Versal ACAP Integrated Block for PCI Express v1.0

LogiCORE IP Product Guide

Vivado Design Suite

PG343 (v1.0) April 15, 2021

Table of Contents

Introduction

Introduction to the Core

The Xilinx® Versal ACAP Integrated Block for PCI Express® core is a high-bandwidth, scalable, and reliable serial interconnect solution for use with the Versal™ adaptive compute acceleration platform (ACAP). Based on the specific Versal ACAP used for a design, the programmable logic integrated block for PCI Express (PL PCIe) can represent a specific implementation of the PCI Express Base Specification 4.0 (PL PCIE4).

PL PCIE4 Features

- Designed to the PCI Express Base Specification 4.0, and Errata updates
- PCI Express Endpoint, Switch Port Upstream, Switch Port Downstream, Legacy Endpoint, and Root Port Modes
- x1, x2, x4, x8, or x16 link widths (x16 configuration supported only for Gen1, Gen2, and Gen3 speeds)
- Gen1, Gen2, Gen3, or Gen4 link speeds
- AXI4-Stream interface to customer logic
    - Configurable 64-bit/128-bit/256-bit/512-bit data path widths
    - Four Independent Initiator/Target, Request/Completion streams
- Parity protection on internal logic data paths and data interfaces
- Advanced Error Reporting (AER) and End-to-End CRC (ECRC)
- UltraRAM used for Transaction Layer Packet buffering
    - 32 KB – Replay Buffer
    - Configurable 16 KB, or 32 KB – Received Posted Transaction FIFO
    - Configurable 32 KB, or 64 KB – Received Completion Transaction FIFO
    - UltraRAM ECC protection enabled

- One Virtual Channel, eight Traffic Classes
- Supports multiple functions and single-root I/O virtualization (SR-IOV)
  - Up to four Physical Functions
  - Up to 252 Virtual Functions
- Built-in lane reversal and receiver lane to lane de-skew
- 3 x 64-bit, or 6 x 32-bit Base Address Registers (BARs) that are fully configurable
  - Expansion ROM BAR supported
- All Interrupt types are supported
  - INTx
  - 32 multi-vector MSI capability
  - MSI-X capability with up to 2048 vectors with optional to use, built-in vector tables
- Built-in Initiator Read Request/Completion Tag Manager
  - Up to 256 or 768 outstanding Initiator Read Request Transactions supported
- Advanced Peripheral Bus (APB3) interface is available to perform DRP operations.
- Features that enable high performance applications
  - AXI4-Stream TLP Straddle on Requester Completion Interface
  - Up to 1024 RX Completion Header Credits, and 64 KB RX Completion Payload Space
  - Relaxed Transaction Ordering in the Receive Data Path
  - Address Translation Services (ATS) Messaging
  - Atomic Operation Transactions Support
  - Transaction Tag Scaling as Completer
  - Flow Control Scaling
  - Low latency PIPE interface operation 32b at 500 MHz
- Several ease of use and configurability features are supported
  - BAR and ID based filtering of Received Transactions
  - ASPM Optionality
  - Configuration Extend Interface
  - AXI4-Stream Interfaces Address Align Mode
  - Debug and Diagnostics Interface

- ◦ Self Train over Loopback on PCI Express link

IP Facts

LogiCORE™ IP Facts Table	
Core Specifics	
Supported Device Family[1]	Versal™ ACAP
Supported User Interfaces	AXI4-Stream
Resources	Performance and Resource Use web page
Provided with Core	
Design Files	Verilog
Example Design	Verilog
Test Bench	Verilog
Constraints File	Xilinx Constraints file (XDC)
Simulation Model	Verilog
Supported S/W Driver	N/A
Tested Design Flows[2]	
Design Entry	Vivado® Design Suite
Simulation	For supported simulators, see the Xilinx Design Tools: Release Notes Guide.
Synthesis	Vivado synthesis
Support	
Release Notes and Known Issues	Master Answer Record: 73083
All Vivado IP Change Logs	Master Vivado IP Change Logs: 72775
Xilinx Support web page	

Notes:

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

Chapter 2

Overview

Navigating Content by Design Process

Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. All Versal™ ACAP design process Design Hubs can be found on the Xilinx.com website. This document covers the following design processes:

- **System and Solution Planning:** Identifying the components, performance, I/O, and data transfer requirements at a system level. Includes application mapping for the solution to PS, PL, and AI Engine. Topics in this document that apply to this design process include:
  - PL PCIE4 Features
  - AXI4-Stream Interface Description
- **Embedded Software Development:** Creating the software platform from the hardware platform and developing the application code using the embedded CPU. Also covers XRT and Graph APIs. Topics in this document that apply to this design process include:
  - Configuration Space
  - Configuration Management Interface
- **Host Software Development:** Developing the application code, accelerator development, including library, XRT, and Graph API use. Topics in this document that apply to this design process include:
  - Configuration Space
- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, subsystem functional simulation, and evaluating the Vivado® timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:
  - Chapter 4: Designing with the Core
  - Chapter 5: Design Flow Steps
  - Chapter 6: Example Design
  - Chapter 7: Test Bench

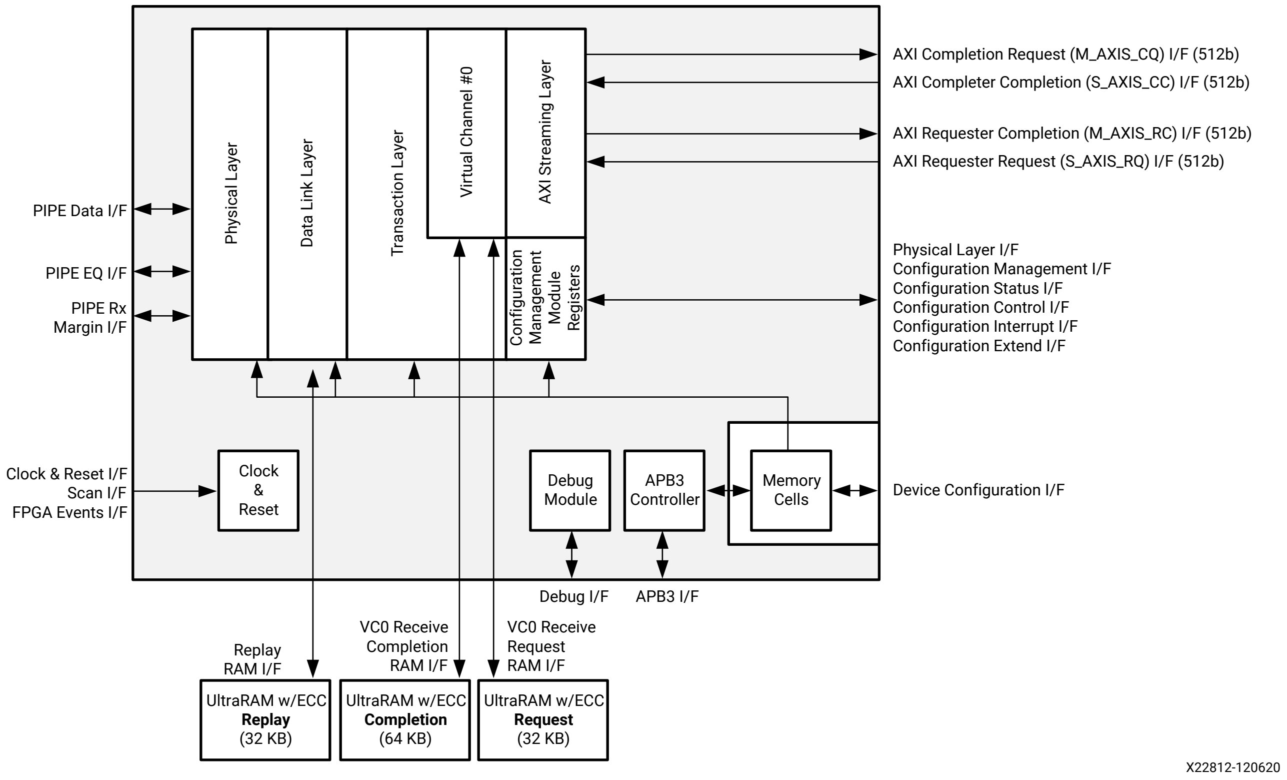
- Appendix B: GT Selection and Pin Planning
- Appendix C: PCIe Link Debug Enablement

Core Overview

The Versal™ ACAP Integrated Block for PCI Express® (PCIe®) core is a reliable, high-bandwidth, scalable serial interconnection for use with Versal devices. The core instantiates one of the available programmable logic integrated blocks for PCIe found in the Versal devices.

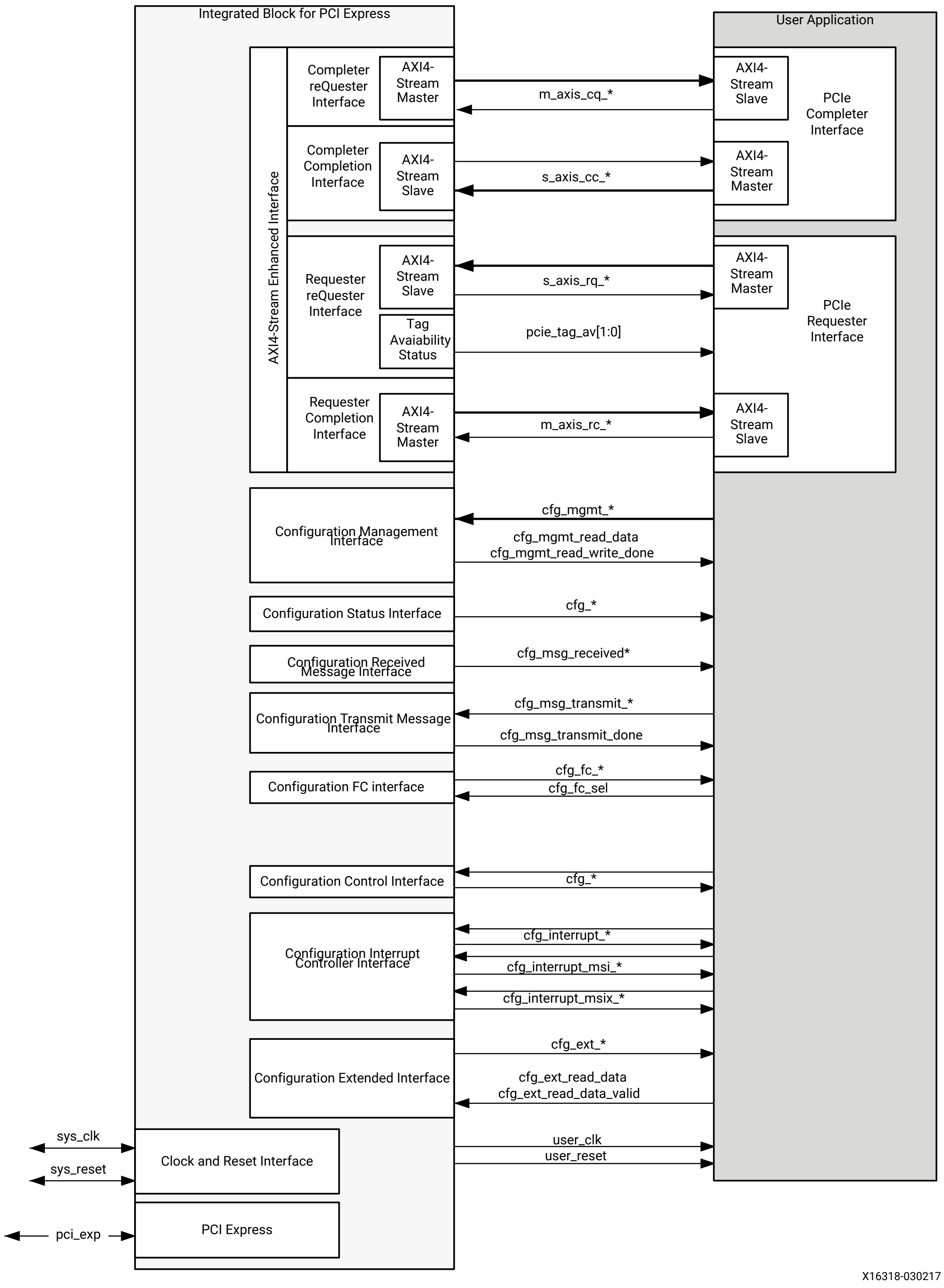
The following figure shows the block diagram of the core.

Figure 1: **Block Diagram for Programmable Logic Integrated Block for PCIe**


Physical Layer
Data Link Layer
Transaction Layer
Virtual Channel #0
AXI Streaming Layer
Configuration Management Module Registers
AXI Completion Request (M_AXIS_CQ) I/F (512b)
AXI Completer Completion (S_AXIS_CC) I/F (512b)
AXI Requester Completion (M_AXIS_RC) I/F (512b)
AXI Requester Request (S_AXIS_RQ) I/F (512b)
PIPE Data I/F
PIPE EQ I/F
PIPE Rx Margin I/F
Physical Layer I/F
Configuration Management I/F
Configuration Status I/F
Configuration Control I/F
Configuration Interrupt I/F
Configuration Extend I/F
Clock & Reset I/F
Scan I/F
FPGA Events I/F
Clock & Reset
Debug Module
APB3 Controller
Memory Cells
Device Configuration I/F
Debug I/F
APB3 I/F
Replay RAM I/F
VC0 Receive Completion RAM I/F
VC0 Receive Request RAM I/F
UltraRAM w/ECC Replay (32 KB)
UltraRAM w/ECC Completion (64 KB)
UltraRAM w/ECC Request (32 KB)
X22812-120620

The following figure shows the interfaces for the core.

Figure 2: **Core Interfaces**



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 Specification 4.0 has many optional features. The following features are not supported in the core:

- Resizable BAR extended capability
- ID-based TLP ordering
- TPH Capability
- Fast PCI Express Endpoint Enumeration using Tandem Configuration.

Related Information

Features Not Available, or Limited Usage Features

Limitations

Speed Change Related Issue

- **Description:** Repeated speed changes can result in the link not coming up to the intended targeted speed.
- **Workaround:** A follow-on attempt should bring the link back.

Link Autonomous Bandwidth Status (LABS) Bit

- **Description:** While performing the link width changes as a Root Complex, the link width change works as expected. However, the PCIe protocol requires a LABS bit which is not getting set after the link width change.

  Note: This is an informational bit and does not impact actual functionality.

- **Workaround:** None available.

Licensing and Ordering

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

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

Product Specification

Standards for the Integrated Block IP

Using PL PCIE4

This core adheres to the following standards:

- PCI Express Base Specification 4.0
- Legacy PCI Local Bus Specification 3.0

For more information about PCI/PCIe specifications, see *PCI-SIG Specifications* (https://www.pcisig.com/specifications).

Performance and Resource Use

For full details about performance and resource use, visit the Performance and Resource Use web page.

Port Descriptions for PL PCIE4

This section provides detailed port descriptions for the following interfaces:

- AXI4-Stream Core Interfaces
- Other Core Interfaces

AXI4-Stream Core Interfaces

64/128/256-Bit Interfaces

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

Related Information

512-bit Interfaces

Completer Request Interface

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

Table 1: **Completer Request Interface Port Descriptions**

Port	I/O	Width	Description
m_axis_cq_tdata	O	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	O	108	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. Table 2: Sideband Signal Descriptions in m_axis_cq_tuser describes the individual signals in this set.
m_axis_cq_tlast	O	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 1: **Completer Request Interface Port Descriptions** *(cont'd)*

Port	I/O	Width	Description
m_axis_cq_tkeep	O	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	O	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	I	1	CQ Data Ready. Activation of this signal by the user logic indicates to the core that the user application is ready to accept data. Data is transferred across the interface when both m_axis_cq_tvalid and m_axis_cq_tready are asserted in the same cycle. If the user application deasserts the ready signal when m_axis_cq_tvalid is High, the core maintains the data on the bus and keeps the valid signal asserted until the user application has asserted the ready signal.
pcie_cq_np_req	I	2	This input is used by the user application to request the delivery of a Non-Posted request. The core implements a credit-based flow control mechanism to control the delivery of Non-Posted requests across the interface, without blocking Posted TLPs. This input to the core controls an internal credit count. The credit count is updated in each clock cycle based on the setting of pcie_cq_np_req[1:0] as follows: • 00: No change • 01: Increment by 1 • 10 or 11: Reserved (bit [1] only applicable in 512-bit interface) The credit count is decremented on the delivery of each Non-Posted request across the interface. The core temporarily stops delivering Non-Posted requests to the user logic when the credit count is zero. It continues to deliver any Posted TLPs received from the link even when the delivery of Non-Posted requests has been paused. The user application can either set pcie_cq_np_req[1:0] in each cycle based on the status of its Non-Posted request receive buffer, or can set it to 11 permanently if it does not need to exercise selective backpressure on Non-Posted requests. The setting of pcie_cq_np_req[1:0] does not need to be aligned with the packet transfers on the completer request interface.

Table 1: **Completer Request Interface Port Descriptions** *(cont'd)*

Port	I/O	Width	Description
pcie_cq_np_req_count	O	6	This output provides the current value of the credit count maintained by the core for delivery of Non-Posted requests to the user logic. The core delivers a Non-Posted request across the completer request interface only when this credit count is non-zero. This counter saturates at a maximum limit of 32. Because of internal pipeline delays, there can be several cycles of delay between the user application providing credit on the pcie_cq_np_req[1:0] inputs and the PCIe core updating the pcie_cq_np_req_count output in response. This count resets on user_reset and de-assertion of user_lnk_up.

When PASID_CAP_ON is enabled then `m_axis_cq_tuser[107:85]` pins are specific to passing PASID field information. In all other cases those fields are reserved. The following table provides more information.

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

Bit Index	Name	Width	Description
3:0	first_be[3:0]	4	Byte enables for the first Dword of the payload. This field reflects the setting of the First_BE bits in the Transaction-Layer header of the TLP. For Memory Reads and I/O Reads, these four bits indicate the valid bytes to be read in the first Dword. For Memory Writes and I/O Writes, these bits indicate the valid bytes in the first Dword of the payload. For Atomic Operations and Messages with a payload, these bits are set to all 1s. This field is valid in the first beat of a packet, that is, when sop and m_axis_cq_tvalid are both High.
7:4	last_be[3:0]	4	Byte enables for the last Dword. This field reflects the setting of the Last_BE bits in the Transaction-Layer header of the TLP. For Memory Reads, these four bits indicate the valid bytes to be read in the last Dword of the block of data. For Memory Writes, these bits indicate the valid bytes in the ending Dword of the payload. For Atomic Operations and Messages with a payload, these bits are set to all 1s. 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.

Table 2: **Sideband Signal Descriptions in m_axis_cq_tuser** *(cont'd)*

Bit Index	Name	Width	Description
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).
84:53	parity	32	Bit *i* provides the odd parity computed for byte *i* of m_axis_cq_tdata. Only the lower 16 bits are used when the interface width is 128 bits, and only the lower 8 bits are used when the interface width is 64 bits. Bits [31:16] are set permanently to 0 by the core when the interface width is configured as 128 bits, and bits [31:8] are set permanently to 0 when the interface width is configured as 64 bits.
85	PASID TLP Valid	1	Indicates PASID TLP is valid.
105:86	PASID	20	Indicates PASID TLP prefix.
106	Execute Requested	1	Indicates Execute Requested to the user design.
107	Privileged Mode Requested	1	Indicates Privileged Mode Requested to the user design.

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 3: **Completer Completion Interface Port Descriptions**

Port	I/O	Width	Description
s_axis_cc_tdata	I	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	I	33	Completer Completion User Data. This set of signals contain sideband information for the TLP being transferred. These signals are valid when s_axis_cc_tvalid is High. The following tables describe the individual signals in this set.
s_axis_cc_tlast	I	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.
s_axis_cc_tkeep	I	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	I	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	O	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 4: **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.
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 5: **Requester Request Interface Port Descriptions**

Port	I/O	Width	Description
s_axis_rq_tdata	I	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.

Table 5: **Requester Request Interface Port Descriptions** *(cont'd)*

Port	I/O	Width	Description
s_axis_rq_tuser	I	62	Requester reQuest User Data. This set of signals contains sideband information for the TLP being transferred. These signals are valid when s_axis_rq_tvalid is High. The following tables describes the individual signals in this set.
s_axis_rq_tlast	I	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.
s_axis_rq_tkeep	I	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	I	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	O	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.

Table 5: **Requester Request Interface Port Descriptions** *(cont'd)*

Port	I/O	Width	Description
pcie_rq_seq_num0	O	6	Requester reQuest TLP transmit sequence number. You can optionally use this output to track the progress of the request in the core transmit pipeline. To use this feature, provide a sequence number for each request on the seq_num[3:0] bus. The core outputs this sequence number on the pcie_rq_seq_num0[3:0] output when the request TLP has reached a point in the pipeline where a Completion TLP from the user application cannot pass it. This mechanism enables you to maintain ordering between Completions sent to the CC interface of the core and Posted requests sent to the requester request interface. Data on the pcie_rq_seq_num0[3:0] output is valid when pcie_rq_seq_num_vld0 is High.
pcie_rq_seq_num_vld0	O	1	Requester reQuest TLP transmit sequence number valid. This output is asserted by the core for one cycle when it has placed valid data on pcie_rq_seq_num0[3:0].
pcie_rq_tag0 pcie_rq_tag1	O	10	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_vld0 is High. You must copy this tag and use it to associate the completion data with the pending request. There can be a delay of several cycles between the transfer of the request on the s_axis_rq_tdata bus and the assertion of pcie_rq_tag_vld0 by the core to provide the allocated tag for the request. Meanwhile, the user application can continue to send new requests. The tags for requests are communicated on this bus in FIFO order, so the user application can easily associate the tag value with the request it transferred.
pcie_rq_tag_vld0 pcie_rq_tag_vld1	O	1	Requester reQuest Non-Posted tag valid. The core asserts this output for one cycle when it has allocated a tag to an incoming Non-Posted request from the requester request interface and placed it on the pcie_rq_tag0 output.

When PASID_CAP_ON is enabled then `s_axis_rq_tuser[84:62]` pins are shared with `cfg*` ports. The following table provides more information.

Table 6: **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.
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).

Table 6: **Sideband Signal Descriptions in s_axis_rq_tuser** *(cont'd)*

Bit Index	Name	Width	Description
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 (*PCI-SIG Specifications* (https://www.pcisig.com/specifications)), an uncorrectable internal error is an error that occurs within a component that results in improper operation of the component. The only method of recovering from an uncorrectable internal error is a reset or hardware replacement. The parity bits can be permanently tied to 0 if parity check is not enabled in the core.
61:60	seq_num[5:4]	2	Extension of seq_num as in [27:24].
62	PASID TLP Valid	1	Indicates PASID TLP is valid.
82:63	PASID	20	Indicates PASID TLP prefix.
83	Execute Requested	1	Indicates Execute Requested.
84	Privileged Mode Requested	1	Indicates Privileged Mode Requested.

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 7: **Requester Completion Interface Port Descriptions**

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

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

Port	I/O	Width	Description
m_axis_rc_tuser	O	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	O	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	O	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	O	1	Requester Completion Data Valid. The core asserts this output whenever it is driving valid data on the m_axis_rc_tdata bus. The core keeps the valid signal asserted during the transfer of a packet. The user application can pace the data transfer using the m_axis_rc_tready signal.
m_axis_rc_tready	I	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 8: **Sideband Signal Descriptions in m_axis_rc_tuser**

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

Table 8: **Sideband Signal Descriptions in m_axis_rc_tuser** *(cont'd)*

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

512-bit Interfaces

This section provides the description for ports associated with the user interfaces of the core.

Related Information

64/128/256-Bit Interfaces

Completer Request Interface

Table 9: **Completer Request Interface Port Descriptions (512-bit Interface)**

Name	I/O	Width	Description
m_axis_cq_tdata	O	512	Transmit data from the PCIe completer request interface to the user application.
m_axis_cq_tuser	O	229	This is a set of signals containing sideband information for the TLP being transferred. These signals are valid when m_axis_cq_tvalid is High. The individual signals in this set are described in the following table.
m_axis_cq_tlast	O	1	The core asserts this signal in the last beat of a packet to indicate the end of the packet. When a TLP is transferred in a single beat, the core sets this bit in the first beat of the transfer. This output is used only when the straddle option is disabled. When the straddle option is enabled, the core sets this output permanently to 0.
m_axis_cq_tkeep	O	16	The assertion of bit *i* of this bus during a transfer indicates to the user logic that Dword *i* of the m_axis_cq_tdata bus contains valid data. The core sets this bit to 1 contiguously for all Dwords starting from the first Dword of the descriptor to the last Dword of the payload. Thus, m_axis_cq_tdata is set to all 1s in all beats of a packet, except in the final beat when the total size of the packet is not a multiple of the width of the data bus (both in Dwords). This is true for both Dword-aligned and 128b address-aligned modes of payload transfer. The tkeep bits are valid only when straddle is not enabled on the CQ interface. When straddle is enabled, the tkeep bits are permanently set to all 1s in all beats. The user logic must use the is_sop/is_eop signals in the m_axis_cq_tuser bus in that case to determine the start and end of TLPs transferred over the interface.
m_axis_cq_tvalid	O	1	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	I	1	Activation of this signal by the user logic indicates to the PCIe core that the user logic is ready to accept data. Data is transferred across the interface when both m_axis_cq_tvalid and m_axis_cq_tready are asserted in the same cycle. If the user logic deasserts the ready signal when m_axis_cq_tvalid is High, the core maintains the data on the bus and keeps the valid signal asserted until the user logic has asserted the ready signal.

Table 9: **Completer Request Interface Port Descriptions (512-bit Interface)** *(cont'd)*

Name	I/O	Width	Description
pcie_cq_np_req	I	2	This input is used by the user application to request the delivery of a Non-Posted request. The core implements a credit-based flow control mechanism to control the delivery of Non-Posted requests across the interface, without blocking Posted TLPs. This input to the core controls an internal credit count. The credit count is updated in each clock cycle based on the setting of pcie_cq_np_req[1:0] as follows: • 00: No change • 01: Increment by 1 • 10 or 11: Increment by 2 The credit count is decremented on the delivery of each Non-Posted request across the interface. The core temporarily stops delivering Non-Posted requests to the user logic when the credit count is zero. It continues to deliver any Posted TLPs received from the link even when the delivery of Non-Posted requests has been paused. The user application can either set pcie_cq_np_req[1:0] in each cycle based on the status of its Non-Posted request receive buffer, or can set it to 11 permanently if it does not need to exercise selective backpressure on Non-Posted requests. The setting of pcie_cq_np_req[1:0] does not need to be aligned with the packet transfers on the completer request interface.
pcie_cq_np_req_count	O	6	This output provides the current value of the credit count maintained by the core for delivery of Non-Posted requests to the user logic. The core delivers a Non-Posted request across the completer request interface only when this credit count is non-zero. This counter saturates at a maximum limit of 32. Because of internal pipeline delays, there can be several cycles of delay between the user application providing credit on the pcie_cq_np_req[1:0] inputs and the PCIe core updating the pcie_cq_np_req_count output in response. This count resets on user_reset and de-assertion of user_lnk_up.

When PASID_CAP_ON is enabled then `m_axis_cq_tuser[228:183]` pins are specific to passing PASID field information. In all other cases those fields are reserved. The following table provides more information.

Table 10: **Sideband Signals in m_axis_cq_tuser (512-bit Interface)**

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

Table 10: **Sideband Signals in m_axis_cq_tuser (512-bit Interface)** *(cont'd)*

Bit Index	Name	Width	Description
81:80	is_sop[1:0]	2	Signals the start of a new TLP in this beat. These outputs are set in the first beat of a TLP. When straddle is disabled, only is_sop[0] is valid and is_sop[1] is permanently set to 0. When straddle is enabled, the settings are as follows: • 00: No new TLP starting in this beat. • 01: A single new TLP starts in this beat. Its start position is indicated by is_sop0_ptr[1:0]. • 11: Two new TLPs are starting in this beat. is_sop0_ptr[1:0] provides the start position of the first TLP and is_sop1_ptr[1:0] provides the start position of the second TLP. • 10: Reserved. Use of this signal is optional for the user logic when the straddle option is disabled, because a new TLP always starts in the beat following tlast assertion.
83:82	is_sop0_ptr[1:0]	2	Indicates the position of the first byte of the first TLP starting in this beat: • 00: Byte lane 0 • 10: Byte lane 32 • 01, 11: Reserved This field is valid only when the straddle option is enabled on the CQ interface. Otherwise, it is set to 0 permanently, as a TLP can only start in bye lane 0.
85:84	is_sop1_ptr[1:0]	2	Indicates the position of the first byte of the second TLP starting in this beat: • 10: Byte lane 32 • 00, 01, 11: Reserved. This output is used only when the straddle option is enabled on the CQ interface. The core can then straddle two TLPs in the same beat. The output is permanently set to 0 when straddle is disabled.
87:86	is_eop[1:0]	2	Indicates that a TLP is ending in this beat. These outputs are set in the final beat of a TLP. When straddle is disabled, only is_eop[0] is valid and is_eop[1] is permanently set to 0. When straddle is enabled, the settings are as follows: • 00: No TLPs ending in this beat. • 01: A single TLP is ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of this TLP. • 11: Two TLPs are ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of the first TLP and is_eop1_ptr[3:0]provides the offset of the last Dword of the second TLP. • 10: Reserved. The use of this signal is optional for the user logic when the straddle option is not enabled, because tlast Is asserted in the final beat of a TLP.
91:88	is_eop0_ptr[3:0]	4	Offset of the last Dword of the first TLP ending in this beat. This output is valid when is_eop[0] is asserted.
95:92	is_eop1_ptr[3:0]	4	Offset of the last Dword of the second TLP ending in this beat. This output is valid when is_eop[1] is asserted. The output is permanently set to 0 when straddle is disabled.

Table 10: **Sideband Signals in m_axis_cq_tuser (512-bit Interface)** *(cont'd)*

Bit Index	Name	Width	Description
96	discontinue	1	This signal is asserted by the core in the last beat of a TLP, if it has detected an uncorrectable error while reading the TLP payload from its internal FIFO memory. The user application must discard the entire TLP when such an error is signaled by the core. This signal is never asserted when the TLP has no payload. It is asserted only in the last beat of the payload transfer, that is when is_eop[0] is High. When the straddle option is enabled, the core does not start a second TLP if it has asserted discontinue in a beat. When the core is configured as an Endpoint, the error is also reported by the core to the Root Complex it is attached to, using Advanced Error Reporting (AER).
182:119	parity	64	Odd parity for the 512-bit transmit data. Bit *i* provides the odd parity computed for byte *i* of m_axis_cq_tdata.
183	PASID TLP Valid 0	1	Indicates PASID TLP 0 is valid.
184	PASID TLP Valid 1	1	Indicates PASID TLP 1 is valid.
204:185	PASID 0	20	Indicates PASID TLP Prefix for packet0 to the user design.
224:205	PASID 1	20	Indicates PASID TLP Prefix for packet1 to the user design.
225	Execute Requested 0	1	Indicates Execute Requested for packet0
226	Execute Requested 1	1	Indicates Execute Requested for packet1
227	Privileged Mode Requested 0	1	Indicates Privileged Mode Requested for packet0 to the user design.
228	Privileged Mode Requested 1	1	Indicates Privileged Mode Requested for packet 1 to the user design.

Completer Completion Interface

Table 11: **Completer Completion Interface Port Descriptions (512-bit Interface)**

Name	I/O	Width	Description
s_axis_cc_tdata	I	512	Completion data from the user application to the PCIe core.
s_axis_cc_tuser	I	183	This is a set of signals containing sideband information for the TLP being transferred. These signals are valid when s_axis_cc_tvalid is High. The individual signals in this set are described in the following table.
s_axis_cc_tlast	I	1	The user application must assert this signal in the last cycle of a packet to indicate the end of the packet. When the TLP is transferred in a single beat, the user application must set this bit in the first cycle of the transfer. This input is used by the core only when the straddle option is disabled. When the straddle option is enabled, the core ignores the setting of this input, using instead the is_sop/ is_eop signals in the s_axis_cc_tuser bus to determine the start and end of TLPs.

Table 11: **Completer Completion Interface Port Descriptions (512-bit Interface)** *(cont'd)*

Name	I/O	Width	Description
s_axis_cc_tkeep	I	16	The assertion of bit *i* of this bus during a transfer indicates to the core that Dword *i* of the s_axis_cc_tdata bus contains valid data. The user logic must set this bit to 1 contiguously for all Dwords starting from the first Dword of the descriptor to the last Dword of the payload. Thus, s_axis_cc_tdata must be set to all 1s in all beats of a packet, except in the final beat when the total size of the packet is not a multiple of the width of the data bus (both in Dwords). This is true for both Dword-aligned and 128b address-aligned modes of payload transfer. The tkeep bits are valid only when straddle is not enabled on the CC interface. When straddle is enabled, the core ignores the setting of these bits when receiving data across the interface. The user logic must set the is_sop/is_eop signals in the s_axis_cc_tuser bus in that case to signal the start and end of TLPs transferred over the interface.
s_axis_cc_tvalid	I	1	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	O	4	Activation of this signal by the PCIe core indicates that it is ready to accept data. Data is transferred across the interface when both s_axis_cc_tvalid and s_axis_cc_tready are asserted in the same cycle. If the core deasserts the ready signal when the valid signal is High, the user logic must maintain the data on the bus and keep the valid signal asserted until the core has asserted the ready signal. With this output port, each bit indicates the same value, so the user logic can use any of the bit.

Table 12: **Sideband Signals in s_axis_cc_tuser**

Bit Index	Name	Width	Description
1:0	is_sop[1:0]	2	Signals the start of a new TLP in this beat. These outputs are set in the first beat of a TLP. When straddle is disabled, only is_sop[0] is valid. When straddle is enabled, the settings are as follows: • 00: No new TLP starting in this beat. • 01: A single new TLP starts in this beat. Its start position is indicated by is_sop0_ptr[1:0]. • 11: Two new TLPs are starting in this beat. is_sop0_ptr[1:0] provides the start position of the first TLP and is_sop1_ptr[1:0] provides the start position of the second TLP. • 10: Reserved. This field is used by the core only when the straddle option is enabled. When straddle is disabled, the core uses tlast to determine the first beat of an incoming TLP.

Table 12: **Sideband Signals in s_axis_cc_tuser** *(cont'd)*

Bit Index	Name	Width	Description
3:2	is_sop0_ptr[1:0]	2	Indicates the position of the first byte of the first TLP starting in this beat: • 00: Byte lane 0 • 10: Byte lane 32 • 01, 11: Reserved This field is used by the core only when the straddle option is enabled. When straddle is disabled, the user logic must always start a TLP in byte lane 0.
5:4	is_sop1_ptr[1:0]	2	Indicates the position of the first byte of the second TLP starting in this beat: • 10: Byte lane 32 • 00, 01, 11: Reserved. This input is used only when the straddle option is enabled on the CC interface. The user can then straddle two TLPs in the same beat.
7:6	is_eop[1:0]	2	Signals that a TLP is ending in this beat. These outputs are set in the final beat of a TLP. When straddle is disabled, only is_eop[0] is valid. When straddle is enabled, the settings are as follows: • 00: No TLPs ending in this beat. • 01: A single TLP is ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of this TLP. • 11: Two TLPs are ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of the first TLP and is_eop1_ptr[3:0] provides the offset of the last Dword of the second TLP. • 10: Reserved. This field is used by the core only when the straddle option is enabled. When straddle is disabled, the core uses tlast and tkeep to determine the ending beat and position of EOP.
11:8	is_eop0_ptr[3:0]	4	Offset of the last Dword of the first TLP ending in this beat. This output is valid when is_eop[0] is asserted. This field is used by the core only when the straddle option is enabled.
15:12	is_eop1_ptr[3:0]	4	Offset of the last Dword of the second TLP ending in this beat. This output is valid when is_eop[1] is asserted. This field is used by the core only when the straddle option is enabled.

Table 12: **Sideband Signals in s_axis_cc_tuser** *(cont'd)*

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

Requester Request Interface

Table 13: **Requester Request Interface Port Descriptions (512-bit Interface)**

Name	I/O	Width	Description
s_axis_rq_tdata	I	512	Requester-side request data from the user application to the PCIe core.
s_axis_rq_tuser	I	137	This is a set of signals containing sideband information for the TLP being transferred. These signals are valid when s_axis_rq_tvalid is High. The individual signals in this set are described in the following table.
s_axis_rq_tlast	I	1	The user application must assert this signal in the last cycle of a TLP to indicate the end of the packet. When the TLP is transferred in a single beat, the user logic must set this bit in the first cycle of the transfer. This input is used by the core only when the straddle option is disabled. When the straddle option is enabled, the core ignores the setting of this input, using instead the is_sop/ is_eop signals in the s_axis_rq_tuser bus to determine the start and end of TLPs.

Table 13: **Requester Request Interface Port Descriptions (512-bit Interface)** *(cont'd)*

Name	I/O	Width	Description
s_axis_rq_tkeep	I	16	The assertion of bit *i* of this bus during a transfer indicates to the core that Dword *i* of the s_axis_rq_tdata bus contains valid data. The user logic must set this bit to 1 contiguously for all Dwords starting from the first Dword of the descriptor to the last Dword of the payload. Thus, s_axis_rq_tdata must be set to all 1s in all beats of a packet, except in the final beat when the total size of the packet is not a multiple of the width of the data bus (both in Dwords). This is true for both Dword-aligned and 128b address-aligned modes of payload transfer. The tkeep bits are valid only when straddle is not enabled on the RQ interface. When straddle is enabled, the core ignores the setting of these bits when receiving data across the interface. The user logic must set the is_sop/is_eop signals in the s_axis_rq_tuser bus in that case to signal the start and end of TLPs transferred over the interface.
s_axis_rq_tvalid	I	1	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	O	4	Activation of this signal by the PCIe core indicates that it is ready to accept data. Data is transferred across the interface when both s_axis_rq_tvalid and s_axis_rq_tready are asserted in the same cycle. If the core deasserts the ready signal when the valid signal is High, the user logic must maintain the data on the bus and keep the valid signal asserted until the core has asserted the ready signal. With this output port, each bit indicates the same value, so the user logic can use any of the bit.
pcie_rq_tag_vld0	O	1	The core asserts this output for one cycle when it has allocated a tag to an incoming Non-Posted request from the requester request interface and placed it on the pcie_rq_tag0 output. The bit is encoded as follows: • 0: No tags being provided in this cycle. • 1: A tag is presented on pcie_rq_tag0.
pcie_rq_tag_vld1	O	1	The core asserts this output for one cycle when it has allocated a tag to an incoming Non-Posted request from the requester request interface and placed it on the pcie_rq_tag1 output. The bit is encoded as follows: • 0: No tag is provided on pcie_rq_tag1 in this cycle. • 1: A tag is presented on pcie_rq_tag1.

Table 13: **Requester Request Interface Port Descriptions (512-bit Interface)** *(cont'd)*

Name	I/O	Width	Description
pcie_rq_tag0	O	10	When tag management for Non-Posted requests is performed by the core (Enable Client Tag is unchecked in the IP customization GUI), this output is used by the core to communicate the allocated tag for each Non-Posted request received from the client. The tag value on pcie_rq_tag0 is valid for one cycle when pcie_rq_tag_vld0 is High. The client must copy this tag and use it to associate the completion data with the pending request. There can be a delay of several cycles between the transfer of the request on the s_axis_rq_tdata bus and the assertion of pcie_rq_tag_vld0 by the core to provide the allocated tag for the request. The client can, meanwhile, continue to send new requests. The tags for requests are communicated on this bus in FIFO order. Therefore, the user application must associate the allocated tags with the requests in the order in which the requests were transferred over the interface. When pcie_rq_tag0 and pcie_rq_tag1 are both valid in the same cycle, the value on pcie_rq_tag0 corresponds to the earlier of the two requests transferred over the interface.
pcie_rq_tag1	O	10	The description of this signal is the same as pcie_rq_tag0, except the tag value on pcie_rq_tag1 is valid for one cycle when pcie_rq_tag_vld1 is asserted.
pcie_rq_seq_num0	O	6	The user may optionally use this output to keep track of the progress of the request in the core's transmit pipeline. To use this feature, the user application must provide a sequence number for each request on the s_axis_rq_seq_num0[5:0] bus. The core outputs this sequence number on the pcie_rq_seq_num0[5:0] output when the request TLP has progressed to a point in the pipeline where a Completion TLP from the client will not be able to pass it. This mechanism enables the client to maintain ordering between Completions sent to the completer completion interface of the core and Posted requests sent to the requester request interface. Data on the pcie_rq_seq_num0[5:0] output is valid when pcie_rq_seq_num_ vld0 is High.
pcie_rq_seq_num1	O	6	This output is identical in function to that of pcie_rq_seq_num0. It is used to provide a second sequence number in the same cycle when a first sequence number is being presented on pcie_rq_seq_num0. Data on the pcie_rq_seq_num1[5:0] output is valid when pcie_rq_seq_num_ vld1 is High.
pcie_rq_seq_num_vld0	O	1	This output is asserted by the core for one cycle when it has placed valid data on pcie_rq_seq_num0[5:0].
pcie_rq_seq_num_vld1	O	1	This output is asserted by the core for one cycle when it has placed valid data on pcie_rq_seq_num1[5:0].

When PASID_CAP_ON is enabled then s_axis_rq_tuser [182:137] pins are shared with cfg* ports. Look in the table below for more info.

Table 14: **Sideband Signals in s_axis_rq_tuser (512-bit Interface)**

Bit Index	Name	Width	Description
7:0	first_be[7:0]	8	Byte enables for the first Dword. This field must be set based on the desired value of the First_BE bits in the Transaction-Layer header of the request TLP. first_be[3:0] corresponds to the byte enables for the first TLP starting in this beat, and first_be[7:4] corresponds to the byte enables for the second TLP starting in this beat (if present). For Memory Reads, I/O Reads and Configuration Reads, these 4 bits indicate the valid bytes to be read in the first Dword. For Memory Writes, I/O Writes and Configuration Writes, these bits indicate the valid bytes in the first Dword of the payload. The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High.
15:8	last_be[7:0]	8	Byte enables for the last Dword. This field must be set based on the desired value of the Last_BE bits in the Transaction-Layer header of the TLP. last_be[3:0] corresponds to the byte enables for the first TLP starting in this beat, and last_be[7:4] corresponds to the byte enables for the second TLP starting in this beat (if present). For Memory Reads and Writes of one DW transfers and zero length transfers, these bits should be 0s. For Memory Reads of 2 Dwords or more, these 4 bits indicate the valid bytes to be read in the last Dword of the block of data. For Memory Writes of 2 Dwords or more, these bits indicate the valid bytes in the last Dword of the payload. The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High.
19:16	addr_offset[3:0]	4	When 128b the address-aligned mode is in use on this interface, the user application must provide the offset where the payload data begins (in multiples of 4 bytes) on the data bus on this sideband bus. This enables the core to determine the alignment of the data block being transferred. addr_offset[1:0] corresponds to the offset for the first TLP starting in this beat, and addr_offset[3:2] is reserved for future use. The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High. When the requester request interface is configured in the Dword-alignment mode, these bits must always be set to 0.
21:20	is_sop[1:0]	2	Signals the start of a new TLP in this beat. These outputs are set in the first beat of a TLP. When straddle is disabled, only is_sop[0] is valid. When straddle is enabled, the settings are as follows: • 00: No new TLP starting in this beat. • 01: A single new TLP starts in this beat. Its start position is indicated by is_sop0_ptr[1:0]. • 11: Two new TLPs are starting in this beat. is_sop0_ptr[1:0] provides the start position of the first TLP and is_sop1_ptr[1:0] provides the start position of the second TLP. • 10: Reserved. Use of this signal is optional for the user logic when the straddle option is not enabled, because a new TLP always starts in the beat following tlast assertion.

Table 14: **Sideband Signals in s_axis_rq_tuser (512-bit Interface)** *(cont'd)*

Bit Index	Name	Width	Description
23:22	is_sop0_ptr[1:0]	2	Indicates the position of the first byte of the first TLP starting in this beat: • 00: Byte lane 0 • 10: Byte lane 32 • 01, 11: Reserved
25:24	is_sop1_ptr[1:0]	2	Indicates the position of the first byte of the second TLP starting in this beat: • 10: Byte lane 32 • 00, 01, 11: Reserved. This output is used only when the straddle option is enabled on the interface.
27:26	is_eop[1:0]	2	Signals that a TLP is ending in this beat. These outputs are set in the final beat of a TLP. When straddle is disabled, only is_eop[0] is valid. When straddle is enabled, the settings are as follows: • 00: No TLPs ending in this beat. • 01: A single TLP is ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of this TLP. • 11: Two TLPs are ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of the first TLP and is_eop1_ptr[3:0] provides the offset of the last Dword of the second TLP. • 10: Reserved. Use of this signal is optional for the user logic when the straddle option is not enabled, because tlast Is asserted in the final beat of a TLP.
31:28	is_eop0_ptr[3:0]	4	Offset of the last Dword of the first TLP ending in this beat. This output is valid when is_eop[0] is asserted.
35:32	is_eop1_ptr[3:0]	4	Offset of the last Dword of the second TLP ending in this beat. This output is valid when is_eop[1] is asserted.
36	discontinue	1	This signal can be asserted by the user application during a transfer if it has detected an error in the data being transferred and needs to abort the packet. The core nullifies the corresponding TLP on the link to avoid data corruption. The user logic can assert this signal in any beat of a TLP except the first beat during its transfer. It can either choose to terminate the packet prematurely in the cycle where the error was signaled, or continue until all bytes of the payload are delivered to the core. In the latter case, the core treats the error as sticky for the following beats of the packet, even if the user logic deasserts the discontinue signal before the end of the packet. The discontinue signal can be asserted only when s_axis_rq_tvalid is High. The core samples this signal only when s_axis_rq_tready is High. Thus, once asserted, it should not be deasserted until s_axis_rq_tready is High. When the straddle option is enabled on the RQ interface, the user should not start a new TLP in the same beat when a TLP is ending with discontinue asserted. When the core is configured as an Endpoint, this error is also reported by the core to the Root Complex it is attached to, using Advanced Error Reporting (AER).
60:37	reserved		TPH ports are reserved.

Table 14: **Sideband Signals in s_axis_rq_tuser (512-bit Interface)** *(cont'd)*

Bit Index	Name	Width	Description
66:61	seq_num0[5:0]	6	The user logic can optionally supply a 6-bit sequence number in this field to keep track of the progress of the request in the core's transmit pipeline. The core outputs this sequence number on its pcie_rq_seq_num0 or pcie_rq_seq_num1 output when the request TLP has progressed to a point in the pipeline where a Completion TLP from the user logic is not able to pass it. The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High. This input can be hardwired to 0 when the user logic is not monitoring the pcie_rq_seq_num* outputs of the core.
72:67	seq_num1[5:0]	6	If there is a second TLP starting in the same beat, the user logic can optionally provide a 6-bit sequence number for this TLP on this input. This sequence number is used in the same manner as seq_num0. The core samples this field in the first beat of a packet, when s_axis_rq_tvalid and s_axis_rq_tready are both High. This input can be hardwired to 0 when the user logic is not monitoring the pcie_rq_seq_num* outputs of the core.
136:73	parity	64	Odd parity for the 512-bit data. When parity checking is enabled in the core, user logic must set bit *i* of this bus to the odd parity computed for byte *i* of s_axis_rq_tdata. On detection of a parity error, the core nullifies the corresponding TLP on the link and reports it as an Uncorrectable Internal Error. These bits can be set to 0 if parity checking is disabled in the core.
137	PASID TLP Valid 0	1	Indicates PASID TLP is valid for packet 0. pcie_posted_req_delivered is repurposed to pass PASID TLP VALID0 information from the user design.
138	PASID TLP Valid 1	1	Indicates PASID TLP is valid for packet 1. pcie_cq_pipeline_empty is repurposed to pass PASID TLP VALID1 information from the user design.
158:139	PASID 0	20	Indicates PASID TLP Prefix for packet 0. Repurpose the following signals to pass PASID 0 information from the user design. [139]: pcie_cq_np_user_credit_rcvd. [141:140]: pcie_compl_delivered. [149:142]: pcie_compl_delivered_tag0. [157:150]: pcie_compl_delivered_tag1. user_spare_in[0]: tl_rx_posted_credit_released.
178:159	PASID 1	20	Indicates PASID TLP Prefix for packet 1. Repurpose the following list of signals to pass PASID 1 information from the user design. user_spare_in[20:1] indexed as below: [165:159]: tl_rx_posted_payload_credit_released_value. [166]: tl_rx_nonposted_credit_released. [168:167]: tl_rx_nonposted_payload_credit_released_value. [169]: tl_rx_compl_credit_released. [171:170]: tl_rx_compl_header_credit_released_value. [178:172]: tl_rx_compl_payload_credit_released_value.
179	Execute Requested 0	1	Indicates Execute Requested for packet 0.

Table 14: **Sideband Signals in s_axis_rq_tuser (512-bit Interface)** *(cont'd)*

Bit Index	Name	Width	Description
180	Execute Requested 1	1	Indicates Execute Requested for packet 1.
181	Privileged Mode Requested 0	1	Indicates Privileged Mode Requested for packet 0.
182	Privileged Mode Requested 1	1	Indicates Privileged Mode Requested for packet 1.

Requester Completion Interface

Table 15: **Requester Completion Interface Port Descriptions (512-bit Interface)**

Name	I/O	Width	Description
m_axis_rc_tdata	O	512	Transmit data from the PCIe requester completion interface to the user application.
m_axis_rc_tuser	O	161	This is a set of signals containing sideband information for the TLP being transferred. These signals are valid when m_axis_rc_tvalid is High. The individual signals in this set are described in the following table.
m_axis_rc_tlast	O	1	The core asserts this signal in the last beat of a packet to indicate the end of the packet. When a TLP is transferred in a single beat, the core sets this bit in the first beat of the transfer. This output is used only when the straddle option is disabled. When the straddle option is enabled, the core sets this output permanently to 0.
m_axis_rc_tkeep	O	16	The assertion of bit *i* of this bus during a transfer indicates to the user logic that Dword *i* of the m_axis_rc_tdata bus contains valid data. The core sets this bit to 1 contiguously for all Dwords starting from the first Dword of the descriptor to the last Dword of the payload. Thus, m_axis_rc_tkeep is set to all 1s in all beats of a packet, except in the final beat when the total size of the packet is not a multiple of the width of the data bus (both in Dwords). This is true for both Dword-aligned and address-aligned modes of payload transfer. These outputs are permanently set to all 1s when the straddle option is enabled. The user logic must use the signals in m_axis_rc_tuser in that case to determine the start and end of Completion TLPs transferred over the interface.
m_axis_rc_tvalid	O	1	The core asserts this output whenever it is driving valid data on the m_axis_rc_tdata bus. The core keeps the valid signal asserted during the transfer of a packet. The user application can pace the data transfer using the m_axis_rc_tready signal.
m_axis_rc_tready	I	1	Activation of this signal by the user logic indicates to the PCIe core that the user logic is ready to accept data. Data is transferred across the interface when both m_axis_rc_tvalid and m_axis_rc_tready are asserted in the same cycle. If the user logic deasserts the ready signal when the valid signal is High, the core maintains the data on the bus and keep the valid signal asserted until the user logic has asserted the ready signal.

Table 16: **Sideband Signals in m_axis_rc_tuser (512-bit Interface)**

Bit Index	Name	Width	Description
63:0	byte_en	64	The client logic may optionally use these byte enable bits to determine the valid bytes in the payload of a packet being transferred. The assertion of bit *i* of this bus during a transfer indicates to the client that byte *i* of the m_axis_cq_tdatabuscontains a valid payload byte. This bit is not asserted for descriptor bytes. Although the byte enables can be generated by client logic from information in the request descriptor (address and length), the client has the option of using these signals directly instead of generating them from other interface signals. The 1 bits in this bus for the payload of a TLP are always contiguous.
67:64	is_sop[3:0]	4	Signals the start of a new TLP in this beat. These outputs are set in the first beat of a TLP. When straddle is disabled, only is_sop[0] is valid and is_sop[3:1] are permanently set to 0. When straddle is enabled, the settings are as follows: • 0000: No new TLP starting in this beat. • 0001: A single new TLP starts in this beat. ts start position is indicated by is_sop0_ptr[1:0]. • 0011: Two new TLPs are starting in this beat. is_sop0_ptr[1:0] provides the start position of the first TLP and is_sop1_ptr[1:0] provides the start position of the second TLP. • 0111: Three new TLPs are starting in this beat. is_sop0_ptr[1:0] provides the start position of the first TLP, is_sop1_ptr[1:0] provides the start position of the second TLP, and is_sop2_ptr[1:0] provides the start position of the third TLP. • 1111: Four new TLPs are starting in this beat. is_sop0_ptr[1:0] provides the start position of the first TLP, is_sop1_ptr[1:0] provides the start position of the second TLP, is_sop2_ptr[1:0] provides the start position of the third TLP, and is_sop3_ptr[1:0] provides the start position of the fourth TLP. • All other settings are reserved. Use of this signal is optional for the client when the straddle option is not enabled, because a new TLP always starts in the beat following m_axis_rc_tlast assertion.
69:68	is_sop0_ptr[1:0]	2	Indicates the position of the first byte of the first TLP starting in this beat: • 00: Byte lane 0 • 01: Byte lane 16 • 10: Byte lane 32 • 11: Byte lane 48 This field is valid only when the straddle option is enabled on the RC interface. Otherwise, it is set to 0 permanently, as a TLP can only start in bye lane 0.

Table 16: **Sideband Signals in m_axis_rc_tuser (512-bit Interface)** *(cont'd)*

Bit Index	Name	Width	Description
71:70	is_sop1_ptr[1:0]	2	Indicates the position of the first byte of the second TLP starting in this beat: • 00: Reserved • 01: Byte lane 16 • 10: Byte lane 32 • 11: Byte lane 48 This output is used only when the straddle option is enabled on the RC interface. The output is permanently set to 0 when straddle is disabled.
73:72	is_sop2_ptr[1:0]	2	Indicates the position of the first byte of the third TLP starting in this beat: • 00: Reserved • 01: Reserved • 10: Byte lane 32 • 11: Byte lane 48 This output is used only when the straddle option is enabled on the RC interface. The output is permanently set to 0 when straddle is disabled.
75:74	is_sop3_ptr[1:0]	2	Indicates the position of the first byte of the fourth TLP starting in this beat: • 00, 01, 10: Reserved • 11: Byte lane 48 This output is used only when the straddle option is enabled on the RC interface. The output is permanently set to 0 when straddle is disabled.
79:76	is_eop[3:0]	4	Signals that one or more TLPs are ending in this beat only when straddle is enabled. These outputs are set in the final beat of a TLP. The settings are as follows: • 0000: No TLPs ending in this beat. • 0001: A single TLP is ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of this TLP. • 0011: Two TLPs are ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of the first TLP and is_eop1_ptr[3:0] provides the offset of the last Dword of the second TLP. • 0111: Three TLPs are ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of the first TLP, is_eop1_ptr[3:0] provides the offset of the last Dword of the second TLP, and is_eop2_ptr[3:0] provides the offset of the last Dword of the third TLP. • 1111: Four TLPs are ending in this beat. is_eop0_ptr[3:0] provides the offset of the last Dword of the first TLP, is_eop1_ptr[3:0] provides the offset of the last Dword of the second TLP, is_eop2_ptr[3:0] provides the offset of the last Dword of the third TLP, and is_eop3_ptr[3:0] provides the offset of the last Dword of the fourth TLP. • All other settings are reserved. When the straddle option is disabled, m_axis_rc_tlast indicates the final beat of a TLP.

Table 16: **Sideband Signals in m_axis_rc_tuser (512-bit Interface)** *(cont'd)*

Bit Index	Name	Width	Description
83:80	is_eop0_ptr[3:0]	4	Offset of the last Dword of the first TLP ending in this beat. This output is valid when is_eop[0] is asserted. This output is used only when the straddle option is enabled on the RC interface. The output is permanently set to 0 when straddle is disabled.
87:84	is_eop1_ptr[3:0]	4	Offset of the last Dword of the second TLP ending in this beat. This output is valid when is_eop[1] is asserted. This output is used only when the straddle option is enabled on the RC interface. The output is permanently set to 0 when straddle is disabled.
91:88	is_eop2_ptr[3:0]	4	Offset of the last Dword of the third TLP ending in this beat. This output is valid when is_eop[2] is asserted. This output is used only when the straddle option is enabled on the RC interface. The output is permanently set to 0 when straddle is disabled.
95:92	is_eop3_ptr[3:0]	4	Offset of the last Dword of the fourth TLP ending in this beat. This output is valid when is_eop[3] is asserted. This output is used only when the straddle option is enabled on the RC interface. The output is permanently set to 0 when straddle is disabled.
96	discontinue	1	This signal is asserted by the core in the last beat of a TLP, if it has detected an uncorrectable error while reading the TLP payload from its internal FIFO memory. The client application must discard the entire TLP when such an error is signaled by the core. This signal is never asserted when the TLP has no payload. It is asserted only in the last beat of the payload transfer, that is when is_eop[0] is High. When the straddle option is enabled, the core does not start a new TLP if it has asserted discontinue in a beat. When the core is configured as an Endpoint, the error is also reported by the core to the Root Complex it is attached to, using Advanced Error Reporting (AER).
160:97	parity	64	Odd parity for the 512-bit transmit data. Bit *i* provides the odd parity computed for byte *i* of m_axis_cq_tdata.

Other Core Interfaces

The core also provides the interfaces described in this section.

Power Management Interface

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

Table 17: **Power Management Interface Ports**

Port	I/O	Width	Description
cfg_pm_aspm_l1_entry_reject	I	1	Configuration Power Management ASPM L1 Entry Reject: When driven to 1b, Downstream Port rejects transition requests to L1 state.

Table 17: **Power Management Interface Ports** *(cont'd)*

Port	I/O	Width	Description
cfg_pm_aspm_tx_l0s_entry_disable	I	1	Configuration Power Management ASPM L0s Entry Disable: When driven to 1b, prevents the Port from entering TX L0s.

Configuration Management Interface

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

Table 18: **Configuration Management Interface Port Descriptions**

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

Configuration Status Interface

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

Table 19: **Configuration Status Interface Port Descriptions**

Port	I/O	Width	Description
cfg_phy_link_down	O	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	O	2	Configuration Link Status Status of the PCI Express link. • 00b: No receivers detected • 01b: Link training in progress • 10b: Link up, DL initialization in progress • 11b: Link up, DL initialization completed
cfg_negotiated_width	O	3	Negotiated Link Width This output indicates the negotiated width of the given PCI Express Link and is valid when cfg_phy_link_status[1:0] == 11b (DL Initialization is complete). Negotiated Link Width values: • 000b = x1 • 001b = x2 • 010b = x4 • 011b = x8 • 100b = x16 • Other values are reserved.
cfg_current_speed	O	2	Current Link Speed This signal outputs the current link speed of the given PCI Express Link. • 00b: 2.5 GT/s PCI Express Link Speed • 01b: 5.0 GT/s PCI Express Link Speed • 10b: 8.0 GT/s PCI Express Link Speed • 11b: 16.0 GT/s PCI Express Link Speed

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

Port	I/O	Width	Description
cfg_max_payload	O	2	Max_Payload_Size This signal outputs the maximum payload size from Device Control register bits 7 down to 5. This field sets the maximum TLP payload size. As a Receiver, the logic must handle TLPs as large as the set value. As a Transmitter, the logic must not generate TLPs exceeding the set value. • 00b: 128 bytes maximum payload size • 01b: 256 bytes maximum payload size • 10b: 512 bytes maximum payload size • 11b: 1024 bytes maximum payload size
cfg_max_read_req	O	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
cfg_function_status	O	16	Configuration Function Status These outputs indicate the states of the Command register bits in the PCI configuration space of each function. These outputs are used to enable requests and completions from the host logic. The assignment of bits is as follows: • Bit 0: Function 0 I/O Space Enable • Bit 1: Function 0 Memory Space Enable • Bit 2: Function 0 Bus Master Enable • Bit 3: Function 0 INTx Disable • Bit 4: Function 1 I/O Space Enable • Bit 5: Function 1 Memory Space Enable • Bit 6: Function 1 Bus Master Enable • Bit 7: Function 1 INTx Disable • Bit 8: Function 2 I/O Space Enable • Bit 9: Function 2 Memory Space Enable • Bit 10: Function 2 Bus Master Enable • Bit 11: Function 2 INTx Disable • Bit 12: Function 3 I/O Space Enable • Bit 13: Function 3 Memory Space Enable • Bit 14: Function 3 Bus Master Enable • Bit 15: Function 3 INTx Disable

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

Port	I/O	Width	Description
cfg_vf_status	O	504	Configuration Virtual Function Status • Bit 0: Virtual function 0: Configured/Enabled by the software. • Bit 1: Virtual function 0: PCI Command register, Bus Master Enable. • Bit 2: Virtual function 1: Configured/Enabled by software. • Bit 3: Virtual function 1: PCI Command register, Bus Master Enable.
cfg_function_power_state	O	12	Configuration Function Power State These outputs indicate the current power state of the physical functions. Bits [2:0] capture the power state of function 0, and bits [5:3] capture that of function 1, and so on. The possible power states are: • 000: D0_uninitialized • 001: D0_active • 010: D1 • 100: D3_hot • Other values are reserved.
cfg_vf_power_state	O	756	Configuration Virtual Function Power State These outputs indicate the current power state of the virtual functions. Bits [2:0] capture the power state of virtual function 0, and bits [5:3] capture that of virtual function 1, and so on. The possible power states are: • 000: D0_uninitialized • 001: D0_active • 010: D1 • 100: D3_hot • Other values are reserved.
cfg_link_power_state	O	2	Current power state of the PCI Express link, and is valid when cfg_phy_link_status[1:0] == 11b (DL Initialization is complete). • 00: L0 • 01: TX L0s • 10: L1 • 11: L2/3 Ready

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

Port	I/O	Width	Description
cfg_local_error_out	O	5	Local Error Conditions: Error priority is noted and Priority 0 has the highest priority. • 00000b - Reserved • 00001b - Physical Layer Error Detected (Priority 16) • 00010b - Link Replay Timeout (Priority 12) • 00011b - Link Replay Rollover (Priority 13) • 00100b - Link Bad TLP Received (Priority 10) • 00101b - Link Bad DLLP Received (Priority 11) • 00110b - Link Protocol Error (Priority 9) • 00111b - Replay Buffer RAM Correctable ECC Error (Priority 22) • 01000b - Replay Buffer RAM Uncorrectable ECC Error (Priority 3) • 01001b - Receive Posted Request RAM Correctable ECC Error (Priority 20) • 01010b - Receive Posted Request RAM Uncorrectable ECC Error (Priority 1) • 01011b - Receive Completion RAM Correctable ECC Error (Priority 21) • 01100b - Receive Completion RAM Uncorrectable ECC Error (Priority 2) • 01101b - Receive Posted Buffer Overflow Error (Priority 5) • 01110b - Receive Non Posted Buffer Overflow Error (Priority 6) • 01111b - Receive Completion Buffer Overflow Error (Priority 7) • 10000b - Flow Control Protocol Error (Priority 8) • 10001b - Transmit Parity Error Detected (Priority 4) • 10010b - Unexpected Completion Received (Priority 15) • 10011b - Completion Timeout Detected (Priority 14) • 10100b - AXI4ST RQ INTFC Packet Drop (Priority 17) • 10101b - AXI4ST CC INTFC Packet Drop (Priority 18) • 10110b - AXI4ST CQ Poisoned Drop (Priority 19) • 10111b - User Signaled Internal Correctable Error (Priority 23) • 11000b - User Signaled Internal Uncorrectable Error (Priority 0) • 11001b - 11111b - Reserved
cfg_local_error_valid	O	1	Local Error Conditions Valid: Block activates this output for one cycle when any of the errors in cfg_local_error_out[4:0] are encountered. When driven 1b cfg_local_error_out[4:0] indicates local error type. Priority of error reporting (for the case of concurrent errors) is noted.

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

Port	I/O	Width	Description
cfg_rx_pm_state	O	2	Current RX Active State Power Management L0s State: Encoding is listed below and valid when cfg_ltssm_state is indicating L0: • RX_NOT_IN_L0s = 0 • RX_L0s_ENTRY = 1 • RX_L0s_IDLE = 2 • RX_L0s_FTS = 3
cfg_tx_pm_state	O	2	Current TX Active State Power Management L0s State: Encoding is listed below and valid when cfg_ltssm_state is indicating L0: • TX_NOT_IN_L0s = 0 • TX_L0s_ENTRY = 1 • TX_L0s_IDLE = 2 • TX_L0s_FTS = 3

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

Port	I/O	Width	Description
cfg_ltssm_state	O	6	LTSSM State. Shows the current LTSSM state: • 00: Detect.Quiet • 01: Detect.Active • 02: Polling.Active • 03: Polling.Compliance • 04: Polling.Configuration • 05: Configuration.Linkwidth.Start • 06: Configuration.Linkwidth.Accept • 07: Configuration.Lanenum.Accept • 08: Configuration.Lanenum.Wait • 09: Configuration.Complete • 0A: Configuration.Idle • 0B: Recovery.RcvrLock • 0C: Recovery.Speed • 0D: Recovery.RcvrCfg • 0E: Recovery.Idle • 10: L0 • 11-16: Reserved • 17: L1.Entry • 18: L1.Idle • 19-1A: Reserved • 20: Disabled • 21: Loopback_Entry_Master • 22: Loopback_Active_Master • 23: Loopback_Exit_Master • 24: Loopback_Entry_Slave • 25: Loopback_Active_Slave • 26: Loopback_Exit_Slave • 27: Hot_Reset • 28: Recovery_Equalization_Phase0 • 29: Recovery_Equalization_Phase1 • 2a: Recovery_Equalization_Phase2 • 2b: Recovery_Equalization_Phase3
cfg_rcb_status	O	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 19: **Configuration Status Interface Port Descriptions** *(cont'd)*

Port	I/O	Width	Description
cfg_dpa_substate_change	O	4	Dynamic Power Allocation Substate Change. In Endpoint mode, the core generates a one-cycle pulse on one of these outputs when a Configuration Write transaction writes into the Dynamic Power Allocation Control register to modify the DPA power state of the device. A pulse on bit 0 indicates such a DPA event for PF0 and a pulse on bit 1 indicates the same for PF1. The other 2 bits are reserved.These outputs are not active in Root Port mode.
cfg_obff_enable	O	2	Optimized Buffer Flush Fill Enable. This output reflects the setting of the OBFF Enable field in the Device Control 2 register. • 00: OBFF disabled. • 01: OBFF enabled using message signaling, Variation A. • 10: OBFF enabled using message signaling, Variation B. • 11: OBFF enabled using WAKE# signaling.
cfg_pl_status_change	O	1	This output is used by the core in Root Port mode to signal one of the following link training-related events: • The link bandwidth changed as a result of the change in the link width or operating speed and the change was initiated locally (not by the link partner), without the link going down. This interrupt is enabled by the Link Bandwidth Management Interrupt Enable bit in the Link Control register. The status of this interrupt can be read from the Link Bandwidth Management Status bit of the Link Status register; or • The link bandwidth changed autonomously as a result of the change in the link width or operating speed and the change was initiated by the remote node. This interrupt is enabled by the Link Autonomous Bandwidth Interrupt Enable bit in the Link Control register. The status of this interrupt can be read from the Link Autonomous Bandwidth Status bit of the Link Status register; or • The Link Equalization Request bit in the Link Status 2 register was set by the hardware because it received a link equalization request from the remote node. This interrupt is enabled by the Link Equalization Interrupt Enable bit in the Link Control 3 register. The status of this interrupt can be read from the Link Equalization Request bit of the Link Status 2 register. The pl_interrupt output is not active when the core is configured as an Endpoint.
cfg_ext_tag_enable	O	1	Extended Tag Enable:Per function state of Device Control Register Ext Tag (8-Bit) Enable bit.
cfg_atomic_requester_enable	O	4	Atomic Operation Requester Enable: Per function state of Device Control2 Register AtomicOp Requester Enable bit.
cfg_10b_tag_requester_enabl e	O	4	10b Tag Requester Enable: Per function state of Device Control2 Register 10-Bit Tag Requester Enable bit.

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

Port	I/O	Width	Description
pcie_tfc_nph_av	O	4	This output provides an indication of the currently available header credit for Non-Posted TLPs on the transmit side of the core. The user logic can check this output before transmitting a Non-Posted request on the requester request interface, to avoid blocking the interface when no credit is available. The encodings are: • 0000: No credit available • 0001: 1 credit available • 0010: 2 credits available • ... • 1110: 14 credits available • 1111: 15 or more credits available Because of pipeline delays, the value on this output can not include the credit consumed by the Non-Posted requests in the last eight cycles or less. The user logic must adjust the value on this output by the credit consumed by the Non-Posted requests it sent in the previous clock cycles, if any.
pcie_tfc_npd_av	O	4	This output provides an indication of the currently available payload credit for Non-Posted TLPs on the transmit side of the core. The user logic checks this output before transmitting a Non-Posted request on the requester request interface, to avoid blocking the interface when no credit is available. The encodings are: • 0000: No credit available • 0001: 1 credit available • 0010: 2 credits available • ... • 1110: 14 or more credits available • 1111: 15 or more credits available Because of pipeline delays, the value on this output does not include the credit consumed by the Non-Posted requests sent by the user logic in the last eight clock cycles or less. The user logic must adjust the value on this output by the credit consumed by the Non-Posted requests it sent in the previous clock cycles, if any.
pcie_rq_tag_av	O	4	This output provides an indication of the number of free tags available for allocation to Non-Posted requests on the PCIe master side of the core. The user logic checks this output before transmitting a Non-Posted request on the requester request interface, to avoid blocking the interface when no tags are available. The encodings are: • 0000: No tags available • 0001: 1 tag available • 0010: 2 tags available • ... • 1110: 14 tags available • 1111: 15 or more tags available Because of pipeline delays, the value on this output does not include the tags consumed by the Non-Posted requests sent by the user logic in the last 8 clock cycles or less. The user logic must adjust the value on this output by the number of Non-Posted requests it sent in the previous clock cycles, if any.

Configuration Received Message Interface

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

Table 20: **Configuration Received Message Interface**

Port	I/O	Width	Description
cfg_msg_received	O	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. The following table lists the number of cycles of cfg_msg_received assertion, and the parameters transferred on cfg_msg_data[7:0] in each cycle, for each type of message. 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	O	8	This bus is used to transfer any parameters associated with the Received Message. The information it carries in each cycle for various message types is listed in the previous table.
cfg_msg_received_type	O	5	Received message type. When cfg_msg_received is High, these five bits indicate the type of message being signaled by the core. The various message types are listed in the previous table.

Table 21: **Message Type Encoding on Receive Message Interface**

cfg_msg_received_type[4:0]	Message Type
0	ERR_COR
1	ERR_NONFATAL
2	ERR_FATAL
3	Assert_INTA
4	Deassert_ INTA
5	Assert_INTB
6	Deassert_ INTB
7	Assert_INTC
8	Deassert_ INTC
9	Assert_INTD
10	Deassert_ INTD
11	PM_PME

Table 21: **Message Type Encoding on Receive Message Interface** *(cont'd)*

cfg_msg_received_type[4:0]	Message Type
12	PME_TO_Ack
13	PME_Turn_Off
14	PM_Active_State_Nak
15	Set_Slot_Power_Limit
16	Latency Tolerance Reporting (LTR)
17	Reserved
18	Unlock
19	Vendor_Defined Type 0
20	Vendor_Defined Type 1
25 – 31	Reserved

Table 22: **Message Parameters on Receive Message Interface**

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

Table 22: **Message Parameters on Receive Message Interface** *(cont'd)*

Message Type	Number of cycles of cfg_msg_received assertion	Parameter transferred on cfg_msg_received_data[7:0]
Vendor_Defined Type 0	4 cycles when no data present, 8 cycles when data present.	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number Cycle 3: Vendor ID[7:0] Cycle 4: Vendor ID[15:8] Cycle 5: bits [7:0] of payload Cycle 6: bits [15:8] of payload Cycle 7: bits [23:16] of payload Cycle 8: bits [31:24] of payload
Vendor_Defined Type 1	4 cycles when no data present, 8 cycles when data present.	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number Cycle 3: Vendor ID[7:0] Cycle 4: Vendor ID[15:8] Cycle 5: bits [7:0] of payload Cycle 6: bits [15:8] of payload Cycle 7: bits [23:16] of payload Cycle 8: bits [31:24] of payload

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 23: **Configuration Transmit Message Interface**

Port	I/O	Width	Description
cfg_msg_transmit	I	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. ***Note*:** When PASID_CAP_ON = TRUE, this port is not available for use.

Table 23: **Configuration Transmit Message Interface** *(cont'd)*

Port	I/O	Width	Description
cfg_msg_transmit_type	I	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 ***Note*:** When PASID_CAP_ON = TRUE, this port is not available for use.
cfg_msg_transmit_data	I	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[30:29] <= Repurposing to pass PASID information. assigned to s_axis_rq_tuser[182:181] inside the IP. • 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[14:13] <= Repurposing to pass PASID information. assigned to s_axis_rq_tuser[180:179] inside the IP. • cfg_msg_transmit_data[12:10] < No-Snoop Latency Scale • cfg_msg_transmit_data[9:0] < No-Snoop Latency Value 001b: Reserved. 010b: SSPL - • cfg_msg_transmit_data[9:0] < {Slot Power Limit Scale, Slot Power Limit Value} 011b: PM_PME - • cfg_msg_transmit_data[7:0] <= 8'h00-8'hFF PF0-3 - 8'h00-8'h03 Other encodings are reserved 100b - 111b: Reserved ***Note*:** When PASID_CAP_ON = TRUE, this port is not available for use.
cfg_msg_transmit_done	O	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 24: **Configuration Flow Control Interface**

Port	I/O	Width	Description
cfg_fc_ph	O	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_ph_scale	O	2	Posted Header Flow Control Credits Scale This output provides the scale of Posted Header Flow Control Credit number (cfg_fc_ph). The flow control information to bring out on this core is selected by the cfg_fc_sel[2:0] input.
cfg_fc_pd	O	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_pd_scale	O	2	Posted Data Flow Control Credits Scale This output provides the scale of Posted Data Flow Control Credit number (cfg_fc_pd). The flow control information to bring out on this core is selected by the cfg_fc_sel[2:0] input
cfg_fc_nph	O	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_nph_scale	O	2	Non-Posted Header Flow Control Credits Scale This output provides the scale of Non-Posted Header Flow Control Credit number (cfg_fc_nph). The flow control information to bring out on this core is selected by the cfg_fc_sel[2:0] input
cfg_fc_npd	O	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_npd_scale	O	2	Non-Posted Data Flow Control Credits Scale This output provides the scale of Non-Posted Data Flow Control Credit number (cfg_fc_npd). The flow control information to bring out on this core is selected by the cfg_fc_sel[2:0] input
cfg_fc_cplh	O	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.

Table 24: **Configuration Flow Control Interface** *(cont'd)*

Port	I/O	Width	Description
cfg_fc_cplh_scale	O	2	Completion Header Flow Control Credit Scale This output provides the scale of Completion Header Flow Control Credit number (cfg_fc_cplh). The flow control information to bring out on this core is selected by the cfg_fc_sel[2:0] input
cfg_fc_cpld	O	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].
cfg_fc_cpld_scale	O	2	Completion Data Flow Control Credit Scale This output provides the scale of Completion Data Flow Control Credit number (cfg_fc_cpld). The flow control information to bring out on this core is selected by the cfg_fc_sel[2:0] input
cfg_fc_sel	I	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 credit limit to link partner (rlimit) • 001: Transmit credits consumed (tconsumed) & wide posted and completion • 010: Receive credits consumed by link partner (rconsumed) • 011: Transmit credits consumed (tconsumed) wide nonposted and completion • 100: Transmit user credits available (tlimit - tconsumed) • 101: Transmit credit limit (tlimit) • 110: Transmit credits consumed (tconsumed) • 111: Receive free 32B word count 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. Infinite credit for transmit credits available (cfg_fc_sel == 3'b100) is signaled as 8'h80, 12'h800 for header and data credits, respectively. For all other cfg_fc_sel selections, infinite credit is signaled as 8'h00, 12'h000, respectively, for header and data categories.

Configuration Control Interface

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

- Set the configuration space.

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

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

Table 25: **Configuration Control Interface Port Descriptions**

Port	I/O	Width	Description
cfg_hot_reset_in	I	1	Configuration Hot Reset In In RP mode, assertion transitions LTSSM to hot reset state, active-High.
cfg_hot_reset_out	O	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	I	1	Configuration Configuration Space Enable When this input is set to 0 in the Endpoint mode, the core generates a CRS Completion in response to Configuration Requests. This port should be held deasserted when the core configuration registers are loaded from the DRP due to a change in attributes. This prevents the core from responding to Configuration Requests before all the registers are loaded. This input can be High when the power-on default values of the Configuration registers do not need to be modified before Configuration space enumeration. This input is not applicable for Root Port mode.
cfg_dsn	I	64	Configuration Device Serial Number Indicates the value that should be transferred to the Device Serial Number Capability on PF0. Bits [31:0] are transferred to the first (Lower) Dword (byte offset `0x4h` of the Capability), and bits [63:32] are transferred to the second (Upper) Dword (byte offset 0x8h of the Capability). If this value is not statically assigned, the user application must pulse user_cfg_input_update after it is stable.
cfg_ds_bus_number	I	8	Configuration Downstream Bus Number Downstream Port: Provides the bus number portion of the Requester ID (RID) of the Downstream Port. This is used in TLPs generated inside the core, such as UR Completions and Power-management messages; it does not affect TLPs presented on the AXI interface. Upstream Port: No role.
cfg_ds_device_number	I	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 25: **Configuration Control Interface Port Descriptions** *(cont'd)*

Port	I/O	Width	Description
cfg_ds_function_number	I	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	I	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	O	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_ds_port_number	I	8	Configuration Downstream Port Number Provides the port number field in the Link Capabilities register.
cfg_err_cor_in	I	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. ***Note*:** When PASID_CAP_ON = TRUE, this pin is not available for use.

Table 25: **Configuration Control Interface Port Descriptions** *(cont'd)*

Port	I/O	Width	Description
cfg_err_cor_out	O	1	Correctable Error Detected In the Endpoint mode, the Block activates this output for one cycle when it has detected a correctable error and its reporting is not masked. When multiple functions are enabled, this is the logical OR of the correctable error status bits in the Device Status Registers of all functions.
cfg_err_fatal_out	O	1	Fatal Error Detected In the Endpoint mode, the block activates this output for one cycle when it has detected a fatal error and its reporting is not masked. When multiple functions are enabled, this output is the logical OR of the fatal error status bits in the Device Status Registers of all functions. In the Root Port mode, this output is activated on detection of a local fatal error, when its reporting is not masked. This output does not respond to any errors signaled by remote devices using PCI Express error messages. These error messages are delivered to the user through the message interface.
cfg_err_nonfatal_out	O	1	Non Fatal Error Detected In the Endpoint mode, the block activates this output for one cycle when it has detected a non fatal error and its reporting is not masked. When multiple functions are enabled, this output is the logical OR of the non fatal error status bits in the Device Status Registers of all functions. In the Root Port mode, this output is activated on detection of a local non fatal error, when its reporting is not masked. This output does not respond to any errors signaled by remote devices using PCI Express error messages. These error messages are delivered through the message interface.
cfg_err_uncor_in	I	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. ***Note*:** When PASID_CAP_ON = TRUE, this pin is not available for use.
cfg_flr_done	I	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. The core will issue CRS to configurations requests to a particular Physical Function till cfg_flr_done is not asserted when cfg_flr_in_process =1 for that Physical Function.

Table 25: **Configuration Control Interface Port Descriptions** *(cont'd)*

Port	I/O	Width	Description
cfg_vf_flr_done	I	1	Function Level Reset for Virtual Function is Complete The user application must assert this input when it has completed the reset operation of the Virtual Function. This causes the core to deassert cfg_vf_flr_in_process for function *i* and to re-enable configuration accesses to the virtual function. The core will issue CRS to configuration requests to a particular Virtual Function till cfg_vf_flr_done is not asserted when cfg_vf_flr_in_process = 1 for that Virtual Function.
cfg_vf_flr_func_num	I	8	Function Level Reset for Virtual Function i is Complete. The user application drives a valid Virtual Function number on this input along with asserting cfg_vf_flr_done when the reset operation of Virtual Function i completes. Valid entries are 8'h04-8'hFF for VF0-VF251. Values 8'h00-8'h03 are reserved.
cfg_flr_in_process	O	4	Function Level Reset In Process The core asserts bit *i* of this bus when the host initiates a reset of physical function *i* through its FLR bit in the configuration space. The core continues to hold the output High until the user sets the corresponding cfg_flr_done input for the corresponding physical function to indicate the completion of the reset operation.
cfg_vf_flr_in_process	O	252	Function Level Reset In Process for Virtual Function The core asserts bit *i* of this bus when the host initiates a reset of virtual function *i* though its FLR bit in the configuration space. The core continues to hold the output High until the user sets the cfg_vf_flr_done input and drives cfg_vf_flr_func_num with the corresponding function to indicate the completion of the reset operation.
cfg_req_pm_transition_l23_ready	I	1	When the core is configured as an Endpoint, the user application asserts this input to transition the power management state of the core to L23_READY (see Chapter 5 of the PCI Express Specification (see *PCI-SIG Specifications* (https://www.pcisig.com/specifications) for a detailed description of power management). This is done after the PCI functions in the core are placed in the D3 state and after the user application acknowledges the PME_Turn_Off message from the Root Complex. Asserting this input causes the link to transition to the L3 state, and requires a hard reset to resume operation. This input can be hardwired to 0 if the link is not required to transition to L3. This input is not used in Root Complex mode.
cfg_link_training_enable	I	1	This input must be set to 1 to enable the Link Training Status State Machine (LTSSM) to bring up the link. Setting it to 0 forces the LTSSM to stay in the Detect.Quiet state.
cfg_bus_number	O	8	Bus Number Captured from received CfgWr Type0 is presented. Active only in the Endpoint Configuration.
cfg_vend_id	I	16	Configuration Vendor ID: Indicates the value that should be transferred to the PCI Capability Structure Vendor ID field on all PFs.

Table 25: **Configuration Control Interface Port Descriptions** *(cont'd)*

Port	I/O	Width	Description
cfg_subsys_vend_id	I	16	Configuration Subsystem Vendor ID: Indicates the value that should be transferred to the Type 0 PCI Capability Structure Subsystem Vendor ID field on all PFs.
cfg_dev_id_pf0	I	16	Configuration Device ID PF0: Indicates the value that should be transferred to the PCI Capability Structure Device ID field on PF0.
cfg_dev_id_pf1	I	16	Configuration Device ID PF1: Indicates the value that should be transferred to the PCI Capability Structure Device ID field on PF1.
cfg_dev_id_pf2	I	16	Configuration Device ID PF2: Indicates the value that should be transferred to the PCI Capability Structure Device ID field on PF2.
cfg_dev_id_pf3	I	16	Configuration Device ID PF3: Indicates the value that should be transferred to the PCI Capability Structure Device ID field on PF3.
cfg_rev_id_pf0	I	8	Configuration Revision ID PF0: Indicates the value that should be transferred to the PCI Capability Structure Revision ID field on PF0.
cfg_rev_id_pf1	I	8	Configuration Revision ID PF1: Indicates the value that should be transferred to the PCI Capability Structure Revision ID field on PF1.
cfg_rev_id_pf2	I	8	Configuration Revision ID PF2: Indicates the value that should be transferred to the PCI Capability Structure Revision ID field on PF2.
cfg_rev_id_pf3	I	8	Configuration Revision ID PF3: Indicates the value that should be transferred to the PCI Capability Structure Revision ID field on PF3.
cfg_subsys_id_pf0	I	16	Configuration Subsystem ID PF0: Indicates the value that should be transferred to the Type 0 PCI Capability Structure Subsystem ID field on PF0.
cfg_subsys_id_pf1	I	16	Configuration Subsystem ID PF1: Indicates the value that should be transferred to the Type 0 PCI Capability Structure Subsystem ID field on PF1.
cfg_subsys_id_pf2	I	16	Configuration Subsystem ID PF2: Indicates the value that should be transferred to the Type 0 PCI Capability Structure Subsystem ID field on PF2.
cfg_subsys_id_pf3	I	16	Configuration Subsystem ID PF3: Indicates the value that should be transferred to the Type 0 PCI Capability Structure Subsystem ID field on PF3.

