LogiCORE IP UltraScale FPGAs Gen3 Integrated Block for PCI Express v3.0

Product Guide

Vivado Design Suite

PG156 June 4, 2014

XILINX
ALL PROGRAMMABLE™

Table of Contents

Introduction

The Xilinx® LogiCORE™ IP UltraScale FPGAs Gen3 Integrated Block for PCIe® core is a high-bandwidth, scalable, and reliable serial interconnect building block solution for use with UltraScale™ architecture-based devices. The Integrated Block for PCI Express (PCIe) solution supports 1-lane, 2-lane, 4-lane, and 8-lane Endpoint configurations, including Gen1 (2.5 GT/s), Gen2 (5.0 GT/s) and Gen3 (8 GT/s) speeds. It is compliant with *PCI Express Base Specification*, *rev. 3.0* [Ref 2]. This solution supports the AXI4-Stream interface for the customer user interface.

PCI Express offers a serial architecture that alleviates many limitations of parallel bus architectures by using clock data recovery (CDR) and differential signaling. Using CDR (as opposed to source synchronous clocking) lowers pin count, enables superior frequency scalability, and makes data synchronization easier. PCI Express technology, adopted by the PCI-SIG® as the next generation PCI™, is backward-compatible to the existing PCI software model.

With higher bandwidth per pin, low overhead, low latency, reduced signal integrity issues, and CDR architecture, the integrated block sets the industry standard for a high-performance, cost-efficient PCIe solution.

The UltraScale FPGAs Gen3 Integrated Block for PCIe solution is compatible with industry-standard application form factors such as the PCI Express Card Electromechanical (CEM) v3.0 and the PCI Industrial Computer Manufacturers Group (PICMG) v3.4 specifications [Ref 2].

For a list of features, see Feature Summary.

LogiCORE IP Facts Table	
Core Specifics	
Supported Device Family[1]	UltraScale Architecture
Supported User Interfaces	AXI4-Stream
Resources	See Table 2-1.
Provided with Core	
Design Files	Verilog
Example Design	Verilog
Test Bench	Verilog
Constraints File	XDC
Simulation Model	Verilog
Supported S/W Driver[2]	N/A
Tested Design Flows[2]	
Design Entry	Vivado® Design Suite IP Integrator
Simulation	For supported simulators, see the Xilinx Design Tools: Release Notes Guide
Synthesis	Vivado Synthesis
Support	
Provided by Xilinx @ www.xilinx.com/support	

Notes:

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

Overview

The LogiCORE™ IP UltraScale FPGAs Gen3 Integrated Block for PCIe core is a reliable, high-bandwidth, scalable serial interconnect building block for use with UltraScale™ FPGAs. The core instantiates the integrated block found in UltraScale FPGAs.

Figure 1-1 shows the interfaces for the core.

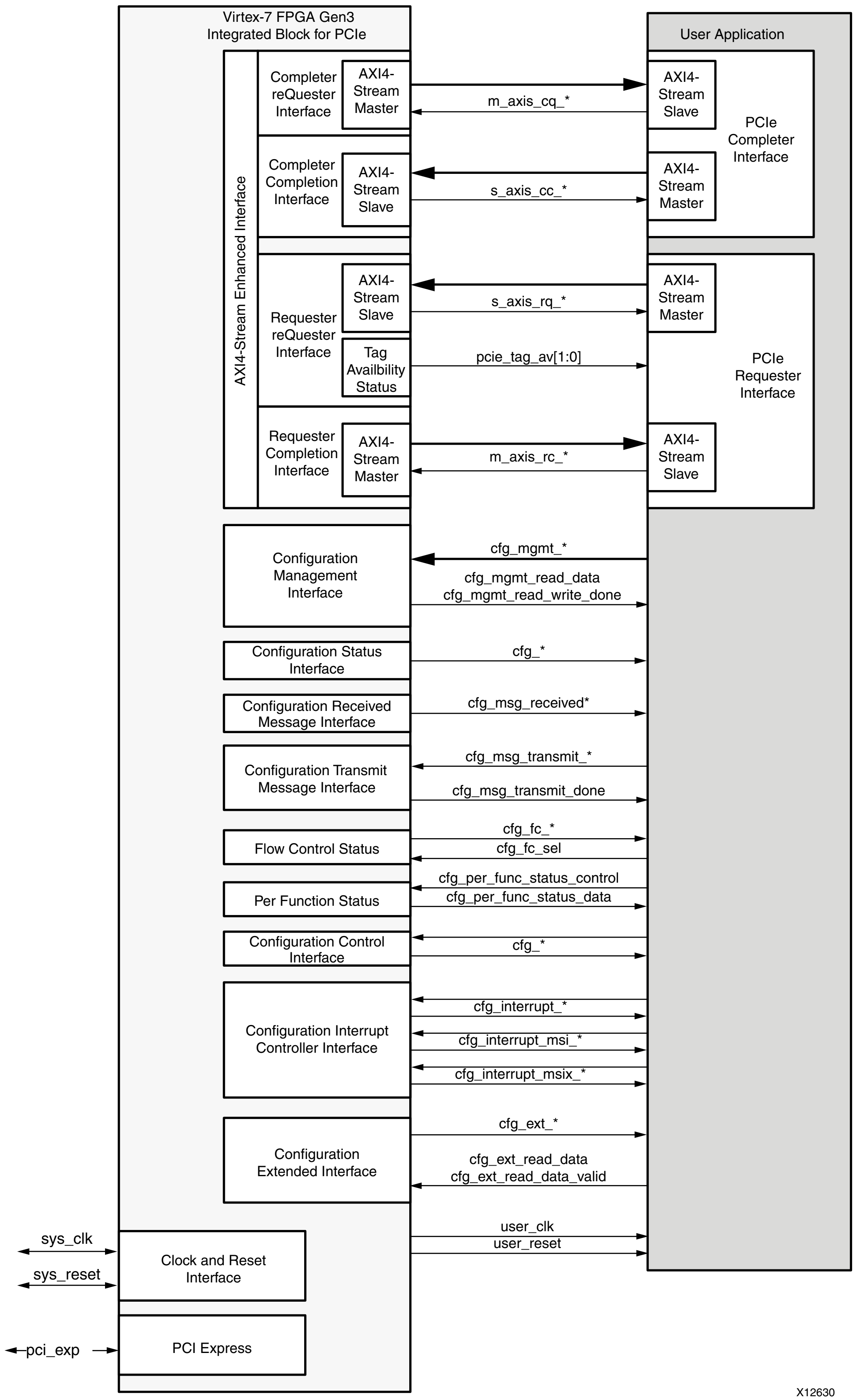

Figure 1-1: **UltraScale FPGAs Gen3 Integrated Block for PCIe Interfaces**

Feature Summary

The core is a high-bandwidth, scalable, and flexible general-purpose I/O core for use with most UltraScale devices. The GTH transceivers in the Integrated Block for PCI Express (PCIe®) solution support 1-lane, 2-lane, 4-lane, and 8-lane operation, running at 2.5 GT/s (Gen1), 5.0 GT/s (Gen2), and 8.0 GT/s (Gen3) line speeds. Endpoint configurations are supported.

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

The key features of the core are:

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

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

Applications

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

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

Unsupported Features

The integrated block does not implement the Address Translation Service, but allows its implementation in external soft logic.

Switch ports and the Resizable BAR Extended Capability are not supported.

Licensing and Ordering Information

The LogiCORE IP UltraScale FPGAs Gen3 Integrated Block for PCIe core is provided at no additional cost with the Vivado Design Suite under the terms of the Xilinx End User License. Information about this and other Xilinx® LogiCORE IP modules is available at the Xilinx Intellectual Property page. For information about pricing and availability of other Xilinx LogiCORE IP modules and tools, contact your local Xilinx sales representative.

Product Specification

Standards Compliance

The UltraScale FPGAs Gen3 Integrated Block for PCIe solution is compatible with industry-standard application form factors such as the PCI Express Card Electromechanical (CEM) v3.0 and the PCI Industrial Computer Manufacturers Group (PICMG) 3.4 specifications [Ref 2].

Resource Utilization

Resources required for the UltraScale FPGAs Gen3 Integrated Block for PCIe core have been estimated for the Kintex® UltraScale™ devices (Table 2-1). These values were generated using the Vivado® Design Suite. The resources listed in Table 2-1 are for the default core configuration.

Table 2-1: **Resource Estimates**

<table>
<tr><th rowspan="2">Lanes</th><th rowspan="2">GTHE3</th><th rowspan="2">FF[1]</th><th rowspan="2">LUT[1]</th><th rowspan="2">CMPS[2]</th><th rowspan="2">RX Completion Buffer Size (KB)</th><th rowspan="2">RX Request Buffer Size (KB)</th><th rowspan="2">TX Replay Buffer Size (KB)</th><th colspan="2">Block RAM Usage</th></tr>
<tr><th>RAMB18</th><th>RAMB36</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">566</td><td rowspan="2">832</td><td rowspan="8">128-1024</td><td rowspan="2">16</td><td rowspan="8">8</td><td rowspan="8">8</td><td>8</td><td rowspan="8">3</td></tr>
<tr><td>12</td></tr>
<tr><td rowspan="2">2</td><td rowspan="2">2</td><td rowspan="2">957</td><td rowspan="2">1384</td><td rowspan="2">16</td><td>8</td></tr>
<tr><td>12</td></tr>
<tr><td rowspan="2">4</td><td rowspan="2">4</td><td rowspan="2">1740</td><td rowspan="2">2500</td><td rowspan="2">16</td><td>8</td></tr>
<tr><td>12</td></tr>
<tr><td rowspan="2">8</td><td rowspan="2">8</td><td rowspan="2">3399</td><td rowspan="2">4818</td><td rowspan="2">16</td><td>8</td></tr>
<tr><td>12</td></tr>
</table>

Notes:

1. Numbers are for the default core configuration. Actual LUT and FF utilization values vary based on specific configurations.
2. Capability Maximum Payload Size (CMPS).

Block Selection

Table 2-2 lists the Integrated Block for PCI Express available for use in FPGAs containing multiple integrated blocks. In some cases, not all integrated blocks can be used due to lack of bonded transceiver sites adjacent to the integrated block.

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

Device Selection		Integrated Block for PCI Express Location			
Device	**Package**	**X0Y0**	**X0Y1**	**X0Y2**	**X0Y3**
XCKU035 ES1	FBVA676 FBVA900 FFVA1156	Yes	Yes		
XCKU040ES1	FBVA676 FBVA900	Yes	Yes		
	FFVA1156	Yes	Yes	Yes	
XCKU060ES1	FFVA1156 FVA1517	Yes	Yes		
XCKU075	FFVA1156	Yes	Yes	Yes	
	FFVA1517 FFVA1760	Yes	Yes	Yes	Yes
XCVU065	FFVC1517	Yes	Yes		
XCVU095	FFVC1517	Yes	Yes	Yes	
	FFVA1760 FFVD1924 FFVJ1924 FFVE1924	Yes	Yes	Yes	Yes

Core Pinouts

The recommended core pinouts for the available UltraScale-based architecture FPGA part and package combinations are available in Appendix B, Core Pinouts. The Vivado Design Suite provides an XDC for the selected part and package that matches the contents of the tables.

For the complete lists of core pinouts, see:

- Kintex UltraScale Device Core Pinouts
- Virtex UltraScale Device Core Pinouts

Port Descriptions

This section provides detailed port descriptions for the following interfaces:

- AXI4-Stream Core Interfaces
- Other Core Interfaces

AXI4-Stream Core Interfaces

In addition to status and control interfaces, the core has the following four required AXI4-Stream interfaces used to transfer and receive transactions:

- Completer reQuest (CQ) Interface: The interface through which all received requests from the link are delivered to the user application.
- Completer Completion (CC) Interface: The interface 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, it can continue to accept new requests on the Requester Completion interface while sending a completion for a request.
- Requester reQuest (RQ) Interface: The interface through which the user application generates requests to remote PCIe® devices.
- Requester Completion (RC) Interface: The interface through which the completions received from the link in response to your requests are presented to the user application.

Completer reQuest (CQ) Interface

Table 2-3 defines the ports in the CQ interface of the core. In the Width column, DW denotes the configured data bus width (64, 128, or 256 bits).

Table 2-3: **CQ Interface Port Descriptions**

Port	Direction	Width	Description
m_axis_cq_tdata	Output	DW/32	Transmit Data from the Completer reQuest Interface. Only the lower 128 bits are to be used when the interface width is 128 bits, and only the lower 64 bits are to be used when the interface width is 64 bits. Bits [255:128] are set permanently to 0 by the core when the interface width is configured as 128 bits, and bits [255:64] are set permanently to 0 when the interface width is configured as 64 bits.
m_axis_cq_tuser	Output	85	Completer reQuest User Data. This set of signals contains sideband information for the TLP being transferred. These signals are valid when `m_axis_cq_tvalid` is High. Table 2-4, page 15 describes the individual signals in this set.
m_axis_cq_tlast	Output	1	TLAST indication for Completer reQuest Data. The core asserts this signal in the last beat of a packet to indicate the end of the packet. When a TLP is transferred in a single beat, the core sets this signal in the first beat of the transfer.
m_axis_cq_tkeep	Output	DW/32	TKEEP indication for Completer reQuest 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 (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.
m_axis_cq_tvalid	Output	1	Completer reQuest Data Valid. The core asserts this output whenever it is driving valid data on the `m_axis_cq_tdata` bus. The core keeps the valid signal asserted during the transfer of a packet. The user application can pace the data transfer using the `m_axis_cq_tready` signal.
m_axis_cq_tready	Input	22	Completer reQuest Data Ready. Activation of this signal by the user logic indicates to the core that the user application is ready to accept data. Data is transferred across the interface when both `m_axis_cq_tvalid` and `m_axis_cq_tready` are asserted in the same cycle. If the user application deasserts the ready signal when `m_axis_cq_tvalid` is High, the core maintains the data on the bus and keeps the valid signal asserted until the user application has asserted the ready signal.

Table 2-3: **CQ Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
pcie_cq_np_req	Input	1	Completer reQuest Non-Posted Request. This input is used by the user application to request the delivery of a Non-Posted request. The core implements a credit-based flow control mechanism to control the delivery of Non-Posted requests across the interface, without blocking Posted TLPs. This input to the core controls an internal credit count. The credit count is incremented in each clock cycle when `pcie_cq_np_req` is High, and decremented on the delivery of each Non-Posted request across the interface. The core temporarily stops delivering Non-Posted requests to the user application when the credit count is zero. It continues to deliver any Posted TLPs received from the link even when the delivery of Non-Posted requests has been paused. The user application can either provide a one-cycle pulse on `pcie_cq_np_req` each time it is ready to receive a Non-Posted request, or can keep it High permanently if it does not need to exercise selective backpressure on Non-Posted requests. The assertion and deassertion of the `pcie_cq_np_req` signal does not need to be aligned with the packet transfers on the completer request interface. There is a minimum of five user_clk from the presentation of completion on `m_axis_rc_tuser` and the reuse of the tag that was returned on the completion.
pcie_cq_np_req_count	Output	6	Completer reQuest Non-Posted Request Count. This output provides the current value of the credit count maintained by the core for delivery of Non-Posted requests to the user application. The core delivers a Non-Posted request across the completer request interface only when this credit count is non-zero. This counter saturates at a maximum limit of 32. Because of internal pipeline delays, there can be several cycles of delay between the core receiving a pulse on the `pcie_cq_np_req` input and updating the `pcie_cq_np_req_count` output in response.

Table 2-4: **Sideband Signal Descriptions in m_axis_cq_tuser**

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

Table 2-4: **Sideband Signal Descriptions in m_axis_cq_tuser *(Cont'd)***

Bit Index	Name	Width	Description
42	tph_present	1	This bit indicates the presence of a Transaction Processing Hint (TPH) in the request TLP being delivered across the interface. This bit is valid when `sop` and `m_axis_cq_tvalid` are both High.
44:43	tph_type[1:0]	2	When a TPH is present in the request TLP, these two bits provide the value of the PH[1:0] field associated with the hint. These bits are valid when `sop` and `m_axis_cq_tvalid` are both High.
52:45	tph_st_tag[7:0]	8	When a TPH is present in the request TLP, this output provides the 8-bit Steering Tag associated with the hint. These bits are valid when `sop` and `m_axis_cq_tvalid` are both High.
84:53	parity	32	Odd parity for the 256-bit transmit data. Bit *i* provides the odd parity computed for byte *i* of `m_axis_cq_tdata`. Only the lower 16 bits are to be used when the interface width is 128 bits, and only the lower 8 bits are to be used when the interface width is 64 bits. Bits [31:16] are set permanently to 0 by the core when the interface width is configured as 128 bits, and bits [31:8] are set permanently to 0 when the interface width is configured as 64 bits.

Completer Completion (CC) Interface

Table 2-5 defines the ports in the CC interface of the core. In the Width column, DW denotes the configured data bus width (64, 128, or 256 bits).

Table 2-5: **CC Interface Port Descriptions**

Port	Direction	Width	Description
s_axis_cc_tdata	Input	DW	Completer Completion Data bus. Completion data from the user application to the core. Only the lower 128 bits are to be used when the interface width is 128 bits, and only the lower 64 bits are to be used when the interface width is 64 bits.
s_axis_cc_tuser	Input	33	Completer Completion User Data. This set of signals contain sideband information for the TLP being transferred. These signals are valid when `s_axis_cc_tvalid` is High. Table 2-6, page 17 describes the individual signals in this set.
s_axis_cc_tlast	Input	1	TLAST indication for Completer Completion Data. The user application must assert this signal in the last cycle of a packet to indicate the end of the packet. When the TLP is transferred in a single beat, the user applicationmust set this bit in the first cycle of the transfer.

Table 2-5: **CC Interface Port Descriptions *(Cont'd)***

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

Table 2-6: **Sideband Signal Descriptions in s_axis_cc_tuser**

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

Table 2-6: **Sideband Signal Descriptions in s_axis_cc_tuser *(Cont'd)***

Bit Index	Name	Width	Description
32:1	parity	32	Odd parity for the 256-bit data. When parity checking is enabled in the core, user logic must set bit *i* of this bus to the odd parity computed for byte *i* of `s_axis_cc_tdata`. Only the lower 16 bits are to be used when the interface width is 128 bits, and only the lower 8 bits are to be used when the interface width is 64 bits. 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 (RQ) Interface

Table 2-7 defines the ports in the RQ interface of the core. In the Width column, DW denotes the configured data bus width (64, 128, or 256 bits).

Table 2-7: **RQ Interface Port Descriptions**

Port	Direction	Width	Description
s_axis_rq_tdata	Input	DW	Requester reQuest Data bus. This input contains the requester-side request data from the user application to the core. Only the lower 128 bits are to be used when the interface width is 128 bits, and only the lower 64 bits are to be used when the interface width is 64 bits.
s_axis_rq_tuser	Input	60	Requester reQuest User Data. This set of signals contains sideband information for the TLP being transferred. These signals are valid when `s_axis_rq_tvalid` is High. Table 2-8, page 20 describes the individual signals in this set.
s_axis_rq_tlast	Input	1	TLAST Indication for Requester reQuest Data. The user application must assert this signal in the last cycle of a TLP to indicate the end of the packet. When the TLP is transferred in a single beat, the user application must set this bit in the first cycle of the transfer.
s_axis_rq_tkeep	Input	DW/32	TKEEP Indication for Requester reQuest Data. The assertion of bit *i* of this bus during a transfer indicates to the core that Dword *i* of the `s_axis_rq_tdata` bus contains valid data. The user application must set this bit to 1 contiguously for all Dwords, starting from the first Dword of the descriptor to the last Dword of the payload. Thus, `s_axis_rq_tdata` must be set to all 1s in all beats of a packet, except in the final beat when the total size of the packet is not a multiple of the width of the data bus (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.

Table 2-7: **RQ Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
s_axis_rq_tready	Output	4	Requester reQuest Data Ready. Activation of this signal by the core indicates that it is ready to accept data. Data is transferred across the interface when both `s_axis_rq_tvalid` and `s_axis_rq_tready` are asserted in the same cycle. If the core deasserts the `ready` signal when the `valid` signal is High, the user application must maintain the data on the bus and keep the valid signal asserted until the core has asserted the ready signal. You can assign all 4 bits to 1 or 0.
s_axis_rq_tvalid	Input	1	Requester reQuest Data Valid. The user application must assert this output whenever it is driving valid data on the `s_axis_rq_tdata` bus. The user application must keep the valid signal asserted during the transfer of a packet. The core paces the data transfer using the `s_axis_rq_tready` signal.
pcie_rq_seq_num	Output	4	Requester reQuest TLP transmit sequence number. You can optionally use this output to track the progress of the request in the core transmit pipeline. To use this feature, provide a sequence number for each request on the `seq_num[3:0]` bus. The core outputs this sequence number on the `pcie_rq_seq_num[3:0]` output when the request TLP has reached a point in the pipeline where a Completion TLP from the user application cannot pass it. This mechanism enables you to maintain ordering between Completions sent to the completer completion interface of the core and Posted requests sent to the requester request interface. Data on the `pcie_rq_seq_num[3:0]` output is valid when `pcie_rq_seq_num_vld` is High.
pcie_rq_seq_num_vld	Output	1	Requester reQuest TLP transmit sequence number valid. This output is asserted by the core for one cycle when it has placed valid data on `pcie_rq_seq_num[3:0]`.

Table 2-7: **RQ Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
pcie_rq_tag	Output	6	Requester reQuest Non-Posted tag. When tag management for Non-Posted requests is performed by the core (AXISTEN_IF_ENABLE_CLIENT_TAG is 0), this output is used by the core to communicate the allocated tag for each Non-Posted request received. The tag value on this bus is valid for one cycle when `pcie_rq_tag_vld` is High. You must copy this tag and use it to associate the completion data with the pending request. There can be a delay of several cycles between the transfer of the request on the `s_axis_rq_tdata` bus and the assertion of `pcie_rq_tag_vld` by the core to provide the allocated tag for the request. Meanwhile, the user application can continue to send new requests. The tags for requests are communicated on this bus in FIFO order, so the user application can easily associate the tag value with the request it transferred.
pcie_rq_tag_vld	Output	1	Requester reQuest Non-Posted tag valid. The core asserts this output for one cycle when it has allocated a tag to an incoming Non-Posted request from the requester request interface and placed it on the `pcie_rq_tag` output.

Table 2-8: **Sideband Signal Descriptions in s_axis_rq_tuser**

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

Table 2-8: **Sideband Signal Descriptions in s_axis_rq_tuser *(Cont'd)***

Bit Index	Name	Width	Description
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 desires 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. 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_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. When the core is configured as an Endpoint, this error is also reported by the core to the Root Complex to which it is attached, using Advanced Error Reporting (AER).
12	tph_present	1	This bit indicates the presence of a Transaction Processing Hint (TPH) in the request TLP being delivered across the interface. The core samples this field in the first beat of a packet, when `s_axis_rq_tvalid` and `s_axis_rq_tready` are both High. This bit must be permanently tied to 0 if the TPH capability is not in use.
14:13	tph_type[1:0]	2	When a TPH is present in the request TLP, these two bits provide the value of the PH[1:0] field associated with the hint. The core samples this field in the first beat of a packet, when `s_axis_rq_tvalid` and `s_axis_rq_tready` are both High. These bits can be set to any value if tph_present is set to 0.
15	tph_indirect_tag_en	1	When this bit is set, the core uses the lower bits of `tph_st_tag[7:0]` as an index into its Steering Tag Table, and insert the tag from this location in the transmitted request TLP. When this bit is 0, the core uses the value on `tph_st_tag[7:0]` directly as the Steering Tag. The core samples this bit in the first beat of a packet, when `s_axis_rq_tvalid` and `s_axis_rq_tready` are both High. This bit can be set to any value if `tph_present` is set to 0.
23:16	tph_st_tag[7:0]	8	When a TPH is present in the request TLP, this output provides the 8-bit Steering Tag associated with the hint. The core samples this field in the first beat of a packet, when `s_axis_rq_tvalid` and `s_axis_rq_tready` are both High. These bits can be set to any value if `tph_present` is set to 0.

Table 2-8: **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 to be used when the interface width is 128 bits, and only the lower 8 bits are to be used when the interface width is 64 bits. On detection of a parity error, the core nullifies the corresponding TLP on the link and reports it as an Uncorrectable Internal Error. These bits can be set to 0 if parity checking is disabled in the core.

Requester Completion (RC) Interface

Table 2-9 defines the ports in the RC interface of the core. In the Width column, DW denotes the configured data bus width (64, 128, or 256 bits).

Table 2-9: **RC Interface Port Descriptions**

Port	Direction	Width	Description
m_axis_rc_tdata	Output	DW	Requester Completion Data bus. Transmit data from the Core requester completion interface to the user application. Only the lower 128 bits are used when the interface width is 128 bits, and only the lower 64 bits are used when the interface width is 64 bits. Bits [255:128] are set permanently to 0 by the core when the interface width is configured as 128 bits, and bits [255:64] are set permanently to 0 when the interface width is configured as 64 bits.
m_axis_rc_tuser	Output	75	Requester Completion User Data. This set of signals contains sideband information for the TLP being transferred. These signals are valid when `m_axis_rc_tvalid` is High. Table 2-10, page 23 describes the individual signals in this set.
m_axis_rc_tlast	Output	1	TLAST indication for Requester Completion Data. The core asserts this signal in the last beat of a packet to indicate the end of the packet. When a TLP is transferred in a single beat, the core sets this bit in the first beat of the transfer. This output is used only when the straddle option is disabled. When the straddle option is enabled (for 256-bit interface), the core sets this output permanently to 0.

Table 2-9: **RC Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
m_axis_rc_tkeep	Output	DW/32	TKEEP indication for Requester Completion Data. The assertion of bit *i* of this bus during a transfer indicates that Dword *i* of the `m_axis_rc_tdata` bus contains valid data. The core sets this bit to 1 contiguously for all Dwords starting from the first Dword of the descriptor to the last Dword of the payload. Thus, `m_axis_rc_tkeep` sets to 1s in all beats of a packet, except in the final beat when the total size of the packet is not a multiple of the width of the data bus (both in Dwords). This is true for both Dword-aligned and address-aligned modes of payload transfer. Bits [7:4] of this bus are set permanently to 0 by the core when the interface width is configured as 128 bits, and bits [7:2] are set permanently to 0 when the interface width is configured as 64 bits. These outputs are permanently set to all 1s when the interface width is 256 bits and the straddle option is enabled. The user logic must use the signals in `m_axis_rc_tuser` in that case to determine the start and end of Completion TLPs transferred over the interface.
m_axis_rc_tvalid	Output	1	Requester Completion Data Valid. The core asserts this output whenever it is driving valid data on the `m_axis_rc_tdata` bus. The core keeps the valid signal asserted during the transfer of a packet. The user application can pace the data transfer using the `m_axis_rc_tready` signal.
m_axis_rc_tready	Input	22	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. You can assign all 22 bits to 1 or 0.

Table 2-10: **Sideband Signal Descriptions in m_axis_rc_tuser**

Bit Index	Name	Width	Description
31:0	byte_en	32	The user logic can optionally use these byte enable bits to determine the valid bytes in the payload of a packet being transferred. The assertion of bit *i* of this bus during a transfer indicates that byte *i* of the `m_axis_rc_tdata` bus contains a valid payload byte. This bit is not asserted for descriptor bytes. Although the byte enables can be generated by user logic from information in the request descriptor (address and length), the logic has the option to use these signals directly instead of generating them from other interface signals. The 1 bit in this bus for the payload of a TLP is always contiguous. Bits [31:16] of this bus are set permanently to 0 by the core when the interface width is configured as 128 bits, and bits [31:8] are set permanently to 0 when the interface width is configured as 64 bits.

Table 2-10: **Sideband Signal Descriptions in m_axis_rc_tuser *(Cont'd)***

Bit Index	Name	Width	Description
32	is_sof_0	1	Start of a first Completion TLP. For 64-bit and 128-bit interfaces, and for the 256-bit interface with no straddling, `is_sof_0` is asserted by the core in the first beat of a packet to indicate the start of the TLP. On these interfaces, only a single TLP can be started in a data beat, and `is_sof_1` is permanently set to 0. Use of this signal is optional when the straddle option is not enabled. When the interface width is 256 bits and the straddle option is enabled, the core can straddle two Completion TLPs in the same beat. In this case, the Completion TLPs are not formatted as AXI4-Stream packets. The assertion of `is_sof_0` indicates a Completion TLP starting in the beat. The first byte of this Completion TLP is in byte lane 0 if the previous TLP ended before this beat, or in byte lane 16 if the previous TLP continues in this beat.
33	is_sof_1	1	Start of a second Completion TLP. This signal is used when the interface width is 256 bits and the straddle option is enabled, when the core can straddle two Completion TLPs in the same beat. The output is permanently set to 0 in all other cases. The assertion of `is_sof_1` indicates a second Completion TLP starting in the beat, with its first bye in byte lane 16. The core starts a second TLP at byte position 16 only if the previous TLP ended in one of the byte positions 0-15 in the same beat; that is, only if `is_eof_0[0]` is also set in the same beat.
37:34	is_eof_0[3:0]	4	End of a first Completion TLP and the offset of its last Dword. These outputs are used only when the interface width is 256 bits and the straddle option is enabled. The assertion of the bit `is_eof_0[0]` indicates the end of a first Completion TLP in the current beat. When this bit is set, the bits `is_eof_0[3:1]` provide the offset of the last Dword of this TLP.
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.

Table 2-10: **Sideband Signal Descriptions in m_axis_rc_tuser *(Cont'd)***

Bit Index	Name	Width	Description
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.

Other Core Interfaces

The core also provides the interfaces described in this section.

Transmit Flow Control Interface

The Transmit Flow Control interface is used by the user application to request which flow control information the core provides. This interface provides the Posted/Non-Posted Header Flow Control Credits, Posted/Non-Posted Data Flow Control Credits, the Completion Header Flow Control Credits, and the Completion Data Flow Control Credits to the user application based upon the setting flow control select input to the core.

Table 2-11 defines the ports in the Transmit Flow Control interface of the core.

Table 2-11: **Transmit Flow Control Interface Port Descriptions**

Port	Direction	Width	Description
pcie_tfc_nph_av	Output	2	Transmit flow control Non-Posted header credits available. This output indicates the currently available header credit for Non-Posted TLPs on the transmit side of the core. The user logic can check this output before transmitting a Non-Posted request on the requester request interface, to avoid blocking the interface when no credit is available. The encodings are: • 00: No credits available • 01: 1 credit available • 10: 2 credits available • 11: 3 or more credits available Because of pipeline delays, the value on this output does not include the credit consumed by the Non-Posted requests in the last two clock cycles. The user logic must adjust the value on this output by the credit consumed by the Non-Posted requests it sent in the previous two clock cycles, if any.
pcie_tfc_npd_av	Output	2	Transmit flow control Non-Posted data credits available. This output indicates the currently available payload credit for Non-Posted TLPs on the transmit side of the core. The user logic can check this output before transmitting a Non-Posted request on the requester request interface, to avoid blocking the interface when no credit is available. The encodings are: • 00: No credits available • 01: 1 credit available • 10: 2 credits available • 11: 3 or more credits available Because of pipeline delays, the value on this output does not include the credit consumed by the Non-Posted requests sent by the user application in the last two clock cycles. The user logic must adjust the value on this output by the credit consumed by the Non-Posted requests it sent in the previous two clock cycles, if any.

Configuration Management Interface

The Configuration Management interface is used to read and write to the Configuration Space Registers. Table 2-12 defines the ports in the Configuration Management interface of the core.

Table 2-12: **Configuration Management Interface Port Descriptions**

Port	Direction	Width	Description
cfg_mgmt_addr	Input	19	Read/Write Address. Address is in the Configuration and Management register space, and is Dword aligned. For accesses from the local management bus: for the address bits `cfg_mgmt_addr[17:10]`, select the PCI Function associated with the configuration register; and for the bits `cfg_mgmt_addr[9:0]`, select the register within the Function. The address bit `cfg_mgmt_addr[18]` must be set to zero (0) when accessing the PCI or PCI Express configuration registers, and to one (1) when accessing the local management registers.
cfg_mgmt_write	Input	1	Write Enable. Asserted for a write operation. Active-High.
cfg_mgmt_write_data	Input	32	Write data. Write data is used to configure the Configuration and Management registers.
cfg_mgmt_byte_enable	Input	4	Byte Enable. Byte enable for write data, where `cfg_mgmt_byte_enable[0]` corresponds to `cfg_mgmt_write_data[7:0]`, and so on.
cfg_mgmt_read	Input	1	Read Enable. Asserted for a read operation. Active-High.
cfg_mgmt_read_data	Output	32	Read data out. Read data provides the configuration of the Configuration and Management registers.
cfg_mgmt_read_write_done	Output	1	Read/Write operation complete. Asserted for 1 cycle when operation is complete. Active-High.
cfg_mgmt_type1_cfg_reg_access	Input	1	Type 1 RO, Write. When the core is configured in the Root Port mode, asserting this input during a write to a Type-1 PCI™ Config Register forces a write into certain read-only fields of the register (see description of RC-mode Config registers). This input has no effect when the core is in the Endpoint mode, or when writing to any register other than a Type-1 Config Register.

Configuration Status Interface

The Configuration Status interface provides information on how the core is configured, such as the negotiated link width and speed, the power state of the core, and configuration errors. Table 2-13 defines the ports in the Configuration Status interface of the core.

Table 2-13: **Configuration Status Interface Port Descriptions**

Port	Direction	Width	Description
cfg_phy_link_down	Output	1	Configuration Link Down. Status of the PCI Express link based on Physical Layer LTSSM. • 1b: Link is Down (LinkUp state variable is 0b) • 0b: Link is Up (LinkUp state variable is `1b`) ***Note:*** Per the *PCI Express Base Specification, rev. 3.0* [Ref 2], LinkUp is `1b` in the Recovery, L0, L0s, L1, and L2 cfg_ltssm states. In the Configuration state, LinkUp can be `0b` or `1b`. It is always `0b` when the Configuration state is reached using **Detect** > **Polling** > **Configuration**. LinkUp is `1b` if the configuration state is reached through any other state transition. ***Note:*** While reset is asserted, the output of this signal are `0b` until reset is released.
cfg_phy_link_status	Output	2	Configuration Link Status. Status of the PCI Express link. • 00b: No receivers detected • 01b: Link training in progress • 10b: Link up, DL initialization in progress • 11b: Link up, DL initialization completed
cfg_negotiated_width	Output	4	Configuration Link Status. Negotiated Link Width: PCI Express Link Status Register, Negotiated Link Width field. This field 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).
cfg_current_speed	Output	3	Current Link Speed. This signal outputs the current link speed from Link Status register bits 1 down to 0. This field indicates the negotiated Link speed of the given PCI Express Link. • 001b: 2.5 GT/s PCI Express Link • 010b: 5.0 GT/s PCI Express Link • 100b: 8.0 GT/s PCI Express Link
cfg_max_payload	Output	3	Max_Payload_Size. This signal outputs the maximum payload size from Device Control Register bits 7 down to 5. This field sets the maximum TLP payload size. As a Receiver, the logic must handle TLPs as large as the set value. As a Transmitter, the logic must not generate TLPs exceeding the set value. • 000b: 128 bytes maximum payload size • 001b: 256 bytes maximum payload size • 010b: 512 bytes maximum payload size • 011b: 1024 bytes maximum payload size • 100b: 2048 bytes maximum payload size • 101b: 4096 bytes maximum payload size

Table 2-13: **Configuration Status Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_max_read_req	Output	3	Max_Read_Request_Size. This signal outputs the maximum read request size from Device Control register bits 14 down to 12. This field sets the maximum Read Request size for the logic as a Requester. The logic must not generate Read Requests with size exceeding the set value. • 000b: 128 bytes maximum Read Request size • 001b: 256 bytes maximum Read Request size • 010b: 512 bytes maximum Read Request size • 011b: 1024 bytes maximum Read Request size • 100b: 2048 bytes maximum Read Request size • 101b: 4096 bytes maximum Read Request size
cfg_function_status	Output	8	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
cfg_vf_status	Output	12	Configuration Virtual Function Status. These outputs indicate the status of Virtual Functions, two bits each per Virtual Function. • Bit 0: Virtual Function 0: Configured/Enabled by the software • Bit 1: Virtual Function 0: PCI Command Register, Bus Master Enable, etc.
cfg_function_power_state	Output	6	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. The possible power states are: • 000: D0_uninitialized • 001: D0_active • 010: D1 • 100: D3_hot

Table 2-13: **Configuration Status Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_vf_power_state	Output	18	Configuration Virtual Function Power State. These outputs indicate the current power state of the Virtual Functions. Bits [2:0] capture the power state of Virtual Function 0, and bits [5:3] capture that of Virtual Function 1, and so on. The possible power states are: • 000: D0_uninitialized • 001: D0_active • 010: D1 • 100: D3_hot
cfg_link_power_state	Output	2	Current power state of the PCI Express link: • 00: L0 • 01: L0s • 10: L1 • 11: L2/Reserved
cfg_err_cor_out	Output	1	Correctable Error Detected. In Endpoint mode, the core activates this output for one cycle when it has detected a correctable error and its reporting is not masked. In a multi-Function Endpoint, this is the logical OR of the correctable error status bits in the Device Status Registers of all Functions. In Root Port mode, this output is activated on detection of a local correctable error, when its reporting is not masked. This output does not respond to any errors signaled by remote devices using PCI Express error messages. These error messages are delivered through the message interface.
cfg_err_nonfatal_out	Output	1	Non-Fatal Error Detected. In Endpoint mode, the core activates this output for one cycle when it has detected a non fatal error and its reporting is not masked. In a multi-Function Endpoint, this is the logical OR of the non fatal error status bits in the Device Status Registers of all Functions. In Root Port mode, this output is activated on detection of a local non-fatal error, when its reporting is not masked. This output does not respond to any errors signaled by remote devices using PCI Express error messages. These error messages are delivered through the message interface.
cfg_err_fatal_out	Output	1	Fatal Error Detected. In Endpoint mode, the core activates this output for one cycle when it has detected a fatal error and its reporting is not masked. In a multi-Function Endpoint, this is the logical OR of the fatal error status bits in the Device Status Registers of all Functions. In Root Port mode, this output is activated on detection of a local fatal error, when its reporting is not masked. This output does not respond to any errors signaled by remote devices using PCI Express error messages. These error messages are delivered through the message interface.

Table 2-13: **Configuration Status Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_ltr_enable	Output	1	Latency Tolerance Reporting Enable. The state of this output reflects the setting of the LTR Mechanism Enable bit in the Device Control 2 Register of Physical Function 0. When the core is configured as an Endpoint logic uses this output to enable the generation of LTR messages. This output is not to be used when the core is configured as a Root Port.

Table 2-13: **Configuration Status Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_ltssm_state	Output	6	Current LTSSM State. Shows the current LTSSM state: 00: Detect.Quiet 01: Detect.Active 02: Polling.Active 03: Polling.Compliance 04: Polling.Configuration 05: Configuration.Linkwidth.Start 06: Configuration.Linkwidth.Accept 07: Configuration.Lanenum.Accept 08: Configuration.Lanenum.Wait 09: Configuration.Complete 0A: Configuration.Idle 0B: Recovery.RcvrLock 0C: Recovery.Speed 0D: Recovery.RcvrCfg 0E: Recovery.Idle 10: L0 11: Rx_L0s.Entry 12: Rx_L0s.Idle 13: Rx_L0s.FTS 14: Tx_L0s.Entry 15: Tx_L0s.Idle 16: Tx_L0s.FTS 17: L1.Entry 18: L1.Idle 19: L2.Idle 1A: L2.TransmitWake 20: DISABLED 21: LOOPBACK_ENTRY_MASTER 22: LOOPBACK_ACTIVE_MASTER 23: LOOPBACK_EXIT_MASTER 24: LOOPBACK_ENTRY_SLAVE 25: LOOPBACK_ACTIVE_SLAVE 26: LOOPBACK_EXIT_SLAVE 27: HOT_RESET 28: RECOVERY_EQUALIZATION_PHASE0 29: RECOVERY_EQUALIZATION_PHASE1 2A: RECOVERY_EQUALIZATION_PHASE2 2B: RECOVERY_EQUALIZATION_PHASE3

Table 2-13: **Configuration Status Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_rcb_status	Output	2	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 PF 0, and so on. In RC mode, bit 0 indicates the RCB setting of the Link Control Register of the RP, bit 1 is reserved. For each bit, a value of 0 indicates an RCB of 64 bytes and a value of 1 indicates 128 bytes.
cfg_dpa_substate_change	Output	2	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 PF 0 and so on. These outputs are not active in Root Port mode.
cfg_obff_enable	Output	2	Optimized Buffer Flush Fill Enable. This output reflects the setting of the OBFF Enable field in the Device Control 2 Register. • 00: OBFF disabled • 01: OBFF enabled using message signaling, Variation A • 10: OBFF enabled using message signaling, Variation B • 11: OBFF enabled using WAKE# signaling.
cfg_pl_status_change	Output	1	This output is used by the core in Root Port mode to signal one of the following link training-related events: (a) The link bandwidth changed as a result of the change in the link width or operating speed and the change was initiated locally (not by the link partner), without the link going down. This interrupt is enabled by the Link Bandwidth Management Interrupt Enable bit in the Link Control Register. The status of this interrupt can be read from the Link Bandwidth Management Status bit of the Link Status Register; or (b) The link bandwidth changed autonomously as a result of the change in the link width or operating speed and the change was initiated by the remote node. This interrupt is enabled by the Link Autonomous Bandwidth Interrupt Enable bit in the Link Control Register. The status of this interrupt can be read from the Link Autonomous Bandwidth Status bit of the Link Status Register; or (c) The Link Equalization Request bit in the Link Status 2 Register was set by the hardware because it received a link equalization request from the remote node. This interrupt is enabled by the Link Equalization Interrupt Enable bit in the Link Control 3 Register. The status of this interrupt can be read from the Link Equalization Request bit of the Link Status 2 Register. The `pl_interrupt` output is not active when the core is configured as an Endpoint.

Table 2-13: **Configuration Status Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_tph_requester_enable	Output	2	Bit 0 of this output reflect the setting of the TPH Requester Enable bit [8] of the TPH Requester Control Register in the TPH Requester Capability Structure of Physical Function 0. Bit 1 corresponds to Physical Function 1.
cfg_tph_st_mode	Output	6	Bits [2:0] of this output reflect the setting of the ST Mode Select bits in the TPH Requester Control Register of Physical Function 0. Bits [5:3] reflect the setting of the same register field of PF 1.
cfg_vf_tph_requester_enable	Output	6	Each of the six bits of this output reflects the setting of the TPH Requester Enable bit 8 of the TPH Requester Control Register in the TPH Requester Capability Structure of the corresponding Virtual Function.
cfg_vf_tph_st_mode	Output	18	Bits [2:0] of this output reflect the setting of the ST Mode Select bits in the TPH Requester Control Register of Virtual Function 0. Bits [5:3] reflect the setting of the same register field of VF 1, and so on.

Configuration Received Message Interface

The Configuration Received Message interface indicates to the logic that a decodable message from the link, the parameters associated with the data, and the type of message have been received. Table 2-14 defines the ports in the Configuration Received Message interface of the core.

Table 2-14: **Configuration Received Message Interface Port Descriptions**

Port	Direction	Width	Description
cfg_msg_received	Output	1	Configuration Received a Decodable Message. The core asserts this output for one or more consecutive clock cycles when it has received a decodable message from the link. The duration of its assertion is determined by the type of message. The core transfers any parameters associated with the message on the `cfg_msg_data[7:0]` output in one or more cycles when `cfg_msg_received` is High. Table 3-9 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. This output is active only when the AXISTEN_IF_ENABLE_RX_MSG_INTFC attribute is set. The Configuration Received Message interface must be enabled during core configuration in the Vivado IDE.
cfg_msg_received_data	Output	8	This bus is used to transfer any parameters associated with the Received Message. The information it carries in each cycle for various message types is listed in Table 3-9.

Table 2-14: **Configuration Received Message Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_msg_received_type	Output	5	Received message type. When `cfg_msg_received` is High, these five bits indicate the type of message being signaled by the core. The various message types are listed in Table 3-8.

Configuration Transmit Message Interface

The Configuration Transmit Message interface is used by the user application to transmit messages to the core. The user application supplies the transmit message type and data information to the core, which responds with the `Done` signal. Table 2-15 defines the ports in the Configuration Transmit Message interface of the core.

Table 2-15: **Configuration Transmit Message Interface Port Descriptions**

Port	Direction	Width	Description
cfg_msg_transmit	Input	1	Configuration Transmit Encoded Message. This signal is asserted together with `cfg_msg_transmit_type`, which supplies the encoded message type and `cfg_msg_transmit_data`, which supplies optional data associated with the message, until `cfg_msg_transmit_done` is asserted in response.
cfg_msg_transmit_type	Input	3	Configuration Transmit Encoded Message Type. Indicates the type of PCI Express message to be transmitted. Encodings supported are: • 000b: Latency Tolerance Reporting (LTR) • 001b: Optimized Buffer Flush/Fill (OBFF) • 010b: Set Slot Power Limit (SSPL) • 011b: Power Management (PM PME) • 100b -111b: Reserved

Table 2-15: **Configuration Transmit Message Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_msg_transmit_data	Input	32	Configuration Transmit Encoded Message Data. Indicates message data associated with particular message type. • 000b: LTR - `cfg_msg_transmit_data[31]` ≤ Snoop Latency Req., `cfg_msg_transmit_data[28:26]` ≤ Snoop Latency Scale, `cfg_msg_transmit_data[25:16]` ≤ Snoop Latency Value, `cfg_msg_transmit_data[15]` ≤ No-Snoop Latency Requirement, `cfg_msg_transmit_data[12:10]` ≤ No-Snoop Latency Scale, `cfg_msg_transmit_data[9:0]` ≤ No-Snoop Latency Value. • 001b: OBFF - `cfg_msg_transmit_data[3:0]` ≤ OBFF Code. • 010b: SSPL - `cfg_msg_transmit_data[9:0]` ≤ {Slot Power Limit Scale, Slot Power Limit Value}. • 011b: PM_PME - `cfg_msg_transmit_data[1:0]` ≤ PF1, PF0; `cfg_msg_transmit_data[9:4]` ≤ VF5, VF4, VF3, VF2, VF1, VF0, where one or more PFs or VFs can signal PM_PME simultaneously. • 100b - 111b: Reserved
cfg_msg_transmit_done	Output	1	Configuration Transmit Encoded Message Done. Asserted in response to `cfg_mg_transmit` assertion, for 1 cycle after the request is complete.

Configuration Flow Control Interface

Table 2-16 defines the ports in the Configuration Flow Control interface of the core.

Table 2-16: **Configuration Flow Control Interface Port Descriptions**

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

Table 2-16: **Configuration Flow Control Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_fc_npd	Output	12	Non-Posted Data Flow Control Credits. This output provides the number of Non-Posted Data Flow Control Credits. This multiplexed output can be used to bring out various flow control parameters and variables related to Non-Posted Data Credit maintained by the core. The flow control information to bring out on this core is selected by the `cfg_fc_sel[2:0]` input.
cfg_fc_cplh	Output	8	Completion Header Flow Control Credits. This output provides the number of Completion Header Flow Control Credits. This multiplexed output can be used to bring out various flow control parameters and variables related to Completion Header Credit maintained by the core. The flow control information to bring out on this core is selected by the `cfg_fc_sel[2:0]` input.
cfg_fc_cpld	Output	12	Completion Data Flow Control Credits. This output provides the number of Completion Data Flow Control Credits. This multiplexed output can be used to bring out various flow control parameters and variables related to Completion Data Credit maintained by the core. The flow control information to bring out on this core is selected by the `cfg_fc_sel[2:0]`.
cfg_fc_sel	Input	3	Flow Control Informational Select. These inputs select the type of flow control to bring out on the `cfg_fc_*` outputs of the core. The various flow control parameters and variables that can be accessed for the different settings of these inputs are: • 000: Receive credits currently available to the link partner • 001: Receive credit limit • 010: Receive credits consumed • 011: Available space in receive buffer • 100: Transmit credits available • 101: Transmit credit limit • 110: Transmit credits consumed • 111: Reserved This value represents the actual unused credits in the receiver FIFO, and the recommendation is to use it only as an approximate indication of receiver FIFO fullness, relative to the initial credit limit value advertized, such as, ¼ full, ½ full, ¾ full, full. ***Note:*** Infinite credit for transmit credits available (`cfg_fc_sel == 3'b100`) is signaled as `8'h80`, `12'h800` for header and data credits, respectively. For all other `cfg_fc_sel` selections, infinite credit is signaled as `8'h00`, `12'h000`, respectively, for header and data categories.

Per Function Status Interface

The Function Status interface provides status data as requested by the user application through the selected function. Table 2-17 and Table 2-18 define the ports in the Function Status interface of the core.

Table 2-17: **Overview of Function Status Interface Port Descriptions**

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

Table 2-18: **Detailed Function Status Interface Port Descriptions**

cfg_per_func_ status_control [bit]	cfg_per_func_ status_data [bit/slice]	Status Output	Width	Description
0	0	cfg_command_ io_enable	1	Configuration Command - I/O Space Enable: Command[0]. Endpoints: If 1, allows the device to receive I/O Space accesses. Otherwise, the core filters these and respond with an Unsupported Request. Root/Switch: Core takes no action based on this setting. If 0, logic must not generate TLPs downstream.
0	1	cfg_command_ mem_enable	1	Configuration Command - Memory Space Enable: Command[1]. Endpoints: If 1, allows the device to receive Memory Space accesses. Otherwise, the core filters these and respond with an Unsupported Request. Root/Switch: Core takes no action based on this setting. If 0, logic must not generate TLPs downstream.
0	2	cfg_command_ bus_master_enable	1	Configuration Command - Bus Master Enable: Command[2]. The core takes no action based on this setting; logic must do that. Endpoints: When asserted, the logic is allowed to issue Memory or I/O Requests (including MSI/X interrupts); otherwise it must not. Root and Switch Ports: When asserted, received Memory or I/O Requests might be forwarded upstream; otherwise they are handled as Unsupported Requests (UR), and for Non-Posted Requests a Completion with UR completion status is returned.

Table 2-18: **Detailed Function Status Interface Port Descriptions *(Cont'd)***

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
0	3	cfg_command_interrupt_disable	1	Configuration Command - Interrupt Disable: Command[10]. When asserted, the core is prevented from asserting INTx interrupts.
0	4	cfg_command_serr_en	1	Configuration Command - SERR Enable: Command[8]. When asserted, this bit enables reporting of Non-fatal and Fatal errors. Note that errors are reported if enabled either through this bit or through the PCI Express specific bits in the Device Control register. In addition, for a Root Complex or Switch, this bit controls transmission by the primary interface of ERR_NONFATAL and ERR_FATAL error messages forwarded from the secondary interface.
0	5	cfg_bridge_serr_en	1	Configuration Bridge Control - SERR Enable: Bridge Ctrl[1]. When asserted, this bit enables the forwarding of Correctable, Non-fatal and Fatal errors (you must enforce that).
0	6	cfg_aer_ecrc_check_en	1	Configuration AER - ECRC Check Enable: AER_Cap_and_Ctl[8]. When asserted, this bit indicates that ECRC checking has been enabled by the host.
0	7	cfg_aer_ecrc_gen_en	1	Configuration AER - ECRC Generation Enable: AER_Cap_and_Ctl[6]. When asserted, this bit indicates that ECRC generation has been enabled by the host.
0	15:8	0	8	Reserved
1	0	cfg_dev_status_corr_err_detected	1	Configuration Device Status - Correctable Error Detected: Device_Status[0]. Indicates status of correctable errors detected. Errors are logged in this register regardless of whether error reporting is enabled or not in the Device Control register.
1	1	cfg_dev_status_non_fatal_err_detected	1	Configuration Device Status - Non-Fatal Error Detected: Device_Status[1]. Indicates status of Nonfatal errors detected. Errors are logged in this register regardless of whether error reporting is enabled or not in the Device Control register.

Table 2-18: **Detailed Function Status Interface Port Descriptions *(Cont'd)***

cfg_per_func_ status_control [bit]	cfg_per_func_ status_data [bit/slice]	Status Output	Width	Description
1	2	cfg_dev_status_ fatal_err_detected	1	Configuration Device Status - Fatal Error Detected: Device_Status[2]. Indicates status of Fatal errors detected. Errors are logged in this register regardless of whether error reporting is enabled or not in the Device Control register.
1	3	cfg_dev_status_ur_ detected	1	Configuration Device Status - Unsupported Request Detected: Device_Status[3]. Indicates that the core received an Unsupported Request. Errors are logged in this register regardless of whether error reporting is enabled or not in the Device Control register.
1	4	cfg_dev_control_ corr_err_reporting_ en	1	Configuration Device Control - Correctable Error Reporting Enable: Device_Ctrl[0]. This bit, in conjunction with other bits, controls sending ERR_COR Messages. For a Root Port, the reporting of correctable errors is internal to the root; no external ERR_COR Message is generated.
1	5	cfg_dev_control_ non_fatal_ reporting_en	1	Configuration Device Control - Non-Fatal Error Reporting Enable: Device_Ctrl[1]. This bit, in conjunction with other bits, controls sending ERR_NONFATAL Messages. For a Root Port, the reporting of correctable errors is internal to the root; no external ERR_NONFATAL Message is generated.
1	6	cfg_dev_control_ fatal_err_reporting_ en	1	Configuration Device Control - Fatal Error Reporting Enable: Device_Ctrl[2]. This bit, in conjunction with other bits, controls sending ERR_FATAL Messages. For a Root Port, the reporting of correctable errors is internal to the root; no external ERR_FATAL Message is generated.
1	7	cfg_dev_control_ ur_err_reporting_en	1	Configuration Device Control - UR Reporting Enable: Device_Ctrl[3]. This bit, in conjunction with other bits, controls the signaling of Unsupported Requests by sending Error Messages.

Table 2-18: **Detailed Function Status Interface Port Descriptions *(Cont'd)***

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
1	10:8	cfg_dev_control_max_payload	3	Configuration Device Control - Max_Payload_Size: Device_Ctrl[7:5]. This field sets maximum TLP payload size. As a Receiver, the logic must handle TLPs as large as the set value. As a Transmitter, the logic must not generate TLPs exceeding the set value. • 000b = 128 bytes max payload size • 001b = 256 bytes max payload size • 010b = 512 bytes max payload size • 011b = 1024 bytes max payload size • 100b = 2048 bytes max payload size • 101b = 4096 bytes max payload size
1	11	cfg_dev_control_enable_ro	1	Configuration Device Control - Enable Relaxed Ordering: Device_Ctrl[4]. When asserted, the logic is permitted to set the Relaxed Ordering bit in the Attributes field of transactions it initiates that do not require strong write ordering.
1	12	cfg_dev_control_ext_tag_en	1	Configuration Device Control - Tag Field Enable: Device_Ctrl[8]. When asserted, enables the logic to use an 8-bit Tag field as a Requester. If deasserted, the logic is restricted to a 5-bit Tag field. Note that the core does not enforce the number of Tag bits used, either in outgoing request TLPs or incoming Completions.
1	13	cfg_dev_control_no_snoop_en	1	Configuration Device Control - Enable No Snoop: Device_Ctrl[11]. When asserted, the logic is permitted to set the No Snoop bit in TLPs it initiates that do not require hardware enforced cache coherency.
1	15:14	0	2	Reserved
2	2:0	cfg_dev_control_max_read_req	3	Configuration Device Control - Max_Read_Request_Size: Device_Ctrl[14:12]. This field sets the maximum Read Request size for the logic as a Requester. The logic must not generate Read Requests with size exceeding the set value. • 000b = 128 bytes maximum Read Request size • 001b = 256 bytes maximum Read Request size • 010b = 512 bytes maximum Read Request size • 011b = 1024 bytes maximum Read Request size • 100b = 2048 bytes maximum Read Request size • 101b = 4096 bytes maximum Read Request size

Table 2-18: **Detailed Function Status Interface Port Descriptions *(Cont'd)***

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
2	3	cfg_link_status_link_training	1	Configuration Link Status - Link Training: Link_Status[11]. Indicates that the Physical Layer LTSSM is in the Configuration or Recovery state, or that 1b was written to the Retrain Link bit but Link training has not yet begun. The core clears this bit when the LTSSM exits the Configuration/Recovery state.
2	6:4	cfg_link_status_current_speed	3	Configuration Link Status - Current Link Speed: Link_Status[1:0]. This field indicates the negotiated Link speed of the given PCI Express Link. • 001b = 2.5 GT/s PCI Express Link • 010b = 5.0 GT/s PCI Express Link • 100b = 8.0 GT/s PCI Express Link
2	10:7	cfg_link_status_negotiated_width	4	Configuration Link Status - Negotiated Link Width: Link_Status[7:4]. This field indicates the negotiated width of the given PCI Express Link (only widths up to x8 displayed). • 0001b = x1 • 0010b = x2 • 0100b = x4 • 1000b = x8
2	11	cfg_link_status_bandwidth_status	1	Configuration Link Status - Link Bandwidth Management Status: Link_Status[14]. Indicates that either of the following has occurred without the Port transitioning through DL_Down status: • A Link retraining has completed following a write of 1b to the Retrain Link bit. ***Note:*** This bit is set following any write of 1b to the Retrain Link bit, including when the Link is in the process of retraining for some other reason. • Hardware has changed Link speed or width to attempt to correct unreliable Link operation, either through an LTSSM timeout or a higher level process. This bit is set if the Physical Layer reports a speed or width change was initiated by the Downstream component that was not indicated as an autonomous change.

Table 2-18: **Detailed Function Status Interface Port Descriptions *(Cont'd)***

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
2	12	cfg_link_status_auto_bandwidth_status	1	Configuration Link Status - Link Autonomous Bandwidth Status: Link_Status[15]. Indicates the core has autonomously changed Link speed or width, without the Port transitioning through DL_Down status, for reasons other than to attempt to correct unreliable Link operation. This bit must be set if the Physical Layer reports a speed or width change was initiated by the Downstream component that was indicated as an autonomous change.
2	15:13	0	3	Reserved
3	1:0	cfg_link_control_aspm_control	2	Configuration Link Control - ASPM Control: Link_Ctrl[1:0]. Indicates the level of ASPM supported, where: • 00b = Disabled • 01b = L0s Entry Enabled • 10b = L1 Entry Enabled • 11b = L0s and L1 Entry Enabled
3	2	cfg_link_control_rcb	1	Configuration Link Control - RCB: Link_Ctrl[3]. Indicates the Read Completion Boundary value, where, • 0=64B • 1=128B
3	3	cfg_link_control_link_disable	1	Configuration Link Control - Link Disable: Link_Ctrl[4]. When asserted, indicates the Link is disabled and directs the LTSSM to the Disabled state.
3	4	cfg_link_control_common_clock	1	Configuration Link Control - Common Clock Configuration: Link_Ctrl[6]. When asserted, indicates that this component and the component at the opposite end of this Link are operating with a distributed common reference clock. When deasserted, indicates they are operating with an asynchronous reference clock.
3	5	cfg_link_control_extended_sync	1	Configuration Link Control - Extended Synch: Link_Ctrl[7]. When asserted, forces the transmission of additional Ordered Sets when exiting the L0s state and when in the Recovery state.

