UltraScale Devices Gen3 Integrated Block for PCI Express v4.4

LogiCORE IP Product Guide

PG156 (v4.4) December 6, 2024

AMD Adaptive Computing is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this link for more information.

Table of Contents

Chapter 1

Introduction

The AMD UltraScale™ Devices Gen3 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, and 8-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, rev. 3.0. This solution supports the AXI4-Stream interface for the customer user interface.

Features

- High-performance, highly flexible, scalable, and reliable general-purpose I/O core
- Separate Requestor and Completer interfaces simplify design and increase performance
- Endpoint or Root Port configuration
- Multiple Function and Single-Root I/O Virtualization in the Endpoint configuration
- Compliant with PCI and PCI Express power management functions
- Optional Tag Management feature
- Maximum Payload Size (MPS) of up to 1024 bytes
- End-to-End Cyclic Redundancy Check (ECRC)
- Advanced Error Reporting (AER)
- Multi-Vector MSI up to 32 vectors
- MSI-X
- Atomic operations and (transaction layer packets) TLP processing hints

For a full list of features, see Feature Summary.

IP Facts

AMD LogiCORE™ IP Facts Table	
Core Specifics	
Supported Device Family	AMD UltraScale+™ Devices
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	Xilinx Design Constraints (XDC)
Simulation Model	Verilog
Supported S/W Driver	Root Port Driver
Tested Design Flows[2]	
Design Entry	AMD Vivado™ Design Suite
Simulation	For supported simulators, see the *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973).
Synthesis	Vivado synthesis
Support	
Release Notes and Known Issues	Master Answer Record: 57945
All Vivado IP Change Logs	Master Vivado IP Change Logs: 72775
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 *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973).

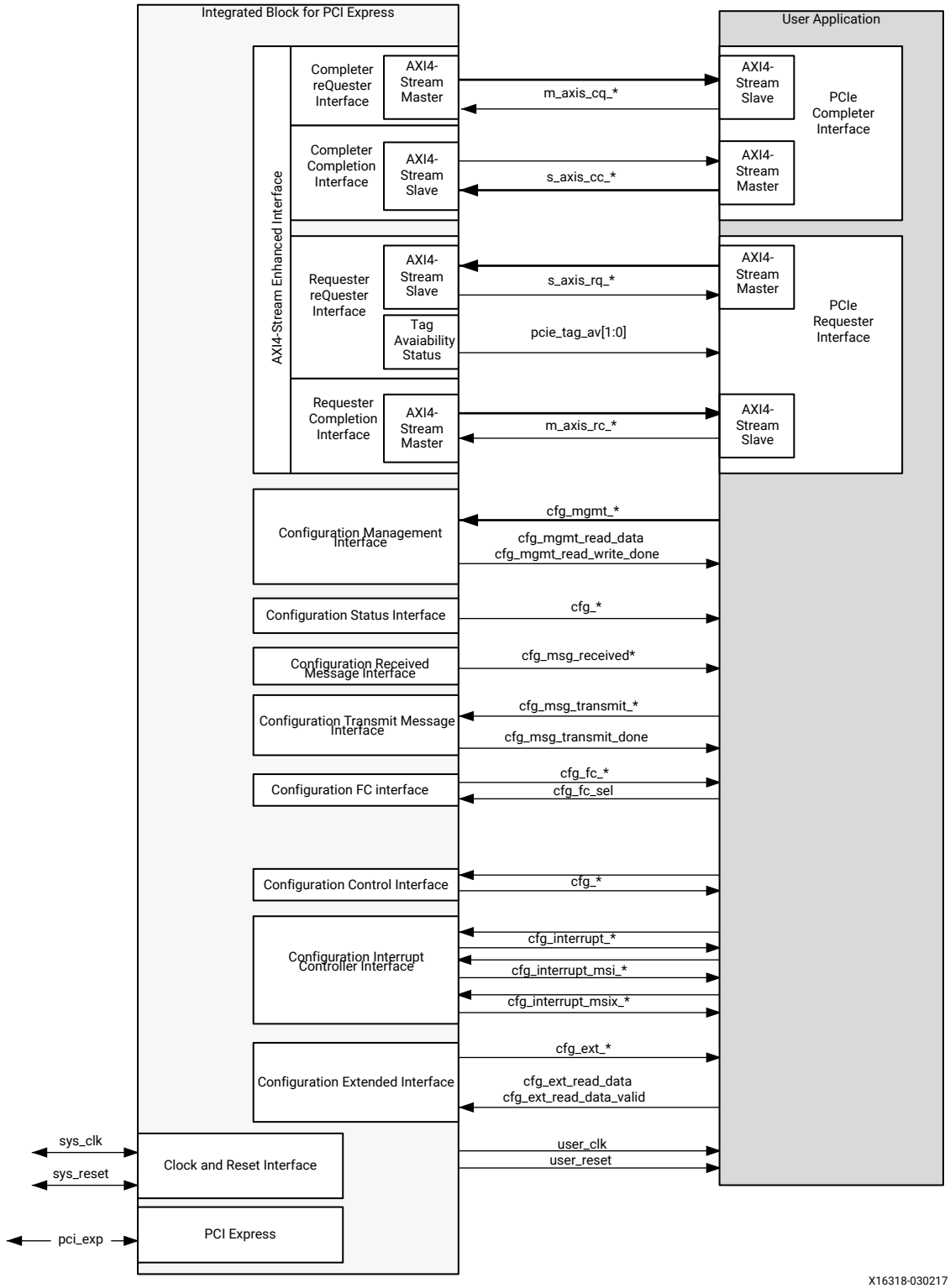
Overview

The UltraScale Devices Gen3 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 Integrated Block for PCI Express LogiCORE IP Product Guide (PG213).*

The following figure shows the interfaces for the core.

Figure 1: **Core Interfaces**



Feature Summary

The GTH transceivers in the Integrated Block for PCI® Express (PCIe®) solution support 1-lane, 2-lane, 4-lane, and 8-lane operation, running at 2.5 GT/s (Gen1), 5.0 GT/s (Gen2), and 8.0 GT/s (Gen3) line speeds. Endpoint 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:

- High-performance, highly flexible, scalable, and reliable general-purpose I/O core
  - Compliant with the PCI Express Base Specification, rev. 3.0
  - Compatible with conventional PCI software model
- GTH transceivers
  - 2.5 GT/s, 5.0 GT/s, and 8.0 GT/s line speeds
  - 1-lane, 2-lane, 4-lane, and 8-lane operation
- Endpoint or Root Port configuration
- Multiple Functions and Single Root I/O Virtualization (SR-IOV) in the Endpoint configuration
  - Up to two physical functions
  - Up to six virtual functions
- Standardized user interface(s)
  - Compliant to AXI4-Stream
  - Separate Requester, Completion, and Message interfaces
  - Flexible Data Alignment
  - Parity generation and checking on AXI4-Stream interfaces
  - Easy-to-use packet-based protocol
  - Full-duplex communication enabling
  - Optional back-to-back transactions to enable greater link bandwidth utilization

    - Support for flow control of data and discontinuation of an in-process transaction in transmit direction
    - Support for flow control of data in receive direction
- Compliant with PCI and PCI Express power management functions
- Optional Tag Management feature
- Maximum transaction payload of up to 1024 bytes
- End-to-End Cyclic Redundancy Check (ECRC)
- Advanced Error Reporting (AER)
- Multi-Vector MSI for up to 32 vectors and MSI-X
- Atomic operations and Transaction Layer Packets (TLP) processing hints

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
- No Write (NW) flag in Translation Request

- Does not work with GTY Transceivers

Licensing and Ordering

This AMD LogiCORE™ IP module is provided at no additional cost with the AMD Vivado™ Design Suite under the terms of the End User License.

For more information about this core, visit the <short core name> product web page.

Information about other AMD LogiCORE™ IP modules is available at the Intellectual Property page. For information about pricing and availability of other AMD LogiCORE IP modules and tools, contact your local sales representative.

Chapter 3

Product Specification

Resource Utilization

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

Minimum Device Requirements

The minimum device requirements for the core are as follows:

Table 1: **Minimum Device Requirements**

Capability Link Speed	Capability Link Widths	Supported Speed Grades
Gen1	x1, x2, x4, x8	-3,-2,-1,-1L, -1LV, -1H, and -1HV
Gen2	x1, x2, x4, x8	-3,-2,-1,-1L, -1LV, -1H and -1HV
Gen3	x1, x2, x4	-3,-2,-1,-1L, -1LV, -1H and -1HV[1]
Gen3	x8	-3, -2, -1, -1H and -1HV

Notes:

1. The Core Clock Frequency option must be set to 250 MHz for -1LV and -1L speed grades. The Core Clock Frequency option set to 500 MHz is supported for -3 and -2 speed grades only.
2. Gen3x8 is possible for -1, -1H and -1HV speed grades, depending on user design, but might require additional timing closure efforts. Gen3x8 is not supported for -1L and -1LV (0.9 V and 0.95 V) speed grade.
3. Engineering Samples (ES) might have additional restrictions. For more information, see the corresponding errata documents.
4. Speed grades -1L, -1LV are supported only for Kintex UltraScale devices.
   Speed grades -1H and -1HV are supported only for Virtex UltraScale devices.

Available Integrated Blocks for PCI Express

The following table lists the integrated blocks for PCI Express available for use in FPGAs containing multiple integrated blocks. In some cases, not all integrated blocks can be used due to lack of bonded GTH transceiver sites adjacent to the integrated block.

Table 2: **Available Integrated Blocks for PCI Express**

Device Selection		PCI Express Block Location					
Device	**Package**	**X0Y0**	**X0Y1**	**X0Y2**	**X0Y3**	**X0Y4**	**X0Y5**
XCKU025	FFVA1156	Yes					
XCKU035	FBVA676	Yes	Yes	Yes[2]			
	FBVA900	Yes	Yes	Yes[2]			
	FFVA1156	Yes	Yes	Yes[2]			
	SFVA784						
XCKU040	FBVA676	Yes	Yes	Yes[2]			
	FBVA900	Yes	Yes	Yes[2]			
	FFVA1156	Yes	Yes	Yes			
	SFVA784						
XQKU040	RBA676	Yes	Yes	Yes[2]			
	RFA1156	Yes	Yes	Yes			
XCKU060	FFVA1156	Yes	Yes	Yes			
	FFVA1517	Yes	Yes	Yes			
XQKU060	RFA1156	Yes	Yes	Yes			
XCKU085	FLVA1517	Yes	Yes	Yes	Yes	Yes	
	FLVB1760	Yes	Yes	Yes	Yes	Yes	
	FLVF1924	Yes	Yes	Yes	Yes	Yes	
XCKU095	FFVB1760	Yes	Yes	Yes	Yes		
	FFVB2104	Yes	Yes	Yes	Yes		
	FFVA1156	Yes	Yes	Yes			
	FFVC1517	Yes	Yes	Yes	Yes[2]		
XQKU095	RFA1156	Yes	Yes	Yes			
XCKU115	FLVA1517	Yes	Yes	Yes	Yes	Yes	Yes[2]
	FLVB1760	Yes	Yes	Yes	Yes	Yes	Yes
	FLVF1924	Yes	Yes	Yes	Yes	Yes	Yes
	FLVD1924	Yes	Yes	Yes[2]	Yes	Yes	Yes
	FLVD1517	Yes	Yes	Yes	Yes	Yes	Yes
	FLVA2104	Yes	Yes	Yes	Yes	Yes	Yes
	FLVB2104	Yes	Yes	Yes	Yes	Yes	Yes
XQKU115	RLD1517	Yes	Yes	Yes	Yes	Yes	Yes
	RLF1924	Yes	Yes	Yes	Yes	Yes	Yes
XCVU065	FFVC1517	Yes	Yes				
XCVU080	FFVA2104	Yes	Yes	Yes	Yes		
	FFVB2104	Yes	Yes	Yes	Yes		
	FFVB1760	Yes	Yes	Yes	Yes		
	FFVC1517	Yes	Yes	Yes	Yes[2]		
	FFVD1517	Yes	Yes	Yes	Yes		

Table 2: **Available Integrated Blocks for PCI Express** *(cont'd)*

Device Selection		PCI Express Block Location					
Device	**Package**	**X0Y0**	**X0Y1**	**X0Y2**	**X0Y3**	**X0Y4**	**X0Y5**
XCVU095	FFVA2104	Yes	Yes	Yes	Yes		
	FFVB2104	Yes	Yes	Yes	Yes		
	FFVB1760	Yes	Yes	Yes	Yes		
	FFVC2104	Yes	Yes	Yes	Yes		
	FFVC1517	Yes	Yes	Yes	Yes[2]		
	FFVD1517	Yes	Yes	Yes	Yes		
XCVU125	FLVB1760	Yes	Yes	Yes	Yes		
	FLVB2104	Yes	Yes	Yes	Yes		
	FLVC2104	Yes	Yes	Yes	Yes		
	FLVA2104	Yes	Yes	Yes	Yes		
	FLVD1517	Yes	Yes	Yes	Yes		
XCVU160	FLGB2104		Yes	Yes	Yes	Yes	
	FLGC2104	Yes	Yes	Yes	Yes	Yes	
XCVU190	FLGB2104			Yes	Yes	Yes	Yes
	FLGC2104	Yes	Yes	Yes	Yes	Yes	Yes
	FLGA2577	Yes	Yes	Yes	Yes	Yes	Yes
XCVU440	FLGA2892	Yes	Yes	Yes	Yes	Yes	Yes
	FLGB2377	Yes	Yes	Yes	Yes	Yes	Yes

Notes:

1. The software supports only two PCIe blocks for XCKU035 devices (for non SFVA784 packages), four PCIe blocks for XCKU085 devices, and four PCIe blocks for XCVU160/FLGC2104 devices.
2. Available only when the device/package migration option Enable GT Quad selection is selected. See Package Migration of UltraScale Devices PCI Express Designs.

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 AMD Vivado™ Design Suite provides an XDC for the selected part and package that matches the contents of the tables.

- Virtex UltraScale Devices Available GT Quads
- Kintex UltraScale Devices Available GT Quads

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. The following table defines the ports in the CQ interface of the core. In the Width column, DW denotes the configured data bus width (64, 128, or 256 bits).

Table 3: **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	85	CQ User Data. This set of signals contains sideband information for the transaction layer packets (TLP) being transferred. These signals are valid when m_axis_cq_tvalid is High. The following table 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.

Table 3: **Completer Request Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
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.
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.

Table 4: **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.

Table 4: **Sideband Signal Descriptions in m_axis_cq_tuser** *(cont'd)*

Bit Index	Name	Width	Description
7:4	ast_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. For Memory Reads and Writes of one DW transfers and zero length transfers, these bits should be 0s. 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 4: **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.

Completer Completion Interface

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

Table 5: **Completer Completion Interface Port Descriptions**

Port	Direction	Width	Description
s_axis_cc_tdata	Input	DW	Completer Completion Data bus. Completion data from the user application to the core. Only the lower 128 bits are used when the interface width is 128 bits, and only the lower 64 bits are used when the interface width is 64 bits.
s_axis_cc_tuser	Input	33	Completer Completion User Data. This set of signals contain sideband information for the TLP being transferred. These signals are valid when s_axis_cc_tvalid is High. The following table 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 5: **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 6: **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 6: **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. The following table defines the ports in the RQ interface of the core. In the Width column, DW denotes the configured data bus width (64, 128, or 256 bits).

Table 7: **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	60	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. The following table 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 7: **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_tkeep must be set to all 1s in all beats of a packet, except in the final beat when the total size of the packet is not a multiple of the width of the data bus (in both Dwords). This is true for both Dword-aligned and address-aligned modes of payload transfer. 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_num	Output	4	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_num[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_num[3:0] output is valid when pcie_rq_seq_num_vld is High.
pcie_rq_seq_num_vld	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_num[3:0].

Table 7: **Requester Request Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
pcie_rq_tag	Output	6	Requester reQuest Non-Posted tag. When tag management for Non-Posted requests is performed by the core (AXISTEN_IF_ENABLE_CLIENT_TAG is 0), this output is used by the core to communicate the allocated tag for each Non-Posted request received. The tag value on this bus is valid for one cycle when `pcie_rq_tag_vld` 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_vld` 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_vld	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_tag output.

Table 8: **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 Reads and Writes of one DW transfers and zero length transfers, these bits should be 0s. For Memory Writes of two Dwords or more, these bits indicate the valid bytes in the last Dword of the payload. 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 8: **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 8: **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.

Requester Completion Interface

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

Table 9: **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. The following table 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.

Table 9: **Requester Completion Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
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 10: **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.

Table 10: **Sideband Signal Descriptions in m_axis_rc_tuser** *(cont'd)*

Bit Index	Name	Width	Description
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.
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).

Table 10: **Sideband Signal Descriptions in m_axis_rc_tuser** *(cont'd)*

Bit Index	Name	Width	Description
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.

Other Core Interfaces

The core also provides the interfaces described in this section.

Configuration Management Interface

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

Table 11: **Configuration Management Interface Port Descriptions**

Port	Direction	Width	Description
cfg_mgmt_addr	Input	19	Read/Write Address Address is in the Configuration and Management register space, and is Dword aligned. For accesses from the local management bus: for the address bits cfg_mgmt_addr[17:10], select the PCI function associated with the configuration register; and for the bits cfg_mgmt_addr[9:0], select the register within the function. The address bit cfg_mgmt_addr[18] must be set to zero (0) when accessing the PCI or PCI Express configuration registers, and to one (1) when accessing the local management registers.
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.

Table 11: **Configuration Management Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
cfg_mgmt_read_write_done	Output	1	Read/Write operation complete Asserted for 1 cycle when operation is complete. Active-High.
cfg_mgmt_type1_cfg_reg_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. The following table defines the ports in the Configuration Status interface of the core.

Table 12: **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, 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

Table 12: **Configuration Status Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
cfg_negotiated_width	Output	4	Configuration Link Status. Negotiated Link Width: PCI Express Link Status register, Negotiated Link Width field. This field 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: • 0001b = x1 • 0010b = x2 • 0100b = x4 • 1000b = x8 Other values are reserved.
cfg_current_speed	Output	3	Current Link Speed This signal outputs the current link speed from Link Status register bits 1 down to 0. This field indicates the negotiated Link speed of the given PCI Express Link. • 001b: 2.5 GT/s PCI Express Link • 010b: 5.0 GT/s PCI Express Link • 100b: 8.0 GT/s PCI Express Link Other values are reserved
cfg_max_payload	Output	3	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. 000b: 128 bytes maximum payload size 001b: 256 bytes maximum payload size 010b: 512 bytes maximum payload size 011b: 1024 bytes maximum payload size Other values are reserved.
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 Other values are reserved

Table 12: **Configuration Status Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
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 Bits [15:8] are reserved
cfg_vf_status	Output	16	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 Bits [15:12] are reserved.
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. Bits [11:6] are reserved. The possible power states are: 000: D0_uninitialized 001: D0_active 010: D1 100: D3_hot Other values are reserved.
cfg_vf_power_state	Output	24	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. Bits [23:18] are reserved. The possible power states are: 000: D0_uninitialized 001: D0_active 010: D1 100: D3_hot Other values are reserved.
cfg_link_power_state	Output	2	Current power state of the PCI Express link. 00: L0 01: L0s 10: L1 11: L2 (Note: L2 state is not supported)

Table 12: **Configuration Status Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
cfg_err_cor_out	Output	1	Correctable Error Detected In Endpoint mode, the core activates this output for one cycle when it has detected a correctable error and its reporting is not masked. In a multi-function Endpoint, this is the logical OR of the correctable error status bits in the Device Status registers of all functions. In Root Port mode, this output is activated on detection of a local correctable error, when its reporting is not masked. This output does not respond to any errors signaled by remote devices using PCI Express error messages. These error messages are delivered through the message interface. ***Note***: This signal is not reliable when the user `clk` and core `clk` are different.
cfg_err_nonfatal_out	Output	1	Non-Fatal Error Detected In Endpoint mode, the core activates this output for one cycle when it has detected a non-fatal error and its reporting is not masked. In a multi-function Endpoint, this is the logical OR of the non-fatal error status bits in the Device Status registers of all functions. In Root Port mode, this output is activated on detection of a local non-fatal error, when its reporting is not masked. This output does not respond to any errors signaled by remote devices using PCI Express error messages. These error messages are delivered through the message interface. ***Note***: This status output might not be reliable for all Link/Speed configurations.
cfg_err_fatal_out	Output	1	Fatal Error Detected In Endpoint mode, the core activates this output for one cycle when it has detected a fatal error and its reporting is not masked. In a multi-function Endpoint, this is the logical OR of the fatal error status bits in the Device Status registers of all functions. In Root Port mode, this output is activated on detection of a local fatal error, when its reporting is not masked. This output does not respond to any errors signaled by remote devices using PCI Express error messages. These error messages are delivered through the message interface. ***Note***: This status output might not be reliable for all Link/Speed configurations.

Table 12: **Configuration Status Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
cfg_ltr_enable	Output	1	Latency Tolerance Reporting Enable. The state of this output reflects the setting of the LTR Mechanism Enable bit in the Device Control 2 register of physical function 0. When the core is configured as an Endpoint, the logic uses this output to enable the generation of LTR messages. This output is not to be used when the core is configured as a Root Port.
pcie_rq_tag_av	Output	2	Transmit flow control tag available, 1 if 1 or more tags are available
pcie_tfc_nph_av	Output	2	Transmit flow control non posted header credits available 00 - 0 or less credits available 01 - 1 credit available 10 - 2 credits available 11 - 3 or more credits available

Table 12: **Configuration Status Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
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: Rx_L0s.Entry 12: Rx_L0s.Idle 13: Rx_L0s.FTS 14: Tx_L0s.Entry 15: Tx_L0s.Idle 16: Tx_L0s.FTS 19: L2.Idle 1A: L2.TransmitWake 20: Disabled 21: Loopback_Entry_Master 22: Loopback_Active_Master 23: Loopback_Exit_Master 24: Loopback_Entry_Slave 25: Loopback_Active_Slave 26: Loopback_Exit_Slave 27: Hot_Reset 28: Recovery_Equalization_Phase0 29: Recovery_Equalization_Phase1 2a: Recovery_Equalization_Phase2 2b: Recovery_Equalization_Phase3
cfg_rcb_status	Output	4	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.

Table 12: **Configuration Status Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
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.
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. Bits [3:2] are reserved.
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. Bits [11:6] are reserved.
cfg_vf_tph_requester_enable	Output	8	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. Bits [7:6] are reserved.
cfg_vf_tph_st_mode	Output	24	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. Bits [23:18] are reserved.

Table 12: **Configuration Status Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
pcie_cq_np_req	Input	1	CQ Non-Posted Request. 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 incremented in each clock cycle when pcie_cq_np_req is High, and decremented on the delivery of each Non-Posted request across the interface. The core temporarily stops delivering Non-Posted requests to the user application 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 provide a one-cycle pulse on pcie_cq_np_req each time it is ready to receive a Non-Posted request, or can keep it High permanently if it does not need to exercise selective back pressure on Non-Posted requests. The assertion and deassertion of the pcie_cq_np_req signal does not need to be aligned with the packet transfers on the completer request interface. There is a minimum of five user_clk from the presentation of completion on m_axis_rc_tuser and the reuse of the tag that was returned on the completion.
pcie_cq_np_req_count	Output	6	CQ Non-Posted Request Count. This output provides the current value of the credit count maintained by the core for delivery of Non-Posted requests to the user application. 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 core receiving a pulse on the pcie_cq_np_req input and updating the pcie_cq_np_req_count output in response. This count is reset on user_reset and de-assertion of user_lnk_up.
pcie_tfc_nph_av	Output	2	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: 00: No credits available 01: 1 credit available 10: 2 credits available 11: 3 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 two 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 12: **Configuration Status Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
pcie_tfc_npd_av	Output	2	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: 00: No credits available 01: 1 credit available 10: 2 credits available 11: 3 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 two 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.

Configuration Received Message Interface

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

Table 13: **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 2 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 2.
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 1.

Configuration Transmit Message Interface

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

Table 14: **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

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

Table 15: **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 15: **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: Receive credit limit 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.

Per Function Status Interface

The Function Status interface provides status data as requested by the user application through the selected function. The following tables define the ports in the Function Status interface of the core.

Table 16: **Overview of Function Status Interface Port Descriptions**

Port	Direction	Width	Description
cfg_per_func_status_control	Input	3	Configuration Per Function Control. Controls information presented on the multi-function output cfg_per_func_status_data. Supported encodings are `000b`, `001b`, `010b`, `011b`, `100b`, and `101b`. All other encodings are reserved.
cfg_per_func_status_data	Output	16	Configuration Per Function Status Data. Provides a 16-bit status output for the selected function. Information presented depends on the values of cfg_per_func_status_control and cfg_per_function_number.

Table 17: **Detailed Function Status Interface Port Descriptions**

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
0	0	cfg_command_io_enable	1	Configuration Command – I/O Space Enable: Command[0]. Endpoints: If 1, allows the device to receive I/O Space accesses. Otherwise, the core filters these and respond with an Unsupported Request. Root/Switch: Core takes no action based on this setting. If 0, logic must not generate TLPs downstream.
0	1	cfg_command_mem_enable	1	Configuration Command – Memory Space Enable: Command[1]. Endpoints: If 1, allows the device to receive Memory Space accesses. Otherwise, the core filters respond with an Unsupported Request. Root/Switch: Core takes no action based on this setting. If 0, logic must not generate TLPs downstream.
0	2	cfg_command_bus_master_enable	1	Configuration Command – Bus Master Enable: Command[2]. The core takes no action based on this setting; logic must do that. Endpoints: When asserted, the logic is allowed to issue Memory or I/O Requests (including MSI/X interrupts); otherwise it must not. Root and Switch Ports: When asserted, received Memory or I/O Requests might be forwarded upstream; otherwise they are handled as Unsupported Requests (UR), and for Non-Posted Requests a Completion with UR completion status is returned.
0	3	cfg_command_interrupt_disable	1	Configuration Command – Interrupt Disable: Command[10]. When asserted, the core is prevented from asserting INTx interrupts.
0	4	cfg_command_serr_en	1	Configuration Command – SERR Enable: Command[8]. When asserted, this bit enables reporting of Non-fatal and Fatal errors. Errors are reported if enabled either through this bit or through the PCI Express specific bits in the Device Control register. In addition, for a Root Complex or Switch, this bit controls transmission by the primary interface of ERR_NONFATAL and ERR_FATAL error messages forwarded from the secondary interface.

Table 17: **Detailed Function Status Interface Port Descriptions** *(cont'd)*

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
0	5	cfg_bridge_ser_en	1	Configuration Bridge Control – SERR Enable: Bridge Ctrl[1]. When asserted, this bit enables the forwarding of Correctable, Non-fatal and Fatal errors (you must enforce that).
0	6	cfg_aer_ecrc_check_ en	1	Configuration AER – ECRC Check Enable: AER_Cap_and_Ctl[8]. When asserted, this bit indicates that ECRC checking has been enabled by the host.
0	7	cfg_aer_ecrc_gen_ en	1	Configuration AER – ECRC Generation Enable: AER_Cap_and_Ctl[6]. When asserted, this bit indicates that ECRC generation has been enabled by the host.
0	15:8	0	8	Reserved
1	0	cfg_dev_status_ corr_err_detected	1	Configuration Device Status – Correctable Error Detected: Device_Status[0]. Indicates status of correctable errors detected. Errors are logged in this register regardless of whether error reporting is enabled or not in the Device Control register.
1	1	cfg_dev_status_ non_fatal_err_detected	1	Configuration Device Status – Non-Fatal Error Detected: Device_Status[1]. Indicates status of Nonfatal errors detected. Errors are logged in this register regardless of whether error reporting is enabled or not in the Device Control register.
1	2	cfg_dev_status_ fatal_err_detected	1	Configuration Device Status – Fatal Error Detected: Device_Status[2]. Indicates status of Fatal errors detected. Errors are logged in this register regardless of whether error reporting is enabled or not in the Device Control register.
1	3	cfg_dev_status_ur_ detected	1	Configuration Device Status – Unsupported Request Detected: Device_Status[3]. Indicates that the core received an Unsupported Request. Errors are logged in this register regardless of whether error reporting is enabled or not in the Device Control register.

Table 17: **Detailed Function Status Interface Port Descriptions** *(cont'd)*

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
1	4	cfg_dev_control_corr_err_reporting_ en	1	Configuration Device Control – Correctable Error Reporting Enable: Device_Ctrl[0]. This bit, with other bits, controls sending ERR_COR Messages. For a Root Port, the reporting of correctable errors is internal to the root; no external ERR_COR Message is generated.
1	5	cfg_dev_control_ non_fatal_reporting_en	1	Configuration Device Control – Non-Fatal Error Reporting Enable: Device_Ctrl[1]. This bit, with other bits, controls sending ERR_NONFATAL Messages. For a Root Port, the reporting of correctable errors is internal to the root; no external ERR_NONFATAL Message is generated.
1	6	cfg_dev_control_fatal_err_reporting_ en	1	Configuration Device Control – Fatal Error Reporting Enable: Device_Ctrl[2]. This bit, with other bits, controls sending ERR_FATAL Messages. For a Root Port, the reporting of correctable errors is internal to the root; no external ERR_FATAL Message is generated.
1	7	cfg_dev_control_ur_err_reporting_en	1	Configuration Device Control – UR Reporting Enable: Device_Ctrl[3]. This bit, with other bits, controls the signaling of Unsupported Requests by sending Error Messages.
1	10:8	cfg_dev_control_max_payload	3	Configuration Device Control – Max_Payload_Size: Device_Ctrl[7:5]. This field sets 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. • 000b = 128 bytes max payload size • 001b = 256 bytes max payload size • 010b = 512 bytes max payload size • 011b = 1024 bytes max payload size
1	11	cfg_dev_control_ enable_ro	1	Configuration Device Control – Enable Relaxed Ordering: Device_Ctrl[4]. When asserted, the logic is permitted to set the Relaxed Ordering bit in the Attributes field of transactions it initiates that do not require strong write ordering.

Table 17: **Detailed Function Status Interface Port Descriptions** *(cont'd)*

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
1	12	cfg_dev_control_ ext_tag_en	1	Configuration Device Control – Tag Field Enable: Device_Ctrl[8]. When asserted, enables the logic to use an 8-bit Tag field as a Requester. If deasserted, the logic is restricted to a 5-bit Tag field. Note that the core does not enforce the number of Tag bits used, either in outgoing request TLPs or incoming Completions.
1	13	cfg_dev_control_ no_snoop_en	1	Configuration Device Control – Enable No Snoop: Device_Ctrl[11]. When asserted, the logic is permitted to set the No Snoop bit in TLPs it initiates that do not require hardware enforced cache coherency.
1	15:14	0	2	Reserved
2	2:0	cfg_dev_control_ max_read_req	3	Configuration Device Control – Max_Read_Request_Size: Device_Ctrl[14:12]. 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
2	3	cfg_link_status_ link_training	1	Configuration Link Status – Link Training: Link_Status[11]. Indicates that the Physical Layer LTSSM is in the Configuration or Recovery state, or that 1b was written to the Retrain Link bit but Link training has not yet begun. The core clears this bit when the LTSSM exits the Configuration/Recovery state.
2	6:4	cfg_link_status_ current_speed	3	Configuration Link Status – Current Link Speed: Link_Status[1:0]. This field indicates the negotiated Link speed of the given PCI Express Link. • 001b = 2.5 GT/s PCI Express Link • 010b = 5.0 GT/s PCI Express Link • 100b = 8.0 GT/s PCI Express Link

Table 17: **Detailed Function Status Interface Port Descriptions** *(cont'd)*

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
2	10:7	cfg_link_status_negotiated_width	4	Configuration Link Status – Negotiated Link Width: Link_Status[7:4]. This field indicates the negotiated width of the given PCI Express Link (only widths up to x8 displayed). • 0001b = x1 • 0010b = x2 • 0100b = x4 • 1000b = x8
2	11	cfg_link_status_bandwidth_status	1	Configuration Link Status – Link Bandwidth Management Status: Link_Status[14]. Indicates that either of the following has occurred without the Port transitioning through DL_Down status: • A Link retraining has completed following a write of 1b to the Retrain Link bit. • This bit is set following any write of 1b to the Retrain Link bit, including when the Link is in the process of retraining for some other reason. • Hardware has changed Link speed or width to attempt to correct unreliable Link operation, either through an LTSSM timeout or a higher level process. This bit is set if the Physical Layer reports a speed or width change was initiated by the Downstream component that was not indicated as an autonomous change.
2	12	cfg_link_status_auto_bandwidth_ status	1	Configuration Link Status – Link Autonomous Bandwidth Status: Link_Status[15]. Indicates the core has autonomously changed Link speed or width, without the Port transitioning through DL_Down status, for reasons other than to attempt to correct unreliable Link operation. This bit must be set if the Physical Layer reports a speed or width change was initiated by the Downstream component that was indicated as an autonomous change.
2	15:13	0	3	Reserved

Table 17: **Detailed Function Status Interface Port Descriptions** *(cont'd)*

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
3	1:0	cfg_link_control_aspm_control	2	Configuration Link Control – ASPM Control: Link_Ctrl[1:0]. Indicates the level of ASPM supported, where: • 00b = Disabled • 01b = L0s Entry Enabled • 10b = L1 Entry Enabled • 11b = L0s and L1 Entry Enabled
3	2	cfg_link_control_ rcb	1	Configuration Link Control – RCB: Link_Ctrl[3]. Indicates the Read Completion Boundary value, where, • 0=64B • 1=128B
3	3	cfg_link_control_link_disable	1	Configuration Link Control – Link Disable: Link_Ctrl[4]. When asserted, indicates the Link is disabled and directs the LTSSM to the Disabled state.
3	4	cfg_link_control_ core_clock	1	Configuration Link Control – Common Clock Configuration: Link_Ctrl[6]. When asserted, indicates that this component and the component at the opposite end of this Link are operating with a distributed common reference clock. When deasserted, indicates they are operating with an asynchronous reference clock.
3	5	cfg_link_control_extendedness	1	Configuration Link Control – Extended Sync: Link_Ctrl[7]. When asserted, forces the transmission of additional Ordered Sets when exiting the L0s state and when in the Recovery state.
3	6	cfg_link_control_clock_pm_en	1	Configuration Link Control – Enable Clock Power Management: Link_Ctrl[8]. For Upstream Ports that support a CLKREQ# mechanism, indicates: • 0b = Clock power management disabled. • 1b = The device is permitted to use CLKREQ#. The core takes no action based on the setting of this bit; external logic must implement this.

Table 17: **Detailed Function Status Interface Port Descriptions** *(cont'd)*

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
3	7	cfg_link_control_hw_auto_width_dis	1	Configuration Link Control – Hardware Autonomous Width Disable: Link_Ctrl[9]. When asserted, disables the core from changing the Link width for reasons other than attempting to correct unreliable Link operation by reducing Link width.
3	8	cfg_link_control_bandwidth_int_en	1	Configuration Link Control – Link Bandwidth Management Interrupt Enable: Link_Ctrl[10]. When asserted, enables the generation of an interrupt to indicate that the Link Bandwidth Management Status bit has been set. The core takes no action based on the setting of this bit; the logic must create the interrupt.
3	9	cfg_link_control_auto_bandwidth_ int_en	1	Configuration Link Control – Link Autonomous Bandwidth Interrupt Enable: Link_Ctrl[11]. When asserted, this bit enables the generation of an interrupt to indicate that the Link Autonomous Bandwidth Status bit has been set. The core takes no action based on the setting of this bit; the logic must create the interrupt.
3	10	cfg_tph_requester_ enable	1	TPH Requester Enable: Bit [8] of the TPH Requester Control register in the TPH Requester Capability Structure of the function. These bits are active only in the Endpoint mode. Indicates whether the software has enabled the device to generate requests with TPH Hints from the associated function.
3	13:11	cfg_tph_steering_ tag_mode	3	TPH Steering Tag Mode. Reflect the setting of the ST Mode Select bits in the TPH Requester Control register. These bits are active only in the Endpoint mode. They indicate the allowed modes for generation of TPH Hints by the corresponding function.
3	15:14	0	2	Reserved

Table 17: **Detailed Function Status Interface Port Descriptions** *(cont'd)*

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
4	3:0	cfg_dev_control2_cpl_timeout_val	4	Configuration Device Control 2 – Completion Timeout Value: Device_Ctrl2[3:0]. This is the time range that the logic regard as a Request is pending Completion as a Completion Timeout. The core takes no action based on this setting. • 0000b = 50 µs to 50 ms (default) • 0001b = 50 µs to 100 µs • 0010b = 1 ms to 10 ms • 0101b = 16 ms to 55 ms • 0110b = 65 ms to 210 ms • 1001b = 260 ms to 900 ms • 1010b = 1 s to 3.5 s • 1101b = 4 s to 13 s • 1110b = 17 s to 64 s
4	4	cfg_dev_control2_cpl_timeout_dis	1	Configuration Device Control 2 – Completion Timeout Disable: Device_Ctrl2[4]. This disables the Completion Timeout counters.
4	5	cfg_dev_control2_atomic_requester_ en	1	Configuration Device Control 2 – Atomic Requester Enable: Device_Ctrl2[6]. Applicable only to Endpoints and Root Ports; must be hardwired to `0b` for other function types. The function is allowed to initiate AtomicOp Requests only if this bit and the Bus Master Enable bit in the Command register are both set. This bit is required to be RW if the Endpoint or Root Port is capable of initiating AtomicOp Requests, but otherwise is permitted to be hardwired to `0b`. This bit does not serve as a capability bit. This bit is permitted to be RW even if no AtomicOp Requester capabilities are supported by the Endpoint or Root Port. Default value of this bit is `0b`. 32 nm

Table 17: **Detailed Function Status Interface Port Descriptions** *(cont'd)*

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
4	6	cfg_dev_control2_ido_req_en	1	Configuration Device Control 2 – IDO Request Enable: Device_Ctrl2[8]. If this bit is set, the function is permitted to set the ID-Based Ordering (IDO) bit (Attribute[2]) of Requests it initiates (see Section 2.2.6.3 and Section 2.4). Endpoints, including RC Integrated Endpoints, and Root Ports are permitted to implement this capability. A function is permitted to hardwire this bit to 0b if it never sets the IDO attribute in Requests. Default value of this bit is 0b. 32 nm
4	7	cfg_dev_control2_ ido_cpl_en	1	Configuration Device Control 2 – IDO Completion Enable: Device_Ctrl2[9]. If this bit is set, the function is permitted to set the ID-Based Ordering (IDO) bit (Attribute[2]) of Completions it returns (see Section 2.2.6.3 and Section 2.4). Endpoints, including RC Integrated Endpoints, and Root Ports are permitted to implement this capability. A function is permitted to hardwire this bit to 0b if it never sets the IDO attribute in Requests. Default value of this bit is 0b. 32 nm
4	8	cfg_dev_control2_ ltr_en	1	Configuration Device Control 2 – LTR Mechanism Enable: Device_Ctrl2[10]. If this bit is set, the function is permitted to set the ID-Based Ordering (IDO) bit (Attribute[2]) of Completions it returns (see Section 2.2.6.3 and Section 2.4). Endpoints, including RC Integrated Endpoints, and Root Ports are permitted to implement this capability. A function is permitted to hardwire this bit to 0b if it never sets the IDO attribute in Requests. Default value of this bit is 0b. 32 nm
4	13:9	cfg_dpa_substate	5	Dynamic Power Allocation Substate: Reflect the setting of the Dynamic Power Allocation Substate field in the DPA Control register.
4	15:14	0	1	Reserved
5	0	cfg_root_control_syserr_corr_err_en	1	Configuration Root Control – System Error on Correctable Error Enable: Root_Control[0]. This bit enables the logic to generate a System Error for reported Correctable Errors.

Table 17: **Detailed Function Status Interface Port Descriptions** *(cont'd)*

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
5	1	cfg_root_control_syserr_non_fatal_err_en	1	Configuration Root Control – System Error on Non-Fatal Error Enable: Root_Control[1]. This bit enables the logic to generate a System Error for reported Non-Fatal Errors.
5	2	cfg_root_control_syserr_fatal_er_en	1	Configuration Root Control – System Error on Fatal Error Enable: Root_Control[2]. This bit enables the logic to generate a System Error for reported Fatal Errors.
5	3	cfg_root_control_ pme_int_en	1	Configuration Root Control – PME Interrupt Enable: Root_Control[3]. This bit enables the logic to generate an Interrupt for received PME Messages.
5	4	cfg_aer_rooter_corr_err_reporting_ en	1	Configuration AER – Correctable Error Reporting Enable: AER_Root_Error_Command[0]. This bit enables the logic to generate interrupts for reported Correctable Errors.
5	5	cfg_aer_rooter_non_fatal_err_reporting_en	1	Configuration AER – Non-Fatal Error Reporting Enable: AER_Root_Error_Command[1]. This bit enables the user logic to generate interrupts for reported Non-Fatal Errors.
5	6	cfg_aer_rooter_fatal_err_reporting_ en	1	Configuration AER – Fatal Error Reporting Enable: AER_Root_Error_Command[2]. This bit enables the user logic to generate interrupts for reported Fatal Errors.
5	7	cfg_aer_rooter_corr_err_received	1	Configuration AER – Correctable Error Messages Received: AER_Root_Error_Status[0]. Indicates that an ERR_COR Message was received.
5	8	cfg_aer_rooter_non_fatal_err_received	1	Configuration AER – Non-Fatal Error Messages Received: AER_Root_Error_Status[5]. Indicates that an ERR_NFE Message was received.
5	9	cfg_aer_rooter_fatal_err_received	1	Configuration AER – Fatal Error Messages Received: AER_Root_Error_Status[6]. Indicates that an ERR_FATAL Message was received.
5	15:10	0	6	Reserved

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.

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

Table 18: **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. ***Note*:** The input must be asserted until the LTSSM enters Hot_Reset state.
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_per_function_update_done	Output	1	Configuration per Function Update Complete Asserted in response to cfg_per_function_output_request assertion, for one cycle after the request is complete.

Table 18: **Configuration Control Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
cfg_per_function_number	Input	4	Configuration Per Function Target Function Number The user provides the function number (0-7), where value 0–1 corresponds to PF0–1, and value 2–7 corresponds to VF0–5, and asserts cfg_per_function_output_request to obtain per function output values for the selected function. All other values are reserved
cfg_per_function_output_request	Input	1	Configuration Per Function Output Request When this port is asserted with a function number value on cfg_per_function_number, the core presents information on per-function configuration output pins and asserts cfg_update_done when complete.
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). After the user logic updates cfg_dsn port, the new cfg_dsn port should appear on the Extended Configuration Space. No additional qualifying control signal is required.
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_port_number	Input	8	Configuration Downstream Port Number Provides the port number field in the Link Capabilities register.
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 18: **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_subsys_id	Input	16	Configuration Subsystem ID Indicates the value that should be transferred to the Type 0 PCI Capability Strucute Subsytem ID field on PF0.

Table 18: **Configuration Control Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
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 (AER) mechanism. In response, the core sets the Corrected Internal Error Status bit in the AER Correctable Error Status register of all enabled functions, and also sends an error message if enabled to do so. This error is not considered function-specific.
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. Bits [3:2] are reserved.
cfg_vf_flr_done	Input	8	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. Bits [7:6] 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. Bits [3:2] are reserved.

Table 18: **Configuration Control Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
cfg_vf_flr_in_process	Output	8	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 corresponding cfg_vf_flr_done input for the corresponding function to indicate the completion of the reset operation.
cfg_req_pm_transition_l23_ready	Input	1	When the core is configured as an Endpoint, the user application asserts this input to transition the power management state of the core to L23_READY (see Chapter 5 of the *PCI Express Specification* 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.

Table 18: **Configuration Control Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
cfg_local_error	Output	1	Local Error Conditions Output is logic High when any of the local error conditions listed in the Local Error Status Register occur. Each of the conditions can be selectively masked by setting the corresponding bits in the Local Interrupt Mask Register. Up to eight of the following back to back block internal events (if not masked) are reported. 1) tx_replay_buffer_ram_uncorrectable_parity_error 2) rx_request_fifo_ram_uncorrectable_parity_error 3) rx_completion_fifo_ram_uncorrectable_parity_error 4) rx_receive_fifo_overflow 5) rx_completion_receive_fifo_overflow 6) link_replay_timeout 7) link_replay_num_rollover 8) phy_error_detected 9) malformed_tlp_received_reg 10) unexpected_completion_received 11) flow_control_protocol_error_detected 12) completion_timeout ***Note*:** This signal might not work for all PCIe Link Width/Speed configurations. Do not rely solely on this signal to indicate an error.

Configuration Interrupt Controller Interface

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

Table 19: **Configuration Interrupt Controller Interface Port Descriptions**

Port	Direction	Width	Description
cfg_interrupt_int	Input	4	Configuration INTx Vector When the core is configured as EP, these four inputs are used by the user application to signal an interrupt from any of its PCI functions to the RC using the Legacy PCI Express Interrupt Delivery mechanism of PCI Express. These four inputs correspond to INTA, INTB, INTC, and INTD. 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.

Table 19: **Configuration Interrupt Controller Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
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.
cfg_interrupt_pending	Input	4	Configuration INTx Interrupt Pending (active-High) Per function indication of a pending interrupt. cfg_interrupt_pending[0] corresponds to physical function 0 and so on. Bits [3:2] are reserved.
cfg_interrupt_msi_enable	Output	4	Configuration Interrupt MSI Function Enabled Indicates that Message Signaling Interrupt (MSI) messaging is enabled per function. Bits [3:2] are reserved.
cfg_interrupt_msi_vf_enable	Output	8	Configuration Interrupt MSI on VF Enabled Indicates that MSI messaging is enabled, per virtual function. Bits [7:6] are reserved.
cfg_interrupt_msi_int	Input	32	Configuration Interrupt MSI Vector When the core is configured in 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 cfg_interrupt_msi_function_number input. 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_sent	Output	1	Configuration Interrupt MSI Interrupt Sent The core generates a one-cycle pulse on this output to signal that an MSI 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 Interrupt Operation Failed A one-cycle pulse on this output indicates that an MSI interrupt message was aborted before transmission on the link. The user application must retransmit the MSI interrupt in this case.
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 registers 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 and bits [5:4] to physical function 1. Bits [12:6] are reserved.

Table 19: **Configuration Interrupt Controller Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
cfg_interrupt_msi_pending_status	Input	32	Configuration Interrupt MSI Pending Status These inputs provide the status of the MSI pending interrupts for the physical functions. The setting of these pins determines the value read from the MSI Pending bits register of the corresponding PF. The MSI Pending bits register contains the pending bits for MSI Interrupts. A read from this location returns the state of MSI_MASK inputs of the core. This is a 32-bit wide RO register with a default value of MSI MASK inputs. Assert 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 Update Asserted for one cycle when any enabled functions in the MSI Mask register change value. MSI Mask register contains the Mask bits for MSI interrupts. The Multiple Message Capable field in the MSI Control register specifies the number of distinct interrupts for the function, which determines the number of valid mask bits. This is a 32-bit wide RW register with a default value of 0.
cfg_interrupt_msi_select	Input	4	Configuration Interrupt MSI Select Values 0000b-0001b correspond to PF0-1 selection, and values 0010b-0111b correspond to VF0-5 selection. cfg_interrupt_msi_data[31:0] presents the value of the MSI Mask register from the selected function. When this input is driven to 1111b, cfg_interrupt_msi_data[17:0] presents the Multiple Message Enable bits of the MSI Control registers associated with all virtual functions. These bits encode the number of allocated MSI interrupt vectors for the corresponding function. cfg_interrupt_msi_data[2:0] correspond to virtual function 0, and so on.
cfg_interrupt_msi_data	Output	32	Configuration Interrupt MSI Data The value presented depends on cfg_interrupt_msi_select.
cfg_interrupt_msi_pending_status_function_num	Input	4	Configuration Interrupt MSI Pending Target Function Number You provide the function number (0-11), where values 0-3 corresponds to PF0-3, and value 4-11 corresponds to VF0-7.
cfg_interrupt_msi_pending_status_data_enable	Input	1	Configuration Interrupt MSI Pending Data Valid Assert 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_msix_enable	Output	4	Configuration Interrupt MSI-X Function Enabled When asserted, indicates that the Message Signaling Interrupt (MSI-X) messaging is enabled, per function.
cfg_interrupt_msix_mask	Output	4	Configuration Interrupt MSI-X Function Mask Indicates the state of the Function Mask bit in the MSI-X Message Control field, per function.
cfg_interrupt_msix_vf_enable	Output	8	Configuration Interrupt MSI-X on VF Enabled When asserted, indicates that Message Signaling Interrupt (MSI-X) messaging is enabled, per virtual function.
cfg_interrupt_msix_vf_mask	Output	8	Configuration Interrupt MSI-X VF Mask Indicates the state of the Function Mask bit in the MSI-X Message Control field, per virtual function.

Table 19: **Configuration Interrupt Controller Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
cfg_interrupt_msix_address	Input	64	Configuration Interrupt MSI-X Address When the core is configured to support MSI-X interrupts, this bus is used by the user logic to communicate the address to be used for an MSI-X message.
cfg_interrupt_msix_data	Input	32	Configuration Interrupt MSI-X Data When the core is configured to support MSI-X interrupts, this bus is used by the user logic to communicate the data to be used for an MSI-X message.
cfg_interrupt_msix_int	Input	1	Configuration Interrupt MSI-X Data Valid This signal indicates that valid information has been placed on the cfg_interrupt_msix_address[63:0] and cfg_interrupt_msix_data[31:0] buses, and the originating function number has been placed on cfg_interrupt_msi_function_number[3:0]. The core internally registers the associated address and data from cfg_interrupt_msix_address and cfg_interrupt_msix_data on the 0-to-1 transition of this valid signal. The user application must ensure that the cfg_interrupt_msix_enable bit corresponding to function in use is set before asserting cfg_interrupt_msix_int. After asserting an interrupt, the user logic must wait for the cfg_interrupt_msix_sent or cfg_interrupt_msix_fail indication from the core before asserting a new interrupt.
cfg_interrupt_msix_sent	Output	1	Configuration Interrupt MSI-X Interrupt Sent The core generates a one-cycle pulse on this output to indicate that it has accepted the information placed on the cfg_interrupt_msix_address[63:0] and cfg_interrupt_msix_data[31:0] buses, and an MSI-X interrupt message has been transmitted on the link. The user application must wait for this pulse before signaling another interrupt condition to the core.
cfg_interrupt_msix_fail	Output	1	Configuration Interrupt MSI-X Interrupt Operation Failed A one-cycle pulse on this output indicates that the interrupt controller has failed to transmit MSI-X interrupt on the link. The user application must retransmit the MSI-X interrupt in this case.
cfg_interrupt_msi_attr	Input	3	Configuration Interrupt MSI/MSI-X TLP Attr These bits provide the setting of the Attribute bits to be used for the MSI/MSI-X interrupt request. 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 or cfg_interrupt_msix_int.
cfg_interrupt_msi_tph_present	Input	1	Configuration Interrupt MSI/MSI-X TPH Present Indicates the presence of a Transaction Processing Hint (TPH) in the MSI/MSI-X interrupt request. The user application must set this bit while asserting cfg_interrupt_msi_int or cfg_interrupt_msix_int, if it includes a TPH in the MSI or MSI-X transaction.
cfg_interrupt_msi_tph_type	Input	2	Configuration Interrupt MSI/MSI-X TPH Type When cfg_interrupt_msi_tph_present is `1'b1`, these two bits supply the two-bit type associated with the hint. The core samples these bits on a 0-to-1 transition on cfg_interrupt_msi_int or cfg_interrupt_msix_int.

Table 19: **Configuration Interrupt Controller Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
cfg_interrupt_msi_tph_st_tag	Input	9	Configuration Interrupt MSI/MSI-X TPH Steering Tag When cfg_interrupt_msi_tph_present is `1'b1`, the Steering Tag associated with the Hint must be placed on cfg_interrupt_msi_tph_st_tag[7:0]. Setting cfg_interrupt_msi_tph_st_tag[8] to 1b activates the Indirect Tag mode. In the Indirect Tag mode, the core uses bits [5:0] of cfg_interrupt_msi_tph_st_tag as an index into its Steering Tag Table (STT) in the TPH Capability Structure (STT is limited to 64 entries per function), and inserts the tag from this location in the transmitted request MSI/X TLP. Setting cfg_interrupt_msi_tph_st_tag[8] to `0b` activates the Direct Tag mode. In the Direct Tag mode, the core inserts cfg_interrupt_msi_tph_st_tag[7:0] directly as the Tag in the transmitted MSI/X TLP. The core samples these bits on a 0-to-1 transition on any cfg_interrupt_msi_int bits or cfg_interrupt_msix_int.
cfg_interrupt_msi_function_number	Input	4	Configuration MSI/MSI-X Initiating Function Indicates the Endpoint function number initiating the MSI or MSI-X transaction: • 0: PF0 • 1: PF1 • 2: Not used • 3: Not used • 4: VF0 • 5: VF1 • 6: VF2 • 7: VF3 • 8: VF4 • 9: VF5

Configuration Extend Interface

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

Table 20: **Configuration Extend Interface Port Descriptions**

Port	Direction	Width	Description
cfg_ext_read_received	Output	1	Configuration Extend Read Received The core asserts this output when it has received a configuration read request from the link. When neither user-implemented legacy or extended configuration space is enabled, receipt of a configuration read results in a one-cycle assertion of this signal, together with valid cfg_ext_register_number and cfg_ext_function_number. When user-implemented legacy, extended configuration space, or both are enabled, for the cfg_ext_register_number ranges, `0x10-0x1f` or `0x120-0x3ff`, respectively, this signal is asserted, until user logic presents cfg_ext_read_data and cfg_ext_read_data_valid. For cfg_ext_register_number ranges outside `0x10` - `0x1f` or `0x120` - `0x3ff`, receipt of a configuration read always results in a one-cycle assertion of this signal.
cfg_ext_write_received	Output	1	Configuration Extend Write Received The core generates a one-cycle pulse on this output when it has received a configuration write request from the link.
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.
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. ***Note***: The core will time-out the read request after 40000h (2^{18}) clock cycles of user_clk if cfg_ext_read_received signal is not asserted.

Clock and Reset Interface

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

***Note*:** The phy ready indication signal, `phy_rdy_out`, indicates that the GT Wizard is ready. This signal is driven by `phy_rst` FSM on receiving the phy status from the GT Wizard core.

Table 21: **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 AMD 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, 125 MHz, or 250 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, 125 MHz, or 250 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. You can reset to active-High using an option in the Vivado IDE, but this can result in incompatibility with the PCIe edge connector. Dedicated routing between the FPGA PERSTN0 package pin and the PCIe integrated block is enabled by default where available. Supported Devicesidentifies the PCIe sites and their corresponding dedicated sys_reset IOB location. No other PCIe integrated block locations have dedicated sys_reset connections. Use the dedicated routing and the associated IOB when possible. To use another sys_reset pin location, the *Use the dedicated PERST routing resources* parameter must be disabled in the Vivado IDE. In addition, use the PERSTN1 package pin for the sys_reset location of endpoint configurations for PCIe sites not listed in Supported Devices.
pcie_perstn0_out	Output	1	Output that is a direct pass-through from the PERSTN0 package pin through the sys_reset input port for the PCIe site listed in Supported Devices. This port is only available when dedicated routing (through the *Use the dedicated PERST routing resources* parameter) is enabled (default), and sys_reset is driven by the PERSTN0 package pin. For all other configurations and PCIe locations, this port should not be connected.
pcie_perstn1_in	Input	1	Input to a dedicated route from the PERSTN1 package pin to the pcie_perstn1_out output. This input can be driven only by the PERSTN1 package pin and should only be used when dedicated routing (through the *Use the dedicated PERST routing resources* parameter) is enabled (default). For PCIe locations that do not support dedicated reset routing, this port should be tied to a constant zero (1'b0).
pcie_perstn1_out	Output	1	Output that is a direct pass-through from the PERSTN1 package pin through the pcie_perstn1_in input port for the PCIe integrated blocks that support dedicated routing. This port can only be used when dedicated reset routing (through the *Use the dedicated PERST routing resources* parameter) is enabled, and pcie_perstn1_in is driven by the PERSTN1 package pin. For all other configurations, this port should not be connected. Optionally, the PERSTN1 package pin can be used to drive the sys_reset input port for PCIe endpoint configurations that do not support dedicate reset routing.

The PERSTN0 / PERSTN1 package pins and the reset input ports described in the preceding table are used for dedicated PCIe reset routing. These are dedicated ports from the PERSTN package pins to specific PCIe integrated block locations. Users who need Tandem Configuration support should use these pins as described in the preceding table. The general guidelines for using PERSTN0 and PERSTN1 pins are as follows:

- Root Port configurations can use any pin to drive the edge connector reset.
- Endpoint configurations should always use PERSTN0 as PCIe edge connector reset input pin if dedicated routing is available.
- Endpoint configurations should give priority to PERSTN1 as the PCIe edge connector reset input pin, if dedicated reset routing is not available, but can use any pin as desired.

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, and the 8-lane core supports lanes 0–7. Transmit and receive signals of the PCI_EXP interface are defined in the following table.

Table 22: **PCI Express Interface Signals for 1-, 2-, 4- and 8-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 (–)

Table 22: **PCI Express Interface Signals for 1-, 2-, 4- and 8-Lane Cores** *(cont'd)*

Lane Number	Name	Direction	Description
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 (–)
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 (–)

Table 22: **PCI Express Interface Signals for 1-, 2-, 4- and 8-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 (–)
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 (–)
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 (–)

Table 22: **PCI Express Interface Signals for 1-, 2-, 4- and 8-Lane Cores** *(cont'd)*

Lane Number	Name	Direction	Description
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_rxn3	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_rxp6	Input	PCI Express Receive Negative: Serial Differential Input 6 (–)
7	pci_exp_txp7	Output	PCI Express Transmit Positive: Serial Differential Output 7 (+)
	pci_exp_txn7	Output	PCI Express Transmit Negative: Serial Differential Output 7 (–)
	pci_exp_rxp7	Input	PCI Express Receive Positive: Serial Differential Input 7 (+)
	pci_exp_rxn7	Input	PCI Express Receive Negative: Serial Differential Input 7 (–)
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 (–)

Table 22: **PCI Express Interface Signals for 1-, 2-, 4- and 8-Lane Cores** *(cont'd)*

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

Attribute Descriptions

User Interface

The following table lists the configuration attributes controlling the operation of the user interface of the core.

Table 23: **Configuration Attributes of the Integrated Block User Interface**

Attribute Name	Type	Description
USER_CLK2_FREQ	Integer	• 2: 62.50 MHz (default) • 3: 125.00 MHz • 4: 250.00 MHz
PL_LINK_CAP_MAX_LINK_SPEED[2:0]	Bit vector	Defines the maximum speed of the PCIe link. • 001: 2.5 GT/s • 010: 5.0 GT/s • 100: 8.0 GT/s All other encodings are reserved.
PL_LINK_CAP_MAX_LINK_WIDTH[3:0]	Bit vector	Maximum Link Width. Valid settings are: • 0001b: x1 • 0010b: x2 • 0100b: x4 • 1000b: x8 All other encodings are reserved. This setting is propagated to all layers in the core.
C_DATA_WIDTH	Integer	Configures the width of the AXI4-Stream interfaces. • 64 bit interface • 128 bit interface • 256 bit interface
AXISTEN_IF_CQ_ALIGNMENT_MODE	String	Defines the data alignment mode for the completer request interface. • FALSE: Dword-aligned Mode • TRUE: Address-aligned Mode
AXISTEN_IF_CC_ALIGNMENT_MODE	String	Defines the data alignment mode for the completer completion interface. • FALSE: Dword-aligned Mode • TRUE: Address-aligned Mode
AXISTEN_IF_RQ_ALIGNMENT_MODE	String	Defines the data alignment mode for the requester request interface. • FALSE: Dword-aligned Mode • TRUE: Address-aligned Mode
AXISTEN_IF_RC_ALIGNMENT_MODE	String	Defines the data alignment mode for the requester completion interface. • FALSE: Dword-aligned Mode • TRUE: Address-aligned Mode

Table 23: **Configuration Attributes of the Integrated Block User Interface** *(cont'd)*

Attribute Name	Type	Description
AXISTEN_IF_RC_STRADDLE	String	This attribute enables the straddle option on the requester completion interface. • FALSE: Straddle option disabled • TRUE: Straddle option enabled
AXISTEN_IF_RQ_PARITY_CHECK	String	This attribute enables parity checking on the requester request interface. • FALSE: Parity check disabled • TRUE: Parity check enabled
AXISTEN_IF_CC_PARITY_CHECK	String	This attribute enables parity checking on the completer completion interface. • FALSE: Parity check disabled • TRUE: Parity check enabled
AXISTEN_IF_ENABLE_RX_MSG_INTFC	String	This attribute controls how the core delivers a message received from the link. When this attribute is set to FALSE, the core delivers the received message TLPs on the completer request interface using the AXI4-Stream protocol. In this mode, you can select the message types to receive using the AXISTEN_IF_ENABLE_MSG_ROUTE attributes. The receive message interface is inactive in this mode. When this attribute is set to TRUE, the core internally decodes messages received from the link, and signals them to the user by activating the cfg_msg_received signal on the receive message interface. The core does not transfer any message TLPs on the completer request interface. The settings of the AXISTEN_ENABLE_MSG_ROUTE attributes have no effect on the operation of the receive message interface in this mode.
AXISTEN_IF_ENABLE_MSG_ROUTE[17:0]	Bit vector	When the `AXISTEN_IF_ENABLE_RX_MSG_INTFC` attribute is set to 0, you can use these attributes to select the specific message types you want to receive on the completer request interface. Setting a 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. The following table defines the attribute bit definitions corresponding to the various message types.
AXISTEN_IF_ENABLE_CLIENT_TAG	String	When set to FALSE, tag management for Non-Posted transactions initiated from the requester request interface is performed by the integrated block. That is, for each Non-Posted request, the core allocates the tag for the transaction and communicates it to the user interface. Setting set to TRUE, disables the internal tag management, allowing the user logic to supply the tag to be used for each request. The user logic must present the Tag field in the Request descriptor header in the range 0–31 when the PF0_DEV_CAP_EXT_TAG_SUPPORTED attribute is FALSE, while the Tag field can be in the range 0–63 when the PF0_DEV_CAP_EXT_TAG_SUPPORTED attribute is TRUE.

Table 24: **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	Optimized Buffer Flush/Fill (OBFF)
14	Unlock
15	Vendor_Defined Type 0
16	Vendor_Defined Type 1
17	Invalid Request, Invalid Completion, Page Request, PRG Response

Configuration Space

The PCI configuration space consists of three primary parts, illustrated in the following table. These include:

- Legacy PCI v3.0 Type 0/1 Configuration Space Header
  - Type 0 Configuration Space Header used by Endpoint applications (see Table 26: Type 0 PCI Configuration Space Header)
  - Type 1 Configuration Space Header used by Root Port applications (see Table 27: Type 1 PCI Configuration Space Header)
- 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)
  - Power Budgeting Enhanced
  - Capability Header (PB) (optional)
  - Latency Tolerance Reporting (LTR) (optional)
  - Dynamic Power Allocation (DPA) (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 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 `480h`. 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.