Configuration Interrupt Controller Interface

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

Legacy Interrupt Interface

Table 26: **Legacy Interrupt Interface Port Descriptions**

Name	I/O	Width	Description
cfg_interrupt_int	I	4	Configuration INTx Vector: When the core is configured as EP, these four inputs are used by the user application to signal an interrupt from any of its PCI Functions to the RC using the Legacy PCI Express Interrupt Delivery mechanism of PCI Express. These four inputs correspond to INTA, INTB, INTC, and INTD of the PCI bus, respectively. Asserting one of these signals causes the core to send out an Assert_INTx message, and deasserting the signal causes the core to transmit a Deassert_INTx message. ***Note***: When PASID_CAP_ON = TRUE, this pin is not available for use.
cfg_interrupt_sent	O	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	I	4	Configuration INTx Interrupt Pending: Per Function indication of a pending interrupt from the user. cfg_interrupt_pending[0] corresponds to Function #0. Each of these inputs is connected to the Interrupt Pending bits of the PCI Status Register of the corresponding Function. ***Note***: When PASID_CAP_ON = TRUE, this pin is not available for use.

MSI Interrupt Interface

Table 27: **MSI Interrupt Interface Port Descriptions**

Name	I/O	Width	Description
cfg_interrupt_msi_enable	O	4	Configuration Interrupt MSI Function Enabled Indicates that the Message Signaling Interrupt (MSI) messaging is enabled, per Function. These outputs reflect the setting of the MSI Enable bits in the MSI Control Register of Physical Functions 0 – 3.

Table 27: **MSI Interrupt Interface Port Descriptions** *(cont'd)*

Name	I/O	Width	Description
cfg_interrupt_msi_int	I	32	Configuration Interrupt MSI Vector When configured in the Endpoint mode to support MSI interrupts, these inputs are used to signal the 32 distinct interrupt conditions associated with a PCI Function (Physical or Virtual) from the user logic to the core. The Function number must be specified on the input cfg_interrupt_msi_function_number. After placing the Function number on the input cfg_interrupt_msi_function_number, the user logic must activate one of these signals for one cycle to transmit an interrupt. The user logic must not activate more than one of the 32 interrupt inputs in the same cycle. The core internally registers the interrupt condition on the 0-to-1 transition of any bit in cfg_interrupt_msi_int. After asserting an interrupt, the user logic must wait for the cfg_interrupt_msi_sent or cfg_interrupt_msi_fail indication from the core before asserting a new interrupt.
cfg_interrupt_msi_function_number	I	8	Configuration MSI Initiating Function Indicates the Endpoint Function # initiating the MSI interrupt. • 8'h00 – 8'h03: PF 0 – PF 3 • 8’h04 – 8’hFF: VF 0 – VF 252 • Other encodings are reserved.
cfg_interrupt_msi_sent	O	1	Configuration Interrupt MSI Interrupt Sent The core generates a one-cycle pulse on this output to signal that an MSI or MSI-X interrupt message has been transmitted on the link. The user logic must wait for this pulse before signaling another interrupt condition to the core.
cfg_interrupt_msi_fail	O	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 logic must retransmit the MSI interrupt in this case.
cfg_interrupt_msi_mmenable	O	12	Configuration Interrupt MSI Function Multiple Message Enable When the core is configured in the Endpoint mode to support MSI interrupts, these outputs are driven by the 'Multiple Message Enable' bits of the MSI Control Register associated with Physical Functions. These bits encode the number of allocated MSI interrupt vectors for the corresponding Function. Bits [2:0] correspond to Physical Function 0, bits [5:3] correspond to PF 1, and so on. The valid encodings of the 3 bits are: • 000b: 1 vector • 001b: 2 vectors • 010b: 4 vectors • 011b: 8 vectors • 100b: 16 vectors • 101b: 32 vectors

Table 27: **MSI Interrupt Interface Port Descriptions** *(cont'd)*

Name	I/O	Width	Description
cfg_interrupt_msi_pending_status	I	32	Configuration MSI Interrupt Pending Status These inputs are provided for the user to indicate the interrupt pending status of the MSI interrupts associated with the Physical Functions. When the status of a MSI interrupt associated with a PF changes, the user must place the new interrupt status on these inputs, along with the corresponding Function number on the cfg_interrupt_msi_pending_status_function_num input, and activate the cfg_interrupt_msi_pending_status_data_enable input for one cycle. The core then latches the new status in its MSI Pending Bits Register of the corresponding Physical Function.
cfg_interrupt_msi_pending_status_function_num	I	2	Configuration Interrupt MSI Pending Target Function Number • 00 = PF 0 • 01 = PF 1 • 10 = PF 2 • 11 = PF 3 This input is used to identify the Function number when the user places interrupt status on the cfg_interrupt_msi_pending_status inputs.
cfg_interrupt_msi_pending_status_data_enable	I	1	Configuration Interrupt MSI Pending Data Valid The user application asserts this signal together with cfg_interrupt_msi_pending_status and cfg_interrupt_msi_pending_status_function_num values to update the MSI Pending Bits in the corresponding function.
cfg_interrupt_msi_mask_update	O	1	Configuration Interrupt MSI Function Mask Updated The SR-IOV core asserts this for 1 cycle when the MSI Mask Register of any enabled PFs has changed its value. The user can then read the new mask settings from the cfg_interrupt_msi_data outputs.
cfg_interrupt_msi_select	I	2	Configuration Interrupt MSI Select These inputs are used to select the Function number for reading the MSI Mask Register setting from the core. Values 0 – 3 correspond to Physical Functions 0 – 3, respectively. The mask MSI Mask Register contents of the selected PF appear on the output cfg_interrupt_msi_data after one cycle.
cfg_interrupt_msi_data	O	32	Configuration Interrupt MSI Data These output reflect the MSI Mask Register setting of the Physical Function selected by the cfg_interrupt_msi_select input.
cfg_interrupt_msi_attr	I	3	Configuration Interrupt MSI TLP Attribute These bits enable you to set the Attribute bits that are used for both MSI and MSI-X interrupt requests. • Bit 0 is the No Snoop bit. • Bit 1 is the Relaxed Ordering bit. • Bit 2 is the ID-Based Ordering bit. The core samples these bits on a 0-to-1 transition on cfg_interrupt_msi_int bits (when using MSI) or cfg_interrupt_msix_int (when using MSI-X).

MSI-X Interrupt External Interface

Table 28: **MSI-X Interrupt External Interface Port Descriptions**

Name	I/O	Width	Description
cfg_interrupt_msix_enable	O	4	Configuration Interrupt MSI-X Function Enabled These outputs reflect the setting of the MSI-X Enable bits of the MSI-X Control Register of Physical Functions 0 – 3.
cfg_interrupt_msix_mask	O	4	Configuration Interrupt MSI-X Function Mask These outputs reflect the setting of the MSI-X Function Mask bits of the MSI-X Control Register of Physical Functions 0 – 3.
cfg_interrupt_msix_vf_enable	O	252	Configuration Interrupt MSI-X Enable from VFs These outputs reflect the setting of the MSI-X Enable bits of the MSI-X Control Register of Virtual Functions 0 – 251.
cfg_interrupt_msix_vf_mask	O	252	Configuration Interrupt MSI-X VF Mask These outputs reflect the setting of the MSI-X Function Mask bits of the MSI-X Control Register of Virtual Functions 0 – 251.
cfg_interrupt_msix_address	I	64	Configuration Interrupt MSI-X Address When the core is configured to support MSI-X interrupts and when the MSI-X Table is implemented in user memory, this bus is used by the user logic to communicate the address to be used to generate an MSI-X interrupt.
cfg_interrupt_msix_data	I	32	Configuration Interrupt MSI-X Data When the core is configured to support MSI-X interrupts and when the MSI-X Table is implemented in user memory, this bus is used by the user logic to communicate the data to be used to generate an MSI-X interrupt.
cfg_interrupt_msix_int	I	1	Configuration Interrupt MSI-X Data Valid The assertion of this signal by the user indicates a request from the user to send an MSI-X interrupt. The user must place the identifying information on the designated inputs before asserting this interrupt. When the MSI-X Table and Pending Bit Array are implemented in user memory, the identifying information consists of the memory address, data, and the originating Function number for the interrupt. These must be placed on the cfg_interrupt_msix_address[63:0], cfg_interrupt_msix_data[31:0], and cfg_interrupt_msi_function_number[7:0], respectively. The core internally registers these parameters on the 0-to-1 transition of cfg_interrupt_msix_int. When the MSI-X Table and Pending Bit Array are implemented by the core, the identifying information consists o the originating Function number for the interrupt and the interrupt vector. These must be placed on cfg_interrupt_msi_function_number[7:0] and cfg_interrupt_msi_int[31:0], respectively. Bit i of cfg_interrupt_msi_int[31:0] represents interrupt vector i, and only one of the bits of this bus can be set to 1 when asserting cfg_interrupt_msix_int. After asserting an interrupt, the user logic must wait for the cfg_interrupt_msi_sent or cfg_interrupt_msi_fail indication from the core before asserting a new interrupt.

Table 28: **MSI-X Interrupt External Interface Port Descriptions** *(cont'd)*

Name	I/O	Width	Description
cfg_interrupt_msix_vec_pending	I	2	Configuration Interrupt MSI-X Pending Bit Query/Clear These mode bits are used only when the core is configured to include the MSI-X Table and Pending Bit Array. These two bits are set when asserting cfg_interrupt_msix_int to send an MSI-X interrupt, to perform certain actions on the MSI-X Pending Bit associated with the selected Function and interrupt vector. The various modes are: • 00b: Normal interrupt generation. If the Mask bit associated with the vector was 0 when cfg_interrupt_msix_int was asserted, the core transmits the MSI-X request TLP on the link. If the Mask bit was 1, the core does not immediately send the interrupt, but instead sets the Pending Bit associated with the interrupt vector in its MSI-X Pending Bit Array (and subsequently transmits the MSI-X request TLP when the Mask clears). In both cases, the core asserts cfg_interrupt_msi_sent for one cycle to indicate that the interrupt request was accepted. The user can distinguish these two cases by sampling the cfg_interrupt_msix_vec_pending_status output, which reflects the current setting of the MSI-X Pending Bit corresponding to the interrupt vector. • 01b: Pending Bit Query. In this mode, the core treats the assertion of one of the bits of cfg_interrupt_msix_int as a query for the status of its Pending Bit. The user must also place the Function number of the Pending Bit being queried on the cfg_interrupt_msi_function_number input. The core does not transmit a MSI-X request in response, but asserts cfg_interrupt_msi_sent for one cycle, along with the status of the MSI-X Pending Bit on the cfg_interrupt_msix_vec_pending_status output. • 10b: Pending Bit Clear. In this mode, the core treats the assertion of one of the bits of cfg_interrupt_msix_int as a request to clear its Pending Bit. The user must also place the Function number of the Pending Bit being queried on the cfg_interrupt_msi_function_number input. The core does not transmit a MSI-X request in response, but clears he MSI-X Pending Bit of the vector (if it is set), and activates cfg_interrupt_msi_sent for one cycle as the acknowledgment. The core also provides the previous state of the MSI-X Pending Bit on the cfg_interrupt_msix_vec_pending_status output, which can be sampled by the user to determine if the Pending Bit was cleared by the core before the user request (because the pending interrupt was actually transmitted). This mode can be used to implement a “polling mode” for MSI-X interrupts, where the interrupt is permanently masked and the software polls the Pending Bit to detect and service the interrupt. After each interrupt is serviced, the Pending Bit can be cleared through this interface.
cfg_interrupt_msix_vec_pending_ status	O	1	Configuration Interrupt MSI-X Pending Bit Status This output provides the status of the Pending Bit associated with an MSI-X interrupt, in response to query using the cfg_interrupt_msix_vec_pending input. It is active only when the core is configured to include the MSI-X Table and Pending Bit Array.

MSI-X Interrupt Internal Interface

Table 29: **MSI-X Interrupt Internal Interface Port Descriptions**

Name	I/O	Width	Description
cfg_interrupt_msi_int	I	8	The core supports eight vectors per function and it is one-hot encoding, so each bit corresponds to one vector. See the description found in MSI Interrupt Interface.
cfg_interrupt_msi_function_number	I	8	See the description found in MSI Interrupt Interface.
cfg_interrupt_msi_attr	I	3	See the description found in MSI Interrupt Interface.
cfg_interrupt_msi_sent	O	1	See the description found in MSI Interrupt Interface.
cfg_interrupt_msi_fail	O	1	See the description found in MSI Interrupt Interface.
cfg_interrupt_msix_int	I	1	See the description found in MSI Interrupt Interface.
cfg_interrupt_msix_vec_pending	I	2	See the description found in MSI Interrupt Interface.
cfg_interrupt_msix_vec_pending_status	O	1	See the description found in MSI Interrupt Interface.
cfg_interrupt_msix_enable	O	4	See the description found in MSI Interrupt Interface.
cfg_interrupt_msix_mask	O	4	See the description found in MSI Interrupt Interface.
cfg_interrupt_msix_vf_enable	O	252	See the description found in MSI Interrupt Interface.
cfg_interrupt_msix_vf_mask	O	252	See the description found in MSI Interrupt Interface.

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 30: **Configuration Extend Interface Port Descriptions**

Port	I/O	Width	Description
cfg_ext_read_received	O	1	Configuration Extend Read Received. The Block asserts this output when it has received a configuration read request from the link. Set when **PCI Express Extended Configuration Space Enable** is selected in User Defined Configuration Capabilities in core configuration in the Vivado IDE. All received configuration reads with cfg_ext_register_number in the following ranges is considered to be the PCIe Extended Configuration Space. • Versal™ PCIe core: 0xE80-0xFFF All received configuration reads regardless of its address will be indicated by 1 cycle assertion of cfg_ext_read_received and valid data is driven on cfg_ext_register_number and cfg_ext_function_number. Only received configuration reads within the aforementioned ranges need to be responded by User Application outside of the IP.

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

Port	I/O	Width	Description
cfg_ext_write_received	O	1	Configuration Extend Write Received. The Block asserts this output when it has received a configuration write request from the link. Set when **PCI Express Extended Configuration Space Enable** is selected in User Defined Configuration Capabilities in the core configuration in the Vivado IDE. Data corresponding to all received configuration writes with cfg_ext_register_number in the range 0xb0-0xbf is presented on cfg_ext_register_number, cfg_ext_function_number, cfg_ext_write_data and cfg_ext_write_byte_enable. All received configuration writes with cfg_ext_register_number in the following ranges are presented on cfg_ext_register_number, cfg_ext_function_number, cfg_ext_wrte_data and cfg_ext_write_byte_enable. • Versal PCIe core: 0xE80-0xFFF
cfg_ext_register_number	O	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	O	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	O	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	O	4	Configuration Extend Write Byte Enable Byte enables for a configuration write transaction.
cfg_ext_read_data	I	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	I	1	Configuration Extend Read Data Valid The user application asserts this input to the core to supply data from an externally implemented configuration register. The core samples this input data on the next positive edge of the clock after it sets cfg_ext_read_received High. The core expects the assertions of this signal within 262144 ('h4_0000) clock cycles of user clock after receiving the read request on cfg_ext_read_received signal. If no response is received by this time, the core will send auto-response with 'h0 payload, and the user application must discard the response and terminate that particular request immediately.

Configuration VC1 Status Interface

The Configuration VC1 Status interface is used to determine the VC1 enablement or disablement by the software, and determine the VC1 resource status.

Table 31: **Configuration VC1 Status Interface Port Descriptions**

Port	I/O	Width	Description
cfg_vc1_enable	O	1	Configuration VC1 Enable: VC1 Resource Control Register:VC Enable bit. When 1b, indicates software has enabled VC1 operation. When 0b, indicates that VC1 is disabled by software.
cfg_vc1_negotiation_pending	O	1	Configuation VC1 Negotiation Pending: VC1 Resource Status Register:VC Negotiation Pending bit. When 1b, indicates VC1 negotiation (initialization or disabling) is in pending state.

Configuration PASID Interface

The Configuration PASID interface is used to determine the enablement of the PASID per function status by the software.

Table 32: **Configuration PASID Interface Port Descriptions**

Port	I/O	Width	Description
cfg_pasid_enable	O	4	Configuation PASID Enable: Per Function PASID Enable.
cfg_pasid_exec_permission_enable	O	4	Configuation PASID Exec Permission Enable: Per Function PASID Exec Permission Enable.
cfg_pasid_privil_mode_enable	O	4	Configuation PASID Privil Mode Enable: Per Function PASID Privil Mode Enable.

APB3 Interface

The APB3 (Advanced Peripheral Bus) interface is similar to the DRP interface in previous generations of the programmable logic integrated block for PCIe in earlier architectures.

***Note*:** The APB3 interface is not supported for simulation.

Table 33: **APB3 Interface Port Descriptions**

Port	I/O	Width	Description
apb3_clk	I	1	APB3 Clock. The rising edge of PCLK times all transfers on the APB.
apb3_paddr	I	9	APB3 Address. This is the APB address bus (DWORD (32-bit) addresses). It is 9-bits wide and is driven by the peripheral bus bridge unit.
apb3_penable	I	1	APB3 Enable. This signal indicates the second and subsequent cycles of an APB transfer.

Table 33: **APB3 Interface Port Descriptions** *(cont'd)*

Port	I/O	Width	Description
apb3_pwdata	I	32	APB3 Write Data. This bus is driven by the peripheral bus bridge unit during write cycles when PWRITE is HIGH. This bus can be up to 32 bits wide.
apb3_pwrite	I	1	APB3 Direction. This signal indicates an APB write access when HIGH and an APB read access when LOW.
apb3_psel	I	1	APB3 Select. The APB bridge unit generates this signal to each peripheral bus slave. It indicates that the slave device is selected and that a data transfer is required.
apb3_prdata	O	32	APB3 Read Data. The selected slave drives this bus during read cycles when PWRITE is LOW. This bus can be up to 32-bits wide.
apb3_pready	O	1	APB3 Ready. The slave uses this signal to extend an APB transfer.
apb3_pslverr	O	1	APB3 Slave Error. This signal indicates a transfer failure.
apb3_presetn	I	1	APB3 Reset This signal indicates an APB reset

Clock and Reset Interface

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

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

Table 34: **Clock and Reset Interface Port Descriptions**

Port	I/O	Width	Description
user_clk	O	1	User clock output (62.5, 125, or 250 MHz) This clock has a fixed frequency and is configured in the Vivado® Integrated Design Environment (IDE).
user_reset	O	1	This signal is deasserted synchronously with respect to user_clk. It is deasserted and asserted asynchronously with sys_reset assertion.
sys_clk	I	1	Reference clock This clock has a selectable frequency of 100 MHz,125 MHz and 250 MHz.
sys_clk_gt	I	1	PCIe reference clock for GT. This clock must be driven directly from IBUFDS (same definition and frequency as sys_clk). This clock has a selectable frequency of 100 MHz, 125 MHz and 250 MHz, which is the same as in sys_clk.

Table 34: **Clock and Reset Interface Port Descriptions** *(cont'd)*

Port	I/O	Width	Description
sys_reset	I	1	Fundamental reset input to the core (asynchronous) This input is active-Low by default to match the PCIe edge connector reset polarity.
phy_rdy_out	O	1	The phy ready signal indicates that the GT Wizard is ready. This signal is driven by phy_rst FSM on receiving the phy status from the GT Wizard core.

The PL PCIE4 does not have dedicated reset pin routing.

PCIe PHY IP Interface

The signals described in this section are based on a single-lane application. Signals can be per lane or per design; if not indicated in the description, the default is per design. Per design indicates that one signal controls all lanes (0 to N-1 lane). A per-lane signal on the PCIe PHY IP is in the form of {LaneN-1[Width-1:0], ...Lane1 [Width-1:0], Lane0[Width-1:0]}.

Clock and Reset Signals

Table 35: **Clock and Reset Signals**

Name	I/O	Width	Clock Domain	Description
phy_coreclk	I	1	coreclk	Core clock options: • 250 MHz • 500 MHz
phy_userclk	I	1	userclk	User clock options: • 62.5 MHz • 125 MHz • 250 MHz phy_userclk is edge aligned and phase aligned to phy_coreclk.
phy_mcapclk	I	1	mcapclk	Additional clock options: • 62.5 MHz • 125 MHz phy_mcapclk is edge aligned and phase aligned to phy_coreclk.

Table 35: **Clock and Reset Signals** *(cont'd)*

Name	I/O	Width	Clock Domain	Description
phy_pclk	I	1	pclk	PIPE interface clock options: • 125 MHz: Gen1 operating speed • 250 MHz: Gen2, Gen3, Gen4 operating speed • 500 MHz: Gen4 operating speed phy_pclk is edge aligned, but not phase aligned, to phy_coreclk and phy_userclk.

TX Data Signals

Table 36: **TX Data Signals**

Name	I/O	Width	Description
phy_txdata[63:0]	O	64	Parallel data output. Bits [63:32] are used for Gen4 only and must be ignored in Gen1, Gen2, and Gen3. Bits [31:16] are used for Gen3 only and must be ignored in Gen1 and Gen2. Per lane.
phy_txdatak[1:0]	O	2	Indicates whether TXDATA is control or data for Gen1 and Gen2 only. Per lane. • 0b: Data • 1b: Control
phy_txdata_valid	O	1	This signal allows the MAC to instruct the PHY to ignore TXDATA for one PCLK cycle. When High, this indicates that the PHY is to use TXDATA. When Low, this indicates the PHY is *not*to use TXDATA for one PCLK cycle. Gen3 and Gen4 only. Per lane.
phy_txstart_block	O	1	This signal allows the MAC to tell the PHY the starting byte for a 128b block. The starting byte for a 128b block must always start at bit [0] of TXDATA. Gen3 and Gen4 only. Per lane.
phy_txsync_header[1:0]	O	2	Provide the sync header for the PHY to use the next 130b block. The PHY reads this value when the txsync_block is asserted. Gen3 and Gen4 only. Per lane.

RX Data Signals

Table 37: **RX Data Signals**

Name	I/O	Width	Description
phy_rxdata[63:0]	I	64	PIPE data output from receiver. Bits[63:32] are used for Gen4 only and must be ignored in Gen1, Gen2, and Gen3. Bits[31:16] are used for Gen3 only and must be ignored in Gen1 and Gen2. Per lane.

Table 37: **RX Data Signals** *(cont'd)*

Name	I/O	Width	Description
phy_rxdatak[1:0]	I	2	Indicates whether RXDATA is control or data. Gen1 and Gen2 only. Per-lane. • 0b: Data • 1b: Control
phy_rxdata_valid	I	1	This signal allows the PHY to instruct the MAC to ignore RXDATA for one pclk cycle. When High, this indicates that RXDATA should be used. When Low, this indicates the RXDATA should be ignored for one pclk cycle. Gen3 and Gen4 only. Per lane.
phy_rxstart_block[1:0]	I	2	This signal allows the PHY to tell the MAC the starting byte for a 128b block. • 00b: Data with no start • 01b: A block starts at lower 32 bits • 10b: A block starts at upper 32 bits, inactive when operating at Gen3 speed • 11b: Lower 32 bits has valid data and upper 32 bits are invalid, inactive when operating at Gen3 speed. Gen3 and Gen4 only per lane.
phy_rxsync_header[1:0]	I	2	Provides the sync header for the MAC to use the next 128b block. The MAC reads this value when the RXSYNC_BLOCK is asserted. Gen3 and Gen4 only. Per lane.

Command Signals

Table 38: **Command Signals**

Name	I/O	Width	Description
phy_txdetectrx	O	1	Tells the PHY to perform receiver detection when this signal is High and POWERDOWN is in P1 low power state. Receiver detection is complete when phystatus asserts for one pclk cycle. The status of receiver detection is indicated in rxstatus when phystatus is High for one pclk cycle. • rxstatus = 000b: Receiver not present • rxstatus = 001b: Receiver present
phy_txelecidle	O	1	Forces the tx[p/n] to electrical idle when this signal is High. During electrical idle, tx[p/n] are driven to the DC common mode voltage. Per lane.
phy_txcompliance	O	1	Sets the running disparity to negative when this signal is logic High. Used when transmitting the PCIe compliance pattern. Per lane.
phy_rxpolarity	O	1	Requests the PHY to perform polarity inversion on the received data when this signal is High. Per lane.

Table 38: **Command Signals** *(cont'd)*

Name	I/O	Width	Description
phy_powerdown[1:0]	O	2	Requests PHY to enter power saving state or return to normal power state. Power management is complete when PHYSTATUS asserts for one PCLK cycle. • 00b: P0, normal operation. • 01b: P0s, power saving state with low recovery time latency. • 10b: P1, power saving state with longer recovery time latency. • 11b: P2, lowest power state. P2 is not supported.
phy_rate[1:0]	O	2	Requests the PHY to perform a dynamic rate change. Rate change is complete when PHYSTATUS asserts for one PCLK cycle. rxvalid, rxdata, and rxstatus must be ignored while the PHY is in rate change. • 00b: Gen1 • 01b: Gen2 • 10b: Gen3 In the simulation mode (PHY_SIM_EN = TRUE), PHY status assertion takes about 45 μs for Gen3 speed change.

Status Signals

Table 39: **Status Signals**

Name	I/O	Width	Description
phy_rxvalid	I	1	Indicates symbol lock and valid data on rxdata when High. This signal must be ignored during reset and rate change. Per lane. Gen1 and Gen2 only.
phy_phystatus	I	1	Used to communicate completion of several PIPE operations including reset, receiver detection, power management, and rate change. Except for reset, this signal indicates done when asserted for one pclk cycle. This signal is held High and asynchronous during reset. In error situations, such as PHY not responding with PHYSTATUS, the MAC should perform the necessary error recovery. Per lane.
phy_phystatus_rst	I	1	Similar to phystatus, except this port is used to communicate completion of reset only. This signal is High immediately upon reset. After the PHY and GT resets are complete, this signal transitions from High to Low.
phy_rxelecidle	I	1	RXELECIDLE = High indicates RX electrical idle detected. Gen1 and Gen2 only. Per lane.

Table 39: **Status Signals** *(cont'd)*

Name	I/O	Width	Description
phy_rxstatus[2:0]	I	3	Encodes RX status and error codes for the RX data. Per lane. • 000b: Received data OK • 001b: 1 SKP added • 010b: 1 SKP removed • 011b: Receiver detected • 100b: 8b/10b (Gen1/Gen2) or 128b/130b (Gen3) decode error • 101b: Elastic buffer overflow • 110b: Elastic buffer underflow • 111b: Receive disparity error (Gen1/Gen2)
phy_ready	I	1	Indicates Master Lane PHY GT is ready.

TX Driver Signals for Gen1 and Gen2

Table 40: **TX Driver Signals for Gen1 and Gen2**

Name	I/O	Width	Description
phy_txmargin[2:0]	O	3	Selects TX voltage levels. The recommendation is to set this port to 000b for the normal operating voltage range. • 000b: Programmable (default) • 001b: Programmable • 010b: Programmable • 011b: Programmable • 100b: Programmable • 101b: Programmable • 110b: Programmable • 111b: Programmable
phy_txswing	O	1	Controls TX voltage swing level. Gen1 and Gen2 only. • 0b: Full swing (default) • 1b: Low swing
phy_txdeemph	O	1	Selects TX de-emphasis. Gen1 and Gen2 only. • 0b: -6.0 dB de-emphasis • 1b: -3.5 dB de-emphasis (default)

Configuration Space

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

- **Legacy PCI v3.0 Type 0/1 Configuration Space Header:**

- Type 0 Configuration Space Header used by Endpoint applications (see Table 41: PCI Config Space Header (Type 0 and 1))
- Type 1 Configuration Space Header used by Root Port applications (see Table 41: PCI Config Space Header (Type 0 and 1))

- **Legacy Extended Capability Items:**
  - PCIe Capability Item
  - Power Management Capability Item
  - Message Signaled Interrupt (MSI) Capability Item
  - MSI-X Capability Item (optional)

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 `600h` when extended large is selected, and `E00h` when extended small is selected. If you choose to implement registers in this space, you can select the starting location of this space, and this space must be implemented in the user application.

Table 41: **PCI Config Space Header (Type 0 and 1)**

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

Table 42: **PCI Express Config Space**

Byte Offset (DW Offset)	Register (Endpoint)				Register (Root/DS Port)	
40h (10h)	PM Capability		NxtCap	PM Cap ID	*same as Endpoint*	
44h (11h)	Data	BSE	PMCSR			
48h (12h)	MSI Control		NxtCap	MSI Cap ID		
4Ch (13h)	Message Address (Lower)					
50h (14h)	Message Address (Upper)					
54h (15h)	Reserved		Message Data			
58h (16h)	Mask Bits					
5Ch (17h)	Pending Bits					
60h (18h)	MSIX Control		NxtCap	MSIX Cap ID	Reserved	
64h (19h)	Table Offset			Table BIR	Reserved	
68h (1Ah)	PBA Offset			PBA BIR	Reserved	
6Ch (1Bh)	Reserved				Reserved	
70h (1Ch)	PCIE Capability		NxtCap	PCIE Cap ID	*same as Endpoint*	
74h (1Dh)	Device Capabilities					
78h (1Eh)	Device Status		Device Control			
7Ch (1Fh)	Link Capabilities					
80h (20h)	Link Status		Link Control			
84h (21h)	Reserved				Slot Capabilities	
88h (22h)	Reserved				Slot Status	Slot Control
8Ch (23h)	Reserved				Root Capabilities[1]	Root Control[1]
90h (24h)	Reserved				Root Status[1]	
94h (25h)	Device Capabilities 2				*same as Endpoint*	
98h (26h)	Device Status 2		Device Control 2			
9Ch (27h)	Link Capabilities 2					
A0h (28h)	Link Status 2		Link Control 2			
A4-FCh	Unimplemented Configuration Space (Returns 00000000h)					

Notes:

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

Table 43: **PCIe Capability Order with PCIE4 Capabilities**

PF0	PF1-3	VF	Start Address	PF0 Next Pointer
Legacy PCI CSH	Legacy PCI CSH	Legacy PCI CSH	0x00	0x40
PM	PM	-	0x40	0x48
MSI	MSI	-	0x48	0x60
MSI-X	MSI-X	MSI-X	0x60	0x70
PCIE	PCIE	PCIE	0x70	0x0

Table 43: **PCIe Capability Order with PCIE4 Capabilities** *(cont'd)*

PF0	PF1-3	VF	Start Address	PF0 Next Pointer
Extend	Extend		0xB0	

Table 44: **PCI Express Extended Configuration Space: Byte Offset 100h (40h) to 32Ch**

Byte Offset (DW Offset)	Register (Endpoint)			Register (Root Port)
100h (40h)	Nxt Cap	Cap Ver	AER Ext Cap	*same as Endpoint*
104h (41h)	Uncorrectable Error Status Register			
108h (42h)	Uncorrectable Error Mask Register			
10Ch (43h)	Uncorrectable Error Severity Register			
110h (44h)	Correctable Error Status Register			
114h (45h)	Correctable Error Mask Register			
118h (46h)	Advanced Error Cap. & Control Register			
11Ch (47h)	Header Log Register 1			
120h (48h)	Header Log Register 2			
124h (49h)	Header Log Register 3			
128h (4Ah)	Header Log Register 4			
12Ch (4Bh)	Reserved			Root Error Command Register
130h (4Ch)	Reserved			Root Error Status Register
134h (4Dh)	Reserved			Error Source ID Register

Table 44: **PCI Express Extended Configuration Space: Byte Offset 100h (40h) to 32Ch** *(cont'd)*

Byte Offset (DW Offset)	Register (Endpoint)				Register (Root Port)
140h (50h)	Nxt Cap	Cap Ver	SR-IOV Ext Cap		Reserved
144h (51h)	Capability Register				
148h (52h)	SR-IOV Status		Control		
14Ch (53h)	Total VFs		Initial VFs		
150h (54h)	Func Dep Link		Number VFs		
154h (55h)	VF Stride		First VF Offset		
158h (56h)	VF Device ID		Reserved		
15Ch (57h)	Supported Page Sizes				
160h (58h)	System Page Size				
164h (59h)	VF Base Address Register 0				
168h (5Ah)	VF Base Address Register 1				
16Ch (5Bh)	VF Base Address Register 2				
170h (5Ch)	VF Base Address Register 3				
174h (5Dh)	VF Base Address Register 4				
178h (5Eh)	VF Base Address Register 5				
180h (60h)	Nxt Cap	Cap Ver	ARI Ext Cap		
184h (61h)	Control		NxtFn	FnGrp	
188h - 19Ch	Reserved				
1A0h (68h)	Nxt Cap	Cap Ver	DSN Ext Cap		
1A4h (69h)	Device Serial Number (1st)				
1A8h (6Ah)	Device Serial Number (2nd)				
1ACh - 1BCh	Reserved				
1C0h (70h)	Nxt Cap	Cap Ver	2nd PCIE Ext Cap		same as Endpoint
1C4h (71h)	Lane Control				
1C8h (72h)	Reserved		Lane Error Status		
1CCh (73h)	Lane 1 Eq Ctrl Reg		Lane 0 Eq Ctrl Reg		
1D0h (74h)	Lane 3 Eq Ctrl Reg		Lane 2 Eq Ctrl Reg		
1D4h (75h)	Lane 5 Eq Ctrl Reg		Lane 4 Eq Ctrl Reg		
1D8h (76h)	Lane 7 Eq Ctrl Reg		Lane 6 Eq Ctrl Reg		
1DCh (77h)	Lane 9 Eq Ctrl Reg		Lane 8 Eq Ctrl Reg		
1E0h (78h)	Lane 11 Eq Ctrl Reg		Lane 10 Eq Ctrl Reg		
1E4h (79h)	Lane 13 Eq Ctrl Reg		Lane 12 Eq Ctrl Reg		
1E8h (7Ah)	Lane 15 Eq Ctrl Reg		Lane 14 Eq Ctrl Reg		
1ECh (7Bh)	Reserved				
1F0h (7Ch)	Nxt Cap	Cap Ver	VC Ext Cap		
1F4h (7Dh)	Port VC Capability Register 1				
1F8h (7Eh)	Port VC Capability Register 2				

Table 44: **PCI Express Extended Configuration Space: Byte Offset 100h (40h) to 32Ch** *(cont'd)*

Byte Offset (DW Offset)	Register (Endpoint)	Register (Root Port)
1FCh (7Fh)	Port VC Status	Reserved
200h(80h)	VC Resource Capability Register 0	
204h(81h)	VC Resource Control Register 0	
208h(82h)	VC Resource Stat 0	
20Ch(83h)	VC Resource Capability Register 1	
210h (84h)	VC Resource Control Register 1	
214h (85h)	VC Resource Stat 1	
218h (86h)	Reserved	
21Ch (87h)		

Table 45: **PCI Express Extended Configuration Space: Byte Offset 330h (CCh) to FFCh**

Byte Offset (DW Offset)	Register (Endpoint)			Register (Root Port)		
330h (CCh)	Reserved			Nxt Cap	Cap Ver	Loopback VSEC
334h (CDh)				Loopback Header		
338h (CEh)				Loopback Control		
33Ch (CFh)				Loopback Status		
340h (D0h)				Error Count 1		
344h (D1h)				Error Count 2		
348h (D2h)				Error Count 3		
34Ch (D3h)				Error Count 4		
380h (E0h)	Nxt Cap	Cap Ver	ATS Ext Cap	Reserved		
384h (E1h)	Control Reg		Capability Reg			
388-38Ch	Reserved					
390h (E4h)	Nxt Cap	Cap Ver	PRI Ext Cap			
394h (E5h)	Status Reg		Control Reg			
398h (E6h)	Outstanding Page Request Capacity					
39Ch (E7h)	Outstanding Page Request Allocation					

Table 45: **PCI Express Extended Configuration Space: Byte Offset 330h (CCh) to FFCh** *(cont'd)*

Byte Offset (DW Offset)	Register (Endpoint)			Register (Root Port)
3A0h (E8h)	Nxt Cap	Cap Ver	DL Feature Ext Cap	
3A4h (E9h)	Capabilities Register			
3A8h (EAh)	Status Register			
3ACh (EBh)	1 DW Reserved			
3B0h (ECh)	Nxt Cap	Cap Ver	16 GT/s Capability	
3B4h (EDh)	Capabilities Register			
3B8h (EEh)	Control Register			
3BCh (EFh)	Status Register			
3C0h (F0h)	Local Data Parity Mismatch Register			
3C4h (F1h)	First Retimer Data Parity Mismatch Status Register			
3C8h (F2h)	Second Retimer Data Parity Mismatch Status Register			
3CCh (F3h)	Lane 3-0 Eq Control Register			
3D0h (F4h)	Lane 7-4 Eq Control Register			
3D4h (F5h)	Lane 11-8 Eq Control Register			
3D8h (F6h)	Lane 15-12 Eq Control Register			
3DCh- 3FCh	5 DW Reserved			
400h (100h)	Nxt Cap	Cap Ver	Margining Ext Cap	
404h (101h)	Port Status		Port Capabilities	*same as Endpoint*
408h (102h)	Lane 0 Status		Lane 0 Control	
40Ch (103h)	Lane 1 Status		Lane 1 Control	
410h (104h)	Lane 2 Status		Lane 2 Control	
414h (105h)	Lane 3 Status		Lane 3 Control	
418h (106h)	Lane 4 Status		Lane 4 Control	
41Ch (107h)	Lane 5 Status		Lane 5 Control	
420h (108h)	Lane 6 Status		Lane 6 Control	
424h (109h)	Lane 7 Status		Lane 7 Control	
428h (10Ah)	Lane 8 Status		Lane 8 Control	
42Ch (10Bh)	Lane 9 Status		Lane 9 Control	
430h (10Ch)	Lane 10 Status		Lane 10 Control	
434h (10Dh)	Lane 11 Status		Lane 11 Control	
438h (10Eh)	Lane 12 Status		Lane 12 Control	
43Ch (10Fh)	Lane 13 Status		Lane 13 Control	
440h (110h)	Lane 14 Status		Lane 14 Control	
444h (111h)	Lane 15 Status		Lane 15 Control	
448h-44Ch	2 DW Reserved			
450h (114h)	Nxt Cap	Cap Ver	ACS Ext Cap	

Table 45: **PCI Express Extended Configuration Space: Byte Offset 330h (CCh) to FFCh** *(cont'd)*

Byte Offset (DW Offset)	Register (Endpoint)		Register (Root Port)
454h (115h)	ACS Control	ACS Capabilties	*same as Endpoint*
458h (116h)	Egress Control Vector		
45Ch	1 DW Reserved		
460h – 4BCh	PL 32 GT/s / Reserved		
4C0h – 500h	17 DW Reserved		
504h (141h)	VC Arb Table		
544h -5ECh	44 DW Reserved		
5F0h (17Ch)	PASID		
600h-DE0h	Extend-Large	T& PDVSEC	
DE4h-DFCh		PER Msg	
E00h-FFCh		Extend-Small	

Designing with the Core

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

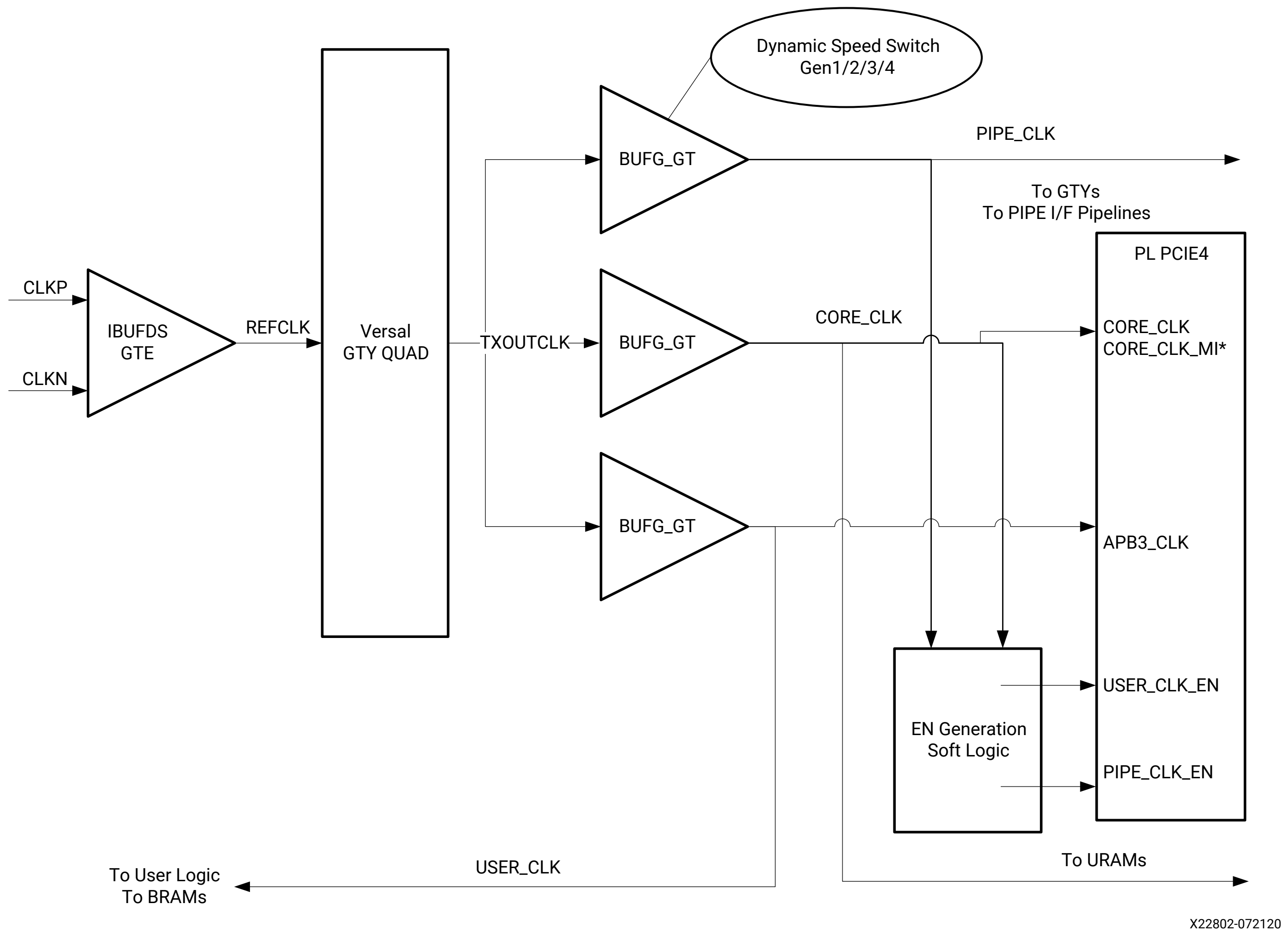
Clocking

The core requires a 100, 125, or 250 MHz reference clock input.

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_GT`s are used to generate the core clocks. These clocks are all driven from the `TXOUTCLK` pin which is a derived clock from `GTREFCLK0` through a RPLL. LCPLL is only provided to the GT PCS/PMA block while `TXOUTCLK` continues to be derived from a RPLL.
- The source of the Versal™ GTY reference clock must come directly from `IBUFDS_GTE5`.
- To use the reference clock for ACAP general interconnect, another `BUFG_GT` must be used.
- The PIPE clock module is not part of the core. It is visible in the Versal PCI Express PHY IP generated as part of this subsystem, and is available in the PCI Express PHY example design.

Figure 4: **Clocking Architecture**



All PCIe clocks (`pipe_clk`, `core_clk`, `user_clk`, and `mcap_clk`) are all driven by `BUFG_GT` sourced from the `TXOUTCLK` pin. These clocks are derived clock from `GTREFCLK0` through a RPLL. All user interface signals of the core are timed with respect to the same clock (`user_clk`) which can have a frequency of 62.5, 125, or 250 MHz depending on the link speed and width configured (see the previous figure).

In a typical PCI Express® solution, the PCI Express reference clock is a spread spectrum clock (SSC), provided at 100 MHz. In most commercial PCI Express systems, SSC cannot be disabled. For more information regarding SSC and PCI Express, see *PCI Express Base Specification 4.0* (https://www.pcisig.com/specifications).

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

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

Note: The clocking diagrams show high-level representations of the board layout. Ensure that coupling, termination, and details are correct when laying out a board.

Figure 5: **Embedded System Using 100 MHz Reference Clock**

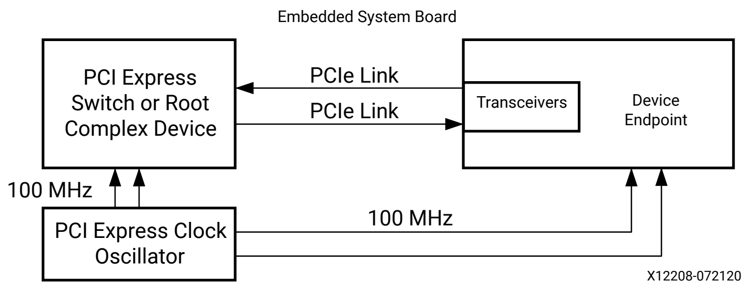
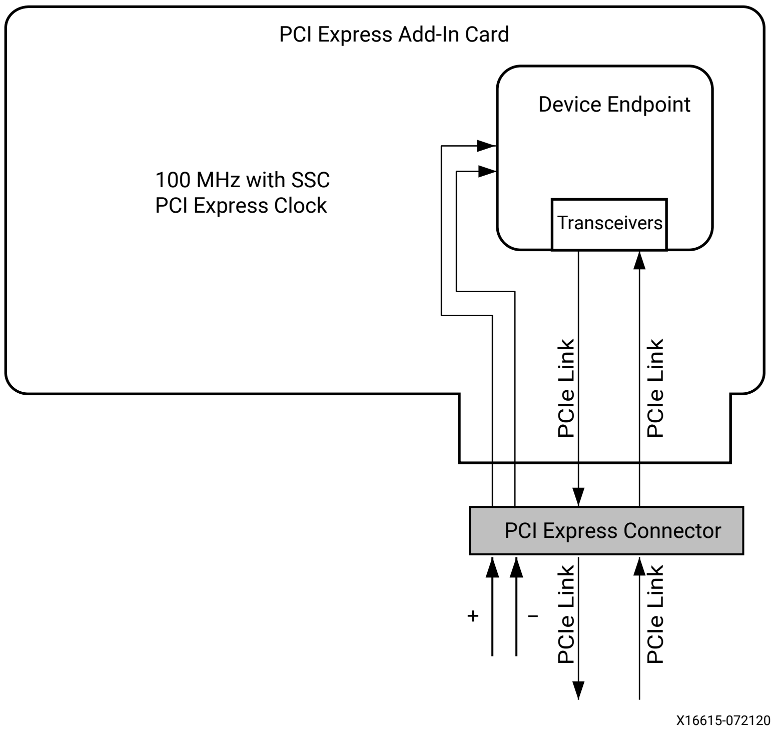


Figure 6: **Open System Add-In Card Using 100 MHz Reference Clock**



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

- This results in a logic driven CE (rather than VCC) for the `BUFG_GT` that is driven by `IBUFDS_GTE5` (PCIe ref clock).

- Before this change in CE, if you had another (parallel) `BUFG_GT` connected to the `IBUFDS_GTE5` with CE driven by VCC, the `BUFG_GT_SYNC` inserted by opt_design/MLO could drive both `BUFG_GT`s.
- 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_GTE5` driven`BUFG_GT`s, the router does not know how to handle the second `BUFG_GT_SYNC` and does not route the `IBUFDS_GTE5/ODIV2` driven clock net.
- You must ensure that the `BUFG_GT`s driven by the `IBUFDS_GTE5` have the same CE/CLR pins.

Resets

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

RECOMMENDED: *Recommendation for reset use:*

- If your board is designed to have the same PCIe edge connectors to operate with CPM and PL PCIe, Xilinx recommends using PS reset using CIPS. Please refer to Versal ACAP CPM mode for PCIe PG for more information.
- If your board design is specifically planned to just use PL-PCIE in the PCIe edge connectors, then any PL reset pin can be used.

Four reset events can occur in PCI Express:

- **Cold Reset:** A Fundamental Reset that occurs at the application of power. The `sys_reset` signal is asserted to cause the cold reset of the core.
- **Warm Reset:** A Fundamental Reset triggered by hardware without the removal and reapplication of power. The `sys_reset` signal is asserted to cause the warm reset to the core.
- **Hot Reset:** In-band propagation of a reset across the PCI Express Link through the protocol, resetting the entire Endpoint device. In this case, `sys_reset` is not used. In the case of Hot Reset, the `cfg_hot_reset_out` signal is asserted to indicate the source of the reset.

- **Function-Level Reset:** In-band propagation of a reset across the PCI Express Link through the protocol, resetting only a specific function. In this case, the core asserts the bit of either `cfg_flr_in_process` and/or `cfg_vf_flr_in_process` that corresponds to the function being reset. Logic associated with the function being reset must assert the corresponding bit of `cfg_flr_done` or `cfg_vf_flr_done` to indicate it has completed the reset process.

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

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

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

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

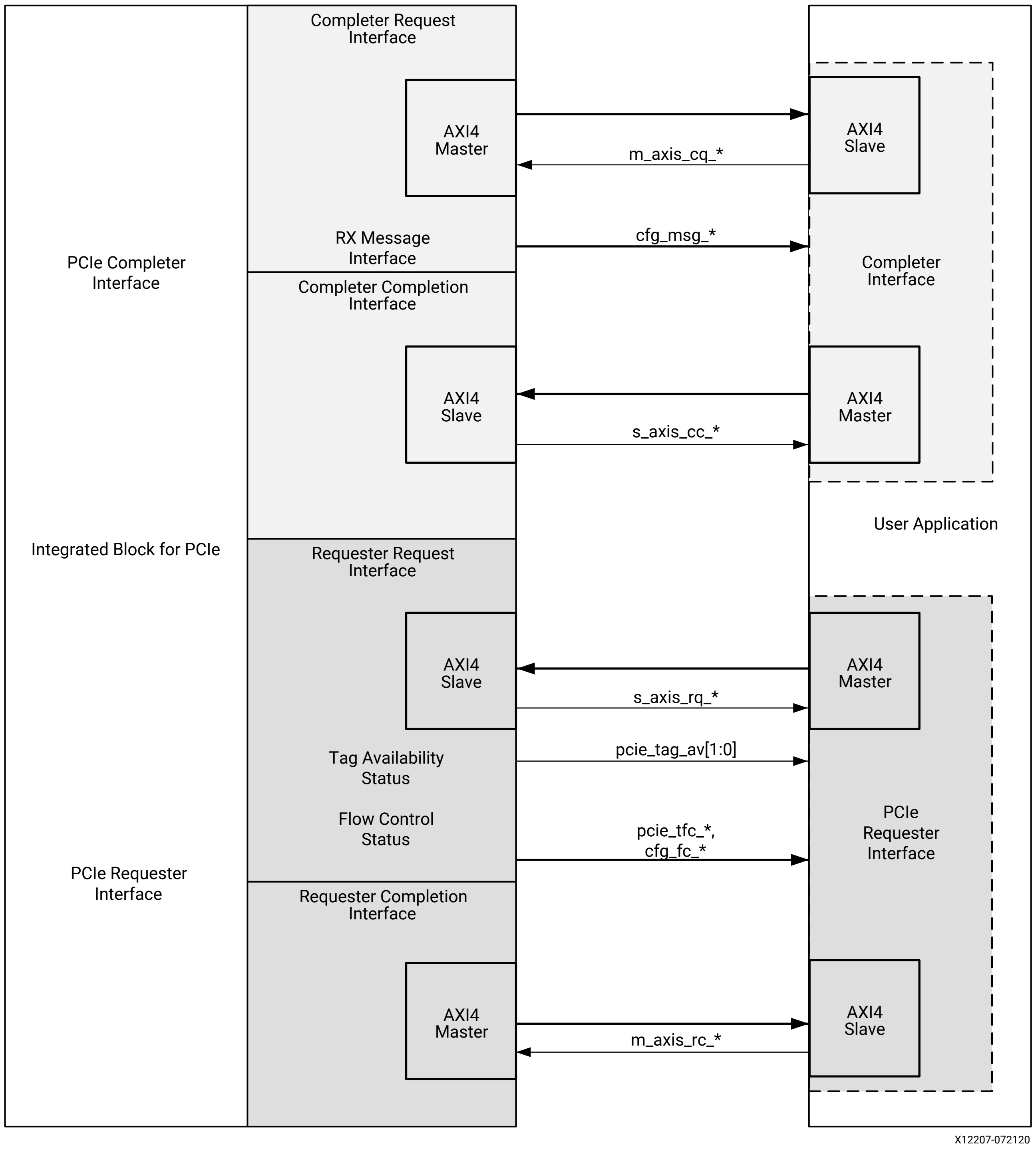
AXI4-Stream Interface Description

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

Feature Overview

The following figure illustrates the user interface of the core.

Figure 7: **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 user application receives completions from the link (in response to the user application requests as PCIe requester).

Each of the four interfaces is based on the AMBA® AXI4-Stream Protocol Specification. The width of these interfaces can be configured as 64, 128, 256, or 512 bits, and the user clock frequencies can be selected as 62.5, 125, or 250 MHz, depending on the number of lanes and data rate you choose.

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

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

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

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

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

Table 46: **Clock Frequencies and Interface Widths Supported For Various Configurations** *(cont'd)*

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

Data Alignment Options

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

64/128/256-bit interface:

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

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

   ```
   n = A mod w
   ```

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

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

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

512-bit interface:

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

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

   ```
   n = (S + D + (A mod 4)) mod 64
   ```

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

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

   ```
   n = (S + 16 + (A mod 16)) mod 64
   ```

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

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

   - **CQ Interface:** For data transferred from the core to the user application over the CQ interface, the address bits used for alignment are the lower address specified in the descriptor, which is the starting address of the data block in user memory.
   - **CC Interface:** For Completion data transferred from the user application to the core over the CC interface, the alignment is based on address bits supplied by the user in the descriptor.

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

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

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

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

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

Straddle Option on CQ, CC, and RQ Interfaces

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

Straddle Option on RC Interface

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

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

- Restrict the number of outstanding Non Posted requests, so as to keep the total number of completions received less than 64 and within the completion of the FIFO size selected, or
- Use the RC interface straddle option. See the waveform figures for 256 bits (Figure 60: Transfer of Completion TLPs on the Requester Completion Interface with the Straddle Option Enabled) and 512 bits (Figure 91: Transfer of Completion TLPs on the Requester Completion Interface with the Straddle Option Enabled), respectively showing this option.

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

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

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

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

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

Receive Transaction Ordering

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

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

  The integrated block delivers a Non-Posted request to the user application only when the available credit is non-zero. The integrated block continues to deliver Posted requests while the delivery of Non-Posted requests has been paused for lack of credit. When no back pressure is applied by the credit mechanism for the delivery of Non-Posted requests, the integrated block delivers Posted and Non-Posted requests in the same order as received from the link. For more information on controlling the flow of Non-Posted requests, see Selective Flow Control for Non-Posted Requests.
- PCIe ordering requires that a completion TLP not be allowed to pass a Posted request, except in the following cases:
  - Completions with the Relaxed Ordering attribute bit set can pass Posted requests.
  - Completions with the ID-based ordering bit set can pass a Posted request if the Completer ID is different from the Posted Requester ID.

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

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

Table 48: **Receive Ordering Rules**

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

Transmit Transaction Ordering

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

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

64/128/256-Bit Completer Interface

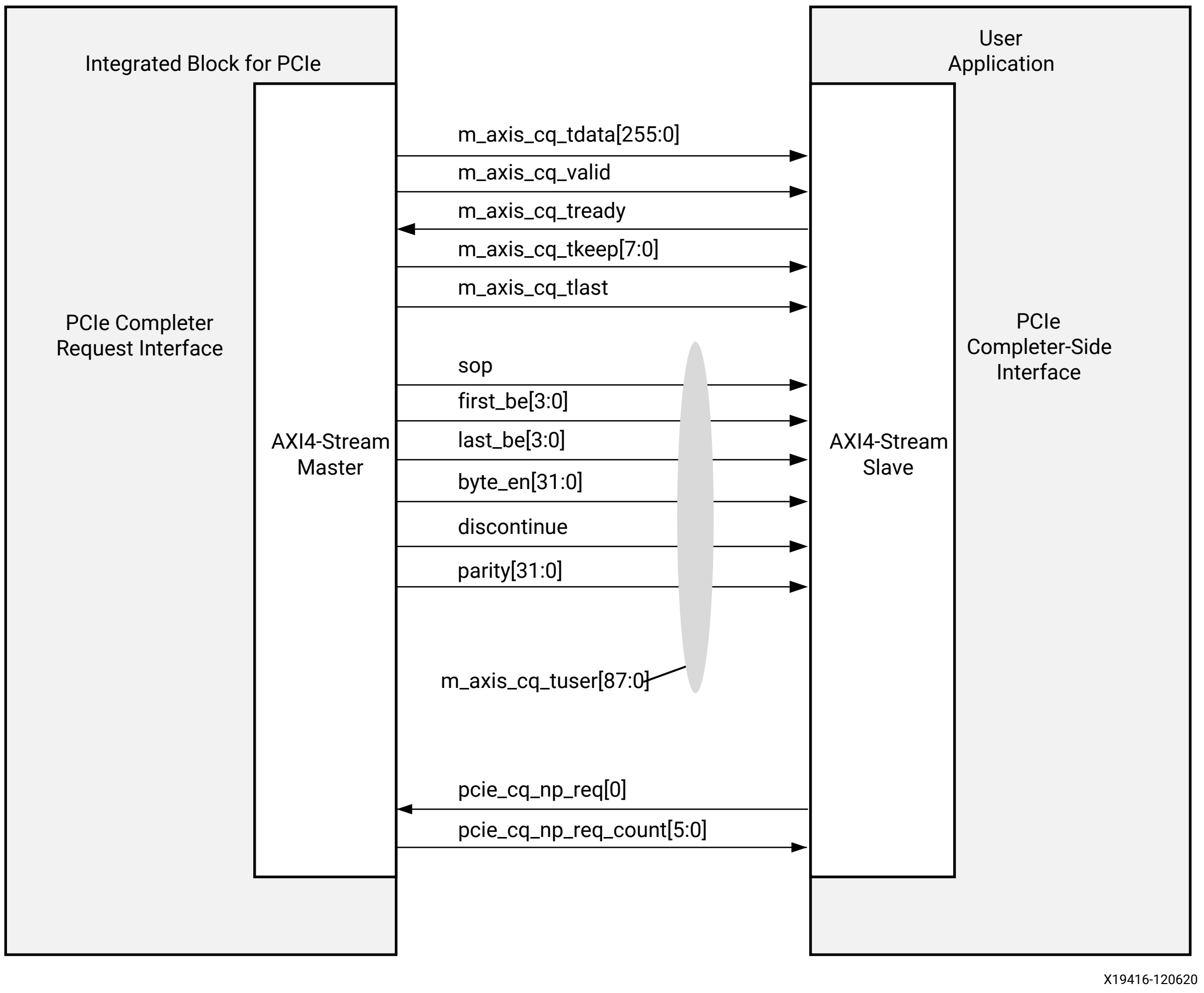
This section describes the operation of the user interfaces of the core for 64/128/256-bit interfaces.

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

Completer Request Interface Operation

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

Figure 8: **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 $n = (16 + A \bmod 4) \bmod w$, where A is the byte-level starting address of the data block being transferred, and w is the width of the interface in bytes.

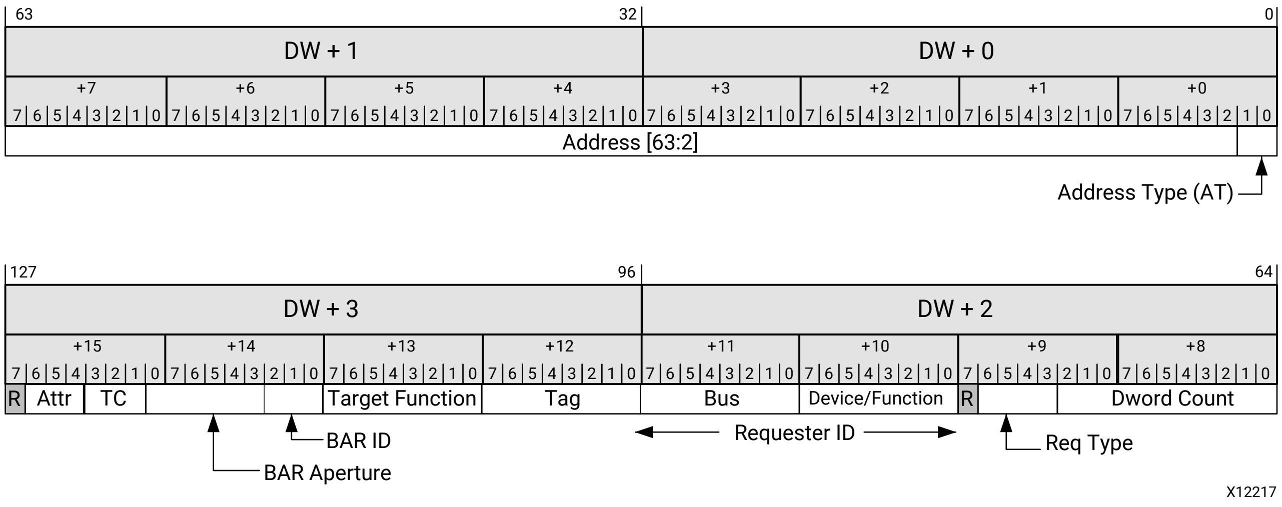
In the address-aligned mode, the data always starts in a new beat after the descriptor has ended, and its first valid byte is on lane $n = A \bmod w$, where w is the width of the interface in bytes. For memory, I/O, and Atomic Operation requests, address A is the address contained in the request. For messages, the address is always taken as 0 for the purpose of determining the alignment of its payload.

Completer Request Descriptor Formats

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

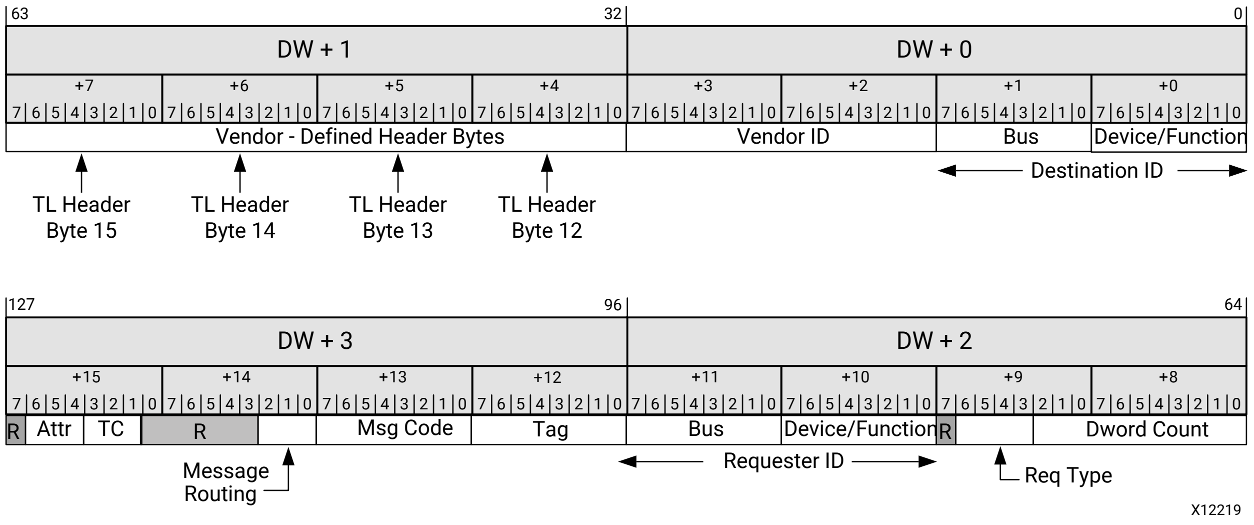
The format of the following figure applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request.

Figure 9: **Completer Request Descriptor Format for Memory, I/O, and Atomic Op Requests**



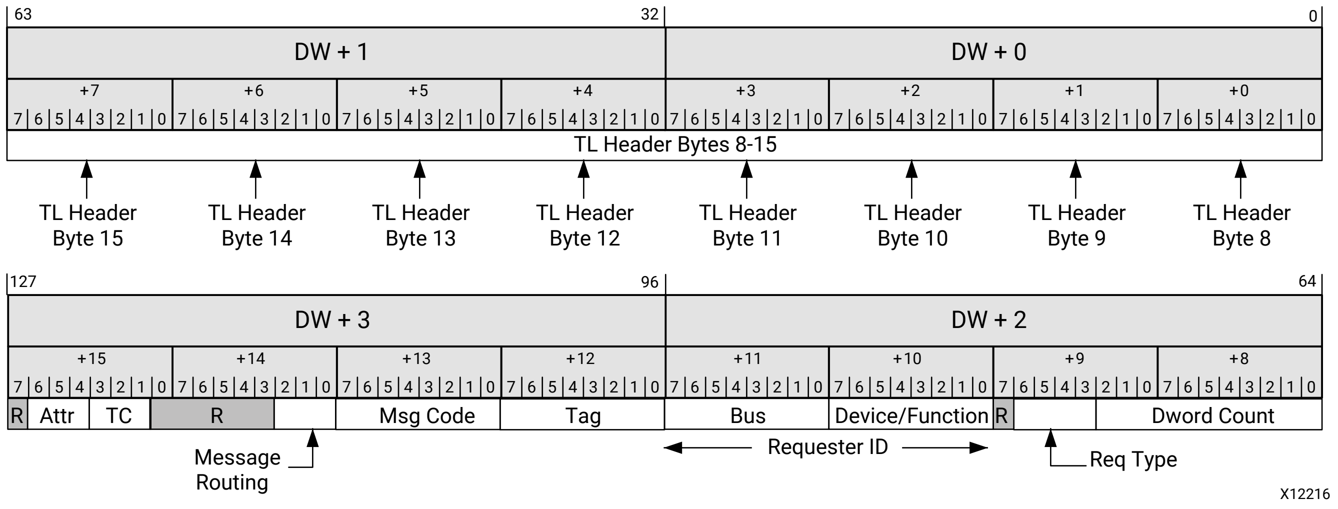
The format of the following figure is used for Vendor-Defined Messages (Type 0 or Type 1) only.

Figure 10: **Completer Request Descriptor Format for Vendor-Defined Messages**



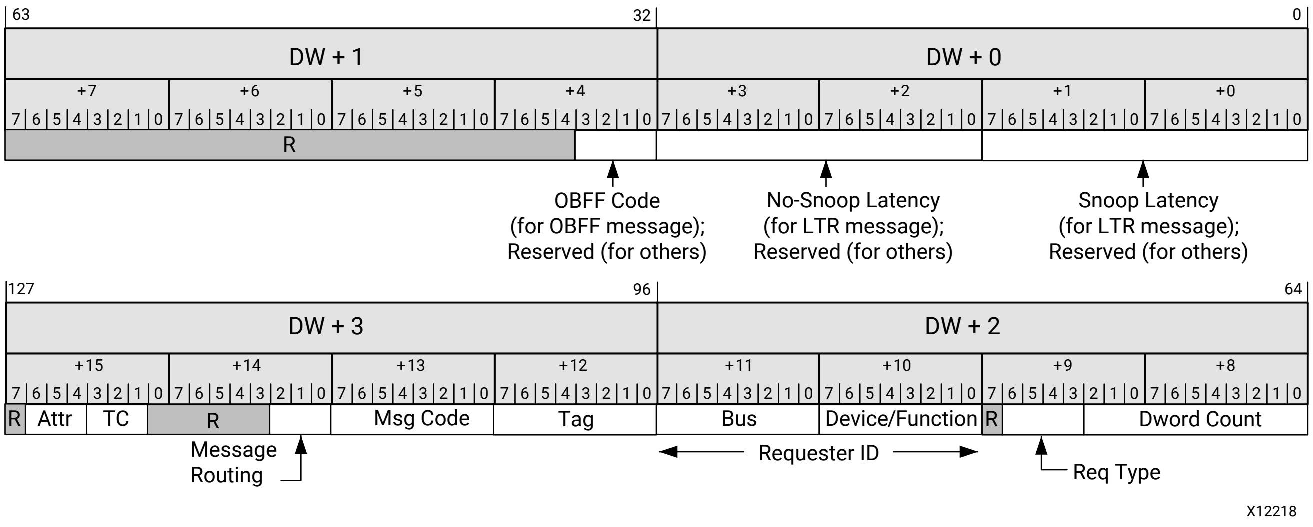
The format of the following figure is used for all ATS messages (Invalid Request, Invalid Completion, Page Request, PRG Response).

Figure 11: **Completer Request Descriptor Format for ATS Messages**



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

Figure 12: **Completer Request Descriptor Format for All Other Messages**



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

Table 49: **Completer Request Descriptor Fields**

Bit Index	Field Name	Description
1:0	Address Type	This field is defined for memory transactions and Atomic Operations only. It contains the AT bits extracted from the TL header of the request. • 00: Address in the request is untranslated • 01: Transaction is a Translation Request • 10: Address in the request is a translated address • 11: Reserved
63:2	Address	This field applies to memory, I/O, and Atomic Op requests. It provides the address from the TLP header. This is the address of the first Dword referenced by the request. The First_BE bits from m_axis_cq_tuser must be used to determine the byte-level address. When the transaction specifies a 32-bit address, bits [63:32] of this field are 0.
74:64	Dword Count	These 11 bits indicate the size of the block (in Dwords) to be read or written (for messages, size of the message payload). Its range is 0 - 256 Dwords. For I/O accesses, the Dword count is always 1. For a zero length memory read/write request, the Dword count is 1, with the First_BE bits set to all 0s.
78:75	Request Type	Identifies the transaction type. The transaction types and their encodings are listed in Table 50: Transaction Types.
95:80	Requester ID	PCI Requester ID associated with the request. With legacy interpretation of RIDs, these 16 bits are divided into an 8-bit bus number [95:88], 5-bit device number [87:83], and 3-bit Function number [82:80]. When ARI is enabled, bits [95:88] carry the 8-bit bus number and [87:80] provide the Function number. When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the integrated block with the completion data.

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

Bit Index	Field Name	Description
103:96	Tag	PCIe Tag associated with the request. When the request is a Non-Posted transaction, the user logic must store this field and supply it back to the integrated block with the completion data. This field can be ignored for memory writes and messages.
111:104	Target Function	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 • 2: PF2 • 3: PF3 • 4: VF0 • 5: VF1 • 6: VF2 • 7: VF3
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. In RP mode, BAR ID is always 000. • 001: BAR 1 (VF-BAR 1 for VFs) • 010: BAR 2 (VF-BAR 2 for VFs) • 011: BAR 3 (VF-BAR 3 for VFs) • 100: BAR 4 (VF-BAR 4 for VFs) • 101: BAR 5 (VF-BAR 5 for VFs) • 110: Expansion ROM Access For 64-bit transactions, the BAR number is given as the lower address of the matching pair of BARs (that is, 0, 2, or 4).
120:115	BAR Aperture	This 6-bit field is defined for memory, I/O, and Atomic Op requests only. It provides the aperture setting of the BAR matching the request. This information is useful in determining the bits to be used in addressing its memory or I/O space. For example, a value of 12 indicates that the aperture of the matching BAR is 4K, and the user application can therefore ignore bits [63:12] of the address. For VF BARs, the value provided on this output is based on the memory space consumed by a single VF covered by the BAR.
123:121	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request. When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the integrated block with the completion data.
126:124	Attributes	These bits provide the setting of the Attribute bits associated with the request. Bit 124 is the No Snoop bit and bit 125 is the Relaxed Ordering bit. Bit 126 is the ID-Based Ordering bit, and can be set only for memory requests and messages. When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the integrated block with the completion data.
15:0	Snoop Latency	This field is defined for LTR messages only. It provides the value of the 16-bit Snoop Latency field in the TLP header of the message.
31:16	No-Snoop Latency	This field is defined for LTR messages only. It provides the value of the 16-bit No-Snoop Latency field in the TLP header of the message.

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

Bit Index	Field Name	Description
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. Users should treat a descriptor with unsupported Message Code as UR, and toggle the signal cfg_err_uncor_in to indicate that Non-fatal error is detected.
114:112	Message Routing	This field is defined for all messages. These bits provide the 3-bit Routing field r[2:0] from the TLP header.
15:0	Destination ID	This field applies to Vendor-Defined Messages only. When the message is routed by ID (that is, when the Message Routing field is 010 binary), this field provides the Destination ID of the message.
63:32	Vendor-Defined Header	This field applies to Vendor-Defined Messages only. It contains the bytes extracted from Dword 3 of the TLP header.
63:0	ATS Header	This field is applicable to ATS messages only. It contains the bytes extracted from Dwords 2 and 3 of the TLP header.

Table 50: **Transaction Types**

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

Completer Memory Write Operation

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

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

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

The `m_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.