Table 2-18: **Detailed Function Status Interface Port Descriptions *(Cont'd)***

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
3	6	cfg_link_control_clock_pm_en	1	Configuration Link Control - Enable Clock Power Management: Link_Ctrl[8]. For Upstream Ports that support a CLKREQ# mechanism, indicates: • 0b = Clock power management disabled. • 1b = The device is permitted to use CLKREQ#. The core takes no action based on the setting of this bit; external logic must implement this.
3	7	cfg_link_control_hw_auto_width_dis	1	Configuration Link Control - Hardware Autonomous Width Disable: Link_Ctrl[9]. When asserted, disables the core from changing the Link width for reasons other than attempting to correct unreliable Link operation by reducing Link width.
3	8	cfg_link_control_bandwidth_int_en	1	Configuration Link Control - Link Bandwidth Management Interrupt Enable: Link_Ctrl[10]. When asserted, enables the generation of an interrupt to indicate that the Link Bandwidth Management Status bit has been set. The core takes no action based on the setting of this bit; the logic must create the interrupt.
3	9	cfg_link_control_auto_bandwidth_int_en	1	Configuration Link Control - Link Autonomous Bandwidth Interrupt Enable: Link_Ctrl[11]. When asserted, this bit enables the generation of an interrupt to indicate that the Link Autonomous Bandwidth Status bit has been set. The core takes no action based on the setting of this bit; the logic must create the interrupt.
3	10	cfg_tph_requester_enable	1	TPH Requester Enable: Bit [8] of the TPH Requester Control Register in the TPH Requester Capability Structure of the function. These bits are active only in the Endpoint mode. Indicates whether the software has enabled the device to generate requests with TPH Hints from the associated Function.
3	13:11	cfg_tph_steering_tag_mode	3	TPH Steering Tag Mode. Reflect the setting of the ST Mode Select bits in the TPH Requester Control Register. These bits are active only in the Endpoint mode. They indicate the allowed modes for generation of TPH Hints by the corresponding Function.
3	15:14	0	2	Reserved

Table 2-18: **Detailed Function Status Interface Port Descriptions *(Cont'd)***

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
4	3:0	cfg_dev_control2_cpl_timeout_val	4	Configuration Device Control 2 - Completion Timeout Value: Device_Ctrl2[3:0]. This is the time range that the logic regard as a Request is pending Completion as a Completion Timeout. The core takes no action based on this setting. • 0000b = 50 µs to 50 ms (default) • 0001b = 50 µs to 100 µs • 0010b = 1 ms to 10 ms • 0101b = 16 ms to 55 ms • 0110b = 65 ms to 210 ms • 1001b = 260 ms to 900 ms • 1010b = 1 s to 3.5 s • 1101b = 4 s to 13 s • 1110b = 17 s to 64 s
4	4	cfg_dev_control2_cpl_timeout_dis	1	Configuration Device Control 2 - Completion Timeout Disable: Device_Ctrl2[4]. This disables the Completion Timeout counters.
4	5	cfg_dev_control2_atomic_requester_en	1	Configuration Device Control 2 - Atomic Requester Enable: Device_Ctrl2[6]. Applicable only to Endpoints and Root Ports; must be hardwired to `0b` for other Function types. The Function is allowed to initiate AtomicOp Requests only if this bit and the Bus Master Enable bit in the Command register are both Set. This bit is required to be RW if the Endpoint or Root Port is capable of initiating AtomicOp Requests, but otherwise is permitted to be hardwired to `0b`. This bit does not serve as a capability bit. This bit is permitted to be RW even if no AtomicOp Requester capabilities are supported by the Endpoint or Root Port. Default value of this bit is `0b`. 32 nm
4	6	cfg_dev_control2_ido_req_en	1	Configuration Device Control 2 - IDO Request Enable: Device_Ctrl2[8]. If this bit is Set, the Function is permitted to set the ID-Based Ordering (IDO) bit (Attribute[2]) of Requests it initiates (see Section 2.2.6.3 and Section 2.4). Endpoints, including RC Integrated Endpoints, and Root Ports are permitted to implement this capability. A Function is permitted to hardwire this bit to `0b` if it never sets the IDO attribute in Requests. Default value of this bit is `0b`. 32 nm

Table 2-18: **Detailed Function Status Interface Port Descriptions *(Cont'd)***

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
4	7	cfg_dev_control2_ido_cpl_en	1	Configuration Device Control 2 - IDO Completion Enable: Device_Ctrl2[9]. If this bit is Set, the Function is permitted to set the ID-Based Ordering (IDO) bit (Attribute[2]) of Completions it returns (see Section 2.2.6.3 and Section 2.4). Endpoints, including RC Integrated Endpoints, and Root Ports are permitted to implement this capability. A Function is permitted to hardwire this bit to `0b` if it never sets the IDO attribute in Requests. Default value of this bit is `0b`. 32 nm
4	8	cfg_dev_control2_ltr_en	1	Configuration Device Control 2 - LTR Mechanism Enable: Device_Ctrl2[10]. If this bit is Set, the Function is permitted to set the ID-Based Ordering (IDO) bit (Attribute[2]) of Completions it returns (see Section 2.2.6.3 and Section 2.4). Endpoints, including RC Integrated Endpoints, and Root Ports are permitted to implement this capability. A Function is permitted to hardwire this bit to `0b` if it never sets the IDO attribute in Requests. Default value of this bit is `0b`. 32 nm
4	13:9	cfg_dpa_substate	5	Dynamic Power Allocation Substate: Reflect the setting of the Dynamic Power Allocation Substate field in the DPA Control Register.
4	15:14	0	1	Reserved
5	0	cfg_root_control_syserr_corr_err_en	1	Configuration Root Control - System Error on Correctable Error Enable: Root_Control[0]. This bit enables the logic to generate a System Error for reported Correctable Errors.
5	1	cfg_root_control_syserr_non_fatal_err_en	1	Configuration Root Control - System Error on Non-Fatal Error Enable: Root_Control[1]. This bit enables the logic to generate a System Error for reported Non-Fatal Errors.
5	2	cfg_root_control_syserr_fatal_err_en	1	Configuration Root Control - System Error on Fatal Error Enable: Root_Control[2]. This bit enables the logic to generate a System Error for reported Fatal Errors.
5	3	cfg_root_control_pme_int_en	1	Configuration Root Control - PME Interrupt Enable: Root_Control[3]. This bit enables the logic to generate an Interrupt for received PME Messages.

Table 2-18: **Detailed Function Status Interface Port Descriptions *(Cont'd)***

cfg_per_func_status_control [bit]	cfg_per_func_status_data [bit/slice]	Status Output	Width	Description
5	4	cfg_aer_rooterr_corr_err_reporting_en	1	Configuration AER - Correctable Error Reporting Enable: AER_Root_Error_Command[0]. This bit enables the logic to generate interrupts for reported Correctable Errors.
5	5	cfg_aer_rooterr_non_fatal_err_reporting_en	1	Configuration AER - Non Fatal Error Reporting Enable: AER_Root_Error_Command[1]. This bit enables the user logic to generate interrupts for reported Non-Fatal Errors.
5	6	cfg_aer_rooterr_fatal_err_reporting_en	1	Configuration AER - Fatal Error Reporting Enable: AER_Root_Error_Command[2]. This bit enables the user logic to generate interrupts for reported Fatal Errors.
5	7	cfg_aer_rooterr_corr_err_received	1	Configuration AER - Correctable Error Messages Received: AER_Root_Error_Status[0]. Indicates that an ERR_COR Message was received.
5	8	cfg_aer_rooterr_non_fatal_err_received	1	Configuration AER - Non-Fatal Error Messages Received: AER_Root_Error_Status[5]. Indicates that an ERR_NFE Message was received.
5	9	cfg_aer_rooterr_fatal_err_received	1	Configuration AER - Fatal Error Messages Received: AER_Root_Error_Status[6]. Indicates that an ERR_FATAL Message was received.
5	15:10	0	6	Reserved

Configuration Control Interface

The Configuration Control interface signals allow a broad range of information exchange between the user application and the core. The user application uses this interface to 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; and receive per function configuration information. This interface also provides handshaking between the user application and the core when a Power State change or function level reset occurs.

Table 2-19 defines the ports in the Configuration Control interface of the core.

Table 2-19: **Configuration Control Interface Port Descriptions**

Port	Direction	Width	Description
cfg_hot_reset_in	Input	1	Configuration Hot Reset In. In RP mode, assertion transitions LTSSM to hot reset state, active-High.
cfg_hot_reset_out	Output	1	Configuration Hot Reset Out. In EP mode, assertion indicates that EP has transitioned to the hot reset state, active-High.
cfg_config_space_enable	Input	1	Configuration Configuration Space Enable. When this input is set to 0 in the Endpoint mode, the core generates a CRS Completion in response to Configuration Requests. This port should be held deasserted when the core configuration registers are loaded from the DRP due to a change in attributes. This prevents the core from responding to Configuration Requests before all the registers are loaded. This input can be High when the power-on default values of the Configuration Registers do not need to be modified before Configuration space enumeration. This input is not applicable for Root Port mode.
cfg_per_function_update_done	Output	1	Configuration per Function Update Complete. Asserted in response to `cfg_per_function_output_request` assertion, for one cycle after the request is complete.
cfg_per_function_number	Input	3	Configuration Per Function Target Function Number. You provide the function number (0-7), where value 0–1 corresponds to PF0–1, and value 2–7 corresponds to VF0–5, and asserts `cfg_per_function_output_request` to obtain per function output values for the selected function.
cfg_per_function_output_request	Input	1	Configuration Per Function Output Request. When this port is asserted with a function number value on `cfg_per_function_number`, the core presents information on per-function configuration output pins and asserts `cfg_update_done` when complete.
cfg_dsn	Input	64	Configuration Device Serial Number. Indicates the value that should be transferred to the Device Serial Number Capability on PF0. Bits [31:0] are transferred to the first (Lower) Dword (byte offset `0x4h` of the Capability), and bits [63:32] are transferred to the second (Upper) Dword (byte offset `0x8h` of the Capability). If this value is not statically assigned, the user application must pulse `user_cfg_input_update` after it is stable.

Table 2-19: **Configuration Control Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_ds_bus_number	Input	8	Configuration Downstream Bus Number. • Downstream Port: Provides the bus number portion of the Requester ID (RID) of the Downstream Port. This is used in TLPs generated inside the core, such as UR Completions and Power-management messages; it does not affect TLPs presented on the TRN interface. • Upstream Port: No role.
cfg_ds_device_number	Input	5	Configuration Downstream Device Number: • Downstream Port: Provides the device number portion of the RID of the Downstream Port. This is used in TLPs generated inside the core, such as UR Completions and Power-management messages; it does not affect TLPs presented on the TRN interface. • Upstream Port: No role.
cfg_ds_function_number	Input	3	Configuration Downstream Function Number. • Downstream Port: Provides the function number portion of the RID of the Downstream Port. This is used in TLPs generated inside the core, such as UR Completions and Power-management messages; it does not affect TLPs presented on the TRN interface. • Upstream Port: No role.
cfg_power_state_change_ack	Input	1	Configuration Power State Ack. You must assert this input to the core for one cycle in response to the assertion of `cfg_power_state_change_interrupt`, when it is ready to transition to the low-power state requested by the configuration write request. The user application can permanently hold this input High if it does not need to delay the return of the completions for the configuration write transactions, causing power-state changes.

Table 2-19: **Configuration Control Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_power_state_change_interrupt	Output	1	Power State Change Interrupt. The core asserts this output when the power state of a Physical or Virtual Function is being changed to the D1 or D3 states by a write into its Power Management Control Register. The core holds this output High until the user application asserts the `cfg_power_state_change_ack` input to the core. While `cfg_power_state_change_interrupt` remains High, the core does not return completions for any pending configuration read or write transaction received by the core. The purpose is to delay the completion for the configuration write transaction that caused the state change until the user application is ready to transition to the low-power state. When `cfg_power_state_change_interrupt` is asserted, the Function number associated with the configuration write transaction is provided on the `cfg_snp_function_number[7:0]` output. When the user application asserts `cfg_power_state_change_ack`, the new state of the Function that underwent the state change is reflected on `cfg_function_power_state` (for PFs) or the `cfg_vf_power_state` (for VFs) outputs of the core.
cfg_err_cor_in	Input	1	Correctable Error Detected. The user application activates this input for one cycle to indicate a correctable error detected within the user logic that needs to be reported as an internal error through the PCI Express Advanced Error Reporting mechanism. In response, the core sets the Corrected Internal Error Status bit in the AER Correctable Error Status Register of all enabled Functions, and also sends an error message if enabled to do so. This error is not considered Function-specific.
cfg_err_uncor_in	Input	1	Uncorrectable Error Detected. The user application activates this input for one cycle to indicate a uncorrectable error detected within the user logic that needs to be reported as an internal error through the PCI Express Advanced Error Reporting mechanism. In response, the core sets the uncorrected Internal Error Status bit in the AER Uncorrectable Error Status Register of all enabled Functions, and also sends an error message if enabled to do so. This error is not considered Function-specific.
cfg_flr_done	Input	2	Function Level Reset Complete. You must assert bit *i* of this bus when the reset operation of Function *i* completes. This causes the core to deassert `cfg_flr_in_process` for Function *i* and to re-enable configuration accesses to the Function.

Table 2-19: **Configuration Control Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_vf_flr_done	Input	6	Function Level Reset for virtual Function is Complete. You must assert bit *i* of this bus the reset operation of Virtual Function *i* completes. This causes the core to deassert `cfg_vf_flr_in_process` for Function *i* and to re-enable configuration accesses to the Virtual Function.
cfg_flr_in_process	Output	2	Function Level Reset In Process. The core asserts bit *i* of this bus when the host initiates a reset of 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 Function to indicate the completion of the reset operation.
cfg_vf_flr_in_process	Output	6	Function Level Reset In Process for Virtual Function. The core asserts bit *i* of this bus when the host initiates a reset of Virtual Function *i* though its FLR bit in the configuration space. The core continues to hold the output High until the user sets the corresponding `cfg_vf_flr_done` input for the corresponding Function to indicate the completion of the reset operation.
cfg_req_pm_transition_l23_ready	Input	1	When the core is configured as an Endpoint, the user application asserts this input to transition the power management state of the core to L23_READY (see Chapter 5 of the *PCI Express Specification* for a detailed description of power management). This is done after the PCI Functions in the core are placed in the D3 state and after the user application acknowledges the PME_Turn_Off message from the Root Complex. Asserting this input causes the link to transition to the L3 state, and requires a hard reset to resume operation. This input can be hardwired to 0 if the link is not required to transition to L3. This input is not used in Root Complex mode.
cfg_link_training_enable	Input	1	This input must be set to 1 to enable the Link Training Status State Machine (LTSSM) to bring up the link. Setting it to 0 forces the LTSSM to stay in the Detect.Quiet state.

Configuration Interrupt Controller Interface

The Configuration Interrupt Controller interface allows the user application to set Legacy PCIe interrupts, MSI interrupts, or MSI-X interrupts. The core provides the interrupt status on the configuration interrupt sent and fail signals. Table 2-20 defines the ports in the Configuration Interrupt Controller interface of the core.

Table 2-20: **Configuration Interrupt Controller Interface Port Descriptions**

Port	Direction	Width	Description
cfg_interrupt_int	Input	4	Configuration INTx Vector. When the core is configured as EP, these four inputs are used by the user application to signal an interrupt from any of its PCI Functions to the RC using the Legacy PCI Express Interrupt Delivery mechanism of PCI Express. These four inputs correspond to INTA, INTB, INTC, and INTD. Asserting one of these signals causes the core to send out an Assert_INTx message, and deasserting the signal causes the core to transmit a Deassert_INTx message.
cfg_interrupt_sent	Output	1	Configuration INTx Sent. A pulse on this output indicates that the core has sent an INTx Assert or Deassert message in response to a change in the state of one of the `cfg_interrupt_int` inputs.
cfg_interrupt_pending	Input	2	Configuration INTx Interrupt Pending (active-High). Per Function indication of a pending interrupt. `cfg_interrupt_pending[0]` corresponds to Function #0.
cfg_interrupt_msi_enable	Output	2	Configuration Interrupt MSI Function Enabled. Indicates that Message Signaling Interrupt (MSI) messaging is enabled per Function.
cfg_interrupt_msi_vf_enable	Output	6	Configuration Interrupt MSI on VF Enabled. Indicates that MSI messaging is enabled, per Virtual Function.
cfg_interrupt_msi_int	Input	32	Configuration Interrupt MSI Vector. When the core is configured in Endpoint mode to support MSI interrupts, these inputs are used to signal the 32 distinct interrupt conditions associated with a PCI Function (Physical or Virtual) from the user logic to the core. The Function number must be specified on the `cfg_interrupt_msi_function_number` input. After placing the Function number on the input `cfg_interrupt_msi_function_number`, the user logic must activate one of these signals for one cycle to transmit an interrupt. The user logic must not activate more than one of the 32 interrupt inputs in the same cycle. The core internally registers the interrupt condition on the 0-to-1 transition of any bit in `cfg_interrupt_msi_int`. After asserting an interrupt, the user logic must wait for the `cfg_interrupt_msi_sent` or `cfg_interrupt_msi_fail` indication from the core before asserting a new interrupt.

Table 2-20: **Configuration Interrupt Controller Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_interrupt_msi_sent	Output	1	Configuration Interrupt MSI Interrupt Sent. The core generates a one-cycle pulse on this output to signal that an MSI interrupt message has been transmitted on the link. The user logic must wait for this pulse before signaling another interrupt condition to the core.
cfg_interrupt_msi_fail	Output	1	Configuration Interrupt MSI Interrupt Operation Failed. A one-cycle pulse on this output indicates that an MSI interrupt message was aborted before transmission on the link. The user application must retransmit the MSI interrupt in this case.
cfg_interrupt_msi_mmenable	Output	6	Configuration Interrupt MSI Function Multiple Message Enable. When the core is configured in the Endpoint mode to support MSI interrupts, these outputs are driven by the "Multiple Message Enable" bits of the MSI Control Registers associated with Physical Functions. These bits encode the number of allocated MSI interrupt vectors for the corresponding Function. Bits [2:0] correspond to Physical Function 0.
cfg_interrupt_msi_pending_status	Input	64	Configuration Interrupt MSI Pending Status. These inputs provide the status of the MSI pending interrupts for the Physical Functions. The setting of these pins determines the value read from the MSI Pending Bits Register of the corresponding PF. Bits [31:0] belong to PF 0, bits [63:32] to PF 1. The MSI Pending bits register contains the pending bits for MSI Interrupts. A read from this location returns the state of MSI_MASK inputs of the core. This is a 32-bit wide RO register with a default value of MSI MASK inputs.
cfg_interrupt_msi_mask_update	Output	1	Configuration Interrupt MSI Function Mask Update. Asserted for one cycle when any enabled functions in the MSI Mask Register change value. MSI Mask register contains the Mask bits for MSI interrupts. The Multiple Message Capable field in the MSI Control Register specifies the number of distinct interrupts for the Function, which determines the number of valid mask bits. This is a 32-bit wide RW register with a default value of 0.

Table 2-20: **Configuration Interrupt Controller Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_interrupt_msi_select	Input	4	Configuration Interrupt MSI Select. Values `0000b-0001b` correspond to PF0-1 selection, and values `0010b-0111b` correspond to VF0-5 selection. `cfg_interrupt_msi_data[31:0]` presents the value of the MSI Mask register from the selected function. When this input is driven to `1111b`, `cfg_interrupt_msi_data[17:0]` presents the "Multiple Message Enable" bits of the MSI Control Registers associated with all Virtual Functions. These bits encode the number of allocated MSI interrupt vectors for the corresponding Function. `cfg_interrupt_msi_data[2:0]` correspond to Virtual Function 0, and so on.
cfg_interrupt_msi_data	Output	32	Configuration Interrupt MSI Data. The value presented depends on `cfg_interrupt_msi_select`.
cfg_interrupt_msix_enable	Output	2	Configuration Interrupt MSI-X Function Enabled. When asserted, indicates that the Message Signaling Interrupt (MSI-X) messaging is enabled, per Function.
cfg_interrupt_msix_mask	Output	2	Configuration Interrupt MSI-X Function Mask. Indicates the state of the Function Mask bit in the MSI-X Message Control field, per Function.
cfg_interrupt_msix_vf_enable	Output	6	Configuration Interrupt MSI-X on VF Enabled. When asserted, indicates that Message Signaling Interrupt (MSI-X) messaging is enabled, per Virtual Function.
cfg_interrupt_msix_vf_mask	Output	6	Configuration Interrupt MSI-X VF Mask. Indicates the state of the Function Mask bit in the MSI-X Message Control field, per Virtual Function.
cfg_interrupt_msix_address	Input	64	Configuration Interrupt MSI-X Address. When the core is configured to support MSI-X interrupts, this bus is used by the user logic to communicate the address to be used for an MSI-X message.
cfg_interrupt_msix_data	Input	32	Configuration Interrupt MSI-X Data. When the core is configured to support MSI-X interrupts, this bus is used by the user logic to communicate the data to be used for an MSI-X message.

Table 2-20: **Configuration Interrupt Controller Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_interrupt_msix_int	Input	1	Configuration Interrupt MSI-X Data Valid. This signal indicates that valid information has been placed on the `cfg_interrupt_msix_address[63:0]` and `cfg_interrupt_msix_data[31:0]` buses, and the originating Function number has been placed on `cfg_interrupt_msi_function_number[3:0]`. The core internally registers the associated address and data from `cfg_interrupt_msix_address` and `cfg_interrupt_msix_data` on the 0-to-1 transition of this valid signal. After asserting an interrupt, the user logic must wait for the `cfg_interrupt_msix_sent` or `cfg_interrupt_msix_fail` indication from the core before asserting a new interrupt.
cfg_interrupt_msix_sent	Output	1	Configuration Interrupt MSI-X Interrupt Sent. The core generates a one-cycle pulse on this output to indicate that it has accepted the information placed on the `cfg_interrupt_msix_address[63:0]` and `cfg_interrupt_msix_data[31:0]` buses, and an MSI-X interrupt message has been transmitted on the link. The user application must wait for this pulse before signaling another interrupt condition to the core.
cfg_interrupt_msix_fail	Output	1	Configuration Interrupt MSI-X Interrupt Operation Failed. A one-cycle pulse on this output indicates that the interrupt controller has failed to transmit MSI-X interrupt on the link. The user application must retransmit the MSI-X interrupt in this case.
cfg_interrupt_msi_attr	Input	3	Configuration Interrupt MSI/MSI-X TLP Attr. These bits provide the setting of the Attribute bits to be used for the MSI/MSI-X interrupt request. Bit 0 is the No Snoop bit, and bit 1 is the Relaxed Ordering bit. Bit 2 is the ID-Based Ordering bit. The core samples these bits on a 0-to-1 transition on `cfg_interrupt_msi_int` or `cfg_interrupt_msix_int`.
cfg_interrupt_msi_tph_present	Input	1	Configuration Interrupt MSI/MSI-X TPH Present. Indicates the presence of a Transaction Processing Hint (TPH) in the MSI/MSI-X interrupt request. The user application must set this bit while asserting `cfg_interrupt_msi_int` or `cfg_interrupt_msix_int`, if it includes a TPH in the MSI or MSI-X transaction.
cfg_interrupt_msi_tph_type	Input	2	Configuration Interrupt MSI/MSI-X TPH Type. When `cfg_interrupt_msi_tph_present` is `1'b1`, these two bits supply the two-bit type associated with the hint. The core samples these bits on a 0-to-1 transition on `cfg_interrupt_msi_int` or `cfg_interrupt_msix_int`.

Table 2-20: **Configuration Interrupt Controller Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
cfg_interrupt_msi_tph_st_tag	Input	9	Configuration Interrupt MSI/MSI-X TPH Steering Tag. When `cfg_interrupt_msi_tph_present` is `1'b1`, the Steering Tag associated with the Hint must be placed on `cfg_interrupt_msi_tph_st_tag[7:0]`. Setting `cfg_interrupt_msi_tph_st_tag[8]` to `1b` activates the Indirect Tag mode. In the Indirect Tag mode, the core uses bits [5:0] of `cfg_interrupt_msi_tph_st_tag` as an index into its Steering Tag Table (STT) in the TPH Capability Structure (STT is limited to 64 entries per Function), and inserts the tag from this location in the transmitted request MSI/X TLP. Setting `cfg_interrupt_msi_tph_st_tag[8]` to `0b` activates the Direct Tag mode. In the Direct Tag mode, the core inserts `cfg_interrupt_msi_tph_st_tag[7:0]` directly as the Tag in the transmitted MSI/X TLP. The core samples these bits on a 0-to-1 transition on any `cfg_interrupt_msi_int` bits or `cfg_interrupt_msix_int`.
cfg_interrupt_msi_function_number	Input	3	Configuration MSI/MSI-X Initiating Function. Indicates the Endpoint function number initiating the MSI or MSI-X transaction: • 0: PF0 • 1: PF1 • 2: VF0 • 3: VF1 • 4: VF2 ... • 7: VF5

Configuration Extend Interface

The Configuration Extend interface allows the core to transfer configuration information with the user application when externally implemented configuration registers are implemented. Table 2-21 defines the ports in the Configuration Extend interface of the core.

Table 2-21: **Configuration Extend Interface Port Descriptions**

Port	Direction	Width	Description
cfg_ext_read_received	Output	1	Configuration Extend Read Received. The core asserts this output when it has received a configuration read request from the link. When neither user-implemented legacy or extended configuration space is enabled, receipt of a configuration read results in a one-cycle assertion of this signal, together with valid `cfg_ext_register_number` and `cfg_ext_function_number`. When user-implemented legacy, extended configuration space, or both are enabled, for the `cfg_ext_register_number` ranges, `0x10-0x1f` or `0x100-0x3ff`, respectively, this signal is asserted, until user logic presents `cfg_ext_read_data` and `cfg_ext_read_data_valid`. For `cfg_ext_register_number` ranges outside `0x10-0x1f` or `0x100-0x3ff`, receipt of a configuration read always results in a one-cycle assertion of this signal.
cfg_ext_write_received	Output	1	Configuration Extend Write Received. The core generates a one-cycle pulse on this output when it has received a configuration write request from the link.
cfg_ext_register_number	Output	10	Configuration Extend Register Number. The 10-bit address of the configuration register being read or written. The data is valid when `cfg_ext_read_received` or `cfg_ext_write_received` is High.
cfg_ext_function_number	Output	8	Configuration Extend Function Number. The 8-bit Function Number corresponding to the configuration read or write request. The data is valid when `cfg_ext_read_received` or `cfg_ext_write_received` is High.
cfg_ext_write_data	Output	32	Configuration Extend Write Data. Data being written into a configuration register. This output is valid when `cfg_snp_write_received` is High.
cfg_ext_write_byte_enable	Output	4	Configuration Extend Write Byte Enable. Byte enables for a configuration write transaction.
cfg_ext_read_data	Input	32	Configuration Extend Read Data. You can provide data from an externally implemented configuration register to the core through this bus. The core samples this data on the next positive edge of the clock after it sets `cfg_snp_read_received` High, if you have set `cfg_snp_read_data_valid`.
cfg_ext_read_data_valid	Input	1	Configuration Extend Read Data Valid. The user application asserts this input to the core to supply data from an externally implemented configuration register. The core samples this input data on the next positive edge of the clock after it sets `cfg_snp_read_received` High.

Clock and Reset Interface

Fundamental to the operation of the core, the Clock and Reset interface provides the system-level clock and reset to the core as well as the user application clock and reset signal. Table 2-22 defines the ports in the Clock and Reset interface of the core.

Table 2-22: **Clock and Reset Interface Port Descriptions**

Port	Direction	Width	Description
user_clk	Output	1	User clock output (62.5, 125, or 250 MHz). This clock has a fixed frequency and is configured in the Vivado® Integrated Design Environment (IDE).
user_reset	Output	1	This signal is deasserted synchronously with respect to `user_clk`. It is deasserted and asserted asynchronously with `sys_reset` assertion.
sys_clk	Input	1	Reference clock. This clock has a selectable frequency of 100 MHz, 125 MHz, or 250 MHz.
sys_reset	Input	1	Fundamental reset input to the core (asynchronous). This input is active-Low by default to match the PCIe edge connector reset polarity. You can reset to active-High using an option in the Vivado IDE, but this can result in incompatibility with the PCIe edge connector. For Endpoint configurations using the X0Y0 PCIe location, `sys_reset` should be directly driven by the PERSTN0 package pin. To drive sys_reset on the X0Y0 PCIe block from any other package pin or from the configurable logic, disable dedicated reset routing through the Vivado IDE. It is recommended that the PERSTN0 package pin always be used to drive `sys_reset` for the X0Y0 PCIe block. For other PCIe blocks configured as endpoints, the PERSTN1 package pin should be used to drive `sys_reset` when possible.
pcie_perstn0_out	Output	1	Output that is a direct pass-through from the PERSTN0 package pin through the `sys_reset` input port for the X0Y0 PCIe block. This port should only be used when the X0Y0 PCIe block is selected, dedicated routing is enabled (default), and `sys_reset` is driven by the PERSTN0 package pin. For all other configurations and PCIe locations, this port should not be connected. PERSTN0 should be used to drive the `sys_reset` input port for endpoint configurations that use the X0Y0 PCIe location.

Table 2-22: **Clock and Reset Interface Port Descriptions *(Cont'd)***

Port	Direction	Width	Description
pcie_perstn1_in	Input	1	Input to a dedicated route from the PERSTN1 package pin to the PCIe_X0Y0 pcie_perstn1_out output. This input can be driven only by the PERSTN1 package pin and should only be used for the X0Y0 PCIe location. For all other PCIe locations and when unused, this port should be tied to a constant zero (1'b0)
pcie_perstn1_out	Output	1	Output that is a direct pass-through from the PERSTN1 package pin through the `pcie_perstn1_in` input port for the X0Y0 PCIe block. This port can only be used when the X0Y0 PCIe block is selected and `pcie_perstn1_in` is driven by the PERSTN1 package pin. For all other configurations and PCIe locations, this port should not be connected. PERSTN1 should be used to drive the `sys_reset` input port for PCIe endpoint configurations that are not set to the X0Y0 PCIe location.

The PERSTN0/PERSTN1 package pins and the ports described in Table 2-24 are used for dedicated PCIe reset routing. These are dedicated ports from the PERSTN package pins to the PCIe core (X0Y0). Users who need Tandem Configuration support should use these pins as described in Table 2-24. The general guidelines for using PERSTN0 and PERSTN1 pins are as follows:

- PCIe_X0Y* RootPort configurations may use any pin to drive the edge connector reset.
- PCIe_X0Y0 EndPoint configurations should always use PERSTN0 as PCIe edge connector reset input pin.
- PCIe_X0Y(non-zero) EndPoint configurations should give priority to PERST1 as the PCIe edge connector reset input pin, but may use any pin if desired.

PCI Express Interface

The PCI Express (PCI_EXP) interface consists of differential transmit and receive pairs organized in multiple lanes. A PCI Express lane consists of a pair of transmit differential signals (`pci_exp_txp`, `pci_exp_txn`) and a pair of receive differential signals {`pci_exp_rxp`, `pci_exp_rxn`}. The 1-lane core supports only Lane 0, the 2-lane core supports lanes 0–1, the 4-lane core supports lanes 0-3, and the 8-lane core supports lanes 0–7. Transmit and receive signals of the PCI_EXP interface are defined in Table 2-23.

Table 2-23: **PCI Express Interface Signals for 1-, 2-, 4- and 8-Lane Cores**

Lane Number	Name	Direction	Description
1-Lane Cores			
0	pci_exp_txp0	Output	PCI Express Transmit Positive: Serial Differential Output 0 (+)
	pci_exp_txn0	Output	PCI Express Transmit Negative: Serial Differential Output 0 (–)
	pci_exp_rxp0	Input	PCI Express Receive Positive: Serial Differential Input 0 (+)
	pci_exp_rxn0	Input	PCI Express Receive Negative: Serial Differential Input 0 (–)

Table 2-23: **PCI Express Interface Signals for 1-, 2-, 4- and 8-Lane Cores *(Cont'd)***

Lane Number	Name	Direction	Description
2-Lane Cores			
0	pci_exp_txp0	Output	PCI Express Transmit Positive: Serial Differential Output 0 (+)
	pci_exp_txn0	Output	PCI Express Transmit Negative: Serial Differential Output 0 (–)
	pci_exp_rxp0	Input	PCI Express Receive Positive: Serial Differential Input 0 (+)
	pci_exp_rxn0	Input	PCI Express Receive Negative: Serial Differential Input 0 (–)
1	pci_exp_txp1	Output	PCI Express Transmit Positive: Serial Differential Output 1 (+)
	pci_exp_txn1	Output	PCI Express Transmit Negative: Serial Differential Output 1 (–)
	pci_exp_rxp1	Input	PCI Express Receive Positive: Serial Differential Input 1 (+)
	pci_exp_rxn1	Input	PCI Express Receive Negative: Serial Differential Input 1 (–)
4-Lane Cores			
0	pci_exp_txp0	Output	PCI Express Transmit Positive: Serial Differential Output 0 (+)
	pci_exp_txn0	Output	PCI Express Transmit Negative: Serial Differential Output 0 (–)
	pci_exp_rxp0	Input	PCI Express Receive Positive: Serial Differential Input 0 (+)
	pci_exp_rxn0	Input	PCI Express Receive Negative: Serial Differential Input 0 (–)
1	pci_exp_txp1	Output	PCI Express Transmit Positive: Serial Differential Output 1 (+)
	pci_exp_txn1	Output	PCI Express Transmit Negative: Serial Differential Output 1 (–)
	pci_exp_rxp1	Input	PCI Express Receive Positive: Serial Differential Input 1 (+)
	pci_exp_rxn1	Input	PCI Express Receive Negative: Serial Differential Input 1 (–)
2	pci_exp_txp2	Output	PCI Express Transmit Positive: Serial Differential Output 2 (+)
	pci_exp_txn2	Output	PCI Express Transmit Negative: Serial Differential Output 2 (–)
	pci_exp_rxp2	Input	PCI Express Receive Positive: Serial Differential Input 2 (+)
	pci_exp_rxn2	Input	PCI Express Receive Negative: Serial Differential Input 2 (–)
3	pci_exp_txp3	Output	PCI Express Transmit Positive: Serial Differential Output 3 (+)
	pci_exp_txn3	Output	PCI Express Transmit Negative: Serial Differential Output 3 (–)
	pci_exp_rxp3	Input	PCI Express Receive Positive: Serial Differential Input 3 (+)
	pci_exp_rxn3	Input	PCI Express Receive Negative: Serial Differential Input 3 (–)
8-Lane Cores			
0	pci_exp_txp0	Output	PCI Express Transmit Positive: Serial Differential Output 0 (+)
	pci_exp_txn0	Output	PCI Express Transmit Negative: Serial Differential Output 0 (–)
	pci_exp_rxp0	Input	PCI Express Receive Positive: Serial Differential Input 0 (+)
	pci_exp_rxn0	Input	PCI Express Receive Negative: Serial Differential Input 0 (–)

Table 2-23: **PCI Express Interface Signals for 1-, 2-, 4- and 8-Lane Cores *(Cont'd)***

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

Configuration Space

The PCI configuration space consists of three primary parts, illustrated in Table 2-24. These include:

- Legacy PCI v3.0 Type 0/1 Configuration Space Header
  - Type 0 Configuration Space Header used by Endpoint applications (see Table 2-25)
  - Type 1 Configuration Space Header used by Root Port applications (see Table 2-26)
- Legacy Extended Capability Items
  - PCIe Capability Item
  - Power Management Capability Item
  - Message Signaled Interrupt (MSI) Capability Item
  - MSI-X Capability Item (optional)
- PCIe Capabilities
  - Advanced Error Reporting Extended Capability Structure (AER)
  - Alternate Requestor ID (ARI) (optional)
  - Device Serial Number Extended Capability Structure (DSN) (optional)
  - Power Budgeting Enhanced
  - Capability Header (PB) (optional)
  - Resizable BAR (RBAR) (optional)
  - Latency Tolerance Reporting (LTR) (optional)
  - Dynamic Power Allocation (DPA) (optional)
  - Single Root I/O Virtualization (SR-IOV) (optional)
  - Transaction Processing Hints (TPH) (optional)
  - Virtual Channel Extended Capability Structure (VC) (optional)
- PCIe Extended Capabilities
  - Device Serial Number Extended Capability Structure (optional)
  - Virtual Channel Extended Capability Structure (optional)
  - Advanced Error Reporting Extended Capability Structure (optional)

The core implements up to four legacy extended capability items.

For more information about enabling this feature, see Chapter 4, Customizing and Generating the Core.

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

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

Table 2-24: **Common PCI Configuration Space Header**

<table>
<tr><th></th><th colspan="2">31 … 16</th><th colspan="2">15 … 0</th><th></th></tr>
<tr><td></td><td colspan="2">Device ID</td><td colspan="2">Vendor ID</td><td>000h</td></tr>
<tr><td></td><td colspan="2">Status</td><td colspan="2">Command</td><td>004h</td></tr>
<tr><td></td><td colspan="3">Class Code</td><td>Rev ID</td><td>008h</td></tr>
<tr><td></td><td>BIST</td><td>Header</td><td>Lat Timer</td><td>Cache Ln</td><td>00Ch</td></tr>
<tr><td></td><td colspan="4" rowspan="12">Header Type Specific
(see Table 2-25 and Table 2-26)</td><td>010h</td></tr>
<tr><td></td><td>014h</td></tr>
<tr><td></td><td>018h</td></tr>
<tr><td></td><td>01Ch</td></tr>
<tr><td></td><td>020h</td></tr>
<tr><td></td><td>024h</td></tr>
<tr><td></td><td>028h</td></tr>
<tr><td></td><td>02Ch</td></tr>
<tr><td></td><td>030h</td></tr>
<tr><td></td><td>034h</td></tr>
<tr><td></td><td>038h</td></tr>
<tr><td></td><td>03Ch</td></tr>
<tr><td></td><td colspan="3"></td><td>CapPtr</td><td>034h</td></tr>
<tr><td></td><td colspan="2"></td><td>Intr Pin</td><td>Intr Line</td><td>03Ch</td></tr>
<tr><td></td><td colspan="4">Reserved</td><td>040h-07Ch</td></tr>
<tr><td></td><td colspan="2">PM Capability</td><td>NxtCap</td><td>PM Cap</td><td>080h</td></tr>
<tr><td></td><td>Data</td><td>Reserved</td><td colspan="2">PMCSR</td><td>084h</td></tr>
<tr><td></td><td colspan="4">Reserved</td><td>088h-08Ch</td></tr>
<tr><td rowspan="6">Customizable[1]</td><td colspan="2">MSI Control</td><td>NxtCap</td><td>MSI Cap</td><td>090h</td></tr>
<tr><td colspan="4">Message Address (Lower)</td><td>094h</td></tr>
<tr><td colspan="4">Message Address (Upper)</td><td>098h</td></tr>
<tr><td colspan="2">Reserved</td><td colspan="2">Message Data</td><td>09Ch</td></tr>
<tr><td colspan="4">Mask Bits</td><td>0A0h</td></tr>
<tr><td colspan="4">Pending Bits</td><td>0A4h</td></tr>
</table>

Table 2-24: **Common PCI Configuration Space Header *(Cont'd)***

	31	16 15	0	
	Reserved			0A8h-0ACh
Optional[3]	MSI-X Control	NxtCap	MSI-X Cap	0B0h
	Table Offset		Table BIR	0B4h
	PBA Offset		PBA BIR	0B8h
	Reserved			0BCh
	PE Capability	NxtCap	PE Cap	0C0h
	PCI Express Device Capabilities			0C4h
	Device Status	Device Control		0C8h
	PCI Express Link Capabilities			0CCh
	Link Status	Link Control		0D0h
Root Port Only[2]	Slot Capabilities			0D4h
	Slot Status	Slot Control		0D8h
	Root Capabilities	Root Control		0DCh
	Root Status			0E0h
	PCI Express Device Capabilities 2			0E4h
	Device Status 2	Device Control 2		0E8h
	PCI Express Link Capabilities 2			0ECh
	Link Status 2	Link Control 2		0F0h
	Unimplemented Configuration Space (Returns `0x00000000`)			0F4h-0FCh
Always Enabled	Next Cap	Cap. Ver.	PCI Express Extended Cap. ID (AER)	100h
	Uncorrectable Error Status Register			104h
	Uncorrectable Error Mask Register			108h
	Uncorrectable Error Severity Register			10Ch
	Correctable Error Status Register			110h
	Correctable Error Mask Register			114h
	Advanced Error Cap. & Control Register			118h
	Header Log Register 1			11Ch
	Header Log Register 2			120h
	Header Log Register 3			124h
	Header Log Register 4			128h
	Reserved			12Ch

Table 2-24: **Common PCI Configuration Space Header *(Cont'd)***

	31		16 15	0	
Optional, Root Port only[3]	Root Error Command Register				130h
	Root Error Status Register				134h
	Error Source ID Register				138h
	Reserved				13Ch
Optional[3][4]	Next Cap	Cap. Ver.	PCI Express Extended Capability - Alternate Requester ID (ARI)		140h
	Control		Next Function	Function Groups	144h
	Reserved				148h-14Ch
Optional[3]	Next Cap	Cap. Ver.	PCI Express Extended Capability - DSN		150h
	PCI Express Device Serial Number (1st)				154h
	PCI Express Device Serial Number (2nd)				158h
	Reserved				15Ch
Optional[3]	Next Cap	Cap. Ver.	PCI Express Extended Capability - Power Budgeting Enhanced Capability Header		160h
	Reserved			DS	164h
	Reserved	Power Budget Data - State D0, D1, D3, ...			168h
	Power Budget Capability				16Ch
	Reserved				170h-1B4h
Optional[3]	Next Cap	Cap. Ver.	PCI Express Extended Capability ID - Latency Tolerance Reporting (LTR)		1B8h
	No-Snoop		Snoop		1BCh
Optional[3]	Next Cap	Cap. Ver.	PCI Express Extended Capability ID - Dynamic Power Allocation		1C0h
	Capability Register				1C4h
	Latency Indicator				1C8h
	Control		Status		1CCh
	Power Allocation Array Register 0				1D0h
	Power Allocation Array Register 1				1D4h
	Reserved				1D8h-1FCh

Table 2-24: **Common PCI Configuration Space Header *(Cont'd)***

<table>
<tr><th></th><th colspan="2">31   16</th><th colspan="2">15   0</th><th></th></tr>
<tr><td rowspan="17">Optional[3]</td><td>Next Cap</td><td>Cap. Ver.</td><td colspan="2">PCI Express Extended Capability ID - Single Root I/O Virtualization (SR-IOV)</td><td>200h</td></tr>
<tr><td colspan="4">Capability Register</td><td>204h</td></tr>
<tr><td colspan="2">SR-IOV Status (not supported)</td><td colspan="2">Control</td><td>208h</td></tr>
<tr><td colspan="2">Total VFs</td><td colspan="2">Initial VFs</td><td>20Ch</td></tr>
<tr><td colspan="2">Function Dependency Link</td><td colspan="2">Number VFs</td><td>210h</td></tr>
<tr><td colspan="2">VF Stride</td><td colspan="2">First VF Offset</td><td>214h</td></tr>
<tr><td colspan="2">VF Device ID</td><td colspan="2">Reserved</td><td>218h</td></tr>
<tr><td colspan="4">Supported Page Sizes</td><td>21Ch</td></tr>
<tr><td colspan="4">System Page Size</td><td>220h</td></tr>
<tr><td colspan="4">VF Base Address Register 0</td><td>224h</td></tr>
<tr><td colspan="4">VF Base Address Register 1</td><td>228h</td></tr>
<tr><td colspan="4">VF Base Address Register 2</td><td>22Ch</td></tr>
<tr><td colspan="4">VF Base Address Register 3</td><td>230h</td></tr>
<tr><td colspan="4">VF Base Address Register 4</td><td>234h</td></tr>
<tr><td colspan="4">VF Base Address Register 5</td><td>238h</td></tr>
<tr><td colspan="4">Reserved</td><td>23Ch</td></tr>
<tr></tr>
<tr><td></td><td colspan="4">Reserved</td><td>240h-270h</td></tr>
<tr><td rowspan="4">Optional[3]</td><td>Next Cap</td><td>Cap. Ver.</td><td colspan="2">PCI Express Extended Capability ID - Transaction Processing Hints (TPH)</td><td>274h</td></tr>
<tr><td colspan="4">Capability Register</td><td>278h</td></tr>
<tr><td colspan="4">Requester Control Register</td><td>27Ch</td></tr>
<tr><td colspan="2">Reserved</td><td>Steering Tag Upper</td><td>Steering Tag Lower</td><td>280h</td></tr>
<tr><td></td><td colspan="4">Reserved</td><td>284h - 2FCh</td></tr>
<tr><td rowspan="7">Optional[3]</td><td>Next Cap</td><td>Cap. Ver.</td><td colspan="2">PCI Express Extended Capability ID - Secondary PCIe Extended Capability</td><td>300h</td></tr>
<tr><td colspan="4">Lane Control (not supported)</td><td>304h</td></tr>
<tr><td colspan="3">Reserved</td><td>Lane Error Status</td><td>308h</td></tr>
<tr><td colspan="4">Lane Equalization Control Register 0</td><td>30Ch</td></tr>
<tr><td colspan="4">Lane Equalization Control Register 1</td><td>310h</td></tr>
<tr><td colspan="4">Lane Equalization Control Register 2</td><td>314h</td></tr>
<tr><td colspan="4">Lane Equalization Control Register 3</td><td>318h</td></tr>
<tr><td></td><td colspan="4">Reserved</td><td>31Ch-3BCh</td></tr>
</table>

Table 2-24: **Common PCI Configuration Space Header *(Cont'd)***

	31	16	15	0	
Optional[3]	Next Cap	Cap. Ver.	PCI Express Extended Capability - VC		3C0h
	Port VC Capability Register 1				3C4h
	Port VC Capability Register 2				3C8h
	Port VC Status		Port VC Control		3CCh
	VC Resource Capability Register 0				3D0h
	VC Resource Control Register 0				3D4h
	VC Resource Status Register 0				3D8h
	Reserved				400h-FFFh

Notes:

1. The MSI Capability Structure varies depending on the selections in the Vivado IDE.
2. Reserved for Endpoint configurations (returns `0x00000000`).
3. The layout of the PCI Express Extended Configuration Space (`100h-FFFh`) can change dependent on which optional capabilities are enabled. This table represents the Extended Configuration space layout when all optional extended capability structures, except RBAR, are enabled.
4. Enabled by default if the SR-IOV option is enabled.

Table 2-25: **Type 0 PCI Configuration Space Header**

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

Table 2-26: **Type 1 PCI Configuration Space Header**

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

Designing with the Core

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

Shared Logic

This feature allows you to share common logic across multiple instances of PCIe® blocks or with other cores, with certain limitations. Shared logic minimizes the HDL modifications needed by locating the logic to be shared to the top module of the design. It also enables additional ports on the top module to facilitate sharing. Shared logic is applicable only for Endpoint mode and not Root Port mode.

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

There are two types of logic sharing:

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

In both cases, the GT_COMMON block is shared.

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

Shared Logic in the Core

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

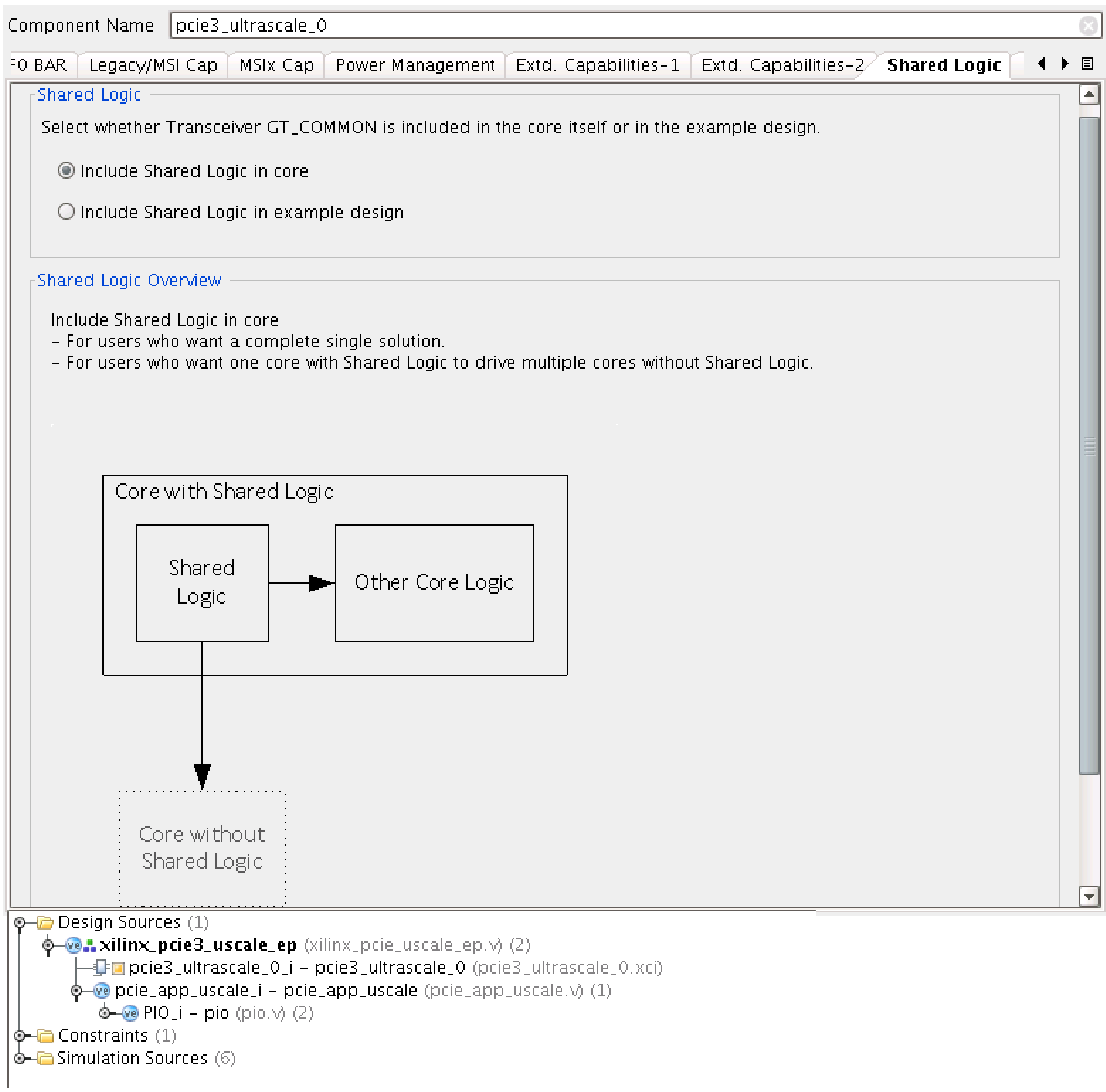

Figure 3-1: **Shared Logic in the Core**

Shared GT_COMMON

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

GT_CHANNEL instances can be used by other cores by sharing the same QPLL and GT_COMMON.

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

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

Shared GT_COMMON Use Case with GTH

Table 3-1: **Shared GT_COMMON Use Case**

GT – PCIe Max Link Speed	Device – PCIe Max Link Speed	Shared GT_COMMON
GTH	Kintex U (040) – PCIe gen3	PCIe pipe wrappers use QPLL for Gen3 and CPLL for Gen1/Gen2. If PCIe is Gen3 capable but operating at a lower speed, other IP can use it.

Limitations

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

- The settings of the GT_COMMON should not be changed because they are optimized for the PCIe core.

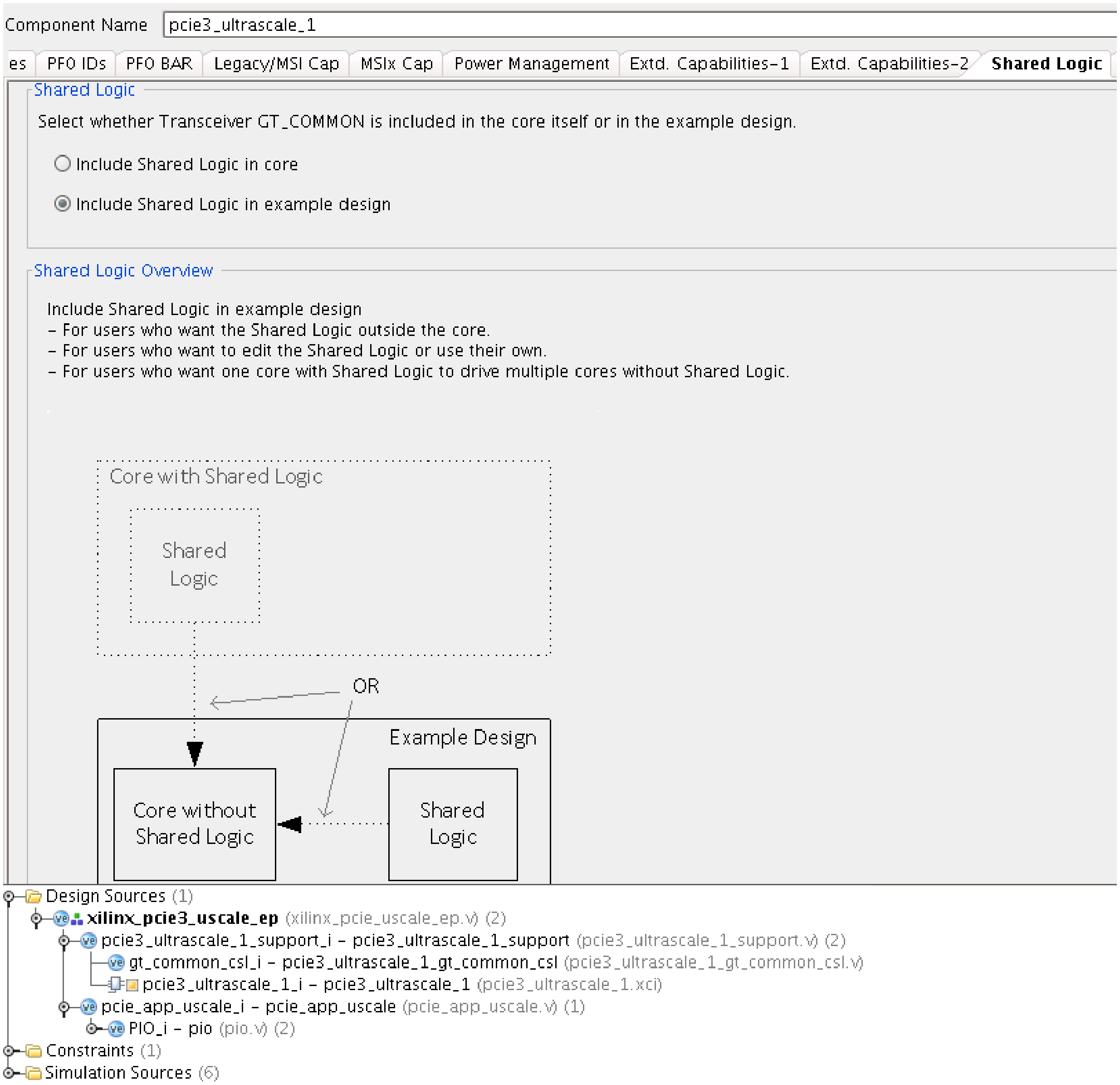

Figure 3-2: **Shared Logic in the Example Design**

Clocking

The input system clock signal of the Gen3 Integrated Block for PCIe core is called `ref_clk`. The core requires a 100 MHz clock input. For more information, see the Answer Records at the Xilinx PCI Express Solution Center.

In a typical PCI Express solution, the PCI Express reference clock is a spread spectrum clock (SSC), provided at 100 MHz. In most commercial PCI Express systems, SSC cannot be disabled. For more information regarding SSC and PCI Express, see Section 4.3.7.1.1 of the *PCI Express Base Specification, rev. 3.0* [Ref 2].

IMPORTANT: *All add-in card designs must use synchronous clocking due to the characteristics of the provided reference clock. For devices using the Slot clock, the* ***Slot Clock Configuration*** *setting in the Link Status Register must be enabled in the Vivado IP catalog. See Clocking Requirements, page 82 for additional information regarding reference clock requirements.*

Each link partner device shares the same clock source. Figure 3-3 and Figure 3-4 show a system using a 100 MHz reference clock.

Even if the device is part of an embedded system, if the system uses commercial PCI Express root complexes or switches along with typical motherboard clocking schemes, synchronous clocking should still be used.

Note: Figure 3-3 and Figure 3-4 are high-level representations of the board layout. Ensure that coupling, termination, and details are correct when laying out a board.

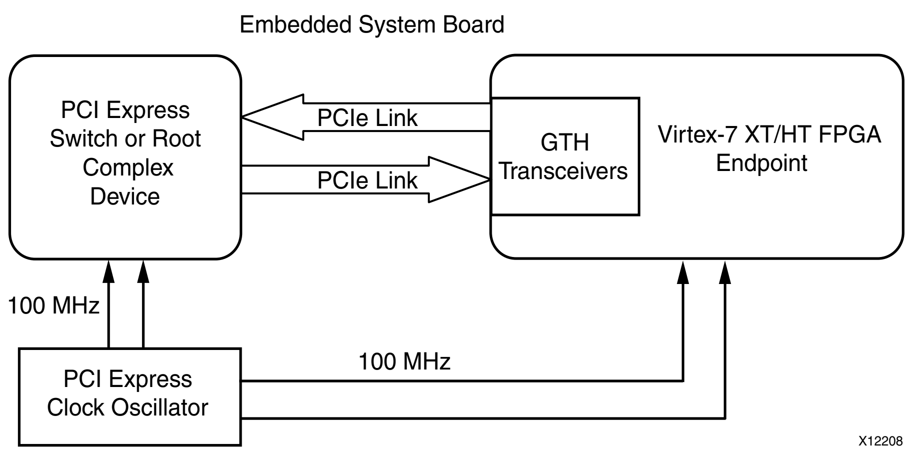

Figure 3-3: **Embedded System Using 100 MHz Reference Clock**

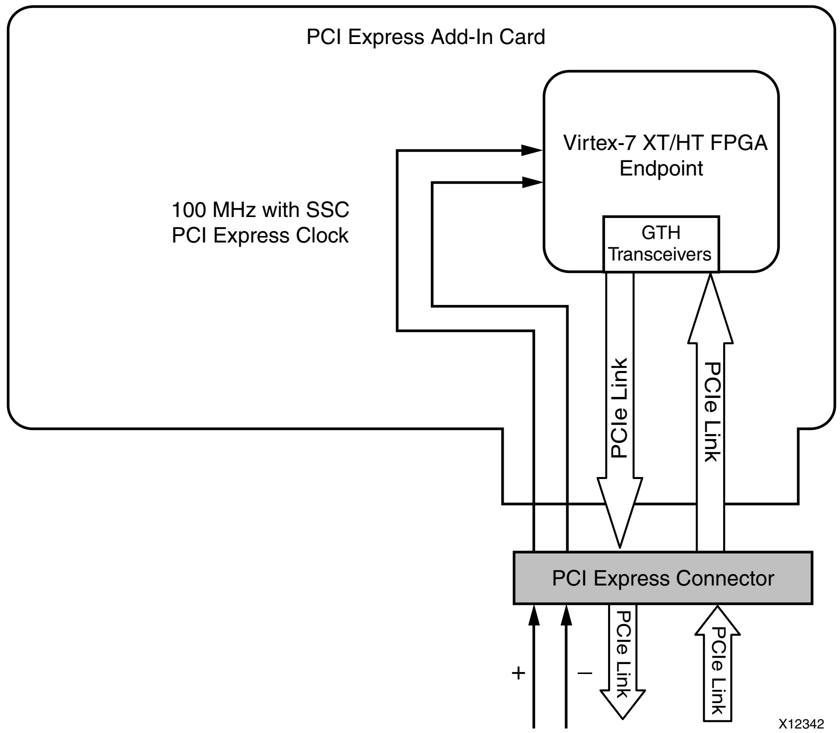

Figure 3-4: **Open System Add-In Card Using 100 MHz Reference Clock**

Resets

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

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

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

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

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

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

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

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

AXI4-Stream Interface Description

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

Overview of Features

Figure 3-5 illustrates the client-side interface of the core.

