UltraScale+ Devices Integrated Block for PCI Express v1.2

Product Guide

Vivado Design Suite

PG213 June 7, 2017

XILINX
ALL PROGRAMMABLE™

XILINX
ALL PROGRAMMABLE™

Table of Contents

Introduction

The Xilinx® UltraScale+ Devices Integrated Block for PCIe® solution IP core is a high-bandwidth, scalable, and reliable serial interconnect building block solution for use with UltraScale+™ devices. The core supports 1-lane, 2-lane, 4-lane, 8-lane, and 16-lane Endpoint 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* [Ref 2]. This solution supports the AXI4-Stream interface for the customer user interface.

Features

- Designed to *PCI Express Base Specification 3.1* [Ref 2].
- PCI Express Endpoint, Legacy Endpoint or Root Port Modes.
- x1, x2, x4, x8 or x16 link widths.
- Gen1, Gen2 and Gen3 link speeds.
- AXI4 Streaming Interface to customer logic.
- Parity protection on internal logic data paths and data interfaces.
- Advanced Error Reporting (AER) and End-to-End CRC (ECRC).
- Block RAM used for Transaction buffering.
- One PCI Express Virtual Channel, eight Traffic Classes.
- Up to 4 Physical Functions and 252 Virtual Functions.
- Built-in lane reversal and receiver lane-lane de-skew.
- 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.

LogiCORE™ IP Facts Table	
Core Specifics	
Supported Device Family[1]	UltraScale+
Supported User Interfaces	AXI4-Stream
Resources	Performance and Resource Utilization web page
Provided with Core	
Design Files	Verilog
Example Design	Verilog
Test Bench	Verilog
Constraints File	XDC
Simulation Model	Verilog
Supported S/W Driver	N/A
Tested Design Flows[2]	
Design Entry	Vivado® Design Suite
Simulation	For supported simulators, see the Xilinx Design Tools: Release Notes Guide
Synthesis	Vivado synthesis
Support	
Provided by Xilinx at the Xilinx Support web page	

Notes:
1. For a complete list of supported devices, see the Vivado IP catalog.
2. For the supported versions of the tools, see the Xilinx Design Tools: Release Notes Guide.

Overview

The UltraScale+ Devices 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: *If you want to implement a design in UltraScale devices, see the UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide (PG156) [Ref 3].*

Figure 1-1 shows the interfaces for the core.

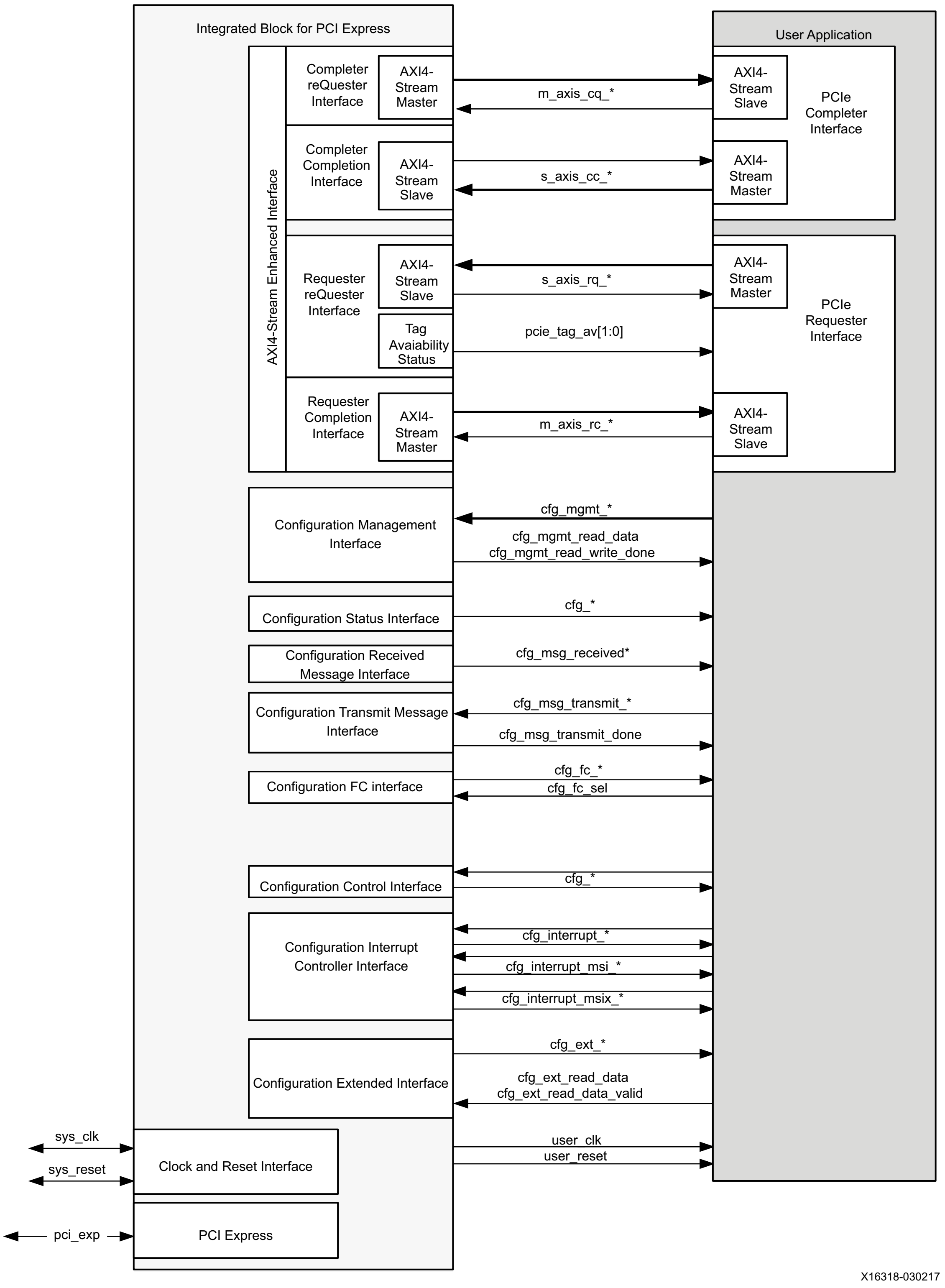


Figure 1-1: **Core Interfaces**

Feature Summary

The GTH and GTY transceivers in the Integrated Block for PCI Express (PCIe®) 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. 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:

- Designed to *PCI Express Base Specification 3.1* [Ref 2]
- PCI Express Endpoint, Legacy Endpoint or Root Port Modes
- x1, x2, x4, x8 or x16 link widths
- Gen1, Gen2 and Gen3 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)
- Block RAM used for Transaction buffering
  - 16 KB - Replay Buffer
  - Configurable 4 KB or 16 KB - Received Posted Transaction FIFO
  - Configurable 8KB or 16 KB or 32 KB - Received Completion Transaction FIFO
- 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-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 32KB 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 (MCAP) and 100ms power on to configuration (Support in the IP is planned for future release)
  - Debug and Diagnostics Interface

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

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

- Does not implement the Address Translation Service but allows its implementation in external soft logic
- Switch ports
- Resizable BAR extended capability

Licensing and Ordering Information

The UltraScale+ Devices Integrated Block for PCIe 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 PCIe solution is compatible with industry-standard application form factors such as the PCI Express® Card Electromechanical (CEM) v3.0 and the PCI™ Industrial Computer Manufacturers Group (PICMG) 3.4 specifications [Ref 2].

Resource Utilization

For full details about performance and resource utilization, visit the Performance and Resource Utilization web page.

Available Integrated Blocks for PCI Express

Table 2-1 lists the supported devices. Table 2-2, Table 2-3, and Table 2-4 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 2-1: **Supported Devices**

Device Selection		GTH	GTY	PCIe
FFVC1760	XCZU17EG	32	16	4
	XCZU19EG	32	16	5
FFVE1924	XCZU17EG	44		4
	XCZU19EG	44		5
FFVB1517	XCZU19EG	16		5
FFVE1517	XCKU11P	32	20	4
	XCKU15P	32	24	5
FFVC1517	XCVU3P		40	2

Table 2-1: **Supported Devices *(Cont'd)***

Device Selection		GTH	GTY	PCIe
FLVA2104	XCVU5P		52	4
	XCVU7P		52	4
	XCVU9P		52	6
FLVB2104	XCVU5P		76	4
	XCVU7P		76	4
	XCVU9P		76	6
FLVC2104	XCVU5P		80	4
	XCVU7P		80	4
	XCVU9P		104	6
FLVA2577	XVCU9P		120	6

Table 2-2: **Available Integrated Blocks for PCI Express - Virtex UltraScale+**

Device Selection		PCI Express Block Location							
Device	**Package**	**X0Y0**	**X0Y1**	**X0Y2**	**X0Y3**	**X0Y5**	**X1Y0**	**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
	FSGD2104		Yes		Yes	Yes		Yes	Yes
XCVU11P	FLGA2577	Yes	Yes	Yes					
	FLGB2104	Yes	Yes	Yes					
	FLGC2104	Yes	Yes	Yes					
	FLGF1924	Yes	Yes	Yes					
	FSGD2104	Yes	Yes	Yes					

Table 2-2: **Available Integrated Blocks for PCI Express - Virtex UltraScale+ *(Cont'd)***

Device Selection		PCI Express Block Location							
Device	**Package**	**X0Y0**	**X0Y1**	**X0Y2**	**X0Y3**	**X0Y5**	**X1Y0**	**X1Y2**	**X1Y4**
XCVU13P	FHGA2104		Yes	Yes					
	FHGB2014		Yes	Yes	Yes				
	FHGC2104	Yes	Yes	Yes	Yes				
	FLGA2577	Yes	Yes	Yes	Yes				
	FIGD2104	Yes	Yes	Yes	Yes				

Table 2-3: **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
XCZU19EG	FFVC1760			Yes	Yes	Yes	Yes	Yes
	FFVE1924					Yes	Yes	Yes
	FFVB1517					Yes	Yes	Yes
	FFVD1760			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					

Table 2-3: **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**
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						
XCZU28DR	FFVE1156	Yes	Yes					
	FFVG1517	Yes	Yes					
XCZU29DR	FFVF1760	Yes	Yes					
XCZU27DR	FFVE1156	Yes	Yes					
	FFVG1517	Yes	Yes					

Table 2-4: **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					

Table 2-4: **Available Integrated Blocks for PCI Express - Kintex UltraScale+ *(Cont'd)***

Device Selection		PCI Express block location					
Device	**Package**	**X0Y0**	**X0Y2**	**X0Y3**	**X1Y0**	**X1Y1**	**X1Y2**
XCKU5P	FFVA676	Yes					
	FFVB676	Yes					
	FFVD900	Yes					
	SFVB784	Yes					

Table 2-5: **Minimum Device Requirements**

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

Notes:

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

GT Locations

The recommended GT locations for the available device part and package combinations are available in Appendix B, GT Locations. The package pins are derived directly from the GT X-Y locations listed in the appendix. The Vivado Design Suite provides an XDC for the selected part and package that matches the contents of the tables.

For the recommended list of GT locations, see:

- Virtex UltraScale+ Device GT Locations
- Kintex UltraScale+ Device GT Locations
- Zynq UltraScale+ Device GT Locations

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.

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. Table 2-6 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 2-6: **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.
m_axis_cq_tuser	Output	88	CQ User Data. This set of signals contains sideband information for the TLP being transferred. These signals are valid when m_axis_cq_tvalid is High. Table 2-7, page 17 describes the individual signals in this set.
m_axis_cq_tlast	Output	1	TLAST indication for CQ Data. 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.
m_axis_cq_tkeep	Output	DW/32	TKEEP indication for CQ Data. The assertion of bit *i* of this bus during a transfer indicates to the user application that Dword *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 Dwords). This is true for both Dword-aligned and address-aligned modes of payload transfer. 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.

Table 2-6: **Completer Request Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
m_axis_cq_tvalid	Output	1	CQ Data Valid. 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	Input	1	CQ Data Ready. 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. 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.
pcie_cq_np_req	2	Input	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. 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: 00: No change 01: Increment by 1 10 or 11: Reserved (bit [1] only applicable in 512-bit interface) 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. 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. The setting of pcie_cq_np_req[1:0] does not need to be aligned with the packet transfers on the completer request interface.
pcie_cq_np_req_count	6	Output	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. 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. This count resets on user_reset and de-assertion of user_lnk_up.

Table 2-7: **Sideband Signal Descriptions in m_axis_cq_tuser**

Bit Index	Name	Width	Description
3:0	first_be[3:0]	4	Byte enables for the first Dword of the payload. 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. This field is valid in the first beat of a packet, that is, when `sop` and m_axis_cq_tvalid are both High.
7:4	last_be[3:0]	4	Byte enables for the last Dword. 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. This field is valid in the first beat of a packet, that is, when sop and m_axis_cq_tvalid are both High.
39:8	byte_en[31:0]	32	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 m_axis_cq_tdata bus contains a valid payload byte. This bit is not asserted for descriptor bytes. 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. 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. 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. 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.
40	sop	1	Start of packet. 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.
41	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 user 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 a cycle when m_axis_cq_tlast is High. 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).

Table 2-7: **Sideband Signal Descriptions in m_axis_cq_tuser *(Cont'd)***

Bit Index	Name	Width	Description
42	tph_present	1	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.
44:43	tph_type[1:0]	2	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.
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	Odd parity for the 256-bit transmit data. 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	is_eof_ptr[2:0]	3	Reserved. Only valid 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.Table 2-8 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 2-8: **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. Table 2-9, page 19 describes 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 2-8: **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 2-9: **Sideband Signal Descriptions in s_axis_cc_tuser**

Bit Index	Name	Width	Description
0	discontinue	1	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. 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. 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. 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.

Table 2-9: **Sideband Signal Descriptions in s_axis_cc_tuser *(Cont'd)***

Bit Index	Name	Width	Description
32:1	parity	32	Odd parity for the 256-bit data. 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. 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.* The parity bits can be permanently tied to 0 if parity check is not enabled in the core.

Requester Request Interface

The Requester Request (RQ) interface consists of the ports through which the user application generates requests to remote PCIe® devices. Table 2-10 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 2-10: **Requester Request Interface Port Descriptions**

Port	Direction	Width	Description
s_axis_rq_tdata	Input	DW	Requester reQuest Data bus. 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.
s_axis_rq_tuser	Input	62	Requester reQuest User Data. This set of signals contains sideband information for the TLP being transferred. These signals are valid when s_axis_rq_tvalid is High. Table 2-11, page 22 describes the individual signals in this set.
s_axis_rq_tlast	Input	1	TLAST Indication for Requester reQuest Data. 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.

Table 2-10: **Requester Request Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
s_axis_rq_tkeep	Input	DW/32	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_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 (in both 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_rq_tvalid	Input	1	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.
s_axis_rq_tready	Output	4	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. 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. You can assign all 4 bits to 1 or 0.
pcie_rq_seq_num0	Output	6	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.
pcie_rq_seq_num_vld0	Output	1	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].

Table 2-10: **Requester Request Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
pcie_rq_tag0	Output	8	Requester reQuest Non-Posted tag. When tag management for Non-Posted requests is performed by the core, this output is used by the core to communicate the allocated tag for each Non-Posted request received. 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. 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.
pcie_rq_tag_vld0	Output	1	Requester reQuest Non-Posted tag valid. 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.

Table 2-11: **Sideband Signal Descriptions in s_axis_rq_tuser**

Bit Index	Name	Width	Description
3:0	first_be[3:0]	4	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. 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. 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.
7:4	last_be[3:0]	4	Byte enables for the last Dword. 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 Writes of two Dwords or more, these bits indicate the valid bytes in the last Dword of the payload. 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.

Table 2-11: **Sideband Signal Descriptions in s_axis_rq_tuser *(Cont'd)***

Bit Index	Name	Width	Description
10:8	addr_offset[2:0]	3	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. 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. When the requester request interface is configured in the Dword-alignment mode, this field must always be set to 0. In Root Port configuration, Configuration Packets must always be aligned to DW0, and therefore for this type of packets, this field must be set to 0 in both alignment modes.
11	discontinue	1	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. 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. 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, when asserted, it should not be deasserted until s_axis_rq_tready is High. 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. 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).
12	tph_present	1	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. This bit must be permanently tied to 0 if the TPH capability is not in use.
14:13	tph_type[1:0]	2	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. These bits can be set to any value if tph_present is set to 0.

Table 2-11: **Sideband Signal Descriptions in s_axis_rq_tuser *(Cont'd)***

Bit Index	Name	Width	Description
15	tph_indirect_tag_en	1	When this bit is set, the core uses the lower bits of tph_st_tag[7:0] as an index into its Steering Tag Table, and inserts the tag from this location in the transmitted request TLP. When this bit is 0, the core uses the value on tph_st_tag[7:0] directly as the Steering Tag. 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. This bit can be set to any value if tph_present is set to 0.
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, *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. Table 2-12 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 2-12: **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.
m_axis_rc_tuser	Output	75	Requester Completion User Data. This set of signals contains sideband information for the TLP being transferred. These signals are valid when m_axis_rc_tvalid is High. Table 2-13, page 26 describes the individual signals in this set.
m_axis_rc_tlast	Output	1	TLAST indication for Requester Completion Data. 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.
m_axis_rc_tkeep	Output	DW/32	TKEEP indication for Requester Completion Data. 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. 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. 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.
m_axis_rc_tvalid	Output	1	Requester Completion Data Valid. 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	Input	1	Requester Completion Data Ready. 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. 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.

Table 2-13: **Sideband Signal Descriptions in m_axis_rc_tuser**

Bit Index	Name	Width	Description
31:0	byte_en	32	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 m_axis_rc_tdata bus contains a valid payload byte. This bit is not asserted for descriptor bytes. 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. 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.
32	is_sof_0	1	Start of a first Completion TLP. For 64-bit and 128-bit interfaces, and for the 256-bit interface with no straddling, is_sof_0 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 is_sof_1 is permanently set to 0. Use of this signal is optional when the straddle option is not enabled. 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 is_sof_0 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.
33	is_sof_1	1	Start of a second Completion TLP. 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. The assertion of is_sof_1 indicates a second Completion TLP starting in the beat, with its first bye 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 is_eof_0[0] is also set in the same beat.
37:34	is_eof_0[3:0]	4	End of a first Completion TLP and the offset of its last Dword. These outputs are used only when the interface width is 256 bits and the straddle option is enabled. The assertion of the bit is_eof_0[0] indicates the end of a first Completion TLP in the current beat. When this bit is set, the bits is_eof_0[3:1] provide the offset of the last Dword of this TLP.

Table 2-13: **Sideband Signal Descriptions in m_axis_rc_tuser *(Cont'd)***

Bit Index	Name	Width	Description
41:38	is_eof_1[3:0]	4	End of a second Completion TLP and the offset of its last Dword. 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. 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). 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]. If is_eof_1[0] is High, the signals is_eof_0[0] and is_sof_1 are also High in the same beat.
42	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 user 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_eof_0[0] is High. When the straddle option is enabled, the core does not start a second 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 to which it is attached, using Advanced Error Reporting (AER).
74:43	parity	32	Odd parity for the 256-bit transmit data. 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.

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.

Completer Request Interface

Table 2-14: **Completer Request Interface Port Descriptions**

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 Table 2-15.
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	The assertion of bit *i* of this bus during a transfer indicates to the user logic that Dword *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. 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.
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 2-14: **Completer Request Interface Port Descriptions *(Cont'd)***

Name	Width	Direction	Description
pcie_cq_np_req	2	Input	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. 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: 00: No change 01: Increment by 1 10 or 11: Increment by 2 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. 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. The setting of pcie_cq_np_req[1:0] does not need to be aligned with the packet transfers on the completer request interface.
pcie_cq_np_req_count	6	Output	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. 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. This count resets on user_reset and de-assertion of user_lnk_up.

Table 2-15: **Sideband Signals in m_axis_cq_tuser**

Bit Index	Name	Width	Description
7:0	first_be[7:0]	8	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. 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. 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.
15:8	last_be[7:0]	8	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 Atomic Operations and Messages with a payload, these bits are set to all 1s. 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. 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.
79:16	byte_en[63:0]	64	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 to the user logic that byte *i* of the m_axis_cq_tdata bus contains a valid payload byte. This bit is not asserted for descriptor bytes. 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. 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. 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-DW payload is being transferred.

Table 2-15: **Sideband Signals in m_axis_cq_tuser *(Cont'd)***

Bit Index	Name	Width	Description
81:80	is_sop[1:0]	2	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: 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: Forbidden. 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.
83:82	is_sop0_ptr[1:0]	2	Indicates the position of the first byte of the first TLP starting in this beat: 00: Byte lane 0 10: Byte lane 32 01, 11: Reserved 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 bye lane 0.
85:84	is_sop1_ptr[1:0]	2	Indicates the position of the first byte of the second TLP starting in this beat: 10: Byte lane 32 00, 01, 11: Reserved. 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.
87:86	is_eop[1:0]	2	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: 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: Forbidden. 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.
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 2-15: **Sideband Signals in m_axis_cq_tuser *(Cont'd)***

Bit Index	Name	Width	Description
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 user 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 second 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).
98:97	tph_present[1:0]	2	These bits indicate the presence of a Transaction Processing Hint (TPH) in the request TLP being delivered across the interface. • tph_present[0] indicates the presence of a Hint in the first TLP of this beat. • tph_present[1] indicates the presence of a Hint in the second TLP of this beat. tph_present[0] is valid when m_axis_cq_tvalid and is_sop[0] are both High. tph_present[1] is valid when m_axis_cq_tvalid and is_sop[1] are both High.
102:99	tph_type[3:0]	4	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. • tph_type[1:0] provides the TPH type associated with the first TLP of this beat. • tph_type[3:2] provides the TPH type associated with the second TLP of this beat. tph_present[0] is valid when m_axis_cq_tvalid and is_sop[0] are both High. tph_present[1] is valid when m_axis_cq_tvalid and is_sop[1] are both High.
118:103	tph_st_tag[15:0]	8	When a TPH is present in the request TLP, this output provides the 8-bit Steering Tag associated with the hint. • tph_st_tag[7:0] provides the Steering Tag associated with the first TLP of this beat. • tph_st_tag[15:8] provides the Steering Tag associated with the second 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_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* provides the odd parity computed for byte *i* of m_axis_cq_tdata.

Completer Completion Interface

Table 2-16: **Completer Completion Interface Port Descriptions**

Name	Width	Direction	Description
s_axis_cc_tdata	512	Input	Completion data from the user application to the PCIe core.
s_axis_cc_tuser	81	Input	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. The individual signals in this set are described in Table 2-17.
s_axis_cc_tlast	1	Input	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. 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.
s_axis_cc_tkeep	16	Input	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. 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_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 128b address-aligned modes of payload transfer. The tkeep 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.
s_axis_cc_tvalid	1	Input	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	1	Output	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. 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.

Table 2-17: **Sideband Signals in s_axis_cc_tuser**

Bit Index	Name	Width	Description
1:0	is_sop[1:0]	2	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: 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: Forbidden. 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.
3:2	is_sop0_ptr[1:0]	2	Indicates the position of the first byte of the first TLP starting in this beat: 00: Byte lane 0 10: Byte lane 32 01, 11: Reserved 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.
5:4	is_sop1_ptr[1:0]	2	Indicates the position of the first byte of the second TLP starting in this beat: 10: Byte lane 32 00, 01, 11: Reserved. 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.
7:6	is_eop[1:0]	2	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: 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: Forbidden. 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.
11:8	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. This field is used by the core only when the straddle option is enabled.

Table 2-17: **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	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. 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. 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. 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. 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).
80:17	parity	64	Odd parity for the 256-bit data. 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. On detection of a parity error, the core nullifies the corresponding TLP on the link and reports it as an Uncorrectable Internal Error. The parity bits can be permanently tied to 0 if parity check is not enabled in the core.

Requester Request Interface

Table 2-18: **Requester Request Interface Port Descriptions**

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 Table 2-19.

Table 2-18: **Requester Request Interface Port Descriptions *(Cont'd)***

Name	Width	Direction	Description
s_axis_rq_tlast	1	Input	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. 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.
s_axis_rq_tkeep	16	Input	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 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 1's 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. 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.
s_axis_rq_tvalid	1	Input	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.
s_axis_rq_tready	1	Output	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. 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.
pcie_rq_tag_vld	2	Output	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_tag output. The two bits are encoded as follows: 00: no tags being provided in this cycle. 01: A tag is being presented on pcie_rq_tag[7:0]. 11: Tags are being presented simultaneously on pcie_rq_tag[7:0] and pcie_rq_tag[15:8]. The tag on pcie_rq_tag[7:0] corresponds to the earlier of the two requests transferred over the interface. 10: Reserved.

Table 2-18: **Requester Request Interface Port Descriptions *(Cont'd)***

Name	Width	Direction	Description
pcie_rq_tag	16	Output	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_tag[7:0] is valid for one cycle when pcie_rq_tag_vld[0] is High; and the tag value on pcie_rq_tag[15:8] is valid for one cycle when pcie_rq_tag_vld[1] is High. The client must copy this tag and use it to associate the completion data with the pending request. 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 client may, 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. When pcie_rq_tag[7:0] and pcie_rq_tag[15:8] are both valid in the same cycle, the value on pcie_rq_tag[7:0] corresponds to the earlier of the two requests transferred over the interface.
pcie_rq_seq_num0	6	Output	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. Data on the pcie_rq_seq_num0[5:0] output is valid when pcie_rq_seq_num_ vld0 is High.
pcie_rq_seq_num1	6	Output	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. Data on the pcie_rq_seq_num1[5:0] output is valid when pcie_rq_seq_num_ vld1 is High.
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 2-19: **Sideband Signals in s_axis_rq_tuser**

Bit Index	Name	Width	Description
7:0	first_be[7:0]	8	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). 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. 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.
15:8	last_be[7:0]	8	Byte enables for the last Dword. 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). For Memory Reads of 1 Dword, these 4 bits should be driven to 0H. For Memory Writes of 1 Dword, these 4 bits should be driven to 0H. 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. 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.
19:16	addr_offset[3:0]	4	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. 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. 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. When the requester request interface is configured in the Dword-alignment mode, these bits must always be set to 0.

Table 2-19: **Sideband Signals in s_axis_rq_tuser *(Cont'd)***

Bit Index	Name	Width	Description
21:20	is_sop[1:0]	2	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: 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: Forbidden. 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.
23:22	is_sop0_ptr[1:0]	2	Indicates the position of the first byte of the first TLP starting in this beat: 00: Byte lane 0 10: Byte lane 32 01, 11: Reserved
25:24	is_sop1_ptr[1:0]	2	Indicates the position of the first byte of the second TLP starting in this beat: 10: Byte lane 32 00, 01, 11: Reserved. This output is used only when the straddle option is enabled on the interface.
27:26	is_eop[1:0]	2	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: 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: Forbidden. 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.
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 2-19: **Sideband Signals in s_axis_rq_tuser *(Cont'd)***

Bit Index	Name	Width	Description
36	discontinue	1	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. 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. 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. 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. 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).
38:37	tph_present[1:0]	2	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. • 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). These inputs must be permanently tied to 0 if the TPH capability is not in use.
42:39	tph_type[3:0]	4	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. • 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). These bits can be set to any value if the corresponding tph_present bit is set to 0.

Table 2-19: **Sideband Signals in s_axis_rq_tuser *(Cont'd)***

Bit Index	Name	Width	Description
44:43	tph_indirect_tag_en[1:0]	2	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. • 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). 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. These inputs can be set to any value if the corresponding tph_present bit is set to 0.
60:45	tph_st_tag[15:0]	16	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. • 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). These inputs can be set to any value if the corresponding tph_present bit is set to 0.
66:61	seq_num0[5:0]	6	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. 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 logic is not monitoring the pcie_rq_seq_num* outputs of the core.
72:67	seq_num1[5:0]	6	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. 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 logic is not monitoring the pcie_rq_seq_num* outputs of the core.
136:73	parity	64	Odd parity for the 512-bit data. 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_rq_tdata. On detection of a parity error, the core nullifies the corresponding TLP on the link and reports it as an Uncorrectable Internal Error. These bits can be set to 0 if parity checking is disabled in the core.

Requester Completion Interface

Table 2-20: **Requester Completion Interface Port Descriptions**

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 Table 2-21.
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	The assertion of bit *i* of this bus during a transfer indicates to the user logic 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 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. 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.
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	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. 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.

Table 2-21: **Sideband Signals in m_axis_rc_tuser**

Bit Index	Name	Width	Description
63:0	byte_en	64	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* of this bus during a transfer indicates to the client that byte *i* of the m_axis_cq_tdata bus contains a valid payload byte. This bit is not asserted for descriptor bytes. 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.
67:64	is_sop[3:0]	4	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: 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. 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.
69:68	is_sop0_ptr[1:0]	2	Indicates the position of the first byte of the first TLP starting in this beat: 00: Byte lane 0 01: Byte lane 16 10: Byte lane 32 11: Byte lane 48 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 bye lane 0.
71:70	is_sop1_ptr[1:0]	2	Indicates the position of the first byte of the second TLP starting in this beat: 00: Reserved 01: Byte lane 16 10: Byte lane 32 11: Byte lane 48 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.

Table 2-21: **Sideband Signals in m_axis_rc_tuser *(Cont'd)***

Bit Index	Name	Width	Description
73:72	is_sop2_ptr[1:0]	2	Indicates the position of the first byte of the third TLP starting in this beat: 00: Reserved 01: Reserved 10: Byte lane 32 11: Byte lane 48 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.
75:74	is_sop3_ptr[1:0]	2	Indicates the position of the first byte of the fourth TLP starting in this beat: 00, 01, 10: Reserved 11: Byte lane 48 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.
79:76	is_eop[3:0]	4	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: 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. When the straddle option is disabled, m_axis_rc_tlast indicates the final beat of a TLP.
83:80	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. 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.
87:84	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 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.
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.

Table 2-21: **Sideband Signals in m_axis_rc_tuser *(Cont'd)***

Bit Index	Name	Width	Description
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* provides the odd parity computed for byte *i* of m_axis_cq_tdata.

Other Core Interfaces

The core also provides the interfaces described in this section.

Power Management Interface

Table 2-22 defines the ports in the Power Management interface of the core.

Table 2-22: **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. Table 2-23 defines the ports in the Configuration Management interface of the core.

Table 2-23: **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 PCI™ Config 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 Config 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. Table 2-24 defines the ports in the Configuration Status interface of the core.

Table 2-24: **Configuration Status Interface Port Descriptions**

Port	Direction	Width	Description
cfg_phy_link_down	Output	1	Configuration Link Down Status of the PCI Express link based on the Physical Layer LTSSM. • 1b: Link is Down (LinkUp state variable is 0b) • 0b: Link is Up (LinkUp state variable is `1b`) ***Note:*** Per the *PCI Express Base Specification, rev. 3.0* [Ref 2], 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. ***Note:*** While reset is asserted, the output of this signal are `0b` until reset is released.
cfg_phy_link_status	Output	2	Configuration Link Status Status of the PCI Express link. • 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	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). Negotiated Link Width values: • 000b = x1 • 001b = x2 • 010b = x4 • 011b = x8 • 100b = x16 Other values are reserved.
cfg_current_speed	Output	2	Current Link Speed This signal outputs the current link speed of the given PCI Express Link. • 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
cfg_max_payload	Output	2	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. • 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 2-24: **Configuration Status Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_max_read_req	Output	3	Max_Read_Request_Size 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. • 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
cfg_function_status	Output	16	Configuration Function Status 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: • 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	Configuration Virtual Function Status These outputs indicate the status of virtual functions, two bits each per virtual function. • 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 2-24: **Configuration Status Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_function_power_state	Output	12	Configuration Function Power State 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: • 000: D0_uninitialized • 001: D0_active • 010: D1 • 100: D3_hot
cfg_vf_power_state	Output	756	Configuration Virtual Function Power State 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: • 000: D0_uninitialized • 001: D0_active • 010: D1 • 100: D3_hot
cfg_link_power_state	Output	2	Current power state of the PCI Express link, and is valid when cfg_phy_link_status[1:0] == 11b (DL Initialization is complete). • 00: L0 • 01: TX L0s • 10: L1 • 11: L2/3 Ready

Table 2-24: **Configuration Status Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_local_error_out	Output	5	Local Error Conditions: Error priority is noted and Priority 0 has the highest priority. 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	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.

cfg_rx_pm_state	Output	2	Current Rx Active State Power Management L0s State: Encoding is listed below and valid when cfg_ltssm_state is indicating L0:
			RX_NOT_IN_L0s = 0
			RX_L0s_ENTRY = 1
			RX_L0s_IDLE = 2
			RX_L0s_FTS = 3

Table 2-24: **Configuration Status Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description

cfg_tx_pm_state	Output	2	Current TX Active State Power Management L0s State: Encoding is listed below and valid when cfg_ltssm_state is indicating L0:
			TX_NOT_IN_L0s = 0
			TX_L0s_ENTRY = 1
			TX_L0s_IDLE = 2
			TX_L0s_FTS = 3

cfg_ltssm_state	Output	6	LTSSM State. Shows the current LTSSM state: 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

Table 2-24: **Configuration Status Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_rcb_status	Output	4	RCB Status. 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. For each bit, a value of 0 indicates an RCB of 64 bytes and a value of 1 indicates 128 bytes.
cfg_dpa_substate_change	Output	4	Dynamic Power Allocation Substate Change. 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.
cfg_obff_enable	Output	2	Optimized Buffer Flush Fill Enable. This output reflects the setting of the OBFF Enable field in the Device Control 2 register. • 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	This output is used by the core in Root Port mode to signal one of the following link training-related events: (a) 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 (b) 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 (c) 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. The pl_interrupt output is not active when the core is configured as an Endpoint.

Table 2-24: **Configuration Status Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_tph_requester_enable	Output	4	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. And so on for other physical functions.
cfg_tph_st_mode	Output	12	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. And so on for other physical functions.
cfg_vf_tph_requester_enable	Output	252	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.
cfg_vf_tph_st_mode	Output	756	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.
pcie_tfc_nph_av	Output	4	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: 0000: No credit available 0001: 1 credit available 0010: 2 credits available … 1110: 14 credits available 1111: 15 or more credits available. 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.

Table 2-24: **Configuration Status Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
pcie_tfc_npd_av	Output	4	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: 0000: No credit available 0001: 1 credit available 0010: 2 credits available ... 1110: 14 or more credits available 1111: 15 or more credits available 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.
pcie_rq_tag_av	Output	4	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. Its encodings are: 0000: No tags available 0001: 1 tag available 0010: 2 tags available ... 1110: 14 tags available 1111: 15 or more tags available 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.

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. Table 2-25 defines the ports in the Configuration Received Message interface of the core.

Table 2-25: **Configuration Received Message Interface Port Descriptions**

Port	Direction	Width	Description
cfg_msg_received	Output	1	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. Table 3-23 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. 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. The Configuration Received Message interface must be enabled during core configuration in the Vivado IDE.
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 Table 3-23.
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 Table 3-22.

Table 2-26: **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	PME_TO_Ack
13	PME_Turn_Off
14	PM_Active_State_Nak
15	Set_Slot_Power_Limit

Table 2-26: **Message Type Encoding on Receive Message Interface *(Cont'd)***

cfg_msg_received_type[4:0]	Message Type
16	Latency Tolerance Reporting (LTR)
17	Reserved
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 2-27: **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
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

Table 2-27: **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]
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

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. Table 2-28 defines the ports in the Configuration Transmit Message interface of the core.

Table 2-28: **Configuration Transmit Message Interface Port Descriptions**

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: • 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 - 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 - cfg_msg_transmit_data[3:0] < OBFF Code. • 010b: SSPL - cfg_msg_transmit_data[9:0] < {Slot Power Limit Scale, Slot Power Limit Value}. • 011b: PM_PME - 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_mg_transmit assertion, for 1 cycle after the request is complete.

Configuration Flow Control Interface

Table 2-29 defines the ports in the Configuration Flow Control interface of the core.

Table 2-29: **Configuration Flow Control Interface Port Descriptions**

Port	Direction	Width	Description
cfg_fc_ph	Output	8	Posted Header Flow Control Credits. 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.
cfg_fc_pd	Output	12	Posted Data Flow Control Credits. 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.
cfg_fc_nph	Output	8	Non-Posted Header Flow Control Credits. 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.
cfg_fc_npd	Output	12	Non-Posted Data Flow Control Credits. 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.
cfg_fc_cplh	Output	8	Completion Header Flow Control Credits. 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.
cfg_fc_cpld	Output	12	Completion Data Flow Control Credits. 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].

Table 2-29: **Configuration Flow Control Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_fc_sel	Input	3	Flow Control Informational Select. 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: • 000: Receive credits currently available to the link partner • 001: Reserved • 010: Receive credits consumed • 011: Available space in receive buffer • 100: Transmit credits available • 101: Transmit credit limit • 110: Transmit credits consumed • 111: Reserved 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, ¼ full, ½ full, ¾ full, full. ***Note:*** 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.

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 2-30 defines the ports in the Configuration Control interface of the core.

Table 2-30: **Configuration Control Interface Port Descriptions**

Port	Direction	Width	Description
cfg_hot_reset_in	Input	1	Configuration Hot Reset In In RP mode, assertion transitions LTSSM to hot reset state, active-High.
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). If this value is not statically assigned, the user application must pulse user_cfg_input_update after it is stable.
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.

Table 2-30: **Configuration Control Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
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.
cfg_power_state_change_interrupt	Output	1	Power State Change Interrupt 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.
cfg_err_cor_in	Input	1	Correctable Error Detected 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 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.

Table 2-30: **Configuration Control Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_err_uncor_in	Input	1	Uncorrectable Error Detected 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.
cfg_flr_done	Input	4	Function Level Reset Complete 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* and to re-enable configuration accesses to the physical function.
cfg_vf_flr_done	Input	1	Function Level Reset for Virtual Function is Complete 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* and to re-enable configuration accesses to the virtual function.
cfg_vf_flr_func_num	Input	8	Function Level Reset for Virtual Function i is Complete. 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 completes. Valid entries are 8'h04-8'hFF for VF0-VF251. Values 8'h00-8'h03 are reserved.
cfg_flr_in_process	Output	4	Function Level Reset In Process The core asserts bit *i* of this bus when the host initiates a reset of physical function *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.
cfg_vf_flr_in_process	Output	252	Function Level Reset In Process for Virtual Function The core asserts bit *i* of this bus when the host initiates a reset of virtual function *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.

Table 2-30: **Configuration Control Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
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* 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 2-30: **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. Table 2-31 defines the ports in the Configuration Interrupt Controller interface of the core.

Table 2-31: **Configuration Interrupt Controller Interface Port Descriptions**

Name	Direction	Width	Description
Legacy Interrupt Interface			
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 2-31: **Configuration Interrupt Controller 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			
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/MSIX 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/MSIX Initiating Function: Indicates the Endpoint Function # initiating the MSI or MSIX interrupt. 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/MSIX Interrupt Sent: The core generates a one-cycle pulse on this output to signal that an MSI or MSIX 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/MSIX Interrupt Operation Failed: A one-cycle pulse on this output indicates that an MSI or MSIX interrupt message was aborted before transmission on the link. The user logic must retransmit the MSI or MSIX interrupt in this case.

Table 2-31: **Configuration Interrupt Controller Interface Port Descriptions *(Cont'd)***

Name	Direction	Width	Description
cfg_interrupt_msi_mmenable	Output	12	Configuration Interrupt MSI Function Multiple Message Enable: 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: 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	Configuration MSI Interrupt Pending Status: 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.
cfg_interrupt_msi_pending_status_function_num	Input	2	Configuration Interrupt MSI Pending Target Function Number: 00 = PF 0. 01 = PF 1, 10 = PF 2, 11 = PF 3. This input is used to identify the Function number when the user places interrupt status on the cfg_interrupt_msi_pending_status inputs.
cfg_interrupt_msi_pending_status_data_enable	Input	1	Configuration Interrupt MSI Pending Data Valid: User 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.
cfg_interrupt_msi_mask_update	Output	1	Configuration Interrupt MSI Function Mask Updated. 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.
cfg_interrupt_msi_select	Input	2	Configuration Interrupt MSI Select. 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.

Table 2-31: **Configuration Interrupt Controller 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/MSIX TLP Attr: These bits provide the setting of the Attribute bits to be used for both MSI and MSIX interrupt requests. Bit 0 is the No Snoop bit and 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 MSIX).
cfg_interrupt_msi_ tph_present	Input	1	Configuration Interrupt MSI/MSIX TPH Present: Indicates the presence of an optional Transaction Processing Hint (TPH) in the MSI/MSIX 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 MSIX), if it is including a TPH in the MSI or MSIX transaction.
cfg_interrupt_msi_tph_type	Input	2	Configuration Interrupt MSI/MSIX 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_imterrupt_msi_int or cfg_interrupt_msix_int, depending on whether MSI or MSIX interrupts are being used.
cfg_interrupt_msi_tph_st_tag	Input	8	Configuration Interrupt MSI/MSIX 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 MSIX interrupts are being used.
MSIX Interrupt External Interface			
cfg_interrupt_msix_enable	Output	4	Configuration Interrupt MSIX Function Enabled. These outputs reflect the setting of the MSIX Enable bits of the MSIX Control Register of Physical Functions 0 – 3.
cfg_interrupt_msix_mask	Output	4	Configuration Interrupt MSIX Function Mask. These outputs reflect the setting of the MSIX Function Mask bits of the MSIX Control Register of Physical Functions 0 – 3.
cfg_interrupt_msix_vf_enable	Output	252	Configuration Interrupt MSIX Enable from VFs. These outputs reflect the setting of the MSIX Enable bits of the MSIX Control Register of Virtual Functions 0 – 251.
cfg_interrupt_msix_vf_mask	Outputs	252	Configuration Interrupt MSIX VF Mask. These outputs reflect the setting of the MSIX Function Mask bits of the MSIX Control Register of Virtual Functions 0 – 251.

Table 2-31: **Configuration Interrupt Controller Interface Port Descriptions *(Cont'd)***

Name	Direction	Width	Description
cfg_interrupt_msix_address	Input	64	Configuration Interrupt MSIX Address: When the core is configured to support MSIX interrupts and when the MSIX Table is implemented in user memory, this bus is used by the user logic to communicate the address to be used to generate an MSIX interrupt.
cfg_interrupt_msix_data	Input	32	Configuration Interrupt MSIX Data: When the core is configured to support MSIX interrupts and when the MSIX Table is implemented in user memory, this bus is used by the user logic to communicate the data to be used to generate an MSIX interrupt.
cfg_interrupt_msix_int	Input	1	Configuration Interrupt MSIX Data Valid: The assertion of this signal by the user indicates a request from the user to send an MSIX interrupt. The user must place the identifying information on the designated inputs before asserting this interrupt. When the MSIX 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 MSIX Table and Pending Bit Array are implemented by the core, the identifying information consists o 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 2-31: **Configuration Interrupt Controller Interface Port Descriptions *(Cont'd)***

Name	Direction	Width	Description
cfg_interrupt_msix_vec_ pending	Input	2	Configuration Interrupt MSIX Pending Bit Query/Clear: These mode bits are used only when the core is configured to include the MSIX Table and Pending Bit Array. These two bits are set when asserting cfg_interrupt_msix_int to send an MSIX interrupt, to perform certain actions on the MSIX Pending Bit associated with the selected Function and interrupt vector. The various modes are: **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 MSIX 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 MSIX Pending Bit Array (and subsequently transmits the MSIX 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 MSIX 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 MSIX request in response, but asserts cfg_interrupt_msi_sent for one cycle, along with the status of the MSIX 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 MSIX request in response, but clears he MSIX Pending Bit of the vector (if it was set), and activates cfg_interrupt_msi_sent for one cycle as the acknowledgment. The core also provides the previous state of the MSIX 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 MSIX 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.

Table 2-31: **Configuration Interrupt Controller Interface Port Descriptions *(Cont'd)***

Name	Direction	Width	Description
cfg_interrupt_msix_vec_ pending_ status	Output	1	Configuration Interrupt MSIX Pending Bit Status: This output provides the status of the Pending Bit associated with an MSIX 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 MSIX Table and Pending Bit Array.
MSI-X Interrupt Internal Interface			
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_staus	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. Table 2-32 defines the ports in the Configuration Extend interface of the core.

Table 2-32: **Configuration Extend Interface Port Descriptions**

Port	Direction	Width	Description
cfg_ext_read_received	Output	1	Configuration Extend Read Received. The Block asserts this output when it has received a configuration read request from the link. Set when **PCI Express Extended Configuration Space Enable** is selected in User Defined Configuration Capabilities in core configuration in the Vivado IDE. • All received configuration reads with cfg_ext_register_number in the range of 0xb0–0xbf is considered to be PCIe Legacy Extended Configuration Space. • All received configuration reads with cfg_ext_register_number in the range of 0x100–0x3FF is considered to be PCIe Extended Configuration Space. • 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 two aforementioned ranges need to be responded by User Application outside of the IP.
cfg_ext_write_received	Output	1	Configuration Extend Write Received. The Block asserts this output when it has received a configuration write request from the link. Set when **PCI Express Extended Configuration Space Enable** is selected in User Defined Configuration Capabilities in the core configuration in the Vivado IDE. • 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. • All received configuration writes with cfg_ext_register_number in the range 0x100-0x3ff is presented on cfg_ext_register_number, cfg_ext_function_number, cfg_ext_write_data and cfg_ext_write_byte_enable.
cfg_ext_register_number	Output	10	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.
cfg_ext_function_number	Output	8	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.

Table 2-32: **Configuration Extend Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_ext_write_data	Output	32	Configuration Extend Write Data Data being written into a configuration register. This output is valid when cfg_ext_write_received is High.
cfg_ext_write_byte_enable	Output	4	Configuration Extend Write Byte Enable Byte enables for a configuration write transaction.
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.

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. Table 2-33 defines the ports in the Clock and Reset interface of the core.

Table 2-33: **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 IBUFDS_GTE3 (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.

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 Table 2-34.

Table 2-34: **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 (–)
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 (–)

Table 2-34: **PCI Express Interface Signals for 1-, 2-, 4-, 8- and 16-Lane Cores *(Cont'd)***

Lane Number	Name	Direction	Description
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 (–)
8-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 2-34: **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 (–)
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 2-34: **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 (–)

Table 2-34: **PCI Express Interface Signals for 1-, 2-, 4-, 8- and 16-Lane Cores *(Cont'd)***

Lane Number	Name	Direction	Description
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 three primary parts, illustrated in Table 2-36. These include:

- Legacy PCI v3.0 Type 0/1 Configuration Space Header
  - Type 0 Configuration Space Header used by Endpoint applications (see Table 2-35)
  - Type 1 Configuration Space Header used by Root Port applications (see Table 2-35)
- 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)
  - Transaction Processing Hints (TPH) (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 Chapter 4, 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.

For more information about enabling this feature, see Extended Capabilities 1 and Extended Capabilities 2 in Chapter 4.

Table 2-35: **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		*same as Endpoint*			
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 2-36: **PCI Express Config Space**

Byte Offset	Register (Endpoint)				Register (Root/DS Port)	
40h (10h)	PM Capability		NxtCap	PM Cap ID	*same as Endpoint*	
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	Reserved	
64h (19h)	Table Offset			Table BIR	Reserved	
68h (1Ah)	PBA Offset			PBA BIR	Reserved	
6Ch (1Bh)	Reserved				Reserved	
70h (1Ch)	PCIE Capability		NxtCap	PCIE Cap ID	*same as Endpoint*	
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				*same as Endpoint*	
98h (26h)	Device Status 2		Device Control 2			
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 2-37: **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

Table 2-37: **PCIe Capability List *(Cont'd)***

PF0	PF1-3	VF	Start Address
MSI-X	MSI-X	MSI-X	0x60
PCIE	PCIE	PCIE	0x70
Extend	Extend		0xB0

Table 2-38: **PCI Express Extended Configuration Space**

Byte Offset (dw#)	Register (Endpoint)			Register (Root Port)
100h (40h)	Nxt Cap	Cap Ver	AER Ext Cap	*same as Endpoint*
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			
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

Table 2-38: **PCI Express Extended Configuration Space *(Cont'd)***

Byte Offset (dw#)	Register (Endpoint)				Register (Root Port)
140h (50h)	Nxt Cap	Cap Ver	SR-IOV Ext Cap		Reserved
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 2-38: **PCI Express Extended Configuration Space *(Cont'd)***

Byte Offset (dw#)	Register (Endpoint)			Register (Root Port)
1C0h (70h)	Nxt Cap	Cap Ver	2nd PCIE Ext Cap	same as Endpoint
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	
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			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)	Capability Register			
228h (8Ah)	Requester Control Register			
22Ch (8Bh) - 32Ch	TPH Table			

Table 2-38: **PCI Express Extended Configuration Space *(Cont'd)***

Byte Offset (dw#)	Register (Endpoint)			Register (Root Port)		
330h (CCh)	Reserved			Nxt Cap	Cap Ver	Loopback VSEC
334h (CDh)				Loopback Header		
338h (CEh)				Loopback Control		
33Ch (CFh)				Loopback Status		
340h (D0h)				Error Count 1		
344h (D1h)				Error Count 2		
348h (D2h)				Error Count 3		
34Ch (D3h)				Error Count 4		
350h (D4h)	Nxt Cap	Cap Ver	MCAP VSEC	Reserved		
354h (D5h)	MCAP Header					
358h (D6h)	FPGA JTAG ID					
35Ch (D7h)	FPGA 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					

Designing with the Core

This chapter 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 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.
- **Tandem + Partial Reconfiguration**: This is a more general case of Tandem Configuration followed by Partial Reconfiguration (PR) of any size or number of PR regions.
- **Partial Reconfiguration over PCIe**: This is a standard configuration followed by PR, 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 Partial Reconfiguration** option according to your particular case:
   - **Tandem PROM** for the Tandem PROM use case.
   - **Tandem PCIe** for Tandem PCIe or Tandem + Partial Reconfiguration use cases.
   - **Tandem PCIe with Field Updates** ONLY for the predefined Field Updates use case.

     Note: This solution is available as beta only for the following devices: ZU19EG, KU15P, VU3P, VU7P, VU9P. Beta means that implementation is available for all users but bitstream generation is gated by a parameter. Enhancements to bitstream generation and management is coming in a future Vivado release.
   - **PR over PCIe** to enable the MCAP link for Partial Reconfiguration, without enabling Tandem Configuration.

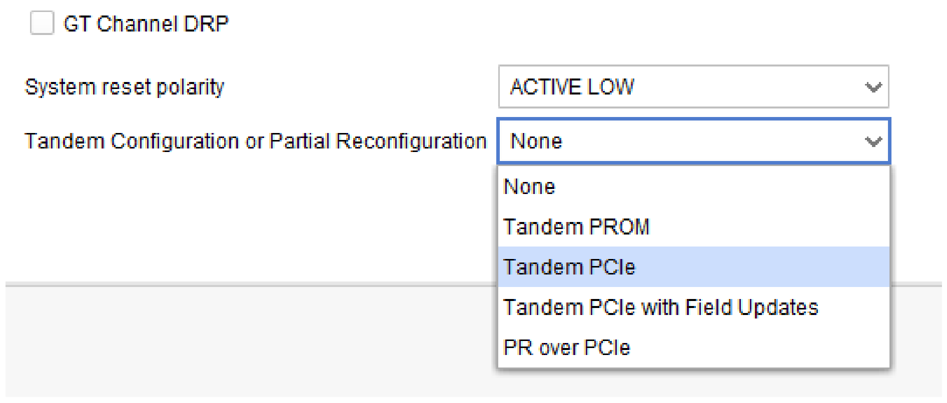


Figure 3-1: **Tandem Configuration or Partial Reconfiguration Option**

Both the AXI Bridge for PCI Express Gen3 Subsystem and the DMA Subsystem for PCI Express IP will support Tandem Configuration and Partial Reconfiguration features for all UltraScale+ devices, including Tandem with Field Updates, but this support is not yet in place with the current Vivado release. These IP subsystems are documented in the *AXI Bridge for PCI Express Gen3 Subsystem Product Guide* (PG194) [Ref 4] and *DMA Subsystem for PCI Express Product Guide* (PG195) [Ref 5] respectively, but the Tandem details are presented in detail only here within this document.

Supported Devices

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

For the Vivado Design Suite 2017.2 release, Tandem Configuration is available as a production solution for a limited set of UltraScale+ devices. Bitstream generation is disabled by default for all ES silicon. Tandem Configuration supports the configurations found in Table 3-1.

Table 3-1: **Tandem PROM/PCIe Supported Configurations**

HDL	Verilog Only			
PCIe Configuration	All configurations (max: X16Gen3 or X8Gen4)			
Xilinx Reference Board Support	None at this time			
Device Support	Supported Part/Package Combinations:			
	Part[1]	**Package**	**PCIe Block Location**	**Status**[2]
Kintex UltraScale+	KU3P	All	PCIE40E4_X0Y0	Not yet supported
	KU5P	All	PCIE40E4_X0Y0	Not yet supported
	KU11P	All	PCIE40E4_X1Y0	Not yet supported
	KU15P	All	PCIE40E4_X1Y0	Production
Virtex UltraScale+	VU3P	All	PCIE40E4_X1Y0	Production
	VU5P	All	PCIE40E4_X1Y0	Production
	VU7P	All	PCIE40E4_X1Y0	Production
	VU9P	All	PCIE40E4_X1Y2	Production
	VU11P	All	PCIE40E4_X0Y0	Not yet supported
	VU13P	All	PCIE40E4_X0Y1	Production
Zynq MPSoC	ZU4EV	All	PCIE40E4_X0Y1	Not yet supported
	ZU5EV	All	PCIE40E4_X0Y1	Not yet supported
	ZU7EV	All	PCIE40E4_X0Y1	Production
	ZU11EG	All	PCIE40E4_X1Y0	Not yet supported
	ZU17EG	All	PCIE40E4_X1Y0	Not yet supported
	ZU19EG	All	PCIE40E4_X1Y0	Production

Notes:

1. Only production silicon is officially supported. Bitstream generation is disabled for all engineering sample silicon (ES1, ES2) devices.
2. Status is for the AXI streaming core without Field Updates. Field Updates support is a smaller set of devices listed earlier.

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 Table 3-1, select the **Advanced** configuration **Mode** option, and select **Tandem PROM or Tandem PCIe** for the Tandem Configuration or Partial Reconfiguration 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 Configuration is currently supported only for the AXI4-Stream version of the core except for two devices (VU9P and VU13P), and must be generated through the IP catalog.

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 Figure 3-2, the resulting BIT file is monolithic and contains both stage 1 and stage 2.

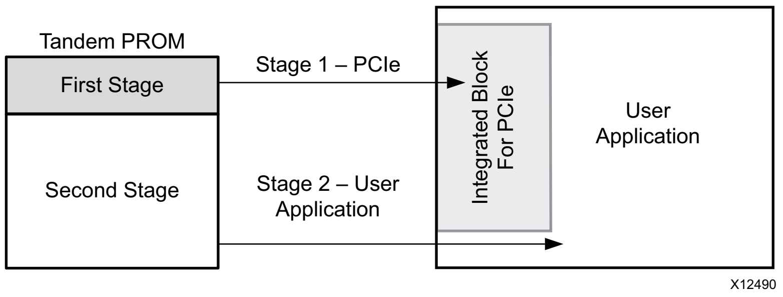


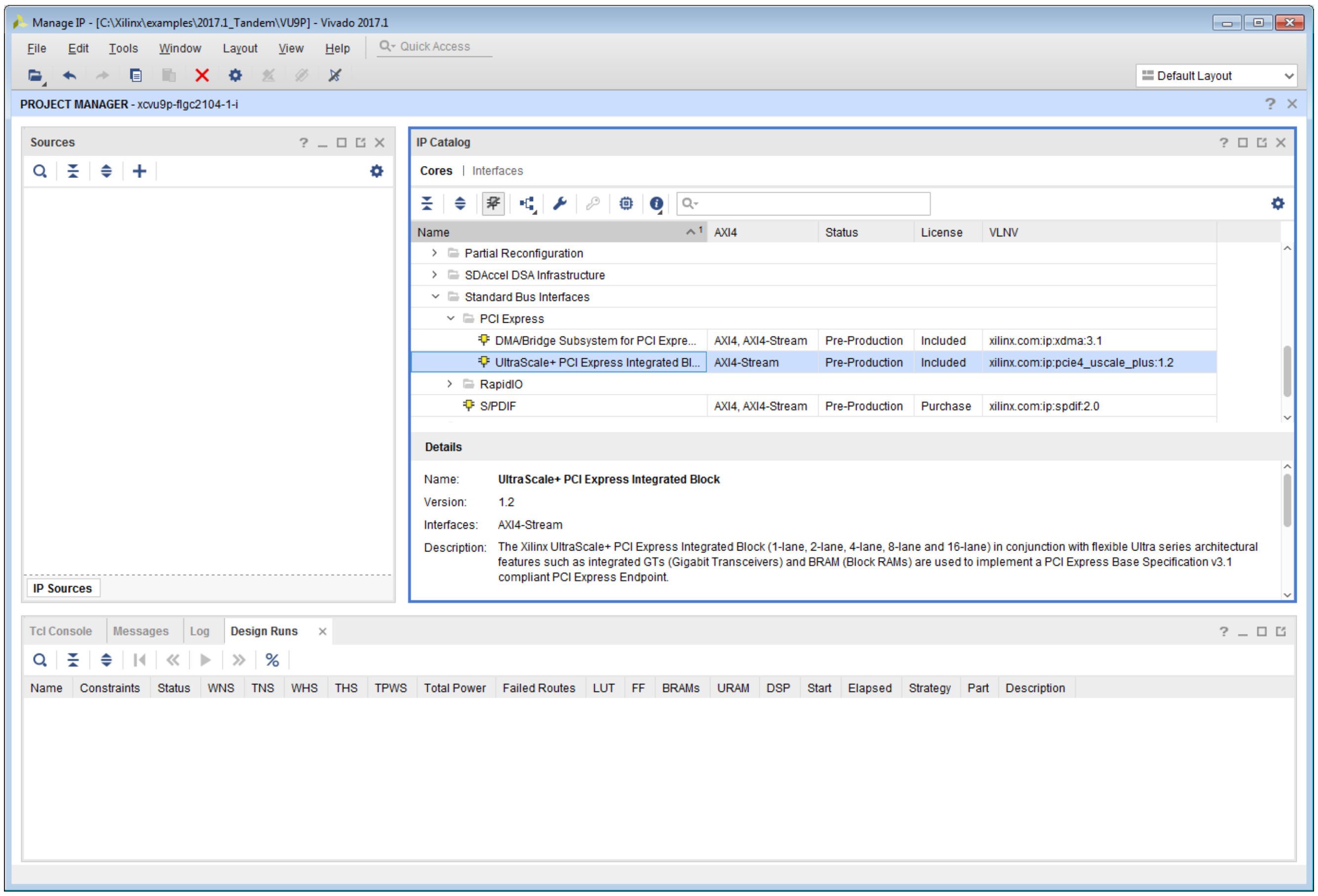
Figure 3-2: **Tandem PROM Bitstream Load Steps**

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 Table 3-1.

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.



Name	AXI4	Status	License	VLNV
Partial Reconfiguration				
SDAccel DSA Infrastructure				
Standard Bus Interfaces				
PCI Express				
DMA/Bridge Subsystem for PCI Expre...	AXI4, AXI4-Stream	Pre-Production	Included	xilinx.com:ip:xdma:3.1
UltraScale+ PCI Express Integrated Bl...	AXI4-Stream	Pre-Production	Included	xilinx.com:ip:pcie4_uscale_plus:1.2
RapidIO				
S/PDIF	AXI4, AXI4-Stream	Pre-Production	Purchase	xilinx.com:ip:spdif:2.0

Name: **UltraScale+ PCI Express Integrated Block**
Version: 1.2
Interfaces: AXI4-Stream
Description: The Xilinx UltraScale+ PCI Express Integrated Block (1-lane, 2-lane, 4-lane, 8-lane and 16-lane) in conjunction with flexible Ultra series architectural features such as integrated GTs (Gigabit Transceivers) and BRAM (Block RAMs) are used to implement a PCI Express Base Specification v3.1 compliant PCI Express Endpoint.

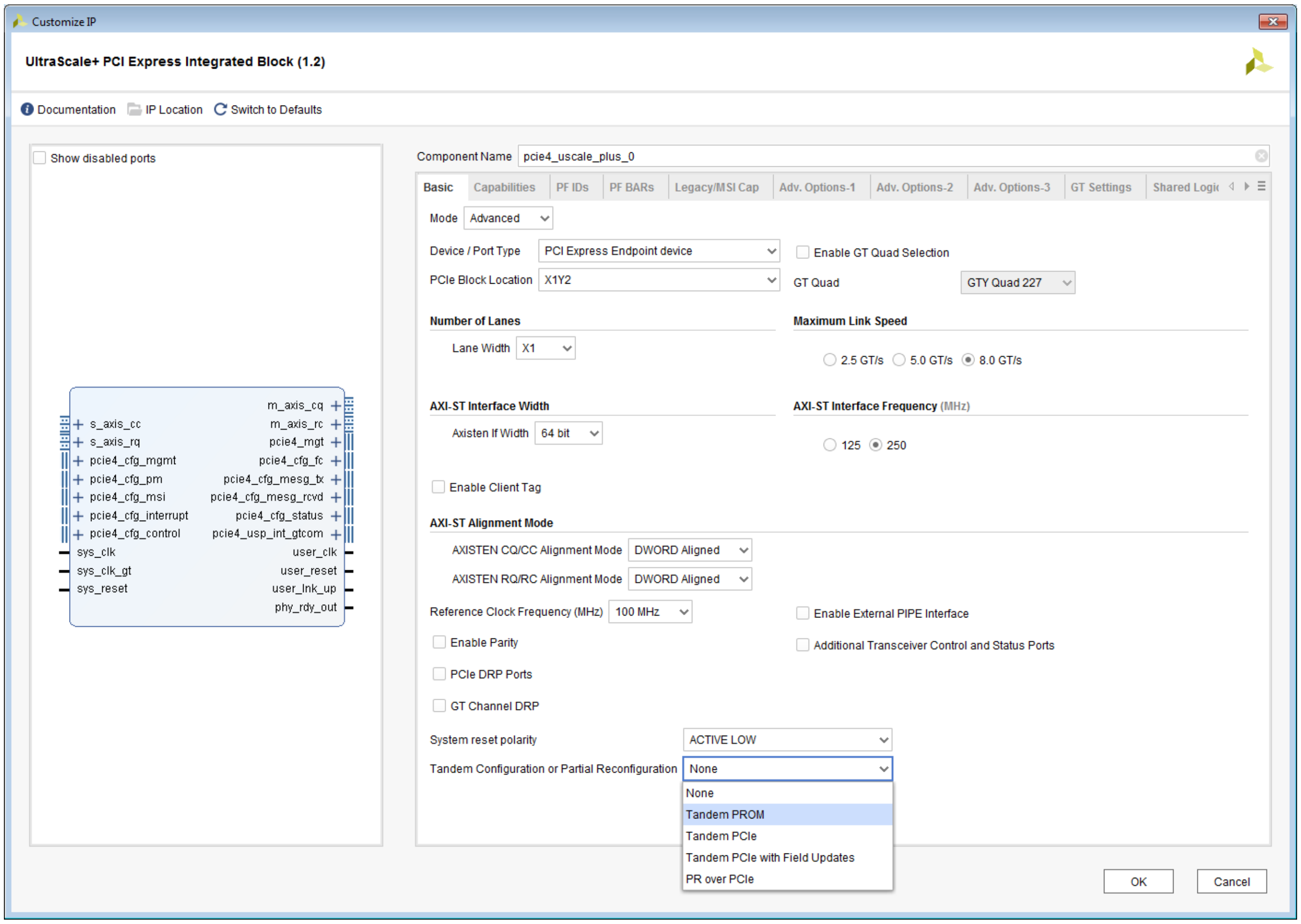
Figure 3-3: **Vivado IP Catalog**

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 Table 3-1. This design example targets a VU9P.

   - Tandem Configuration or Partial Reconfiguration: **Tandem PROM**



UltraScale+ PCI Express Integrated Block (1.2)

Component Name pcie4_uscale_plus_0

Mode Advanced

Device / Port Type PCI Express Endpoint device

PCIe Block Location X1Y2

Enable GT Quad Selection

GT Quad GTY Quad 227

Number of Lanes

Lane Width X1

Maximum Link Speed

2.5 GT/s 5.0 GT/s [✓ 8.0 GT/s]

AXI-ST Interface Width

Axisten If Width 64 bit

AXI-ST Interface Frequency (MHz)

125 [✓ 250]

Enable Client Tag

AXI-ST Alignment Mode

AXISTEN CQ/CC Alignment Mode DWORD Aligned

AXISTEN RQ/RC Alignment Mode DWORD Aligned

Reference Clock Frequency (MHz) 100 MHz

Enable External PIPE Interface

Enable Parity

Additional Transceiver Control and Status Ports

PCIe DRP Ports

GT Channel DRP

System reset polarity ACTIVE LOW

Tandem Configuration or Partial Reconfiguration None

None
Tandem PROM
Tandem PCIe
Tandem PCIe with Field Updates
PR over PCIe

Figure 3-4: **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. Setup 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 (e.g. `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, page 103.

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 BPI -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.*

Figure 3-5 illustrates the bitstream loading flow.

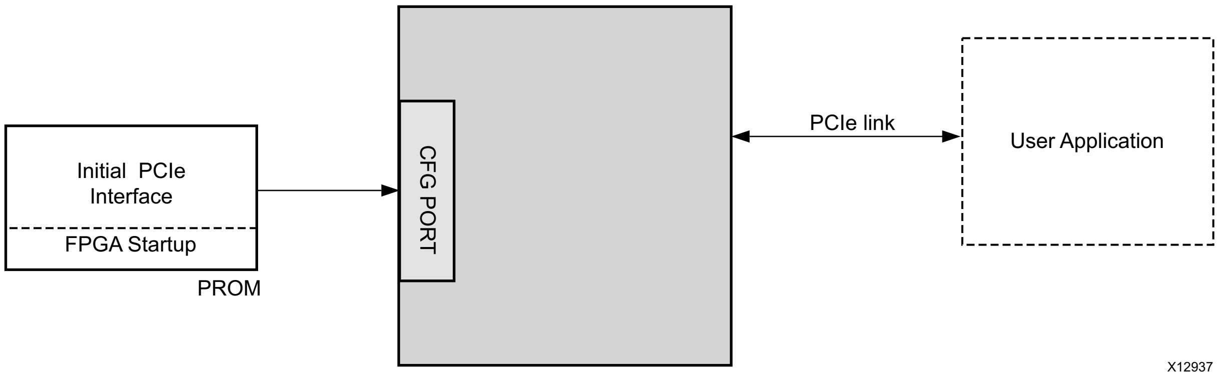


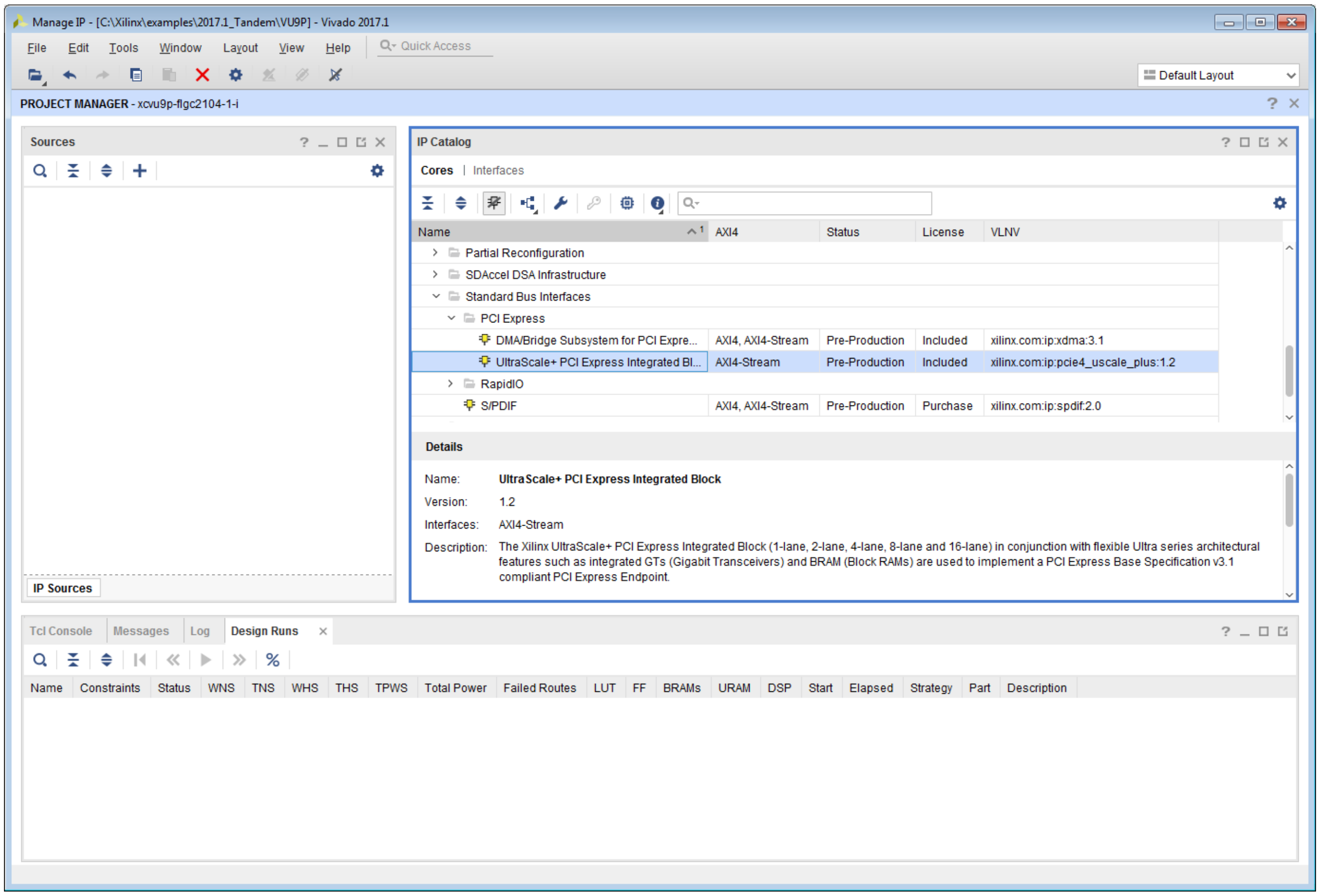
Figure 3-5: **Tandem PCIe Bitstream Load Steps**

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 is downloaded into the PROM while the other BIT file representing the user application (stage 2) configures the remainder 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 Table 3-1.
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.


