DMA Subsystem for PCI Express v2.0

Product Guide

Vivado Design Suite

PG195 June 8, 2016

Table of Contents

Introduction

The Xilinx® DMA Subsystem for PCI Express® (PCIe™) implements a high performance, configurable Scatter Gather DMA for use with the PCI Express® 3.x Integrated Block. The IP provides an optional AXI4 or AXI4-Stream user interface.

Features

- Supports UltraScale™ architecture and Virtex®-7 XT FPGA Gen3 Integrated (Endpoint only) Blocks for PCI Express
- Support for 64, 128, and 256-bit datapath width matching the PCI Express Integrated Block interface datapath width
- 64-bit source, destination, and descriptor address
- Up to four host-to-card (H2C/Read) data channels
- Up to four card-to-host (C2H/Write) data channels
- Configurable user interface - Common AXI4 memory mapped (MM) user interface or a separate AXI4-Stream user interface
- Compliant with AXI4, AXI4-Lite, and AXI4-Stream protocols
- AXI4 MM Master DMA host or peer initiated bypass interface for high bandwidth access to user logic
- Host configuration access to user logic through an AXI4-Lite Master interface
- IP internal configuration and status registers access to user logic through an AXI4-Lite Slave interface
- Scatter Gather descriptor list supporting unlimited list size
- 28-bit max transfer length per descriptor
- Legacy, MSI, and MSI-X interrupts
- Block fetches of contiguous descriptors
- Poll Mode
- Descriptor Bypass interface
- Arbitrary source and destination address
- Linux driver available (see AR 65444)

IP Facts Table	
Core Specifics	
Supported Device Family[1]	UltraScale Architecture, Virtex-7 XT
Supported User Interfaces	AXI4, AXI4-Lite, AXI4-Stream
Resources	Performance and Resource Utilization web page
Provided with Core	
Design Files	Encrypted System Verilog
Example Design	Verilog
Test Bench	Verilog
Constraints File	XDC
Simulation Model	Verilog
Supported S/W Driver	Linux Driver
Tested Design Flows[2]	
Design Entry	Vivado® Design Suite
Simulation	For supported simulators, see the Xilinx Design Tools: Release Notes Guide.
Synthesis	Vivado synthesis
Support	
Provided by Xilinx at the Xilinx Support web page	

Notes:

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

Overview

The DMA Subsystem for PCI Express® (PCIe™) is designed for the Vivado® IP integrator in the Vivado Design Suite. The IP provides a flexible hardware and software solution to offload PCIe memory transfers from the host. The IP driver has a character interface. The driver is responsible for generating a descriptor list from the user workload and initializing the IP. The IP fetches the descriptor lists from host memory. To perform a host-to-card DMA transfer, the IP masters memory reads to the PCIe Gen3 core and writes the completion data to the user AXI4 write interface. For a card-to-host transfer, the IP acquires data from the AXI4 read interface and masters memory write requests to PCIe.

Figure 1-1 shows an overview of the DMA Subsystem for PCIe.

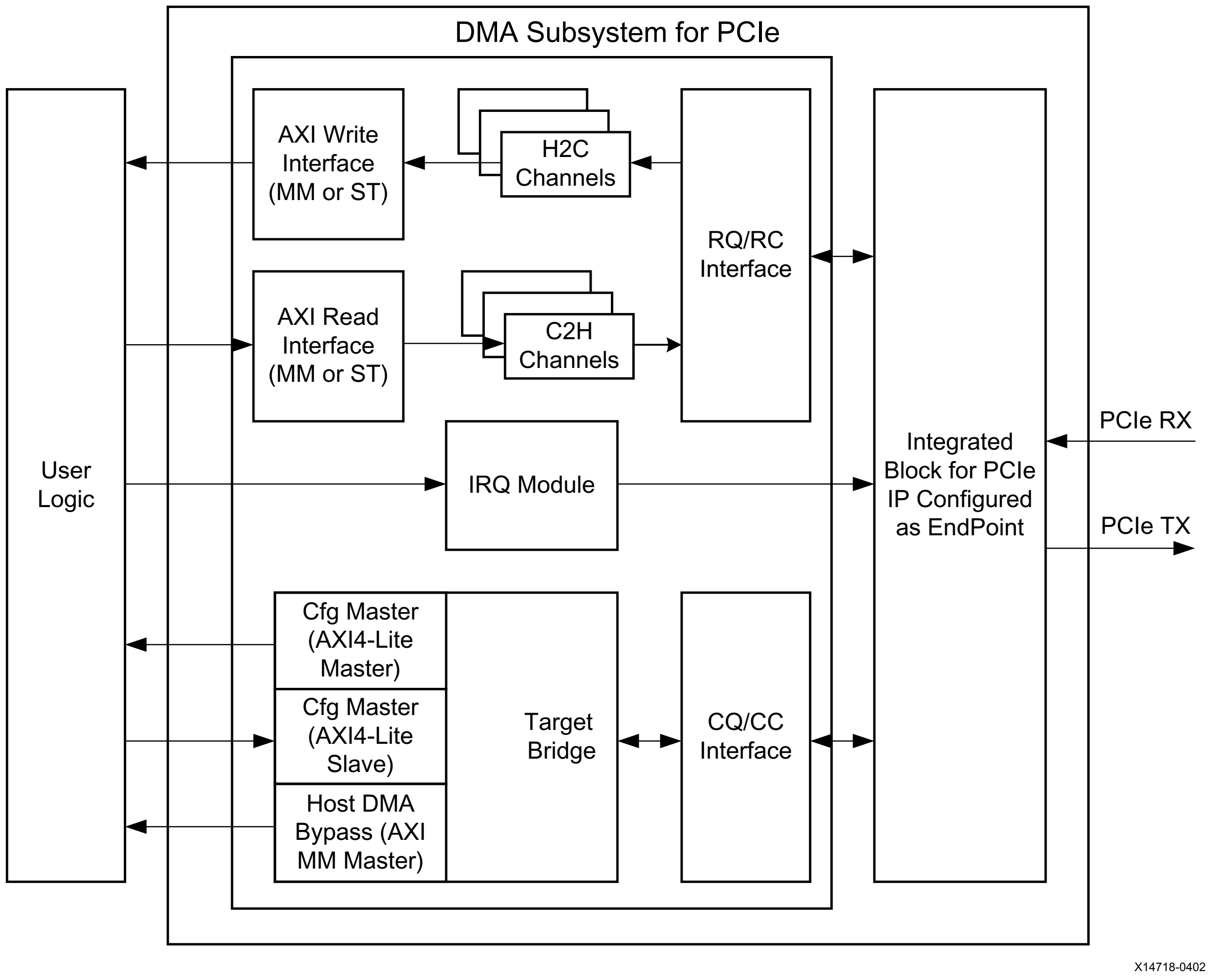

DMA Subsystem for PCIe
User Logic
AXI Write Interface (MM or ST)
H2C Channels
RQ/RC Interface
AXI Read Interface (MM or ST)
C2H Channels
IRQ Module
Integrated Block for PCIe IP Configured as EndPoint
PCIe RX
PCIe TX
Cfg Master (AXI4-Lite Master)
Cfg Master (AXI4-Lite Slave)
Host DMA Bypass (AXI MM Master)
Target Bridge
CQ/CC Interface
X14718-0402