Figure 13: **Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode, 64-Bit Interface)**

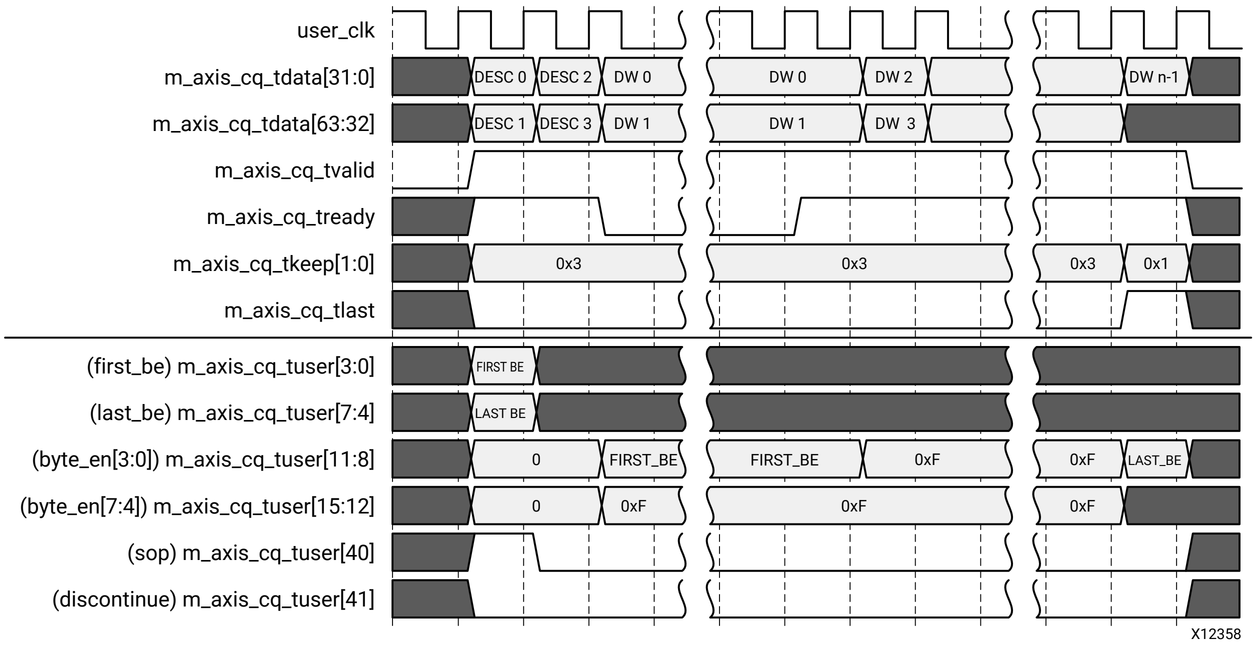


Figure 14: **Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode, 128-Bit Interface)**

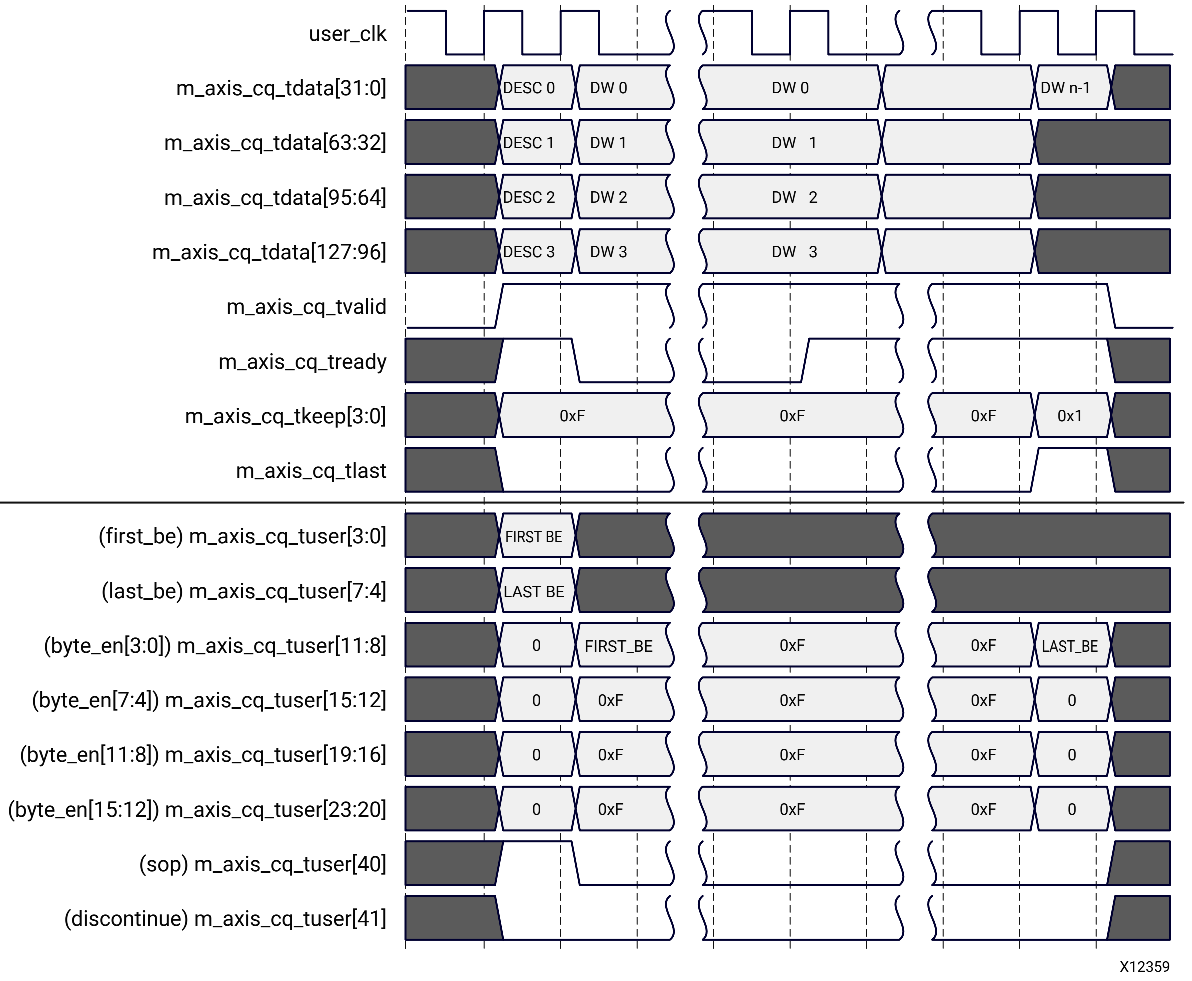
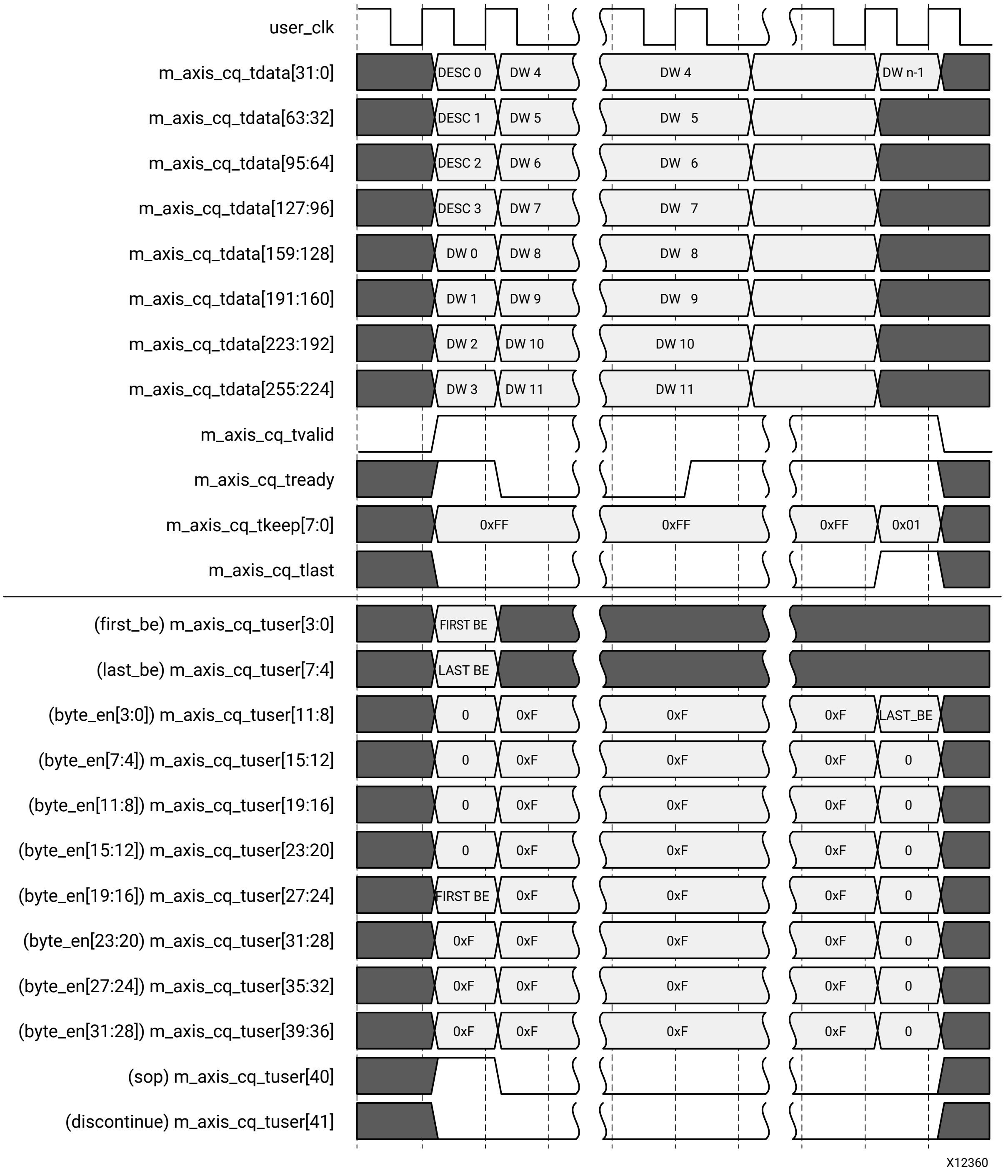

user_clk
m_axis_cq_tdata[31:0]
m_axis_cq_tdata[63:32]
m_axis_cq_tdata[95:64]
m_axis_cq_tdata[127:96]
m_axis_cq_tvalid
m_axis_cq_tready
m_axis_cq_tkeep[3:0]
m_axis_cq_tlast
(first_be) m_axis_cq_tuser[3:0]
(last_be) m_axis_cq_tuser[7:4]
(byte_en[3:0]) m_axis_cq_tuser[11:8]
(byte_en[7:4]) m_axis_cq_tuser[15:12]
(byte_en[11:8]) m_axis_cq_tuser[19:16]
(byte_en[15:12]) m_axis_cq_tuser[23:20]
(sop) m_axis_cq_tuser[40]
(discontinue) m_axis_cq_tuser[41]
DESC 0
DW 0
DW n-1
DESC 1
DW 1
DESC 2
DW 2
DESC 3
DW 3
0xF
0x1
FIRST BE
LAST BE
0
FIRST_BE
LAST_BE
X12359

Figure 15: **Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode, 256-Bit Interface)**


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

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

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

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

Figure 16: **Memory Write Transaction on the Completer Request Interface (Address-Aligned Mode, 64-Bit Interface)**

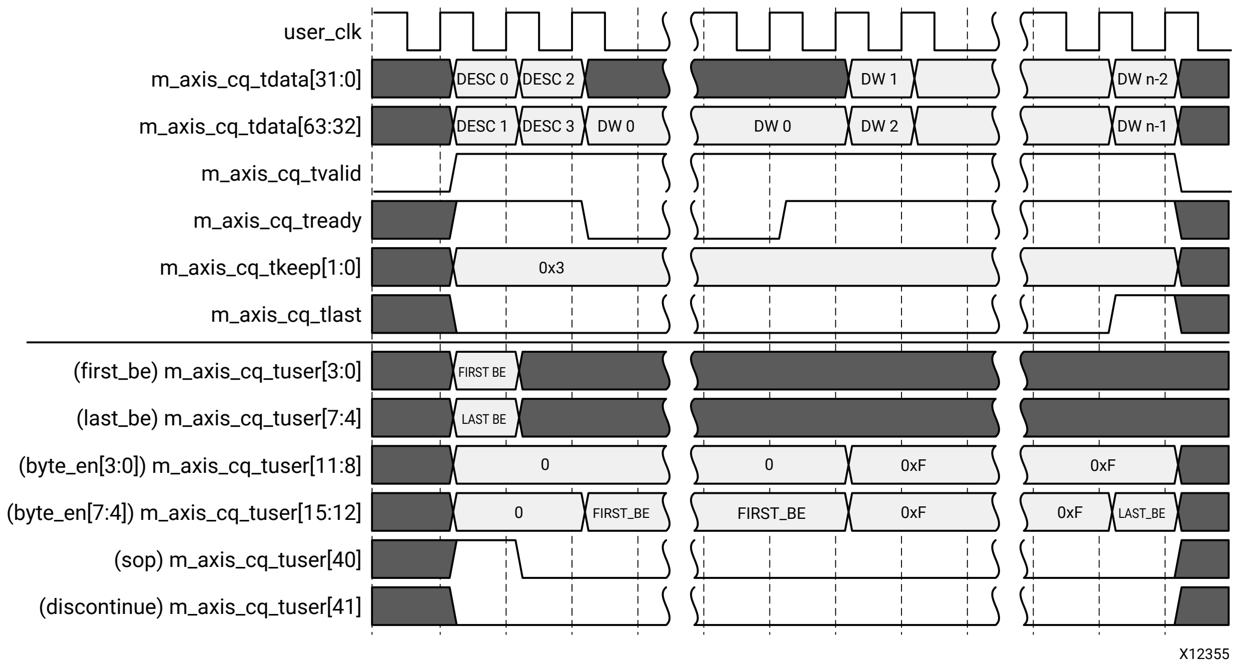


Figure 17: **Memory Write Transaction on the Completer Request Interface (Address-Aligned Mode, 128-Bit Interface)**

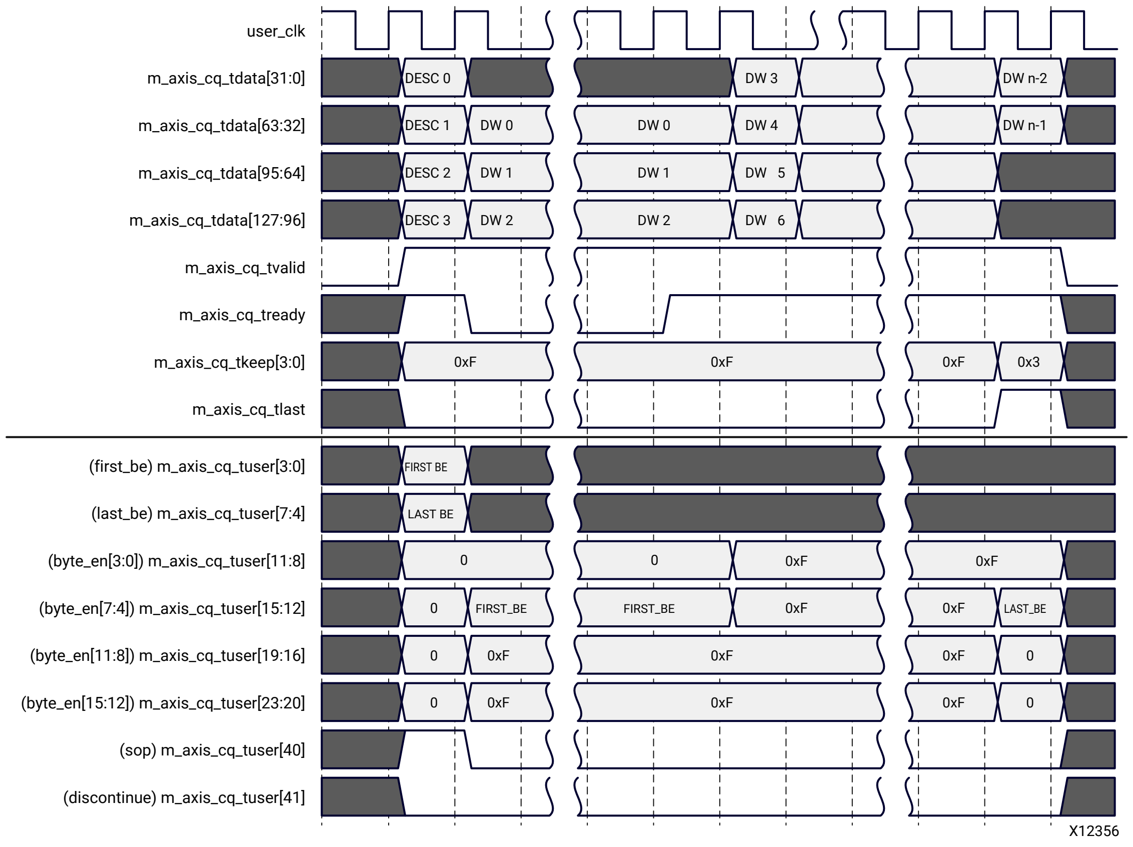
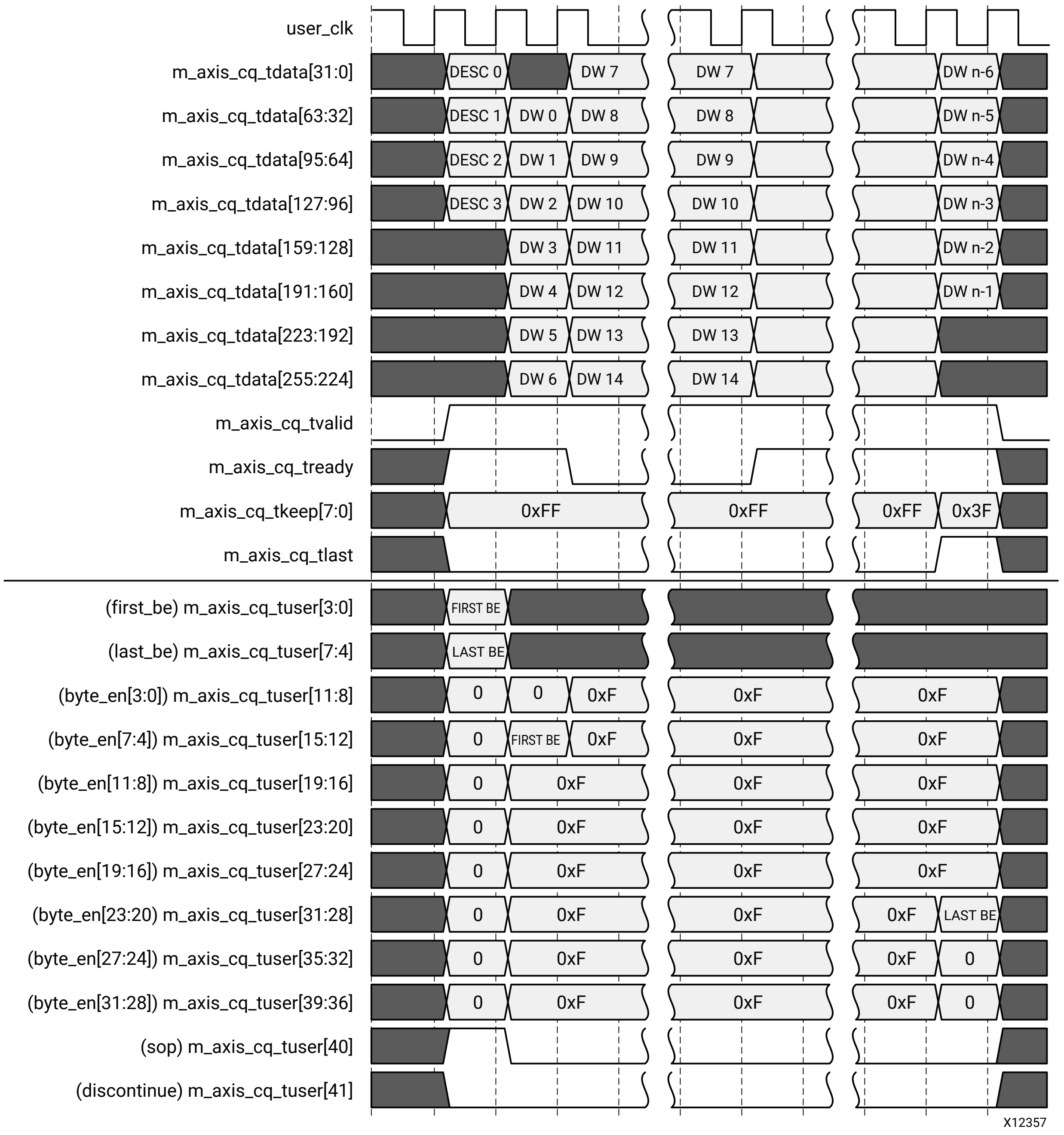


Figure 18: **Memory Write Transaction on the Completer Request Interface (Address-Aligned Mode, 256-Bit Interface)**


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

Send Feedback

Completer Memory Read Operation

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

Figure 19: **Memory Read Transaction on the Completer Request Interface (64-Bit Interface)**

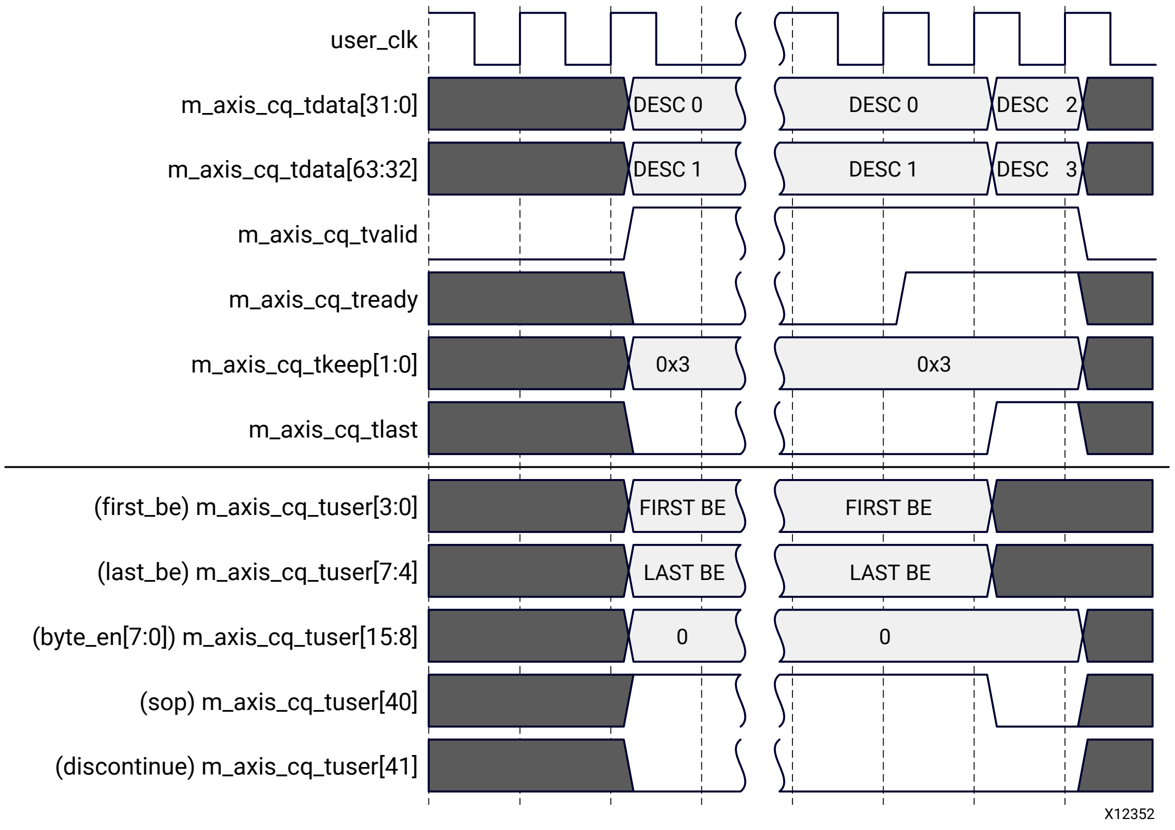


Figure 20: **Memory Read Transaction on the Completer Request Interface (128-Bit Interface)**

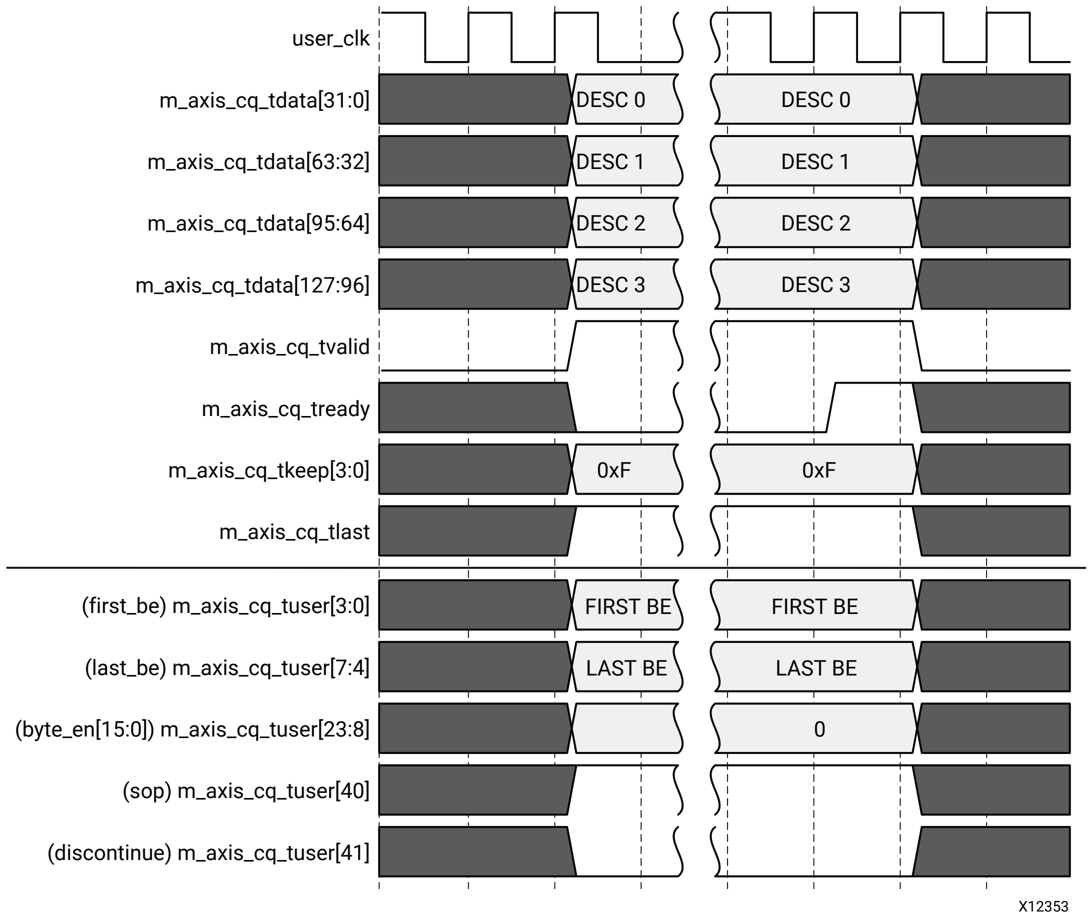
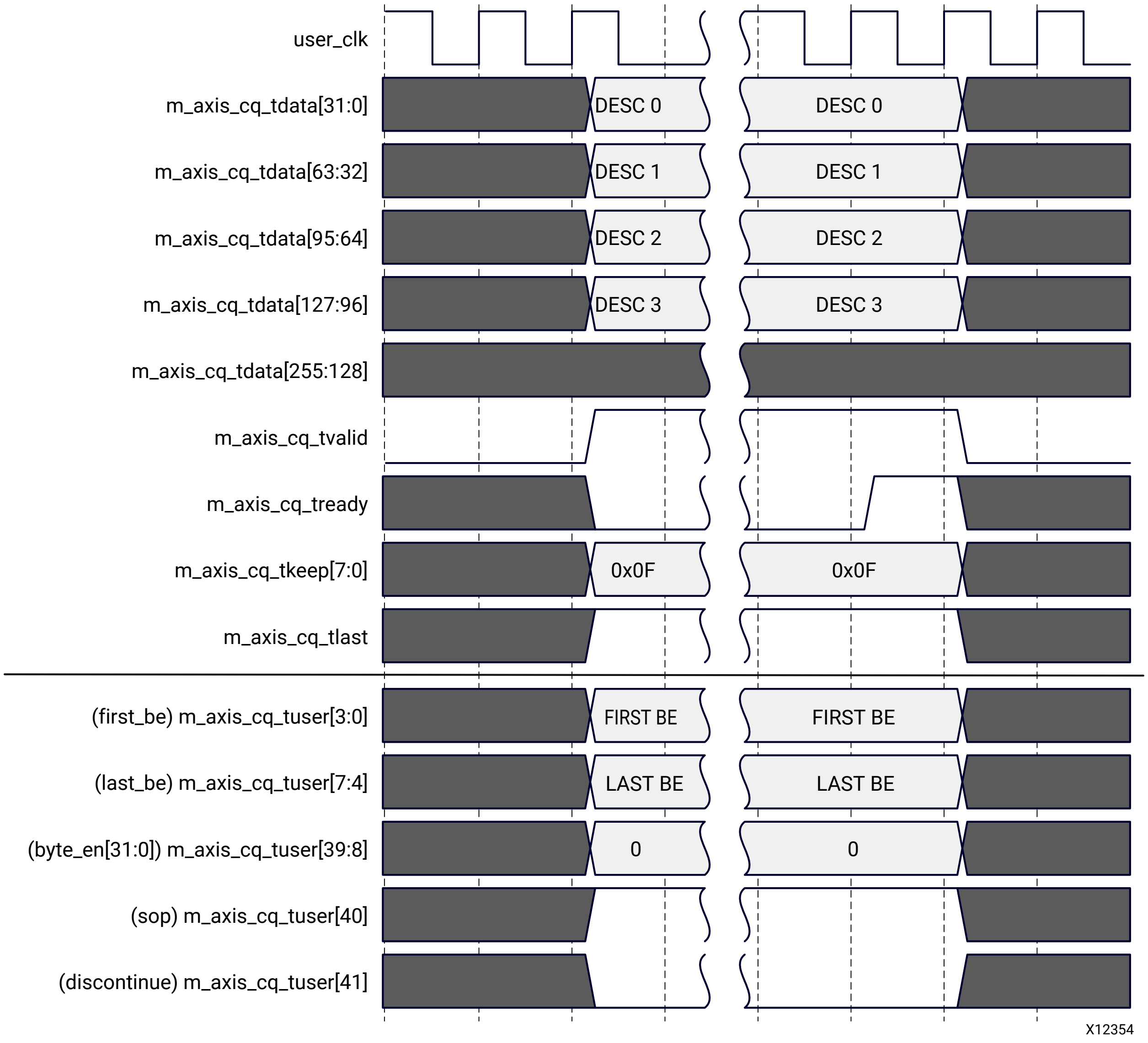


Figure 21: **Memory Read Transaction on the Completer Request Interface (256-Bit Interface)**



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.

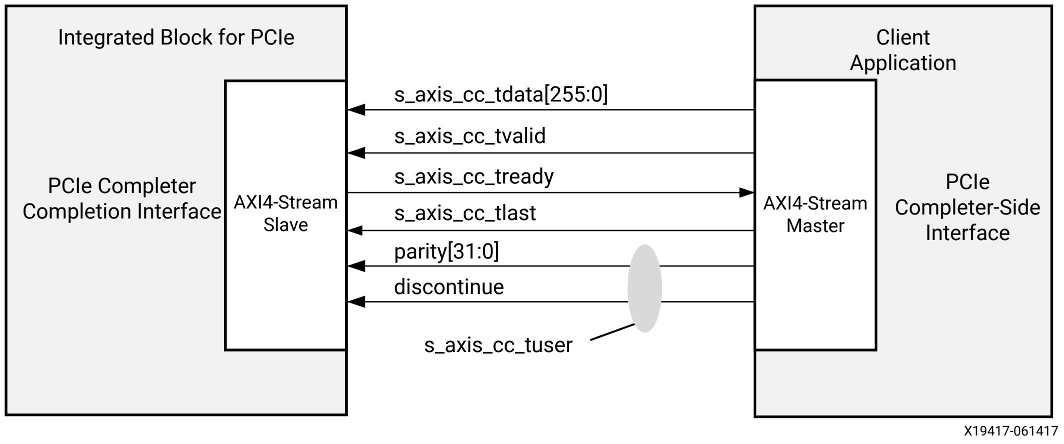
The assertion and deassertion of the `pcie_cq_np_req[0]` signal does not need to be aligned with the packet transfers on the completer request interface.

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 22: **Completer Completion Interface Signals**



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

Completer Completion Descriptor Format

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

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

Figure 23: **Completer Completion Descriptor Format**

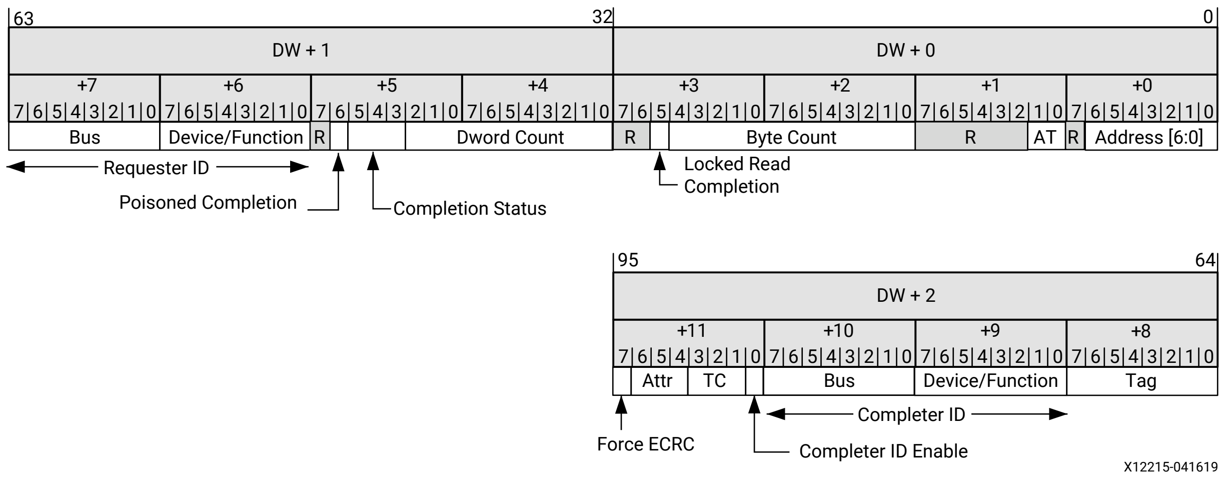


Table 51: **Completer Completion Descriptor Fields**

Bit Index	Field Name	Description
6:0	Lower Address	For memory read Completions, this field must be set to the least significant 7 bits of the starting byte-level address of the memory block being transferred. For the first (or only) Completion, the Completer can generate this field from the least significant 5 bits of the address of the Request concatenated with 2 bits of byte-level address formed by the byte enables for the first Dword of the Request as shown below. **first_be[3:0]** → **Lower Address[1:0]** 4'b0000 → 2'b00 4'bxxx1 → 2'b00 4'bxx10 → 2'b01 4'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 51: **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	Completion Status	This bit can be used to poison the Completion TLP being sent. This bit must be set to 0 for all Completions, except when the user application detects an error in the block of data following the descriptor and wants to communicate this information using the Data Poisoning feature of PCI Express.
63:48	Requester ID	PCI Requester ID associated with the request (copied from the request).
71:64	Tag	PCI Express Tag associated with the request (copied from the request).

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

Bit Index	Field Name	Description
79:72	Target Function/ Device Number	Device and/or Function number of the Completer Function. Endpoint mode: ARI enabled: • Bits [79:72] must be set to the Completer Function number. ARI disabled: • Bits [74:72] must be set to the Completer Function number. • Bits [79:75] are not used Upstream Port for Switch use case (Endpoint mode is selected within the IP): ARI enabled: • Bits [79:72] must be set to the Completer Function number. ARI disabled: • Bits [74:72] must be set to the Completer Function number. • Bits [79:75] are not used if the Completion is originating from the switch itself. These bits must be set to the Completer Device number where the Completion was originated if the switch is relaying the Completion (Completer is external to the switch). This is used in conjunction with Completer ID Enable bit in the descriptor. Root Port mode (Downstream Port): ARI enabled: • Bits [79:72] must be set to the Completer Function number. ARI disabled: • Bits [74:72] must be set to the Completer Function number. • Bits [79:75] must be set to the Completer Device number. This is used in conjunction with Completer ID Enable bit in the descriptor.
87:80	Completer Bus Number	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 in conjunction with Completer ID Enable bit in the descriptor. Root Port mode (Downstream Port): • Must be set to the Completer Bus number. This is used in conjunction with Completer ID Enable bit in the descriptor.

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

Bit Index	Field Name	Description
88	Completer ID Enable	Values are: • 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 in conjunction 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 in conjunction with Completer Bus Number bits [95:88] and Completer Function/Device Number bits [87:83] when ARI is not enabled.
91:89	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request. The user application must copy this value from the TC field of the associated request descriptor.
94:92	Attributes	PCIe attributes associated with the request (copied from the request). Bit 92 is the No Snoop bit, bit 93 is the Relaxed Ordering bit, and bit 94 is the ID-Based Ordering bit.
95	Force ECRC	Force ECRC insertion. Setting this bit to 1 forces the integrated block to append a TLP Digest containing ECRC to the Completion TLP, even when ECRC is not enabled for the Function sending the Completion.

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

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

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

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

Completions with Successful Completion Status

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

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

The CC interface supports two data alignment modes: Dword-aligned and address-aligned. The following timing diagrams illustrate the Dword-aligned transfer of a Completion from the user application across the CC interface, when the interface width is configured as 64, 128, and 256 bits, respectively. In this case, the first Dword of the payload starts immediately after the descriptor. When the data block is not a multiple of four bytes, or when the start of the payload is not aligned on a Dword address boundary, the user application must add null bytes to align the start of the payload on a Dword boundary and make the payload a multiple of Dwords. For example, when the data block starts at byte address 7 and has a size of 3 bytes, the user application must add three null bytes before the first byte and two null bytes at the end of the block to make it two 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 24: **Transfer of a Normal Completion on the Completer Completion Interface (Dword-Aligned Mode, 64-Bit Interface)**

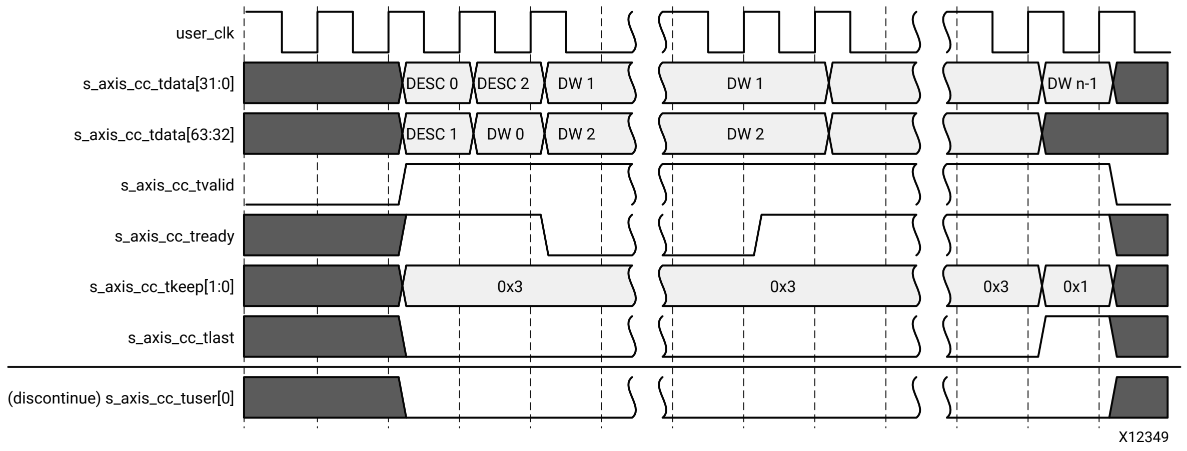


Figure 25: **Transfer of a Normal Completion on the Completer Completion Interface (Dword-Aligned Mode, 128-Bit Interface)**

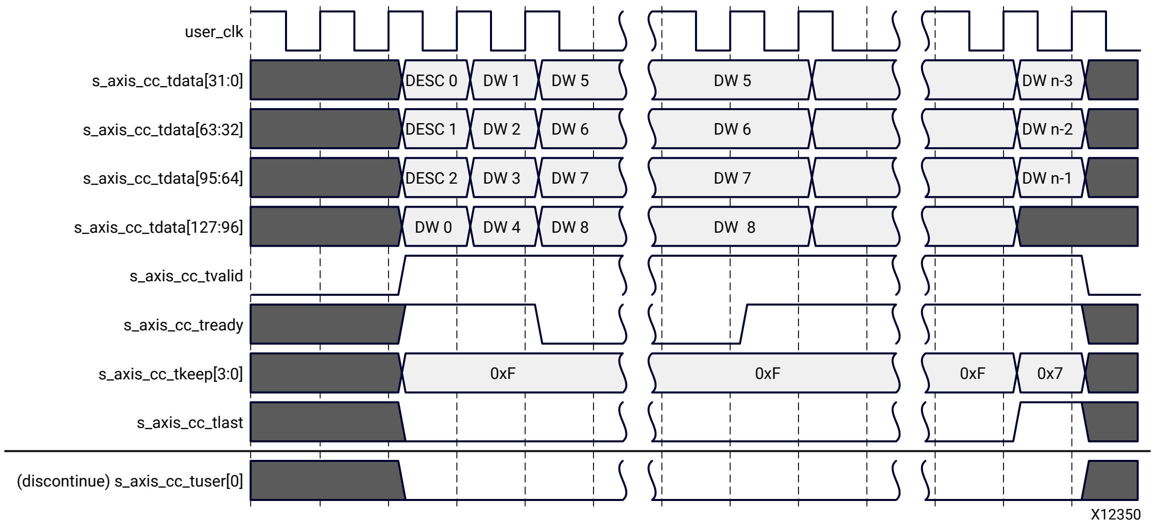
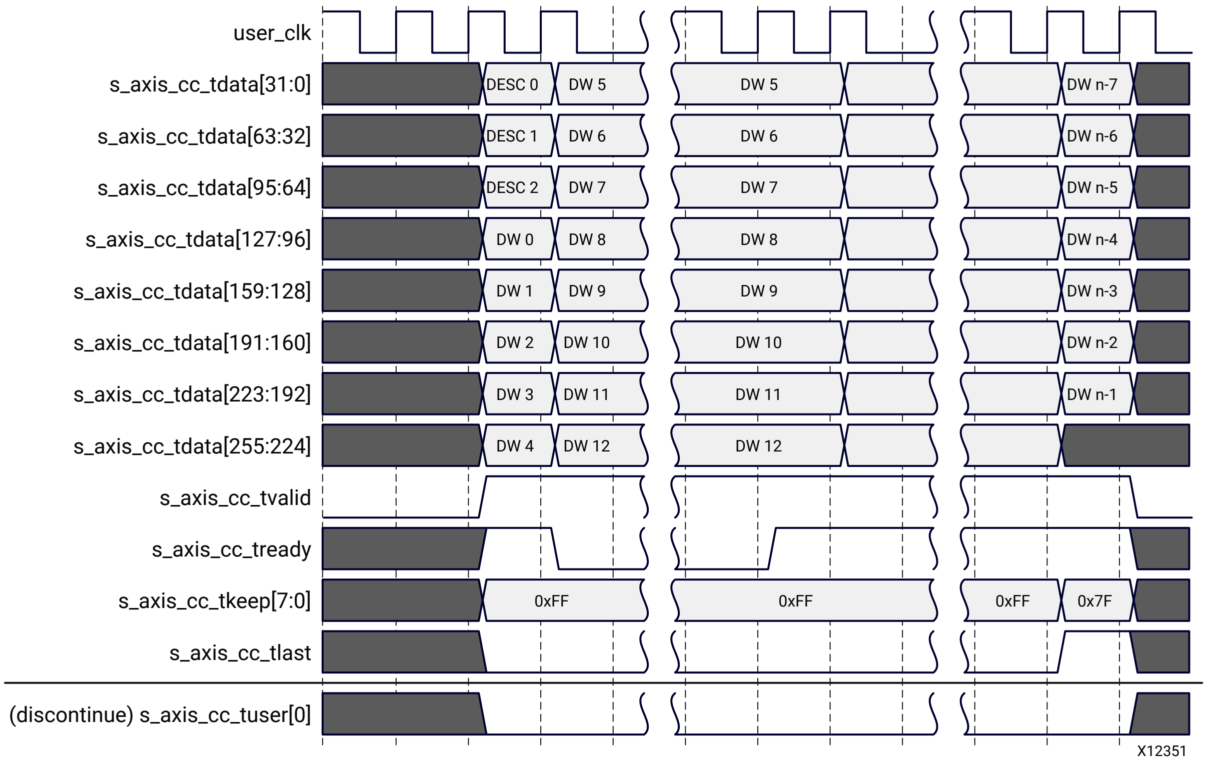


Figure 26: **Transfer of a Normal Completion on the Completer Completion Interface (Dword-Aligned Mode, 256-Bit Interface)**



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

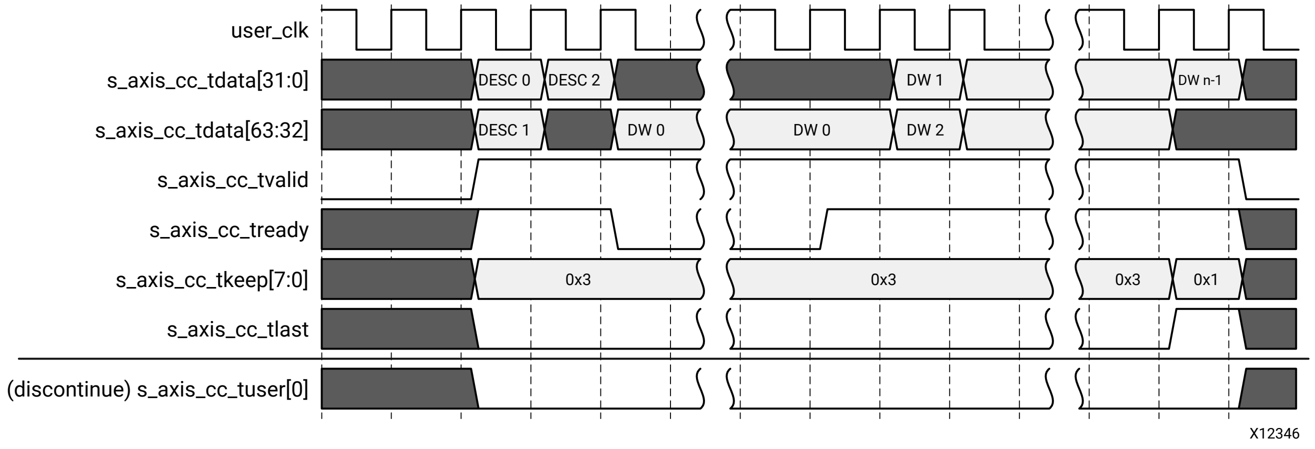


Figure 28: **Transfer of a Normal Completion on the Completer Completion Interface (Address-Aligned Mode, 128-Bit Interface)**

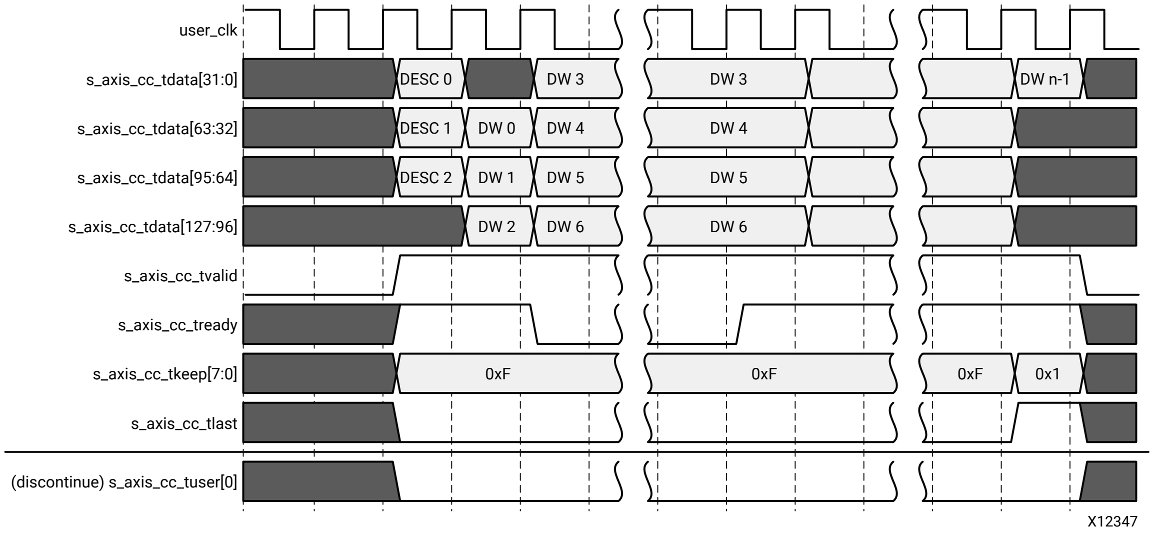
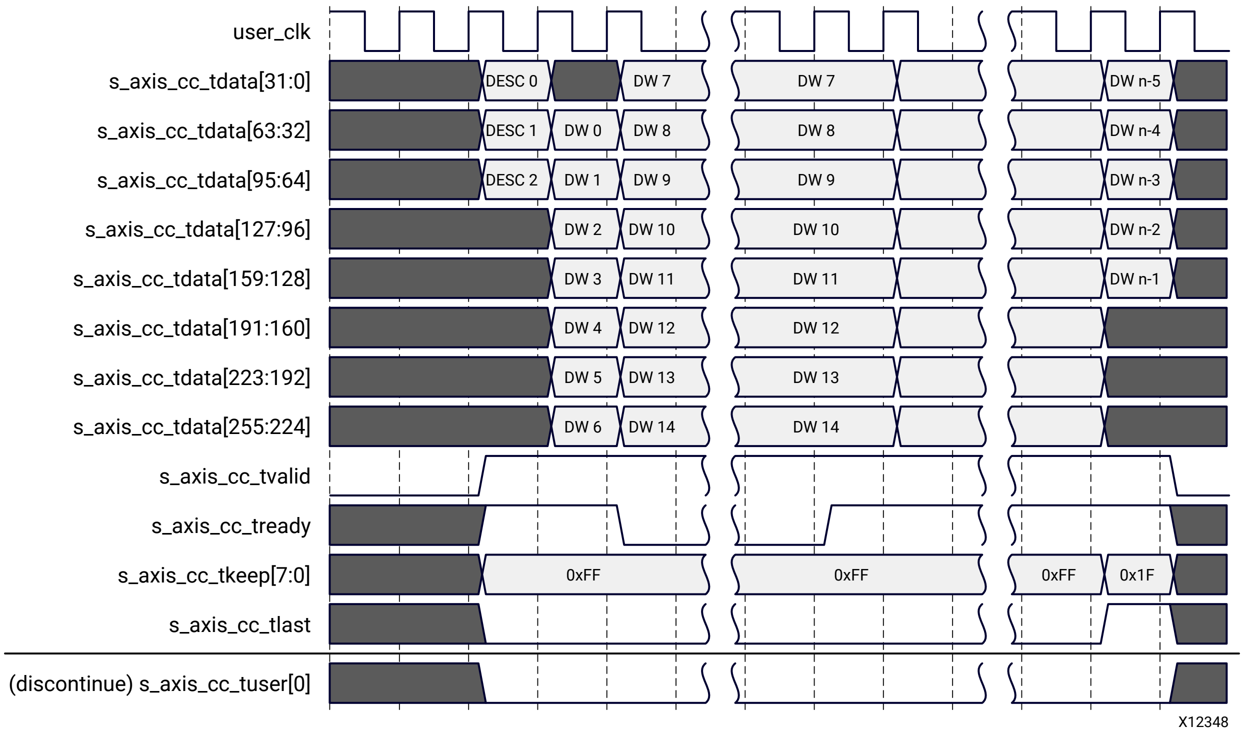


Figure 29: **Transfer of a Normal Completion on the Completer Completion Interface (Address-Aligned Mode, 256-Bit Interface)**



Aborting a Completion Transfer

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

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

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

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

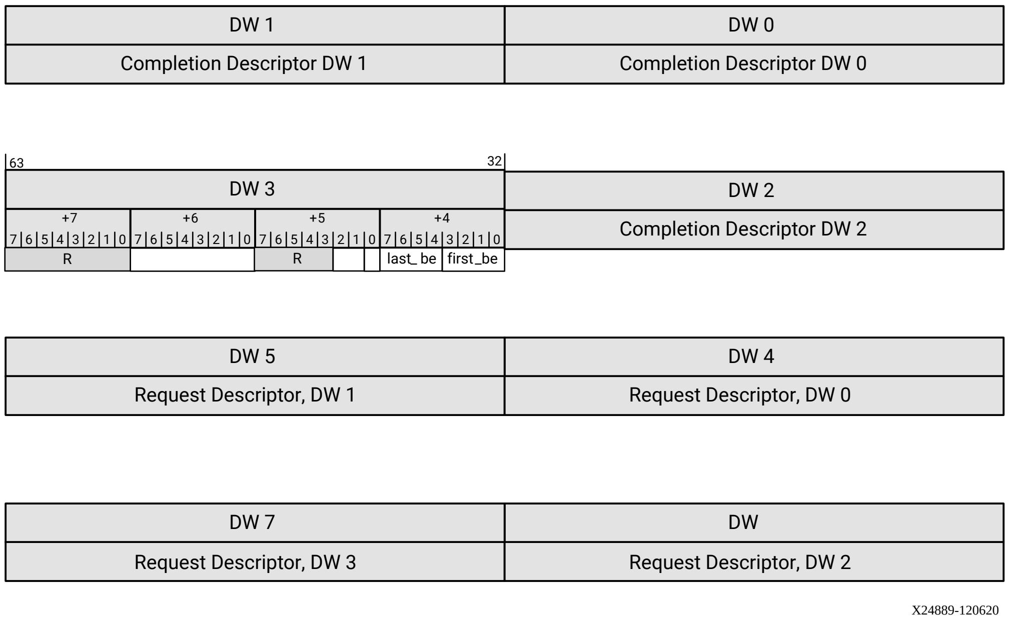
Completions with Error Status (UR and CA)

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

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

- The first three Dwords contain the completion descriptor in the format of the *Completer Completion Descriptor Format* figure in Completer Completion Descriptor Format.
- The fourth Dword contains the state of the following signals in `m_axis_cq_tuser`, copied from the request:
  - The First Byte Enable bits `first_be[3:0]` in `m_axis_cq_tuser`.
  - The Last Byte Enable bits `last_be[3:0]` in `m_axis_cq_tuser`.

Figure 30: **Composition of the AXI4-Stream Packet for UR and CA Completions**



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

64/128/256-Bit Requester Interface

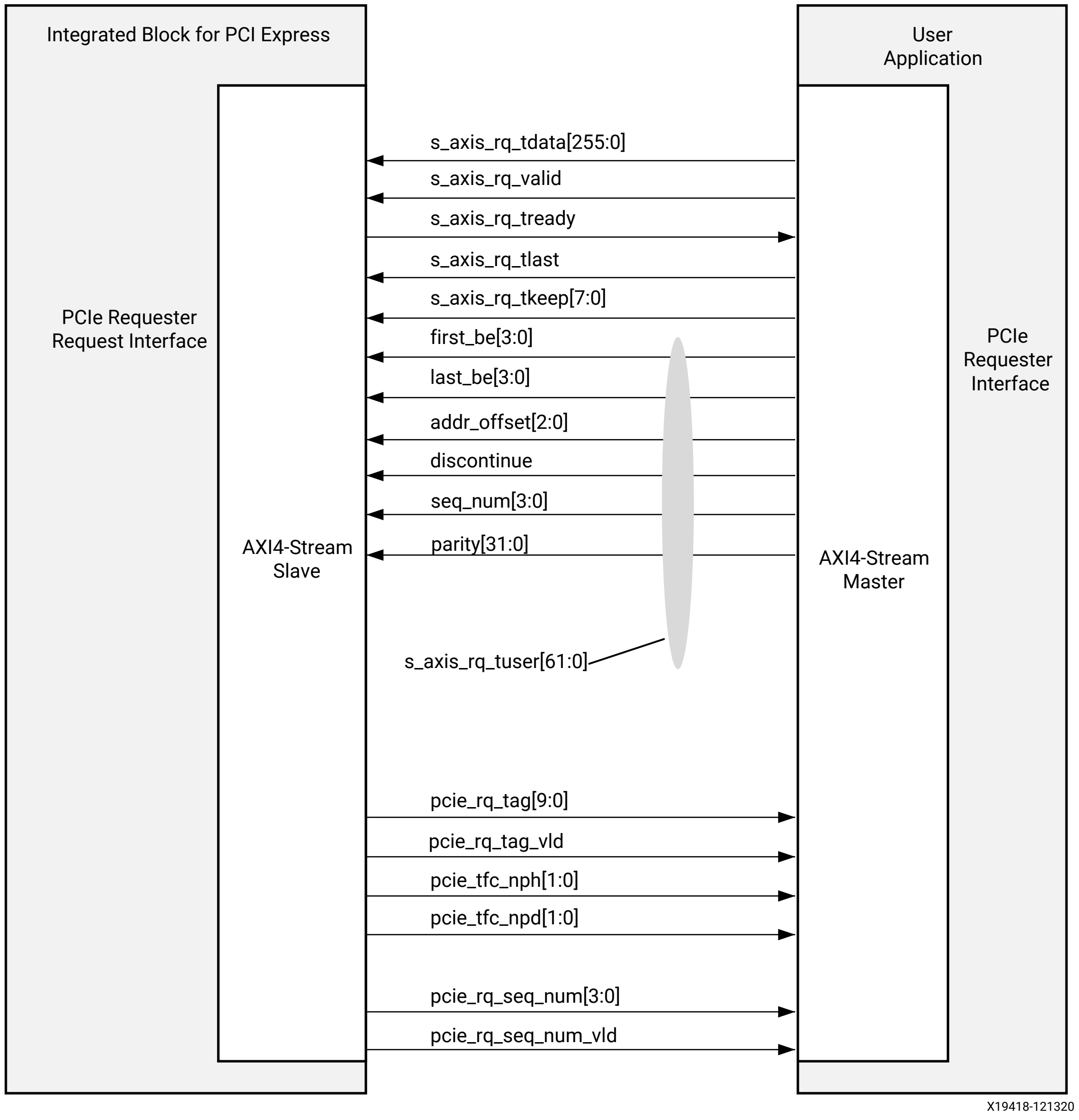
The requester interface enables a user Endpoint application to initiate PCIe transactions as a bus master across the PCIe link to the host memory. For Root Complexes, this interface is also used to initiate I/O and configuration requests. This interface can also be used by both Endpoints and Root Complexes 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 31: **Requester Request Interface**



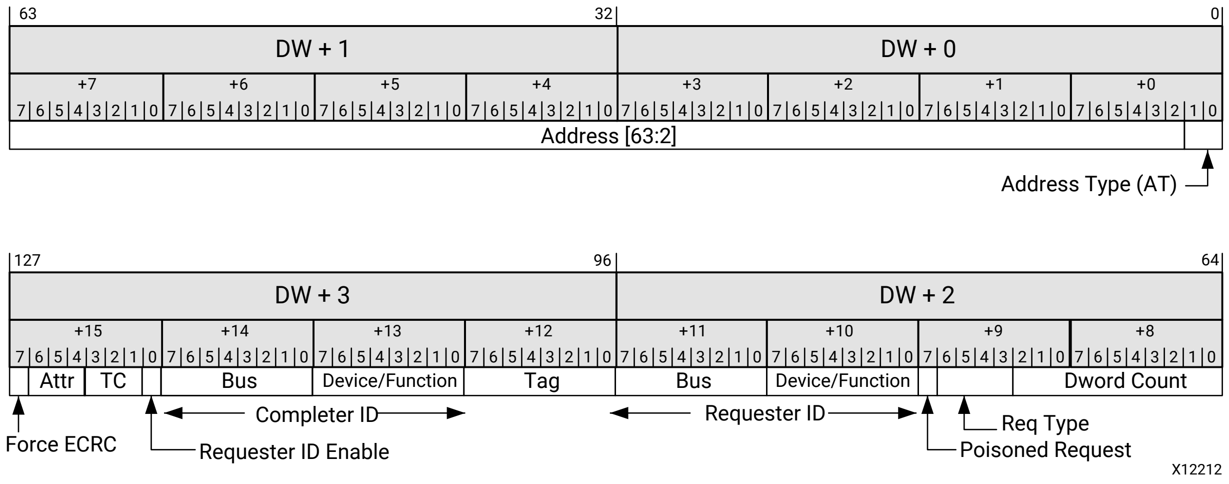
The RQ interface supports two distinct data alignment modes for transferring payloads. In the Dword-aligned mode, the user logic must provide the first Dword of the payload immediately after the last Dword of the descriptor. It must also set the bits in `first_be[3:0]` to indicate the valid bytes in 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.

Requester Request Descriptor Formats

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

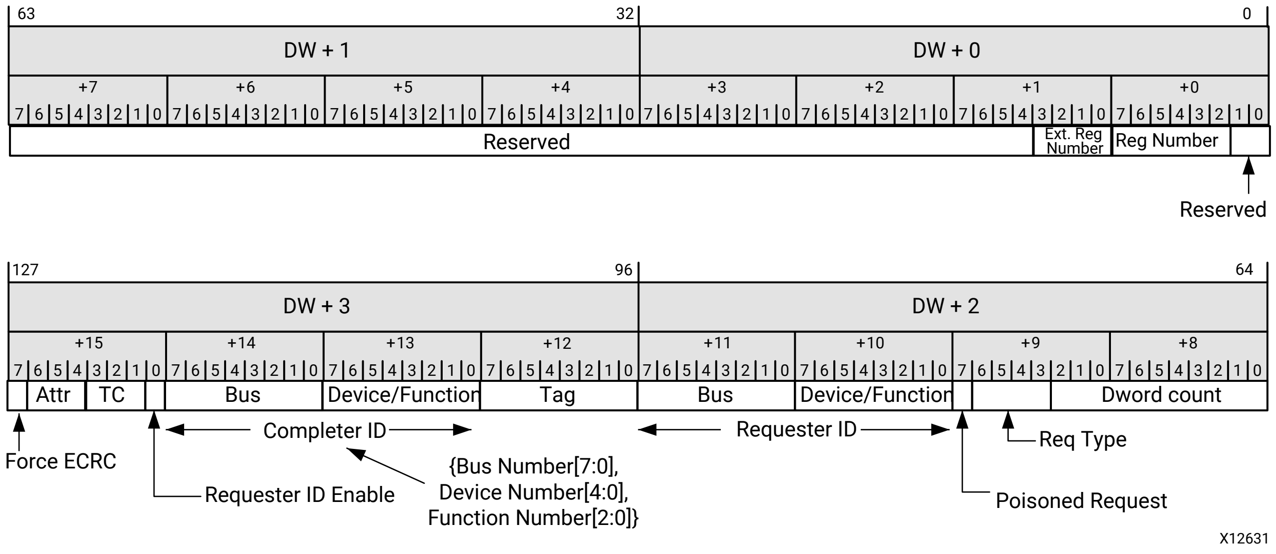
The format of the following figure applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request.

Figure 32: **Requester Request Descriptor Format for Memory, I/O, and Atomic Operation Requests**



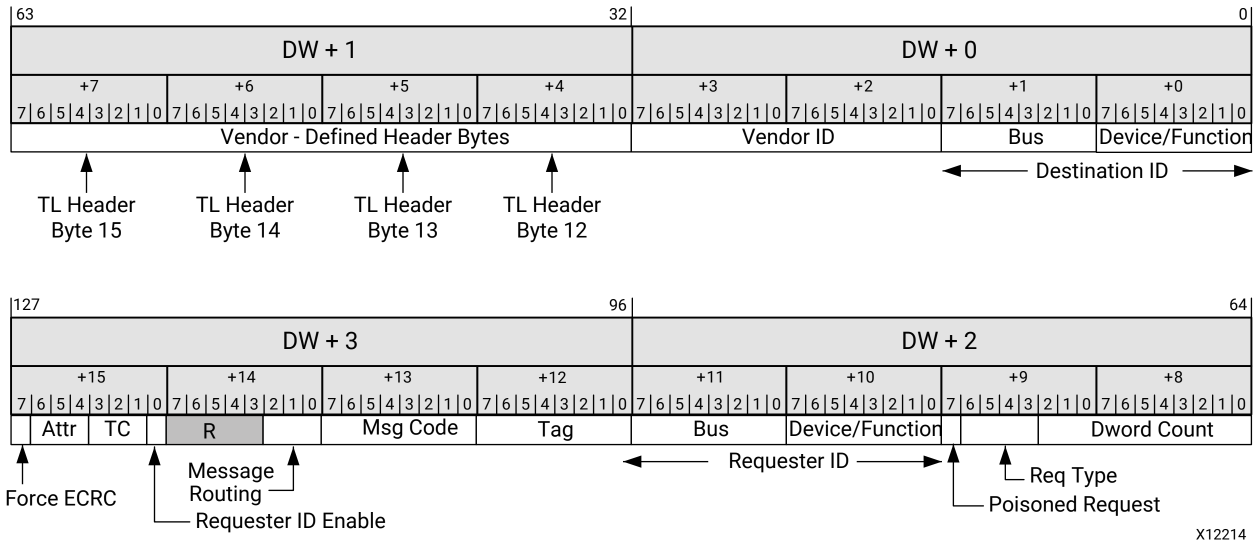
The format in the following figure is used for Configuration Requests.

Figure 33: **Requester Request Descriptor Format for Configuration Requests**



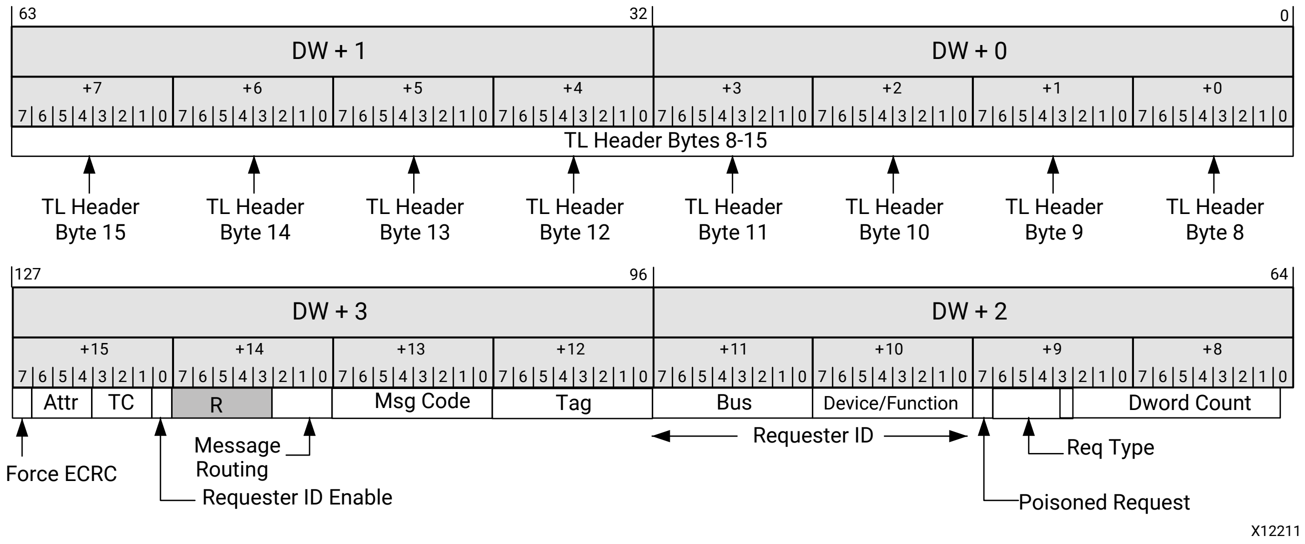
The format in the following figure is used for Vendor-Defined Messages (Type 0 or Type 1) only.

Figure 34: **Requester Request Descriptor Format for Vendor-Defined Messages**



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

Figure 35: **Requester Request Descriptor Format for ATS Messages**



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

Figure 36: **Requester Request Descriptor Format for all other Messages**

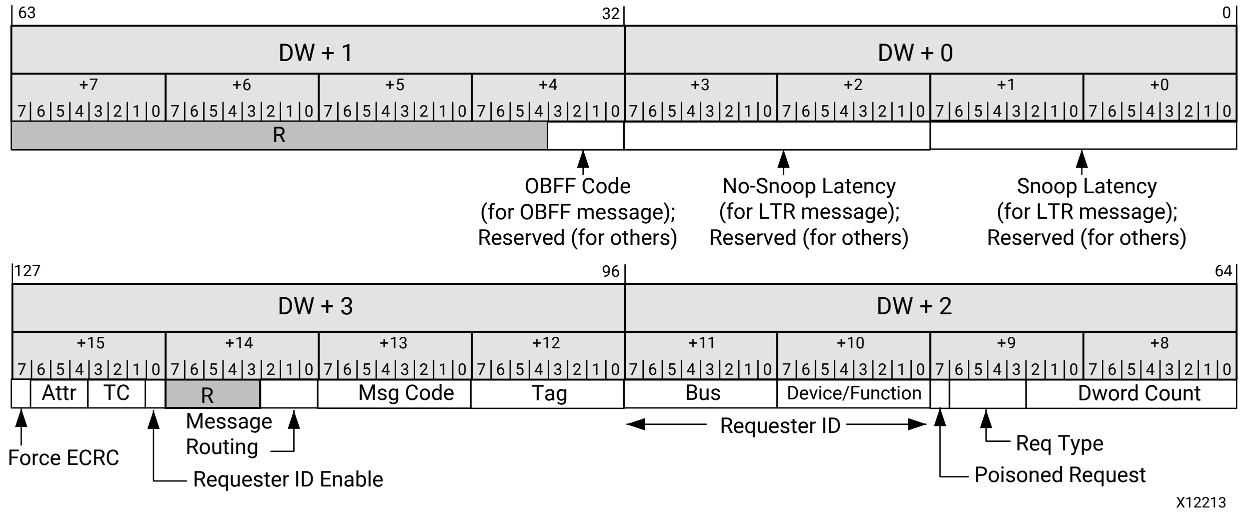


Table 53: **Requester Request Descriptor Fields**

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

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

Bit Index	Field Name	Description
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 transactions types and their encodings are listed in Table 50: Transaction Types.
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 in conjunction 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 in conjunction with Requester ID Enable bit in the descriptor.

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

Bit Index	Field Name	Description
95:88	Requester Bus Number	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 in conjunction with Requester ID Enable bit in the descriptor. Root Port mode (Downstream Port): • Must be set to the Requester Bus number. This is used in conjunction with Requester ID Enable bit in the descriptor.
103:96	Tag	PCIe Tag associated with the request. For 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).
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. When Requester ID enable is 0 the device number fields in descriptor should also be 0. 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 in conjunction 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 in conjunction with Requester Bus Number bits [95:88] and Requester Function/Device Number bits [87:83] when ARI is not enabled.
123:121	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request.
126:124	Attributes	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.

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

Bit Index	Field Name	Description
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.
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 (*PCI-SIG Specifications* (https://www.pcisig.com/specifications)) provides a complete list of the supported Message Codes.
114:112	Message Routing	This field is defined for all messages. The integrated block copies these bits into the 3-bit Routing field r[2:0] of the TLP header of the Request TLP.
15:0	Destination ID	This field applies to Vendor-Defined Messages only. When the message is routed by ID (that is, when the Message Routing field is 010 binary), this field must be set to the Destination ID of the message.
63:32	Vendor-Defined Header	This field applies to Vendor-Defined Messages only. It is copied into Dword 3 of the TLP header.
63:0	ATS Header	This field is applicable to ATS messages only. It contains the bytes that the integrated block copies into Dwords 2 and 3 of the TLP header.

Requester Memory Write Operation

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

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

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

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

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

Figure 37: **Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, 64-Bit Interface)**

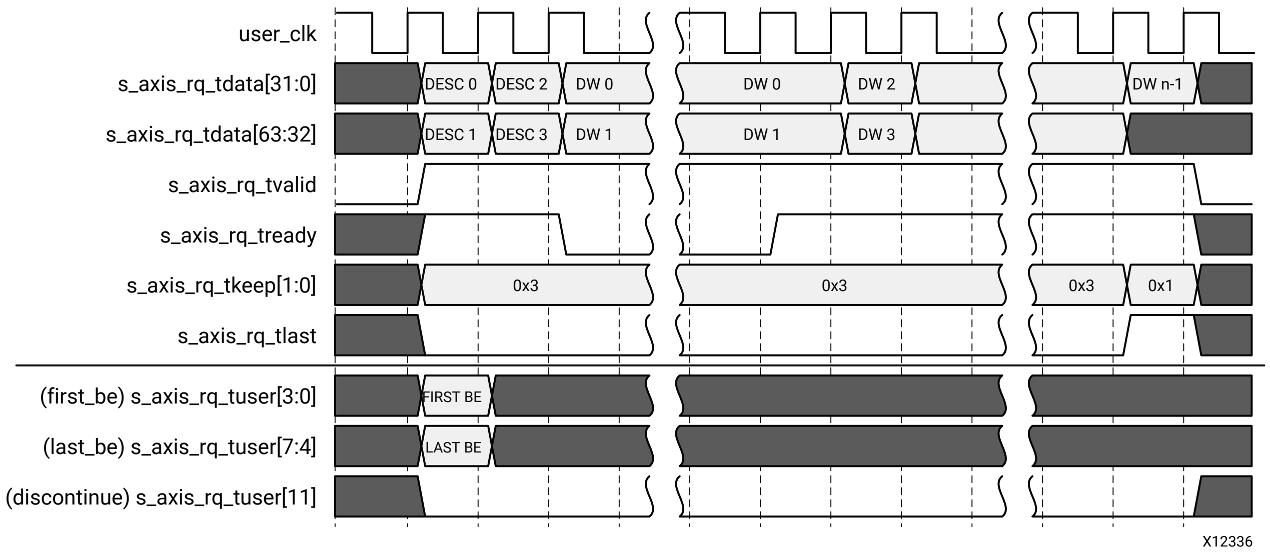


Figure 38: **Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, 128-Bit Interface)**

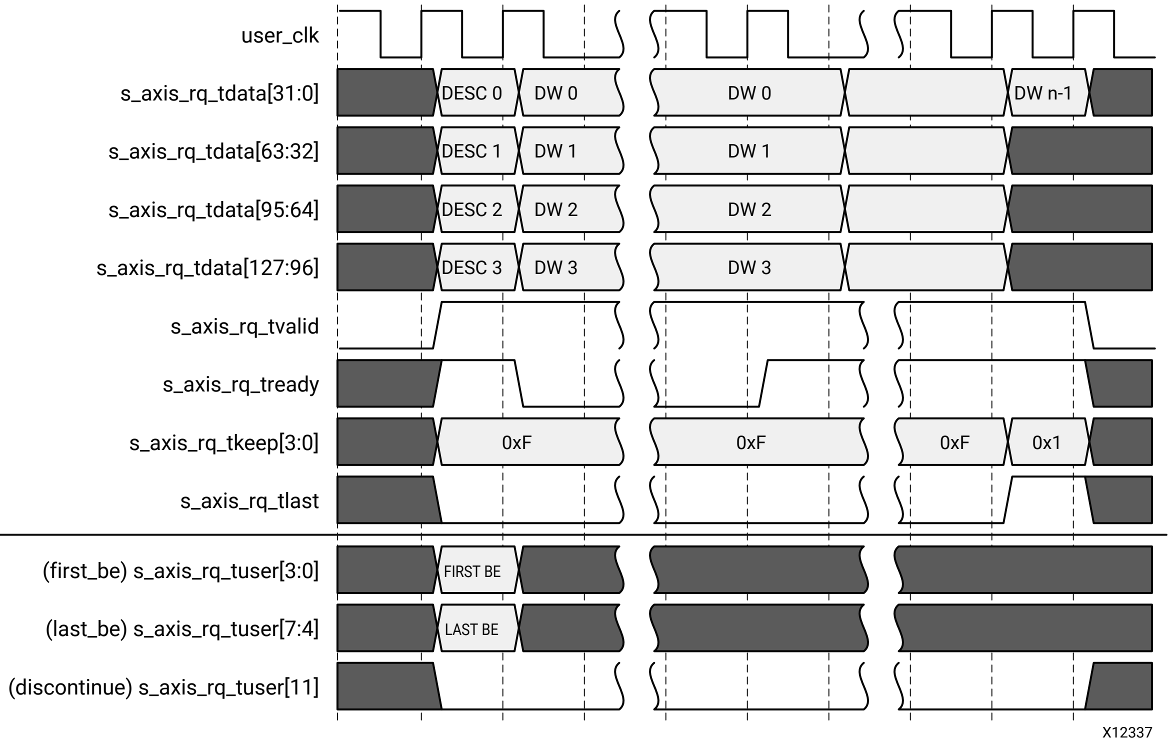
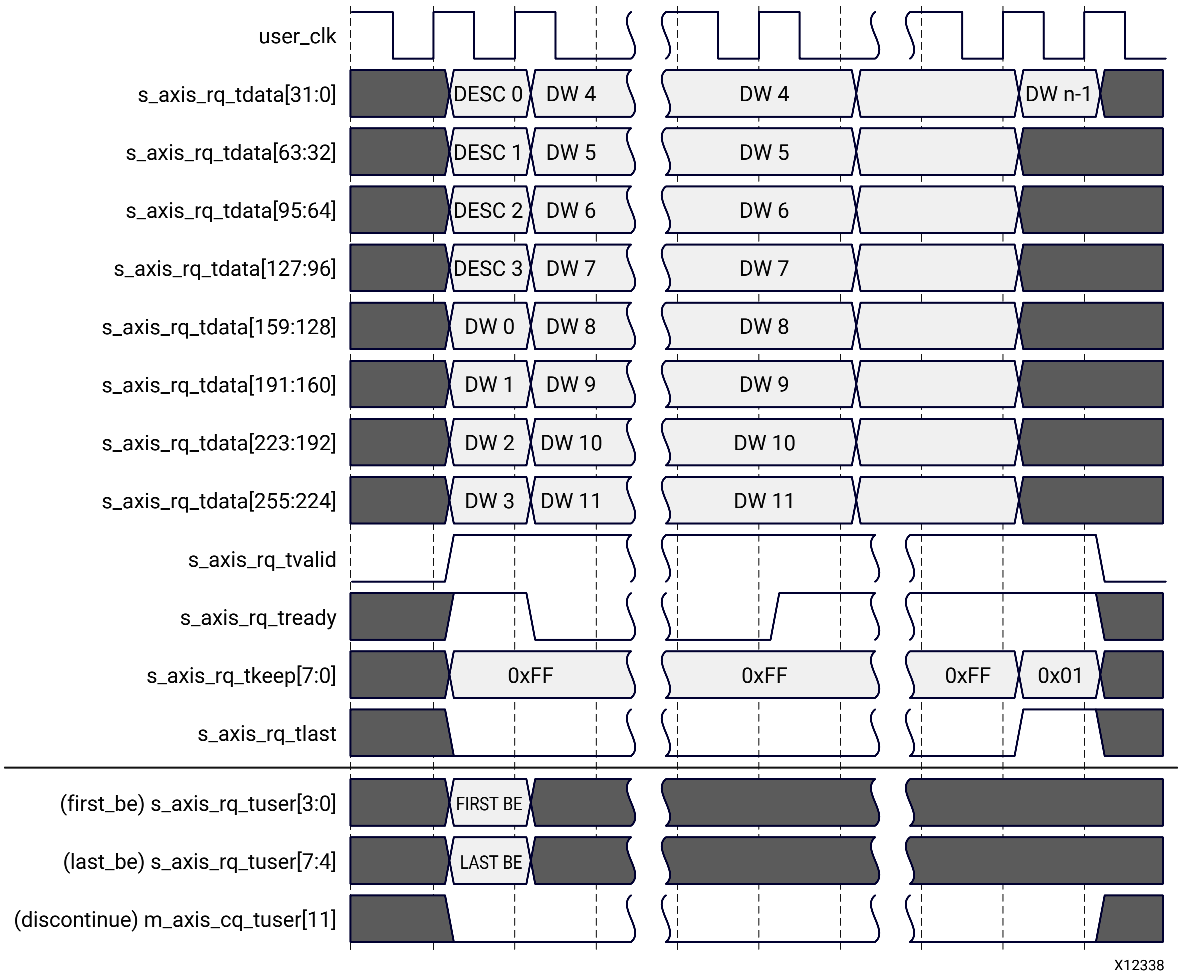


Figure 39: **Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, 256-Bit Interface)**



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

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

Figure 40: **Memory Write Transaction on the Requester Request Interface (Address-Aligned Mode, 64-Bit Interface)**

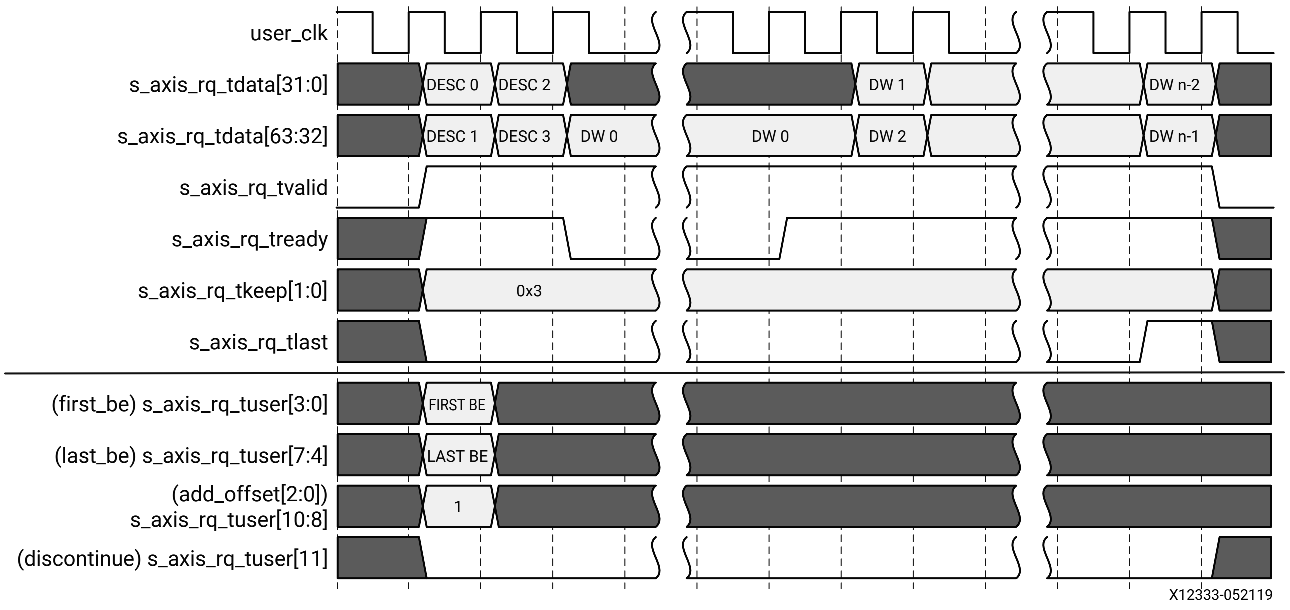


Figure 41: **Memory Write Transaction on the Requester Request Interface (Address-Aligned Mode, 128-Bit Interface)**

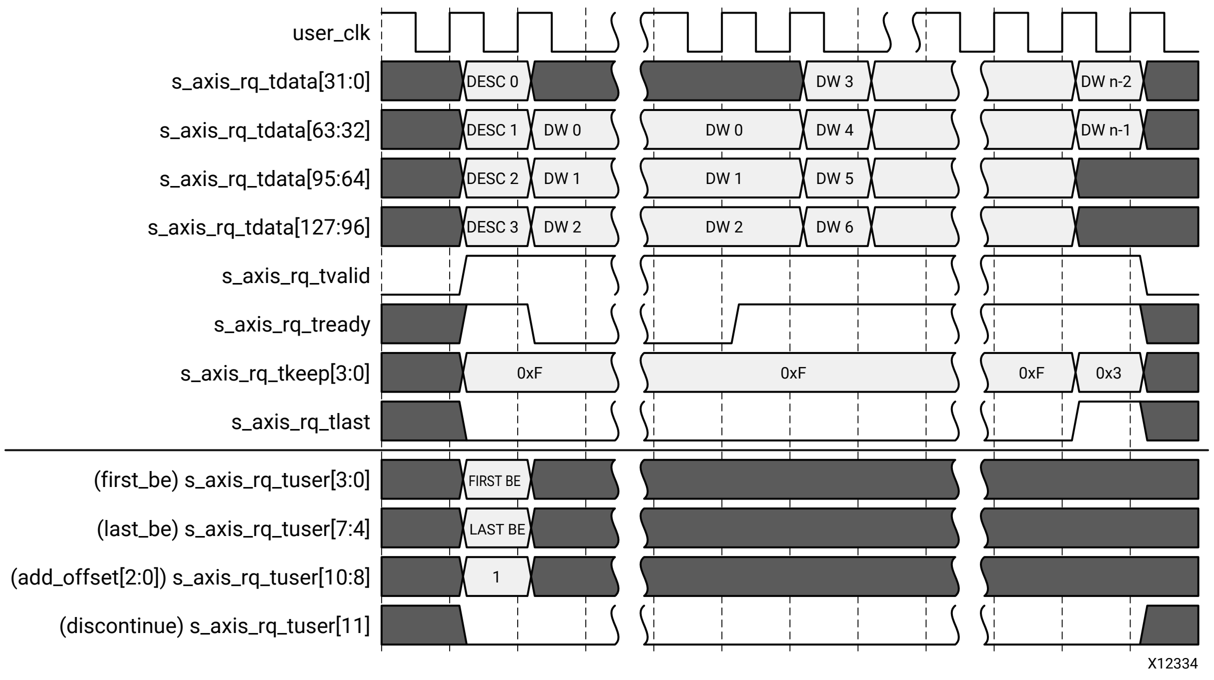
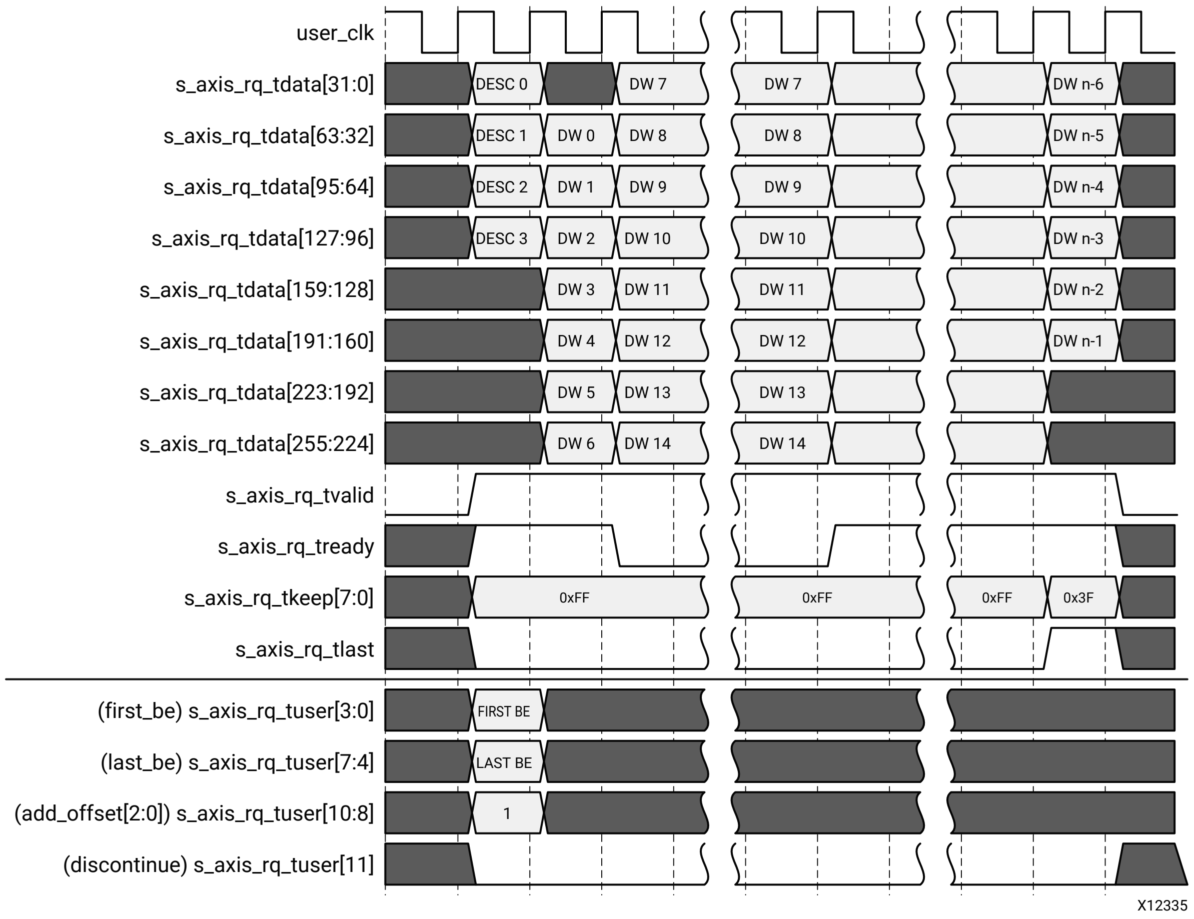


Figure 42: **Memory Write Transaction on the Requester Request Interface (Address-Aligned Mode, 256-Bit Interface)**



Non-Posted Transactions with No Payload

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

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

Figure 43: **Memory Read Transaction on the Requester Request Interface (64-Bit Interface)**

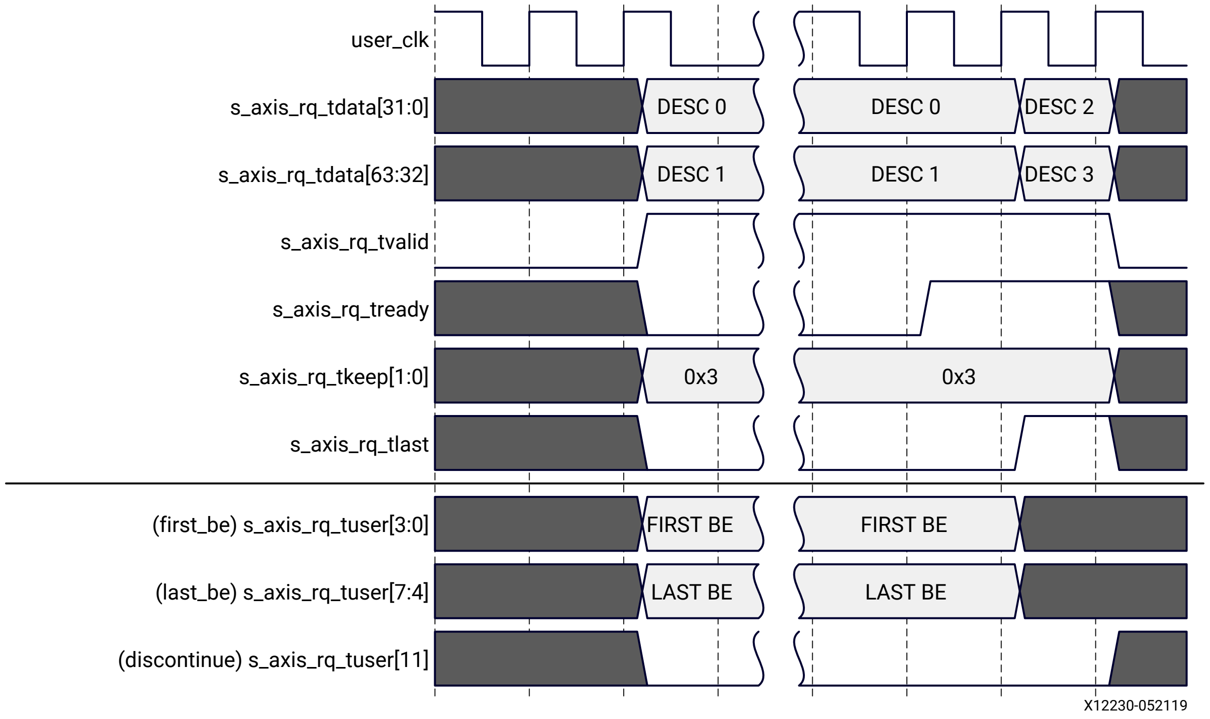


Figure 44: **Memory Read Transaction on the Requester Request Interface (128-Bit Interface)**

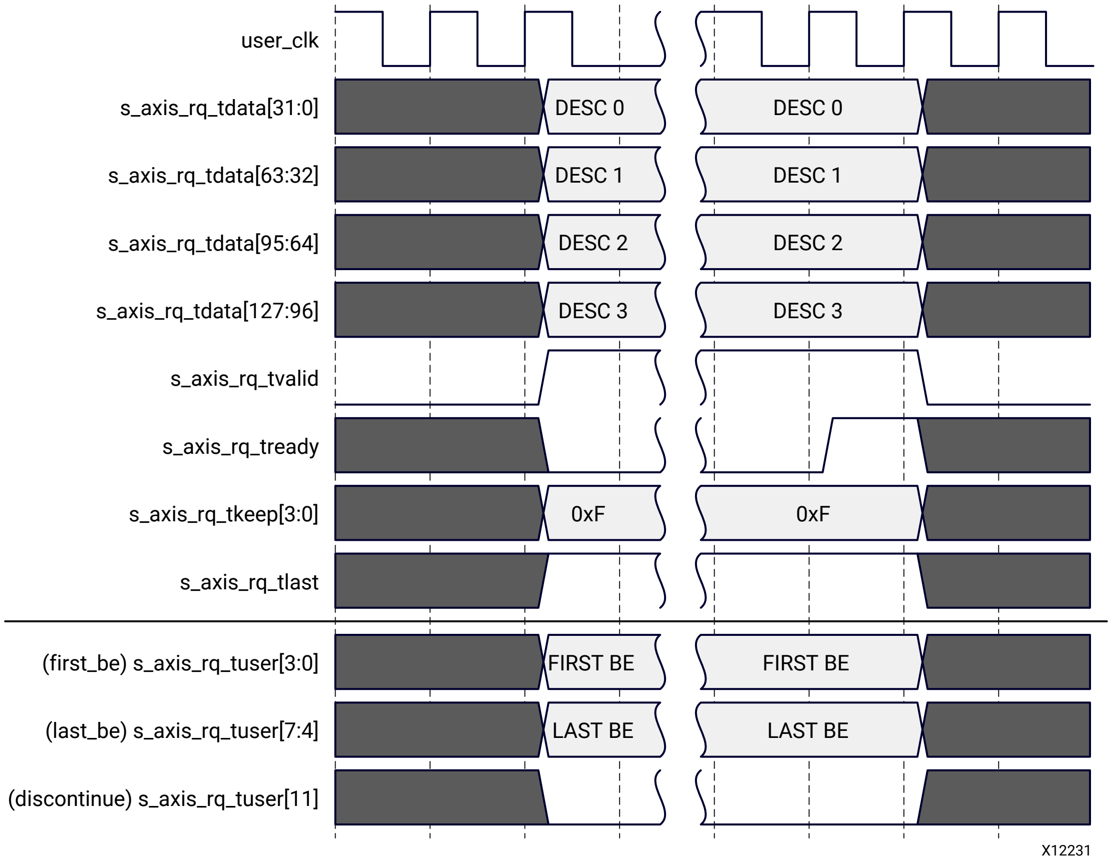
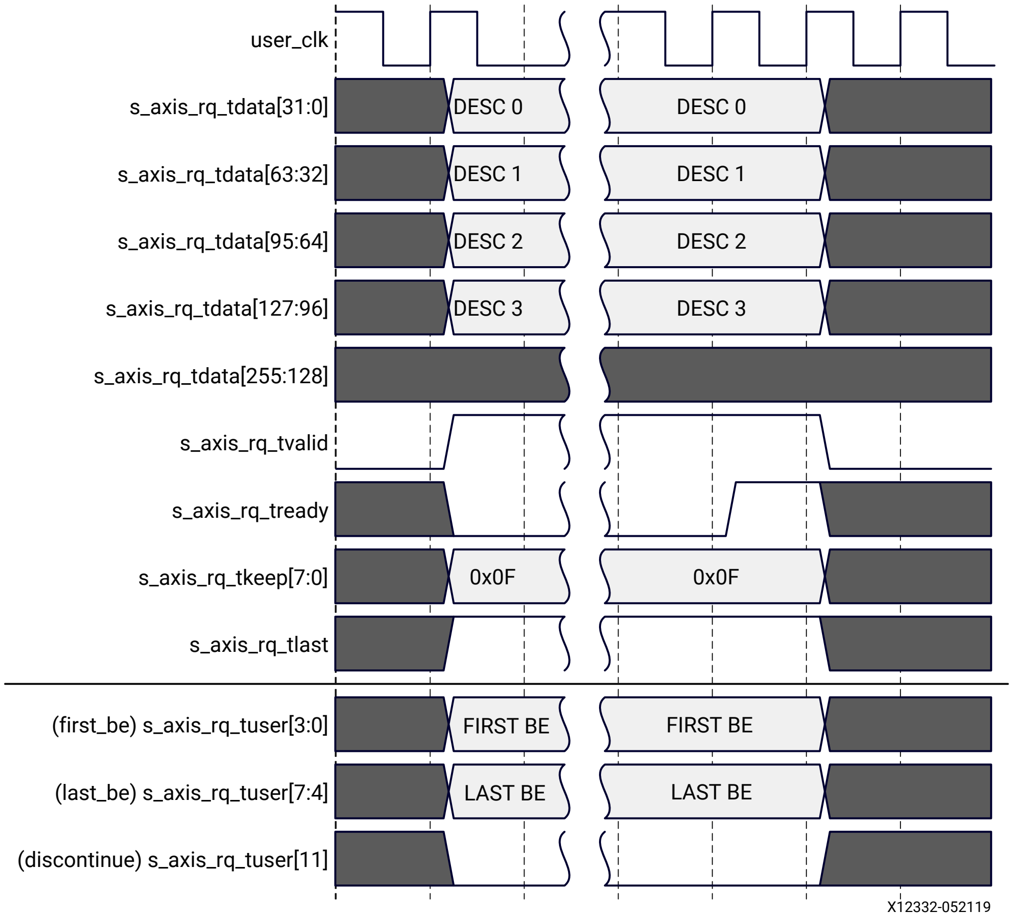


Figure 45: **Memory Read Transaction on the Requester Request Interface (256-Bit Interface)**



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 up to 768 Non-Posted transactions. The returning Completions are matched with the pending requests using an 5-/8-/10-bit tag. There are two options for management of these tags.