Table 25: **Common PCI Configuration Space Header**

	31	*16*	*15*	*0*	
	Device ID		*Vendor ID*		*000h*
	Status		*Command*		*004h*
	Class Code			*Rev ID*	*008h*
	BIST	*Header*	*Lat Timer*	*Cache Ln*	*00Ch*

Table 25: **Common PCI Configuration Space Header** *(cont'd)*

	Header Type Specific (see Table 26 and Table 28)				*010h*
					014h
					018h
					01Ch
					020h
					024h
					028h
					02Ch
					030h
				CapPtr	*034h*
					038h
			Intr Pin	*Intr Line*	*03Ch*
	Reserved				*040h-07Ch*
	PM Capability		*NxtCap*	*PM Cap*	*080h*
	Data	*Reserved*	*PMCSR*		*084h*
	Reserved				*088h-08Ch*
Customizable	*MSI Control*		*NxtCap*	*MSI Cap*	*090h*
	Message Address (Lower)				*094h*
	Message Address (Upper)				*098h*
	Reserved		*Message Data*		*09Ch*
	Mask Bits				*0A0h*
	Pending Bits				*0A4h*
	Reserved				*0A8h-0ACh*
Optional	*MSI-X Control*		*NxtCap*	*MSI-X Cap*	*0B0h*
	Table Offset			*Table BIR*	*0B4h*
	PBA Offset			*PBA BIR*	*0B8h*
	Reserved				*0BCh*
	PE Capability		*NxtCap*	*PE Cap*	*0C0h*
	PCI Express Device Capabilities				*0C4h*
	Device Status		*Device Control*		*0C8h*
	PCI Express Link Capabilities				*0CCh*
	Link Status		*Link Control*		*0D0h*
Root Port Only	*Slot Capabilities*				*0D4h*
	Slot Status		*Slot Control*		*0D8h*
	Root Capabilities		*Root Control*		*0DCh*
	Root Status				*0E0h*
	PCI Express Device Capabilities 2				*0E4h*
	Device Status 2		*Device Control 2*		*0E8h*
	PCI Express Link Capabilities 2				*0ECh*
	Link Status 2		*Link Control 2*		*0F0h*

Table 25: **Common PCI Configuration Space Header** *(cont'd)*

	Unimplemented Configuration Space(Returns 0x00000000)				*0F4h-0FCh*
Always Enabled	*Next Cap*	*Cap. Ver.*	*PCI Express Extended Cap. ID (AER)*		*100h*
	Uncorrectable Error Status Register				*104h*
	Uncorrectable Error Mask Register				*108h*
	Uncorrectable Error Severity Register				*10Ch*
	Correctable Error Status Register				*110h*
	Correctable Error Mask Register				*114h*
	Advanced Error Cap. and Control Register				*118h*
	Header Log Register 1				*11Ch*
	Header Log Register 2				*120h*
	Header Log Register 3				*124h*
	Header Log Register 4				*128h*
	Reserved				*12Ch*
Optional, Root Port only	*Root Error Command Register*				*130h*
	Root Error Status Register				*134h*
	Error Source ID Register				*138h*
	Reserved				*13Ch*
Optional	*Next Cap*	*Cap. Ver.*	*PCI Express Extended Capability - Alternate Requester ID (ARI)*		*140h*
	Control		*Next Function*	*Function Groups*	*144h*
	Reserved				*148h-14Ch*
Optional	*Next Cap*	*Cap. Ver.*	*PCI Express Extended Capability - DSN*		*150h*
	PCI Express Device Serial Number (1st)				*154h*
	PCI Express Device Serial Number (2nd)				*158h*
	Reserved				*15Ch*
Optional	*Next Cap*	*Cap. Ver.*	*PCI Express Extended Capability - Power Budgeting Enhanced Capability Header*		*160h*
	Reserved			*DS*	*164h*
	Reserved	*Power Budget Data - State D0, D1, D3, ...*			*168h*
	Power Budget Capability				*16Ch*
	Reserved				*170h-1B4h*
Optional	*Next Cap*	*Cap. Ver.*	*PCI Express Extended Capability ID - Latency Tolerance Reporting (LTR)*		*1B8h*
	No-Snoop		*Snoop*		*1BCh*
Optional	*Next Cap*	*Cap. Ver.*	*PCI Express Extended Capability ID - Dynamic Power Allocation*		*1C0h*
	Capability Register				*1C4h*
	Latency Indicator				*1C8h*
	Control		*Status*		*1CCh*
	Power Allocation Array Register 0				*1D0h*
	Power Allocation Array Register 1				*1D4h*

Table 25: **Common PCI Configuration Space Header** *(cont'd)*

	Reserved				*1D8h-1FCh*
Optional	*Next Cap*	*Cap. Ver.*	*PCI Express Extended Capability ID - Single Root I/O Virtualization (SR-IOV)*		*200h*
	Capability Register				*204h*
	SR-IOV Status (not supported)		*Control*		*208h*
	Total VFs		*Initial VFs*		*20Ch*
	Function Dependency Link		*Number VFs*		*210h*
	VF Stride		*First VF Offset*		*214h*
	VF Device ID		*Reserved*		*218h*
	Supported Page Sizes				*21Ch*
	System Page Size				*220h*
	VF Base Address Register 0				*224h*
	VF Base Address Register 1				*228h*
	VF Base Address Register 2				*22Ch*
	VF Base Address Register 3				*230h*
	VF Base Address Register 4				*234h*
	VF Base Address Register 5				*238h*
	Reserved				*23Ch*
	Reserved				*240h-270h*
Optional	*Next Cap*	*Cap. Ver.*	*PCI Express Extended Capability ID - Transaction Processing Hints (TPH)*		*274h*
	Capability Register				*278h*
	Requester Control Register				*27Ch*
	Reserved		*Steering Tag Upper*	*Steering Tag Lower*	*280h*
	Reserved				*284h - 2FCh*
Optional	*Next Cap*	*Cap. Ver.*	*PCI Express Extended Capability ID - Secondary PCIe Extended Capability*		*300h*
	Lane Control (not supported)				*304h*
	Reserved			*Lane Error Status*	*308h*
	Lane Equalization Control Register 0				*30Ch*
	Lane Equalization Control Register 1				*310h*
	Lane Equalization Control Register 2				*314h*
	Lane Equalization Control Register 3				*318h*

Table 25: **Common PCI Configuration Space Header** *(cont'd)*

	Reserved			*31Ch-33Ch*
Optional	*Next Cap*	*Cap. Ver.*	*PCI Express Extended Capability ID - MCAP*	*340h*
	Capability Register			*344h*
	FPGA JTAG ID			*348h*
	FPGA Bitstream Version			*34Ch*
	Status Register			*350h*
	Control Register			*354h*
	Data Write Register			*358h*
	Read Data 0 Register			*35Ch*
	Read Data 1 Register			*360h*
	Read Data 2 Register			*364h*
	Read Data 3 Register			*368h*
	Reserved			*36Ch-3BCh*
Optional	*Next Cap*	*Cap. Ver.*	*PCI Express Extended Capability - VC*	*3C0h*
	Port VC Capability Register 1			*3C4h*
	Port VC Capability Register 2			*3C8h*
	Port VC Status		*Port VC Control*	*3CCh*
	VC Resource Capability Register 0			*3D0h*
	VC Resource Control Register 0			*3D4h*
	VC Resource Status Register 0			*3D8h*
	Reserved			*400h-FFFh*
The MSI Capability Structure varies depending on the selections in the AMD Vivado™ IDE.				

Table 26: **Type 0 PCI Configuration Space Header**

31	16	15	0	
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Rev ID	08h
BIST	Header	Lat Timer	Cache Ln	0Ch
Base Address Register 0				10h
Base Address Register 1				14h
Base Address Register 2				18h
Base Address Register 3				1Ch
Base Address Register 4				20h
Base Address Register 5				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved			CapPtr	34h
Reserved				38h
Max Lat	Min Gnt	Intr Pin	Intr Line	3Ch

Table 27: **Type 1 PCI Configuration Space Header**

31	16	15	0	
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Rev ID	08h
BIST	Header	Lat Timer	Cache Ln	0Ch
Base Address Register 0				10h
Base Address Register 1				14h
Second Lat Timer	Sub Bus Number	Second Bus Number	Primary Bus Number	18h
Secondary Status		I/O Limit	I/O Base	1Ch
Memory Limit		Memory Base		20h
Prefetchable Memory Limit		Prefetchable Memory Base		24h
Prefetchable Base Upper 32 Bits				28h
Prefetchable Limit Upper 32 Bits				2Ch
I/O Limit Upper 16 Bits		I/O Base Upper 16 Bits		30h
Reserved			CapPtr	34h
Expansion ROM Base Address				38h
Bridge Control		Intr Pin	Intr Line	3Ch

Chapter 4

Designing with the Core

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

Shared Logic

This feature comes with two different sharing options.

- GT common in the core or example design.
- GT Wizard in the core or example design.

GT COMMON Option

This feature allows you to share common logic across multiple instances of PCIe® blocks or with other cores, with certain limitations. Shared logic minimizes the HDL modifications needed by locating the logic to be shared to the top module of the design. It also enables additional ports on the top module to facilitate sharing. Shared logic is applicable for both Endpoint mode and Root Port mode.

In the AMD Vivado™ Design Suite, the shared logic options are available in the Shared Logic page when customizing the core.

There are two types of logic sharing:

- Shared logic in the core
- Shared logic in the example design

In both cases, the GT_COMMON block is shared.

IMPORTANT! *For the Include Shared Logic in Example Design option to generate the corresponding modules in the support directory, run the **Open IP Example Design** command after the output products are generated. For the Include Shared Logic in Core (default) option, these modules are generated in the source directory.*

IMPORTANT! *The Shared Logic page is visible only when the link speed other than Gen1 is selected. In the shared logic feature, you can use only the QPLL1 block.*

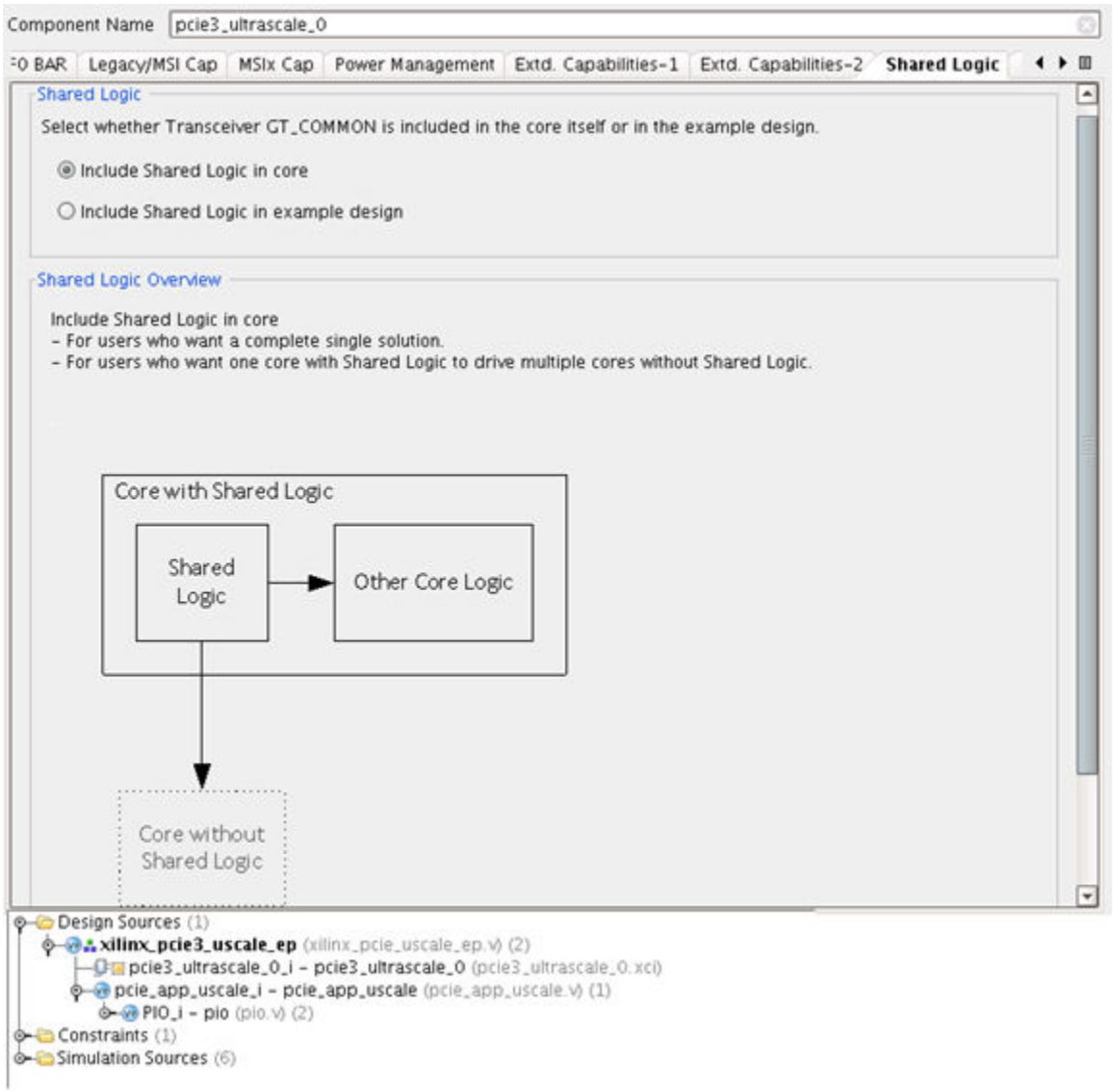
- In the case of Gen1 speeds, the design uses CPLL, hence it cannot be shared and the shared logic feature is disabled.

- In the case of Gen2 speeds, QPLL1 or CPLL can be selected. If CPLL is selected (using the PLL Selection option on GT Settings Page) the Shared Logic page is disabled.

Shared Logic in the Core

This feature allows sharing of the GT_COMMON block while it is still internal to the core (not at the support wrapper). Enable it by selecting **Include Shared Logic in Core** in the Shared Logic page (the default option).

Figure 2: **Shared Logic in the Core**



Shared GT_COMMON

A quad phase-locked loop (QPLL) in GT_COMMON can serve a quad of GT_CHANNEL instances. If the PCIe core is configured as X1 or X2 and is using a QPLL, the remaining GT_CHANNEL instances can be used by other cores by sharing the same QPLL and GT_COMMON.

To use the shared GT_COMMON instances, select the **Include Shared Logic in example design** option on the Shared Logic tab. When this feature is selected, the GT_COMMON instance is moved from the pipe wrappers to the support wrapper of the example design. It also enables additional ports to the top level to facilitate sharing of GT_COMMON.

Shared logic for GT_COMMON helps conserve FPGA resources and also reduces dedicated clock routing within the single GT quad.

Shared GT_COMMON Use Case with GTH

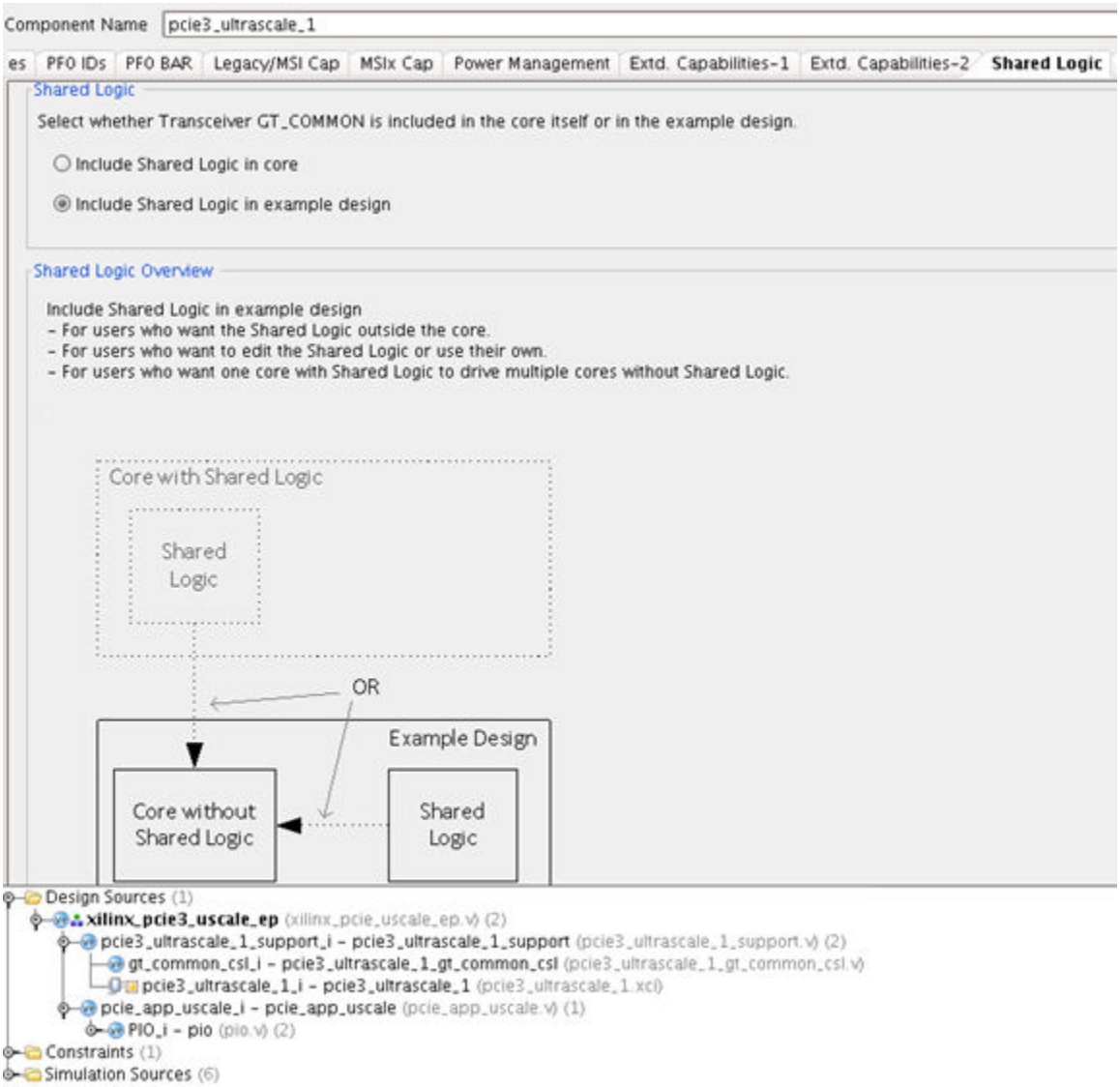
Table 28: **Shared GT_COMMON Use Case**

GT – PCIe Max Link Speed	Device – PCIe Max Link Speed	Shared GT_COMMON
GTH	Kintex UltraScale (040) – PCIe Gen3	PCIe pipe wrappers use QPLL for Gen3 and CPLL for Gen1/Gen2. If PCIe is Gen3 capable but operating at a lower speed, other IP can use it.

Limitations

- GTH pipe wrappers reset the QPLL when the PCIe change the rate to Gen3. The sharing core must be able to handle this situation.
- Pipe wrappers commonly use a channel phase-locked loop (CPLL) for Gen1 or Gen2 PCIe, and QPLL for Gen3. If the Gen3 PCIe can operate at a lower speed, pipe wrappers might not require a QPLL.
- The settings of the GT_COMMON should not be changed because they are optimized for the PCIe core.

Figure 3: **Shared Logic in the Example Design**



GT Wizard Option

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) or *UltraScale Architecture GTY Transceivers User Guide* (UG578).

Figure 4: **GT Wizard Shared Logic**

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 uses 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:

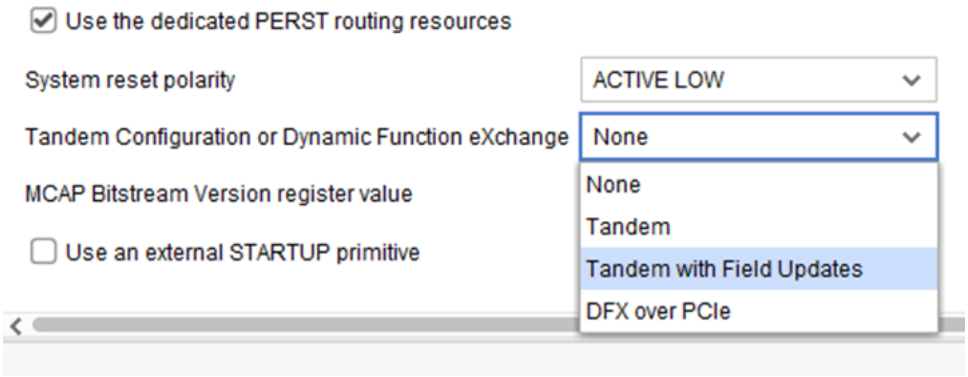
- **Tandem PROM:** Load the single two-stage bitstream from the flash.
- **Tandem PCIe:** Load the first stage bitstream from flash, and deliver the second stage bitstream over the PCIe link to the MCAP.
- **Tandem with Field Updates:** After a Tandem PROM or Tandem PCIe initial configuration, update 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 + Dynamic Function eXchange:** This is a more general case of Tandem Configuration followed by Dynamic Function eXchange (DFX) of any size or number of dynamic regions.
- **Dynamic Function eXchange over PCIe:** This is a standard configuration followed by DFX, using the PCIe / MCAP as the delivery path of partial bitstreams.

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

1. Change the **Mode** to Advanced.
2. Change the **Tandem Configuration or Dynamic Function eXchange** option according to your particular case:

- **Tandem** for Tandem PROM, Tandem PCIe or Tandem + Dynamic Function eXchange use cases.
- **Tandem with Field Updates** ONLY for the predefined Field Updates use case.
- **DFX over PCIe** to enable the MCAP link for Dynamic Function eXchange, without enabling Tandem Configuration.

Figure 5: **Tandem Configuration or Dynamic Function eXchange Option**



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

Supported Devices

The UltraScale Devices Gen3 Integrated Block for PCIe core and Vivado tool flow support implementations targeting AMD reference boards and specific part/package combinations.

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

Table 29: **Tandem PROM/PCIe Supported Configurations**

Device Support	Part[1]	PCIe Block Location	PCIe Reset Location	Tandem Configuration	Tandem with Field Updates
HDL	Verilog Only				
PCIe Configuration	All configurations (max: X8Gen3)				
AMD Reference Board Support	KCU105 Evaluation Board for Kintex UltraScale FPGA VCU108 Evaluation Board for Virtex UltraScale FPGA				

Table 29: **Tandem PROM/PCIe Supported Configurations** *(cont'd)*

Device Support	Part[1]	PCIe Block Location	PCIe Reset Location	Tandem Configuration	Tandem with Field Updates
Kintex UltraScale	XCKU025	PCIE_3_1_X0Y0	IOB_X1Y103	Production	Production
	XCKU035	PCIE_3_1_X0Y0	IOB_X1Y103	Production	Production
	XCKU040	PCIE_3_1_X0Y0	IOB_X1Y103	Production	Production
	XCKU060	PCIE_3_1_X0Y0	IOB_X2Y103	Production	Production
	XCKU085	PCIE_3_1_X0Y0	IOB_X2Y103	Production	Production
	XCKU095	PCIE_3_1_X0Y0	IOB_X1Y103	Production	Production
	XCKU115	PCIE_3_1_X0Y0	IOB_X2Y103	Production	Production
Virtex UltraScale	XCVU065	PCIE_3_1_X0Y0	IOB_X1Y103	Production	Production
	XCVU080	PCIE_3_1_X0Y0	IOB_X1Y103	Production	Production
	XCVU095	PCIE_3_1_X0Y0	IOB_X1Y103	Production	Production
	XCVU125	PCIE_3_1_X0Y0	IOB_X1Y103	Production	Production
	XCVU160	PCIE_3_1_X0Y1	IOB_X1Y363	Production	Production
	XCVU190	PCIE_3_1_X0Y2	IOB_X1Y363	Production	Production
	XCVU440	PCIE_3_1_X0Y2	IOB_X1Y363	Production	Production

Notes:

1. Only production silicon is officially supported. Bitstream generation is disabled for all engineering sample silicon (ES1 and ES2) devices.

Overview of Tandem Tool Flow

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

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

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

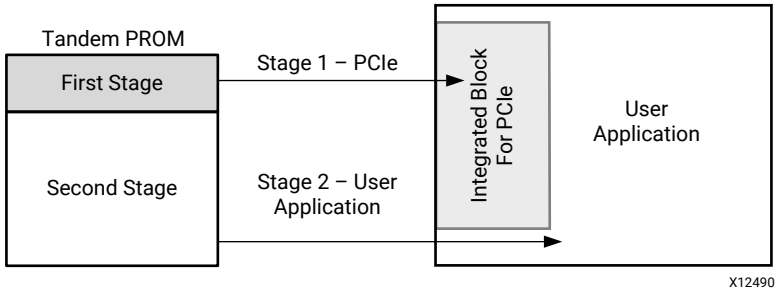
When the example design is created, an example XDC file is generated with certain constraints that need to be copied over into your XDC file for your specific project. The specific constraints are documented in the example design XDC file. In addition, this example design XDC file contains examples of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

When generating the PCIe IP, you will see there is no distinction between Tandem PROM and Tandem PCIe. Both methodologies generate the same IP core, so the selection in the Vivado IDE is simply Tandem. The divergence point is at the `write_bitstream` step, where a property (HD.TANDEM_BITSTREAMS) defines whether one BIT file (Tandem PROM) or two BIT files (Tandem PCIe) are needed. The core and corresponding implementation results are identical.

Tandem PROM

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

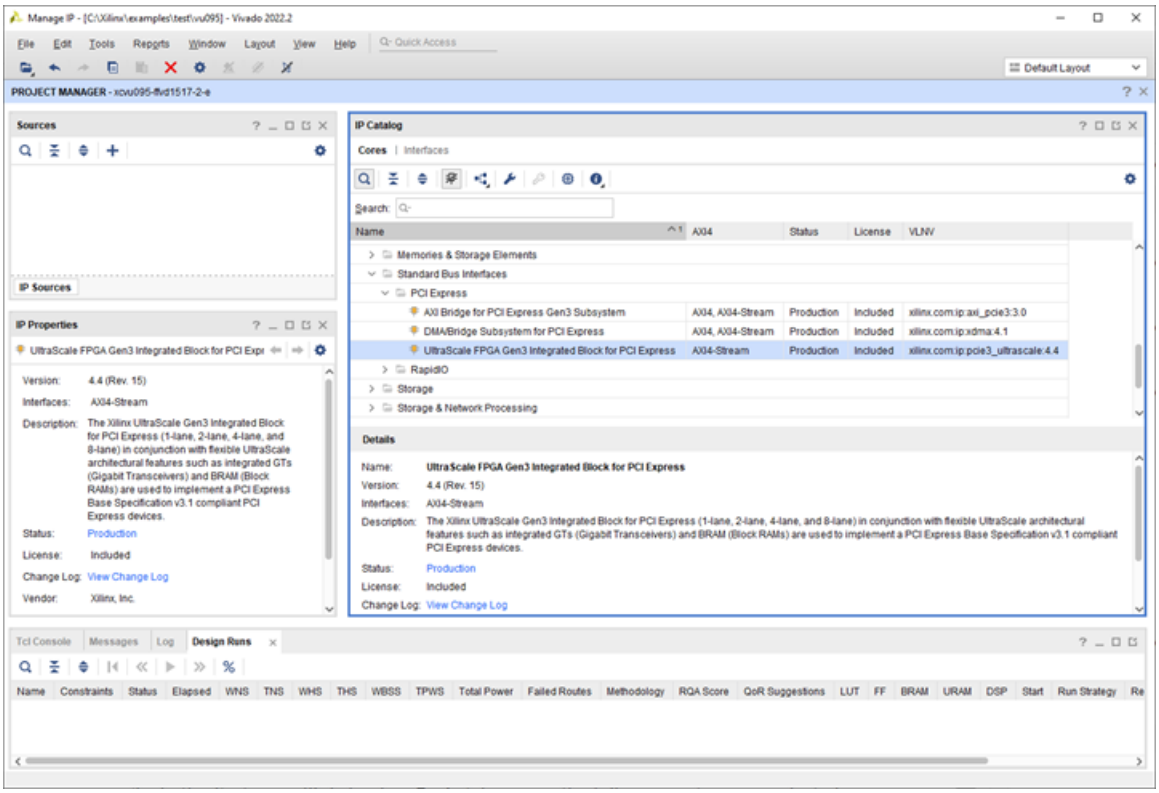
Figure 6: **Tandem PROM Bitstream Load Steps**



Tandem PROM VCU108 Example Tool Flow

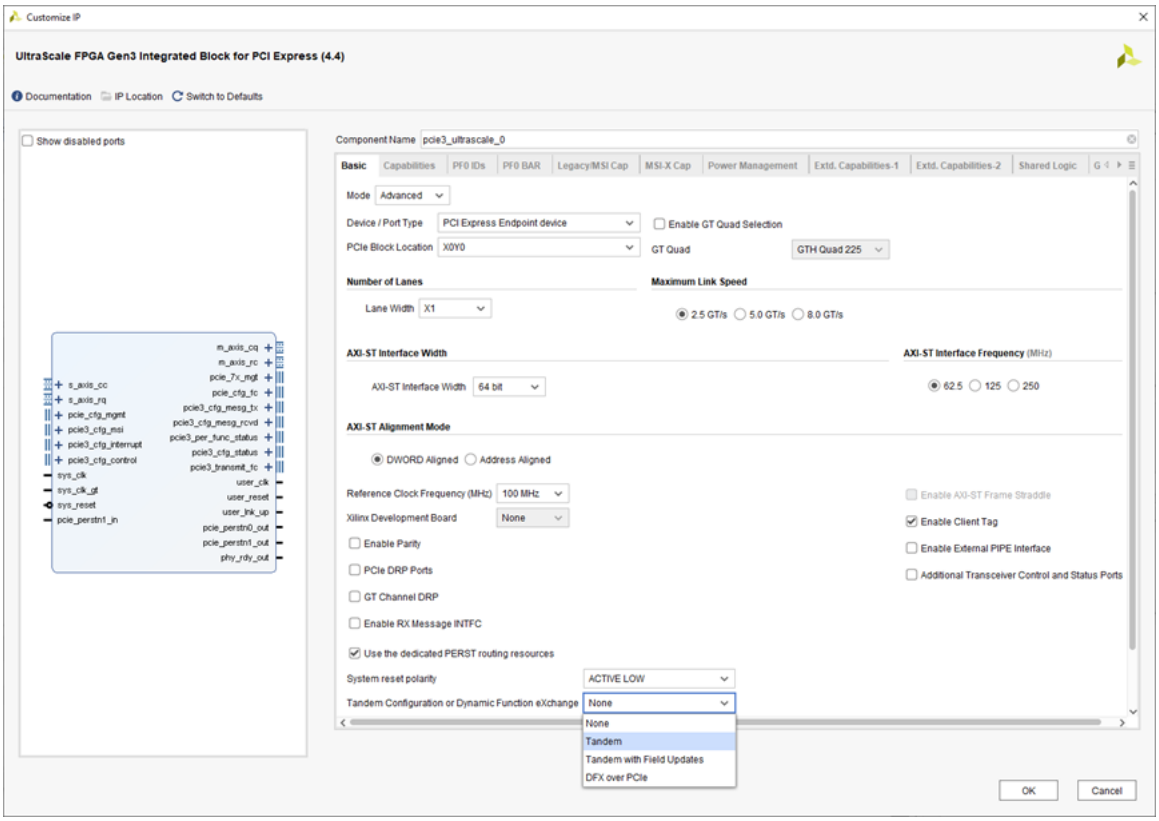
This section demonstrates the Vivado tool flow from start to finish when targeting the VCU108 reference board. Paths and pointers within this flow description assume the default component name “pcie3_UltraScale_0” is used.

1. Create a new Vivado project, and select a supported part/package shown in the preceding table.
2. In the Vivado IP catalog, expand **Standard Bus Interfaces → PCI Express**, and double-click **UltraScale FPGA Gen3 Integrated Block for PCI Express** to open the Customize IP dialog box.



3. In the Customize IP dialog box Basic tab, ensure the following options are selected:
   - Mode: *Advanced*
   - PCIe Block Location: *X0Y0*

     ***Note*:** Use the required PCIe Block Location for the device targeted, as listed in the preceding table.
   - Tandem Configuration or Dynamic Function eXchange: *Tandem*



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 (for example, `xilinx_pcie3_uscale_ep_x8g3.xdc`):

   - Uncommend and customize any constraints that define the configuration settings.
   - The one constraint that is required is CONFIG_MODE. For example: `set_property CONFIG_MODE BPI16 [current_design]`

   For more information, see Programming the Device.

9. Generate the bitstream.

   After Synthesis and Implementation is complete, click **Generate Bitstream** in the Flow Navigator. A bitstream supporting Tandem configuration is generated in the `runs` directory, for example: `./pcie_ultrascale_0_example.runs/impl/xilinx_pcie3_uscale_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_pcie3_uscale_ep_tandem1.bit` and `xilinx_pcie3_uscale_ep_tandem2.bit`.

10. Generate the PROM file.

    Run the following command in the Vivado Tcl Console to create a PROM file supported on the VCU108 development board.

    ```
    write_cfgmem -format mcs -interface BPI -size 256 -loadbit up 0x0
    xilinx_pcie3_uscale_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 Gen3 Integrated Block for PCIe core manages many design details, allowing you to focus your attention on the user application.

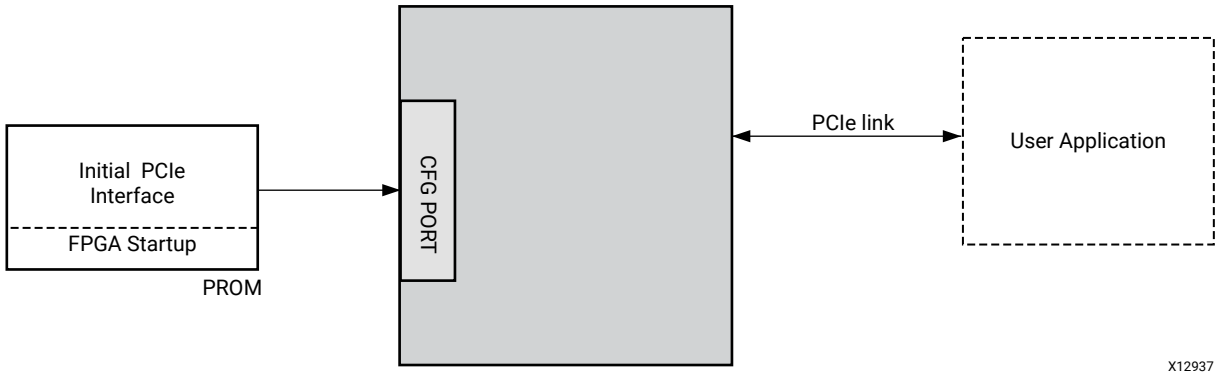
Tandem PCIe

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

VIDEO: *Create a Tandem PCIe Design for the KCU105 explains how to create a Tandem design targeting the KCU105 Evaluation Kit.*

The following figure illustrates the bitstream loading flow.

Figure 7: **Tandem PCIe Bitstream Load Steps**

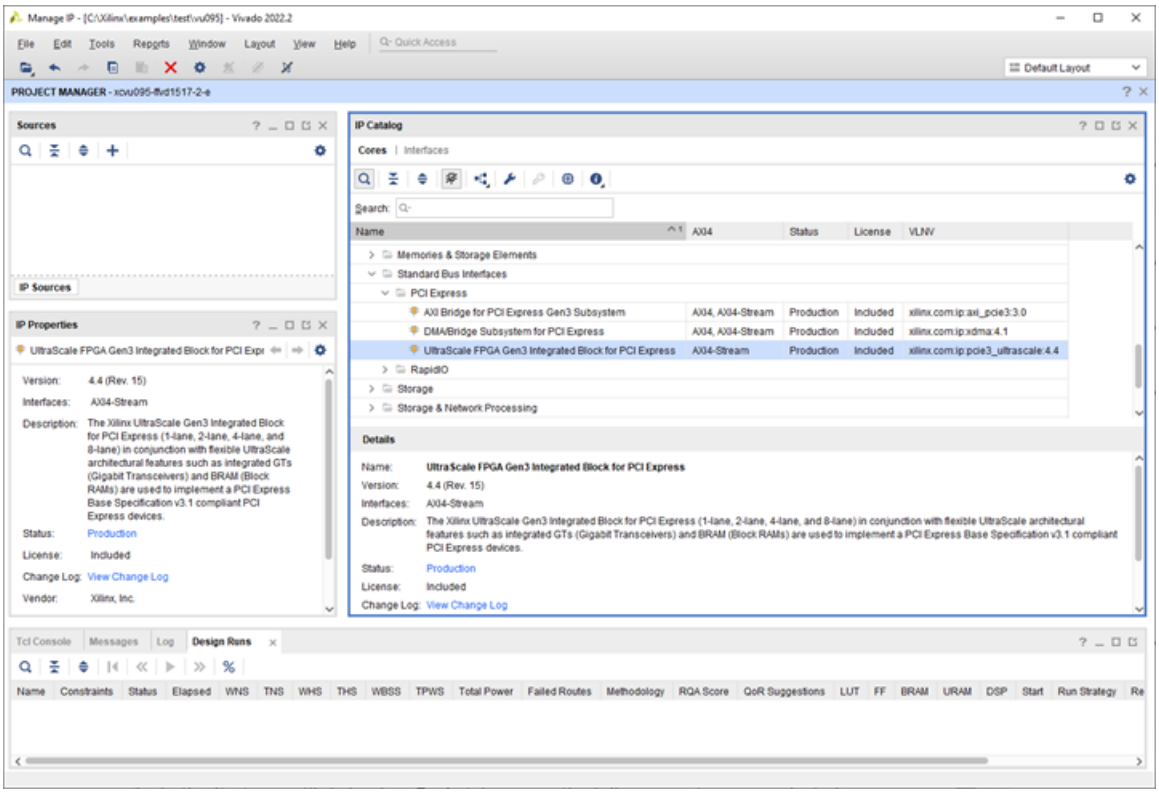


Tandem PCIe is similar to the standard model used today in terms of tool flow and bitstream generation. Two bitstreams are produced when running bitstream generation. One BIT file representing the stage 1 is downloaded into the PROM while the other BIT file representing the user application (stage 2)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

Tandem PCIe VCU108 Example Tool Flow

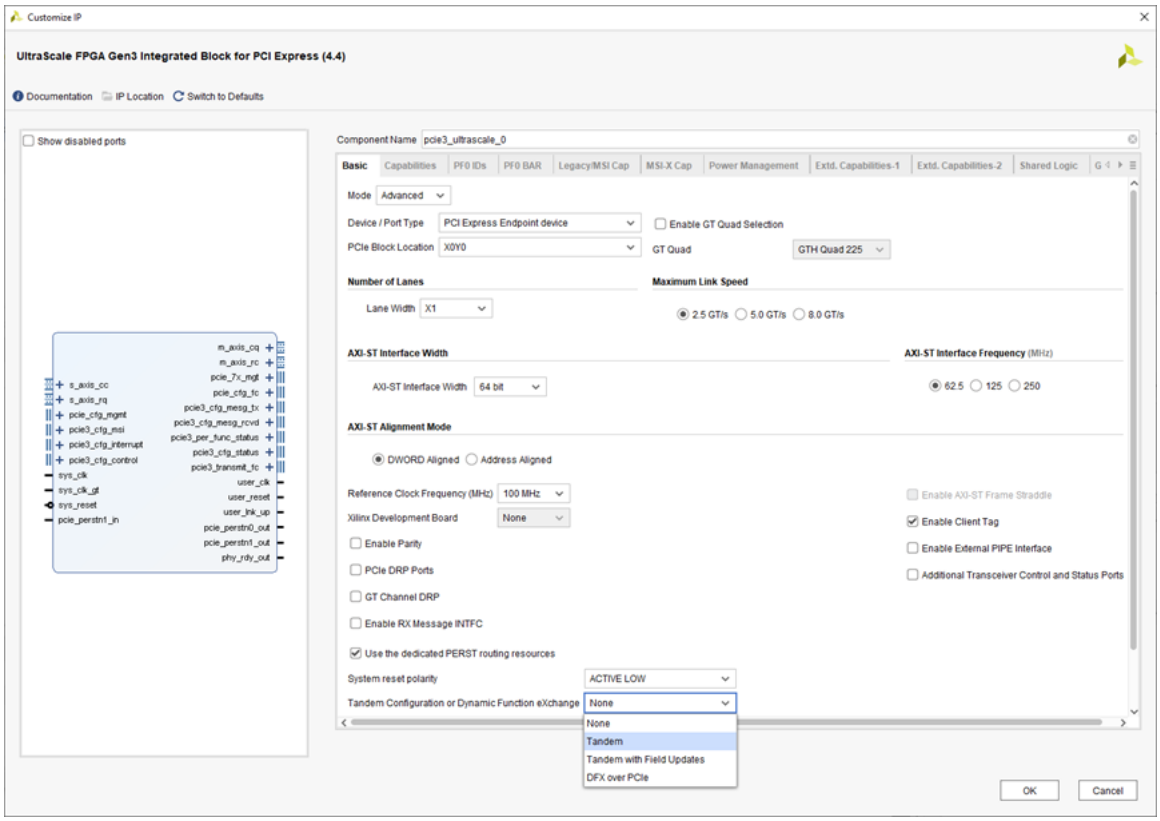
This section demonstrates the Vivado tool flow from start to finish when targeting the VCU108 reference board. Paths and pointers within this flow description assume the default component name `pcie3_ultrascale_0` is used.

1. When creating a new Vivado project, select a supported part/package shown in the preceding table.
2. In the Vivado IP catalog, expand **Standard Bus Interfaces → PCI Express** > , and double-click **UltraScale FPGA Gen3 Integrated Block for PCI Express** to open the Customize IP dialog box.



3. In the Customize IP dialog box Basic tab, ensure the following options are selected:
   - Mode: *Advanced*
   - PCIe Block Location: *X0Y0*

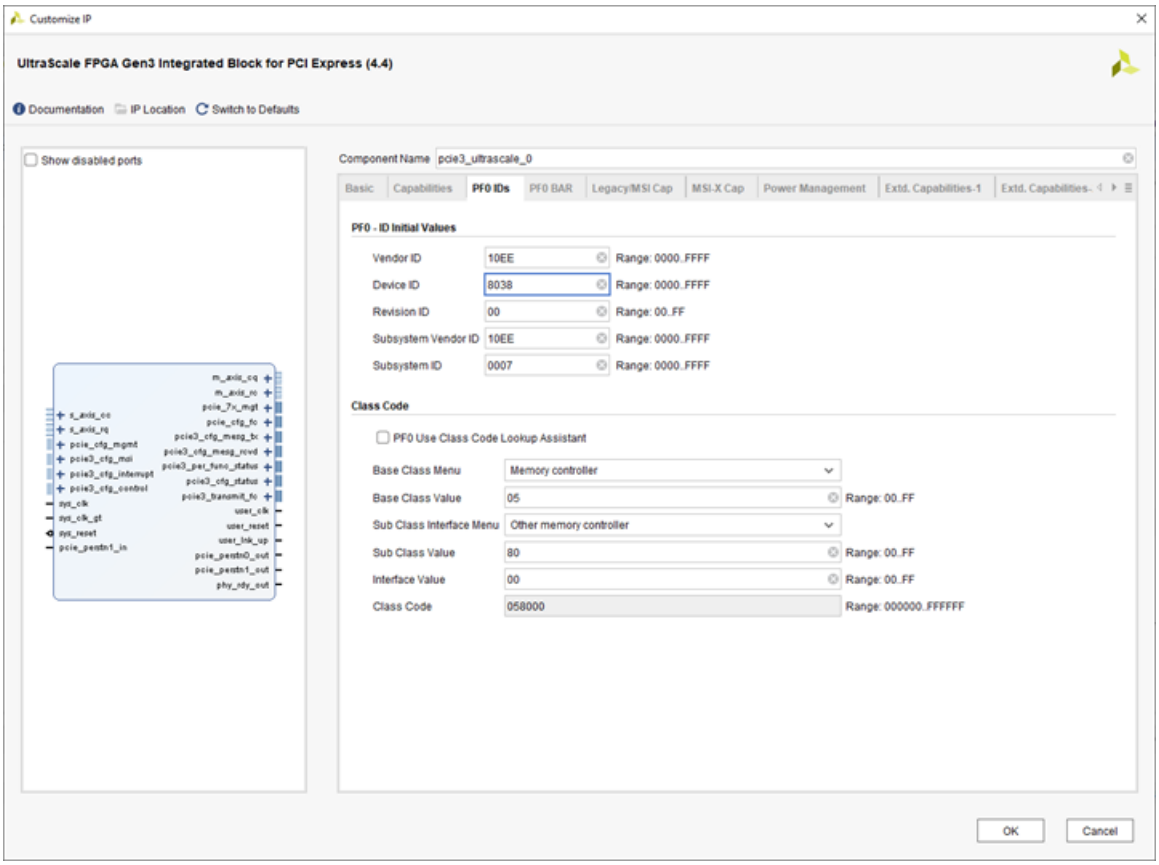
     ***Note*:** Use the required PCIe Block Location for the device targeted, as listed in the preceding table.
   - Tandem Configuration or Dynamic Function eXchange: *Tandem*



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'h 8038`.In the PF0 IDs tab, set:

   - Vendor ID: *10EE*
   - Device ID: *8038*

   ***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.



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

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

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

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

7. Run Synthesis and Implementation.

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

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

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

   - modifying the constraints in the PCIe IP constraint file (for example, `pcie3_ultrascale_0_tandem`).
   - requesting two explicit bitstreams by setting these properties, as seen in the example design constraint file:

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

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

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

9. Generate the bitstream.

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

   ```
   xilinx_pcie3_uscale_ep_tandem1.bit|
   ```

   ```
   xilinx_pcie3_uscale_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 the VCU108 development board.

    ```
    write_cfgmem -format mcs -interface BPI -size 256 -loadbit up 0x0
    xilinx_pcie3_uscale_ep_tandem1.bit
    xilinx_pcie3_uscale_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 Gen3 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. The solution uses Tandem Configuration to initially configure the device when the power is turned on, followed by Dynamic Function eXchange (DFX) of a predefined region in the FPGA. This allows the stage 1 bitstream to be locked in flash, with updates to that image are only required if the UltraScale Devices Gen3 Integrated Block for PCIe core characteristics or the I/O or clock management blocks in the configuration bank (bank 65) must change. Field Updates allows the user application, basically everything else in the design, to be dynamically reloaded as new features or functionality are needed.

Tandem with Field Updates is a specific, predefined use case of a more general Tandem + DFX solution. These two solutions (Tandem and DFX) are also supported in general in the same design, allowing you to partially reconfigure smaller and multiple regions within the user application.

After the UltraScale Devices Gen3 Integrated Block for PCIe core is generated, a sample design can be created that provides the template for the Tandem with Field Updates structure. This design shows the required structure, floorplan, properties, and scripts that can be adapted for your design. Follow this example to sort your design into two sections, mapping them to the two bitstream stages. The sample design is delivered in a scripted non-project mode, but project mode can be used as well.

Differences from Tandem Configuration

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

Design Layout

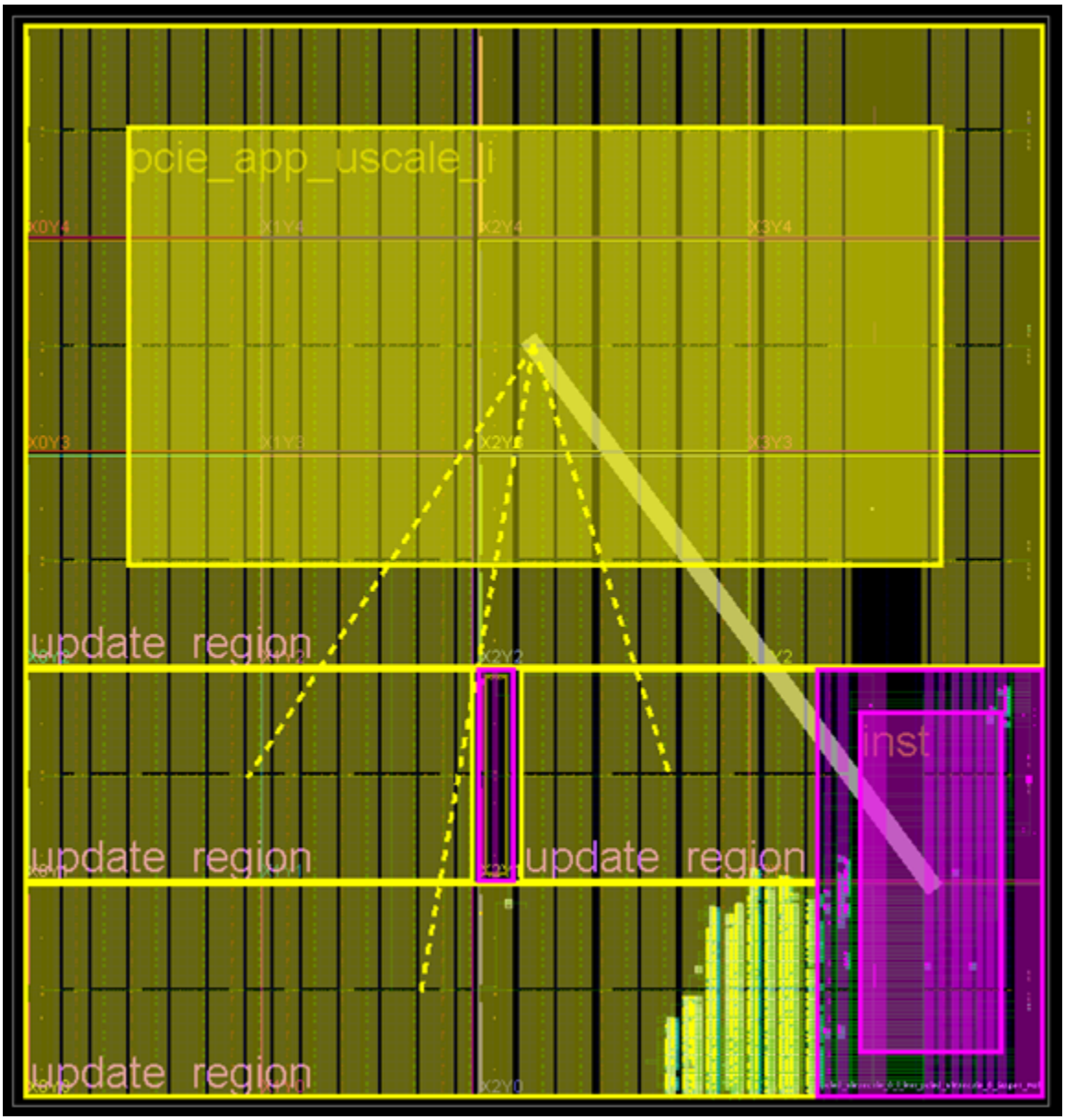
First, for both variations, the floorplan is established as part of the IP creation and should not be modified, but Tandem with Field Updates creates two sets of Pblocks instead of one. In addition to the same Pblocks tagged with HD.TANDEM for the stage 1 logic, a second, much larger set of Pblocks tagged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

The following figure shows the floorplan generated for the KU040 sample design for Tandem with Field Updates. The pink region is reserved for the UltraScale Devices Gen3 Integrated Block for PCIe core. This region includes the PCIe hard block, CLB, block RAM and transceiver sites for implementing the IP, and one I/O bank to enable the physical reset pin. The yellow region is the inverse of the pink and represents the Reconfigurable Partition (RP) for the user application. It covers remaining resources including all clocks, transceivers, I/O and logic not covered by the PCIe IP.

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

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

Figure 8: **Tandem with Field Updates Floorplan for the KU040**



Design Structure

The second difference is the design structure. In order to swap one user application from one version to the next, it must be completely enclosed within its own level of hierarchy. The interface of this instantiation cannot change; otherwise, the top-level static design needs to be recompiled. Everything other than the UltraScale Devices Gen3 Integrated Block for PCIe core (in its own level of hierarchy below top) and any I/O logic (buffers, MMCM, PLL, XiPhy, and so on.) that are placed in bank 65 are in this level of hierarchy (and below). This means that all I/O logic for all other banks must be placed here and not inferred at the top level, so instantiation of I/O buffers is required.

TIP: *To see a simple example of this design structure, generate an example design while targeting the KCU105 demo board.*

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

All other considerations for Tandem Configuration are still applicable for Tandem with Field Updates when working with UltraScale devices. For example, PERSIST must be used for Tandem PROM, and stage 1 must remain linked to stage 2 from the same implementation result. Reconfigurable Stage Twos, the feature that allows you to pair new stage 2 bitstreams with a given stage 1 as well as the ability to dynamically reconfigure with stage 2 bitstreams, this is only supported for UltraScale+ devices.

Tandem Configuration with Field Updates Software Flow

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

1. Launch the Customize IP dialog box to customize the UltraScale Devices Gen3 Integrated Block for PCIe core.

   ***Note*:** This solution is supported only in version 4.2 (and newer) of the IP.

2. Customize the PCIe IP core with **Tandem with Field Updates** selected. The **Advanced** Mode option must be selected to see this option. If debug capabilities are desired (most designs are expected to require this), select the **Use an external STARTUP primitive** option.
3. Generate output products by using the default **Out of context per IP** synthesis option. This synthesizes the IP to create a checkpoint that can be inserted in your full design.
4. Right-click the IP in the Design Sources tab, and select **Open IP Example Design**.

   IMPORTANT! *Do not implement this sample design within the Vivado IDE in project mode.*

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

5. In a Vivado Tcl shell, source `design_field_updates.tcl`, which is found in the project directory. This file compiles the example design with two versions:

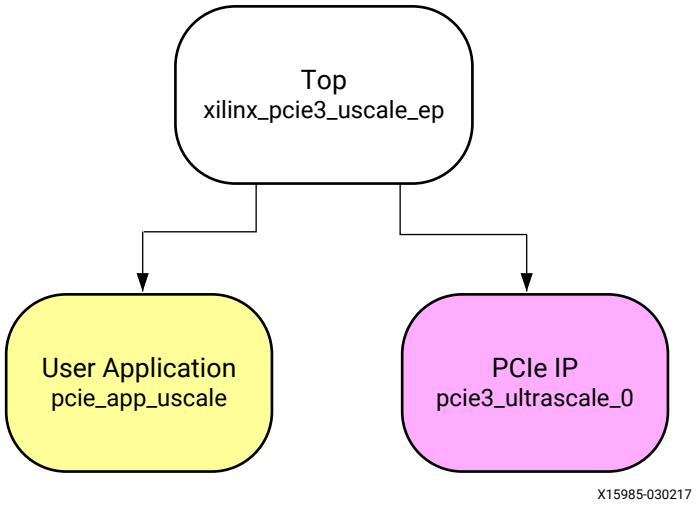
- With the default settings, Ver1 is the initial design, and is the one that is expected to be used for initial boot, so Tandem bitstreams are created.
- Ver2 is intended to be the update to Ver1, so only partial bitstreams are created.

Details on the Sample Design

The simple PIO design generated with the IP represents the required design structure for Tandem with Field Updates. The top-level design file, `xilinx_pcie_uscale_ep.v`, declares the top level pin list and instantiates `pcie3_ultrascale_0` (the IP) and `pcie_app_uscale` (everything else). These two submodules are tagged with the HD.TANDEM and HD.RECONFIGURABLE properties, respectively, to instruct the implementation tools to follow both sets of rules during place and route.

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

Figure 9: **Required Design Hierarchy for Tandem With Field Updates**



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

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

Details on the Design Scripts

The `design_field_updates.tcl` script is the master, calling other scripts in the `field_update_scripts` folder. Set variables in this master script to synthesize and implement only the first version (Ver1) or only the second version, (Ver2) and use it as a template if additional versions are needed. The fundamental flow is a Dynamic Function eXchange flow. For more details about DFX, see the *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909). Sample scripts are provided with the *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947).

Here are a few things to note about the scripts.

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

Bitstream Generation

The supplied scripts create bitstreams for any versions requested. The `set runUpdateVerXBitstreams 1` flag (where X represents the version number) calls the bitstream generation routine later script. As seen on line 159, multiple values can be supplied to generate the different bitstream types, depending on what is needed. The following values generate these bitstreams (showing ver1 as an example):

- **TandemPCIe:** Generates the following bitstreams, and the partial bitstreams below.
  - `ver1_tpcie_tandem1` – stage 1 bitstream for Tandem PCIe, to be stored in flash.
  - `ver1_tpcie_tandem2` – stage 2 bitstream for Tandem PCIe, to be delivered over PCIe link.
- **TandemPROM:** Generates the following bitstream, and the partial bitstream below.
  - `ver1_tprom` – two-stage Tandem PROM bitstream, to be stored in flash.
- **PR:** Generates only these partial bitstreams, where t* is either “tpcie” or “tprom."
  - `ver1_t*_update_region_partial_clear` – clearing bitstream to prepare the user update region to reconfigure from ver1 to another version.
  - `ver1_t*_update_region_partial` – partial bitstream to load in functionality of ver1.

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

Ver1 does not have to be the version that is booted from flash. Rather, this version should be the most challenging design version available. The place and route results of this first version determine the partition pin locations, which locks the routing on the interface between the PCIe core and the user application. So if multiple design versions are to be compiled at once, specify **TandemPCIe** or **TandemPROM** for the version that is to be stored in flash for the initial configuration, regardless of which version number it happens to be; for the remaining versions, simply select **PR**.

Hardware Operation Details

The initial configuration of the device is no different than the normal Tandem Configuration. Initial configuration of the device consists of two stages, either as a single bitstream from flash using the Tandem PROM approach, or as two separate bitstreams (one loaded using flash, one loaded over PCIe) using the Tandem PCIe approach. Both of these variations are supported in Tandem with Field Updates for UltraScale devices only, as the IP core is identical for these two approaches. After the stage 1 is loaded, only the PCIe IP is operational, with limited functionality as the rest of the design behind it does not yet exist. It is able to link train and be recognized by the Root Port in the system. After the stage 2 bitstream is loaded, the devices is in normal operational mode.

Subsequent dynamic updates follow the fundamental rules for UltraScale device partial reconfiguration. One such rule is the application of clearing files. The clearing file for the current application must be loaded before the new partial bitstream is delivered. For example, if the "ver1" version of the design is configured at power up, the `ver1_partial_clear.bin` must be sent to prepare the user application for `ver2_partial.bin` (or any new partial image version). In the next FPGA update, `ver2_partial_clear.bin` would be sent to prepare the region for a version 3 (ver3, or any new image, including a return to ver1).

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

1. Power on the FPGA. `ver1_tpcie_tandem1.mcs` is sent from local flash.
   - At this point, the PCIe Endpoint is functional, and enumeration can occur.
   - The PCIe core is isolated from the rest of the unconfigured device.
2. Deliver `ver1_tpcie_tandem2.bin` over the PCIe link.
   - The complete device is now programmed, and the design switches automatically to full use mode by establishing communication between PCIe and the rest of the design.
3. Operate the FPGA for as long as you would like.
4. When a request to update is received, deliver `ver1_tpcie_update_region_partial_clear.bin` over the PCIe link.
   - This readies the user application region; clocks to this region are disabled, so the user application is now non-functional.
   - The PCIe core is isolated from the user application region as part of the software driver instructions.
5. Deliver the new user application by sending `ver2_tpcie_update_region_partial.bin` over the PCIe link.
   - The full device is once again operational, now with new functionality.
   - The switchover from isolation of the PCIe to full operation happens automatically.

6. Repeat step 4 and step 5 to move on to a new version (or return to ver1), this time starting with delivery of `ver2_tpcie_update_region_partial_clear.bin`. This would be followed by the next version of the user application, or a return to ver1.

Debugging Tandem with Field Updates Designs

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

Important Considerations

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

- When using Tandem PCIe, the stage 1 and stage 2 bitstreams must remain linked together. The stage 2 bitstream must always be the one created with the stage 1 bitstream delivered from flash. You cannot follow a stage 1 bitstream with a partial or clearing bitstream, or a stage 2 bitstream created from a different implementation result of a design even if Field Updates is selected.
- Always be sure that the partial and clearing bitstreams are compatible with the current static design in the FPGA before loading them. PR_Verify is a fundamental part of the Dynamic Function eXchange solution and must be used for Tandem with Field Updates for the same reason. PR_Verify confirms that multiple design configurations (that is, versions) are compatible with each other and therefore safe to overlay in hardware.
- The initial Tandem configuration of the device must be done with a bitstream set compiled as a version within a Tandem with Field Updates flow. If the initial bitstream load has been done with a standard Tandem bitstream set, it will not be compatible with later Field Update clearing or partial bitstreams. Contention could occur and device damage is possible.

Known Limitations

- Tandem PROM requires PERSIST to be set to reserve dual-purpose I/O for programming usage. Because of this, the configuration flash cannot be accessed by the user design after Tandem PROM configuration completes. Tandem PCIe should be used if configuration flash update from the FPGA user design is required.
- The use of Tandem PROM with PERSIST also disables the ICAP. This resource is not available post-configuration for DFX or other functions.

- Tandem with Field Updates is supported for UltraScale and UltraScale+ devices only. This solution will never support 7 series devices. For this type of approach in 7 series, a general Dynamic Function eXchange solution should be considered.
- General Tandem + Dynamic Function eXchange is also supported. To enable, simply use the standard **Tandem** option when generating the PCIe IP then add it to a DFX design to be compiled in non-project mode as described in Non-Project Flow, and the *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909).
- Bitstream compression, which is enabled by default for Tandem Configuration solutions, is not compatible with the Dynamic Function eXchange per-frame CRC feature. If per-frame CRC checking is desired for any of the "update" partial bitstreams, rerun bitstream generation with that feature enabled and bitstream compression disabled. The `write_bitstream -cell` option can be used to create only the partial and clearing bitstreams needed for each design image.

Using Tandem With a User Hardware Design

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

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

Method 1 – Using the Existing PCI Express Example Design

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

1. Create the example design.

   Generate the example design as described in the Tandem PROM VCU108 Example Tool Flow and Tandem PCIe VCU108 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 VCU108 Example Tool Flow and Tandem PCIe VCU108 Example Tool Flow.

2. Migrate the clock module.

   If the Include Shared Logic (Clocking) in the example design option is set in the Shared Logic tab during core generation, then the `pipe_clock_i` clock module is instantiated in the top level of the example design. This clock module should be migrated to the user design to provide the necessary PCIe clocks.

   Note: These clocks can be used in other parts of the user design if desired.

3. Migrate the top-level constraint.

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

4. Migrate the top-level Pblock constraint.

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

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

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

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

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

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

7. Implement the design as normal.

Tandem Configuration RTL Design

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

MUXing Critical Inputs

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

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

TLP Requests

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

Tandem Configuration Logic

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

Table 30: **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.
mcap_eos_out	Output	Active-High	Pass through output from the End of Startup (EOS) pin on the STARTUP primitive.

Table 30: **Handshaking Ports** *(cont'd)*

Name	Direction	Polarity	Description
cap_req	Output	Active-High	Configuration Access Port arbitration request signal. This signal should be used to arbitrate the use of the FPGA configuration logic between multiple user implemented configuration interfaces. If the Media Configuration Access Port (MCAP) is the only user implemented configuration interface used, this signal should remain unconnected.
cap_rel	Input	Active-High	Configuration Access Port arbitration request for release signal. This signal should be used to arbitrate the use of the FPGA configuration logic between multiple user implemented configuration interfaces. If the MCAP is the only user implemented configuration interface used, this signal should be tied Low (1'b0). This allows the MCAP access to the FPGA configuration logic as needed.
cap_gnt	Input	Active-High	Configuration Access Port arbitration grant signal. This signal should be used to arbitrate the use of the FPGA configuration logic between multiple user implemented configuration interfaces. If the MCAP is the only user implemented configuration interface used, this signal tied High (1'b1). This grants the MCAP access to the FPGA configuration logic upon request.
user_reset	Output	Active-High	Can be used to reset PCIe interfacing logic when the PCIe core is reset. Synchronized with `user_clock`.
user_clk	Output	N/A	Clock to be used by PCIe interfacing logic.
user_lnk_up	Output	Active-High	Identifies that the 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).
- `mcap_eos_out` is a pass through output of the EOS signal from the STARTUP primitive. This output is deasserted until stage1 is loaded, asserted between stage 1 and stage 2, and asserted again at the end of stage 2. The `FPGA DONE` pin also shows similar behavior when loading Tandem bitstreams.
- `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. The Tandem solution uses the STARTUP block and the dedicated End Of Startup (EOS) signal to detect the completion of stage 2 programming and then switch control of the PCI Express Block to the user application. When this switch occurs, `mcap_design_switch` is asserted.

If the STARTUP block is required for other functionality within your design, generate the IP with the STARTUP primitive external to the IP and connect the EOS output to the IP `mcap_eos_in` input within your design. To generate the STARTUP primitive external to the IP, select the **Use an external STARTUP primitive** option when customizing the core in the Vivado IDE.

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 know 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. Because of this, the configuration flash cannot be accessed by the user design after Tandem PROM configuration completes. Tandem PCIe should be used if configuration flash update from the FPGA user design is required.

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

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

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

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

Avoiding the Configuration Bank

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

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

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

PROM Selection

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

1. Supported by AMD 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) 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) 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).

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 use the dedicated reset routing and dedicated PCIe reset package pin (PERSTN0). This selection is enabled by default where applicable. If your system design does not allow for the use of this dedicated reset, you must disable the use of the dedicated PERST routing resources in the Vivado IDE. When selecting a new location for the reset pin, 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 VCU108 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 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.

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

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

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

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

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

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

   The following is a snippet from the `write_bitstream` log showing the bitstream size for stage 1 in a VU095 device:

   ```
   Creating bitstream...