Figure 1-1: **DMA Subsystem for PCIe Overview**

Feature Summary

The DMA Subsystem for PCIe masters read and write requests on the PCI Express Gen3 core and user logic to perform direct memory transfers between the two interfaces. The core can be configured to have a common AXI4 memory mapped interface shared by all channels or an AXI4-Stream interface per channel. Memory transfers are specified on a per-channel basis in descriptor linked lists, which the DMA fetches from host memory and processes. Events such as descriptor completion and errors are signaled using interrupts. The core also provides a configurable number of user interrupt wires that generate interrupts to the host.

The host is able to directly access the user logic through two interfaces:

- **The AXI4-Lite Master Configuration port**: This port is a fixed 32-bit port and is intended for non-performance-critical access to user configuration and status registers.
- **The AXI Memory Mapped Master CQ Bypass port**: The width of this port is the same as the DMA channel datapaths and is intended for high-bandwidth access to user memory that might be required in applications such as peer-to-peer transfers.

The user logic is able to access the DMA Subsystem for PCIe internal configuration and status registers through an AXI4-Lite Slave Configuration interface. Requests that are mastered on this port are not forwarded to PCI Express.

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 following are not supported by this core:

- Root Port configurations
- Tandem Configuration solutions (Tandem PROM, Tandem PCIe, Tandem with Field Updates, PR over PCIe) are not supported for Virtex-7 XT devices
- SRIOV
- ECRC
- Parity generation and checking
- Example design not supported for all configurations
- Narrow burst

Limitations

PCIe Transaction Type

The PCIe transactions that can be generated are those that are compatible with the AXI4 specification. Table 1-1 lists the supported PCIe transaction types.

Table 1-1: **Supported PCIe Transaction Types**

TX	RX
MRd32	MRd32
MRd64	MRd64
MWr32	MWr32
MWr64	MWr64
Msg(INT/Error)	Msg(SSPL,INT,Error)
Cpl	Cpl
CplD	CplD

PCIe Capability

For the DMA Subsystem for PCIe, only the following PCIe capabilities are supported due to the AXI4 specification:

- 1 PF
- MSI
- MSI-X
- PM
- AER

Others

- Only supports the INCR burst type. Other types result in the Slave Illegal Burst (SIB) interrupt.
- No memory type support (AxCACHE)
- No protection type support (AxPROT)
- No lock type support (AxLOCK)
- No non-contiguous byte enable support (WSTRB)

PCIe to DMA Bypass Master

- Only issues the INCR burst type
- Only issues the Data, Non-secure, and Unprivileged protection type

Licensing and Ordering Information

This Xilinx IP module is provided at no additional cost with the Xilinx Vivado Design Suite under the terms of the Xilinx End User License. Information about this and other Xilinx IP modules is available at the Xilinx Intellectual Property page. For information about pricing and availability of other Xilinx IP modules and tools, contact your local Xilinx sales representative.

For more information, visit the DMA Subsystem for PCI Express product page.

Product Specification

The DMA Subsystem for PCI Express® (PCIe™), in conjunction with the Integrated Block for PCI Express IP, provides a highly configurable DMA Subsystem for PCIe, and a high performance DMA solution.

Configurable Components of the Core

Internally, the core can be configured to implement up to eight independent physical DMA engines. These DMA engines can be mapped to individual AXI4-Stream interfaces or a shared AXI4 memory mapped (MM) interface to the user application. On the AXI4 MM interface, the DMA Subsystem for PCIe generates requests and expected completions. The AXI4-Stream interface is data-only. On the PCIe side, the DMA has internal arbitration and bridge logic to generate read and write transaction level packets (TLPs) to the PCIe over the Integrated Block for the PCIe core Requester Request (RQ) bus, and to accept completions from PCIe over the Integrated Block for the PCIe Request Completion (RC) bus. For details about the RQ and RC, see the *Virtex-7 FPGA Integrated Block for PCI Express Product Guide* (PG023) [Ref 3], or *UltraScale Architecture Gen3 Integrated Block for PCI Express Product Guide* (PG156) [Ref 4].

The type of channel configured determines the transactions on which bus.

- An Host-to-Card (H2C) channel generates read requests to PCIe and provides the data or generates a write request to the user application.
- Similarly, a Card-to-Host (C2H) channel either waits for data on the user side or generates a read request on the user side and then generates a write request containing the data received to PCIe.

The DMA Subsystem for PCIe also enables the host to access the user logic. Write requests that reach ‘PCIe to DMA bypass Base Address Register (BAR)’ are processed by the DMA. The data from the write request is forwarded to the user application through the AXI4-Stream CQ forwarding interface.

The host access to the configuration and status registers in the user logic is provided through an AXI4-Lite master port. These requests are 32-bit reads or writes. The user application also has access to internal DMA configuration and status registers through an AXI4-Lite slave port.

Target Bridge

The target bridge receives requests from the host. Based on BARs, the requests are directed to the internal target user through the AXI4-Lite master, or the CQ bypass port. After the downstream user logic has returned data for a non-posted request, the target bridge generates a read completion TLP and sends it to the PCIe IP over the CC bus.