Manage IP - [C:\Xilinx\examples\2017.1_Tandem\VU9P] - Vivado 2017.1

PROJECT MANAGER - xcvu9p-flgc2104-1-i

IP Catalog — Cores | Interfaces

Name	AXI4	Status	License	VLNV
Partial Reconfiguration				
SDAccel DSA Infrastructure				
Standard Bus Interfaces				
PCI Express				
DMA/Bridge Subsystem for PCI Expre...	AXI4, AXI4-Stream	Pre-Production	Included	xilinx.com:ip:xdma:3.1
UltraScale+ PCI Express Integrated Bl...	AXI4-Stream	Pre-Production	Included	xilinx.com:ip:pcie4_uscale_plus:1.2
RapidIO				
S/PDIF	AXI4, AXI4-Stream	Pre-Production	Purchase	xilinx.com:ip:spdif:2.0

Details

Name:	UltraScale+ PCI Express Integrated Block
Version:	1.2
Interfaces:	AXI4-Stream
Description:	The Xilinx UltraScale+ PCI Express Integrated Block (1-lane, 2-lane, 4-lane, 8-lane and 16-lane) in conjunction with flexible Ultra series architectural features such as integrated GTs (Gigabit Transceivers) and BRAM (Block RAMs) are used to implement a PCI Express Base Specification v3.1 compliant PCI Express Endpoint.

Design Runs

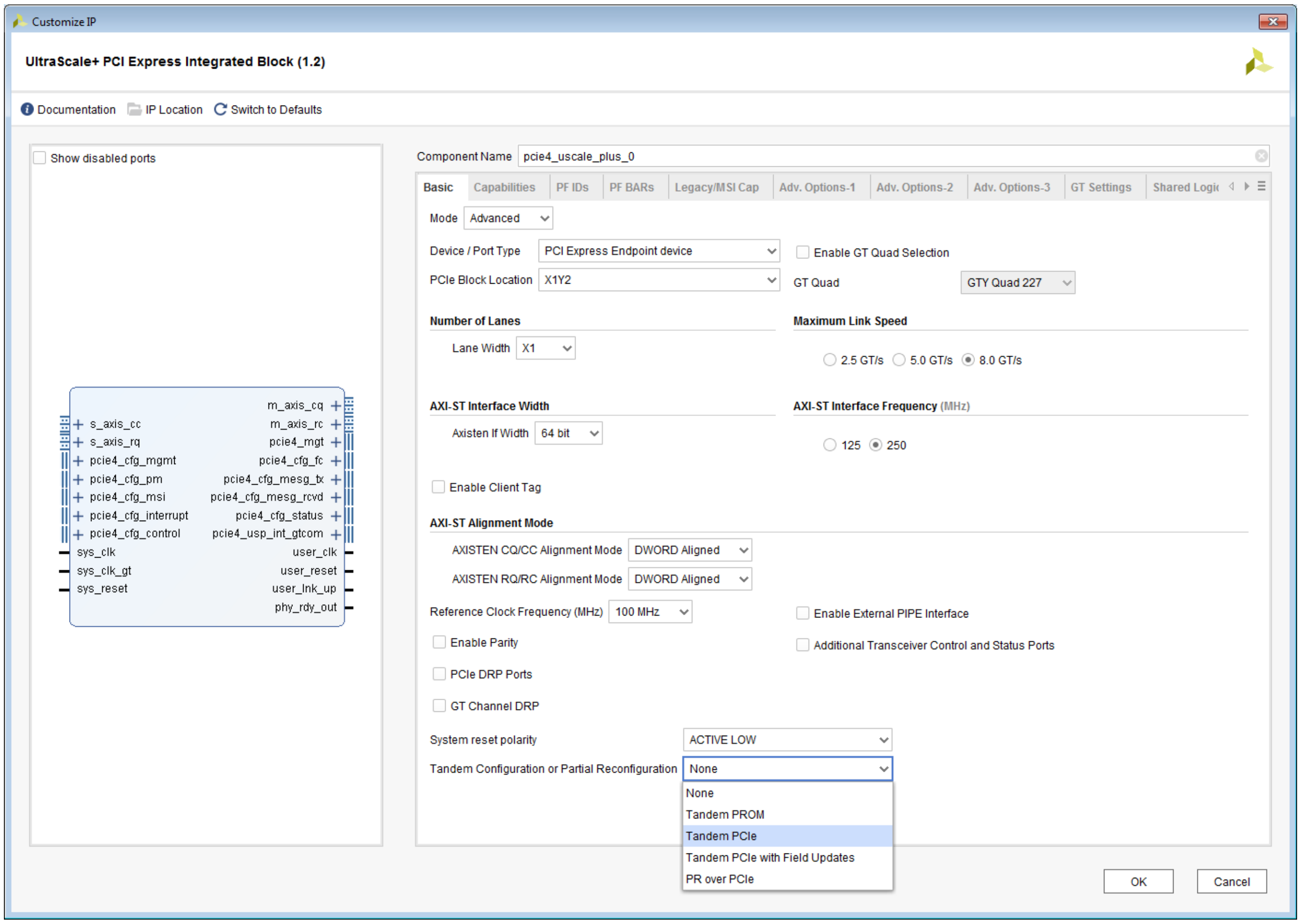
Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP	Start	Elapsed	Strategy	Part	Description

Figure 3-6: **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 Table 3-1. This design example targets a VU9P.

   - Tandem Configuration or Partial Reconfiguration: **Tandem PCIe**


Customize IP

UltraScale+ PCI Express Integrated Block (1.2)

Documentation | IP Location | Switch to Defaults

Show disabled ports

Component Name: pcie4_uscale_plus_0

Tabs: Basic | Capabilities | PF IDs | PF BARs | Legacy/MSI Cap | Adv. Options-1 | Adv. Options-2 | Adv. Options-3 | GT Settings | Shared Logic

Mode: Advanced

Device / Port Type: PCI Express Endpoint device

Enable GT Quad Selection [ ]

PCIe Block Location: X1Y2

GT Quad: GTY Quad 227

Number of Lanes

Lane Width: X1

Maximum Link Speed

2.5 GT/s 5.0 GT/s [✓ 8.0 GT/s]

AXI-ST Interface Width

Axisten If Width: 64 bit

AXI-ST Interface Frequency (MHz)

125 [✓ 250]

Enable Client Tag [ ]

AXI-ST Alignment Mode

AXISTEN CQ/CC Alignment Mode: DWORD Aligned

AXISTEN RQ/RC Alignment Mode: DWORD Aligned

Reference Clock Frequency (MHz): 100 MHz

Enable External PIPE Interface [ ]

Enable Parity [ ]

Additional Transceiver Control and Status Ports [ ]

PCIe DRP Ports [ ]

GT Channel DRP [ ]

System reset polarity: ACTIVE LOW

Tandem Configuration or Partial Reconfiguration: None

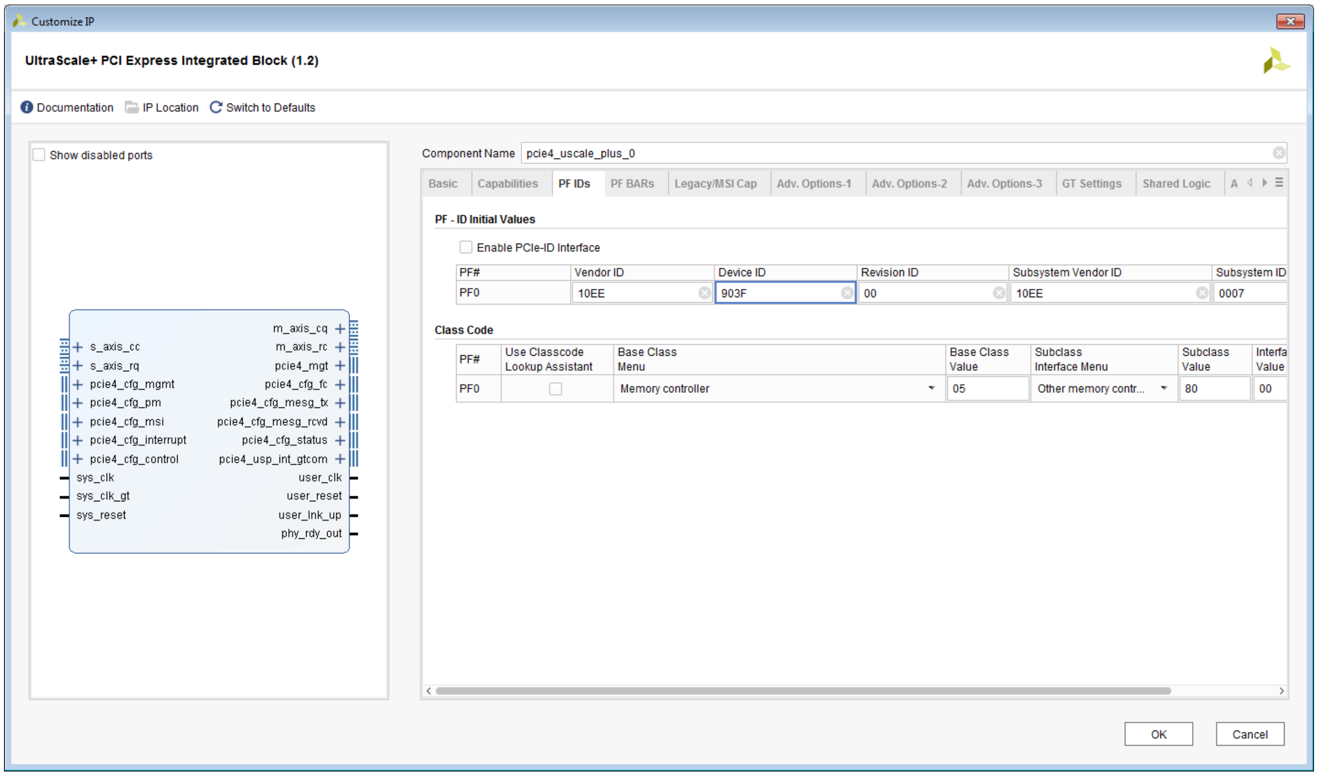
None
Tandem PROM
Tandem PCIe
Tandem PCIe with Field Updates
PR over PCIe

OK | Cancel

Figure 3-7: **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 **ID** 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.



Component Name pcie4_uscale_plus_0

PF - ID Initial Values

Enable PCIe-ID Interface

PF#	Vendor ID	Device ID	Revision ID	Subsystem Vendor ID	Subsystem ID
PF0	10EE	903F	00	10EE	0007

Class Code

PF#	Use Classcode Lookup Assistant	Base Class Menu	Base Class Value	Subclass Interface Menu	Subclass Value	Interfa Value
PF0	☐	Memory controller	05	Other memory contr...	80	00

Figure 3-8: **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 (e.g. `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, page 103.

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_pcie4_uscale_plus_ep_tandem1.bit|
xilinx_pcie4_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_pcie4_uscale_plus_ep_tandem1.bit” xilinx_pcie4_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.

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 with Field Updates

Tandem 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. This solution is only planned for the Tandem PCIe flow. No support for Tandem PROM with Field Updates is planned.

This solution is beta in Vivado 2017.1 and 2017.2, and as such the bitstreams are gated by a parameter; contact Xilinx support for access. In a future version of Vivado, enhancements to this solution will be available when it becomes production. One enhancement, aligned with Partial Reconfiguration, is the elimination of clearing bitstreams, simplifying bitstream management.

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 the 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 the 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. Table 3-2 defines the handshaking ports on the core.

Table 3-2: **Handshaking Ports**

Name	Direction	Polarity	Description
mcap_design_switch	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.
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`.

Table 3-2: **Handshaking Ports *(Cont'd)***

Name	Direction	Polarity	Description
user_clk	Output	N/A	Clock to be used by PCIe interfacing logic.
user_lnk_up	Output	Active-High	Identifies that the PCI 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 stage2 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_out_obuf (.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_out_obuf`.

```
# 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_out_obuf]

    # 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_out_obuf]
```

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)

Configuration Persist is required in Tandem PROM configuration for UltraScale+ devices. Dual purpose I/O used for stage 1 and stage 2 configuration cannot be re-purposed as user I/O after stage 2 configuration is complete.

If the PERSIST 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.

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)* [Ref 7] for a list of supported PROMs and device bitstream sizes.

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 capabilities, thus preventing proper stage 2 loading.

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)* [Ref 7] for details on the use of EMCCLK with the Design Suite.

For more information on configuration in the Vivado Design Suite, see the *Vivado Design Suite User Guide: Programming and Debugging (UG908)* [Ref 21].

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.

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 No Wait 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.

The following is a sample flow in a non-project environment:

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.

   ```
   synth_design -top <top_level>
   ```

   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
   ```

Simulating the Tandem IP Core

Because the functionality of the Tandem PROM or Tandem PCIe core relies on the STARTUP module, this must be taken into consideration during simulation.

The PCI Express core relies on the STARTUP block to assert the EOS output status signal in order to know when the stage 2 bitstream has been loaded into the device. You must simulate the STARTUP block behavior to release the PCIe core to work with the stage 2 logic. This is done using a hierarchical reference to force the EOS signal on the STARTUP block because result simulators, which do not support hierarchical reference, cannot be used to simulate Tandem designs. The following pseudo code shows how this could be done.

```
// Initialize EOS at time 0
force board.EP.pcie_ip_support_i.pcie_ip_i.inst.startup_i.EOS = 1'b1;
```

<delay until after PCIe reset is released>

```
// Deassert EOS to simulate the starting of the 2nd stage bitstream loading
force board.EP.pcie_ip_support_i.pcie_ip_i.inst.startup_i.EOS = 1'b0;
```

<delay a minimum of 4 user_clk cycles>