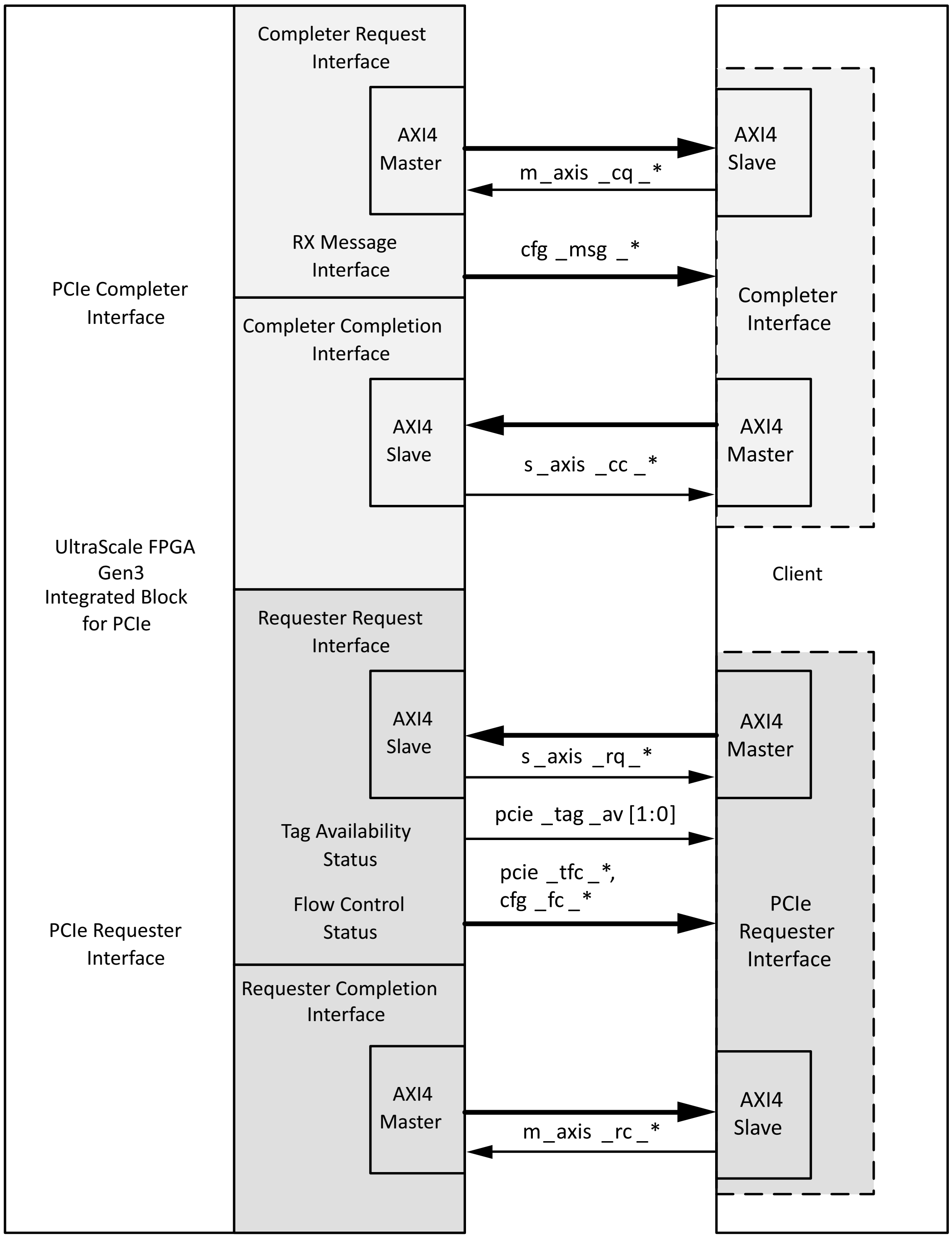

Figure 3-5: **Block Diagram of UltraScale FPGA Gen3 Integrated Block Client Interfaces**

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

- A PCIe Completer reQuest (CQ) interface through which requests arriving from the link are delivered to the user application.
- A PCIe Completer Completion (CC) interface through which the user application can send back responses to the completer requests. The user application can process all

Non-Posted transactions as split transactions. That is, it can continue to accept new requests on the completer request interface while sending a completion for a request.

- A PCIe Requester reQuest (RQ) interface through which the user application can generate requests to remote PCIe devices attached to the link.
- A PCIe Requester Completion (RC) interface through which the integrated block returns the completions received from the link (in response to the user application requests as PCIe requester) to the user application.

Each of the four interfaces is based on the AMBA4® AXI4-Stream Protocol Specification [Ref 1]. The width of these interfaces can be configured as 64, 128, or 256 bytes, and the user clock frequencies can be selected as 62.5, 125, or 250 MHz, depending on the number of lanes and PCIe generation you choose. Table 3-2 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 two interfaces through which status information is communicated to the PCIe master side of the user application:

- A flow control status interface that provides information on currently available transmit credit, so that the user application can schedule requests based on available credit.
- A tag availability status interface that provides information on the number of tags available to assign to Non-Posted requests, so that the user application can schedule requests without the risk of being blocked by all tags being in use within the PCIe controller.

Finally, the integrated block also has a received-message interface which indicates to the user logic when a message is received from the link, rather than transferring the entire message over the CQ interface.

Table 3-2: **Data Width and Clock Frequency Settings for the Client Interfaces**

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

Table 3-2: **Data Width and Clock Frequency Settings for the Client Interfaces *(Cont'd)***

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

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 client), the integrated block extracts parameters from the headers of received TLP and constructs the descriptors for delivering to the user application. Each TLP is transferred as a packet, as defined in the AXI4-Stream Interface Protocol.

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

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

   $n = A \bmod w$

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

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

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

The Vivado IP catalog applies the data alignment option globally to all four interfaces. However, advanced users can select the alignment mode independently for each of the four AXI4-Stream interfaces. This is done by setting the corresponding alignment mode parameter, with the constraint that the Requester Completion (RC) interface can be set to the address-aligned mode. See Interface Operation, page 82 for more details on address alignment and example diagrams.

Straddle Option on Requester Completion Interface

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

- Restrict the number of outstanding Non Posted requests, so as to keep the total number of completions received less than 64 and within the completion of the FIFO size selected, or
- Use the RC interface straddle option. See Figure 3-59 for waveforms showing this option.

The straddle option, available only on the 256-bit wide RC interface, is enabled through the Vivado IP catalog. See Chapter 4, Design Flow Steps for instructions on enabling the option in the IP catalog. When this option is enabled, the integrated block can start a new Completion TLP on byte lane 16 when the previous TLP has ended at or before byte lane 15 in the same beat. Thus, with this option enabled, it is possible for the integrated block to send two Completion TLPs entirely in the same beat on the RC interface, if neither of them has more than one Dword of payload.

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

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

Table 3-3: **Valid Combinations of Interface Width, Alignment Mode, and Straddle**

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

Receive Transaction Ordering

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

- Posted requests must be able to pass Non-Posted requests on the Completer reQuest (CQ) interface. To enable this capability, the integrated block implements a flow control mechanism on the CQ interface through which user logic can control the flow of Non-Posted requests without affecting Posted requests. The user logic signals the availability of a buffer to receive a Non-Posted request by asserting the `pcie_cq_np_req` 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 backpressure is applied by the credit mechanism for the delivery of Non-Posted requests, the integrated block delivers Posted and Non-Posted requests in the same order as received from the link. For more information on controlling the flow of Non-Posted requests, see Selective Flow Control for Non-Posted Requests, page 100.

- 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 Requestor 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 client side, it is the responsibility of the user application to enforce ordering constraints whenever needed.

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 re-order 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 assert `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.

Clocking Requirements

All client interface signals of the core are timed with respect to the user clock (`user_clk`), which can have a frequency of 62.5, 125, or 250 MHz, depending on the link speed and link width configured (see Table 3-2).

Interface Operation

This section describes the operation of the client-side interfaces of the core.

Completer Interface

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

Completer Request Descriptor Formats

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

The formats of the descriptor for different request types are illustrated in Figure 3-6, Figure 3-7, Figure 3-8, and Figure 3-9. The format of Figure 3-6 applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request. The format of Figure 3-7 is used for Vendor-Defined Messages (Type 0 or Type 1) only. The format of Figure 3-8 is used for all ATS messages (Invalid Request, Invalid Completion, Page Request, PRG Response). For all other messages, the descriptor takes the format of Figure 3-9.

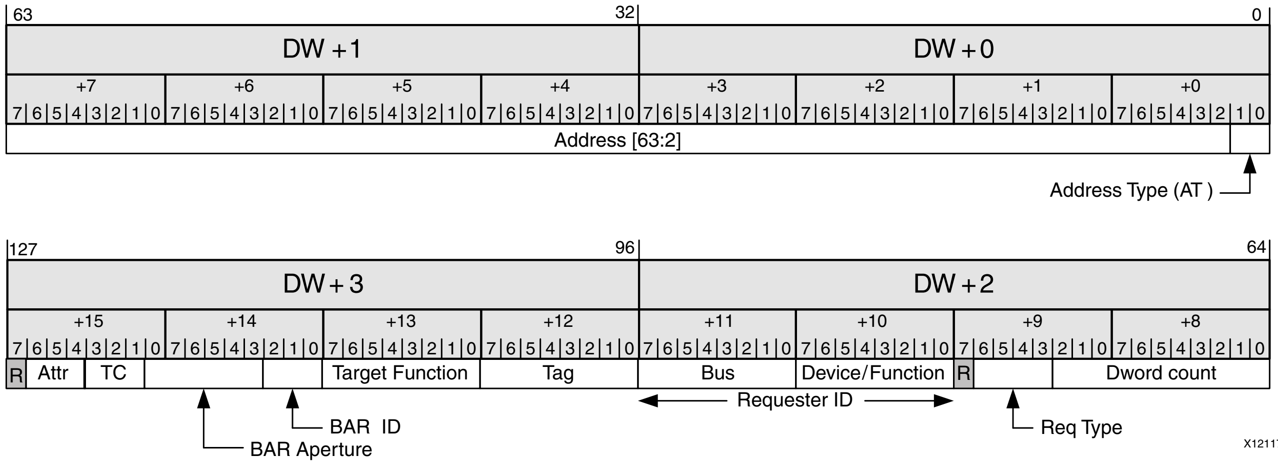

Figure 3-6: **Completer Request Descriptor Format for Memory, I/O, and Atomic Op Requests**

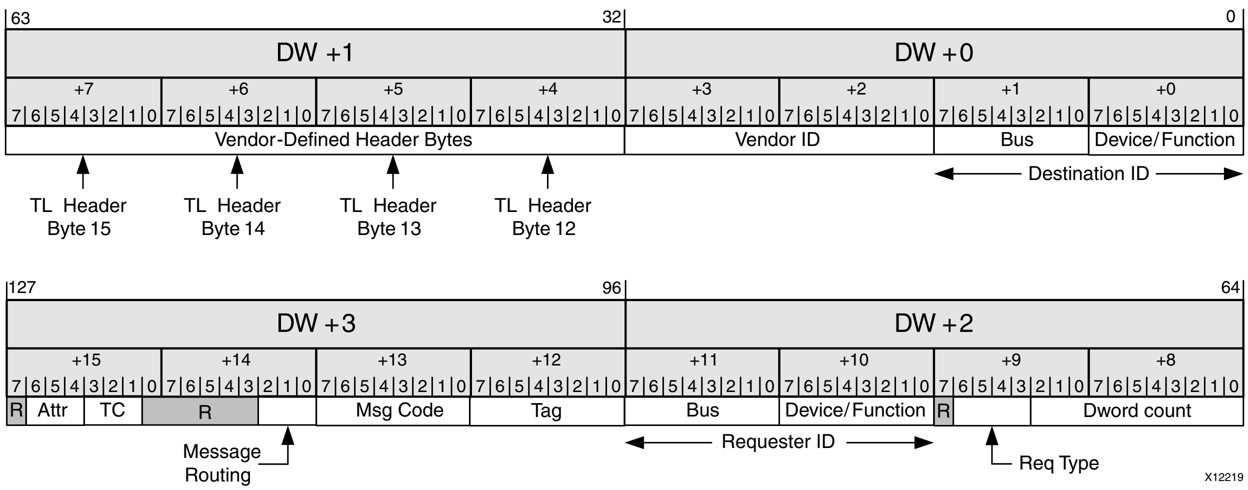

Figure 3-7: **Completer Request Descriptor Format for Vendor-Defined Messages**

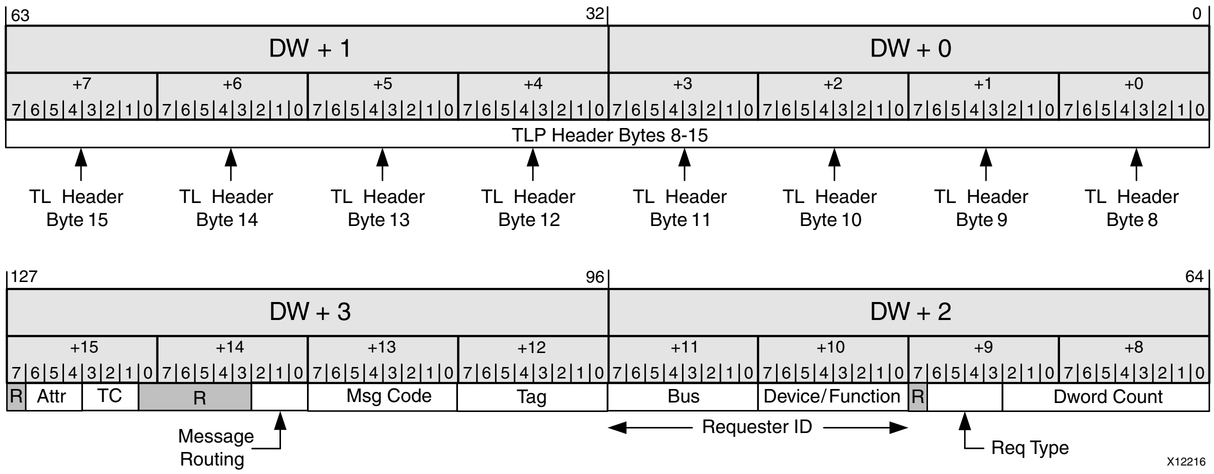

Figure 3-8: **Completer Request Descriptor Format for ATS Messages**

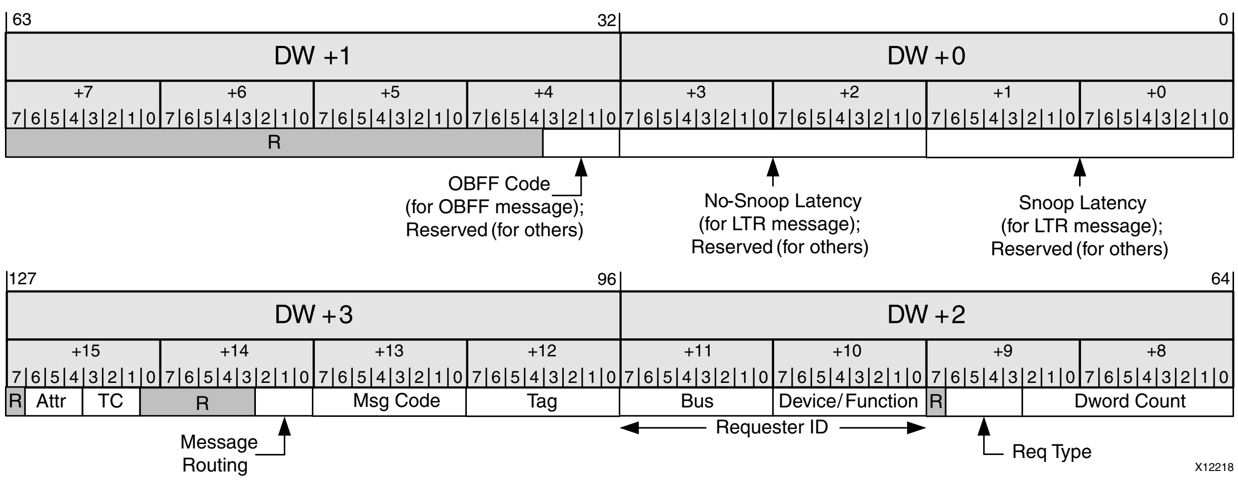

Figure 3-9: **Completer Request Descriptor Format for All Other Messages**

Table 3-4 describes the individual fields of the completer request descriptor.

Table 3-4: **Completer Request Descriptor Fields**

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

Table 3-4: **Completer Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
63:2	Address	This field applies to memory, I/O, and Atomic Op requests. It provides the address from the TLP header. This is the address of the first Dword referenced by the request. The `First_BE` bits from `m_axis_cq_tuser` must be used to determine the byte-level address. When the transaction specifies a 32-bit address, bits [63:32] of this field are 0.
74:64	Dword Count	These 11 bits indicate the size of the block (in Dwords) to be read or written (for messages, size of the message payload). Its range is 0 - 256 Dwords. For I/O accesses, the Dword count is always 1. For a zero length memory read/write request, the Dword count is 1, with the First_BE bits set to all 0s.
78:75	Request Type	Identifies the transaction type. The transaction types and their encodings are listed in Table 3-5.
95:80	Requester ID	PCI Requester ID associated with the request. With legacy interpretation of RIDs, these 16 bits are divided into an 8-bit bus number [95:88], 5-bit device number [87:83], and 3-bit Function number [82:80]. When ARI is enabled, bits [95:88] carry the 8-bit bus number and [87:80] provide the Function number. When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the integrated block with the completion data.
103:96	Tag	PCIe Tag associated with the request. When the request is a Non-Posted transaction, the user logic must store this field and supply it back to the integrated block with the completion data. This field can be ignored for memory writes and messages.
111:104	Target Function	This field is defined for memory, I/O, and Atomic Op requests only. It provides the Function number the request is targeted at, determined by the BAR check. When ARI is in use, all 8 bits of this field are valid. Otherwise, only bits [106:104] are valid. Following are Target Function Value to PF/VF map mappings: • 0: PF0 • 1: PF1 • 64: VF0 • 65: VF1 • 66: VF2 • 67: VF3 • 68: VF4 • 69: VF5

Table 3-4: **Completer Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
114:112	BAR ID	This field is defined for memory, I/O, and Atomic Op requests only. It provides the matching BAR number for the address in the request. • 000: BAR 0 (VF-BAR 0 for VFs) • 001: BAR 1 (VF-BAR 1 for VFs) • 010: BAR 2 (VF-BAR 2 for VFs) • 011: BAR 3 (VF-BAR 3 for VFs) • 100: BAR 4 (VF-BAR 4 for VFs) • 101: BAR 5 (VF-BAR 5 for VFs) • 110: Expansion ROM Access For 64-bit transactions, the BAR number is given as the lower address of the matching pair of BARs (that is, 0, 2, or 4).
120:115	BAR Aperture	This 6-bit field is defined for memory, I/O, and Atomic Op requests only. It provides the aperture setting of the BAR matching the request. This information is useful in determining the bits to be used in addressing its memory or I/O space. For example, a value of 12 indicates that the aperture of the matching BAR is 4K, and the user application can therefore ignore bits [63:12] of the address. For VF BARs, the value provided on this output is based on the memory space consumed by a single VF covered by the BAR.
123:121	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request. When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the integrated block with the completion data.
126:124	Attributes	These bits provide the setting of the Attribute bits associated with the request. Bit 124 is the No Snoop bit and bit 125 is the Relaxed Ordering bit. Bit 126 is the ID-Based Ordering bit, and can be set only for memory requests and messages. When the request is a Non-Posted transaction, the user completer application must store this field and supply it back to the integrated block with the completion data.
15:0	Snoop Latency	This field is defined for LTR messages only. It provides the value of the 16-bit Snoop Latency field in the TLP header of the message.
31:16	No-Snoop Latency	This field is defined for LTR messages only. It provides the value of the 16-bit No-Snoop Latency field in the TLP header of the message.
35:32	OBFF Code	This field is defined for OBFF messages only. The OBFF Code field is used to distinguish between various OBFF cases: • 1111b: CPU Active – System fully active for all device actions including bus mastering and interrupts • 0001b: OBFF – System memory path available for device memory read/write bus master activities • 0000b: Idle – System in an idle, low power state All other codes are reserved.

Table 3-4: **Completer Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
111:104	Message Code	This field is defined for all messages. It contains the 8-bit Message Code extracted from the TLP header. Appendix F of the *PCI Express Base Specification, rev. 3.0* [Ref 2] provides a complete list of the supported Message Codes.
114:112	Message Routing	This field is defined for all messages. These bits provide the 3-bit Routing field r[2:0] from the TLP header.
15:0	Destination ID	This field applies to Vendor-Defined Messages only. When the message is routed by ID (that is, when the Message Routing field is 010 binary), this field provides the Destination ID of the message.
63:32	Vendor-Defined Header	This field applies to Vendor-Defined Messages only. It contains the bytes extracted from Dword 3 of the TLP header.
63:0	ATS Header	This field is applicable to ATS messages only. It contains the bytes extracted from Dwords 2 and 3 of the TLP header.

Table 3-5: **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 Request Interface Operation

Figure 3-10 illustrates the signals associated with the completer request interface of the core. The core delivers each TLP on this interface as an AXI4-Stream packet. The packet starts with a 128-bit descriptor, followed by data in the case of TLPs with a payload.

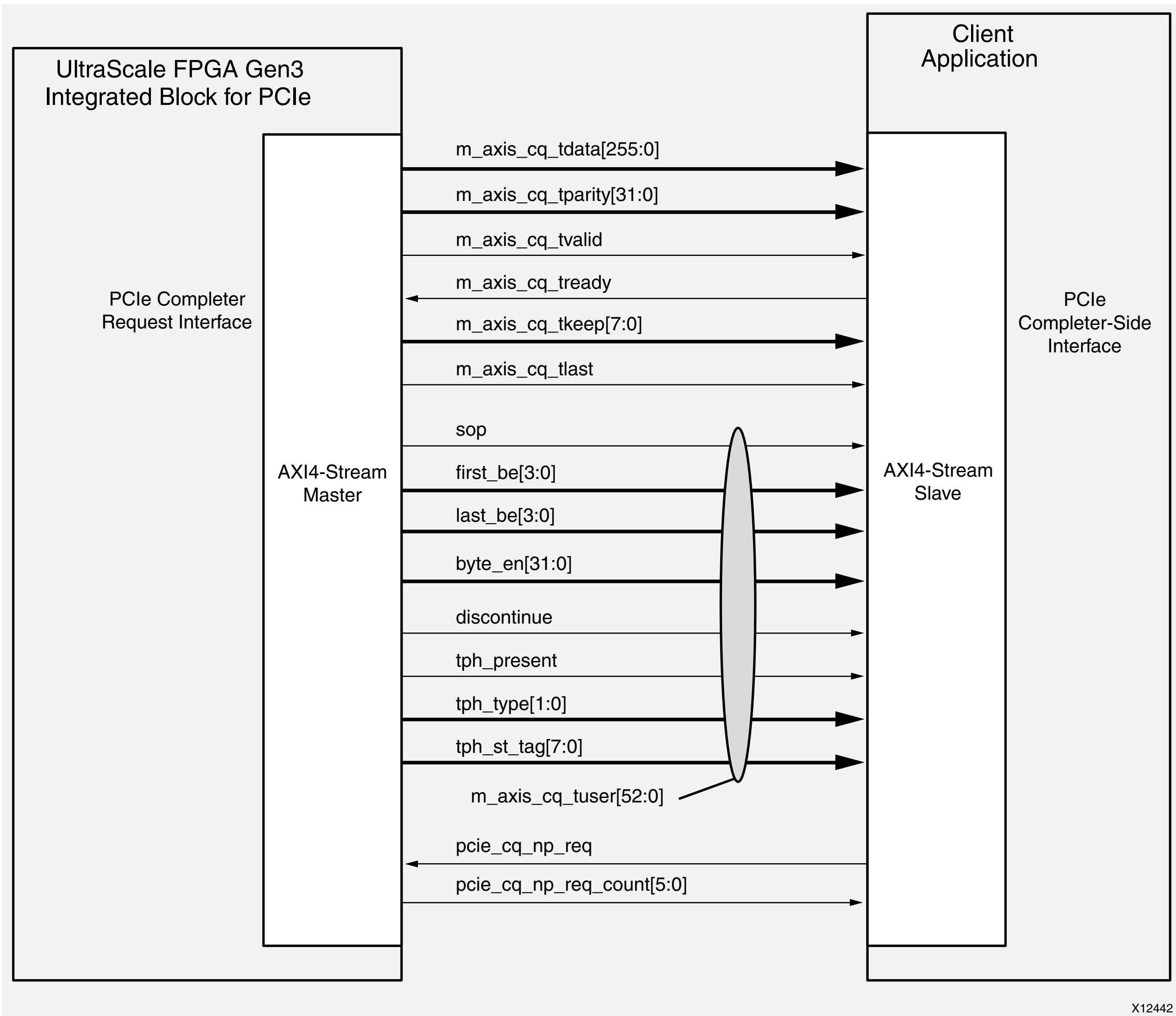

Figure 3-10: **Completer Request Interface Signals**

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
- 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 Memory Write Operation

The timing diagrams in Figure 3-11, Figure 3-12, and Figure 3-13 illustrate the Dword-aligned transfer of a memory write TLP received from the link across the Completer reQuest (CQ) interface, when the interface width is configured as 64, 128, and 256 bits, respectively. For illustration purposes, the starting Dword address of the data block being written into memory is assumed to be (m * 32 + 1), for an integer $m > 0$. Its size is assumed to be n Dwords, for some $n = k * 32 + 29$, $k > 0$.

In both Dword-aligned and address-aligned modes, the transfer starts with the 16 descriptor bytes, followed immediately by the payload bytes. The `m_axis_cq_tvalid` signal remains asserted over the duration of the packet. You can prolong a beat at any time by deasserting `m_axis_cq_tready`. The AXI4-Stream interface signals `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 client side of the interface, or to support additional features. The `sop` signal is asserted in the first beat of every packet, when its descriptor is on the bus. The byte enable outputs `byte_en[31:0]` (one per byte lane) indicate the valid bytes in the payload. The bits of `byte_en` are asserted only when a valid payload byte is in the corresponding lane (that is, not asserted for descriptor or padding bytes between the descriptor and payload). The asserted byte enable bits are always contiguous from the start of the payload, except when the payload size is two Dwords or less. For cases of one-Dword and two-Dword writes, the byte enables can be non-contiguous. Another special case is that of a zero-length memory write, when the integrated block transfers a one-Dword payload with all `byte_en` bits set to 0. Thus, in all cases the user logic can use the `byte_en` signals directly to enable the writing of the associated bytes into memory.

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

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

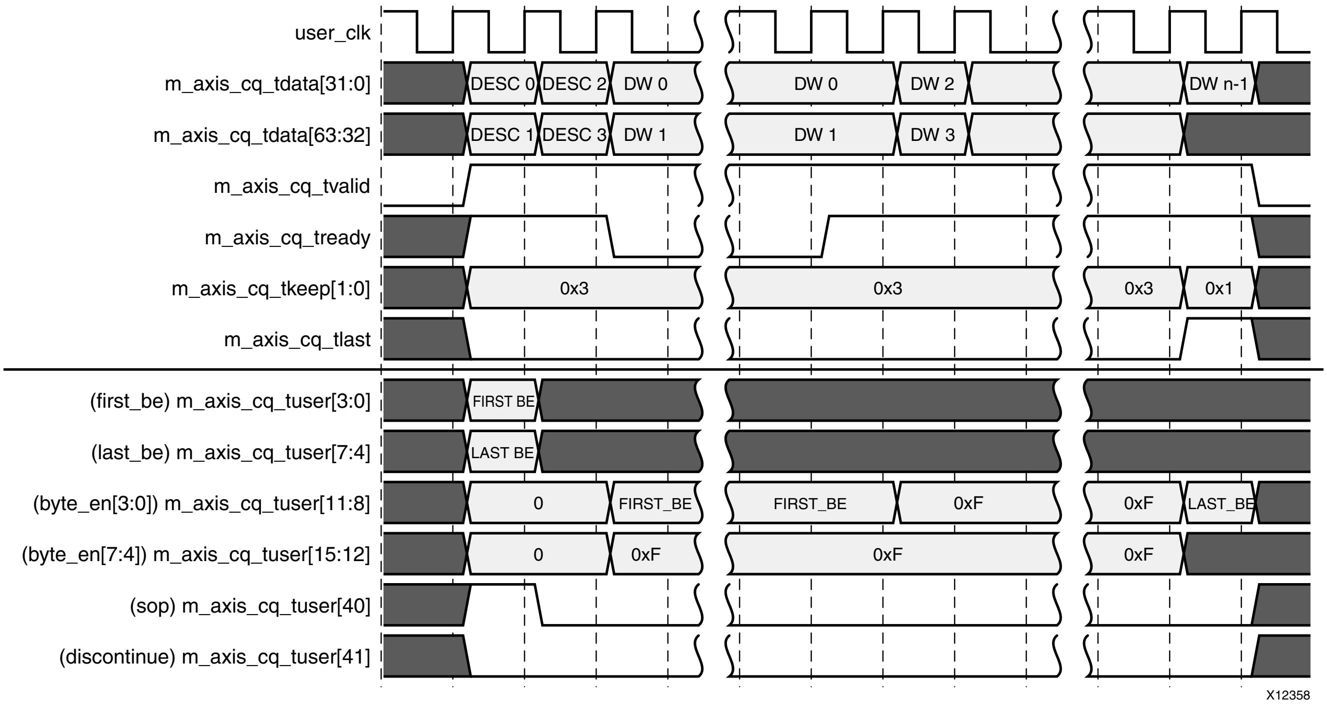

Figure 3-11: **Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode, Interface Width = 64 Bits)**

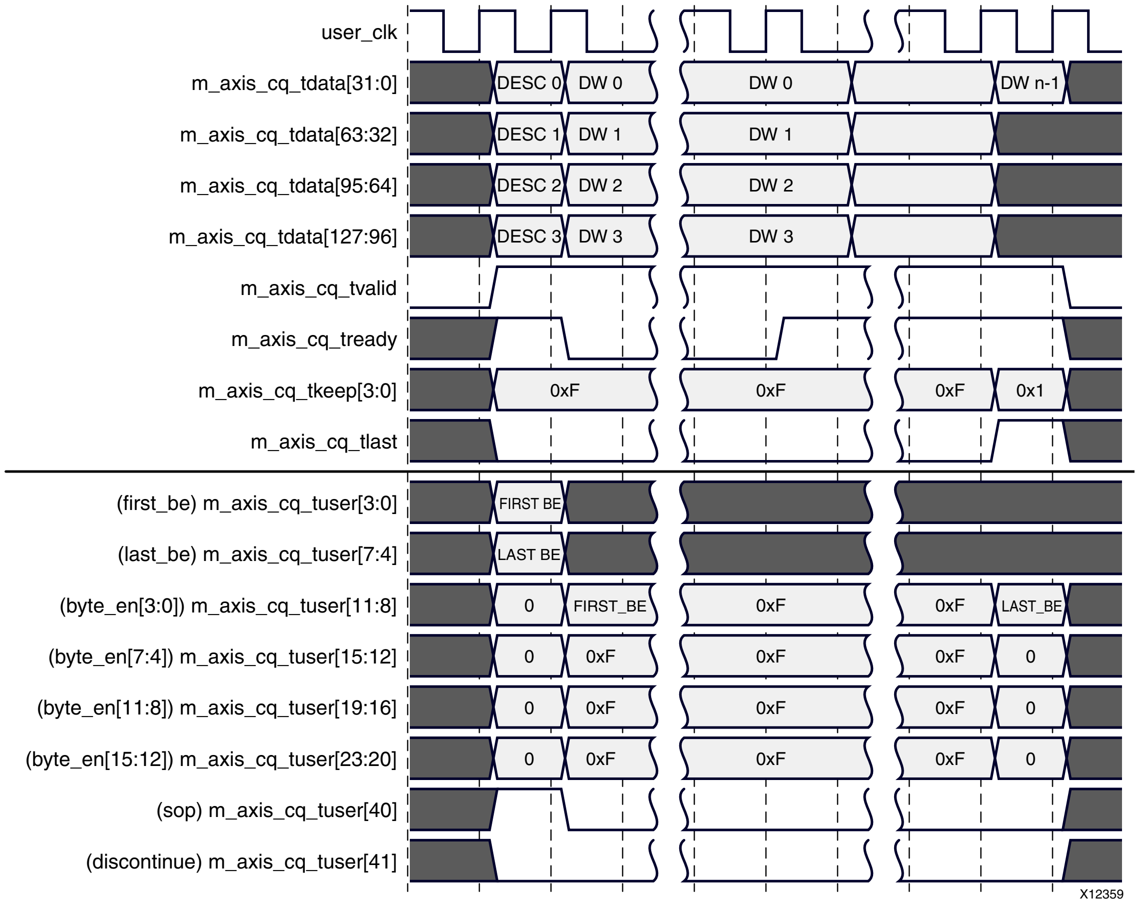

Figure 3-12: **Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode, Interface Width = 128 Bits)**

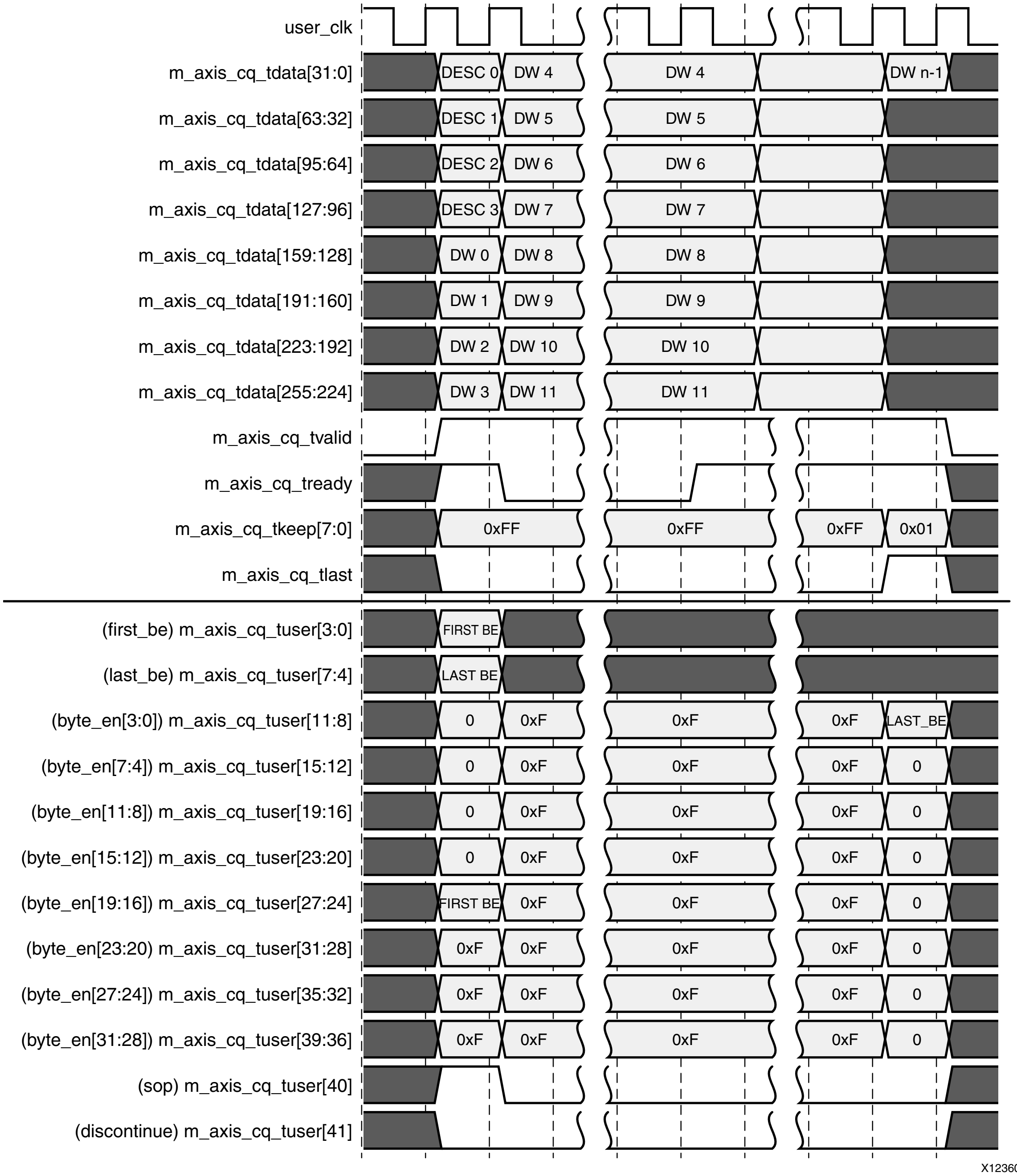

Figure 3-13: **Memory Write Transaction on the Completer Request Interface (Dword-Aligned Mode, Interface Width = 256 Bits)**

The timing diagrams in Figure 3-14, Figure 3-15, and Figure 3-16 illustrate the address-aligned transfer of a memory write TLP received from the link across the CQ interface, when the interface width is configured as 64, 128 and 256 bits, respectively. For the purpose of illustration, the starting Dword address of the data block being written into memory is assumed to be (m * 32 + 1), for an integer m > 0. Its size is assumed to be n Dwords, for some n = k * 32 + 29, k > 0.

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

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

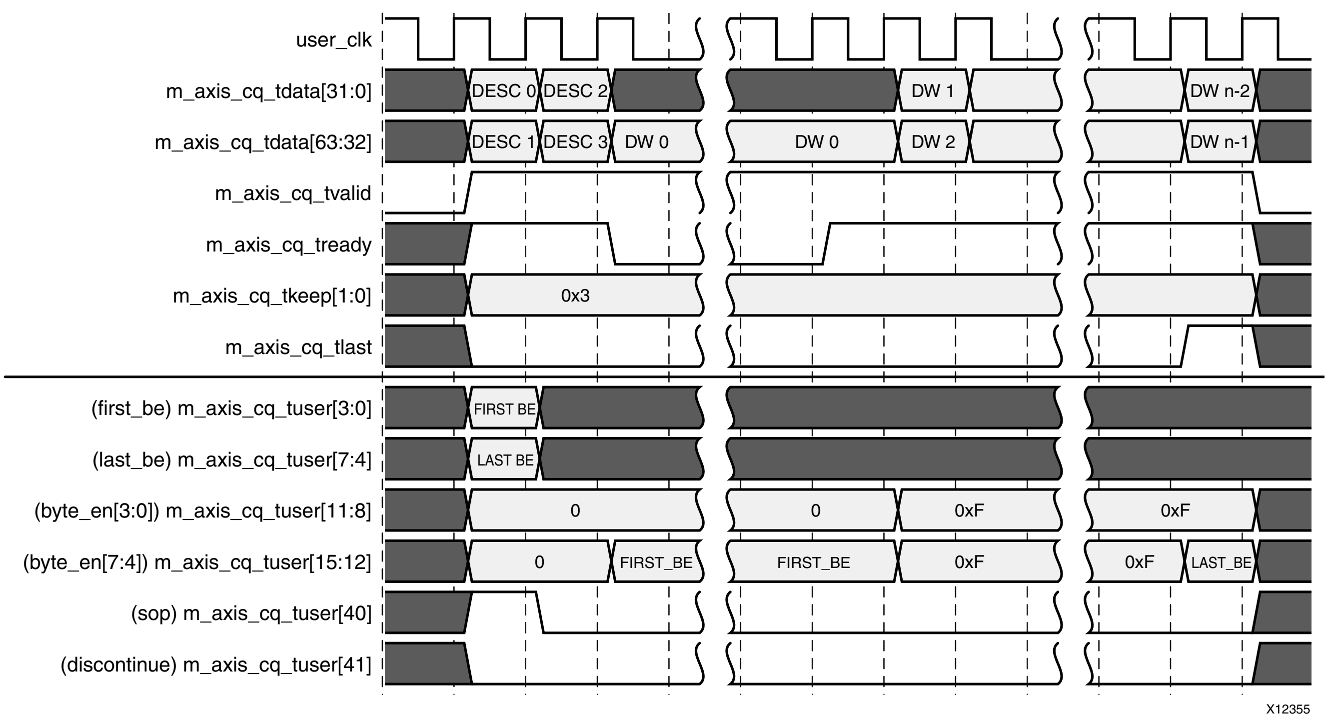

Figure 3-14: **Memory Write Transaction on the Completer Request Interface (Address-Aligned Mode, Interface Width = 64 Bits)**

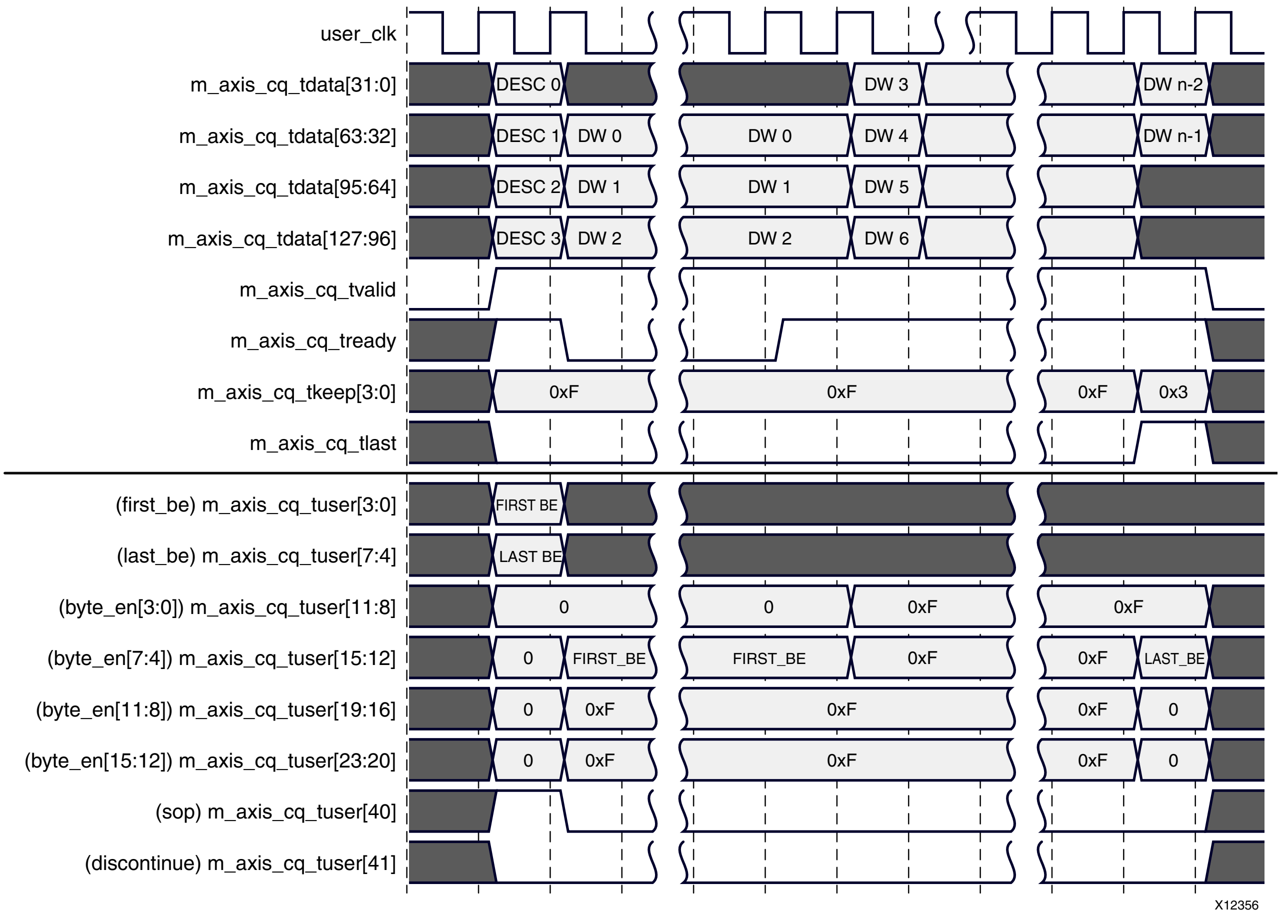

Figure 3-15: **Memory Write Transaction on the Completer Request Interface (Address-Aligned Mode, Interface Width = 128 Bits)**

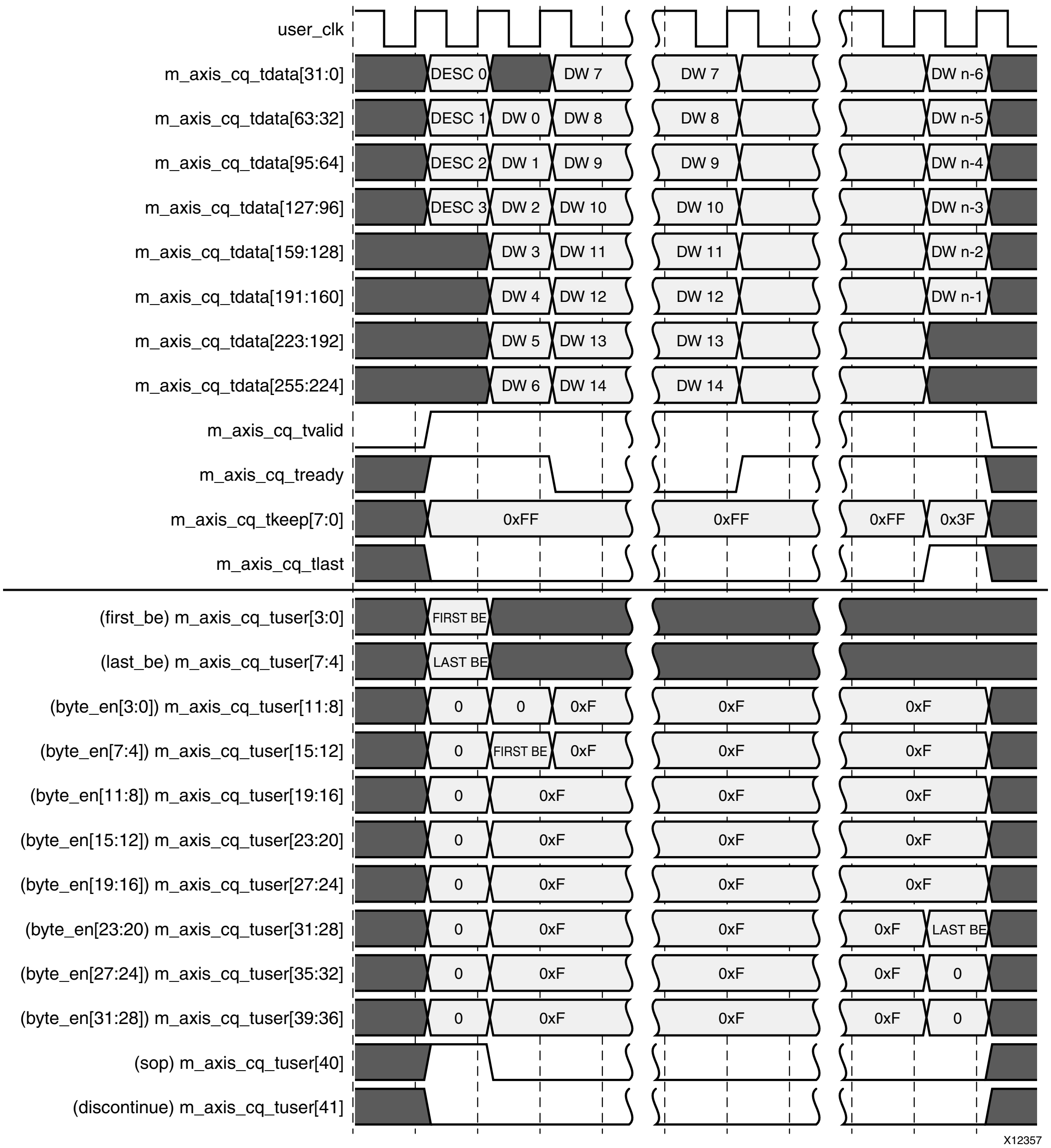

Figure 3-16: **Memory Write Transaction on the Completer Request Interface (Address-Aligned Mode, Interface Width = 256 Bits)**

Completer Memory Read Operation

A memory read request is transferred across the completer request interface in the same manner as a memory write request, except that the AXI4-Stream packet contains only the 16-byte descriptor. The timing diagrams in Figure 3-17, Figure 3-18, and Figure 3-19 illustrate the transfer of a memory read TLP received from the link across the completer request interface, when the interface width is configured as 64, 128, and 256 bits, respectively. The packet occupies two consecutive beats on the 64-bit interface, while it is transferred in a single beat on the 128- and 256-bit interfaces. The `m_axis_cq_tvalid` signal remains asserted over the duration of the packet. You can prolong a beat at any time by deasserting `m_axis_cq_tready`. The `sop` signal in the `m_axis_cq_tuser` bus is asserted when the first descriptor byte is on the bus.

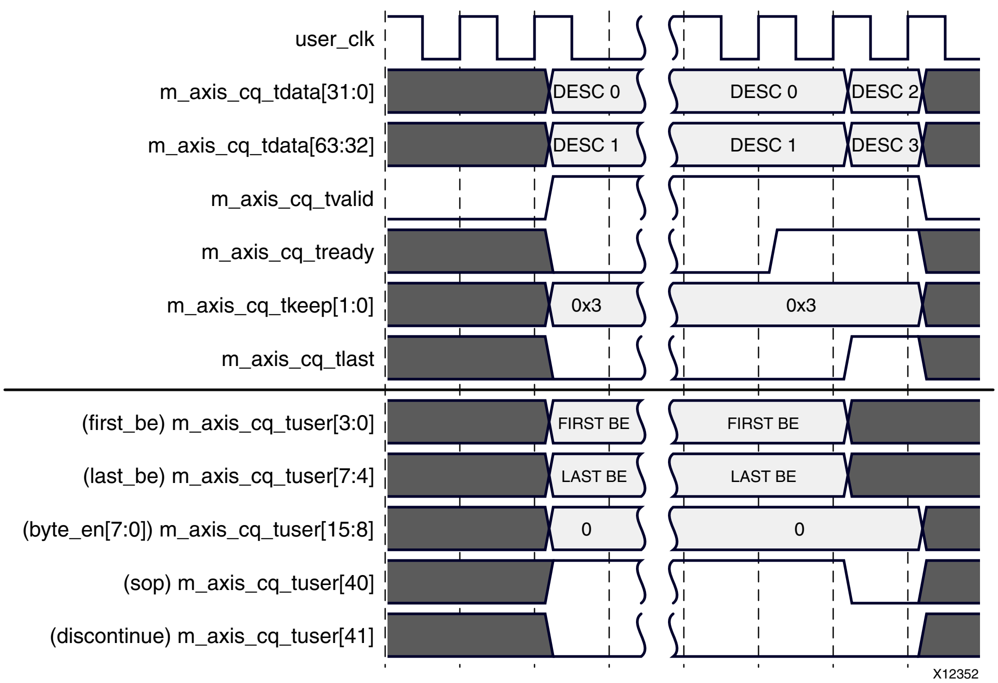

Figure 3-17: **Memory Read Transaction on the Completer Request Interface (Interface Width = 64 Bits)**

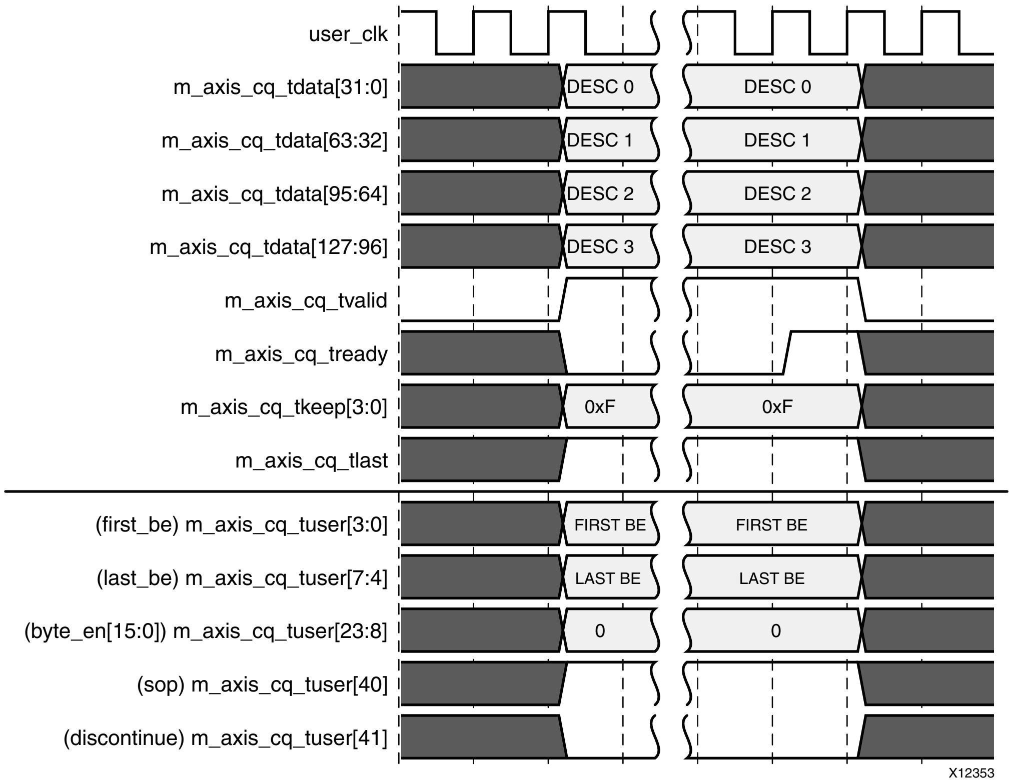

Figure 3-18: **Memory Read Transaction on the Completer Request Interface (Interface Width = 128 Bits)**

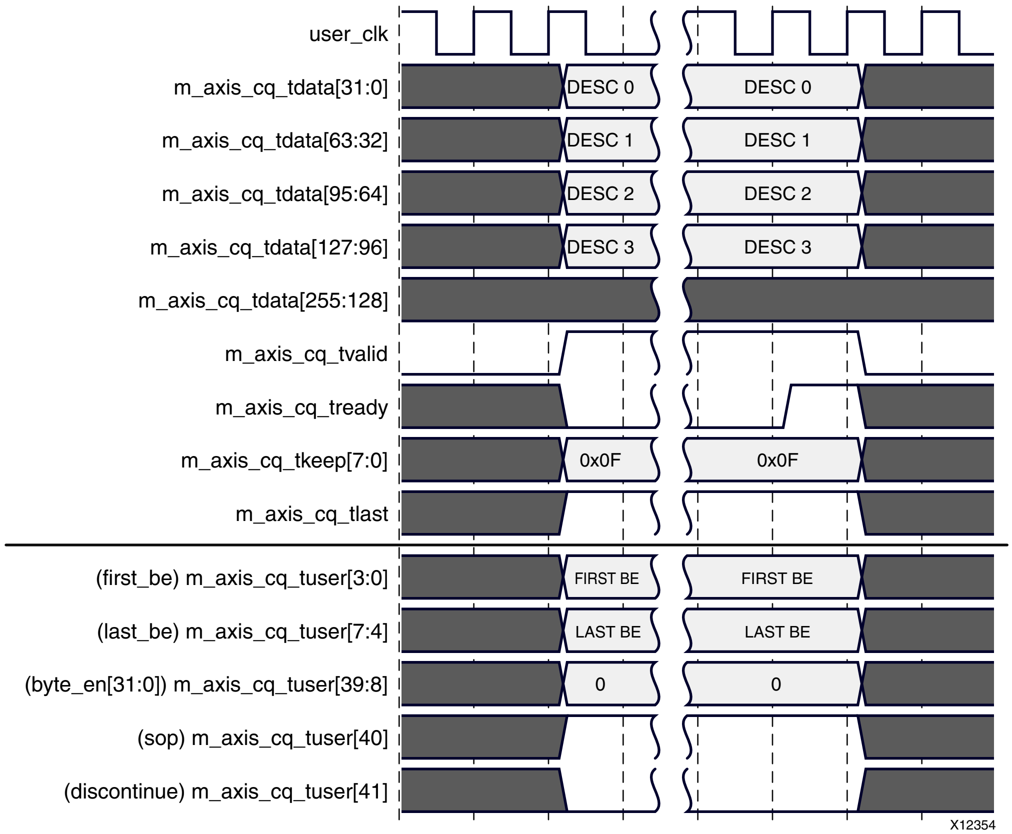

Figure 3-19: **Memory Read Transaction on the Completer Request Interface (Interface Width = 256 Bits)**

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

The user application must respond to each memory read request with a Completion. The data requested by the read can be sent as a single Completion or multiple Split Completions. These Completions must be sent through the Completer Completion (CC) interface of the integrated block. The Completions for two distinct requests can be sent in any order, but the Split Completions for the same request must be in order. The operation of the CC interface is described in Completer Completion Interface Operation, page 101.

I/O Write Operation

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

Because an I/O write is a Non-Posted transaction, the user logic must respond to it with a Completion containing no data payload. The Completions for I/O requests can be sent in any order. Errors associated with the I/O write transaction can be signaled to the requester by setting the Completion Status field in the completion descriptor to CA (Completer Abort) or UR (Unsupported Request), as is appropriate. The operation of the Completer Completion interface is described in Completer Completion Interface Operation, page 101.

