable for write data, where cfg_mgmt_byte_enable[0] corresponds to cfg_mgmt_write_data[7:0], and so on.
cfg_mgmt_read	Input	1	Read Enable Asserted for a read operation. Active-High.
cfg_mgmt_read_data	Output	32	Read data out Read data provides the configuration of the Configuration and Management registers.
cfg_mgmt_read_write_done	Output	1	Read/Write operation complete Asserted for 1 cycle when operation is complete. Active-High.
cfg_mgmt_debug_access	Input	1	Type 1 RO, Write When the core is configured in the Root Port mode, asserting this input during a write to a Type-1 configuration space register forces a write into certain read-only fields of the register (see description of RC-mode Config registers). This input has no effect when the core is in the Endpoint mode, or when writing to any register other than a Type-1 configuration space register.

Configuration Status Interface

The Configuration Status interface provides information on how the core is configured, such as the negotiated link width and speed, the power state of the core, and configuration errors. The following table defines the ports in the Configuration Status interface of the core.

Table 24: Configuration Status Interface Port Descriptions

Port	Direction	Width	Description
cfg_phy_link_down	Output	1	<p>Configuration Link Down Status of the PCI Express link based on the Physical Layer LTSSM.</p> <ul style="list-style-type: none"> • 1b: Link is Down (LinkUp state variable is 0b) • 0b: Link is Up (LinkUp state variable is 1b) <p>Note: Per the PCI Express Base Specification, rev. 3.0, LinkUp is 1b in the Recovery, L0, L0s, L1, and L2 cfg_ltssm states. In the Configuration state, LinkUp can be 0b or 1b. It is always 0b when the Configuration state is reached using Detect > Polling > Configuration. LinkUp is 1b if the configuration state is reached through any other state transition.</p> <p>Note: While reset is asserted, the output of this signal are 0b until reset is released.</p>
cfg_phy_link_status	Output	2	<p>Configuration Link Status Status of the PCI Express link.</p> <ul style="list-style-type: none"> • 00b: No receivers detected • 01b: Link training in progress • 10b: Link up, DL initialization in progress • 11b: Link up, DL initialization completed
cfg_negotiated_width	Output	3	<p>Negotiated Link Width This output indicates the negotiated width of the given PCI Express Link and is valid when cfg_phy_link_status[1:0] == 11b (DL Initialization is complete).</p> <p>Negotiated Link Width values:</p> <ul style="list-style-type: none"> • 000b = x1 • 001b = x2 • 010b = x4 • 0011b = x8 • 100b = x16 • Other values are reserved.
cfg_current_speed	Output	2	<p>Current Link Speed This signal outputs the current link speed of the given PCI Express Link.</p> <ul style="list-style-type: none"> • 00b: 2.5 GT/s PCI Express Link Speed • 01b: 5.0 GT/s PCI Express Link Speed • 10b: 8.0 GT/s PCI Express Link Speed • 11b: Reserved
cfg_max_payload	Output	2	<p>Max_Payload_Size This signal outputs the maximum payload size from Device Control register bits 7 down to 5. This field sets the maximum TLP payload size. As a Receiver, the logic must handle TLPs as large as the set value. As a Transmitter, the logic must not generate TLPs exceeding the set value.</p> <ul style="list-style-type: none"> • 00b: 128 bytes maximum payload size • 01b: 256 bytes maximum payload size • 10b: 512 bytes maximum payload size • 11b: 1024 bytes maximum payload size

Table 24: Configuration Status Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
cfg_max_read_req	Output	3	<p>Max_Read_Request_Size</p> <p>This signal outputs the maximum read request size from Device Control register bits 14 down to 12. This field sets the maximum Read Request size for the logic as a Requester. The logic must not generate Read Requests with size exceeding the set value.</p> <ul style="list-style-type: none"> • 000b: 128 bytes maximum Read Request size • 001b: 256 bytes maximum Read Request size • 010b: 512 bytes maximum Read Request size • 011b: 1024 bytes maximum Read Request size • 100b: 2048 bytes maximum Read Request size • 101b: 4096 bytes maximum Read Request size • Other values are reserved
cfg_function_status	Output	16	<p>Configuration Function Status</p> <p>These outputs indicate the states of the Command register bits in the PCI configuration space of each function. These outputs are used to enable requests and completions from the host logic. The assignment of bits is as follows:</p> <ul style="list-style-type: none"> • Bit 0: Function 0 I/O Space Enable • Bit 1: Function 0 Memory Space Enable • Bit 2: Function 0 Bus Master Enable • Bit 3: Function 0 INTx Disable • Bit 4: Function 1 I/O Space Enable • Bit 5: Function 1 Memory Space Enable • Bit 6: Function 1 Bus Master Enable • Bit 7: Function 1 INTx Disable • Bit 8: Function 2 I/O Space Enable • Bit 9: Function 2 Memory Space Enable • Bit 10: Function 2 Bus Master Enable • Bit 11: Function 2 INTx Disable • Bit 12: Function 3 I/O Space Enable • Bit 13: Function 3 Memory Space Enable • Bit 14: Function 3 Bus Master Enable • Bit 15: Function 3 INTx Disable
cfg_vf_status	Output	504	<p>Configuration Virtual Function Status</p> <ul style="list-style-type: none"> • Bit 0: Virtual function 0: Configured/Enabled by the software. • Bit 1: Virtual function 0: PCI Command register, Bus Master Enable. • Bit 2: Virtual function 1: Configured/Enabled by software. • Bit 3: Virtual function 1: PCI Command register, Bus Master Enable.

Table 24: Configuration Status Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
cfg_function_power_state	Output	12	<p>Configuration Function Power State</p> <p>These outputs indicate the current power state of the physical functions. Bits [2:0] capture the power state of function 0, and bits [5:3] capture that of function 1, and so on. The possible power states are:</p> <ul style="list-style-type: none"> • 000: D0_uninitialized • 001: D0_active • 010: D1 • 100: D3_hot • Other values are reserved.
cfg_vf_power_state	Output	756	<p>Configuration Virtual Function Power State</p> <p>These outputs indicate the current power state of the virtual functions. Bits [2:0] capture the power state of virtual function 0, and bits [5:3] capture that of virtual function 1, and so on. The possible power states are:</p> <ul style="list-style-type: none"> • 000: D0_uninitialized • 001: D0_active • 010: D1 • 100: D3_hot • Other values are reserved.
cfg_link_power_state	Output	2	<p>Current power state of the PCI Express link, and is valid when cfg_phy_link_status[1:0] == 11b (DL Initialization is complete).</p> <ul style="list-style-type: none"> • 00: L0 • 01: TX L0s • 10: L1 • 11: L2/3 Ready

Table 24: Configuration Status Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
cfg_local_error_out	Output	5	<p>Local Error Conditions: Error priority is noted and Priority 0 has the highest priority.</p> <ul style="list-style-type: none"> • 00000b - Reserved • 00001b - Physical Layer Error Detected (Priority 16) • 00010b - Link Replay Timeout (Priority 12) • 00011b - Link Replay Rollover (Priority 13) • 00100b - Link Bad TLP Received (Priority 10) • 00101b - Link Bad DLLP Received (Priority 11) • 00110b - Link Protocol Error (Priority 9) • 00111b - Replay Buffer RAM Correctable ECC Error (Priority 22) • 01000b - Replay Buffer RAM Uncorrectable ECC Error (Priority 3) • 01001b - Receive Posted Request RAM Correctable ECC Error (Priority 20) • 01010b - Receive Posted Request RAM Uncorrectable ECC Error (Priority 1) • 01011b - Receive Completion RAM Correctable ECC Error (Priority 21) • 01100b - Receive Completion RAM Uncorrectable ECC Error (Priority 2) • 01101b - Receive Posted Buffer Overflow Error (Priority 5) • 01110b - Receive Non Posted Buffer Overflow Error (Priority 6) • 01111b - Receive Completion Buffer Overflow Error (Priority 7) • 10000b - Flow Control Protocol Error (Priority 8) • 10001b - Transmit Parity Error Detected (Priority 4) • 10010b - Unexpected Completion Received (Priority 15) • 10011b - Completion Timeout Detected (Priority 14) • 10100b - AXI4ST RQ INTFC Packet Drop (Priority 17) • 10101b - AXI4ST CC INTFC Packet Drop (Priority 18) • 10110b - AXI4ST CQ Poisoned Drop (Priority 19) • 10111b - User Signaled Internal Correctable Error (Priority 23) • 11000b - User Signaled Internal Uncorrectable Error (Priority 0) • 11001b - 11111b - Reserved
cfg_local_error_valid	Output	1	<p>Local Error Conditions Valid: Block activates this output for one cycle when any of the errors in cfg_local_error_out[4:0] are encountered. When driven 1b cfg_local_error_out[4:0] indicates local error type. Priority of error reporting (for the case of concurrent errors) is noted.</p> <p>Note: This signal may not work for all PCIe Link Width/Speed configurations. Do not rely solely on this signal to indicate an error. Alternatively, you can decode AER register to accurately detect errors.</p>

Table 24: Configuration Status Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
cfg_rx_pm_state	Output	2	<p>Current RX Active State Power Management L0s State: Encoding is listed below and valid when cfg_ltssm_state is indicating L0:</p> <ul style="list-style-type: none"> • RX_NOT_IN_L0s = 0 • RX_L0s_ENTRY = 1 • RX_L0s_IDLE = 2 • RX_L0s_FTS = 3
cfg_tx_pm_state	Output	2	<p>Current TX Active State Power Management L0s State: Encoding is listed below and valid when cfg_ltssm_state is indicating L0:</p> <ul style="list-style-type: none"> • TX_NOT_IN_L0s = 0 • TX_L0s_ENTRY = 1 • TX_L0s_IDLE = 2 • TX_L0s_FTS = 3

Table 24: Configuration Status Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
cfg_ltssm_state	Output	6	<p>LTSSM State. Shows the current LTSSM state:</p> <ul style="list-style-type: none"> • 00: Detect.Quiet • 01: Detect.Active • 02: Polling.Active • 03: Polling.Compliance • 04: Polling.Configuration • 05: Configuration.Linkwidth.Start • 06: Configuration.Linkwidth.Accept • 07: Configuration.Lanenum.Accept • 08: Configuration.Lanenum.Wait • 09: Configuration.Complete • 0A: Configuration.Idle • 0B: Recovery.RcvrLock • 0C: Recovery.Speed • 0D: Recovery.RcvrCfg • 0E: Recovery.Idle • 10: L0 • 11-16: Reserved • 17: L1.Entry • 18: L1.Idle • 19-1A: Reserved • 20: Disabled • 21: Loopback_Entry_Master • 22: Loopback_Active_Master • 23: Loopback_Exit_Master • 24: Loopback_Entry_Slave • 25: Loopback_Active_Slave • 26: Loopback_Exit_Slave • 27: Hot_Reset • 28: Recovery_Equalization_Phase0 • 29: Recovery_Equalization_Phase1 • 2a: Recovery_Equalization_Phase2 • 2b: Recovery_Equalization_Phase3
cfg_rcb_status	Output	4	<p>RCB Status.</p> <p>Provides the setting of the Read Completion Boundary (RCB) bit in the Link Control register of each physical function. In Endpoint mode, bit 0 indicates the RCB for Physical Function 0 (PF 0), bit 1 indicates the RCB for PF 1, and so on. In RC mode, bit 0 indicates the RCB setting of the Link Control register of the RP, bit 1 is reserved.</p> <p>For each bit, a value of 0 indicates an RCB of 64 bytes and a value of 1 indicates 128 bytes.</p>

Table 24: Configuration Status Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
cfg_dpa_substate_change	Output	4	<p>Dynamic Power Allocation Substate Change.</p> <p>In Endpoint mode, the core generates a one-cycle pulse on one of these outputs when a Configuration Write transaction writes into the Dynamic Power Allocation Control register to modify the DPA power state of the device. A pulse on bit 0 indicates such a DPA event for PF0 and a pulse on bit 1 indicates the same for PF1. The other 2 bits are reserved. These outputs are not active in Root Port mode.</p>
cfg_obff_enable	Output	2	<p>Optimized Buffer Flush Fill Enable.</p> <p>This output reflects the setting of the OBFF Enable field in the Device Control 2 register.</p> <ul style="list-style-type: none"> • 00: OBFF disabled. • 01: OBFF enabled using message signaling, Variation A. • 10: OBFF enabled using message signaling, Variation B. • 11: OBFF enabled using WAKE# signaling.
cfg_pl_status_change	Output	1	<p>This output is used by the core in Root Port mode to signal one of the following link training-related events:</p> <ul style="list-style-type: none"> • The link bandwidth changed as a result of the change in the link width or operating speed and the change was initiated locally (not by the link partner), without the link going down. This interrupt is enabled by the Link Bandwidth Management Interrupt Enable bit in the Link Control register. The status of this interrupt can be read from the Link Bandwidth Management Status bit of the Link Status register; or • The link bandwidth changed autonomously as a result of the change in the link width or operating speed and the change was initiated by the remote node. This interrupt is enabled by the Link Autonomous Bandwidth Interrupt Enable bit in the Link Control register. The status of this interrupt can be read from the Link Autonomous Bandwidth Status bit of the Link Status register; or • The Link Equalization Request bit in the Link Status 2 register was set by the hardware because it received a link equalization request from the remote node. This interrupt is enabled by the Link Equalization Interrupt Enable bit in the Link Control 3 register. The status of this interrupt can be read from the Link Equalization Request bit of the Link Status 2 register. <p>The pl_interrupt output is not active when the core is configured as an Endpoint.</p>
cfg_tph_requester_enable	Output	4	<p>Bit 0 of this output reflect the setting of the TPH Requester Enable bit [8] of the TPH Requester Control register in the TPH Requester Capability Structure of physical function 0. Bit 1 corresponds to physical function 1.</p>
cfg_tph_st_mode	Output	12	<p>Bits [2:0] of this output reflect the setting of the ST Mode Select bits in the TPH Requester Control register of physical function 0. Bits [5:3] reflect the setting of the same register field of PF 1.</p>
cfg_vf_tph_requester_enable	Output	252	<p>Each bit of this output reflects the setting of the TPH Requester Enable bit 8 of the TPH Requester Control register in the TPH Requester Capability Structure of the corresponding virtual function.</p>

Table 24: Configuration Status Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
cfg_vf_tph_st_mode	Output	756	<p>Bits [2:0] of this output reflect the setting of the ST Mode Select bits in the TPH Requester Control register of virtual function 0. Bits [5:3] reflect the setting of the same register field of VF 1, and so on.</p>
pcie_tfc_nph_av	Output	4	<p>This output provides an indication of the currently available header credit for Non-Posted TLPs on the transmit side of the core. The user logic can check this output before transmitting a Non-Posted request on the requester request interface, to avoid blocking the interface when no credit is available. The encodings are:</p> <ul style="list-style-type: none"> • 0000: No credit available • 0001: 1 credit available • 0010: 2 credits available • ... • 1110: 14 credits available • 1111: 15 or more credits available <p>Because of pipeline delays, the value on this output can not include the credit consumed by the Non-Posted requests in the last eight cycles or less. The user logic must adjust the value on this output by the credit consumed by the Non-Posted requests it sent in the previous clock cycles, if any.</p>
pcie_tfc_npd_av	Output	4	<p>This output provides an indication of the currently available payload credit for Non-Posted TLPs on the transmit side of the core. The user logic checks this output before transmitting a Non-Posted request on the requester request interface, to avoid blocking the interface when no credit is available. The encodings are:</p> <ul style="list-style-type: none"> • 0000: No credit available • 0001: 1 credit available • 0010: 2 credits available • ... • 1110: 14 or more credits available • 1111: 15 or more credits available <p>Because of pipeline delays, the value on this output does not include the credit consumed by the Non-Posted requests sent by the user logic in the last eight clock cycles or less. The user logic must adjust the value on this output by the credit consumed by the Non-Posted requests it sent in the previous clock cycles, if any.</p>

Table 24: Configuration Status Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
pcie_rq_tag_av	Output	4	<p>This output provides an indication of the number of free tags available for allocation to Non-Posted requests on the PCIe master side of the core. The user logic checks this output before transmitting a Non-Posted request on the requester request interface, to avoid blocking the interface when no tags are available. The encodings are:</p> <ul style="list-style-type: none"> • 0000: No tags available • 0001: 1 tag available • 0010: 2 tags available • ... • 1110: 14 tags available • 1111: 15 or more tags available <p>Because of pipeline delays, the value on this output does not include the tags consumed by the Non-Posted requests sent by the user logic in the last 8 clock cycles or less. The user logic must adjust the value on this output by the number of Non-Posted requests it sent in the previous clock cycles, if any.</p>

Configuration Received Message Interface

The Configuration Received Message interface indicates to the logic that a decodable message from the link, the parameters associated with the data, and the type of message have been received. The following table defines the ports in the Configuration Received Message interface of the core.

Table 25: Configuration Received Message Interface

Port	Direction	Width	Description
cfg_msg_received	Output	1	<p>Configuration Received a Decodable Message. The core asserts this output for one or more consecutive clock cycles when it has received a decodable message from the link. The duration of its assertion is determined by the type of message. The core transfers any parameters associated with the message on the cfg_msg_data[7:0] output in one or more cycles when cfg_msg_received is High. The following table lists the number of cycles of cfg_msg_received assertion, and the parameters transferred on cfg_msg_data[7:0] in each cycle, for each type of message.</p> <p>The core inserts at least a one-cycle gap between two consecutive messages delivered on this interface when the cfg_msg_received interface is enabled.</p> <p>The Configuration Received Message interface must be enabled during core configuration in the Vivado IDE.</p>
cfg_msg_received_data	Output	8	This bus is used to transfer any parameters associated with the Received Message. The information it carries in each cycle for various message types is listed in the previous table.

Table 25: Configuration Received Message Interface (cont'd)

Port	Direction	Width	Description
cfg_msg_received_type	Output	5	Received message type. When cfg_msg_received is High, these five bits indicate the type of message being signaled by the core. The various message types are listed in the previous table.

Table 26: Message Type Encoding on Receive Message Interface

cfg_msg_received_type[4:0]	Message Type
0	ERR_COR
1	ERR_NONFATAL
2	ERR_FATAL
3	Assert_INTA
4	Deassert_INTA
5	Assert_INTB
6	Deassert_INTB
7	Assert_INTC
8	Deassert_INTC
9	Assert_INTD
10	Deassert_INTD
11	PM_PME
12	PM_E_TO_Ack
13	PM_E_Turn_Off
14	PM_Active_State_Nak
15	Set_Slot_Power_Limit
16	Latency Tolerance Reporting (LTR)
17	Reserved
18	Unlock
19	Vendor Defined Type 0
20	Vendor Defined Type 1
25 – 31	Reserved

Table 27: Message Parameters on Receive Message Interface

Message Type	Number of cycles of cfg_msg_received assertion	Parameter transferred on cfg_msg_received_data[7:0]
ERR_COR, ERR_NONFATAL, ERR_FATAL	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
Assert_INTx, Deassert_INTx	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number

Table 27: Message Parameters on Receive Message Interface (cont'd)

Message Type	Number of cycles of cfg_msg_received assertion	Parameter transferred on cfg_msg_received_data[7:0]
PM_PME, PME_TO_Ack, PME_Turn_off, PM_Active_State_Nak	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
Set_Slot_Power_Limit	6	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number Cycle 3: bits [7:0] of payload Cycle 4: bits [15:8] of payload Cycle 5: bits [23:16] of payload Cycle 6: bits [31:24] of payload
Latency Tolerance Reporting (LTR)	6	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number Cycle 3: bits [7:0] of Snoop Latency Cycle 4: bits [15:8] of Snoop Latency Cycle 5: bits [7:0] of No-Snoop Latency Cycle 6: bits [15:8] of No-Snoop Latency
Unlock	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
Vendor_Defined Type 0	4 cycles when no data present, 8 cycles when data present.	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number Cycle 3: Vendor ID[7:0] Cycle 4: Vendor ID[15:8] Cycle 5: bits [7:0] of payload Cycle 6: bits [15:8] of payload Cycle 7: bits [23:16] of payload Cycle 8: bits [31:24] of payload
Vendor_Defined Type 1	4 cycles when no data present, 8 cycles when data present.	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number Cycle 3: Vendor ID[7:0] Cycle 4: Vendor ID[15:8] Cycle 5: bits [7:0] of payload Cycle 6: bits [15:8] of payload Cycle 7: bits [23:16] of payload Cycle 8: bits [31:24] of payload

Configuration Transmit Message Interface

The Configuration Transmit Message interface is used by the user application to transmit messages to the core. The user application supplies the transmit message type and data information to the core, which responds with the `done` signal. The following table defines the ports in the Configuration Transmit Message interface of the core.

Table 28: Configuration Transmit Message Interface

Port	Direction	Width	Description
cfg_msg_transmit	Input	1	Configuration Transmit Encoded Message. This signal is asserted together with cfg_msg_transmit_type, which supplies the encoded message type and cfg_msg_transmit_data, which supplies optional data associated with the message, until cfg_msg_transmit_done is asserted in response.
cfg_msg_transmit_type	Input	3	Configuration Transmit Encoded Message Type. Indicates the type of PCI Express message to be transmitted. Encodings supported are: <ul style="list-style-type: none">• 000b: Latency Tolerance Reporting (LTR)• 001b: Optimized Buffer Flush/Fill (OBFF)• 010b: Set Slot Power Limit (SSPL)• 011b: Power Management (PM PME)• 100b -111b: Reserved
cfg_msg_transmit_data	Input	32	Configuration Transmit Encoded Message Data. Indicates message data associated with particular message type. 000b: LTR - <ul style="list-style-type: none">• cfg_msg_transmit_data[31] < Snoop Latency Req• cfg_msg_transmit_data[28:26] < Snoop Latency Scale• cfg_msg_transmit_data[25:16] < Snoop Latency Value• cfg_msg_transmit_data[15] < No-Snoop Latency Requirement• cfg_msg_transmit_data[12:10] < No-Snoop Latency Scale• cfg_msg_transmit_data[9:0] < No-Snoop Latency Value 001b: OBFF - <ul style="list-style-type: none">• cfg_msg_transmit_data[3:0] < OBFF Code 010b: SSPL - <ul style="list-style-type: none">• cfg_msg_transmit_data[9:0] < {Slot Power Limit Scale, Slot Power Limit Value} 011b: PM_PME - <ul style="list-style-type: none">• cfg_msg_transmit_data[1:0] < PF1, PF0• cfg_msg_transmit_data[9:4] < VF5, VF4, VF3, VF2, VF1, VF0, where one or more PFs or VFs can signal PM_PME simultaneously. 100b - 111b: Reserved
cfg_msg_transmit_done	Output	1	Configuration Transmit Encoded Message Done. Asserted in response to cfg_msg_transmit assertion, for 1 cycle after the request is complete.

Configuration Flow Control Interface

The following table defines the ports in the Configuration Flow Control interface of the core.

Table 29: Configuration Flow Control Interface

Port	Direction	Width	Description
cfg_fc_ph	Output	8	<p>Posted Header Flow Control Credits.</p> <p>This output provides the number of Posted Header Flow Control Credits. This multiplexed output can be used to bring out various flow control parameters and variables related to Posted Header Credit maintained by the core. The flow control information to bring out on this core is selected by the cfg_fc_sel[2:0] input.</p>
cfg_fc_pd	Output	12	<p>Posted Data Flow Control Credits.</p> <p>This output provides the number of Posted Data Flow Control Credits. This multiplexed output can be used to bring out various flow control parameters and variables related to Posted Data Credit maintained by the core. The flow control information to bring out on this core is selected by the cfg_fc_sel[2:0] input.</p>
cfg_fc_nph	Output	8	<p>Non-Posted Header Flow Control Credits.</p> <p>This output provides the number of Non-Posted Header Flow Control Credits. This multiplexed output can be used to bring out various flow control parameters and variables related to Non-Posted Header Credit maintained by the core. The flow control information to bring out on this core is selected by the cfg_fc_sel[2:0] input.</p>
cfg_fc_npd	Output	12	<p>Non-Posted Data Flow Control Credits.</p> <p>This output provides the number of Non-Posted Data Flow Control Credits. This multiplexed output can be used to bring out various flow control parameters and variables related to Non-Posted Data Credit maintained by the core. The flow control information to bring out on this core is selected by the cfg_fc_sel[2:0] input.</p>
cfg_fc_cplh	Output	8	<p>Completion Header Flow Control Credits.</p> <p>This output provides the number of Completion Header Flow Control Credits. This multiplexed output can be used to bring out various flow control parameters and variables related to Completion Header Credit maintained by the core. The flow control information to bring out on this core is selected by the cfg_fc_sel[2:0] input.</p>
cfg_fc_cpld	Output	12	<p>Completion Data Flow Control Credits.</p> <p>This output provides the number of Completion Data Flow Control Credits. This multiplexed output can be used to bring out various flow control parameters and variables related to Completion Data Credit maintained by the core. The flow control information to bring out on this core is selected by the cfg_fc_sel[2:0].</p>

Table 29: Configuration Flow Control Interface (cont'd)

Port	Direction	Width	Description
cfg_fc_sel	Input	3	<p>Flow Control Informational Select.</p> <p>These inputs select the type of flow control to bring out on the cfg_fc_* outputs of the core. The various flow control parameters and variables that can be accessed for the different settings of these inputs are:</p> <ul style="list-style-type: none"> • 000: Reserved • 001: Reserved • 010: Receive credits consumed • 011: Reserved • 100: Transmit credits available • 101: Transmit credit limit • 110: Transmit credits consumed • 111: Reserved <p>This value represents the actual unused credits in the receiver FIFO, and the recommendation is to use it only as an approximate indication of receiver FIFO fullness, relative to the initial credit limit value advertized, such as, $\frac{1}{4}$ full, $\frac{1}{2}$ full, $\frac{3}{4}$ full, full.</p> <p>Infinite credit for transmit credits available (cfg_fc_sel == 3'b100) is signaled as 8'h80, 12'h800 for header and data credits, respectively. For all other cfg_fc_sel selections, infinite credit is signaled as 8'h00, 12'h000, respectively, for header and data categories.</p>

Configuration Control Interface

The Configuration Control interface signals allow a broad range of information exchange between the user application and the core. The user application uses this interface to do the following:

- Set the configuration space.
- Indicate if a correctable or uncorrectable error has occurred.
- Set the device serial number.
- Set the downstream bus, device, and function number.
- Receive per function configuration information.

This interface also provides handshaking between the user application and the core when a Power State change or function level reset occurs.

Table 30: Configuration Control Interface Port Descriptions

Port	Direction	Width	Description
cfg_hot_reset_in	Input	1	<p>Configuration Hot Reset In</p> <p>In RP mode, assertion transitions LTSSM to hot reset state, active-High.</p>

Table 30: Configuration Control Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
cfg_hot_reset_out	Output	1	Configuration Hot Reset Out In EP mode, assertion indicates that EP has transitioned to the hot reset state, active-High.
cfg_config_space_enable	Input	1	Configuration Configuration Space Enable When this input is set to 0 in the Endpoint mode, the core generates a CRS Completion in response to Configuration Requests. This port should be held deasserted when the core configuration registers are loaded from the DRP due to a change in attributes. This prevents the core from responding to Configuration Requests before all the registers are loaded. This input can be High when the power-on default values of the Configuration registers do not need to be modified before Configuration space enumeration. This input is not applicable for Root Port mode.
cfg_dsn	Input	64	Configuration Device Serial Number Indicates the value that should be transferred to the Device Serial Number Capability on PF0. Bits [31:0] are transferred to the first (Lower) Dword (byte offset 0x4h of the Capability), and bits [63:32] are transferred to the second (Upper) Dword (byte offset 0x8h of the Capability).
cfg_ds_bus_number	Input	8	Configuration Downstream Bus Number Downstream Port: Provides the bus number portion of the Requester ID (RID) of the Downstream Port. This is used in TLPs generated inside the core, such as UR Completions and Power-management messages; it does not affect TLPs presented on the AXI interface. Upstream Port: No role.
cfg_ds_device_number	Input	5	Configuration Downstream Device Number Downstream Port: Provides the device number portion of the RID of the Downstream Port. This is used in TLPs generated inside the core, such as UR Completions and Power-management messages; it does not affect TLPs presented on the TRN interface. Upstream Port: No role.
cfg_ds_function_number	Input	3	Configuration Downstream Function Number Downstream Port: Provides the function number portion of the RID of the Downstream Port. This is used in TLPs generated inside the core, such as UR Completions and power-management messages; it does not affect TLPs presented on the TRN interface. Upstream Port: No role.
cfg_power_state_change_ack	Input	1	Configuration Power State Ack You must assert this input to the core for one cycle in response to the assertion of cfg_power_state_change_interrupt, when it is ready to transition to the low-power state requested by the configuration write request. The user application can permanently hold this input High if it does not need to delay the return of the completions for the configuration write transactions, causing power-state changes.

Table 30: Configuration Control Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
cfg_power_state_change_interrupt	Output	1	<p>Power State Change Interrupt</p> <p>The core asserts this output when the power state of a physical or virtual function is being changed to the D1 or D3 states by a write into its Power Management Control register. The core holds this output High until the user application asserts the cfg_power_state_change_ack input to the core. While cfg_power_state_change_interrupt remains High, the core does not return completions for any pending configuration read or write transaction received by the core. The purpose is to delay the completion for the configuration write transaction that caused the state change until the user application is ready to transition to the low-power state. When cfg_power_state_change_interrupt is asserted, the function number associated with the configuration write transaction is provided on the cfg_ext_function_number[7:0] output. When the user application asserts cfg_power_state_change_ack, the new state of the function that underwent the state change is reflected on cfg_function_power_state (for PFs) or the cfg_vf_power_state (for VFs) outputs of the core.</p>
cfg_ds_port_number	Input	8	<p>Configuration Downstream Port Number</p> <p>Provides the port number field in the Link Capabilities register.</p>
cfg_err_cor_in	Input	1	<p>Correctable Error Detected</p> <p>The user application activates this input for one cycle to indicate a correctable error detected within the user logic that needs to be reported as an internal error through the PCI Express Advanced Error Reporting (AER) mechanism. In response, the core sets the Corrected Internal Error Status bit in the AER Correctable Error Status register of all enabled functions, and also sends an error message if enabled to do so. This error is not considered function-specific.</p>
cfg_err_cor_out	Output	1	<p>Correctable Error Detected</p> <p>In the Endpoint mode, the Block activates this output for one cycle when it has detected a correctable error and its reporting is not masked. When multiple functions are enabled, this is the logical OR of the correctable error status bits in the Device Status Registers of all functions.</p>
cfg_err_fatal_out	Output	1	<p>Fatal Error Detected</p> <p>In the Endpoint mode, the block activates this output for one cycle when it has detected a fatal error and its reporting is not masked. When multiple functions are enabled, this output is the logical OR of the fatal error status bits in the Device Status Registers of all functions.</p> <p>In the Root Port mode, this output is activated on detection of a local fatal error, when its reporting is not masked. This output does not respond to any errors signaled by remote devices using PCI Express error messages. These error messages are delivered to the user through the message interface.</p>

Table 30: Configuration Control Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
cfg_err_nonfatal_out	Output	1	<p>Non Fatal Error Detected</p> <p>In the Endpoint mode, the block activates this output for one cycle when it has detected a non fatal error and its reporting is not masked. When multiple functions are enabled, this output is the logical OR of the non fatal error status bits in the Device Status Registers of all functions.</p> <p>In the Root Port mode, this output is activated on detection of a local non fatal error, when its reporting is not masked. This output does not respond to any errors signaled by remote devices using PCI Express error messages. These error messages are delivered through the message interface.</p>
cfg_err_uncor_in	Input	1	<p>Uncorrectable Error Detected</p> <p>The user application activates this input for one cycle to indicate a uncorrectable error detected within the user logic that needs to be reported as an internal error through the PCI Express Advanced Error Reporting mechanism. In response, the core sets the uncorrected Internal Error Status bit in the AER Uncorrectable Error Status register of all enabled functions, and also sends an error message if enabled to do so. This error is not considered function-specific.</p>
cfg_flr_done	Input	4	<p>Function Level Reset Complete</p> <p>The user application must assert this input when it has completed the reset operation of the Virtual Function. This causes the core to deassert cfg_flr_in_process for physical function <i>i</i> and to re-enable configuration accesses to the physical function.</p>
cfg_vf_flr_done	Input	1	<p>Function Level Reset for Virtual Function is Complete</p> <p>The user application must assert this input when it has completed the reset operation of the Virtual Function. This causes the core to deassert cfg_vf_flr_in_process for function <i>i</i> and to re-enable configuration accesses to the virtual function.</p>
cfg_vf_flr_func_num	Input	8	<p>Function Level Reset for Virtual Function <i>i</i> is Complete.</p> <p>This user application drives a valid Virtual Function number on this input along with asserting cfg_vf_flr_done when the reset operation of Virtual Function <i>i</i> completes.</p> <p>Valid entries are 8'h04-8'hFF for VF0-VF251. Values 8'h00-8'h03 are reserved.</p>
cfg_flr_in_process	Output	4	<p>Function Level Reset In Process</p> <p>The core asserts bit <i>i</i> of this bus when the host initiates a reset of physical function <i>i</i> through its FLR bit in the configuration space. The core continues to hold the output High until the user sets the corresponding cfg_flr_done input for the corresponding physical function to indicate the completion of the reset operation.</p>

Table 30: Configuration Control Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
cfg_vf_flr_in_process	Output	252	Function Level Reset In Process for Virtual Function The core asserts bit <i>i</i> of this bus when the host initiates a reset of virtual function <i>i</i> though its FLR bit in the configuration space. The core continues to hold the output High until the user sets the cfg_vf_flr_done input and drives cfg_vf_flr_func_num with the corresponding function to indicate the completion of the reset operation.
cfg_req_pm_transition_l23_ready	Input	1	When the core is configured as an Endpoint, the user application asserts this input to transition the power management state of the core to L23_READY (see Chapter 5 of the PCI Express Specification (see <i>PCI-SIG Specifications</i> (http://www.pcisig.com/specifications) for a detailed description of power management). This is done after the PCI functions in the core are placed in the D3 state and after the user application acknowledges the PME_Turn_Off message from the Root Complex. Asserting this input causes the link to transition to the L3 state, and requires a hard reset to resume operation. This input can be hardwired to 0 if the link is not required to transition to L3. This input is not used in Root Complex mode.
cfg_link_training_enable	Input	1	This input must be set to 1 to enable the Link Training Status State Machine (LTSSM) to bring up the link. Setting it to 0 forces the LTSSM to stay in the Detect.Quiet state.
cfg_bus_number	Output	8	Bus Number Captured from received CfgWr Type0 is presented. Active only in the Endpoint Configuration.
cfg_vend_id	Input	16	Configuration Vendor ID: Indicates the value that should be transferred to the PCI Capability Structure Vendor ID field on all PFs.
cfg_subsys_vend_id	Input	16	Configuration Subsystem Vendor ID: Indicates the value that should be transferred to the Type 0 PCI Capability Structure Subsystem Vendor ID field on all PFs.
cfg_dev_id_pf0	Input	16	Configuration Device ID PF0: Indicates the value that should be transferred to the PCI Capability Structure Device ID field on PF0.
cfg_dev_id_pf1	Input	16	Configuration Device ID PF1: Indicates the value that should be transferred to the PCI Capability Structure Device ID field on PF1.
cfg_dev_id_pf2	Input	16	Configuration Device ID PF2: Indicates the value that should be transferred to the PCI Capability Structure Device ID field on PF2.
cfg_dev_id_pf3	Input	16	Configuration Device ID PF3: Indicates the value that should be transferred to the PCI Capability Structure Device ID field on PF3.
cfg_rev_id_pf0	Input	8	Configuration Revision ID PF0: Indicates the value that should be transferred to the PCI Capability Structure Revision ID field on PF0.
cfg_rev_id_pf1	Input	8	Configuration Revision ID PF1: Indicates the value that should be transferred to the PCI Capability Structure Revision ID field on PF1.

Table 30: Configuration Control Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
cfg_rev_id_pf2	Input	8	Configuration Revision ID PF2: Indicates the value that should be transferred to the PCI Capability Structure Revision ID field on PF2.
cfg_rev_id_pf3	Input	8	Configuration Revision ID PF3: Indicates the value that should be transferred to the PCI Capability Structure Revision ID field on PF3.
cfg_subsys_id_pf0	Input	16	Configuration Subsystem ID PF0: Indicates the value that should be transferred to the Type 0 PCI Capability Structure Subsystem ID field on PF0.
cfg_subsys_id_pf1	Input	16	Configuration Subsystem ID PF1: Indicates the value that should be transferred to the Type 0 PCI Capability Structure Subsystem ID field on PF1.
cfg_subsys_id_pf2	Input	16	Configuration Subsystem ID PF2: Indicates the value that should be transferred to the Type 0 PCI Capability Structure Subsystem ID field on PF2.
cfg_subsys_id_pf3	Input	16	Configuration Subsystem ID PF3: Indicates the value that should be transferred to the Type 0 PCI Capability Structure Subsystem ID field on PF3.

Configuration Interrupt Controller Interface

The Configuration Interrupt Controller interface allows the user application to set Legacy PCIe interrupts, MSI interrupts, or MSI-X interrupts. The core provides the interrupt status on the configuration interrupt sent and fail signals. The following tables define the interface ports associated with the Configuration Interrupt Controller interface of the core.

Legacy Interrupt Interface

Table 31: Legacy Interrupt Interface Port Descriptions

Name	Direction	Width	Description
cfg_interrupt_int	Input	4	Configuration INTx Vector: When the core is configured as EP, these four inputs are used by the user application to signal an interrupt from any of its PCI Functions to the RC using the Legacy PCI Express Interrupt Delivery mechanism of PCI Express. These four inputs correspond to INTA, INTB, INTC, and INTD of the PCI bus, respectively. Asserting one of these signals causes the core to send out an Assert_INTx message, and deasserting the signal causes the core to transmit a Deassert_INTx message.
cfg_interrupt_sent	Output	1	Configuration INTx Sent: A pulse on this output indicates that the core has sent an INTx Assert or Deassert message in response to a change in the state of one of the cfg_interrupt_int inputs.

Table 31: Legacy Interrupt Interface Port Descriptions (cont'd)

Name	Direction	Width	Description
cfg_interrupt_pending	Input	4	Configuration INTx Interrupt Pending: Per Function indication of a pending interrupt from the user. cfg_interrupt_pending[0] corresponds to Function #0. Each of these inputs is connected to the Interrupt Pending bits of the PCI Status Register of the corresponding Function.

MSI Interrupt Interface

Table 32: MSI Interrupt Interface Port Descriptions

Name	Direction	Width	Description
cfg_interrupt_msi_enable	Output	4	Configuration Interrupt MSI Function Enabled Indicates that the Message Signaling Interrupt (MSI) messaging is enabled, per Function. These outputs reflect the setting of the MSI Enable bits in the MSI Control Register of Physical Functions 0 – 3.
cfg_interrupt_msi_int	Input	32	Configuration Interrupt MSI/MSI-X Vector When configured in the Endpoint mode to support MSI interrupts, these inputs are used to signal the 32 distinct interrupt conditions associated with a PCI Function (Physical or Virtual) from the user logic to the core. The Function number must be specified on the input cfg_interrupt_msi_function_number. After placing the Function number on the input cfg_interrupt_msi_function_number, the user logic must activate one of these signals for one cycle to transmit an interrupt. The user logic must not activate more than one of the 32 interrupt inputs in the same cycle. The core internally registers the interrupt condition on the 0-to-1 transition of any bit in cfg_interrupt_msi_int. After asserting an interrupt, the user logic must wait for the cfg_interrupt_msi_sent or cfg_interrupt_msi_fail indication from the core before asserting a new interrupt.
cfg_interrupt_msi_function_number	Input	8	Configuration MSI/MSI-X Initiating Function Indicates the Endpoint Function # initiating the MSI or MSI-X interrupt. <ul style="list-style-type: none"> • 8'h00 – 8'h03: PF 0 – PF 3 • 8'h04 – 8'hFF: VF 0 – VF 252 • Other encodings are reserved.
cfg_interrupt_msi_sent	Output	1	Configuration Interrupt MSI/MSI-X Interrupt Sent The core generates a one-cycle pulse on this output to signal that an MSI or MSI-X interrupt message has been transmitted on the link. The user logic must wait for this pulse before signaling another interrupt condition to the core.
cfg_interrupt_msi_fail	Output	1	Configuration Interrupt MSI/MSI-X Interrupt Operation Failed A one-cycle pulse on this output indicates that an MSI or MSI-X interrupt message was aborted before transmission on the link. The user logic must retransmit the MSI or MSI-X interrupt in this case.

Table 32: MSI Interrupt Interface Port Descriptions (cont'd)

Name	Direction	Width	Description
cfg_interrupt_msi_mmenable	Output	12	<p>Configuration Interrupt MSI Function Multiple Message Enable</p> <p>When the core is configured in the Endpoint mode to support MSI interrupts, these outputs are driven by the 'Multiple Message Enable' bits of the MSI Control Register associated with Physical Functions. These bits encode the number of allocated MSI interrupt vectors for the corresponding Function. Bits [2:0] correspond to Physical Function 0, bits [5:3] correspond to PF 1, and so on. The valid encodings of the 3 bits are:</p> <ul style="list-style-type: none"> • 000b: 1 vector • 001b: 2 vectors • 010b: 4 vectors • 011b: 8 vectors • 100b: 16 vectors • 101b: 32 vectors
cfg_interrupt_msi_pending_status	Input	32	<p>Configuration MSI Interrupt Pending Status</p> <p>These inputs are provided for the user to indicate the interrupt pending status of the MSI interrupts associated with the Physical Functions. When the status of a MSI interrupt associated with a PF changes, the user must place the new interrupt status on these inputs, along with the corresponding Function number on the cfg_interrupt_msi_pending_status_function_num input, and activate the cfg_interrupt_msi_pending_status_data_enable input for one cycle. The core then latches the new status in its MSI Pending Bits Register of the corresponding Physical Function.</p>
cfg_interrupt_msi_pending_status_function_num	Input	2	<p>Configuration Interrupt MSI Pending Target Function Number</p> <ul style="list-style-type: none"> • 00 = PF 0 • 01 = PF 1 • 10 = PF 2 • 11 = PF 3 <p>This input is used to identify the Function number when the user places interrupt status on the cfg_interrupt_msi_pending_status inputs.</p>
cfg_interrupt_msi_pending_status_data_enable	Input	1	<p>Configuration Interrupt MSI Pending Data Valid</p> <p>The user application asserts this signal together with cfg_interrupt_msi_pending_status and cfg_interrupt_msi_pending_status_function_num values to update the MSI Pending Bits in the corresponding function.</p>
cfg_interrupt_msi_mask_update	Output	1	<p>Configuration Interrupt MSI Function Mask Updated</p> <p>The SR-IoV core asserts this for 1 cycle when the MSI Mask Register of any enabled PFs has changed its value. The user can then read the new mask settings from the cfg_interrupt_msi_data outputs.</p>
cfg_interrupt_msi_select	Input	2	<p>Configuration Interrupt MSI Select</p> <p>These inputs are used to select the Function number for reading the MSI Mask Register setting from the core. Values 0 – 3 correspond to Physical Functions 0 – 3, respectively. The mask MSI Mask Register contents of the selected PF appear on the output cfg_interrupt_msi_data after one cycle.</p>

Table 32: MSI Interrupt Interface Port Descriptions (cont'd)

Name	Direction	Width	Description
cfg_interrupt_msi_data	Output	32	Configuration Interrupt MSI Data These output reflect the MSI Mask Register setting of the Physical Function selected by the cfg_interrupt_msi_select input.
cfg_interrupt_msi_attr	Input	3	Configuration Interrupt MSI/MSI-X TLP Attribute These bits enable you to set the Attribute bits that are used for both MSI and MSI-X interrupt requests. <ul style="list-style-type: none"> • Bit 0 is the No Snoop bit. • Bit 1 is the Relaxed Ordering bit. • Bit 2 is the ID-Based Ordering bit. The core samples these bits on a 0-to-1 transition on cfg_interrupt_msi_int bits (when using MSI) or cfg_interrupt_msix_int (when using MSI-X).
cfg_interrupt_msi_tph_present	Input	1	Configuration Interrupt MSI/MSI-X TPH Present Indicates the presence of an optional Transaction Processing Hint (TPH) in the MSI/MSI-X interrupt request. The user application must set this bit while asserting cfg_interrupt_msi_int bits (when using MSI), or cfg_interrupt_msix_int (when using MSI-X), if it is including a TPH in the MSI or MSI-X transaction.
cfg_interrupt_msi_tph_type	Input	2	Configuration Interrupt MSI/MSI-X TPH Type When cfg_interrupt_msi_tph_present is 1'b1, these two bits are used to supply the 2-bit type associated with the Hint. The core samples these bits on 0-to-1 transition on any bit of cfg_interrupt_msi_int or cfg_interrupt_msix_int, depending on whether MSI or MSI-X interrupts are being used.
cfg_interrupt_msi_tph_st_tag	Input	8	Configuration Interrupt MSI/MSI-X TPH Steering Tag When cfg_interrupt_msi_tph_present is asserted, the Steering Tag associated with the Hint must be placed on cfg_interrupt_msi_tph_st_tag[7:0]. The core samples these bits on 0-to-1 transition on any bit of cfg_interrupt_msi_int or cfg_interrupt_msix_int, depending on whether MSI or MSI-X interrupts are being used.

MSI-X Interrupt External Interface

Table 33: MSI-X Interrupt External Interface Port Descriptions

Name	Direction	Width	Description
cfg_interrupt_msix_enable	Output	4	Configuration Interrupt MSI-X Function Enabled These outputs reflect the setting of the MSI-X Enable bits of the MSI-X Control Register of Physical Functions 0 – 3.
cfg_interrupt_msix_mask	Output	4	Configuration Interrupt MSI-X Function Mask These outputs reflect the setting of the MSI-X Function Mask bits of the MSI-X Control Register of Physical Functions 0 – 3.
cfg_interrupt_msix_vf_enable	Output	252	Configuration Interrupt MSI-X Enable from VFs These outputs reflect the setting of the MSI-X Enable bits of the MSI-X Control Register of Virtual Functions 0 – 251.

Table 33: MSI-X Interrupt External Interface Port Descriptions (cont'd)

Name	Direction	Width	Description
cfg_interrupt_msix_vf_mask	Outputs	252	Configuration Interrupt MSI-X VF Mask These outputs reflect the setting of the MSI-X Function Mask bits of the MSI-X Control Register of Virtual Functions 0 – 251.
cfg_interrupt_msix_address	Input	64	Configuration Interrupt MSI-X Address When the core is configured to support MSI-X interrupts and when the MSI-X Table is implemented in user memory, this bus is used by the user logic to communicate the address to be used to generate an MSI-X interrupt.
cfg_interrupt_msix_data	Input	32	Configuration Interrupt MSI-X Data When the core is configured to support MSI-X interrupts and when the MSI-X Table is implemented in user memory, this bus is used by the user logic to communicate the data to be used to generate an MSI-X interrupt.
cfg_interrupt_msix_int	Input	1	Configuration Interrupt MSI-X Data Valid The assertion of this signal by the user indicates a request from the user to send an MSI-X interrupt. The user must place the identifying information on the designated inputs before asserting this interrupt. When the MSI-X Table and Pending Bit Array are implemented in user memory, the identifying information consists of the memory address, data, and the originating Function number for the interrupt. These must be placed on the cfg_interrupt_msix_address[63:0], cfg_interrupt_msix_data[31:0], and cfg_interrupt_msi_function_number[7:0], respectively. The core internally registers these parameters on the 0-to-1 transition of cfg_interrupt_msix_int. When the MSI-X Table and Pending Bit Array are implemented by the core, the identifying information consists of the originating Function number for the interrupt and the interrupt vector. These must be placed on cfg_interrupt_msi_function_number[7:0] and cfg_interrupt_msi_int[31:0], respectively. Bit i of cfg_interrupt_msi_int[31:0] represents interrupt vector i, and only one of the bits of this bus can be set to 1 when asserting cfg_interrupt_msix_int. After asserting an interrupt, the user logic must wait for the cfg_interrupt_msi_sent or cfg_interrupt_msi_fail indication from the core before asserting a new interrupt.

Table 33: MSI-X Interrupt External Interface Port Descriptions (cont'd)

Name	Direction	Width	Description
cfg_interrupt_msix_vec_pending	Input	2	<p>Configuration Interrupt MSI-X Pending Bit Query/Clear These mode bits are used only when the core is configured to include the MSI-X Table and Pending Bit Array. These two bits are set when asserting cfg_interrupt_msix_int to send an MSI-X interrupt, to perform certain actions on the MSI-X Pending Bit associated with the selected Function and interrupt vector. The various modes are:</p> <ul style="list-style-type: none"> • 00b: Normal interrupt generation. If the Mask bit associated with the vector was 0 when cfg_interrupt_msix_int was asserted, the core transmits the MSI-X request TLP on the link. If the Mask bit was 1, the core does not immediately send the interrupt, but instead sets the Pending Bit associated with the interrupt vector in its MSI-X Pending Bit Array (and subsequently transmits the MSI-X request TLP when the Mask clears). In both cases, the core asserts cfg_interrupt_msi_sent for one cycle to indicate that the interrupt request was accepted. The user can distinguish these two cases by sampling the cfg_interrupt_msix_vec_pending_status output, which reflects the current setting of the MSI-X Pending Bit corresponding to the interrupt vector. • 01b: Pending Bit Query. In this mode, the core treats the assertion of one of the bits of cfg_interrupt_msix_int as a query for the status of its Pending Bit. The user must also place the Function number of the Pending Bit being queried on the cfg_interrupt_msi_function_number input. The core does not transmit a MSI-X request in response, but asserts cfg_interrupt_msi_sent for one cycle, along with the status of the MSI-X Pending Bit on the cfg_interrupt_msix_vec_pending_status output. • 10b: Pending Bit Clear. In this mode, the core treats the assertion of one of the bits of cfg_interrupt_msix_int as a request to clear its Pending Bit. The user must also place the Function number of the Pending Bit being queried on the cfg_interrupt_msi_function_number input. The core does not transmit a MSI-X request in response, but clears the MSI-X Pending Bit of the vector (if it is set), and activates cfg_interrupt_msi_sent for one cycle as the acknowledgment. The core also provides the previous state of the MSI-X Pending Bit on the cfg_interrupt_msix_vec_pending_status output, which can be sampled by the user to determine if the Pending Bit was cleared by the core before the user request (because the pending interrupt was actually transmitted). This mode can be used to implement a “polling mode” for MSI-X interrupts, where the interrupt is permanently masked and the software polls the Pending Bit to detect and service the interrupt. After each interrupt is serviced, the Pending Bit can be cleared through this interface.
cfg_interrupt_msix_vec_pending_status	Output	1	<p>Configuration Interrupt MSI-X Pending Bit Status This output provides the status of the Pending Bit associated with an MSI-X interrupt, in response to query using the cfg_interrupt_msix_vec_pending input. It is active only when the core is configured to include the MSI-X Table and Pending Bit Array.</p>

MSI-X Interrupt Internal Interface

Table 34: MSI-X Interrupt Internal Interface Port Descriptions

Name	Direction	Width	Description
cfg_interrupt_msi_int	Input	8	See cfg_interrupt_msi_int . The core supports eight vectors per function and it is one-hot encoding, so each bit corresponds to one vector.
cfg_interrupt_msi_function_number	Input	8	See cfg_interrupt_msi_function_number .
cfg_interrupt_msi_attr	Input	3	See cfg_interrupt_msi_attr .
cfg_interrupt_msi_tph_present	Input	1	See cfg_interrupt_msi_tph_present .
cfg_interrupt_msi_tph_type	Input	2	See cfg_interrupt_msi_tph_type .
cfg_interrupt_msi_tph_st_tag	Input	8	See cfg_interrupt_msi_tph_st_tag .
cfg_interrupt_msi_sent	Output	1	See cfg_interrupt_msi_sent .
cfg_interrupt_msi_fail	Output	1	See cfg_interrupt_msi_fail .
cfg_interrupt_msix_int	Input	1	See cfg_interrupt_msix_int .
cfg_interrupt_msix_vec_pending	Input	2	See cfg_interrupt_msix_vec_pending .
cfg_interrupt_msix_vec_pending_status	Output	1	See cfg_interrupt_msix_vec_pending_status .
cfg_interrupt_msix_enable	Output	4	See cfg_interrupt_msix_enable .
cfg_interrupt_msix_mask	Output	4	See cfg_interrupt_msix_mask .
cfg_interrupt_msix_vf_enable	Output	252	See cfg_interrupt_msix_vf_enable .
cfg_interrupt_msix_vf_mask	Output	252	See cfg_interrupt_msix_vf_mask .

Configuration Extend Interface

The Configuration Extend interface allows the core to transfer configuration information with the user application when externally implemented configuration registers are implemented. The following table defines the ports in the Configuration Extend interface of the core.

Table 35: Configuration Extend Interface Port Descriptions

Port	Direction	Width	Description
cfg_ext_read_received	Output	1	<p>Configuration Extend Read Received. The Block asserts this output when it has received a configuration read request from the link.</p> <p>Set when PCI Express Extended Configuration Space Enable is selected in User Defined Configuration Capabilities in core configuration in the Vivado IDE.</p> <p>All received configuration reads with cfg_ext_register_number in the following ranges is considered to be the PCIe Extended Configuration Space.</p> <ul style="list-style-type: none"> UltraScale+™ PCIe4 core: 0x480-0x4FF UltraScale+ HBM PCIe4C core: 0xE80-0xFFFF <p>All received configuration reads regardless of its address will be indicated by 1 cycle assertion of cfg_ext_read_received and valid data is driven on cfg_ext_register_number and cfg_ext_function_number. Only received configuration reads within the aforementioned ranges need to be responded by User Application outside of the IP.</p>
cfg_ext_write_received	Output	1	<p>Configuration Extend Write Received. The Block asserts this output when it has received a configuration write request from the link.</p> <p>Set when PCI Express Extended Configuration Space Enable is selected in User Defined Configuration Capabilities in the core configuration in the Vivado IDE.</p> <p>Data corresponding to all received configuration writes with cfg_ext_register_number in the range 0xb0-0xbf is presented on cfg_ext_register_number, cfg_ext_function_number, cfg_ext_write_data and cfg_ext_write_byte_enable.</p> <p>All received configuration writes with cfg_ext_register_number in the following ranges are presented on cfg_ext_register_number, cfg_ext_function_number, cfg_ext_wrte_data and cfg_ext_write_byte_enable.</p> <ul style="list-style-type: none"> UltraScale+ PCIe4 core: 0x480-0x4FF UltraScale+ HBM PCIe4C core: 0xE80-0xFFFF
cfg_ext_register_number	Output	10	<p>Configuration Extend Register Number The 10-bit address of the configuration register being read or written. The data is valid when cfg_ext_read_received or cfg_ext_write_received is High.</p>
cfg_ext_function_number	Output	8	<p>Configuration Extend Function Number The 8-bit function number corresponding to the configuration read or write request. The data is valid when cfg_ext_read_received or cfg_ext_write_received is High.</p>
cfg_ext_write_data	Output	32	<p>Configuration Extend Write Data Data being written into a configuration register. This output is valid when cfg_ext_write_received is High.</p>
cfg_ext_write_byte_enable	Output	4	<p>Configuration Extend Write Byte Enable Byte enables for a configuration write transaction.</p>

Table 35: Configuration Extend Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
cfg_ext_read_data	Input	32	Configuration Extend Read Data You can provide data from an externally implemented configuration register to the core through this bus. The core samples this data on the next positive edge of the clock after it sets cfg_ext_read_received High, if you have set cfg_ext_read_data_valid.
cfg_ext_read_data_valid	Input	1	Configuration Extend Read Data Valid The user application asserts this input to the core to supply data from an externally implemented configuration register. The core samples this input data on the next positive edge of the clock after it sets cfg_ext_read_received High. The core expects the assertions of this signal within 262144 ('h4_0000) clock cycles of user clock after receiving the read request on cfg_ext_read_received signal. If no response is received by this time, the core will send auto-response with 'h0 payload, and the user application must discard the response and terminate that particular request immediately.

Clock and Reset Interface

Fundamental to the operation of the core, the Clock and Reset interface provides the system-level clock and reset to the core as well as the user application clock and reset signal. The table below defines the ports in the Clock and Reset interface of the core.

The `user_clk` signal is the derived clock from the TXOUTCLK pin which is the output from the GT Wizard IP. TXOUTCLK is dependent on the `pmareset`, `progdivreset`, and `txpisopd` signals, and also on `sys_clk` or `refclk` which is connected to GT Wizard IP. So, `user_clk` is not expected to run continuously. For more details about TXOUTCLK, refer the corresponding GT Wizard documents.

Table 36: Clock and Reset Interface Port Descriptions

Port	Direction	Width	Description
user_clk	Output	1	User clock output (62.5, 125, or 250 MHz) This clock has a fixed frequency and is configured in the Vivado® Integrated Design Environment (IDE).
user_reset	Output	1	This signal is deasserted synchronously with respect to user_clk. It is deasserted and asserted asynchronously with sys_reset assertion.
sys_clk	Input	1	Reference clock This clock has a selectable frequency of 100 MHz.
sys_clk_gt	Input	1	PCIe reference clock for GT. This clock must be driven directly from IBUFS (same definition and frequency as sys_clk). This clock has a selectable frequency of 100 MHz, which is the same as in sys_clk.
sys_reset	Input	1	Fundamental reset input to the core (asynchronous) This input is active-Low by default to match the PCIe edge connector reset polarity.

Table 36: Clock and Reset Interface Port Descriptions (cont'd)

Port	Direction	Width	Description
phy_rdy_out	Output	1	The phy ready signal indicates that the GT Wizard is ready. This signal is driven by phy_RST FSM on receiving the phy status from the GT Wizard core.

The UltraScale+ integrated block does not support dedicated reset pin routing. You need to carefully update your design while upgrading from UltraScale device to UltraScale+ device.

PCI Express Interface

The PCI Express (PCI_EXP) interface consists of differential transmit and receive pairs organized in multiple lanes. A PCI Express lane consists of a pair of transmit differential signals {pci_exp_txp, pci_exp_txn} and a pair of receive differential signals {pci_exp_rxp, pci_exp_rxn}. The 1-lane core supports only Lane 0, the 2-lane core supports lanes 0-1, the 4-lane core supports lanes 0-3, the 8-lane core supports lanes 0-7, and the 16-lane core supports lanes 0-15. Transmit and receive signals of the PCI_EXP interface are defined in the following table.

Table 37: PCI Express Interface Signals for 1-, 2-, 4-, 8- and 16-Lane Cores

Lane Number	Name	Direction	Description
1-Lane Cores			
0	pci_exp_txp0	Output	PCI Express Transmit Positive: Serial Differential Output 0 (+)
	pci_exp_txn0	Output	PCI Express Transmit Negative: Serial Differential Output 0 (-)
	pci_exp_rxp0	Input	PCI Express Receive Positive: Serial Differential Input 0 (+)
	pci_exp_rxn0	Input	PCI Express Receive Negative: Serial Differential Input 0 (-)
2-Lane Cores			
0	pci_exp_txp0	Output	PCI Express Transmit Positive: Serial Differential Output 0 (+)
	pci_exp_txn0	Output	PCI Express Transmit Negative: Serial Differential Output 0 (-)
	pci_exp_rxp0	Input	PCI Express Receive Positive: Serial Differential Input 0 (+)
	pci_exp_rxn0	Input	PCI Express Receive Negative: Serial Differential Input 0 (-)
1	pci_exp_txp1	Output	PCI Express Transmit Positive: Serial Differential Output 1 (+)
	pci_exp_txn1	Output	PCI Express Transmit Negative: Serial Differential Output 1 (-)
	pci_exp_rxp1	Input	PCI Express Receive Positive: Serial Differential Input 1 (+)
	pci_exp_rxn1	Input	PCI Express Receive Negative: Serial Differential Input 1 (-)
4-Lane Cores			
0	pci_exp_txp0	Output	PCI Express Transmit Positive: Serial Differential Output 0 (+)
	pci_exp_txn0	Output	PCI Express Transmit Negative: Serial Differential Output 0 (-)
	pci_exp_rxp0	Input	PCI Express Receive Positive: Serial Differential Input 0 (+)
	pci_exp_rxn0	Input	PCI Express Receive Negative: Serial Differential Input 0 (-)

Table 37: PCI Express Interface Signals for 1-, 2-, 4-, 8- and 16-Lane Cores (cont'd)

Lane Number	Name	Direction	Description
1	pci_exp_txp1	Output	PCI Express Transmit Positive: Serial Differential Output 1 (+)
	pci_exp_tnx1	Output	PCI Express Transmit Negative: Serial Differential Output 1 (-)
	pci_exp_rxp1	Input	PCI Express Receive Positive: Serial Differential Input 1 (+)
	pci_exp_rxn1	Input	PCI Express Receive Negative: Serial Differential Input 1 (-)
2	pci_exp_txp2	Output	PCI Express Transmit Positive: Serial Differential Output 2 (+)
	pci_exp_tnx2	Output	PCI Express Transmit Negative: Serial Differential Output 2 (-)
	pci_exp_rxp2	Input	PCI Express Receive Positive: Serial Differential Input 2 (+)
	pci_exp_rxn2	Input	PCI Express Receive Negative: Serial Differential Input 2 (-)
3	pci_exp_txp3	Output	PCI Express Transmit Positive: Serial Differential Output 3 (+)
	pci_exp_tnx3	Output	PCI Express Transmit Negative: Serial Differential Output 3 (-)
	pci_exp_rxp3	Input	PCI Express Receive Positive: Serial Differential Input 3 (+)
	pci_exp_rxn3	Input	PCI Express Receive Negative: Serial Differential Input 3 (-)
8-Lane Cores			
0	pci_exp_txp0	Output	PCI Express Transmit Positive: Serial Differential Output 0 (+)
	pci_exp_tnx0	Output	PCI Express Transmit Negative: Serial Differential Output 0 (-)
	pci_exp_rxp0	Input	PCI Express Receive Positive: Serial Differential Input 0 (+)
	pci_exp_rxn0	Input	PCI Express Receive Negative: Serial Differential Input 0 (-)
1	pci_exp_txp1	Output	PCI Express Transmit Positive: Serial Differential Output 1 (+)
	pci_exp_tnx1	Output	PCI Express Transmit Negative: Serial Differential Output 1 (-)
	pci_exp_rxp1	Input	PCI Express Receive Positive: Serial Differential Input 1 (+)
	pci_exp_rxn1	Input	PCI Express Receive Negative: Serial Differential Input 1 (-)
2	pci_exp_txp2	Output	PCI Express Transmit Positive: Serial Differential Output 2 (+)
	pci_exp_tnx2	Output	PCI Express Transmit Negative: Serial Differential Output 2 (-)
	pci_exp_rxp2	Input	PCI Express Receive Positive: Serial Differential Input 2 (+)
	pci_exp_rxn2	Input	PCI Express Receive Negative: Serial Differential Input 2 (-)
3	pci_exp_txp3	Output	PCI Express Transmit Positive: Serial Differential Output 3 (+)
	pci_exp_tnx3	Output	PCI Express Transmit Negative: Serial Differential Output 3 (-)
	pci_exp_rxp3	Input	PCI Express Receive Positive: Serial Differential Input 3 (+)
	pci_exp_rxn3	Input	PCI Express Receive Negative: Serial Differential Input 3 (-)
4	pci_exp_txp4	Output	PCI Express Transmit Positive: Serial Differential Output 4 (+)
	pci_exp_tnx4	Output	PCI Express Transmit Negative: Serial Differential Output 4 (-)
	pci_exp_rxp4	Input	PCI Express Receive Positive: Serial Differential Input 4 (+)
	pci_exp_rxn4	Input	PCI Express Receive Negative: Serial Differential Input 4 (-)
5	pci_exp_txp5	Output	PCI Express Transmit Positive: Serial Differential Output 5 (+)
	pci_exp_tnx5	Output	PCI Express Transmit Negative: Serial Differential Output 5 (-)
	pci_exp_rxp5	Input	PCI Express Receive Positive: Serial Differential Input 5 (+)
	pci_exp_rxn5	Input	PCI Express Receive Negative: Serial Differential Input 5 (-)

Table 37: PCI Express Interface Signals for 1-, 2-, 4-, 8- and 16-Lane Cores (cont'd)

Lane Number	Name	Direction	Description
6	pci_exp_txp6	Output	PCI Express Transmit Positive: Serial Differential Output 6 (+)
	pci_exp_txn6	Output	PCI Express Transmit Negative: Serial Differential Output 6 (-)
	pci_exp_rxp6	Input	PCI Express Receive Positive: Serial Differential Input 6 (+)
	pci_exp_rxn6	Input	PCI Express Receive Negative: Serial Differential Input 6 (-)
7	pci_exp_txp7	Output	PCI Express Transmit Positive: Serial Differential Output 7 (+)
	pci_exp_txn7	Output	PCI Express Transmit Negative: Serial Differential Output 7 (-)
	pci_exp_rxp7	Input	PCI Express Receive Positive: Serial Differential Input 7 (+)
	pci_exp_rxn7	Input	PCI Express Receive Negative: Serial Differential Input 7 (-)
16-Lane Cores			
0	pci_exp_txp0	Output	PCI Express Transmit Positive: Serial Differential Output 0 (+)
	pci_exp_txn0	Output	PCI Express Transmit Negative: Serial Differential Output 0 (-)
	pci_exp_rxp0	Input	PCI Express Receive Positive: Serial Differential Input 0 (+)
	pci_exp_rxn0	Input	PCI Express Receive Negative: Serial Differential Input 0 (-)
1	pci_exp_txp1	Output	PCI Express Transmit Positive: Serial Differential Output 1 (+)
	pci_exp_txn1	Output	PCI Express Transmit Negative: Serial Differential Output 1 (-)
	pci_exp_rxp1	Input	PCI Express Receive Positive: Serial Differential Input 1 (+)
	pci_exp_rxn1	Input	PCI Express Receive Negative: Serial Differential Input 1 (-)
2	pci_exp_txp2	Output	PCI Express Transmit Positive: Serial Differential Output 2 (+)
	pci_exp_txn2	Output	PCI Express Transmit Negative: Serial Differential Output 2 (-)
	pci_exp_rxp2	Input	PCI Express Receive Positive: Serial Differential Input 2 (+)
	pci_exp_rxn2	Input	PCI Express Receive Negative: Serial Differential Input 2 (-)
3	pci_exp_txp3	Output	PCI Express Transmit Positive: Serial Differential Output 3 (+)
	pci_exp_txn3	Output	PCI Express Transmit Negative: Serial Differential Output 3 (-)
	pci_exp_rxp3	Input	PCI Express Receive Positive: Serial Differential Input 3 (+)
	pci_exp_rxn3	Input	PCI Express Receive Negative: Serial Differential Input 3 (-)
4	pci_exp_txp4	Output	PCI Express Transmit Positive: Serial Differential Output 4 (+)
	pci_exp_txn4	Output	PCI Express Transmit Negative: Serial Differential Output 4 (-)
	pci_exp_rxp4	Input	PCI Express Receive Positive: Serial Differential Input 4 (+)
	pci_exp_rxn4	Input	PCI Express Receive Negative: Serial Differential Input 4 (-)
5	pci_exp_txp5	Output	PCI Express Transmit Positive: Serial Differential Output 5 (+)
	pci_exp_txn5	Output	PCI Express Transmit Negative: Serial Differential Output 5 (-)
	pci_exp_rxp5	Input	PCI Express Receive Positive: Serial Differential Input 5 (+)
	pci_exp_rxn5	Input	PCI Express Receive Negative: Serial Differential Input 5 (-)
6	pci_exp_txp6	Output	PCI Express Transmit Positive: Serial Differential Output 6 (+)
	pci_exp_txn6	Output	PCI Express Transmit Negative: Serial Differential Output 6 (-)
	pci_exp_rxp6	Input	PCI Express Receive Positive: Serial Differential Input 6 (+)
	pci_exp_rxn6	Input	PCI Express Receive Negative: Serial Differential Input 6 (-)

Table 37: PCI Express Interface Signals for 1-, 2-, 4-, 8- and 16-Lane Cores (cont'd)

Lane Number	Name	Direction	Description
7	pci_exp_txp7	Output	PCI Express Transmit Positive: Serial Differential Output 7 (+)
	pci_exp_txn7	Output	PCI Express Transmit Negative: Serial Differential Output 7 (-)
	pci_exp_rxp7	Input	PCI Express Receive Positive: Serial Differential Input 7 (+)
	pci_exp_rxn7	Input	PCI Express Receive Negative: Serial Differential Input 7 (-)
8	pci_exp_txp8	Output	PCI Express Transmit Positive: Serial Differential Output 8 (+)
	pci_exp_txn8	Output	PCI Express Transmit Negative: Serial Differential Output 8 (-)
	pci_exp_rxp8	Input	PCI Express Receive Positive: Serial Differential Input 8 (+)
	pci_exp_rxn8	Input	PCI Express Receive Negative: Serial Differential Input 8 (-)
9	pci_exp_txp9	Output	PCI Express Transmit Positive: Serial Differential Output 9 (+)
	pci_exp_txn9	Output	PCI Express Transmit Negative: Serial Differential Output 9 (-)
	pci_exp_rxp9	Input	PCI Express Receive Positive: Serial Differential Input 9 (+)
	pci_exp_rxn9	Input	PCI Express Receive Negative: Serial Differential Input 9 (-)
10	pci_exp_txp10	Output	PCI Express Transmit Positive: Serial Differential Output 10 (+)
	pci_exp_txn10	Output	PCI Express Transmit Negative: Serial Differential Output 10 (-)
	pci_exp_rxp10	Input	PCI Express Receive Positive: Serial Differential Input 10 (+)
	pci_exp_rxn10	Input	PCI Express Receive Negative: Serial Differential Input 10 (-)
11	pci_exp_txp11	Output	PCI Express Transmit Positive: Serial Differential Output 11 (+)
	pci_exp_txn11	Output	PCI Express Transmit Negative: Serial Differential Output 11 (-)
	pci_exp_rxp11	Input	PCI Express Receive Positive: Serial Differential Input 11 (+)
	pci_exp_rxn11	Input	PCI Express Receive Negative: Serial Differential Input 11 (-)
12	pci_exp_txp12	Output	PCI Express Transmit Positive: Serial Differential Output 12 (+)
	pci_exp_txn12	Output	PCI Express Transmit Negative: Serial Differential Output 12 (-)
	pci_exp_rxp12	Input	PCI Express Receive Positive: Serial Differential Input 12 (+)
	pci_exp_rxn12	Input	PCI Express Receive Negative: Serial Differential Input 12 (-)
13	pci_exp_txp13	Output	PCI Express Transmit Positive: Serial Differential Output 13 (+)
	pci_exp_txn13	Output	PCI Express Transmit Negative: Serial Differential Output 13 (-)
	pci_exp_rxp13	Input	PCI Express Receive Positive: Serial Differential Input 13 (+)
	pci_exp_rxn13	Input	PCI Express Receive Negative: Serial Differential Input 13 (-)
14	pci_exp_txp14	Output	PCI Express Transmit Positive: Serial Differential Output 14 (+)
	pci_exp_txn14	Output	PCI Express Transmit Negative: Serial Differential Output 14 (-)
	pci_exp_rxp14	Input	PCI Express Receive Positive: Serial Differential Input 14 (+)
	pci_exp_rxn14	Input	PCI Express Receive Negative: Serial Differential Input 14 (-)
15	pci_exp_txp15	Output	PCI Express Transmit Positive: Serial Differential Output 15 (+)
	pci_exp_txn15	Output	PCI Express Transmit Negative: Serial Differential Output 15 (-)
	pci_exp_rxp15	Input	PCI Express Receive Positive: Serial Differential Input 15 (+)
	pci_exp_rxn15	Input	PCI Express Receive Negative: Serial Differential Input 15 (-)

Configuration Space

The PCI configuration space consists of the following primary parts, illustrated in the following tables. They include:

- **Legacy PCI v3.0 Type 0/1 Configuration Space Header:**
 - Type 0 Configuration Space Header used by Endpoint applications (see [Table 38: PCI Config Space Header \(Type 0 and 1\)](#))
 - Type 1 Configuration Space Header used by Root Port applications (see [Table 38: PCI Config Space Header \(Type 0 and 1\)](#))
- **Legacy Extended Capability Items:**
 - PCIe Capability Item
 - Power Management Capability Item
 - Message Signaled Interrupt (MSI) Capability Item
 - MSI-X Capability Item (optional)
- **PCIe Capabilities:**
 - Advanced Error Reporting Extended Capability Structure (AER)
 - Alternate Requester ID (ARI) (optional)
 - Device Serial Number Extended Capability Structure (DSN) (optional)
 - Single Root I/O Virtualization (SR-IOV) (optional)
 - Virtual Channel Extended Capability Structure (VC) (optional)
- **PCIe Extended Capabilities:**
 - Device Serial Number Extended Capability Structure (optional)
 - Virtual Channel Extended Capability Structure (optional)
 - Advanced Error Reporting Extended Capability Structure (optional)
 - Media Configuration Access Port (MCAP) Extended Capability Structure (optional)

The core implements up to four legacy extended capability items.

For more information about enabling this feature, see [Customizing and Generating the Core](#).

The core can implement up to ten PCI Express Extended Capabilities. The remaining PCI Express Extended Capability Space is available for users to implement. The starting address of the space available to users begins at 3DCh. If you choose to implement registers in this space, you can select the starting location of this space, and this space must be implemented in the user application.

Table 38: PCI Config Space Header (Type 0 and 1)

Byte Offset	Register (Type 0: Endpoint)				Register Type 1: Root/DS Port						
00h	Device ID		Vendor ID			<i>same as Endpoint</i>					
04h	Status		Command								
08h	Class Code			Rev ID							
0Ch	BIST	Header	Lat Tim	CacheL							
10h	BAR0										
14h	BAR1										
18h	BAR2			SecLTim	SubBus#	SecBus#	PrimBus#				
1Ch	BAR3			Secondary Status		I/O Lim	I/O Base				
20h	BAR4			Memory Limit		Memory Base					
24h	BAR5			PrefetchMemLimit		PrefetchMemBase					
28h	Cardbus CIS Pointer				Prefetchable Base Upper 32 Bits						
2Ch	Subsystem ID	Subsystem Vendor ID			Prefetchable Limit Upper 32 Bits						
30h	Expansion ROM BAR				I/O Limit Upper 16		I/O Base Upper 16				
34h	Reserved		CapPtr	Reserved			CapPtr				
38h	Reserved			Expansion ROM BAR							
3Ch	Max_Lat	Min_Gnt	IntrPin	IntrLine	Bridge Control	IntrPin	IntrLine				

Table 39: PCI Express Config Space

Byte Offset (DW Offset)	Register (Endpoint)				Register (Root/DS Port)				
40h (10h)	PM Capability		NxtCap	PM Cap ID	<i>same as Endpoint</i>				
44h (11h)	Data	BSE	PMCSR						
48h (12h)	MSI Control		NxtCap	MSI Cap ID					
4Ch (13h)	Message Address (Lower)								
50h (14h)	Message Address (Upper)								
54h (15h)	Reserved		Message Data						
58h (16h)	Mask Bits								
5Ch (17h)	Pending Bits								
60h (18h)	MSIX Control		NxtCap	MSIX Cap ID					
64h (19h)	Table Offset			Table BIR					
68h (1Ah)	PBA Offset			PBA BIR					
6Ch (1Bh)	Reserved				Reserved				

Table 39: PCI Express Config Space (cont'd)

Byte Offset (DW Offset)	Register (Endpoint)			Register (Root/DS Port)	
70h (1Ch)	PCIE Capability	NxtCap	PCIE Cap ID	<i>same as Endpoint</i>	
74h (1Dh)	Device Capabilities				
78h (1Eh)	Device Status	Device Control			
7Ch (1Fh)	Link Capabilities				
80h (20h)	Link Status	Link Control			
84h (21h)	Reserved			Slot Capabilities	
88h (22h)	Reserved			Slot Status Slot Control	
8Ch (23h)	Reserved			Root Capabilities ¹ Root Control ¹	
90h (24h)	Reserved			Root Status ¹	
94h (25h)	Device Capabilities 2				
98h (26h)	Device Status 2	Device Control 2		<i>same as Endpoint</i>	
9Ch (27h)	Link Capabilities 2				
A0h (28h)	Link Status 2	Link Control 2			
A4-FCh	Unimplemented Configuration Space (Returns 00000000h)				

Notes:

1. Root Port only; Reserved in Switch DS Ports.

Table 40: PCIe Capability List

PF0	PF1-3	VF	Start Address
Legacy PCI CSH	Legacy PCI CSH	Legacy PCI CSH	0x00
PM	PM	-	0x40
MSI	MSI	-	0x48
MSI-X	MSI-X	MSI-X	0x60
PCIE	PCIE	PCIE	0x70
Extend	Extend		0xB0

Table 41: PCI Express Extended Configuration Space

Byte Offset (DW Offset)	Register (Endpoint)			Register (Root Port)		
100h (40h)	Nxt Cap	Cap Ver	AER Ext Cap			
104h (41h)	Uncorrectable Error Status Register					
108h (42h)	Uncorrectable Error Mask Register					
10Ch (43h)	Uncorrectable Error Severity Register					
110h (44h)	Correctable Error Status Register					
114h (45h)	Correctable Error Mask Register			<i>same as Endpoint</i>		
118h (46h)	Advanced Error Cap. & Control Register					
11Ch (47h)	Header Log Register 1					
120h (48h)	Header Log Register 2					
124h (49h)	Header Log Register 3					
128h (4Ah)	Header Log Register 4					
12Ch (4Bh)	Reserved			Root Error Command Register		
130h (4Ch)	Reserved			Root Error Status Register		
134h (4Dh)	Reserved			Error Source ID Register		
140h (50h)	Nxt Cap	Cap Ver	SR-IOV Ext Cap			
144h (51h)	Capability Register					
148h (52h)	SR-IOV Status	Control				
14Ch (53h)	Total VFs	Initial VFs				
150h (54h)	Func Dep Link	Number VFs				
154h (55h)	VF Stride	First VF Offset				
158h (56h)	VF Device ID	Reserved				
15Ch (57h)	Supported Page Sizes					
160h (58h)	System Page Size					
164h (59h)	VF Base Address Register 0					
168h (5Ah)	VF Base Address Register 1					
16Ch (5Bh)	VF Base Address Register 2					
170h (5Ch)	VF Base Address Register 3					
174h (5Dh)	VF Base Address Register 4					
178h (5Eh)	VF Base Address Register 5					
180h (60h)	Nxt Cap	Cap Ver	ARI Ext Cap			
184h (61h)	Control		NxtFn	FnGrp		
188h - 19Ch	Reserved					
1A0h (68h)	Nxt Cap	Cap Ver	DSN Ext Cap			
1A4h (69h)	Device Serial Number (1st)					
1A8h (6Ah)	Device Serial Number (1st)					
1ACh - 1BCh	Reserved					

Table 41: PCI Express Extended Configuration Space (cont'd)

Byte Offset (DW Offset)	Register (Endpoint)			Register (Root Port)
1C0h (70h)	Nxt Cap	Cap Ver	2nd PCIE Ext Cap	
1C4h (71h)	Lane Control			
1C8h (72h)	Reserved	Lane Error Status		
1CCh (73h)	Lane 1 Eq Ctrl Reg	Lane 0 Eq Ctrl Reg		
1D0h (74h)	Lane 3 Eq Ctrl Reg	Lane 2 Eq Ctrl Reg		
1D4h (75h)	Lane 5 Eq Ctrl Reg	Lane 4 Eq Ctrl Reg		
1D8h (76h)	Lane 7 Eq Ctrl Reg	Lane 6 Eq Ctrl Reg		
1DCh (77h)	Lane 9 Eq Ctrl Reg	Lane 8 Eq Ctrl Reg		<i>same as Endpoint</i>
1E0h (78h)	Lane 11 Eq Ctrl Reg	Lane 10 Eq Ctrl Reg		
1E4h (79h)	Lane 13 Eq Ctrl Reg	Lane 12 Eq Ctrl Reg		
1E8h (7Ah)	Lane 15 Eq Ctrl Reg	Lane 14 Eq Ctrl Reg		
1ECh (7Bh)	Lane 1 Eq Ctrl 2 Reg	Lane 0 Eq Ctrl 2 Reg		
1F0h (7Ch)	Lane 3 Eq Ctrl 2 Reg	Lane 2 Eq Ctrl 2 Reg		
1F4h (7Dh)	Lane 5 Eq Ctrl 2 Reg	Lane 4 Eq Ctrl 2 Reg		
1F8h (7Eh)	Lane 7 Eq Ctrl 2 Reg	Lane 6 Eq Ctrl 2 Reg		
1FCh (7Fh)	Reserved			
200h(80h)	Nxt Cap	Cap Ver	VC Ext Cap	
204h(81h)	Port VC Capability Register 1			
208h(82h)	Port VC Capability Register 2			
20Ch(83h)	Port VC Status			
210h (84h)	VC Resource Capability Register 0			
214h (85h)	VC Resource Control Register 0			
218h (86h)	VC Resource Stat 0			
21Ch (87h)	Reserved			
220h (88h)	Nxt Cap	Cap Ver	TPH Ext Cap	
224h (89h)	TPH Requester Capability Register			
228h (8Ah)	TPH Requester Control Register			
22Ch (8Bh) - 32Ch	TPH ST Table			
330h (CCh)	Reserved			Nxt Cap
334h (CDh)				Cap Ver
338h (CEh)				Loopback VSEC
33Ch (CFh)				Loopback Header
340h (D0h)				Loopback Control
344h (D1h)				Loopback Status
348h (D2h)				Error Count 1
34Ch (D3h)				Error Count 2
				Error Count 3
				Error Count 4

Table 41: PCI Express Extended Configuration Space (cont'd)

Byte Offset (DW Offset)	Register (Endpoint)			Register (Root Port)
350h (D4h)	Nxt Cap	Cap Ver	MCAP VSEC	
354h (D5h)	MCAP Header			
358h (D6h)	JTAG ID			
35Ch (D7h)	Bitstream Version			
360h (D8h)	Status Register			
364h (D9h)	Control Register			
368h (DAh)	Data Register			
36Ch (DBh)	Register Read Data 0			
370h (DCh)	Register Read Data 1			
374h (DDh)	Register Read Data 2			
378h (DEh)	Register Read Data 3			
37Ch - FFCh	Reserved			Reserved

Table 42: User Design Extended Configuration List

PF0	PF1-3	VF	Start Address	PF0 Next Pointer
PCI Express Extended Configuration Space Enable	PCI Express Extended Configuration Space Enable	PCI Express Extended Configuration Space Enable	0x0	PCIE4: 0x480
				PCIE4C: 0xE80

Designing with the Core

This section includes guidelines and additional information to facilitate designing with the core.

Tandem Configuration

PCI Express® is a plug-and-play protocol, meaning that at power up the PCIe® host will enumerate the system. This process consists of the host reading the requested address size from each device and then assigning a base address to the device. As such, PCIe interfaces must be ready when the host queries them or they will not get assigned a base address. The PCI Express specification states that PERST# must deassert 100 ms after the power good of the systems has occurred, and a PCI Express port must be ready to link train no more than 20 ms after PERST# has deasserted. This is commonly referred to as the 100 ms boot time requirement.

Tandem Configuration utilizes a two-stage methodology that enables the IP to meet the configuration time requirements indicated in the PCI Express specification. Multiple use cases are supported with this technology:

- **Tandem PROM:** Load the single two-stage bitstream from the flash.
- **Tandem PCIe:** Load the first stage bitstream from flash, and deliver the second stage bitstream over the PCIe link to the MCAP.
- **Tandem PCIe with Field Updates:** After a Tandem PCIe initial configuration, update the entire user design while the PCIe link remains active. The update region (floorplan) and design structure are predefined, and Tcl scripts are provided.

Note: In UltraScale+™ devices, Tandem PCIe must be used for Field Updates. Tandem PROM does not support Field Updates.

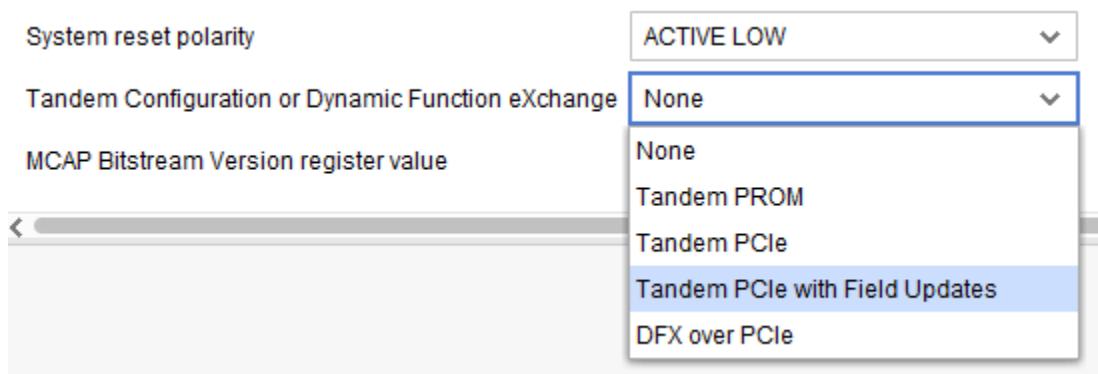
- **Tandem + Dynamic Function eXchange:** This is a more general case of Tandem Configuration followed by Dynamic Function eXchange (DFX) of any size or number of dynamic regions.
- **DFX over PCIe:** This is a standard configuration followed by DFX, using the PCIe / MCAP as the delivery path of partial bitstreams.

To enable any of these capabilities, select the appropriate option when customizing the core. In the Basic tab:

1. Change the Mode to **Advanced**.

2. Change the Tandem Configuration or Dynamic Function eXchange option according to your particular case:
 - **Tandem PROM** for the Tandem PROM use case.
 - **Tandem PCIe** for Tandem PCIe or Tandem and Dynamic Function eXchange use cases.
 - **Tandem PCIe with Field Updates** only for the predefined Field Updates use case.
 - **DFX over PCIe** to enable the MCAP link for DFX, without enabling Tandem Configuration.

Figure 2: Tandem Configuration or Dynamic Function eXchange Option



The AXI DMA/Bridge Subsystem for PCI Express supports Tandem Configuration and Dynamic Function eXchange features for UltraScale+ devices, including Tandem with Field Updates. Device support details are documented in the *DMA/Bridge Subsystem for PCI Express Product Guide (PG195)*, but the Tandem implementation details are presented thoroughly only here within this document.

Supported Devices

The Integrated Block for PCIe core and Vivado® tool flow support implementations targeting Xilinx® reference boards and specific part/package combinations.

Tandem Configuration is available as a production solution for most UltraScale+™ devices. Bitstream generation is disabled by default for all ES silicon. Tandem Configuration supports the configurations found in the following table.

HDL	Verilog Only
PCIe Configuration	All configurations (max: X16Gen3)
Xilinx Reference Board Support	KCU116 Evaluation Board for Kintex® UltraScale+™ FPGA VCU118 Evaluation Board for Virtex® UltraScale+™ FPGA

Device Support	Part ¹	PCIe Block Location	Tandem Configuration	Tandem PCIe with Field Updates
Kintex UltraScale+	KU3P	PCIE40E4_X0Y0	Production	Production
	KU5P	PCIE40E4_X0Y0	Production	Production
	KU11P	PCIE40E4_X1Y0	Production	Production
	KU15P	PCIE40E4_X1Y0	Production	Production
Virtex UltraScale+	VU3P	PCIE40E4_X1Y0	Production	Production
	VU5P	PCIE40E4_X1Y0	Production	Production
	VU7P	PCIE40E4_X1Y0	Production	Production
	VU9P	PCIE40E4_X1Y2	Production	Production
	VU11P	PCIE40E4_X0Y0	Production	Production
	VU13P	PCIE40E4_X0Y1	Production	Production
	VU27P	PCIE40E4_X0Y0	Production	Production
	VU29P	PCIE40E4_X0Y0	Production	Production
	VU31P	PCIE4CE4_X1Y0	Production	Production
	VU33P	PCIE4CE4_X1Y0	Production	Production
	VU35P	PCIE4CE4_X1Y0	Production	Production
	VU37P	PCIE4CE4_X1Y0	Production	Production
Zynq® UltraScale+™ MPSoC	ZU4CG/EG/EV	PCIE40E4_X0Y1	Production	Production
	ZU5CG/EG/EV	PCIE40E4_X0Y1	Production	Production
	ZU7CG/EG/EV	PCIE40E4_X0Y1	Production	Production
	ZU11EG	PCIE40E4_X1Y0	Production	Production
	ZU17EG	PCIE40E4_X1Y0	Production	Production
	ZU19EG	PCIE40E4_X1Y0	Production	Production

Zynq® UltraScale+™ RFSoC ²	ZU21DR	PCIE40E4_X0Y0	Production (Tandem PROM only)	Not supported
	ZU25DR	PCIE40E4_X0Y0	Production (Tandem PROM only)	Not supported
	ZU27DR	PCIE40E4_X0Y0	Production (Tandem PROM only)	Not supported
	ZU28DR	PCIE40E4_X0Y0	Production (Tandem PROM only)	Not supported
	ZU29DR	PCIE40E4_X0Y0	Production (Tandem PROM only)	Not supported
	ZU43DR	PCIE4CE4_X0Y0	Not Supported	Not Supported
	ZU45DR	PCIE4CE4_X0Y0	Beta (Tandem PROM only)	Not Supported
	ZU46DR	PCIE4CE4_X0Y0	Beta (Tandem PROM only)	Not Supported
	ZU47DR	PCIE4CE4_X0Y0	Beta (Tandem PROM only)	Not Supported
	ZU48DR	PCIE4CE4_X0Y0	Beta (Tandem PROM only)	Not Supported

Notes:

1. Only production silicon is officially supported. Bitstream generation is disabled for all engineering sample silicon (ES1, ES2) devices.
2. Zynq RFSoC devices do not have MCAP-enabled PCIe block locations so this support is for the Tandem PROM solution only.

Overview of Tandem Tool Flow

Tandem PROM and Tandem PCIe solutions are only supported in the Vivado® Design Suite. The tool flow for both solutions is as follows:

1. Customize the core: select a supported device from the previous table, select the Advanced configuration Mode option, and select **Tandem PROM** or **Tandem PCIe** for the Tandem Configuration or Dynamic Function eXchange option.
2. Generate the core.
3. Open the example project, and implement the example design.
4. Use the IP and XDC from the example project in your project, and instantiate the core.
5. Synthesize and implement your design.
6. Generate bit and then prom files.

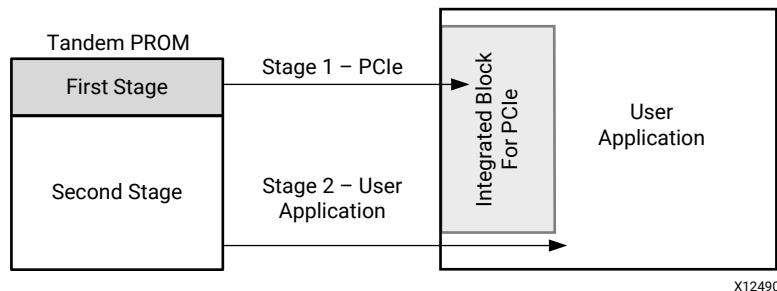
As part of the Tandem flows, certain elements located outside of the PCIe core logic must also be brought up as part of the stage 1 bitstream. Vivado design rule checks (DRCs) identify these situations and provide direction on how to resolve the issue. This normally consists of modifying or adding additional constraints to the design.

When the example design is created, an example XDC file is generated with certain constraints that need to be copied over into your XDC file for your specific project. The specific constraints are documented in the example design XDC file. In addition, this example design XDC file contains examples of how to set options for flash memory devices, such as BPI and SPI.

Tandem PROM

The Tandem PROM solution splits a bitstream into two parts and both of those parts are loaded from an onboard local configuration memory (typically, any PROM or flash memory device). The first part of the bitstream configures the PCI Express® portion of the design and the second part configures the rest of the FPGA. Although the design is viewed to have two unique stages, shown in the following figure, the resulting BIT file is monolithic and contains both stage 1 and stage 2.

Figure 3: Tandem PROM Bitstream Load Steps

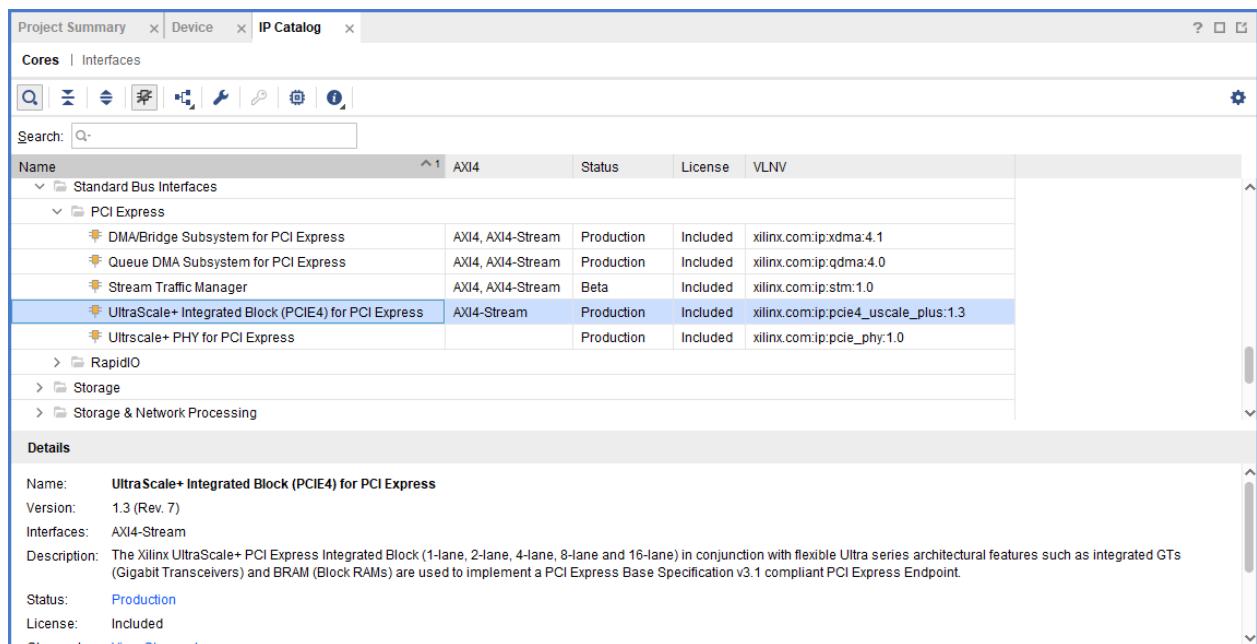


Tandem PROM UltraScale+ Example Tool Flow

This section demonstrates the Vivado tool flow from start to finish when targeting an UltraScale+ device. Paths and pointers within this flow description assume the default component name `pcie4_ultrascale_plus_0` is used.

1. Create a new Vivado project, and select a supported part/package shown in the previous table.
2. In the Vivado IP catalog, expand **Standard Bus Interfaces** → **PCI Express**, and double-click **UltraScale+ PCI Express Integrated Block** to open the Customize IP dialog box.

Figure 4: Vivado IP Catalog



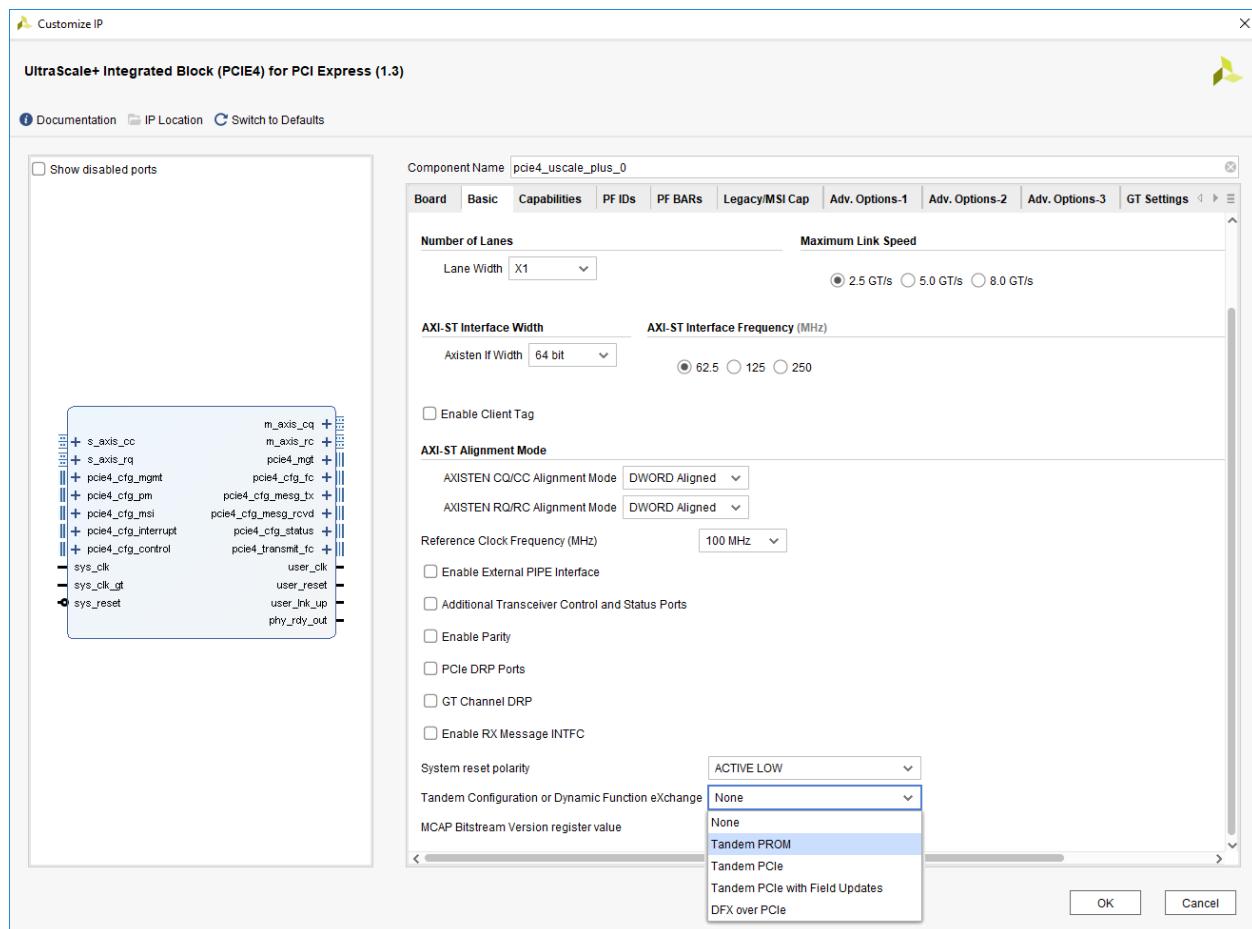
3. In the Customize IP dialog box Basic tab, ensure the following options are selected:

- Mode: **Advanced**
- PCIe Block Location: **X1Y2**

Note: Use the required PCIe Block Location for the device targeted, as listed in the previous table. This design example targets a VU9P.

- Tandem Configuration or Dynamic Function eXchange: **Tandem PROM**

Figure 5: Tandem PROM



4. Perform additional PCIe customizations, and click **OK** to generate the core.
5. Click **Generate** when asked about which output products to create.
6. In the Sources tab, right-click the core, and select **Open IP Example Design**.

A new instance of Vivado is created and the example design is automatically loaded into the Vivado IDE.

7. Run Synthesis and Implementation.

Click **Run Implementation** in the Flow Navigator. Select **OK** to run through synthesis first. The design runs through the complete tool flow and the result is a fully routed design that supports Tandem PROM.

8. Set up PROM or Flash settings.

Set the appropriate settings to correctly generate a bitstream for a PROM or flash memory device. In the PCIe core constraint file (for example, `xilinx_pcie4_uscale_plus_x1y2.xdc`):

- Uncomment and customize any constraints that define the configuration settings.

- The one constraint that is required is CONFIG_MODE. For example: `set_property CONFIG_MODE BPI16 [current_design]`

For more information, see [Programming the Device](#).

9. Generate the bitstream.

After Synthesis and Implementation is complete, click **Generate Bitstream** in the Flow Navigator. A bitstream supporting Tandem configuration is generated in the runs directory, for example: `./pcie_ultrascale_plus_0_example.runs/impl/xilinx_pcie4_uscale_plus_ep.bit`.

Note: You have the option of creating the first and stage 2 bitstreams independently. This flow allows you to control the loading of each stage through the JTAG interface for testing purposes. These bitstreams are the same as the ones used for the Tandem PCIe solution when loaded using JTAG. Attempting to load only the stage1 bitstream from flash memory does not work in hardware due to the difference in the `HD.OVERRIDE_PERSIST` setting that is used for Tandem PCIe designs.

```
set_property HD.TANDEM_BITSTREAMS SEPARATE [current_design]
```

The resulting bit files created are named

```
xilinx_pcie4_uscale_plus_ep_tandem1.bit and  
xilinx_pcie4_uscale_plus_ep_tandem2.bit.
```

10. Generate the PROM file.

Run the following command in the Vivado Tcl Console to create a PROM file supported on a Xilinx development board.

```
write_cfgmem -format mcs -interface BPIx16 -size 256 -loadbit "up 0x0  
xilinx_pcie4_uscale_plus_ep.bit" xilinx_pcie3_uscale_ep.mcs
```

Tandem PROM Summary

By using Tandem PROM, you can significantly reduce the amount of time required to configure the PCIe portion of an UltraScale+ device design. The UltraScale+ Devices Integrated Block for PCIe core manages many design details, allowing you to focus your attention on the user application.

Tandem PCIe

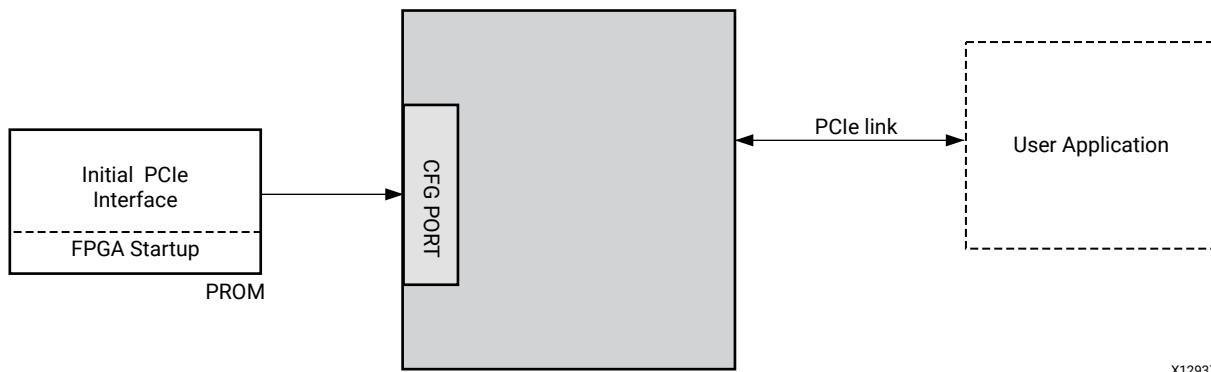
Tandem PCIe is similar to Tandem PROM. In the first stage bitstream, only the configuration memory cells that are necessary for PCI Express operation are loaded from the PROM. After the stage 1 bitstream is loaded, the PCI Express port is capable of responding to enumeration traffic. Subsequently, the stage 2 bitstream is transmitted through the PCI Express link.



VIDEO: [Create a Tandem PCIe Design for the KCU105](#) explains how to create a Tandem design targeting the KCU105 Evaluation Kit.

The following figure illustrates the bitstream loading flow.

Figure 6: Tandem PCIe Bitstream Load Steps



X12937

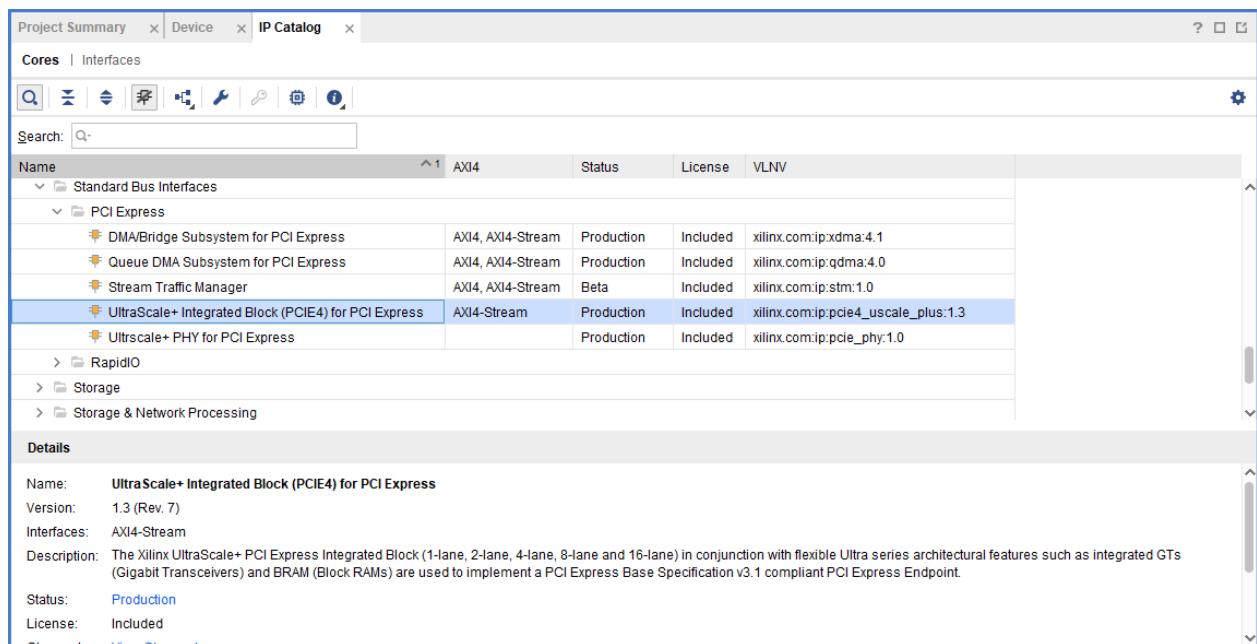
Tandem PCIe is similar to the standard model used today in terms of tool flow and bitstream generation. Two bitstreams are produced when running bitstream generation. One BIT file representing the stage 1 bitstream is downloaded into the PROM while the other BIT file representing the user application (stage 2) configures the rest of the FPGA using the Media Configuration Access Port (MCAP).

Tandem PCIe UltraScale+ Example Tool Flow

This section demonstrates the Vivado® tool flow from start to finish when targeting an UltraScale+ reference board. Paths and pointers within this flow description assume the default component name `pcie4_ultrascale_plus_0` is used.

1. When creating a new Vivado project, select a supported part/package shown in the previous table.
2. In the Vivado IP catalog, expand **Standard Bus Interfaces** → **PCI Express**, and double-click **UltraScale+ PCI Express Integrated Block** to open the Customize IP dialog box.

Figure 7: Vivado IP Catalog



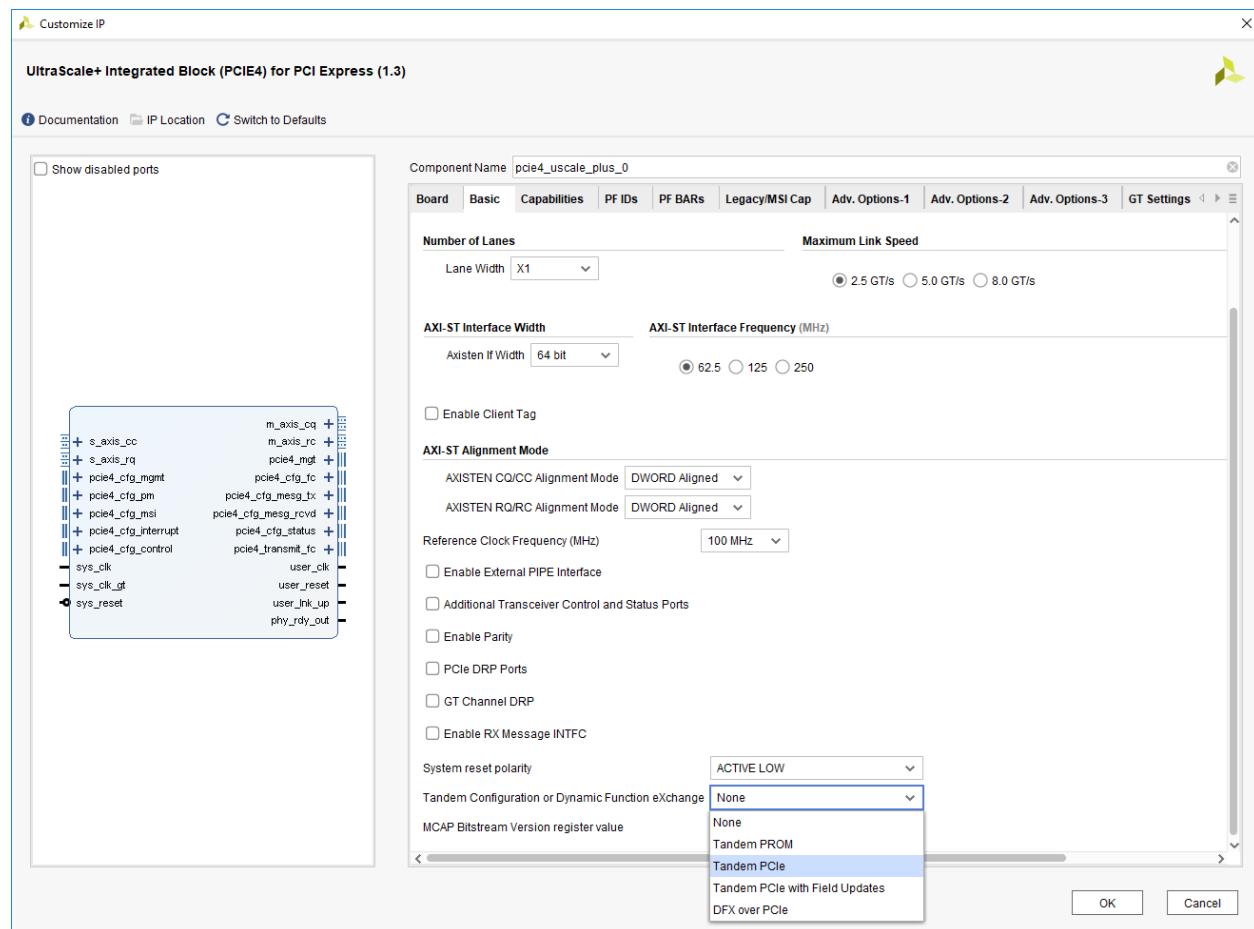
3. In the Customize IP dialog box Basic tab, ensure the following options are selected:

- Mode: **Advanced**
- PCIe Block Location: **X1Y2**

Note: Use the required PCIe Block Location for the device targeted, as listed in the previous table. This design example targets a VU9P.

- Tandem Configuration or Dynamic Function eXchange: **Tandem PCIe**.