```
// Reassert EOS to simulate that 2nd stage bitstream completed loading
force board.EP.pcie_ip_support_i.pcie_ip_i.inst.startup_i.EOS = 1'b1;
// Simulate as normal from this point on.
```

The hierarchy to the PCIe core in the line above must be changed to match that of the user design. This line can also be found in the example simulation provided with the core in the file named `board.v`.

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.

   Minimum Clock Frequency = Nominal Clock - 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.

   PROM Bandwidth = Minimum Clock Frequency × 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`.

   Stage 1 Load Time = (PROM Bandwidth) / (Stage 1 Bitstream Size)

   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 MHz × (1 - 0.0002) = 65.98 MHz

2. Calculate the minimum PROM bandwidth:

   4 bits × 65.98 MHz = 263.92 Mb/s

3. Calculate the first-stage bitstream loading time:

   11.27 Mb / 263.92 Mb/s = ~0.0427 or 42.7 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 MHz × (1 - 0.0001) = 49.995 MHz

2. Calculate the minimum PROM bandwidth:

   16 bits × 49.995 MHz = 799.92 Mb/s

3. Calculate the first-stage bitstream loading time:

   11.27 Mb / 799.92 Mb/s = ~0.0141 s or 14.1 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)* [Ref 7].

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 Architecture Data Sheet: DC and AC Switching Characteristics (DS922)* [Ref 13], and *Virtex UltraScale Architecture Data Sheet: DC and AC Switching Characteristics (DS923)* [Ref 14].

Consider two cases for Example 1 (Quad SPI flash [x4] operating at 66 MHz ± 200 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} + *Bitstream Load Time* + T_{FPGA_PWR} < 100 ms for non-ATX

T_{POR} + *Bitstream Load Time* + T_{FPGA_PWR} - 100 ms < 100 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*).

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

94.7 ms < 100 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 ms (T_{POR}) + 42.7 ms (bitstream time) + 2 ms (ramp time) - 100 ms = -5.3 ms

-5.3 ms < 100 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 3-3: **Example Bitstream Size and Configuration Times**[1]

Device	Full Bitstream	Full: BPI16 at 50 MHz	Tandem Stage 1[2]	Tandem: BPI16 at 50 MHz
KU15P	277.3 Mb	346.6 ms	17.6 Mb	22.0 ms
VU9P	611.6 Mb	764.5 ms	17.5 Mb	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 2017.2.

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 100/125/250 MHz reference clock input. For more information, see the Answer Records at the Xilinx PCI Express Solution Center.

Figure 3-9 shows the example clocking architecture.

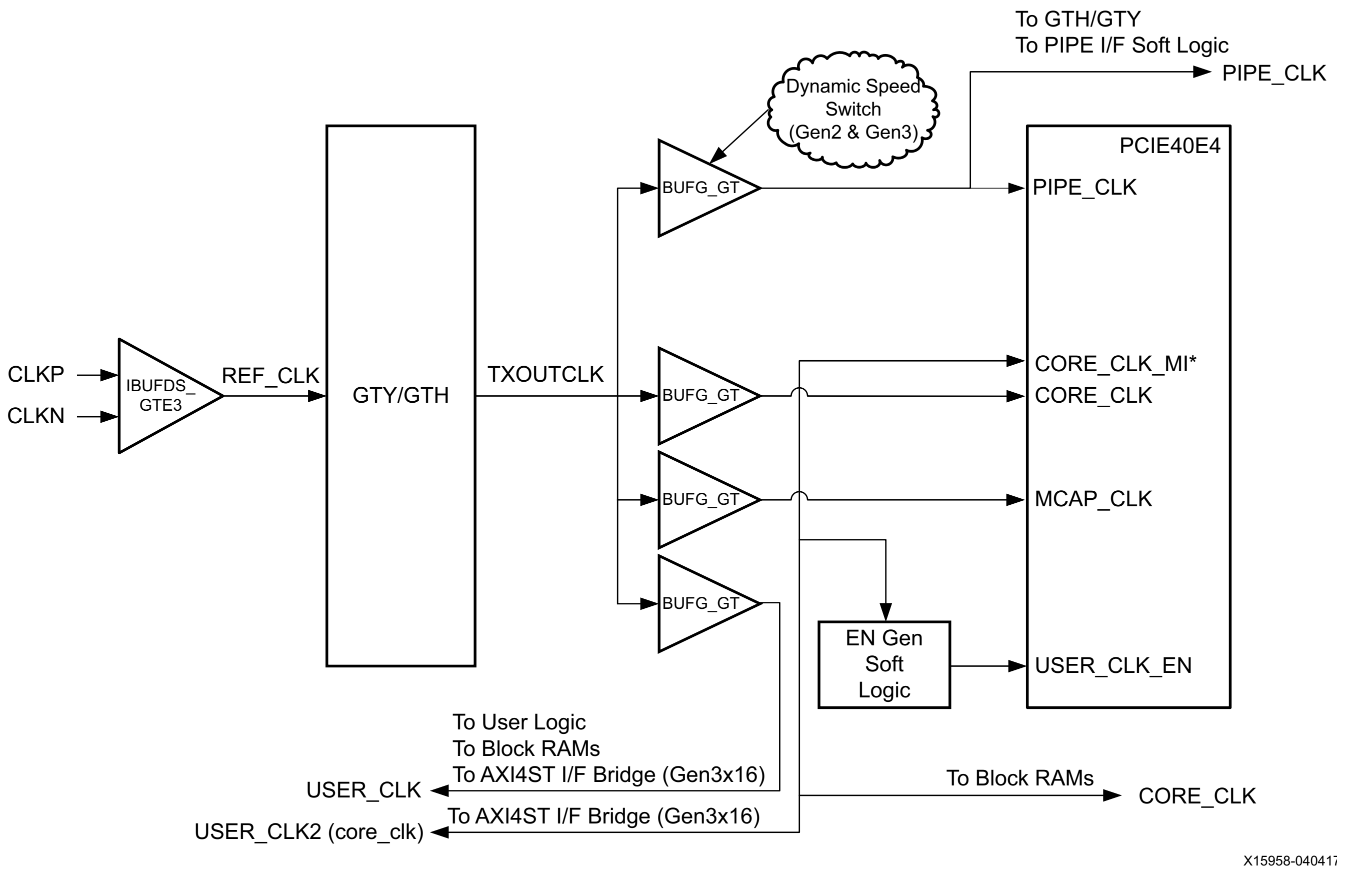


Figure 3-9: **Clocking Architecture**

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 Figure 3-9).

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* [Ref 2].

IMPORTANT: *All add-in card designs must use synchronous clocking due to the characteristics of the provided reference clock. 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. Figure 3-10 and Figure 3-11 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: Figure 3-10 and Figure 3-11 are high-level representations of the board layout. Ensure that coupling, termination, and details are correct when laying out a board. See Board Design Guidelines in the *UltraScale Architecture GTH Transceivers User Guide* (UG576) [Ref 11].

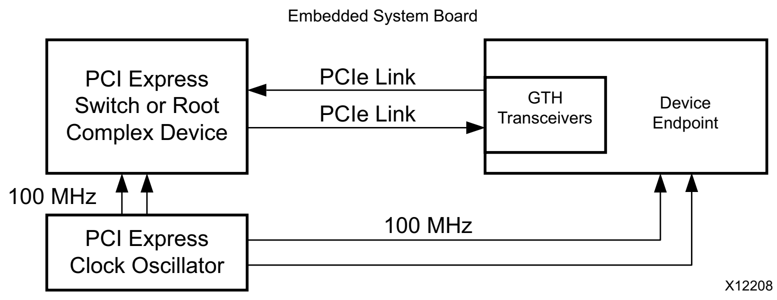


Figure 3-10: **Embedded System Using 100 MHz Reference Clock**

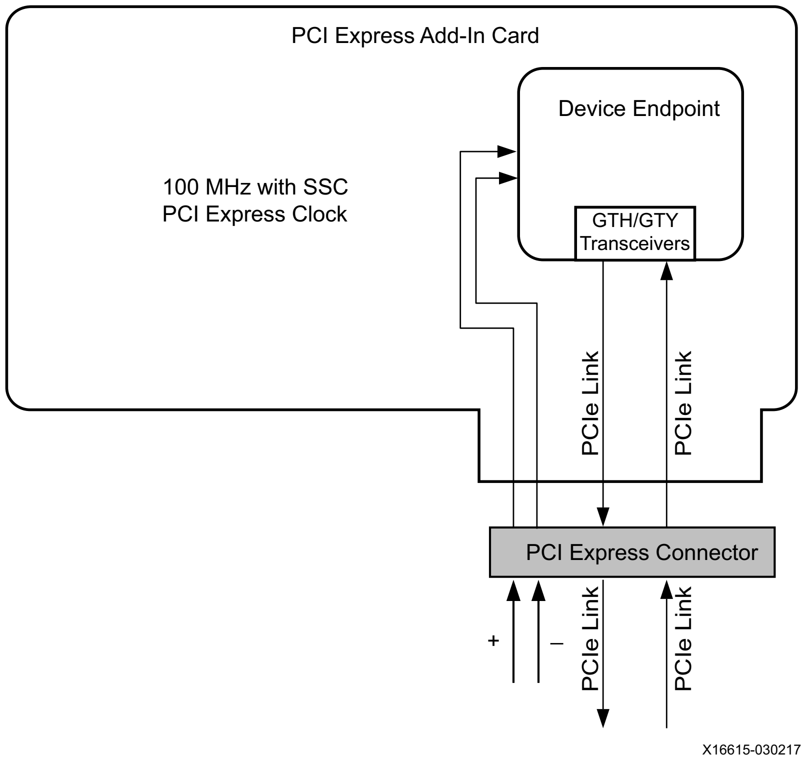


Figure 3-11: **Open System Add-In Card Using 100 MHz Reference Clock**

Starting 2017.1, 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 GTH 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.

  Before FLR is initiated, the software temporarily disables the traffic targeting the specific functions. When the FLR is initiated, Requests and Completions are silently discarded without logging or signaling an error.

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

Figure 3-12 illustrates the user interface of the core.

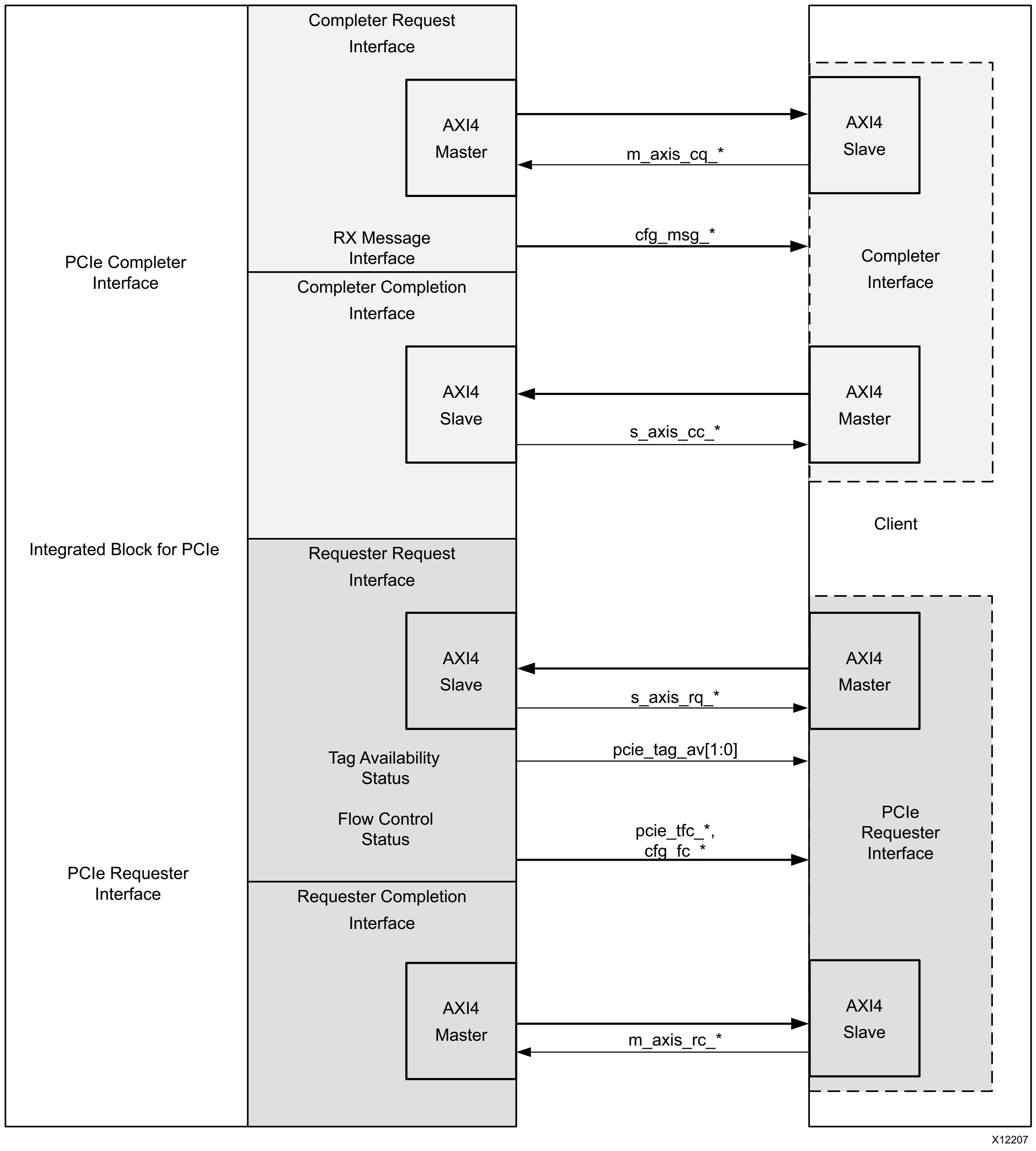


Figure 3-12: **Block Diagram of Integrated Block User Interfaces**

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 integrated block returns the completions received from the link (in response to the user application requests as PCIe requester) to the user application.

Each of the four interfaces is based on the AMBA4® AXI4-Stream Protocol Specification [Ref 1]. 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 PCIe generation 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.

Table 3-4 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 3-4: **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)	GTH/GTY 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	250	250	250	125	250
		16/32	256	125/250	500	250	250	125	500
	X16	16/32	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 mod 4)) mod 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 mod 16)) mod 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.

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 the 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 Figure 3-65 and Figure 3-98 for waveforms for 256 bits and 512 bits 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 4, 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.

Table 3-5 lists the valid combinations of interface width, addressing mode, and the straddle option.

Table 3-5: **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, page 140.

- 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 3-6: **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

Figure 3-13 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.

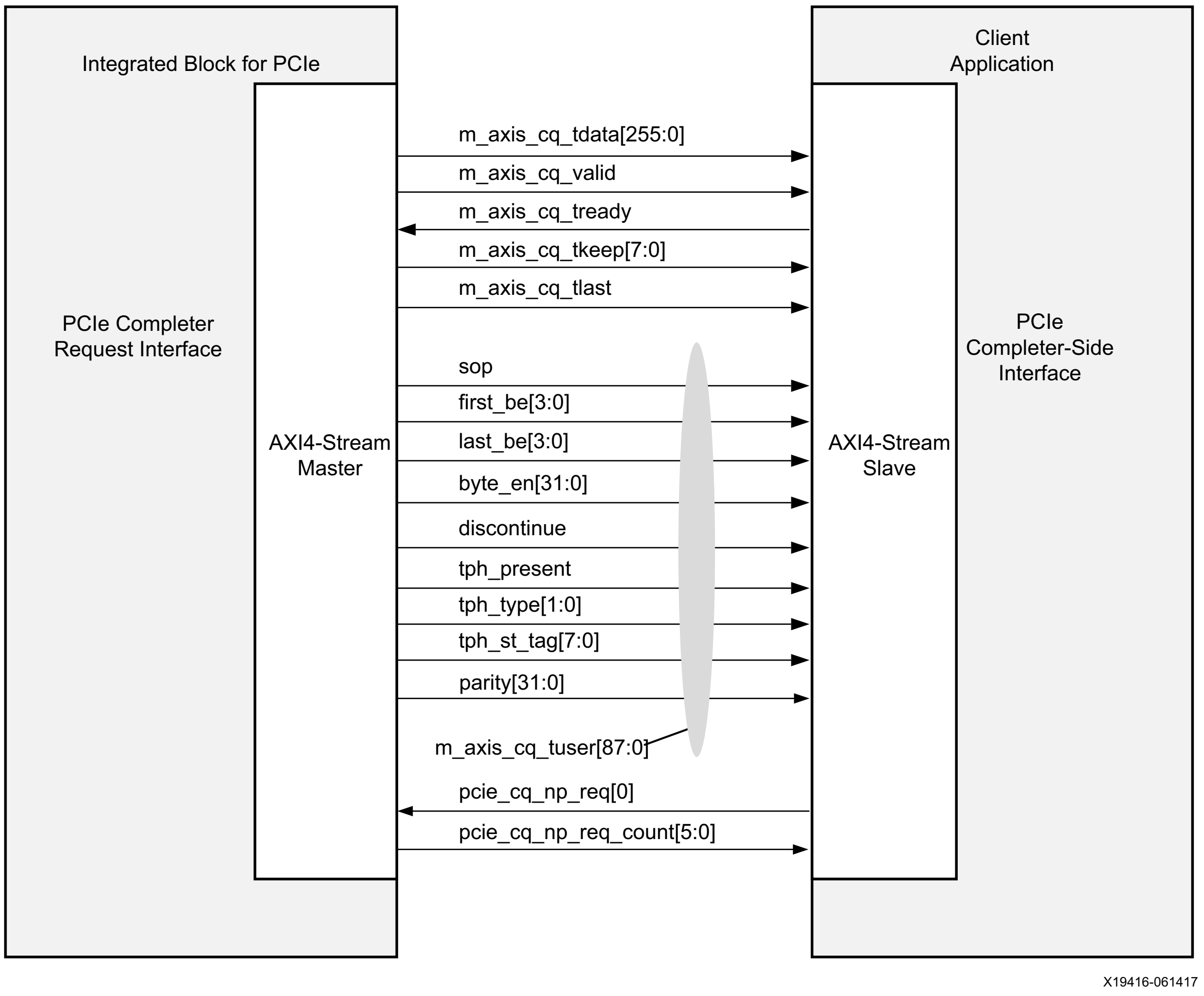


Figure 3-13: **Completer Request Interface Signals**

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 mod 4) mod w, where:

- A is the byte-level starting address of the data block being transferred
- 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 mod 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 Figure 3-14, Figure 3-15, Figure 3-16, and Figure 3-17. The format of Figure 3-14 applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request. The format of Figure 3-15 is used for Vendor-Defined Messages (Type 0 or Type 1) only. The format of Figure 3-16 is used for all ATS messages (Invalid Request, Invalid Completion, Page Request, PRG Response). For all other messages, the descriptor takes the format of Figure 3-17.

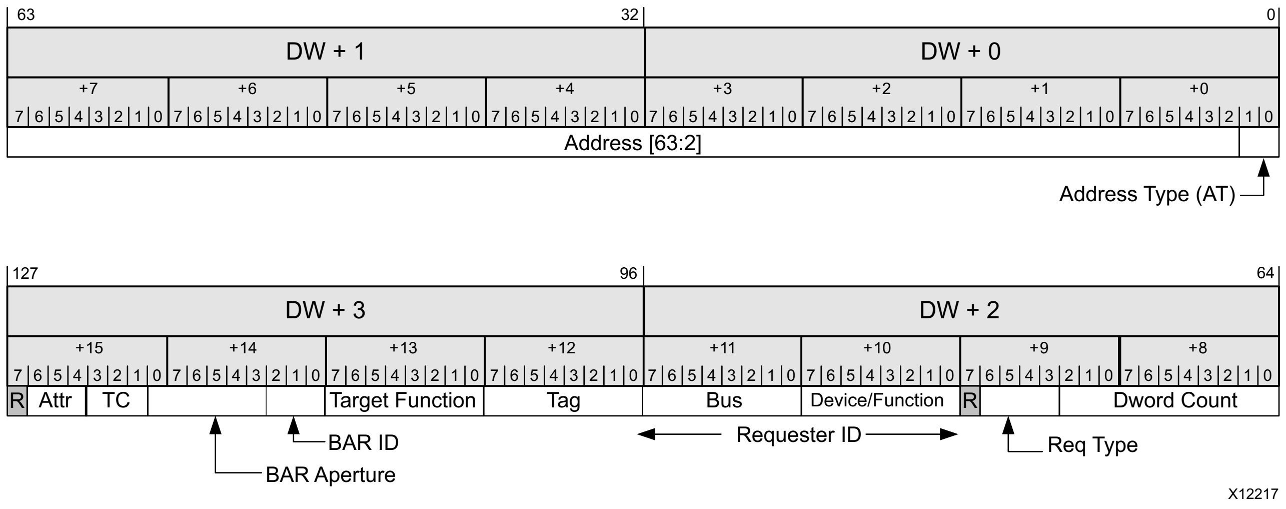


Figure 3-14: **Completer Request Descriptor Format for Memory, I/O, and Atomic Op Requests**

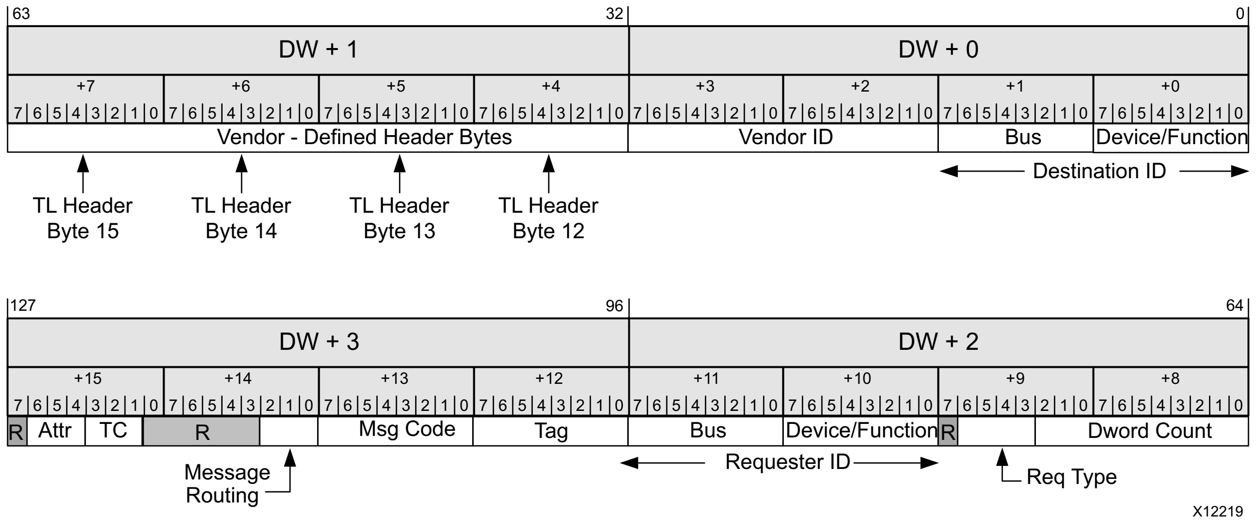


Figure 3-15: **Completer Request Descriptor Format for Vendor-Defined Messages**

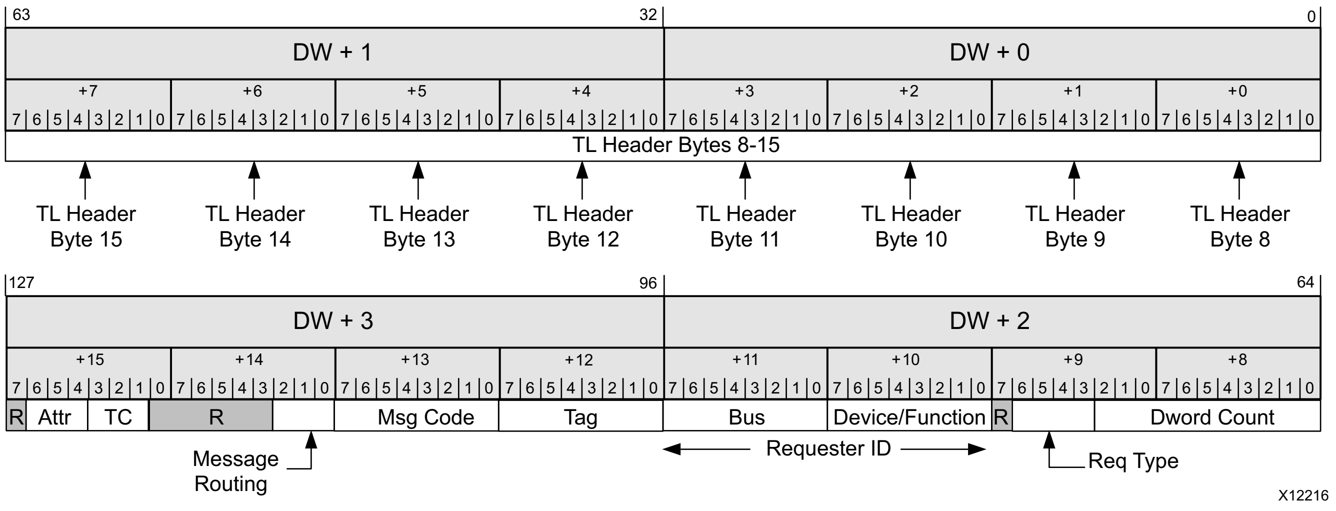


Figure 3-16: **Completer Request Descriptor Format for ATS Messages**

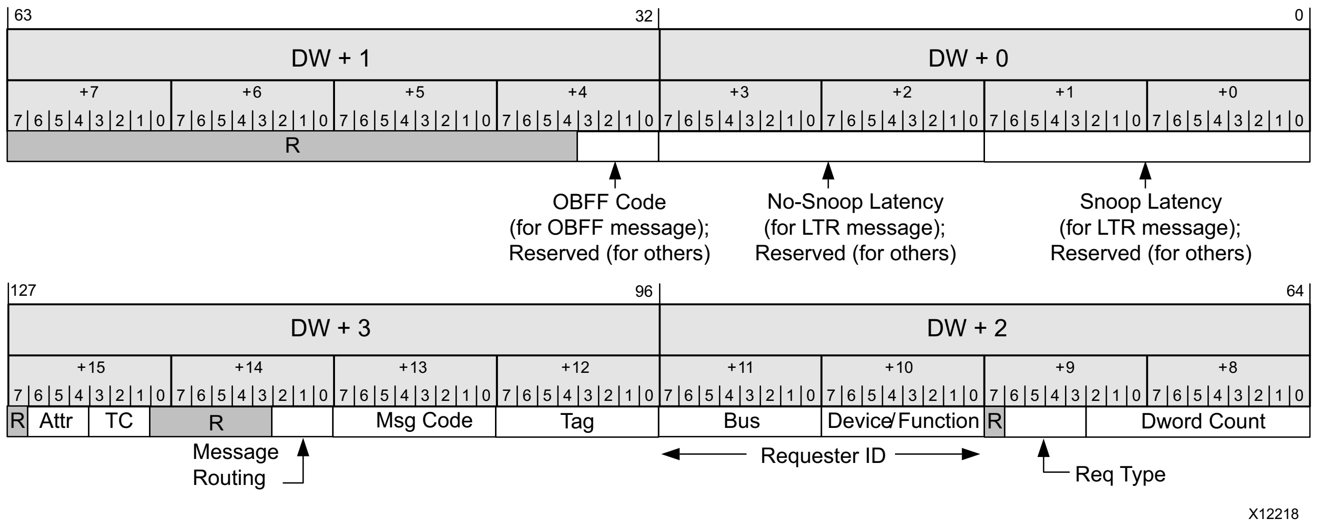


Figure 3-17: **Completer Request Descriptor Format for All Other Messages**

Table 3-7 describes the individual fields of the completer request descriptor.

Table 3-7: **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. 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 3-7: **Completer Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
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 3-8.
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.
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	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. Following are Target Function Value to PF/VF map mappings: • 0: PF0 • 1: PF1 • 64: VF0 • 65: VF1 • 66: VF2 • 67: VF3 • 68: VF4 • 69: VF5

Table 3-7: **Completer Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
114:112	BAR ID	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. • 000: BAR 0 (VF-BAR 0 for VFs). **Note:** In RP mode, BAR ID is always 000. • 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 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).
120:115	BAR Aperture	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. For VF BARs, the value provided on this output is based on the memory space consumed by a single VF covered by the BAR.
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	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 integrated block with the completion data.
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	This field is defined for OBFF messages only. The OBFF Code field is used to distinguish between various OBFF cases: • 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.

Table 3-7: **Completer Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
111:104	Message Code	This field is defined for all messages. It contains the 8-bit Message Code extracted from the TLP header. Appendix F of the *PCI Express Base Specification, rev. 3.0* [Ref 2] provides a complete list of the supported Message Codes. 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.
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 3-8: **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 timing diagrams in Figure 3-18, Figure 3-19, and Figure 3-20 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_axi_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.

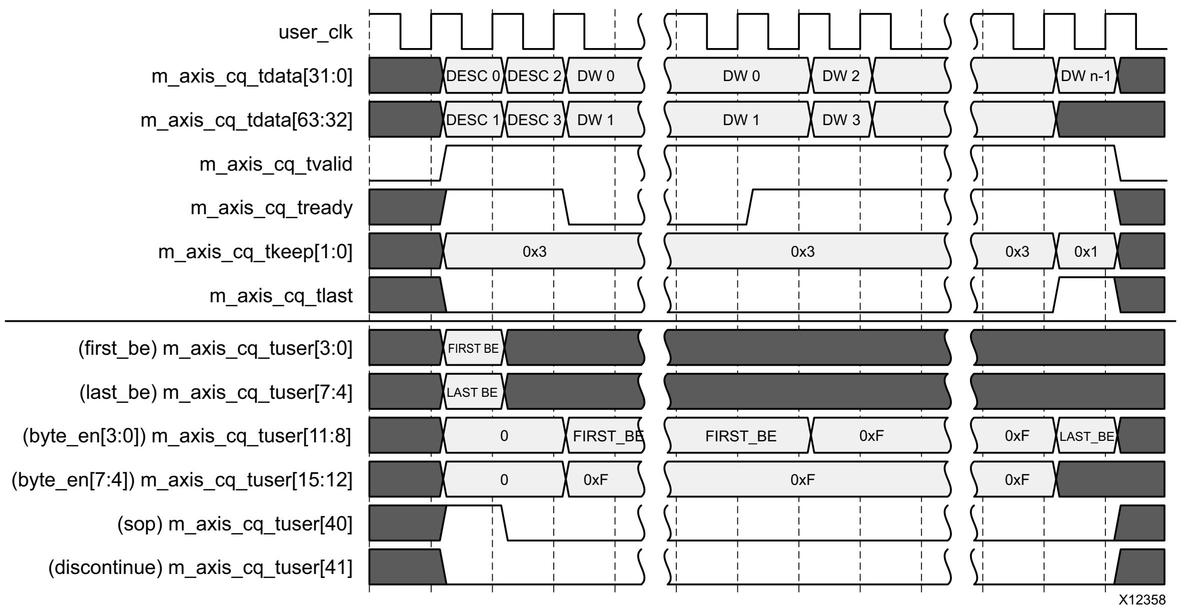


Figure 3-18: **Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode, 64-Bit Interface)**

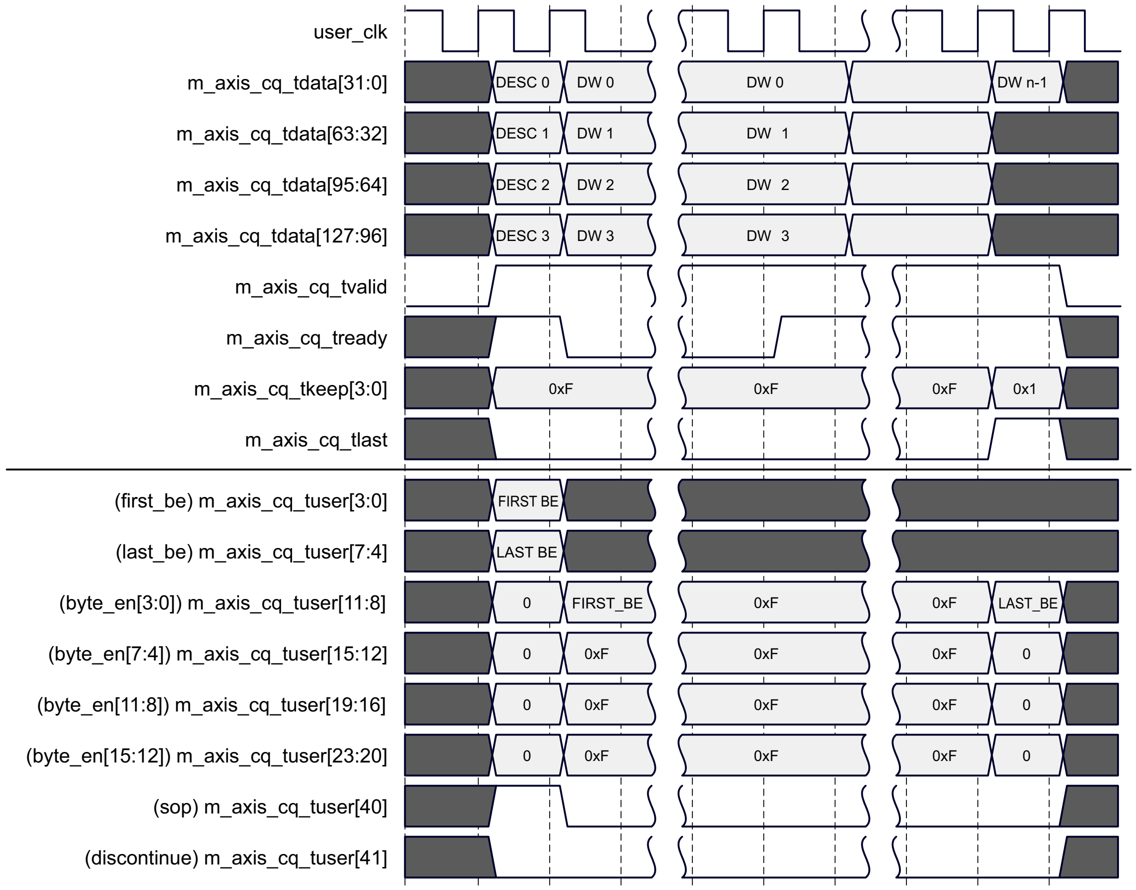


Figure 3-19: **Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode, 128-Bit Interface)**

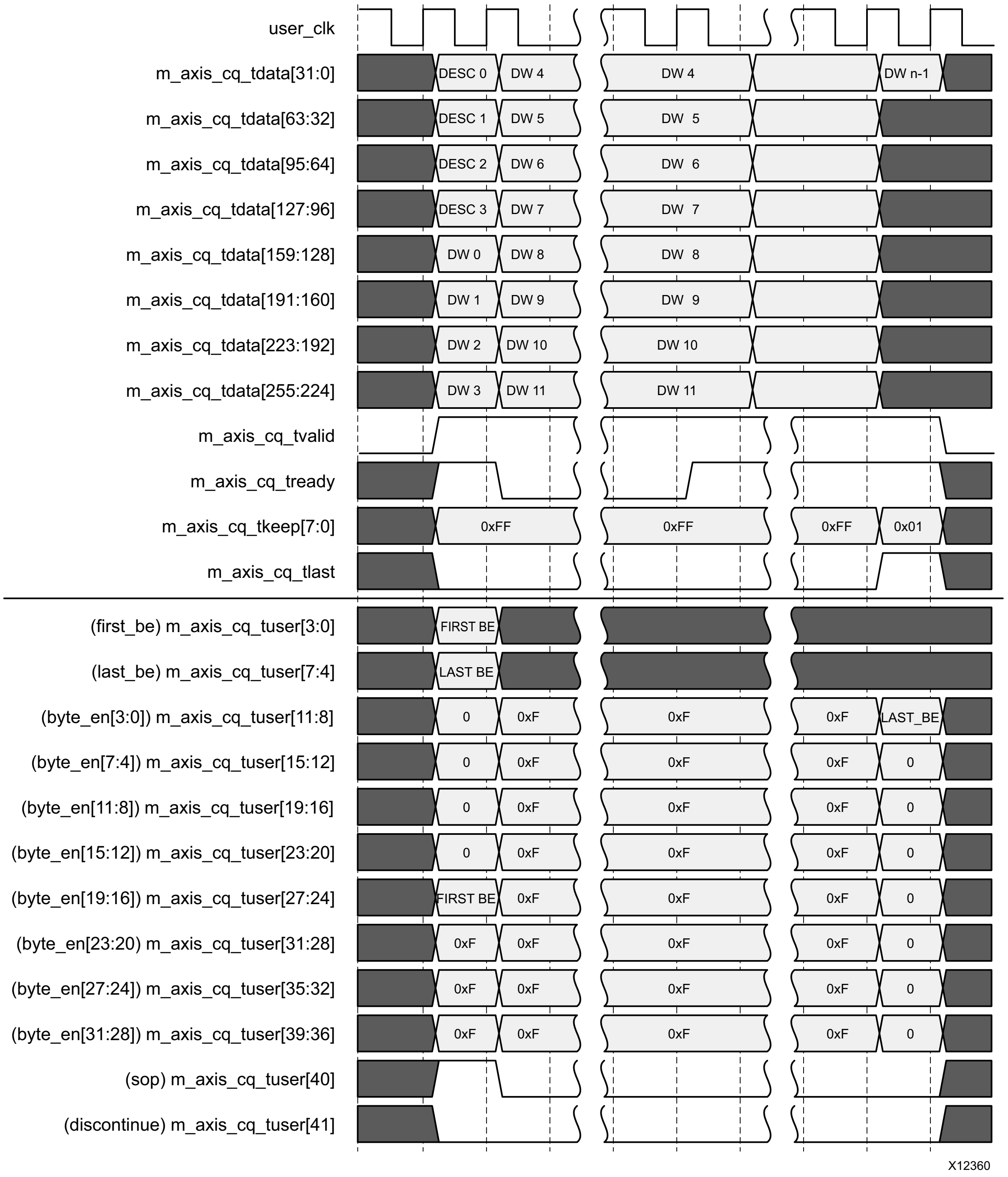

user_clk
m_axis_cq_tdata[31:0]
m_axis_cq_tdata[63:32]
m_axis_cq_tdata[95:64]
m_axis_cq_tdata[127:96]
m_axis_cq_tdata[159:128]
m_axis_cq_tdata[191:160]
m_axis_cq_tdata[223:192]
m_axis_cq_tdata[255:224]
m_axis_cq_tvalid
m_axis_cq_tready
m_axis_cq_tkeep[7:0]
m_axis_cq_tlast
(first_be) m_axis_cq_tuser[3:0]
(last_be) m_axis_cq_tuser[7:4]
(byte_en[3:0]) m_axis_cq_tuser[11:8]
(byte_en[7:4]) m_axis_cq_tuser[15:12]
(byte_en[11:8]) m_axis_cq_tuser[19:16]
(byte_en[15:12]) m_axis_cq_tuser[23:20]
(byte_en[19:16]) m_axis_cq_tuser[27:24]
(byte_en[23:20) m_axis_cq_tuser[31:28]
(byte_en[27:24]) m_axis_cq_tuser[35:32]
(byte_en[31:28]) m_axis_cq_tuser[39:36]
(sop) m_axis_cq_tuser[40]
(discontinue) m_axis_cq_tuser[41]
DESC 0
DESC 1
DESC 2
DESC 3
DW 0
DW 1
DW 2
DW 3
DW 4
DW 5
DW 6
DW 7
DW 8
DW 9
DW 10
DW 11
DW n-1
0xFF
0x01
FIRST BE
LAST BE
LAST_BE
0
0xF
X12360

Figure 3-20: **Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode, 256-Bit Interface)**

The timing diagrams in Figure 3-21, Figure 3-22, and Figure 3-23 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.

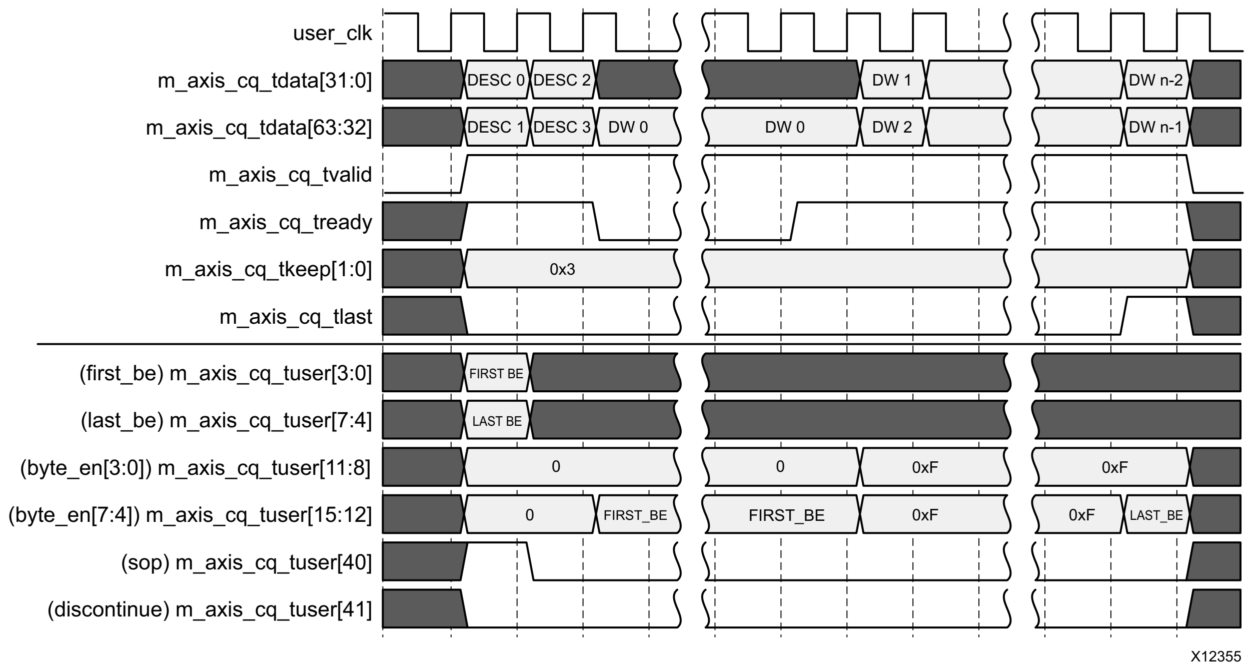


Figure 3-21: **Memory Write Transaction on the Completer Request Interface (Address-Aligned Mode, 64-Bit Interface)**

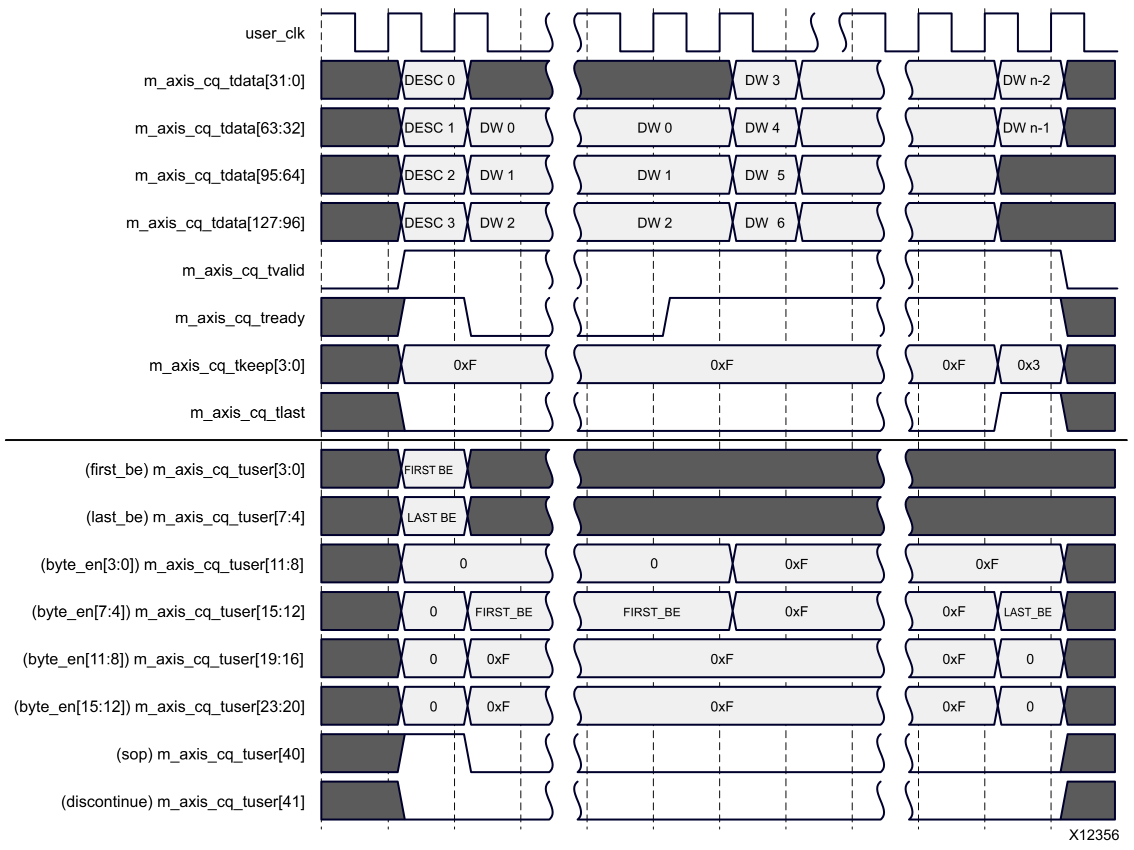


Figure 3-22: **Memory Write Transaction on the Completer Request Interface (Address-Aligned Mode, 128-Bit Interface)**

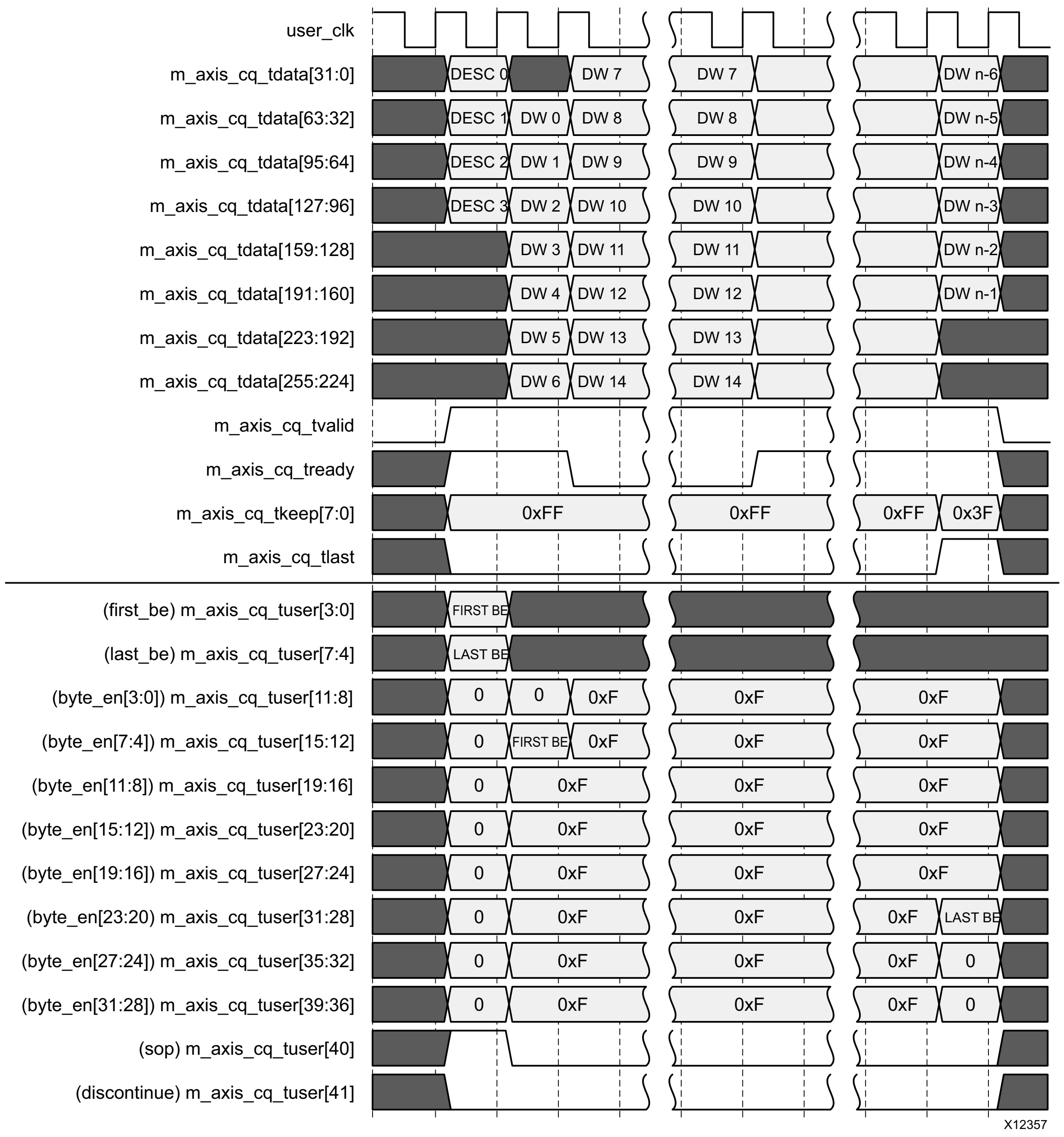

user_clk
m_axis_cq_tdata[31:0]
m_axis_cq_tdata[63:32]
m_axis_cq_tdata[95:64]
m_axis_cq_tdata[127:96]
m_axis_cq_tdata[159:128]
m_axis_cq_tdata[191:160]
m_axis_cq_tdata[223:192]
m_axis_cq_tdata[255:224]
m_axis_cq_tvalid
m_axis_cq_tready
m_axis_cq_tkeep[7:0]
m_axis_cq_tlast
(first_be) m_axis_cq_tuser[3:0]
(last_be) m_axis_cq_tuser[7:4]
(byte_en[3:0]) m_axis_cq_tuser[11:8]
(byte_en[7:4]) m_axis_cq_tuser[15:12]
(byte_en[11:8]) m_axis_cq_tuser[19:16]
(byte_en[15:12]) m_axis_cq_tuser[23:20]
(byte_en[19:16]) m_axis_cq_tuser[27:24]
(byte_en[23:20) m_axis_cq_tuser[31:28]
(byte_en[27:24]) m_axis_cq_tuser[35:32]
(byte_en[31:28]) m_axis_cq_tuser[39:36]
(sop) m_axis_cq_tuser[40]
(discontinue) m_axis_cq_tuser[41]
DESC 0
DESC 1
DESC 2
DESC 3
DW 0
DW 1
DW 2
DW 3
DW 4
DW 5
DW 6
DW 7
DW 8
DW 9
DW 10
DW 11
DW 12
DW 13
DW 14
DW n-6
DW n-5
DW n-4
DW n-3
DW n-2
DW n-1
0xFF
0x3F
FIRST BE
LAST BE
0
0xF
X12357

Figure 3-23: **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 timing diagrams in Figure 3-24, Figure 3-25, and Figure 3-26 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.

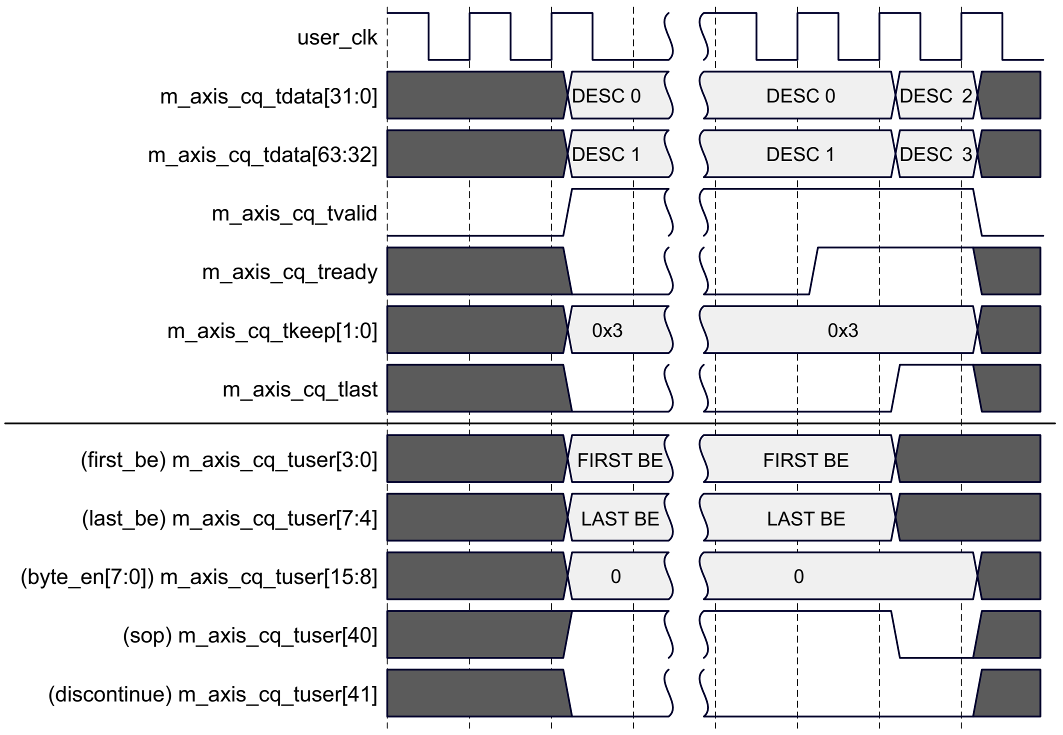


Figure 3-24: **Memory Read Transaction on the Completer Request Interface (64-Bit Interface)**

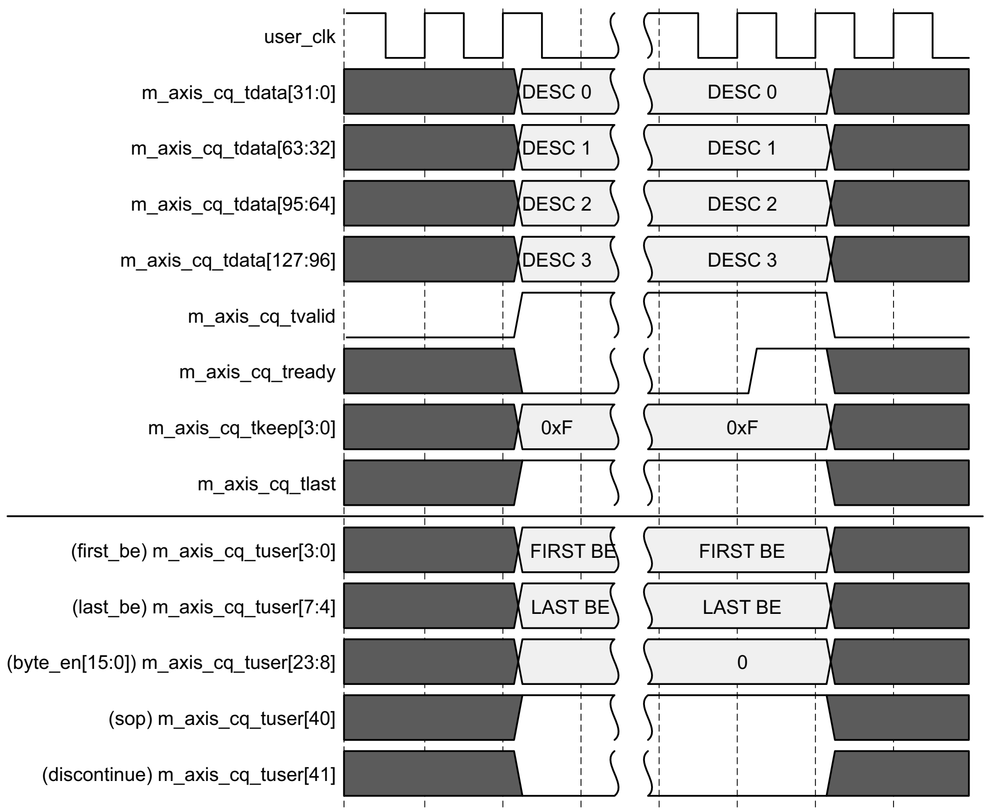


Figure 3-25: **Memory Read Transaction on the Completer Request Interface (128-Bit Interface)**

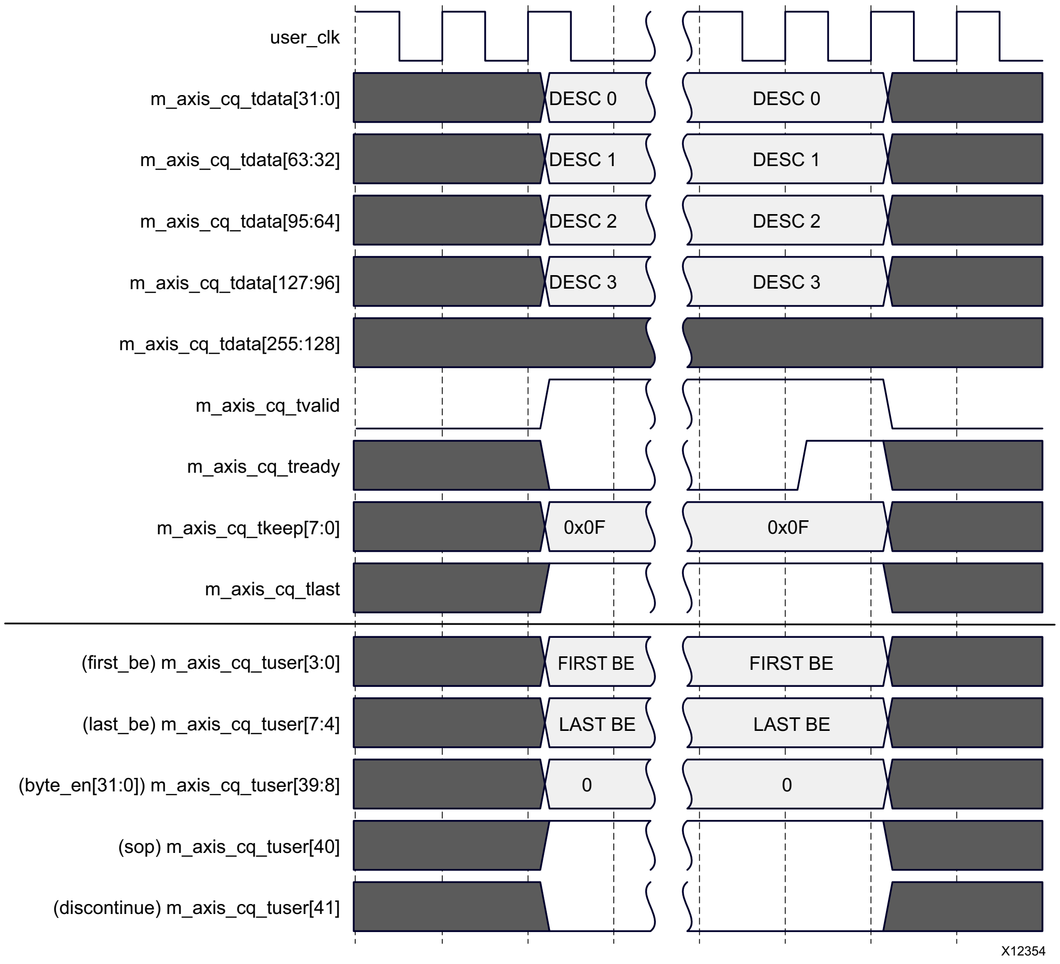


Figure 3-26: **Memory Read Transaction on the Completer Request Interface (256-Bit Interface)**

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, page 141.

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, page 141.

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, page 141.

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 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 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, page 141.

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* [Ref 2] 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

Figure 3-27 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.

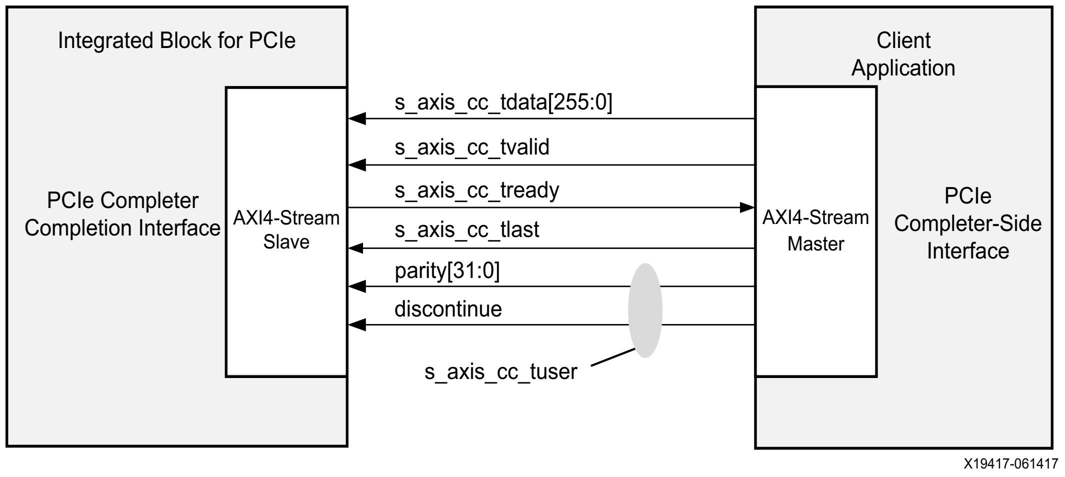


Figure 3-27: **Completer Completion Interface Signals**

The core delivers each TLP on the Completer Completion (CC) 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 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 mod 4) mod 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 mod 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 Figure 3-28. The individual fields of the completer request descriptor are described in Table 3-9.

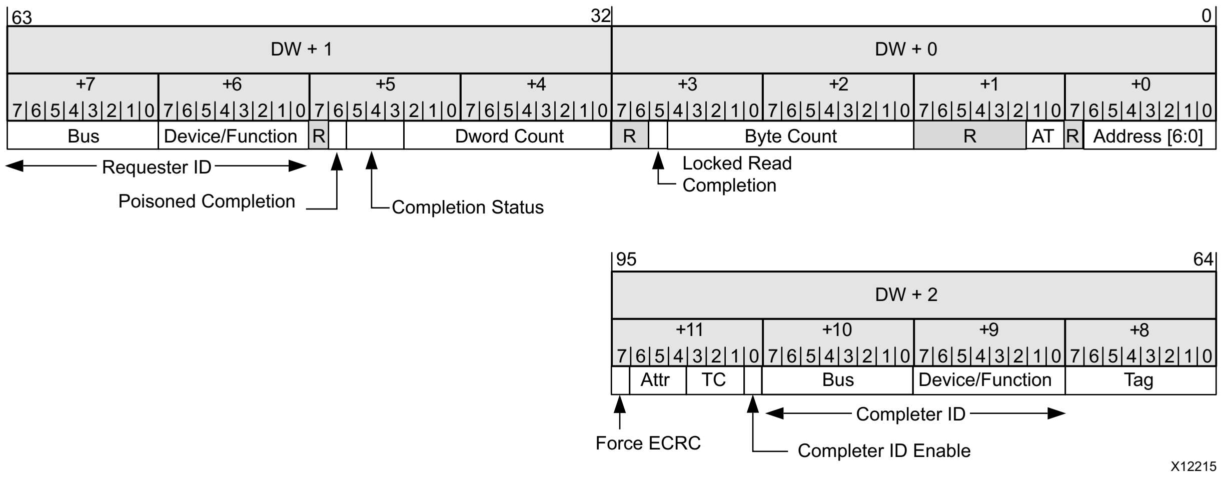


Figure 3-28: **Completer Completion Descriptor Format**

Table 3-9: **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'bx100 → 2'b10; 4'b1000 → 2'b11 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.

first_be[3:0]	Lower Address[1:0]
4'b0000	2'b00
4'bxxx1	2'b00
4'bxx10	2'b01
4'bx100	2'b10
4'b1000	2'b11

Table 3-9: **Completer Completion Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
28:16	Byte Count	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. 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. The total number of bytes required to complete a Memory Read Request is calculated as shown in Table 3-10, page 146.
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	These bits must be set based on the type of Completion being sent. The only valid settings are: • 000: Successful Completion • 001: Unsupported Request (UR) • 100: Completer Abort (CA)
46	Poisoned Completion	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	PCIe Tag associated with the request (copied from the request).

Table 3-9: **Completer Completion Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
79:72	Target Function/ Device Number	Function number of the completer Function. The user application must always supply the function number.When ARI is in use, all 8 bits of this field must be set to the target Function number. Otherwise, bits [74:72] must be set to the target Function number. The user application must copy this value from the Target Function field of the descriptor of the corresponding request. Otherwise, bits [74:72] must be set to the target Function number. When ARI is not in use, and the integrated block is configured as a Root Complex, the user application must supply the 5-bit Device Number of the completer on bits [79:75]. When ARI is not used and the integrated block is configured as an Endpoint, the user application can optionally supply a 5-bit Device Number of the completer on bits [79:75]. The user application must set the Completer ID Enable bit in the descriptor if a Device Number is supplied on bits [79:75]. This value is used by the integrated block when sending the Completion TLP, instead of the stored value of the Device Number captured by the integrated block from Configuration Requests.
87:80	Completer Bus Number	Bus number associated with the completer Function. When the integrated block is configured as a Root Complex, the user application must supply the 8-bit Bus Number of the completer in this field. When the integrated block is configured as an Endpoint, the user application can optionally supply a Bus Number in this field. The user application must set the Completer ID Enable bit in the descriptor if a Bus Number is supplied in this field. This value is used by the integrated block when sending the Completion TLP, instead of the stored value of the Bus Number captured by the integrated block from Configuration Requests.
88	Completer ID Enable	The purpose of this field is to enable the user application to supply the bus and device numbers to be used in the Completer ID. This field is applicable only to Endpoint configurations. If this field is 0, the integrated block uses the captured values of the bus and device numbers to form the Completer ID. If this input is 1, the integrated block uses the bus and device numbers supplied in the descriptor to form the Completer ID.
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 3-10: **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
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 timing diagrams in Figure 3-29, Figure 3-30, and Figure 3-31 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 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 `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`.

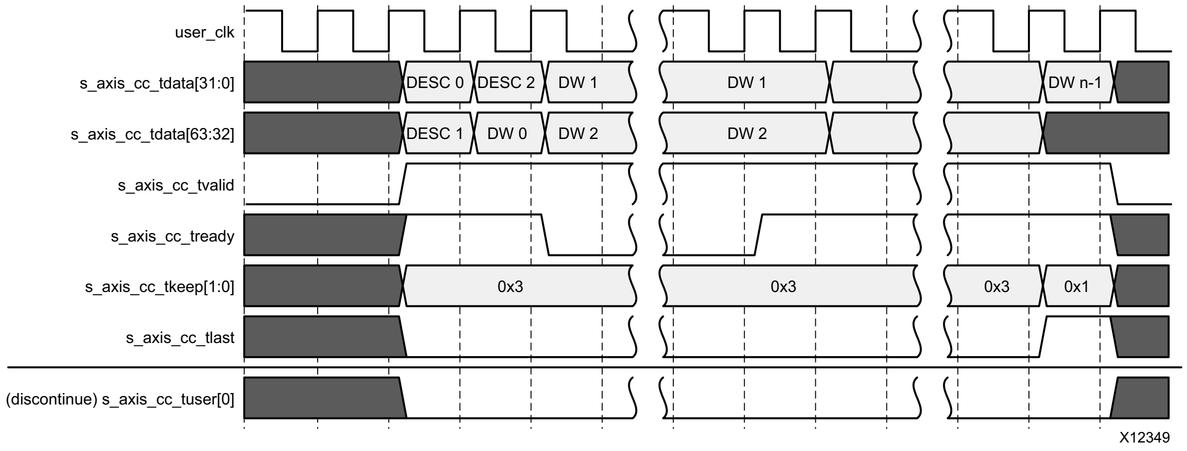


Figure 3-29: **Transfer of a Normal Completion on the Completer Completion Interface (Dword-Aligned Mode, 64-Bit Interface)**

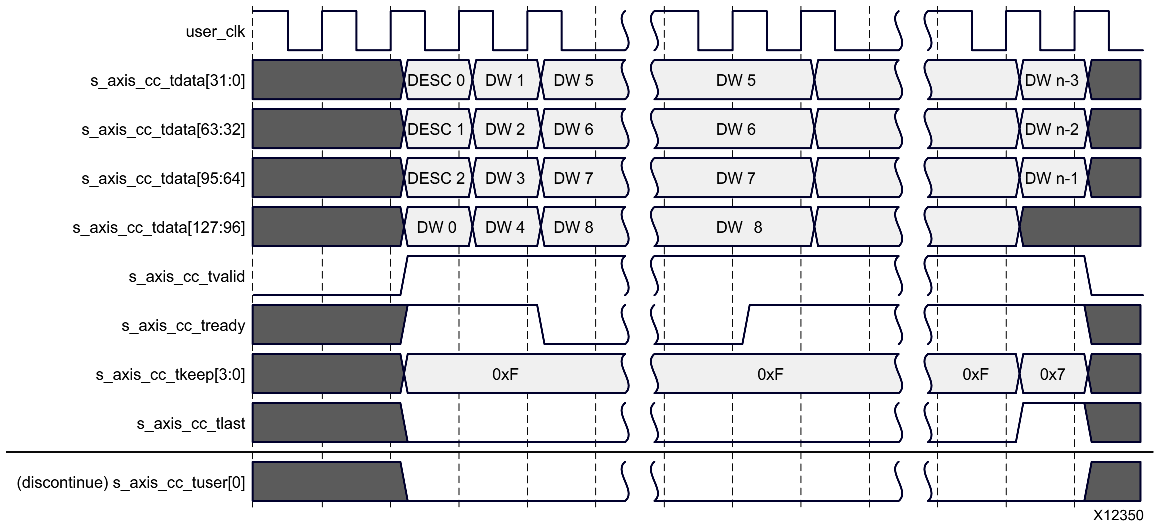


Figure 3-30: **Transfer of a Normal Completion on the Completer Completion Interface (Dword-Aligned Mode, 128-Bit Interface)**

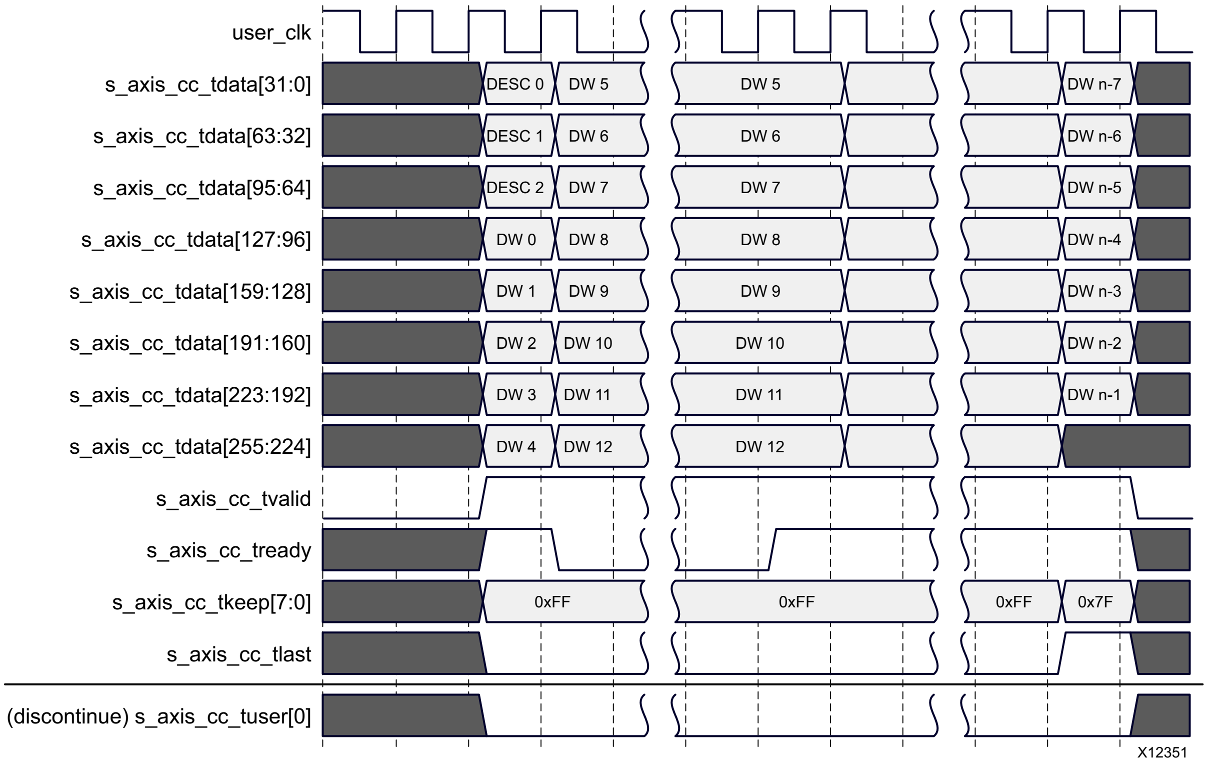


Figure 3-31: **Transfer of a Normal Completion on the Completer Completion Interface (Dword-Aligned Mode, 256-Bit Interface)**

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 timing diagrams in Figure 3-32, Figure 3-33, and Figure 3-34 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 × 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.

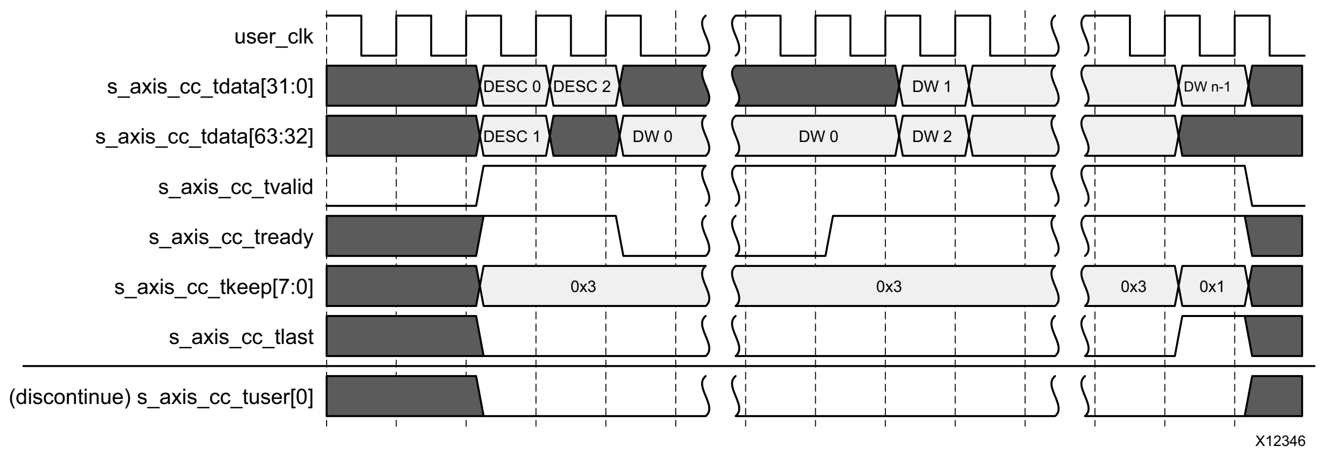


Figure 3-32: **Transfer of a Normal Completion on the Completer Completion Interface (Address-Aligned Mode, 64-Bit Interface)**

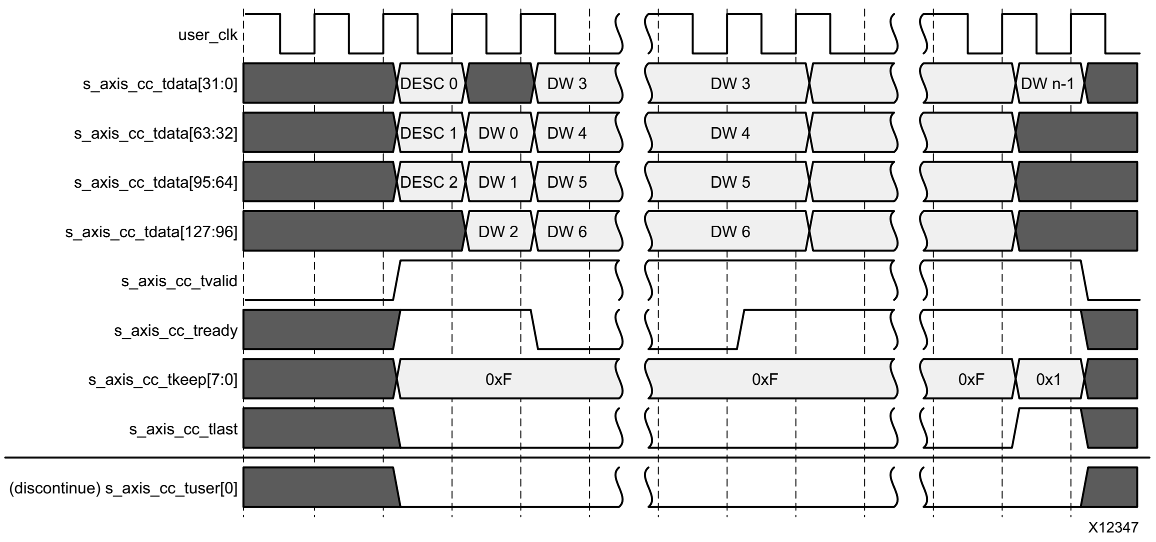


Figure 3-33: **Transfer of a Normal Completion on the Completer Completion Interface (Address-Aligned Mode, 128-Bit Interface)**

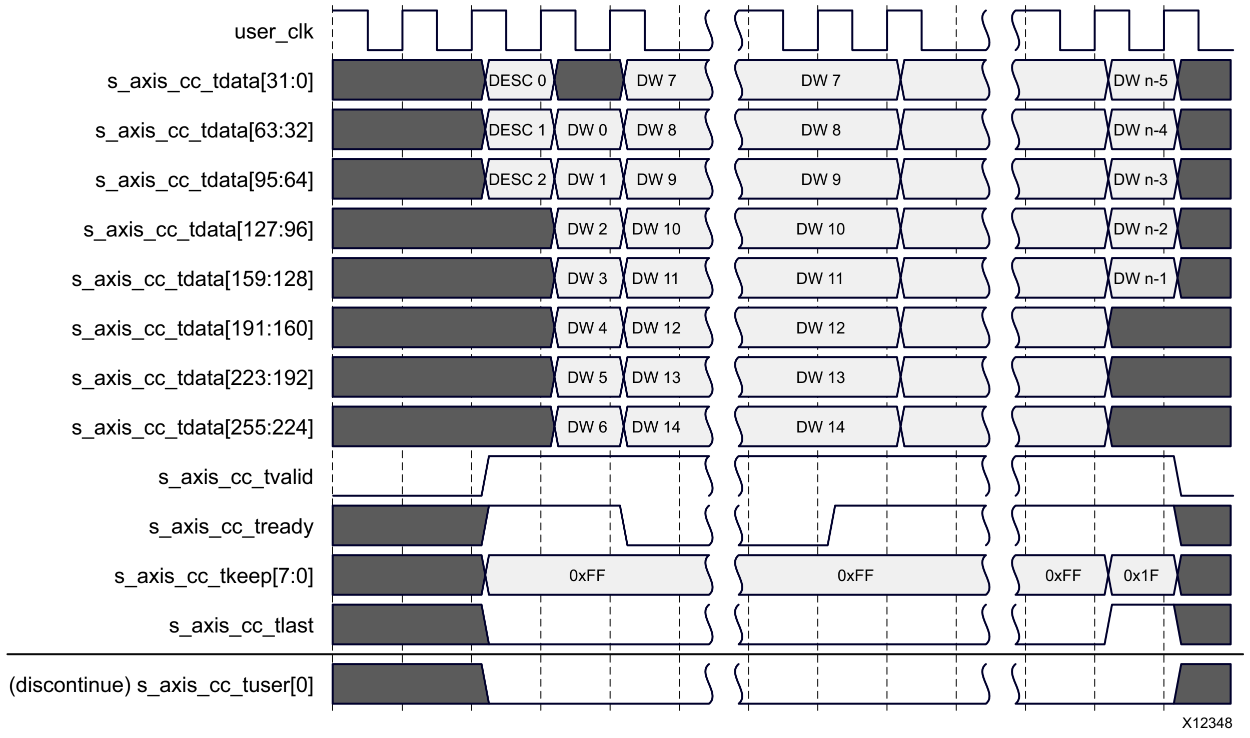


Figure 3-34: **Transfer of a Normal Completion on the Completer Completion Interface (Address-Aligned Mode, 256-Bit Interface)**

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 Figure 3-28, 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.

Figure 3-35 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 Figure 3-28.
- 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 integrated block with the request.

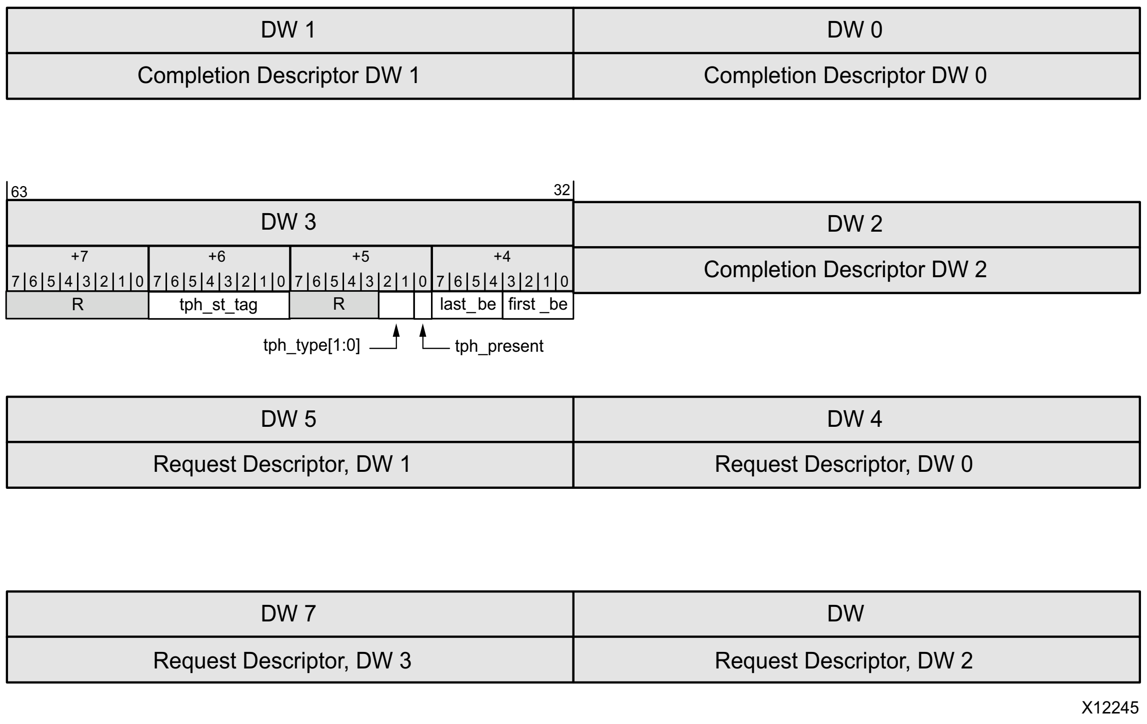


Figure 3-35: **Composition of the AXI4-Stream Packet for UR and CA Completions**

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. Figure 3-36 shows the signals associated with the requester request interface.

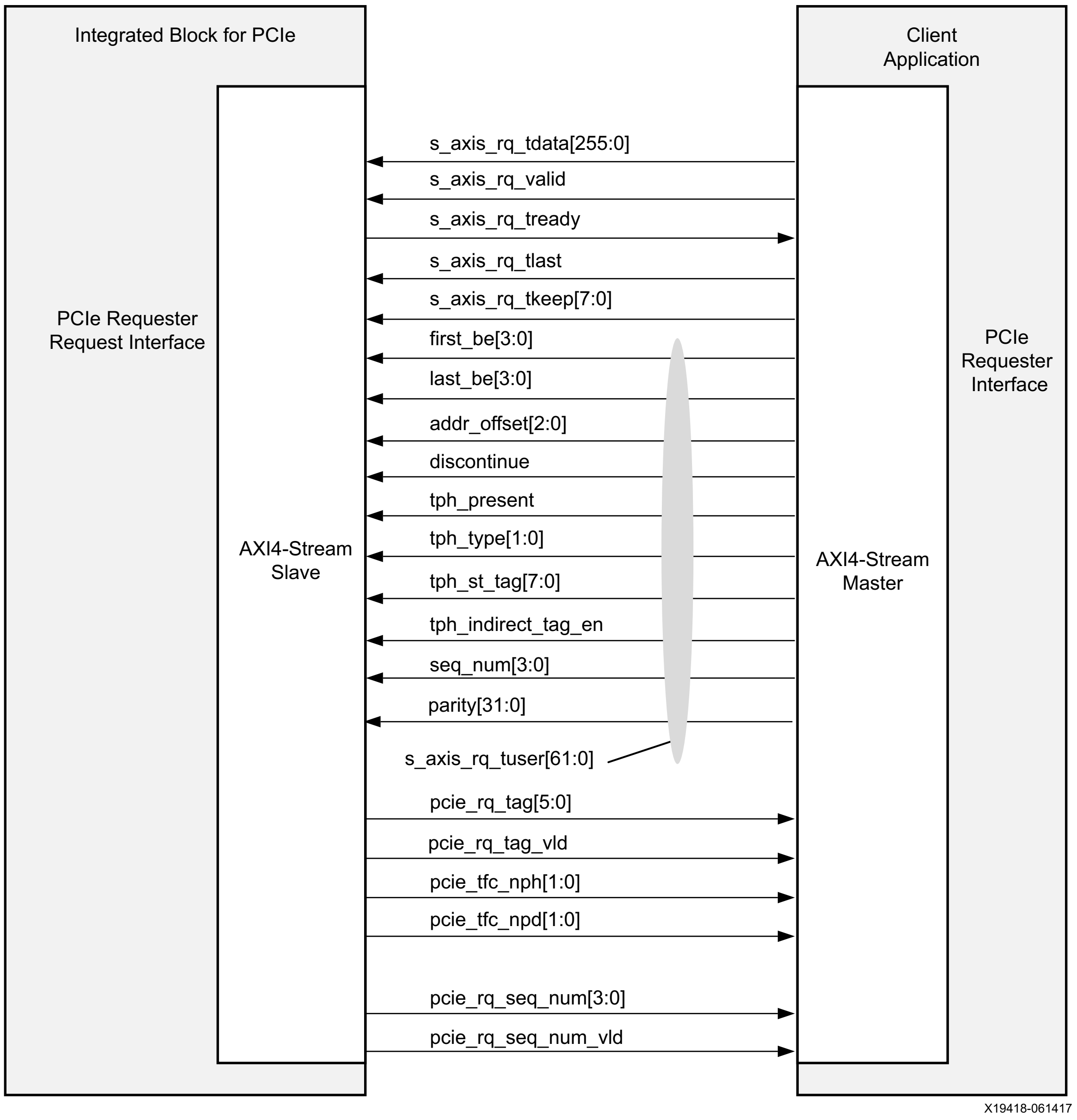


Figure 3-36: **Requester Request Interface**

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 Figure 3-37 through Figure 3-41. The format of Figure 3-37 applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request. The format in Figure 3-38 is used for Configuration Requests. The format in Figure 3-39 is used for Vendor-Defined Messages (Type 0 or Type 1) only. The format in Figure 3-40 is used for all ATS messages (Invalid Request, Invalid Completion, Page Request, PRG Response). For all other messages, the descriptor takes the format shown in Figure 3-41.

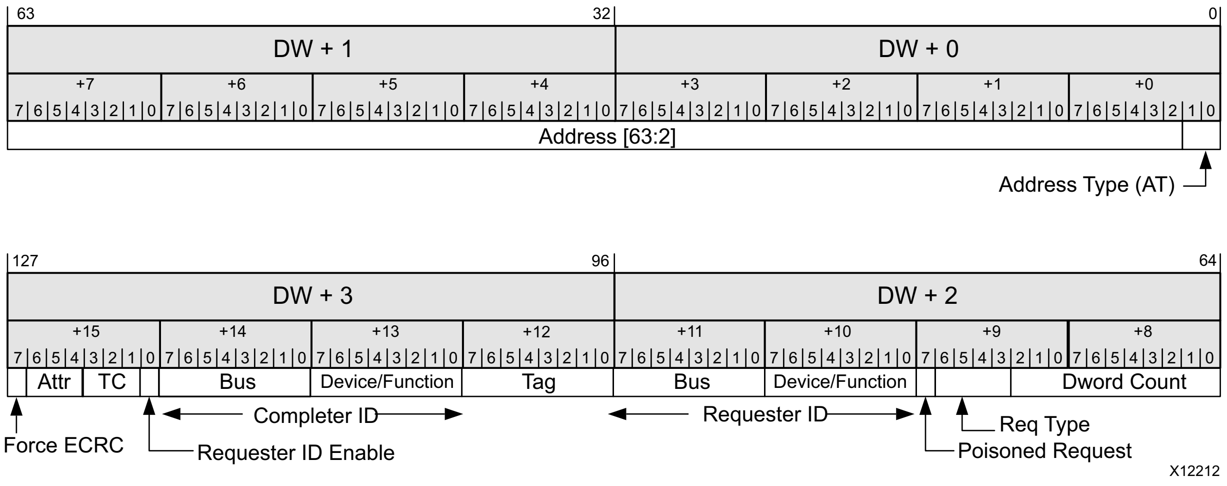


Figure 3-37: **Requester Request Descriptor Format for Memory, I/O, and Atomic Op Requests**

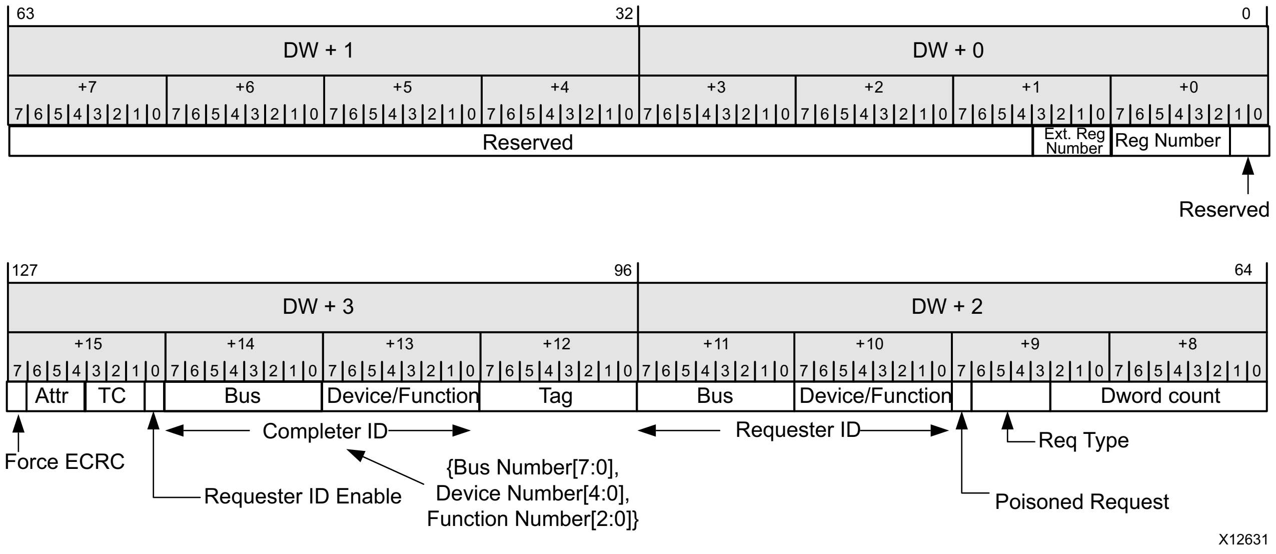


Figure 3-38: **Requester Request Descriptor Format for Configuration Requests**

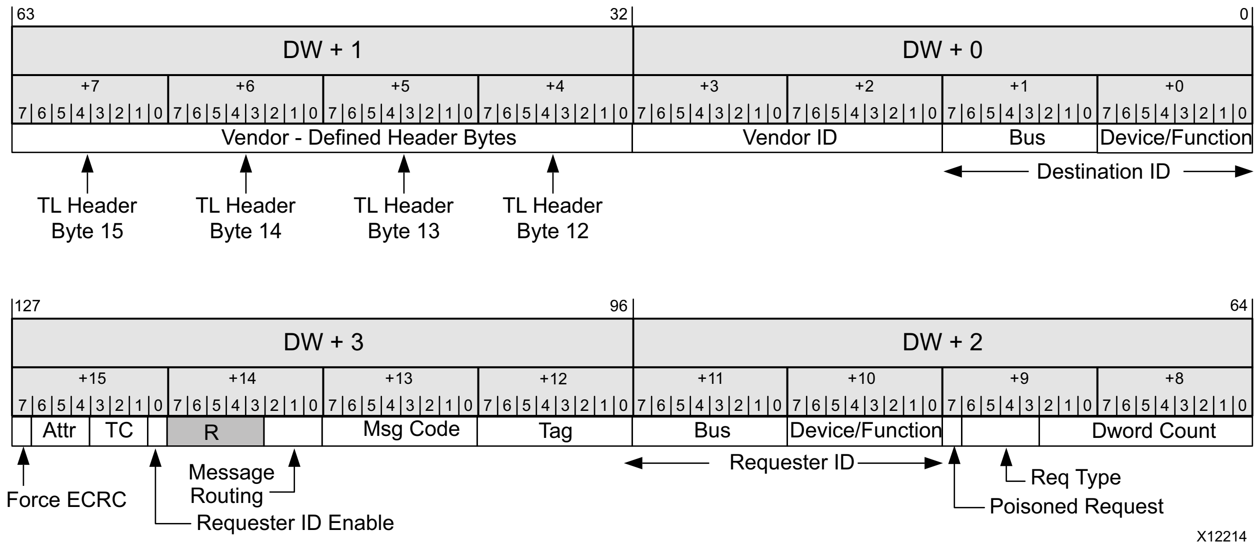


Figure 3-39: **Requester Request Descriptor Format for Vendor-Defined Messages**

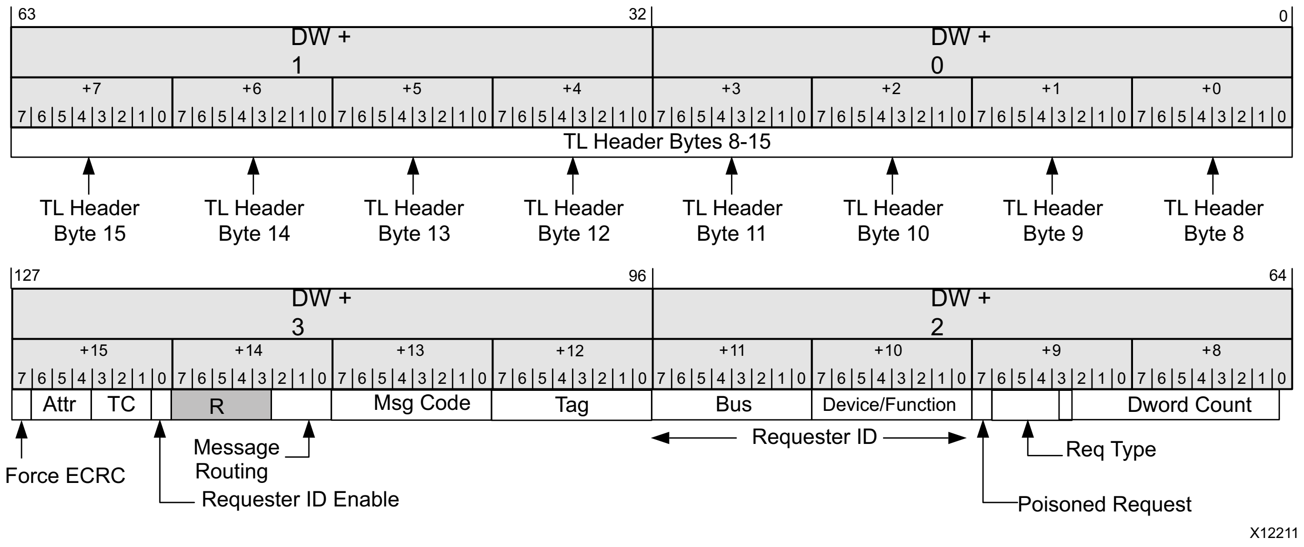


Figure 3-40: **Requester Request Descriptor Format for ATS Messages**

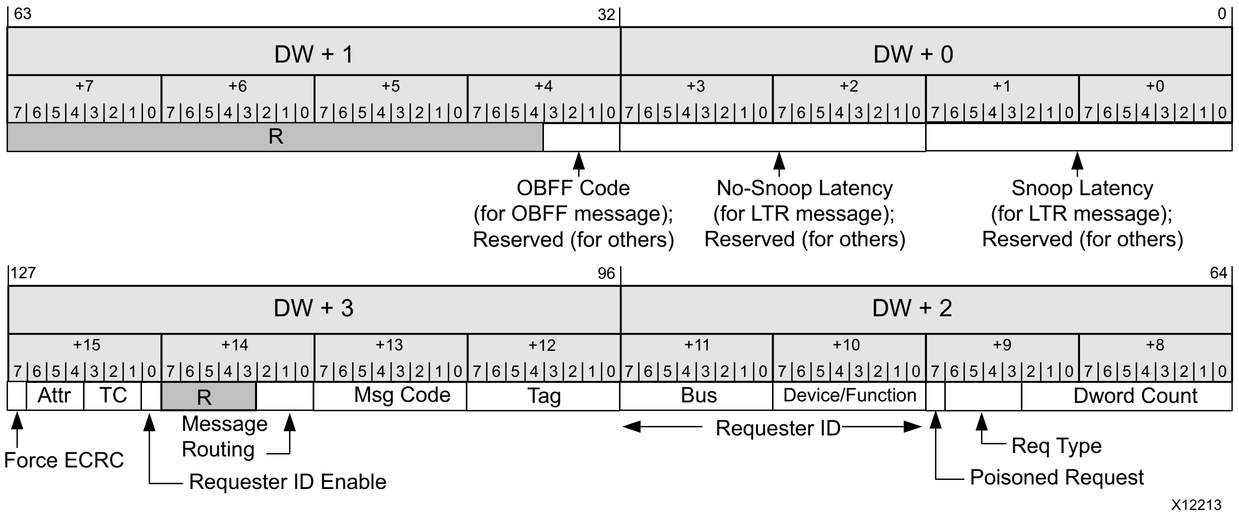


Figure 3-41: **Requester Request Descriptor Format for all other Messages**

Table 3-11 describes the individual fields of the completer request descriptor.

Table 3-11: **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 integrated block copies this field into the AT of the TL header of the request TLP. • 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. 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. 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). 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. For a zero length memory read/write request, the Dword count must be 1, with the First_BE bits set to all zeros. 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.

Table 3-11: **Requester Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
78:75	Request Type	Identifies the transaction type. The transaction types and their encodings are listed in Table 3-8.
79	Poisoned Request	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. This feature is supported on all request types except Type 0 and Type 1 Configuration Write Requests.
87:80	Requester Function/ Device Number	Function number of the Requester Function. When ARI is in use, all 8 bits of the field must be set to the Function number. Otherwise bits [84:82] must be set to the completer Function number. When ARI is not in use and the core is configured as a Root Complex,client must supply the 5 bit device number of the requester on bits [87:83] When ARI is not in use and the core is configured as an EndPoint, the client must optionally supply a 5 bit device number on bits [87:83]. The client must set the Requester ID Enable but in the descriptor if a Device Number is supplied on bits [87:83]. This value must be used by the core when sending the Request TLP, instead of stored value of the Device number captured by the core from configuration requests
95:88	Requester Bus Number	Bus number associated with the Requester Function. When the integrated block is configured as an Endpoint, the user application must set the Requester ID Enable bit in the descriptor to 0b and supply 8'b0 in this field. In this case, the stored value of the Bus Number captured by the integrated block from Configuration Requests is used. Otherwise, when the integrated block is configured as a Root Complex, the user application must set the Requester ID Enable bit in the descriptor to 1b and supply the 8-bit Bus Number of the requester in this field.
103:96	Tag	PCIe Tag associated with the request. 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.

Table 3-11: **Requester Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
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.).
120	Requester ID Enable	The purpose of this field is to enable the user application to supply the bus, device and function numbers to be used in the Requester ID. This field must be set to 1b when the integrated block is configured as a Root Complex and must be set to 0b when the integrated block is configured as an Endpoint. See the Requester Bus Number and Requester Function/Device Number fields for further requirements.
123:121	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request.
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. 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.
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	The OBFF Code field is used to distinguish between various OBFF cases: • 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.

Table 3-11: **Requester Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
111:104	Message Code	This field is defined for all messages. It contains the 8-bit Message Code to be set in the TL header. Appendix F of the *PCI Express Base Specification, rev. 3.0* [Ref 2] provides a complete list of the supported Message Codes.
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 Dwords 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 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 integrated block, 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` 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. In all other cases, the 1 bits in `first_be` and `last_be` must be contiguous.

The timing diagrams in Figure 3-42, Figure 3-43, and Figure 3-44 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* Dwords, for some $n = k \times 32 + 29$, $k > 0$.

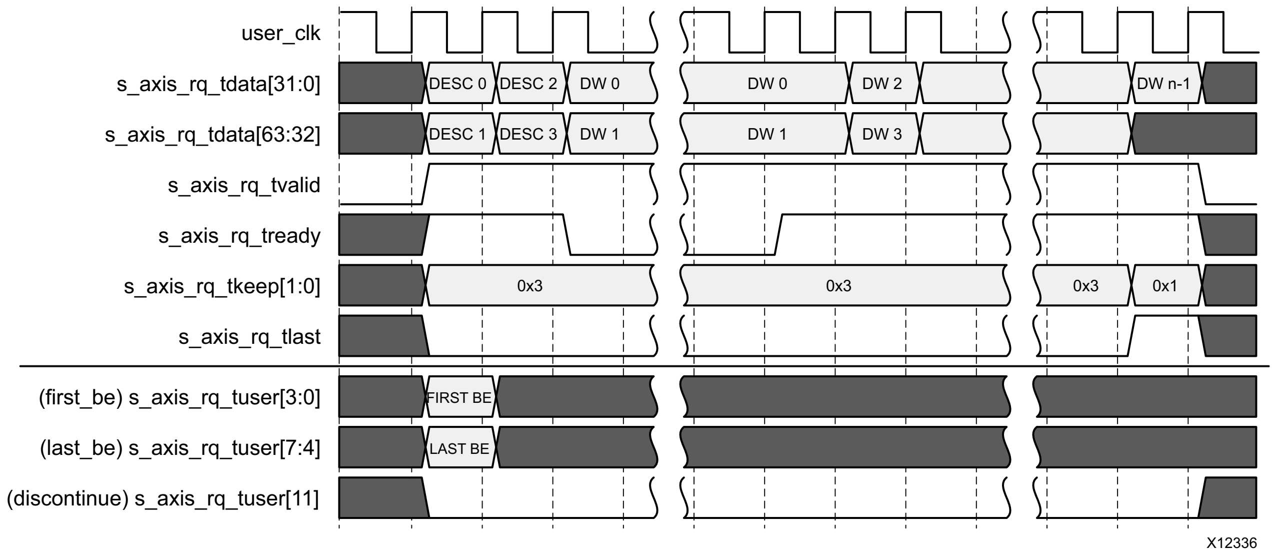


Figure 3-42: **Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, 64-Bit Interface)**

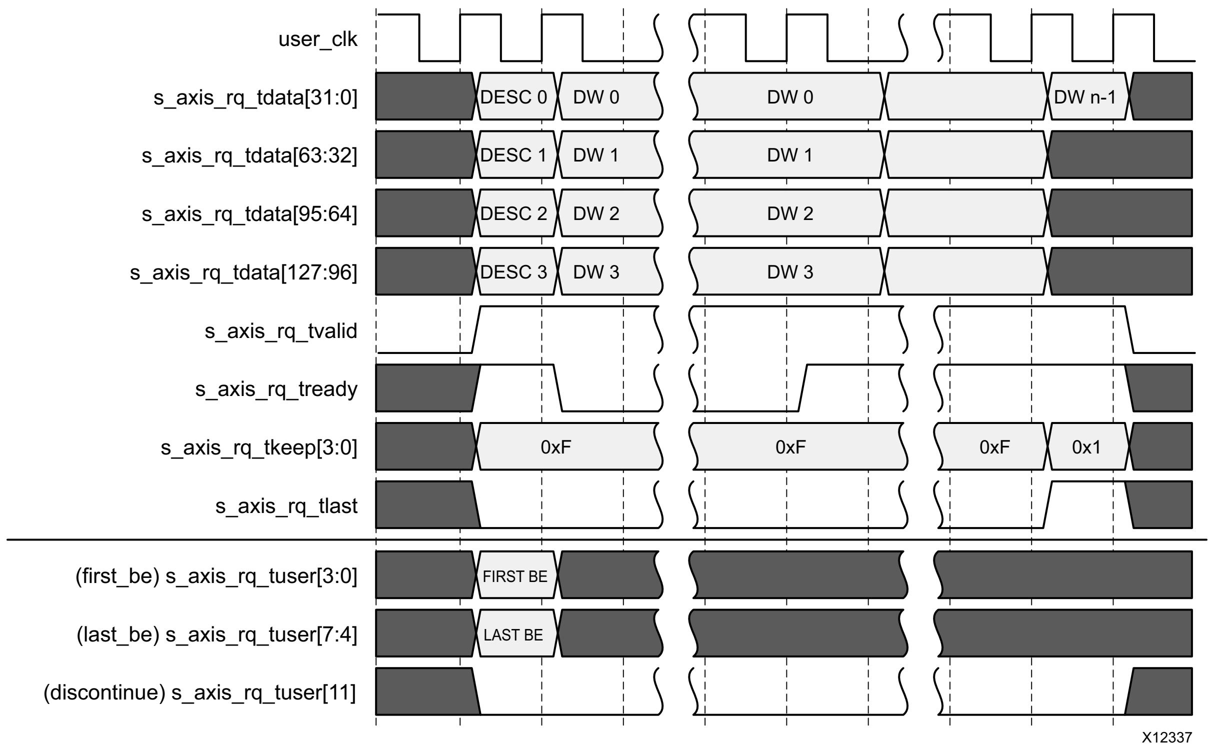


Figure 3-43: **Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, 128-Bit Interface)**

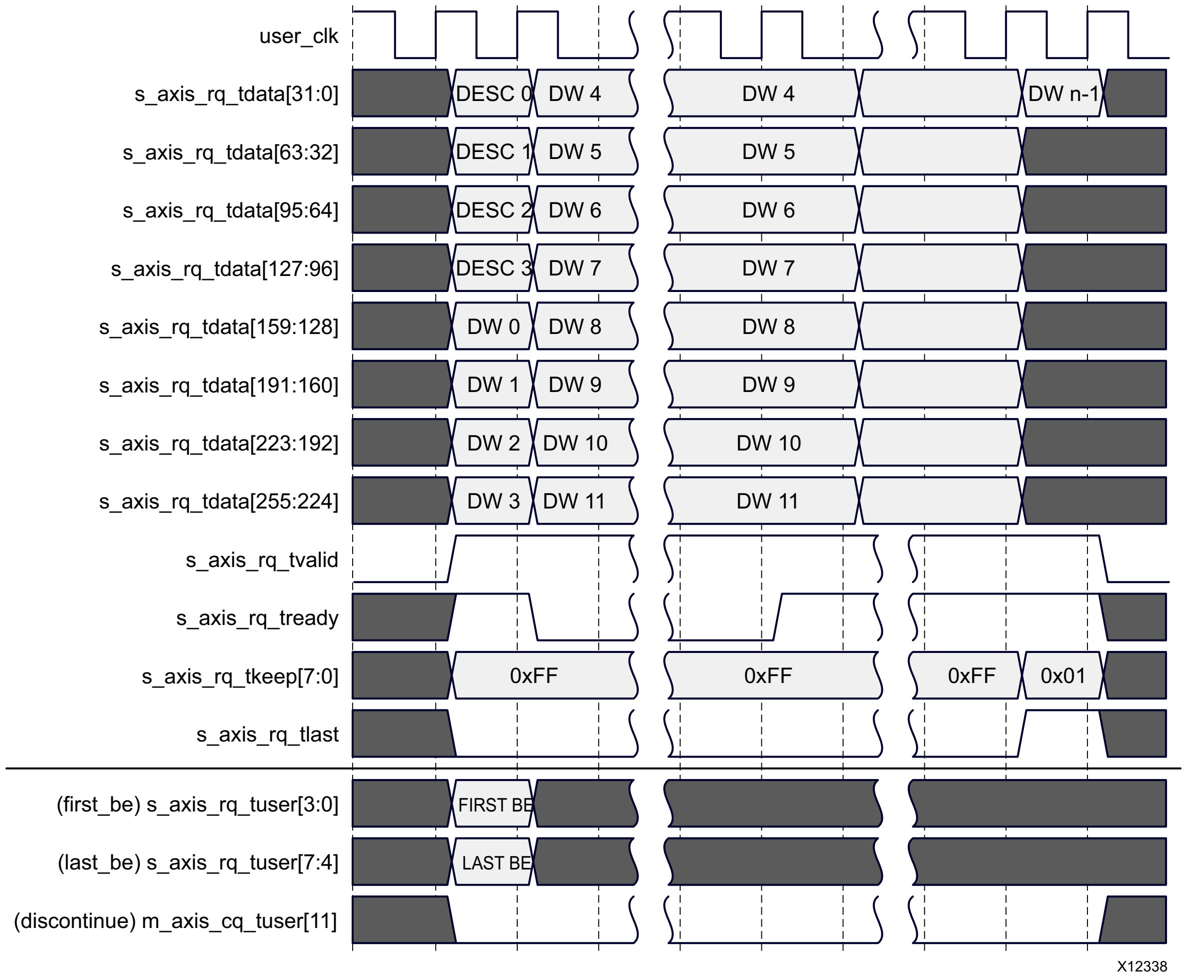


Figure 3-44: **Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, 256-Bit Interface)**

The timing diagrams in Figure 3-45, Figure 3-46, and Figure 3-47 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.

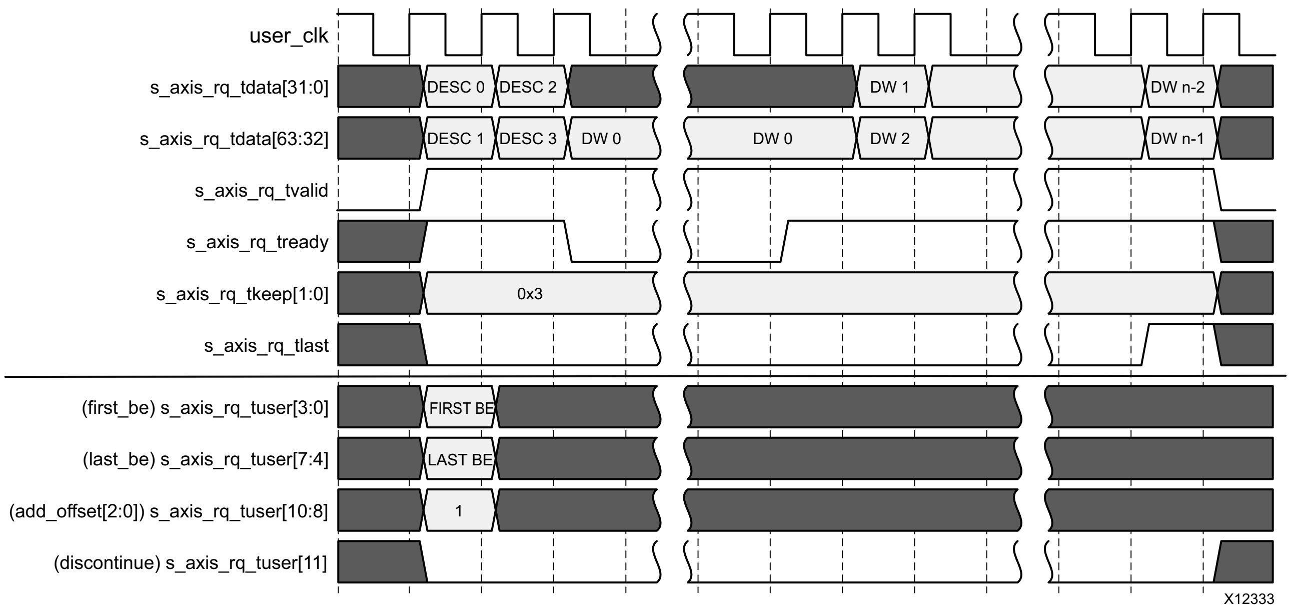


Figure 3-45: **Memory Write Transaction on the Requester Request Interface (Address-Aligned Mode, 64-Bit Interface)**

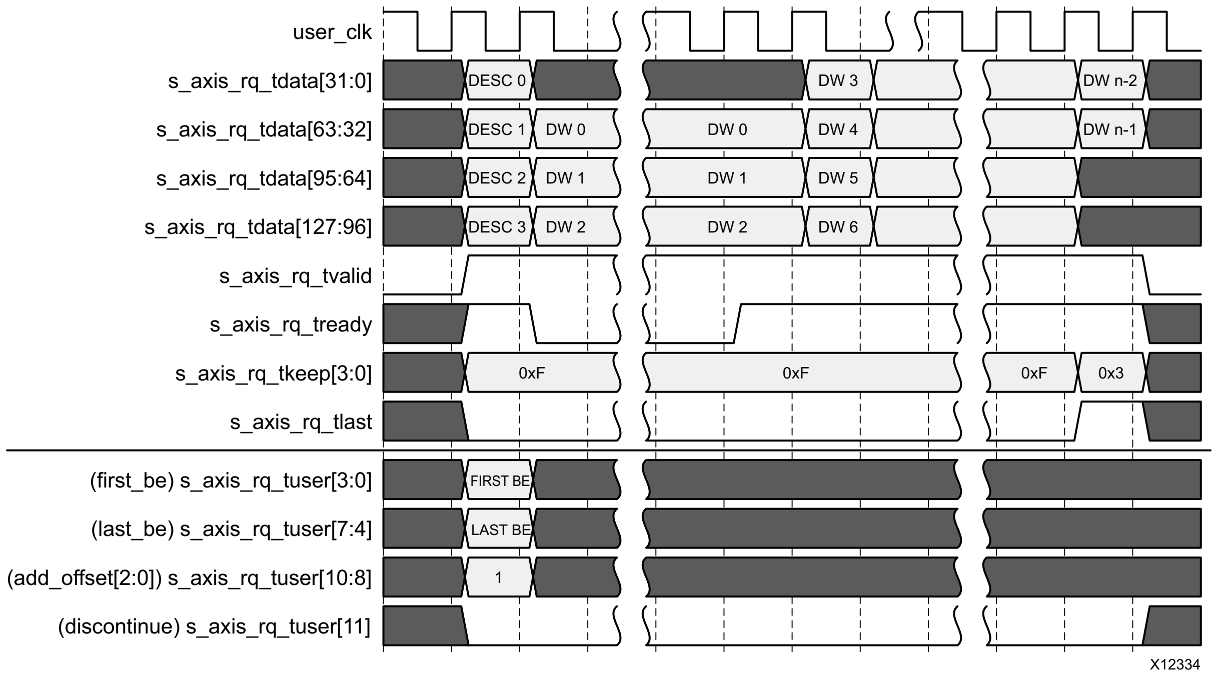


Figure 3-46: **Memory Write Transaction on the Requester Request Interface (Address-Aligned Mode, 128-Bit Interface)**

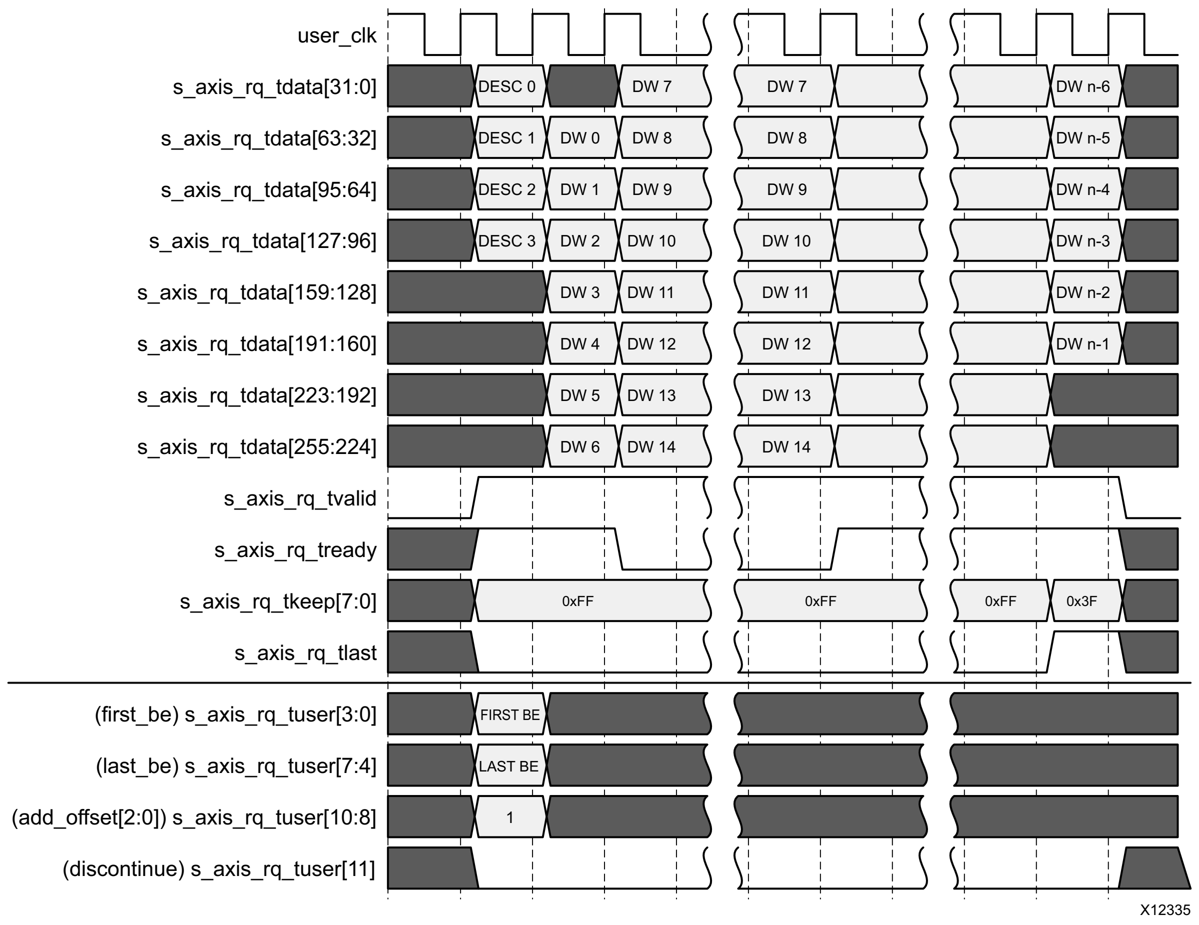


Figure 3-47: **Memory Write Transaction on the Requester Request Interface (Address-Aligned Mode, 256-Bit Interface)**

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 timing diagrams in Figure 3-48, Figure 3-49, and Figure 3-50 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.

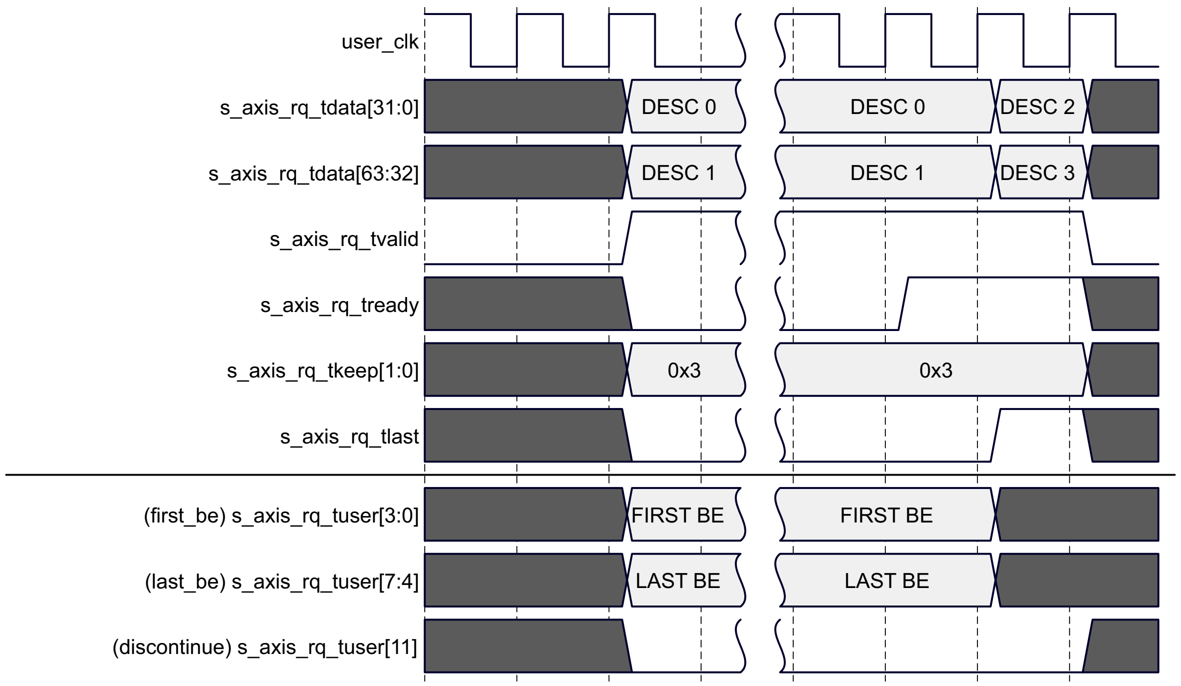


Figure 3-48: **Memory Read Transaction on the Requester Request Interface (64-Bit Interface)**

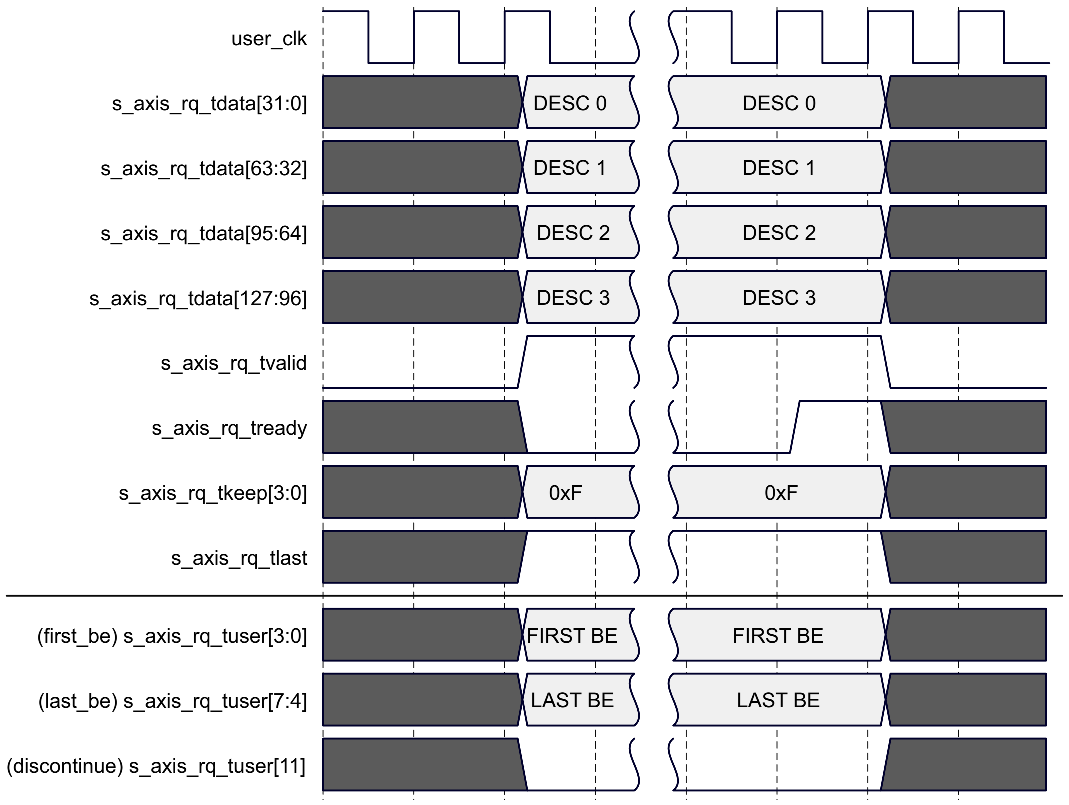


Figure 3-49: **Memory Read Transaction on the Requester Request Interface (128-Bit Interface)**

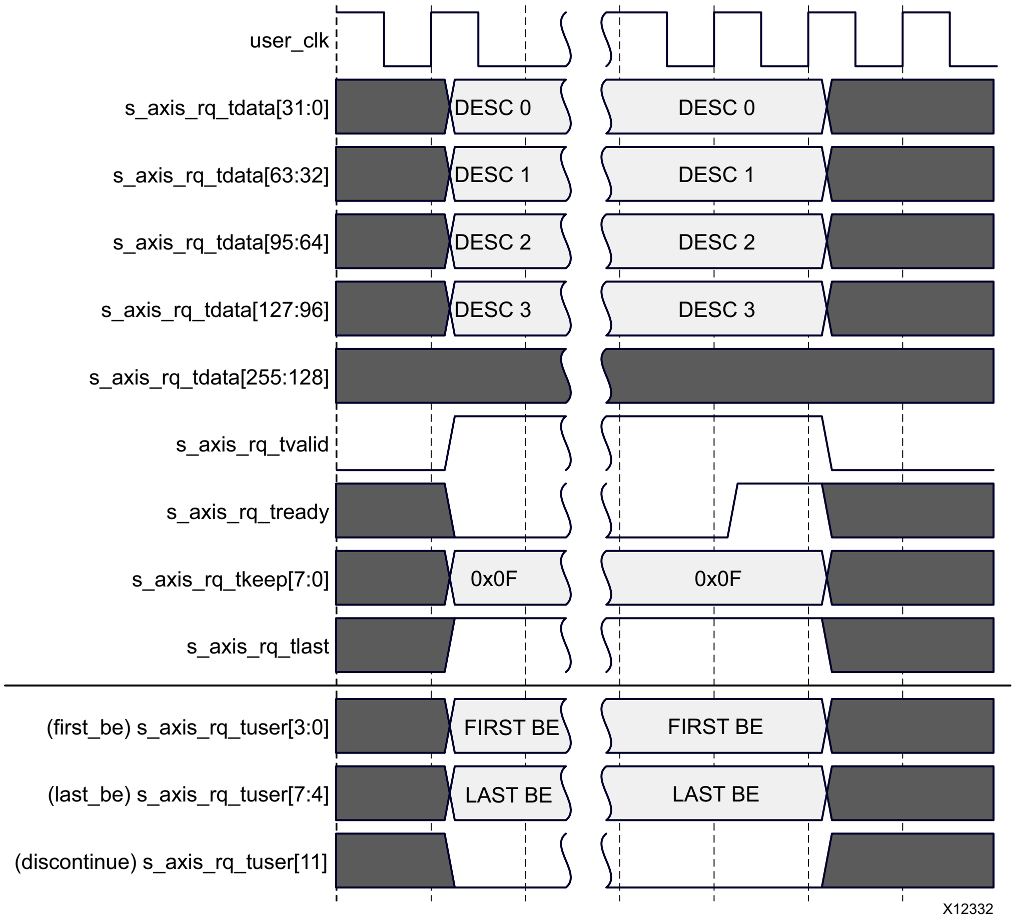


Figure 3-50: **Memory Read Transaction on the Requester Request Interface (256-Bit Interface)**

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 a 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** option 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[5:0]. 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. Figure 3-51 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 3-52 and Figure 3-53 illustrate the timing of the credit and tag available signals for the same example, for interface width of 128 bits and 64 bits, respectively.

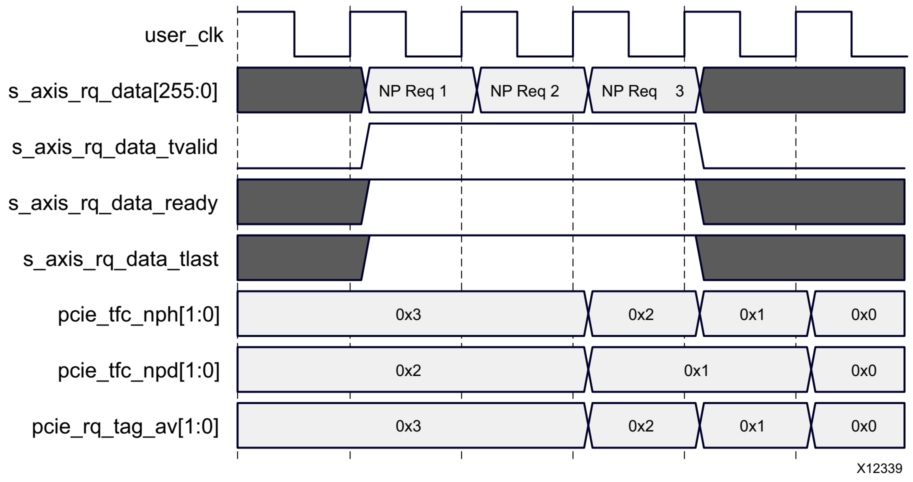


Figure 3-51: **Credit and Tag Availability Signals on the Requester Request Interface (256-Bit Interface)**

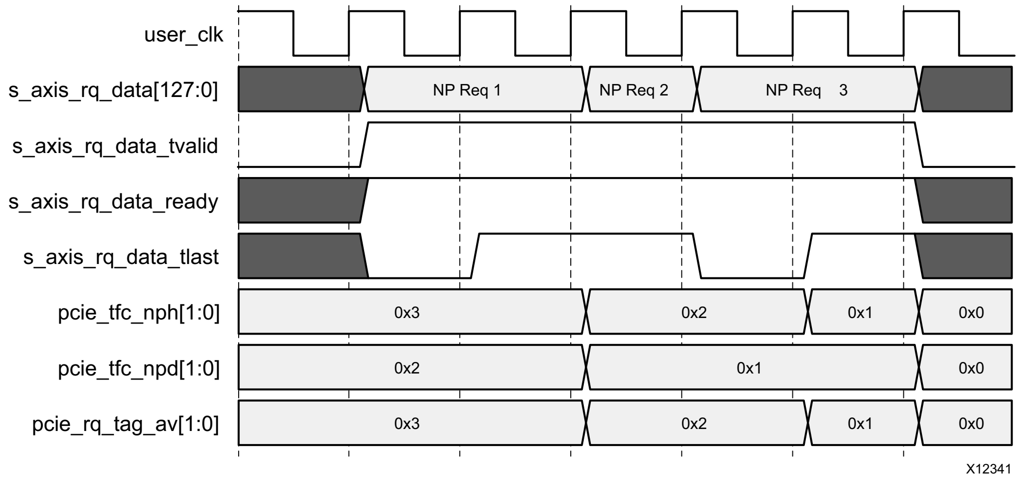


Figure 3-52: **Credit and Tag Availability Signals on the Requester Request Interface (128-Bit Interface)**

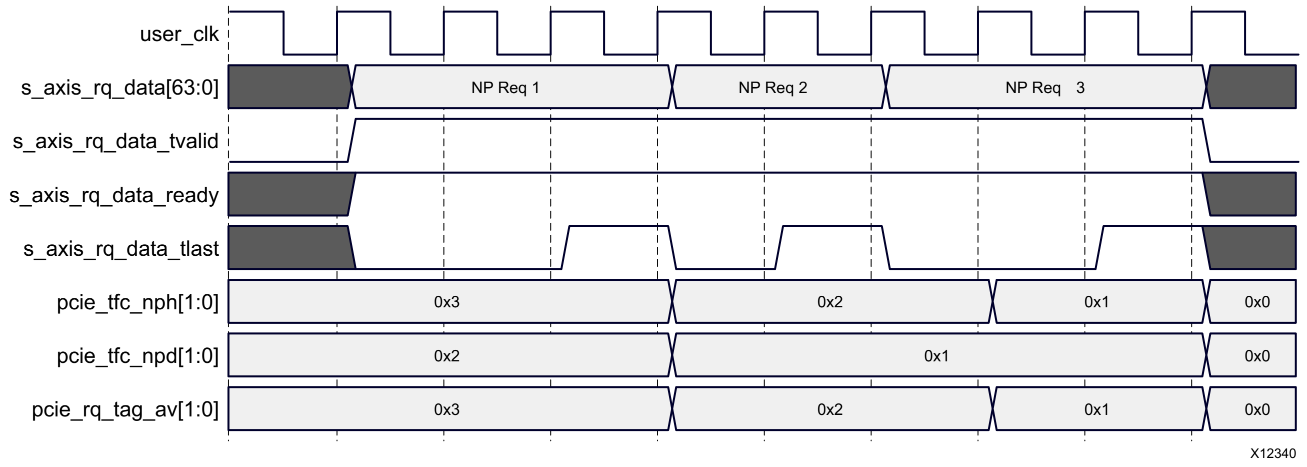


Figure 3-53: **Credit and Tag Availability Signals on the Requester Request Interface (64-Bit Interface)**

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 Figure 3-54 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.

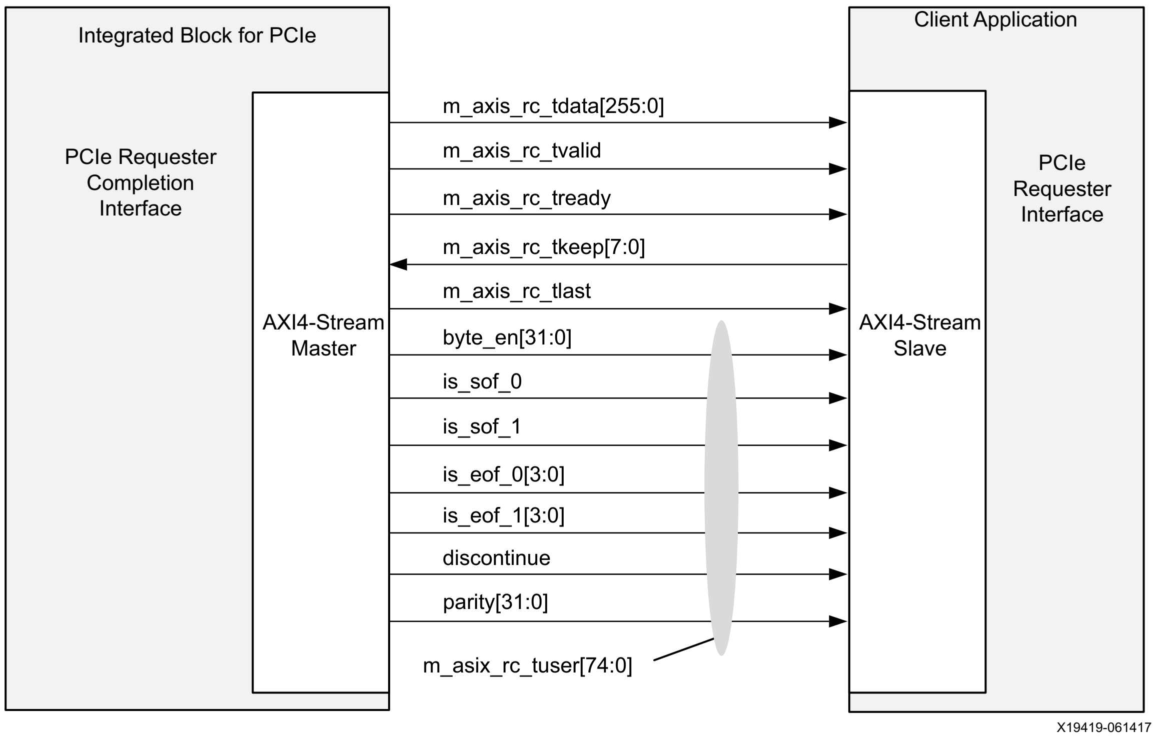


Figure 3-54: **Requester Completion Interface**

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 Figure 3-55. The individual fields of the RC descriptor are described in Table 3-12.

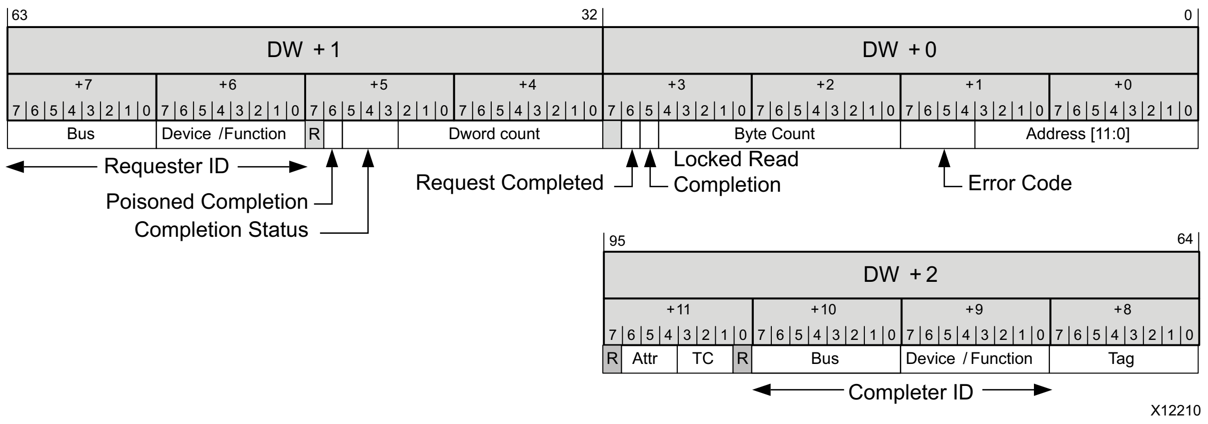


Figure 3-55: **Requester Completion Descriptor Format**

Table 3-12: **Requester Completion Descriptor Fields**

Bit Index	Field Name	Description
11:0	Lower Address	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. When the Completion delivered has an error, only bits [6:0] of the address should be considered valid. This is a byte-level address.

Table 3-12: **Requester Completion Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
15:12	Error Code	Completion error code. These three bits encode error conditions detected from error checking performed by the integrated block on received Completions. Its encodings are: • 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	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. 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.
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 3-12: **Requester Completion Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