I/O Read Operation

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

The user logic must respond to an I/O read request with a one-Dword Completion (or a Completion with no data in the case of an error). The Completions for two distinct I/O read requests can be sent in any order. Errors associated with an I/O read transaction can be signaled to the requester by setting the Completion Status field in the completion descriptor to CA (Completer Abort) or UR (Unsupported Request), as is appropriate. The operation of the Completer Completion interface is described in Completer Completion Interface Operation, page 101.

Atomic Operations on the Completer Request Interface

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

Because an Atomic Operation is a Non-Posted transaction, the user logic must respond to it with a Completion containing the result of the operation. Errors associated with the operation can be signaled to the requester by setting the Completion Status field in the completion descriptor to Completer Abort (CA) or Unsupported Request (UR), as is appropriate. The operation of the Completer Completion interface is described in Completer Completion Interface Operation, page 101.

Message Requests on the Completer Request Interface

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

Aborting a Transfer

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

Selective Flow Control for Non-Posted Requests

The *PCI Express Base Specification* [Ref 2] requires that the Completer Request interface continue to deliver Posted transactions even when the user application is unable to accept Non-Posted transactions. To enable this capability, the integrated block implements a credit-based flow control mechanism on the CQ interface through which user logic can control the flow of Non-Posted requests without affecting Posted requests. The user logic signals the availability of buffers for receive Non-Posted requests using the `pcie_cq_np_req` 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 backpressure 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` is 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 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 each time it is ready to receive a Non-Posted request, or keep it permanently asserted if it does not need to exercise selective backpressure 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 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 input and updating the pcie_cq_np_req_count output in response. Thus, when the user application has adequate buffer space available, it should provide the credit in advance so that Non-Posted requests are not held up by the core for lack of credit.

Completer Completion Interface Operation

Figure 3-20 illustrates the signals associated with the completer completion interface of the core. The core delivers each TLP on this interface as an AXI4-Stream packet.

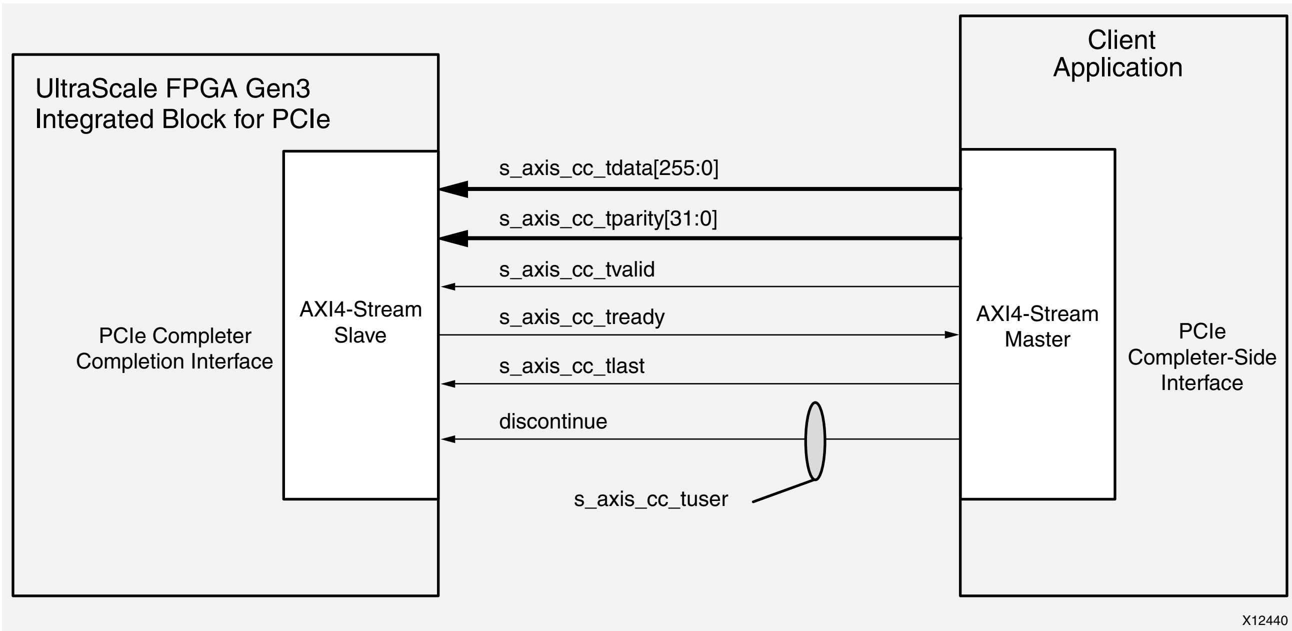

Figure 3-20: **Completer Completion Interface Signals**

The core delivers each TLP on the Completer Completion (CC) interface as an AXI4-Stream packet. The packet starts with a 96-bit descriptor, followed by data in the case of Completions with a payload.

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

Completer Completion Descriptor Format

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

The format of the completer completion descriptor is illustrated in Figure 3-21. The individual fields of the completer request descriptor are described in Table 3-6.

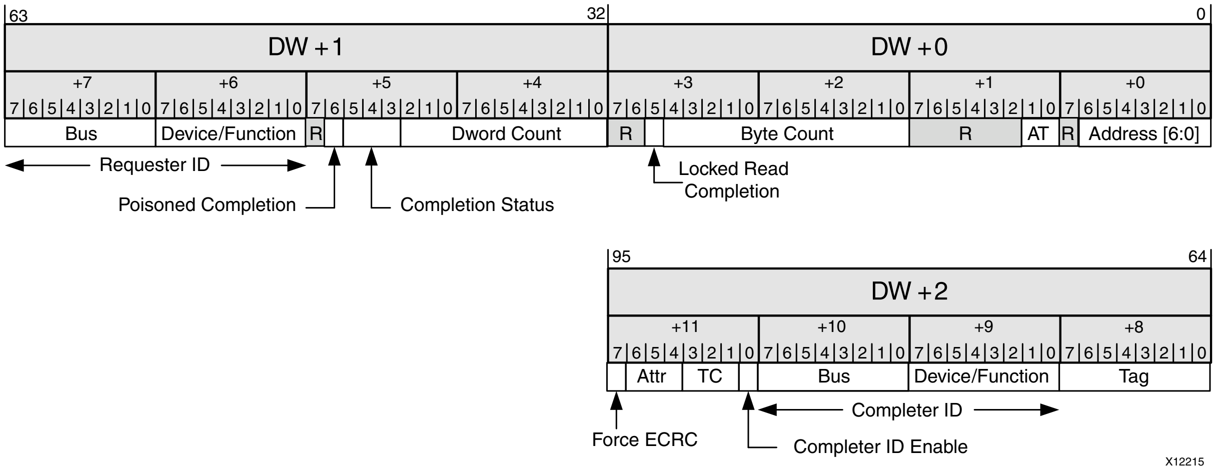

Figure 3-21: **Completer Completion Descriptor Format**

Table 3-6: **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 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.
28:16	Byte Count	These 13 bits can have values in the range of 0 – 4096 bytes. If a Memory Read Request is completed using a single Completion, the Byte Count value indicates Payload size in bytes. This field must be set to 4 for I/O read Completions and I/O write Completions. The byte count must be set to 1 while sending a Completion for a zero-length memory read, and a dummy payload of 1 Dword must follow the descriptor. For each Memory Read Completion, the Byte Count field must indicate the remaining number of bytes required to complete the Request, including the number of bytes returned with the Completion. If a Memory Read Request is completed using multiple Completions, the Byte Count value for each successive Completion is the value indicated by the preceding Completion minus the number of bytes returned with the preceding Completion. The total number of bytes required to complete a Memory Read Request is calculated as shown in Table 3-7, page 105.
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.

Table 3-6: **Completer Completion Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
45:43	Completion Status	These bits must be set based on the type of Completion being sent. The only valid settings are: • 000: Successful Completion • 001: Unsupported Request (UR) • 100: Completer Abort (CA)
46	Poisoned Completion	This bit can be used to poison the Completion TLP being sent. This bit must be set to 0 for all Completions, except when the user application detects an error in the block of data following the descriptor and wants to communicate this information using the Data Poisoning feature of PCI Express.
63:48	Requester ID	PCI Requester ID associated with the request (copied from the request).
71:64	Tag	PCIe Tag associated with the request (copied from the request).
79:72	Target Function/ Device Number	Function number of the completer Function. The user application must always supply the function number.When ARI is in use, all 8 bits of this field must be set to the target Function number. Otherwise, bits [74:72] must be set to the target Function number. The user application must copy this value from the Target Function field of the descriptor of the corresponding request. Otherwise, bits [74:72] must be set to the target Function number. When ARI is not in use, and the integrated block is configured as a Root Complex, the user application must supply the 5-bit Device Number of the completer on bits [79:75]. When ARI is not used and the integrated block is configured as an Endpoint, the user application can optionally supply a 5-bit Device Number of the completer on bits [79:75]. The user application must set the Completer ID Enable bit in the descriptor if a Device Number is supplied on bits [79:75]. This value is used by the integrated block when sending the Completion TLP, instead of the stored value of the Device Number captured by the integrated block from Configuration Requests.
87:80	Completer Bus Number	Bus number associated with the completer Function. When the integrated block is configured as a Root Complex, the user application must supply the 8-bit Bus Number of the completer in this field. When the integrated block is configured as an Endpoint, the user application can optionally supply a Bus Number in this field. The user application must set the Completer ID Enable bit in the descriptor if a Bus Number is supplied in this field. This value is used by the integrated block when sending the Completion TLP, instead of the stored value of the Bus Number captured by the integrated block from Configuration Requests.
88	Completer ID Enable	The purpose of this field is to enable the user application to supply the bus and device numbers to be used in the Completer ID. This field is applicable only to Endpoint configurations. If this field is 0, the integrated block uses the captured values of the bus and device numbers to form the Completer ID. If this input is 1, the integrated block uses the bus and device numbers supplied in the descriptor to form the Completer ID.
91:89	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request. The user application must copy this value from the TC field of the associated request descriptor.

Table 3-6: **Completer Completion Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
94:92	Attributes	PCIe attributes associated with the request (copied from the request). Bit 92 is the No Snoop bit, bit 93 is the Relaxed Ordering bit, and bit 94 is the ID-Based Ordering bit.
95	Force ECRC	Force ECRC insertion. Setting this bit to 1 forces the integrated block to append a TLP Digest containing ECRC to the Completion TLP, even when ECRC is not enabled for the Function sending the Completion.

Table 3-7: **Calculating Byte Count from Completer Request first_be[3:0], last_be[3:0], Dword Count[10:0]**

first_be[3:0]	last_be[3:0]	Total Byte Count
1xx1	0000	4
01x1	0000	3
1x10	0000	3
0011	0000	2
0110	0000	2
1100	0000	2
0001	0000	1
0010	0000	1
0100	0000	1
1000	0000	1
0000	0000	1
xxx1	1xxx	Dword_count*4
xxx1	01xx	(Dword_count*4)-1
xxx1	001x	(Dword_count*4)-2
xxx1	0001	(Dword_count*4)-3
xx10	1xxx	(Dword_count*4)-1
xx10	01xx	(Dword_count*4)-2
xx10	001x	(Dword_count*4)-3
xx10	0001	(Dword_count*4)-4
x100	1xxx	(Dword_count*4)-2
x100	01xx	(Dword_count*4)-3
x100	001x	(Dword_count*4)-4
x100	0001	(Dword_count*4)-5
1000	1xxx	(Dword_count*4)-3
1000	01xx	(Dword_count*4)-4
1000	001x	(Dword_count*4)-5
1000	0001	(Dword_count*4)-6

Completions with Successful Completion Status

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

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

The CC interface supports two data alignment modes: Dword-aligned and address-aligned. The timing diagrams in Figure 3-22, Figure 3-23, and Figure 3-24 illustrate the Dword-aligned transfer of a Completion from the user application across the CC interface, when the interface width is configured as 64, 128, and 256 bits, respectively. In this case, the first Dword of the payload starts immediately after the descriptor. When the data block is not a multiple of four bytes, or when the start of the payload is not aligned on a Dword address boundary, the user application must add null bytes to align the start of the payload on a Dword boundary and make the payload a multiple of Dwords. For example, when the data block starts at byte address 7 and has a size of 3 bytes, the user application must add three null bytes before the first byte and two null bytes at the end of the block to make it two Dwords long. Also, in the case of non-contiguous reads, not all bytes in the data block returned are valid. In that case, the user application must return the valid bytes in the proper positions, with null bytes added in gaps between valid bytes, when needed. The interface does not have any signals to indicate the valid bytes in the payload. This is not required, as the requester is responsible for keeping track of the byte enables in the request and discarding invalid bytes from the Completion.

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

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

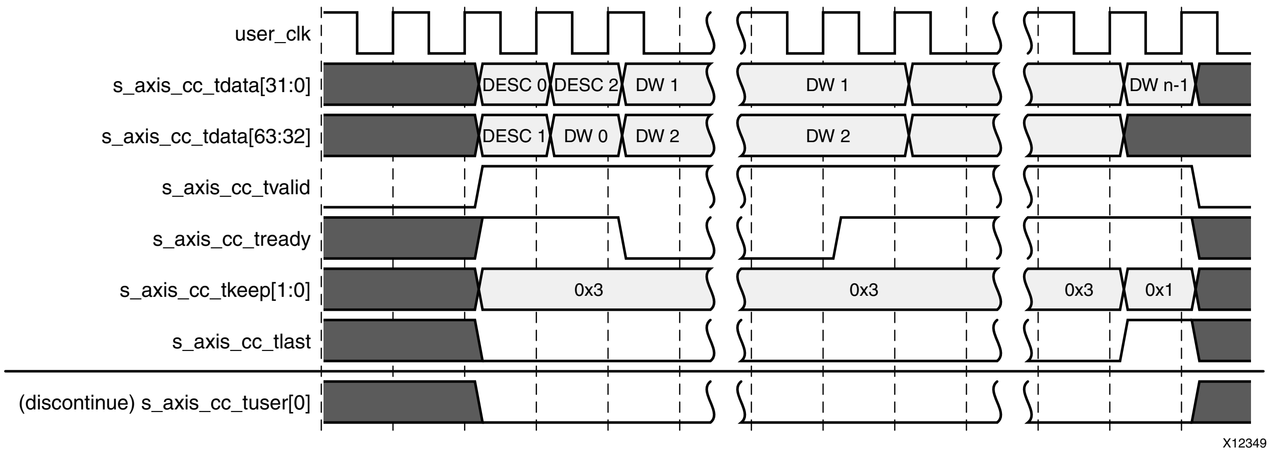

Figure 3-22: **Transfer of a Normal Completion on the Completer Completion Interface (Dword-Aligned Mode, Interface Width = 64 Bits)**

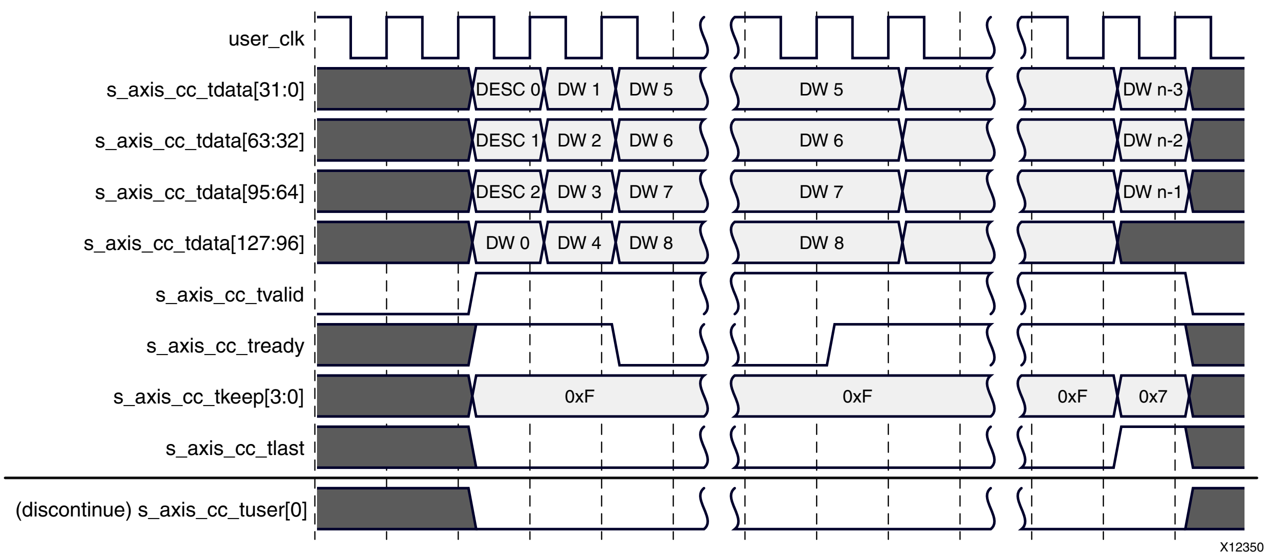

Figure 3-23: **Transfer of a Normal Completion on the Completer Completion Interface (Dword-Aligned Mode, Interface Width = 128 Bits)**

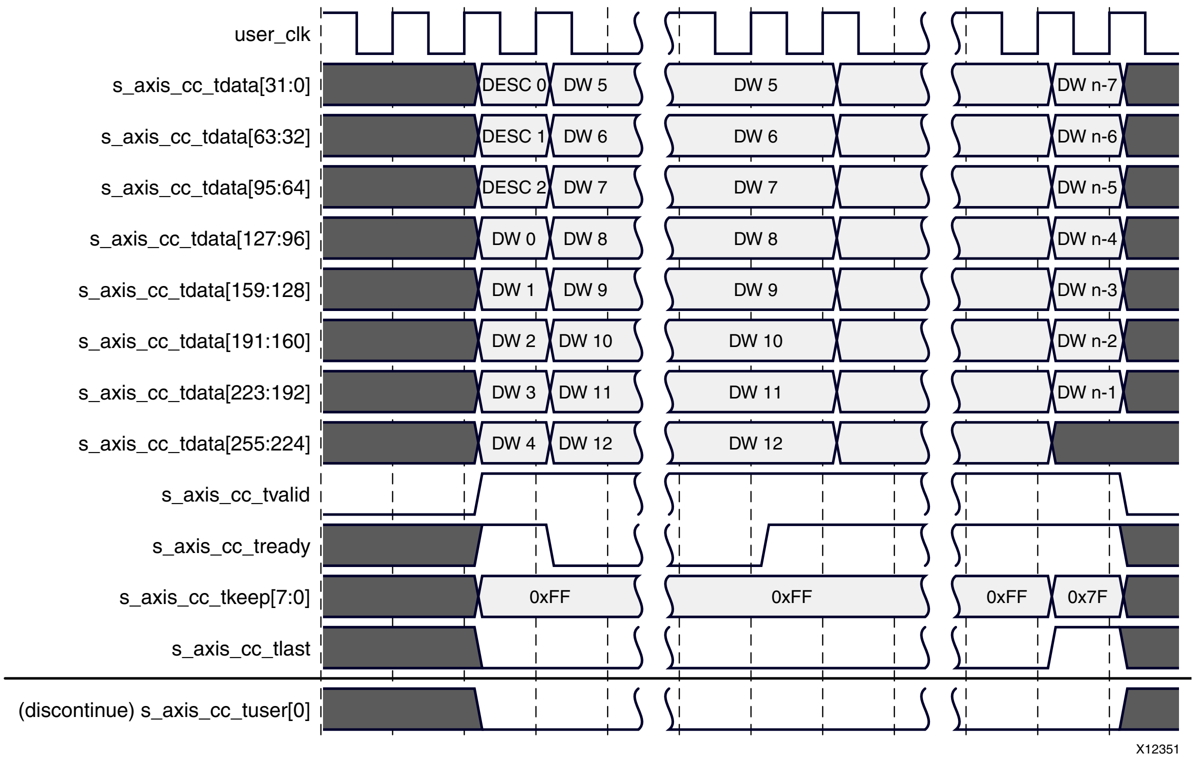

Figure 3-24: **Transfer of a Normal Completion on the Completer Completion Interface (Dword-Aligned Mode, Interface Width = 256 Bits)**

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

The timing diagrams in Figure 3-25, Figure 3-26, and Figure 3-27 illustrate the address-aligned transfer of a memory read Completion across the completer completion interface, when the interface width is configured as 64, 128, and 256 bits, respectively. For the purpose of illustration, the starting Dword address of the data block being transferred (as conveyed in bits [6:2] of the Lower Address field of the descriptor) is assumed to be (m * 8 +1), for some integer m. The size of the data block is assumed to be n Dwords, for some n = k * 32 + 28, k > 0.

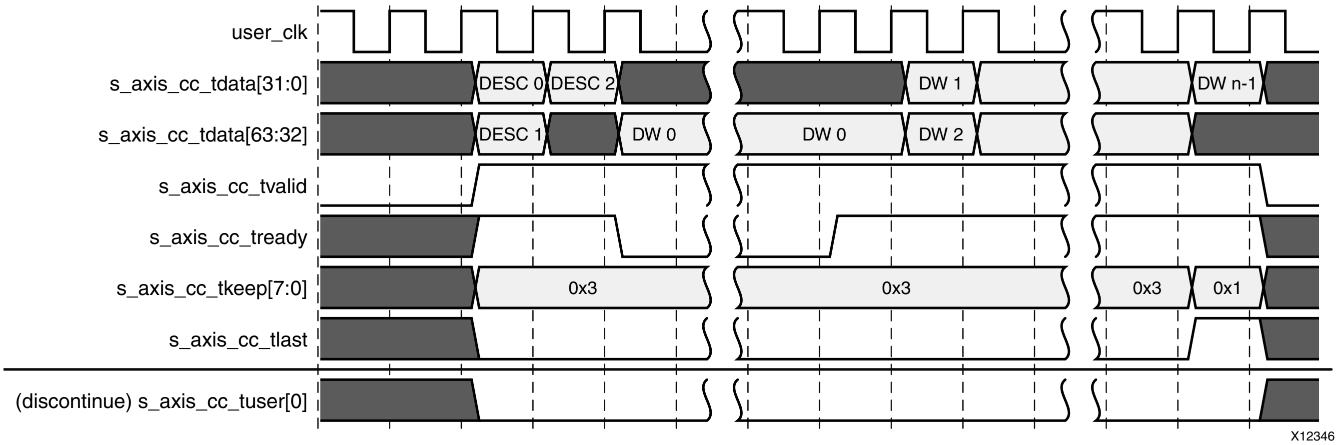

Figure 3-25: **Transfer of a Normal Completion on the Completer Completion Interface (Address-Aligned Mode, Interface Width = 64 Bits)**

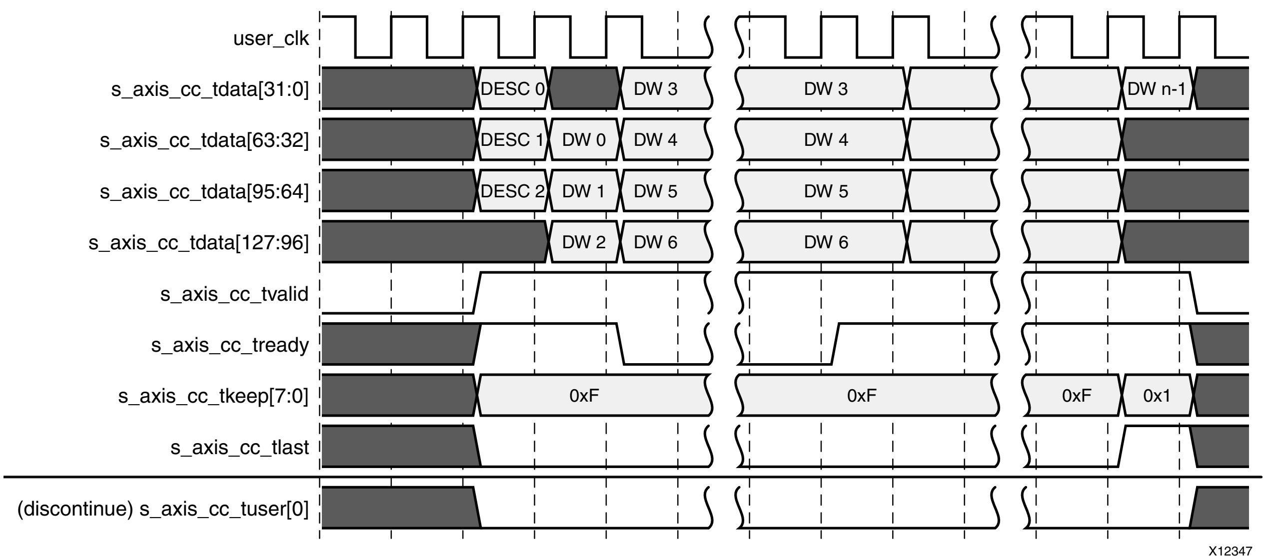

Figure 3-26: **Transfer of a Normal Completion on the Completer Completion Interface (Address-Aligned Mode, Interface Width = 128 Bits)**

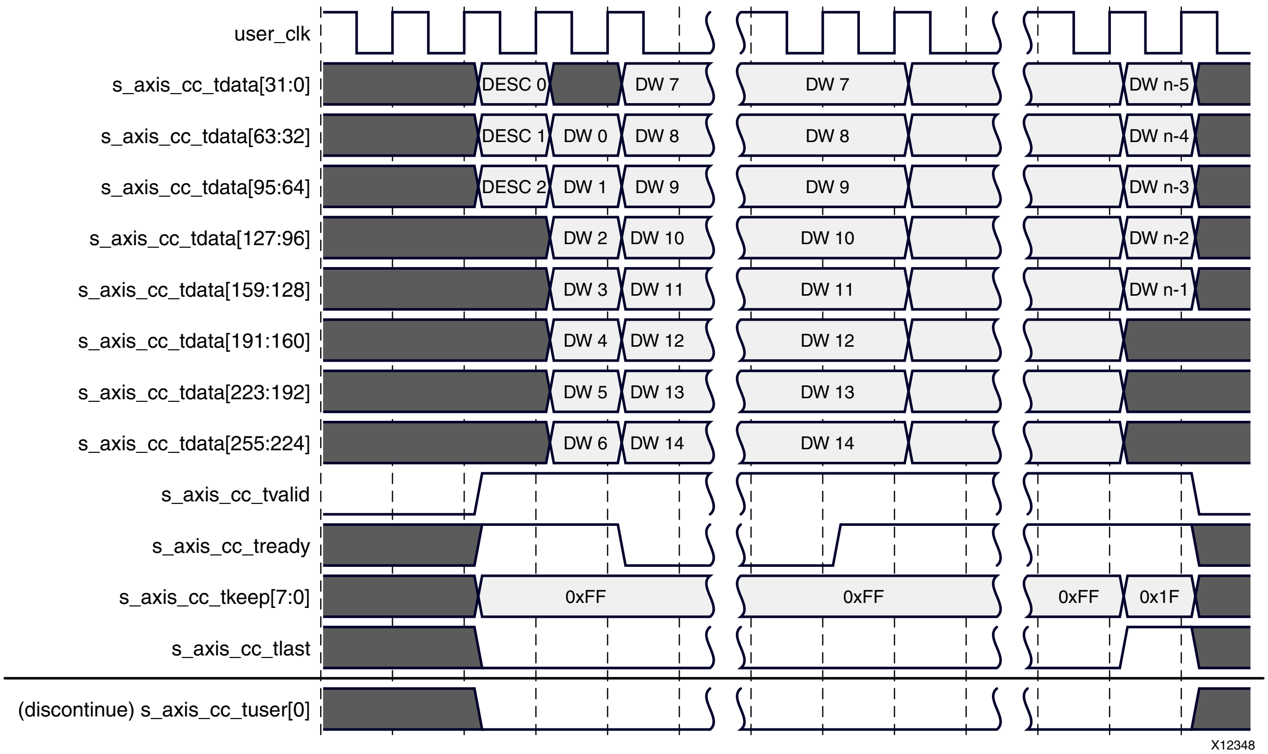

Figure 3-27: **Transfer of a Normal Completion on the Completer Completion Interface (Address-Aligned Mode, Interface Width = 256 Bits)**

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

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

Completions with Error Status (UR and CA)

When responding to a request received on the completer request interface with an Unsupported Request (UR) or Completion Abort (CA) status, the user application must send a three-Dword completion descriptor in the format of Figure 3-21, followed by five additional Dwords containing information on the request that generated the Completion. These five Dwords are necessary for the integrated block to log information about the request in its AER header log registers.

Figure 3-28 shows the sequence of information transferred when sending a Completion with UR or CA status. The information is formatted as an AXI4-Stream packet with a total of 8 Dwords, which are organized as follows:

- The first three Dwords contain the completion descriptor in the format of Figure 3-21.
- The fourth Dword contains the state of the following signals in `m_axis_cq_tuser`, copied from the request:
    - The First Byte Enable bits `first_be[3:0]` in `m_axis_cq_tuser`.
    - The Last Byte Enable bits `last_be[3:0]` in `m_axis_cq_tuser`.
    - Signals carrying information on Transaction Processing Hint: `tph_present`, `tph_type[1:0]`, and `tph_st_tag[7:0]` in `m_axis_cq_tuser`.
- The four Dwords of the request descriptor received from the integrated block with the request.

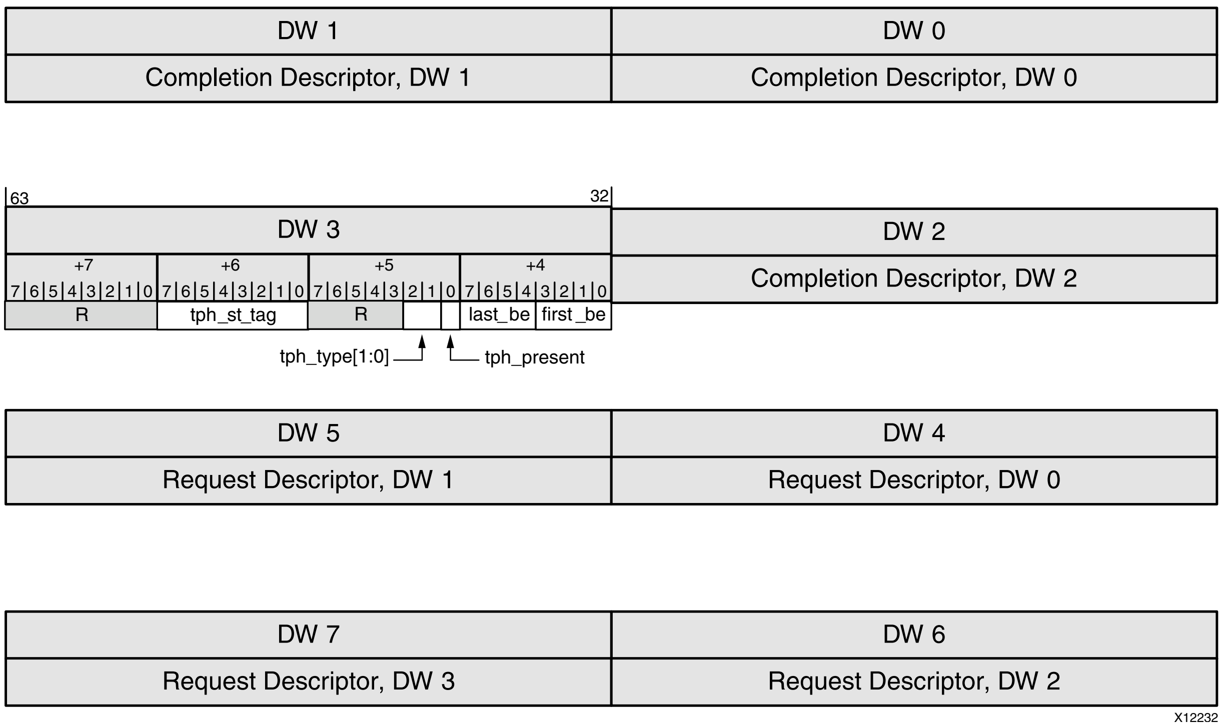

Figure 3-28: **Composition of the AXI4-Stream Packet for UR and CA Completions**

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`.

Receive Message Interface

The core provides a separate receive-message interface which the user application can use to receive indications of messages received from the link. When the receive message interface is enabled, the integrated block signals the arrival of a message from the link by setting the `cfg_msg_received_type[4:0]` output to indicate the type of message (see Table 3-8) and pulsing the `cfg_msg_received` signal for one or more cycles. The duration of assertion of `cfg_msg_received` is determined by the type of message received (see Table 3-9). When `cfg_msg_received` is High, the integrated block transfers any parameters associated with the message on the bus 8 bits at a time on the bus `cfg_msg_received_data`. The parameters transferred on this bus in each cycle of `cfg_msg_received` assertion for various message types are listed in Table 3-9. For Vendor-Defined Messages, the integrated block transfers only the first Dword of any associated payload across this interface. When larger payloads are in use, the completer request interface should be used for the delivery of messages.

Table 3-8: **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 3-9: **Message Parameters on Receive Message Interface**

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

Table 3-9: **Message Parameters on Receive Message Interface *(Cont'd)***

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

Table 3-9: **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]
ATS Page Request	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number
ATS PRG Response	2	Cycle 1: Requester ID, Bus Number Cycle 2: Requester ID, Device/Function Number

Figure 3-29 is a timing diagram showing the example of a Set_Slot_Power_Limit message on the receive message interface. This message has an associated one-Dword payload. For this message, the parameters are transferred over six consecutive cycles. The following information appears on the `cfg_msg_received_data` bus in each cycle:

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

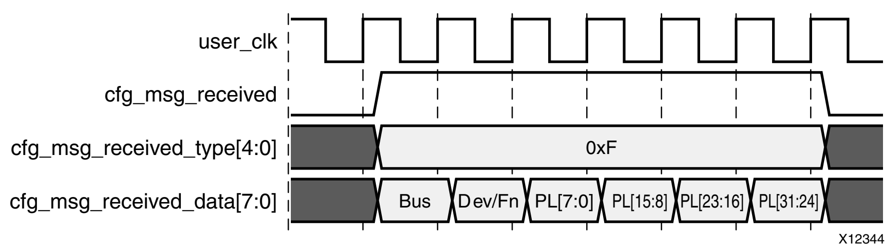

Figure 3-29: **Receive Message Interface**

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 backpressure 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.

Receive Message Interface Design Requirements

When configured as an Endpoint, the user application must implement one of the following:

1. The user application must issue Non-Posted Requests that result in Completions with the RO bit set.
2. The user application must not exceed the configured completion space.

This requirement ensures the RX Completion buffer does not overflow.

Requester Interface

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

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

Requester Request Interface Operation

On the RQ interface, the user application delivers each TLP as an AXI4-Stream packet. The packet starts with a 128-bit descriptor, followed by data in the case of TLPs with a payload. Figure 3-30 shows the signals associated with the requester request interface.

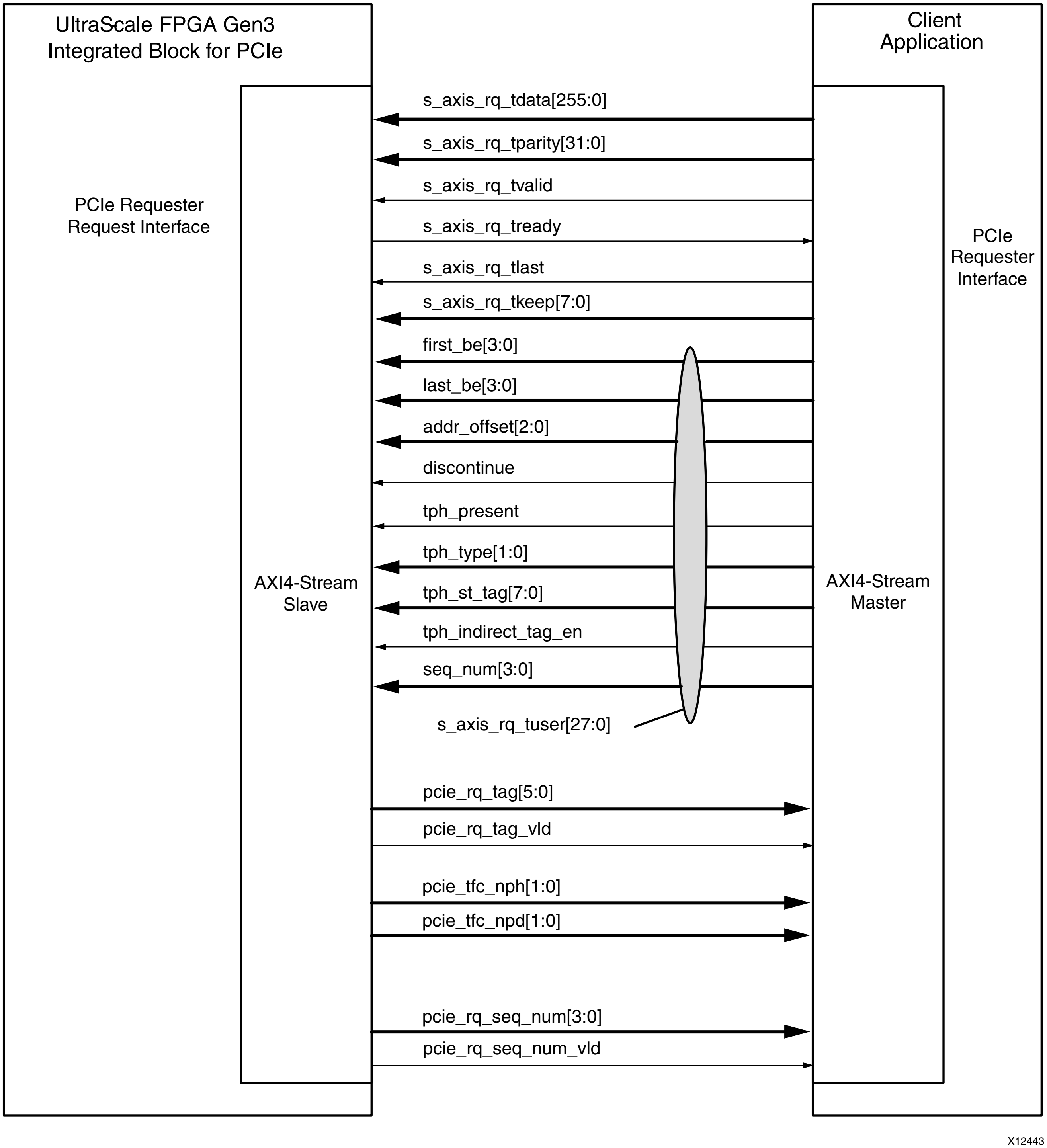

Figure 3-30: **Requester Request Interface**

The RQ interface supports two distinct data alignment modes for transferring payloads. In the Dword-aligned mode, the user logic must provide the first Dword of the payload immediately after the last Dword of the descriptor. It must also set the bits in `first_be[3:0]` to indicate the valid bytes in the first Dword and the bits in `last_be[3:0]` (both part of the bus `s_axis_rq_tuser`) to indicate the valid bytes in the last Dword of the payload. In the address-aligned mode, the user application must start the payload transfer in the beat following the last Dword of the descriptor, and its first Dword can be in any of the possible Dword positions on the datapath. The user application communicates the offset of the first Dword on the datapath using the `addr_offset[2:0]` signals in `s_axis_rq_tuser`. As in the case of the Dword-aligned mode, the user application must also set the bits in `first_be[3:0]` to indicate the valid bytes in the first Dword and the bits in `last_be[3:0]` to indicate the valid bytes in the last Dword of the payload.

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

Requester Request Descriptor Formats

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

The formats of the descriptor for different request types are illustrated in Figure 3-31 through Figure 3-35. The format of Figure 3-31 applies when the request TLP being transferred is a memory read/write request, an I/O read/write request, or an Atomic Operation request. The format in Figure 3-32 is used for Configuration Requests. The format in Figure 3-33 is used for Vendor-Defined Messages (Type 0 or Type 1) only. The format in Figure 3-34 is used for all ATS messages (Invalid Request, Invalid Completion, Page Request, PRG Response). For all other messages, the descriptor takes the format shown in Figure 3-35.

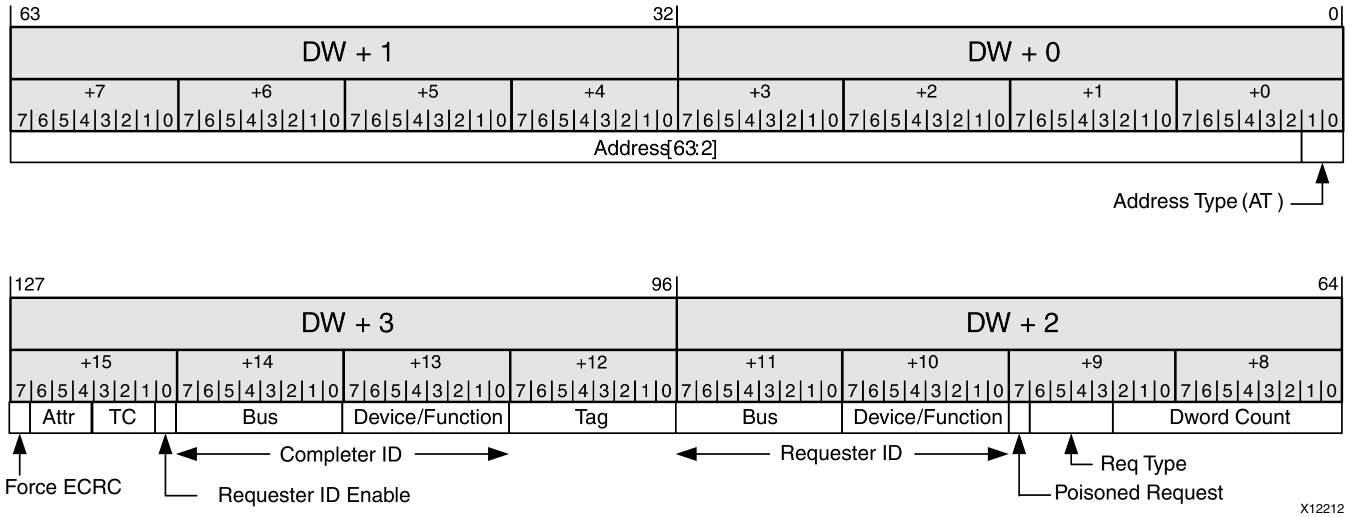

Figure 3-31: **Requester Request Descriptor Format for Memory, I/O, and Atomic Op Requests**

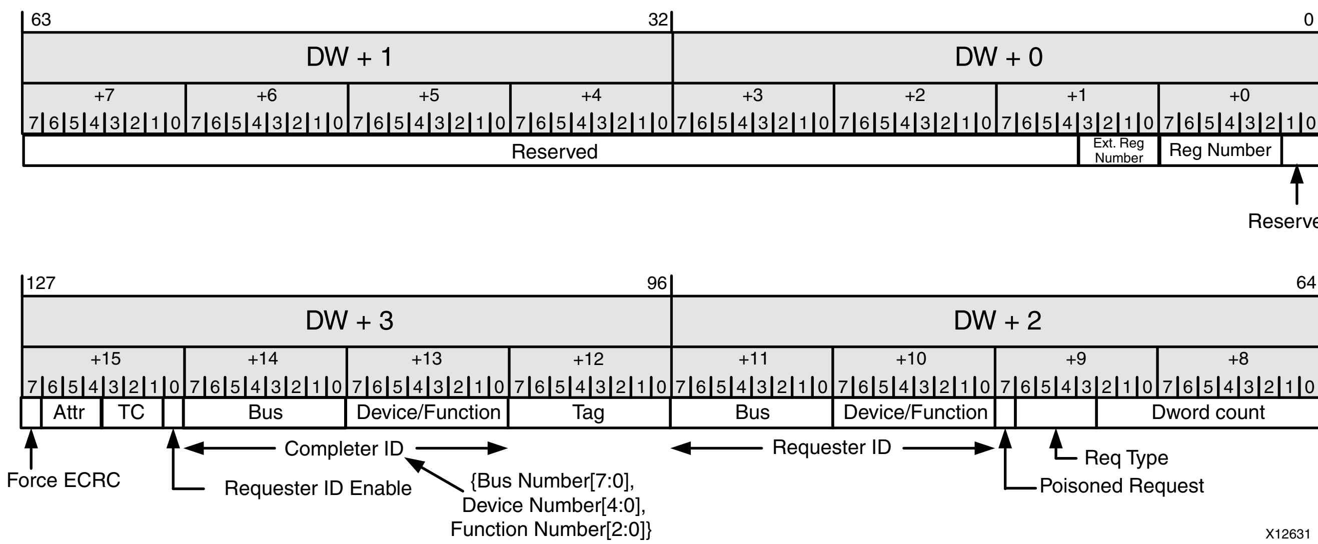

Figure 3-32: **Requester Request Descriptor Format for Configuration Requests**

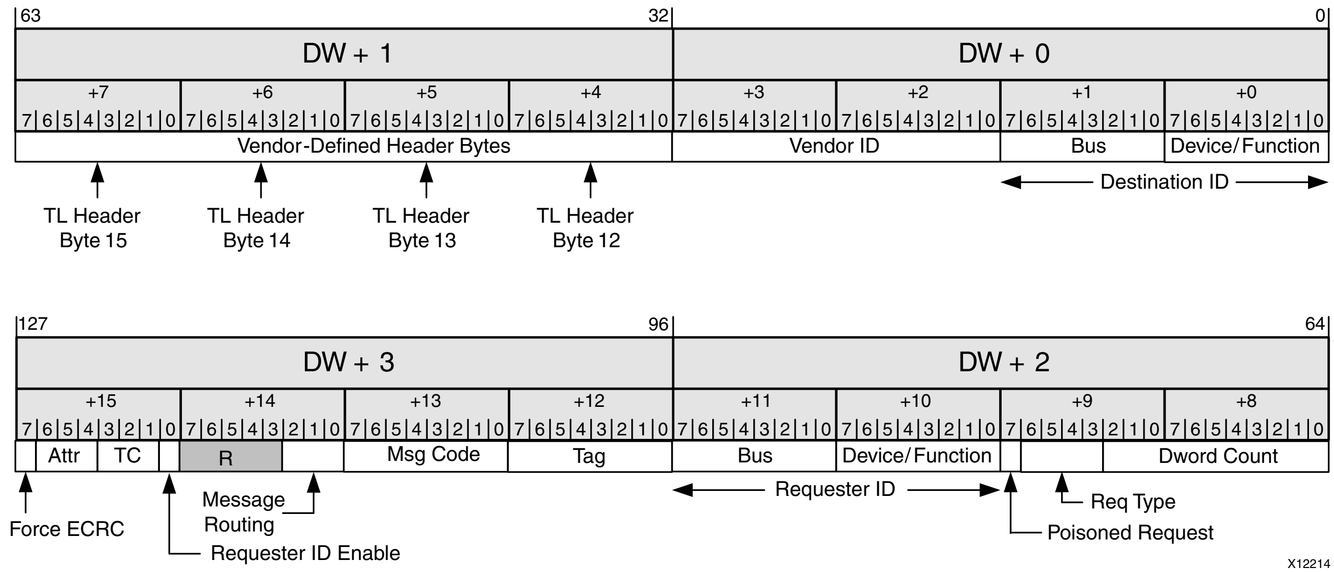

Figure 3-33: **Requester Request Descriptor Format for Vendor-Defined Messages**

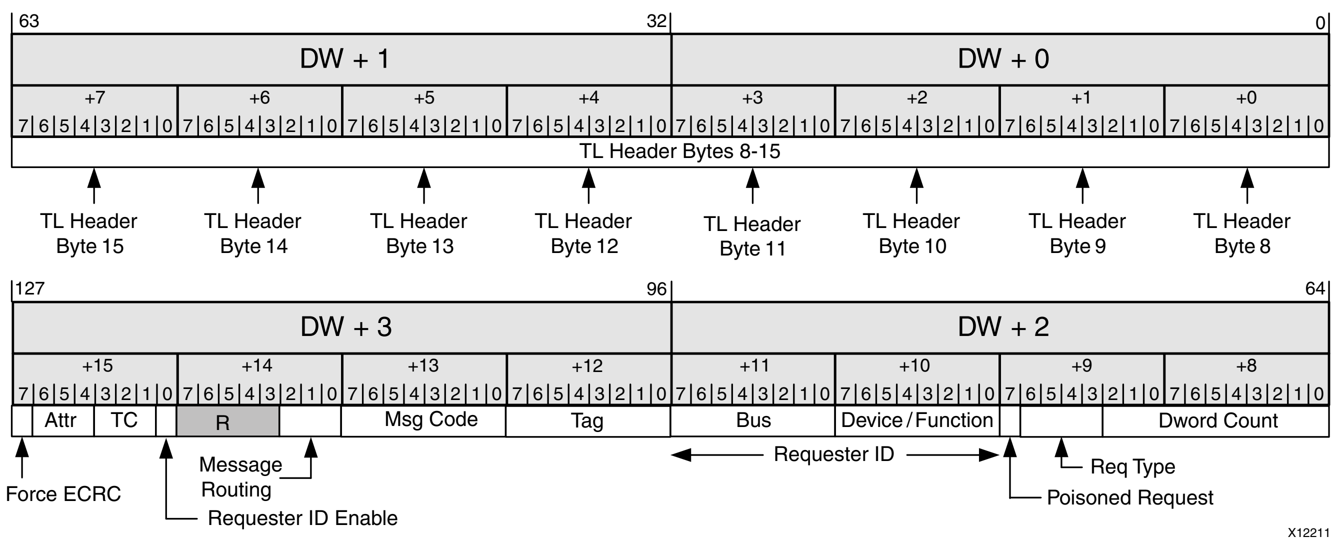

Figure 3-34: **Requester Request Descriptor Format for ATS Messages**

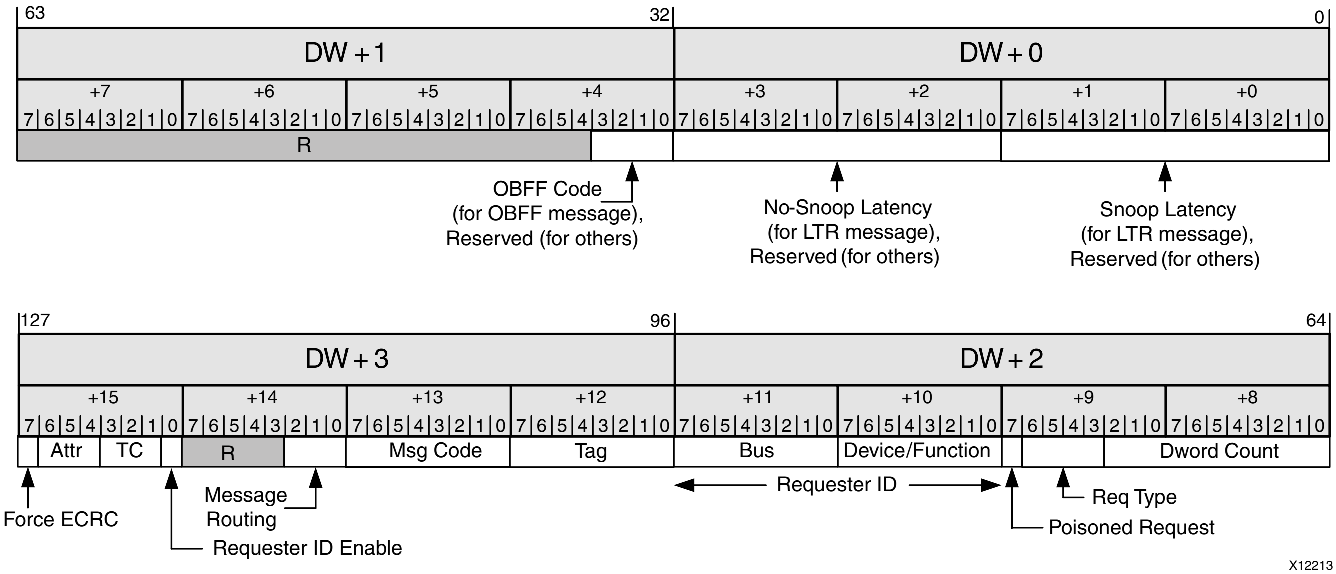

Figure 3-35: **Requester Request Descriptor Format for all other Messages**

Table 3-10 describes the individual fields of the completer request descriptor.

Table 3-10: **Requester Request Descriptor Fields**

Bit Index	Field Name	Description
1:0	Address Type	This field is defined for memory transactions and Atomic Operations only. The integrated block copies this field into the AT of the TL header of the request TLP. • 00: Address in the request is untranslated • 01: Transaction is a Translation Request • 10: Address in the request is a translated address • 11: Reserved
63:2	Address	This field applies to memory, I/O, and Atomic Op requests. This is the address of the first Dword referenced by the request. The user application must also set the First_BE and Last_BE bits in `s_axis_rq_tuser` to indicate the valid bytes in the first and last Dwords, respectively. When the transaction specifies a 32-bit address, bits [63:32] of this field must be set to 0.
74:64	Dword Count	These 11 bits indicate the size of the block (in Dwords) to be read or written (for messages, size of the message payload). The valid range for Memory Write Requests is 0-256 Dwords. Memory Read Requests have a valid range of 1-1024 Dwords. For I/O accesses, the Dword count is always 1. For a zero length memory read/write request, the Dword count must be 1, with the First_BE bits set to all zeros. The integrated block does not check the setting of this field against the actual length of the payload supplied (for requests with payload), nor against the maximum payload size or read request size settings of the integrated block.

Table 3-10: **Requester Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
78:75	Request Type	Identifies the transaction type. The transaction types and their encodings are listed in Table 3-5.
79	Poisoned Request	This bit can be used to poison the request TLP being sent. This feature is supported on all request types except Type 0 and Type 1 Configuration Write Requests. This bit must be set to 0 for all requests, except when the user application detects an error in the block of data following the descriptor and wants to communicate this information using the Data Poisoning feature of PCI Express. This feature is supported on all request types except Type 0 and Type 1 Configuration Write Requests.
87:80	Requester Function/ Device Number	Function number of the Requester Function. When ARI is in use, all 8 bits of this field must be set to the Function number. Otherwise, bits [84:82] must be set to the completer Function number. When ARI is not in use, and the integrated block is configured as a Root Complex, the user application must supply the 5-bit Device Number of the requester on bits [87:83]. When ARI is not use, and the integrated block is configured as an Endpoint, the user application can optionally supply a 5-bit Device Number of the requester on bits [87:83]. The user application must set the Requester ID Enable bit in the descriptor if a Device Number is supplied on bits [87:83]. This value is used by the integrated block when sending the Request TLP, instead of the stored value of the Device Number captured by the integrated block from Configuration Requests.
95:88	Requester Bus Number	Bus number associated with the requester Function. When the integrated block is configured as a Root Complex, the user application must supply the 8-bit Bus Number of the requester in this field. When the integrated block is configured as an Endpoint, the user application can optionally supply a Bus Number in this field. The user application must set the Requester ID Enable bit in the descriptor if a Bus Number is supplied in this field. This value is used by the integrated block when sending the Request TLP, instead of the stored value of the Bus Number captured by the integrated block from Configuration Requests.
103:96	Tag	PCIe Tag associated with the request. For Posted transactions, the integrated block always uses the value from this field as the tag for 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). If this parameter is not set, tag management logic in the integrated block generates the tag to be used, and the value in the tag field of the descriptor is not used.

Table 3-10: **Requester Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
119:104	Completer ID	This field is applicable only to Configuration requests and messages routed by ID. For these requests, this field specifies the PCI Completer ID associated with the request (these 16 bits are divided into an 8-bit bus number, 5-bit device number, and 3-bit function number in the legacy interpretation mode. In the ARI mode, these 16 bits are treated as an 8-bit bus number + 8-bit Function number.).
120	Requester ID Enable	The purpose of this field is to enable the user application to supply the bus and device numbers to be used in the Requester ID. This field is applicable only to Endpoints. If this field is 0, the integrated block uses the captured values of the bus and device numbers to form the Requester ID. If this input is 1, the integrated block uses the bus and device numbers supplied in the descriptor to form the Requester ID.
123:121	Transaction Class (TC)	PCIe Transaction Class (TC) associated with the request.
126:124	Attributes	These bits provide the setting of the Attribute bits associated with the request. Bit 124 is the No Snoop bit and bit 125 is the Relaxed Ordering bit. Bit 126 is the ID-Based Ordering bit, and can be set only for memory requests and messages. The integrated block forces the attribute bits to 0 in the request sent on the link if the corresponding attribute is not enabled in the Function's PCI Express Device Control Register.
127	Force ECRC	Force ECRC insertion. Setting this bit to 1 forces the integrated block to append a TLP Digest containing ECRC to the Request TLP, even when ECRC is not enabled for the Function sending request.
15:0	Snoop Latency	This field is defined for LTR messages only. It provides the value of the 16-bit Snoop Latency field in the TLP header of the message.
31:16	No-Snoop Latency	This field is defined for LTR messages only. It provides the value of the 16-bit No-Snoop Latency field in the TLP header of the message.
35:32	OBFF Code	The OBFF Code field is used to distinguish between various OBFF cases: • 1111b: "CPU Active" – System fully active for all device actions including bus mastering and interrupts • 0001b: "OBFF" – System memory path available for device memory read/write bus master activities • 0000b: "Idle" – System in an idle, low power state All other codes are reserved.

Table 3-10: **Requester Request Descriptor Fields *(Cont'd)***

Bit Index	Field Name	Description
111:104	Message Code	This field is defined for all messages. It contains the 8-bit Message Code to be set in the TL header. Appendix F of the *PCI Express Base Specification, rev. 3.0* [Ref 2] provides a complete list of the supported Message Codes.
114:112	Message Routing	This field is defined for all messages. The integrated block copies these bits into the 3-bit Routing field r[2:0] of the TLP header of the Request TLP.
15:0	Destination ID	This field applies to Vendor-Defined Messages only. When the message is routed by ID (that is, when the Message Routing field is `010` binary), this field must be set to the Destination ID of the message.
63:32	Vendor-Defined Header	This field applies to Vendor-Defined Messages only. It is copied into Dword 3 of the TLP header.
63:0	ATS Header	This field is applicable to ATS messages only. It contains the bytes that the integrated block copies into Dwords 2 and 3 of the TLP header.

Requester Memory Write Operation

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

The user application must also assert the `s_axis_rq_tlast` signal in the last beat of the packet. The integrated block can deassert `s_axis_rq_tready` in any cycle if it is not ready to accept data. The user application must not change the values on the RQ interface during cycles when the integrated block has deasserted `s_axis_rq_tready`. The AXI4-Stream interface signals `m_axis_cq_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 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 integrated block, and must ensure that the payload does not cross a 4 Kbyte boundary. For memory writes of two Dwords or less, the 1s in `first_be` and `last_be` can be non-contiguous. For the special case of a zero-length memory write request, the user application must provide a dummy one-Dword payload with `first_be` and `last_be` both set to all 0s. In all other cases, the 1 bits in `first_be` and `last_be` must be contiguous.

The timing diagrams in Figure 3-36, Figure 3-37, and Figure 3-38 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 $n = k * 32 + 29$, $k > 0$.