Table 2-1: **32-Bit BARs**

	BAR0 (32-bit)	BAR1 (32-bit)	BAR2 (32-bit)
Default	DMA		
PCIe to AXI Lite Master enabled	PCIe to AXI Lite Master	DMA	
PCIe to AXI Lite Master and **PCIe to DMA Bypass** enabled	PCIe to AXI Lite Master	DMA	PCIe to DMA Bypass
PCIe to DMA Bypass enabled	DMA	PCIe to DMA Bypass	

Table 2-2: **64-Bit BARs**

	BAR0 (64-bit)	BAR2 (64-bit)	BAR4 (64-bit)
Default	DMA		
PCIe to AXI Lite Master enabled	PCIe to AXI Lite Master	DMA	
PCIe to AXI Lite Master and **PCIe to DMA Bypass** enabled	PCIe to AXI Lite Master	DMA	PCIe to DMA Bypass
PCIe to DMA Bypass enabled	DMA	PCIe to DMA Bypass	

H2C Channel

The number of H2C channels is configured in the Vivado® Integrated Design Environment (IDE). The H2C channel handles DMA transfers from the host to the card. It is responsible for splitting read requests based on maximum read request size, and available internal resources. The DMA channel maintains a maximum number of outstanding requests based on the RNUM_RIDS, which is the number of outstanding H2C channel request ID parameter. Each split, if any, of a read request consumes an additional read request entry. A request is outstanding after the DMA channel has issued the read to the PCIe RQ block to when it receives confirmation that the write has completed on the user interface in-order. After a transfer is complete, the DMA channel issues a writeback or interrupt to inform the host.

The H2C channel also splits transaction on both its read and write interfaces. On the read interface to the host, transactions are split to meet the maximum read request size configured, and based on available Data FIFO space. Data FIFO space is allocated at the time of the read request to ensure space for the read completion. The PCIe RC block returns completion data to the allocated Data Buffer locations. To minimize latency, upon receipt of any completion data, the H2C channel begins issuing write requests to the user interface. It also breaks the write requests into maximum payload size. On an AXI4-Stream user interface, this splitting is transparent.

C2H Channel

The C2H channel handles DMA transfers from the card to the host. The instantiated number of C2H channels is controlled in the Vivado IDE. Similarly the number of outstanding transfers is configured through the WNUM_RIDS, which is the number of C2H channel request IDs. In an AXI4-Stream configuration, the details of the DMA transfer are set up in advance of receiving data on the AXI4-Stream interface. This is normally accomplished through receiving a DMA descriptor. After the a request ID has been prepared and the channel is enabled, the AXI4-Stream interface of the channel can receive data and perform the DMA to the host. In an AXI4 MM interface configuration, the request IDs are allocated as the read requests to the AXI4 MM interface are issued. Similar to the H2C channel, a given request ID is outstanding until the write request has been completed. In the case of the C2H channel, write request completion is when the write request has been issued as indicated by the PCIe IP.

AXI4-Lite Master

This module implements the AXI4-Lite master bus protocol. The host can use this interface to generate 32-bit read and 32-bit write requests to the user logic. The read or write request is received over the PCIe IP CQ bus and must target 'PCIe to AXI-Lite Master BAR'. Read completion data is returned back to the host through the target bridge over the PCIe IP CC bus.

AXI4-Lite Slave

This module implements the AXI4-Lite Slave bus protocol. The user logic can master 32-bit reads or writes on this interface to DMA internal registers. This interface does not generate requests to the host.

IRQ Module

The IRQ module receives a configurable number of interrupt wires from the user logic, and one interrupt wire from each DMA channel and is responsible for generating an interrupt over PCIe. Support for MSI-X, MSI, and legacy interrupts can be specified during IP configuration.

Legacy Interrupts

Asserting one or more bits of `usr_irq_req` when legacy interrupts are enabled causes the DMA to issue a legacy interrupt over PCIe. Multiple bits may be asserted simultaneously but each bit must remain asserted until the corresponding `usr_irq_ack` bit has been asserted. Assertion of this `usr_irq_ack` bit indicates the message was sent over PCIe. After the `user_irq_req` bit is deasserted, it cannot be reasserted until the corresponding `usr_irq_ack` bit has been asserted for a second time. This indicates the deassertion message for the legacy interrupt has been sent over PCIe. After a second usr_irq_ack has occurred, the usr_irq_req wire can be reasserted to generate another legacy interrupt.

The usr_irq_req bit and DMA interrupts can be mapped to legacy interrupt INTA, INTB, INTC, INTD through the configuration registers. Figure 2-1 shows the legacy interrupts.

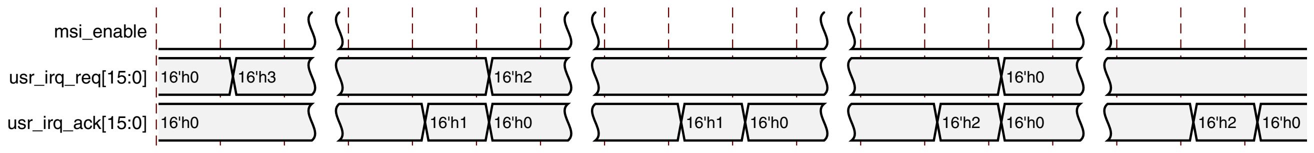


Figure 2-1: **Legacy Interrupts**

MSI and MSI-X Interrupts

Asserting one or more bits of usr_irq_req causes the generation of an MSI or MSI-X interrupt if MSI or MSI-X is enabled. If both MSI and MSI-X capabilities are enabled, an MSI-X interrupt is generated.

After a usr_irq_req bit is asserted, it must remain asserted until the corresponding usr_irq_ack bit is asserted. The usr_irq_ack bit assertion indicates the requested interrupt has been sent on PCIe. Once this ack has been observed, the usr_irq_req bit can be deasserted.

Configuration registers are available to map usr_irq_req and DMA interrupts to MSI or MSI-X vectors. For MSI-X support, there is also a vector table and PBA table. Figure 2-2 shows the MSI interrupt.

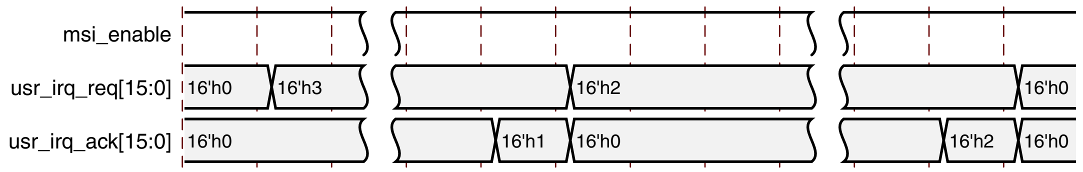


Figure 2-2: **MSI Interrupts**

Figure 2-3 shows the MSI-X interrupt.