30	Request Completed	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. 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.
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: • 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 timing diagrams in Figure 3-56, Figure 3-57, and Figure 3-58 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, page 184.

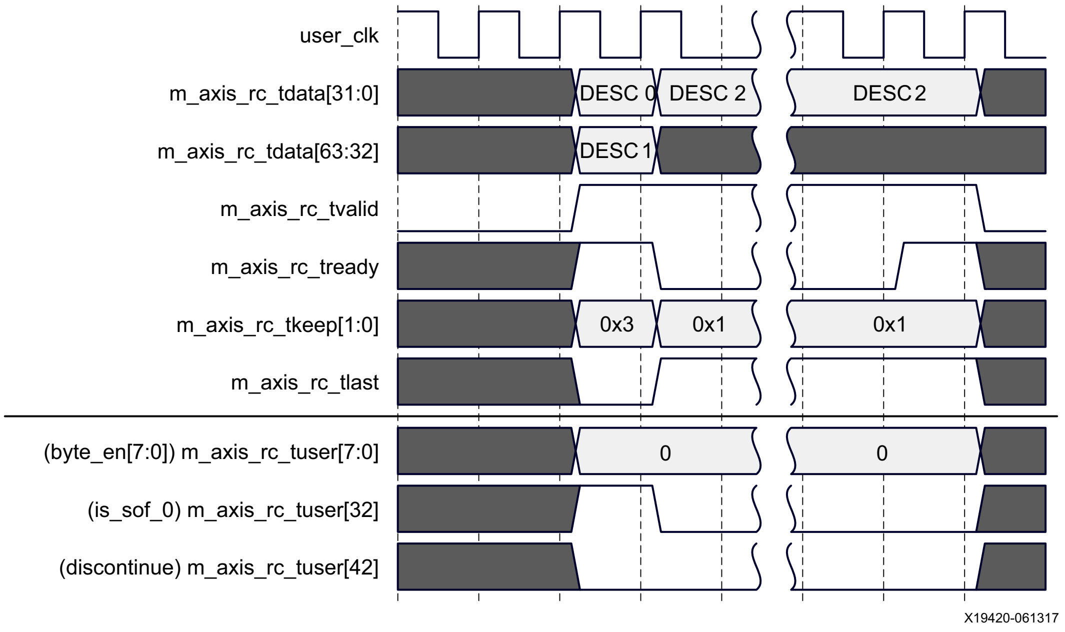


Figure 3-56: **Transfer of a Completion with no Data on the Requester Completion Interface (64-Bit Interface)**

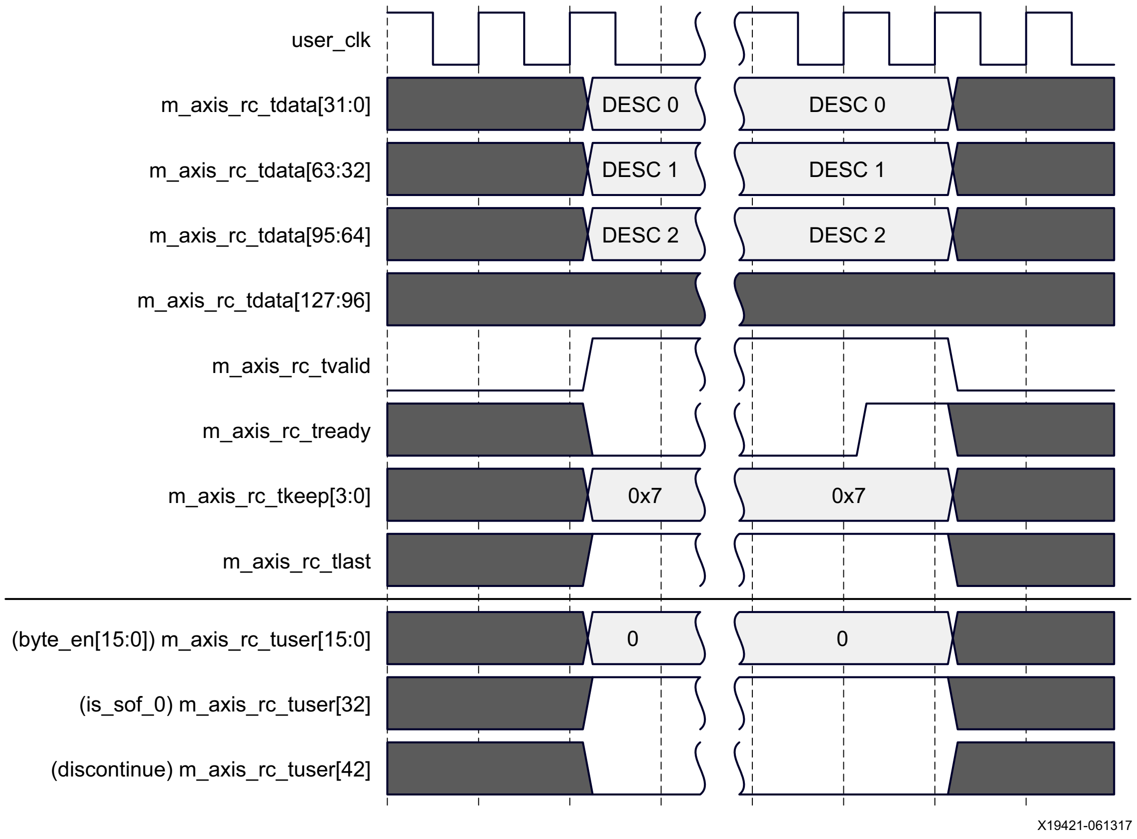


Figure 3-57: **Transfer of a Completion with no Data on the Requester Completion Interface (128-Bit Interface)**

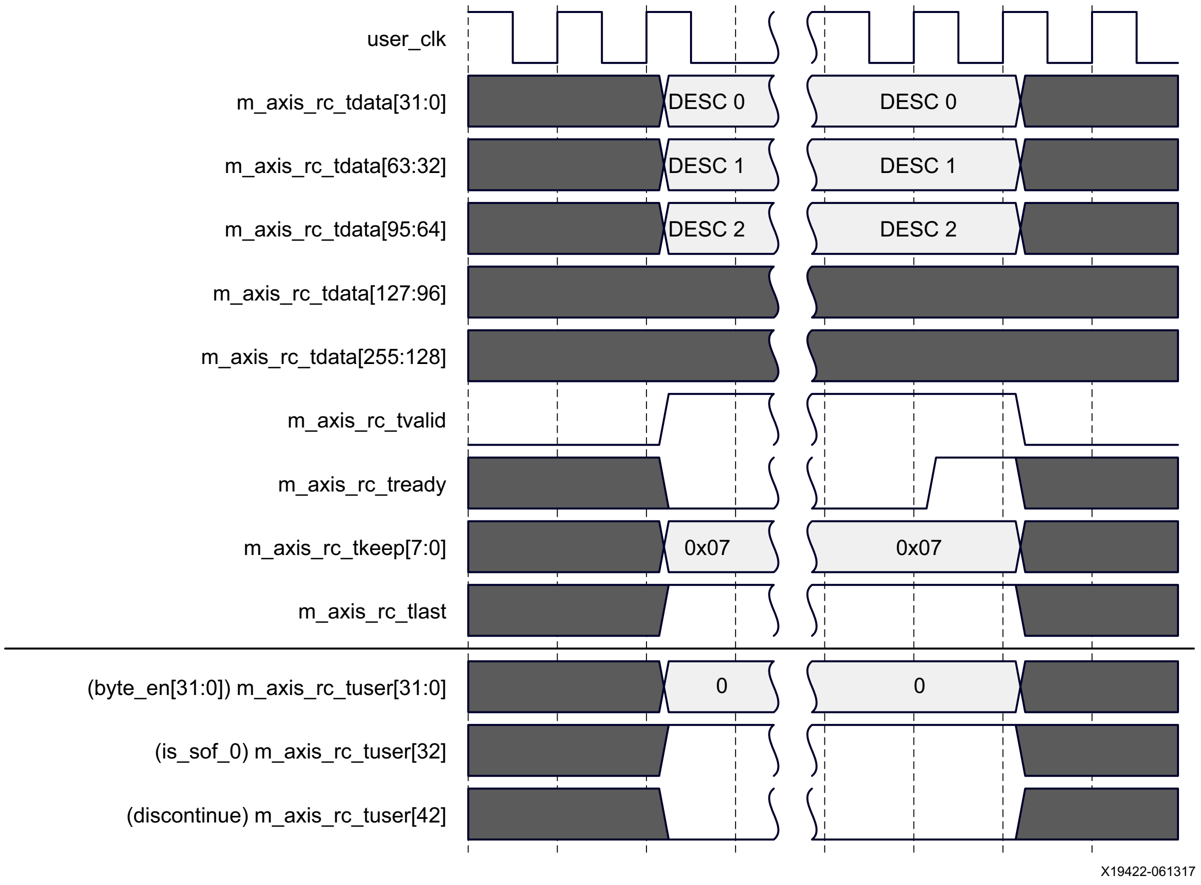


Figure 3-58: **Transfer of a Completion with no Data on the Requester Completion Interface (256-Bit Interface)**

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 Dwords 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 timing diagrams in Figure 3-59, Figure 3-60, and Figure 3-61 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, page 184.

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.

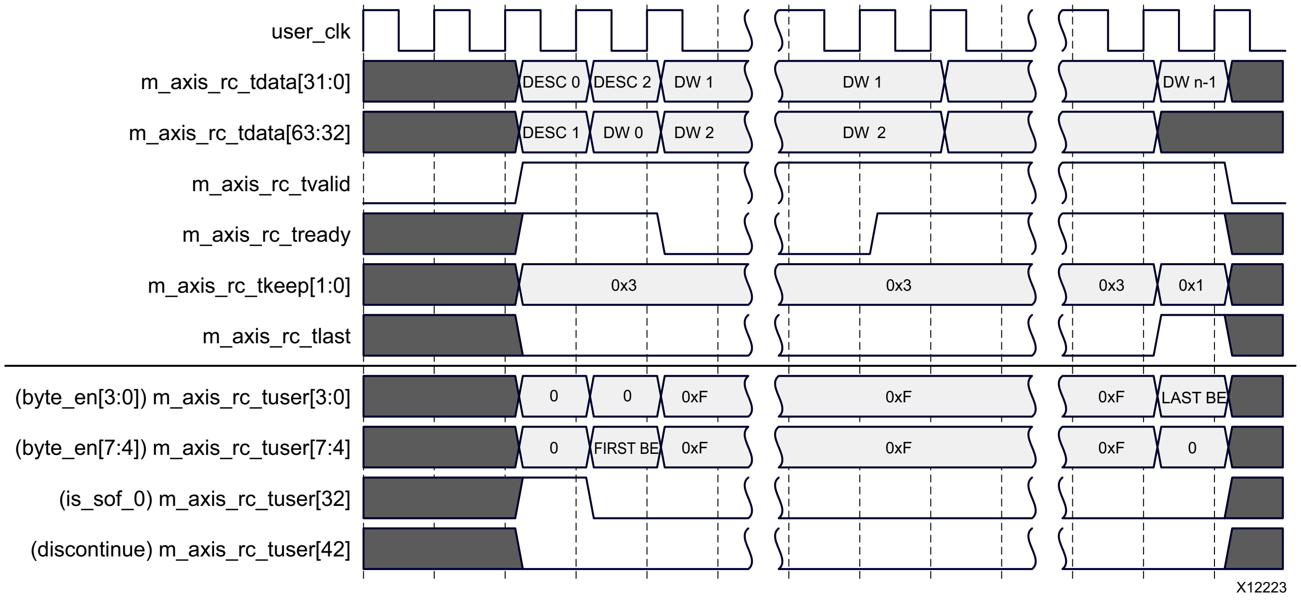


Figure 3-59: **Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode, 64-Bit Interface)**

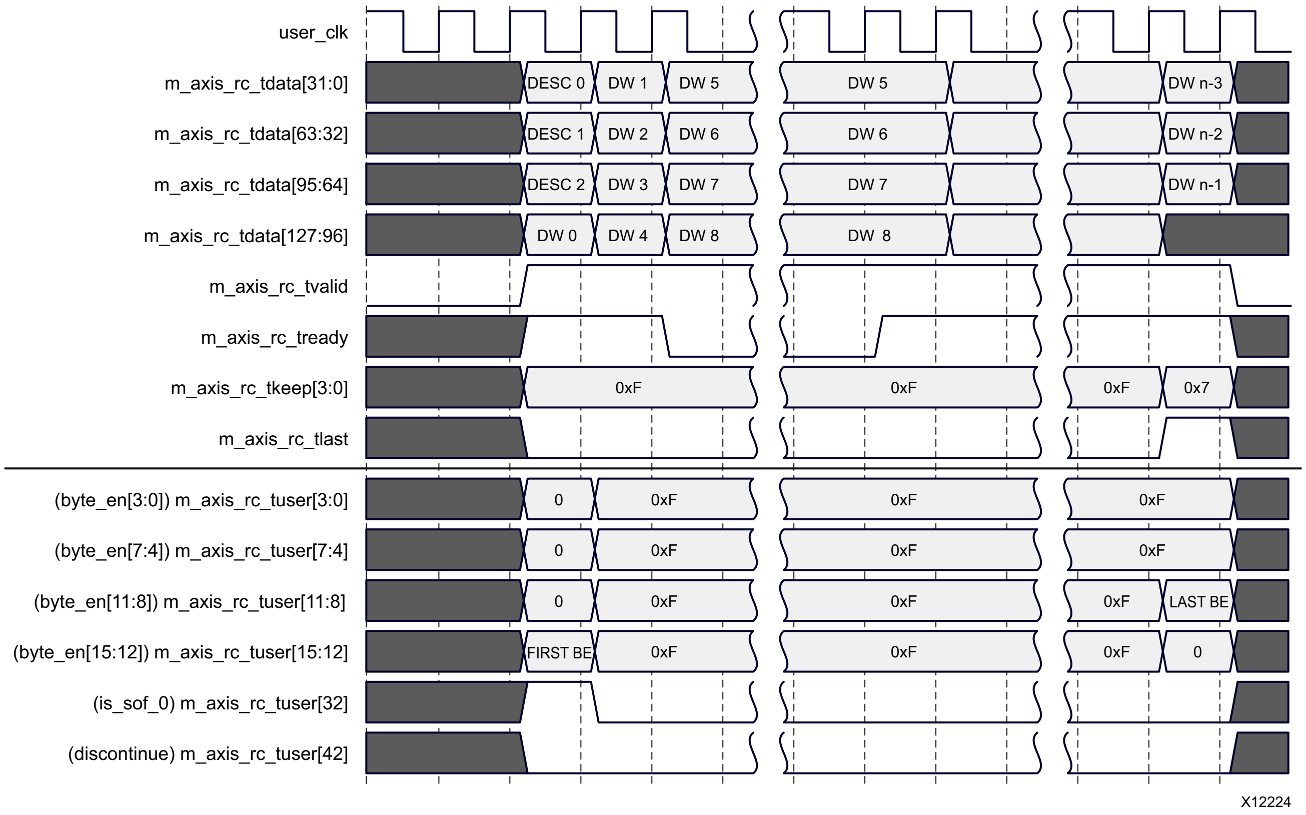


Figure 3-60: **Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode, 128-Bit Interface)**

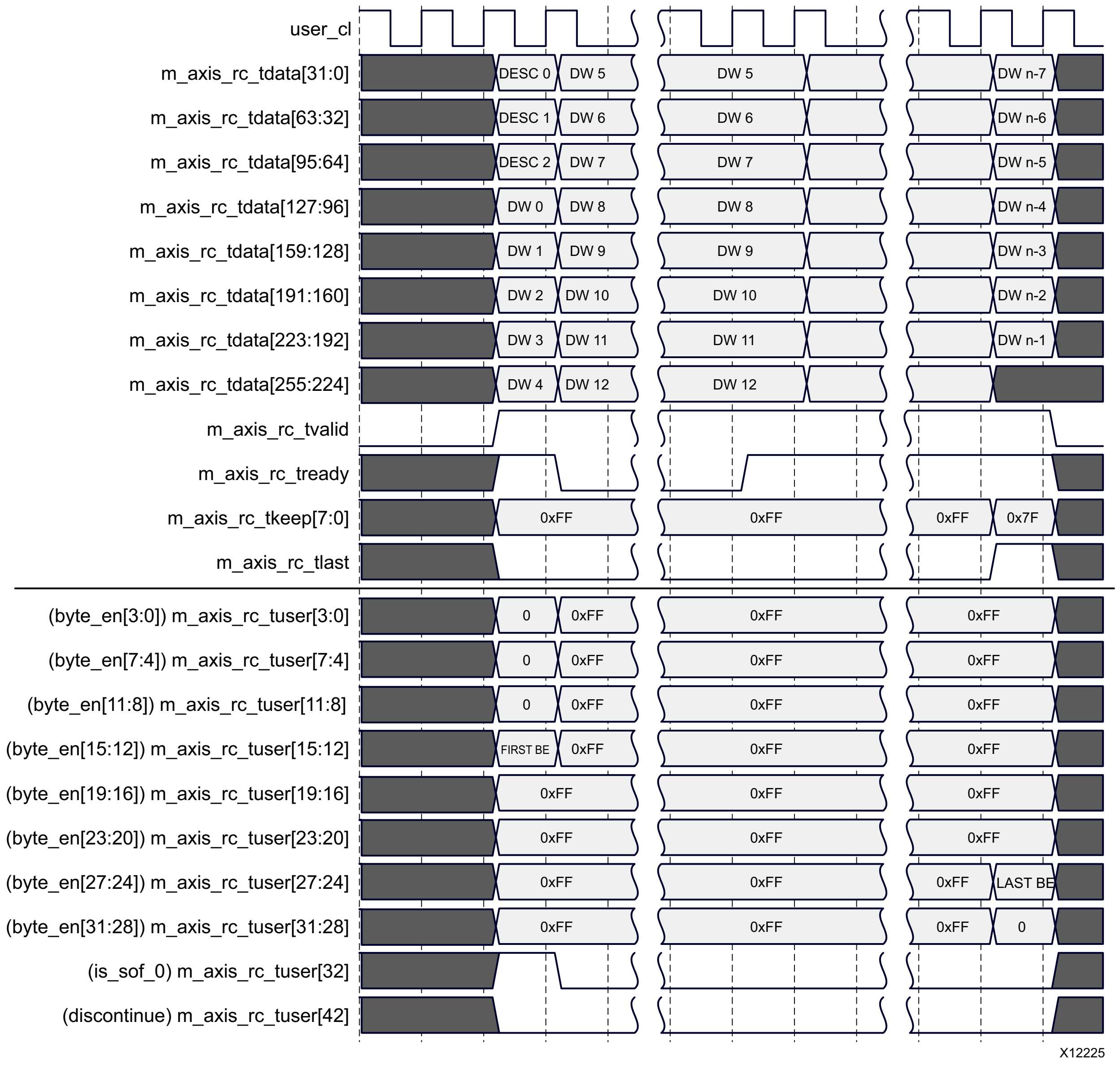


Figure 3-61: **Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode, 256-Bit Interface)**

The timing diagrams in Figure 3-62, Figure 3-63, and Figure 3-64 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.

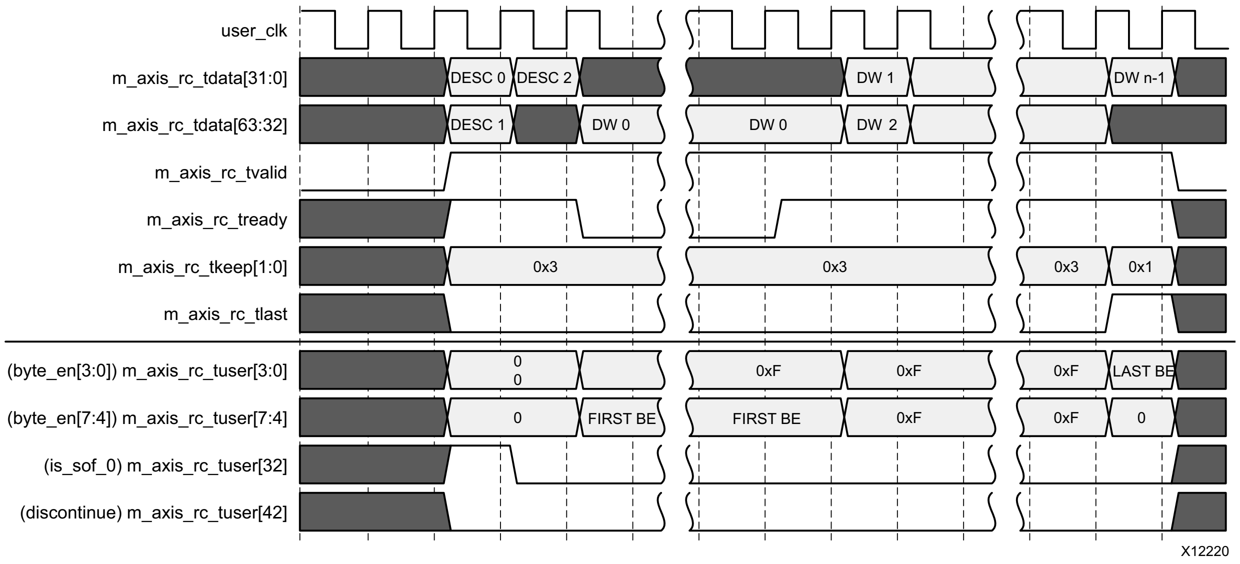


Figure 3-62: **Transfer of a Completion with Data on the Requester Completion Interface (Address-Aligned Mode, 64-Bit Interface)**

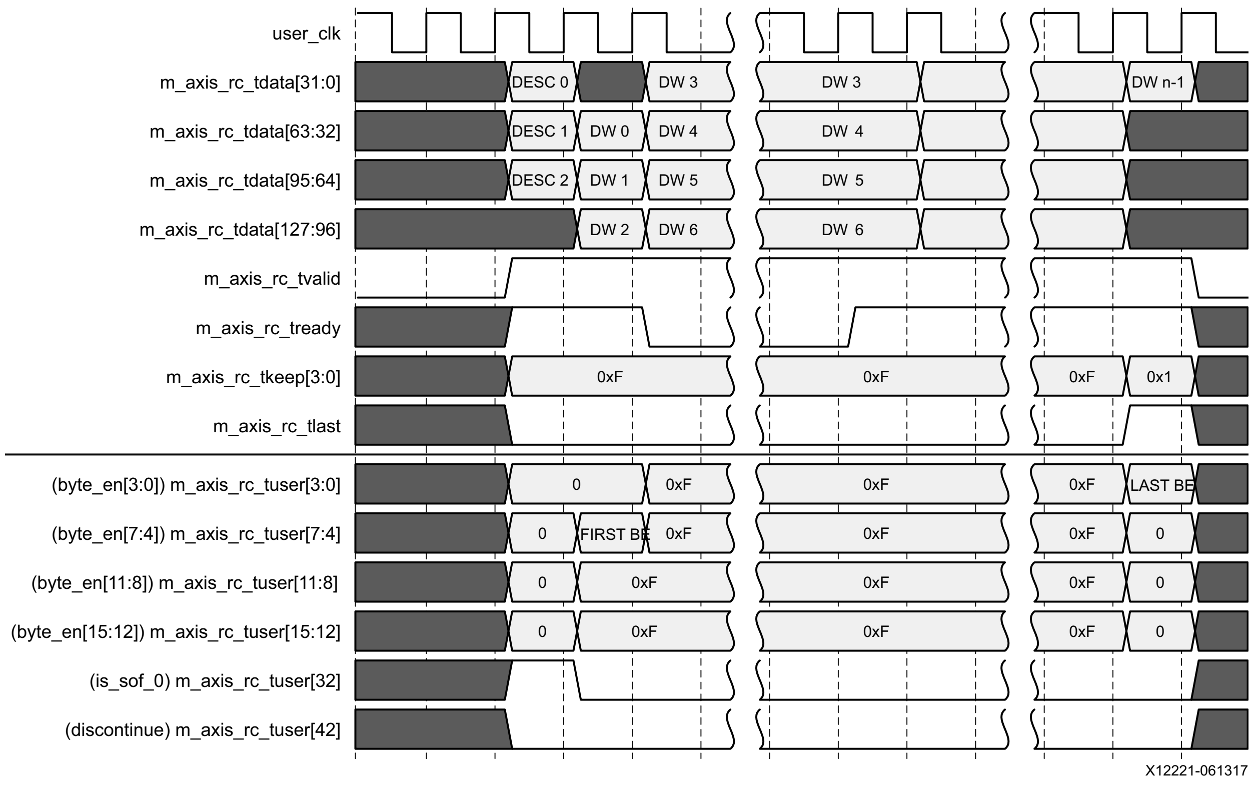


Figure 3-63: **Transfer of a Completion with Data on the Requester Completion Interface (Address-Aligned Mode, 128-Bit Interface)**

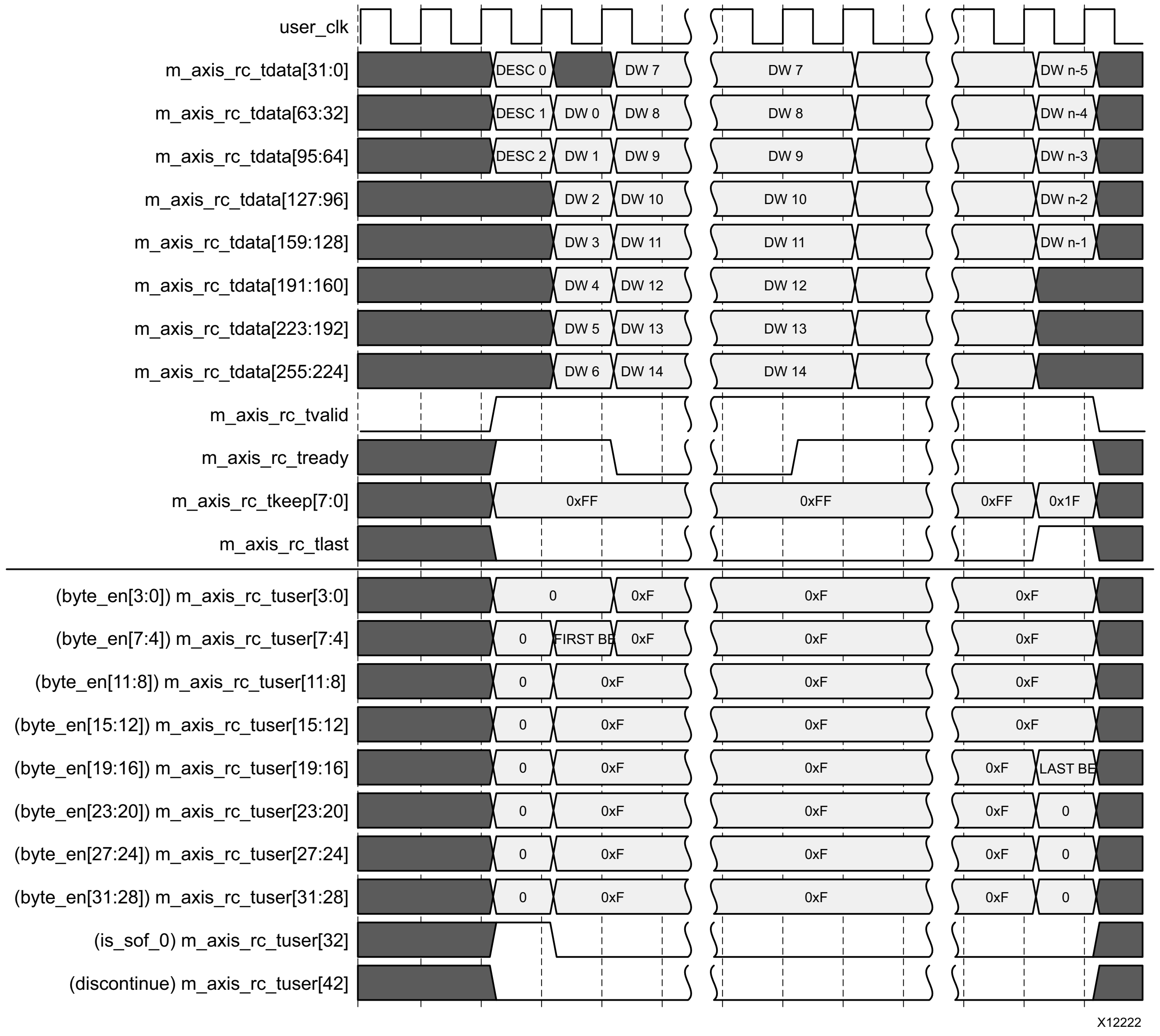


Figure 3-64: **Transfer of a Completion with Data on the Requester Completion Interface (Address-Aligned Mode, 256-Bit Interface)**

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.

Figure 3-65 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.

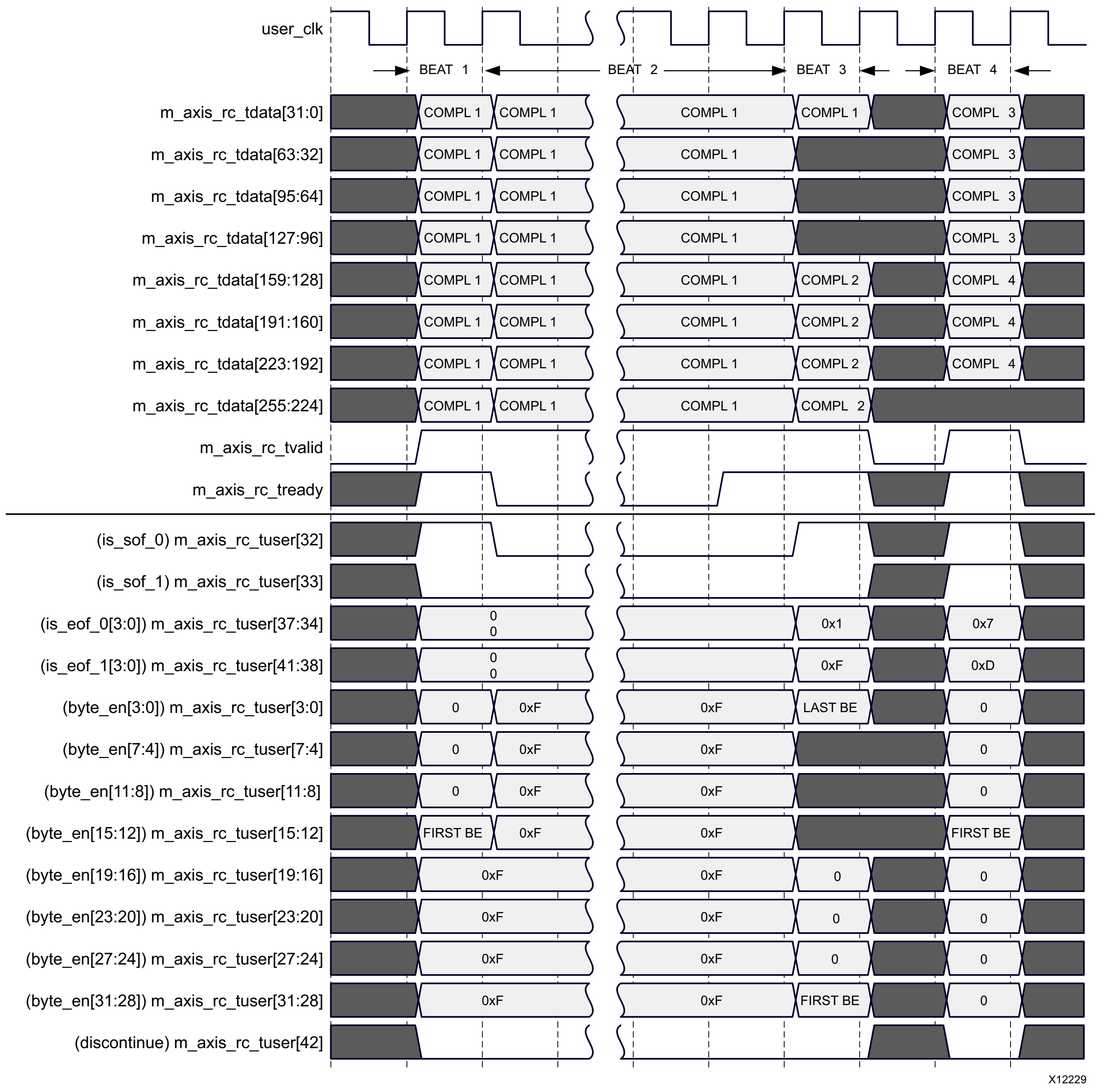

user_clk
BEAT 1
BEAT 2
BEAT 3
BEAT 4
m_axis_rc_tdata[31:0]
m_axis_rc_tdata[63:32]
m_axis_rc_tdata[95:64]
m_axis_rc_tdata[127:96]
m_axis_rc_tdata[159:128]
m_axis_rc_tdata[191:160]
m_axis_rc_tdata[223:192]
m_axis_rc_tdata[255:224]
m_axis_rc_tvalid
m_axis_rc_tready
(is_sof_0) m_axis_rc_tuser[32]
(is_sof_1) m_axis_rc_tuser[33]
(is_eof_0[3:0]) m_axis_rc_tuser[37:34]
(is_eof_1[3:0]) m_axis_rc_tuser[41:38]
(byte_en[3:0]) m_axis_rc_tuser[3:0]
(byte_en[7:4]) m_axis_rc_tuser[7:4]
(byte_en[11:8]) m_axis_rc_tuser[11:8]
(byte_en[15:12]) m_axis_rc_tuser[15:12]
(byte_en[19:16]) m_axis_rc_tuser[19:16]
(byte_en[23:20]) m_axis_rc_tuser[23:20]
(byte_en[27:24]) m_axis_rc_tuser[27:24]
(byte_en[31:28]) m_axis_rc_tuser[31:28]
(discontinue) m_axis_rc_tuser[42]
COMPL 1
COMPL 2
COMPL 3
COMPL 4
0x1
0x7
0xF
0xD
FIRST BE
LAST BE
0xF
0
X12229

Figure 3-65: **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. Table 3-13 defines the error conditions signaled by the various error codes.

Table 3-13: **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 3-13: **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 AXI Stream Interface. Figure 3-66 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.

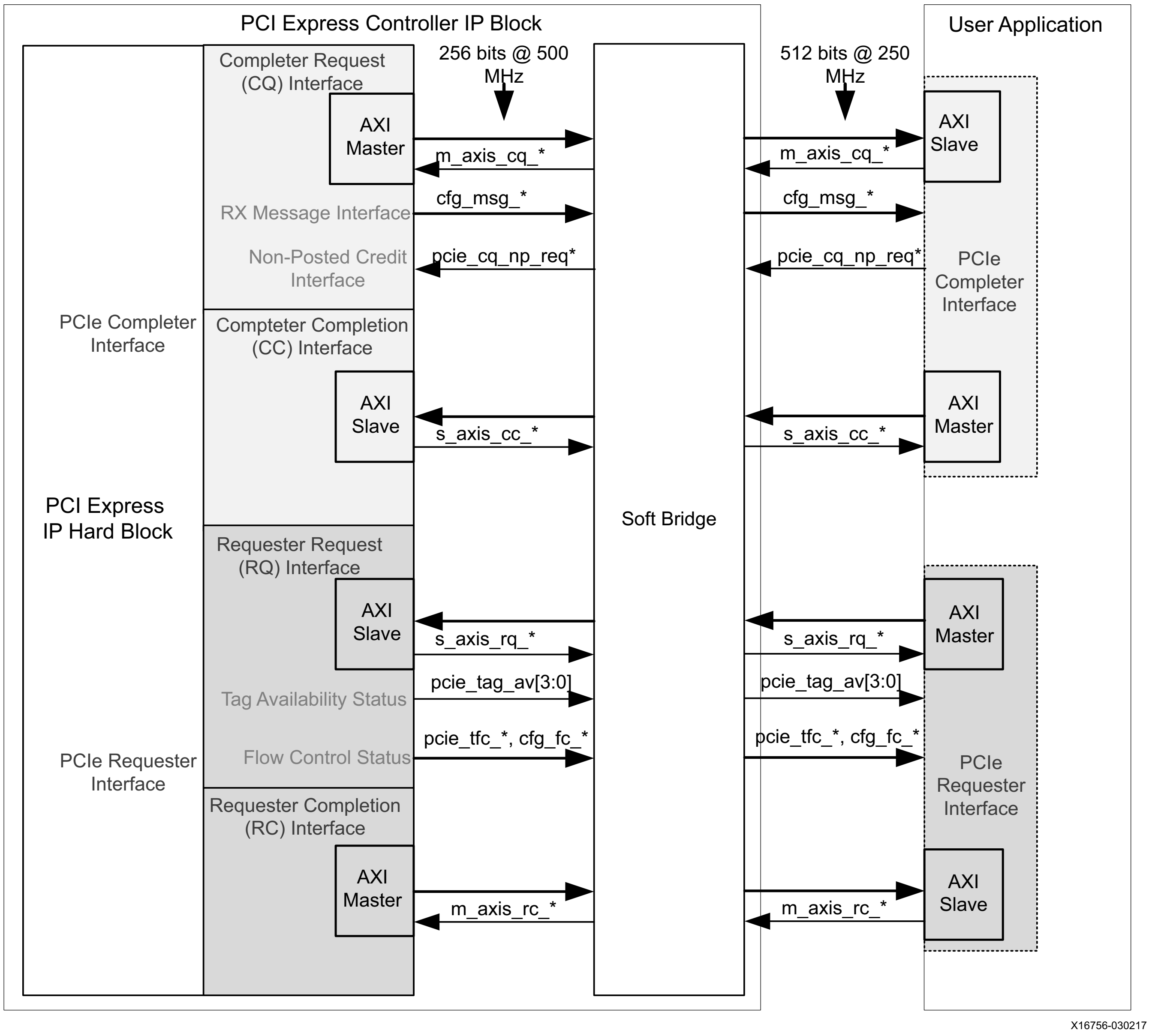


Figure 3-66: **Block Diagram of PCIe IP With Soft Bridge**

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)

Figure 3-67 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.

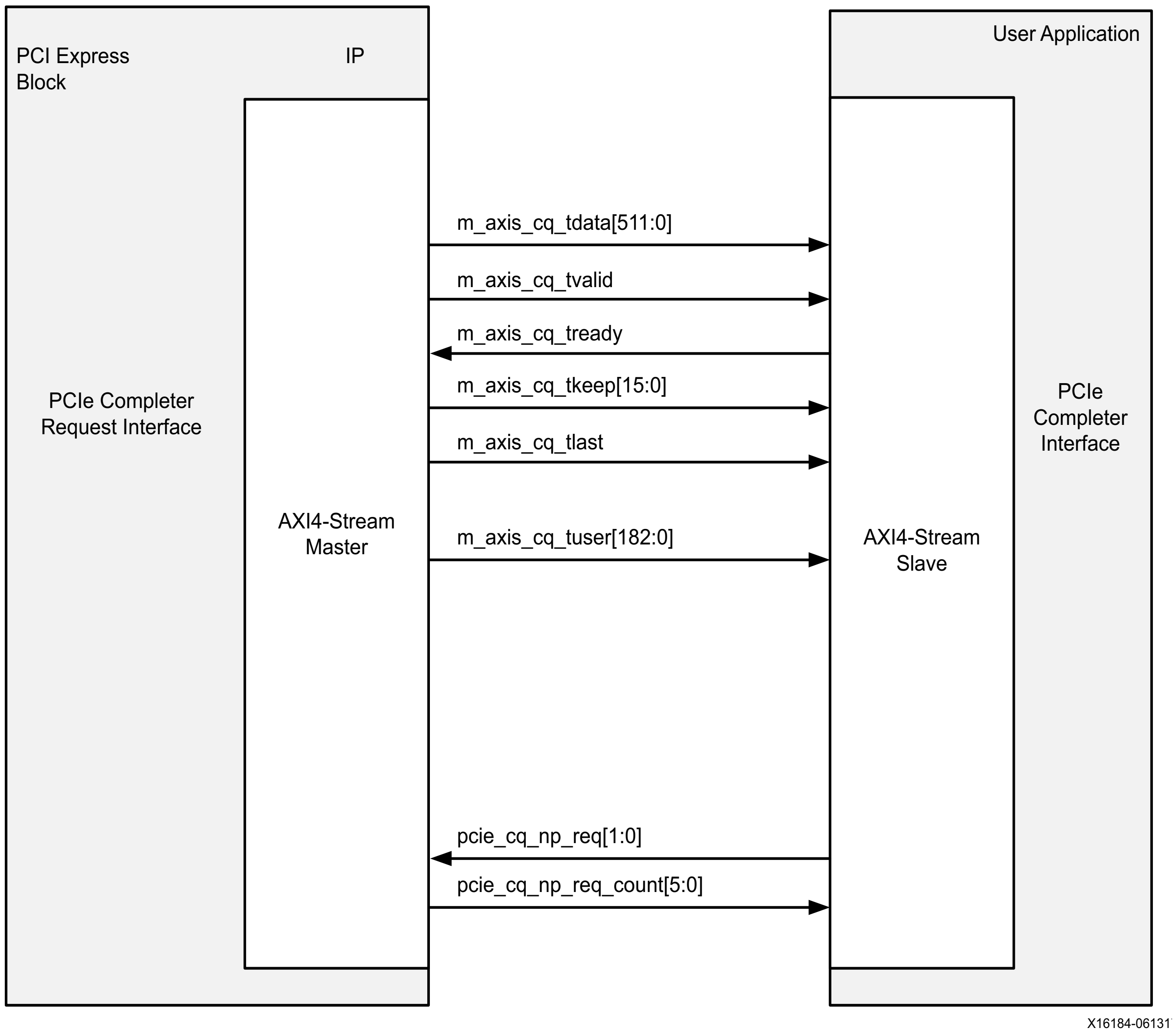


Figure 3-67: **Completer Request Interface Signals**

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 mod 4) mod 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. The lane number corresponding to the first byte of the payload is determined as n = (S + 16 + (A mod 16)) mod 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 memory or I/O address corresponding to the first byte of the payload. This means that the payload can start only at one of four byte lanes: 16, 20, 24 and 28.

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 Figure 3-68, Figure 3-69, Figure 3-70 and Figure 3-71. The format of Figure 3-68 applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request. The format of Figure 3-69 is used for Vendor-Defined Messages (Type 0 or Type 1) only. The format of Figure 3-70 is used for all ATS messages (Invalid Request, Invalid Completion, Page Request, PRG Response). For all other messages, the descriptor takes the format of Figure 3-71.

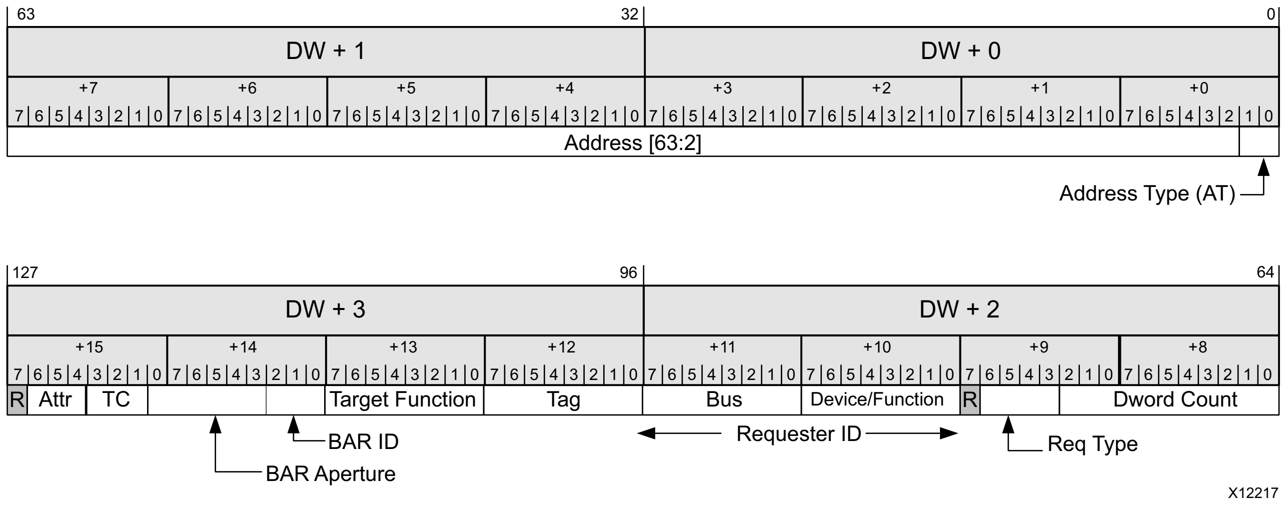


Figure 3-68: **Completer Request Descriptor Format for Memory, I/O, and Atomic Op Requests**

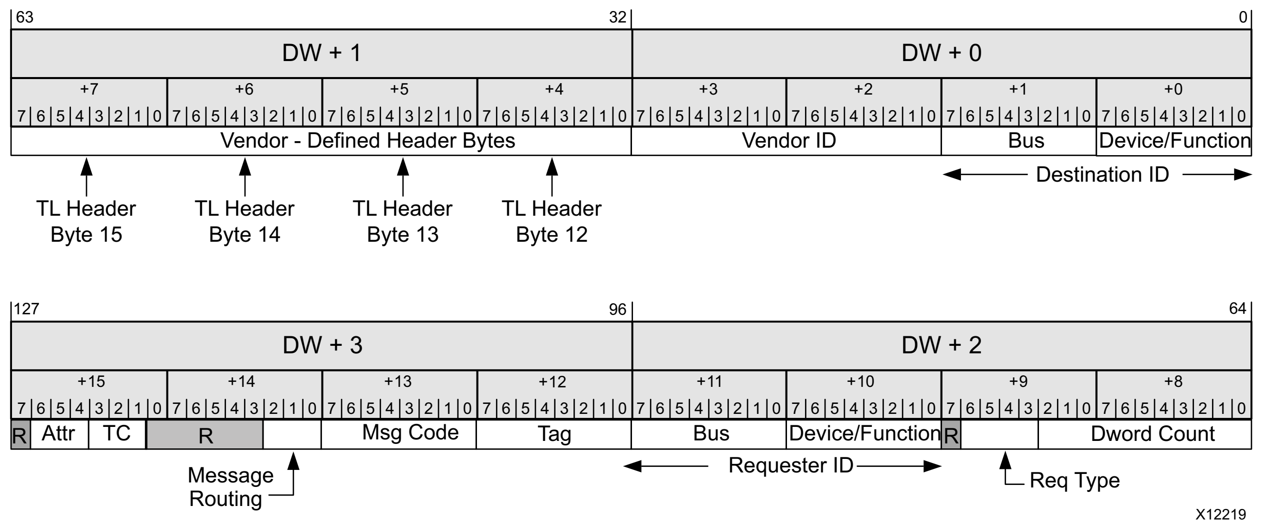


Figure 3-69: **Completer Request Descriptor Format for Vendor-Defined Messages**

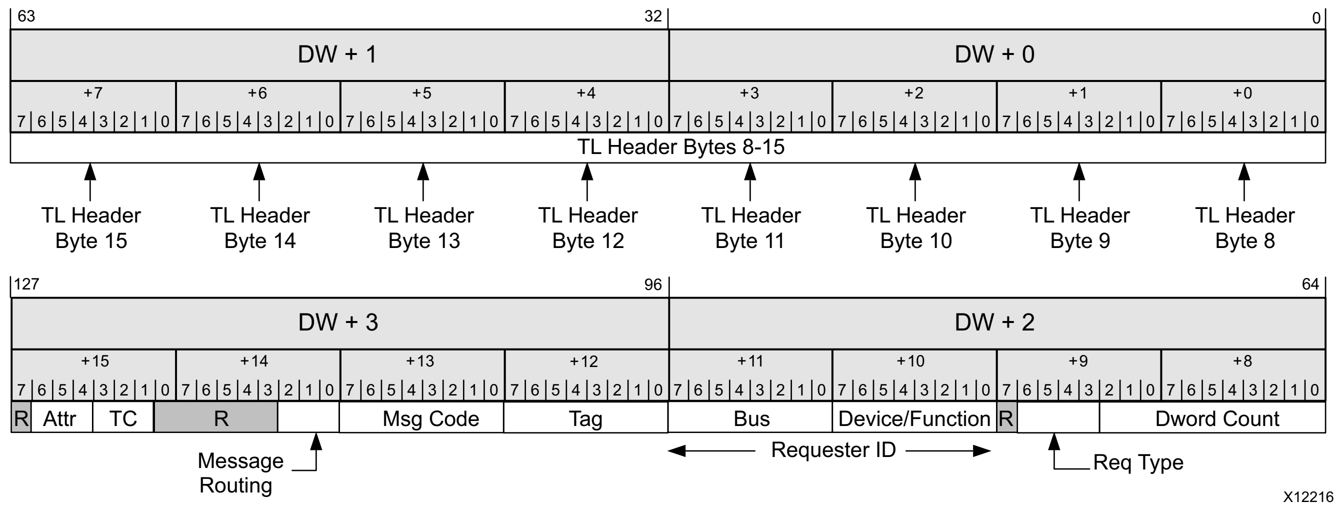


Figure 3-70: **Completer Request Descriptor Format for ATS Messages**

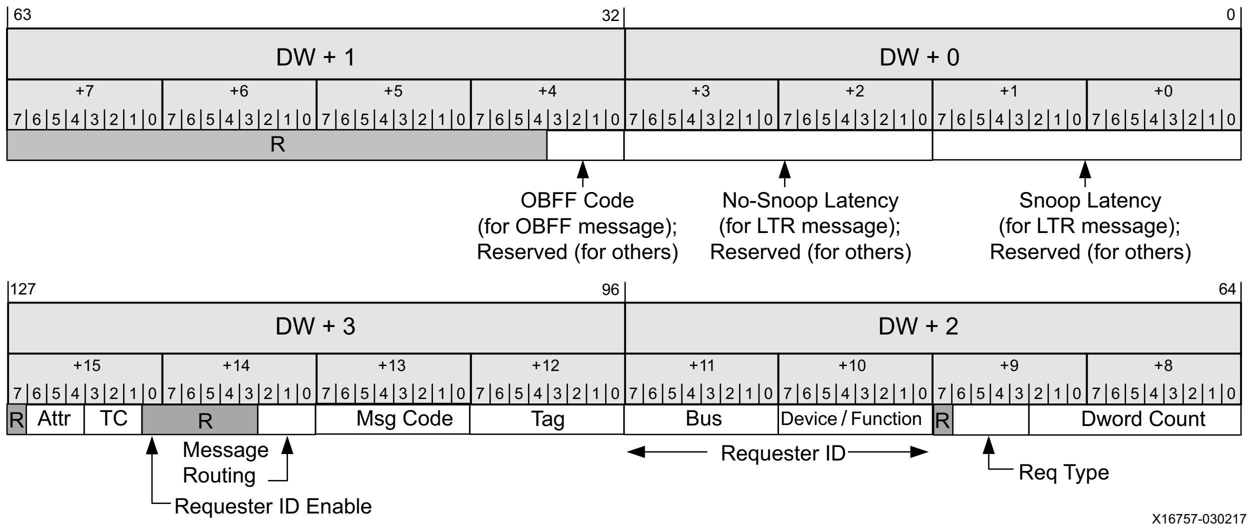


Figure 3-71: **Completer Request Descriptor Format for All Other Messages**

Table 3-14: **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. 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. It provides the address from the TL 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 is 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 zeroes.
78:75	Request Type	Identifies the transaction type. The transaction types and their encodings are listed in Table 3-15.

Table 3-14: **Completer Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
95:80	Requester ID	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. 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.
103:96	Tag	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.
111:104	Target Function	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.
114:112	BAR ID	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. 000 = BAR 0 (VF-BAR 0 for VFs) **_Note:_** In RP mode, BAR ID is always 000. 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 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).
120:115	BAR Aperture	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. For VF BARs, the value provided on this output is based on the memory space consumed by a single VF covered by the BAR.
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.

Table 3-14: **Completer Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
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 3-15: **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

Figure 3-72 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, for some *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 AXI-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.

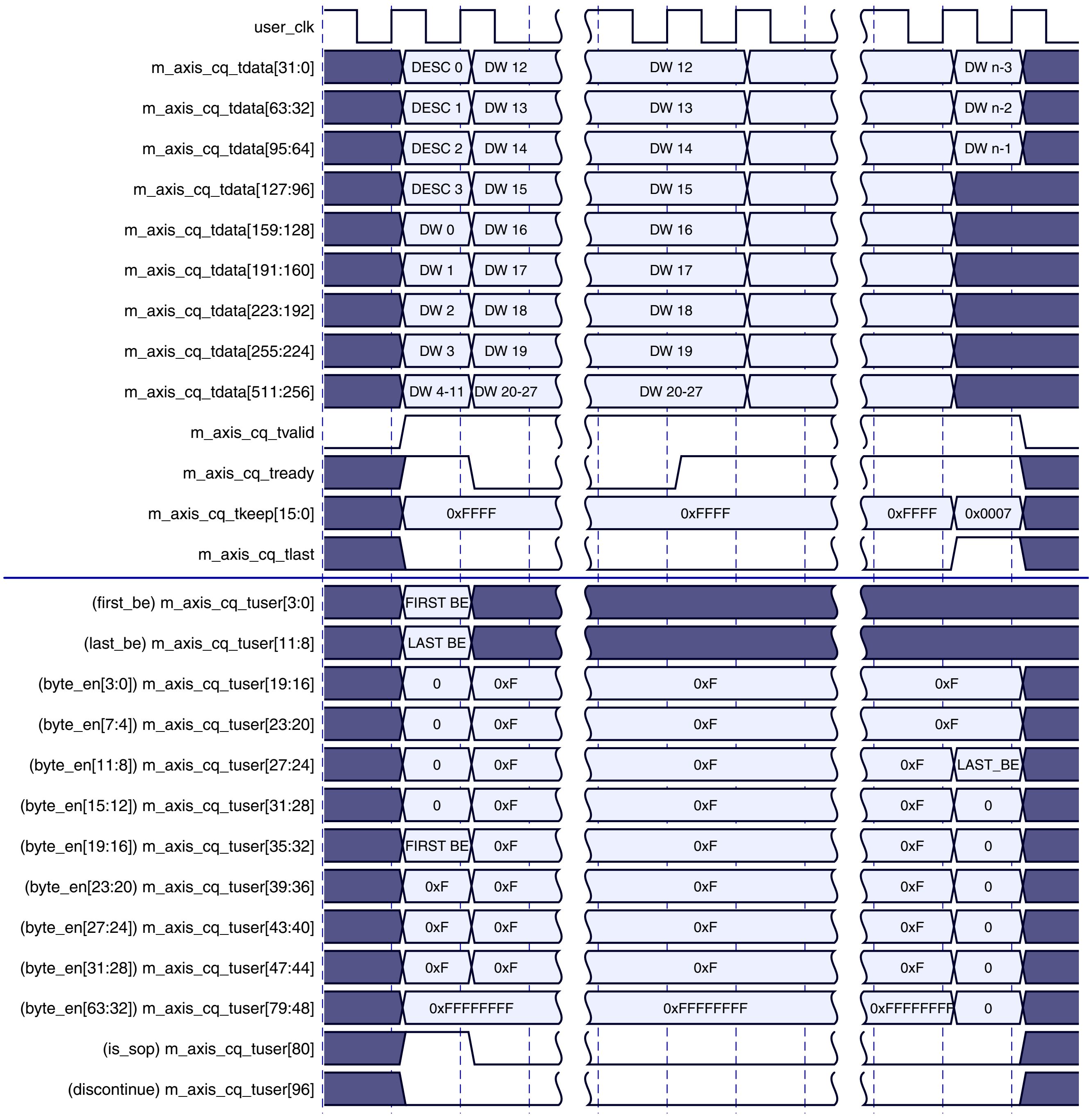

user_clk
m_axis_cq_tdata[31:0]
m_axis_cq_tdata[63:32]
m_axis_cq_tdata[95:64]
m_axis_cq_tdata[127:96]
m_axis_cq_tdata[159:128]
m_axis_cq_tdata[191:160]
m_axis_cq_tdata[223:192]
m_axis_cq_tdata[255:224]
m_axis_cq_tdata[511:256]
m_axis_cq_tvalid
m_axis_cq_tready
m_axis_cq_tkeep[15:0]
m_axis_cq_tlast
(first_be) m_axis_cq_tuser[3:0]
(last_be) m_axis_cq_tuser[11:8]
(byte_en[3:0]) m_axis_cq_tuser[19:16]
(byte_en[7:4]) m_axis_cq_tuser[23:20]
(byte_en[11:8]) m_axis_cq_tuser[27:24]
(byte_en[15:12]) m_axis_cq_tuser[31:28]
(byte_en[19:16]) m_axis_cq_tuser[35:32]
(byte_en[23:20) m_axis_cq_tuser[39:36]
(byte_en[27:24]) m_axis_cq_tuser[43:40]
(byte_en[31:28]) m_axis_cq_tuser[47:44]
(byte_en[63:32]) m_axis_cq_tuser[79:48]
(is_sop) m_axis_cq_tuser[80]
(discontinue) m_axis_cq_tuser[96]
DESC 0
DESC 1
DESC 2
DESC 3
DW 0
DW 1
DW 2
DW 3
DW 4-11
DW 12
DW 13
DW 14
DW 15
DW 16
DW 17
DW 18
DW 19
DW 20-27
DW n-3
DW n-2
DW n-1
0xFFFF
0x0007
FIRST BE
LAST BE
LAST_BE
0
0xF
0xFFFFFFFF

Figure 3-72: **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` (see Table 2-7, page 17).

The timing diagram in Figure 9 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.

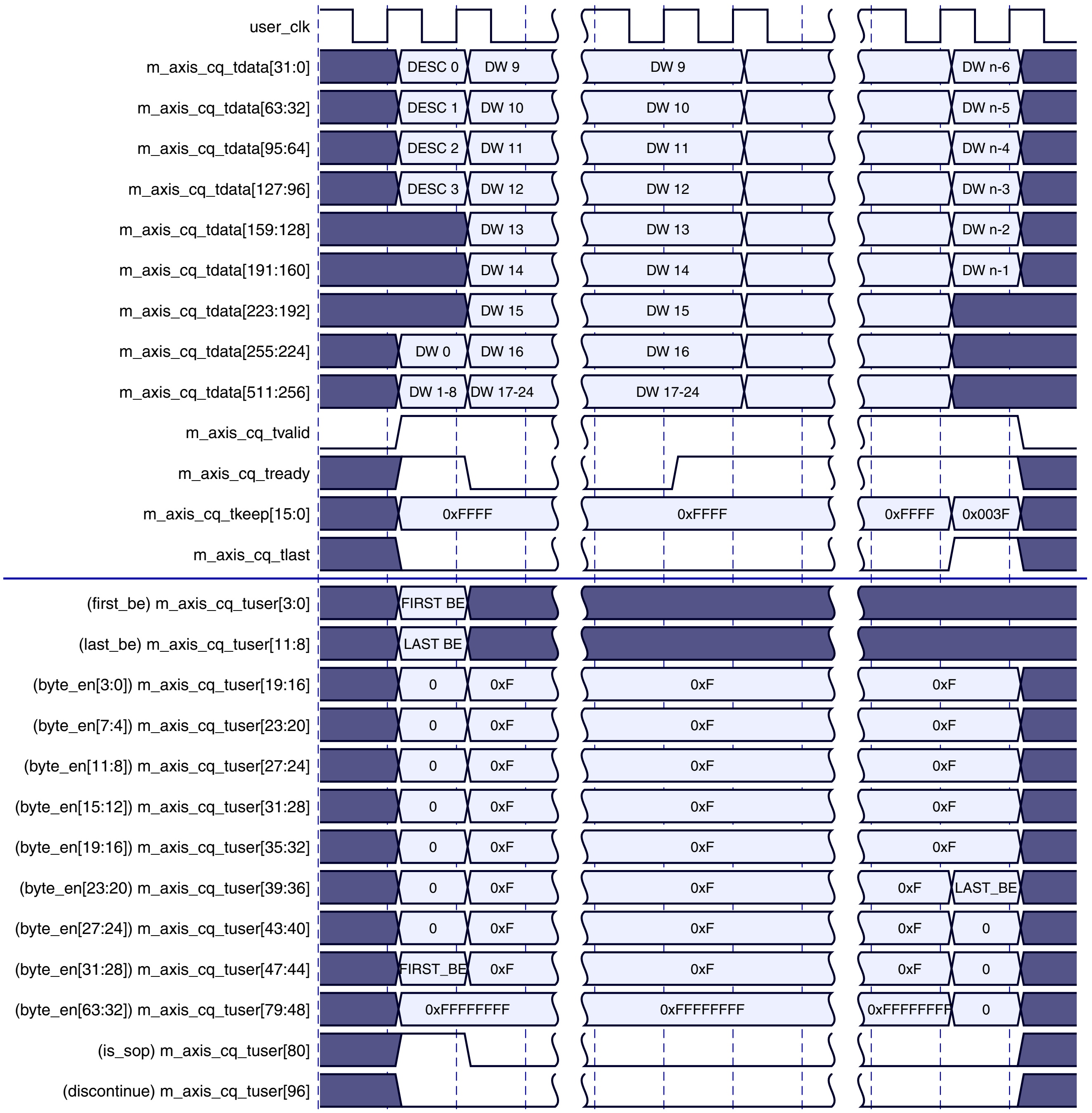

user_clk
m_axis_cq_tdata[31:0]
m_axis_cq_tdata[63:32]
m_axis_cq_tdata[95:64]
m_axis_cq_tdata[127:96]
m_axis_cq_tdata[159:128]
m_axis_cq_tdata[191:160]
m_axis_cq_tdata[223:192]
m_axis_cq_tdata[255:224]
m_axis_cq_tdata[511:256]
m_axis_cq_tvalid
m_axis_cq_tready
m_axis_cq_tkeep[15:0]
m_axis_cq_tlast
(first_be) m_axis_cq_tuser[3:0]
(last_be) m_axis_cq_tuser[11:8]
(byte_en[3:0]) m_axis_cq_tuser[19:16]
(byte_en[7:4]) m_axis_cq_tuser[23:20]
(byte_en[11:8]) m_axis_cq_tuser[27:24]
(byte_en[15:12]) m_axis_cq_tuser[31:28]
(byte_en[19:16]) m_axis_cq_tuser[35:32]
(byte_en[23:20]) m_axis_cq_tuser[39:36]
(byte_en[27:24]) m_axis_cq_tuser[43:40]
(byte_en[31:28]) m_axis_cq_tuser[47:44]
(byte_en[63:32]) m_axis_cq_tuser[79:48]
(is_sop) m_axis_cq_tuser[80]
(discontinue) m_axis_cq_tuser[96]
DESC 0
DESC 1
DESC 2
DESC 3
DW 0
DW 1-8
DW 9
DW 10
DW 11
DW 12
DW 13
DW 14
DW 15
DW 16
DW 17-24
DW n-6
DW n-5
DW n-4
DW n-3
DW n-2
DW n-1
0xFFFF
0x003F
FIRST BE
LAST BE
LAST_BE
FIRST_BE
0
0xF
0xFFFFFFFF

Figure 3-73: **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. Figure 3-75 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 bye is on the bus.

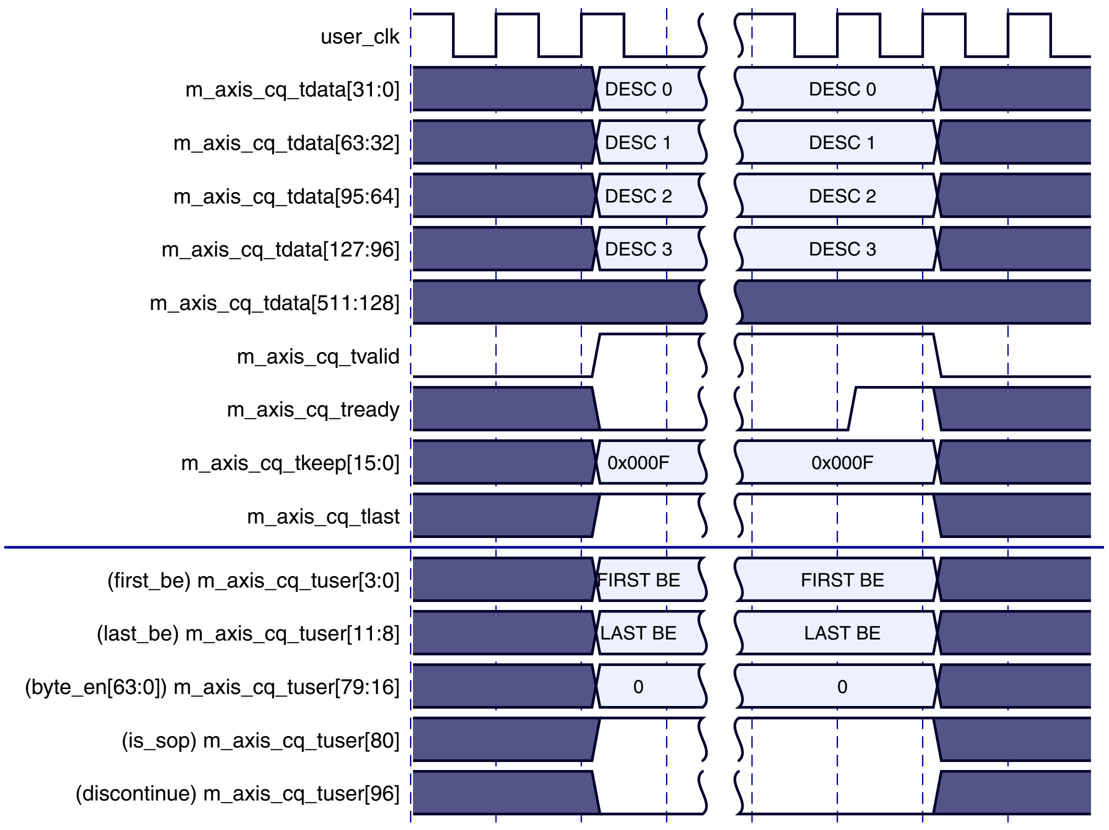


Figure 3-75: **Memory Read Transaction on the Completer Request Interface**

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 bye 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 be 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 be 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 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 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 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 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` indicate 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.

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 (see Table 7). 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 3-16: **AXISTEN_IF_ENABLE_MSG_ROUTE Attribute Bit Descriptions**

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

Table 3-16: **AXISTEN_IF_ENABLE_MSG_ROUTE Attribute Bit Descriptions *(Cont'd)***

Bit Index	Message Type
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 (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 1's 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.

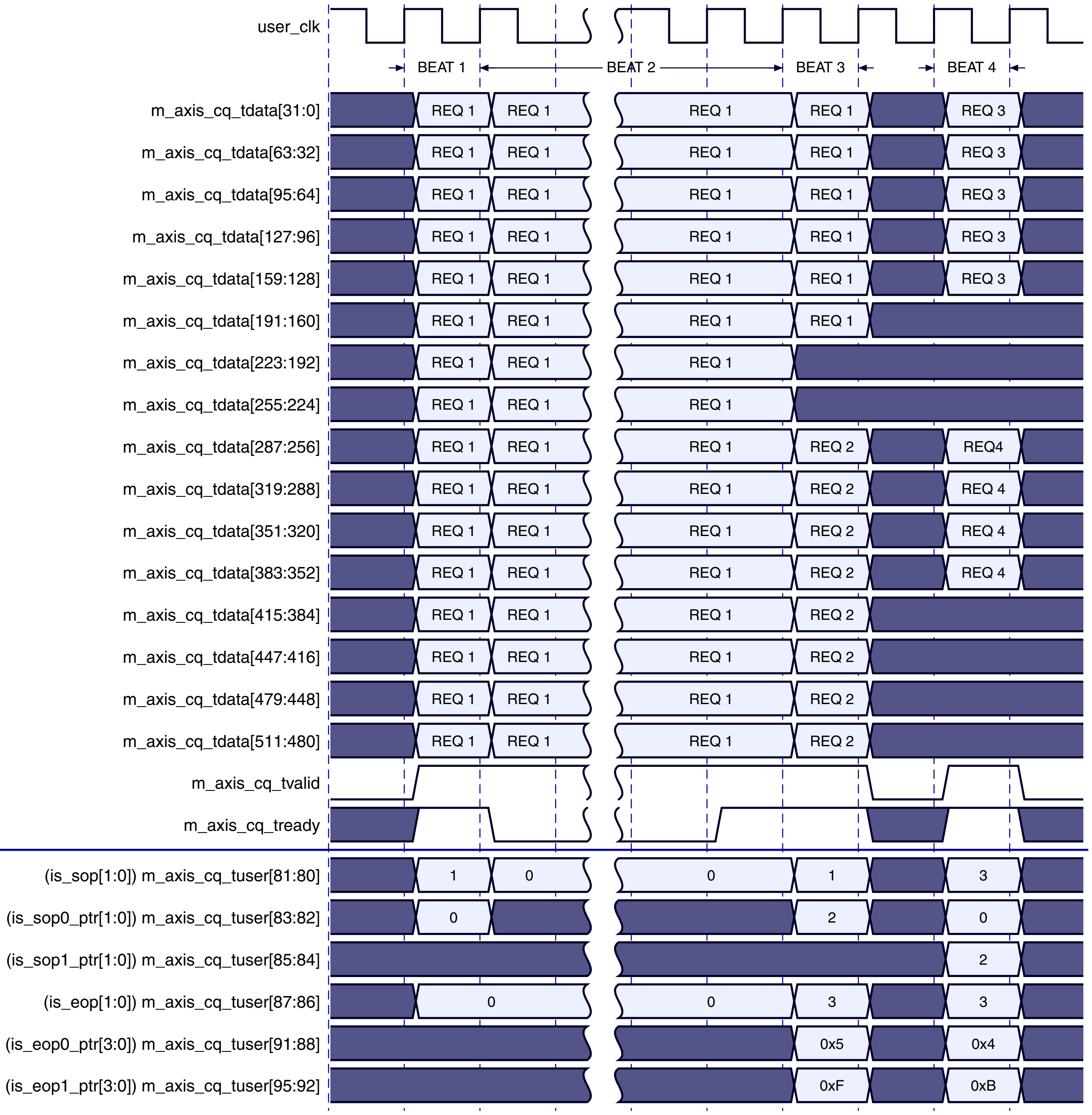


Figure 3-76: **Transfer of Request TLPs on the Completer Request Interface with the Straddle Option Enabled**

Figure 3-76 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)

Figure 3-77 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.

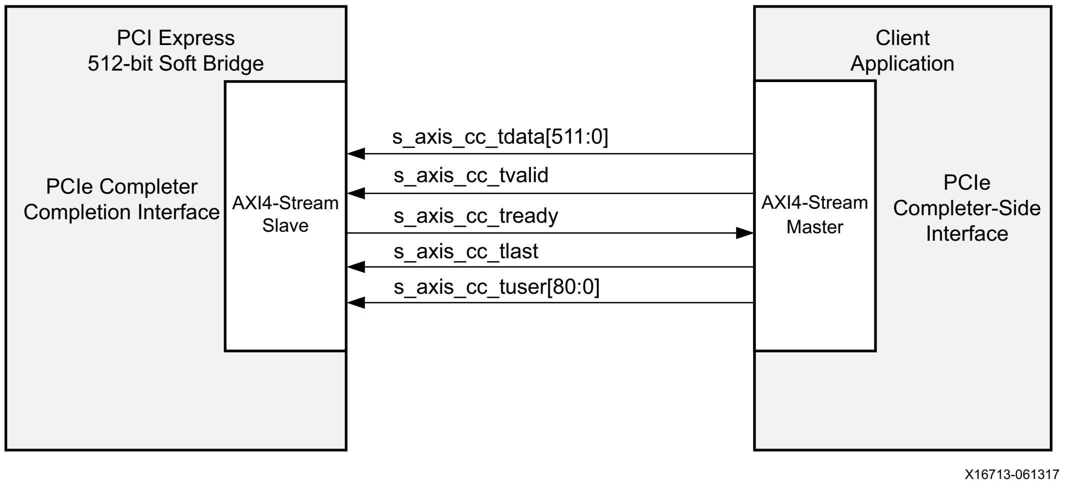


Figure 3-77: **Completer Completion Interface Signals**

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* mod 4)) mod 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* mod 16)) mod 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 Figure 3-77. The individual fields of the completer request descriptor are described in Table 3-17.

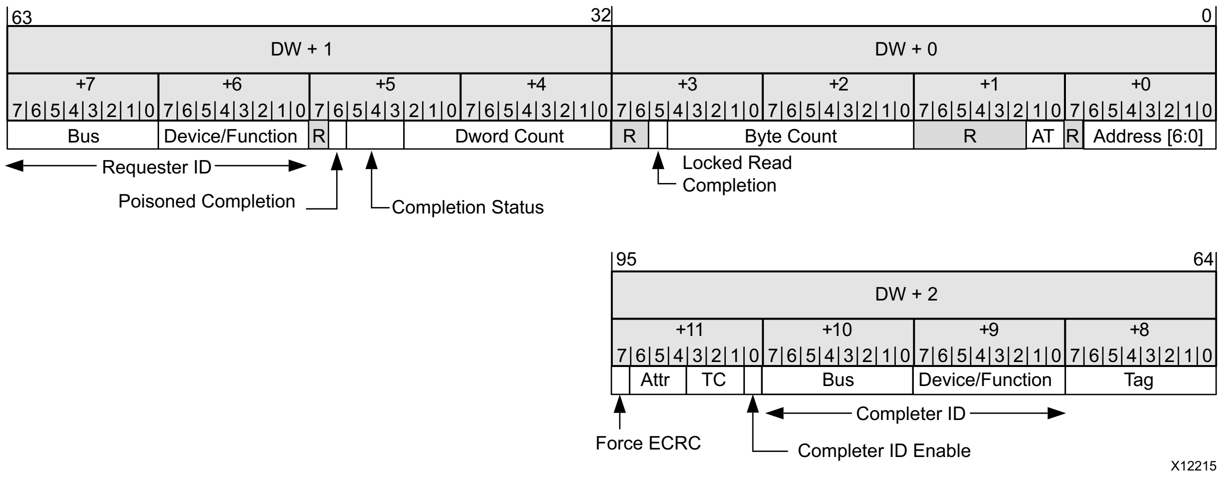


Figure 3-78: **Completer Completion Descriptor Format**

Table 3-17: **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 3-17: **Completer Completion Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
28:16	Byte Count	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. 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
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	These bits must be set based on the type of Completion being sent. The only valid settings are: 000: Successful Completion 001: Unsupported Request (UR) 100: Completer Abort (CA)
46	Poisoned Completion	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.
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 3-17: **Completer Completion Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
79:72	Target Function/Device Number	Function number of the completer Function. The user logic must copy this value from the Target Function field of the descriptor of the corresponding request. When ARI is in use, all 8 bits of this field must be set to the target Function number. Otherwise, bits [74:72] must be set to the target Function number. When ARI is not in use, and the core is configured as a Root Complex, the user application must supply the 5-bit Device Number of the completer on bits [79:75]. When ARI is not use, and the core is configured as an Endpoint, the user logic can optionally supply a 5-bit Device Number of the completer on bits [79:75]. The user logic must set the Completer ID Enable bit in the descriptor if a Device Number is supplied on bits [79:75]. This value is used by core when sending the Completion TLP, instead of the stored value of the Device Number captured by the core from Configuration Requests.
87:80	Completer Bus Number	Bus number associated with the completer Function. When the core is configured as a Root Complex, the user logic must supply the 8-bit Bus Number of the completer in this field. When the core is configured as an Endpoint, the user logic can optionally supply a Bus Number in this field. The user logic must set the Completer ID Enable bit in the descriptor if a Bus Number is supplied in this field. This value is used by core when sending the Completion TLP, instead of the stored value of the Bus Number captured by the core from Configuration Requests.
88	Completer ID Enable	The purpose of this field is to enable the user logic to supply the bus and device numbers to be used in the Completer ID. This field is applicable only to Endpoint cores. If this field is 0, the core uses the captured values of the bus and device numbers to form the Completer ID. If this input is 1, the core uses the bus and device numbers supplied by the user logic in the descriptor to form the Completer ID.
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 example timing diagrams (Figure 3-79), 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, $k > 0$.

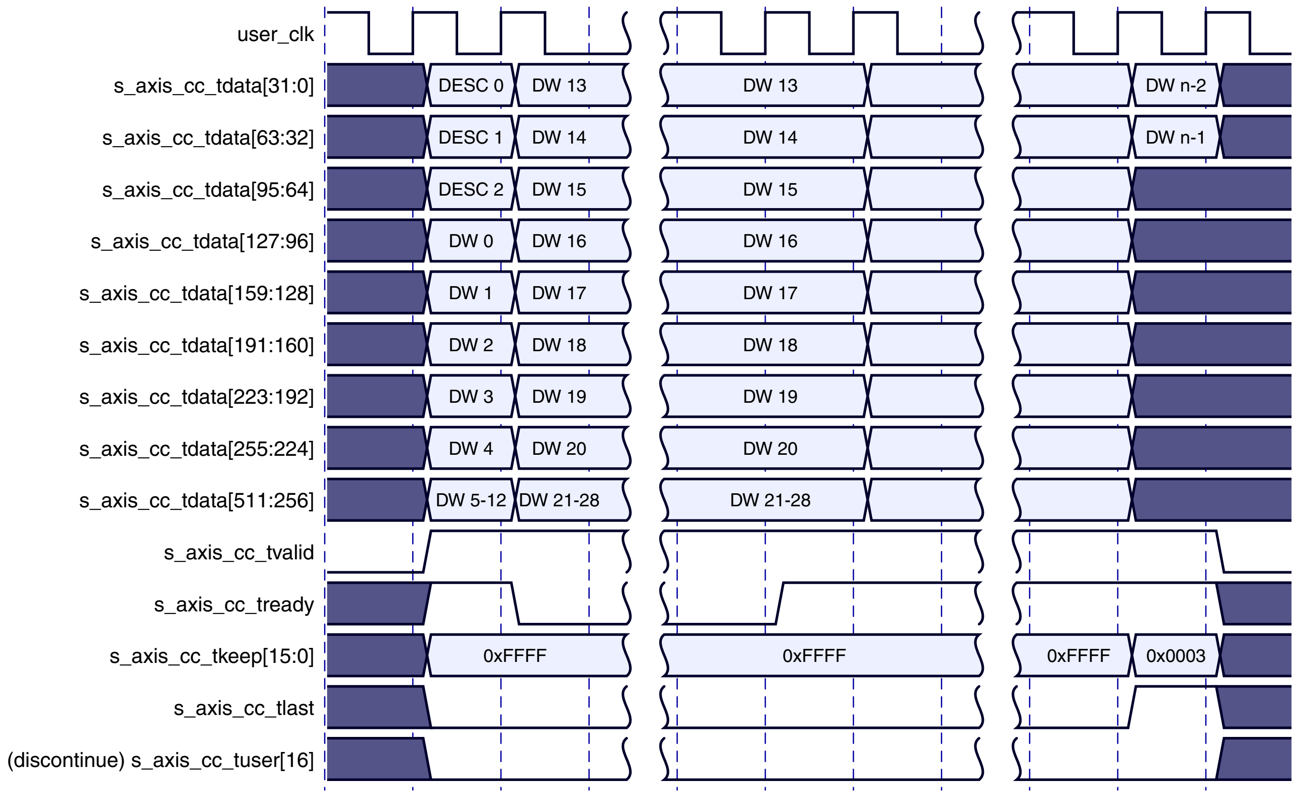


Figure 3-79: **Transfer of a Normal Completion on the Completer Completion Interface (Dword-Aligned Mode)**

Figure 3-79 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 be 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 second128-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 timing diagram in Figure 3-80 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 $k > 1$.

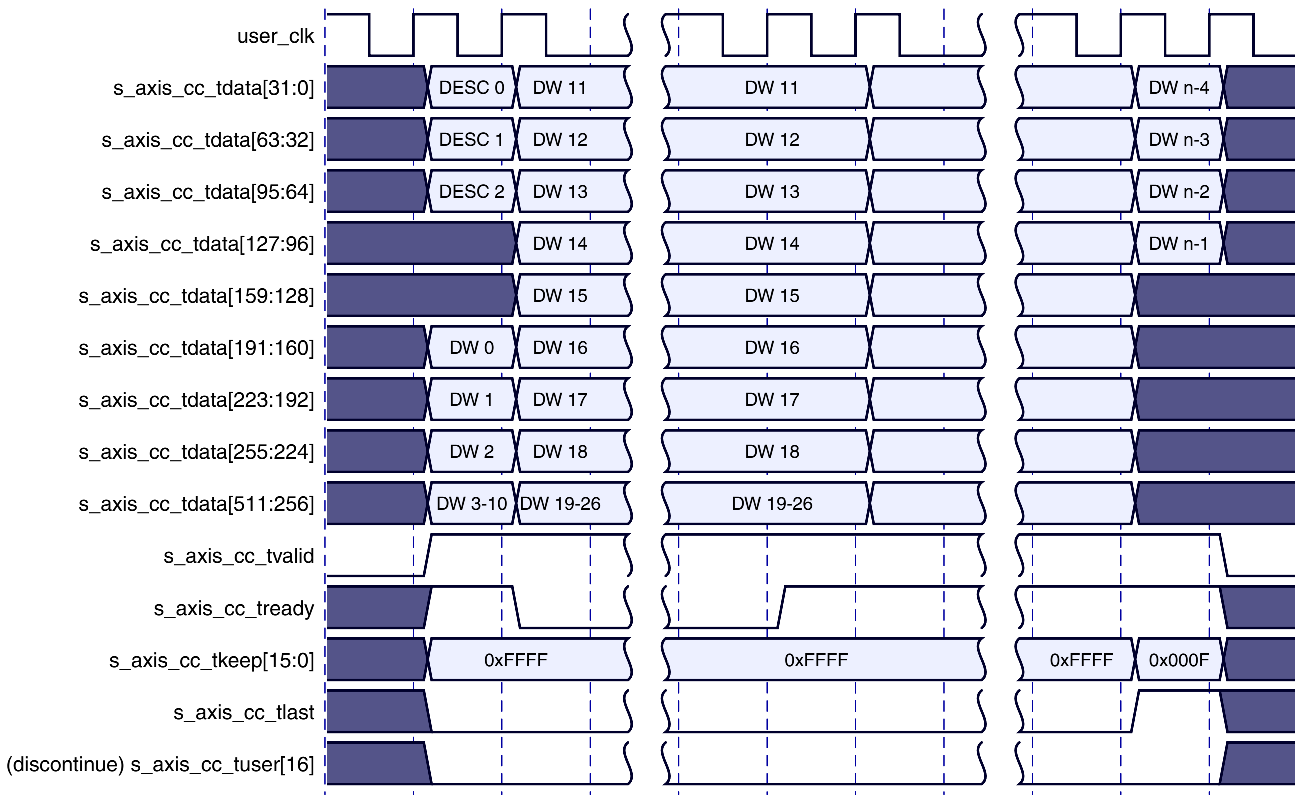


Figure 3-80: **Transfer of a Normal Completion on the Completer Completion Interface (128-bit Address Aligned Mode)**

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 Figure 3-78, followed by five additional Dwords containing information on the request that generated the Completion. These five Dwords are necessary for the PCIe core to log information about the request in its AER header log registers.

Figure 3-81 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 a follows:

- The first three Dwords contain the completion descriptor in the format of Figure 3-78.
- 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.

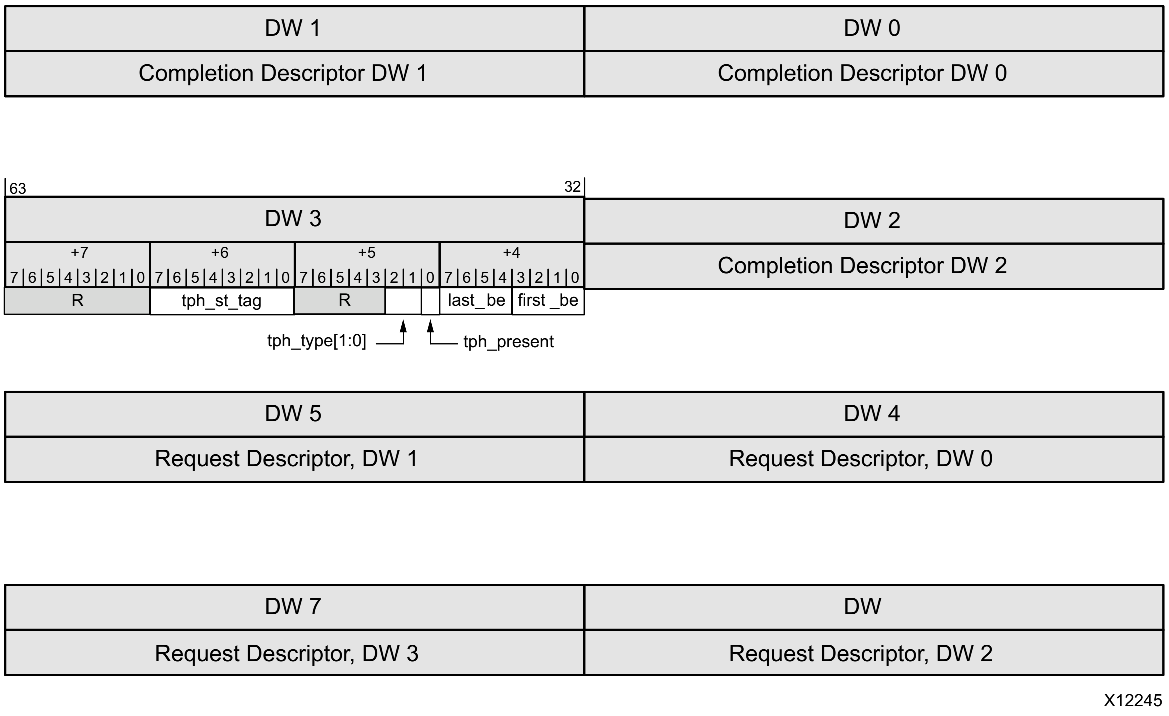


Figure 3-81: **Composition of the AXI-Stream Packet for UR and CA Completions**

Straddle Option on CC Interface

The PCIe 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 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 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.

Figure 3-82 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.

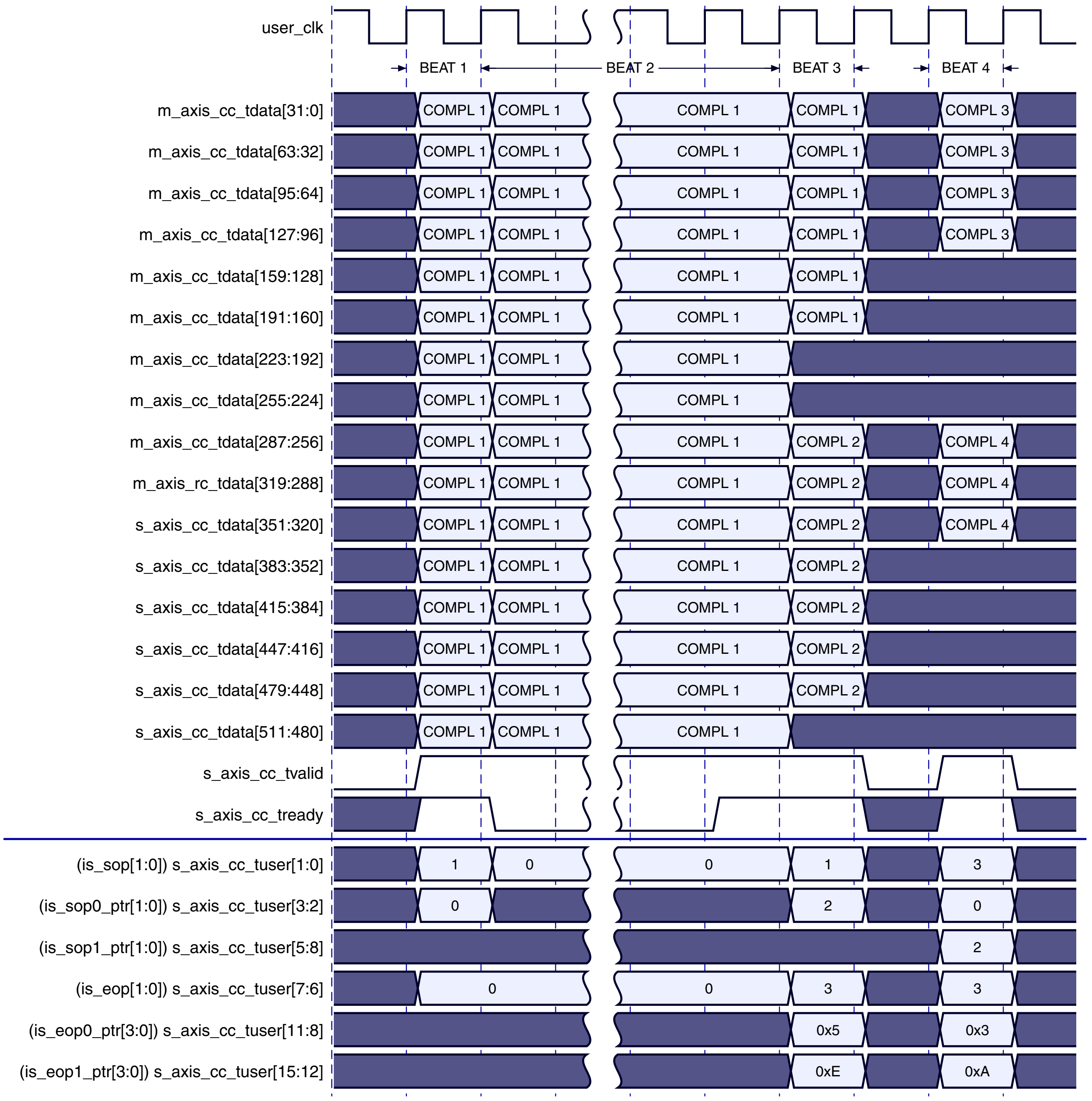


Figure 3-82: **Transfer of Completion TLPs on the Completer Completion Interface with the Straddle Option Enabled**

512-bit Requester Interface

This section describes the operation of the user-side Requester interface associated with the 512-bit AXI4- Stream Interface. Figure 3-66 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)

Figure 3-83 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.

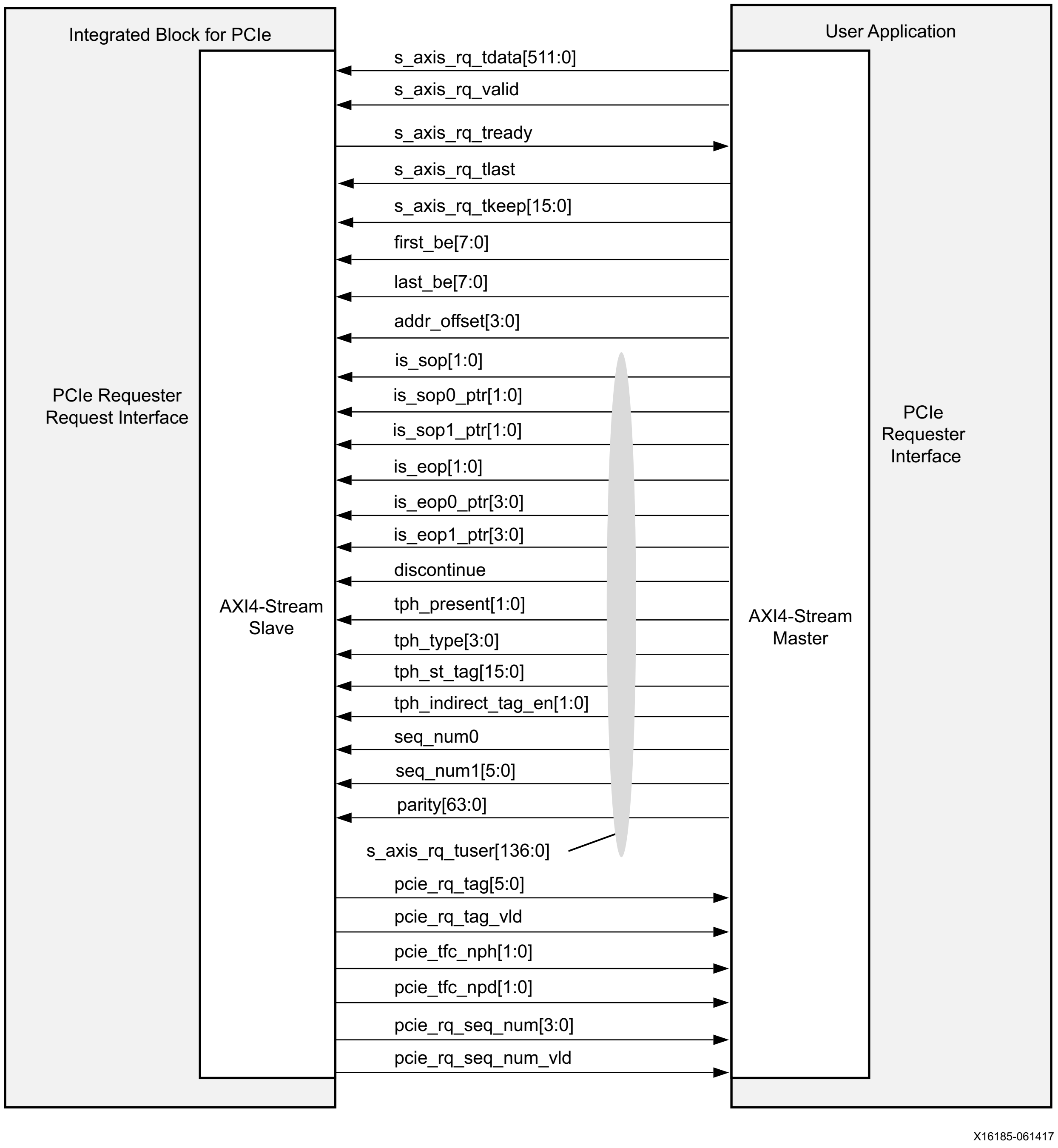


Figure 3-83: **Requester Request Interface Signals**

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 Figure 3-84, Figure 3-85, Figure 3-86 and Figure 3-87. The format of Figure 3-84 applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request. The format of Figure 3-85 is used for Vendor-Defined Messages (Type 0 or Type 1) only. The format of Figure 3-86 is used for all ATS messages (Invalid Request, Invalid Completion, Page Request, PRG Response). For all other messages, the descriptor takes the format of Figure 3-87.

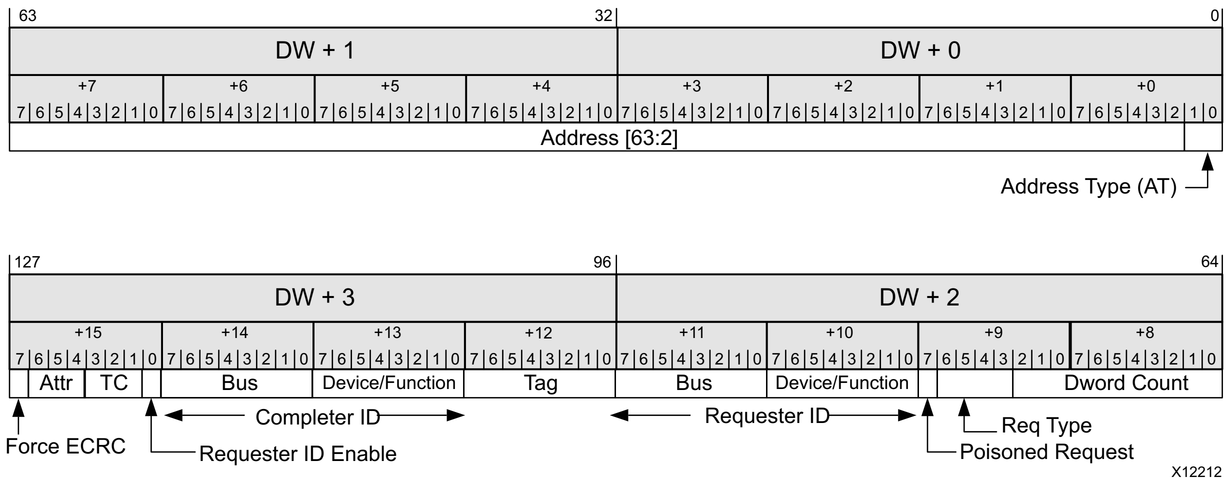


Figure 3-84: **Requester Request Descriptor Format for Memory, I/O, and Atomic Op Requests**

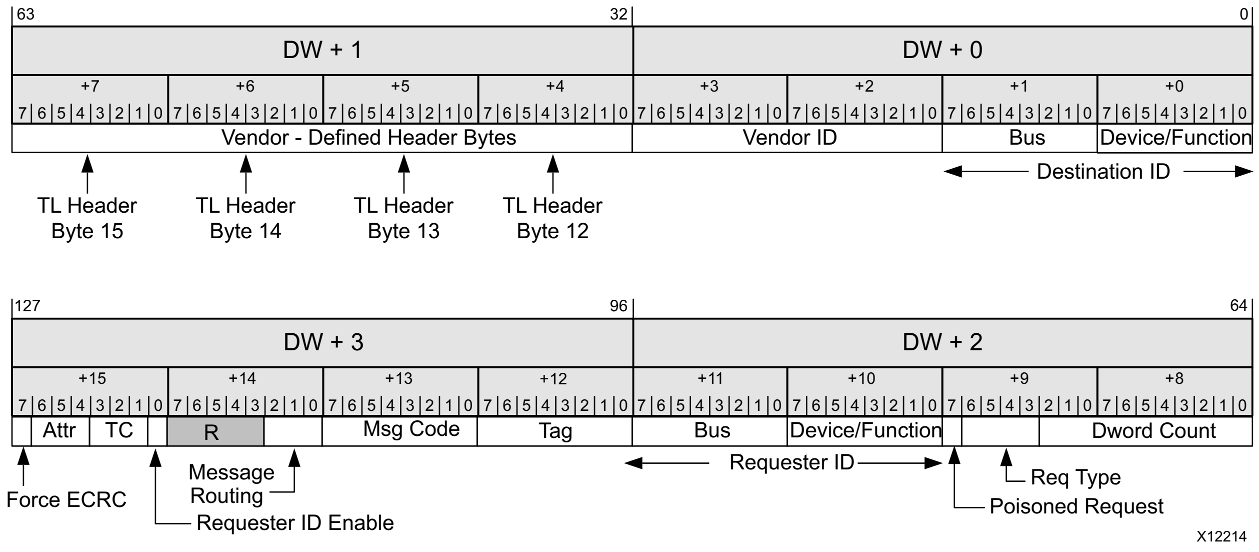


Figure 3-85: **Requester Request Descriptor Format for Vendor-Defined Messages**

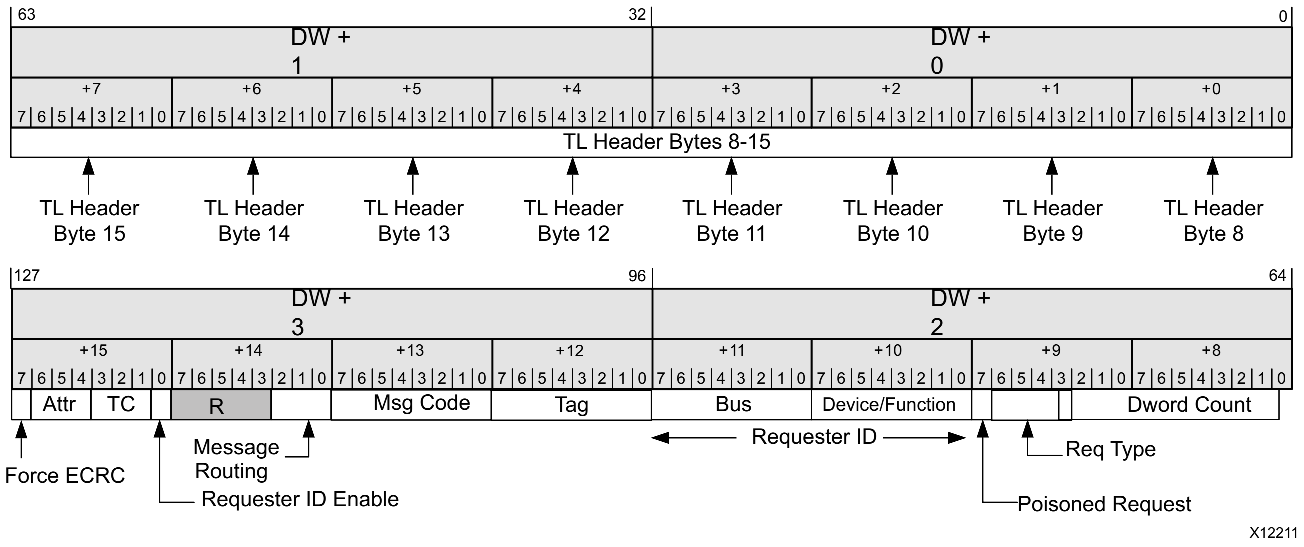


Figure 3-86: **Requester Request Descriptor Format for ATS Messages**

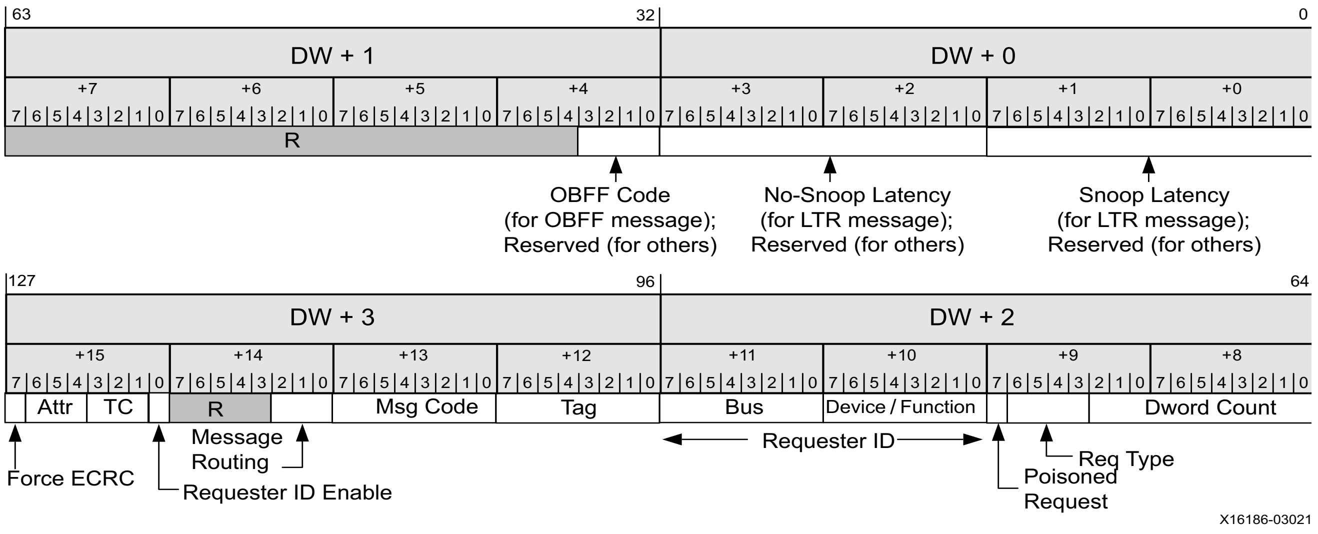


Figure 3-87: **Requester Request Descriptor Format for all other Messages**

The individual fields of the completer request descriptor are described in the following table.

Table 3-18: **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. 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 Table 2-19.

Table 3-18: **Requester Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
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.
87:80	Requester Function/ Device Number	Function number of the Requester Function. When ARI is in use, all 8 bits of this field must be set to the Function number. Otherwise, bits [84:82] must be set to the completer Function number. When ARI is not in use, and the core is configured as a Root Complex, the user logic must supply the 5-bit Device Number of the requester on bits [87:83]. When ARI is not in use, and the core is configured as an Endpoint, the user logic can optionally supply a 5-bit Device Number of the requester on bits [87:83]. The user logic must set the Requester ID Enable bit in the descriptor if a Device Number is supplied on bits [87:83]. This value is used by core when sending the Request TLP, instead of the stored value of the Device Number captured by the core from Configuration Requests.
85:88	Requester Bus Number	Bus number associated with the requester Function. When the core is configured as a Root Complex, the user logic must supply the 8-bit Bus Number of the requester in this field. When the core is configured as an Endpoint, the user logic can optionally supply a Bus Number in this field. The user logic must set the Requester ID Enable bit in the descriptor if a Bus Number is supplied in this field. This value is used by the core when sending the Request TLP, instead of the stored value of the Bus Number captured by the core from Configuration Requests.
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.).
120	Requester ID Enable	The purpose of this field is to enable the user logic to supply the bus and device numbers to be used in the Requester ID. This field is applicable only to Endpoint cores. If this field is 0, the core uses the captured values of the bus and device numbers to form the Requester ID. If this input is 1, the core uses the bus and device numbers supplied by the user logic in the descriptor to form the Requester ID.

Table 3-18: **Requester Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
123:121	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request.
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. 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.
111:104	Message Code	This field is defined for all messages. It contains the 8-bit Message Code to be set in the TL header. Appendix F of the PCI Express 3.0 Specifications provides a complete list of the supported Message Codes.
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 AXI-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.

Figure 3-88 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, *k* > 1.

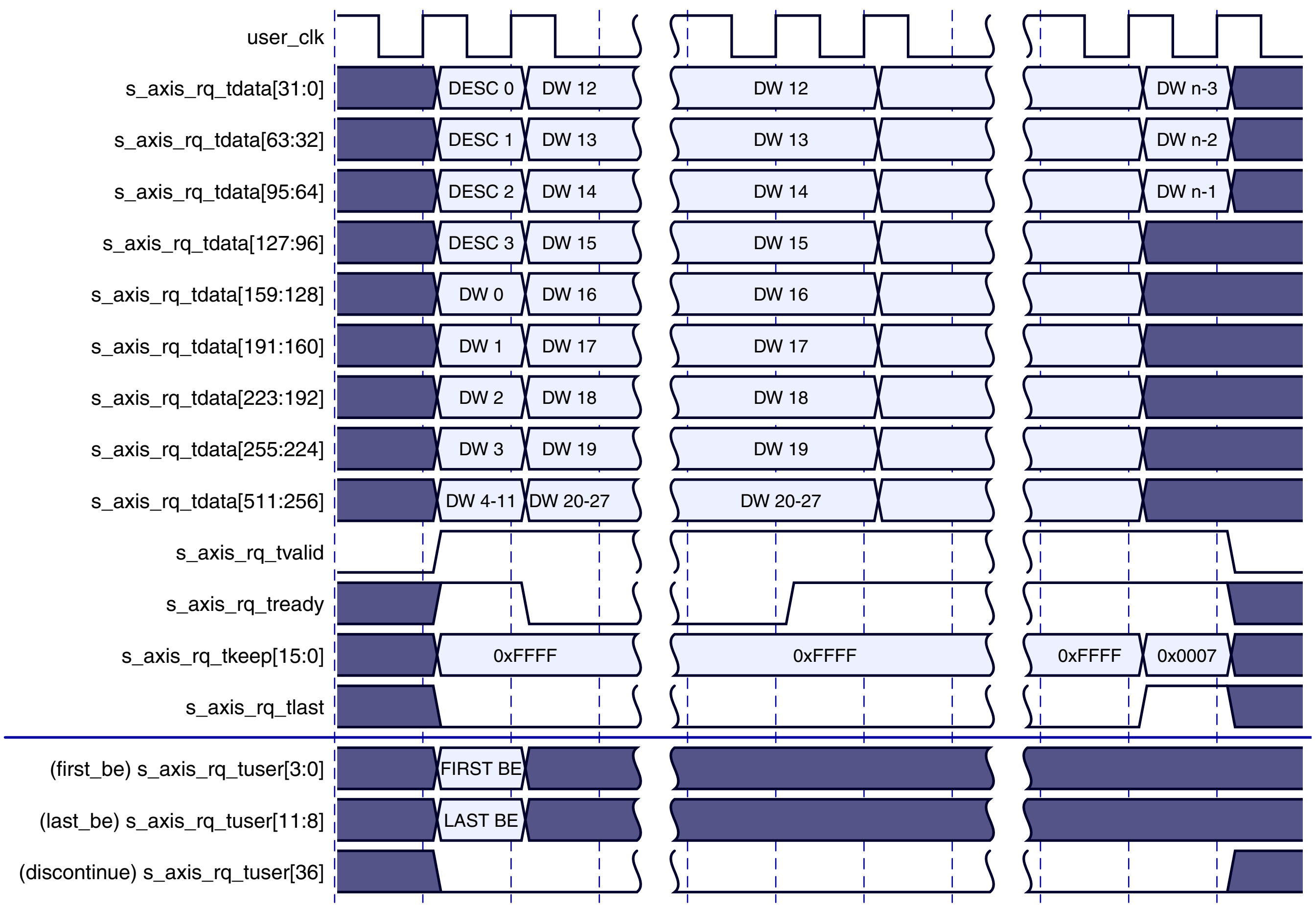


Figure 3-88: **Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode)**

Figure 3-89 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.

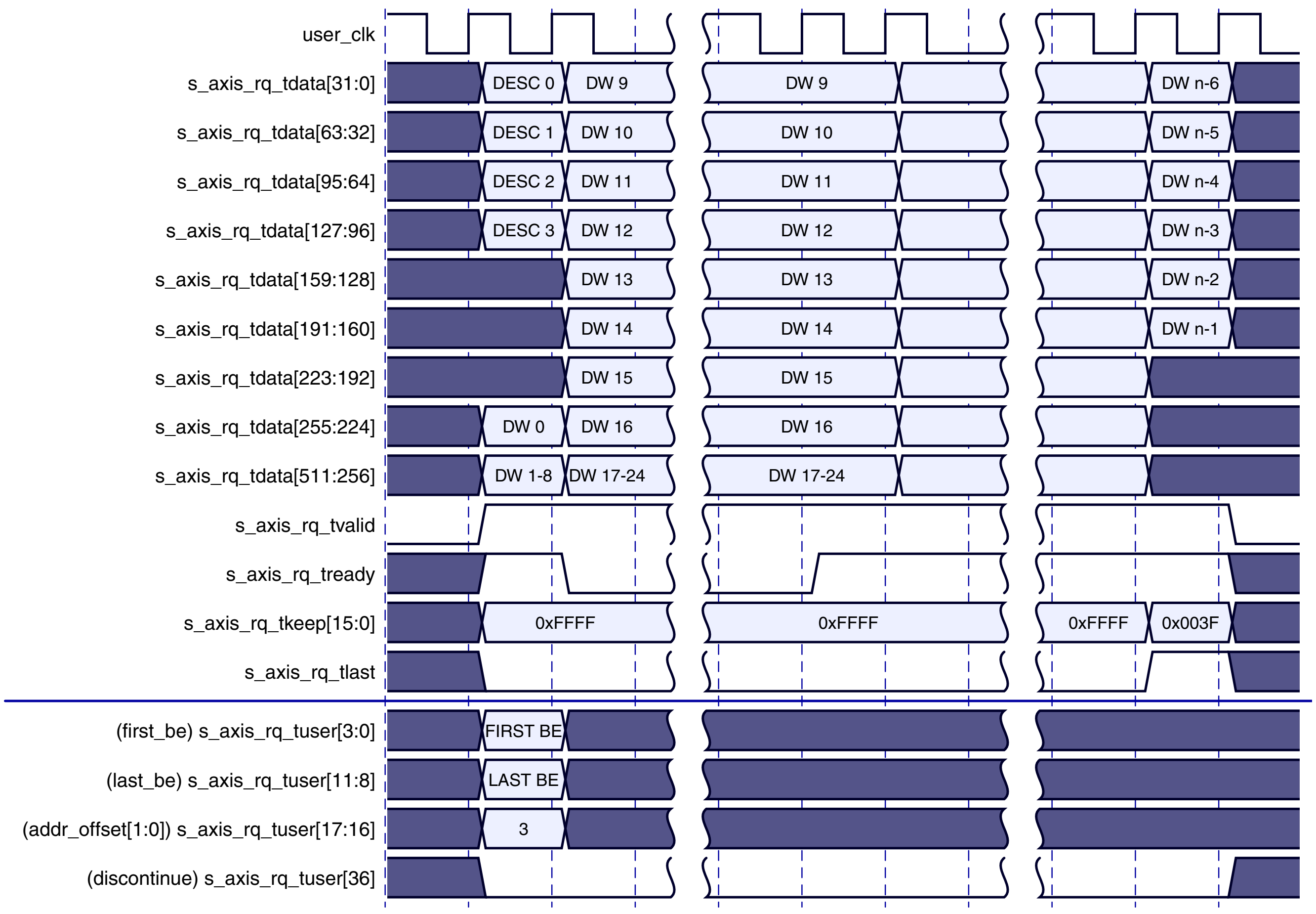


Figure 3-89: **Memory Write Transaction on the Requester Request Interface (128-bit Address Aligned Mode)**

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. Figure 3-90 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.

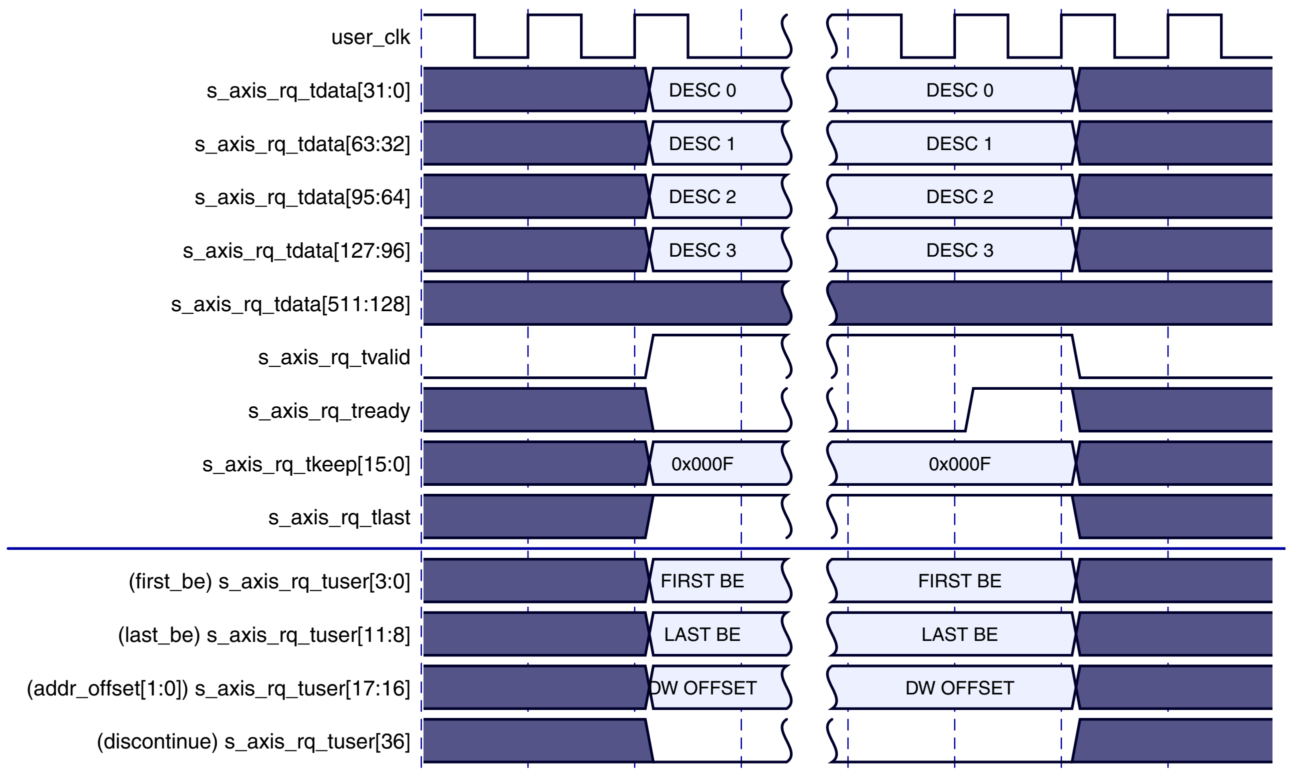


Figure 3-90: **Memory Read Transaction on the Requester Request Interface**

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.

Figure 3-91 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.

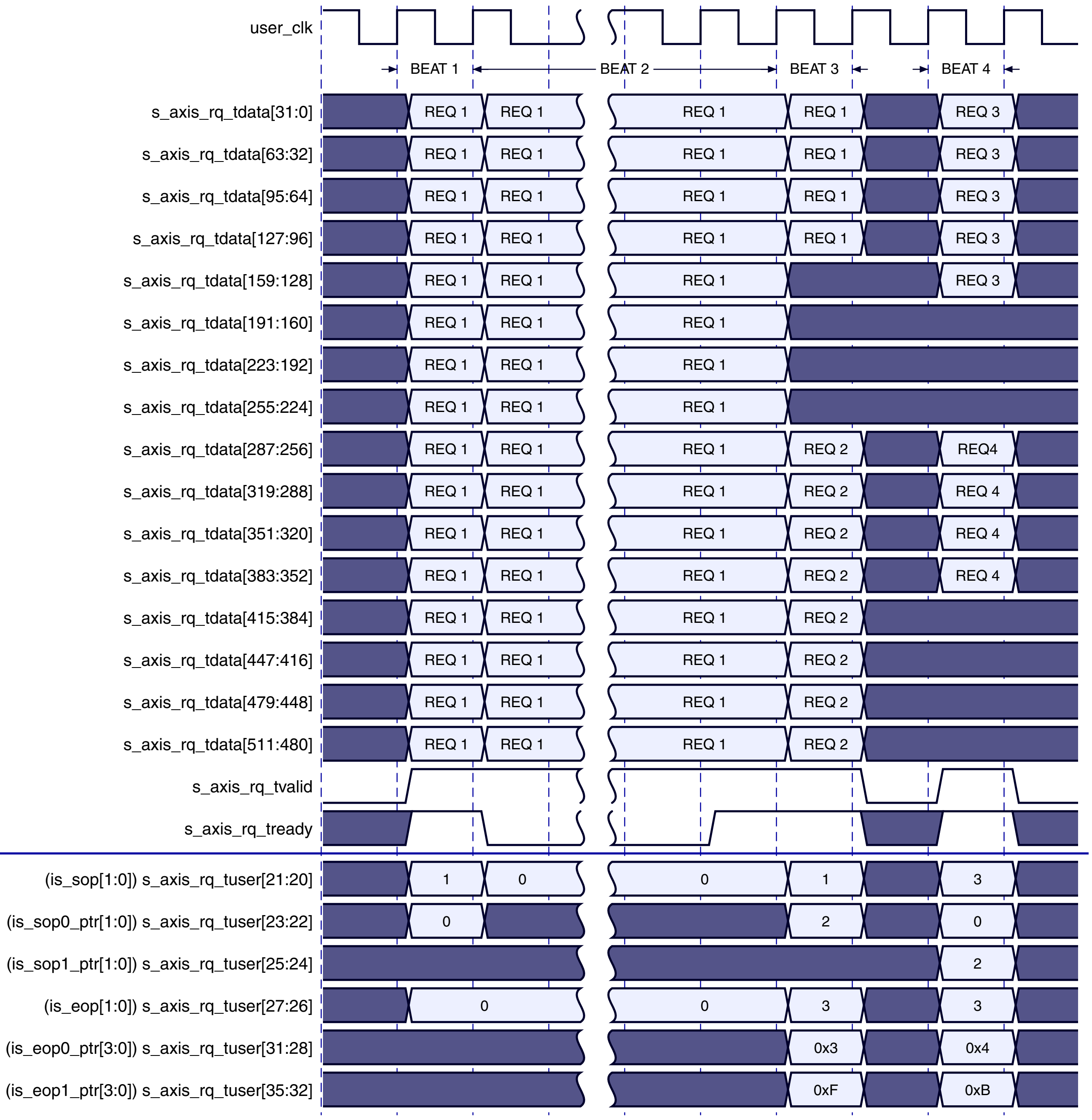


Figure 3-91: **Transfer of Request TLPs on the Requester Request Interface with the Straddle Option Enabled**

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 PCIe 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]`. When Straddle option is disabled, only `pcie_rq_tag0[7:0]` is used. The value on this bus is valid when the core asserts `pcie_rq_tag_vld0` and `pcie_rq_tag_vld1`. When Straddle option is disabled, only `pcie_rq_tag_vld0` is used. The user logic must copy this 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 are provide up to two allocated tags in the same cycle on this interface. The states of the signals `pcie_rq_tag` and `pcie_rq_tag` must be interpreted as follows:

- Assertion of `pcie_rq_tag_vld[0]` in any cycle indicates that the core has placed an allocated tag on `pcie_rq_tag[7:0]`.
- Simultaneous assertion of `pcie_rq_tag_vld[0]` and `pcie_rq_tag_vld[1]` in the same cycle indicates that the core has placed two allocated tags, the first on `pcie_rq_tag[7:0]` and the second on `pcie_rq_tag[15:8]`. The tag on `pcie_rq_tag[7:0]` corresponds to an earlier request sent by the user logic and the tag on `pcie_rq_tag[15:8]` corresponds to a later request.
- `pcie_rq_tag_vld[1]` is never asserted when `pcie_rq_tag_vld[0]` is not asserted. That is, when there is only one tag to communicate in any cycle, it is always communicated on

- When straddle is not in use, only a single tag can be communicated in any cycle, and `pcie_rq_tag_vld[1]` 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. Figure 3-92 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.

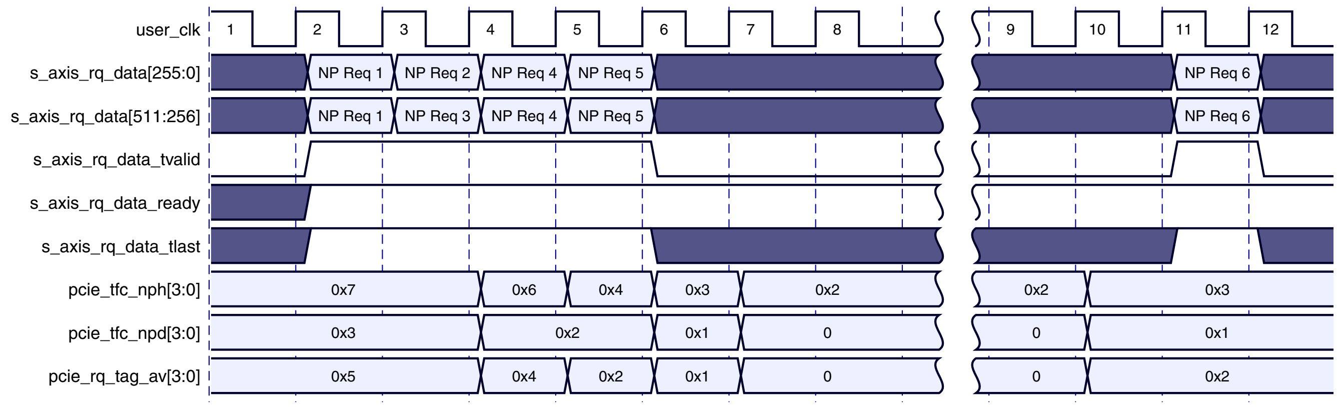


Figure 3-92: **Operation of credit and tag availability signals on the Requester Request Interface**

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)

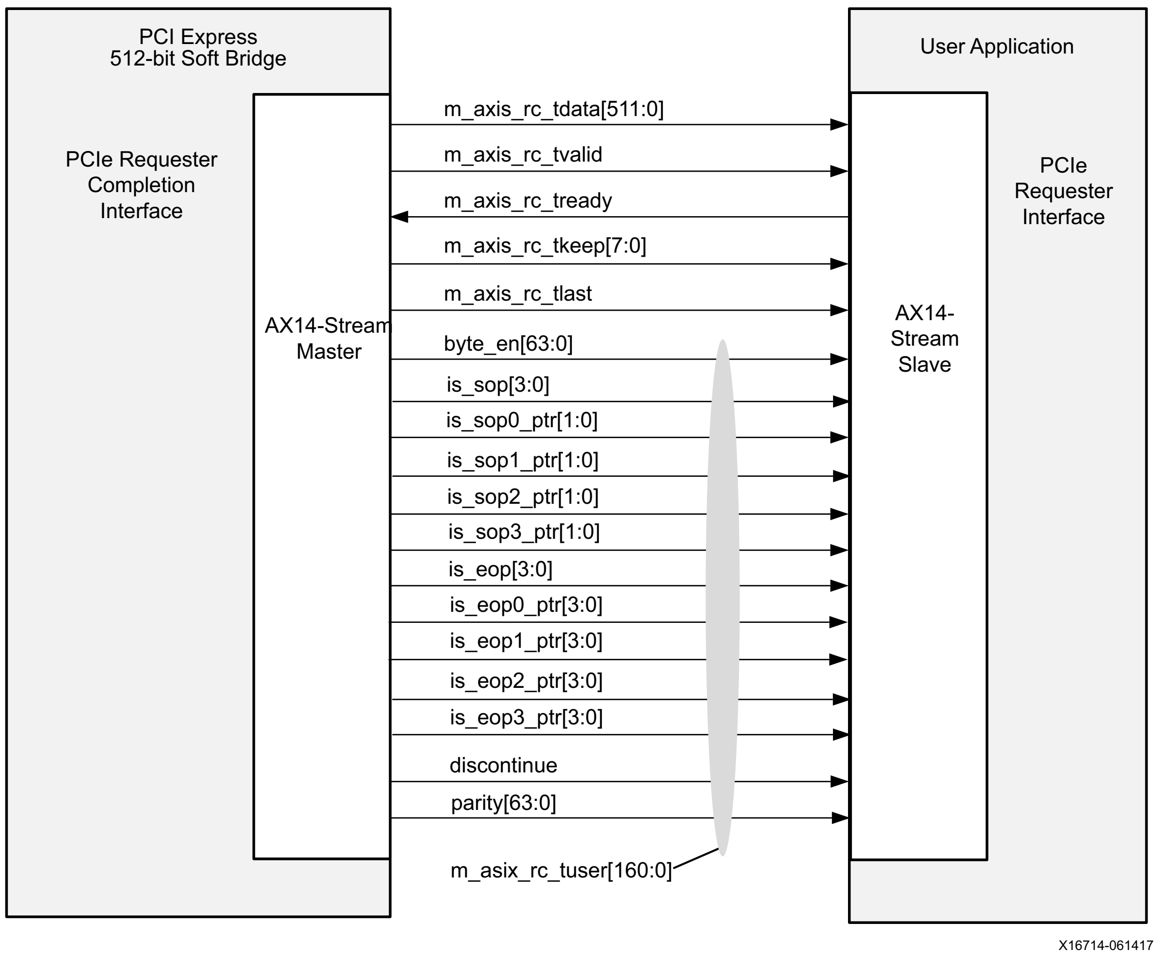


Figure 3-93: **Requester Completion Interface Signals**

Figure 3-93 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 the 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 Figure 3-94. The individual fields of the requester completion descriptor are described in Figure 3-19.

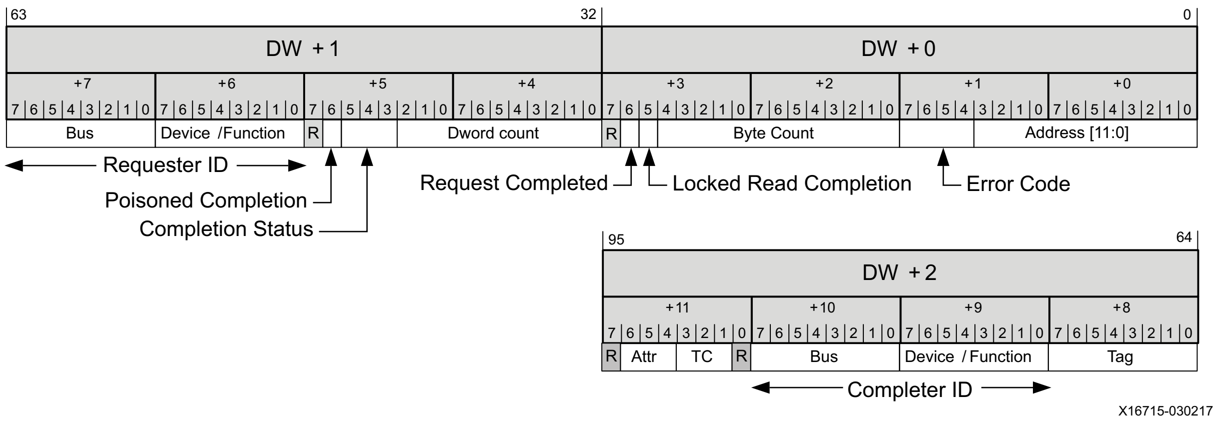


Figure 3-94: **Requester Completion Descriptor Format**

Table 3-19: **Requester Completion Descriptor Fields**

Bit Index	Field Name	Description
11:0	Lower Address	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. When the Completion delivered has an error, only bits [6:0] of the address should be considered valid. This is a byte-level address.
15:12	Error Code	Completion error code. These three bits encode error conditions detected from error checking performed by the core on received Completions. Its encodings are: • 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* 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. • 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.

Table 3-19: **Requester Completion Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
28:16	Byte Count	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. 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.
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	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: • 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.

Table 3-19: **Requester Completion Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
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

Figure 3-95 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.

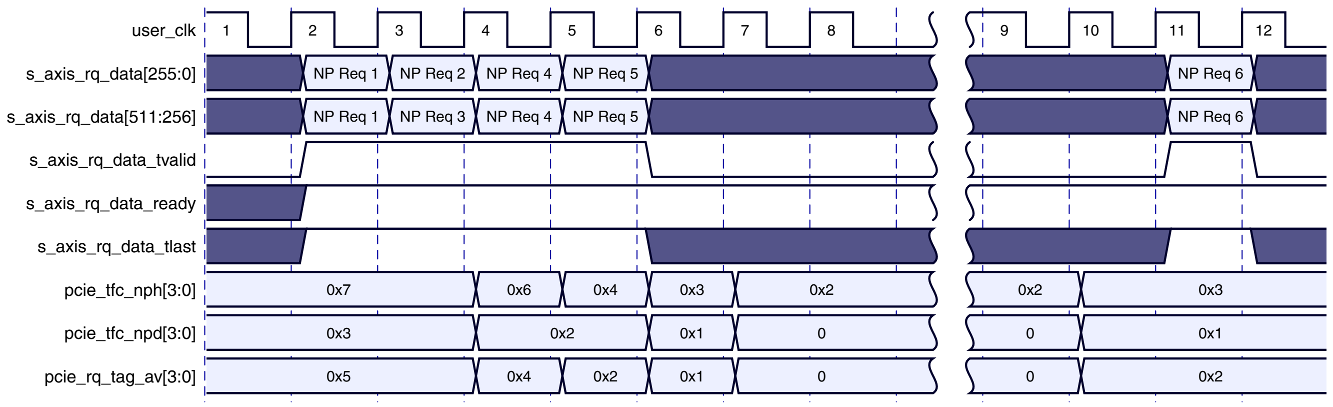


Figure 3-95: **Transfer of a Completion with no Data on the Requester Completion Interface**

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 AXI-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

Figure 3-96 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* Dwords, 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.

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 AXI-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 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 Dwords or less. For Completion payloads of two Dwords 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.

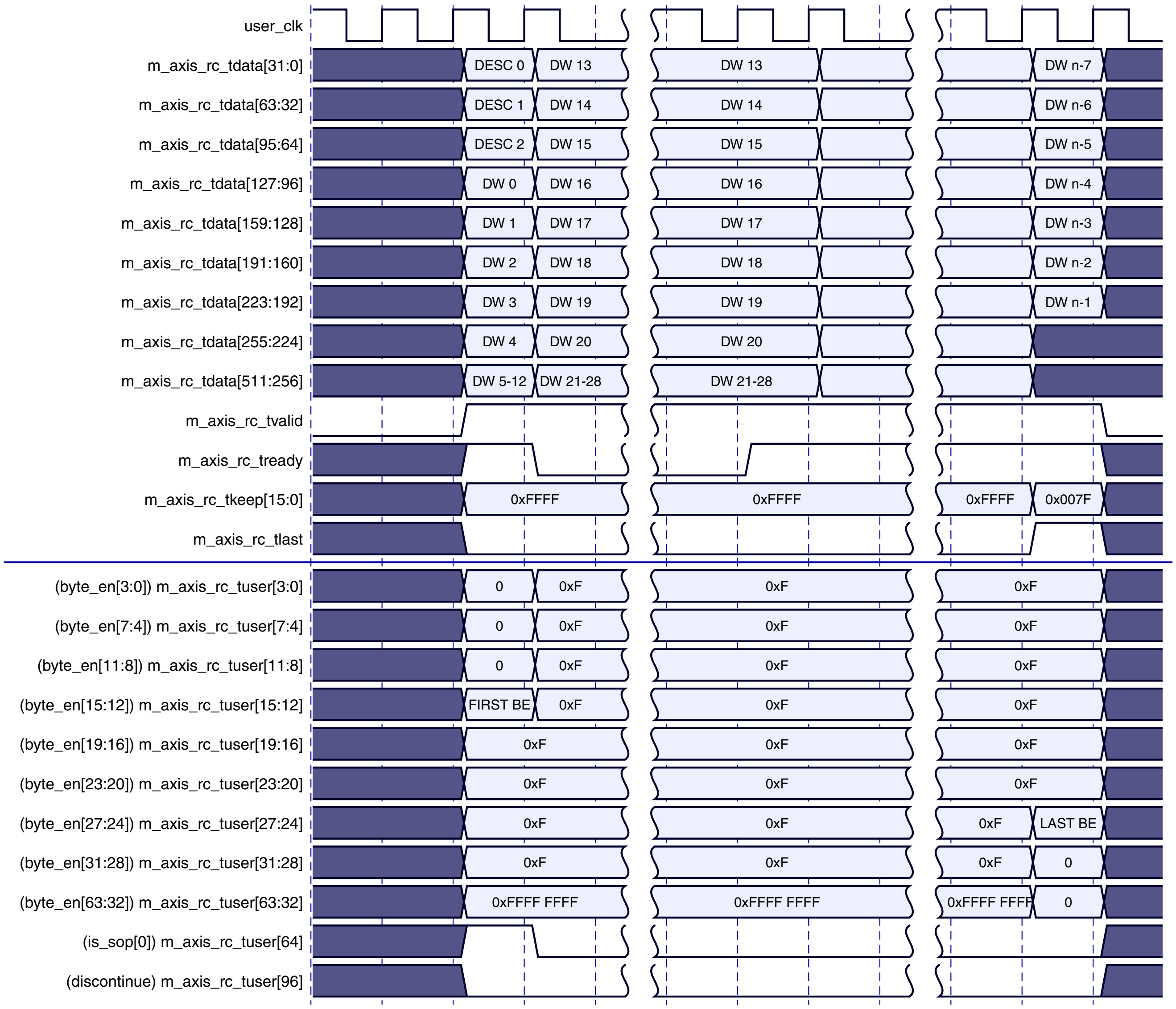


Figure 3-96: **Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode)**

Figure 3-97 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.

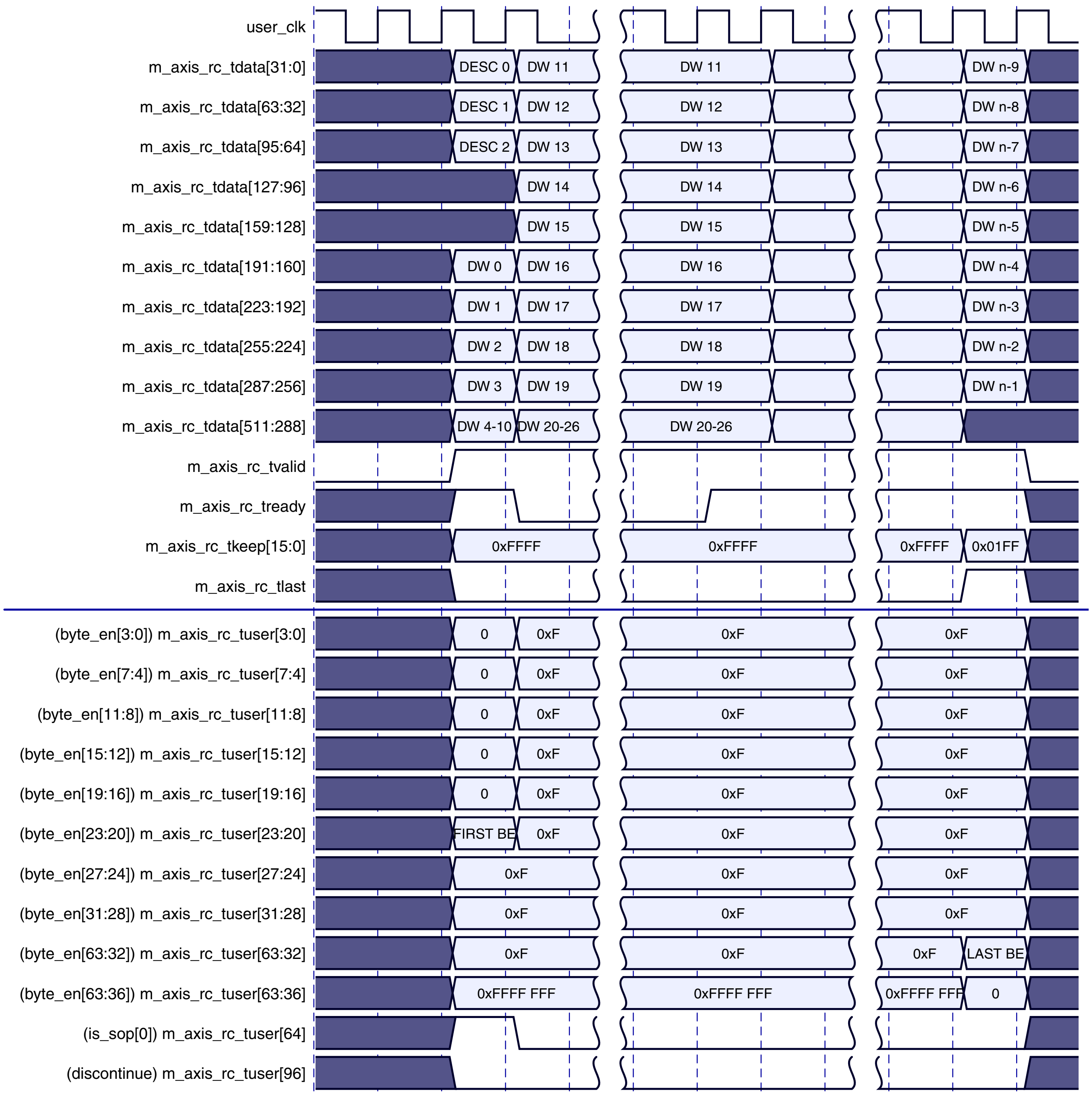


Figure 3-97: **Transfer of a Completion with Data on the Requester Completion Interface (128-bit Address Aligned Mode)**

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.

Figure 3-98 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.

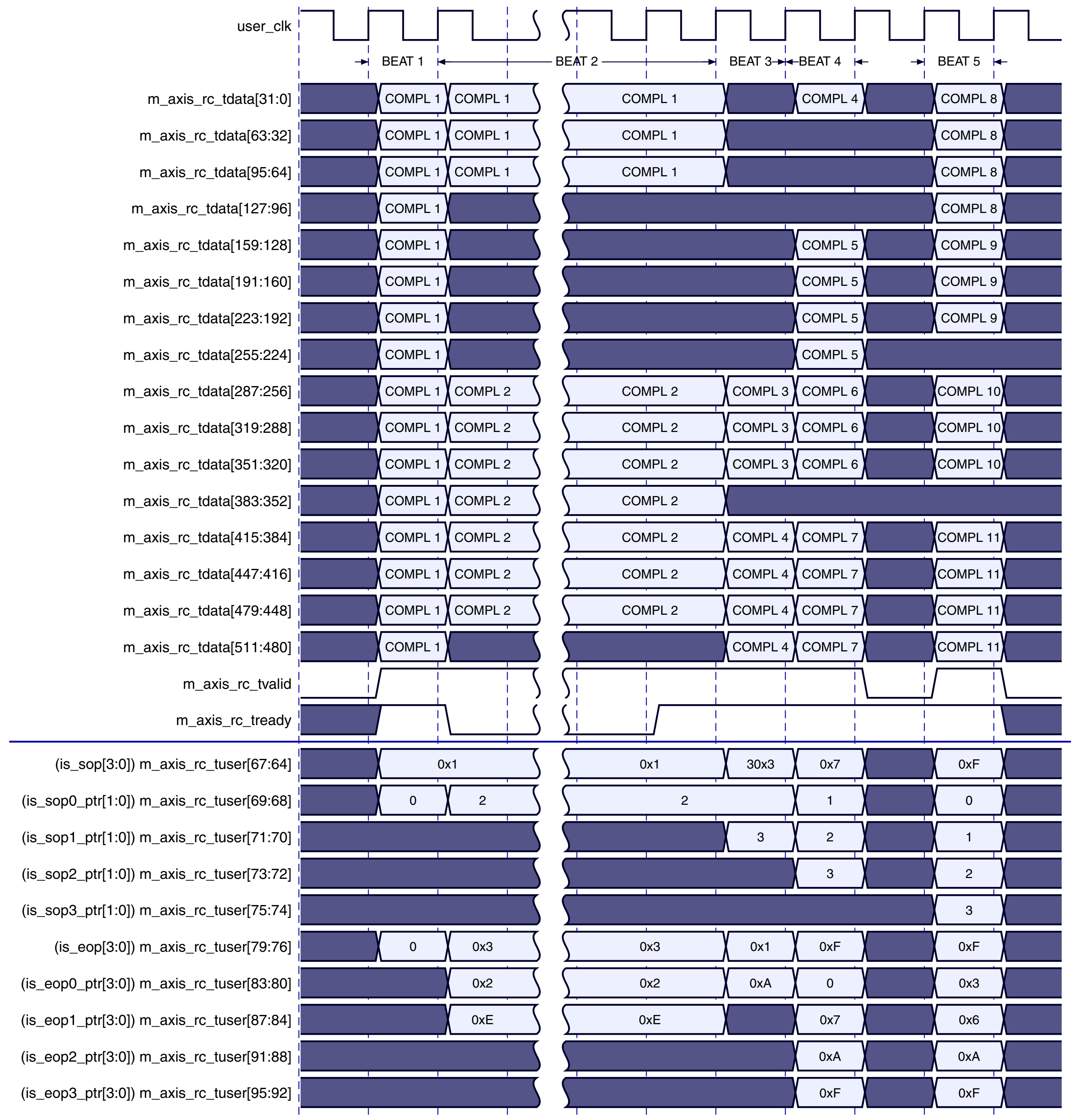


Figure 3-98: **Transfer of Completion TLPs on the Requester Completion Interface with the Straddle Option Enabled**

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:

- 0000: No errors detected.
- 0001: The Completion TLP received from the link was Poisoned. 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 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 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 corresponding to parity of one byte in the 512-bit AXIS `tdata`. There are 64-bit parity bits corresponding to the 512-bit AXIS `tdata` and are valid based on the AXIS `tkeep[15:0]`. For example, if `tkeep[15:0]` = 0x0001 then only lower 4 parity bits are valid. If `tkeep[15:0]` = 0xFFFF then all 64 parity bits are valid.

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* [Ref 2].

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.

   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.
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` (see Table 3-20) 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 3-20: **Power Management Handshaking Signals**

Port Name	Direction	Description
cfg_power_state_change_interrupt	Output	Asserted if a power-down request TLP is received from the upstream device. After assertion, `cfg_power_state_change_interrupt` remains asserted until the user application asserts `cfg_power_state_change_ack`.
cfg_power_state_change_ack	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`.

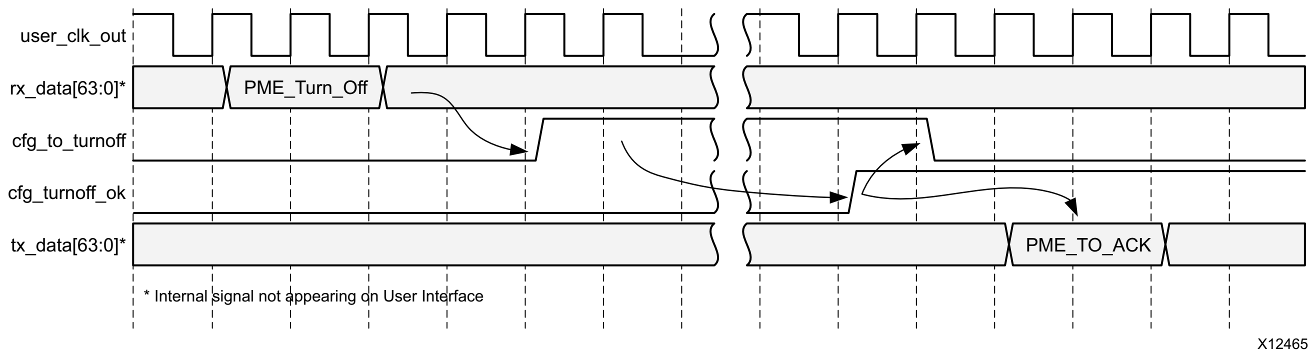


Figure 3-99: **Power Management Handshaking: 64-Bit**

Generating Interrupt Requests

See the `cfg_interrupt_msi*` and `cfg_interrupt_msix_*` descriptions in Table 2-31, page 65.

Note: This section only applies to the Endpoint Configuration of the UltraScale+ Devices Integrated Block for PCIe core.

The integrated block 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. For more information on the MSI and MSI-X capability structures, see section 6.8 of the *PCI Local Base Specification v3.0*.

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. Table 3-21 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 3-21: **Interrupt Modes**

	cfg_interrupt_msixenable=0	**cfg_interrupt_msixenable=1**
cfg_interrupt_ msi_enable=0	Legacy Interrupt (INTx) mode. The cfg_interrupt interface only sends INTx messages.	MSI-X mode. MSI-X interrupts can be generated using the `cfg_interrupt` interface.
cfg_interrupt_ msi_enable=1	MSI mode. The cfg_interrupt interface only sends MSI interrupts (MWr TLPs).	Undefined. System software is not supposed to permit this. However, the `cfg_interrupt` interface is active and sends MSI interrupts (MWr 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 Table 2-24.