- **Internal Tag Management:** This mode of operation is selected by setting the Enable Client Tag checkbox is deselected (not set) the Vivado® IDE, which is the default setting for the core. In this mode, logic within the integrated block is responsible for allocating the tag for each Non-Posted request initiated from the requester side. The integrated block maintains a list of free tags and assigns one of them to each request when the user application initiates a Non-Posted transaction, and communicates the assigned tag value to the user application through the output `pcie_rq_tag0[9:0]`. The value on this bus is valid when the integrated block asserts `pcie_rq_tag_vld0`. The user logic must copy this tag so that any Completions delivered by the integrated block in response to the request can be matched to the request.

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

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

When internal tag management is in use, the integrated block asserts `pcie_rq_tag_vld0` for one cycle for each Non-Posted request, after it has placed its allocated tag on `pcie_rq_tag[9:0]`. There can be a delay of several cycles between the transfer of the request on the RQ interface and the assertion of `pcie_rq_tag_vld0` by the integrated block to provide the allocated tag for the request. The user application can, meanwhile, continue to send new requests. The tags for requests are communicated on the `pcie_rq_tag0` bus in FIFO order, so it is easy to associate the tag value with the request it transferred. A tag is reused when the end-of-frame (EOF) of the last completion of a split completion is accepted by the user application.

Avoiding Head-of-Line Blocking for Posted Requests

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

- `pcie_tfc_nph_av[1:0]`: These outputs indicate the Header Credit currently available for Non-Posted requests, where:

  00 = no credit available

  01 = 1 credit

  10 = 2 credits

  11 = 3 or more credits

- pcie_tfc_npd_av[1:0]: These outputs indicate the Data Credit currently available for Non-Posted requests, where:

  00 = no credit available

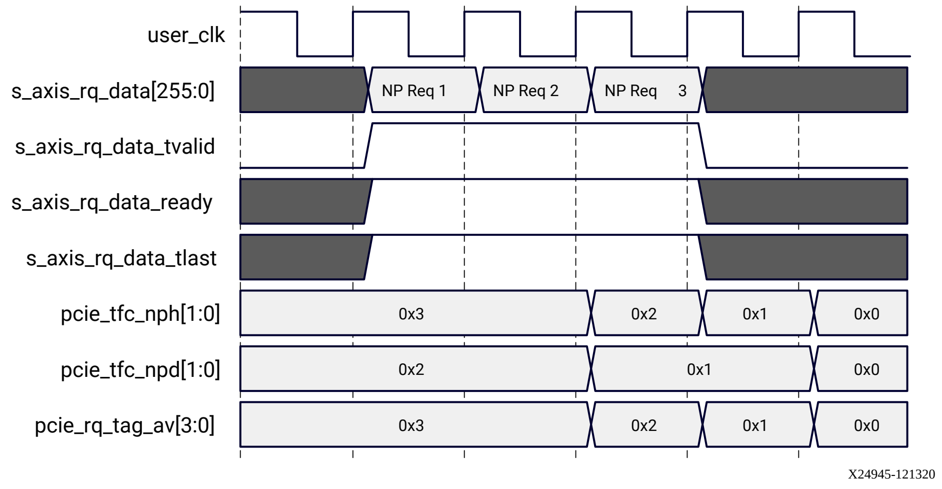
  01 = 1 credit

  10 = 2 credits

  11 = 3 or more credits

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

Figure 46: **Credit and Tag Availability Signals on the Requester Request Interface (256-Bit Interface)**



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

Figure 47: **Credit and Tag Availability Signals on the Requester Request Interface (128-Bit Interface)**

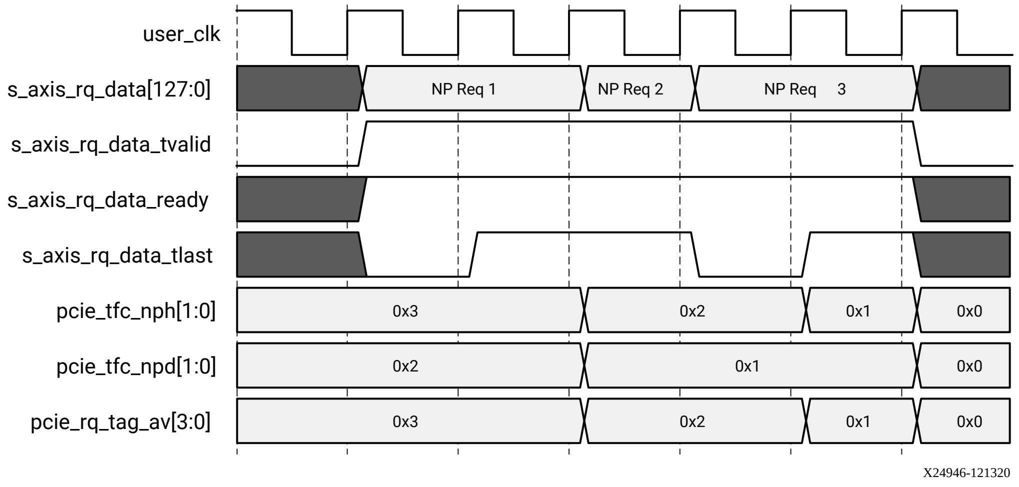
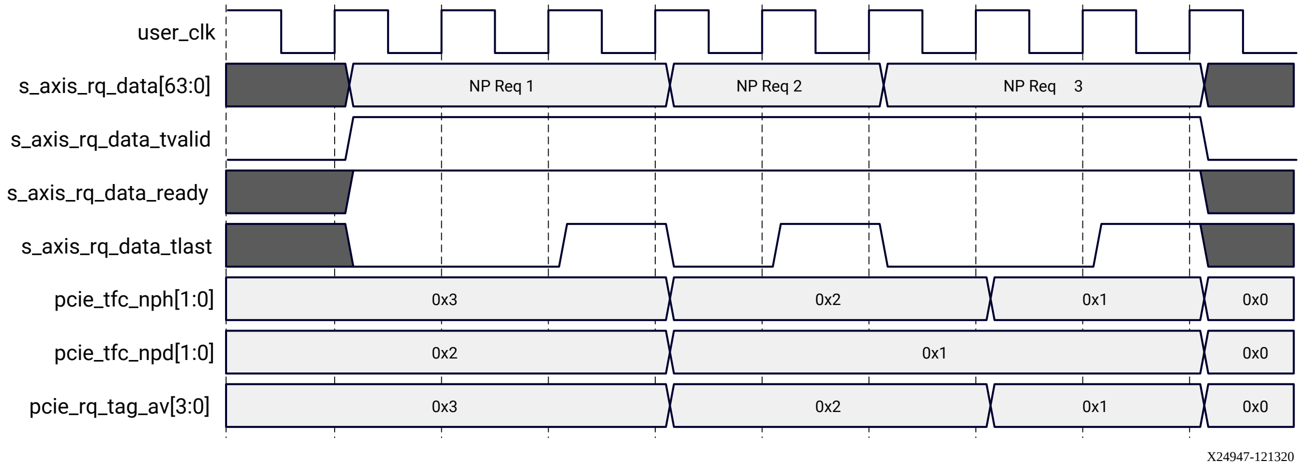


Figure 48: **Credit and Tag Availability Signals on the Requester Request Interface (64-Bit Interface)**



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

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

Maintaining Transaction Order

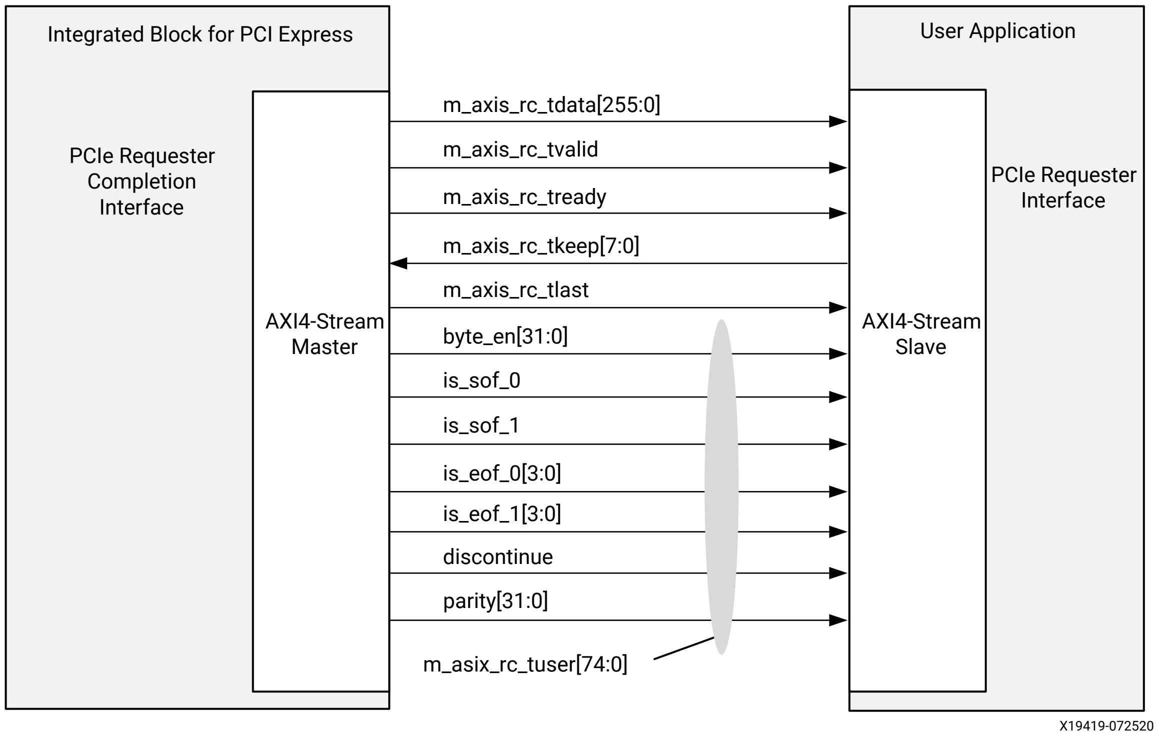
The integrated block does not change the order of requests received from the user application on its requester interface when it transmits them on the link. In cases where the user application would like to have precise control of the order of transactions sent on the RQ interface and the CC interface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

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

Requester Completion Interface Operation

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

Figure 49: **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 50: **Requester Completion Descriptor Format**

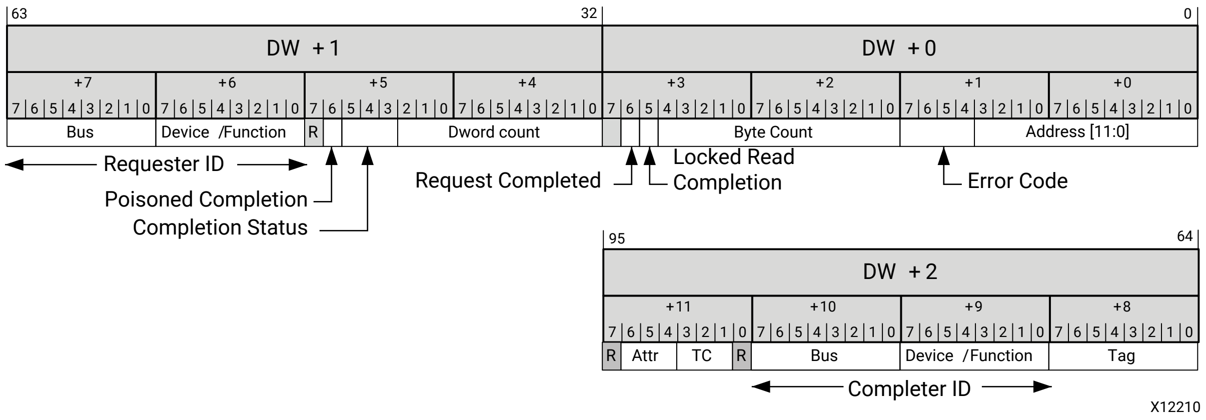


Table 54: **Requester Completion Descriptor Fields**

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

Table 54: **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.
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.

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

Bit Index	Field Name	Description
42:32	Dword Count	These 11 bits indicate the size of the payload of the current packet in Dwords. Its range is 0 - 1K Dwords. This field is set to 1 for I/O read Completions and 0 for I/O write Completions. The Dword count is also set to 1 while transferring a Completion for a zero-length memory read. In all other cases, the Dword count corresponds to the actual number of Dwords in the payload of the current packet.
45:43	Completion Status	These bits reflect the setting of the Completion Status field of the received Completion TLP. The valid settings are: • 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 51: **Transfer of a Completion with no Data on the Requester Completion Interface (64-Bit Interface)**

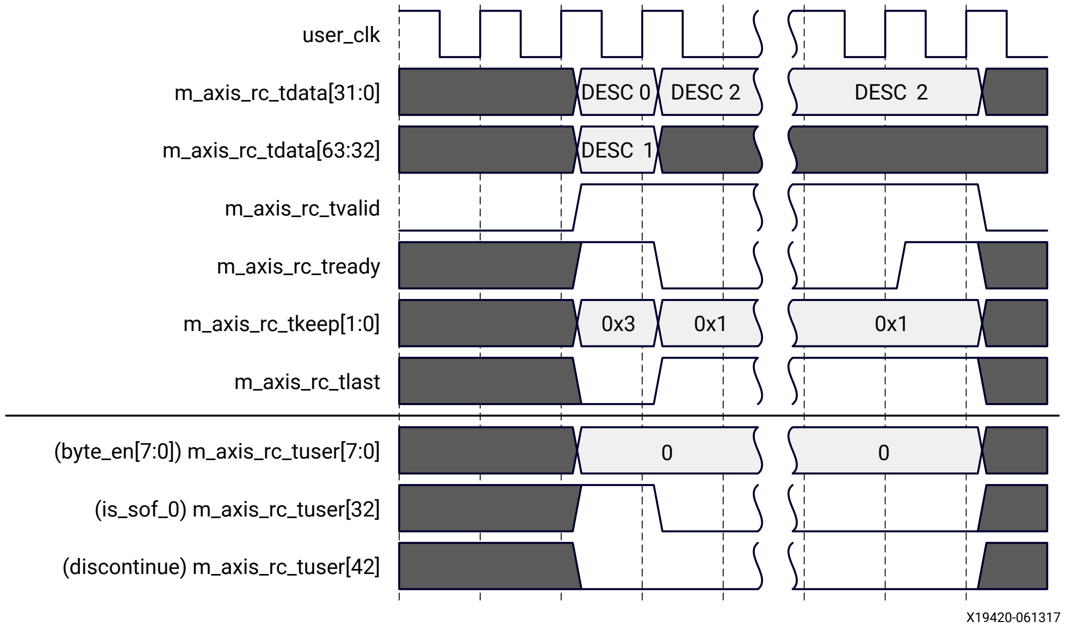


Figure 52: **Transfer of a Completion with no Data on the Requester Completion Interface (128-Bit Interface)**

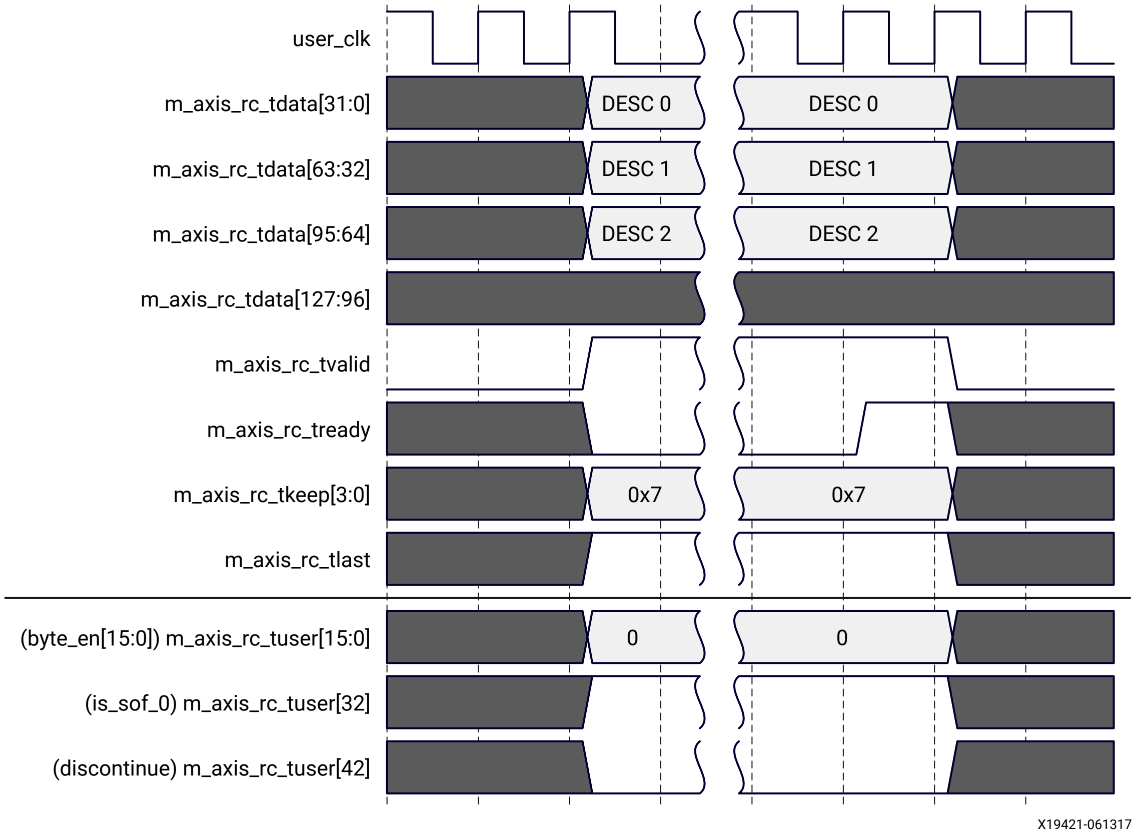
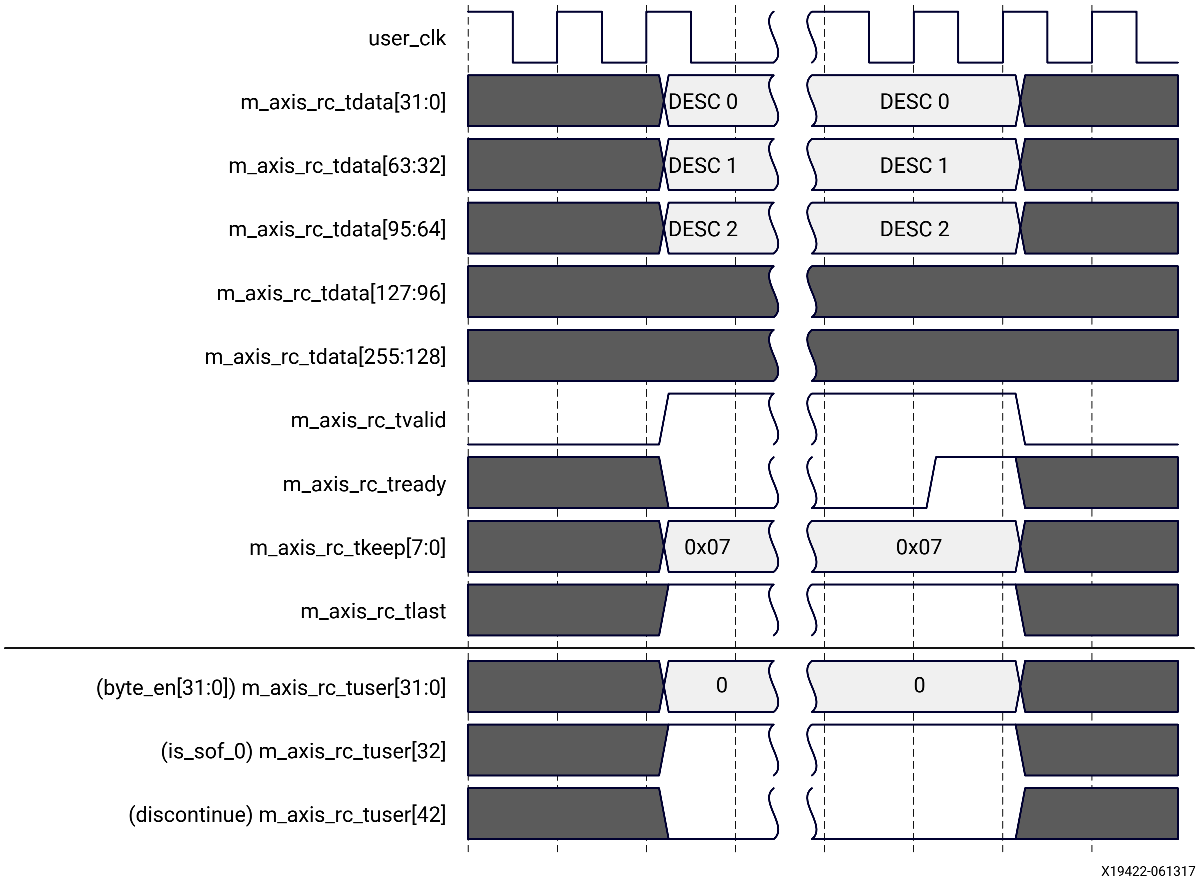


Figure 53: **Transfer of a Completion with no Data on the Requester Completion Interface (256-Bit Interface)**



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

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

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

The `m_axi_rc_tuser` bus provides several informational signals that can be used to simplify the logic associated with the user application side of the interface, or to support additional features. The is_`sof_0` signal is asserted in the first beat of every packet, when its descriptor is on the bus. The byte enable outputs `byte_en[31:0]` (one per byte lane) indicate the valid bytes in the payload. These signals are asserted only when a valid payload byte is in the corresponding lane (it is not asserted for descriptor or null bytes). The asserted byte enable bits are always contiguous from the start of the payload, except when payload size is 2 Dwords or less. For Completion payloads of two Dwords or less, the 1s on `byte_en` might not be contiguous. Another special case is that of a zero-length memory read, when the integrated block transfers a one-Dword payload with the `byte_en` bits all set to 0. Thus, the user logic can, in all cases, use the `byte_en` signals directly to enable the writing of the associated bytes into memory.

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

Figure 54: **Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode, 64-Bit Interface)**

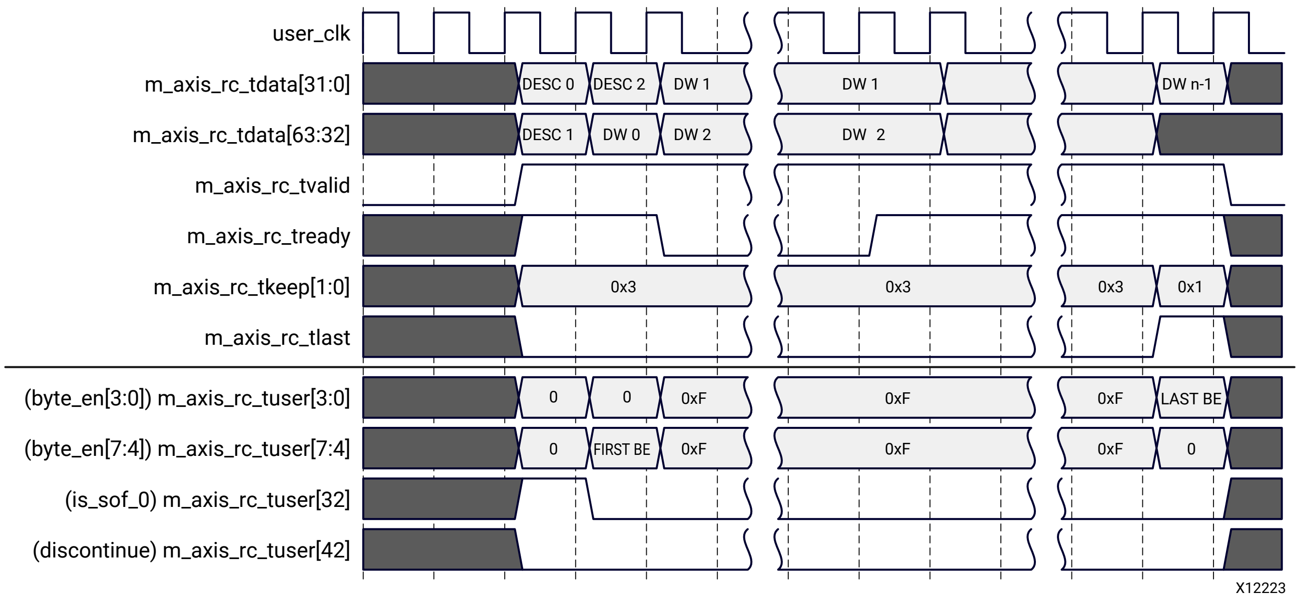


Figure 55: **Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode, 128-Bit Interface)**

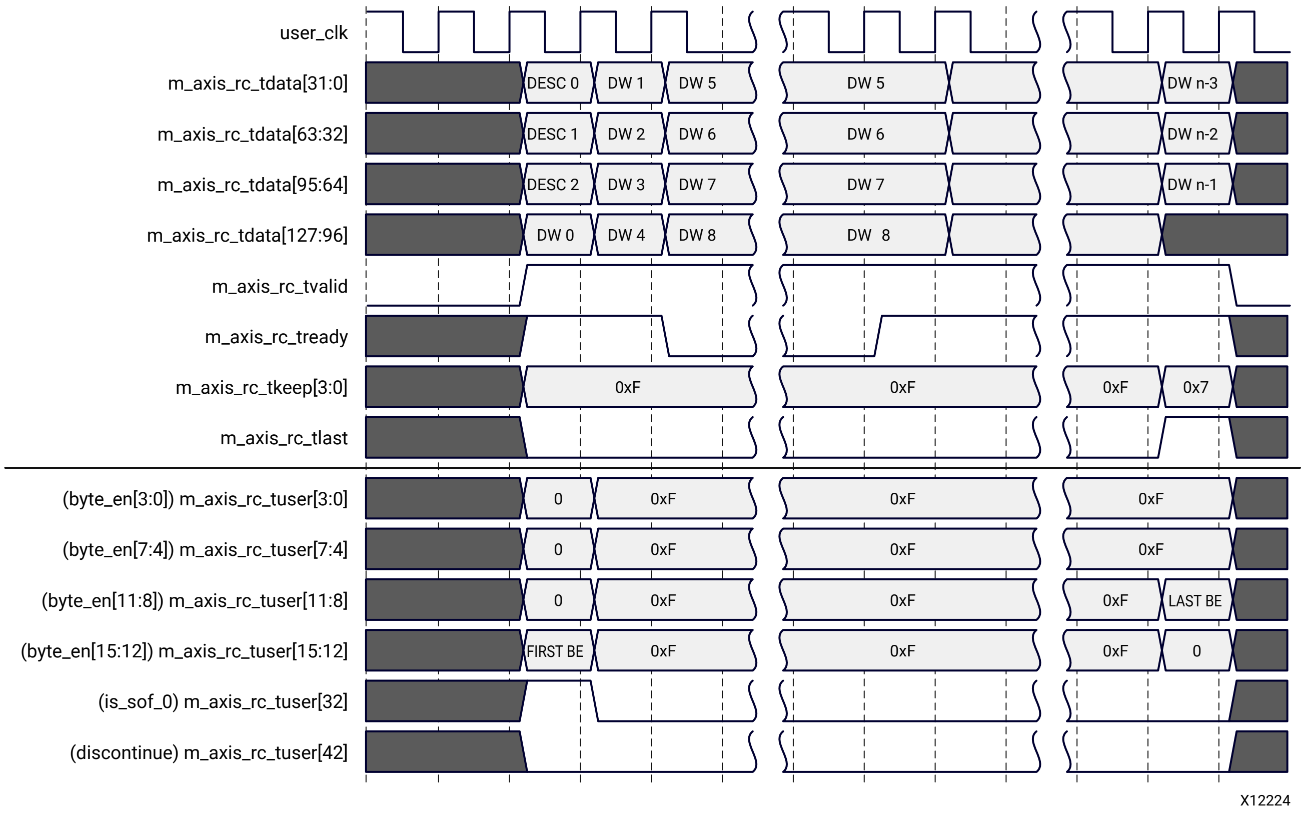
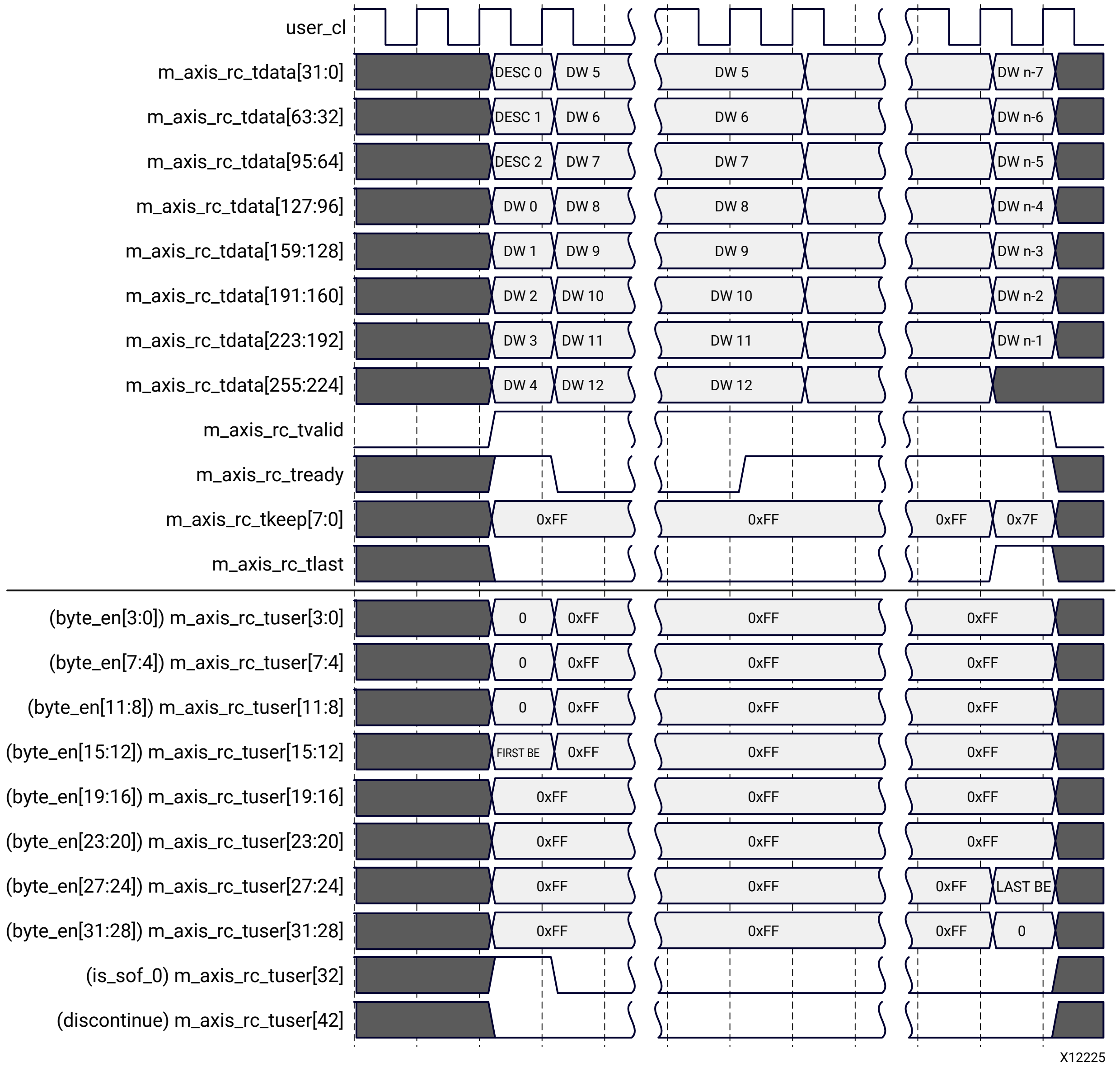


Figure 56: **Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode, 256-Bit Interface)**



The following timing diagrams illustrate the address-aligned transfer of a Completion TLP received from the link with an associated payload across the RC interface, when the interface width is configured as 64, 128, and 256 bits, respectively. In the example timing diagrams, the starting Dword address of the data block being transferred (as conveyed in bits [6:2] of the Lower Address field of the descriptor) is assumed to be ($m \times 8 + 1$), for an integer m. The size of the data block is assumed to be n Dwords, for some $\mathrm{n} = \mathrm{k} \times 32 + 28$, $\mathrm{k} > 0$. 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 57: **Transfer of a Completion with Data on the Requester Completion Interface (Address-Aligned Mode, 64-Bit Interface)**

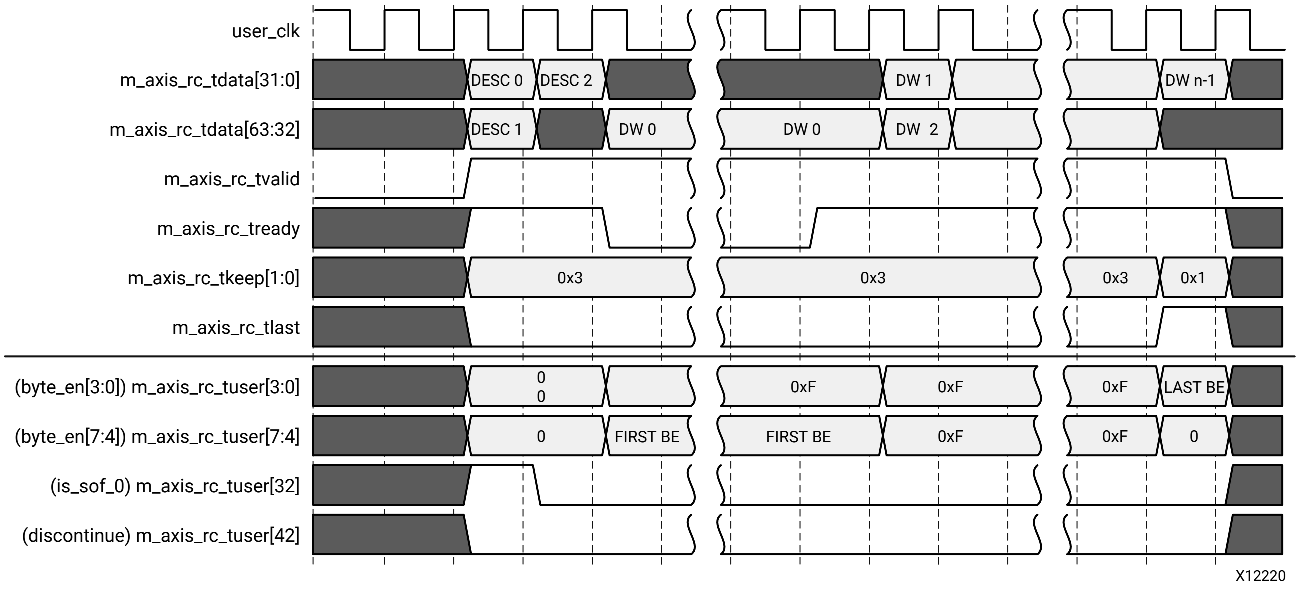


Figure 58: **Transfer of a Completion with Data on the Requester Completion Interface (Address-Aligned Mode, 128-Bit Interface)**

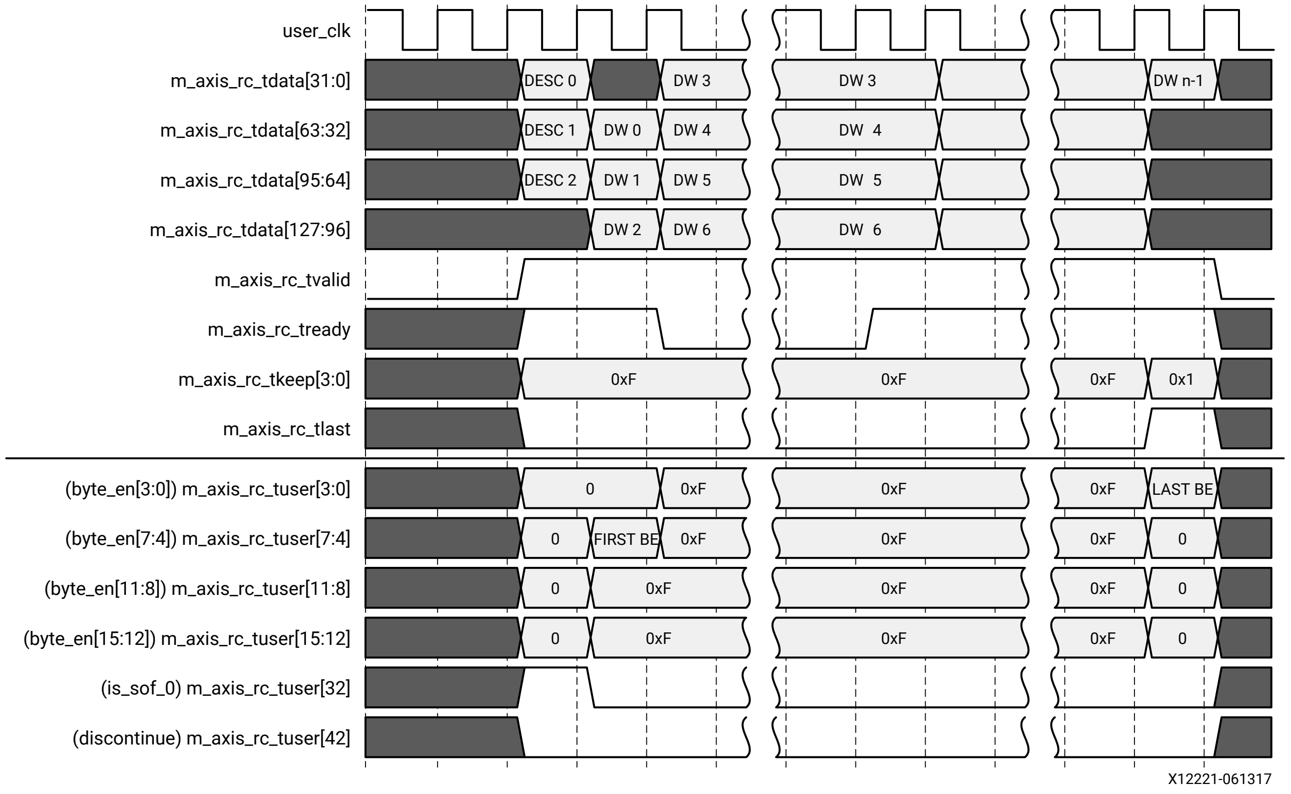
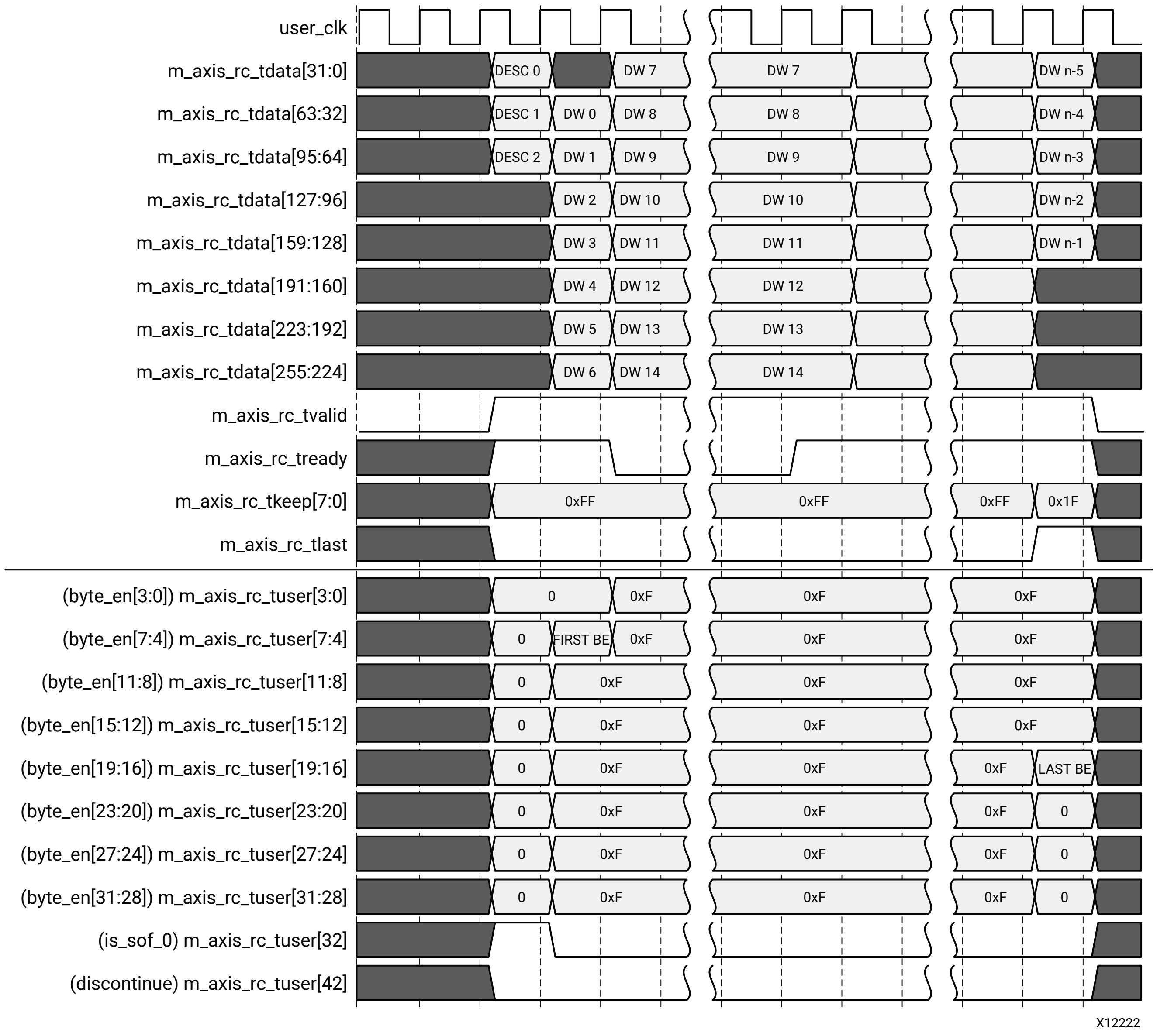


Figure 59: **Transfer of a Completion with Data on the Requester Completion Interface (Address-Aligned Mode, 256-Bit Interface)**



Straddle Option for 256-Bit Interface

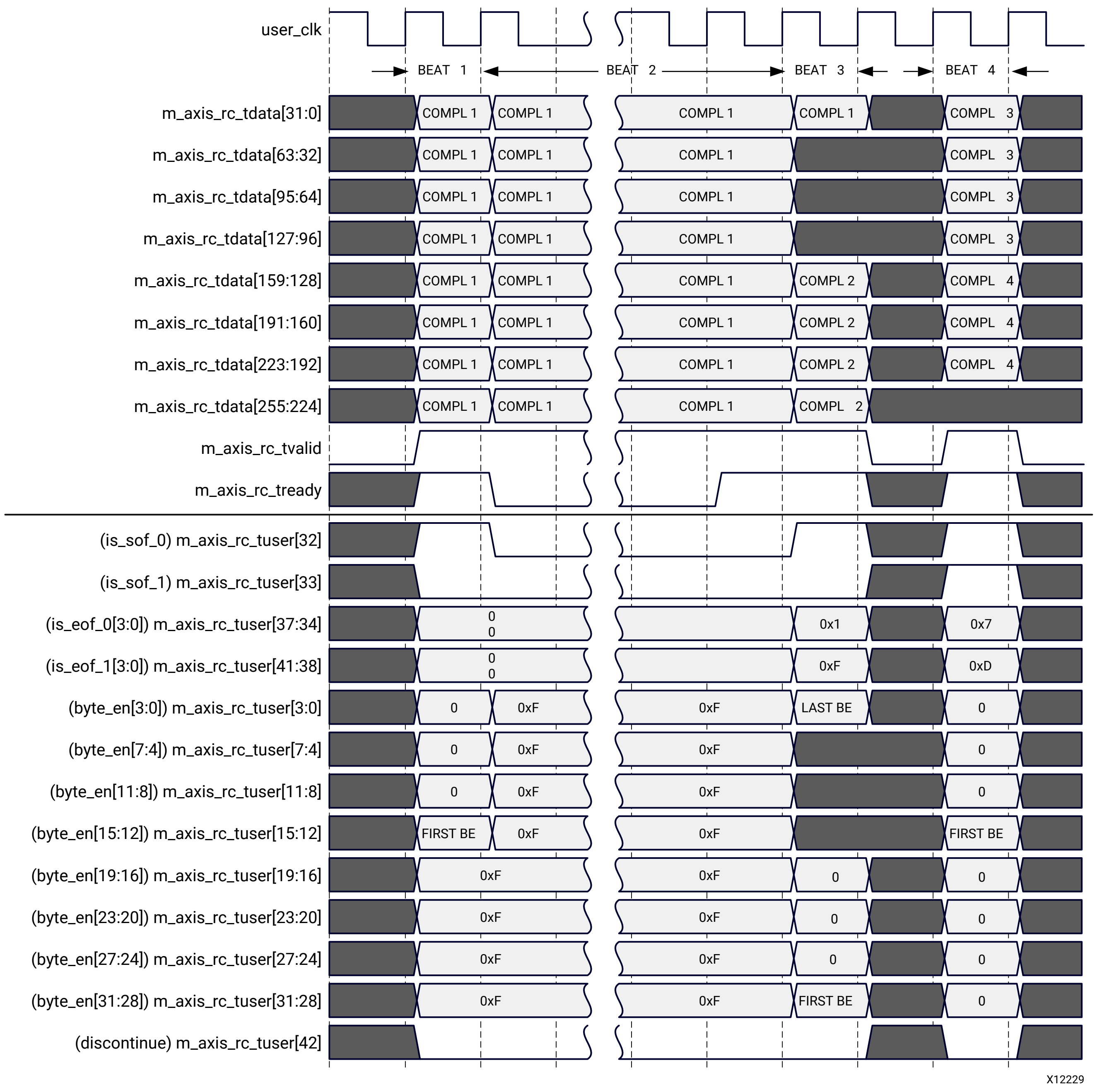
When the interface width is configured as 256 bits, the integrated block can start a new Completion transfer on the RC interface in the same beat when the previous Completion has ended on or before Dword position 3 on the data bus. The straddle option can be used only with the Dword-aligned mode.

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


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

Aborting a Completion Transfer

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

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

Handling of Completion Errors

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

Table 55: **Encoding of Error Codes**

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

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

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

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

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

512-Bit Completer Interface

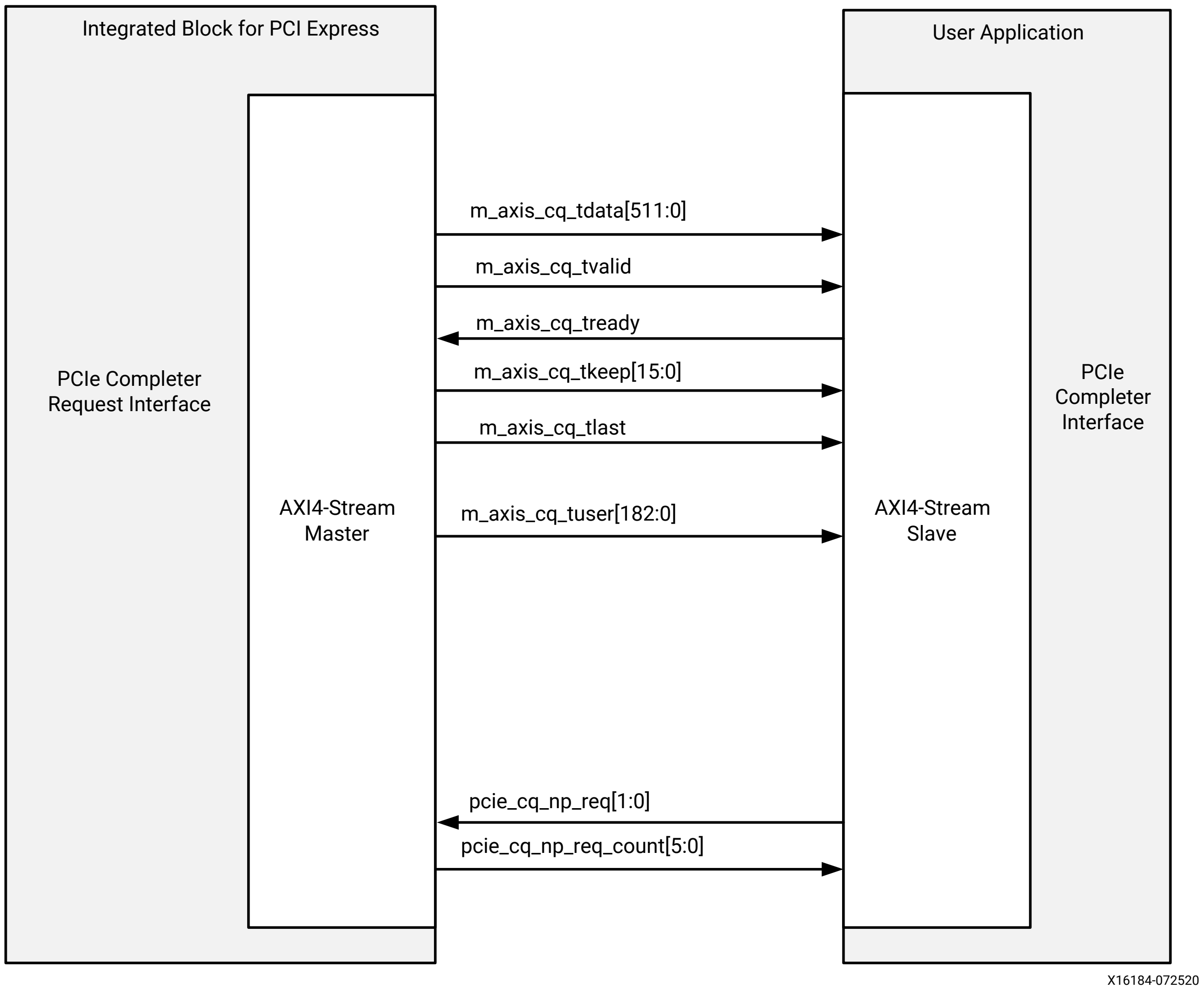
This section describes the operation of the completer interface in the user-side interfaces associated with the 512-bit AXI4-Stream Interface.

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

Completer Request Interface Operation (512-bits)

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

Figure 61: **Completer Request Interface Signals**



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

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

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

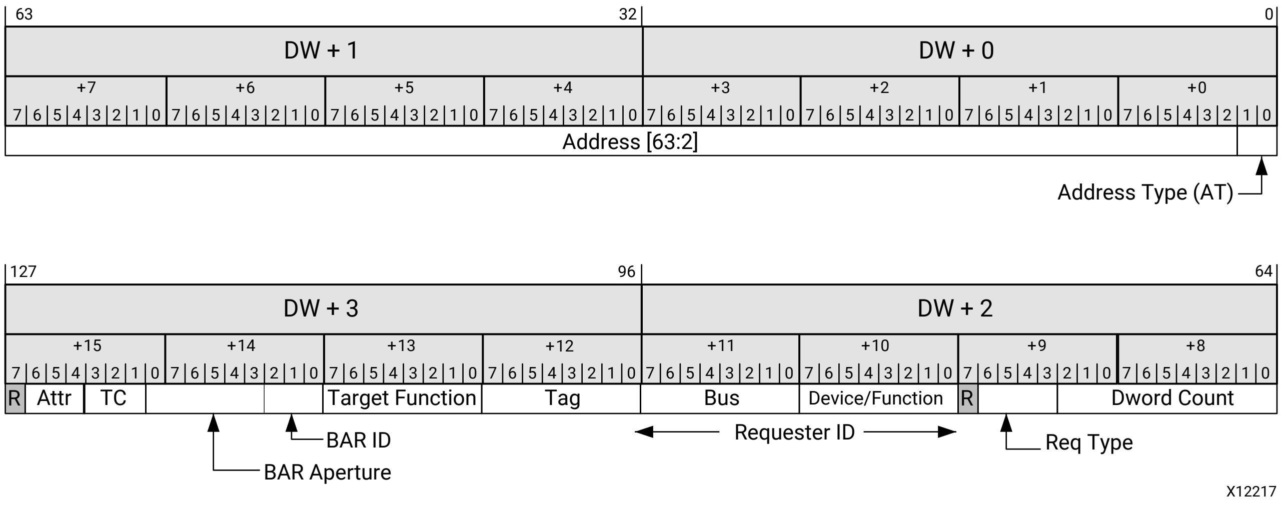
The interface also supports a straddle option that allows the transfer of up to two TLPs in the same beat across the interface. The straddle option can be used only with the Dword-aligned mode, and is not supported when using the 128-bit address aligned mode. The descriptions in the next sections assume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

Completer Request Descriptor Formats

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

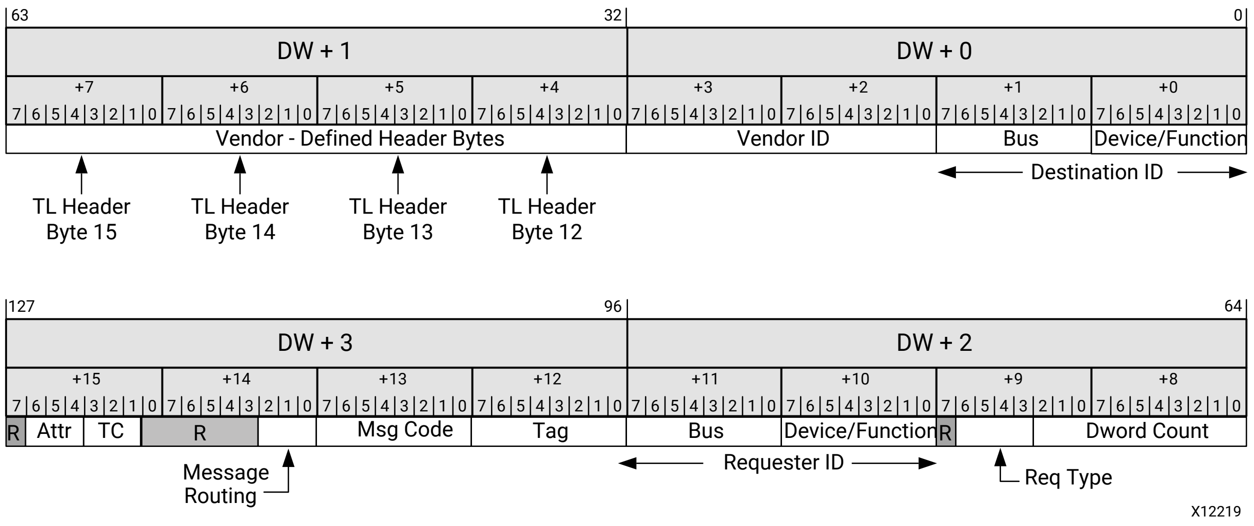
The format of the following figure applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request.

Figure 62: **Completer Request Descriptor Format for Memory, I/O, and Atomic Op Requests**



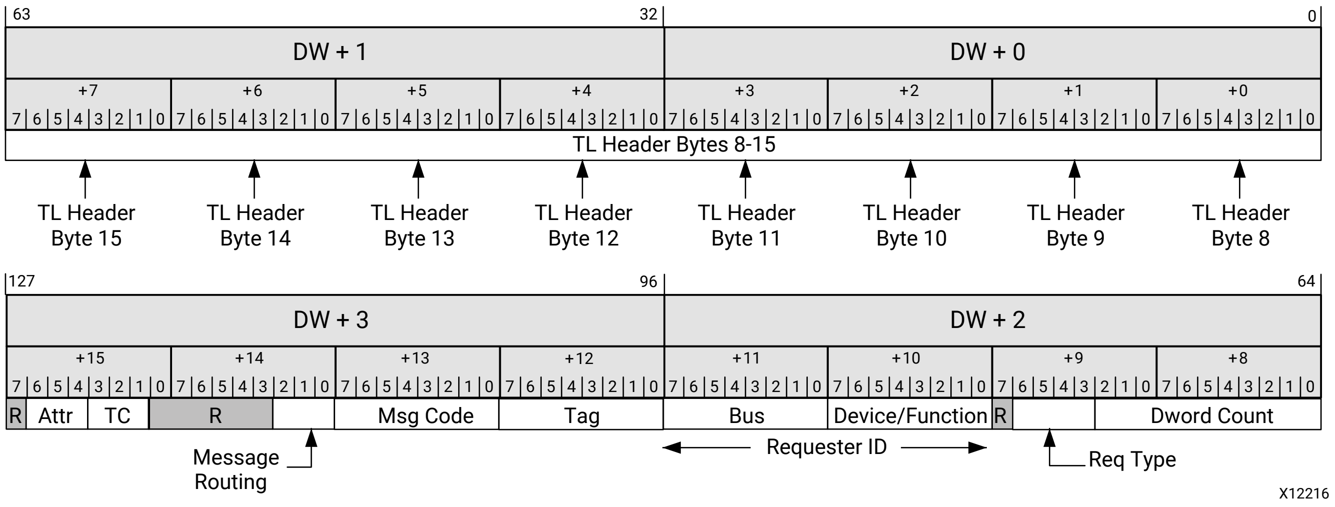
The format of the following figure is used for Vendor-Defined Messages (Type 0 or Type 1) only.

Figure 63: **Completer Request Descriptor Format for Vendor-Defined Messages**



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

Figure 64: **Completer Request Descriptor Format for ATS Messages**



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

Figure 65: **Completer Request Descriptor Format for All Other Messages**

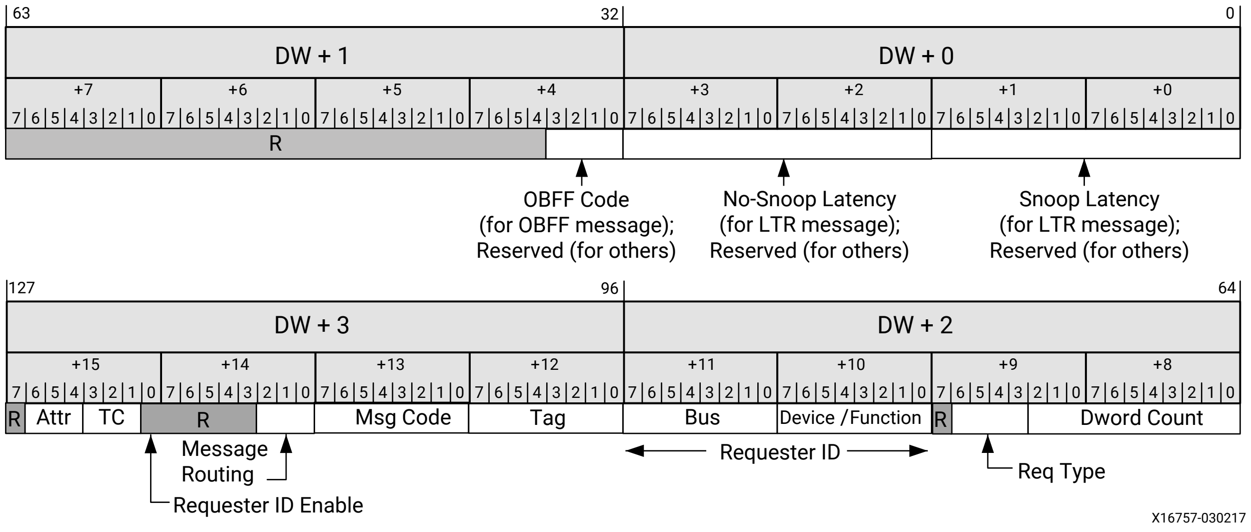


Table 56: **Completer Request Descriptor Fields**

Bit Index	Field Name	Description
1:0	Address Type	This field is defined for memory transactions and Atomic Operations only. It contains the AT bits extracted from the TL header of the request. • 00: Address in the request is un-translated • 01: Transaction is a Translation Request • 10: Address in the request is a translated address • 11: Reserved

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

Bit Index	Field Name	Description
63:2	Address	This field applies to memory, I/O and Atomic Op requests. It provides the address from the TL header. This is the address of the first Dword referenced by the request. The `First_BE` bits from `m_axis_cq_tuser` must be used to determine the byte-level address. When the transaction specifies a 32-bit address, bits [63:32] of this field is 0.
74:64	Dword Count	These 11 bits indicate the size of the block (in Dwords) to be read or written (for messages, size of the message payload). Its range is 0 – 256 Dwords. For I/O accesses, the Dword count is always 1. For a zero length memory read/write request, the Dword count is 1, with the `First_BE` bits set to all zeroes.
78:75	Request Type	Identifies the transaction type. The transaction types and their encodings are listed in Table 57: Transaction Types.
95:80	Requester ID	PCI Requester ID associated with the request. With the legacy interpretation of RIDs, these 16 bits are divided into an 8-bit bus number [95:88], 5-bit device number [87:83], and 3-bit function number [82:80]. When ARI is enabled, bits [95:88] carry the 8-bit bus number and [87:80] provide the function number. When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the core with the completion data.
103:96	Tag	PCIe Tag associated with the request. When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the core with the completion data. This field can be ignored for memory writes and messages.
111:104	Target Function	This field is defined for memory, I/O and Atomic Op requests only. It provides the function number the request is targeted at, determined by the BAR check. When ARI is in use, all 8 bits of this field are valid. Otherwise, only bits [106:104] are valid.
114:112	BAR ID	This field is defined for memory, I/O and Atomic Op requests only. It provides the matching BAR number for the address in the request. • 000 = BAR 0 (VF-BAR 0 for VFs) • 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 ***Note***: In Root Port (RP) mode, BAR ID is always 000. For 64-bit transactions, the BAR number is given as the lower address of the matching pair of BARs (that is, 0, 2 or 4).
120:115	BAR Aperture	This 6-bit field is defined for memory, I/O and Atomic Op requests only. It provides the aperture setting of the BAR matching the request. This information is useful in determining the bits to be used by the user in addressing its memory or I/O space. For example, a value of 12 indicates that the aperture of the matching BAR is 4K, and the user can therefore ignore bits [63:12] of the address. For VF BARs, the value provided on this output is based on the memory space consumed by a single VF covered by the BAR.

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

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

Table 57: **Transaction Types**

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

Completer Memory Write Operation

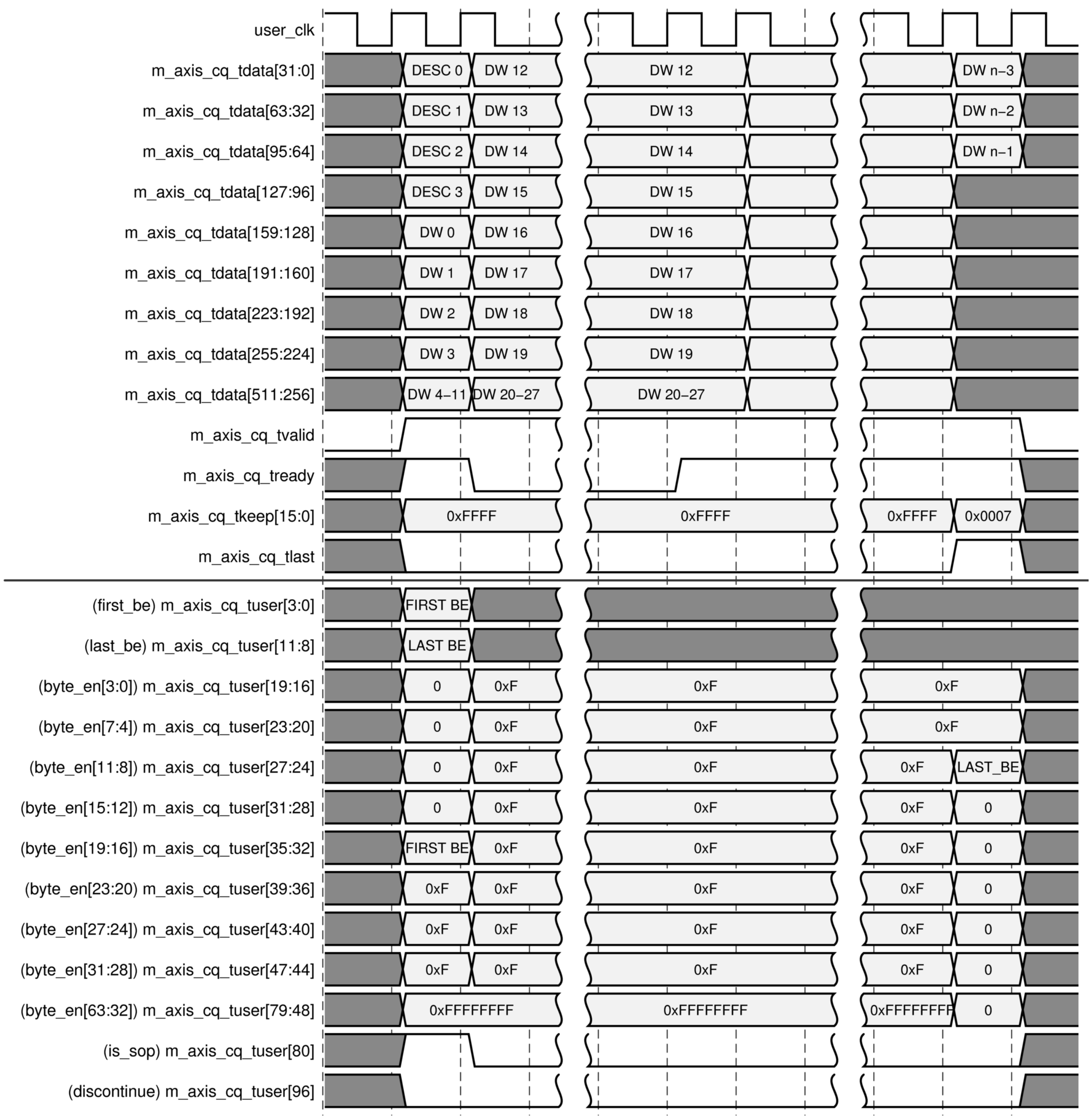
The following figure illustrates the Dword-aligned transfer of a memory write TLP received from the link across the completer request interface. For the purpose of illustration, the starting Dword address of the data block being written into user memory is assumed to be (*m**16 +3), for some integer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

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

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

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

Figure 66: **Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode)**


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

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

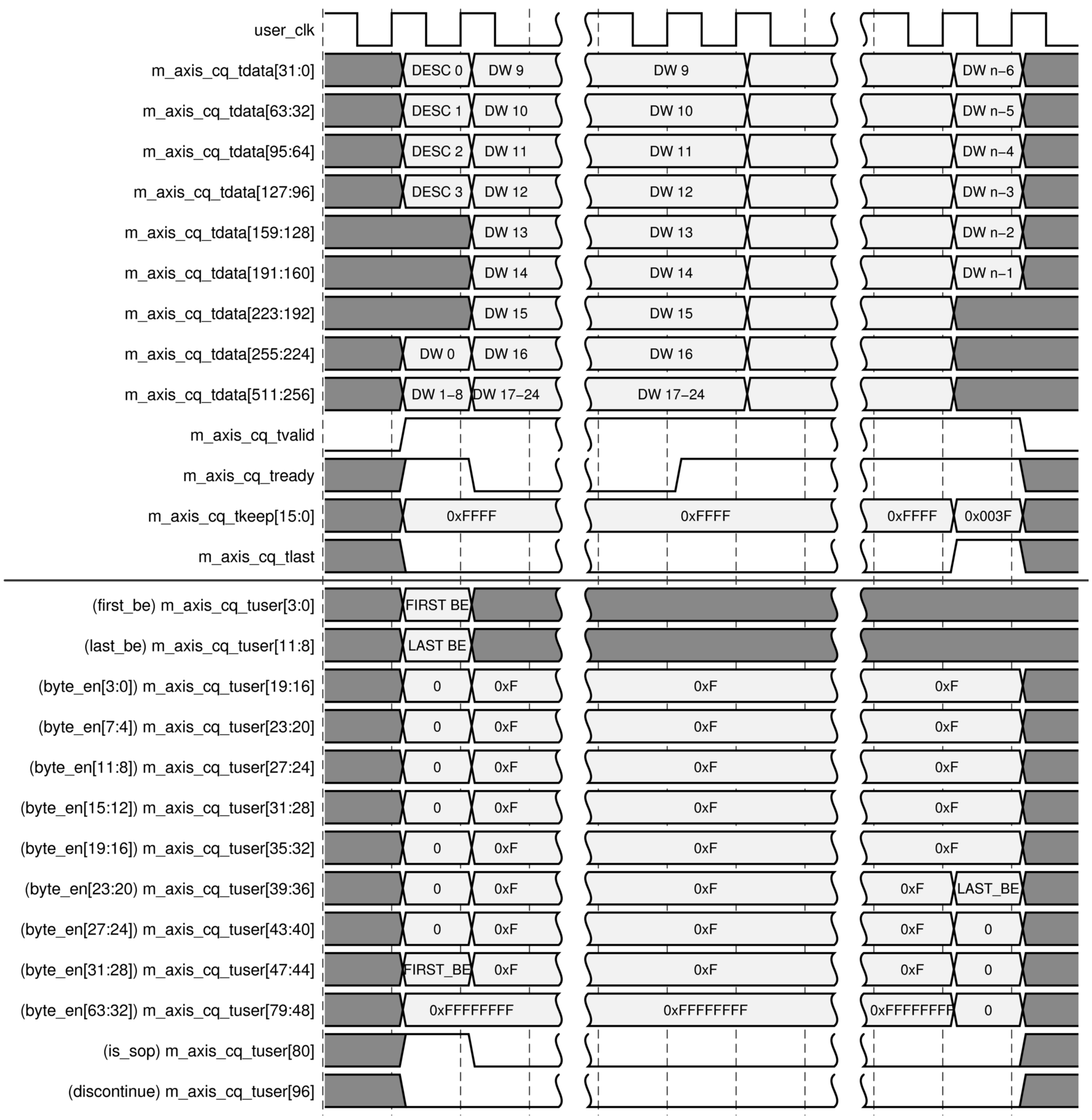
In the Dword-aligned mode, there can be a gap of zero, one, two, or three byte positions between the end of the descriptor and the first payload byte, based on the address of the first valid byte of the payload. The actual position of the first valid byte in the payload can be determined either from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

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

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

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

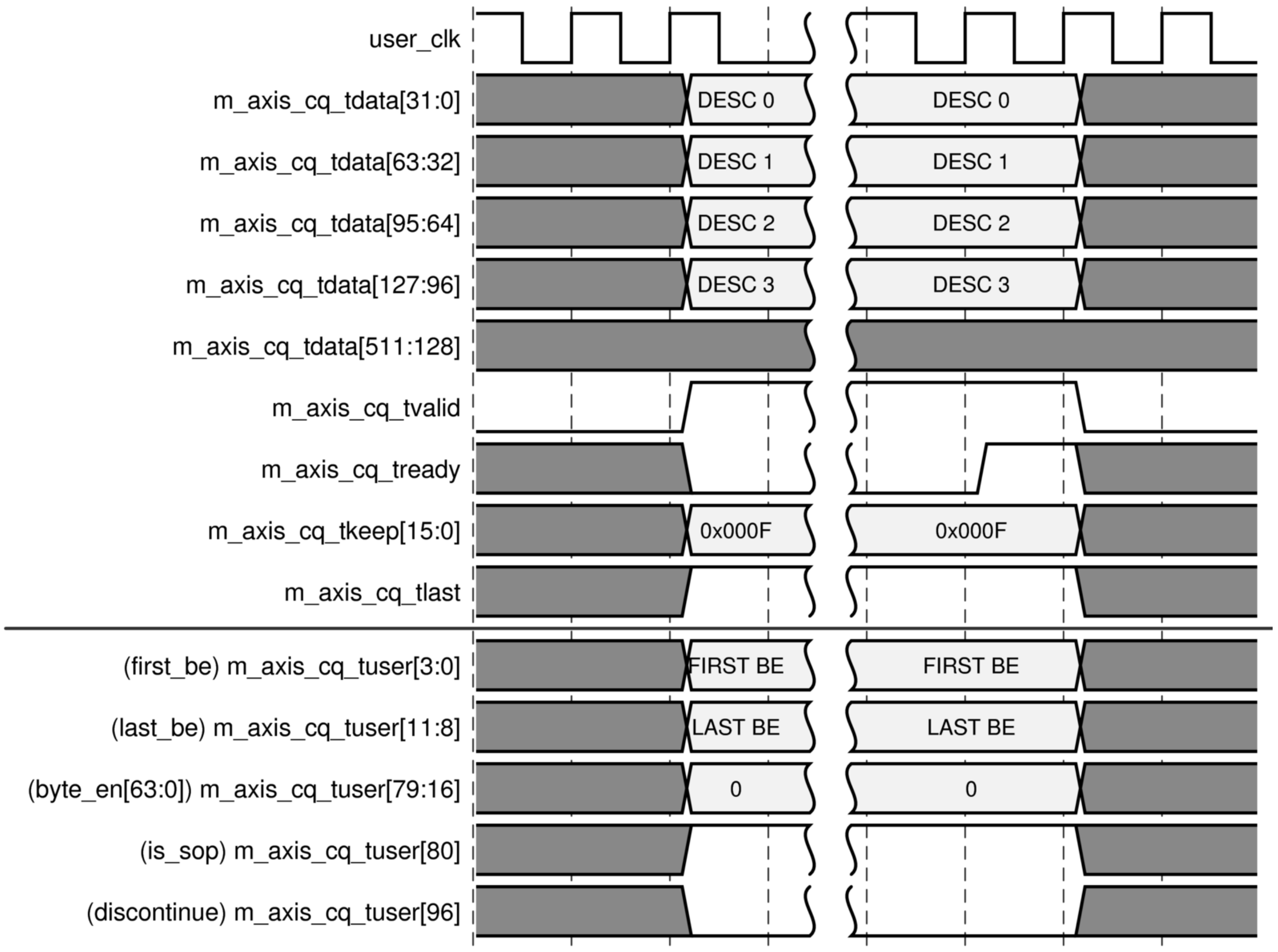
Figure 67: **Memory Write Transaction on the Completer Request Interface (128-bit Address Aligned Mode)**


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

Completer Memory Read Operation

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

Figure 68: **Memory Read Transaction on the Completer Request Interface**



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

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

I/O Write Operation

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

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

I/O Read Operation

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

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

Atomic Operations on the Completer Request Interface

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

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

Message Requests on the Completer Request Interface

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

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

Table 58: **AXISTEN_IF_ENABLE_MSG_ROUTE Attribute Bit Descriptions**

Bit Index	Message Type
0	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

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

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

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

Aborting a Transfer

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

Selective Flow Control for Non-Posted Requests

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

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

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

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

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

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

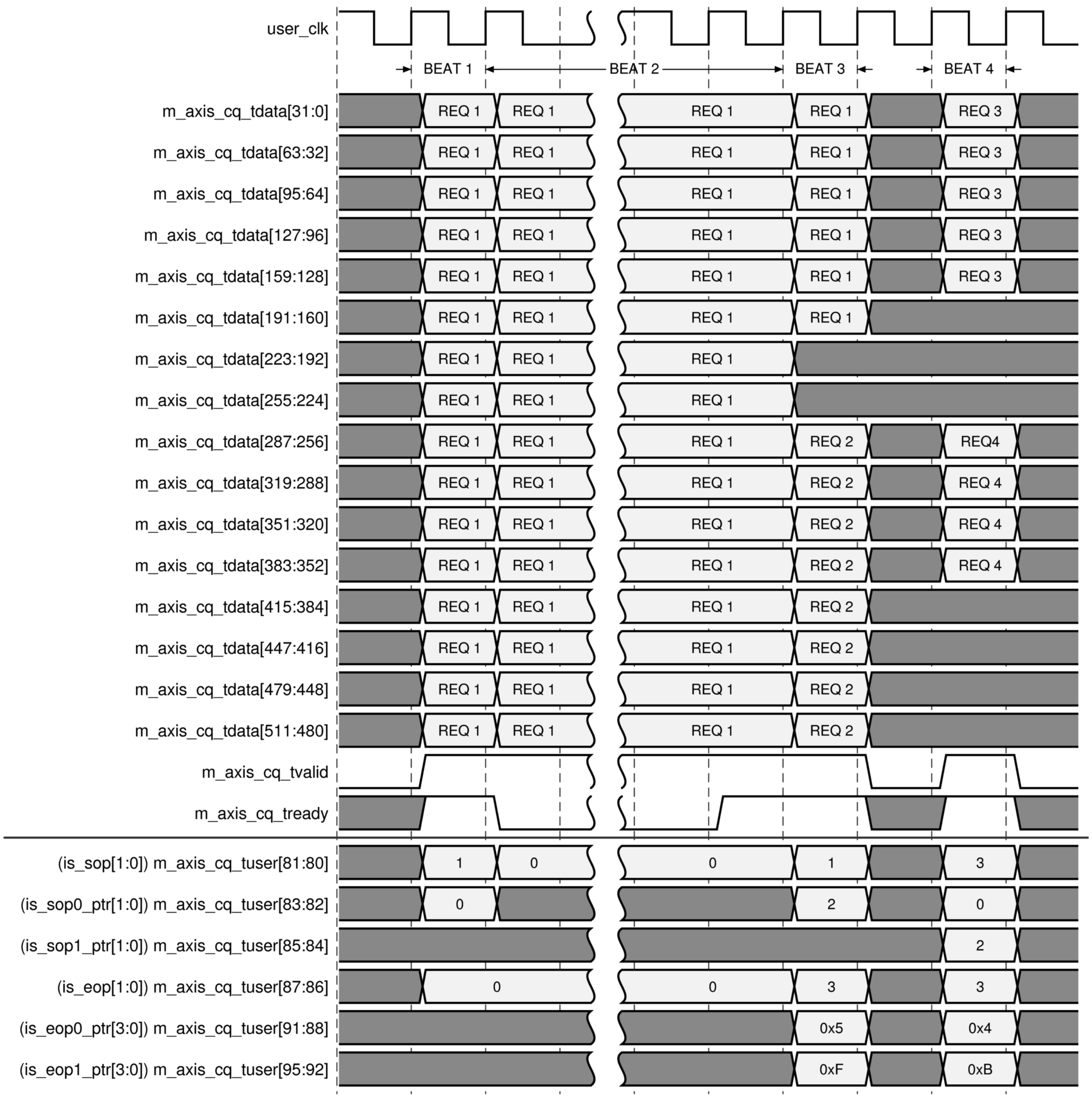
Straddle Option on CQ Interface

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

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

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

Figure 69: **Transfer of Request TLPs on the Completer Request Interface with the Straddle Option Enabled**

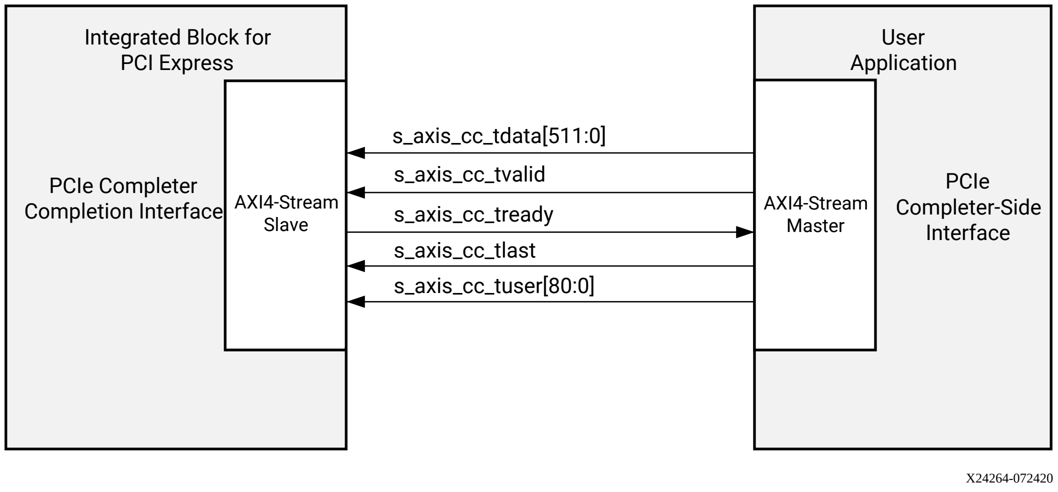


The previous figure illustrates the transfer of four request TLPs on the completer request interface when the straddle option is enabled. For all TLPs, the first Dword of the payload always follows the descriptor without any gaps. The first request TLP (REQ 1) starts at Dword position 0 of Beat 1 and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

Completer Completion Interface Operation (512-bits)

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

Figure 70: **Completer Completion Interface Signals**



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

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

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

Completer Completion Descriptor Format

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

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

Figure 71: **Completer Completion Descriptor Format**

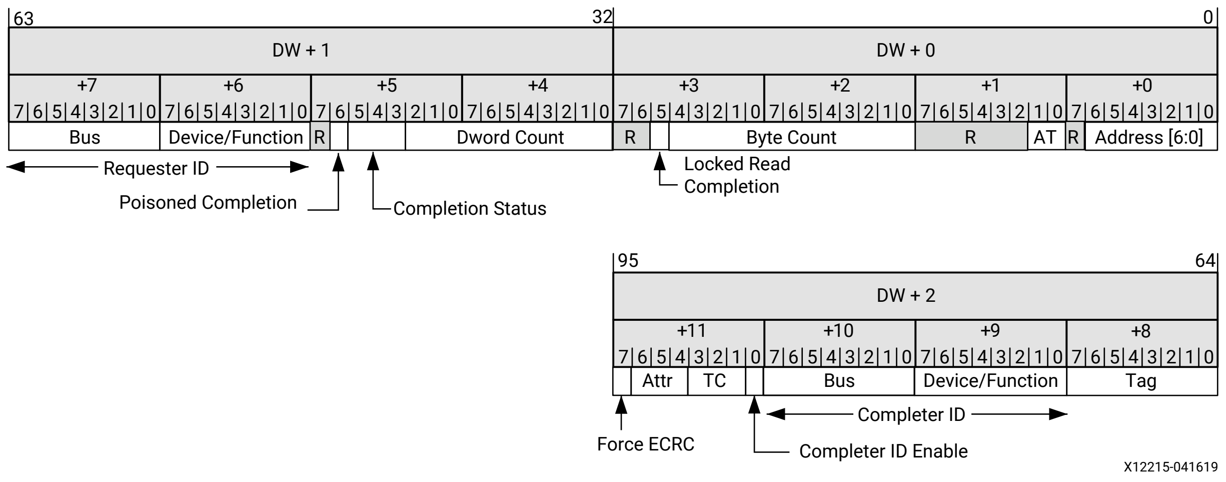


Table 59: **Completer Completion Descriptor Fields**

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

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

Bit Index	Field Name	Description
28:16	Byte Count	These 13 bits can have values in the range of 0 – 4,096 bytes. If a Memory Read Request is completed using a single Completion, the Byte Count value indicates Payload size in bytes. This field must be set to 4 for I/O read Completions and I/O write Completions. The byte count must be set to 1 while sending a Completion for a zero-length memory read, and a dummy payload of 1 Dword must follow the descriptor. For each Memory Read Completion, the Byte Count field must indicate the remaining number of bytes required to complete the Request, including the number of bytes returned with the Completion. If a Memory Read Request is completed using multiple Completions, the Byte Count value for each successive Completion is the value indicated by the preceding Completion minus the number of bytes returned with the preceding Completion
29	Locked Read Completion	This bit must be set when the Completion is in response to a Locked Read request. It must be set to 0 for all other Completions.
42:32	Dword Count	These 11 bits indicate the size of the payload of the current packet in Dwords. Its range is 0 – 1K Dwords. This field must be set to 1 for I/O read Completions and 0 for I/O write Completions. The Dword count must be set to 1 while sending a Completion for a zero-length memory read. The Dword count must be set to 0 when sending a UR or CA Completion. In all other cases, the Dword count must correspond to the actual number of Dwords in the payload of the current packet.
45:43	Completion Status	These bits must be set based on the type of Completion being sent. The only valid settings are: • 000: Successful Completion • 001: Unsupported Request (UR) • 100: Completer Abort (CA)
46	Poisoned Completion	This bit can be used by the user logic to poison the Completion TLP being sent. This bit must be set to 0 for all Completions, except when the user logic has detected an error in the block of data following the descriptor and wants to communicate this information using the Data Poisoning feature of PCI Express.
63:48	Requester ID	PCI Requester ID associated with the request (copied by the user logic from the request).
71:64	Tag	PCIe Tag associated with the request (copied by the user logic from the request).