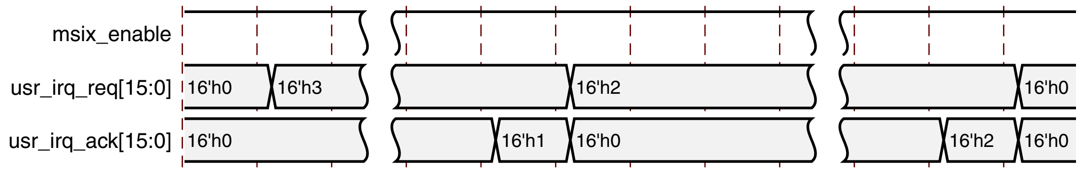


Figure 2-3: **MSI-X Interrupts**

For more details, see Interrupt Processing in Appendix A.

Host-to-Card Bypass Master

Host requests that reach the 'PCIe to DMA bypass' BAR are sent to this module. The bypass master port is an AXI4 MM interface and supports read and write accesses.

Config Block

The config module, the DMA register space which contains PCIe solution IP configuration information and DMA control registers, stores PCIe IP configuration information that is relevant to the DMA Subsystem for PCIe. This configuration information can be read through register reads to the appropriate register offset within the config module.

DMA Operations

The DMA Subsystem for PCIe uses a linked list of descriptors that specify the source, destination, and length of the DMA transfers. Descriptor lists are created by the driver and stored in host memory. The DMA channel is initialized by the driver with a few control registers to begin fetching the descriptor lists and executing the DMA operations.

Descriptors

Descriptors describe the memory transfers that the DMA Subsystem for PCIe should perform. Each channel has its own descriptor list. The start address of each channel's descriptor list is initialized in hardware registers by the driver. After the channel is enabled, the descriptor channel begins to fetch descriptors from the initial address. Thereafter, it fetches from the Nxt_adr[63:0] field of the last descriptor that was fetched. Descriptors must be aligned to a 32 byte boundary.

The size of the initial block of adjacent descriptors are specified with the Dsc_Adj register. After the initial fetch, the descriptor channel uses the Nxt_adj field of the last fetched descriptor to determine the number of descriptors at the next descriptor address. A block of adjacent descriptors must not cross a 4K address boundary. The descriptor channel fetches as many descriptors in a single request as it can, limited by MRRS, the number the adjacent descriptors, and the available space in the channel's descriptor buffer.

Every descriptor in the descriptor list must accurately describe the descriptor or block of descriptors that follows. In a block of adjacent descriptors, the Nxt_adj value decrements from the first descriptor to the second to last descriptor which has a value of zero. Likewise, each descriptor in the block points to the next descriptor in the block, except for the last descriptor which might point to a new block or might terminate the list.

Termination of the descriptor list is indicated by the Stop control bit. After a descriptor with the Stop control bit is observed, no further descriptor fetches are issued for that list. The Stop control bit can only be set on the last descriptor of a block.

Table 2-3: **Descriptor Format**

Offset	Fields			
0x0	Magic[15:0]	Rsv[1:0]	Nxt_adj[5:0]	Control[7:0]
0x04	4'h0, Len[27:0]			
0x08	Src_adr[31:0]			
0x0C	Src_adr[63:32]			
0x10	Dst_adr[31:0]			
0x14	Dst_adr[63:32]			
0x18	Nxt_adr[31:0]			
0x1C	Nxt_adr[63:32]			

Table 2-4: **Descriptor Fields**

Field	Bit Index	Sub Field	Description
Magic	15:0		16'had4b. Code to verify that the driver generated descriptor is valid.
Nxt_adj	5:0		The number of additional adjacent descriptors after the descriptor located at the next descriptor address field. A block of adjacent descriptors cannot cross a 4k boundary.
Control	5, 6, 7		Reserved
	4	EOP	End of packet for stream interface.
	2, 3		Reserved
	1	Completed	Set to 1 to interrupt after the engine has completed this descriptor. This requires global IE_DESCRIPTOR_COMPLETED control flag set in the SGDMA control register.
	0	Stop	Set to 1 to stop fetching descriptors for this descriptor list. The stop bit can only be set on the last descriptor of an adjacent block of descriptors.
Length	27:0		Length of the data in bytes.
Src_adr	63:0		Source address for H2C and memory mapped transfers. Metadata writeback address for C2H transfers.
Dst_adr	63:0		Destination address for C2H and memory mapped transfers. Not used for H2C stream.
Nxt_adr	63:0		Address of the next descriptor in the list.

Descriptor Bypass

The descriptor fetch engine can be bypassed on a per channel basis through Vivado IDE parameters. A channel with descriptor bypass enabled accepts descriptor from its respective `c2h_dsc_byp` or `h2c_dsc_byp` bus. Before the channel accepts descriptors, the run bit must be set. The NextDescriptorAddress and NextAdjacentCount, and Magic descriptor fields are not used when descriptors are bypassed. The descriptor Control.Stop bit does not prevent the user logic from writing additional descriptors. All descriptors written to the channel are processed, barring writing of new descriptors when the channel buffer is full.

Poll Mode

Each engine is capable of writing back completed descriptor counts to host memory. This allows the driver to post host memory to determine when the DMA is complete instead of waiting for an interrupt.

Completed descriptor count writebacks occur for any descriptor with the DscControl.Complete flag set when the DMA engine in the Control.Pollmode_Wb_Enable register is set. The completed descriptor count reported is the total number of completed descriptors since the DMA was initiated (not just those descriptors with the Completed flag set). The writeback address is defined by the Pollmode_hi_wb_addr and Pollmode_lo_wb_addr registers.

Table 2-5: **Completed Descriptor Count Writeback Format**

Offset	Fields		
0x0	Sts_err	7’h0	Compl_descriptor_count[23:0]

Table 2-6: **Completed Descriptor Count Writeback Fields**

Field	Description
Sts_err	The bitwise OR of any error status bits in the channel Status register.
Compl_descriptor_count[23:0]	The lower 24 bits of the Complete Descriptor Count register.

DMA H2C Stream

For host-to-card transfers, data is read from the host at the source address, but the destination address in the descriptor is unused. Packets can span multiple descriptors. The termination of a packet is indicated by the EOP control bit. A descriptor with an EOP bit asserts `tlast` on the AXI4-Stream user interface on the last beat of data.

DMA C2H Stream

For card-to-host transfers, the data is received from the AXI4-Stream interface and written to the destination address. Packets can span multiple descriptors. The C2H channel accepts data when it is enabled, and has valid descriptors. As data is received, it fills descriptors in order. When a descriptor is filled completely or closed due to an end of packet on the interface, the C2H channel writes back information to the writeback address on the host with new Magic, and updated EOP and Length as appropriate. For valid data cycles on the C2H AXI4-Stream interface, all data associated with a given packet must be contiguous. The DMA channel transfers the data beat ‘as is’ to the PCIe block.