The user application requests interrupt service in one of two ways, each of which are 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). On the following clock cycle, 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 until the interrupt has been serviced and should be set along with the `cfg_interrupt_int` signal assertion, otherwise the interrupt status bit in the status register is not updated correctly. 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.

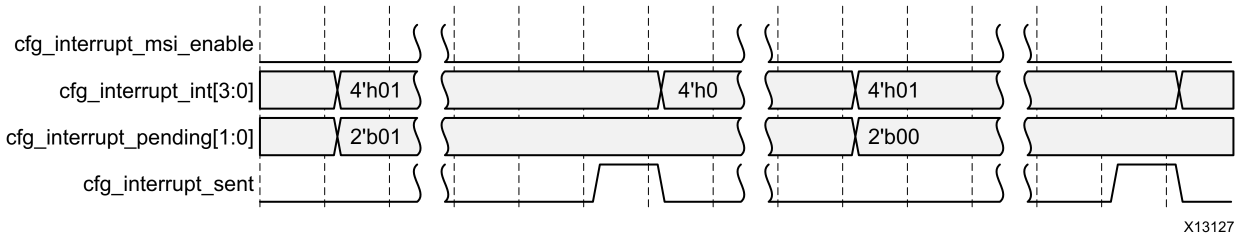


Figure 3-100: **Legacy Interrupt Signaling**

MSI Mode

The user application first asserts a value on `cfg_interrupt_msi_int`, as shown in Figure 3-100. The core asserts `cfg_interrupt_msi_sent` to indicate that the interrupt is accepted and the core sends an MSI Memory Write TLP.

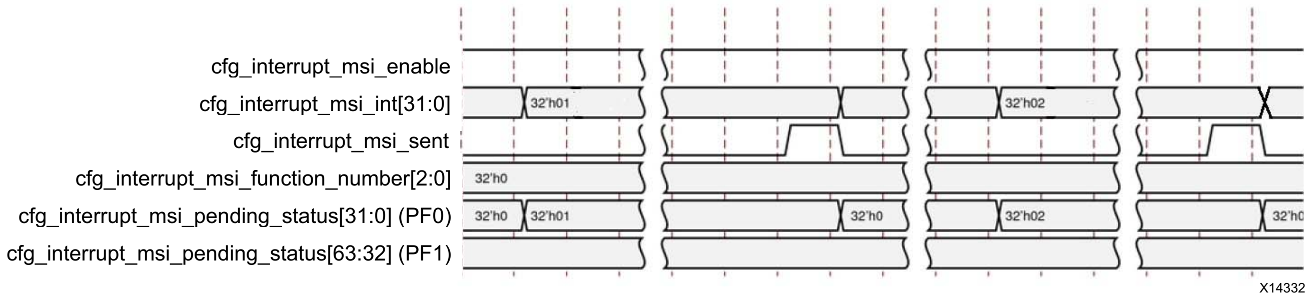

cfg_interrupt_msi_enable
cfg_interrupt_msi_int[31:0]
cfg_interrupt_msi_sent
cfg_interrupt_msi_function_number[2:0]
cfg_interrupt_msi_pending_status[31:0] (PF0)
cfg_interrupt_msi_pending_status[63:32] (PF1)
32'h01
32'h02
32'h0
32'h0
32'h01
32'h0
32'h02
32'h0
X14332

Figure 3-101: **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 UltraScale+ Devices Integrated Block for PCIe core supports the MSI-X interrupt and its signaling, which is shown in Figure 3-102. 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.

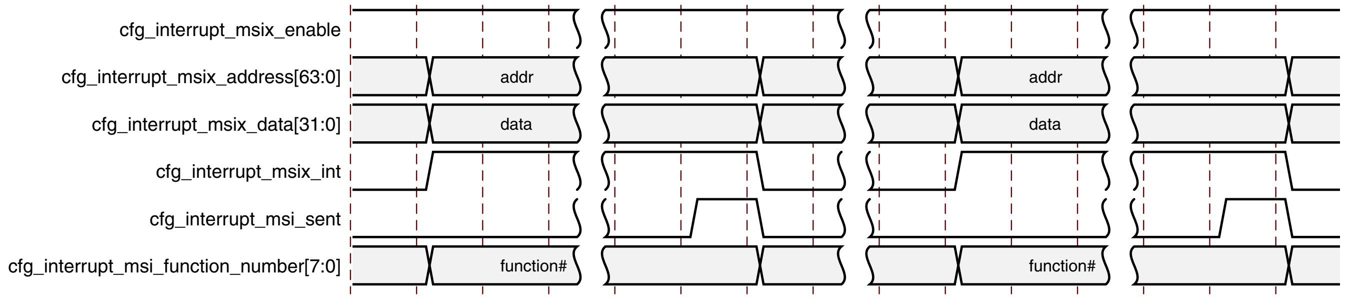

cfg_interrupt_msix_enable
cfg_interrupt_msix_address[63:0]
cfg_interrupt_msix_data[31:0]
cfg_interrupt_msix_int
cfg_interrupt_msi_sent
cfg_interrupt_msi_function_number[7:0]
addr
data
function#
addr
data
function#

Figure 3-102: **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 Figure 3-103, the user application first asserts `cfg_interrupt_msix_int` with the vector number set in `cfg_interrupt_msi_int`.
- The core asserts `cfg_interrupt_msi_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.

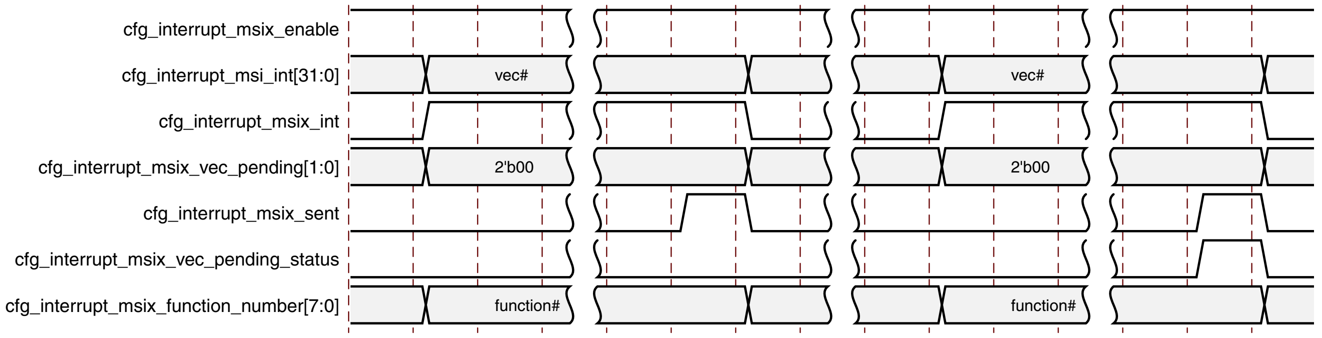

cfg_interrupt_msix_enable
cfg_interrupt_msi_int[31:0]
cfg_interrupt_msix_int
cfg_interrupt_msix_vec_pending[1:0]
cfg_interrupt_msix_sent
cfg_interrupt_msix_vec_pending_status
cfg_interrupt_msix_function_number[7:0]
vec#
2'b00
function#
vec#
2'b00
function#

Figure 3-103: **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 Figure 3-104.
- In the query and clear cases, `cfg_interrupt_msix_vec_pending_status` reflects the pending status before the query or clear.

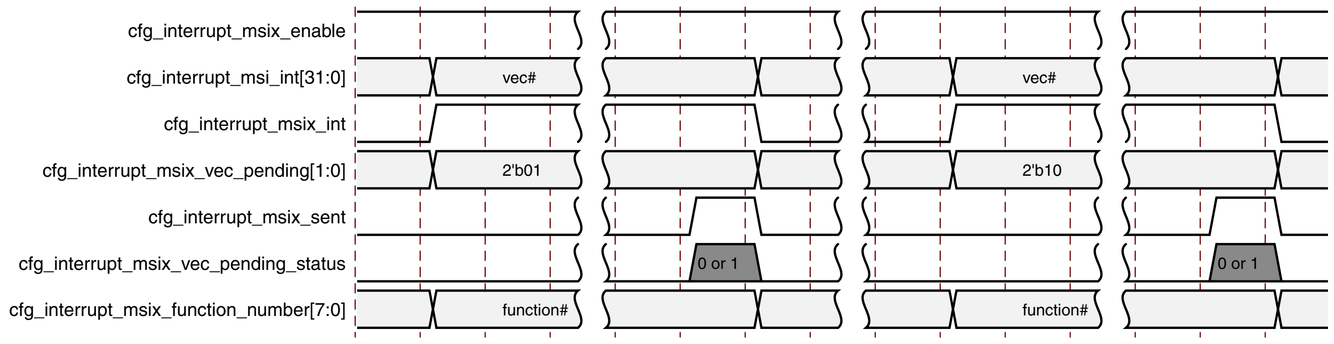

cfg_interrupt_msix_enable
cfg_interrupt_msi_int[31:0]
cfg_interrupt_msix_int
cfg_interrupt_msix_vec_pending[1:0]
cfg_interrupt_msix_sent
cfg_interrupt_msix_vec_pending_status
cfg_interrupt_msix_function_number[7:0]
vec#
2'b01
0 or 1
function#
vec#
2'b10
0 or 1
function#

Figure 3-104: **MSI-X Pending Bit Array Query and Clear**

Note: Applications that need to generate MSI/MSIX 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 Table 3-22) 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 3-23). 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 Table 3-23. 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 3-22: **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

Table 3-22: **Message Type Encoding on Receive Message Interface *(Cont'd)***

cfg_msg_received_type[4:0]	Message Type
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 3-23: **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 3-23: **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

Figure 3-105 is a 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

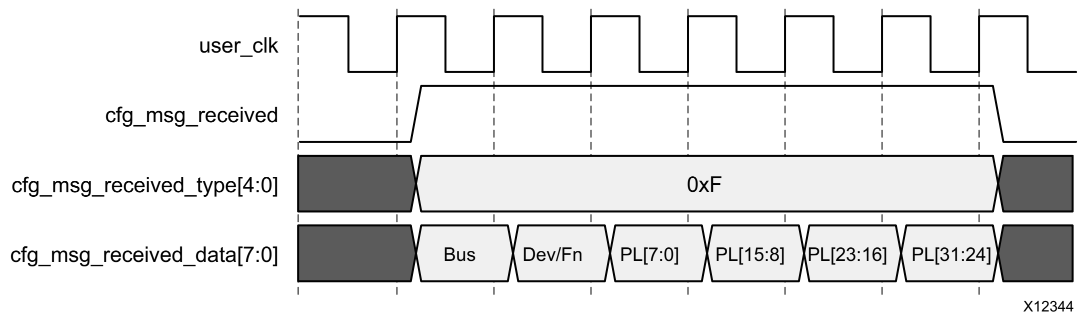


Figure 3-105: **Receive Message Interface**

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 Table 2-24, page 47. 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. Figure 3-106 illustrates an example with read from the Configuration Space.

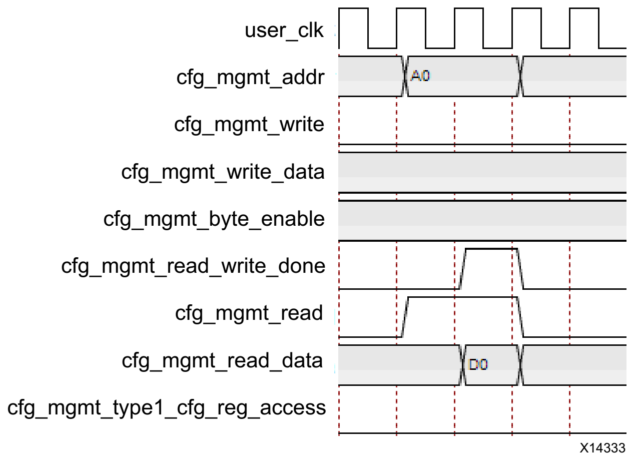


Figure 3-106: **cfg_mgmt_read_type0_type1**

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.

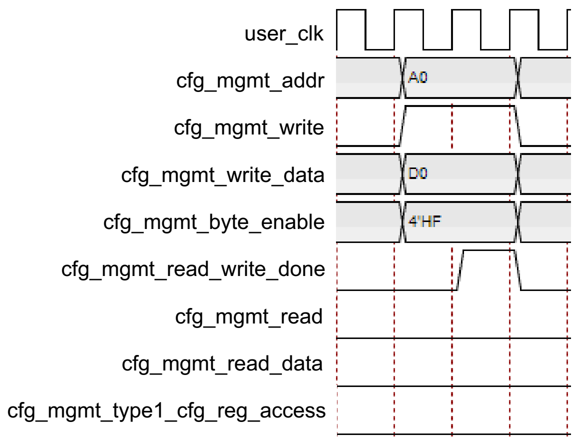


Figure 3-107: **cfg_mgmt_write_type0**

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.

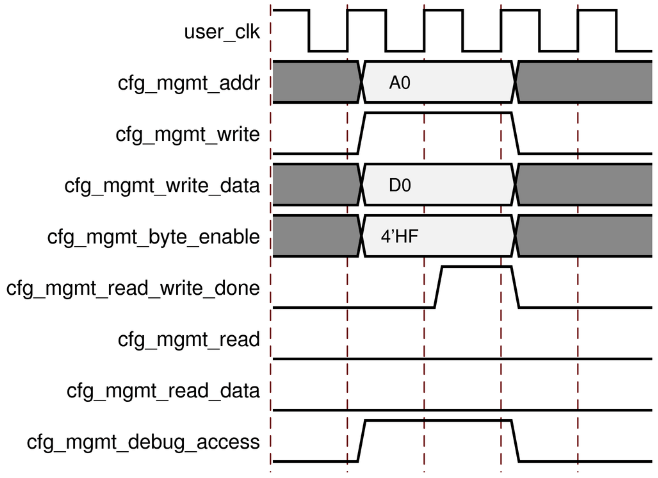


Figure 3-108: **cfg_mgmt_debug_access**

Link Training: 2-Lane, 4-Lane, 8-Lane, and 16-Lane Components

The 2-lane, 4-lane, and 8-lane core can operate at less than the maximum lane width as required by the *PCI Express Base Specification* [Ref 2]. Two cases cause 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 Figure 3-109. 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.

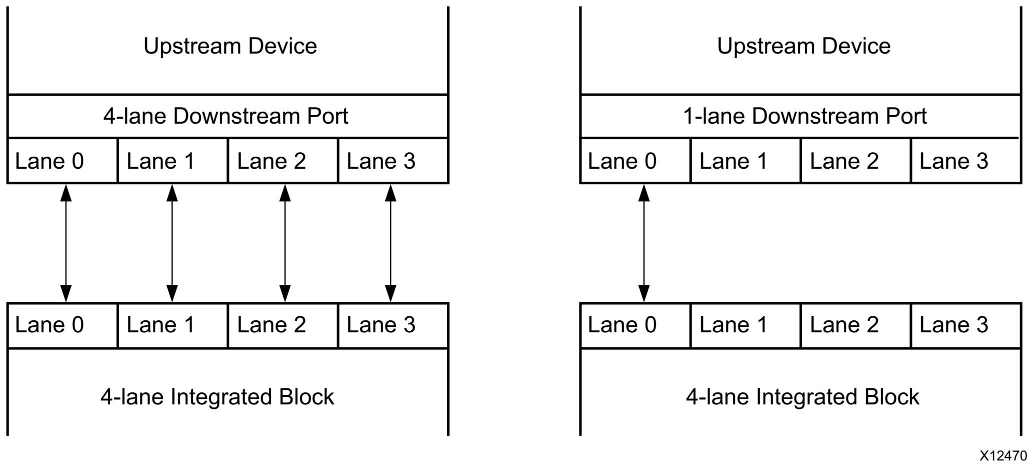


Figure 3-109: **Scaling of 4-Lane Endpoint Block from 4-Lane to 1-Lane Operation**

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. Table 3-24 describes the several possible combinations including downshift modes and availability of lane reversal support.

Table 3-24: **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:

1. 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.

Chapter 4

Design Flow Steps

This chapter 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) [Ref 16]
- *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 15]
- *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 17]
- *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 19]

Customizing and Generating the Core

This section includes information about using Xilinx tools to customize and generate the core in the Vivado Design Suite.

IMPORTANT: *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) [Ref 16] 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 you can run the `validate_bd_design` command in the Tcl console.*

You can customize the core 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 Vivado 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) [Ref 15], and the *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 17]

Note: Figures in this chapter are illustrations of the Vivado Integrated Design Environment (IDE). This layout might vary from the current version.

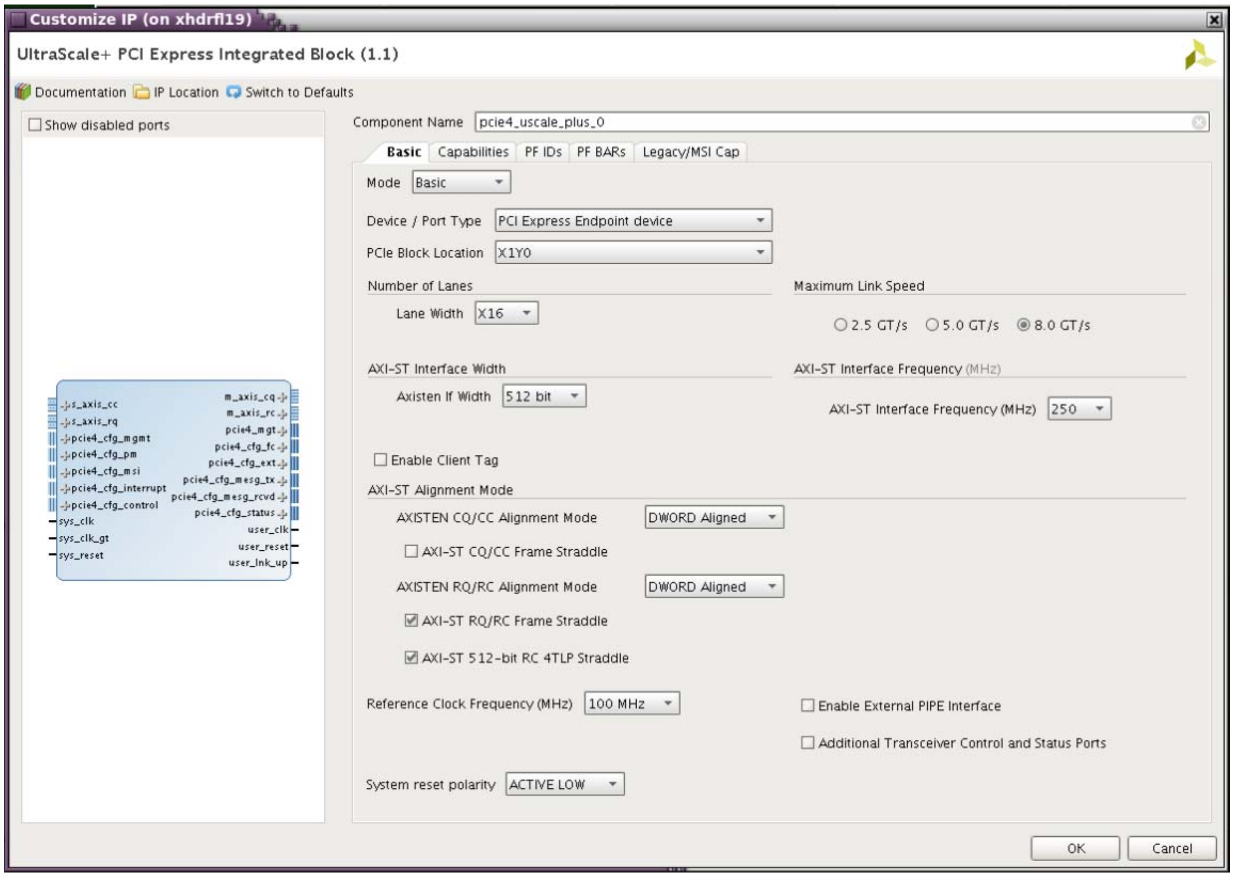
The Customize IP dialog box for the UltraScale+ Devices Integrated Block for PCIe core consists of two modes: Basic Mode Parameters and Advanced Mode Parameters. To select a mode, use the **Mode** drop-down list on the first page of the Customize IP dialog box. The following sections explain the parameters available in each of these modes.

Basic Mode Parameters

The **Basic** mode parameters are explained in this section.

Basic Tab

Figure 4-1 shows the initial customization page, used to define the core basic parameters.



UltraScale+ PCI Express Integrated Block (1.1)

Component Name: pcie4_uscale_plus_0

Mode: Basic

Device / Port Type: PCI Express Endpoint device

PCIe Block Location: X1Y0

Number of Lanes

Lane Width: X16

Maximum Link Speed

2.5 GT/s [✓ 8.0 GT/s] 5.0 GT/s

AXI-ST Interface Width

Axisten If Width: 512 bit

AXI-ST Interface Frequency (MHz)

AXI-ST Interface Frequency (MHz): 250

☐ Enable Client Tag

AXI-ST Alignment Mode

AXISTEN CQ/CC Alignment Mode: DWORD Aligned

☐ AXI-ST CQ/CC Frame Straddle

AXISTEN RQ/RC Alignment Mode: DWORD Aligned

☑ AXI-ST RQ/RC Frame Straddle

☑ AXI-ST 512-bit RC 4TLP Straddle

Reference Clock Frequency (MHz): 100 MHz

☐ Enable External PIPE Interface

☐ Additional Transceiver Control and Status Ports

System reset polarity: ACTIVE LOW

Figure 4-1: **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. Table 4-1 defines the available widths and associated generated core. Wider lane width cores can train down to smaller lane widths if attached to a smaller lane-width device. See Link Training: 2-Lane, 4-Lane, 8-Lane, and 16-Lane Components for more information.

Table 4-1: **Lane Width and Product Generated**

Lane Width	Product Generated
16	16-Lane UltraScale+ Device Integrated Block for PCI Express

Maximum Link Speed

The core allows you to select the Maximum Link Speed supported by the device. Table 4-2 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.

Table 4-2: **Lane Width and Link Speed**

Lane Width	Link Speed
x16	8 Gb/s

AXI-ST Interface Width

The core allows you to select the Interface Width, as defined in Table 4-3. The default interface width set in the Customize IP dialog box is the lowest possible interface width.

Table 4-3: **Lane Width, Link Speed, and Interface Width**

Lane Width	Link Speed (Gb/s)	Interface Width (Bits)
x16	8.0	512

AXI-ST Interface Frequency

The frequency is set to 250 MHz.

Enable Client Tag

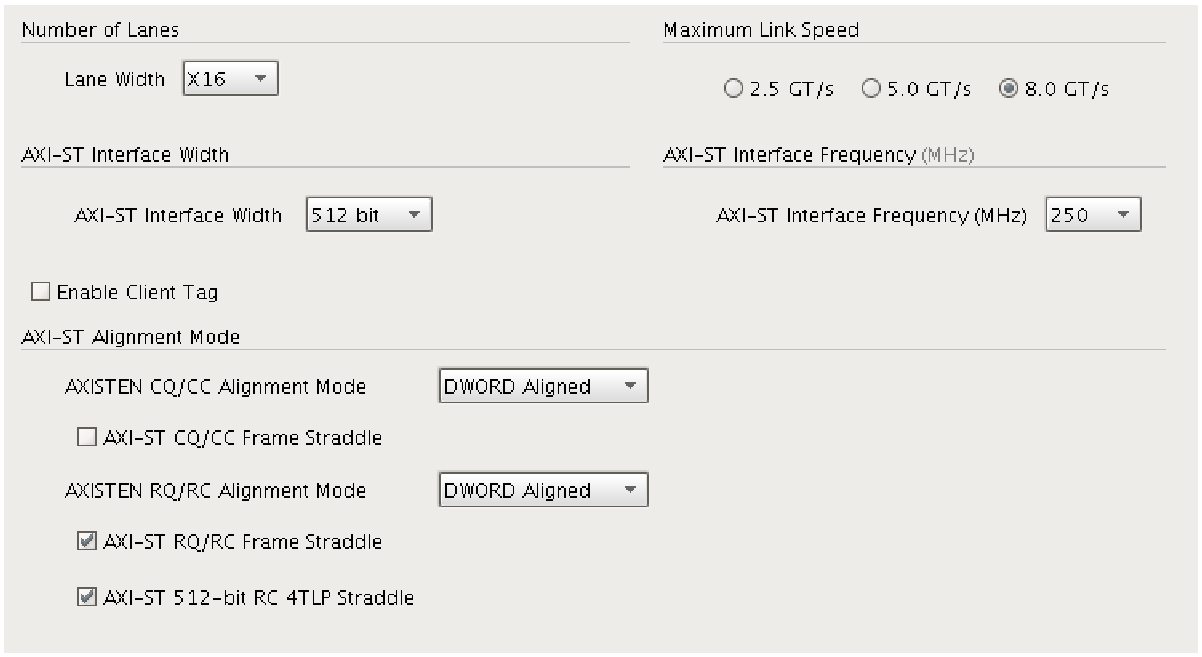
Enables you to use the client tag.

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, page 118.

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, page 184.



Number of Lanes

Lane Width: X16

Maximum Link Speed

( ) 2.5 GT/s ( ) 5.0 GT/s (●) 8.0 GT/s

AXI-ST Interface Width

AXI-ST Interface Width: 512 bit

AXI-ST Interface Frequency (MHz)

AXI-ST Interface Frequency (MHz): 250

☐ Enable Client Tag

AXI-ST Alignment Mode

AXISTEN CQ/CC Alignment Mode: DWORD Aligned

☐ AXI-ST CQ/CC Frame Straddle

AXISTEN RQ/RC Alignment Mode: DWORD Aligned

☑ AXI-ST RQ/RC Frame Straddle

☑ AXI-ST 512-bit RC 4TLP Straddle

Figure 4-2: **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 XAPP1184 [Ref 24], which provides examples of using Gen2 and Gen3 cores in Endpoint configurations. Refer to these designs to connect the External PIPE Interface ports of the UltraScale device core to third-party BFMs.

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. Please refer Transceiver Control and Status Ports in Appendix C.

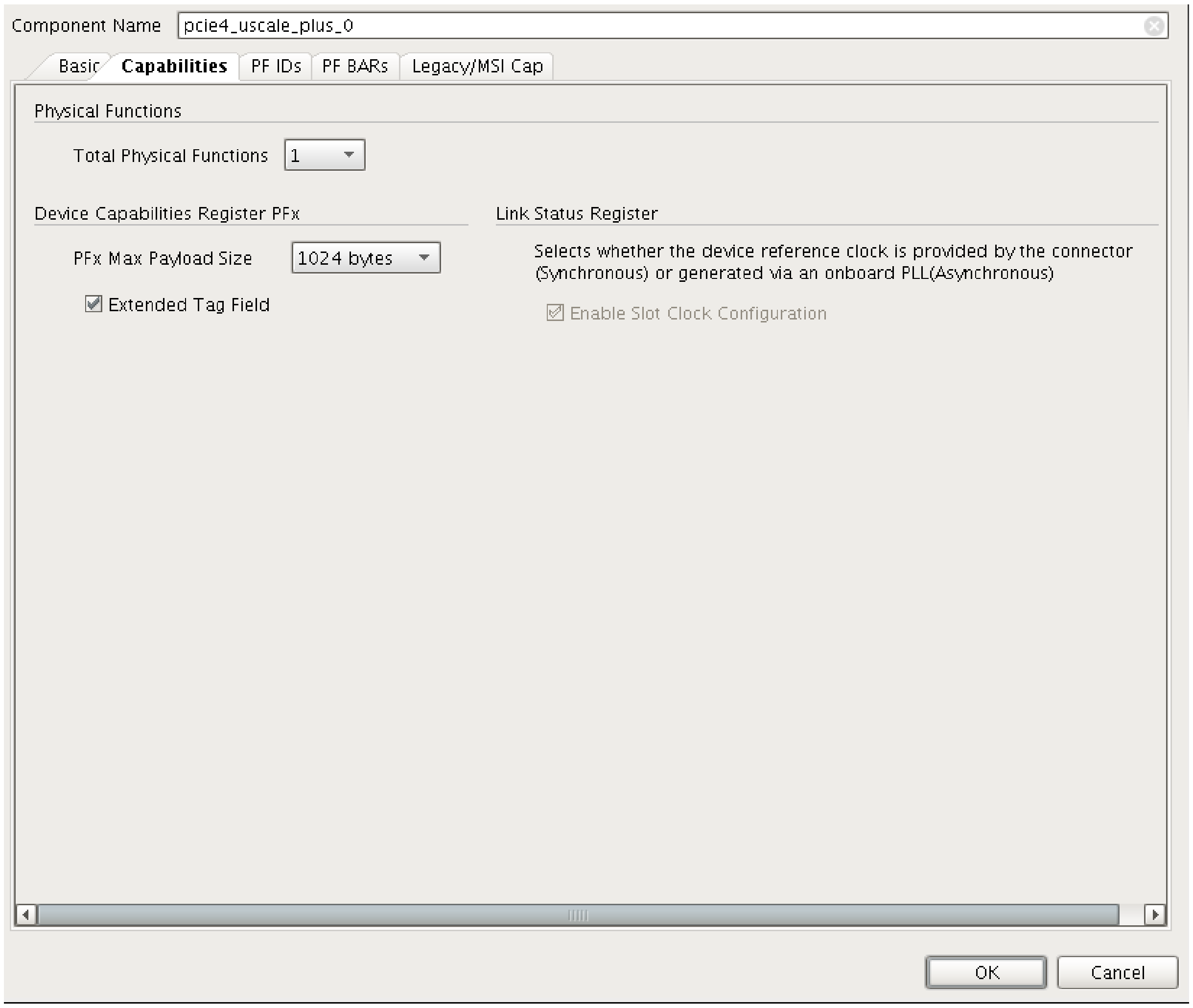
IMPORTANT: *The* ***Enable In System IBERT*** *and* ***Enable JTAG Debugger*** *options should be used only for hardware debug purposes. Simulations are not supported for the cores generated using these options.*

System reset polarity

This parameter is used to set the polarity of the sys_rst ACTIVE_HIGH or ACTIVE_LOW.

Capabilities Tab

The Capabilities settings (Figure 4-3) are explained in this section.


Component Name: pcie4_uscale_plus_0

Physical Functions
Total Physical Functions: 1

Device Capabilities Register PFx
PFx Max Payload Size: 1024 bytes
☑ Extended Tag Field

Link Status Register
Selects whether the device reference clock is provided by the connector (Synchronous) or generated via an onboard PLL(Asynchronous)
☑ Enable Slot Clock Configuration

Figure 4-3: **Capabilities Tab**

Physical Functions

Enables you to select the number of physical functions. The number of physical functions supported are 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:

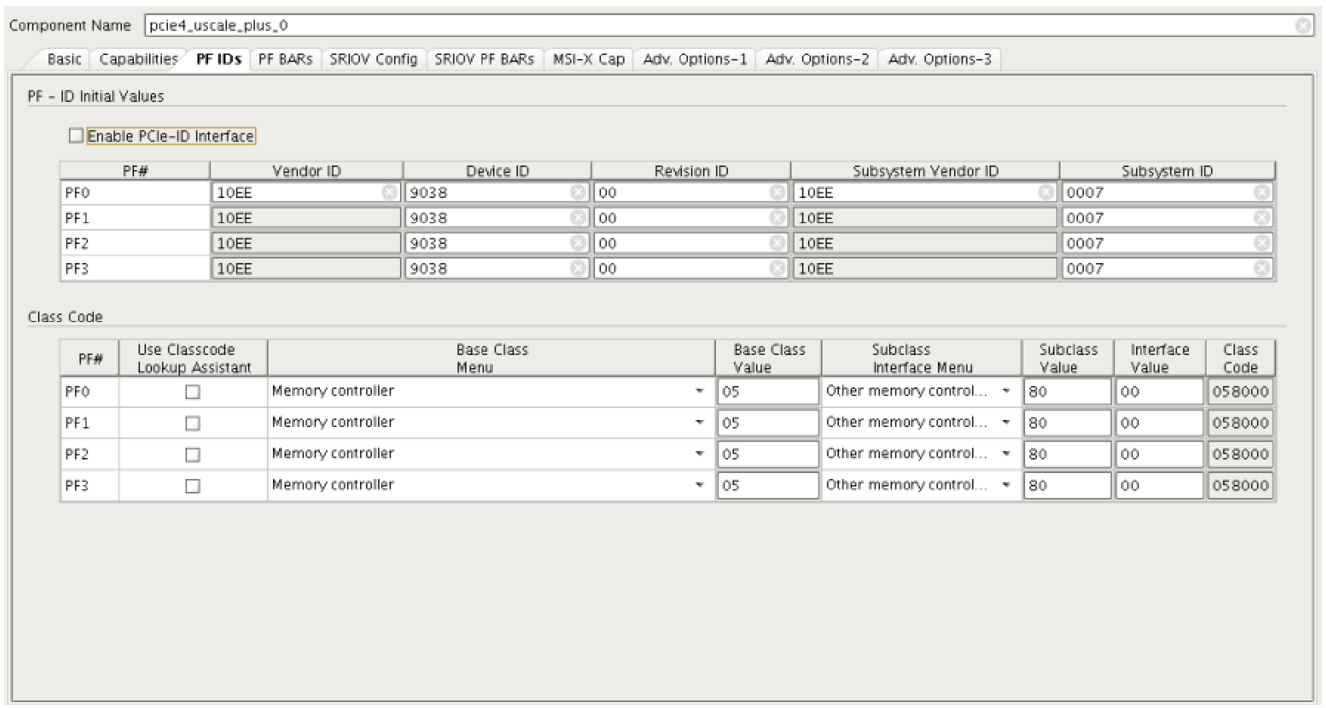
- When selected, 6-bit Tag field support (64 tags)
- When deselected, 5-bit Tag field support (32 tags)

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 Identity Settings parameters shown in Figure 4-4 are described in this section.



Component Name: pcie4_uscale_plus_0

PF - ID Initial Values

☐ Enable PCIe-ID Interface

PF#	Vendor ID	Device ID	Revision ID	Subsystem Vendor ID	Subsystem ID
PF0	10EE	9038	00	10EE	0007
PF1	10EE	9038	00	10EE	0007
PF2	10EE	9038	00	10EE	0007
PF3	10EE	9038	00	10EE	0007

Class Code

PF#	Use Classcode Lookup Assistant	Base Class Menu	Base Class Value	Subclass Interface Menu	Subclass Value	Interface Value	Class Code
PF0	☐	Memory controller	05	Other memory control...	80	00	058000
PF1	☐	Memory controller	05	Other memory control...	80	00	058000
PF2	☐	Memory controller	05	Other memory control...	80	00	058000
PF3	☐	Memory controller	05	Other memory control...	80	00	058000

Figure 4-4: **PF IDs Tab**

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.
- **Revision ID:** Indicates the revision of the device or application; an extension of the Device ID. The default value is `00h`; enter values 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.

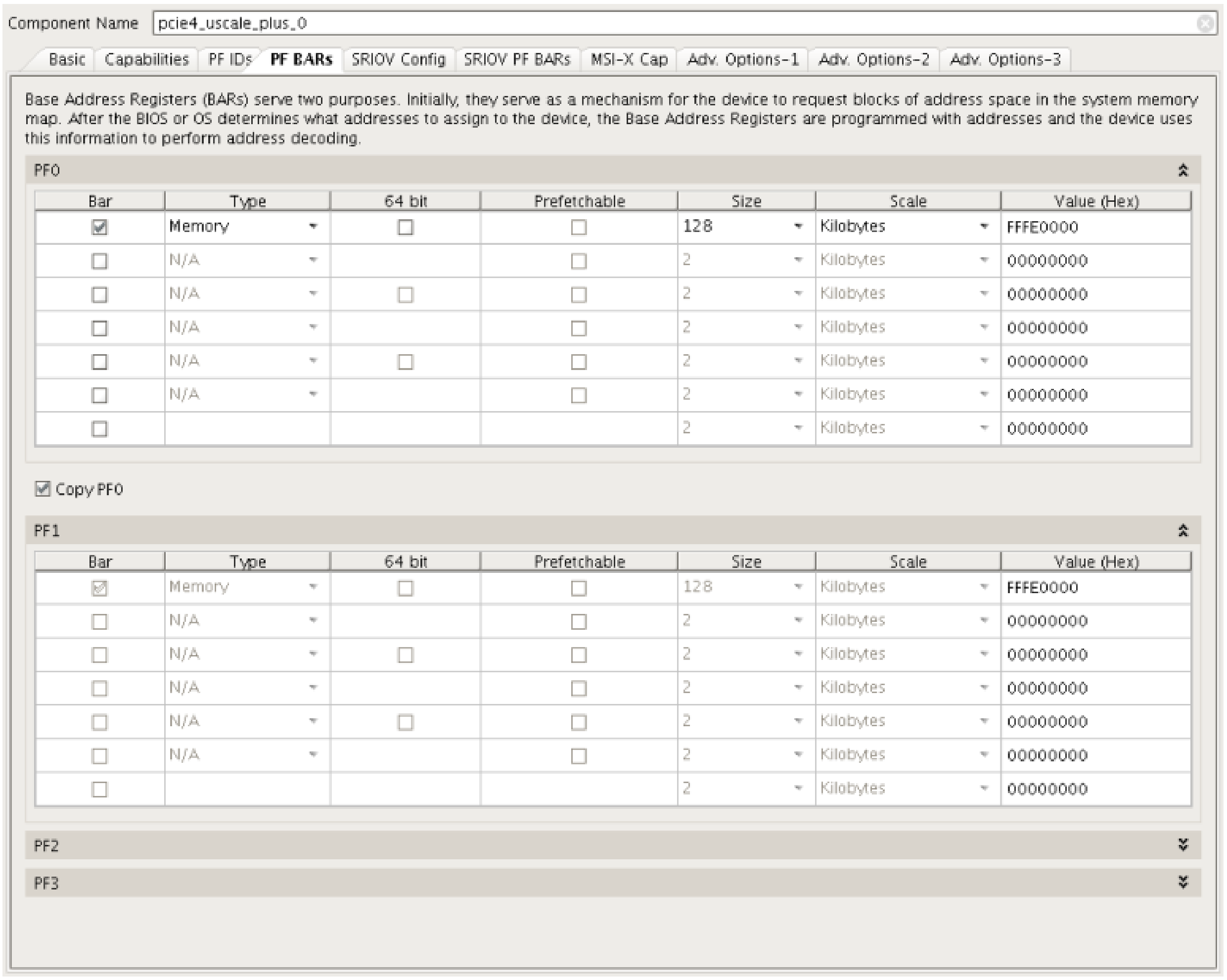
Class code encoding can be found at the PCI SIG website.

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 page (Figure 4-5) 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.



Component Name pcie4_uscale_plus_0

Basic | Capabilities | PF IDs | **PF BARs** | SRIOV Config | SRIOV PF BARs | MSI-X Cap | Adv. Options-1 | Adv. Options-2 | Adv. Options-3

Base Address Registers (BARs) serve two purposes. Initially, they serve as a mechanism for the device to request blocks of address space in the system memory map. After the BIOS or OS determines what addresses to assign to the device, the Base Address Registers are programmed with addresses and the device uses this information to perform address decoding.

PF0

Bar	Type	64 bit	Prefetchable	Size	Scale	Value (Hex)
☑	Memory	☐	☐	128	Kilobytes	FFFE0000
☐	N/A		☐	2	Kilobytes	00000000
☐	N/A	☐	☐	2	Kilobytes	00000000
☐	N/A		☐	2	Kilobytes	00000000
☐	N/A	☐	☐	2	Kilobytes	00000000
☐	N/A		☐	2	Kilobytes	00000000
☐				2	Kilobytes	00000000

☑ Copy PF0

PF1

Bar	Type	64 bit	Prefetchable	Size	Scale	Value (Hex)
☑	Memory	☐	☐	128	Kilobytes	FFFE0000
☐	N/A		☐	2	Kilobytes	00000000
☐	N/A	☐	☐	2	Kilobytes	00000000
☐	N/A		☐	2	Kilobytes	00000000
☐	N/A	☐	☐	2	Kilobytes	00000000
☐	N/A		☐	2	Kilobytes	00000000
☐				2	Kilobytes	00000000

PF2

PF3

Figure 4-5: **PF BARs Tab Showing PF0 and PF1 Only**

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 to 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 the Legacy PCI Express Endpoint core.
    - *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. Table 4-4 lists the available BAR size ranges.

Table 4-4: **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 – 256 GB
Legacy PCI Express Endpoint	32-bit Memory	128 B – 2 GB
	64-bit Memory	128 B – 256 GB
	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.

For more information about managing the Base Address Register settings, see Managing Base Address Register Settings.

Expansion ROM Base Address Register

If selected, the Expansion ROM is activated and can be a value from 2 KB to 4 GB. According to the *PCI 3.0 Local Bus Specification* [Ref 2], the maximum size for the Expansion ROM BAR should be no larger than 16 MB. Selecting an address space larger than 16 MB can result in a non-compliant core.

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.

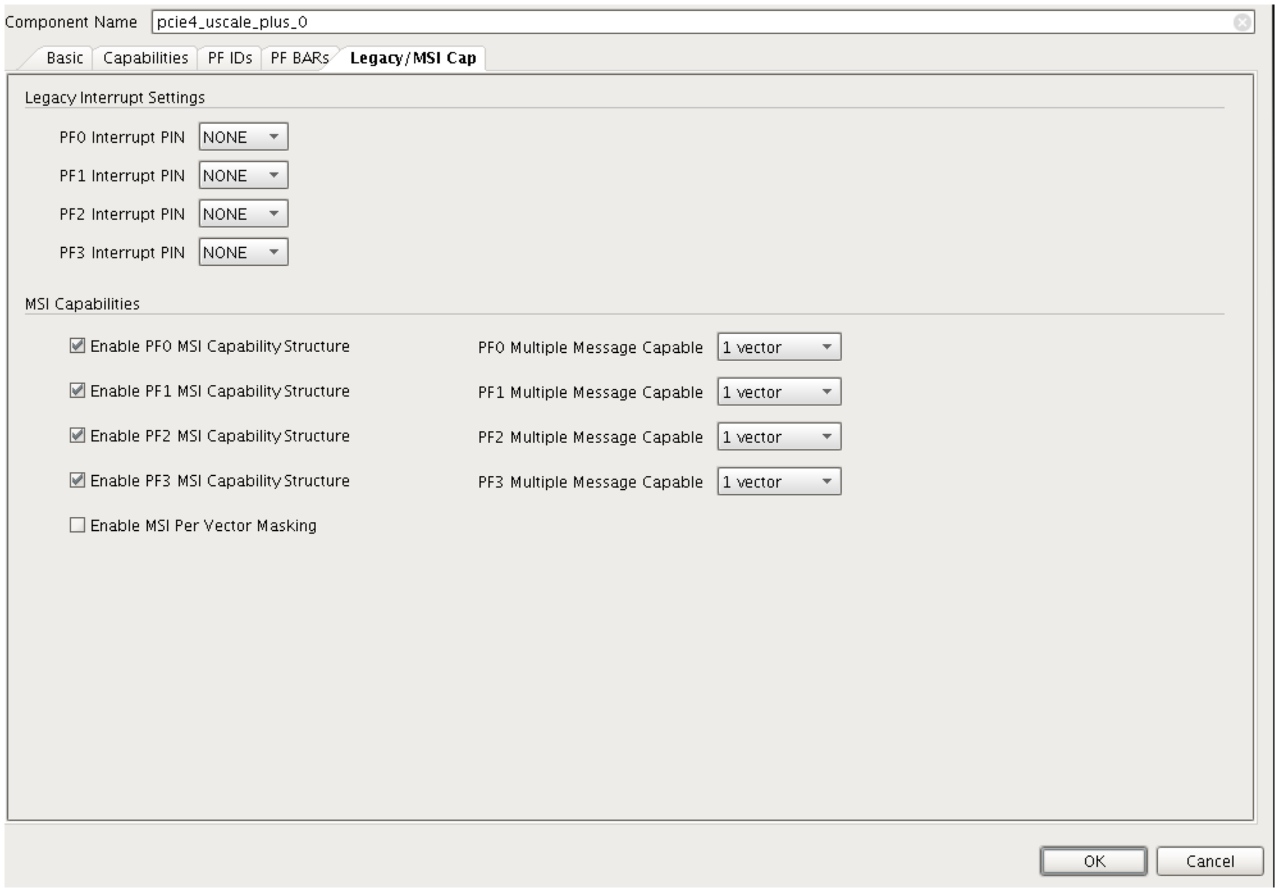
When configuring the core as an Endpoint for PCIe (non-Legacy), 64-bit addressing must be supported for all BARs (except BAR5) that have the prefetchable bit set. 32-bit addressing is permitted for all BARs that do not have the prefetchable bit set. The prefetchable bit-related requirement does not apply 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 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).



Component Name pcie4_uscale_plus_0

Legacy Interrupt Settings

PF0 Interrupt PIN NONE
PF1 Interrupt PIN NONE
PF2 Interrupt PIN NONE
PF3 Interrupt PIN NONE

MSI Capabilities

- [✓] Enable PF0 MSI Capability Structure — PF0 Multiple Message Capable: 1 vector
- [✓] Enable PF1 MSI Capability Structure — PF1 Multiple Message Capable: 1 vector
- [✓] Enable PF2 MSI Capability Structure — PF2 Multiple Message Capable: 1 vector
- [✓] Enable PF3 MSI Capability Structure — PF3 Multiple Message Capable: 1 vector
- [ ] Enable MSI Per Vector Masking

Figure 4-6: **Legacy/MSI Cap Tab**

Legacy Interrupt Settings

- **PF0/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

- **PF0/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* [Ref 2] requires that MSI, MSI-X, or both be enabled. No MSI capabilities are supported when Inbuilt or MSI-X Internal is enabled 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.

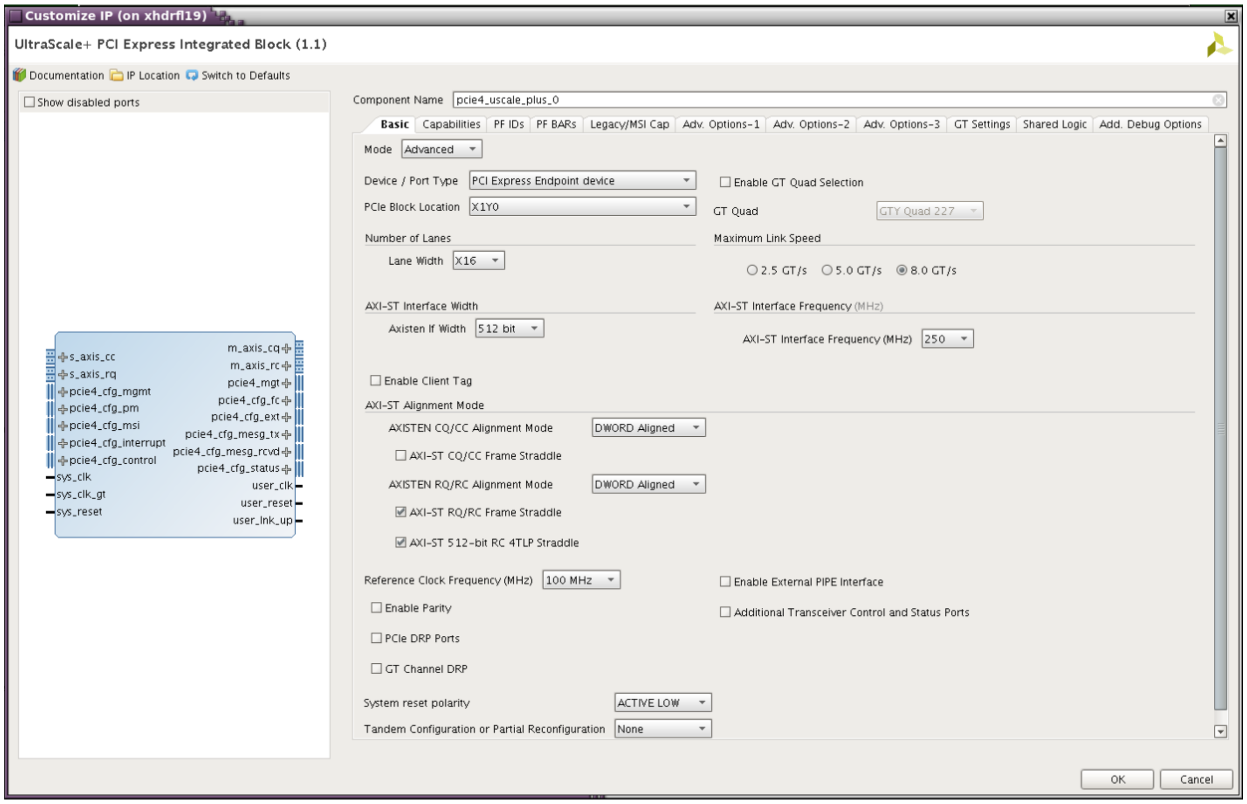
  Note: Note: Enabling this option for individual physical functions is not supported.

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 for Advanced mode (Figure 4-7) includes some additional settings. The following parameters are on the Basic page when the **Advanced** mode is selected.



UltraScale+ PCI Express Integrated Block (1.1)

☐ Show disabled ports

Component Name: pcie4_uscale_plus_0

Mode: Advanced

Device / Port Type: PCI Express Endpoint device

PCIe Block Location: X1Y0

☐ Enable GT Quad Selection

GT Quad: GTY Quad 227

Number of Lanes

Lane Width: X16

Maximum Link Speed

○ 2.5 GT/s ○ 5.0 GT/s ● 8.0 GT/s

AXI-ST Interface Width

Axisten If Width: 512 bit

AXI-ST Interface Frequency (MHz)

AXI-ST Interface Frequency (MHz): 250

☐ Enable Client Tag

AXI-ST Alignment Mode

AXISTEN CQ/CC Alignment Mode: DWORD Aligned

☐ AXI-ST CQ/CC Frame Straddle

AXISTEN RQ/RC Alignment Mode: DWORD Aligned

☑ AXI-ST RQ/RC Frame Straddle

☑ AXI-ST 512-bit RC 4TLP Straddle

Reference Clock Frequency (MHz): 100 MHz

☐ Enable External PIPE Interface

☐ Enable Parity

☐ Additional Transceiver Control and Status Ports

☐ PCIe DRP Ports

☐ GT Channel DRP

System reset polarity: ACTIVE LOW

Tandem Configuration or Partial Reconfiguration: None

Figure 4-7: **Basic Tab, Advanced Mode**

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

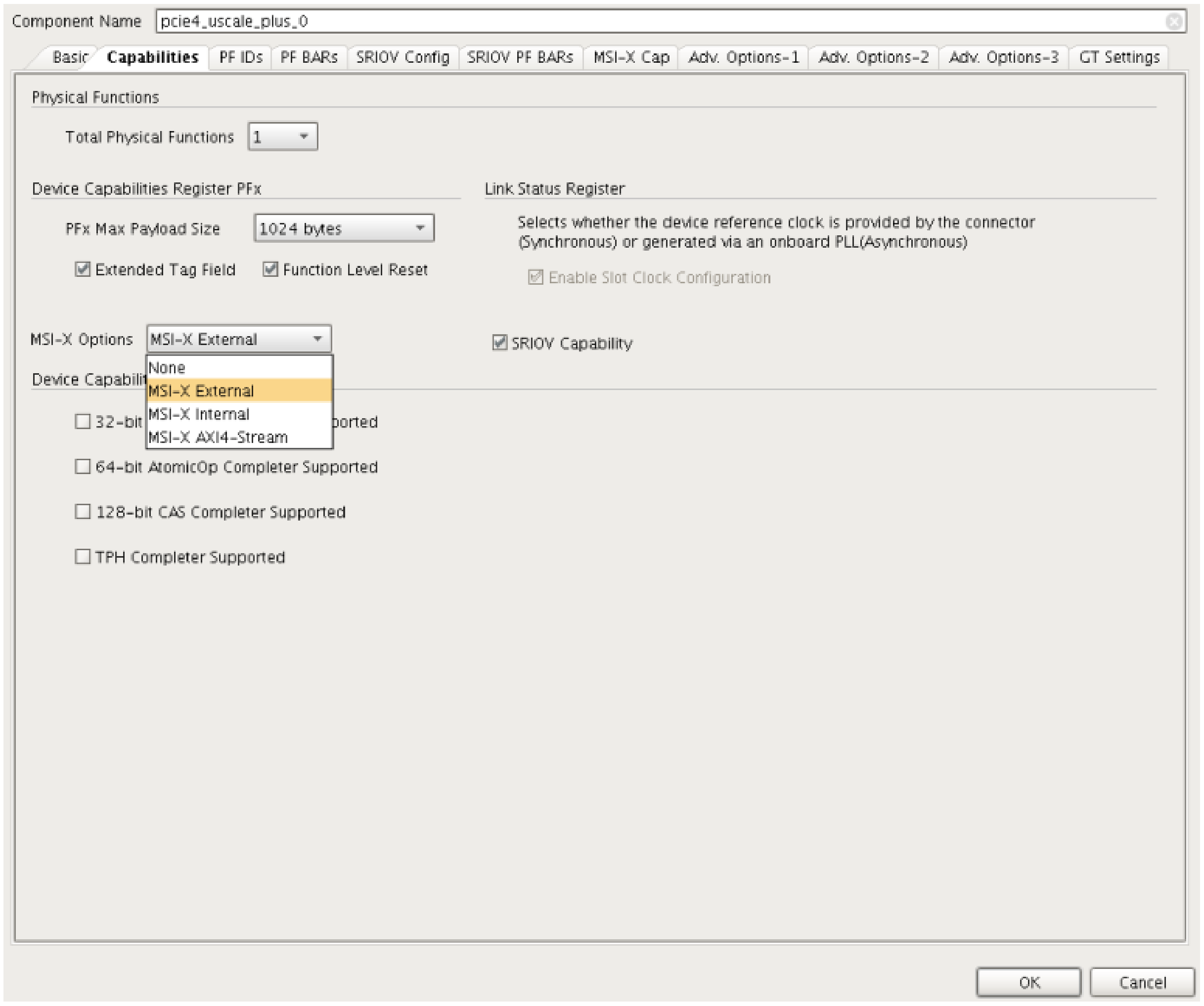
When checked, enables the PCIe DRP interface.

GT Channel DRP

When checked, enables the GT channel DRP interface.

Capabilities Tab

The Capabilities settings for Advanced mode (Figure 4-8) contains two additional parameters to those for Basic mode and are described below.


Component Name: pcie4_uscale_plus_0

Physical Functions
Total Physical Functions: 1

Device Capabilities Register PFx
PFx Max Payload Size: 1024 bytes
[✓] Extended Tag Field
[✓] Function Level Reset

Link Status Register
Selects whether the device reference clock is provided by the connector (Synchronous) or generated via an onboard PLL(Asynchronous)
[✓] Enable Slot Clock Configuration

MSI-X Options: MSI-X External
[✓] SRIOV Capability

Device Capabilities
[ ] 32-bit AtomicOp Completer Supported
[ ] 64-bit AtomicOp Completer Supported
[ ] 128-bit CAS Completer Supported
[ ] TPH Completer Supported

Figure 4-8: **Capabilities Tab, Advanced Mode**

SRIOV Capabilities

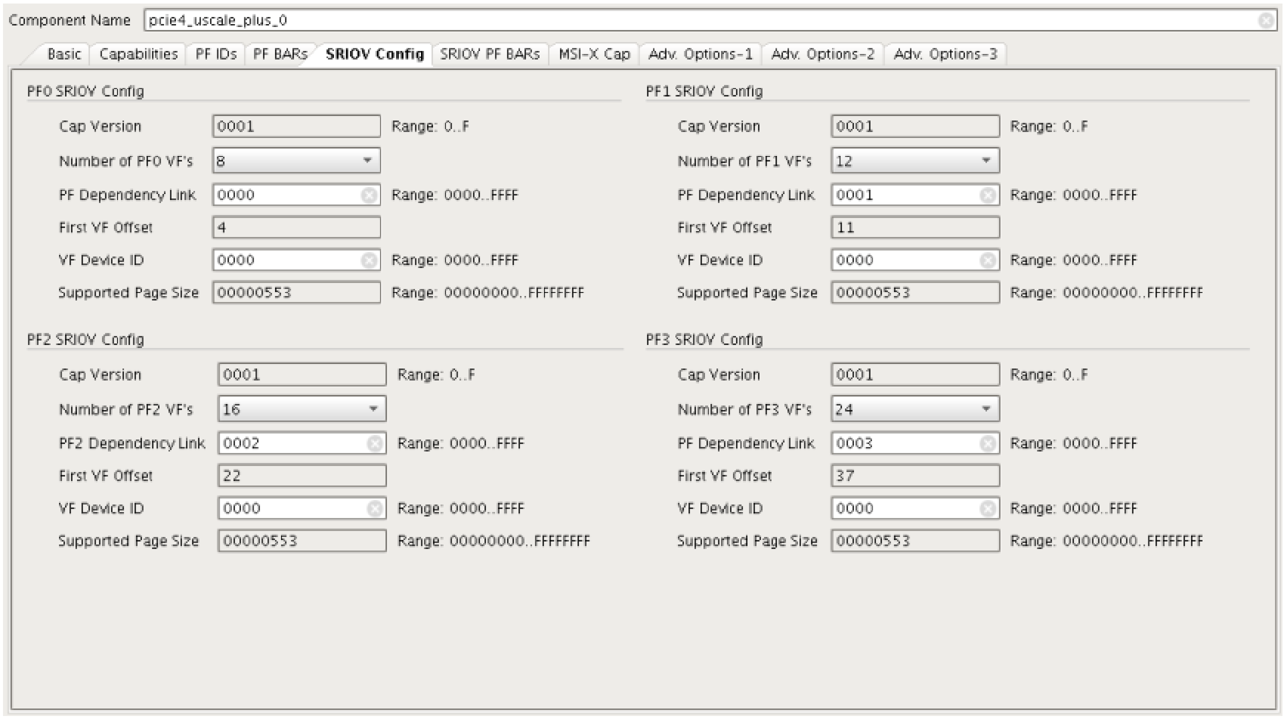
Enables Single Root Port I/O Virtualization (SR-IOV) capabilities. The integrated block implements extended Single Root Port I/O Virtualization PCIe. When this is enabled, SR-IOV is implemented on all the selected physical functions. When SR-IOV capabilities are enabled MSI support is disabled and you can use MSI-X support as shown in Figure 4-8.

Device Capabilities Registers 2

Specifies options for AtomicOps and TPH Completer support. See the Device Capability register 2 description in Chapter 7 of the *PCI Express Base Specification* [Ref 2] for more information. These settings apply to all the selected physical functions.

SRIOV Config Tab

The SRIOV Configuration Advanced parameters (Figure 4-9) are described in this section.



Component Name: pcie4_uscale_plus_0

PF0 SRIOV Config

Cap Version	0001	Range: 0..F
Number of PF0 VF's	8	
PF Dependency Link	0000	Range: 0000..FFFF
First VF Offset	4	
VF Device ID	0000	Range: 0000..FFFF
Supported Page Size	00000553	Range: 00000000..FFFFFFFF

PF1 SRIOV Config

Cap Version	0001	Range: 0..F
Number of PF1 VF's	12	
PF Dependency Link	0001	Range: 0000..FFFF
First VF Offset	11	
VF Device ID	0000	Range: 0000..FFFF
Supported Page Size	00000553	Range: 00000000..FFFFFFFF

PF2 SRIOV Config

Cap Version	0001	Range: 0..F
Number of PF2 VF's	16	
PF2 Dependency Link	0002	Range: 0000..FFFF
First VF Offset	22	
VF Device ID	0000	Range: 0000..FFFF
Supported Page Size	00000553	Range: 00000000..FFFFFFFF

PF3 SRIOV Config

Cap Version	0001	Range: 0..F
Number of PF3 VF's	24	
PF Dependency Link	0003	Range: 0000..FFFF
First VF Offset	37	
VF Device ID	0000	Range: 0000..FFFF
Supported Page Size	00000553	Range: 00000000..FFFFFFFF

Figure 4-9: **SRIOV Configuration Tab**

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). PF0 always resides at Offset 0, and PF1 always resides at Offset 1. Six virtual functions are available in the Gen3 Integrated Block for PCIe core and reside at the function number range 64–69.

Virtual functions are mapped sequentially with VFs for PF0 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 = (64 - 0) = 64
> PF1_FIRST_VF_OFFSET = (66 - 1) = 65
> ```
>
> PF0 is always 64 assuming that PF0 has one or more virtual functions. 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 = 63 + NUM_PF0_VFS
> ```

VF Device ID

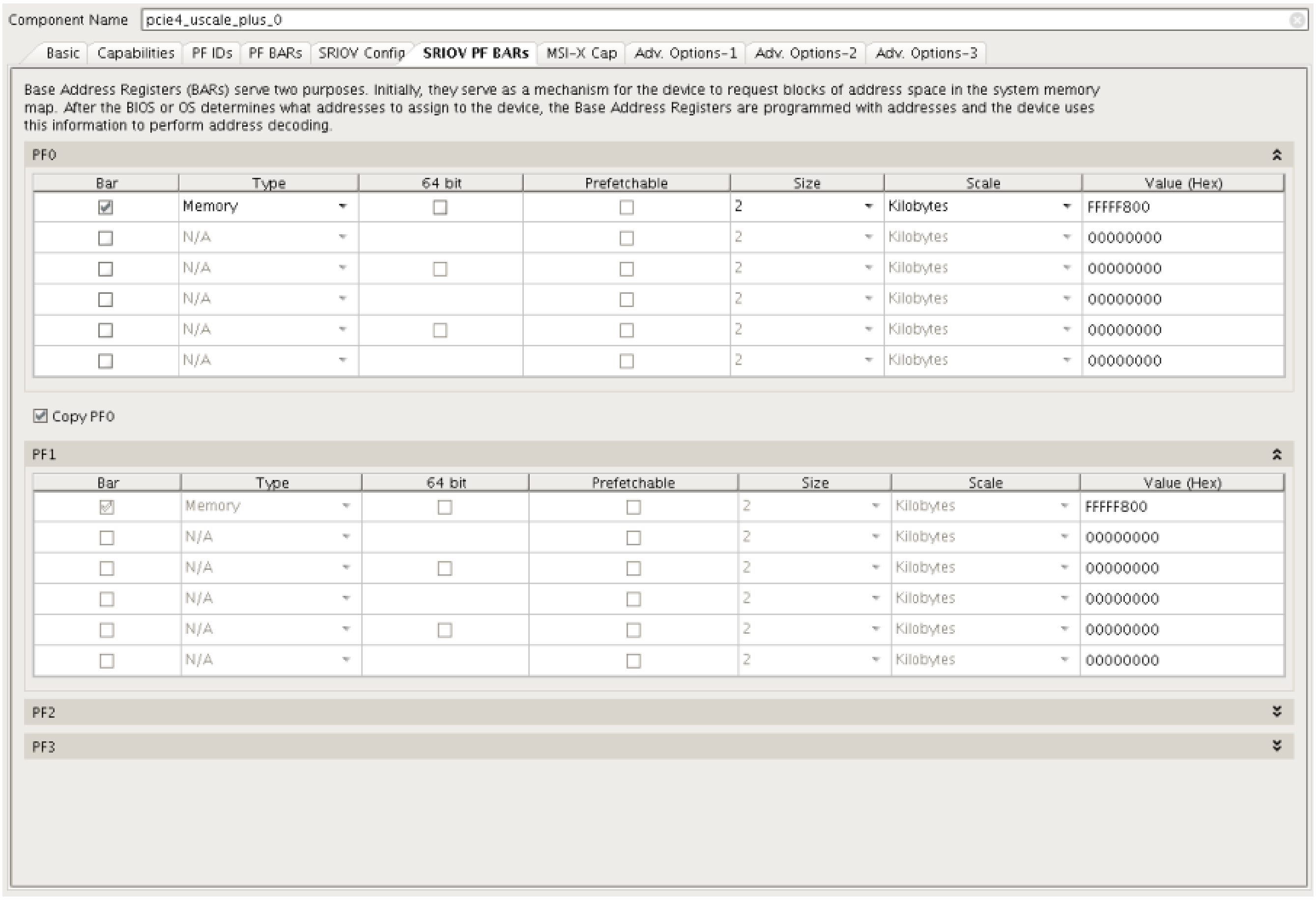
Indicates the 16-bit Device ID for all virtual functions associated with the physical function.

SRIOV Supported Page Size

Indicates the page size supported by the physical function. This physical function supports a page size of 2n+12, if bit n of the 32-bit register is set.

SRIVO 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 SRIOV BAR Aperture Size and SRIOV Control attributes.



Component Name pcie4_uscale_plus_0

Base Address Registers (BARs) serve two purposes. Initially, they serve as a mechanism for the device to request blocks of address space in the system memory map. After the BIOS or OS determines what addresses to assign to the device, the Base Address Registers are programmed with addresses and the device uses this information to perform address decoding.

PF0

Bar	Type	64 bit	Prefetchable	Size	Scale	Value (Hex)
☑	Memory	☐	☐	2	Kilobytes	FFFFF800
☐	N/A		☐	2	Kilobytes	00000000
☐	N/A	☐	☐	2	Kilobytes	00000000
☐	N/A		☐	2	Kilobytes	00000000
☐	N/A	☐	☐	2	Kilobytes	00000000
☐	N/A		☐	2	Kilobytes	00000000

☑ Copy PF0

PF1

Bar	Type	64 bit	Prefetchable	Size	Scale	Value (Hex)
☑	Memory	☐	☐	2	Kilobytes	FFFFF800
☐	N/A		☐	2	Kilobytes	00000000
☐	N/A	☐	☐	2	Kilobytes	00000000
☐	N/A		☐	2	Kilobytes	00000000
☐	N/A	☐	☐	2	Kilobytes	00000000
☐	N/A		☐	2	Kilobytes	00000000

PF2

PF3

Figure 4-10: **SRIOV BARs Tab, Advanced Mode**

Table 4-5: **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. SRIOV BARs can be one of two sizes:

- **32-bit BARs**: The address space can be as small as 16 bytes or as large as 2 Gbytes. 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 SRIOV BAR registers have these options:

- **Checkbox**: Click the checkbox to enable the BAR; deselect the checkbox to disable the BAR.
- **Type**: SRIOV 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 the Legacy PCI Express Endpoint core.
  - *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. Table 4-6 lists the available BAR size ranges.

Table 4-6: **SRIOV BAR Size Ranges for Device Configuration**

PCIe Device / Port Type	BAR Type	BAR Size Range
PCI Express Endpoint	32-bit Memory	128 B – 2 GB
	64-bit Memory	128 B – 8 Exabytes
Legacy PCI Express Endpoint	32-bit Memory	16 B – 2 GB
	64-bit Memory	16 B – 8 Exabytes
	I/O	16 B – 2 GB
Root Port Mode	32-bit Memory	16 B – 2 GB
	64-bit Memory	4 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.

For more information about managing the SRIOV Base Address Register settings, see Managing Base Address Register Settings.

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 SRIOV BARs (except BAR5) that have the prefetchable bit set. 32-bit addressing is permitted for all SRIOV 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

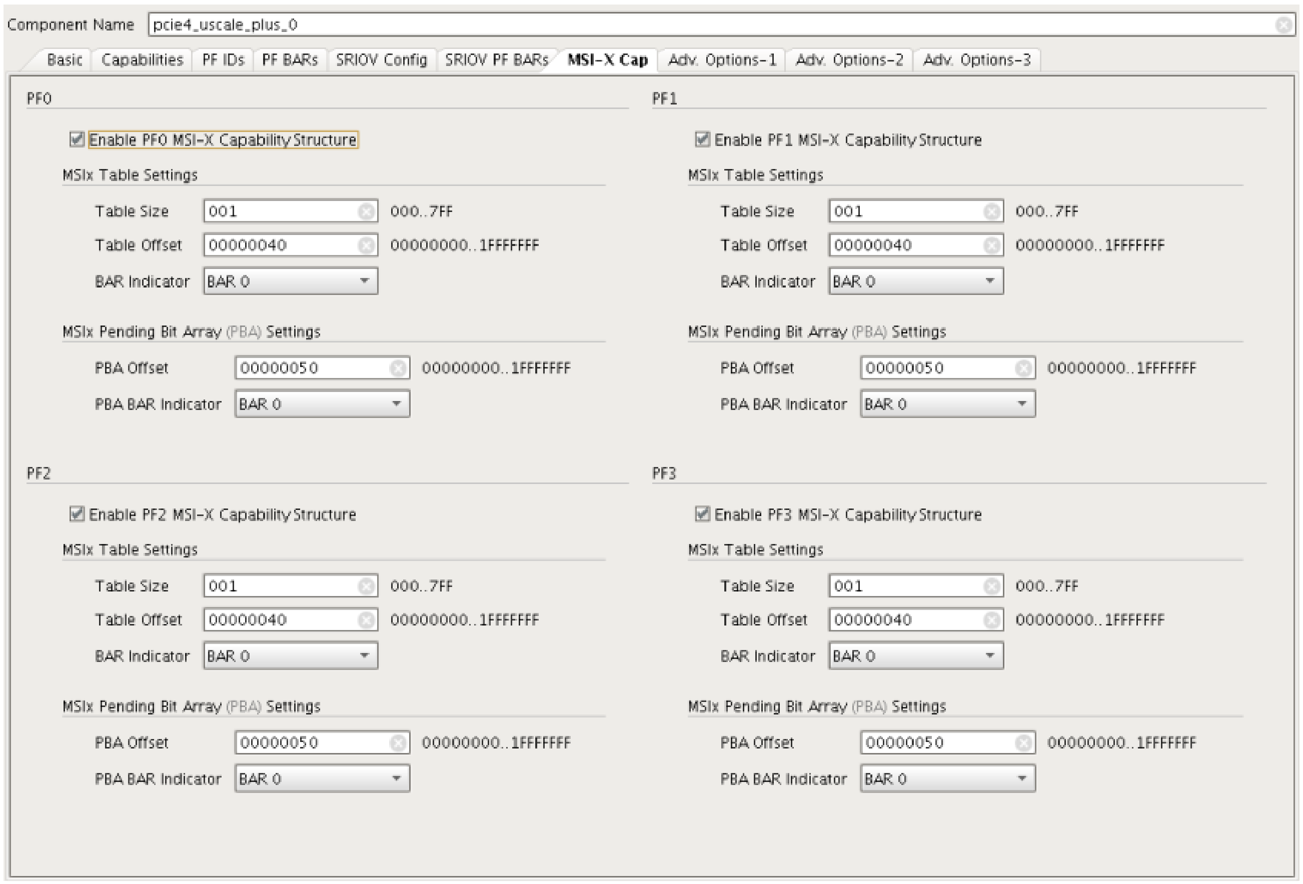
For best results, disable unused base address registers to conserve system resources. Disable base address register by deselecting unused BARs in the Customize IP dialog box.

MSI-X Capabilities Tab

The MSI-X Capabilities parameters (Figure 4-11) are available in Advanced mode only. To enable MSI-X capabilities, select **Advanced** mode and then select the required options on the Capabilities page. As shown in Figure 4-8 there are four options to choose from.

1. **MSI-X External**: In this mode you need to implement MSI-X External interface driving logic, MSI-X Table and PBA buffers outside the PCIe core. You can configure the MSI-X BARs.
2. **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 PCIe core. You can configure the MSI-X BARs.
3. **MSI-X AXI4-Stream**: In this mode user is expected to drive MSI-X interrupts on the AXI4-Streaming interface. You can configure the MSI-X BARs.
4. **None**: No MSI-X is supported.

The same MSI-X options are applicable even when SR-IOV capability is selected.



Component Name: pcie4_uscale_plus_0

Basic | Capabilities | PF IDs | PF BARs | SRIOV Config | SRIOV PF BARs | **MSI-X Cap** | Adv. Options-1 | Adv. Options-2 | Adv. Options-3

PF0

[✓] Enable PF0 MSI-X Capability Structure

MSIx Table Settings

Table Size	001	000..7FF
Table Offset	00000040	00000000..1FFFFFFF
BAR Indicator	BAR 0	

MSIx Pending Bit Array (PBA) Settings

PBA Offset	00000050	00000000..1FFFFFFF
PBA BAR Indicator	BAR 0	

PF1

[✓] Enable PF1 MSI-X Capability Structure

MSIx Table Settings

Table Size	001	000..7FF
Table Offset	00000040	00000000..1FFFFFFF
BAR Indicator	BAR 0	

MSIx Pending Bit Array (PBA) Settings

PBA Offset	00000050	00000000..1FFFFFFF
PBA BAR Indicator	BAR 0	

PF2

[✓] Enable PF2 MSI-X Capability Structure

MSIx Table Settings

Table Size	001	000..7FF
Table Offset	00000040	00000000..1FFFFFFF
BAR Indicator	BAR 0	

MSIx Pending Bit Array (PBA) Settings

PBA Offset	00000050	00000000..1FFFFFFF
PBA BAR Indicator	BAR 0	

PF3

[✓] Enable PF3 MSI-X Capability Structure

MSIx Table Settings

Table Size	001	000..7FF
Table Offset	00000040	00000000..1FFFFFFF
BAR Indicator	BAR 0	

MSIx Pending Bit Array (PBA) Settings

PBA Offset	00000050	00000000..1FFFFFFF
PBA BAR Indicator	BAR 0	

Figure 4-11: **MSI-X Cap Tab, Advanced Mode**

Enable MSIx Capability Structure

Indicates that the MSI-X Capability 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.

MSIx Table Settings:

Defines the MSI-X Table structure.

- **Table Size**: Specifies the MSI-X Table size.
- **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.

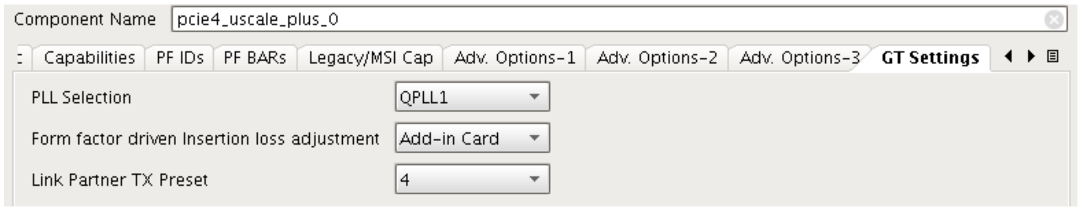
MSIx Pending Bit Array (PBA) Settings

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.

GT Settings Tab

Settings in this page allow you to customize specific transceiver settings that are normally not accessible.


Component Name: pcie4_uscale_plus_0

PLL Selection: QPLL1

Form factor driven Insertion loss adjustment: Add-in Card

Link Partner TX Preset: 4

Figure 4-12: **GT Settings Tab**

PLL Selection

(Only available when Gen2 link speed is selected), allows for either the **QPLL** 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 QPLL, and Gen1 speeds always use the CPLL.

IMPORTANT: *The remainder of the settings should not be modified unless instructed to do so by Xilinx.*

Table 4-7 shows the options and default for each line speed.

Table 4-7: **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.

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:

1. **Chip-to-Chip**: The value is 5 dB.

2. **Add-in Card**: The value is 15 dB and is the default option.
3. **Backplane**: The value is 20 dB.

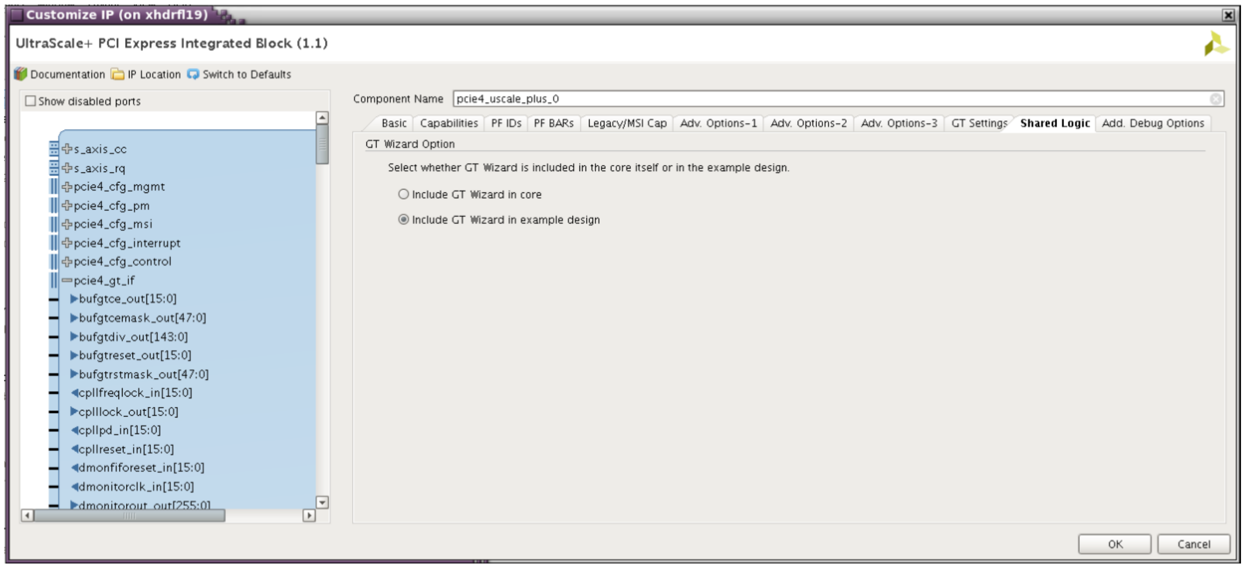
These insertion loss values are applied to the GT Wizard subcore.

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.

Shared Logic

Figure 4-13 shows the Shared Logic tab.



Component Name: pcie4_uscale_plus_0

GT Wizard Option

Select whether GT Wizard is included in the core itself or in the example design.

Include GT Wizard in core [✓ Include GT Wizard in example design]

Figure 4-13: **Shared Logic**

Currently this core is sharing GT Wizard logic only. You can select include GT Wizard in 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 GTH Transceivers User Guide (UG576)* [Ref 11] or *UltraScale Architecture GTY Transceivers User Guide (UG578)* [Ref 12].

Add. Debug Options

You can select additional debug portions for debugging purposes. The parameters are described below. For port level descriptions, see Hardware Debug in Appendix C.

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, see *In-System IBERT LogiCORE IP Product Guide* (PG246) [Ref 22].

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 the **Gen3, Gen2 or Gen1 link speed** at any link width.
   - Select the **Enable In System IBERT** option 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 Manger (HM) and configure the FPGA using the generated .bit and .ltx file.
7. Reboot the machine to rescan and to 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 2 instead 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.*

Enable Descrambler for Gen3 Mode

This debug option integrates encrypted version of the descrambler module inside the PCIe core, which will be used to descrambler the PIPE data to/from PCIe integrated block in Gen3 link speed mode. This provides hardware-only support to debug on the board.

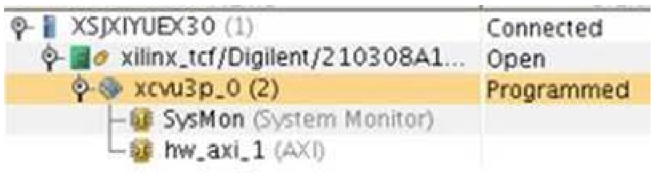
Enable JTAG Debugger

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 and connect to the board.
2. You should see `hw_axi_1`.