   Tandem stage1 bitstream contains 9175424 bits.

   Tandem stage2 bitstream contains 277708576 bits.

   Writing bitstream ./xilinx_pcie_ip.bit...
   ```

4. 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 = 9175424 bits = 8.75 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:

   8.75 Mb / 263.92 Mb/s = ~0.0332 s or 33.2 ms

Example 2

The configuration for Example 2 is:

- BPI (x16) Synchronous mode, operating at 50 MHz ± 100 ppm
- Stage 1 size = 9175424 bits = 8.75 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:

   8.75 Mb / 799.92 Mb/s = ~0.0109 s or 10.9 ms

Using Bitstream Compression

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

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

Other Bitstream Load Time Considerations

Bitstream configuration times can also be affected by:

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

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

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

T_{POR} is 57 ms for standard power ramp rates, and 20 ms for fast ramp rates for UltraScale devices. See *Kintex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics* (DS892), and *Virtex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics* (DS893).

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}) + 33.2 ms (bitstream time) + 2 ms (ramp time) = 85.2 ms

85.2 ms < 100 ms PCIe standard (okay)

In this case, the margin is 14.8 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}) + 33.2 ms (bitstream time) + 2 ms (ramp time) - 100 ms = -14.8 ms

-14.8 ms < 100 ms PCIe standard (okay)

In this case, the margin is 114.8 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.
- **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 31: **Example Bitstream Size and Configuration Times**[1]

Device	Full Bitstream	Full: BPI16 at 50 MHz	Tandem Stage 1[2]	Tandem: BPI16 at 50 MHz
KU040	122.1 Mb	152.7 ms	7.6 Mb	9.5 ms
VU095	273.5 Mb	341.8 ms	8.8 Mb	10.9 ms
VU190	577.1 Mb	721.4 ms	11.2 Mb	14.1 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 Gen3x8 configuration in Vivado Design Suite 2015.1.

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

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

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

Clocking

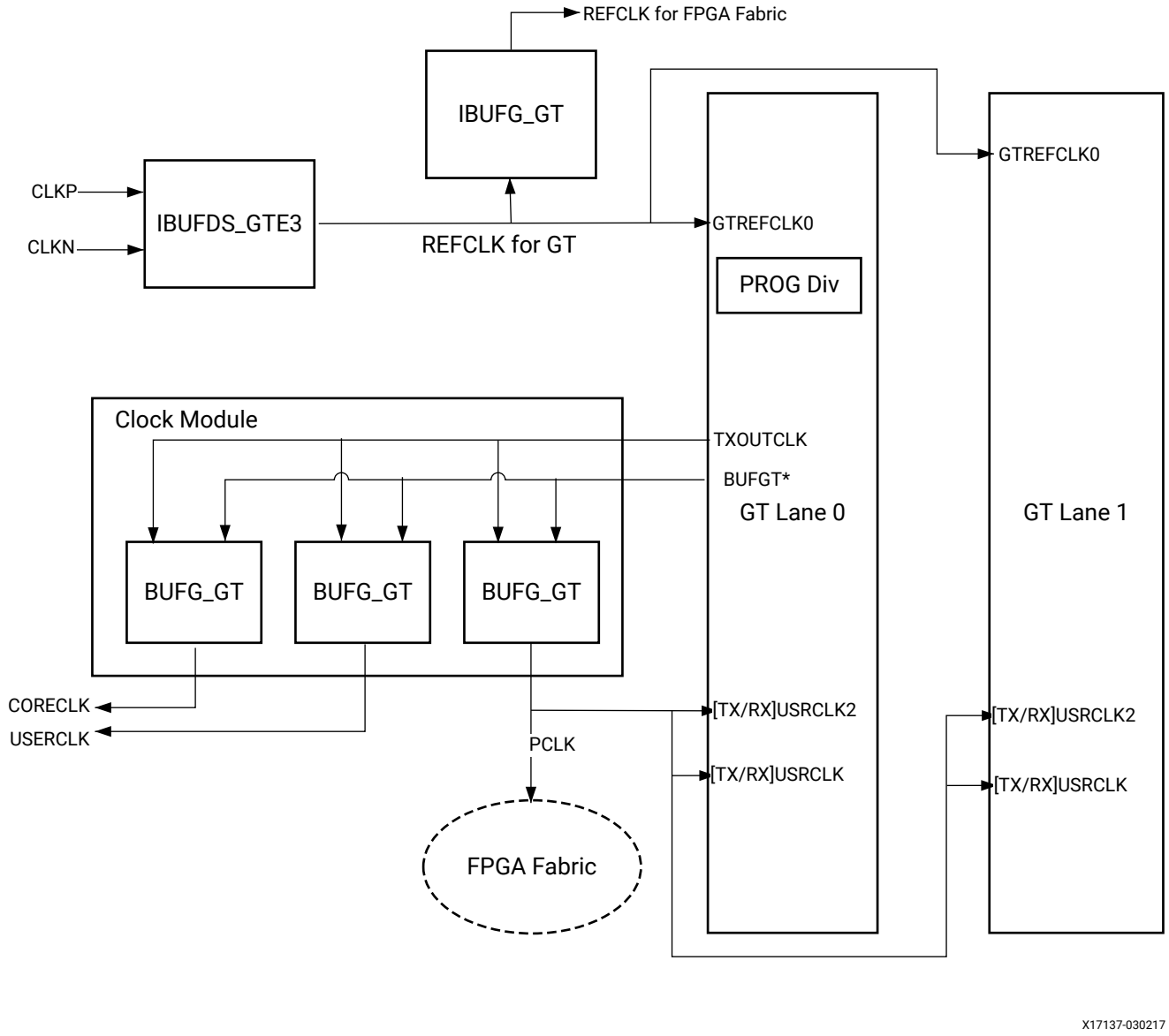
The UltraScale Devices Gen3 Integrated Block for PCIe core can be configured to use 100 MHz, 125 MHz, or 250 MHz reference clock. For more information, see the Answer Records at the AMD PCI Express Solution Center.

The following applies:

- The reference clock can be synchronous or asynchronous with up to ±300 PPM or 600 PPM worst case. (If spread spectrum clock (SSC) is enabled, the link must be synchronous.)
- The PCLK is the primary clock for the PIPE interface.
- In addition to PCLK, two other clocks (CORECLK and USERCLK) are required to support the core.
- BUFG_GTs are used to generate these clocks. These clocks are all driven from the `TXOUTCLK` pin which is a derived clock from `GTREFCLK0` through a CPLL. In an application where QPLL is used, QPLL is only provided to the GT PCS/PMA block while `TXOUTCLK` continues to be derived from a CPLL.
- The source of the UltraScale GTH reference clock must come directly from IBUFDS_GTE3.
- To use the reference clock for FPGA general interconnect, another BUFG_GT must be used.

The following figure shows an x2 PCIe clocking architecture example.

Figure 10: **Clocking**



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

IMPORTANT! *All add-in card designs must use synchronous clocking due to 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. The following figures show a system using a 100 MHz reference clock. Even if the device is part of an embedded system, if the system uses commercial PCI Express root complexes or switches along with typical motherboard clocking schemes, synchronous clocking should still be used.

***Note*:** The following figures 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).

Figure 11: **Embedded System Using 100 MHz Reference Clock**

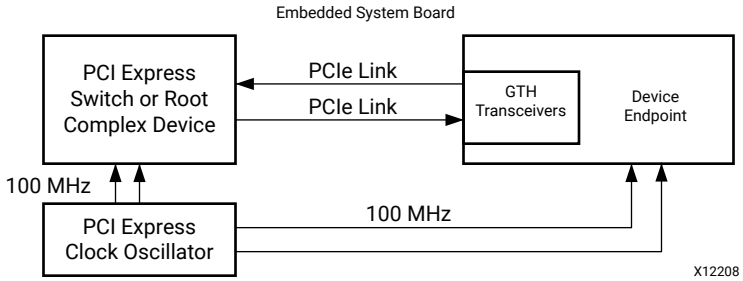
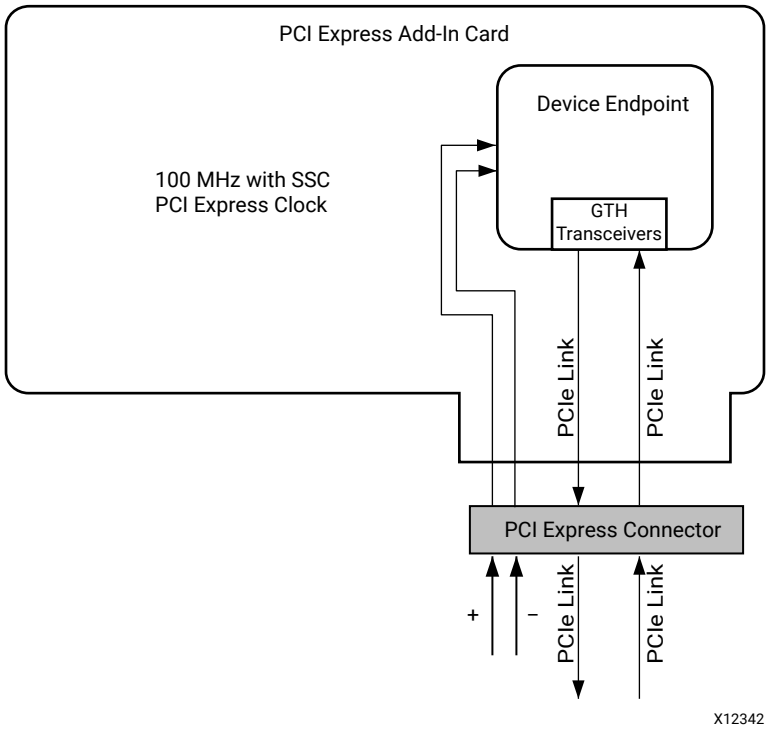


Figure 12: **Open System Add-In Card Using 100 MHz Reference Clock**



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.

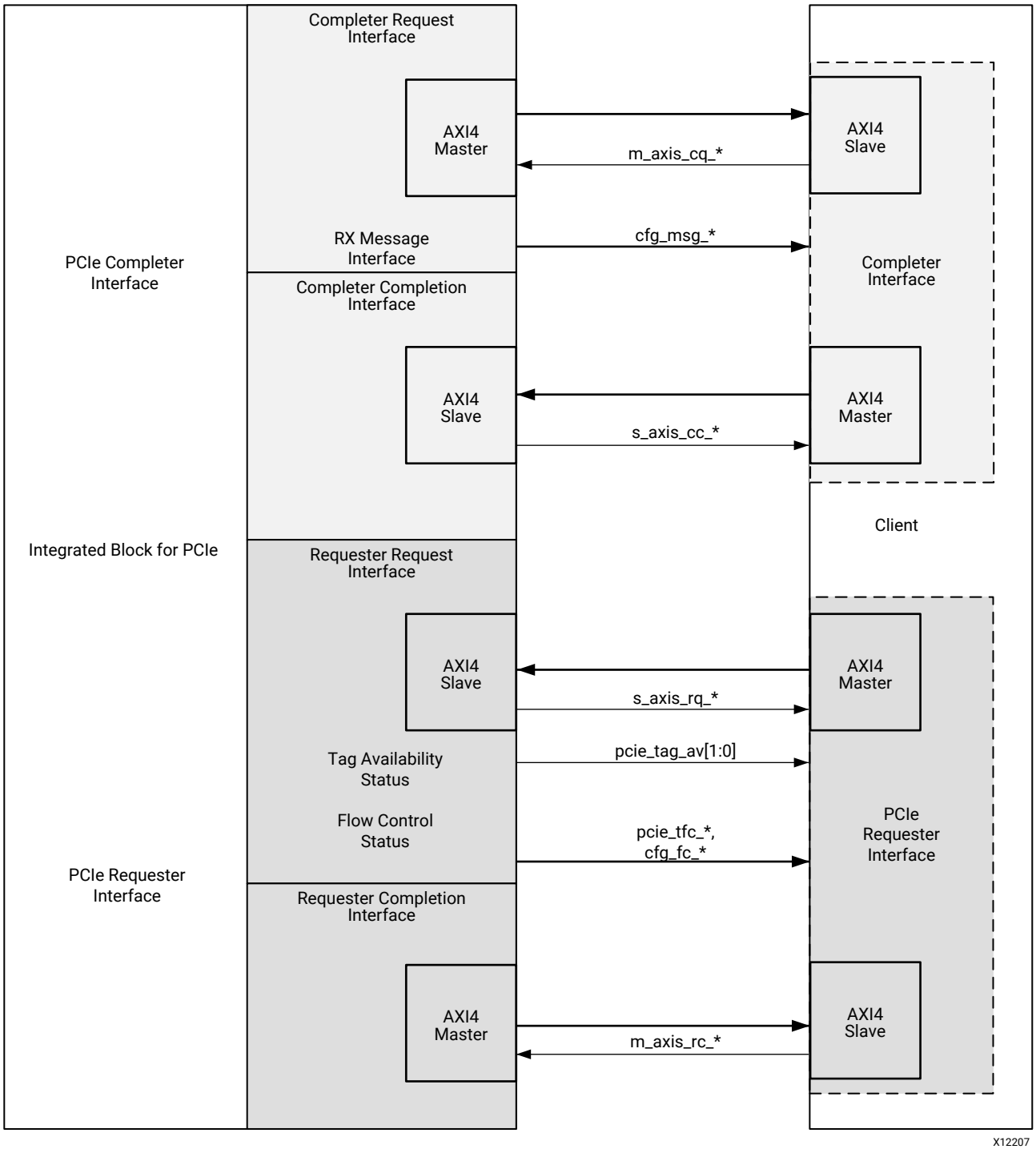
AXI4-Stream Interface Description

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

Feature Overview

The following figure illustrates the user interface of the core.

Figure 13: **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. The width of these interfaces can be configured as 64, 128, or 256 bytes, 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.

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

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

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

Table 32: **Data Width and Clock Frequency Settings for the User Interfaces**

PCI Express Generation/Maximum Link Speed	Maximum Link Width Capability	AXI4-Stream Interface Width	User Clock Frequency (MHz)
Gen1 (2.5 GT/s)	x1	64 bits	62.5, 125, or 250
	x2	64 bits	62.5, 125, or 250
	x4	64 bits	125, or 250
	x8	64 bits	250
		128 bits	125
Gen2 (5.0 GT/s)\	x1	64 bits	62.5, 125, or 250
	x2	64 bits	125, or 250
	x4	64 bits	250
		128 bits	125
	x8	128 bits	250
		256 bits	125
Gen3 (8.0 GT/s)	x1	64 bits	125, or 250
	x2	64 bits	250
		128 bits	125
	x4	128 bits	250
		256 bits	125
	x8	256 bits	250

Notes:

1. 250 MHz user clock frequency is not supported for -1LV speed grade, non-x8 configurations when AXI4-Stream width 64-bit is selected.

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.

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.

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 Completer Interface for more details on address alignment and example diagrams.

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 might 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 bits, depending on the type of TLP and Payload size, there can be significant interface utilization inefficiencies, if a maximum of 1 TLP 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 66: Transfer of Completion TLPs on the Requester Completion Interface with the Straddle Option Enabled for waveforms showing this option.

The straddle option, available only on the 256-bit wide RC interface, is enabled through the Vivado IP catalog. See Chapter 5: Design Flow Steps for instructions on enabling the option in the IP catalog. When this option is enabled, the integrated block can start a new Completion TLP on byte lane 16 when the previous TLP has ended at or before byte lane 15 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, and the RC interface is set to Dword-aligned mode.

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

Table 33: **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

Receive Transaction Ordering

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

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

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

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

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

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

Table 34: **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.

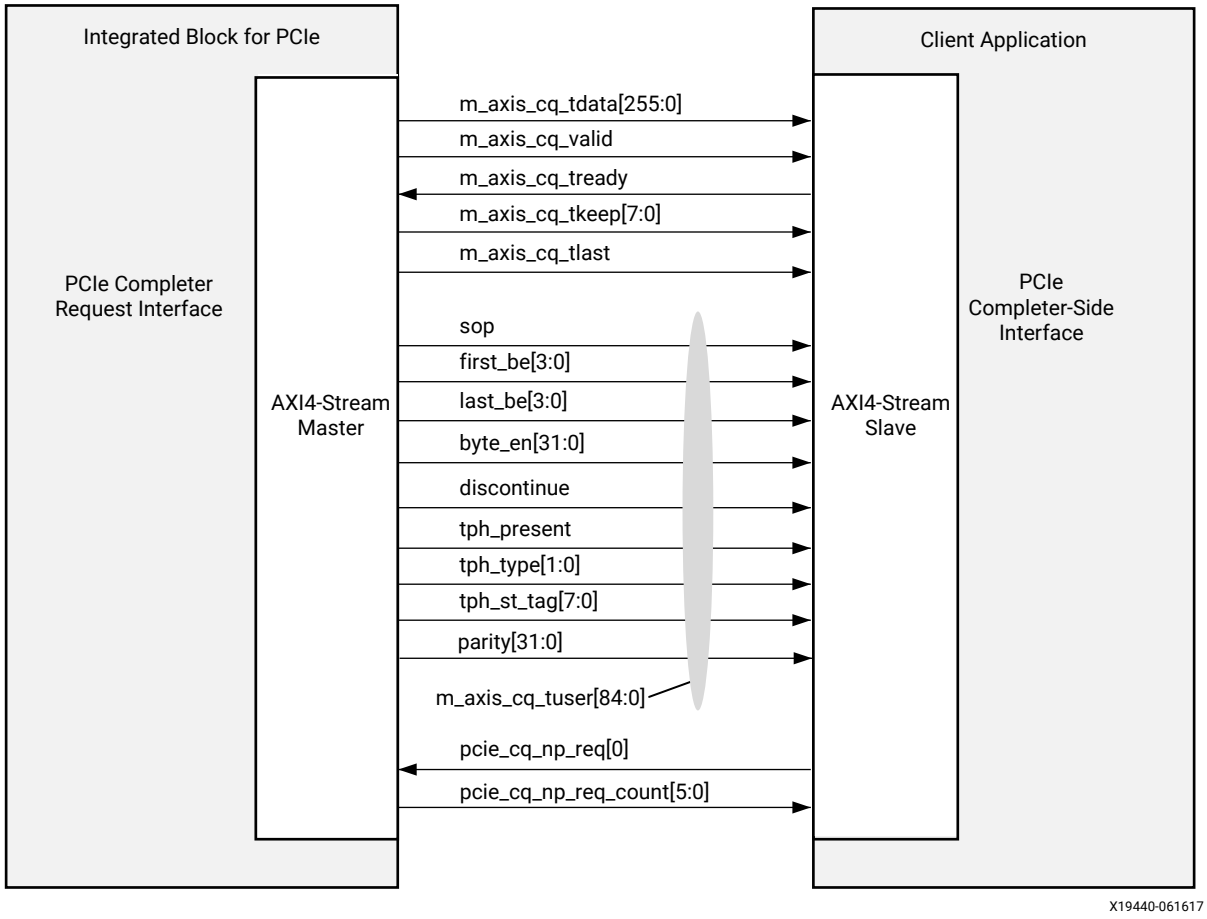
Completer Interface

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

Completer Request Interface Operation

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

Figure 14: **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 the following figures. The format of the following figure applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request. The format of Figure 16: Completer Request Descriptor Format for Vendor-Defined Messagesis used for Vendor-Defined Messages (Type 0 or Type 1) only. The format of Figure 17: Completer Request Descriptor Format for ATS Messages 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 18: Completer Request Descriptor Format for All Other Messages.

Figure 15: **Completer Request Descriptor Format for Memory, I/O, and Atomic Op Requests**

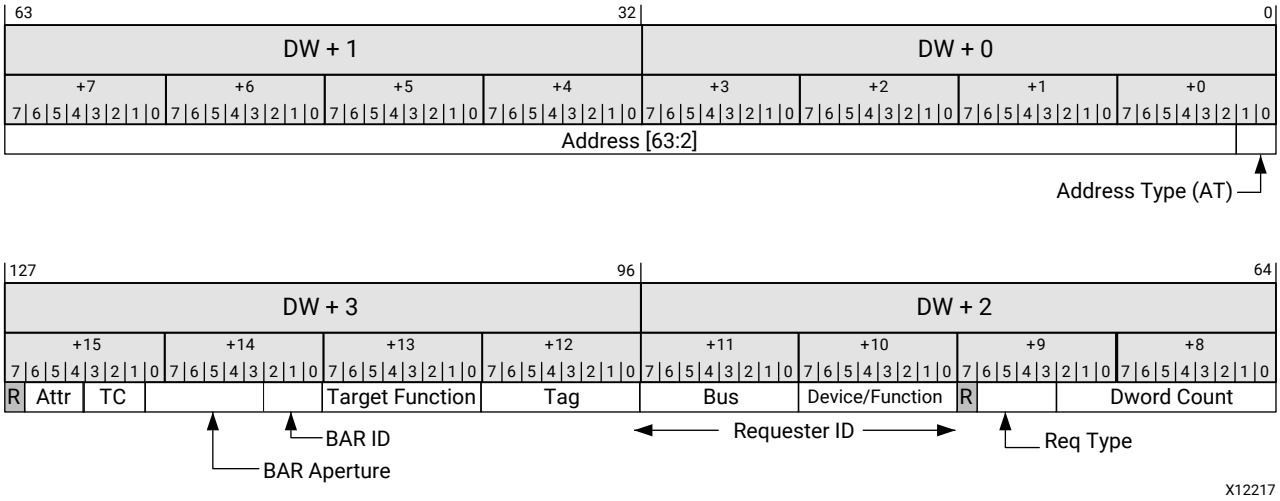


Figure 16: **Completer Request Descriptor Format for Vendor-Defined Messages**

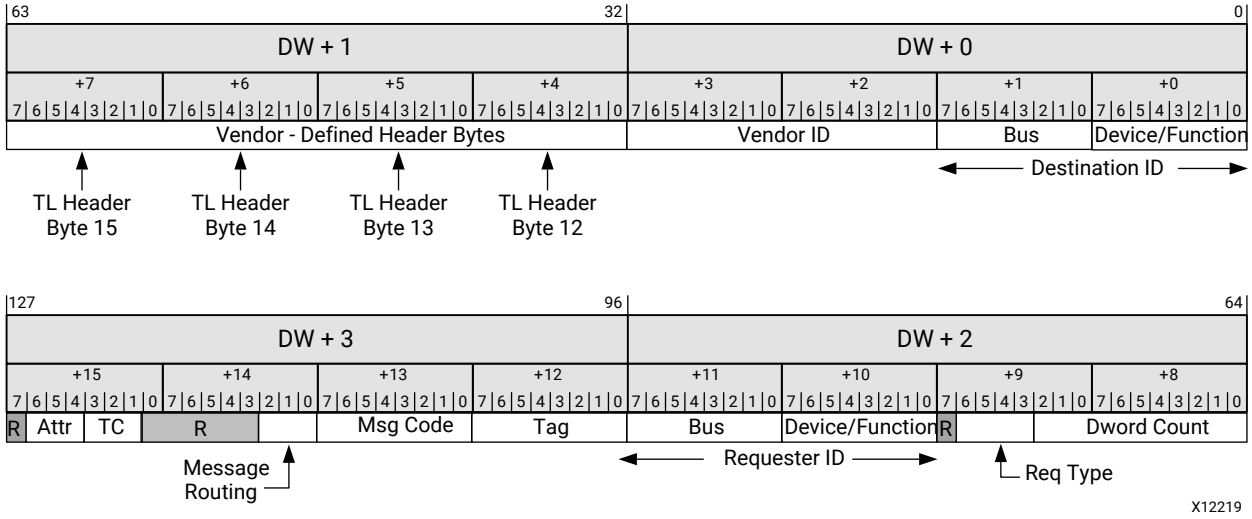


Figure 17: **Completer Request Descriptor Format for ATS Messages**

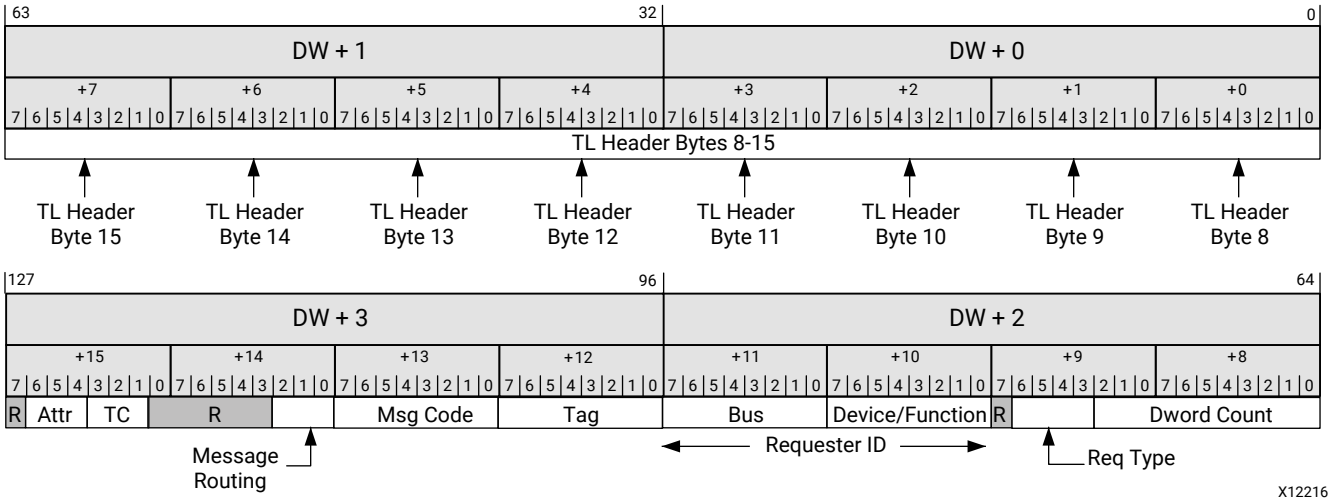
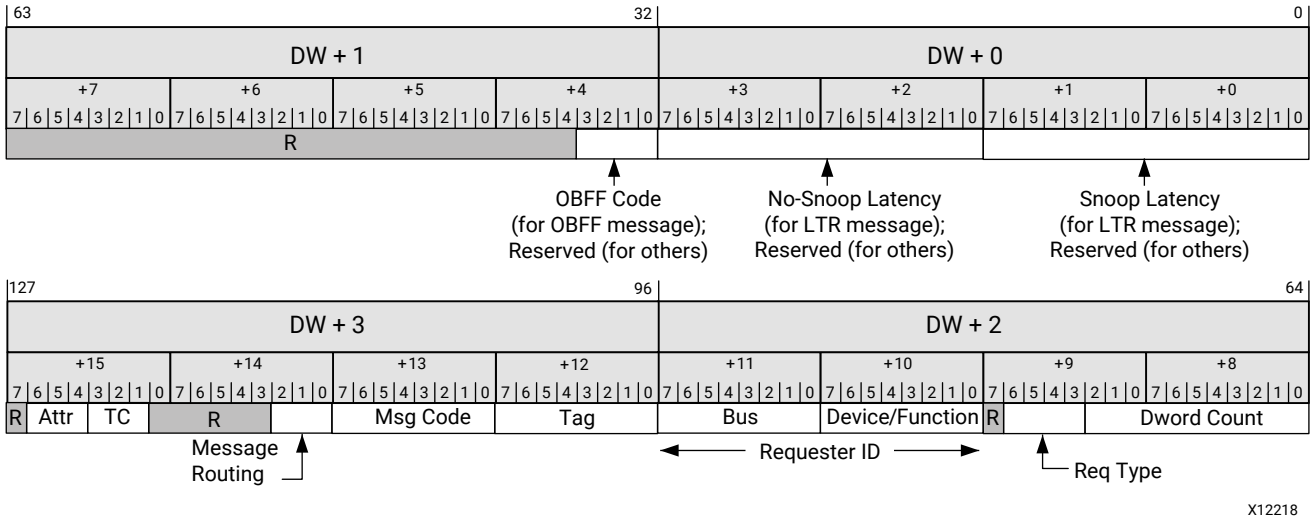


Figure 18: **Completer Request Descriptor Format for All Other Messages**



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

Table 35: **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
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 the following table.
95:80	Requester ID	PCI Requester ID associated with the request. With legacy interpretation of RIDs, these 16 bits are divided into an 8-bit bus number [95:88], 5-bit device number [87:83], and 3-bit Function number [82:80]. When ARI is enabled, bits [95:88] carry the 8-bit bus number and [87:80] provide the Function number. When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the integrated block with the completion data.

Table 35: **Completer Request Descriptor Fields** *(cont'd)*

Bit Index	Field Name	Description
103:96	Tag	PCIe Tag associated with the request. When the request is a Non-Posted transaction, the user logic must store this field and supply it back to the integrated block with the completion data. This field can be ignored for memory writes and messages.
111:104	Target Function	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
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). • 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 • 111: No BAR Check (Valid for Root Port only) 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.

Table 35: **Completer Request Descriptor Fields** *(cont'd)*

Bit Index	Field Name	Description
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.
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 provides a complete list of the supported Message Codes.
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 36: **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

Table 36: **Transaction Types** *(cont'd)*

Request Type (binary)	Description
1110	ATS Message
1111	Reserved

Completer Memory Write Operation

The following timing diagrams illustrate the Dword-aligned transfer of a memory write TLP received from the link across the Completer reQuest (CQ) interface, when the interface width is configured as 64, 128, and 256 bits, respectively. For illustration purposes, the starting Dword address of the data block being written into memory is assumed to be (m × 32 + 1), for an integer m > 0. Its size is assumed to be n Dwords, for some n = k × 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 `s_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 19: **Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode, 64-Bit Interface)**

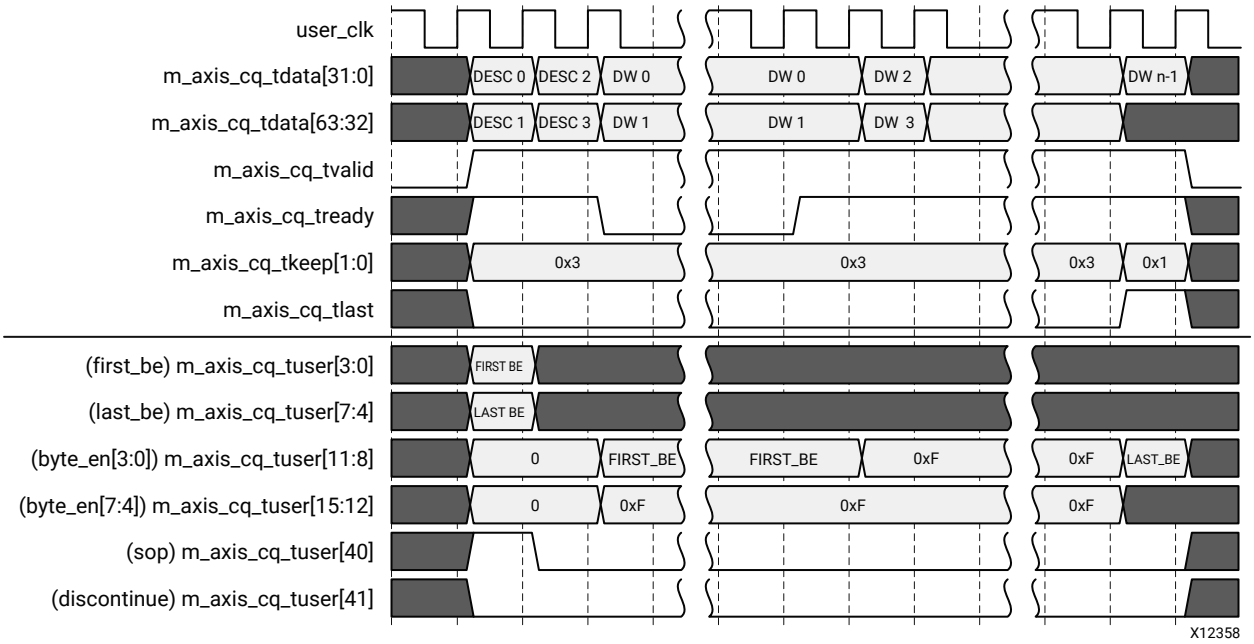


Figure 20: **Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode, 128-Bit Interface)**

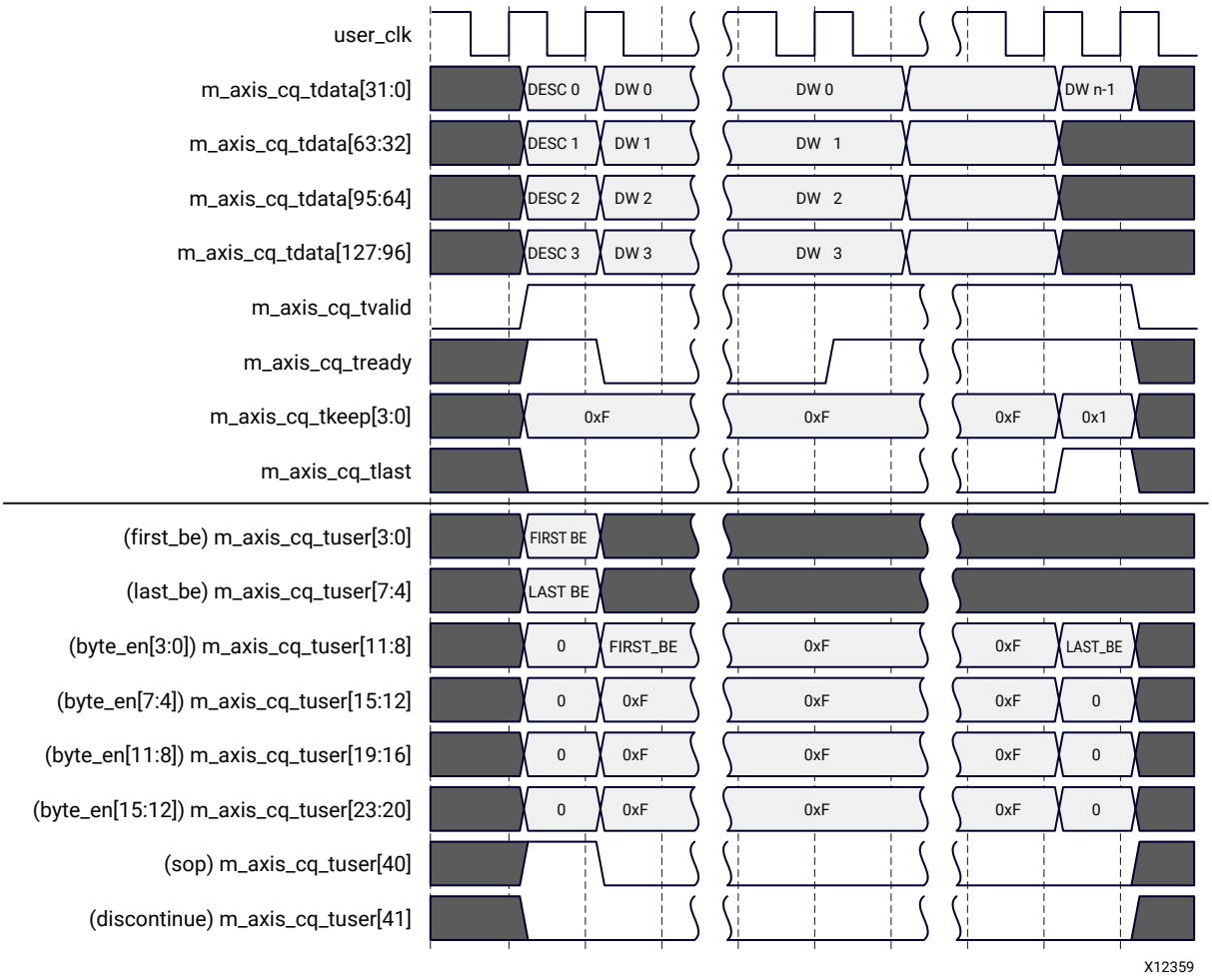
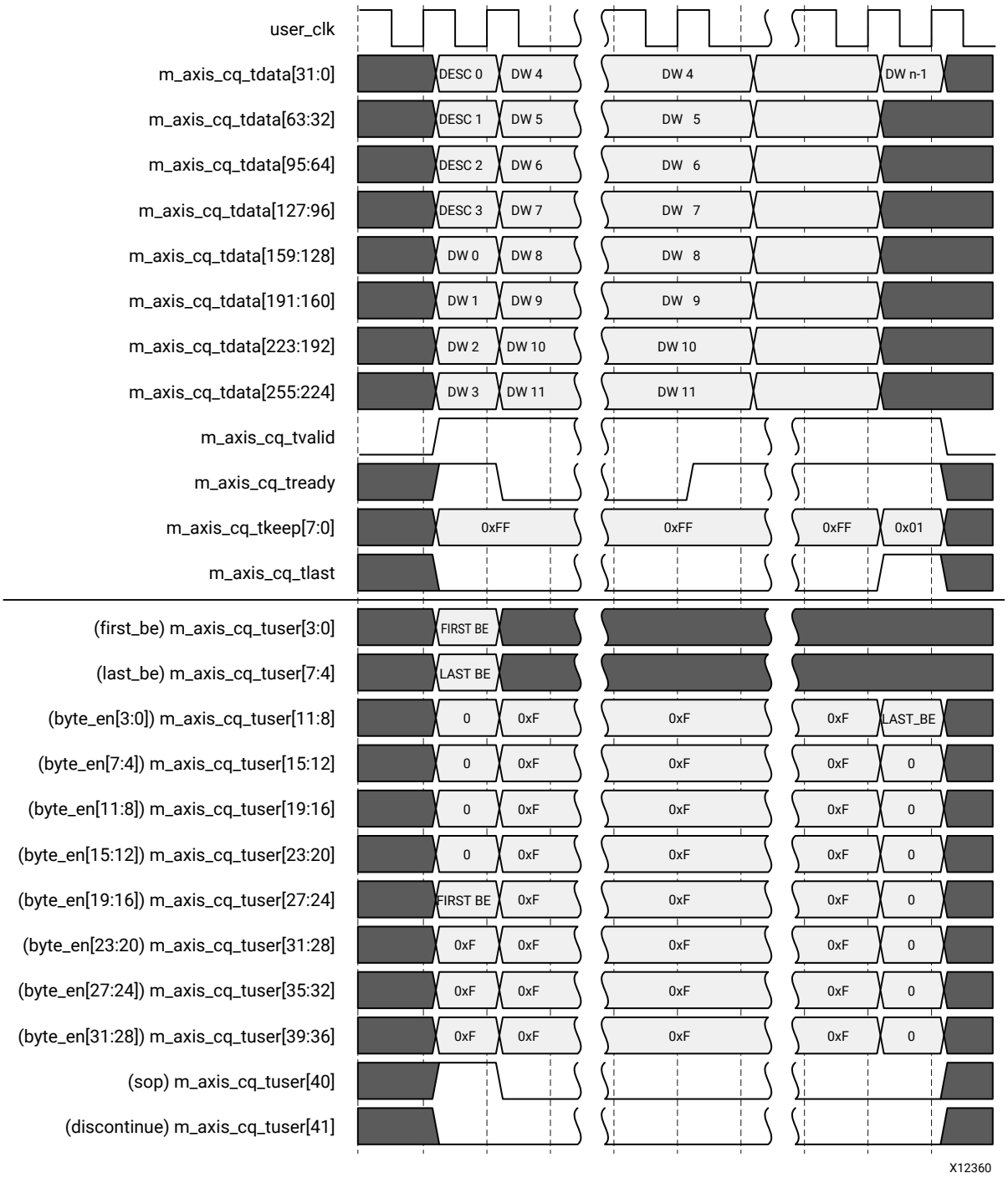


Figure 21: **Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode, 256-Bit Interface)**



The following timing diagrams illustrate the address-aligned transfer of a memory write TLP received from the link across the CQ interface, when the interface width is configured as 64, 128 and 256 bits, respectively. For the purpose of illustration, the starting Dword address of the data block being written into memory is assumed to be (m × 32 + 1), for an integer m > 0. Its size is assumed to be n Dwords, for some n = k × 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 22: **Memory Write Transaction on the Completer Request Interface (Address-Aligned Mode, 64-Bit Interface)**

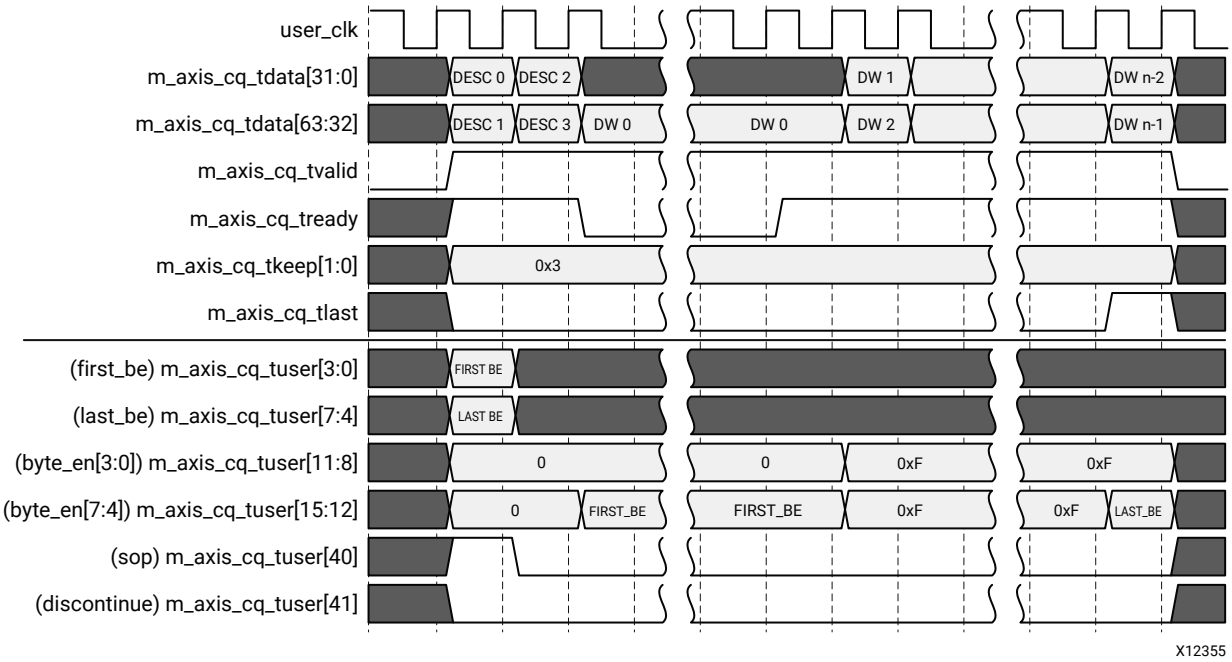


Figure 23: **Memory Write Transaction on the Completer Request Interface (Address-Aligned Mode, 128-Bit Interface)**

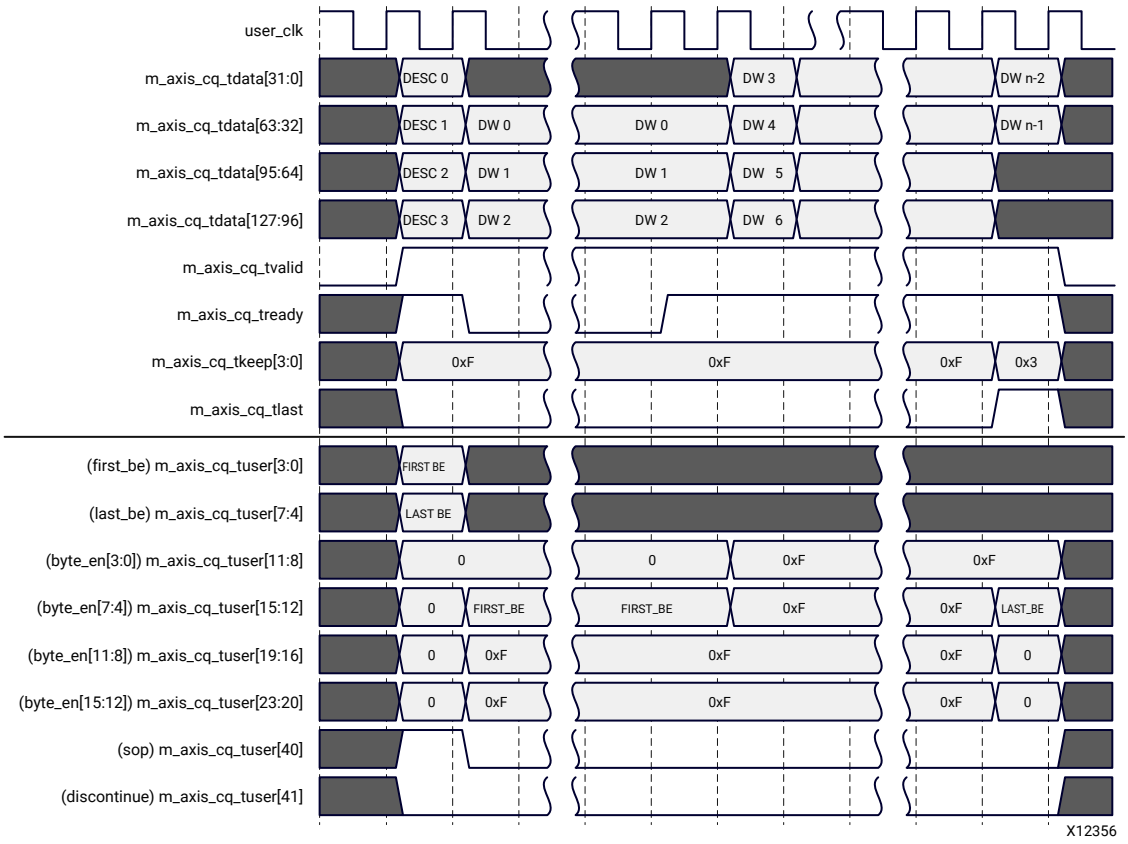
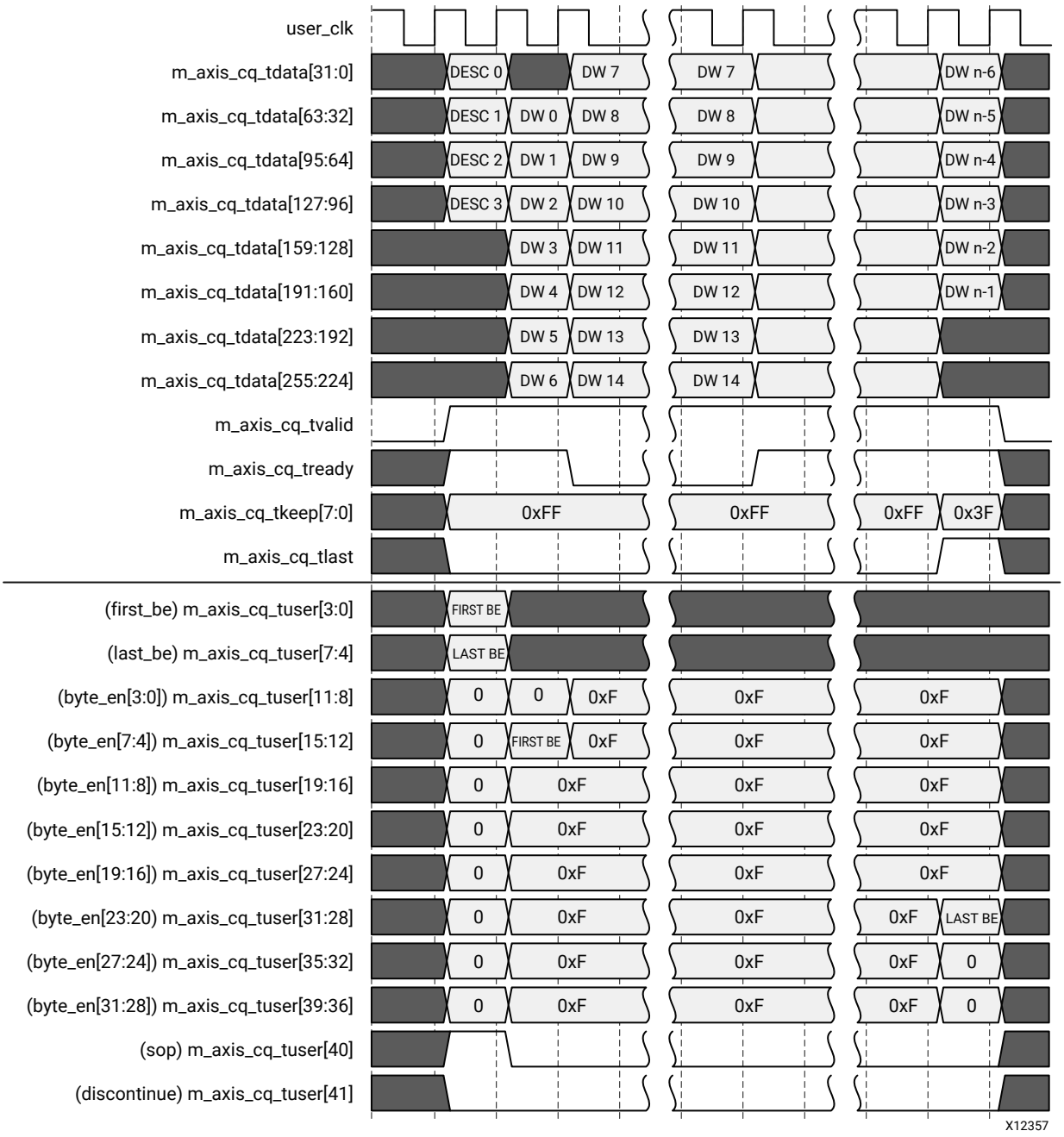


Figure 24: **Memory Write Transaction on the Completer Request Interface (Address-Aligned Mode, 256-Bit Interface)**



Completer Memory Read Operation

A memory read request is transferred across the completer request interface in the same manner as a memory write request, except that the AXI4-Stream packet contains only the 16-byte descriptor. The following timing diagrams illustrate the transfer of a memory read TLP received from the link across the completer request interface, when the interface width is configured as 64, 128, and 256 bits, respectively. The packet occupies two consecutive beats on the 64-bit interface, while it is transferred in a single beat on the 128-bit 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 25: **Memory Read Transaction on the Completer Request Interface (64-Bit Interface)**

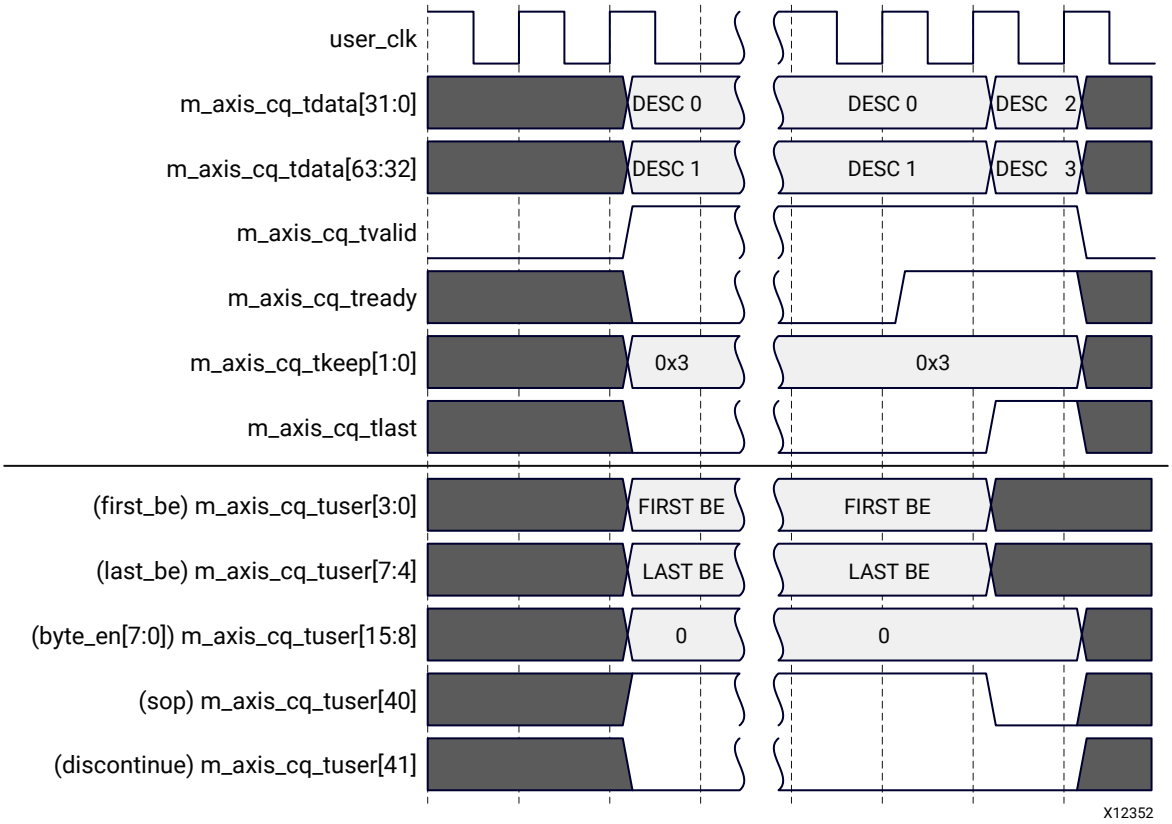


Figure 26: **Memory Read Transaction on the Completer Request Interface (128-Bit Interface)**

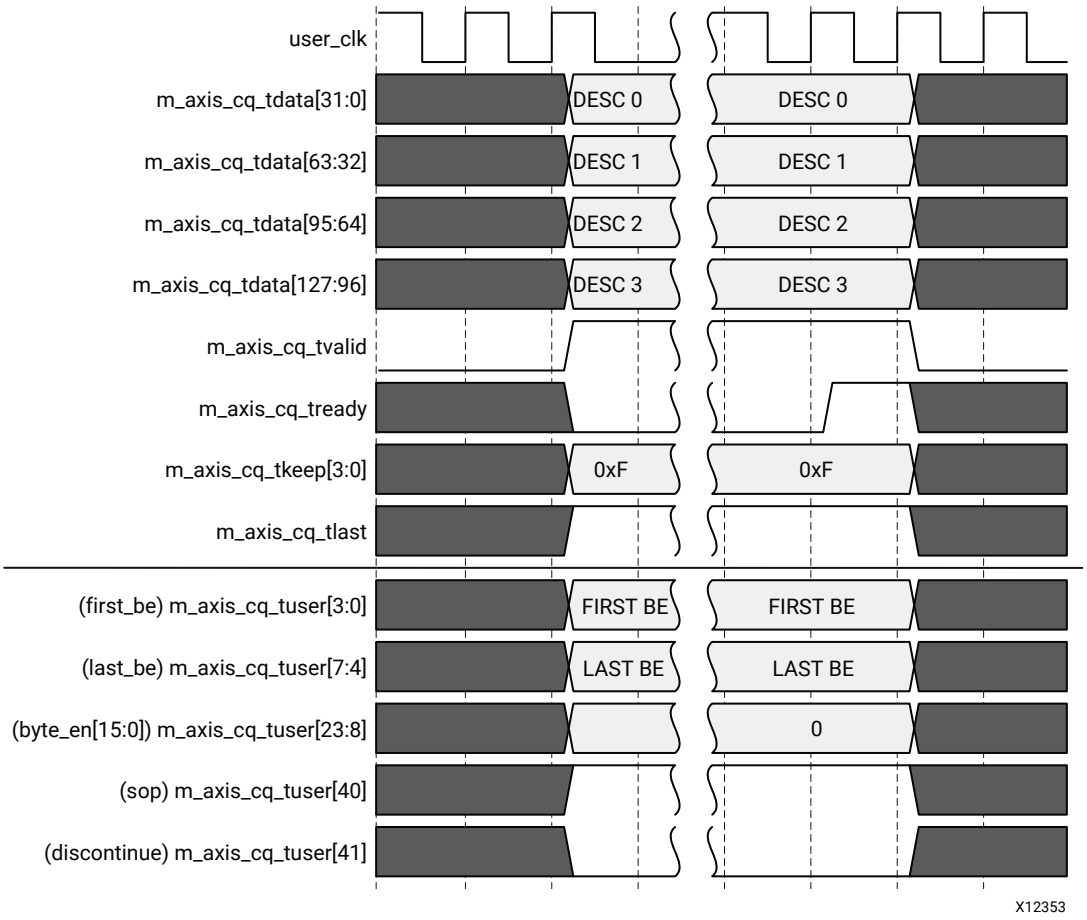
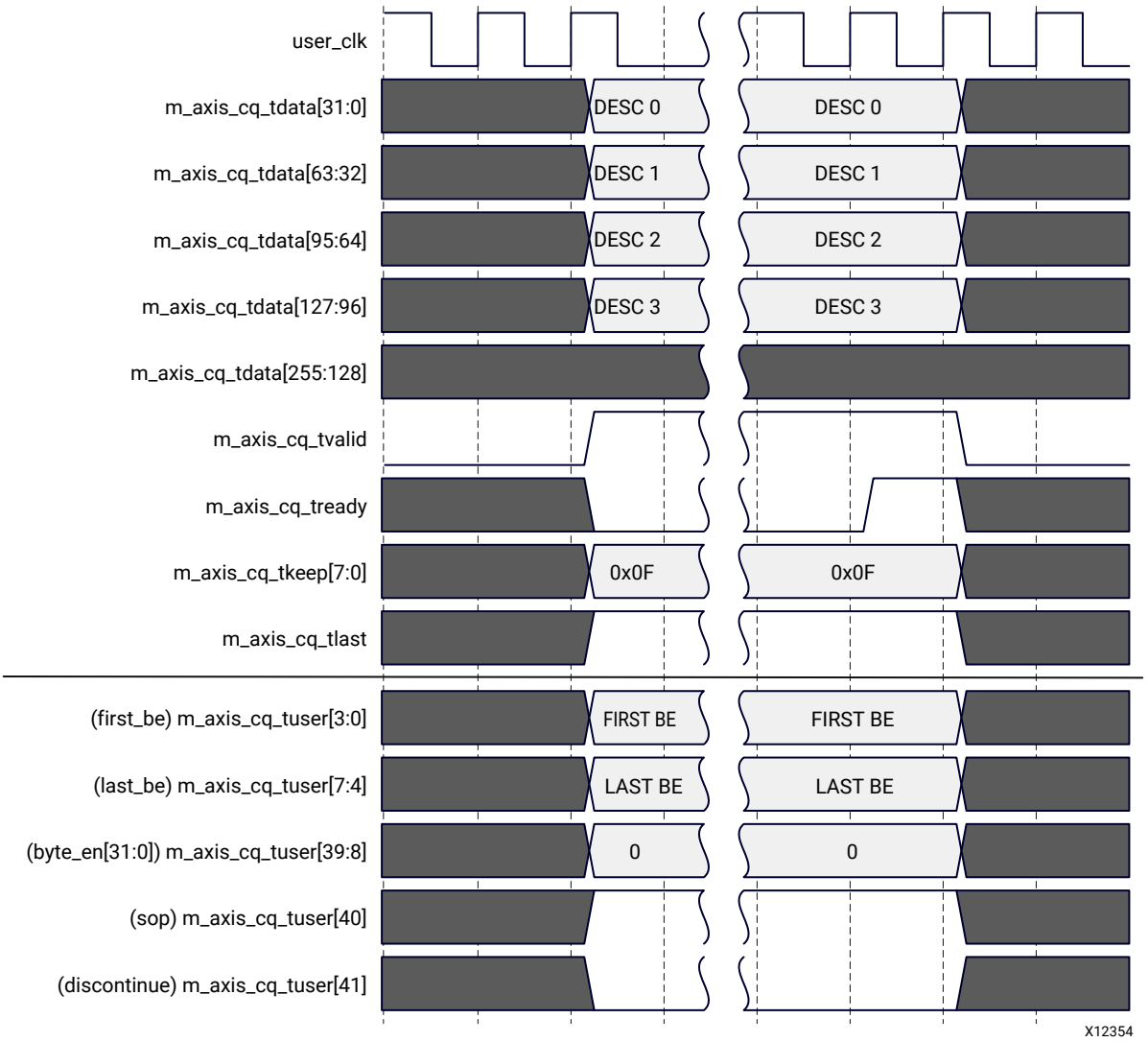


Figure 27: **Memory Read Transaction on the Completer Request Interface (256-Bit Interface)**



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

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

I/O Write Operation

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

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

I/O Read Operation

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

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

Atomic Operations on the Completer Request Interface

The transfer of an Atomic Op request on the completer request interface is similar to that of a memory write request. The payload for an Atomic Op can range from one Dword to eight 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.

Message Requests on the Completer Request Interface

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

Aborting a Transfer

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

Selective Flow Control for Non-Posted Requests

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

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

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

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

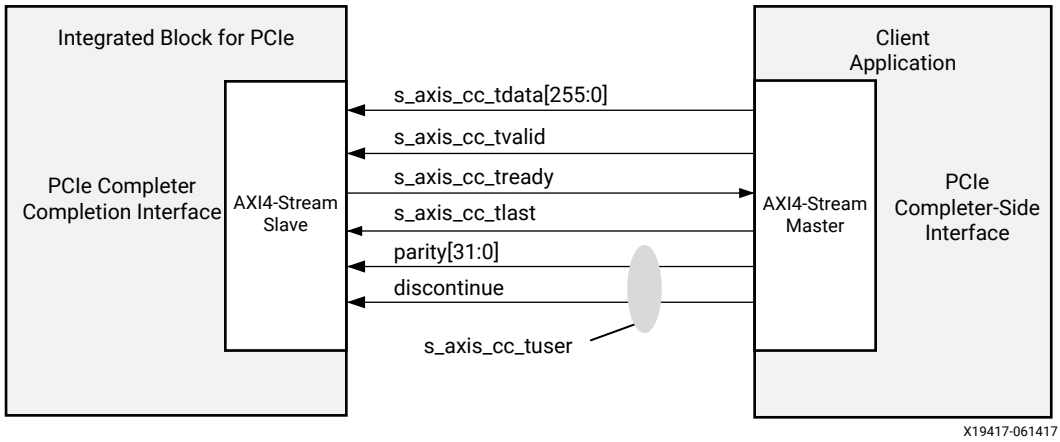
The assertion and deassertion of the `pcie_cq_np_req[0]` signal does not need to be aligned with the packet transfers on the completer request interface.

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

Completer Completion Interface Operation

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

Figure 28: **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 $n = (12 + A \bmod 4) \bmod w$, where

$$A$$

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

Completer Completion Descriptor Format

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

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

Figure 29: **Completer Completion Descriptor Format**

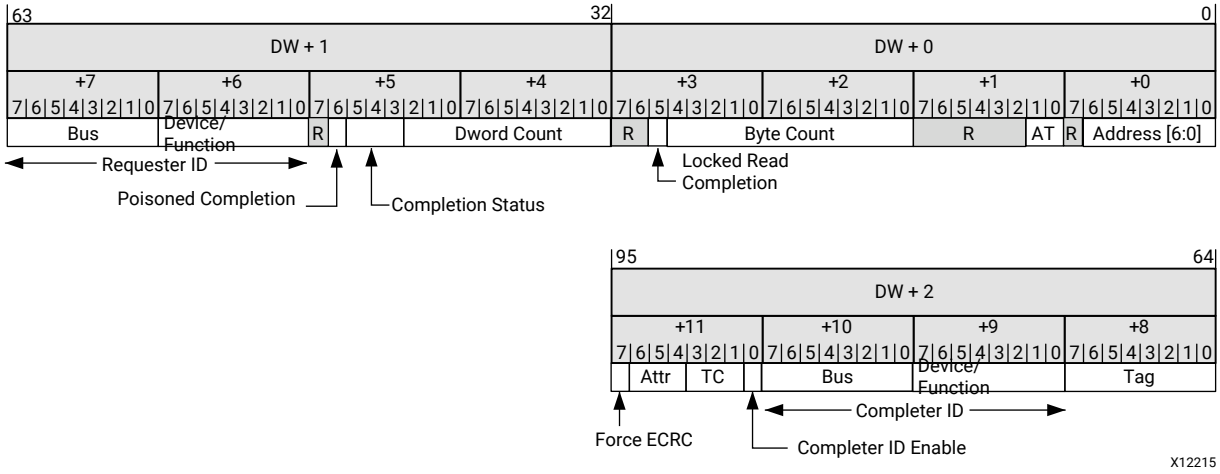


Table 37: **Completer Completion Descriptor Fields**

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. *Table 37:* **Completer Completion Descriptor Fields** 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.

Table 37: **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 the following table. MSB of the Byte Count field is reserved.
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 37: **Completer Completion Descriptor Fields** *(cont'd)*

Bit Index	Field Name	Description
79:72	Target Function/ Device Number	Device and/or Function number of the Completer Function. Endpoint mode: • ARI enabled: ◦ Bits [87:80] must be set to the Completer Function number. • ARI disabled: ◦ Bits [82:80] must be set to the Completer Function number. ◦ Bits [87:83] are not used Upstream Port for Switch use case (Endpoint mode is selected within the IP): • ARI enabled: ◦ Bits [87:80] must be set to the Completer Function number. • ARI disabled: ◦ Bits [82:80] must be set to the Completer Function number. ◦ Bits [87:83] are not used if the Completion is originating from the switch itself. These bits must be set to the Completer Device number where the Completion was originated if the switch is relaying the Completion (Completer is external to the switch). This is used with Completer ID Enable bit in the descriptor. Root Port mode (Downstream Port): • ARI enabled: ◦ Bits [87:80] must be set to the Completer Function number. • ARI disabled: ◦ Bits [82:80] must be set to the Completer Function number. ◦ Bits [87:83] must be set to the Completer Device number. This is used with Completer ID Enable bit in the descriptor.
87:80	Completer Bus Number	Bus number associated with the Completer Function. Endpoint mode: • Not used Upstream Port for Switch use case (Endpoint mode is selected within the IP): • Not used if the Completion is originating from the switch itself. These bits must be set to the Completer Bus number where the Completion was originated if the switch is relaying the Completion (Completer is external to the switch). This is used with Completer ID Enable bit in the descriptor. Root Port mode (Downstream Port): • Must be set to the Completer Bus number. This is used with Completer ID Enable bit in the descriptor.

Table 37: **Completer Completion Descriptor Fields** *(cont'd)*

Bit Index	Field Name	Description
88	Completer ID Enable	1'b1: The client supplies Bus, Device, and Function numbers in the descriptor to be populated as the Completer ID field in the TLP header. 1'b0: IP uses Bus and Device numbers captured from received Configuration requests and the client supplies Function numbers in the descriptor to be populated as the Completer ID field in the TLP header. Endpoint mode: • Must be set to 1'b0. Upstream Port for Switch use case (Endpoint mode is selected within the IP): • Set to 1'b0 when the Completion is originating from the switch itself. • Set to 1'b1 when the switch is relaying the Completion (Completer is external to the switch). This is used with Completer Bus Number bits [95:88] and Completer Function/Device Number bits [87:83] when ARI is not enabled. Root Port mode: • Must be set to 1'b1. This is used with Completer Bus Number bits [95:88] and Completer Function/Device Number bits [87:83] when ARI is not enabled.
91:89	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request. The user 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 38: **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

Table 38: **Calculating Byte Count from Completer Request first_be[3:0], last_be[3:0], Dword Count[10:0]** *(cont'd)*

first_be[3:0]	last_be[3:0]	Total Byte Count
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 following timing diagrams illustrate the Dword-aligned transfer of a Completion from the user application across the CC interface, when the interface width is configured as 64, 128, and 256 bits, respectively. In this case, the first Dword of the payload starts immediately after the descriptor. When the data block is not a multiple of four bytes, or when the start of the payload is not aligned on a Dword address boundary, the user application must add null bytes to align the start of the payload on a Dword boundary and make the payload a multiple of Dwords. For example, when the data block starts at byte address 7 and has a size of 3 bytes, the user application must add three null bytes before the first byte and two null bytes at the end of the block to make it two 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 30: **Transfer of a Normal Completion on the Completer Completion Interface (Dword-Aligned Mode, 64-Bit Interface)**

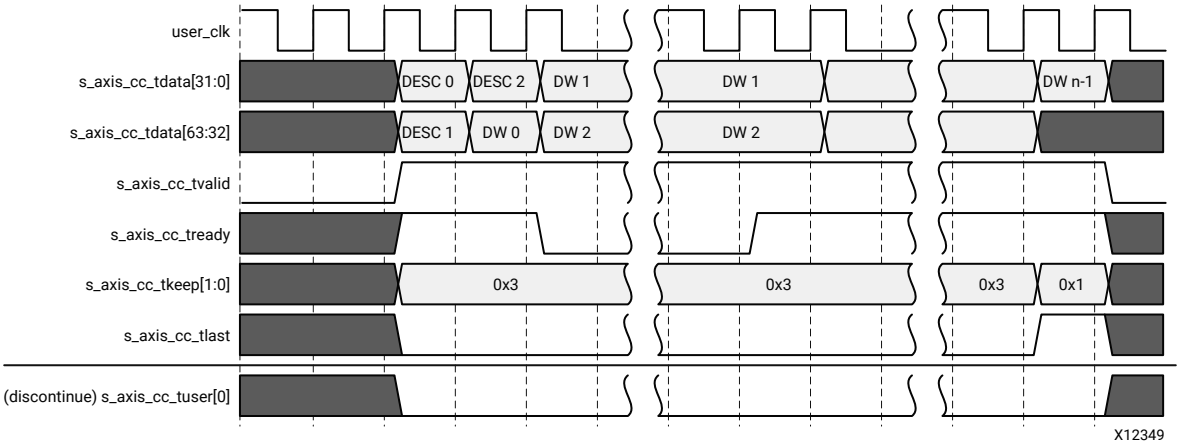


Figure 31: **Transfer of a Normal Completion on the Completer Completion Interface (Dword-Aligned Mode, 128-Bit Interface)**

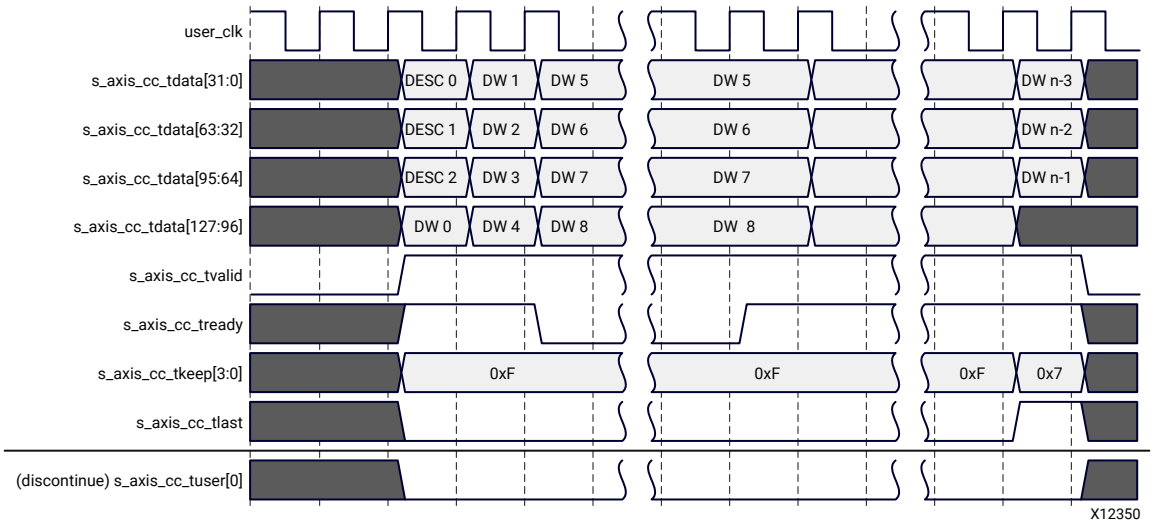
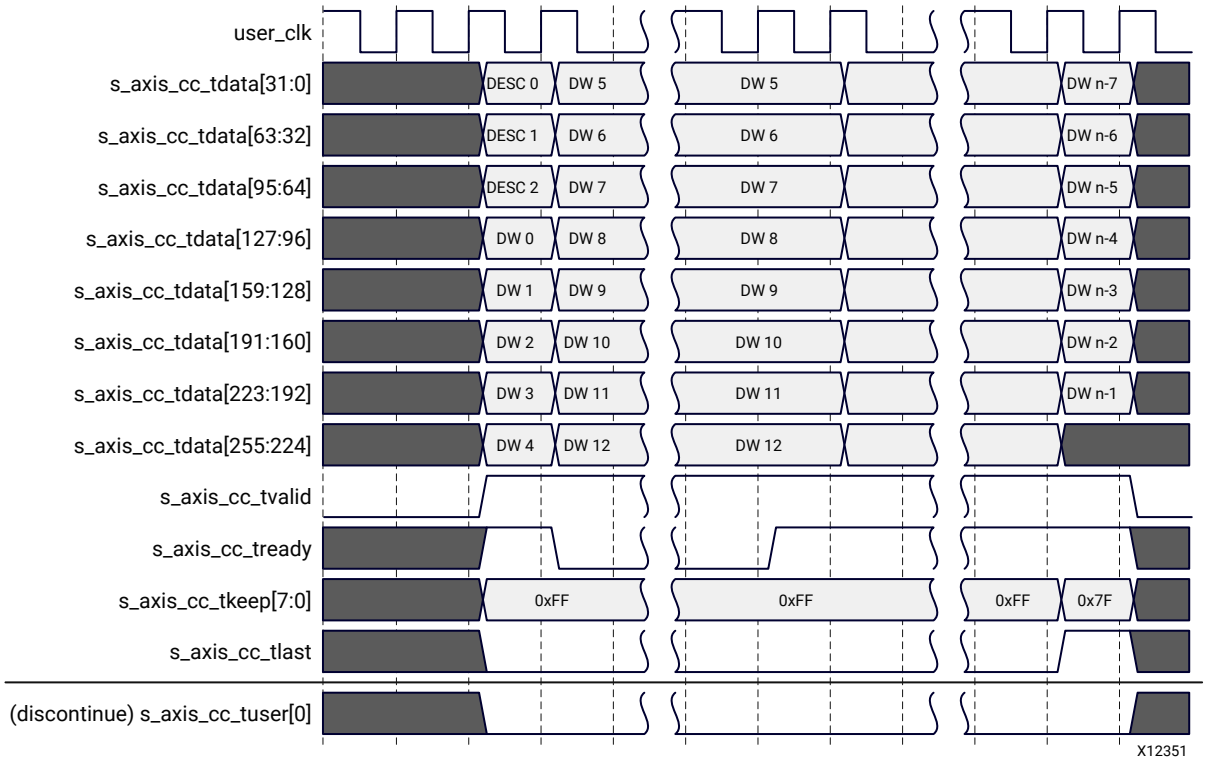


Figure 32: **Transfer of a Normal Completion on the Completer Completion Interface (Dword-Aligned Mode, 256-Bit Interface)**



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

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

Figure 33: **Transfer of a Normal Completion on the Completer Completion Interface (Address-Aligned Mode, 64-Bit Interface)**

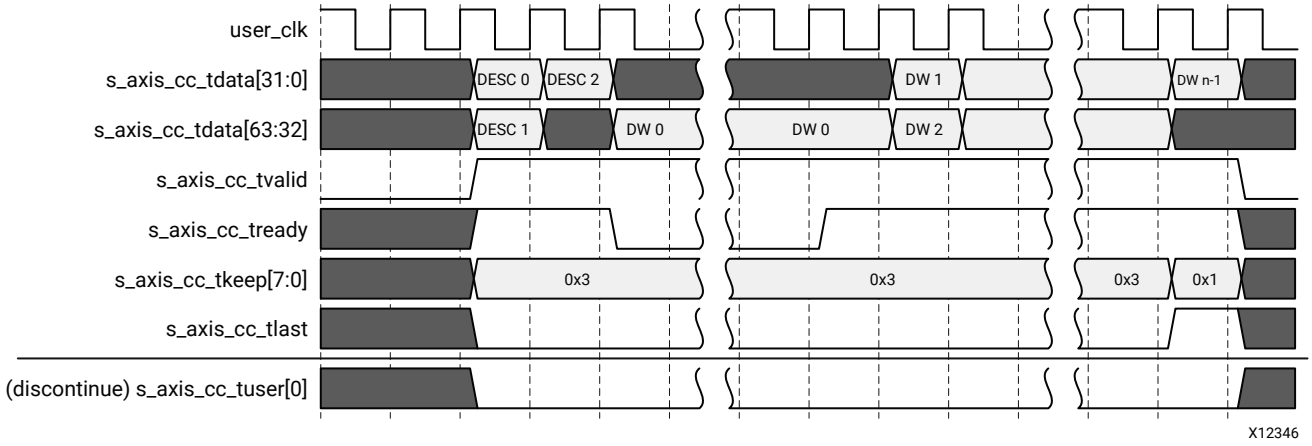


Figure 34: **Transfer of a Normal Completion on the Completer Completion Interface (Address-Aligned Mode, 128-Bit Interface)**

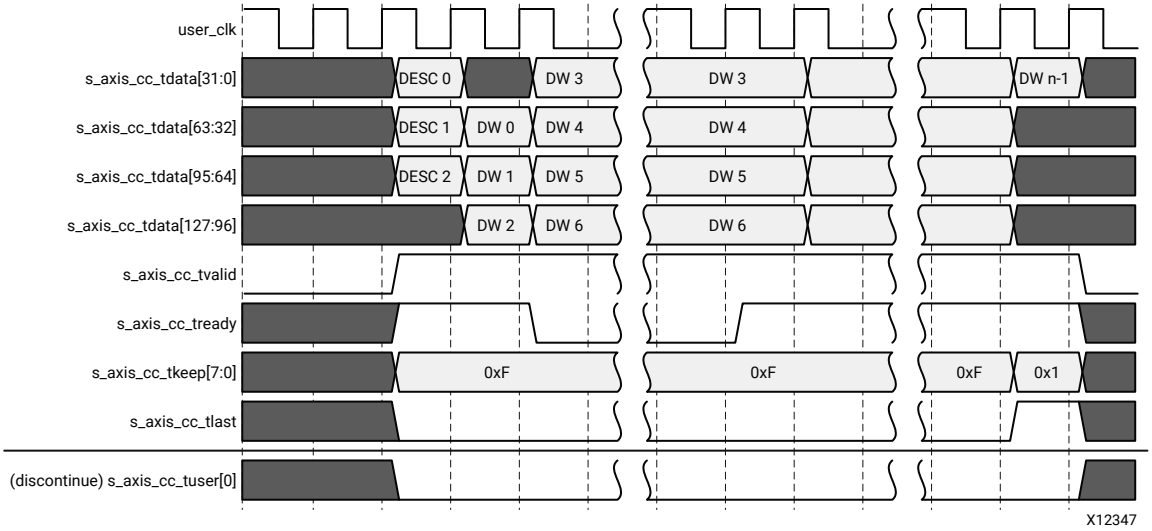
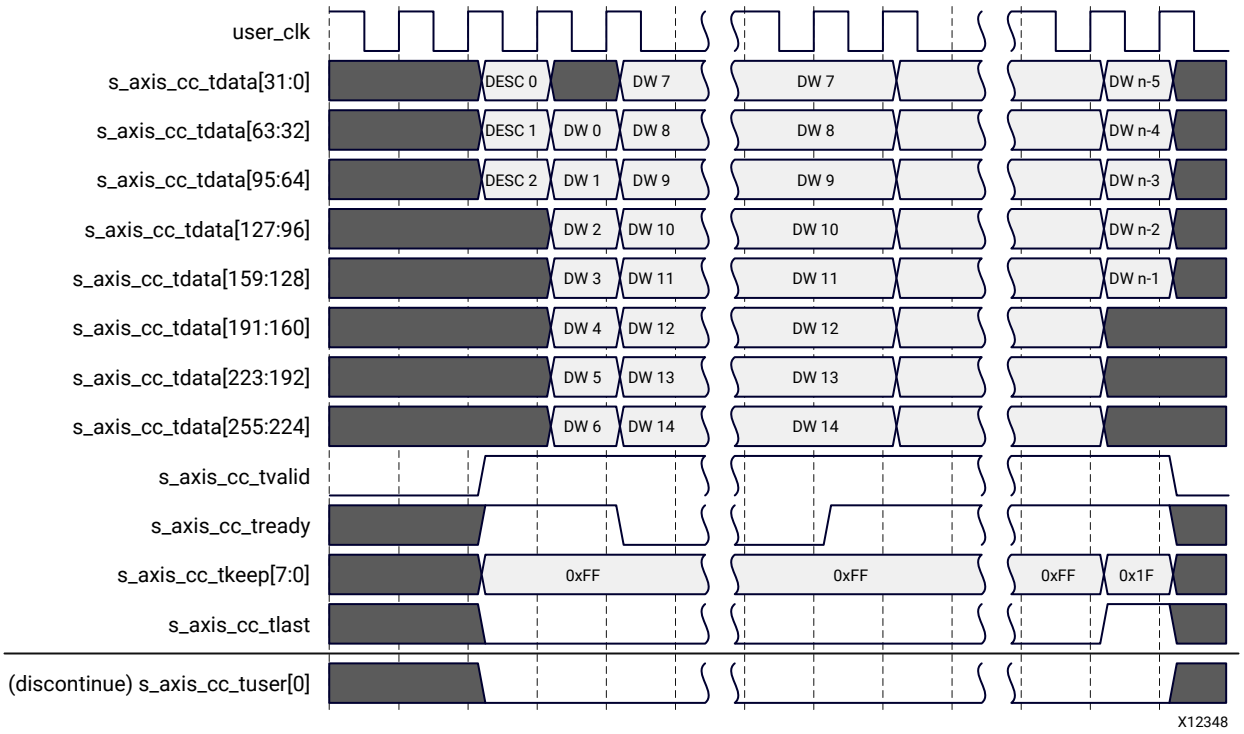


Figure 35: **Transfer of a Normal Completion on the Completer Completion Interface (Address-Aligned Mode, 256-Bit Interface)**



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.