Table 2-7: **C2H Stream Writeback Format**

Offset	Fields		
0x0	WB Magic[15:0]	Reserved [14:0]	Status[0]
0x04	Length[31:0]		

Table 2-8: **C2H Stream Writeback Fields**

Field	Bit Index	Sub Field	Description
Status	0	EOP	End of packet
Reserved	14:0		Reserved
WB Magic	15:0		16'h52b4. Code to verify the C2H writeback is valid.
Length	31:0		Length of the data in bytes.

Address Alignment

Table 2-9: **Address Alignment**

Interface Type	Datapath Width	Address Restriction
AXI4 MM	64	None
AXI4 MM	128	None
AXI4 MM	256	None
AXI4-Stream or AXI4 MM fixed address	64	Source_addr[2:0] == Destination_addr[2:0] == 3'h0
AXI4-Stream or AXI4 MM fixed address	128	Source_addr[3:0] == Destination_addr[3:0] == 4'h0
AXI4-Stream or AXI4 MM fixed address	256	Source_addr[4:0] == Destination_addr[4:0] == 5'h0

Length Granularity

Table 2-10: **Length Granularity**

Interface Type	Datapath Width	Length Granularity Restriction
AXI4 MM	64, 128, 256	None
AXI4-Stream or AXI4 MM fixed address	64	Length[2:0] == 3'h0
AXI4-Stream or AXI4 MM fixed address	128	Length[3:0] == 4'h0
AXI4-Stream or AXI4 MM fixed address	256	Length[4:0] == 5'h0

Standards

The DMA Subsystem for PCIe is compliant with the ARM® AMBA® AXI4 Protocol Specification and the PCI Express Base Specification v3.0 [Ref 1].

Performance and Resource Utilization

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

Port Descriptions

The DMA Subsystem for PCIe connects directly to the PCIe Integrated Block. The datapath interfaces to the PCIe Integrated Block IP are 64, 128 or 256-bits wide, and runs at up to 250 MHz depending on the configuration of the IP. The datapath width applies to all data interfaces except for the AXI4-Lite interfaces. AXI4-Lite interfaces are fixed at 32-bits wide.

Ports associated with this core are described in Tables 2-11 to 2-31.

Table 2-11: **Top-Level Interface Signals**

Signal Name	Direction	Description
sys_clk	Input	**Virtex-7**: PCIe reference clock. Should be driven from the O port of reference clock IBUFDS_GTE2. **UltraScale**: DRP clock and internal system clock (Half the frequency of sys_clk_gt). Should be driven by the ODIV2 port of reference clock IBUFDS_GTE3
sys_clk_gt	Input	**UltraScale only**: PCIe reference clock. Should be driven from the O port of reference clock IBUFDS_GTE3. See the *UltraScale Architecture Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156) [Ref 4].
sys_rst_n	Input	Reset from the PCIe edge connector reset signal
axi_aclk	Output	PCIe derived clock output for M_AXI and S_AXI interfaces
axi_aresetn	Output	AXI reset signal synchronous with the clock provided on the axi_aclk output. This reset should drive all corresponding AXI Interconnect aresetn signals.
user_lnk_up	Output	Output Active-High Identifies that the PCI Express core is linked up with a host device.

Table 2-12: **PCIe Interface Signals**

Signal Name	Direction	Description
pci_exp_rxp[PL_LINK_CAP_MAX_LINK_WIDTH-1:0]	Input	PCIe RX serial interface
pci_exp_rxn[PL_LINK_CAP_MAX_LINK_WIDTH-1:0]	Input	PCIe RX serial interface
pci_exp_txp[PL_LINK_CAP_MAX_LINK_WIDTH-1:0]	Output	PCIe TX serial interface
pci_exp_txn[PL_LINK_CAP_MAX_LINK_WIDTH-1:0]	Output	PCIe TX serial interface

Table 2-13: **H2C Channel 0-3 AXI4-Stream Interface Signals**

Signal Name	Direction	Description
m_axis_h2c_tready_*x*	Input	Assertion of this signal by the user logic indicates that it is ready to accept data. Data is transferred across the interface when m_axis_h2c_tready and m_axis_h2c_tvalid are asserted in the same cycle. If the user logic deasserts the signal when the valid signal is High, the DMA keeps the valid signal asserted until the ready signal is asserted.
m_axis_h2c_tlast_*x*	Output	The DMA asserts this signal in the last beat of the DMA packet to indicate the end of the packet.
m_axis_h2c_tdata_*x* [DAT_WIDTH-1:0]	Output	Transmit data from the DMA to the user logic.
m_axis_h2c_tvalid_*x*	Output	The DMA asserts this whenever it is driving valid data on m_axis_c2h_tdata.

Table 2-14: **C2H Channel 0-3 AXI4-Stream Interface Signals**

Signal Name	Direction	Description
s_axis_c2h_tready_*x*	Output	Assertion of this signal indicates that the DMA is ready to accept data. Data is transferred across the interface when s_axis_h2c_tready and s_axis_c2h_tvalid are asserted in the same cycle. If the DMA deasserts the signal when the valid signal is High, the user logic must keep the valid signal asserted until the ready signal is asserted.
s_axis_c2h_tlast_*x*	Input	The user logic asserts this signal to indicate the end of the DMA packet.
s_axis_c2h_tdata_*x*	Input	Transmits data from the user logic to the DMA.
s_axis_c2h_tvalid_*x*	Input	The user logic asserts this whenever it is driving valid data on s_axis_h2c_tdata.

Table 2-15: **AXI4 Memory Mapped Read Address Interface Signals**

Signal Name	Direction	Description
m_axi_araddr [AXI_ADR_WIDTH-1:0]	Output	This signal is the address for a memory mapped read to the user logic from the DMA.
m_axi_arid [ID_WIDTH-1:0]	Output	Standard AXI4 description, which is found in the AXI4 Protocol Specification [Ref 1].
m_axi_arlen[7:0]	Output	Master read burst length.
m_axi_arsize[2:0]	Output	Master read burst size.

Table 2-15: **AXI4 Memory Mapped Read Address Interface Signals *(Cont'd)***

Signal Name	Direction	Description
m_axi_arprot[2:0]	Output	3'h0
m_axi_arvalid	Output	The assertion of this signal means there is a valid read request to the address on m_axi_araddr.
m_axi_arready	Input	Master read address ready.
m_axi_arlock	Output	1'b0
m_axi_arcache[3:0]	Output	4'h0
m_axi_arburst	Output	Master read burst type.