```
XSJXIYUEX30 (1)                    Connected
  xilinx_tcf/Digilent/210308A1...  Open
    xcvu3p_0 (2)                   Programmed
      SysMon (System Monitor)
      hw_axi_1 (AXI)
```

3. Type `source test_rd.tcl` in the Vivado Tcl Console.

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.

Output Generation

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 15].

Constraining the Core

This section contains information about constraining the core in the Vivado® Design Suite.

Required Constraints

The UltraScale+ Devices Integrated Block for PCIe 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_GTE3 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) [Ref 18].

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 UltraScale+ Devices Integrated Block for PCIe cores are designed 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 3, Designing with the Core, for detailed information about clock requirements.

Clock Management

See Chapter 3, Designing with the Core, for detailed information about clock requirements.

Clock Placement

See Chapter 3, 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, transceivers, and the PCIe 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, you must 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 (i.e., 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. Then, 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) [Ref 19].

For information regarding simulating the example design, see Simulating the Example Design in Chapter 5.

PIPE Mode Simulation

The UltraScale+ Devices Integrated Block for PCIe core 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 page 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, page 271.

Table 4-8 and Table 4-9 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 4-8: **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

Table 4-8: **Common In/Out Commands and Endpoint PIPE Signals Mappings *(Cont'd)***

In Commands	Endpoint PIPE Signals Mapping	Out Commands	Endpoint PIPE Signals Mapping
		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. This ports functionality has been deprecated and can be left unconnected.

Table 4-9: **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_syncheader_gt	pipe_tx_0_sigs[38:37]	pipe_tx0_syncheader_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. This ports functionality has been deprecated and can be left unconnected.

Synthesis and Implementation

For details about synthesis and implementation, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 15].

For information regarding synthesizing and implementing the example design, see Synthesizing and Implementing the Example Design in Chapter 5.

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 PCIe 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, SR-IOV, MSI-X, MSI.

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. Figure 5-1 illustrates the simulation design provided with the core. For more information about the Root Port Model, see Root Port Model Test Bench for Endpoint, page 312.

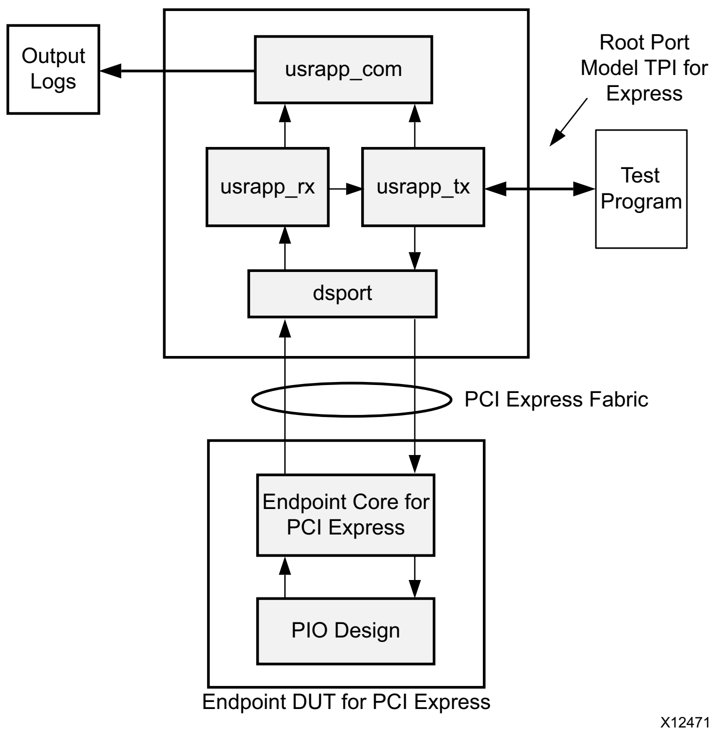


Figure 5-1: **Simulation Example Design Block Diagram**

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 Figure 5-2. 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, page 297.

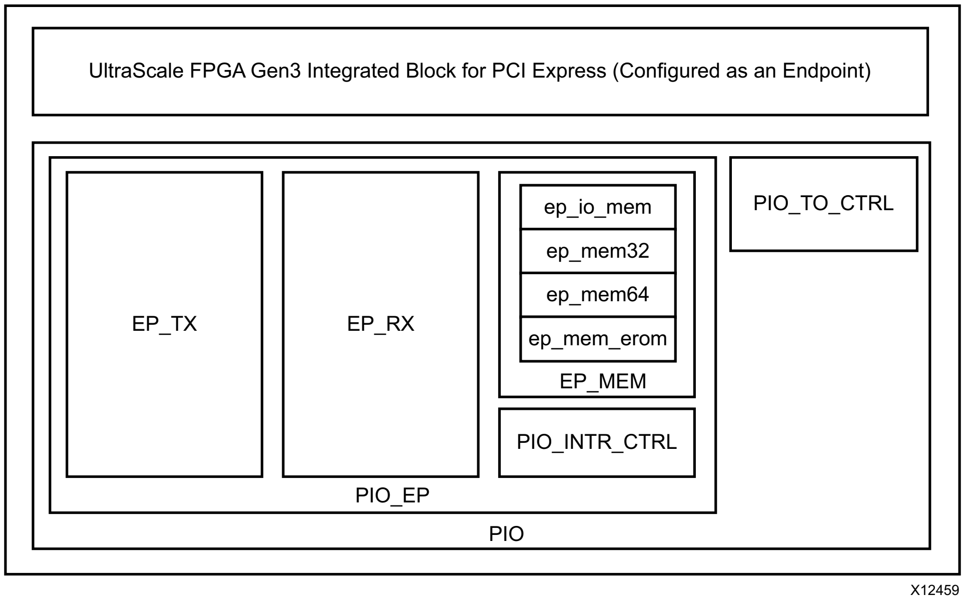


Figure 5-2: **Implementation Example Design Block Diagram**

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
- Mentor Graphics QuestaSim
- Cadence Incisive Enterprise Simulator (IES)
- Synopsys Verilog Compiler Simulator (VCS)

For the supported versions of these tools, see the *Xilinx Design Tools: Release Notes Guide*[(2)].

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:

- Four transaction-specific 2 KB target regions using the internal FPGA block RAMs, providing a total target space of 8,192 bytes
- 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
- 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, and 256-bit AXI4-Stream interfaces

Figure 5-3 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.

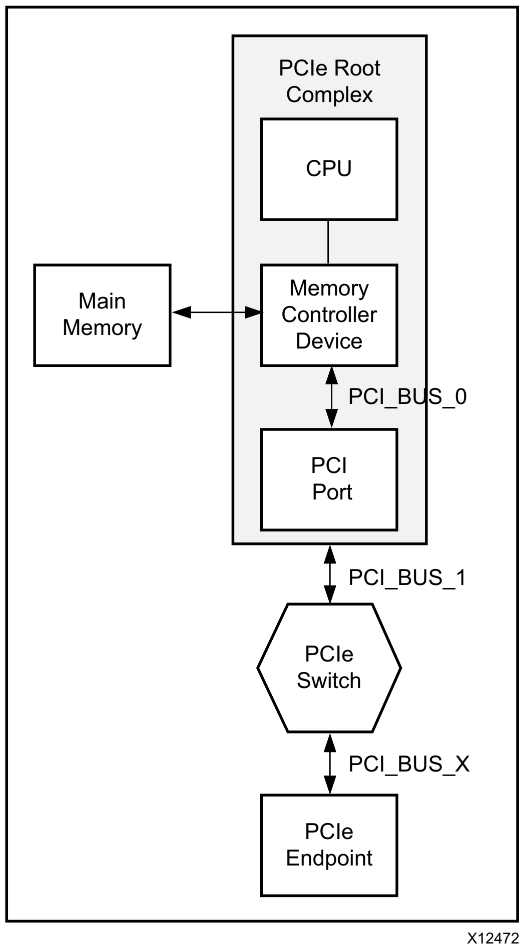


Figure 5-3: **System Overview**

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

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.

The PIO design can initiate:

- 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 by updating the payload into the target address in the FPGA block RAM space.

Base Address Register Support

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 Table 5-1.

Table 5-1: **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`

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 and I/O Read and Write TLPs. Memory Read or Memory Write TLPs of lengths larger than one Dword are not processed correctly by the PIO design; 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.

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.

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.

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.

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. 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

Table 5-2 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 5-2: **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

Three configurations of the PIO design are provided: PIO_64, PIO_128, and PIO_256 with 64-, 128-, and 256-bit AXI4-Stream interfaces, respectively. The PIO configuration that is generated depends on the selected Endpoint type (that is, UltraScale™ device integrated block, PIPE, PCI Express, and Block Plus) as well as the number of PCI Express lanes and the interface width selected. Table 5-3 identifies the PIO configuration generated based on your selection.

Table 5-3: **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.

Figure 5-4 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.

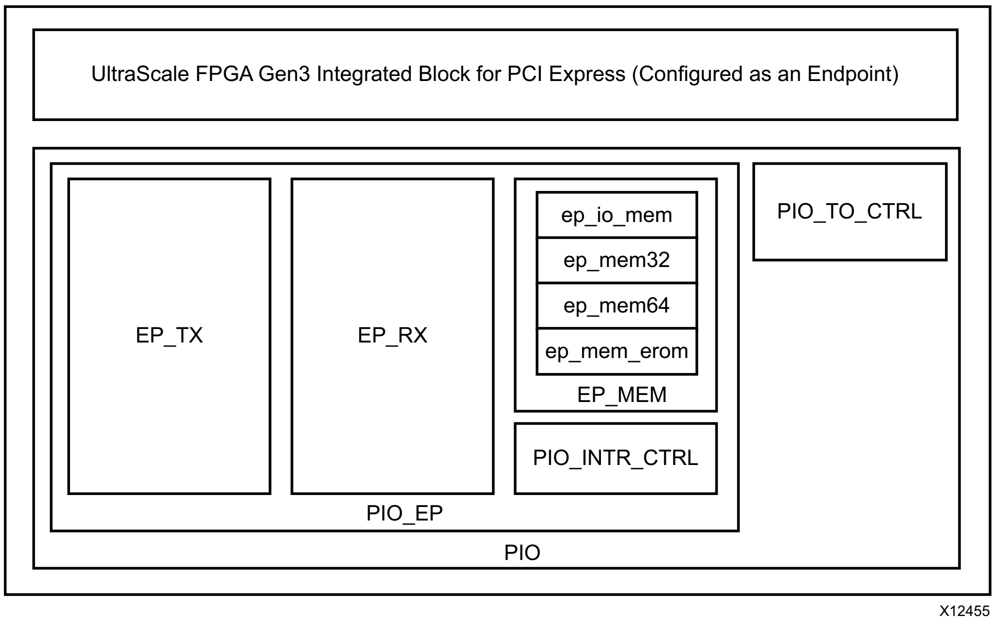


Figure 5-4: **PIO Design Components**

PIO Operation

PIO Read Transaction

Figure 5-5 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.

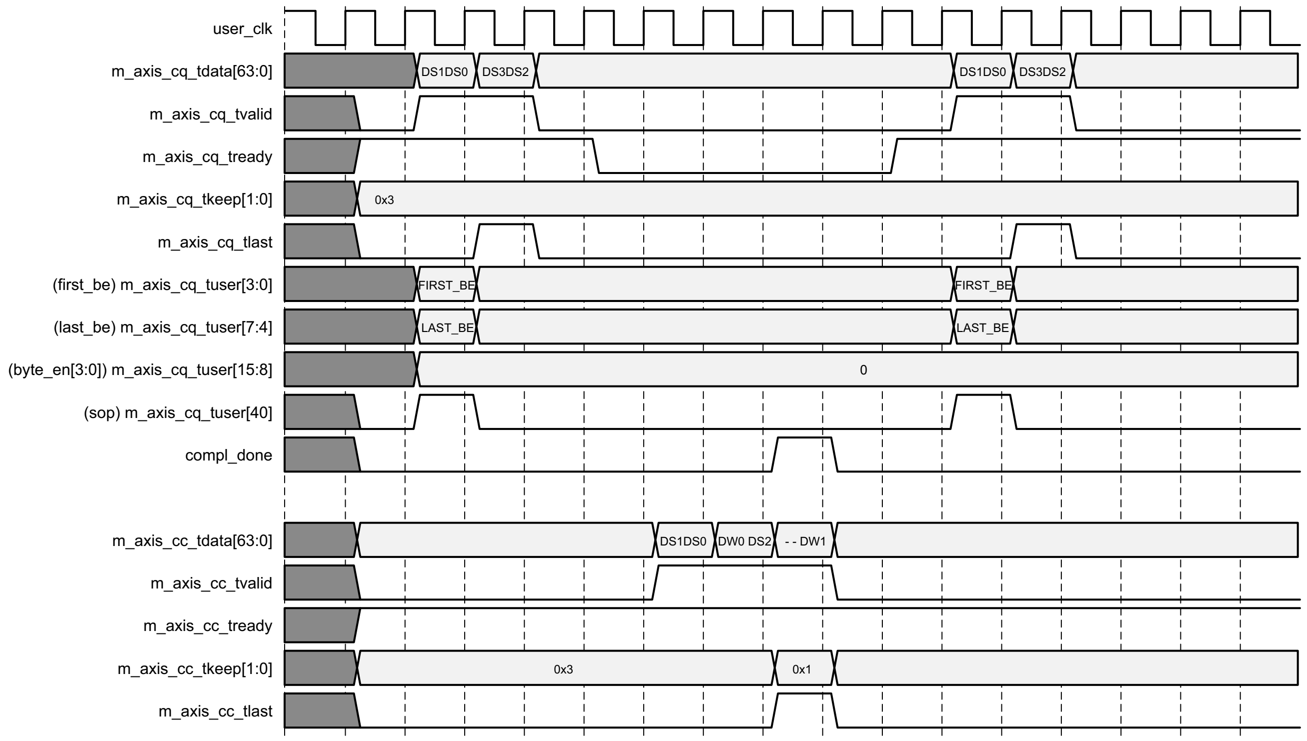


Figure 5-5: **Back-to-Back Read Transactions**

PIO Write Transaction

Figure 5-6 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.

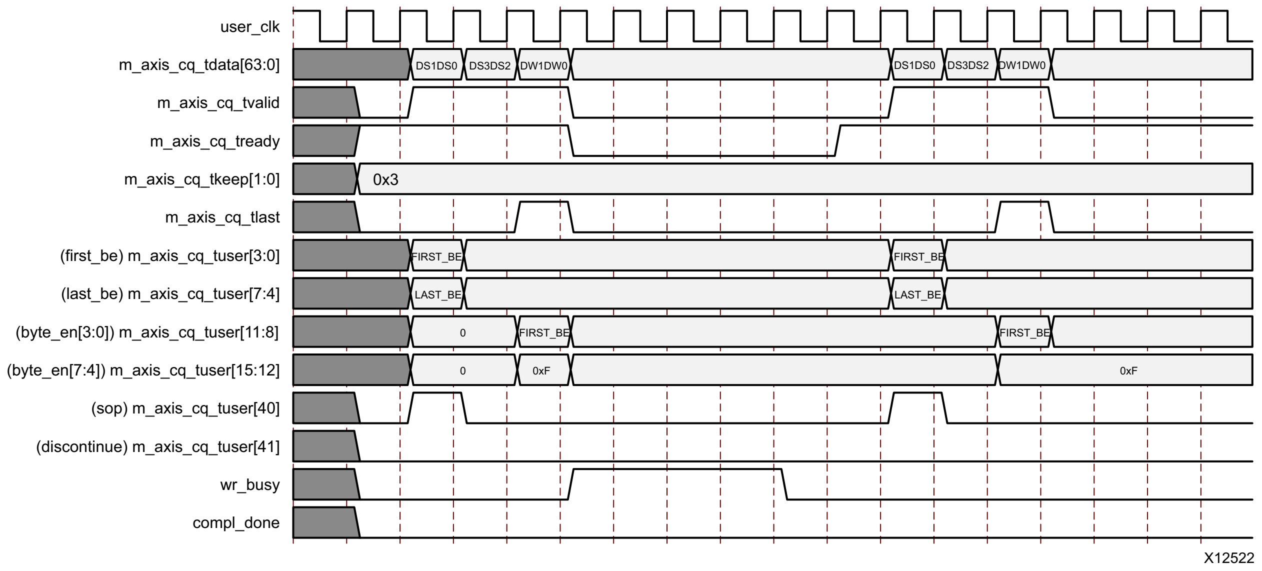

user_clk
m_axis_cq_tdata[63:0]
DS1DS0
DS3DS2
DW1DW0
m_axis_cq_tvalid
m_axis_cq_tready
m_axis_cq_tkeep[1:0]
0x3
m_axis_cq_tlast
(first_be) m_axis_cq_tuser[3:0]
FIRST_BE
(last_be) m_axis_cq_tuser[7:4]
LAST_BE
(byte_en[3:0]) m_axis_cq_tuser[11:8]
0
FIRST_BE
(byte_en[7:4]) m_axis_cq_tuser[15:12]
0
0xF
(sop) m_axis_cq_tuser[40]
(discontinue) m_axis_cq_tuser[41]
wr_busy
compl_done
X12522

Figure 5-6: **Back-to-Back Write Transactions**

Figure 5-7 shows how the blocks are connected in an overall system view.

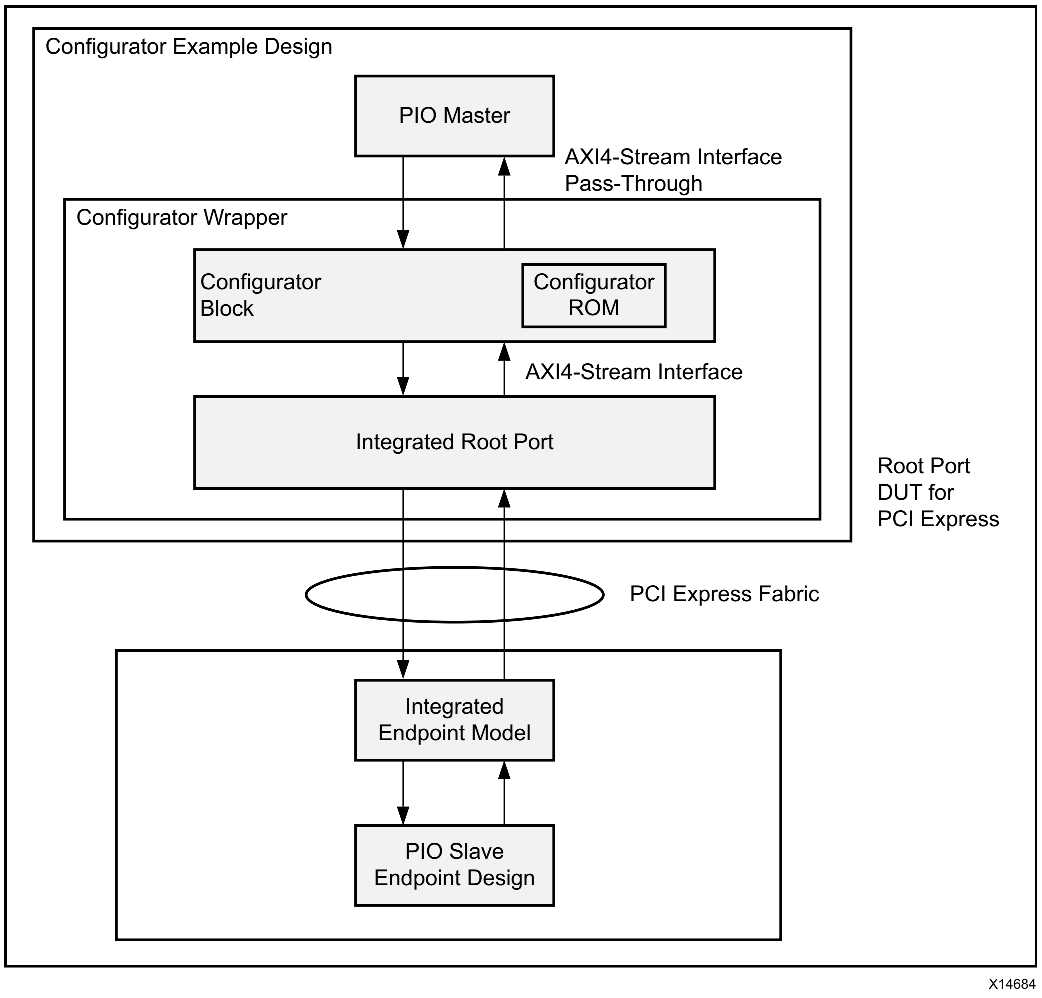

Configurator Example Design
PIO Master
AXI4-Stream Interface Pass-Through
Configurator Wrapper
Configurator Block
Configurator ROM
AXI4-Stream Interface
Integrated Root Port
Root Port DUT for PCI Express
PCI Express Fabric
Integrated Endpoint Model
PIO Slave Endpoint Design
X14684

Figure 5-7: **Configurator Example Design**

Configurator File Structure

Table 5-4 defines the Configurator example design file structure.

Table 5-4: **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

Table 5-4: **Example Design File Structure *(Cont'd)***

File	Description
cgator_controller.v	Configurator transmit engine
cgator_cfg_rom.data	Configurator ROM file
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.

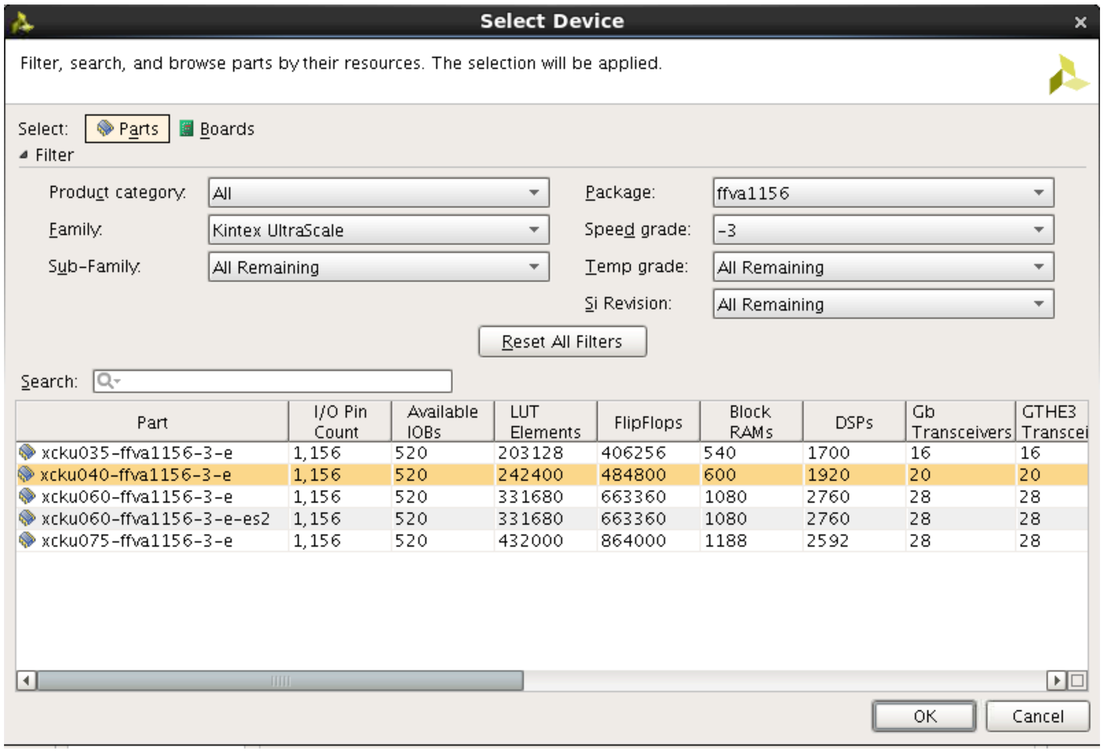
Generating the Core

To generate a core using the default values in the Vivado IDE, follow these steps:

1. Start the Vivado IP catalog.
2. Select **File** > **New Project**.

3. Enter a project name and location, then click **Next.** This example uses `project_name.cpg` and `project_dir`.
4. In the New Project wizard pages, *do not* add sources, existing IP, or constraints.
5. From the Part tab (Figure 5-8), select these options:
   - **Family**: Kintex UltraScale
   - **Device**: xcku040
   - **Package**: ffva1156
   - **Speed Grade**: -3

   Note: If an unsupported silicon device is selected, the core is grayed out (unavailable) in the list of cores.


Select Device

Filter, search, and browse parts by their resources. The selection will be applied.

Select: Parts Boards

Filter

Product category: All
Family: Kintex UltraScale
Sub-Family: All Remaining
Package: ffva1156
Speed grade: -3
Temp grade: All Remaining
Si Revision: All Remaining

Reset All Filters

Search:

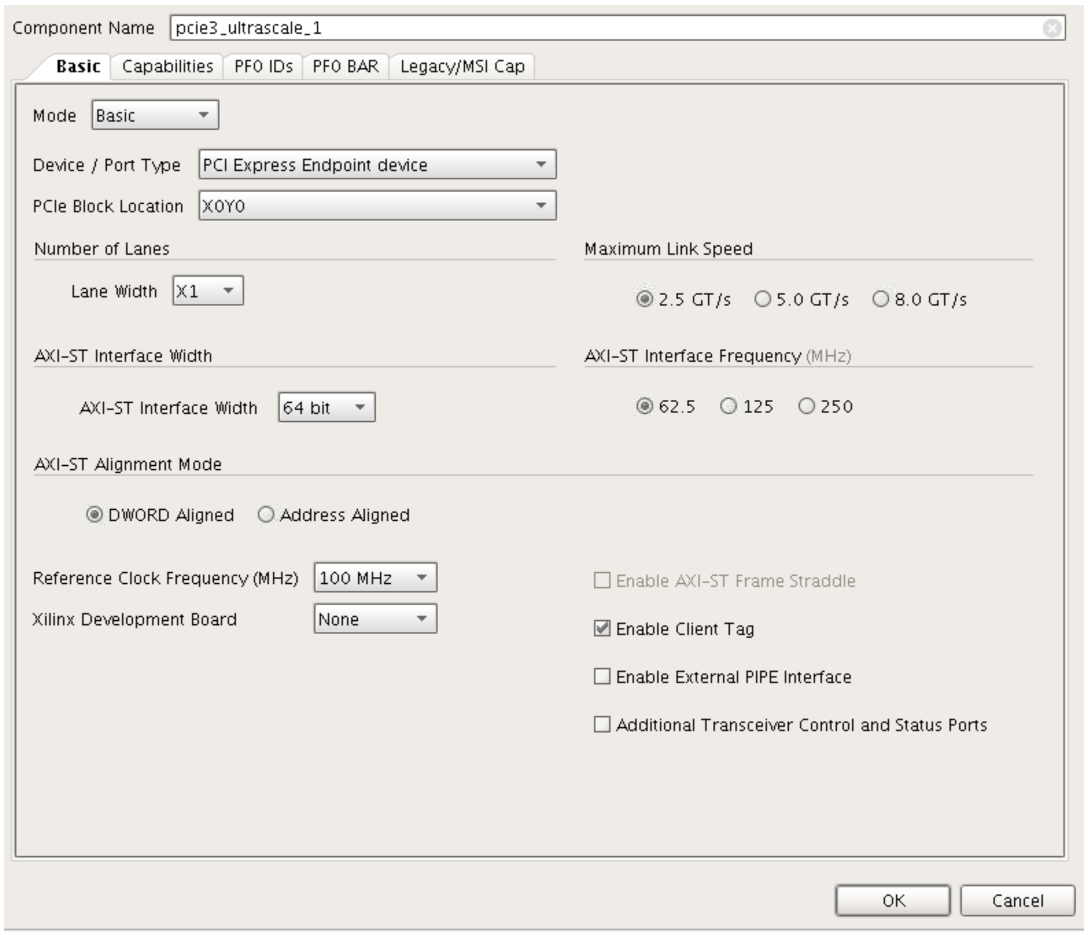
Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	DSPs	Gb Transceivers	GTHE3 Transcei
xcku035-ffva1156-3-e	1,156	520	203128	406256	540	1700	16	16
xcku040-ffva1156-3-e	1,156	520	242400	484800	600	1920	20	20
xcku060-ffva1156-3-e	1,156	520	331680	663360	1080	2760	28	28
xcku060-ffva1156-3-e-es2	1,156	520	331680	663360	1080	2760	28	28
xcku075-ffva1156-3-e	1,156	520	432000	864000	1188	2592	28	28

OK Cancel

Figure 5-8: **Part Selection**

6. In the final project summary page, click **OK**.
7. In the Vivado IP catalog, expand **Standard Bus Interfaces** > **PCI Express**, and double-click the **UltraScale+ Devices Integrated Block for PCIe** core to display the Customize IP dialog box.
8. In the Component Name field, enter a name for the core.

   Note: `<component_name>` is used in this example.



Component Name: pcie3_ultrascale_1

Mode: Basic

Device / Port Type: PCI Express Endpoint device

PCIe Block Location: X0Y0

Number of Lanes — Lane Width: X1

Maximum Link Speed: [✓ 2.5 GT/s] 5.0 GT/s 8.0 GT/s

AXI-ST Interface Width — AXI-ST Interface Width: 64 bit

AXI-ST Interface Frequency (MHz): [✓ 62.5] 125 250

AXI-ST Alignment Mode: [✓ DWORD Aligned] Address Aligned

Reference Clock Frequency (MHz): 100 MHz

Xilinx Development Board: None

☐ Enable AXI-ST Frame Straddle

[✓ Enable Client Tag]

☐ Enable External PIPE Interface

☐ Additional Transceiver Control and Status Ports

Figure 5-9: **Configuration Parameters**

9. From the Device/Port Type drop-down menu, select the appropriate device/port type of the core (**Endpoint** or **Root Port**).
10. Click **OK** to generate the core using the default parameters.
11. In the Design sources tab, right-click the XCI file, and select **Generate**.
12. Select **All** to generate the core with 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)
- Mentor Graphics QuestaSim
- Cadence Incisive Enterprise Simulator (IES)
- Synopsys Verilog Compiler Simulator (VCS)

The simulator uses the example design test bench and test cases provided along with the example design for both the design configurations.

For any project (PCI Express core) generated out of the box, the simulations using the default Vivado simulator can be 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**.

In Vivado IDE, change the simulation settings as follows:

1. In the Flow Navigator, under Simulation, select **Simulation Settings**.
2. Set the **Target simulator** to **QuestaSim/ModelSim Simulator**, **Incisive Enterprise Simulator (IES)** or **Verilog Compiler Simulator**.
3. In the simulator tab, select **Run Simulation > Run behavioral simulation**.
4. When prompted, click **Yes** to change and then run the simulator.

Endpoint Configuration

The simulation environment provided with the UltraScale+ Devices Integrated Block for PCIe 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, page 312.

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

This chapter contains information about the test bench provided in the Vivado® Design Suite.

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 are complete, allowing you to dedicate efforts to verifying the correct 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

Figure 6-1 illustrates the illustrates the Root Port Model coupled with the PIO design.

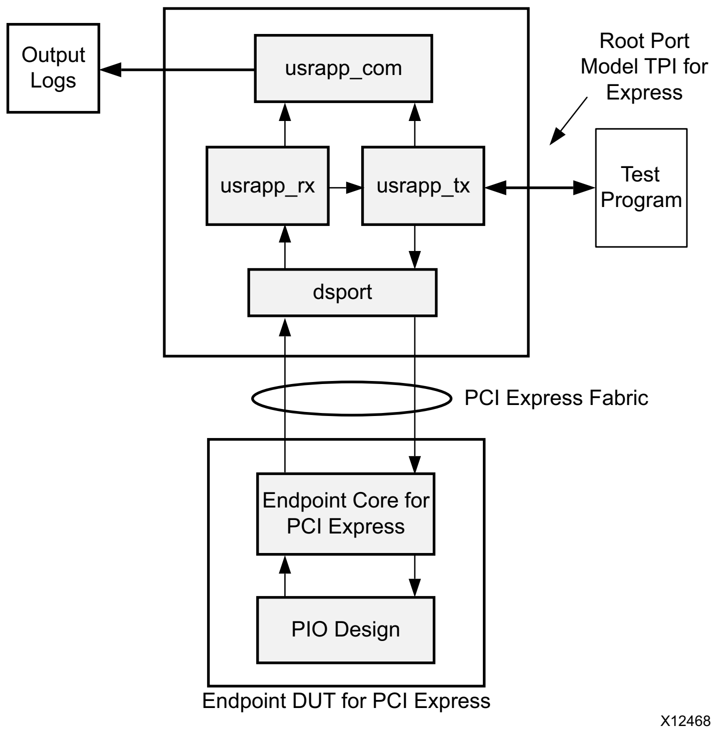


Figure 6-1: **Root Port Model and Top-Level Endpoint**

Architecture

The Root Port Model consists of these blocks, illustrated in Figure 6-1:

- 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* [Ref 2], 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 in simulation, except in the Recovery Speed_1 LTSSM state, where the timeouts are not scaled.

Test Selection

Table 6-1 describes the tests provided with the Root Port Model, followed by specific sections for Verilog test selection.

Table 6-1: **Root Port Model Provided Tests**

Test Name	Test in Verilog	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 value provided to TESTNAME, which is defined in the test files `sample_tests1.v` and `pio_tests.v`. This mechanism is used for Mentor Graphics QuestaSim. 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)* [Ref 19].

Verilog Flow

The Root Port Model provides a mechanism for outputting the simulation waveform to a file by specifying the `+dump_all` command line parameter to the simulator.

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 files (`tx.dat`, `rx.dat`, and `error.dat`) during each simulation run. The log files, `rx.dat` and `tx.dat`, 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 log file `error.dat` 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 located within `error.dat`.

Parallel Test Programs

There are two classes of tests are 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, page 317 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 PCIe 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 1DW 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