Figure 8: Tandem PCIe

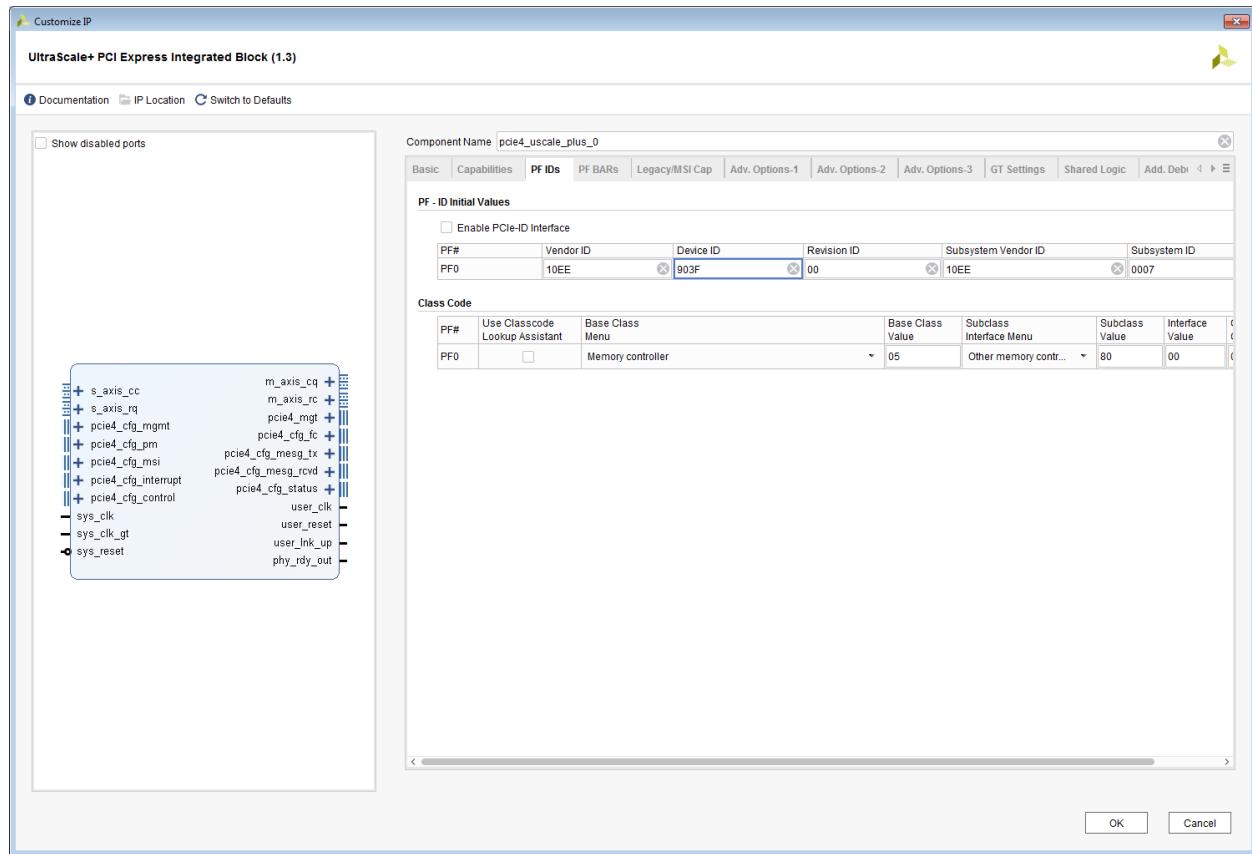


4. The example design software attaches to the device through the Vendor ID and Device ID. The Vendor ID must be `16'h10EE` and the Device ID must be `16'h903F`. In the PF0 IDs tab, set:

- Vendor ID: **10EE**
- Device ID: **903F**

Note: An alternative solution is the Vendor ID and Device ID can be changed, and the driver and host PC software updated to match the new values.

Figure 9: IDs



5. Perform additional PCIe customizations, and select **OK** to generate the core.

After core generation, the core hierarchy is available in the Sources tab in the Vivado IDE.

6. In the Sources tab, right-click the core, and select **Open IP Example Design**.

A new instance of Vivado is created and the example design project automatically loads in the Vivado IDE.

7. Run Synthesis and Implementation.

Click **Run Implementation** in the Flow Navigator. Select **OK** to run through synthesis first. The design runs through the complete tool flow, and the end result is a fully routed design supporting Tandem PCIe.

8. Setup PROM or Flash settings, and request two explicit bit files.

Set the appropriate settings to correctly generate a bitstream for a PROM or flash memory device by:

- modifying the constraints in the PCIe IP constraint file (for example `pcie4_ultrascale_plus_0_tandem`).

- requesting two explicit bitstreams by setting these properties, as seen in the example design constraint file:

```
set_property HD.OVERRIDE_PERSIST FALSE [current_design]
set_property HD.TANDEM_BITSTREAMS Separate [current_design]
```

Other values for HD.TANDEM_BITSTREAMS are Combined (default), which is used for the Tandem PROM solution, and None, which generates a standard single-stage bitstream for the entire device. For more information, see [Programming the Device](#).

9. Generate the bitstream.

After Synthesis and Implementation are complete, click **Generate Bitstream** in the Flow Navigator. The following two files are created and placed in the runs directory:

```
xilinx_pcnie4_uscale_plus_ep_tandem1.bit
xilinx_pcnie4_uscale_plus_ep_tandem2.bit
```

10. Generate the PROM file for the stage 1.

Run the following command in the Vivado Tcl Console to create a PROM file supported on an UltraScale+ development board.

```
write_cfgmem -format mcs -interface BPI -size 256 -loadbit "up 0x0
xilinx_pcnie4_uscale_plus_ep_tandem1.bit"
xilinx_pcnie4_uscale_plus_ep_tandem1.mcs
```

Loading Stage 2 Through PCI Express

An example kernel mode driver and user space application is provided with the IP. For information on retrieving the software and documentation, see [AR 64761](#).

Using Tandem PCIe on Zynq MPSoC Devices

While the processing system (PS) and processor configuration access port (PCAP) are in control of the configuration logic, the programmable logic (PL) and the media configuration access port (MCAP) (as well as the internal configuration access port (ICAP)) are locked out of the configuration logic. By default, the PCAP is enabled and the MCAP does not have access to load Stage 2 until the PCAP_PR bit is switched.

The PCAP and MCAP/ICAP interfaces are mutually exclusive and cannot be used simultaneously. Switching between PCAP and MCAP/ICAP is possible, but you must ensure that no commands or data are being transmitted or received before changing interfaces. Failure to do this could lead to unexpected behavior. Bit 0 (PCAP_PR) of the Control Register (PCAP . CTRL) selects between MCAP/ICAP and PCAP for PL configuration. The default is PCAP (1), but that can be changed to MCAP/ICAP (0) to enable this configuration port. To enable the MCAP/ICAP for Zynq® UltraScale+™ MPSoC devices, set the PCAP_PR field of the pcap_ctrl (CSU) register. This bit selects between MCAP or ICAP, and PCAP for PL configuration. There are two ways to clear the PCAP_PR bit and grant the ICAP/MCAP access for loading a stage 2 image.

1. Write the PCAP_CTRL register to set MCP/ICAP as configuration port through JTAG:
 - Use the Vitis™ Xilinx Software Command-Line Tools (XSCT) to connect to the JTAG chain.
 - Select PSU as the target.
 - Use memory read (MRD) and memory write (MWR) to change the default behavior.
2. Write to the PCAP_CTRL register to set the MCAP/ICAP as the enabled configuration port through FSBL:
 - Place the logic under XFsb1_HookBeforeHandoff().

A list of FSBL hooks can be found at <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842019/FSBL>

```
//Command to SET ICAP as Configuration port//  
Xil_Out32(0xFFCA3008, 0x0); // write PCAP_CTRL
```

Tandem PCIe Summary

By using Tandem PCIe, you can significantly reduce the amount of time required for configuration of the PCIe portion of an UltraScale device design, and can reduce the bitstream flash memory storage requirements. The UltraScale+™ Devices Integrated Block for PCIe core manages many design details, allowing you to focus your attention on the user application.

Tandem PCIe with Field Updates

Tandem PCIe with Field Updates is a solution for UltraScale+ devices that allows designers to meet fast configuration needs and dynamically change the user application by loading a new bitstream over the PCIe link without the PCIe link going down. The solution uses Tandem PCIe Configuration to initially configure the device when the power is turned on, followed by Dynamic Function eXchange (DFX) of a predefined region in the FPGA. This allows the stage 1 bitstream to be locked in flash, with updates to that image are only required if the UltraScale+ Devices Integrated Block for PCIe core characteristics or the I/O or clock management blocks in the configuration bank (bank 65) must change. Field Updates allows the user application, basically everything else in the design, to be dynamically reloaded as new features or functionality are needed.

The Tandem with Field Updates approach uses DFX, and no special license is required. Tandem with Field Updates is a specific, predefined use case of a more general Tandem + DFX solution. These two solutions (Tandem and DFX) are also supported in general in the same design, allowing you to partially reconfigure smaller and multiple regions within the user application.

After the PCIe IP is generated, a sample design can be created that provides the template for the Field Updates structure. This design shows the required structure, floorplan, properties, and scripts that can be adapted for your design. Follow this example to sort your design into two sections, mapping them to the two bitstream stages. The sample design is delivered in a scripted non-project mode, but project mode can be used as well.

Reconfigurable Stage Twos

UltraScale+ devices support a new methodology for Field Updates, called Reconfigurable Stage Twos. This solution provides more flexibility in how bitstreams are delivered to configure and reconfigure the user application, and reduces the number of bitstreams that must be managed.

Note: This feature is not available for UltraScale, because it relies on silicon features not available in the older architecture.

Essentially with Reconfigurable Stage Twos, stage 2 bitstreams can act as partial bitstreams, and therefore they are interchangeable when paired with a fixed stage 1 bitstream. You can pick between any compatible stage 2 bitstream to complete the initial configuration of a device, then return to these same stage 2 bitstreams to partially reconfigure the device, loading a new user application on the fly. The PCIe endpoint remains up and linked through these transitions and is used as the pathway for bitstream delivery.

Compatible in this case means that these stage 2 bitstreams have been created using the Dynamic Function eXchange design flow, where the PCIe IP is the static design and everything else is placed within a Reconfigurable Partition. As long as the implementation of the PCIe IP design is locked, PR Verify will validate the compatibility of all stage 2 designs, ensuring a safe environment where these bitstreams are delivered to the target device.



CAUTION! While it is still possible to create traditional partial bitstreams for Tandem with Field Updates in UltraScale+ devices, these bitstreams must not be used as stage 2 bitstreams. They can only be used for dynamic reconfiguration of user applications and must not be used to complete an initial boot of the device.



CAUTION! Through Vivado 2019.2, stage 2 bitstreams are missing some configuration frames which may cause designs to behave incorrectly after a dynamic reconfiguration. This issue was resolved in Vivado 2020.1, but if older versions must be used continue to use the solution with separate bitstreams for stage 2 and partial reconfiguration. For more information and details on the workaround, see [AR 71877](#).

Differences from Tandem Configuration

There are two primary differences between standard Tandem Configuration and Tandem with Field Updates, in terms of the design layout and structure.

Design Layout

First, for both variations, the floorplan is established as part of the IP creation and should not be modified, but Tandem PCIe with Field Updates creates two sets of Pblocks instead of one. In addition to the same Pblocks tagged with HD . TANDEM for the stage 1 logic, a second, much larger set of Pblocks tagged with HD . RECONFIGURABLE for the user application are inferred. The former applies the same stage 1 creation rules as the standard Tandem Configuration solution. The latter enforces all rules for Dynamic Function eXchange, most notably routing containment to ensure the partial bitstream contains the entirety of the implementation for the user application.

The following figure shows the floorplan generated for the KU5P sample design for Tandem PCIe with Field Updates. The pink region is reserved for the PCIe IP. This region includes the PCIe hard block, CLB, block RAM and transceiver sites for implementing the IP, and one I/O bank to enable the physical reset pin. The yellow region is the inverse of the pink and represents the Reconfigurable Partition (RP) for the user application. It covers remaining resources including all clocks, transceivers, I/O and logic not covered by the PCIe IP.

Note: In the lower right corner of the RP (named `update_region`), you can find the PIO example design logic, as well as the collection of partition pins used to connect the two sections of the design.

For more information on partition pins or other aspects of the DFX solution, see the *Vivado Design Suite User Guide: Dynamic Function eXchange (UG909)*.

Figure 10: Tandem PCIe with Field Updates Floorplan for the KU5P



Design Structure

The second difference is the design structure. In order to swap one user application from one version to the next, it must be completely enclosed within its own level of hierarchy. The interface of this instantiation cannot change; otherwise, the top-level static design needs to be recompiled. Everything other than the UltraScale+ Devices Integrated Block for PCIe core (in its own level of hierarchy below top) and any I/O logic (buffers, XiPhy, etc.) that are placed in bank 65 are in this level of hierarchy (and below). This means that all I/O logic for all other banks must be placed here and not inferred at the top level, so instantiation of I/O buffers is required.

Note: To see a simple example of this design structure, generate an example design while targeting the KCU116 demo board.

Another requirement of the design structure is that all elements to be placed in the configuration frame must also be part of this top-level design (or another level of hierarchy separate from the PCIe IP and the user application). These elements include the BSCAN, ICAP, STARTUP and related components. See *Vivado Design Suite User Guide: Dynamic Function eXchange (UG909)* for the complete list. The elements must be hierarchically isolated because they are not permitted to be dynamically reconfigured. The implications of this mean that IP cores that require these elements, such as the Vivado Debug Hub and the Memory Interface Generator (MIG), which both use BSCAN, must take special precautions to be safely implemented. For details, see [Debugging Tandem with Field Updates Designs](#).

All other considerations for Tandem Configuration are still applicable for Tandem PCIe with Field Updates.

Tandem Configuration with Field Updates Software Flow

Follow these steps to build the Tandem IP and compile the sample design. The Vivado® Design Suite processes the design from IP customization to bitstream generation for two design configurations.

1. Launch the Customize IP dialog box to customize the UltraScale+ Devices Integrated Block for PCIe core.
Note: This solution is supported only in version 1.3 (and newer) of the IP.
2. Customize the PCIe IP core with **Tandem PCIe with Field Updates** selected. The **Advanced Mode** option must be selected to see this option.
3. Generate output products by using the default Out of context per IP synthesis option. This synthesizes the IP to create a checkpoint that can be inserted in your full design.
4. Right-click the IP in the Design Sources tab, and select **Open IP Example Design**.



IMPORTANT! *The design, as delivered, must be processed in a scripted non-project mode, because the sample design's project is not yet enabled for DFX.*

The example design comes with a set of scripts for use with the non-project Tcl flow. The sample scripts are located in the `field_update_scripts` folder, but these are all referenced by the master script in the design example folder. You can convert this example design to project mode, if needed.

5. In a Vivado Tcl shell, source `design_field_updates.tcl`, which is found in the project directory. This file compiles the example design with two versions:
 - With the default settings, Ver1 is the initial design so the static design (essentially just the PCIe IP) is used from this version.

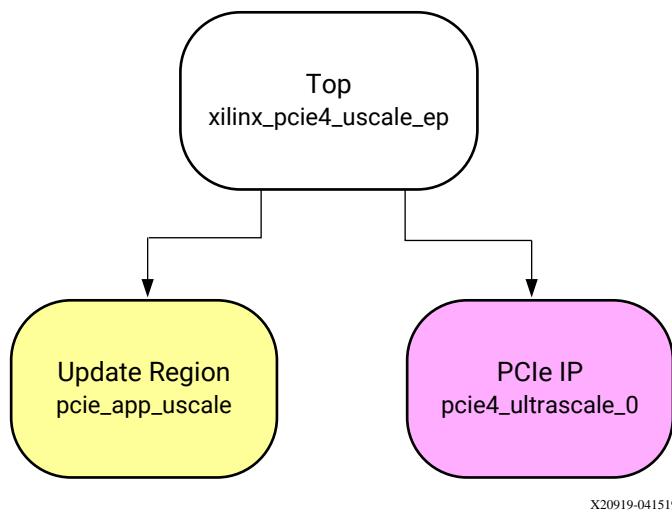
- Ver2 is logically the same as Ver1 but shown as a separate configuration to give an example of what processing a second design revision looks like. In an actual user design, these logical modules would be replaced by different functionality.

Details on the Sample Design

The simple PIO design generated with the IP represents the required design structure for Tandem with Field Updates. The top-level design file, `xilinx_pcie_uscale_ep.v`, declares the top level pin list and instantiates `pcie4_ultrascale_0` (the IP) and `pcie_app_uscale` (everything else). These two submodules are tagged with the `HD.TANDEM` and `HD.RECONFIGURABLE` properties respectively to instruct the implementation tools to follow both sets of rules during place and route.

The following figure shows the basic hierarchy of a Tandem PCIe with Field Updates design. The stage 1 bitstream is the pink module only, with stage 2 being comprised of the yellow and white. The partial bitstreams are constructed of just the yellow module. The names of the hierarchical instances from the sample design are shown.

Figure 11: Required Design Hierarchy for Tandem PCIe With Field Updates



Because the vast majority of the design must be placed in a Reconfigurable Partition, this vast majority must reside in the `pcie_app_uscale` hierarchical tree. This includes everything except any I/O and I/O logic that is located in bank 65 alongside the PCIe IP PERSTN pin. All user I/O buffers and logic, clocks, GTs and everything else must be in this level of hierarchy, so it can be swapped for the new version when ready. This requirement means that designs with IP that contain embedded I/Os that must be placed in the same bank as the reset should not be considered for Field Updates. Manual extraction of these I/Os would be necessary and is difficult to manage.

The three partitions in the design (Top, PCIe IP, and User Application) are all synthesized separately, with the two submodules marked out-of-context. This ensures the separate pieces are not optimized across boundaries, which would prevent the ability to swap these blocks for implementation. Any synthesis tool can be used, so long as automatic I/O insertion is disabled. With Vivado synthesis, this is done by selecting the `-mode out_of_context` option. Implementation of each version is done with the entire design in context. From the second version onward, the place and route results of the PCIe IP and the minimal top-level logic is locked so it cannot change.

Details on the Design Scripts

The `design_field_updates.tcl` script is the master, calling other scripts in the `field_update_scripts` folder. Set variables in this master script to synthesize and implement only the first version (ver1) or only the second version, (ver2) and use it as a template if additional versions are needed. The fundamental flow is a Dynamic Function eXchange flow. For more details about DFX, see the *Vivado Design Suite User Guide: Dynamic Function eXchange (UG909)*. Sample scripts are provided with the *Vivado Design Suite Tutorial: Dynamic Function eXchange (UG947)*.

Here are a few things to note about the scripts.

- A specific directory structure is defined within the scripts, so it is easiest to follow what is created and used by the examples.
- Flags can be set on lines 99 to 109 to determine which configurations must be compiled.
 - If only the initial design version is available, all the version 2 flags should be set to 0, as well as for PR_verify, because there is nothing to compare.
 - Likewise, if version 1 has already been done and you are ready to compile version 2, flip the flags completely to run the version 2 configuration and PR_verify steps.
- The script is set up for an initial configuration (ver1) and one field update (ver2). To create additional field updates, copy all the references for ver2 to create ver3 and beyond. Another approach is to globally replace ver2 with ver3, set the flags, and use the second case noted above. If ver1 and ver2 are complete, there is no reason to revisit them.
- Out-of-context synthesis of the user application modules is managed in the `update_verX_synth.tcl` scripts (where X represents the version number). Declaration of sources, constraints and options are done in these files.
- In-context implementation of each version is managed by the `update_verX_impl.tcl` scripts. Modify these to change options, add constraint files or generate reports for any version.

Bitstream Generation

The supplied scripts create bitstreams for any versions requested. The set `runUpdateVerXBitstreams 1` flag (where X represents the version number) calls the bitstream generation routine later script. As seen in the "Running Bitstream Generation" section in the `design_field_updates.tcl` script, multiple values can be supplied to generate the different bitstream types, depending on what is needed. In most cases, and certainly for the streaming core, only TandemPCIe should be used. The following values generate these bitstreams (showing ver1 as an example):

- **TandemPCIe:** Generates the following bitstreams.
 - `ver1_tpcie_tandem1` – stage 1 bitstream for Tandem PCIe, to be stored in flash.
 - `ver1_tpcie_tandem2` – stage 2 bitstream for Tandem PCIe, to be delivered over PCIe link; this bitstream is reloadable on the fly.
- **PR:** Generates only this partial bitstream.
 - `ver1_tpcie_update_region_partial` – partial bitstream to load in functionality of ver1.

Because stage 2 bitstreams can be used as partial bitstreams in the UltraScale+ Field Updates solution, there is no need to use this option for the AXI4 streaming core. For the DMA version of the core, the partial bit file will have a different size than the stage 2 bit file. The partial bit file does not include the DMA portion of the design (so it will not be reset during reconfiguration) but it does use expanded routing, which in some cases covers the entire device.

Note that multiple formats (`.bit`, `.bin`, `.mcs`, `.prm`) are generated by default, as requested in `generate_bitstreams.tcl`. To adjust which files are created, or to change bitstream generation settings, edit this Tcl file.

Ver1 does not have to be the version that is booted from flash. Rather, this version should be the most challenging design version available. The place and route results of this first version determine the partition pin locations, which locks the routing on the interface between the PCIe core and the user application.

Hardware Operation Details

The initial configuration of the device is no different than the normal Tandem Configuration. Initial configuration of the device consists of two stages, as two separate bitstreams (one loaded using flash, one loaded over PCIe) using the Tandem PCIe approach. After the stage 1 is loaded, only the PCIe IP is operational, with limited functionality as the rest of the design behind it does not yet exist. It is able to link train and be recognized by the Root Port in the system. After the stage 2 bitstream is loaded, the device is in normal operational mode.

Subsequent dynamic updates follow the fundamental rules for UltraScale+ device Dynamic Function eXchange, but using the different verX stage 2 bitstreams as partial bitstreams. When loading over the link, logic decoupling is managed by the PCIe drivers automatically.

Using the bitstream names for a Tandem PCIe design and the supplied scripts, the sequence would look like this:

1. Power on the FPGA. `ver1_tpcie_tandem1.mcs` is sent from local flash.
 - At this point, the PCIe Endpoint is functional, and enumeration can occur.
 - The PCIe core is isolated from the rest of the unconfigured device.
2. Deliver `ver1_tpcie_tandem2.bin` over the PCIe link.
 - The complete device is now programmed, and the design switches automatically to full use mode by establishing communication between PCIe and the rest of the design.
 - Alternately, `ver2_tpcie_update_region_partial.bin` could also be sent over the PCIe link to complete the initial programming of the device.
3. Operate the FPGA for as long as you would like.
4. When a request to update is received, deliver the new user application by sending `ver2_tpcie_update_region_partial.bin` over the PCIe link.
 - The full device is once again operational, now with new functionality.
 - The switchover from isolation of the PCIe to full operation happens automatically.
5. Repeat step 4 to move on to a new version (or return to ver1).

Debugging Tandem with Field Updates Designs

Tandem with Field Updates designs are designs that use Dynamic Function eXchange flows. The Update Region is a Reconfigurable Partition, and each User Application that is inserted into that partition is a Reconfigurable Module. General debug within Reconfigurable Modules is now supported, and the example design that can be generated for any Tandem with Field Updates IP instance contains debug circuitry. The top level of the Reconfigurable Partition (`pci_app_uscale`) includes the boundary scan ports necessary for automatic insertion of the debug hub that permits debug cores to be instantiated within each Reconfigurable Module. Look for the `S_BSCAN` ports but do not change the pin names. For complete details, see Chapter 8 of Vivado Design Suite User Guide: Dynamic Function eXchange ([UG909](#)).

Important Considerations

These considerations are critical for safe and reliable operation of the target device.

- Always be sure that the stage 2 bitstreams are compatible with the current static design in the FPGA before loading them. PR_Verify is a fundamental part of the Dynamic Function eXchange solution and must be used for Tandem with Field Updates for the same reason. PR_Verify confirms that multiple design configurations (i.e., versions) are compatible with each other and therefore safe to overlay in hardware.
- The initial Tandem configuration of the device must be done with a bitstream set compiled as a version within a Tandem with Field Updates flow. If the initial bitstream load has been done with a standard Tandem bitstream set, it will not be compatible with later Field Update partial bitstreams. Contention could occur and device damage is possible.
- Tandem PCIe with Field Updates using Reconfigurable Stage Twos is for UltraScale+ devices only. Field Updates will never support 7 series devices, and the Reconfigurable Stage Twos enhancement will never support UltraScale devices. For this type of approach, a general Dynamic Function eXchange solution should be considered.
- General Tandem + DFX is also supported. This use case allows you to create smaller and/or more numerous Reconfigurable Partitions in a design that utilizes a Tandem initial boot. This flow should be considered more like a standard DFX flow that also has the PCIe IP in the static design. To enable, simply use the standard Tandem PCIe option when generating the PCIe IP, and then add it to a DFX design. This can be compiled in project or non-project mode, with the only unique requirement being that the Pblocks for the respective solutions cannot overlap.

Known Issue and Limitation

- Bitstream compression, which is enabled by default for Tandem Configuration solutions, is not compatible with the Dynamic Function eXchange per-frame CRC feature. If per-frame CRC checking is desired for any of the “update” partial bitstreams, rerun bitstream generation with that feature enabled and bitstream compression disabled. The `write_bitstream -cell` option can be used to create only the partial and clearing bitstreams needed for each design image.

Using Tandem With a User Hardware Design

There are two methods available to apply the Tandem flow to a user design. The first method is to use the example design that comes with the core. The second method is to import the PCIe IP into an existing design and change the hierarchy of the design if required.

Regardless of which method you use, the PCIe example design should be created to get the example clocking structure, timing constraints, and physical block (Pblock) constraints needed for the Tandem solution.

Method 1 – Using the Existing PCI Express Example Design

This is the simplest method in terms of what must be done with the PCI Express core, but might not be feasible for all users. If this approach meets your design structure needs, follow these steps.

1. Create the example design.

Generate the example design as described in [Tandem PROM UltraScale+ Example Tool Flow](#) and [Tandem PCIe UltraScale+ Example Tool Flow](#).

2. Insert the user application.

Replace the PIO example design with the user design. It is recommended that the global and top-level elements, such as I/O and global clocking, be inserted in the top-level design.

3. Uncomment and modify the SPI or BPI flash memory programming settings as required by your board design.

4. Implement the design as normal.

Method 2 – Migrating the PCIe Design into a New Vivado Project

In cases where it is not possible to use Method 1 above, the following steps should be followed to use the PCIe core and the desired Tandem flow (PROM or PCIe) in a new project. The example project has many of the required RTL and scripts that must be migrated into the user design.

1. Create the example design.

Generate the example design as described in [Tandem PROM UltraScale+ Example Tool Flow](#) and [Tandem PCIe UltraScale+ Example Tool Flow](#).

2. Migrate the external GT wizard.

If the Include GT Wizard in example design option is set in the Shared Logic tab during core generation, then the GT Wizard IP is instantiated in the top level of the example design. This GT Wizard IP should be migrated to the user design to provide the necessary GT connections.

3. Migrate the top-level constraint.

The example Xilinx design constraints (XDC) file contains timing constraints, location constraints, and Pblock constraints for the PCIe core. All of these constraints (other than the I/O location and I/O standard constraints) need to be migrated to the user design. Several of the constraints contain hierarchical references that require updating if the hierarchy of the design is different than the example design.

4. Migrate the top-level Pblock constraint.

The following constraint is easy to miss so it is called out specifically in this step. The Pblock constraint should point to the top level of the PCIe core.

```
add_cells_to_pblock [get_pblocks main_pblock_boot] [get_cells -quiet [<path>]]
```



IMPORTANT! Do not make any changes to the physical constraints defined in the XDC file because the constraints are device dependent.

5. Add the Tandem PCIe IP to the Vivado project.

Click **Add Sources** in the Flow Navigator. In the Add Source wizard, select **Add Existing IP** and then browse to the XCI file that was used to create the Tandem PCIe example design.

6. Copy the appropriate SPI or BPI flash memory settings from the example design XDC file and paste them in your design XDC file.
7. Implement the design as normal.

Tandem Configuration RTL Design

Tandem Configuration requires slight modifications from the non-tandem PCI Express® product. This section indicates the additional logic integrated within the core and the additional responsibilities of the user application to implement a Tandem PROM solution.

MUXing Critical Inputs

Certain input ports to the core are multiplexed so that they are disabled during the stage 2 configuration process. These MUXes are controlled by the `mcap_design_switch` signal.

These inputs are held in a deasserted state while the stage 2 bitstream is loaded. This masks off any unwanted glitches due to the absence of stage 2 logic and keeps the PCIe core in a valid state. When `mcap_design_switch` is asserted, the MUXes are switched, and all interface signals behave as described in this document.

TLP Requests

In addition to receiving configuration request packets, the PCI Express Endpoint might receive TLP requests that are not processed within the PCI Express hard block. Typical TLP requests received are Vendor Defined Messages and Read Requests. Before stage 2 is loaded, TLP requests return unsupported requests (URs). After stage 2 has been loaded, the `mcap_design_switch` output is asserted and TLP requests function as defined by the user design.

Tandem Configuration Logic

The core and example design contain ports (signals) specific to Tandem Configuration. These signals provide handshaking between stage 1 (the core) and stage 2 (the user logic). Handshaking is necessary for interaction between the core and the user logic. The following table defines the handshaking ports on the core.

Table 44: Handshaking Ports

Name	Direction	Polarity	Description
<code>mcap_design_switch</code>	Output	Active-High	Identifies when the switch to stage 2 user logic is complete. 0: Stage 2 is not yet loaded. 1: Stage 2 is loaded.

Table 44: Handshaking Ports (cont'd)

Name	Direction	Polarity	Description
cap_req	Output	Active-High	Configuration Access Port arbitration request signal. This signal should be used to arbitrate the use of the FPGA configuration logic between multiple user implemented configuration interfaces. If the Media Configuration Access Port (MCAP) is the only user implemented configuration interface used, this signal should remain unconnected.
cap_rel	Input	Active-High	Configuration Access Port arbitration request for release signal. This signal should be used to arbitrate the use of the FPGA configuration logic between multiple user implemented configuration interfaces. If the MCAP is the only user implemented configuration interface used, this signal should be tied Low (1'b0). This allows the MCAP access to the FPGA configuration logic as needed.
cap_gnt	Input	Active-High	Configuration Access Port arbitration grant signal. This signal should be used to arbitrate the use of the FPGA configuration logic between multiple user implemented configuration interfaces. If the MCAP is the only user implemented configuration interface used, this signal tied High (1'b1). This grants the MCAP access to the FPGA configuration logic upon request.
user_reset	Output	Active-High	Can be used to reset PCIe interfacing logic when the PCIe core is reset. Synchronized with user_clock.
user_clk	Output	N/A	Clock to be used by PCIe interfacing logic.
user_Lnk_up	Output	Active-High	Identifies that the PCIe Express core is linked up with a host device.

These signals can coordinate events in the user application, such as the release of output 3-state buffers described in Tandem Configuration Details. Here is some additional information about these signals:

- `mcap_design_switch` is asserted after stage 2 is loaded. After stage 2 is loaded this output is controlled by the Root Port system. Whenever this signal is deasserted the PCIe solution IP is isolated from the rest of the user design and TLP BAR accesses return Unsupported Requests (URs).
- `cap_req`, `cap_rel`, and `cap_gnt` signals should be used to arbitrate the use of the FPGA configuration logic between multiple configuration interfaces such as the Internal Configuration Access Port (ICAP). The ICAP can be used as part of other IP cores or be instantiated directly in the user design. To arbitrate between the MCAP and the ICAP arbitration, logic must be created and use the `cap_*` signals to allow access to each interface as desired by the user design. The MCAP should always be granted exclusive access to the configuration logic until stage 2 is fully loaded. This is identified by the assertion of the `mcap_design_switch` output. After the initial stage 2 design is loaded the MCAP interface can be used as desired by the system level design. `cap_req` asserts when the Root Port connection requests access to the configuration logic. The user design can grant access by asserting `cap_gnt` in response. The user design can then request that the MCAP release control of the configuration logic by asserting the `cap_rel`. The Root Port connection release control by deasserting `cap_req`. The MCAP should not be accessed if the user logic does not assert `cap_gnt`. Similarly, other configuration interfaces should not attempt to access the configuration logic while access has been granted to the MCAP interface.

- `user_reset` can likewise be used to reset any logic that communicates with the core when the core itself is reset.
- `user_clk` is simply the main internal clock for the PCIe IP core. Use this clock to synchronize any user logic that communicates directly with the core.
- `user_lnk_up`, as the name implies, indicates that the PCIe core is currently running with an established link.

User Application Handshake

An internal completion event must exist within the FPGA for Tandem solutions to perform the hand-off between core control of the PCI Express Block and the user application. [MUXing Critical Inputs](#) explains why this hand-off mechanism is required. When this switch occurs, `mcap_design_switch` is asserted.

Tandem Configuration Details

I/O Behavior

For each I/O that is required for stage 1 of a Tandem Configuration design, the entire bank in which that I/O resides must be configured in the stage 1 bitstream. In addition to this bank, the configuration bank (65) is enabled also, so the following details apply to these two banks (or one, if the reset pin is in the configuration bank). For PCI Express, the only signal needed in the stage 1 design is the `sys_reset` input port. Therefore, any stage 2 I/O in the same I/O bank as `sys_reset` port is also configured with stage 1. Any pins in the same I/O bank as `sys_reset` are unconnected internally, so output pins demonstrate unknown behavior until their internal connections are completed by the stage 2 configuration. Also, components requiring initialization for the stage 2 functionality should not be placed in these I/O banks unless these components are reset by the design after stage 2 is programmed.

If output pins must reside in the same bank as the `sys_reset` pin and their value cannot float prior to stage 2 completion, the following approach can be taken. Use an OBUFT that is held in 3-state between stage 1 completion (when the output becomes active) and stage 2 completion (when the driver logic becomes active). The `mcap_design_switch` signal can be used to control the enable pin, releasing that output when the handshake events complete.



TIP: In your top-level design, infer or instantiate an OBUFT. Control the enable (port named `T`) with `mcap_design_switch` - watch the polarity!

```
OBUFT test_outobuf (.O(test_out), .I(test_internal), .T(!mcap_design_switch));
```

Using the syntax below as an example, create a Pblock to contain the reset pin location. This Pblock should contain the entire bank of I/O along with the associated XiPhy and clocking primitives. The first column of FPGA slice resources should also be included in the Pblock so that it is aligned with partial configuration boundaries. Any logic that should be placed in this region should be added to the Pblock and identified as stage 1 logic using the HD.TANDEM property. It is important to note that this logic becomes active after stage 1 is loaded whereas the driving logic might not become active until stage 2 is loaded. The system design should be created with this consideration in mind. It is recommended that they be grouped together in their own Pblock. The following is an example for an output port named `test_outobuf`.

```
# Create a new Pblock
create_pblock IO_pblock
set_property HD.TANDEM 1 [get_cells <my_cell>]
# Range the Pblock to include the entire IO Bank and the associate XiPhy
and clocking
primitives.
resize_pblock [get_pblocks IO_pblock] -add { \
IOB_X1Y52:IOB_X1Y103 \
SLICE_X86Y60:SLICE_X86Y119 \
MMCME3_ADV_X1Y1 \
PLLE3_ADV_X1Y2:PLLE3_ADV_X1Y3 \
PLL_SELECT_SITE_X1Y8:PLL_SELECT_SITE_X1Y15 \
BITSLICE_CONTROL_X1Y8:BITSLICE_CONTROL_X1Y15 \
BITSLICE_TX_X1Y8:BITSLICE_TX_X1Y15 \
BITSLICE_RX_TX_X1Y52:BITSLICE_RX_TX_X1Y103 \
XIPHY_FEEDTHROUGH_X4Y1:XIPHY_FEEDTHROUGH_X7Y1 \
RIU_OR_X1Y4:RIU_OR_X1Y7 \
}
# Add components and routes to stage 1 external Pblock
# This constraint should be repeated for each primitive within this pblock
region
add_cells_to_pblock [get_pblocks IO_pblock] [get_cells test_outobuf]
# Identify the logic within this pblock as stage1 logic by applying the
HD.TANDEM
property.
# This constraint should be repeated for each primitive within this pblock
region
set_property HD.TANDEM 1 [get_cells test_outobuf]
```

The remaining user I/O in the design are pulled active-High, by default, during the stage 2 configuration. The use of the PUDC_B pin, when held active-High, forces all I/O in banks beyond the three noted above to be in 3-state mode. Between stage 1 and stage 2, which for Tandem PCIe could be a considerable amount of time, these pins are pulled Low by the internal weak pull-down for each I/O as these pins are unconfigured at that time.

Configuration Pin Behavior

The DONE pin indicates completion of configuration with standard approaches. DONE is also used for Tandem Configuration, but in a slightly different manner. DONE pulses High at the end of stage 1, when the start-up sequences are run. It returns Low when stage 2 loading begins. For Tandem PROM, this happens immediately because stage 2 is in the same bit file. For Tandem PCIe, this happens when the second bitstream is delivered to the PCIe MCAP interface. It pulls High and stays High at the end of the stage 2 configuration.

Configuration Persist (Tandem PROM Only)

Tandem PROM configurations do not allow the dual purpose configuration I/O to be used for stage 1 or stage 2 configurations. This prevents access to the external configuration flash for Tandem PROM designs. Because of this, the configuration flash cannot be accessed by the user design after Tandem PROM configuration completes. Tandem PCIe should be used if configuration flash update from the FPGA user design is required. Configuration Persist is set automatically during `write_bitstream`, and is set in such a way that the ICAP (which must remain out of stage 1) is functional in the user design.

If the `CONFIG_MODE` option is set correctly for the needed configuration mode, but necessary dual-mode I/O pins are still occupied by user I/O, the following error is issued for each instance during `write_bitstream`:

```
ERROR: [Designutils 12-1767] Cannot add persist programming for site  
IOB_X0Y151.  
ERROR: [Designutils 12-1767] Cannot add persist programming for site  
IOB_X0Y152.
```

The user I/O occupying these sites must be relocated to use Tandem PROM.

Avoiding the Configuration Bank

Bank 65 contains dedicated and dual-mode configuration pins. Tandem Configuration can use many of these pins, including the dedicated PCIe reset pin (`PERSTN`) in UltraScale+ devices, and many configuration pins, such as `EMCCLK`, `CSI_B` and address and data pins for wider interfaces.



IMPORTANT! Xilinx advises Tandem Configuration users to avoid using bank 65 for design applications, especially when using Tandem PROM, to avoid complications because the programming bitstream is split into two stages. Specifically, IP cores built by the Memory Interface Generator (MIG) must not use bank 65 I/O. This ensures that IP can remain completely within stage 2, and avoid complications with its embedded I/O and demanding timing constraints.

To see the pins required for your desired configuration mode, see the configuration diagrams in the *UltraScale Architecture Configuration User Guide (UG570)*.

PROM Selection

Configuration PROMs have no specific requirements unique to Tandem Configuration. However, to meet the 100 ms specification, you must select a PROM that meets the following three criteria:

1. Supported by Xilinx configuration.
2. Sized appropriately for both stage 1 and stage 2; that is, the PROM must be able to contain the entire bitstream.
 - For Tandem PROM, both stage 1 and stage 2, are stored here; this bitstream is slightly larger (4-5%) than a standard bitstream.

- For Tandem PCIe, the bitstream size is typically about 1 MB, but this can vary slightly due to design implementation results, device selection, and effectiveness of compression.
3. Meets the configuration time requirement for PCI Express based on the first-stage bitstream size and the calculations for the bitstream loading time. See [Calculating Bitstream Load Time for Tandem](#).

See the *UltraScale Architecture Configuration User Guide* ([UG570](#)).

Programming the Device

There are no differences for programming Tandem bitstreams versus standard bitstreams into a PROM. You can program a Tandem bitstream using all standard flash memory programming methods, such as JTAG, Slave and Master SelectMAP, SPI, and BPI. Regardless of the programming method used, the DONE pin is asserted after the first stage is loaded and operation begins.

Note: Do not set the mode pins to 101 for JTAG Only mode. This restricts this ICAP and MCAP capabilities, thus preventing proper stage 2 loading. When programming stage 1 using JTAG and later using the MCAP, the mode pins should be set to 000, 001, or 111.

To prepare for SPI or BPI flash memory programming, the appropriate settings must be enabled prior to bitstream generation. This is done by adding the specific flash memory device settings in the design XDC file, as shown here. Examples can be seen in the constraints generated with the PCI Express example design. Copy the existing (commented) options to meet your board and flash memory programming requirements.

Here are examples for Tandem PROM:

```
# BPI Flash Programming
set_property CONFIG_MODE BPI16 [current_design]
set_property BITSTREAM.CONFIG.BPI_SYNC_MODE Type1 [current_design]
set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_design]
set_property CONFIG_VOLTAGE 1.8 [current_design]
set_property CFGBVS GND [current_design]
```

Both internally generated CCLK and externally provided EMCCLK are supported for SPI and BPI programming. EMCCLK can be used to provide faster configuration rates due to tighter tolerances on the configuration clock. See the *UltraScale Architecture Configuration User Guide* ([UG570](#)).

For more information on configuration in the Vivado® Design Suite, see the *Vivado Design Suite User Guide: Programming and Debugging* ([UG908](#)).

Bitstream Encryption

Bitstream encryption is supported for Tandem Configuration, for both Tandem PROM and Tandem PCIe approaches. For Tandem PCIe, the stage 2 bitstream must remain encrypted using the same key as the stage 1 bitstream, because the MCAP (unlike the ICAP) cannot receive unencrypted bitstreams after an encrypted initial load. Bitstream authentication is only supported for Zynq® devices, not for FPGAs.

Tandem PROM/PCIe Resource Restrictions

The PCIe IP must be isolated from the global chip reset (GSR) that occurs right after the stage 2 bitstream has completed loading into the FPGA. As a result, stage 1 and stage 2 logic cannot reside within the same configuration frames. Configuration frames used by the PCIe IP consist of serial transceivers, I/O, FPGA logic, block RAM, or Clocking, and they (vertically) span a single clock region. The resource restrictions are as follows:

- A GT quad contains four serial transceivers. In a X1 or X2 designs, the entire GT quad is consumed and the unused serial transceivers are not available to the user application. The number of GT quads consumed depends on the GT quad selection made when customizing the core in the Vivado® IDE.
- DCI Cascading between a stage 1 I/O bank and a stage 2 I/O bank is not supported.
- Set the DCI Match_Cycle option to `NoWait` to minimize stage 1 configuration time:

```
set_property BITSTREAM.STARTUP.MATCH_CYCLE NoWait [current_design]
```

Moving the PCIe Reset Pins

In general, to achieve the best (smallest) first-stage bitstream size, you should place the PCIe reset package pin in bank 65 with the other configuration pins. If a new location for the reset pin is needed, you should consider the location for any I/Os that are intended to be configured in stage 1. I/Os that are physically placed a long distance from the core cause extra configuration frames to be included in the first stage. This is due to extra routing resources that are required to include these I/Os in the first stage.

Regardless of where the reset pin is located, bank 65 should still be kept in stage 1. Even if configuration modes such as QSPI are used, the `EMCCLK` is required for the fastest possible configuration, and that dual-mode pin is located in bank 65.

Non-Project Flow

In a non-project environment, the same basic approach as the project environment is used. First, create the IP using the IP catalog as shown in the [Tandem PCIe UltraScale+ Example Tool Flow](#). One of the results of core generation is an `.xci` file, which is a listing of all the core details. This file is used to regenerate all the required design sources.

1. Read in design sources, either the example design or your design.

```
read_verilog <verilog_sources>
read_vhdl <vhdl_sources>
read_xdc <xdc_sources>
```

2. Define the target device.

```
set_property PART <part> [current_project]
```

Note: Even though this is a non-project flow, there is an implied project behind the scenes. This must be done to establish an explicit device before the IP is read in.

3. Read in the PCIe IP.

```
read_ip pcie_ip_0.xci
```

4. Synthesize the design. This step generates the IP sources from the .xci input.

Note: When out of context synthesis is used, you might need to apply the Pblock constraints using a constraints file that is only applied during implementation. This is because some constraints depend on the entire design being combined to apply the constraints.

5. Ensure that any customizations to the design, such as the identification of the configuration mode to set the persisted pins, are done in the design XDC file.

6. Implement the design.

```
opt_design
place_design
route_design
```

7. Generate the bit files. The -bin_file option should be used for Tandem PCIe. The BIN file is aligned to a 32-bit boundary and can facilitate the software loading of the stage 2 bitstream over PCIe.

```
write_bitstream -bin_file <file>.bit
```

Calculating Bitstream Load Time for Tandem

The configuration loading time is a function of the configuration clock frequency and precision, data width of the configuration interface, and bitstream size. The calculation is broken down into three steps:

1. Calculate the minimum clock frequency based on the nominal clock frequency and subtract any variation from the nominal.

$$\text{Minimum Clock Frequency} = \text{Nominal Clock} - \text{Clock Variation}$$

2. Calculate the minimum PROM bandwidth, which is a function of the data bus width, clock frequency, and PROM type. The PROM bandwidth is the minimum clock frequency multiplied by the bus width.

$$\text{PROM Bandwidth} = \text{Minimum Clock Frequency} \times \text{Bus Width}$$

3. Calculate the first-stage bitstream loading time, which is the minimum PROM bandwidth from step 2, divided by the first-stage bitstream size as reported by write_bitstream.

$$\text{Stage 1 Load Time} = (\text{Stage 1 Bitstream Size}) / (\text{PROM Bandwidth})$$

The stage 1 bitstream size, reported by write_bitstream, can be read directly from the terminal or from the log file.

The following is a snippet from the `write_bitstream` log showing the bitstream size for stage 1 in a VU9P device (with default settings, including compression):

```
Creating bitstream...
Tandem stage1 bitstream contains 11822112 bits.
Tandem stage2 bitstream contains 110742368 bits.
Writing bitstream ./xilinx_pcie_ip.bit...
```

These values represent the explicit values of the bitstream stages, whether in one bit file or two. The effects of bitstream compression are reflected in these values.

Example 1

The configuration for Example 1 is:

- Quad SPI flash (x4) operating at 66 MHz ± 200 ppm
- Stage 1 size = 11822112 bits = 11.27 Mb

The steps to calculate the configuration loading time are:

1. Calculate the minimum clock frequency:

$$66 \text{ MHz} \times (1 - 0.0002) = 65.98 \text{ MHz}$$

2. Calculate the minimum PROM bandwidth:

$$4 \text{ bits} \times 65.98 \text{ MHz} = 263.92 \text{ Mb/s}$$

3. Calculate the first-stage bitstream loading time:

$$11.27 \text{ Mb} / 263.92 \text{ Mb/s} = \sim 0.0427 \text{ or } 42.7 \text{ ms}$$

Example 2

The configuration for Example 2 is:

- BPI (x16) Synchronous mode, operating at 50 MHz ± 100 ppm
- Stage 1 size = 11822112 bits = 11.27 Mb

The steps to calculate the configuration loading time are:

1. Calculate the minimum clock frequency:

$$50 \text{ MHz} \times (1 - 0.0001) = 49.995 \text{ MHz}$$

2. Calculate the minimum PROM bandwidth:

$$16 \text{ bits} \times 49.995 \text{ MHz} = 799.92 \text{ Mb/s}$$

3. Calculate the first-stage bitstream loading time:

$$11.27 \text{ Mb} / 799.92 \text{ Mb/s} = \sim 0.0141 \text{ s or } 14.1 \text{ ms}$$

Using Bitstream Compression

Minimizing the stage 1 bitstream size is the ultimate goal of Tandem Configuration, and the use of bitstream compression aids in this effort. This option uses a multi-frame write technique to reduce the size of the bitstream and therefore the configuration time required. The amount of compression varies from design to design. When Tandem is selected, compression is turned on in the IP level constraints. This can be overridden in the user design constraints as desired. The following command can be used to enable or disable bitstream compression.

```
set_property BITSTREAM.GENERAL.COMPRESS <TRUE|FALSE> [current_design]
```

Other Bitstream Load Time Considerations

Bitstream configuration times can also be affected by:

- Power supply ramp times, including any delays due to regulators
- T_{POR} (power on reset)

Power-supply ramp times are design-dependent. Take care to not design in large ramp times or delays. The FPGA power supplies that must be provided to begin FPGA configuration are listed in *UltraScale Architecture Configuration User Guide* ([UG570](#)).

In many cases, the FPGA power supplies can ramp up simultaneously or even slightly before the system power supply. In these cases, the design gains timing margin because the 100 ms does not start counting until the system supplies are stable. Again, this is design-dependent. Systems should be characterized to determine the relationship between FPGA supplies and system supplies.

T_{POR} is 57 ms for standard power ramp rates, and 15 ms for fast ramp rates for UltraScale+ devices. See *Kintex UltraScale+ FPGAs Data Sheet: DC and AC Switching Characteristics* ([DS922](#)), and *Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics* ([DS923](#)).

Consider two cases for Example 1 (Quad SPI flash [x4] operating at $66 \text{ MHz} \pm 200 \text{ ppm}$) from Calculating Bitstream Load Time for Tandem:

- Case 1: Without ATX Supply
- Case 2: With ATX Supply

Assume that the FPGA power supplies ramp to a stable level (2 ms) after the 3.3V and 12V system power supplies. This time difference is called T_{FPGA_PWR} . In this case, because the FPGA supplies ramp after the system supplies, the power supply ramp time takes away from the 100 ms margin.

The equations to test are:

$$T_{POR} + \text{Bitstream Load Time} + T_{FPGA_PWR} < 100 \text{ ms for non-ATX}$$

$$T_{POR} + \text{Bitstream Load Time} + T_{FPGA_PWR} - 100 \text{ ms} < 100 \text{ ms for ATX}$$

- **Case 1: Without ATX Supply:** Because there is no ATX supply, the 100 ms begins counting when the 3.3V and 12 V system supplies reach within 9% and 8% of their nominal voltages, respectively (see the PCI Express Card Electromechanical Specification *PCI-SIG Specifications* (<http://www.pcisig.com/specifications>)).

$$50 \text{ ms (} T_{POR} \text{)} + 42.7 \text{ ms (bitstream time)} + 2 \text{ ms (ramp time)} = 94.7 \text{ ms}$$

$$94.7 \text{ ms} < 100 \text{ ms PCIe standard (okay)}$$

In this case, the margin is 5.3 ms.

- **Case 2: With ATX Supply:** ATX supplies provide a `PWR_OK` signal that indicates when system power supplies are stable. This signal is asserted at least 100 ms after actual supplies are stable. Thus, this extra 100 ms can be added to the timing margin.

$$50 \text{ ms (} T_{POR} \text{)} + 42.7 \text{ ms (bitstream time)} + 2 \text{ ms (ramp time)} - 100 \text{ ms} = -5.3 \text{ ms}$$

$$-5.3 \text{ ms} < 100 \text{ ms PCIe standard (okay)}$$

In this case, the margin is 105.3 ms.

Sample Bitstream Sizes

The final size of the stage 1 bitstream varies based on many factors, including:

- IP: The size and shape of the first-stage Pblocks determine the number of frames required for stage 1. x8 and x16 configurations will require more GT quads in the stage 1 floorplan, which will lead to a larger stage 1 bitstream.
- Device: Wider devices require more routing frames to connect the IP to clocking resources.
- Design: Location of the reset pin is one of many factors introduced by the addition of the user application.
- GT Locations: The selection of the GT quads used affects the size of the stage 1 bitstream. For the most efficient use of resources, the GT quad adjacent to the PCI Express hard block should be used.
- Compression: As the device utilization increases, the effectiveness of compression decreases.

As a baseline, here are some sample bitstream sizes and configuration times for the example (PIO) design generated along with the PCIe IP.

Table 45: Example Bitstream Size and Configuration Times¹

Device	Full Bitstream	Full: BPI16 at 50 MHz	Tandem Stage 1 ²	Tandem: BPI16 at 50 MHz
VU9P	277.3 Mb	346.6 ms	17.6 Mb	22.0 ms
VU9P	346.6 ms	764.5 ms	22.0 ms	21.8 ms

Notes:

1. The configuration times shown here do not include T_{POR} .
2. Because the PIO design is very small, compression is very effective in reducing the bitstream size. These numbers were obtained without compression to give a more accurate estimate of what a full design might show. These numbers were generated using a PCIe Gen3x16 configuration in Vivado® Design Suite 2018.1.

The amount of time it takes to load the stage 2 bitstream using the Tandem PCIe methodology depends on three additional factors:

- The width and speed of PCI Express link.
- The frequency of the clock used to program the MCAP.
- The efficiency at which the Root Port host can deliver the bitstream to the Endpoint FPGA design. For most designs this is the limiting factor.

The lower bandwidth of these three factors determines how fast the stage 2 bitstream is loaded.

Clocking

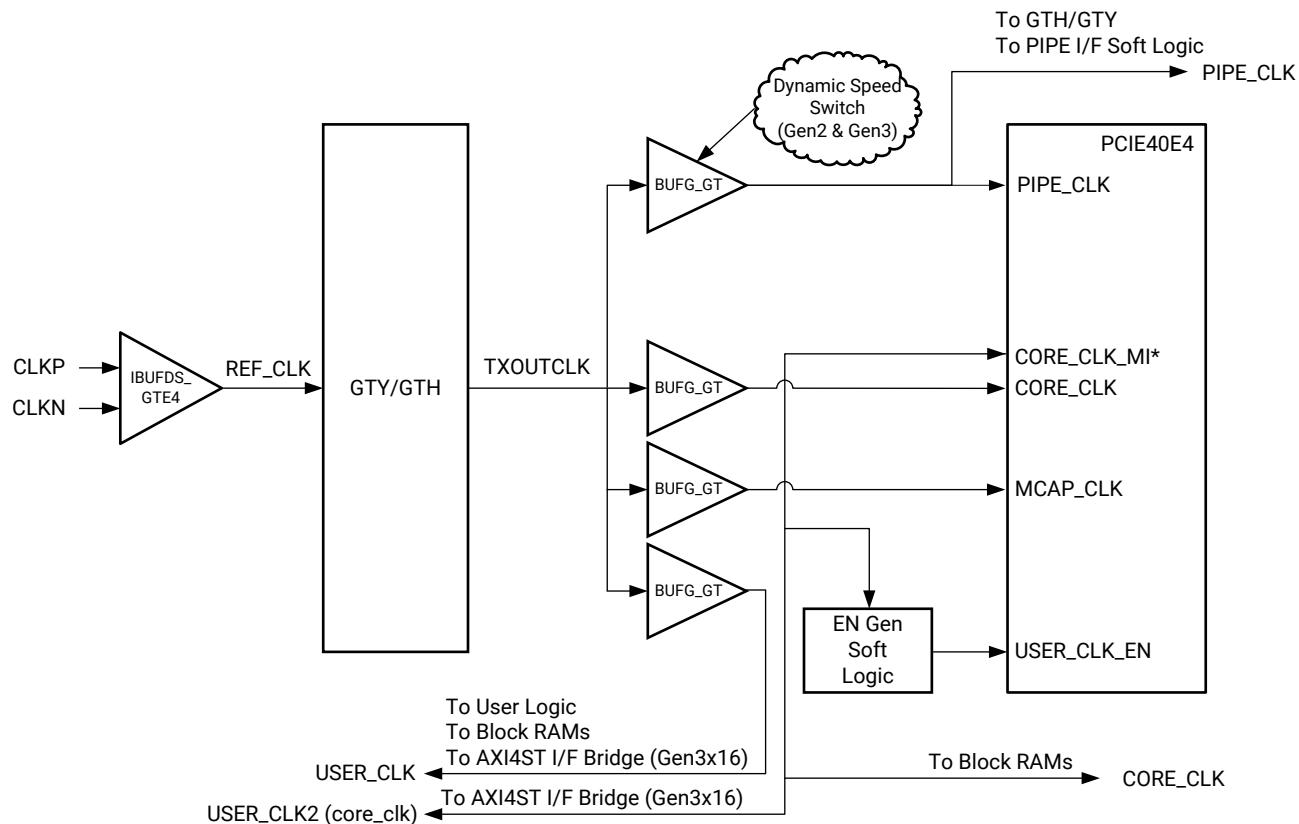
The core requires a 100, 125, or 250 MHz reference clock input. For more information, see the Answer Records at the [Xilinx Solution Center for PCIe](#).

The following applies:

- The reference clock can be synchronous or asynchronous with up to ± 300 PPM or 600 PPM worst case. (If spread spectrum clock (SSC) is enabled, the link must be synchronous.)
- The `PCLK` is the primary clock for the PIPE interface.
- In addition to `PCLK`, two other clocks (`CORECLK` and `USERCLK`) are required to support the core.
- `BUFG_GTs` are used to generate these clocks. These clocks are all driven from the `TXOUTCLK` pin which is a derived clock from `GTREFCLK0` through a CPLL. In an application where QPLL is used, QPLL is only provided to the GT PCS/PMA block while `TXOUTCLK` continues to be derived from a CPLL.

- The source of the UltraScale+™ GTH reference clock must come directly from IBUFDS_GTE4.
- To use the reference clock for FPGA general interconnect, another BUFG_GT must be used.

Figure 12: Clocking Architecture



X15958-091117

All PCIe clocks (pipe_clk, core_clk, user_clk, and mcap_clk) are all driven by BUFG_GT sourced from the TXOUTCLK pin. These clocks are derived clock from GTREFCLK0 through a CPLL. In an application where QPLL is used, QPLL is only provided to the GT PCS/PMA block while TXOUTCLK continues to be derived from a CPLL. All user interface signals of the core are timed with respect to the same clock (user_clk) which can have a frequency of 62.5, 125, or 250 MHz depending on the link speed and width configured (see the previous figure).

In a typical PCI Express® solution, the PCI Express reference clock is a spread spectrum clock (SSC), provided at 100 MHz. In most commercial PCI Express systems, SSC cannot be disabled. For more information regarding SSC and PCI Express, see Section 4.3.7.1.1 of the [PCI Express Base Specification, rev. 3.0](#).



IMPORTANT! All add-in card designs must use synchronous clocking due to SSC on the reference clock of most host systems. For devices using the Slot clock, the Slot Clock Configuration setting in the Link Status register must be enabled in the Vivado® IP catalog.

Each link partner device shares the same clock source. The following figures show a system using a 100 MHz reference clock. Even if the device is part of an embedded system, if the system uses commercial PCI Express root complexes or switches along with typical motherboard clocking schemes, synchronous clocking should still be used.

Note: The clocking diagrams show high-level representations of the board layout. Ensure that coupling, termination, and details are correct when laying out a board. See *UltraScale Architecture GTH Transceivers User Guide (UG576)*.

Figure 13: Embedded System Using 100 MHz Reference Clock

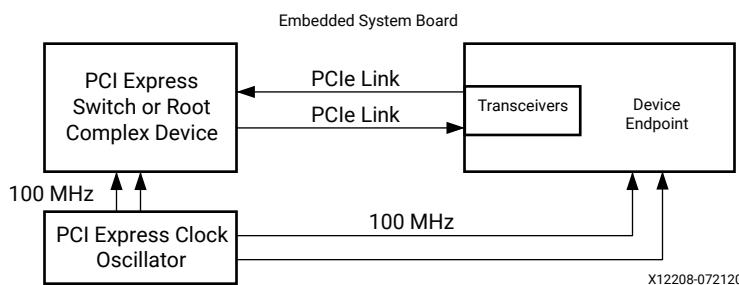
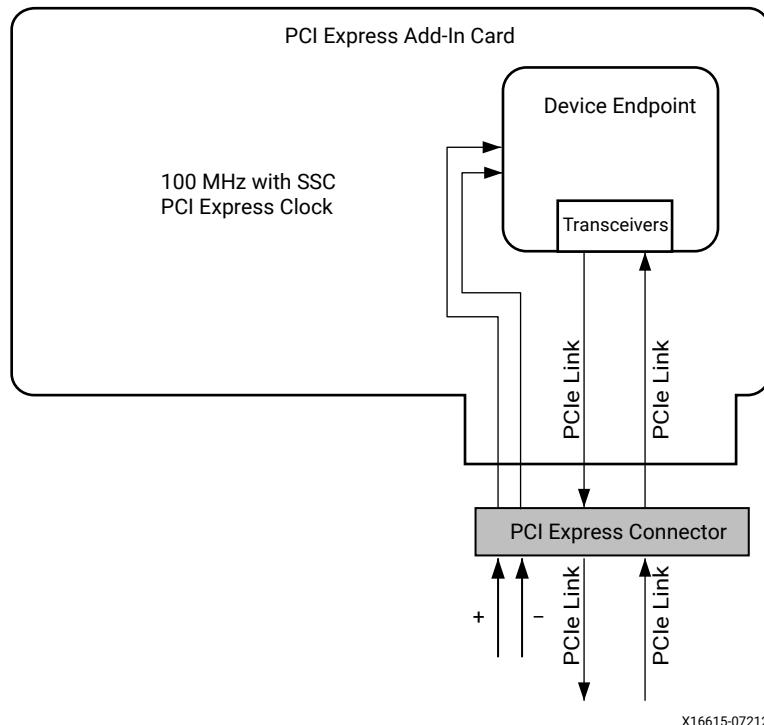


Figure 14: Open System Add-In Card Using 100 MHz Reference Clock



The PCIe core checks for GT power to be stable before the clock is enabled.

- This results in a logic driven CE (rather than VCC) for the BUFG_GT that is driven by IBUFDS_GTE4 (PCIe ref clock).

- Before this change in CE, if you had another (parallel) BUFG_GT connected to the IBUFDS_GTE4 with CE driven by VCC, the BUFG_GT_SYNC inserted by opt_design/MLO could drive both BUFG_GTs.
- If there is a parallel BUFG_GT that does not share the same CE as the PCIe BUFG_GT clock, then two BUFG_GT_SYNC are inserted by opt_design/MLO.
- Because you can only have one BUFG_GT_SYNC for IBUFDS_GTE4 driven BUFG_GTs, the router does not know how to handle the second BUFG_GT_SYNC and does not route the IBUFDS_GTE4/ODIV2 driven clock net.
- You must ensure that the BUFG_GTs driven by the IBUFDS_GTE4 have the same CE/CLR pins.

Resets

The core resets the system using sys_reset, an asynchronous, active-Low reset signal asserted during the PCI Express® Fundamental Reset. Asserting this signal causes a hard reset of the entire core, including the transceivers. After the reset is released, the core attempts to link train and resume normal operation. In a typical Endpoint application, for example an add-in card, a sideband reset signal is normally present and should be connected to sys_reset. For Endpoint applications that do not have a sideband system reset signal, the initial hardware reset should be generated locally. Four reset events can occur in PCI Express:

- **Cold Reset:** A Fundamental Reset that occurs at the application of power. The sys_reset signal is asserted to cause the cold reset of the core.
- **Warm Reset:** A Fundamental Reset triggered by hardware without the removal and reapplication of power. The sys_reset signal is asserted to cause the warm reset to the core.
- **Hot Reset:** In-band propagation of a reset across the PCI Express Link through the protocol, resetting the entire Endpoint device. In this case, sys_reset is not used. In the case of Hot Reset, the cfg_hot_reset_out signal is asserted to indicate the source of the reset.
- **Function-Level Reset:** In-band propagation of a reset across the PCI Express Link through the protocol, resetting only a specific function. In this case, the core asserts the bit of either cfg_flr_in_process and/or cfg_vf_flr_in_process that corresponds to the function being reset. Logic associated with the function being reset must assert the corresponding bit of cfg_flr_done or cfg_vf_flr_done to indicate it has completed the reset process.

After an FLR has been initiated by writing a 1b to the Initiate Function Level Reset bit, the function must complete the FLR and any function-specific initialization within 100 ms.

The User Application interface of the core has an output signal, `user_reset`. This signal is deasserted synchronously with respect to `user_clk`. The `user_reset` signal is asserted as a result of any of these conditions:

- **Fundamental Reset:** Occurs (cold or warm) due to assertion of `sys_reset`.
- **PLL within the Core Wrapper:** Loses lock, indicating an issue with the stability of the clock input.
- **Loss of Transceiver PLL Lock:** Any transceiver loses lock, indicating an issue with the PCI Express Link.

The `user_reset` signal is deasserted synchronously with `user_clk` after all of the listed conditions are resolved, allowing the core to attempt to train and resume normal operation.

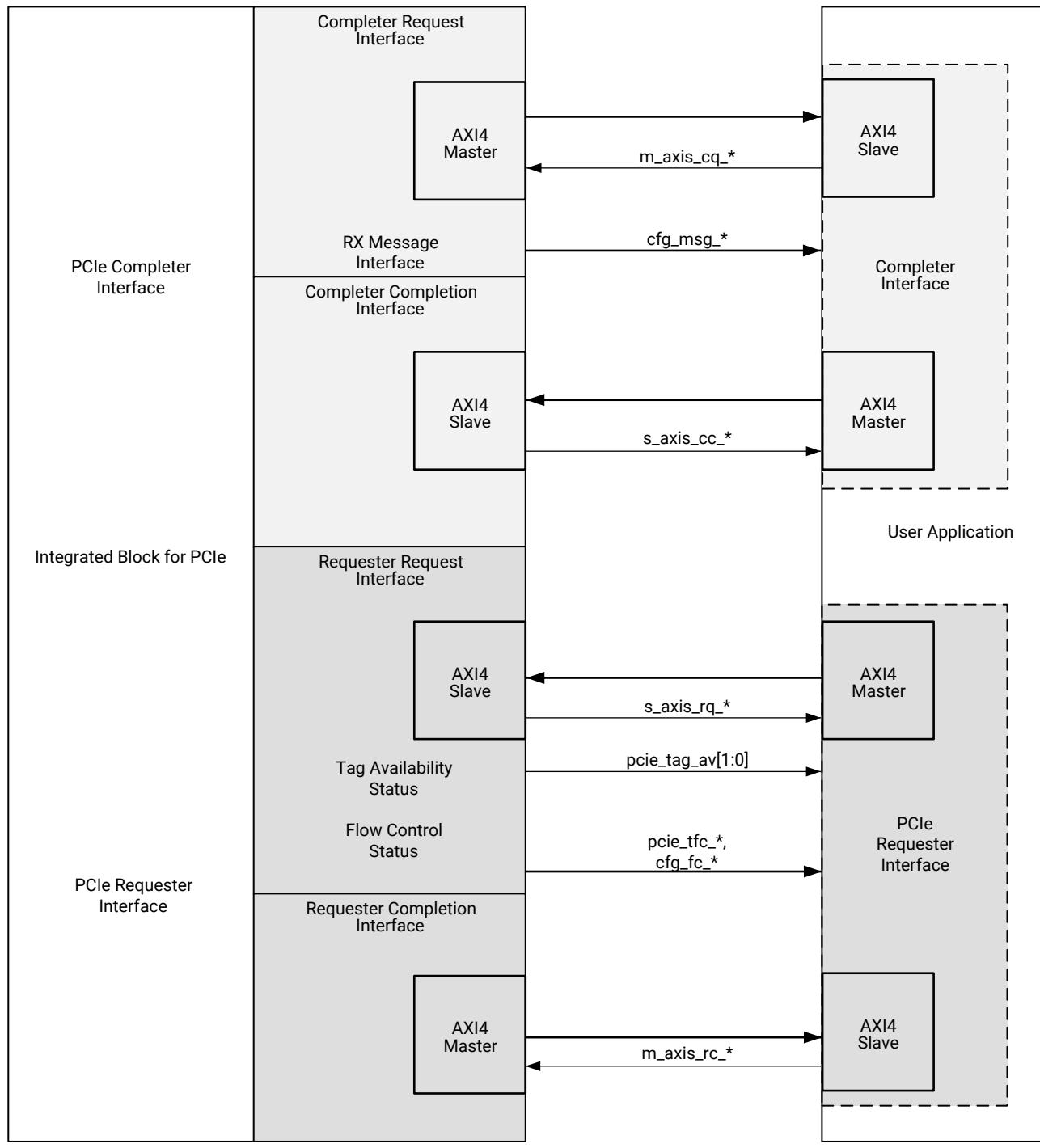
AXI4-Stream Interface Description

This section provides a detailed description of the features, parameters, and signals associated with the user interfaces of the core.

Feature Overview

The following figure illustrates the user interface of the core.

Figure 15: Block Diagram of Integrated Block User Interfaces



X12207-072120

The interface is organized as four separate interfaces through which data can be transferred between the PCIe link and the user application:

- A PCIe Completer Request (CQ) interface through which requests arriving from the link are delivered to the user application.
- A PCIe Completer Completion (CC) interface through which the user application can send back responses to the completer requests. The user application can process all Non-Posted transactions as split transactions. That is, it can continue to accept new requests on the completer request interface while sending a completion for a request.
- A PCIe Requester Request (RQ) interface through which the user application can generate requests to remote PCIe devices attached to the link.
- A PCIe Requester Completion (RC) interface through which the user application receives completions from the link (in response to the user application requests as PCIe requester).

Each of the four interfaces is based on the AMBA® AXI4-Stream Protocol Specification. The width of these interfaces can be configured as 64, 128, 256, or 512 bits, and the user clock frequencies can be selected as 62.5, 125, or 250 MHz, depending on the number of lanes and data rate you choose. Only the Gen3 x16 interface has a data of 512 bits (64 bytes), and operates at a clock frequency of 250 bits, providing a peak transfer rate of 16 GB/s in each direction, adequate to support a Gen3 x16 PCI Express® link.

The following table lists the valid combinations of interface width and user clock frequency for the different link widths and link speeds supported by the integrated block. All four AXI4-Stream interfaces are configured with the same width in all cases.

In addition, the integrated block contains the following interfaces through which status information is communicated to the PCIe master side of the user application:

- A flow control status interface attached to the requester request (RQ) interface that provides information on currently available transmit credit. This enables the user application to schedule requests based on available credit, avoiding blocking in the internal pipeline of the controller due to lack of credit from its link partner.
- A tag availability status interface attached to the requester request (RQ) interface that provides information on the number of tags available to assign to Non-Posted requests. This allows the client to schedule requests without the risk of being blocked when the tag management unit in the PCIe IP has exhausted all the tags available for outgoing Non-Posted requests.
- A receive message interface attached to the completer request (CQ) interface for delivery of message TLPs received from the link. It can optionally provide indications to the user logic when a message is received from the link (instead of transferring the entire message to the user application over the AXI4 interface).

Table 46: Clock Frequencies and Interface Widths Supported For Various Configurations

PCIe Link Speed Capability	PCIe Link Width Capability	PIPE Interface Data Widths (bits)	AXI4 Streaming Interface Data Width (bits)	pipe_clk Frequency (MHz)	core_clk Frequency (MHz)	user_clk2 Frequency (MHz) (axi4st)	user_clk Frequency (MHz) (cfg, axi4st)	mcap_clk Frequency (MHz)	GT TxOutClk (MHz)
Gen1	X1	16	64	125	250	62.5	62.5	62.5/125	250
		16	64	125	250	125	125	125	250
		16	64	125	250	250	250	125	250
	X2	16	64	125	250	62.5	62.5	62.5/125	250
		16	64	125	250	125	125	125	250
		16	64	125	250	250	250	125	250
	X4	16	64	125	250	125	125	125	250
		16	64	125	250	250	250	125	250
	X8	16	64	125	250	250	250	125	250
		16	128	125	250	125	125	125	250
	X16	16	128	125	250	250	250	125	250
Gen2	X1	16	64	125/250	250	62.5	62.5	62.5/125	250
		16	64	125/250	250	125	125	125	250
		16	64	125/250	250	250	250	125	250
	X2	16	64	125/250	250	125	125	125	250
		16	64	125/250	250	250	250	125	250
	X4	16	64	125/250	250	250	250	125	250
		16	128	125/250	250	125	125	125	250
	X8	16	128	125/250	250	250	250	125	250
		16	256	125/250	250	125	125	125	250
	X16	16	256	125/250	250	250	250	125	250
Gen3	X1	16/32	64	125/250	250	125	125	125	250
		16/32	64	125/250	250	250	250	125	250
	X2	16/32	64	125/250	250	250	250	125	250
		16/32	128	125/250	250	125	125	125	250
	X4	16/32	128	125/250	250	250	250	125	250
		16/32	256	125/250	250	125	125	125	250
	X8	16/32	256	125/250	500	250	250	125	500
	X16	16/32	512	125/250	500	500	250	125	500
Gen4	X1	16/32/64	64	125/250	250	250	250	125	500
		16/32/64	128	125/250	250	125	125	125	500
	X2	16/32/64	128	125/250	250	250	250	125	500
		16/32/64	256	125/250	250	125	125	125	500
	X4	16/32/64	256	125/250	250	250	250	125	500
	X8	16/32/64	512	125/250	500	500	250	125	500

Data Alignment Options

A transaction layer packet (TLP) is transferred on each of the AXI4-Stream interfaces as a descriptor followed by payload data (when the TLP has a payload). The descriptor has a fixed size of 16 bytes on the request interfaces and 12 bytes on the completion interfaces. On its transmit side (towards the link), the integrated block assembles the TLP header from the parameters supplied by the user application in the descriptor. On its receive side (towards the user interface), the integrated block extracts parameters from the headers of received TLP and constructs the descriptors for delivering to the user application. Each TLP is transferred as a packet, as defined in the AXI4-Stream Interface protocol.

64/128/256-bit interface:

When a payload is present, there are two options for aligning the first byte of the payload with respect to the datapath.

1. Dword-aligned mode: In this mode, the descriptor bytes are followed immediately by the payload bytes in the next Dword position, whenever a payload is present.
2. Address-Aligned Mode: In this mode, the payload can begin at any byte position on the datapath. For data transferred from the integrated block to the user application, the position of the first byte is determined as

$$n = A \bmod w$$

where A is the memory or I/O address specified in the descriptor (for message and configuration requests, the address is taken as 0), and w is the configured width of the data bus in bytes. Any gap between the end of the descriptor and the start of the first byte of the payload is filled with null bytes.

For data transferred from the integrated block to the user application, the data alignment is determined based on the starting address where the data block is destined to in user memory. For data transferred from the user application to the integrated block, the user application must explicitly communicate the position of the first byte to the integrated block using the tuser sideband signals when the address-aligned mode is in use.

In the address-aligned mode, the payload and descriptor are not allowed to overlap. That is, the transmitter begins a new beat to start the transfer of the payload after it has transmitted the descriptor. The transmitter fills any gaps between the last byte of the descriptor and the first byte of the payload with null bytes.

512-bit interface:

When a payload is present, there are two options for aligning the first byte of the payload with respect to the data path.

1. Dword-aligned Mode: In this mode, the descriptor bytes are followed immediately by the payload bytes in the next Dword position, whenever a payload is present. If D is the size of the descriptor in bytes, the lane number corresponding to the first byte of the payload is determined as:

$$n = (S + D + (A \bmod 4)) \bmod 64$$

where S is the lane number where the first byte of the descriptor appears (which can be 0, 16, 32 or 48), D is the width of the descriptor (which can be 12 or 16 bytes), and A is the address of the first byte of the data block in user memory (for message and configuration requests, the address is taken as 0).

2. 128b Address-aligned Mode: In this mode, the start of the payload on the 512-bit bus is aligned on a 128-bit boundary. The lane number corresponding to the first byte of the payload is determined as:

$$n = (S + 16 + (A \bmod 16)) \bmod 64$$

where S is the lane number where the first byte of the descriptor appears (which can be 0, 16, 32 or 48) and A is the memory or I/O address corresponding to the first byte of the payload (for message and configuration requests, the address is taken as 0). Any gap between the end of the descriptor and the start of the first byte of the payload is filled with null bytes.

The source of address A used for alignment of the data varies among the four user interfaces, as described below:

- **CQ Interface:** For data transferred from the core to the user application over the CQ interface, the address bits used for alignment are the lower address specified in the descriptor, which is the starting address of the data block in user memory.
- **CC Interface:** For Completion data transferred from the user application to the core over the CC interface, the alignment is based on address bits supplied by the user in the descriptor.
- **RQ Interface:** For memory requests transferred from the user application to the core over the RQ interface, the alignment is based on address bits supplied by the user alongside the request using sideband signals. The user may specify any value for A, independent of the setting of the address field in the descriptor.
- **RC Interface:** For Completion data transferred from the core to the user application over the RC interface, the alignment is based on address bits supplied by the user along with the request using sideband signals when it was issued on the RQ interface. The core saves the alignment information from the request and uses it to align the payload of the corresponding Completion when delivering the Completion payload over the RC interface.

The 128b address-aligned mode divides the 512-bit AXI beat into four sub-beats of 128 bits each. The payload can begin only in the sub-beat following the descriptor. The payload and the descriptor are not allowed to overlap in the same sub-beat. The transmitter fills any gaps between the last byte of the descriptor and the first byte of the payload with null bytes.

The alignment mode can be selected independently for requester (RQ, RC) and completer (CQ, CC) interfaces by setting the IP customization GUI.

Note: If performance is a critical factor in the design, dword aligned mode should be used instead of address aligned mode.

The Vivado® IP catalog applies the data alignment option globally to all four interfaces. However, advanced users can select the alignment mode independently for each of the four AXI4-Stream interfaces. This is done by setting the corresponding alignment mode parameter. See [64/128/256-Bit Completer Interface](#) and [512-Bit Completer Interface](#) for more details on address alignment and example diagrams.

Straddle Option on CQ, CC, and RQ Interfaces

The CQ, CC and RQ interfaces have a straddle option that allows up to two TLPs to be transferred over the interface in the same beat. This improves the throughput for small TLPs, as well as when TLPs end in the first half a beat. Straddle can be enabled independently for each of these interfaces during core configuration in the Vivado® IDE. The straddle option can be used with the Dword-aligned mode only.

Straddle Option on RC Interface

The RC interface supports a straddle option that allows up to four TLPs to be transferred over the interface in the same beat. This option can be enabled during core configuration in the Vivado® IDE. When enabled, the core may start a new Completion TLP on byte lanes 0, 16, 32, or 48. Thus, with this option enabled, it is possible for the core to send four Completion TLPs entirely in the same beat on the AXI bus, if each of them has a payload of size one Dword or less. The straddle option can only be used when the RC interface is configured in the Dword-aligned mode.

When the Requester Completion (RC) interface is configured for a width of 256 or 512 bits, depending on the type of TLP and Payload size, there can be significant interface utilization inefficiencies, if a maximum of 1 TLP for 256 bits or 2 TLPs for 512 bits is allowed to start or end per interface beat. This inefficient use of RC interface can lead to overflow of the completion FIFO when Infinite Receiver Credits are advertized. You must either:

- Restrict the number of outstanding Non Posted requests, so as to keep the total number of completions received less than 64 and within the completion of the FIFO size selected, or
- Use the RC interface straddle option. See the waveform figures for 256 bits ([Figure 68: Transfer of Completion TLPs on the Requester Completion Interface with the Straddle Option Enabled](#)) and 512 bits ([Figure 100: Transfer of Completion TLPs on the Requester Completion Interface with the Straddle Option Enabled](#)), respectively showing this option.

The straddle option, available only on the 256-bit or 512-bit wide RC interface, is enabled through the Vivado IP catalog. See [Chapter 5: Design Flow Steps](#) for instructions on enabling the option in the IP catalog. When this option is enabled, the integrated block can start a new Completion TLP on byte lane 16/32/48 when the previous TLP has ended at or before byte lane 15/31/47 in the same beat. Thus, with this option enabled, it is possible for the integrated block to send multiple Completion TLPs entirely in the same beat on the RC interface, if neither of them has more than one Dword of payload.

The straddle setting is only available when the interface width is set to 256 bits or 512 bits, and the RC interface is set to Dword-aligned mode.

The following table lists the valid combinations of interface width, addressing mode, and the straddle option.

Table 47: Valid Combinations of Interface Width, Alignment Mode, and Straddle

Interface Width	Alignment Mode	Straddle Option	Description
64 bits	Dword-aligned	Not applicable	64-bit, Dword-aligned
64 bits	Address-aligned	Not applicable	64-bit, Address-aligned
128 bits	Dword-aligned	Not applicable	128-bit, Dword-aligned
128 bits	Address-aligned	Not applicable	128-bit, Address-aligned
256 bits	Dword-aligned	Disabled	256-bit, Dword-aligned, straddle disabled
256 bits	Dword-aligned	Enabled	256-bit, Dword-aligned, straddle enabled (only allowed for the Requester Completion interface)
256 bits	Address-aligned	Not applicable	256-bit, Address-aligned
512 bits	Dword-aligned	Disabled	512-bit, Dword-aligned, straddle disabled
512 bits	Dword-aligned	Enabled	512-bit, Dword-aligned, straddle enabled (2-TLP straddle allowed for all interfaces, 4-TLP straddle only allowed for the Requester Completion interface)
512 bits	Address-aligned	Not applicable	512-bit, 128-bit Address-aligned

Receive Transaction Ordering

The core contains logic on its receive side to ensure that TLPs received from the link and delivered on its completer request interface and requester completion interface do not violate the PCI Express® transaction ordering constraints. The ordering actions performed by the integrated block are based on the following key rules:

- Posted requests must be able to pass Non-Posted requests on the Completer reQuest (CQ) interface. To enable this capability, the integrated block implements a flow control mechanism on the CQ interface through which user logic can control the flow of Non-Posted requests without affecting Posted requests. The user logic signals the availability of a buffer to receive a Non-Posted request by asserting the `pcie_cq_np_req[0]` signal.

The integrated block delivers a Non-Posted request to the user application only when the available credit is non-zero. The integrated block continues to deliver Posted requests while the delivery of Non-Posted requests has been paused for lack of credit. When no back pressure is applied by the credit mechanism for the delivery of Non-Posted requests, the integrated block delivers Posted and Non-Posted requests in the same order as received from the link. For more information on controlling the flow of Non-Posted requests, see [Selective Flow Control for Non-Posted Requests](#).

- PCIe ordering requires that a completion TLP not be allowed to pass a Posted request, except in the following cases:
 - Completions with the Relaxed Ordering attribute bit set can pass Posted requests.
 - Completions with the ID-based ordering bit set can pass a Posted request if the Completer ID is different from the Posted Requester ID.

The integrated block does not start the transfer of a Completion TLP received from the link on the Requester Completion (RC) interface until it has completely transferred all Posted TLPs that arrived before it, unless one of the two rules applies.

After a TLP has been transferred completely to the user interface, it is the responsibility of the user application to enforce ordering constraints whenever needed.

Table 48: Receive Ordering Rules

Row Pass	Posted	Non-Posted	Completion
Posted	No	Yes	Yes
Non-Posted	No	No	Yes
Completion	a) No b) Yes (Relaxing Ordering) c) Yes (ID Based Ordering)	Yes	No

Transmit Transaction Ordering

On the transmit side, the integrated block receives TLPs on two different interfaces: the Requester reQuest (RQ) interface and the Completer Completion (CC) interface. The integrated block does not reorder transactions received from each of these interfaces. It is difficult to predict how the requester-side requests and completer-side completions are ordered in the transmit pipeline of the integrated block, after these have been multiplexed into a single traffic stream. In cases where completion TLPs must maintain ordering with respect to requests, user logic can supply a 4-bit sequence number with any request that needs to maintain strict ordering with respect to a Completion transmitted from the CC interface, on the `seq_num[3 : 0]` inputs within the `s_axis_rq_tuser` bus. The integrated block places this sequence number on its `pcie_rq_seq_num[3 : 0]` output and asserts `pcie_rq_seq_num_vld` when the request TLP has reached a point in the transmit pipeline at which no new completion TLP from the user application can pass it. This mechanism can be used in the following situations to maintain TLP order:

- The user logic requires ordering to be maintained between a request TLP and a completion TLP that follows it. In this case, user logic must wait for the sequence number of the requester request to appear on the `pcie_rq_seq_num[3:0]` output before starting the transfer of the completion TLP on the target completion interface.
- The user logic requires ordering to be maintained between a request TLP and MSI/MSI-X TLP signaled through the MSI Message interface. In this case, the user logic must wait for the sequence number of the requester request to appear on the `pcie_rq_seq_num[3:0]` output before signaling MSI or MSI-X on the MSI Message interface.

64/128/256-Bit Completer Interface

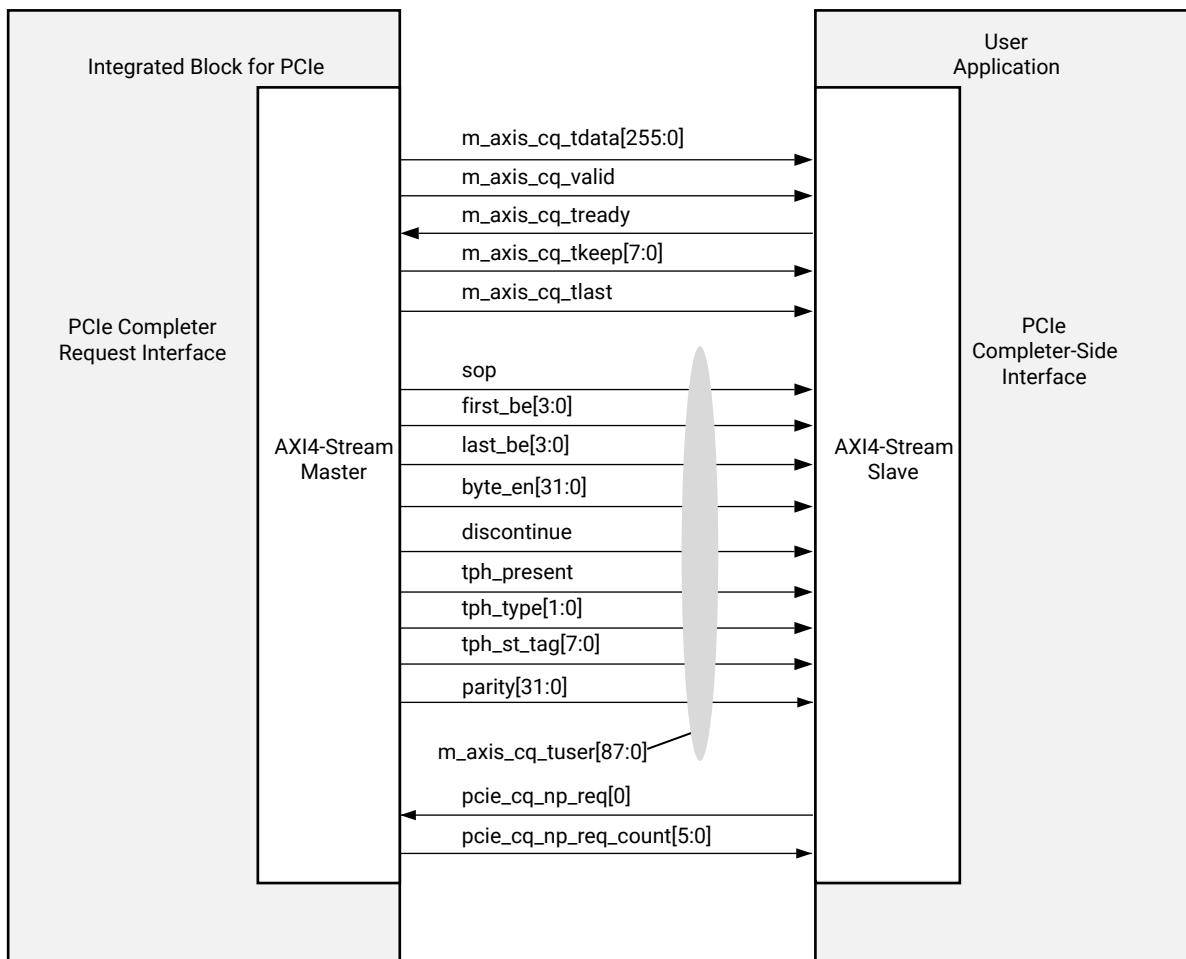
This section describes the operation of the user interfaces of the core for 64/128/256-bit interfaces.

This interface maps the transactions (memory, I/O read/write, messages, Atomic Operations) received from the PCIe link into transactions on the Completer reQuest (CQ) interface based on the AXI4-Stream protocol. The completer interface consists of two separate interfaces, one for data transfers in each direction. Each interface is based on the AXI4-Stream protocol, and its width can be configured as 64, 128, or 256 bits. The CQ interface is for transfer of requests (with any associated payload data) to the user application, and the Completer Completion (CC) interface is for transferring the Completion data (for a Non-Posted request) from the user application for forwarding on the link. The two interfaces operate independently. That is, the integrated block can transfer new requests over the CQ interface while receiving a Completion for a previous request.

Completer Request Interface Operation

The following figure illustrates the signals associated with the completer request interface of the core. The core delivers each TLP on this interface as an AXI4-Stream packet. The packet starts with a 128-bit descriptor, followed by data in the case of TLPs with a payload.

Figure 16: Completer Request Interface Signals



X19416-072420

The completer request interface supports two distinct data alignment modes. In the Dword-aligned mode, the first byte of valid data appears in lane $n = (16 + A \bmod 4) \bmod w$, where A is the byte-level starting address of the data block being transferred, and w is the width of the interface in bytes.

In the address-aligned mode, the data always starts in a new beat after the descriptor has ended, and its first valid byte is on lane $n = A \bmod w$, where w is the width of the interface in bytes.

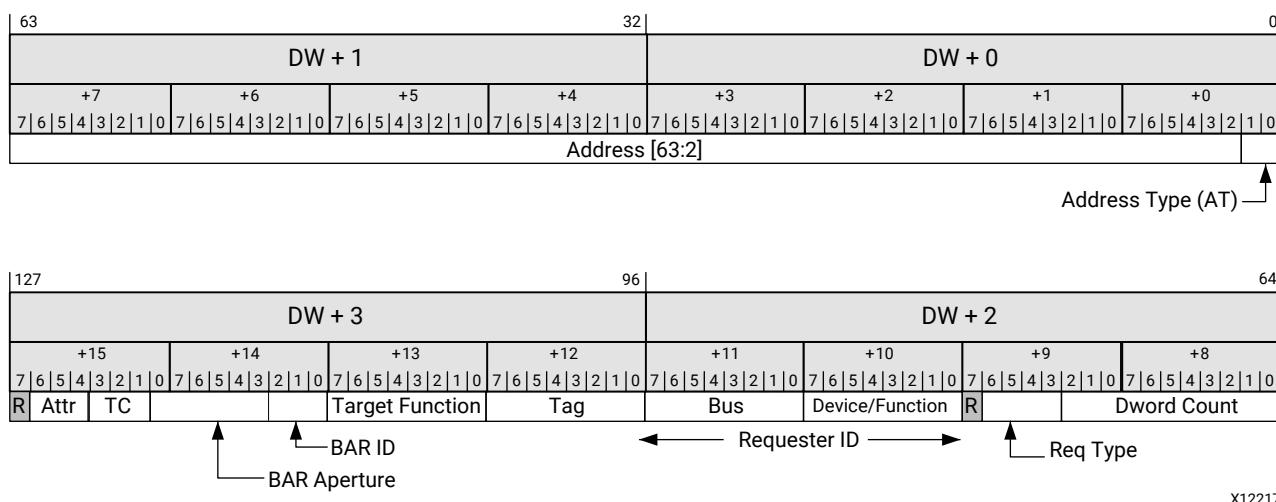
For memory, I/O, and Atomic Operation requests, address A is the address contained in the request. For messages, the address is always taken as 0 for the purpose of determining the alignment of its payload.

Completer Request Descriptor Formats

The integrated block transfers each request TLP received from the link over the CQ interface as an independent AXI4-Stream packet. Each packet starts with a descriptor and can have payload data following the descriptor. The descriptor is always 16 bytes long, and is sent in the first 16 bytes of the request packet. The descriptor is transferred during the first two beats on a 64-bit interface, and in the first beat on a 128-bit or 256-bit interface. The formats of the descriptor for different request types are illustrated in the following figures.

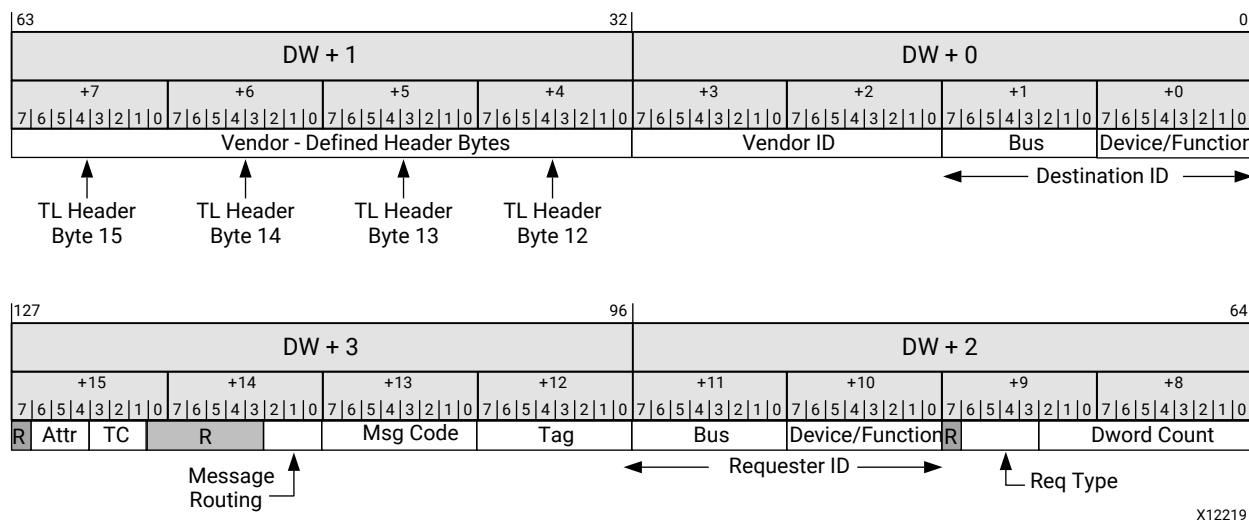
The format of the following figure applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request.

Figure 17: Completer Request Descriptor Format for Memory, I/O, and Atomic Op Requests



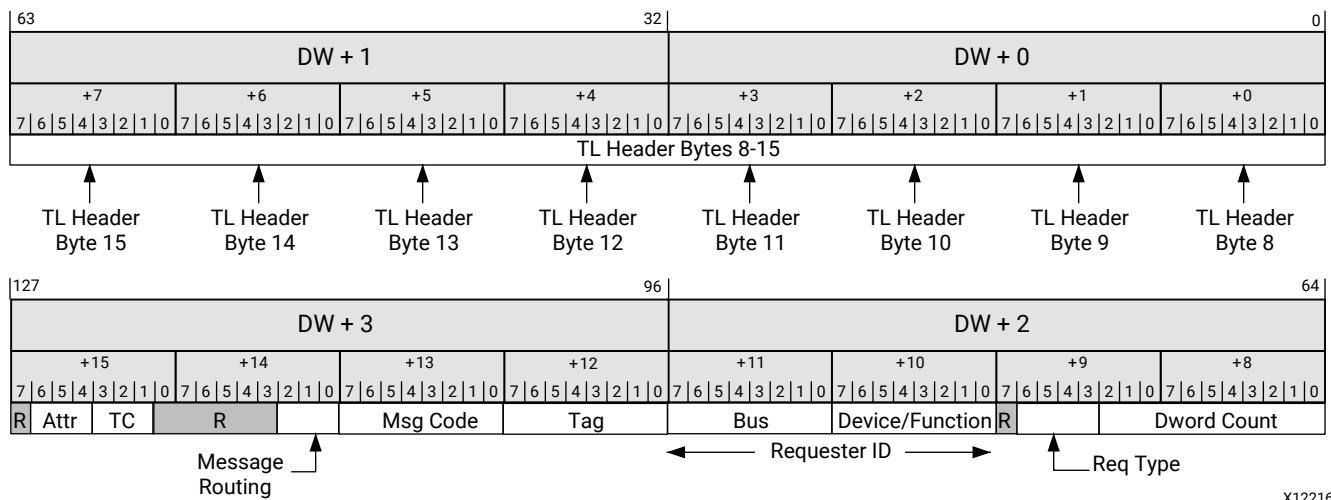
The format of the following figure is used for Vendor-Defined Messages (Type 0 or Type 1) only.

Figure 18: Completer Request Descriptor Format for Vendor-Defined Messages



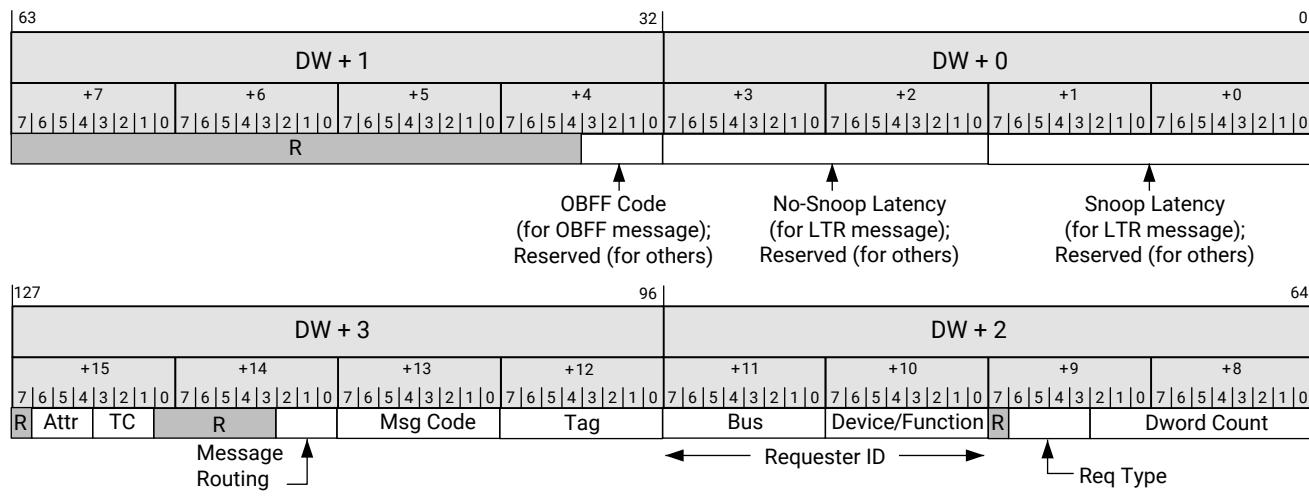
The format of the following figure is used for all ATS messages (Invalid Request, Invalid Completion, Page Request, PRG Response).

Figure 19: Completer Request Descriptor Format for ATS Messages



For all other messages, the descriptor takes the format of the following figure.

Figure 20: Completer Request Descriptor Format for All Other Messages



The following table describes the individual fields of the completer request descriptor.

Table 49: Completer Request Descriptor Fields

Bit Index	Field Name	Description
1:0	Address Type	This field is defined for memory transactions and Atomic Operations only. It contains the AT bits extracted from the TL header of the request. <ul style="list-style-type: none"> • 00: Address in the request is untranslated • 01: Transaction is a Translation Request • 10: Address in the request is a translated address • 11: Reserved
63:2	Address	This field applies to memory, I/O, and Atomic Op requests. It provides the address from the TLP header. This is the address of the first Dword referenced by the request. The First_BE bits from m_axis_cq_tuser must be used to determine the byte-level address. When the transaction specifies a 32-bit address, bits [63:32] of this field are 0.
74:64	Dword Count	These 11 bits indicate the size of the block (in Dwords) to be read or written (for messages, size of the message payload). Its range is 0 - 256 Dwords. For I/O accesses, the Dword count is always 1. For a zero length memory read/write request, the Dword count is 1, with the First_BE bits set to all 0s.
78:75	Request Type	Identifies the transaction type. The transaction types and their encodings are listed in Table 50: Transaction Types .
95:80	Requester ID	PCI Requester ID associated with the request. With legacy interpretation of RIDs, these 16 bits are divided into an 8-bit bus number [95:88], 5-bit device number [87:83], and 3-bit Function number [82:80]. When ARI is enabled, bits [95:88] carry the 8-bit bus number and [87:80] provide the Function number. When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the integrated block with the completion data.

Table 49: Completer Request Descriptor Fields (cont'd)

Bit Index	Field Name	Description
103:96	Tag	PCIe Tag associated with the request. When the request is a Non-Posted transaction, the user logic must store this field and supply it back to the integrated block with the completion data. This field can be ignored for memory writes and messages.
111:104	Target Function	<p>This field is defined for memory, I/O, and Atomic Op requests only. It provides the Function number the request is targeted at, determined by the BAR check. When ARI is in use, all 8 bits of this field are valid. Otherwise, only bits [106:104] are valid.</p> <p>Following are Target Function Value to PF/VF map mappings:</p> <ul style="list-style-type: none"> • 0: PF0 • 1: PF1 • 2: PF2 • 3: PF3 • 4: VF0 • 5: VF1 • 6: VF2 • 7: VF3
114:112	BAR ID	<p>This field is defined for memory, I/O, and Atomic Op requests only. It provides the matching BAR number for the address in the request. In RP mode, BAR ID is always 000.</p> <ul style="list-style-type: none"> • 001: BAR 1 (VF-BAR 1 for VFs) • 010: BAR 2 (VF-BAR 2 for VFs) • 011: BAR 3 (VF-BAR 3 for VFs) • 100: BAR 4 (VF-BAR 4 for VFs) • 101: BAR 5 (VF-BAR 5 for VFs) • 110: Expansion ROM Access <p>For 64-bit transactions, the BAR number is given as the lower address of the matching pair of BARs (that is, 0, 2, or 4).</p>
120:115	BAR Aperture	<p>This 6-bit field is defined for memory, I/O, and Atomic Op requests only. It provides the aperture setting of the BAR matching the request. This information is useful in determining the bits to be used in addressing its memory or I/O space. For example, a value of 12 indicates that the aperture of the matching BAR is 4K, and the user application can therefore ignore bits [63:12] of the address.</p> <p>For VF BARs, the value provided on this output is based on the memory space consumed by a single VF covered by the BAR.</p>
123:121	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request. When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the integrated block with the completion data.
126:124	Attributes	<p>These bits provide the setting of the Attribute bits associated with the request. Bit 124 is the No Snoop bit and bit 125 is the Relaxed Ordering bit. Bit 126 is the ID-Based Ordering bit, and can be set only for memory requests and messages.</p> <p>When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the integrated block with the completion data.</p>
15:0	Snoop Latency	This field is defined for LTR messages only. It provides the value of the 16-bit Snoop Latency field in the TLP header of the message.
31:16	No-Snoop Latency	This field is defined for LTR messages only. It provides the value of the 16-bit No-Snoop Latency field in the TLP header of the message.

Table 49: Completer Request Descriptor Fields (cont'd)

Bit Index	Field Name	Description
35:32	OBFF Code	<p>This field is defined for OBFF messages only. The OBFF Code field is used to distinguish between various OBFF cases:</p> <ul style="list-style-type: none"> • 1111b: CPU Active – System fully active for all device actions including bus mastering and interrupts. • 0001b: OBFF – System memory path available for device memory read/write bus master activities. • 0000b: Idle – System in an idle, low power state. • All other codes are reserved.
111:104	Message Code	<p>This field is defined for all messages. It contains the 8-bit Message Code extracted from the TLP header.</p> <p>Appendix F of the PCI Express Base Specification, rev. 3.0 provides a complete list of the supported Message Codes.</p> <p>Users should treat a descriptor with unsupported Message Code as UR, and toggle the signal cfg_err_uncor_in to indicate that Non-fatal error is detected.</p>
114:112	Message Routing	This field is defined for all messages. These bits provide the 3-bit Routing field r[2:0] from the TLP header.
15:0	Destination ID	This field applies to Vendor-Defined Messages only. When the message is routed by ID (that is, when the Message Routing field is 010 binary), this field provides the Destination ID of the message.
63:32	Vendor-Defined Header	This field applies to Vendor-Defined Messages only. It contains the bytes extracted from Dword 3 of the TLP header.
63:0	ATS Header	This field is applicable to ATS messages only. It contains the bytes extracted from Dwords 2 and 3 of the TLP header.

Table 50: Transaction Types

Request Type (binary)	Description
0000	Memory Read Request
0001	Memory Write Request
0010	I/O Read Request
0011	I/O Write Request
0100	Memory Fetch and Add Request
0101	Memory Unconditional Swap Request
0110	Memory Compare and Swap Request
0111	Locked Read Request (allowed only in Legacy Devices)
1000	Type 0 Configuration Read Request (on Requester side only)
1001	Type 1 Configuration Read Request (on Requester side only)
1010	Type 0 Configuration Write Request (on Requester side only)
1011	Type 1 Configuration Write Request (on Requester side only)
1100	Any message, except ATS and Vendor-Defined Messages
1101	Vendor-Defined Message
1110	ATS Message
1111	Reserved

Completer Memory Write Operation

The following timing diagrams illustrate the Dword-aligned transfer of a memory write TLP received from the link across the Completer reQuest (CQ) interface, when the interface width is configured as 64, 128, and 256 bits, respectively. For illustration purposes, the starting Dword address of the data block being written into memory is assumed to be $(m \times 32 + 1)$, for an integer $m > 0$. Its size is assumed to be n Dwords, for some $n = k \times 32 + 29, k > 0$.

In both Dword-aligned and address-aligned modes, the transfer starts with the 16 descriptor bytes, followed immediately by the payload bytes. The `m_axis_cq_tvalid` signal remains asserted over the duration of the packet. You can prolong a beat at any time by deasserting `m_axis_cq_tready`. The AXI4-Stream interface signals `m_axis_cq_tkeep` (one per Dword position) indicate the valid Dwords in the packet including the descriptor and any null bytes inserted between the descriptor and the payload. That is, the `tkeep` bits are set to 1 contiguously from the first Dword of the descriptor until the last Dword of the payload. During the transfer of a packet, the `tkeep` bits can be 0 only in the last beat of the packet, when the packet does not fill the entire width of the interface. The `m_axis_cq_tlast` signal is always asserted in the last beat of the packet.

The CQ interface also includes the First Byte Enable and the Last Enable bits in the `m_axis_cq_tuser` bus. These are valid in the first beat of the packet, and specify the valid bytes of the first and last Dwords of payload.

The `m_axis_cq_tuser` bus also provides several informational signals that can be used to simplify the logic associated with the user interface, or to support additional features. The `sop` signal is asserted in the first beat of every packet, when its descriptor is on the bus. The byte enable outputs `byte_en[31:0]` (one per byte lane) indicate the valid bytes in the payload. The bits of `byte_en` are asserted only when a valid payload byte is in the corresponding lane (that is, not asserted for descriptor or padding bytes between the descriptor and payload). The asserted byte enable bits are always contiguous from the start of the payload, except when the payload size is two Dwords or less. For cases of one-Dword and two-Dword writes, the byte enables can be non-contiguous. Another special case is that of a zero-length memory write, when the integrated block transfers a one-Dword payload with all `byte_en` bits set to 0. Thus, in all cases the user logic can use the `byte_en` signals directly to enable the writing of the associated bytes into memory.

In the Dword-aligned mode, there can be a gap of zero, one, two, or three byte positions between the end of the descriptor and the first payload byte, based on the address of the first valid byte of the payload. The actual position of the first valid byte in the payload can be determined either from `first_be[3:0]` or `byte_en[31:0]` in the `m_axis_cq_tuser` bus.

When a Transaction Processing Hint is present in the received TLP, the integrated block transfers the parameters associated with the hint (TPH Steering Tag and Steering Tag Type) on signals within the `m_axis_cq_tuser` bus.

Figure 21: Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode, 64-Bit Interface)

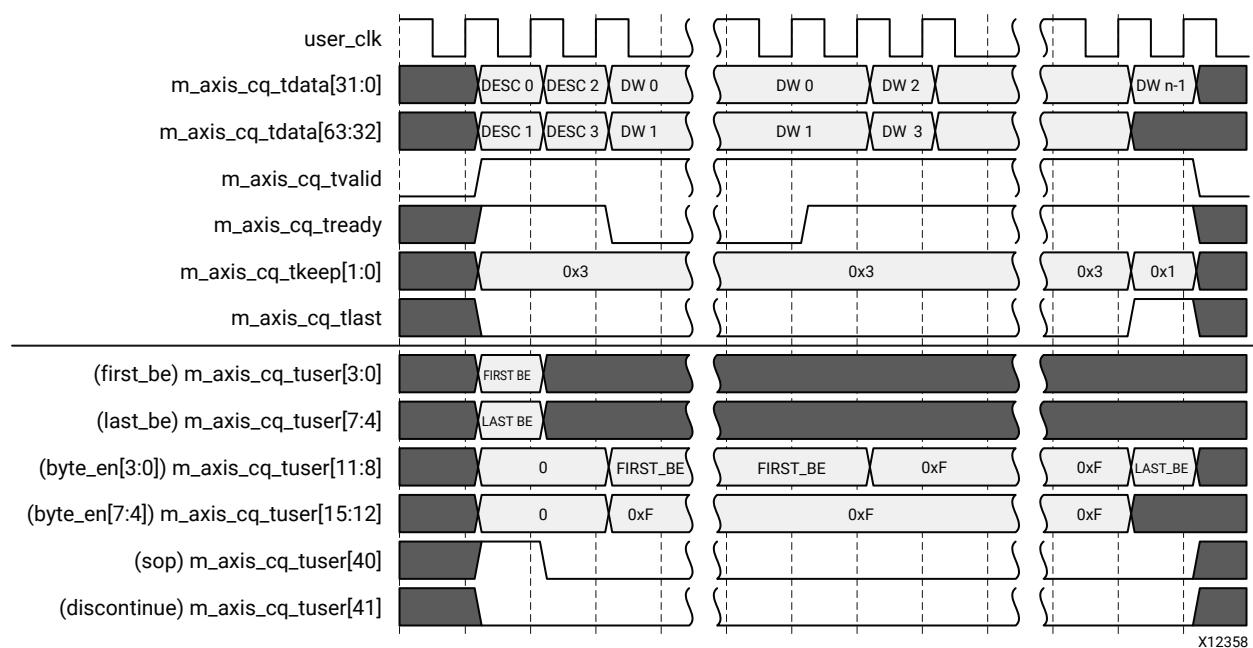
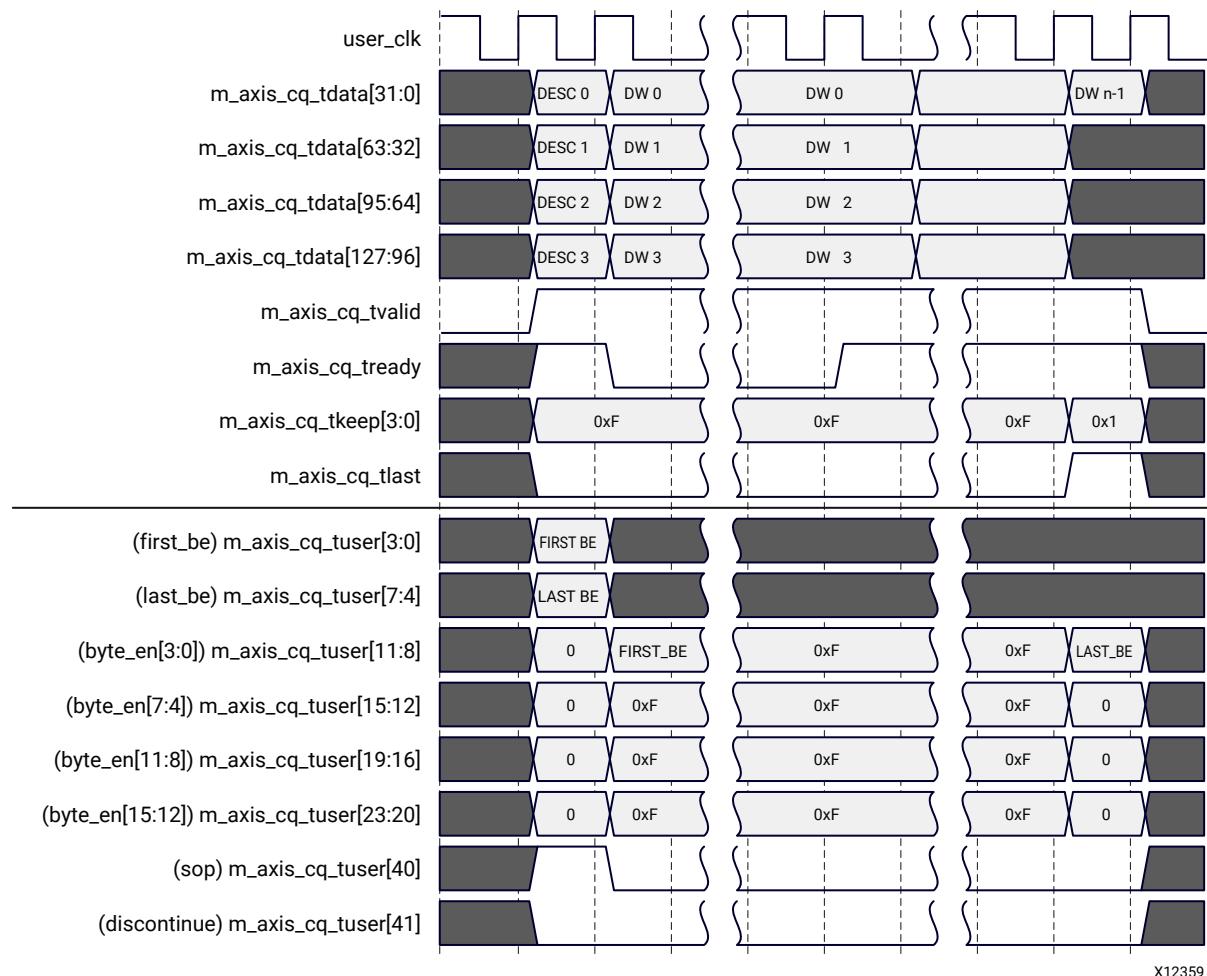
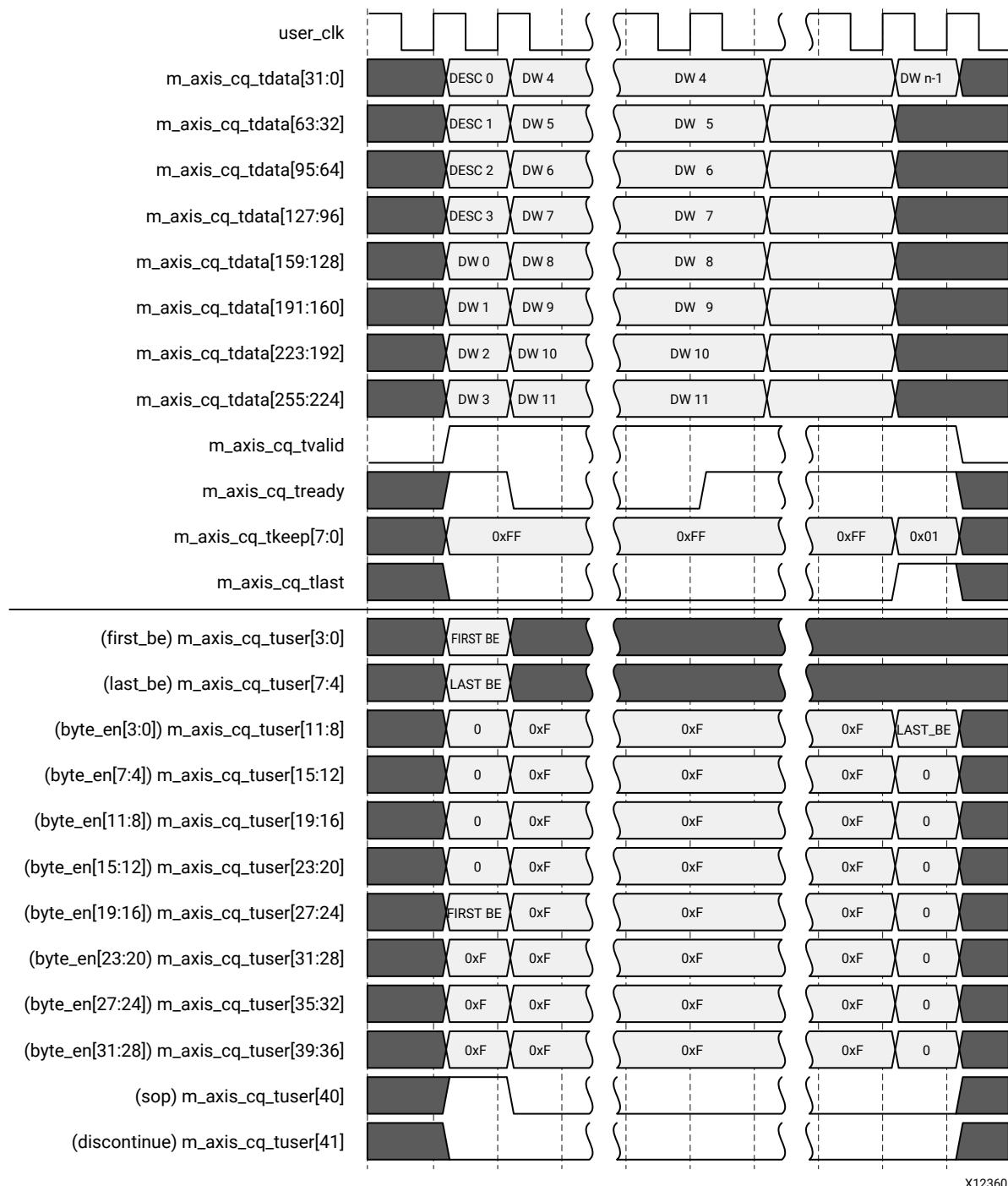


Figure 22: Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode, 128-Bit Interface)



X12359

Figure 23: Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode, 256-Bit Interface)



The following timing diagrams illustrate the address-aligned transfer of a memory write TLP received from the link across the CQ interface, when the interface width is configured as 64, 128 and 256 bits, respectively. For the purpose of illustration, the starting Dword address of the data block being written into memory is assumed to be $(m \times 32 + 1)$, for an integer $m > 0$. Its size is assumed to be n Dwords, for some $n = k \times 32 + 29$, $k > 0$.