Table 59: **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 [79:72] must be set to the Completer Function number. ARI disabled: • Bits [74:72] must be set to the Completer Function number. • Bits [79:75] are not used Upstream Port for Switch use case (Endpoint mode is selected within the IP): ARI enabled: • Bits [79:72] must be set to the Completer Function number. ARI disabled: • Bits [74:72] must be set to the Completer Function number. • Bits [79:75] are not used if the Completion is originating from the switch itself. These bits must be set to the Completer Device number where the Completion was originated if the switch is relaying the Completion (Completer is external to the switch). This is used with Completer ID Enable bit in the descriptor. Root Port mode (Downstream Port): ARI enabled: Bits [79:72] must be set to the Completer Function number. ARI disabled: • Bits [74:72] must be set to the Completer Function number. • Bits [79:75] must be set to the Completer Device number. This is used with Completer ID Enable bit in the descriptor.

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

Bit Index	Field Name	Description
87:80	Completer Bus Number	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 59: **Completer Completion Descriptor Fields** *(cont'd)*

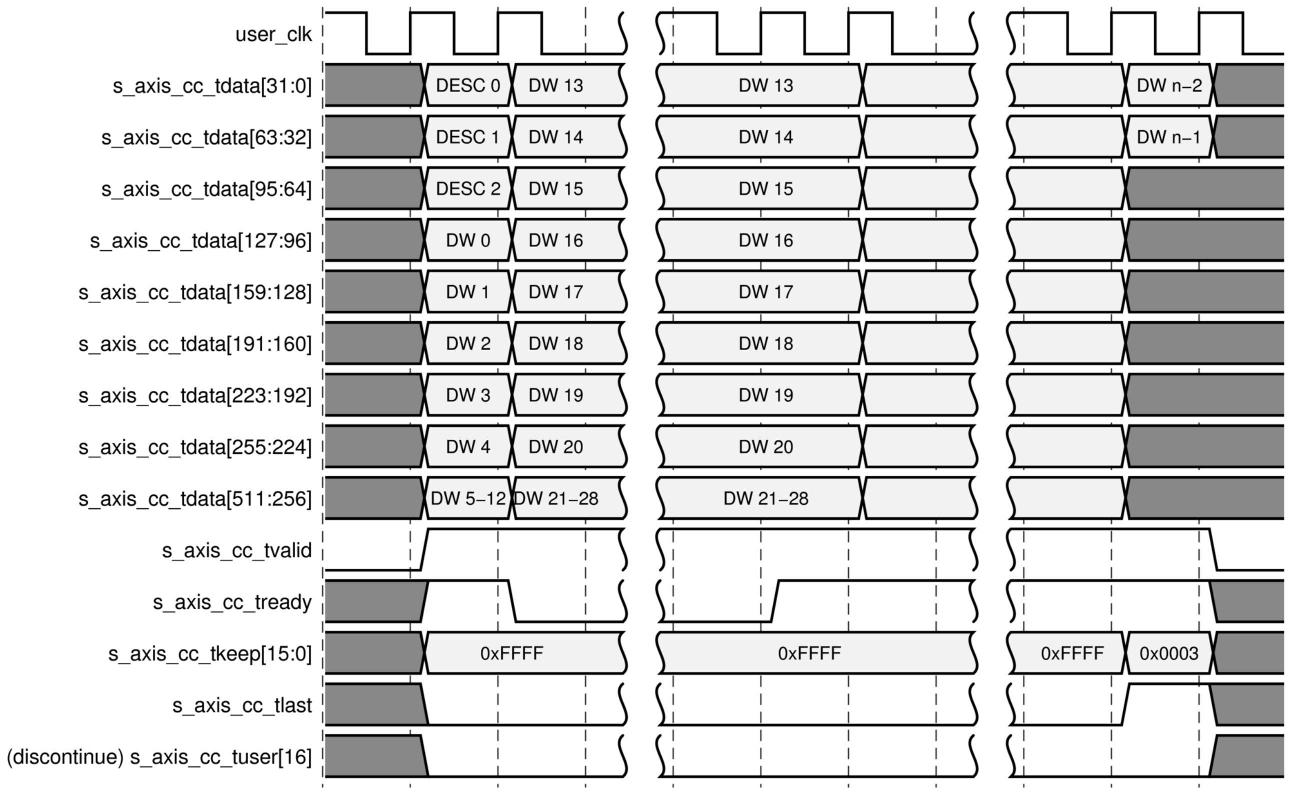
Bit Index	Field Name	Description
88	Completer ID Enable	1'b1: The client supplies Bus, Device, and Function numbers in the descriptor to be populated as the Completer ID field in the TLP header. 1'b0: IP uses Bus and Device numbers captured from received Configuration requests and the client supplies 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 logic must copy this value from the TC field of the associated request descriptor.
94:92	Attributes	PCIe attributes associated with the request (copied from the request). Bit 92 is the No Snoop bit, bit 93 is the Relaxed Ordering bit, and bit 94 is the ID-Based Ordering bit.
95	Reserved	Reserved for future use.

Completions with Successful Completion (SC) Status

The user logic must return a Completion to the completer completion interface of the core for every Non-Posted request it receives from the completer request interface. When the request completes with no errors, the user logic must return a Completion with Successful Completion (SC) status. Such a Completion might contain a payload, depending on the type of request. Furthermore, the data associated with the request can be broken up into multiple Split Completions when the size of the data block exceeds the maximum payload size configured. User logic is responsible for splitting the data block into multiple Split Completions when needed. The user logic must transfer each Split Completion over the completer completion interface as a separate AXI4-Stream packet, with its own 12-byte descriptor.

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

Figure 72: **Transfer of a Normal Completion on the Completer Completion Interface (512-bit Interface, Dword-Aligned Mode)**



The previous figure illustrates the Dword-aligned transfer of a Completion from the user logic across the completer completion interface. In this case, the first Dword of the payload starts immediately after the descriptor. When the data block is not a multiple of 4 bytes, or when the start of the payload is not aligned on a Dword address boundary, the user application must add null bytes to align the start of the payload on a Dword boundary and make the payload a multiple of Dwords. For example, when the data block starts at byte address 7 and has a size of 3 bytes, the user logic must add 3 null bytes before the first byte and two null bytes at the end of the block to make it 2 Dwords long. Also, in the case of non-contiguous reads, not all bytes in the data block returned are be valid. In that case, the user application must return the valid bytes in the proper positions, with null bytes added in gaps between valid bytes, when needed. The interface does not have any signals to indicate the valid bytes in the payload. This is not required, as the requester is responsible for keeping track of the byte enables in the request and discarding invalid bytes from the Completion.

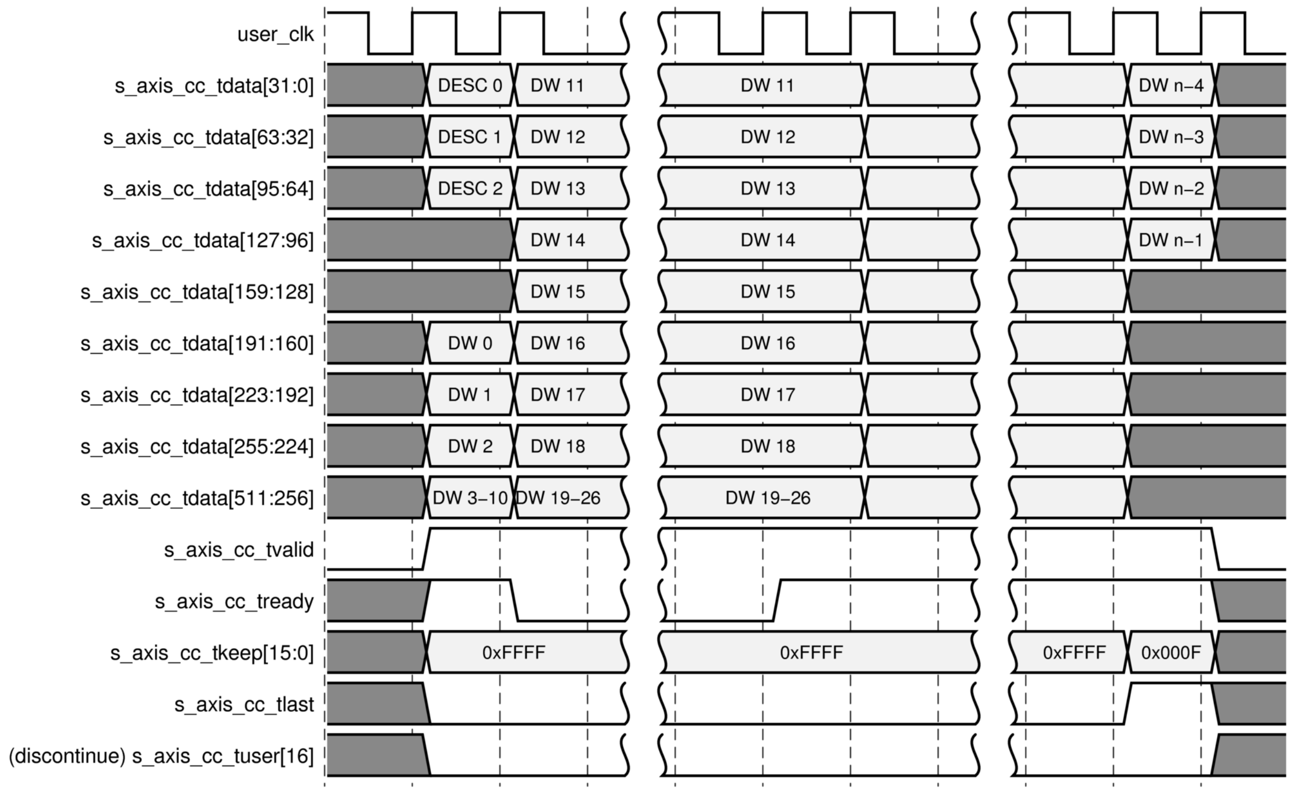
In the Dword-aligned mode, the transfer starts with the 12 descriptor bytes, followed immediately by the payload bytes. The user application must keep the signal `s_axis_cc_tvalid` asserted over the duration of the packet. The core treats the deassertion of `s_axis_cc_tvalid` during the packet transfer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

The user application must also assert the signal `s_axis_cc_tlast` in the last beat of the packet. The core are by pull down `s_axis_cc_tready` in any cycle if it is not ready to accept data. The user application must not change the values on `s_axis_cc_tdata` and `s_axis_cc_tlast` during the transfer when the core has deasserted `s_axis_cc_tready`.

In the 128-bit address aligned mode, the delivery of the payload must always start in the second128-bit quarter of the 512-bit word, following the descriptor in the first quarter. That is, if the first byte of the descriptor is on byte lane 0, the payload must start on one of the byte lanes 16 – 31. Within its 128-bit quarter, the offset of the first payload byte must correspond to the least significant bits of the Lower Address field setting in the corresponding descriptor.

The following timing diagram illustrates the 128-bit address-aligned transfer of a memory read Completion across the completer completion interface. For the purpose of illustration, the starting Dword address of the data block being transferred (as conveyed in bits [6:2] of the Lower Address field of the descriptor) is assumed to be (*m*16+1*), for some integer *m*. The size of the data block is assumed to be n Dwords, for some n = k*16 - 1, for some k > 1.

Figure 73: **Transfer of a Normal Completion on the Completer Completion Interface (128-bit Address Aligned Mode)**



Aborting a Completion Transfer

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

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

The discontinue signal can be asserted only when `s_axis_cc_tvalid` is active-High. The core samples this signal when `s_axis_cc_tvalid` and `s_axis_cc_tready` are both active-High. Thus, once asserted, it should not be deasserted until `s_axis_cc_tready` is active-High.

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

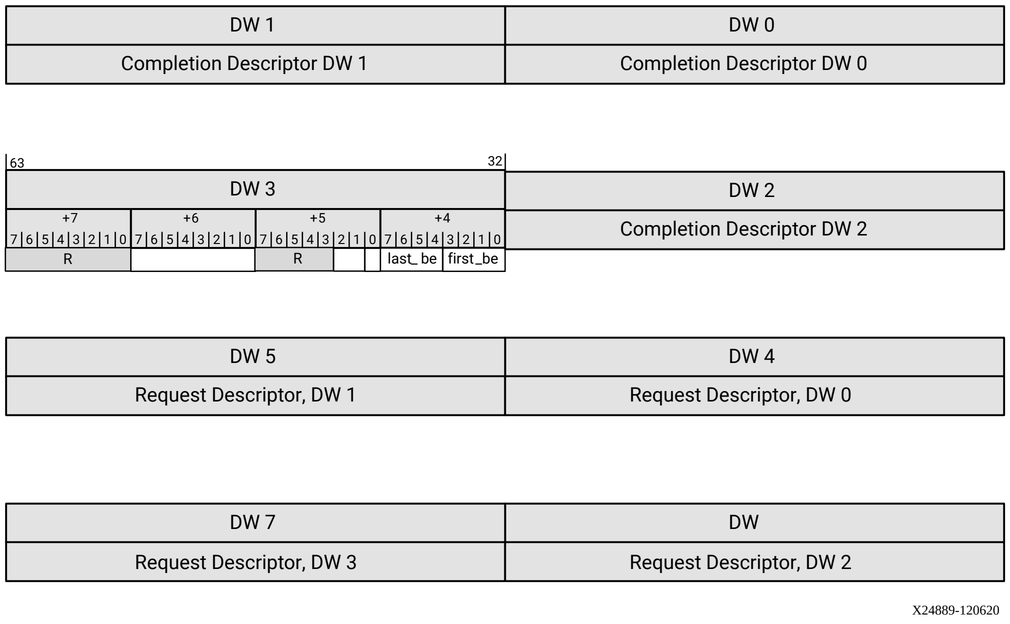
Completions with Error Status (UR and CA)

When responding to a request received on the completer request interface with an Unsupported Request (UR) or Completion Abort (CA) status, the user logic must send a 3-Dword completion descriptor in the format of the *Completer Completion Descriptor Format* figure in Completer Completion Descriptor Format, followed by five additional Dwords containing information on the request that generated the Completion. These five Dwords are necessary for the core to log information about the request in its AER header log registers.

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

- The first three Dwords contain the completion descriptor in the format of the *Completer Completion Descriptor Format* figure in Completer Completion Descriptor Format.
- The fourth Dword contains the state of the following signals in `m_axis_cq_tuser`, copied from the request:
    - The First Byte Enable bits `first_be[3:0]` in `m_axis_cq_tuser`.
    - The Last Byte Enable bits `last_be[3:0]` in `m_axis_cq_tuser`.
    - The four Dwords of the request descriptor received from the core with the request.

Figure 74: **Composition of the AXI4-Stream Packet for UR and CA Completions**



Straddle Option on CC Interface

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

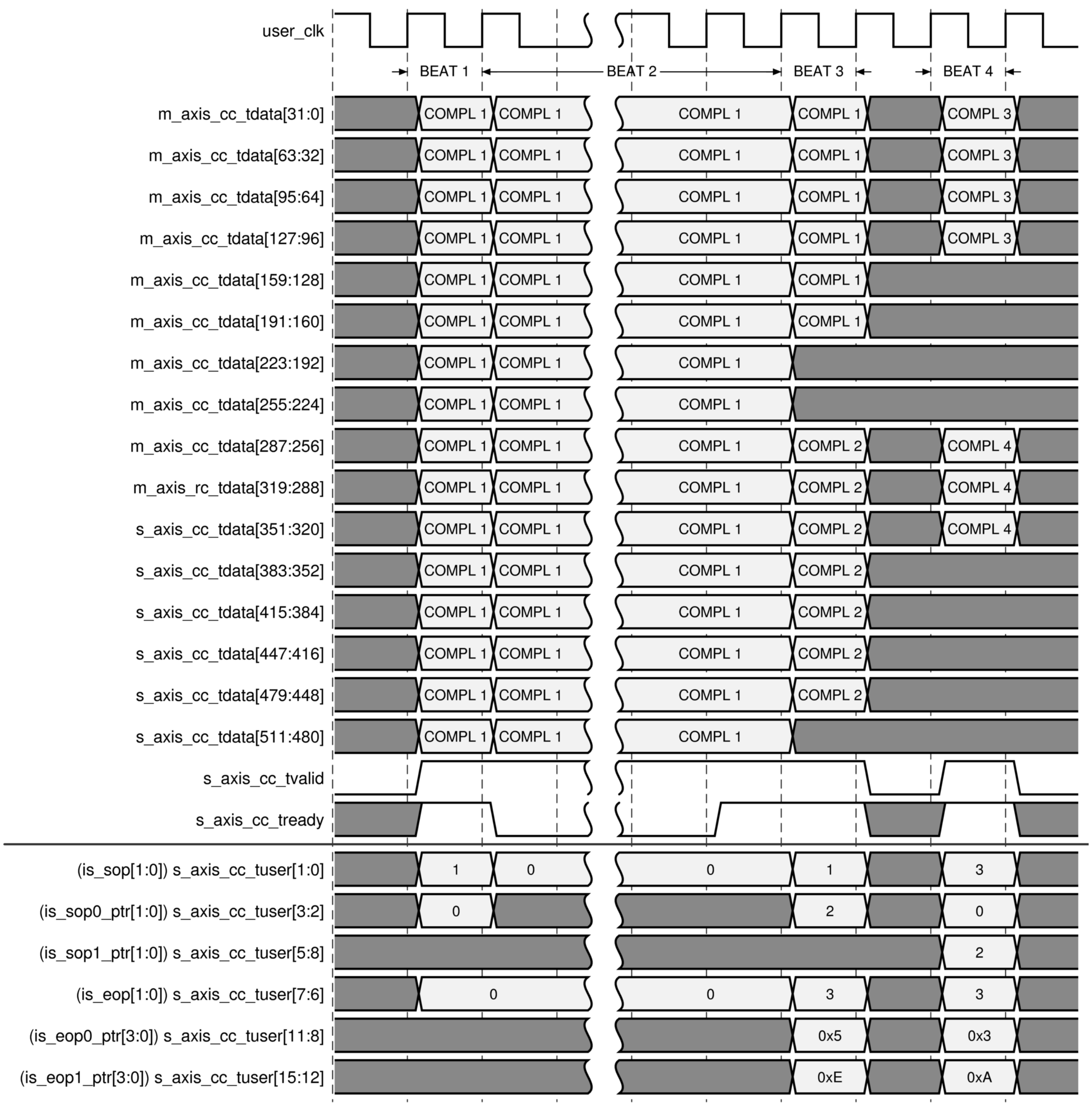
When the straddle option is enabled, Completion TLPs are transferred on the AXI4-Stream interface as a continuous stream, with no packet boundaries. Thus, the signals `m_axis_cc_tkeep` and `m_axis_cc_tlast` are not useful in determining the boundaries of TLPs delivered on the interface. Instead, delineation of TLPs is performed using the following signals provided within the `m_axis_cc_tuser` bus.

- `is_sop[0]`: This input must be set High in a beat when there is at least one Completion TLP starting in the beat. The position of the first byte of the descriptor of this TLP is determined as follows:
    - If the previous TLP ended before this beat, the first byte of the descriptor is in byte lane 0.
    - If a previous TLP is continuing in this beat, the first byte of this descriptor is in byte lane 32. This is possible only when the previous TLP ends in the current beat, that is when `is_eop[0]` is also set.
- `is_sop0_ptr[1:0]`: When `is_sop[0]` is set, this field must indicate the offset of the first Completion TLP starting in the current beat. Valid settings are `2'b00` (TLP starting at Dword 0) and `2'b10` (TLP starting at Dword 8).

- `is_sop[1]`: This input must be set High in a beat when there are two Completion TLPs starting in the same beat. The first TLP must always start at byte position 0 and the second TLP at byte position 32. The user application are start a second TLP at byte position 32 only if the previous TLP ended before byte position 32 in the same beat, that is only if `is_eop[0]` is also set in the same beat.
- `is_sop1_ptr[1:0]`: When `is_sop[1]` is set, this field must provide the offset of the second TLP starting in the current beat. Its only valid setting is `2'b10` (TLP starting at Dword 8).
- `is_eop[0]`: This input is used to indicate the end of a Completion TLP. Its assertion signals that there is at least one TLP ending in this beat.
- `is_eop0_ptr[3:0]`: When `is_eop[0]` is asserted, `is_eop0_ptr[3:0]` must provide the offset of the last Dword of the corresponding TLP ending in this beat.
- `is_eop[1]`: This input is set High when there are two TLPs ending in the current beat. is_eop[1] can be set only when the signals `is_eop[0]` and `is_sop[0]` are also be High in the same beat.
- `is_eop1_ptr[3:0]`: When `is_eop[1]` is asserted, `is_eop1_ptr[3:0]` must provide the offset of the last Dword of the second TLP ending in this beat. Because the second TLP can start only on byte lane 32, it can only end at a byte lane in the range 43-63. Thus the offset `is_eop1_ptr[3:0]` can only take a value in the range 10-15.

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

Figure 75: **Transfer of Completion TLPs on the Completer Completion Interface with the Straddle Option Enabled (512-bit Interface)**



512-Bit Requester Interface

This section describes the operation of the user-side Requester interface associated with the 512-bit AXI4-Stream Interface. The block diagram in 512-Bit Completer Interface illustrates the introduction of a soft bridge within the core between the controller and the user application.

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

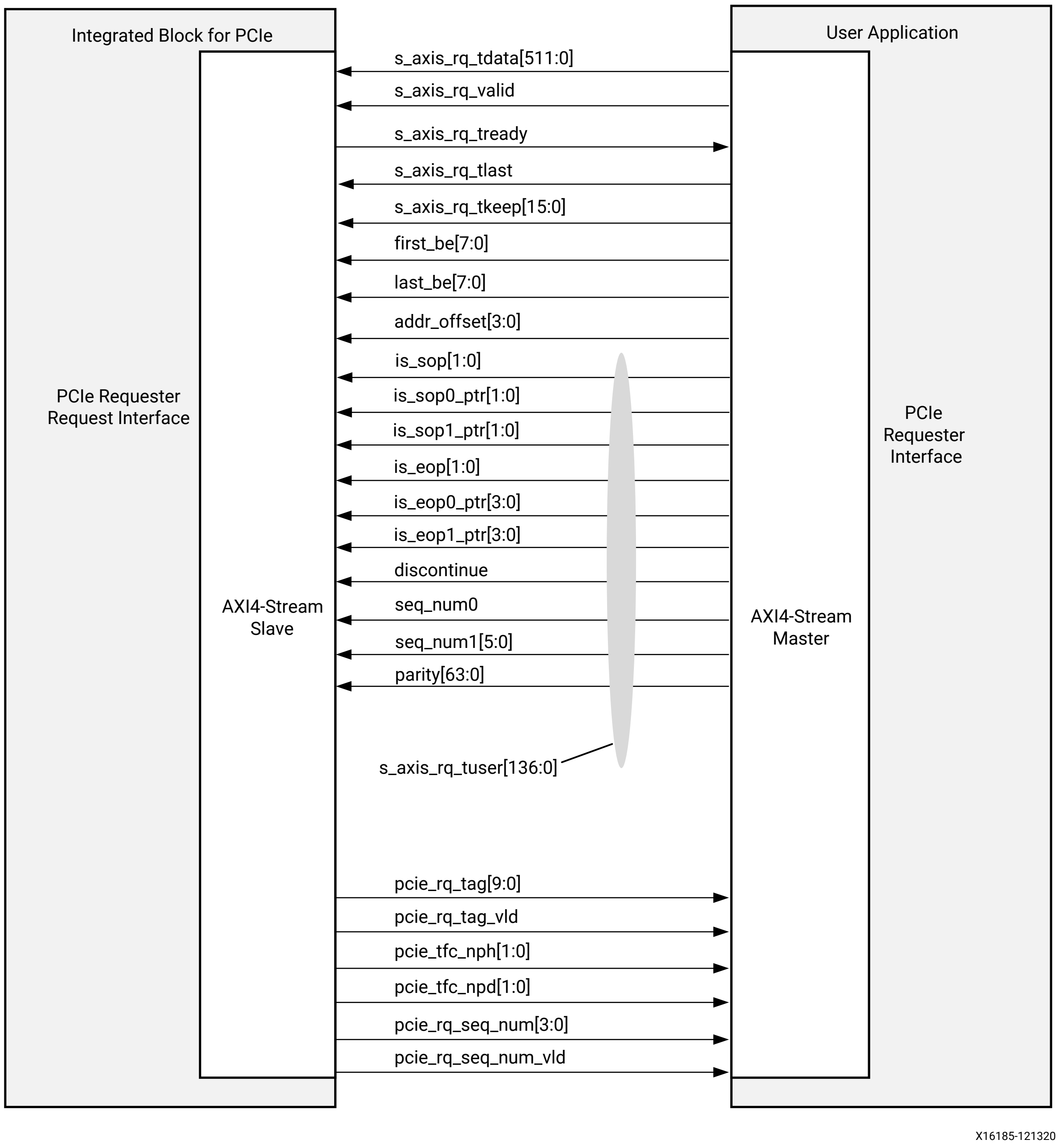
The requester interface consists of two separate interfaces, one for data transfer in each direction. Each interface is based on the AXI4-Stream protocol, and its width can be configured as 64, 128 or 256 bits. The requester request interface is used for transfer of requests (with any associated payload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

Requester Request Interface Operation (512-bits)

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

The requester request interface supports two distinct data alignment modes for transferring payloads, which are set during core customization in the Vivado® IDE. In the Dword-aligned mode, the user logic must provide the first Dword of the payload immediately after the last Dword of the descriptor. It must also set the bits in `first_be[7:0]` to indicate the valid bytes in the first Dword and the bits in `last_be[7:0]` (both part of the `s_axis_rq_tuser` bus) to indicate the valid bytes in the last Dword of the payload. In the address-aligned mode, the user logic must start the payload transfer in the beat following the last Dword of the descriptor, and its first Dword can be in any of the possible Dword positions on the data path. The user application communicates the offset of the first Dword on the data path using the signals `addr_offset[3:0]` in `s_axis_rq_tuser`. As in the case of the Dword-aligned mode, the user application must also set the bits in `first_be[7:0]` to indicate the valid bytes in the first Dword and the bits in `last_be[7:0]` to indicate the valid bytes in the last Dword of the payload. In Straddled case, `addr_offset[3:2]`, `first_be[7:4]`, and `last_be[7:4]` are used to indicate second TLP information while `addr_offset[1:0]`, `first_be[3:0]`, and `last_be[3:0]` are used to indicate the first TLP information on that data beat.

Figure 76: **Requester Request Interface Signals**



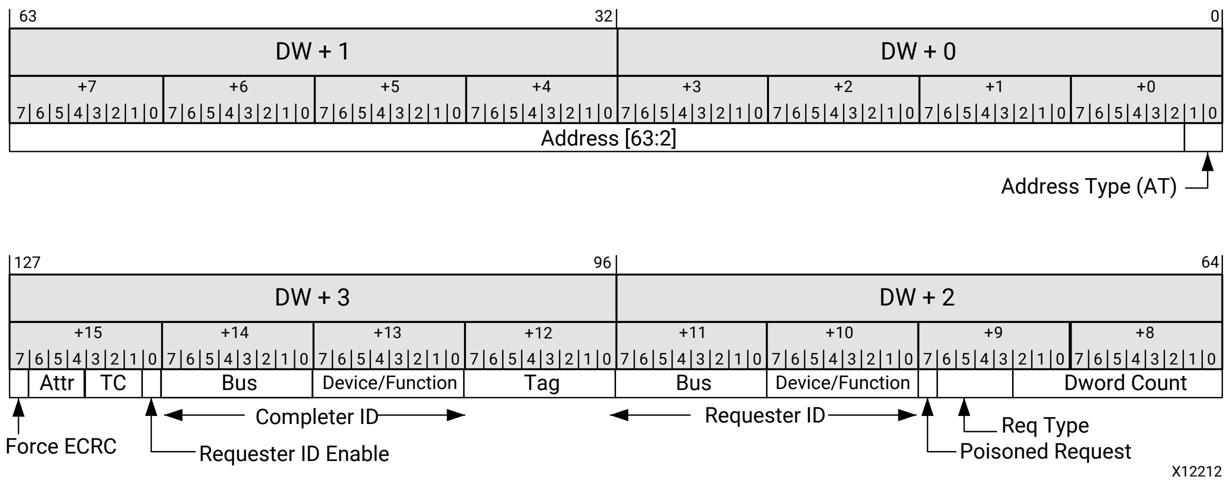
The interface also supports a straddle option that allows the transfer of up to two TLPs in the same beat across the interface. The straddle option can be used only with the Dword-aligned mode, and is not supported when using the 128-bit address aligned mode. The descriptions in the sections below assume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

Requester Request Descriptor Formats

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

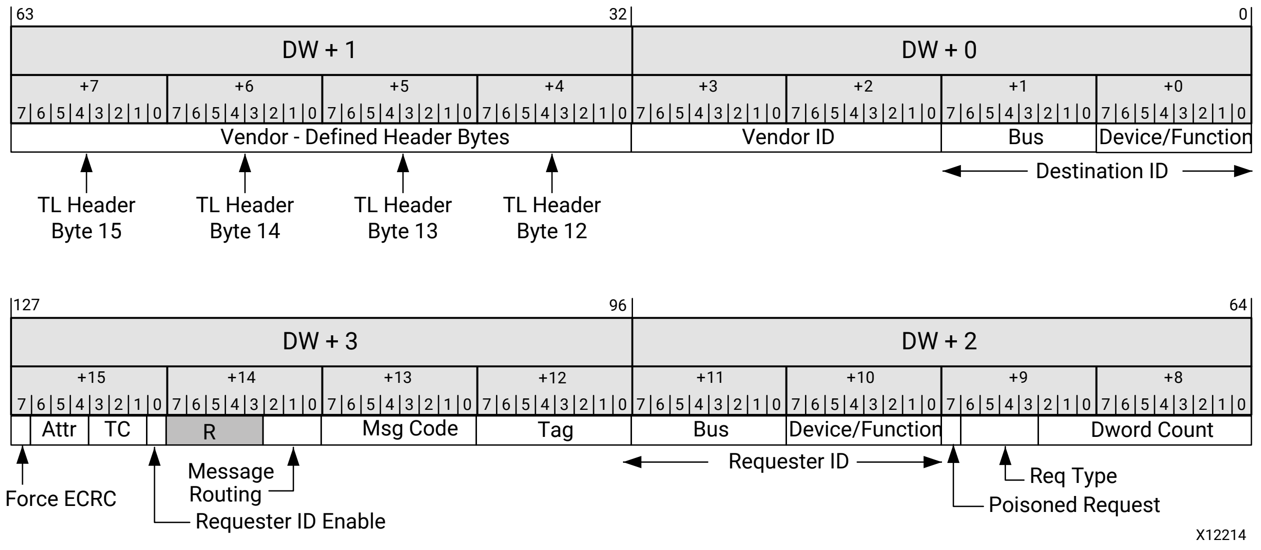
The format of the following figure applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request.

Figure 77: **Requester Request Descriptor Format for Memory, I/O, and Atomic Op Requests**



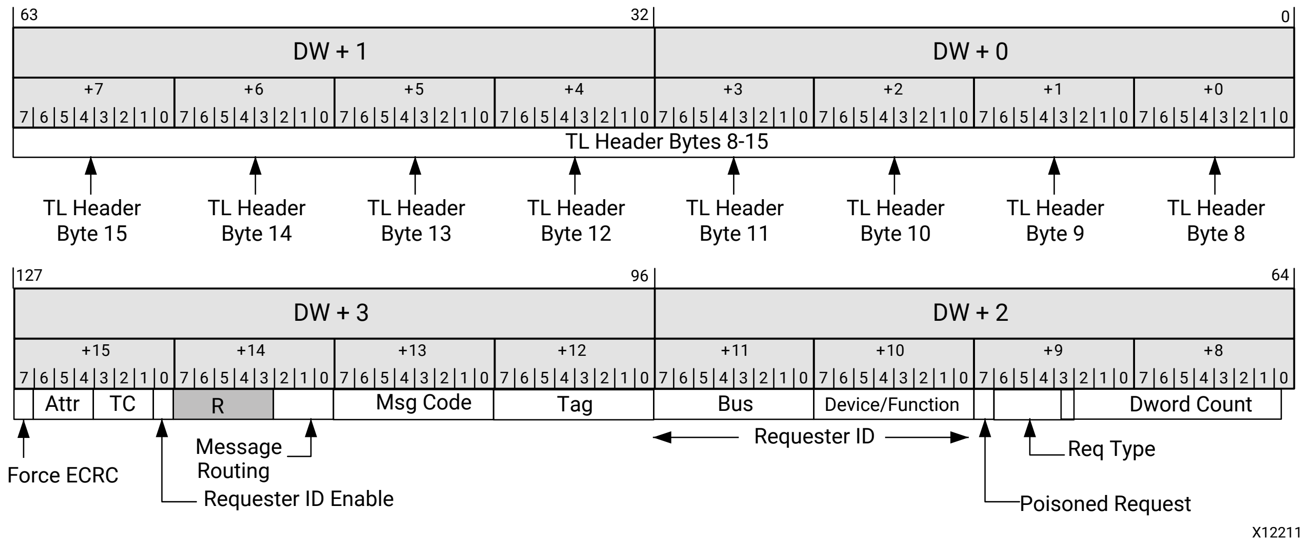
The format of the following figure is used for Vendor-Defined Messages (Type 0 or Type 1) only.

Figure 78: **Requester Request Descriptor Format for Vendor-Defined Messages**



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

Figure 79: **Requester Request Descriptor Format for ATS Messages**



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

Figure 80: **Requester Request Descriptor Format for all other Messages**

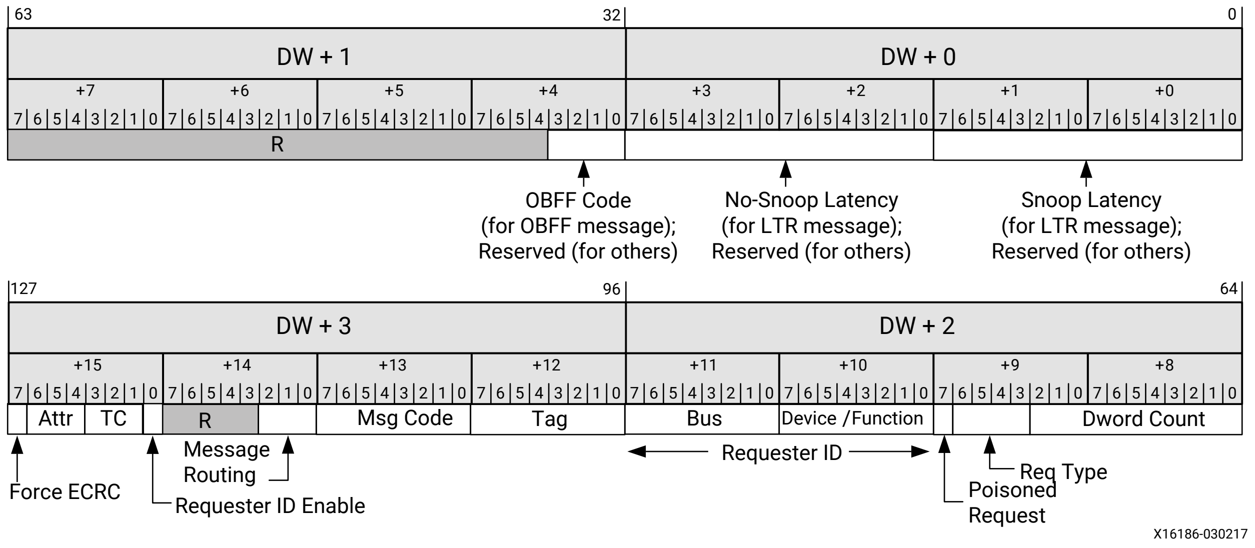


Table 60: **Requester Request Descriptor Fields**

Bit Index	Field Name	Description
1:0	Address Type	This field is defined for memory transactions and Atomic Operations only. The core copies this field into the AT of the TL header of the request TLP. • 00: Address in the request is un-translated • 01: Transaction is a Translation Request • 10: Address in the request is a translated address • 11: Reserved
63:2	Address	This field applies to memory, I/O and Atomic Op requests. This is the address of the first Dword referenced by the request. The user logic must also set the First_BE and Last_BE bits in `s_axis_rq_tuser` to indicate the valid bytes in the first and last Dwords, respectively. When the transaction specifies a 32-bit address, bits [63:32] of this field must be set to 0.
74:64	Dword Count	These 11 bits indicate the size of the block (in Dwords) to be read or written (for messages, size of the message payload). Its range is 0 - 256 Dwords. For I/O accesses, the Dword count is always 1. For a zero length memory read/write request, the Dword count must be 1, with the `First_BE` bits set to all zeroes. The core does not check the setting of this field against the actual length of the payload supplied (for requests with payload), nor against the maximum payload size or read request size settings of the core.
78:75	Request Type	Identifies the transaction type. The transaction types and their encodings are listed in the following table.
79	Poisoned Request	This bit can be used by the user logic to poison the request TLP being sent. This bit must be set to 0 for all requests, except when the user logic has detected an error in the block of data following the descriptor and wants to communicate this information using the Data Poisoning feature of PCI Express.

Table 60: **Requester Request Descriptor Fields** *(cont'd)*

Bit Index	Field Name	Description
87:80	Requester Function/ Device Number	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 in conjunction 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 in conjunction with Requester ID Enable bit in the descriptor.
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 in conjunction with Requester ID Enable bit in the descriptor. Root Port mode (Downstream Port): • Must be set to the Requester Bus number. This is used in conjunction with Requester ID Enable bit in the descriptor.
103:96	Tag	PCIe Tag associated with the request. For Posted transactions, the core always uses the value from this field as the tag for the request. For Non-Posted transactions, the core uses the value from this field if the Enable Client Tag is set during core configuration in the Vivado IDE (that is, when tag management is performed by the user logic). If this attribute is not set, tag management logic in the core is responsible for generating the tag to be used, and the value in the tag field of the descriptor is not used.

Table 60: **Requester Request Descriptor Fields** *(cont'd)*

Bit Index	Field Name	Description
119:104	Completer ID	This field is applicable only to Configuration requests and messages routed by ID. For these requests, this field specifies the PCI Completer ID associated with the request (these 16 bits are divided into an 8-bit bus number, 5-bit device number, and 3-bit function number in the legacy interpretation mode. In the ARI mode, these 16 bits are treated as an 8-bit bus number + 8-bit Function number.).
120	Requester ID Enable	`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. When Requester ID enable is 0 the device number fields in descriptor should also be 0. 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 in conjunction 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 in conjunction with Requester Bus Number bits [95:88] and Requester Function/Device Number bits [87:83] when ARI is not enabled.
123:121	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request.
126:124	Attributes	These bits provide the setting of the Attribute bits associated with the request. Bit 124 is the No Snoop bit, and bit 125 is the Relaxed Ordering bit. Bit 126 is the ID-Based Ordering bit, and can be set only for memory requests and messages. The core forces the attribute bits to 0 in the request sent on the link if the corresponding attribute is not enabled in the Function's PCI Express Device Control Register.
111:104	Message Code	This field is defined for all messages. It contains the 8-bit Message Code to be set in the TL header. Appendix F of the PCI Express 3.0 Specifications (available at http://www.pcisig.com/specifications) provides a complete list of the supported Message Codes.
114:112	Message Routing	This field is defined for all messages. The core copies these bits into the 3-bit Routing field r[2:0] of the TL header of the Request TLP.
15:0	Destination ID	This field applies to Vendor-Defined Messages only. When the message is routed by ID (that is, when the Message Routing field is `010` binary), this field must be set to the Destination ID of the message.
63:32	Vendor-Defined Header	This field applies to Vendor-Defined Messages only. It is copied into Dword 3 of the TL header.
63:0	ATS Header	This field is applicable to ATS messages only. It contains the bytes that the core copies into Dwords 2 and 3 of the TL header.

Requester Memory Write Operation

In both Dword-aligned and 128-bit address aligned modes, the transfer starts with the sixteen descriptor bytes, followed by the payload bytes. The user application must keep the signal `s_axis_rq_tvalid` asserted over the duration of the packet. The core treats the deassertion of `s_axis_rq_tvalid` during the packet transfer as an error, and nullifies the corresponding request TLP transmitted on the link to avoid data corruption.

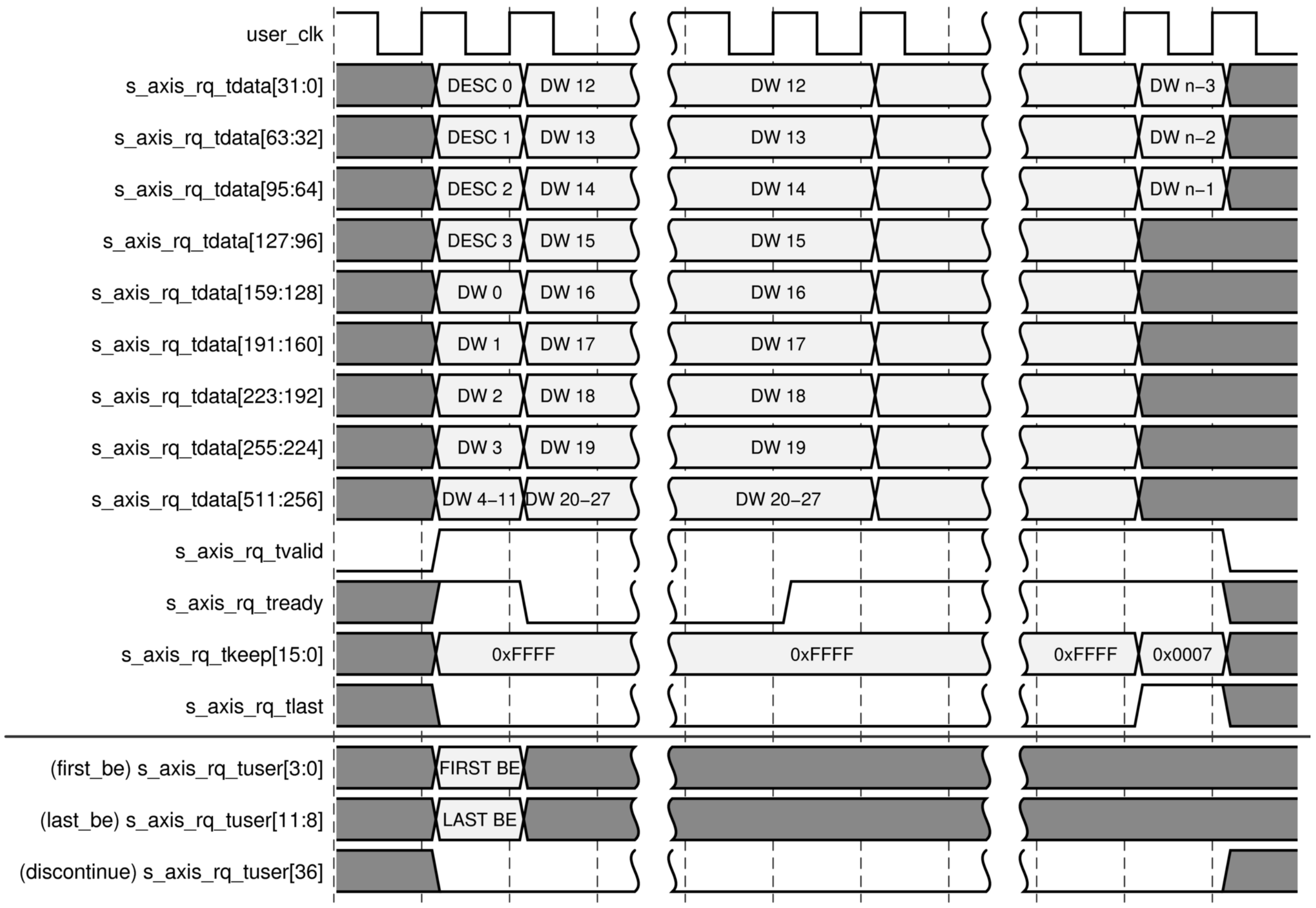
The user application must also assert the signal s_axis_rq_tlast in the last beat of the packet. The core are by pull down s_axis_rq_tready in any cycle if it is not ready to accept data. The user application must not change the values on `s_axis_rq_tdata` and `s_axis_rq_tlast` during the transfer when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

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

The user application must limit the size of the payload transferred in a single request to the maximum payload size configured in the core, and must ensure that the payload does not cross a 4 Kbyte boundary. For memory writes of two Dwords or less, the 1s in `first_be[7:0]` and `last_be[7:0]` are not be contiguous. For the special case of a zero-length memory write request, the user application must provide a dummy `one_dword` payload with `first_be[7:0]` and `last_be[7:0]` both set to all 0s. In all other cases, the 1 bits in `first_be[7:0]` and `last_be[7:0]` must be contiguous. In Straddled case, `addr_offset[3:2]`, `first_be[7:4]`, and `last_be[7:4]` are used to indicate second TLP information while `addr_offset[1:0]`, `first_be[3:0]`, and `last_be[3:0]` are used to indicate the first TLP information on that data beat.

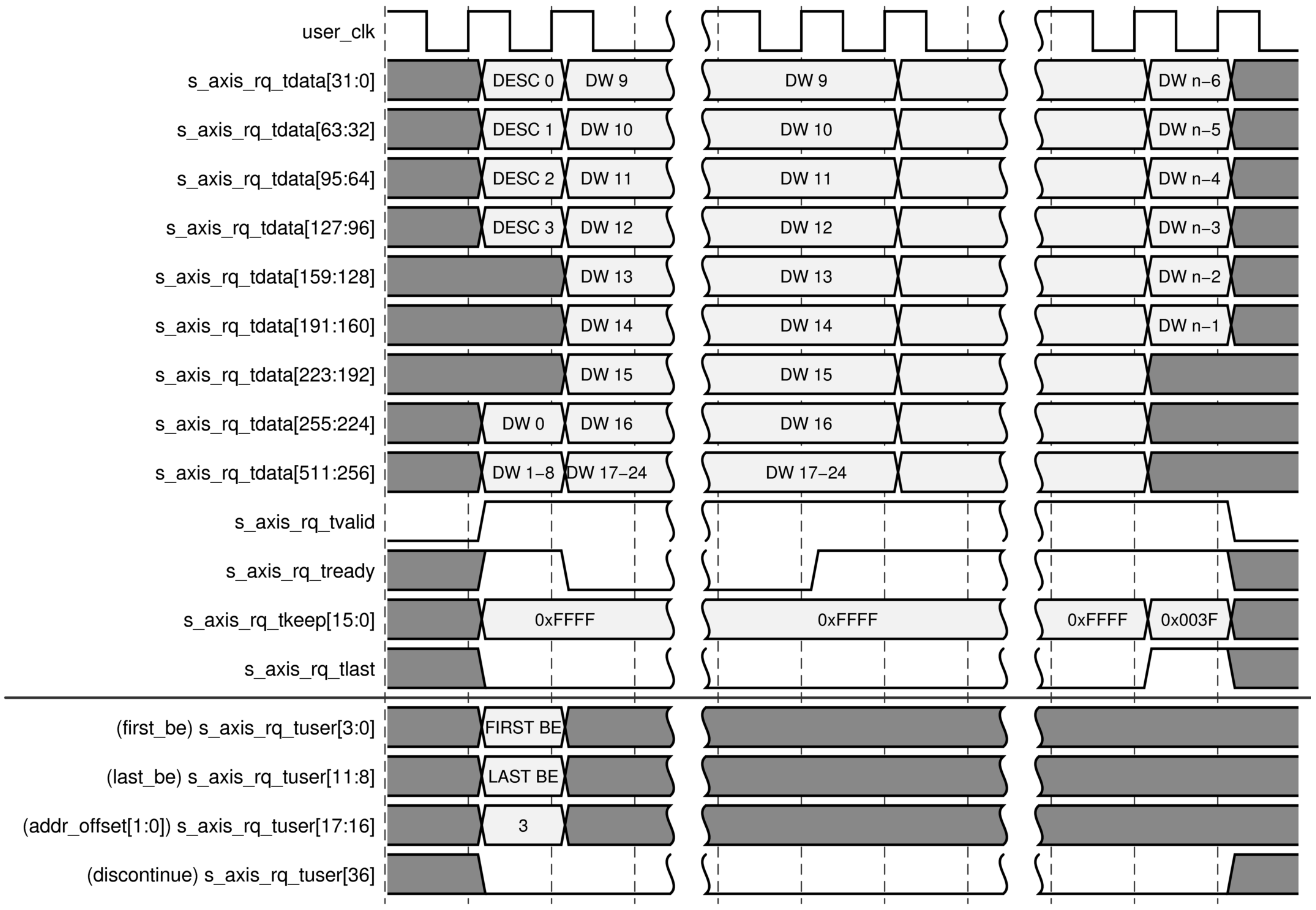
The following figure illustrates the Dword-aligned transfer of a memory write request from the user logic across the requester request interface. For the purpose of illustration, the size of the data block being written into user memory is assumed to be n Dwords, for some $n = k*16 - 1$, where $k > 1$.

Figure 81: **Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode)**



The following figure illustrates the 128-bit address aligned transfer of a memory write request from the user application across the requester request interface. For the purpose of illustration, the starting Dword offset of the data block is assumed to be (*m**16 +3), for some integer *m > 0*. Its size is assumed to be n Dwords, for some n = k*16 -1, k > 1. In the 128-bit address-aligned mode, the delivery of the payload always starts in the second 128-bit quarter of the 512-bit word, following the descriptor in the first quarter. The user application must communicate the offset of the first Dword of the payload in the `addr_offset[3:0]` field of the `s_axis_rq_tuser` bus. The user application must also set the bits in `first_be[7:0]` to indicate the valid bytes in the first Dword and the bits in `last_be[7:0]` to indicate the valid bytes in the last Dword of the payload.

Figure 82: **Memory Write Transaction on the Requester Request Interface (128-bit Address Aligned Mode)**

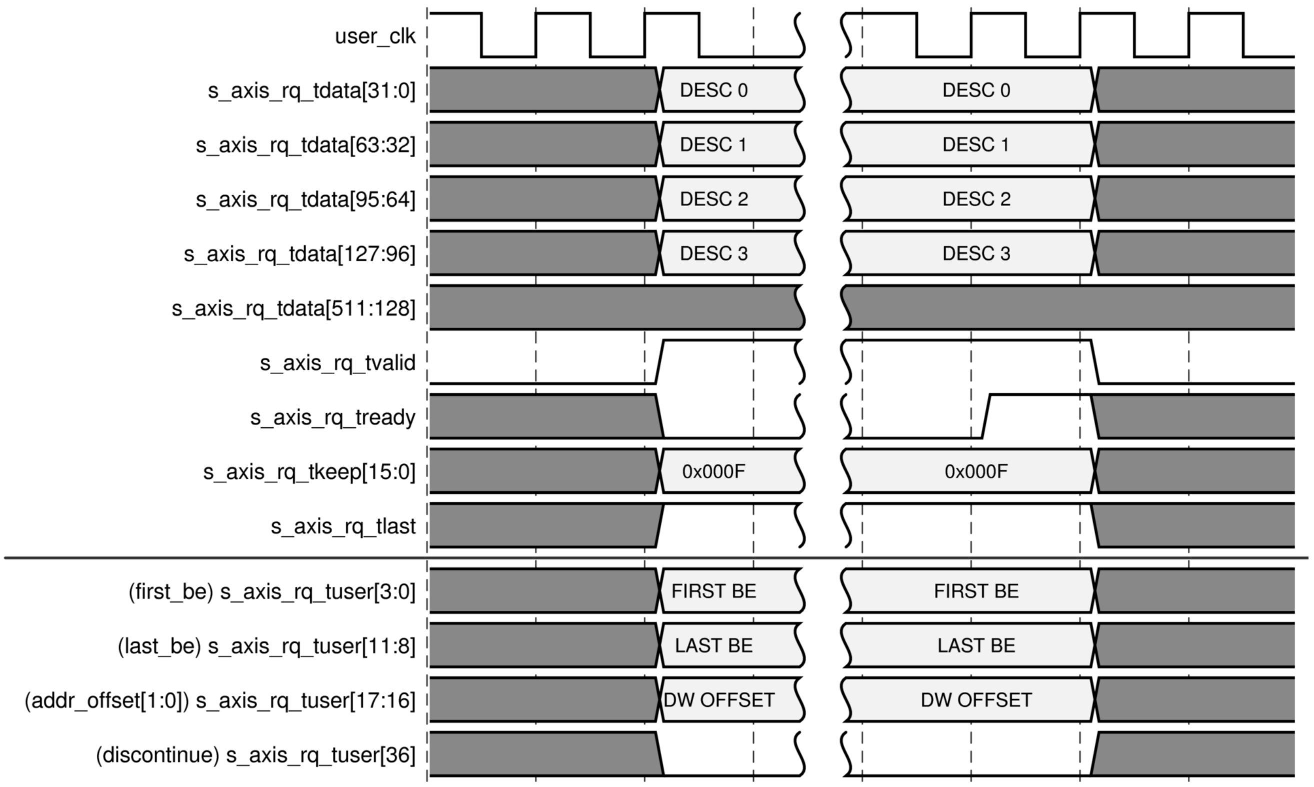


Non-Posted Transactions with No Payload

Non-Posted transactions with no payload (memory read requests, I/O read requests, Configuration read requests) are transferred across the requester request interface in the same manner as a memory write request, except that the AXI4-Stream packet contains only the 16-byte descriptor. The following figure illustrates the transfer of a memory read request across the requester request interface. The signal `s_axis_rq_tvalid` must remain asserted over the duration of the packet. The core are pull down `s_axis_rq_tready` to prolong the beat. The signal `s_axis_rq_tlast` must be set in the last beat of the packet, and the bits in `s_axis_rq_tkeep[15:0]` must be set in all Dword positions where a descriptor is present.

The user application must indicate the valid bytes in the first and last Dwords of the data block using the fields `first_be[7:0]` and `last_be[7:0]`, respectively, in the `s_axis_rq_tuser` bus. For the special case of a zero-length memory read, the length of the request must be set to one Dword, with both `first_be[7:0]` and `last_be[7:0]` set to all 0s. The user application must also communicate the offset of the first Dword of the payload of the resulting Completion, when delivered over the requester completion interface, in the `addr_offset[3:0]` field of the `s_axis_rq_tuser` bus. In Straddled case, `addr_offset[3:2]`, `first_be[7:4]`, and `last_be[7:4]` are used to indicate second TLP information while `addr_offset[1:0]`, `first_be[3:0]`, and `last_be[3:0]` are used to indicate the first TLP information on that data beat.

Figure 83: **Memory Read Transaction on the Requester Request Interface**



Non-Posted Transactions with a Payload

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

- In the Dword-aligned mode, the first Dword of the payload follows the last Dword of the descriptor, with no gaps between them.
- In the 128-bit address aligned mode, the payload must start in the second 128-bit quarter of the first beat, following the descriptor. The payload are start at any of four Dword positions in this quarter. The offset of its first Dword must be specified in the field `addr_offset[3:0]` of the `s_axis_rq_tuser bus`.

In the case of I/O and Configuration write requests, the valid bytes in the one-Dword payload must be indicated using `first_be[7:0]`. For Atomic Operation requests, all bytes in the first and last Dwords are assumed valid.

Message Requests on the Requester Interface

The transfer of a message on the requester request interface is similar to that of a memory write request, except that a payload are not always be present. The transfer starts with the 128-bit descriptor, followed by the payload, if present. The first Dword of the payload must immediately follow the descriptor, regardless of the address alignment mode in use. The `addr_offset[3:0]` field in the `s_axis_rq_tuser` bus must be set to 0 for messages when the address-aligned mode is in use. The core determines the end of the payload from `s_axis_rq_tlast` and `s_axis_rq_tkeep` signals. The First Byte Enable and Last Byte Enable bits (`first_be[7:0]` and `last_be[7:0]`) are not used for message requests.

Aborting a Transfer

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

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

The `discontinue` signal can be asserted only when `s_axis_rq_tvalid` is High. The core samples this signal when `s_axis_rq_tvalid` and `s_axis_rq_tready` are both High. Thus, once asserted, it should not be deasserted until `s_axis_rq_tready` is High. The user application must not start a new packet in the same beat when a previous packet is aborted by asserting the `discontinue` input.

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

Straddle Option on RQ Interface

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

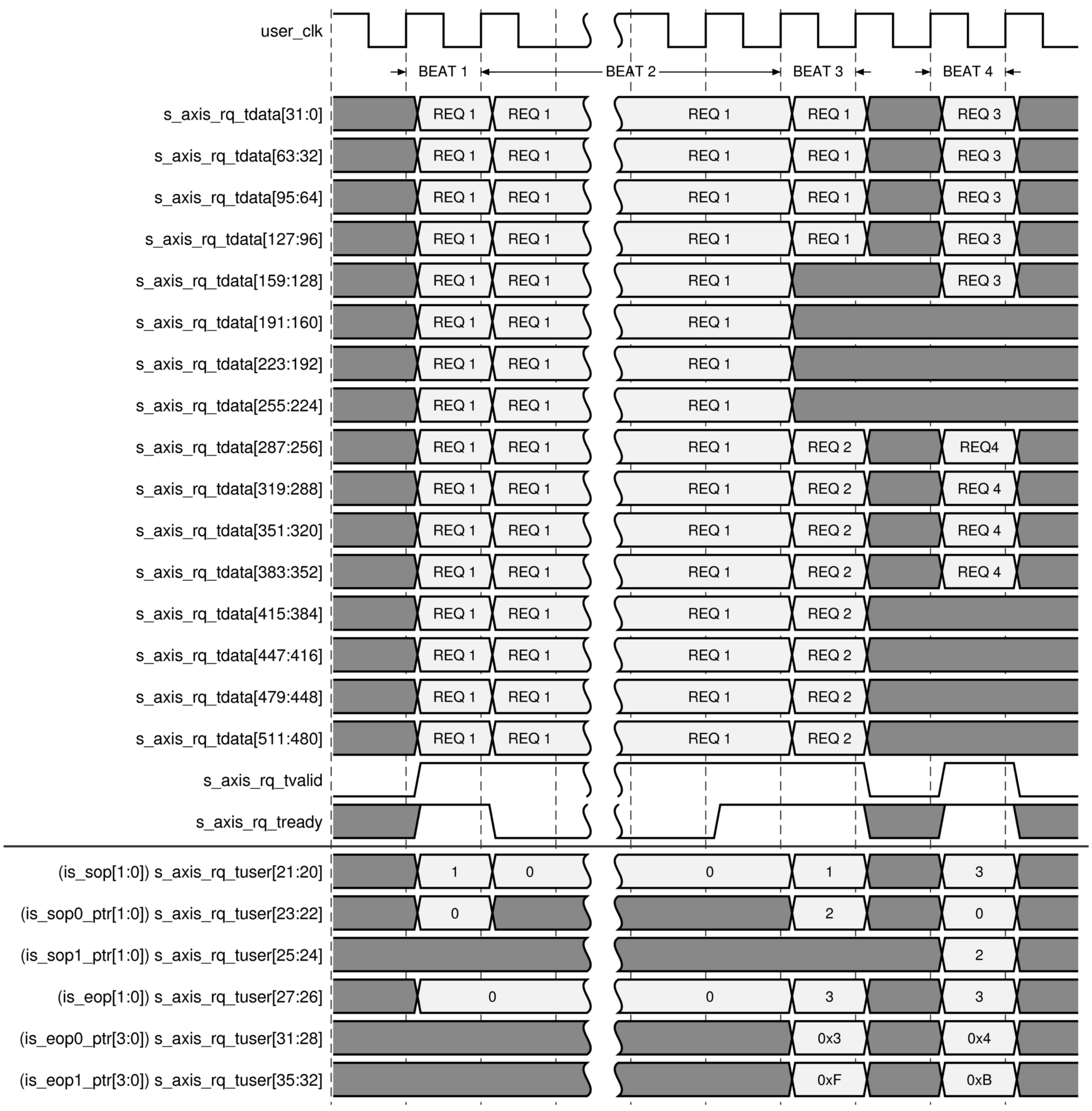
When the straddle option is enabled, request TLPs are transferred on the AXI4-Stream interface as a continuous stream, with no packet boundaries. Thus, the signals `m_axis_rq_tkeep` and `m_axis_rq_tlast` are not useful in determining the boundaries of TLPs delivered on the interface. Instead, delineation of TLPs is performed using the following signals provided within the `m_axis_rq_tuser` bus.

- `is_sop[0]`: This input must be set High in a beat when there is at least one request TLP starting in the beat. The position of the first byte of the descriptor of this TLP is determined as follows:
  - If the previous TLP ended before this beat, the first byte of the descriptor is in byte lane 0.
  - If a previous TLP is continuing in this beat, the first byte of this descriptor is in byte lane 32. This is possible only when the previous TLP ends in the current beat, that is when `is_eop[0]` is also set.
- `is_sop0_ptr[1:0]`: When `is_sop[0]` is set, this field must indicate the offset of the first request TLP starting in the current beat. Valid settings are `2'b00` (TLP starting at Dword 0) and `2'b10` (TLP starting at Dword 8).
- `is_sop[1]`: This input must be set High in a beat when there are two request TLPs starting in the same beat. The first TLP must always start at byte position 0 and the second TLP at byte position 32. The user application are start a second TLP at byte position 32 only if the previous TLP ended before byte position 32 in the same beat, that is only if `is_eop[0]` is also set in the same beat.
- `is_sop1_ptr[1:0]`: When `is_sop[1]` is set, this field must provide the offset of the second TLP starting in the current beat. Its only valid setting is `2'b10` (TLP starting at Dword 8).
- `is_eop[0]`: This input is used to indicate the end of a request TLP. Its assertion signals that there is at least one TLP ending in this beat.
- `is_eop0_ptr[3:0]`: When `is_eop[0]` is asserted, `is_eop0_ptr[3:0]` must provide the offset of the last Dword of the corresponding TLP ending in this beat.
- `is_eop[1]`: This input is set High when there are two TLPs ending in the current beat. `is_eop[1]` can be set only when the signals `is_eop[0]` and `is_sop[0]` are also be High in the same beat.
- `is_eop1_ptr[3:0]`: When `is_eop[1]` is asserted, `is_eop1_ptr[3:0]` must provide the offset of the last Dword of the second TLP ending in this beat. Because the second TLP can start only on byte lane 32, it can only end at a byte lane in the range 43-63. Thus the offset `is_eop1_ptr[3:0]` can only take a value in the range 10-15.

When a second TLP starts in the same beat, the First Byte Enable and Last Byte Enable bits of the second TLP are specified by the bit fields `first_be[7:4]` and `last_be[7:4]`, respectively, in the tuser bus.

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

Figure 84: **Transfer of Request TLPs on the Requester Request Interface with the Straddle Option Enabled**



Tag Management for Non-Posted Transactions

The requester side of the core maintains the state of all pending Non-Posted transactions (memory reads, I/O reads and writes, configuration reads and writes, Atomic Operations) initiated by the user application, so that the completions returned by the targets can be matched against the corresponding requests. The state of each outstanding transaction is held in a Split Completion Table in the requester side of the interface, which has a capacity of up to 768 Non-Posted transactions. The returning Completions are matched with the pending requests using an 5-/8-/10-bit tag. There are two options for management of these tags:

- **Internal Tag Management:** This mode of operation is selected during core customization in the Vivado® IDE. In this mode, logic within the core is responsible for allocating the tag for each Non-Posted request initiated from the requester side. The core maintains a list of free tags and assigns one of them to each request when the user logic initiates a Non-Posted transaction, and communicates the assigned tag value to the user logic through the output `pcie_rq_tag0[9:0]` and `pcie_rq_tag1[9:0]`. The value on this bus is valid when the core asserts `pcie_rq_tag_vld0` and `pcie_rq_tag_vld1`. Use of `pcie_rq_tag_vld0` and `pcie_rq_tag_vld1` are orthogonal to whether the Straddle option is enabled. The integrated block can use either the `pcie_rq_tag_vld0` or `pcie_rq_tag_vld1` port to showcase the valid tags. The user logic must copy the tag so that any Completions delivered by the core in response to the request can be matched to the request.

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

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

When internal tag management is in use, the core asserts `pcie_rq_tag_vld` for one cycle for each Non-Posted request, after it has placed its allocated tag on `pcie_rq_tag`. When straddle option is enabled, the core are provide up to two allocated tags in the same cycle on this interface. The states of the signals `pcie_rq_tag_vld` and `pcie_rq_tag` must be interpreted as follows:

- Assertion of `pcie_rq_tag_vld0` in any cycle indicates that the core has placed an allocated tag on `pcie_rq_tag0[9:0]`.

- Simultaneous assertion of `pcie_rq_tag_vld0` and `pcie_rq_tag_vld1` in the same cycle indicates that the core has placed two allocated tags, the first on `pcie_rq_tag0[9:0]` and the second on `pcie_rq_tag1[9:0]`. The tag on `pcie_rq_tag0[9:0]` corresponds to an earlier request sent by the user logic and the tag on `pcie_rq_tag1[9:0]` corresponds to a later request.
- `pcie_rq_tag_vld1` is never asserted when `pcie_rq_tag_vld0` is not asserted. That is, when there is only one tag to communicate in any cycle, it is always communicated on `pcie_rq_tag0[9:0]`.
- When straddle is not in use, only a single tag can be communicated in any cycle, and `pcie_rq_tag_vld1` is never asserted.

There can be a delay of several cycles between the transfer of the request on the `s_axis_rq_tdata` bus and the assertion of `pcie_rq_tag_vld` by the core to provide the allocated tag for the request. The user logic are, meanwhile, continue to send new requests. The tags for requests are communicated on the `pcie_rq_tag` bus in FIFO order, so it is easy for the user logic to associate the tag value with the request it transferred.

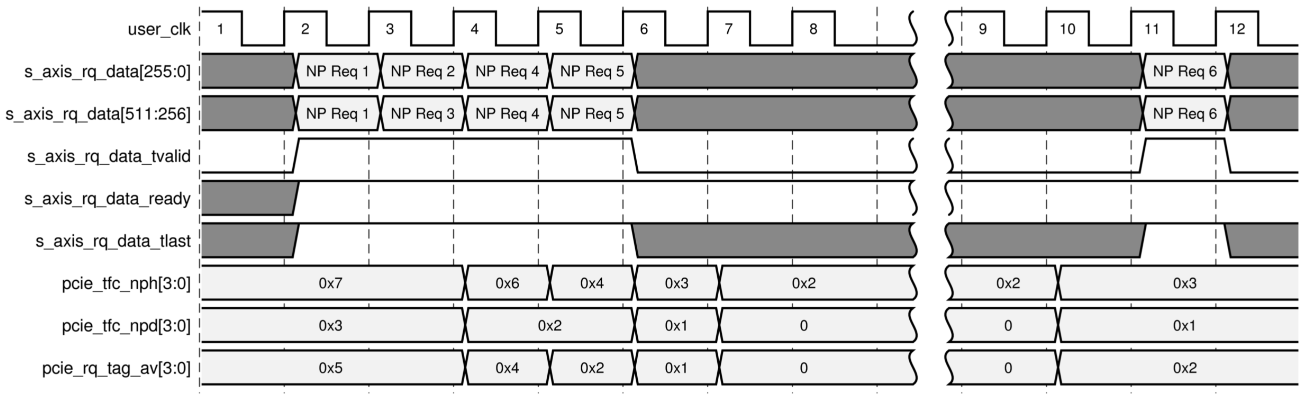
Avoiding Head-of-Line Blocking for Posted Requests

The core holds a Non-Posted request received on its requester request interface for lack of transmit credit or lack of available tags. This could potentially result in HOL blocking for Posted transactions. Such a condition can be prevented if the user logic has the ability to check the availability of transmit credit and tags for Non-Posted transactions. The core provides the following signals for this purpose:

- `pcie_tfc_nph_av[3:0]`: These outputs indicate the Header Credit currently available for Non-Posted requests (0000 = no credit available, 0001 = 1 credit available, 0010 = 2 credits, ..., 1111 = 15 or more credits available).
- `pcie_tfc_npd_av[3:0]`: These outputs indicate the Data Credit currently available for Non-Posted requests (0000= no credit available, 0001 = 1 credit available, 0010 = 2 credits, ..., 1111 = 15 or more credits available).
- `pcie_rq_tag_av[3:0]`: These outputs indicate the number of free tags currently available for allocation to Non-Posted requests (0000 = no tags available, 0001 = 1 tag available, 0010 = 2 tags available, ..., 1111 = 15 or more tags available).

The user logic are optionally check these outputs before transmitting Non-Posted requests. Because of internal pipeline delays, the information on these outputs is delayed by two user clock cycles from the cycle in which the last byte of the descriptor is transferred on the requester request interface, so the user logic must adjust these values taking into account any Non-Posted requests transmitted in the two previous clock cycles. The following figure illustrates the operation of these signals. In this example, the core initially had 7 Non-Posted Header Credits and 3 Non-Posted Data Credits, and had 5 free tags available for allocation. Request 1 from the user logic had a one-Dword payload, and therefore consumed 1 header and data credit each, and also one tag. Requests 2 and 3 (straddled) in the next clock cycle 3 consumed 1 header credit each, but no data credit. When the user logic presents Request 4 in clock cycle 4, it must adjust the available credit and available tag count by taking into account Requests 1, 2 and 3, presented in the two previous cycles. Request 4 consumes 1 header credit and one data credit. When the user logic presents Request 5 in clock cycle 5, it must adjust the available credit and available tag count by taking into account Requests 2, 3 and 4. If Request 5 consumes one header credit and one data credit the available data credit is two cycles later, as also the number of available tags. Thus, Request 6 must wait for the availability of new credit.

Figure 85: **Operation of credit and tag availability signals on the Requester Request Interface**



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

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

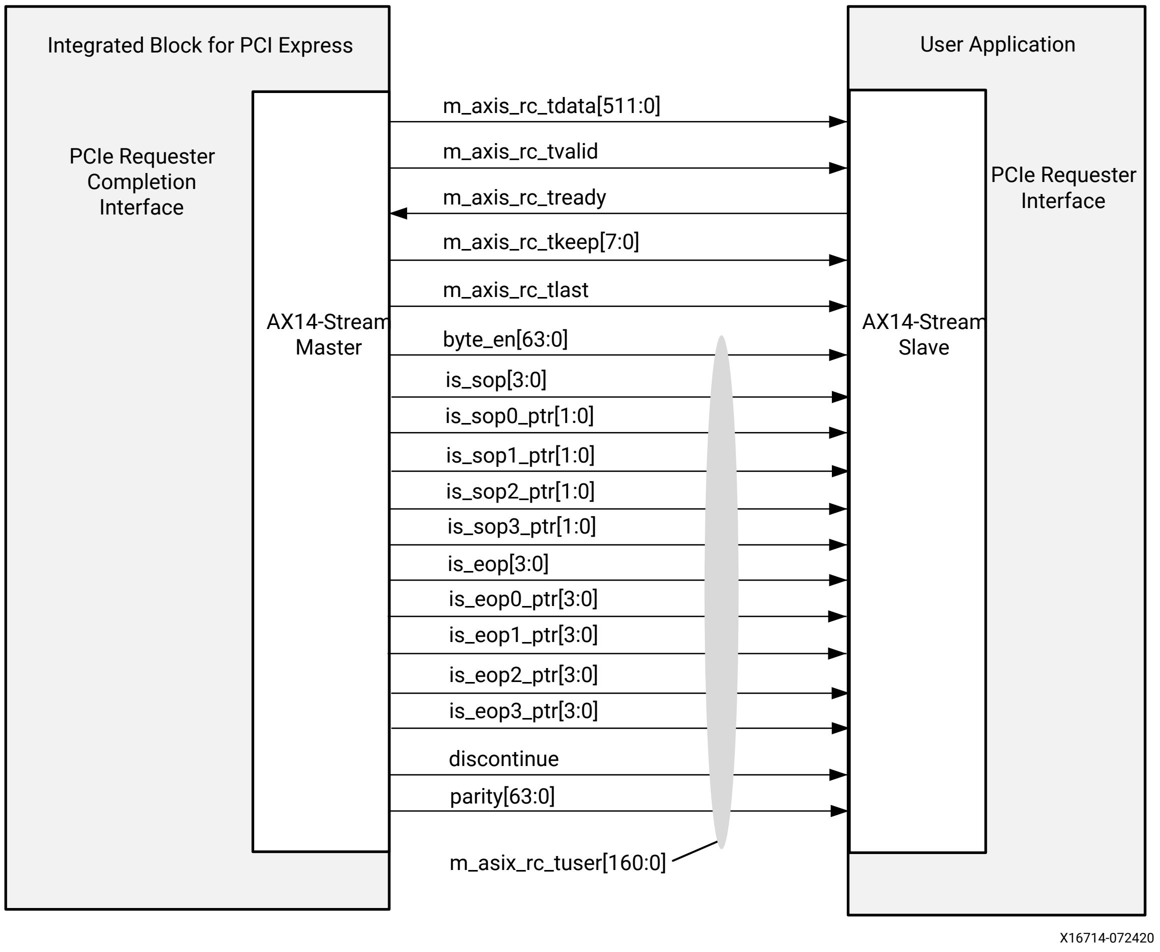
Maintaining Transaction Order

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

When transferring a Posted request (memory write transactions or messages) across the requester request interface, the user logic are provide an optional 6-bit sequence number to the PCIe® core in its first beat. The sequence number field `seq_num0[5:0]` within `s_axis_rq_tuser` is used to send the sequence number for the first TLP starting in the beat, and the `field seq_num1[5:0]` is used to send the sequence number for the second TLP starting in the beat (if present). The user logic can then monitor the `pcie_rq_seq_num0[5:0]` and `pcie_rq_seq_num1[5:0]` outputs of the core for these sequence numbers to appear. When the transaction has reached a stage in the internal transmit pipeline of the core where a Completion is unable to pass it, the core asserts `pcie_rq_seq_num_vld0` for one cycle and provides the sequence number of the Posted request on the `pcie_rq_seq_num0[5:0]` output. If there is a second Posted request in the pipeline in the same cycle, the core also asserts `pcie_rq_seq_num_vld1` in the same cycle and provides the sequence number of the second Posted request on the `pcie_rq_seq_num1[5:0]` output. The user logic must therefore monitor both sets of the sequence number outputs to check if a specific TLP has reached the pipeline stage. Any Completions transmitted by the core after the sequence number has appeared on `pcie_rq_seq_num0[5:0]` or `pcie_rq_seq_num1[5:0]` is guaranteed not to pass the corresponding Posted request in the internal transmit pipeline of the core.

Requester Completion Interface Operation (512-bits)

Figure 86: **Requester Completion Interface Signals**



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

The requester completion interface supports two distinct data alignment modes for transferring payloads, which are during core customization in the Vivado® IDE. In the Dword-aligned mode, the core transfers the first Dword of the Completion payload immediately after the last Dword of the descriptor. In the 128-bit address aligned mode, the core starts the payload transfer in the second 128-bit quarter of the 512-bit word, following the descriptor in the first quarter. The first Dword of the payload can be in any of the four possible Dword positions in the second quarter, and its offset f the is determined by address offset provided by the user logic when it sent the request to the core (that is, the setting of the `addr_offset` input of the requester request interface). Thus, the 128-bit address aligned mode can be used on the requester completion interface only if the requester request interface is also configured to use the 128-bit address aligned mode.

Requester Completion Descriptor Format

The requester completion interface of the core sends completion data received from the link to the user application as AXI4-Stream packets. Each packet starts with a descriptor, and can have payload data following the descriptor. The descriptor is always 12 bytes long, and is sent in the first 12 bytes of the completion packet. When the completion data is split into multiple Split Completions, the core sends each Split Completion as a separate AXI4-Stream packet, with its own descriptor.

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

Figure 87: **Requester Completion Descriptor Format**

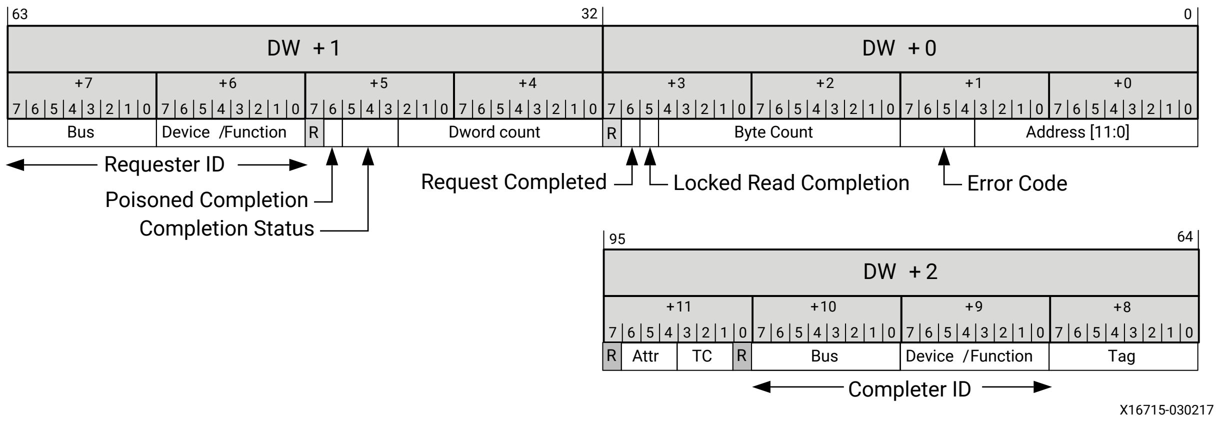


Table 61: **Requester Completion Descriptor Fields**

Bit Index	Field Name	Description
11:0	Lower Address	This field provides the 12 least significant bits of the first byte referenced by the request. The core returns this address from its Split Completion Table, where it stores the address and other parameters of all pending Non-Posted requests on the requester side. When the Completion delivered has an error, only bits [6:0] of the address should be considered valid. This is a byte-level address. For ATS translation requests, this field is reserved and implied to be zero.
15:12	Error Code	Completion error code. These three bits encode error conditions detected from error checking performed by the core on received Completions. Its encodings are: • 0000: Normal termination (all data received). • 0001: The Completion TLP is Poisoned. • 0010: Request terminated by a Completion with UR, CA or CRS status. • 0011: Request terminated by a Completion with no data. • 0100: The current Completion being delivered has the same tag of an outstanding request, but its Requester ID, TC, or Attr field does not match with the parameters of the outstanding request. • 0101: Error in starting address. The low address bits in the Completion TLP header did not match with the starting address of the next expected byte for the request. • 0110: Invalid tag. This Completion does not match the tags of any outstanding request. • 0111: Invalid byte count. The byte count in the Completion was higher than the total number of bytes expected for the request. • 1000: Request terminated by a Completion timeout. The other fields in the descriptor, except bit [30], the requester Function [55:48], and the tag field [71:64], are invalid in this case, because the descriptor does not correspond to a Completion TLP. • 1001: Request terminated by a Function-Level Reset (FLR) targeted at the Function that generated the request. The other fields in the descriptor, except bit [30], the requester Function [55:48], and the tag field [71:64], are invalid in this case, because the descriptor does not correspond to a Completion TLP.
28:16	Byte Count	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.

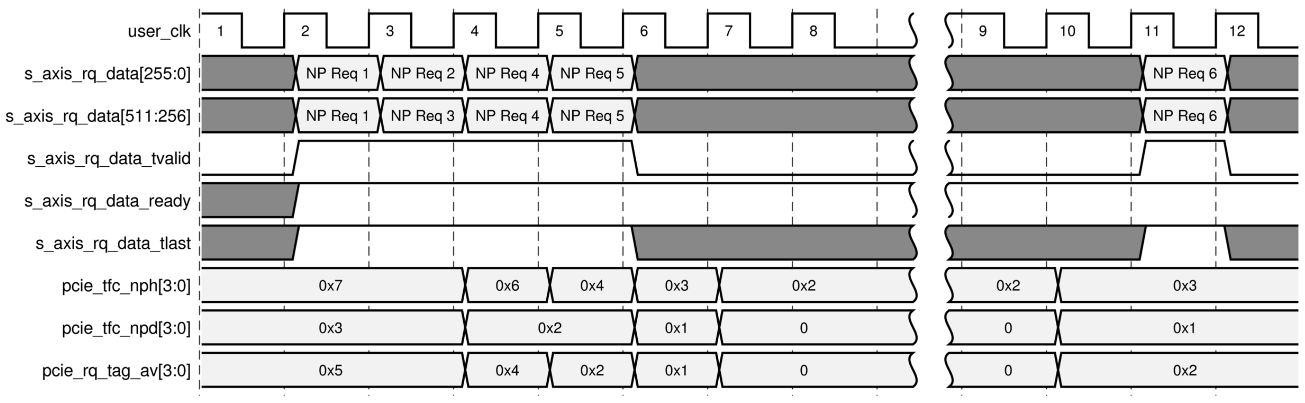
Table 61: **Requester Completion Descriptor Fields** *(cont'd)*

Bit Index	Field Name	Description
30	Request Completed	The core asserts this bit in the descriptor of the last Completion of a request. The assertion of the bit indicates normal termination of the request (because all data has been received), or abnormal termination because of an error condition. The user logic can use this indication to clear its outstanding request. When tags are assigned by the user logic, the user logic should not reassign a tag allocated to a request until it has received a Completion Descriptor from the core with a matching tag field and the Request Completed bit set to 1.
42:32	Dword Count	These 11 bits indicate the size of the payload of the current packet in Dwords. Its range is 0 – 1K Dwords. This field is set to 1 for I/O read Completions and 0 for I/O write Completions. The Dword count is also set to 1 while transferring a Completion for a zero-length memory read. In all other cases, the Dword count corresponds to the actual number of Dwords in the payload of the current packet.
45:43	Completion Status	These bits reflect the setting of the Completion Status field of the received Completion TLP. The valid settings are: • 000: Successful Completion. • 001: Unsupported Request (UR). • 010: Configuration Request Retry Status (CRS). • 100: Completer Abort (CA).
46	Poisoned Completion	This bit is set to indicate that the Poison bit in the Completion TLP was set. Data in the packet should then be considered corrupted.
63:48	Requester ID	PCI Requester ID associated with the Completion.
71:64	Tag	PCIe Tag associated with the Completion.
87:72	Completer ID	Completer ID received in the Completion TLP. (These 16 bits are divided into an 8-bit bus number, 5-bit device number, and 3-bit function number in the legacy interpretation mode. In ARI mode, these 16 bits must be treated as an 8-bit bus number + 8-bit Function number.)
91:89	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the Completion.
94:92	Attributes	PCIe attributes associated with the Completion. Bit 92 is the No Snoop bit, bit 93 is the Relaxed Ordering bit, and bit 94 is reserved.

Transfer of Completions with No Data

The following figure illustrates the transfer of a Completion TLP received from the link with no associated payload across the requester completion interface. The timing diagrams in this section assume that the Completions are not straddled on the interface. The straddle feature is described in Straddle Option for RC Interface.