Table 2-16: **AXI4 Memory Mapped Read Interface Signals**

Signal Name	Direction	Description
m_axi_rdata [DATA_WIDTH-1:0]	Input	Master read data.
m_axi_rid [ID_WIDTH-1:0]	Input	Master read ID.
m_axi_rresp[1:0]	Input	Master read response.
m_axi_rlast	Input	Master read last.
m_axi_rvalid	Input	Master read valid.
m_axi_rready	Output	Master read ready.

Table 2-17: **AXI4 Memory Mapped Write Address Interface Signals**

Signal Name	Direction	Description
m_axi_awaddr [AXI_ADR_WIDTH-1:0]	Output	This signal is the address for a memory mapped write to the user logic from the DMA.
m_axi_awid [ID_WIDTH-1:0]	Output	Master write address ID.
m_axi_awlen[7:0]	Output	Master write address length.
m_axi_awsize[2:0]	Output	Master write address size.
m_axi_awburst[1:0]	Output	Master write address burst type.
m_axi_awprot[2:0]	Output	3'h0
m_axi_awvalid	Output	The assertion of this signal means there is a valid write request to the address on m_axi_araddr.
m_axi_awready	Input	Master write address ready.
m_axi_awlock	Output	1'b0
m_axi_awcache[3:0]	Output	4'h0

Table 2-18: **AXI4 Memory Mapped Write Interface Signals**

Signal Name	Direction	Description
m_axi_wdata [DATA_WIDTH-1:0]	Output	Master write data.
m_axi_wlast	Output	Master write last.
m_axi_wstrb	Output	Master write strobe.
m_axi_wvalid	Output	Master write valid.
m_axi_wready	Input	Master write ready.

Table 2-19: **AXI4 Memory Mapped Write Response Interface Signals**

Signal Name	Direction	Description
m_axi_bvalid	Input	Master write response valid.
m_axi_bresp[1:0]	Input	Master write response.
m_axi_bid [ID_WIDTH-1:0]	Input	Master response ID.
m_axi_bready	Output	Master response ready.

Table 2-20: **AXI4 Memory Mapped Master Bypass Read Address Interface Signals**

Signal Name	Direction	Description
m_axib_araddr [AXI_ADR_WIDTH-1:0]	Output	This signal is the address for a memory mapped read to the user logic from the host.
m_axib_arid [ID_WIDTH-1:0]	Output	Master read address ID.
m_axib_arlen[7:0]	Output	Master read address length.
m_axib_arsize[2:0]	Output	Master read address size.
m_axib_arprot[2:0]	Output	3’h0
m_axib_arvalid	Output	The assertion of this signal means there is a valid read request to the address on m_axib_araddr.
m_axib_arready	Input	Master read address ready.
m_axib_arlock	Output	1’b0
m_axib_arcache[3:0]	Output	4’h0
m_axib_arburst	Output	Master read address burst type.

Table 2-21: **AXI4 Memory Mapped Master Bypass Read Interface Signals**

Signal Name	Direction	Description
m_axib_rdata [DATA_WIDTH-1:0]	Input	Master read data.
m_axib_rid [ID_WIDTH-1:0]	Input	Master read ID.
m_axib_rresp[1:0]	Input	Master read response.

Table 2-21: **AXI4 Memory Mapped Master Bypass Read Interface Signals *(Cont'd)***

Signal Name	Direction	Description
m_axib_rlast	Input	Master read last.
m_axib_rvalid	Input	Master read valid.
m_axib_rready	Output	Master read ready.

Table 2-22: **AXI4 Memory Mapped Master Bypass Write Address Interface Signals**

Signal Name	Direction	Description
m_axib_awaddr [AXI_ADR_WIDTH-1:0]	Output	This signal is the address for a memory mapped write to the user logic from the host.
m_axib_awid [ID_WIDTH-1:0]	Output	Master write address ID.
m_axib_awlen[7:0]	Output	Master write address length.
m_axib_awsize[2:0]	Output	Master write address size.
m_axib_awburst[1:0]	Output	Master write address burst type.
m_axib_awprot[2:0]	Output	3'h0
m_axib_awvalid	Output	The assertion of this signal means there is a valid write request to the address on m_axib_araddr.
m_axib_awready	Input	Master write address ready.
m_axib_awlock	Output	1'b0
m_axib_awcache[3:0]	Output	4'h0

Table 2-23: **AXI4 Memory Mapped Master Bypass Write Interface Signals**

Signal Name	Direction	Description
m_axib_wdata [DATA_WIDTH-1:0]	Output	Master write data.
m_axib_wlast	Output	Master write last.
m_axib_wstrb	Output	Master write strobe.
m_axib_wvalid	Output	Master write valid.
m_axib_wready	Input	Master write ready.

Table 2-24: **AXI4 Memory Mapped Master Bypass Write Response Interface Signals**

Signal Name	Direction	Description
m_axib_bvalid	Input	Master write response valid.
m_axib_bresp[1:0]	Input	Master write response.
m_axib_bid [ID_WIDTH-1:0]	Input	Master write response ID.
m_axib_bready	Output	Master response ready.

Table 2-25: **Config AXI4-Lite Memory Mapped Write Master Interface Signals**

Signal Name	Direction	Description
m_axil_awaddr[31:0]	Output	This signal is the address for a memory mapped write to the user logic from the host.
m_axil_awprot[2:0]	Output	3’h0
m_axil_awvalid	Output	The assertion of this signal means there is a valid write request to the address on m_axil_awaddr.
m_axil_awready	Input	Master write address ready.
m_axil_wdata[31:0]	Output	Master write data.
m_axil_wstrb	Output	Master write strobe.
m_axil_wvalid	Output	Master write valid.
m_axil_wready	Input	Master write ready.
m_axil_bvalid	Input	Master response valid.
m_axil_bready	Output	Master response valid.

Table 2-26: **Config AXI4-Lite Memory Mapped Read Master Interface Signals**

Signal Name	Direction	Description
m_axil_araddr[31:0]	Output	This signal is the address for a memory mapped read to the user logic from the host.
m_axil_arprot[2:0]	Output	3’h0
m_axil_arvalid	Output	The assertion of this signal means there is a valid read request to the address on m_axil_araddr.
m_axil_arready	Input	Master read address ready.
m_axil_rdata[31:0]	Input	Master read data.
m_axil_rresp	Input	Master read response.
m_axil_rvalid	Input	Master read valid.
m_axil_rready	Output	Master read ready.