In the address-aligned mode, the delivery of the payload always starts in the beat following the last byte of the descriptor. The first byte of the payload can appear on any byte lane, based on the address of the first valid byte of the payload. The keep outputs `m_axis_cq_tkeep` remain active-High in the gap between the descriptor and the payload. The actual position of the first valid byte in the payload can be determined either from the least significant bits of the address in the descriptor or from the byte enable bits `byte_en[31:0]` in the `m_axis_cq_tuser` bus.

For writes of two Dwords or less, the 1s on `byte_en` cannot be contiguous from the start of the payload. In the case of a zero-length memory write, the integrated block transfers a one-Dword payload with the `byte_en` bits all set to 0 for the payload bytes.

Figure 24: Memory Write Transaction on the Completer Request Interface (Address-Aligned Mode, 64-Bit Interface)

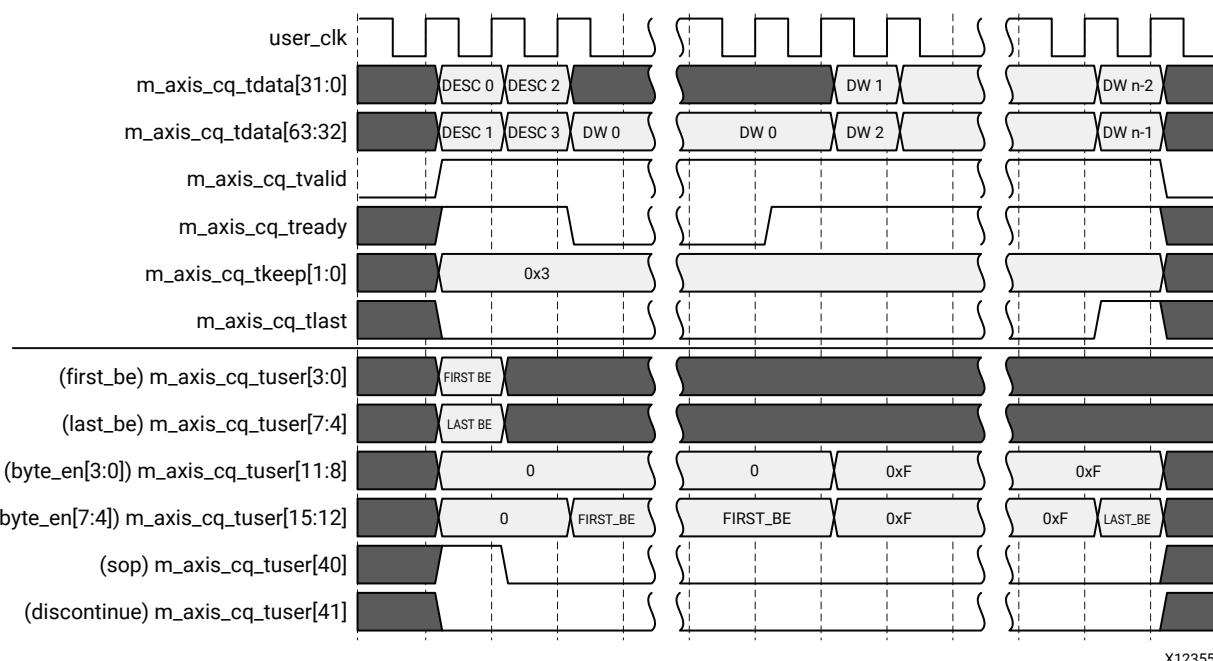


Figure 25: Memory Write Transaction on the Completer Request Interface (Address-Aligned Mode, 128-Bit Interface)

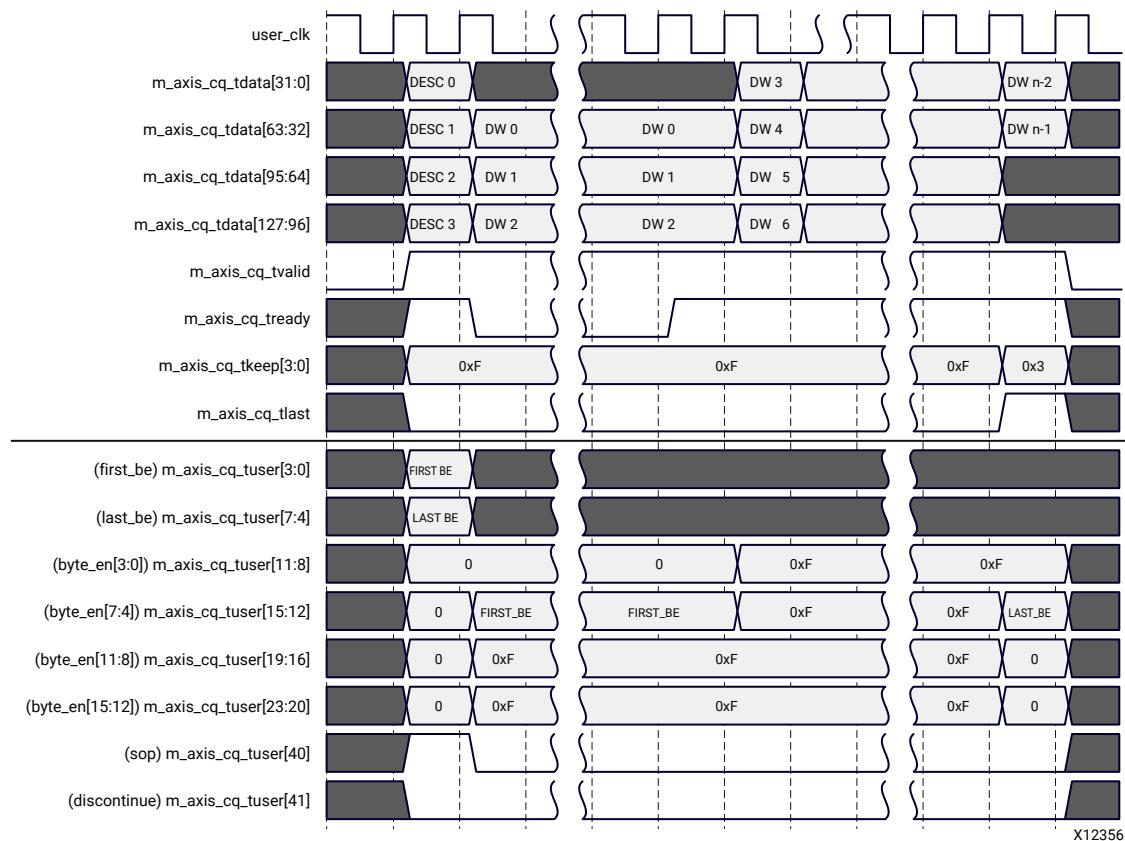
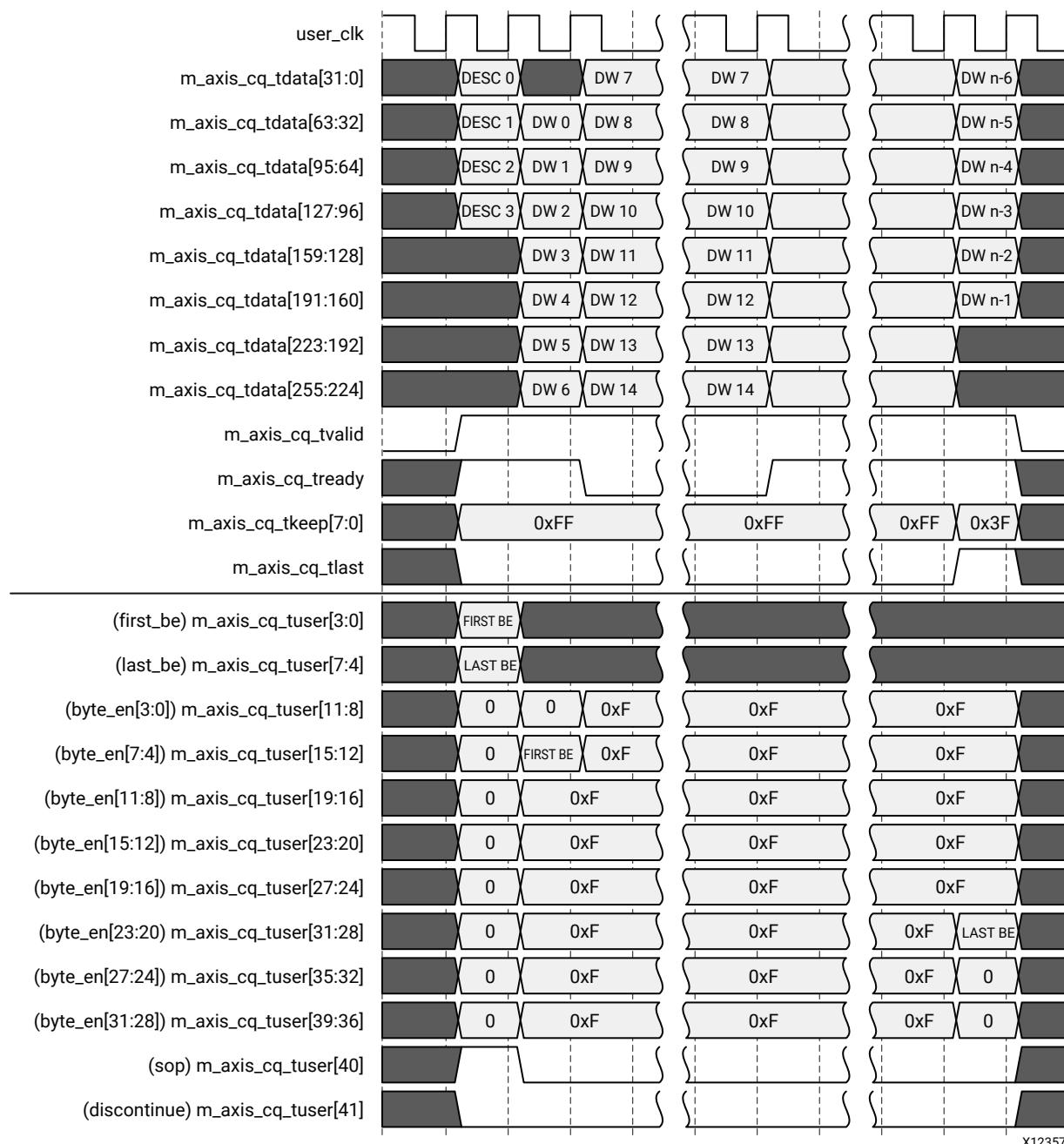


Figure 26: Memory Write Transaction on the Completer Request Interface (Address-Aligned Mode, 256-Bit Interface)



Completer Memory Read Operation

A memory read request is transferred across the completer request interface in the same manner as a memory write request, except that the AXI4-Stream packet contains only the 16-byte descriptor. The following timing diagrams illustrate the transfer of a memory read TLP received from the link across the completer request interface, when the interface width is configured as 64, 128, and 256 bits, respectively. The packet occupies two consecutive beats on the 64-bit interface, while it is transferred in a single beat on the 128- and 256-bit interfaces. The `m_axis_cq_tvalid` signal remains asserted over the duration of the packet. You can prolong a beat at any time by deasserting `m_axis_cq_tready`. The `sop` signal in the `m_axis_cq_tuser` bus is asserted when the first descriptor byte is on the bus.

Figure 27: Memory Read Transaction on the Completer Request Interface (64-Bit Interface)

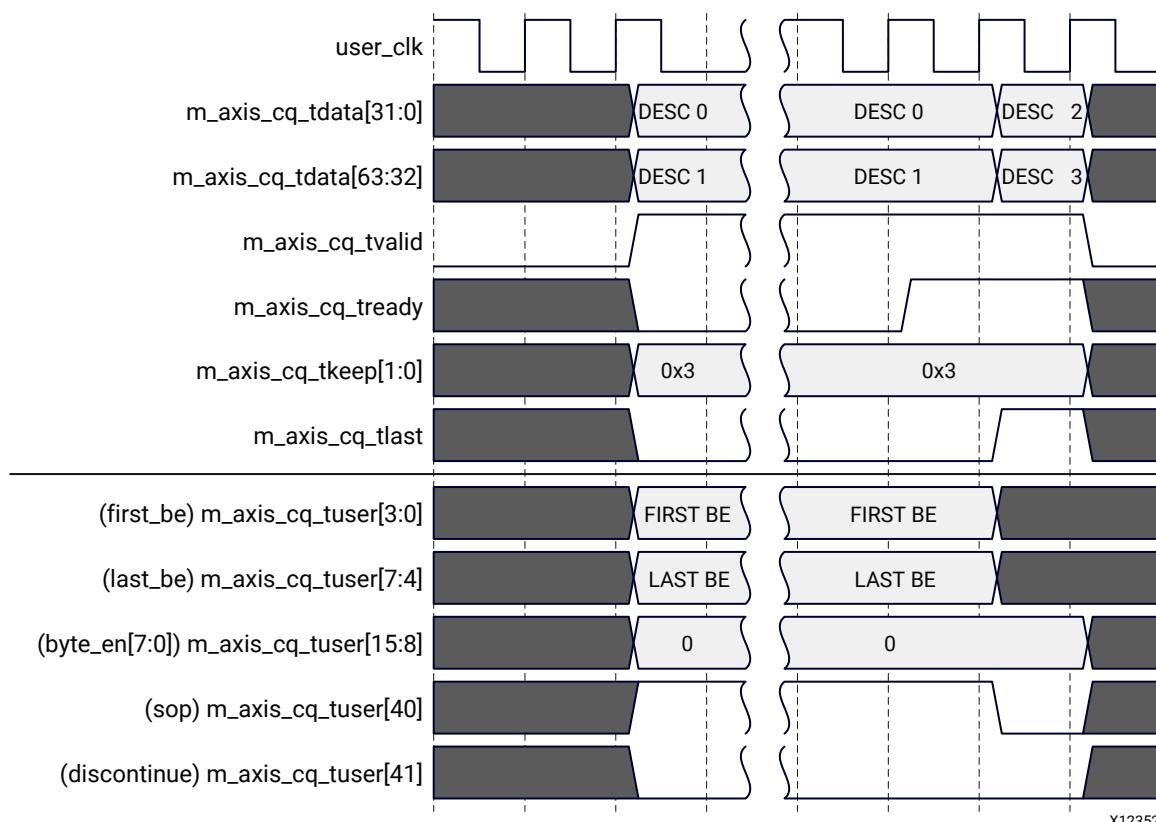


Figure 28: Memory Read Transaction on the Completer Request Interface (128-Bit Interface)

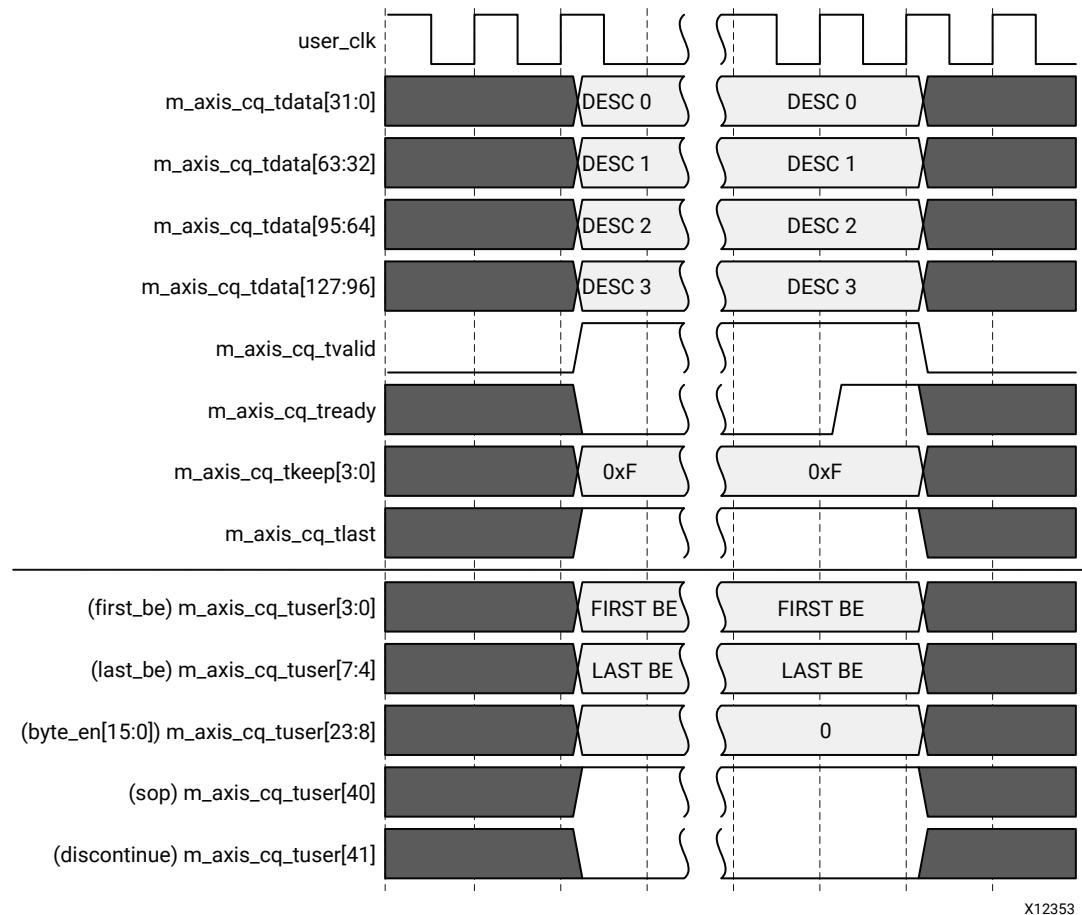
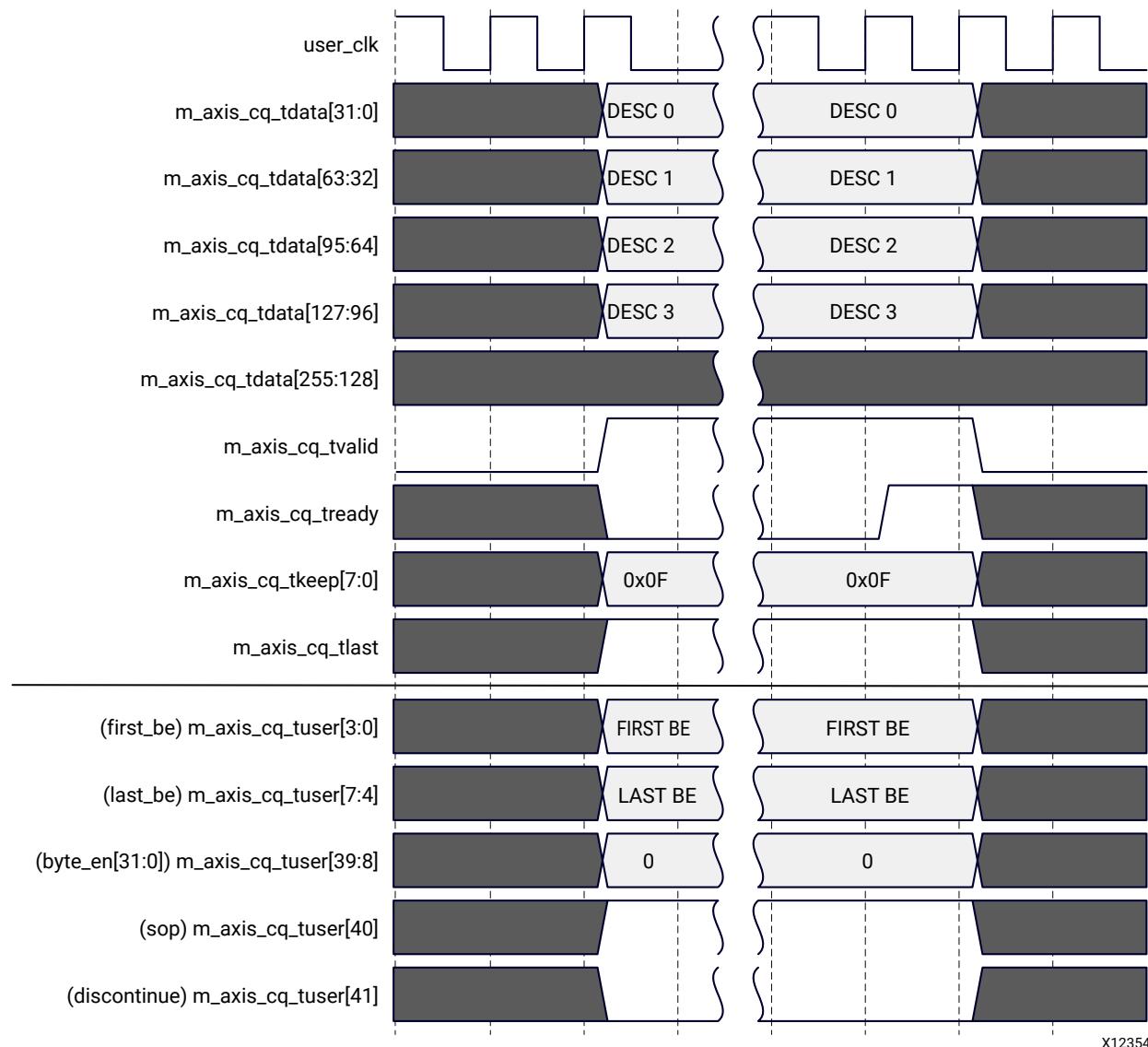


Figure 29: Memory Read Transaction on the Completer Request Interface (256-Bit Interface)



The byte enable bits associated with the read request for the first and last Dwords are supplied by the integrated block on the `m_axis_cq_tuser` sideband bus. These bits are valid when the first descriptor byte is being transferred, and must be used to determine the byte-level starting address and the byte count associated with the request. For the special cases of one-Dword and two-Dword reads, the byte enables can be non-contiguous. The byte enables are contiguous in all other cases. A zero-length memory read is sent on the CQ interface with the Dword count field in the descriptor set to 1 and the first and last byte enables set to 0.

The user application must respond to each memory read request with a Completion. The data requested by the read can be sent as a single Completion or multiple Split Completions. These Completions must be sent through the Completer Completion (CC) interface of the integrated block. The Completions for two distinct requests can be sent in any order, but the Split Completions for the same request must be in order. The operation of the CC interface is described in [Completer Completion Interface Operation](#).

I/O Write Operation

The transfer of an I/O write request on the CQ interface is similar to that of a memory write request with a one-Dword payload. The transfer starts with the 128-bit descriptor, followed by the one-Dword payload. When the Dword-aligned mode is in use, the payload Dword immediately follows the descriptor. When the address-alignment mode is in use, the payload Dword is supplied in a new beat after the descriptor, and its alignment in the datapath is based on the address in the descriptor. The First Byte Enable bits in the `m_axis_cq_tuser` indicate the valid bytes in the payload. The byte enable bits `byte_en` also provide this information.

Because an I/O write is a Non-Posted transaction, the user logic must respond to it with a Completion containing no data payload. The Completions for I/O requests can be sent in any order. Errors associated with the I/O write transaction can be signaled to the requester by setting the Completion Status field in the completion descriptor to CA (Completer Abort) or UR (Unsupported Request), as is appropriate. The operation of the Completer Completion interface is described in [Completer Completion Interface Operation](#).

I/O Read Operation

The transfer of an I/O read request on the CQ interface is similar to that of a memory read request, and involves only the descriptor. The length of the requested data is always one Dword, and the First Byte Enable bits in `m_axis_cq_tuser` indicate the valid bytes to be read.

The user logic must respond to an I/O read request with a one-Dword Completion (or a Completion with no data in the case of an error). The Completions for two distinct I/O read requests can be sent in any order. Errors associated with an I/O read transaction can be signaled to the requester by setting the Completion Status field in the completion descriptor to CA (Completer Abort) or UR (Unsupported Request), as is appropriate. The operation of the Completer Completion interface is described in [Completer Completion Interface Operation](#).

Atomic Operations on the Completer Request Interface

The transfer of an Atomic Op request on the completer request interface is similar to that of a memory write request. The payload for an Atomic Op can range from one Dword to eight Dwors, and its starting address is always aligned on a Dword boundary. The transfer starts with the 128-bit descriptor, followed by the payload. When the Dword-aligned mode is in use, the first payload Dword immediately follows the descriptor. When the address-alignment mode is in

use, the payload starts in a new beat after the descriptor, and its alignment is based on the address in the descriptor. The `m_axis_cq_tkeep` output indicates the end of the payload. The `byte_en` signals in `m_axis_cq_tuser` also indicate the valid bytes in the payload. The First Byte Enable and Last Byte Enable bits in `m_axis_cq_tuser` should not be used for Atomic Operations.

Because an Atomic Operation is a Non-Posted transaction, the user logic must respond to it with a Completion containing the result of the operation. Errors associated with the operation can be signaled to the requester by setting the Completion Status field in the completion descriptor to Completer Abort (CA) or Unsupported Request (UR), as is appropriate. The operation of the Completer Completion interface is described in [Completer Completion Interface Operation](#).

Message Requests on the Completer Request Interface

The transfer of a message on the CQ interface is similar to that of a memory write request, except that a payload might not always be present. The transfer starts with the 128-bit descriptor, followed by the payload, if present. When the Dword-aligned mode is in use, the payload immediately follows the descriptor. When the address-alignment mode is in use, the first Dword of the payload is supplied in a new beat after the descriptor, and always starts in byte lane 0. You can determine the end of the payload from the states of the `m_axis_cq_tlast` and `m_axis_cq_tkeep` signals. The `byte_en` signals in `m_axis_cq_tuser` also indicate the valid bytes in the payload. The First Byte Enable and Last Byte Enable bits in `m_axis_cq_tuser` should not be used for Message transactions.

Aborting a Transfer

For any request that includes an associated payload, the integrated block can signal an error in the transferred payload by asserting the `discontinue` signal in the `m_axis_cq_tuser` bus in the last beat of the packet (along with `m_axis_cq_tlast`). This occurs when the integrated block has detected an uncorrectable error while reading data from its internal memories. The user application must discard the entire packet when it has detected `discontinue` asserted in the last beat of a packet. This condition is considered a fatal error in the integrated block.

Selective Flow Control for Non-Posted Requests

The PCI Express® Base Specification requires that the Completer Request interface continue to deliver Posted transactions even when the user application is unable to accept Non-Posted transactions. To enable this capability, the integrated block implements a credit-based flow control mechanism on the CQ interface through which user logic can control the flow of Non-Posted requests without affecting Posted requests. The user logic signals the availability of buffers for receive Non-Posted requests using the `pcie_cq_np_req[0]` signal. The core delivers a Non-Posted request only when the available credit is non-zero. The integrated block continues to deliver Posted requests while the delivery of Non-Posted requests has been paused for lack of credit. When no back pressure is applied by the credit mechanism for the delivery of Non-Posted requests, the integrated block delivers Posted and Non-Posted requests in the same order as received from the link.

The integrated block maintains an internal credit counter to track the credit available for Non-Posted requests on the completer request interface. The following algorithm is used to keep track of the available credit:

- On reset, the counter is set to 0.
- After the integrated block comes out of reset, in every clock cycle:
 - If `pcie_cq_np_req[0]` is active-High and no Non-Posted request is being delivered this cycle, the credit count is incremented by 1, unless it has already reached its saturation limit of 32.
 - If `pcie_cq_np_req[0]` is Low and a Non-Posted request is being delivered this cycle, the credit count is decremented by 1, unless it is already 0.
 - Otherwise, the credit count remains unchanged.
- The integrated block starts delivery of a Non-Posted TLP only if the credit count is greater than 0.

The user application can either provide a one-cycle pulse on `pcie_cq_np_req[0]` each time it is ready to receive a Non-Posted request, or keep it permanently asserted if it does not need to exercise selective back pressure of Non-Posted requests. If the credit count is always non-zero, the integrated block delivers Posted and Non-Posted requests in the same order as received from the link. If it remains 0 for some time, Non-Posted requests can accumulate in the integrated block FIFO. When the credit count becomes non-zero later, the integrated block first delivers the accumulated Non-Posted requests that arrived before Posted requests already delivered, and then reverts to delivering the requests in the order received from the link.

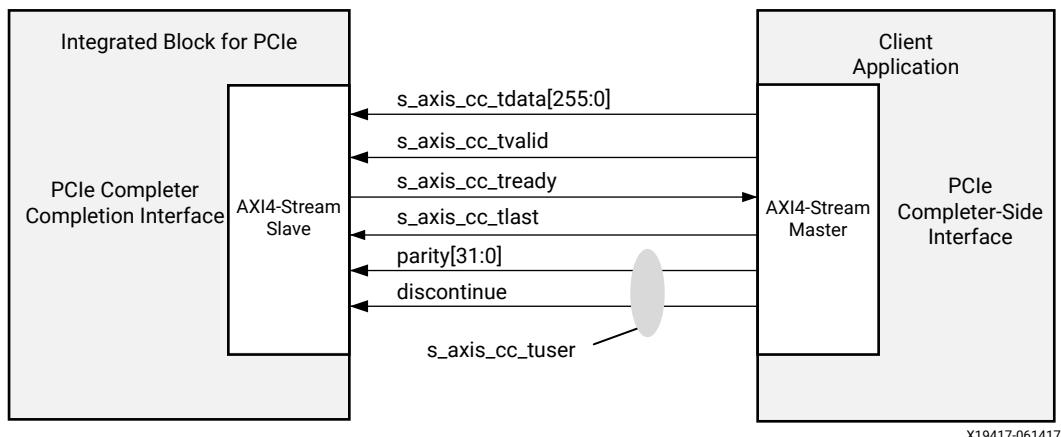
The assertion and deassertion of the `pcie_cq_np_req[0]` signal does not need to be aligned with the packet transfers on the completer request interface.

You can monitor the current value of the credit count on the output `pcie_cq_np_req_count[5:0]`. The counter saturates at 32. Because of internal pipeline delays, there can be several cycles of delay between the integrated block receiving a pulse on the `pcie_cq_np_req[0]` input and updating the `pcie_cq_np_req_count[5:0]` output in response. Thus, when the user application has adequate buffer space available, it should provide the credit in advance so that Non-Posted requests are not held up by the core for lack of credit.

Completer Completion Interface Operation

The following figure illustrates the signals associated with the completer completion interface of the core. The core delivers each TLP on this interface as an AXI4-Stream packet.

Figure 30: Completer Completion Interface Signals



The CC interface supports two distinct data alignment modes. In the Dword-aligned mode, the first byte of valid data must be presented in lane $n = (12 + A \bmod 4) \bmod w$, where A is the byte-level starting address of the data block being transferred (as conveyed in the Lower Address field of the descriptor) and w the width of the interface in bytes (8, 16, or 32). In the address-aligned mode, the data always starts in a new beat after the descriptor has ended. When transferring the Completion payload for a memory or I/O read request, its first valid byte is on lane $n = A \bmod w$. For all other Completions, the payload is aligned with byte lane 0.

Completer Completion Descriptor Format

The user application sends completion data for a completer request to the CC interface of the integrated block as an independent AXI4-Stream packet. Each packet starts with a descriptor and can have payload data following the descriptor. The descriptor is always 12 bytes long, and is sent in the first 12 bytes of the completion packet. The descriptor is transferred during the first two beats on a 64-bit interface, and in the first beat on a 128- or 256-bit interface. When the user application splits the completion data for a request into multiple Split Completions, it must send each Split Completion as a separate AXI4-Stream packet, with its own descriptor.

The format of the completer completion descriptor is illustrated in the following figure. The individual fields of the completer request descriptor are described in the following table.

Figure 31: Completer Completion Descriptor Format

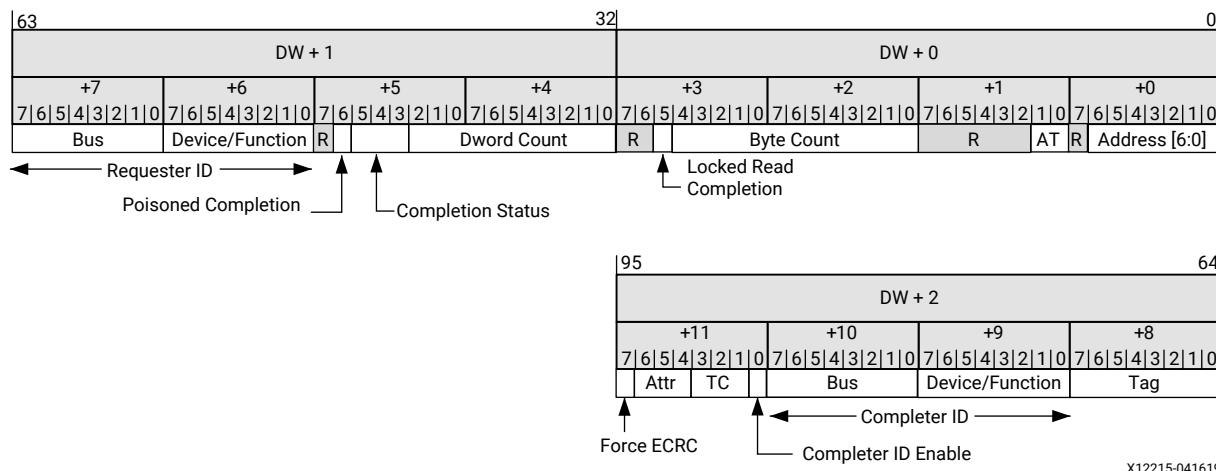


Table 51: Completer Completion Descriptor Fields

Bit Index	Field Name	Description	
6:0	Lower Address	For memory read Completions, this field must be set to the least significant 7 bits of the starting byte-level address of the memory block being transferred. For the first (or only) Completion, the Completer can generate this field from the least significant 5 bits of the address of the Request concatenated with 2 bits of byte-level address formed by the byte enables for the first Dword of the Request as shown below.	
		first_be[3:0]	Lower Address[1:0]
		4'b0000	2'b00
		4'bxxx1	2'b00
		4'bxx10	2'b01
		4'bxx100	2'b10
		4'b1000	2'b11
9:8	Address Type	For any subsequent Completions, the Lower Address field is always zero except for Completions generated by a Root Complex with a Read Completion Boundary (RCB) value of 64 bytes. In this case the least significant 6 bits of the Lower Address field is always zero and the most significant bit of the Lower Address field toggles according to the alignment of the 64-byte data payload.	
		For all other Completions, the Lower Address must be set to all zeros.	
9:8	Address Type	This field is defined for Completions of memory transactions and Atomic Operations only. For these Completions, the user logic must copy the AT bits from the corresponding request descriptor into this field. This field must be set to 0 for all other Completions.	

Table 51: Completer Completion Descriptor Fields (cont'd)

Bit Index	Field Name	Description
28:16	Byte Count	<p>These 13 bits can have values in the range of 0 – 4,096 bytes. If a Memory Read Request is completed using a single Completion, the Byte Count value indicates Payload size in bytes. This field must be set to 4 for I/O read Completions and I/O write Completions. The byte count must be set to 1 while sending a Completion for a zero-length memory read, and a dummy payload of 1 Dword must follow the descriptor.</p> <p>For each Memory Read Completion, the Byte Count field must indicate the remaining number of bytes required to complete the Request, including the number of bytes returned with the Completion.</p> <p>If a Memory Read Request is completed using multiple Completions, the Byte Count value for each successive Completion is the value indicated by the preceding Completion minus the number of bytes returned with the preceding Completion. The total number of bytes required to complete a Memory Read Request is calculated as shown in the following table. MSB of the Byte Count field is reserved.</p>
29	Locked Read Completion	This bit must be set when the Completion is in response to a Locked Read request. It must be set to 0 for all other Completions.
42:32	Dword Count	These 11 bits indicate the size of the payload of the current packet in Dwords. Its range is 0 - 1K Dwords. This field must be set to 1 for I/O read Completions and 0 for I/O write Completions. The Dword count must be set to 1 while sending a Completion for a zero-length memory read. The Dword count must be set to 0 when sending a UR or CA Completion. In all other cases, the Dword count must correspond to the actual number of Dwords in the payload of the current packet.
45:43	Completion Status	<p>These bits must be set based on the type of Completion being sent. The only valid settings are:</p> <ul style="list-style-type: none"> • 000: Successful Completion • 001: Unsupported Request (UR) • 100: Completer Abort (CA)
46	Completion Status	This bit can be used to poison the Completion TLP being sent. This bit must be set to 0 for all Completions, except when the user application detects an error in the block of data following the descriptor and wants to communicate this information using the Data Poisoning feature of PCI Express.
63:48	Requester ID	PCI Requester ID associated with the request (copied from the request).
71:64	Tag	PCI Express Tag associated with the request (copied from the request).

Table 51: Completer Completion Descriptor Fields (cont'd)

Bit Index	Field Name	Description
79:72	Target Function/ Device Number	<p>Device and/or Function number of the Completer Function.</p> <p>Endpoint mode:</p> <p>ARI enabled:</p> <ul style="list-style-type: none"> Bits [79:72] must be set to the Completer Function number. <p>ARI disabled:</p> <ul style="list-style-type: none"> Bits [74:72] must be set to the Completer Function number. Bits [79:75] are not used <p>Upstream Port for Switch use case (Endpoint mode is selected within the IP):</p> <p>ARI enabled:</p> <ul style="list-style-type: none"> Bits [79:72] must be set to the Completer Function number. <p>ARI disabled:</p> <ul style="list-style-type: none"> Bits [74:72] must be set to the Completer Function number. Bits [79:75] are not used if the Completion is originating from the switch itself. These bits must be set to the Completer Device number where the Completion was originated if the switch is relaying the Completion (Completer is external to the switch). This is used in conjunction with Completer ID Enable bit in the descriptor. <p>Root Port mode (Downstream Port):</p> <p>ARI enabled:</p> <ul style="list-style-type: none"> Bits [79:72] must be set to the Completer Function number. <p>ARI disabled:</p> <ul style="list-style-type: none"> Bits [74:72] must be set to the Completer Function number. Bits [79:75] must be set to the Completer Device number. This is used in conjunction with Completer ID Enable bit in the descriptor.
87:80	Completer Bus Number	<p>Bus number associated with the Completer Function.</p> <p>Endpoint mode:</p> <ul style="list-style-type: none"> Not Used <p>Upstream Port for Switch use case (Endpoint mode is selected within the IP):</p> <ul style="list-style-type: none"> Not used if the Completion is originating from the switch itself. These bits must be set to the Completer Bus number where the Completion was originated if the switch is relaying the Completion (Completer is external to the switch). This is used in conjunction with Completer ID Enable bit in the descriptor. <p>Root Port mode (Downstream Port):</p> <ul style="list-style-type: none"> Must be set to the Completer Bus number. This is used in conjunction with Completer ID Enable bit in the descriptor.

Table 51: Completer Completion Descriptor Fields (cont'd)

Bit Index	Field Name	Description
88	Completer ID Enable	<p>Values are:</p> <ul style="list-style-type: none"> • 1'b1: The client supplies Bus, Device, and Function numbers in the descriptor to be populated as the Completer ID field in the TLP header. • 1'b0: IP uses Bus and Device numbers captured from received Configuration requests and the client supplies Function numbers in the descriptor to be populated as the Completer ID field in the TLP header. <p>Endpoint mode:</p> <ul style="list-style-type: none"> • Must be set to 1'b0. <p>Upstream Port for Switch use case (Endpoint mode is selected within the IP):</p> <ul style="list-style-type: none"> • Set to 1'b0 when the Completion is originating from the switch itself. • Set to 1'b1 when the switch is relaying the Completion (Completer is external to the switch). This is used in conjunction with Completer Bus Number bits [95:88] and Completer Function/Device Number bits [87:83] when ARI is not enabled. <p>Root Port mode:</p> <ul style="list-style-type: none"> • Must be set to 1'b1. This is used in conjunction with Completer Bus Number bits [95:88] and Completer Function/Device Number bits [87:83] when ARI is not enabled.
91:89	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request. The user application must copy this value from the TC field of the associated request descriptor.
94:92	Attributes	PCIe attributes associated with the request (copied from the request). Bit 92 is the No Snoop bit, bit 93 is the Relaxed Ordering bit, and bit 94 is the ID-Based Ordering bit.
95	Force ECRC	Force ECRC insertion. Setting this bit to 1 forces the integrated block to append a TLP Digest containing ECRC to the Completion TLP, even when ECRC is not enabled for the Function sending the Completion.

Table 52: Calculating Byte Count from Completer Request first_be[3:0], last_be[3:0], Dword Count[10:0]

first_be[3:0]	last_be[3:0]	Total Byte Count
1xx1	0000	4
01x1	0000	3
1x10	0000	3
0011	0000	2
0110	0000	2
1100	0000	2
0001	0000	1
0010	0000	1
0100	0000	1
1000	0000	1
0000	0000	1

Table 52: Calculating Byte Count from Completer Request `first_be[3:0]`, `last_be[3:0]`, Dword Count[10:0] (cont'd)

<code>first_be[3:0]</code>	<code>last_be[3:0]</code>	Total Byte Count
xxx1	1xxx	Dword_count × 4
xxx1	01xx	(Dword_count × 4)-1
xxx1	001x	(Dword_count × 4)-2
xxx1	0001	(Dword_count × 4)-3
xx10	1xxx	(Dword_count × 4)-1
xx10	01xx	(Dword_count × 4)-2
xx10	001x	(Dword_count × 4)-3
xx10	0001	(Dword_count × 4)-4
x100	1xxx	(Dword_count × 4)-2
x100	01xx	(Dword_count × 4)-3
x100	001x	(Dword_count × 4)-4
x100	0001	(Dword_count × 4)-5
1000	1xxx	(Dword_count × 4)-3
1000	01xx	(Dword_count × 4)-4
1000	001x	(Dword_count × 4)-5
1000	0001	(Dword_count × 4)-6

Completions with Successful Completion Status

The user application must return a Completion to the CC interface of the core for every Non-Posted request it receives from the completer request interface. When the request completes with no errors, the user application must return a Completion with Successful Completion (SC) status. Such a Completion might or might not contain a payload, depending on the type of request. Furthermore, the data associated with the request can be broken up into multiple Split Completions when the size of the data block exceeds the maximum payload size configured. The user logic is responsible for splitting the data block into multiple Split Completions when needed. The user application must transfer each Split Completion over the completer completion interface as a separate AXI4-Stream packet, with its own 12-byte descriptor.

In the example timing diagrams of this section, the starting Dword address of the data block being transferred (as conveyed in bits [6:2] of the Lower Address field of the descriptor) is assumed to be $(m \times 8 + 1)$, for an integer m . The size of the data block is assumed to be n Dwords, for some $n = k \times 32 + 28$, $k > 0$.

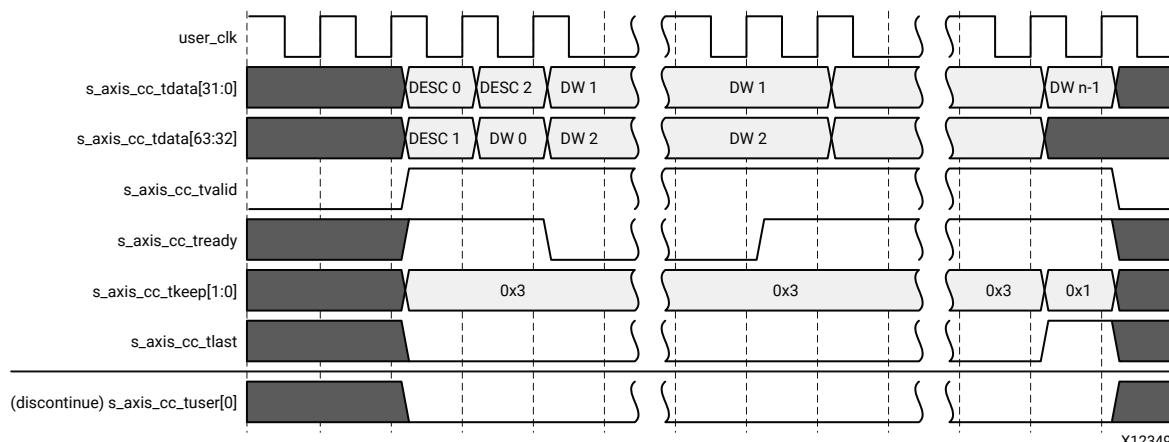
The CC interface supports two data alignment modes: Dword-aligned and address-aligned. The following timing diagrams illustrate the Dword-aligned transfer of a Completion from the user application across the CC interface, when the interface width is configured as 64, 128, and 256 bits, respectively. In this case, the first Dword of the payload starts immediately after the descriptor. When the data block is not a multiple of four bytes, or when the start of the payload is not aligned on a Dword address boundary, the user application must add null bytes to align the

start of the payload on a Dword boundary and make the payload a multiple of Dwords. For example, when the data block starts at byte address 7 and has a size of 3 bytes, the user application must add three null bytes before the first byte and two null bytes at the end of the block to make it two Dwors long. Also, in the case of non-contiguous reads, not all bytes in the data block returned are valid. In that case, the user application must return the valid bytes in the proper positions, with null bytes added in gaps between valid bytes, when needed. The interface does not have any signals to indicate the valid bytes in the payload. This is not required, as the requester is responsible for keeping track of the byte enables in the request and discarding invalid bytes from the Completion.

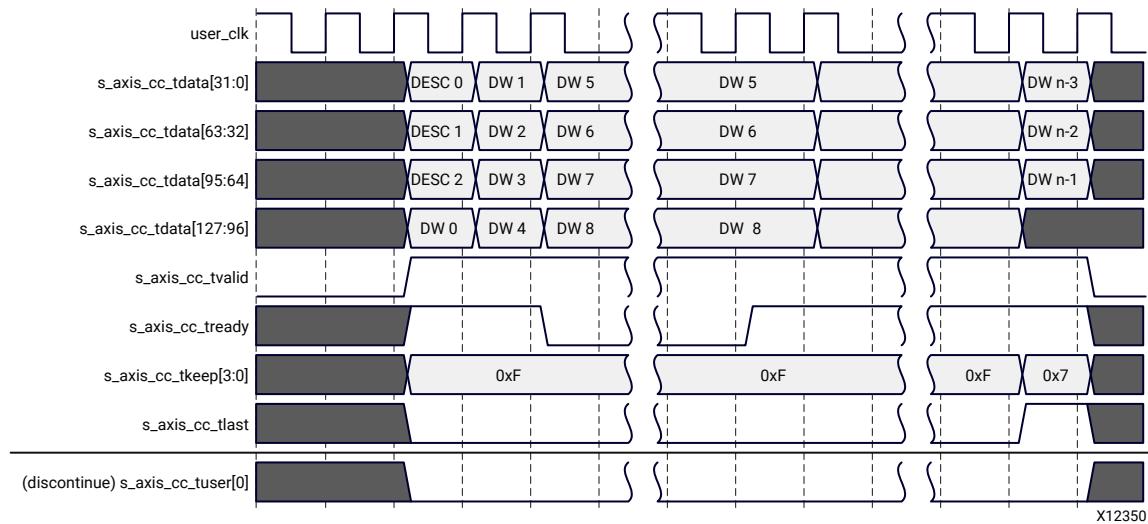
In the Dword-aligned mode, the transfer starts with the 12 descriptor bytes, followed immediately by the payload bytes. The user application must keep the `s_axis_cc_tvalid` signal asserted over the duration of the packet. The integrated block treats the deassertion of `s_axis_cc_tvalid` during the packet transfer as an error, and nullifies the corresponding Completion TLP transmitted on the link to avoid data corruption.

The user application must also assert the `s_axis_cc_tlast` signal in the last beat of the packet. The integrated block can deassert `s_axis_cc_tready` in any cycle if it is not ready to accept data. The user application must not change the values on the CC interface during a clock cycle that the integrated block has deasserted `s_axis_cc_tready`.

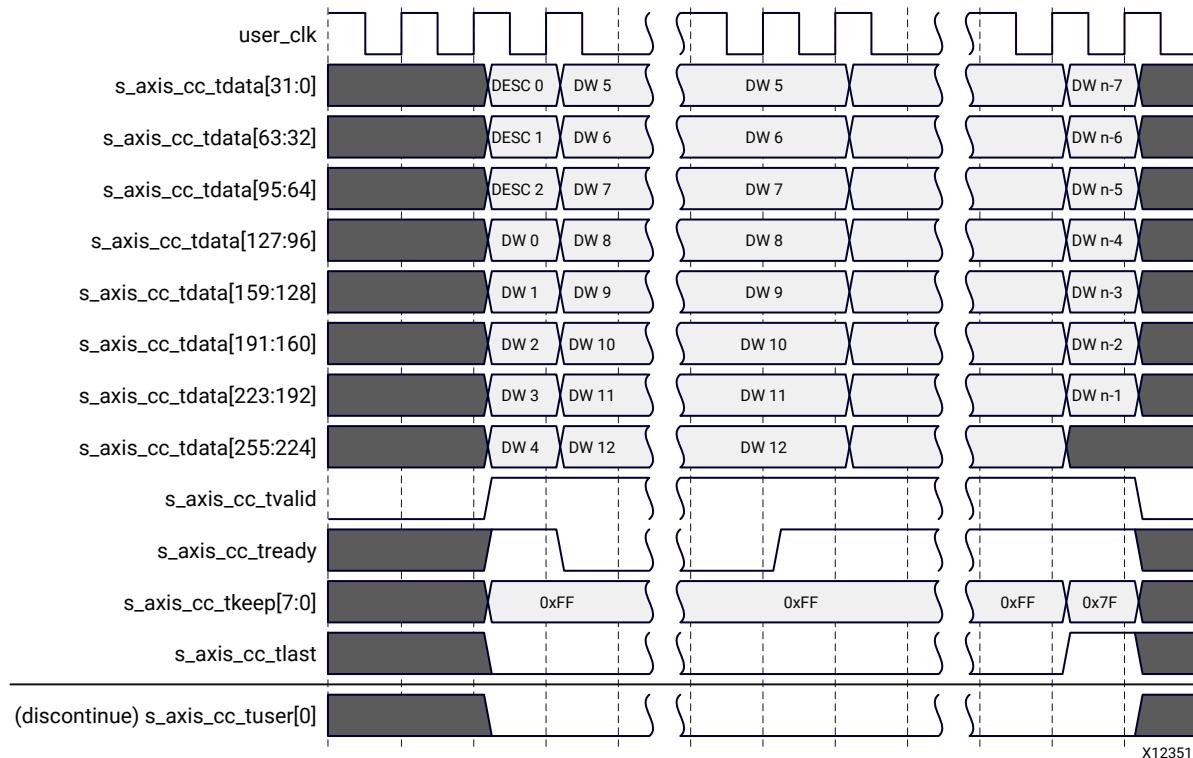
Figure 32: Transfer of a Normal Completion on the Completer Completion Interface (Dword-Aligned Mode, 64-Bit Interface)



**Figure 33: Transfer of a Normal Completion on the Completer Completion Interface
(Dword-Aligned Mode, 128-Bit Interface)**



**Figure 34: Transfer of a Normal Completion on the Completer Completion Interface
(Dword-Aligned Mode, 256-Bit Interface)**



In the address-aligned mode, the delivery of the payload always starts in the beat following the last byte of the descriptor. For memory read Completions, the first byte of the payload can appear on any byte lane, based on the address of the first valid byte of the payload. For all other Completions, the payload must start in byte lane 0.

The following timing diagrams illustrate the address-aligned transfer of a memory read Completion across the completer completion interface, when the interface width is configured as 64, 128, and 256 bits, respectively. For the purpose of illustration, the starting Dword address of the data block being transferred (as conveyed in bits [6:2] of the Lower Address field of the descriptor) is assumed to be $(m \times 8 + 1)$, for some integer m . The size of the data block is assumed to be n Dwords, for some $n = k \times 32 + 28$, $k > 0$.

Figure 35: Transfer of a Normal Completion on the Completer Completion Interface (Address-Aligned Mode, 64-Bit Interface)

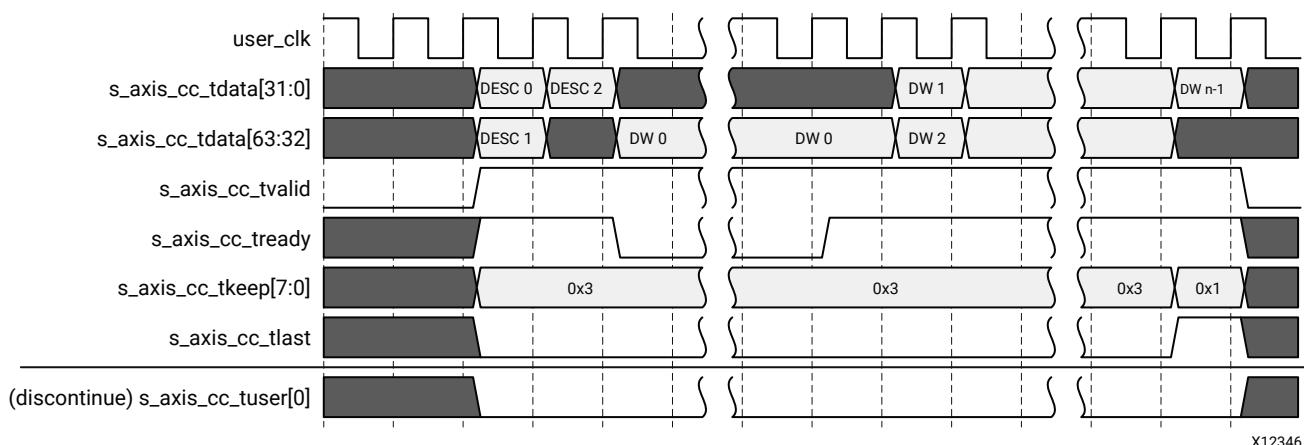
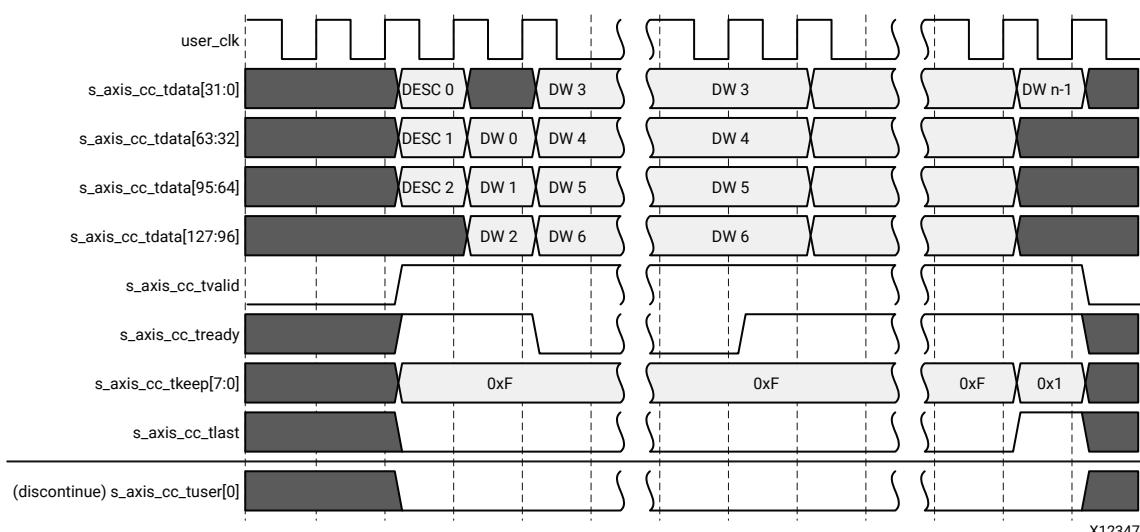
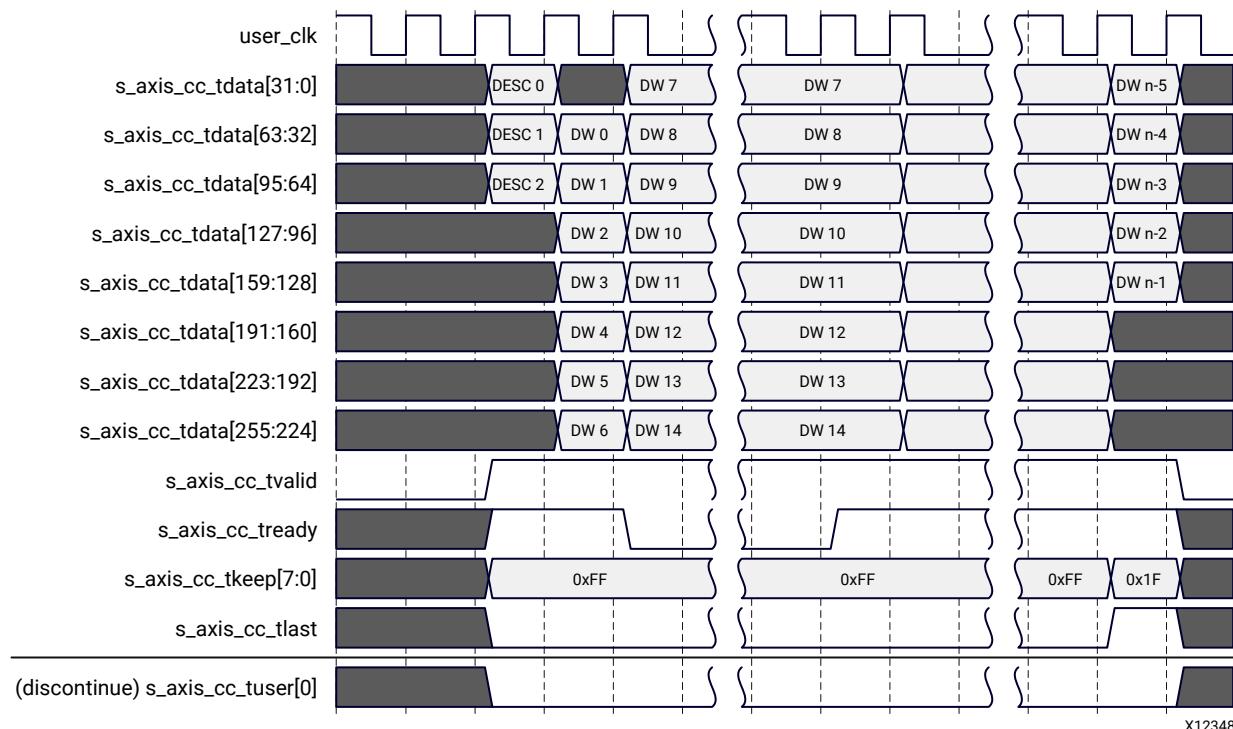


Figure 36: Transfer of a Normal Completion on the Completer Completion Interface (Address-Aligned Mode, 128-Bit Interface)



**Figure 37: Transfer of a Normal Completion on the Completer Completion Interface
(Address-Aligned Mode, 256-Bit Interface)**



Aborting a Completion Transfer

The user application can abort the transfer of a completion transaction on the completer completion interface at any time during the transfer of the payload by asserting the `discontinue` signal in the `s_axis_cc_tuser` bus. The integrated block nullifies the corresponding TLP on the link to avoid data corruption.

The user application can assert this signal in any cycle during the transfer, when the Completion being transferred has an associated payload. The user application can either choose to terminate the packet prematurely in the cycle where the error was signaled (by asserting `s_axis_cc_tlast`), or can continue until all bytes of the payload are delivered to the integrated block. In the latter case, the integrated block treats the error as sticky for the following beats of the packet, even if the user application deasserts the `discontinue` signal before reaching the end of the packet.

The `discontinue` signal can be asserted only when `s_axis_cc_tvalid` is active-High. The integrated block samples this signal when `s_axis_cc_tvalid` and `s_axis_cc_tready` are both asserted. Thus, after assertion, the `discontinue` signal should not be deasserted until `s_axis_cc_tready` is asserted.

When the integrated block is configured as an Endpoint, this error is reported by the integrated block to the Root Complex to which it is attached, as an Uncorrectable Internal Error using the Advanced Error Reporting (AER) mechanisms.

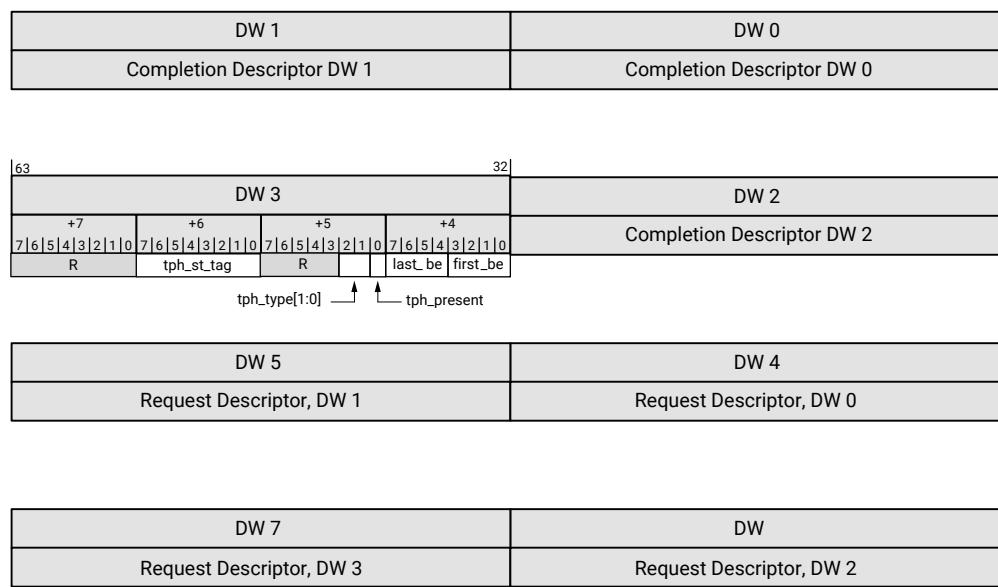
Completions with Error Status (UR and CA)

When responding to a request received on the completer request interface with an Unsupported Request (UR) or Completion Abort (CA) status, the user application must send a three-Dword completion descriptor in the format of the *Completer Completion Descriptor Format* figure in [Completer Completion Descriptor Format](#), followed by five additional Dwords containing information on the request that generated the Completion. These five Dwords are necessary for the integrated block to log information about the request in its AER header log registers.

The following figure shows the sequence of information transferred when sending a Completion with UR or CA status. The information is formatted as an AXI4-Stream packet with a total of 8 Dwords, which are organized as follows:

- The first three Dwords contain the completion descriptor in the format of the *Completer Completion Descriptor Format* figure in [Completer Completion Descriptor Format](#).
- The fourth Dword contains the state of the following signals in `m_axis_cq_tuser`, copied from the request:
 - The First Byte Enable bits `first_be[3:0]` in `m_axis_cq_tuser`.
 - The Last Byte Enable bits `last_be[3:0]` in `m_axis_cq_tuser`.
 - Signals carrying information on Transaction Processing Hint: `tph_present`, `tph_type[1:0]`, and `tph_st_tag[7:0]` in `m_axis_cq_tuser`.

Figure 38: Composition of the AXI4-Stream Packet for UR and CA Completions



X12245

The entire packet takes four beats on the 64-bit interface, two beats on the 128-bit interface, and a single beat on the 256-bit interface. The packet is transferred in an identical manner in both the Dword-aligned mode and the address-aligned mode, with the Dwords packed together. The user application must keep the `s_axis_cc_tvalid` signal asserted over the duration of the packet. It must also assert the `s_axis_cc_tlast` signal in the last beat of the packet. The integrated block can deassert `s_axis_cc_tready` in any cycle if it is not ready to accept. The user application must not change the values on the CC interface in any cycle that the integrated block has deasserted `s_axis_cc_tready`.

64/128/256-Bit Requester Interface

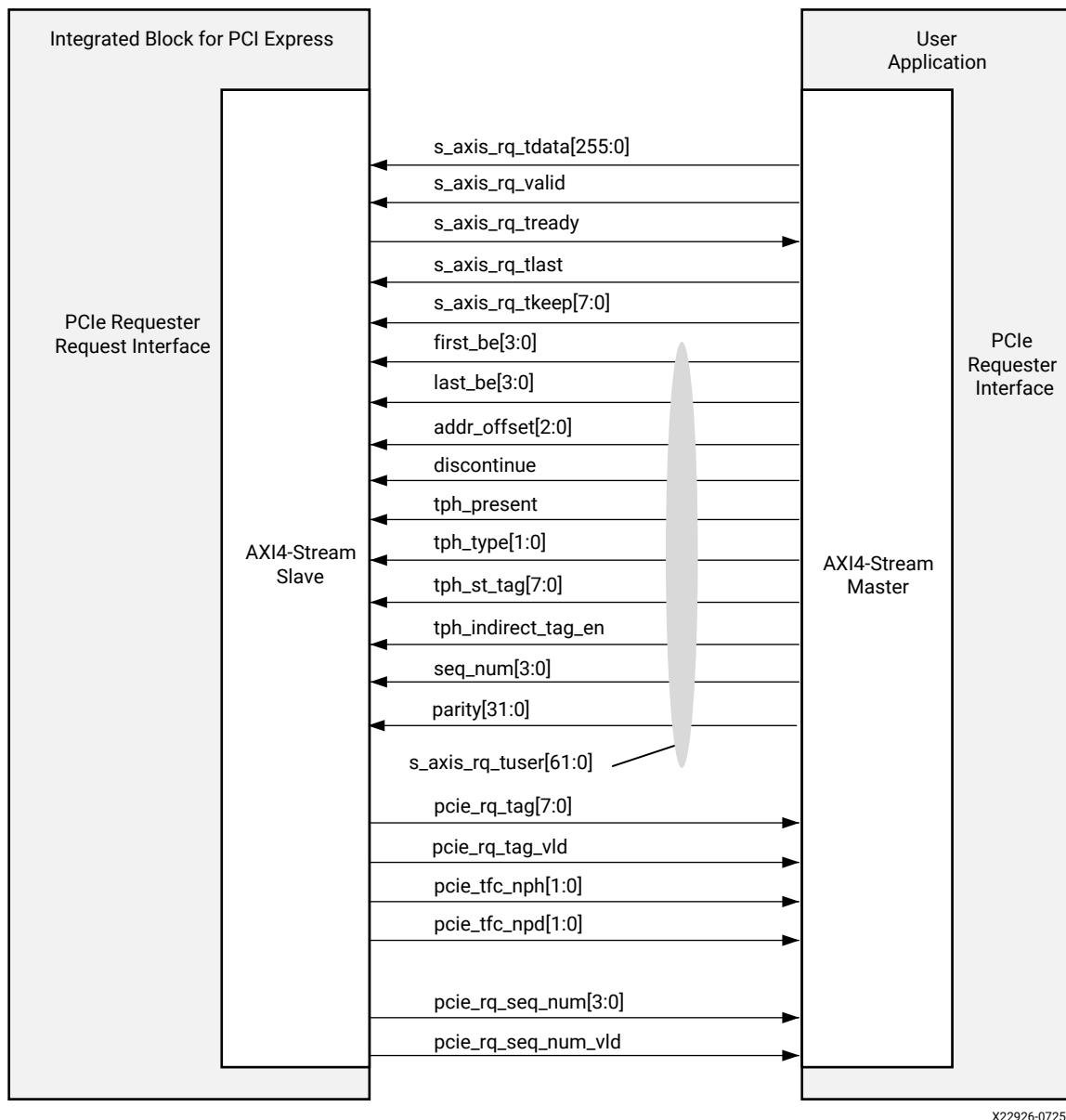
The requester interface enables a user Endpoint application to initiate PCI transactions as a bus master across the PCIe link to the host memory. For Root Complexes, this interface is also used to initiate I/O and configuration requests. This interface can also be used by both Endpoints and Root Complexes to send messages on the PCIe® link. The transactions on this interface are similar to those on the completer interface, except that the roles of the core and the user application are reversed. Posted transactions are performed as single indivisible operations and Non-Posted transactions as split transactions.

The requester interface consists of two separate interfaces, one for data transfer in each direction. Each interface is based on the AXI4-Stream protocol, and its width can be configured as 64, 128, or 256 bits. The Requester reQuest (RQ) interface is for transfer of requests (with any associated payload data) from the user application to the integrated block, and the Requester Completion (RC) interface is used by the integrated block to deliver Completions received from the link (for Non-Posted requests) to the user application. The two interfaces operate independently. That is, the user application can transfer new requests over the RQ interface while receiving a completion for a previous request.

Requester Request Interface Operation

On the RQ interface, the user application delivers each TLP as an AXI4-Stream packet. The packet starts with a 128-bit descriptor, followed by data in the case of TLPs with a payload. The following figure shows the signals associated with the requester request interface.

Figure 39: Requester Request Interface



The RQ interface supports two distinct data alignment modes for transferring payloads. In the Dword-aligned mode, the user logic must provide the first Dword of the payload immediately after the last Dword of the descriptor. It must also set the bits in `first_be[3:0]` to indicate the valid bytes in the first Dword and the bits in `last_be[3:0]` (both part of the bus `s_axis_rq_tuser`) to indicate the valid bytes in the last Dword of the payload. In the address-aligned mode, the user application must start the payload transfer in the beat following the last Dword of the descriptor, and its first Dword can be in any of the possible Dword positions on the

datapath. The user application communicates the offset of the first Dword on the datapath using the `addr_offset[2:0]` signals in `s_axis_rq_tuser`. As in the case of the Dword-aligned mode, the user application must also set the bits in `first_be[3:0]` to indicate the valid bytes in the first Dword and the bits in `last_be[3:0]` to indicate the valid bytes in the last Dword of the payload.

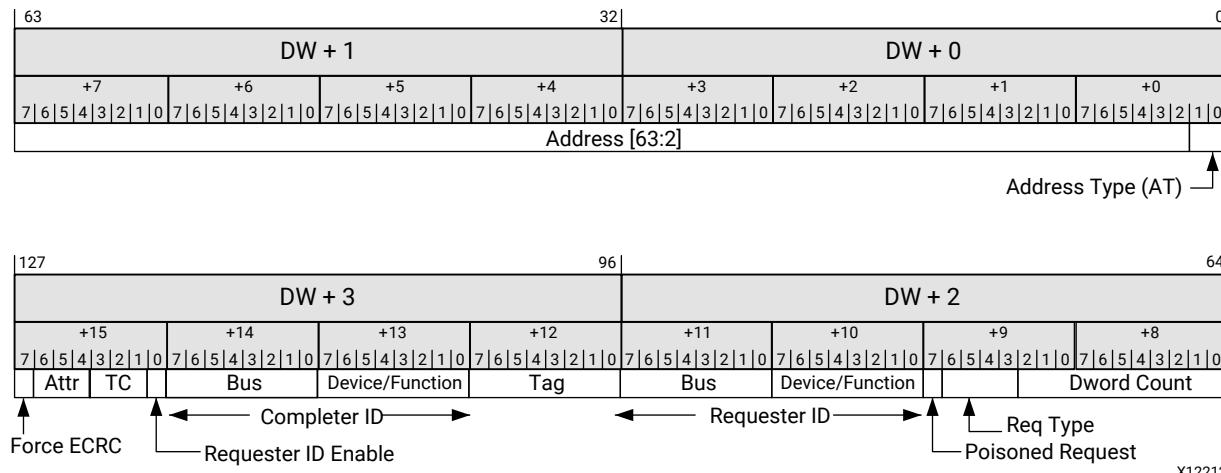
When the Transaction Processing Hint Capability is enabled in the integrated block, the user application can provide an optional Hint with any memory transaction using the `tph_*` signals included in the `s_axis_rq_tuser` bus. To supply a Hint with a request, the user logic must assert `tph_present` in the first beat of the packet, and provide the TPH Steering Tag and Steering Tag Type on `tph_st_tag[7:0]` and `tph_st_type[1:0]`, respectively. Instead of supplying the value of the Steering Tag to be used, the user application also has the option of providing an indirect Steering Tag. This is done by setting the `tph_indirect_tag_en` signal to 1 when `tph_present` is asserted, and placing an index on `tph_st_tag[7:0]`, instead of the tag value. The integrated block then reads the tag stored in its Steering Tag Table associated with the requester Function at the offset specified in the index and inserts it in the request TLP.

Requester Request Descriptor Formats

The user application must transfer each request to be transmitted on the link to the RQ interface of the integrated block as an independent AXI4-Stream packet. Each packet must start with a descriptor and can have payload data following the descriptor. The descriptor is always 16 bytes long, and must be sent in the first 16 bytes of the request packet. The descriptor is transferred during the first two beats on a 64-bit interface, and in the first beat on a 128-bit or 256-bit interface. The formats of the descriptor for different request types are illustrated in the following figures.

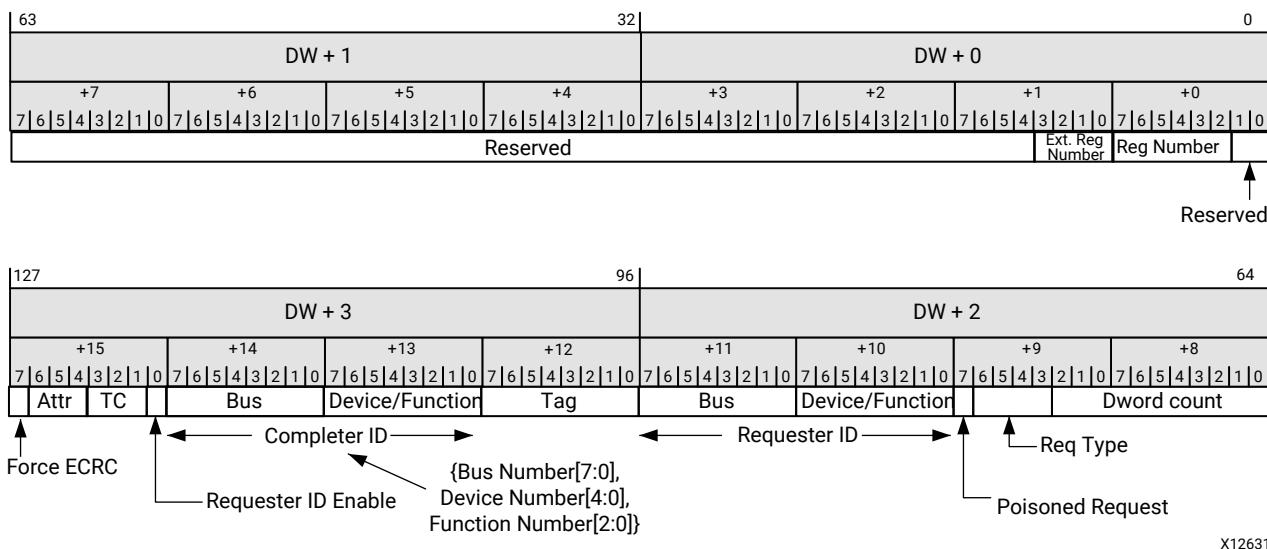
The format of the following figure applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request.

Figure 40: Requester Request Descriptor Format for Memory, I/O, and Atomic Operation Requests



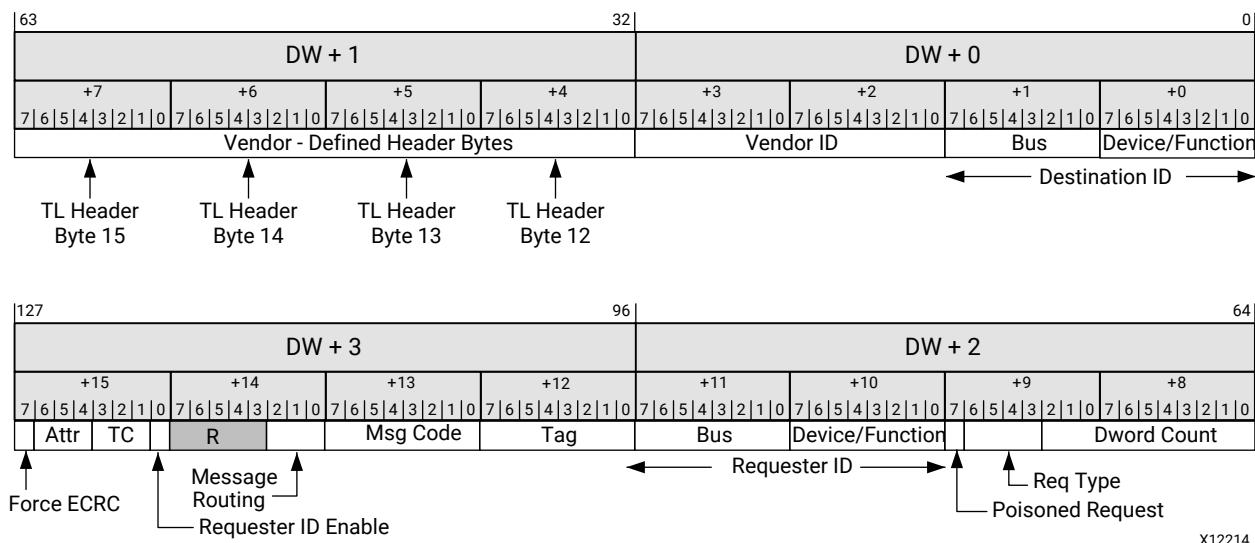
The format in the following figure is used for Configuration Requests.

Figure 41: Requester Request Descriptor Format for Configuration Requests



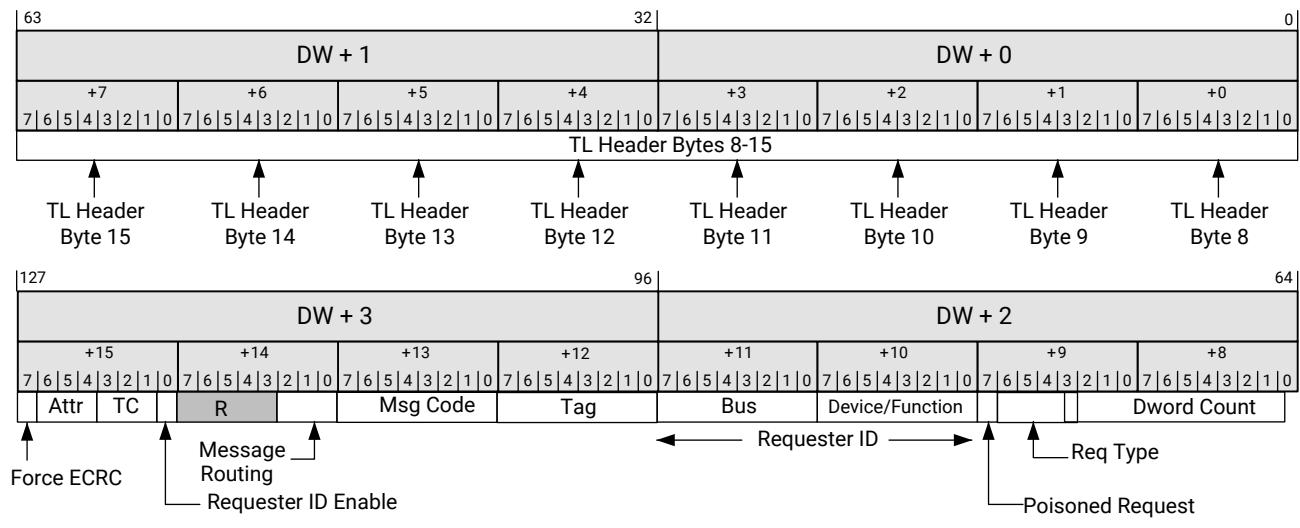
The format in the following figure is used for Vendor-Defined Messages (Type 0 or Type 1) only.

Figure 42: Requester Request Descriptor Format for Vendor-Defined Messages



The format in the following figure is used for all ATS messages (Invalid Request, Invalid Completion, Page Request, PRG Response).

Figure 43: Requester Request Descriptor Format for ATS Messages



For all other messages, the descriptor takes the format shown in the following figure.

Figure 44: Requester Request Descriptor Format for all other Messages

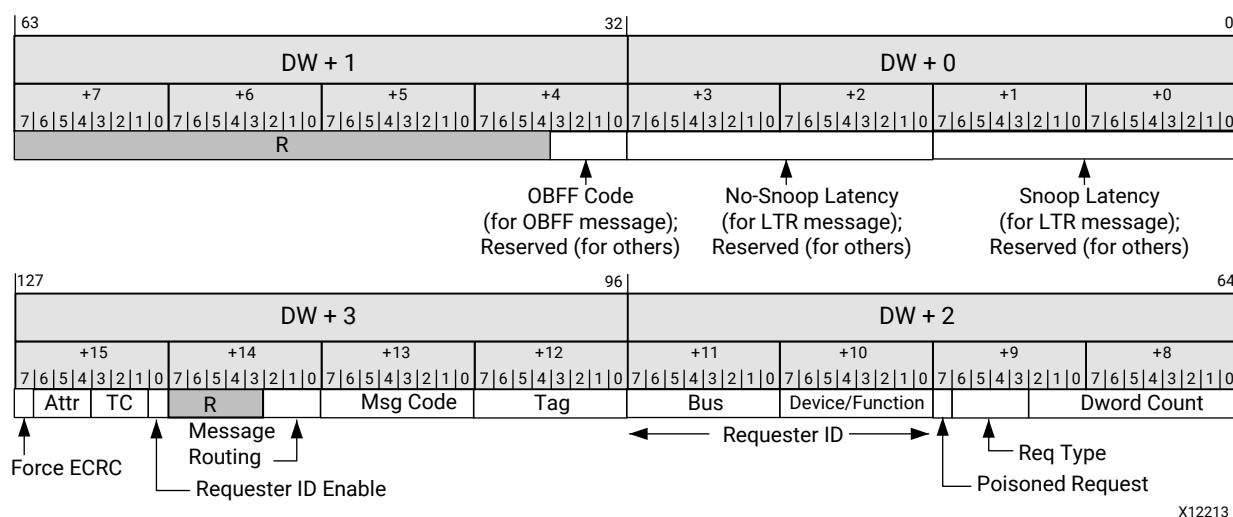


Table 53: Requester Request Descriptor Fields

Bit Index	Field Name	Description
1:0	Address Type	<p>This field is defined for memory transactions and Atomic Operations only. The integrated block copies this field into the AT of the TL header of the request TLP.</p> <ul style="list-style-type: none"> • 00: Address in the request is untranslated • 01: Transaction is a Translation Request • 10: Address in the request is a translated address • 11: Reserved

Table 53: Requester Request Descriptor Fields (cont'd)

Bit Index	Field Name	Description
63:2	Address	<p>This field applies to memory, I/O, and Atomic Op requests. This is the address of the first Dword referenced by the request. The user application must also set the First_BE and Last_BE bits in s_axis_rq_tuser to indicate the valid bytes in the first and last Dwords, respectively.</p> <p>When the transaction specifies a 32-bit address, bits [63:32] of this field must be set to 0.</p>
74:64	Dword Count	<p>These 11 bits indicate the size of the block (in Dwords) to be read or written (for messages, size of the message payload). The valid range for Memory Write Requests is 0-256 Dwords. Memory Read Requests have a valid range of 1-1024 Dwords. For I/O accesses, the Dword count is always 1.</p> <p>For a zero length memory read/write request, the Dword count must be 1, with the First_BE bits set to all zeros.</p> <p>The integrated block does not check the setting of this field against the actual length of the payload supplied (for requests with payload), nor against the maximum payload size or read request size settings of the integrated block.</p>
78:75	Request Type	<p>Identifies the transaction type. The transactions types and their encodings are listed in Table 50: Transaction Types.</p>
79	Poisoned Request	<p>This bit can be used to poison the request TLP being sent. This feature is supported on all request types except Type 0 and Type 1 Configuration Write Requests. This bit must be set to 0 for all requests, except when the user application detects an error in the block of data following the descriptor and wants to communicate this information using the Data Poisoning feature of PCI Express.</p> <p>This feature is supported on all request types except Type 0 and Type 1 Configuration Write Requests.</p>
87:80	Requester Function/Device Number	<p>Device and/or Function number of the Requester Function.</p> <p>Endpoint mode:</p> <ul style="list-style-type: none"> • ARI enabled: <ul style="list-style-type: none"> ◦ Bits [87:80] must be set to the Requester Function number. • ARI disabled: <ul style="list-style-type: none"> ◦ Bits [82:80] must be set to the Requester Function number. ◦ Bits [87:83] are not used <p>Upstream Port for Switch use case (Endpoint mode is selected within the IP):</p> <ul style="list-style-type: none"> • ARI enabled: <ul style="list-style-type: none"> ◦ Bits [87:80] must be set to the Requester Function number. • ARI disabled: <ul style="list-style-type: none"> ◦ Bits [82:80] must be set to the Requester Function number. ◦ Bits [87:83] are not used if the request is originating from the switch itself. These bits must be set to the Requester Device number where the request was originated if the switch is relaying the request (Requester is external to the switch). This is used in conjunction with Requester ID Enable bit in the descriptor. <p>Root Port mode (Downstream Port):</p> <ul style="list-style-type: none"> • ARI enabled: <ul style="list-style-type: none"> ◦ Bits [87:80] must be set to the Requester Function number. • ARI disabled: <ul style="list-style-type: none"> ◦ Bits [87:80] must be set to the Requester Function number. ◦ Bits [87:83] must be set to the Requester Device number. This is used in conjunction with Requester ID Enable bit in the descriptor.

Table 53: Requester Request Descriptor Fields (cont'd)

Bit Index	Field Name	Description
95:88	Requester Bus Number	<p>Bus number associated with the Requester Function.</p> <p>Endpoint mode:</p> <ul style="list-style-type: none"> Not Used <p>Upstream Port for Switch use case (Endpoint mode is selected within the IP):</p> <ul style="list-style-type: none"> Not used if the request is originating from the switch itself. These bits must be set to the Requester Bus number where the request was originated if the switch is relaying the request (Requester is external to the switch). This is used in conjunction with Requester ID Enable bit in the descriptor. <p>Root Port mode (Downstream Port):</p> <ul style="list-style-type: none"> Must be set to the Requester Bus number. This is used in conjunction with Requester ID Enable bit in the descriptor.
103:96	Tag	<p>PCIe Tag associated with the request.</p> <p>For Non-Posted transactions, the integrated block uses the value from this field if the AXISTEN_IF_ENABLE_CLIENT_TAG parameter is set (that is, when tag management is performed by the user application). Bits [101:96] are used as the tag. Bits [103:102] are reserved. If this parameter is not set, tag management logic in the integrated block generates the tag to be used, and the value in the tag field of the descriptor is not used.</p>
119:104	Completer ID	<p>This field is applicable only to Configuration requests and messages routed by ID. For these requests, this field specifies the PCI Completer ID associated with the request (these 16 bits are divided into an 8-bit bus number, 5-bit device number, and 3-bit function number in the legacy interpretation mode. In the ARI mode, these 16 bits are treated as an 8-bit bus number + 8-bit Function number).</p>
120	Requester ID Enable	<p>1'b1: The client supplies Bus, Device, and Function numbers in the descriptor to be populated as the Requester ID field in the TLP header.</p> <p>1'b0: IP uses Bus and Device numbers captured from received Configuration requests and the client supplies Function numbers in the descriptor to be populated as the Requester ID field in the TLP header. When Requester ID enable is 0 the device number fields in descriptor should also be 0.</p> <p>Endpoint mode:</p> <ul style="list-style-type: none"> Must be set to 1'b0. <p>Upstream Port for Switch use case (Endpoint mode is selected within the IP):</p> <ul style="list-style-type: none"> Set to 1'b0 when the request is originating from the switch itself. Set to 1'b1 when the switch is relaying the request (Requester is external to the switch). This is used in conjunction with Requester Bus Number bits [95:88] and Requester Function/Device Number bits [87:83] when ARI is not enabled. <p>Root Port mode:</p> <ul style="list-style-type: none"> Must be set to 1'b1. This is used in conjunction with Requester Bus Number bits [95:88] and Requester Function/Device Number bits [87:83] when ARI is not enabled.
123:121	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request.

Table 53: Requester Request Descriptor Fields (cont'd)

Bit Index	Field Name	Description
126:124	Attributes	<p>These bits provide the setting of the Attribute bits associated with the request. Bit 124 is the No Snoop bit and bit 125 is the Relaxed Ordering bit. Bit 126 is the ID-Based Ordering bit, and can be set only for memory requests and messages.</p> <p>The integrated block forces the attribute bits to 0 in the request sent on the link if the corresponding attribute is not enabled in the Function's PCI Express Device Control register.</p>
127	Force ECRC	Force ECRC insertion. Setting this bit to 1 forces the integrated block to append a TLP Digest containing ECRC to the Request TLP, even when ECRC is not enabled for the Function sending request.
15:0	Snoop Latency	This field is defined for LTR messages only. It provides the value of the 16-bit Snoop Latency field in the TLP header of the message.
31:16	No-Snoop Latency	This field is defined for LTR messages only. It provides the value of the 16-bit No-Snoop Latency field in the TLP header of the message.
35:32	OBFF Code	<p>The OBFF Code field is used to distinguish between various OBFF cases:</p> <ul style="list-style-type: none"> • 1111b: CPU Active – System fully active for all device actions including bus mastering and interrupts. • 0001b: OBFF – System memory path available for device memory read/write bus master activities. • 0000b: Idle – System in an idle, low power state. • All other codes are reserved.
111:104	Message Code	<p>This field is defined for all messages. It contains the 8-bit Message Code to be set in the TL header.</p> <p>Appendix F of the PCI Express® Base Specification, rev. 3.0 (<i>PCI-SIG Specifications</i> (http://www.pcisig.com/specifications)) provides a complete list of the supported Message Codes.</p>
114:112	Message Routing	This field is defined for all messages. The integrated block copies these bits into the 3-bit Routing field r[2:0] of the TLP header of the Request TLP.
15:0	Destination ID	This field applies to Vendor-Defined Messages only. When the message is routed by ID (that is, when the Message Routing field is 010 binary), this field must be set to the Destination ID of the message.
63:32	Vendor-Defined Header	This field applies to Vendor-Defined Messages only. It is copied into Dword 3 of the TLP header.
63:0	ATS Header	This field is applicable to ATS messages only. It contains the bytes that the integrated block copies into Dwords 2 and 3 of the TLP header.

Requester Memory Write Operation

In both Dword-aligned, the transfer starts with the sixteen descriptor bytes, followed immediately by the payload bytes. The user application must keep the `s_axis_rq_tvalid` signal asserted over the duration of the packet. The integrated block treats the deassertion of `s_axis_rq_tvalid` during the packet transfer as an error, and nullifies the corresponding Request TLP transmitted on the link to avoid data corruption.

The user application must also assert the `s_axis_rq_tlast` signal in the last beat of the packet. The integrated block can deassert `s_axis_rq_tready` in any cycle if it is not ready to accept data. The user application must not change the values on the RQ interface during cycles when the integrated block has deasserted `s_axis_rq_tready`. The AXI4-Stream interface signals `m_axis_rq_tkeep` (one per Dword position) must be set to indicate the valid Dwors in the packet including the descriptor and any null bytes inserted between the descriptor and the payload. That is, the `tkeep` bits must be set to 1 contiguously from the first Dword of the descriptor until the last Dword of the payload. During the transfer of a packet, the `tkeep` bits can be 0 only in the last beat of the packet, when the packet does not fill the entire width of the interface.

The requester request interface also includes the First Byte Enable and the Last Enable bits in the `s_axis_rq_tuser` bus. These must be set in the first beat of the packet, and provide information of the valid bytes in the first and last Dwors of the payload.

The user application must limit the size of the payload transferred in a single request to the maximum payload size configured in the integrated block, and must ensure that the payload does not cross a 4 Kbyte boundary. For memory writes of two Dwors or less, the 1s in `first_be` and `last_be` can be non-contiguous. For the special case of a zero-length memory write request, the user application must provide a dummy one-Dword payload with `first_be` and `last_be` both set to all 0s. For one DW transfers, `last_be[3:0]` should be 0 and `first_be[3:0]` indicates the valid bytes. In all other cases, the 1 bits in `first_be` and `last_be` must be contiguous.

The following timing diagrams illustrate the Dword-aligned transfer of a memory write request from the user application across the requester request interface, when the interface width is configured as 64, 128, and 256 bits, respectively. For illustration purposes, the size of the data block being written into user application memory is assumed to be n Dwors, for some $n = k \times 32 + 29$, $k > 0$.

Figure 45: Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, 64-Bit Interface)

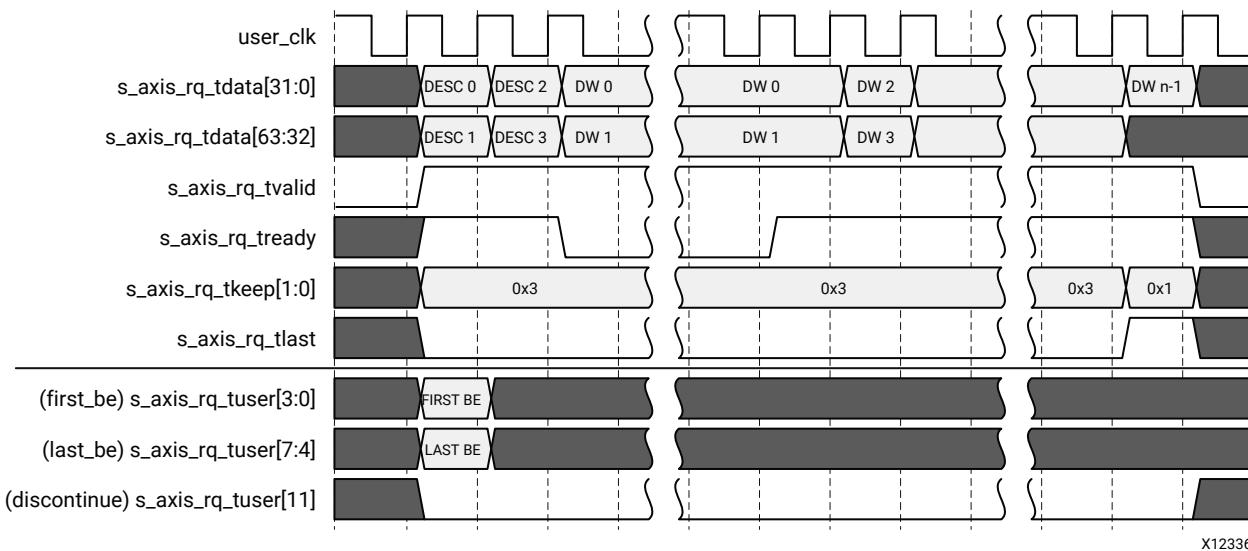


Figure 46: Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, 128-Bit Interface)

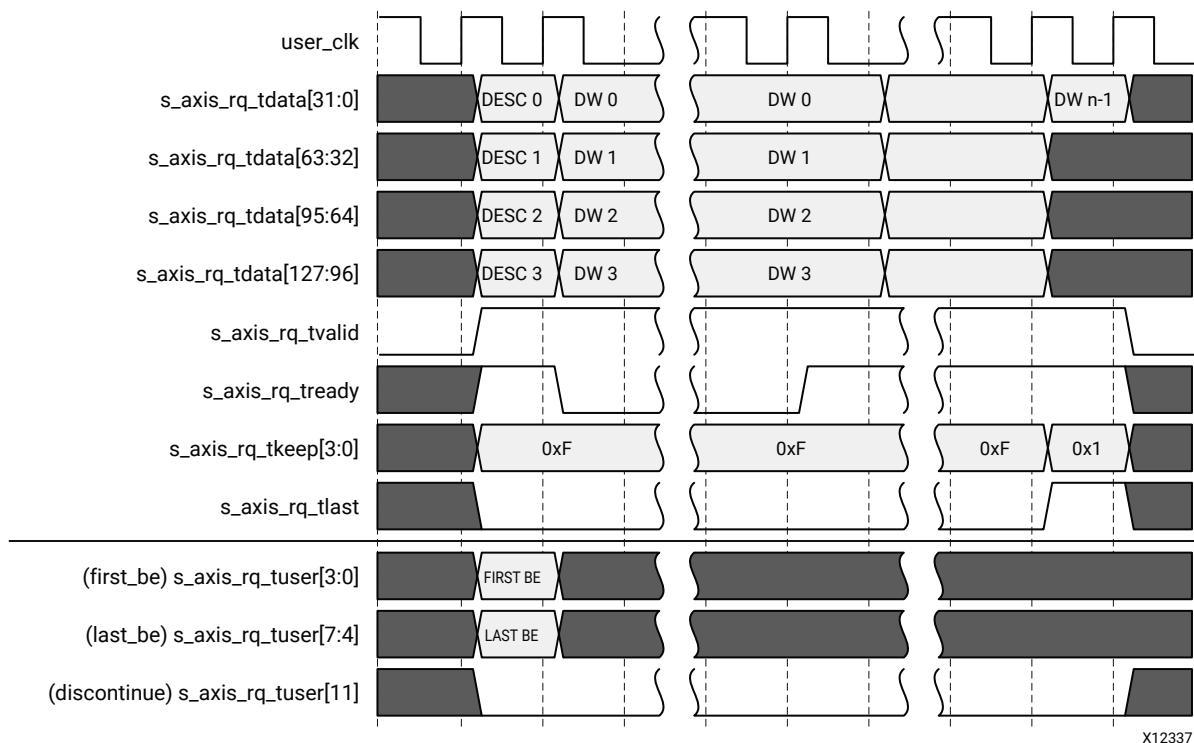
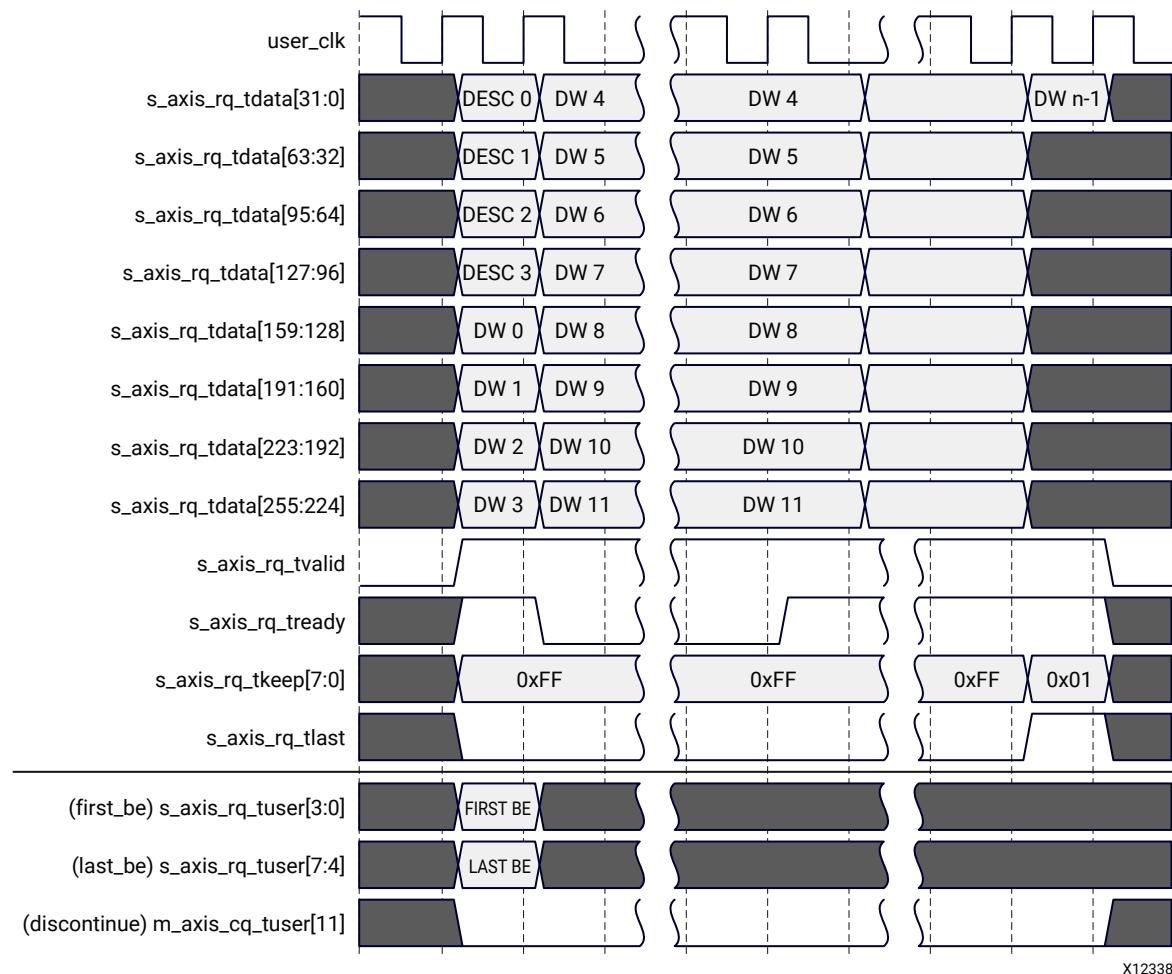


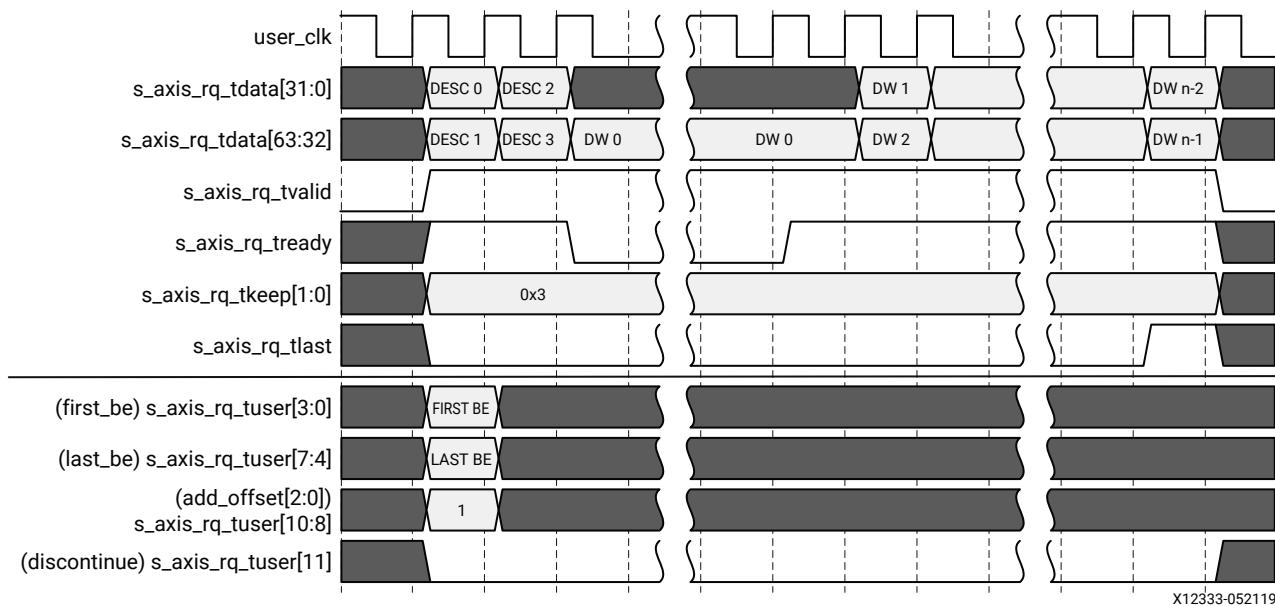
Figure 47: Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, 256-Bit Interface)



The following timing diagrams illustrate the address-aligned transfer of a memory write request from the user application across the RQ interface, when the interface width is configured as 64, 128, and 256 bits, respectively. For illustration purposes, the starting Dword offset of the data block being written into user application memory is assumed to be $(m \times 32 + 1)$, for some integer $m > 0$. Its size is assumed to be n Dwords, for some $n = k \times 32 + 29$, $k > 0$.

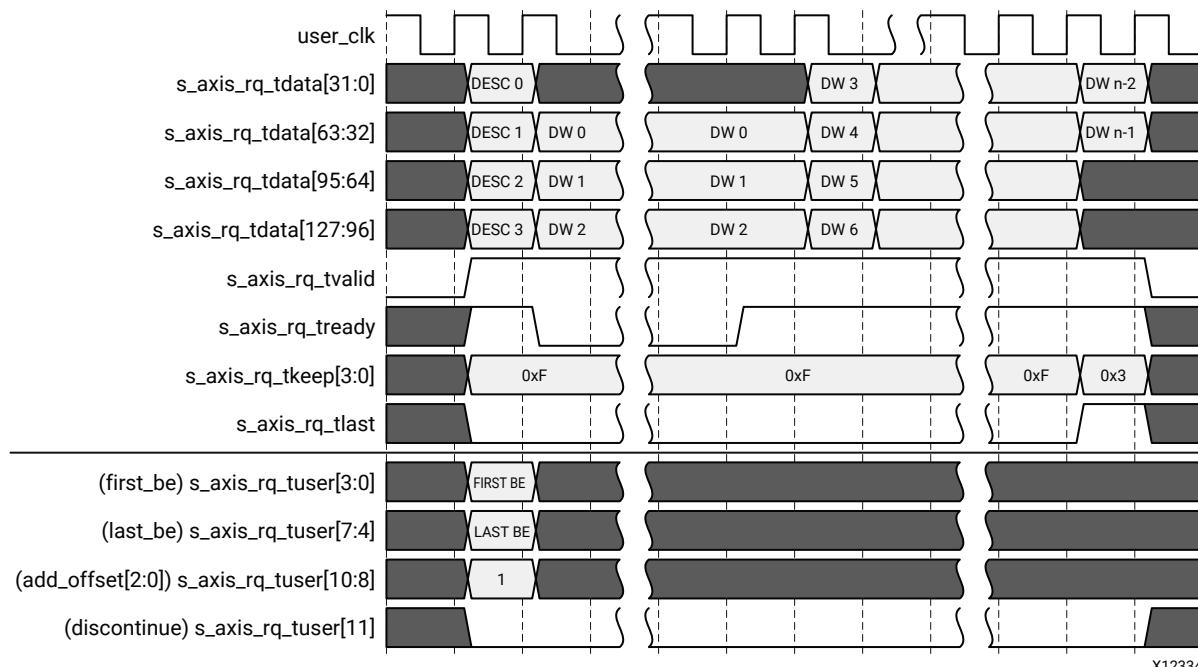
In the address-aligned mode, the delivery of the payload always starts in the beat following the last byte of the descriptor. The first Dword of the payload can appear at any Dword position. The user application must communicate the offset of the first Dword of the payload on the datapath using the `addr_offset[2:0]` signal in `s_axis_rq_tuser`. The user application must also set the bits in `first_be[3:0]` to indicate the valid bytes in the first Dword and the bits in `last_be[3:0]` to indicate the valid bytes in the last Dword of the payload.

Figure 48: Memory Write Transaction on the Requester Request Interface (Address-Aligned Mode, 64-Bit Interface)



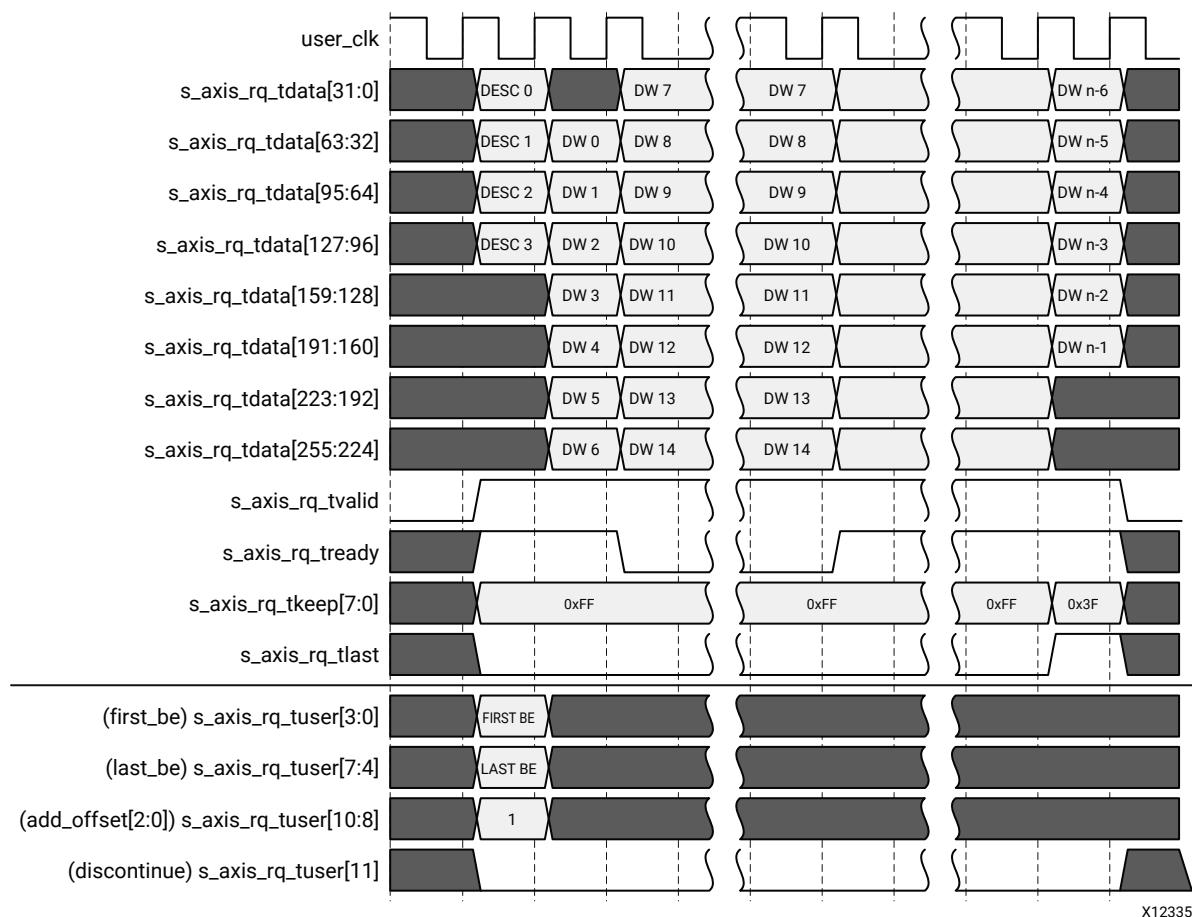
X12333-052119

Figure 49: Memory Write Transaction on the Requester Request Interface (Address-Aligned Mode, 128-Bit Interface)



X12334

Figure 50: Memory Write Transaction on the Requester Request Interface (Address-Aligned Mode, 256-Bit Interface)



Non-Posted Transactions with No Payload

Non-Posted transactions with no payload (memory read requests, I/O read requests, Configuration read requests) are transferred across the RQ interface in the same manner as a memory write request, except that the AXI4-Stream packet contains only the 16-byte descriptor. The following timing diagrams illustrate the transfer of a memory read request across the RQ interface, when the interface width is configured as 64, 128, and 256 bits, respectively. The packet occupies two consecutive beats on the 64-bit interface, while it is transferred in a single beat on the 128- and 256-bit interfaces. The `s_axis_rq_tvalid` signal must remain asserted over the duration of the packet. The integrated block can deassert `s_axis_rq_tready` to prolong the beat. The `s_axis_rq_tlast` signal must be set in the last beat of the packet, and the bits in `s_axis_rq_tkeep[7:0]` must be set in all Dword positions where a descriptor is present.