Figure 88: **Transfer of a Completion with no Data on the Requester Completion Interface**



The entire transfer of the Completion TLP takes only a single beat on the interface. The core keeps the signal `m_axis_rc_tvalid` asserted over the duration of the packet. The user logic can prolong a beat at any time by pulling down `m_axis_rc_tready`. The AXI4-Stream interface signals `m_axis_rc_tkeep` (one per Dword position) indicate the valid descriptor Dwords in the packet. That is, the `m_axis_rc_tkeep` bits are set to 1 contiguously from the first Dword of the descriptor until its last Dword. The signal `m_axis_rc_tlast` is always asserted, indicating that the packet ends in its current beat.

The `m_axi_rc_tuser` bus also includes a signal `is_sop[0]`, which is asserted in the first beat of every packet. The user logic are optionally use this signal to qualify the start of the descriptor on the interface. When the straddle option is not in use, none of the other sop and eop indications within `m_axi_rc_tuser` are relevant to the transfer of Completions.

Transfer of Completions with Data

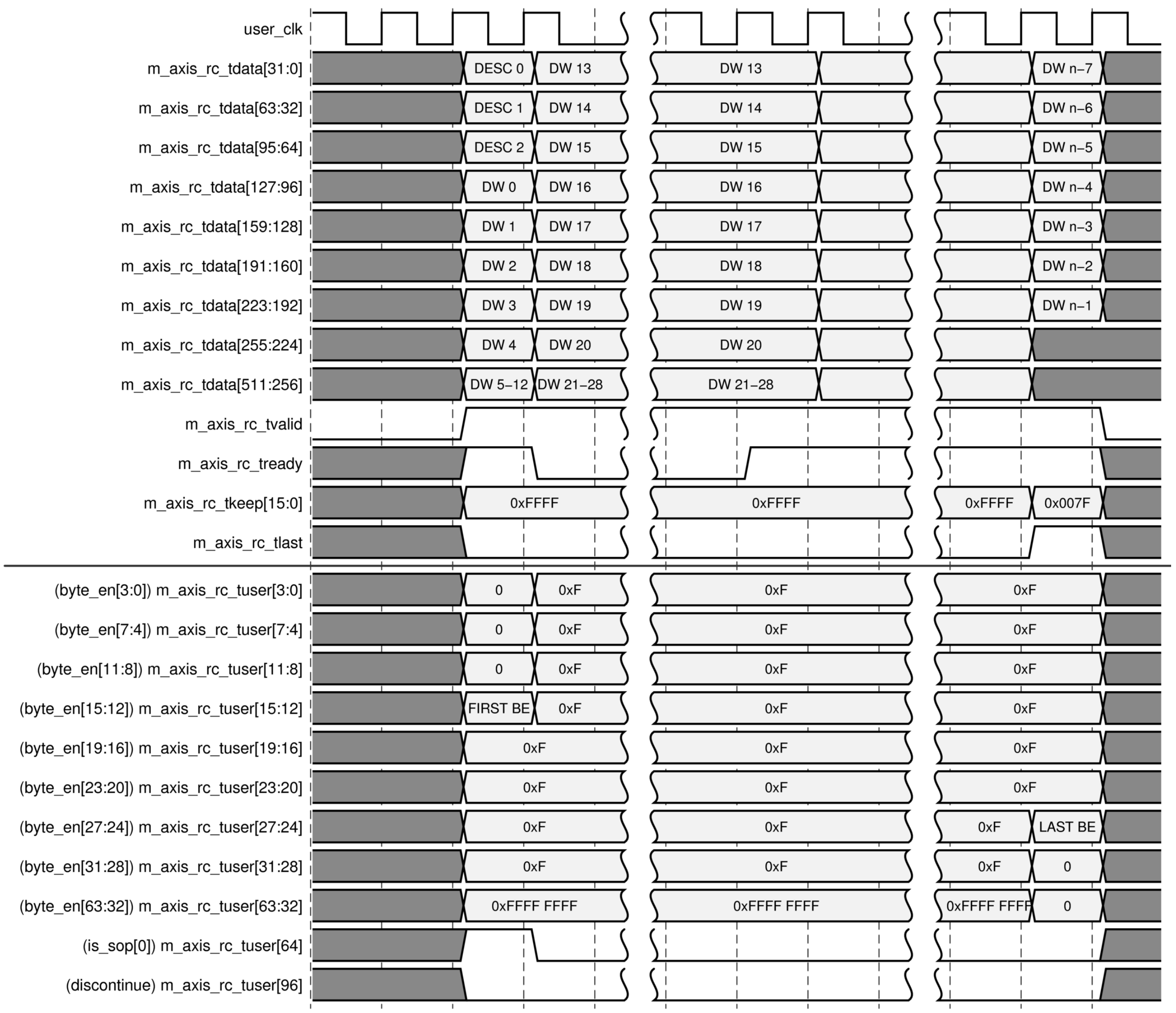
In the Dword-aligned mode, the transfer starts with the three descriptor Dwords, followed immediately by the payload Dwords. The entire TLP, consisting of the descriptor and payload, is transferred as a single AXI4-Stream packet. Data within the payload is always a contiguous stream of bytes when the length of the payload exceeds two Dwords. The positions of the first valid byte within the first Dword of the payload and the last valid byte in the last Dword can then be determined from the Lower Address and Byte Count fields of the Request Completion Descriptor. When the payload size is 2 Dwords or less, the valid bytes in the payload are not be contiguous. In these cases, the user logic must store the First Byte Enable and the Last Byte Enable fields associated with each request sent out on the requester request interface and use them to determine the valid bytes in the completion payload. The user logic are optionally use the byte enable outputs `byte_en[63:0]` within the `m_axi_rc_tuser` bus to determine the valid bytes in the payload, in the cases of both contiguous and non-contiguous payloads.

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

The `m_axi_rc_tuser` bus provides several optional signals that can be used to simplify the logic associated with the user side of the interface, or to support additional features. The signal `is_sop[0]` is asserted in the first beat of every packet, when its descriptor is on the bus. When the straddle option is not in use, none of the other sop and eop indications within `m_axi_rc_tuser` are relevant to the transfer of Completions. The byte enable outputs `byte_en[63:0]` (one per byte lane) indicate the valid bytes in the payload. These signals are asserted only when a valid payload byte is in the corresponding lane (it is not asserted for descriptor or null bytes). The asserted byte enable bits are always contiguous from the start of the payload, except when payload size is 2 Dwords or less. For Completion payloads of two Dwords or less, the 1s on `byte_en` are not be contiguous. Another special case is that of a zero-length memory read, when the core transfers a one-Dword payload with the `byte_en` bits all set to 0. Thus, the user logic can, in all cases, use the `byte_en` signals directly to enable the writing of the associated bytes into memory.

The following figure illustrates the Dword-aligned transfer of a Completion TLP received from the link with an associated payload across the requester completion interface. For the purpose of illustration, the size of the data block being written into user memory is assumed to be n Dwords, where $n = k*16 + 4$, for some *k > 1*. The timing diagrams in this section assume that the Completions are not straddled on the interface. The straddle feature is described in Straddle Option for RC Interface.

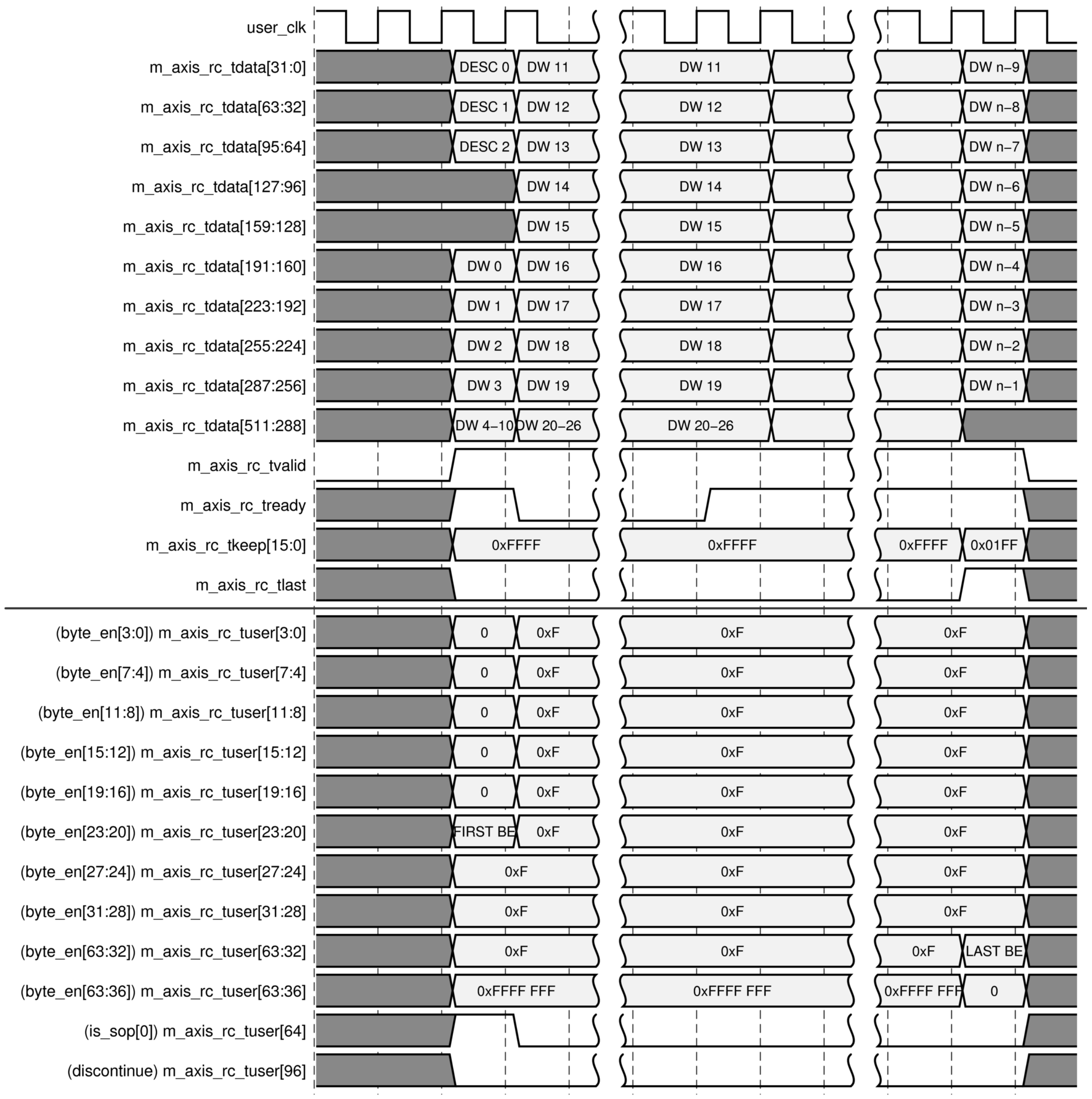
Figure 89: **Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode)**



The following figure illustrates the address-aligned transfer of a Completion TLP received from the link with an associated payload across the requester completion interface. In the example timing diagrams, the starting Dword address of the data block being transferred (as conveyed in the Lower Address field of the descriptor) is assumed to be ($m*16+1$), for some integer m. The size of the data block is assumed to be n Dwords, where $n = k * 16 + 4$, for some $k > 0$. The straddle option is not valid for 128-bit address aligned transfers, so the timing diagrams assume that the Completions are not straddled on the interface.

In the 128-bit address aligned mode, the delivery of the payload always starts in the beat following the last byte of the descriptor. The first byte of the payload can appear on any of the bytes lanes 16 - 32, based on the address of the first valid byte of the payload. The `m_axis_rc_tkeep` bits are set to 1 contiguously from the first Dword of the descriptor until the last Dword of the payload. The alignment of the first Dword on the data bus within its 128-bit field is determined by the setting of the `addr_offset[1:0]` input of the requester request interface when the user application sent the request to the core. The user application are optionally use the byte enable outputs `byte_en[63:0]` to determine the valid bytes in the payload.

Figure 90: **Transfer of a Completion with Data on the Requester Completion Interface (128-bit Interface, Address Aligned Mode)**



Straddle Option for RC Interface

The RC interface of the PCIe® core has the capability to start up to four Completions in the same beat on the requester completion interface. This straddle option is enabled during core customization in the Vivado® IDE. The straddle option can be used only with the Dword-aligned mode.

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

- `is_sop[3:0]`: The core sets this output to a non-zero value in a beat when there is at least one Completion TLP starting in the beat. When straddle is disabled, only `is_sop[0]` is valid and `is_sop[3:1]` are permanently set to 0. When straddle is enabled, the settings are as follows:
    - `0000`: No new TLP starting in this beat
    - `0001`: A single new TLP starts in this beat. Its start position is indicated by `is_sop0_ptr[1:0]`.
    - `0011`: Two new TLPs are starting in this beat. `is_sop0_ptr[1:0]` provides the starting position of the first TLP and `is_sop1_ptr[1:0]` provides the starting position of the second TLP.
    - `0111`: Three new TLPs are starting in this beat. `is_sop0_ptr[1:0]` provides the starting position of the first TLP, `is_sop1_ptr[1:0]` provides the starting position of the second TLP, and `is_sop2_ptr[1:0]` provides the starting position of the third TLP.
    - `1111`: Four new TLPs are starting in this beat. `is_sop0_ptr[1:0]` provides the starting position of the first TLP, `is_sop1_ptr[1:0]` provides the starting position of the second TLP, `is_sop2_ptr[1:0]` provides the starting position of the third TLP, and `is_sop3_ptr[1:0]` provides the starting position of the fourth TLP.
    - All other settings are reserved.
- `is_sop0_ptr[1:0]`: When `is_sop[0]` is set, this field indicates the offset of the first Completion TLP starting in the current beat. Valid settings are `2'b00` (TLP starting at Dword 0), and `2'b01` (TLP starting at Dword 4), `2'b10` (TLP starting at Dword 8), and `2'b11` (TLP starting at Dword 12).
- `is_sop1_ptr[1:0]`: When `is_sop[1]` is set, this field indicates the offset of the second Completion TLP starting in the current beat. Valid settings are `2'b01` (TLP starting at Dword 4), `2'b10` (TLP starting at Dword 8), and `2'b11` (TLP starting at Dword 12).
- `is_sop2_ptr[1:0]`: When `is_sop[2]` is set, this field indicates the offset of the third Completion TLP starting in the current beat. Valid settings are `2'b10` (TLP starting at Dword 8), and `2'b11` (TLP starting at Dword 12).
- `is_sop3_ptr[1:0]`: When `is_sop[3]` is set, this field indicates the offset of the fourth Completion TLP starting in the current beat. Its only valid setting is `2'b11` (TLP starting at Dword 12).

- `is_eop[3:0]`: These outputs signals that one or more TLPs are ending in this beat. These outputs are set in the final beat of a TLP. When straddle is disabled, only `is_eop[0]` is valid and `is_eop[3:1]` are permanently set to 0. When straddle is enabled, the settings are as follows:
    - `0000`: No TLPs are ending in this beat.
    - `0001`: A single TLP is ending in this beat. The setting of `is_eop0_ptr[3:0]` provides the offset of the last Dword of this TLP.
    - `0011`: Two TLPs are ending in this beat. `is_eop0_ptr[3:0]` provides the offset of the last Dword of the first TLP and `is_eop1_ptr[3:0]` provides the offset of the last Dword of the second TLP.
    - `0111`: Three TLPs are ending in this beat. `is_eop0_ptr[3:0]` provides the offset of the last Dword of the first TLP, `is_eop1_ptr[3:0]` provides the offset of the last Dword of the second TLP, and `is_eop2_ptr[3:0]` provides the offset of the last Dword of the third TLP.
    - `1111`: Four TLPs are ending in this beat. `is_eop0_ptr[3:0]` provides the offset of the last Dword of the first TLP, `is_eop1_ptr[3:0]` provides the offset of the last Dword of the second TLP, `is_eop2_ptr[3:0]` provides the offset of the last Dword of the third TLP, and `is_eop3_ptr[3:0]` provides the offset of the last Dword of the fourth TLP.
    - All other settings are reserved.
- `is_eop0_ptr[3:0]`: When `is_eop[0]` is set, this field provides the offset of the last Dword of the first TLP ending in this beat. It can take any value from 0 through 15. The offset for the last byte can be determined from the starting address and length of the TLP, or from the byte enable signals `byte_en[63:0]`.
- `is_eop1_ptr[3:0]`: When `is_eop[1]` is set, this field provides the offset of the last Dword of the second TLP ending in this beat. It can take any value from 6 through 15.
- `is_eop2_ptr[3:0]`: When `is_eop[2]` is set, this field provides the offset of the last Dword of the third TLP ending in this beat. It can take any value from 10 through 15.
- `is_eop3_ptr[3:0]`: When `is_eop[3]` is set, this field provides the offset of the last Dword of the fourth TLP ending in this beat. It can take values of 14 or 15.

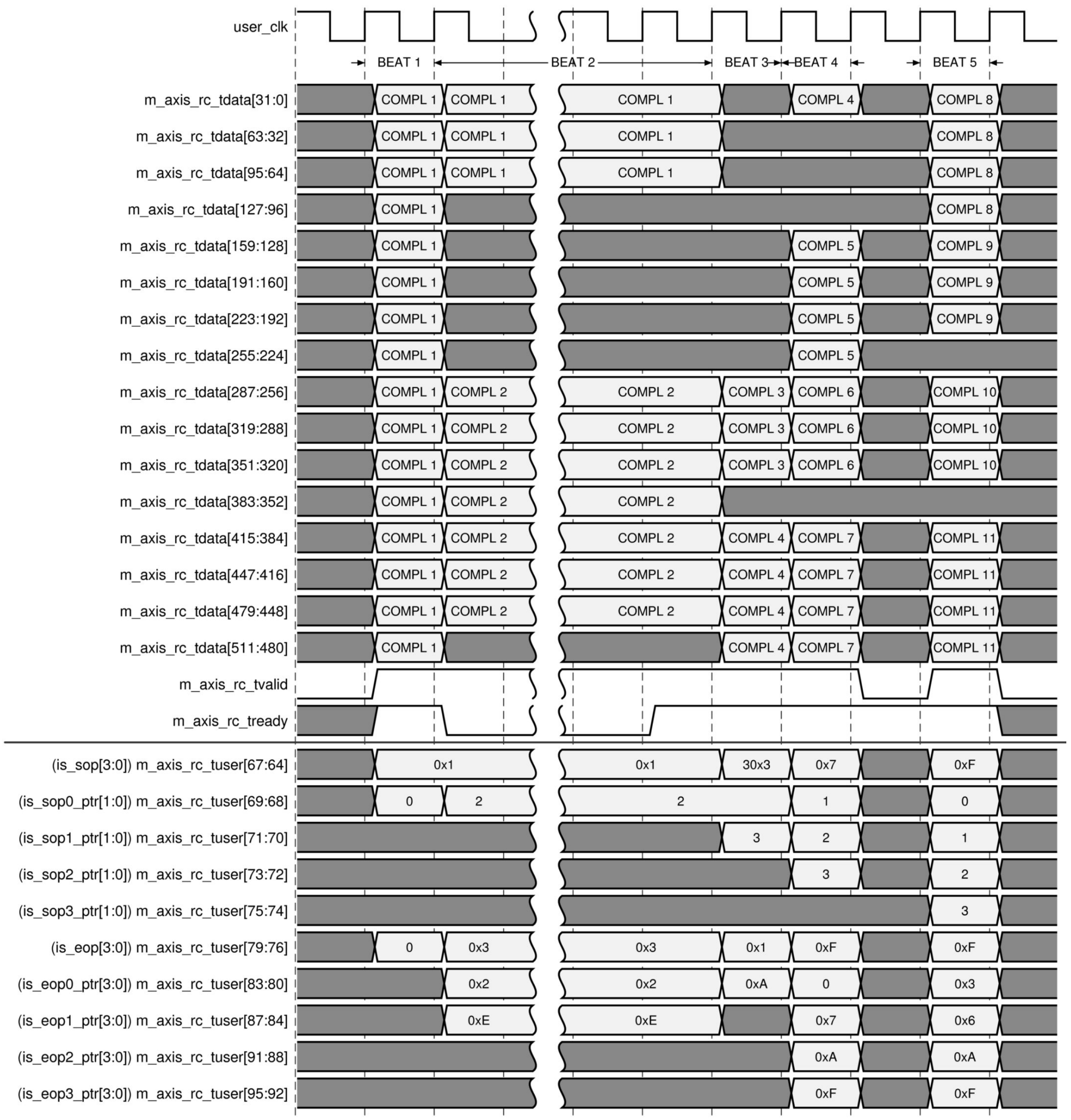
The following figure illustrates the transfer of 11 Completion TLPs on the requester completion interface when the straddle option is enabled. The first Completion TLP (COMPL 1) starts at Dword position 0 of Beat 1 and ends in Dword position 2 of Beat 2. The second TLP (COMPL 2) starts in Dword position 8 of the same beat and ends in Dword position 14. Thus, there is one TLP starting in Beat 1, whose starting position is indicated by is_sop0_ptr, and two TLPs ending, whose ending Dword positions are indicated by `is_eop0_ptr` and `is_eop1_ptr`, respectively.

Beat 3 has COMPL 3 starting at Dword offset 8, ending at Dword offset10. There is also a second TLP (CMPL 4) in the same beat, starting at Dword offset 12 and continuing to the next beat. In this beat, `is_sop0_ptr` points to the starting Dword offset of COMPL 3 and `is_sop1_ptr` points to the starting Dword offset of COMPL 4. `is_eop0_ptr` points to the offset of the last Dword offset of COMPL 4.

Beat 4 has COMPL 4 ending with Dword offset 0, and has three new complete TLPs in it (COMPL 5, 6 and 7). The starting Dword offsets of the new Completions 5, 6 and 7 are provided by `is_sop0_ptr`, `is_sop1_ptr`, and `is_sop2_ptr`, respectively. The ending offsets of Completions 4, 5, 6 and 7 are indicated by `is_eop0_ptr`, `is_eop1_ptr`, `is_eop2_ptr` and `is_eop3_ptr`, respectively.

Finally, Beat 5 contains four complete TLPs (COMPL 8 – 11). Their starting Dword offsets are signaled by `is_sop0_ptr`, `is_sop1_ptr`, `is_sop2_ptr` and `is_sop3_ptr`, respectively. The ending offsets are indicated by `is_eop0_ptr`, `is_eop1_ptr`, `is_eop2_ptr` and `is_eop3_ptr`, respectively. Thus, all the four SOP and EOP pointers provide valid information in this beat.

Figure 91: **Transfer of Completion TLPs on the Requester Completion Interface with the Straddle Option Enabled**



Aborting a Completion Transfer

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

When the straddle option is in use, the core does not start a new Completion TLP in the same beat when it has asserted discontinue to abort the Completion TLP ending in the beat.

Handling of Completion Errors

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

- `0010`: Request terminated by a Completion TLP with UR, CA or CRS status. In this case, there is no data associated with the completion, and the Request Completed bit in the completion descriptor is set. On receiving such a Completion from the core, the user logic can discard the corresponding request.
- `0011`: Read Request terminated by a Completion TLP with incorrect byte count. This condition occurs when a Completion TLP is received with a byte count not matching the expected count. The Request Completed bit in the completion descriptor is set. On receiving such a completion from the core, the user logic can discard the corresponding request.
- `0100`: This code indicates the case when the current Completion being delivered has the same tag of an outstanding request, but its Requester ID, TC, or Attr fields did not match with the parameters of the outstanding request. The user logic should discard any data that follows the descriptor. In addition, if the Request Completed bit in the descriptor is not set, the user logic should continue to discard the data subsequent completions for this tag until it receives a completion descriptor with the Request Completed bit set. On receiving a completion descriptor with the Request Completed bit set, the user logic can remove all state associated with the request.
- `0101`: Error in starting address. The low address bits in the Completion TLP header did not match with the starting address of the next expected byte for the request. The user logic should discard any data that follows the descriptor. In addition, if the Request Completed bit in the descriptor is not set, the user logic should continue to discard the data subsequent Completions for this tag until it receives a completion descriptor with the Request Completed bit set. On receiving a completion descriptor with the Request Completed bit set, the user logic can discard the corresponding request.

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

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

***Note*:** Each parity bit corresponds to parity of one byte in AXIS `tdata`. There are 64 bit parity bits corresponding to the 512 bit AXI `tdata` (and 32 bit parity bits corresponding to 256 bit AXI `tdata`). The received parity bits, on `m_axis_cq_tuser` and `m_axis_rc_tuser` signals, are valid for the following::

- Descriptor bytes in AXIS tdata.
- The valid payload byte in AXIS `tdata` indicated by the `byte_en` field in the AXIS `tuser`. For example, if `byte_en[63:0]=0x0000_0000_0000_FFFF`, then only lower 16 parity bits are valid. If `byte_en[63:0] = 0xFFFF_FFFF_FFFF_FFFF`, then all 64 parity bits are enabled.

Power Management

The core supports these power management modes:

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

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

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

Active State Power Management

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

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

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

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

Programmed Power Management

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

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

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

PPM L0 State

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

PPM L1 State

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

1. The transition to a lower power PPM L1 state is always initiated by an upstream device, by programming the PCI Express® device power state to D3-hot (or to D1 or D2, if they are supported).
2. The device power state is communicated to the user logic through the `cfg_function_power_state` output.
3. The core then throttles/stalls the user logic from initiating any new transactions on the user interface by deasserting `s_axis_rq_tready`. Any pending transactions on the user interface are, however, accepted fully and can be completed later.
    - The core is configured as an Endpoint and the User Configuration Space is enabled. In this situation, the user application must refrain from sending new Request TLPs if `cfg_function_power_state` indicates non-D0, but the user application can return Completions to Configuration transactions targeting User Configuration space.
    - The core is configured as a Root Port. To be compliant in this situation, the user application should refrain from sending new Requests if `cfg_function_power_state` indicates non-D0.
4. The core exchanges appropriate power management DLLPs with its link partner to successfully transition the link to a lower power PPM L1 state. This action is transparent to the user logic.
5. All user transactions are stalled for the duration of time when the device power state is non-D0, with the exceptions indicated in step 3.

PPM L3 State

These steps outline the transition of the Endpoint for PCI Express® to the PPM L3 state:

1. The core negotiates a transition to the L23 Ready Link State upon receiving a PME_Turn_Off message from the upstream link partner.
2. Upon receiving a PME_Turn_Off message, the core initiates a handshake with the user logic through cfg_power_state_change_interrupt (as shown in the following table) and expects a `cfg_power_state_change_ack` back from the user logic.
3. A successful handshake results in a transmission of the Power Management Turn-off Acknowledge (`PME-turnoff_ack`) Message by the core to its upstream link partner.

4. The core closes all its interfaces, disables the Physical/Data-Link/Transaction layers and is ready for removal of power to the core.

   There are two exceptions to this rule:

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

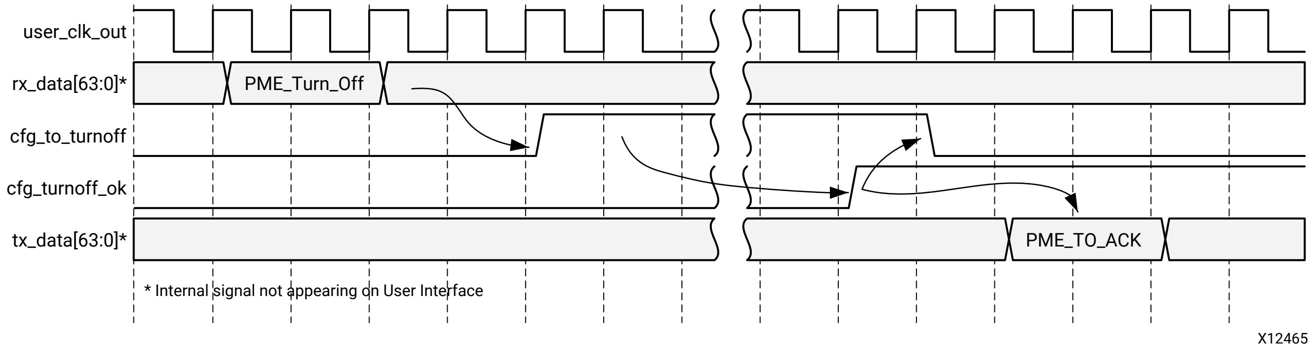
Table 62: **Power Management Handshaking Signals**

Port Name	Direction	Description
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 92: **Power Management Handshaking: 64-Bit**



Generating Interrupt Requests

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

***Note*:** This section only applies to the Endpoint Configuration of the Versal ACAP Integrated Block for PCIe® core.

The core supports sending interrupt requests as either legacy, Message MSI, or MSI-X interrupts. The mode is programmed using the MSI Enable bit in the Message Control register of the MSI Capability Structure and the MSI-X Enable bit in the MSI-X Message Control register of the MSI-X Capability Structure.

The state of the MSI Enable and MSI-X Enabled bits is reflected by the `cfg_interrupt_msi_enable` and `cfg_interrupt_msix_enable` outputs, respectively. The following table describes the Interrupt Mode to which the device has been programmed, based on the `cfg_interrupt_msi_enable` and `cfg_interrupt_msix_enable` outputs of the core.

Table 63: **Interrupt Modes**

	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 core can be configured to advertise multiple interrupt modes support, however at run time, only one interrupt mode can be enabled at a time across all functions. Xilinx does not recommend enabling multiple interrupt modes at once, however in the event that MSI and MSI-X interrupts simultaneous enablement cannot be avoided, MSI-X interrupt must be implemented externally of the core and interrupt packet is formed and sent through the Requester Request Interface Port (`s_axis_rq`).

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

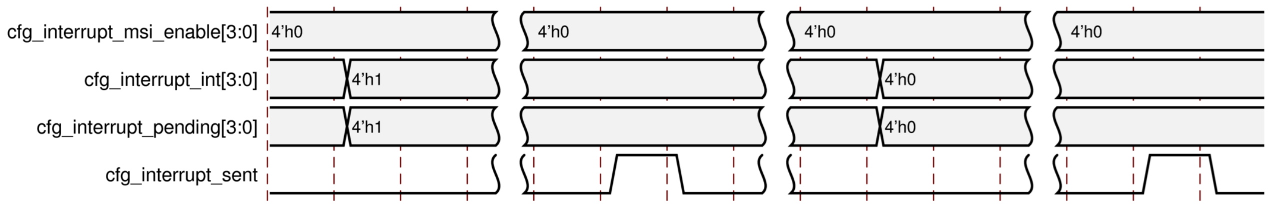
Legacy Interrupt Mode

- The user application first asserts `cfg_interrupt_int` and `cfg_interrupt_pending` to assert the interrupt.
- The core then asserts cfg_interrupt_sent to indicate the interrupt is accepted. If the Interrupt Disable bit in the PCI Command register is set to 0, the core sends an assert interrupt message (`Assert_INTA`). After the interrupt has been serviced, the user application `deasserts cfg_interrupt_int`.
- After the user application `deasserts cfg_interrupt_int`, the core sends a deassert interrupt message (`Deassert_INTA`). This is indicated by the assertion of `cfg_interrupt_sent` a second time.

`cfg_interrupt_int` must be asserted until the user application receives confirmation of the assert interrupt message (`Assert_INTA`), which is indicated by the assertion of `cfg_interrupt_sent`, and the interrupt has been serviced/cleared by the Root's Interrupt Service Routine (ISR). Deasserting `cfg_interrupt_int` causes the core to send the deassert interrupt message (`Deassert_INTA`). `cfg_interrupt_pending` must be asserted along with the assertion of `cfg_interrupt_int` until the interrupt has been serviced, otherwise, the interrupt status bit in the status register is not updated correctly. `cfg_interrupt_pending` can be deasserted along with the deassertion of `cfg_interrupt_int` after the first assertion of `cfg_interrupt_sent`. When the software/Root's ISR receives an assert interrupt message, it reads this interrupt status bit to determine whether there is an interrupt pending for this function.

Note: For PCIE4C block, INTx interrupts are not blocked by the core when the interrupt disable bit is set in command register, i.e.,Command Register bit 10 = 1. The user application must monitor `cfg_function_status` to check whether INTx interrupts are enabled or disabled, and assert `cfg_interrupt_int` only if interrupts are enabled in the command register.

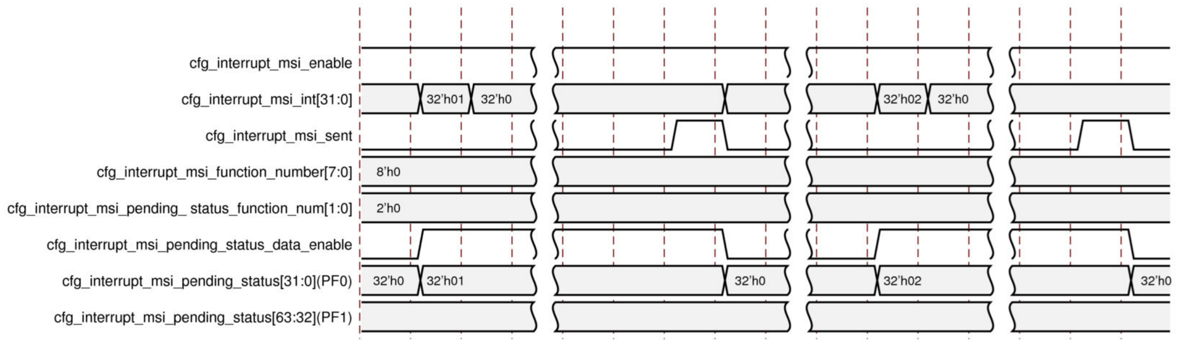
Figure 93: **Legacy Interrupt Signaling**



MSI Mode

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

Figure 94: **MSI Mode**


cfg_interrupt_msi_enable
cfg_interrupt_msi_int[31:0]
cfg_interrupt_msi_sent
cfg_interrupt_msi_function_number[7:0]
cfg_interrupt_msi_pending_ status_function_num[1:0]
cfg_interrupt_msi_pending_status_data_enable
cfg_interrupt_msi_pending_status[31:0](PF0)
cfg_interrupt_msi_pending_status[63:32](PF1)
32'h01
32'h0
32'h02
32'h0
8'h0
2'h0
32'h0
32'h01
32'h0
32'h02
32'h0

The MSI request is either a 32-bit addressable Memory Write TLP or a 64-bit addressable Memory Write TLP. The address is taken from the Message Address and Message Upper Address fields of the MSI Capability Structure, while the payload is taken from the Message Data field. These values are programmed by system software through configuration writes to the MSI Capability structure. When the core is configured for Multi-Vector MSI, system software can permit Multi-Vector MSI messages by programming a non-zero value to the Multiple Message Enable field.

The type of MSI TLP sent (32-bit addressable or 64-bit addressable) depends on the value of the Upper Address field in the MSI capability structure. By default, MSI messages are sent as 32-bit addressable Memory Write TLPs. MSI messages use 64-bit addressable Memory Write TLPs only if the system software programs a non-zero value into the Upper Address register.

When Multi-Vector MSI messages are enabled, the user application can override one or more of the lower-order bits in the Message Data field of each transmitted MSI TLP to differentiate between the various MSI messages sent upstream. The number of lower-order bits in the Message Data field available to the user application is determined by the lesser of the value of the Multiple Message Capable field, as set in the IP catalog, and the Multiple Message Enable field, as set by system software and available as the `cfg_interrupt_msi_mmenable[2:0]` core output. The core masks any bits in `cfg_interrupt_msi_select` which are not configured by system software through Multiple Message Enable.

This pseudo code shows the processing required:

```
// Value MSI_Vector_Num must be in range: 0 £ MSI_Vector_Num £
(2^cfg_interrupt_mmenable)-1

if (cfg_interrupt_msienable) {        // MSI Enabled
  if (cfg_interrupt_mmenable > 0) {   // Multi-Vector MSI Enabled
    cfg_interrupt_msi_int[MSI_Vector_Num] = 1;
  } else {                            // Single-Vector MSI Enabled
    cfg_interrupt_msi_int[MSI_Vector_Num] = 0;
  }
} else {
  // Legacy Interrupts Enabled
}
```

For example:

1. If `cfg_interrupt_mmenable[2:0] == 000b`, that is, 1 MSI Vector Enabled, then `cfg_interrupt_msi_int = 01h;`
2. If `cfg_interrupt_mmenable[2:0] == 101b`, that is, 32 MSI Vectors Enabled, then `cfg_interrupt_msi_int = {32'b1 << {MSI_Vector#}};`

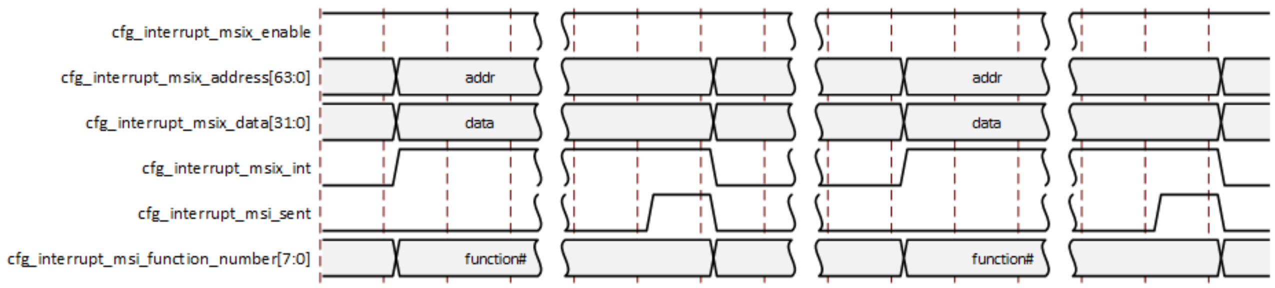
where `MSI_Vector#` is a 5-bit value and is allowed to be `00000b ≤ MSI_Vector# ≤ 11111b`.

If Per-Vector Masking is enabled, first verify that the vector being signaled is not masked in the Mask register. This is done by reading this register on the Configuration interface (the core does not look at the Mask register).

MSI-X Mode

The core supports the MSI-X interrupt and its signaling, which is shown in the following figure. The MSI-X vector table and the MSI-X Pending Bit Array need to be implemented as part of the user logic, by claiming a BAR aperture if the built-in MSI-X vector tables are not used.

Figure 95: **MSI-X Mode**

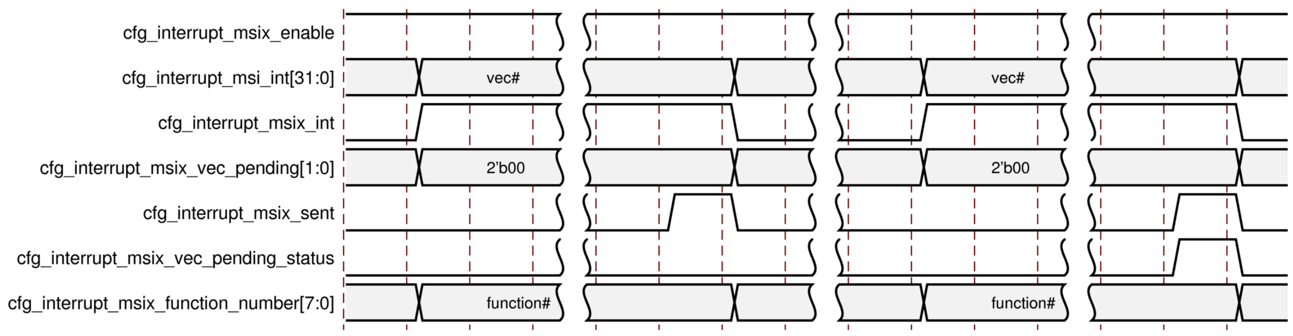

cfg_interrupt_msix_enable
cfg_interrupt_msix_address[63:0]
cfg_interrupt_msix_data[31:0]
cfg_interrupt_msix_int
cfg_interrupt_msi_sent
cfg_interrupt_msi_function_number[7:0]
addr
data
function#
addr
data
function#

MSI-X Mode with Built-in MSI-X Vector Tables

The core optionally supports built-in MSI-X vector tables including the Pending Bit Array.

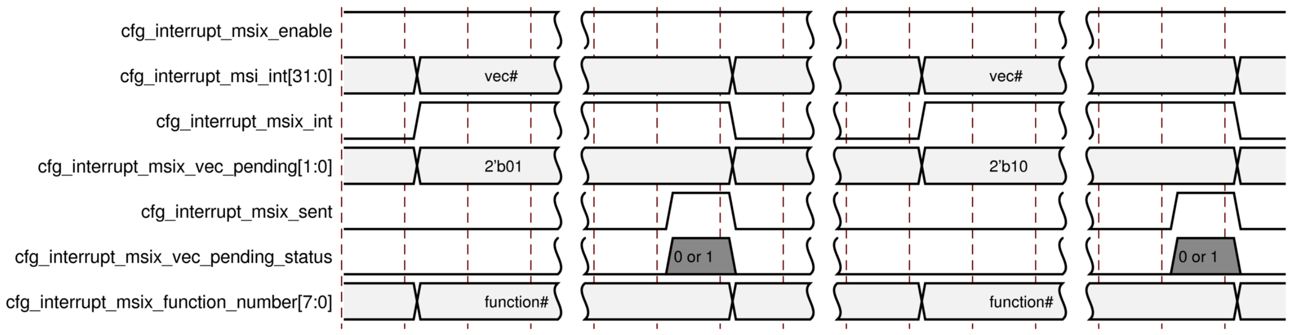
- As shown in the following figure, the user application first asserts `cfg_interrupt_msix_int` with the vector number set in `cfg_interrupt_msi_int`.
- The core asserts `cfg_interrupt_msi_sent` to signal that the interrupt is accepted. If `cfg_interrupt_msix_vec_pending_status` is clear, the core sends a MSI-X Memory Write TLP. Otherwise, the core waits to send a MSI-X Memory Write TLP until the function mask is cleared.

Figure 96: **MSI-X Signaling with Built-In MSI-X Vector Tables**


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

- Instead of generating an interrupt, the user application can query or clear the Pending Bit Array by additionally setting `cfg_interrupt_msix_vec_pending` to `2'b01` or `2'b10` respectively, as shown in the following figure.
- In the query and clear cases, `cfg_interrupt_msix_vec_pending_status` reflects the pending status before the query or clear.
- `cfg_interrupt_msi_int[31:0]` is a shared signal between MSI [31:0] and MSI-X [7:0].

Figure 97: **MSI-X Pending Bit Array Query and Clear**


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

***Note*:** Applications that need to generate MSI/MSI-X interrupts with traffic class bits not equal to 0 or address translation bits not equal to 0 must use the RQ interface to generate the interrupt (memory write descriptor).

Receive Message Interface

The core provides a separate receive-message interface which the user application can use to receive indications of messages received from the link. When the receive message interface is enabled, the integrated block signals the arrival of a message from the link by setting the `cfg_msg_received_type[4:0]` output to indicate the type of message (see the following table) and pulsing the `cfg_msg_received` signal for one or more cycles. The duration of assertion of cfg_msg_received is determined by the type of message received (see Table 64: Message Type Encoding on Receive Message Interface). When `cfg_msg_received` is active-High, the integrated block transfers any parameters associated with the message on the bus 8 bits at a time on the bus `cfg_msg_received_data`. The parameters transferred on this bus in each cycle of `cfg_msg_received` assertion for various message types are listed in the Table 65: Message Parameters on Receive Message Interface table. For Vendor-Defined Messages, the integrated block transfers only the first Dword of any associated payload across this interface. When larger payloads are in use, the completer request interface should be used for the delivery of messages.

Table 64: **Message Type Encoding on Receive Message Interface**

cfg_msg_received_type[4:0]	Message Type
0	ERR_COR
1	ERR_NONFATAL
2	ERR_FATAL
3	Assert_INTA
4	Deassert_ INTA
5	Assert_INTB
6	Deassert_ INTB
7	Assert_INTC
8	Deassert_ INTC
9	Assert_INTD
10	Deassert_ INTD
11	PM_PME
12	PME_TO_Ack
13	PME_Turn_Off
14	PM_Active_State_Nak
15	Set_Slot_Power_Limit
16	Latency Tolerance Reporting (LTR)
17	Optimized Buffer Flush/Fill (OBFF)
18	Unlock
19	Vendor_Defined Type 0
20	Vendor_Defined Type 1
21	ATS Invalid Request
22	ATS Invalid Completion
23	ATS Page Request
24	ATS PRG Response
25 – 31	Reserved

Table 65: **Message Parameters on Receive Message Interface**

Message Type	Number of Cycles of cfg_msg_received Assertion	Parameter Transferred on cfg_msg_received_data[7:0]
ERR_COR, ERR_NONFATAL, ERR_FATAL	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
Assert_INTx, Deassert_INTx	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
PM_PME, PME_TO_Ack, PME_Turn_off, PM_Active_State_Nak	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number

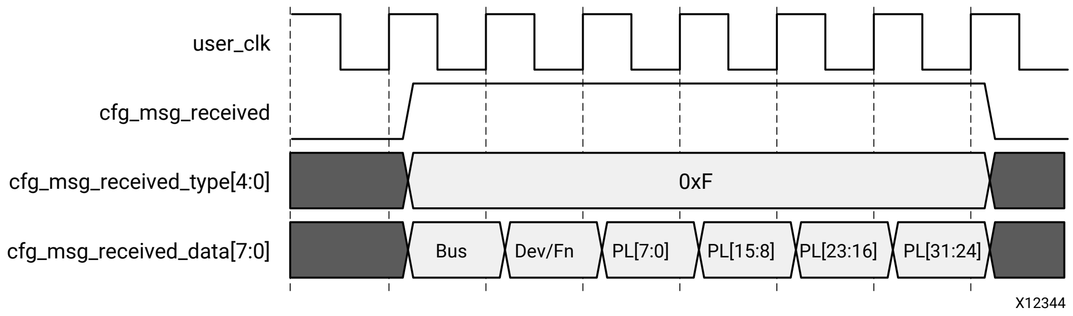
Table 65: **Message Parameters on Receive Message Interface** *(cont'd)*

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

The following timing diagram showing the example of a `Set_Slot_Power_Limit` message on the receive message interface. This message has an associated one-Dword payload. For this message, the parameters are transferred over six consecutive cycles. The following information appears on the `cfg_msg_received_data` bus in each cycle:

- Cycle 1: Bus number of Requester ID
- Cycle 2: Device/Function Number of Requester ID
- Cycle 3: Bits [7:0] of the payload Dword
- Cycle 4: Bits [15:8] of the payload Dword
- Cycle 5: Bits [23:16] of the payload Dword
- Cycle 6: Bits [31:24] of the payload Dword

Figure 98: **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 Management Interface is described in Configuration Management Interface. Root Ports must use the Configuration Management Interface to set up the Configuration Space. Endpoints can also use the Configuration Management Interface to read and write; however, care must be taken to avoid adverse system side effects.

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

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

For example:

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

- The Dword address of is `01h`.

  Note: The byte address is `04h`.

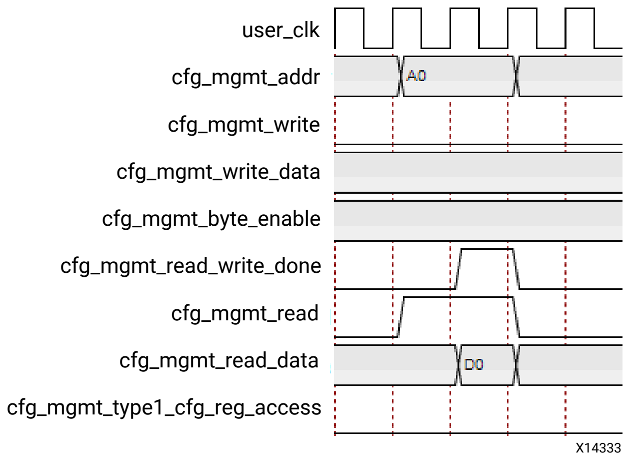
For BAR0:

- The Dword address is `04h`.

  Note: The byte address is `10h`.

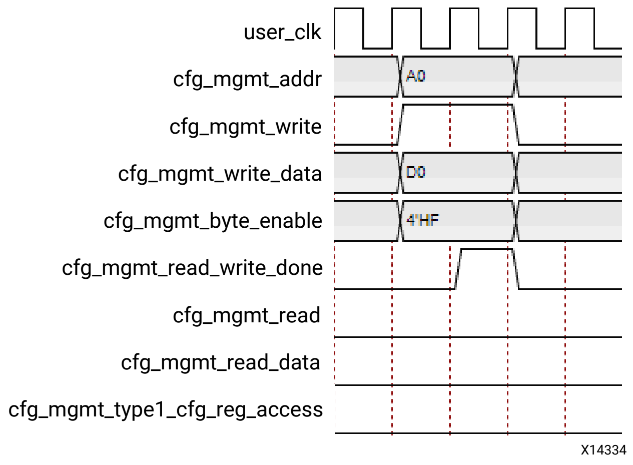
To read any register in configuration space, the user application drives the register Dword address onto `cfg_mgmt_addr[9:0]`. `cfg_mgmt_function_number[7:0]` selects the PCI Function associated with the configuration register. The core drives the content of the addressed register onto `cfg_mgmt_read_data[31:0]`. The value on `cfg_mgmt_read_data[31:0]` is qualified by signal assertion on `cfg_mgmt_read_write_done`. The following figure illustrates an example with read from the Configuration Space.

Figure 99: **cfg_mgmt_read_type0_type1**



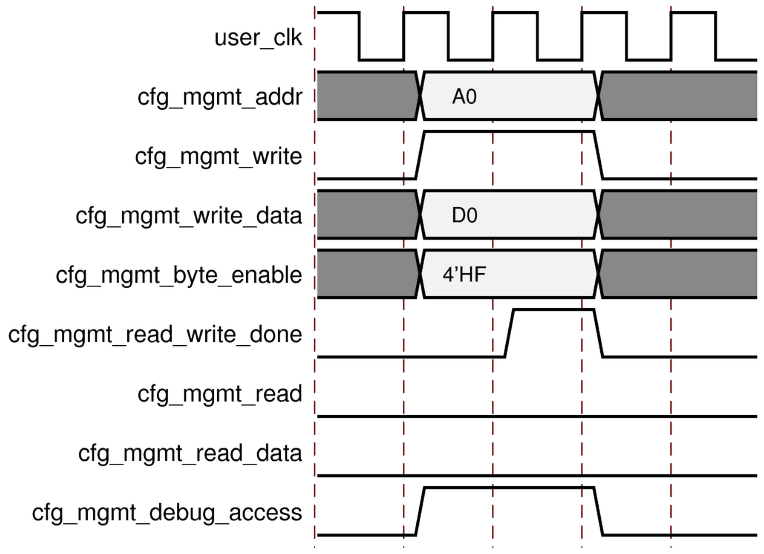
To write any register in configuration space, the user logic places the address on the `cfg_mgmt_addr[9:0]`, the function number on `cfg_mgmt_function_number[7:0]`, write data on `cfg_mgmt_write_data`, byte-valid on `cfg_mgmt_byte_enable [3:0]`, and asserts the `cfg_mgmt_write` signal. In response, the core asserts the `cfg_mgmt_read_write_done` signal when the write is complete (which can take several cycles). The user logic must keep `cfg_mgmt_addr`, `cfg_mgmt_function_number`, `cfg_mgmt_write_data`, `cfg_mgmt_byte_enable` and `cfg_mgmt_write` stable until `cfg_mgmt_read_write_done` is asserted. The user logic must also deassert `cfg_mgmt_write` in the cycle following the `cfg_mgmt_read_write_done` from the core.

Figure 100: **cfg_mgmt_write_type0**



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

Figure 101: **cfg_mgmt_debug_access**



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

The 2-lane, 4-lane, and 8-lane cores can operate at less than the maximum lane width as required by the PCI Express® Base Specification. Two cases cause the core to operate at less than its specified maximum lane width, as defined in these subsections.

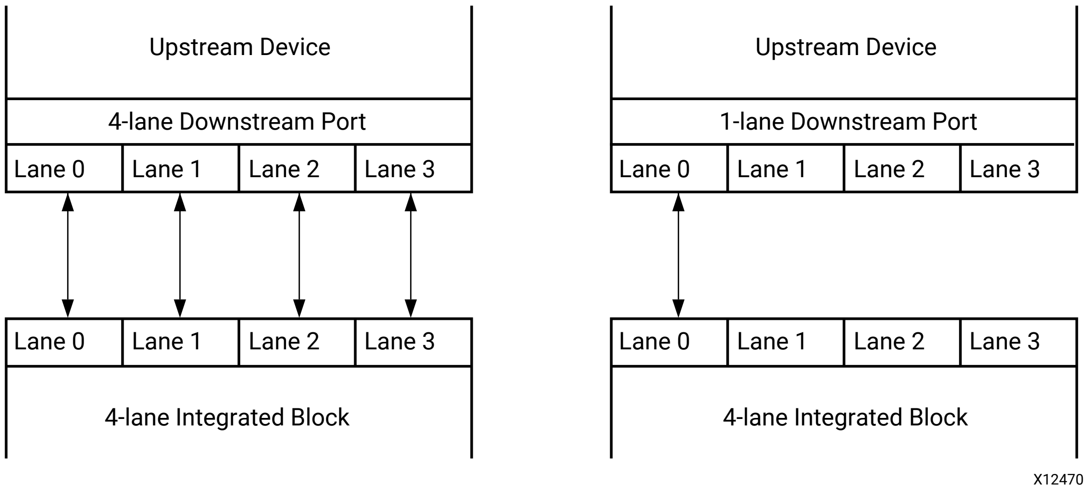
Link Partner Supports Fewer Lanes

When the 2-lane core is connected to a device that implements only 1 lane, the 2-lane core trains and operates as a 1-lane device using lane 0.

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

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

Figure 102: **Scaling of 4-Lane Endpoint Block from 4-Lane to 1-Lane Operation**



Lane Becomes Faulty

If a link becomes faulty after training to the maximum lane width supported by the core and the link partner device, the core attempts to recover and train to a lower lane width, if available. If lane 0 becomes faulty, the link is irrecoverably lost. If any or all of lanes 1–7 become faulty, the link goes into recovery and attempts to recover the largest viable link with whichever lanes are still operational.

For example, when using the 8-lane core, loss of lane 1 yields a recovery to 1-lane operation on lane 0, whereas the loss of lane 6 yields a recovery to 4-lane operation on lanes 0-3. After recovery occurs, if the failed lane(s) becomes alive again, the core does not attempt to recover to a wider link width. The only way a wider link width can occur is if the link actually goes down and it attempts to retrain from scratch.

The `user_clk` clock output is a fixed frequency configured in IP catalog. `user_clk` does not shift frequencies in case of link recovery or training down.

Lane Reversal

The integrated block supports limited lane reversal capabilities and therefore provides flexibility in the design of the board for the link partner. The link partner can choose to lay out the board with reversed lane numbers and the integrated block continues to link train successfully and operate normally. The configurations that have lane reversal support are 16x, x8 and x4 (excluding downshift modes). Downshift refers to the link width negotiation process that occurs when link partners have different lane width capabilities advertised. As a result of lane width negotiation, the link partners negotiate down to the smaller of the advertised lane widths. The following table describes the several possible combinations including downshift modes and availability of lane reversal support.

Table 66: **Lane Reversal Support**

Integrated Block Advertised Lane Width	Negotiated Lane Width	Lane Number Mapping (Endpoint Link Partner)		Lane Reversal Supported
		Endpoint	Link Partner	
x16	x16	Lane 0... Lane15	Lane15... Lane 0	Yes
x16	x8	Lane 0... Lane7	Lane7... Lane 0	No
x16	x4	Lane 0... Lane3	Lane3... Lane 0	No
x16	x2	Lane 0... Lane1	Lane1... Lane 0	No
x8	x8	Lane 0... Lane 7	Lane 7... Lane 0	Yes
x8	x4	Lane 0... Lane 3	Lane 7... Lane 4	No[1]
x8	x2	Lane 0... Lane 3	Lane 7... Lane 6	No[1]
x4	x4	Lane 0... Lane 3	Lane 3... Lane 0	Yes

Table 66: **Lane Reversal Support** *(cont'd)*

Integrated Block Advertised Lane Width	Negotiated Lane Width	Lane Number Mapping (Endpoint Link Partner)		Lane Reversal Supported
		Endpoint	Link Partner	
x4	x2	Lane 0... Lane 1	Lane 3... Lane 2	No[1]
x2	x2	Lane 0... Lane 1	Lane 1... Lane 0	Yes
x2	x1	Lane 0... Lane 1	Lane 1	No[1]

Notes:

1. When the lanes are reversed in the board layout and a downshift adapter card is inserted between the Endpoint and link partner, Lane 0 of the link partner remains unconnected (as shown by the lane mapping in this table) and therefore does not link train.

Design Flow Steps

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

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

Customizing and Generating the Core

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

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

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

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

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

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

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

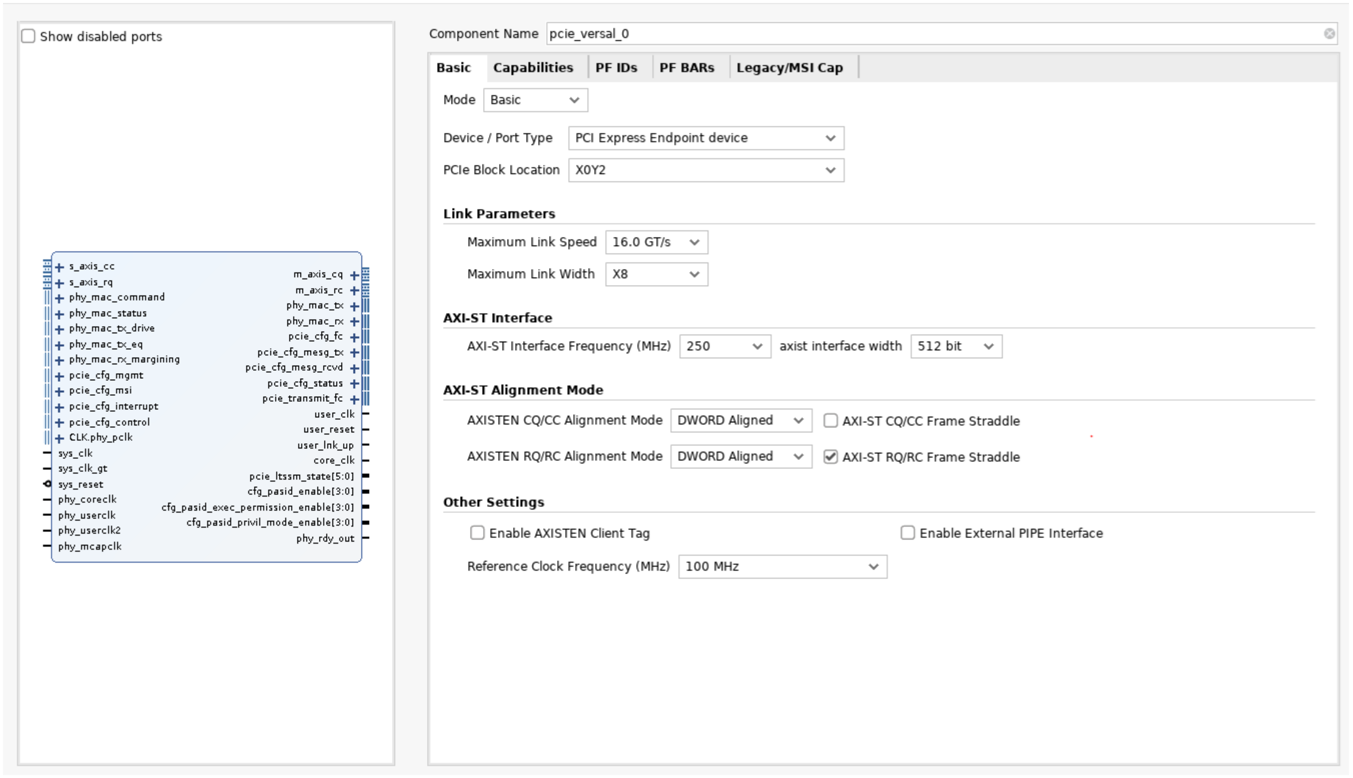
Basic Mode Parameters

The Basic mode parameters are explained in this section.

Basic Tab

The following figure shows the initial customization page, used to set the Basic mode parameters.

Figure 103: **Basic Tab**



Component Name: pcie_versal_0

Mode: Basic

Device / Port Type: PCI Express Endpoint device

PCIe Block Location: X0Y2

Link Parameters

Maximum Link Speed: 16.0 GT/s

Maximum Link Width: X8

AXI-ST Interface

AXI-ST Interface Frequency (MHz): 250

axist interface width: 512 bit

AXI-ST Alignment Mode

AXISTEN CQ/CC Alignment Mode: DWORD Aligned

☐ AXI-ST CQ/CC Frame Straddle

AXISTEN RQ/RC Alignment Mode: DWORD Aligned

☑ AXI-ST RQ/RC Frame Straddle

Other Settings

☐ Enable AXISTEN Client Tag

☐ Enable External PIPE Interface

Reference Clock Frequency (MHz): 100 MHz

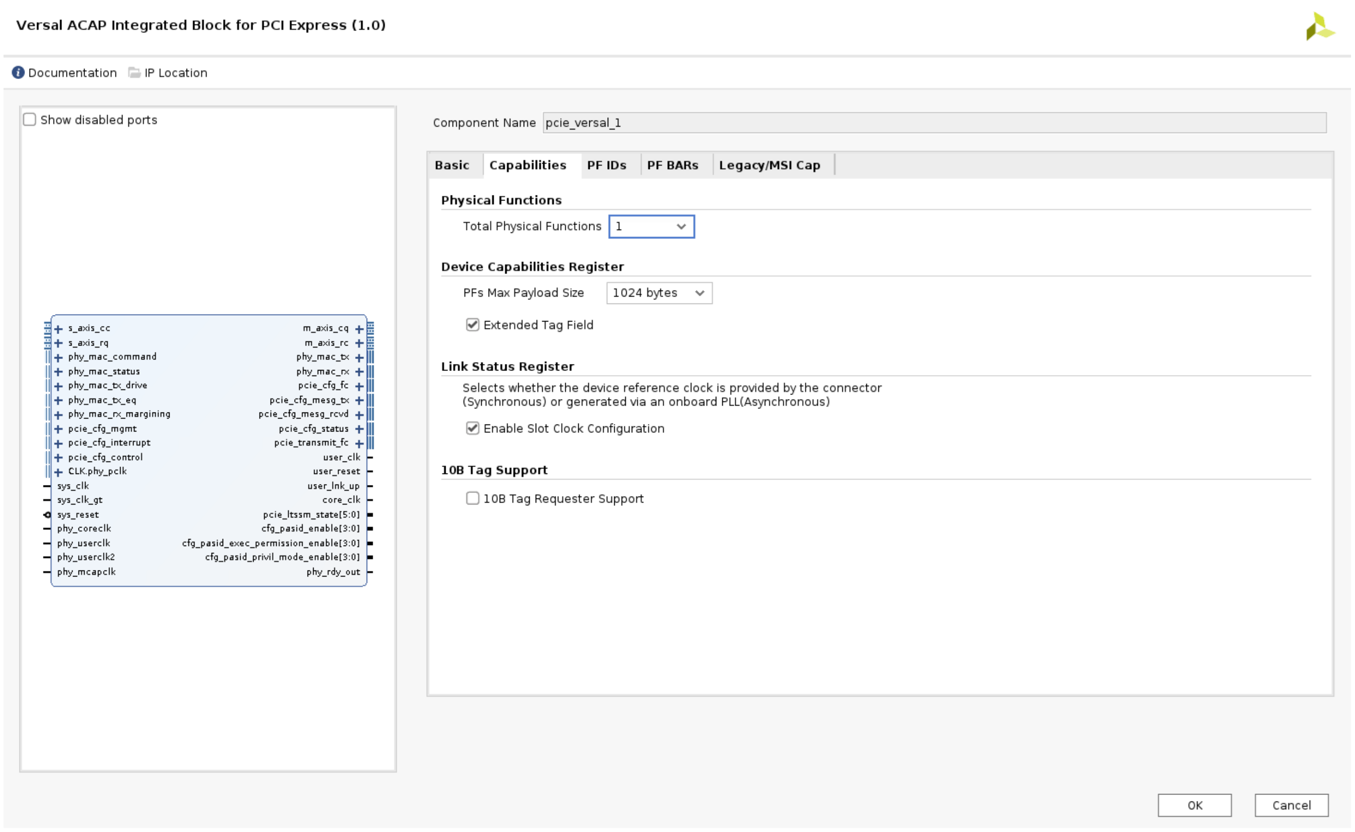
- **Component Name:** Base name of the output files generated for the core. The name must begin with a letter and can be composed of these characters: a to z, 0 to 9, and "_."
- **Mode:** Allows you to select the Basic or Advanced mode configuration of the core.
- **Device/Port Type:** Indicates the PCI Express logical device type.

- **Maximum Link Speed:** The core allows you to select the Maximum Link Speed 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.
- **Maximum Link Width:** The core requires the selection of the initial lane width. See Clock Frequencies and Interface Widths Supported For Various Configurations for the supported link width configuration. Wider lane width cores can train down to smaller lane widths if attached to a smaller lane-width device. For more information, see Link Training: 2-Lane, 4-Lane, 8-Lane, and 16-Lane Components.
- **AXI-ST Interface Frequency:** Enables you to specify the AXI-ST Interface frequency.
- **AXI-ST Interface Width:** The core allows you to select the Interface Width. The default interface width set in the Customize IP dialog box is the lowest possible interface width.
- **AXI-ST Alignment Mode:** When a payload is present, there are two options for aligning the first byte of the payload with respect to the datapath. The options are provided to select the CQ/CC and RQ/RC interfaces.
- **AXI-ST CQ/CC Frame Straddle and AXI-ST RQ/RC Frame Straddle:** When 512-bit AXI-ST interface width is selected AXI-ST frame Straddle is supported for CQ, CC, RQ and RC AXI-ST interfaces. Option to select CQ and CC AXI-ST frame straddle together and for RQ and RC interfaces.
- **Enable Client Tag:** Enables you to use the client tag.
- **Reference Clock Frequency:** Selects the frequency of the reference clock provided on `sys_clk`.
- **Enable External PIPE Interface:** When selected, this option enables an external third-party bus functional model (BFM) to connect to the PIPE interface of integrated block for PCIe. For details, see *PIPE Mode Simulation Using Integrated Endpoint PCI Express Block in Gen3 x8 and Gen2 x8 Configurations* (XAPP1184). Refer to these designs to connect the External PIPE Interface ports of the UltraScale™ device core to third-party BFMs.

Capabilities Tab

The Capabilites settings are explained in this section as shown in the following figure.

Figure 104: **Capabilities Tab**



Versal ACAP Integrated Block for PCI Express (1.0)

☐ Show disabled ports

Component Name pcie_versal_1

Physical Functions

Total Physical Functions 1

Device Capabilities Register

PFs Max Payload Size 1024 bytes

☑ Extended Tag Field

Link Status Register

Selects whether the device reference clock is provided by the connector (Synchronous) or generated via an onboard PLL(Asynchronous)

☑ Enable Slot Clock Configuration

10B Tag Support

☐ 10B Tag Requester Support

- **Total Physical Functions:** Enables you to select the number of physical functions. The number of physical functions supported is 4.
- **PFs Max Payload Size:** This field indicates the maximum payload size that the device or function can support for TLPs. This is the value advertised to the system in the Device Capabilities Register.
- **Extended Tag Field:** This field indicates the maximum supported size of the Tag field. The options are:
  - When selected, 8-bit Tag field support (256 tags)
  - When deselected, 5-bit Tag field support (32 tags)
- **10-bit Tag Field:** This field indicates the maximum supported size of the Tag field as a Requester. The options are:
  - When selected, 10-bit Tag field support (768 tags)
  - When deselected, 8-/5-bit Tag supported, depending on Extended Tag Field selection
- **Enable Slot Clock Configuration:**

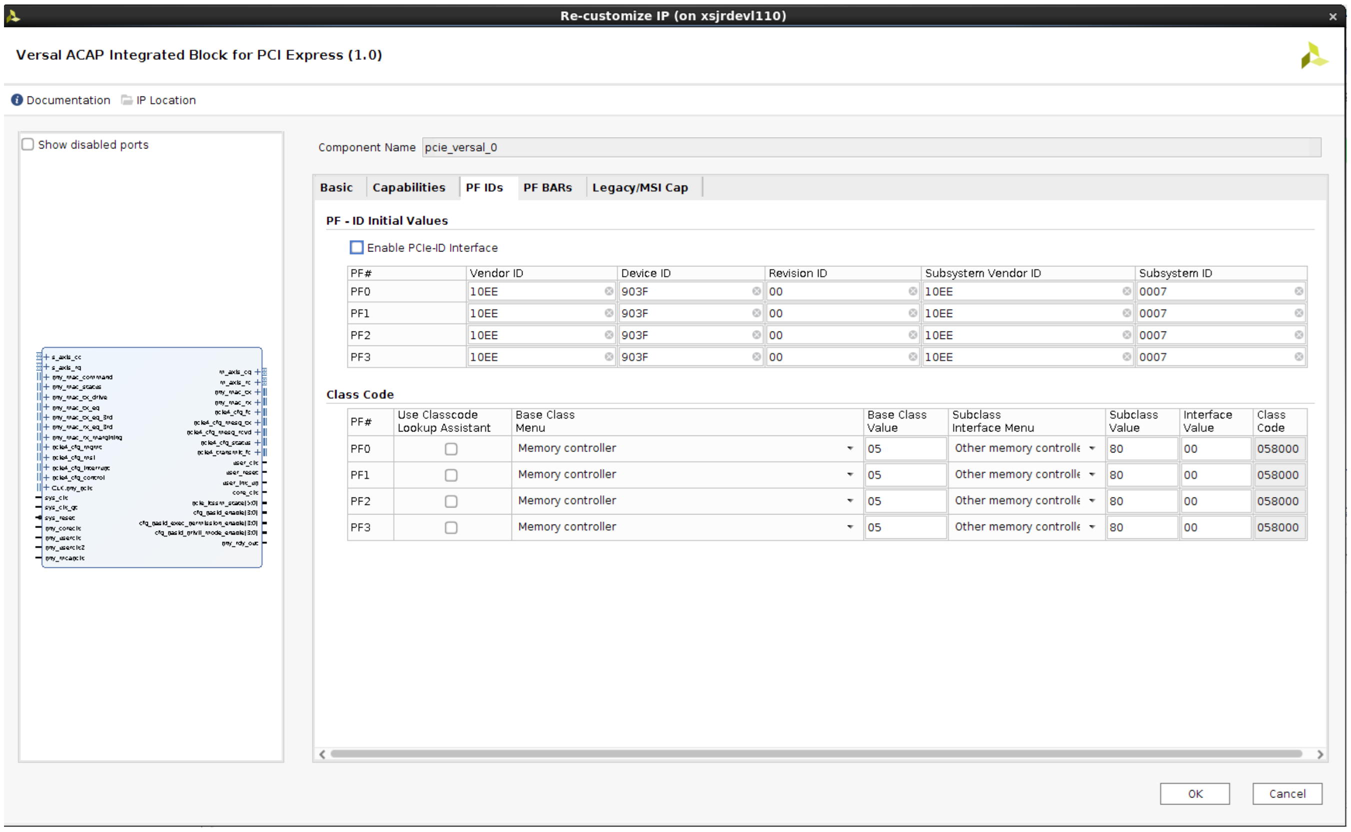
  Enables the Slot Clock Configuration bit in the Link Status register.

- When this option is selected, the link is synchronously clocked.
- When this option is deselected, asynchronous clock in SRNS mode is supported. SRNS refers to a separate reference clock with No SSC (an asynchronous clock without SRIS support).

PF IDs Tab

The following figure shows the Identity Settings parameters.

Figure 105: **PF IDs Tab**



Re-customize IP (on xsjrdevl110)
Versal ACAP Integrated Block for PCI Express (1.0)
Documentation
IP Location
Show disabled ports
OK
Cancel

Component Name pcie_versal_0

Basic | **Capabilities** | **PF IDs** | **PF BARs** | **Legacy/MSI Cap**

PF - ID Initial Values

☐ Enable PCIe-ID Interface

PF#	Vendor ID	Device ID	Revision ID	Subsystem Vendor ID	Subsystem ID
PF0	10EE	903F	00	10EE	0007
PF1	10EE	903F	00	10EE	0007
PF2	10EE	903F	00	10EE	0007
PF3	10EE	903F	00	10EE	0007

Class Code

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

- **Enable PCIe-ID Interface:** If this parameter is selected the PCIe ID ports `cfg_vend_id`, `cfg_subsys_vend_id`, `cfg_dev_id_pf*`, `cfg_rev_id_pf*`, and `cfg_subsys_id_pf*` appears at the boundary of core top depending on the number of PFx that are selected and available to be driven by user logic. If unselected they do not appear at the top level and are driven with the values set at the time of customization.
- **PF0 ID Initial Values:**
    - **Vendor ID:** Identifies the manufacturer of the device or application. Valid identifiers are assigned by the PCI Special Interest Group to guarantee that each identifier is unique. The default value, `10EEh`, is the Vendor ID for Xilinx. Enter a vendor identification number here. `FFFFh` is reserved.

- **Device ID:** A unique identifier for the application; the default value depends on the configuration selected. The default value is B0<link speed><link width>h. This field can be any value; change this value for the application. The default Device ID parameter is based on:

  - The device family: B for Versal ACAP.
  - EP or RP mode.
  - Link width: 1 for x1, 2 for x2, 4 for x4, 8 for x8, and F for x16.
  - Link speed: 1 for Gen1, 2 for Gen2, 3 for Gen3, and 4 for Gen4.

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

  > **RECOMMENDED:** *It is always recommended that the link width, speed, and Device Port type be changed first and then the Device ID value. Make sure the Device ID value is set correctly before generating the IP.*

- **Revision ID:** Indicates the revision of the device or application; an extension of the Device ID. The default value is `00h`; enter a value appropriate for the application.
- **Subsystem Vendor ID:** Further qualifies the manufacturer of the device or application. Enter a Subsystem Vendor ID here; the default value is `10EEh`. Typically, this value is the same as Vendor ID. Setting the value to `0000h` can cause compliance testing issues.
- **Subsystem ID:** Further qualifies the manufacturer of the device or application. This value is typically the same as the Device ID; the default value depends on the lane width and link speed selected. Setting the value to `0000h` can cause compliance testing issues.

- **Class Code:** The Class Code identifies the general function of a device, and is divided into three byte-size fields:

  - **Base Class:** Broadly identifies the type of function performed by the device.
  - **Sub-Class:** More specifically identifies the device function.
  - **Interface:** Defines a specific register-level programming interface, if any, allowing device-independent software to interface with the device.

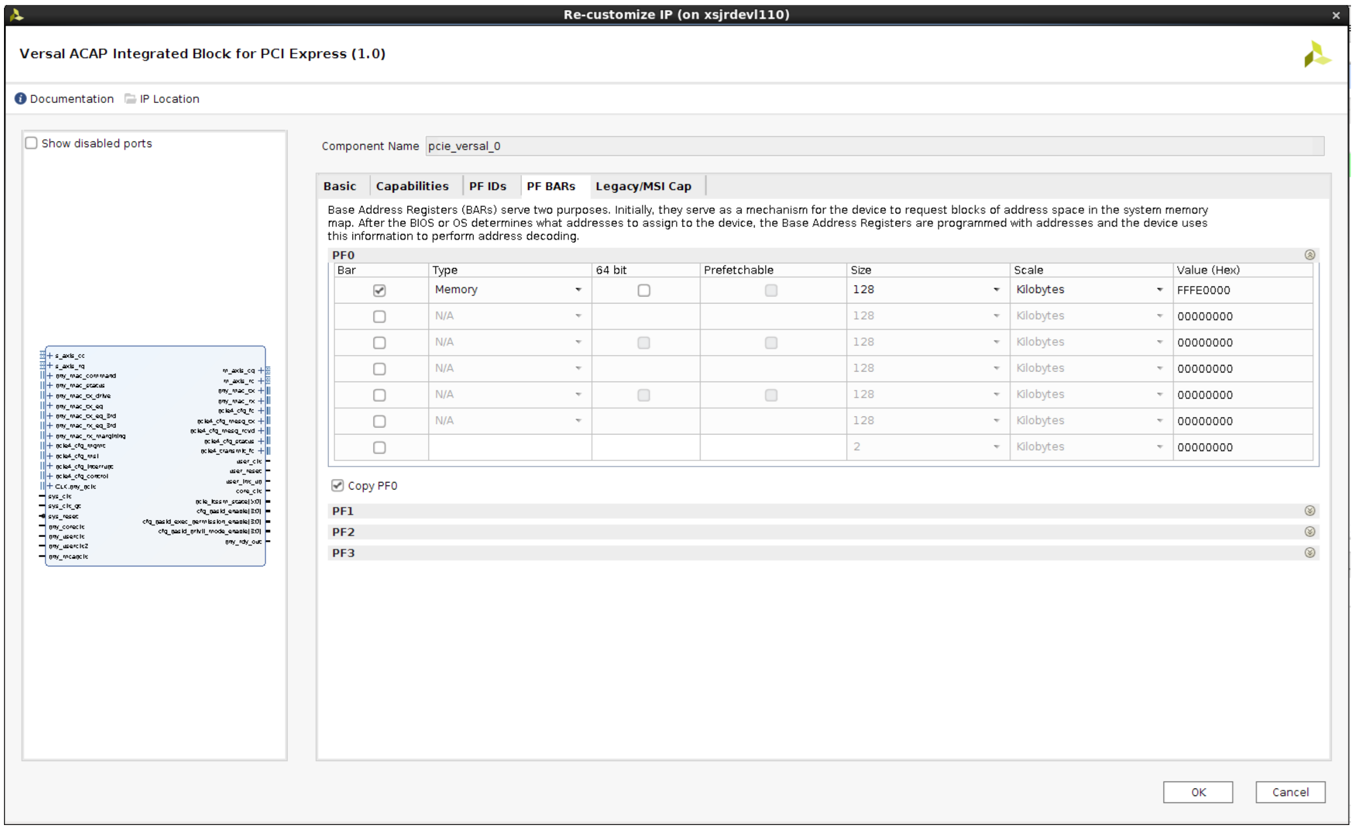
  Class code encoding can be found at the PCI-SIG website.

- **Class Code Look-up Assistant:** The Class Code Look-up Assistant provides the Base Class, Sub-Class and Interface values for a selected general function of a device. This Look-up Assistant tool only displays the three values for a selected function. You must enter the values in Class Code for these values to be translated into device settings.

PF BARs Tab

The PF BARs tab, shown in the following figure, sets the base address register space for the Endpoint configuration. Each BAR (0 through 5) configures the BAR Aperture Size and Control attributes of the physical function.

Figure 106: **PF BARs Tab Showing PF0 and PF1 Only**



Re-customize IP (on xsjrdevl110)

Versal ACAP Integrated Block for PCI Express (1.0)

Documentation   IP Location

Show disabled ports

Component Name pcie_versal_0

Basic | Capabilities | PF IDs | PF BARs | Legacy/MSI Cap

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

PF0

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

✓ Copy PF0

PF1

PF2

PF3

OK   Cancel

- **Base Address Register Overview:** In Endpoint configuration, the core supports up to six 32-bit BARs or three 64-bit BARs, and the Expansion read-only memory (ROM) BAR. In Root Port configuration, the core supports up to two 32-bit BARs or one 64-bit BAR, and the Expansion ROM BAR. BARs can be one of two sizes:

  - **32-bit BARs:** The address space can be as small as 128 bytes or as large as 2 gigabytes. Used for Memory or I/O.

  - **64-bit BARs:** The address space can be as small as 128 bytes or as large as 8 Exabytes. Used for Memory only.

  All BAR registers share these options:

  - **Checkbox:** Click the checkbox to enable the BAR; deselect the checkbox to disable the BAR.

  - **Type:** Bars can either be I/O or Memory.

- **I/O:** I/O BARs can only be 32-bit; the Prefetchable option does not apply to I/O BARs. I/O BARs are only enabled for a Legacy PCI Express Endpoint.
- **Memory:** Memory BARs can be either 64-bit or 32-bit and can be prefetchable. When a BAR is set as 64 bits, it uses the next BAR for the extended address space and makes the next BAR inaccessible.

- **Size:** The available Size range depends on the PCIe Device/Port Type and the Type of BAR selected. The following table lists the available BAR size ranges.

Table 67: **BAR Size Ranges for Device Configuration**

PCIe Device / Port Type	BAR Type	BAR Size Range
PCI Express Endpoint	32-bit Memory	128 bytes (B) – 2 gigabytes (GB)
	64-bit Memory	128 B – 8 Exabytes
Legacy PCI Express Endpoint	32-bit Memory	128 B – 2 GB
	64-bit Memory	128 B – 8 Exabytes
	I/O	16 B – 2 GB

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

- **Expansion ROM Base Address Register:** If selected, the Expansion ROM is activated and can be sized from 2 KB to 4 GB. According to the PCI Local Bus Specification Revision 3.0 on the PCI-SIG website, the maximum size for the Expansion ROM BAR should be no larger than 16 MB. Selecting an address space larger than 16 MB can cause compliance testing issues.

- **Managing Base Address Register Settings:** Memory, I/O, Type, and Prefetchable settings are handled by setting the appropriate settings for the desired base address register.

  Memory or I/O settings indicate whether the address space is defined as memory or I/O. The base address register only responds to commands that access the specified address space. Generally, memory spaces less than 4 KB in size should be avoided. The minimum I/O space allowed is 16 bytes; use of I/O space should be avoided in all new designs.

  Prefetchability is the ability of memory space to be prefetched. A memory space is prefetchable if there are no side effects on reads (that is, data is not destroyed by reading, as from a RAM). Byte-write operations can be merged into a single double word write, when applicable.

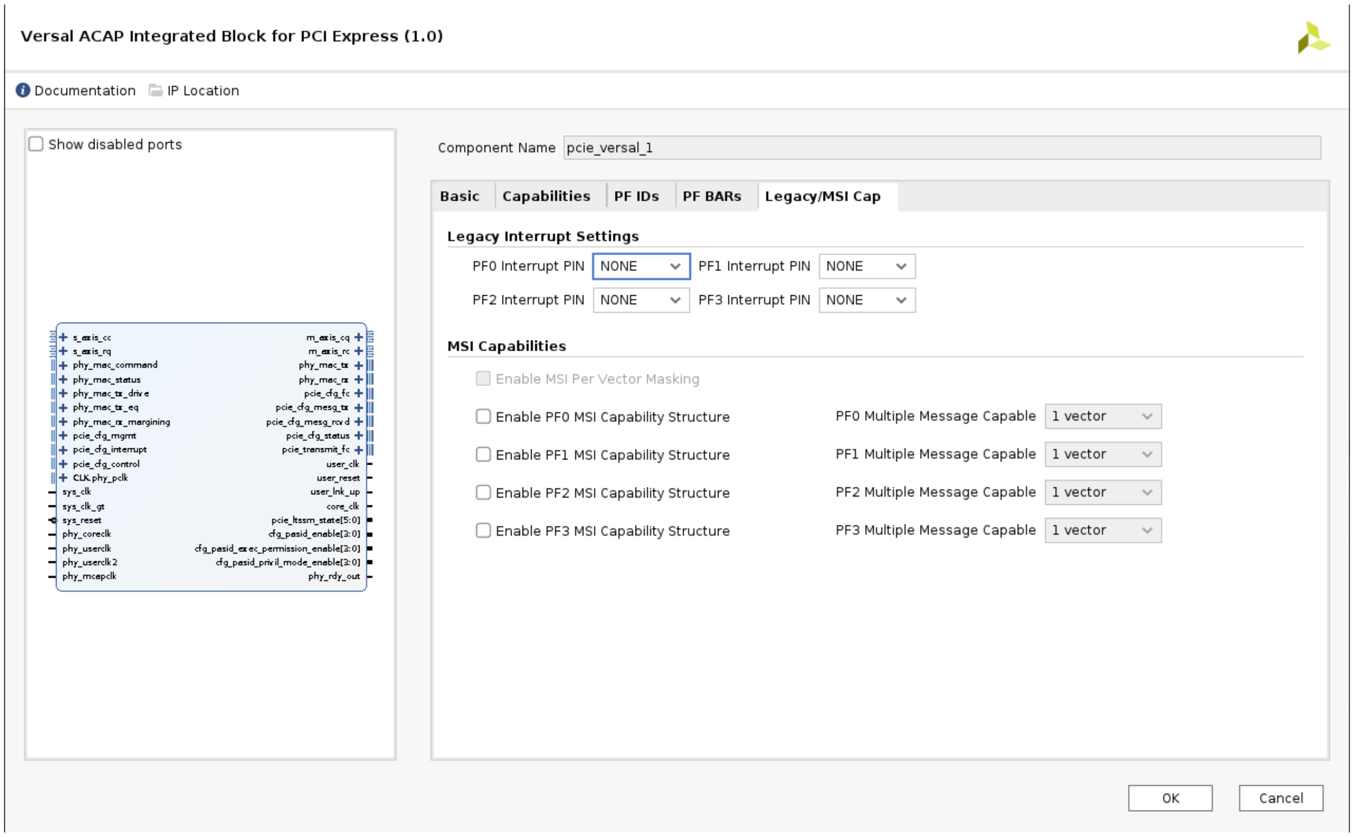
  When configuring the core as an Endpoint for PCIe (non-Legacy), 64-bit addressing must be supported for all BARs (except BAR5) that have the prefetchable bit set. 32-bit addressing is permitted for all BARs that do not have the prefetchable bit set. The prefetchable bit-related requirement does not apply to a Legacy Endpoint. The minimum memory address range supported by a BAR is 128 bytes for a PCI Express Endpoint and 16 bytes for a Legacy PCI Express Endpoint.