```
else if(testname == "pio_writeReadBack_test1"
begin
// This test performs a 32 bit write to a 32 bit Memory space and performs a read back
TSK_SIMULATION_TIMEOUT(10050);
TSK_SYSTEM_INITIALIZATION;
TSK_BAR_INIT;
for (ii = 0; ii <= 6; ii = ii + 1) begin
    if (BAR_INIT_P_BAR_ENABLED[ii] > 2'b00) // bar is enabled
     case(BAR_INIT_P_BAR_ENABLED[ii])
             2'b01 : // IO SPACE
            begin
                 $display("[%t] : NOTHING: to IO 32 Space BAR %x", $realtime, ii);
            end
             2'b10 : // MEM 32 SPACE
               begin
                $display("[%t] : Transmitting TLPs to Memory 32 Space BAR %x",
                             $realtime, ii);
        //-----------------------------------------------------------------------
        // Event : Memory Write 32 bit TLP
        //-----------------------------------------------------------------------
                  DATA_STORE[0] = 8'h04;
                  DATA_STORE[1] = 8'h03;
                  DATA_STORE[2] = 8'h02;
                  DATA_STORE[3] = 8'h01;
                  P_READ_DATA = 32'hffff_ffff; // make sure P_READ_DATA has known initial value
                  TSK_TX_MEMORY_WRITE_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, BAR_INIT_P_BAR[ii][31:0] , 4'hF,
4'hF, 1'b0);
                  TSK_TX_CLK_EAT(10);
                  DEFAULT_TAG = DEFAULT_TAG + 1;
            //-----------------------------------------------------------------------
            // Event : Memory Read 32 bit TLP
            //-----------------------------------------------------------------------
                  TSK_TX_MEMORY_READ_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, BAR_INIT_P_BAR[ii][31:0], 4'hF,
4'hF);
                  TSK_WAIT_FOR_READ_DATA;
                  if  (P_READ_DATA != {DATA_STORE[3], DATA_STORE[2], DATA_STORE[1], DATA_STORE[0] })
                    begin
                     $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);
                   end
                else
                  begin
                     $display("[%t] : Test PASSED --- Write Data: %x successfully received", $realtime,
P_READ_DATA);
                  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 these tasks, which are further defined in these tables.