The valid bytes in the first and last Dwords of the data block to be read must be indicated using `first_be[3:0]` and `last_be[3:0]`, respectively. For the special case of a zero-length memory read, the length of the request must be set to one Dword, with both `first_be[3:0]` and `last_be[3:0]` set to all 0s. Additionally when in address-aligned mode, `addr_offset[2:0]` in `s_axis_rq_tuser` specifies the desired starting alignment of data returned on the Requester Completion interface. The alignment is not required to be correlated to the address of the request.

Figure 51: Memory Read Transaction on the Requester Request Interface (64-Bit Interface)

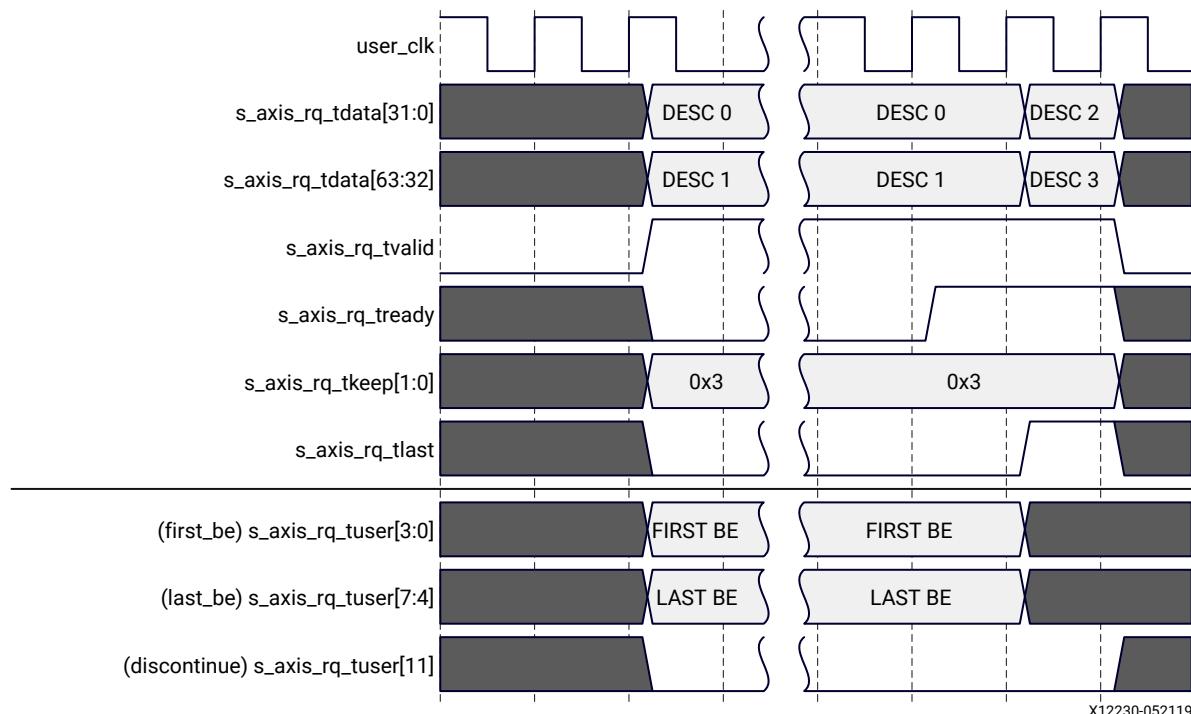


Figure 52: Memory Read Transaction on the Requester Request Interface (128-Bit Interface)

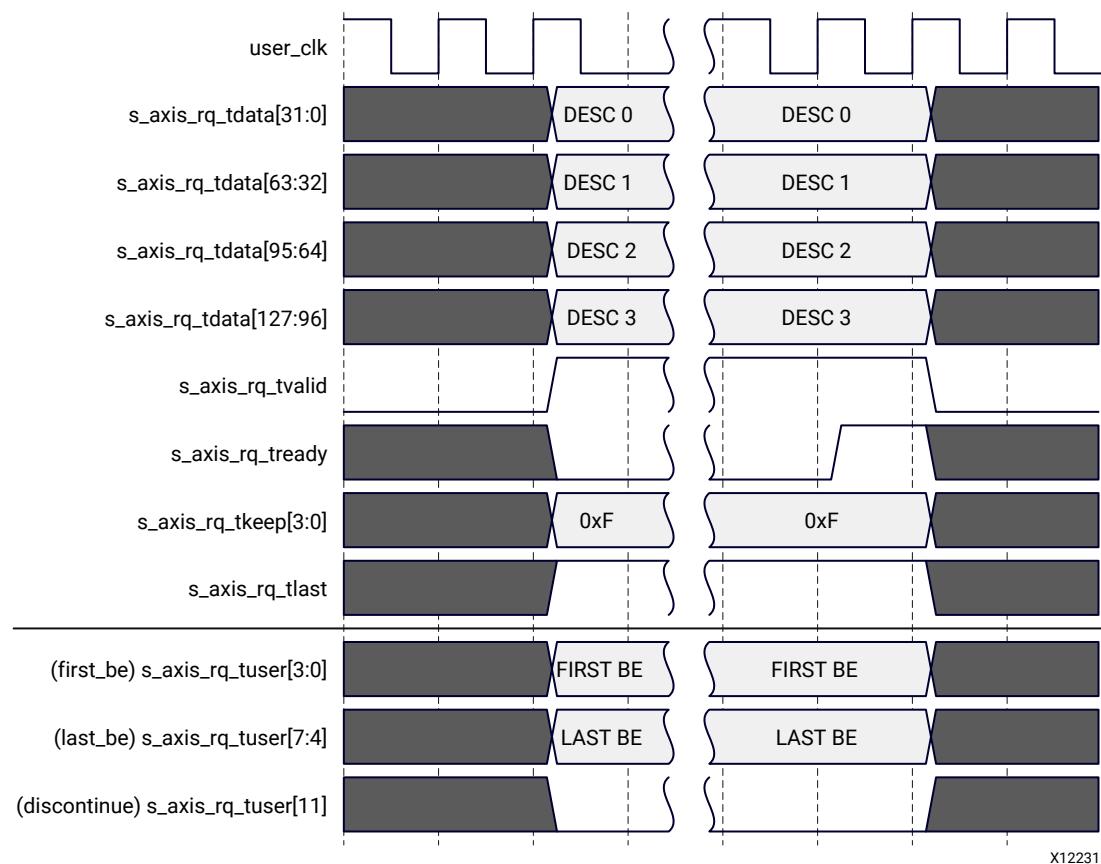
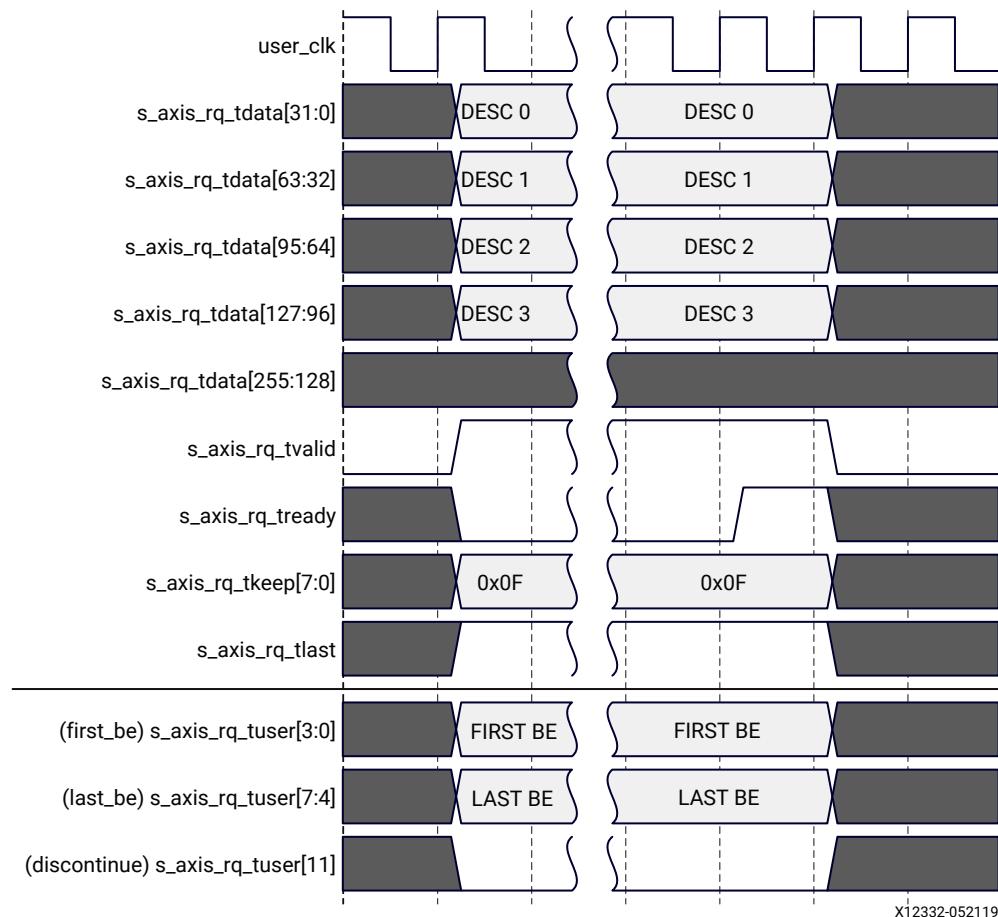


Figure 53: Memory Read Transaction on the Requester Request Interface (256-Bit Interface)



X12332-052119

Non-Posted Transactions with a Payload

The transfer of a Non-Posted request with payload (an I/O write request, Configuration write request, or Atomic Operation request) is similar to the transfer of a memory request, with the following changes in how the payload is aligned on the datapath:

- In the Dword-aligned mode, the first Dword of the payload follows the last Dword of the descriptor, with no gaps between them.
- In the address-aligned mode, the payload must start in the beat following the last Dword of the descriptor. The payload can start at any Dword position on the datapath. The offset of its first Dword must be specified using the `addr_offset [2 : 0]` signal.

For I/O and Configuration write requests, the valid bytes in the one-Dword payload must be indicated using `first_be [3 : 0]`. For Atomic Operation requests, all bytes in the first and last Dwords are assumed valid.

Message Requests on the Requester Interface

The transfer of a message on the RQ interface is similar to that of a memory write request, except that a payload might not always be present. The transfer starts with the 128-bit descriptor, followed by the payload, if present. When the Dword-aligned mode is in use, the first Dword of the payload must immediately follow the descriptor. When the address-alignment mode is in use, the payload must start in the beat following the descriptor, and must be aligned to byte lane 0. The `addr_offset` input to the integrated block must be set to 0 for messages when the address-aligned mode is in use. The integrated block determines the end of the payload from `s_axis_rq_tlast` and `s_axis_rq_tkeep` signals. The First Byte Enable and Last Byte Enable bits (`first_be` and `last_be`) are not used for message requests.

Aborting a Transfer

For any request that includes an associated payload, the user application can abort the request at any time during the transfer of the payload by asserting the `discontinue` signal in the `s_axis_rq_tuser` bus. The integrated block nullifies the corresponding TLP on the link to avoid data corruption.

The user application can assert this signal in any cycle during the transfer, when the request being transferred has an associated payload. The user application can either choose to terminate the packet prematurely in the cycle where the error was signaled (by asserting `s_axis_rq_tlast`), or can continue until all bytes of the payload are delivered to the integrated block. In the latter case, the integrated block treats the error as sticky for the following beats of the packet, even if the user application deasserts the `discontinue` signal before reaching the end of the packet.

The `discontinue` signal can be asserted only when `s_axis_rq_tvalid` is active-High. The integrated block samples this signal when `s_axis_rq_tvalid` and `s_axis_rq_tready` are both active-High. Thus, after assertion, the `discontinue` signal should not be deasserted until `s_axis_rq_tready` is active-High.

When the integrated block is configured as an Endpoint, this error is reported by the integrated block to the Root Complex it is attached to, as an Uncorrectable Internal Error using the Advanced Error Reporting (AER) mechanisms.

Tag Management for Non-Posted Transactions

The requester side of the integrated block maintains the state of all pending Non-Posted transactions (memory reads, I/O reads and writes, configuration reads and writes, Atomic Operations) initiated by the user application, so that the completions returned by the targets can be matched against the corresponding requests. The state of each outstanding transaction is held in a Split Completion Table in the requester side of the interface, which has a capacity of 256 Non-Posted transactions. The returning Completions are matched with the pending requests using an 8-bit tag. There are two options for management of these tags.

- **Internal Tag Management:** This mode of operation is selected by setting the Enable Client Tag checkbox is deselected (not set) in the Vivado® IDE, which is the default setting for the core. In this mode, logic within the integrated block is responsible for allocating the tag for each Non-Posted request initiated from the requester side. The integrated block maintains a list of free tags and assigns one of them to each request when the user application initiates a Non-Posted transaction, and communicates the assigned tag value to the user application through the output `pcie_rq_tag0[7:0]`. The value on this bus is valid when the integrated block asserts `pcie_rq_tag_vld0`. The user logic must copy this tag so that any Completions delivered by the integrated block in response to the request can be matched to the request.

In this mode, logic within the integrated block checks for the Split Completion Table full condition, and back pressures a Non-Posted request from the user application (using `s_axis_rq_tready`) if the total number of Non-Posted requests currently outstanding has reached its limit (256).

- **External Tag Management:** In this mode, the user logic is responsible for allocating the tag for each Non-Posted request initiated from the requester side. The user logic must choose the tag value without conflicting with the tags of all other Non-Posted transactions outstanding at that time, and must communicate this chosen tag value to the integrated block through the request descriptor. The integrated block still maintains the outstanding requests in its Split Completion Table and matches the incoming Completions to the requests, but does not perform any checks for the uniqueness of the tags, or for the Split Completion Table full condition.

When internal tag management is in use, the integrated block asserts `pcie_rq_tag_vld0` for one cycle for each Non-Posted request, after it has placed its allocated tag on `pcie_rq_tag[7:0]`. There can be a delay of several cycles between the transfer of the request on the RQ interface and the assertion of `pcie_rq_tag_vld0` by the integrated block to provide the allocated tag for the request. The user application can, meanwhile, continue to send new requests. The tags for requests are communicated on the `pcie_rq_tag0` bus in FIFO order, so it is easy to associate the tag value with the request it transferred. A tag is reused when the end-of-frame (EOF) of the last completion of a split completion is accepted by the user application.

Avoiding Head-of-Line Blocking for Posted Requests

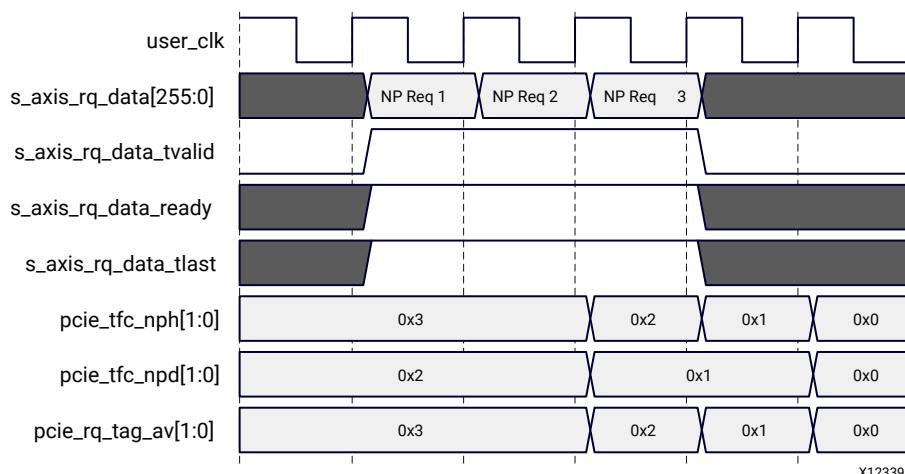
The integrated block can hold a Non-Posted request received on its RQ interface for lack of transmit credit or lack of available tags. This could potentially result in head-of-line (HOL) blocking for Posted transactions. The integrated block provides a mechanism for the user logic to avoid this situation through these signals:

- `pcie_tfc_nph_av[1:0]`: These outputs indicate the Header Credit currently available for Non-Posted requests, where:
 - 00 = no credit available
 - 01 = 1 credit
 - 10 = 2 credits
 - 11 = 3 or more credits

- pcie_tfc_npd_av[1:0]: These outputs indicate the Data Credit currently available for Non-Posted requests, where:
 - 00 = no credit available
 - 01 = 1 credit
 - 10 = 2 credits
 - 11 = 3 or more credits

The user logic can optionally check these outputs before transmitting Non-Posted requests. Because of internal pipeline delays, the information on these outputs is delayed by two user clock cycles from the cycle in which the last byte of the descriptor is transferred on the RQ interface. Thus, the user logic must adjust these values, taking into account any Non-Posted requests transmitted in the two previous clock cycles. The following figure illustrates the operation of these signals for the 256-bit interface. In this example, the integrated block initially had three Non-Posted Header Credits and two Non-Posted Data Credits, and had three free tags available for allocation. Request 1 from the user application had a one-Dword payload, and therefore consumed one header and data credit each, and also one tag. Request 2 in the next clock cycle consumed one header credit, but no data credit. When the user application presents Request 3 in the following clock cycle, it must adjust the available credit and available tag count by taking into account requests 1 and 2. If Request 3 consumes one header credit and one data credit, both available credits are 0 two cycles later, as also the number of available tags.

Figure 54: Credit and Tag Availability Signals on the Requester Request Interface (256-Bit Interface)



The following figures illustrate the timing of the credit and tag available signals for the same example, for interface widths of 128 bits and 64 bits, respectively.

Figure 55: Credit and Tag Availability Signals on the Requester Request Interface (128-Bit Interface)

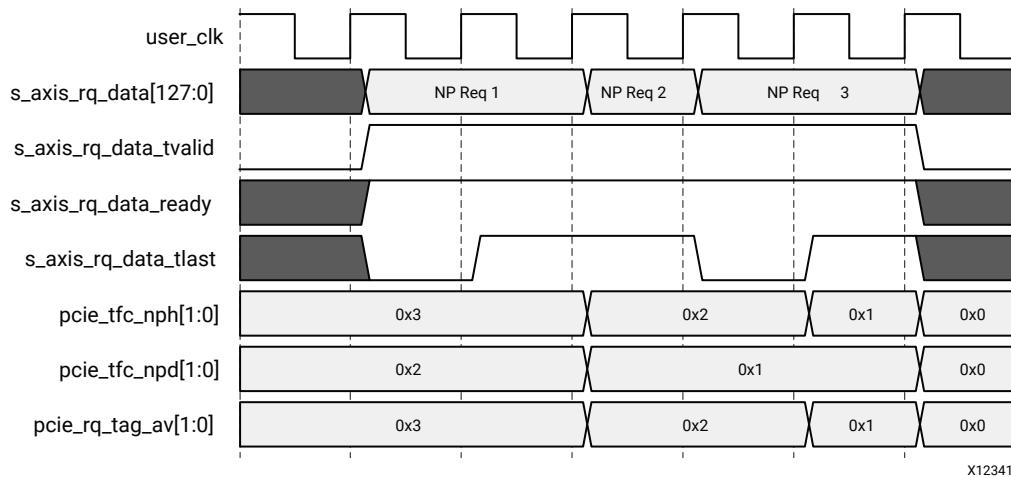
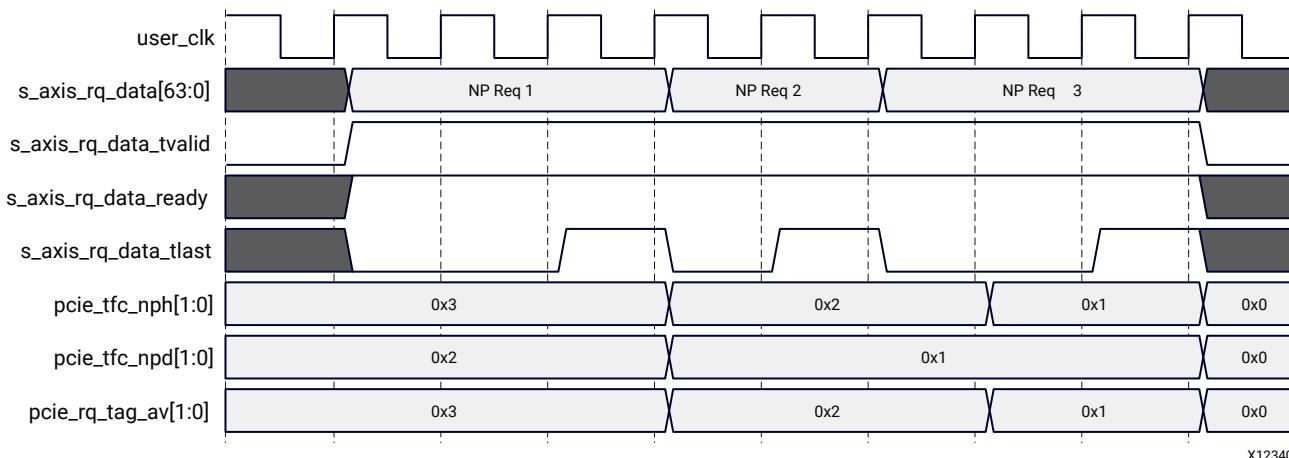


Figure 56: Credit and Tag Availability Signals on the Requester Request Interface (64-Bit Interface)



Note: If the user logic opts in to use the `pcie_tfc_*` interface to monitor transmit credit availability, ensure that no more non-posted packets go into the RQ interface after `pcie_tfc_npd_av` or `pcie_tfc_nph_av` reaches 0. The integrated block will not lose the non-posted packets issued beyond this point; however, the `pcie_tfc_*` interface no longer provides an accurate credit accounting.

Similar transmit credit information is also provided in the `cfg_fc_npd` and `cfg_fc_nph` interface when `cfg_fc_sel` is set to the Transmit credits available mode.

Maintaining Transaction Order

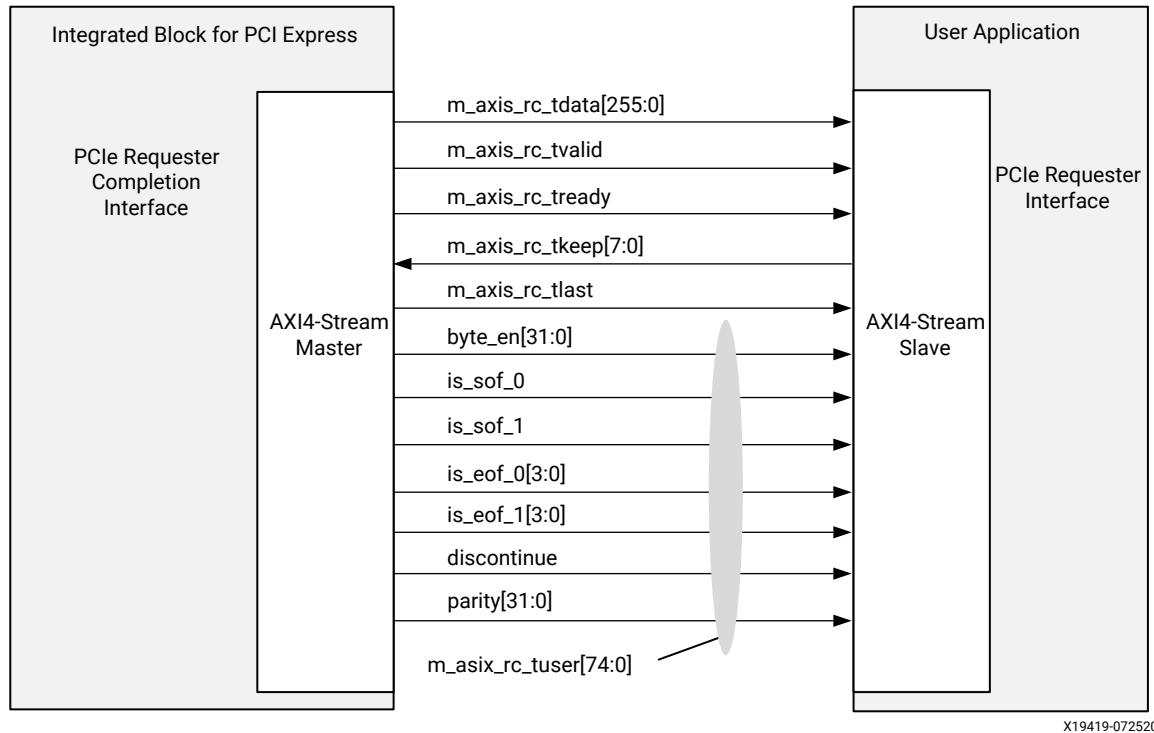
The integrated block does not change the order of requests received from the user application on its requester interface when it transmits them on the link. In cases where the user application would like to have precise control of the order of transactions sent on the RQ interface and the CC interface (typically to avoid Completions from passing Posted requests when using strict ordering), the integrated block provides a mechanism for the user application to monitor the progress of a Posted transaction through its pipeline, so that it can determine when to schedule a Completion on the completer completion interface without the risk of passing a specific Posted request transmitted from the requester request interface.

When transferring a Posted request (memory write transactions or messages) across the requester request interface, the user application can provide an optional 4-bit sequence number to the integrated block on its `seq_num[3:0]` input within `s_axis_rq_tuser`. The sequence number must be valid in the first beat of the packet. The user application can then monitor the `pcie_rq_seq_num[3:0]` output of the core for this sequence number to appear. When the transaction has reached a stage in the internal transmit pipeline of the integrated block where a Completion cannot pass it, the integrated block asserts `pcie_rq_seq_num_valid` for one cycle and provides the sequence number of the Posted request on the `pcie_rq_seq_num[3:0]` output. Any Completions transmitted by the integrated block after the sequence number has appeared on `pcie_rq_seq_num[3:0]` cannot pass the Posted request in the internal transmit pipeline.

Requester Completion Interface Operation

Completions for requests generated by the user logic are presented on the integrated block Request Completion (RC) interface. See the following figure for an illustration of signals associated with the requester completion interface. When straddle is not enabled, the integrated block delivers each TLP on this interface as an AXI4-Stream packet. The packet starts with a 96-bit descriptor, followed by data in the case of Completions with a payload.

Figure 57: Requester Completion Interface



The RC interface supports two distinct data alignment modes for transferring payloads. In the Dword-aligned mode, the integrated block transfers the first Dword of the Completion payload immediately after the last Dword of the descriptor. In the address-aligned mode, the integrated block starts the payload transfer in the beat following the last Dword of the descriptor, and its first Dword can be in any of the possible Dword positions on the datapath. The alignment of the first Dword of the payload is determined by an address offset provided by the user application when it sent the request to the integrated block (that is, the setting of the `addr_offset[2:0]` input of the RQ interface). Thus, the address-aligned mode can be used on the RC interface only if the RQ interface is also configured to use the address-aligned mode.

Requester Completion Descriptor Format

The RC interface of the integrated block sends completion data received from the link to the user application as AXI4-Stream packets. Each packet starts with a descriptor and can have payload data following the descriptor. The descriptor is always 12 bytes long, and is sent in the first 12 bytes of the completion packet. The descriptor is transferred during the first two beats on a 64-bit interface, and in the first beat on a 128- or 256-bit interface. When the completion data is split into multiple Split Completions, the integrated block sends each Split Completion as a separate AXI4-Stream packet, with its own descriptor.

The format of the Requester Completion descriptor is illustrated in the following figure. The individual fields of the RC descriptor are described in the following table.

Figure 58: Requester Completion Descriptor Format

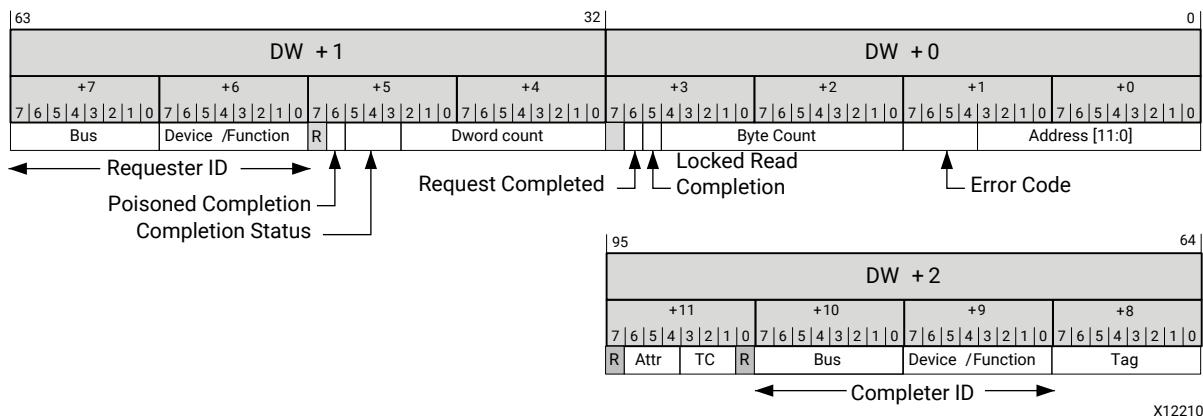


Table 54: Requester Completion Descriptor Fields

Bit Index	Field Name	Description
11:0	Lower Address	<p>This field provides the 12 least significant bits of the first byte referenced by the request. The integrated block returns this address from its Split Completion Table, where it stores the address and other parameters of all pending Non-Posted requests on the requester side.</p> <p>When the Completion delivered has an error, only bits [6:0] of the address should be considered valid.</p> <p>This is a byte-level address.</p> <p>For ATS translation requests, this field is reserved and implied to be zero.</p>

Table 54: Requester Completion Descriptor Fields (cont'd)

Bit Index	Field Name	Description
15:12	Error Code	<p>Completion error code.</p> <p>These three bits encode error conditions detected from error checking performed by the integrated block on received Completions. Its encodings are:</p> <ul style="list-style-type: none"> • 0000: Normal termination (all data received). • 0001: The Completion TLP is Poisoned. • 0010: Request terminated by a Completion with UR, CA or CRS status. • 0011: Request terminated by a Completion with no data, or the byte count in the Completion was higher than the total number of bytes expected for the request. • 0100: The current Completion being delivered has the same tag of an outstanding request, but its Requester ID, TC, or Attr fields did not match with the parameters of the outstanding request. • 0101: Error in starting address. The low address bits in the Completion TLP header did not match with the starting address of the next expected byte for the request. • 0110: Invalid tag. This Completion does not match the tags of any outstanding request. • 1001: Request terminated by a Completion timeout. The other fields in the descriptor, except bit [30], the requester Function [55:48], and the tag field [71:64], are invalid in this case, because the descriptor does not correspond to a Completion TLP. • 1000: Request terminated by a Function-Level Reset (FLR) targeted at the Function that generated the request. The other fields in the descriptor, except bit [30], the requester Function [55:48], and the tag field [71:64], are invalid in this case, because the descriptor does not correspond to a Completion TLP.
28:16	Byte Count	<p>These 13 bits can have values in the range of 0 – 4,096 bytes. If a Memory Read Request is completed using a single Completion, the Byte Count value indicates Payload size in bytes. This field must be set to 4 for I/O read Completions and I/O write Completions. The byte count must be set to 1 while sending a Completion for a zero-length memory read, and a dummy payload of 1 Dword must follow the descriptor.</p> <p>For each Memory Read Completion, the Byte Count field must indicate the remaining number of bytes required to complete the Request, including the number of bytes returned with the Completion.</p> <p>If a Memory Read Request is completed using multiple Completions, the Byte Count value for each successive Completion is the value indicated by the preceding Completion minus the number of bytes returned with the preceding Completion.</p>
29	Locked Read Completion	This bit is set to 1 when the Completion is in response to a Locked Read request. It is set to 0 for all other Completions.
30	Request Completed	<p>The integrated block asserts this bit in the descriptor of the last Completion of a request. The assertion of the bit can indicate normal termination of the request (because all data has been received) or abnormal termination because of an error condition. The user logic can use this indication to clear its outstanding request status.</p> <p>When tags are assigned, the user logic should not reassign a tag allocated to a request until it has received a Completion Descriptor from the integrated block with a matching tag field and the Request Completed bit set to 1.</p>

Table 54: Requester Completion Descriptor Fields (cont'd)

Bit Index	Field Name	Description
42:32	Dword Count	These 11 bits indicate the size of the payload of the current packet in Dwords. Its range is 0 - 1K Dwords. This field is set to 1 for I/O read Completions and 0 for I/O write Completions. The Dword count is also set to 1 while transferring a Completion for a zero-length memory read. In all other cases, the Dword count corresponds to the actual number of Dwords in the payload of the current packet.
45:43	Completion Status	These bits reflect the setting of the Completion Status field of the received Completion TLP. The valid settings are: <ul style="list-style-type: none"> • 000: Successful Completion • 001: Unsupported Request (UR) • 010: Configuration Request Retry Status (CRS) • 100: Completer Abort (CA)
46	Poisoned Completion	This bit is set to indicate that the Poison bit in the Completion TLP was set. Data in the packet should then be considered corrupted.
63:48	Requester ID	PCI Requester ID associated with the Completion.
71:64	Tag	PCIe Tag associated with the Completion.
87:72	Completer ID	Completer ID received in the Completion TLP. (These 16 bits are divided into an 8-bit bus number, 5-bit device number, and 3-bit function number in the legacy interpretation mode. In the ARI mode, these 16 bits must be treated as an 8-bit bus number + 8-bit Function number.).
91:89	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the Completion.
94:92	Attributes	PCIe attributes associated with the Completion. Bit 92 is the No Snoop bit, bit 93 is the Relaxed Ordering bit, and bit 94 is the ID-Based Ordering bit.

Transfer of Completions with No Data

The following timing diagrams illustrate the transfer of a Completion TLP received from the link with no associated payload across the RC interface, when the interface width is configured as 64, 128, and 256 bits, respectively. The timing diagrams in this section assume that the Completions are not straddled on the 256-bit interface. The straddle feature is described in [Straddle Option for 256-Bit Interface](#).

Figure 59: Transfer of a Completion with no Data on the Requester Completion Interface (64-Bit Interface)

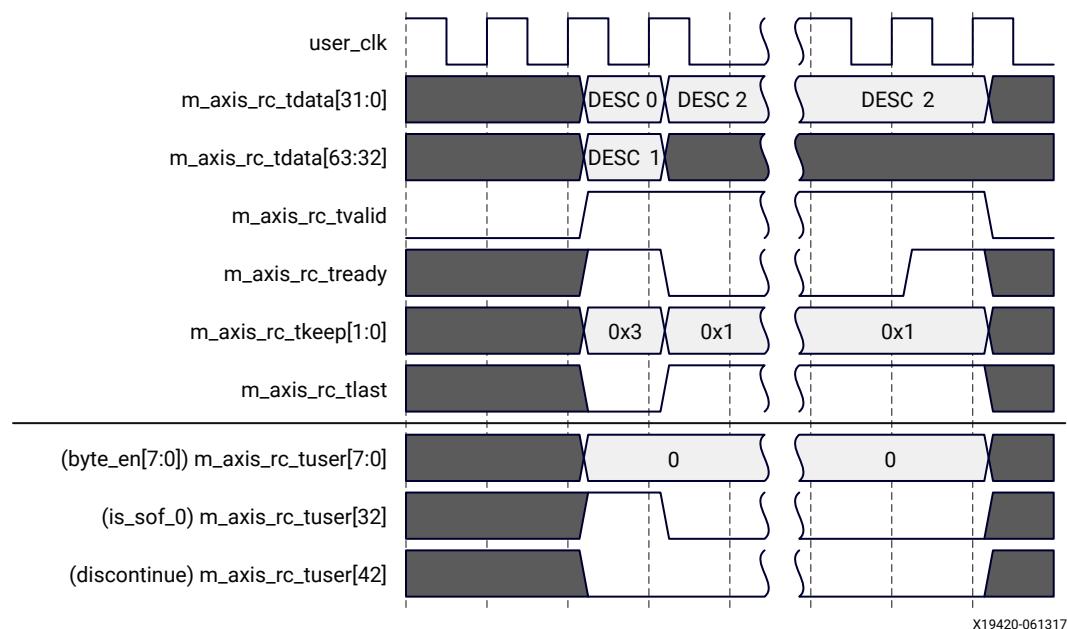


Figure 60: Transfer of a Completion with no Data on the Requester Completion Interface (128-Bit Interface)

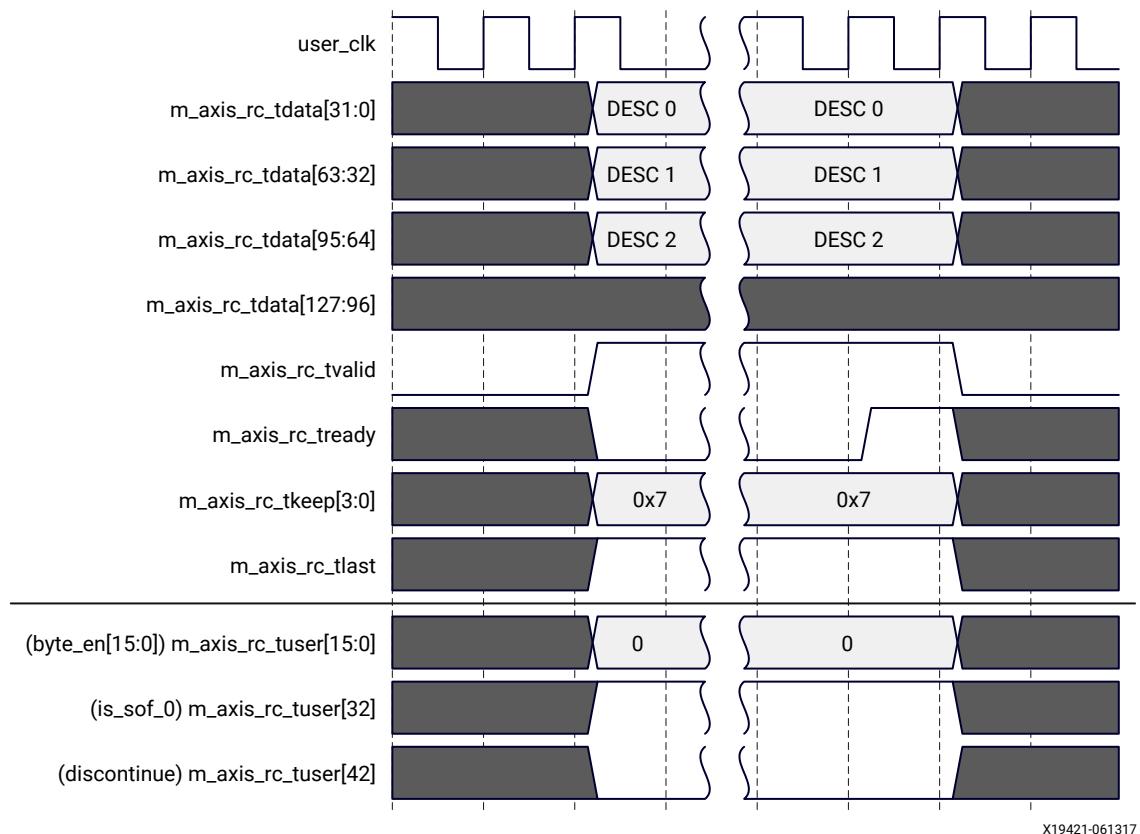
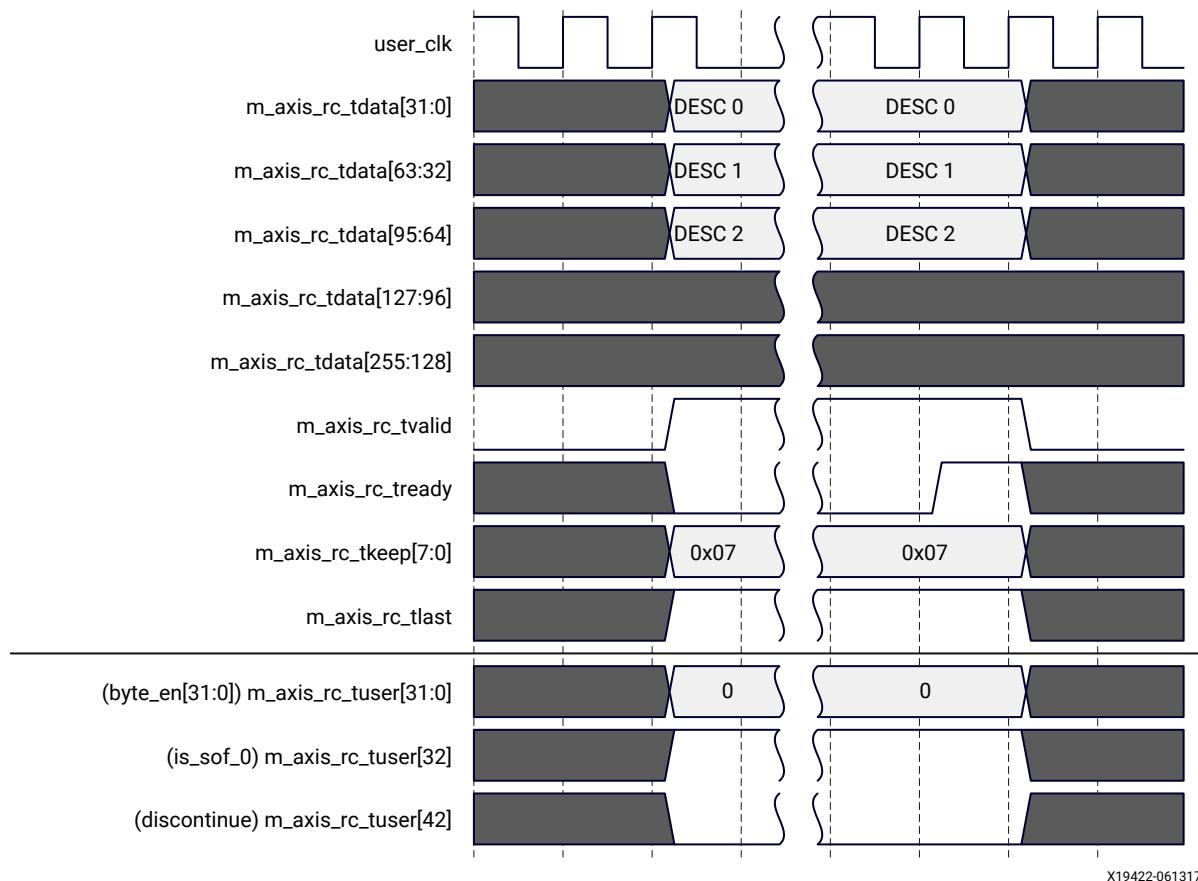


Figure 61: Transfer of a Completion with no Data on the Requester Completion Interface (256-Bit Interface)



The entire transfer of the Completion TLP takes only a single beat on the 256- and 128-bit interfaces, and two beats on the 64-bit interface. The integrated block keeps the `m_axis_rc_tvalid` signal asserted over the duration of the packet. The user application can prolong a beat at any time by deasserting `m_axis_rc_tready`. The AXI4-Stream interface signals `m_axis_rc_tkeep` (one per Dword position) indicate the valid descriptor Dwds in the packet. That is, the `tkeep` bits are set to 1 contiguously from the first Dword of the descriptor until its last Dword. During the transfer of a packet, the `tkeep` bits can be 0 only in the last beat of the packet. The `m_axis_rc_tlast` signal is always asserted in the last beat of the packet.

The `m_axi_rc_tuser` bus also includes an `is_sof_0` signal, which is asserted in the first beat of every packet. The user application can optionally use this signal to qualify the start of the descriptor on the interface. No other signals within `m_axi_rc_tuser` are relevant to the transfer of Completions with no data, when the straddle option is not in use.

Transfer of Completions with Data

The following timing diagrams illustrate the Dword-aligned transfer of a Completion TLP received from the link with an associated payload across the RC interface, when the interface width is configured as 64, 128, and 256 bits, respectively. For illustration purposes, the size of the data block being written into user application memory is assumed to be n Dwords, for some $n = k \times 32 + 28$, $k > 0$. The timing diagrams in this section assume that the Completions are not straddled on the 256-bit interface. The straddle feature is described in [Straddle Option for 256-Bit Interface](#).

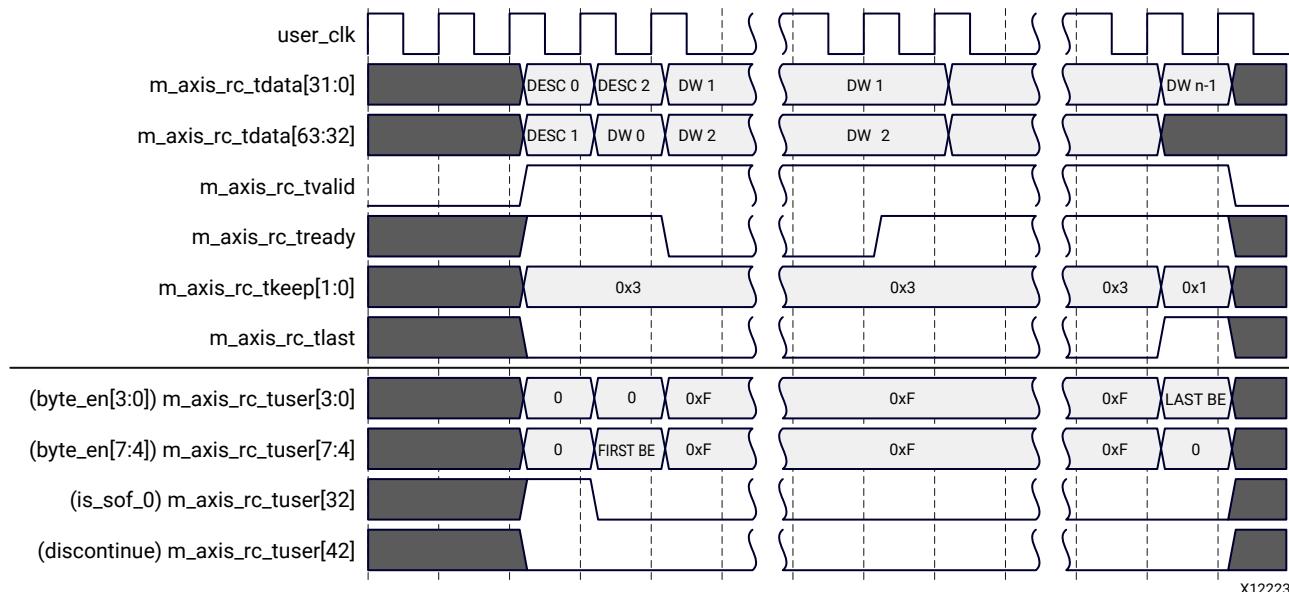
In the Dword-aligned mode, the transfer starts with the three descriptor Dwords, followed immediately by the payload Dwords. The entire TLP, consisting of the descriptor and payload, is transferred as a single AXI4-Stream packet. Data within the payload is always a contiguous stream of bytes when the length of the payload exceeds two Dwords. The positions of the first valid byte within the first Dword of the payload and the last valid byte in the last Dword can then be determined from the Lower Address and Byte Count fields of the Request Completion Descriptor. When the payload size is two Dwords or less, the valid bytes in the payload cannot be contiguous. In these cases, the user application must store the First Byte Enable and the Last Byte Enable fields associated with each request sent out on the RQ interface and use them to determine the valid bytes in the completion payload. The user application can optionally use the byte enable outputs `byte_en[31:0]` within the `m_axi_rc_tuser` bus to determine the valid bytes in the payload, in the cases of contiguous as well as non-contiguous payloads.

The integrated block keeps the `m_axis_rc_tvalid` signal asserted over the entire duration of the packet. The user application can prolong a beat at any time by deasserting `m_axis_rc_tready`. The AXI4-Stream interface signals `m_axis_rc_tkeep` (one per Dword position) indicate the valid Dwords in the packet including the descriptor and any null bytes inserted between the descriptor and the payload. That is, the `tkeep` bits are set to 1 contiguously from the first Dword of the descriptor until the last Dword of the payload. During the transfer of a packet, the `tkeep` bits can be 0 only in the last beat of the packet, when the packet does not fill the entire width of the interface. The `m_axis_rc_tlast` signal is always asserted in the last beat of the packet.

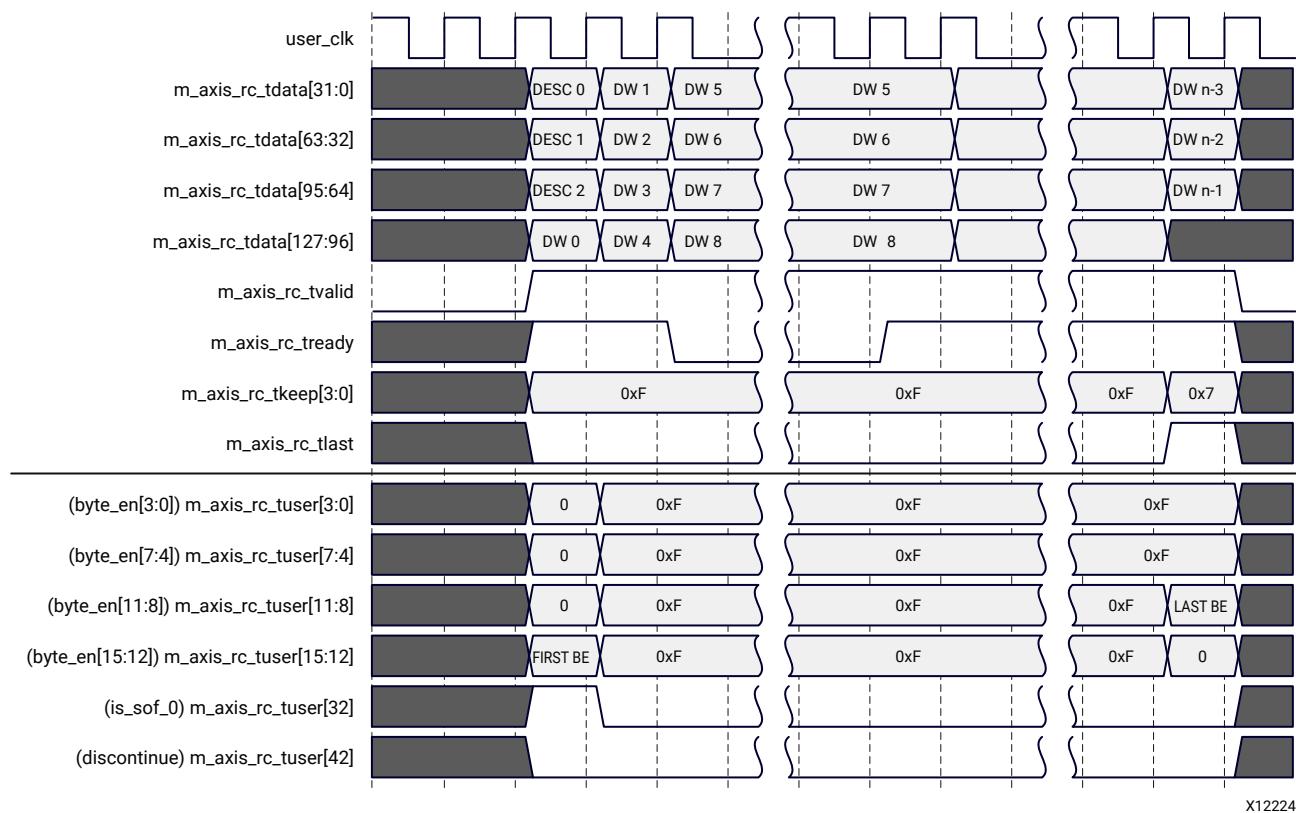
The `m_axi_rc_tuser` bus provides several informational signals that can be used to simplify the logic associated with the user application side of the interface, or to support additional features. The `is_sof_0` signal is asserted in the first beat of every packet, when its descriptor is on the bus. The byte enable outputs `byte_en[31:0]` (one per byte lane) indicate the valid bytes in the payload. These signals are asserted only when a valid payload byte is in the corresponding lane (it is not asserted for descriptor or null bytes). The asserted byte enable bits are always contiguous from the start of the payload, except when payload size is 2 Dwords or less. For Completion payloads of two Dwords or less, the 1s on `byte_en` might not be contiguous. Another special case is that of a zero-length memory read, when the integrated block transfers a one-Dword payload with the `byte_en` bits all set to 0. Thus, the user logic can, in all cases, use the `byte_en` signals directly to enable the writing of the associated bytes into memory.

The `is_sof_1`, `is_eof_0[3:0]`, and `is_eof_1[3:0]` signals within the `m_axis_rc_tuser` bus are not to be used for 64-bit and 128-bit interfaces, and for 256-bit interfaces when the straddle option is not enabled.

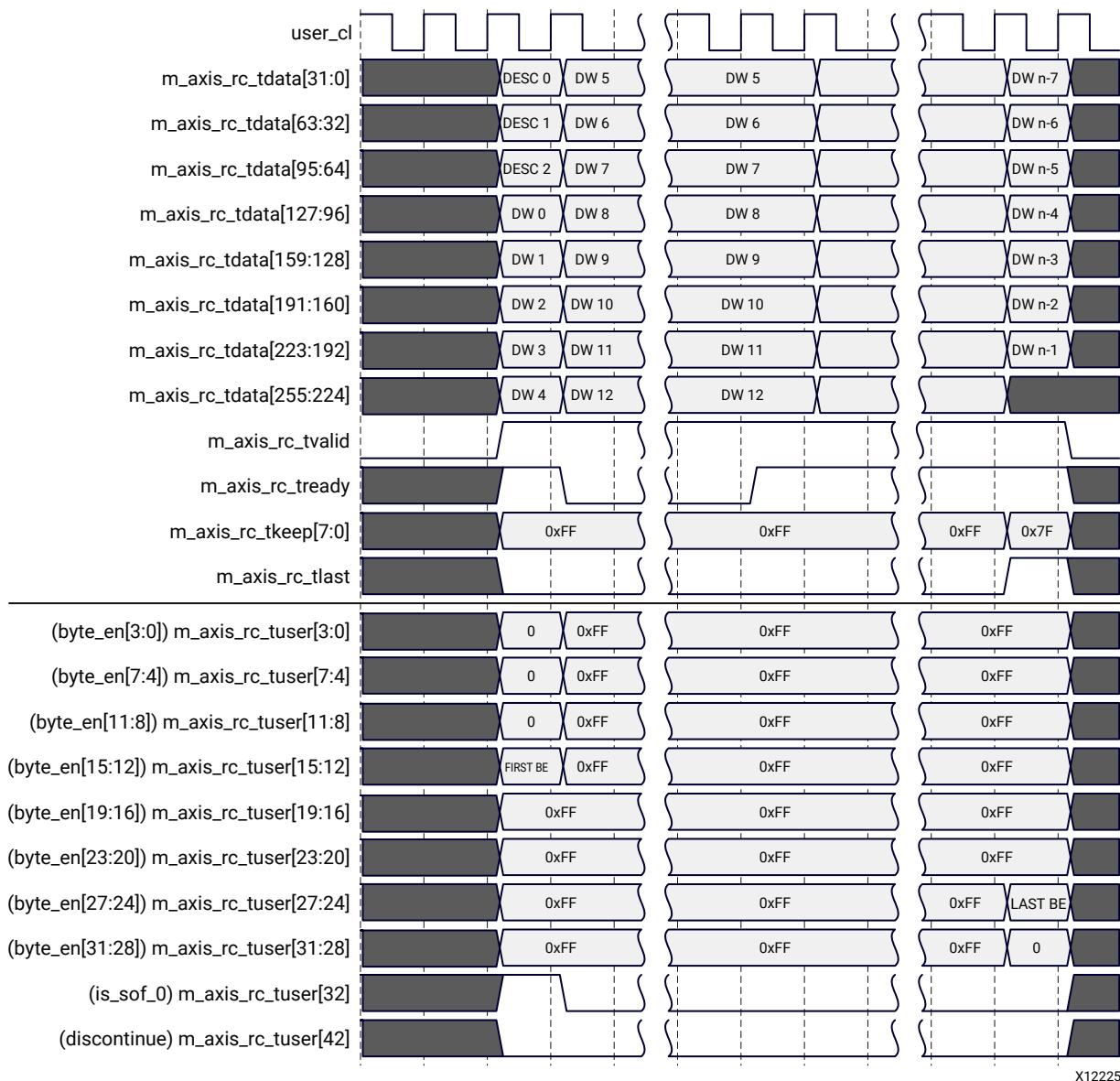
**Figure 62: Transfer of a Completion with Data on the Requester Completion Interface
(Dword-Aligned Mode, 64-Bit Interface)**



**Figure 63: Transfer of a Completion with Data on the Requester Completion Interface
(Dword-Aligned Mode, 128-Bit Interface)**



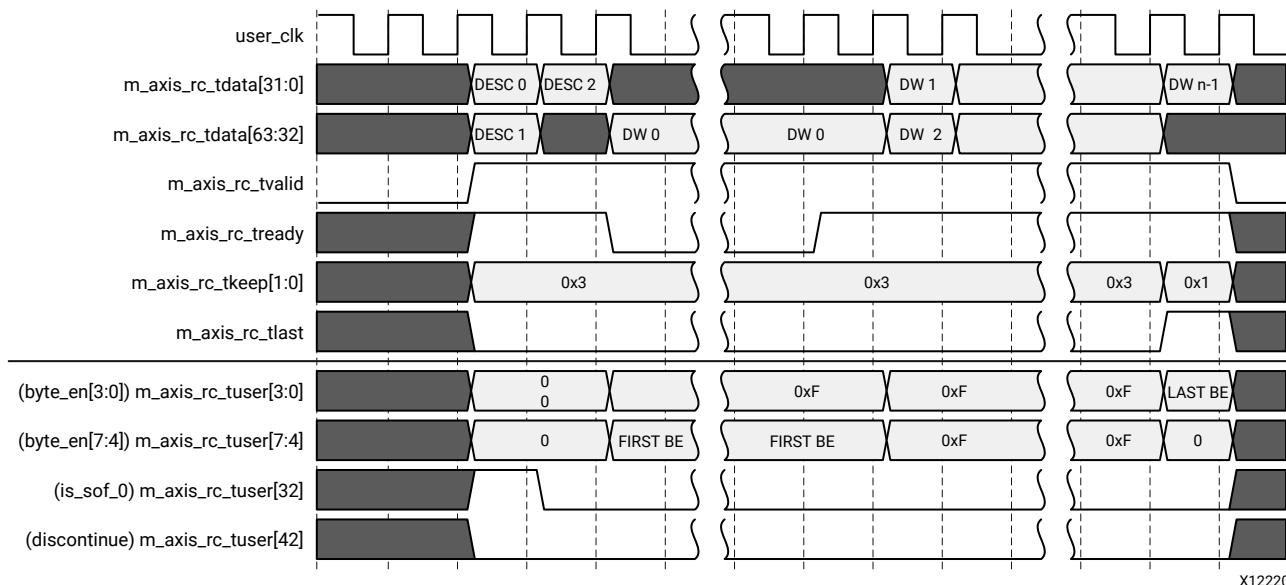
**Figure 64: Transfer of a Completion with Data on the Requester Completion Interface
(Dword-Aligned Mode, 256-Bit Interface)**



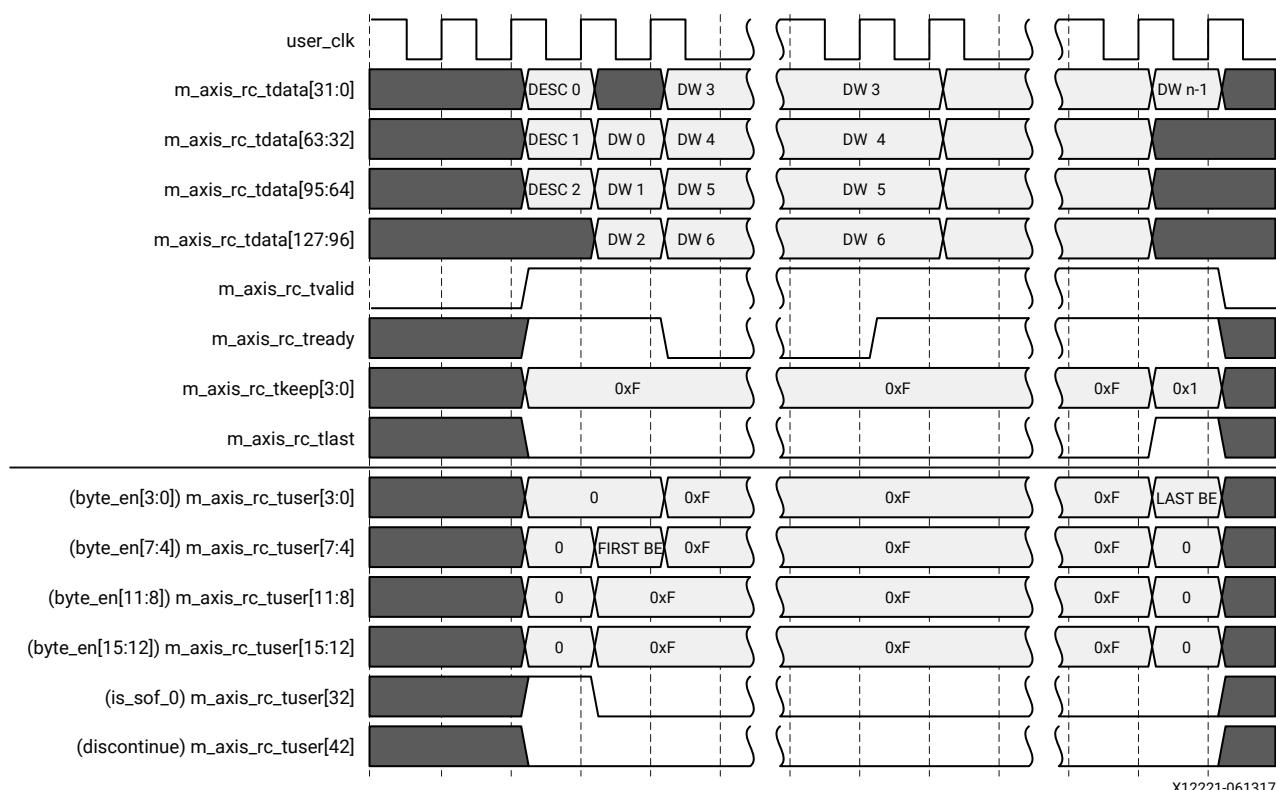
The following timing diagrams illustrate the address-aligned transfer of a Completion TLP received from the link with an associated payload across the RC interface, when the interface width is configured as 64, 128, and 256 bits, respectively. In the example timing diagrams, the starting Dword address of the data block being transferred (as conveyed in bits [6:2] of the Lower Address field of the descriptor) is assumed to be $(m \times 8 + 1)$, for an integer m . The size of the data block is assumed to be n Dwords, for some $n = k \times 32 + 28$, $k > 0$. The straddle option is not valid for address-aligned transfers, so the timing diagrams assume that the Completions are not straddled on the 256-bit interface.

In the address-aligned mode, the delivery of the payload always starts in the beat following the last byte of the descriptor. The first byte of the payload can appear on any byte lane, based on the address of the first valid byte of the payload. The `tkeep` bits are set to 1 contiguously from the first Dword of the descriptor until the last Dword of the payload. The alignment of the first Dword on the data bus is determined by the setting of the `addr_offset[2:0]` input of the requester request interface when the user application sent the request to the integrated block. The user application can optionally use the byte enable outputs `byte_en[31:0]` to determine the valid bytes in the payload.

Figure 65: Transfer of a Completion with Data on the Requester Completion Interface (Address-Aligned Mode, 64-Bit Interface)

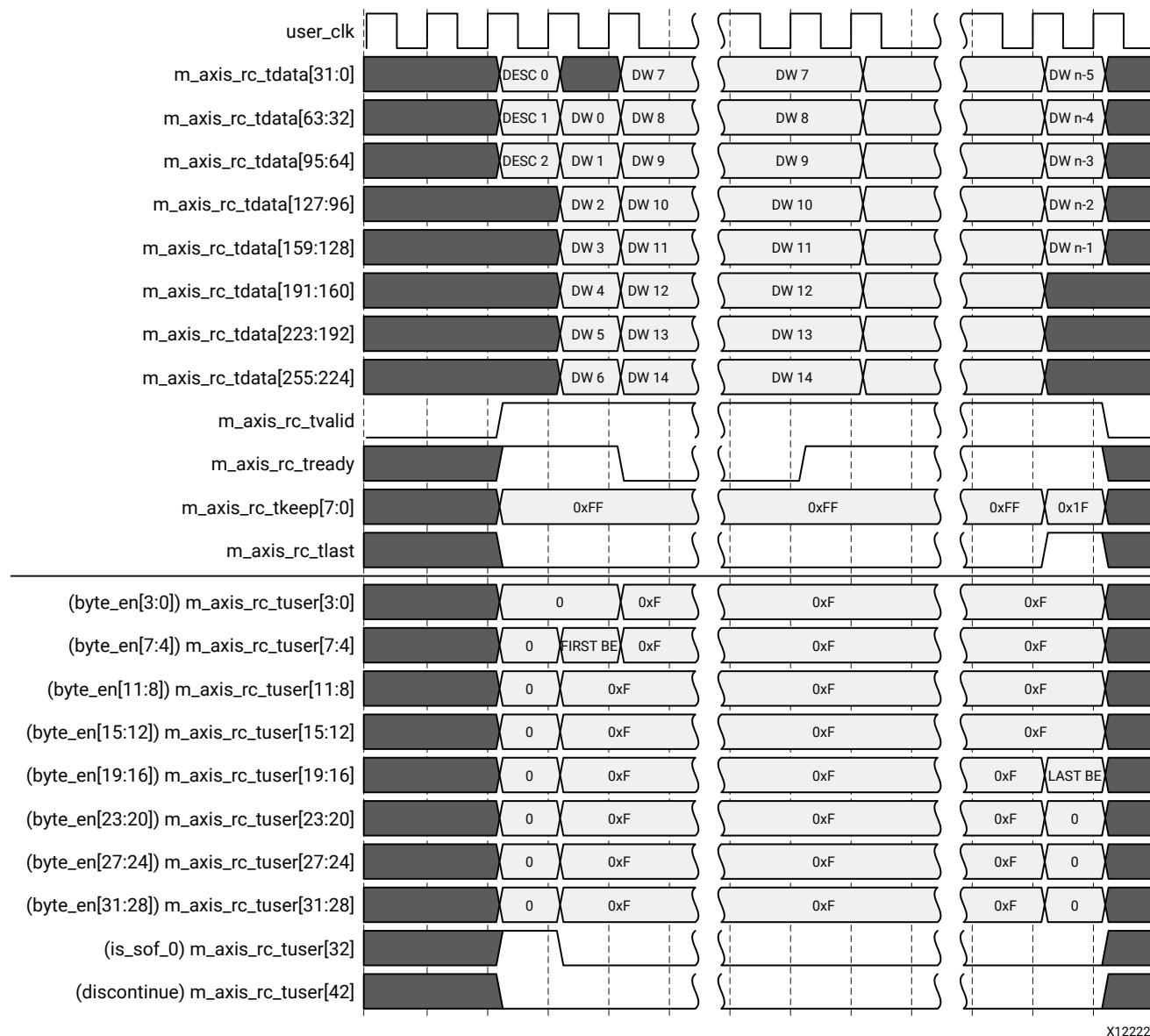


**Figure 66: Transfer of a Completion with Data on the Requester Completion Interface
(Address-Aligned Mode, 128-Bit Interface)**



X12221-061317

**Figure 67: Transfer of a Completion with Data on the Requester Completion Interface
(Address-Aligned Mode, 256-Bit Interface)**



Straddle Option for 256-Bit Interface

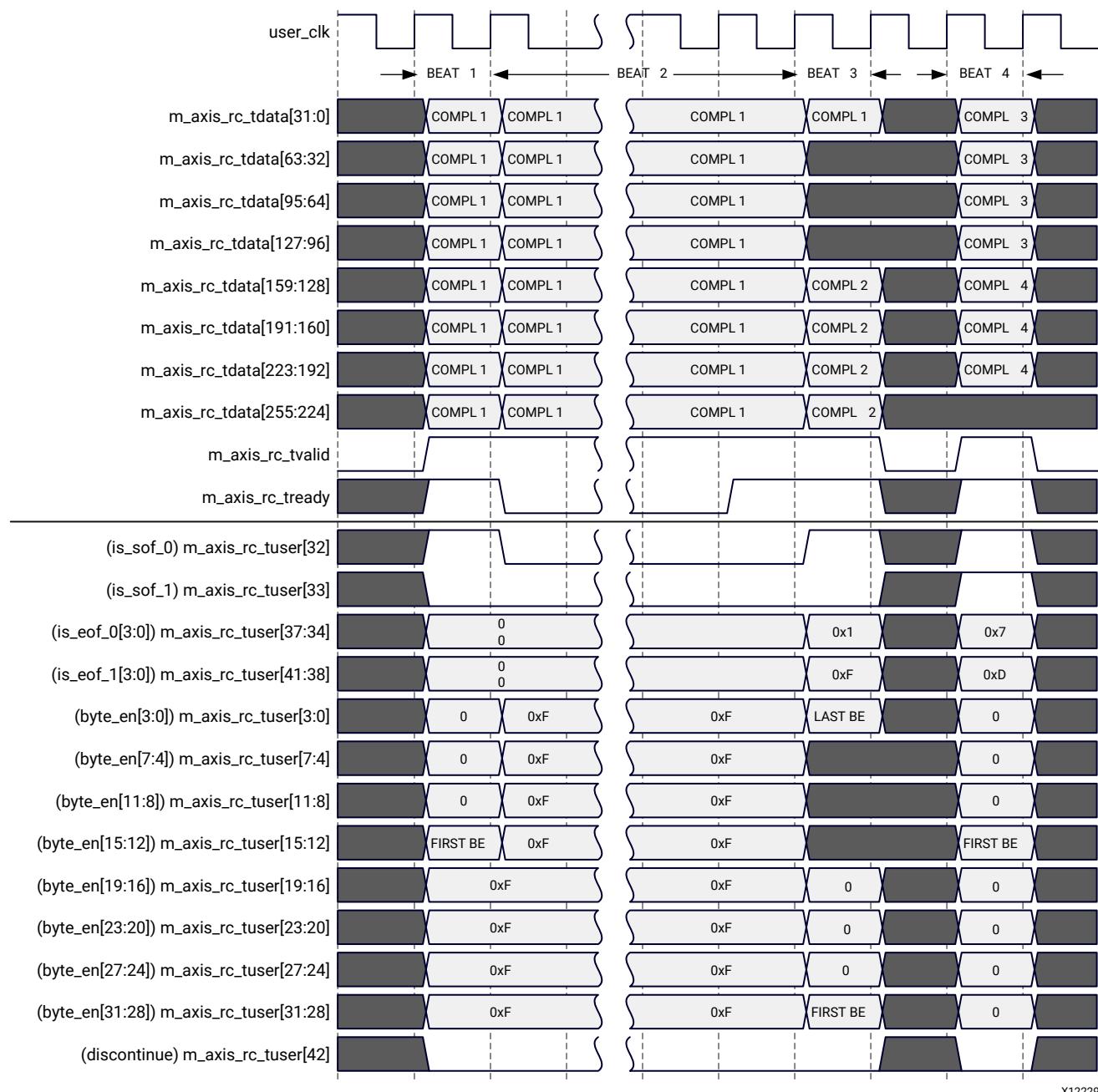
When the interface width is configured as 256 bits, the integrated block can start a new Completion transfer on the RC interface in the same beat when the previous Completion has ended on or before Dword position 3 on the data bus. The straddle option can be used only with the Dword-aligned mode.

When the straddle option is enabled, Completion TLPs are transferred on the RC interface as a continuous stream, with no packet boundaries (from an AXI4-Stream perspective). Thus, the `m_axis_rc_tkeep` and `m_axis_rc_tlast` signals are not useful in determining the boundaries of Completion TLPs delivered on the interface (the integrated block sets `m_axis_rc_tkeep` to all 1s and `m_axis_rc_tlast` to 0 permanently when the straddle option is in use). Instead, delineation of TLPs is performed using the following signals provided within the `m_axis_rc_tuser` bus:

- `is_sof_0`: The integrated block drives this output active-High in a beat when there is at least one Completion TLP starting in the beat. The position of the first byte of this Completion TLP is determined as follows:
 - If the previous Completion TLP ended before this beat, the first byte of this Completion TLP is in byte lane 0.
 - If a previous TLP is continuing in this beat, the first byte of this Completion TLP is in byte lane 16. This is possible only when the previous TLP ends in the current beat, that is when `is_eof_0[0]` is also set.
- `is_sof_1`: The integrated block asserts this output in a beat when there are two Completion TLPs starting in the beat. The first TLP always starts at byte position 0 and the second TLP at byte position 16. The integrated block starts a second TLP at byte position 16 only if the previous TLP ended before byte position 16 in the same beat, that is only if `is_eof_0[0]` is also set in the same beat.
- `is_eof_0[3:0]`: These outputs are used to indicate the end of a Completion TLP and the position of its last Dword on the data bus. The assertion of the bit `is_eof_0[0]` indicates that there is at least one Completion TLP ending in this beat. When bit 0 of `is_eof_0` is set, bits [3:1] provide the offset of the last Dword of the TLP ending in this beat. The offset for the last byte can be determined from the starting address and length of the TLP, or from the byte enable signals `byte_en[31:0]`. When there are two Completion TLPs ending in a beat, the setting of `is_eof_0[3:1]` is the offset of the last Dword of the first Completion TLP (in that case, its range is 0 through 3).
- `is_eof_1[3:0]`: The assertion of `is_eof_1[0]` indicates a second TLP ending in the same beat. When bit 0 of `is_eof_1` is set, bits [3:1] provide the offset of the last Dword of the second TLP ending in this beat. Because the second TLP can start only on byte lane 16, it can only end at a byte lane in the range 27–31. Thus the offset `is_eof_1[3:1]` can only take one of two values: 6 or 7. If `is_sof_1[0]` is active-High, the signals `is_eof_0[0]` and `is_sof_0` are also active-High in the same beat. If `is_sof_1` is active-High, `is_sof_0` is active-High. If `is_eof_1` is active-High, `is_eof_0` is active-High.

The following figure illustrates the transfer of four Completion TLPs on the 256-bit RC interface when the straddle option is enabled. The first Completion TLP (COMPL 1) starts at Dword position 0 of Beat 1 and ends in Dword position 0 of Beat 3. The second TLP (COMPL 2) starts in Dword position 4 of the same beat. This second TLP has only a one-Dword payload, so it also ends in the same beat. The third and fourth Completion TLPs are transferred completely in Beat 4, because Completion 3 has only a one-Dword payload and Completion 4 has no payload.

Figure 68: Transfer of Completion TLPs on the Requester Completion Interface with the Straddle Option Enabled



Aborting a Completion Transfer

For any Completion that includes an associated payload, the integrated block can signal an error in the transferred payload by asserting the discontinue signal in the `m_axis_rc_tuser` bus in the last beat of the packet. This occurs when the integrated block has detected an uncorrectable error while reading data from its internal memories. The user application must discard the entire packet when it has detected the discontinue signal asserted in the last beat of a packet. This is also considered a fatal error in the integrated block.

When the straddle option is in use, the integrated block does not start a second Completion TLP in the same beat when it has asserted discontinue, aborting the Completion TLP ending in the beat.

Handling of Completion Errors

When a Completion TLP is received from the link, the integrated block matches it against the outstanding requests in the Split Completion Table to determine the corresponding request, and compares the fields in its header against the expected values to detect any error conditions. The integrated block then signals the error conditions in a 4-bit error code sent to the user application as part of the completion descriptor. The integrated block also indicates the last completion for a request by setting the Request Completed bit (bit 30) in the descriptor. The following table defines the error conditions signaled by the various error codes.

Table 55: Encoding of Error Codes

Error Code	Description
0000	No errors detected.
0001	The Completion TLP received from the link was poisoned. The user application should discard any data that follows the descriptor. In addition, if the Request Completed bit in the descriptor is not set, the user application should continue to discard the data subsequent completions for this tag until it receives a completion descriptor with the Request Completed bit set. On receiving a completion descriptor with the Request Completed bit set, the user application can remove all state for the corresponding request.
0010	Request terminated by a Completion TLP with UR, CA, or CRS status. In this case, there is no data associated with the completion, and the Request Completed bit in the completion descriptor is set. On receiving such a Completion from the integrated block, the user application can discard the corresponding request.
0011	Read Request terminated by a Completion TLP with incorrect byte count. This condition occurs when a Completion TLP is received with a byte count not matching the expected count. The Request Completed bit in the completion descriptor is set. On receiving such a completion from the integrated block, the user application can discard the corresponding request.
0100	This code indicates the case when the current Completion being delivered has the same tag of an outstanding request, but its Requester ID, TC, or Attr fields did not match with the parameters of the outstanding request. The user application should discard any data that follows the descriptor. In addition, if the Request Completed bit in the descriptor is not set, the user application should continue to discard the data subsequent completions for this tag until it receives a completion descriptor with the Request Completed bit set. On receiving a completion descriptor with the Request Completed bit set, the user application can remove all state associated with the request.

Table 55: Encoding of Error Codes (cont'd)

Error Code	Description
0101	Error in starting address. The low address bits in the Completion TLP header did not match with the starting address of the next expected byte for the request. The user application should discard any data that follows the descriptor. In addition, if the Request Completed bit in the descriptor is not set, the user application should continue to discard the data subsequent Completions for this tag until it receives a completion descriptor with the Request Completed bit set. On receiving a completion descriptor with the Request Completed bit set, the user application can discard the corresponding request.
0110	Invalid tag. This error code indicates that the tag in the Completion TLP did not match with the tags of any outstanding request. The user application should discard any data following the descriptor.
0111	Invalid byte count. The byte count in the Completion was higher than the total number of bytes expected for the request. In this case, the Request Completed bit in the completion descriptor is also set. On receiving such a completion from the integrated block, the user application can discard the corresponding request.
1001	Request terminated by a Completion timeout. This error code is used when an outstanding request times out without receiving a Completion from the link. The integrated block maintains a completion timer for each outstanding request, and responds to a completion timeout by transmitting a dummy completion descriptor on the requester completion interface to the user application, so that the user application can terminate the pending request, or retry the request. Because this descriptor does not correspond to a Completion TLP received from the link, only the Request Completed bit (bit 30), the tag field (bits [71:64]) and the requester Function field (bits [55:48]) are valid in this descriptor.
1000	Request terminated by a Function-Level Reset (FLR) targeting the Function that generated the request. In this case, the integrated block transmits a dummy completion descriptor on the requester completion interface to the user application, so that the user application can terminate the pending request. Because this descriptor does not correspond to a Completion TLP received from the link, only the Request Completed bit (bit 30), the tag field (bits [71:64]) and the requester Function field (bits [55:48]) are valid in this descriptor.

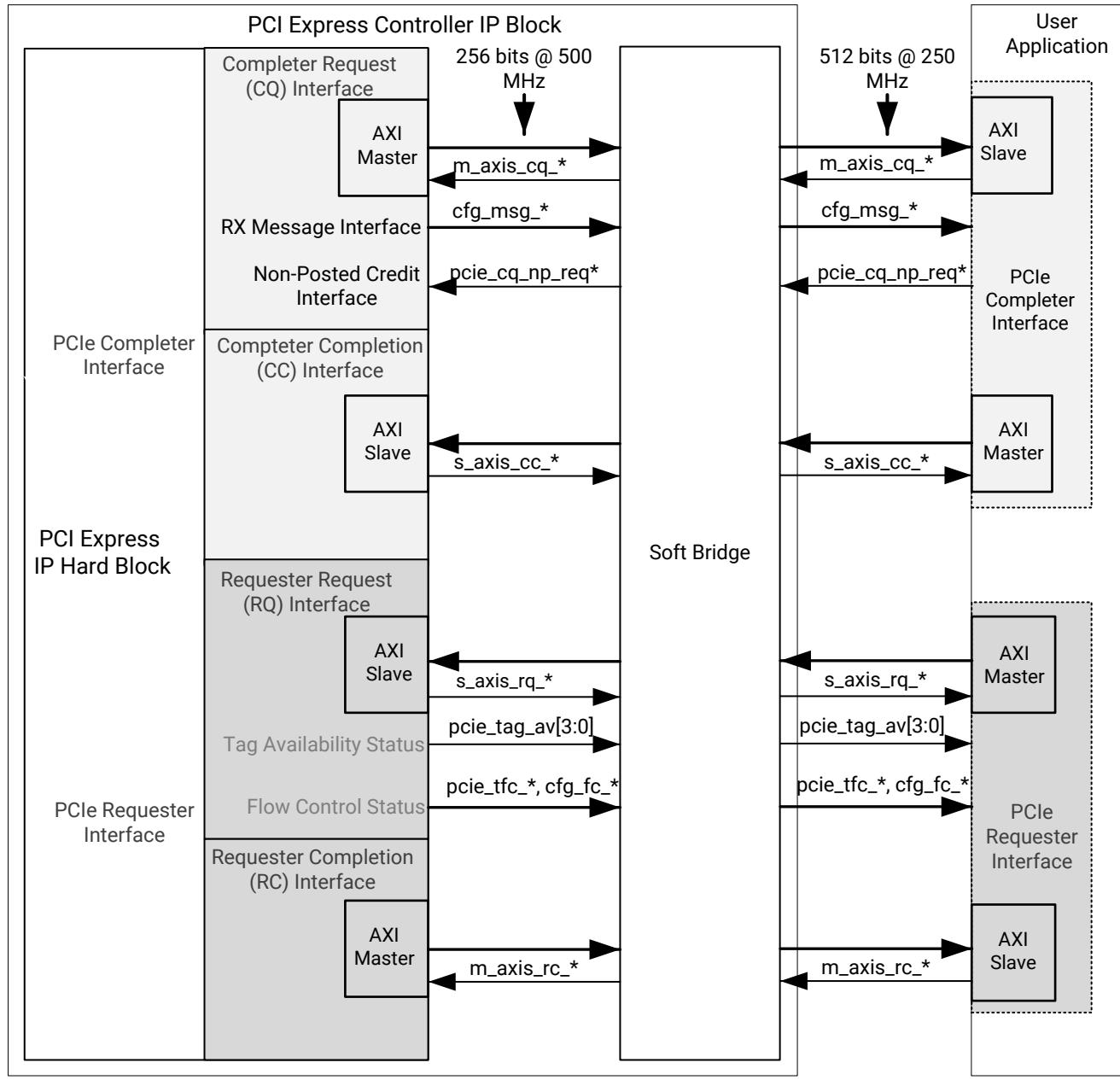
When the tags are managed internally by the integrated block, logic within the integrated block ensures that a tag allocated to a pending request is not reused until either all the Completions for the request were received or the request was timed out.

When tags are managed by the user application, however, the user application must ensure that a tag assigned to a request is not reused until the integrated block has signaled the termination of the request by setting the Request Completed bit in the completion descriptor. The user application can close out a pending request on receiving a completion with a non-zero error code, but should not free the associated tag if the Request Completed bit in the completion descriptor is not set. Such a situation might occur when a request receives multiple split completions, one of which has an error. In this case, the integrated block can continue to receive Completion TLPs for the pending request even after the error was detected, and these Completions are incorrectly matched to a different request if its tag is reassigned too soon. In some cases, the integrated block might have to wait for the request to time out even when a split completion is received with an error, before it can allow the tag to be reused.

512-Bit Completer Interface

This section describes the operation of the completer interface in the user-side interfaces associated with the 512-bit AXI4-Stream Interface. The following figure illustrates the connections between the soft bridge, PCIe® core and user application. The soft bridge converts the 256-bit packets at 500 MHz into 512-bit packets at 250 MHz.

Figure 69: Block Diagram of Core With Soft Bridge



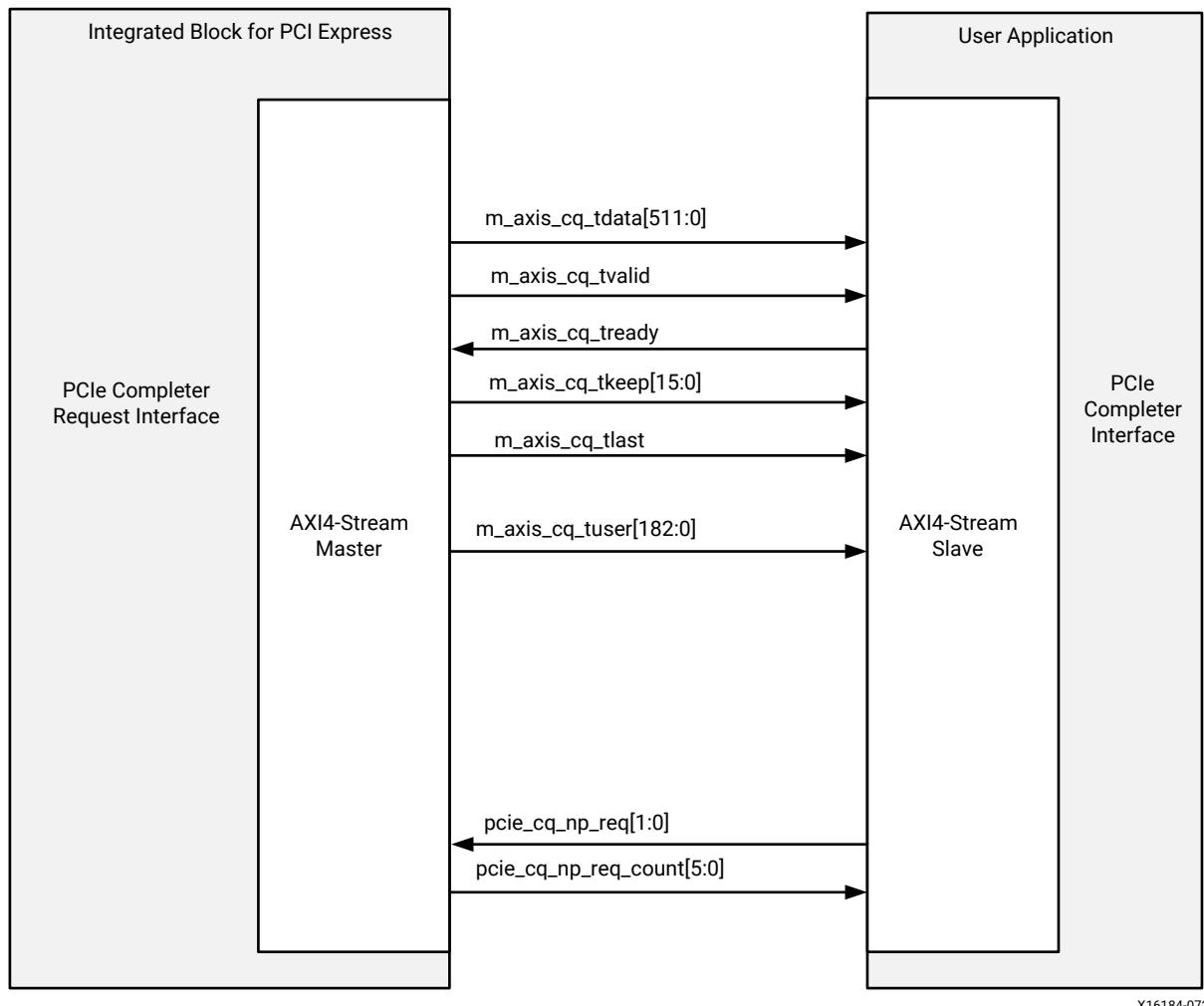
X16756-030217

The completer interface maps the transactions (memory, I/O read/write, messages, Atomic Operations) received from the PCIe link into transactions on the completer request interface based on the AXI4-Stream protocol. The completer interface is required to be connected to the user application in all PCIe Endpoint implementations, but is optional for Root Complexes. The completer interface consists of two separate interfaces, one for data transfer in each direction. Each interface is based on the AXI4-Stream protocol, with a data width of 512 bits. The completer request interface is for transfer of requests (with any associated payload data) to the user application, and the completer completion interface is for receiving the Completion data (for a Non-Posted request) from the user application for forwarding on the link. The two interfaces operate independently. That is, the core can transfer new requests over the completer request interface while receiving a Completion for a previous request.

Completer Request Interface Operation (512-bits)

The following figure illustrates the signals associated with the completer request interface of the core. The core delivers each TLP on this interface as an AXI4-Stream packet. The packet starts with a 128-bit descriptor, followed by data in the case of TLPs with a payload.

Figure 70: Completer Request Interface Signals



X16184-072520

The completer request interface supports two distinct data alignment modes, selected during core customization in the Vivado® IDE. In the Dword-aligned mode, the first byte of valid data appears in lane $n = S + 16 + (A \bmod 4) \bmod 64$, where A is the byte-level starting address of the data block being transferred and S is the lane number where the first byte of the descriptor appears. For messages and Configuration Requests, the address A is taken as 0. The starting lane number S is always 0 when the straddle option is not used, but can be 0 or 32 when straddle is enabled.

In the 128-bit address-aligned mode, the start of the payload on the 512-bit bus is always aligned on a 128-bit boundary. However the start of the descriptor on the 512-bit bus is always aligned to byte 0 or byte 32 only. The byte offset corresponding to the first byte of the payload is determined as $n = (S + 16 + (A \bmod 16)) \bmod 64$, where S is the byte offset where the first byte of the descriptor appears (which can be 0 or 32) and A is the memory or I/O address corresponding to the first byte of the payload. This means that the payload can start at one of four byte lanes: 16, 20, 24 and 28 if the descriptor starts at byte 0, or payload at one of four byte lanes: 48, 52, 56, and 60 if the descriptor starts at byte 32.

Any gap between the end of the descriptor and the start of the first byte of the payload is filled with null bytes.

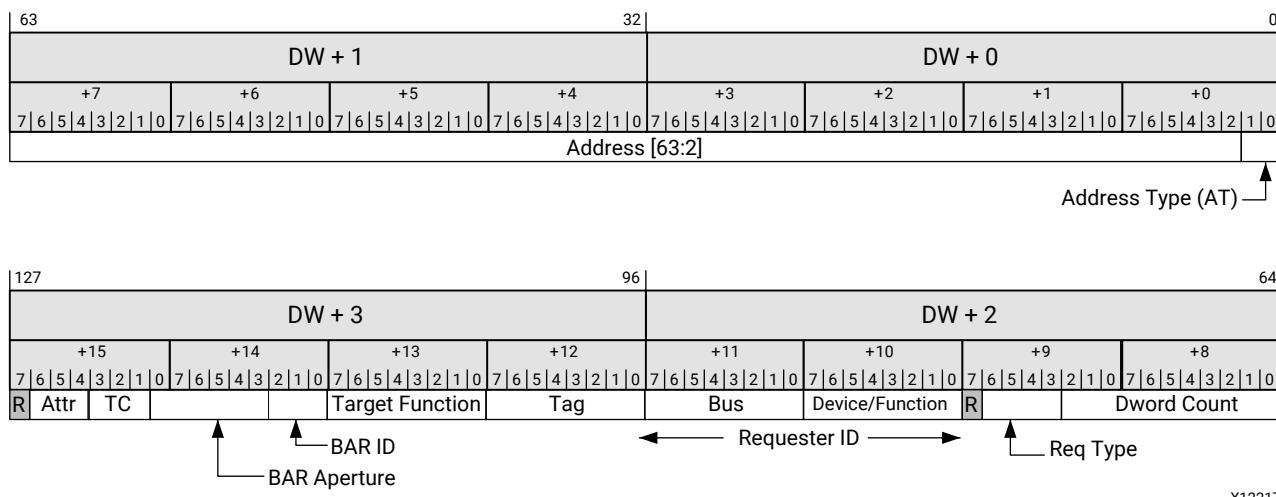
The interface also supports a straddle option that allows the transfer of up to two TLPs in the same beat across the interface. The straddle option can be used only with the Dword-aligned mode, and is not supported when using the 128-bit address aligned mode. The descriptions in the next sections assume a single TLP per beat. The operation of the interface with the straddle option enabled is described in [Straddle Option on CQ Interface](#).

Completer Request Descriptor Formats

The core transfers each request TLP received from the link over the completer request interface as an independent AXI4-Stream packet. Each packet starts with a descriptor, and can have payload data following the descriptor. The descriptor is always 16 bytes long, and is sent in the first 16 bytes of the request packet. The descriptor is always transferred during the first beat on the 512-bit interface. The formats of the descriptor for different request types are illustrated in the following figures.

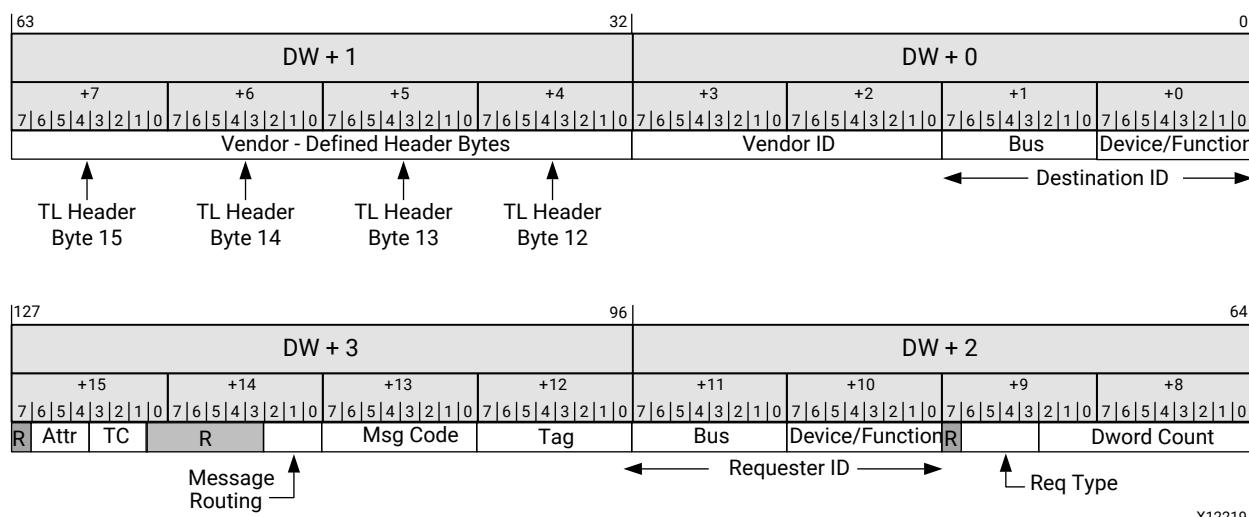
The format of the following figure applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request.

Figure 71: Completer Request Descriptor Format for Memory, I/O, and Atomic Op Requests



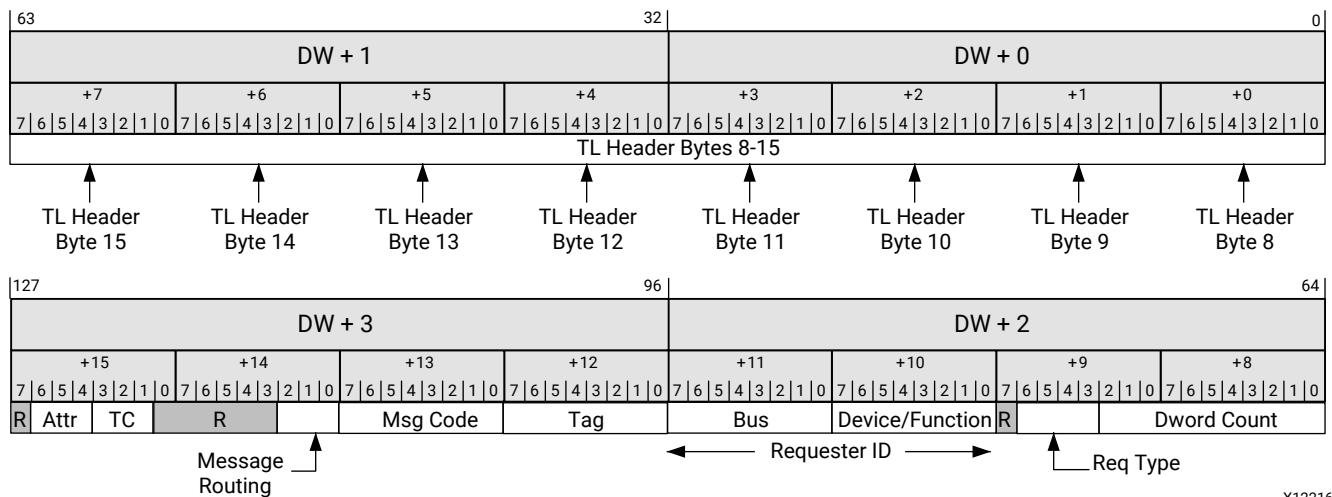
The format of the following figure is used for Vendor-Defined Messages (Type 0 or Type 1) only.

Figure 72: Completer Request Descriptor Format for Vendor-Defined Messages



The format of the following figure is used for all ATS messages (Invalid Request, Invalid Completion, Page Request, PRG Response).

Figure 73: Completer Request Descriptor Format for ATS Messages



For all other messages, the descriptor takes the format of the following figure.

Figure 74: Completer Request Descriptor Format for All Other Messages

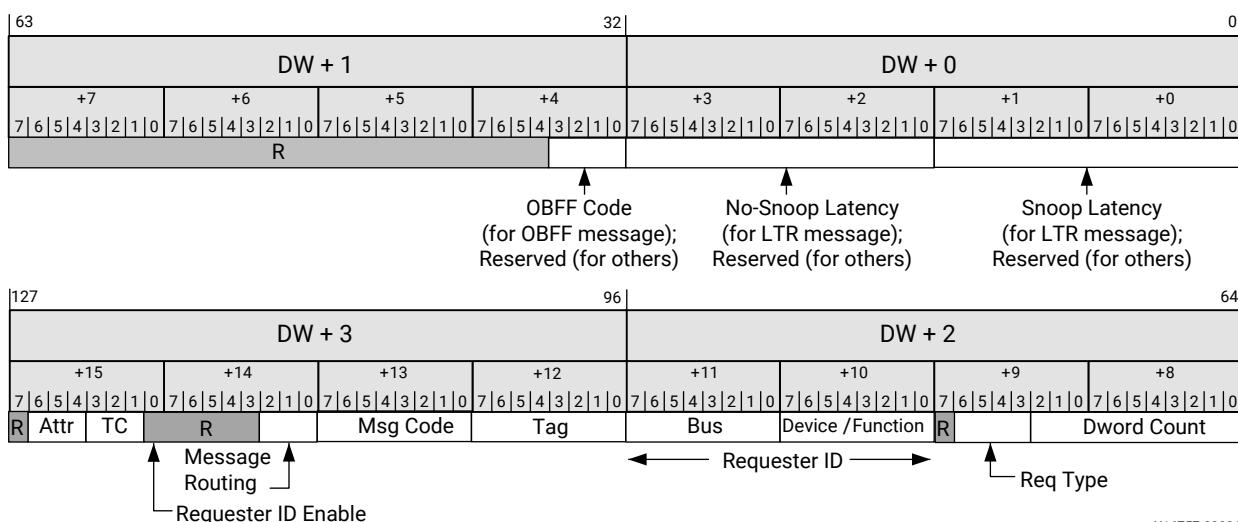


Table 56: Completer Request Descriptor Fields

Bit Index	Field Name	Description
1:0	Address Type	<p>This field is defined for memory transactions and Atomic Operations only. It contains the AT bits extracted from the TL header of the request.</p> <ul style="list-style-type: none"> • 00: Address in the request is un-translated • 01: Transaction is a Translation Request • 10: Address in the request is a translated address • 11: Reserved

Table 56: Completer Request Descriptor Fields (cont'd)

Bit Index	Field Name	Description
63:2	Address	<p>This field applies to memory, I/O and Atomic Op requests. It provides the address from the TL header. This is the address of the first Dword referenced by the request. The <code>First_BE</code> bits from <code>m_axis_cq_tuser</code> must be used to determine the byte-level address.</p> <p>When the transaction specifies a 32-bit address, bits [63:32] of this field is 0.</p>
74:64	Dword Count	<p>These 11 bits indicate the size of the block (in Dwords) to be read or written (for messages, size of the message payload). Its range is 0 – 256 Dwords. For I/O accesses, the Dword count is always 1.</p> <p>For a zero length memory read/write request, the Dword count is 1, with the <code>First_BE</code> bits set to all zeroes.</p>
78:75	Request Type	<p>Identifies the transaction type. The transaction types and their encodings are listed in Table 57: Transaction Types.</p>
95:80	Requester ID	<p>PCI Requester ID associated with the request. With the legacy interpretation of RIDs, these 16 bits are divided into an 8-bit bus number [95:88], 5-bit device number [87:83], and 3-bit function number [82:80]. When ARI is enabled, bits [95:88] carry the 8-bit bus number and [87:80] provide the function number.</p> <p>When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the core with the completion data.</p>
103:96	Tag	<p>PCIe Tag associated with the request. When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the core with the completion data. This field can be ignored for memory writes and messages.</p>
111:104	Target Function	<p>This field is defined for memory, I/O and Atomic Op requests only. It provides the function number the request is targeted at, determined by the BAR check. When ARI is in use, all 8 bits of this field are valid. Otherwise, only bits [106:104] are valid.</p>
114:112	BAR ID	<p>This field is defined for memory, I/O and Atomic Op requests only. It provides the matching BAR number for the address in the request.</p> <ul style="list-style-type: none"> • 000 = BAR 0 (VF-BAR 0 for VFs) • 001 = BAR 1 (VF-BAR 1 for VFs) • 010 = BAR 2 (VF-BAR 2 for VFs) • 011 = BAR 3 (VF-BAR 3 for VFs) • 100 = BAR 4 (VF-BAR 4 for VFs) • 101 = BAR 5 (VF-BAR 5 for VFs) • 110 = Expansion ROM Access <p>Note: In Root Port (RP) mode, BAR ID is always 000.</p> <p>For 64-bit transactions, the BAR number is given as the lower address of the matching pair of BARs (that is, 0, 2 or 4).</p>
120:115	BAR Aperture	<p>This 6-bit field is defined for memory, I/O and Atomic Op requests only. It provides the aperture setting of the BAR matching the request. This information is useful in determining the bits to be used by the user in addressing its memory or I/O space. For example, a value of 12 indicates that the aperture of the matching BAR is 4K, and the user can therefore ignore bits [63:12] of the address.</p> <p>For VF BARs, the value provided on this output is based on the memory space consumed by a single VF covered by the BAR.</p>

Table 56: Completer Request Descriptor Fields (cont'd)

Bit Index	Field Name	Description
123:121	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request. When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the core with the completion data.
126:124	Attributes	These bits provide the setting of the Attribute bits associated with the request. Bit 124 is the No Snoop bit and bit 125 is the Relaxed Ordering bit. Bit 126 is the ID-Based Ordering bit, and can be set only for memory requests and messages. When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the core with the completion data.
114:112	Message Routing	This field is defined for all messages. These bits provide the 3-bit Routing field r[2:0] from the TL header.
15:0	Destination ID	This field applies to Vendor-Defined Messages only. When the message is routed by ID (that is, when the Message Routing field is 010 binary), this field provides the Destination ID of the message.
63:32	Vendor-Defined Header	This field applies to Vendor-Defined Messages only. It contains the bytes extracted from Dword 3 of the TL header.
63:0	ATS Header	This field is applicable to ATS messages only. It contains the bytes extracted from Dwords 2 and 3 of the TL header.

Table 57: Transaction Types

Request Type (binary)	Description
0000	Memory Read Request
0001	Memory Write Request
0010	I/O Read Request
0011	I/O Write Request
0100	Memory Fetch and Add Request
0101	Memory Unconditional Swap Request
0110	Memory Compare and Swap Request
0111	Locked Read Request (allowed only in Legacy Devices)
1000	Type 0 Configuration Read Request (on Requester side only)
1001	Type 1 Configuration Read Request (on Requester side only)
1010	Type 0 Configuration Write Request (on Requester side only)
1011	Type 1 Configuration Write Request (on Requester side only)
1100	Any message, except ATS and Vendor-Defined Messages
1101	Vendor-Defined Message
1110	ATS Message
1111	Reserved

Completer Memory Write Operation

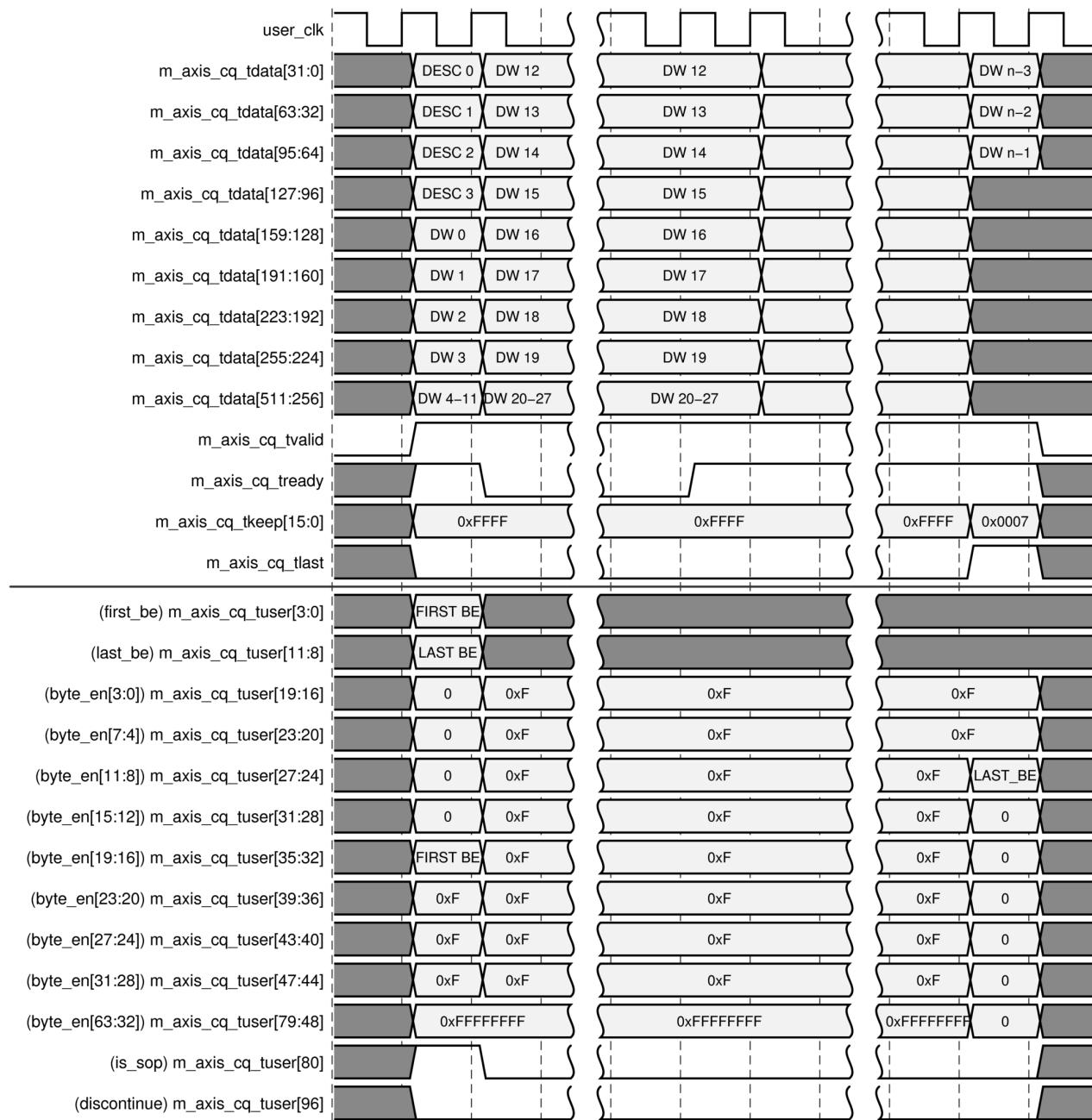
The following figure illustrates the Dword-aligned transfer of a memory write TLP received from the link across the completer request interface. For the purpose of illustration, the starting Dword address of the data block being written into user memory is assumed to be $(m * 16 + 3)$, for some integer $m > 0$. Its size is assumed to be n Dwords, for some $n = k * 16 - 1$, where $k > 1$.

The transfer starts with the sixteen descriptor bytes, followed immediately by the payload bytes. The signal `m_axis_cq_tvalid` remains asserted over the duration of the packet. The user logic can prolong a beat at any time by pulling down `m_axis_cq_tready`. The AXI4-Stream interface signals `m_axis_cq_tkeep` (one bit per Dword position) indicate the valid Dwords in the packet including the descriptor and any null bytes inserted between the descriptor and the payload. That is, the `m_axis_cq_tkeep` bits are set to 1 contiguously from the first Dword of the descriptor until the last Dword of the payload. During the transfer of a packet, the `tkeep` bits can be 0 only in the last beat of the packet, when the packet does not fill the entire width of the interface. The signal `m_axis_cq_tlast` is always asserted in the last beat of the packet.

The completer request interface also includes the First Byte Enable and the Last Enable bits in the `m_axis_cq_tuser` bus. These are activated in the first beat of the packet, and provides information of the valid bytes in the first and last Dwords of the payload.

The `m_axi_cq_tuser` bus also provides several optional signals that can be used to simplify the logic associated with the user side of the interface, or to support additional features. The signal `is_sop` is asserted in the first beat of every packet, when its descriptor is on the bus. When the straddle option is not in use, none of the other sop and eop indications within `m_axi_cq_tuser` are relevant to the transfer of Requests. The byte enable outputs `byte_en[63:0]` (one per byte lane) indicate the valid bytes in the payload. These signals are asserted only when a valid payload byte is in the corresponding lane (it is not asserted for descriptor or null bytes). The asserted byte enable bits are always contiguous from the start of the payload, except when payload size is two Dwords or less. For writes of two Dwords or less, the 1s on `byte_en` are not be contiguous.

Figure 75: Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode)



Another special case is that of a zero-length memory write, when the core transfers a one-Dword payload with the `byte_en` bits all set to 0. Thus, the user logic can, in all cases, use the `byte_en` signals directly to enable the writing of the associated bytes into memory.

In the Dword-aligned mode, there can be a gap of zero, one, two, or three byte positions between the end of the descriptor and the first payload byte, based on the address of the first valid byte of the payload. The actual position of the first valid byte in the payload can be determined either from `first_be[3:0]` or `byte_en[63:0]` in the `m_axis_cq_tuser` bus.