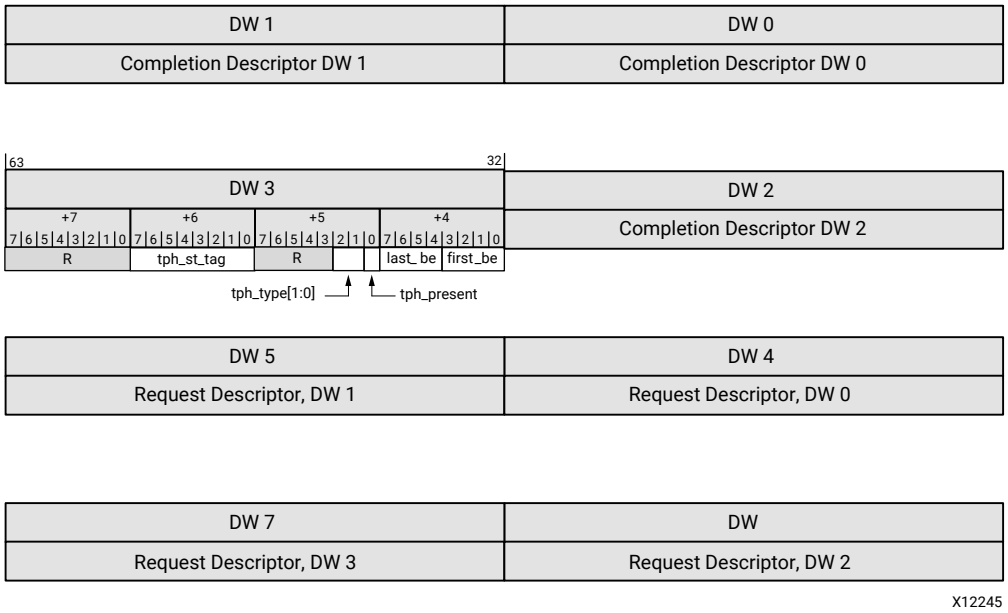
Completions with Error Status (UR and CA)

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 29: Completer Completion Descriptor Format, followed by five additional Dwords containing information on the request that generated the Completion. These five Dwords are necessary for the integrated block to log information about the request in its AER header log registers.

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

- The first three Dwords contain the completion descriptor in the format of Figure 29: Completer Completion Descriptor Format.
- The fourth Dword contains the state of the following signals in `m_axis_cq_tuser`, copied from the request:
  - The First Byte Enable bits `first_be[3:0]` in `m_axis_cq_tuser`.
  - The Last Byte Enable bits `last_be[3:0]` in `m_axis_cq_tuser`.
  - Signals carrying information on Transaction Processing Hint: `tph_present`, `tph_type[1:0]` , and `tph_st_tag[7:0]` in `m_axis_cq_tuser`.
- The four Dwords of the request descriptor received from the integrated block with the request.

Figure 36: **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`.

Requester Interface

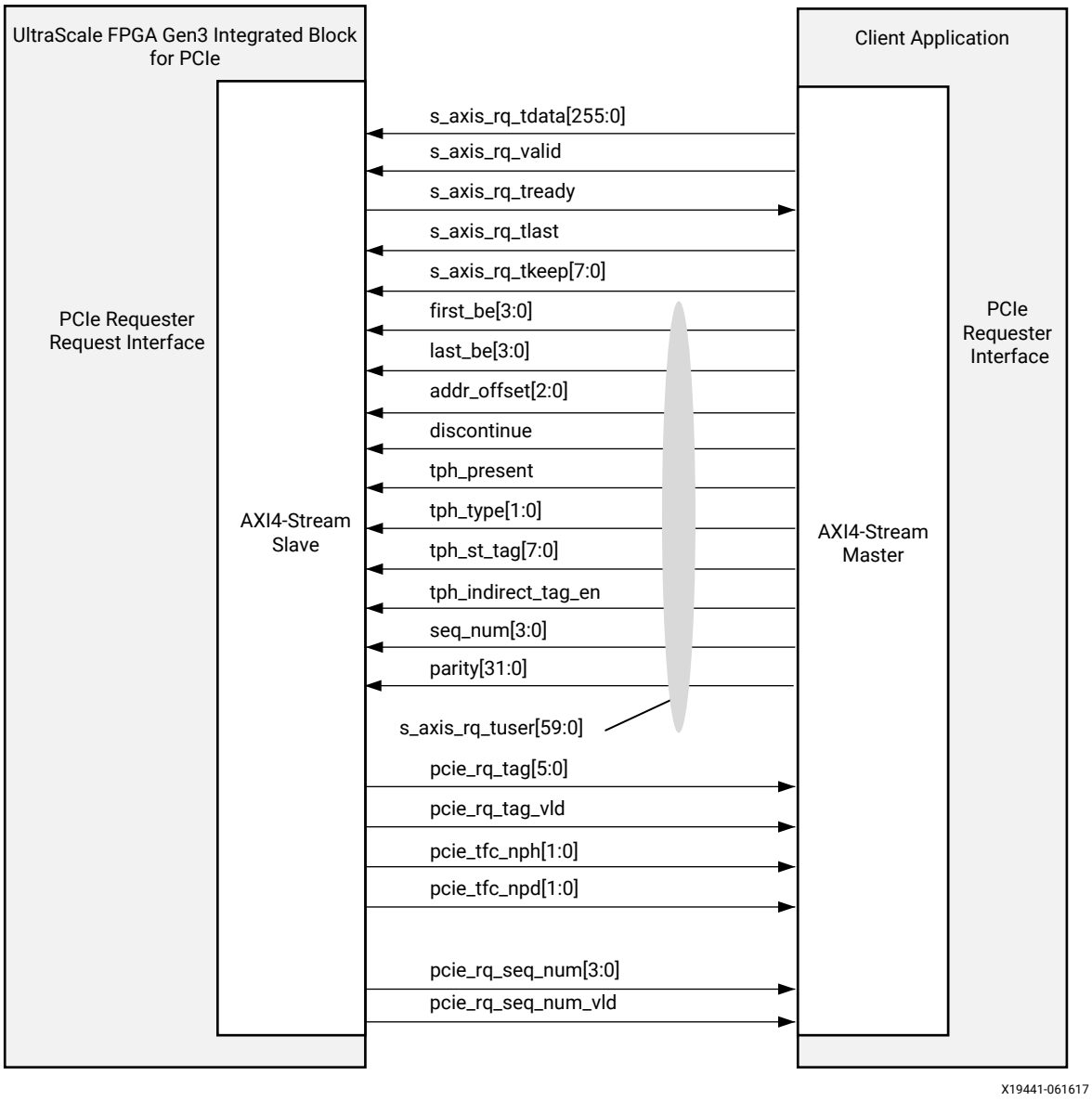
The requester interface enables a user Endpoint application to initiate PCI transactions as a bus master across the PCIe link to the host memory. For Root Complexes, this interface is also used to initiate I/O and configuration requests. This interface can also be used by both Endpoints and Root Complexes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

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

Requester Request Interface Operation

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

Figure 37: **Requester Request Interface**



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

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

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

Requester Request Descriptor Formats

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

The formats of the descriptor for different request types are illustrated in the following figures. The format of the following figure applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request. The format in Figure 39: Requester Request Descriptor Format for Configuration Requests is used for Configuration Requests. The format in Figure 40: Requester Request Descriptor Format for Vendor-Defined Messages is used for Vendor-Defined Messages (Type 0 or Type 1) only. The format in Figure 41: Requester Request Descriptor Format for ATS Messages 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 42: Requester Request Descriptor Format for all other Messages.

Figure 38: **Requester Request Descriptor Format for Memory, I/O, and Atomic Op Requests**

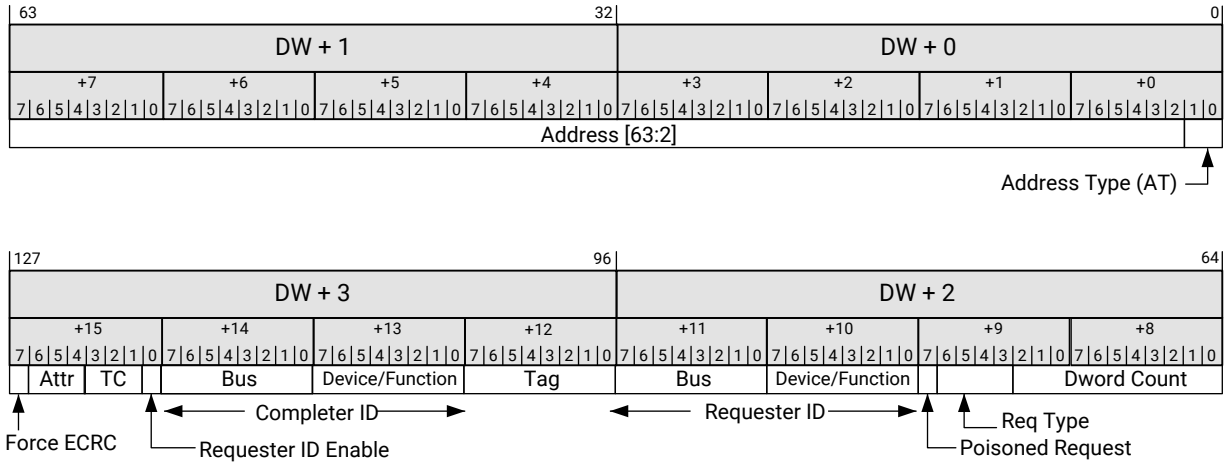


Figure 39: **Requester Request Descriptor Format for Configuration Requests**

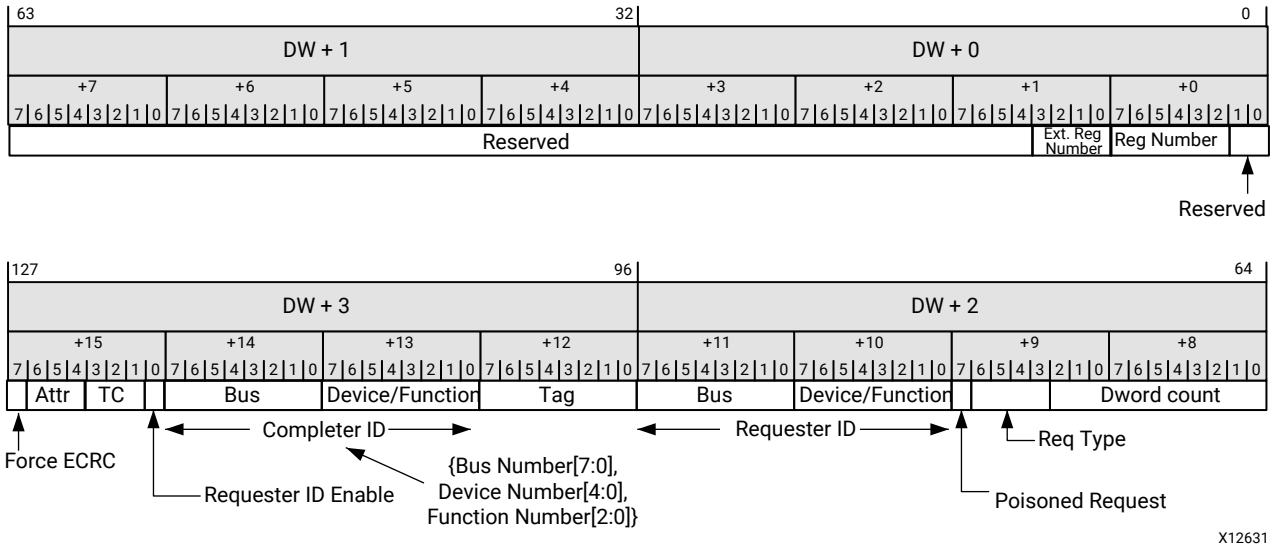


Figure 40: **Requester Request Descriptor Format for Vendor-Defined Messages**

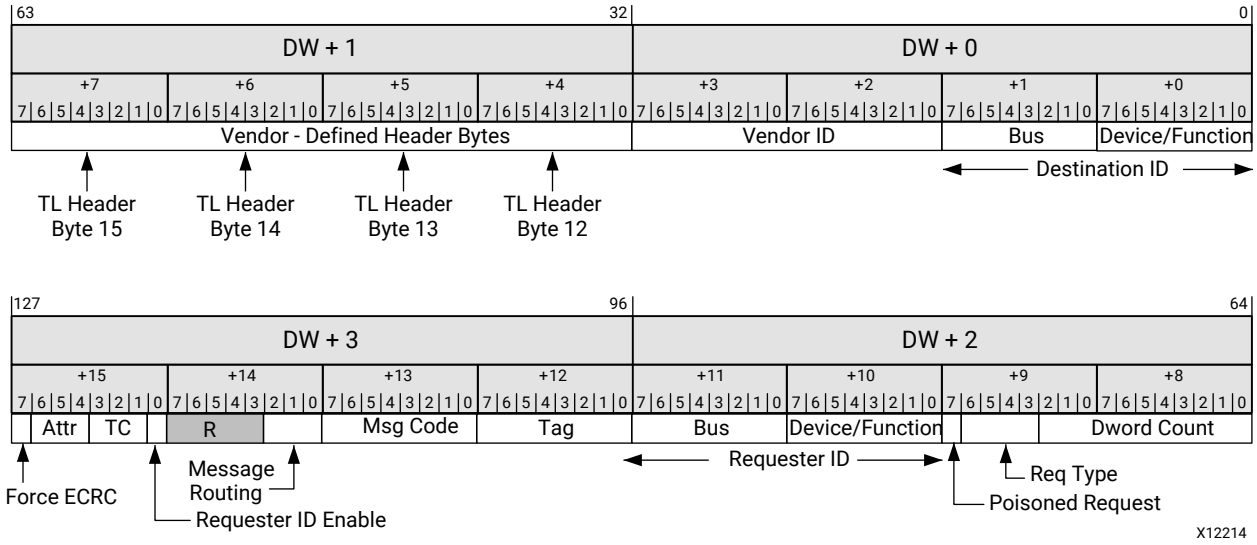


Figure 41: **Requester Request Descriptor Format for ATS Messages**

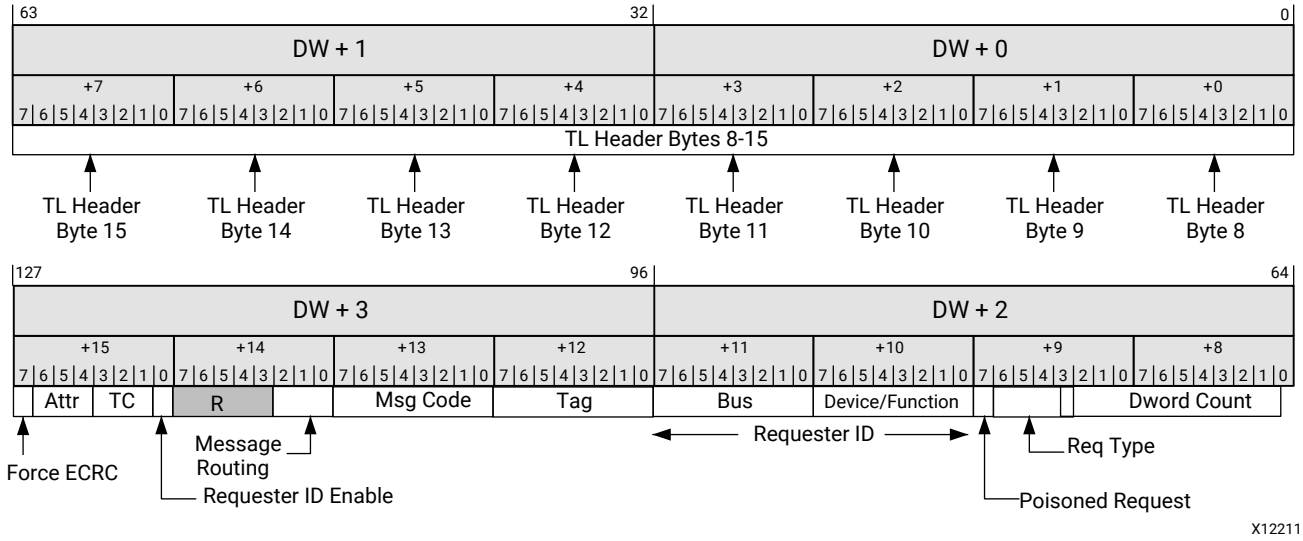
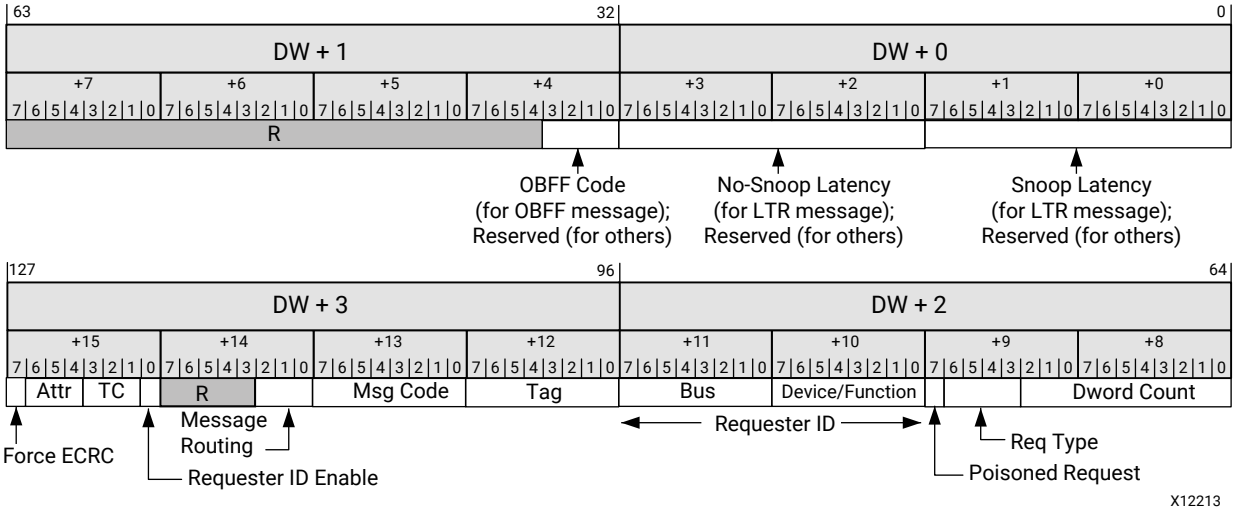


Figure 42: **Requester Request Descriptor Format for all other Messages**



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

Table 39: **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.
78:75	Request Type	Identifies the transaction type. The transaction types and their encodings are listed in Table 6.

Table 39: **Requester Request Descriptor Fields** *(cont'd)*

Bit Index	Field Name	Description
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	Device and/or Function number of the Requester Function. Endpoint mode: • ARI enabled: ◦ Bits [87:80] must be set to the Requester Function number. • ARI disabled: ◦ Bits [82:80] must be set to the Requester Function number. ◦ Bits [87:83] are not used Upstream Port for Switch use case (Endpoint mode is selected within the IP): • ARI enabled: ◦ Bits [87:80] must be set to the Requester Function number. • ARI disabled: ◦ Bits [82:80] must be set to the Requester Function number. ◦ Bits [87:83] are not used if the request is originating from the switch itself. These bits must be set to the Requester Device number where the request was originated if the switch is relaying the request (Requester is external to the switch). This is used with Requester ID Enable bit in the descriptor. Root Port mode (Downstream Port): • ARI enabled: ◦ Bits [87:80] must be set to the Requester Function number. • ARI disabled: ◦ Bits [87:80] must be set to the Requester Function number. • Bits [87:83] must be set to the Requester Device number. This is used with Requester ID Enable bit in the descriptor.

Table 39: **Requester Request Descriptor Fields** *(cont'd)*

Bit Index	Field Name	Description
95:88	Requester Bus Number	Bus number associated with the Requester Function. Endpoint mode: • Not used Upstream Port for Switch use case (Endpoint mode is selected within the IP): • Not used if the request is originating from the switch itself. These bits must be set to the Requester Bus number where the request was originated if the switch is relaying the request (Requester is external to the switch). This is used with Requester ID Enable bit in the descriptor. Root Port mode (Downstream Port): • Must be set to the Requester Bus number. This is used with Requester ID Enable bit in the descriptor.
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.
119:104	Completer ID	This field is applicable only to Configuration requests and messages routed by ID. For these requests, this field specifies the PCI Completer ID associated with the request (these 16 bits are divided into an 8-bit bus number, 5-bit device number, and 3-bit function number in the legacy interpretation mode. In the ARI mode, these 16 bits are treated as an 8-bit bus number + 8-bit Function number.).

Table 39: **Requester Request Descriptor Fields** *(cont'd)*

Bit Index	Field Name	Description
120	Requester ID Enable	1'b1: The client supplies Bus, Device, and Function numbers in the descriptor to be populated as the Requester ID field in the TLP header. 1'b0: IP uses Bus and Device numbers captured from received Configuration requests and the client supplies Function numbers in the descriptor to be populated as the Requester ID field in the TLP header. Endpoint mode: • Must be set to 1'b0. Upstream Port for Switch use case (Endpoint mode is selected within the IP): • Set to 1'b0 when the request is originating from the switch itself. • Set to 1'b1 when the switch is relaying the request (Requester is external to the switch). This is used with Requester Bus Number bits [95:88] and Requester Function/Device Number bits [87:83] when ARI is not enabled. Root Port mode: • Must be set to 1'b1. This is used with Requester Bus Number bits [95:88] and Requester Function/Device Number bits [87:83] when ARI is not enabled.
123:121	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request.
126:124	Attributes	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 39: **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 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 `s_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. For memory writes and reads of one DW transfers, `last_be` should be 0 and bits in `first_be` indicate valid bytes. In all other cases, the 1 bits in `first_be` and `last_be` must be contiguous.

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

Figure 43: **Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, 64-Bit Interface)**

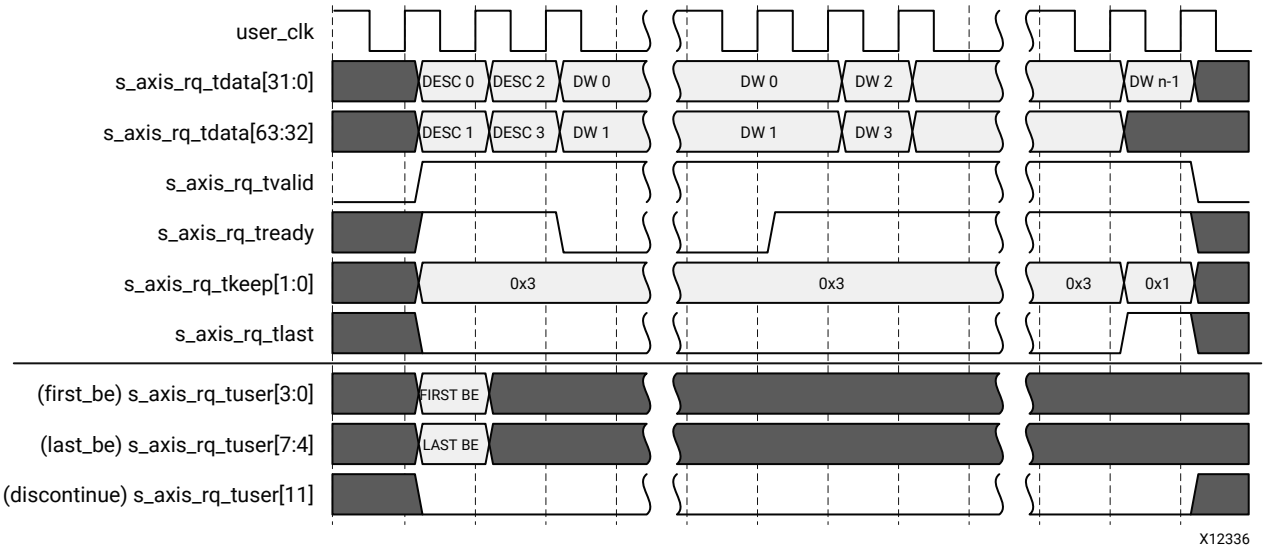


Figure 44: **Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, 128-Bit Interface)**

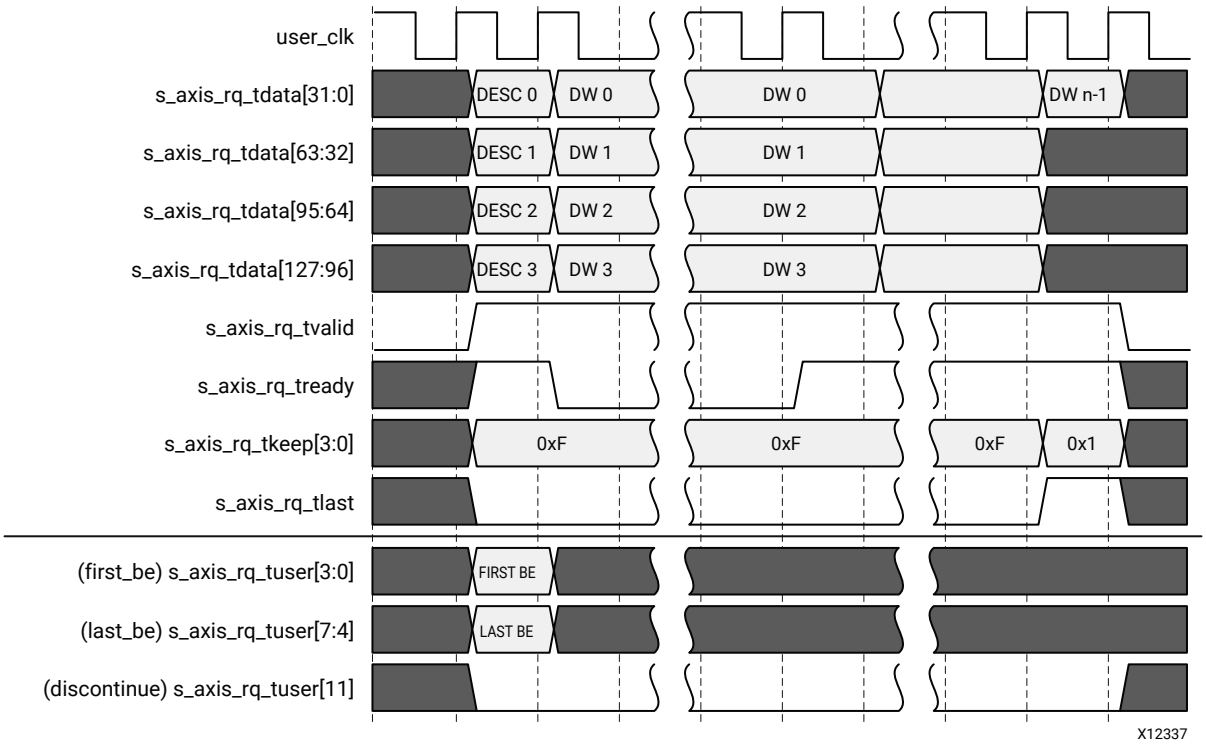
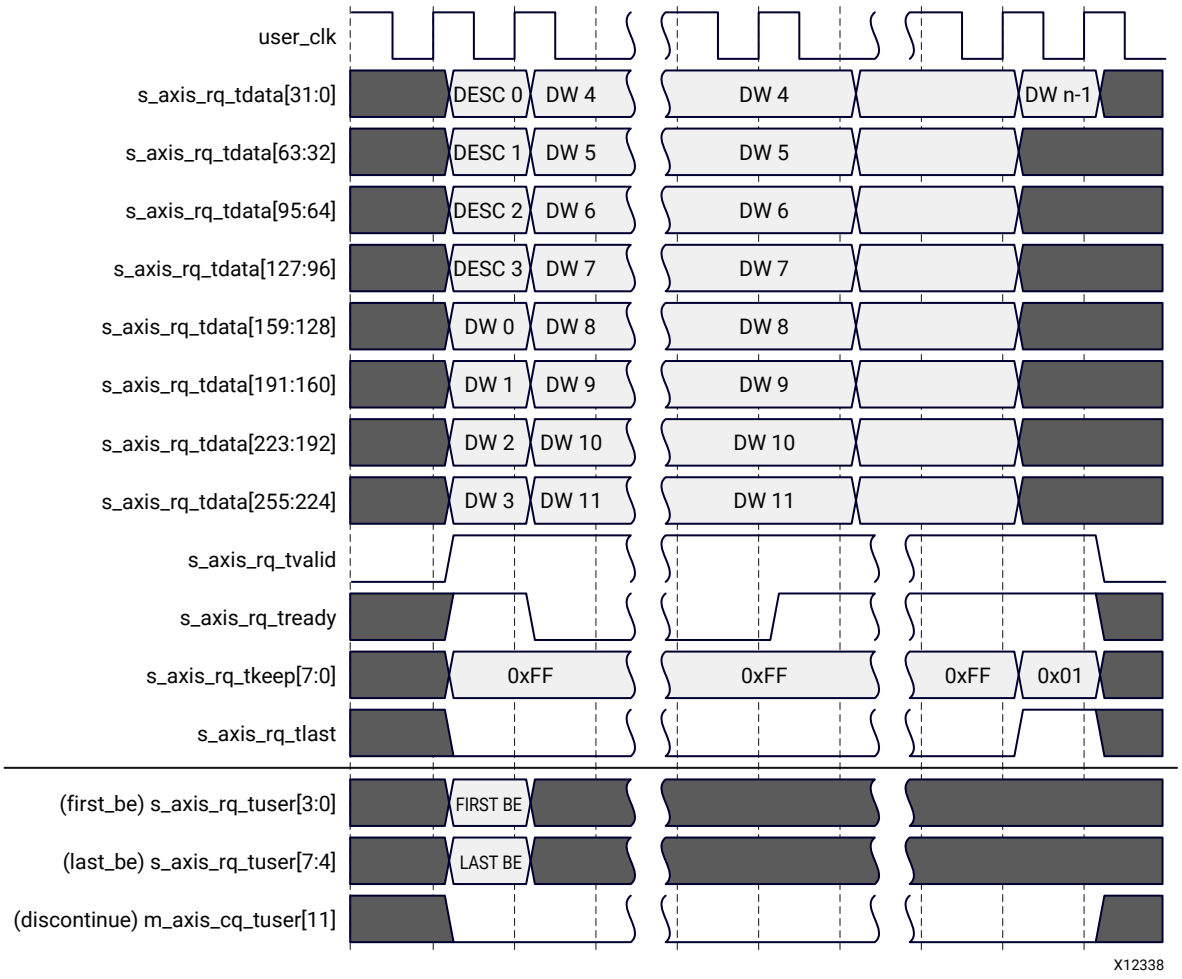


Figure 45: **Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, 256-Bit Interface)**



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

In the address-aligned mode, the delivery of the payload always starts in the beat following the last byte of the descriptor. The first Dword of the payload can appear at any Dword position. The user application must communicate the offset of the first Dword of the payload on the datapath using the `addr_offset[2:0]` signal in `s_axis_rq_tuser`. The user application must also set the bits in `last_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 46: **Memory Write Transaction on the Requester Request Interface (Address-Aligned Mode, 64-Bit Interface)**

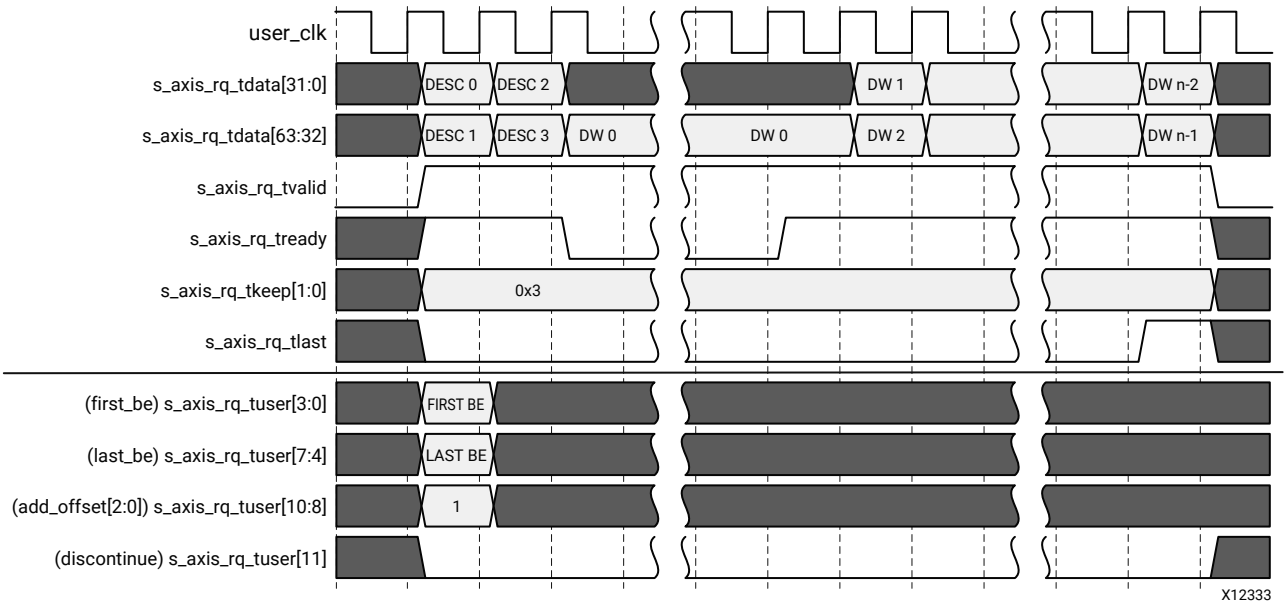


Figure 47: **Memory Write Transaction on the Requester Request Interface (Address-Aligned Mode, 128-Bit Interface)**

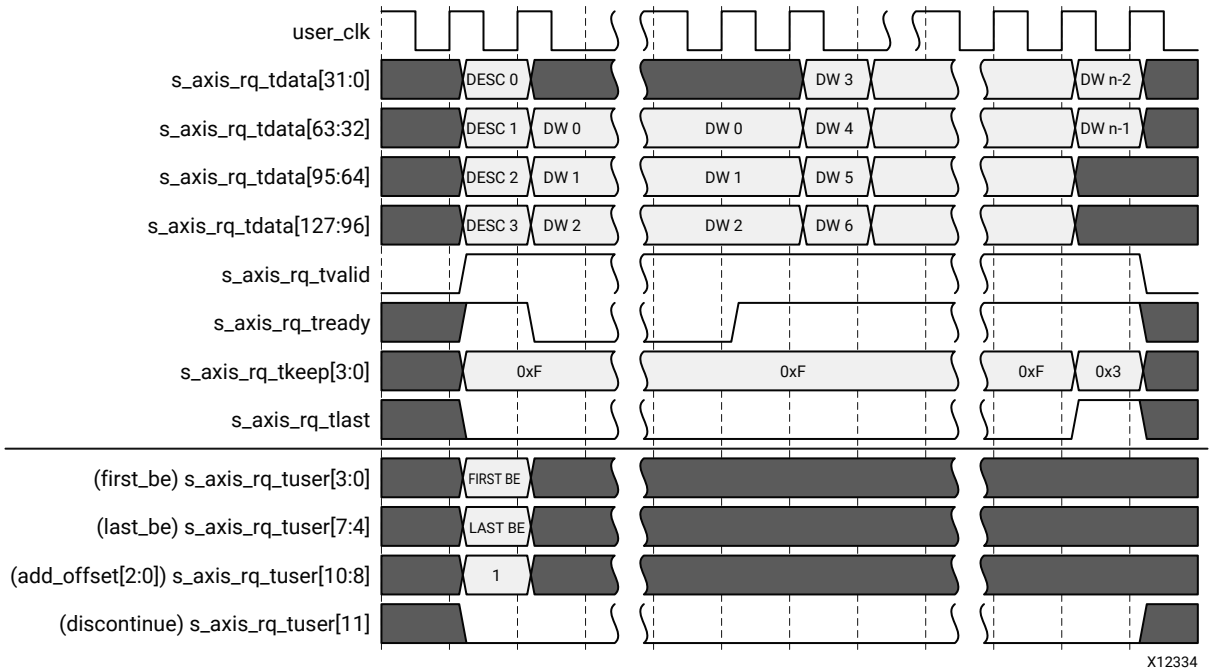
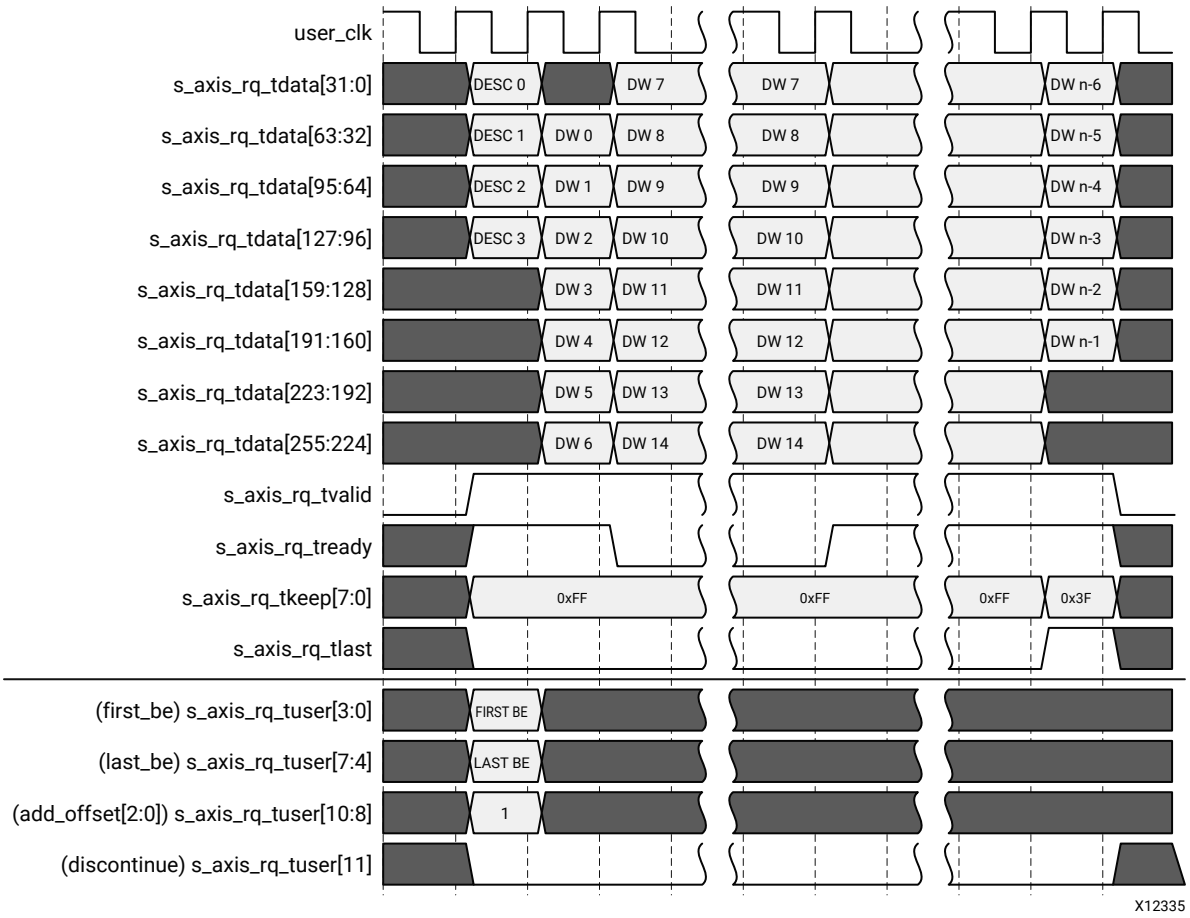


Figure 48: **Memory Write Transaction on the Requester Request Interface (Address-Aligned Mode, 256-Bit Interface)**



Non-Posted Transactions with No Payload

Non-Posted transactions with no payload (memory read requests, I/O read requests, Configuration read requests) are transferred across the RQ interface in the same manner as a memory write request, except that the AXI4-Stream packet contains only the 16-byte descriptor. The following timing diagrams in 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. For memory writes and reads of one DW transfers, `last_be[3:0]` should be 0s and bits in `first_be[3:0]` indicate the valid bytes. 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 49: **Memory Read Transaction on the Requester Request Interface (64-Bit Interface)**

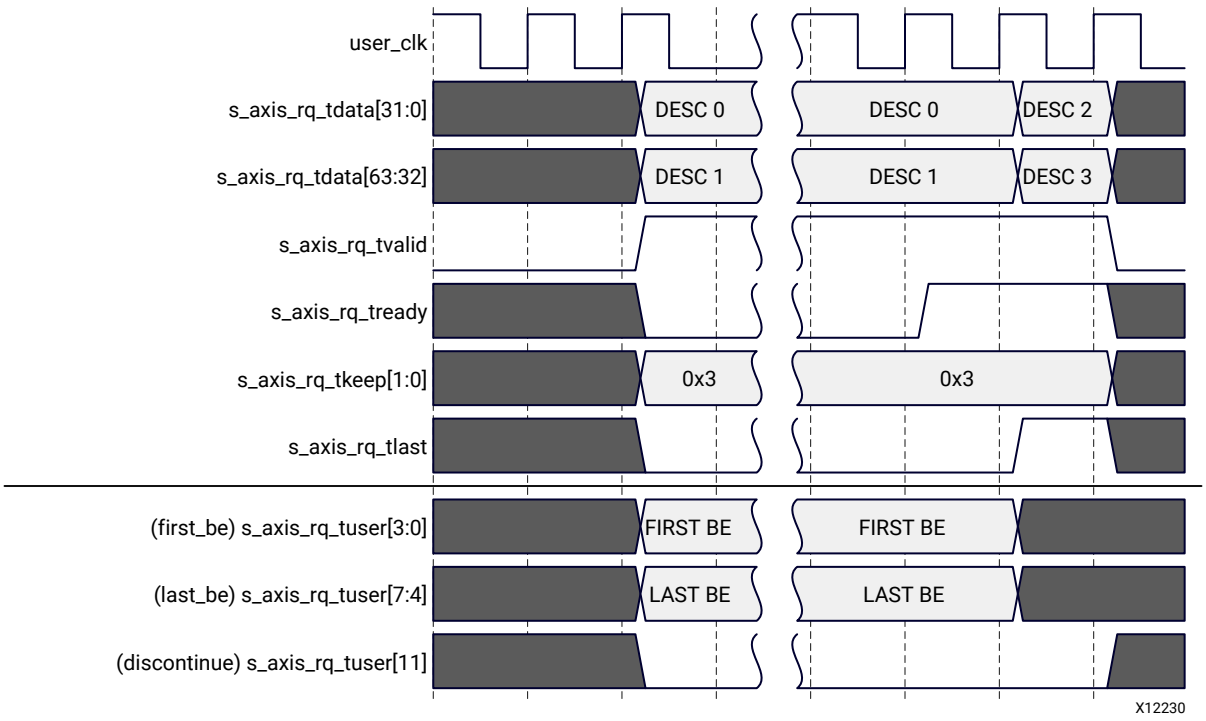


Figure 50: **Memory Read Transaction on the Requester Request Interface (128-Bit Interface)**

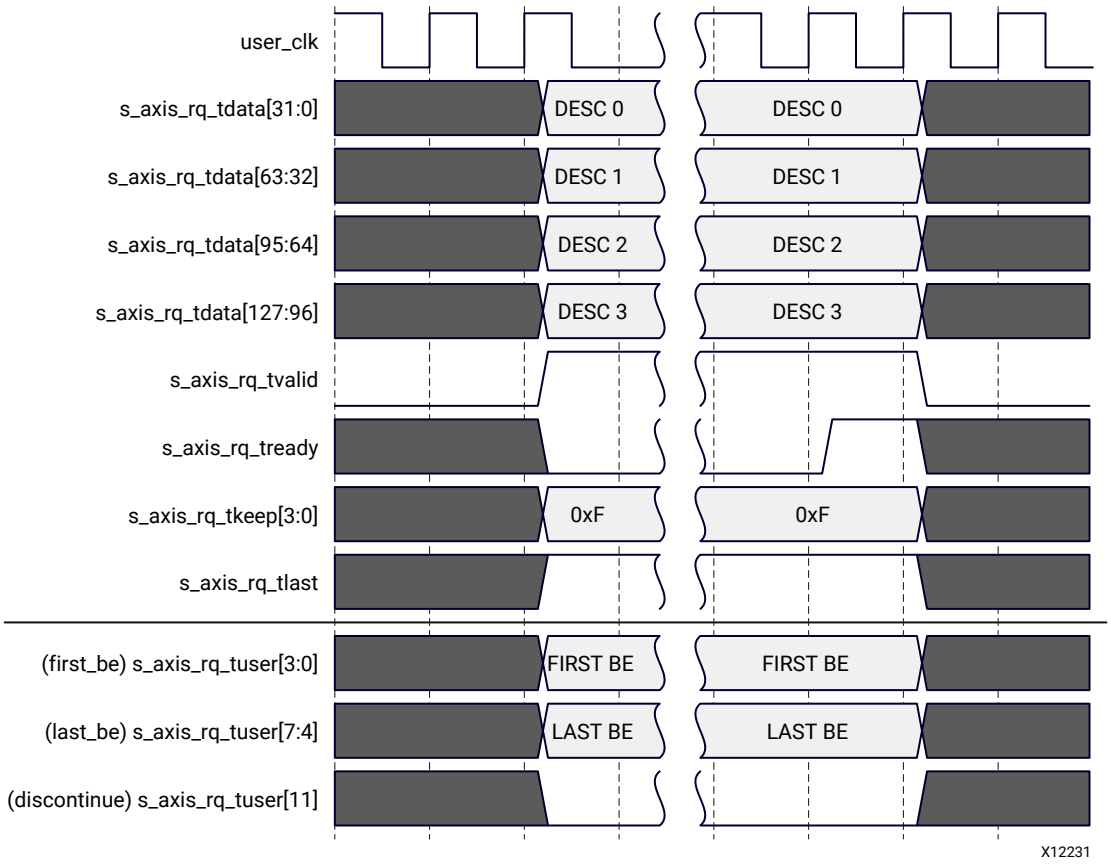
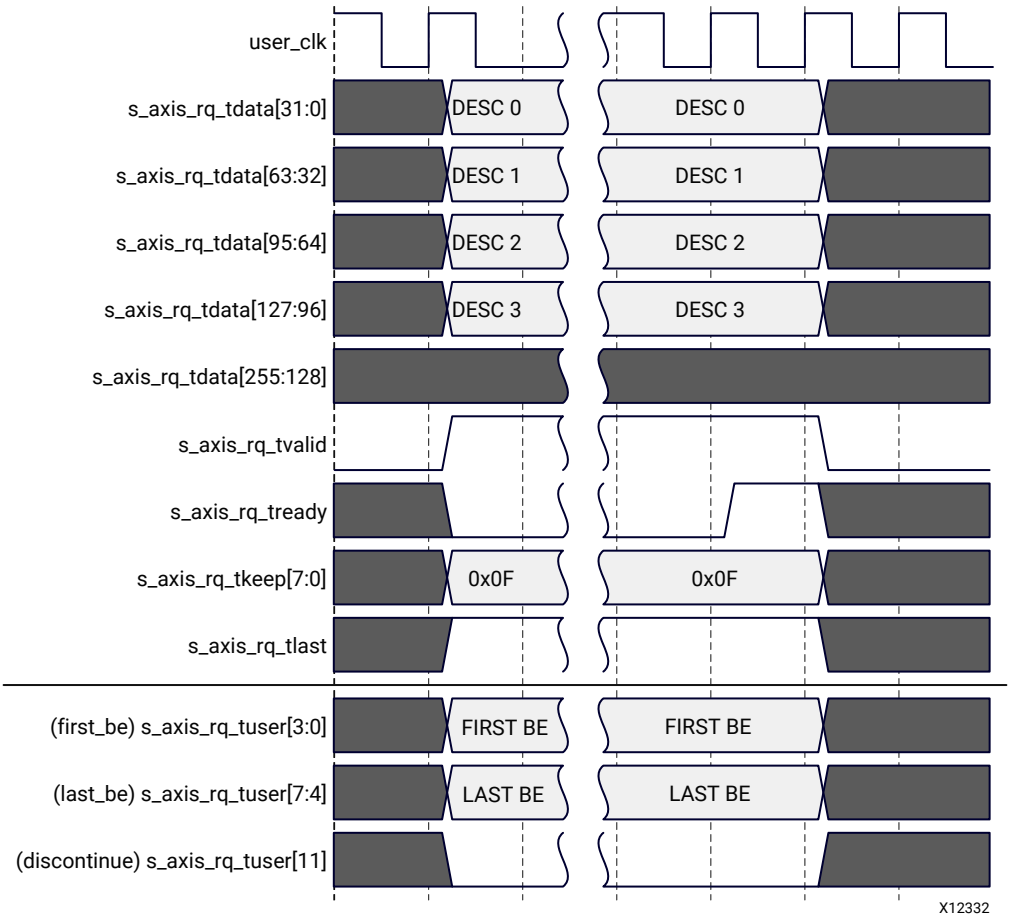


Figure 51: **Memory Read Transaction on the Requester Request Interface (256-Bit Interface)**



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 64 Non-Posted transactions. The returning Completions are matched with the pending requests using a 6-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_tag[5:0]`. The value on this bus is valid when the integrated block asserts `pcie_rq_tag_vld`. 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 (64).

- **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_vld` for one cycle for each Non-Posted request, after it has placed its allocated tag on `pcie_rq_tag[5:0]`. 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_vld` 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_tag

bus in FIFO order, so it is easy to associate the tag value with the request it transferred. A tag is reused when the end-of-frame (EOF) of the last completion of a split completion is accepted by the user application.

Avoiding Head-of-Line Blocking for Posted Requests

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

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

The user logic can optionally check these outputs before transmitting Non-Posted requests. Because of internal pipeline delays, the information on these outputs is delayed by two user clock cycles from the cycle in which the last byte of the descriptor is transferred on the RQ interface. Thus, the user logic must adjust these values, taking into account any Non-Posted requests transmitted in the two previous clock cycles. The following figures 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 53: Credit and Tag Availability Signals on the Requester Request Interface (128-Bit Interface) and Figure 54: Credit and Tag Availability Signals on the Requester Request Interface (64-Bit Interface) 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 52: **Credit and Tag Availability Signals on the Requester Request Interface (256-Bit Interface)**

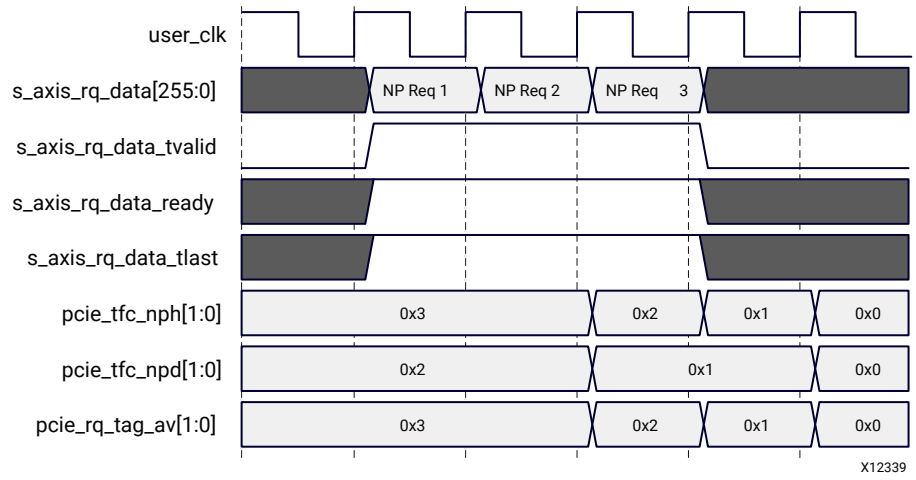


Figure 53: **Credit and Tag Availability Signals on the Requester Request Interface (128-Bit Interface)**

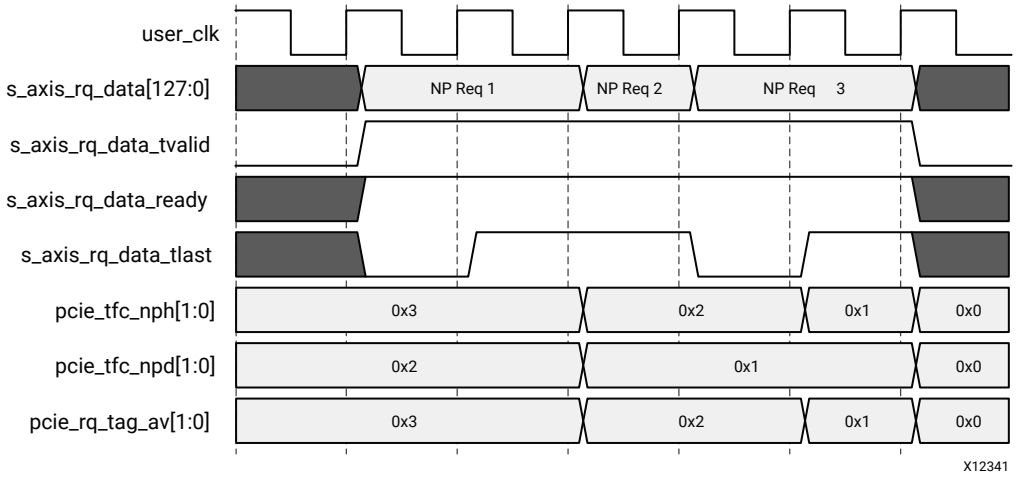
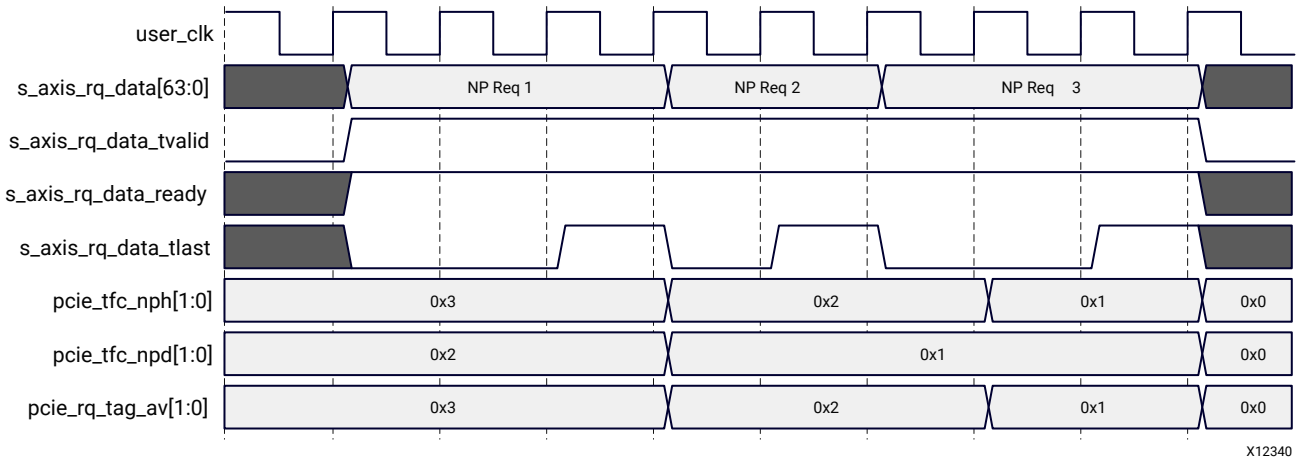


Figure 54: **Credit and Tag Availability Signals on the Requester Request Interface (64-Bit Interface)**



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

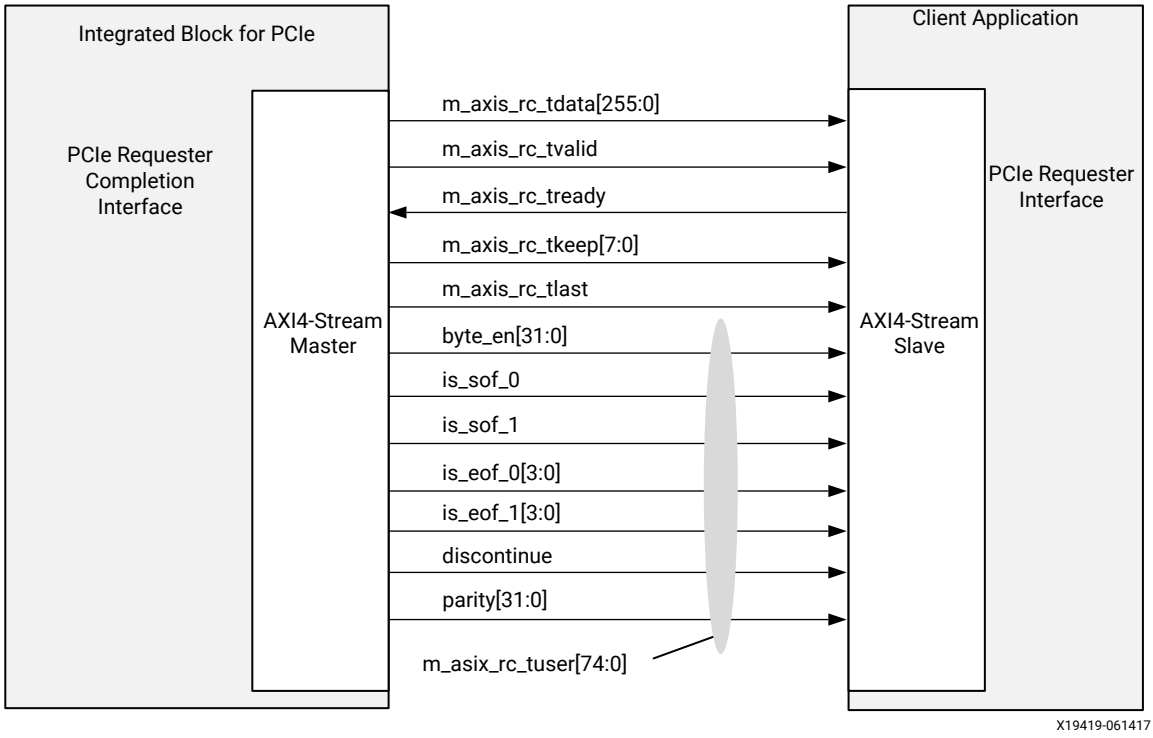
pcie_rq_seq_num[3:0]

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 the following figure for an illustration of signals associated with the requester completion interface. When straddle is not enabled, the integrated block delivers each TLP on this interface as an AXI4-Stream packet. The packet starts with a 96-bit descriptor, followed by data in the case of Completions with a payload.

Figure 55: **Requester Completion Interface**



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

Requester Completion Descriptor Format

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

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

Figure 56: **Requester Completion Descriptor Format**

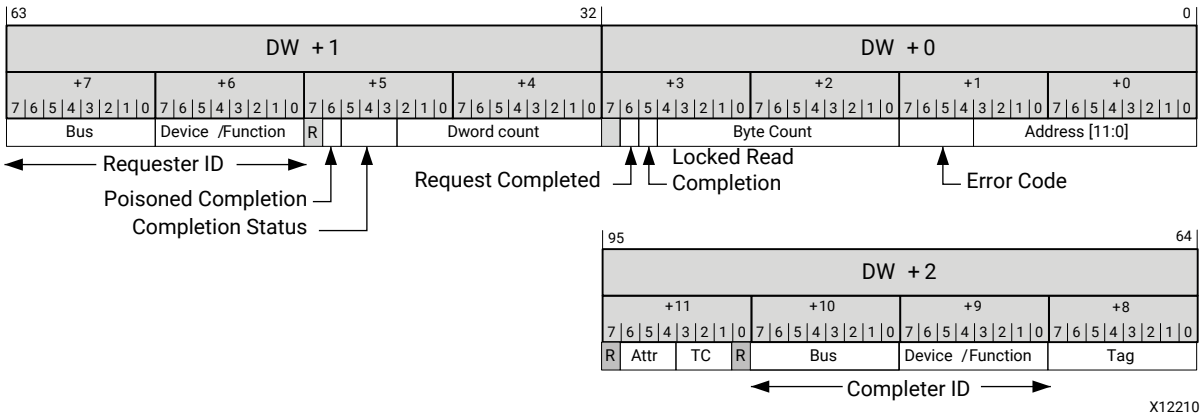


Table 40: **Requester Completion Descriptor Fields**

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. ***Note***: This is a byte-level address.

Table 40: **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 40: **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 following timing diagrams illustrate the transfer of a Completion TLP received from the link with no associated payload across the RC interface, when the interface width is configured as 64, 128, and 256 bits, respectively. The timing diagrams in this section assume that the Completions are not straddled on the 256-bit interface. The straddle feature is described in Straddle Option for 256-Bit Interface.

Figure 57: **Transfer of a Completion with no Data on the Requester Completion Interface (64-Bit Interface)**

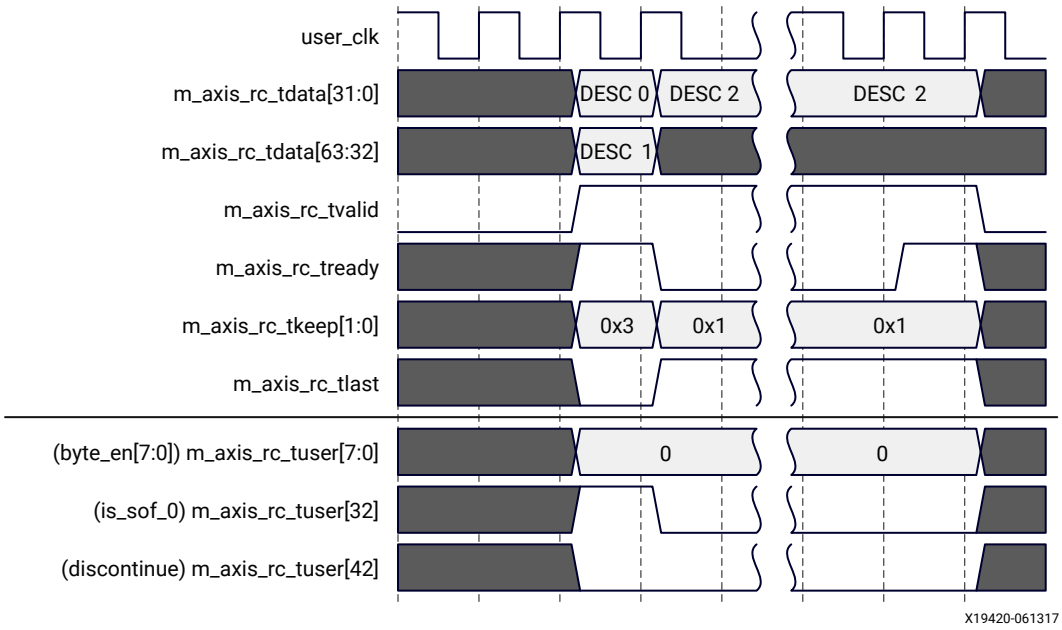


Figure 58: **Transfer of a Completion with no Data on the Requester Completion Interface (128-Bit Interface)**

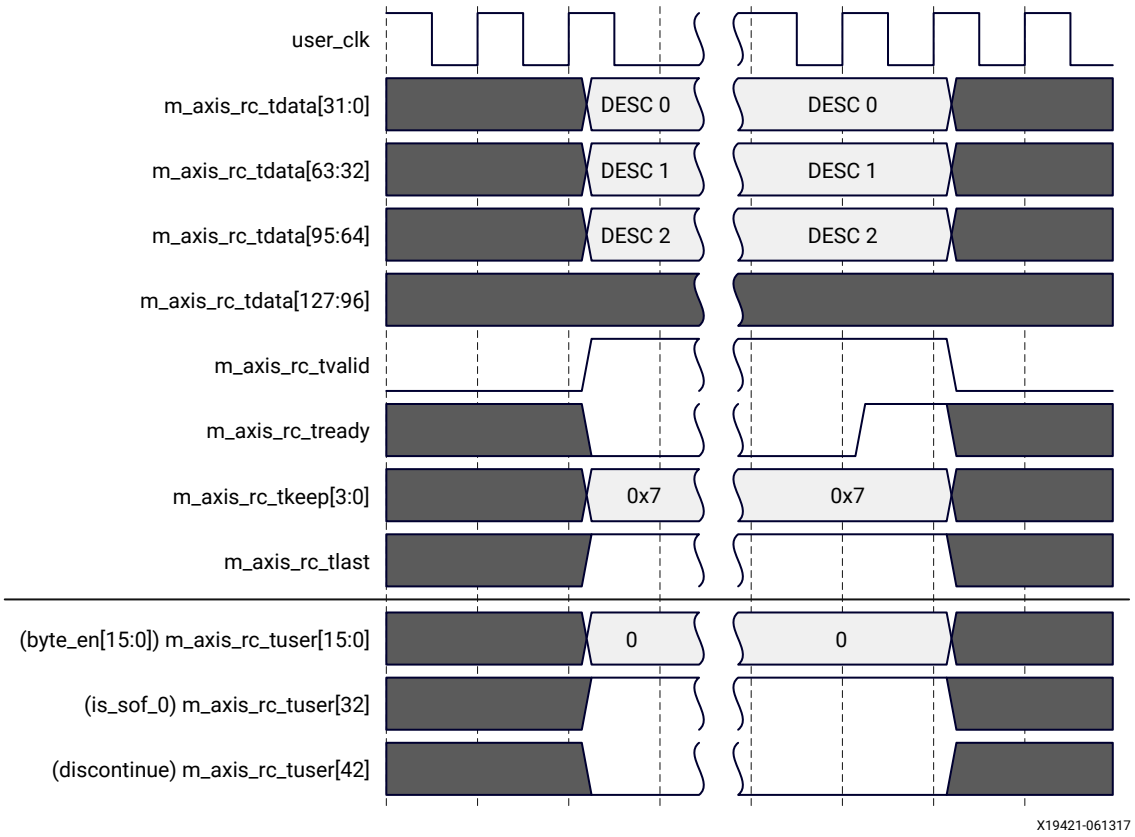
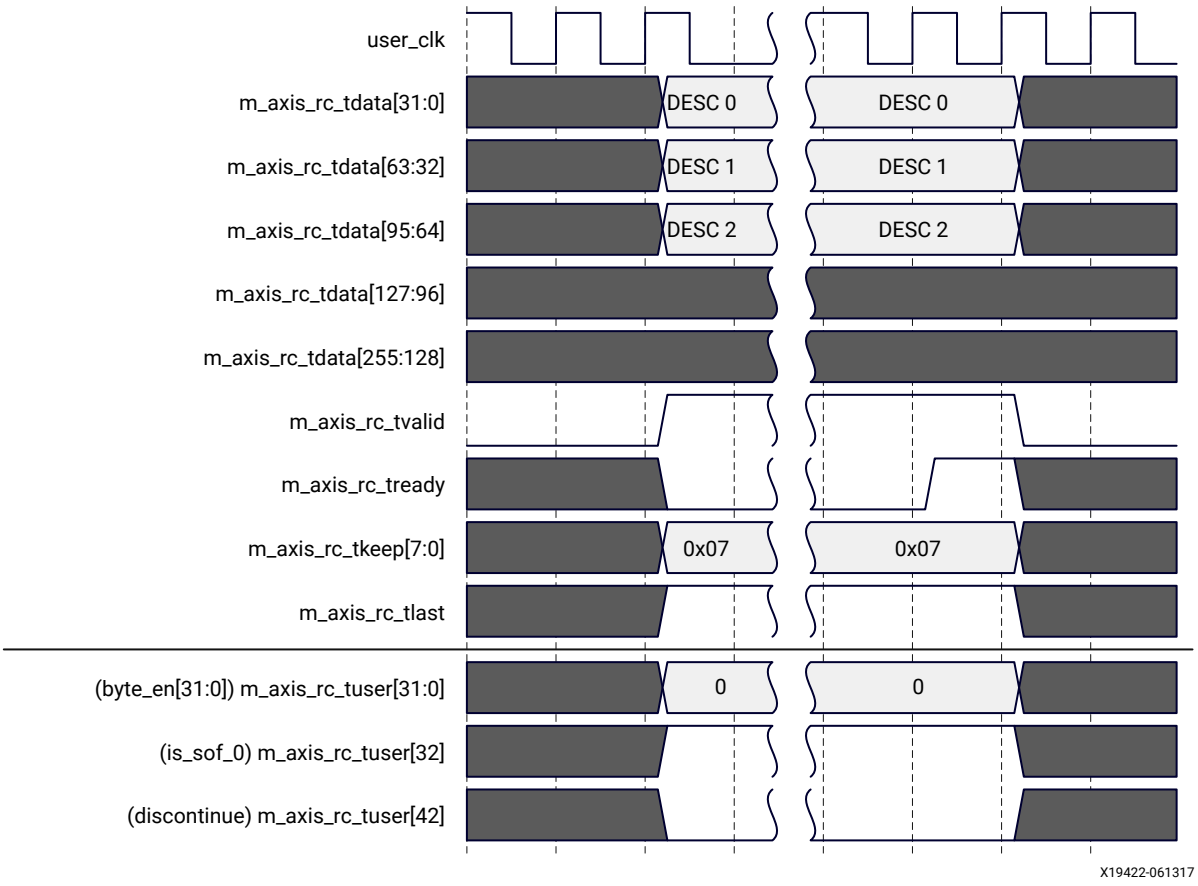


Figure 59: **Transfer of a Completion with no Data on the Requester Completion Interface (256-Bit Interface)**



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 following timing diagrams in 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* × 32 + 28, *k* > 0. The timing diagrams in this section assume that the Completions are not straddled on the 256-bit interface. The straddle feature is described in Straddle Option for 256-Bit Interface.

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

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

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

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

Figure 60: **Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode, 64-Bit Interface)**

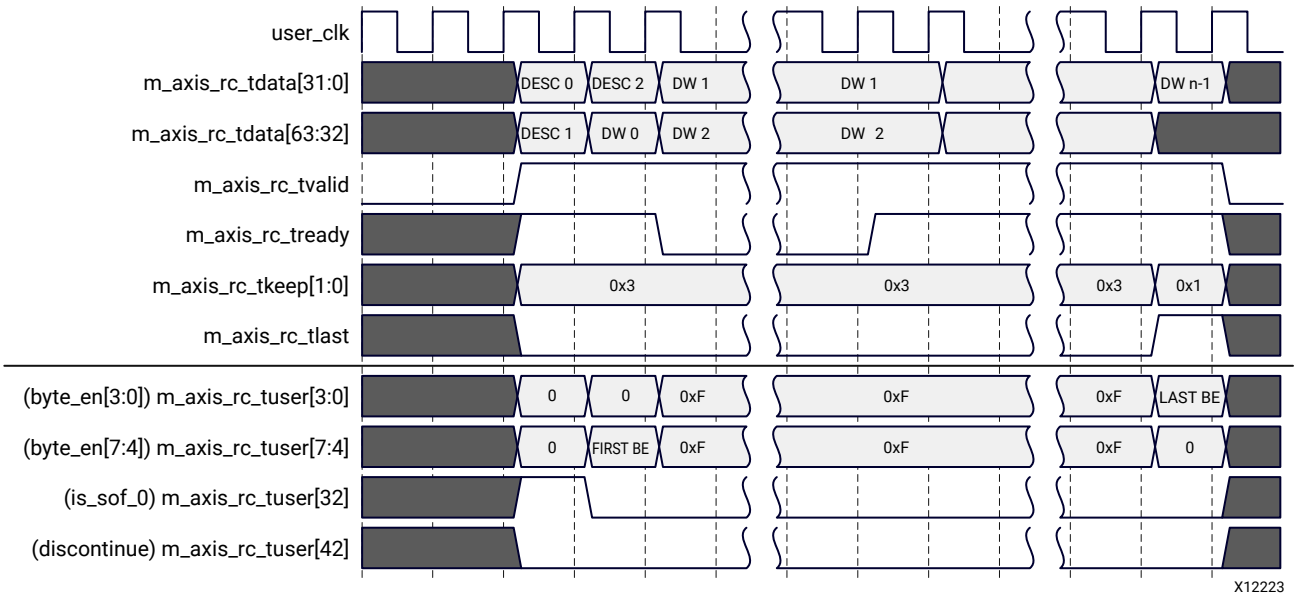


Figure 61: **Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode, 128-Bit Interface)**

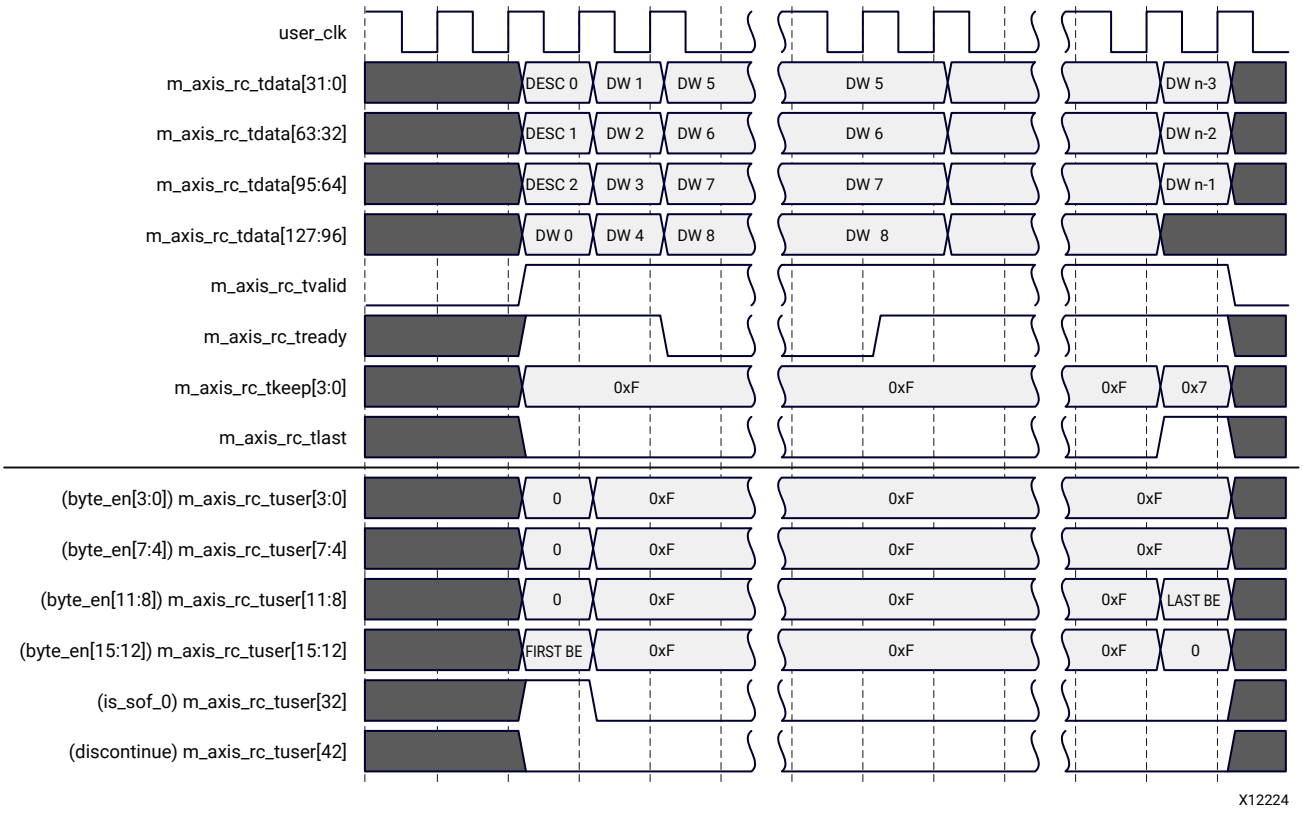
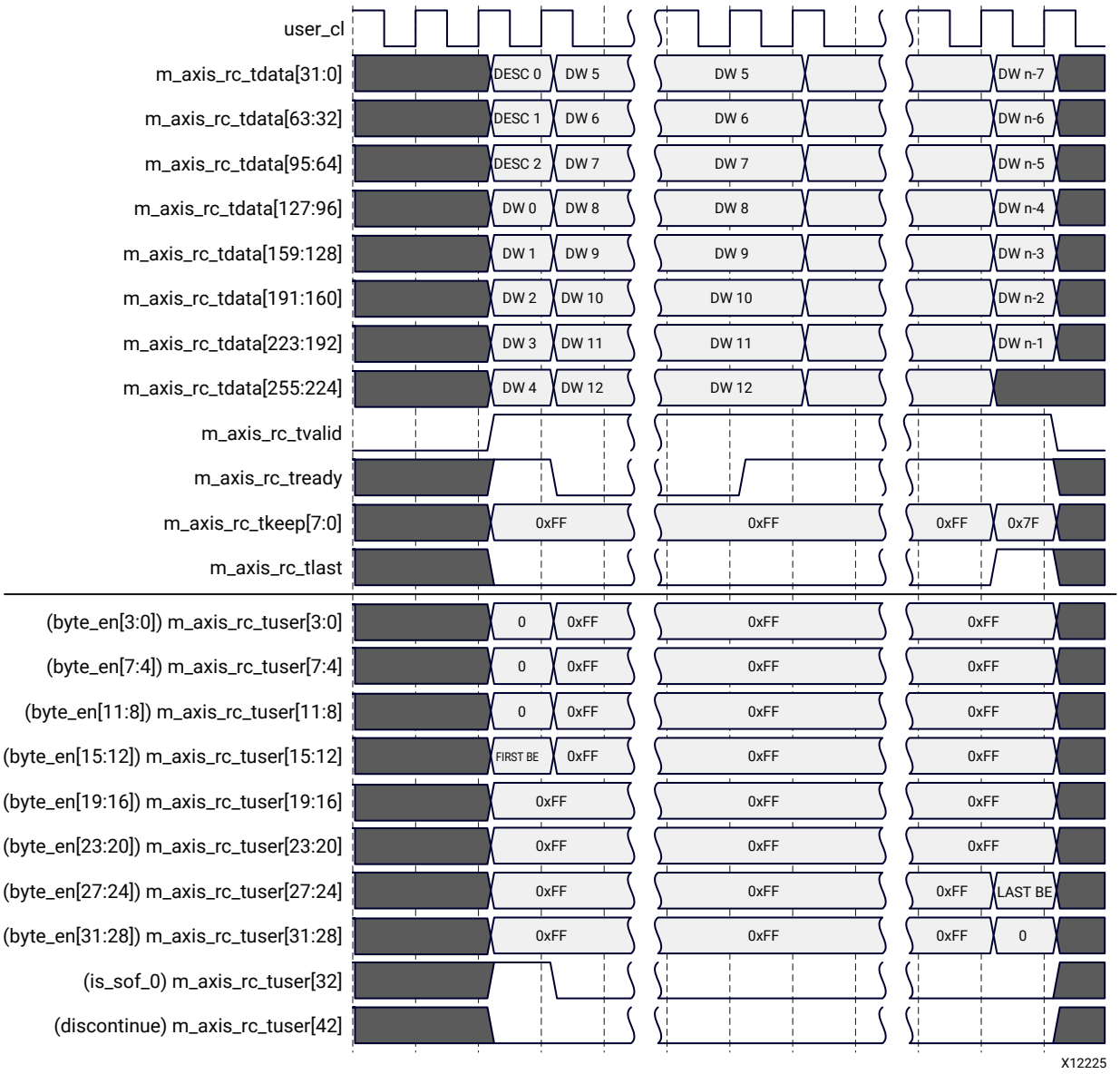


Figure 62: **Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode, 256-Bit Interface)**



The following timing diagrams in 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 63: **Transfer of a Completion with Data on the Requester Completion Interface (Address-Aligned Mode, 64-Bit Interface)**

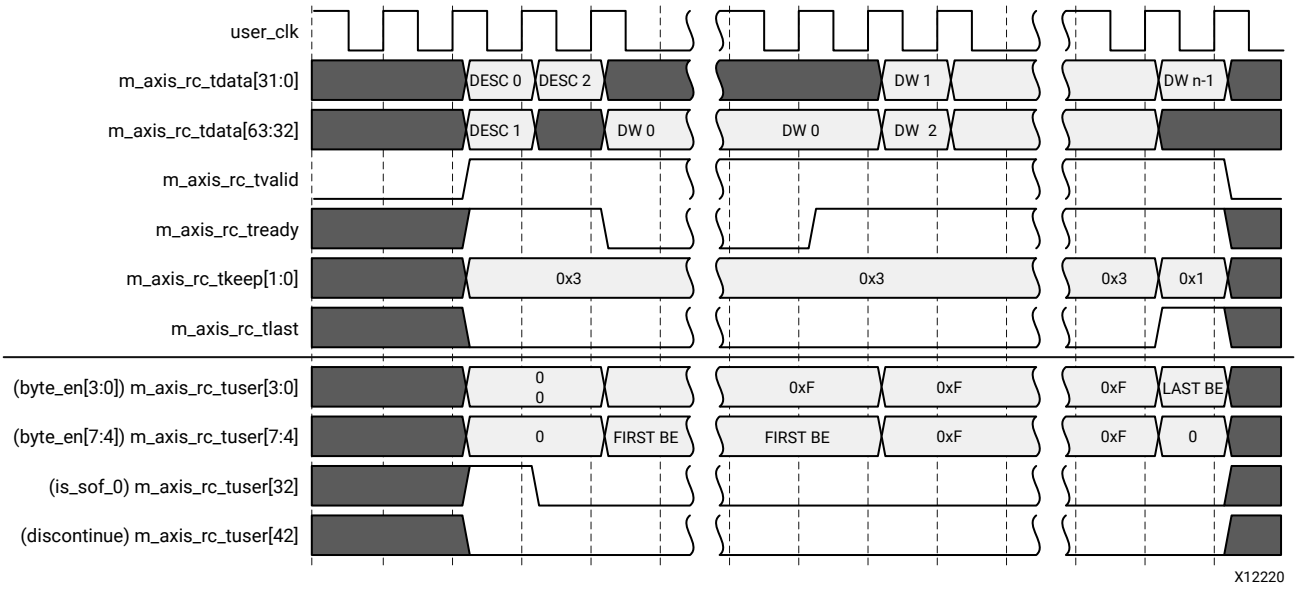


Figure 64: **Transfer of a Completion with Data on the Requester Completion Interface (Address-Aligned Mode, 128-Bit Interface)**

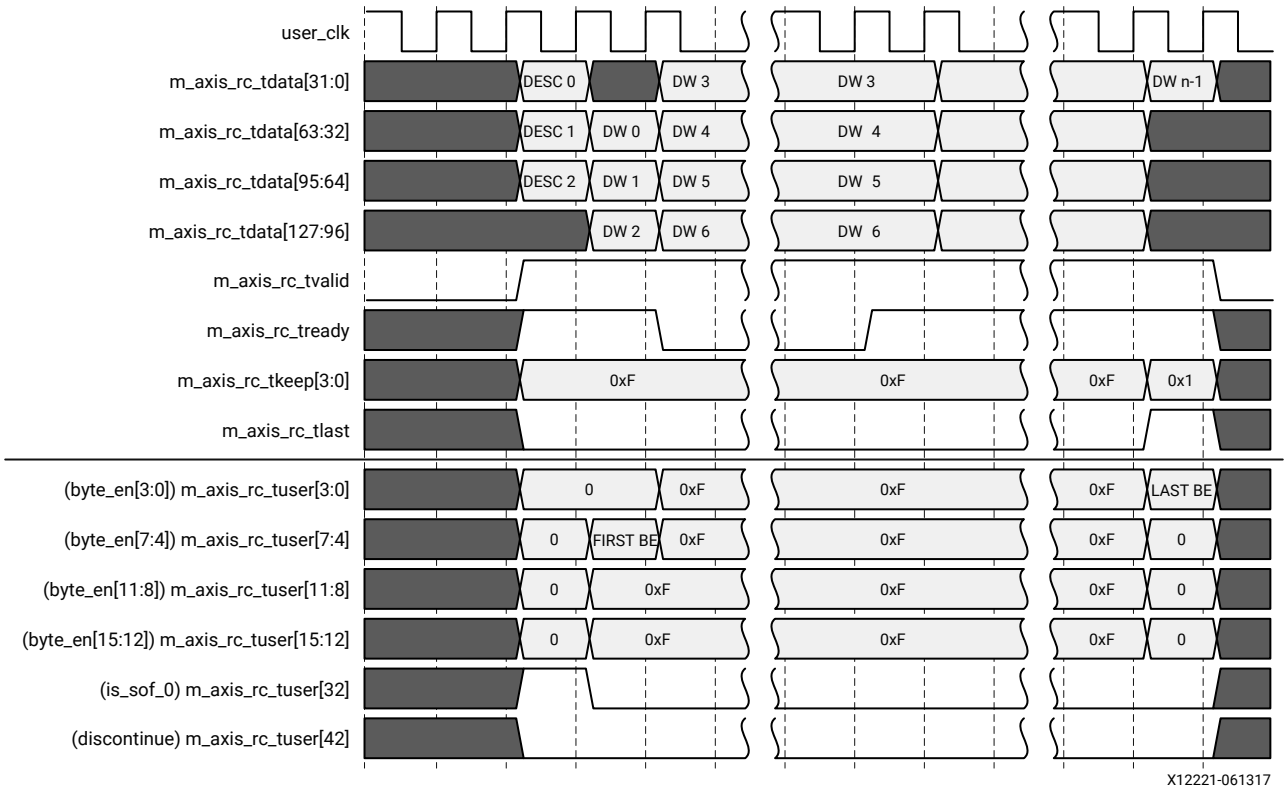
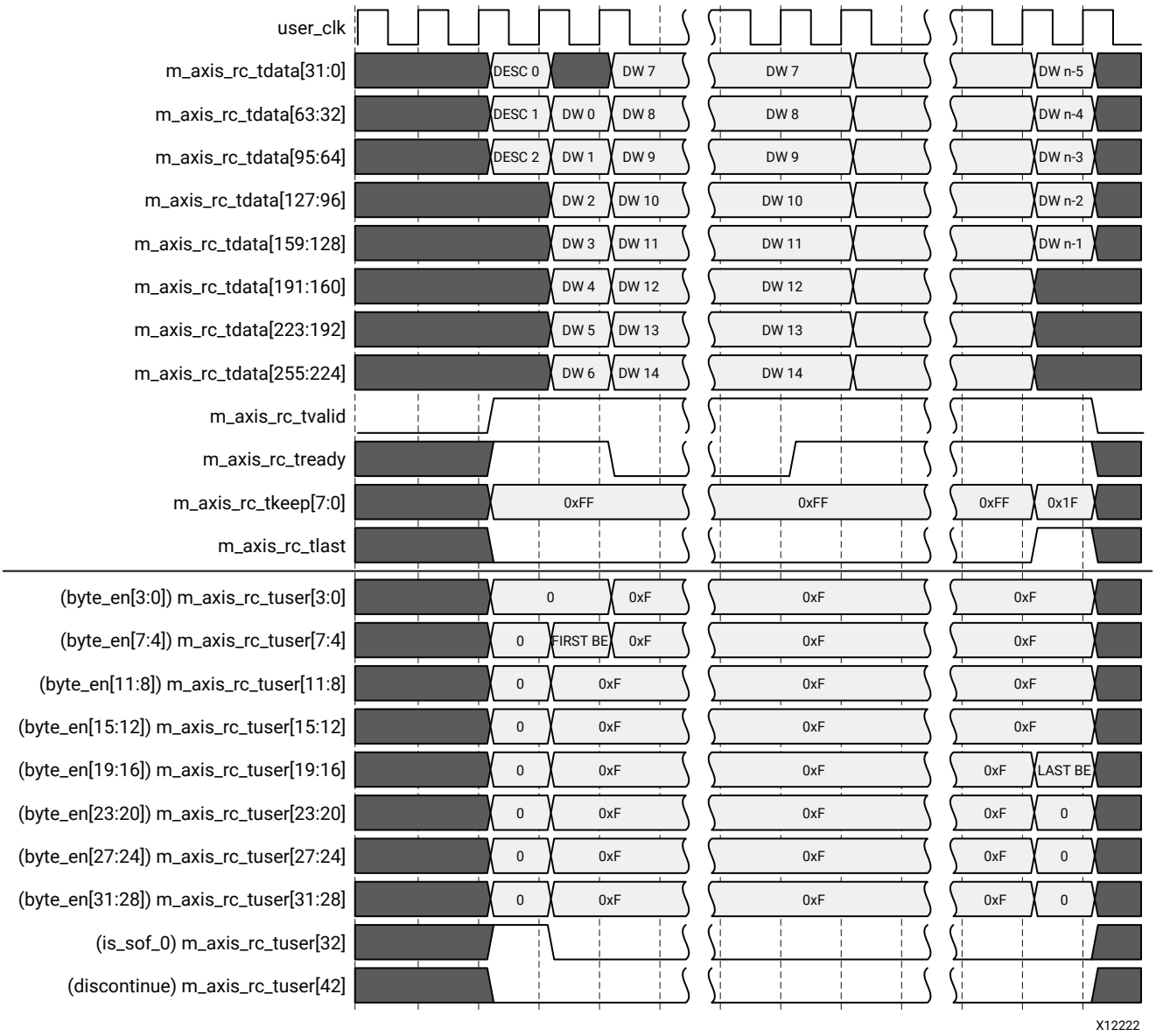


Figure 65: **Transfer of a Completion with Data on the Requester Completion Interface (Address-Aligned Mode, 256-Bit Interface)**



Straddle Option for 256-Bit Interface

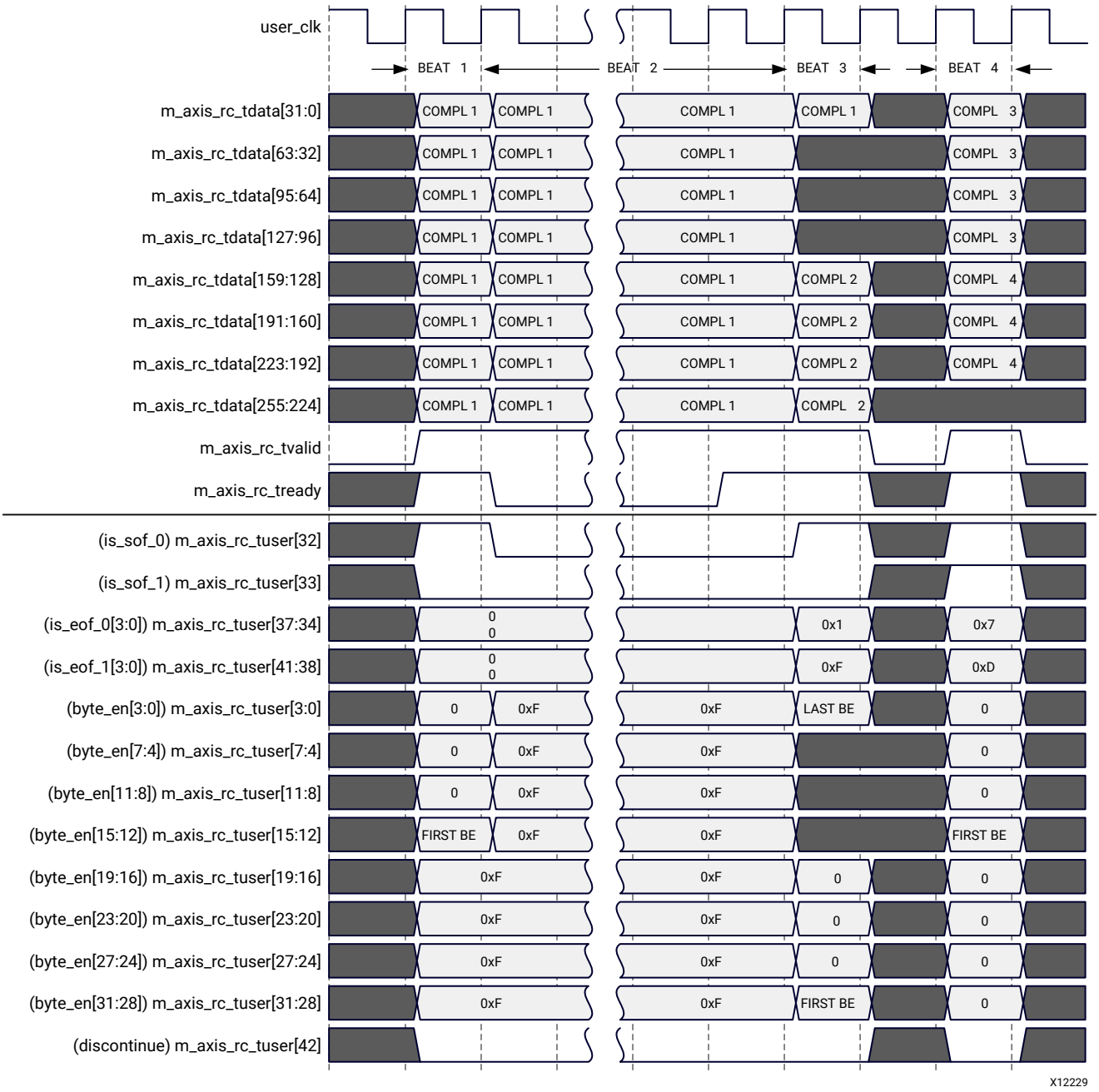
When the interface width is configured as 256 bits, the integrated block can start a new Completion transfer on the RC interface in the same beat when the previous Completion has ended on or before Dword position 3 on the data bus. The straddle option can be used only with the Dword-aligned mode.

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

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

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

Figure 66: **Transfer of Completion TLPs on the Requester Completion Interface with the Straddle Option Enabled**



Aborting a Completion Transfer

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

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

Handling of Completion Errors

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

Table 41: **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 41: **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.

Power Management

The core supports these power management modes:

- Active State Power Management (ASPM)
- Programmed Power Management (PPM)

Implementing these power management functions as part of the PCI Express design enables the PCI Express hierarchy to seamlessly exchange power-management messages to save system power. All power management message identification functions are implemented. The subsections in this section describe the user logic definition to support the above modes of power management.

For additional information on ASPM and PPM implementation, see the PCI Express Base Specification.

Active State Power Management

The core advertises an N_FTS value of 255 to ensure proper alignment when exiting L0s. If the N_FTS value is modified, you must ensure enough FTS sequences are received to properly align and avoid transition into the Recovery state.

The Active State Power Management (ASPM) functionality is autonomous and transparent from a user-logic function perspective. The core supports the conditions required for ASPM. The integrated block supports ASPM L0s and ASPM L1. L0s and L1 should not be enabled in parallel.

***Note*:** ASPM is not supported in non-synchronous clocking mode.

***Note*:** L0s is not supported for Gen3 capable designs. It is supported only on designs generated for Gen1 and Gen2.

Programmed Power Management

To achieve considerable power savings on the PCI Express hierarchy tree, the core supports these link states of Programmed Power Management (PPM):

- L0: Active State (data exchange state)
- L1: Higher Latency, lower power standby state
- L3: Link Off State

The Programmed Power Management Protocol is initiated by the Downstream Component/ Upstream Port.

PPM L0 State

The L0 state represents *normal* operation and is transparent to the user logic. The core reaches the L0 (active state) after a successful initialization and training of the PCI Express Link(s) as per the protocol.

PPM L1 State

These steps outline the transition of the core to the PPM L1 state:

1. The transition to a lower power PPM L1 state is always initiated by an upstream device, by programming the PCI Express device power state to D3-hot (or to D1 or D2, if they are supported).
2. The device power state is communicated to the user logic through the `cfg_function_power_state` output.
3. The core then throttles/stalls the user logic from initiating any new transactions on the user interface by deasserting `s_axis_rq_tready`. Any pending transactions on the user interface are, however, accepted fully and can be completed later.

   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 the following table) and expects a `cfg_power_state_change_ack` back from the user logic.

3. A successful handshake results in a transmission of the Power Management Turn-off Acknowledge (PME-turnoff_ack) Message by the core to its upstream link partner.
4. The core closes all its interfaces, disables the Physical/Data-Link/Transaction layers and is ready for *removal* of power to the core.

   There are two exceptions to this rule:

   - The core is configured as an Endpoint and the User Configuration Space is enabled. In this situation, the user application must refrain from sending new Request TLPs if *cfg_function_power_state* indicates non-D0, but the user application can return Completions to Configuration transactions targeting User Configuration space.
   - The core is configured as a Root Port. To be compliant in this situation, the user application should refrain from sending new Requests if `cfg_function_power_state` indicates non-D0.

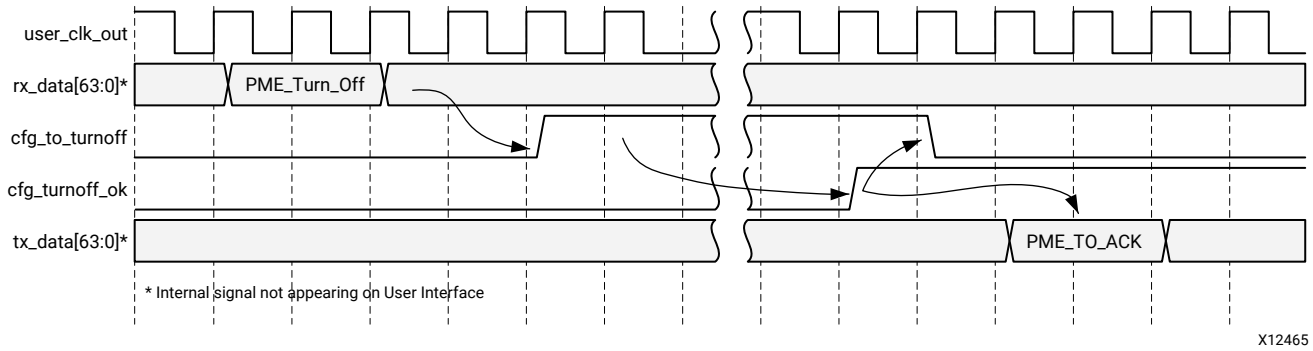
Table 42: **Power Management Handshaking Signals**

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 67: **Power Management Handshaking: 64-Bit**



Generating Interrupt Requests

See the `cfg_interrupt_msi_*` and `cfg_interrupt_msix_*` descriptions in Configuration Interrupt Controller Interface.