- Table 6-2, Test Setup Tasks
- Table 6-3, TLP Tasks
- Table 6-4, BAR Initialization Tasks
- Table 6-5, Example PIO Design Tasks
- Table 6-6, Expectation Tasks

Table 6-2: **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.
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.

Table 6-3: **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 6-3: **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.

Table 6-3: **TLP Tasks *(Cont'd)***

Name	Input(s)		Description
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.
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.

Table 6-3: **TLP Tasks *(Cont'd)***

Name	Input(s)		Description
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.
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.

Table 6-3: **TLP Tasks *(Cont'd)***

Name	Input(s)		Description
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.

Table 6-4: **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.
TSK_BAR_SCAN	None	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. The task stores this information in the global array BAR_INIT_P_BAR_RANGE[]. This task should only be called after TSK_SYSTEM_INITIALIZATION.
TSK_BUILD_PCIE_MAP	None	Performs memory and I/O mapping algorithm and allocates Memory 32, Memory 64, and I/O space based on the Endpoint requirements. This task has been customized to work in conjunction with the limitations of the PIO design and should only be called after completion of TSK_BAR_SCAN.
TSK_DISPLAY_PCIE_MAP	None	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 TSK_BUILD_PCIE_MAP.

Table 6-5: **Example PIO Design Tasks**

Name	Input(s)		Description
TSK_TX_READBACK_CONFIG	None		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. This task should only be called after TSK_SYSTEM_INITIALIZATION.
TSK_MEM_TEST_DATA_BUS	bar_index	2:0	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 bar_index. For an exhaustive test, this task should be called four times, once for each block RAM used in the PIO design.
TSK_MEM_TEST_ADDR_BUS	bar_index nBytes	2:0 31:0	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. 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.
TSK_MEM_TEST_DEVICE	bar_index nBytes	2:0 31:0	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. 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.
TSK_RESET	Reset	0	Initiates PERSTn. Forces the `PERSTn` signal to assert the reset. Use TSK_RESET (1'b1) to assert the reset and TSK_RESET (1'b0) to release the reset signal.
TSK_MALFORMED	malformed_bits	7:0	Control bits for creating malformed TLPs: 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

Table 6-6: **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 6-6: **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. As 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

Figure 6-1 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 ep 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

A simulation script file ,`simulate_mti.do`, is provided with the model to facilitate simulation with the Mentor Graphics QuestaSim 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 Chapter 4.

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* [Ref 2], 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 in 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)* [Ref 19].

Output Logging

The test bench outputs messages, captured in the simulation log, indicating 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™ device PCIe core to the UltraScale+™ PCIe core.

New Ports

Table A-1 lists the new ports in the UltraScale+ device core relative to the UltraScale device core.

Table A-1: **New Ports in the UltraScale+ Core**

Port Names	Direction	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	

Table A-1: **New Ports in the UltraScale+ Core *(Cont'd)***

Port Names	Direction	Notes
cfg_dev_id_pf0[15:0]	I	IDs are user accessible through I/Os
cfg_dev_id_pf1[15:0]	I	
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

Table A-2 lists the ports for which widths were changed between the UltraScale device core and the UltraScale+ device core.

Table A-2: **Port Width Changes in the UltraScale+ Device Core**

Port Name	Direction	Notes
pcie_rq_tag_av[3:0]	O	
pcie_tfc_nph_av[3:0]	O	
pcie_tfc_npd_av[3:0]	O	

Table A-2: **Port Width Changes in the UltraScale+ Device Core *(Cont'd)***

Port Name	Direction	Notes
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	
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

Table A-3 lists the ports that were deprecated in the UltraScale+ device core relative to the UltraScale device core.

Table A-3: **Ports Not Available in the UltraScale+ Core**

Port Name	Direction	Notes
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	

Table A-3: **Ports Not Available in the UltraScale+ Core *(Cont'd)***

Port Name	Direction	Notes
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	

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

Table A-4 shows the changes to parameters in the current version of the core.

Table A-4: **New Parameter**

User Parameter Name	Display Name	New/Change/ Removed	Details	Default Value
legacy_ext_pcie_cfg_space_enabled	PCI Express Legacy Extended Configuration Space Enable	New	Enabled legacy extended pcie configuration space	Unchecked
ext_xvc_vsec_enable	Add the PCIe XVC-VSEC to the Examlpe Design	New	Adds the PCIe XVC-VSEC to the Example Design when selected. PCIe Extended Configuration Space must be enabled to select this option.	Unchecked
gt_drp_clk_src	GT DRP Clock Source	New	Selects the clock source from GT DRP interface	Internal

Table A-4: **New Parameter *(Cont'd)***

User Parameter Name	Display Name	New/Change/ Removed	Details	Default Value
free_run_freq	Free Run Clock Frequency (MHz)	New	Allows to select the 100_MHz or 125_MHz frequency when gt drp external clock soure and(or) IBERT is enabled.	100_MHz
en_l23_entry	Support PM_L23 Entry	New	When checked supports PM_L23 Entry. PM_ENABLE_L23_ENTRY attribute will be set to TRUE	Unchecked

Port Changes

There are no port changes with this core release.

GT Locations

This appendix provides a list of recommended GT locations for this IP core.

- Virtex UltraScale+ Device GT Locations
- Kintex UltraScale+ Device GT Locations
- Zynq UltraScale+ Device GT Locations

The FPGA package pins are derived directly from the GT locations listed in Available Integrated Blocks for PCI Express. The specific package pins are not listed in this guide because they can change between device packages. From the Vivado Device view, use the following commands to print out the FPGA package pins and their associated function for a specific GT location:

```
foreach pin [get_package_pins -of_objects [get_sites GTHE3_CHANNEL_<location>]]
{puts "Pin $pin: function [get_property PIN_FUNC $pin]"}
```

This appendix also provides the available GT Quads for this IP core.

- Available GT Quads

Virtex UltraScale+ Device GT Locations

Table B-1 provides a list of the recommended Virtex® UltraScale+™ device GT locations.

Table B-1: **Virtex UltraScale+ Device GT Locations**

Device	Package	PCIe Blocks	x1 (Lane0)	x2 (Lane0 to Lane1)	x4 (Lane0 to Lane3)	x8 (Lane0 to Lane7)	x16 (Lane0 to Lane15)
XCVU3P	FFVC1517	X0Y1	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X1Y0	X1Y15	X1Y15 - X1Y14	X1Y15 - X1Y12	X1Y15 - X1Y8	X1Y15 - X1Y0

Table B-1: **Virtex UltraScale+ Device GT Locations *(Cont'd)***

Device	Package	PCIe Blocks	x1 (Lane0)	x2 (Lane0 to Lane1)	x4 (Lane0 to Lane3)	x8 (Lane0 to Lane7)	x16 (Lane0 to Lane15)
XCVU5P XCVU7P	FLVA2104	X0Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	Not supported
		X0Y3	X0Y35	X0Y35 - X0Y34	X0Y35 - X0Y32	X0Y35 - X0Y28	
		X1Y0	X1Y15	X1Y15 - X1Y14	X1Y15 - X1Y12	X1Y15 - X1Y8	X1Y15 - X1Y0
		X1Y2	X1Y35	X1Y35 - X1Y34	X1Y35 - X1Y32	X1Y35 - X1Y28	Not supported
	FLVB2104	X0Y1	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X0Y3	X0Y39	X0Y39 - X0Y38	X0Y39 - X0Y36	X0Y39 - X0Y32	X0Y39 - X0Y24
		X1Y0	X1Y15	X1Y15 - X1Y14	X1Y15 - X1Y12	X1Y15 - X1Y8	X1Y15 - X1Y0
		X1Y2	X1Y35	X1Y35 - X1Y34	X1Y35 - X1Y32	X1Y35 - X1Y28	X1Y35 - X1Y20
XCVU5P XCVU7P cont'd	FLVC2104	X0Y1	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X0Y3	X0Y39	X0Y39 - X0Y38	X0Y39 - X0Y36	X0Y39 - X0Y32	X0Y39 - X0Y24
		X1Y0	X1Y15	X1Y15 - X1Y14	X1Y15 - X1Y12	X1Y15 - X1Y8	X1Y15 - X1Y0
		X1Y2	X1Y35	X1Y35 - X1Y34	X1Y35 - X1Y32	X1Y35 - X1Y28	X1Y35 - X1Y20

Table B-1: **Virtex UltraScale+ Device GT Locations *(Cont'd)***

Device	Package	PCIe Blocks	x1 (Lane0)	x2 (Lane0 to Lane1)	x4 (Lane0 to Lane3)	x8 (Lane0 to Lane7)	x16 (Lane0 to Lane15)
XCVU9P	FLGA2104	X0Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	Not supported
		X0Y3	X0Y35	X0Y35 - X0Y34	X0Y35 - X0Y32	X0Y35 - X0Y28	
		X1Y2	X1Y35	X1Y35 - X1Y34	X1Y35 - X1Y32	X1Y35 - X1Y28	X1Y35 - X1Y20
		X1Y4	X1Y55	X1Y55 - X1Y54	X1Y55 - X1Y52	X1Y55 - X1Y48	Not supported
	FLGB2104	X0Y1	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X0Y3	X0Y39	X0Y39 - X0Y38	X0Y39 - X0Y36	X0Y39 - X0Y32	X0Y39 - X0Y24
		X1Y2	X1Y35	X1Y35 - X1Y34	X1Y35 - X1Y32	X1Y35 - X1Y28	X1Y35 - X1Y20
		X1Y4	X1Y55	X1Y55 - X1Y54	X1Y55 - X1Y52	X1Y55 - X1Y48	X1Y55 - X1Y40
	FLGC2104	X0Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	Not supported
		X0Y3	X0Y39	X0Y39 - X0Y38	X0Y39 - X0Y36	X0Y39 - X0Y32	X0Y39 - X0Y24
		X0Y5	X0Y59	X0Y59 - X0Y58	X0Y59 - X0Y56	X0Y59 - X0Y52	X0Y59 - X0Y44
		X1Y0	X1Y11	X1Y11 - X1Y10	X1Y11 - X1Y8	X1Y11 - X1Y4	Not supported
		X1Y2	X1Y35	X1Y35 - X1Y34	X1Y35 - X1Y32	X1Y35 - X1Y28	X1Y35 - X1Y20
		X1Y4	X1Y55	X1Y55 - X1Y54	X1Y55 - X1Y52	X1Y55 - X1Y48	X1Y55 - X1Y40
	FLGA2577	X0Y1	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X0Y3	X0Y39	X0Y39 - X0Y38	X0Y39 - X0Y36	X0Y39 - X0Y32	X0Y39 - X0Y24
		X0Y5	X0Y59	X0Y59 - X0Y58	X0Y59 - X0Y56	X0Y59 - X0Y52	X0Y59 - X0Y44
		X1Y0	X1Y15	X1Y15 - X1Y14	X1Y15 - X1Y12	X1Y15 - X1Y8	X1Y15 - X1Y0
		X1Y2	X1Y35	X1Y35 - X1Y34	X1Y35 - X1Y32	X1Y35 - X1Y28	X1Y35 - X1Y20
		X1Y4	X1Y55	X1Y55 - X1Y54	X1Y55 - X1Y52	X1Y55 - X1Y48	X1Y55 - X1Y40
	FSGD2104	X0Y1	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X0Y3	X0Y35	X0Y35 - X0Y34	X0Y35 - X0Y32	X0Y35 - X0Y28	X0Y35 - X0Y20
		X0Y5	X0Y50	X0Y50 - X0Y50	X0Y51 - X0Y48	Not supported	Not supported
		X1Y2	X1Y35	X1Y35 - X1Y34	X1Y35 - X1Y32	X1Y35 - X1Y28	X1Y35 - X1Y20
		X1Y4	X1Y55	X1Y55 - X1Y54	X1Y55 - X1Y52	X1Y55 - X1Y48	X1Y55 - X1Y40

Table B-1: **Virtex UltraScale+ Device GT Locations *(Cont'd)***

Device	Package	PCIe Blocks	x1 (Lane0)	x2 (Lane0 to Lane1)	x4 (Lane0 to Lane3)	x8 (Lane0 to Lane7)	x16 (Lane0 to Lane15)
XCVU11P	FLGA2577	X0Y0	X1Y15	X1Y15 - X1Y14	X1Y15 - X1Y12	X1Y15 - X1Y8	X1Y15 - X1Y0
		X0Y1	X1Y31	X1Y31 - X1Y30	X1Y31 - X1Y28	X1Y31 - X1Y24	X1Y31 - X1Y16
		X0Y2	X1Y47	X1Y47 - X1Y46	X1Y47 - X1Y44	X1Y47 - X1Y40	X1Y47 - X1Y32
	FLGB2104	X0Y0	X1Y15	X1Y15 - X1Y14	X1Y15 - X1Y12	X1Y15 - X1Y8	X1Y15 - X1Y0
		X0Y1	X1Y31	X1Y31 - X1Y30	X1Y31 - X1Y28	X1Y31 - X1Y24	X1Y31 - X1Y16
		X0Y2	X1Y39	X1Y39 - X1Y38	X1Y39 - X1Y36	X1Y39 - X1Y32	X1Y39 - X1Y24
	FLGC2104	X0Y0	X1Y15	X1Y15 - X1Y14	X1Y15 - X1Y12	X1Y15 - X1Y8	X1Y15 - X1Y0
		X0Y1	X1Y31	X1Y31 - X1Y30	X1Y31 - X1Y28	X1Y31 - X1Y24	X1Y31 - X1Y16
		X0Y2	X1Y47	X1Y47 - X1Y46	X1Y47 - X1Y44	X1Y47 - X1Y40	X1Y47 - X1Y32
	FLGF1924	X0Y0	X1Y15	X1Y15 - X1Y14	X1Y15 - X1Y12	X1Y15 - X1Y8	X1Y15 - X1Y0
		X0Y1	X1Y31	X1Y31 - X1Y30	X1Y31 - X1Y28	X1Y31 - X1Y24	X1Y31 - X1Y16
		X0Y2	X1Y39	X1Y39 - X1Y38	X1Y39 - X1Y36	X1Y39 - X1Y32	X1Y39 - X1Y24
	FSGD2104	X0Y0	X1Y15	X1Y15 - X1Y14	X1Y15 - X1Y12	X1Y15 - X1Y8	X1Y15 - X1Y0
		X0Y1	X1Y31	X1Y31 - X1Y30	X1Y31 - X1Y28	X1Y31 - X1Y24	X1Y31 - X1Y16
		X0Y2	X1Y39	X1Y39 - X1Y38	X1Y39 - X1Y36	X1Y39 - X1Y32	Not supported
XCVU13P	FHGA2104	X0Y0	X1Y15	X1Y15 - X1Y14	X1Y15 - X1Y12	X1Y15 - X1Y8	X1Y15 - X1Y0
		X0Y1	X1Y31	X1Y31 - X1Y30	X1Y31 - X1Y28	X1Y31 - X1Y24	X1Y31 - X1Y16
		X0Y2	X1Y63	X1Y63 - X1Y62	X1Y63 - X1Y60	X1Y63 - X1Y56	Not supported
	FHGB2104	X0Y1	X1Y31	X1Y31 - X1Y30	X1Y31 - X1Y28	X1Y31 - X1Y24	X1Y31 - X1Y16
		X0Y2	X1Y47	X1Y47 - X1Y46	X1Y47 - X1Y44	X1Y47 - X1Y40	X1Y47 - X1Y32
		X0Y3	X1Y55	X1Y55 - X1Y54	X1Y55 - X1Y52	X1Y55 - X1Y48	Not supported
	FHGC2014	X0Y0	X1Y15	X1Y15 - X1Y14	X1Y15 - X1Y12	X1Y15 - X1Y8	Not supported
		X0Y1	X1Y31	X1Y31 - X1Y30	X1Y31 - X1Y28	X1Y31 - X1Y24	X1Y31 - X1Y16
		X0Y2	X1Y47	X1Y47 - X1Y46	X1Y47 - X1Y44	X1Y47 - X1Y40	X1Y47 - X1Y32
		X0Y3	X1Y55	X1Y55 - X1Y54	X1Y55 - X1Y52	X1Y55 - X1Y48	Not supported
	FLGA2577	X0Y0	X1Y15	X1Y15 - X1Y14	X1Y15 - X1Y12	X1Y15 - X1Y8	X1Y15 - X1Y0
		X0Y1	X1Y31	X1Y31 - X1Y30	X1Y31 - X1Y28	X1Y31 - X1Y24	X1Y31 - X1Y16
		X0Y2	X1Y47	X1Y47 - X1Y46	X1Y47 - X1Y44	X1Y47 - X1Y40	X1Y47 - X1Y32
		X0Y3	X1Y63	X1Y63 - X1Y62	X1Y63 - X1Y60	X1Y63 - X1Y56	X1Y47 - X1Y32
	FIGD2104	X0Y0	X1Y15	X1Y15 - X1Y14	X1Y15 - X1Y12	X1Y15 - X1Y8	X1Y15 - X1Y0
		X0Y1	X1Y31	X1Y31 - X1Y30	X1Y31 - X1Y28	X1Y31 - X1Y24	X1Y31 - X1Y16
		X0Y2	X1Y47	X1Y47 - X1Y46	X1Y47 - X1Y44	X1Y47 - X1Y40	X1Y47 - X1Y32
		X0Y3	X1Y55	X1Y55 - X1Y54	X1Y55 - X1Y52	X1Y55 - X1Y48	Not supported

Kintex UltraScale+ Device GT Locations

Table B-2 provides a list of the recommended Kintex® UltraScale+ device GT locations.

Table B-2: **Kintex UltraScale+ Device GT Locations**

Device	Package	PCIe Blocks	x1 (Lane0)	x2 (Lane0 to Lane1)	x4 (Lane0 to Lane3)	x8 (Lane0 to Lane7)	x16 (Lane0 to Lane15)
XCKU11P	FFVE1517	X0Y2	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X0Y3	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
	FFVA1156	X0Y2	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	Not supported
		X0Y3	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	Not supported
		X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
	FFVD900	X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
XCKU15P	FFVE1517	X0Y2	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X0Y3	X0Y23	X0Y23 - X0Y22	X0Y23 - X0Y20	X0Y23 - X0Y16	X0Y23 - X0Y8
		X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y23	X0Y23 - X0Y22	X0Y23 - X0Y20	X0Y23 - X0Y16	X0Y23 - X0Y8
		X1Y2	X0Y31	X0Y31 - X0Y30	X0Y31 - X0Y28	X0Y31 - X0Y24	X0Y31 - X0Y16
	FFVA1156	X0Y2	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	Not supported
		X0Y3	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	Not supported
		X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y2	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
	FFVA1760	X0Y2	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X0Y3	X0Y23	X0Y23 - X0Y22	X0Y23 - X0Y20	X0Y23 - X0Y16	X0Y23 - X0Y8
		X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y23	X0Y23 - X0Y22	X0Y23 - X0Y20	X0Y23 - X0Y16	X0Y23 - X0Y8
		X1Y2	X0Y31	X0Y31 - X0Y30	X0Y31 - X0Y28	X0Y31 - X0Y24	X0Y31 - X0Y16
	FFVE1760	X0Y2	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X0Y3	X0Y23	X0Y23 - X0Y22	X0Y23 - X0Y20	X0Y23 - X0Y16	X0Y23 - X0Y8
		X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y23	X0Y23 - X0Y22	X0Y23 - X0Y20	X0Y23 - X0Y16	X0Y23 - X0Y8
		X1Y2	X0Y31	X0Y31 - X0Y30	X0Y31 - X0Y28	X0Y31 - X0Y24	X0Y31 - X0Y16

Table B-2: **Kintex UltraScale+ Device GT Locations *(Cont'd)***

Device	Package	PCIe Blocks	x1 (Lane0)	x2 (Lane0 to Lane1)	x4 (Lane0 to Lane3)	x8 (Lane0 to Lane7)	x16 (Lane0 to Lane15)
XCKU3P XCKU5P	FFVA676	X0Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
	FFVB676	X0Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
	FFVD900	X0Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
	SFVB784	X0Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0

Zynq UltraScale+ Device GT Locations

Table B-3 provides a list of the recommended Zynq® UltraScale+™ device GT locations.

Table B-3: **Zynq UltraScale+ Core Pinouts**

Device	Package	PCIe Blocks	x1 (Lane0)	x2 (Lane0 to Lane1)	x4 (Lane0 to Lane3)	x8 (Lane0 to Lane7)	x16 (Lane0 to Lane15)
XCZU11EG	FFVC1760	X0Y2	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X0Y3	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
	FFVB1517	X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
	FFVC1156	X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
	FFVF1517	X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
XCZU17EG	FFCV1760	X0Y2	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X0Y3	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y2	X0Y31	X0Y31 - X0Y30	X0Y31 - X0Y28	X0Y31 - X0Y24	X0Y31 - X0Y16
	FFVE1924	X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y2	X0Y31	X0Y31 - X0Y30	X0Y31 - X0Y28	X0Y31 - X0Y24	X0Y31 - X0Y16
	FFVB1517	X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0

Table B-3: **Zynq UltraScale+ Core Pinouts *(Cont'd)***

Device	Package	PCIe Blocks	x1 (Lane0)	x2 (Lane0 to Lane1)	x4 (Lane0 to Lane3)	x8 (Lane0 to Lane7)	x16 (Lane0 to Lane15)
XCZU17EG *continued*	FFVD1760	X0Y2	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X0Y3	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y2	X0Y31	X0Y31 - X0Y30	X0Y31 - X0Y28	X0Y31 - X0Y24	X0Y31 - X0Y16
XCZU19EG	FFVC1760	X0Y2	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X0Y3	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y2	X0Y31	X0Y31 - X0Y30	X0Y31 - X0Y28	X0Y31 - X0Y24	X0Y31 - X0Y16
	FFVE1924	X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y2	X0Y31	X0Y31 - X0Y30	X0Y31 - X0Y28	X0Y31 - X0Y24	X0Y31 - X0Y16
	FFVB1517	X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y2	X0Y31	X0Y31 - X0Y30	X0Y31 - X0Y28	X0Y31 - X0Y24	X0Y31 - X0Y16
	FFVD1760	X0Y2	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X0Y3	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X1Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X1Y2	X0Y31	X0Y31 - X0Y30	X0Y31 - X0Y28	X0Y31 - X0Y24	X0Y31 - X0Y16
XCZU4EV XCZU5EV	FBVB900	X0Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X0Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
	SFVC784	X0Y0	X0Y7	X0Y7 - X0Y6	X0Y7 - X0Y4	Not supported	Not supported
		X0Y1	X0Y7	X0Y7 - X0Y6	X0Y7 - X0Y4		
XCZU7EV	FBVB900	X0Y0	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X0Y1	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
	FFVC1156	X0Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X0Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
	FFVF1517	X0Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X0Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
XCZU21DR	FFVD1156	X0Y0	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X0Y1	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4

Table B-3: **Zynq UltraScale+ Core Pinouts *(Cont'd)***

Device	Package	PCIe Blocks	x1 (Lane0)	x2 (Lane0 to Lane1)	x4 (Lane0 to Lane3)	x8 (Lane0 to Lane7)	x16 (Lane0 to Lane15)
XCZU25DR	FFVE1156	X0Y0	X0Y11	X0Y11 - X0Y10	X0Y11 - X0Y8	X0Y11 - X0Y4	Not supported
XCZU28DR XCZU27DR		X0Y0	X0Y11	X0Y11 - X0Y10	X0Y11 - X0Y8	X0Y11 - X0Y4	Not supported
		X0Y1	X0Y11	X0Y11 - X0Y10	X0Y11 - X0Y8	X0Y11 - X0Y4	
XCZU28DR XCZU27DR	FFVG1517	X0Y0	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X0Y1	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
XCZU25DR	FFVG1517	X0Y0	X0Y11	X0Y11 - X0Y10	X0Y11 - X0Y8	X0Y11 - X0Y4	Not supported
XCZU29DR	FFVF1760	X0Y0	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X0Y1	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
XCZU4CG XCZU5CG XCZU4EG XCZU5EG	FBVB900	X0Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X0Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
	SFVC784	X0Y0	X0Y7	X0Y7 - X0Y6	X0Y7 - X0Y4	Not supported	Not supported
		X0Y1	X0Y7	X0Y7 - X0Y6	X0Y7 - X0Y4		
XCZU7CG XCZU7EG	FBVB900	X0Y0	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
		X0Y1	X0Y19	X0Y19 - X0Y18	X0Y19 - X0Y16	X0Y19 - X0Y12	X0Y19 - X0Y4
	FFVC1156	X0Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X0Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
	FFVF1517	X0Y0	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0
		X0Y1	X0Y15	X0Y15 - X0Y14	X0Y15 - X0Y12	X0Y15 - X0Y8	X0Y15 - X0Y0

Available GT Quads

Table B-4, Table B-5, and Table B-6 show the different GT Quad options available for different devices and packages. These options can be seen in the IP Customization GUI when in the **GT Quad** drop down menu when the **Advanced** option is selected for Mode parameter and when **Enable GT Selection** option is checked.

Virtex UltraScale+ Devices

Table B-4: **Virtex Ultrascale+ Devices**

Device	Package	PCIe Blocks Left	Quads Available for Selection	PCIe Blocks Right	Quads Available for Selection
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
XCVU5P	FLVA2104	X0Y1	GTY_Quad_127, GTY_Quad_126, GTY_Quad_125	X1Y0	GTY_Quad_227, GTY_Quad_226, GTY_Quad_225, GTY_Quad_224
		X0Y3	GTY_Quad_132, GTY_Quad_131, GTY_Quad_130	X1Y2	GTY_Quad_233, GTY_Quad_232, GTY_Quad_231
	FLVB2104	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
		X0Y3	GTY_Quad_133, GTY_Quad_132, GTY_Quad_131, GTY_Quad_130	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	X1Y0	GTY_Quad_228, GTY_Quad_227, GTY_Quad_226, GTY_Quad_225, GTY_Quad_224
		X0Y3	GTY_Quad_133, GTY_Quad_132, GTY_Quad_131, GTY_Quad_130	X1Y2	GTY_Quad_233, GTY_Quad_232, GTY_Quad_231, GTY_Quad_230, GTY_Quad_229
XCVU7P	FLVA2104	X0Y1	GTY_Quad_127, GTY_Quad_126, GTY_Quad_125	X1Y0	GTY_Quad_227, GTY_Quad_226, GTY_Quad_225, GTY_Quad_224
		X0Y3	GTY_Quad_132, GTY_Quad_131, GTY_Quad_130	X1Y2	GTY_Quad_233, GTY_Quad_232, GTY_Quad_231
	FLVB2104	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
		X0Y3	GTY_Quad_133, GTY_Quad_132, GTY_Quad_131, GTY_Quad_130	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	X1Y0	GTY_Quad_228, GTY_Quad_227, GTY_Quad_226, GTY_Quad_225, GTY_Quad_224
		X0Y3	GTY_Quad_133, GTY_Quad_132, GTY_Quad_131, GTY_Quad_130	X1Y2	GTY_Quad_233, GTY_Quad_232, GTY_Quad_231, GTY_Quad_230, GTY_Quad_229

Table B-4: **Virtex Ultrascale+ Devices *(Cont'd)***

Device	Package	PCIe Blocks Left	Quads Available for Selection	PCIe Blocks Right	Quads Available for Selection
XCVU9P	FLGA2104	X0Y1	GTY_Quad_122, GTY_Quad_121, GTY_Quad_120	X1Y0	No bonded GTs
		X0Y3	GTY_Quad_127, GTY_Quad_126, GTY_Quad_125	X1Y2	GTY_Quad_227, GTY_Quad_226, GTY_Quad_225, GTY_Quad_224
		X0Y5	No Bonded GTs	X1Y4	GTY_Quad_233, GTY_Quad_232, GTY_Quad_231
	FLGB2104	X0Y1	GTY_Quad_123, GTY_Quad_122, GTY_Quad_121, GTY_Quad_120	X1Y0	No bonded GTs
		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
		X0Y5	No Bonded GTs	X1Y4	GTY_Quad_233, GTY_Quad_232, GTY_Quad_231, GTY_Quad_230, GTY_Quad_229
	FLGC2104	X0Y1	GTY_Quad_122, GTY_Quad_121, GTY_Quad_120	X1Y0	GTY_Quad_222, GTY_Quad_221, GTY_Quad_220
		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
		X0Y5	GTY_Quad_133, GTY_Quad_132, GTY_Quad_131, GTY_Quad_130	X1Y4	GTY_Quad_233, GTY_Quad_232, GTY_Quad_231, GTY_Quad_230, GTY_Quad_229
	FLGA2577	X0Y1	GTY_Quad_123, GTY_Quad_122, GTY_Quad_121, GTY_Quad_120	X1Y0	GTY_Quad_223, GTY_Quad_222, GTY_Quad_221, GTY_Quad_220, GTY_Quad_219
		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
		X0Y5	GTY_Quad_133, GTY_Quad_132, GTY_Quad_131, GTY_Quad_130	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	X1Y0	No bonded GTs
		X0Y3	GTY_Quad_127, GTY_Quad_126, GTY_Quad_125, GTY_Quad_124	X1Y2	GTY_Quad_228, GTY_Quad_227, GTY_Quad_226, GTY_Quad_225, GTY_Quad_224
		X0Y5	GTY_Quad_131	X1Y4	GTY_Quad_233, GTY_Qyad_232, GTY_Quad_231, GTY_Quad_230, GTY_Quad_229

Table B-4: **Virtex Ultrascale+ Devices *(Cont'd)***

Device	Package	PCIe Blocks Left	Quads Available for Selection	PCIe Blocks Right	Quads Available for Selection
XCVU11P	FLGA2577	None	No PCIE Blocks on Left Side	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	None	No PCIE Blocks on Left Side	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_233, GTY_Quad_232
	FLGC2104	None	No PCIE Blocks on Left Side	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	None	No PCIE Blocks on Left Side	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_233, GTY_Quad_232
	FSGD2104	None	No PCIE Blocks on Left Side	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_233, GTY_Quad_232
XCVU13P	FHGA2104	None	No PCIE Blocks on Left Side	X0Y0	No bonded GTs
				X0Y1	GTY_Quad_227, GTY_Quad_226, GTY_Quad_225, GTY_Quad_224
				X0Y2	GTY_Quad_231, GTY_Quad_230, GTY_Quad_229
				X0Y3	No bonded GTs
	FHGB2104	None	No PCIE Blocks on Left Side	X0Y0	No bonded GTs
				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_233, GTY_Quad_232

Table B-4: **Virtex Ultrascale+ Devices *(Cont'd)***

Device	Package	PCIe Blocks Left	Quads Available for Selection	PCIe Blocks Right	Quads Available for Selection
XCVU13P cont'd	FHGC2014	None	No PCIE Blocks on Left Side	X0Y0	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	GTY_Quad_233, GTY_Quad_232
	FLGA2577	None	No PCIE Blocks on Left Side	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	None	No PCIE Blocks on Left Side	X0Y0	No bonded GTs
				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_233, GTY_Quad_232

Kintex UltraScale+ Devices

Table B-5: **Kintex UltraScale+ Devices**

Device	Package	PCIE Blocks Left	Quads Available for selection	PCIE Blocks Right	Quads Available for selection
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
XCVU5P	FLVA2104	X0Y1	GTY_Quad_127,GTY_Quad_126, GTY_Quad_125	X1Y0	GTY_Quad_227, GTY_Quad_226, GTY_Quad_225, GTY_Quad_224
		X0Y3	GTY_Quad_132,GTY_Quad_131, GTY_Quad_130	X1Y2	GTY_Quad_233,GTY_Quad_232, GTY_Quad_231
	FLVB2104	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
		X0Y3	GTY_Quad_133,GTY_Quad_132, GTY_Quad_131,GTY_Quad_130	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	X1Y0	GTY_Quad_228, GTY_Quad_227, GTY_Quad_226, GTY_Quad_225, GTY_Quad_224
		X0Y3	GTY_Quad_133,GTY_Quad_132, GTY_Quad_131,GTY_Quad_130	X1Y2	GTY_Quad_233,GTY_Quad_232, GTY_Quad_231,GTY_Quad_230, GTY_Quad_229
XCVU7P	FLVA2104	X0Y1	GTY_Quad_127,GTY_Quad_126, GTY_Quad_125	X1Y0	GTY_Quad_227, GTY_Quad_226, GTY_Quad_225, GTY_Quad_224
		X0Y3	GTY_Quad_132, GTY_Quad_131,GTY_Quad_130	X1Y2	GTY_Quad_233,GTY_Quad_232, GTY_Quad_231
	FLVB2104	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
		X0Y3	GTY_Quad_133,GTY_Quad_132, GTY_Quad_131,GTY_Quad_130	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	X1Y0	GTY_Quad_228, GTY_Quad_227, GTY_Quad_226, GTY_Quad_225, GTY_Quad_224
		X0Y3	GTY_Quad_133,GTY_Quad_132, GTY_Quad_131,GTY_Quad_130	X1Y2	GTY_Quad_233,GTY_Quad_232, GTY_Quad_231,GTY_Quad_230, GTY_Quad_229

Table B-5: **Kintex UltraScale+ Devices *(Cont'd)***

Device	Package	PCIE Blocks Left	Quads Available for selection	PCIE Blocks Right	Quads Available for selection
XCVU9P	FLGA2104	X0Y1	GTY_Quad_122,GTY_Quad_121, GTY_Quad_120	X1Y0	No bonded GTs
		X0Y3	GTY_Quad_127,GTY_Quad_126, GTY_Quad_125	X1Y2	GTY_Quad_227,GTY_Quad_226, GTY_Quad_225,GTY_Quad_224
		X0Y5	No Bonded GTs	X1Y4	GTY_Quad_233,GTY_Quad_232, GTY_Quad_231
	FLGB2104	X0Y1	GTY_Quad_123,GTY_Quad_122, GTY_Quad_121,GTY_Quad_120	X1Y0	No bonded GTs
		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
		X0Y5	No Bonded GTs	X1Y4	GTY_Quad_233,GTY_Quad_232, GTY_Quad_231,GTY_Quad_230, GTY_Quad_229
	FLGC2104	X0Y1	GTY_Quad_122,GTY_Quad_121, GTY_Quad_120	X1Y0	GTY_Quad_222, GTY_Quad_221, GTY_Quad_220
		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
		X0Y5	GTY_Quad_133,GTY_Quad_132, GTY_Quad_131,GTY_Quad_130	X1Y4	GTY_Quad_233,GTY_Quad_232, GTY_Quad_231,GTY_Quad_230, GTY_Quad_229
	FLGA2577	X0Y1	GTY_Quad_123,GTY_Quad_122, GTY_Quad_121,GTY_Quad_120	X1Y0	GTY_Quad_223, GTY_Quad_222, GTY_Quad_221, GTY_Quad_220, GTY_Quad_219
		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
		X0Y5	GTY_Quad_133, GTY_Quad_132, GTY_Quad_131, GTY_Quad_130	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	X1Y0	No bonded GTs
		X0Y3	GTY_Quad_127,GTY_Quad_126, GTY_Quad_125,GTY_Quad_124	X1Y2	GTY_Quad_228,GTY_Quad_227, GTY_Quad_226, GTY_Quad_225, GTY_Quad_224
		X0Y5	GTY_Quad_131	X1Y4	GTY_Quaqd_233,GTY_Qyad_232, GTY_Quad_231, GTY_Quad_230, GTY_Quad_229

Table B-5: **Kintex UltraScale+ Devices *(Cont'd)***

Device	Package	PCIE Blocks Left	Quads Available for selection	PCIE Blocks Right	Quads Available for selection
XCVU11P	FLGA2577	None	No PCIE Blocks on Left Side	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	None	No PCIE Blocks on Left Side	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_233,GTY_Quad_232
	FLGC2104	None	No PCIE Blocks on Left Side	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	None	No PCIE Blocks on Left Side	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_233,GTY_Quad_232
	FSGD2104	None	No PCIE Blocks on Left Side	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_233,GTY_quad_232

Table B-5: **Kintex UltraScale+ Devices *(Cont'd)***

Device	Package	PCIE Blocks Left	Quads Available for selection	PCIE Blocks Right	Quads Available for selection
XCVU13P	FHGA2104	None	No PCIE Blocks on Left Side	X0Y0	No bonded GTs
				X0Y1	GTY_Quad_227, GTY_Quad_226, GTY_Quad_225, GTY_Quad_224
				X0Y2	GTY_Quad_231,GTY_Quad_230, GTY_Quad_229
				X0Y3	No bonded GTs
	FHGB2104	None	No PCIE Blocks on Left Side	X0Y0	No bonded GTs
				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_233,GTY_Quad_232
	FHGC2014	None	No PCIE Blocks on Left Side	X0Y0	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	GTY_Quad_233,GTY_Quad_232
	FLGA2577	None	No PCIE Blocks on Left Side	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	None	No PCIE Blocks on Left Side	X0Y0	No bonded GTs
				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_233,GTY_Quad_232

Zynq Ultrascale+ Devices

Table B-6: **Zynq Ultrascale+ Devices**

Device	Package	PCIe Blocks Left	Quads Available for Selection	PCIe Blocks Right	Quads Available for Selection
XCZU11EG	FFVC1760	X0Y3	GTY_Quad_131, GTY_Quad_130, GTY_Quad_129, GTY_quad_128	X1Y0	GTH_Quad_227,GTH_Quad_226, GTH_Quad_225,GTH_Quad_224
		X0Y2	GTY_Quad_130,GTY_Quad_129, GTY_Quad_128	X1Y1	GTH_Quad_230,GTH_Quad_229, GTH_Quad_228,GTH_Quad_227, GTH_Quad_226
	FFVB1517	X0Y3	No Bonded GTs	X1Y0	GTH_Quad_227,GTH_Quad_226, GTH_Quad_225,GTH_Quad_224
		X0Y2	No Bonded GTs	X1Y1	GTH_Quad_227, GTH_Quad_226
	FFVC1156	X0Y3	No Bonded GTs	X1Y0	GTH_Quad_227,GTH_Quad_226, GTH_Quad_225,GTH_Quad_224
		X0Y2	No Bonded GTs	X1Y1	GTH_Quad_228, GTH_Quad_227,GTH_Quad_226
	FFVF1517	X0Y3	No Bonded GTs	X1Y0	GTH_Quad_227,GTH_Quad_226, GTH_Quad_225,GTH_Quad_224
		X0Y2	No Bonded GTs	X1Y1	GTH_Quad_229,GTH_Quad_228, GTH_Quad_227,GTH_Quad_226
XCZU17EG	FFVC1760	X0Y3	GTY_Quad_131, GTY_Quad_130, GTY_Quad_129, GTY_quad_128	X1Y0	GTH_Quad_227,GTH_Quad_226, GTH_Quad_225,GTH_Quad_224
		X0Y2	GTY_Quad_130,GTY_Quad_129, GTY_Quad_128	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	X0Y3	No Bonded GTs	X1Y0	GTH_Quad_227,GTH_Quad_226, GTH_Quad_225,GTH_Quad_224
		X0Y2		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	X0Y3	No Bonded GTs	X1Y0	GTH_Quad_227,GTH_Quad_226, GTH_Quad_225,GTH_Quad_224
		X0Y2	No Bonded GTs	X1Y1	GTH_Quad_227, GTH_Quad_226
	FFVD1760	X0Y3	GTY_Quad_132,GTY_Quad_131, GTY_Quad_130, GTY_Quad_129	X1Y0	GTH_Quad_227,GTH_Quad_226, GTH_Quad_225,GTH_Quad_224
		X0Y2	GTY_Quad_130,GTY_Quad_129, GTY_Quad_128	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 B-6: **Zynq Ultrascale+ Devices *(Cont'd)***

Device	Package	PCIe Blocks Left	Quads Available for Selection	PCIe Blocks Right	Quads Available for Selection
XCZU19EG	FFVC1760	X0Y3	GTY_Quad_131, GTY_Quad_130, GTY_Quad_129, GTY_quad_128	X1Y0	GTH_Quad_227,GTH_Quad_226, GTH_Quad_225,GTH_Quad_224
		X0Y2	GTY_Quad_130,GTY_Quad_129, GTY_Quad_128	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	X0Y3	No Bonded GTs	X1Y0	GTH_Quad_227,GTH_Quad_226, GTH_Quad_225,GTH_Quad_224
		X0Y2		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	X0Y3	No Bonded GTs	X1Y0	GTH_Quad_227,GTH_Quad_226, GTH_Quad_225,GTH_Quad_224
		X0Y2		X1Y1	GTH_Quad_227,GTH_Quad_226
				X1Y2	GTH_Quad_227
	FFVD1760	X0Y3	GTY_Quad_132,GTY_Quad_131, GTY_Quad_130, GTY_Quad_129	X1Y0	GTH_Quad_227,GTH_Quad_226, GTH_Quad_225,GTH_Quad_224
		X0Y2	GTY_Quad_130,GTY_Quad_129, GTY_Quad_128	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
XCZU4EV	FBVB900	None	No PCIE Blocks on Left Side	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	None	No PCIE Blocks on Left Side	X0Y0	GTH_Quad_224
				X0Y1	GTH_Quad_224
XCZU5EV	FBVB900	None	No PCIE Blocks on Left Side	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	None	No PCIE Blocks on Left Side	X0Y0	GTH_Quad_224
				X0Y1	GTH_Quad_224

Table B-6: **Zynq Ultrascale+ Devices *(Cont'd)***

Device	Package	PCIe Blocks Left	Quads Available for Selection	PCIe Blocks Right	Quads Available for Selection
XCZU7EV	FBVB900	None	No PCIE Blocks on Left Side	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	None	No PCIE Blocks on Left Side	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	None	No PCIE Blocks on Left Side	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
XCZU21DR	FFVD1156	X0Y0	GTY_Quad_131, GTY_Quad_130, GTY_Quad_129, GTY_quad_128	None	No PCIE Blocks on Left Side
		X0Y1	GTY_Quad_131, GTY_Quad_130, GTY_Quad_129, GTY_quad_128		
XCZU25DR	FFVE1156	X0Y0	GTY_Quad_129, GTY_Quad_128	None	No PCIE Blocks on Left Side
	FFVG1517	X0Y0	GTY_Quad_129, GTY_Quad_128	None	No PCIE Blocks on Left Side
XCZU28DR	FFVG1517	X0Y0	GTY_Quad_131, GTY_Quad_130, GTY_Quad_129, GTY_quad_128	None	No PCIE Blocks on Left Side
		X0Y1	GTY_Quad_131, GTY_Quad_130, GTY_Quad_129, GTY_quad_128		
	FFVE1156	X0Y0	GTY_Quad_129, GTY_Quad_128	None	No PCIE Blocks on Left Side
		X0Y1	GTY_Quad_129, GTY_Quad_128		
XCZU29DR	FFVF1760	X0Y1	GTY_Quad_131, GTY_Quad_130, GTY_Quad_129, GTY_quad_128	None	No PCIE Blocks on Left Side
		X0Y0	GTY_Quad_131, GTY_Quad_130, GTY_Quad_129, GTY_quad_128		
XCZU4CG	FBVB900	None	No PCIE blocks on the Left side	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	None	No PCIE blocks on the Left side	X0Y0	GTH_Quad_224
				X0Y1	GTH_Quad_224

Table B-6: **Zynq Ultrascale+ Devices *(Cont'd)***

Device	Package	PCIe Blocks Left	Quads Available for Selection	PCIe Blocks Right	Quads Available for Selection
XCZU5CG	FBVB900	None	No PCIE blocks on the Left side	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	None	No PCIE blocks on the Left side	X0Y0	GTH_Quad_224
				X0Y1	GTH_Quad_224
XCZU7CG	FBVB900	None	No PCIE blocks on the Left side	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	None	No PCIE blocks on the Left side	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	None	No PCIE blocks on the Left side	X0Y0	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
XCZU4EG	FBVB900	None	No PCIE blocks on the Left side	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	None	No PCIE blocks on the Left side	X0Y0	GTH_Quad_224
				X0Y1	GTH_Quad_224
XCZU5EG	FBVB900	None	No PCIE blocks on the Left side	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	None	No PCIE blocks on the Left side	X0Y0	GTH_Quad_224
				X0Y1	GTH_Quad_224

Table B-6: **Zynq Ultrascale+ Devices *(Cont'd)***

Device	Package	PCIe Blocks Left	Quads Available for Selection	PCIe Blocks Right	Quads Available for Selection
XCZU7EG	FBVB900	None	No PCIE blocks on the Left side	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	None	No PCIE blocks on the Left side	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	None	No PCIE blocks on the Left side	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

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 UltraScale+ Devices Integrated Block for PCIe, 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.

Documentation

This product guide is the main document associated with the UltraScale+ Devices Integrated Block for PCIe. 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 proper 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 UltraScale+ Devices Integrated Block for PCIe

AR: 65751

Technical Support

Xilinx provides technical support in the Xilinx Support web page 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 contact Xilinx Technical Support, navigate to the Xilinx Support web page.

Hardware Debug

Transceiver Control and Status Ports

Table C-1 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 Table C-1 might result in unpredictable behavior of the IP core.*

Table C-1: **Ports Used for Transceiver Debug**

Port	Direction	Width	Description
gt_pcieuserratedone	I	1	Connects to PCIEUSERRATEDONE 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

Table C-1: **Ports Used for Transceiver Debug *(Cont'd)***

Port	Direction	Width	Description
gt_txresetdone	O	1	Connects to TXRESETDONE on transceiver channel primitives
gt_rxresetdone	O	1	Connects to RXRECCLKOUT 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	17	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_pcieuserratestart	O	1	Connects to PCIEUSERRATESTART 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
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

Table C-1: **Ports Used for Transceiver Debug *(Cont'd)***

Port	Direction	Width	Description
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

PCIe DRP Ports

The signals in Table C-2 are available when **PCIe DRP Ports** option is selected.

Table C-2: **PCIe DRP Ports**

Name	Direction	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 signals shown in Table C-3 are available when the **GT Channel DRP** parameter is enabled.

Table C-3: **GT DRP Ports**

Name	Direction	Width	Description
ext_ch_gt_drpaddr	I	No Of Lanes x 10	GT Wizard DRP address
ext_ch_gt_drpen	I	No Of Lanes x 1	GT Wizard DRP enable
ext_ch_gt_drpdi	I	No Of Lanes x 16	GT Wizard DRP data in
ext_ch_gt_drpdo	O	No Of Lanes x 16	GT Wizard DRP data out
ext_ch_gt_drprdy	O	No Of Lanes x 1	GT Wizard DRP ready
ext_ch_gt_drpwe	I	No Of Lanes x 1	GT Wizard DRP write/read

Using Xilinx Virtual Cable to Debug

Introduction

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 Figure D-1 and connect to the Vivado Design Suite debug feature through a TCP/IP socket.

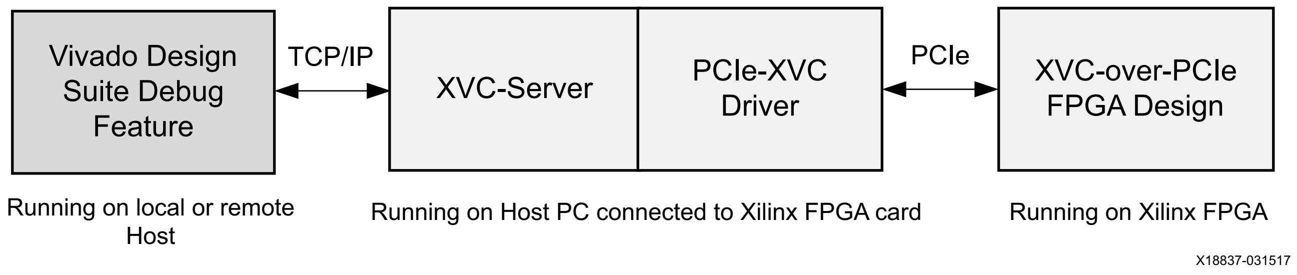

Vivado Design Suite Debug Feature
TCP/IP
XVC-Server
PCIe-XVC Driver
PCIe
XVC-over-PCIe FPGA Design
Running on local or remote Host
Running on Host PC connected to Xilinx FPGA card
Running on Xilinx FPGA
X18837-031517

Figure D-1: **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. 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 debug 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. Figure D-2 describes the connectivity between the PCIe IP and the Debug Bridge IP to implement the PCIe-XVC-VSEC.

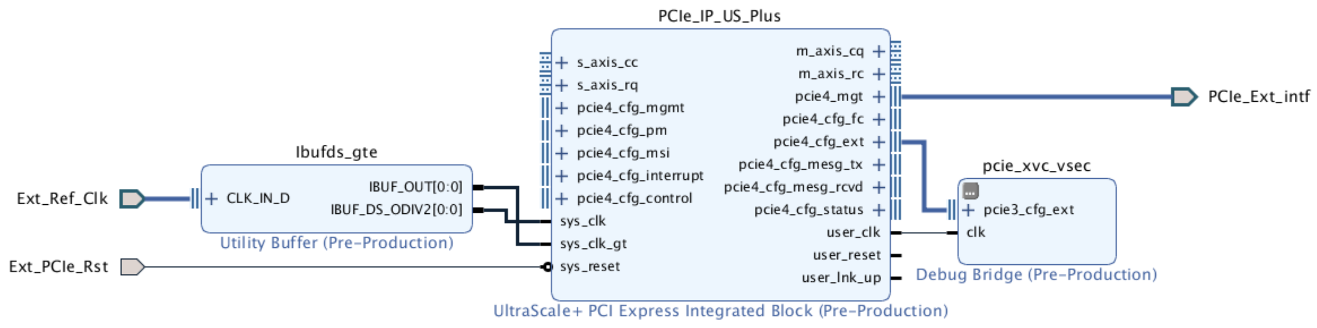


Figure D-2: **XVC-over-PCIe with PCIe Extended Capability Interface**

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. Figure D-3 describes the connectivity between the DMA Subsystem for PCIe IP and the Debug Bridge IP for this implementation.

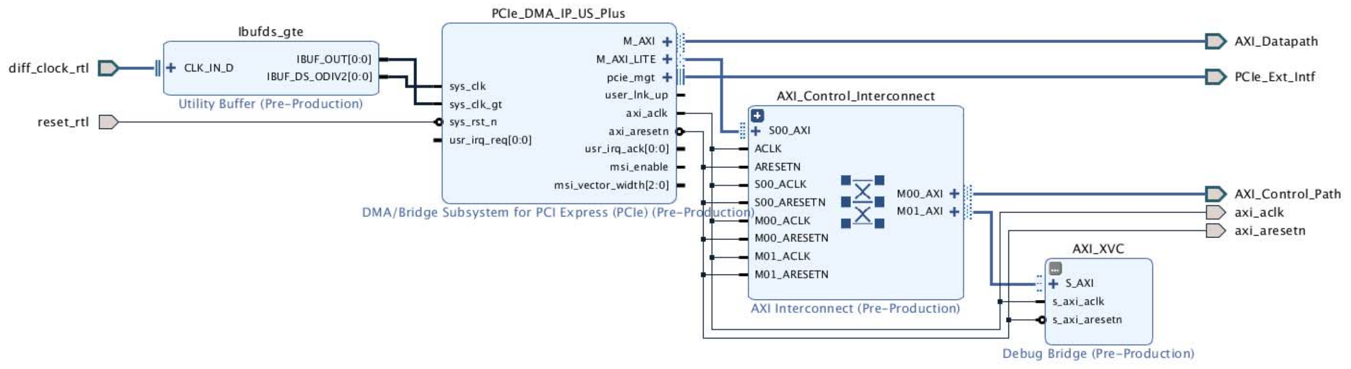


Figure D-3: **XVC over PCIe with AXI4-Lite Interface**

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 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 Table D-1, and Table D-2 show the byte-offset from the base address.

- The PCIe-XVC-VSEC base address is at an offset into the PCIe configuration space within the range of 0x400 and 0xFE0. 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.

Table D-1, and Table D-2 describes 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 D-1: **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 Reserved Register		N/A

Table D-2: **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

PCIe Ext Capability Header (PCIe-XVC-VSEC Only)

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. Table D-3 describes the fields within this register.

Table D-3: **PCIe Ext Capability Header Register Description**

Bit Location	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 range of 0x400 – 0xFF0.	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 D-4: **PCIe XVC VSEC Header Register Description**

Bit Location	Description	Initial Value	Type
15:0	**VSEC ID**: This 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 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 PCIe-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 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 Partial Reconfiguration Designs

Tandem and Partial Reconfiguration (PR) 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 PR or Tandem Field Updates region, an additional debug BSCAN interface should be added to the PR region module definition and left unconnected in the PR region module instantiation.

To add the BSCAN interface to the PR module definition the expropriate ports and port attributes should be added to the PR module 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 PR module definition
// and left unconnected in the PR 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 runtest" *)
(* DEBUG="true" *)
input  S_BSCAN_runtest,
(* 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 will 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

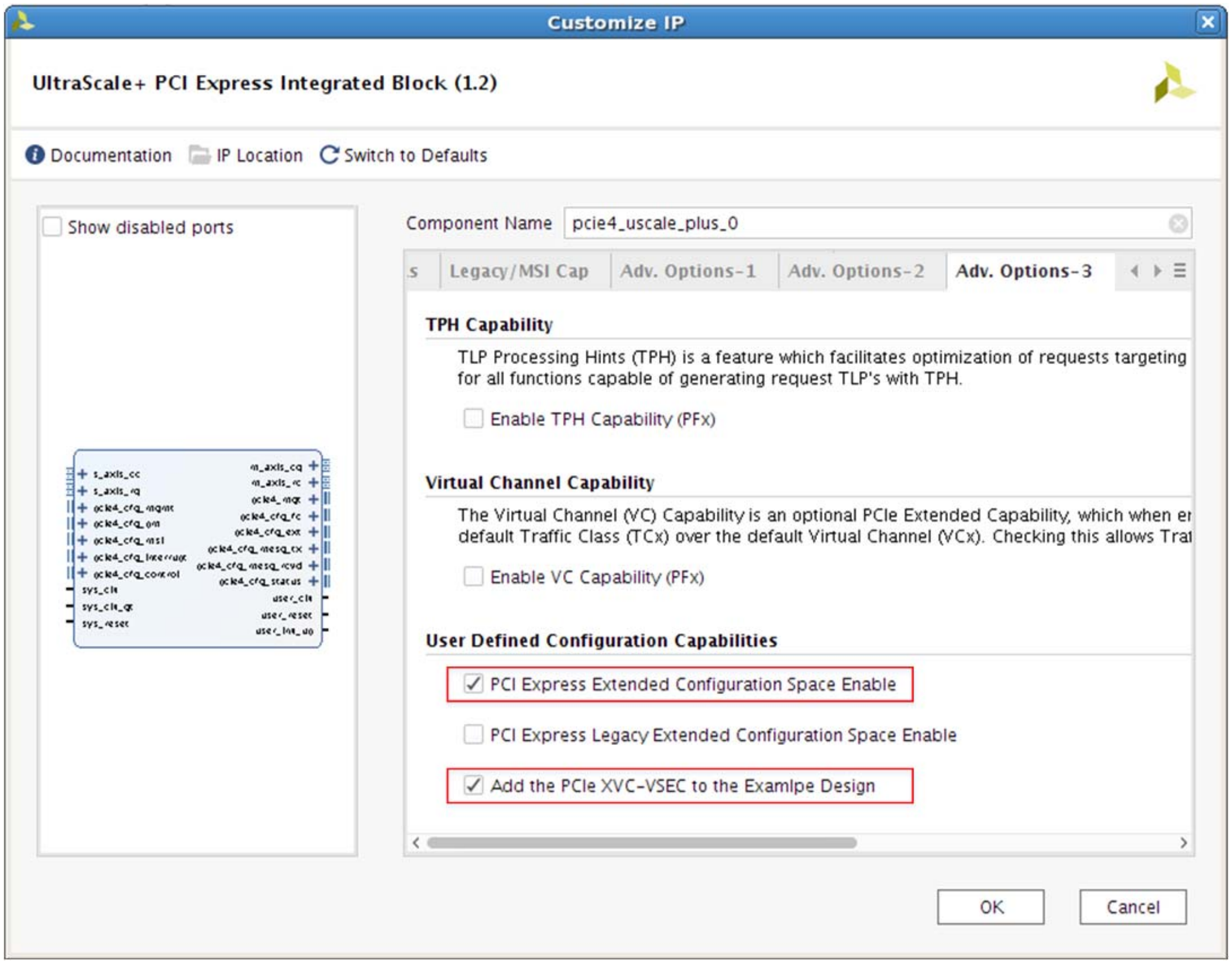
The PCIe-XVC-VSEC has been integrated into the PCIe example design as part of the **Advanced** settings for the Ultrascale+ PCIe Integrated Block IP. This section provides instruction of how to generate the PCIe example design with the PCIe-XVC-VSEC, and then debug the FPGA through PCIe using provided 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 and 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.



Component Name pcie4_uscale_plus_0

TPH Capability

TLP Processing Hints (TPH) is a feature which facilitates optimization of requests targeting 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 er default Traffic Class (TCx) over the default Virtual Channel (VCx). Checking this allows Tra

☐ 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 Examlpe Design

Figure D-4: **Selections in the Adv. Options-3 Tab**

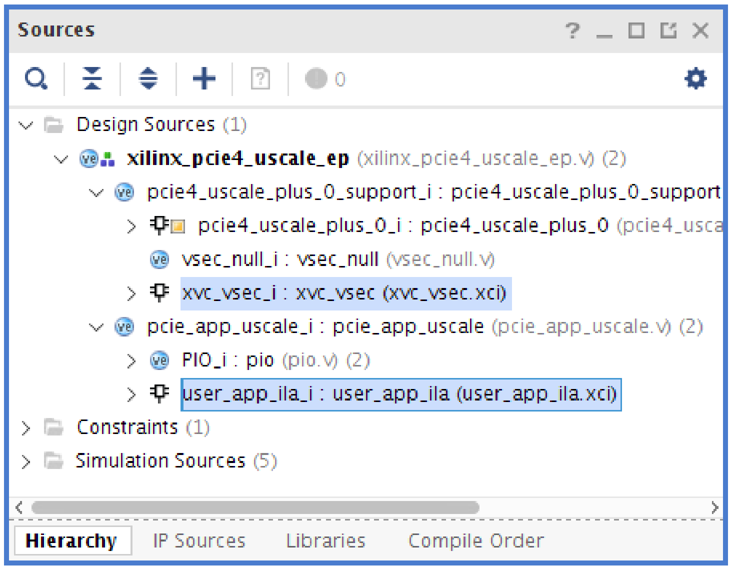
4. Verify the other configuration selections for the PCIe IP. The following selections are needed to configure the driver for your hardware implementation.
   - PCIe Vendor ID (0x10EE for Xilinx)
   - PCIe Device ID (dependent on user selection)
5. Click **OK** to finalize the selection and generate 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.



Sources

Design Sources (1)
xilinx_pcie4_uscale_ep (xilinx_pcie4_uscale_ep.v) (2)
pcie4_uscale_plus_0_support_i : pcie4_uscale_plus_0_support
pcie4_uscale_plus_0_i : pcie4_uscale_plus_0 (pcie4_usca
vsec_null_i : vsec_null (vsec_null.v)
xvc_vsec_i : xvc_vsec (xvc_vsec.xci)
pcie_app_uscale_i : pcie_app_uscale (pcie_app_uscale.v) (2)
PIO_i : pio (pio.v) (2)
user_app_ila_i : user_app_ila (user_app_ila.xci)
Constraints (1)
Simulation Sources (5)

Figure D-5: **Example Design Hierarchy**

Note: The connectivity provided in the example design only responds to valid addresses within the PCIe descriptor chain. Reads to invalid addresses are not responded to with this design and can lock-up the Host PC. Linux tools like `lspci -xxxx` read the entire configuration space, including invalid extended configuration addresses, and can lock-up the Host PC. To avoid this, proper drivers and software should be used so that invalid addresses are not accessed.

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. Since 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 `depmod` 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
   ```

   You should at least see the following message if you run the `dmesg` command:

   ```
   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 in order to 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 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
```

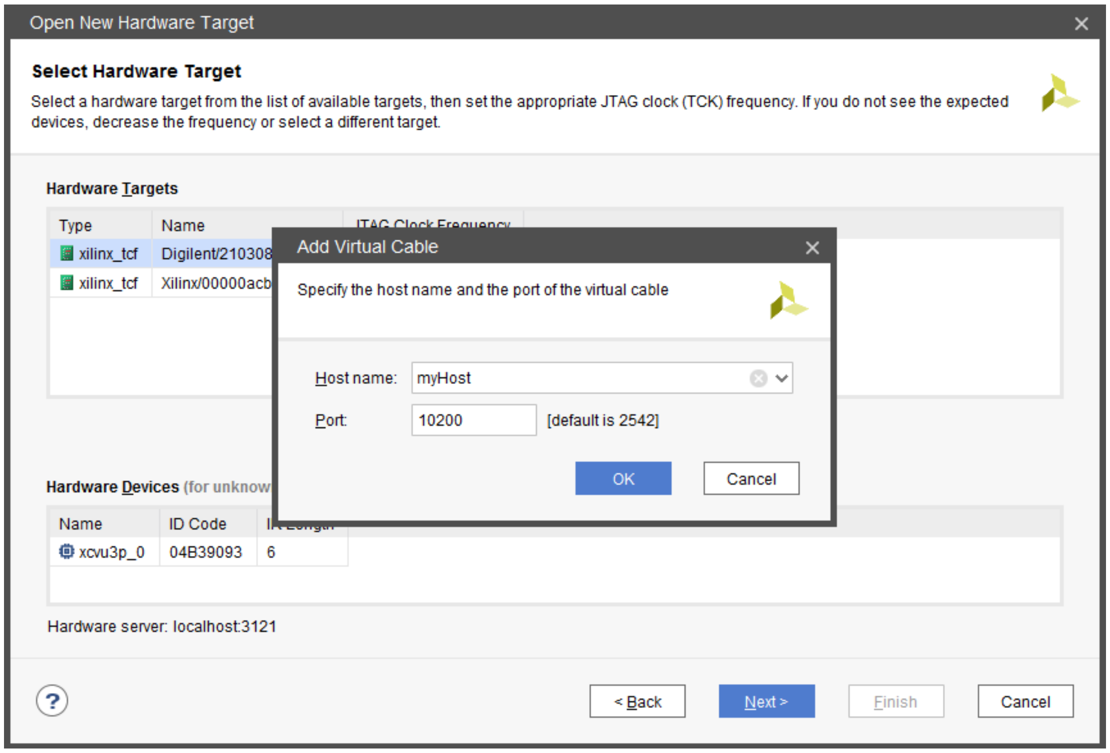
Connecting the Vivado Design Suite to the XVC-Server Application

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 should be used and are 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 in order to connect to the `xvcserver` application. Click **OK**.


Open New Hardware Target

Select Hardware Target

Select a hardware target from the list of available targets, then set the appropriate JTAG clock (TCK) frequency. If you do not see the expected devices, decrease the frequency or select a different target.

Hardware Targets

Type	Name	JTAG Clock Frequency
xilinx_tcf	Digilent/210308	
xilinx_tcf	Xilinx/00000acb	

Hardware Devices (for unknow

Name	ID Code	IR Length
xcvu3p_0	04B39093	6

Hardware server: localhost:3121

Add Virtual Cable

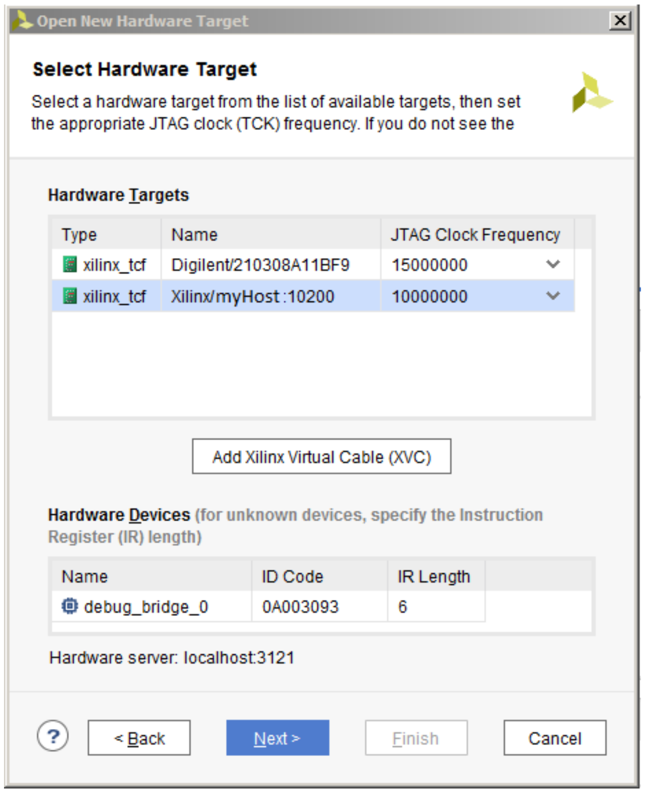
Specify the host name and the port of the virtual cable

Host name: myHost

Port: 10200 [default is 2542]

Figure D-6: **Add Virtual Cable Dialog Box**

8. Select the newly added XVC target from the Hardware Targets table, and click **Next**.


Open New Hardware Target

Select Hardware Target

Select a hardware target from the list of available targets, then set the appropriate JTAG clock (TCK) frequency. If you do not see the

Hardware Targets

Type	Name	JTAG Clock Frequency
xilinx_tcf	Digilent/210308A11BF9	15000000
xilinx_tcf	Xilinx/myHost:10200	10000000

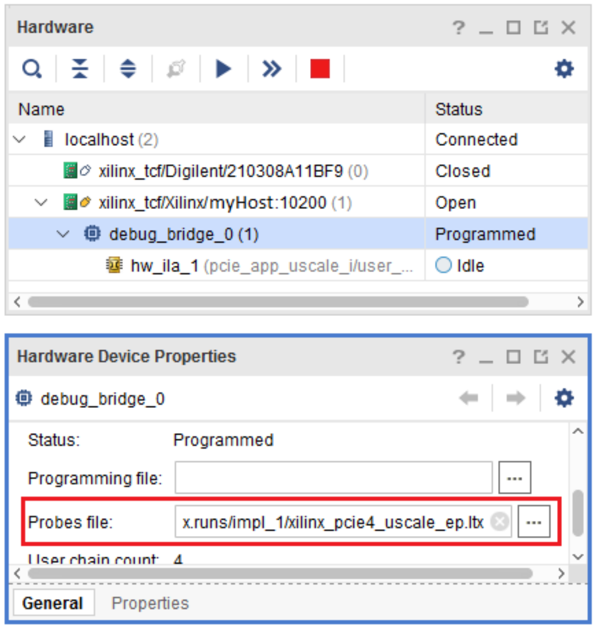
Hardware Devices (for unknown devices, specify the Instruction Register (IR) length)

Name	ID Code	IR Length
debug_bridge_0	0A003093	6

Hardware server: localhost:3121

Figure D-7: **XVC Target**

9. Click **Finish**.
10. In the Hardware Device Properties panel, select the debug bridge target, and assign the appropriate probes `.ltx` file.


Hardware

Name	Status
localhost (2)	Connected
xilinx_tcf/Digilent/210308A11BF9 (0)	Closed
xilinx_tcf/Xilinx/myHost:10200 (1)	Open
debug_bridge_0 (1)	Programmed
hw_ila_1 (pcie_app_uscale_i/user_...	Idle

Hardware Device Properties

debug_bridge_0

Status:	Programmed
Programming file:	
Probes file:	x.runs/impl_1/xilinx_pcie4_uscale_ep.ltx
User chain count:	4

Figure D-8: **Hardware Device Properties**

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 connected 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 PR designs, it is important to terminate the connection during PR operations. During a PR operation where debug cores are present inside the PR 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 PR operation.

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

References

These documents provide supplemental material useful with this product guide:

1. *AMBA AXI4-Stream Protocol Specification*
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 v2.1 Product Guide* (PG194)
5. *DMA Subsystem for PCI Express Product Guide* (PG195)
6. *Virtex-7 FPGA Gen3 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 Architecture GTH Transceivers User Guide* (UG576)
12. *UltraScale Architecture GTY Transceivers User Guide* (UG578)
13. *Kintex UltraScale+ FPGAs Data Sheet: DC and AC Switching Characteristics* (DS922)
14. *Virtex UltraScale+ FPGAs Data Sheet: DC and AC Switching Characteristics* (DS923)
15. *Vivado Design Suite User Guide: Designing with IP* (UG896)
16. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)

17. *Vivado Design Suite User Guide: Getting Started* (UG910)
18. *Vivado Design Suite User Guide: Using Constraints* (UG903)
19. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
20. *ISE to Vivado Design Suite Migration Guide* (UG911)
21. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
22. *In-System IBERT LogiCORE IP Product Guide* (PG246)
23. *ATX Power Supply Design Guide*
24. *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.

Date	Version	Revision
06/07/2017	1.2	• Updated the Available Integrated Blocks for PCI Express - Zynq UltraScale+ table. • Updated port description for cfg_interrupt_msi_function_number. • Updated the TUSER signal in all AXI4-Stream Interface diagrams. • Updated the TUSER signal definition for the 512-bit AXI Interface including Straddle operation. • Minor updates to the Tandem Configuration section. • Updated pcie_cq_np_req to pcie_cq_np_req[0]. • Added missing AXISTEN_IF_ENABLE_MSG_ROUTE Attribute Bit Descriptions table. • Updated the Zynq UltraScale+ Device GT Locations table. • Clarified the XCV driver and software example location in the Using Xilinx Virtual Cable to Debug appendix.
04/05/2017	1.2	• Added the new Using Xilinx Virtual Cable to Debug appendix. • Updated the Tandem Configuration section. • Updated new Ports and Parameters information in the Upgrading appendix.
11/30/2016	1.1	Design Flow Steps chapter: • Updated the SRIOV BAR Size Ranges for Device Configuration table. • Added the PLL Selection and Link Partner TX Preset options in the GT Settings tab. • 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. GT Locations appendix: • Added the Available GT Quads section.

Date	Version	Revision
10/05/2016	1.1	• Moved the performance and resource utilization data to the web. • Updated the Minimum Device Requirements table. • Added tandem configuration support. • Added IBERT ports, GT DRP ports and PCIe ports used for transceiver debug, and the Vivado Design Suite core customization options that support them. • Added to Port Changes table in the Migrating and Upgrading appendix. • Updated the GT Locations table, and the Available Integrated Blocks for PCI Express table for Virtex UltraScale+ devices.
06/08/2016	1.1	• Completer Model option added for Root Port Model test bench. • MXI-X Interrupt Internal (built-in) support added. • GT Setting for insertion loss adjustment added. • QPLL1 support added in Gen2 (5Gb/s) mode.
04/06/2016	1.1	Xilinx Initial Release.

Please Read: Important Legal Notices