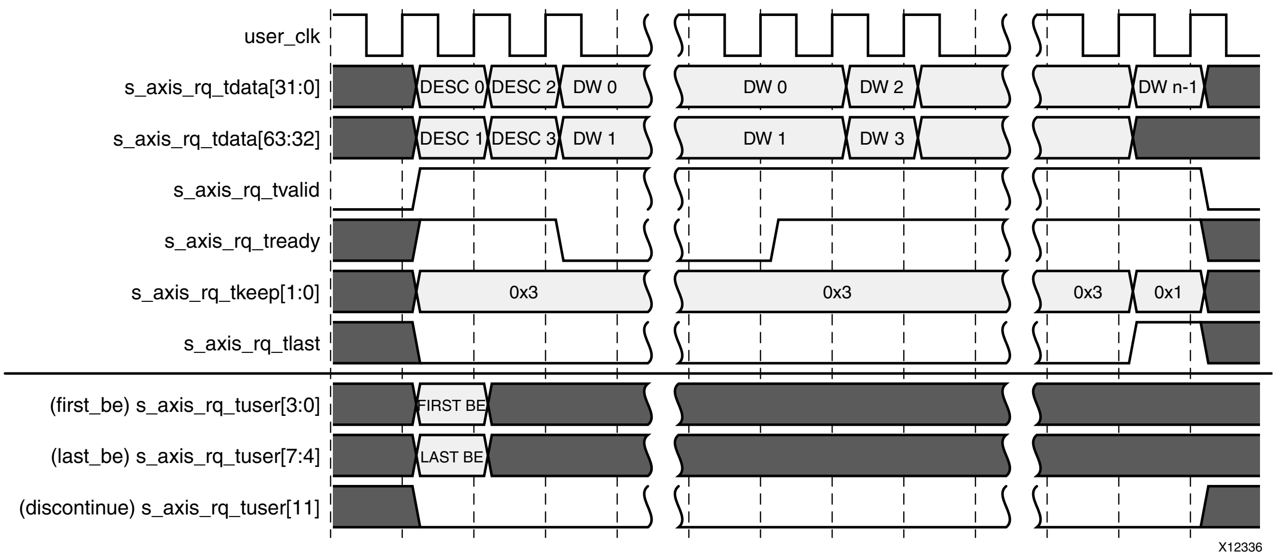

Figure 3-36: **Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, Interface Width = 64 Bits)**

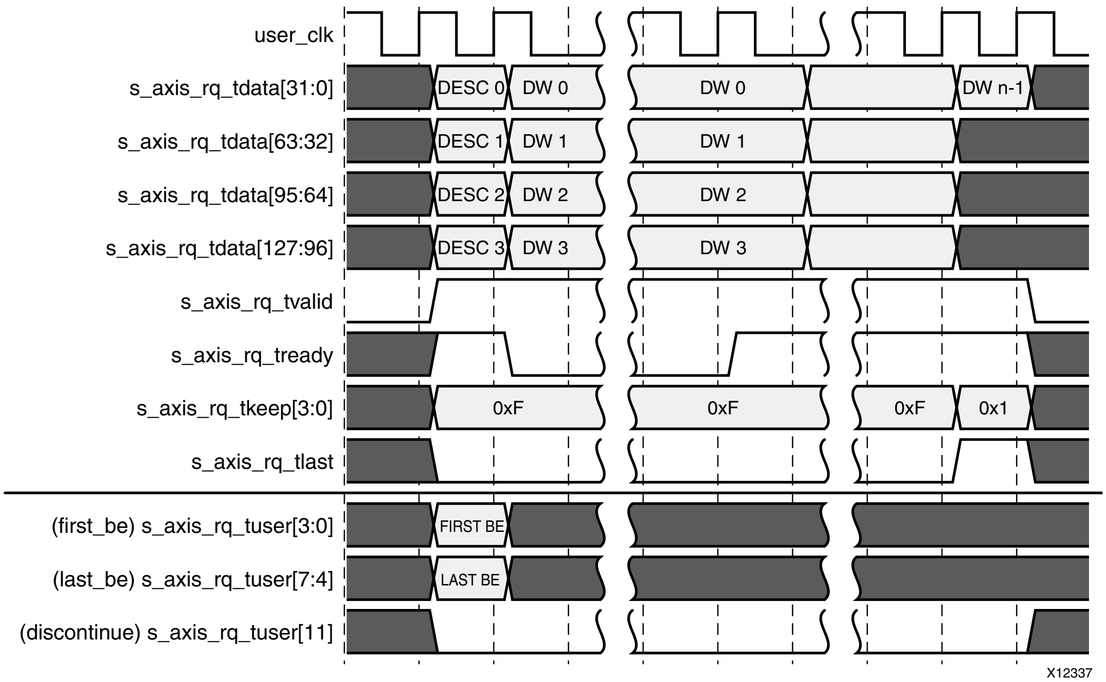

Figure 3-37: **Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, Interface Width = 128 Bits)**

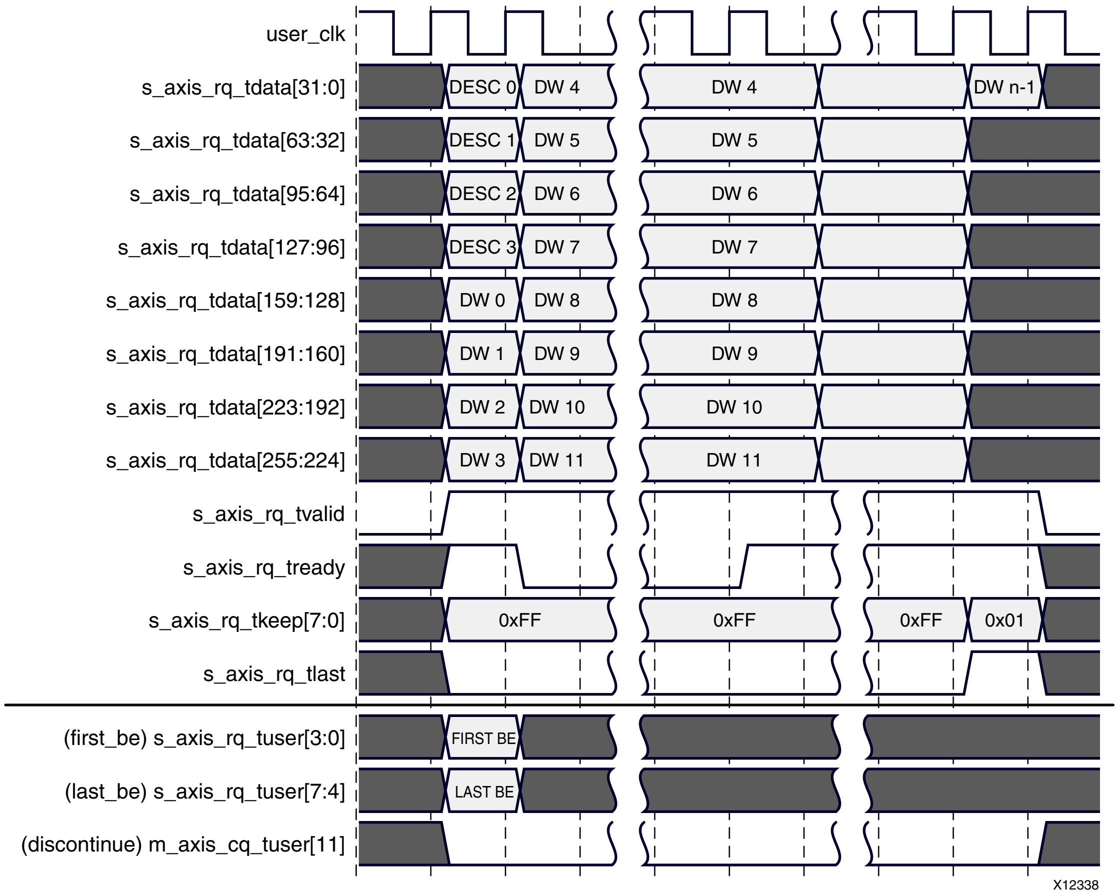

Figure 3-38: **Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, Interface Width = 256 Bits)**

The timing diagrams in Figure 3-39, Figure 3-40, and Figure 3-41 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 * 32 + 1), for some integer m > 0. Its size is assumed to be n Dwords, for some n = k * 32 + 29, k > 0.

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

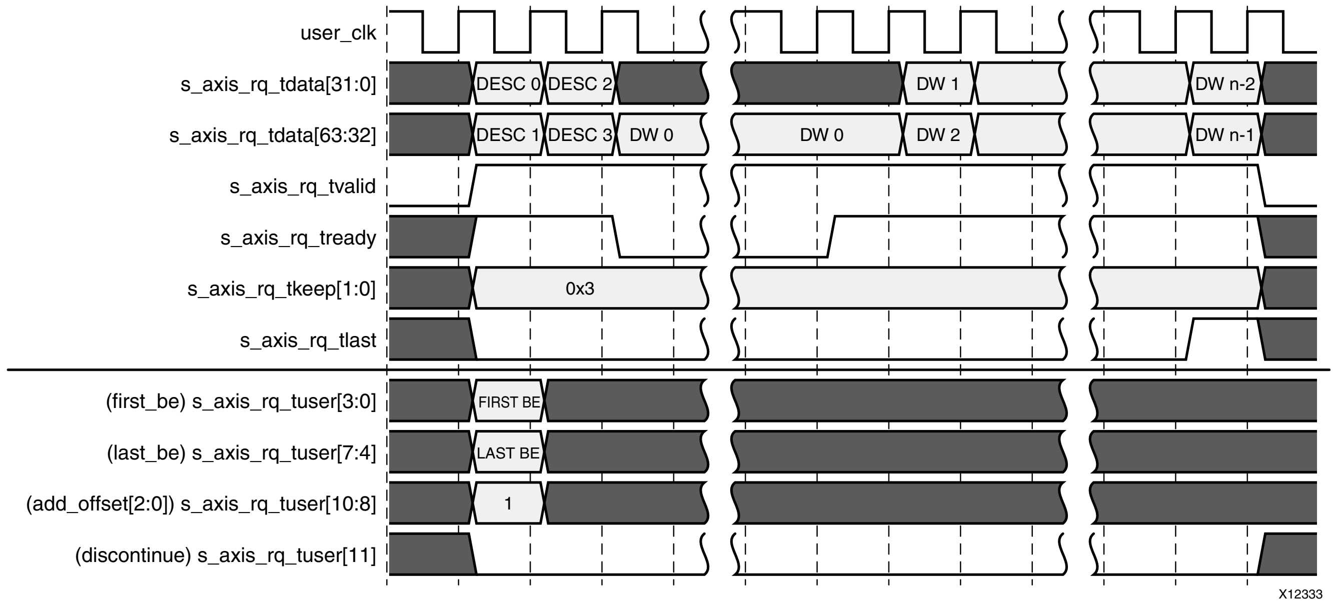

Figure 3-39: **Memory Write Transaction on the Requester Request Interface (Address-Aligned Mode, Interface Width = 64 Bits)**

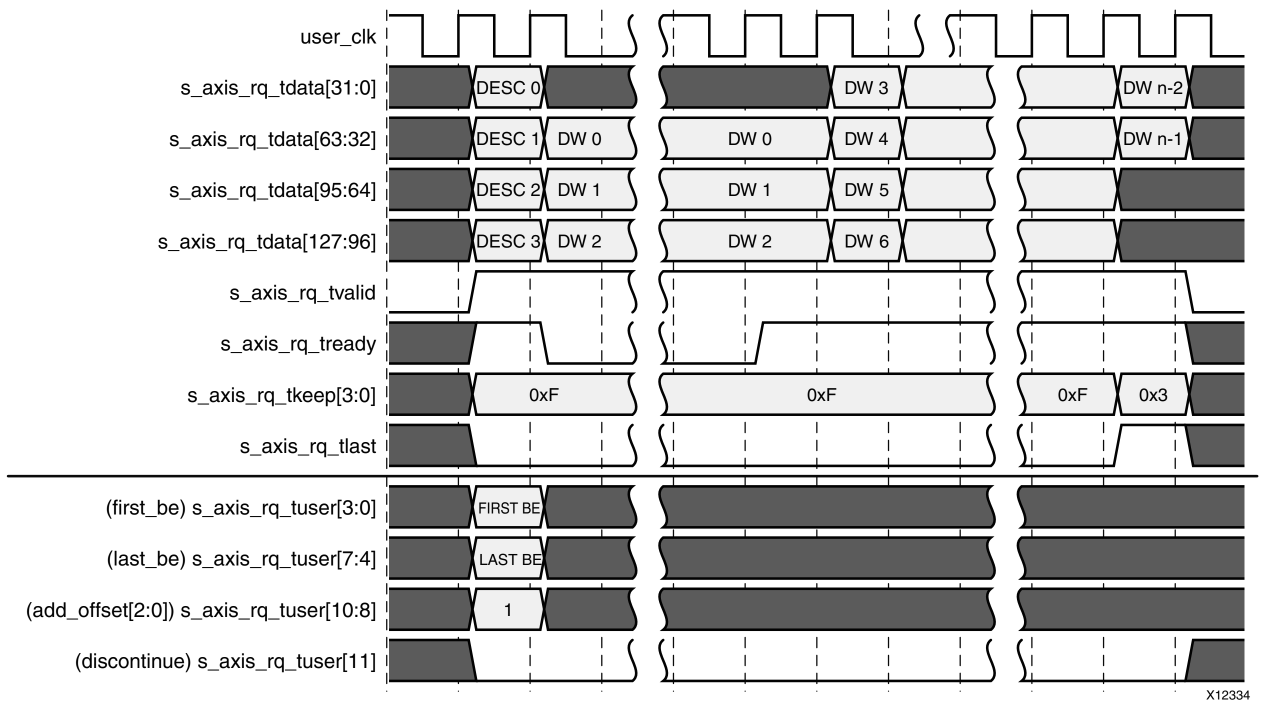

Figure 3-40: **Memory Write Transaction on the Requester Request Interface (Address-Aligned Mode, Interface Width = 128 Bits)**

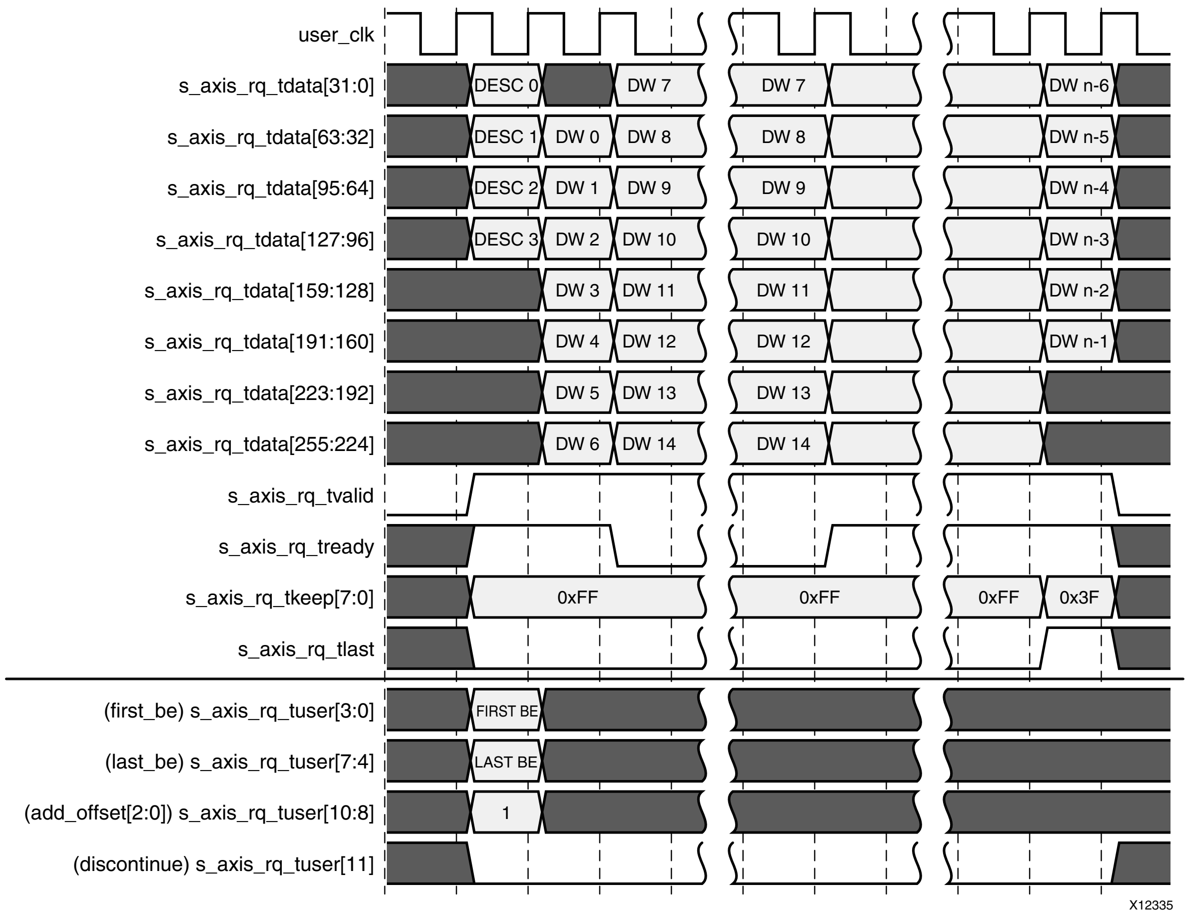

Figure 3-41: **Memory Write Transaction on the Requester Request Interface (Address-Aligned Mode, Interface Width = 256 Bits)**

Non-Posted Transactions with No Payload

Non-Posted transactions with no payload (memory read requests, I/O read requests, Configuration read requests) are transferred across the RQ interface in the same manner as a memory write request, except that the AXI4-Stream packet contains only the 16-byte descriptor. The timing diagrams in Figure 3-42, Figure 3-43, and Figure 3-44 illustrate the transfer of a memory read request across the RQ interface, when the interface width is configured as 64, 128, and 256 bits, respectively. The packet occupies two consecutive beats on the 64-bit interface, while it is transferred in a single beat on the 128- and 256-bit interfaces. The `s_axis_rq_tvalid` signal must remain asserted over the duration of the packet. The integrated block can deassert `s_axis_rq_tready` to prolong the beat. The `s_axis_rq_tlast` signal must be set in the last beat of the packet, and the bits in `s_axis_rq_tkeep[7:0]` must be set in all Dword positions where a descriptor is present.

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

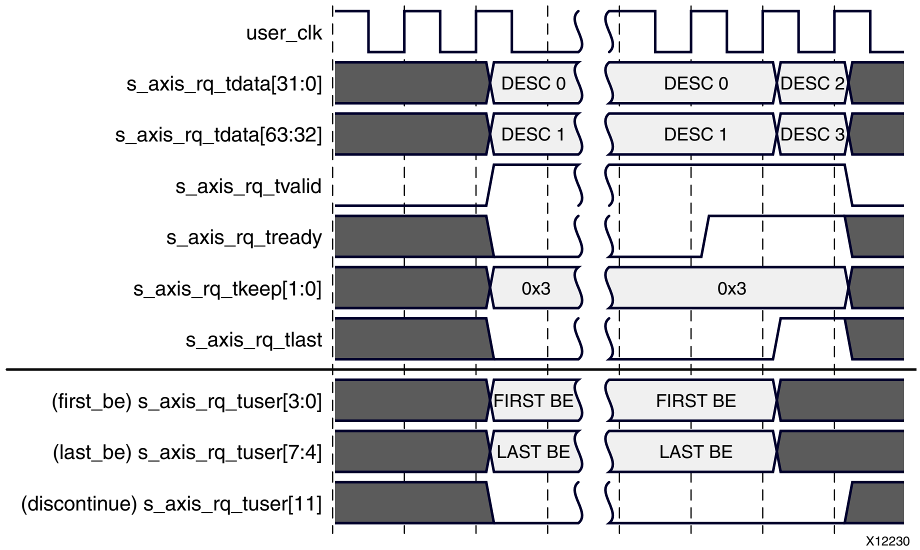

Figure 3-42: **Memory Read Transaction on the Requester Request Interface (Interface Width = 64 Bits)**

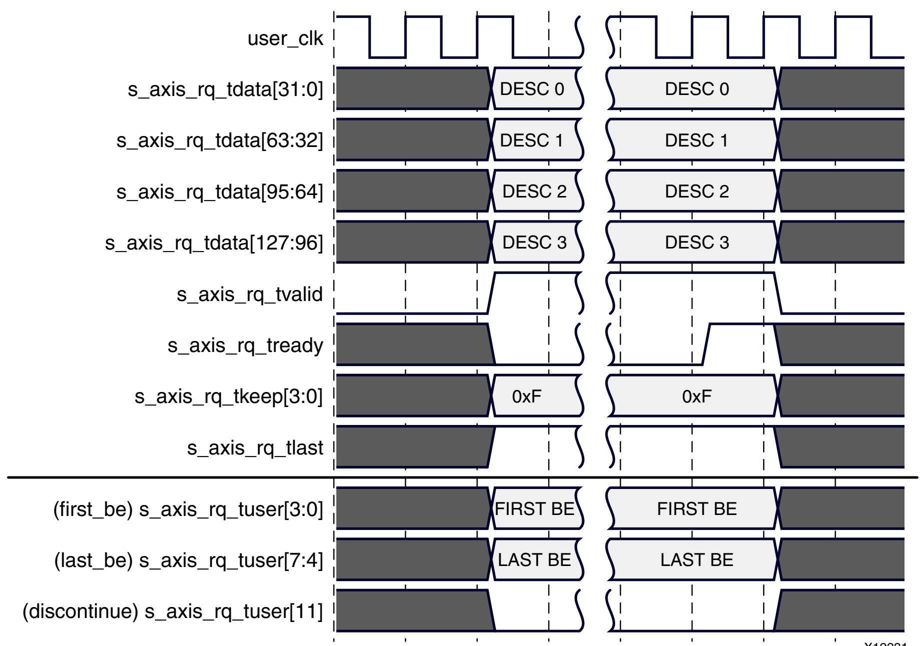

Figure 3-43: **Memory Read Transaction on the Requester Request Interface (Interface Width = 128 Bits)**

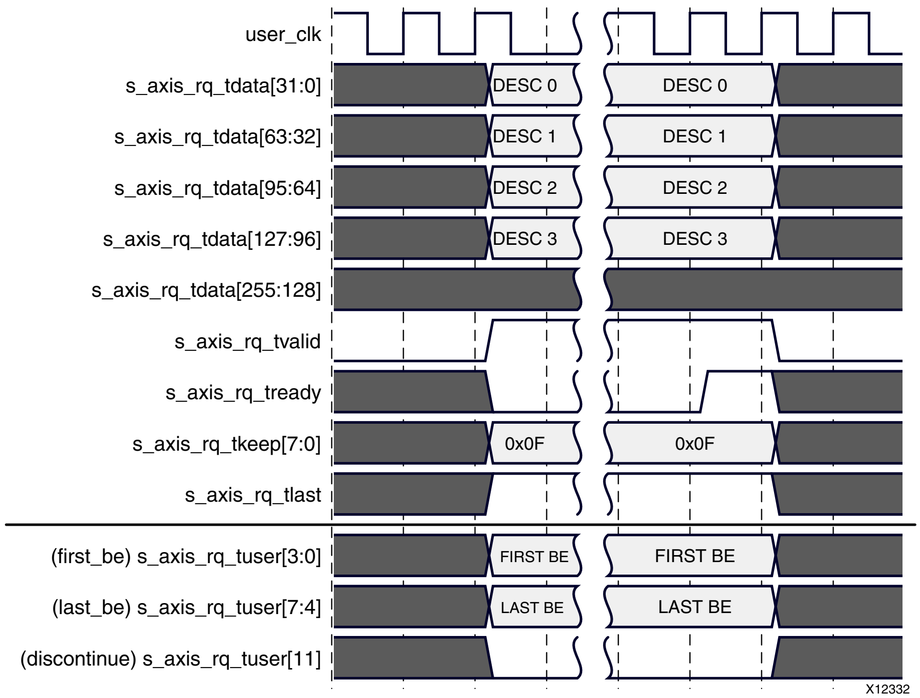

Figure 3-44: **Memory Read Transaction on the Requester Request Interface (Interface Width = 256 Bits)**

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

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

Tag Management for Non-Posted Transactions

The requester side of the integrated block maintains the state of all pending Non-Posted transactions (memory reads, I/O reads and writes, configuration reads and writes, Atomic Operations) initiated by the user application, so that the completions returned by the targets can be matched against the corresponding requests. The state of each outstanding transaction is held in a Split Completion Table in the requester side of the interface, which has a capacity of 64 Non-Posted transactions. The returning Completions are matched with the pending requests using a 6-bit tag. There are two options for management of these tags.

- Internal Tag Management: This mode of operation is selected by setting the AXISTEN_IF_ENABLE_CLIENT_TAG parameter to FALSE, which is the default setting for the core. In this mode, logic within the integrated block is responsible for allocating the tag for each Non-Posted request initiated from the requester side. The integrated block maintains a list of free tags and assigns one of them to each request when the user application initiates a Non-Posted transaction, and communicates the assigned tag value to the user application through the output `pcie_rq_tag[5:0]`. The value on this bus is valid when the integrated block asserts `pcie_rq_tag_vld`. The user logic must copy this tag so that any Completions delivered by the integrated block in response to the request can be matched to the request.

In this mode, logic within the integrated block checks for the Split Completion Table full condition, and backpressures a Non-Posted request from the user application (using `s_axis_rq_tready`) if the total number of Non-Posted requests currently outstanding has reached its limit (64).

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

When internal tag management is in use, the integrated block asserts `pcie_rq_tag_vld` for one cycle for each Non-Posted request, after it has placed its allocated tag on `pcie_rq_tag[5:0]`. There can be a delay of several cycles between the transfer of the request on the RQ interface and the assertion of `pcie_rq_tag_vld` by the integrated block to provide the allocated tag for the request. The user application can, meanwhile, continue to send new requests. The tags for requests are communicated on the `pcie_rq_tag` bus in FIFO order, so it is easy to associate the tag value with the request it transferred. A tag is reused when the end-of-frame (EOF) of the last completion of a split completion is accepted by the user application.

Avoiding Head-of-Line Blocking for Posted Requests

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

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

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

The user logic can optionally check these outputs before transmitting Non-Posted requests. Because of internal pipeline delays, the information on these outputs is delayed by two user clock cycles from the cycle in which the last byte of the descriptor is transferred on the RQ interface. Thus, the user logic must adjust these values, taking into account any Non-Posted requests transmitted in the two previous clock cycles. Figure 3-45 illustrates the operation of these signals for the 256-bit interface. In this example, the integrated block initially had three Non-Posted Header Credits and two Non-Posted Data Credits, and had three free tags available for allocation. Request 1 from the user application had a one-Dword payload, and therefore consumed one header and data credit each, and also one tag. Request 2 in the next clock cycle consumed one header credit, but no data credit. When the user application presents Request 3 in the following clock cycle, it must adjust the available credit and available tag count by taking into account requests 1 and 2. If Request 3 consumes one header credit and one data credit, both available credits are 0 two cycles later, as also the number of available tags.

Figure 3-46 and Figure 3-47 illustrate the timing of the credit and tag available signals for the same example, for interface width of 128 bits and 64 bits, respectively.

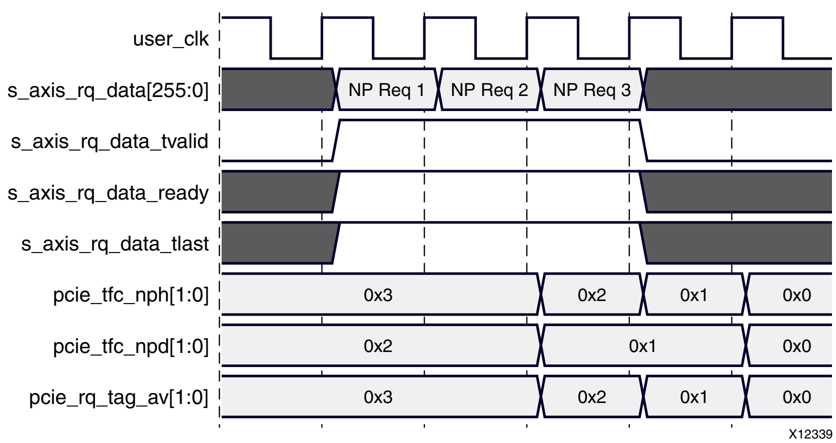

Figure 3-45: **Credit and Tag Availability Signals on the Requester Request Interface (Interface Width = 256 Bits)**

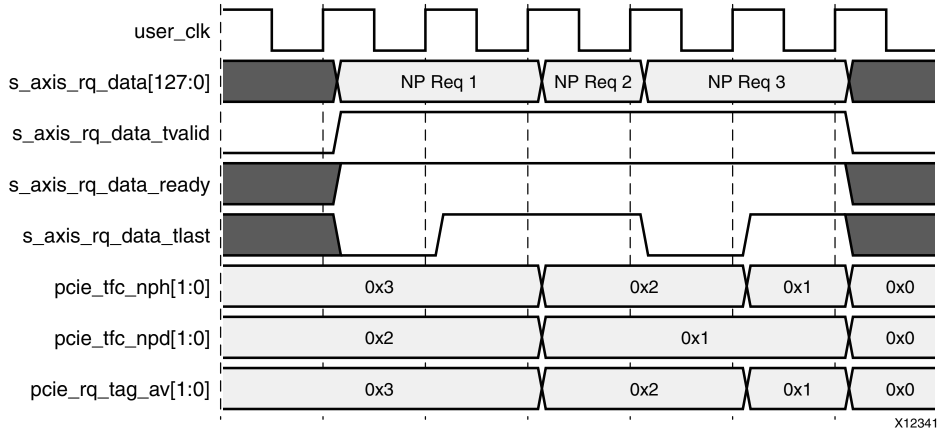

Figure 3-46: **Credit and Tag Availability Signals on the Requester Request Interface (Interface Width = 128 Bits)**

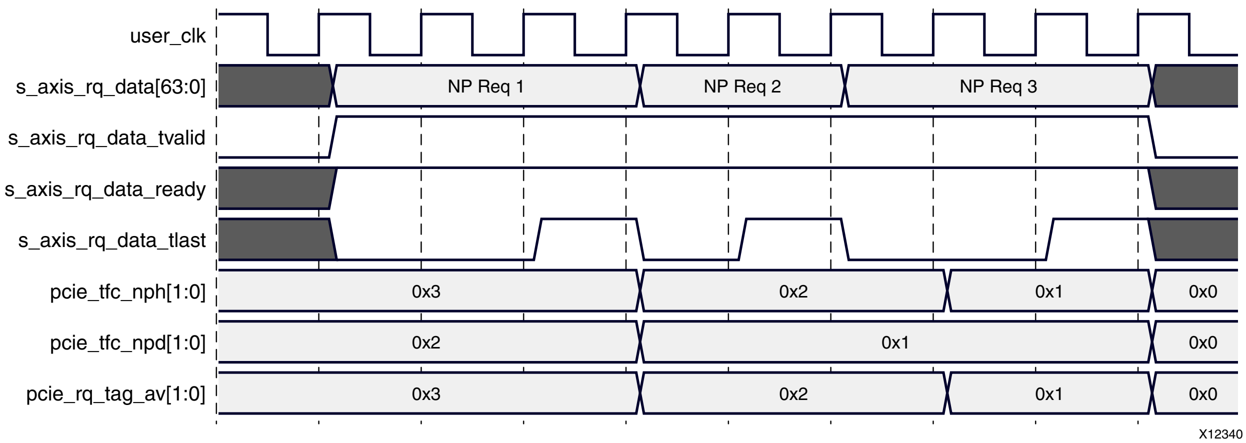

Figure 3-47: **Credit and Tag Availability Signals on the Requester Request Interface (Interface Width = 64 Bits)**

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 user logic are presented on the integrated block Request Completion (RC) interface. See Figure 3-48 for an illustration of signals associated with the requester completion interface. When straddle is not enabled, the integrated block delivers each TLP on this interface as an AXI4-Stream packet. The packet starts with a 96-bit descriptor, followed by data in the case of Completions with a payload.

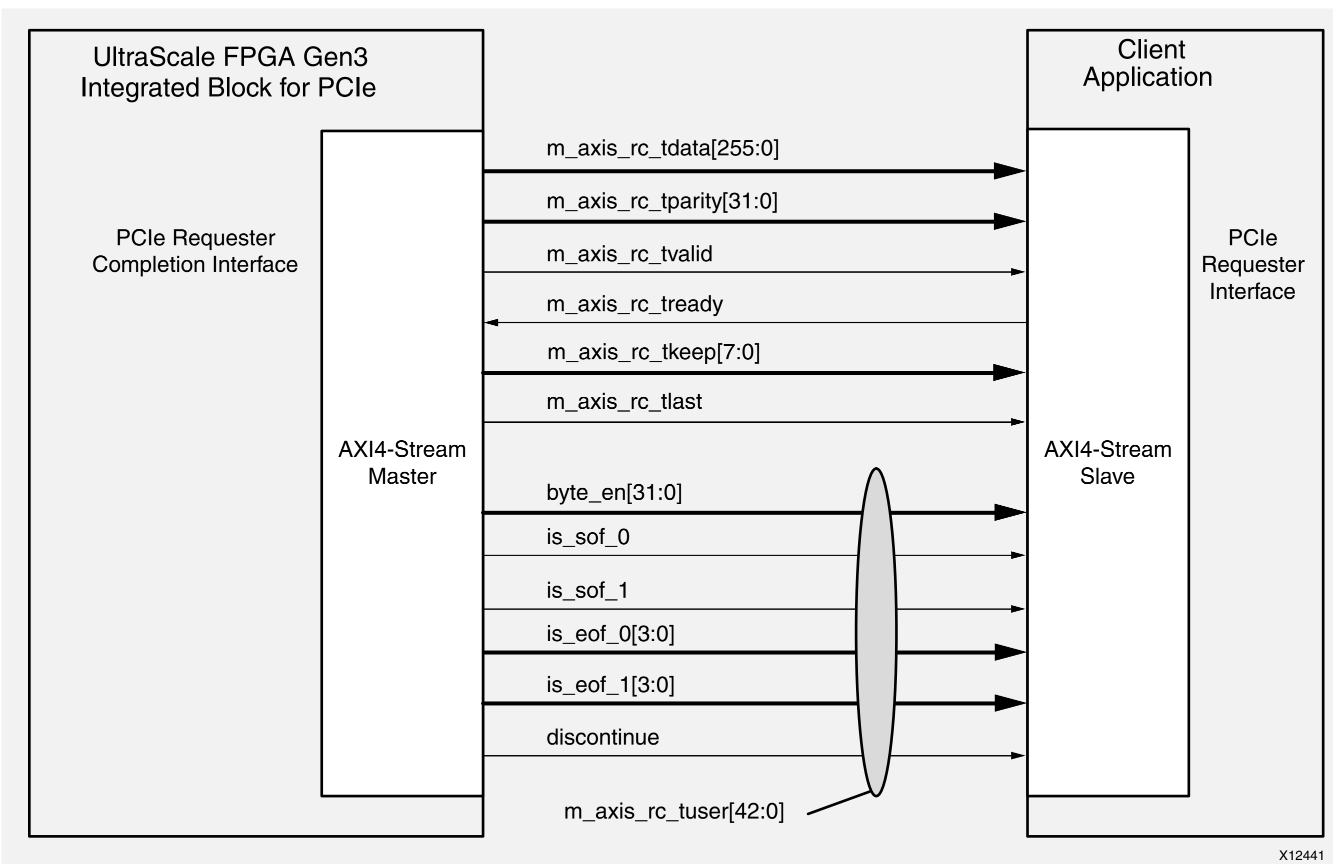

Figure 3-48: **Requester Completion Interface**

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 address offset provided by the user application when it sent the request to the integrated block (that is, the setting of the `addr_offset[2:0]` input of the RQ interface). Thus, the address-aligned mode can be used on the RC interface only if the RQ interface is also configured to use the address-aligned mode.

Requester Completion Descriptor Format

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

The format of the Requester Completion descriptor is illustrated in Figure 3-49. The individual fields of the RC descriptor are described in Table 3-11.

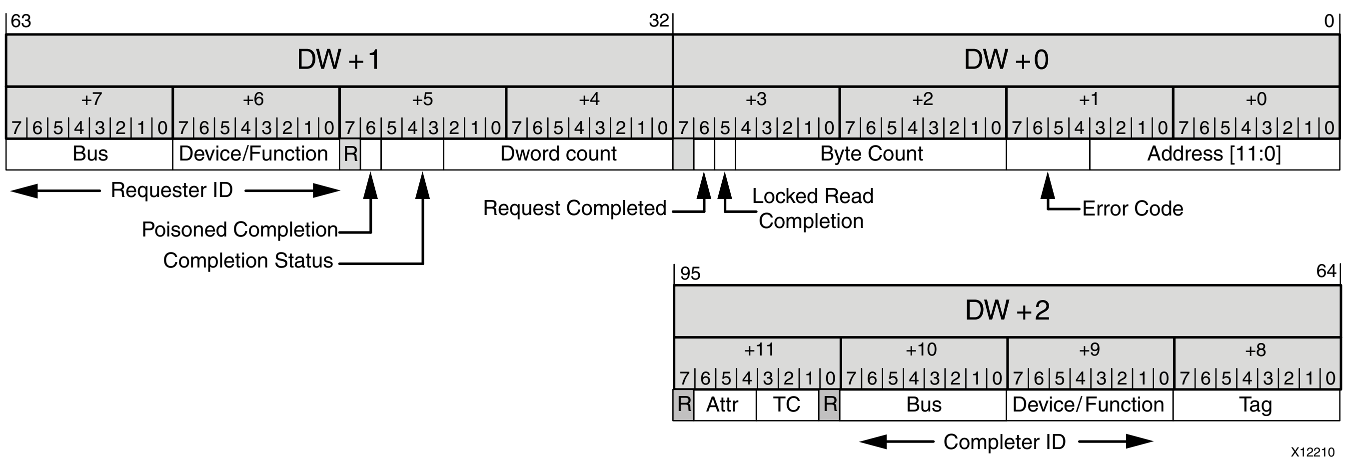

Figure 3-49: **Requester Completion Descriptor Format**

Table 3-11: **Requester Completion Descriptor Fields**

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

Table 3-11: **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 – 4096 bytes. If a Memory Read Request is completed using a single Completion, the Byte Count value indicates Payload size in bytes. This field must be set to 4 for I/O read Completions and I/O write Completions. The byte count must be set to 1 while sending a Completion for a zero-length memory read, and a dummy payload of 1 Dword must follow the descriptor. For each Memory Read Completion, the Byte Count field must indicate the remaining number of bytes required to complete the Request, including the number of bytes returned with the Completion. If a Memory Read Request is completed using multiple Completions, the Byte Count value for each successive Completion is the value indicated by the preceding Completion minus the number of bytes returned with the preceding Completion.
29	Locked Read Completion	This bit is set to 1 when the Completion is in response to a Locked Read request. It is set to 0 for all other Completions.

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

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

Transfer of Completions with no Data

The timing diagrams in Figure 3-50, Figure 3-51, and Figure 3-52 illustrate the transfer of a Completion TLP received from the link with no associated payload across the RC interface, when the interface width is configured as 64, 128, and 256 bits, respectively. The timing diagrams in this section assume that the Completions are not straddled on the 256-bit interface. The straddle feature is described in Straddle Option for 256-Bit Interface, page 147.

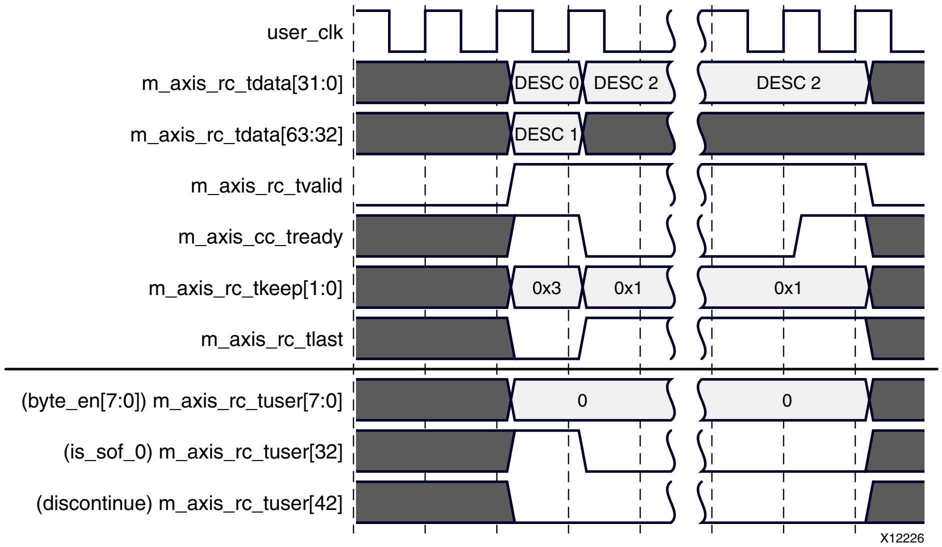

Figure 3-50: **Transfer of a Completion with no Data on the Requester Completion Interface (Interface Width = 64 Bits)**

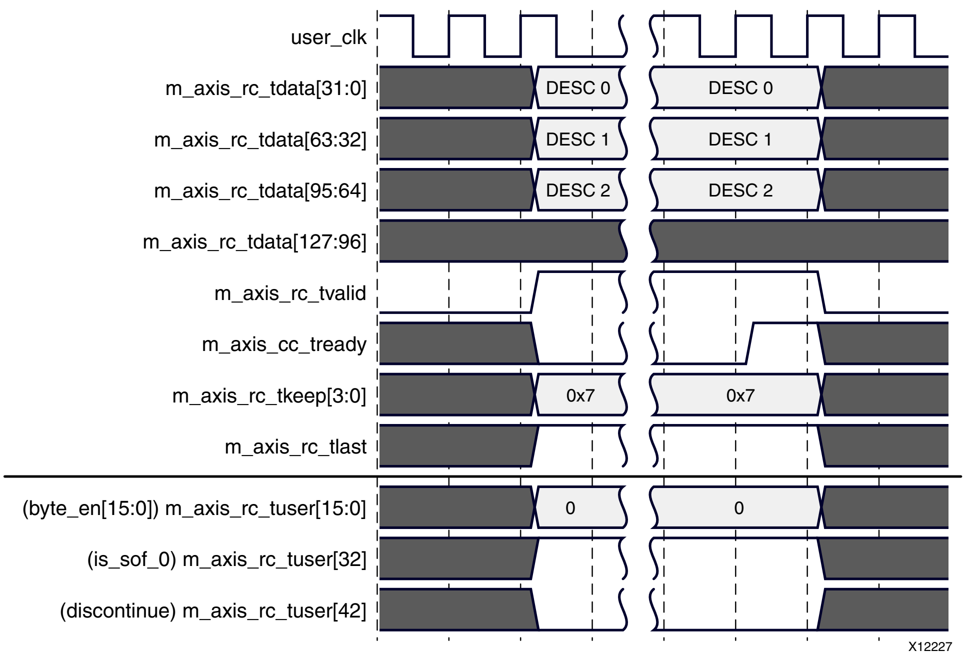

Figure 3-51: **Transfer of a Completion with no Data on the Requester Completion Interface (Interface Width = 128 Bits)**

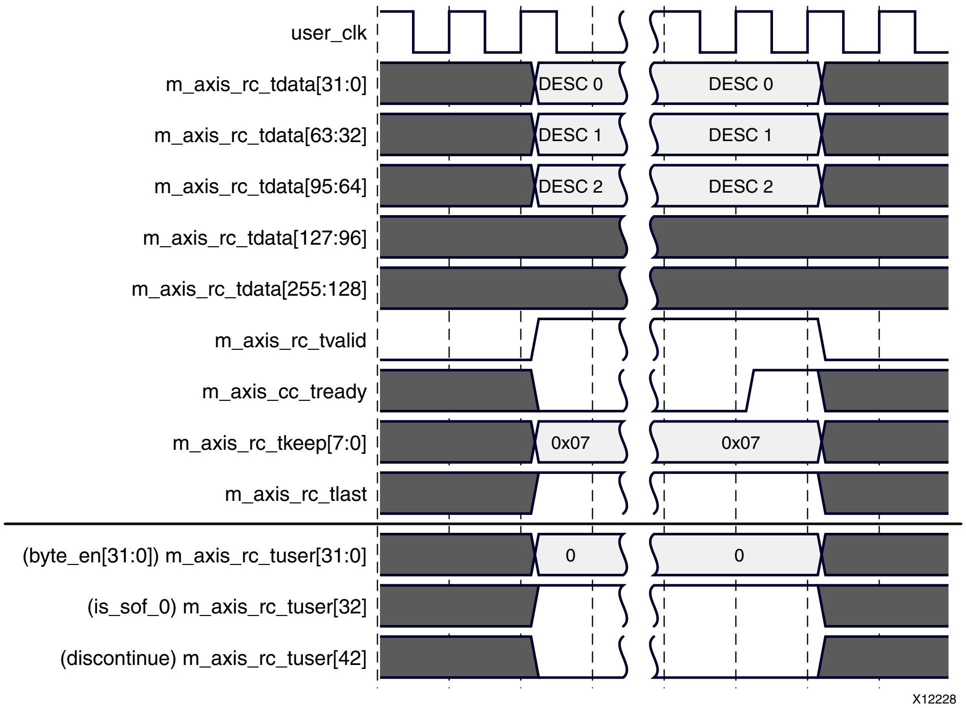

Figure 3-52: **Transfer of a Completion with no Data on the Requester Completion Interface (Interface Width = 256 Bits)**

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_cq_tlast` signal is always asserted in the last beat of the packet.

The `m_axi_cq_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_cq_tuser` are relevant to the transfer of Completions with no data, when the straddle option is not in use.

Transfer of Completions with Data

The timing diagrams in Figure 3-53, Figure 3-54, and Figure 3-55 illustrate the Dword-aligned transfer of a Completion TLP received from the link with an associated payload across the RC interface, when the interface width is configured as 64, 128, and 256 bits, respectively. For illustration purposes, the size of the data block being written into user application memory is assumed to be *n* Dwords, for some $n = k * 32 + 28$, $k > 0$. The timing diagrams in this section assume that the Completions are not straddled on the 256-bit interface. The straddle feature is described in Straddle Option for 256-Bit Interface, page 147.

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_cq_tuser` bus to determine the valid bytes in the payload, in the cases of contiguous as well as non-contiguous payloads.

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

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

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

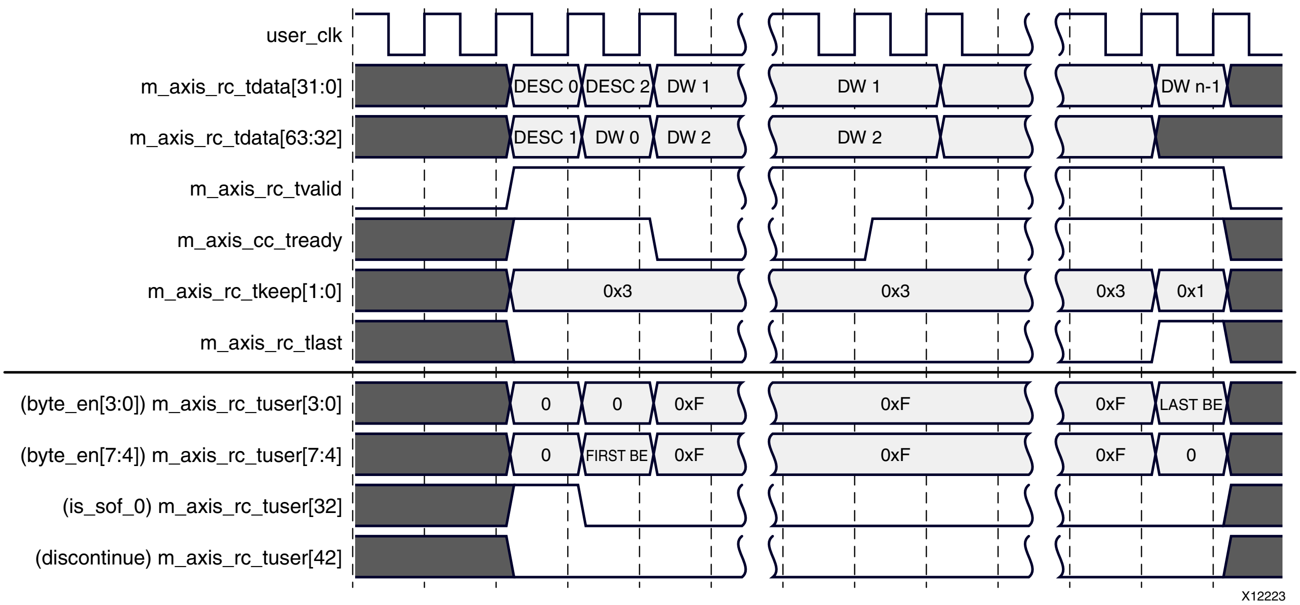

Figure 3-53: **Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode, Interface Width = 64 Bits)**

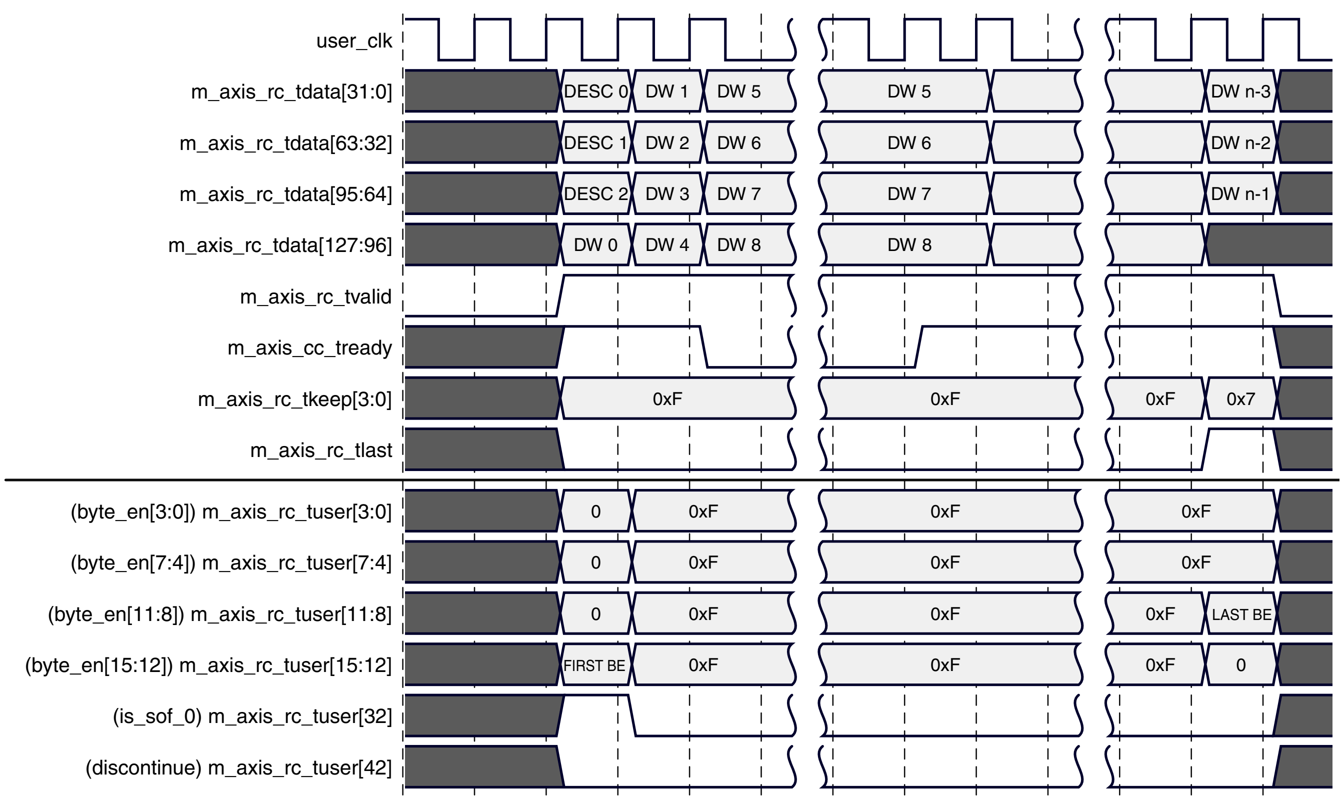

Figure 3-54: **Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode, Interface Width = 128 Bits)**

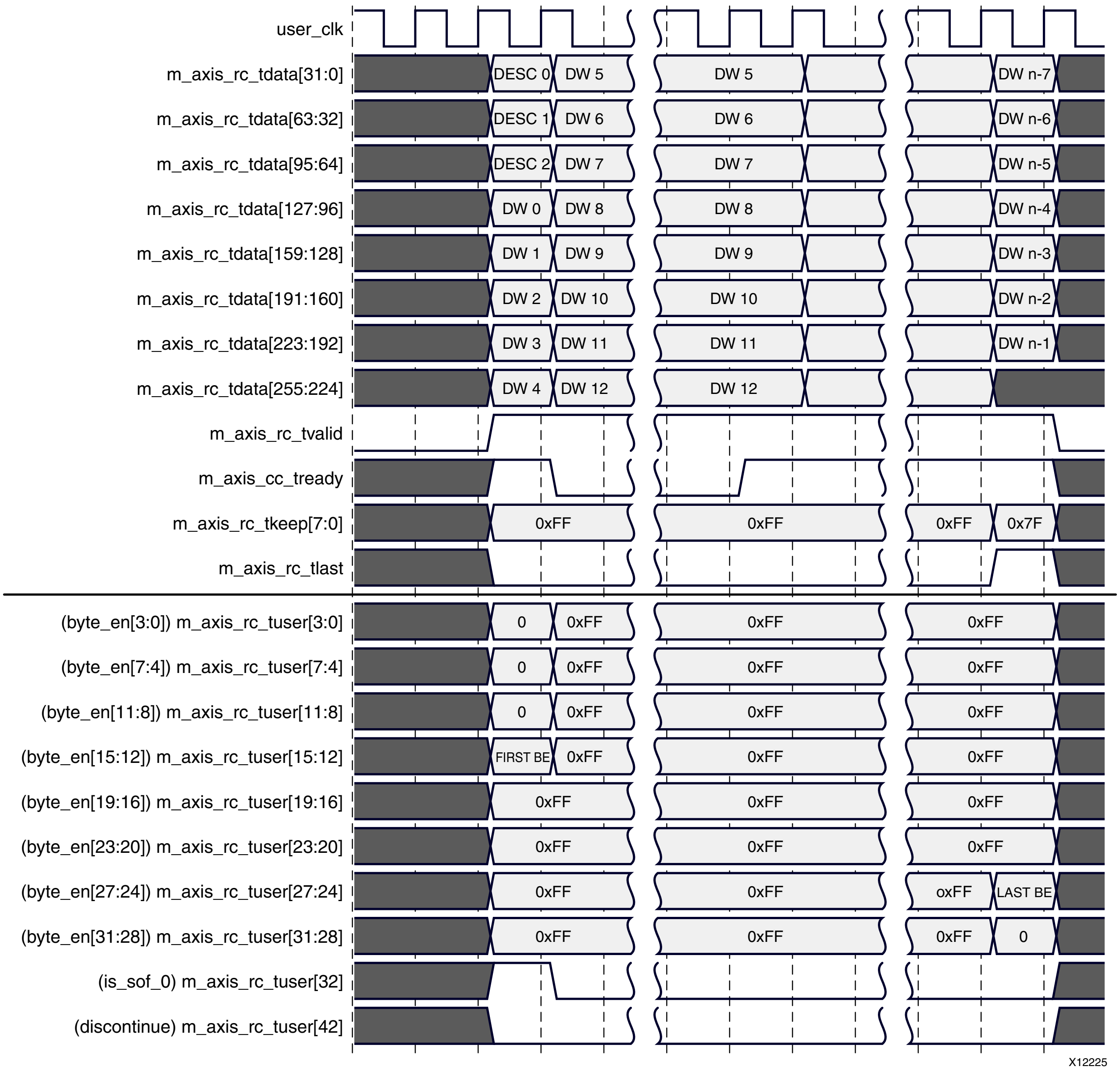

Figure 3-55: **Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode, Interface Width = 256 Bits)**

The timing diagrams in Figure 3-56, Figure 3-57, and Figure 3-58 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 * 8 + 1), for an integer m. The size of the data block is assumed to be n Dwords, for some $n = k * 32 + 28$, $k > 0$. The straddle option is not valid for address-aligned transfers, so the timing diagrams assume that the Completions are not straddled on the 256-bit interface.