Table 2-27: **Config AXI4-Lite Memory Mapped Write Slave Interface Signals**

Signal Name	Direction	Description
s_axil_awaddr[31:0]	Input	This signal is the address for a memory mapped write to the DMA from the user logic.
s_axil_awvalid	Input	The assertion of this signal means there is a valid write request to the address on s_axil_awaddr.
s_axil_awprot[2:0]	Input	Unused
s_axil_awready	Output	Slave write address ready.
s_axil_wdata[31:0]	Input	Slave write data.
s_axil_wstrb	Input	Slave write strobe.
s_axil_wvalid	Input	Slave write valid.
s_axil_wready	Output	Slave write ready.
s_axil_bvalid	Output	Slave write response valid.
s_axil_bready	Input	Save response ready.

Table 2-28: **Config AXI4-Lite Memory Mapped Read Slave Interface Signals**

Signal Name	Direction	Description
s_axil_araddr[31:0]	Input	This signal is the address for a memory mapped read to the DMA from the user logic.
s_axil_arprot[2:0]	Input	Unused
s_axil_arvalid	Input	The assertion of this signal means there is a valid read request to the address on s_axil_araddr.
s_axil_arready	Output	Slave read address ready.
s_axil_rdata[31:0]	Output	Slave read data.
s_axil_rresp	Output	Slave read response.
s_axil_rvalid	Output	Slave read valid.
s_axil_rready	Input	Slave read ready.

Table 2-29: **Interrupt Interface**

Signal Name	Direction	Description
usr_irq_req	Input	Assert to generate an interrupt. Maintain assertion until interrupt is serviced.
usr_irq_ack	Output	Indicates that the interrupt has been sent on PCIe. Two acks are generated for legacy interrupts. One ack is generated for MSI interrupts.

Table 2-30: **Channel 0-3 Status Ports**

Signal Name	Direction	Description
h2c_sts [7:0]	Output	Status bits for each channel. Bit: 6: Control.Run 5: IRQ event pending 4: Packet Done event (AXI4-Stream) 3: Descriptor Done event. Pulses for one cycle for each descriptor that is completed, regardless of the Descriptor.Completed field 2: Status register Descriptor_stop bit 1: Status register Descriptor_completed bit 0: Status register busy bit
c2h_sts [7:0]	Output	Status bits for each channel. Bit: 6: Control.Run 5: IRQ event pending 4: Packet Done event (AXI4-Stream) 3: Descriptor Done event. Pulses for one cycle for each descriptor that is completed, regardless of the Descriptor.Completed field 2: Status register Descriptor_stop bit 1: Status register Descriptor_completed bit 0: Status register busy bit

Descriptor Bypass Mode

If Descriptor Bypass for Read (H2C) or Descriptor Bypass for Write (C2H) are selected, these ports are present. Here is the instruction for selecting Descriptor bypass option.

In the PCIe: DMA Tab, select either **Descriptor Bypass for Read (H2C)** or **Descriptor Bypass for Write (C2H)**. Each binary bit correspond to channel. LSB correspond to Channel 0. Value 1 in bit positions means corresponding channel descriptor bypass enabled.

Table 2-31: **H2C 0-3 Descriptor Bypass Port**

Port	Direction	Description
h2c_dsc_byp_ready	Output	Channel is ready to accept new descriptors. After h2c_dsc_byp_ready is deasserted, one additional descriptor can be written. The Control.Run bit must be asserted before the channel accepts descriptors.
h2c_dsc_byp_load	Input	Write the descriptor presented at h2c_dsc_byp_data into the channel's descriptor buffer.
h2c_dsc_byp_src_addr	Input	Descriptor source address to be loaded
h2c _dsc_byp_dst_addr[63:0]	Input	Descriptor destination address to be loaded
h2c _dsc_byp_len[27:0]	Input	Descriptor length to be loaded
h2c _dsc_byp_ctl[15:0]	Input	Descriptor control to be loaded

Table 2-32: **C2H 0-3 Descriptor Bypass Ports**

Port	Direction	Description
c2h_dsc_byp_ready	Output	Channel is ready to accept new descriptors. After c2h_dsc_byp_ready is deasserted, one additional descriptor can be written. The Control.Run bit must be asserted before the channel accepts descriptors.
c2h_dsc_byp_load	Input	Descriptor presented at c2h_dsc_byp_data is valid.
c2h_dsc_byp_src_addr[63:0]	Input	Descriptor source address to be loaded.
c2h_dsc_byp_dst_addr[63:0]	Input	Descriptor destination address to be loaded.
c2h_dsc_byp_len[27:0]	Input	Descriptor length to be loaded.
c2h_dsc_byp_ctl[15:0]	Input	Descriptor control to be loaded.

Register Space

Configuration and status registers internal to the DMA Subsystem for PCIe and those in the user logic can be accessed from the host through mapping the read or write request to a Base Address Register (BAR). Based on the BAR hit, the request is routed to the appropriate location. For PCIe BAR assignments, see Table 2-1 and Table 2-2.

PCIe to AXI-Lite Master (BAR0) Address Map

Transactions that hit the PCIe to AXI-Lite Master are routed to the AXI4-Lite Memory Mapped user interface. This interface supports 32 bits of address space and 32-bit read and write requests. The PCIe to AXI-Lite Master address map is defined by the user logic.

PCIe to DMA (BAR1) Address Map

Transactions that hit the PCIe to DMA space are routed to the DMA Subsystem for the PCIe internal configuration register bus. This bus supports 32 bits of address space and 32-bit read and write requests.

DMA Subsystem for PCIe registers can be accessed from the host or from the AXI-Lite Slave interface. These registers should be used for programming the DMA and checking status.

PCIe to DMA Address Format

Table 2-33: **PCIe to DMA Address Format**

31:16	15:12	11:8	7:0
Reserved	Target	Channel	Byte Offset

Table 2-34: **PCIe to DMA Address Field Descriptions**

Bit Index	Field	Description
15:12	Target	The destination submodule within the DMA 4'h0: H2C Channels 4'h1: C2H Channels 4'h2: IRQ Block 4'h3: Config 4'h4: H2C SGDMA 4'h5: C2H SGDMA 4'h6: SGDMA Common 4'h8: MSI-X
11:8	Channel ID[3:0]	For H2C and C2H targets (0x0-0x1, 0x4-0x5), this field indicates which engine is being addressed. For all other targets this field must be 0.
7:0	Byte Offset	The byte address of the register to be accessed within the target. Bits[1:0] must be 0.