***Note*:** This section only applies to the Endpoint Configuration of the UltraScale Devices Gen3 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. The following table describes the Interrupt Mode to which the device has been programmed, based on the `cfg_interrupt_msi_enable` and `cfg_interrupt_msix_enable` outputs of the core.

Table 43: **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 Configuration Status Interface.

The user application requests interrupt service in one of two ways, each of which are described in the following section.

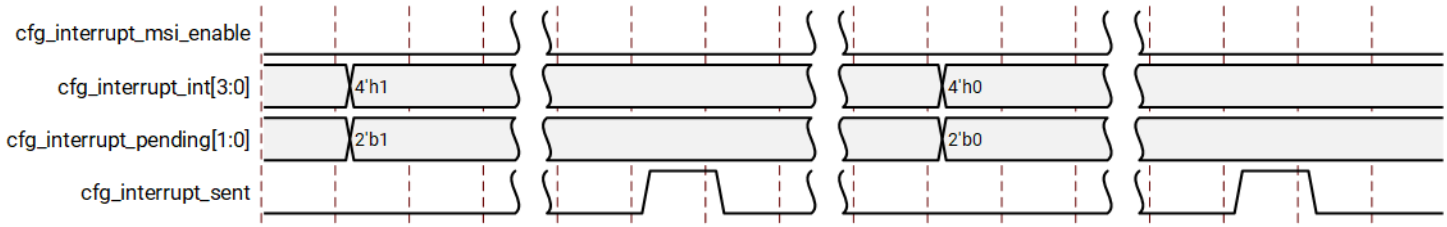
Legacy Interrupt Mode

- 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). After the interrupt has been serviced, the user application deasserts `cfg_interrupt_int`.
- After the user application deasserts `cfg_interrupt_int`, the core sends a deassert interrupt message (Deassert_INTA). This is indicated by the assertion of `cfg_interrupt_sent` a second time.

`cfg_interrupt_int` must be asserted until the user application receives confirmation of the assert interrupt message (Assert_INTA), which is indicated by the assertion of `cfg_interrupt_sent` , and the interrupt has been serviced/cleared by the Root's Interrupt Service Routine (ISR). Deasserting `cfg_interrupt_int` causes the core to send the deassert interrupt message (Deassert_INTA). `cfg_interrupt_pending` must be asserted along with the assertion of `cfg_interrupt_int` until the interrupt has been serviced, otherwise, the

interrupt status bit in the status register is not updated correctly. `cfg_interrupt_pending` can be deasserted along with the deassertion of `cfg_interrupt_int` after the first assertion of `cfg_interrupt_sent`. When the software/Root's ISR receives an assert interrupt message, it reads this interrupt status bit to determine whether there is an interrupt pending for this function.

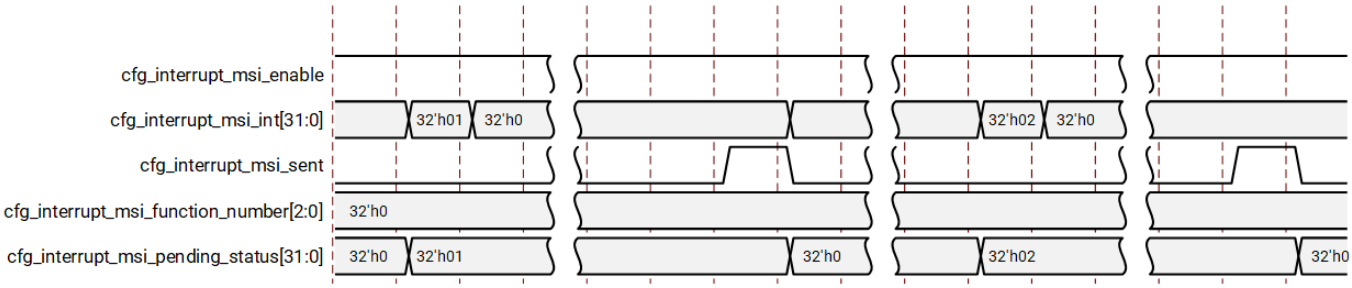
Figure 68: **Legacy Interrupt Signaling**



MSI Mode

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

Figure 69: **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#}};`

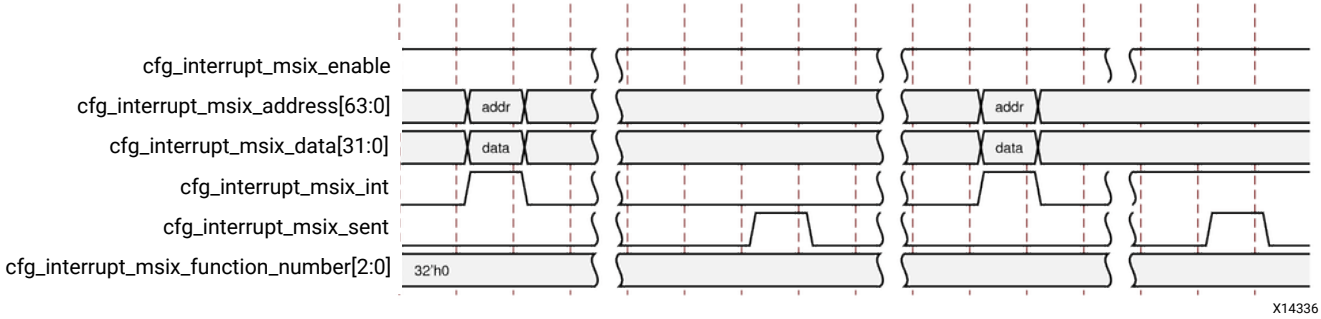
   where MSI_Vector# is a 5-bit value and is allowed to be `00000b` ≤ MSI_Vector# ≤ `11111b`.

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 Gen3 Integrated Block for PCIe core optionally supports the MSI-X interrupt and its signaling, which is shown in the following figure. The MSI-X vector table and the MSI-X Pending Bit Array need to be implemented as part of the user logic, by claiming a BAR aperture.

Figure 70: **MSI-X Mode**



Receive Message Interface

The core provides a separate receive-message interface which the user application can use to receive indications of messages received from the link. When the receive message interface is enabled, the integrated block signals the arrival of a message from the link by setting the `cfg_msg_received_type[4:0]` output to indicate the type of message (see the following table) and pulsing the `cfg_msg_received` signal for one or more cycles. The duration of assertion of `cfg_msg_received` is determined by the type of message received (see Table 45: Message Parameters on Receive Message Interface). When `cfg_msg_received` is active-High, the integrated block transfers any parameters associated with the message on the bus 8 bits at a time on the bus `cfg_msg_received_data`. The parameters transferred on this bus in each cycle of

cfg_msg_received

assertion for various message types are listed in Table 45: Message Parameters on Receive Message Interface. 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 44: **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 44: **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 45: **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

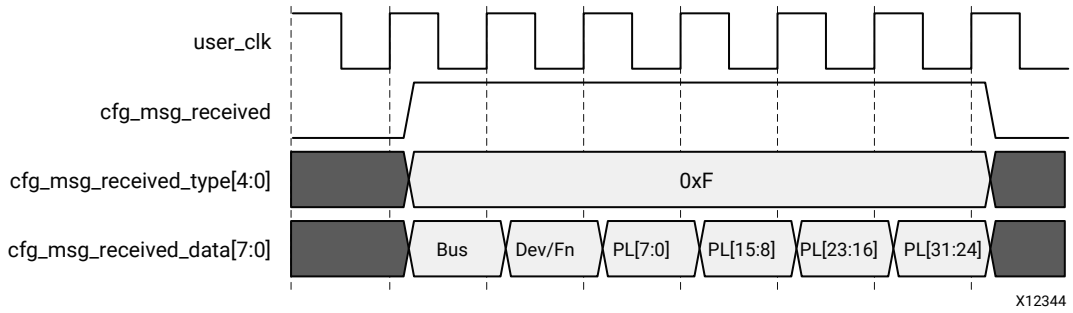
Table 45: **Message Parameters on Receive Message Interface** *(cont'd)*

Message Type	Number of Cycles of cfg_msg_received Assertion	Parameter Transferred on cfg_msg_received_data[7:0]
Optimized Buffer Flush/Fill (OBFF)	3	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number Cycle 3: OBFF Code
Unlock	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
Vendor_Defined Type 0	4 cycles when no data present, 8 cycles when data present.	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number Cycle 3: Vendor ID[7:0] Cycle 4: Vendor ID[15:8] Cycle 5: bits [7:0] of payload Cycle 6: bits [15:8] of payload Cycle 7: bits [23:16] of payload Cycle 8: bits [31:24] of payload
Vendor_Defined Type 1	4 cycles when no data present, 8 cycles when data present.	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number Cycle 3: Vendor ID[7:0] Cycle 4: Vendor ID[15:8] Cycle 5: bits [7:0] of payload Cycle 6: bits [15:8] of payload Cycle 7: bits [23:16] of payload Cycle 8: bits [31:24] of payload
ATS Invalid Request	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
ATS Invalid Completion	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
ATS Page Request	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
ATS PRG Response	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number

The following figure 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 71: **Receive Message Interface**



The integrated block inserts a gap of at least one clock cycle between successive pulses on the `cfg_msg_received` output. There is no mechanism to apply back pressure on the message indications delivered through the receive message interface. When using this interface, the user logic must always be ready to receive message indications.

Configuration Management Interface

The ports used by configuration registers are described in Configuration Status Interface. Root Ports must use the Configuration Port to set up the Configuration Space. Endpoints can also use the Configuration Port to read and write; however, care must be taken to avoid adverse system side effects.

The user application must supply the address as a Dword address, not a byte address.

TIP: *To calculate the Dword address for a register, divide the byte address by four.*

For example:

For the Command/Status register in the PCI Configuration Space Header:

- The Dword address of is `01h`.

  ***Note*:** The byte address is `04h`.

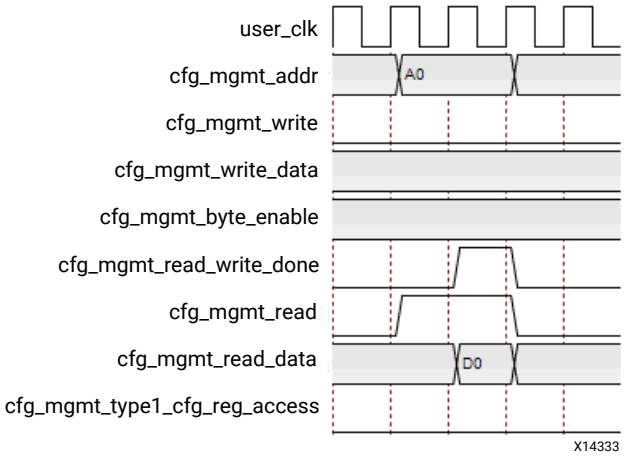
For BAR0:

- The Dword address is `04h`.

  ***Note*:** The byte address is `10h`.

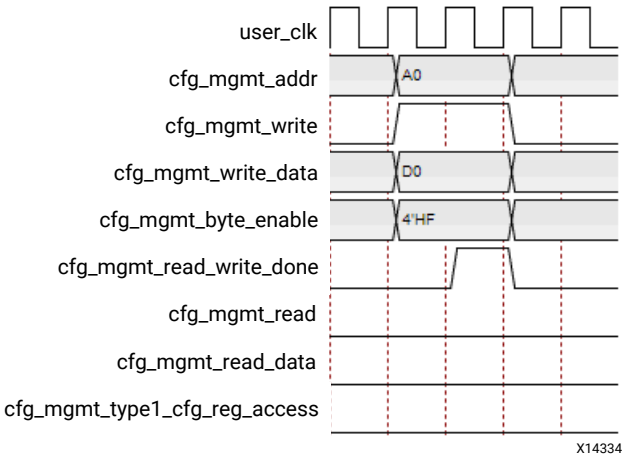
To read any register in configuration space, the user application drives the register Dword address onto `cfg_mgmt_addr[9:0]`. `cfg_mgmt_addr[17:10]` selects the PCI Function associated with the configuration register. The core drives the content of the addressed register onto cfg_mgmt_read_data[31:0]. The value on `cfg_mgmt_read_data[31:0]` is qualified by signal assertion on `cfg_mgmt_read_write_done`. The following figure illustrates an example with read from the Configuration Space.

Figure 72: **cfg_mgmt_read_type0_type1**



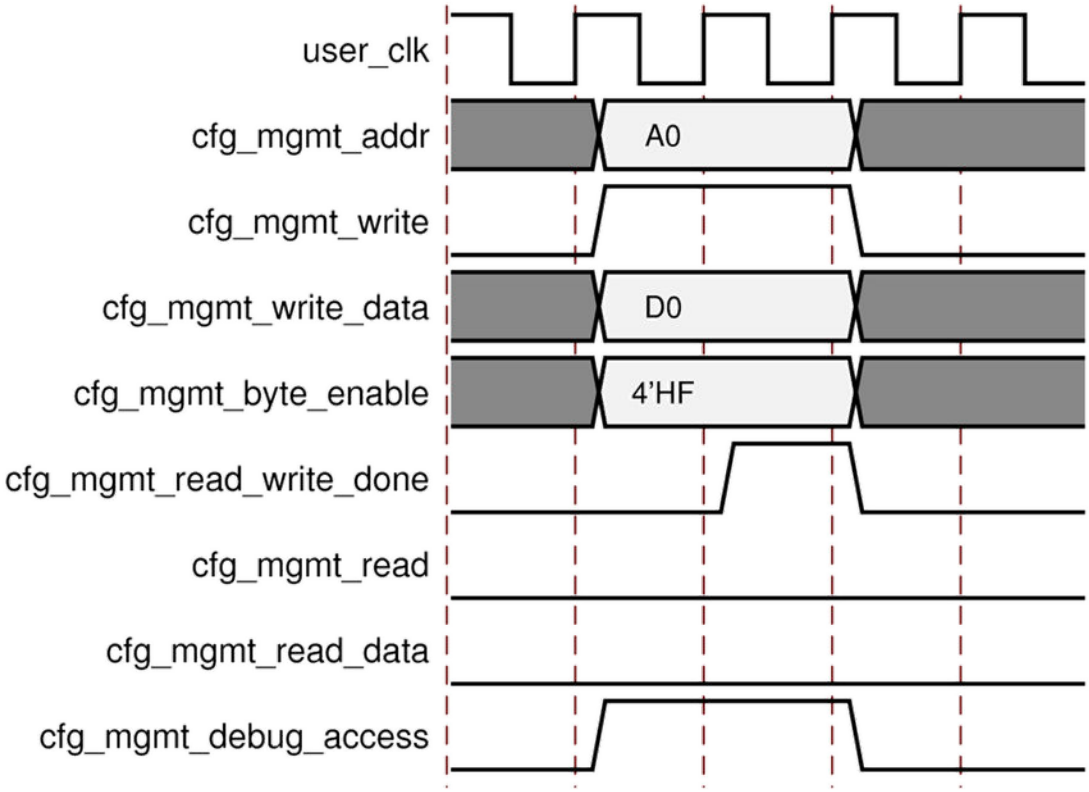
To write any register in configuration space, the user logic places the address on `cfg_mgmt_addr` bus, write data on `cfg_mgmt_write_data` , byte-valid on `cfg_mgmt_byte_enable [3:0]`, and asserts the `cfg_mgmt_write` signal. In response, the core asserts the `cfg_mgmt_read_write_done` signal when the write is complete (which can take several cycles). The user logic must keep `cfg_mgmt_addr`, `cfg_mgmt_write_data`, `cfg_mgmt_byte_enable` and c `fg_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 73: **cfg_mgmt_write_type0**



When the core is configured in the Root Port mode, when you assert cfg_mgmt_type1_cfg_reg_access input during a write to a Type-1 PCI™ Configuration register forces a write into certain read-only fields of the register.

Figure 74: **cfg_mgmt_debug_access**



Link Training: 2-Lane, 4-Lane, and 8-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. Two cases cause core to operate at less than its specified maximum lane width, as defined in these subsections.

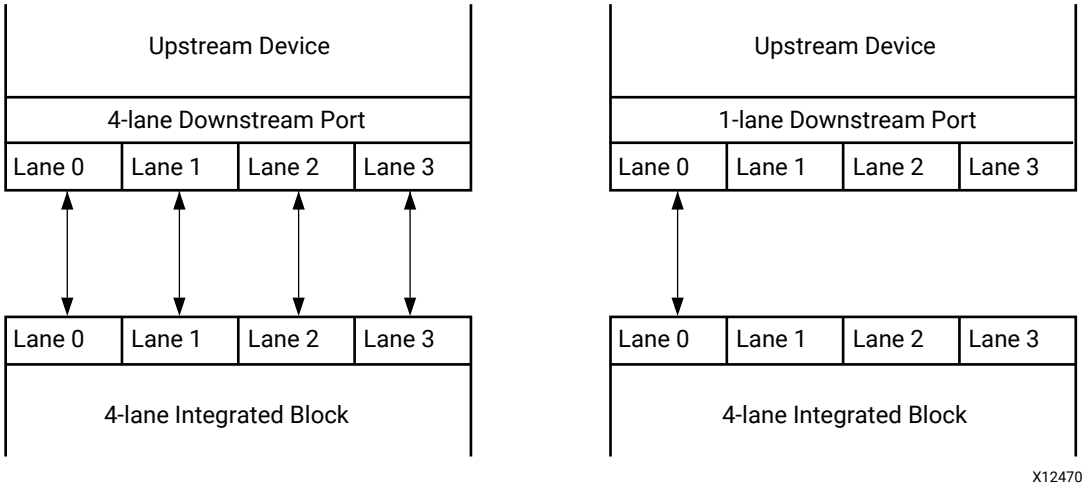
Link Partner Supports Fewer Lanes

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 the following figure. Similarly, if the 4-lane core is connected to a 2-lane device, the core trains and operates as a 2-lane device using lanes 0 and 1.

When the 8-lane core is connected to a device that only implements 4 lanes, it trains and operates as a 4-lane device using lanes 0-3. Additionally, if the connected device only implements 1 or 2 lanes, the 8-lane core trains and operates as a 1- or 2-lane device.

Figure 75: **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 x8 and x4 (excluding downshift modes). Downshift refers to the link width negotiation process that occurs when link partners have different lane width capabilities advertised. As a result of lane width negotiation, the link partners negotiate down to the smaller of the advertised lane widths. The following table describes the several possible combinations including downshift modes and availability of lane reversal support.

Table 46: **Lane Reversal Support**

Integrated Block Advertised Lane Width	Negotiated Lane Width	Lane Number Mapping (Endpoint c)		Lane Reversal SupportedLane 0... Lane 7
		Endpoint	Link Partner	
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

Table 46: **Lane Reversal Support** *(cont'd)*

Integrated Block Advertised Lane Width	Negotiated Lane Width	Lane Number Mapping (Endpoint c)		Lane Reversal SupportedLane 0... Lane 7
		Endpoint	Link Partner	
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.

Design Flow Steps

This section describes customizing and generating the core, constraining the core, and the simulation, synthesis, and implementation steps that are specific to this IP core. More detailed information about the standard AMD Vivado™ design flows and the IP integrator can be found in the following Vivado Design Suite user guides:

- *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
- *Vivado Design Suite User Guide: Designing with IP* (UG896)
- *Vivado Design Suite User Guide: Getting Started* (UG910)
- *Vivado Design Suite User Guide: Logic Simulation* (UG900)

Customizing and Generating the Core

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

If you are customizing and generating the core in the Vivado IP integrator, see the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) for detailed information. IP integrator might auto-compute certain configuration values when validating or generating the design. To check whether the values do change, see the description of the parameter in this chapter. To view the parameter value, run the `validate_bd_design` command in the Tcl console.

You can customize the IP for use in your design by specifying values for the various parameters associated with the IP core using the following steps:

1. Select the IP from the IP catalog.
2. Double-click the selected IP or select the Customize IP command from the toolbar or right-click menu.

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) and the *Vivado Design Suite User Guide: Getting Started* (UG910).

Figures in this chapter are illustrations of the Vivado IDE. The layout depicted here might vary from the current version.

Basic Mode Parameters

The Basic mode parameters are explained in this section.

Basic Tab

The initial customization page is used to define the core basic parameters, including the component name, reference clock frequency, and silicon type.

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.

PCIe 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.

This option is not available if an AMD Development Board is selected.

Number of Lanes

The core requires the selection of the initial lane width. The following table 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, and 8-Lane Components for more information.

Table 47: **Lane Width and Product Generated**

Lane Width	Product Generated
x8	8-Lane UltraScale FPGA Gen3 Integrated Block for PCI Express

Maximum Link Speed

The core allows you to select the Maximum Link Speed supported by the device. The following table 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 48: **Lane Width and Link Speed**

Lane Width	Link Speed
x8	8 Gb/s

AXI-ST Interface Width

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

Table 49: **Lane Width, Link Speed, and Interface Width**

Lane Width	Link Speed (Gb/s)	Interface Width (Bits)
x8	8.0	256

AXI-ST Interface Frequency

The frequency is set to 250 MHz.

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. See Data Alignment Options.

Enable AXI-ST Frame Straddle

The core provides an option to straddle packets on the requester completion interface when the interface width is 256 bits. See Straddle Option for 256-Bit Interface.

Enable Client Tag

Enables you to use the client tag.

Reference Clock Frequency

Selects the frequency of the reference clock provided on `sys_clk`. For important information about clocking the core, see Clocking.

AMD Development Board

Selects the AMD Development Board to enable the generation of AMD Development Board-specific constraints files.

Enable External PIPE Interface

When selected, this option enables an external third-party bus functional model (BFM) to connect to the PIPE interface of integrated block for PCIe. For details, see *PIPE Mode Simulation Using Integrated Endpoint PCI Express Block in Gen2 x8 and Gen3 x8 Configurations* (XAPP1184) , 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 AMD UltraScale™ device core to third-party BFMs.

Additional Transceiver Control and Status Ports

When this option is selected, transceiver debug and status ports are brought to the core boundary level.

Capabilities Tab

The Capabilities settings are explained in this section.

Enable Physical Function 0

The core implements an additional physical function (PF).

The integrated block implements up to six virtual functions that are associated to PF0 (if enabled).

PF0 Legacy Interrupt PIN

This parameter allows you to select the Interrupt pin INTA, INTB, INTC and INTD. The default value is INTA.

MPS

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

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)

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.

Identity Settings (PF0 IDs and PF1 IDs) Tab

The Identity Settings customize the IP initial values, class code, and Cardbus CIS pointer information. The page for physical function 1 (PF1) is only displayed when PF1 is enabled.

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 AMD. Enter a vendor identification number here. `FFFFh` is reserved.
- **Device ID:** A unique identifier for the application; the default value, which depends on the configuration selected, is 70<*link speed* ><*link width* >h. This field can be any value; change this value for the application.

  The Device ID parameter is evaluated based on:

  - The device family (9 for UltraScale+, 8 for UltraScale, 7 for 7 series devices),
  - EP or RP mode,
  - Link width, and
  - Link speed.

    If any of the above values are changed, the Device ID value will be re-evaluated, replacing the previous set value.

    RECOMMENDED: *It is always recommended that the link width, speed and Device Port type be changed first and then the Device ID value. Make sure the Device ID value is set correctly before generating the IP.*
- **Revision ID:** Indicates the revision of the device or application; an extension of the Device ID. The default value is `00h`; enter 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.

Base Address Registers (PF0 and PF1) Tab

The Base Address Registers (BARs) page 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.

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 256 gigabytes. 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. The following table lists the available BAR size ranges.

Table 50: **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
Root port of PCI Express Root Complex	32-bit Memory	4 B – 2 GB
	64-bit Memory	4 B – 8 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, 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 Capabilities Tab

On this page, you set the Legacy Interrupt Settings and MSI Capabilities for all applicable physical and virtual functions.

Legacy Interrupt Settings

- **Enable MSI Per Vector Masking:** Enables MSI Per Vector Masking Capability of all the Physical functions enabled.

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

- **PF0/PF1 Interrupt PIN:** Indicates the mapping for Legacy Interrupt messages. A setting of *None* indicates that no Legacy Interrupts are used.

MSI Capabilities

- **PF0/PF1 Enable MSI Capability Structure:** Indicates that the MSI Capability structure exists.

  ***Note*:** Although it is possible to not enable MSI or MSI-X, the result would be a non-compliant core. The PCI Express Base Specification requires that MSI, MSI-X, or both be enabled.

- **Multiple Message Capable:** Selects the number of MSI vectors to request from the Root Complex.

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 includes some additional settings. The following parameters are on the Basic page when the Advanced mode is selected.

Enable Parity

Enables the Parity feature of the UltraScale IP when checked. The two model parameters AXISTEN_IF_CC_PARITY_CHK and AXISTEN_IF_RQ_PARITY_CHK are set to TRUE. The default value of this parameter is FALSE.

Use the dedicated PERST routing resources

Enables sys_rst dedicated routing for applicable PCIe locations (see Available Integrated Blocks for PCI Express).

System reset polarity

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

PCIe DRP Ports

When checked, enables the PCIe DRP interface.

The signals in the following table are available when PCIe DRP Ports option is selected.

Table 51: **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
drp_clk	I	1 bit	drp_clk used for drp interface, frequencies supported are 62.5 MHz, 125 MHz and 250 MHz

GT Channel DRP

When checked, enables the GT channel DRP interface.

The signals shown in the following table are available when GT Channel DRP parameter is enabled.

Table 52: **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

Enable RX Message INTFC

When checked, messages are routed to the `cfg_msg_received` signal at the Receive Message Interface. Otherwise, they are routed to the CQ Interface

Enable GT Quad Selection

This parameter is used to enable the device/package migration. See Package Migration of UltraScale Devices PCI Express Designs.

GT Quad

This parameter has drop-down menu to select the desired GTH quad. This is available only when *Enable GT Quad Selection* is checked. For more information about the available GT Quad for each device, see Appendix B: GT Locations.

CORE CLOCK Frequency

This parameter allows you to select the core clock frequencies.

For Gen3 link speed:

- The values of 250 MHz and 500 MHz are available for selection for speed grades -1, -2, -3, -1H and -1HV, and for a link width other than x8. For this configuration, this parameter is available when *Advanced* mode is selected.
- For speed grades -1, -2, -3, -1H and -1HV, and for a link width of x8, this parameter defaults to 500 MHz and is not available for selection.
- For a -1L or -1LV speed grade and a link width other than x8, this parameter defaults to 250 MHz and is not available for selection.

For Gen1 and Gen2 link speeds:

- This parameter defaults to 250 MHz and is not available for selection.

***Note*:** When a -1L or -1LV speed grade is selected, and non production parts of XCVU440 (ES2), XCKU060 (ES2) and XCKU115 (ES2) is selected, this parameter defaults to 250 MHz and is not available for selection.

Capabilities Tab

The Capabilities settings for Advanced mode contains three additional parameters to those for Basic mode and are described below.

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 for both PF0 and PF1 (if selected).

Function Level Reset

Indicates that the Function Level Reset is enabled. You can reset a specific device function. This applicable only to Endpoint configurations.

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 for more information. These settings apply to both physical functions if PF1 is enabled.

PF0 ID and PF1 ID Tab

The Identity settings (PF0 and PF1 Initial ID) are the same for both Basic and Advanced modes.

PF0 BAR and PF1 BAR Tab

The PF0 and PF1 BAR settings are the same for both Basic and Advanced modes.

SRIOV Config (PF0 and PF1) Tab

SRIOV Capability Version

Indicates the 4-bit SR-IOV Capability version for the physical function.

SRIOV Function Select

Indicates the number of virtual functions associated to the physical function. A total of six virtual functions are available to PF0 and PF1.

SRIOV Functional 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.

SRIOV 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
```