- **Disabling Unused Resources:** For best results, disable unused base address registers to conserve system resources. A base address register is disabled by deselecting unused BARs in the Customize IP dialog box.

Legacy/MSI Cap Tab

On this page, you set the Legacy Interrupt Settings and MSI Capabilities for all applicable physical and virtual functions. This page is not visible when the **SRIOV Capability** parameter is selected on the Capabilities page.

Figure 107: **Legacy/MSI Cap Tab**



Component Name: pcie_versal_1

Legacy Interrupt Settings

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

MSI Capabilities

☐ Enable MSI Per Vector Masking

☐ Enable PF0 MSI Capability Structure — PF0 Multiple Message Capable: 1 vector
☐ Enable PF1 MSI Capability Structure — PF1 Multiple Message Capable: 1 vector
☐ Enable PF2 MSI Capability Structure — PF2 Multiple Message Capable: 1 vector
☐ Enable PF3 MSI Capability Structure — PF3 Multiple Message Capable: 1 vector

- **Legacy Interrupt Settings:**
  - **PF0/PF1/PF2/PF3 Interrupt PIN:** Indicates the mapping for Legacy Interrupt messages. A setting of None indicates that no Legacy Interrupts are used.

    ***Note*:** When PASID is enabled, legacy interrupts cannot be used and are disabled.
- **MSI Capabilities:**
  - **PF0/PF1/PF2/PF3 Enable MSI Capability Structure:** Indicates that the MSI Capability structure exists.

Note: Although it is possible to not enable MSI or MSI-X, the result would be a non-compliant core. The PCI Express Base Specification requires that MSI, MSI-X, or both be enabled. No MSI capabilities are supported when **MSI-X Internal** is enabled in the MSI-X Capabilities Tab (Advanced mode), because MSI-X Internal uses some of the MSI interface signals.

- **PF0/PF1/PF2/PF3 Multiple Message Capable:** Selects the number of MSI vectors to request from the Root Complex.
- **Enable MSI Per Vector Masking:** Enables MSI Per Vector Masking Capability of all the Physical functions enabled.

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

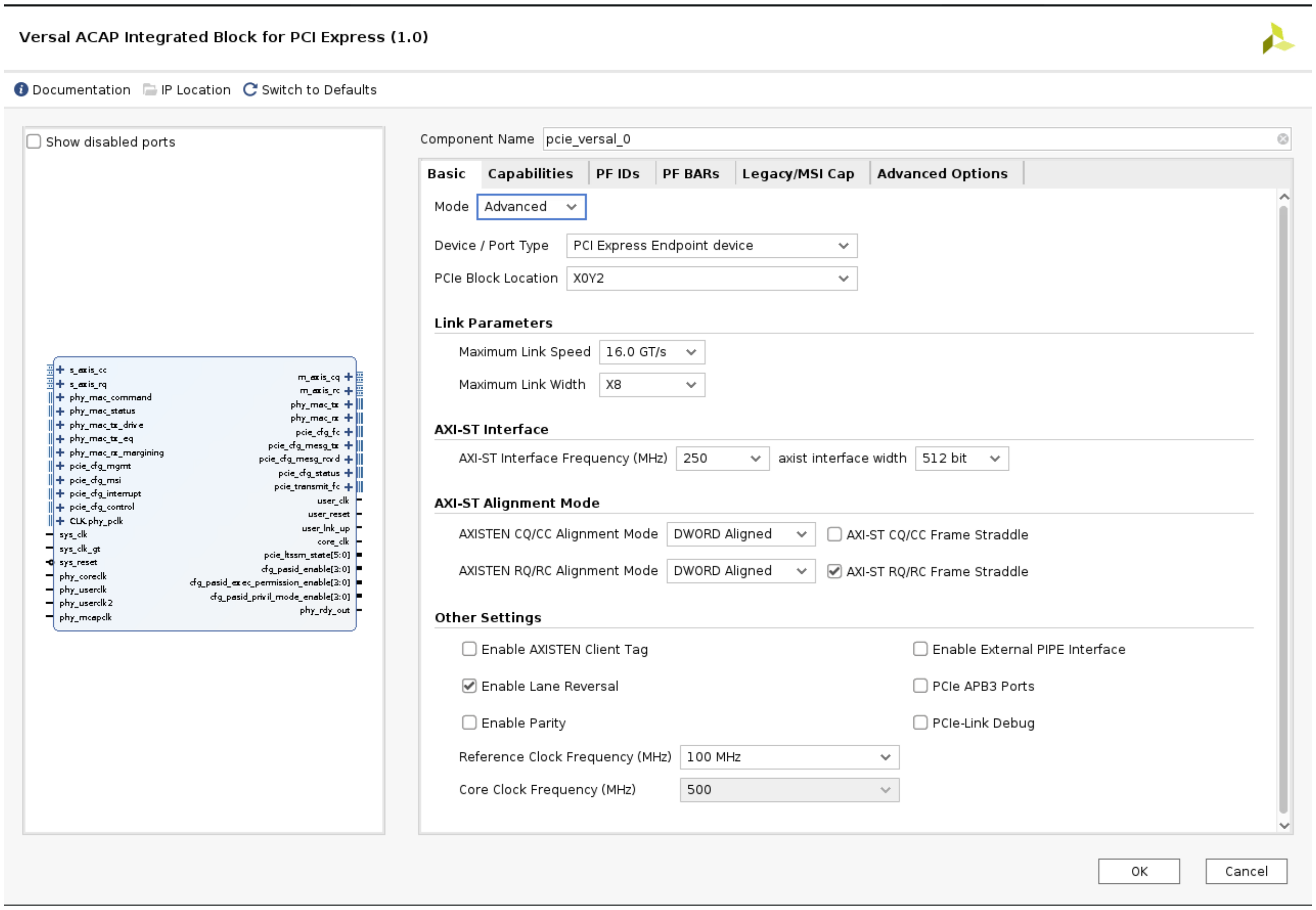
Advanced Mode Parameters

The following parameters appear on different pages of the IP catalog when **Advanced** mode is selected for Mode on the Basic page.

Basic Tab

The Basic page with Advanced mode selected (shown in the following figure) includes additional settings. The following parameters are visible on the Basic page when the **Advanced** mode is selected.

Figure 108: **Basic Tab, Advanced Mode**



Component Name: pcie_versal_0

Mode: Advanced

Device / Port Type: PCI Express Endpoint device

PCIe Block Location: X0Y2

Link Parameters

Maximum Link Speed: 16.0 GT/s

Maximum Link Width: X8

AXI-ST Interface

AXI-ST Interface Frequency (MHz): 250 axist interface width: 512 bit

AXI-ST Alignment Mode

AXISTEN CQ/CC Alignment Mode: DWORD Aligned ☐ AXI-ST CQ/CC Frame Straddle

AXISTEN RQ/RC Alignment Mode: DWORD Aligned ☑ AXI-ST RQ/RC Frame Straddle

Other Settings

☐ Enable AXISTEN Client Tag ☐ Enable External PIPE Interface

☑ Enable Lane Reversal ☐ PCIe APB3 Ports

☐ Enable Parity ☐ PCIe-Link Debug

Reference Clock Frequency (MHz): 100 MHz

Core Clock Frequency (MHz): 500

- **Core Clock Frequency:**

  For Gen1 and Gen2 it is 250 MHz always.

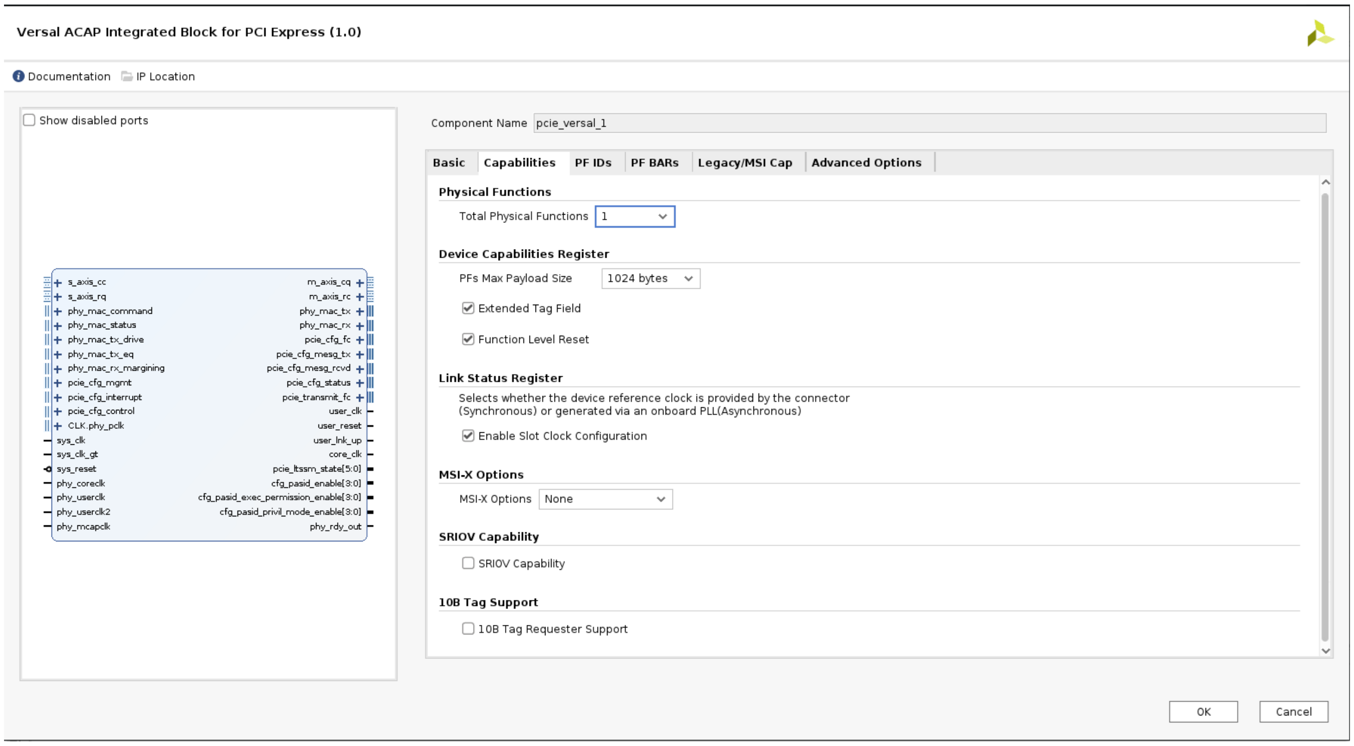
  For Gen3 and Gen4 it is 500 MHz always.

- **Enable Parity:** Enables Parity on TX/RX interfaces including MSI-X.
- **PCIe APB3 Ports:** When checked, enables the PCIe APB3 interface.
- **PCIe Link Debug:** This enables the link debug option to be activated.
- **Enable Lane Reversal:** This enables the lane reversal feature.

Capabilities Tab

The Capabilities settings for Advanced mode (as shown in the following figure) contains two additional parameters to those for Basic mode and are described below.

Figure 109: **Capabilities Tab, Advanced Mode**



Versal ACAP Integrated Block for PCI Express (1.0)

Component Name pcie_versal_1

Physical Functions

Total Physical Functions 1

Device Capabilities Register

PFs Max Payload Size 1024 bytes

[✓] Extended Tag Field

[✓] Function Level Reset

Link Status Register

Selects whether the device reference clock is provided by the connector (Synchronous) or generated via an onboard PLL(Asynchronous)

[✓] Enable Slot Clock Configuration

MSI-X Options

MSI-X Options None

SRIOV Capability

[ ] SRIOV Capability

10B Tag Support

[ ] 10B Tag Requester Support

- **Function Level Reset:** Enable Function Level Reset (FLR). FLR is supported when the PCIe IP is configured as Endpoint.
- **SRIOV Capabilities:** Enables Single Root Port I/O Virtualization (SR-IOV) capabilities. The integrated block implements extended Single Root Port I/O Virtualization PCIe. When this is enabled, SR-IOV is implemented on all the selected physical functions. When SR-IOV capabilities are enabled MSI support is disabled and you can use MSI-X support as shown in the above figure.

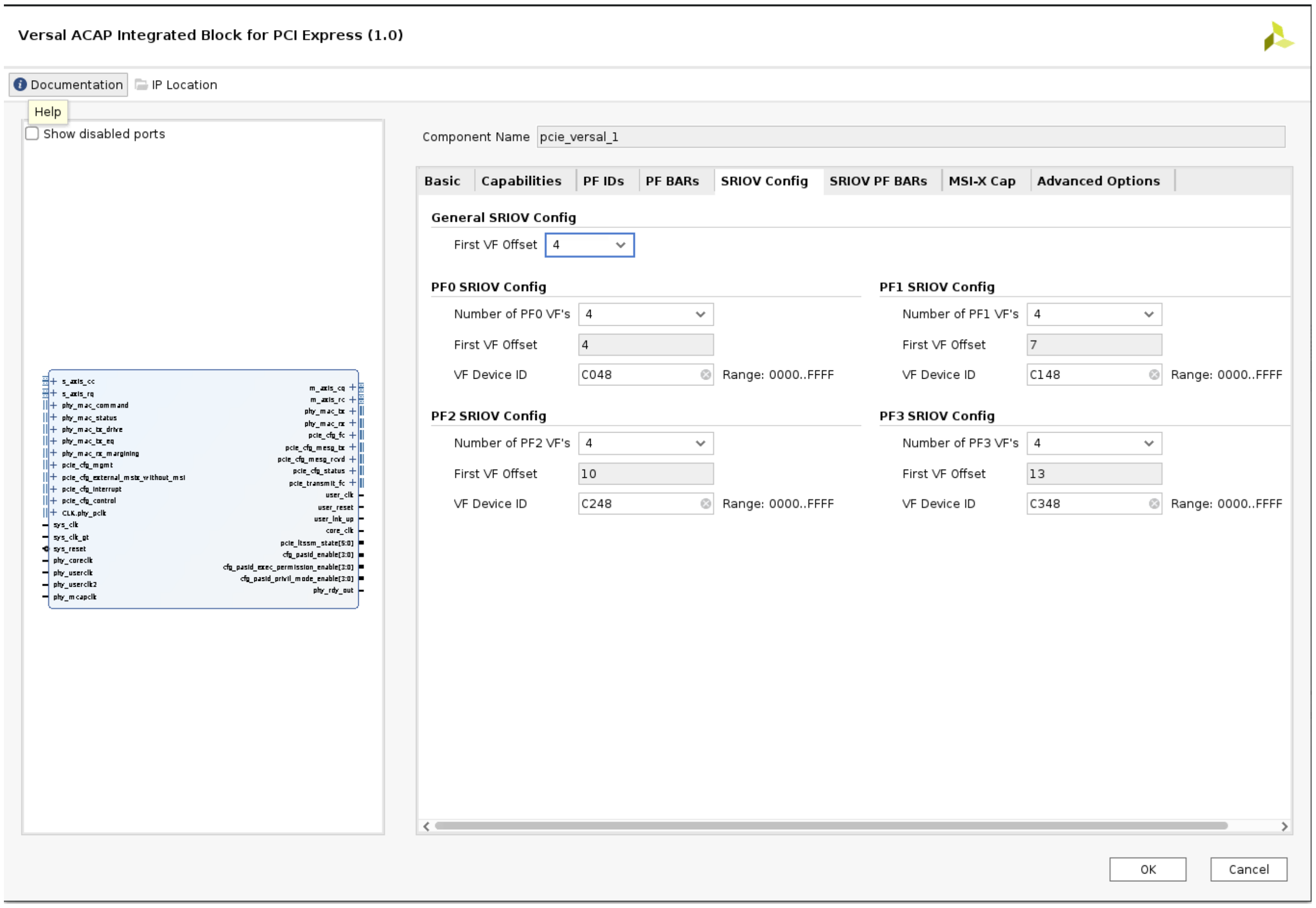
  ***Note*:** When SR-IOV capabilities are enabled, MSI support is disabled and you can use MSI-X support.
- **MSI-X Options:** To enable MSI-X capabilities, select **Advanced** mode and then select the required options on the Capabilities tab. There are four options to choose from:
  - **MSI-X External:** In this mode you need to implement MSI-X External interface driving logic, MSI-X Table and PBA buffers outside the PCIe core. You can configure the MSI-X BARs.
  - **MSI-X Internal:** In this mode you need to implement the MSI-X Internal interface driving logic only. MSI-X Table and PBA buffers are built into the PCIe core. You can configure the MSI-X BARs.
  - **MSI-X AXI4-Stream:** In this mode user is expected to drive MSI-X interrupts on the AXI4-Stream interface. You can configure the MSI-X BARs.
  - **None:** No MSI-X is supported.

The same MSI-X options are applicable when SRIOV capability is selected.

SRIOV Config Tab

The SRIOV Configuration Advanced parameters, as shown in the following figure, are described in this section.

Figure 110: **SRIOV Configuration Tab**



Versal ACAP Integrated Block for PCI Express (1.0)

Component Name pcie_versal_1

General SRIOV Config

First VF Offset 4

PF0 SRIOV Config

Number of PF0 VF's 4
First VF Offset 4
VF Device ID C048 Range: 0000..FFFF

PF1 SRIOV Config

Number of PF1 VF's 4
First VF Offset 7
VF Device ID C148 Range: 0000..FFFF

PF2 SRIOV Config

Number of PF2 VF's 4
First VF Offset 10
VF Device ID C248 Range: 0000..FFFF

PF3 SRIOV Config

Number of PF3 VF's 4
First VF Offset 13
VF Device ID C348 Range: 0000..FFFF

- **General SRIOV Config:** This value specifies the offset of the first PF with at least one enabled VF. The total number of VF in all PFs plus this field must not be greater than 256.
- **Number of PFx VFs:** Indicates the number of virtual functions associated to the physical function. A total of 252 virtual functions are available that can be flexibly used across the four physical functions.
- **First VF Offset:** Indicates the offset of the first virtual function (VF) for the physical function (PF). PFx offset is always fixed. PF0 resides at offset 0, PF1 resides at offset 1, PF2 resides at offset 2, and PF3 resides at offset 3.

A total of 252 virtual functions are available. They reside at the function number range 4 to 255.

First VF offset always start from 4.

Virtual functions are mapped sequentially with VFs with PFs taking precedence. For example, if PF0 has two virtual functions and PF1 has three, the following mapping occurs:

The PFx_FIRST_VF_OFFSET is calculated by taking the first offset of the virtual function and subtracting that from the offset of the physical function.

```
PFx_FIRST_VF_OFFSET = (PFx first VF offset - PFx offset)
```

In the example above, the following offsets are used:

```
PF0_FIRST_VF_OFFSET = (4 - 0) = 4
PF1_FIRST_VF_OFFSET = (6 - 1) = 5
```

The initial offset for PF1 is a function of how many VFs are attached to PF0 and is defined in the following pseudo code:

```
PF1_FIRST_VF_OFFSET = FIRST_VF_OFFSET + NUM_PF0_VFs - 1
```

Similarly, for other PFs:

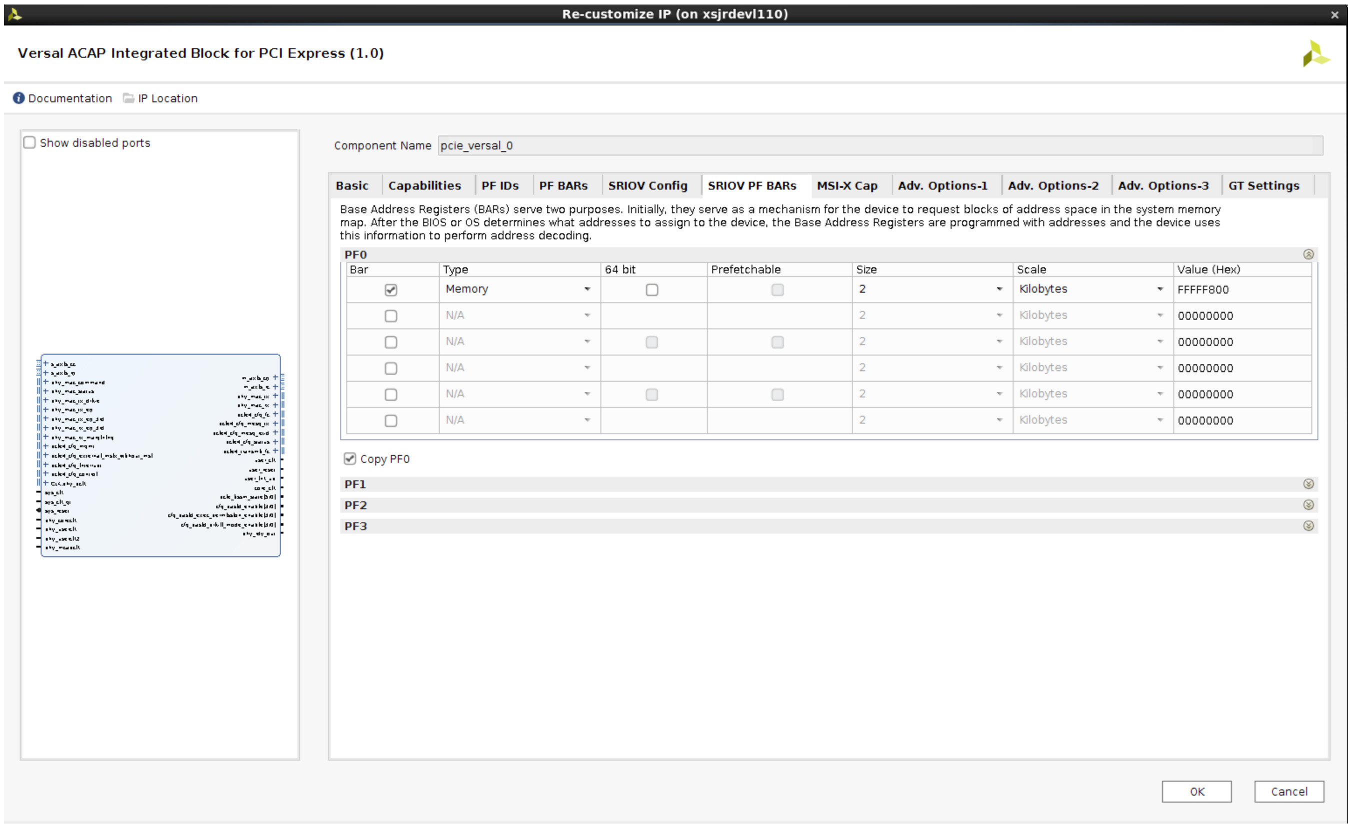
```
PF2_FIRST_VF_OFFSET = FIRST_VF_OFFSET + NUM_PF0_VFs + NUM_PF1_VFs - 2
PF3_FIRST_VF_OFFSET =
        FIRST_VF_OFFSET + NUM_PF0_VFs + NUM_PF1_VFs + NUM_PF2_VFs - 3
```

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

SRIOV PF BARs Tab

The SRIOV Base Address Registers (BARs) set the base address register space for the Endpoint configuration. Each BAR (0 through 5) configures the SR-IOV BAR aperture size and SR-IOV control attributes.

Figure 111: **SRIOV BARs Tab, Advanced Mode**



Re-customize IP (on xsjrdevl110)

Versal ACAP Integrated Block for PCI Express (1.0)

Documentation   IP Location

Show disabled ports

Component Name: pcie_versal_0

Tabs: Basic | Capabilities | PF IDs | PF BARs | SRIOV Config | SRIOV PF BARs | MSI-X Cap | Adv. Options-1 | Adv. Options-2 | Adv. Options-3 | GT Settings

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

PF0

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

✓ Copy PF0

PF1

PF2

PF3

OK   Cancel

Table 68: **Example Virtual Function Mappings**

Physical Function	Virtual Function	Function Number Range
PF0	VF0	64
PF0	VF1	65
PF1	VF0	68
PF1	VF1	69
PF1	VF1	70

- **SRIOV Base Address Register Overview:** In Endpoint configuration, the core supports up to six 32-bit BARs or three 64-bit BARs. In Root Port configuration, the core supports up to two 32-bit BARs or one 64-bit BAR. SR-IOV BARs can be one of two sizes:
  - **32-bit BARs:** The address space can be as small as 16 bytes or as large as 3 gigabytes. Used for memory to I/O.
  - **64-bit BARs:** The address space can be as small as 128 bytes or as large as 256 gigabytes. Used for memory only.

  All SR-IOV BAR registers have these options:

- **Checkbox:** Click the checkbox to enable the BAR; deselect the checkbox to disable the BAR.
- **Type:** SR-IOV BARs can be either I/O or Memory.
  - **I/O:** I/O BARs can only be 32-bit; the Prefetchable option does not apply to I/O BARs. I/O BARs are only enabled for a Legacy PCI Express Endpoint.
  - **Memory:** Memory BARs can be either 64-bit or 32-bit and can be prefetchable. When a BAR is set to 64-bits, it uses the next BAR for the extended address space and makes the next BAR inaccessible.
- **Size:** The available size range depends on the PCIe device/port type and the type of BAR selected. The following table lists the available BAR size ranges.

Table 69: **SRIOV BAR Size Ranges for Device Configuration**

PCIe Device / Port Type	BAR Type	BAR Size Range
PCI Express Endpoint	32-bit Memory	128 bytes – 2 gigabytes
	64-bit Memory	128 bytes – 8 exabytes
Legacy PCI Express Endpoint	32-bit Memory	16 bytes – 2 gigabytes
	64-bit Memory	16 bytes – 8 exabytes
	I/O	16 bytes – 2 gigabytes

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

- **Managing SRIOV Base Address Register Settings:** Memory, I/O, Type, and Prefetchable settings are handled by setting the appropriate Customize IP dialog box settings for the desired base address register.

  Memory or I/O settings indicate whether the address space is defined as memory or I/O. The base address register only responds to commands that access the specified address space. Generally, memory spaces less than 4 KB in size should be avoided. The minimum I/O space allowed is 16 bytes. I/O space should be avoided in all new designs.

  A memory space is prefetchable if there are no side effects on reads (that is, data is not destroyed by reading, as from RAM). Byte-write operations can be merged into a single double-word write, when applicable.

  When configuring the core as an Endpoint for PCIe (non-Legacy), 64-bit addressing must be supported for all SR-IOV BARs (except BAR5) that have the prefetchable bit set. 32-bit addressing is permitted for all SR-IOV BARs that do not have the prefetchable bit set. The prefetchable bit related requirement does not apply to a Legacy Endpoint. The minimum memory address range supported by a BAR is 128 bytes for a PCI Express Endpoint and 16 bytes for a Legacy PCI Express Endpoint.

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

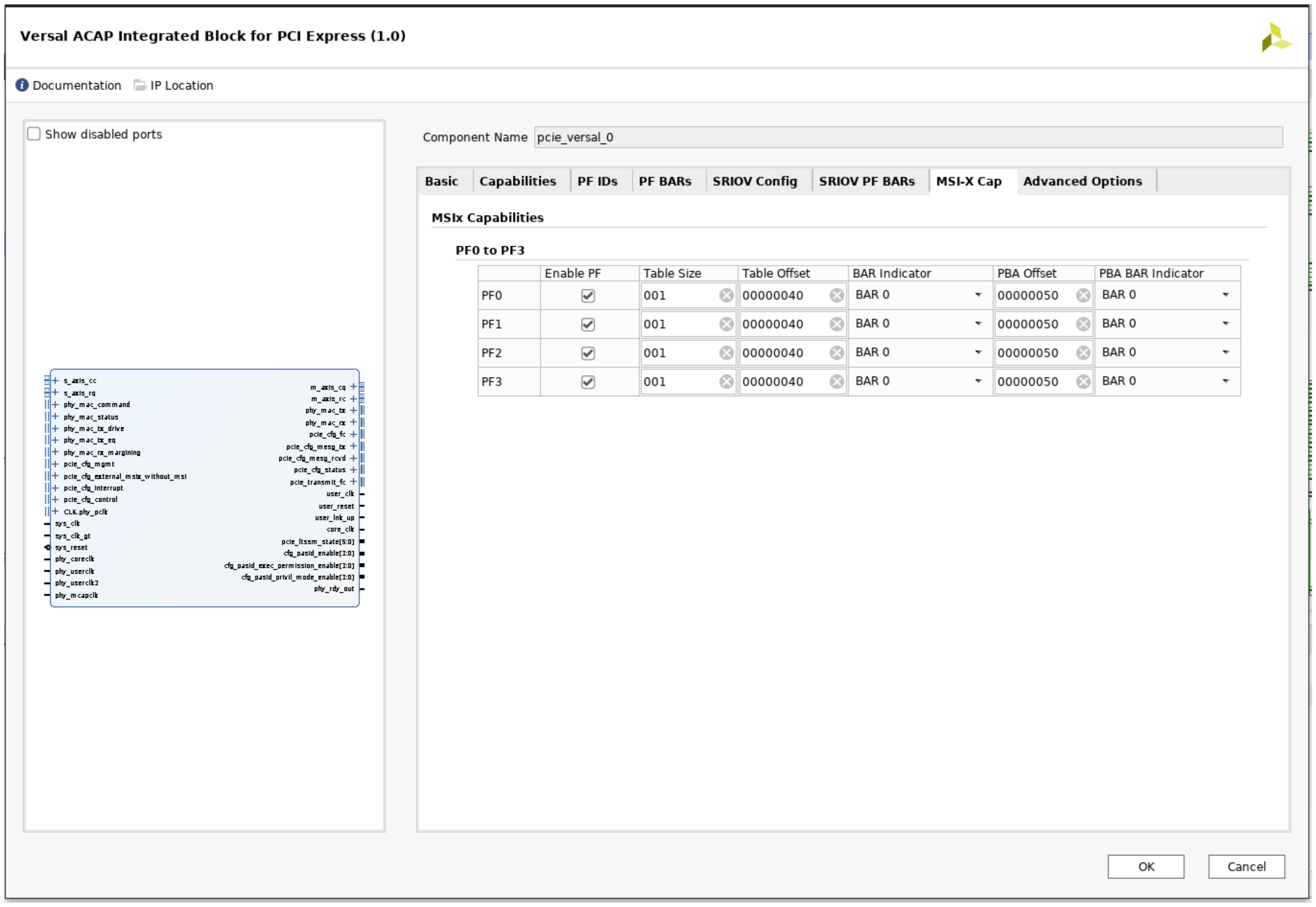
MSI-X Capabilities Tab

The MSI-X Capabilities parameters, shown in the following figure, are available in Advanced mode only. To enable MSI-X capabilities, select **Advanced** mode and then select the required options on the Capabilities page. There are four options to choose from.

- **MSI-X External:** In this mode you need to implement MSI-X External interface driving logic, MSI-X Table and PBA buffers outside the core. You can configure the MSI-X BARs.
- **MSI-X Internal:** In this mode you need to implement the MSI-X Internal interface driving logic only. MSI-X Table and PBA buffers are built into the core. You can configure the MSI-X BARs.
- **MSI-X AXI4-Stream:** In this mode user is expected to drive MSI-X interrupts on the AXI4-Stream interface. You can configure the MSI-X BARs.
- **None:** No MSI-X is supported.

The same MSI-X options are applicable when SRIOV capability is selected.

Figure 112: **MSI-X Cap Tab, Advanced Mode**



Versal ACAP Integrated Block for PCI Express (1.0)

Component Name: pcie_versal_0

MSIx Capabilities

PF0 to PF3

	Enable PF	Table Size	Table Offset	BAR Indicator	PBA Offset	PBA BAR Indicator
PF0	✓	001	00000040	BAR 0	00000050	BAR 0
PF1	✓	001	00000040	BAR 0	00000050	BAR 0
PF2	✓	001	00000040	BAR 0	00000050	BAR 0
PF3	✓	001	00000040	BAR 0	00000050	BAR 0

- **Enable MSI-X Capability Structure:** Indicates that the MSI-X Capabilities structure exists.

  ***Note*:** The Capability Structure needs at least one Memory BAR to be configured. You must maintain the MSI-X Table and Pending Bit Array in the application.

- **MSI-X Table Settings:** Defines the MSI-X Table structure.

  - **Table Size:** Specifies the MSI-X Table size. Table Size field is expecting N-1 interrupts (0x0F will configure a table count of 16).

  - **Table Offset:** Specifies the offset from the Base Address Register that points to the base of the MSI-X Table.

  - **BAR Indicator:** Indicates the Base Address Register in the Configuration Space used to map the function in the MSI-X Table onto memory space. For a 64-bit Base Address Register, this indicates the lower DWORD.

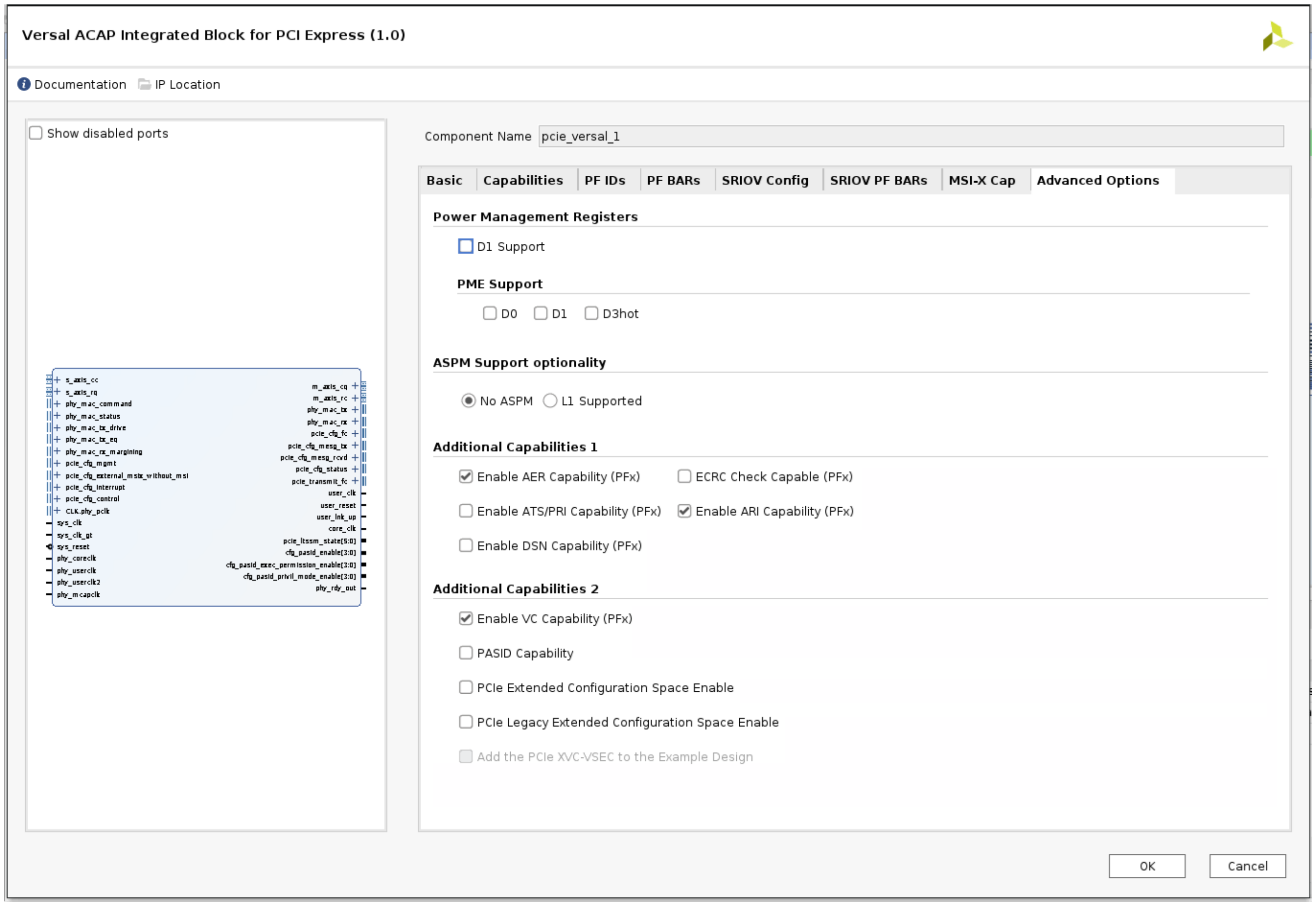
- **MSIx Pending Bit Array (PBA) Settings:** Defines the MSI-X Pending Bit Array (PBA) structure.

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

Related Information

Clocking

Advanced Options Tab



Versal ACAP Integrated Block for PCI Express (1.0)

☐ Show disabled ports

Component Name pcie_versal_1

Power Management Registers

☐ D1 Support

PME Support

☐ D0 ☐ D1 ☐ D3hot

ASPM Support optionality

[✓ No ASPM] L1 Supported

Additional Capabilities 1

☑ Enable AER Capability (PFx) ☐ ECRC Check Capable (PFx)

☐ Enable ATS/PRI Capability (PFx) ☑ Enable ARI Capability (PFx)

☐ Enable DSN Capability (PFx)

Additional Capabilities 2

☑ Enable VC Capability (PFx)

☐ PASID Capability

☐ PCIe Extended Configuration Space Enable

☐ PCIe Legacy Extended Configuration Space Enable

☐ Add the PCIe XVC-VSEC to the Example Design

- **Power Management and ASPM Support:** The section allows you to enable Power Management Registers and ASPM. L0s is only supported when the link speed is 2.5 Gb/s and 5.0 Gb/s. L1 is not supported in Root Port configuration.
- **Additional Capabilities 1:** The section enables you to choose AER, ECRC, ATS,/PRI , ARI, and DSN capabilities for the core.

- **Additional Capabilities 2:** The section enables you to choose VC, PASID, and user-defined capabilities (PCIe Extended Configuration Space and PCIe Legacy Extended Configuration Space) for the core.

Output Generation

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

Constraining the Core

Required Constraints

The Versal ACAP Integrated Block for PCI Express® solution requires the specification of timing and other physical implementation constraints to meet specified performance requirements for PCI Express®. These constraints are provided with the Endpoint and Root Port solutions in a Xilinx Design Constraints (XDC) file. Pinouts and hierarchy names in the generated XDC correspond to the provided example design.

IMPORTANT! *If the example design top file is not used, copy the IBUFDS instance for the reference clock, IBUF Instance for sys_rst and also the location and timing constraints associated with them into your local design top. In addition, the GT location constraints need to be added in top-level xdc. For more information look at GT Locations in GT Selection and Pin Planning.*

To achieve consistent implementation results, an XDC containing these original, unmodified constraints must be used when a design is run through the Xilinx tools. For additional details on the definition and use of an XDC or specific constraints, see *Vivado Design Suite User Guide: Using Constraints* (UG903).

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

Clock Frequencies

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

Clock Management

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

Clock Placement

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

Banking

This section is not applicable for this IP core.

Transceiver Placement

This section is not applicable for this IP core.

I/O Standard and Placement

This section is not applicable for this IP core.

Relocating the Integrated Block Core

By default, the IP core-level constraints lock block RAMs, UltraRAMs, transceivers, and the integrated block to the recommended location. To relocate these blocks, you must override the constraints for these blocks in the XDC constraint file. To do so:

1. Copy the constraints for the block that needs to be overwritten from the core-level XDC constraint file.
2. Place the constraints in the user XDC constraint file.
3. Update the constraints with the new location.

The user XDC constraints are usually scoped to the top-level of the design; therefore, ensure that the cells referred by the constraints are still valid after copying and pasting them. Typically, you need to update the module path with the full hierarchy name.

***Note*:** If there are locations that need to be swapped (that is, the new location is currently being occupied by another module), there are two ways to do this.

- If there is a temporary location available, move the first module out of the way to a new temporary location first. Then, move the second module to the location that was occupied by the first module. Next, move the first module to the location of the second module. These steps can be done in XDC constraint file.
- If there is no other location available to be used as a temporary location, use the `reset_property` command from Tcl command window on the first module before relocating the second module to this location. The `reset_property` command cannot be done in XDC constraint file and must be called from the Tcl command file or typed directly into the Tcl Console.

Simulation

For comprehensive information about Vivado® simulation components, as well as information about using supported third-party tools, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900).

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

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 Vivado® Design Suite.

Overview of the Example Design

This section provides an overview of the Versal ACAP Integrated Block for PCI Express® core example design.

Integrated Block Endpoint Configuration Overview

This IP can support two example designs in Endpoint configuration. One is Programmed Input/Output (PIO) example design and other one is Bus Master DMA (BMD) example design.

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

- The Root Port Model, a test bench that generates, consumes, and checks PCI Express® bus traffic.
- The Programmed Input/Output (PIO) example design, a completer application for PCI Express. The PIO example design responds to Read and Write requests to its memory space and can be synthesized for testing in hardware.

***Note*:** Not all modes have example design support, for example, Straddle, Address aligned mode, SRIOV, MSI-X, and MSI.

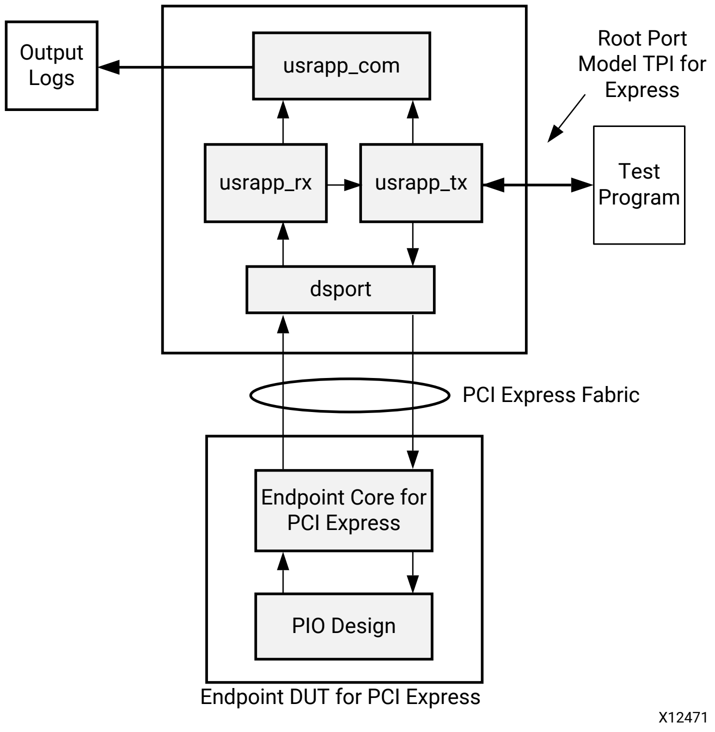
***Note*:** Currently, the BMD design is the default example design. For details about the BMD design, see the *Bus Master Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions* (XAPP1052). To use the PIO design, enter the command below at the Vivado Tcl Console prompt after the Versal ACAP Integrated Block for PCIe® IP is generated.

```
CONFIG.bmd_pio_mode {false}
```

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.

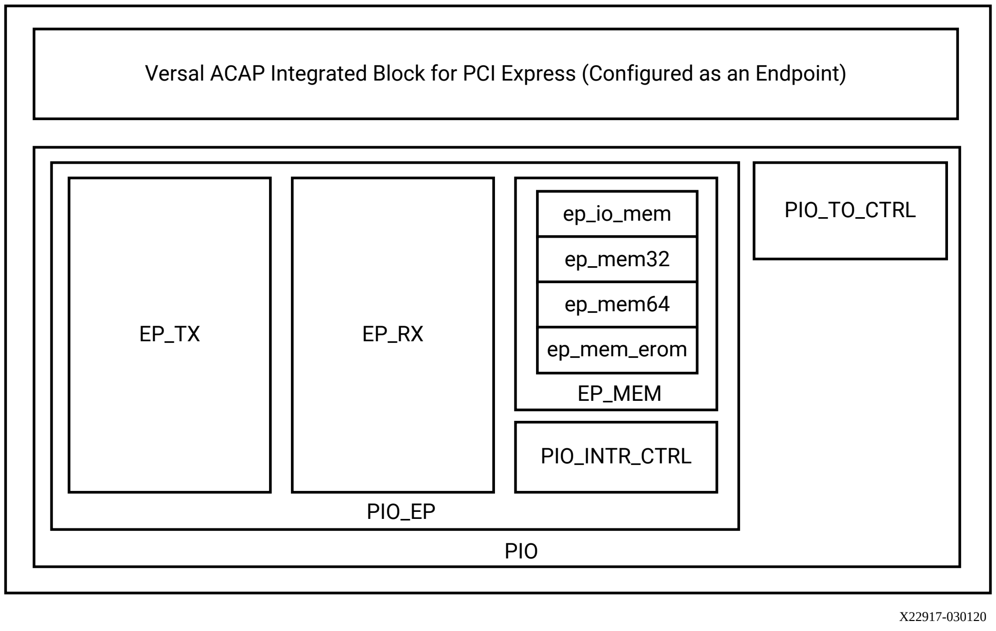
Figure 113: **Simulation Example Design Block Diagram**



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

Figure 114: **Implementation Example Design Block Diagram**



Example Design Elements

The PIO example design elements include:

- Core wrapper
- An example Verilog HDL wrapper (instantiates the cores and example design)
- A customizable demonstration test bench to simulate the example design

The example design has been tested and verified with Vivado Design Suite and these simulators:

- Vivado simulator
- Questa Advanced Simulator
- Synopsys Verilog Compiler Simulator (VCS)

For the supported versions of these tools, see the Xilinx Design Tools: Release Notes Guide.

Programmed Input/Output: Endpoint Example Design

Programmed Input/Output (PIO) transactions are generally used by a PCI Express® system host CPU to access Memory Mapped Input/Output (MMIO) and Configuration Mapped Input/Output (CMIO) locations in the PCI Express logic. Endpoints for PCI Express accept Memory and I/O Write transactions and respond to Memory and I/O Read transactions with Completion with Data transactions.

The PIO example design (PIO design) is included with the core in Endpoint configuration generated by the Vivado® IP catalog, which allows you to bring up your system board with a known established working design to verify the link and functionality of the board.

This section generically represents all solutions using the name Endpoint for PCI Express (or Endpoint for PCIe®).

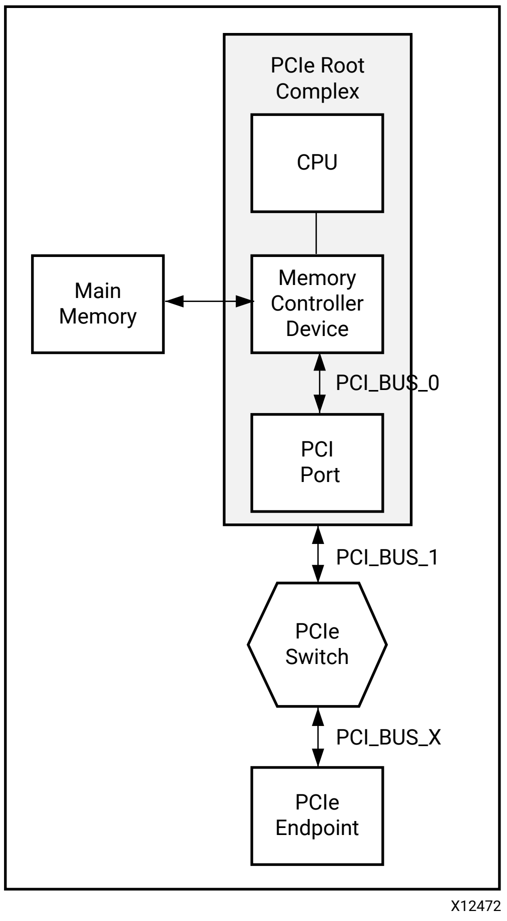
System Overview

The PIO design is a simple target-only application that interfaces with the Endpoint for the PCIe core Transaction (AXI4-Stream) interface and is provided as a starting point for you to build your own designs. These features are included:

- In Address Align Mode, four transaction-specific 2 KB target regions using the internal ACAP block RAMs, providing a total target space of 8,192 bytes.
- In Address Align Mode, supports single Dword payload Read and Write PCI Express transactions to 32-/64-bit address memory spaces and I/O space with support for completion TLPs.
- In the case of Dword Align Mode, the PIO Design supports multiple Dword payload (Up to 256 DW) read and write PCI Express transactions to 32-bit Address Memory Spaces with support for completion TLPs. Utilizes the BAR ID[2:0] and Completer Request Descriptor[114:112] of the core to differentiate between TLP destination Base Address Registers.
- Provides separate implementations optimized for 64-bit, 128-bit, 256-bit, and 512-bit AXI4-Stream interfaces.

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

Figure 115: **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

For Address Align Mode, the PIO design implements an 8,192 byte target space in ACAP block RAM, behind the Endpoint for PCIe. This 32-bit target space is accessible through single Dword I/O Read, I/O Write, Memory Read 64, Memory Write 64, Memory Read 32, and Memory Write 32 TLPs.

The PIO design generates a completion with one Dword of payload in response to a valid Memory Read 32 TLP, Memory Read 64 TLP, or I/O Read TLP request presented to it by the core. In addition, the PIO design returns a completion without data with successful status for I/O Write TLP request. For Dword Align Mode, the PIO design implements 2048-byte target space in ACAP block RAM. This target space, and data width varies based on the AXI4-Stream interface and is equal to the width of the AXI4-Stream interface. This target space is accessible through Memory Write 32 and Memory Read 32 TLPs.

The PIO generates a completion with the payload size in response to a valid Memory Read 32 TLP request from the core.

The PIO design can initiate the following:

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

The PIO design processes a Memory or I/O Write TLP with one Dword payload in case of address align mode and multi-Dword in case of Dword Align Mode by updating the payload into the target address in the ACAP block RAM space.

Base Address Register Support

In case of Address Align Mode, the PIO design supports four discrete target spaces, each consisting of a 2 KB block of memory represented by a separate Base Address Register (BAR). Using the default parameters, the Vivado® IP catalog produces a core configured to work with the PIO design defined in this section, consisting of:

- One 64-bit addressable Memory Space BAR
- One 32-bit Addressable Memory Space BAR

You can change the default parameters used by the PIO design; however, in some cases you might need to change the user application depending on your system. See Changing IP Catalog Tool Default BAR Settings for information about changing the default Vivado Design Suite IP parameters and the effect on the PIO design.

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

Table 70: **TLP Traffic Types**

Block RAM	TLP Transaction Type	Default BAR	BAR ID[2:0]
ep_io_mem	I/O TLP transactions	Disabled	Disabled
ep_mem32	32-bit address Memory TLP transactions	2	000b
ep_mem64	64-bit address Memory TLP transactions	0-1	001b
ep_mem_erom	32-bit address Memory TLP transactions destined for EROM	Expansion ROM	110b

For Dword Align Mode, the PIO design supports one target space, consisting of 2048 Bytes of memory. The memory is implemented using SDRAM. As transaction are received by the core, the core presents the TLP to the PIO design and asserts the bits of BAR ID[2:0],and completer request descriptor [114:112] as `001b`.

Changing IP Catalog Tool Default BAR Settings

You can change the Vivado® IP catalog parameters and continue to use the PIO design to create customized Verilog source to match the selected BAR settings. However, because the PIO design parameters are more limited than the core parameters, consider the following example design limitations when changing the default IP catalog parameters:

- The example design supports one I/O space BAR, one 32-bit Memory space (that cannot be the Expansion ROM space), and one 64-bit Memory space. If these limits are exceeded, only the first space of a given type is active—accesses to the other spaces do not result in completions.
- Each space is implemented with a 2 KB memory. If the corresponding BAR is configured to a wider aperture, accesses beyond the 2 KB limit wrap around and overlap the 2 KB memory space.
- The PIO design supports one I/O space BAR, which by default is disabled, but can be changed if desired.

Although there are limitations to the PIO design, Verilog source code is provided so you can tailor the example design to your specific needs.

TLP Data Flow

This section defines the data flow of a TLP successfully processed by the PIO design.

The PIO design successfully processes single Dword payload Memory Read and Write TLPs for Address Align Mode and multi-Dword payload in case of Dword Align Mode and I/O Read and Write TLPs supported only for Address Align Mode. Memory Read or Memory Write TLPs of lengths larger than one Dword are not processed correctly by the PIO design. In case of Address Align Mode, however, the core does accept these TLPs and passes them along to the PIO design. If the PIO design receives a TLP with a length of greater than one Dword, the TLP is received completely from the core and discarded. No corresponding completion is generated. For Dword Align Mode, payload containing multiple Dword for Memory Read and Memory Write TLPs are supported and are processed correctly by the PIO design. The TLP is received completely from the core and then corresponding completion is generated.

Memory and I/O Write TLP Processing

When the Endpoint for PCIe® receives a Memory or I/O Write TLP, the TLP destination address and transaction type are compared with the values in the core BARs. If the TLP passes this comparison check, the core passes the TLP to the Receive AXI4-Stream interface of the PIO design. The PIO design handles Memory writes and I/O TLP writes in different ways: the PIO design responds to I/O writes by generating a Completion Without Data (cpl), a requirement of the PCI Express specification.

Along with the start of packet, end of packet, and ready handshaking signals, the Completer Requester AXI4-Stream interface also asserts the appropriate (BAR ID[2:0]), Completer Request Descriptor[114:112] signal to indicate to the PIO design the specific destination BAR that matched the incoming TLP. On reception, the PIO design RX State Machine processes the incoming Write TLP and extracts the TLPs data and relevant address fields so that it can pass this along to the PIO design internal block RAM write request controller.

In case of Address align mode, based on the specific BAR ID[2:0] signals asserted, the RX state machine indicates to the internal write controller the appropriate 2 KB block RAM to use prior to asserting the write enable request. For example, if an I/O Write Request is received by the core targeting BAR0, the core passes the TLP to the PIO design and sets BAR ID[2:0] to 000b. The RX state machine extracts the lower address bits and the data field from the I/O Write TLP and instructs the internal Memory Write controller to begin a write to the block RAM.

While in case of Dword Align mode, when the BAR ID[2:0] = 01b, the RX state machine asserts the write enable request. The RX state machine extracts the lower address bits and the data from the Memory 32 Write TLP and instructs the internal memory write controller to begin a write to the block RAM.

In this example, the assertion of setting BAR ID[2:0] to 000b instructed the PIO memory write controller to access `ep_mem0` (which by default represents 2 KB of I/O space). While the write is being carried out to the ACAP block RAM, the PIO design RX state machine deasserts `m_axis_cq_tready`, causing the Receive AXI4-Stream interface to stall receiving any further TLPs until the internal Memory Write controller completes the write to the block RAM. Deasserting `m_axis_cq_tready` in this way is not required for all designs using the core; the PIO design uses this method to simplify the control logic of the RX state machine.

Memory and I/O Read TLP Processing

When the Endpoint for PCIe® receives a Memory or I/O Read TLP, the TLP destination address and transaction type are compared with the values programmed in the core BARs. If the TLP passes this comparison check, the core passes the TLP to the Receive AXI4-Stream interface of the PIO design.

Along with the start of packet, end of packet, and ready handshaking signals, the Completer Requester AXI4-Stream interface also asserts the appropriate BAR ID[2:0] signal to indicate to the PIO design the specific destination BAR that matched the incoming TLP. On reception, the PIO design state machine processes the incoming Read TLP and extracts the relevant TLP information and passes it along to the internal block RAM read request controller of the PIO design.

In case of Address Align Mode, based on the specific BAR ID[2:0] signal asserted, the RX state machine indicates to the internal read request controller the appropriate 2 KB block RAM to use before asserting the read enable request. While for Dword Align Mode, the RX state machine checks if the request is for Memory Read 32 TLP based on the BAR ID [2:0] to enable the read request and discard all the other request. For example, if a Memory Read 32 Request TLP is received by the core targeting the default Mem32 BAR2, the core passes the TLP to the PIO design and sets BAR ID[2:0] to 010b. The RX state machine extracts the lower address bits from the Memory 32 Read TLP and instructs the internal Memory Read Request controller to start a read operation.

In this example, the setting BAR ID[2:0] to 010b instructs the PIO memory read controller to access the Mem32 space, which by default represents 2 KB of memory space. A notable difference in handling of memory write and read TLPs is the requirement of the receiving device to return a Completion with Data TLP in the case of memory or I/O read request.

While the read is being processed, the PIO design RX state machine deasserts `m_axis_cq_tready`, causing the Receive AXI4-Stream interface to stall receiving any further TLPs until the internal Memory Read controller completes the read access from the block RAM and generates the completion. Deasserting `m_axis_cq_tready` in this way is not required for all designs using the core. The PIO design uses this method to simplify the control logic of the RX state machine.

PIO File Structure

The table below defines the PIO design file structure. Based on the specific core targeted, not all files delivered by the Vivado® IP catalog are necessary, and some files might not be delivered. The major difference is that some of the Endpoint for PCIe® solutions use a 32-bit user datapath, others use a 64-bit datapath, and the PIO design works with both. The width of the datapath depends on the specific core being targeted.

Table 71: **PIO Design File Structure**

File	Description
PIO.v	Top-level design wrapper
PIO_INTR_CTRL.v	PIO interrupt controller
PIO_EP.v	PIO application module
PIO_TO_CTRL.v	PIO turn-off controller module
PIO_RX_ENGINE.v	32-bit Receive engine
PIO_TX_ENGINE.v	32-bit Transmit engine
PIO_EP_MEM_ACCESS.v	Endpoint memory access module
PIO_EP_MEM.v	Endpoint memory
PIO_EP_XPM_SDRAM_WRAP.v	Endpoint Memory in case of dword align mode

Four configurations of the PIO design are provided: PIO_64, PIO_128, and PIO_256 with 64-, 128-, 256-bit, and 512-bit AXI4-Stream interfaces, respectively. The PIO configuration that is generated depends on the selected Endpoint type, the number of PCI Express lanes, and the interface width selected. The following table identifies the PIO configuration generated based on your selection.

Table 72: **PIO Configuration**

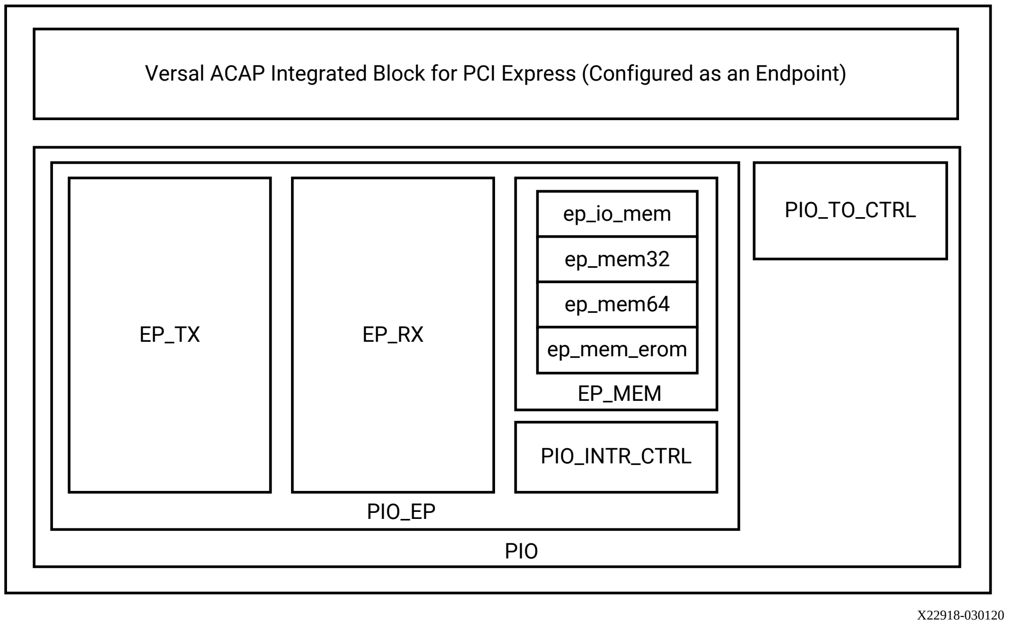
Core	x1	x2	x4	x8
Integrated Block for PCIe	PIO_64	PIO_64, PIO_128	PIO_64, PIO_128, PIO_256	PIO_64, PIO_128[1], PIO_256

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 116: **PIO Design Components**

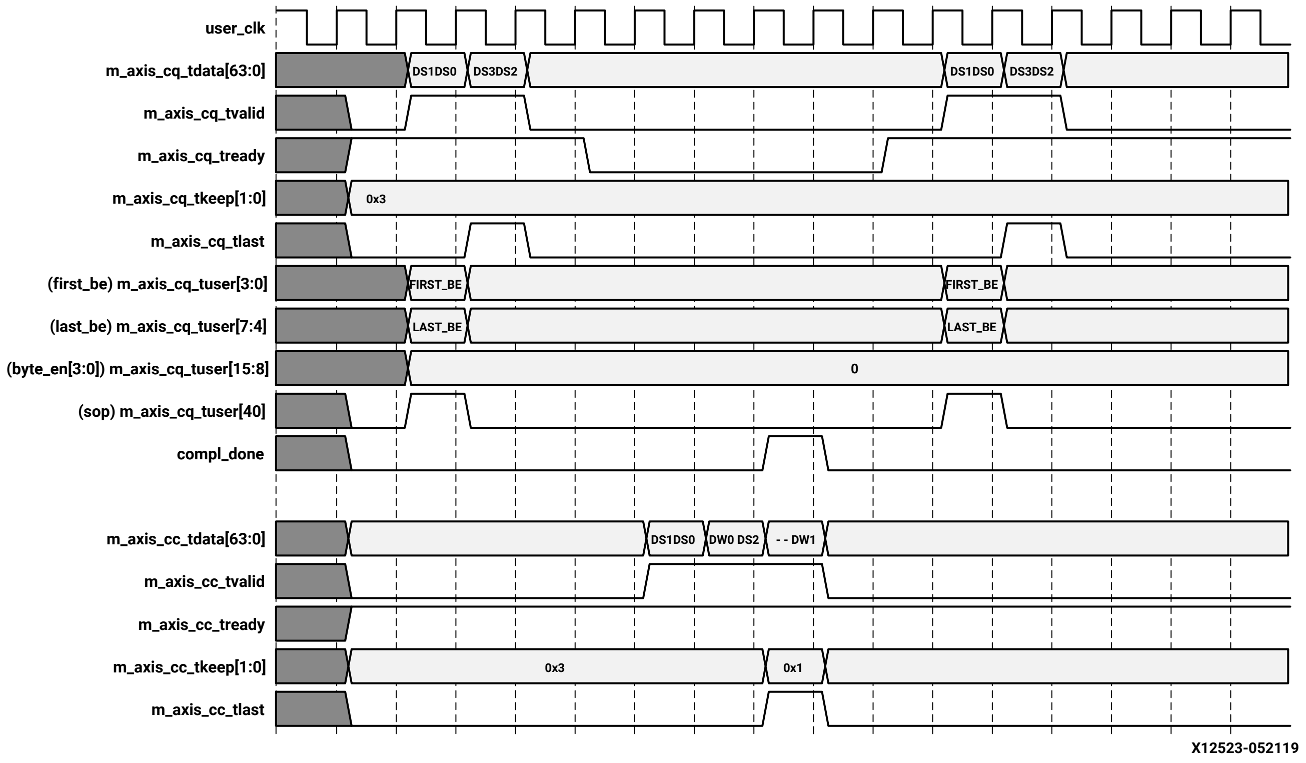


PIO Operation

PIO Read Transaction

The figure below depicts a Back-to-Back Memory Read request to the PIO design. The receive engine deasserts `m_axis_rx_tready` as soon as the first TLP is completely received. The next Read transaction is accepted only after `compl_done_o` is asserted by the transmit engine, indicating that Completion for the first request was successfully transmitted.

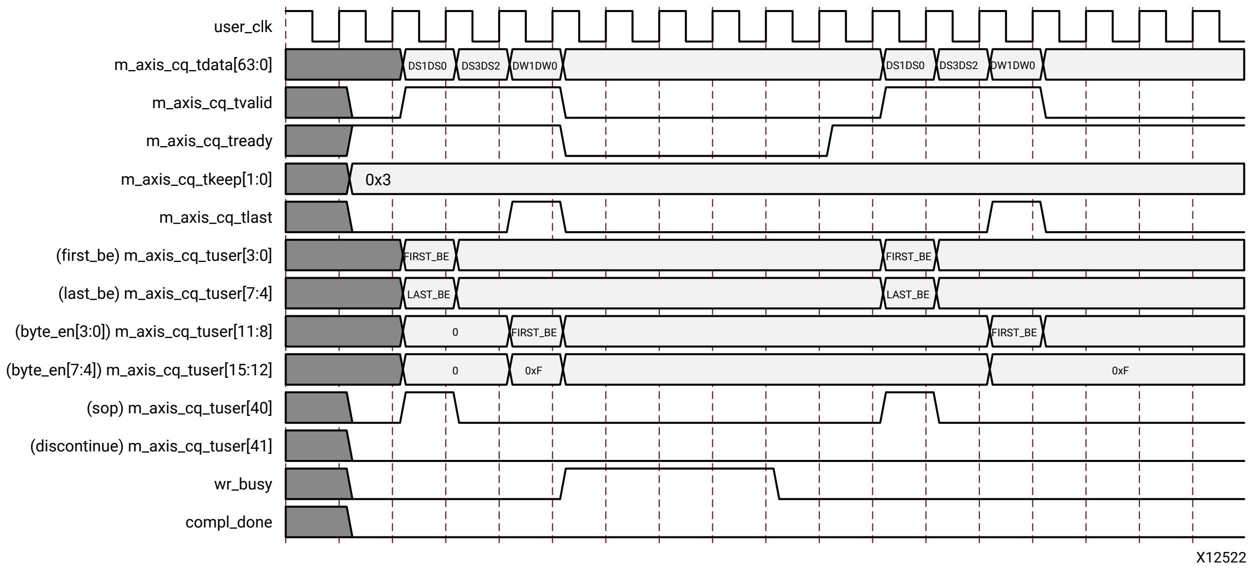
Figure 117: **Back-to-Back Read Transactions**



PIO Write Transaction

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

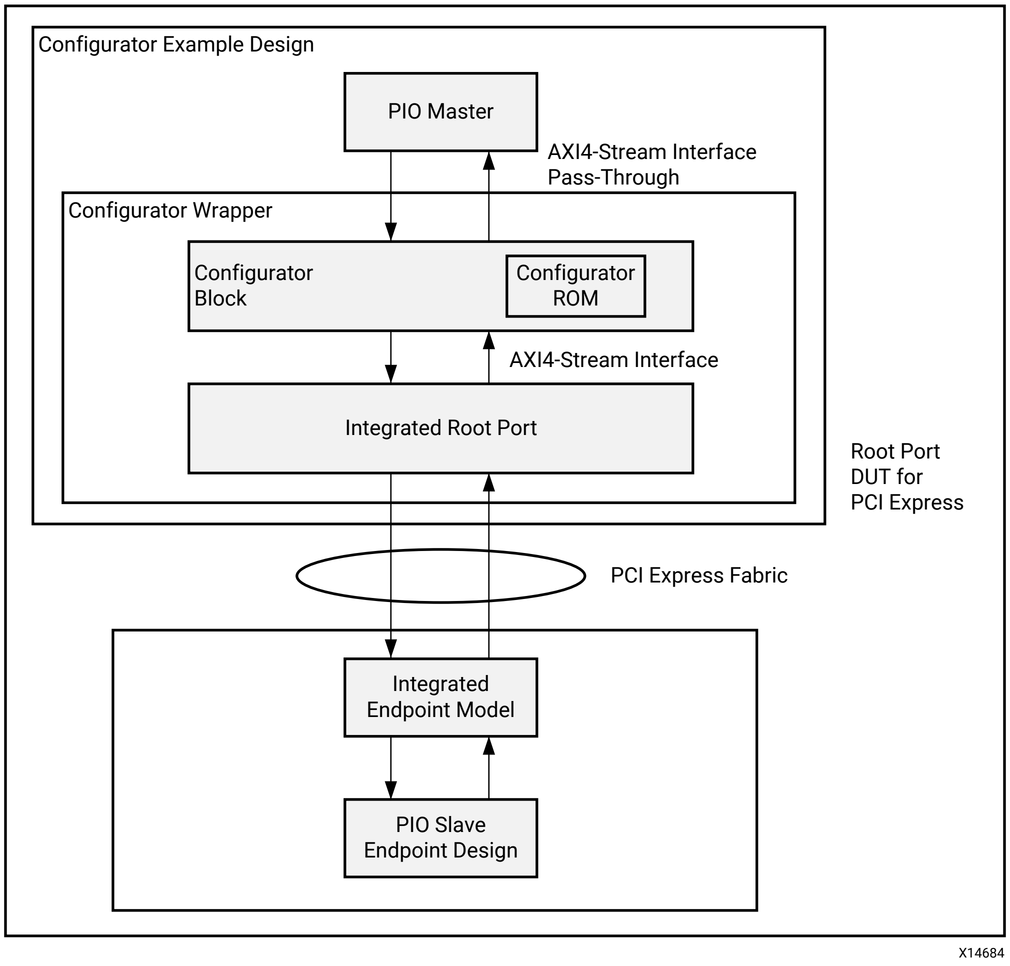
Figure 118: **Back-to-Back Write Transactions**


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

Configurator: Rootport Example Design

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

Figure 119: **Configurator Example Design**


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

Configurator File Structure

The following table defines the Configurator example design file structure.

Table 73: **Example Design File Structure**

File	Description
xilinx_pcie4_uscale_rp.v	Top-level wrapper file for Configurator example design
cgator_wrapper.v	Wrapper for Configurator and Root Port
cgator.v	Wrapper for Configurator sub-blocks
cgator_cpl_decoder.v	Completion decoder
cgator_pkt_generator.v	Configuration TLP generator
cgator_tx_mux.v	Transmit AXI4-Stream muxing logic
cgator_gen2_enabler.v	5.0 Gb/s directed speed change module
cgator_controller.v	Configurator transmit engine
cgator_cfg_rom.data	Configurator ROM file

Table 73: **Example Design File Structure** *(cont'd)*

File	Description
pio_master.v	Wrapper for PIO Master
pio_master_controller.v	TX and RX Engine for PIO Master
pio_master_checker.v	Checks incoming User-Application Completion TLPs
pio_master_pkt_generator.v	Generates User-Application TLPs

The hierarchy of the Configurator example design is:

`xilinx_pcie_uscale_rp.v`

- `cgator_wrapper`
  - `pcie_uscale_core_top`(in the source directory): This directory contains all the source files for the core in Root Port Configuration.
  - `cgator`
    - `cgator_cpl_decoder`
    - `cgator_pkt_generator`
    - `cgator_tx_mux`
    - `cgator_gen2_enabler`
    - `cgator_controller`: This directory contains `<cgator_cfg_rom.data>` (specified by ROM_FILE).
- `pio_master`
  - `pio_master_controller`
  - `pio_master_checker`
  - `pio_master_pkt_generator`

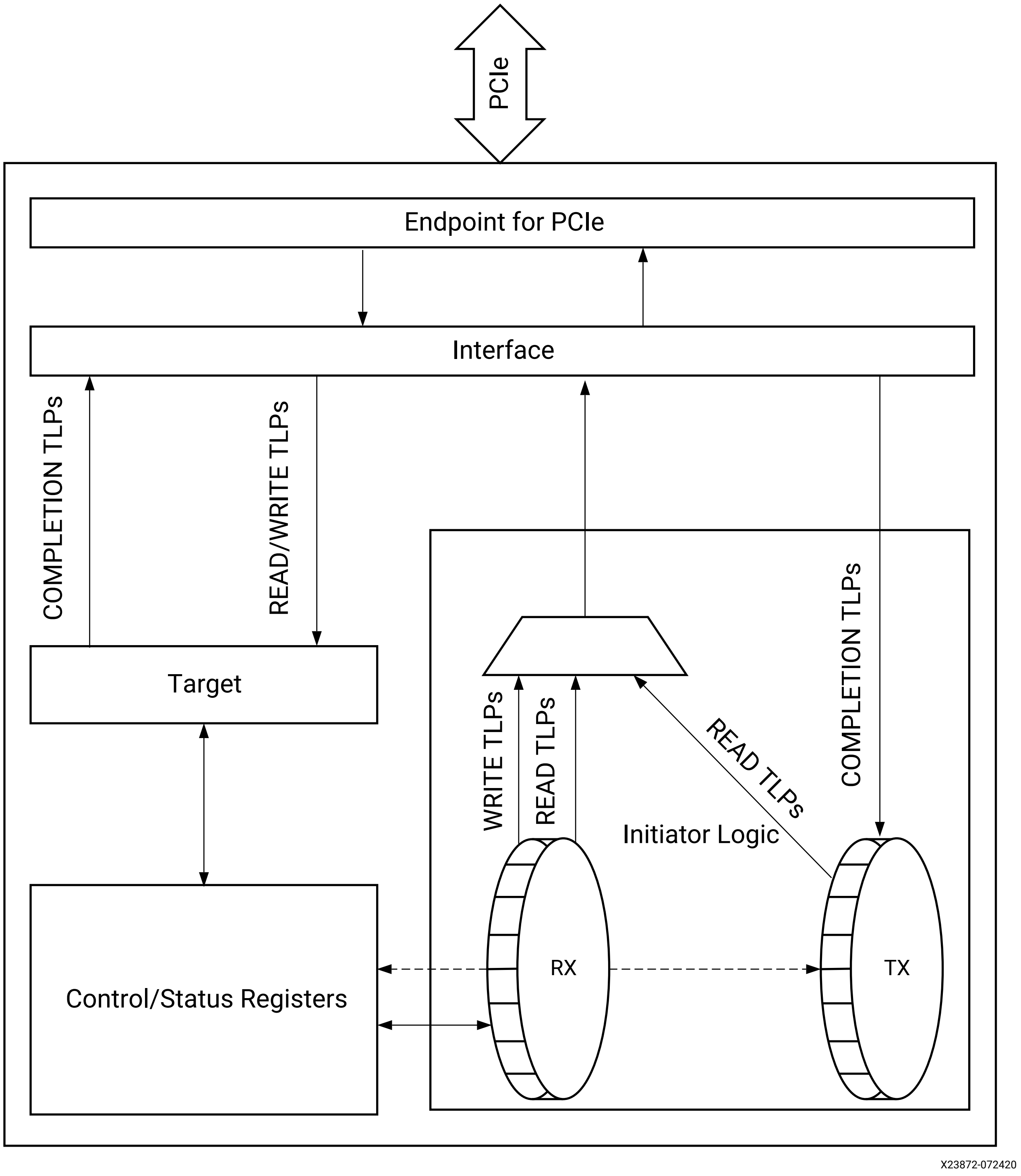
Note: `cgator_cfg_rom.data` is the default name of the ROM data file. You can override this by changing the value of the ROM_FILE parameter.

Bus Master DMA: Endpoint Example Design

A Bus Master DMA (BMD) implementation is the most common type of DMA found in systems based on PCI Express. BMD implementations reside within the Endpoint device and are called Bus Masters because they initiate the movement of data to (Memory Writes) and from (Memory Reads) system memory.

The BMD architecture, shown in the figure below, consists of initiator logic, target logic, status/control registers, interface logic, and the endpoint core for PCI Express.

Figure 120: **Bus Master DMA Design Architecture**



Target Logic

Target logic is responsible for capturing single Dword memory write (MWr) and memory read (MRd) transaction layer packets (TLPs) presented on the interface. MWr and MRd TLPs are sent to the endpoint through Programmed Input/Output (PIO) transactions, and are used to monitor and control the DMA hardware. The function of the target logic is to update the status and control registers during MWr transactions and return Completions with Data for all incoming MRd transactions. All incoming MWr packets are 32-bit and contain a one Dword (32-bits) payload. Incoming MRd packets should only request 1 Dword of data at a time resulting in Completions with Data of a single Dword.

Control and Status Registers

The control and status registers contain operational information for the DMA controller. It is important to note that the example BMD design provided is primarily used to measure performance of data transfers and, consequently, contains status registers that may not be needed in typical designs. You can choose to remove these and their associated logic if needed.

Initiator Logic

The function of the initiator block is to generate memory write or memory read TLPs depending on whether an upstream or downstream transfer is selected. The Bus Master DMA design only supports generating one type of a data flow at a single time. The Bus Master enable bit (Bit 2 of PCI Command Register) must be set to initiate TLP traffic upstream. No transactions are allowed to cross the 4K boundary.

The initiator logic generates memory write TLPs when transferring data from the endpoint to system memory. The Write DMA control and status registers specify the address, size, payload content, and number of TLPs to be sent.

All registers are defined in "Appendix A: Design Descriptor Registers" of *Bus Master Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions* (XAPP1052).

The table below shows BMD design file structure for 64/128/256-bit configuration.

Table 74: **BMD 64/128/256-bit Design File Structure**

File	Description
BMD_AXIST.v	Top-level design wrapper
BMD_AXIST_EP.v	Top-level module
BMD_AXIST_EP_MEM_ACCESS.v	Memory access module
BMD_AXIST_EP_MEM.v	Memory module
BMD_AXIST_TX_ENGINE.v	BMD Transmit engine
BMD_AXIST_RX_ENGINE.v	BMD Receive engine
BMD_AXIST_INTR_CTRL.v	Interrupt controller

Table 74: **BMD 64/128/256-bit Design File Structure** *(cont'd)*

File	Description
BMD_AXIST_TO_CTRL.v	Turn-off controller module

The table below shows BMD design file structure for 512-bit configuration.

Table 75: **BMD 512-bit Design File Structure**

File	Description
BMD_AXIST_512.v	Top-level design wrapper
BMD_AXIST_EP_512.v	Top-level module
BMD_AXIST_EP_MEM_ACCESS.v	Memory access module
BMD_AXIST_EP_MEM.v	Memory module
BMD_AXIST_RC_512.v	BMD Requester Completion module
BMD_AXIST_CQ_512.v	BMD Completer Request module
BMD_AXIST_RQ_512.v	BMD Requester Request module
BMD_AXIST_RQ_WRITE_512.v	BMD Requester Request write module
BMD_AXIST_RQ_READ_512.v	BMD Requester Request read module
BMD_AXIST_RQ_MUX_512.v	BMD Requester Write/Read MUX module
BMD_AXIST_CC_512.v	BMD Completer Completion module
BMD_AXIST_INTR_CTRL.v	Interrupt controller
BMD_AXIST_TO_CTRL.v	Turn-off controller module

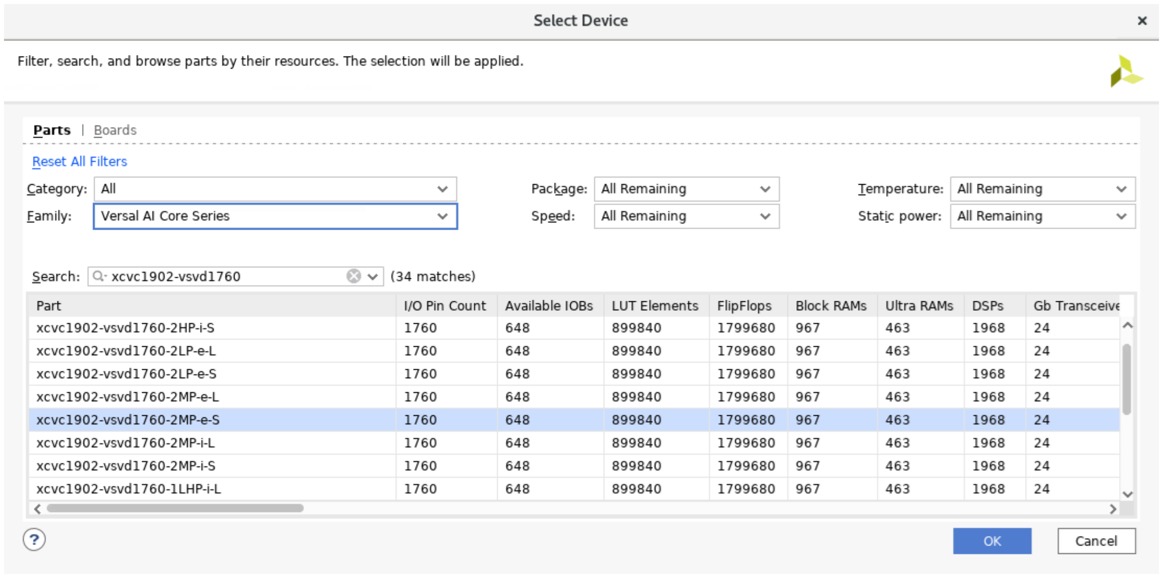
Generating the Core

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

1. Start the Vivado IP catalog.
2. Select **File→ Project→ New**.
3. Enter a project name and location, then click **Next**. This example uses `project_name.xpr` and `project_dir`.
4. In the New Project wizard pages, do not add sources, existing IP, or constraints.
5. From the Part tab (below), select these filter options:
    - Family: Versal™
    - Device: xcvc1902
    - Package: vsvd1760
    - Speed Grade: -2MP

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

6. Select `xcvc1902-vsvd1760-2MP-e-S` from the list.



Select Device

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

Parts | Boards

Reset All Filters

Category: All

Family: Versal AI Core Series

Package: All Remaining

Speed: All Remaining

Temperature: All Remaining

Static power: All Remaining

Search: xcvc1902-vsvd1760 (34 matches)

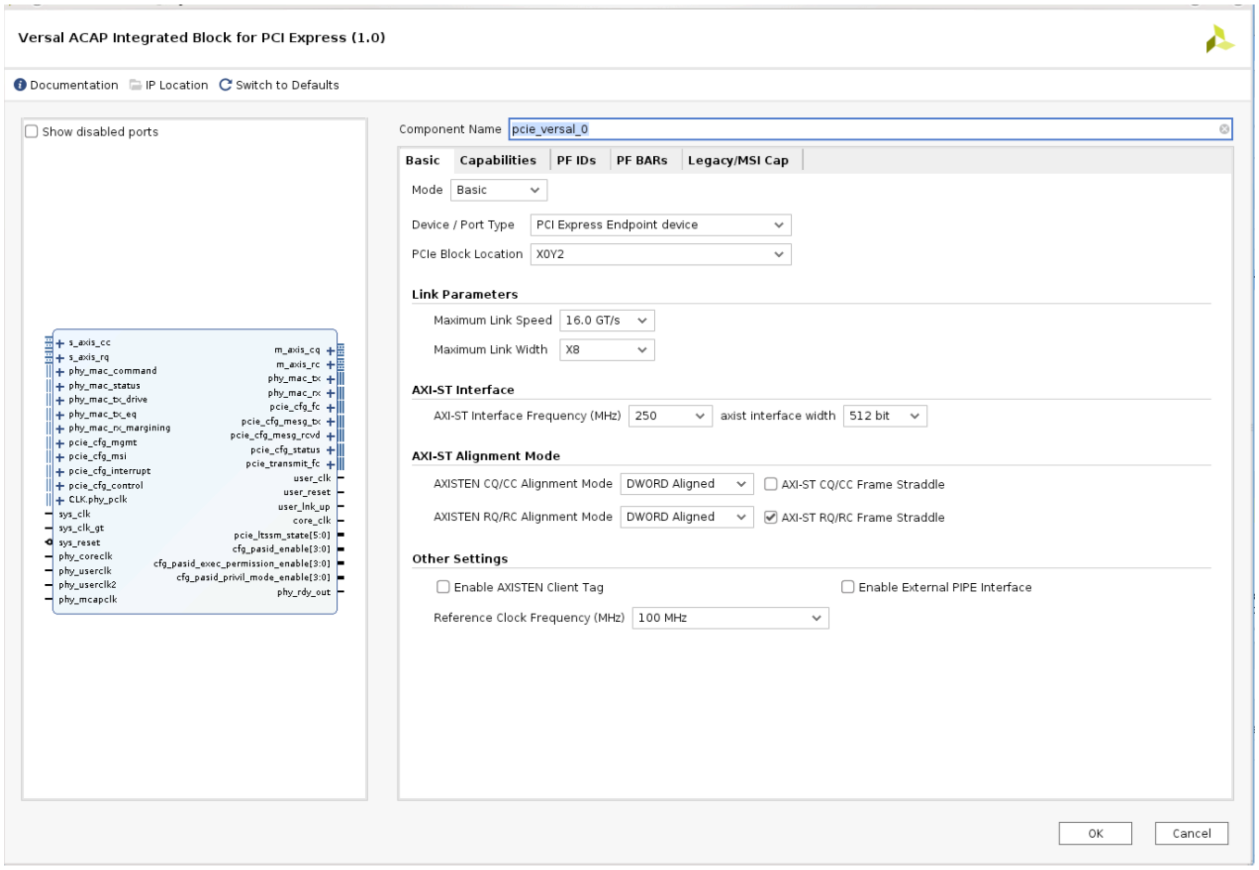
Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	Ultra RAMs	DSPs	Gb Transceive
xcvc1902-vsvd1760-2HP-i-S	1760	648	899840	1799680	967	463	1968	24
xcvc1902-vsvd1760-2LP-e-L	1760	648	899840	1799680	967	463	1968	24
xcvc1902-vsvd1760-2LP-e-S	1760	648	899840	1799680	967	463	1968	24
xcvc1902-vsvd1760-2MP-e-L	1760	648	899840	1799680	967	463	1968	24
xcvc1902-vsvd1760-2MP-e-S	1760	648	899840	1799680	967	463	1968	24
xcvc1902-vsvd1760-2MP-i-L	1760	648	899840	1799680	967	463	1968	24
xcvc1902-vsvd1760-2MP-i-S	1760	648	899840	1799680	967	463	1968	24
xcvc1902-vsvd1760-1LHP-i-L	1760	648	899840	1799680	967	463	1968	24

7. In the final project summary page, click **OK**.

8. In the Vivado IP catalog, expand **Standard Bus Interfaces → PCI Express**, and double-click **Versal ACAP Integrated Block for PCIe** to display the Customize IP dialog box.