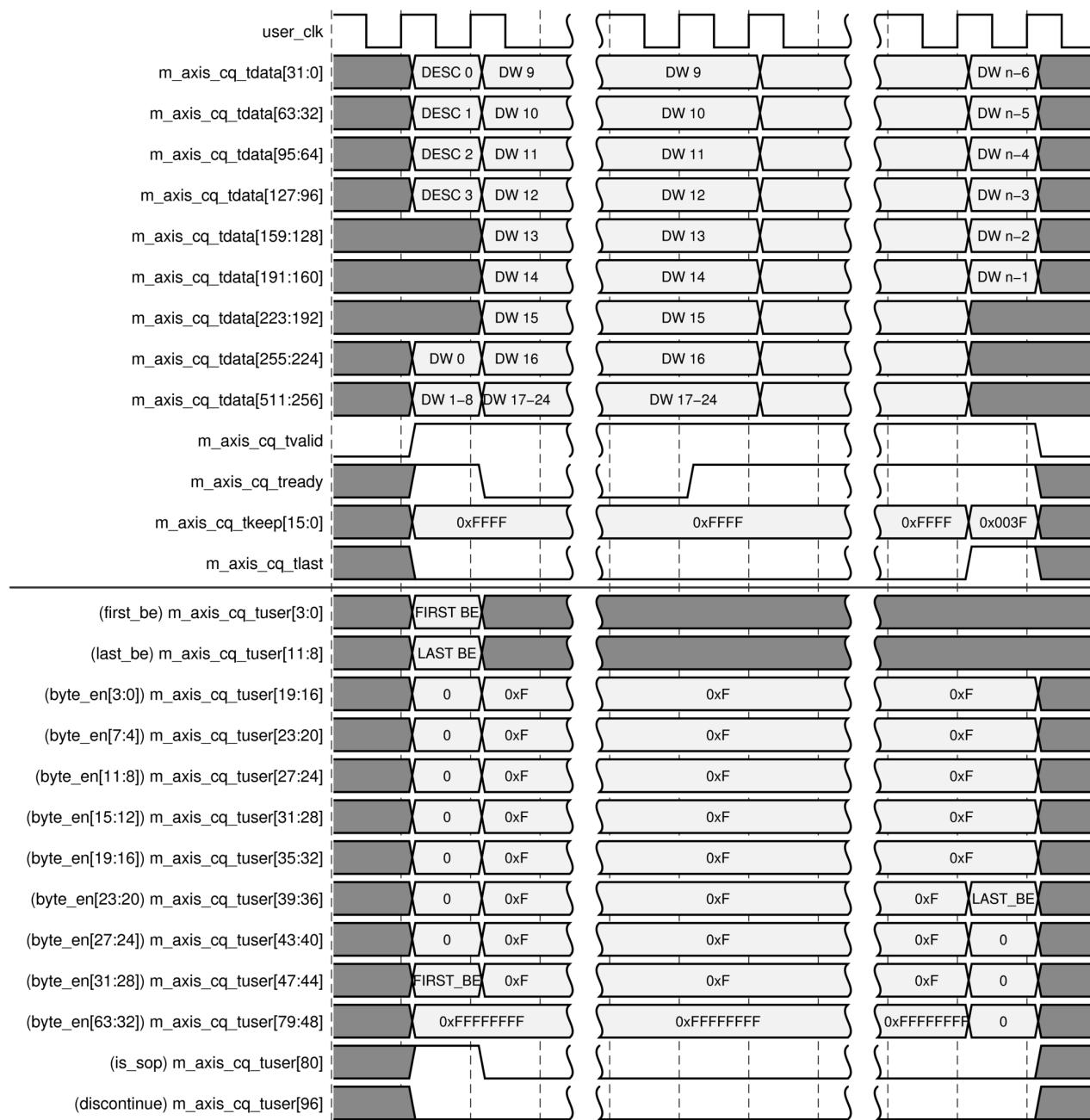
When a Transaction Processing Hint is present in the received TLP, the core transfers the parameters associated with the hint (TPH Steering Tag and Steering Tag Type) on signals within the `m_axis_cq_tuser` bus.

The timing diagram in the following figure illustrates the 128-bit address aligned transfer of a memory write TLP received from the link across the completer request interface. For the purpose of illustration, the starting Dword address of the data block being written into user memory is assumed to be $(m*16 + 3)$, for some integer $m > 0$. Its size is assumed to be n Dwords, for some $n = k*16 - 1$, $k > 1$.

In the address-aligned mode, the delivery of the payload always starts in the second quarter (bits 255:128) of the first beat, following the descriptor in the first quarter. The first Dword the payload can appear on any of the four Dword positions in the second quarter, based on the address of the first valid Dword of the payload. The keep outputs `m_axis_cq_tkeep` remain High in the gap between the descriptor and the payload. The actual position of the first valid byte in the payload can be determined either from the least significant bits of the address in the descriptor or from the byte enable bits `byte_en[63:0]` in the `m_axis_cq_tuser` bus.

For writes of two Dwords or less, the 1s on `byte_en` are not contiguous from the start of the payload. In the case of a zero-length memory write, the core transfers a one-Dword payload with the `byte_en` bits all set to 0 for the payload bytes.

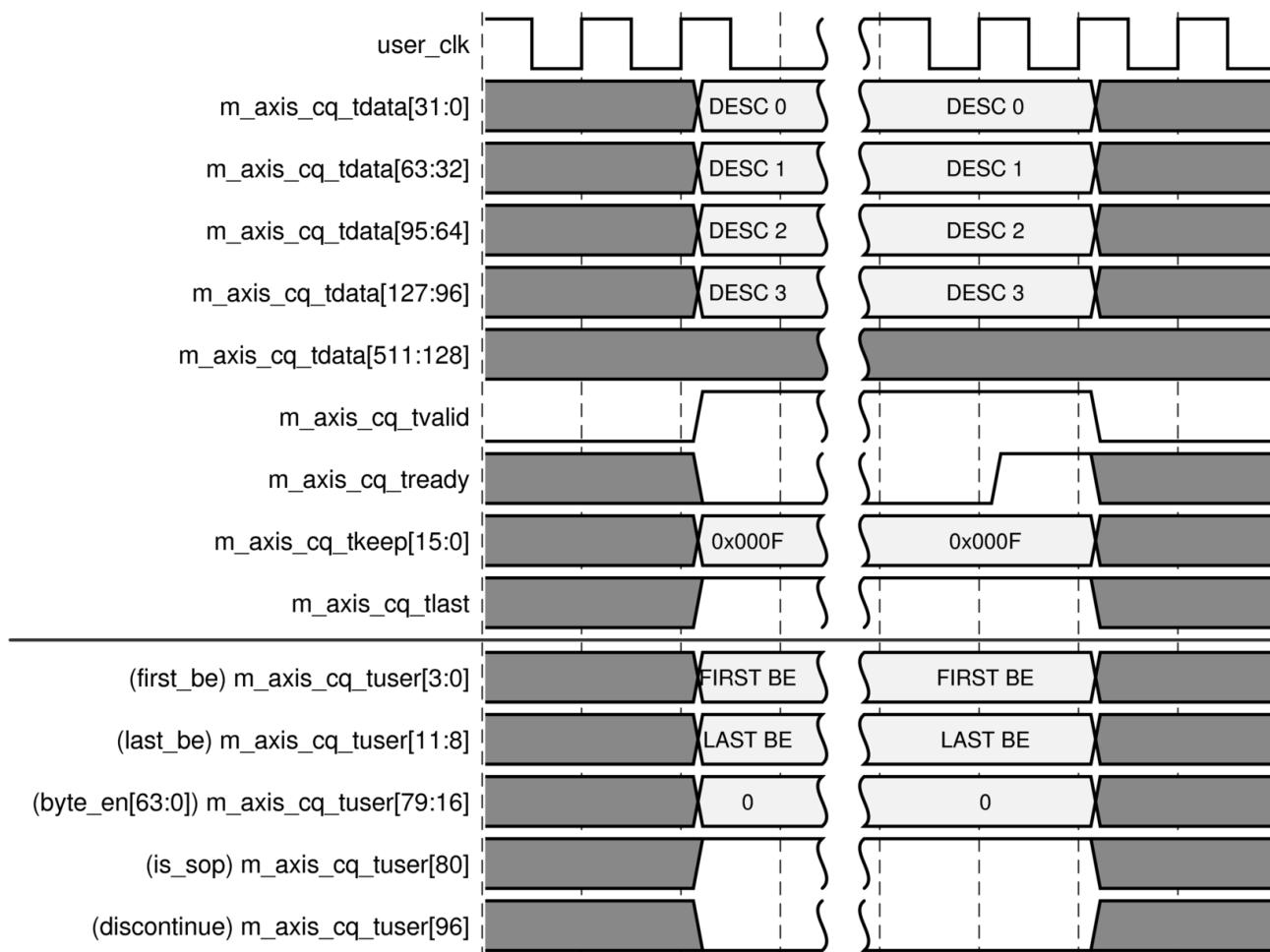
Figure 76: Memory Write Transaction on the Completer Request Interface (128-bit Address Aligned Mode)



Completer Memory Read Operation

A memory read request is transferred across the completer request interface in the same manner as a memory write request, except that the AXI4-Stream packet contains only the 16-byte descriptor. The following figure illustrates the transfer of a memory read TLP received from the link across the completer request interface. The packet is transferred in a single beat on the interface. The signal `m_axis_cq_tvalid` remains asserted over the duration of the packet. The user logic can prolong a beat by pulling down `m_axis_cq_tready`. The signal `is_sop` in the `m_axis_cq_tuser` bus is asserted when the first descriptor byte is on the bus.

Figure 77: Memory Read Transaction on the Completer Request Interface



The byte enable bits associated with the read request for the first and last Dwords are supplied by the core on the sideband bus `m_axis_cq_tuser`. These bits are valid when the descriptor is being transferred, and must be used by the user logic to determine the byte-level starting address and the byte count associated with the request. For the special cases of one-Dword and two-Dword reads, the byte enables can be non-contiguous. The byte enables are contiguous in all other cases. A zero-length memory read is sent on the completer request interface with the Dword count field in the descriptor set to 1 and the first and last byte enables set to 0.

The user logic must respond to each memory read request with a Completion. The data requested by the read are sent as a single Completion or multiple Split Completions. These Completions must be sent to the completer completion interface of the core. The Completions for two distinct requests are sent in any order, but the Split Completions for the same request must be in order. The operation of the completer completion interface is described in [64/128/256-Bit Completer Interface](#) and [512-Bit Completer Interface](#).

I/O Write Operation

The transfer of an I/O write request on the completer request interface is similar to that of a memory write request with a one-Dword payload. The transfer starts with the 128-bit descriptor, followed by the one-Dword payload. When the Dword-aligned mode is in use, the payload Dword immediately follows the descriptor. When the 128-bit address aligned mode is in use, the payload Dword is supplied in bits 255:128, and its alignment is based on the address in the descriptor. The First Byte Enable bits in the `m_axis_cq_tuser` indicate the valid bytes in the payload. The byte enable bits `byte_en` also provide this information.

Because an I/O write is a Non-Posted transaction, the user logic must respond to it with a Completion containing no data payload. The Completions for I/O requests are sent in any order. Errors associated with the I/O write transaction can be signaled to the requester by setting the Completion Status field in the completion descriptor to CA (Completer Abort) or UR (Unsupported Request), as is appropriate. The operation of the completer completion interface is described in [64/128/256-Bit Completer Interface](#) and [512-Bit Completer Interface](#).

I/O Read Operation

The transfer of an I/O read request on the completer request interface is similar to that of a memory read request, and involves only the descriptor. The length of the requested data is always one Dword, and the First Byte Enable bits in `m_axis_cq_tuser` indicate the valid bytes to be read.

The user logic must respond to an I/O read request with a one-Dword Completion (or a Completion with no data in the case of an error). The Completions for two distinct I/O read requests are sent in any order. Errors associated with an I/O read transaction can be signaled to the requester by setting the Completion Status field in the completion descriptor to CA (Completer Abort) or UR (Unsupported Request), as is appropriate. The operation of the completer completion interface is described in [64/128/256-Bit Completer Interface](#) and [512-Bit Completer Interface](#).

Atomic Operations on the Completer Request Interface

The transfer of an Atomic Op request on the completer request interface is similar to that of a memory write request. The payload for an Atomic Op can range from one to eight Dwords, and its starting address is always aligned on a Dword boundary. The transfer starts with the 128-bit descriptor, followed by the payload. When the Dword-aligned mode is in use, the first payload Dword immediately follows the descriptor. When the 128-bit address aligned mode is in use, the payload starts on bits 255:128, and its alignment is based on the address in the descriptor. The keep outputs `m_axis_cq_tkeep` and `m_axis_cq_tuser` also indicate the valid bytes in the payload. The First Byte Enable and Last Byte Enable bits indicate the end of the payload. The `byte_en` signals in `m_axis_cq_tuser` should not be used.

Because an Atomic Operation is a Non-Posted transaction, the user logic must respond to it with a Completion containing the result of the operation. Errors associated with the operation can be signaled to the requester by setting the Completion Status field in the completion descriptor to CA (Completer Abort) or UR (Unsupported Request), as is appropriate. The operation of the completer completion interface is described in [64/128/256-Bit Completer Interface](#) and [512-Bit Completer Interface](#).

Message Requests on the Completer Request Interface

The transfer of a message on the completer request interface is similar to that of a memory write request, except that a payload are not always be present. The transfer starts with the 128-bit descriptor, followed immediately by the payload, if present. The payload always starts in byte lane 16, regardless of the addressing mode in use. The user logic can determine the end of the payload from the states of the signals `m_axis_cq_tlast` and `m_axis_cq_tkeep`. The `byte_en` signals in `m_axis_cq_tuser` also indicate the valid bytes in the payload. The First Byte Enable and Last Byte Enable bits in `m_axis_cq_tuser` should not be used.

The attribute `ATTR_AXISTEN_IF_ENABLE_RX_MSG_INTFC` must be set to 0 to enable the delivery of messages through the completer request interface. When this attribute is set to 0, the attribute `ATTR_AXISTEN_IF_ENABLE_MSG_ROUTE` can be used to select the specific message types that the user wants delivered over the completer request interface. Setting an attribute bit to 1 enables the delivery of the corresponding type of messages on the interface, and setting it to 0 results in the core filtering the message.

Table 58: AXISTEN_IF_ENABLE_MSG_ROUTE Attribute Bit Descriptions

Bit Index	Message Type
0	<code>ERR_COR</code>
1	<code>ERR_NONFATAL</code>
2	<code>ERR_FATAL</code>
3	<code>Assert_INTA</code> and <code>Deassert_INTA</code>
4	<code>Assert_INTB</code> and <code>Deassert_INTB</code>
5	<code>Assert_INTC</code> and <code>Deassert_INTC</code>

Table 58: AXISTEN_IF_ENABLE_MSG_ROUTE Attribute Bit Descriptions (cont'd)

Bit Index	Message Type
6	Assert_INTD and Deassert_INTD
7	PM_PME
8	PME_TO_Ack
9	PME_Turn_Off
10	PM_Active_State_Nak
11	Set_Slot_Power_Limit
12	Latency Tolerance Reporting (LTR)
13	Reserved
14	Unlock
15	Vendor_Defined Type 0
16	Vendor_Defined Type 1
17	Invalid Request, Invalid Completion, Page Request, PRG Response

When ATTR_AXISTEN_IF_ENABLE_RX_MSG_INTFC is set to 1, no messages are delivered on the completer request interface. Indications of received message are instead sent through a dedicated receive message interface (see [Receive Message Interface](#)).

Aborting a Transfer

For any request that includes an associated payload, the interface are signal an error in the transferred payload by asserting the `discontinue` signal in the `m_axis_cq_tuser` bus in the final beat of the packet (along with `m_axis_cq_tlast`). This occurs when the core has detected an uncorrectable error while reading data from its internal memories. The user application must discard the entire packet when it has detected `discontinue` asserted in the final beat of a packet. The interface does not start the transfer of a new packet in the beat in which `discontinue` is asserted, even when the straddle option is enabled.

Selective Flow Control for Non-Posted Requests

The PCI Express® Specifications require that the completer request interface continue to deliver Posted transactions even when the user logic is unable to accept Non-Posted transactions the interface. To enable this capability, the core implements a credit-based flow control mechanism on the completer interface through which user logic can control the flow of Non-Posted requests across the interface, without affecting Posted requests. The user logic signals the availability of buffers to receive Non-Posted requests to the core using the `pcie_cq_np_req[1:0]` signal. The core delivers a Non-Posted request to the user logic only when the available credit is non-zero. The core continues to deliver Posted requests while the delivery of Non-Posted requests has been paused for lack of credit. When no backpressure is applied by the credit mechanism for the delivery of Non-Posted requests, the core delivers Posted and Non-Posted requests in the same order as received from the link.

The core maintains an internal credit counter to track the credit available for Non-Posted requests on the completer request interface. The following algorithm is used to keep track of the available credit:

- On reset, the counter is set to 0.
- After the interface comes out of reset, in every clock cycle:
 - If `pcie_cq_np_req` is non-zero and no Non-Posted request is being delivered this cycle, the credit count is incremented by 1, unless it has already reached its saturation limit of 32. The increment amount is 1 when `pcie_cq_np_req` = 2'b01 and 2 when `pcie_cq_np_req` = 2'b10 or 2'b11.
 - If `pcie_cq_np_req` = 2'b00 and a single Non-Posted request is being delivered this cycle, the credit count is decremented by 1, unless it is already 0.
 - If `pcie_cq_np_req` = 2'b00 and two Non-Posted requests are being delivered this cycle, the credit count is decremented twice, unless it has already reached 0.
 - Otherwise, the credit count remains unchanged.
- The core starts delivery of a Non-Posted TLP to the user logic only if the credit count is greater than 0.

The user application can either provide one or two credits on `pcie_cq_np_req` each time it is ready to receive Non-Posted requests, or can keep it permanently set to 2'b11 if it does not need to exercise selective backpressure on Non-Posted requests. If the credit count is always non-zero, the core delivers Posted and Non-Posted requests in the same order as received from the link. If it remains 0 for some time, Non-Posted requests can accumulate in the core's FIFO. When the credit count becomes non-zero later, the core first delivers the accumulated Non-Posted requests that arrived before Posted requests already delivered to the user application, and then reverts to delivering the requests in the order received from the link.

The setting of `pcie_cq_np_req` does not need to be aligned with the packet transfers on the completer request interface.

The user application can monitor the current value of the credit count on the output `pcie_cq_np_req_count[5:0]`. The counter saturates at 32. Because of internal pipeline delays, there can be several cycles of delay between the core receiving a pulse on the `pcie_cq_np_req` input and updating the `pcie_cq_np_req_count` output in response. Thus, when the user logic has adequate buffer space available, it should provide the credit in advance so that Non-Posted requests are not held up by the core for lack of credit.

Straddle Option on CQ Interface

The core has the capability to start the transfer of a new request on the requester completion interface in the same beat when the previous request has ended on or before Dword position 7 on the data bus. This straddle option is enabled during core customization in the Vivado® IDE. The straddle option can be used only with the Dword-aligned mode.

When the straddle option is enabled, request TLPs are transferred on the AXI4-Stream interface as a continuous stream, with no packet boundaries. Thus, the signals `m_axis_rc_tkeep` and `m_axis_rc_tlast` are not useful in determining the boundaries of TLPs delivered on the interface (the core sets `m_axis_rc_tkeep` to all 1s and `m_axis_rc_tlast` to 0 permanently when the straddle option is in use.). Instead, delineation of TLPs is performed using the following signals provided within the `m_axis_rc_tuser` bus.

- `is_sop[0]`: The core sets this output to active-High in a beat when there is at least one request TLP starting in the beat. The position of the first byte of the descriptor of this TLP is determined as follows:
 - If the previous TLP ended before this beat, the first byte of the descriptor is in byte lane 0.
 - If a previous TLP is continuing in this beat, the first byte of this descriptor is in byte lane 32. This is possible only when the previous TLP ends in the current beat, that is when `is_eop[0]` is also set.
- `is_sop[1]`: The core asserts this output in a beat when there are two request TLPs starting in the same beat. The first TLP always starts at byte position 0 and the second TLP at byte position 32. The core starts a second TLP at byte position 32 only if the previous TLP ended before byte position 32 in the same beat, that is only if `is_eop[0]` is also set in the same beat.
- `is_eop[0]`: This output is used to indicate the end of a request TLP. Its assertion signals that there is at least one TLP ending in this beat.
- `is_eop0_ptr[3:0]`: When `is_eop[0]` is asserted, `is_eop0_ptr[3:0]` provides the offset of the last Dword of the corresponding TLP ending in this beat. For TLPs with a payload, the offset for the last byte can be also be determined from the starting address and length of the TLP, or from the byte enable signals `byte_en[63:0]`.
- `is_eop[1]`: This output is used to indicate that there are two TLPs ending in a beat. Its assertion signals that there is at least one TLP ending in this beat. `is_eop[1]` can be set only when `is_eop[0]` is also set.
- `is_eop1_ptr[3:0]`: When `is_eop[1]` is asserted, `is_eop1_ptr[3:0]` provides the offset of the last Dword of the second TLP ending in this beat. For TLPs with a payload, the offset for the last byte can be also be determined from the starting address and length of the TLP, or from the byte enable signals `byte_en[63:0]`. Because the second TLP can start only on byte lane 32, it can only end at a byte lane in the range 47-63. Thus the offset `is_eop1_ptr[3:0]` can only take a value in the range 11-15.

Figure 78: Transfer of Request TLPs on the Completer Request Interface with the Straddle Option Enabled

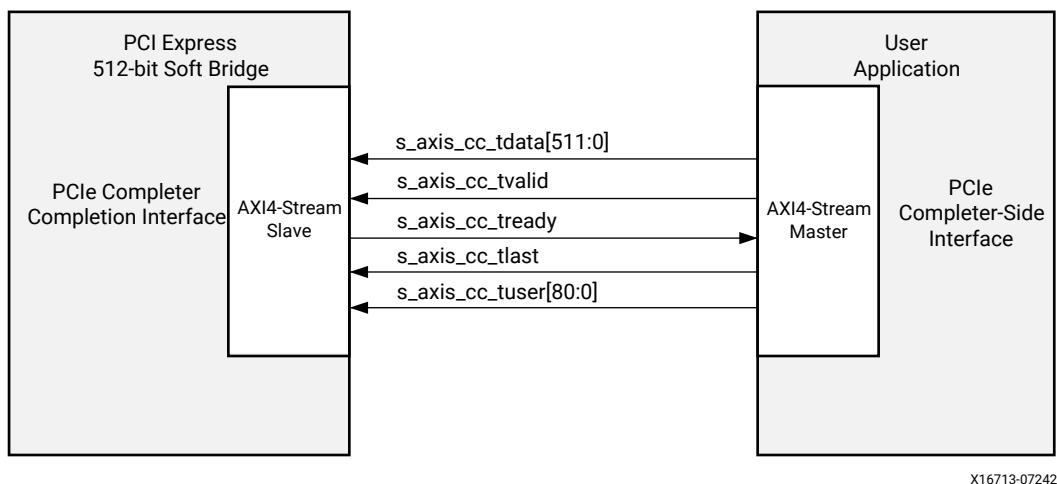


The previous figure illustrates the transfer of four request TLPs on the completer request interface when the straddle option is enabled. For all TLPs, the first Dword of the payload always follows the descriptor without any gaps. The first request TLP (REQ 1) starts at Dword position 0 of Beat 1 and ends in Dword position 5 of Beat 3. The second TLP (REQ 2) starts in Dword position 8 of the same beat. This second TLP has only a four-Dword payload, so it also ends in the same beat. The third and fourth request TLPs are transferred completely in Beat 4, as REQ 3 has only a one-Dword payload and REQ 4 has no payload.

Completer Completion Interface Operation (512-bits)

The following figure illustrates the signals associated with the completer completion interface of the core. The core delivers each TLP on this interface as an AXI4-Stream packet. The packet starts with a 96-bit descriptor, followed by data in the case of Completions with a payload.

Figure 79: Completer Completion Interface Signals



X16713-072420

The completer request interface supports two distinct data alignment modes, selected during core customization in the Vivado® IDE. In the Dword-aligned mode, the first byte of valid data must be presented on lane $n = (S + 12 + (A \bmod 4)) \bmod 64$, where A is the byte-level starting address of the data block being transferred and S is the lane number where the first byte of the descriptor appears. The address A is taken as the value in the Lower Address field of the descriptor. The starting lane number S is always 0 when the straddle option is not used, but can be 0 or 32 when straddle is enabled.

In the 128-bit address-aligned mode, the lane number corresponding to the first byte of the payload is determined as $n = (S + 16 + (A \bmod 16)) \bmod 64$, where S is the lane number where the first byte of the descriptor appears (which can be 0 or 32) and A is the address corresponding to the first byte of the payload. Any gap between the end of the descriptor and the start of the first byte of the payload is filled with null bytes.

The interface also supports a straddle option that allows the transfer of up to two TLPs in the same beat across the interface. The straddle option can be used only with the Dword-aligned mode, and is not supported when using the 128-bit address aligned mode. The descriptions in the sections below assume a single TLP per beat. The operation of the interface with the straddle option enabled is described in [Straddle Option on CC Interface](#).

Completer Completion Descriptor Format

The user application sends completion data for a completer request to the completer completion interface of the core as an independent AXI4-Stream packet. Each packet starts with a descriptor, and can have payload data following the descriptor. The descriptor is always 12 bytes long, and is sent in the first 12 bytes of the completion packet. The descriptor is always transferred in the first beat of a Completion TLP. When the user application splits the completion data for a request into multiple Split Completions, it must send each Split Completion as a separate AXI4-Stream packet, with its own descriptor.

The format of the completer completion descriptor is illustrated in the following figure. The individual fields of the completer request descriptor are described in the following table.

Figure 80: Completer Completion Descriptor Format

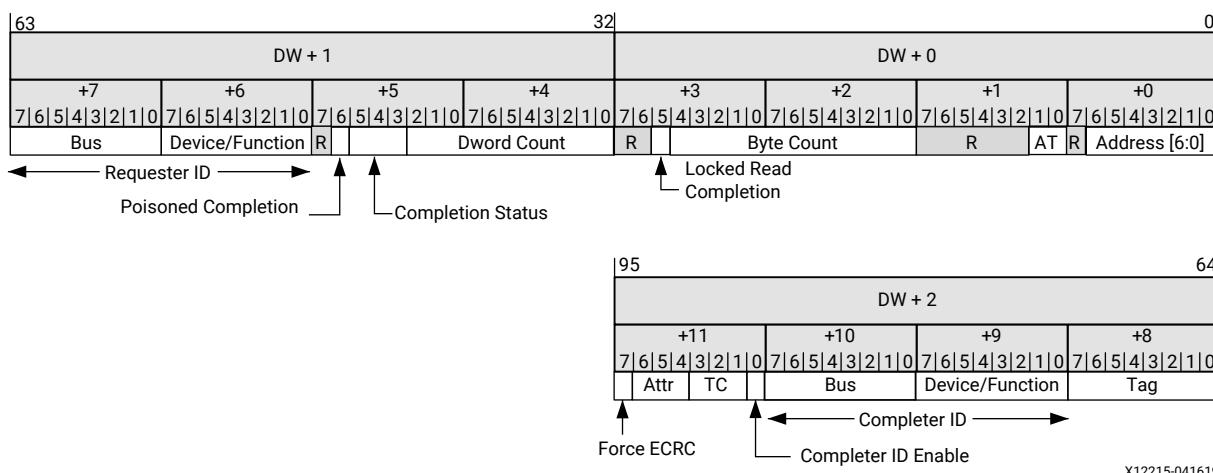


Table 59: Completer Completion Descriptor Fields

Bit Index	Field Name	Description
6:0	Lower Address	For memory read Completions, this field must be set to the least significant 7 bits of the starting byte-level address of the memory block being transferred. For all other Completions, the Lower Address must be set to all zeroes.
9:8	Address Type	This field is defined for Completions of memory transactions and Atomic Operations only. For these Completions, the user logic must copy the AT bits from the corresponding request descriptor into this field. This field must be set to 0 for all other Completions.

Table 59: Completer Completion Descriptor Fields (cont'd)

Bit Index	Field Name	Description
28:16	Byte Count	<p>These 13 bits can have values in the range of 0 – 4,096 bytes. If a Memory Read Request is completed using a single Completion, the Byte Count value indicates Payload size in bytes. This field must be set to 4 for I/O read Completions and I/O write Completions. The byte count must be set to 1 while sending a Completion for a zero-length memory read, and a dummy payload of 1 Dword must follow the descriptor.</p> <p>For each Memory Read Completion, the Byte Count field must indicate the remaining number of bytes required to complete the Request, including the number of bytes returned with the Completion. If a Memory Read Request is completed using multiple Completions, the Byte Count value for each successive Completion is the value indicated by the preceding Completion minus the number of bytes returned with the preceding Completion</p>
29	Locked Read Completion	This bit must be set when the Completion is in response to a Locked Read request. It must be set to 0 for all other Completions.
42:32	Dword Count	<p>These 11 bits indicate the size of the payload of the current packet in Dwords. Its range is 0 – 1K Dwolds. This field must be set to 1 for I/O read Completions and 0 for I/O write Completions. The Dword count must be set to 1 while sending a Completion for a zero-length memory read. The Dword count must be set to 0 when sending a UR or CA Completion. In all other cases, the Dword count must correspond to the actual number of Dwolds in the payload of the current packet.</p>
45:43	Completion Status	<p>These bits must be set based on the type of Completion being sent. The only valid settings are:</p> <ul style="list-style-type: none"> • 000: Successful Completion • 001: Unsupported Request (UR) • 100: Completer Abort (CA)
46	Poisoned Completion	<p>This bit can be used by the user logic to poison the Completion TLP being sent. This bit must be set to 0 for all Completions, except when the user logic has detected an error in the block of data following the descriptor and wants to communicate this information using the Data Poisoning feature of PCI Express.</p>
63:48	Requester ID	PCI Requester ID associated with the request (copied by the user logic from the request).
71:64	Tag	PCIe Tag associated with the request (copied by the user logic from the request).

Table 59: Completer Completion Descriptor Fields (cont'd)

Bit Index	Field Name	Description
79:72	Target Function/Device Number	<p>Device and/or Function number of the Completer Function.</p> <p>Endpoint mode:</p> <p>ARI enabled:</p> <ul style="list-style-type: none"> Bits [79:72] must be set to the Completer Function number. <p>ARI disabled:</p> <ul style="list-style-type: none"> Bits [74:72] must be set to the Completer Function number. Bits [79:75] are not used <p>Upstream Port for Switch use case (Endpoint mode is selected within the IP):</p> <p>ARI enabled:</p> <ul style="list-style-type: none"> Bits [79:72] must be set to the Completer Function number. <p>ARI disabled:</p> <ul style="list-style-type: none"> Bits [74:72] must be set to the Completer Function number. Bits [79:75] are not used if the Completion is originating from the switch itself. These bits must be set to the Completer Device number where the Completion was originated if the switch is relaying the Completion (Completer is external to the switch). This is used with Completer ID Enable bit in the descriptor. <p>Root Port mode (Downstream Port):</p> <p>ARI enabled:</p> <ul style="list-style-type: none"> Bits [79:72] must be set to the Completer Function number. <p>ARI disabled:</p> <ul style="list-style-type: none"> Bits [74:72] must be set to the Completer Function number. Bits [79:75] must be set to the Completer Device number. This is used with Completer ID Enable bit in the descriptor.

Table 59: Completer Completion Descriptor Fields (cont'd)

Bit Index	Field Name	Description
87:80	Completer Bus Number	<p>Device and/or Function number of the Requester Function.</p> <p>Endpoint mode:</p> <p>ARI enabled:</p> <ul style="list-style-type: none"> Bits [87:80] must be set to the Requester Function number. <p>ARI disabled:</p> <ul style="list-style-type: none"> Bits [82:80] must be set to the Requester Function number. Bits [87:83] are not used <p>Upstream Port for Switch use case (Endpoint mode is selected within the IP):</p> <p>ARI enabled:</p> <ul style="list-style-type: none"> Bits [87:80] must be set to the Requester Function number. <p>ARI disabled:</p> <ul style="list-style-type: none"> Bits [82:80] must be set to the Requester Function number. Bits [87:83] are not used if the request is originating from the switch itself. These bits must be set to the Requester Device number where the request was originated if the switch is relaying the request (Requester is external to the switch). This is used with Requester ID Enable bit in the descriptor. <p>Root Port mode (Downstream Port):</p> <p>ARI enabled:</p> <ul style="list-style-type: none"> Bits [87:80] must be set to the Requester Function number. <p>ARI disabled:</p> <ul style="list-style-type: none"> Bits [87:80] must be set to the Requester Function number. Bits [87:83] must be set to the Requester Device number. This is used with Requester ID Enable bit in the descriptor.

Table 59: Completer Completion Descriptor Fields (cont'd)

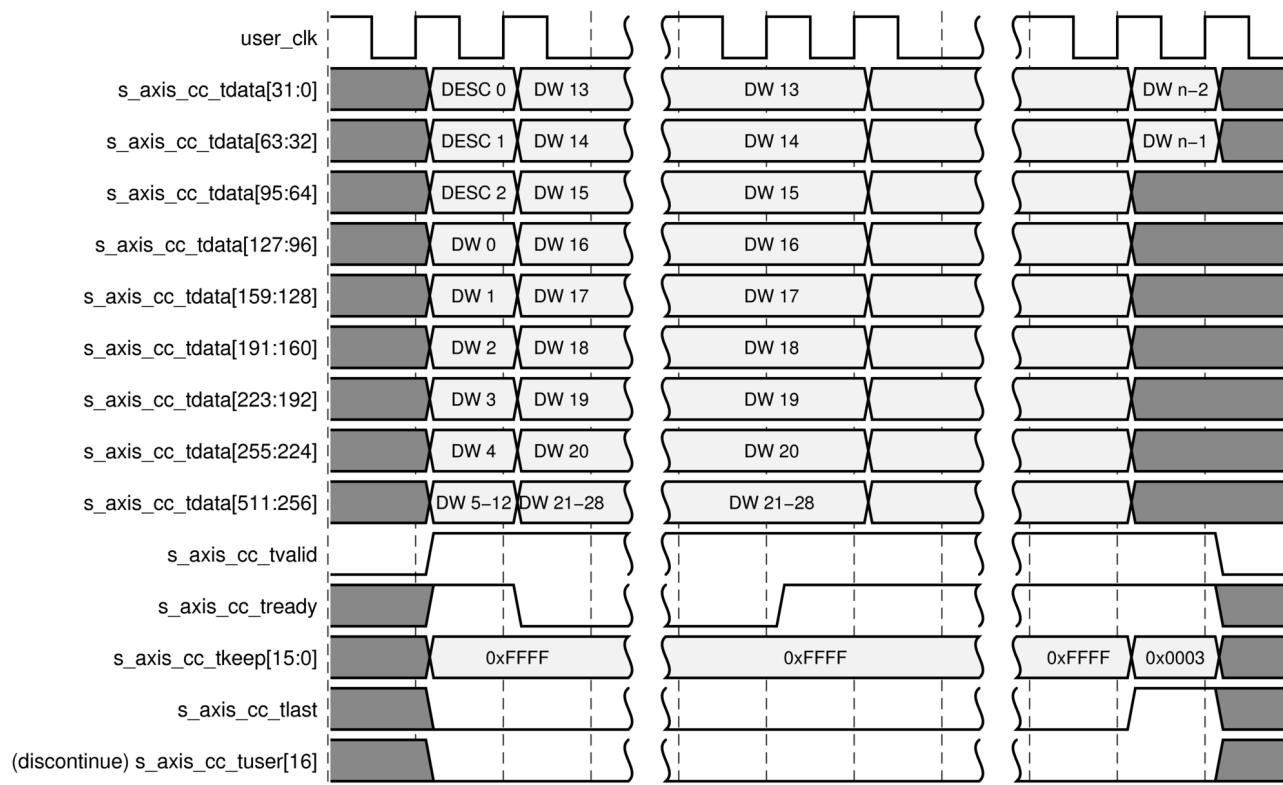
Bit Index	Field Name	Description
88	Completer ID Enable	<p>1'b1: The client supplies Bus, Device, and Function numbers in the descriptor to be populated as the Completer ID field in the TLP header.</p> <p>1'b0: IP uses Bus and Device numbers captured from received Configuration requests and the client supplies Function numbers in the descriptor to be populated as the Completer ID field in the TLP header.</p> <p>Endpoint mode:</p> <ul style="list-style-type: none"> Must be set to 1'b0. <p>Upstream Port for Switch use case (Endpoint mode is selected within the IP):</p> <ul style="list-style-type: none"> Set to 1'b0 when the Completion is originating from the switch itself. Set to 1'b1 when the switch is relaying the Completion (Completer is external to the switch). This is used with Completer Bus Number bits [95:88] and Completer Function/Device Number bits [87:83] when ARI is not enabled. <p>Root Port mode:</p> <ul style="list-style-type: none"> Must be set to 1'b1. This is used with Completer Bus Number bits [95:88] and Completer Function/Device Number bits [87:83] when ARI is not enabled.
91:89	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request. The user logic must copy this value from the TC field of the associated request descriptor.
94:92	Attributes	PCIe attributes associated with the request (copied from the request). Bit 92 is the No Snoop bit, bit 93 is the Relaxed Ordering bit, and bit 94 is the ID-Based Ordering bit.
95	Reserved	Reserved for future use.

Completions with Successful Completion (SC) Status

The user logic must return a Completion to the completer completion interface of the core for every Non-Posted request it receives from the completer request interface. When the request completes with no errors, the user logic must return a Completion with Successful Completion (SC) status. Such a Completion might contain a payload, depending on the type of request. Furthermore, the data associated with the request can be broken up into multiple Split Completions when the size of the data block exceeds the maximum payload size configured. User logic is responsible for splitting the data block into multiple Split Completions when needed. The user logic must transfer each Split Completion over the completer completion interface as a separate AXI4-Stream packet, with its own 12-byte descriptor.

In the following example timing diagrams, the starting Dword address of the data block being transferred (as conveyed in bits [6:2] of the Lower Address field of the descriptor) is assumed to be $(m*8+1)$, for some integer m . The size of the data block is assumed to be n Dwords, for some $n = k*32+28$, $k > 0$.

Figure 81: Transfer of a Normal Completion on the Completer Completion Interface (512-bit Interface, Dword-Aligned Mode)



The previous figure illustrates the Dword-aligned transfer of a Completion from the user logic across the completer completion interface. In this case, the first Dword of the payload starts immediately after the descriptor. When the data block is not a multiple of 4 bytes, or when the start of the payload is not aligned on a Dword address boundary, the user application must add null bytes to align the start of the payload on a Dword boundary and make the payload a multiple of Dwords. For example, when the data block starts at byte address 7 and has a size of 3 bytes, the user logic must add 3 null bytes before the first byte and two null bytes at the end of the block to make it 2 Dwords long. Also, in the case of non-contiguous reads, not all bytes in the data block returned are valid. In that case, the user application must return the valid bytes in the proper positions, with null bytes added in gaps between valid bytes, when needed. The interface does not have any signals to indicate the valid bytes in the payload. This is not required, as the requester is responsible for keeping track of the byte enables in the request and discarding invalid bytes from the Completion.

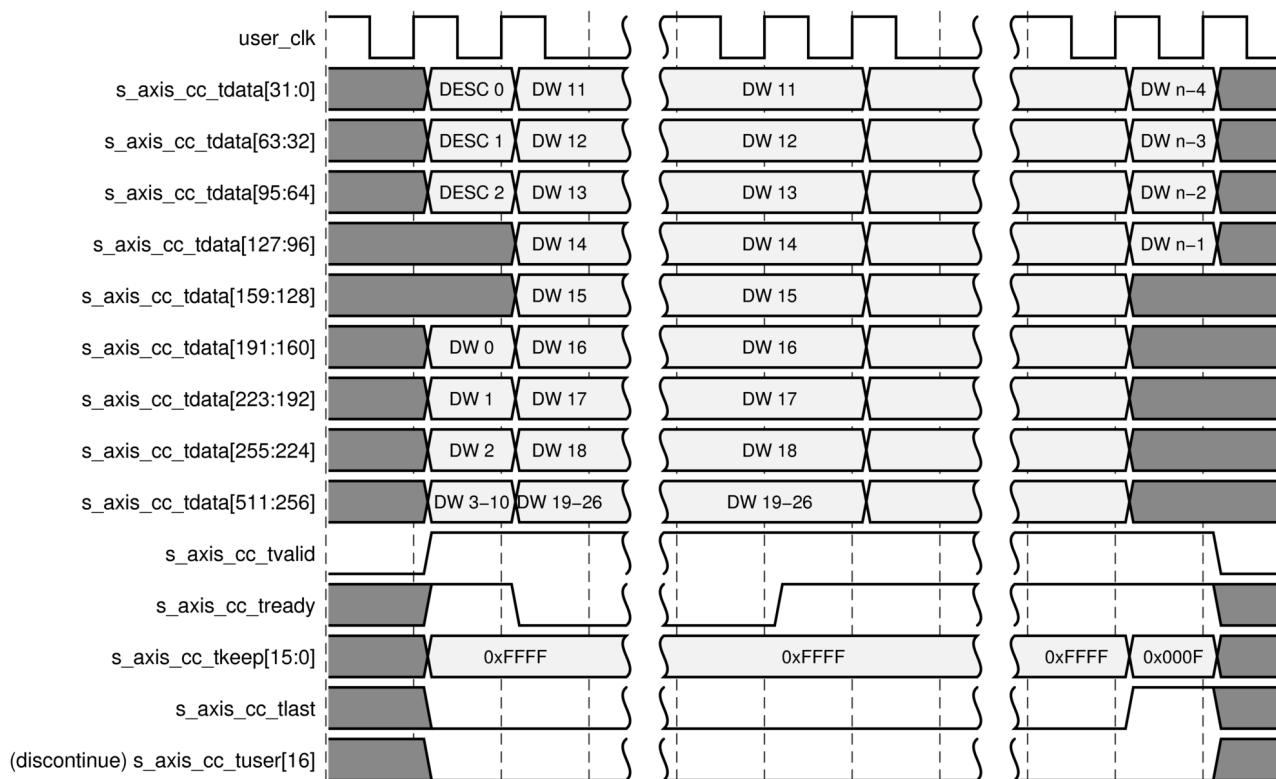
In the Dword-aligned mode, the transfer starts with the 12 descriptor bytes, followed immediately by the payload bytes. The user application must keep the signal `s_axis_cc_tvalid` asserted over the duration of the packet. The core treats the deassertion of `s_axis_cc_tvalid` during the packet transfer as an error, and nullifies the corresponding Completion TLP transmitted on the link to avoid data corruption.

The user application must also assert the signal `s_axis_cc_tlast` in the last beat of the packet. The core are by pull down `s_axis_cc_tready` in any cycle if it is not ready to accept data. The user application must not change the values on `s_axis_cc_tdata` and `s_axis_cc_tlast` during the transfer when the core has deasserted `s_axis_cc_tready`.

In the 128-bit address aligned mode, the delivery of the payload must always start in the second 128-bit quarter of the 512-bit word, following the descriptor in the first quarter. That is, if the first byte of the descriptor is on byte lane 0, the payload must start on one of the byte lanes 16 – 31. Within its 128-bit quarter, the offset of the first payload byte must correspond to the least significant bits of the Lower Address field setting in the corresponding descriptor.

The following timing diagram illustrates the 128-bit address-aligned transfer of a memory read Completion across the completer completion interface. For the purpose of illustration, the starting Dword address of the data block being transferred (as conveyed in bits [6:2] of the Lower Address field of the descriptor) is assumed to be $(m*16+1)$, for some integer m . The size of the data block is assumed to be n Dwords, for some $n = k*16 - 1$, for some $k > 1$.

Figure 82: Transfer of a Normal Completion on the Completer Completion Interface (128-bit Address Aligned Mode)



Aborting a Completion Transfer

The user logic can abort the transfer of a Completion on the completer completion interface at any time during the transfer of the payload by asserting the `discontinue` signal in the `s_axis_cc_tuser` bus. The core nullifies the corresponding TLP on the link to avoid data corruption.

The user logic can assert this signal in any cycle during the transfer, when the Completion being transferred has an associated payload. The user logic can either choose to terminate the packet prematurely in the cycle where the error was signaled (by asserting `s_axis_cc_tlast`), or can continue until all bytes of the payload are delivered to the core. In the latter case, the core treats the error as sticky for the following beats of the packet, even if the user logic deasserts the `discontinue` signal before reaching the end of the packet.

The `discontinue` signal can be asserted only when `s_axis_cc_tvalid` is active-High. The core samples this signal when `s_axis_cc_tvalid` and `s_axis_cc_tready` are both active-High. Thus, once asserted, it should not be deasserted until `s_axis_cc_tready` is active-High.

When the core is configured as an Endpoint, this error is reported by the core to the Root Complex it is attached to, as an Uncorrectable Internal Error using the Advanced Error Reporting (AER) mechanisms.

Completions with Error Status (UR and CA)

When responding to a request received on the completer request interface with an Unsupported Request (UR) or Completion Abort (CA) status, the user logic must send a 3-Dword completion descriptor in the format of the *Completer Completion Descriptor Format* figure in [Completer Completion Descriptor Format](#), followed by five additional Dwords containing information on the request that generated the Completion. These five Dwords are necessary for the core to log information about the request in its AER header log registers.

The following figure shows the sequence of information transferred when sending a Completion with UR or SC status. The information is formatted as an AXI4-Stream packet with a total of 8 Dwords, which are organized as follows:

- The first three Dwords contain the completion descriptor in the format of the *Completer Completion Descriptor Format* figure in [Completer Completion Descriptor Format](#).
- The fourth Dword contains the state of the following signals in `m_axis_cq_tuser`, copied from the request:
 - The First Byte Enable bits `first_be[3:0]` in `m_axis_cq_tuser`.
 - The Last Byte Enable bits `last_be[3:0]` in `m_axis_cq_tuser`.
 - Signals carrying information on Transaction Processing Hint: `tph_present`, `tph_type[1:0]` and `tph_st_tag[7:0]` in `m_axis_cq_tuser`.
 - The four Dwords of the request descriptor received from the core with the request.

Figure 83: Composition of the AXI4-Stream Packet for UR and CA Completions



X12245

Straddle Option on CC Interface

The core has the capability to start the transfer of a new Completion packet on the completer completion interface in the same beat when the previous request has ended on or before Dword position 7 on the data bus. This straddle option is enabled during core customization in the Vivado® IDE. The straddle option can be used only with the Dword-aligned mode.

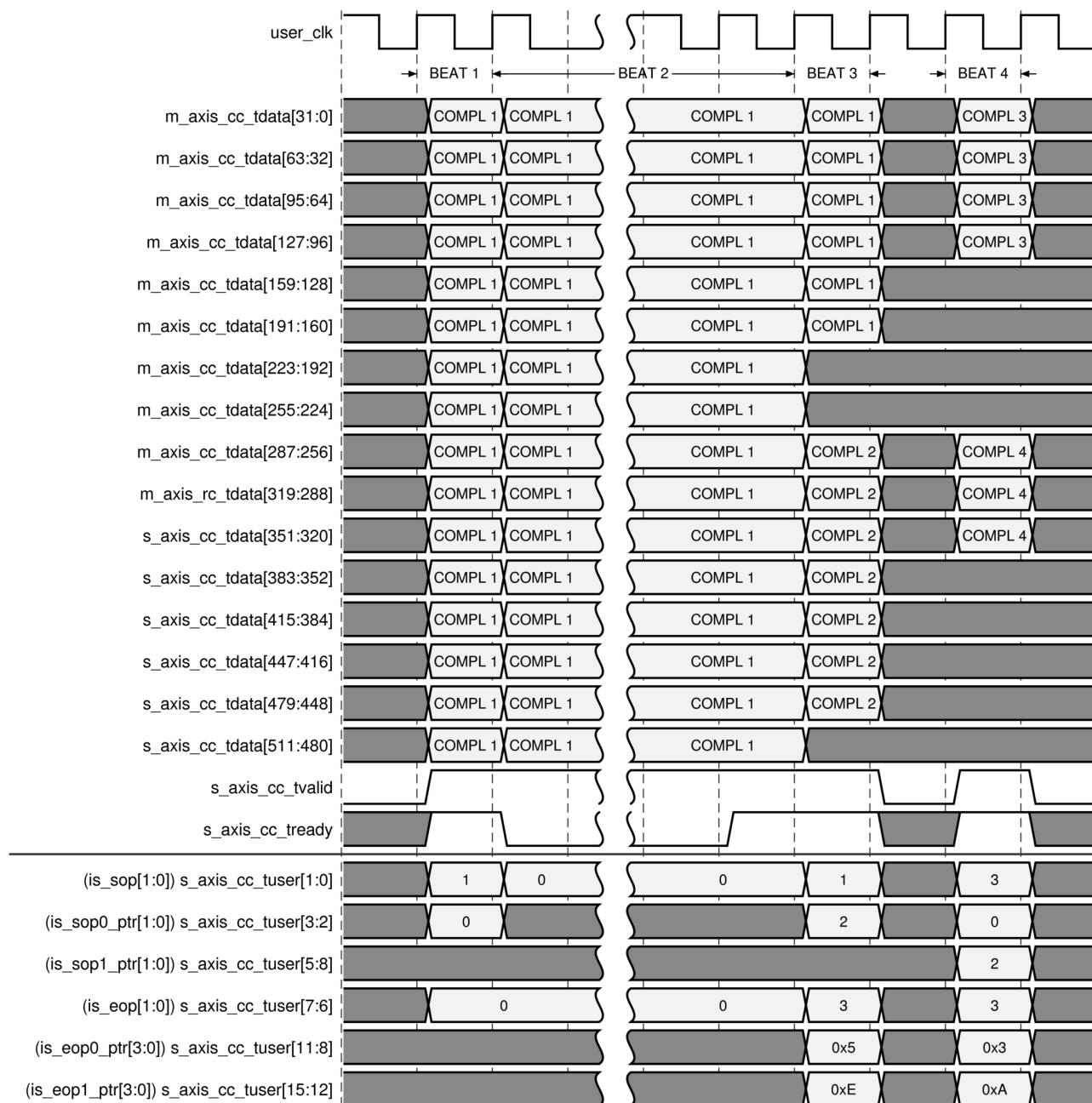
When the straddle option is enabled, Completion TLPs are transferred on the AXI4-Stream interface as a continuous stream, with no packet boundaries. Thus, the signals `m_axis_cc_tkeep` and `m_axis_cc_tlast` are not useful in determining the boundaries of TLPs delivered on the interface. Instead, delineation of TLPs is performed using the following signals provided within the `m_axis_cc_tuser` bus.

- `is_sop[0]`: This input must be set High in a beat when there is at least one Completion TLP starting in the beat. The position of the first byte of the descriptor of this TLP is determined as follows:
 - If the previous TLP ended before this beat, the first byte of the descriptor is in byte lane 0.
 - If a previous TLP is continuing in this beat, the first byte of this descriptor is in byte lane 32. This is possible only when the previous TLP ends in the current beat, that is when `is_eop[0]` is also set.
- `is_sop0_ptr[1:0]`: When `is_sop[0]` is set, this field must indicate the offset of the first Completion TLP starting in the current beat. Valid settings are `2'b00` (TLP starting at Dword 0) and `2'b10` (TLP starting at Dword 8).

- `is_sop[1]`: This input must be set High in a beat when there are two Completion TLPs starting in the same beat. The first TLP must always start at byte position 0 and the second TLP at byte position 32. The user application can start a second TLP at byte position 32 only if the previous TLP ended before byte position 32 in the same beat, that is only if `is_eop[0]` is also set in the same beat.
- `is_sop1_ptr[1:0]`: When `is_sop[1]` is set, this field must provide the offset of the second TLP starting in the current beat. Its only valid setting is `2'b10` (TLP starting at Dword 8).
- `is_eop[0]`: This input is used to indicate the end of a Completion TLP. Its assertion signals that there is at least one TLP ending in this beat.
- `is_eop0_ptr[3:0]`: When `is_eop[0]` is asserted, `is_eop0_ptr[3:0]` must provide the offset of the last Dword of the corresponding TLP ending in this beat.
- `is_eop[1]`: This input is set High when there are two TLPs ending in the current beat. `is_eop[1]` can be set only when the signals `is_eop[0]` and `is_sop[0]` are also be High in the same beat.
- `is_eop1_ptr[3:0]`: When `is_eop[1]` is asserted, `is_eop1_ptr[3:0]` must provide the offset of the last Dword of the second TLP ending in this beat. Because the second TLP can start only on byte lane 32, it can only end at a byte lane in the range 43-63. Thus the offset `is_eop1_ptr[3:0]` can only take a value in the range 10-15.

The following figure illustrates the transfer of four Completion TLPs on the completer completion interface when the straddle option is enabled. For all TLPs, the first Dword of the payload always follows the descriptor without any gaps. The first Completion TLP (COMPL 1) starts at Dword position 0 of Beat 1 and ends in Dword position 5 of Beat 3. The second TLP (COMPL 2) starts in Dword position 8 of the same beat. This second TLP has only a four-Dword payload, so it also ends in the same beat. The third and fourth Completion TLPs are transferred completely in Beat 4, as COMPL 3 has only a one-Dword payload and COMPL 4 has no payload.

Figure 84: Transfer of Completion TLPs on the Completer Completion Interface with the Straddle Option Enabled (512-bit Interface)



512-Bit Requester Interface

This section describes the operation of the user-side Requester interface associated with the 512-bit AXI4-Stream Interface. The block diagram in [512-Bit Completer Interface](#) illustrates the connections between the soft bridge, PCIe® core and user application. The soft bridge converts the 256-bit packets at 500 MHz into 512-bit packets at 250 MHz.

The Requester interface enables a user Endpoint application to initiate PCI transactions as a bus master across the PCIe link to the host memory. For Root Complexes, this interface is also used to initiate I/O and configuration requests. This interface can also be used by both Endpoints and Root Complexes to send messages on the PCIe link. The transactions on this interface are similar to those on the completer interface, except that the roles of the core and the user application are reversed. Posted transactions are performed as single indivisible operations and Non-Posted transactions as split transactions.

The requester interface consists of two separate interfaces, one for data transfer in each direction. Each interface is based on the AXI4-Stream protocol, and its width can be configured as 64, 128 or 256 bits. The requester request interface is used for transfer of requests (with any associated payload data) from the user application to the core, and the requester completion interface is used by the core to deliver Completions received from the link (for Non-Posted requests) to the user application. The two interfaces operate independently, that is, the user application can transfer new requests over the requester request interface while receiving a completion for a previous request.

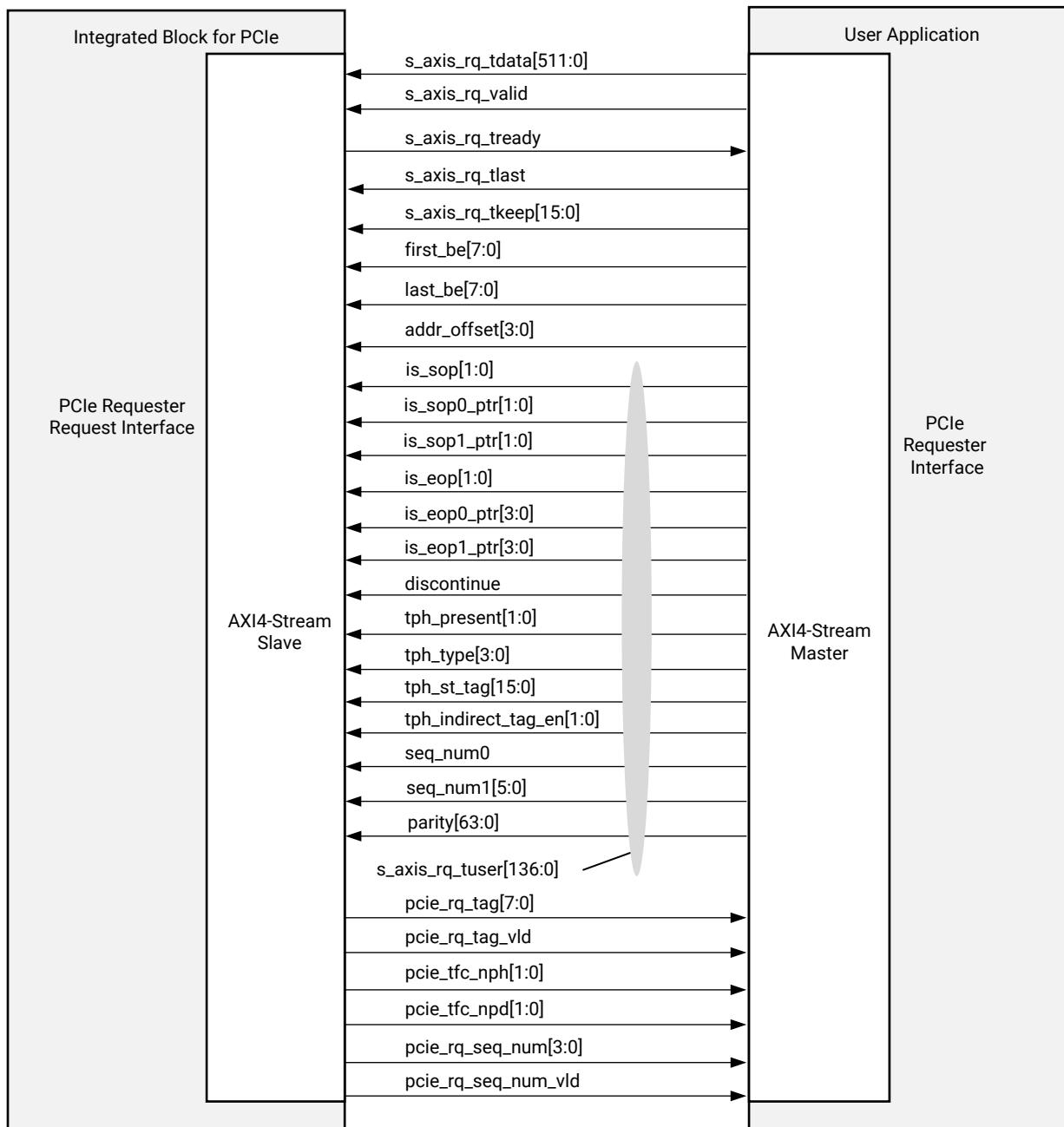
Requester Request Interface Operation (512-bits)

The following figure illustrates the signals associated with the requester request interface of the core. The core delivers each TLP on this interface as an AXI4-Stream packet. The packet starts with a 128-bit descriptor, followed by data in the case of TLPs with a payload.

The requester request interface supports two distinct data alignment modes for transferring payloads, which are set during core customization in the Vivado® IDE. In the Dword-aligned mode, the user logic must provide the first Dword of the payload immediately after the last Dword of the descriptor. It must also set the bits in `first_be[7:0]` to indicate the valid bytes in the first Dword and the bits in `last_be[7:0]` (both part of the `s_axis_rq_tuser` bus) to indicate the valid bytes in the last Dword of the payload. In the address-aligned mode, the user logic must start the payload transfer in the beat following the last Dword of the descriptor, and its first Dword can be in any of the possible Dword positions on the data path. The user application communicates the offset of the first Dword on the data path using the signals `addr_offset[3:0]` in `s_axis_rq_tuser`. As in the case of the Dword-aligned mode, the user application must also set the bits in `first_be[7:0]` to indicate the valid bytes in the first Dword and the bits in `last_be[7:0]` to indicate the valid bytes in the last Dword of the payload. In Straddled case, `addr_offset[3:2]`, `first_be[7:4]`, and `last_be[7:4]` are used to indicate second TLP information while `addr_offset[1:0]`, `first_be[3:0]`, and `last_be[3:0]` are used to indicate the first TLP information on that data beat.

When the Transaction Processing Hint Capability is enabled in the core, the user logic can provide an optional hint with any memory transaction using the `tph_*` signals included in the `s_axis_rq_tuser` bus. To supply a Hint with a request, the user logic must assert `tph_present` in the first beat of the packet, and provide the TPH Steering Tag and Steering Tag Type on `tph_st_tag[7:0]` and `tph_st_type[1:0]`, respectively.

Figure 85: Requester Request Interface Signals



X22927-052119

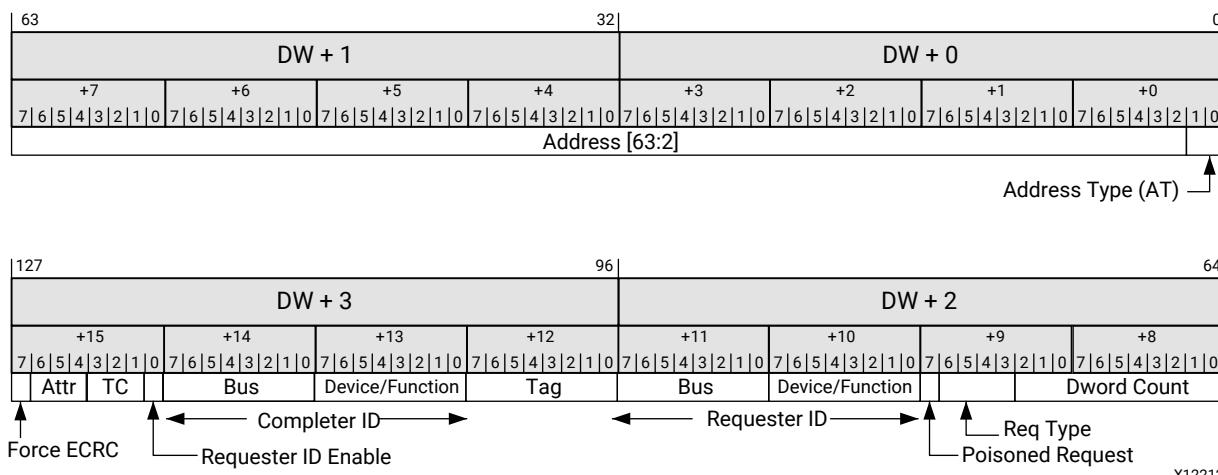
The interface also supports a straddle option that allows the transfer of up to two TLPs in the same beat across the interface. The straddle option can be used only with the Dword-aligned mode, and is not supported when using the 128-bit address aligned mode. The descriptions in the sections below assume a single TLP per beat. The operation of the interface with the straddle option enabled is described in [Straddle Option on RQ Interface](#).

Requester Request Descriptor Formats

The user application must transfer each request to be transmitted on the link to the requester request interface of the core as an independent AXI4-Stream packet. Each packet must start with a descriptor, and can have payload data following the descriptor. The descriptor is always 16 bytes long, and must be sent in the first 16 bytes of the request packet. The descriptor is transferred during the first two beats on a 64-bit interface, and in the first beat on a 128-bit or 256-bit interface. The formats of the descriptor for different request types are illustrated in the following figures.

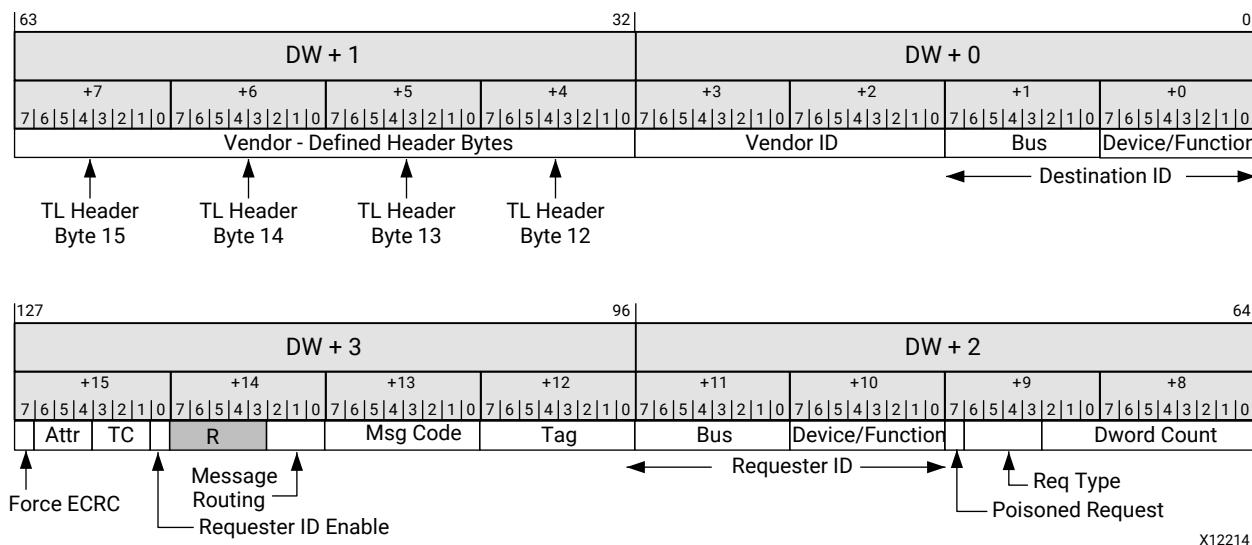
The format of the following figure applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request.

Figure 86: Requester Request Descriptor Format for Memory, I/O, and Atomic Op Requests



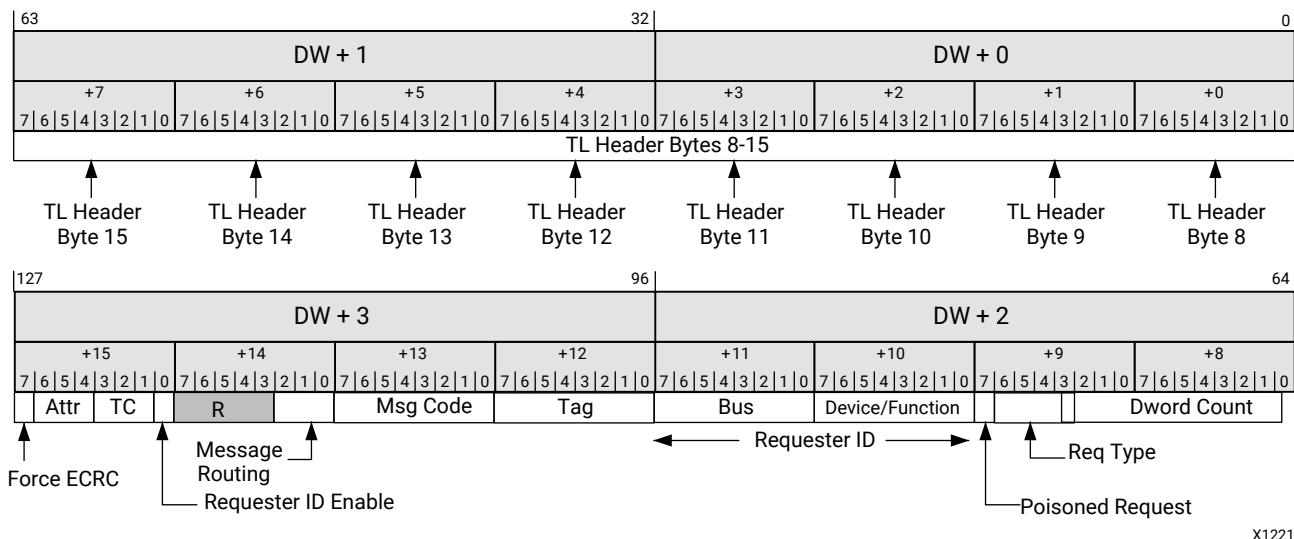
The format of the following figure is used for Vendor-Defined Messages (Type 0 or Type 1) only.

Figure 87: Requester Request Descriptor Format for Vendor-Defined Messages



The format of the following figure is used for all ATS messages (Invalid Request, Invalid Completion, Page Request, PRG Response).

Figure 88: Requester Request Descriptor Format for ATS Messages



For all other messages, the descriptor takes the format of the following figure.

Figure 89: Requester Request Descriptor Format for all other Messages

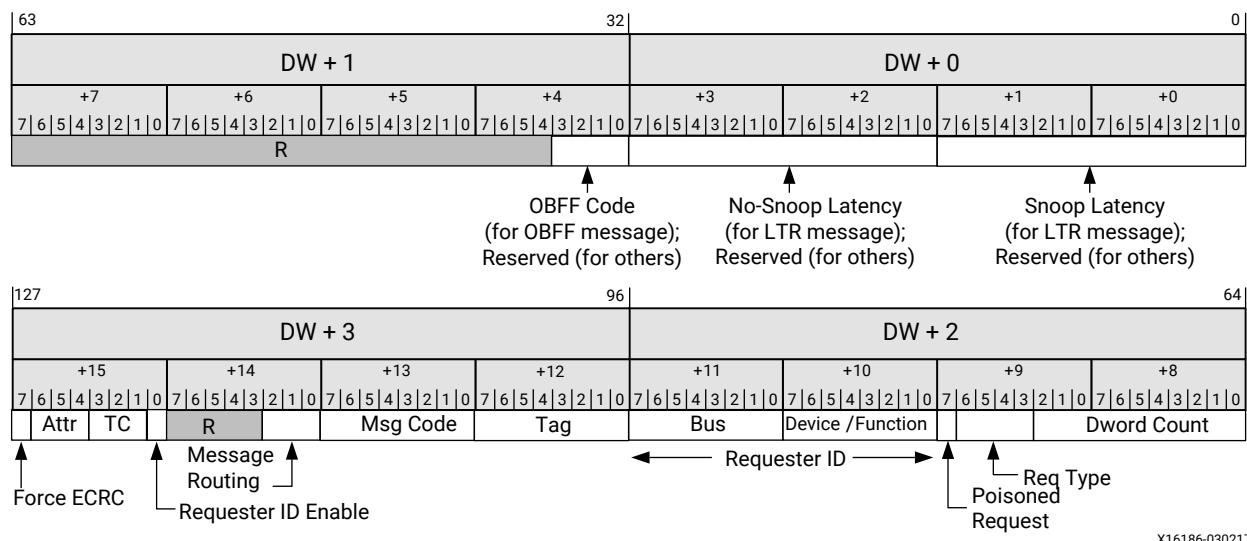


Table 60: Requester Request Descriptor Fields

Bit Index	Field Name	Description
1:0	Address Type	This field is defined for memory transactions and Atomic Operations only. The core copies this field into the AT of the TL header of the request TLP. <ul style="list-style-type: none">• 00: Address in the request is un-translated• 01: Transaction is a Translation Request• 10: Address in the request is a translated address• 11: Reserved
63:2	Address	This field applies to memory, I/O and Atomic Op requests. This is the address of the first Dword referenced by the request. The user logic must also set the First_BE and Last_BE bits in s_axis_rq_tuser to indicate the valid bytes in the first and last Dwords, respectively. When the transaction specifies a 32-bit address, bits [63:32] of this field must be set to 0.
74:64	Dword Count	These 11 bits indicate the size of the block (in Dwords) to be read or written (for messages, size of the message payload). Its range is 0 – 256 Dwords. For I/O accesses, the Dword count is always 1. For a zero length memory read/write request, the Dword count must be 1, with the First_BE bits set to all zeroes. The core does not check the setting of this field against the actual length of the payload supplied (for requests with payload), nor against the maximum payload size or read request size settings of the core.
78:75	Request Type	Identifies the transaction type. The transaction types and their encodings are listed in the following table.
79	Poisoned Request	This bit can be used by the user logic to poison the request TLP being sent. This bit must be set to 0 for all requests, except when the user logic has detected an error in the block of data following the descriptor and wants to communicate this information using the Data Poisoning feature of PCI Express.

Table 60: Requester Request Descriptor Fields (cont'd)

Bit Index	Field Name	Description
87:80	Requester Function/ Device Number	<p>Device and/or Function number of the Requester Function.</p> <p>Endpoint mode:</p> <ul style="list-style-type: none"> • ARI enabled: <ul style="list-style-type: none"> ◦ Bits [87:80] must be set to the Requester Function number. • ARI disabled: <ul style="list-style-type: none"> ◦ Bits [82:80] must be set to the Requester Function number. ◦ Bits [87:83] are not used <p>Upstream Port for Switch use case (Endpoint mode is selected within the IP):</p> <ul style="list-style-type: none"> • ARI enabled: <ul style="list-style-type: none"> ◦ Bits [87:80] must be set to the Requester Function number. • ARI disabled: <ul style="list-style-type: none"> ◦ Bits [82:80] must be set to the Requester Function number. ◦ Bits [87:83] are not used if the request is originating from the switch itself. These bits must be set to the Requester Device number where the request was originated if the switch is relaying the request (Requester is external to the switch). This is used in conjunction with Requester ID Enable bit in the descriptor. <p>Root Port mode (Downstream Port):</p> <ul style="list-style-type: none"> • ARI enabled: <ul style="list-style-type: none"> ◦ Bits [87:80] must be set to the Requester Function number. • ARI disabled: <ul style="list-style-type: none"> ◦ Bits [87:80] must be set to the Requester Function number. ◦ Bits [87:83] must be set to the Requester Device number. This is used in conjunction with Requester ID Enable bit in the descriptor.

Table 60: Requester Request Descriptor Fields (cont'd)

Bit Index	Field Name	Description
95:88	Requester Bus Number	<p>Bus number associated with the Requester Function.</p> <p>Endpoint mode:</p> <ul style="list-style-type: none"> Not used <p>Upstream Port for Switch use case (Endpoint mode is selected within the IP):</p> <ul style="list-style-type: none"> Not used if the request is originating from the switch itself. These bits must be set to the Requester Bus number where the request was originated if the switch is relaying the request (Requester is external to the switch). This is used in conjunction with Requester ID Enable bit in the descriptor. <p>Root Port mode (Downstream Port):</p> <ul style="list-style-type: none"> Must be set to the Requester Bus number. This is used in conjunction with Requester ID Enable bit in the descriptor.
103:96	Tag	PCIe Tag associated with the request. For Posted transactions, the core always uses the value from this field as the tag for the request. For Non-Posted transactions, the core uses the value from this field if the Enable Client Tag is set during core configuration in the Vivado IDE (that is, when tag management is performed by the user logic). If this attribute is not set, tag management logic in the core is responsible for generating the tag to be used, and the value in the tag field of the descriptor is not used.
119:104	Completer ID	This field is applicable only to Configuration requests and messages routed by ID. For these requests, this field specifies the PCI Completer ID associated with the request (these 16 bits are divided into an 8-bit bus number, 5-bit device number, and 3-bit function number in the legacy interpretation mode. In the ARI mode, these 16 bits are treated as an 8-bit bus number + 8-bit Function number.).

Table 60: Requester Request Descriptor Fields (cont'd)

Bit Index	Field Name	Description
120	Requester ID Enable	<p>1'b1: The client supplies Bus, Device, and Function numbers in the descriptor to be populated as the Requester ID field in the TLP header.</p> <p>1'b0: IP uses Bus and Device numbers captured from received Configuration requests and the client supplies Function numbers in the descriptor to be populated as the Requester ID field in the TLP header. When Requester ID enable is 0 the device number fields in descriptor should also be 0.</p> <p>Endpoint mode:</p> <ul style="list-style-type: none"> Must be set to 1'b0. <p>Upstream Port for Switch use case (Endpoint mode is selected within the IP):</p> <ul style="list-style-type: none"> Set to 1'b0 when the request is originating from the switch itself. Set to 1'b1 when the switch is relaying the request (Requester is external to the switch). This is used in conjunction with Requester Bus Number bits [95:88] and Requester Function/Device Number bits [87:83] when ARI is not enabled. <p>Root Port mode:</p> <ul style="list-style-type: none"> Must be set to 1'b1. This is used in conjunction with Requester Bus Number bits [95:88] and Requester Function/Device Number bits [87:83] when ARI is not enabled.
123:121	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request.
126:124	Attributes	<p>These bits provide the setting of the Attribute bits associated with the request. Bit 124 is the No Snoop bit, and bit 125 is the Relaxed Ordering bit. Bit 126 is the ID-Based Ordering bit, and can be set only for memory requests and messages.</p> <p>The core forces the attribute bits to 0 in the request sent on the link if the corresponding attribute is not enabled in the Function's PCI Express Device Control Register.</p>
111:104	Message Code	<p>This field is defined for all messages. It contains the 8-bit Message Code to be set in the TL header.</p> <p>Appendix F of the PCI Express 3.0 Specifications (available at http://www.pcisig.com/specifications) provides a complete list of the supported Message Codes.</p>
114:112	Message Routing	This field is defined for all messages. The core copies these bits into the 3-bit Routing field r[2:0] of the TL header of the Request TLP.
15:0	Destination ID	This field applies to Vendor-Defined Messages only. When the message is routed by ID (that is, when the Message Routing field is 010 binary), this field must be set to the Destination ID of the message.
63:32	Vendor-Defined Header	This field applies to Vendor-Defined Messages only. It is copied into Dword 3 of the TL header.
63:0	ATS Header	This field is applicable to ATS messages only. It contains the bytes that the core copies into Dwords 2 and 3 of the TL header.

Requester Memory Write Operation

In both Dword-aligned and 128-bit address aligned modes, the transfer starts with the sixteen descriptor bytes, followed by the payload bytes. The user application must keep the signal `s_axis_rq_tvalid` asserted over the duration of the packet. The core treats the deassertion of `s_axis_rq_tvalid` during the packet transfer as an error, and nullifies the corresponding request TLP transmitted on the link to avoid data corruption.

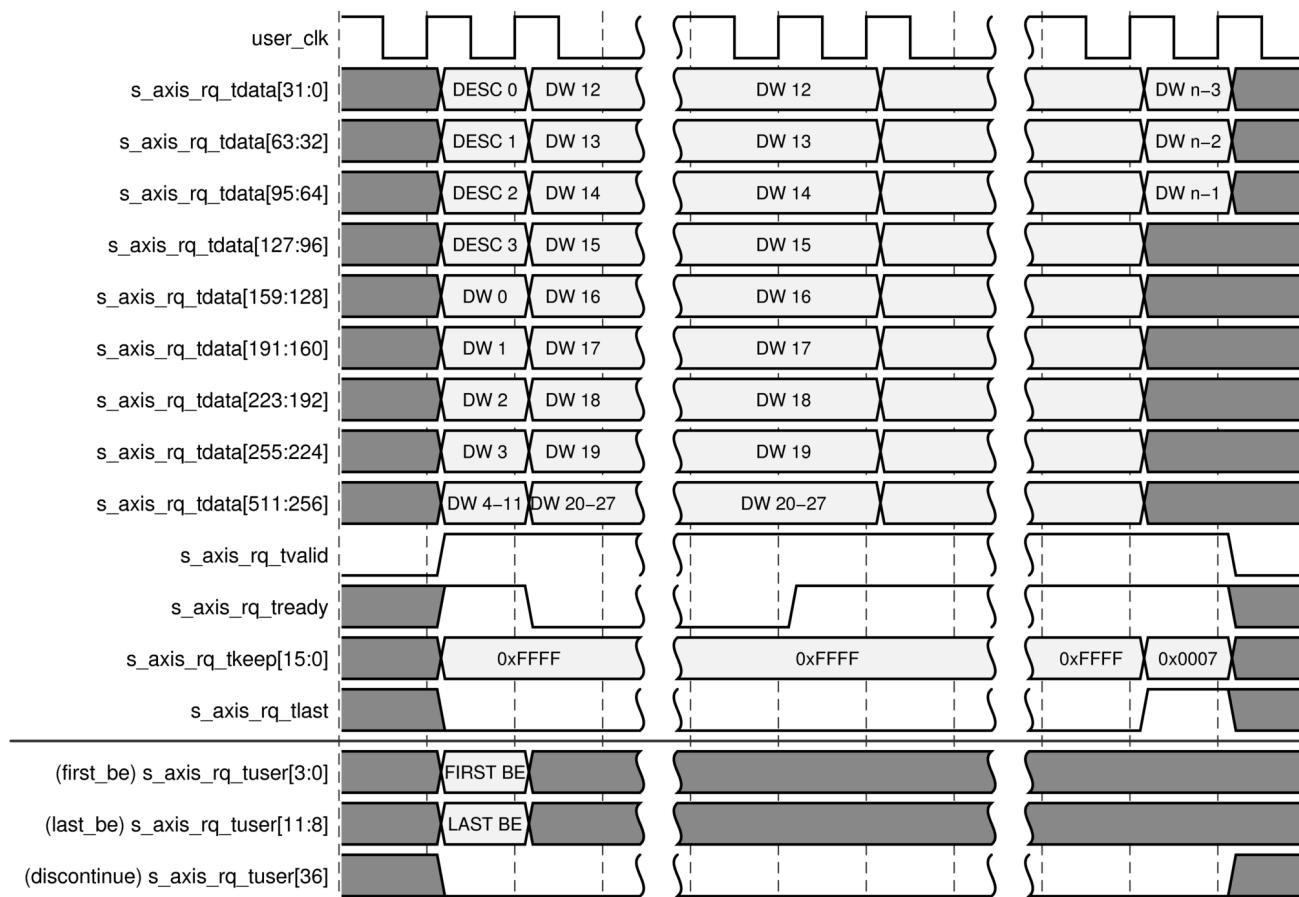
The user application must also assert the signal `s_axis_rq_tlast` in the last beat of the packet. The core are by pull down `s_axis_rq_tready` in any cycle if it is not ready to accept data. The user application must not change the values on `s_axis_rq_tdata` and `s_axis_rq_tlast` during the transfer when the core has deasserted `s_axis_rq_tready`. The AXI4-Stream interface signals `m_axis_rq_tkeep` (one per Dword position) must be set to indicate the valid Dwords in the packet including the descriptor and any null bytes inserted between the descriptor and the payload. That is, the `m_axis_rq_tkeep` bits must be set to 1 contiguously from the first Dword of the descriptor until the last Dword of the payload. During the transfer of a packet, the `m_axis_rq_tkeep` bits can be 0 only in the last beat of the packet, when the packet does not fill the entire width of the interface.

The requester request interface also includes the First Byte Enable and the Last Enable bits in the `s_axis_rq_tuser` bus. These must be set in the first beat of the packet, and provides information of the valid bytes in the first and last Dwords of the payload.

The user application must limit the size of the payload transferred in a single request to the maximum payload size configured in the core, and must ensure that the payload does not cross a 4 Kbyte boundary. For memory writes of two Dwords or less, the 1s in `first_be[7:0]` and `last_be[7:0]` are not be contiguous. For the special case of a zero-length memory write request, the user application must provide a dummy `one_dword` payload with `first_be[7:0]` and `last_be[7:0]` both set to all 0s. In all other cases, the 1 bits in `first_be[7:0]` and `last_be[7:0]` must be contiguous. In Straddled case, `addr_offset[3:2]`, `first_be[7:4]`, and `last_be[7:4]` are used to indicate second TLP information while `addr_offset[1:0]`, `first_be[3:0]`, and `last_be[3:0]` are used to indicate the first TLP information on that data beat.

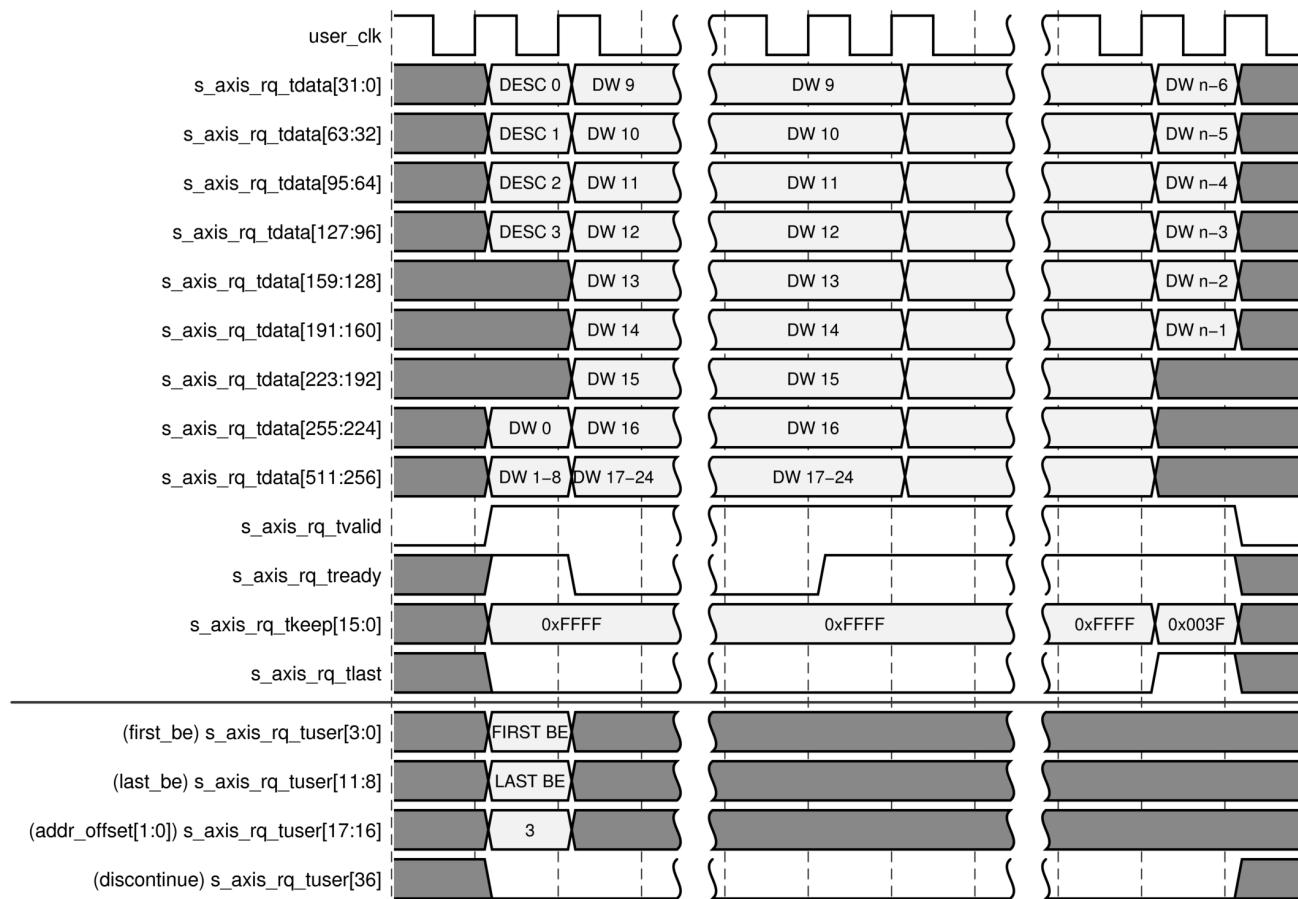
The following figure illustrates the Dword-aligned transfer of a memory write request from the user logic across the requester request interface. For the purpose of illustration, the size of the data block being written into user memory is assumed to be n Dwords, for some $n = k^*16 - 1$, where $k > 1$.

Figure 90: Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode)



The following figure illustrates the 128-bit address aligned transfer of a memory write request from the user application across the requester request interface. For the purpose of illustration, the starting Dword offset of the data block is assumed to be $(m*16 + 3)$, for some integer $m > 0$. Its size is assumed to be n Dwords, for some $n = k*16 - 1$, $k > 1$. In the 128-bit address-aligned mode, the delivery of the payload always starts in the second 128-bit quarter of the 512-bit word, following the descriptor in the first quarter. The user application must communicate the offset of the first Dword of the payload in the `addr_offset [3 : 0]` field of the `s_axis_rq_tuser` bus. The user application must also set the bits in `first_be [7 : 0]` to indicate the valid bytes in the first Dword and the bits in `last_be [7 : 0]` to indicate the valid bytes in the last Dword of the payload.

Figure 91: Memory Write Transaction on the Requester Request Interface (128-bit Address Aligned Mode)



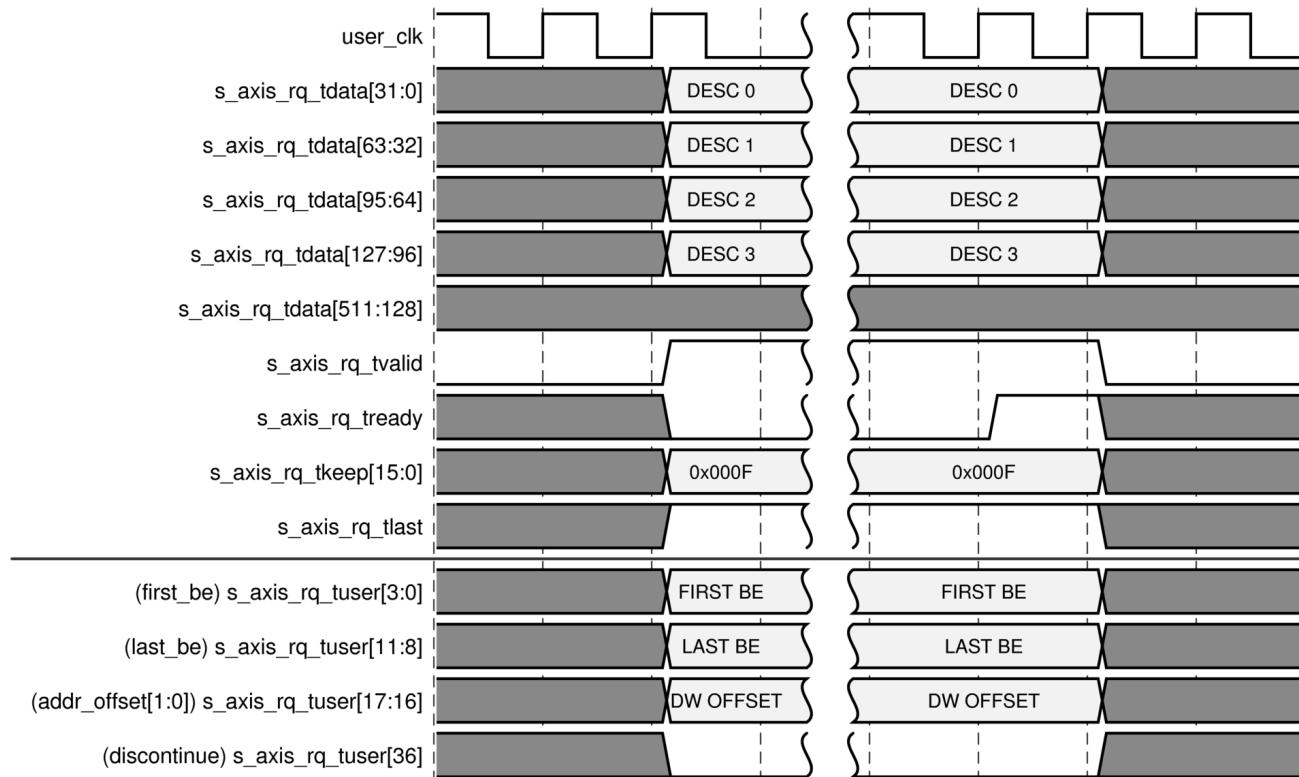
Non-Posted Transactions with No Payload

Non-Posted transactions with no payload (memory read requests, I/O read requests, Configuration read requests) are transferred across the requester request interface in the same manner as a memory write request, except that the AXI4-Stream packet contains only the 16-byte descriptor. The following figure illustrates the transfer of a memory read request across the requester request interface. The signal `s_axis_rq_tvalid` must remain asserted over the duration of the packet. The core are pull down `s_axis_rq_tready` to prolong the beat. The signal `s_axis_rq_tlast` must be set in the last beat of the packet, and the bits in `s_axis_rq_tkeep[15:0]` must be set in all Dword positions where a descriptor is present.

The user application must indicate the valid bytes in the first and last Dwords of the data block using the fields `first_be[7:0]` and `last_be[7:0]`, respectively, in the `s_axis_rq_tuser` bus. For the special case of a zero-length memory read, the length of the request must be set to one Dword, with both `first_be[7:0]` and `last_be[7:0]` set to all 0s. The user application must also communicate the offset of the first Dword of the payload of the resulting Completion,

when delivered over the requester completion interface, in the `addr_offset[3:0]` field of the `s_axis_rq_tuser` bus. In Straddled case, `addr_offset[3:2]`, `first_be[7:4]`, and `last_be[7:4]` are used to indicate second TLP information while `addr_offset[1:0]`, `first_be[3:0]`, and `last_be[3:0]` are used to indicate the first TLP information on that data beat.

Figure 92: Memory Read Transaction on the Requester Request Interface



Non-Posted Transactions with a Payload

The transfer of a Non-Posted request with a payload (an I/O write request, Configuration write request, or Atomic Operation request) is similar to the transfer of a memory write request, with the following changes in how the payload is aligned on the data path:

- In the Dword-aligned mode, the first Dword of the payload follows the last Dword of the descriptor, with no gaps between them.
- In the 128-bit address aligned mode, the payload must start in the second 128-bit quarter of the first beat, following the descriptor. The payload are start at any of four Dword positions in this quarter. The offset of its first Dword must be specified in the field `addr_offset[3:0]` of the `s_axis_rq_tuser` bus.

In the case of I/O and Configuration write requests, the valid bytes in the one-Dword payload must be indicated using `first_be[7:0]`. For Atomic Operation requests, all bytes in the first and last Dwords are assumed valid.

Message Requests on the Requester Interface

The transfer of a message on the requester request interface is similar to that of a memory write request, except that a payload are not always be present. The transfer starts with the 128-bit descriptor, followed by the payload, if present. The first Dword of the payload must immediately follow the descriptor, regardless of the address alignment mode in use. The `addr_offset[3:0]` field in the `s_axis_rq_tuser` bus must be set to 0 for messages when the address-aligned mode is in use. The core determines the end of the payload from `s_axis_rq_tlast` and `s_axis_rq_tkeep` signals. The First Byte Enable and Last Byte Enable bits (`first_be[7:0]` and `last_be[7:0]`) are not used for message requests.

Aborting a Transfer

For any request that includes an associated payload, The user application are abort the request at any time during the transfer of the payload by asserting the discontinue signal in the `s_axis_rq_tuser` bus. The core nullifies the corresponding TLP on the link to avoid data corruption.

The user application are assert this signal in any cycle during the transfer, when the request being transferred has an associated payload. The user application are either choose to terminate the packet prematurely in the cycle where the error was signaled (by asserting `s_axis_rq_tlast`), or are continue until all bytes of the payload are delivered to the core. In the latter case, the core treats the error as sticky for the following beats of the packet, even if the user logic deasserts the discontinue signal before reaching the end of the packet.

The `discontinue` signal can be asserted only when `s_axis_rq_tvalid` is High. The core samples this signal when `s_axis_rq_tvalid` and `s_axis_rq_tready` are both High. Thus, once asserted, it should not be deasserted until `s_axis_rq_tready` is High. The user application must not start a new packet in the same beat when a previous packet is aborted by asserting the `discontinue` input.

When the core is configured as an Endpoint, this error is reported by the core to the Root Complex it is attached to, as an Uncorrectable Internal Error using the Advanced Error Reporting (AER) mechanisms.

Straddle Option on RQ Interface

The PCIe® core has the capability to start the transfer of a new request packet on the requester request interface in the same beat when the previous request has ended on or before Dword position 7 on the data bus. This straddle option is enabled during core customization in the Vivado® IDE. The straddle option can be used only with the Dword-aligned mode.

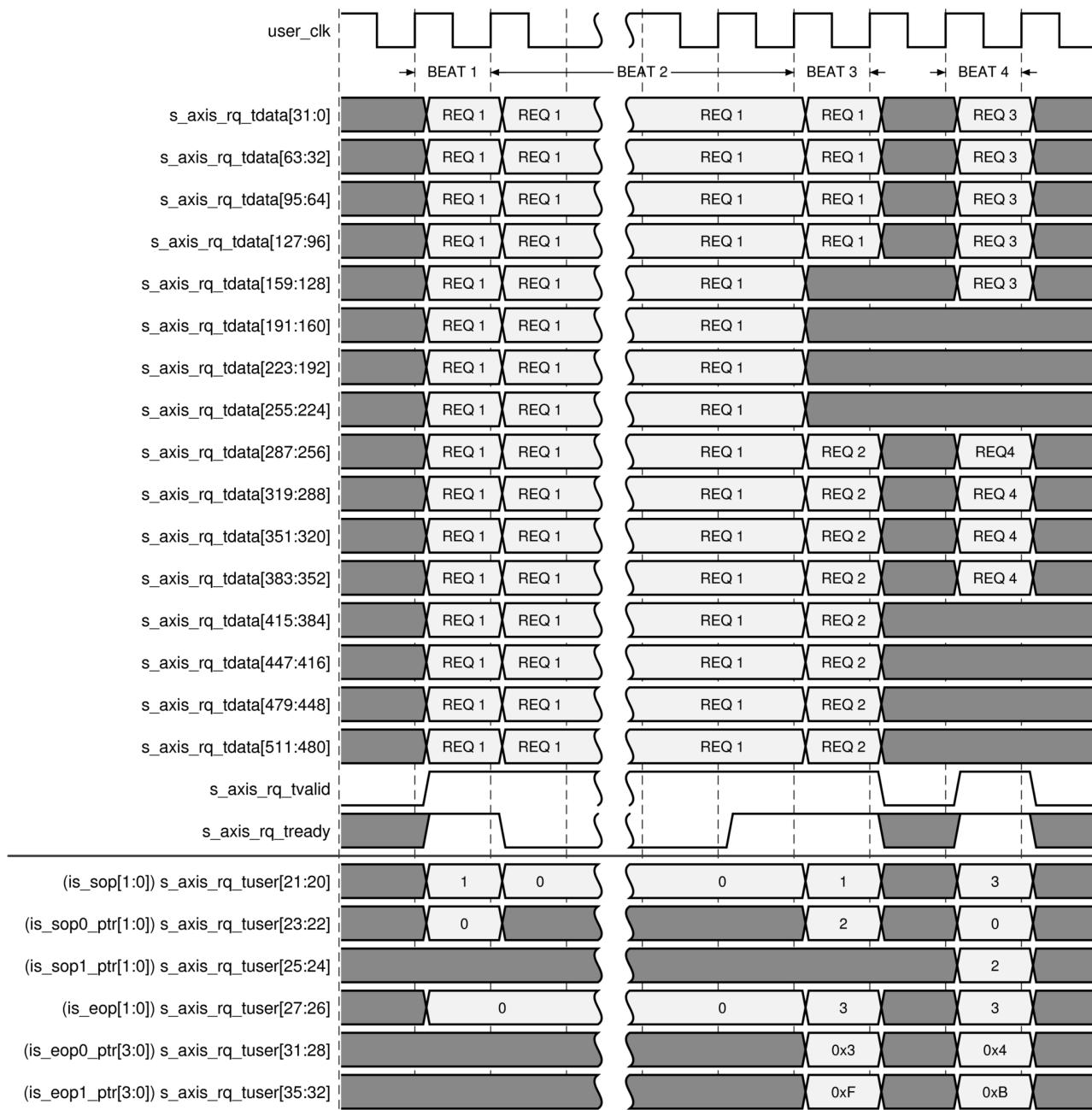
When the straddle option is enabled, request TLPs are transferred on the AXI4-Stream interface as a continuous stream, with no packet boundaries. Thus, the signals `m_axis_rq_tkeep` and `m_axis_rq_tlast` are not useful in determining the boundaries of TLPs delivered on the interface. Instead, delineation of TLPs is performed using the following signals provided within the `m_axis_rq_tuser` bus.

- `is_sop[0]`: This input must be set High in a beat when there is at least one request TLP starting in the beat. The position of the first byte of the descriptor of this TLP is determined as follows:
 - If the previous TLP ended before this beat, the first byte of the descriptor is in byte lane 0.
 - If a previous TLP is continuing in this beat, the first byte of this descriptor is in byte lane 32. This is possible only when the previous TLP ends in the current beat, that is when `is_eop[0]` is also set.
- `is_sop0_ptr[1:0]`: When `is_sop[0]` is set, this field must indicate the offset of the first request TLP starting in the current beat. Valid settings are `2'b00` (TLP starting at Dword 0) and `2'b10` (TLP starting at Dword 8).
- `is_sop[1]`: This input must be set High in a beat when there are two request TLPs starting in the same beat. The first TLP must always start at byte position 0 and the second TLP at byte position 32. The user application are start a second TLP at byte position 32 only if the previous TLP ended before byte position 32 in the same beat, that is only if `is_eop[0]` is also set in the same beat.
- `is_sop1_ptr[1:0]`: When `is_sop[1]` is set, this field must provide the offset of the second TLP starting in the current beat. Its only valid setting is `2'b10` (TLP starting at Dword 8).
- `is_eop[0]`: This input is used to indicate the end of a request TLP. Its assertion signals that there is at least one TLP ending in this beat.
- `is_eop0_ptr[3:0]`: When `is_eop[0]` is asserted, `is_eop0_ptr[3:0]` must provide the offset of the last Dword of the corresponding TLP ending in this beat.
- `is_eop[1]`: This input is set High when there are two TLPs ending in the current beat. `is_eop[1]` can be set only when the signals `is_eop[0]` and `is_sop[0]` are also be High in the same beat.
- `is_eop1_ptr[3:0]`: When `is_eop[1]` is asserted, `is_eop1_ptr[3:0]` must provide the offset of the last Dword of the second TLP ending in this beat. Because the second TLP can start only on byte lane 32, it can only end at a byte lane in the range 43-63. Thus the offset `is_eop1_ptr[3:0]` can only take a value in the range 10-15.

When a second TLP starts in the same beat, the First Byte Enable and Last Byte Enable bits of the second TLP are specified by the bit fields `first_be[7:4]` and `last_be[7:4]`, respectively, in the tuser bus.

The following figure illustrates the transfer of four request TLPs on the requester request interface when the straddle option is enabled. For all TLPs, the first Dword of the payload always follows the descriptor without any gaps. The first request TLP (REQ 1) starts at Dword position 0 of Beat 1 and ends in Dword position 3 of Beat 3. The second TLP (REQ 2) starts in Dword position 8 of the same beat. This second TLP has only a four-Dword payload, so it also ends in the same beat. The third and fourth Completion TLPs are transferred completely in Beat 4, as REQ 3 has only a one-Dword payload and REQ 4 has no payload.

Figure 93: Transfer of Request TLPs on the Requester Request Interface with the Straddle Option Enabled



Tag Management for Non-Posted Transactions

The requester side of the core maintains the state of all pending Non-Posted transactions (memory reads, I/O reads and writes, configuration reads and writes, Atomic Operations) initiated by the user application, so that the completions returned by the targets can be matched against the corresponding requests. The state of each outstanding transaction is held in a Split Completion Table in the requester side of the interface, which has a capacity of up to 256 Non-Posted transactions. The returning Completions are matched with the pending requests using an 8-bit tag. There are two options for management of these tags:

- **Internal Tag Management:** This mode of operation is selected during core customization in the Vivado® IDE. In this mode, logic within the core is responsible for allocating the tag for each Non-Posted request initiated from the requester side. The core maintains a list of free tags and assigns one of them to each request when the user logic initiates a Non-Posted transaction, and communicates the assigned tag value to the user logic through the output `pcie_rq_tag0[7:0]` and `pcie_rq_tag1[7:0]`. The value on this bus is valid when the core asserts `pcie_rq_tag_vld0` and `pcie_rq_tag_vld1`. Use of `pcie_rq_tag_vld0` and `pcie_rq_tag_vld1` are orthogonal to whether the Straddle option is enabled. The integrated block can use either the `pcie_rq_tag_vld0` or `pcie_rq_tag_vld1` port to showcase the valid tags. The user logic must copy the tag so that any Completions delivered by the core in response to the request can be matched to the request.

In this mode, logic within the core checks for the Split Completion Table full condition, and backpressures a Non-Posted request from the user logic (using `s_axis_rq_tready`) if the total number of Non-Posted requests currently outstanding has reached its limit.

- **External Tag Management:** This mode of operation is selected during core customization in the Vivado IDE. In this mode, the user logic is responsible for allocating the tag for each Non-Posted request initiated from the requester side. The user logic must choose the tag value without conflicting with the tags of all other Non-Posted transactions outstanding at that time, and must communicate this chosen tag value to the core within the request descriptor. The core still maintains the outstanding requests in its Split Completion Table and matches the incoming Completions to the requests, but does not perform any checks for the uniqueness of the tags, or for the Split Completion Table full condition.

When internal tag management is in use, the core asserts `pcie_rq_tag_vld` for one cycle for each Non-Posted request, after it has placed its allocated tag on `pcie_rq_tag`. When straddle option is enabled, the core provides up to two allocated tags in the same cycle on this interface. The states of the signals `pcie_rq_tag_vld` and `pcie_rq_tag` must be interpreted as follows:

- Assertion of `pcie_rq_tag_vld0` in any cycle indicates that the core has placed an allocated tag on `pcie_rq_tag0[7:0]`.

- Simultaneous assertion of `pcie_rq_tag_vld0` and `pcie_rq_tag_vld1` in the same cycle indicates that the core has placed two allocated tags, the first on `pcie_rq_tag0[7:0]` and the second on `pcie_rq_tag1[7:0]`. The tag on `pcie_rq_tag0[7:0]` corresponds to an earlier request sent by the user logic and the tag on `pcie_rq_tag1[7:0]` corresponds to a later request.
- `pcie_rq_tag_vld1` is never asserted when `pcie_rq_tag_vld0` is not asserted. That is, when there is only one tag to communicate in any cycle, it is always communicated on `pcie_rq_tag0[7:0]`.
- When straddle is not in use, only a single tag can be communicated in any cycle, and `pcie_rq_tag_vld1` is never asserted.

There can be a delay of several cycles between the transfer of the request on the `s_axis_rq_tdata` bus and the assertion of `pcie_rq_tag_vld` by the core to provide the allocated tag for the request. The user logic are, meanwhile, continue to send new requests. The tags for requests are communicated on the `pcie_rq_tag` bus in FIFO order, so it is easy for the user logic to associate the tag value with the request it transferred.

Avoiding Head-of-Line Blocking for Posted Requests

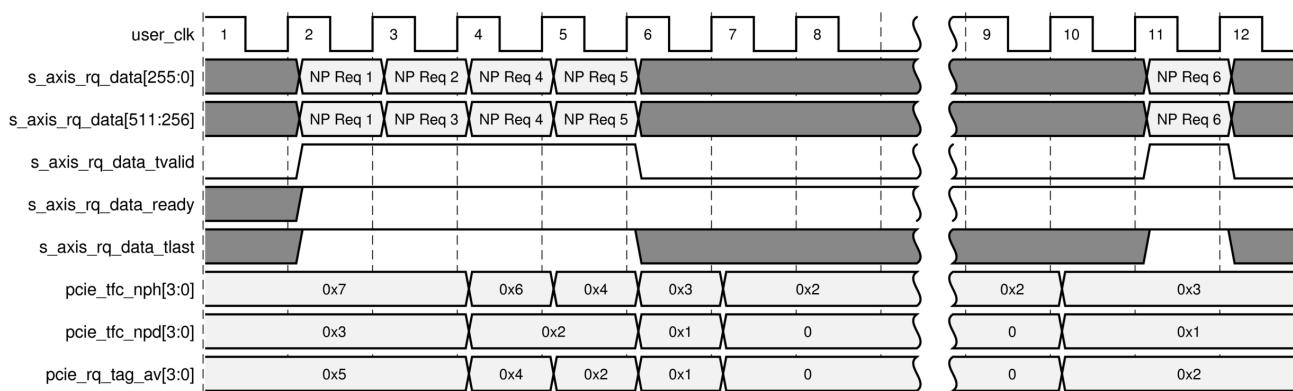
The core holds a Non-Posted request received on its requester request interface for lack of transmit credit or lack of available tags. This could potentially result in HOL blocking for Posted transactions. Such a condition can be prevented if the user logic has the ability to check the availability of transmit credit and tags for Non-Posted transactions. The core provides the following signals for this purpose:

- `pcie_tfc_nph_av[3:0]`: These outputs indicate the Header Credit currently available for Non-Posted requests (0000 = no credit available, 0001 = 1 credit available, 0010 = 2 credits, ..., 1111 = 15 or more credits available).
- `pcie_tfc_npd_av[3:0]`: These outputs indicate the Data Credit currently available for Non-Posted requests (0000= no credit available, 0001 = 1 credit available, 0010 = 2 credits, ..., 1111 = 15 or more credits available).
- `pcie_rq_tag_av[3:0]`: These outputs indicate the number of free tags currently available for allocation to Non-Posted requests (0000 = no tags available, 0001 = 1 tag available, 0010 = 2 tags available, ..., 1111 = 15 or more tags available).

The user logic are optionally check these outputs before transmitting Non-Posted requests. Because of internal pipeline delays, the information on these outputs is delayed by two user clock cycles from the cycle in which the last byte of the descriptor is transferred on the requester request interface, so the user logic must adjust these values taking into account any Non-Posted requests transmitted in the two previous clock cycles. The following figure illustrates the operation of these signals. In this example, the core initially had 7 Non-Posted Header Credits and 3 Non-Posted Data Credits, and had 5 free tags available for allocation. Request 1 from the user logic had a one-Dword payload, and therefore consumed 1 header and data credit each, and also one tag. Requests 2 and 3 (straddled) in the next clock cycle 3 consumed 1 header credit each, but no data credit. When the user logic presents Request 4 in clock cycle 4, it must adjust

the available credit and available tag count by taking into account Requests 1, 2 and 3, presented in the two previous cycles. Request 4 consumes 1 header credit and one data credit. When the user logic presents Request 5 in clock cycle 5, it must adjust the available credit and available tag count by taking into account Requests 2, 3 and 4. If Request 5 consumes one header credit and one data credit the available data credit is two cycles later, as also the number of available tags. Thus, Request 6 must wait for the availability of new credit.

Figure 94: Operation of credit and tag availability signals on the Requester Request Interface



Note: If the user logic opts in to use the `pcie_tfc_*` interface to monitor transmit credit availability, ensure that no more non-posted packets go into the RQ interface after `pcie_tfc_npd_av` or `pcie_tfc_nph_av` reaches 0. The integrated block will not lose the non-posted packets issued beyond this point; however, the `pcie_tfc_*` interface will no longer provide an accurate credit accounting.

Similar transmit credit information is also provided in the `cfg_fc_npd` and `cfg_fc_nph` interface when `cfg_fc_sel` is set to the Transmit credits available mode.

Maintaining Transaction Order

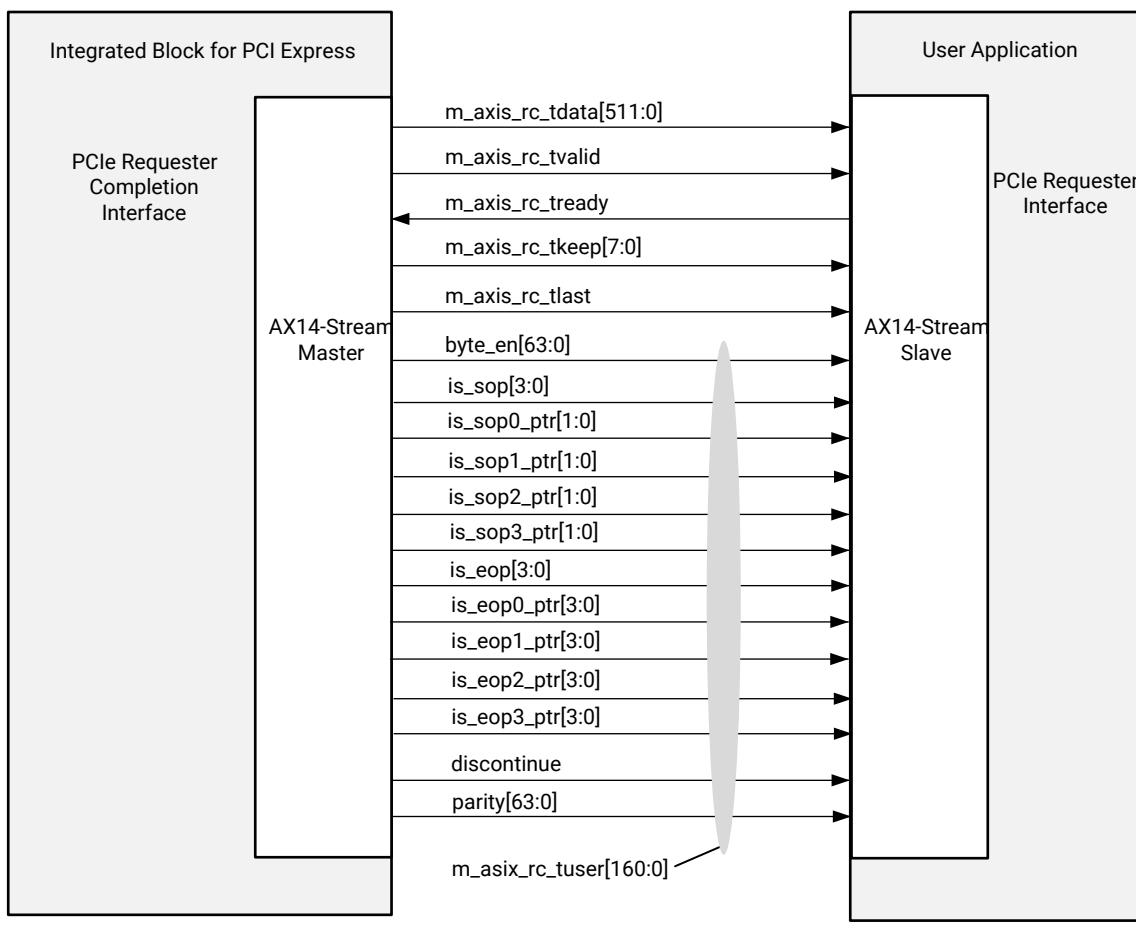
The core does not change the order of requests received from the user on its requester interface when it transmits them on the link. In cases where the user logic would like to have precise control of the order of transactions sent on the requester request interface and the completer completion interface (typically to avoid Completions from passing Posted requests when using strict ordering), the core provides a mechanism for the user logic to monitor the progress of a Posted transaction through its pipeline, so that it can determine when to schedule a Completion on the completer completion interface without the risk of passing a specific Posted request transmitted from the requester request interface.