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

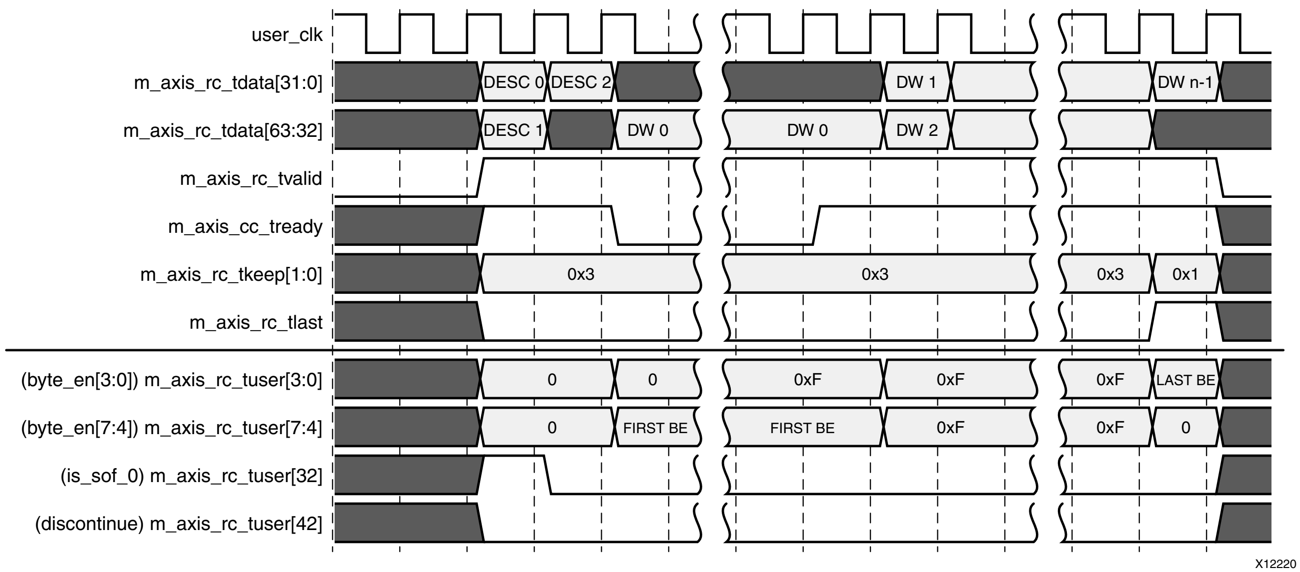

Figure 3-56: **Transfer of a Completion with Data on the Requester Completion Interface (Address-Aligned Mode, Interface Width = 64 Bits)**

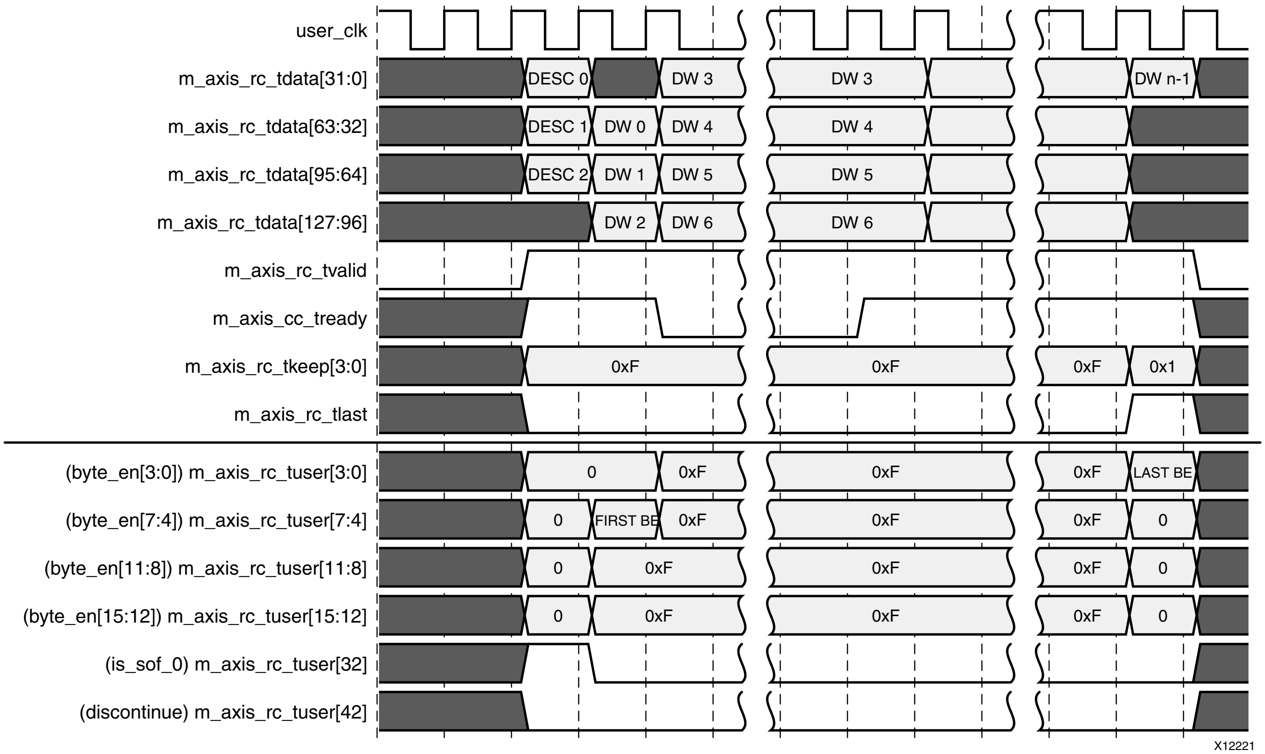

Figure 3-57: **Transfer of a Completion with Data on the Requester Completion Interface (Address-Aligned Mode, Interface Width = 128 Bits)**

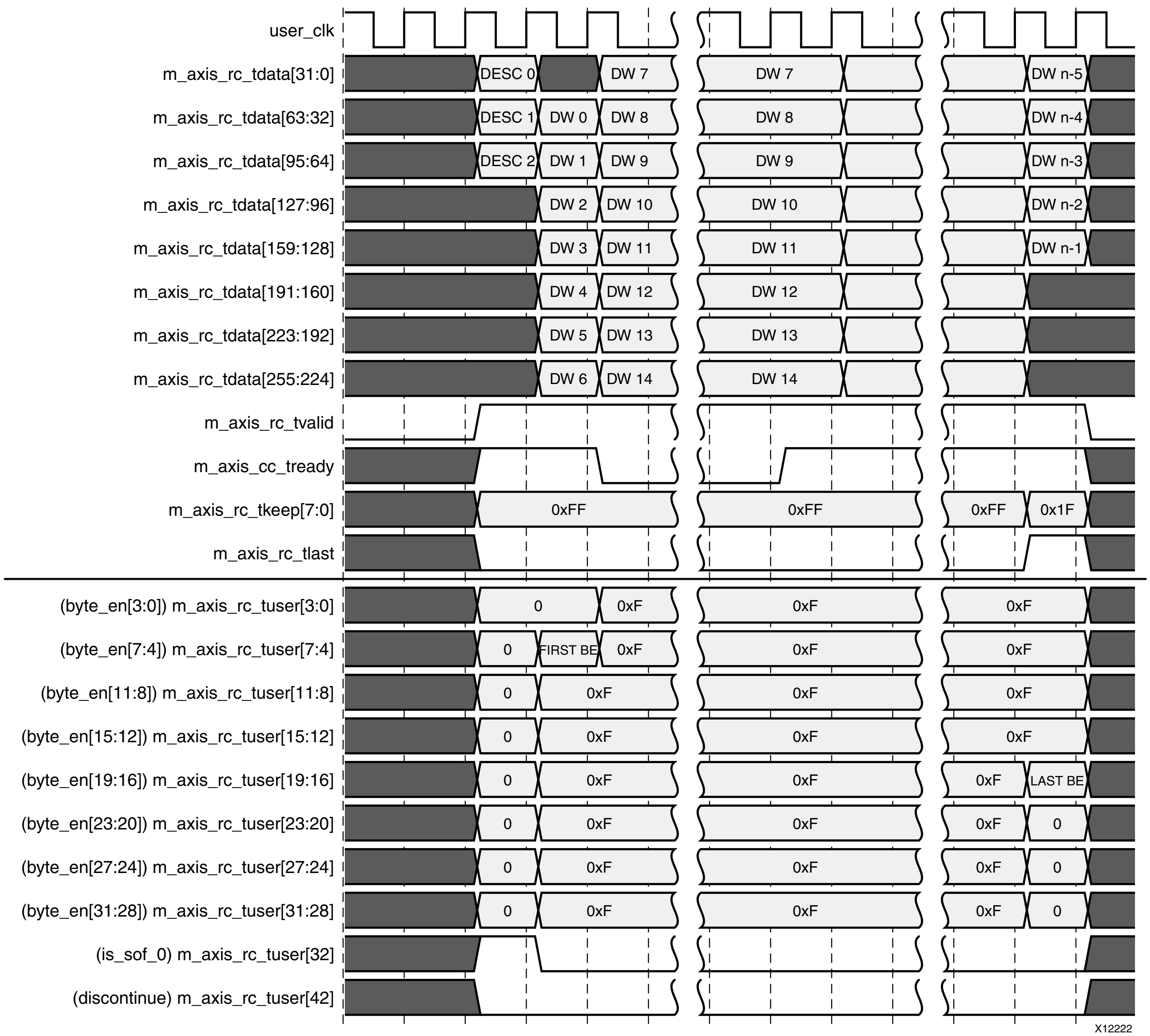

Figure 3-58: **Transfer of a Completion with Data on the Requester Completion Interface (Address-Aligned Mode, Interface Width = 256 Bits)**

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 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 High, the signals is_eof_0[0] and is_sof_0 are also High in the same beat. If is_sof_1 is High, is_sof_0 is High. If is_eof_1 is High, is_eof_0 is High.

Figure 3-59 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.

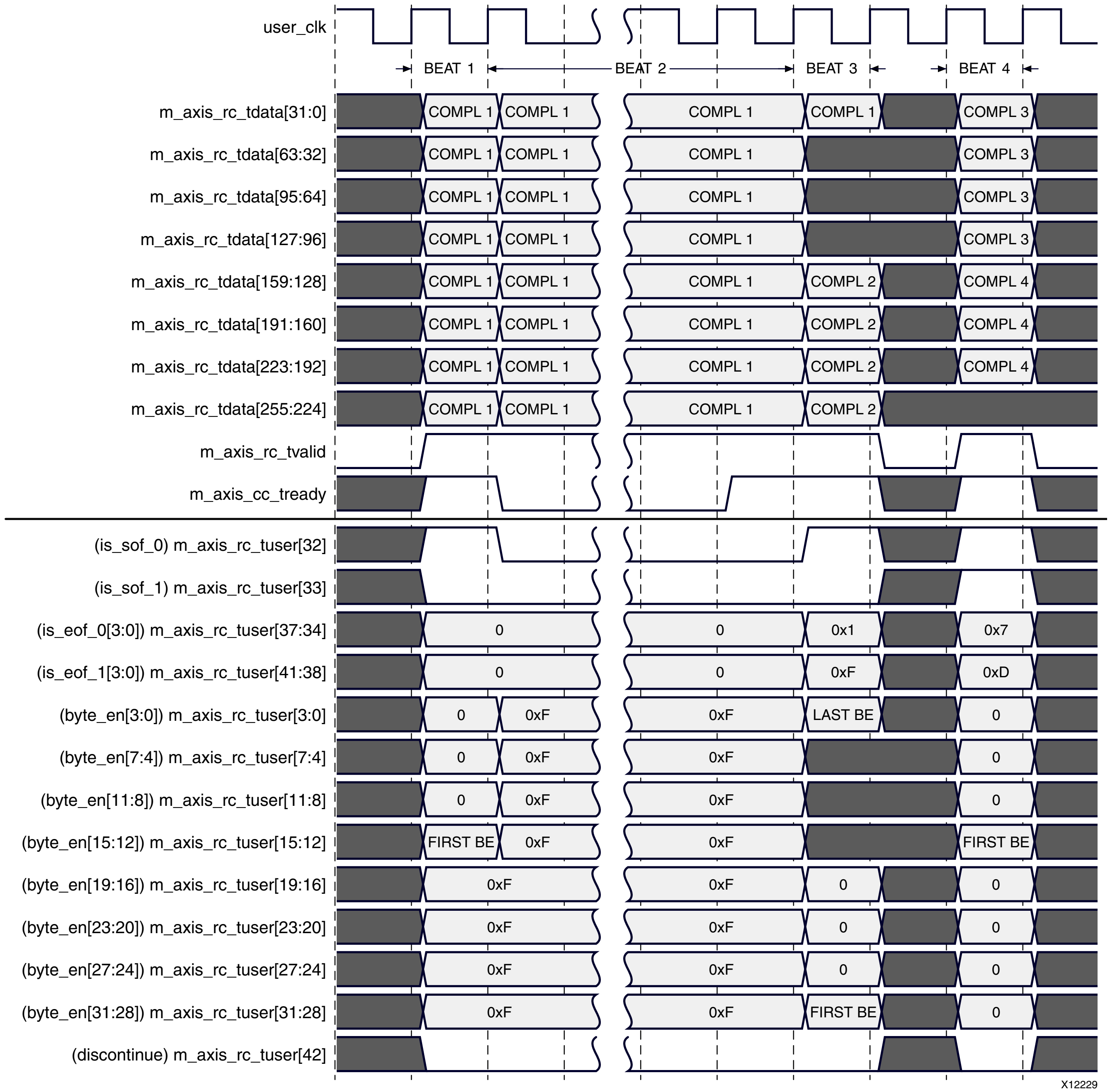

Figure 3-59: **Transfer of Completion TLPs on the Requester Completion Interface with the Straddle Option Enabled**

Aborting a Completion Transfer

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

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

Handling of Completion Errors

When a Completion TLP is received from the link, the integrated block matches it against the outstanding requests in the Split Completion Table to determine the corresponding request, and compares the fields in its header against the expected values to detect any error conditions. The integrated block then signals the error conditions in a 4-bit error code sent to the user application as part of the completion descriptor. The integrated block also indicates the last completion for a request by setting the Request Completed bit (bit 30) in the descriptor. Table 3-12 defines the error conditions signaled by the various error codes.

Table 3-12: **Encoding of Error Codes**

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

Table 3-12: **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 re-used 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 re-used 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 re-assigned 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 re-used.

Power Management

The core supports these power management modes:

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

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

For additional information on ASPM and PPM implementation, see the *PCI Express Base Specification* [Ref 2].

Active State Power Management

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

The Active State Power Management (ASPM) functionality is autonomous and transparent from a user-logic function perspective. The core supports the conditions required for ASPM. The integrated block supports ASPM L0s and ASPM L1. L0 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 targeted designs. It is supported only on designs generated for Gen1 and Gen2.

Programmed Power Management

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

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

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

PPM L0 State

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

PPM L1 State

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

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

   There are two exceptions to this rule:

   - The core is configured as an Endpoint and the User Configuration Space is enabled. In this situation, the user application must refrain from sending new Request TLPs if `cfg_function_power_state` indicates non-D0, but the user application can return Completions to Configuration transactions targeting User Configuration space.
   - The core is configured as a Root Port. To be compliant in this situation, the user application should refrain from sending new Requests if `cfg_function_power_state` indicates non-D0.
4. The core exchanges appropriate power management DLLPs with its link partner to successfully transition the link to a lower power PPM L1 state. This action is transparent to the user logic.
5. All user transactions are stalled for the duration of time when the device power state is non-D0, with the exceptions indicated in step 3.

PPM L3 State

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

1. The core negotiates a transition to the L23 Ready Link State upon receiving a PME_Turn_Off message from the upstream link partner.

2. Upon receiving a PME_Turn_Off message, the core initiates a handshake with the user logic through `cfg_power_state_change_interrupt` (see Table 3-13) and expects a `cfg_power_state_change_ack` back from the user logic.
3. A successful handshake results in a transmission of the Power Management Turn-off Acknowledge (PME-turnoff_ack) Message by the core to its upstream link partner.
4. The core closes all its interfaces, disables the Physical/Data-Link/Transaction layers and is ready for *removal* of power to the core.

   There are two exceptions to this rule:

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

Table 3-13: **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 when it detects assertion of `cfg_power_state_change_ack`.

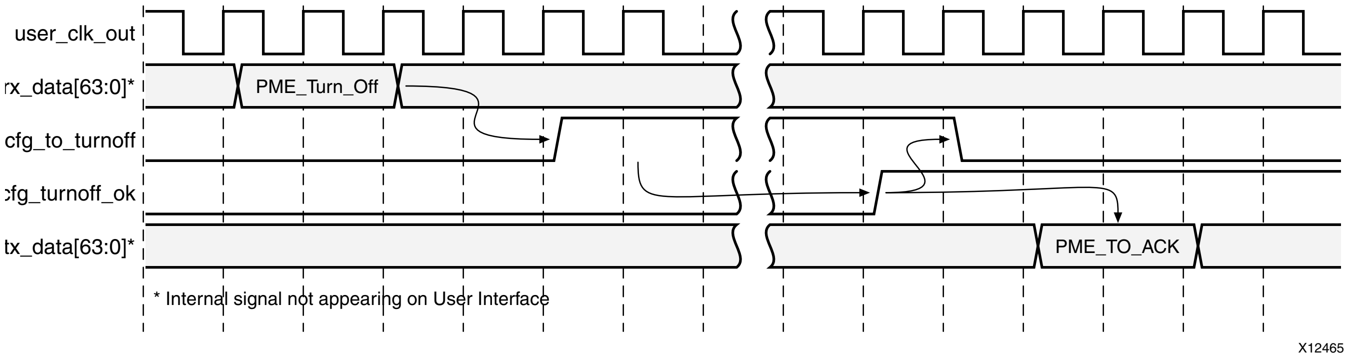

Figure 3-60: **Power Management Handshaking: 64-Bit**

Generating Interrupt Requests

See the `cfg_interrupt_msi*` and `cfg_interrupt_msix_*` descriptions in Table 2-20, page 52.

Note: This section only applies to the Endpoint Configuration of the Gen3 Integrated Block for PCIe core.

The integrated block core supports sending interrupt requests as either legacy, Message MSI, or MSI-X interrupts. The mode is programmed using the MSI Enable bit in the Message Control Register of the MSI Capability Structure and the MSI-X Enable bit in the MSI-X Message Control Register of the MSI-X Capability Structure. For more information on the MSI and MSI-X capability structures, see section 6.8 of the *PCI Local Base Specification v3.0*.

The state of the MSI Enable and MSI-X Enabled bits is reflected by the `cfg_interrupt_msi_enable` and `cfg_interrupt_msix_enable` outputs, respectively. Table 3-14 describes the Interrupt Mode to which the device has been programmed, based on the `cfg_interrupt_msi_enable` and `cfg_interrupt_msix_enable` outputs of the core.

Table 3-14: **Interrupt Modes**

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

The MSI Enable bit in the MSI control register, the MSI-X Enable bit in the MSI-X Control Register, and the Interrupt Disable bit in the PCI Command register are programmed by the Root Complex. The user application has no direct control over these bits.

The Internal Interrupt Controller in the core only generates Legacy Interrupts and MSI Interrupts. MSI-X Interrupts need to be generated by the user application and presented on the transmit AXI4-Stream interface. The status of `cfg_interrupt_msi_enable` determines the type of interrupt generated by the internal Interrupt Controller:

If the MSI Enable bit is set to a 1, then the core generates MSI requests by sending Memory Write TLPs. If the MSI Enable bit is set to `0`, the core generates legacy interrupt messages as long as the Interrupt Disable bit in the PCI Command Register is set to 0.

- `cfg_interrupt_msi_enable = 0`: Legacy interrupt
- `cfg_interrupt_msi_enable = 1`: MSI
- Command register bit 10 = `0`: INTx interrupts enabled
- Command register bit 10 = `1`: INTx interrupts disabled (requests are blocked by the core)

The user application can monitor `cfg_function_status` to check whether INTx interrupts are enabled or disabled. For more information, see Table 2-13.

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

Legacy Interrupt Mode

- The User Application first asserts `cfg_interrupt_int` and `cfg_interrupt_pending` to assert the interrupt.
- The core then asserts `cfg_interrupt_sent` to indicate the interrupt is accepted. On the following clock cycle, the User Application deasserts `cfg_interrupt_int` and, 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 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 ASSERT_INTA, which is indicated by the assertion of `cfg_interrupt_sent`. Deasserting `cfg_interrupt_int` causes the core to send DEASSERT_INTA. `cfg_interrupt_pending` must be asserted until the interrupt has been serviced, otherwise the interrupt status bit in the status register will not be updated correctly. If the software reads this bit, it detects no interrupt pending.

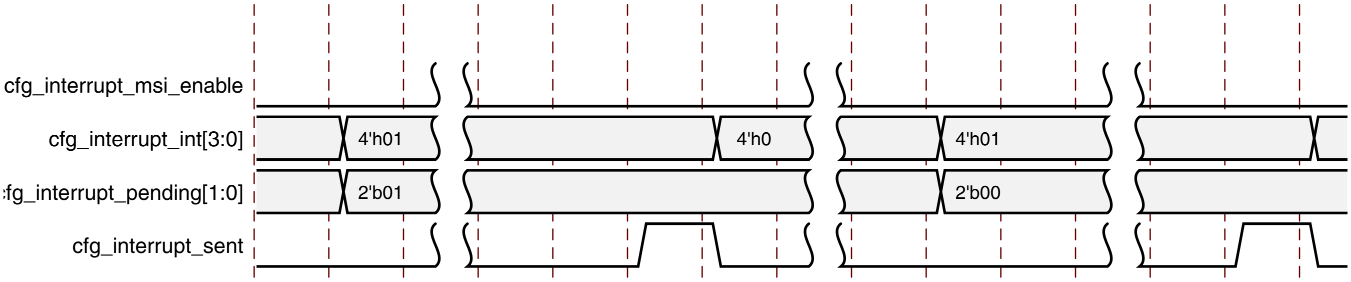

Figure 3-61: **Legacy Interrupt Signaling**

MSI Mode

- As shown in Figure 3-61, the User Application first asserts a value on `cfg_interrupt_msi_int`.
- The core asserts `cfg_interrupt_msi_sent` to signal that the interrupt is accepted and the core sends a MSI Memory Write TLP.

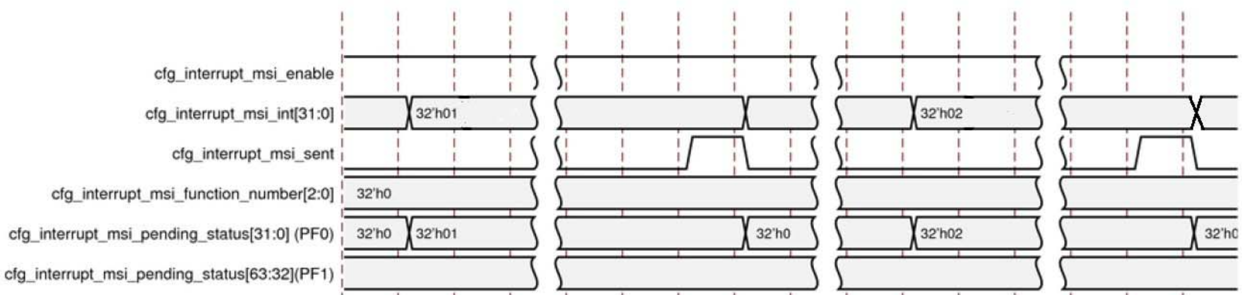

Figure 3-62: **MSI Mode**

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

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

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

This pseudo code shows the processing required:

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

if (cfg_interrupt_msienable) {       // MSI Enabled
  if (cfg_interrupt_mmenable > 0) {  // Multi-Vector MSI Enabled
    cfg_interrupt_msi_int = {Padding_0s, MSI_Vector_Num};
  } else {                           // Single-Vector MSI Enabled
    cfg_interrupt_msi_int = Padding_0s;
  }
} 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 = 00h`;
2. if `cfg_interrupt_mmenable[2:0] == 101b`, that is, 32 MSI Vectors Enabled, then `cfg_interrupt_msi_int` = {{27'b0}, {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 Gen3 Integrated Block for PCIe core optionally supports the MSI-X Capability Structure, as shown in Figure 3-63. The MSI-X vector table and the MSI-X Pending Bit Array need to be implemented as part of the user logic, by claiming a BAR aperture.

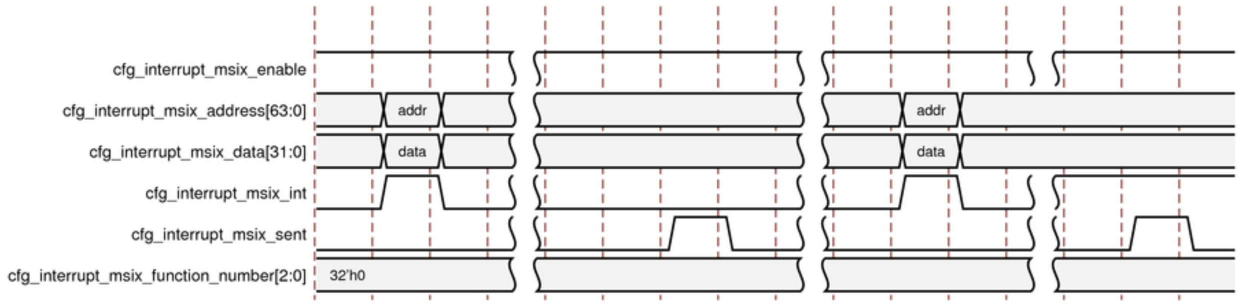

Figure 3-63: **MSI-X Mode**

Designing with Configuration Space Registers and Configuration Interface

The ports used by configuration registers are described in Table 2-13, page 28. Root Ports must use the Configuration Port to set up the Configuration Space. Endpoints can also use the Configuration Port to read and write; however, care must be taken to avoid adverse system side effects.

The User Application must supply the address as a Dword address, not a byte address.

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

For example:

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

- The Dword address of is `01h`.

  Note: The byte address is `04h`.

For BAR0:

- The Dword address is `04h`.

  Note: The byte address is `10h`.

To read any register in configuration space, the user application drives the register Dword address onto `cfg_mgmt_addr[9:0]`. `cfg_mgmt_addr[17:10]` selects the PCI Function associated with the configuration register. The core drives the content of the addressed register onto `cfg_mgmt_read_data[31:0]`. The value on `cfg_mgmt_read_data [31:0]` is qualified by signal assertion on `cfg_mgmt_read_write_done`. Figure 3-64 illustrates an example with read from the Configuration Space.

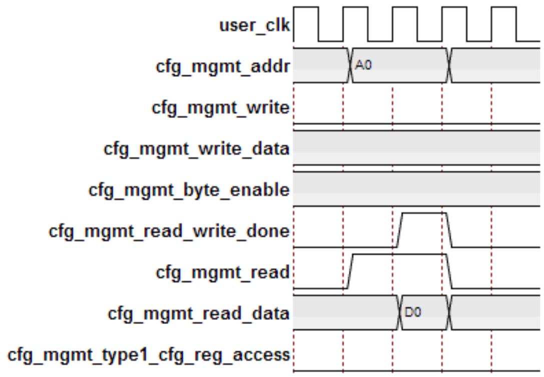

Figure 3-64: **cfg_mgmt_read_type0_type1**

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

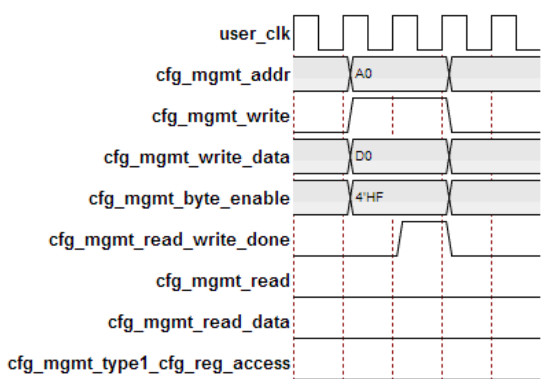

Figure 3-65: **cfg_mgmt_write_type0**

When the core is configured in the Root Port mode, when you assert `cfg_mgmt_type1_cfg_reg_access` input during a write to a Type-1 PCI™ Config Register forces a write into certain read-only fields of the register. This input has no effect when the core is in the Endpoint mode, or when writing to any register other than a Type-1 Config Register.

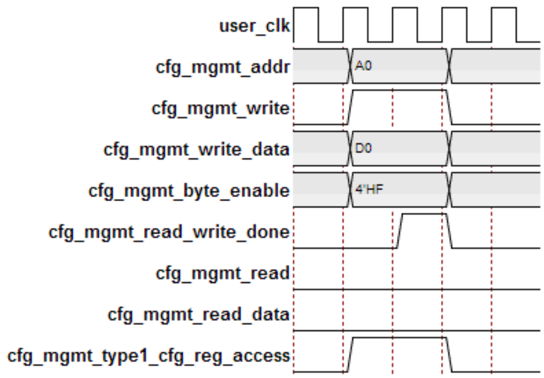

Figure 3-66: **cfg_mgmt_write_type1_override**

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

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

Link Partner Supports Fewer Lanes

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

When the 4-lane core is connected to a device that implements 1 lane, the 4-lane core trains and operates as a 1-lane device using lane 0, as shown in Figure 3-67. Similarly, if the 4-lane core is connected to a 2-lane device, the core trains and operates as a 2-lane device using lanes 0 and 1.

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

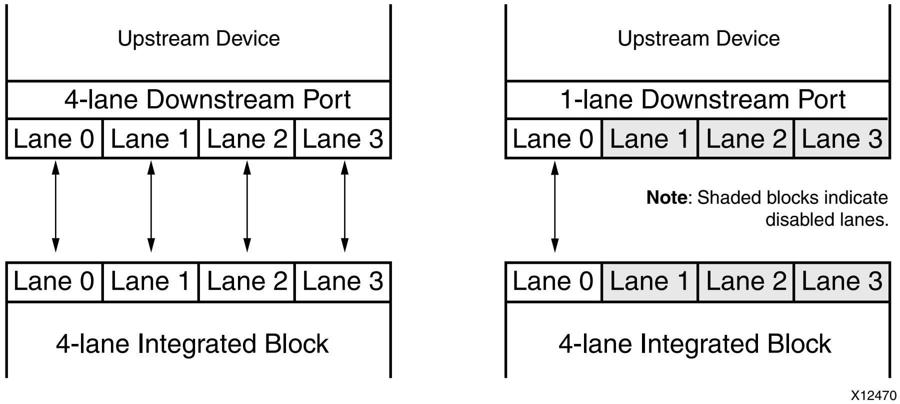

Figure 3-67: **Scaling of 4-Lane Endpoint Block from 4-Lane to 1-Lane Operation**

Lane Becomes Faulty

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

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

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

Lane Reversal

The integrated block supports limited lane reversal capabilities and therefore provides flexibility in the design of the board for the link partner. The link partner can choose to lay out the board with reversed lane numbers and the integrated block continues to link train successfully and operate normally. The configurations that have lane reversal support are x8 and x4 (excluding downshift modes). Downshift refers to the link width negotiation process that occurs when link partners have different lane width capabilities advertised. As a result of lane width negotiation, the link partners negotiate down to the smaller of the two advertised lane widths. Table 3-15 describes the several possible combinations including downshift modes and availability of lane reversal support.

Table 3-15: **Lane Reversal Support**

Integrated Block Advertised Lane Width	Negotiated Lane Width	Lane Number Mapping (Endpoint Link Partner)		Lane Reversal Supported
		Endpoint	Link Partner	
x8	x8	Lane 0... Lane 7	Lane 7... Lane 0	Yes
x8	x4	Lane 0... Lane 3	Lane 7... Lane 4	No(1)
x8	x2	Lane 0... Lane 3	Lane 7... Lane 6	No(1)
x4	x4	Lane 0... Lane 3	Lane 3... Lane 0	Yes
x4	x2	Lane 0... Lane 1	Lane 3... Lane 2	No(1)
x2	x2	Lane 0... Lane 1	Lane 1... Lane 0	Yes
x2	x1	Lane 0... Lane 1	Lane 1	No(1)

Notes:

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

Chapter 4

Design Flow Steps

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

- *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [Ref 5]
- *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 4]
- *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 6]
- *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 8]

Customizing and Generating the Core

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

IMPORTANT: *If you are customizing and generating the core in the Vivado IP Integrator, see the Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator (UG994) [Ref 5] for detailed information. IP Integrator might auto-compute certain configuration values when validating or generating the design. To check whether the values do change, see the description of the parameter in this chapter. To view the parameter value you can run the* `validate_bd_design` *command in the Tcl console.*

You can customize the Gen3 Integrated Block for PCIe core for use in your design by specifying values for the various parameters associated with the IP core using the following steps:

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

For further details:

- See the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 4]
- See the *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 6]

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

The Customize IP dialog box for the UltraScale FPGAs Gen3 Integrated Block for PCIe FPGAs consists of two modes: Basic Mode and Advanced Mode. To select a mode, use the **Mode** drop-down list on the first page of the Customize IP dialog box.

Basic Mode

The Basic mode parameters are explained in this section.

Basic Parameter Settings

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

Component Name

Base name of the output files generated for the core. The name must begin with a letter and can be composed of these characters: a to z, 0 to 9, and "_."

Mode

Allows you to select the Basic or Advanced mode of the configuration of core.

PCIe Device / Port Type

Indicates the PCI Express logical device type.

PCIe Block Location

Selects from the available integrated blocks to enable generation of location-specific constraint files and pinouts. This selection is used in the default example design scripts.

This option is not available if a Xilinx Development Board is selected.

Number of Lanes

The core requires the selection of the initial lane width. Table 4-1 defines the available widths and associated generated core. Wider lane width cores can train down to smaller lane widths if attached to a smaller lane-width device. See Link Training: 2-Lane, 4-Lane, and

8-Lane Components for more information.

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

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

Maximum Link Speed

The core allows you to select the Maximum Link Speed supported by the device. Table 4-2 defines the lane widths and link speeds supported by the device. Higher link speed cores are capable of training to a lower link speed if connected to a lower link speed capable device.

Note: Currently only x8 lane width at 8 Gb/s is supported.

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

Lane Width	Link Speed
x8	8 Gb/s

AXI-ST Interface Width

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

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

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

AXI-ST Interface Frequency

The frequency is set to 250 Mhz.

AXI-ST Alignment Mode

When a payload is present, there are two options for aligning the first byte of the payload with respect to the datapath. See Data Alignment Options, page 78.

Requestor Completion Straddle

The core provides an option to straddle packets on the Requestor Completion interface when the interface width is 256 bits. See Straddle Option for 256-Bit Interface, page 147.

Enable Client Tag

Enables you to use the client Tag.

Reference Clock Frequency

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

Xilinx Development Board

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

Silicon Revision

Selects the silicon revision. The possible option is Production.

Enable External PIPE Interface

When this option is enabled, an external third-party bus functional model (BFM) can be connected to the PIPE interface of the PCIe Integrated Block. This option is available for both Endpoint and Root Port mode. This option is available only when the **Include Shared Logic (Clocking) in example design** option is selected.

Capabilities

The Capabilities settings are explained in this section.

Enable Physical Function 0

The core implements an additional Physical Function (PF).

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

MPS

This field indicates the maximum payload size that the device or function can support for TLPs. This is the value advertised to the system in the Device Capabilities Register.

Extended Tag

This field indicates the maximum supported size of the Tag field as a Requester. The options are:

- When selected, 6-bit Tag field support
- When deselected, 5-bit Tag field support

Slot Clock Configuration

Enables the Slot Clock Configuration bit in the Link Status register. When you select this option, the link is synchronously clocked. For more information on clocking options, see Clocking.

Identity Settings (PF0 IDs and PF1 IDs)

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

PF0 ID Initial Values

- **Vendor ID:** Identifies the manufacturer of the device or application. Valid identifiers are assigned by the PCI Special Interest Group to guarantee that each identifier is unique. The default value, `10EEh`, is the Vendor ID for Xilinx. Enter a vendor identification number here. `FFFFh` is reserved.
- **Device ID:** A unique identifier for the application; the default value, which depends on the configuration selected, is 70<*link speed*><*link width*>h. This field can be any value; change this value for the application.
- **Revision ID:** Indicates the revision of the device or application; an extension of the Device ID. The default value is `00h`; enter values appropriate for the application.
- **Subsystem Vendor ID:** Further qualifies the manufacturer of the device or application. Enter a Subsystem Vendor ID here; the default value is `10EEh`. Typically, this value is the same as Vendor ID. Setting the value to `0000h` can cause compliance testing issues.
- **Subsystem ID:** Further qualifies the manufacturer of the device or application. This value is typically the same as the Device ID; the default value depends on the lane width and link speed selected. Setting the value to `0000h` can cause compliance testing issues.

Class Code

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

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

Class code encoding can be found at www.pcisig.com.

Class Code Look-up Assistant

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

Base Address Registers (PF0 and PF1)

The Base Address Registers (BARs) screens set the base address register space for the Endpoint configuration. Each BAR (0 through 5) configures the BAR Aperture Size and Control attributes of the Physical Function.

Base Address Register Overview

In Endpoint configuration, the core supports up to six 32-bit BARs or three 64-bit BARs, and the Expansion read-only memory (ROM) BAR. In Root Port configuration, the core supports up to two 32-bit BARs or one 64-bit BAR, and the Expansion ROM BAR.

BARs can be one of two sizes:

- **32-bit BARs:** The address space can be as small as 16 bytes or as large as 2 gigabytes. Used for Memory to I/O.
- **64-bit BARs:** The address space can be as small as 128 bytes or as large as 8 exabytes. Used for Memory only.

All BAR registers share these options:

- **Checkbox:** Click the checkbox to enable the BAR; deselect the checkbox to disable the BAR.
- **Type:** BARs can either be I/O or Memory.
    - *I/O*: I/O BARs can only be 32-bit; the Prefetchable option does not apply to I/O BARs. I/O BARs are only enabled for the Legacy PCI Express Endpoint core.
    - *Memory*: Memory BARs can be either 64-bit or 32-bit and can be prefetchable. When a BAR is set as 64 bits, it uses the next BAR for the extended address space and makes the next BAR inaccessible.
- **Size:** The available Size range depends on the PCIe Device/Port Type and the Type of BAR selected. Table 4-4 lists the available BAR size ranges.

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

PCIe Device / Port Type	BAR Type	BAR Size Range
PCI Express Endpoint	32-bit Memory	128 Bytes – 2 Gigabytes
	64-bit Memory	128 Bytes – 8 Exabytes

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

PCIe Device / Port Type	BAR Type	BAR Size Range
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.

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

Expansion ROM Base Address Register

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

Managing Base Address Register Settings

Memory, I/O, Type, and Prefetchable settings are handled by setting the appropriate settings for the desired base address register.

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

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

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

Disabling Unused Resources

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

Legacy/MSI Capabilities

On this page, you set the Legacy Interrupt Settings and MSI Capabilities for all applicable Physical and Virtual Functions.

Legacy Interrupt Settings

- **Enable INTX**: Enables the ability of the PCI Express function to generate INTx interrupts.
- **Interrupt PIN**: Indicates the mapping for Legacy Interrupt messages. A setting of "None" indicates no Legacy Interrupts are used.

MSI Capabilities

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

  Note: Although it is possible not to enable MSI or MSI-X, the result would be a non-compliant core. The *PCI Express Base Specification* [Ref 2] requires that MSI, MSI-X, or both be enabled.
- **Multiple Message Capable**: Selects the number of MSI vectors to request from the Root Complex.
- **Per Vector Masking Capable**: Indicates that the function supports MSI per-vector Masking.

Advanced Mode

The Customize IP dialog box provides configuration options described in this section.

Basic

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

Use the dedicated PERST routing resources

Enables sys_rst dedicated routing for the PCIE_X0Y0 block.

System reset polarity

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

Capabilities

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

SRIOV Capabilities

Enables Single Root Port I/O Virtualization (SRIOV) capabilities. The integrated block implements extended Single Root Port I/O Virtualization PCIe. When this is enabled, SRIOV is implemented for both PF0 and PF1 (if selected).

Function Level Reset

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

Device Capabilities Registers 2

Specifies options for AtomicOps and TPH Completer support. See the Device Capability Register 2 description in Chapter 7 of the *PCI Express Base Specification* [Ref 2] for more information. These settings apply to both physical functions if PF1 is enabled.

PF0 ID and PF1 ID

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

PF0 BAR and PF1 BAR

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

SRIOV Config (PF0 and PF1)

SRIOV Capability Version

Indicates the 4-bit SRIOV Capability version for the Physical Function.

SRIOV Function Select

Indicates the number of Virtual Functions associated to the Physical Function. A maximum of six Virtual Functions are available to PF0 and PF1.

SRIOV Functional Dependency Link

Indicates the SRIOV Functional Dependency Link for the Physical Function. The programming model for a device can have vendor-specific dependencies between sets of functions. The Function Dependency Link field is used to describe these dependencies.

SRIOV First VF Offset

Indicates the offset of the first Virtual Function (VF) for the Physical Function (PF). PF0 always resides at Offset 0, and PF1 always resides at Offset 1. Six Virtual Functions are available in the Gen3 Integrated Block for PCIe core and reside at the function number range 64–69.

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

The PFx_FIRST_VF_OFFSET is calculated by taking the first offset of the Virtual Function and subtracting that from the offset of the Physical Function.

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

In the example above, the following offsets are used:

```
PF0_FIRST_VF_OFFSET = (64 - 0) = 64
PF1_FIRST_VF_OFFSET = (66 - 1) = 65
```

PF0 is always 64 assuming that PF0 has one or more Virtual Functions. The initial offset for PF1 is a function of how many VFs are attached to PF0 and is defined in the following pseudo code:

```
PF1_FIRST_VF_OFFSET = 63 + NUM_PF0_VFS
```

SRIOV VF Device ID

Indicates the 16-bit Device ID for all Virtual Functions associated with the Physical Function.

SRIOV Supported Page Size

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

PF0 SRIOV BARs and PF1 SRIVO BARs

The SRIOV Base Address Registers (BARs) set the base address register space for the Endpoint configuration. Each BAR (0 through 5) configures the SRIOV BAR Aperture Size and SRIOV Control attributes.

Table 4-5: **Example Virtual Function Mappings**

Physical Function	Virtual Function	Function Number Range
PF0	VF0	64
PF0	VF1	65
PF1	VF0	66
PF1	VF1	67
PF1	VF1	68

SRIOV Base Address Register Overview

In Endpoint configuration, the core supports up to six 32-bit BARs or three 64-bit BARs. In Root Port configuration, the core supports up to two 32-bit BARs or one 64-bit BAR. SRIOV BARs can be one of two sizes:

- **32-bit BARs**: The address space can be as small as 16 bytes or as large as 2 GB. Used for memory to I/O.
- **64-bit BARs**: The address space can be as small as 128 bytes or as large as 8 exabytes. Used for memory only.

All SRIOV BAR registers have these options:

- **Checkbox**: Click the checkbox to enable the BAR; deselect the checkbox to disable the BAR.
- **Type**: SRIOV BARs can either be I/O or Memory.
  - *I/O*: I/O BARs can only be 32-bit; the Prefetchable option does not apply to I/O BARs. I/O BARs are only enabled for the Legacy PCI Express Endpoint core.
  - *Memory*: Memory BARs can be either 64-bit or 32-bit and can be prefetchable. When a BAR is set as 64 bits, it uses the next BAR for the extended address space and makes the next BAR inaccessible.
- **Size**: The available size range depends on the PCIe device/port type and the type of BAR selected. Table 4-6 lists the available BAR size ranges.

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

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

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

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

Managing SRIOV Base Address Register Settings

Memory, I/O, Type, and Prefetchable settings are handled by setting the appropriate Customize IP dialog box settings for the desired base address register.

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

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

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

Disabling Unused Resources

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

Legacy/MSI Capabilities

This page is the same as that of Basic mode.

MSI-X Capabilities

Available in Advanced mode only.

- **Enable MSIx Capability Structure**: Indicates that the MSI-X Capability structure exists.

  Note: The Capability Structure needs at least one Memory BAR to be configured. You must maintain the MSI-X Table and Pending Bit Array in the application.

- **MSIx Table Settings**: Defines the MSI-X Table structure.
  - **Table Size**: Specifies the MSI-X Table size.
  - **Table Offset**: Specifies the offset from the Base Address Register that points to the base of the MSI-X Table.
  - **BAR Indicator**: Indicates the Base Address Register in the Configuration Space used to map the function in the MSI-X Table onto memory space. For a 64-bit Base Address Register, this indicates the lower DWORD.

- **MSIx Pending Bit Array (PBA) Settings**: Defines the MSI-X Pending Bit Array (PBA) structure.
    - **PBA Offset**: Specifies the offset from the Base Address Register that points to the base of the MSI-X PBA.
    - **PBA BAR Indicator**: Indicates the Base Address Register in the Configuration Space used to map the function in the MSI-X PBA onto Memory Space.

Power Management

The Power Management page includes settings for the Power Management Registers, power consumption, and power dissipation options. These settings apply to both Physical Functions, if PF1 is enabled.

- **D1 Support**: Indicates that the function supports the D1 Power Management State. See section 3.2.3 of the *PCI Bus Power Management Interface Specification Revision 1.2* [Ref 2].
- **PME Support From**: Indicates the power states in which the function can assert `cfg_pm_wake`. See section 3.2.3 of the *PCI Bus Power Management Interface Specification Revision 1.2* [Ref 2].
- **BRAM Configuration Options**: Specify the number of receive block RAMs used for the solution. The table displays the number of receiver credits available for each packet type.

Extended Capabilities 1 and Extended Capabilities 2

The PCIe Extended Capabilities allow you to enable PCI Express Extended Capabilities. The Advanced Error Reporting Capability (offset `0x100h`) is always enabled. The Customize IP dialog box sets up the link list based on the capabilities enabled. After enabling, you must configure the capability by setting the applicable attributes in the core top-level defined in Output Generation.

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

Additional Optional Capabilities

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

Shared Logic

Enables you to share common blocks across multiple instantiations by selecting one or more of the options on this page. For more information, see Shared Logic in Chapter 3.

Core Interface Parameters

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

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

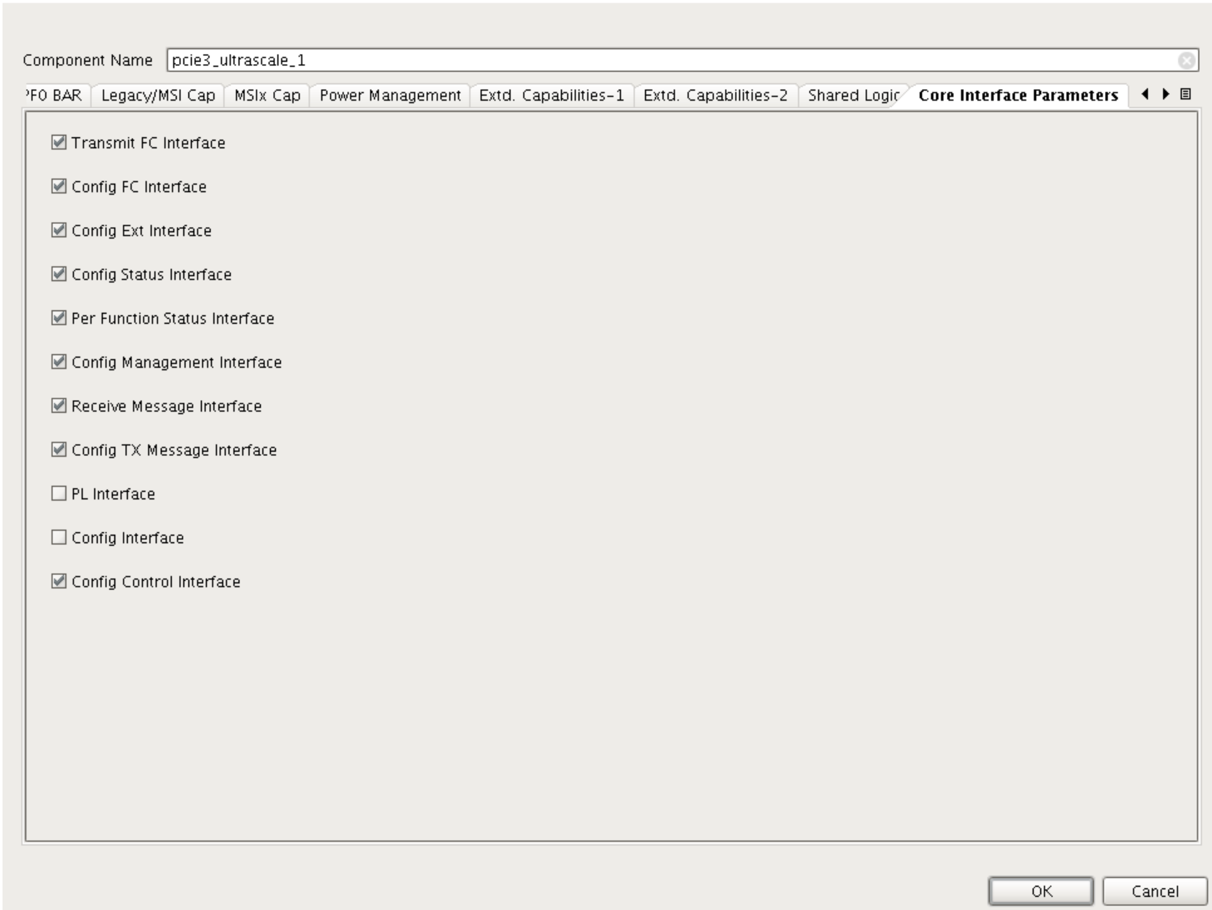

Figure 4-1: **Core Interfaces Parameters**

Tansmit FC Interface

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

- pcie_tfc_nph_av
- pcie_tfc_npd_av

Config FC Interface

Enables you to control the configuration flow control for the UltraScale FPGAs Gen3 Integrated Block for PCIe core. When you disable the Config Flow Control (FC) Interface option, the following ports are removed from the core:

- cfg_fc_ph
- cfg_fc_pd
- cfg_fc_nph
- cfg_fc_npd
- cfg_fc_cplh
- cfg_fc_cpld
- cfg_fc_sel

Config External Interface

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

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

Config Status Interface

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

- cfg_phy_link_down
- cfg_phy_link_status
- cfg_negotiated_width
- cfg_current_speed
- cfg_max_payload
- cfg_max_read_req
- cfg_function_status
- cfg_vf_status
- cfg_function_power_state
- cfg_vf_power_state
- cfg_link_power_state
- cfg_err_cor_out
- cfg_err_nonfatal_out
- cfg_err_fatal_out
- cfg_ltr_enable
- cfg_ltssm_state
- cfg_rcb_status
- cfg_dpa_substate_change
- cfg_obff_enable
- cfg_pl_status_change
- cfg_tph_requester_enable
- cfg_tph_st_mode
- cfg_vf_tph_requester_enable
- cfg_vf_tph_st_mode
- pcie_rq_seq_num
- pcie_rq_seq_num_vld
- pcie_cq_np_req_count
- pcie_rq_tag

- pcie_rq_tag_vld
- pcie_cq_np_req

Per Function Status Interface

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

- cfg_per_func_status_control
- cfg_per_func_status_data

Config Management Interface

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

- cfg_mgmt_addr
- cfg_mgmt_write
- cfg_mgmt_write_data
- cfg_mgmt_byte_enable
- cfg_mgmt_read
- cfg_mgmt_read_data
- cfg_mgmt_read_write_done
- cfg_mgmt_type1_cfg_reg_access

Receive Message Interface

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

- cfg_msg_received
- cfg_msg_received_data
- cfg_msg_received_type

Config Transmit Message Interface

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

- cfg_msg_transmit
- cfg_msg_transmit_type
- cfg_msg_transmit_data
- cfg_msg_transmit_done

Physical Layer Interface

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

- pl_eq_in_progress
- pl_eq_phase
- pl_eq_reset_eieos_count
- pl_gen2_upstream_prefer_deemph

Config Interface

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

- conf_req_data
- conf_req_ready
- conf_req_reg_num
- conf_req_type
- conf_req_valid
- conf_resp_rdata
- conf_resp_valid

Config Control Interface

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

- cfg_hot_reset_in
- cfg_hot_reset_out

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

Output Generation

For details, see "Generating IP Output Products" in the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 4].

Constraining the Core

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

Required Constraints

The UltraScale FPGAs Gen3 Integrated Block for PCIe solution requires the specification of timing and other physical implementation constraints to meet specified performance requirements for PCI Express®. These constraints are provided with the Endpoint and Root

Port solutions in a Xilinx Device Constraints (XDC) file. Pinouts and hierarchy names in the generated XDC correspond to the provided example design.

To achieve consistent implementation results, an XDC containing these original, unmodified constraints must be used when a design is run through the Xilinx tools. For additional details on the definition and use of an XDC or specific constraints, see *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 7].

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

Device, Package, and Speed Grade Selections

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

IMPORTANT: *Because Gen3 Integrated Block for PCIe cores are designed for specific part and package combinations, this section should not be modified.*

The device selection section always contains a part selection line, but can also contain part or package-specific options. An example part selection line follows:

```
CONFIG PART = XCKU040-ffva1156-3-e-es1
```

Clock Frequencies

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

Clock Management

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

Clock Placement

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

Simulation

For comprehensive information about Vivado simulation components, as well as information about using supported third-party tools, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 8].

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

Pipe Mode Simulation

The UltraScale FPGAs Gen3 Integrated Block for PCIe core supports the External PIPE mode interface, which can be hooked to third-party PCI Express VIPs/BFMs. The current Vivado solution provides the hook to the Rootport model for Endpoint mode and to the EP model for RP mode.

Enable this feature by selecting the Enable External PIPE Interface parameter on the Basic page of the Customizing IP dialog box. The External PIPE Interface signals shown in Table A-5 are generated at the core boundary for access to the external device.

Selected this mode to increase the simulation speed. In this mode, the PIPE interface of the core is connected to the PIPE interface of the link partner.

TIP: *PIPE mode is for simulation only. Implementation is not supported.*

External PIPE Interface

There is another method for PIPE mode simulations where any external BFM/VIP can be connected to the PIPE interface of the Endpoint device to speed up the simulation time. Use the **Enable External PIPE Interface** option to enable or disable this feature. For details, see Enable External PIPE Interface, page 167.

Table 4-7 and Table 4-8 describe the PIPE bus signals available at the top level of the core and their corresponding mapping inside the EP core (`pcie_top`) PIPE signals.

Table 4-7: **Common In/Out Commands and Endpoint PIPE Signals Mappings**

In Commands	Endpoint PIPE Signals Mapping	Out Commands	Endpoint PIPE Signals Mapping
common_commands_in[0]	pipe_clk[(1)]	common_commands_out[0]	pipe_tx_rcvr_det_gt
common_commands_in[1]	core_clk[(2)]	common_commands_out[2:1]	pipe_tx_rate_gt
common_commands_in[2]	user_clk[(3)]	common_commands_out[3]	pipe_tx_deemph_gt
common_commands_in[3]	rec_clk[(4)]	common_commands_out[6:4]	pipe_tx_margin_gt
common_commands_in[4]	phy_rdy[(5)]	common_commands_out[7]	pipe_tx_swing_gt
common_commands_in[5]	mmcm_lock[(6)]	common_commands_out[8]	pipe_tx_reset_gt
common_commands_in[11:6]	pipe_tx_eqfs[(7)]	common_commands_out[16:9]	pipe_tx_slide_gt
common_commands_in[17:12]	pipe_rx_eqlf[(8)]		
common_commands_in[25:18]	pipe_rx_syncdone[(9)]		

Notes:

1. `pipe_clk` is a regenerated clock based on the phase of the AveryDesign Systems BFM clock signal aclk250M. When the link speed is Gen1, `pipe_clk` is 125 MHz. In Gen3, `pipe_clk` is 250 MHz.
2. `core_clk` is a Xilinx PCI Express Endpoint clock. In Gen3 x8 configuration, `core_clk` = 500 MHz.
3. `user_clk` is a Xilinx PCI Express Endpoint clock. In Gen3 x8 configuration, `user_clk` = 250 MHz.
4. `rec_clk` is a Xilinx PCI Express Endpoint clock. Tie it to the `pipe_clk` signal.
5. `phy_rdy` should be asserted after 10 μs.
6. `mmcm_lock` can be asserted after 10 ns.
7. Assign `6'd40` to `pipe_tx_eqfs`.
8. Assign `6'd15` to `pipe_tx_eqlf`.
9. Assign `8'd0` to `pipe_rx_syncdone`.

Table 4-8: **Input/Output Bus with Endpoint PIPE Signals Mapping**

Input Bus	Endpoint PIPE Signals Mapping	Output Bus	Endpoint PIPE Signals Mapping
pipe_rx_0_sigs[31:0]	pipe_rx0_data_gt	pipe_tx_0_sigs[31: 0]	pipe_tx0_data_gt
pipe_rx_0_sigs[33:32]	pipe_rx0_char_is_k_gt	pipe_tx_0_sigs[33:32]	pipe_tx0_char_is_k_gt
pipe_rx_0_sigs[34]	pipe_rx0_data_valid_gt	pipe_tx_0_sigs[34]	pipe_tx0_elec_idle_gt
pipe_rx_0_sigs[35]	pipe_rx0_elec_idle_gt	pipe_tx_0_sigs[35]	pipe_tx0_data_valid_gt
pipe_rx_0_sigs[36]	pipe_rx0_start_block_gt	pipe_tx_0_sigs[36]	pipe_tx0_start_block_gt
pipe_rx_0_sigs[38:37]	pipe_rx0_syncheader_gt	pipe_tx_0_sigs[38:37]	pipe_tx0_syncheader_gt
pipe_rx_0_sigs[41:39]	pipe_rx0_status_gt	pipe_tx_0_sigs[39]	pipe_tx0_polarity_gt
pipe_rx_0_sigs[42]	pipe_rx0_valid_gt	pipe_tx_0_sigs[41:40]	pipe_tx0_powerdown_gt
pipe_rx_0_sigs[43]	pipe_rx0_phy_status_gt	pipe_tx_0_sigs[43:42]	pipe_tx0_eqcontrol_gt
pipe_rx_0_sigs[44][(1)]	pipe_rx0_eqdone_gt	pipe_tx_0_sigs[47:44][(7)]	pipe_tx0_eqpreset_gt
pipe_rx_0_sigs[62:45][(2)]	pipe_rx0_eqcoeff_gt	pipe_tx_0_sigs[53:48][(7)]	pipe_tx0_eqdeemph_gt
pipe_rx_0_sigs[80:63][(3)]	pipe_rx0_eqlp_new_txcoef_forpreset_gt	pipe_tx_0_sigs[55:54]	pipe_rx0_eqcontrol_gt
pipe_rx_0_sigs[81][(4)]	pipe_rx0_eqlp_lffs_sel_gt	pipe_tx_0_sigs[58:56][(7)]	pipe_rx0_eqpreset_gt
pipe_rx_0_sigs[82][(5)]	pipe_rx0_eqlp_adaptdone_gt	pipe_tx_0_sigs[64:59][(7)]	pipe_rx0_eqlp_lffs_gt
pipe_rx_0_sigs[83][(6)]	pipe_rx0_eqdone_gt	pipe_tx_0_sigs[68:65][(7)]	pipe_rx0_eqlp_txpreset_gt
		pipe_tx_0_sigs[69]	pipe_tx0_compliance_gt

Notes:

1. Asserted whenever `pipe_tx0_eqcontrol_gt` (`pipe_tx_0_sigs[43:42]`) is toggled.
2. Assign `18'd2`.
3. Assign `18'd0`.
4. Assign `1'b1`.
5. Assign `1'b0`.
6. Asserted whenever `pipe_rx0_eqcontrol_gt` (`pipe_tx_0_sigs[55:54]`) is toggled.
7. Ignore these signals.

Synthesis and Implementation

For details about synthesis and implementation, see “Synthesizing IP” and “Implementing IP” in the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 4].

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

Chapter 5

Example Design

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

Overview of the Example Design

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

Integrated Block Endpoint Configuration Overview

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

- The Root Port Model, a test bench that generates, consumes, and checks PCI Express® bus traffic.
- The Programmed Input/Output (PIO) example design, a completer application for PCI Express. The PIO example design responds to Read and Write requests to its memory space and can be synthesized for testing in hardware.

Simulation Design Overview

For the simulation design, transactions are sent from the Root Port Model to the core (configured as an Endpoint) and processed by the PIO example design. Figure 5-1 illustrates the simulation design provided with the core. For more information about the Root Port Model, see Root Port Model Test Bench for Endpoint, page 202.