SRIOV 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.

PF0 SRIOV BARs and PF1 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.

Table 53: **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. The following table lists the available BAR size ranges.

Table 54: **SRIOV BAR Size Ranges for Device Configuration**

PCIe Device / Port Type	BAR Type	BAR Size Range
PCI Express Endpoint	32-bit Memory	128 bytes – 2 gigabytes
	64-bit Memory	128 bytes – 256 gigabytes
Legacy PCI Express Endpoint	32-bit Memory	16 bytes – 2 gigabytes
	64-bit Memory	16 bytes – 256 gigabytes
	I/O	16 bytes – 2 gigabytes

- **Prefetchable:** Identifies the ability of the memory space to be prefetched.
- **Value:** The value assigned to the BAR based on the current selections.

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.

Legacy/MSI Capabilities Tab

This page is the same as that of Basic mode.

MSI-X Capabilities Tab

The MSI-X Capabilities parameters are available in Advanced mode only.

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 Size field is expecting N-1 interrupts (0x0F will configure a table count of 16).
- **Table Offset:** Specifies the offset from the Base Address Register that points to the base of the MSI-X Table.
- **BAR Indicator:** Indicates the Base Address Register in the Configuration Space used to map the function in the MSI-X Table onto memory space. For a 64-bit Base Address Register, this indicates the lower DWORD.

MSIx Pending Bit Array (PBA) Settings

Defines the MSI-X Pending Bit Array (PBA) structure.

- **PBA Offset:** Specifies the offset from the Base Address Register that points to the base of the MSI-X PBA.
- **PBA BAR Indicator:** Indicates the Base Address Register in the Configuration Space used to map the function in the MSI-X PBA onto Memory Space.

Power Management

The Power Management page includes settings for the Power Management registers, power consumption, and power dissipation options. These settings apply to both physical functions, if PF1 is enabled.

- **D1 Support:** Indicates that the function supports the D1 Power Management State. See section 3.2.3 of the PCI Bus Power Management Interface Specification Revision 1.2.
- **PME Support From:** Indicates the power states in which the function can assert cfg_pm_wake. See section 3.2.3 of the PCI Bus Power Management Interface Specification Revision 1.2.
- **Block RAM Configuration Options:** Specify the number of receive block RAMs used for the solution. The table displays the number of receiver credits available for each packet type.

Extended Capabilities 1 and Extended Capabilities 2

The PCIe Extended Capabilities allow you to enable PCI Express Extended Capabilities. The Advanced Error Reporting Capability (offset 0x100 h) is always enabled. The Customize IP dialog box sets up the link list based on the capabilities enabled. After enabling, you must configure the capability by setting the applicable attributes in the core top-level defined in Output Generation.

- **Device Serial Number Capability:** An optional PCIe Extended Capability containing a unique Device Serial Number. When this Capability is enabled, the DSN identifier must be presented on the Device Serial Number input pin of the port. This Capability must be turned on to enable the Virtual Channel and Vendor Specific Capabilities
- **Virtual Channel Capability:** An optional PCIe Extended Capability which allows the user application to be operated in TCn/VC0 mode. Checking this allows Traffic Class filtering to be supported. This capability only exists for physical function 0.
- **Reject Snoop Transactions (Root Port Configuration Only):** When enabled, any transactions for which the No Snoop attribute is applicable, but is not set in the TLP header, can be rejected as an Unsupported Request.
- **Enable AER Capability:** Optional PCIe Extended Capability that allows Advanced Error Reporting. This capability is always enabled.
  - **ECRC check capable:** If *ECRC check capable is* enabled, the core does not send the TLP with ECRC error to the user logic and it is discarded.

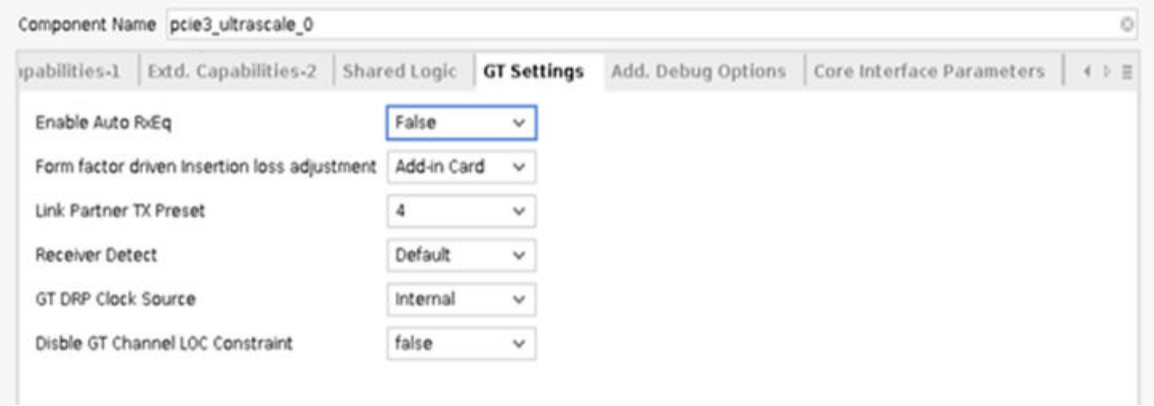
Additional Optional Capabilities

- **Enable ARI:** Allows Alternate Requester ID. This capability is automatically enabled and should not be disabled if SRIOV is enabled.
- **Enable PB:** Implements the Power Budgeting Enhanced capability header.
- **Enable LTR:** Implements the Latency Tolerance Reporting capability.
- **Enable DPA:** Implements Dynamic Power Allocation capability.
- **Enable TPH:** Implements Transaction Processing Hints capability.

GT Settings Tab

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

Figure 76: **GT Settings Tab**



PLL Selection

(Only available when Gen2 link speed is selected), allows for either the QPLL1 or CPLL to be selected as the clock source. This feature is useful when additional protocols are desired to be in the same GT Quad when operating at Gen2 links speeds. Gen3 speeds require the QPLL1, and Gen1 speeds always use the CPLL.

IMPORTANT! *The rest of the settings should not be modified unless instructed to do so by AMD.*

The following table shows the options and default for each line speed.

Table 55: **PLL Type**

Link Speed	PLL Type	Comments
2.5_GT/s	CPLL	The default is CPLL, and not available for selection.

Table 55: **PLL Type** *(cont'd)*

Link Speed	PLL Type	Comments
5.0_GT/s	QPLL1, CPLL	The default is QPLL1, and available for selection.
8.0_GT/s	QPLL1	The default is QPLL1, and not available for selection.

Enable Auto RxEq

When this parameter is set to True, it auto selects the Receiver Equalization (LPM or DFE) mode.

Table 56: **Determining the Receiver Equalization (LPM or DFE) Mode**

Auto RxEQ	Behavior
True	The default is DFE but it will change LPM based on the channel characteristics.
False	The default is DFE and can be changed by setting the Form Factor Driven Insertion Loss Adjustment.

Form Factor Driven Insertion Loss Adjustment

Indicates the transmitter to receiver insertion loss at the Nyquist frequency depending on the form factor selection. Three options are provided:

- **Chip-to-Chip:** The value is 5 dB (LPM).
- **Add-in Card:** The value is 15 dB and is the default option (DFE).
- **Backplane:** The value is 20 dB (DFE).

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

***Note*:** The options above are for Gen3 only. For Gen1/Gen2 designs, LPM is automatically for all Form Factor Driven Insertion Loss Adjustment selections.

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.

Receiver Detect

Indicates the type of Receiver Detect Default or Falling Edge. This parameter is available on the GT Settings Tab when Advanced mode is selected. This parameter is available only for Production devices. When the Falling Edge option is selected, the GT Channel DRP Parameter on the Basic tab (in Advanced mode) is disabled. For more information about this option, see the *UltraScale Architecture GTH Transceivers User Guide* (UG576).

GT DRP Clock Source

This option is added to select the GT clock source external or internal. When external source is selected, the DRP clock is supplied from an external clock source of 300 MHz, and it is divided into 100/125 MHz in the AMD top module. The default GT DRP clock source is internal.

Disable GT Channel LOC Constraints

This option disables GT channel LOC constraints in the GT Wizard IP level XDC file. So that you can LOC in their top XDC file.

Shared Logic

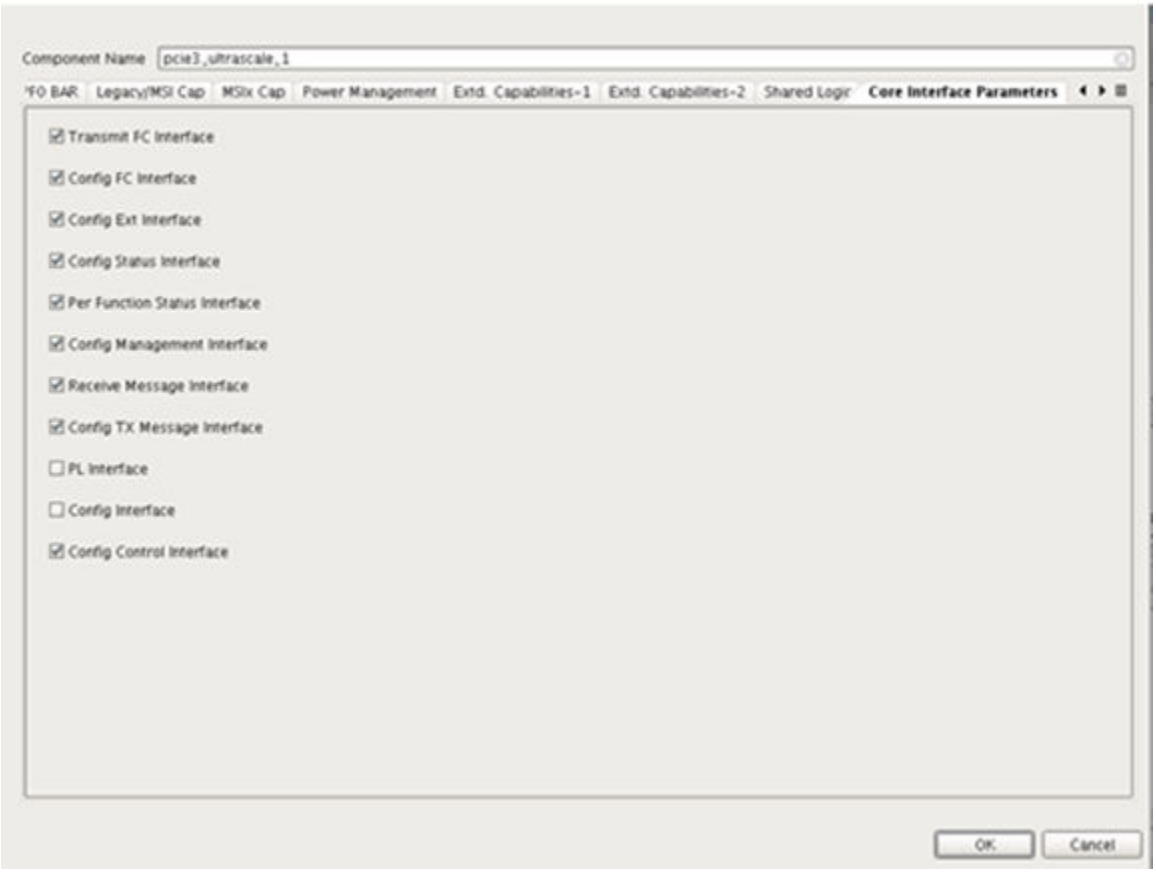
Enables you to share common blocks across multiple instantiations by selecting one or more of the options on this page. For more information, see Shared Logic.

Core Interface Parameters

You can select the core interface parameters. By default, all ports are brought out. You can disable some interfaces if they are not used. When disabled, the interfaces (ports) are removed from the core top.

RECOMMENDED: *For a typical use case, do not disable the interfaces. Disable the ports only in special cases.*

Figure 77: **Core Interfaces Parameters**



Transmit FC Interface

Enables you to request which flow control information the core provides. When you disable the Transmit Flow Control (FC) Interface option, the following ports are removed:

- pcie_tfc_nph_av
- pcie_tfc_npd_av

Config FC Interface

Enables you to control the configuration flow control for the core. When you disable the Config Flow Control (FC) Interface option, the following ports are removed from the core:

- cfg_fc_ph
- cfg_fc_pd
- cfg_fc_nph
- cfg_fc_npd

- cfg_fc_cplh
- cfg_fc_cpld
- cfg_fc_sel

Config External Interface

Allows the core to transfer configuration information with the user application when externally implemented configuration registers are implemented. When you disable the Config Ext Interface option, the following ports are removed from the core:

- cfg_ext_read_received
- cfg_ext_write_received
- cfg_ext_register_number
- cfg_ext_function_number
- cfg_ext_write_data
- cfg_ext_write_byte_enable
- cfg_ext_read_data
- cfg_ext_read_data_valid

Config Status Interface

Provides information on how the core is configured. When you disable the Config Status Interface option, the following ports are removed from the core:

- cfg_phy_link_down
- cfg_phy_link_status
- cfg_negotiated_width
- cfg_current_speed
- cfg_max_payload
- cfg_max_read_req
- cfg_function_status
- cfg_vf_status
- cfg_function_power_state
- cfg_vf_power_state
- cfg_link_power_state
- cfg_err_cor_out
- cfg_err_nonfatal_out

- cfg_err_fatal_out
- cfg_ltr_enable
- cfg_ltssm_state
- cfg_rcb_status
- cfg_dpa_substate_change
- cfg_obff_enable
- cfg_pl_status_change
- cfg_tph_requester_enable
- cfg_tph_st_mode
- cfg_vf_tph_requester_enable
- cfg_vf_tph_st_mode
- pcie_rq_seq_num
- pcie_rq_seq_num_vld
- pcie_cq_np_req_count
- pcie_rq_tag
- pcie_rq_tag_vld
- pcie_cq_np_req

Per Function Status Interface

Provides status data as requested by the user application through the selected function. When you disable the Per Function Status Interface option, the following ports are removed from the core:

- cfg_per_func_status_control
- cfg_per_func_status_data

Config Management Interface

Used to read and write to the Configuration Space registers. When you disable the Config Management Interface option, the following ports are removed from the core:

- cfg_mgmt_addr
- cfg_mgmt_write
- cfg_mgmt_write_data
- cfg_mgmt_byte_enable
- cfg_mgmt_read

- cfg_mgmt_read_data
- cfg_mgmt_read_write_done
- cfg_mgmt_type1_cfg_reg_access

Receive Message Interface

Indicates to the logic that a decodable message from the link, the parameters associated with the data, and type of message have been received. When you disable the Receive Message Interface option, the following ports are removed from the core:

- cfg_msg_received
- cfg_msg_received_data
- cfg_msg_received_type

Config Transmit Message Interface

Used by the user application to transmit messages to the PCIe Gen3 core. When you disable the Config Transmit Message Interface option, the following ports are removed from the core:

- cfg_msg_transmit
- cfg_msg_transmit_type
- cfg_msg_transmit_data
- cfg_msg_transmit_done

Physical Layer Interface

The Physical Layer (PL) Interface parameter is set to false by default (unchecked), so these ports do not appear at the core boundary. To enable these ports, turn on this parameter.

- pl_eq_in_progress
- pl_eq_phase
- pl_eq_reset_eieos_count
- pl_gen2_upstream_prefer_deemph

Config Interface

This parameter is set to false by default (unchecked), so these ports do not appear at the core boundary. To enable these ports, turn on this parameter.

- conf_req_data
- conf_req_ready
- conf_req_reg_num

- conf_req_type
- conf_req_valid
- conf_resp_rdata
- conf_resp_valid

Config Control Interface

Allows a broad range of information exchange between the user application and the core. When you disable the Config Control Interface option, the following ports are removed:

- cfg_hot_reset_in
- cfg_hot_reset_out
- cfg_config_space_enable
- cfg_per_function_update_done
- cfg_per_function_number
- cfg_per_function_output_request
- cfg_dsn
- cfg_ds_port_number
- cfg_ds_bus_number
- cfg_ds_device_number
- cfg_ds_function_number
- cfg_power_state_change_ack
- cfg_power_state_change_interrupt
- cfg_err_cor_in
- cfg_err_uncor_in
- cfg_flr_done
- cfg_vf_flr_done
- cfg_flr_in_process
- cfg_vf_flr_in_process
- cfg_req_pm_transition_l23_ready
- cfg_link_training_enable

Additional Debug Options

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

Figure 78: **Additional Debug Options**

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).

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 AMD 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. Open Hardware Manger (HM) and configure the FPGA using the generated .bit and .ltx file.
6. Reboot the machine to rescan and to run the enumeration process again.
7. Select the Serial I/O links tab at the bottom of the HM, and create links for the scan window.
8. Select any one of the links in Serial I/O links tab, and right-click and choose scan link option.
9. For better results try Horizontal and Vertical increment by 2 instead the default value.
10. 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.

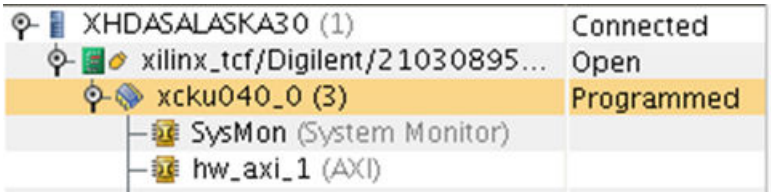
Enable JTAG Debugger

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`.



3. Type `source test_rd.tcl` in the Vivado Tcl Console.
4. 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).

Constraining the Core

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

Required Constraints

The UltraScale Devices Gen3 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 AMD tools. For additional details on the definition and use of an XDC or specific constraints, see *Vivado Design Suite User Guide: Using Constraints* (UG903).

Constraints provided with the integrated block solution have been tested in hardware and provide consistent results. Constraints can be modified, but modifications should only be made with a thorough understanding of the effect of each constraint. Additionally, support is not provided for designs that deviate from the provided constraints.

Device, Package, and Speed Grade Selections

The device selection portion of the XDC informs the implementation tools which part, package, and speed grade to target for the design.

IMPORTANT! *Because UltraScale Devices Gen3 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 4: Designing with the Core, for detailed information about clock requirements.

Clock Management

See Chapter 4: Designing with the Core, for detailed information about clock requirements.

Clock Placement

See Chapter 4: Designing with the Core, for detailed information about clock requirements.

Banking

This section is not applicable for this IP core.

Transceiver Placement

This section is not applicable for this IP core.

I/O Standard and Placement

This section is not applicable for this IP core.

Relocating the Integrated Block Core

By default, the IP core-level constraints lock block RAMs, 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 (that is, the new location is currently being occupied by another module), there are two ways to do this.

- If there is a temporary location available, move the first module out of the way to a new temporary location first. Then, move the second module to the location that was occupied by the first module. 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).

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

PIPE Mode Simulation

The UltraScale Devices Gen3 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.

The following tables 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 57: **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
		common_commands_out[2:1]	pipe_tx_rate_gt
		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 57: **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 58: **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

Notes:

1. This ports functionality has been deprecated and can be left unconnected.

Post-Synthesis/Post-Implementation Netlist Simulation

The UltraScale Devices Gen3 Integrated Block for PCIe core supports post-synthesis/post-implementation netlist functional simulations. However, some configurations do not support this feature in this release. See the following table for the configuration support of netlist functional simulations.

***Note*:** Post-synthesis/implementation netlist timing simulations are not supported for any of the configurations this release.

Table 59: **Configuration Support for Functional Simulation**

Configuration	Verilog	VHDL	External PIPE Interface Mode	Shared Logic in Core	Shared Logic in Example Design
Endpoint	Yes	Yes (Except Tandem mode with **External Startup Primitive** selected)	No	Yes	Yes
Root Port	Not Supported at this time				

Post-Synthesis Netlist Functional Simulation

To run a post-synthesis netlist functional simulation:

1. Generate the core with required configuration
2. Open the example design and run Synthesis
3. After synthesis is completed, in the Flow Navigator, right-click the **Run Simulation** option and select **Run Post-Synthesis Functional Simulation**.

Post-Implementation Netlist Functional Simulation

To run post-implementation netlist functional simulations:

1. Complete the above steps post-synthesis netlist function simulation.
2. Run the implementation for the generated example design.
3. After implementation is completed, in the Flow Navigator, right-click the **Run Simulation** option and select **Run Post-Implementation Functional Simulation**.

Synthesis and Implementation

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

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

Chapter 6

Example Design

This chapter contains information about the example design provided in the AMD Vivado™ Design Suite.

Overview of the Example Design

This section provides an overview of the AMD UltraScale™ Devices Gen3 Integrated Block for PCIe example design.

Integrated Block Endpoint Configuration Overview

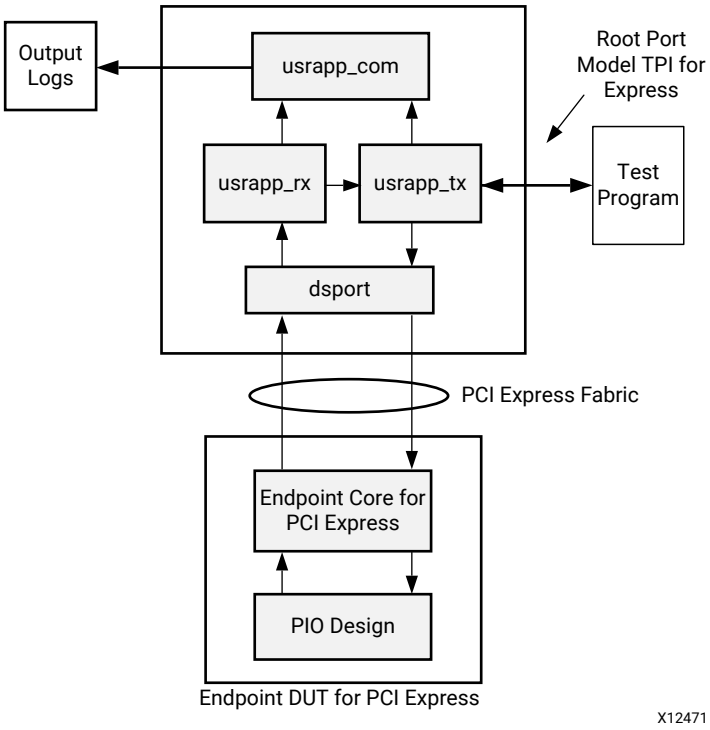
The example simulation design for the Endpoint configuration of the integrated block consists of two discrete parts:

- 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.

Simulation Design Overview

For the simulation design, transactions are sent from the Root Port Model to the core (configured as an Endpoint) and processed by the PIO example design. The following figure illustrates the simulation design provided with the core. For more information about the Root Port Model, see Root Port Model Test Bench for Endpoint.

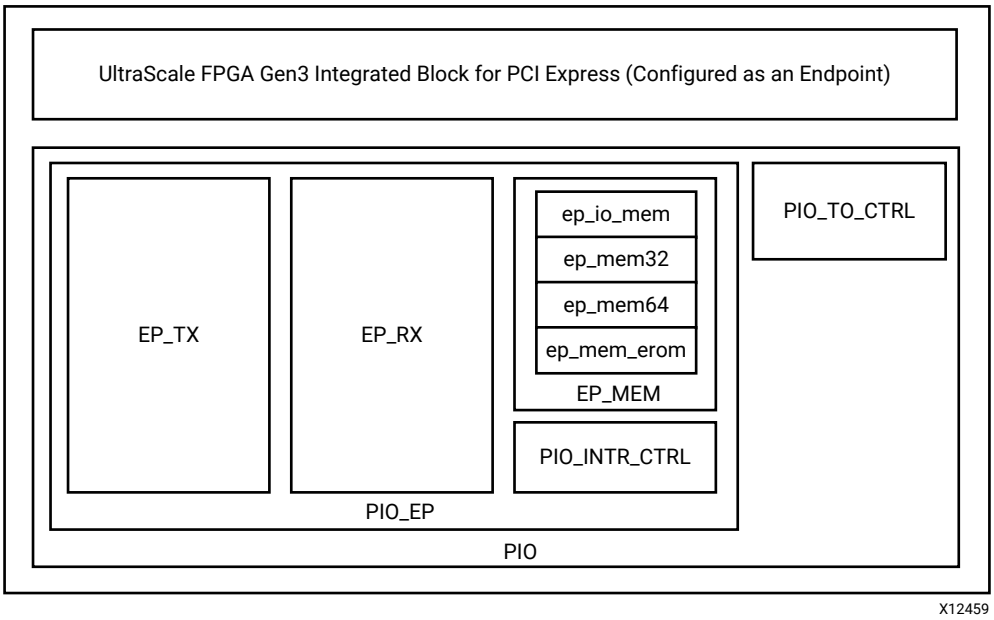
Figure 79: **Simulation Example Design Block Diagram**



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 the following figure. Source code for the example is provided with the core. For more information about the PIO example design, see Programmed Input/Output: Endpoint Example Design.

Figure 80: **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 *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973).

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).

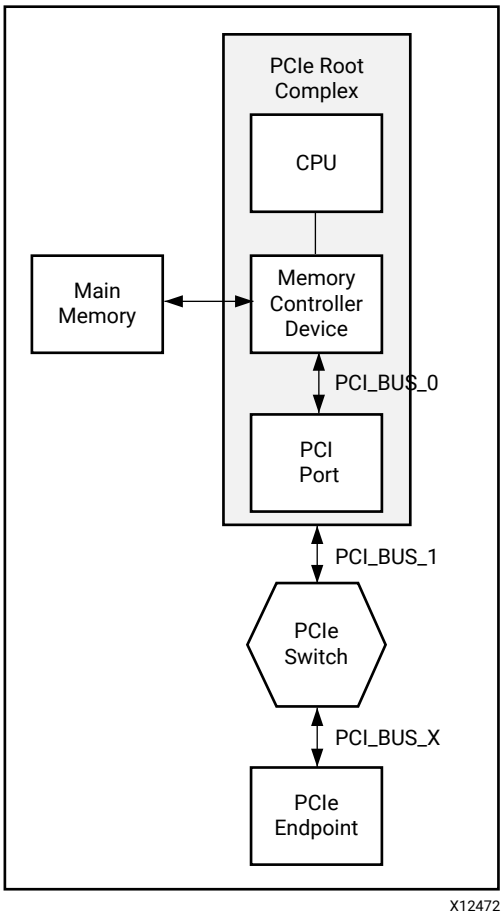
System Overview

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.
- Uses 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.

The following figure illustrates the PCI Express system architecture components, consisting of a Root Complex, a PCI Express switch device, and an Endpoint for PCIe. PIO operations move data *downstream* from the Root Complex (CPU register) to the Endpoint, and/or *upstream* from the Endpoint to the Root Complex (CPU register). In either case, the PCI Express protocol request to move the data is initiated by the host CPU.

Figure 81: **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 the following:

- a Memory Read transaction when the received write address is `11'hEA8` and the write data is `32'hAAAA_BBBB`, and Targeting the BAR0.
- a Legacy Interrupt when the received write address is `11'hEEC` and the write data is `32'hCCCC_DDDD`, and Targeting the BAR0.
- an MSI when the received write address is `11'hEEC` and the write data is `32'hEEEE_FFFF`, and Targeting the BAR0.
- an MSIx when the received write address is `11'hEEC` and the write data is `32'hDEAD_BEEF`, and Targeting the BAR0.

The PIO design processes a Memory or I/O Write TLP with one Dword payload 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 for information about changing the default Vivado Design Suite IP parameters and the effect on the PIO design.

Each of the four 2 KB address spaces represented by the BARs corresponds to one of four 2 KB address regions in the PIO design. Each 2 KB region is implemented using a 2 KB dual-port block RAM. As transactions are received by the core, the core decodes the address and determines which of the four regions is being targeted. The core presents the TLP to the PIO design and asserts the appropriate bits of (BAR ID[2:0]), Completer Request Descriptor[114:112], as defined in the following table.

Table 60: **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

Table 60: **TLP Traffic Types** *(cont'd)*

Block RAM	TLP Transaction Type	Default BAR	BAR ID[2:0]
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

The following table 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 61: **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. The following table identifies the PIO configuration generated based on your selection.

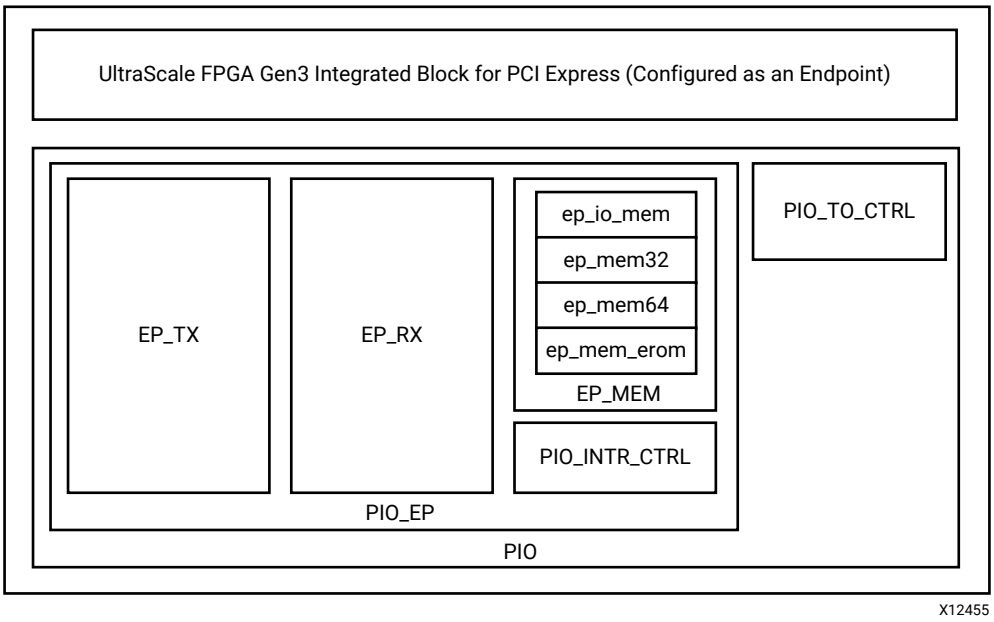
Table 62: **PIO Configuration**

Core	x1	x2	x4	x8
Integrated Block for PCIe	PIO_64	PIO_64, PIO_128	PIO_64, PIO_128, PIO_256	PIO_64, PIO_128[1], PIO_256

Notes:
1. The core does not support 128-bit x8 8.0 Gb/s configuration and 500 MHz user clock frequency.

The following figure shows the various components of the PIO design, which is separated into four main parts: the TX Engine, RX Engine, Memory Access Controller, and Power Management Turn-Off Controller.

Figure 82: **PIO Design Components**

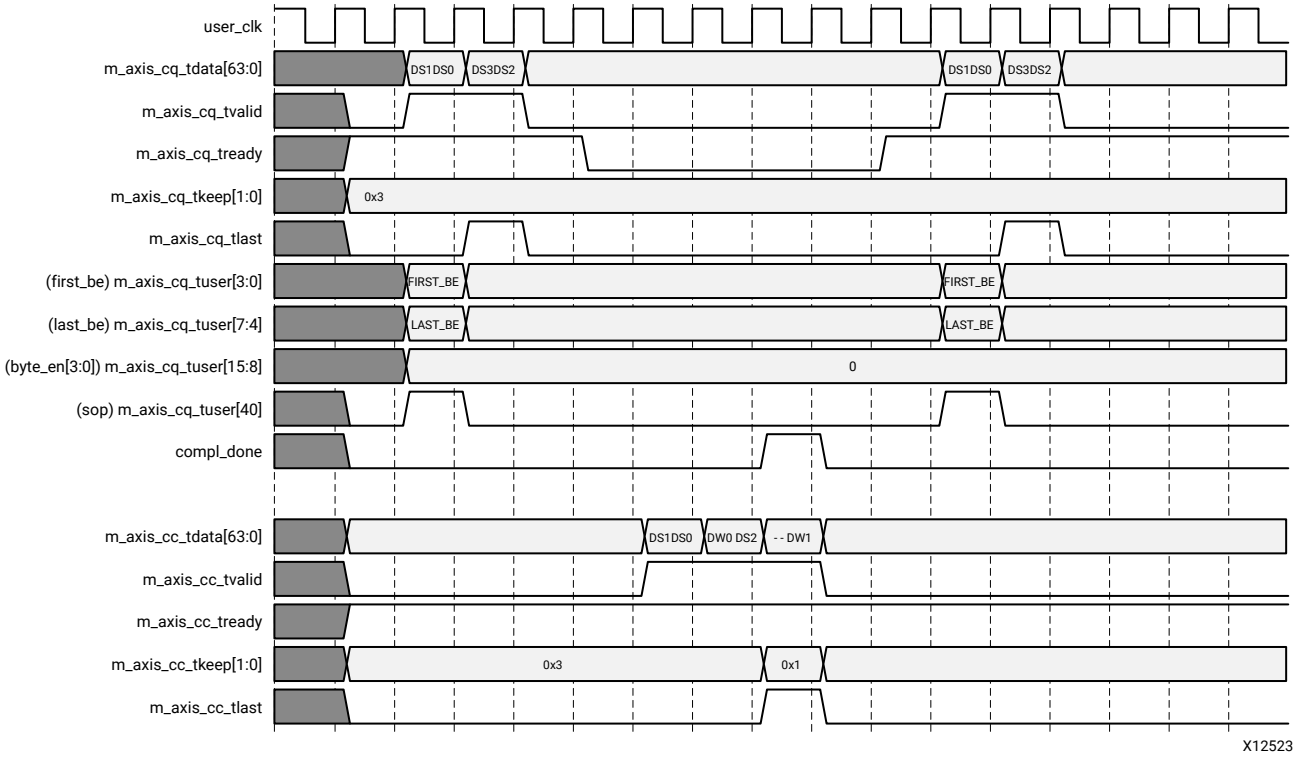


PIO Operation

PIO Write Transaction

The following figure depicts a back-to-back Memory Write to the PIO design. The next Write transaction is accepted only after `wr_busy_o` is deasserted by the memory access unit, indicating that data associated with the first request was successfully written to the memory aperture.

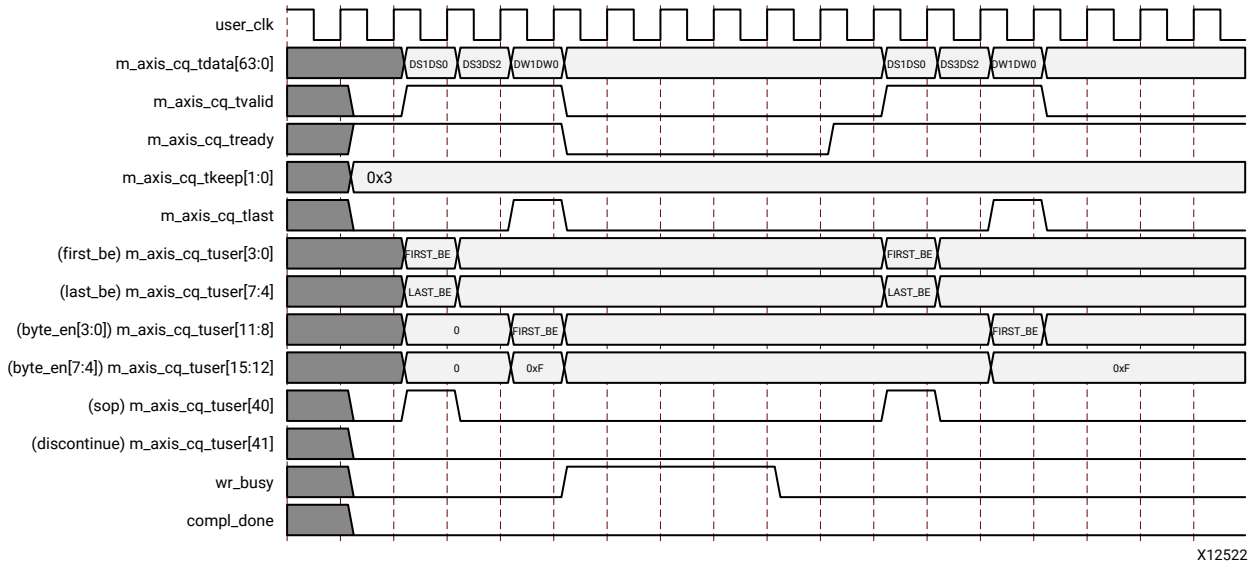
Figure 83: **Back-to-Back Write Transactions**



PIO Read Transaction

The following figure 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 84: **Back-to-Back Read Transactions**



Device Utilization

The following table shows the PIO design FPGA resource utilization.

Table 63: **PIO Design FPGA Resources**

Resources	Utilization
LUTs	300
Flip-Flops	500
Block RAMs	4

Configurator Example Design

The Configurator example design, included with the UltraScale Devices Gen3 Integrated Block for PCIe® in Root Port configuration generated by the Vivado IDE, is a synthesizable, lightweight design that demonstrates the minimum setup required for the integrated block in Root Port configuration to begin application-level transactions with an Endpoint.

System Overview

PCI Express devices require setup after power-on, before devices in the system can begin application specific communication with each other. At least two devices connected through a PCI Express Link must have their Configuration spaces initialized and be enumerated to communicate.

Root Ports facilitate PCI Express enumeration and configuration by sending Configuration Read (CfgRd) and Write (CfgWr) TLPs to the downstream devices such as Endpoints and Switches to set up the configuration spaces of those devices. When this process is complete, higher-level interactions, such as Memory Reads (MemRd TLPs) and Writes (MemWr TLPs), can occur within the PCI Express System.

The Configurator example design described here performs the configuration transactions required to enumerate and configure the Configuration space of a single connected PCI Express Endpoint and allow application-specific interactions to occur.

Configurator Example Design Hardware

The Configurator example design consists of four high-level blocks:

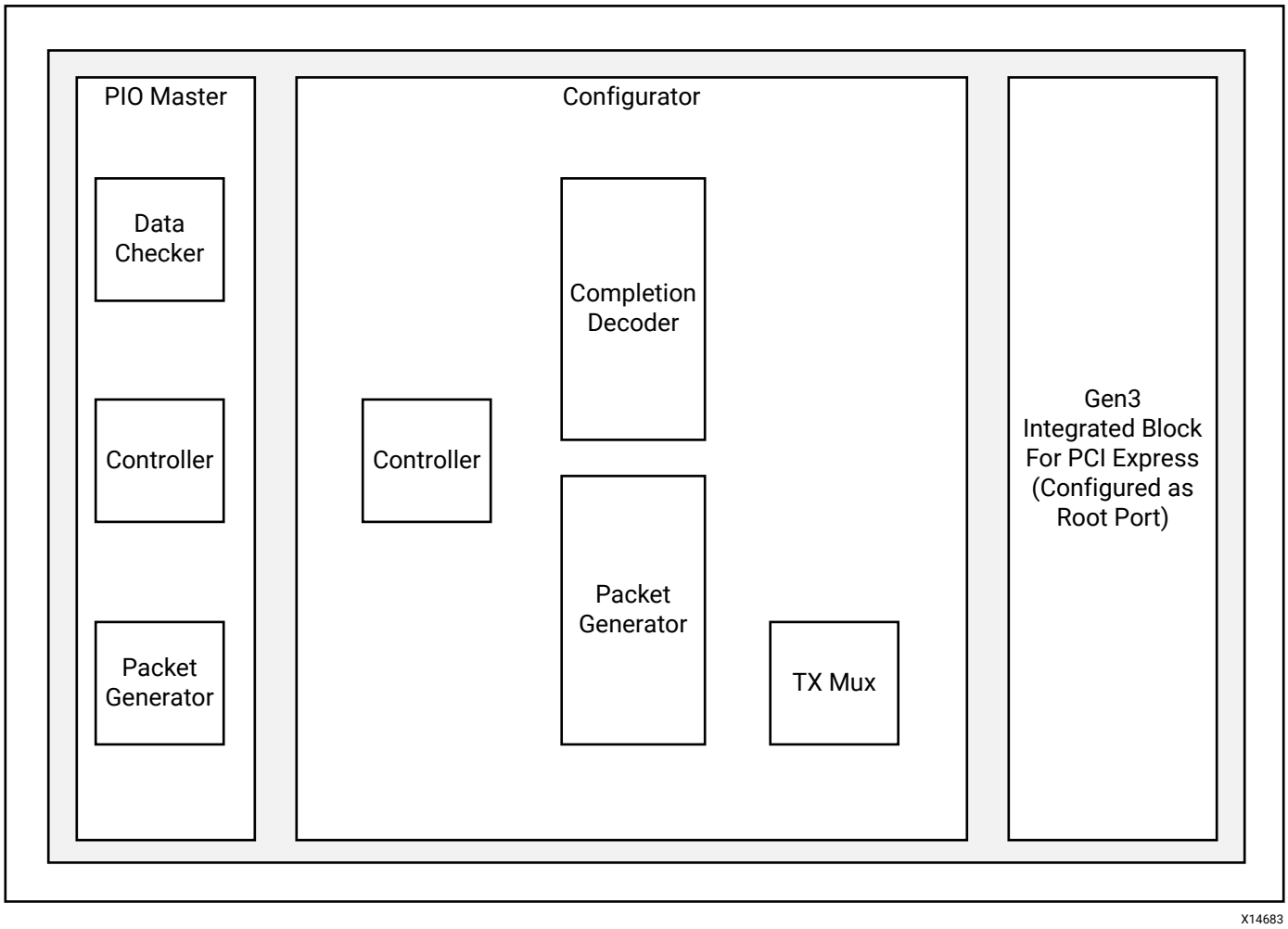
- **Root Port:** The UltraScale Devices Gen3 Integrated Block for PCIe core in Root Port configuration.
- **Configurator Block:** Logical block which interacts with the configuration space of a PCI Express Endpoint device connected to the Root Port.
- **Configurator ROM:** Read-only memory that sources configuration transactions to the Configurator Block.
- **PIO Master:** Logical block which interacts with the user logic connected to the Endpoint by exchanging data packets and checking the validity of the received data. The data packets are limited to a single DWORD and represent the type of traffic that would be generated by a CPU.

***Note*:** The Configurator Block, Configurator ROM, and Root Port are logically grouped in the RTL code within a wrapper file called the Configurator Wrapper.

The Configurator example design, as delivered, is designed to be used with the PIO Slave example included with AMD Endpoint cores and described in Chapter 7: Test Bench. The PIO Master is useful for simple bring-up and debugging, and is an example of how to interact with the Configurator Wrapper. The Configurator example design can be modified to be used with other Endpoints.

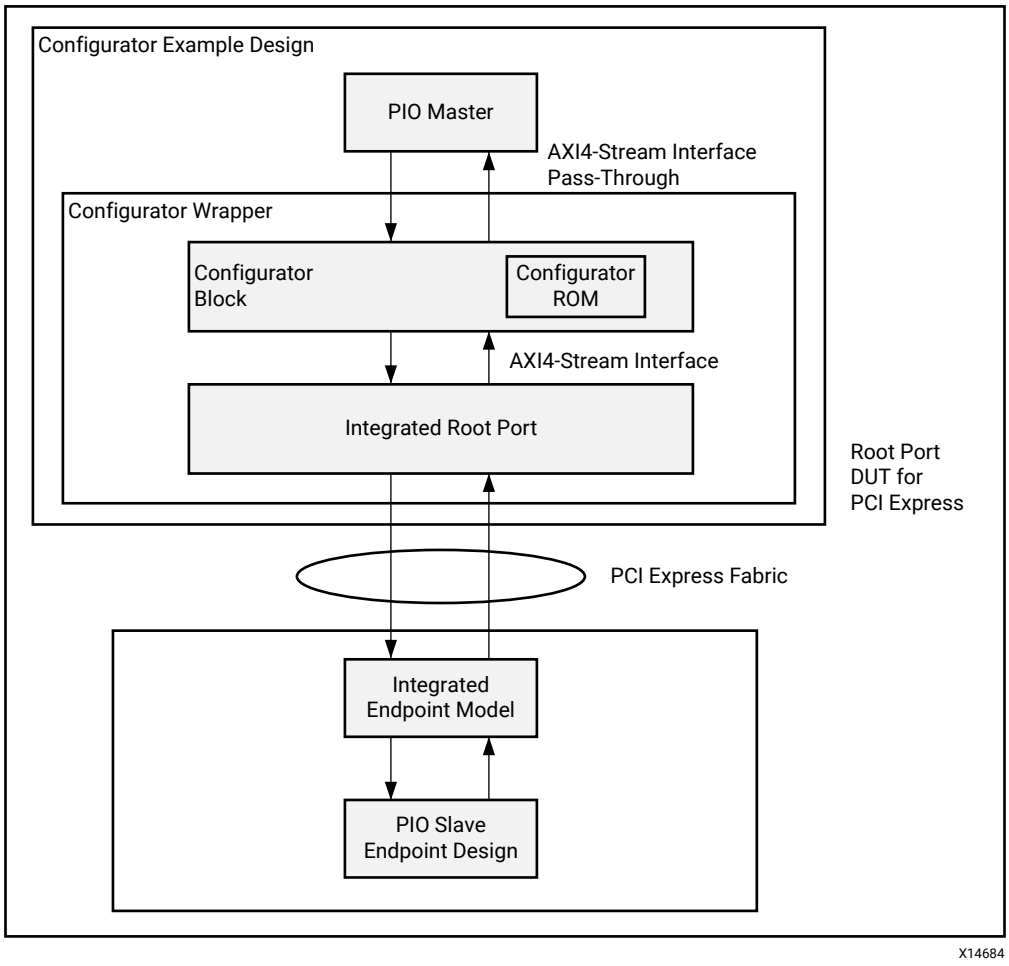
The following figure shows the various components of the Configurator example design.

Figure 85: **Configurator Example Design Components**



The following figure shows how the blocks are connected in an overall system view.

Figure 86: **Configurator Example Design**



Configurator Block

The Configurator Block generates CfgRd and CfgWr TLPs and presents them to the AXI4-Stream interface of the integrated block in Root Port configuration. The TLPs that the Configurator Block generates are determined by the contents of the Configurator ROM.

The generated configuration traffic is predetermined by you to address your particular system requirements. The configuration traffic is encoded in a memory-initialization file (the Configurator ROM) which is synthesized as part of the Configurator. The Configurator Block and the attached Configurator ROM is intended to be usable as a part of a real-world embedded design.