When transferring a Posted request (memory write transactions or messages) across the requester request interface, the user logic are provide an optional 6-bit sequence number to the PCIe® core in its first beat. The sequence number field `seq_num0 [5 : 0]` within `s_axis_rq_tuser` is used to send the sequence number for the first TLP starting in the beat, and the field `seq_num1 [5 : 0]` is used to send the sequence number for the second TLP

starting in the beat (if present). The user logic can then monitor the `pcie_rq_seq_num0[5:0]` and `pcie_rq_seq_num1[5:0]` outputs of the core for these sequence numbers to appear. When the transaction has reached a stage in the internal transmit pipeline of the core where a Completion is unable to pass it, the core asserts `pcie_rq_seq_num_vld0` for one cycle and provides the sequence number of the Posted request on the `pcie_rq_seq_num0[5:0]` output. If there is a second Posted request in the pipeline in the same cycle, the core also asserts `pcie_rq_seq_num_vld1` in the same cycle and provides the sequence number of the second Posted request on the `pcie_rq_seq_num1[5:0]` output. The user logic must therefore monitor both sets of the sequence number outputs to check if a specific TLP has reached the pipeline stage. Any Completions transmitted by the core after the sequence number has appeared on `pcie_rq_seq_num0[5:0]` or `pcie_rq_seq_num1[5:0]` is guaranteed not to pass the corresponding Posted request in the internal transmit pipeline of the core.

Requester Completion Interface Operation (512-bits)

Figure 95: Requester Completion Interface Signals



X16714-072420

The previous figure illustrates the signals associated with the requester completion interface of the core. When straddle is not enabled, the core delivers each TLP on this interface as an AXI4-Stream packet. The packet starts with a 96-bit descriptor, followed by data in the case of Completions with a payload.

The requester completion interface supports two distinct data alignment modes for transferring payloads, which are during core customization in the Vivado® IDE. In the Dword-aligned mode, the core transfers the first Dword of the Completion payload immediately after the last Dword of the descriptor. In the 128-bit address aligned mode, the core starts the payload transfer in the second 128-bit quarter of the 512-bit word, following the descriptor in the first quarter. The first Dword of the payload can be in any of the four possible Dword positions in the second quarter, and its offset f is determined by address offset provided by the user logic when it sent the request to the core (that is, the setting of the `addr_offset` input of the requester request interface). Thus, the 128-bit address aligned mode can be used on the requester completion interface only if the requester request interface is also configured to use the 128-bit address aligned mode.

Requester Completion Descriptor Format

The requester completion interface of the core sends completion data received from the link to the user application as AXI4-Stream packets. Each packet starts with a descriptor, and can have payload data following the descriptor. The descriptor is always 12 bytes long, and is sent in the first 12 bytes of the completion packet. When the completion data is split into multiple Split Completions, the core sends each Split Completion as a separate AXI4-Stream packet, with its own descriptor.

The format of the requester completion descriptor is illustrated in the following figure. The individual fields of the requester completion descriptor are described in the following table.

Figure 96: Requester Completion Descriptor Format

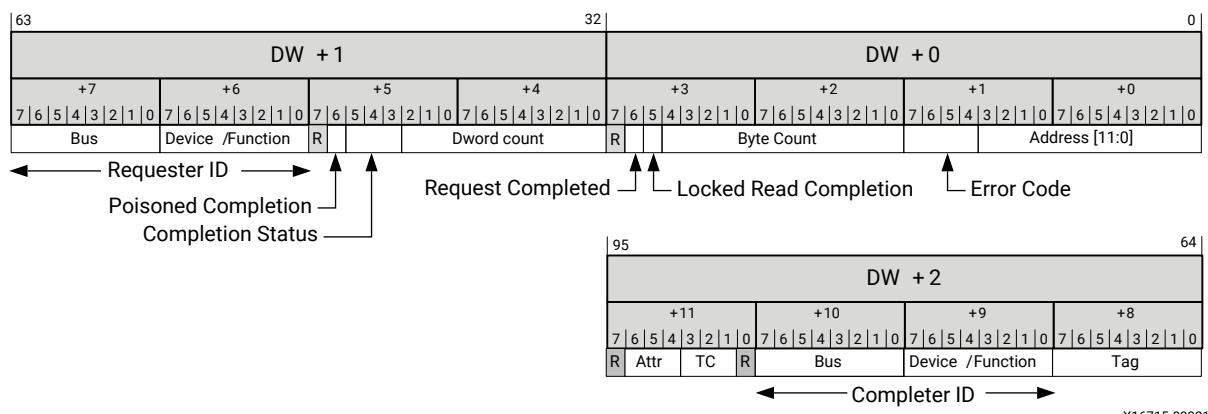


Table 61: Requester Completion Descriptor Fields

Bit Index	Field Name	Description
11:0	Lower Address	<p>This field provides the 12 least significant bits of the first byte referenced by the request. The core returns this address from its Split Completion Table, where it stores the address and other parameters of all pending Non-Posted requests on the requester side.</p> <p>When the Completion delivered has an error, only bits [6:0] of the address should be considered valid.</p> <p>This is a byte-level address.</p> <p>For ATS translation requests, this field is reserved and implied to be zero.</p>
15:12	Error Code	<p>Completion error code. These three bits encode error conditions detected from error checking performed by the core on received Completions. Its encodings are:</p> <ul style="list-style-type: none"> • 0000: Normal termination (all data received). • 0001: The Completion TLP is Poisoned. • 0010: Request terminated by a Completion with UR, CA or CRS status. • 0011: Request terminated by a Completion with no data. • 0100: The current Completion being delivered has the same tag of an outstanding request, but its Requester ID, TC, or Attr field does not match with the parameters of the outstanding request. • 0101: Error in starting address. The low address bits in the Completion TLP header did not match with the starting address of the next expected byte for the request. • 0110: Invalid tag. This Completion does not match the tags of any outstanding request. • 0111: Invalid byte count. The byte count in the Completion was higher than the total number of bytes expected for the request. • 1000: Request terminated by a Completion timeout. The other fields in the descriptor, except bit [30], the requester Function [55:48], and the tag field [71:64], are invalid in this case, because the descriptor does not correspond to a Completion TLP. • 1001: Request terminated by a Function-Level Reset (FLR) targeted at the Function that generated the request. The other fields in the descriptor, except bit [30], the requester Function [55:48], and the tag field [71:64], are invalid in this case, because the descriptor does not correspond to a Completion TLP.
28:16	Byte Count	<p>These 13 bits can have values in the range of 0 – 4,096 bytes. If a Memory Read Request is completed using a single Completion, the Byte Count value indicates Payload size in bytes. This field must be set to 4 for I/O read Completions and I/O write Completions. The byte count must be set to 1 while sending a Completion for a zero-length memory read, and a dummy payload of 1 Dword must follow the descriptor.</p> <p>For each Memory Read Completion, the Byte Count field must indicate the remaining number of bytes required to complete the Request, including the number of bytes returned with the Completion.</p> <p>If a Memory Read Request is completed using multiple Completions, the Byte Count value for each successive Completion is the value indicated by the preceding Completion minus the number of bytes returned with the preceding Completion.</p>
29	Locked Read Completion	This bit is set to 1 when the Completion is in response to a Locked Read request. It is set to 0 for all other Completions.

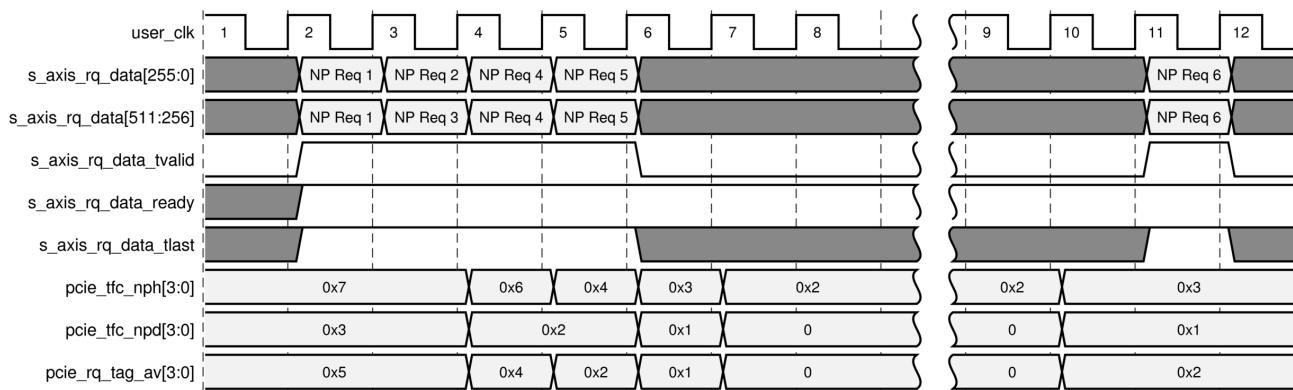
Table 61: Requester Completion Descriptor Fields (cont'd)

Bit Index	Field Name	Description
30	Request Completed	The core asserts this bit in the descriptor of the last Completion of a request. The assertion of the bit indicates normal termination of the request (because all data has been received), or abnormal termination because of an error condition. The user logic can use this indication to clear its outstanding request. When tags are assigned by the user logic, the user logic should not reassign a tag allocated to a request until it has received a Completion Descriptor from the core with a matching tag field and the Request Completed bit set to 1.
42:32	Dword Count	These 11 bits indicate the size of the payload of the current packet in Dwords. Its range is 0 – 1K Dwords. This field is set to 1 for I/O read Completions and 0 for I/O write Completions. The Dword count is also set to 1 while transferring a Completion for a zero-length memory read. In all other cases, the Dword count corresponds to the actual number of Dwords in the payload of the current packet.
45:43	Completion Status	These bits reflect the setting of the Completion Status field of the received Completion TLP. The valid settings are: <ul style="list-style-type: none"> • 000: Successful Completion. • 001: Unsupported Request (UR). • 010: Configuration Request Retry Status (CRS). • 100: Completer Abort (CA).
46	Poisoned Completion	This bit is set to indicate that the Poison bit in the Completion TLP was set. Data in the packet should then be considered corrupted.
63:48	Requester ID	PCI Requester ID associated with the Completion.
71:64	Tag	PCIe Tag associated with the Completion.
87:72	Completer ID	Completer ID received in the Completion TLP. (These 16 bits are divided into an 8-bit bus number, 5-bit device number, and 3-bit function number in the legacy interpretation mode. In ARI mode, these 16 bits must be treated as an 8-bit bus number + 8-bit Function number.)
91:89	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the Completion.
94:92	Attributes	PCIe attributes associated with the Completion. Bit 92 is the No Snoop bit, bit 93 is the Relaxed Ordering bit, and bit 94 is reserved.

Transfer of Completions with No Data

The following figure illustrates the transfer of a Completion TLP received from the link with no associated payload across the requester completion interface. The timing diagrams in this section assume that the Completions are not straddled on the interface. The straddle feature is described in [Straddle Option for RC Interface](#).

Figure 97: Transfer of a Completion with no Data on the Requester Completion Interface



The entire transfer of the Completion TLP takes only a single beat on the interface. The core keeps the signal `m_axis_rc_tvalid` asserted over the duration of the packet. The user logic can prolong a beat at any time by pulling down `m_axis_rc_tready`. The AXI4-Stream interface signals `m_axis_rc_tkeep` (one per Dword position) indicate the valid descriptor Dwords in the packet. That is, the `m_axis_rc_tkeep` bits are set to 1 contiguously from the first Dword of the descriptor until its last Dword. The signal `m_axis_rc_tlast` is always asserted, indicating that the packet ends in its current beat.

The `m_axi_rc_tuser` bus also includes a signal `is_sop[0]`, which is asserted in the first beat of every packet. The user logic are optionally use this signal to qualify the start of the descriptor on the interface. When the straddle option is not in use, none of the other sop and eop indications within `m_axi_rc_tuser` are relevant to the transfer of Completions.

Transfer of Completions with Data

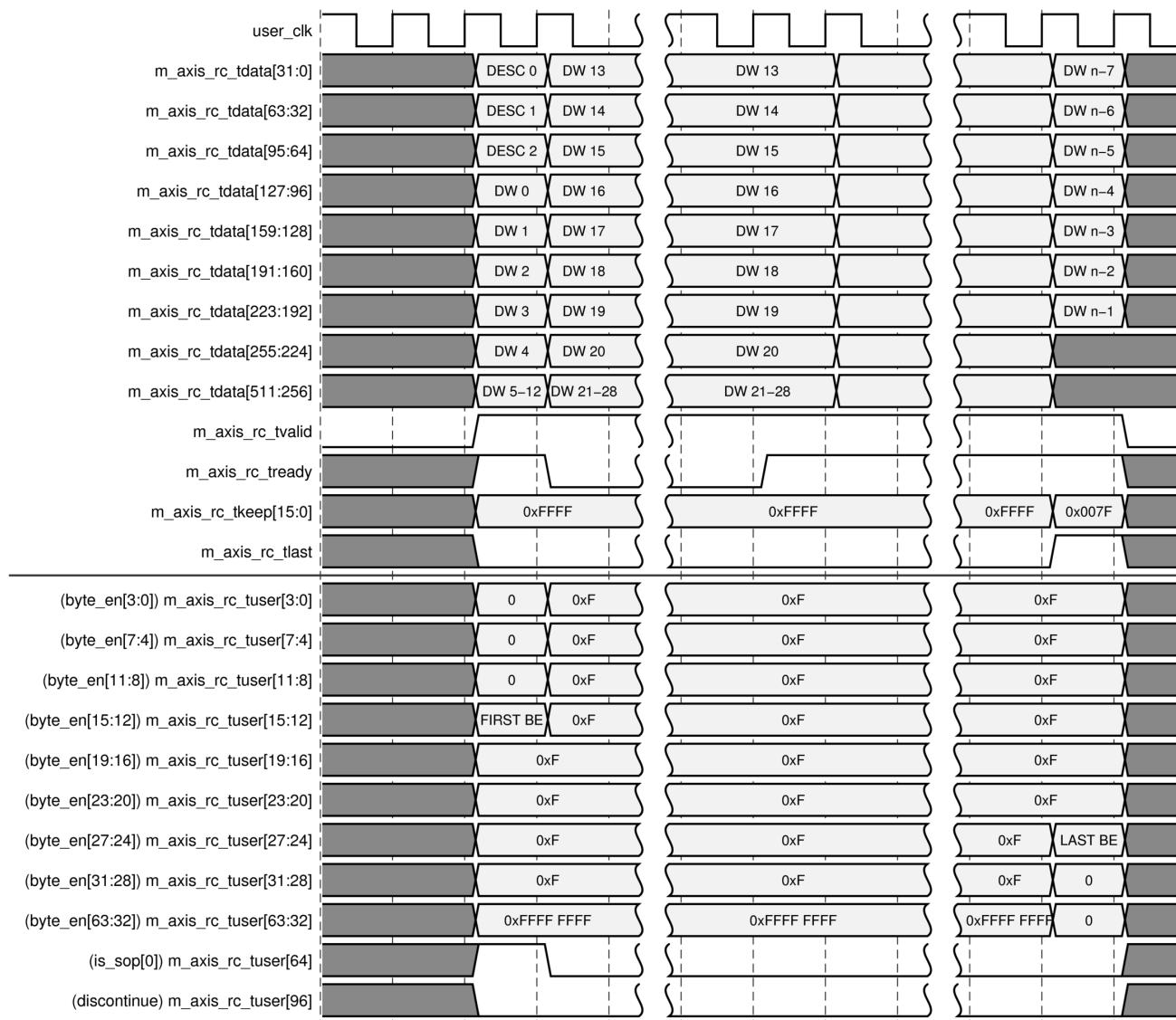
In the Dword-aligned mode, the transfer starts with the three descriptor Dwords, followed immediately by the payload Dwords. The entire TLP, consisting of the descriptor and payload, is transferred as a single AXI4-Stream packet. Data within the payload is always a contiguous stream of bytes when the length of the payload exceeds two Dwords. The positions of the first valid byte within the first Dword of the payload and the last valid byte in the last Dword can then be determined from the Lower Address and Byte Count fields of the Request Completion Descriptor. When the payload size is 2 Dwords or less, the valid bytes in the payload are not be contiguous. In these cases, the user logic must store the First Byte Enable and the Last Byte Enable fields associated with each request sent out on the requester request interface and use them to determine the valid bytes in the completion payload. The user logic are optionally use the byte enable outputs `byte_en[63:0]` within the `m_axi_rc_tuser` bus to determine the valid bytes in the payload, in the cases of both contiguous and non-contiguous payloads.

The core keeps the signal `m_axis_rc_tvalid` asserted over the entire duration of the packet. The user logic can prolong a beat at any time by pulling down `m_axis_rc_tready`. The AXI4-Stream interface signals `m_axis_rc_tkeep` (one per Dword position) indicate the valid Dwors in the packet including the descriptor and any null bytes inserted between the descriptor and the payload. That is, the `tkeep` bits are set to 1 contiguously from the first Dword of the descriptor until the last Dword of the payload. During the transfer of a packet, the `m_axis_rc_tkeep` bits can be 0 only in the last beat of the packet, when the packet does not fill the entire width of the interface. The signal `m_axis_rc_tlast` is always asserted in the last beat of the packet.

The `m_axi_rc_tuser` bus provides several optional signals that can be used to simplify the logic associated with the user side of the interface, or to support additional features. The signal `is_sop[0]` is asserted in the first beat of every packet, when its descriptor is on the bus. When the straddle option is not in use, none of the other sop and eop indications within `m_axi_rc_tuser` are relevant to the transfer of Completions. The byte enable outputs `byte_en[63:0]` (one per byte lane) indicate the valid bytes in the payload. These signals are asserted only when a valid payload byte is in the corresponding lane (it is not asserted for descriptor or null bytes). The asserted byte enable bits are always contiguous from the start of the payload, except when payload size is 2 Dwors or less. For Completion payloads of two Dwors or less, the 1s on `byte_en` are not be contiguous. Another special case is that of a zero-length memory read, when the core transfers a one-Dword payload with the `byte_en` bits all set to 0. Thus, the user logic can, in all cases, use the `byte_en` signals directly to enable the writing of the associated bytes into memory.

The following figure illustrates the Dword-aligned transfer of a Completion TLP received from the link with an associated payload across the requester completion interface. For the purpose of illustration, the size of the data block being written into user memory is assumed to be n Dwors, where $n = k * 16 + 4$, for some $k > 1$. The timing diagrams in this section assume that the Completions are not straddled on the interface. The straddle feature is described in [Straddle Option for RC Interface](#).

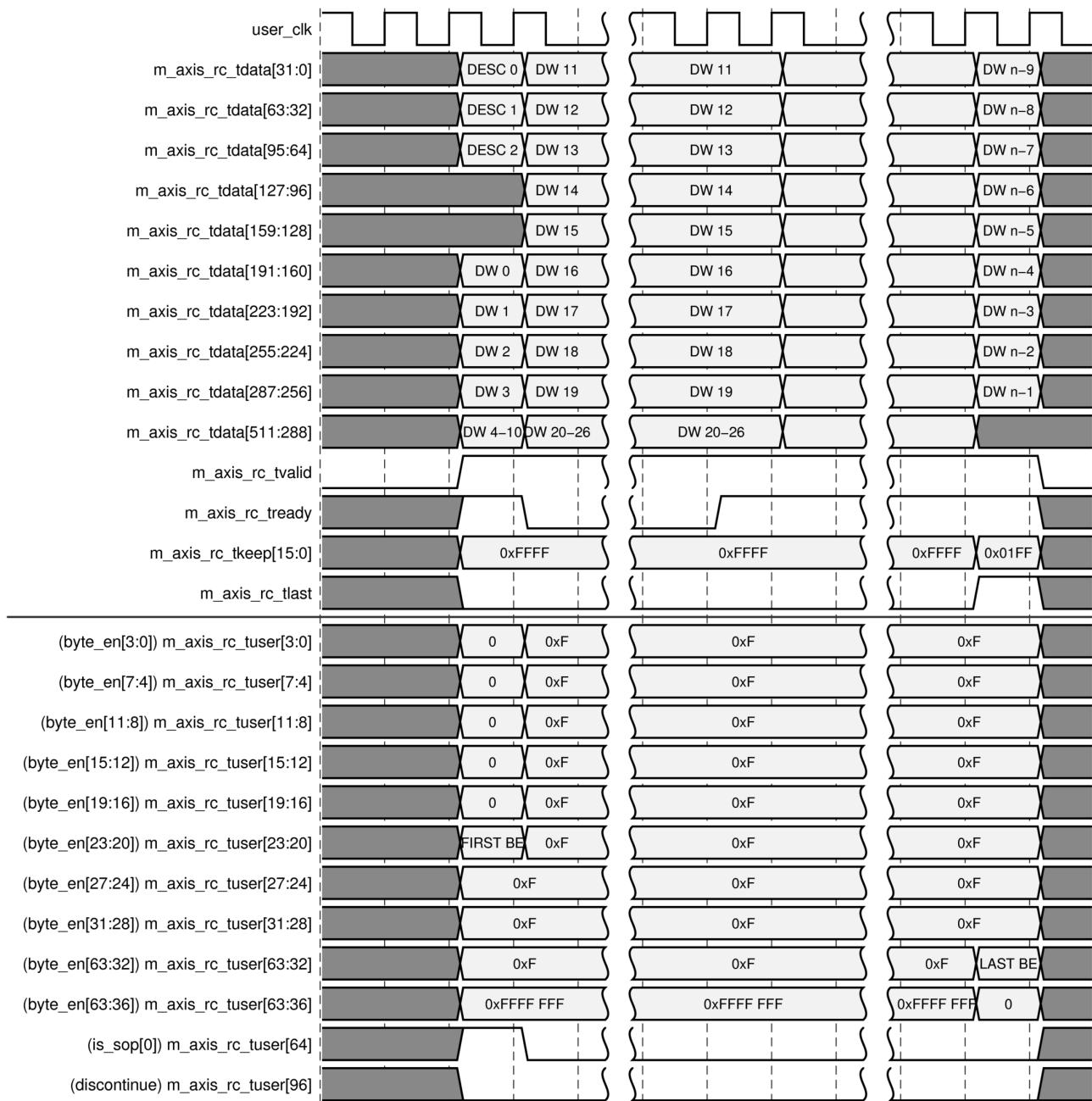
Figure 98: Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode)



The following figure illustrates the address-aligned transfer of a Completion TLP received from the link with an associated payload across the requester completion interface. In the example timing diagrams, the starting Dword address of the data block being transferred (as conveyed in the Lower Address field of the descriptor) is assumed to be $(m * 16 + 1)$, for some integer m . The size of the data block is assumed to be n Dwords, where $n = k * 16 + 4$, for some $k > 0$. The straddle option is not valid for 128-bit address aligned transfers, so the timing diagrams assume that the Completions are not straddled on the interface.

In the 128-bit address aligned mode, the delivery of the payload always starts in the beat following the last byte of the descriptor. The first byte of the payload can appear on any of the bytes lanes 16 - 32, based on the address of the first valid byte of the payload. The `m_axis_rc_tkeep` bits are set to 1 contiguously from the first Dword of the descriptor until the last Dword of the payload. The alignment of the first Dword on the data bus within its 128-bit field is determined by the setting of the `addr_offset[1:0]` input of the requester request interface when the user application sent the request to the core. The user application are optionally use the byte enable outputs `byte_en[63:0]` to determine the valid bytes in the payload.

**Figure 99: Transfer of a Completion with Data on the Requester Completion Interface
(128-bit Interface, Address Aligned Mode)**



Straddle Option for RC Interface

The RC interface of the PCIe® core has the capability to start up to four Completions in the same beat on the requester completion interface. This straddle option is enabled during core customization in the Vivado® IDE. The straddle option can be used only with the Dword-aligned mode.

When the straddle option is enabled, Completion TLPs are transferred on the AXI4-Stream interface as a continuous stream, with no packet boundaries. Thus, the signals `m_axis_rc_tkeep` and `m_axis_rc_tlast` are not useful in determining the boundaries of Completion TLPs delivered on the interface (the core sets `m_axis_rc_tkeep` to all 1s and `m_axis_rc_tlast` to 0 permanently when the straddle option is in use.). Instead, delineation of TLPs is performed using the following signals provided within the `m_axis_rc_tuser` bus.

- `is_sop[3:0]`: The core sets this output to a non-zero value in a beat when there is at least one Completion TLP starting in the beat. When straddle is disabled, only `is_sop[0]` is valid and `is_sop[3:1]` are permanently set to 0. When straddle is enabled, the settings are as follows:
 - 0000: No new TLP starting in this beat
 - 0001: A single new TLP starts in this beat. Its start position is indicated by `is_sop0_ptr[1:0]`.
 - 0011: Two new TLPs are starting in this beat. `is_sop0_ptr[1:0]` provides the starting position of the first TLP and `is_sop1_ptr[1:0]` provides the starting position of the second TLP.
 - 0111: Three new TLPs are starting in this beat. `is_sop0_ptr[1:0]` provides the starting position of the first TLP, `is_sop1_ptr[1:0]` provides the starting position of the second TLP, and `is_sop2_ptr[1:0]` provides the starting position of the third TLP.
 - 1111: Four new TLPs are starting in this beat. `is_sop0_ptr[1:0]` provides the starting position of the first TLP, `is_sop1_ptr[1:0]` provides the starting position of the second TLP, `is_sop2_ptr[1:0]` provides the starting position of the third TLP, and `is_sop3_ptr[1:0]` provides the starting position of the fourth TLP.
 - All other settings are reserved.
- `is_sop0_ptr[1:0]`: When `is_sop[0]` is set, this field indicates the offset of the first Completion TLP starting in the current beat. Valid settings are `2'b00` (TLP starting at Dword 0), and `2'b01` (TLP starting at Dword 4), `2'b10` (TLP starting at Dword 8), and `2'b11` (TLP starting at Dword 12).
- `is_sop1_ptr[1:0]`: When `is_sop[1]` is set, this field indicates the offset of the second Completion TLP starting in the current beat. Valid settings are `2'b01` (TLP starting at Dword 4), `2'b10` (TLP starting at Dword 8), and `2'b11` (TLP starting at Dword 12).
- `is_sop2_ptr[1:0]`: When `is_sop[2]` is set, this field indicates the offset of the third Completion TLP starting in the current beat. Valid settings are `2'b10` (TLP starting at Dword 8), and `2'b11` (TLP starting at Dword 12).
- `is_sop3_ptr[1:0]`: When `is_sop[3]` is set, this field indicates the offset of the fourth Completion TLP starting in the current beat. Its only valid setting is `2'b11` (TLP starting at Dword 12).

- `is_eop[3:0]`: These outputs signals that one or more TLPs are ending in this beat. These outputs are set in the final beat of a TLP. When straddle is disabled, only `is_eop[0]` is valid and `is_eop[3:1]` are permanently set to 0. When straddle is enabled, the settings are as follows:
 - 0000: No TLPs are ending in this beat.
 - 0001: A single TLP is ending in this beat. The setting of `is_eop0_ptr[3:0]` provides the offset of the last Dword of this TLP.
 - 0011: Two TLPs are ending in this beat. `is_eop0_ptr[3:0]` provides the offset of the last Dword of the first TLP and `is_eop1_ptr[3:0]` provides the offset of the last Dword of the second TLP.
 - 0111: Three TLPs are ending in this beat. `is_eop0_ptr[3:0]` provides the offset of the last Dword of the first TLP, `is_eop1_ptr[3:0]` provides the offset of the last Dword of the second TLP, and `is_eop2_ptr[3:0]` provides the offset of the last Dword of the third TLP.
 - 1111: Four TLPs are ending in this beat. `is_eop0_ptr[3:0]` provides the offset of the last Dword of the first TLP, `is_eop1_ptr[3:0]` provides the offset of the last Dword of the second TLP, `is_eop2_ptr[3:0]` provides the offset of the last Dword of the third TLP, and `is_eop3_ptr[3:0]` provides the offset of the last Dword of the fourth TLP.
 - All other settings are reserved.
- `is_eop0_ptr[3:0]`: When `is_eop[0]` is set, this field provides the offset of the last Dword of the first TLP ending in this beat. It can take any value from 0 through 15. The offset for the last byte can be determined from the starting address and length of the TLP, or from the byte enable signals `byte_en[63:0]`.
- `is_eop1_ptr[3:0]`: When `is_eop[1]` is set, this field provides the offset of the last Dword of the second TLP ending in this beat. It can take any value from 6 through 15.
- `is_eop2_ptr[3:0]`: When `is_eop[2]` is set, this field provides the offset of the last Dword of the third TLP ending in this beat. It can take any value from 10 through 15.
- `is_eop3_ptr[3:0]`: When `is_eop[3]` is set, this field provides the offset of the last Dword of the fourth TLP ending in this beat. It can take values of 14 or 15.

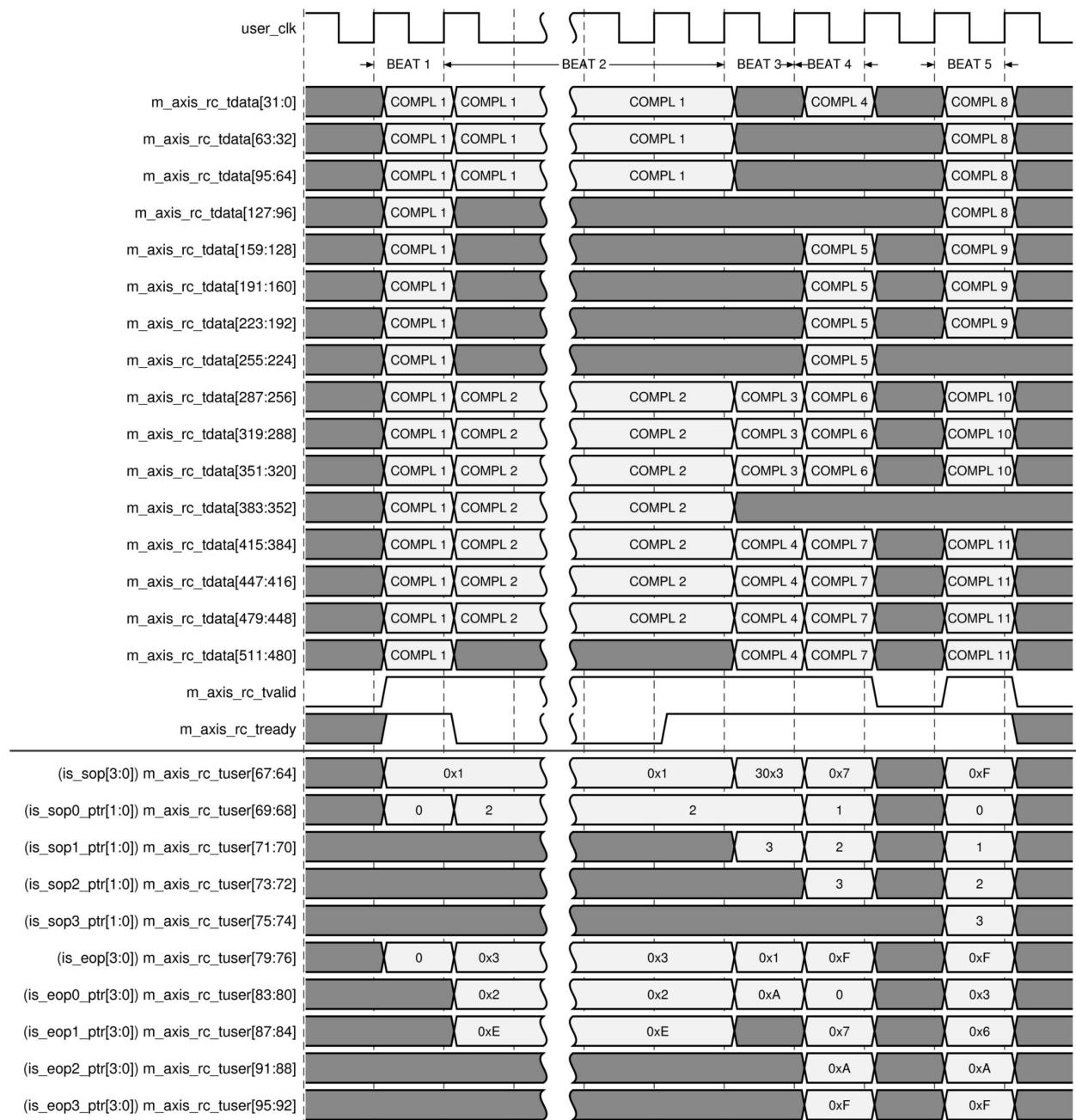
The following figure illustrates the transfer of 11 Completion TLPs on the requester completion interface when the straddle option is enabled. The first Completion TLP (COMPL 1) starts at Dword position 0 of Beat 1 and ends in Dword position 2 of Beat 2. The second TLP (COMPL 2) starts in Dword position 8 of the same beat and ends in Dword position 14. Thus, there is one TLP starting in Beat 1, whose starting position is indicated by `is_sop0_ptr`, and two TLPs ending, whose ending Dword positions are indicated by `is_eop0_ptr` and `is_eop1_ptr`, respectively.

Beat 3 has COMPL 3 starting at Dword offset 8, ending at Dword offset10. There is also a second TLP (CMPL 4) in the same beat, starting at Dword offset 12 and continuing to the next beat. In this beat, `is_sop0_ptr` points to the starting Dword offset of COMPL 3 and `is_sop1_ptr` points to the starting Dword offset of COMPL 4. `is_eop0_ptr` points to the offset of the last Dword offset of COMPL 4.

Beat 4 has COMPL 4 ending with Dword offset 0, and has three new complete TLPs in it (COMPL 5, 6 and 7). The starting Dword offsets of the new Completions 5, 6 and 7 are provided by `is_sop0_ptr`, `is_sop1_ptr`, and `is_sop2_ptr`, respectively. The ending offsets of Completions 4, 5, 6 and 7 are indicated by `is_eop0_ptr`, `is_eop1_ptr`, `is_eop2_ptr` and `is_eop3_ptr`, respectively.

Finally, Beat 5 contains four complete TLPs (COMPL 8 – 11). Their starting Dword offsets are signaled by `is_sop0_ptr`, `is_sop1_ptr`, `is_sop2_ptr` and `is_sop3_ptr`, respectively. The ending offsets are indicated by `is_eop0_ptr`, `is_eop1_ptr`, `is_eop2_ptr` and `is_eop3_ptr`, respectively. Thus, all the four SOP and EOP pointers provide valid information in this beat.

Figure 100: Transfer of Completion TLPs on the Requester Completion Interface with the Straddle Option Enabled



Aborting a Completion Transfer

For any Completion that includes an associated payload, the core signals an error in the transferred payload by asserting the discontinue signal in the `m_axis_rc_tuser` bus in the last beat of the packet. This occurs when the core has detected an uncorrectable error while reading data from its internal memories. The user application must discard the entire packet when it has detected the signal `discontinue` asserted in the last beat of a packet.

When the straddle option is in use, the core does not start a new Completion TLP in the same beat when it has asserted `discontinue` to abort the Completion TLP ending in the beat.

Handling of Completion Errors

When a Completion TLP is received from the link, the core matches it against the outstanding requests in the Split Completion Table to determine the corresponding request, and compares the fields in its header against the expected values to detect any error conditions. The core then signals the error conditions in a 4-bit error code sent to the user logic as part of the completion descriptor. The core also indicates the last completion for a request by setting the Request Completed bit (bit 30) in the descriptor. The error conditions signaled by the various error codes are described below:

- 0010: Request terminated by a Completion TLP with UR, CA or CRS status. In this case, there is no data associated with the completion, and the Request Completed bit in the completion descriptor is set. On receiving such a Completion from the core, the user logic can discard the corresponding request.
- 0011: Read Request terminated by a Completion TLP with incorrect byte count. This condition occurs when a Completion TLP is received with a byte count not matching the expected count. The Request Completed bit in the completion descriptor is set. On receiving such a completion from the core, the user logic can discard the corresponding request.
- 0100: This code indicates the case when the current Completion being delivered has the same tag of an outstanding request, but its Requester ID, TC, or Attr fields did not match with the parameters of the outstanding request. The user logic should discard any data that follows the descriptor. In addition, if the Request Completed bit in the descriptor is not set, the user logic should continue to discard the data subsequent completions for this tag until it receives a completion descriptor with the Request Completed bit set. On receiving a completion descriptor with the Request Completed bit set, the user logic can remove all state associated with the request.
- 0101: Error in starting address. The low address bits in the Completion TLP header did not match with the starting address of the next expected byte for the request. The user logic should discard any data that follows the descriptor. In addition, if the Request Completed bit in the descriptor is not set, the user logic should continue to discard the data subsequent Completions for this tag until it receives a completion descriptor with the Request Completed bit set. On receiving a completion descriptor with the Request Completed bit set, the user logic can discard the corresponding request.

- 0110: Invalid tag. This error code indicates that the tag in the Completion TLP did not match with the tags of any outstanding request. The user logic should discard any data following the descriptor.
- 0111: Invalid byte count. The byte count in the Completion was higher than the total number of bytes expected for the request. In this case, the Request Completed bit in the completion descriptor is also set. On receiving such a completion from the core, the user logic can discard the corresponding request.
- 1000: Request terminated by a Completion timeout. This error code is used when an outstanding request times out without receiving a Completion from the link. The core maintains a completion timer for each outstanding request, and responds to a completion timeout by transmitting a dummy completion descriptor on the requester completion interface to the user logic, so that the user logic can terminate the pending request, or retry the request. Because this descriptor does not correspond to a Completion TLP received from the link, only the Request Completed bit (bit 30), the tag field (bits 71:64) and the requester Function field (bits [55:48]) are valid in this descriptor.
- 1000: Request terminated by a Function-Level Reset (FLR) targeting the Function that generated the request. In this case, the core transmits a dummy completion descriptor on the requester completion interface to the user logic, so that the user logic can terminate the pending request. Because this descriptor does not correspond to a Completion TLP received from the link, only the Request Completed bit (bit 30), the tag field (bits 71:64) and the requester Function field (bits [55:48]) are valid in this descriptor.

When the tags are managed internally by the core, logic within the core ensures that a tag allocated to a pending request is not reused until either all the Completions for the request were received or the request was timed out. When tags are managed by the user logic, however, the user logic must ensure that a tag assigned to a request is not reused until the core has signaled the termination of the request by setting the Request Completed bit in the completion descriptor. The user logic can close out a pending request on receiving a completion with a non-zero error code, but should not free the associated tag if the Request Completed bit in the completion descriptor is not set. Such a situation might occur when a request receives multiple split completions, one of which has an error. In this case the core can continue to receive Completion TLPs for the pending request even after the error was detected, and these Completions would be incorrectly matched to a different request if its tag was reassigned too soon. Note that, in some cases, the core might need to wait for the request to time out even when a split completion was received with an error, before it can allow the tag to be reused.

Note: Each parity bit corresponds to parity of one byte in AXIS tdata. There are 64 bit parity bits corresponding to the 512 bit AXI tdata (and 32 bit parity bits corresponding to 256 bit AXI tdata). The received parity bits, on m_axis_cq_tuser and m_axis_rc_tuser signals, are valid for the following::

- Descriptor bytes in AXIS tdata.
- The valid payload byte in AXIS tdata indicated by the byte_en field in the AXIS tuser. For example, if byte_en[63:0] = 0x0000_0000_0000_FFFF, then only lower 16 parity bits are valid. If byte_en[63:0] = 0xFFFF_FFFF_FFFF_FFFF, then all 64 parity bits are enabled.

Power Management

The core supports these power management modes:

- Active State Power Management (ASPM)
- Programmed Power Management (PPM)

Implementing these power management functions as part of the PCI Express design enables the PCI Express® hierarchy to seamlessly exchange power-management messages to save system power. All power management message identification functions are implemented. The subsections in this section describe the user logic definition to support the above modes of power management.

For additional information on ASPM and PPM implementation, see the [PCI Express Base Specification](#).

Active State Power Management

The core advertises an N_FTS value of 255 to ensure proper alignment when exiting L0s. If the N_FTS value is modified, you must ensure enough FTS sequences are received to properly align and avoid transition into the Recovery state.

The Active State Power Management (ASPM) functionality is autonomous and transparent from a user-logic function perspective. The core supports the conditions required for ASPM. The integrated block supports ASPM L0s and ASPM L1. L0s and L1 should not be enabled in parallel.

Note: ASPM is not supported in non-synchronous clocking mode.

Note: L0s is not supported for Gen3 capable designs. It is supported only on designs generated for Gen1 and Gen2.

Programmed Power Management

To achieve considerable power savings on the PCI Express® hierarchy tree, the core supports these link states of Programmed Power Management (PPM):

- L0: Active State (data exchange state)
- L1: Higher Latency, lower power standby state
- L3: Link Off State

The Programmed Power Management Protocol is initiated by the Downstream Component/Upstream Port.

PPM L0 State

The L0 state represents normal operation and is transparent to the user logic. The core reaches the L0 (active state) after a successful initialization and training of the PCI Express® Link(s) as per the protocol.

PPM L1 State

These steps outline the transition of the core to the PPM L1 state:

1. The transition to a lower power PPM L1 state is always initiated by an upstream device, by programming the PCI Express® device power state to D3-hot (or to D1 or D2, if they are supported).
2. The device power state is communicated to the user logic through the `cfg_function_power_state` output.
3. The core then throttles/stalls the user logic from initiating any new transactions on the user interface by deasserting `s_axis_rq_tready`. Any pending transactions on the user interface are, however, accepted fully and can be completed later.
 - The core is configured as an Endpoint and the User Configuration Space is enabled. In this situation, the user application must refrain from sending new Request TLPs if `cfg_function_power_state` indicates non-D0, but the user application can return Completions to Configuration transactions targeting User Configuration space.
 - The core is configured as a Root Port. To be compliant in this situation, the user application should refrain from sending new Requests if `cfg_function_power_state` indicates non-D0.
4. The core exchanges appropriate power management DLLPs with its link partner to successfully transition the link to a lower power PPM L1 state. This action is transparent to the user logic.
5. All user transactions are stalled for the duration of time when the device power state is non-D0, with the exceptions indicated in step 3.

PPM L3 State

These steps outline the transition of the Endpoint for PCI Express® to the PPM L3 state:

1. The core negotiates a transition to the L23 Ready Link State upon receiving a `PME_Turn_Off` message from the upstream link partner.
2. Upon receiving a `PME_Turn_Off` message, the core initiates a handshake with the user logic through `cfg_power_state_change_interrupt` (as shown in the following table) and expects a `cfg_power_state_change_ack` back from the user logic.
3. A successful handshake results in a transmission of the Power Management Turn-off Acknowledge (`PME_turnoff_ack`) Message by the core to its upstream link partner.

4. The core closes all its interfaces, disables the Physical/Data-Link/Transaction layers and is ready for removal of power to the core.

There are two exceptions to this rule:

- The core is configured as an Endpoint and the User Configuration Space is enabled. In this situation, the user application must refrain from sending new Request TLPs if `cfg_function_power_state` indicates non-D0, but the user application can return Completions to Configuration transactions targeting User Configuration space.
- The core is configured as a Root Port. To be compliant in this situation, the user application should refrain from sending new Requests if `cfg_function_power_state` indicates non-D0.

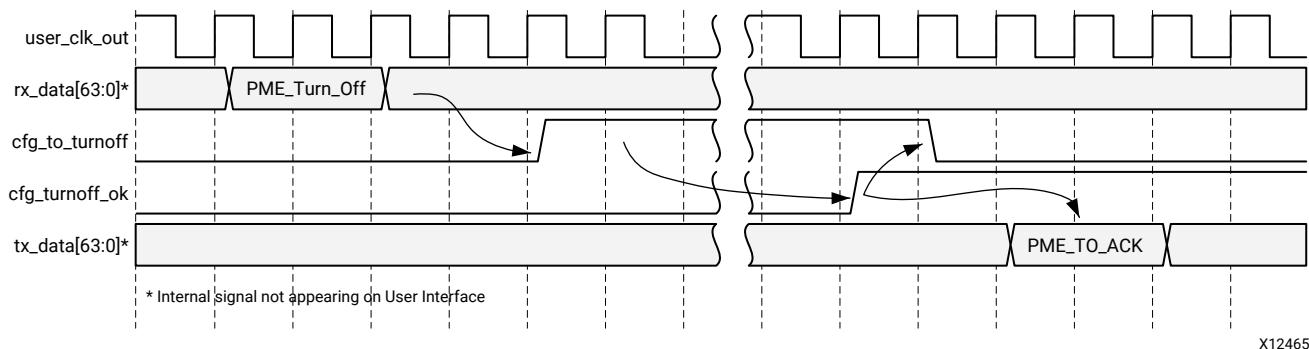
Table 62: Power Management Handshaking Signals

Port Name	Direction	Description
<code>cfg_power_state_change_interrupt</code>	Output	Asserted if a power-down request TLP is received from the upstream device. After assertion, <code>cfg_power_state_change_interrupt</code> remains asserted until the user application asserts <code>cfg_power_state_change_ack</code> .
<code>cfg_power_state_change_ack</code>	Input	Asserted by the user application when it is safe to power down.

Power-down negotiation follows these steps:

1. Before power and clock are turned off, the Root Complex or the Hot-Plug controller in a downstream switch issues a `PME_Turn_Off` broadcast message.
2. When the core receives this TLP, it asserts `cfg_power_state_change_interrupt` to the user application and starts polling the `cfg_power_state_change_ack` input.
3. When the user application detects the assertion of `cfg_to_turnoff`, it must complete any packet in progress and stop generating any new packets. After the user application is ready to be turned off, it asserts `cfg_power_state_change_ack` to the core. After assertion of `cfg_power_state_change_ack`, the user application is committed to being turned off.
4. The core sends a `PME_TO_Ack` message when it detects assertion of `cfg_power_state_change_ack`.

Figure 101: Power Management Handshaking: 64-Bit



Generating Interrupt Requests

See the `cfg_interrupt_msi*` and `cfg_interrupt_msix_*` descriptions in the tables in [Configuration Interrupt Controller Interface](#).

Note: This section only applies to the Endpoint Configuration of the Integrated Block for PCIe core.

The core supports sending interrupt requests as either legacy, Message MSI, or MSI-X interrupts. The mode is programmed using the MSI Enable bit in the Message Control register of the MSI Capability Structure and the MSI-X Enable bit in the MSI-X Message Control register of the MSI-X Capability Structure.

The state of the MSI Enable and MSI-X Enabled bits is reflected by the `cfg_interrupt_msi_enable` and `cfg_interrupt_msix_enable` outputs, respectively. The following table describes the Interrupt Mode to which the device has been programmed, based on the `cfg_interrupt_msi_enable` and `cfg_interrupt_msix_enable` outputs of the core.

Table 63: Interrupt Modes

	<code>cfg_interrupt_msixenable=0</code>	<code>cfg_interrupt_msixenable=1</code>
<code>cfg_interrupt_msi_enable = 0</code>	Legacy Interrupt (INTx) mode. The <code>cfg_interrupt</code> interface only sends INTx messages.	MSI-X mode. MSI-X interrupts can be generated using the <code>cfg_interrupt</code> interface.
<code>cfg_interrupt_msi_enable = 1</code>	MSI mode. The <code>cfg_interrupt</code> interface only sends MSI interrupts (MW _r TLPs).	Undefined. System software is not supposed to permit this. However, the <code>cfg_interrupt</code> interface is active and sends MSI interrupts (MW _r TLPs) if you choose to do so.

The MSI Enable bit in the MSI control register, the MSI-X Enable bit in the MSI-X Control register, and the Interrupt Disable bit in the PCI Command register are programmed by the Root Complex. The user application has no direct control over these bits.

The Internal Interrupt Controller in the core only generates Legacy Interrupts and MSI Interrupts. MSI-X Interrupts need to be generated by the user application and presented on the transmit AXI4-Stream interface. The status of `cfg_interrupt_msi_enable` determines the type of interrupt generated by the internal Interrupt Controller:

If the MSI Enable bit is set to a 1, then the core generates MSI requests by sending Memory Write TLPs. If the MSI Enable bit is set to 0, the core generates legacy interrupt messages as long as the Interrupt Disable bit in the PCI Command register is set to 0.

- `cfg_interrupt_msi_enable = 0`: Legacy interrupt
- `cfg_interrupt_msi_enable = 1`: MSI
- Command register bit 10 = 0: INTx interrupts enabled
- Command register bit 10 = 1: INTx interrupts disabled (requests are blocked by the core)

The user application can monitor `cfg_function_status` to check whether INTx interrupts are enabled or disabled. For more information, see [Configuration Status Interface](#).

The user application requests interrupt service in one of two ways, each of which is described in the following section.

Legacy Interrupt Mode

- The user application first asserts `cfg_interrupt_int` and `cfg_interrupt_pending` to assert the interrupt.
- The core then asserts `cfg_interrupt_sent` to indicate the interrupt is accepted. If the Interrupt Disable bit in the PCI Command register is set to 0, the core sends an assert interrupt message (Assert_INTA). After the interrupt has been serviced, the user application deasserts `cfg_interrupt_int`.
- After the user application deasserts `cfg_interrupt_int`, the core sends a deassert interrupt message (Deassert_INTA). This is indicated by the assertion of `cfg_interrupt_sent` a second time.

`cfg_interrupt_int` must be asserted until the user application receives confirmation of the assert interrupt message (Assert_INTA), which is indicated by the assertion of `cfg_interrupt_sent`, and the interrupt has been serviced/cleared by the Root's Interrupt Service Routine (ISR). Deasserting `cfg_interrupt_int` causes the core to send the deassert interrupt message (Deassert_INTA). `cfg_interrupt_pending` must be asserted along with the assertion of `cfg_interrupt_int` until the interrupt has been serviced, otherwise, the

interrupt status bit in the status register is not updated correctly. `cfg_interrupt_pending` can be deasserted along with the deassertion of `cfg_interrupt_int` after the first assertion of `cfg_interrupt_sent`. When the software/Root's ISR receives an assert interrupt message, it reads this interrupt status bit to determine whether there is an interrupt pending for this function.

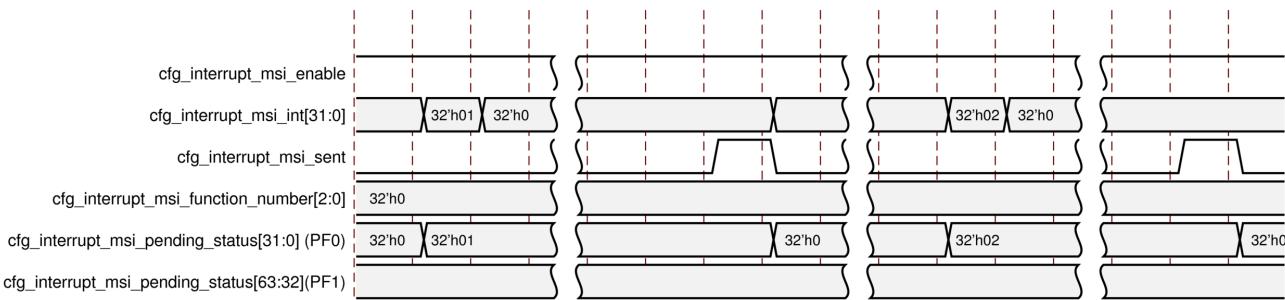
Figure 102: Legacy Interrupt Signaling



MSI Mode

The user application first asserts a value on `cfg_interrupt_msi_int`, as shown in the previous figure. The core asserts `cfg_interrupt_msi_sent` to indicate that the interrupt is accepted and the core sends an MSI Memory Write TLP.

Figure 103: MSI Mode



The MSI request is either a 32-bit addressable Memory Write TLP or a 64-bit addressable Memory Write TLP. The address is taken from the Message Address and Message Upper Address fields of the MSI Capability Structure, while the payload is taken from the Message Data field. These values are programmed by system software through configuration writes to the MSI Capability structure. When the core is configured for Multi-Vector MSI, system software can permit Multi-Vector MSI messages by programming a non-zero value to the Multiple Message Enable field.

The type of MSI TLP sent (32-bit addressable or 64-bit addressable) depends on the value of the Upper Address field in the MSI capability structure. By default, MSI messages are sent as 32-bit addressable Memory Write TLPs. MSI messages use 64-bit addressable Memory Write TLPs only if the system software programs a non-zero value into the Upper Address register.

When Multi-Vector MSI messages are enabled, the user application can override one or more of the lower-order bits in the Message Data field of each transmitted MSI TLP to differentiate between the various MSI messages sent upstream. The number of lower-order bits in the Message Data field available to the user application is determined by the lesser of the value of the Multiple Message Capable field, as set in the IP catalog, and the Multiple Message Enable field, as set by system software and available as the `cfg_interrupt_msi_mmenable[2:0]` core output. The core masks any bits in `cfg_interrupt_msi_select` which are not configured by system software through Multiple Message Enable.

This pseudo code shows the processing required:

```
// Value MSI_Vector_Num must be in range: 0 ≤ MSI_Vector_Num ≤
(2^cfg_interrupt_mmenable)-1

if (cfg_interrupt_msienable) {           // MSI Enabled
    if (cfg_interrupt_mmenable > 0) {     // Multi-Vector MSI Enabled
        cfg_interrupt_msi_int[MSI_Vector_Num] = 1;
    } else {                           // Single-Vector MSI Enabled
        cfg_interrupt_msi_int[MSI_Vector_Num] = 0;
    }
} else {
    // Legacy Interrupts Enabled
}
```

For example:

1. If `cfg_interrupt_mmenable[2:0] == 000b`, that is, 1 MSI Vector Enabled, then
`cfg_interrupt_msi_int = 01h;`
2. If `cfg_interrupt_mmenable[2:0] == 101b`, that is, 32 MSI Vectors Enabled, then
`cfg_interrupt_msi_int = {32'b1 << {MSI_Vector#}};`

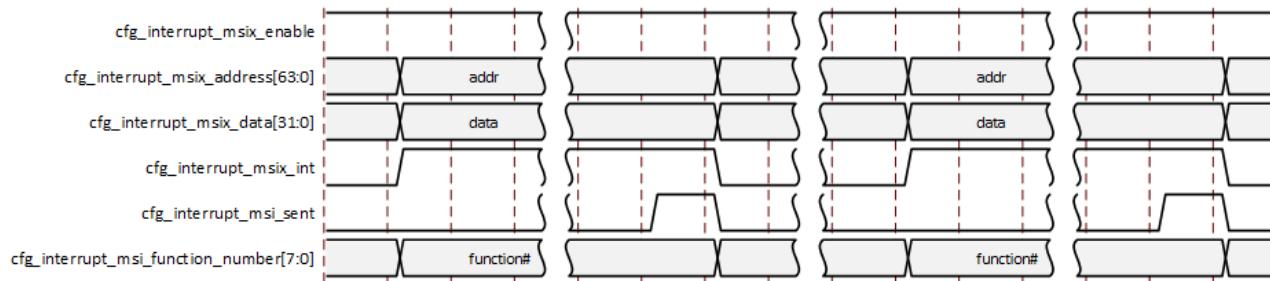
where `MSI_Vector#` is a 5-bit value and is allowed to be `00000b ≤ MSI_Vector# ≤ 11111b`.

If Per-Vector Masking is enabled, first verify that the vector being signaled is not masked in the Mask register. This is done by reading this register on the Configuration interface (the core does not look at the Mask register).

MSI-X Mode

The core supports the MSI-X interrupt and its signaling, which is shown in the following figure. The MSI-X vector table and the MSI-X Pending Bit Array need to be implemented as part of the user logic, by claiming a BAR aperture if the built-in MSI-X vector tables are not used.

Figure 104: MSI-X Mode

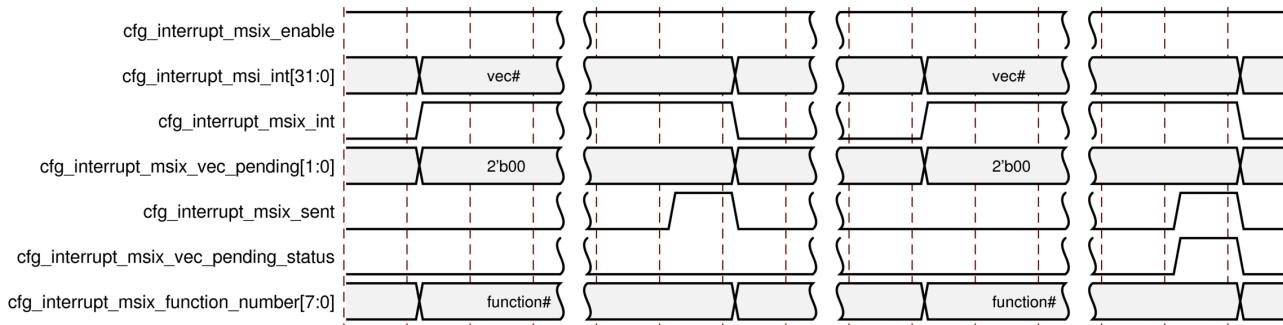


MSI-X Mode with Built-in MSI-X Vector Tables

The core optionally supports built-in MSI-X vector tables including the Pending Bit Array.

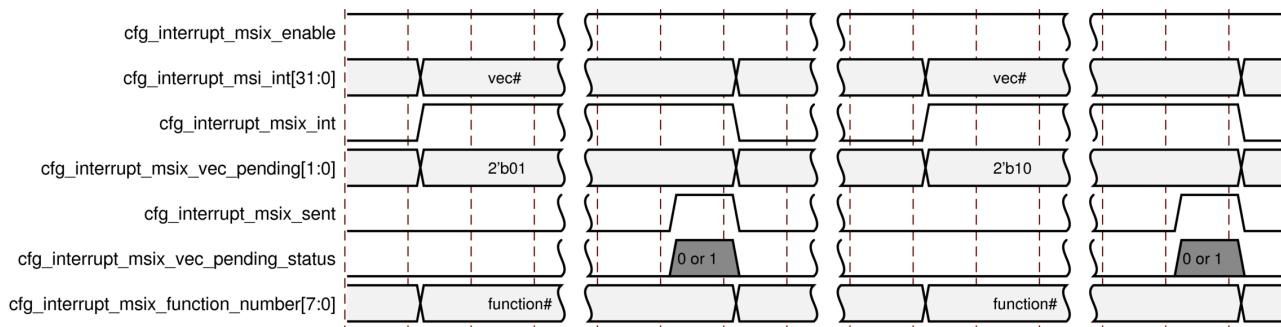
- As shown in the following figure, the user application first asserts `cfg_interrupt_msix_int` with the vector number set in `cfg_interrupt_msix_int`.
- The core asserts `cfg_interrupt_msix_sent` to signal that the interrupt is accepted. If `cfg_interrupt_msix_vec_pending_status` is clear, the core sends a MSI-X Memory Write TLP. Otherwise, the core waits to send a MSI-X Memory Write TLP until the function mask is cleared.

Figure 105: MSI-X Signaling with Built-In MSI-X Vector Tables



- Instead of generating an interrupt, the user application can query or clear the Pending Bit Array by additionally setting `cfg_interrupt_msix_vec_pending` to 2'b01 or 2'b10 respectively, as shown in the following figure.
- In the query and clear cases, `cfg_interrupt_msix_vec_pending_status` reflects the pending status before the query or clear.
- `cfg_interrupt_msi_int[31:0]` is a shared signal between MSI [31:0] and MSI-X [7:0].

Figure 106: MSI-X Pending Bit Array Query and Clear



Note: Applications that need to generate MSI/MSI-X interrupts with traffic class bits not equal to 0 or address translation bits not equal to 0 must use the RQ interface to generate the interrupt (memory write descriptor).

Receive Message Interface

The core provides a separate receive-message interface which the user application can use to receive indications of messages received from the link. When the receive message interface is enabled, the integrated block signals the arrival of a message from the link by setting the `cfg_msg_received_type[4:0]` output to indicate the type of message (see the following table) and pulsing the `cfg_msg_received` signal for one or more cycles. The duration of assertion of `cfg_msg_received` is determined by the type of message received (see [Table 64: Message Type Encoding on Receive Message Interface](#)). When `cfg_msg_received` is active-High, the integrated block transfers any parameters associated with the message on the bus 8 bits at a time on the bus `cfg_msg_received_data`. The parameters transferred on this bus in each cycle of `cfg_msg_received` assertion for various message types are listed in the [Table 65: Message Parameters on Receive Message Interface](#) table. For Vendor-Defined Messages, the integrated block transfers only the first Dword of any associated payload across this interface. When larger payloads are in use, the completer request interface should be used for the delivery of messages.

Table 64: Message Type Encoding on Receive Message Interface

<code>cfg_msg_received_type[4:0]</code>	Message Type
0	ERR_COR
1	ERR_NONFATAL
2	ERR_FATAL
3	Assert_INTA
4	Deassert_INTA
5	Assert_INTB

Table 64: Message Type Encoding on Receive Message Interface (cont'd)

cfg_msg_received_type[4:0]	Message Type
6	Deassert_INTB
7	Assert_INTC
8	Deassert_INTC
9	Assert_INTD
10	Deassert_INTD
11	PM_PME
12	PME_TO_Ack
13	PME_Turn_Off
14	PM_Active_State_Nak
15	Set_Slot_Power_Limit
16	Latency Tolerance Reporting (LTR)
17	Optimized Buffer Flush/Fill (OBFF)
18	Unlock
19	Vendor Defined Type 0
20	Vendor Defined Type 1
21	ATS Invalid Request
22	ATS Invalid Completion
23	ATS Page Request
24	ATS PRG Response
25 – 31	Reserved

Table 65: Message Parameters on Receive Message Interface

Message Type	Number of Cycles of cfg_msg_received Assertion	Parameter Transferred on cfg_msg_received_data[7:0]
ERR_COR, ERR_NONFATAL, ERR_FATAL	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
Assert_INTx, Deassert_INTx	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
PM_PME, PME_TO_Ack, PME_Turn_off, PM_Active_State_Nak	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
Set_Slot_Power_Limit	6	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number Cycle 3: bits [7:0] of payload Cycle 4: bits [15:8] of payload Cycle 5: bits [23:16] of payload Cycle 6: bits [31:24] of payload

Table 65: Message Parameters on Receive Message Interface (cont'd)

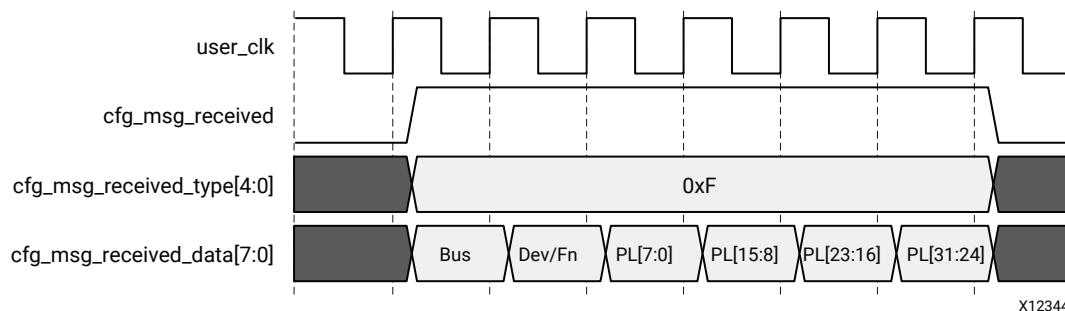
Message Type	Number of Cycles of cfg_msg_received Assertion	Parameter Transferred on cfg_msg_received_data[7:0]
Latency Tolerance Reporting (LTR)	6	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number Cycle 3: bits [7:0] of Snoop Latency Cycle 4: bits [15:8] of Snoop Latency Cycle 5: bits [7:0] of No-Snoop Latency Cycle 6: bits [15:8] of No-Snoop Latency
Optimized Buffer Flush/Fill (OBFF)	3	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number Cycle 3: OBFF Code
Unlock	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
Vendor_Defined Type 0	4 cycles when no data present, 8 cycles when data present.	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number Cycle 3: Vendor ID[7:0] Cycle 4: Vendor ID[15:8] Cycle 5: bits [7:0] of payload Cycle 6: bits [15:8] of payload Cycle 7: bits [23:16] of payload Cycle 8: bits [31:24] of payload
Vendor_Defined Type 1	4 cycles when no data present, 8 cycles when data present.	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number Cycle 3: Vendor ID[7:0] Cycle 4: Vendor ID[15:8] Cycle 5: bits [7:0] of payload Cycle 6: bits [15:8] of payload Cycle 7: bits [23:16] of payload Cycle 8: bits [31:24] of payload
ATS Invalid Request	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
ATS Invalid Completion	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
ATS Page Request	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
ATS PRG Response	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number

The following timing diagram showing the example of a Set_Slot_Power_Limit message on the receive message interface. This message has an associated one-Dword payload. For this message, the parameters are transferred over six consecutive cycles. The following information appears on the cfg_msg_received_data bus in each cycle:

- Cycle 1: Bus number of Requester ID
- Cycle 2: Device/Function Number of Requester ID

- Cycle 3: Bits [7:0] of the payload Dword
- Cycle 4: Bits [15:8] of the payload Dword
- Cycle 5: Bits [23:16] of the payload Dword
- Cycle 6: Bits [31:24] of the payload Dword

Figure 107: Receive Message Interface



The integrated block inserts a gap of at least one clock cycle between successive pulses on the `cfg_msg_received` output. There is no mechanism to apply back pressure on the message indications delivered through the receive message interface. When using this interface, the user logic must always be ready to receive message indications.

Configuration Management Interface

The ports used by configuration registers are described in [Configuration Status Interface](#). Root Ports must use the Configuration Port to set up the Configuration Space. Endpoints can also use the Configuration Port to read and write; however, care must be taken to avoid adverse system side effects.

The user application must supply the address as a Dword address, not a byte address.



TIP: To calculate the Dword address for a register, divide the byte address by four.

For example:

For the Command/Status register in the PCI Configuration Space Header:

- The Dword address of is 01h.

Note: The byte address is 04h.

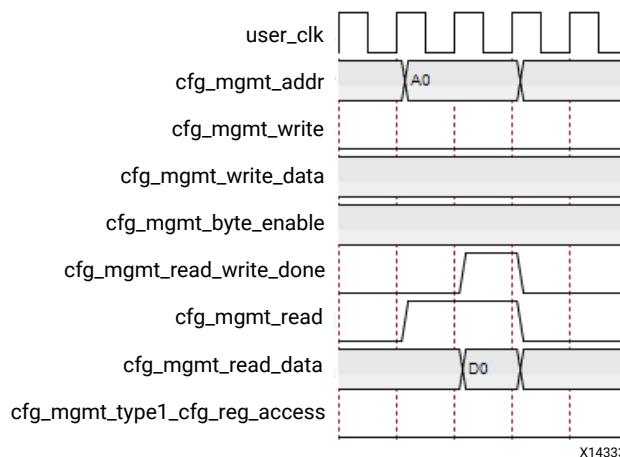
For BAR0:

- The Dword address is 04h.

Note: The byte address is 10h.

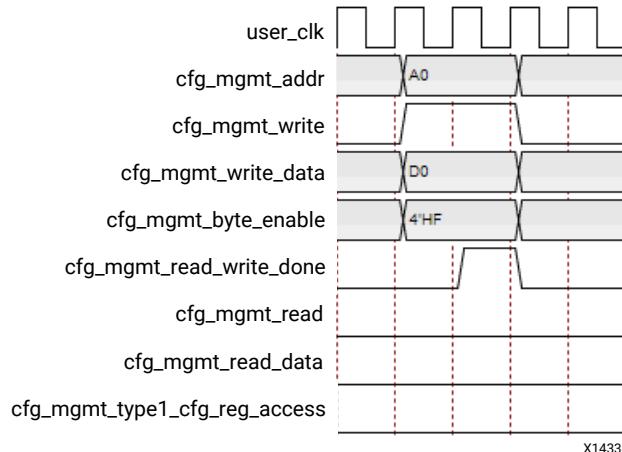
To read any register in configuration space, the user application drives the register Dword address onto `cfg_mgmt_addr[9:0]`. `cfg_mgmt_function_number[7:0]` selects the PCI Function associated with the configuration register. The core drives the content of the addressed register onto `cfg_mgmt_read_data[31:0]`. The value on `cfg_mgmt_read_data[31:0]` is qualified by signal assertion on `cfg_mgmt_read_write_done`. The following figure illustrates an example with read from the Configuration Space.

Figure 108: `cfg_mgmt_read_type0_type1`



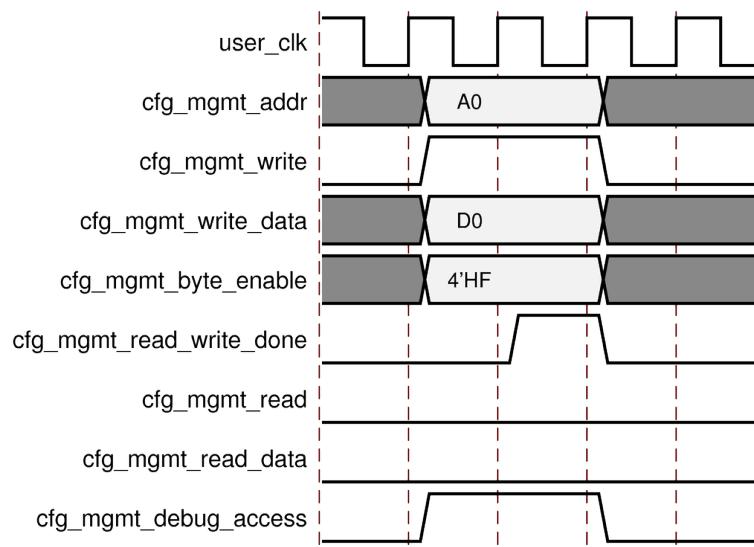
To write any register in configuration space, the user logic places the address on the `cfg_mgmt_addr[9:0]`, the function number on `cfg_mgmt_function_number[7:0]`, write data on `cfg_mgmt_write_data`, byte-valid on `cfg_mgmt_byte_enable [3:0]`, and asserts the `cfg_mgmt_write` signal. In response, the core asserts the `cfg_mgmt_read_write_done` signal when the write is complete (which can take several cycles). The user logic must keep `cfg_mgmt_addr`, `cfg_mgmt_function_number`, `cfg_mgmt_write_data`, `cfg_mgmt_byte_enable` and `cfg_mgmt_write` stable until `cfg_mgmt_read_write_done` is asserted. The user logic must also deassert `cfg_mgmt_write` in the cycle following the `cfg_mgmt_read_write_done` from the core.

Figure 109: `cfg_mgmt_write_type0`



When the core is configured in the Root Port mode, when you assert `cfg_mgmt_debug_access` input during a write to a Type-1 PCI™ Configuration register forces a write into certain read-only fields of the register.

Figure 110: `cfg_mgmt_debug_access`



Enabling Loopback Master on Root Port

The Loopback Master feature is implemented in the VSEC space (see [PCI Express Extended Configuration Space](#)). Loopback Master capability for Root Port starts at byte address '`h330`. To exercise Loopback Start, use the `cfg_mgmt` interface to write `1'b1` to bit 0 in byte address '`h338` (Loopback Control).

Because `cfg_mgmt` uses DW addresses, use `cfg_mgmt_write_data = 32'h1` to address:

- `cfg_mgmt_address = 10'hCE`
- `cfg_mgmt_write = 1'b1`
- `cfg_mgmt_byte_enable = 4'b1`

After Start Loopback, retrain the link by writing to the link control register (byte address: '`h80`, DW address: '`h20`). Use `cfg_mgmt_write_data = 'h20` to address:

- `cfg_mgmt_address = 10'h20`
- `cfg_mgmt_write = 1'b1`
- `cfg_mgmt_byte_enable = 4'b1`

Link Training: 2-Lane, 4-Lane, 8-Lane, and 16-Lane Components

The 2-lane, 4-lane, and 8-lane cores can operate at less than the maximum lane width as required by the PCI Express® Base Specification. Two cases cause the core to operate at less than its specified maximum lane width, as defined in these subsections.

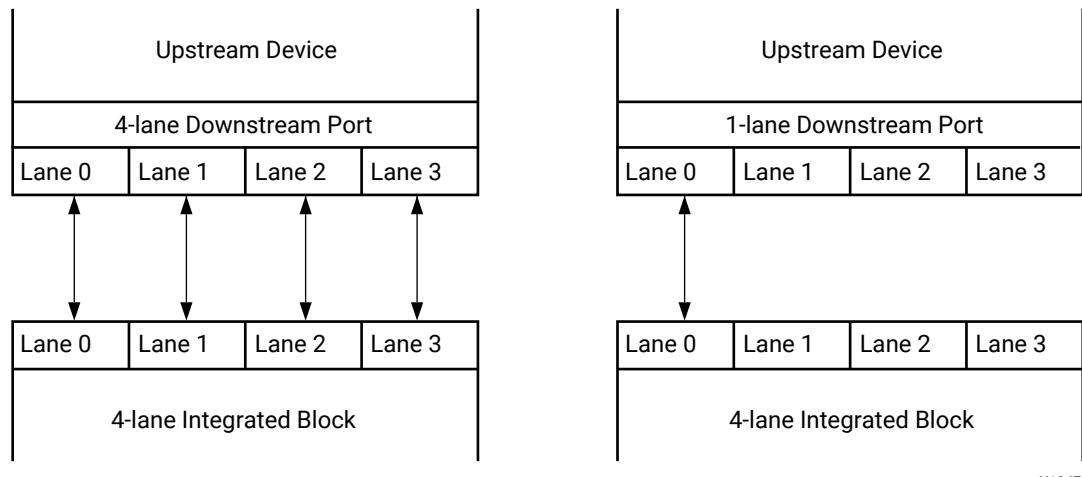
Link Partner Supports Fewer Lanes

When the 2-lane core is connected to a device that implements only 1 lane, the 2-lane core trains and operates as a 1-lane device using lane 0.

When the 4-lane core is connected to a device that implements 1 lane, the 4-lane core trains and operates as a 1-lane device using lane 0, as shown in the following figure. Similarly, if the 4-lane core is connected to a 2-lane device, the core trains and operates as a 2-lane device using lanes 0 and 1.

When the 8-lane core is connected to a device that only implements 4 lanes, it trains and operates as a 4-lane device using lanes 0-3. Additionally, if the connected device only implements 1 or 2 lanes, the 8-lane core trains and operates as a 1- or 2-lane device.

Figure 111: Scaling of 4-Lane Endpoint Block from 4-Lane to 1-Lane Operation



X12470

Lane Becomes Faulty

If a link becomes faulty after training to the maximum lane width supported by the core and the link partner device, the core attempts to recover and train to a lower lane width, if available. If lane 0 becomes faulty, the link is irrecoverably lost. If any or all of lanes 1-7 become faulty, the link goes into recovery and attempts to recover the largest viable link with whichever lanes are still operational.

For example, when using the 8-lane core, loss of lane 1 yields a recovery to 1-lane operation on lane 0, whereas the loss of lane 6 yields a recovery to 4-lane operation on lanes 0-3. After recovery occurs, if the failed lane(s) becomes alive again, the core does not attempt to recover to a wider link width. The only way a wider link width can occur is if the link actually goes down and it attempts to retrain from scratch.

The `user_clk` clock output is a fixed frequency configured in IP catalog. `user_clk` does not shift frequencies in case of link recovery or training down.

Lane Reversal

The integrated block supports limited lane reversal capabilities and therefore provides flexibility in the design of the board for the link partner. The link partner can choose to lay out the board with reversed lane numbers and the integrated block continues to link train successfully and operate normally. The configurations that have lane reversal support are 16x, x8 and x4 (excluding downshift modes). Downshift refers to the link width negotiation process that occurs when link partners have different lane width capabilities advertised. As a result of lane width negotiation, the link partners negotiate down to the smaller of the advertised lane widths. The following table describes the several possible combinations including downshift modes and availability of lane reversal support.

Table 66: Lane Reversal Support

Integrated Block Advertised Lane Width	Negotiated Lane Width	Lane Number Mapping (Endpoint Link Partner)		Lane Reversal Supported
		Endpoint	Link Partner	
x16	x16	Lane 0... Lane15	Lane15... Lane 0	Yes
x16	x8	Lane 0... Lane7	Lane7... Lane 0	No
x16	x4	Lane 0... Lane3	Lane3... Lane 0	No
x16	x2	Lane 0... Lane1	Lane1... Lane 0	No
x8	x8	Lane 0... Lane 7	Lane 7... Lane 0	Yes
x8	x4	Lane 0... Lane 3	Lane 7... Lane 4	No ¹
x8	x2	Lane 0... Lane 3	Lane 7... Lane 6	No ¹
x4	x4	Lane 0... Lane 3	Lane 3... Lane 0	Yes
x4	x2	Lane 0... Lane 1	Lane 3... Lane 2	No ¹
x2	x2	Lane 0... Lane 1	Lane 1... Lane 0	Yes
x2	x1	Lane 0... Lane 1	Lane 1	No ¹

Notes:

- When the lanes are reversed in the board layout and a downshift adapter card is inserted between the Endpoint and link partner, Lane 0 of the link partner remains unconnected (as shown by the lane mapping in this table) and therefore does not link train.



IMPORTANT! By default EndPoint configuration has Lane Reversal Support disabled through the attribute `DISABLE_LANE_REVERSAL`. Lane Reversal as an EndPoint must not be enabled if the Link Partner has Lane Reversal capability.

Design Flow Steps

This section describes customizing and generating the core, constraining the core, and the simulation, synthesis, and implementation steps that are specific to this IP core. More detailed information about the standard Vivado® design flows and the IP integrator can be found in the following Vivado Design Suite user guides:

- Vivado Design Suite User Guide: *Designing IP Subsystems using IP Integrator* ([UG994](#))
- Vivado Design Suite User Guide: *Designing with IP* ([UG896](#))
- Vivado Design Suite User Guide: *Getting Started* ([UG910](#))
- Vivado Design Suite User Guide: *Logic Simulation* ([UG900](#))

Customizing and Generating the Core

This section includes information about using Xilinx® tools to customize and generate the core in the Vivado® Design Suite.

If you are customizing and generating the core in the Vivado IP integrator, see the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* ([UG994](#)) for detailed information. IP integrator might auto-compute certain configuration values when validating or generating the design. To check whether the values do change, see the description of the parameter in this chapter. To view the parameter value, run the `validate_bd_design` command in the Tcl console.

You can customize the IP for use in your design by specifying values for the various parameters associated with the IP core using the following steps:

1. Select the IP from the IP catalog.
2. Double-click the selected IP or select the Customize IP command from the toolbar or right-click menu.

For details, see the *Vivado Design Suite User Guide: Designing with IP* ([UG896](#)) and the *Vivado Design Suite User Guide: Getting Started* ([UG910](#)).

Figures in this chapter are illustrations of the Vivado IDE. The layout depicted here might vary from the current version.

The Customize IP dialog box for the UltraScale+ Devices Integrated Block for PCI Express core consists of two modes: **Basic Mode Parameters** and **Advanced Mode Parameters**. To select a mode, use the Mode drop-down list on the first page of the Customize IP dialog box. The following sections explain the parameters available in each of these modes.

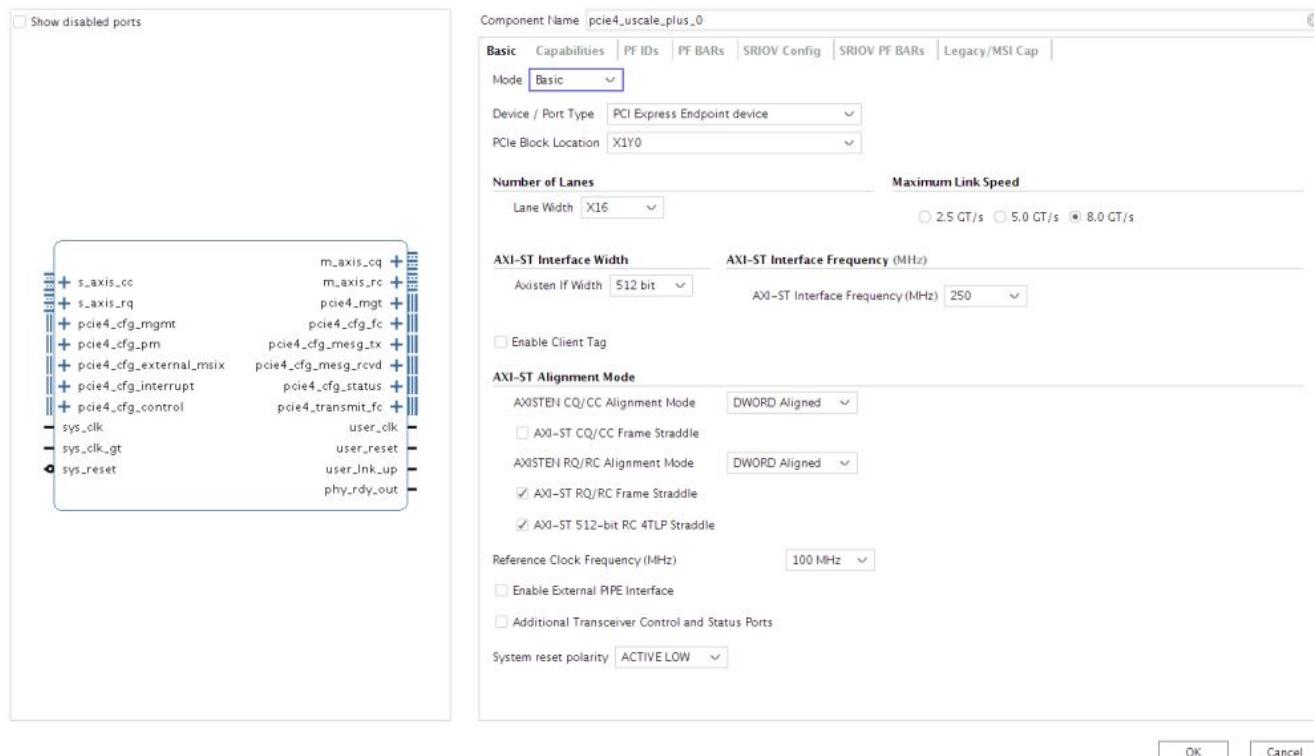
Basic Mode Parameters

The Basic mode parameters are explained in this section.

Basic Tab

The following figure shows the initial customization page, used to set the Basic mode parameters.

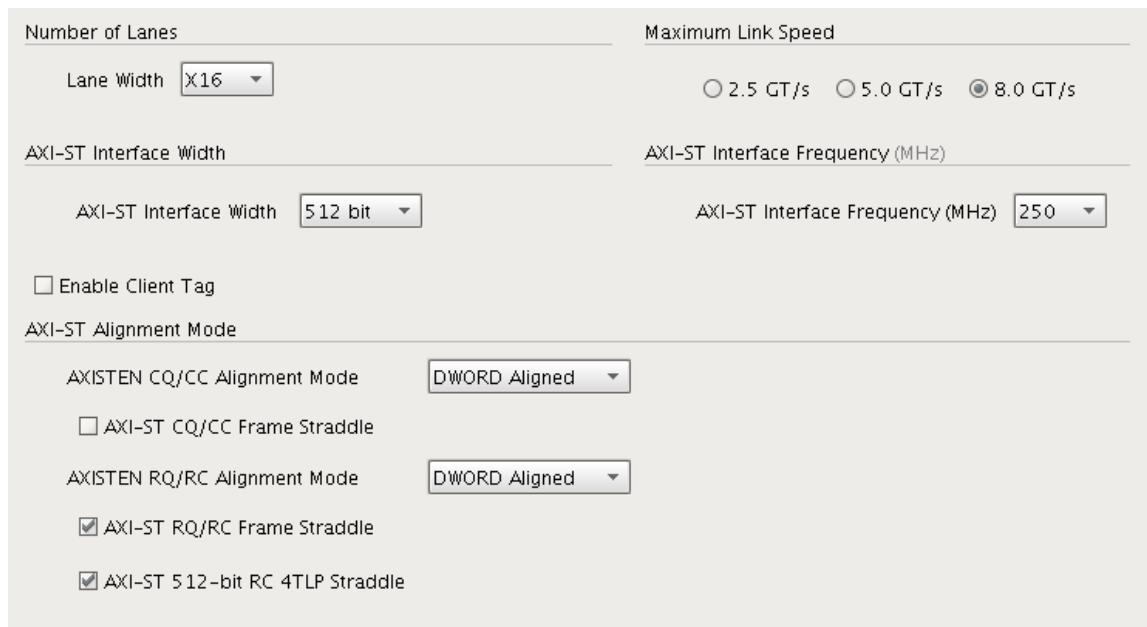
Figure 112: Basic Tab



- **Component Name:** Base name of the output files generated for the core. The name must begin with a letter and can be composed of these characters: a to z, 0 to 9, and “_”.
- **Mode:** Allows you to select the Basic or Advanced mode of the configuration of core.
- **Device/Port Type:** Indicates the PCI Express logical device type.

- **PCIe Block Location:** Selects from the available integrated blocks to enable generation of location-specific constraint files and pinouts. This selection is used in the default example design scripts.
- **Number of Lanes:** The core requires the selection of the initial lane width. See [Clock Frequencies and Interface Widths Supported For Various Configurations](#) for the supported link width configuration. Wider lane width cores can train down to smaller lane widths if attached to a smaller lane-width device. For more information, see [Link Training: 2-Lane, 4-Lane, 8-Lane, and 16-Lane Components](#).
- **Maximum Link Speed:** The core allows you to select the Maximum Link Speed supported by the device. [Clock Frequencies and Interface Widths Supported For Various Configurations](#) defines the lane widths and link speeds supported by the device. Higher link speed cores are capable of training to a lower link speed if connected to a lower link speed capable device.
- **AXI-ST Interface Width:** The core allows you to select the Interface Width. The default interface width set in the Customize IP dialog box is the lowest possible interface width.
- **AXI-ST Interface Frequency:** [Table 46: Clock Frequencies and Interface Widths Supported For Various Configurations](#) defines the desired AXI-ST Interface frequency based on the PCIe IP configuration.
- **Enable Client Tag:** Enables you to use the client tag.
- **AXI-ST Alignment Mode:** When a payload is present, there are two options for aligning the first byte of the payload with respect to the datapath. The options are provided to select the CQ/CC and RQ/RC interfaces. See [Data Alignment Options](#).
- **Enable AXI-ST Frame Straddle:** The core provides an option to straddle packets on the requester completion interface when the interface width is 256 bits. See [Straddle Option for 256-Bit Interface](#).

Figure 113: Basic Tab, Straddle Options



- AXI-ST CQ/CC Frame Straddle and AXI-ST RQ/RC Frame Straddle:** When 512-bit AXI-ST interface width is selected AXI-ST frame Straddle is supported for CQ, CC, RQ and RC AXI-ST interfaces. Option to select CQ and CC AXI-ST frame straddle together and for RQ and RC interfaces
- AXI-ST 512-bit RC 4TLP Straddle:** When AXI-ST 512-bit RC 4TLP straddle is set to TRUE, then it is 4 TLP straddle mode, otherwise it is 2 TLP straddle mode, which can start a new TLP at byte lane 32 when the previous TLP has ended at or before byte lane 31. See [512-Bit Completer Interface](#).
- Reference Clock Frequency:** Selects the frequency of the reference clock provided on `sys_clk`. For important information about clocking the core, see [Clocking](#).
- Enable External PIPE Interface:** When selected, this option enables an external third-party bus functional model (BFM) to connect to the PIPE interface of integrated block for PCIe. For details, see [PIPE Mode Simulation Using Integrated Endpoint PCI Express Block in Gen3 x8 and Gen2 x8 Configurations \(XAPP1184\)](#). Refer to these designs to connect the External PIPE Interface ports of the UltraScale™ device core to third-party BFM.
- Additional Transceiver Control and Status Ports:** These ports are used to debug transceiver-related issues and PCIe-related signals. You have to drive in accordance with the appropriate GT user guide.



IMPORTANT! The Enable In System IBERT and Enable JTAG Debugger options should be used only for hardware debug purposes. Simulations are not supported for the cores generated using these options.

- System reset polarity:** This parameter is used to set the polarity of the `sys_rst` ACTIVE_HIGH or ACTIVE_LOW.

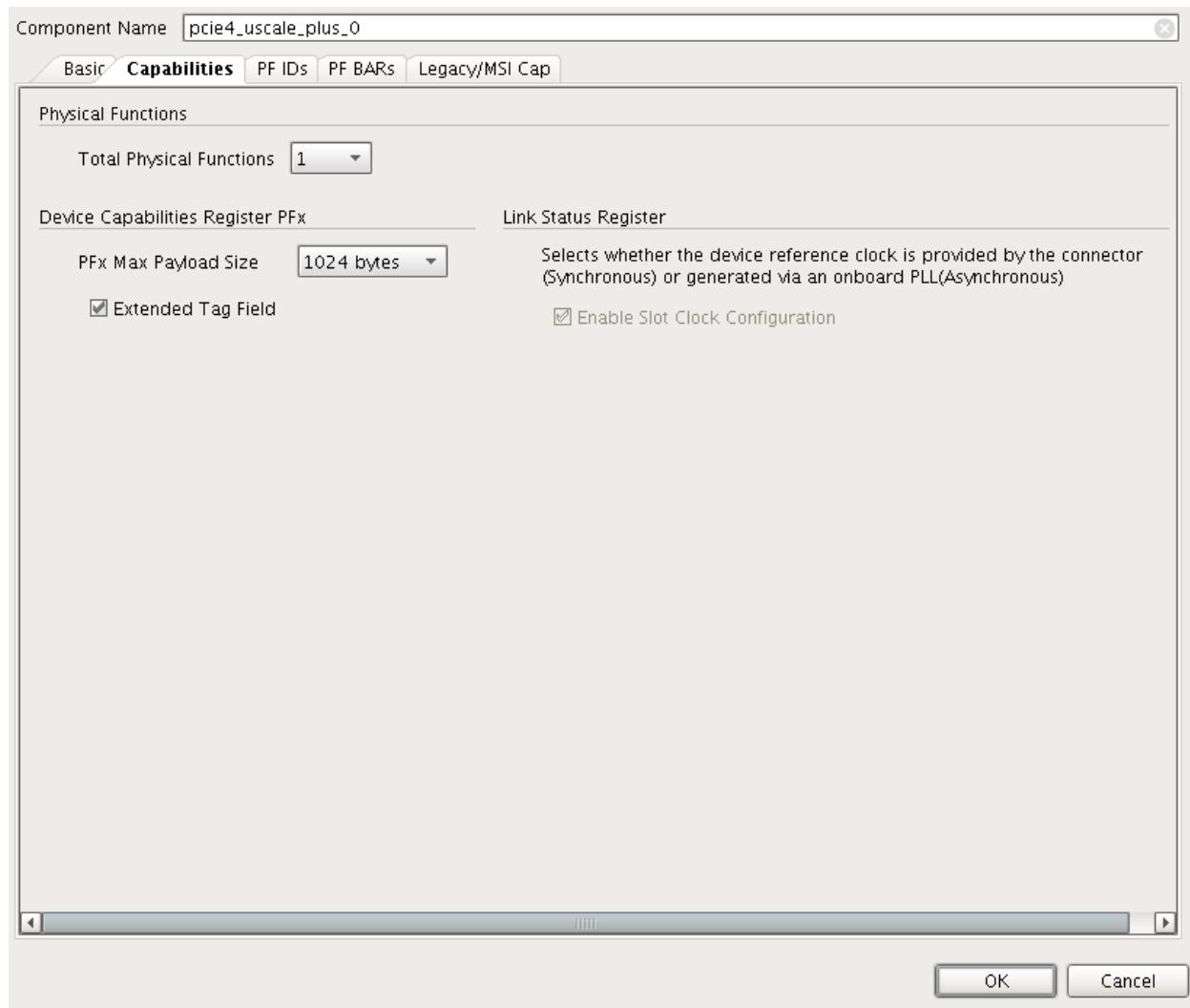
Related Information

[Transceiver Control and Status Ports](#)

Capabilities Tab

The Capabilities settings are explained in this section as shown in the following figure.

Figure 114: Capabilities Tab



- **Physical Functions:** Enables you to select the number of physical functions. The number of physical functions supported is 4.
- **PFx Max Payload Size:** This field indicates the maximum payload size that the device or function can support for TLPs. This is the value advertised to the system in the Device Capabilities Register.

- Extended Tag Field:** This field indicates the maximum supported size of the Tag field as a Requester. The options are:
 - When selected, 8-bit Tag field support (256 tags)
 - When deselected, 5-bit Tag field support (32 tags)
- Enable Slot Clock Configuration:** Enables the Slot Clock Configuration bit in the Link Status register. When you select this option, the link is synchronously clocked. For more information on clocking options, see [Clocking](#).

PF IDs Tab

The following figure shows the Identity Settings parameters.

Figure 115: PF IDs Tab

PF#	Vendor ID	Device ID	Revision ID	Subsystem Vendor ID	Subsystem ID
PF0	10EE	903F	00	10EE	0007
PF1	10EE	903F	00	10EE	0007
PF2	10EE	903F	00	10EE	0007
PF3	10EE	903F	00	10EE	0007

PF#	Use Classcode Lookup Assistant	Base Class Menu	Base Class Value	Subclass Interface Menu	Subclass Value	Interface Value	Class Code
PF0	<input type="checkbox"/>	Memory controller	05	X RAM	00	X 00	X 050000
PF1	<input type="checkbox"/>	Memory controller	05	X RAM	00	X 00	X 050000
PF2	<input type="checkbox"/>	Memory controller	05	X RAM	00	X 00	X 050000
PF3	<input type="checkbox"/>	Memory controller	05	X RAM	00	X 00	X 050000

- Enable PCIe-ID Interface:** If this parameter is selected the PCIe ID ports `cfg_vend_id`, `cfg_subsys_vend_id`, `cfg_dev_id_pf*`, `cfg_rev_id_pf*`, and `cfg_subsys_id_pf*` appears at the boundary of core top depending on the number of PFx that are selected and available to be driven by user logic. If unselected they do not appear at the top level and are driven with the values set at the time of customization.
- PF0 ID Initial Values:**
 - Vendor ID:** Identifies the manufacturer of the device or application. Valid identifiers are assigned by the PCI Special Interest Group to guarantee that each identifier is unique. The default value, `10EEh`, is the Vendor ID for Xilinx. Enter a vendor identification number here. `FFFFh` is reserved.
 - Device ID:** A unique identifier for the application; the default value, which depends on the configuration selected, is `70<link speed><link width>h`. This field can be any value; change this value for the application. The default Device ID parameter is based on:
 - The device family: 9 for UltraScale+, 8 for UltraScale, or 7 for 7 series devices.
 - EP or RP mode.

- Link width: 1 for x1, 2 for x2, 4 for x4, 8 for x8, and F for x16.
- Link speed: 1 for Gen1, 2 for Gen2, 3 for Gen3, and 4 for Gen4.

If any of the above values are changed, the Device ID value will be re-evaluated, replacing the previous set value.



RECOMMENDED: It is always recommended that the link width, speed, and Device Port type be changed first and then the Device ID value. Make sure the Device ID value is set correctly before generating the IP.

- **Revision ID:** Indicates the revision of the device or application; an extension of the Device ID. The default value is 00h; enter a value appropriate for the application.
- **Subsystem Vendor ID:** Further qualifies the manufacturer of the device or application. Enter a Subsystem Vendor ID here; the default value is 10EEh. Typically, this value is the same as Vendor ID. Setting the value to 0000h can cause compliance testing issues.
- **Subsystem ID:** Further qualifies the manufacturer of the device or application. This value is typically the same as the Device ID; the default value depends on the lane width and link speed selected. Setting the value to 0000h can cause compliance testing issues.
- **Class Code:** The Class Code identifies the general function of a device, and is divided into three byte-size fields:
 - **Base Class:** Broadly identifies the type of function performed by the device.
 - **Sub-Class:** More specifically identifies the device function.
 - **Interface:** Defines a specific register-level programming interface, if any, allowing device-independent software to interface with the device.

Class code encoding can be found at the [PCI-SIG website](#).

- **Class Code Look-up Assistant:** The Class Code Look-up Assistant provides the Base Class, Sub-Class and Interface values for a selected general function of a device. This Look-up Assistant tool only displays the three values for a selected function. You must enter the values in Class Code for these values to be translated into device settings.

PF BARs Tab

The PF BARs tab, shown in the following figure, sets the base address register space for the Endpoint configuration. Each BAR (0 through 5) configures the BAR Aperture Size and Control attributes of the physical function.

Figure 116: PF BARs Tab Showing PF0 and PF1 Only

Bar	Type	64 bit	Prefetchable	Size	Scale	Value (Hex)
<input checked="" type="checkbox"/>	Memory	<input type="checkbox"/>	<input type="checkbox"/>	128	Kilobytes	FFFE0000
<input type="checkbox"/>	N/A	<input type="checkbox"/>	<input type="checkbox"/>	128	Kilobytes	00000000
<input type="checkbox"/>	N/A	<input type="checkbox"/>	<input type="checkbox"/>	128	Kilobytes	00000000
<input type="checkbox"/>	N/A	<input type="checkbox"/>	<input type="checkbox"/>	128	Kilobytes	00000000
<input type="checkbox"/>	N/A	<input type="checkbox"/>	<input type="checkbox"/>	128	Kilobytes	00000000
<input type="checkbox"/>	N/A	<input type="checkbox"/>	<input type="checkbox"/>	128	Kilobytes	00000000
<input type="checkbox"/>		<input type="checkbox"/>	<input type="checkbox"/>	2	Kilobytes	00000000

Bar	Type	64 bit	Prefetchable	Size	Scale	Value (Hex)
<input checked="" type="checkbox"/>	Memory	<input type="checkbox"/>	<input type="checkbox"/>	128	Kilobytes	FFFE0000
<input type="checkbox"/>	N/A	<input type="checkbox"/>	<input type="checkbox"/>	128	Kilobytes	00000000
<input type="checkbox"/>	N/A	<input type="checkbox"/>	<input type="checkbox"/>	128	Kilobytes	00000000
<input type="checkbox"/>	N/A	<input type="checkbox"/>	<input type="checkbox"/>	128	Kilobytes	00000000
<input type="checkbox"/>	N/A	<input type="checkbox"/>	<input type="checkbox"/>	128	Kilobytes	00000000
<input type="checkbox"/>	N/A	<input type="checkbox"/>	<input type="checkbox"/>	128	Kilobytes	00000000
<input type="checkbox"/>		<input type="checkbox"/>	<input type="checkbox"/>	2	Kilobytes	00000000

PF2 PF3

- Base Address Register Overview:** In Endpoint configuration, the core supports up to six 32-bit BARs or three 64-bit BARs, and the Expansion read-only memory (ROM) BAR. In Root Port configuration, the core supports up to two 32-bit BARs or one 64-bit BAR, and the Expansion ROM BAR. BARs can be one of two sizes:

- 32-bit BARs:** The address space can be as small as 128 bytes or as large as 2 gigabytes. Used for Memory or I/O.
- 64-bit BARs:** The address space can be as small as 128 bytes or as large as 8 Exabytes. Used for Memory only.

All BAR registers share these options:

- Checkbox:** Click the checkbox to enable the BAR; deselect the checkbox to disable the BAR.
- Type:** Bars can either be I/O or Memory.
 - I/O:** I/O BARs can only be 32-bit; the Prefetchable option does not apply to I/O BARs. I/O BARs are only enabled for a Legacy PCI Express Endpoint.
 - Memory:** Memory BARs can be either 64-bit or 32-bit and can be prefetchable. When a BAR is set as 64 bits, it uses the next BAR for the extended address space and makes the next BAR inaccessible.
- Size:** The available Size range depends on the PCIe Device/Port Type and the Type of BAR selected. The following table lists the available BAR size ranges.

Table 67: BAR Size Ranges for Device Configuration

PCIe Device / Port Type	BAR Type	BAR Size Range
PCI Express Endpoint	32-bit Memory	128 bytes (B) – 2 gigabytes (GB)
	64-bit Memory	128 B – 8 Exabytes
Legacy PCI Express Endpoint	32-bit Memory	128 B – 2 GB
	64-bit Memory	128 B – 8 Exabytes
	I/O	16 B – 2 GB

- **Prefetchable:** Identifies the ability of the memory space to be prefetched.
- **Value:** The value assigned to the BAR based on the current selections.
- **Expansion ROM Base Address Register:** If selected, the Expansion ROM is activated and can be sized from 2 KB to 4 GB. According to the PCI Local Bus Specification Revision 3.0 on the [PCI-SIG website](#), the maximum size for the Expansion ROM BAR should be no larger than 16 MB. Selecting an address space larger than 16 MB can cause compliance testing issues.
- **Managing Base Address Register Settings:** Memory, I/O, Type, and Prefetchable settings are handled by setting the appropriate settings for the desired base address register.

Memory or I/O settings indicate whether the address space is defined as memory or I/O. The base address register only responds to commands that access the specified address space. Generally, memory spaces less than 4 KB in size should be avoided. The minimum I/O space allowed is 16 bytes; use of I/O space should be avoided in all new designs.

Prefetchability is the ability of memory space to be prefetched. A memory space is prefetchable if there are no side effects on reads (that is, data is not destroyed by reading, as from a RAM). Byte-write operations can be merged into a single double word write, when applicable.

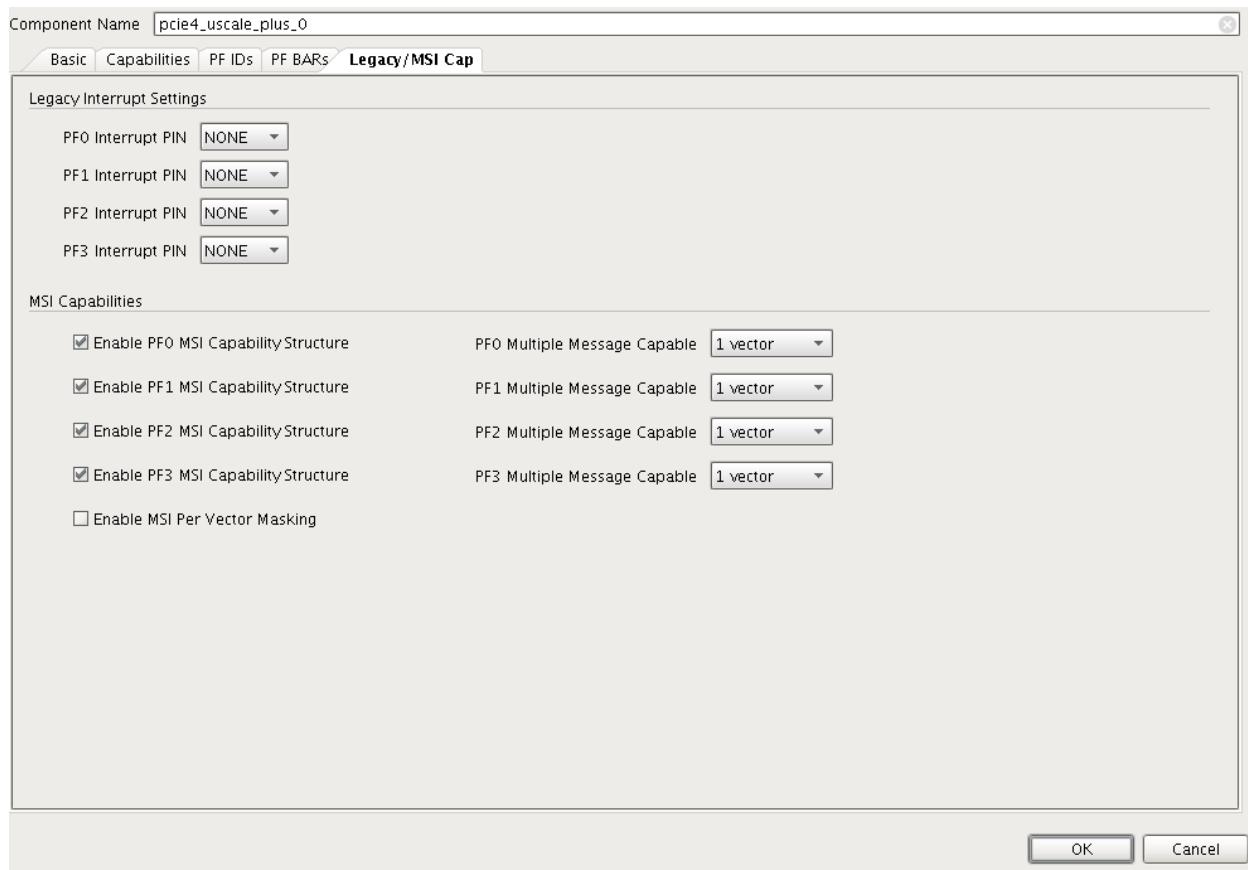
When configuring the core as an Endpoint for PCIe (non-Legacy), 64-bit addressing must be supported for all BARs (except BAR5) that have the prefetchable bit set. 32-bit addressing is permitted for all BARs that do not have the prefetchable bit set. The prefetchable bit-related requirement does not apply to a Legacy Endpoint. The minimum memory address range supported by a BAR is 128 bytes for a PCI Express Endpoint and 16 bytes for a Legacy PCI Express Endpoint.

- **Disabling Unused Resources:** For best results, disable unused base address registers to conserve system resources. A base address register is disabled by deselecting unused BARs in the Customize IP dialog box.

Legacy/MSI Cap Tab

On this page, you set the Legacy Interrupt Settings and MSI Capabilities for all applicable physical and virtual functions. This page is not visible when the **SRIOV Capability** parameter is selected on the Capabilities page (which is visible when the **Advanced** mode is selected). This page is visible in both Basic mode and Advanced mode (when the SRIOV Capability parameter is not selected).

Figure 117: Legacy/MSI Cap Tab



- **Legacy Interrupt Settings:**

- **PFO/PF1/PF2/PF3 Interrupt PIN:** Indicates the mapping for Legacy Interrupt messages. A setting of None indicates that no Legacy Interrupts are used.

- **MSI Capabilities:**

- **PFO/PF1/PF2/PF3 Enable MSI Capability Structure:** Indicates that the MSI Capability structure exists.

Note: Although it is possible to not enable MSI or MSI-X, the result would be a non-compliant core. The PCI Express Base Specification requires that MSI, MSI-X, or both be enabled. No MSI capabilities are supported when **MSI-X Internal** is enabled in the [MSI-X Capabilities Tab](#) (Advanced mode), because MSI-X Internal uses some of the MSI interface signals.

- **PF0/PF1/PF2/PF3 Multiple Message Capable:** Selects the number of MSI vectors to request from the Root Complex.
- **Enable MSI Per Vector Masking:** Enables MSI Per Vector Masking Capability of all the Physical functions enabled.

Note: Enabling this option for individual physical functions is not supported.

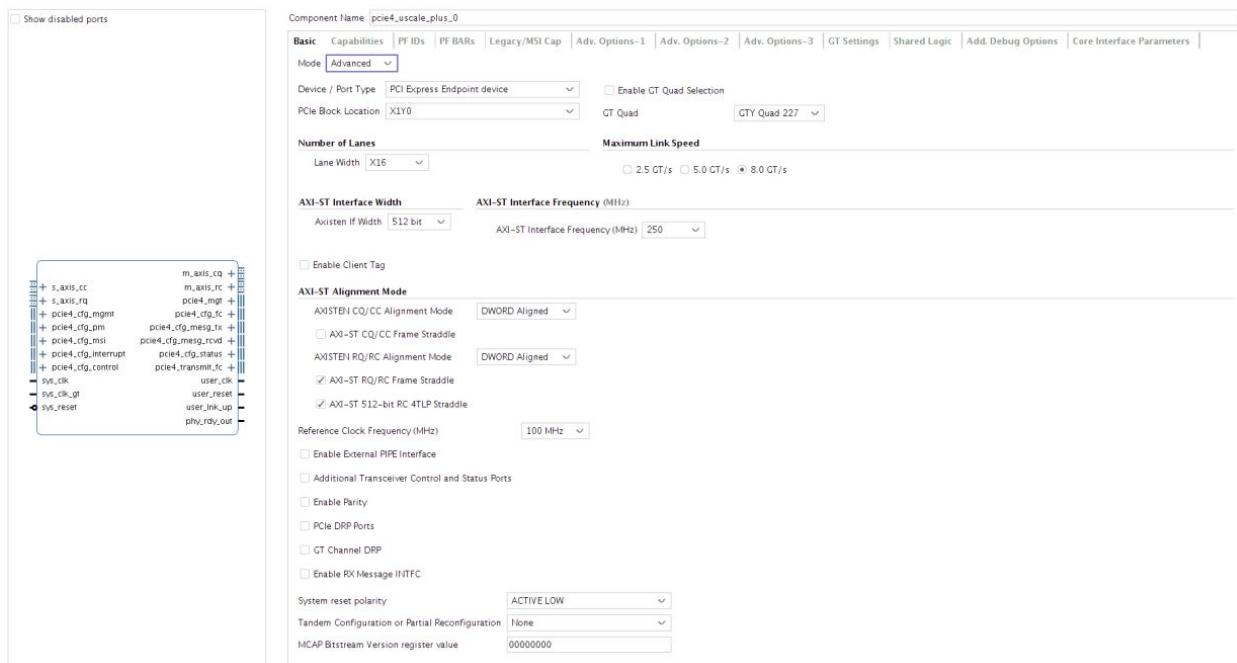
Advanced Mode Parameters

The following parameters appear on different pages of the IP catalog when **Advanced** mode is selected for Mode on the Basic page.

Basic Tab

The Basic page with Advanced mode selected (shown in the following figure) includes additional settings. The following parameters are visible on the Basic page when the **Advanced** mode is selected.

Figure 118: Basic Tab, Advanced Mode



- **Enable GT Quad Selection:** When selected, this parameter enables you to select the different GT Quads available for a PCIe block.
- **GT Quad:** This drop-down menu lists all the available GT quads for a given PCIe block. Note that depending on the GT Quad selected, the Maximum link width that can be configured for that particular quad will change.
- **Core Clock Frequency:** This parameter allows you to select the core clock frequencies.

For Gen3 link speed:

- For a -1, -2 and -3 speed grade and an x8 link width, the values of 250 MHz and 500 MHz are available for selection
- For a -2L, -2LV, -1L and -1LV speed grade and an x8 link width, this parameter defaults to 250 MHz and is not available for selection.
- For x16 link widths, the value of this parameter defaults to 500 MHz and is not available for selection.
- For x1,x2 and x4 link widths, this parameter defaults to 250 MHz and is not available for selection.

For Gen1 and Gen2 link speeds:

- For link widths other than x8, this parameter defaults to 250 MHz and is not available for selection.
- For x8 link widths, this parameter defaults to 500 MHz and is not available for selection.
- **Enable Parity:** Enables Parity on TX/RX interfaces including MSI-X.
- **PCIe DRP Ports:** When checked, enables the PCIe DRP interface.
- **GT Channel DRP:** When checked, enables the GT channel DRP interface.

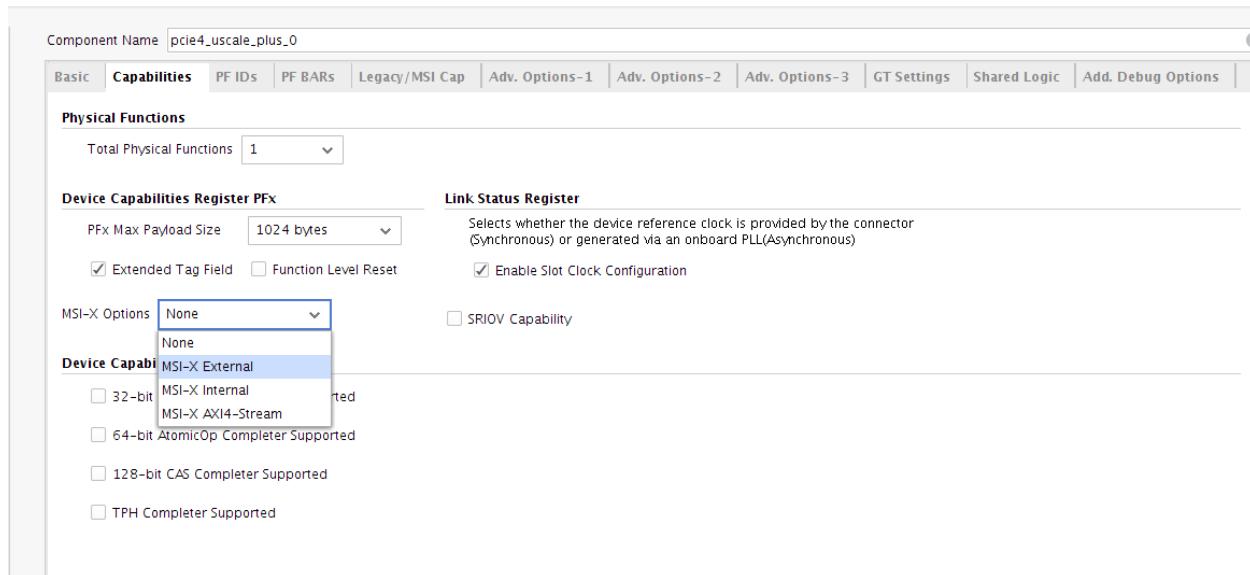
Related Information

[GT Locations](#)

Capabilities Tab

The Capabilities settings for Advanced mode (as shown in the following figure) contains two additional parameters to those for Basic mode and are described below.