9. In the Component Name field, enter a name for the core.

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



Component Name: pcie_versal_0

Mode: Basic

Device / Port Type: PCI Express Endpoint device

PCIe Block Location: X0Y2

Link Parameters

Maximum Link Speed: 16.0 GT/s

Maximum Link Width: X8

AXI-ST Interface

AXI-ST Interface Frequency (MHz): 250 axist interface width: 512 bit

AXI-ST Alignment Mode

AXISTEN CQ/CC Alignment Mode: DWORD Aligned ☐ AXI-ST CQ/CC Frame Straddle

AXISTEN RQ/RC Alignment Mode: DWORD Aligned ☑ AXI-ST RQ/RC Frame Straddle

Other Settings

☐ Enable AXISTEN Client Tag ☐ Enable External PIPE Interface

Reference Clock Frequency (MHz): 100 MHz

10. From the Device/Port Type drop-down menu, select the appropriate device/port type of the core (**Endpoint** or **Root Port**).

TIP: *The PCIe reset pin for PL PCIe designs can be connected to any compatible single ended PL I/O pin location. If your board is compatible for either CPM4 or PL PCIe usage, you can use the CPM4 pin MIO38 to route the* `sys_rst_n`. *When this is done, the PL PCIe can use the reset as routed to the PL.*

Before opening the example design, set the following Tcl property to use the reset on the MIO38 pin:

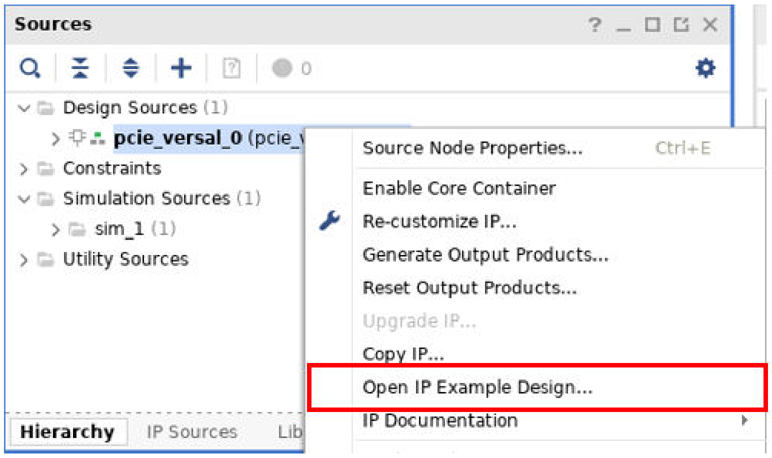
```
set_property CONFIG.insert_cips {true} [get_ips pcie_versal_0]
```

11. Click **OK** to generate the core using the default parameters.

Opening the Example Design

1. To open IP example design, right-click on the generated IP core, and select **Open IP Example Design**.

   ***Note*:** For core generation, see the Generating the Core section above.

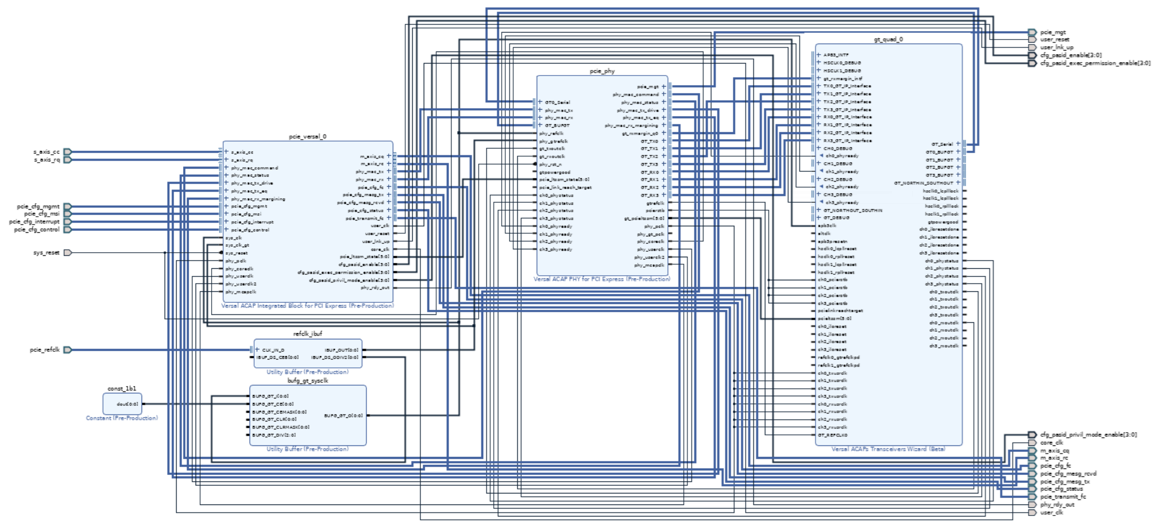


2. In the open window, click **OK**. Vivado creates a directory named `<core name>_ex` in the Example project directory. Adjust the path if needed, and click **OK**.

The generated example design consists of two block designs:

- endpoint (`design_ep`)
- root port (`design_rp`)

The block design for a Gen4x4 PCIe endpoint example design is shown below:


pcie_versal_0
pcie_phy
gt_quad_0
refclk_ibuf
bufg_gt_sysclk
const_1b1
Versal ACAP Integrated Block for PCI Express (Pre-Production)
Versal ACAP PHY for PCI Express (Pre-Production)
Versal ACAP Transceivers Wizard (Beta)
Utility Buffer (Pre-Production)
Constant (Pre-Production)
s_axis_cc
s_axis_rq
pcie_cfg_mgmt
sys_reset
pcie_refclk
pcie_mgt
user_reset
user_lnk_up
core_clk
m_axis_cq
m_axis_rc
user_clk

- The `pcie_versal_0` block is the PCIe IP Core with the configuration set before opening example design.
- The `pcie_phy` and `gt_quad_0` blocks are the PHY IP and GT Wizard for the PCIe core. Unlike in UltraScale+™ devices, where the PHY IP and GT Wizard are within PCIe IP in Versal, in Versal ACAP, the two blocks are external to PCIe IP core.

- The `refclk_ibuf`, `bufg_gt_sysclk`, and `const_1b1` blocks are used for the `sys_clk` buffer. The `refclk_ibuf` block is taking input clock pins `sys_clk_n` and `sys_clk_p`. The output of the `bufg_gt_sysclk` is the system reference clock, which is input for PCIe IP. Similar to UltraScale+, `sys_rst_n` is also an input, and all three inputs are constrained in the top XDC file.
- The Root Port block design is similar to Endpoint, and has an PCIe core with the required blocks generated. Root Port block design is used in simulation.

Simulating the Example Design

The example design provides a quick way to simulate and observe the behavior of the core for PCI Express® Endpoint and Root port Example design projects generated using the Vivado Design Suite.

The currently supported simulators are:

- Vivado simulator (default)
- Questa Advanced Simulator
- Cadence Incisive Enterprise Simulator (IES)
- Synopsys Verilog Compiler Simulator (VCS)

You can generate an example design project and run simulation on the example project. The simulator uses the example design test bench and test cases provided along with the example design for both the design configurations.

A simulation, using the default Vivado simulator, is run as follows:

1. In the Sources Window, right-click the example project file (`.xci`), and select **Open IP Example Design**.

   The example project is created.

2. In the Flow Navigator (left-hand pane), under Simulation, right-click **Run Simulation** and select **Run Behavioral Simulation**.

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

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

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

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

Endpoint Configuration

The simulation environment provided with the Versal ACAP Integrated Block for PCI Express® core in Endpoint configuration performs simple memory access tests on the PIO example design. Transactions are generated by the Root Port Model and responded to by the PIO example design.

- PCI Express Transaction Layer Packets (TLPs) are generated by the test bench transmit user application (`pci_exp_usrapp_tx`). As it transmits TLPs, it also generates a log file, `tx.dat`.
- PCI Express TLPs are received by the test bench receive user application (`pci_exp_usrapp_rx`). As the user application receives the TLPs, it generates a log file, `rx.dat`.

  For more information about the test bench, see Root Port Model Test Bench for Endpoint in the next chapter.

Synthesizing and Implementing the Example Design

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

1. Go to the XCI file, right-click, and select **Open IP Example Design**.

   A new Vivado tool window opens with the project name “example_project” within the project directory.
2. In the Flow Navigator, click **Run Synthesis** and **Run Implementation**.

> **TIP:** *Click **Run Implementation** first to run both synthesis and implementation. Click **Generate Bitstream** to run synthesis, implementation, and then bitstream.*

Chapter 7

Test Bench

Root Port Model Test Bench for Endpoint

The PCI Express Root Port Model is a robust test bench environment that provides a test program interface that can be used with the provided Programmed Input/Output (PIO) design or with your design. The purpose of the Root Port Model is to provide a source mechanism for generating downstream PCI Express TLP traffic to stimulate the customer design, and a destination mechanism for receiving upstream PCI Express TLP traffic from the customer design in a simulation environment.

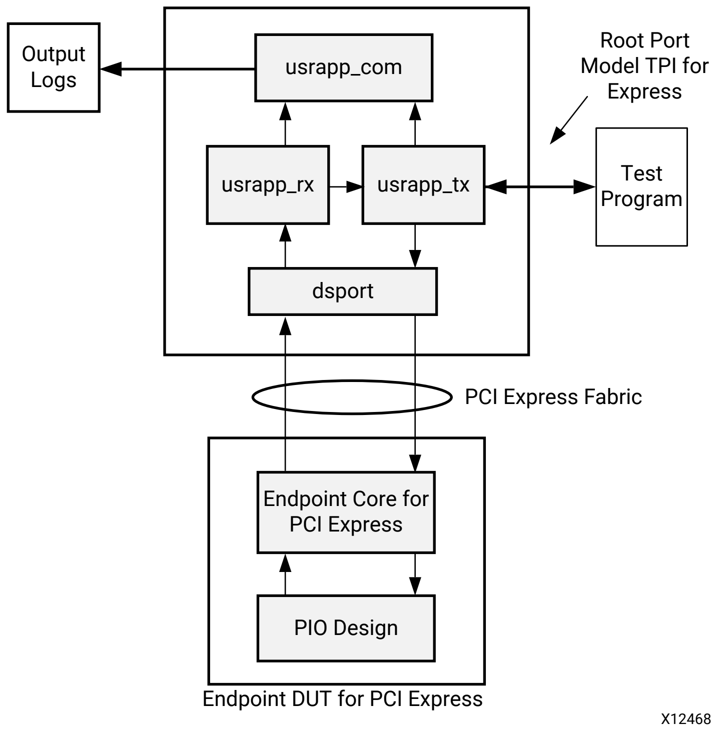
Source code for the Root Port Model is included to provide the model for a starting point for your test bench. All the significant work for initializing the core configuration space, creating TLP transactions, generating TLP logs, and providing an interface for creating and verifying tests is complete. This allows you to focus on verifying the functionality of the design rather than spending time developing an Endpoint core test bench infrastructure.

The Root Port Model consists of:

- Test Programming Interface (TPI), which allows you to stimulate the Endpoint device for the PCI Express.
- Example tests that illustrate how to use the test program TPI.
- Verilog source code for all Root Port Model components, which allow you to customize the test bench.

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

Figure 121: **Root Port Model and Top-Level Endpoint**



Architecture

The Root Port Model consists of these blocks:

- `dsport` (Root Port)
- `usrapp_tx`
- `usrapp_rx`
- `usrapp_com` (Verilog only)

The `usrapp_tx` and `usrapp_rx` blocks interface with the `dsport` block for transmission and reception of TLPs to/from the Endpoint Design Under Test (DUT). The Endpoint DUT consists of the Endpoint for PCIe and the PIO design (displayed) or customer design.

The `usrapp_tx` block sends TLPs to the `dsport` block for transmission across the PCI Express Link to the Endpoint DUT. In turn, the Endpoint DUT device transmits TLPs across the PCI Express Link to the `dsport` block, which are subsequently passed to the `usrapp_rx` block. The `dsport` and core are responsible for the data link layer and physical link layer processing when communicating across the PCI Express logic. Both `usrapp_tx` and `usrapp_rx` use the `usrapp_com` block for shared functions, for example, TLP processing and log file outputting.

Transaction sequences or test programs are initiated by the `usrapp_tx` block to stimulate the Endpoint device fabric interface. TLP responses from the Endpoint device are received by the `usrapp_rx` block. Communication between the `usrapp_tx` and `usrapp_rx` blocks allow the `usrapp_tx` block to verify correct behavior and act accordingly when the `usrapp_rx` block has received TLPs from the Endpoint device.

Scaled Simulation Timeouts

The simulation model of the core uses scaled-down times during link training to allow for the link to train in a reasonable amount of time during simulation. According to the *PCI Express Specification, rev. 3.0* (http://www.pcisig.com/specifications) , there are various timeouts associated with the link training and status state machine (LTSSM) states. The core scales these timeouts by a factor of 256 during simulation, except in the Recovery Speed_1 LTSSM state, where the timeouts are not scaled.

Test Selection

Available Tests

The following table describes the tests provided with the Root Port Model.

Table 76: **Root Port Model Provided Tests**

Test Name	Language	Description
sample_smoke_test0	Verilog	Issues a PCI Type 0 Configuration Read TLP and waits for the completion TLP; then compares the value returned with the expected Device/Vendor ID value.
sample_smoke_test1	Verilog	Performs the same operation as sample_smoke_test0 but makes use of expectation tasks. This test uses two separate test program threads: one thread issues the PCI Type 0 Configuration Read TLP and the second thread issues the Completion with Data TLP expectation task. This test illustrates the form for a parallel test that uses expectation tasks. This test form allows for confirming reception of any TLPs from your design. Additionally, this method can be used to confirm reception of TLPs when ordering is unimportant.

Verilog Test Selection

The Verilog test model used for the Root Port Model lets you specify the name of the test to be run as a command line parameter to the simulator.

To change the test to be run, change the `TESTNAME` value, which is defined in the test files `sample_tests1.v` and `pio_tests.v`. This mechanism is used for Mentor Graphics Advanced Simulator. The Vivado simulator uses the `-testplusarg` option to specify `TESTNAME`. For example:

```
demo_tb.exe-gui -view wave.wcfg -wdb wave_isim -tclbatch isim_cmd.tcl
-testplusarg TESTNAME=sample_smoke_test0
```

Waveform Dumping

For information on simulator waveform dumping, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900).

Verilog Flow

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

Output Logging

When a test fails on the example or customer design, the test programmer debugs the offending test case. Typically, the test programmer inspects the wave file for the simulation and cross-reference this to the messages displayed on the standard output. Because this approach can be very time consuming, the Root Port Model offers an output loggingmechanism to assist the tester with debugging failing test cases to speed the process.

The Root Port Model creates three output log files during each simulation run. They are `tx.dat`, `rx.dat`, and `error.dat`. The `rx.dat` and `tx.dat` files each contain a detailed record of every TLP that was received and transmitted, respectively, by the Root Port Model.

TIP: *With an understanding of the expected TLP transmission during a specific test case, you can isolate the failure.*

The `error.dat` file is used in conjunction with the expectation tasks. Test programs that use the expectation tasks generate a general error message to standard output. Detailed information about the specific comparison failures that have occurred due to the expectation error is found in `error.dat`.

Parallel Test Programs

There are two classes of tests 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 (later in this chapter) is an example of a sequential test. Sequential tests are very useful when verifying behavior that have events with a known order.

- **Parallel tests:** Tests involving more than one process thread. The test `sample_smoke_test1` is an example of a parallel test with two process threads. Parallel tests are very useful when verifying that a specific set of events have occurred, however the order of these events are not known.

A typical parallel test uses the form of one command thread and one or more expectation threads. These threads work together to verify the device functionality. The role of the command thread is to create the necessary TLP transactions that cause the device to receive and generate TLPs. The role of the expectation threads is to verify the reception of an expected TLP. The Root Port Model TPI has a complete set of expectation tasks to be used in conjunction with parallel tests.

Because the example design is a target-only device, only Completion TLPs can be expected by parallel test programs while using the PIO design. However, the full library of expectation tasks can be used for expecting any TLP type when used in conjunction with the customer design (which can include bus-mastering functionality).

Completer Model

The Completer Model is enabled through the Vivado Tcl Console by executing the following command after a core has been configured:

```
set_property-dict [list CONFIG.completer_model {true} [get_ips <PCIE
IP Core Name>]
```

When the core is configured with the 512-bit AXI Interface, you can opt in for this Completer Model test bench which can be used in conjunction with your design to exercise bus-mastering functionality (upstream direction traffic from the Endpoint DUT to the Root Port Model).

The Completer Model provides a Root Port side memory array (DATA_STORE_2) that can be written through a Memory Write transaction and be read through a Memory Read transaction from the Endpoint DUT. This memory can be configured through two different parameters available at the top level of the Root Port Model module (`xilinx_pcie_uscale_rp.v`).

- **RP_BAR[63:0]:** Provides the address of the first byte of the DATA_STORE_2 array.
- **RP_BAR_SIZE[5:0] :** Provides the number of byte address bits -1 of the DATA_STORE_2 array. For example, a value of 11 provides 2^(11+1) bytes or 4 KB of available memory.

Each memory transaction is checked against the memory array location based on the two aforementioned parameters, byte enables, 4K boundaries, Max Payload Size, and Max Read Request Size rules set at the Root Port model. Each Memory Read Completion returned is split according to Max Payload Size and Read Completion Boundary rules. The Completer Model also supports a Zero Length Write packet which intercepts the packet but does not store its payload data, and a Zero Length Read packet which returns a one DW payload data.

Test Description

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

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

Test Program: pio_writeReadBack_test0

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

Expanding the Root Port Model

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

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

Root Port Model TPI Task List

The Root Port Model TPI tasks include the following tasks.

Test Setup Tasks

Table 77: **Test Setup Tasks**

Name	Input(s)		Description
TSK_SYSTEM_INITIALIZATION	None		Waits for transaction interface reset and link-up between the Root Port Model and the Endpoint DUT. This task must be invoked prior to the Endpoint core initialization.
TSK_USR_DATA_SETUP_SEQ	None		Initializes global 4096 byte DATA_STORE array and resizable DATA_STORE_2 array entries to sequential values from zero to 4095.
TSK_TX_CLK_EAT	clock count	31:30	Waits clock_count transaction interface clocks.

Table 77: **Test Setup Tasks** *(cont'd)*

Name	Input(s)		Description
TSK_SIMULATION_TIMEOUT	timeout	31:0	Sets master simulation timeout value in units of transaction interface clocks. This task should be used to ensure that all DUT tests complete.

TLP Tasks

Table 78: **TLP Tasks**

Name	Input(s)		Description
TSK_TX_TYPE0_CONFIGURATION_READ	tag_ reg_addr_ first_dw_be_	7:0 11:0 3:0	Sends a Type 0 PCI Express Config Read TLP from Root Port Model to reg_addr of Endpoint DUT with tag_ and first_dw_be_ inputs. Cpld returned from the Endpoint DUT uses the contents of global EP_BUS_DEV_FNS as the completer ID.
TSK_TX_TYPE1_CONFIGURATION_READ	tag_ reg_addr_ first_dw_be_	7:0 11:0 3:0	Sends a Type 1 PCI Express Config Read TLP from Root Port Model to reg_addr_ of Endpoint DUT with tag_ and first_dw_be_ inputs. CplD returned from the Endpoint DUT uses the contents of global EP_BUS_DEV_FNS as the completer ID.
TSK_TX_TYPE0_CONFIGURATION_WRITE	tag_ reg_addr_ reg_data_ first_dw_be_	7:0 11:0 31:0 3:0	Sends a Type 0 PCI Express Config Write TLP from Root Port Model to reg_addr_ of Endpoint DUT with tag_ and first_dw_be_ inputs. Cpl returned from the Endpoint DUT uses the contents of global EP_BUS_DEV_FNS as the completer ID.
TSK_TX_TYPE1_CONFIGURATION_WRITE	tag_ reg_addr_ reg_data_ first_dw_be_	7:0 11:0 31:0 3:0	Sends a Type 1 PCI Express Config Write TLP from Root Port Model to reg_addr_ of Endpoint DUT with tag_ and first_dw_be_ inputs. Cpl returned from the Endpoint DUT uses the contents of global EP_BUS_DEV_FNS as the completer ID.
TSK_TX_MEMORY_READ_32	tag_ tc_ len_ addr_ last_dw_be_ first_dw_be_	7:0 2:0 10:0 31:0 3:0 3:0	Sends a PCI Express Memory Read TLP from Root Port to 32-bit memory address addr_ of Endpoint DUT. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.
TSK_TX_MEMORY_READ_64	tag_ tc_ len_ addr_ last_dw_be_ first_dw_be_	7:0 2:0 10:0 63:0 3:0 3:0	Sends a PCI Express Memory Read TLP from Root Port Model to 64-bit memory address addr_ of Endpoint DUT. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.

Table 78: **TLP Tasks** *(cont'd)*

Name	Input(s)		Description
TSK_TX_MEMORY_WRITE_32	tag_ tc_ len_ addr_ last_dw_be_ first_dw_be_ ep_	7:0 2:0 10:0 31:0 3:0 3:0 –	Sends a PCI Express Memory Write TLP from Root Port Model to 32-bit memory address addr_ of Endpoint DUT. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID. The global DATA_STORE byte array is used to pass write data to task.
TSK_TX_MEMORY_WRITE_64	tag_ tc_ len_ addr_ last_dw_be_ first_dw_be_ ep_	7:0 2:0 10:0 63:0 3:0 3:0 –	Sends a PCI Express Memory Write TLP from Root Port Model to 64-bit memory address addr_ of Endpoint DUT. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID. The global DATA_STORE byte array is used to pass write data to task.
TSK_TX_COMPLETION	req_id_ tag_ tc_ len_ byte_count_ lower_addr_ comp_status_ ep_	15:0 7:0 2:0 10:0 2:0 11:0 6:0 -	Sends a PCI Express Completion TLP from Root Port Model to the Endpoint DUT using global RP_BUS_DEV_FNS as the completer ID, req_id_ input as the requester ID. comp_status_ input can be set to one of the following: 3'b000 = Successful Completion 3'b001 = Unsupported Request 3'b010 = Configuration Request Retry Status 3'b100 = Completer Abort
TSK_TX_COMPLETION_DATA	req_id_ tag_ tc_ len_ byte_count_ lower_addr_ ram_ptr comp_status_ ep_	15:0 7:0 2:0 10:0 11:0 6:0 RP_BAR_ SIZE:0 2:0 –	Sends a PCI Express Completion with Data TLP from Root Port Model to the Endpoint DUT using global RP_BUS_DEV_FNS as the completer ID, req_id_ input as the requester ID. The global DATA_STORE_2 byte array is used to pass completion data to task and the ram_ptr input is used to offset the starting byte within this array.
TSK_TX_MESSAGE	tag_ tc_ len_ data_ message_rtg_ message_code_	7:0 2:0 10:0 63:0 2:0 7:0	Sends a PCI Express Message TLP from Root Port Model to Endpoint DUT. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.
TSK_TX_MESSAGE_DATA	tag_ tc_ len_ data_ message_rtg_ message_code_	7:0 2:0 10:0 63:0 2:0 7:0	Sends a PCI Express Message with Data TLP from Root Port Model to Endpoint DUT. The global DATA_STORE byte array is used to pass message data to task. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.

Table 78: **TLP Tasks** *(cont'd)*

Name	Input(s)		Description
TSK_TX_IO_READ	tag_ addr_ first_dw_be_	7:0 31:0 3:0	Sends a PCI Express I/O Read TLP from Root Port Model to I/O address addr_[31:2] of the Endpoint DUT. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.
TSK_TX_IO_WRITE	tag_ addr_ first_dw_be_ data	7:0 31:0 3:0 31:0	Sends a PCI Express I/O Write TLP from Root Port Model to I/O address addr_[31:2] of the Endpoint DUT. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.
TSK_TX_BAR_READ	bar_index byte_offset tag_ tc_	2:0 31:0 7:0 2:0	Sends a PCI Express one Dword Memory 32, Memory 64, or I/O Read TLP from the Root Port Model to the target address corresponding to offset byte_offset from BAR bar_index of the Endpoint DUT. This task sends the appropriate Read TLP based on how BAR bar_index has been configured during initialization. This task can only be called after TSK_BAR_INIT has successfully completed. The request uses the contents of global RP_BUS_DEV_FNS as the Requester ID.
TSK_TX_BAR_WRITE	bar_index byte_offset tag_ tc_ data_	2:0 31:0 7:0 2:0 31:0	Sends a PCI Express one Dword Memory 32, Memory 64, or I/O Write TLP from the Root Port to the target address corresponding to offset byte_offset from BAR bar_index of the Endpoint DUT. This task sends the appropriate Write TLP based on how BAR bar_index has been configured during initialization. This task can only be called after TSK_BAR_INIT has successfully completed.
TSK_WAIT_FOR_READ_DATA	None		Waits for the next completion with data TLP that was sent by the Endpoint DUT. On successful completion, the first Dword of data from the CplD is stored in the global P_READ_DATA. This task should be called immediately following any of the read tasks in the TPI that request Completion with Data TLPs to avoid any race conditions. By default this task locally times out and terminate the simulation after 1000 transaction interface clocks. The global cpld_to_finish can be set to zero so that local timeout returns execution to the calling test and does not result in simulation timeout. For this case test programs should check the global cpld_to, which when set to one indicates that this task has timed out and that the contents of P_READ_DATA are invalid.

Table 78: **TLP Tasks** *(cont'd)*

Name	Input(s)		Description
TSK_TX_SYNCHRONIZE	first_ active_ last_call_ tready_sw_	- - - -	Waits for assertion of AXI4-Stream Requester Request or Completer Completion Interface Ready signal and synchronizes the output in the log file to each transaction currently active. first_ input indicates start of packet. active_ input indicates a transaction is currently in progress last_call_ input indicates end of packet tready_sw input selects Requester Request or Completer Completion Interface Ready signal
TSK_BUILD_RC_TO_PCIE_PKT	rc_data_QW0 rc_data_QW1 m_axis_rc_tkeep m_axis_rc_tlast	63:0 63:0 KEEP_ WIDTH-1:0 -	Converts AXI4-Stream packet at Requester Completion Interface from a Descriptor packet format to PCIe TLP packet format for logging purposes.
TSK_BUILD_CQ_TO_PCIE_PKT	cq_data cq_be m_axis_cq_tdata	63:0 7:0 63:0	Converts AXI4-Stream packet at Completer Request Interface from a Descriptor packet format to PCIe TLP packet format for logging purposes.
TSK_BUILD_CPLD_PKT	cq_addr cq_be m_axis_cq_tdata	63:0 15:0 63:0	Returns Completion or Completion with Data for Memory Read received from the Endpoint DUT. When the Completer Model is used, the completion produced is split according to Max Payload Size and Read Completion Boundary rules set at the Root Port Model. Completion with Data uses data stored in the global DATA_STORE_2 array.

BAR Initialization Tasks

Table 79: **BAR Initialization Tasks**

Name	Input(s)	Description
TSK_BAR_INIT	None	Performs a standard sequence of Base Address Register initialization tasks to the Endpoint device using the PCI Express fabric. Performs a scan of the Endpoint PCI BAR range requirements, performs the necessary memory and I/O space mapping calculations, and finally programs the Endpoint so that it is ready to be accessed. On completion, the user test program can begin memory and I/O transactions to the device. This function displays to standard output a memory and I/O table that details how the Endpoint has been initialized. This task also initializes global variables within the Root Port Model that are available for test program usage. This task should only be called after TSK_SYSTEM_INITIALIZATION.

Table 79: **BAR Initialization Tasks** *(cont'd)*

Name	Input(s)	Description
TSK_BAR_SCAN	None	Performs a sequence of PCI Type 0 Configuration Writes and Configuration Reads using the PCI Express logic to determine the memory and I/O requirements for the Endpoint. The task stores this information in the global array BAR_INIT_P_BAR_RANGE[]. This task should only be called after TSK_SYSTEM_INITIALIZATION.
TSK_BUILD_PCIE_MAP	None	Performs memory and I/O mapping algorithm and allocates Memory 32, Memory 64, and I/O space based on the Endpoint requirements. This task has been customized to work in conjunction with the limitations of the PIO design and should only be called after completion of TSK_BAR_SCAN.
TSK_DISPLAY_PCIE_MAP	None	Displays the memory mapping information of the Endpoint core PCI Base Address Registers. For each BAR, the BAR value, the BAR range, and BAR type is given. This task should only be called after completion of TSK_BUILD_PCIE_MAP.

Example PIO Design Tasks

Table 80: **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 ACAP block RAM data bus interface is correctly connected by performing a 32-bit walking ones data test to the I/O or memory address pointed to by the input bar_index. For an exhaustive test, this task should be called four times, once for each block RAM used in the PIO design.
TSK_MEM_TEST_ADDR_BUS	bar_index nBytes	2:0 31:0	Tests whether the PIO design ACAP 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.

Table 80: **Example PIO Design Tasks** *(cont'd)*

Name	Input(s)		Description
TSK_MEM_TEST_DEVICE	bar_index nBytes	2:0 31:0	Tests the integrity of each bit of the PIO design ACAP 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 sys_rst_n signal in board.v file. Forces the sys_rst_n signal to assert the reset. Use TSK_RESET (1'b1) to assert the reset and TSK_RESET (1'b0) to release the reset signal.
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

Expectation Tasks

Table 81: **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 81: **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 in conjunction with Bus Master designs.
TSK_EXPECT_MEMRD64	traffic_class td ep attr length requester_id tag last_dw_be first_dw_be address	2:0 - - 1:0 10:0 15:0 7:0 3:0 3:0 61:0	Expect status	Waits for a 64-bit Address Memory Read TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. This task can only be used in conjunction with Bus Master designs.
TSK_EXPECT_MEMWR	traffic_class td ep attr length requester_id tag last_dw_be first_dw_be address	2:0 - - 1:0 10:0 15:0 7:0 3:0 3:0 29:0	Expect status	Waits for a 32-bit Address Memory Write TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. This task can only be used in conjunction with Bus Master designs.

Table 81: **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 in conjunction with Bus Master designs.
TSK_EXPECT_IOWR	td ep requester_id tag first_dw_be address data	- - 15:0 7:0 3:0 31:0 31:0	Expect status	Waits for an I/O Write TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. This task can only be used in conjunction with Bus Master designs.

Endpoint Model Test Bench for Root Port

The Endpoint model test bench for the core in Root Port configuration is a simple example test bench that connects the Configurator example design and the PCI Express Endpoint model allowing the two to operate like two devices in a physical system. Because the Configurator example design consists of logic that initializes itself and generates and consumes bus traffic, the example test bench only implements logic to monitor the operation of the system and terminate the simulation.

The Endpoint model test bench consists of:

- Verilog or VHDL source code for all Endpoint model components.
- PIO slave design.

The figure earlier in this chapter illustrates the Endpoint model coupled with the Configurator example design.

Architecture

The Endpoint model consists of these blocks:

- PCI Express Endpoint (the core in Endpoint configuration) model.

- PIO slave design, consisting of:
  - `PIO_RX_ENGINE`
  - `PIO_TX_ENGINE`
  - `PIO_EP_MEM`
  - `PIO_TO_CTRL`

The `PIO_RX_ENGINE` and `PIO_TX_ENGINE` blocks interface with the Endpoint block for reception and transmission of TLPs from/to the Root Port Design Under Test (DUT). The Root Port DUT consists of the core configured as a Root Port and the Configurator Example Design, which consists of a Configurator block and a PIO Master design, or customer design.

The PIO slave design is described in detail in Programmed Input/Output: Endpoint Example Design.

Simulating the Design

The `simulate_mti.do` simulation script file is provided with the model to facilitate simulation with the Mentor Graphics Advanced simulator.

The example simulation script files are located in this directory:

```
<project_dir>/<component_name>/simulation/functional
```

Instructions for simulating the Configurator example design with the Endpoint model are provided in "Simulation" in the Design Flow Steps chapter.

Note: For Cadence IES users, the work construct must be manually inserted into the `cds.lib` file:

```
DEFINE WORK WORK
```

Scaled Simulation Timeouts

The simulation model of the core uses scaled-down times during link training to allow for the link to train in a reasonable amount of time during simulation. According to the *PCI Express Specification, rev. 3.0* (http://www.pcisig.com/specifications) , there are various timeouts associated with the link training and status state machine (LTSSM) states. The core scales these timeouts by a factor of 256 during simulation, except in the Recovery Speed_1 LTSSM state, where the timeouts are not scaled.

Waveform Dumping

For information on simulator waveform dumping, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900).

Output Logging

The test bench outputs messages, captured in the simulation log, indicate the time at which these occur:

- `user_reset` deasserted
- `user_lnk_up` asserted
- `cfg_done` asserted by the Configurator
- `pio_test_finished` asserted by the PIO Master
- Simulation Timeout (if `pio_test_finished` or `pio_test_failed` never asserted)

Migrating and Upgrading

Migrating from Other Device Cores

This section provides information for migrating from the UltraScale+™ device PCIe core to the core.

Ports

New Ports

Table 82: **New Ports in the Versal PL PCIe Core**

Name	I/O	Notes
cfg_ext_tag_enable	O	For details, see Table 19: Configuration Status Interface Port Descriptions.
cfg_atomic_requester_enable	O	
cfg_10b_tag_requester_enable	O	
cfg_fc_ph_scale	O	For details, see Table 24: Configuration Flow Control Interface.
cfg_fc_pd_scale	O	
cfg_fc_nph_scale	O	
cfg_fc_npd_scale	O	
cfg_fc_cplh_scale	O	
cfg_fc_cpld_scale	O	
cfg_pasid_enable	O	For details, see Table 32: Configuration PASID Interface Port Descriptions.
cfg_pasid_exec_permission_enable	O	
cfg_pasid_privil_mode_enable	O	
apb3_*		For details, see Table 33: APB3 Interface Port Descriptions.

Port Updates

Table 83: **Port Width Changes between UltraScale Integrated Block and Versal Integrated Block Cores**

Name	I/O	UltraScale+ Width	Versal PCIe Width	Notes
pcie_rq_tag0	O	8	10	For details, see Table 14: Sideband Signals in s_axis_rq_tuser (512-bit Interface)
pcie_rq_tag1	O	8	10	
pcie0_s_axis_cq_tuser	O	183	229	
pcie0_s_axis_rq_tuser	O	137	183	For details, see Table 10: Sideband Signals in m_axis_cq_tuser (512-bit Interface).

Ports Not Available

The following ports from the UltraScale+ Integrated Block for PCIe IP that are not available the Versal ACAP Integrated Block for PCIe IP.

Table 84: **Ports Not Available in Versal ACAP Integrated Block IP**

Name	I/O	Width	Notes
drp_di	I	16	PCIe DRP ports are replaced with APB3 ports in Versal ACAP. For port details, see Table 33: APB3 Interface Port Descriptions.
drp_do	O	16	
drp_rdy	O	1	
drp_we	I	1	

Generating GT and PHY IP

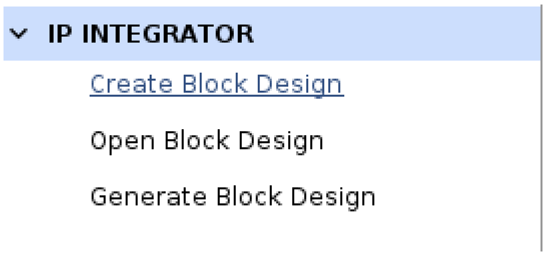
GT Wizards and PHY IP are outside of PCIe core instead of under PCIe hierarchy like in UltraScale+ devices. There are two ways to generate the two cores according to your PCIe core configuration:

- Open the example design – refer to Opening the Example Design.
- Run block automation.

Run Block Automation in IP Integrator

To run block automation:

1. In the Flow Navigator, select **Create Block Design**.



2. Add the `pcie_versal` IP to your block design.

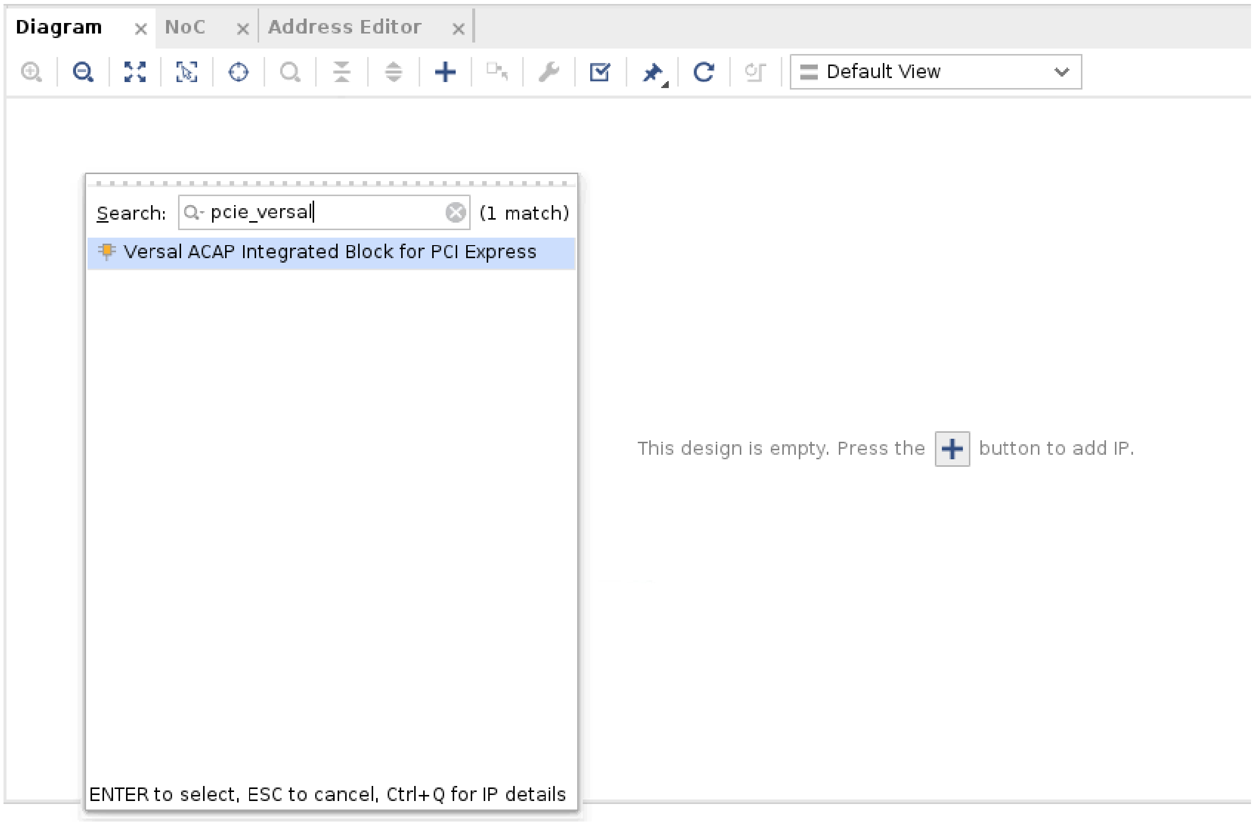
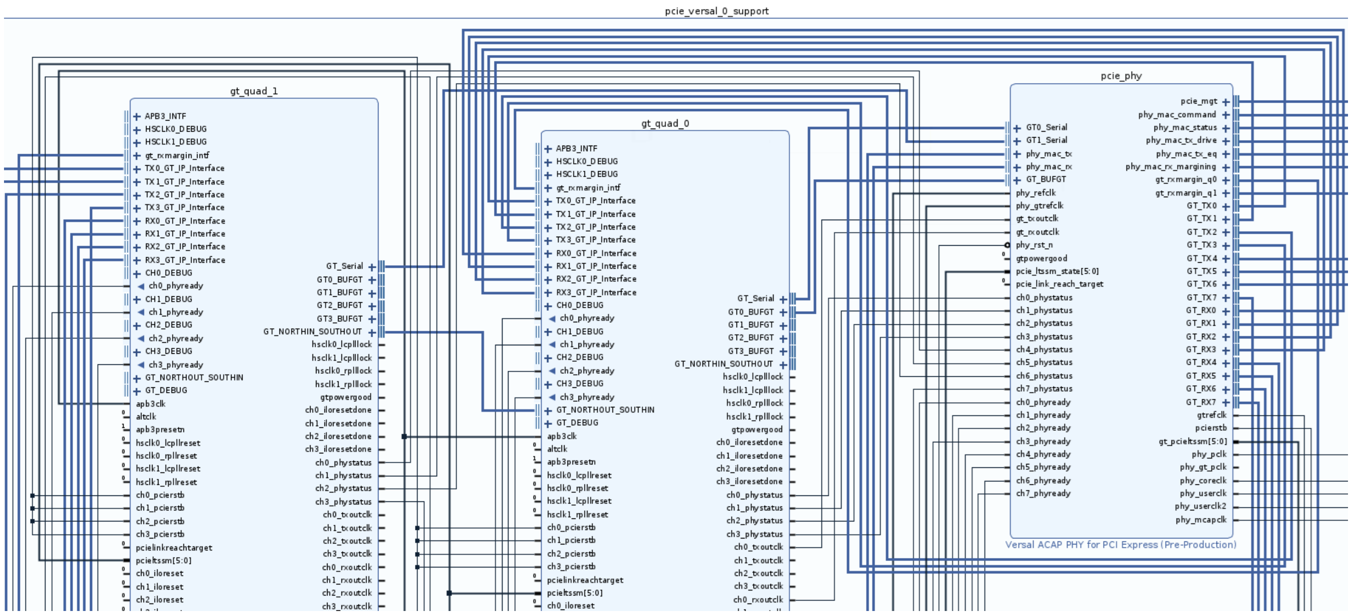


Diagram
NoC
Address Editor
Default View
Search:
pcie_versal
(1 match)
Versal ACAP Integrated Block for PCI Express
ENTER to select, ESC to cancel, Ctrl+Q for IP details
This design is empty. Press the + button to add IP.

3. Configure the pcie_versal core by double-clicking on `pcie_versal` block in your block design (BD).
4. Click **Run Block Automation**, and click **OK**.

The PHY IP and GT quads are found in the generated Vivado IP integrator design, `pcie_versal_0_support`, along with the helper blocks for reset and clock, as seen in the following figure. For more details, see Chapter 6: Example Design.



GT Quad locations can only be set using user constraints in the Xilinx Design Constraints (XDC) file. For more information, see GT Locations.

Clocking

The Versal ACAP Integrated Block for PCIe® core clock topology is similar to the UltraScale+ Device Integrated Block for PCIe. You will find the `phy_clk` module in `pcie_phy` after you generate the PHY IP through the example design or through block automation.

Reset

This core uses the same reset routing as the UltraScale+ devices integrated block, and reset is connected to the input pin. Alternatively, `sys_rst` can be connected from CIPS MIO 38. To use CIPS MIO 38 as a reset source, use either of these methods:

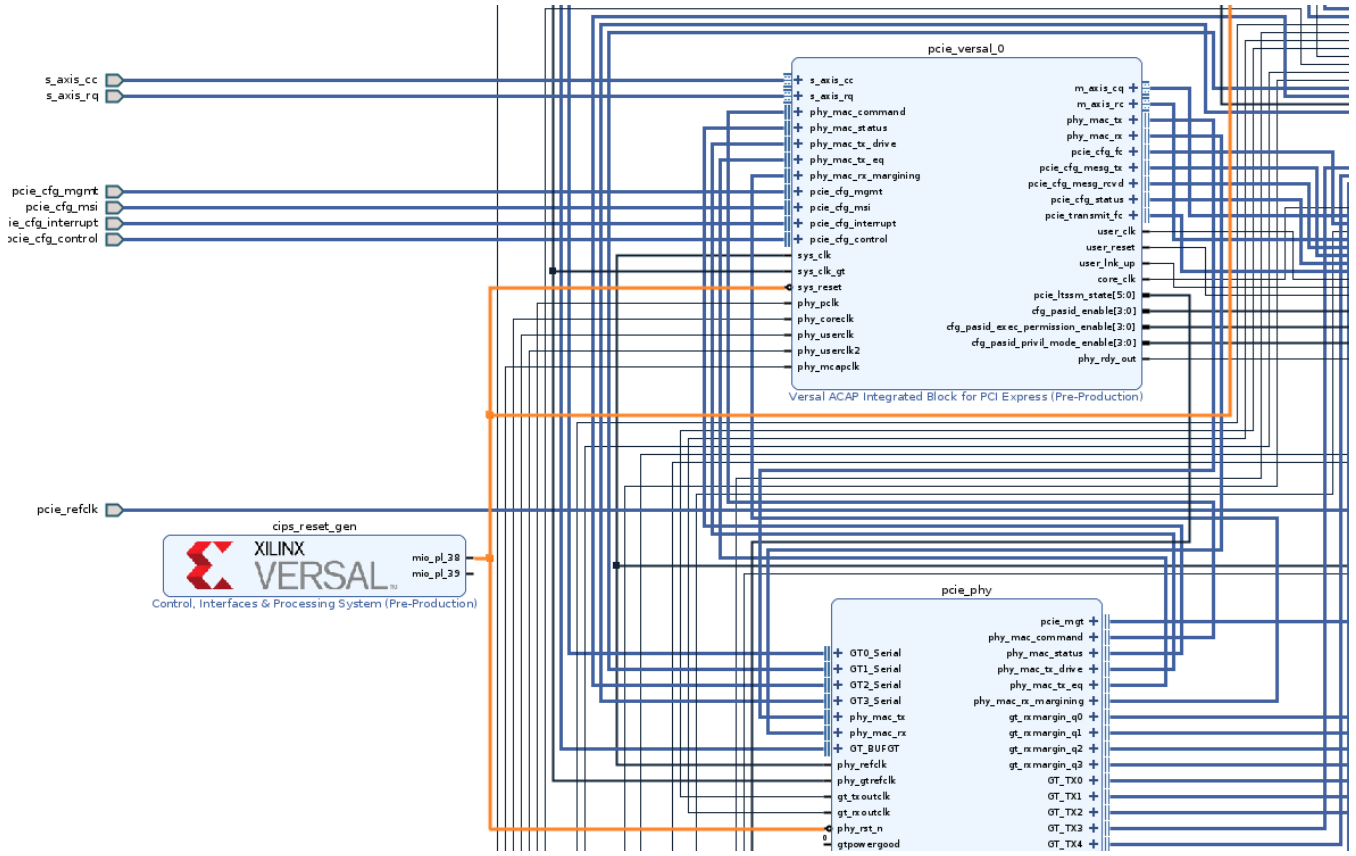
- Enter the following command to enable CIPS before opening the example design:

```
set_property config.insert_cips {true} [get_ips pcie_versal_0]
```

- Set the property in your block design before running block automation:

```
set_property config.insert_cips {true} [get_bd_cells pcie_versal_0]
```

The connection will be like the following diagram, `mio_pl_38` is connected to `sys_reset` of `pcie_versal`, `phy_rst_n` of `pcie_phy`, and as an output for user application to use:



Related Information

CPM4 Additional Considerations

Features

New Features

PASID

PASID Extended Capability structure has been added with which the core supports sending and receiving TLPs containing a PASID TLP Prefix.

10 Bit Tag

The CPM4 PCIe controller supports 10-bit Tag feature, when enabled management of up to 768 tags is possible.

Feature DLLP

Data Link Feature Extended Capability structure has been added for link speed of 16.0 GT/s. It contains programmable control/status information about the local and peer support of the "Data Link Feature Support".

Lane Margining

Lane Margining at the Receiver Extended Capability structure has been added for link speed of 16.0 GT/s

Physical Layer 16.0GT/s Extended Capability

Physical Layer 16.0 GT/s Extended Capability structure has been added for link speed of 16.0 GT/s with which Gen4 equalization status can be read.

Retimers Supported

Link Extension devices (retimers) is supported to interoperate with CPM4 PCIe block for link speed of 16.0 GT/s.

Flow Control Informational Select

More combinations of `cfg_fc_sel` values are supported relative to UltraScale+. See the port description for more details.

MSIX – Additional Vectors

When configured as internal, can support up to 32 vectors per physical function, as compared to 8 vectors per function for VF in UltraScale+. The total number of vectors (2048) remains the same.

Features Not Available, or Limited Usage Features

- TPH capability is not supported.
- CCIX is not supported in Versal PL PCIE4. CCIX is only supported in Versal CPM4.
- Fast PCI Express Endpoint Enumeration using Tandem Configuration not support. This use case addresses the ability to initially load a fully configurable PCI Express protocol solution from a small external ROM, so as to meet the 100 ms enumeration requirement. Support for Tandem Configuration for the PL PCIE block in Versal devices is not currently planned.

  ***Note*:** Any user requiring fast PCIe enumeration should use the PCIe controllers in the CPM, noting that not all Versal devices contain this particular resource. For details, see *Versal ACAP CPM Mode for PCI Express Product Guide* (PG346).

Upgrading

This section is not applicable for this release of the core.

GT Selection and Pin Planning

This appendix provides guidance on gigabit transceiver (GT) selection for Versal™ devices and some key recommendations that should be considered when selecting the GT locations. The GT locations for Versal devices can be customized through the IP customization wizard. This appendix provides guidance for CPM, PL PCIe and PHY IP based solutions. In this guide, the PL PCIe related guidance is of primary importance, while the other related guidance might be relevant and is provided for informational purposes.

A GT Quad is comprised of four GT lanes. When selecting GT Quads for the PL PCIe-based solution , Xilinx® recommends that you use the GT Quad most adjacent to the integrated block. While this is not required, it will improve place, route, and timing for the design.

- Link widths of x1, x2, and x4 require one bonded GT Quad and should not split lanes between two GT Quads.
- A link width of x8 requires two adjacent GT Quads that are bonded and are in the same SLR.
- A link width of x16 requires four adjacent GT Quads that are bonded and are in the same SLR.
- PL PCIe blocks should use GTs adjacent to the PCIe block where possible.
- CPM has a fixed connectivity to GTs based on the CPM configuration.

For GTs on the **left side of the device**, PCIe lane 0 is placed in the bottom-most GT of the bottom-most GT Quad. Subsequent lanes use the next available GTs moving vertically up the device as the lane number increments. This means that the highest PCIe lane number uses the top-most GT in the top-most GT Quad that is used for PCIe.

For GTs on the **right side of the device**, PCIe lane 0 is placed in the top-most GT of the top-most GT Quad. Subsequent lanes use the next available GTs moving vertically down the device as the lane number increments. This means that the highest PCIe lane number uses the bottom-most GT in the bottom-most GT Quad that is used for PCIe.

The PCIe reference clock uses GTREFCLK0 in the PCIe lane 0 GT Quad for x1, x2, x4, and x8 configurations. For x16 configurations the PCIe reference clock should use GTREFCLK0 on a GT Quad associated with lanes 8-11. This allows the clock to be forwarded to all 16 PCIe lanes.

The PCIe reset pins for CPM designs must connect to one of specified pins for each of the two PCIe controllers. The PCIe reset pin for PL PCIe and PHY IP designs can be connected to any compatible PL pin location, or the CPM PCIe reset pins when the corresponding CPM PCIe controller is not in use. This is summarized in the table below.

Table 85: **PCIe Controller Reset Pin Locations**

Versal PCIe Controller	Versal Reset Pin Location
CPM PCIe Controller 0	PS MIO 18
	PMC MIO 24
	PMC MIO 38
CPM PCIe Controller 1	PS MIO 19
	PMC MIO 25
	PMC MIO 39
PL PCIe Controllers	Any compatible single-ended PL I/O pin.
Versal ACAP PHY IP	Any compatible single-ended PL I/O pin.

PL PCIe GT Selection

For PL PCIe blocks the most adjacent GTs should be used and connected to the PCIe solution IP where possible. The PL PCIe block supports x1, x2, x4, x8, and x16 link widths. This will provide the best place, route and timing result for the PCIe solution.

For GTs on the **left side of the device**, PCIe lane 0 is placed in the bottom-most GT of the bottom-most GT Quad. Subsequent lanes use the next available GTs moving vertically up the device as the lane number increments. This means that the highest PCIe lane number uses the top-most GT in the top-most GT Quad that is used for PCIe.

For GTs on the **right side of the device**, PCIe lane 0 is placed in the top-most GT of the top-most GT Quad. Subsequent lanes use the next available GTs moving vertically down the device as the lane number increments. This means that the highest PCIe lane number uses the bottom-most GT in the bottom-most GT Quad that is used for PCIe.

For Versal implementations the GTs are external to the PCIe IP and can be customized as needed beyond the default settings generated with the PCIe example and design automation.

CPM4 Additional Considerations

To facilitate migration from UltraScale™ or UltraScale+™ designs, boards may be designed to use either CPM4 or PL PCIe integrated blocks to implement PCIe solutions. When designing a board to use either CPM4 or the PL PCIe hardblock, the CPM4 pin selection and planning guidelines should be followed because they are more restrictive. By doing this a board can be designed that will work for either a CPM4 or PL PCIe implementation. To route the PCIe reset from the CPM4 to the PL for use with a PL PCIe implementation the following parameter must be set in Vivado prior to customizing the CIPS IP.

```
set_param pcw.enplpciereset 1
```

When this parameter is enabled the PCIe reset for each disabled CPM4 PCIe controller will be routed to the PL. The same CPM4 pin selection limitations will apply and the additional PCIe reset output pins will be exposed at the boundary of the CIPS IP. If the CPM4 PCIe controller is enabled, the PCIe reset will be used internal to the CPM4 and will not be routed to the PL for connectivity to PL PCIe controllers.

Related Information

Reset

GT Locations

Assigning GT Locations

Unlike in UltraScale+ and previous devices where direct assignment of GTs are not possible in the user constraints, in Versal the GT locations assignment can be done in the user constraints, while changing GT locations in GT customization IP is not available. Below is an example of assigning GT locations in a user constraint file.

Note: The gt_quad instances should be assigned contiguously.

```
set_property LOC GTY_QUAD_X0Y6   [get_cells $gt_quads -filter NAME=~*/
gt_quad_3/*]
set_property LOC GTY_QUAD_X0Y5   [get_cells $gt_quads -filter NAME=~*/
gt_quad_2/*]
set_property LOC GTY_QUAD_X0Y4   [get_cells $gt_quads -filter NAME=~*/
gt_quad_1/*]
set_property LOC GTY_QUAD_X0Y3   [get_cells $gt_quads -filter NAME=~*/
gt_quad_0/*]
```

GT Quad Locations

The following table identifies the PCIe lane0 GT Quad(s) that can be used for each PCIe controller location. The Quad shown in bold is the most adjacent or suggested GT Quad for each PCIe lane0 location.

Table 86: **GT Locations**

Device	Package	PCIe Blocks	GT QUAD (X16)	GT QUAD (X8)	GT QUAD (X4 and Below)
XCVC1902	VIVA1596	CPM Controller 0	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**
		CPM Controller 1	N/A	**GTY_QUAD_X0Y5**	**GTY_QUAD_X0Y5**
		X0Y2	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3
		X0Y1	**GTY_QUAD_X0Y3**	GTY_QUAD_X0Y5 GTY_QUAD_X0Y4 **GTY_QUAD_X0Y3**	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3
		X1Y2	**GTY_QUAD_X1Y5**	**GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3	**GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3 GTY_QUAD_X1Y2
		X1Y0	**GTY_QUAD_X1Y5**	GTY_QUAD_X1Y5 GTY_QUAD_X1Y4 **GTY_QUAD_X1Y3**	GTY_QUAD_X1Y5 GTY_QUAD_X1Y4 GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2**
XCVC1902	VSVA2197	CPM Controller 0	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**
		CPM Controller 1	N/A	**GTY_QUAD_X0Y5**	**GTY_QUAD_X0Y5**
		X0Y2	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3
		X0Y1	**GTY_QUAD_X0Y3**	GTY_QUAD_X0Y5 GTY_QUAD_X0Y4 **GTY_QUAD_X0Y3**	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3
		X1Y2	**GTY_QUAD_X1Y6**	**GTY_QUAD_X1Y6** GTY_QUAD_X1Y5 GTY_QUAD_X1Y4	GTY_QUAD_X1Y6 **GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3
		X1Y0	**GTY_QUAD_X1Y3**	**GTY_QUAD_X1Y3** GTY_QUAD_X1Y2 GTY_QUAD_X1Y1	GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2** GTY_QUAD_X1Y1 GTY_QUAD_X1Y0

Table 86: **GT Locations** *(cont'd)*

Device	Package	PCIe Blocks	GT QUAD (X16)	GT QUAD (X8)	GT QUAD (X4 and Below)
XCVC1902	VSVD1760	CPM Controller 0	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**
		CPM Controller 1	N/A	**GTY_QUAD_X0Y5**	**GTY_QUAD_X0Y5**
		X0Y2	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3
		X0Y1	**GTY_QUAD_X0Y3**	GTY_QUAD_X0Y5 GTY_QUAD_X0Y4 **GTY_QUAD_X0Y3**	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3
		X1Y2	N/A	**GTY_QUAD_X1Y4**	**GTY_QUAD_X1Y4** GTY_QUAD_X1Y3
		X1Y0	N/A	**GTY_QUAD_X1Y4**	GTY_QUAD_X1Y4 **GTY_QUAD_X1Y3**
XCVM1802	VFVC1760	CPM Controller 0	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**
		CPM Controller 1	N/A	**GTY_QUAD_X0Y5**	**GTY_QUAD_X0Y5**
		X0Y2	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3
		X0Y1	**GTY_QUAD_X0Y3**	GTY_QUAD_X0Y5 GTY_QUAD_X0Y4 **GTY_QUAD_X0Y3**	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3
		X1Y2	**GTY_QUAD_X1Y6**	**GTY_QUAD_X1Y6** GTY_QUAD_X1Y5 GTY_QUAD_X1Y4	GTY_QUAD_X1Y6 **GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3
		X1Y0	**GTY_QUAD_X1Y3**	**GTY_QUAD_X1Y3** GTY_QUAD_X1Y2 GTY_QUAD_X1Y1	GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2** GTY_QUAD_X1Y1 GTY_QUAD_X1Y0

Table 86: **GT Locations** *(cont'd)*

Device	Package	PCIe Blocks	GT QUAD (X16)	GT QUAD (X8)	GT QUAD (X4 and Below)
XCVM1802	VSVA2197	CPM Controller 0	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**
		CPM Controller 1	N/A	**GTY_QUAD_X0Y5**	**GTY_QUAD_X0Y5**
		X0Y2	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3
		X0Y1	**GTY_QUAD_X0Y3**	GTY_QUAD_X0Y5 GTY_QUAD_X0Y4 **GTY_QUAD_X0Y3**	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3
		X1Y2	**GTY_QUAD_X1Y6**	**GTY_QUAD_X1Y6** GTY_QUAD_X1Y5 GTY_QUAD_X1Y4	GTY_QUAD_X1Y6 **GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3
		X1Y0	**GTY_QUAD_X1Y3**	**GTY_QUAD_X1Y3** GTY_QUAD_X1Y2 GTY_QUAD_X1Y1	GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2** GTY_QUAD_X1Y1 GTY_QUAD_X1Y0
XCVM1802	VSVD1760	CPM Controller 0	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**
		CPM Controller 1	N/A	**GTY_QUAD_X0Y5**	**GTY_QUAD_X0Y5**
		X0Y2	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3
		X0Y1	**GTY_QUAD_X0Y3**	GTY_QUAD_X0Y5 GTY_QUAD_X0Y4 **GTY_QUAD_X0Y3**	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3
		X1Y2	N/A	**GTY_QUAD_X1Y4**	**GTY_QUAD_X1Y4** GTY_QUAD_X1Y3
		X1Y0	N/A	**GTY_QUAD_X1Y4**	GTY_QUAD_X1Y4 **GTY_QUAD_X1Y3**

Appendix C

PCIe Link Debug Enablement

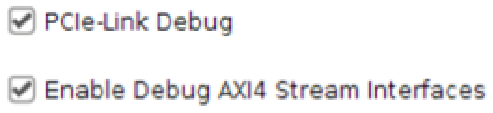
The Versal ACAP Integrated Block for PCI Express® customization provides an option to enable PCIe® Link Debug. Enabling this option will insert a debug core inside the IP core that will be recognized by the Vivado® Hardware Manager and provide PCIe specific debug information and view. The debug view provides information relating to the current link speed, current link width, and LTSSM state transitions, which can facilitate debug of PCIe link related issues.

***Note*:** This appendix provides guidance for both CPM and PL PCIe based solutions. For this core, the PL PCIe related guidance is of primary importance, while the CPM related guidance might be relevant and is provided for informational purposes.

Enabling PCIe Link Debug

Use this guide to enable and connect PCIe Link Debug in a Vivado IP integrator design. This section only describes the additional connections that should be added to enable PCIe Link Debug in a design. It does not discuss how to properly connect the PCIe enabled IPs to create a working design. Block automation can be used, or the connectivity and connections described below should be added to an existing design and IP configuration

1. Enable this option in the core customization wizard, and select the options in the customization GUI, as shown below. The CPM PCIe cores are customized through the CIPS IP and for PL PCIe cores are customized through the Versal ACAP Integrated Block for PCIe IP.

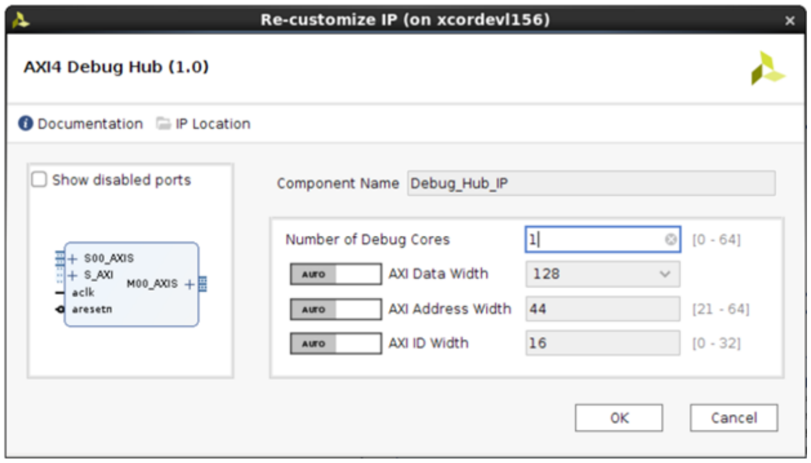
   

   ✓ PCIe-Link Debug

   ✓ Enable Debug AXI4 Stream Interfaces

   This adds the PCIe debug core to the PCIe IP and exposes the debug AXI4-Stream interfaces and ports. The debug AXI4-Stream and interface ports should be connected to a Debug Hub IP within the Versal design to enable debug for the design. The PCIe example design provides one implementation of how the Debug Hub IP can be connected in Versal designs. This is also detailed in the description below.

2. Add the Debug Hub IP to the design and use the following configuration options to enable the Debug Hub AXI Memory Mapped interface along with one set of AXI4-Stream interfaces. Additional AXI4-Stream interfaces can be enabled and connected in your design as desired.



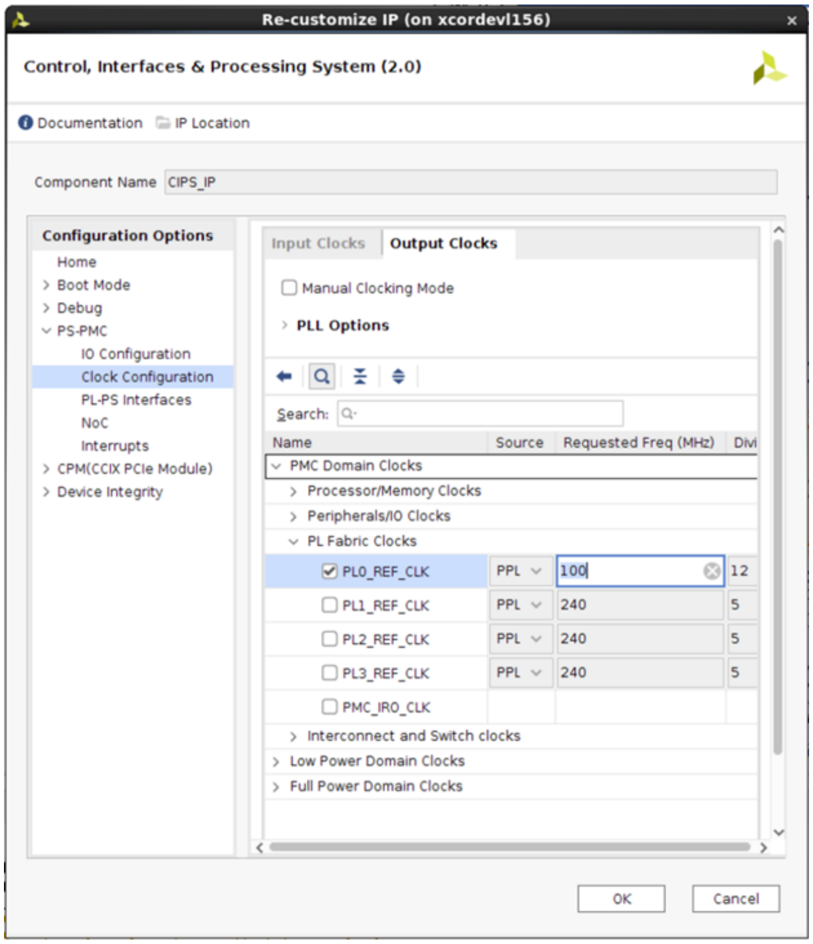
AXI4 Debug Hub (1.0)

Component Name: Debug_Hub_IP

Number of Debug Cores	1	[0 - 64]
AXI Data Width	128	
AXI Address Width	44	[21 - 64]
AXI ID Width	16	[0 - 32]

3. Add the CIPS IP to the design or configure the existing CIPS IP and include the following configuration options. These options will enable an AXI Master, clock, and reset that can be connected to the Debug Hub IP. To do so:

   a. Select **PS-PMC → Clock Configuration → Output Clocks → PMC Domain Clocks → PL Fabric Clocks** selection enable a 100 MHz or similar output clock.

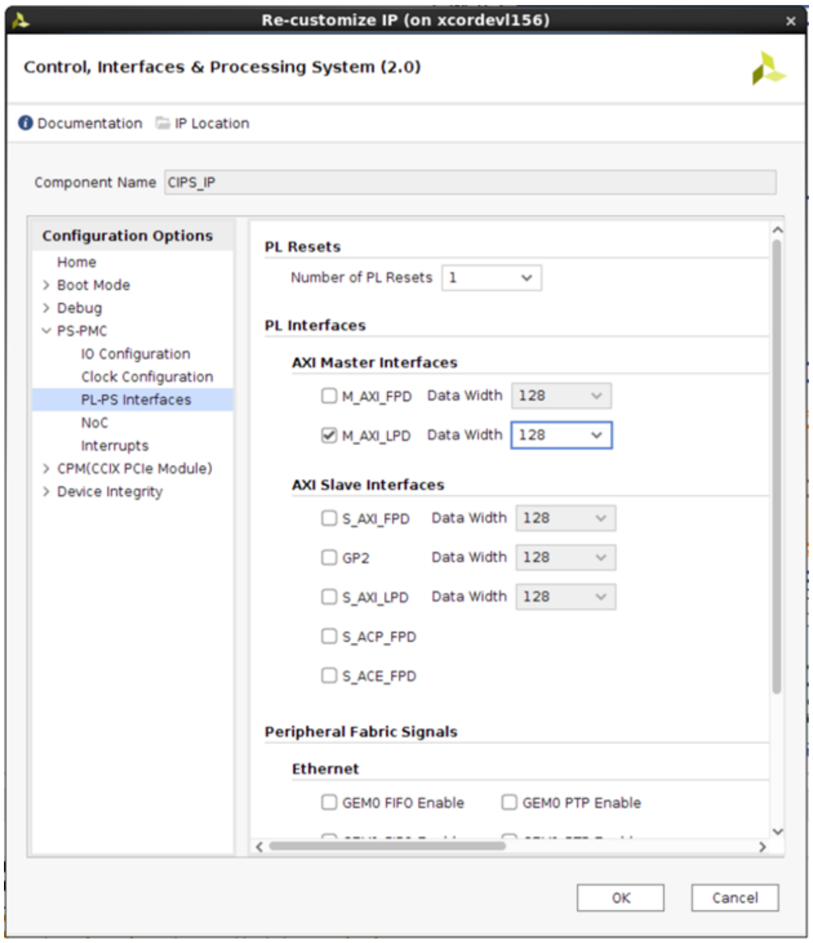


Control, Interfaces & Processing System (2.0)

Component Name: CIPS_IP

Name	Source	Requested Freq (MHz)	Divi
PMC Domain Clocks			
Processor/Memory Clocks			
Peripherals/IO Clocks			
PL Fabric Clocks			
☑ PL0_REF_CLK	PPL	100	12
☐ PL1_REF_CLK	PPL	240	5
☐ PL2_REF_CLK	PPL	240	5
☐ PL3_REF_CLK	PPL	240	5
☐ PMC_IRO_CLK			
Interconnect and Switch clocks			
Low Power Domain Clocks			
Full Power Domain Clocks			

b. Select **PS-PMC→PL-PS Interfaces**, and enable at least one PL reset in **Number of PL Resets**, and the M_AXI_LPD AXI master.



Re-customize IP (on xcordevl156)

Control, Interfaces & Processing System (2.0)

Documentation | IP Location

Component Name: CIPS_IP

Configuration Options: Home, Boot Mode, Debug, PS-PMC (IO Configuration, Clock Configuration, PL-PS Interfaces, NoC, Interrupts), CPM(CCIX PCIe Module), Device Integrity

PL Resets
- Number of PL Resets: 1

PL Interfaces

AXI Master Interfaces
- [ ] M_AXI_FPD — Data Width: 128
- [✓] M_AXI_LPD — Data Width: 128

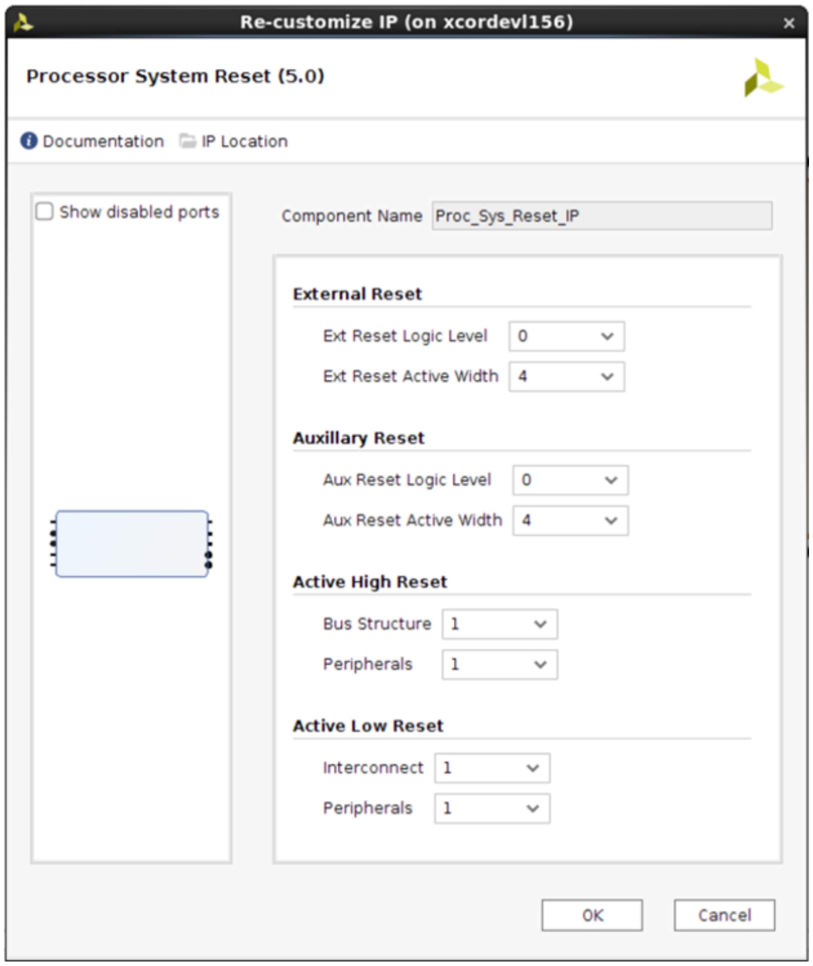
AXI Slave Interfaces
- [ ] S_AXI_FPD — Data Width: 128
- [ ] GP2 — Data Width: 128
- [ ] S_AXI_LPD — Data Width: 128
- [ ] S_ACP_FPD
- [ ] S_ACE_FPD

Peripheral Fabric Signals

Ethernet
- [ ] GEM0 FIFO Enable
- [ ] GEM0 PTP Enable

OK | Cancel

4. Add and configure the Processor System Reset IP.


Re-customize IP (on xcordevl156)

Processor System Reset (5.0)

Component Name: Proc_Sys_Reset_IP

External Reset

Ext Reset Logic Level: 0
Ext Reset Active Width: 4

Auxillary Reset

Aux Reset Logic Level: 0
Aux Reset Active Width: 4

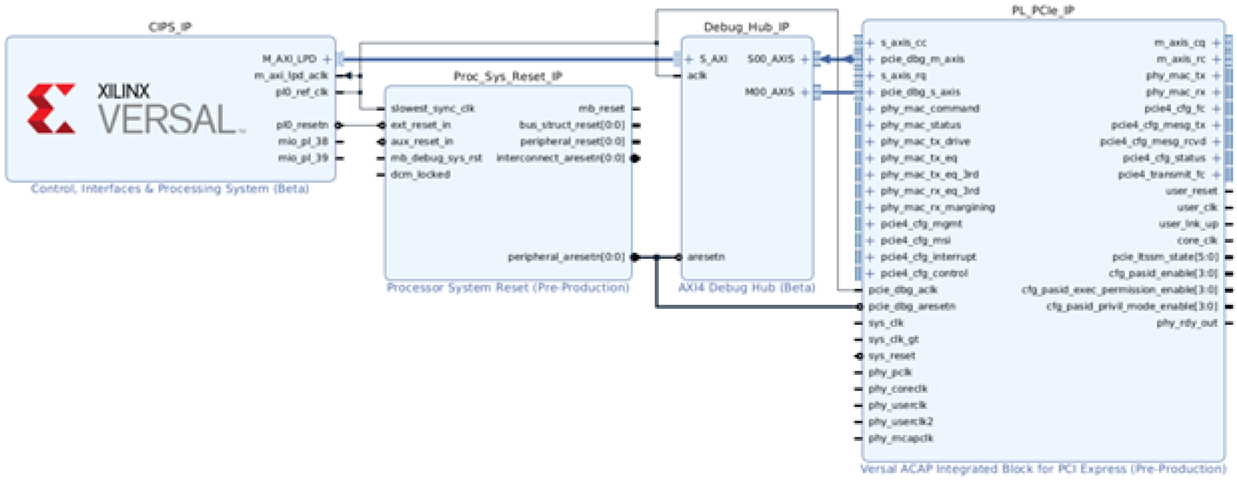
Active High Reset

Bus Structure: 1
Peripherals: 1

Active Low Reset

Interconnect: 1
Peripherals: 1

5. Connect the IPs as shown in the following figures. This may need to be customized to fit with existing design connectivity.



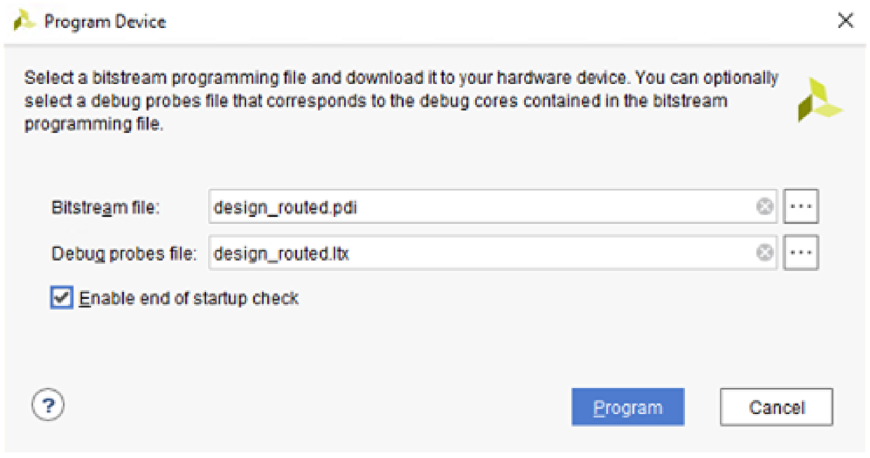
After the debug connections have been added to an Vivado IP integrator design, as shown above, PCIe Link debug is enabled in the generated `.pdi` image. The connections shown above should be added to a full design and are not sufficient to create a working design alone. The PCIe IP ports and the remainder of the design must be created and configured as per the desired operation of the PCIe-enabled IP.

Connecting to PCIe Link Debug in Vivado

Use the following steps to connect Viviado Hardware Manager to the FPGA device and associated PCIe Link Debug enabled design.

1. Open the Hardware Manager.
2. Select the device from the **Tools → Program Device...** drop-down menu.
3. Select the `.pdi` and `.ltx` files for programming the device, and select **Program**.

   ***Note*:** You should not load the `.ltx` file and refresh the target until after the `.pdi` file has been programmed.



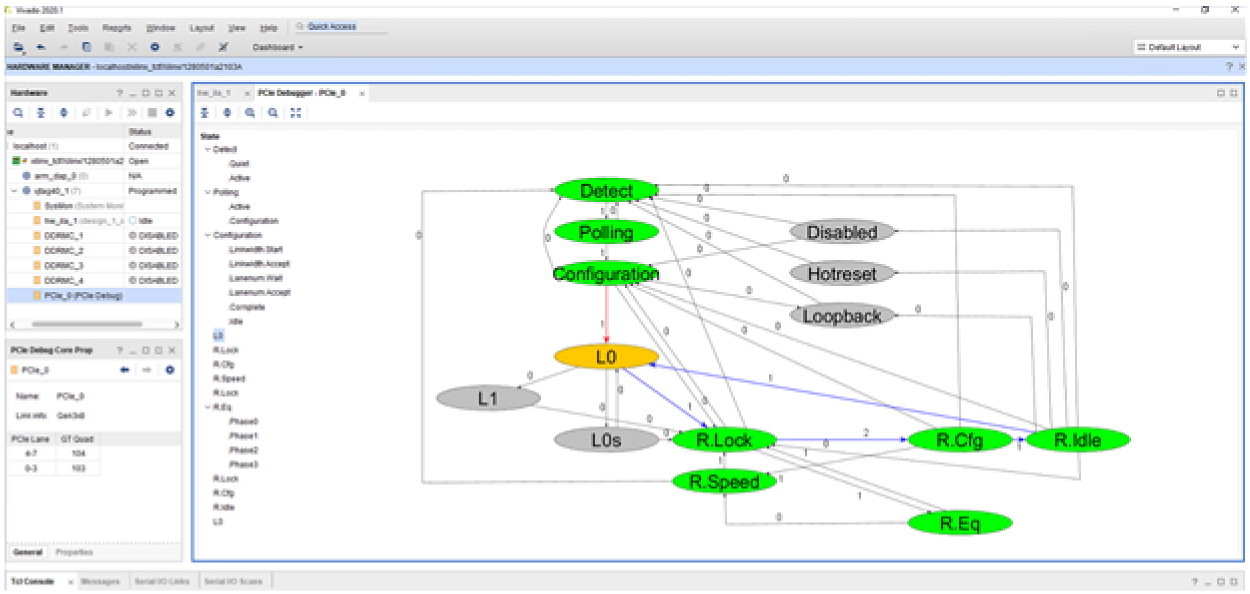
Select a bitstream programming file and download it to your hardware device. You can optionally select a debug probes file that corresponds to the debug cores contained in the bitstream programming file.

Bitstream file: design_routed.pdi

Debug probes file: design_routed.ltx

[✓] Enable end of startup check

4. Select the PCIe Debug core in the Hardware window. You will see three main views that include the PCIe Debug Core Properties, PCIe Link LTSSM State Trace, and the PCIe Link LTSSM State Diagram with transitions.



Using this view, you can observe the active PCIe link status and state transitions. In the PCIe Debug Core Properties window, you can see the name of the PCIe debug core (PCIe_0), the current link status (Gen3x8), and the connected GTs (Quads 103 and 104). The PCIe LTSSM State Trace view shows a hierarchical view of the PCIe LTSSM state machine transitions. The PCIe LTSSM State Diagram provides a graphical display of the PCIe LTSSM states transitions that were traversed during the PCIe link up process. Visited LTSSM states are shown in green, the final or current LTSSM state is shown in yellow and the number of times each transition was traversed is identified on the arcs between states.

In addition to the graphical display, the `report_hw_pcie` command can be used to generate a console text report that contains the PCIe debug information. This information can be shared with others to aid in debugging PCIe Link issues. For this example, the name of the debug core is PCIe_0, and is inserted into the command.

```
report_hw_pcie PCIe_0
```

This information helps determine where in the PCIe link-up process an issue occurred and can guide further debug of link related issues.

Appendix D

Debugging

This appendix includes details about resources available on the Xilinx® Support website and debugging tools.

Finding Help on Xilinx.com

To help in the design and debug process when using the core, the Xilinx Support web page contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support. The Xilinx Community Forums are also available where members can learn, participate, share, and ask questions about Xilinx solutions.

Documentation

This product guide is the main document associated with the core. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page or by using the Xilinx® Documentation Navigator. Download the Xilinx Documentation Navigator from the Downloads page. For more information about this tool and the features available, open the online help after installation.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

The Solution Center specific to the Versal ACAP Integrated Block for PCIe® is listed below.

- Xilinx Solution Center for PCI Express

Answer Records

Answer Records include information about commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a Xilinx product. Answer Records are created and maintained daily ensuring that users have access to the most accurate information available.

Answer Records for this core can be located by using the Search Support box on the main Xilinx support web page. To maximize your search results, use keywords such as:

- Product name
- Tool message(s)
- Summary of the issue encountered

A filter search is available after results are returned to further target the results.

Master Answer Record for the Core

AR 73083.

Technical Support

Xilinx provides technical support on the Xilinx Community Forums for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support if you do any of the following:

- Implement the solution in devices that are not defined in the documentation.
- Customize the solution beyond that allowed in the product documentation.
- Change any section of the design labeled DO NOT MODIFY.

To ask questions, navigate to the Xilinx Community Forums.

Appendix E

Additional Resources and Legal Notices

Xilinx Resources

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

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

***Note*:** For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. *PCI-SIG Specifications* (https://www.pcisig.com/specifications)
2. *Versal ACAP DMA and Bridge Subsystem for PCI Express Product Guide* (PG344)
3. *Versal ACAP GTY and GTYP Transceivers Architecture Manual* (AM002)
4. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
5. *Vivado Design Suite User Guide: Designing with IP* (UG896)
6. *Vivado Design Suite User Guide: Getting Started* (UG910)
7. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
8. *In-System IBERT LogiCORE IP Product Guide* (PG246)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
04/15/2021 Version 1.0	
Limitations	Added topic to report known issues in the release.
Generating the Core	Add note regarding connecting the PCIe reset pin to the MIO38 pin location using a Tcl command.
01/20/2021 Version 1.0	
Customizing and Generating the Core	Updated GUI figures, and descriptions throughout.
Performance and Resource Use	Added section with link to resource use data.
Appendix A: Migrating and Upgrading	New appendix. Added detailed migration information.
11/03/2020 Version 1.0	
General Update	Updated document with correct version 1.0.
07/27/2020 Version 1.0	
Initial Xilinx release.	N/A

Please Read: Important Legal Notices

Copyright