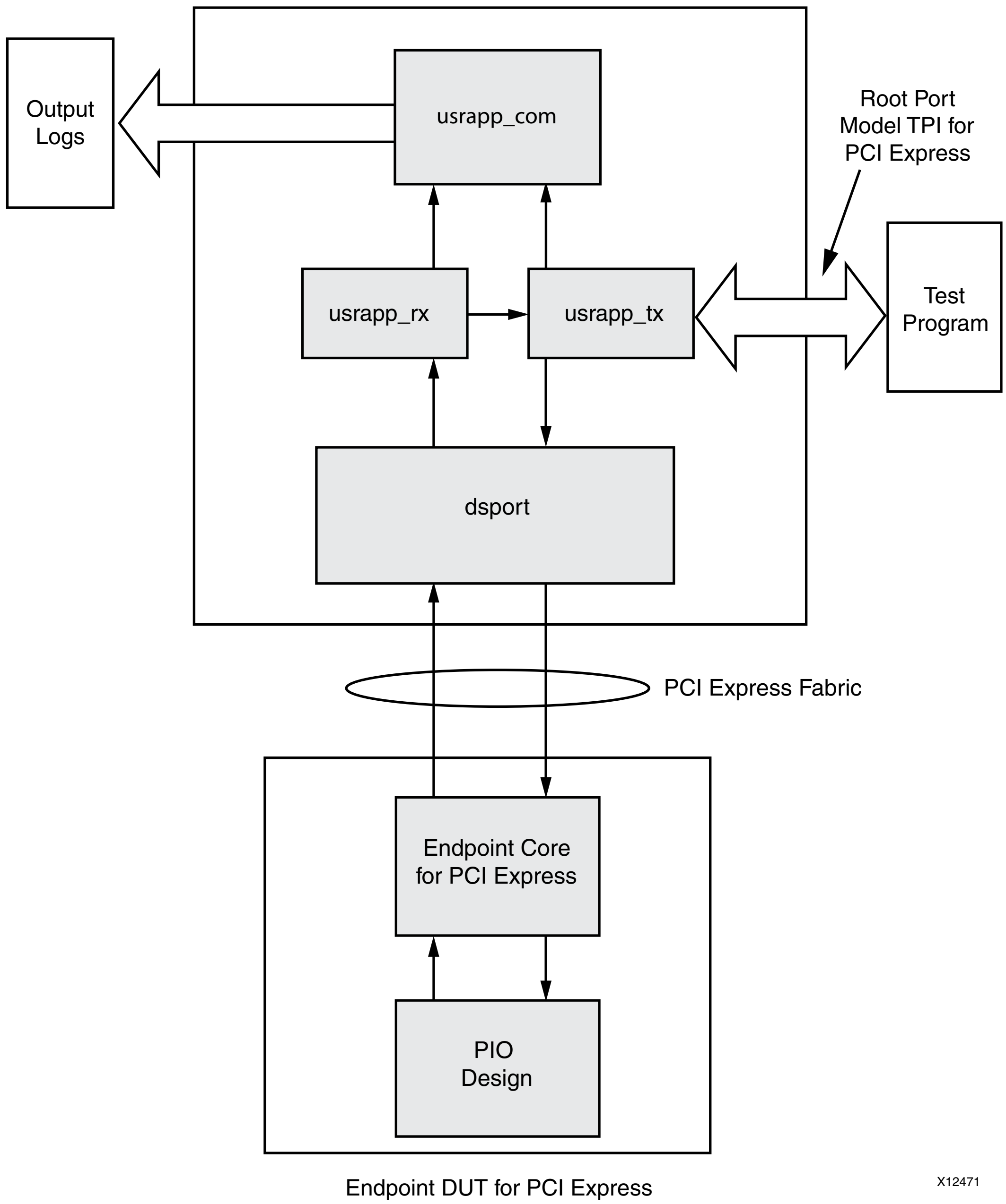

Figure 5-1: **Simulation Example Design Block Diagram**

Implementation Design Overview

The implementation design consists of a simple PIO example that can accept read and write transactions and respond to requests, as illustrated in Figure 5-2. Source code for the example is provided with the core. For more information about the PIO example design, see Programmed Input/Output: Endpoint Example Design, page 191.

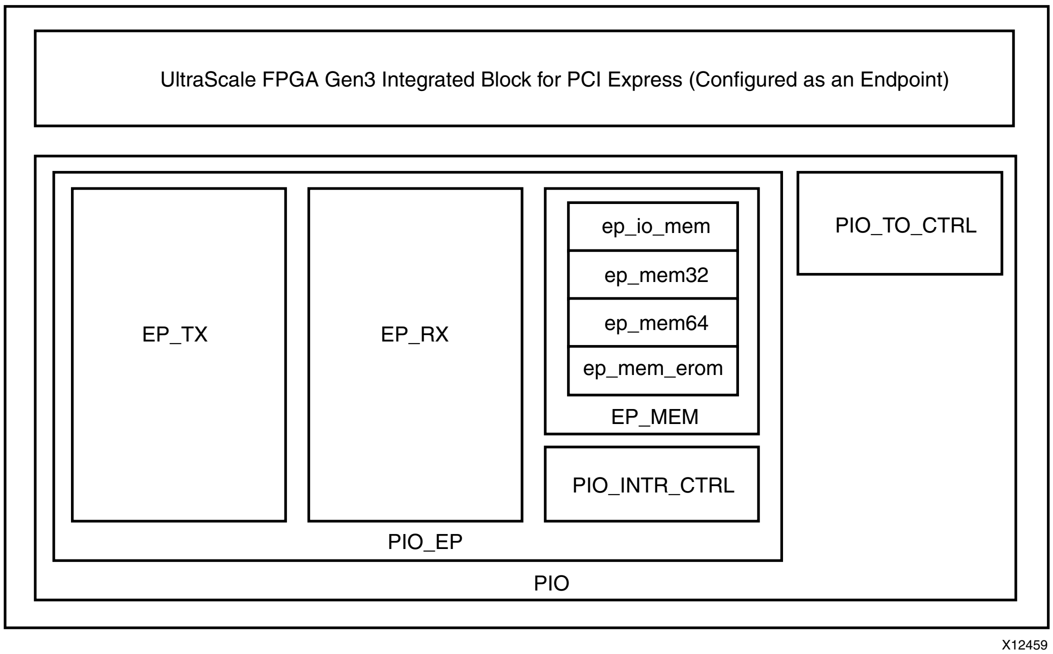

Figure 5-2: **Implementation Example Design Block Diagram**

Example Design Elements

The PIO example design elements include:

- Core wrapper
- An example Verilog HDL wrapper (instantiates the cores and example design)
- A customizable demonstration test bench to simulate the example design

The example design has been tested and verified with Vivado Design Suite and these simulators:

- Mentor Graphics Questa® SIM
- Vivado simulator

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

Programmed Input/Output: Endpoint Example Design

Programmed Input/Output (PIO) transactions are generally used by a PCI Express system host CPU to access Memory Mapped Input/Output (MMIO) and Configuration Mapped Input/Output (CMIO) locations in the PCI Express logic. Endpoints for PCI Express accept Memory and I/O Write transactions and respond to Memory and I/O Read transactions with Completion with Data transactions.

The PIO example design (PIO design) is included with the core in Endpoint configuration generated by the Vivado IP catalog, which allows users to bring up their system board with a known established working design to verify the link and functionality of the board.

The PIO design Port Model is shared by the core, Endpoint Block Plus for PCI Express, and Endpoint PIPE for PCI Express solutions. This section generically represents all solutions using the name Endpoint for PCI Express (or Endpoint for PCIe™).

System Overview

The PIO design is a simple target-only application that interfaces with the Endpoint for the PCIe core Transaction (AXI4-Stream) interface and is provided as a starting point for you to build their own designs. These features are included:

- Four transaction-specific 2 KB target regions using the internal FPGA block RAMs, providing a total target space of 8192 bytes
- Supports single Dword payload Read and Write PCI Express transactions to 32-/64-bit address memory spaces and I/O space with support for completion TLPs
- Utilizes the BAR ID[2:0] and Completer Request Descriptor[114:112] of the core to differentiate between TLP destination Base Address Registers
- Provides separate implementations optimized for 64-bit, 128-bit, and 256-bit AXI4-Stream interfaces

Figure 5-3 illustrates the PCI Express system architecture components, consisting of a Root Complex, a PCI Express switch device, and an Endpoint for PCIe. PIO operations move data *downstream* from the Root Complex (CPU register) to the Endpoint, and/or *upstream* from the Endpoint to the Root Complex (CPU register). In either case, the PCI Express protocol request to move the data is initiated by the host CPU.

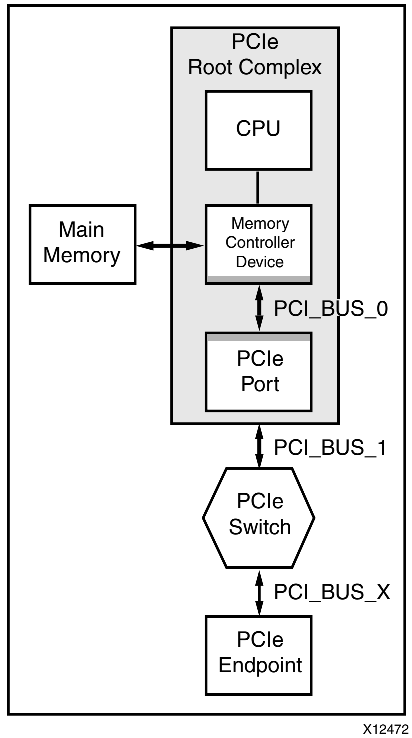

Figure 5-3: **System Overview**

Data is moved downstream when the CPU issues a store register to a MMIO address command. The Root Complex typically generates a Memory Write TLP with the appropriate MMIO location address, byte enables, and the register contents. The transaction terminates when the Endpoint receives the Memory Write TLP and updates the corresponding local register.

Data is moved upstream when the CPU issues a load register from a MMIO address command. The Root Complex typically generates a Memory Read TLP with the appropriate MMIO location address and byte enables. The Endpoint generates a Completion with Data TLP after it receives the Memory Read TLP. The Completion is steered to the Root Complex and payload is loaded into the target register, completing the transaction.

PIO Hardware

The PIO design implements an 8192 byte target space in FPGA block RAM, behind the Endpoint for PCIe. This 32-bit target space is accessible through single Dword I/O Read, I/O Write, Memory Read 64, Memory Write 64, Memory Read 32, and Memory Write 32 TLPs.

The PIO design generates a completion with one Dword of payload in response to a valid Memory Read 32 TLP, Memory Read 64 TLP, or I/O Read TLP request presented to it by the core. In addition, the PIO design returns a completion without data with successful status for I/O Write TLP request.

The PIO design can initiate:

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

The PIO design processes a Memory or I/O Write TLP with one Dword payload by updating the payload into the target address in the FPGA block RAM space.

Base Address Register Support

The PIO design supports four discrete target spaces, each consisting of a 2 KB block of memory represented by a separate Base Address Register (BAR). Using the default parameters, the Vivado IP catalog produces a core configured to work with the PIO design defined in this section, consisting of:

- One 64-bit addressable Memory Space BAR
- One 32-bit Addressable Memory Space BAR

You can change the default parameters used by the PIO design; however, in some cases you might need to change the user application depending on their system. See Changing IP Catalog Tool Default BAR Settings for information about changing the default Vivado Design Suite IP parameters and the effect on the PIO design.

Each of the four 2 KB address spaces represented by the BARs corresponds to one of four 2 KB address regions in the PIO design. Each 2 KB region is implemented using a 2 KB dual-port block RAM. As transactions are received by the core, the core decodes the address and determines which of the four regions is being targeted. The core presents the TLP to the PIO design and asserts the appropriate bits of (BAR ID[2:0]), Completer Request Descriptor[114:112], as defined in Table 5-1.

Table 5-1: **TLP Traffic Types**

Block RAM	TLP Transaction Type	Default BAR	BAR ID[2:0]
ep_io_mem	I/O TLP transactions	Disabled	Disabled
ep_mem32	32-bit address Memory TLP transactions	2	`000b`
ep_mem64	64-bit address Memory TLP transactions	0-1	`001b`
ep_mem_erom	32-bit address Memory TLP transactions destined for EROM	Expansion ROM	`110b`

Changing IP Catalog Tool Default BAR Settings

You can change the Vivado IP catalog parameters and continue to use the PIO design to create customized Verilog source to match the selected BAR settings. However, because the PIO design parameters are more limited than the core parameters, consider the following example design limitations when changing the default IP catalog parameters:

- The example design supports one I/O space BAR, one 32-bit Memory space (that cannot be the Expansion ROM space), and one 64-bit Memory space. If these limits are exceeded, only the first space of a given type is active—accesses to the other spaces do not result in completions.
- Each space is implemented with a 2 KB memory. If the corresponding BAR is configured to a wider aperture, accesses beyond the 2 KB limit wrap around and overlap the 2 KB memory space.
- The PIO design supports one I/O space BAR, which by default is disabled, but can be changed if desired.

Although there are limitations to the PIO design, Verilog source code is provided so users can tailor the example design to their specific needs.

TLP Data Flow

This section defines the data flow of a TLP successfully processed by the PIO design.

The PIO design successfully processes single Dword payload Memory Read and Write TLPs and I/O Read and Write TLPs. Memory Read or Memory Write TLPs of lengths larger than one Dword are not processed correctly by the PIO design; however, the core does accept these TLPs and passes them along to the PIO design. If the PIO design receives a TLP with a length of greater than one Dword, the TLP is received completely from the core and discarded. No corresponding completion is generated.

Memory and I/O Write TLP Processing

When the Endpoint for PCIe receives a Memory or I/O Write TLP, the TLP destination address and transaction type are compared with the values in the core BARs. If the TLP passes this comparison check, the core passes the TLP to the Receive AXI4-Stream interface of the PIO design. The PIO design handles Memory writes and I/O TLP writes in different ways: the PIO design responds to *I/O writes* by generating a Completion Without Data (cpl), a requirement of the PCI Express specification.

Along with the start of packet, end of packet, and ready handshaking signals, the Completer Requester AXI4-Stream interface also asserts the appropriate (BAR ID[2:0]), Completer Request Descriptor[114:112] signal to indicate to the PIO design the specific destination BAR that matched the incoming TLP. On reception, the PIO design RX State Machine processes the incoming Write TLP and extracts the TLPs data and relevant address fields so that it can pass this along to the PIO design internal block RAM write request controller.

Based on the specific BAR ID[2:0] signals asserted, the RX state machine indicates to the internal write controller the appropriate 2 KB block RAM to use prior to asserting the write enable request. For example, if an I/O Write Request is received by the core targeting BAR0, the core passes the TLP to the PIO design and sets BAR ID[2:0] to `000b`. The RX state machine extracts the lower address bits and the data field from the I/O Write TLP and instructs the internal Memory Write controller to begin a write to the block RAM.

In this example, the assertion of setting BAR ID[2:0] to `000b` instructed the PIO memory write controller to access `ep_mem0` (which by default represents 2 KB of I/O space). While the write is being carried out to the FPGA block RAM, the PIO design RX state machine deasserts `m_axis_cq_tready`, causing the Receive AXI4-Stream interface to stall receiving any further TLPs until the internal Memory Write controller completes the write to the block RAM. Deasserting `m_axis_cq_tready` in this way is not required for all designs using the core; the PIO design uses this method to simplify the control logic of the RX state machine.

Memory and I/O Read TLP Processing

When the Endpoint for PCIe receives a Memory or I/O Read TLP, the TLP destination address and transaction type are compared with the values programmed in the core BARs. If the TLP passes this comparison check, the core passes the TLP to the Receive AXI4-Stream interface of the PIO design.

Along with the start of packet, end of packet, and ready handshaking signals, the Completer Requester AXI4-Stream interface also asserts the appropriate BAR ID[2:0] signal to indicate to the PIO design the specific destination BAR that matched the incoming TLP. On reception, the PIO design state machine processes the incoming Read TLP and extracts the relevant TLP information and passes it along to the internal block RAM read request controller of the PIO design.

Based on the specific BAR ID[2:0] signal asserted, the RX state machine indicates to the internal read request controller the appropriate 2 KB block RAM to use before asserting the read enable request. For example, if a Memory Read 32 Request TLP is received by the core targeting the default Mem32 BAR2, the core passes the TLP to the PIO design and sets BAR ID[2:0] to `010b`. The RX state machine extracts the lower address bits from the Memory 32 Read TLP and instructs the internal Memory Read Request controller to start a read operation.

In this example, the setting BAR ID[2:0] to `010b` instructs the PIO memory read controller to access the Mem32 space, which by default represents 2 KB of memory space. A notable difference in handling of memory write and read TLPs is the requirement of the receiving device to return a Completion with Data TLP in the case of memory or I/O read request.

While the read is being processed, the PIO design RX state machine deasserts `m_axis_cq_tready`, causing the Receive AXI4-Stream interface to stall receiving any further TLPs until the internal Memory Read controller completes the read access from the block RAM and generates the completion. Deasserting `m_axis_cq_tready` in this way is not required for all designs using the core. The PIO design uses this method to simplify the control logic of the RX state machine.

PIO File Structure

Table 5-2 defines the PIO design file structure. Based on the specific core targeted, not all files delivered by the Vivado IP catalog are necessary, and some files might not be delivered. The major difference is that some of the Endpoint for PCIe solutions use a 32-bit user datapath, others use a 64-bit datapath, and the PIO design works with both. The width of the datapath depends on the specific core being targeted.

Table 5-2: **PIO Design File Structure**

File	Description
PIO.v	Top-level design wrapper
PIO_INTR_CTRL.v	PIO interrupt controller
PIO_EP.v	PIO application module
PIO_TO_CTRL.v	PIO turn-off controller module
PIO_RX_ENGINE.v	32-bit Receive engine
PIO_TX_ENGINE.v	32-bit Transmit engine
PIO_EP_MEM_ACCESS.v	Endpoint memory access module
PIO_EP_MEM.v	Endpoint memory

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

Table 5-3: **PIO Configuration**

Core	x1	x2	x4	x8
UltraScale FPGA Gen3 Integrated Block	PIO_64	PIO_64, PIO_128	PIO_64, PIO_128, PIO_256	PIO_64, PIO_128[(1)], PIO_256

Notes:

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

Figure 5-4 shows the various components of the PIO design, which is separated into four main parts: the TX Engine, RX Engine, Memory Access Controller, and Power Management Turn-Off Controller.

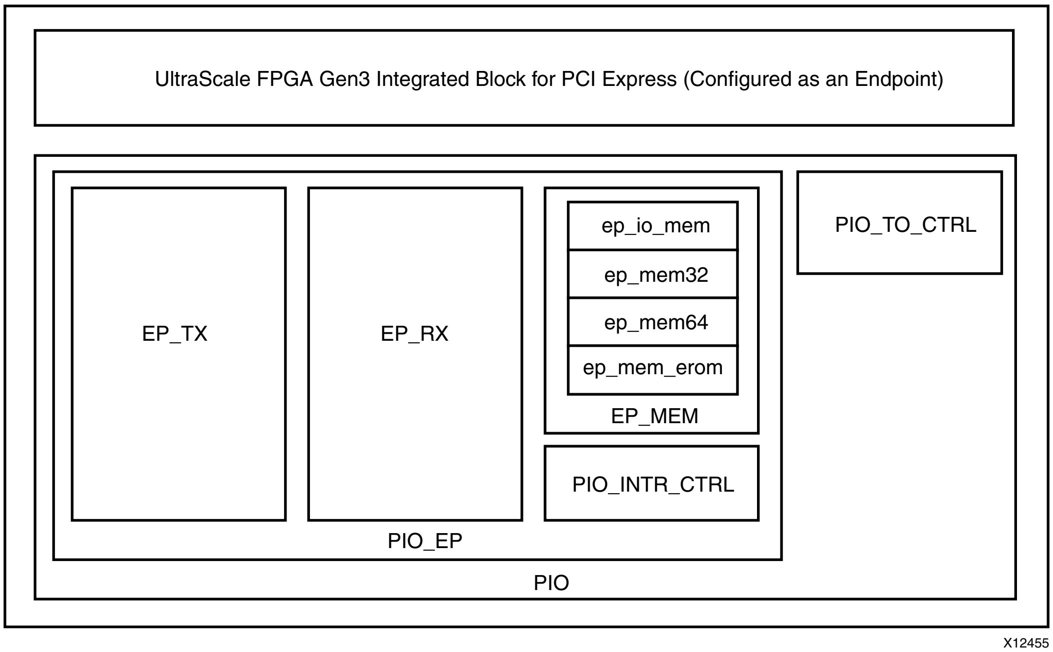

Figure 5-4: **PIO Design Components**

PIO Operation

PIO Read Transaction

Figure 5-5 depicts a Back-to-Back Memory Read request to the PIO design. The receive engine deasserts m_axis_rx_tready as soon as the first TLP is completely received. The next Read transaction is accepted only after compl_done_o is asserted by the transmit engine, indicating that Completion for the first request was successfully transmitted.

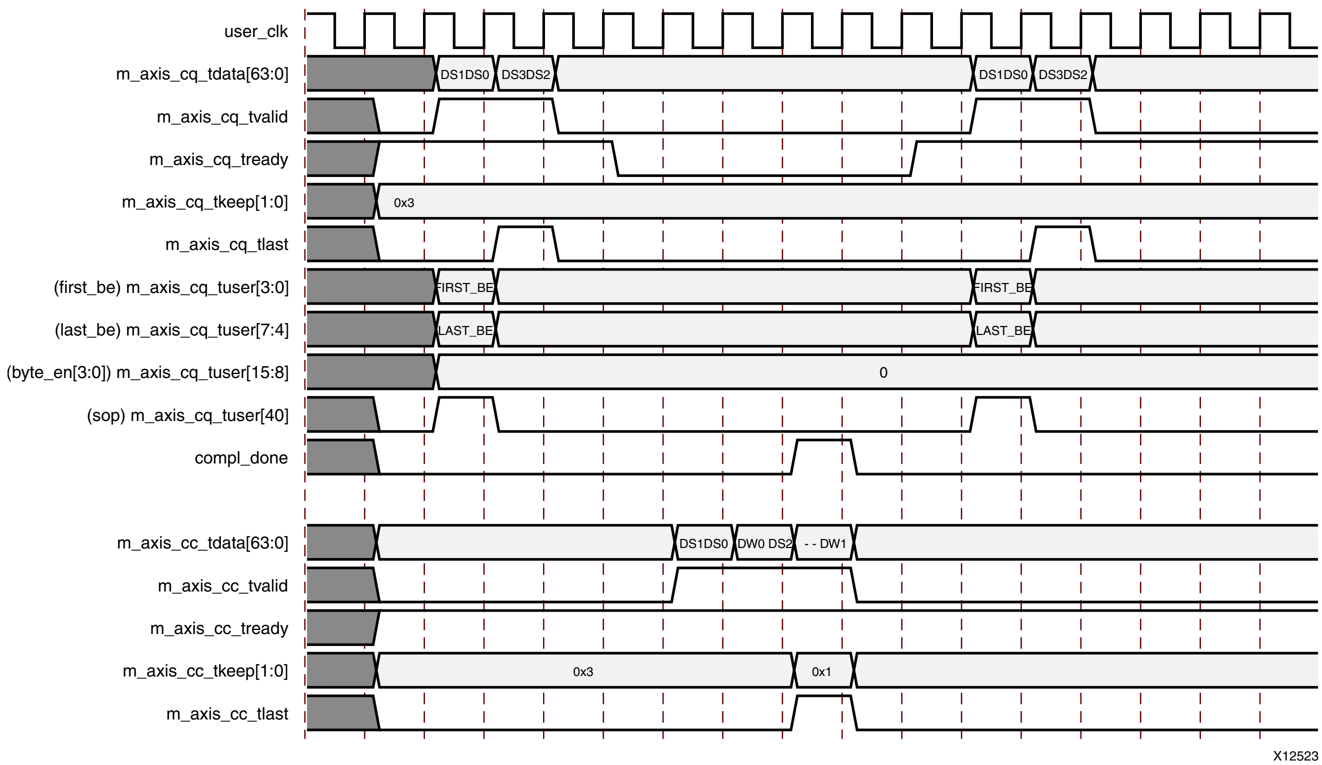

Figure 5-5: **Back-to-Back Read Transactions**

PIO Write Transaction

Figure 5-6 depicts a back-to-back Memory Write to the PIO design. The next Write transaction is accepted only after `wr_busy_o` is deasserted by the memory access unit, indicating that data associated with the first request was successfully written to the memory aperture.

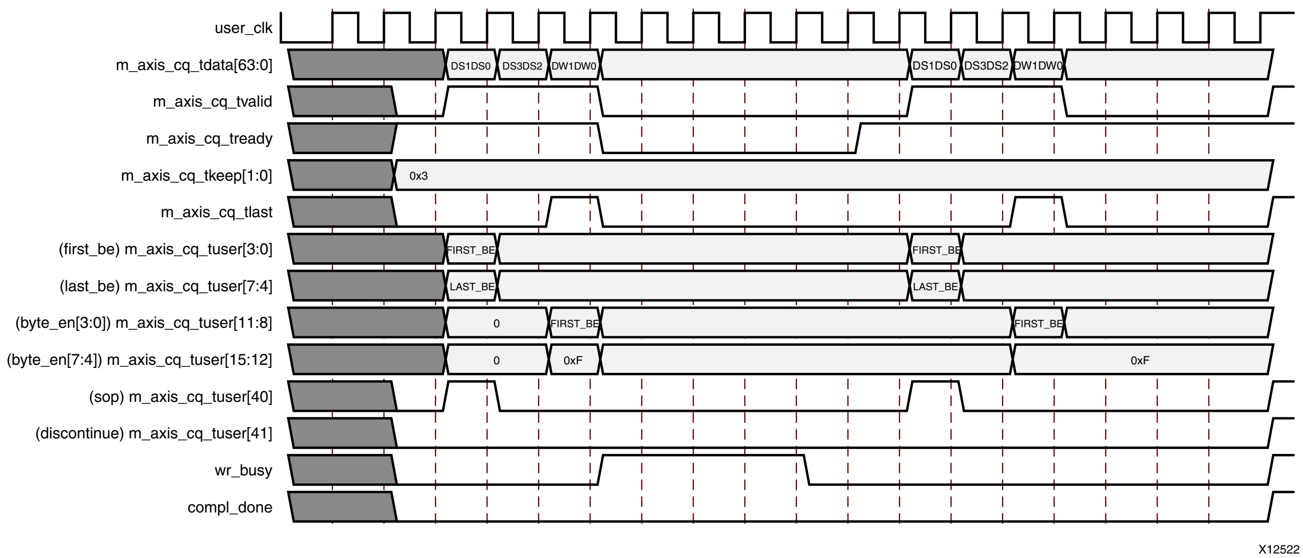

Figure 5-6: **Back-to-Back Write Transactions**

Device Utilization

Table 5-4 shows the PIO design FPGA resource utilization.

Table 5-4: **PIO Design FPGA Resources**

Resources	Utilization
LUTs	300
Flip-Flops	500
Block RAMs	4

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)
- ModelSim Questa® SIM

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

For any project (PCI Express core) generated out of the box, the simulations can be run as follows:

1. In the Sources Window, right-click the example project file (`.xci`), and select **Open IP Example Design**.

   The example project is created.

2. In the Flow Navigator (left-hand pane), under Simulation, right-click **Run Simulation** and select **Run Behavioral Simulation**.

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

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

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

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

To change the simulators:

1. In the Flow Navigator, under Simulation, select **Simulation Settings**.
2. In the Project Settings for Simulation dialog box, change the Target Simulator to **Questa Sim/ModelSim**.
3. When prompted, click **Yes** to change and then run the simulator.

Endpoint Configuration

The simulation environment provided with the Gen3 Integrated Block for PCIe core in Endpoint configuration performs simple memory access tests on the PIO example design. Transactions are generated by the Root Port Model and responded to by the PIO example design.

- PCI Express Transaction Layer Packets (TLPs) are generated by the test bench transmit User Application (`pci_exp_usrapp_tx`). As it transmits TLPs, it also generates a log file, `tx.dat`.
- PCI Express TLPs are received by the test bench receive User Application (`pci_exp_usrapp_rx`). As the User Application receives the TLPs, it generates a log file, `rx.dat`.

For more information about the test bench, see Root Port Model Test Bench for Endpoint, page 202.

Synthesizing and Implementing the Example Design

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

1. Go to the XCI file, right-click, and select **Open IP Example Design**.

   A new Vivado tool window opens with the project name "example_project" within the project directory.

2. In the Flow Navigator, click **Run Synthesis** and **Run Implementation**.

TIP: *Click **Run Implementation** first to run both synthesis and implementation.*
*Click **Generate Bitstream** to run synthesis, implementation, and then bitstream.*

Test Bench

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

Root Port Model Test Bench for Endpoint

The PCI Express Root Port Model is a robust test bench environment that provides a test program interface that can be used with the provided PIO design or with your design. The purpose of the Root Port Model is to provide a source mechanism for generating downstream PCI Express TLP traffic to stimulate the customer design, and a destination mechanism for receiving upstream PCI Express TLP traffic from the customer design in a simulation environment.

Source code for the Root Port Model is included to provide the model for a starting point for your test bench. All the significant work for initializing the core configuration space, creating TLP transactions, generating TLP logs, and providing an interface for creating and verifying tests are complete, allowing you to dedicate efforts to verifying the correct functionality of the design rather than spending time developing an Endpoint core test bench infrastructure.

The Root Port Model consists of:

- Test Programming Interface (TPI), which allows you to stimulate the Endpoint device for the PCI Express
- Example tests that illustrate how to use the test program TPI
- Verilog source code for all Root Port Model components, which allow you to customize the test bench

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

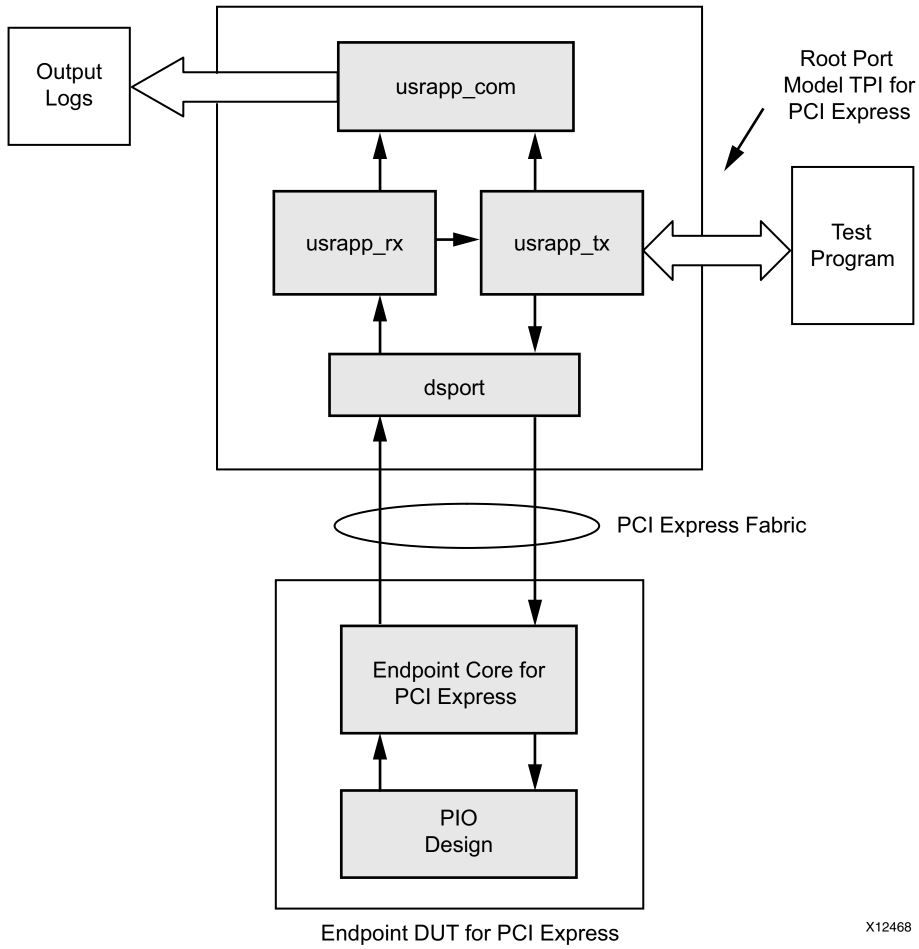

Figure 6-1: **Root Port Model and Top-Level Endpoint**

Architecture

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

- dsport (Root Port)
- usrapp_tx
- usrapp_rx
- usrapp_com (Verilog only)

The usrapp_tx and usrapp_rx blocks interface with the dsport block for transmission and reception of TLPs to/from the Endpoint Design Under Test (DUT). The Endpoint DUT consists of the Endpoint for PCIe and the PIO design (displayed) or customer design.

The usrapp_tx block sends TLPs to the dsport block for transmission across the PCI Express Link to the Endpoint DUT. In turn, the Endpoint DUT device transmits TLPs across the PCI Express Link to the dsport block, which are subsequently passed to the usrapp_rx block. The dsport and core are responsible for the data link layer and physical link layer processing when communicating across the PCI Express logic. Both usrapp_tx and usrapp_rx utilize the usrapp_com block for shared functions, for example, TLP processing and log file outputting. Transaction sequences or test programs are initiated by the usrapp_tx block to stimulate the Endpoint device fabric interface. TLP responses from the Endpoint device are received by the usrapp_rx block. Communication between the usrapp_tx and usrapp_rx blocks allow the usrapp_tx block to verify correct behavior and act accordingly when the usrapp_rx block has received TLPs from the Endpoint device.

Scaled Simulation Timeouts

The simulation model of the core uses scaled down times during link training to allow for the link to train in a reasonable amount of time during simulation. According to the *PCI Express Specification, rev. 3.0* [Ref 2], there are various timeouts associated with the link training and status state machine (LTSSM) states. The core scales these timeouts by a factor of 256 in simulation, except in the Recovery Speed_1 LTSSM state, where the timeouts are not scaled.

Test Selection

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

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

Test Name	Test in Verilog	Description
sample_smoke_test0	Verilog	Issues a PCI Type 0 Configuration Read TLP and waits for the completion TLP; then compares the value returned with the expected Device/Vendor ID value.
sample_smoke_test1	Verilog	Performs the same operation as sample_smoke_test0 but makes use of expectation tasks. This test uses two separate test program threads: one thread issues the PCI Type 0 Configuration Read TLP and the second thread issues the Completion with Data TLP expectation task. This test illustrates the form for a parallel test that uses expectation tasks. This test form allows for confirming reception of any TLPs from your design. Additionally, this method can be used to confirm reception of TLPs when ordering is unimportant.

Verilog Test Selection

The Verilog test model used for the Root Port Model lets you specify the name of the test to be run as a command line parameter to the simulator.

To change the test to be run, change the value provided to TESTNAME, which is defined in the test files `sample_tests1.v` and `pio_tests.v`. This mechanism is used for Mentor Graphics Questa® SIM. Vivado simulator uses the `-testplusarg` options to specify TESTNAME, for example:

```
demo_tb.exe -gui -view wave.wcfg -wdb wave_isim -tclbatch isim_cmd.tcl -testplusarg
TESTNAME=sample_smoke_test0.
```

Waveform Dumping

For information on simulator waveform dumping, see the *Vivado Design Suite User Guide: Logic Simulation (UG900)* [Ref 8].

Verilog Flow

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

Output Logging

When a test fails on the example or customer design, the test programmer debugs the offending test case. Typically, the test programmer inspects the wave file for the simulation and cross-reference this to the messages displayed on the standard output. Because this approach can be very time consuming, the Root Port Model offers an output logging mechanism to assist the tester with debugging failing test cases to speed the process.

The Root Port Model creates three output files (`tx.dat`, `rx.dat`, and `error.dat`) during each simulation run. The log files, `rx.dat` and `tx.dat`, each contain a detailed record of every TLP that was received and transmitted, respectively, by the Root Port Model.

TIP: *With an understanding of the expected TLP transmission during a specific test case, you can isolate the failure.*

The log file `error.dat` is used in conjunction with the expectation tasks. Test programs that use the expectation tasks generate a general error message to standard output. Detailed information about the specific comparison failures that have occurred due to the expectation error is located within `error.dat`.

Parallel Test Programs

There are two classes of tests are supported by the Root Port Model:

- Sequential tests. Tests that exist within one process and behave similarly to sequential programs. The test depicted in Test Program: pio_writeReadBack_test0, page 207 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).

Test Description

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

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

Test Program: pio_writeReadBack_test0

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

Expanding the Root Port Model

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

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

Root Port Model TPI Task List

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

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

Table 6-2: **Test Setup Tasks**

Name	Input(s)		Description
TSK_SYSTEM_INITIALIZATION	None		Waits for transaction interface reset and link-up between the Root Port Model and the Endpoint DUT. This task must be invoked prior to the Endpoint core initialization.
TSK_USR_DATA_SETUP_SEQ	None		Initializes global 4096 byte DATA_STORE array entries to sequential values from zero to 4095.
TSK_TX_CLK_EAT	clock count	31:30	Waits clock_count transaction interface clocks.
TSK_SIMULATION_TIMEOUT	timeout	31:0	Sets master simulation timeout value in units of transaction interface clocks. This task should be used to ensure that all DUT tests complete.

Table 6-3: **TLP Tasks**

Name	Input(s)		Description
TSK_TX_TYPE0_CONFIGURATION_READ	tag_ reg_addr_ first_dw_be_	7:0 11:0 3:0	Waits for transaction interface reset and link-up between the Root Port Model and the Endpoint DUT. This task must be invoked prior to Endpoint core initialization.
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 COMPLETE_ID_CFG as the completion ID.

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

Name	Input(s)		Description
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 COMPLETE_ID_CFG as the completion 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 COMPLETE_ID_CFG as the completion 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. CplD returned from the Endpoint DUT uses the contents of global COMPLETE_ID_CFG as the completion 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. CplD returned from the Endpoint DUT uses the contents of global COMPLETE_ID_CFG as the completion ID.
TSK_TX_MEMORY_WRITE_32	tag_ tc_ len_ addr_ last_dw_be_ first_dw_be_ ep_	7:0 2:0 10:0 31:0 3:0 3:0 –	Sends a PCI Express Memory Write TLP from Root Port Model to 32-bit memory address addr_ of Endpoint DUT. CplD returned from the Endpoint DUT uses the contents of global COMPLETE_ID_CFG as the completion 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. CplD returned from the Endpoint DUT uses the contents of global COMPLETE_ID_CFG as the completion ID. The global DATA_STORE byte array is used to pass write data to task.
TSK_TX_COMPLETION	tag_ tc_ len_ comp_status_	7:0 2:0 10:0 2:0	Sends a PCI Express Completion TLP from Root Port Model to the Endpoint DUT using global COMPLETE_ID_CFG as the completion ID.

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

Name	Input(s)		Description
TSK_TX_COMPLETION_DATA	tag_ tc_ len_ byte_count lower_addr comp_status ep_	7:0 2:0 10:0 11:0 6:0 2:0 –	Sends a PCI Express Completion with Data TLP from Root Port Model to the Endpoint DUT using global COMPLETE_ID_CFG as the completion ID. The global DATA_STORE byte array is used to pass completion data to task.
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. Completion returned from the Endpoint DUT uses the contents of global COMPLETE_ID_CFG as the completion 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. Completion returned from the Endpoint DUT uses the contents of global COMPLETE_ID_CFG as the completion ID.
TSK_TX_IO_READ	tag_ addr_ first_dw_be_	7:0 31:0 3:0	Sends a PCI Express I/O Read TLP from Root Port Model to I/O address addr_[31:2] of the Endpoint DUT. CplD returned from the Endpoint DUT uses the contents of global COMPLETE_ID_CFG as the completion 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. CplD returned from the Endpoint DUT uses the contents of global COMPLETE_ID_CFG as the completion 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. CplD returned from the Endpoint DUT use the contents of global COMPLETE_ID_CFG as the completion ID.

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

Name	Input(s)		Description
TSK_TX_BAR_WRITE	bar_index byte_offset tag_ tc_ data_	2:0 31:0 7:0 2:0 31:0	Sends a PCI Express one Dword Memory 32, Memory 64, or I/O Write TLP from the Root Port to the target address corresponding to offset byte_offset from BAR bar_index of the Endpoint DUT. This task sends the appropriate Write TLP based on how BAR bar_index has been configured during initialization. This task can only be called after TSK_BAR_INIT has successfully completed.
TSK_WAIT_FOR_READ_DATA	None		Waits for the next completion with data TLP that was sent by the Endpoint DUT. On successful completion, the first Dword of data from the CplD is stored in the global P_READ_DATA. This task should be called immediately following any of the read tasks in the TPI that request Completion with Data TLPs to avoid any race conditions. By default this task locally times out and terminate the simulation after 1000 transaction interface clocks. The global cpld_to_finish can be set to zero so that local timeout returns execution to the calling test and does not result in simulation timeout. For this case test programs should check the global cpld_to, which when set to one indicates that this task has timed out and that the contents of P_READ_DATA are invalid.

Table 6-4: **BAR Initialization Tasks**

Name	Input(s)	Description
TSK_BAR_INIT	None	Performs a standard sequence of Base Address Register initialization tasks to the Endpoint device using the PCI Express fabric. Performs a scan of the Endpoint PCI BAR range requirements, performs the necessary memory and I/O space mapping calculations, and finally programs the Endpoint so that it is ready to be accessed. On completion, the user test program can begin memory and I/O transactions to the device. This function displays to standard output a memory and I/O table that details how the Endpoint has been initialized. This task also initializes global variables within the Root Port Model that are available for test program usage. This task should only be called after TSK_SYSTEM_INITIALIZATION.
TSK_BAR_SCAN	None	Performs a sequence of PCI Type 0 Configuration Writes and Configuration Reads using the PCI Express logic to determine the memory and I/O requirements for the Endpoint. The task stores this information in the global array BAR_INIT_P_BAR_RANGE[]. This task should only be called after TSK_SYSTEM_INITIALIZATION.
TSK_BUILD_PCIE_MAP	None	Performs memory and I/O mapping algorithm and allocates Memory 32, Memory 64, and I/O space based on the Endpoint requirements. This task has been customized to work in conjunction with the limitations of the PIO design and should only be called after completion of TSK_BAR_SCAN.
TSK_DISPLAY_PCIE_MAP	None	Displays the memory mapping information of the Endpoint core PCI Base Address Registers. For each BAR, the BAR value, the BAR range, and BAR type is given. This task should only be called after completion of TSK_BUILD_PCIE_MAP.

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

Name	Input(s)		Description
TSK_TX_READBACK_CONFIG	None		Performs a sequence of PCI Type 0 Configuration Reads to the Endpoint device Base Address Registers, PCI Command Register, and PCIe Device Control Register using the PCI Express logic. This task should only be called after TSK_SYSTEM_INITIALIZATION.
TSK_MEM_TEST_DATA_BUS	bar_index	2:0	Tests whether the PIO design FPGA block RAM data bus interface is correctly connected by performing a 32-bit walking ones data test to the I/O or memory address pointed to by the input bar_index. For an exhaustive test, this task should be called four times, once for each block RAM used in the PIO design.

Table 6-5: **Example PIO Design Tasks *(Cont'd)***

Name	Input(s)		Description
TSK_MEM_TEST_ADDR_BUS	bar_index nBytes	2:0 31:0	Tests whether the PIO design FPGA block RAM address bus interface is accurately connected by performing a walking ones address test starting at the I/O or memory address pointed to by the input bar_index. For an exhaustive test, this task should be called four times, once for each block RAM used in the PIO design. Additionally, the nBytes input should specify the entire size of the individual block RAM.
TSK_MEM_TEST_DEVICE	bar_index nBytes	2:0 31:0	Tests the integrity of each bit of the PIO design FPGA block RAM by performing an increment/decrement test on all bits starting at the block RAM pointed to by the input bar_index with the range specified by input nBytes. For an exhaustive test, this task should be called four times, once for each block RAM used in the PIO design. Additionally, the nBytes input should specify the entire size of the individual block RAM.
TSK_RESET	Reset	0	Initiates PERSTn. Forces the `PERSTn` signal to assert the reset. Use `TSK_RESET` (1'b1) to assert the reset and `TSK_RESET` (1'b0) to release the reset signal.
TSK_MALFORMED	malformed _bits	7:0	Control bits for creating malformed TLPs: • 0001: Generate Malformed TLP for I/O Requests and Configuration Requests called immediately after this task • 0010: Generate Malformed Completion TLPs for Memory Read requests received at the Root Port

Table 6-6: **Expectation Tasks**

Name	Input(s)		Output	Description
TSK_EXPECT_CPLD	traffic_class td ep attr length completer_id completer_status bcm byte_count requester_id tag address_low	2:0 - - 1:0 10:0 15:0 2:0 - 11:0 15:0 7:0 6:0	Expect status	Waits for a Completion with Data TLP that matches traffic_class, td, ep, attr, length, and payload. Returns a 1 on successful completion; 0 otherwise.
TSK_EXPECT_CPL	traffic_class td ep attr completer_id completer_status bcm byte_count requester_id tag address_low	2:0 - - 1:0 15:0 2:0 - 11:0 15:0 7:0 6:0	Expect status	Waits for a Completion without Data TLP that matches traffic_class, td, ep, attr, and length. Returns a 1 on successful completion; 0 otherwise.
TSK_EXPECT_MEMRD	traffic_class td ep attr length requester_id tag last_dw_be first_dw_be address	2:0 - - 1:0 10:0 15:0 7:0 3:0 3:0 29:0	Expect status	Waits for a 32-bit Address Memory Read TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. This task can only be used in conjunction with Bus Master designs.
TSK_EXPECT_MEMRD64	traffic_class td ep attr length requester_id tag last_dw_be first_dw_be address	2:0 - - 1:0 10:0 15:0 7:0 3:0 3:0 61:0	Expect status	Waits for a 64-bit Address Memory Read TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. This task can only be used in conjunction with Bus Master designs.

Table 6-6: **Expectation Tasks *(Cont'd)***

Name	Input(s)		Output	Description
TSK_EXPECT_MEMWR	traffic_class td ep attr length requester_id tag last_dw_be first_dw_be address	2:0 - - 1:0 10:0 15:0 7:0 3:0 3:0 29:0	Expect status	Waits for a 32-bit Address Memory Write TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. This task can only be used in conjunction with Bus Master designs.
TSK_EXPECT_MEMWR64	traffic_class td ep attr length requester_id tag last_dw_be first_dw_be address	2:0 - - 1:0 10:0 15:0 7:0 3:0 3:0 61:0	Expect status	Waits for a 64-bit Address Memory Write TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. This task can only be used in conjunction with Bus Master designs.
TSK_EXPECT_IOWR	td ep requester_id tag first_dw_be address data	- - 15:0 7:0 3:0 31:0 31:0	Expect status	Waits for an I/O Write TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. This task can only be used in conjunction with Bus Master designs.

Endpoint Model Test Bench for Root Port

The Endpoint model test bench for the core in Root Port configuration is a simple example test bench that connects the Configurator example design and the PCI Express Endpoint model allowing the two to operate like two devices in a physical system. As the Configurator example design consists of logic that initializes itself and generates and consumes bus traffic, the example test bench only implements logic to monitor the operation of the system and terminate the simulation.

The Endpoint model test bench consists of:

- Verilog or VHDL source code for all Endpoint model components
- PIO slave design

Figure 6-1 illustrates the Endpoint model coupled with the Configurator example design.

Architecture

The Endpoint model consists of these blocks:

- PCI Express Endpoint (the core in Endpoint configuration) model.
- PIO slave design, consisting of:
  - PIO_RX_ENGINE
  - PIO_TX_ENGINE
  - PIO_EP_MEM
  - PIO_TO_CTRL

The PIO_RX_ENGINE and PIO_TX_ENGINE blocks interface with the ep block for reception and transmission of TLPs from/to the Root Port Design Under Test (DUT). The Root Port DUT consists of the core configured as a Root Port and the Configurator Example Design, which consists of a Configurator block and a PIO Master design, or customer design.

The PIO slave design is described in detail in Programmed Input/Output: Endpoint Example Design.

Simulating the Design

A simulation script file ,`simulate_mti.do`, is provided with the model to facilitate simulation with the Mentor Graphics Questa SIM simulator.

The example simulation script files are located in this directory:

```
<project_dir>/<component_name>/simulation/functional
```

Instructions for simulating the Configurator example design with the Endpoint model are provided in Simulation in Chapter 4.

Note: For Cadence IES users, the work construct must be manually inserted into the `cds.lib` file:

```
DEFINE WORK WORK.
```

Scaled Simulation Timeouts

The simulation model of the core uses scaled down times during link training to allow for the link to train in a reasonable amount of time during simulation. According to the *PCI Express Specification, rev. 3.0* [Ref 2], there are various timeouts associated with the link training and status state machine (LTSSM) states. The core scales these timeouts by a factor of 256 in simulation, except in the Recovery Speed_1 LTSSM state, where the timeouts are not scaled.

Waveform Dumping

For information on simulator waveform dumping, see the *Vivado Design Suite User Guide: Logic Simulation (UG900)* [Ref 8].

Output Logging

The test bench outputs messages, captured in the simulation log, indicating the time at which these occur:

- user_reset deasserted
- user_lnk_up asserted
- cfg_done asserted by the Configurator
- pio_test_finished asserted by the PIO Master
- Simulation Timeout (if pio_test_finished or pio_test_failed never asserted)

Appendix A

Migrating and Upgrading

This appendix contains information about upgrading to a more recent version of the IP core. For customers upgrading in the Vivado® Design Suite, important details (where applicable) about any port changes and other impact to user logic are included.

Migrating to the Vivado Design Suite

For information on migrating to the Vivado Design Suite, see *ISE to Vivado Design Suite Migration Methodology Guide* (UG911) [Ref 9].

Upgrading in the Vivado Design Suite

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

Parameter Changes

Figure A-1 shows the changes to parameters in the current version of the core.

Table A-1: **Parameter Changes**

#	User Parameter name	Display Name	New/ Changed/ Removed	Details	Default Value
1	en_pcie_drp	PCIe DRP Ports	New	Enables/disables the PCIe DRP interface. Visible only when advanced mode is selected on Basic page.	Unchecked (FALSE)
2	pipe_sim	Enable Pipe Simulation	New	Enables/disables external pipe interface to connect to external third-party BFM.	Unchecked (FALSE)

Table A-1: **Parameter Changes *(Cont'd)***

#	User Parameter name	Display Name	New/ Changed/ Removed	Details	Default Value
3	en_transceiver_status_ports	Additional Transceiver Control and Status Ports	New	Enables/disables transceiver core debug interface.	Unchecked (FALSE)
4	Shared_Logic	A radio button that indicates one of the following: • Include Shared Logic in core • Include Shared Logic in example design	New	Enables/disables shared logic in the core. Visible only when advanced mode is selected on the Basic page, and for Endpoint configuration only.	Default option is Shared logic in core
5	tx_fc_if	Transmit FC Interface	New	Enables/disables transmit FC interface signals. Visible only when advanced mode is selected on the Basic page; located on the Core Interface Parameters page.	Checked (TRUE)
6	cfg_fc_if	Config FC Interface	New	Enables/disables Config FC interface signals. Visible only when advanced mode is selected on the Basic page; located on the Core Interface Parameters page.	Checked (TRUE)
7	cfg_ext_if	Config Ext Interface	New	Enables/disables Config External interface signals. Visible only when advanced mode is selected on Basic page; located on the Core Interface Parameters page.	Checked (TRUE)
8	cfg_status_if	Config Status interface	New	Enables/disables Config status interface signals. Visible only when advanced mode is selected on Basic page; located on the Core Interface Parameters page.	Checked (TRUE)

Table A-1: **Parameter Changes *(Cont'd)***

#	User Parameter name	Display Name	New/ Changed/ Removed	Details	Default Value
9	per_func_status_if	Per Function Status Interface	New	Enables/disables Per Function Status interface signals.Visible only when advanced mode is selected on Basic page; located on the Core Interface Parameters page.	Checked (TRUE)
10	cfg_mgmt_if	Config Management Interface	New	Enables/disables Config Management interface signals.Visible only when advanced mode is selected on Basic page; located on the Core Interface Parameters page.	Checked (TRUE)
11	rcv_msg_if	Receive Message Interface	New	Enables/disables Receive Message interface signals. Visible only when advanced mode is selected on Basic page; located on the Core Interface Parameters page.	Checked (TRUE)
12	cfg_tx_msg_if	Config TX Message Interface	New	Enables/disables Config TX Message interface signals. Visible only when advanced mode is selected on Basic page; located on the Core Interface Parameters page.	Checked (TRUE)
13	cfg_ctl_if	Config Control Interface	New	Enables/disables Config Control Message interface signals. Visible only when advanced mode; located on the Core Interface Parameters page.	Checked (TRUE)

Table A-1: **Parameter Changes *(Cont'd)***

#	User Parameter name	Display Name	New/ Changed/ Removed	Details	Default Value
14	en_pl_ifc	PL Interface	New	Enables/disables PL Interface signals. Visible only when advanced mode is selected on Basic page; located on the Core Interface Parameters page.	
15	en_pcie_conf	Config Interface	New	Enables/disables Config interface signals. Visible only when advanced mode is selected on Basic page; located on the Core Interface Parameters page.	

Port Changes

The ports in Table A-2 are enabled when the option Shared Logic is set to Shared Logic in Core.

Table A-2: **Shared Logic in Core Ports**

Port	Direction	Width
int_qpll1lock_out	O	2 bits
int_qpll1outclk_out	O	2 bits
int_qpll1outrefclk_out	O	2

The ports in Table A-3 are enabled when the option Shared Logic is set to Shared Logic in Example Design.

Table A-3: **Shared Logic in Example Design Ports**

Port	Direction	Width
ext_qpll1lock_out	I	2 bits
ext_qpll1outclk_out	I	2 bits
ext_qpll1outrefclk_out	I	2 bits
ext_qpll1pd	O	2 bits
ext_qpll1rate	O	6 bits
ext_qpll1refclk	O	2 bits
ext_qpll1reset	O	2 bits

The ports in Table A-4 are enabled when the option Additional Transceiver Control and Status Ports is set to TRUE.

Table A-4: **Additional Transceiver Control and Status Ports**

Port	Direction	Width
gt_bufgtdiv	O	9 bits
gt_cplllock	O	8 bits
gt_dmonitorout	O	136 bits
gt_eyescandataerror	O	8 bits
gt_gtpowergood	O	8 bits
gt_loopback	I	24 bits
gt_pcieuserratedone	I	8 bits
gt_pcieuserratestart	O	8 bits
gt_phystatus	O	8 bits
phy_prst_n	O	1 bit
gt_qpll1lock	O	2 bits
gt_pcierateidle	O	8 bits
phy_rrst_n	O	1 bit
phy_rst_fsm	O	4 bits
phy_rst_idle	O	1 bit
gt_rxbufstatus	O	24 bits
gt_rxcdrlock	O	8 bits
gt_rxcommadet	O	8 bits
gt_rxdlysresetdone	O	8 bits
phy_rxeq_fsm	O	24 bits
gt_rxoutclk	O	8 bits
gt_rxphaligndone	O	8 bits
gt_rxpmaresetdone	O	8 bits
gt_rxprbscntreset	I	8 bits
gt_rxprbserr	O	8 bits
gt_rxprbssel	I	32 bits
gt_rxrecclkout	O	8 bits
gt_rxresetdone	O	8 bits
gt_rxstatus	O	24 bits
gt_rxsyncdone	O	8 bits
gt_rxvalid	O	8 bits
gt_txdlysresetdone	O	8 bits
gt_txelecidle	O	8 bits

Table A-4: **Additional Transceiver Control and Status Ports *(Cont'd)***

Port	Direction	Width
phy_txeq_ctrl	O	16 bits
phy_txeq_fsm	O	24 bits
phy_txeq_preset	O	32 bits
gt_txphaligndone	O	8 bits
gt_txphinitdone	O	8 bits
gt_txprbsforceerr	I	8 bits
gt_txprbssel	I	32 bits
gt_txresetdone	O	8 bits

The ports in Table A-5 are enabled when the **External PIPE Simulation** is set to TRUE.

Table A-5: **External Pipe Interface Ports**

Port	Direction	Width
common_commands_in	I	26 bits
common_commands_out	O	17 bits
pipe_rx_0_sigs	I	84 bits
pipe_rx_1_sigs	I	84 bits
pipe_rx_2_sigs	I	84 bits
pipe_rx_3_sigs	I	84 bits
pipe_rx_4_sigs	I	84 bits
pipe_rx_5_sigs	I	84 bits
pipe_rx_6_sigs	I	84 bits
pipe_rx_7_sigs	I	84 bits
pipe_tx_0_sigs	O	70 bits
pipe_tx_1_sigs	O	70 bits
pipe_tx_2_sigs	O	70 bits
pipe_tx_3_sigs	O	70 bits
pipe_tx_4_sigs	O	70 bits
pipe_tx_5_sigs	O	70 bits
pipe_tx_6_sigs	O	70 bits
pipe_tx_7_sigs	O	70 bits

Migrating from 7 Series Cores to UltraScale Architecture-Based Cores

This section provides guidance for users migrating from the 7 series Gen2 core to the UltraScale® architecture-based Gen3 core. The 7 Series Gen3 core interface is the same as that of the UltraScale architecture-based Gen3 core. Figure 1-1 shows the AXI4 Streaming TX and RX interfaces.

PCIe 3.1 AXI4 ST Enhanced Interface

Completer Request (CQ) Interface

Table A-6: **Signal mapping of AXI-4 ST Basic Receive Interface to AXI4-ST Enhanced CQ Interface**

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

m_axis_rx_tstrb (64-Bit Interface Only)

Table A-7 shows the CR Interface signals used to generate the `m_axis_rx_tstrb` signal bus.

Table A-7: **CR Interface Signals for m_axis_rx_tstrb**

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

m_axis_rx_tuser

Table A-8 shows the CR Interface signals used to generate the `m_axis_rx_tuser` signal bus.

Table A-8: **CR Interface Signals for m_axis_rx_tuser**

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

AXI4-Stream Requester Completion (RC) Interface

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

Table A-9: **AXI4-Stream RC Interface Signal Mapping**

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

Table A-9: **AXI4-Stream RC Interface Signal Mapping *(Cont'd)***

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

m_axis_rx_tstrb (64-Bit Interface Only)

Table A-10 shows the Requester Completion interface signals used to generate the `m_axis_rx_tstrb` signal bus.

Table A-10: **RC Interface Signals for m_axis_rx_tstrb**

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

m_axis_rx_tuser

Table A-11 shows the Requester Completion interface signals used to generate the `m_axis_rx_tuser` signal bus.

Table A-11: **RC Interface Signals for m_axis_rx_tuser**

AXI4-Stream Receive Interface Name	Mnemonic	AXI4-Stream Completer Request Interface Name	Mnemonic	Notes
m_axis_rx_tuser[0]	rx_ecrc_err	m_axis_rc_tuser[41]	Discontinue	Not exact equivalent
m_axis_rx_tuser[1]	rx_err_fwd	m_axis_rx_tdata[46]	Poisoned completion	Valid only when Descriptor is present on the data bus (is_sof0/ is_sof1=1).
m_axis_rx_tuser[9:2]	rx_bar_hit[7:0]	N/A (Refer to CQ interface)	N/A	N/A
m_axis_rx_tuser[14:10] (128-bit only)	rx_is_sof[4:0]	m_axis_rc_tuser[32] m_axis_rc_tuser[33] (only for 256-bit straddle)	is_sof_0 is_sof_1	• 256-bit RC interface provides straddling option. If enabled, the core can straddle two completion TLPs in the same beat. • is_sof_1 is used only when straddling is enabled for 256-bit interface.