The Configurator Block steps through the Configuration ROM file and sends the TLPs specified therein. Supported TLP types are Message, Message w/Data, Configuration Write (Type 0), and Configuration Read (Type 0). For the Configuration packets, the Configurator Block waits for a Completion to be returned before transmitting the next TLP. If the Completion TLP fields do not match the expected values, PCI Express configuration fails. However, the Data field of Completion TLPs is ignored and not checked.

***Note*:** There is no completion timeout mechanism in the Configurator Block, so if no completion is returned, the Configurator Block waits forever.

The Configurator Block has these parameters, which you can modify:

- **TCQ:** Clock-to-out delay modeled by all registers in design.
- **EXTRA_PIPELINE:** Controls insertion of an extra pipeline stage on the Receive AXI4-Stream interface for timing.
- **ROM_FILE:** File name containing configuration steps to perform.
- **ROM_SIZE:** Number of lines in ROM_FILE containing data (equals number of TLPs to send/2).
- **REQUESTER_ID:** Value for the Requester ID field in outgoing TLPs.

When the Configurator Block design is used, all TLP traffic must pass through the Configurator Block. The user design is responsible for asserting the `start_config` input (for one clock cycle) to initiate the configuration process when `user_lnk_up` has been asserted by the core. Following `start_config`, the Configurator Block performs whatever configuration steps have been specified in the Configuration ROM. During configuration, the Configurator Block controls the core AXI4-Stream interface. Following configuration, all AXI4-Stream traffic is routed to/from the user application, which in the case of this example design is the PIO Master. The end of configuration is signaled by the assertion of `finished_config`. If configuration is unsuccessful for some reason, `failed_config` is also asserted.

If used in a system that supports PCIe v2.2 5.0 Gb/s links, the Configurator Block begins its process by attempting to up-train the link from 2.5 Gb/s to 5.0 Gb/s. This feature is enabled depending on the LINK_CAP_MAX_LINK_SPEED parameter on the Configurator Wrapper.

The Configurator does not support the user throttling received data on the Receive AXI4-Stream interface. Because of this, the Root Port inputs which control throttling are not included on the Configurator Wrapper. These signals are `m_axis_rx_tready` and `rx_np_ok`. This is a limitation of the Configurator example design and not of the core in Root Port configuration. This means that the user design interfacing with the Configurator example design must be able to accept received data at line rate.

Configurator ROM

The Configurator ROM stores the necessary configuration transactions to configure a PCI Express Endpoint. This ROM interfaces with the Configurator Block to send these transactions over the PCI Express link.

The example ROM file included with this design shows the operations needed to configure a UltraScale Devices Gen3 Integrated Block for PCIe and PIO Example Design.

The Configurator ROM can be customized for other Endpoints and PCI Express system topologies. The unique set of configuration transactions required depends on the Endpoint that interacts with the Root Port. This information can be obtained from the documentation provided with the Endpoint.

The ROM file follows the format specified in the Verilog specification (IEEE 1364-2001) section 17.2.8, which describes using the $readmemb function to pre-load data into a RAM or ROM. Verilog-style comments are allowed.

The file is read by the simulator or synthesis tool and each memory value encountered is used as a single location in memory. Digits can be separated by an underscore character (_) for clarity without constituting a new location.

Each configuration transaction specified uses two adjacent memory locations:

- The first location specifies the header fields. Header fields are on even addresses.
- The second location specifies the 32-bit data payload. (For CfgRd TLPs and Messages without data, the data location is unused but still present.) Data payloads are on odd addresses.

For headers, Messages and CfgRd/CfgWr TLPs use different fields. For all TLPs, two bits specify the TLP type. For Messages, Message Routing and Message Code are specified. For CfgRd/CfgWr TLPs, Function Number, Register Number, and first DWORD Byte-Enable are specified. The specific bit layout is shown in the example ROM file.

PIO Master

The PIO Master demonstrates how a user application design might interact with the Configurator Block. It directs the Configurator Block to bring up the link partner at the appropriate time, and then (after successful bring-up) generates and consumes bus traffic. The PIO Master performs writes and reads across the PCI Express Link to the PIO Slave Example Design (from the Endpoint core) to confirm basic operation of the link and the Endpoint.

The PIO Master waits until user_lnk_up is asserted by the Root Port. It then asserts start_config to the Configurator Block. When the Configurator Block asserts finished_config, the PIO Master writes and reads to/from each BAR in the PIO Slave design. If the readback data matches what was written, the PIO Master asserts its `pio_test_finished` output. If there is a data mismatch or the Configurator Block fails to configure the Endpoint, the PIO Master asserts its `pio_test_failed` output. The PIO Master operation can be restarted by asserting its `pio_test_restart` input for one clock cycle.

Configurator File Structure

The following table defines the Configurator example design file structure.

Table 64: **Example Design File Structure**

File	Description
xilinx_pcie_uscale_rp.v	Top-level wrapper file for Configurator example design
cgator_wrapper.v	Wrapper for Configurator and Root Port
cgator.v	Wrapper for Configurator sub-blocks
cgator_cpl_decoder.v	Completion decoder
cgator_pkt_generator.v	Configuration TLP generator
cgator_tx_mux.v	Transmit AXI4-Stream muxing logic
cgator_gen2_enabler.v	5.0 Gb/s directed speed change module
cgator_controller.v	Configurator transmit engine
cgator_cfg_rom.data	Configurator ROM file
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.

Summary

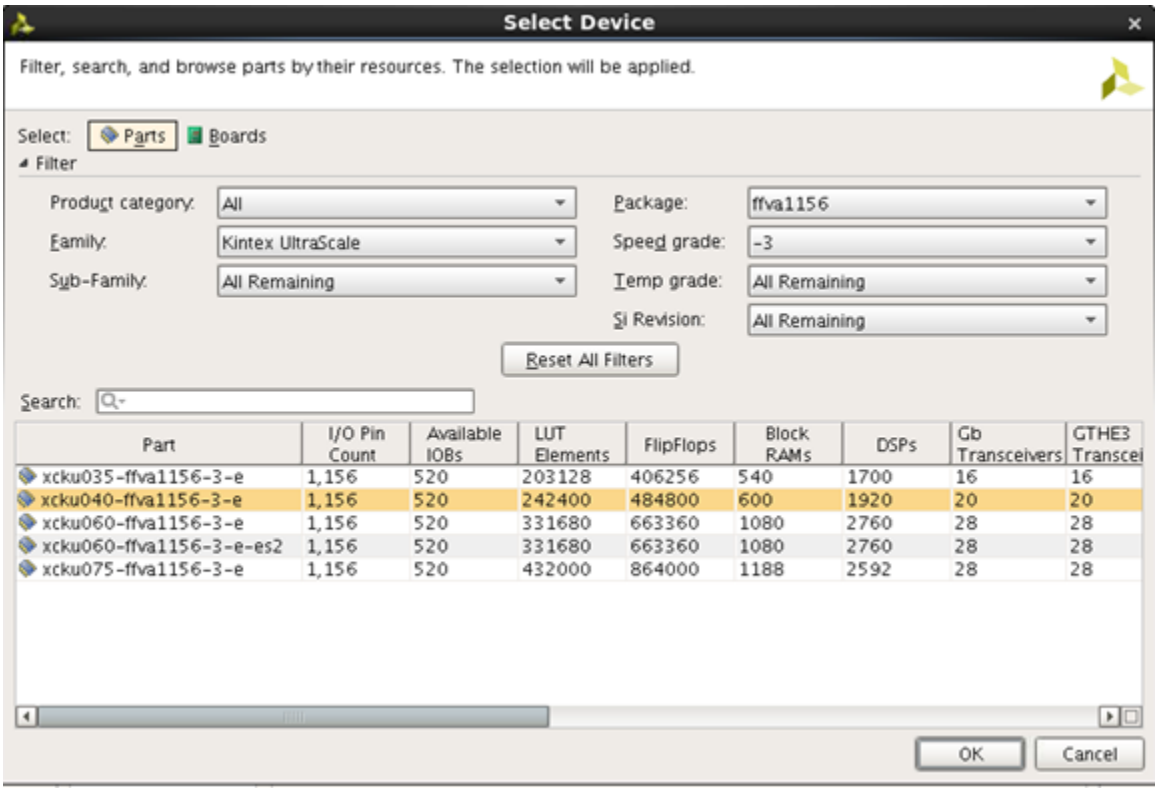
The Configurator example design is a synthesizable design that demonstrates the capabilities of the UltraScale Devices Gen3 Integrated Block for PCIe when configured as a Root Port. The example is provided through the Vivado IDE and uses the Endpoint PIO example as a target for PCI Express enumeration and configuration. The design can be modified to target other Endpoints by changing the contents of a ROM file.

Generating the Core

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

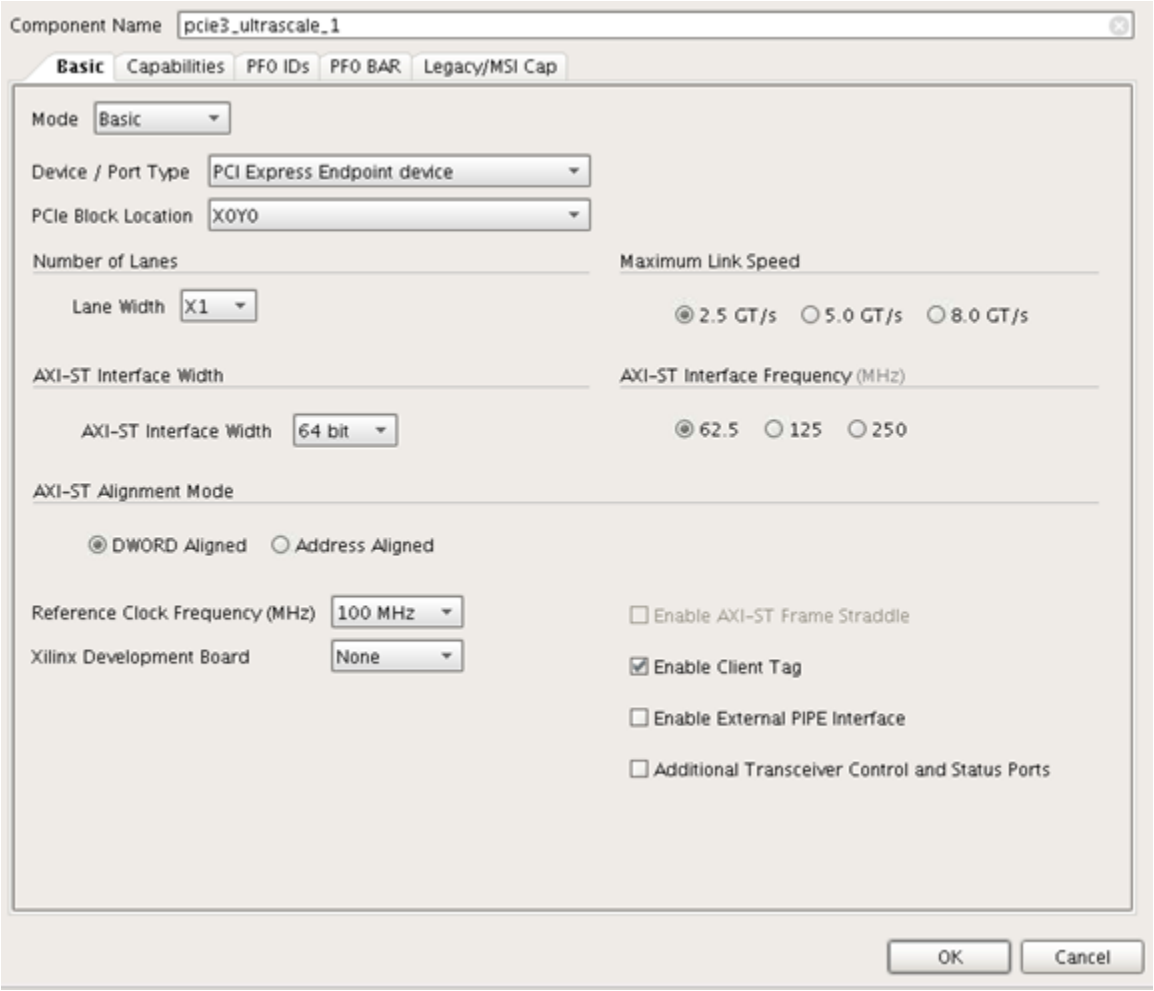
1. Start the Vivado IP catalog.
2. Select **File → Project → New** .
3. Enter a project name and location, then click **Next**. This example uses `project_name.xpr` and `project_dir`.
4. In the New Project wizard pages, do *not* add sources, existing IP, or constraints.
5. From the Part tab in the following figure, select these options:
   - Family: Kintex AMD 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.



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 Gen3 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.



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.

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**.

   ***Note*:** The post-synthesis and post-implementation simulation options are not supported for the PCI Express block.

   After the Run Behavioral Simulation Option is running, you can observe the compilation and elaboration phase through the activity in the Tcl Console, and in the Simulation tab of the Log Window.

3. In Tcl Console, type the `run all` command and press **Enter**. This runs the complete simulation as per the test case provided in example design test bench.

   After the simulation is complete, the result can be viewed in the Tcl Console .

Endpoint Configuration

The simulation environment provided with the UltraScale Devices Gen3 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.

Synthesizing and Implementing the Example Design

To run synthesis and implementation on the example design in the Vivado Design Suite:

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.*

Chapter 7

Test Bench

This chapter contains information about the test bench provided in the AMD 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.

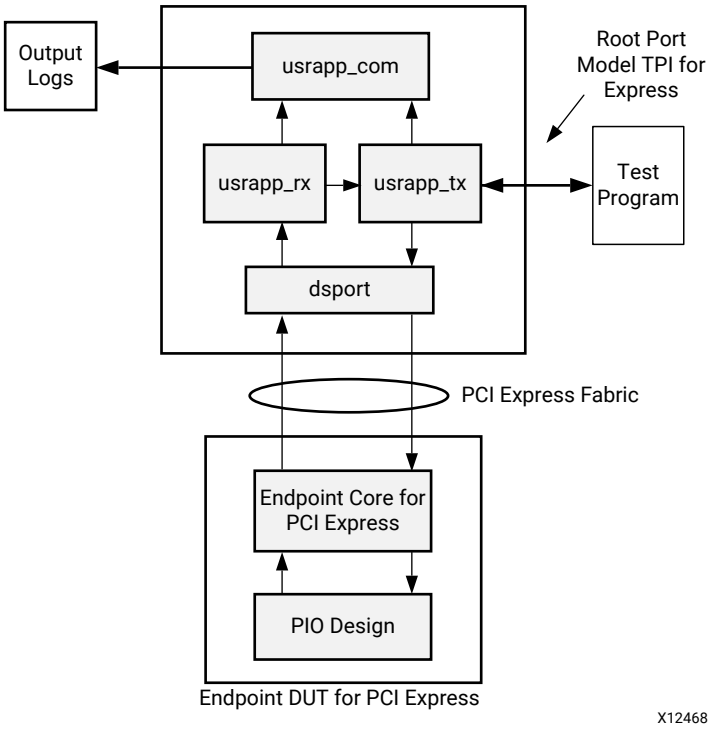
Source code for the Root Port Model is included to provide the model for a starting point for your test bench. All the significant work for initializing the core configuration space, creating TLP transactions, generating TLP logs, and providing an interface for creating and verifying tests are complete, 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

The following figure illustrates the illustrates the Root Port Model coupled with the PIO design.

Figure 87: **Root Port Model and Top-Level Endpoint**



Architecture

The Root Port Model consists of these blocks, illustrated in the preceding figure :

- 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, 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

The following table describes the tests provided with the Root Port Model, followed by specific sections for Verilog test selection.

Table 65: **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).

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.

Note: 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 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 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 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 with the customer design (which can include bus-mastering functionality).

Test Description

The Root Port Model provides a Test Program Interface (TPI). The TPI provides the means to create tests by invoking a series of Verilog tasks. All Root Port Model tests should follow the same six steps:

1. Perform conditional comparison of a unique test name
2. Set up master timeout in case simulation hangs
3. Wait for Reset and link-up
4. Initialize the configuration space of the Endpoint
5. Transmit and receive TLPs between the Root Port Model and the Endpoint DUT
6. Verify that the test succeeded

Test Program: pio_writeReadBack_test0

```
1. else if(testname == "pio_writeReadBack_test1"
2. begin
3. // This test performs a 32 bit write to a 32 bit Memory space and performs a read back
4. TSK_SIMULATION_TIMEOUT(10050);
5. TSK_SYSTEM_INITIALIZATION;
6. TSK_BAR_INIT;
7. for (ii = 0; ii <= 6; ii = ii + 1) begin
8. if (BAR_INIT_P_BAR_ENABLED[ii] > 2'b00) // bar is enabled
9. case(BAR_INIT_P_BAR_ENABLED[ii])
10. 2'b01 : // IO SPACE
11. begin
12. $display("[%t] : NOTHING: to IO 32 Space BAR %x", $realtime, ii);
13. end
14. 2'b10 : // MEM 32 SPACE
15. begin
16. $display("[%t] : Transmitting TLPs to Memory 32 Space BAR %x",
17. $realtime, ii);
18. //------------------------------------------------------------------------
19. // Event : Memory Write 32 bit TLP
20. //------------------------------------------------------------------------
21. DATA_STORE[0] = 8'h04;
22. DATA_STORE[1] = 8'h03;
23. DATA_STORE[2] = 8'h02;
24. DATA_STORE[3] = 8'h01;
25. P_READ_DATA = 32'hffff_ffff; // make sure P_READ_DATA has known initial value
26. TSK_TX_MEMORY_WRITE_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, BAR_INIT_P_BAR[ii][31:0] , 4'hF, 4'hF, 1'b0);
27. TSK_TX_CLK_EAT(10);
28. DEFAULT_TAG = DEFAULT_TAG + 1;
29. //------------------------------------------------------------------------
30. // Event : Memory Read 32 bit TLP
31. //------------------------------------------------------------------------
32. TSK_TX_MEMORY_READ_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, BAR_INIT_P_BAR[ii][31:0], 4'hF, 4'hF);
33. TSK_WAIT_FOR_READ_DATA;
34. if (P_READ_DATA != {DATA_STORE[3], DATA_STORE[2], DATA_STORE[1], DATA_STORE[0] })
35. begin
36. $display("[%t] : Test FAILED --- Data Error Mismatch, Write Data %x != Read Data %x", $realtime,{DATA_STORE[3],
DATA_STORE[2], DATA_STORE[1], DATA_STORE[0]}, P_READ_DATA);
37. end
38. else
39. begin
40. $display("[%t] : Test PASSED --- Write Data: %x successfully received", $realtime, P_READ_DATA);
41. end
```

Expanding the Root Port Model

The Root Port Model was created to work with the PIO design, and for this reason is tailored to make specific checks and warnings based on the limitations of the PIO design. These checks and warnings are enabled by default when the Root Port Model is generated by the Vivado IP catalog. However, these limitations can be disabled so that they do not affect the customer design.

Because the PIO design was created to support at most one I/O BAR, one Mem64 BAR, and two Mem32 BARs (one of which must be the EROM space), the Root Port Model by default makes a check during device configuration that verifies that the core has been configured to meet this requirement. A violation of this check causes a warning message to be displayed as well as for the offending BAR to be gracefully disabled in the test bench. This check can be disabled by setting the `pio_check_design` variable to zero in the `pci_exp_usrapp_tx.v` file.

Root Port Model TPI Task List

The Root Port Model TPI tasks include these tasks, which are further defined in the following tables.

Table 66: **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 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 67: **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.

Table 67: **TLP Tasks** *(cont'd)*

Name	Input(s)		Description
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.
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.

Table 67: **TLP Tasks** *(cont'd)*

Name	Input(s)		Description
TSK_TX_COMPLETION	req_id_ tag_ tc_ len_ comp_status_	15:0 7:0 2:0 10:0 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_ comp_status_ ep_	15:0 7:0 2:0 10:0 11:0 6:0 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.
TSK_TX_MESSAGE	tag_ tc_ len_ data_ message_rtg_ message_code_	7:0 2:0 10:0 63:0 2:0 7:0	Sends a PCI Express Message TLP from Root Port Model to Endpoint DUT. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.
TSK_TX_MESSAGE_DATA	tag_ tc_ len_ data_ message_rtg_ message_code_	7:0 2:0 10:0 63:0 2:0 7:0	Sends a PCI Express Message with Data TLP from Root Port Model to Endpoint DUT. The global DATA_STORE byte array is used to pass message data to task. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.
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.

Table 67: **TLP Tasks** *(cont'd)*

Name	Input(s)		Description
TSK_TX_BAR_WRITE	bar_index byte_offset tag_ tc_ data_	2:0 31:0 7:0 2:0 31:0	Sends a PCI Express one Dword Memory 32, Memory 64, or I/O Write TLP from the Root Port to the target address corresponding to offset byte_offset from BAR bar_index of the Endpoint DUT. This task sends the appropriate Write TLP based on how BAR bar_index has been configured during initialization. This task can only be called after TSK_BAR_INIT has successfully completed.
TSK_WAIT_FOR_READ_DATA	None		Waits for the next completion with data TLP that was sent by the Endpoint DUT. On successful completion, the first Dword of data from the CplD is stored in the global P_READ_DATA. This task should be called immediately following any of the read tasks in the TPI that request Completion with Data TLPs to avoid any race conditions. By default this task locally times out and terminate the simulation after 1000 transaction interface clocks. The global cpld_to_finish can be set to zero so that local timeout returns execution to the calling test and does not result in simulation timeout. For this case test programs should check the global cpld_to, which when set to one indicates that this task has timed out and that the contents of P_READ_DATA are invalid.
TSK_TX_SYNCHRONIZE	first_ active_ last_call_ tready_sw_	- - - -	Waits for assertion of AXI4-Stream Requester Request or Completer Completion Interface Ready signal and synchronizes the output in the log file to each transaction currently active. first_ input indicates start of packet. active_ input indicates a transaction is currently in progress last_call_ input indicates end of packet tready_sw input selects Requester Request or Completer Completion Interface Ready signal
TSK_BUILD_RC_TO_PCIE_PKT	rc_data_QW0 rc_data_QW1 m_axis_rc_tkeep m_axis_rc_tlast	63:0 63:0 KEEP_ WIDTH-1:0 -	Converts AXI4-Stream packet at Requester Completion Interface from a Descriptor packet format to PCIe TLP packet format for logging purposes.
TSK_BUILD_CQ_TO_PCIE_PKT	cq_data cq_be m_axis_cq_tdata	63:0 7:0 63:0	Converts AXI4-Stream packet at Completer Request Interface from a Descriptor packet format to PCIe TLP packet format for logging purposes.
TSK_BUILD_CPLD_PKT	cq_addr cq_be m_axis_cq_tdata	63:0 7:0 63:0	Returns Completion or Completion with Data for Memory Read received from the Endpoint DUT.

Table 68: **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 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 69: **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.

Table 69: **Example PIO Design Tasks** *(cont'd)*

Name	Input(s)		Description
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 70: **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.

Table 70: **Expectation Tasks** *(cont'd)*

Name	Input(s)		Output	Description
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 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 with Bus Master designs.
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 with Bus Master designs.

Table 70: **Expectation Tasks** *(cont'd)*

Name	Input(s)		Output	Description
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 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 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

The figure in Root Port Model Test Bench for Endpoint 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.

***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, 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).

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 is not applicable for the first release of the core.

Upgrading in the Vivado Design Suite

This section provides information about any changes to the user logic or port designations that take place when you upgrade to a more current version of this IP core in the AMD Vivado™ Design Suite.

Parameter Changes

The following table shows the changes to parameters in the current version of the core.

Table 71: **Parameter Changes**

User Parameter Name	Display Name	New/Change/ Removed	Details	Default Value
ext_xvc_vsec_enable	Add the PCIe XVC-VSEC to the Example Design	New	Adds the PCIe XVC-VSEC to the Example Design when selected. PCIe Extended Configuration Space must be enabled to select this option.	False

Port Changes

The ports in the following table appear when Shared Logic option GT Wizard in Core is selected.

Table 72: **GT Wizard in Core Ports**

Name	Direction	Width (depends on link width selected)
rxdfeagchold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfecfokhold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfelfhold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfekhhold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfetap2hold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfetap3hold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH

Table 72: **GT Wizard in Core Ports** *(cont'd)*

Name	Direction	Width (depends on link width selected)
rxdfetap4hold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfetap5hold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfetap6hold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfetap7hold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfetap8hold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfetap9hold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfetap10hold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfetap11hold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfetap12hold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfetap13hold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfetap14hold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfetap15hold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfeuthold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxdfevphold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxoshold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxlpmgchold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxlpmhfhold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxlpmlfhold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH
rxlpmoshold_in	Out	PL_LINK_CAP_MAX_LINK_WIDTH

The ports in the following table appear at the core boundary when the MSI-X is enabled.

Table 73: **MSI-X Ports**

Name	Direction	Width (depends on link width selected)
cfg_interrupt_msi_function_number	In	4 Bits

Migrating From a 7 Series Gen2 Core to an UltraScale Device Gen3 Core

This section provides guidance for migrating from the 7 series Gen2 core to the AMD UltraScale™ devices Gen3 core.

Note: The 7 series Gen3 core interface is the same as that of the UltraScale devices Gen3 core.

In the 7 series Gen2 core, the AXI4-Stream (and TRN) payload byte ordering matches that of the PCIe bus, because the user application is responsible for the formation of the PCIe packets. However, in the UltraScale devices Gen3 v3.1 core, the byte ordering of the payload (after the descriptor) is endian, compliant with the AXI4-Stream protocol.

The first figure in the Chapter 2: Overview chapter shows the AXI4-Stream TX and RX interfaces.

PCIe 3.1 AXI4 ST Enhanced Interface

Completer Request (CQ) Interface

Table 74: **Signal mapping of AXI-4 ST Basic Receive Interface to AXI4-ST Enhanced CQ Interface**

AXI4-Stream (Basic) Receive Interface Name	AXI4-Stream (Enhanced) CQ Interface Name	Differences
m_axis_rx_tlast	m_axis_cq_tlast	None
m_axis_rx_tdata (64/128)	m_axis_cq_tdata (64/128/256)	None
m_axis_rx_tvalid	m_axis_cq_tvalid	None
m_axis_rx_tready	m_axis_cq_tready	None
m_axis_rx_tstrb	m_axis_cq_tkeep and m_axis_cq_tuser	See m_axis_rx_tstrb (64-Bit Interface Only)
m_axis_rx_tuser	m_axis_cq_tuser and m_axis_cq_tdata (Descriptor)	See m_axis_rx_tuser
rx_np_ok	No equivalent signal	N/A
rx_np_req	pcie_cq_np_req	None
No equivalent signal	pcie_cq_np_req_count	N/A

m_axis_rx_tstrb (64-Bit Interface Only)

The following table shows the CR Interface signals used to generate the `m_axis_rx_tstrb` signal bus.

Table 75: **CR Interface Signals for m_axis_rx_tstrb**

AXI4-Stream(Enhanced) CQ Interface Name	Mnemonic
m_axis_cq_tkeep(Data Width/32)	
m_axis_cq_tuser[3:0]	first_be [3:0]
m_axis_cq_tuser[7:4]	last_be [3:0]
m_axis_cq_tuser[39:8]	byte_en [31:0]

m_axis_rx_tuser

The following table shows the CR Interface signals used to generate the `m_axis_rx_tuser` signal bus.

Table 76: **CR Interface Signals for m_axis_rx_tuser**

AXI4-Stream (Basic) Receive Interface Name	Mnemonic	AXI4-Stream (Enhanced) CQ Interface Name	Mnemonic	Notes
m_axis_rx_tuser[0]	rx_ecrc_err	m_axis_cq_tuser[41]	Discontinue	Not exact equivalent
m_axis_rx_tuser[1]	rx_err_fwd	No equivalent signal	N/A	N/A
m_axis_rx_tuser[9:2]	rx_bar_hit[7:0]	m_axis_cq_tdata[114:112] m_axis_cq_tdata[78:75]	Bar ID [2:0] Request Type [3:0]	• Assumed 128/256bit interface • Valid only when Descriptor is present on data bus (sop = 1)
m_axis_rx_tuser[14:10] (128bit Only)	rx_is_sof[4:0]	m_axis_cq_tuser[40]	sop	m_axis_rx_tuser [13:10] can be tied to all 0s.
m_axis_rx_tuser[21:17] (128-bit only)	rx_is_eof [4:0]	m_axis_cq_tlast m_axis_cq_tuser[39:8]	byte_en[31:0]	
No equivalent signal		m_axis_cq_tuser[10:8]	addr_offset[2:0]	
No equivalent signal		m_axis_cq_tuser[12]	tph_present	
No equivalent signal		m_axis_cq_tuser[14:13]	tph_type[1:0]	
No equivalent signal		m_axis_cq_tuser[15]	tph_indirect_tag_en	
No equivalent signal		m_axis_cq_tuser[23:16]	tph_st_tag[7:0]	
No equivalent signal		m_axis_cq_tuser[27:24]	seq_num[3:0]	
No equivalent signal		m_axis_cq_tuser[59:28]	parity	

AXI4-Stream Requester Completion (RC) Interface

Completions for requests generated by user logic are presented on the Request Completion (RC) interface.

Table 77: **AXI4-Stream RC Interface Signal Mapping**

AXI4-Stream (Basic) Receive Interface Name	AXI4-Stream (Enhanced) RC Interface Name	Differences
m_axis_rx_tlast	m_axis_rc_tlast	None
m_axis_rx_tdata (64/128)	m_axis_rc_tdata (64/128/256)	None
m_axis_rx_tvalid	m_axis_rc_tvalid	None
m_axis_rx_tready	m_axis_rc_tready	None

Table 77: **AXI4-Stream RC Interface Signal Mapping** *(cont'd)*

AXI4-Stream (Basic) Receive Interface Name	AXI4-Stream (Enhanced) RC Interface Name	Differences
m_axis_rx_tstrb	m_axis_rc_tkeep and m_axis_rc_tuser	See m_axis_rx_tstrb (64-Bit Interface Only)
m_axis_rx_tuser	m_axis_rc_tuser and m_axis_rc_tdata (Descriptor)	See m_axis_rx_tuser
rx_np_ok	No equivalent signal	N/A
rx_np_req	No equivalent signal	cq_np_req and cq_np_req_count are used for NP FC.

m_axis_rx_tstrb (64-Bit Interface Only)

The following table shows the Requester Completion interface signals used to generate the `m_axis_rx_tstrb` signal bus.

Table 78: **RC Interface Signals for m_axis_rx_tstrb**

AXI4-Stream (Enhanced)RC Interface Name	Mnemonic
m_axis_rc_tkeep (Data Width/32)	
m_axis_rc_tuser[31:0]	byte_en [31:0]

m_axis_rx_tuser

The following table shows the Requester Completion interface signals used to generate the `m_axis_rx_tuser` signal bus.

Table 79: **RC Interface Signals for m_axis_rx_tuser**

AXI4-Stream Receive Interface Name	Mnemonic	AXI4-Stream Completer Request Interface Name	Mnemonic	Notes
m_axis_rx_tuser[0]	rx_ecrc_err	m_axis_rc_tuser[41]	Discontinue	Not exact equivalent
m_axis_rx_tuser[1]	rx_err_fwd	m_axis_rx_tdata[46]	Poisoned completion	Valid only when Descriptor is present on the data bus (is_sof0/is_sof1=1).
m_axis_rx_tuser[9:2]	rx_bar_hit[7:0]	N/A (Refer to CQ interface)	N/A	N/A

Table 79: **RC Interface Signals for m_axis_rx_tuser** *(cont'd)*

AXI4-Stream Receive Interface Name	Mnemonic	AXI4-Stream Completer Request Interface Name	Mnemonic	Notes
m_axis_rx_tuser[14:10] (128-bit only)	rx_is_sof[4:0]	m_axis_rc_tuser[32] m_axis_rc_tuser[33] (only for 256-bit straddle)	is_sof_0 is_sof_1	256-bit RC interface provides straddling option. If enabled, the core can straddle two completion TLPs in the same beat. is_sof_1 is used only when straddling is enabled for 256-bit interface.
m_axis_rx_tuser[21:17] (128-bit only)	rx_is_eof [4:0]	m_axis_rc_tuser[37:34] m_axis_rx_tuser[41:38] (only for 256-bit straddle)	Is_eof_0[3:0] Is_eof_1[3:0]	is_eof_1 is used only when straddling is enabled for 256-bit interface.
No equivalent signal		m_axis_rc_tuser[74:43]	Parity	

AXI4-Stream (Enhanced) Completer Completion Interface

Table 80: **Signal Mapping of AXI4-Stream (Basic) Transmit Interface to AXI4-Stream (Enhanced) Completer Completion Interface**

AXI4-Stream (Basic) Transmit Interface Name	AXI4-Stream (Enhanced) Completer Completion Interface Name	Differences
s_axis_tx_tlast	s_axis_cc_tlast	None
s_axis_tx_tdata (64/128)	s_axis_cc_tdata (64/128/256)	None
s_axis_tx_tvalid	s_axis_cc_tvalid	None
s_axis_tx_tready	s_axis_cc_tready	None
s_axis_tx_tstrb	s_axis_cc_tkeep s_axis_cc_tdata[28:16]	See s_axis_tx_tstrb
s_axis_tx_tuser	s_axis_cc_tuser	See s_axis_tx_tuser
tx_buf_av[5:0]		
tx_terr_drop		
tx_cfg_req	NA	None
tx_cfg_gnt	NA	None

s_axis_tx_tstrb

Use s_axis_cc_tkeep with Byte Count Descriptor (s_axis_cc_tdata[28:16]) to indicate the byte enables for the last Dword of the payload.

The following table shows the mapping between `s_axis_cc_tkeep` from the Completer Completion interface and the `s_axis_tx_tstrb` signal bus from the AXI4-Stream (Basic) Transmit interface when `tlast` is not asserted.

Table 81: **Mapping Between s_axis_cc_tkeep and s_axis_tx_tstrb**

Interface Width	s_axis_tx_tstrb	s_axis_rq_tkeep
64	0x0F	0x1
	0xFF	0x3
128	0x0F	0x1
	0xFF	0x3
	0xFFF	0x7
	0xFFFF	0xF

s_axis_tx_tuser

The following table shows the mapping between `s_axis_cc_tuser` from the Completer Completion interface and the `s_axis_tx_tuse` r signal bus from the AXI4-Stream (Basic) Transmit interface.

Table 82: **Mapping Between s_axis_cc_tkeep and s_axis_tx_tstrb**

AXI4-Stream (Basic) Receive Interface Name	Mnemonic	AXI4-Stream (Enhanced) Completer Request Interface Name	Mnemonic	Notes
s_axis_tx_tuser[0]	tx_ecrc_gen	s_axis_cc_tdata[95]	Force ECRC	Same functionality
s_axis_tx_tuser[1]	tx_err_fwd	s_axis_cc_tdata[46]	Poisoned completion	Same functionality
s_axis_tx_tuser[2]	tx_str	NA	NA	No equivalent signal
s_axis_tx_tuser[2]	t_src_dsc	s_axis_cc_tuser[0]	Discontinue	Same functionality

AXI4-Stream Requester Request Interface

Table 83: **AXI4-Stream Requester Request Interface Signal Mapping**

AXI4-Stream (Basic) Transmit Interface Name	AXI4-Stream (Enhanced) Requester Request Interface Name	Differences
s_axis_tx_tlast	s_axis_rq_tlast	None
s_axis_tx_tdata (64/128)	s_axis_rq_tdata (64/128/256)	None
s_axis_tx_tvalid	s_axis_rq_tvalid	None
s_axis_tx_tready	s_axis_rq_tready	None
s_axis_tx_tstrb	s_axis_rq_tkeep	See s_axis_tx_tuser

Table 83: **AXI4-Stream Requester Request Interface Signal Mapping** *(cont'd)*

AXI4-Stream (Basic) Transmit Interface Name	AXI4-Stream (Enhanced) Requester Request Interface Name	Differences
s_axis_tx_tuser	s_axis_rq_tuser	See s_axis_tx_tuser
tx_buf_av[5:0]	pcie_tfc_nph_av / pcie_tfc_npd_av/ pcie_rq_tag_av	See tx_buf_av
tx_terr_drop		
tx_cfg_req	NA	The feature is not available.
tx_cfg_gnt	NA	The feature is not available.

s_axis_tx_tstrb

The following table shows the Requester Request interface signals used to generate the `s_axis_tx_tstrb` signal bus.

Table 84: **Requester Request Interface Signals for m_axis_rx_tuser**

AXI4-Stream Requester (Enhanced) Request Interface Name	Mnemonic
s_axis_rq_tkeep	
s_axis_rq_tuser[3:0]	first_be [3:0]
s_axis_rq_tuser[7:4]	last_be [3:0]

The following table shows the mapping between `s_axis_cc_tkeep` from the Completer Completion interface and the `s_axis_tx_tstrb` signal bus from the AXI4-Stream (Basic) Transmit interface when `tlast` is not asserted.

Table 85: **Mapping Between s_axis_cc_tkeep and s_axis_tx_tstrb**

Interface Width	s_axis_tx_tstrb	s_axis_rq_tkeep
64	0x0F	0x1
	0xFF	0x3
128	0x0F	0x1
	0xFF	0x3
	0xFFF	0x7
	0xFFFF	0xF

s_axis_tx_tuser

The following table shows the mapping between `s_axis_rq_tuser` from Requester Request interface and `s_axis_tx_tuser` signal bus from the AXI4-Stream (Basic) Transmit interface.

Table 86: **Mapping between s_axis_rq_tuser and s_axis_tx_tuser**

AXI4-Stream (Basic) Receive Interface Name	Mnemonic	AXI4-Stream (Enhanced) Requester Request Interface Name	Mnemonic	Comments
s_axis_tx_tuser[0]	tx_ecrc_gen	s_axis_rq_tdata[127]	Force ECRC	Same Functionality
s_axis_tx_tuser[1]	tx_err_fwd	s_axis_rq_tdata[79]	Poisoned request	Same Functionality
s_axis_tx_tuser[2]	tx_str	NA	NA	No Equivalent Signal
s_axis_tx_tuser[2]	t_src_dsc	s_axis_rq_tuser[11]	Discontinue	Same Functionality

tx_buf_av

The buffer availability has been split into three individual signals for the AXI4-Stream (Enhanced) Requester Request interface.

- `pcie_tfc_nph_av` indicates the currently available header credit for non-posted TLPs on the transmit side of the core.
- `pcie_tfc_npd_av` indicates the currently available payload credit for non-posted TLPs on the transmit side of the core.
- `pcie_rq_tag_av` indicates the currently available header credit for non-posted TLPs on the transmit side of the core.

Other Interfaces

The following table describes additional interfaces provided by the core.

Table 87: **Additional Interfaces Provided by the Core**

Interfaces	Description	Notes
Transmit Flow Control	Used by the user application to request which flow control information the core provides. Based on the setting flow control input to the core, this interface provides the following to the user application: • Posted/Non-Posted Header Flow Control Credits • Posted/Non-Posted Data Flow Control Credits • Completion Header Flow Control Credits • Completion Data Flow Control Credits	Similar functionality
Configuration Management	Used to read and write to the Configuration Space registers.	Similar functionality

Table 87: **Additional Interfaces Provided by the Core** *(cont'd)*

Interfaces	Description	Notes
Configuration Status	Provides information about how the core is configured, such as the negotiated link width and speed, the power state of the core, and configuration errors.	Similar functionality as Configuration Specific register ports
Configuration Received Message	Indicates the logic of a decodable message from the link, the parameters associated with the data, and the type of message received	Similar functionality as Received Message TLP status ports
Configuration Transmit Message	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.	
Per Function Status	Provides status data requested by the user application through the selected function.	Similar functionality as Error Reporting Ports
Configuration Control	Allows information exchange between the user application and the core. The user application uses this interface to: • 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.	Similar functionality as the Power Management Port
Configuration Interrupt Controller	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.	Similar functionality as Interrupt Generation and Status Ports
Configuration Extended	Allows the core to transfer configuration information with the user application when externally implemented configuration registers are implemented.	Similar functionality as Received Configuration TLP Status Ports

Package Migration of UltraScale Devices PCI Express Designs

The UltraScale portfolio offers many devices that share the same package footprint. This migration path allows you to switch between devices to accommodate changes in design size or the need for specific additional functionality, such as moving from Virtex to Kintex devices when additional DSP blocks are needed.

While most PCI Express configurations can easily migrate in a package, there are some designs and settings where migration of a specific PCI Express implementation might not be possible. This section walks you through how to create PCI Express designs that can be migrated across the desired parts, and identifies migration pin outs that cannot be migrated.

For details on pin migration as a whole for the UltraScale family, see *UltraScale Architecture PCB Design User Guide* (UG583).

Placement Rules

The UltraScale Devices Gen3 Integrated Block for PCIe solution delivered from the Vivado IP catalog has certain placement restrictions to ensure that your design closes timing. Following are two of the rules that impact the ability of the PCI Express solution to migrate across packages.

- **Rule #1:** Lane 0 of the PCIe Interface is limited to the GTH quad one clock region above, in the same clock region or one clock region below the PCI Express hard block. When eight PCI Express lanes are used, the GTH quads must be in adjacent quads.
- **Rule #2:** The integrated block for PCIe and the GTH transceivers that are connected together must reside on the same Super Logic Region (SLR).

These two rules are explained in further detail in the following sections.

GTH Location

With each PCI Express core generated from the Vivado IDE, AMD provides recommended locations of the GTH for the specified PCI Express block. While you can change the GTH locations, AMD cannot guarantee that timing closure will be possible with these alternative locations.

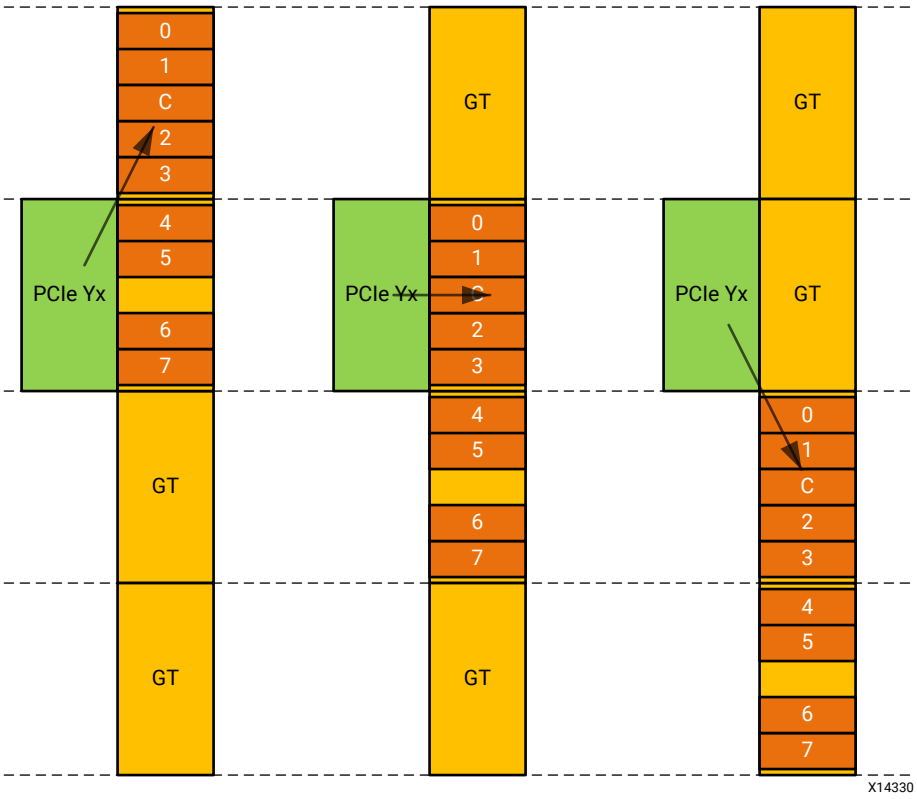
Starting with the Vivado 2014.3 software release, you have more flexibility to choose the GT locations used in a design. You can choose:

- The PCIe block and GTH quad location in which lane 0 is placed.
- The location of the GT quad, which can be one clock region above the PCIe block, in the same region as the PCIe block, or one clock region below the PCIe block.

After the quad location is chosen, the remaining GTH locations are constrained based on the link width selected. SLR boundaries and non-bonded out GTs affect which GTHs are available. When a x8 link width is selected, both GTH quads used must be adjacent to each other.

The following figure shows how lanes are distributed for each initial GTH quad locations for a x8 PCIe link width.

Figure 88: **GTH Quad Locations**

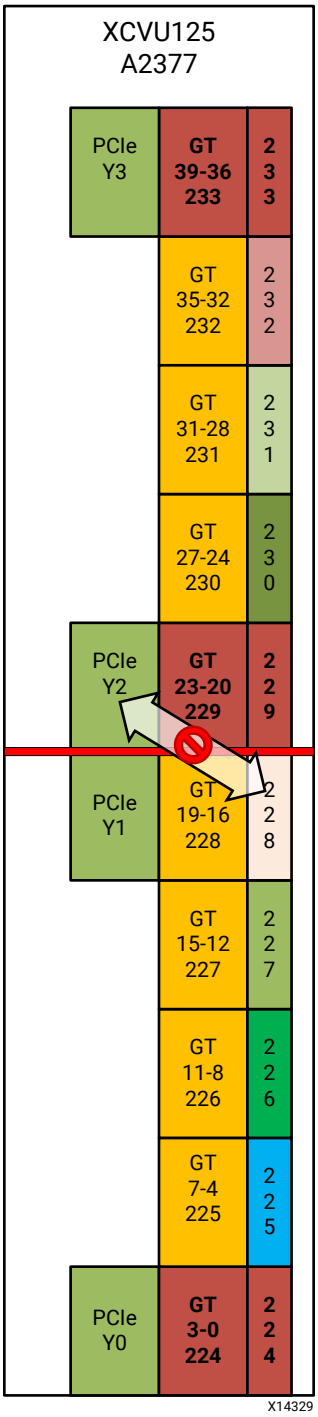


Stacked Silicon Interconnect (SSI) Devices

When SSI devices are used, the PCI Express hard block and the GTH quads connected to the PCIe hard block must be on the same Super Logic Region (SLR).

The following figure shows the XCVU125 device in an A2377 package. The integrated block for PCIe located at location Y2 cannot select the GTH at bank 228, because an SLR boundary would be crossed.

Figure 89: **XCVU125 Device in an A2377 Package**



Creating a Migration Plan for PCI Express

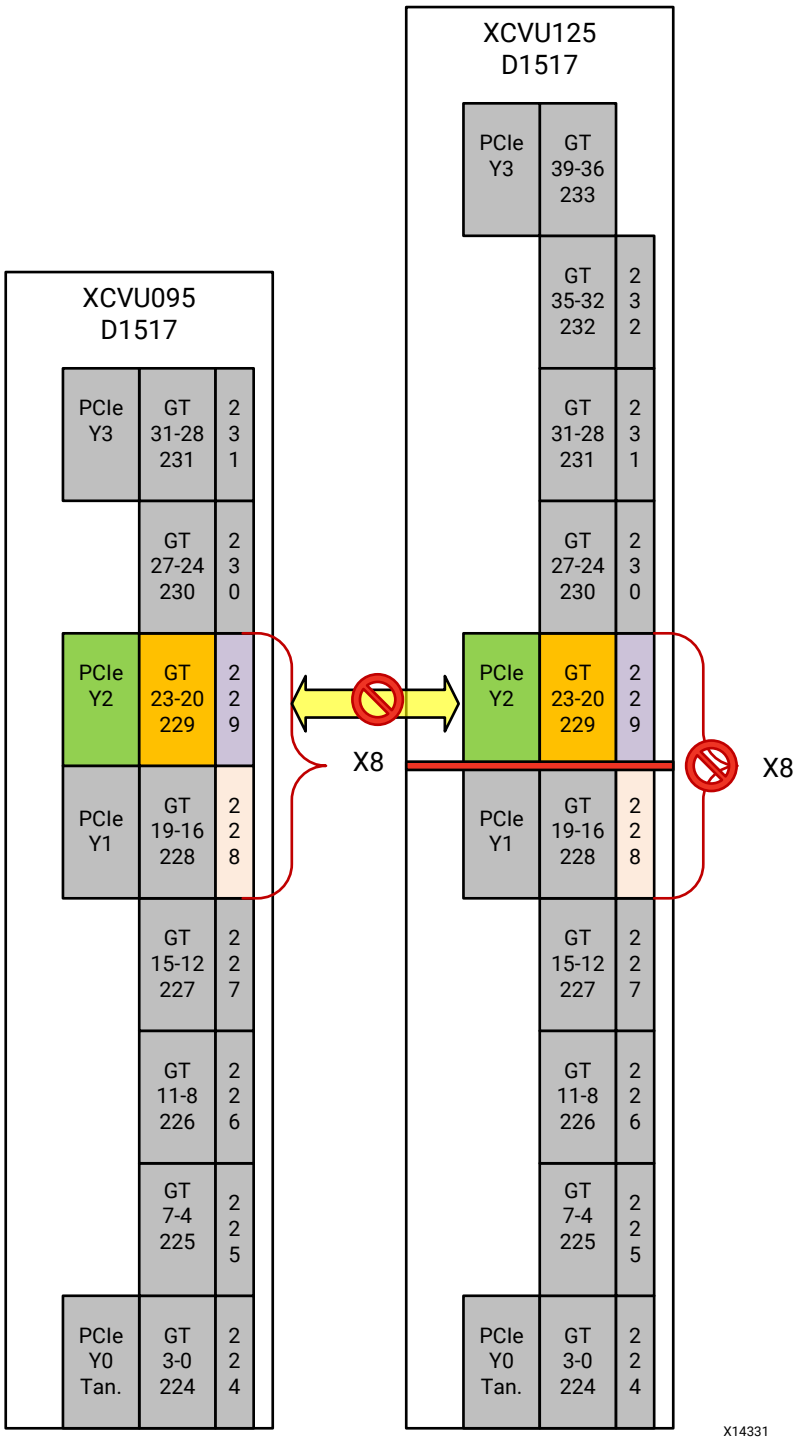
Keeping placement rules in mind, you can make a PCI Express package migration plan as follows:

1. Choose a part and package to which you want to migrate, such as parts for the D1517 package, as in the example in the followi ng figure.
2. Choose the PCI Express link width that is desired.
3. Evaluate how the GTH location for that PCI Express location migrates.
4. Look for possible issues crossing SLR boundaries or for transceivers that are not bonded out.

   For instance, choosing PCIe location Y2 in the XCVU095 with the GTH location of 229 that is eight lanes wide cannot migrate to the XCVU125. This is because the SLR boundary crossing rule would be violated. See the following figure.

   For this example, the GT quad one clock region above the integrated block for PCIe at Y2 can be selected for an X8 design and can migrate across these two parts.
5. It might be necessary to recompile the IP to generate new location constraints in some cases. To accomplish this in the Vivado IDE, update the PCI Express location in the IP settings, and then generate the core.

Figure 90: **Migration Example**



Migrating Tandem Configuration

Migrating Tandem Configuration designs across different parts in the same package is straight forward. Be sure that the PCI Express block that supports Tandem PCIe is selected.

IMPORTANT! *Tandem PCIe is only supported for one PCIe hard block per device.*

For a list of PCI Express block locations and dedicated reset pin locations that support Tandem, see Supported Devices.

Appendix B

GT Locations

This appendix provides a list of GTs locations available for this IP core and lists some key recommendations that should be considered when selecting the GT location. The following sections include tables that identify which GT Banks are available for selection based on the PCIe block location as selected during IP customization.

- Virtex UltraScale Devices Available GT Quads
- Kintex UltraScale Devices Available GT Quads

A GT Quad is comprised of four GT lanes. When selecting GT Quads for the PCIe IP, AMD recommends that you use the GT Quad most adjacent to the PCIe hard block. While this is not required, it will improve place, route, and timing for the design.

- Link widths of x1, x2, and x4 require one bonded GT Quad and should not split lanes between two GT Quads.
- A link width of x8 requires two adjacent GT Quads that are bonded and are in the same SLR.

PCIe lane 0 is placed in the top-most GT of the top-most GT Quad by default (as shown in Vivado Integrated Design Environment (IDE) Device view). Subsequent lanes use the next available GTs moving vertically down the device as the lane number increments. This means that by default the highest PCIe lane number uses the bottom-most GT in the bottom-most GT Quad that is used for PCIe. During IP customization, you can select the desired GT Quad for PCIe lane 0 from the drop-down selections.

The PCIe reference clock (`sys_clk_p` / `sys_clk_n`) uses `GTREFCLK0` in the PCIe lane 0

GT Quad for x1, x2, x4, and x8 configurations by default. You can modify the reference clock default location by adding pin location constraints to the design.

The following diagrams show the ideal GT Quad and reference clock selections for various PCIe link configurations relative to the PCIe block location for a representative device.

Figure 91: **Most Adjacent GT Quad Location For x1, x2, x4 PCIe Link Width**

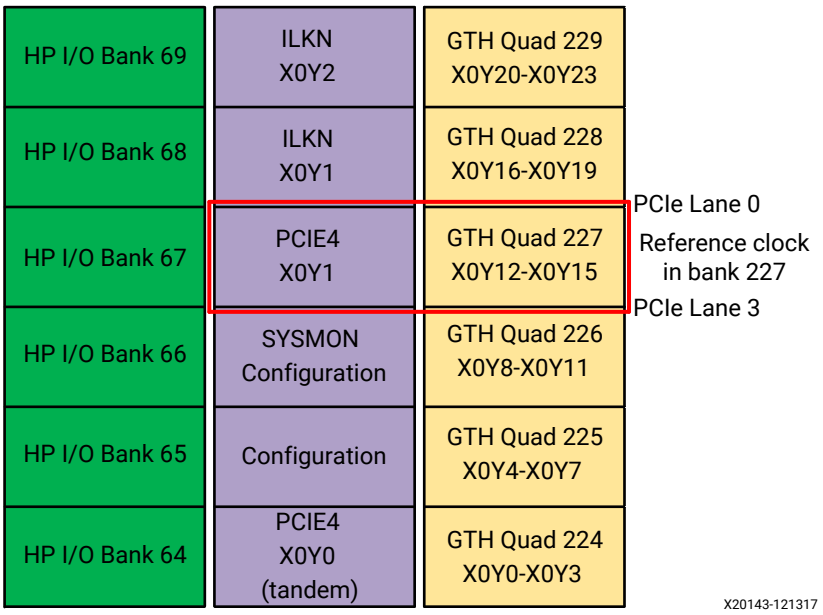
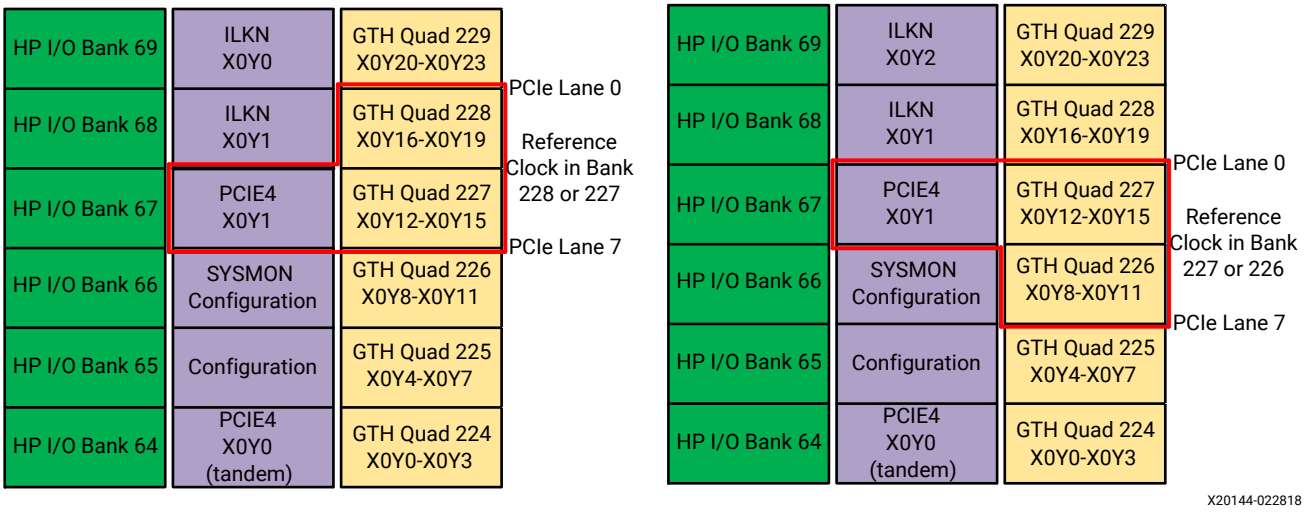
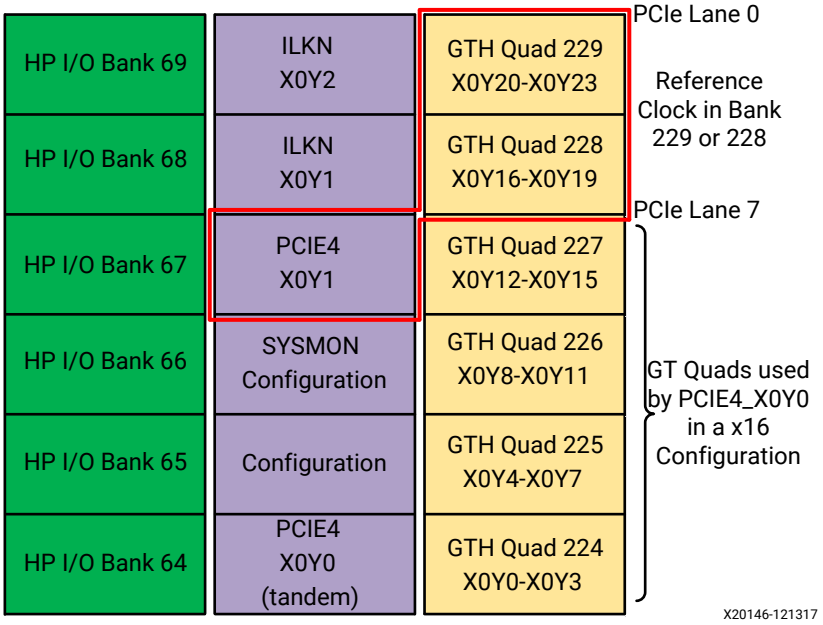


Figure 92: **Most Adjacent GT Quads For x8 PCIe Link Width**



Some PCIe locations have non-ideal GT Quad selections as result of their proximity to the edge of the device, SLR boundary, or other PCIe blocks. In these scenarios, the most adjacent GTs might not be optimal for place and route, but will work as desired. The following figure shows one common example.

Figure 93: **Alternative PCIe GT Location Selection**



Virtex UltraScale Devices Available GT Quads

The following table hows the PCIe lane0 GT Quad options available for the different AMD Virtex™ UltraScale™ devices. The GT Quad location is shown using the GT Quad bank number rather than GT XY coordinates. The *UltraScale and UltraScale+ FPGAs Packaging and Pinouts Product Specification* (UG575) provides a diagram describing the PCIe block locations relative to enabled GT Quads and includes both GT bank numbering and XY coordinates if needed.

Note: The selections in bold are the GT Quad defaults for a given device, package, and PCIe block.

Virtex Ultrascale Devices Available GT Quads

Device	Package	PCIe Blocks	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU065	FFVC1517	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228	GTH_Quad_227
XCVU080	FFVC1517	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228	GTH_Quad_227
		X0Y2	X8 Not Supported	GTH_Quad_228
		X0Y3	X8 Not Supported	GTH_Quad_228

Device	Package	PCIe Blocks	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU080 cont.	FFVA2104	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227
		X0Y2	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228
		X0Y3	GTH_Quad_230	GTH_Quad_229
	FFVB2104	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227
		X0Y2	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228
		X0Y3	GTH_Quad_231	GTH_Quad_230
	FFVB1760	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227
		X0Y2	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228
		X0Y3	GTH_Quad_231	GTH_Quad_230
	FFVD1517	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227
		X0Y2	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228
		X0Y3	GTH_Quad_231	GTH_Quad_230
XCVU095	FFVC1517	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228	GTH_Quad_227
		X0Y2	X8 Not Supported	GTH_Quad_228
		X0Y3	X8 Not Supported	GTH_Quad_228
	FFVA2104	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227
		X0Y2	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228
		X0Y3	GTH_Quad_230	GTH_Quad_229

Device	Package	PCIe Blocks	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU095 cont.	FFVB2104	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227
		X0Y2	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228
		X0Y3	GTH_Quad_231	GTH_Quad_230
	FFVB1760	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227
		X0Y2	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228
		X0Y3	GTH_Quad_231	GTH_Quad_230
	FFVC2104	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227
		X0Y2	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228
		X0Y3	GTH_Quad_231	GTH_Quad_230
	FFVD1517	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227
		X0Y2	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228
		X0Y3	GTH_Quad_231	GTH_Quad_230
XCVU125	FLVB1760	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228	GTH_Quad_227
		X0Y2	X8 Not Supported	GTH_Quad_230
		X0Y3	GTH_Quad_233	GTH_Quad_232
	FLVB2104	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228	GTH_Quad_227
		X0Y2	GTH_Quad_230	GTH_Quad_229
		X0Y3	GTH_Quad_233	GTH_Quad_232
	FLVC2104	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228	GTH_Quad_227
		X0Y2	GTH_Quad_230	GTH_Quad_229
		X0Y3	GTH_Quad_233	GTH_Quad_232

Device	Package	PCIe Blocks	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU125 cont.	FLVA2104	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	X8 Not Supported	GTH_Quad_227
		X0Y2	X8 Not Supported	GTH_Quad_231
		X0Y3	GTH_Quad_233	GTH_Quad_232
	FLVD1517	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228	GTH_Quad_227
		X0Y2	GTH_Quad_230	GTH_Quad_229
		X0Y3	GTH_Quad_233	GTH_Quad_232
XCVU160	FLGB2104	X0Y1	GTH_Quad_225	GTH_Quad_224
		X0Y2	GTH_Quad_228	GTH_Quad_227
		X0Y3	GTH_Quad_230	GTH_Quad_229
		X0Y4	GTH_Quad_233	GTH_Quad_232
	FLGC2104	X0Y0	X8 Not Supported	GTH_Quad_222
		X0Y1	GTH_Quad_225	GTH_Quad_224
		X0Y2	GTH_Quad_228	GTH_Quad_227
		X0Y3	GTH_Quad_230	GTH_Quad_229
		X0Y4	GTH_Quad_233	GTH_Quad_232
XCVU190	FLGA2577	X0Y0	GTH_Quad_220	GTH_Quad_219
		X0Y1	GTH_Quad_223	GTH_Quad_222
		X0Y2	GTH_Quad_225	GTH_Quad_224
		X0Y3	GTH_Quad_228	GTH_Quad_227
		X0Y4	GTH_Quad_230	GTH_Quad_229
		X0Y5	GTH_Quad_233	GTH_Quad_232
	FLGB2104	X0Y2	GTH_Quad_225	GTH_Quad_224
		X0Y3	GTH_Quad_228	GTH_Quad_227
		X0Y4	GTH_Quad_230	GTH_Quad_229
		X0Y5	GTH_Quad_233	GTH_Quad_232
	FLGC2104	X0Y0	X8 Not Supported	GTH_Quad_220
		X0Y1	X8 Not Supported	GTH_Quad_222
		X0Y2	GTH_Quad_225	GTH_Quad_224
		X0Y3	GTH_Quad_228	GTH_Quad_227
		X0Y4	GTH_Quad_230	GTH_Quad_229
		X0Y5	GTH_Quad_233	GTH_Quad_232

Device	Package	PCIe Blocks	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCVU440	FLGA2892	X0Y0	GTH_Quad_220	GTH_Quad_219
		X0Y1	X8 Not Supported	GTH_Quad_222
		X0Y2	GTH_Quad_225	GTH_Quad_224
		X0Y3	X8 Not Supported	GTH_Quad_227
		X0Y4	GTH_Quad_230	GTH_Quad_229
		X0Y5	X8 Not Supported	GTH_Quad_232
	FLGB2377	X0Y0	X8 Not Supported	GTH_Quad_221
		X0Y1	GTH_Quad_223	GTH_Quad_222
		X0Y2	GTH_Quad_225	GTH_Quad_224
		X0Y3	X8 Not Supported	GTH_Quad_226
		X0Y4	X8 Not Supported	GTH_Quad_231
		X0Y5	GTH_Quad_233	GTH_Quad_232

Notes:

1. The GT quads highlighted in bold are the default GT quads for a given device, package, and PCIe block.

Kintex UltraScale Devices Available GT Quads

The following table shows the PCIe lane 0 GT Quad options available for the different AMD Kintex™ UltraScale™ devices. The GT Quad location is shown using the GT Quad bank number rather than GT XY coordinates. The *UltraScale and UltraScale+ FPGAs Packaging and Pinouts Product Specification* (UG575) provides a diagram describing the PCIe block locations relative to enabled GT Quads and includes both GT bank numbering and XY coordinates if needed.