Figure 119: Capabilities Tab, Advanced Mode



- **SRIOV Capabilities:** Enables Single Root Port I/O Virtualization (SR-IOV) capabilities. The integrated block implements extended Single Root Port I/O Virtualization PCIe. When this is enabled, SR-IOV is implemented on all the selected physical functions. When SR-IOV capabilities are enabled MSI support is disabled and you can use MSI-X support as shown in the above figure.

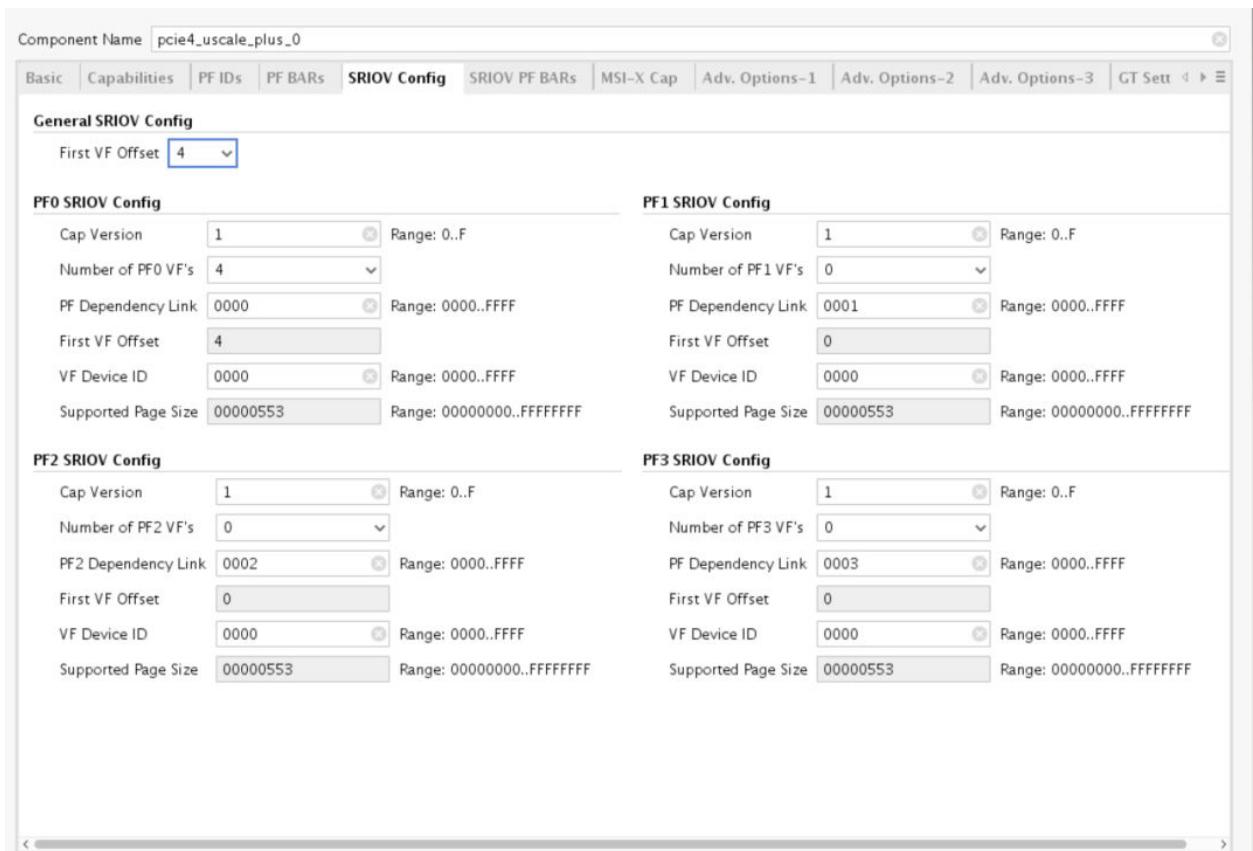
Note: When SR-IOV capabilities are enabled, MSI support is disabled and you can use MSI-X support.

- **Device Capabilities Registers 2:** Specifies options for AtomicOps and TPH Completer support. See the Device Capability register 2 description in the [PCI Express Base Specification](#) for more information. These settings apply to all the selected physical functions.

SRIOV Config Tab

The SRIOV Configuration Advanced parameters, as shown in the following figure, are described in this section.

Figure 120: SRIOV Configuration Tab



- **General SRIOV Config:** This value specifies the offset of the first PF with at least one enabled VF. When ARI is enabled, allowed value is 'd4 or 'd64, and the total number of VF in all PFs plus this field must not be greater than 256. When ARI is disabled, this field will be set to 1 to support 1PF plus 7VF non-ARI SR-IOV configurations only.
- **Cap Version:** Indicates the 4-bit SR-IOV Capability version for the physical function.
- **Number of PFx VFs:** Indicates the number of virtual functions associated to the physical function. A total of 252 virtual functions are available that can be flexibly used across the four physical functions.
- **PFx Dependency Link:** Indicates the SR-IOV Functional Dependency Link for the physical function. The programming model for a device can have vendor-specific dependencies between sets of functions. The Function Dependency Link field is used to describe these dependencies.
- **First VF Offset:** Indicates the offset of the first virtual function (VF) for the physical function (PF). PFx offset is always fixed. PF0 resides at offset 0, PF1 resides at offset 1, PF2 resides at offset 2, and PF3 resides at offset 3.

A total of 252 virtual functions are available. They reside at the function number range 4 to 255.

You can select either 4 or 64 for the first VF offset in the customization GUI, if the last VF offset is not more than 255.

- **Examples:**

- When the total number of enabled VFs is less than 192, select either 4 or 64 for the first VF offset. The last VF offset will be less than 255.
- When the total number of enabled VFs is more than 192, select 4 for first VF offset. In this case, 64 is not permitted for first VF offset, because the last VF offset will become more than 255.

Virtual functions are mapped sequentially with VFs with PFs taking precedence. For example, if PF0 has two virtual functions and PF1 has three, the following mapping occurs:

The PFx_FIRST_VF_OFFSET is calculated by taking the first offset of the virtual function and subtracting that from the offset of the physical function.

```
PFx_FIRST_VF_OFFSET = (PFx first VF offset - PFx offset)
```

In the example above, the following offsets are used:

```
PF0_FIRST_VF_OFFSET = (4 - 0) = 4  
PF1_FIRST_VF_OFFSET = (6 - 1) = 5
```

The initial offset for PF1 is a function of how many VFs are attached to PF0 and is defined in the following pseudo code:

```
PF1_FIRST_VF_OFFSET = FIRST_VF_OFFSET + NUM_PF0_VFs - 1
```

Similarly, for other PFs:

```
PF2_FIRST_VF_OFFSET = FIRST_VF_OFFSET + NUM_PF0_VFs + NUM_PF1_VFs - 2  
PF3_FIRST_VF_OFFSET =  
FIRST_VF_OFFSET + NUM_PF0_VFs + NUM_PF1_VFs + NUM_PF2_VFs - 3
```

- **VF Device ID:** Indicates the 16-bit Device ID for all virtual functions associated with the physical function.
- **SRIOV Supported Page Size:** Indicates the page size supported by the physical function. This physical function supports a page size of $2^{(n+12)}$, if bit n of the 32-bit register is set.

SRIOV BARs Tab

The SRIOV Base Address Registers (BARs) set the base address register space for the Endpoint configuration. Each BAR (0 through 5) configures the SR-IOV BAR aperture size and SR-IOV control attributes.

Figure 121: SRIOV BARs Tab, Advanced Mode

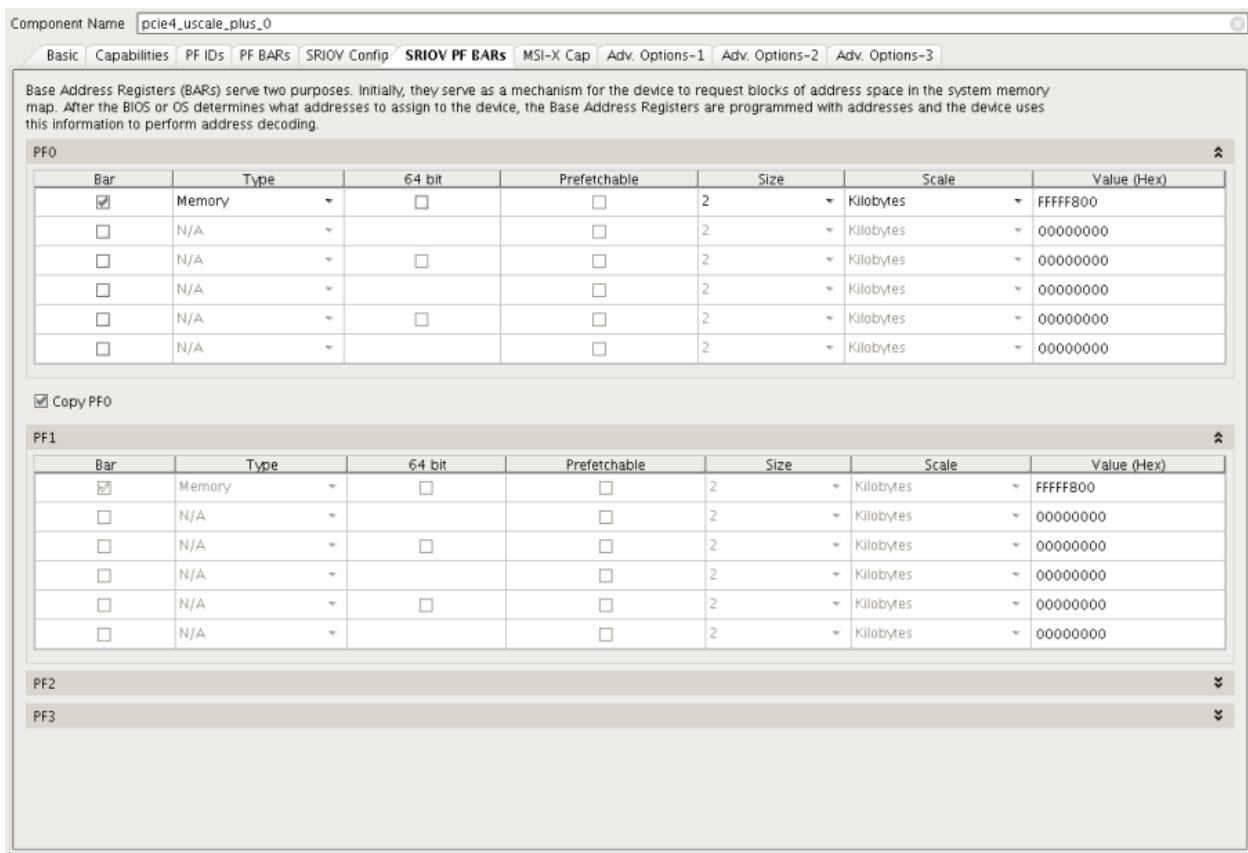


Table 68: Example Virtual Function Mappings

Physical Function	Virtual Function	Function Number Range
PF0	VF0	64
PF0	VF1	65
PF1	VF0	66
PF1	VF1	67
PF1	VF1	68

- SRIOV Base Address Register Overview:** In Endpoint configuration, the core supports up to six 32-bit BARs or three 64-bit BARs. In Root Port configuration, the core supports up to two 32-bit BARs or one 64-bit BAR. SR-IOV BARs can be one of two sizes:
 - 32-bit BARs:** The address space can be as small as 16 bytes or as large as 3 gigabytes. Used for memory to I/O.
 - 64-bit BARs:** The address space can be as small as 128 bytes or as large as 256 gigabytes. Used for memory only.

All SR-IOV BAR registers have these options:

- **Checkbox:** Click the checkbox to enable the BAR; deselect the checkbox to disable the BAR.
- **Type:** SR-IOV BARs can be either I/O or Memory.
 - **I/O:** I/O BARs can only be 32-bit; the Prefetchable option does not apply to I/O BARs. I/O BARs are only enabled for a Legacy PCI Express Endpoint.
 - **Memory:** Memory BARs can be either 64-bit or 32-bit and can be prefetchable. When a BAR is set to 64-bits, it uses the next BAR for the extended address space and makes the next BAR inaccessible.
- **Size:** The available size range depends on the PCIe device/port type and the type of BAR selected. The following table lists the available BAR size ranges.

Table 69: SRIOV BAR Size Ranges for Device Configuration

PCIe Device / Port Type	BAR Type	BAR Size Range
PCI Express Endpoint	32-bit Memory	128 bytes – 2 gigabytes
	64-bit Memory	128 bytes – 8 exabytes
Legacy PCI Express Endpoint	32-bit Memory	16 bytes – 2 gigabytes
	64-bit Memory	16 bytes – 8 exabytes
	I/O	16 bytes – 2 gigabytes
Root Port Mode	32-bit Memory	16 bytes – 2 gigabytes
	64-bit Memory	4 bytes – 8 exabytes
	I/O	16 bytes – 2 gigabytes

- **Prefetchable:** Identifies the ability of the memory space to be prefetched.
- **Value:** The value assigned to the BAR based on the current selections.
- **Managing SRIOV Base Address Register Settings:** Memory, I/O, Type, and Prefetchable settings are handled by setting the appropriate Customize IP dialog box settings for the desired base address register.

Memory or I/O settings indicate whether the address space is defined as memory or I/O. The base address register only responds to commands that access the specified address space. Generally, memory spaces less than 4 KB in size should be avoided. The minimum I/O space allowed is 16 bytes. I/O space should be avoided in all new designs.

A memory space is prefetchable if there are no side effects on reads (that is, data is not destroyed by reading, as from RAM). Byte-write operations can be merged into a single double-word write, when applicable.

When configuring the core as an Endpoint for PCIe (non-Legacy), 64-bit addressing must be supported for all SR-IOV BARs (except BAR5) that have the prefetchable bit set. 32-bit addressing is permitted for all SR-IOV BARs that do not have the prefetchable bit set. The prefetchable bit related requirement does not apply to a Legacy Endpoint. The minimum memory address range supported by a BAR is 128 bytes for a PCI Express Endpoint and 16 bytes for a Legacy PCI Express Endpoint.

- **Disabling Unused Resources:** For best results, disable unused base address registers to conserve system resources. Disable base address register by deselecting unused BARs in the Customize IP dialog box.

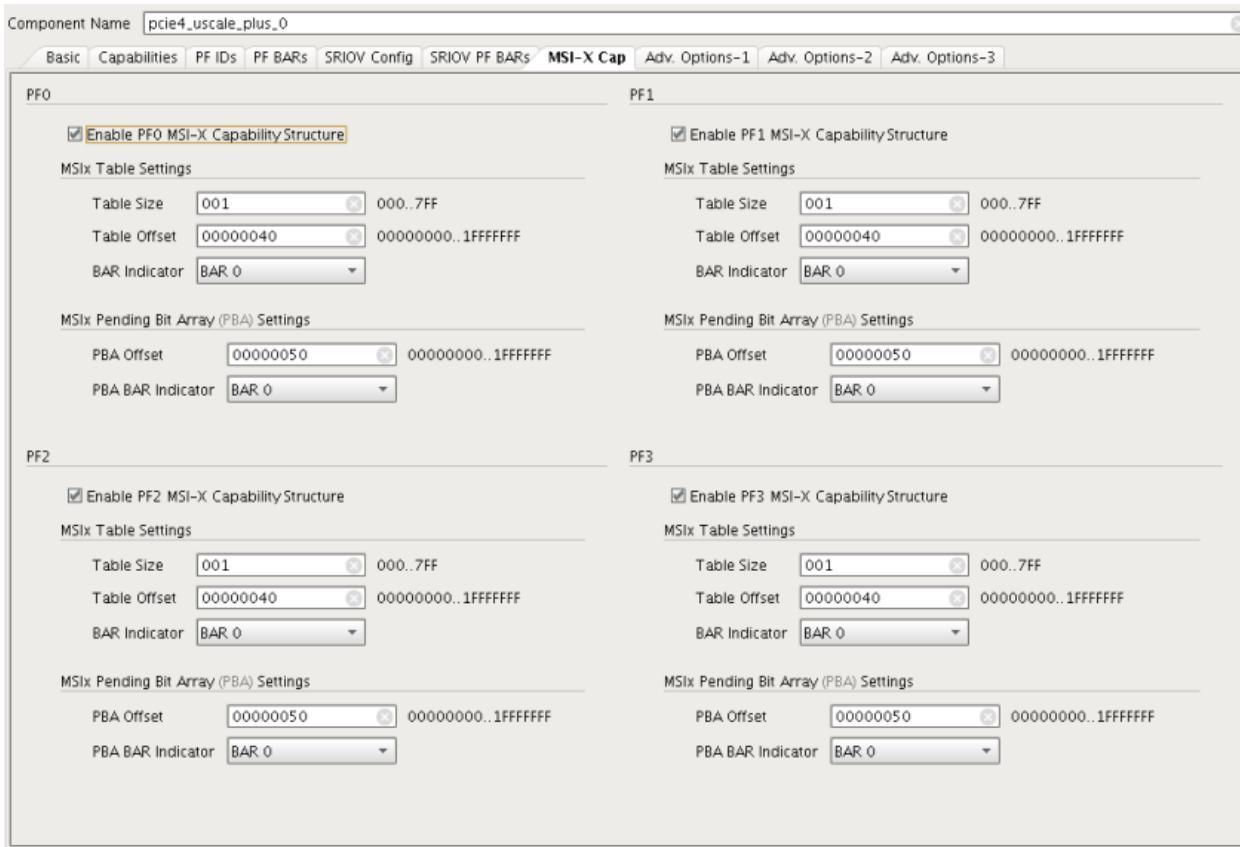
MSI-X Capabilities Tab

The MSI-X Capabilities parameters, shown in the following figure, are available in Advanced mode only. To enable MSI-X capabilities, select **Advanced** mode and then select the required options on the Capabilities page. There are four options to choose from.

- **MSI-X External:** In this mode you need to implement MSI-X External interface driving logic, MSI-X Table and PBA buffers outside the core. You can configure the MSI-X BARs.
- **MSI-X Internal:** In this mode you need to implement the MSI-X Internal interface driving logic only. MSI-X Table and PBA buffers are built into the core. You can configure the MSI-X BARs.
- **MSI-X AXI4-Stream:** In this mode user is expected to drive MSI-X interrupts on the AXI4-Stream interface. You can configure the MSI-X BARs.
- **None:** No MSI-X is supported.

The same MSI-X options are applicable when SRIOV capability is selected.

Figure 122: MSI-X Cap Tab, Advanced Mode



- **Enable MSI-X Capability Structure:** Indicates that the MSI-X Capabilities structure exists.

Note: The Capability Structure needs at least one Memory BAR to be configured. You must maintain the MSI-X Table and Pending Bit Array in the application.

- **MSI-X Table Settings:** Defines the MSI-X Table structure.

- **Table Size:** Specifies the MSI-X Table size. Table Size field is expecting N-1 interrupts (0x0F will configure a table count of 16).
- **Table Offset:** Specifies the offset from the Base Address Register that points to the base of the MSI-X Table.
- **BAR Indicator:** Indicates the Base Address Register in the Configuration Space used to map the function in the MSI-X Table onto memory space. For a 64-bit Base Address Register, this indicates the lower DWORD.

- **MSIx Pending Bit Array (PBA) Settings:** Defines the MSI-X Pending Bit Array (PBA) structure.

- **PBA Offset:** Specifies the offset from the Base Address Register that points to the base of the MSI-X PBA.

- **PBA BAR Indicator:** Indicates the Base Address Register in the Configuration Space used to map the function in the MSI-X PBA onto Memory Space.

Adv. Options

The Advanced Options tab enables Power Management Registers and ASPM. There is no L0s support when the link speed is not 2.5 Gb/s and 5.0 Gb/s.

Figure 123: Adv. Options-1 Tab

Component Name	
pcie4_uscale_plus_0	
Basic	Capabilities
PF IDs	PF BARs
Legacy/MSI Cap	Adv. Options-1
Adv. Options-2	Adv. Options-3
GT Settings	

Power Management Registers

D1 Support

PME Support

D0 D1 D3hot

ASPM Support optionality

No ASPM

L0s Supported

L1 Supported

Adv. Options-2

The Advanced Options tab enables you to choose AER, ARI, and DSN capabilities for the core.

Figure 124: Adv. Options-2 Tab

Component Name pcie4_uscale_plus_0

Basic Capabilities PF IDs PF BARs Legacy/MSI Cap Adv. Options-1 **Adv. Options-2** Adv. Options-3 GT Settings Shared Log

AER Capabilities

The Advanced Error Reporting(AER) Capability is an optional PCIe Extended Capability, which when enabled, allows advanced error control and reporting

Enable AER Capability (PFx) ECRC Check Capable (PFx)

ARI Capability

The Alternative Routing ID-Interpretation (ARI) Capability is an optional PCIe Extended Capability, which when enabled, allows a device to support up to 256 functions by reducing the ID from 3 field vector (Bus Number, Device Number, Function Number) to a 2 field vector (Bus Number, Function Number)

Enable ARI Capability (PFx)

Device Serial Number Capability

The Device Serial Number (DSN) Capability is an optional PCIe Extended Capability, that contains a unique Device Serial Number. This identifier must be presented on the Device Serial Number Input pin of port.

Enable DSN Capability (PFx)

Adv. Options-3

The Advanced Options tab enables you to choose TPH, VC, and user-defined capabilities for the core.

Figure 125: Adv. Options-3 Tab

Component Name pcie4_uscale_plus_0

Basic Capabilities PF IDs PF BARs Legacy/MSI Cap Adv. Options-1 Adv. Options-2 **Adv. Options-3** GT Settings Shared Logic Add. Debug Options C

TPH Capability

TLP Processing Hints (TPH) is a feature which facilitates optimization of requests targeting Memory Space. The TLP Processing Hints Capability is required for all functions capable of generating request TLP's with TPH.

Enable TPH Capability (PFx)

Virtual Channel Capability

The Virtual Channel (VC) Capability is an optional PCIe Extended Capability, which when enabled, allows the port to support functionality beyond the default Traffic Class (TCx) over the default Virtual Channel (VCx). Checking this allows Traffic Class (TC) filtering to be supported

Enable VC Capability (PFx)

User Defined Configuration Capabilities

PCI Express Extended Configuration Space Enable

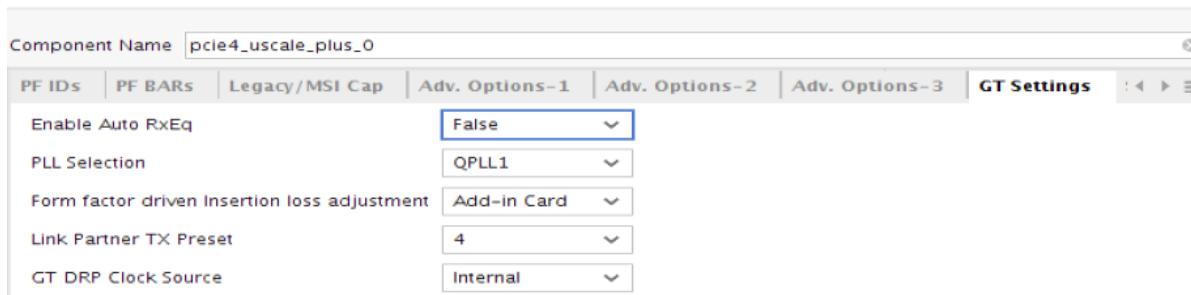
PCI Express Legacy Extended Configuration Space Enable

Add the PCIe XVC-VSEC to the Example Design

GT Settings Tab

Settings in this page allow you to customize specific transceiver settings that are normally not accessible.

Figure 126: GT Settings Tab



- **PLL Selection:** (Only available when Gen2 link speed is selected), allows for either the QPLL1 or CPLL to be selected as the clock source. This feature is useful when additional protocols are desired to be in the same GT Quad when operating at Gen2 links speeds. Gen3 speeds require the QPLL1, and Gen1 speeds always use the CPLL.



IMPORTANT! *The rest of the settings should not be modified unless instructed to do so by Xilinx.*

The following table shows the options and default for each line speed.

Table 70: PLL Type

Link Speed	PLL Type	Comments
2.5_GT/s	CPLL	The default is CPLL, and not available for selection.
5.0_GT/s	QPLL1, CPLL	The default is QPLL1, and available for selection.
8.0_GT/s	QPLL1	The default is QPLL1, and not available for selection.

- **Enable Auto RxEq:** When this parameter is set to True, it auto select the Receiver Equalization (LPM or DFE) mode.
 - **True:** The default is DFE, but it will change LPM based on the channel characteristics.
 - **False:** The default is DFE and can be changed by setting the Form Factor Driven Insertion Loss Adjustment.
- **Form Factor Driven Insertion Loss Adjustment:** Indicates the transmitter to receiver insertion loss at the Nyquist frequency depending on the form factor selection. Three options are provided:
 - **Chip-to-Chip:** The value is 5 dB.
 - **Add-in Card:** The value is 15 dB and is the default option.
 - **Backplane:** The value is 20 dB.

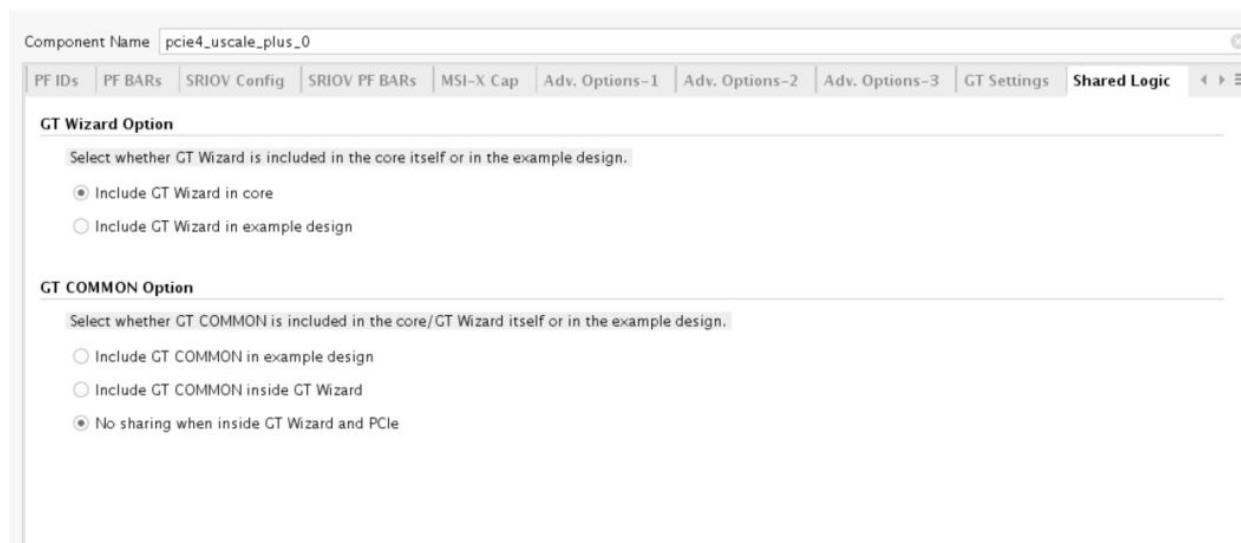
These insertion loss values are applied to the GT Wizard subcore.

- **Link Partner TX Preset:** It is not advisable to change the default value of 4. Preset value of 5 might work better on some systems. This parameter is available on GT Settings tab.
- **GT DRP Clock Source:** This option is added to select the GT clock source external or internal. When External is selected, the DRP Clock is supplied from an external clock source of 300 MHz, and it will be divided into 100/125 MHz in the Xilinx top module. The default GT DRP clock source is Internal but is External when Enable In System IBERT is enabled in the Add. Debug Options page.

Shared Logic Tab

The following figure shows the Shared Logic tab.

Figure 127: Shared Logic



- **GT Wizard Option:** You can select include GT Wizard in the example design and then the GT Wizard IP will be delivered into the example design area. You can reconfigure the IP for further testing purposes. By default, the GT Wizard IP will be delivered in the PCIe IP core as a hierarchical IP and you cannot re-customize it. For signal descriptions and for other details, see the *UltraScale Architecture GTY Transceivers User Guide (UG578)* or *UltraScale Architecture GTH Transceivers User Guide (UG576)*.
- **GT COMMON Option:** This option is used to share the GT COMMON block used in the design when Gen2 (PLL Selection is **QPLL1**) and Gen3 link speeds are selected.
 - When **Include GT COMMON in example design** is selected, GT common block instance will be available in the support wrapper, which is inside the Xilinx top file and can be used either by the core or the external logic.
 - When **Include GT COMMON inside GT Wizard** is used, GT COMMON can be shared by external logic.

- When **No Sharing when inside GT Wizard** and **PCIe** is selected, no sharing of GT COMMON block is allowed.
- When **Include GT COMMON in example design** and **Include GT Wizard in example design** are selected together, you must use the latest GT COMMON settings from the example design project of the GT Wizard IP of the same configuration. This specific option delivers static GT COMMON wrappers which have the latest settings for the 2017.3 time frame only.

Add. Debug Options

You can select additional debug portions for debugging purposes. The parameters are described below.

- **Enable In System IBERT:** This debug option is used to check and see the eye diagram of the serial link at the desired link speed. For more information on In System IBERT, refer to *In-System IBERT LogiCORE IP Product Guide* ([PG246](#)).



IMPORTANT! *This option is used mainly for hardware debug purposes. Simulations are not supported when this option is used.*

Steps to check the eye diagram:

1. Select a suitable Xilinx reference board.
2. Configure the core with the following options:
 - Select **Gen3**, **Gen2**, or **Gen1** link speed at any link width.
 - Select **Enable In System IBERT** in the Add. Debug Options page.
3. Open the Example Design.
4. Generate a **.bit** file and **.ltx** file.
5. Before programming the FPGA, source the following command in the Tcl Console.

```
- set_param xicom.enable_isi_PCIE_fix 1
```
6. Open Hardware Manager (HM) and configure the device using the generated **.bit** and **.ltx** file.
7. Reboot the machine to rescan and run the enumeration process again.
8. Select the Serial I/O links tab at the bottom of the HM, and create links for the scan window.
9. Select any one of the links in Serial I/O links tab, and right-click and choose scan link option.
10. For better results, try Horizontal and Vertical increment by two instead of the default value
11. After the eye scan is selected, the eye diagram will be plotted.

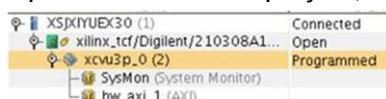


IMPORTANT! Enable In System IBERT should not be used with the Falling Edge Receiver Detect option in GT Settings tab. The Enable In System IBERT option in the Add. Debug Options tab also sets the GT DRP Clock Source to External in the GT Settings tab.

- **Enable Descrambler for Gen3 Mode:** This debug option integrates encrypted version of the descrambler module inside the PCIe core, which will be used to descramble the PIPE data to/from PCIe integrated block in Gen3 link speed mode.
- **Enable JTAG Debugger:** This feature provides ease of debug for the following:
 - **LTSSM state transitions:** This shows all the LTSSM state transitions that have been made starting from link up.
 - **PHY Reset FSM transitions:** This shows the PHY reset FSM (internal state machine that is used by the PCIe solution IP).
 - **Receiver Detect:** This shows all the lanes that have completed Receiver Detect Successfully

Steps are the following:

1. Open a new Vivado project, and connect to the board. You should now see hw_axi_1.



2. In the Vivado Tcl Console, type `source test_rd.tcl`.

3. For post-processing, double-click the following:

- `draw_ltssm.tcl` (Windows) or `wish draw_ltssm.tcl`
- `draw_reset.tcl` (Windows) or `wish draw_reset.tcl`
- `draw_rxdet.tcl` (Windows) or `wish draw_rxdet.tcl`

This displays the pictorial representation of the LTSSM state transitions.

Related Information

[Hardware Debug](#)

Core Interface Parameters Tab

The Core Interface Parameters tab enables you to disable interfaces that are not required for your application.

To support example design simulations, enable:

- **Config Management Interface** to support Endpoint example design simulations.
- **Config Control Interface** to support Root Port example design simulations.

Figure 128: Core Interface Parameters Tab



Output Generation

For details, see the *Vivado Design Suite User Guide: Designing with IP* ([UG896](#)).

Constraining the Core

Required Constraints

The UltraScale+ Devices Integrated Block for PCI Express solution requires the specification of timing and other physical implementation constraints to meet specified performance requirements for PCI Express®. These constraints are provided with the Endpoint and Root Port solutions in a Xilinx Design Constraints (XDC) file. Pinouts and hierarchy names in the generated XDC correspond to the provided example design.



IMPORTANT! If the example design top file is not used, copy the IBUFDS instance for the reference clock, IBUF Instance for sys_rst and also the location and timing constraints associated with them into your local design top.

To achieve consistent implementation results, an XDC containing these original, unmodified constraints must be used when a design is run through the Xilinx tools. For additional details on the definition and use of an XDC or specific constraints, see *Vivado Design Suite User Guide: Using Constraints* ([UG903](#)).

Constraints provided with the integrated block solution have been tested in hardware and provide consistent results. Constraints can be modified, but modifications should only be made with a thorough understanding of the effect of each constraint. Additionally, support is not provided for designs that deviate from the provided constraints.

Device, Package, and Speed Grade Selections

The device selection portion of the XDC informs the implementation tools which part, package, and speed grade to target for the design.



IMPORTANT! Because the XDC is generated for specific part and package combinations, this section should not be modified.

The device selection section always contains a part selection line, but can also contain part or package-specific options. An example part selection line follows:

```
CONFIG PART = XCKU040-ffva1156-3-e-es1
```

Clock Frequencies

See [Chapter 4: Designing with the Core](#), for detailed information about clock requirements.

Clock Management

See [Chapter 4: Designing with the Core](#), for detailed information about clock requirements.

Clock Placement

See [Chapter 4: Designing with the Core](#), for detailed information about clock requirements.

Banking

This section is not applicable for this IP core.

Transceiver Placement

This section is not applicable for this IP core.

I/O Standard and Placement

This section is not applicable for this IP core.

Relocating the Integrated Block Core

By default, the IP core-level constraints lock block RAMs, UltraRAMs, transceivers, and the integrated block to the recommended location. To relocate these blocks, you must override the constraints for these blocks in the XDC constraint file. To do so:

1. Copy the constraints for the block that needs to be overwritten from the core-level XDC constraint file.
2. Place the constraints in the user XDC constraint file.

3. Update the constraints with the new location.

The user XDC constraints are usually scoped to the top-level of the design; therefore, ensure that the cells referred by the constraints are still valid after copying and pasting them. Typically, you need to update the module path with the full hierarchy name.

Note: If there are locations that need to be swapped (that is, the new location is currently being occupied by another module), there are two ways to do this.

- If there is a temporary location available, move the first module out of the way to a new temporary location first. Then, move the second module to the location that was occupied by the first module. Next, move the first module to the location of the second module. These steps can be done in XDC constraint file.
- If there is no other location available to be used as a temporary location, use the `reset_property` command from Tcl command window on the first module before relocating the second module to this location. The `reset_property` command cannot be done in XDC constraint file and must be called from the Tcl command file or typed directly into the Tcl Console.

Simulation

For comprehensive information about Vivado® simulation components, as well as information about using supported third-party tools, see the *Vivado Design Suite User Guide: Logic Simulation (UG900)*.

For more information regarding simulating the example design, see [Simulating the Example Design](#).

PIPE Mode Simulation

The Integrated Block for PCIe supports the PIPE mode simulation where the PIPE interface of the core is connected to the PIPE interface of the link partner. This mode increases the simulation speed.

Use the **Enable External PIPE Interface** option on the Basic tab of the Customize IP dialog box to enable PIPE mode simulation in the current Vivado Design Suite solution example design, in either Endpoint mode or Root Port mode. The External PIPE Interface signals are generated at the core boundary for access to the external device. Enabling this feature also provides the necessary hooks to use third-party PCI Express® VIPs/BFMs instead of the Root Port model provided with the example design.



TIP: PIPE mode is for simulation only. Implementation is not supported.

For details, see [Enable External PIPE Interface](#) option in the [Basic Tab](#).

The tables below describe the PIPE bus signals available at the top level of the core and their corresponding mapping inside the EP core (`pcie_top`) PIPE signals.



IMPORTANT! A new file, `xil_sig2pipe.v`, is delivered in the simulation directory, and the file replaces `phy_sig_gen.v`. BFM/VIPs should interface with the `xil_sig2pipe` instance in `board.v`.

Table 71: Common In/Out Commands and Endpoint PIPE Signals Mappings

In Commands	Endpoint PIPE Signals Mapping	Out Commands	Endpoint PIPE Signals Mapping
common_commands_in[25:0]	not used	common_commands_out[0]	pipe_clk ¹
		common_commands_out[2:1]	pipe_tx_rate_gt ²
		common_commands_out[3]	pipe_tx_rcvr_det_gt
		common_commands_out[6:4]	pipe_tx_margin_gt
		common_commands_out[7]	pipe_tx_swing_gt
		common_commands_out[8]	pipe_tx_reset_gt
		common_commands_out[9]	pipe_tx_deemph_gt
		common_commands_out[16:10]	not used ³

Notes:

1. `pipe_clk` is an output clock based on the core configuration. For Gen1 rate, `pipe_clk` is 125 MHz. For Gen2 and Gen3, `pipe_clk` is 250 MHz.
2. `pipe_tx_rate_gt` indicates the pipe rate (2'b00-Gen1, 2'b01-Gen2, and 2'b10-Gen3).
3. The functionality of this port has been deprecated and it can be left unconnected.

Table 72: Input/Output Buses with Endpoint PIPE Signals Mapping

Input Bus	Endpoint PIPE Signals Mapping	Output Bus	Endpoint PIPE Signals Mapping
pipe_rx_0_sigs[31:0]	pipe_rx0_data_gt	pipe_tx_0_sigs[31: 0]	pipe_tx0_data_gt
pipe_rx_0_sigs[33:32]	pipe_rx0_char_is_k_gt	pipe_tx_0_sigs[33:32]	pipe_tx0_char_is_k_gt
pipe_rx_0_sigs[34]	pipe_rx0_elec_idle_gt	pipe_tx_0_sigs[34]	pipe_tx0_elec_idle_gt
pipe_rx_0_sigs[35]	pipe_rx0_data_valid_gt	pipe_tx_0_sigs[35]	pipe_tx0_data_valid_gt
pipe_rx_0_sigs[36]	pipe_rx0_start_block_gt	pipe_tx_0_sigs[36]	pipe_tx0_start_block_gt
pipe_rx_0_sigs[38:37]	pipe_rx0_synchheader_gt	pipe_tx_0_sigs[38:37]	pipe_tx0_synchheader_gt
pipe_rx_0_sigs[83:39]	not used	pipe_tx_0_sigs[39]	pipe_tx0_polarity_gt
		pipe_tx_0_sigs[41:40]	pipe_tx0_powerdown_gt
		pipe_tx_0_sigs[69:42]	not used ¹

Notes:

1. The functionality of this port has been deprecated and it can be left unconnected.

Synthesis and Implementation

For details about synthesis and implementation, see the *Vivado Design Suite User Guide: Designing with IP (UG896)*.

For information regarding synthesizing and implementing the example design, see [Synthesizing and Implementing the Example Design](#).

Example Design

This chapter contains information about the example design provided in the Vivado® Design Suite.

Overview of the Example Design

This section provides an overview of the UltraScale+ Devices Integrated Block for PCI Express core example design.

Integrated Block Endpoint Configuration Overview

The example simulation design for the Endpoint configuration of the integrated block consists of two discrete parts:

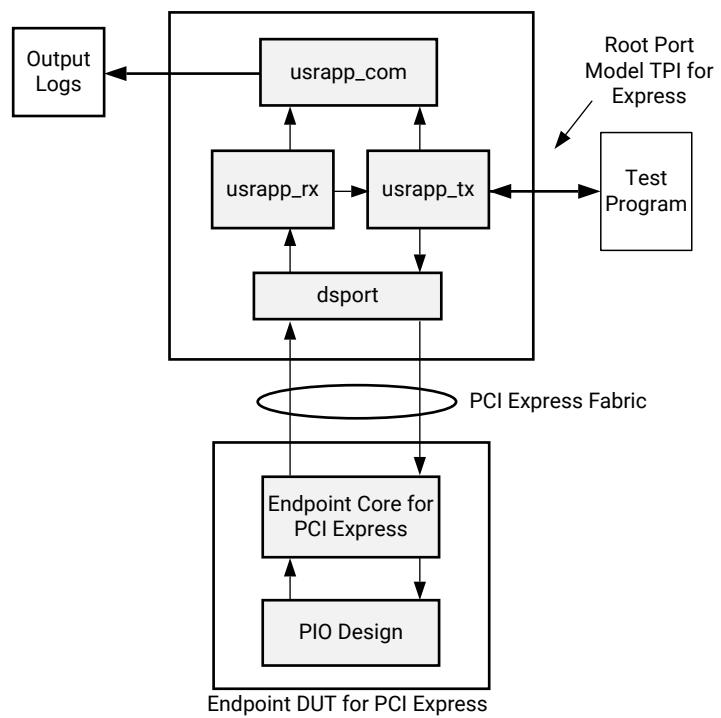
- The Root Port Model, a test bench that generates, consumes, and checks PCI Express® bus traffic.
- The Programmed Input/Output (PIO) example design, a completer application for PCI Express. The PIO example design responds to Read and Write requests to its memory space and can be synthesized for testing in hardware.

Note: Not all modes have example design support, for example, Straddle, Address aligned mode, SRIOV, MSI-X, FLR, and MSI. The FLR for virtual functions is not fully implemented in the example design; for this to work, you need to modify the user logic.

Simulation Design Overview

For the simulation design, transactions are sent from the Root Port Model to the core (configured as an Endpoint) and processed by the PIO example design. The following figure illustrates the simulation design provided with the core. For more information about the Root Port Model, see [Root Port Model Test Bench for Endpoint](#).

Figure 129: Simulation Example Design Block Diagram

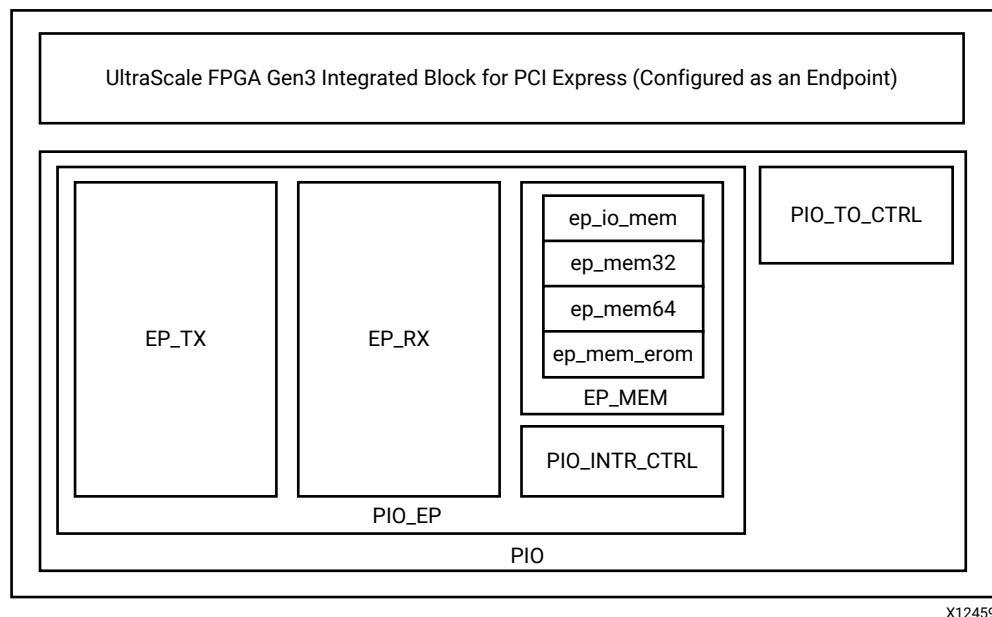


X12471

Implementation Design Overview

The implementation design consists of a simple PIO example that can accept read and write transactions and respond to requests, as illustrated in the figure below. Source code for the example is provided with the core. For more information about the PIO example design, see [Programmed Input/Output: Endpoint Example Design](#).

Figure 130: Implementation Example Design Block Diagram



Example Design Elements

The PIO example design elements include:

- Core wrapper
- An example Verilog HDL wrapper (instantiates the cores and example design)
- A customizable demonstration test bench to simulate the example design

The example design has been tested and verified with Vivado Design Suite and these simulators:

- Vivado simulator
- Questa Advanced Simulator
- Synopsys Verilog Compiler Simulator (VCS)

For the supported versions of these tools, see the [Xilinx Design Tools: Release Notes Guide](#).

Programmed Input/Output: Endpoint Example Design

Programmed Input/Output (PIO) transactions are generally used by a PCI Express® system host CPU to access Memory Mapped Input/Output (MMIO) and Configuration Mapped Input/Output (CMIO) locations in the PCI Express logic. Endpoints for PCI Express accept Memory and I/O Write transactions and respond to Memory and I/O Read transactions with Completion with Data transactions.

The PIO example design (PIO design) is included with the core in Endpoint configuration generated by the Vivado® IP catalog, which allows you to bring up your system board with a known established working design to verify the link and functionality of the board.

The PIO design Port Model is shared by the core, Endpoint Block Plus for PCI Express, and Endpoint PIPE for PCI Express solutions. This section generically represents all solutions using the name Endpoint for PCI Express (or Endpoint for PCIe®).

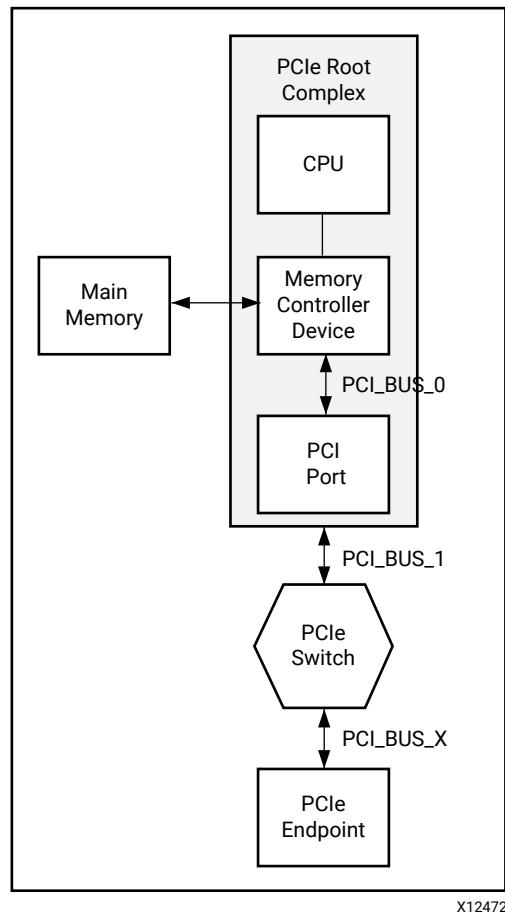
System Overview

The PIO design is a simple target-only application that interfaces with the Endpoint for the PCIe core Transaction (AXI4-Stream) interface and is provided as a starting point for you to build your own designs. These features are included:

- In Address Align Mode, four transaction-specific 2 KB target regions using the internal FPGA block RAMs, providing a total target space of 8,192 bytes.
- In Address Align Mode, supports single Dword payload Read and Write PCI Express transactions to 32-/64-bit address memory spaces and I/O space with support for completion TLPs.
- In the case of Dword Align Mode, the PIO Design supports multiple Dword payload (Up to 256 DW) read and write PCI Express transactions to 32-bit Address Memory Spaces with support for completion TLPs. Utilizes the BAR ID[2:0] and Completer Request Descriptor[114:112] of the core to differentiate between TLP destination Base Address Registers.
- Provides separate implementations optimized for 64-bit, 128-bit, 256-bit, and 512-bit AXI4-Stream interfaces.

The following figure illustrates the PCI Express system architecture components, consisting of a Root Complex, a PCI Express switch device, and an Endpoint for PCIe. PIO operations move data downstream from the Root Complex (CPU register) to the Endpoint, and/or upstream from the Endpoint to the Root Complex (CPU register). In either case, the PCI Express protocol request to move the data is initiated by the host CPU.

Figure 131: System Overview



X12472

Data is moved downstream when the CPU issues a store register to a MMIO address command. The Root Complex typically generates a Memory Write TLP with the appropriate MMIO location address, byte enables, and the register contents. The transaction terminates when the Endpoint receives the Memory Write TLP and updates the corresponding local register.

Data is moved upstream when the CPU issues a load register from a MMIO address command. The Root Complex typically generates a Memory Read TLP with the appropriate MMIO location address and byte enables. The Endpoint generates a Completion with Data TLP after it receives the Memory Read TLP. The Completion is steered to the Root Complex and payload is loaded into the target register, completing the transaction.

PIO Hardware

For Address Align Mode, the PIO design implements an 8,192 byte target space in FPGA block RAM, behind the Endpoint for PCIe. This 32-bit target space is accessible through single Dword I/O Read, I/O Write, Memory Read 64, Memory Write 64, Memory Read 32, and Memory Write 32 TLPs.

The PIO design generates a completion with one Dword of payload in response to a valid Memory Read 32 TLP, Memory Read 64 TLP, or I/O Read TLP request presented to it by the core. In addition, the PIO design returns a completion without data with successful status for I/O Write TLP request. For Dword Align Mode, the PIO design implements 2048-byte target space in FPGA block RAM. This target space, and data width varies based on the AXI4-Stream interface and is equal to the width of the AXI4-Stream interface. This target space is accessible through Memory Write 32 and Memory Read 32 TLPs.

The PIO generates a completion with the payload size in response to a valid Memory Read 32 TLP request from the core.

The PIO design can initiate the following:

- a Memory Read transaction when the received write address is `11'hEA8` and the write data is `32'hAAAA_BBBB`, and targeting the BAR0.
- a Legacy Interrupt when the received write address is `11'hEEC` and the write data is `32'hCCCC_DDDD`, and targeting the BAR0.
- an MSI when the received write address is `11'hEEC` and the write data is `32'hEEEE_FFFF`, and targeting the BAR0.
- an MSIX when the received write address is `11'hEEC` and the write data is `32'hDEAD_BEEF`, and targeting the BAR0.

The PIO design processes a Memory or I/O Write TLP with one Dword payload in case of address align mode and multi-Dword in case of Dword Align Mode by updating the payload into the target address in the FPGA block RAM space.

Base Address Register Support

In case of Address Align Mode, the PIO design supports four discrete target spaces, each consisting of a 2 KB block of memory represented by a separate Base Address Register (BAR). Using the default parameters, the Vivado® IP catalog produces a core configured to work with the PIO design defined in this section, consisting of:

- One 64-bit addressable Memory Space BAR
- One 32-bit Addressable Memory Space BAR

You can change the default parameters used by the PIO design; however, in some cases you might need to change the user application depending on your system. See [Changing IP Catalog Tool Default BAR Settings](#) for information about changing the default Vivado Design Suite IP parameters and the effect on the PIO design.

Each of the four 2 KB address spaces represented by the BARs corresponds to one of four 2 KB address regions in the PIO design. Each 2 KB region is implemented using a 2 KB dual-port block RAM. As transactions are received by the core, the core decodes the address and determines which of the four regions is being targeted. The core presents the TLP to the PIO design and asserts the appropriate bits of (BAR ID[2:0]), Completer Request Descriptor[114:112], as defined in the table below.

Table 73: TLP Traffic Types

Block RAM	TLP Transaction Type	Default BAR	BAR ID[2:0]
ep_io_mem	I/O TLP transactions	Disabled	Disabled
ep_mem32	32-bit address Memory TLP transactions	2	000b
ep_mem64	64-bit address Memory TLP transactions	0-1	001b
ep_mem_erom	32-bit address Memory TLP transactions destined for EROM	Expansion ROM	110b

For Dword Align Mode, the PIO design supports one target space, consisting of 2048 Bytes of memory. The memory is implemented using SDRAM. As transaction are received by the core, the core presents the TLP to the PIO design and asserts the bits of BAR ID[2:0],and completer request descriptor [114:112] as 001b.

Changing IP Catalog Tool Default BAR Settings

You can change the Vivado® IP catalog parameters and continue to use the PIO design to create customized Verilog source to match the selected BAR settings. However, because the PIO design parameters are more limited than the core parameters, consider the following example design limitations when changing the default IP catalog parameters:

- The example design supports one I/O space BAR, one 32-bit Memory space (that cannot be the Expansion ROM space), and one 64-bit Memory space. If these limits are exceeded, only the first space of a given type is active—accesses to the other spaces do not result in completions.
- Each space is implemented with a 2 KB memory. If the corresponding BAR is configured to a wider aperture, accesses beyond the 2 KB limit wrap around and overlap the 2 KB memory space.
- The PIO design supports one I/O space BAR, which by default is disabled, but can be changed if desired.

Although there are limitations to the PIO design, Verilog source code is provided so you can tailor the example design to your specific needs.

TLP Data Flow

This section defines the data flow of a TLP successfully processed by the PIO design.

The PIO design successfully processes single Dword payload Memory Read and Write TLPs for Address Align Mode and multi-Dword payload in case of Dword Align Mode and I/O Read and Write TLPs supported only for Address Align Mode. Memory Read or Memory Write TLPs of lengths larger than one Dword are not processed correctly by the PIO design. In case of Address Align Mode, however, the core does accept these TLPs and passes them along to the PIO design. If the PIO design receives a TLP with a length of greater than one Dword, the TLP is received completely from the core and discarded. No corresponding completion is generated. For Dword Align Mode, payload containing multiple Dword for Memory Read and Memory Write TLPs are supported and are processed correctly by the PIO design. The TLP is received completely from the core and then corresponding completion is generated.

Memory and I/O Write TLP Processing

When the Endpoint for PCIe® receives a Memory or I/O Write TLP, the TLP destination address and transaction type are compared with the values in the core BARs. If the TLP passes this comparison check, the core passes the TLP to the Receive AXI4-Stream interface of the PIO design. The PIO design handles Memory writes and I/O TLP writes in different ways: the PIO design responds to I/O writes by generating a Completion Without Data (cpl), a requirement of the PCI Express specification.

Along with the start of packet, end of packet, and ready handshaking signals, the Completer Requester AXI4-Stream interface also asserts the appropriate (BAR ID[2:0]), Completer Request Descriptor[114:112] signal to indicate to the PIO design the specific destination BAR that matched the incoming TLP. On reception, the PIO design RX State Machine processes the incoming Write TLP and extracts the TLPs data and relevant address fields so that it can pass this along to the PIO design internal block RAM write request controller.

In case of Address align mode, based on the specific BAR ID[2:0] signals asserted, the RX state machine indicates to the internal write controller the appropriate 2 KB block RAM to use prior to asserting the write enable request. For example, if an I/O Write Request is received by the core targeting BAR0, the core passes the TLP to the PIO design and sets BAR ID[2:0] to 000b. The RX state machine extracts the lower address bits and the data field from the I/O Write TLP and instructs the internal Memory Write controller to begin a write to the block RAM.

While in case of Dword Align mode, when the BAR ID[2:0] = 01b, the RX state machine asserts the write enable request. The RX state machine extracts the lower address bits and the data from the Memory 32 Write TLP and instructs the internal memory write controller to begin a write to the block RAM.

In this example, the assertion of setting BAR ID[2:0] to 000b instructed the PIO memory write controller to access `ep_mem0` (which by default represents 2 KB of I/O space). While the write is being carried out to the FPGA block RAM, the PIO design RX state machine deasserts `m_axis_cq_tready`, causing the Receive AXI4-Stream interface to stall receiving any further TLPs until the internal Memory Write controller completes the write to the block RAM. Deasserting `m_axis_cq_tready` in this way is not required for all designs using the core; the PIO design uses this method to simplify the control logic of the RX state machine.

Memory and I/O Read TLP Processing

When the Endpoint for PCIe® receives a Memory or I/O Read TLP, the TLP destination address and transaction type are compared with the values programmed in the core BARs. If the TLP passes this comparison check, the core passes the TLP to the Receive AXI4-Stream interface of the PIO design.

Along with the start of packet, end of packet, and ready handshaking signals, the Completer Requester AXI4-Stream interface also asserts the appropriate BAR ID[2:0] signal to indicate to the PIO design the specific destination BAR that matched the incoming TLP. On reception, the PIO design state machine processes the incoming Read TLP and extracts the relevant TLP information and passes it along to the internal block RAM read request controller of the PIO design.

In case of Address Align Mode, based on the specific BAR ID[2:0] signal asserted, the RX state machine indicates to the internal read request controller the appropriate 2 KB block RAM to use before asserting the read enable request. While for Dword Align Mode, the RX state machine checks if the request is for Memory Read 32 TLP based on the BAR ID [2:0] to enable the read request and discard all the other request. For example, if a Memory Read 32 Request TLP is received by the core targeting the default Mem32 BAR2, the core passes the TLP to the PIO design and sets BAR ID[2:0] to 010b. The RX state machine extracts the lower address bits from the Memory 32 Read TLP and instructs the internal Memory Read Request controller to start a read operation.

In this example, the setting BAR ID[2:0] to 010b instructs the PIO memory read controller to access the Mem32 space, which by default represents 2 KB of memory space. A notable difference in handling of memory write and read TLPs is the requirement of the receiving device to return a Completion with Data TLP in the case of memory or I/O read request.

While the read is being processed, the PIO design RX state machine deasserts `m_axis_cq_tready`, causing the Receive AXI4-Stream interface to stall receiving any further TLPs until the internal Memory Read controller completes the read access from the block RAM and generates the completion. Deasserting `m_axis_cq_tready` in this way is not required for all designs using the core. The PIO design uses this method to simplify the control logic of the RX state machine.

PIO File Structure

The table below defines the PIO design file structure. Based on the specific core targeted, not all files delivered by the Vivado® IP catalog are necessary, and some files might not be delivered. The major difference is that some of the Endpoint for PCIe® solutions use a 32-bit user datapath, others use a 64-bit datapath, and the PIO design works with both. The width of the datapath depends on the specific core being targeted.

Table 74: PIO Design File Structure

File	Description
PIO.v	Top-level design wrapper
PIO_INTR_CTRL.v	PIO interrupt controller
PIO_EP.v	PIO application module
PIO_TO_CTRL.v	PIO turn-off controller module
PIO_RX_ENGINE.v	32-bit Receive engine
PIO_TX_ENGINE.v	32-bit Transmit engine
PIO_EP_MEM_ACCESS.v	Endpoint memory access module
PIO_EP_MEM.v	Endpoint memory
PIO_EP_XPM_SDRAM_WRAP.v	Endpoint Memory in case of dword align mode

Four configurations of the PIO design are provided: PIO_64, PIO_128, and PIO_256 with 64-, 128-, 256-bit, and 512-bit AXI4-Stream interfaces, respectively. The PIO configuration that is generated depends on the selected Endpoint type, the number of PCI Express lanes, and the interface width selected. The following table identifies the PIO configuration generated based on your selection.

Table 75: PIO Configuration

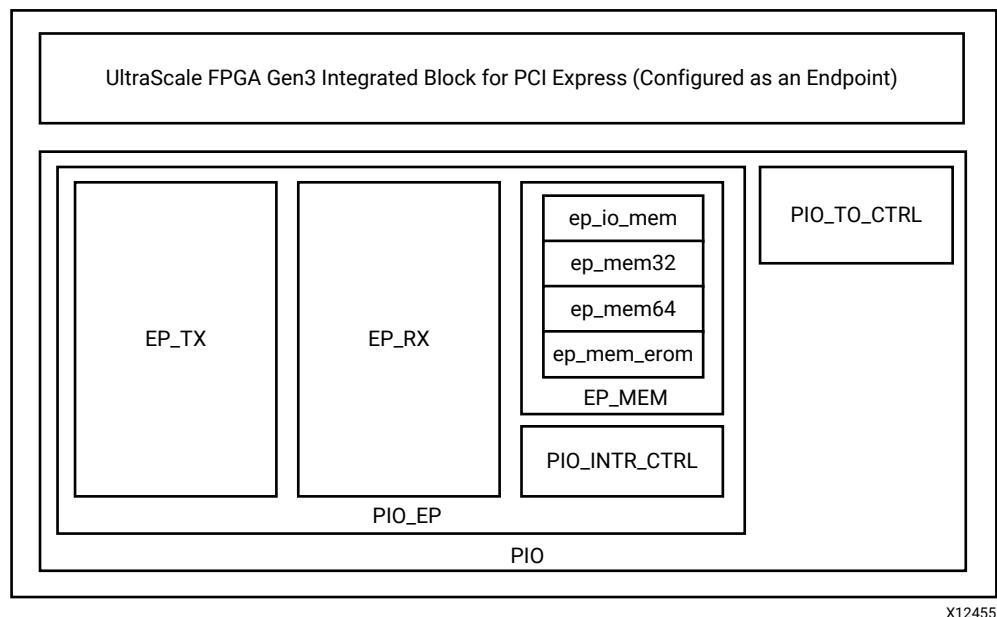
Core	x1	x2	x4	x8
Integrated Block for PCIe	PIO_64	PIO_64, PIO_128	PIO_64, PIO_128, PIO_256	PIO_64, PIO_128 ¹ , PIO_256

Notes:

1. The core does not support 128-bit x8 8.0 Gb/s configuration and 500 MHz user clock frequency.

The following figure shows the various components of the PIO design, which is separated into four main parts: the TX Engine, RX Engine, Memory Access Controller, and Power Management Turn-Off Controller.

Figure 132: PIO Design Components

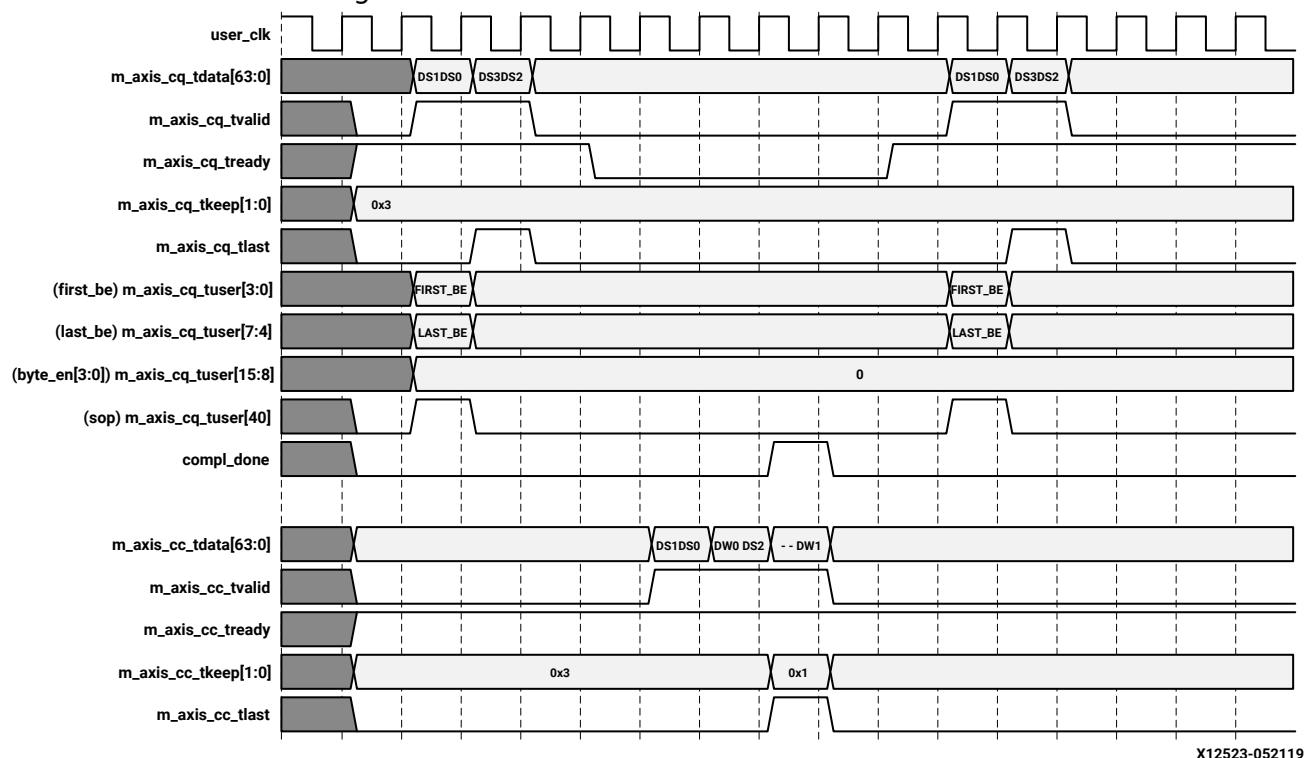


PIO Operation

PIO Read Transaction

The figure below depicts a Back-to-Back Memory Read request to the PIO design. The receive engine deasserts `m_axis_rx_tready` as soon as the first TLP is completely received. The next Read transaction is accepted only after `compl_done_o` is asserted by the transmit engine, indicating that Completion for the first request was successfully transmitted.

Figure 133: Back-to-Back Read Transactions

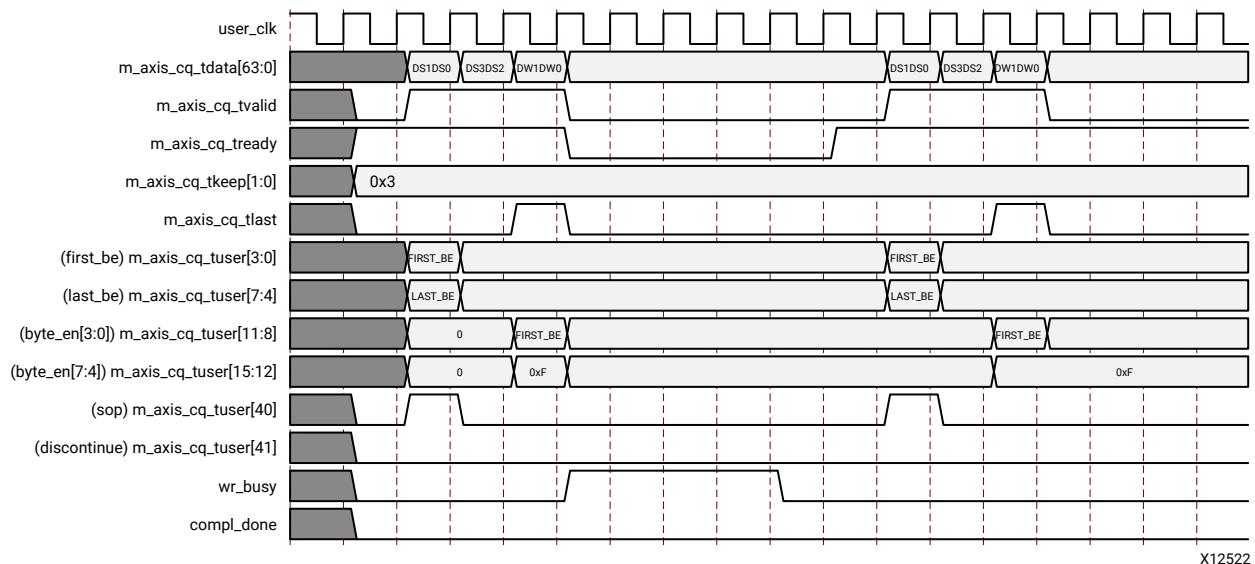


X12523-052119

PIO Write Transaction

The figure below depicts a back-to-back Memory Write to the PIO design. The next Write transaction is accepted only after **wr_busy_o** is deasserted by the memory access unit, indicating that data associated with the first request was successfully written to the memory aperture.

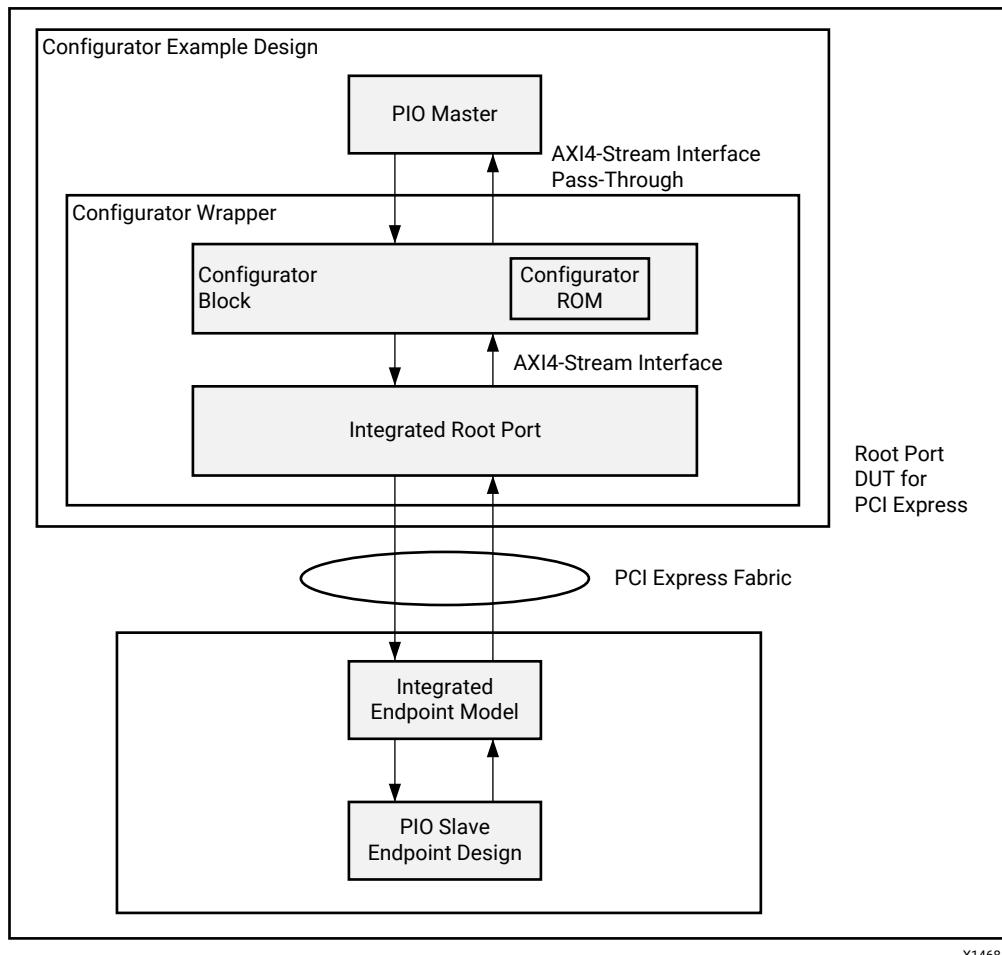
Figure 134: Back-to-Back Write Transactions



Configurator: Rootport Example Design

The following figure shows how the blocks are connected in an overall system view.

Figure 135: Configurator Example Design



X14684

Configurator File Structure

The following table defines the Configurator example design file structure.

Table 76: Example Design File Structure

File	Description
xilinx_pcie4_uscale_rp.v	Top-level wrapper file for Configurator example design
cgator_wrapper.v	Wrapper for Configurator and Root Port
cgator.v	Wrapper for Configurator sub-blocks
cgator_cpl_decoder.v	Completion decoder
cgator_pkt_generator.v	Configuration TLP generator
cgator_tx_mux.v	Transmit AXI4-Stream muxing logic
cgator_gen2_enabler.v	5.0 Gb/s directed speed change module
cgator_controller.v	Configurator transmit engine
cgator_cfg_rom.data	Configurator ROM file

Table 76: Example Design File Structure (cont'd)

File	Description
pio_master.v	Wrapper for PIO Master
pio_master_controller.v	TX and RX Engine for PIO Master
pio_master_checker.v	Checks incoming User-Application Completion TLPs
pio_master_pkt_generator.v	Generates User-Application TLPs

The hierarchy of the Configurator example design is:

`xilinx_pcie_uscale_rp.v`

- `cgator_wrapper`
 - `pcie_uscale_core_top`(in the source directory): This directory contains all the source files for the core in Root Port Configuration.
 - `cgator`
 - `cgator_cpl_decoder`
 - `cgator_pkt_generator`
 - `cgator_tx_mux`
 - `cgator_gen2_enabler`
 - `cgator_controller`: This directory contains `<cgator_cfg_rom.data>` (specified by ROM_FILE).
- `pio_master`
 - `pio_master_controller`
 - `pio_master_checker`
 - `pio_master_pkt_generator`

Note: `cgator_cfg_rom.data` is the default name of the ROM data file. You can override this by changing the value of the ROM_FILE parameter.

Generating the Core

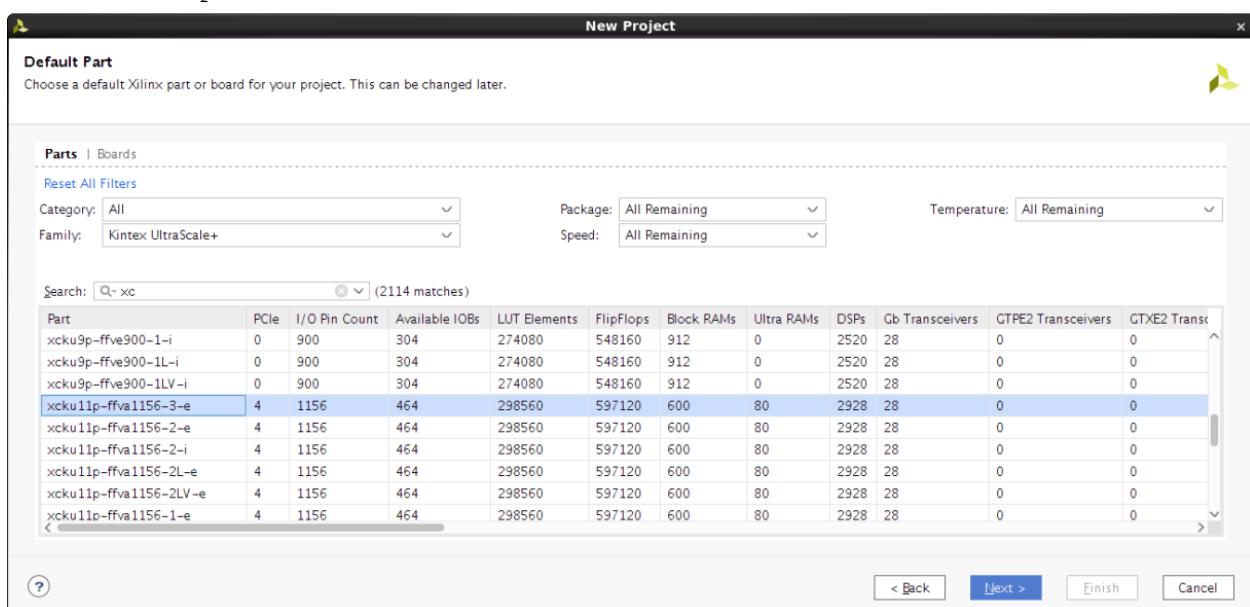
To generate a core using the default values in the Vivado® IDE, follow these steps:

1. Start the Vivado IP catalog.
2. Select **File > Project > New**.
3. Enter a project name and location, then click **Next**. This example uses `project_name.xpr` and `project_dir`.

4. In the New Project wizard pages, do not add sources, existing IP, or constraints.
5. From the Part tab (below), select these filter options:
 - Family: Kintex® UltraScale+™
 - Device: xcku11p
 - Package: ffva1156
 - Speed Grade: -3

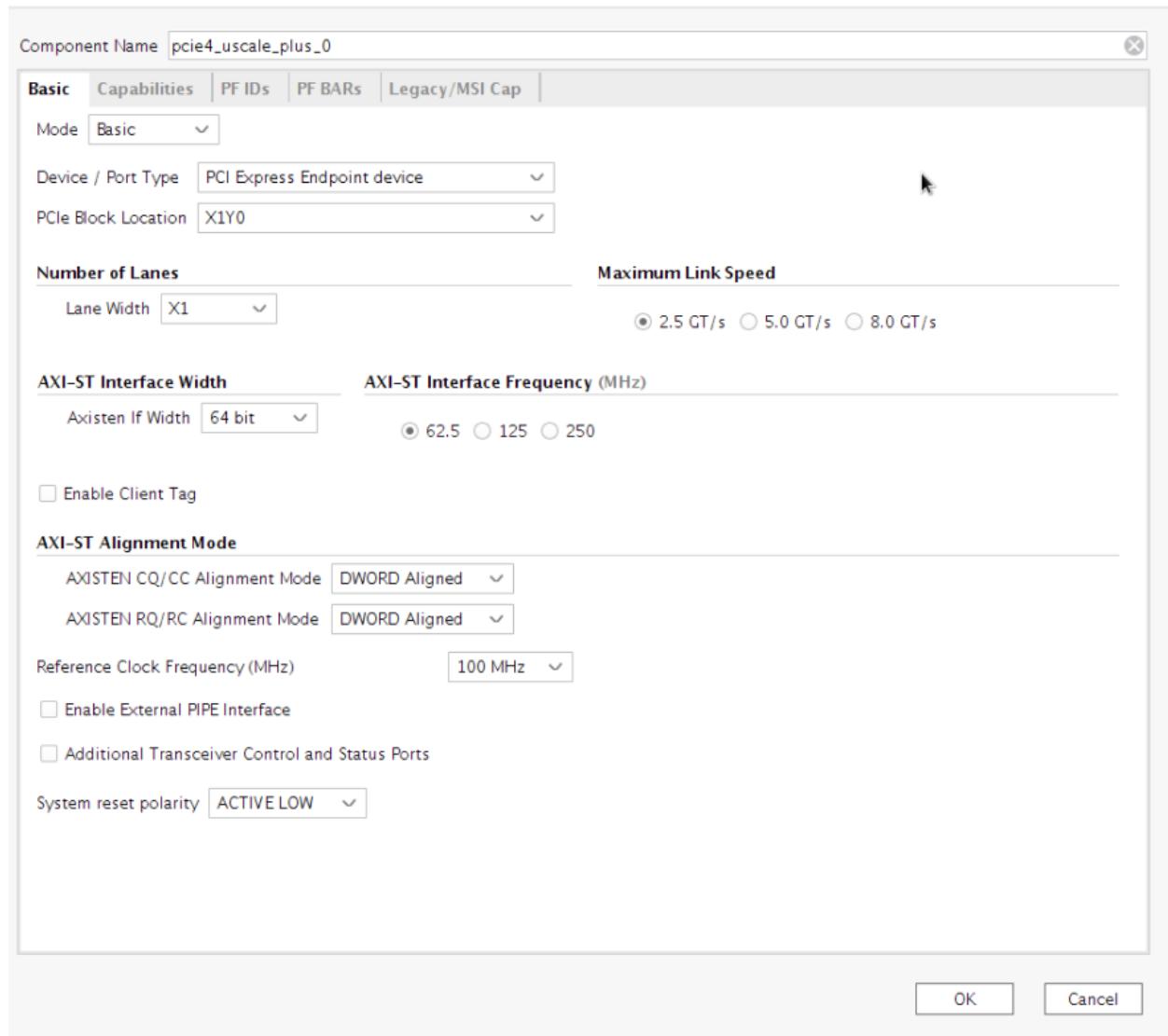
Note: If an unsupported silicon device is selected, the core is grayed out (unavailable) in the list of cores.

6. Select `xcku11p-ffva1156-3-e` from the list.



7. In the final project summary page, click **OK**.
8. In the Vivado IP catalog, expand **Standard Bus Interfaces → PCI Express**, and double-click **UltraScale+ Devices Integrated Block for PCIe** to display the Customize IP dialog box.
9. In the Component Name field, enter a name for the core.

Note: <component_name> is used in this example.



10. From the Device/Port Type drop-down menu, select the appropriate device/port type of the core (**Endpoint** or **Root Port**).
11. Click **OK** to generate the core using the default parameters.

Simulating the Example Design

The example design provides a quick way to simulate and observe the behavior of the core for PCI Express® Endpoint and Root port Example design projects generated using the Vivado Design Suite.

The currently supported simulators are:

- Vivado simulator (default)
- Questa Advanced Simulator
- Cadence Incisive Enterprise Simulator (IES)
- Synopsys Verilog Compiler Simulator (VCS)

You can generate an example design project and run simulation on the example project. The simulator uses the example design test bench and test cases provided along with the example design for both the design configurations.

A simulation, using the default Vivado simulator, is run as follows:

1. In the Sources Window, right-click the example project file (.xci), and select **Open IP Example Design**.
The example project is created.
2. In the Flow Navigator (left-hand pane), under Simulation, right-click **Run Simulation** and select **Run Behavioral Simulation**.



IMPORTANT! *The post-synthesis and post-implementation simulation options are not supported for the PCI Express block.*

After the Run Behavioral Simulation Option is running, you can observe the compilation and elaboration phase through the activity in the Tcl Console, and in the Simulation tab of the Log Window.

3. In Tcl Console, type the `run all` command and press **Enter**. This runs the complete simulation as per the test case provided in example design test bench.

After the simulation is complete, the result can be viewed in the Tcl Console.

Endpoint Configuration

The simulation environment provided with the UltraScale+ Devices Integrated Block for PCI Express core in Endpoint configuration performs simple memory access tests on the PIO example design. Transactions are generated by the Root Port Model and responded to by the PIO example design.

- PCI Express Transaction Layer Packets (TLPs) are generated by the test bench transmit user application (`pci_exp_usrapp_tx`). As it transmits TLPs, it also generates a log file, `tx.dat`.
- PCI Express TLPs are received by the test bench receive user application (`pci_exp_usrapp_rx`). As the user application receives the TLPs, it generates a log file, `rx.dat`.

For more information about the test bench, see [Root Port Model Test Bench for Endpoint](#) in the next chapter.

Synthesizing and Implementing the Example Design

To run synthesis and implementation on the example design in the Vivado Design Suite environment:

1. Go to the XCI file, right-click, and select **Open IP Example Design**.

A new Vivado tool window opens with the project name “example_project” within the project directory.

2. In the Flow Navigator, click **Run Synthesis** and **Run Implementation**.



TIP: Click **Run Implementation** first to run both synthesis and implementation. Click **Generate Bitstream** to run synthesis, implementation, and then bitstream.

Test Bench

Root Port Model Test Bench for Endpoint

The PCI Express Root Port Model is a robust test bench environment that provides a test program interface that can be used with the provided Programmed Input/Output (PIO) design or with your design. The purpose of the Root Port Model is to provide a source mechanism for generating downstream PCI Express TLP traffic to stimulate the customer design, and a destination mechanism for receiving upstream PCI Express TLP traffic from the customer design in a simulation environment.

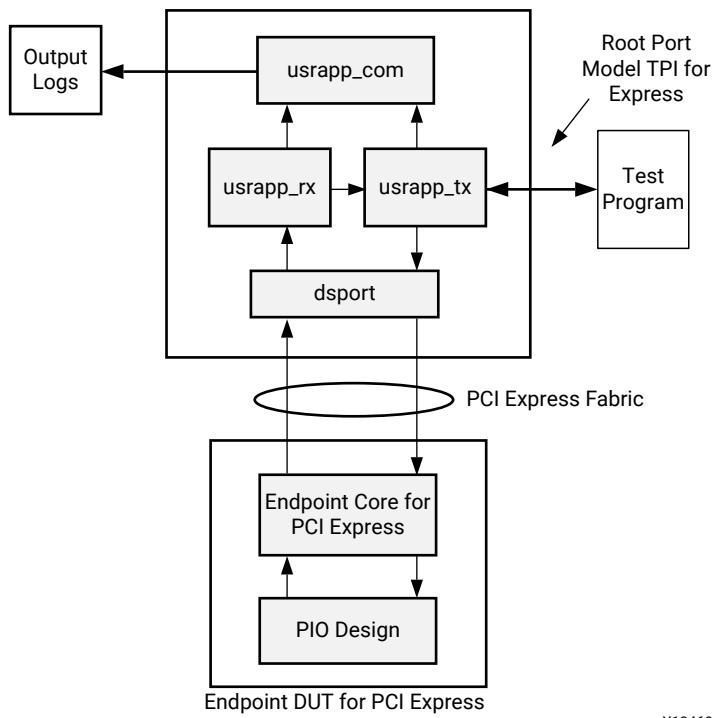
Source code for the Root Port Model is included to provide the model for a starting point for your test bench. All the significant work for initializing the core configuration space, creating TLP transactions, generating TLP logs, and providing an interface for creating and verifying tests is complete. This allows you to focus on verifying the functionality of the design rather than spending time developing an Endpoint core test bench infrastructure.

The Root Port Model consists of:

- Test Programming Interface (TPI), which allows you to stimulate the Endpoint device for the PCI Express.
- Example tests that illustrate how to use the test program TPI.
- Verilog source code for all Root Port Model components, which allow you to customize the test bench.

The following figure illustrates the Root Port Model coupled with the PIO design.

Figure 136: Root Port Model and Top-Level Endpoint



X12468

Architecture

The Root Port Model consists of these blocks:

- `dsport` (Root Port)
- `usrapp_tx`
- `usrapp_rx`
- `usrapp_com` (Verilog only)

The `usrapp_tx` and `usrapp_rx` blocks interface with the `dsport` block for transmission and reception of TLPs to/from the Endpoint Design Under Test (DUT). The Endpoint DUT consists of the Endpoint for PCIe and the PIO design (displayed) or customer design.

The `usrapp_tx` block sends TLPs to the `dsport` block for transmission across the PCI Express Link to the Endpoint DUT. In turn, the Endpoint DUT device transmits TLPs across the PCI Express Link to the `dsport` block, which are subsequently passed to the `usrapp_rx` block. The `dsport` and core are responsible for the data link layer and physical link layer processing when communicating across the PCI Express logic. Both `usrapp_tx` and `usrapp_rx` use the `usrapp_com` block for shared functions, for example, TLP processing and log file outputting.

Transaction sequences or test programs are initiated by the `usrapp_tx` block to stimulate the Endpoint device fabric interface. TLP responses from the Endpoint device are received by the `usrapp_rx` block. Communication between the `usrapp_tx` and `usrapp_rx` blocks allow the `usrapp_tx` block to verify correct behavior and act accordingly when the `usrapp_rx` block has received TLPs from the Endpoint device.

Scaled Simulation Timeouts

The simulation model of the core uses scaled-down times during link training to allow for the link to train in a reasonable amount of time during simulation. According to the *PCI Express Specification, rev. 3.0* (<http://www.pcisig.com/specifications>) , there are various timeouts associated with the link training and status state machine (LTSSM) states. The core scales these timeouts by a factor of 256 during simulation, except in the Recovery Speed_1 LTSSM state, where the timeouts are not scaled.

Test Selection

Available Tests

The following table describes the tests provided with the Root Port Model.

Table 77: Root Port Model Provided Tests

Test Name	Language	Description
sample_smoke_test0	Verilog	Issues a PCI Type 0 Configuration Read TLP and waits for the completion TLP; then compares the value returned with the expected Device/Vendor ID value.
sample_smoke_test1	Verilog	Performs the same operation as sample_smoke_test0 but makes use of expectation tasks. This test uses two separate test program threads: one thread issues the PCI Type 0 Configuration Read TLP and the second thread issues the Completion with Data TLP expectation task. This test illustrates the form for a parallel test that uses expectation tasks. This test form allows for confirming reception of any TLPs from your design. Additionally, this method can be used to confirm reception of TLPs when ordering is unimportant.

Verilog Test Selection

The Verilog test model used for the Root Port Model lets you specify the name of the test to be run as a command line parameter to the simulator.

To change the test to be run, change the `TESTNAME` value, which is defined in the test files `sample_tests1.v` and `pio_tests.v`. This mechanism is used for Mentor Graphics Advanced Simulator. The Vivado simulator uses the `-testplusarg` option to specify `TESTNAME`. For example:

```
demo_tb.exe-gui -view wave.wcfg -wdb wave_isim -tclbatch isim_cmd.tcl  
-testplusarg TESTNAME=sample_smoke_test0
```

Waveform Dumping

For information on simulator waveform dumping, see the *Vivado Design Suite User Guide: Logic Simulation (UG900)*.

Verilog Flow

The Root Port Model provides a mechanism for outputting the simulation waveform to a file using the `+dump_all` command line parameter.

Output Logging

When a test fails on the example or customer design, the test programmer debugs the offending test case. Typically, the test programmer inspects the wave file for the simulation and cross-reference this to the messages displayed on the standard output. Because this approach can be very time consuming, the Root Port Model offers an output logging mechanism to assist the tester with debugging failing test cases to speed the process.

The Root Port Model creates three output log files during each simulation run. They are `tx.dat`, `rx.dat`, and `error.dat`. The `rx.dat` and `tx.dat` files each contain a detailed record of every TLP that was received and transmitted, respectively, by the Root Port Model.



TIP: With an understanding of the expected TLP transmission during a specific test case, you can isolate the failure.

The `error.dat` file is used in conjunction with the expectation tasks. Test programs that use the expectation tasks generate a general error message to standard output. Detailed information about the specific comparison failures that have occurred due to the expectation error is found in `error.dat`.

Parallel Test Programs

There are two classes of tests supported by the Root Port Model.

- **Sequential tests:** Tests that exist within one process and behave similarly to sequential programs. The test depicted in Test Program: `pio_writeReadBack_test0` (later in this chapter) is an example of a sequential test. Sequential tests are very useful when verifying behavior that have events with a known order.

- **Parallel tests:** Tests involving more than one process thread. The test `sample_smoke_test1` is an example of a parallel test with two process threads. Parallel tests are very useful when verifying that a specific set of events have occurred, however the order of these events are not known.

A typical parallel test uses the form of one command thread and one or more expectation threads. These threads work together to verify the device functionality. The role of the command thread is to create the necessary TLP transactions that cause the device to receive and generate TLPs. The role of the expectation threads is to verify the reception of an expected TLP. The Root Port Model TPI has a complete set of expectation tasks to be used in conjunction with parallel tests.

Because the example design is a target-only device, only Completion TLPs can be expected by parallel test programs while using the PIO design. However, the full library of expectation tasks can be used for expecting any TLP type when used in conjunction with the customer design (which can include bus-mastering functionality).

Completer Model

The Completer Model is enabled through the Vivado Tcl console by executing the following command after a core has been configured:

```
set_property -dict [list CONFIG.completer_model {true} [get_ips <PCIE IP Core Name>]]
```

When the core is configured with the 512-bit AXI Interface, you can opt in for this Completer Model test bench which can be used in conjunction with your design to exercise bus-mastering functionality (upstream direction traffic from the Endpoint DUT to the Root Port Model).

The Completer Model provides a Root Port side memory array (`DATA_STORE_2`) that can be written through a Memory Write transaction and be read through a Memory Read transaction from the Endpoint DUT. This memory can be configured through two different parameters available at the top level of the Root Port Model module (`xilinx_pcie_uscale_rp.v`).

- **RP_BAR[63:0]:** Provides the address of the first byte of the `DATA_STORE_2` array.
- **RP_BAR_SIZE[5:0] :** Provides the number of byte address bits -1 of the `DATA_STORE_2` array. For example, a value of 11 provides $2^{(11+1)}$ bytes or 4 KB of available memory.

Each memory transaction is checked against the memory array location based on the two aforementioned parameters, byte enables, 4K boundaries, Max Payload Size, and Max Read Request Size rules set at the Root Port model. Each Memory Read Completion returned is split according to Max Payload Size and Read Completion Boundary rules. The Completer Model also supports a Zero Length Write packet which intercepts the packet but does not store its payload data, and a Zero Length Read packet which returns a one DW payload data.

Test Description

The Root Port Model provides a Test Program Interface (TPI). The TPI provides the means to create tests by invoking a series of Verilog tasks. All Root Port Model tests should follow the same six steps:

1. Perform conditional comparison of a unique test name.
2. Set up master timeout in case simulation hangs.
3. Wait for reset and link-up.
4. Initialize the configuration space of the Endpoint.
5. Transmit and receive TLPs between the Root Port Model and the Endpoint DUT.
6. Verify that the test succeeded.

Test Program: pio_writeReadBack_test0

```

1.      else if(testname == "pio_writeReadBack_test1"
2.      begin
3.          // This test performs a 32 bit write to a 32 bit Memory space and
4.          // performs a read back
5.          TSK_SIMULATION_TIMEOUT(10050);
6.          TSK_SYSTEM_INITIALIZATION;
7.          TSK_BAR_INIT;
8.          for (ii = 0; ii <= 6; ii = ii + 1) begin
9.              if (BAR_INIT_P_BAR_ENABLED[ii] > 2'b00) // bar is enabled
10.                 case(BAR_INIT_P_BAR_ENABLED[ii])
11.                     2'b01 : // IO SPACE
12.                     begin
13.                         $display("[%t] : NOTHING: to IO 32 Space BAR %x", $realtime, ii);
14.                     end
15.                     2'b10 : // MEM 32 SPACE
16.                     begin
17.                         $display("[%t] : Transmitting TLPs to Memory 32 Space BAR %x",
18.                         $realtime, ii);
19.                         //-----
-----// Event : Memory Write 32 bit TLP
20.                         //-----
-----// DATA_STORE[0] = 8'h04;
21.                         DATA_STORE[1] = 8'h03;
22.                         DATA_STORE[2] = 8'h02;
23.                         DATA_STORE[3] = 8'h01;
24.                         P_READ_DATA = 32'hffff_ffff; // make sure P_READ_DATA has known
initial value
25.                         TSK_TX_MEMORY_WRITE_32(DEFAULT_TAG, DEFAULT_TC, 10'd1,
BAR_INIT_P_BAR[ii][31:0], 4'hF, 4'hF, 1'b0);
26.                         TSK_TX_CLK_EAT(10);
27.                         DEFAULT_TAG = DEFAULT_TAG + 1;
28.                         //-----
-----// Event : Memory Read 32 bit TLP
29.                         //-----
-----// TSK_TX_MEMORY_READ_32(DEFAULT_TAG, DEFAULT_TC, 10'd1,

```

```

BAR_INIT_P_BAR[ii][31:0], 4'hF, 4'hF);
33.     TSK_WAIT_FOR_READ_DATA;
34.     if (P_READ_DATA != {DATA_STORE[3], DATA_STORE[2], DATA_STORE[1],
DATA_STORE[0] })
35.     begin
36.         $display("[%t] : Test FAILED --- Data Error Mismatch, Write Data %x !
= Read Data %x",
$realtime,{DATA_STORE[3], DATA_STORE[2], DATA_STORE[1], DATA_STORE[0]}, P_READ_DATA);
37.     end
38.     else
39.     begin
40.         $display("[%t] : Test PASSED --- Write Data: %x successfully
received", $realtime, P_READ_DATA);
41.     end

```

Expanding the Root Port Model

The Root Port Model was created to work with the PIO design, and for this reason is tailored to make specific checks and warnings based on the limitations of the PIO design. These checks and warnings are enabled by default when the Root Port Model is generated by the Vivado IP catalog. However, these limitations can be disabled so that they do not affect the customer design.

Because the PIO design was created to support at most one I/O BAR, one Mem64 BAR, and two Mem32 BARs (one of which must be the EROM space), the Root Port Model by default makes a check during device configuration that verifies that the core has been configured to meet this requirement. A violation of this check causes a warning message to be displayed as well as for the offending BAR to be gracefully disabled in the test bench. This check can be disabled by setting the `pio_check_design` variable to zero in the `pci_exp_usrapp_tx.v` file.

Root Port Model TPI Task List

The Root Port Model TPI tasks include the following tasks.

Test Setup Tasks

Table 78: Test Setup Tasks

Name	Input(s)		Description
TSK_SYSTEM_INITIALIZATION	None		Waits for transaction interface reset and link-up between the Root Port Model and the Endpoint DUT. This task must be invoked prior to the Endpoint core initialization.
TSK_USR_DATA_SETUP_SEQ	None		Initializes global 4096 byte DATA_STORE array and resizable DATA_STORE_2 array entries to sequential values from zero to 4095.
TSK_TX_CLK_EAT	clock count	31:30	Waits clock_count transaction interface clocks.

Table 78: Test Setup Tasks (cont'd)

Name	Input(s)		Description
TSK_SIMULATION_TIMEOUT	timeout	31:0	Sets master simulation timeout value in units of transaction interface clocks. This task should be used to ensure that all DUT tests complete.

TLP Tasks

Table 79: TLP Tasks

Name	Input(s)		Description
TSK_TX_TYPE0_CONFIGURATION_READ	tag_ reg_addr_ first_dw_be_	7:0 11:0 3:0	Sends a Type 0 PCI Express Config Read TLP from Root Port Model to reg_addr of Endpoint DUT with tag_ and first_dw_be_inputs. Cpld returned from the Endpoint DUT uses the contents of global EP_BUS_DEV_FNS as the completer ID.
TSK_TX_TYPE1_CONFIGURATION_READ	tag_ reg_addr_ first_dw_be_	7:0 11:0 3:0	Sends a Type 1 PCI Express Config Read TLP from Root Port Model to reg_addr of Endpoint DUT with tag_ and first_dw_be_inputs. Cpld returned from the Endpoint DUT uses the contents of global EP_BUS_DEV_FNS as the completer ID.
TSK_TX_TYPE0_CONFIGURATION_WRITE	tag_ reg_addr_ reg_data_ first_dw_be_	7:0 11:0 31:0 3:0	Sends a Type 0 PCI Express Config Write TLP from Root Port Model to reg_addr of Endpoint DUT with tag_ and first_dw_be_inputs. Cpl returned from the Endpoint DUT uses the contents of global EP_BUS_DEV_FNS as the completer ID.
TSK_TX_TYPE1_CONFIGURATION_WRITE	tag_ reg_addr_ reg_data_ first_dw_be_	7:0 11:0 31:0 3:0	Sends a Type 1 PCI Express Config Write TLP from Root Port Model to reg_addr of Endpoint DUT with tag_ and first_dw_be_inputs. Cpl returned from the Endpoint DUT uses the contents of global EP_BUS_DEV_FNS as the completer ID.
TSK_TX_MEMORY_READ_32	tag_ tc_ len_ addr_ last_dw_be_ first_dw_be_	7:0 2:0 10:0 31:0 3:0 3:0	Sends a PCI Express Memory Read TLP from Root Port to 32-bit memory address addr of Endpoint DUT. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.
TSK_TX_MEMORY_READ_64	tag_ tc_ len_ addr_ last_dw_be_ first_dw_be_	7:0 2:0 10:0 63:0 3:0 3:0	Sends a PCI Express Memory Read TLP from Root Port Model to 64-bit memory address addr of Endpoint DUT. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.

Table 79: TLP Tasks (cont'd)

Name	Input(s)	Description	
TSK_TX_MEMORY_WRITE_32	tag_ tc_ len_ addr_ last_dw_be_ first_dw_be_ ep_	7:0 2:0 10:0 31:0 3:0 3:0 -	Sends a PCI Express Memory Write TLP from Root Port Model to 32-bit memory address addr_ of Endpoint DUT. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID. The global DATA_STORE byte array is used to pass write data to task.
TSK_TX_MEMORY_WRITE_64	tag_ tc_ len_ addr_ last_dw_be_ first_dw_be_ ep_	7:0 2:0 10:0 63:0 3:0 3:0 -	Sends a PCI Express Memory Write TLP from Root Port Model to 64-bit memory address addr_ of Endpoint DUT. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID. The global DATA_STORE byte array is used to pass write data to task.
TSK_TX_COMPLETION	req_id_ tag_ tc_ len_ byte_count_ lower_addr_ comp_status_ ep_	15:0 7:0 2:0 10:0 2:0 11:0 6:0 -	Sends a PCI Express Completion TLP from Root Port Model to the Endpoint DUT using global RP_BUS_DEV_FNS as the completer ID, req_id_ input as the requester ID. comp_status_ input can be set to one of the following: 3'b000 = Successful Completion 3'b001 = Unsupported Request 3'b010 = Configuration Request Retry Status 3'b100 = Completer Abort
TSK_TX_COMPLETION_DATA	req_id_ tag_ tc_ len_ byte_count_ lower_addr_ ram_ptr comp_status_ ep_	15:0 7:0 2:0 10:0 11:0 6:0 RP_BAR_SIZE:0 2:0 -	Sends a PCI Express Completion with Data TLP from Root Port Model to the Endpoint DUT using global RP_BUS_DEV_FNS as the completer ID, req_id_ input as the requester ID. The global DATA_STORE_2 byte array is used to pass completion data to task and the ram_ptr input is used to offset the starting byte within this array.
TSK_TX_MESSAGE	tag_ tc_ len_ data_ message_rtg_ message_code_	7:0 2:0 10:0 63:0 2:0 7:0	Sends a PCI Express Message TLP from Root Port Model to Endpoint DUT. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.
TSK_TX_MESSAGE_DATA	tag_ tc_ len_ data_ message_rtg_ message_code_	7:0 2:0 10:0 63:0 2:0 7:0	Sends a PCI Express Message with Data TLP from Root Port Model to Endpoint DUT. The global DATA_STORE byte array is used to pass message data to task. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.

Table 79: TLP Tasks (cont'd)

Name	Input(s)			Description
TSK_TX_IO_READ	tag_ addr_ first_dw_be_	7:0 31:0 3:0		Sends a PCI Express I/O Read TLP from Root Port Model to I/O address addr_[31:2] of the Endpoint DUT. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.
TSK_TX_IO_WRITE	tag_ addr_ first_dw_be_ data	7:0 31:0 3:0 31:0		Sends a PCI Express I/O Write TLP from Root Port Model to I/O address addr_[31:2] of the Endpoint DUT. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.
TSK_TX_BAR_READ	bar_index byte_offset tag_ tc_	2:0 31:0 7:0 2:0		Sends a PCI Express one Dword Memory 32, Memory 64, or I/O Read TLP from the Root Port Model to the target address corresponding to offset byte_offset from BAR bar_index of the Endpoint DUT. This task sends the appropriate Read TLP based on how BAR bar_index has been configured during initialization. This task can only be called after TSK_BAR_INIT has successfully completed. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.
TSK_TX_BAR_WRITE	bar_index byte_offset tag_ tc_ data_	2:0 31:0 7:0 2:0 31:0		Sends a PCI Express one Dword Memory 32, Memory 64, or I/O Write TLP from the Root Port to the target address corresponding to offset byte_offset from BAR bar_index of the Endpoint DUT. This task sends the appropriate Write TLP based on how BAR bar_index has been configured during initialization. This task can only be called after TSK_BAR_INIT has successfully completed.
TSK_WAIT_FOR_READ_DATA	None			Waits for the next completion with data TLP that was sent by the Endpoint DUT. On successful completion, the first Dword of data from the CplD is stored in the global P_READ_DATA. This task should be called immediately following any of the read tasks in the TPI that request Completion with Data TLPs to avoid any race conditions. By default this task locally times out and terminate the simulation after 1000 transaction interface clocks. The global cpld_to_finish can be set to zero so that local timeout returns execution to the calling test and does not result in simulation timeout. For this case test programs should check the global cpld_to, which when set to one indicates that this task has timed out and that the contents of P_READ_DATA are invalid.

Table 79: TLP Tasks (cont'd)

Name	Input(s)		Description
TSK_TX_SYNCHRONIZE	first_ active_ last_call_ tready_sw_	- - - -	Waits for assertion of AXI4-Stream Requester Request or Completer Completion Interface Ready signal and synchronizes the output in the log file to each transaction currently active. first_input indicates start of packet. active_input indicates a transaction is currently in progress last_call_input indicates end of packet tready_sw input selects Requester Request or Completer Completion Interface Ready signal
TSK_BUILD_RC_TO_PCIE_PKT	rc_data_QW0 rc_data_QW1 m_axis_rc_tkeep m_axis_rc_tlast	63:0 63:0 KEEP_WIDTH-1:0 -	Converts AXI4-Stream packet at Requester Completion Interface from a Descriptor packet format to PCIe TLP packet format for logging purposes.
TSK_BUILD_CQ_TO_PCIE_PKT	cq_data cq_be m_axis_cq_tdata	63:0 7:0 63:0	Converts AXI4-Stream packet at Completer Request Interface from a Descriptor packet format to PCIe TLP packet format for logging purposes.
TSK_BUILD_CPLD_PKT	cq_addr cq_be m_axis_cq_tdata	63:0 15:0 63:0	Returns Completion or Completion with Data for Memory Read received from the Endpoint DUT. When the Completer Model is used, the completion produced is split according to Max Payload Size and Read Completion Boundary rules set at the Root Port Model. Completion with Data uses data stored in the global DATA_STORE_2 array.

BAR Initialization Tasks

Table 80: BAR Initialization Tasks

Name	Input(s)	Description
TSK_BAR_INIT	None	Performs a standard sequence of Base Address Register initialization tasks to the Endpoint device using the PCI Express fabric. Performs a scan of the Endpoint PCI BAR range requirements, performs the necessary memory and I/O space mapping calculations, and finally programs the Endpoint so that it is ready to be accessed. On completion, the user test program can begin memory and I/O transactions to the device. This function displays to standard output a memory and I/O table that details how the Endpoint has been initialized. This task also initializes global variables within the Root Port Model that are available for test program usage. This task should only be called after TSK_SYSTEM_INITIALIZATION.

Table 80: BAR Initialization Tasks (cont'd)

Name	Input(s)	Description
TSK_BAR_SCAN	None	<p>Performs a sequence of PCI Type 0 Configuration Writes and Configuration Reads using the PCI Express logic to determine the memory and I/O requirements for the Endpoint.</p> <p>The task stores this information in the global array <code>BAR_INIT_P_BAR_RANGE[]</code>. This task should only be called after <code>TSK_SYSTEM_INITIALIZATION</code>.</p>
TSK_BUILD_PCIE_MAP	None	<p>Performs memory and I/O mapping algorithm and allocates Memory 32, Memory 64, and I/O space based on the Endpoint requirements.</p> <p>This task has been customized to work in conjunction with the limitations of the PIO design and should only be called after completion of <code>TSK_BAR_SCAN</code>.</p>
TSK_DISPLAY_PCIE_MAP	None	<p>Displays the memory mapping information of the Endpoint core PCI Base Address Registers. For each BAR, the BAR value, the BAR range, and BAR type is given. This task should only be called after completion of <code>TSK_BUILD_PCIE_MAP</code>.</p>

Example PIO Design Tasks

Table 81: Example PIO Design Tasks

Name	Input(s)		Description
TSK_TX_READBACK_CONFIG	None		<p>Performs a sequence of PCI Type 0 Configuration Reads to the Endpoint device Base Address Registers, PCI Command register, and PCIe Device Control register using the PCI Express logic.</p> <p>This task should only be called after <code>TSK_SYSTEM_INITIALIZATION</code>.</p>
TSK_MEM_TEST_DATA_BUS	bar_index	2:0	<p>Tests whether the PIO design FPGA block RAM data bus interface is correctly connected by performing a 32-bit walking ones data test to the I/O or memory address pointed to by the input <code>bar_index</code>.</p> <p>For an exhaustive test, this task should be called four times, once for each block RAM used in the PIO design.</p>

Table 81: Example PIO Design Tasks (cont'd)

Name	Input(s)		Description
TSK_MEM_TEST_ADDR_BUS	bar_index nBytes	2:0 31:0	<p>Tests whether the PIO design FPGA block RAM address bus interface is accurately connected by performing a walking ones address test starting at the I/O or memory address pointed to by the input bar_index.</p> <p>For an exhaustive test, this task should be called four times, once for each block RAM used in the PIO design. Additionally, the nBytes input should specify the entire size of the individual block RAM.</p>
TSK_MEM_TEST_DEVICE	bar_index nBytes	2:0 31:0	<p>Tests the integrity of each bit of the PIO design FPGA block RAM by performing an increment/decrement test on all bits starting at the block RAM pointed to by the input bar_index with the range specified by input nBytes.</p> <p>For an exhaustive test, this task should be called four times, once for each block RAM used in the PIO design. Additionally, the nBytes input should specify the entire size of the individual block RAM.</p>
TSK_RESET	Reset	0	<p>Initiates sys_rst_n signal in board.v file. Forces the sys_rst_n signal to assert the reset. Use TSK_RESET (1'b1) to assert the reset and TSK_RESET (1'b0) to release the reset signal.</p>
TSK_MALFORMED	malformed_bits	7:0	<p>Control bits for creating malformed TLPs:</p> <ul style="list-style-type: none"> 0001: Generate Malformed TLP for I/O Requests and Configuration Requests called immediately after this task 0010: Generate Malformed Completion TLPs for Memory Read requests received at the Root Port

Expectation Tasks

Table 82: Expectation Tasks

Name	Input(s)		Output	Description
TSK_EXPECT_CPLD	traffic_class td ep attr length completer_id completer_status bcm byte_count requester_id tag address_low	2:0 - - 1:0 10:0 15:0 2:0 - 11:0 15:0 7:0 6:0	Expect status	Waits for a Completion with Data TLP that matches traffic_class, td, ep, attr, length, and payload. Returns a 1 on successful completion; 0 otherwise.
TSK_EXPECT_CPL	traffic_class td ep attr completer_id completer_status bcm byte_count requester_id tag address_low	2:0 - - 1:0 15:0 2:0 - 11:0 15:0 7:0 6:0	Expect status	Waits for a Completion without Data TLP that matches traffic_class, td, ep, attr, and length. Returns a 1 on successful completion; 0 otherwise.
TSK_EXPECT_MEMRD	traffic_class td ep attr length requester_id tag last_dw_be first_dw_be address	2:0 - - 1:0 10:0 15:0 7:0 3:0 3:0 29:0	Expect status	Waits for a 32-bit Address Memory Read TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. This task can only be used in conjunction with Bus Master designs.
TSK_EXPECT_MEMRD64	traffic_class td ep attr length requester_id tag last_dw_be first_dw_be address	2:0 - - 1:0 10:0 15:0 7:0 3:0 3:0 61:0	Expect status	Waits for a 64-bit Address Memory Read TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. This task can only be used in conjunction with Bus Master designs.

Table 82: **Expectation Tasks (cont'd)**

Name	Input(s)		Output	Description
TSK_EXPECT_MEMWR	traffic_class td ep attr length requester_id tag last_dw_be first_dw_be address	2:0 - - 1:0 10:0 15:0 7:0 3:0 3:0 29:0	Expect status	Waits for a 32-bit Address Memory Write TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. This task can only be used in conjunction with Bus Master designs.
TSK_EXPECT_MEMWR64	traffic_class td ep attr length requester_id tag last_dw_be first_dw_be address	2:0 - - 1:0 10:0 15:0 7:0 3:0 3:0 61:0	Expect status	Waits for a 64-bit Address Memory Write TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. This task can only be used in conjunction with Bus Master designs.
TSK_EXPECT_IOWR	td ep requester_id tag first_dw_be address data	- - 15:0 7:0 3:0 31:0 31:0	Expect status	Waits for an I/O Write TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. This task can only be used in conjunction with Bus Master designs.

Endpoint Model Test Bench for Root Port

The Endpoint model test bench for the core in Root Port configuration is a simple example test bench that connects the Configurator example design and the PCI Express Endpoint model allowing the two to operate like two devices in a physical system. Because the Configurator example design consists of logic that initializes itself and generates and consumes bus traffic, the example test bench only implements logic to monitor the operation of the system and terminate the simulation.

The Endpoint model test bench consists of:

- Verilog or VHDL source code for all Endpoint model components.
- PIO slave design.

The figure earlier in this chapter illustrates the Endpoint model coupled with the Configurator example design.

Architecture

The Endpoint model consists of these blocks:

- PCI Express Endpoint (the core in Endpoint configuration) model.
- PIO slave design, consisting of:
 - PIO_RX_ENGINE
 - PIO_TX_ENGINE
 - PIO_EP_MEM
 - PIO_TO_CTRL

The PIO_RX_ENGINE and PIO_TX_ENGINE blocks interface with the Endpoint block for reception and transmission of TLPs from/to the Root Port Design Under Test (DUT). The Root Port DUT consists of the core configured as a Root Port and the Configurator Example Design, which consists of a Configurator block and a PIO Master design, or customer design.

The PIO slave design is described in detail in [Programmed Input/Output: Endpoint Example Design](#).

Simulating the Design

The `simulate_mti.do` simulation script file is provided with the model to facilitate simulation with the Mentor Graphics Advanced simulator.

The example simulation script files are located in this directory:

```
<project_dir>/<component_name>/simulation/functional
```

Instructions for simulating the Configurator example design with the Endpoint model are provided in "Simulation" in the Design Flow Steps chapter.

Note: For Cadence IES users, the work construct must be manually inserted into the `cds.lib` file:

```
DEFINE WORK WORK
```

Scaled Simulation Timeouts

The simulation model of the core uses scaled-down times during link training to allow for the link to train in a reasonable amount of time during simulation. According to the *PCI Express Specification, rev. 3.0* (<http://www.pcisig.com/specifications>) , there are various timeouts associated with the link training and status state machine (LTSSM) states. The core scales these timeouts by a factor of 256 during simulation, except in the Recovery Speed_1 LTSSM state, where the timeouts are not scaled.

Waveform Dumping

For information on simulator waveform dumping, see the *Vivado Design Suite User Guide: Logic Simulation* ([UG900](#)).