Table A-11: **RC Interface Signals for m_axis_rx_tuser *(Cont'd)***

AXI4-Stream Receive Interface Name	Mnemonic	AXI4-Stream Completer Request Interface Name	Mnemonic	Notes
m_axis_rx_tuser[21:17] (128-bit only)	rx_is_eof [4:0]	m_axis_rc_tuser[37:34] m_axis_rx_tuser[41:38] (only for 256-bit straddle)	Is_eof_0[3:0] Is_eof_1[3:0]	is_eof_1 is used only when straddling is enabled for 256-bit interface.
No equivalent signal		m_axis_rc_tuser[74:43]	Parity	

AXI4-Stream (Enhanced) Completer Completion Interface

Table A-12: **Signal Mapping of AXI-4 Stream (Basic) Transmit Interface to AXI4-Stream (Enhanced) Completer Completion Interface**

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

s_axis_tx_tstrb

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

Table A-13 shows the mapping between `s_axis_cc_tkeep` from the Completer Completion interface and the `s_axis_tx_tstrb` signal bus from the AXI4-Stream (Basic) Transmit interface when `tlast` is not asserted.

Table A-13: **Mapping Between s_axis_cc_tkeep and s_axis_tx_tstrb**

Interface Width	s_axis_tx_tstrb	s_axis_cc_tkeep
64	0x0F	0x1
	0xFF	0x3

Table A-13: **Mapping Between s_axis_cc_tkeep and s_axis_tx_tstrb**

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

s_axis_tx_tuser

Table A-14 shows the mapping between `s_axis_cc_tuser` from the Completer Completion interface and the `s_axis_tx_tuser` signal bus from the AXI-Stream (Basic) Transmit interface.

Table A-14: **Mapping Between s_axis_cc_tkeep and s_axis_tx_tstrb**

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

AXI4-Stream Requester Request Interface

Table A-15: **AXI-4 Stream Requester Request Interface Signal Mapping**

AXI4-Stream (Basic) Transmit Interface Name	AXI4-Stream (Enhanced) Requester Request Interface Name	Differences
s_axis_tx_tlast	s_axis_rq_tlast	None
s_axis_tx_tdada (64/128)	s_axis_rq_tdata (64/128/256)	None
s_axis_tx_tvalid	s_axis_rq_tvalid	None
s_axis_tx_tready	s_axis_rq_tready	None
s_axis_tx_tstrb	s_axis_rq_tkeep	See s_axis_tx_tstrb
s_axis_tx_tuser	s_axis_rq_tuser	See s_axis_tx_tuser
tx_buf_av[5:0]	pcie_tfc_nph_av / pcie_tfc_npd_av/ pcie_rq_tag_av	See tx_buf_av
tx_terr_drop		
tx_cfg_req	NA	The feature is not available.
tx_cfg_gnt	NA	The feature is not available.

s_axis_tx_tstrb

Table A-16 shows the Requester Request interface signals used to generate the `s_axis_tx_tstrb` signal bus.

Table A-16: **Requester Request Interface Signals for m_axis_rx_tuser**

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

Table A-17 shows the mapping between `s_axis_cc_tkeep` from the Completer Completion interface and the `s_axis_tx_tstrb` signal bus from the AXI-Stream (Basic) Transmit interface when `tlast` is not asserted.

Table A-17: **Mapping Between s_axis_cc_tkeep and s_axis_tx_tstrb**

Interface Width	s_axis_tx_tstrb	s_axis_rq_tkeep
64	0x0F	0x1
	0xFF	0x3

Table A-17: **Mapping Between s_axis_cc_tkeep and s_axis_tx_tstrb**

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

s_axis_tx_tuser

Table A-18 shows the mapping between `s_axis_rq_tuser` from Requester Request interface and `s_axis_tx_tuser` signal bus from the AXI-Stream (Basic) Transmit interface.

Table A-18: **Mapping between s_axis_rq_tuser and s_axis_tx_tuser**

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

tx_buf_av

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

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

Other Interfaces

Table A-19 describes additional interfaces provided by the core.

Table A-19: **Additional Interfaces Provided by the Core**

Interfaces	Description	Notes
Transmit Flow Control	Used by the user application to request which flow control information the core provides. Based on the setting flow control input to the core, this interface provides the following to the user application: • Posted/Non-Posted Header Flow Control Credits • Posted/Non-Posted Data Flow Control Credits • Completion Header Flow Control Credits • Completion Data Flow Control Credits	Similar functionality
Configuration Management	Used to read and write to the Configuration Space Registers.	Similar functionality
Configuration Status	Provides information about how the core is configured, such as the negotiated link width and speed, the power state of the core, and configuration errors.	Similar functionality as Configuration Specific Register Ports
Configuration Received Message	Indicates the logic of a decodable message from the link, the parameters associated with the data, and the type of message received	Similar functionality as Received Message TLP Status Ports
Configuration Transmit Message	Used by the user application to transmit messages to the core. The user application supplies the transmit message type and data information to the core, which responds with the Done signal.	
Per Function Status	Provides status data requested by the user application through the selected function.	Similar functionality as Error Reporting Ports
Configuration Control	Allows information exchange between the user application and the core. The user application uses this interface to: • set the configuration space • indicate if a correctable or uncorrectable error has occurred • set the device serial number • set the Downstream Bus, Device, and Function Number • receive per-function configuration information. This interface also provides handshaking between the user application and the core when a Power State change or function level reset occurs.	Similar functionality as the Power Management Port
Configuration Interrupt Controller	Allows the user application to set Legacy PCIe interrupts, MSI interrupts, or MSI-X interrupts. The core provides the interrupt status on the configuration interrupt sent and fail signals.	Similar functionality as Interrupt Generation and Status Ports
Configuration Extended	Allows the core to transfer configuration information with the user application when externally implemented configuration registers are implemented.	Similar functionality as Received Configuration TLP Status Ports

Core Pinouts

This appendix provides a list of core pinouts for this IP core.

Kintex UltraScale Device Core Pinouts

Table B-1 provides a complete list of Kintex UltraScale device core pinouts.

Table B-1: **Kintex Ultrascale Core Pinouts**

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FBVA676 FBVA900	XCKU035 XCKU040	X0Y0	Lane0	X0Y7	X0Y7	X0Y7	X0Y7
			Lane1		X0Y6	X0Y6	X0Y6
			Lane2			X0Y5	X0Y5
			Lane3			X0Y4	X0Y4
			Lane4				X0Y3
			Lane5				X0Y2
			Lane6				X0Y1
			Lane7				X0Y0
		X0Y1	Lane0	X0Y15	X0Y15	X0Y15	X0Y15
			Lane1		X0Y14	X0Y14	X0Y14
			Lane2			X0Y13	X0Y13
			Lane3			X0Y12	X0Y12
			Lane4				X0Y11
			Lane5				X0Y10
			Lane6				X0Y9
			Lane7				X0Y8

Table B-1: **Kintex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FFVA1156	XCKU035 XCKU040	X0Y0	Lane0	X0Y7	X0Y7	X0Y7	X0Y7
			Lane1		X0Y6	X0Y6	X0Y6
			Lane2			X0Y5	X0Y5
			Lane3			X0Y4	X0Y4
			Lane4				X0Y3
			Lane5				X0Y2
			Lane6				X0Y1
			Lane7				X0Y0
		X0Y1	Lane0	X0Y15	X0Y15	X0Y15	X0Y15
			Lane1		X0Y14	X0Y14	X0Y14
			Lane2			X0Y13	X0Y13
			Lane3			X0Y12	X0Y12
			Lane4				X0Y11
			Lane5				X0Y10
			Lane6				X0Y9
			Lane7				X0Y8
	XCKU060 XCKU075	X0Y0	Lane0	X1Y7	X1Y7	X1Y7	X1Y7
			Lane1		X1Y6	X1Y6	X1Y6
			Lane2			X1Y5	X1Y5
			Lane3			X1Y4	X1Y4
			Lane4				X1Y3
			Lane5				X1Y2
			Lane6				X1Y1
			Lane7				X1Y0
		X0Y1	Lane0	X1Y15	X1Y15	X1Y15	X1Y15
			Lane1		X1Y14	X1Y14	X1Y14
			Lane2			X1Y13	X1Y13
			Lane3			X1Y12	X1Y12
			Lane4				X1Y11
			Lane5				X1Y10
			Lane6				X1Y9
			Lane7				X1Y8

Table B-1: **Kintex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FFVA1156 (continued)	XCKU040	X0Y2	Lane0	X0Y19	X0Y19	X0Y19	Not supported
			Lane1		X0Y18	X0Y18	
			Lane2			X0Y17	
			Lane3			X0Y16	
	XCKU060 XCKU075		Lane0	X1Y19	X1Y19	X1Y19	Not supported
			Lane1		X1Y18	X1Y18	
			Lane2			X1Y17	
			Lane3			X1Y16	
FFVA1517	XCKU060 XCKU075	X0Y0	Lane0	X1Y7	X1Y7	X1Y7	X1Y7
			Lane1		X1Y6	X1Y6	X1Y6
			Lane2			X1Y5	X1Y5
			Lane3			X1Y4	X1Y4
			Lane4				X1Y3
			Lane5				X1Y2
			Lane6				X1Y1
			Lane7				X1Y0
		X0Y1	Lane0	X1Y15	X1Y15	X1Y15	X1Y15
			Lane1		X1Y14	X1Y14	X1Y14
			Lane2			X1Y13	X1Y13
			Lane3			X1Y12	X1Y12
			Lane4				X1Y11
			Lane5				X1Y10
			Lane6				X1Y9
			Lane7				X1Y8

Table B-1: **Kintex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FFVA1517 (continued)	XCKU060	X0Y2	Lane0	X1Y19	X1Y19	X1Y19	Not supported
			Lane1		X1Y18	X1Y18	
			Lane2			X1Y17	
			Lane3			X1Y16	
	XCKU075	X0Y2 (x8) mode	Lane0	X1Y23	X1Y23	X1Y23	X1Y23
			Lane1		X1Y22	X1Y22	X1Y22
			Lane2			X1Y21	X1Y21
			Lane3			X1Y20	X1Y20
			Lane4				X1Y19
			Lane5				X1Y18
			Lane6				X1Y17
			Lane7				X1Y16
		X0Y2 (x4) mode	Lane0	X1Y19	X1Y19	X1Y19	Not Applicable
			Lane1		X1Y18	X1Y18	
			Lane2			X1Y17	
			Lane3			X1Y16	
		X0Y3	Lane0	X1Y35	X1Y35	X1Y35	X1Y35
			Lane1		X1Y34	X1Y34	X1Y34
			Lane2			X1Y33	X1Y33
			Lane3			X1Y32	X1Y32
			Lane4				X1Y31
			Lane5				X1Y30
			Lane6				X1Y29
			Lane7				X1Y28

Table B-1: **Kintex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FFVA1760	XCKU075	X0Y0	Lane0	X1Y7	X1Y7	X1Y7	X1Y7
			Lane1		X1Y6	X1Y6	X1Y6
			Lane2			X1Y5	X1Y5
			Lane3			X1Y4	X1Y4
			Lane4				X1Y3
			Lane5				X1Y2
			Lane6				X1Y1
			Lane7				X1Y0
		X0Y1	Lane0	X1Y15	X1Y15	X1Y15	X1Y15
			Lane1		X1Y14	X1Y14	X1Y14
			Lane2			X1Y13	X1Y13
			Lane3			X1Y12	X1Y12
			Lane4				X1Y11
			Lane5				X1Y10
			Lane6				X1Y9
			Lane7				X1Y8
		X0Y2	Lane0	X1Y23	X1Y23	X1Y23	X1Y23
			Lane1		X1Y22	X1Y22	X1Y22
			Lane2			X1Y21	X1Y21
			Lane3			X1Y20	X1Y20
			Lane4				X1Y19
			Lane5				X1Y18
			Lane6				X1Y17
			Lane7				X1Y16
		X0Y3	Lane0	X1Y31	X1Y31	X1Y31	X1Y31
			Lane1		X1Y30	X1Y30	X1Y30
			Lane2			X1Y29	X1Y29
			Lane3			X1Y28	X1Y28
			Lane4				X1Y27
			Lane5				X1Y26
			Lane6				X1Y25
			Lane7				X1Y24

Table B-1: **Kintex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVA1517	XCKU100 XCKU115	X0Y0	Lane 0	X1Y7	X1Y7	X1Y7	X1Y7
			Lane 1		X1Y6	X1Y6	X1Y6
			Lane 2			X1Y5	X1Y5
			Lane 3			X1Y4	X1Y4
			Lane 4				X1Y3
			Lane 5				X1Y2
			Lane 6				X1Y1
			Lane 7				X1Y0
		X0Y1	Lane 0	X1Y15	X1Y15	X1Y15	X1Y15
			Lane 1		X1Y14	X1Y14	X1Y14
			Lane 2			X1Y13	X1Y13
			Lane 3			X1Y12	X1Y12
			Lane 4				X1Y11
			Lane 5				X1Y10
			Lane 6				X1Y9
			Lane 7				X1Y8
		X0Y2	Lane 0	X1Y19	X1Y19	X1Y19	Not supported
			Lane 1		X1Y18	X1Y18	
			Lane 2			X1Y17	
			Lane 3			X1Y16	

Table B-1: **Kintex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVA1517 (continued)	XCKU100 XCKU115 (continued)	X0Y3	Lane 0	X1Y27	X1Y27	X1Y27	X1Y27
			Lane 1		X1Y26	X1Y26	X1Y26
			Lane 2			X1Y25	X1Y25
			Lane 3			X1Y24	X1Y24
			Lane 4				X1Y23
			Lane 5				X1Y22
			Lane 6				X1Y21
			Lane 7				X1Y20
		X0Y4	Lane 0	X1Y35	X1Y35	X1Y35	X1Y35
			Lane 1		X1Y34	X1Y34	X1Y34
			Lane 2			X1Y33	X1Y33
			Lane 3			X1Y32	X1Y32
			Lane 4				X1Y31
			Lane 5				X1Y30
			Lane 6				X1Y29
			Lane 7				X1Y28
FLVB1517	XCKU100 XCKU115	X0Y0	Lane 0	X1Y7	X1Y7	X1Y7	X1Y7
			Lane 1		X1Y6	X1Y6	X1Y6
			Lane 2			X1Y5	X1Y5
			Lane 3			X1Y4	X1Y4
			Lane 4				X1Y3
			Lane 5				X1Y2
			Lane 6				X1Y1
			Lane 7				X1Y0
		X0Y1	Lane 0	X1Y15	X1Y15	X1Y15	X1Y15
			Lane 1		X1Y14	X1Y14	X1Y14
			Lane 2			X1Y13	X1Y13
			Lane 3			X1Y12	X1Y12
			Lane 4				X1Y11
			Lane 5				X1Y10
			Lane 6				X1Y9
			Lane 7				X1Y8

Table B-1: **Kintex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVB1517 (continued)	XCKU100 XCKU115 (continued)	X0Y2	Lane 0	X1Y19	X1Y19	X1Y19	Not supported
			Lane 1		X1Y18	X1Y18	
			Lane 2			X1Y17	
			Lane 3			X1Y16	
		X0Y3	Lane 0	X1Y27	X1Y27	X1Y27	X1Y27
			Lane 1		X1Y26	X1Y26	X1Y26
			Lane 2			X1Y25	X1Y25
			Lane 3			X1Y24	X1Y24
			Lane 4				X1Y23
			Lane 5				X1Y22
			Lane 6				X1Y21
			Lane 7				X1Y20
		X0Y4	Lane 0	X1Y35	X1Y35	X1Y35	X1Y35
			Lane 1		X1Y34	X1Y34	X1Y34
			Lane 2			X1Y33	X1Y33
			Lane 3			X1Y32	X1Y32
			Lane 4				X1Y31
			Lane 5				X1Y30
			Lane 6				X1Y29
			Lane 7				X1Y28
		X0Y5	Lane 0	X1Y39	X1Y39	X1Y39	Not supported
			Lane 1		X1Y38	X1Y38	
			Lane 2			X1Y37	
			Lane 3			X1Y36	
FLVA1760	XCKU100 XCKU115	X0Y0	Lane 0	X1Y7	X1Y7	X1Y7	X1Y7
			Lane 1		X1Y6	X1Y6	X1Y6
			Lane 2			X1Y5	X1Y5
			Lane 3			X1Y4	X1Y4
			Lane 4				X1Y3
			Lane 5				X1Y2
			Lane 6				X1Y1
			Lane 7				X1Y0

Table B-1: **Kintex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVA1760 (continued)	XCKU100 XCKU115 (continued)	X0Y1	Lane 0	X1Y15	X1Y15	X1Y15	X1Y15
			Lane 1		X1Y14	X1Y14	X1Y14
			Lane 2			X1Y13	X1Y13
			Lane 3			X1Y12	X1Y12
			Lane 4				X1Y11
			Lane 5				X1Y10
			Lane 6				X1Y9
			Lane 7				X1Y8
		X0Y2	Lane 0	X1Y19	X1Y19	X1Y19	Not supported
			Lane 1		X1Y18	X1Y18	
			Lane 2			X1Y17	
			Lane 3			X1Y16	
		X0Y3	Lane 0	X1Y31	X1Y31	X1Y31	X1Y31
			Lane 1		X1Y30	X1Y30	X1Y30
			Lane 2			X1Y29	X1Y29
			Lane 3			X1Y28	X1Y28
			Lane 4				X1Y27
			Lane 5				X1Y26
			Lane 6				X1Y25
			Lane 7				X1Y24
		X0Y4	Lane 0	X1Y35	X1Y35	X1Y35	Not supported
			Lane 1		X1Y34	X1Y34	
			Lane 2			X1Y33	
			Lane 3			X1Y32	
		X0Y5	Lane 0	X1Y39	X1Y39	X1Y39	Not supported
			Lane 1		X1Y38	X1Y38	
			Lane 2			X1Y37	
			Lane 3			X1Y36	

Table B-1: **Kintex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVD1924	XCKU100 XCKU115	X0Y0	Lane 0	X1Y7	X1Y7	X1Y7	X1Y7
			Lane 1		X1Y6	X1Y6	X1Y6
			Lane 2			X1Y5	X1Y5
			Lane 3			X1Y4	X1Y4
			Lane 4				X1Y3
			Lane 5				X1Y2
			Lane 6				X1Y1
			Lane 7				X1Y0
		X0Y1	Lane 0	X1Y15	X1Y15	X1Y15	X1Y15
			Lane 1		X1Y14	X1Y14	X1Y14
			Lane 2			X1Y13	X1Y13
			Lane 3			X1Y12	X1Y12
			Lane 4				X1Y11
			Lane 5				X1Y10
			Lane 6				X1Y9
			Lane 7				X1Y8
		X0Y3	Lane 0	X1Y31	X1Y31	X1Y31	Not supported
			Lane 1		X1Y30	X1Y30	
			Lane 2			X1Y29	
			Lane 3			X1Y28	
		X0Y4	Lane 0	X1Y35	X1Y35	X1Y35	Not supported
			Lane 1		X1Y34	X1Y34	
			Lane 2			X1Y33	
			Lane 3			X1Y32	
		X0Y5	Lane 0	X1Y39	X1Y39	X1Y39	Not supported
			Lane 1		X1Y38	X1Y38	
			Lane 2			X1Y37	
			Lane 3			X1Y36	

Table B-1: **Kintex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVF1924	XCKU100 XCKU115	X0Y0	Lane 0	X1Y7	X1Y7	X1Y7	X1Y7
			Lane 1		X1Y6	X1Y6	X1Y6
			Lane 2			X1Y5	X1Y5
			Lane 3			X1Y4	X1Y4
			Lane 4				X1Y3
			Lane 5				X1Y2
			Lane 6				X1Y1
			Lane 7				X1Y0
		X0Y1	Lane 0	X1Y15	X1Y15	X1Y15	X1Y15
			Lane 1		X1Y14	X1Y14	X1Y14
			Lane 2			X1Y13	X1Y13
			Lane 3			X1Y12	X1Y12
			Lane 4				X1Y11
			Lane 5				X1Y10
			Lane 6				X1Y9
			Lane 7				X1Y8
		X0Y2	Lane 0	X1Y19	X1Y19	X1Y19	Not supported
			Lane 1		X1Y18	X1Y18	
			Lane 2			X1Y17	
			Lane 3			X1Y16	
		X0Y3	Lane 0	X1Y27	X1Y27	X1Y27	X1Y27
			Lane 1		X1Y26	X1Y26	X1Y26
			Lane 2			X1Y25	X1Y25
			Lane 3			X1Y24	X1Y24
			Lane 4				X1Y23
			Lane 5				X1Y22
			Lane 6				X1Y21
			Lane 7				X1Y20

Table B-1: **Kintex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVF1924 (continued)	XCKU100 XCKU115 (continued)	X0Y4	Lane 0	X1Y35	X1Y35	X1Y35	X1Y35
			Lane 1		X1Y34	X1Y34	X1Y34
			Lane 2			X1Y33	X1Y33
			Lane 3			X1Y32	X1Y32
			Lane 4				X1Y31
			Lane 5				X1Y30
			Lane 6				X1Y29
			Lane 7				X1Y28
		X0Y5	Lane 0	X1Y39	X1Y39	X1Y39	Not supported
			Lane 1		X1Y38	X1Y38	
			Lane 2			X1Y37	
			Lane 3			X1Y36	

Virtex UltraScale Device Core Pinouts

Table B-1 provides a complete list of Kintex UltraScale device core pinouts.

Table B-2: **Virtex Ultrascale Core Pinouts**

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FFVC1517	XCVU065 XCVU080 XCVU095	X0Y0	Lane0	X0Y7	X0Y7	X0Y7	X0Y7
			Lane1		X0Y6	X0Y6	X0Y6
			Lane2			X0Y5	X0Y5
			Lane3			X0Y4	X0Y4
			Lane4				X0Y3
			Lane5				X0Y2
			Lane6				X0Y1
			Lane7				X0Y0
		X0Y1	Lane0	X0Y15	X0Y15	X0Y15	X0Y15
			Lane1		X0Y14	X0Y14	X0Y14
			Lane2			X0Y13	X0Y13
			Lane3			X0Y12	X0Y12
			Lane4				X0Y11
			Lane5				X0Y10
			Lane6				X0Y9
			Lane7				X0Y8
	XCVU080 XCVU095	X0Y2	Lane0	X0Y23	X0Y23	X0Y23	X0Y23
			Lane1		X0Y22	X0Y22	X0Y22
			Lane2			X0Y21	X0Y21
			Lane3			X0Y20	X0Y20
			Lane4				X0Y19
			Lane5				X0Y18
			Lane6				X0Y17
			Lane7				X0Y16

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FFVA1760 FFVB1517 FFVE1924	XCVU080 XCVU095	X0Y0	Lane0	X0Y7	X0Y7	X0Y7	X0Y7
			Lane1		X0Y6	X0Y6	X0Y6
			Lane2			X0Y5	X0Y5
			Lane3			X0Y4	X0Y4
			Lane4				X0Y3
			Lane5				X0Y2
			Lane6				X0Y1
			Lane7				X0Y0
		X0Y1	Lane0	X0Y15	X0Y15	X0Y15	X0Y15
			Lane1		X0Y14	X0Y14	X0Y14
			Lane2			X0Y13	X0Y13
			Lane3			X0Y12	X0Y12
			Lane4				X0Y11
			Lane5				X0Y10
			Lane6				X0Y9
			Lane7				X0Y8
		X0Y2	Lane0	X0Y23	X0Y23	X0Y23	X0Y23
			Lane1		X0Y22	X0Y22	X0Y22
			Lane2			X0Y21	X0Y21
			Lane3			X0Y20	X0Y20
			Lane4				X0Y19
			Lane5				X0Y18
			Lane6				X0Y17
			Lane7				X0Y16
		X0Y3	Lane0	X0Y31	X0Y31	X0Y31	X0Y31
			Lane1		X0Y30	X0Y30	X0Y30
			Lane2			X0Y29	X0Y29
			Lane3			X0Y28	X0Y28
			Lane4				X0Y27
			Lane5				X0Y26
			Lane6				X0Y25
			Lane7				X0Y24

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FFVD1924	XCVU080 XCVU095	X0Y0	Lane 0	X0Y7	X0Y7	X0Y7	X0Y7
			Lane 1		X0Y6	X0Y6	X0Y6
			Lane 2			X0Y5	X0Y5
			Lane 3			X0Y4	X0Y4
			Lane 4				X0Y3
			Lane 5				X0Y2
			Lane 6				X0Y1
			Lane 7				X0Y0
		X0Y1	Lane 0	X0Y15	X0Y15	X0Y15	X0Y15
			Lane 1		X0Y14	X0Y14	X0Y14
			Lane 2			X0Y13	X0Y13
			Lane 3			X0Y12	X0Y12
			Lane 4				X0Y11
			Lane 5				X0Y10
			Lane 6				X0Y9
			Lane 7				X0Y8
		X0Y2	Lane 0	X0Y23	X0Y23	X0Y23	X0Y23
			Lane 1		X0Y22	X0Y22	X0Y22
			Lane 2			X0Y21	X0Y21
			Lane 3			X0Y20	X0Y20
			Lane 4				X0Y19
			Lane 5				X0Y18
			Lane 6				X0Y17
			Lane 7				X0Y16
		X0Y3	Lane 0	X0Y27	X0Y27	X0Y27	Not supported
			Lane 1		X0Y26	X0Y26	
			Lane 2			X0Y25	
			Lane 3			X0Y24	

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FFVJ1924	XCVU095	X0Y0	Lane0	X0Y7	X0Y7	X0Y7	X0Y7
			Lane1		X0Y6	X0Y6	X0Y6
			Lane2			X0Y5	X0Y5
			Lane3			X0Y4	X0Y4
			Lane4				X0Y3
			Lane5				X0Y2
			Lane6				X0Y1
			Lane7				X0Y0
		X0Y1	Lane0	X0Y15	X0Y15	X0Y15	X0Y15
			Lane1		X0Y14	X0Y14	X0Y14
			Lane2			X0Y13	X0Y13
			Lane3			X0Y12	X0Y12
			Lane4				X0Y11
			Lane5				X0Y10
			Lane6				X0Y9
			Lane7				X0Y8
		X0Y2	Lane0	X0Y23	X0Y23	X0Y23	X0Y23
			Lane1		X0Y22	X0Y22	X0Y22
			Lane2			X0Y21	X0Y21
			Lane3			X0Y20	X0Y20
			Lane4				X0Y19
			Lane5				X0Y18
			Lane6				X0Y17
			Lane7				X0Y16
		X0Y3	Lane0	X0Y31	X0Y31	X0Y31	X0Y31
			Lane1		X0Y30	X0Y30	X0Y30
			Lane2			X0Y29	X0Y29
			Lane3			X0Y28	X0Y28
			Lane4				X0Y27
			Lane5				X0Y26
			Lane6				X0Y25
			Lane7				X0Y24

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVB1517	XCVU125	X0Y0	Lane0	X0Y7	X0Y7	X0Y7	X0Y7
			Lane1		X0Y6	X0Y6	X0Y6
			Lane2			X0Y5	X0Y5
			Lane3			X0Y4	X0Y4
			Lane4				X0Y3
			Lane5				X0Y2
			Lane6				X0Y1
			Lane7				X0Y0
		X0Y1	Lane0	X0Y15	X0Y15	X0Y15	X0Y15
			Lane1		X0Y14	X0Y14	X0Y14
			Lane2			X0Y13	X0Y13
			Lane3			X0Y12	X0Y12
			Lane4				X0Y11
			Lane5				X0Y10
			Lane6				X0Y9
			Lane7				X0Y8
		X0Y2	Lane0	X0Y27	X0Y27	X0Y27	X0Y27
			Lane1		X0Y26	X0Y26	X0Y26
			Lane2			X0Y25	X0Y25
			Lane3			X0Y24	X0Y24
			Lane4				X0Y23
			Lane5				X0Y22
			Lane6				X0Y21
			Lane7				X0Y20
		X0Y3	Lane0	X0Y39	X0Y39	X0Y39	X0Y39
			Lane1		X0Y38	X0Y38	X0Y38
			Lane2			X0Y37	X0Y37
			Lane3			X0Y36	X0Y36
			Lane4				X0Y35
			Lane5				X0Y34
			Lane6				X0Y33
			Lane7				X0Y32

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVA1760	XCVU125	X0Y0	Lane0	X0Y7	X0Y7	X0Y7	X0Y7
			Lane1		X0Y6	X0Y6	X0Y6
			Lane2			X0Y5	X0Y5
			Lane3			X0Y4	X0Y4
			Lane4				X0Y3
			Lane5				X0Y2
			Lane6				X0Y1
			Lane7				X0Y0
		X0Y1	Lane0	X0Y15	X0Y15	X0Y15	X0Y15
			Lane1		X0Y14	X0Y14	X0Y14
			Lane2			X0Y13	X0Y13
			Lane3			X0Y12	X0Y12
			Lane4				X0Y11
			Lane5				X0Y10
			Lane6				X0Y9
			Lane7				X0Y8
		X0Y2	Lane0	X0Y31	X0Y31	X0Y31	X0Y31
			Lane1		X0Y30	X0Y30	X0Y30
			Lane2			X0Y29	X0Y29
			Lane3			X0Y28	X0Y28
			Lane4				X0Y27
			Lane5				X0Y26
			Lane6				X0Y25
			Lane7				X0Y24
		X0Y3	Lane0	X0Y39	X0Y39	X0Y39	X0Y39
			Lane1		X0Y38	X0Y38	X0Y38
			Lane2			X0Y37	X0Y37
			Lane3			X0Y36	X0Y36
			Lane4				X0Y35
			Lane5				X0Y34
			Lane6				X0Y33
			Lane7				X0Y32

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVD1924	XCVU125	X0Y0	Lane0	X0Y7	X0Y7	X0Y7	X0Y7
			Lane1		X0Y6	X0Y6	X0Y6
			Lane2			X0Y5	X0Y5
			Lane3			X0Y4	X0Y4
			Lane4				X0Y3
			Lane5				X0Y2
			Lane6				X0Y1
			Lane7				X0Y0
		X0Y1	Lane0	X0Y15	X0Y15	X0Y15	X0Y15
			Lane1		X0Y14	X0Y14	X0Y14
			Lane2			X0Y13	X0Y13
			Lane3			X0Y12	X0Y12
			Lane4				X0Y11
			Lane5				X0Y10
			Lane6				X0Y9
			Lane7				X0Y8
		X0Y2	Lane0	X0Y35	X0Y35	X0Y35	X0Y35
			Lane1		X0Y34	X0Y34	X0Y34
			Lane2			X0Y33	X0Y33
			Lane3			X0Y32	X0Y32
			Lane4				X0Y31
			Lane5				X0Y30
			Lane6				X0Y29
			Lane7				X0Y28
		X0Y3	Lane0	X0Y39	X0Y39	X0Y39	Not supported
			Lane1		X0Y38	X0Y38	
			Lane2			X0Y37	
			Lane3			X0Y36	

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVD1924 (continued)	XCVU160	X0Y1	Lane0	X0Y23	X0Y23	X0Y23	X0Y23
			Lane1		X0Y22	X0Y22	X0Y22
			Lane2			X0Y21	X0Y21
			Lane3			X0Y20	X0Y20
			Lane4				X0Y19
			Lane5				X0Y18
			Lane6				X0Y17
			Lane7				X0Y16
		X0Y2	Lane0	X0Y31	X0Y31	X0Y31	X0Y31
			Lane1		X0Y30	X0Y30	X0Y30
			Lane2			X0Y29	X0Y29
			Lane3			X0Y28	X0Y28
			Lane4				X0Y27
			Lane5				X0Y26
			Lane6				X0Y25
			Lane7				X0Y24
		X0Y3	Lane0	X0Y47	X0Y47	X0Y47	X0Y47
			Lane1		X0Y46	X0Y46	X0Y46
			Lane2			X0Y45	X0Y45
			Lane3			X0Y44	X0Y44
			Lane4				X0Y43
			Lane5				X0Y42
			Lane6				X0Y41
			Lane7				X0Y40

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVE1924	XCVU125	X0Y0	Lane0	X0Y7	X0Y7	X0Y7	X0Y7
			Lane1		X0Y6	X0Y6	X0Y6
			Lane2			X0Y5	X0Y5
			Lane3			X0Y4	X0Y4
			Lane4				X0Y3
			Lane5				X0Y2
			Lane6				X0Y1
			Lane7				X0Y0
		X0Y1	Lane0	X0Y15	X0Y15	X0Y15	X0Y15
			Lane1		X0Y14	X0Y14	X0Y14
			Lane2			X0Y13	X0Y13
			Lane3			X0Y12	X0Y12
			Lane4				X0Y11
			Lane5				X0Y10
			Lane6				X0Y9
			Lane7				X0Y8
		X0Y2	Lane0	X0Y27	X0Y27	X0Y27	X0Y27
			Lane1		X0Y26	X0Y26	X0Y26
			Lane2			X0Y25	X0Y25
			Lane3			X0Y24	X0Y24
			Lane4				X0Y23
			Lane5				X0Y22
			Lane6				X0Y21
			Lane7				X0Y20
		X0Y3	Lane0	X0Y35	X0Y35	X0Y35	X0Y35
			Lane1		X0Y34	X0Y34	X0Y34
			Lane2			X0Y33	X0Y33
			Lane3			X0Y32	X0Y32
			Lane4				X0Y31
			Lane5				X0Y30
			Lane6				X0Y29
			Lane7				X0Y28

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVE1924 (continued)	XCVU160	X0Y1	Lane0	X0Y23	X0Y23	X0Y23	X0Y23
			Lane1		X0Y22	X0Y22	X0Y22
			Lane2			X0Y21	X0Y21
			Lane3			X0Y20	X0Y20
			Lane4				X0Y19
			Lane5				X0Y18
			Lane6				X0Y17
			Lane7				X0Y16
		X0Y2	Lane0	X0Y31	X0Y31	X0Y31	X0Y31
			Lane1		X0Y30	X0Y30	X0Y30
			Lane2			X0Y29	X0Y29
			Lane3			X0Y28	X0Y28
			Lane4				X0Y27
			Lane5				X0Y26
			Lane6				X0Y25
			Lane7				X0Y24
		X0Y3	Lane0	X0Y43	X0Y43	X0Y43	X0Y43
			Lane1		X0Y42	X0Y42	X0Y42
			Lane2			X0Y41	X0Y41
			Lane3			X0Y40	X0Y40
			Lane4				X0Y39
			Lane5				X0Y38
			Lane6				X0Y37
			Lane7				X0Y36
FLVJ1924	XCVU125	X0Y0	Lane0	X0Y7	X0Y7	X0Y7	X0Y7
			Lane1		X0Y6	X0Y6	X0Y6
			Lane2			X0Y5	X0Y5
			Lane3			X0Y4	X0Y4
			Lane4				X0Y3
			Lane5				X0Y2
			Lane6				X0Y1
			Lane7				X0Y0

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVJ1924 (continued)	XCVU125 (continued)	X0Y1	Lane0	X0Y15	X0Y15	X0Y15	X0Y15
			Lane1		X0Y14	X0Y14	X0Y14
			Lane2			X0Y13	X0Y13
			Lane3			X0Y12	X0Y12
			Lane4				X0Y11
			Lane5				X0Y10
			Lane6				X0Y9
			Lane7				X0Y8
		X0Y2	Lane0	X0Y27	X0Y27	X0Y27	X0Y27
			Lane1		X0Y26	X0Y26	X0Y26
			Lane2			X0Y25	X0Y25
			Lane3			X0Y24	X0Y24
			Lane4				X0Y23
			Lane5				X0Y22
			Lane6				X0Y21
			Lane7				X0Y20
		X0Y3	Lane0	X0Y35	X0Y35	X0Y35	X0Y35
			Lane1		X0Y34	X0Y34	X0Y34
			Lane2			X0Y33	X0Y33
			Lane3			X0Y32	X0Y32
			Lane4				X0Y31
			Lane5				X0Y30
			Lane6				X0Y29
			Lane7				X0Y28
	XCVU160	X0Y0	Lane0	X0Y7	X0Y7	X0Y7	X0Y7
			Lane1		X0Y6	X0Y6	X0Y6
			Lane2			X0Y5	X0Y5
			Lane3			X0Y4	X0Y4
			Lane4				X0Y3
			Lane5				X0Y2
			Lane6				X0Y1
			Lane7				X0Y0

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVJ1924 (continued)	XCVU160 (continued)	X0Y1	Lane0	X0Y23	X0Y23	X0Y23	X0Y23
			Lane1		X0Y22	X0Y22	X0Y22
			Lane2			X0Y21	X0Y21
			Lane3			X0Y20	X0Y20
			Lane4				X0Y19
			Lane5				X0Y18
			Lane6				X0Y17
			Lane7				X0Y16
		X0Y2	Lane0	X0Y31	X0Y31	X0Y31	X0Y31
			Lane1		X0Y30	X0Y30	X0Y30
			Lane2			X0Y29	X0Y29
			Lane3			X0Y28	X0Y28
			Lane4				X0Y27
			Lane5				X0Y26
			Lane6				X0Y25
			Lane7				X0Y24
		X0Y3	Lane0	X0Y43	X0Y43	X0Y43	X0Y43
			Lane1		X0Y42	X0Y42	X0Y42
			Lane2			X0Y41	X0Y41
			Lane3			X0Y40	X0Y40
			Lane4				X0Y39
			Lane5				X0Y38
			Lane6				X0Y37
			Lane7				X0Y36
FLVA2377	XCVU125	X0Y0	Lane0	X0Y11	X0Y11	X0Y11	X0Y11
			Lane1		X0Y10	X0Y10	X0Y10
			Lane2			X0Y9	X0Y9
			Lane3			X0Y8	X0Y8
			Lane4				X0Y7
			Lane5				X0Y6
			Lane6				X0Y5
			Lane7				X0Y4

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVA2377 (continued)	XCVU125 (continued)	X0Y1	Lane0	X0Y19	X0Y19	X0Y19	X0Y19
			Lane1		X0Y18	X0Y18	X0Y18
			Lane2			X0Y17	X0Y17
			Lane3			X0Y16	X0Y16
			Lane4				X0Y15
			Lane5				X0Y14
			Lane6				X0Y13
			Lane7				X0Y12
		X0Y2	Lane0	X0Y35	X0Y35	X0Y35	X0Y35
			Lane1		X0Y34	X0Y34	X0Y34
			Lane2			X0Y33	X0Y33
			Lane3			X0Y32	X0Y32
			Lane4				X0Y31
			Lane5				X0Y30
			Lane6				X0Y29
			Lane7				X0Y28
		X0Y3	Lane0	X0Y39	X0Y39	X0Y39	Not supported
			Lane1		X0Y38	X0Y38	
			Lane2			X0Y37	
			Lane3			X0Y36	
	XCVU160	X0Y1	Lane0	X0Y27	X0Y27	X0Y27	X0Y27
			Lane1		X0Y26	X0Y26	X0Y26
			Lane2			X0Y25	X0Y25
			Lane3			X0Y24	X0Y24
			Lane4				X0Y23
			Lane5				X0Y22
			Lane6				X0Y21
			Lane7				X0Y20

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLVA2377 (continued)	XCVU160 (continued)	X0Y2	Lane0	X0Y35	X0Y35	X0Y35	X0Y35
			Lane1		X0Y34	X0Y34	X0Y34
			Lane2			X0Y33	X0Y33
			Lane3			X0Y32	X0Y32
			Lane4				X0Y31
			Lane5				X0Y30
			Lane6				X0Y29
			Lane7				X0Y28
		X0Y3	Lane0	X0Y47	X0Y47	X0Y47	X0Y47
			Lane1		X0Y46	X0Y46	X0Y46
			Lane2			X0Y45	X0Y45
			Lane3			X0Y44	X0Y44
			Lane4				X0Y43
			Lane5				X0Y42
			Lane6				X0Y41
			Lane7				X0Y40
FLGB2377	XCVU440	X0Y0	Lane0	X0Y15	X0Y15	X0Y15	X0Y15
			Lane1		X0Y14	X0Y14	X0Y14
			Lane2			X0Y13	X0Y13
			Lane3			X0Y12	X0Y12
			Lane4				X0Y11
			Lane5				X0Y10
			Lane6				X0Y9
			Lane7				X0Y8
		X0Y1	Lane0	X0Y19	X0Y19	X0Y19	Not supported
			Lane1		X0Y18	X0Y18	
			Lane2			X0Y17	
			Lane3			X0Y16	

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLGB2377 (continued)	XCVU440 (continued)	X0Y2	Lane0	X0Y27	X0Y27	X0Y27	X0Y27
			Lane1		X0Y26	X0Y26	X0Y26
			Lane2			X0Y25	X0Y25
			Lane3			X0Y24	X0Y24
			Lane4				X0Y23
			Lane5				X0Y22
			Lane6				X0Y21
			Lane7				X0Y20
		X0Y3	Lane0	X0Y31	X0Y31	X0Y31	Not supported
			Lane1		X0Y30	X0Y30	
			Lane2			X0Y29	
			Lane3			X0Y28	
		X0Y4	Lane0	X0Y55	X0Y55	X0Y55	X0Y55
			Lane1		X0Y54	X0Y54	X0Y54
			Lane2			X0Y53	X0Y53
			Lane3			X0Y52	X0Y52
			Lane4				X0Y51
			Lane5				X0Y50
			Lane6				X0Y49
			Lane7				X0Y48
		X0Y5	Lane0	X0Y59	X0Y59	X0Y59	Not supported
			Lane1		X0Y58	X0Y58	
			Lane2			X0Y57	
			Lane3			X0Y56	
FLGA2892	XCVU440	X0Y0	Lane0	X0Y7	X0Y7	X0Y7	X0Y7
			Lane1		X0Y6	X0Y6	X0Y6
			Lane2			X0Y5	X0Y5
			Lane3			X0Y4	X0Y4
			Lane4				X0Y3
			Lane5				X0Y2
			Lane6				X0Y1
			Lane7				X0Y0

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLGA2892 (continued)	XCVU440 (continued)	X0Y1	Lane0	X0Y15	X0Y15	X0Y15	X0Y15
			Lane1		X0Y14	X0Y14	X0Y14
			Lane2			X0Y13	X0Y13
			Lane3			X0Y12	X0Y12
			Lane4				X0Y11
			Lane5				X0Y10
			Lane6				X0Y9
			Lane7				X0Y8
		X0Y2	Lane0	X0Y27	X0Y27	X0Y27	X0Y27
			Lane1		X0Y26	X0Y26	X0Y26
			Lane2			X0Y25	X0Y25
			Lane3			X0Y24	X0Y24
			Lane4				X0Y23
			Lane5				X0Y22
			Lane6				X0Y21
			Lane7				X0Y20
		X0Y3	Lane0	X0Y35	X0Y35	X0Y35	X0Y35
			Lane1		X0Y34	X0Y34	X0Y34
			Lane2			X0Y33	X0Y33
			Lane3			X0Y32	X0Y32
			Lane4				X0Y31
			Lane5				X0Y30
			Lane6				X0Y29
			Lane7				X0Y28
		X0Y4	Lane0	X0Y47	X0Y47	X0Y47	X0Y47
			Lane1		X0Y46	X0Y46	X0Y46
			Lane2			X0Y45	X0Y45
			Lane3			X0Y44	X0Y44
			Lane4				X0Y43
			Lane5				X0Y42
			Lane6				X0Y41
			Lane7				X0Y40

Table B-2: **Virtex Ultrascale Core Pinouts *(Cont'd)***

Package	Device	PCIe blocks	Lane	x1	x2	x4	x8
FLGA2892 (continued)	XCVU440 (continued)	X0Y5	Lane0	X0Y51	X0Y51	X0Y55	X0Y55
			Lane1		X0Y50	X0Y54	X0Y54
			Lane2			X0Y53	X0Y53
			Lane3			X0Y52	X0Y52
			Lane4				X0Y51
			Lane5				X0Y50
			Lane6				X0Y49
			Lane7				X0Y48

Managing Receive-Buffer Space for Inbound Completions

The *PCI Express® Base Specification* [Ref 2] requires all Endpoints to advertise infinite Flow Control credits for received Completions to their link partners. This means that an Endpoint must only transmit Non-Posted Requests for which it has space to accept Completion responses. This appendix describes how a user application can manage the receive-buffer space in the Virtex-7 Gen3 Integrated Block for PCIe core to fulfill this requirement.

General Considerations and Concepts

Completion Space

Table C-1 defines the completion space reserved in the receive buffer by the core. The values differ depending on the different Capability Max Payload Size settings of the core and the performance level that you selected. Values are credits, expressed in decimal.

Table C-1: **Receiver-Buffer Completion Space**

Capability Max Payload Size (bytes)	Performance Level: Good		Performance Level: High	
	CPH	CPD	CPH	CPD
128	64	7,936B	64	15,872B
256	64	7,936B	64	15,872B
512	64	7,936B	64	15,872B
1024	64	7,936B	64	15,872B

Maximum Request Size

A Memory Read cannot request more than the value stated in Max_Request_Size, which is given by Configuration bits cfg_dcommand[14:12] as defined in Table C-2. If the user application does not read the Max_Request_Size value, it must use the default value of 128 bytes.

Table C-2: **Max_Request_Size Settings**

cfg_dcommand[14:12]	Max_Request_Size			
	Bytes	DW	QW	Credits
000b	128	32	16	8
001b	256	64	32	16
010b	512	128	64	32
011b	1024	256	128	64
100b	2048	512	256	128
101b	4096	1024	512	256
110b-111b	Reserved			

Read Completion Boundary

A memory read can be answered with multiple completions, which when put together return all requested data. To make room for packet-header overhead, the user application must allocate enough space for the maximum number of completions that might be returned.

To make this process easier, the *PCI Express Base Specification* quantizes the length of all completion packets such that each completion must start and end on a naturally aligned read completion boundary (RCB), unless, it services the starting or ending address of the original request. Requests which cross the address boundaries at integer multiples of RCB bytes can be completed using more than one completion, but the returned data must not be fragmented except along the following address boundaries:

- The first completion must start with the address specified in the request, and must end at one of the following:
  - The address specified in the request plus the length specified by the request (for example, the entire request).
  - An address boundary between the start and end of the request at an integer multiple of RCB bytes.
- The final completion must end with the address specified in the request plus the length specified by the request.
- All completions between, but not including, the first and final completions must be an integer multiple of RCB bytes in length.

The programmed value of RCB is provided on `cfg_rcb_status[1:0]`. Here `cfg_rcb_status[0]` and `cfg_rcb_status[1]` are associated with Physical Functions 0 and 1 respectively (Per Function Link Control register [3]). If the user application does not read the RCB value, it must use the default value of 64 bytes.

Table C-3: **Read Completion Boundary Settings**

cfg_rcb_status[0] or cfg_rcb_status[1]	Read Completion Boundary			
	Bytes	DW	QW	Credits
0	64	16	8	4
1	128	32	16	8

When calculating the number of completion credits a non-posted request requires, you must determine how many RCB-bounded blocks the completion response might be required, which is the same as the number of completion header credits required.

Important Note For High Performance Applications

While programmed RCB value can be used by the user application to compute the maximum number of completions returned for a request, most high performance memory controllers have the optional feature to combine RCB-sized completions in response to large read requests (read lengths multiples of RCB value), into completions that are at or near the programmed Max_Payload_Size value for the link. You are encouraged to take advantage of this feature, if supported, by memory controller on the host CPU. Data exchange based on completions that are integer multiples (>1) of RCB value results in greater PCI Express interface utilization and payload efficiency, as well as, more efficient use of completion space in the Endpoint receiver.

Methods of Managing Completion Space

A user application can choose one of five methods to manage receive-buffer completion space, as listed in Table C-4. For convenience, this discussion refers to these methods as LIMIT_FC, PACKET_FC, RCB_FC, and DATA_FC. Each method has advantages and disadvantages that you need to consider when developing the user application.

Table C-4: **Managing Receive Completion Space Methods**

Method	Description	Advantage	Disadvantage
LIMIT_FC	Limit the total number of outstanding NP Requests	Simplest method to implement in user logic	Much Completion capacity goes unused
PACKET_FC	Track the number of outstanding CplH and CplD credits; allocate and deallocate on a per-packet basis	Relatively simple user logic; finer allocation granularity means less wasted capacity than LIMIT_FC	As with LIMIT_FC, credits for an NP are still tied up until the request is completely satisfied

Table C-4: **Managing Receive Completion Space Methods *(Cont'd)***

Method	Description	Advantage	Disadvantage
RCB_FC	Track the number of outstanding CplH and CplD credits; allocate and deallocate on a per-RCB basis	Ties up credits for less time than PACKET_FC	More complex user logic than LIMIT_FC or PACKET_FC
DATA_FC	Track the number of outstanding CplH and CplD credits; allocate and deallocate on a per-RCB basis	Lowest amount of wasted capacity	More complex user logic than LIMIT_FC, PACKET_FC, and RCB_FC

LIMIT_FC Method

The LIMIT_FC method is the simplest to implement. The user application assesses the maximum number of outstanding Non-Posted Requests allowed at one time, MAX_NP. To calculate this value, perform these steps:

1. Determine the number of CplH credits required by a Max_Request_Size packet:

   Max_Header_Count = ceiling(Max_Request_Size / RCB)

2. Determine the greatest number of maximum-sized completions supported by the CplD credit pool:

   Max_Packet_Count_CplD = floor(CplD / Max_Request_Size)

3. Determine the greatest number of maximum-sized completions supported by the CplH credit pool:

   Max_Packet_Count_CplH = floor(CplH / Max_Header_Count)

4. Use the *smaller* of the two quantities from steps 2 and 3 to obtain the maximum number of outstanding Non-Posted requests:

   MAX_NP = min(Max_Packet_Count_CplH, Max_Packet_Count_CplD)

With knowledge of MAX_NP, the user application can load a register NP_PENDING with zero at reset and make sure it always stays with the range 0 to MAX_NP. When a non-posted request is transmitted, NP_PENDING decreases by one. When *all* completions for an outstanding non-posted request are received, NP_PENDING increases by one.

For example:

- Max_Request_Size = 128B
- RCB = 64B
- CplH = 64
- CplD = 15,872B

- Max_Header_Count = 2
- Max_Packet_Count_CplD = 124
- Max_Packet_Count_CplH = 32
- MAX_NP = 32

Although this method is the simplest to implement, it can waste the greatest receiver space because an entire Max_Request_Size block of completion credit is allocated for each non-posted request, regardless of actual request size. The amount of waste becomes greater when the user application issues a larger proportion of short memory reads (on the order of a single DWORD), I/O reads and I/O writes.

PACKET_FC Method

The PACKET_FC method allocates blocks of credit in finer granularities than LIMIT_FC, using the receive completion space more efficiently with a small increase in user logic.

Start with two registers, CPLH_PENDING and CPLD_PENDING, (loaded with zero at reset), and then perform these steps:

1. When the user application needs to send an NP request, determine the potential number of CplH and CplD credits it might require:

   NP_CplH = ceiling[((Start_Address mod RCB) + Request_Size) / RCB]

   NP_CplD = ceiling[((Start_Address mod 16 bytes) + Request_Size) /16 bytes]
   (except I/O Write, which returns zero data) [(req_size + 15)/16]

   The modulo and ceiling functions ensure that any fractional RCB or credit blocks are rounded up. For example, if a memory read requests 8 bytes of data from address `7Ch`, the returned data can potentially be returned over two completion packets (`7Ch-7Fh`, followed by `80h-83h`). This would require two RCB blocks and two data credits.

2. Check these:

   CPLH_PENDING + NP_CplH $\leq$ Total_CplH

   CPLD_PENDING + NP_CplD $\leq$ Total_CplD

3. If both inequalities are true, transmit the non-posted request, and increase CPLH_PENDING by NP_CplH and CPLD_PENDING by NP_CplD. For each non-posted request transmitted, keep NP_CplH and NP_CplD for later use.

4. When all completion data is returned for an non-posted request, decrease CPLH_PENDING and CPLD_PENDING accordingly.

This method is less wasteful than LIMIT_FC but still ties up all of an non-posted request completion space until the *entire* request is satisfied. RCB_FC and DATA_FC provide finer de-allocation granularity at the expense of more logic.

RCB_FC Method

The RCB_FC method allocates and de-allocates blocks of credit in RCB granularity. Credit is freed on a per-RCB basis.

As with PACKET_FC, start with two registers, CPLH_PENDING and CPLD_PENDING (loaded with zero at reset).

1. Calculate the number of data credits per RCB:

   CplD_PER_RCB = RCB / 16 bytes

2. When the user application needs to send an non-posted request, determine the potential number of CplH credits it might require. Use this to allocate CplD credits with RCB granularity:

   NP_CplH = ceiling[((Start_Address mod RCB) + Request_Size) / RCB]

   NP_CplD = NP_CplH × CplD_PER_RCB

3. Check these:

   CPLH_PENDING + NP_CplH ≤ Total_CplH

   CPLD_PENDING + NP_CplD ≤ Total_CplD

4. If both inequalities are true, transmit the non-posted request, increase CPLH_PENDING by NP_CplH and CPLD_PENDING by NP_CplD.
5. At the start of each incoming completion, or when that completion begins at or crosses an RCB without ending at that RCB, decrease CPLH_PENDING by 1 and CPLD_PENDING by CplD_PER_RCB. Any completion could cross more than one RCB. The number of RCB crossings can be calculated by:

   RCB_CROSSED = ceiling[((Lower_Address mod RCB) + Length) / RCB]

   Lower_Address and Length are fields that can be parsed from the Completion header. Alternatively, you can load a register CUR_ADDR with Lower_Address at the start of each incoming completion, increment per DW or QW as appropriate, then count an RCB whenever CUR_ADDR rolls over.

This method is less wasteful than PACKET_FC but still gives an RCB granularity. If a user application transmits I/O requests, the user application could adopt a policy of only allocating one CplD credit for each I/O read and zero CplD credits for each I/O write. The user application would have to match each tag for incoming completions with the type (Memory Write, I/O Read, I/O Write) of the original non-posted request.

DATA_FC Method

The DATA_FC method provides the finest allocation granularity at the expense of logic.

As with PACKET_FC and RCB_FC, start with two registers, CPLH_PENDING and CPLD_PENDING (loaded with zero at reset).

1. When the user application needs to send an non-posted request, determine the potential number of CplH and CplD credits it might require:

   NP_CplH = ceiling[((Start_Address mod RCB) + Request_Size) / RCB]

   NP_CplD = ceiling[((Start_Address mod 16 bytes) + Request_Size) / 16 bytes]
   (except I/O Write, which returns zero data)

2. Check these:

   CPLH_PENDING + NP_CplH ≤ Total_CplH

   CPLD_PENDING + NP_CplD ≤ Total_CplD

3. If both inequalities are true, transmit the non-posted request, increase CPLH_PENDING by NP_CplH and CPLD_PENDING by NP_CplD.
4. At the start of each incoming completion, or when that completion begins at or crosses an RCB without ending at that RCB, decrease CPLH_PENDING by 1. The number of RCB crossings can be calculated by:

   RCB_CROSSED = ceiling[((Lower_Address mod RCB) + Length) / RCB]

   Lower_Address and Length are fields that can be parsed from the completion header. Alternatively, you can load a register CUR_ADDR with Lower_Address at the start of each incoming completion, increment per DW or QW as appropriate, then count an RCB whenever CUR_ADDR rolls over.

5. At the start of each incoming completion, or when that completion begins at or crosses at a naturally aligned credit boundary, decrease CPLD_PENDING by 1. The number of credit-boundary crossings is given by:

   DATA_CROSSED = ceiling[((Lower_Address mod 16 B) + Length) / 16 B]

   Alternatively, you can load a register CUR_ADDR with Lower_Address at the start of each incoming completion, increment per DW or QW as appropriate, then count an RCB whenever CUR_ADDR rolls over each 16-byte address boundary.

This method is the least wasteful but requires the greatest amount of user logic. If even finer granularity is desired, you can scale the Total_CplD value by 2 or 4 to get the number of completion QWORDs or DWORDs, respectively, and adjust the data calculations accordingly.

Debugging

This appendix includes details about resources available on the Xilinx Support website and debugging tools.

Finding Help on Xilinx.com

To help in the design and debug process when using the UltraScale FPGAs Gen3 Integrated Block for PCIe, the Xilinx Support web page (www.xilinx.com/support) contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support.

Documentation

This product guide is the main document associated with the UltraScale FPGA Gen3 Integrated Block for PCIe. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page (www.xilinx.com/support) or by using the Xilinx® Documentation Navigator.

Download the Xilinx Documentation Navigator from the Design Tools tab on the Downloads page (www.xilinx.com/download). For more information about this tool and the features available, open the online help after installation.

Answer Records

Answer Records include information about commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a Xilinx product. Answer Records are created and maintained daily ensuring that users have access to the most accurate information available.

Answer Records for this core can be located by using the Search Support box on the main Xilinx support web page. To maximize your search results, use proper keywords such as

- Product name
- Tool message(s)
- Summary of the issue encountered

A filter search is available after results are returned to further target the results.

Master Answer Record for the UltraScale FPGA Gen3 Integrated Block for PCIe

AR: 57945

Contacting Technical Support

Xilinx provides technical support at www.xilinx.com/support for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support of product if implemented in devices that are not defined in the documentation, if customized beyond that allowed in the product documentation, or if changes are made to any section of the design labeled DO NOT MODIFY.

To contact Xilinx Technical Support:

1. Navigate to www.xilinx.com/support.
2. Open a WebCase by selecting the WebCase link located under Additional Resources.

When opening a WebCase, include:

- Target FPGA including package and speed grade.
- All applicable Xilinx Design Tools and simulator software versions.
- Additional files based on the specific issue might also be required. See the relevant sections in this debug guide for guidelines about which file(s) to include with the WebCase.

Note: Access to WebCase is not available in all cases. Log in to the WebCase tool to see your specific support options.

Appendix E

Additional Resources and Legal Notices

Xilinx Resources

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

For a glossary of technical terms used in Xilinx documentation, see the Xilinx Glossary.

References

These documents provide supplemental material useful with this product guide:

1. *AMBA AXI4-Stream Protocol Specification*
2. PCI-SIG Documentation (www.pcisig.com/specifications)
3. *LogiCORE IP Virtex-7 FPGA Integrated Block for PCI Express Product Guide* (PG023)
4. *Vivado Design Suite User Guide: Designing with IP* (UG896)
5. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
6. *Vivado Design Suite User Guide: Getting Started* (UG910)
7. *Vivado Design Suite User Guide: Using Constraints* (UG903)
8. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
9. *ISE to Vivado Design Suite Migration Methodology Guide* (UG911)
10. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
11. ATX Power Supply Design Guide

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/04/2014	3.0	• Updated device information.
04/02/2014	3.0	• Updated block selection. • Updated core pinout information. • Updated shared logic information.
12/18/2013	2.0	Initial release.

Please Read: Important Legal Notices