***Note*:** The selections in bold are the default GT quads for a given device, package and PCIe block.

Table 88: **Kintex UltraScale Devices Available GT Quads**

Device	Package	PCIe Blocks	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCKU025	FFVA1156	X0Y0	GTH_Quad_225	GTH_Quad_224

Table 88: **Kintex UltraScale Devices Available GT Quads** *(cont'd)*

Device	Package	PCIe Blocks	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCKU035	FBVA676	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_227	GTH_Quad_226
		X0Y2	X8 Not Supported	GTH_Quad_227
	FBVA900	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_227	GTH_Quad_226
		X0Y2	X8 Not Supported	GTH_Quad_227
	FFVA1156	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_227	GTH_Quad_226
		X0Y2	X8 Not Supported	GTH_Quad_227
	SFVA784	X0Y0	GTH_Quad_225	GTH_Quad_224
XCKU040	FBVA676	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_227	GTH_Quad_226
		X0Y2	X8 Not Supported	GTH_Quad_227
	FBVA900	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_227	GTH_Quad_226
		X0Y2	X8 Not Supported	GTH_Quad_227
	FFVA1156	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X0Y2	GTH_Quad_228	GTH_Quad_227
	SFVA784	X0Y0	GTH_Quad_225	GTH_Quad_224
XQKU040	RFA1156	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X0Y2	GTH_Quad_228	GTH_Quad_227
	RBA676	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_227	GTH_Quad_226
		X0Y2	X8 Not Supported	GTH_Quad_227
XCKU060	FFVA1156	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X0Y2	GTH_Quad_228	GTH_Quad_227
	FFVA1517	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X0Y2	GTH_Quad_228	GTH_Quad_227

Table 88: **Kintex UltraScale Devices Available GT Quads** *(cont'd)*

Device	Package	PCIe Blocks	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XQKU060	RFA1156	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X0Y2	GTH_Quad_228	GTH_Quad_227
XCKU085	FLVA1517	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X0Y2	GTH_Quad_228	GTH_Quad_227
		X0Y3	GTH_Quad_230	GTH_Quad_229
		X0Y4	GTH_Quad_232	GTH_Quad_231
	FLVB1760	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X0Y2	GTH_Quad_228	GTH_Quad_227
		X0Y3	X8 Not Supported	GTH_Quad_230
		X0Y4	GTH_Quad_232	GTH_Quad_231
	FLVF1924	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X0Y2	GTH_Quad_228	GTH_Quad_227
		X0Y3	GTH_Quad_230	GTH_Quad_229
		X0Y4	GTH_Quad_232	GTH_Quad_231

Table 88: **Kintex UltraScale Devices Available GT Quads** *(cont'd)*

Device	Package	PCIe Blocks	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCKU115	FLVA1517	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X0Y2	GTH_Quad_228	GTH_Quad_227
		X0Y3	GTH_Quad_230	GTH_Quad_229
		X0Y4	GTH_Quad_232	GTH_Quad_231
		X0Y5	X8 Not Supported	GTH_Quad_232
	FLVB1760	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X0Y2	GTH_Quad_228	GTH_Quad_227
		X0Y3	X8 Not Supported	GTH_Quad_230
		X0Y4	GTH_Quad_233, GTH_Quad_232	GTH_Quad_231
		X0Y5	GTH_Quad_233	GTH_Quad_232
	FLVD1517	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X0Y2	GTH_Quad_228	GTH_Quad_227
		X0Y3	GTH_Quad_230	GTH_Quad_229
		X0Y4	GTH_Quad_233, GTH_Quad_232	GTH_Quad_231
		X0Y5	GTH_Quad_233	GTH_Quad_232
	FLVA2104	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_227	GTH_Quad_226
		X0Y2	X8 Not Supported	GTH_Quad_227
		X0Y3	X8 Not Supported	GTH_Quad_231
		X0Y4	GTH_Quad_233, GTH_Quad_232	GTH_Quad_231
		X0Y5	GTH_Quad_233	GTH_Quad_232

Table 88: **Kintex UltraScale Devices Available GT Quads** *(cont'd)*

Device	Package	PCIe Blocks	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCKU115 cont.	FLVB2104	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X0Y2	GTH_Quad_228	GTH_Quad_227
		X0Y3	GTH_Quad_230	GTH_Quad_229
		X0Y4	GTH_Quad_233, GTH_Quad_232	GTH_Quad_231
		X0Y5	GTH_Quad_233	GTH_Quad_232
	FLVD1924	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_227	GTH_Quad_226
		X0Y2	X8 Not Supported	GTH_Quad_227
		X0Y3	X8 Not Supported	GTH_Quad_231
		X0Y4	GTH_Quad_233, GTH_Quad_232	GTH_Quad_231
		X0Y5	GTH_Quad_233	GTH_Quad_232
	FLVF1924	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X0Y2	GTH_Quad_228	GTH_Quad_227
		X0Y3	GTH_Quad_230	GTH_Quad_229
		X0Y4	GTH_Quad_233, GTH_Quad_232	GTH_Quad_231
		X0Y5	GTH_Quad_233	GTH_Quad_232
XCKU095	FFVB1760	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227
		X0Y2	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228
		X0Y3	GTH_Quad_231	GTH_Quad_230
	FFVB2104	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_229, GTH_Quad_228	GTH_Quad_227
		X0Y2	GTH_Quad_230, GTH_Quad_229	GTH_Quad_228
		X0Y3	GTH_Quad_231	GTH_Quad_230

Table 88: **Kintex UltraScale Devices Available GT Quads** *(cont'd)*

Device	Package	PCIe Blocks	Quads with Max Link Width X8 Support	Quads with Max Link Width X4 Support
XCKU095 cont.	FFVA1156	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_227	GTH_Quad_226
		X0Y2	X8 Not Supported	GTH_Quad_228
	FFVC1517	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228	GTH_Quad_227
		X0Y2	X8 Not Supported	GTH_Quad_228
		X0Y3	X8 Not Supported	GTH_Quad_228
XQKU095	RFA1156	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_227	GTH_Quad_226
		X0Y2	X8 Not Supported	GTH_Quad_228
XQKU115	RLD1517	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X0Y2	GTH_Quad_228	GTH_Quad_227
		X0Y3	GTH_Quad_230	GTH_Quad_229
		X0Y4	GTH_Quad_233, GTH_Quad_232	GTH_Quad_231
		X0Y5	GTH_Quad_233	GTH_Quad_232
	RLF1924	X0Y0	GTH_Quad_225	GTH_Quad_224
		X0Y1	GTH_Quad_228, GTH_Quad_227	GTH_Quad_226
		X0Y2	GTH_Quad_228	GTH_Quad_227
		X0Y3	GTH_Quad_230	GTH_Quad_229
		X0Y4	GTH_Quad_233, GTH_Quad_232	GTH_Quad_231
		X0Y5	GTH_Quad_233	GTH_Quad_232

Notes:

1. The GT quads highlighted in bold are the default GT quads for a given device, package, and PCIe block.

Appendix C

Managing Receive-Buffer Space for Inbound Completions

The PCI Express® Base Specification requires all Endpoints to advertise infinite Flow Control credits for received Completions to their link partners. This means that an Endpoint must only transmit Non-Posted Requests for which it has space to accept Completion responses. This appendix describes how a user application can manage the receive-buffer space in the UltraScale Devices Gen3 Integrated Block for PCIe core to fulfill this requirement.

General Considerations and Concepts

Completion Space

The following table defines the completion space reserved in the receive buffer by the core. The values differ depending on the different Capability Max Payload Size settings of the core and the performance level that you selected. Values are credits, expressed in decimal.

Table 89: **Receiver-Buffer Completion Space**

Capability Max Payload Size (bytes)	Performance Level: Extreme	
	CPH	CPD
128	64	15,872B
256	64	15,872B
512	64	15,872B
1024	64	15,872B

Maximum Request Size

A Memory Read cannot request more than the value stated in Max_Request_Size, which is given by Configuration bits cfg_dcommand[14:12] as defined in the following table. If the user application does not read the Max_Request_Size value, it must use the default value of 128 bytes.

Table 90: **Max_Request_Size Settings**

cfg_dcommand[14:12]	Max_Request_Size			
	Bytes	DW	QW	Credits
000b	128	32	16	8
001b	256	64	32	16
010b	512	128	64	32
011b	1024	256	128	64
100b	2048	512	256	128
101b	4096	1024	512	256
110b-111b	Reserved			

Read Completion Boundary

A memory read can be answered with multiple completions, which when put together return all requested data. To make room for packet-header overhead, the user application must allocate enough space for the maximum number of completions that might be returned.

To make this process easier, the PCI Express Base Specification quantizes the length of all completion packets such that each completion must start and end on a naturally aligned read completion boundary (RCB), unless, it services the starting or ending address of the original request. Requests which cross the address boundaries at integer multiples of RCB bytes can be completed using more than one completion, but the returned data must not be fragmented except along the following address boundaries:

- The first completion must start with the address specified in the request, and must end at one of the following:
  - The address specified in the request plus the length specified by the request (for example, the entire request).
  - An address boundary between the start and end of the request at an integer multiple of RCB bytes.
- The final completion must end with the address specified in the request plus the length specified by the request.
- All completions between, but not including, the first and final completions must be an integer multiple of RCB bytes in length.

The programmed value of RCB is provided on `cfg_rcb_status[1:0]`. Here `cfg_rcb_status[0]` and `cfg_rcb_status[1]` are associated with physical functions 0 and 1 respectively (Per Function Link Control register [3]). If the user application does not read the RCB value, it must use the default value of 64 bytes.

Table 91: **Read Completion Boundary Settings**

cfg_rcb_status[0] or cfg_rcb_status[1]	Read Completion Boundary			
	Bytes	DW	QW	Credits
0	64	16	8	4
1	128	32	16	8

When calculating the number of completion credits a non-posted request requires, you must determine how many RCB-bounded blocks the completion response might be required, which is the same as the number of completion header credits required.

Important Note For High Performance Applications

While a programmed RCB value can be used by the user application to compute the maximum number of completions returned for a request, most high performance memory controllers have the optional feature to combine RCB-sized completions in response to large read requests (read lengths multiples of RCB value), into completions that are at or near the programmed Max_Payload_Size value for the link. You are encouraged to take advantage of this feature, if supported, by a memory controller on the host CPU. Data exchange based on completions that are integer multiples (>1) of RCB value results in greater PCI Express interface utilization and payload efficiency, as well as, more efficient use of completion space in the Endpoint receiver.

Methods of Managing Completion Space

A user application can choose one of five methods to manage receive-buffer completion space, as listed in the following table. For convenience, this discussion refers to these methods as LIMIT_FC, PACKET_FC, RCB_FC, and DATA_FC. Each method has advantages and disadvantages that you need to consider when developing the user application.

Table 92: **Managing Receive Completion Space Methods**

Method	Description	Advantage	Disadvantage
LIMIT_FC	Limit the total number of outstanding NP Requests	Simplest method to implement in user logic	Much Completion capacity goes unused
PACKET_FC	Track the number of outstanding CplH and CplD credits; allocate and deallocate on a per-packet basis	Relatively simple user logic; finer allocation granularity means less wasted capacity than LIMIT_FC	As with LIMIT_FC, credits for an NP are still tied up until the request is completely satisfied
RCB_FC	Track the number of outstanding CplH and CplD credits; allocate and deallocate on a per-RCB basis	Ties up credits for less time than PACKET_FC	More complex user logic than LIMIT_FC or PACKET_FC

Table 92: **Managing Receive Completion Space Methods** *(cont'd)*

Method	Description	Advantage	Disadvantage
DATA_FC	Track the number of outstanding CplH and CplD credits; allocate and deallocate on a per-RCB basis	Lowest amount of wasted capacity	More complex user logic than LIMIT_FC, PACKET_FC, and RCB_FC

LIMIT_FC Method

The LIMIT_FC method is the simplest to implement. The user application assesses the maximum number of outstanding Non-Posted Requests allowed at one time, MAX_NP. To calculate this value, perform these steps:

1. Determine the number of CplH credits required by a Max_Request_Size packet:

   Max_Header_Count = ceiling(Max_Request_Size / RCB)

2. Determine the greatest number of maximum-sized completions supported by the CplD credit pool:

   Max_Packet_Count_CplD = floor(CplD / Max_Request_Size)

3. Determine the greatest number of maximum-sized completions supported by the CplH credit pool:

   Max_Packet_Count_CplH = floor(CplH / Max_Header_Count)

4. Use the smaller of the two quantities from steps 2 and 3 to obtain the maximum number of outstanding Non-Posted requests:

   MAX_NP = min(Max_Packet_Count_CplH, Max_Packet_Count_CplD)

With knowledge of MAX_NP, the user application can load a register NP_PENDING with zero at reset and make sure it always stays with the range 0 to MAX_NP. When a non-posted request is transmitted, NP_PENDING decreases by one. When all completions for an outstanding non-posted request are received, NP_PENDING increases by one.

For example:

- Max_Request_Size = 128B
- RCB = 64B
- CplH = 64
- CplD = 15,872B
- Max_Header_Count = 2
- Max_Packet_Count_CplD = 124
- Max_Packet_Count_CplH = 32

- MAX_NP = 32

Although this method is the simplest to implement, it can waste the greatest receiver space because an entire Max_Request_Size block of completion credit is allocated for each non-posted request, regardless of actual request size. The amount of waste becomes greater when the user application issues a larger proportion of short memory reads (on the order of a single DWORD), I/O reads and I/O writes.

PACKET_FC Method

The PACKET_FC method allocates blocks of credit in finer granularities than LIMIT_FC, using the receive completion space more efficiently with a small increase in user logic.

Start with two registers, CPLH_PENDING and CPLD_PENDING, (loaded with zero at reset), and then perform these steps:

1. When the user application needs to send an NP request, determine the potential number of CplH and CplD credits it might require:

   NP_CplH = ceiling[((Start_Address mod RCB) + Request_Size) / RCB]

   NP_CplD = ceiling[((Start_Address mod 16 bytes) + Request_Size) /16 bytes]

   (except I/O Write, which returns zero data) [(req_size + 15)/16]

   The modulo and ceiling functions ensure that any fractional RCB or credit blocks are rounded up. For example, if a memory read requests 8 bytes of data from address `7Ch`, the returned data can potentially be returned over two completion packets (`7Ch` - `7Fh`, followed by `80h` - `83h`). This would require two RCB blocks and two data credits.

2. Check these:

   CPLH_PENDING + NP_CplH < Total_CplH

   CPLD_PENDING + NP_CplD < Total_CplD

3. If both inequalities are true, transmit the non-posted request, and increase CPLH_PENDING by NP_CplH and CPLD_PENDING by NP_CplD. For each non-posted request transmitted, keep NP_CplH and NP_CplD for later use.

4. When all completion data is returned for an non-posted request, decrease CPLH_PENDING and CPLD_PENDING accordingly.

   This method is less wasteful than LIMIT_FC but still ties up all of an non-posted request completion space until the entire request is satisfied. RCB_FC and DATA_FC provide finer de-allocation granularity at the expense of more logic.

RCB_FC Method

The RCB_FC method allocates and de-allocates blocks of credit in RCB granularity. Credit is freed on a per-RCB basis.

As with PACKET_FC, start with two registers, CPLH_PENDING and CPLD_PENDING (loaded with zero at reset).

1. Calculate the number of data credits per RCB:

   CplD_PER_RCB = RCB / 16 bytes

2. When the user application needs to send an non-posted request, determine the potential number of CplH credits it might require. Use this to allocate CplD credits with RCB granularity:

   NP_CplH = ceiling[((Start_Address mod RCB) + Request_Size) / RCB]

   NP_CplD = NP_CplH × CplD_PER_RCB

3. Check these:

   CPLH_PENDING + NP_CplH < Total_CplH

   CPLD_PENDING + NP_CplD < Total_CplD

4. If both inequalities are true, transmit the non-posted request, increase CPLH_PENDING by NP_CplH and CPLD_PENDING by NP_CplD.

5. At the start of each incoming completion, or when that completion begins at or crosses an RCB without ending at that RCB, decrease CPLH_PENDING by 1 and CPLD_PENDING by CplD_PER_RCB. Any completion could cross more than one RCB. The number of RCB crossings can be calculated by:

   RCB_CROSSED = ceiling[((Lower_Address mod RCB) + Length) / RCB]

   Lower_Address and Length are fields that can be parsed from the Completion header. Alternatively, you can load a register CUR_ADDR with Lower_Address at the start of each incoming completion, increment per DW or QW as appropriate, then count an RCB whenever CUR_ADDR rolls over.

This method is less wasteful than PACKET_FC but still gives an RCB granularity. If a user application transmits I/O requests, the user application could adopt a policy of only allocating one CplD credit for each I/O read and zero CplD credits for each I/O write. The user application would have to match each tag for incoming completions with the type (Memory Write, I/O Read, I/O Write) of the original non-posted request.

DATA_FC Method

The DATA_FC method provides the finest allocation granularity at the expense of logic.

As with PACKET_FC and RCB_FC, start with two registers, CPLH_PENDING and CPLD_PENDING (loaded with zero at reset).

1. When the user application needs to send an non-posted request, determine the potential number of CplH and CplD credits it might require:

   NP_CplH = ceiling[((Start_Address mod RCB) + Request_Size) / RCB]

   NP_CplD = ceiling[((Start_Address mod 16 bytes) + Request_Size) / 16 bytes]

   (except I/O Write, which returns zero data)

2. Check these:

   CPLH_PENDING + NP_CplH < Total_CplH

   CPLD_PENDING + NP_CplD < Total_CplD

3. If both inequalities are true, transmit the non-posted request, increase CPLH_PENDING by NP_CplH and CPLD_PENDING by NP_CplD.

4. At the start of each incoming completion, or when that completion begins at or crosses an RCB without ending at that RCB, decrease CPLH_PENDING by 1. The number of RCB crossings can be calculated by:

   RCB_CROSSED = ceiling[((Lower_Address mod RCB) + Length) / RCB]

   Lower_Address and Length are fields that can be parsed from the completion header. Alternatively, you can load a register CUR_ADDR with Lower_Address at the start of each incoming completion, increment per DW or QW as appropriate, then count an RCB whenever CUR_ADDR rolls over.

5. At the start of each incoming completion, or when that completion begins at or crosses at a naturally aligned credit boundary, decrease CPLD_PENDING by 1. The number of credit-boundary crossings is given by:

   DATA_CROSSED = ceiling[((Lower_Address mod 16 B) + Length) / 16 B]

Alternatively, you can load a register CUR_ADDR with Lower_Address at the start of each incoming completion, increment per DW or QW as appropriate, then count an RCB whenever CUR_ADDR rolls over each 16-byte address boundary.

This method is the least wasteful but requires the greatest amount of user logic. If even finer granularity is desired, you can scale the Total_CplD value by 2 or 4 to get the number of completion QWORDs or DWORDs, respectively, and adjust the data calculations accordingly.

Debugging

This appendix includes details about resources available on the AMD Support website and debugging tools.

If the IP requires a license key, the key must be verified. The AMD Vivado™ design tools have several license checkpoints for gating licensed IP through the flow. If the license check succeeds, the IP can continue generation. Otherwise, generation halts with an error. License checkpoints are enforced by the following tools:

- Vivado Synthesis
- Vivado Implementation
- write_bitstream (Tcl command)

IMPORTANT! *IP license level is ignored at checkpoints. The test confirms a valid license exists. It does not check IP license level.*

Finding Help with AMD Adaptive Computing Solutions

To help in the design and debug process when using the core, the Support web page contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support. The Community Forums are also available where members can learn, participate, share, and ask questions about AMD Adaptive Computing solutions.

Documentation

This product guide is the main document associated with the core. This guide, along with documentation related to all products that aid in the design process, can be found on the AMD Adaptive Support web page or by using the AMD Adaptive Computing Documentation Navigator. Download the 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 an AMD Adaptive Computing product. Answer Records are created and maintained daily to ensure 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 AMD Adaptive Support web page. To maximize your search results, use keywords such as:

- Product name
- Tool message(s)
- Summary of the issue encountered

A filter search is available after results are returned to further target the results.

Master Answer Record for the Core

AR: 57945.

Technical Support

AMD Adaptive Computing provides technical support on the Community Forums for this AMD LogiCORE™ IP product when used as described in the product documentation. AMD Adaptive Computing cannot guarantee timing, functionality, or support if you do any of the following:

- Implement the solution in devices that are not defined in the documentation.
- Customize the solution beyond that allowed in the product documentation.
- Change any section of the design labeled DO NOT MODIFY.

To ask questions, navigate to the Community Forums.

Hardware Debug

Transceiver Control and Status Ports

The following table describes the ports used to debug transceiver related issues.

IMPORTANT! *The ports in the Transceiver Control And Status Interface must be driven in accordance with the appropriate GT user guide. Using the input signals listed in the following table might result in unpredictable behavior of the IP core.*

Table 93: **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
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	16	Connects to DMONITOROUT on transceiver channel primitives
gt_rxcommadet	O	1	Connects to RXCOMMADETEN on transceiver channel primitives
gt_phystatus	O	1	Connects to PHYSTATUS on transceiver channel primitives
gt_rxvalid	O	1	Connects to RXVALID on transceiver channel primitives
gt_rxcdrlock	O	1	Connects to RXCDRLOCK on transceiver channel primitives
gt_pcierateidle	O	1	Connects to PCIERATEIDLE on transceiver channel primitives

Table 93: **Ports Used for Transceiver Debug** *(cont'd)*

Port	Direction	Width	Description
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
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

AMD

Appendix E

Using the Xilinx Virtual Cable to Debug

Introduction

The Xilinx Virtual Cable (XVC) allows the AMD 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 AMD 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 AMD 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. Because 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. Because 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 AMD interfaces or peripherals such as the PCIe MCAP VSEC and HWICAP IP*

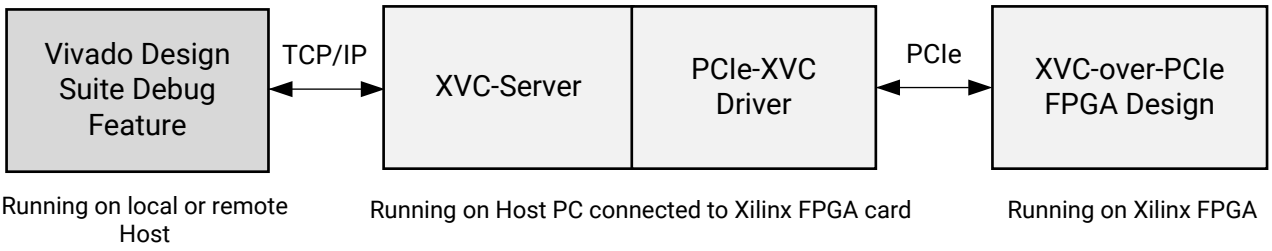
Overview

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 AMD FPGA designs. These three components are shown in the following figure and connect to the Vivado Design Suite debug feature through a TCP/IP socket.

Figure 94: **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 AMD `hw_server` using TCP/IP socket. This allows Vivado (using `hw_server`) and the XVC-Server application to be running on the same PC or separate PCs connected through Ethernet. The XVC-Server application needs to be run on a PC that is directly connected to the FPGA hardware resource. In this scenario the FPGA hardware is connected through PCIe to a Host PC. The XVC-Server application connects to the FPGA hardware device through the PCIe-XVC driver that is also running on the Host PC.

Host PC XVC-over-PCIe Driver

The XVC-over-PCIe driver provides connectivity to the PCIe enabled FPGA hardware resource that is connected to the Host PC. As such this is provided as a Linux kernel mode driver to access the PCIe hardware device, which is available in the following location, `<Vivado_Installation_Path>/data/xicom/driver/pcie/xvc_pcie.zip`. The necessary components of this driver must be added to the driver that is created for a specific FPGA platform. The driver implements the basic functions needed by the XVC-Server application to communicate with the FPGA via PCIe.

XVC-over-PCIe Enabled FPGA Design

Traditionally Vivado debug is performed over JTAG. By default, Vivado debug automation connects the AMD 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 AMD 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).

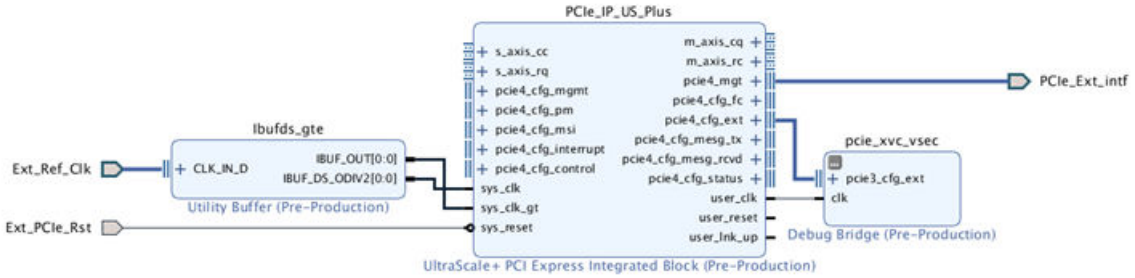
The Debug Bridge IP, when configured for From PCIe to BSCAN or From AXI to BSCAN , provides a connection point for the AMD debug network from either the PCIe Extended Capability or AXI4-Lite interfaces respectively. Vivado debug automation connects this instance of the Debug Bridge to the AMD debug cores found in the design rather than connecting them to the JTAG BSCAN interface. There are design trade-offs to connecting the debug bridge to the PCIe Extended Configuration Space or AXI4-Lite. The following sections describe the implementation considerations and register map for both implementations.

XVC-over-PCIe Through PCIe Extended Configuration Space (PCIe-XVC-VSEC)

Using the PCIe-XVC-VSEC approach, the Debug Bridge IP uses a PCIe Vendor Specific Extended Capability (VSEC) to implement the connection from PCIe to the Debug Bridge IP. The PCIe extended configuration space is set up as a linked list of extended capabilities that are discoverable from a Host PC. This is specifically valuable for platforms where one version of the design implements the PCIe-XVC-VSEC and another design implementation does not. The linked list can be used to detect the existence or absence of the PCIe-XVC-VSEC and respond accordingly.

The PCIe Extended Configuration Interface uses PCIe configuration transactions rather than PCIe memory BAR transactions. While PCIe configuration transactions are much slower, they do not interfere with PCIe memory BAR transactions at the PCIe IP boundary. This allows for separate data and debug communication paths within the FPGA. This is ideal if you expect to debug the datapath. Even if the datapath becomes corrupt or halted, the PCIe Extended Configuration Interface can remain operational to perform debug. The following figure describes the connectivity between the PCIe IP and the Debug Bridge IP to implement the PCIe-XVC-VSEC.

Figure 95: **XVC-over-PCIe with PCIe Extended Capability Interface**



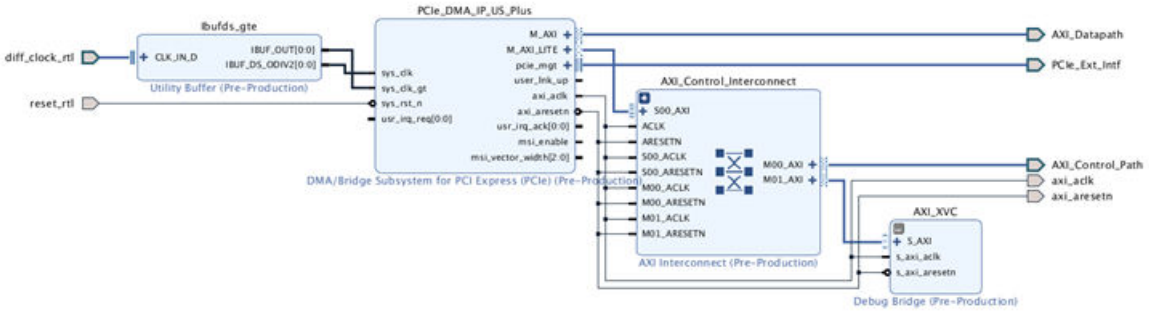
***Note*:** Although the preceding figure shows the UltraScale+ Integrated Block for PCIe IP, other PCIe IP (that is, the UltraScale Integrated Block for PCIe, AXI Bridge for PCIe, or PCIe DMA IP) can be used interchangeably in this diagram.

XVC-over-PCIe Through AXI (AXI-XVC)

Using the AXI-XVC approach, the Debug Bridge IP connects to the PCIe IP through an AXI Interconnect IP. The Debug Bridge IP connects to the AXI Interconnect like other AXI4-Lite Slave IPs and similarly requires that a specific address range be assigned to it. Traditionally the `debug_bridge` IP in this configuration is connected to the control path network rather than the system datapath network. The following figure describes the connectivity between the

DMA Subsystem for PCIe IP and the Debug Bridge IP for this implementation.

Figure 96: **XVC over PCIe with PCIe Interface**



***Note*:** Although the preceding figure shows the PCIe DMA IP, any AXI-enabled PCIe IP can be used interchangeably in this diagram.

The AXI-XVC implementation allows for higher speed transactions. However, XVC debug traffic passes through the same PCIe ports and interconnect as other PCIe control path traffic, making it more difficult to debug transactions along this path. As result the AXI-XVC debug should be used to debug a specific peripheral or a different AXI network rather than attempting to debug datapaths that overlap with the AXI-XVC debug communication path.

XVC-over-PCIe Register Map

The PCIe-XVC-VSEC and AXI-XVC have a slightly different register map that must be taken into account when designing XVC drivers and software. The register maps in the following tables show the byte-offset from the base address.

- The PCIe-XVC-VSEC base address must fall within the valid range of the PCIe Extended Configuration space. This is specified in the Debug Bridge IP configuration.
- The base address of an AXI-XVC Debug Bridge is the offset for the Debug Bridge IP peripheral that was specified in the Vivado Address Editor.

The following tables describe the register map for the Debug Bridge IP as an offset from the base address when configured for the From PCIe-Ext to BSCAN or From AXI to BSCAN modes.

Table 94: **Debug Bridge for XVC-PCIe-VSEC Register Map**

Register Offset	Register Name	Description	Register Type
0x00	PCIe Ext Capability Header	PCIe defined fields for VSEC use.	Read Only
0x04	PCIe VSEC Header	PCIe defined fields for VSEC use.	Read Only
0x08	XVC Version Register	IP version and capabilities information.	Read Only
0x0C	XVC Shift Length Register	Shift length.	Read Write
0x10	XVC TMS Register	TMS data.	Read Write
0x14	XVC TDIO Register	TDO/TDI data.	Read Write
0x18	XVC Control Register	General control register.	Read Write
0x1C	XVC Status Register	General status register.	Read Only

Table 95: **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

Table 95: **Debug Bridge for AXI-XVC Register Map** *(cont'd)*

Register Offset	Register Name	Description	Register Type
0x10	XVC Control Register	General control register.	Read Write
0x14	XVC Status Register	General status register.	Read Only
0x18	XVC Version Register	IP version and capabilities information.	Read Only

PCIe Ext Capability Header

This register is used to identify the PCIe-XVC-VSEC added to a PCIe design. The fields and values in the PCIe Ext Capability Header are defined by PCI-SIG and are used to identify the format of the extended capability and provide a pointer to the next extended capability, if applicable. When used as a PCIe-XVC-VSEC, the appropriate PCIe ID fields should be evaluated prior to interpretation. These can include PCIe Vendor ID, PCIe Device ID, PCIe Revision ID, Subsystem Vendor ID, and Subsystem ID. The provided drivers specifically check for a PCIe Vendor ID that matches AMD (0x10EE) before interpreting this register. The following table describes the fields within this register.

Table 96: **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 valid range of the PCIe Extended Configuration space.	0x000	Read Only

PCIe VSEC Header (PCIe-XVC-VSEC only)

This register is used to identify the PCIe-XVC-VSEC when the Debug Bridge IP is in this mode. The fields are defined by PCI-SIG, but the values are specific to the Vendor ID (0x10EE for AMD). The PCIe Ext Capability Header register values should be qualified prior to interpreting this register.

Table 97: **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 AMD).	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

This register is populated by the AMD 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 Status Register

This register is used for XVC status information.

XVC Driver and Software

Example XVC driver and software has been provided with the Vivado Design Suite installation, which is available at the following location: `<Vivado_Installation_Path>/data/xicom/driver/pcie/xvc_pcie.zip`. This should be used for reference when integrating the XVC capability into AMD 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 AMD FPGA and extracted for use. `README.txt` files have been included; review these files for instructions on installing and running the XVC drivers and software.

Special Considerations for Tandem or Dynamic Function eXchange Designs

Tandem and Dynamic Function eXchange (DFX) designs might require additional considerations as these flows partition the physical resources into separate regions. These physical partitions should be considered when adding debug IPs to a design, such as VIO, ILA, MDM, and MIG-IP. A Debug Bridge IP configured for From PCIe-ext to BSCAN or From AXI to BSCAN should only be placed into the static partition of the design. When debug IPs are used inside of a DFX or Tandem Field Updates region, an additional debug BSCAN interface should be added to the DFX region module definition and left unconnected in the DFX region module instantiation.

To add the BSCAN interface to the DFX module definition the appropriate ports and port attributes should be added to the DFX 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 DFX module definition

// and left unconnected in the DFX module instantiation.

(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN drck" *)

(* DEBUG="true" *)
```

```
input S_BSCAN_drck,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN shift"
*)
(* DEBUG="true" *)
input S_BSCAN_shift,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN tdi" *)
(* DEBUG="true" *)
input S_BSCAN_tdi,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN update"
*)
(* DEBUG="true" *)
input S_BSCAN_update,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN sel" *)
(* DEBUG="true" *)
input S_BSCAN_sel,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN tdo" *)
(* DEBUG="true" *)
output S_BSCAN_tdo,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN tms" *)
(* DEBUG="true" *)
input S_BSCAN_tms,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN tck" *)
(* DEBUG="true" *)
input S_BSCAN_tck,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN 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 might also be an additional `dbg_hub` cell that is added at the top level of the design. For Tandem with Field Updates designs, the dbg_hub and tool inserted clock buffer(s) must be added to the appropriate design partition. The following is an example of the Tcl commands that can be run after `opt_design` to associate the `dbg_hub` primitives with the appropriate design partitions.

```
# Add the inserted dbg_hub cell to the appropriate design partition.

set_property HD.TANDEM_IP_PBLOCK Stage1_Main [get_cells dbg_hub]

# Add the clock buffer to the appropriate design partition.

set_property HD.TANDEM_IP_PBLOCK Stage1_Config_IO [get_cells
dma_pcie_0_support_i/pcie_ext_cap_i/vsec_xvc_inst/
vsec_xvc_dbg_bridge_inst/inst/bsip/ins

t/USE_SOFTBSCAN.U_TAP_TCKBUFG]
```

Using the PCIe-XVC-VSEC Example Design

The PCIe-XVC-VSEC has been integrated into the PCIe example design as part of the Advanced settings for the UltraScale+ 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

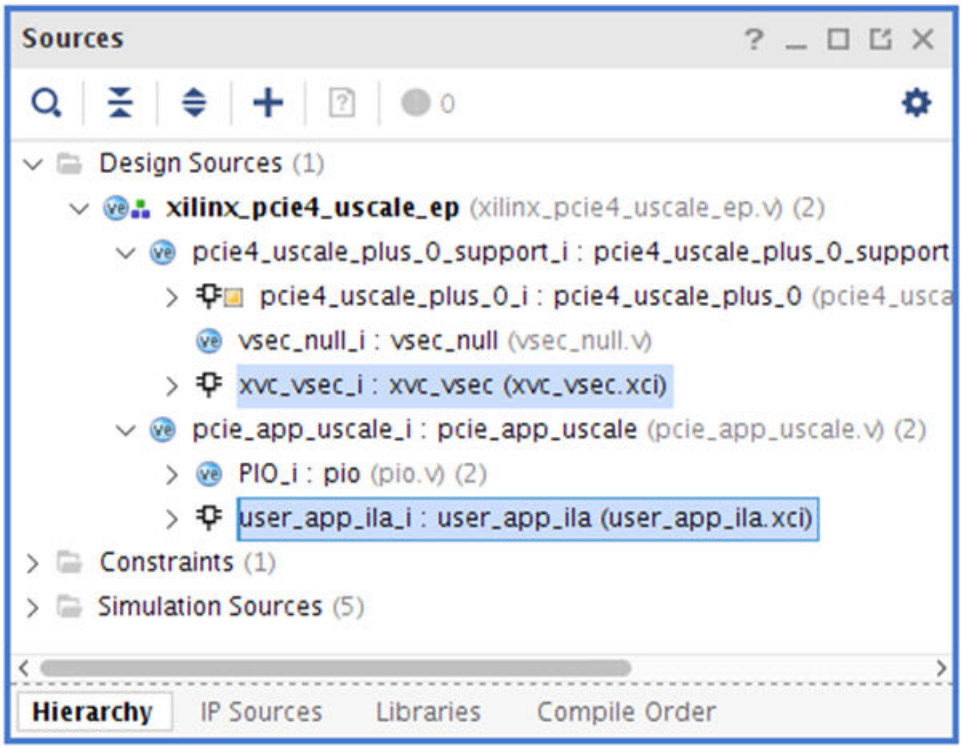
The PCIe-XVC-VSEC can be added to the AMD UltraScale+™ PCIe example design by selecting the following options.

1. Configure the core to the desired configuration.
2. On the Basic tab, select the **Advanced** Mode.
3. On the Adv. Options-3 tab:
   a. Select the **PCI Express Extended Configuration Space Enable** checkbox to enable the PCI Express extended configuration interface. This is where additional extended capabilities can be added to the PCI Express core.
   b. Select the **Add the PCIe-XVC-VSEC to the Example Design** checkbox to enable the PCIe-XVC-VSEC in the example design generation.
4. Verify the other configuration selections for the PCIe IP. The following selections are needed to configure the driver for your hardware implementation.
   - PCIe Vendor ID (0x10EE for AMD)
   - PCIe Device ID (dependent on user selection)
5. Click **OK** to finalize the selection and generate the IP.
6. Generate the output products for the IP as desired for your application.
7. In the Sources window, right-click the IP and select **Open IP Example Design**.
8. Select a directory for generating the example design, and select **OK**.

   After being generated, the example design shows that:

   - the PCIe IP is connected to `xvc_vsec` within the support wrapper, and
   - an ILA IP is added to the user application portion of the design.

   This demonstrates the desired connectivity for the hardware portion of the FPGA design. Additional debug cores can be added as required by your application.



***Note*:** Although the preceding figure shows to the UltraScale+ Integrated Block for PCIe IP, the example design hierarchy is the same for other PCIe IPs.

9. Double-click the Debug Bridge IP identified as `xvc_vsec` to view the configuration option for this IP. Make note of the following configuration parameters because they are 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 an AMD 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 is then loaded onto the FPGA board to enable XVC debug over PCIe.

After the XVC-over-PCIe hardware design has been completed, an appropriate XVC enabled PCIe driver and associated XVC-Server software application can be used to connect the Vivado Design Suite to the PCIe connected FPGA. Vivado can connect to an XVC-Server application that is running local on the same Machine or remotely on another machine using a TCP/IP socket.

System Bring-Up

The first step is to program the FPGA and power on the system such that the PCIe link is detected by the Host system. This can be accomplished by either:

- programming the design file into the flash present on the FPGA board, or
- programming the device directly via JTAG.

If the card is powered by the Host PC, it will need to be powered on to perform this programming using JTAG and then re-started to allow the PCIe link to enumerate. After the system is up and running, you can use the Linux `lspci` utility to list out the details for the FPGA-based PCIe device.

Compiling and Loading the Driver

The provided PCIe drivers and software should be customized to a specific platform. To accomplish this, drivers and software are normally developed to verify the Vendor ID, Device ID, Revision ID, Subsystem Vendor ID, and Subsystem ID before attempting to access device-extended capabilities or peripherals like the PCIe-XVC-VSEC or AXI-XVC. Because the provided driver is generic, it only verifies the Vendor ID and Device ID for compatibility before attempting to identify the PCIe-XVC-VSEC or AXI-XVC peripheral.

The XVC driver and software are provide as a ZIP file included with the Vivado Design Suite installation.

1. Copy the ZIP file from the Vivado install directory to the FPGA connected Host PC and extract (unzip) its contents. This file is located at the following path within the Vivado installation directory.

   ```
   XVC Driver and SW Path: …/data/xicom/driver/pcie/xvc_pcie.zip
   ```

   The `README.txt` files within the `driver_*` and `xvcserver` directories identify how to compile, install, and run the XVC drivers and software, and are summarized in the following steps. Follow the following steps after the driver and software files have been copied to the Host PC and you are logged in as a user with root permissions.

2. Modify the variables within the `driver_*/xvc_pcie_user_config.h` file to match your hardware design and IP settings. Consider modifying the following variables.

   - **PCIE_VENDOR_ID:** The PCIe Vendor ID defined in the PCIe IP customization.
   - **PCIE_DEVICE_ID:** The PCIe Device ID defined in the PCIe IP customization.
   - **Config_space:** Allows for the selection between using a PCIe-XVC-VSEC or an AXI-XVC peripheral. The default value of *AUTO* first attempts to discover the PCIe-XVC-VSEC, then attempts to connect to an AXI-XVC peripheral if the PCIe-XVC-VSEC is not found. A value of *CONFIG* or *BAR* can be used to explicitly select between PCIe-XVC-VSEC and AXI-XVC implementations, as desired.

- **config_vsec_id:** The PCIe XVC VSEC ID (default 0x0008) defined in the Debug Bridge IP when the Bridge Type is configured for *From PCIE to BSCAN*. This value is only used for detection of the PCIe-XVC-VSEC.
- **config_vsec_rev:** The PCIe XVC VSEC Rev ID (default 0x0) defined in the Debug Bridge IP when the Bridge Type is configured for *From PCIe to BSCAN*. This value is only used for detection of the PCIe-XVC-VSEC.
- **bar_index:** The PCIe BAR index that should be used to access the Debug Bridge IP when the Bridge Type is configured for *From AXI to BSCAN* . This BAR index is specified as a combination of the PCIe IP customization and the addressable AXI peripherals in your system design. This value is only used for detection of an AXI-XVC peripheral.
- **bar_offset:** PCIe BAR Offset that should be used to access the Debug Bridge IP when the Bridge Type is configured for *From AXI to BSCAN*. This BAR offset is specified as a combination of the PCIe IP customization and the addressable AXI peripherals in your system design. This value is only used for detection of an AXI-XVC peripheral.

3. Move the source files to the directory of your choice. For example, use:

   ```
   /home/username/xil_xvc or /usr/local/src/xil_xvc
   ```

4. Make sure you have root permissions and change to the directory containing the driver files.

   ```
   # cd /driver_*/
   ```

5. Compile the driver module:

   ```
   # make install
   ```

   The kernel module object file will be installed as:

   ```
   /lib/modules/[KERNEL_VERSION]/kernel/drivers/pci/pcie/Xilinx/
   xil_xvc_driver.ko
   ```

6. Run `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 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.

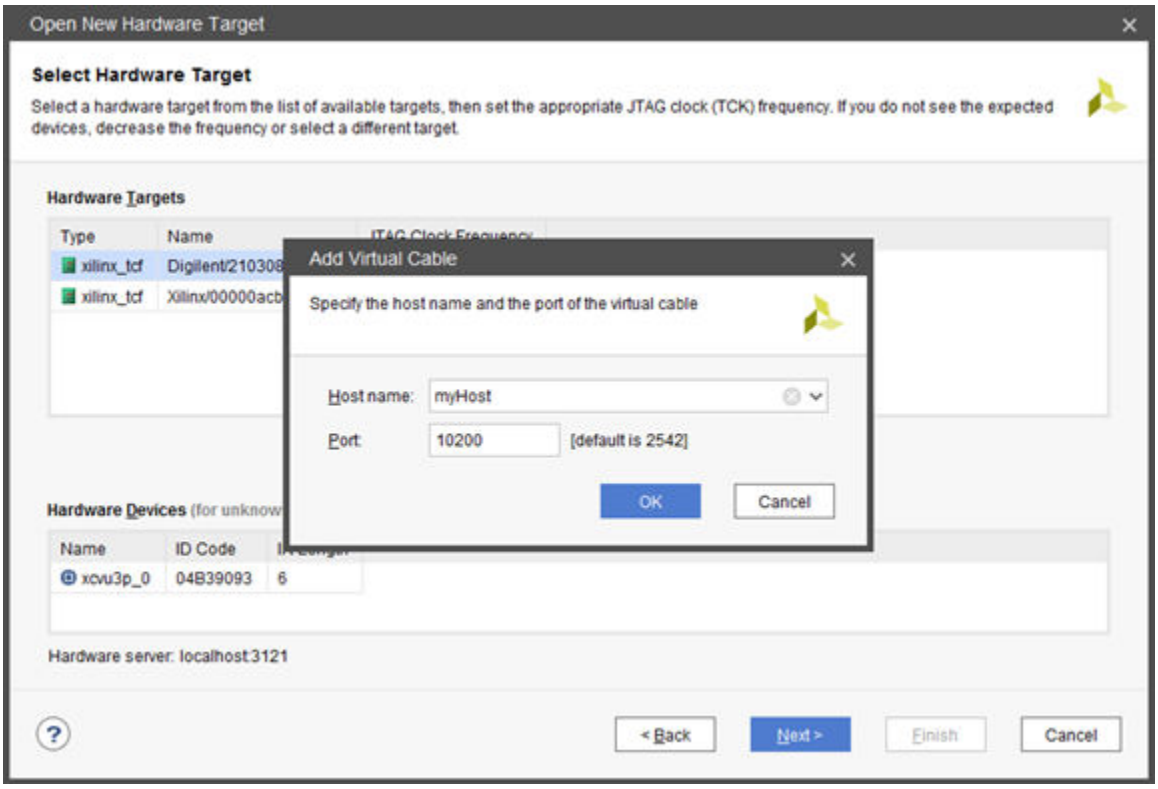
   ```
   Enable verbose by setting VERBOSE evn var.

   Opening /dev/xil_xvc/cfg_ioc0
   ```

Connecting the Vivado Design Suite to the XVC-Server Application

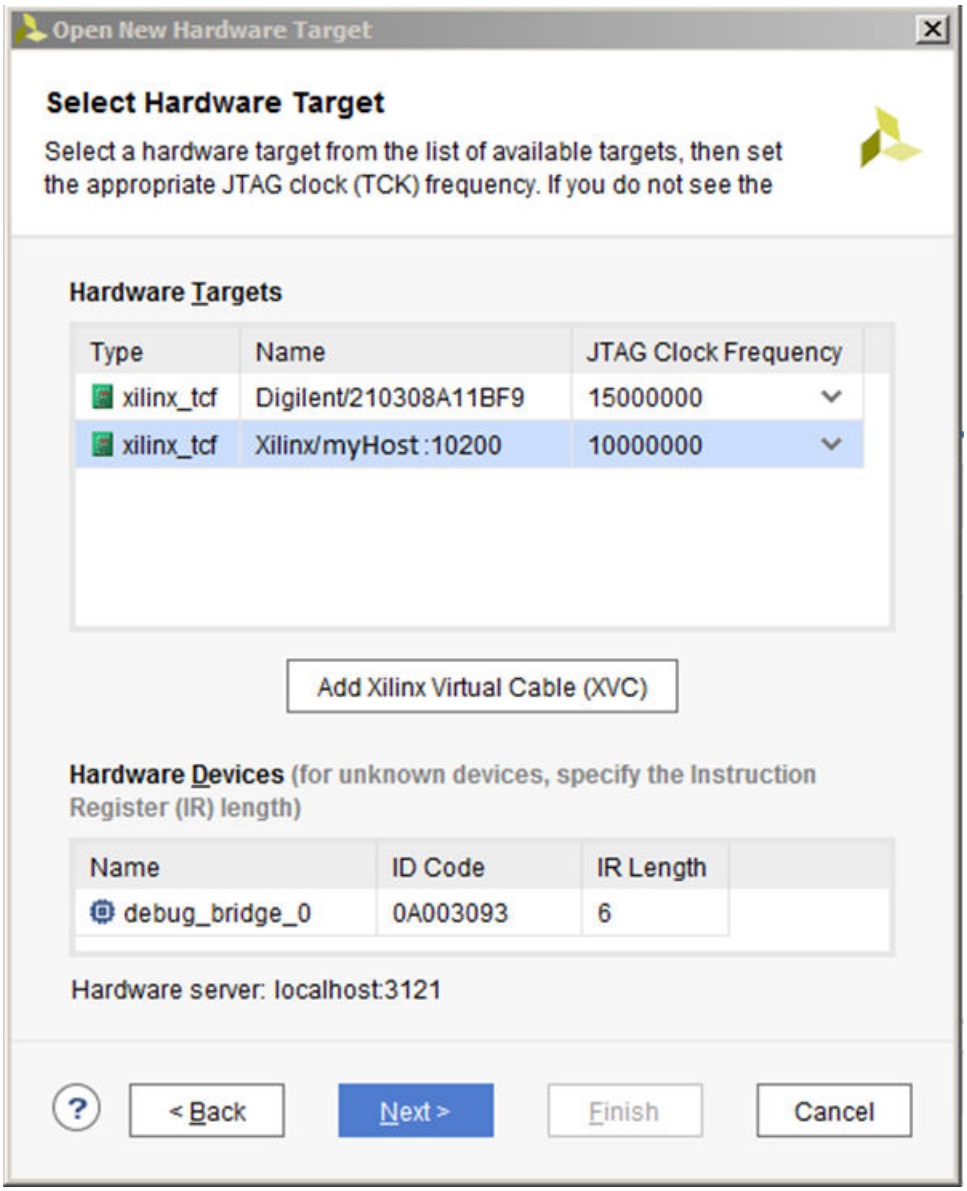
The Vivado Design Suite can be run on the computer that is running the XVC-server application, or it can be run remotely on another computer that is connected over an Ethernet network. The port however must be accessible to the machine running Vivado. To connect Vivado to the XVC-Server application follow 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** .

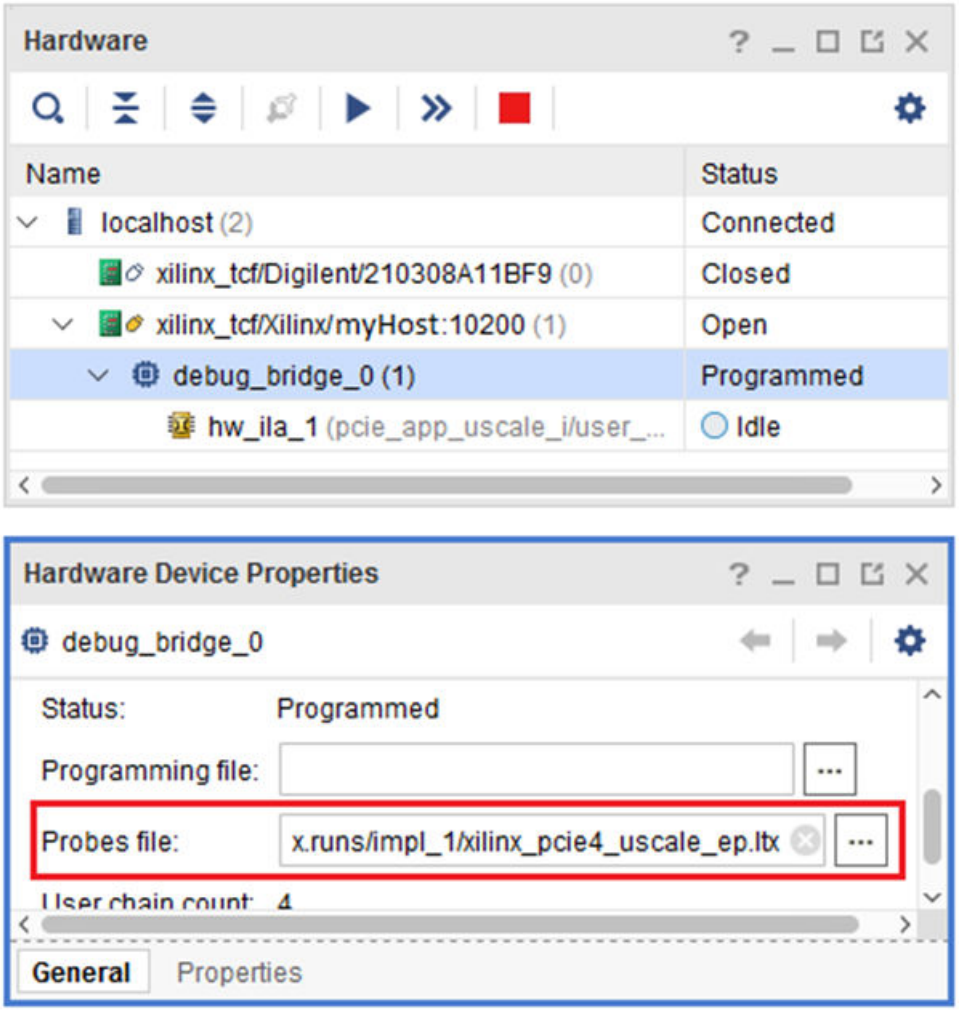
   This launches **hw_server** on the local machine, which then connects to the **xvcserver** application.
6. Select **Add Xilinx Virtual Cable (XVC)**.
7. In the Add Virtual Cable dialog box, type in the appropriate Host name or IP address, and Port to connect to the **xvcserver** application. Click **OK**.



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



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



Vivado now recognizes your debug cores and debug signals, and you can debug your design through the Vivado hardware tools interface using the standard debug approach.

This allows you to debug AMD 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 also be lost.

Runtime Considerations

The Vivado connection to an XVC-Server Application should not be running when a device is programmed. The XVC-Server Application along with the associated connection to Vivado should only be initiated after the device has been programmed and the hardware PCIe interface is active.

For DFX designs, it is important to terminate the connection during DFX operations. During a DFX operation where debug cores are present inside the DFX region, a portion of the debug tree is expected to be reprogrammed. Vivado debug tools should not be actively communicating with the FPGA through XVC during a DFX operation.

AMD

Appendix F

Additional Resources and Legal Notices

Finding Additional Documentation

Technical Information Portal

The AMD Technical Information Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Technical Information Portal, go to https://docs.amd.com.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the AMD Vivado™ IDE, select **Help → Documentation and Tutorials**.
- On Windows, click the **Start** button and select **Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

***Note*:** For more information on DocNav, refer to the *Documentation Navigator User Guide* (UG968).

Design Hubs

AMD Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- Go to the Design Hubs web page.

Support Resources

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

References

These documents provide supplemental material useful with this guide:

1. AMBA AXI4-Stream Protocol Specification
2. PCI-SIG Documentation
3. *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213)
4. *AXI Bridge for PCI Express Gen3 Subsystem Product Guide* (PG194)
5. *DMA/Bridge Subsystem for PCI Express Product Guide* (PG195)
6. *Virtex 7 FPGA Integrated Block for PCI Express LogiCORE IP Product Guide* (PG023)
7. *UltraScale Architecture Configuration User Guide* (UG570)
8. *Kintex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics* (DS892)
9. *Virtex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics* (DS893)
10. *UltraScale Architecture PCB Design User Guide* (UG583)
11. *UltraScale and UltraScale+ FPGAs Packaging and Pinouts Product Specification* (UG575)
12. *UltraScale Architecture GTH Transceivers User Guide* (UG576)
13. *UltraScale Architecture GTY Transceivers User Guide* (UG578)
14. *Vivado Design Suite User Guide: Designing with IP* (UG896)
15. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
16. *Vivado Design Suite User Guide: Getting Started* (UG910)
17. *Vivado Design Suite User Guide: Using Constraints* (UG903)
18. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
19. *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909)
20. *ISE to Vivado Design Suite Migration Guide* (UG911)
21. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
22. *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947)
23. *Debug Bridge LogiCORE IP Product Guide* (PG245)

24. *In-System IBERT LogiCORE IP Product Guide* (PG246)
25. *PIPE Mode Simulation Using Integrated Endpoint PCI Express Block in Gen2 x8 and Gen3 x8 Configurations* (XAPP1184)
26. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
12/06/2024 Version 4.4	
Requester Completion Interface Operation	Updated figure.
11/24/2023 Version 4.4	
N/A	• Updated PCIe DRP Ports table. • Updated Enable In System IBERT section.
12/12/2022 Version 4.4	
N/A	• Updated GT Settings Tab figure. • Added GT DRP Clock Source section. • Added Disable GT Channel LOC Constraints section.
04/04/2018 Version 4.4	
N/A	• All new “GT Locations” appendix. • Added Integrated Debug Options to “Debugging” appendix.
12/20/2017 Version 4.4	
N/A	• Removed documented support for Resizable BAR (RBAR). In Chapter 2, “Product Specification”: • Added details about phy_rdy_output signal in Clock and Reset Interface section. In Chapter 3, “Designing with the Core”: • Added note regarding configuration bank 65 in the Tandem Configuration section. • Updated the Legacy Interrupt Signaling figure and description.
10/04/2017 Version 4.4	
Chapter 2: Overview	Updated the Table: Available Integrated Block for PCI Express.
Chapter 3: Product Specification	• Updated the documented cfg_mgmt_addr width. • Minor updates to description for cfg_current_speed, cfg_max_payload, cfg_max_read_req, cfg_err_cor_out, cfg_err_nonfatal_out, cfg_err_fatal_out, cfg_local_error. • Major updates to description for cfg_interrupt_msi_function_number, cfg_ext_read_data_valid.

Section	Revision Summary
Chapter 4: Designing with the Core	• Updated the Tandem Configuration section. • Major updates to description for Completer Completion Descriptor Fields Bit Indexes 79:72, 87:80, and 88. • Major updates to description for Requester Request Descriptor Fields Bit Indexes 87:80, 95:88, and 120.
Chapter 5: Design Flow Steps	• Added further details to the following Vivado IP catalog option:, ◦ PF0 ID Initial Values > Device ID value (in Identity Setting (PF0 IDs and PF1 IDs) tab). ◦ 2MSIx Table Settings > Table Size (in MSI-X Capabilities tab).
Appendix A: Upgrading	Updated Port and Parameters changes for the current core version.
Appendix E: Using the Xilinx Virtual Cable to Debug	Added new section.
06/07/2017 Version 4.3	
N/A	• Minor updates to the Tandem Configuration section. • Updated port description for cfg_interrupt_msi_pending_status. • Updated MSI Mode figure.
04/05/2017Version 4.3	
N/A	In Chapter 3, “Designing with the Core”: • Added the GT Wizard option to the Shared Logic section. • Updated the Tandem Configuration section. In Chapter 4, “Design Flow Steps”: • Added PCIe DRP Ports and GT DRP Ports tables. • Added support for Gen1 and Gen2 speed links to the Enable In System IBERT option.
12/19/2016 Version 4.3	
Chapter 5: Design Flow Steps	Clarified that the Enable In System IBERT option should be used only for hardware debugging. Simulations are not supported for the cores generated using these options.
11/30/2016 Version 4.2	
N/A	Vivado IP catalog core option changes: • Added ECRC check capable sub option to the Enable AER Capability option in the Extended Capabilities 1 and Extended Capabilities 2 tab. • Clarified the effect of Enable Auto RxEq setting on the LPM or DFE setting in the GT Settings tab.
10/19/2016 Version 4.2	
Chapter 3: Product Specification	Editorial updates.

Section	Revision Summary
10/05/2016 Version 4.2	
N/A	• Moved the performance and resource utilization data to the web. • Updated the Tandem Configuration section. • Updated the BAR Size Ranges for Device Configuration table in the Design Flow Steps chapter. • Added the Enable Parity option to the Basic tab (Advanced Mode), the Enable Auto RxEq option to the GT Settings tab, and added the Enable In System IBERT, Enable Descrambler for Gen3 Mode, and Enable JTAG Debugger parameters in the new Add. Debug Options tab in the Design Flow Steps chapter. • Updated the Parameter Changes table and Ports Changes table in the Migrating and Updating appendix.
06/08/2016 Version 4.2	
N/A	• Updated Tandem Configuration section • Updated Generating Interrupt Requests • New device GT locations added.
04/06/2016 Version 4.2	
N/A	• In the Tandem Configuration section: ◦ Updated the Tandem PROM/PCIe Support Configurations table. ◦ Added the Tandem with Field Updates section. ◦ Added the Debugging Tandem with Field Updates Designs section. • Added new Parameters Changes and Ports Changes table to the Migrating and Updating appendix. • Updated the Kintex UltraScale Device GT Locations table.
01/29/2016 Version 4.1	
N/A	Updated Table- Available Integrated Blocks for PCI Express.
11/20/2015 Version 4.1	
N/A	Updated details for Gen3 in Table: Minimum Device Requirements.

Section	Revision Summary
11/18/2015 Version 4.1	
N/A	• \Added the FFVA1156 package to the XCKU095 device in the Available Integrated Blocks For PCI Express table (Chapter 2), and AMD Kintex™ UltraScale™ Device GT Locations table (Appendix B). • Updated the supported speed grades. • Updated the width for the following ports: cfg_function_status, cfg_vf_status, cfg_function_power_state, cfg_vf_power_state, cfg_rcb_status, cfg_dpa_substate_change, cfg_tph_requester_enable, cfg_tph_st_mode, cfg_vf_tph_requester_enable, cfg_vf_tph_st_mode, cfg_vf_flr_in_process, cfg_per_function_number, cfg_flr_done, cfg_flr_in_process, cfg_interrupt_pending, cfg_interrupt_msi_enable, cfg_interrupt_msi_vf_enable, and cfg_interrupt_msi_mmenable. • Added available negotiated link width values for cfg_negotiated_width. • Updated the status (Not Supported, Beta, Production) in the Tandem PROM/PCIe Supported Configurations table. • Removed the Message Signal Interrupt option from the Vivado IDE. • Added the Enable RX Message INTFC option to the Vivado IDE.
02/23/2015 Version 3.1	
N/A	• Updated the device selection and PCIe integrated block location information. • Updated the device core pinouts. • Clarified information regarding Pipe Mode Simulation. • Corrected the minimum 32-bit BARs number and the maximum 64-bit BARs number for the Base Address Register Overview parameter (per the Vivado IDE).
11/19/2014 Version 3.1	
N/A	• Updated the Configuration Space section with Media Configuration Access Port (MCAP) Extended Capability Structure. • Updated the tandem configuration information. • Added support for Cadence Incisive Enterprise Simulator (IES) and Synopsys Verilog Compiler Simulator (VCS). • Updated the device core pinouts.
10/01/2014 Version 3.1	
N/A	• Updated for core v3.1. • Updated the tandem configuration information. • Added package migration information for UltraScale device designs.
06/04/2014 Version 3.0	
N/A	Updated device information.

Section	Revision Summary
04/02/2014 Version 3.0	
N/A	• Updated block selection. • Updated core pinout information. • Updated shared logic information.
12/18/2013 Version 2.0	
N/A	Initial release.

Please Read: Important Legal Notices

Copyright