Output Logging

The test bench outputs messages, captured in the simulation log, indicate the time at which these occur:

- `user_reset` deasserted
- `user_lnk_up` asserted
- `cfg_done` asserted by the Configurator
- `pio_test_finished` asserted by the PIO Master
- Simulation Timeout (if `pio_test_finished` or `pio_test_failed` never asserted)

Upgrading

This appendix contains information about upgrading to a more recent version of the IP core.

Migrating from UltraScale to UltraScale+ Devices

This section provides information for users migrating from the UltraScale+™ devices Integrated Block for PCIe core to the UltraScale™ devices Integrated Block for PCIe core.

New Ports

The following table lists the new ports in the UltraScale+ device core to the UltraScale device core.

Table 83: New Ports in the UltraScale+ Devices Core

Names	I/O	Notes
pcie_rq_seq_num0[5:0]	O	pcie_rq_seq_num in UltraScale
pcie_rq_seq_num_vld0	O	pcie_rq_seq_num_vld in UltraScale
pcie_rq_tag0[7:0]	O	pcie_rq_tag in UltraScale
pcie_rq_tag_vld0	O	pcie_rq_tag_vld in UltraScale
pcie_rq_seq_num1[5:0]	O	
pcie_rq_seq_num_vld1	O	
pcie_rq_tag1[7:0]	O	
pcie_rq_tag_vld1	O	
cfg_mgmt_function_number[7:0]	I	
cfg_mgmt_debug_access	I	
cfg_local_error_valid	I	
cfg_local_error_out[4:0]	I	
cfg_rx_pm_state[1:0]	O	
cfg_tx_pm_state[1:0]	O	
cfg_bus_number[7:0]	O	
cfg_dev_id_pf0[15:0]	I	IDs are user accessible through I/Os
cfg_dev_id_pf1[15:0]	I	

Table 83: New Ports in the UltraScale+ Devices Core (cont'd)

Names	I/O	Notes
cfg_dev_id_pf2[15:0]	I	
cfg_dev_id_pf3[15:0]	I	
cfg_vend_id[15:0]	I	
cfg_rev_id_pf0[7:0]	I	
cfg_rev_id_pf1[7:0]	I	
cfg_rev_id_pf2[7:0]	I	
cfg_rev_id_pf3[7:0]	I	
cfg_subsys_id_pf0[15:0]	I	
cfg_subsys_id_pf1[15:0]	I	
cfg_subsys_id_pf2[15:0]	I	
cfg_subsys_id_pf3[15:0]	I	
cfg_vf_flr_func_num[7:0]	I	
cfg_interrupt_msi_pending_status_function_num[1:0]	I	
cfg_interrupt_msi_select[1:0]	I	
cfg_pm_aspm_l1_entry_reject	I	
cfg_pm_aspm_tx_l0s_entry_disable	I	
cfg_interrupt_msix_vec_pending[1:0]	I	
cfg_interrupt_msix_vec_pending_status	O	
pl_redo_eq	I	
pl_redo_eq_speed	I	
pl_eq_mismatch	O	
pl_redo_eq_pending	O	

Port Width Changes

The following table lists the ports for which widths were changed between the UltraScale devices core and the UltraScale+ devices core.

Table 84: Port Width Changes

Name	I/O
pcie_rq_tag_av[3:0]	O
pcie_tfc_nph_av[3:0]	O
pcie_tfc_npd_av[3:0]	O
pcie_cq_np_req[1:0]	I
pcie_cq_np_req_count[5:0]	O
cfg_mgmt_addr[9:0]	I
cfg_negotiated_width[2:0]	O
cfg_current_speed[1:0]	O

Table 84: Port Width Changes (cont'd)

Name	I/O
cfg_max_payload[1:0]	O
cfg_vf_status[503:0]	O
cfg_vf_power_state[755:0]	O
cfg_vf_tph_requester_enable[251:0]	O
cfg_vf_tph_st_mode[755:0]	O
cfg_vf_flr_in_process[251:0]	O
cfg_vf_flr_runc_num[7:0]	I
cfg_interrupt_msix_vf_enable[251:0]	O
cfg_interrupt_msix_vf_mask[251:0]	O
cfg_interrupt_msi_tph_st_tag[7:0]	I
cfg_interrupt_msi_function_number[7:0]	I

Deprecated Ports

The following table lists the ports that were deprecated in the UltraScale+ devices core relative to the UltraScale devices core.

Table 85: Ports Not Available in the UltraScale+ Devices Core

Name	I/O
cfg_mgmt_type1_cfg_reg_access	I
cfg_local_error	O
cfg_ltr_enable	O
cfg_dpa_substate_change[3:0]	O
cfg_per_func_status_control[2:0]	I
cfg_per_func_status_data[15:0]	O
cfg_per_function_number[3:0]	I
cfg_per_function_output_request	I
cfg_per_function_update_done	O
cfg_ds_function_number[2:0]	I
cfg_interrupt_msi_vf_enable[7:0]	O
cfg_interrupt_msix_sent	O
cfg_interrupt_msix_fail	O
user_tph_stt_address[4:0]	I
user_tph_function_num[3:0]	I
user_tph_stt_read_data[31:0]	O
user_tph_stt_read_data_valid	O
user_tph_stt_read_enable	I
pl_eq_reset_eieos_count	I

Dedicated PERST Routing

Dedicated reset routing is not used in the UltraScale+ core.

Upgrading in the Vivado Design Suite

This section provides information about any changes to the user logic or port designations that take place when you upgrade to a more current version of this IP core in the Vivado Design Suite.

Parameter Changes

The following table shows the changes to parameters in the current version of the core.

Table 86: New Parameter

User Parameter Name	Display Name	New/Change/Removed	Details	Default Value
axisten_if_enable_rx_msg_intfc	Enable RX Message INTFC	New	When checked, messages are routed to the cfg_msg_received signal at the Receive Message Interface. Otherwise, they are routed to the CQ Interface.	False (not checked)
enable_auto_rxeq	Enable Auto RxEq	New	Selects Receiver Equalization in Auto Mode	False (not checked)
mcap_fpga_bitstream_version	MCAP Bitstream Version register value	New	Specify the value of the MCAP Bitstream Version register within the MCAP register space.	00000000

Port Changes

The port in the following table appears when **Shared logic option GT common in core** is selected, and PLL type is not CPLL.

Table 87: New Port

Name	I/O	Width
ext_qpllxcalenb	O	1 Bit

The ports in the following table appear when **Shared logic option GT-Wizard in core** is selected.

Table 88: New Ports

Name	I/O	Width
ext_phy_clk_bufg_gt_ce	O	1 Bit

Table 88: New Ports (cont'd)

Name	I/O	Width
ext_phy_clk_bufg_gt_reset	O	1 Bit
ext_phy_clk_RST_IDLE	O	1 Bit
ext_phy_clk_txoutclk	O	1 Bit
ext_phy_clk_bufgtcemask	O	1 Bit
ext_phy_clk_gt_bufgrstmask	O	1 Bit
ext_phy_clk_bufgtdiv	O	8 Bits
ext_phy_clk_pclk2_gt	I	1 Bit
ext_phy_clk_int_clock	I	1 Bit
ext_phy_clk_pclk	I	1 Bit
ext_phy_clk_phy_pclk2	I	1 Bit
ext_phy_clk_phy_coreclk	I	1 Bit
ext_phy_clk_phy_userclk	I	1 Bit
ext_phy_clk_phy_mcapclk	I	1 Bit
ext_qpllxcalenb	O	1 Bit

GT Locations

This appendix provides a list of GTs locations available for this IP core and lists some key recommendations that should be considered when selecting the GT location. The following sections include tables which identify which GT Banks are available for selection based on the PCIe block location as selected during IP customization.

- Virtex® UltraScale+™ Devices Available GT Quads
- Kintex® UltraScale+™ Devices Available GT Quads
- Zynq® UltraScale+™ Devices Available GT Quads

A GT Quad is comprised of four GT lanes. When selecting GT Quads for the PCIe IP, Xilinx® recommends that you use the GT Quad most adjacent to the PCIe hard block. While this is not required, it will improve place, route, and timing for the design.

- Link widths of x1, x2, and x4 require one bonded GT Quad and should not split lanes between two GT Quads.
- A link width of x8 requires two adjacent GT Quads that are bonded and are in the same SLR.
- A link width of x16 requires four adjacent GT Quads that are bonded and are in the same SLR.

PCIe lane 0 is placed in the topmost GT of the top-most GT Quad by default (as shown in Vivado® Integrated Design Environment (IDE) Device view). Subsequent lanes use the next available GTs moving vertically down the device as the lane number increments. This means that by default the highest PCIe lane number uses the bottom-most GT in the bottom-most GT Quad that is used for PCIe. During IP customization, you can select the desired GT Quad for PCIe lane 0 from the drop-down selections.

The PCIe reference clock (`sys_clk_p/sys_clk_n`) uses `GTREFCLK0` in the PCIe lane 0 GT Quad for x1, x2, x4, and x8 configurations by default. For x16 configurations, the PCIe reference clock should use `GTREFCLK0` on a GT Quad associated with lanes 4-7 or lanes 8-11. This allows the clock to be forwarded to all 16 PCIe lanes. You can modify the reference clock default location by adding pin location constraints to the design.

The following diagrams show the ideal GT Quad and reference clock selections for various PCIe link configurations relative to the PCIe block location for a representative device.

Figure 137: Most Adjacent GT Quad Location For x1, x2, x4 PCIe Link Width

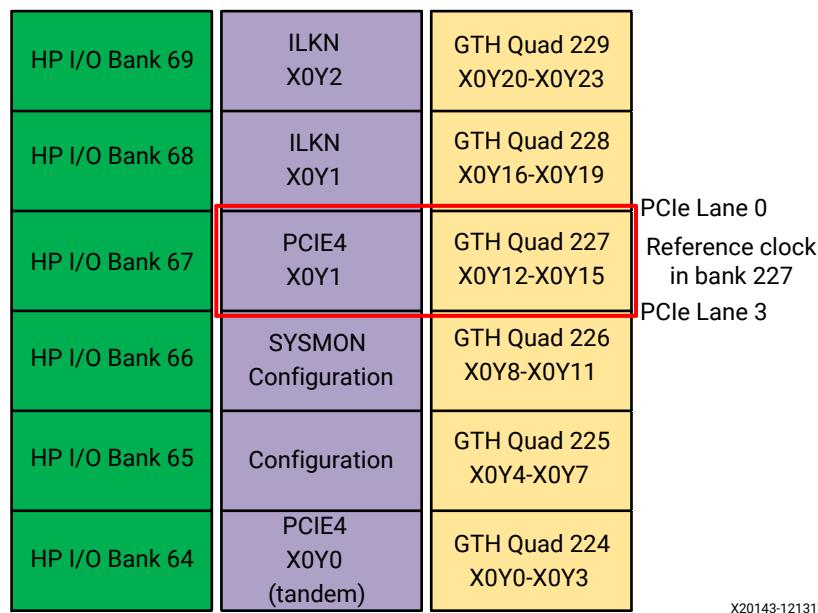


Figure 138: Most Adjacent GT Quads For x8 PCIe Link Width

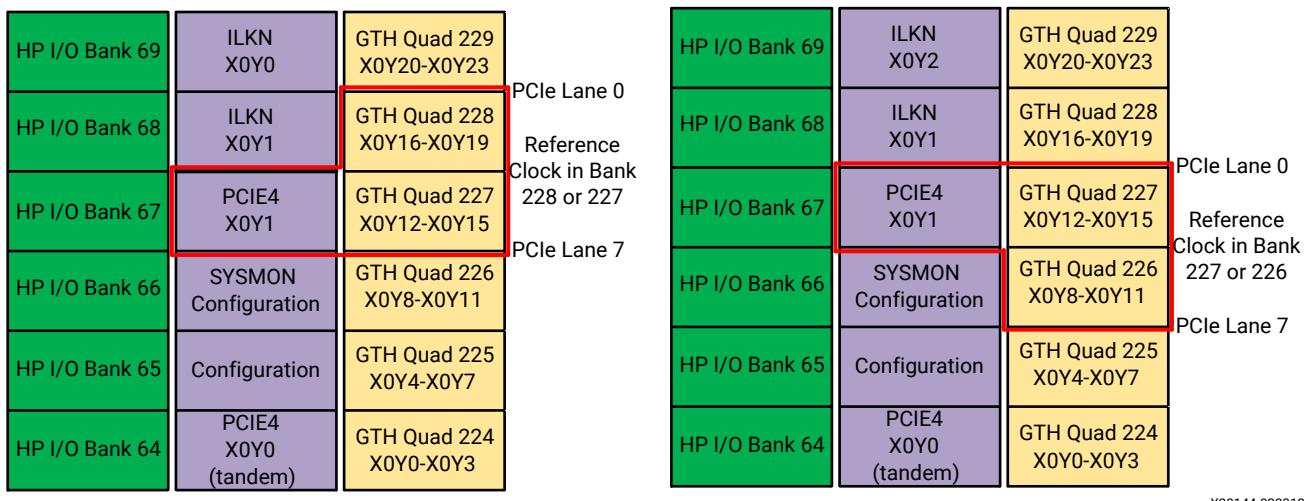
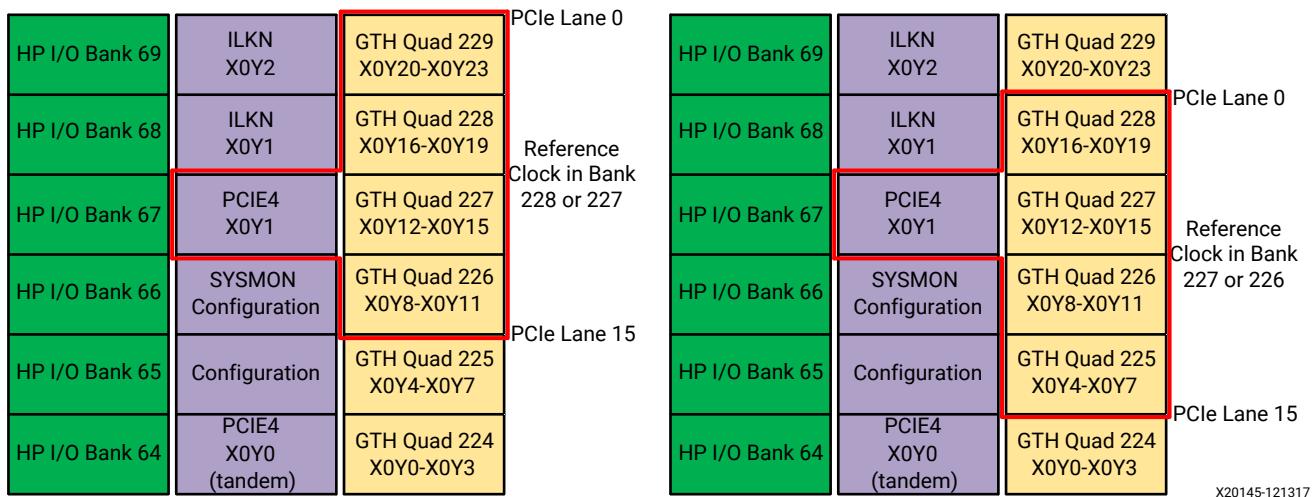
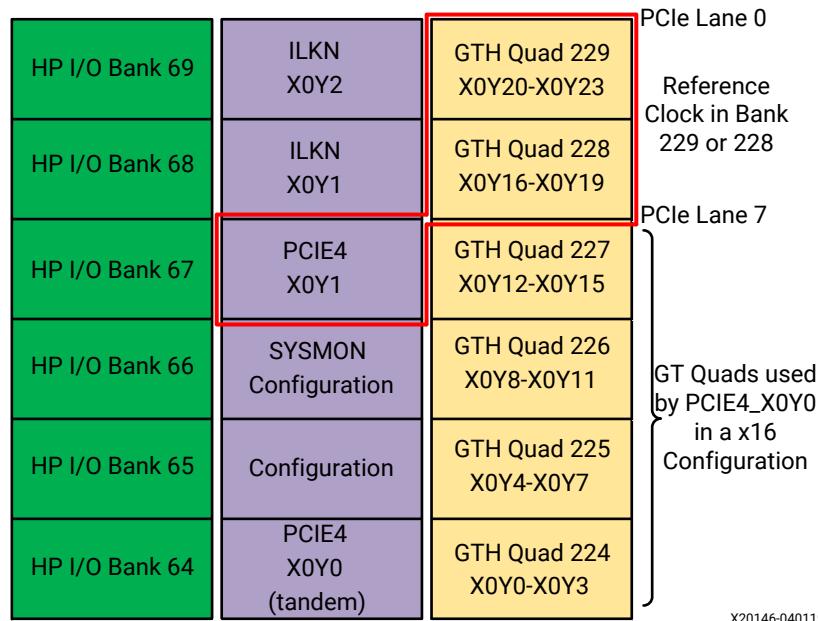


Figure 139: Most Adjacent GT Quads For x16 PCIe Link Width



Some PCIe locations have non-ideal GT Quad selections as result of their proximity to the edge of the device, SLR boundary, or other PCIe blocks. In these scenarios the most adjacent GTs may not be optimal for place and route, but will work as desired. The figure below shows one common example.

Figure 140: Alternative PCIe GT Location Selection



Virtex UltraScale+ Devices Available GT Quads

The following table shows the PCIe® lane 0 GT Quad options available for the different Virtex® UltraScale+™ devices. The GT Quad location is shown using the GT Quad bank number rather than GT XY coordinates. The *UltraScale and UltraScale+ FPGAs Packaging and Pinouts Product Specification (UG575)* provides a diagram describing the PCIe block locations relative to enabled GT Quads and includes both GT bank numbering and XY coordinates if needed.

Note: The table lists the number of GT Quads available for selection per device, package and the PCIe block available. During IP customization, in the Basic Tab, select **Advanced** mode, select **Enable GT Quad Selection**, and make your GT quad selection.

Note: The selections in bold are the default selections for each device and are typically optimized for boards which have an x16 PCIe connector. If no default is indicated, select the GT Quad nearest to the PCIe integrated block.

Table 89: Virtex UltraScale+ Devices Available GT Quads (XCVU27P)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU27P	FSGA2577	PCIE4_X0Y0	GTY_Quad_227	GTY_Quad_225, GTY_Quad_226	GTY_Quad_224
	FIGD2104	PCIE4_X0Y0	GTY_Quad_227	GTY_Quad_225, GTY_Quad_226	GTY_Quad_224

Table 90: Virtex UltraScale+ Devices Available GT Quads (XCVU3P)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU3P	FFVC1517	X0Y1	GTY_Quad_128	GTY_Quad_127, GTY_Quad_126	GTY_Quad_125
		X1Y0	GTY_Quad_228, GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4_X0Y1	GTY_Quad_128	GTY_Quad_127, GTY_Quad_126	GTY_Quad_125
		PCIE4_X1Y0	GTY_Quad_228, GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_124

Table 91: Virtex UltraScale+ Devices Available GT Quads (XCVU5P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU5P	FLVA2104	X0Y1	X16 Not Supported	GTY_Quad_127 , GTY_Quad_126	GTY_Quad_125
		X0Y3	X16 Not Supported	GTY_Quad_132 , GTY_Quad_131	GTY_Quad_130
		X1Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X1Y2	X16 Not Supported	GTY_Quad_233 , GTY_Quad_232	GTY_Quad_231
	FLVB2104	X0Y1	GTY_Quad_128	GTY_Quad_127, GTY_Quad_126	GTY_Quad_125
		X0Y3	GTY_Quad_133	GTY_Quad_132, GTY_Quad_131	GTY_Quad_130
		X1Y0	GTY_Quad_228 , GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X1Y2	GTY_Quad_233, GTY_Quad_232	GTY_Quad_231, GTY_Quad_230	GTY_Quad_229
XCVU5P	FLVC2104	X0Y1	GTY_Quad_128	GTY_Quad_127, GTY_Quad_126	GTY_Quad_125
		X0Y3	GTY_Quad_133	GTY_Quad_132, GTY_Quad_131	GTY_Quad_130
		X1Y0	GTY_Quad_228, GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X1Y2	GTY_Quad_233, GTY_Quad_232	GTY_Quad_231, GTY_Quad_230	GTY_Quad_229

Table 92: Virtex UltraScale+ Devices Available GT Quads (XCVU7P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU7P	FLVA2104	X0Y1	X16 Not Supported	GTY_Quad_127 , GTY_Quad_126	GTY_Quad_125
		X0Y3	X16 Not Supported	GTY_Quad_132 , GTY_Quad_131	GTY_Quad_130
		X1Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X1Y2	X16 Not Supported	GTY_Quad_233 , GTY_Quad_232	GTY_Quad_231
	FLVB2104	X0Y1	GTY_Quad_128	GTY_Quad_127, GTY_Quad_126	GTY_Quad_125
		X0Y3	GTY_Quad_133	GTY_Quad_132, GTY_Quad_131	GTY_Quad_130
		X1Y0	GTY_Quad_228, GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X1Y2	GTY_Quad_233, GTY_Quad_232	GTY_Quad_231, GTY_Quad_230	GTY_Quad_229
	FLVC2104	X0Y1	GTY_Quad_128	GTY_Quad_127, GTY_Quad_126	GTY_Quad_125
		X0Y3	GTY_Quad_133	GTY_Quad_132, GTY_Quad_131	GTY_Quad_130
		X1Y0	GTY_Quad_228 , GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X1Y2	GTY_Quad_233, GTY_Quad_232	GTY_Quad_231, GTY_Quad_230	GTY_Quad_229

Table 93: Virtex UltraScale+ Devices Available GT Quads (XCVU9P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU9P	FLGA2104	X0Y1	X16 Not Supported	GTY_Quad_122 , GTY_Quad_121	GTY_Quad_120
		X0Y3	X16 Not Supported	GTY_Quad_127 , GTY_Quad_126	GTY_Quad_125
		X1Y2	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X1Y4	X16 Not Supported	GTY_Quad_233 , GTY_Quad_232	GTY_Quad_231
	FLGB2104	X0Y1	GTY_Quad_123	GTY_Quad_122, GTY_Quad_121	GTY_Quad_120
		X0Y3	GTY_Quad_128	GTY_Quad_127, GTY_Quad_126	GTY_Quad_125
		X1Y2	GTY_Quad_228, GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X1Y4	GTY_Quad_233, GTY_Quad_232	GTY_Quad_231, GTY_Quad_230	GTY_Quad_229
	FLGC2104	X0Y1	X16 Not Supported	GTY_Quad_122 , GTY_Quad_121	GTY_Quad_120
		X0Y3	GTY_Quad_128	GTY_Quad_127, GTY_Quad_126	GTY_Quad_125
		X0Y5	GTY_Quad_133	GTY_Quad_132, GTY_Quad_131	GTY_Quad_130
		X1Y0	X16 Not Supported	GTY_Quad_222 , GTY_Quad_221	GTY_Quad_220
		X1Y2	GTY_Quad_228, GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X1Y4	GTY_Quad_233, GTY_Quad_232	GTY_Quad_231, GTY_Quad_230	GTY_Quad_229

Table 93: Virtex UltraScale+ Devices Available GT Quads (XCVU9P) (cont'd)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU9P	FLGA2577	X0Y1	GTY_Quad_123	GTY_Quad_122, GTY_Quad_121	GTY_Quad_120
		X0Y3	GTY_Quad_128	GTY_Quad_127, GTY_Quad_126	GTY_Quad_125
		X0Y5	GTY_Quad_133	GTY_Quad_132, GTY_Quad_131	GTY_Quad_130
		X1Y0	GTY_Quad_223, GTY_Quad_222	GTY_Quad_221, GTY_Quad_220	GTY_Quad_219
		X1Y2	GTY_Quad_228, GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X1Y4	GTY_Quad_233, GTY_Quad_232	GTY_Quad_231, GTY_Quad_230	GTY_Quad_229
	FSGD2104	X0Y1	GTY_Quad_123	GTY_Quad_122, GTY_Quad_121	GTY_Quad_120
		X0Y3	GTY_Quad_127	GTY_Quad_126, GTY_Quad_125	GTY_Quad_124
		X0Y5	X16 Not Supported	X8 Not Supported	GTY_Quad_131
		X1Y2	GTY_Quad_228, GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X1Y4	GTY_Quad_233, GTY_Quad_232	GTY_Quad_231, GTY_Quad_230	GTY_Quad_229

Table 94: Virtex UltraScale+ Devices Available GT Quads (XCVU11P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU11P	FLGA2577	X0Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X0Y1	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
		X0Y2	GTY_Quad_235	GTY_Quad_234, GTY_Quad_233	GTY_Quad_232
	FLGB2104	X0Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X0Y1	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
		X0Y2	X16 Not Supported	GTY_Quad_233	GTY_Quad_232
	FLGC2104	X0Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X0Y1	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
		X0Y2	GTY_Quad_235	GTY_Quad_234, GTY_Quad_233	GTY_Quad_232
	FLGF1924	X0Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X0Y1	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
		X0Y2	X16 Not Supported	GTY_Quad_233	GTY_Quad_232
	FSGD2104	X0Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X0Y1	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
		X0Y2	X16 Not Supported	GTY_Quad_233	GTY_Quad_232

Table 95: Virtex UltraScale+ Devices Available GT Quads (XCVU13P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU13P	FHGA2104	X0Y1	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X0Y2	X16 Not Supported	GTY_Quad_231 , GTY_Quad_230	GTY_Quad_229
	FHGB2104	X0Y1	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X0Y2	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
		X0Y3	X16 Not Supported	GTY_Quad_233	GTY_Quad_232
	FHGC2014	X0Y0	X16 Not Supported	GTY_Quad_223 , GTY_Quad_222	GTY_Quad_221
		X0Y1	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X0Y2	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
		X0Y3	X16 Not Supported	GTY_Quad_233	GTY_Quad_232
	FLGA2577	X0Y0	GTY_Quad_223	GTY_Quad_222, GTY_Quad_221	GTY_Quad_220
		X0Y1	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X0Y2	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
		X0Y3	GTY_Quad_235	GTY_Quad_234, GTY_Quad_233	GTY_Quad_232
	FIGD2104	X0Y1	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X0Y2	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
		X0Y3	X16 Not Supported	GTY_Quad_233	GTY_Quad_232
	FSGA2577	X0Y0	GTY_Quad_223	GTY_Quad_222, GTY_Quad_221	GTY_Quad_220
		X0Y1	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X0Y2	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
		X0Y3	GTY_Quad_235	GTY_Quad_234, GTY_Quad_233	GTY_Quad_232

Table 96: Virtex UltraScale+ Devices Available GT Quads (XCVU35P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU35P	FSVH2892	PCIE4C_X0Y0	GTY_Quad_127	GTY_Quad_126, GTY_Quad_125	GTY_Quad_124
		PCIE4C_X0Y1	GTY_Quad_127	GTY_Quad_126, GTY_Quad_125	GTY_Quad_124
		PCIE4C_X1Y1	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4C_X1Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4_X0Y0	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
	FSVH2104	PCIE4C_X0Y0	GTY_Quad_127	GTY_Quad_126, GTY_Quad_125	GTY_Quad_124
		PCIE4C_X0Y1	GTY_Quad_127	GTY_Quad_126, GTY_Quad_125	GTY_Quad_124
		PCIE4C_X1Y1	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4C_X1Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4_X0Y0	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228

Table 97: Virtex UltraScale+ Devices Available GT Quads (XCVU37P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU37P	FSVH2892	PCIE4C_X0Y0	GTY_Quad_127	GTY_Quad_126, GTY_Quad_125	GTY_Quad_124
		PCIE4C_X0Y1	GTY_Quad_127	GTY_Quad_126, GTY_Quad_125	GTY_Quad_124
		PCIE4C_X1Y1	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4C_X1Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4_X0Y0	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
		PCIE4_X0Y1	GTY_Quad_235	GTY_Quad_234, GTY_Quad_233	GTY_Quad_232

Table 98: Virtex UltraScale+ Devices Available GT Quads (XCVU31P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU31P	FSVH1924	PCIE4C_X0Y0	GTY_Quad_127	GTY_Quad_126, GTY_Quad_125	GTY_Quad_124
		PCIE4C_X0Y1	GTY_Quad_127	GTY_Quad_126, GTY_Quad_125	GTY_Quad_124
		PCIE4C_X1Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4C_X1Y1	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224

Table 99: Virtex UltraScale+ Devices Available GT Quads (XCVU33P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU33P	FSVH2104	PCIE4C_X0Y0	GTY_Quad_127	GTY_Quad_126, GTY_Quad_125	GTY_Quad_124
		PCIE4C_X0Y1	GTY_Quad_127	GTY_Quad_126, GTY_Quad_125	GTY_Quad_124
		PCIE4C_X1Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4C_X1Y1	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224

Table 100: Virtex UltraScale+ Devices Available GT Quads (XQVU7P)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XQVU7P	FLRA2104	X0Y1	X16 Not Supported	GTY_Quad_127, GTY_Quad_126	GTY_Quad_125
		X0Y3	X16 Not Supported	GTY_Quad_132, GTY_Quad_131	GTY_Quad_130
		X1Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X1Y2	X16 Not Supported	GTY_Quad_233, GTY_Quad_232	GTY_Quad_231
	FLRB2104	X0Y1	GTY_Quad_128	GTY_Quad_127, GTY_Quad_126	GTY_Quad_125
		X0Y3	GTY_Quad_133	GTY_Quad_132, GTY_Quad_131	GTY_Quad_130
		X1Y0	GTY_Quad_228, GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X1Y2	GTY_Quad_233, GTY_Quad_232	GTY_Quad_231, GTY_Quad_230	GTY_Quad_229

Table 101: Virtex UltraScale+ Devices Available GT Quads (XQVU11P)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XQVU11P	FLRC2104	X0Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		X0Y1	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
		X0Y2	GTY_Quad_235	GTY_Quad_234, GTY_Quad_233	GTY_Quad_232

Table 102: Virtex UltraScale+ Devices Available GT Quads (XQVU29P)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU29P	figd2104	X0Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
	fsga2577	X0Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224

Table 103: Virtex UltraScale+ Devices Available GT Quads (XQVU3P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XQVU3P	FFRC1517	PCIE4_X0Y1	GTY_Quad_128	GTY_Quad_127, GTY_Quad_126	GTY_Quad_125
		PCIE4_X1Y0	GTY_Quad_228, GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_124

Table 104: Virtex UltraScale+ Devices Available GT Quads (XCVU45P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU45P	FSVH2892	PCIE4C_X1Y0	GTY_Quad_127	GTY_Quad_225, GTY_Quad_226	GTY_Quad_224
		PCIE4C_X1Y1	GTY_Quad_127	GTY_Quad_225, GTY_Quad_226	GTY_Quad_224
		PCIE4C_X0Y1	GTY_Quad_127	GTY_Quad_125, GTY_Quad_126	GTY_Quad_124
		PCIE4C_X0Y0	GTY_Quad_127	GTY_Quad_125, GTY_Quad_126	GTY_Quad_124
		PCIE4_X0Y0	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
	FSVH2104	PCIE4C_X1Y0	GTY_Quad_227	GTY_Quad_225, GTY_Quad_226	GTY_Quad_224
		PCIE4C_X1Y1	GTY_Quad_227	GTY_Quad_225, GTY_Quad_226	GTY_Quad_224
		PCIE4C_X0Y1	GTY_Quad_127	GTY_Quad_125, GTY_Quad_126	GTY_Quad_124
		PCIE4C_X0Y0	GTY_Quad_127	GTY_Quad_125, GTY_Quad_126	GTY_Quad_124
		PCIE4_X0Y0	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228

Table 105: Virtex UltraScale+ Devices Available GT Quads (XCVU47P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU47P	FSVH2892	PCIE4C_X1Y0	GTY_Quad_227	GTY_Quad_225, GT Y_Quad_226,	GTY_Quad_224
		PCIE4C_X1Y1	GTY_Quad_227	GTY_Quad_225, GT Y_Quad_226,	GTY_Quad_224
		PCIE4C_X0Y1	GTY_Quad_127	GTY_Quad_125, GT Y_Quad_126,	GTY_Quad_124,
		PCIE4C_X0Y0	GTY_Quad_127	GTY_Quad_125, GT Y_Quad_126,	GTY_Quad_124,
		PCIE4_X0Y1	GTY_Quad_235	GTY_Quad_234, GT Y_Quad_233	GTY_Quad_232
		PCIE4_X0Y0	GTY_Quad_231	GTY_Quad_230, GT Y_Quad_229	GTY_Quad_228

Table 106: Virtex UltraScale+ Devices Available GT Quads (XCVU7P_C)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU7P_C	FLVA2104	PCIE4_X0Y1	X16 Not Supported	GTY_Quad_127 , GTY_Quad_126	GTY_Quad_125
		PCIE4_X0Y3	X16 Not Supported	GTY_Quad_132 , GTY_Quad_131	GTY_Quad_130
		PCIE4_X1Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4_X1Y2	X16 Not Supported	GTY_Quad_233 , GTY_Quad_232	GTY_Quad_231
	FLVB2104	PCIE4_X0Y1	GTY_Quad_128	GTY_Quad_127, GTY_Quad_126	GTY_Quad_125
		PCIE4_X0Y3	GTY_Quad_133	GTY_Quad_132, GTY_Quad_131	GTY_Quad_130
		PCIE4_X1Y0	GTY_Quad_228, GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4_X1Y2	GTY_Quad_233, GTY_Quad_232	GTY_Quad_233, GTY_Quad_232	GTY_Quad_229
	FLVC2104	PCIE4_X0Y1	GTY_Quad_128	GTY_Quad_127, GTY_Quad_126	GTY_Quad_125
		PCIE4_X0Y3	GTY_Quad_133	GTY_Quad_132, GTY_Quad_131	GTY_Quad_130
		PCIE4_X1Y0	GTY_Quad_228 , GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4_X1Y2	GTY_Quad_233, GTY_Quad_232	GTY_Quad_233, GTY_Quad_232	GTY_Quad_229

Table 107: Virtex UltraScale+ Devices Available GT Quads (XQVU9P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XQVU9P	FLQA2104	PCIE4_X0Y1	X16 Not Supported	GTY_Quad_122 , GTY_Quad_121	GTY_Quad_120
		PCIE4_X0Y3	X16 Not Supported	GTY_Quad_127 , GTY_Quad_126	GTY_Quad_125
		PCIE4_X1Y2	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4_X1Y4	X16 Not Supported	GTY_Quad_233 , GTY_Quad_232	GTY_Quad_231
	FLQB2104	PCIE4_X0Y1	GTY_Quad_123	GTY_Quad_122, GTY_Quad_121	GTY_Quad_120
		PCIE4_X0Y3	GTY_Quad_128	GTY_Quad_127, GTY_Quad_126	GTY_Quad_125
		PCIE4_X1Y2	GTY_Quad_228, GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4_X1Y4	GTY_Quad_233 , GTY_Quad_232	GTY_Quad_231, GTY_Quad_230	GTY_Quad_229
	FSQD2104	PCIE4_X0Y1	GTY_Quad_123	GTY_Quad_122, GTY_Quad_121	GTY_Quad_120
		PCIE4_X0Y3	GTY_Quad_127	GTY_Quad_126, GTY_Quad_125	GTY_Quad_124
		PCIE4_X0Y5	X16 Not Supported	X16 Not Supported	GTY_Quad_131
		PCIE4_X1Y2	GTY_Quad_228, GTY_Quad_227	GTY_Quad_226, GT Y_Quad_225	GTY_Quad_224
		PCIE4_X1Y4	GTY_Quad_233, GTY_Quad_232	GTY_Quad_231, GT Y_Quad_230	GTY_Quad_229

Table 108: Virtex UltraScale+ Devices Available GT Quads (XQVU13P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XQVU13P	FHQA2104	PCIE4_X0Y1	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4_X0Y2	X16 Not Supported	GTY_Quad_231 , GTY_Quad_230	GTY_Quad_229
	FHQB2104	PCIE4_X0Y1	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4_X0Y2	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
		PCIE4_X0Y3	X16 Not Supported	GTY_Quad_233	GTY_Quad_232
	FIQD2104	PCIE4_X0Y1	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4_X0Y2	GTY_Quad_231	GTY_Quad_230, GTY_Quad_229	GTY_Quad_228
		PCIE4_X0Y3	X16 Not Supported	GTY_Quad_233	GTY_Quad_232

Table 109: Virtex UltraScale+ Devices Available GT Quads (XQVU37P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XQVU37P	FSQH2892	PCIE4C_X0Y0	GTY_Quad_127	GTY_Quad_126, GTY_Quad_125	GTY_Quad_124
		PCIE4C_X0Y1	GTY_Quad_127	GTY_Quad_126, GTY_Quad_125	GTY_Quad_124
		PCIE4C_X1Y1	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4C_X1Y0	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224

Table 110: Virtex UltraScale+ Devices Available GT Quads (XCVU19P)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU19P	FSVA3824	PCIE4C_X0Y0	X16 Not Supported	GTY_Quad_222, GTY_Quad_221	GTY_Quad_220
		PCIE4C_X0Y1	X16 Not Supported	GTY_Quad_222, GTY_Quad_221	GTY_Quad_220
		PCIE4C_X0Y2	X16 Not Supported	GTY_Quad_227, GTY_Quad_226	GTY_Quad_225
		PCIE4C_X0Y3	X16 Not Supported	GTY_Quad_227, GTY_Quad_226	GTY_Quad_225
		PCIE4C_X0Y4	X16 Not Supported	GTY_Quad_232, GTY_Quad_231	GTY_Quad_230
		PCIE4C_X0Y5	X16 Not Supported	GTY_Quad_232, GTY_Quad_231	GTY_Quad_230
		PCIE4C_X0Y6	X16 Not Supported	GTY_Quad_237, GTY_Quad_236	GTY_Quad_235
		PCIE4C_X0Y7	X16 Not Supported	GTY_Quad_237, GTY_Quad_236	GTY_Quad_235
	FSVB3824	PCIE4C_X0Y0	GTY_Quad_222	GTY_Quad_221, GTY_Quad_220	GTY_Quad_219
		PCIE4C_X0Y1	GTY_Quad_223	GTY_Quad_222 ,GTY_Quad_221	GTY_Quad_220
		PCIE4C_X0Y2	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
		PCIE4C_X0Y3	GTY_Quad_228	GTY_Quad_227 ,GTY_Quad_226	GTY_Quad_225
		PCIE4C_X0Y4	GTY_Quad_233	GTY_Quad_232, GTY_Quad_231	GTY_Quad_230
		PCIE4C_X0Y5	GTY_Quad_234	GTY_Quad_233, GTY_Quad_232	GTY_Quad_231
		PCIE4C_X0Y6	GTY_Quad_237	GTY_Quad_236, GTY_Quad_235	GTY_Quad_234
		PCIE4C_X0Y7	GTY_Quad_238	GTY_Quad_237, GTY_Quad_236	GTY_Quad_235

Kintex UltraScale+ Devices Available GT Quads

The following table shows shows the PCIe® lane 0 GT Quad options available for the different Kintex® UltraScale+™ devices. The GT Quad location is shown using the GT Quad bank number rather than GT XY coordinates. The *UltraScale and UltraScale+ FPGAs Packaging and Pinouts Product Specification* ([UG575](#)) provides a diagram describing the PCIe block locations relative to enabled GT Quads and includes both GT bank numbering and XY coordinates if needed.

Note: The table lists the number of GT Quads available for selection per device, package and the PCIe block available. During IP customization, in the Basic Tab, select **Advanced** mode, select **Enable GT Quad Selection**, and make your GT quad selection.

Note: The selections in bold are the default selections for each device and are typically optimized for boards which have an x16 PCIe connector. If no default is indicated, select the GT Quad nearest to the PCIe integrated block.

Table 111: Kintex UltraScale+ Devices Available GT Quads (XCKU11P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCKU11P	FFVE1517	X0Y3	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X0Y2	GTY_Quad_130	GTY_Quad_129, GTY_Quad_128	GTY_Quad_127
		X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	GTH_Quad_229	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
	FFVA1156	X0Y3	X16 Not Supported	GTY_Quad_130	GTY_Quad_129
		X0Y2	X16 Not Supported	GTY_Quad_130	GTY_Quad_129
		X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	X16 Not Supported	GTH_Quad_228 , GTH_Quad_227	GTH_Quad_226
	FFVD900	X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	X16 Not Supported	GTH_Quad_227	GTH_Quad_226

Table 112: Kintex® UltraScale+™ Devices Available GT Quads (XCKU15P)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCKU15P	FFVE1517	X0Y3	GTY_Quad_132	GTY_Quad_131, GTY_Quad_130	GTY_Quad_129
		X0Y2	GTY_Quad_130	GTY_Quad_129, GTY_Quad_128	GTY_Quad_127
		X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	GTH_Quad_229	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X1Y2	GTH_Quad_231	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228
	FFVA1156	X0Y3	X16 Not Supported	GTY_Quad_130	GTY_Quad_129
		X0Y2	X16 Not Supported	GTY_Quad_130	GTY_Quad_129
		X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	X16 Not Supported	GTH_Quad_228 , GTH_Quad_227	GTH_Quad_226
		X1Y2	X16 Not Supported	GTH_Quad_228 , GTH_Quad_227	GTH_Quad_226
	FFVA1760	X0Y3	GTY_Quad_132	GTY_Quad_131, GTY_Quad_130	GTY_Quad_129
		X0Y2	GTY_Quad_130	GTY_Quad_129, GTY_Quad_128	GTY_Quad_127
		X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	GTH_Quad_229	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X1Y2	GTH_Quad_231	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228
	FFVE1760	X0Y3	GTY_Quad_132	GTY_Quad_131, GTY_Quad_130	GTY_Quad_129
		X0Y2	GTY_Quad_130	GTY_Quad_129, GTY_Quad_128	GTY_Quad_127
		X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	GTH_Quad_229	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X1Y2	GTH_Quad_231	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228

Table 113: Kintex® UltraScale+™ Devices Available GT Quads (XCKU3P)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCKU3P	FFVA676	XOYO	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
	FFVB676	XOYO	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
	FFVD900	XOYO	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
	SFVB784	XOYO	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224

Table 114: Kintex® UltraScale+™ Devices Available GT Quads (XCKU5P)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCKU5P	FFVA676	XOYO	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
	FFVB676	XOYO	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
	FFVD900	XOYO	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224
	SFVB784	XOYO	GTY_Quad_227	GTY_Quad_226, GTY_Quad_225	GTY_Quad_224

Zynq UltraScale+ Devices Available GT Quads

The following table shows the PCIe lane 0 GT Quad options available for the different Zynq® UltraScale+™ devices. The GT Quad location is shown using the GT Quad bank number rather than GT XY coordinates. The *Zynq UltraScale+ Device Packaging and Pinouts Product Specification User Guide (UG1075)*.

Note: The table lists the number of GT Quads available for selection per device, package and the PCIe block available. During IP customization, in the Basic Tab, select **Advanced** mode, select **Enable GT Quad Selection**, and make your GT quad selection.

Note: The selections in bold are the default selections for each device and are typically optimized for boards which have an x16 PCIe connector. If no default is indicated, select the GT Quad nearest to the PCIe integrated block.

Table 115: Zynq UltraScale+ Devices Available GT Quads (XCZU11EG)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU11EG	FFVC1760	X0Y3	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X0Y2	X16 Not Supported	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
	FFVB1517	X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	X16 Not Supported	GTH_Quad_227	GTH_Quad_226
	FFVC1156	X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	X16 Not Supported	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
	FFVF1517	X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	GTH_Quad_229	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226

Table 116: Zynq UltraScale+ Devices Available GT Quads (XCZU17EG)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU17EG	FFVC1760	X0Y3	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X0Y2	X16 Not Supported	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X1Y2	GTH_Quad_231, GTH_Quad_230	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227
	FFVE1924	X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	GTH_Quad_229	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X1Y2	GTH_Quad_230	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227
	FFVB1517	X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	X16 Not Supported	GTH_Quad_227	GTH_Quad_226
	FFVD1760	X0Y3	GTY_Quad_132	GTY_Quad_131, GTY_Quad_130	GTY_Quad_129
		X0Y2	X16 Not Supported	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	GTH_Quad_229	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X1Y2	GTH_Quad_230	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227

Table 117: Zynq UltraScale+ Devices Available GT Quads (XCZU19EG)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU19EG	FFVC1760	X0Y3	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X0Y2	X16 Not Supported	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X1Y2	GTH_Quad_231, GTH_Quad_230	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227
	FFVE1924	X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	GTH_Quad_229	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X1Y2	GTH_Quad_230	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227
	FFVB1517	X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	X16 Not Supported	GTH_Quad_227	GTH_Quad_226
		X1Y2	X16 Not Supported	X8 Not Supported	GTH_Quad_227
	FFVD1760	X0Y3	GTY_Quad_132	GTY_Quad_131, GTY_Quad_130	GTY_Quad_129
		X0Y2	X16 Not Supported	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	GTH_Quad_229	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X1Y2	GTH_Quad_230	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227

Table 118: Zynq UltraScale+ Devices Available GT Quads (XCZU4EV)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU4EV	FBVB900	X0Y0	GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
		X0Y1	GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
	SFVC784	X0Y0	X16 Not Supported	X8 Not Supported	GTH_Quad_224
		X0Y1	X16 Not Supported	X8 Not Supported	GTH_Quad_224

Table 119: Zynq UltraScale+ Devices Available GT Quads (XCZU5EV)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU5EV	FBVB900	X0Y0	GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
		X0Y1	GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
	SFVC784	X0Y0	X16 Not Supported	X8 Not Supported	GTH_Quad_224
		X0Y1	X16 Not Supported	X8 Not Supported	GTH_Quad_224

Table 120: Zynq UltraScale+ Devices Available GT Quads (XCZU7EV)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU7EV	FBVB900	X0Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
	FFVC1156	X0Y0	GTH_Quad_227, GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
		X0Y1	GTH_Quad_227, GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
	FFVF1517	X0Y0	GTH_Quad_227, GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
		X0Y1	GTH_Quad_227, GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223

Table 121: Zynq UltraScale+ Devices Available GT Quads (XCZU21DR)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU21DR	FFVD1156	XOYO	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		XOY1	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128

Table 122: Zynq UltraScale+ Devices Available GT Quads (XCZU25DR)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU25DR	FFVE1156	XOYO	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
	FFVG1517	XOYO	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
	FSVE1156	XOYO	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
	FSVG1517	XOYO	X16 Not Supported	GTY_Quad_129	GTY_Quad_128

Table 123: Zynq UltraScale+ Devices Available GT Quads (XCZU27DR)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU27DR	FFVG1517	XOYO	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		XOY1	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
	FFVE1156	XOYO	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
		XOY1	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
	FSVG1517	XOYO	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		XOY1	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
	FSVE1156	XOYO	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
		XOY1	X16 Not Supported	GTY_Quad_129	GTY_Quad_128

Table 124: Zynq UltraScale+ Devices Available GT Quads (XCZU29DR)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU29DR	FFVF1760	X0Y1	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X0Y0	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
	FSVF1760	X0Y1	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X0Y0	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128

Table 125: Zynq UltraScale+ Devices Available GT Quads (XCZU28DR)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU28DR	FFVG1517	X0Y0	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X0Y1	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
	FFVE1156	X0Y0	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
		X0Y1	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
	FSVG1517	X0Y0	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X0Y1	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
	FSVE1156	X0Y0	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
		X0Y1	X16 Not Supported	GTY_Quad_129	GTY_Quad_128

Table 126: Zynq UltraScale+ Devices Available GT Quads (XCZU4CG)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU4CG	FBVB900	X0Y0	GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
		X0Y1	GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
	SFVC784	X0Y0	X16 Not Supported	X8 Not Supported	GTH_Quad_224
		X0Y1	X16 Not Supported	X8 Not Supported	GTH_Quad_224

Table 127: Zynq UltraScale+ Devices Available GT Quads (XCZU5CG)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU5CG	FBVB900	X0Y0	GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
		X0Y1	GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
	SFVC784	X0Y0	X16 Not Supported	X8 Not Supported	GTH_Quad_224
		X0Y1	X16 Not Supported	X8 Not Supported	GTH_Quad_224

Table 128: Zynq UltraScale+ Devices Available GT Quads (XCZU7CG)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU7CG	FBVB900	X0Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
	FFVC1156	X0Y0	GTH_Quad_227, GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
		X0Y1	GTH_Quad_227, GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
	FFVF1517	X0Y0	GTH_Quad_227, GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
		X0Y1	GTH_Quad_227, GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223

Table 129: Zynq UltraScale+ Devices Available GT Quads (XCZU4EG)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU4EG	FBVB900	X0Y0	GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
		X0Y1	GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
	SFVC784	X0Y0	X16 Not Supported	X8 Not Supported	GTH_Quad_224
		X0Y1	X16 Not Supported	X8 Not Supported	GTH_Quad_224

Table 130: Zynq UltraScale+ Devices Available GT Quads (XCZU5EG)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU5EG	FBVB900	X0Y0	GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
		X0Y1	GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
	SFVC784	X0Y0	X16 Not Supported	X8 Not Supported	GTH_Quad_224
		X0Y1	X16 Not Supported	X8 Not Supported	GTH_Quad_224

Table 131: Zynq UltraScale+ Devices Available GT Quads (XCZU7EG)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU7EG	FBVB900	X0Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
	FFVC1156	X0Y0	GTH_Quad_227, GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
		X0Y1	GTH_Quad_227, GTH_Quad_226	GTH_Quad_225 GTH_Quad_224	GTH_Quad_223
	FFVF1517	X0Y0	GTH_Quad_227, GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
		X0Y1	GTH_Quad_227, GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223

Table 132: Zynq UltraScale+ Devices Available GT Quads (XAZU4EV)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XAZU4EV	SFVC784	X0Y0	X16 Not Supported	X8 Not Supported	GTH_Quad_224
		X0Y1	X16 Not Supported	X8 Not Supported	GTH_Quad_224

Table 133: Zynq UltraScale+ Devices Available GT Quads (XAZU5EV)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XAZU5EV	SFVC784	X0Y0	X16 Not Supported	X8 Not Supported	GTH_Quad_224
		X0Y1	X16 Not Supported	X8 Not Supported	GTH_Quad_224

Table 134: Zynq UltraScale+ Devices Available GT Quads (XQZU19EG)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XQZU19EG	FFRB1517	X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	X16 Not Supported	GTH_Quad_227	GTH_Quad_226
		X1Y2	X16 Not Supported	X8 Not Supported	GTH_Quad_227
	FFRC1760	X0Y3	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X0Y2	X16 Not Supported	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X1Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X1Y1	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X1Y2	GTH_Quad_231, GTH_Quad_230	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227

Table 135: Zynq UltraScale+ Devices Available GT Quads (XQZU5EV)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XQZU5EV	FFRB900	X0Y0	GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
		X0Y1	GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
	SFRC784	X0Y0	X16 Not Supported	X8 Not Supported	GTH_Quad_224
		X0Y1	X16 Not Supported	X8 Not Supported	GTH_Quad_224

Table 136: Zynq UltraScale+ Devices Available GT Quads (XQZU7EV)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XQZU7EV	FFRB900	X0Y0	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_227	GTH_Quad_226, GTH_Quad_225	GTH_Quad_224
	FFRC1156	X0Y0	GTH_Quad_227, GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223
		X0Y1	GTH_Quad_227, GTH_Quad_226	GTH_Quad_225, GTH_Quad_224	GTH_Quad_223

Table 137: Zynq UltraScale+ Devices Available GT Quads (XQZU21DR)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XQZU21DR	FFRD1156	X0Y0	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128
		X0Y1	GTY_Quad_131	GTY_Quad_130, GTY_Quad_129	GTY_Quad_128

Table 138: Zynq UltraScale+ Devices Available GT Quads (XQZU28DR)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XQZU28DR	FFRE1156	X0Y0	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
		X0Y1	X16 Not Supported	GTY_Quad_129	GTY_Quad_128

Table 139: Zynq UltraScale+ Devices Available GT Quads (XCZU39DR)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU39DR	FFVF1760	PCIE4_X0Y0	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
		PCIE4_X0Y1	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
	FSVF1760	PCIE4_X0Y0	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
		PCIE4_X0Y1	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128

Table 140: Zynq UltraScale+ Devices Available GT Quads (XCZU46DR)

Device	Package	PCIE Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU46DR	FFVH1760	PCIE4C_X0Y0	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
		PCIE4C_X0Y1	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
	FSVH1760	PCIE4C_X0Y0	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
		PCIE4C_X0Y1	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128

Table 141: Zynq UltraScale+ Devices Available GT Quads (XCZU47DR)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU47DR	FFVE1156	PCIE4C_X0Y0	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
		PCIE4C_X0Y1	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
	FSVE1156	PCIE4C_X0Y0	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
		PCIE4C_X0Y1	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
	FFVG1517	PCIE4C_X0Y0	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
		PCIE4C_X0Y1	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
	FSVG1517	PCIE4C_X0Y0	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
		PCIE4C_X0Y1	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128

Table 142: Zynq UltraScale+ Devices Available GT Quads (XCZU48DR)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU48DR	FFVE1156	PCIE4C_X0Y0	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
		PCIE4C_X0Y1	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
	FSVE1156	PCIE4C_X0Y0	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
		PCIE4C_X0Y1	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
	FFVG1517	PCIE4C_X0Y0	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
		PCIE4C_X0Y1	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
	FSVG1517	PCIE4C_X0Y0	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
		PCIE4C_X0Y1	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128

Table 143: Zynq UltraScale+ Devices Available GT Quads (XCZU49DR)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU49DR	FFVF1760	PCIE4C_X0Y0	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
		PCIE4C_X0Y1	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
	FSVF1760	PCIE4C_X0Y0	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
		PCIE4C_X0Y1	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128

Table 144: Zynq UltraScale+ Devices Available GT Quads (XCZU58DR)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU58DR	FFVE1156	PCIE4C_X0Y0	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
		PCIE4C_X0Y1	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
	FSVE1156	PCIE4C_X0Y0	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
		PCIE4C_X0Y1	X16 Not Supported	GTY_Quad_129	GTY_Quad_128
	FFVG1517	PCIE4C_X0Y0	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
		PCIE4C_X0Y1	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
	FSVG1517	PCIE4C_X0Y0	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
		PCIE4C_X0Y1	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128

Table 145: Zynq UltraScale+ Devices Available GT Quads (XCZU59DR)

Device	Package	PCIe Blocks	Quads with Max Link Width X16 Support	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCZU59DR	FFVF1760	PCIE4C_X0Y0	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
		PCIE4C_X0Y1	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
	FSVF1760	PCIE4C_X0Y0	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128
		PCIE4C_X0Y1	GTY_Quad_131	GTY_Quad_129, GTY_Quad_130	GTY_Quad_128

Debugging

This appendix includes details about resources available on the Xilinx® Support website and debugging tools.

Finding Help on Xilinx.com

To help in the design and debug process when using the core, the [Xilinx Support web page](#) contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support. The [Xilinx Community Forums](#) are also available where members can learn, participate, share, and ask questions about Xilinx solutions.

Documentation

This product guide is the main document associated with the core. This guide, along with documentation related to all products that aid in the design process, can be found on the [Xilinx Support web page](#) or by using the Xilinx® Documentation Navigator. Download the Xilinx Documentation Navigator from the [Downloads page](#). For more information about this tool and the features available, open the online help after installation.

Answer Records

Answer Records include information about commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a Xilinx product. Answer Records are created and maintained daily ensuring that users have access to the most accurate information available.

Answer Records for this core can be located by using the Search Support box on the main [Xilinx support web page](#). To maximize your search results, use keywords such as:

- Product name
- Tool message(s)
- Summary of the issue encountered

A filter search is available after results are returned to further target the results.

Master Answer Record for the Integrated Block for PCIe

AR [65751](#).

Technical Support

Xilinx provides technical support on the [Xilinx Community Forums](#) for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support if you do any of the following:

- Implement the solution in devices that are not defined in the documentation.
- Customize the solution beyond that allowed in the product documentation.
- Change any section of the design labeled DO NOT MODIFY.

To ask questions, navigate to the [Xilinx Community Forums](#).

Hardware Debug

Integrated Debug Options

The UltraScale Integrated Block PCIe cores comes with three debug options integrated with in the core:

- Viz
- JTAG debugger
- Insystem IBERT and Descrambler mode (for Gen3 Link Speeds)

These options are available in the Vivado IP customization page. See Xilinx Answer Record [68134](#) for detailed description of these debug options.

Transceiver Control and Status Ports

The following table describes the ports used to debug transceiver-related issues.



IMPORTANT! The ports in the Transceiver Control And Status Interface must be driven in accordance with the appropriate GT user guide. Using the input signals listed in the following table might result in unpredictable behavior of the IP core.

Table 146: Ports Used for Transceiver Debug

Port	I/O	Width	Description
gt_pcieuerratedone	I	1	Connects to PCIEUERRATEDONE on transceiver channel primitives
gt_loopback	I	3	Connects to LOOPBACK on transceiver channel primitives
gt_txprbsforceerr	I	1	Connects to TXPRBSFORCEERR on transceiver channel primitives
gt_txinhibit	I	1	Connects to TXINHIBIT on transceiver channel primitives
gt_txprbssel	I	4	PRBS input
gt_rxprbssel	I	4	PRBS input
gt_rxprbscntreset	I	1	Connects to RXPRBSCNTRESET on transceiver channel primitives
gt_txelecidle	O	1	Connects to TXELECIDLE on transceiver channel primitives
gt_txresetdone	O	1	Connects to TXRESETDONE on transceiver channel primitives
gt_rxresetdone	O	1	Connects to RXRECLKOUT on transceiver channel primitives
gt_rxpmaresetdone	O	1	Connects to TXPMARESETDONE on transceiver channel primitives
gt_txphaligndone	O	1	Connects to TXPHALIGNDONE of transceiver channel primitives
gt_txphinitdone	O	1	Connects to TXPHINITDONE of transceiver channel primitives
gt_txdlysresetdone	O	1	Connects to TXDLYSRESETDONE of transceiver channel primitives
gt_rxphaligndone	O	1	Connects to RXPHALIGNDONE of transceiver channel primitives
gt_rxdlysresetdone	O	1	Connects to RXDLYSRESETDONE of transceiver channel primitives
gt_rxsyncdone	O	1	Connects to RXSYNCDONE of transceiver channel primitives
gt_eyescandataerror	O	1	Connects to EYESCANDATAERROR on transceiver channel primitives
gt_rxprbserr	O	1	Connects to RXPRBSERR on transceiver channel primitives
gt_dmonitorout	O	16	Connects to DMONITOROUT on transceiver channel primitives
gt_rxcommadet	O	1	Connects to RXCOMMADETEN on transceiver channel primitives
gt_phystatus	O	1	Connects to PHYSTATUS on transceiver channel primitives
gt_rxvalid	O	1	Connects to RXVALID on transceiver channel primitives
gt_rxcdrlock	O	1	Connects to RXCDRLOCK on transceiver channel primitives
gt_pcierateidle	O	1	Connects to PCIERATEIDLE on transceiver channel primitives
gt_pcieuerratertestart	O	1	Connects to PCIEUERRATESTART on transceiver channel primitives
gt_gtpowergood	O	1	Connects to GTPOWERGOOD on transceiver channel primitives
gt_cplllock	O	1	Connects to CPLLLOCK on transceiver channel primitives
gt_rxoutclk	O	1	Connects to RXOUTCLK on transceiver channel primitives

Table 146: Ports Used for Transceiver Debug (cont'd)

Port	I/O	Width	Description
gt_rxrecclkout	O	1	Connects to RXRECCLKOUT on transceiver channel primitives
gt_qpll1lock	O	1	Connects to QPLL1LOCK on transceiver common primitives
gt_rxstatus	O	3	Connects to RXSTATUS on transceiver channel primitives
gt_rxbufstatus	O	3	Connects to RXBUFSTATUS on transceiver channel primitives
gt_bufgtdiv	O	9	Connects to BUFGTDIV on transceiver channel primitives
phy_txeq_ctrl	O	2	PHY TX Equalization control bits
phy_txeq_preset	O	4	PHY TX Equalization Preset bits
phy_RST_fsm	O	4	PHY RST FSM state bits
phy_txeq_fsm	O	3	PHY RX Equalization FSM state bits (Gen3)
phy_rxeq_fsm	O	3	PHY TX Equalization FSM state bits (Gen3)
phy_RST_IDLE	O	1	PHY is in IDLE state
phy_rrst_n	O	1	Synchronized reset generation by sys_clk
phy_prst_n	O	1	Synchronized reset generation by pipe_clk
gt_gen34_eios_det	O	1	connects to rxctrl0_out port of the gtwizard ip
gt_txoutclk	O	1	connects to TXOUTCLK of the transceiver channel
gt_txoutclkfabric	O	1	connects to TXOUTCLKFABRIC of the transceiver channel
gt_rxoutclkfabric	O	1	connects to RXOUTCLKFABRIC of the transceiver channel
gt_txoutclkpcs	O	1	connects to TXOUTCLKPCS of the transceiver channel
gt_rxoutclkpcs	O	1	connects to RXOUTCLKPCS of the transceiver channel
gt_txpmareset	I	1	connects to TXPMARESET of the transceiver channel
gt_rxpmareset	I	1	connects to RXPMARESET of the transceiver channel
gt_txpcreset	I	1	connects to TXPCRESET of the transceiver channel
gt_rxpcreset	I	1	connects to RXPCRESET of the transceiver channel
gt_rxbufreset	I	1	connects to RXBUFRESET of the transceiver channel
gt_rxcdrrreset	I	1	connects to RXDRRESET of the transceiver channel
gt_rxdfelpmreset	I	1	connects to RXDFELPMRESET of the transceiver channel
gt_txprogdivresetdone	O	1	connects to TXPROGDIVRESETDONE of the transceiver channel
gt_txpmaresetdone	O	1	connects to TXPMARESETDONE of the transceiver channel
gt_txsyncdone	O	1	connects to TXSYNCDONE of the transceiver channel
gt_rxprbslocked	O	1	connects to RXPRBSLOCKED of the transceiver channel
gt_dmonfiforeset	I	1	connects to DMONFIRESET of the transceiver channel
gt_dmonitorclk	I	1	connects to DMONITORCLK of the transceiver channel
gt_qpll0lock	O	1	connects to QPLL0LOCK of the transceiver channel

PCIe DRP Ports

The following table lists the signals that are available when the PCIe DRP Ports option is selected.

Table 147: PCIe DRP Ports

Name	I/O	Width	Description
drp_addr	I	10 bits	PCIe DRP address
drp_en	I	1 bit	PCIe DRP enable
drp_di	I	16 bits	PCIe DRP data in
drp_do	O	16 bits	PCIe DRP data out
drp_rdy	O	1 bit	PCIe DRP ready
drp_we	I	1 bit	PCIe DRP write/read

GT DRP Ports

The following table lists the signals that are available when the GT Channel DRP parameter is enabled.

Table 148: GT DRP Ports

Name	I/O	Width	Description
ext_ch_gt_drpaddr	I	Number Of Lanes x 10	GT Wizard DRP address
ext_ch_gt_drpen	I	Number Of Lanes x 1	GT Wizard DRP enable
ext_ch_gt_drpdi	I	Number Of Lanes x 16	GT Wizard DRP data in
ext_ch_gt_drpdo	O	Number Of Lanes x 16	GT Wizard DRP data out
ext_ch_gt_drprdy	O	Number Of Lanes x 1	GT Wizard DRP ready
ext_ch_gt_drpwe	I	Number Of Lanes x 1	GT Wizard DRP write/read

Using the Xilinx Virtual Cable to Debug

The Xilinx® Virtual Cable (XVC) allows the Vivado® Design Suite to connect to FPGA debug cores through non-JTAG interfaces. The standard Vivado® Design Suite debug feature uses JTAG to connect to physical hardware FPGA resources and perform debug through Vivado. This section focuses on using XVC to perform debug over a PCIe® link rather than the standard JTAG debug interface. This is referred to as XVC-over-PCIe and allows for Vivado ILA waveform capture, VIO debug control, and interaction with other Xilinx debug cores using the PCIe link as the communication channel.

XVC-over-PCIe should be used to perform FPGA debug remotely using the Vivado Design Suite debug feature when JTAG debug is not available. This is commonly used for data center applications where the FPGA is connected to a PCIe Host system without any other connections to the hardware device.

Using debug over XVC requires software, driver, and FPGA hardware design components. Since there is an FPGA hardware design component to XVC-over-PCIe debug, you cannot perform debug until the FPGA is already loaded with an FPGA hardware design that implements XVC-over-PCIe and the PCIe link to the Host PC is established. This is normally accomplished by loading an XVC-over-PCIe enabled design into the configuration flash on the board prior to inserting the card into the data center location. Since debug using XVC-over-PCIe is dependent on the PCIe communication channel this should not be used to debug PCIe link related issue.



IMPORTANT! XVC only provides connectivity to the debug cores within the FPGA. It does not provide the ability to program the device or access device JTAG and configuration registers. These operations can be performed through other standard Xilinx interfaces or peripherals such as the PCIe MCAP VSEC and HWICAP IP.

Overview

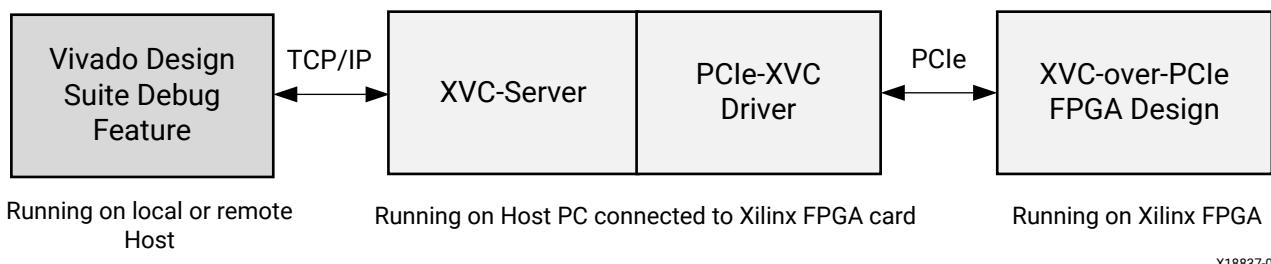
The main components that enable XVC-over-PCIe debug are as follows:

- Host PC XVC-Server application
- Host PC PCIe-XVC driver

- XVC-over-PCIe enabled FPGA design

These components are provided as a reference on how to create XVC connectivity for Xilinx FPGA designs. These three components are shown in the following figure and connect to the Vivado Design Suite debug feature through a TCP/IP socket.

Figure 141: XVC-over-PCIe Software and Hardware Components



Host PC XVC-Server Application

The `hw_server` application is launched by Vivado Design Suite when using the debug feature. Through the Vivado IDE you can connect `hw_server` to local or remote FPGA targets. This same interface is used to connect to local or remote PCIe-XVC targets as well. The Host PCIe XVC-Server application connects to the Xilinx `hw_server` using TCP/IP socket. This allows Vivado (using `hw_server`) and the XVC-Server application to be running on the same PC or separate PCs connected through Ethernet. The XVC-Server application needs to be run on a PC that is directly connected to the FPGA hardware resource. In this scenario the FPGA hardware is connected through PCIe® to a Host PC. The XVC-Server application connects to the FPGA hardware device through the PCIe-XVC driver that is also running on the Host PC.

Host PC XVC-over-PCIe Driver

The XVC-over-PCIe driver provides connectivity to the PCIe enabled FPGA hardware resource that is connected to the Host PC. As such this is provided as a Linux kernel mode driver to access the PCIe hardware device, which is available in the following location,

`<Vivado_Installation_Path>/data/xicom/driver/pcie/xvc_pcie.zip`. The necessary components of this driver must be added to the driver that is created for a specific FPGA platform. The driver implements the basic functions needed by the XVC-Server application to communicate with the FPGA via PCIe.

XVC-over-PCIe Enabled FPGA Design

Traditionally Vivado® debug is performed over JTAG. By default, Vivado tool automation connects the Xilinx debug cores to the JTAG BSCAN resource within the FPGA to perform debug. In order to perform XVC-over-PCIe debug, this information must be transmitted over the PCIe link rather than over the JTAG cable interface. The Xilinx Debug Bridge IP allows you to connect the debug network to PCIe through either the PCIe extended configuration interface (PCIe-XVC-VSEC) or through a PCIe BAR via an AXI4-Lite Memory Mapped interface (AXI-XVC).

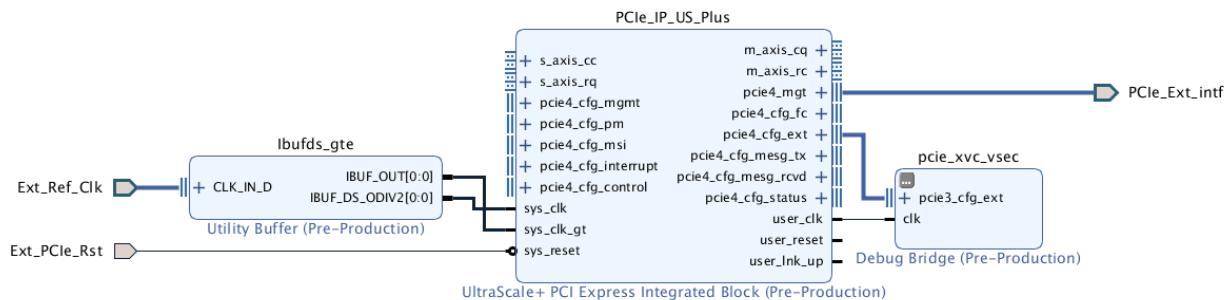
The Debug Bridge IP, when configured for **From PCIe to BSCAN** or **From AXI to BSCAN**, provides a connection point for the Xilinx® debug network from either the PCIe Extended Capability or AXI4-Lite interfaces respectively. Vivado tool automation connects this instance of the Debug Bridge to the Xilinx debug cores found in the design rather than connecting them to the JTAG BSCAN interface. There are design trade-offs to connecting the debug bridge to the PCIe Extended Configuration Space or AXI4-Lite. The following sections describe the implementation considerations and register map for both implementations.

XVC-over-PCIe Through PCIe Extended Configuration Space (PCIe-XVC-VSEC)

Using the PCIe-XVC-VSEC approach, the Debug Bridge IP uses a PCIe Vendor Specific Extended Capability (VSEC) to implement the connection from PCIe to the Debug Bridge IP. The PCIe extended configuration space is set up as a linked list of extended capabilities that are discoverable from a Host PC. This is specifically valuable for platforms where one version of the design implements the PCIe-XVC-VSEC and another design implementation does not. The linked list can be used to detect the existence or absence of the PCIe-XVC-VSEC and respond accordingly.

The PCIe Extended Configuration Interface uses PCIe configuration transactions rather than PCIe memory BAR transactions. While PCIe configuration transactions are much slower, they do not interfere with PCIe memory BAR transactions at the PCIe IP boundary. This allows for separate data and debug communication paths within the FPGA. This is ideal if you expect to debug the datapath. Even if the datapath becomes corrupt or halted, the PCIe Extended Configuration Interface can remain operational to perform debug. The following figure describes the connectivity between the PCIe IP and the Debug Bridge IP to implement the PCIe-XVC-VSEC.

Figure 142: XVC-over-PCIe with PCIe Extended Capability Interface

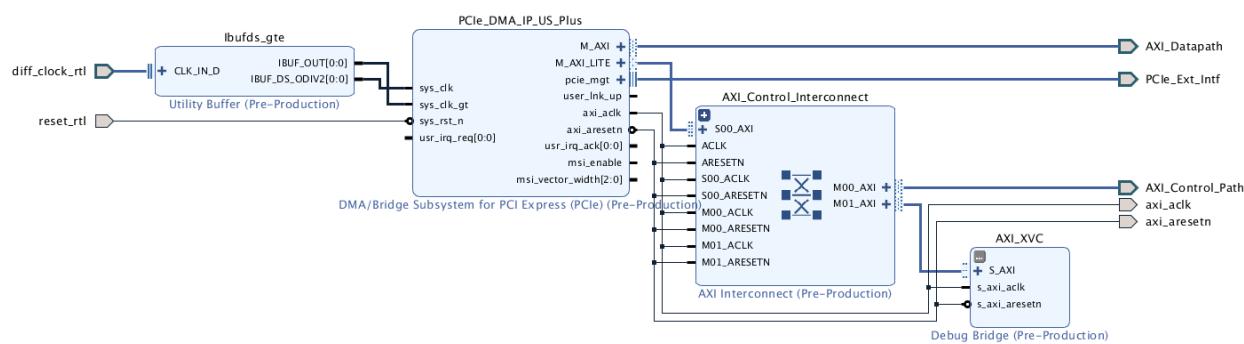


Note: Although the previous figure shows the UltraScale+™ Integrated Block for PCIe IP, other PCIe IP (that is, the UltraScale™ Integrated Block for PCIe, AXI Bridge for PCIe, or PCIe DMA IP) can be used interchangeably in this diagram.

XVC-over-PCIe Through AXI (AXI-XVC)

Using the AXI-XVC approach, the Debug Bridge IP connects to the PCIe IP through an AXI Interconnect IP. The Debug Bridge IP connects to the AXI Interconnect like other AXI4-Lite Slave IPs and similarly requires that a specific address range be assigned to it. Traditionally the debug_bridge IP in this configuration is connected to the control path network rather than the system datapath network. The following figure describes the connectivity between the DMA Subsystem for PCIe IP and the Debug Bridge IP for this implementation.

Figure 143: XVC over PCIe with AXI4-Lite Interface



Note: Although the previous figure shows the PCIe DMA IP, any AXI-enabled PCIe IP can be used interchangeably in this diagram.

The AXI-XVC implementation allows for higher speed transactions. However, XVC debug traffic passes through the same PCIe ports and interconnect as other PCIe control path traffic, making it more difficult to debug transactions along this path. As a result, the AXI-XVC debug should be used to debug a specific peripheral or a different AXI network rather than attempting to debug datapaths that overlap with the AXI-XVC debug communication path.

XVC-over-PCIe Register Map

The PCIe-XVC-VSEC and AXI-XVC have a slightly different register map that must be taken into account when designing XVC drivers and software. The register maps in the following tables show the byte-offset from the base address.

- The PCIe-XVC-VSEC base address must fall within the valid range of the PCIe Extended Configuration space. This is specified in the Debug Bridge IP configuration.
- The base address of an AXI-XVC Debug Bridge is the offset for the Debug Bridge IP peripheral that was specified in the Vivado Address Editor.

The following tables describe the register map for the Debug Bridge IP as an offset from the base address when configured for the **From PCIe-Ext to BSCAN** or **From AXI to BSCAN** modes.

Table 149: Debug Bridge for XVC-PCIe-VSEC Register Map

Register Offset	Register Name	Description	Register Type
0x00	PCIe Ext Capability Header	PCIe defined fields for VSEC use.	Read Only
0x04	PCIe VSEC Header	PCIe defined fields for VSEC use.	Read Only
0x08	XVC Version Register	IP version and capabilities information.	Read Only
0x0C	XVC Shift Length Register	Shift length.	Read Write
0x10	XVC TMS Register	TMS data.	Read Write
0x14	XVC TDIO Register	TDO/TDI data.	Read Write
0x18	XVC Control Register	General control register.	Read Write
0x1C	XVC Status Register	General status register.	Read Only

Table 150: Debug Bridge for AXI-XVC Register Map

Register Offset	Register Name	Description	Register Type
0x00	XVC Shift Length Register	Shift length.	Read Write
0x04	XVC TMS Register	TMS data.	Read Write
0x08	XVC TDI Register	TDI data.	Read Write
0x0C	XVC TDO Register	TDO data.	Read Only
0x10	XVC Control Register	General control register.	Read Write
0x14	XVC Status Register	General status register.	Read Only
0x18	XVC Version Register	IP version and capabilities information.	Read Only

PCIe Ext Capability Header

This register is used to identify the PCIe-XVC-VSEC added to a PCIe design. The fields and values in the PCIe Ext Capability Header are defined by PCI-SIG and are used to identify the format of the extended capability and provide a pointer to the next extended capability, if applicable. When used as a PCIe-XVC-VSEC, the appropriate PCIe ID fields should be evaluated prior to interpretation. These can include PCIe Vendor ID, PCIe Device ID, PCIe Revision ID, Subsystem Vendor ID, and Subsystem ID. The provided drivers specifically check for a PCIe Vendor ID that matches Xilinx (0x10EE) before interpreting this register. The following table describes the fields within this register.

Table 151: PCIe Ext Capability Header Register Description

Bit Location	Field	Description	Initial Value	Type
15:0	PCIe Extended Capability ID	This field is a PCI-SIG defined ID number that indicates the nature and format of the Extended Capability. The Extended Capability ID for a VSEC is 0x000B	0x000B	Read Only
19:16	Capability Version	This field is a PCI-SIG defined version number that indicates the version of the capability structure present. Must be 0x1 for this version of the specification.	0x1	Read Only
31:20	Next Capability Offset	This field is passed in from the user and contains the offset to the next PCI Express Capability structure or 0x000 if no other items exist in the linked list of capabilities. For Extended Capabilities implemented in the PCIe extended configuration space, this value must always be within the valid range of the PCIe Extended Configuration space.	0x000	Read Only

PCIe VSEC Header (PCIe-XVC-VSEC only)

This register is used to identify the PCIe-XVC-VSEC when the Debug Bridge IP is in this mode. The fields are defined by PCI-SIG, but the values are specific to the Vendor ID (0x10EE for Xilinx). The PCIe Ext Capability Header register values should be qualified prior to interpreting this register.

Table 152: PCIe XVC VSEC Header Register Description

Bit Location	Field	Description	Initial Value	Type
15:0	VSEC ID	This field is the ID value that can be used to identify the PCIe-XVC-VSEC and is specific to the Vendor ID (0x10EE for Xilinx).	0x0008	Read Only
19:16	VSEC Rev	This field is the Revision ID value that can be used to identify the PCIe-XVC-VSEC revision.	0x0	Read Only
31:20	VSEC Length	This field indicates the number of bytes in the entire PCIe-XVC-VSEC structure, including the PCIe Ext Capability Header and PCIe VSEC Header registers.	0x020	Read Only

XVC Version Register (PCIe-XVC-VSEC only)

This register is populated by the Xilinx tools and is used by the Vivado Design Suite to identify the specific features of the Debug Bridge IP that is implemented in the hardware design.

XVC Shift Length Register

This register is used to set the scan chain shift length within the debug scan chain.

XVC TMS Register

This register is used to set the TMS data within the debug scan chain.

XVC TDO/TDI Data Register(s)

This register is used for TDO/TDI data access. When using PCIePCI-XVC-VSEC, these two registers are combined into a single field. When using AXI-XVC, these are implemented as two separate registers.

XVC Control Register

This register is used for XVC control data.

XVC Status Register

This register is used for XVC status information.

XVC Driver and Software

Example XVC driver and software has been provided with the Vivado Design Suite installation, which is available at the following location: <Vivado_Installation_Path>/data/xicom/driver/pcie/xvc_pcie.zip. This should be used for reference when integrating the XVC capability into Xilinx FPGA platform design drivers and software. The provided Linux kernel mode driver and software implement XVC-over-PCIe debug for both PCIe-XVC-VSEC and AXI-XVC debug bridge implementations.

When operating in PCIe-XVC-VSEC mode, the driver will initiate PCIe configuration transactions to interface with the FPGA debug network. When operating in AXI-XVC mode, the driver will initiate 32-bit PCIe Memory BAR transactions to interface with the FPGA debug network. By default, the driver will attempt to discover the PCIe-XVC-VSEC and use AXI-XVC if the PCIe-XVC-VSEC is not found in the PCIe configuration extended capability linked list.

The driver is provided in the data directory of the Vivado installation as a .zip file. This .zip file should be copied to the Host PC connected through PCIe to the Xilinx FPGA and extracted for use. README.txt files have been included; review these files for instructions on installing and running the XVC drivers and software.

Special Considerations for Tandem or Dynamic Function eXchange Designs

Tandem Configuration and Dynamic Function eXchange (DFX) designs may require additional considerations as these flows partition the physical resources into separate regions. These physical partitions should be considered when adding debug IPs to a design, such as VIO, ILA, MDM, and MIG-IP. A Debug Bridge IP configured for **From PCIe-ext to BSCAN** or **From AXI to BSCAN** should only be placed into the static partition of the design. When debug IPs are used inside of a DFX or Tandem Field Updates region, an additional debug BSCAN interface should be added to the dynamic region module definition and left unconnected in the dynamic region module instantiation.

To add the BSCAN interface to the Reconfigurable Partition definition the appropriate ports and port attributes should be added to the Reconfigurable Partition definition. The sample Verilog provided below can be used as a template for adding the BSCAN interface to the port declaration.

```
...
// BSCAN interface definition and attributes.
// This interface should be added to the DFX module definition
// and left unconnected in the DFX module instantiation.
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN_drck" *)
(* DEBUG="true" *)
input S_BSCAN_drck,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN_shift" *)
(* DEBUG="true" *)
input S_BSCAN_shift,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN_tdi" *)
(* DEBUG="true" *)
input S_BSCAN_tdi,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN_update" *)
(* DEBUG="true" *)
input S_BSCAN_update,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN_sel" *)
(* DEBUG="true" *)
input S_BSCAN_sel,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN_tdo" *)
(* DEBUG="true" *)
output S_BSCAN_tdo,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN_tms" *)
(* DEBUG="true" *)
input S_BSCAN_tms,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN_tck" *)
(* DEBUG="true" *)
input S_BSCAN_tck,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN_runttest" *)
(* DEBUG="true" *)
input S_BSCAN_runttest,
```

```
( * X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN reset" *)
(* DEBUG= "true" *)
input S_BSCAN_reset,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN capture" *)
(* DEBUG= "true" *)
input S_BSCAN_capture,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN bscanid_en" *)
(* DEBUG= "true" *)
input S_BSCAN_bscanid_en,
....
```

When `link_design` is run, the exposed ports are connected to the static portion of the debug network through tool automation. The ILAs are also connected to the debug network as required by the design. There might also be an additional `dbg_hub` cell that is added at the top level of the design. For Tandem with Field Updates designs, the `dbg_hub` and tool inserted clock buffer(s) must be added to the appropriate design partition. The following is an example of the Tcl commands that can be run after `opt_design` to associate the `dbg_hub` primitives with the appropriate design partitions.

```
# Add the inserted dbg_hub cell to the appropriate design partition.
set_property HD.TANDEM_IP_PBLOCK Stage1_Main [get_cells dbg_hub]
# Add the clock buffer to the appropriate design partition.
set_property HD.TANDEM_IP_PBLOCK Stage1_Config_IO [get_cells
dma_pcie_0_support_i/
pcie_ext_cap_i/vsec_xvc_inst/vsec_xvc_dbg_bridge_inst/inst/bsip/ins
t/USE_SOFTBSCAN.U_TAP_TCKBUFG]
```

Using the PCIe-XVC-VSEC Example Design

The PCIe-XVC-VSEC has been integrated into the PCIe example design as part of the Advanced settings for the UltraScale+™ Integrated Block for PCIe IP. This section provides instruction of how to generate the PCIe example design with the PCIe-XVC-VSEC, and then debug the FPGA through PCIe using provided XVC drivers and software. This is an example for using XVC in customer applications. The FPGA design, driver, and software elements will need to be integrated into customer designs.

Generating a PCIe-XVC-VSEC Example Design

The PCIe-XVC-VSEC can be added to the UltraScale+™ PCIe example design by selecting the following options.

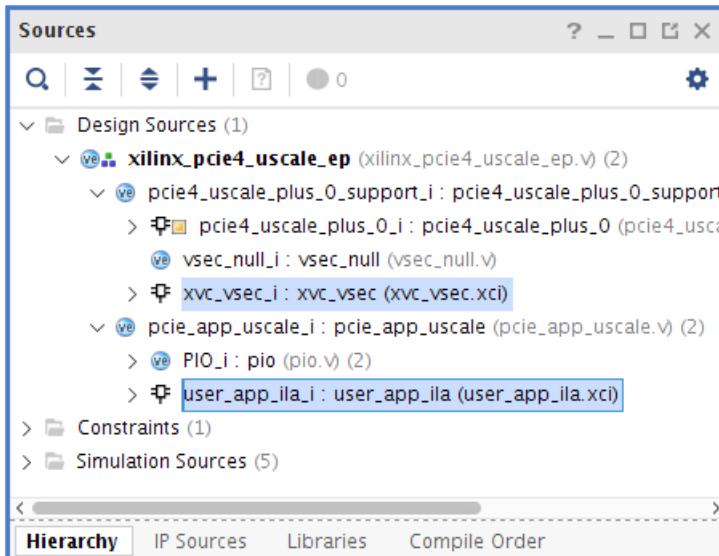
1. Configure the core to the desired configuration.
2. On the Basic tab, select the **Advanced** Mode.

3. On the Adv. Options-3 tab:
 - a. Select the **PCI Express Extended Configuration Space Enable** checkbox to enable the PCI Express extended configuration interface. This is where additional extended capabilities can be added to the PCI Express core.
 - b. Select the **Add the PCIe-XVC-VSEC to the Example Design** checkbox to enable the PCIe-XVC-VSEC in the example design generation.
4. Verify the other configuration selections for the PCIe IP. The following selections are needed to configure the driver for your hardware implementation:
 - PCIe Vendor ID (`0x10EE` for Xilinx)
 - PCIe Device ID (dependent on user selection)
5. Click **OK** to finalize the selection and generate the IP.
6. Generate the output products for the IP as desired for your application.
7. In the Sources window, right-click the IP and select **Open IP Example Design**.
8. Select a directory for generating the example design, and select **OK**.

After being generated, the example design shows that:

- the PCIe IP is connected to `xvc_vsec` within the support wrapper, and
- an ILA IP is added to the user application portion of the design.

This demonstrates the desired connectivity for the hardware portion of the FPGA design. Additional debug cores can be added as required by your application.



Note: Although the previous figure shows to the UltraScale+ Integrated Block for PCIe IP, the example design hierarchy is the same for other PCIe IPs.

9. Double-click the Debug Bridge IP identified as `xvc_vsec` to view the configuration option for this IP. Make note of the following configuration parameters because they will be used to configure the driver.

- PCIe XVC VSEC ID (default `0x0008`)
- PCIe XVC VSEC Rev ID (default `0x0`)



IMPORTANT! Do not modify these parameter values when using a Xilinx Vendor ID or provided XVC drivers and software. These values are used to detect the XVC extended capability. (See the PCIe specification for additional details.)

10. In the Flow Navigator, click **Generate Bitstream** to generate a bitstream for the example design project. This bitstream will be then be loaded onto the FPGA board to enable XVC debug over PCIe.

After the XVC-over-PCIe hardware design has been completed, an appropriate XVC enabled PCIe driver and associated XVC-Server software application can be used to connect the Vivado Design Suite to the PCIe connected FPGA. Vivado can connect to an XVC-Server application that is running local on the same Machine or remotely on another machine using a TCP/IP socket.

System Bring-Up

The first step is to program the FPGA and power on the system such that the PCIe link is detected by the host system. This can be accomplished by either:

- Programming the design file into the flash present on the FPGA board, or
- Programming the device directly via JTAG.

If the card is powered by the Host PC, it will need to be powered on to perform this programming using JTAG and then re-started to allow the PCIe link to enumerate. After the system is up and running, you can use the Linux `lspci` utility to list out the details for the FPGA-based PCIe device.

Compiling and Loading the Driver

The provided PCIe drivers and software should be customized to a specific platform. To accomplish this, drivers and software are normally developed to verify the Vendor ID, Device ID, Revision ID, Subsystem Vendor ID, and Subsystem ID before attempting to access device-extended capabilities or peripherals like the PCIe-XVC-VSEC or AXI-XVC. Because the provided driver is generic, it only verifies the Vendor ID and Device ID for compatibility before attempting to identify the PCIe-XVC-VSEC or AXI-XVC peripheral.

The XVC driver and software are provide as a ZIP file included with the Vivado Design Suite installation.

1. Copy the ZIP file from the Vivado install directory to the FPGA connected Host PC and extract (unzip) its contents. This file is located at the following path within the Vivado installation directory.

```
XVC Driver and SW Path: .../data/xicom/driver/pcie/xvc_PCIE.zip
```

The README.txt files within the driver_* and xvcserver directories identify how to compile, install, and run the XVC drivers and software, and are summarized in the following steps. Follow the following steps after the driver and software files have been copied to the Host PC and you are logged in as a user with root permissions.

2. Modify the variables within the driver_*/xvc_PCIE_user_config.h file to match your hardware design and IP settings. Consider modifying the following variables:

- **PCIE_VENDOR_ID**: The PCIe Vendor ID defined in the PCIe® IP customization.
- **PCIE_DEVICE_ID**: The PCIe Device ID defined in the PCIe® IP customization.
- **Config_space**: Allows for the selection between using a PCIe-XVC-VSEC or an AXI-XVC peripheral. The default value of AUTO first attempts to discover the PCIe-XVC-VSEC, then attempts to connect to an AXI-XVC peripheral if the PCIe-XVC-VSEC is not found. A value of CONFIG or BAR can be used to explicitly select between PCIe®-XVC-VSEC and AXI-XVC implementations, as desired.
- **config_vsec_id**: The PCIe XVC VSEC ID (default 0x0008) defined in the Debug Bridge IP when the Bridge Type is configured for From PCIe to BSCAN. This value is only used for detection of the PCIe®-XVC-VSEC.
- **config_vsec_rev**: The PCIe XVC VSEC Rev ID (default 0x0) defined in the Debug Bridge IP when the Bridge Type is configured for From PCIe to BSCAN. This value is only used for detection of the PCIe-XVC-VSEC.
- **bar_index**: The PCIe BAR index that should be used to access the Debug Bridge IP when the Bridge Type is configured for From AXI to BSCAN. This BAR index is specified as a combination of the PCIe IP customization and the addressable AXI peripherals in your system design. This value is only used for detection of an AXI-XVC peripheral.
- **bar_offset**: PCIe BAR Offset that should be used to access the Debug Bridge IP when the Bridge Type is configured for From AXI to BSCAN. This BAR offset is specified as a combination of the PCIe IP customization and the addressable AXI peripherals in your system design. This value is only used for detection of an AXI-XVC peripheral.

3. Move the source files to the directory of your choice. For example, use:

```
/home/username/xil_xvc or /usr/local/src/xil_xvc
```

4. Make sure you have root permissions and change to the directory containing the driver files.

```
# cd /driver_*/
```

5. Compile the driver module:

```
# make install
```

The kernel module object file will be installed as:

```
/lib/modules/[KERNEL_VERSION]/kernel/drivers/pci/pcie/Xilinx/  
xil_xvc_driver.ko
```

6. Run the `depmod` command to pick up newly installed kernel modules:

```
# depmod -a
```

7. Make sure no older versions of the driver are loaded:

```
# modprobe -r xil_xvc_driver
```

8. Load the module:

```
# modprobe xil_xvc_driver
```

If you run the `dmesg` command, you will see the following message:

```
kernel: xil_xvc_driver: Starting...
```

Note: You can also use `insmod` on the kernel object file to load the module:

```
# insmod xil_xvc_driver.ko
```

However, this is not recommended unless necessary for compatibility with older kernels.

9. The resulting character file, `/dev/xil_xvc/cfg_ioc0`, is owned by user root and group root, and it will need to have permissions of 660. Change permissions on this file if it does not allow the application to interact with the driver.

```
# chmod 660 /dev/xil_xvc/cfg_ioc0
```

10. Build the simple test program for the driver:

```
# make test
```

11. Run the test program:

```
# ./driver_test/verify_xil_xvc_driver
```

You should see various successful tests of differing lengths, followed by the following message:

```
"XVC PCIE Driver Verified Successfully!"
```

Compiling and Launching the XVC-Server Application

The XVC-Server application provides the connection between the Vivado HW server and the XVC enabled PCIe device driver. The Vivado Design Suite connects to the XVC-Server using TCP/IP. The desired port number will need to be exposed appropriately through the firewalls for your network. The following steps can be used to compile and launch the XVC software application, using the default port number of 10200.

1. Make sure the firewall settings on the system expose the port that will be used to connect to the Vivado Design Suite. For this example, port 10200 is used.

2. Make note of the host name or IP address. The host name and port number will be required to connect Vivado to the `xvcserver` application. See the OS help pages for information regarding the firewall port settings for your OS.
3. Move the source files to the directory of your choice. For example, use:

```
/home/username/xil_xvc or /usr/local/src/xil_xvc
```

4. Change to the directory containing the application source files:

```
# cd ./xvcserver/
```

5. Compile the application:

```
# make
```

6. Start the XVC-Server application:

```
# ./bin/xvc_PCIE -s TCP::10200
```

After the Vivado Design Suite has connected to the XVC-server application you should see the following message from the XVC-server.

```
Enable verbose by setting VERBOSE evn var.  
Opening /dev/xil_xvc/cfg_ioc0
```

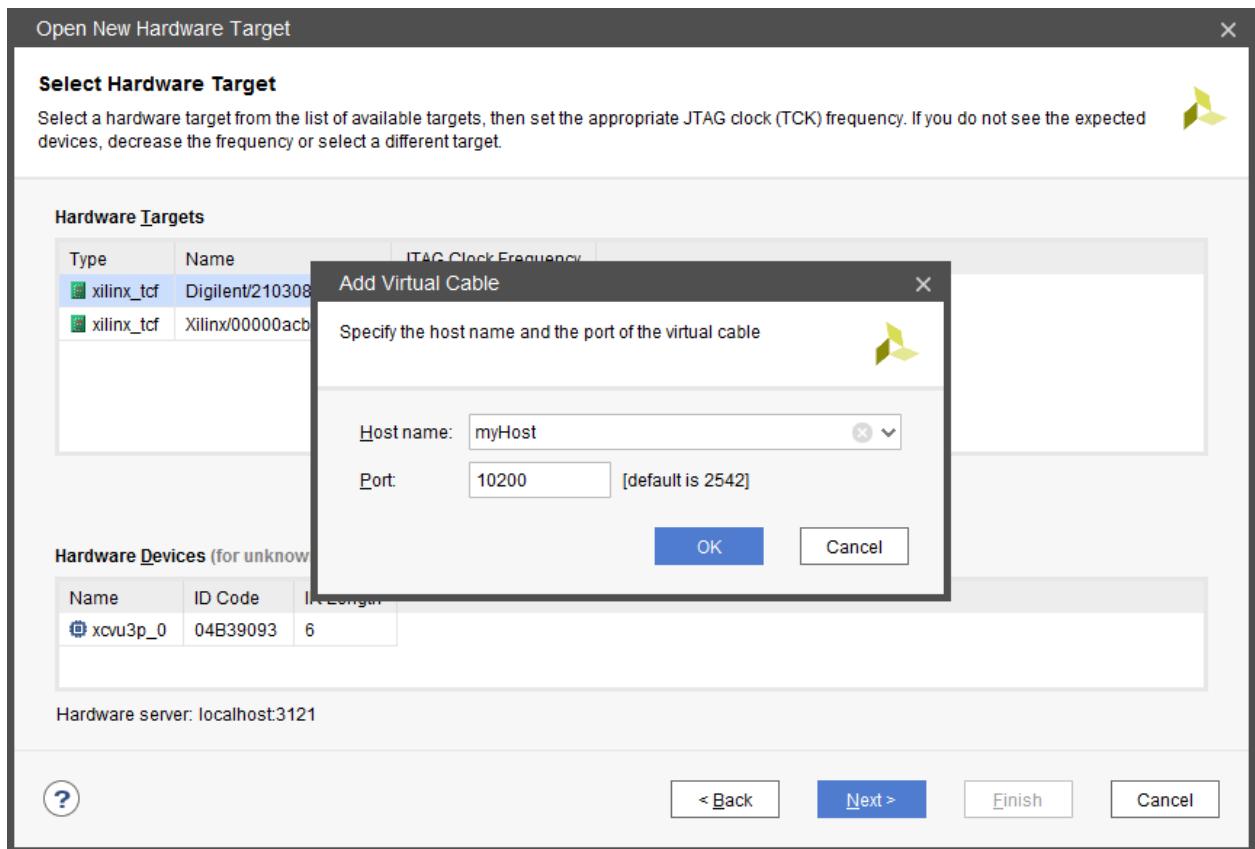
Connecting the Vivado Design Suite to the XVC-Server Application

The Vivado Design Suite can be run on the computer that is running the XVC-server application, or it can be run remotely on another computer that is connected over an Ethernet network. The port however must be accessible to the machine running Vivado. To connect Vivado to the XVC-Server application follow the steps shown using the default port number.

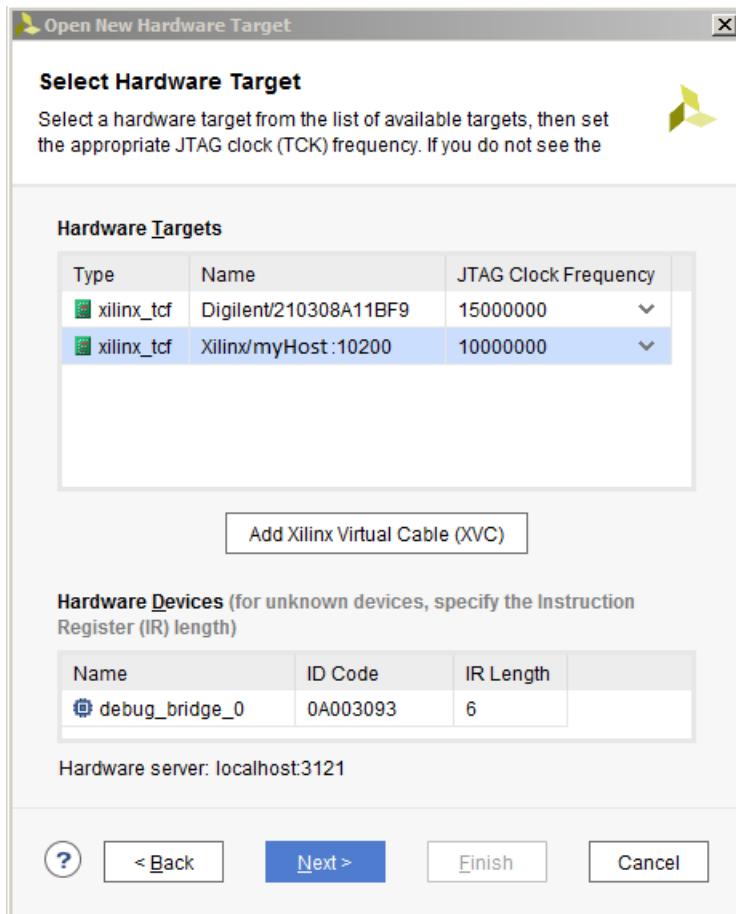
1. Launch the Vivado Design Suite.
2. Select **Open HW Manager**.
3. In the Hardware Manager, select **Open target** → **Open New Target**.
4. Click **Next**.
5. Select **Local server**, and click **Next**.

This launches `hw_server` on the local machine, which then connects to the `xvcserver` application.

6. Select **Add Xilinx Virtual Cable (XVC)**.
7. In the Add Virtual Cable dialog box, type in the appropriate Host name or IP address, and Port to connect to the `xvcserver` application. Click **OK**.

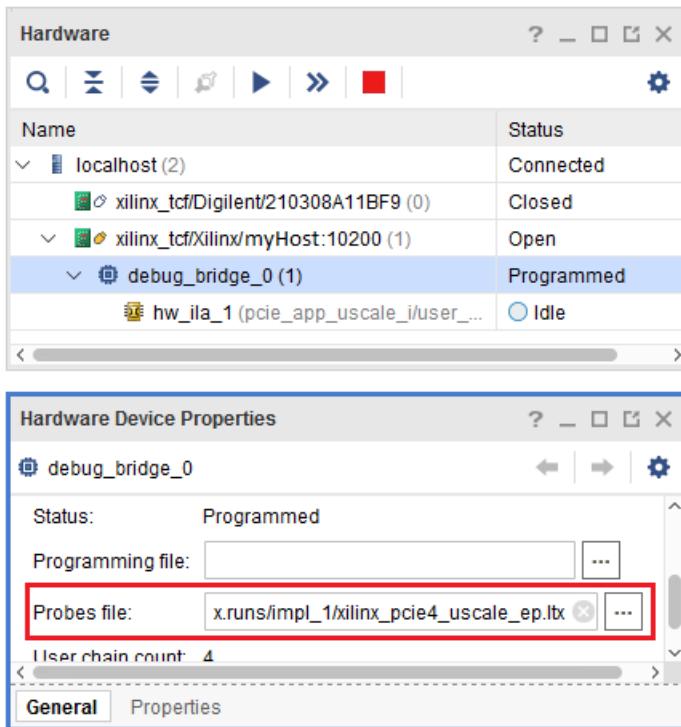


8. Select the newly added XVC target from the Hardware Targets table, and click **Next**.



9. Click **Finish**.

10. In the Hardware Device Properties panel, select the debug bridge target, and assign the appropriate probes .ltx file.



Vivado now recognizes your debug cores and debug signals, and you can debug your design through the Vivado hardware tools interface using the standard debug approach.

This allows you to debug Xilinx FPGA designs through the PCIe connection rather than JTAG using the Xilinx Virtual Cable technology. You can terminate the connection by closing the hardware server from Vivado using the right-click menu. If the PCIe connection is lost or the XVC-Server application stops running, the connection to the FPGA and associated debug cores will also be lost.

Run Time Considerations

The Vivado connection to an XVC-Server Application should not be running when a device is programmed. The XVC-Server Application along with the associated connection to Vivado should only be initiated after the device has been programmed and the hardware PCIe interface is active.

For DFX designs, it is important to terminate the connection during DFX operations. During a DFX operation where debug cores are present inside the dynamic region, a portion of the debug tree is expected to be reprogrammed. Vivado debug tools should not be actively communicating with the FPGA through XVC during a DFX operation.

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help**→**Documentation and Tutorials**.
- On Windows, select **Start**→**All Programs**→**Xilinx Design Tools**→**DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

Note: For more information on DocNav, see the [Documentation Navigator](#) page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. AMBA AXI4-Stream Protocol Specification ([ARM IHI 0051A](#))
2. PCI-SIG Documentation (www.pcisig.com/specifications)
3. UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide ([PG156](#))
4. AXI Bridge for PCI Express Gen3 Subsystem Product Guide ([PG194](#))
5. DMA/Bridge Subsystem for PCI Express Product Guide ([PG195](#))
6. Virtex-7 FPGA Integrated Block for PCI Express LogiCORE IP Product Guide ([PG023](#))
7. UltraScale Architecture Configuration User Guide ([UG570](#))
8. Kintex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics ([DS892](#))
9. Virtex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics ([DS893](#))
10. UltraScale Architecture PCB Design User Guide ([UG583](#))
11. UltraScale and UltraScale+ FPGAs Packaging and Pinouts Product Specification ([UG575](#))
12. UltraScale Architecture GTH Transceivers User Guide ([UG576](#))
13. UltraScale Architecture GTY Transceivers User Guide ([UG578](#))
14. Zynq UltraScale+ Device Packaging and Pinouts Product Specification User Guide ([UG1075](#))
15. Kintex UltraScale+ FPGAs Data Sheet: DC and AC Switching Characteristics ([DS922](#))
16. Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics ([DS923](#))
17. Vivado Design Suite User Guide: Designing with IP ([UG896](#))
18. Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator ([UG994](#))
19. Vivado Design Suite User Guide: Getting Started ([UG910](#))
20. Vivado Design Suite User Guide: Using Constraints ([UG903](#))
21. Vivado Design Suite User Guide: Logic Simulation ([UG900](#))
22. ISE to Vivado Design Suite Migration Guide ([UG911](#))
23. Vivado Design Suite User Guide: Programming and Debugging ([UG908](#))
24. Vivado Design Suite Tutorial: Dynamic Function eXchange ([UG947](#))
25. In-System IBERT LogiCORE IP Product Guide ([PG246](#))
26. ATX Power Supply Design Guide
27. PIPE Mode Simulation Using Integrated Endpoint PCI Express Block in Gen2 x8 and Gen3 x8 Configurations ([XAPP1184](#))

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
09/22/2020 Version 1.3	
Virtex UltraScale+ Devices Available GT Quads	Updated the Virtex UltraScale+ devices available GT quads.
07/22/2020 Version 1.3	
Unsupported PCI Express Base Specification 4.0 Features (PCIE4C)	Moved unsupported content from a table to list.
Completer Request Interface Operation (512-bits), Avoiding Head-of-Line Blocking for Posted Requests, and Avoiding Head-of-Line Blocking for Posted Requests	Clarifying editorial updates.
Tandem Configuration	Updated Partial Reconfiguration to Dynamic Function eXchange throughout. Verified and updated supported devices.
Virtex UltraScale+ Devices Available GT Quads, and Zynq UltraScale+ Devices Available GT Quads	Updated supported devices and available quads.
11/18/2019 Version 1.3	
Configuration Space	Added User Design Extended Configuration List table.
Tandem Configuration	Updated Supported Devices table. Added the Using Tandem PCIe on Zynq MPSoC Devices section.
Legacy Interrupt Mode	Updated the Legacy Interrupt Signaling table.
Virtex UltraScale+ Devices Available GT Quads	Updated GT quad tables for XCVU35P and XCVU37P devices.
06/24/2019 Version 1.3	
Unsupported Features	Added items for PCIe Secure IP model, which does not support simulation of DRP interface, and Tandem PROM simulation, which is not supported.
Tandem Configuration	Added and updated supported devices.
64/128/256-Bit Requester Interface and 512-Bit Requester Interface	Changed pcie_rq_tag[5:0] to pcie_rq_tag[7:0].
Enabling Loopback Master on Root Port	New section.
Integrated Block Endpoint Configuration Overview	Added clarifying information regarding the FLR for virtual functions, which is not fully implemented in the example design.
12/05/2018 Version 1.3	
IP Facts	Added PCIE4C integrated block support details.
Chapter 2: Overview	Added PCIE4C integrated block support details. Added details about PCI Express Base Spec 4.0 compliance limitations.

Section	Revision Summary
Chapter 3: Product Specification	<p>Added Gen4 link speed to the Minimum Device Requirements for PCIE4C.</p> <p>Added PCIEC and PCIE4 block locations to the Available Integrated Blocks for PCI Express - Virtex UltraScale+ table.</p> <p>Added pcie_rq_tag0[7:0] and pcie_rq_tag1[7:0], which replace pcie_rq_tag.</p> <p>Updated the cfg_ext_read_received description details.</p> <p>In the “Configuration Received Message Interface” section, removed ATS message types from the Message Type Encoding on Receive Message Interface table, and the Message Parameters on Receive Message Interface table.</p>
Chapter 5: Design Flow Steps	<p>Updated the First VF Offset parameter details (in the SRIOV Config tab).</p>
Appendix B: GT Locations	<p>Added PCIE4 and PCIE4C block locations for supported Virtex UltraScale+ devices with high bandwidth memory (HBM).</p>
06/06/2018 Version 1.3	
Chapter 2: Overview	<p>Added devices to Available Integrated Blocks for PCI Express - Virtex UltraScale+ table.</p>
Chapter 3: Product Specification	<p>Updated error_out to cfg_local_error_out in Configuration Status Interface Port Descriptions table.</p>
Chapter 4: Designing with the Core	<p>Updated the Tandem PROM/PCIe Supported Configurations table, and the Tandem PCIe with Field Updates section.</p> <p>Added more detail to Clocking section.</p> <p>Added important note to Lane Reversal section.</p>
Appendix B: GT Locations	<p>Added devices and packages to Virtex Ultrascale+ Devices Available GT Quads table, and Zynq UltraScale+ Devices Available GT Quads table.</p>
04/04/2018 Version 1.3	
General Updates	<p>Added Minimum Device Requirements for PCIE4C information.</p> <p>Updated device support in the Tandem PROM/PCIe Supported Configurations table in the Designing with the Core chapter.</p> <p>Added Integrated Debug Options section to Debugging appendix.</p>
12/20/2017 Version 1.3	
Chapter 3: Product Specification	<p>Removed Supported Devices table.</p> <p>Updated description for Target Function field in the Completer Request Descriptor Fields table.</p> <p>Updated description for Lower Address field in the Requester Completion Descriptor Fields table.</p> <p>Added details about user_clk signal to Clock and Reset Interface section.</p>
Chapter 4: Designing with the Core	<p>Added note regarding configuration bank 65 in the Tandem Configuration section.</p> <p>Updated the Legacy Interrupt Signaling figure and description.</p>
General Updates	<p>In Appendix A, “Upgrading”, added enable_auto_rxreq new parameter.</p> <p>In Appendix B, “GT Locations”, all new content.</p>

Section	Revision Summary
10/04/2017 Version 1.3	
Chapter 3: Product Specification	<p>Updated “Available Integrated Blocks for PCI Express - Virtex UltraScale+” table.</p> <p>Updated Sideband Signal Descriptions in m_axis_cq_tuser name/description for Bit Index 87:85.</p> <p>Added cfg_err_cor_out, cfg_err_nonfatal_out, cfg_err_fatal_out, and cfg.ds_port_number.</p> <p>Updated the description for sys_clk_gt.</p>
Chapter 4: Designing with the Core	<p>Updated the Tandem Configuration section.</p> <p>Updated the Clocking Architecture figure.</p> <p>Major updates to description for Completer Completion Descriptor Fields (Bit Indexes 7972, 87:80, and 88), and Requester Request Descriptor Fields (Bit Indexes 87:80, 95:88, and 120).</p> <p>Updated note in Handling of Completion Errors section.</p>
Chapter 5: Design Flow Steps	<p>Updated the following Vivado IP catalog option:</p> <ul style="list-style-type: none"> • Extended Tag Field (in the Capabilities tab). • PF0 ID Initial Values > Device ID value (in Identity Setting (PF0 IDs and PF1 IDs) tab). • MSIx Table Settings > Table Size (in MSI-X Capabilities tab). • General SRIOV Config (in SRIOV Config tab) <p>Added new GT DRP Clock Source (in GT Settings tab), and GT COMMON Option (in Shared Logic tab)</p>
General Updates	<p>In Appendix A, “Upgrading”, added new port and parameter details.</p> <p>In Appendix B, “GT Location”, updated the tables: Virtex UltraScale+ Devices GT Locations, Zynq UltraScale+ Devices GT Locations, Kintex UltraScale+ Devices Available GT Quads, and Zynq Ultrascale+ Devices Available GT Quads</p> <p>In Appendix C, “Debugging”, updated documented width for gt_dmonitorout port.</p> <p>In Appendix D, “Using Xilinx Virtual Cable to Debug”:</p> <ul style="list-style-type: none"> • Added XVC Status Register • Updated the Debug Bridge for XVC-PCIe-VSEC Register Map, and Debug Bridge for AXI-XVC Register Map tables.

Section	Revision Summary
06/07/2017 Version 1.2	
General Updates	<p>Updated the Available Integrated Blocks for PCI Express - Zynq UltraScale+ table.</p> <p>Updated port description for cfg_interrupt_msi_function_number.</p> <p>Updated the TUSER signal in all AXI4-Stream Interface diagrams.</p> <p>Updated the TUSER signal definition for the 512-bit AXI Interface including Straddle operation.</p> <p>Minor updates to the Tandem Configuration section.</p> <p>Updated pcie_cq_np_req to pcie_cq_np_req[0].</p> <p>Added missing AXISTEN_IF_ENABLE_MSG_ROUTE Attribute Bit Descriptions table.</p> <p>Updated the Zynq UltraScale+ Device GT Locations table.</p> <p>Clarified the XCV driver and software example location in the Using Xilinx Virtual Cable to Debug appendix.</p>
04/05/2017 Version 1.2	
General Updates	<p>Added the new Using Xilinx Virtual Cable to Debug appendix.</p> <p>Updated the Tandem Configuration section.</p> <p>Updated new Ports and Parameters information in the Upgrading appendix.</p>
11/30/2016 Version 1.1	
Chapter 5: Design Flow Steps	<p>Updated the SRIOV BAR Size Ranges for Device Configuration table.</p> <p>Added the PLL Selection and Link Partner TX Preset options in the GT Settings tab.</p> <p>Clarified that the Enable In System IBERT, and Enable JTAG Debugger options should be used only for hardware debugging. Simulations are not supported for the cores generated using these options.</p>
Appendix B: GT Locations	Added the Available GT Quads section.
10/05/2016 Version 1.1	
General Updates	<p>Moved the performance and resource utilization data to the web.</p> <p>Updated the Minimum Device Requirements table.</p> <p>Added tandem configuration support.</p> <p>Added IBERT ports, GT DRP ports and PCIe ports used for transceiver debug, and the Vivado Design Suite core customization options that support them.</p> <p>Added to Port Changes table in the Migrating and Upgrading appendix.</p> <p>Updated the GT Locations table, and the Available Integrated Blocks for PCI Express table for Virtex UltraScale+ devices.</p>
06/08/2016 Version 1.1	
General Updates	<p>Completer Model option added for Root Port Model test bench.</p> <p>MXI-X Interrupt Internal (built-in) support added.</p> <p>GT Setting for insertion loss adjustment added.</p> <p>QPLL1 support added in Gen2 (5Gb/s) mode.</p>

Section	Revision Summary
04/06/2016 Version 1.1	
Initial release.	N/A

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright

© Copyright 2016–2020 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries. All other trademarks are the property of their respective owners.