PCIe to DMA Configuration Registers

Table 2-35: **Configuration Register Attribute Definitions**

Attribute	Description
RV	Reserved
RW	Read/Write
RC	Clear on Read.
W1C	Write 1 to Clear
W1S	Write 1 to Set
RO	Read Only
WO	Write Only

Some registers can be accessed with different attributes. In such cases different register offsets are provided for each attribute. Undefined bits and address space is reserved.

In some registers, individual bits in a vector might represent a specific DMA engine. In such cases the LSBs of the vectors correspond to the H2C channel (if any). Channel ID 0 is in the LSB position. Bits representing the C2H channels are packed just above them.

H2C Channel Register Space (0x0)

The H2C channel register space is described in this section.

Table 2-36: **H2C Channel Register Space**

Address (hex)	Register Name
0x00	H2C Channel Identifier (0x00)
0x04	H2C Channel Control (0x04)
0x08	H2C Channel Control (0x08)
0x0C	H2C Channel Control (0x08)
0x40	H2C Channel Status (0x40)
0x44	H2C Channel Status (0x44)
0x48	H2C Channel Completed Descriptor Count (0x48)
0x4C	H2C Channel Alignments (0x4C)
0x88	H2C Poll Mode Low Write Back Address (0x88)
0x8C	H2C Poll Mode High Write Back Address (0x8C)
0x90	H2C Channel Interrupt Enable Mask (0x90)
0x94	H2C Channel Interrupt Enable Mask (0x94)
0x98	H2C Channel Interrupt Enable Mask (0x98)
0xC0	H2C Channel Performance Monitor Control (0xC0)

Table 2-36: **H2C Channel Register Space *(Cont'd)***

Address (hex)	Register Name
0xC4	H2C Channel Performance Cycle Count (0xC4)
0xC8	H2C Channel Performance Cycle Count (0xC8)
0xCC	H2C Channel Performance Data Count (0xCC)
0xD0	H2C Channel Performance Data Count (0xD0)

Table 2-37: **H2C Channel Identifier (0x00)**

Bit Index	Default Value	Access Type	Description
31:20	12'h1fc	RO	DMA Subsystem for PCIe identifier
19:16	4'h0	RO	H2C Channel Target
15	1'b0	RO	Stream 1: AXI4-Stream Interface 0: Memory Mapped AXI4 Interface
14:12	0	RO	Reserved
11:8	Varies	RO	Channel ID Target [3:0]
7:0	8'h03	RO	Version 8'h01: 2015.3 and 2015.4 8'h02: 2016.1 8'h03: 2016.2

Table 2-38: **H2C Channel Control (0x04)**

Bit Index	Default	Access Type	Description
26	0x0	RW	pollmode_wb_enable Poll mode writeback enable. When this bit is set the DMA writes back the completed descriptor count when a descriptor with the Completed bit set, is completed.
25	1'b0	RW	non_inc_mode Non-incrementing address mode. Applies to m_axi_araddr interface only.
23:19	5'h0	RW	ie_desc_error Set to all 1s (0x1F) to enable logging of Status.Desc_error and to stop the engine if the error is detected.
18:14	5'h0	RW	ie_write_error Set to all 1s (0x1F) to enable logging of Status.Write_error and to stop the engine if the error is detected.
13:9	5'h0	RW	ie_read_error Set to all 1s (0x1F) to enable logging of Status.Read_error and to stop the engine if the error is detected.
6	1'b0	RW	ie_idle_stopped Set to 1 to enable logging of Status.Idle_stopped

Table 2-38: **H2C Channel Control (0x04) *(Cont'd)***

Bit Index	Default	Access Type	Description
5	1'b0	RW	ie_invalid_length Set to 1 to enable logging of Status.Invalid_length
4	1'b0	RW	ie_magic_stopped Set to 1 to enable logging of Status.Magic_stopped
3	1'b0	RW	ie_align_mismatch Set to 1 to enable logging of Status.Align_mismatch
2	1'b0	RW	ie_descriptor_completed Set to 1 to enable logging of Status.Descriptor_completed
1	1'b0	RW	ie_descriptor_stopped Set to 1 to enable logging of Status.Descriptor_stopped
0	1'b0	RW	Run Set to 1 to start the SGDMA engine. Reset to 0 to stop it; if it is busy it completes the current descriptor.

Table 2-39: **H2C Channel Control (0x08)**

Bit Index	Default	Access Type	Description
26:0		W1S	Control Bit descriptions are the same as in Table 2-38.

Table 2-40: **H2C Channel Control (0x0C)**

Bit Index	Default	Access Type	Description
26:0		W1C	Control Bit descriptions are the same as in Table 2-38.

Table 2-41: **H2C Channel Status (0x40)**

Bit Index	Default	Access Type	Description
23:19	5'h0	RW1C	descr_error[4:0] Reset (0) on setting Control.Run.Bit position 4: Unexpected completion 3: Header EP 2: Parity error 1: Completer abort 0: Unsupported request
18:14	5'h0	RW1C	write_error[4:0] Reset (0) on setting Control.Run. Bit position: 4-2: Reserved 1: Slave error 0: Decode error

Table 2-41: **H2C Channel Status (0x40) *(Cont'd)***

Bit Index	Default	Access Type	Description
13:9	5'h0	RW1C	read_error[4:0] Reset (0) on setting Control.Run. Bit position 4: Unexpected completion 3: Header EP 2: Parity error 1: Completer abort 0: Unsupported request
6	1'b0	RW1C	idle_stopped Reset (0) on setting Control.Run. Set when the engine is idle after resetting Control.Run if Control.IE_idle_stopped is set.
5	1'b0	RW1C	invalid_length Reset on setting Control.Run. Set when the descriptor length is not a multiple of the data width of an AXI4-Stream channel and Control.ie_invalid_length is set.
4	1'b0	RW1C	magic_stopped Reset on setting Control.Run. Set when the engine encounters a descriptor with invalid magic and stopped if Control.IE_Magic_stopped is set.
3	1'b0	RW1C	align_mismatch Source and destination address on descriptor are not properly aligned to each other.
2	1'b0	RW1C	descriptor_completed Reset on setting Control.Run. Set after the engine has completed a descriptor with the COMPLETE bit set if Control.IE_Descriptor_completed is set.
1	1'b0	RW1C	descriptor_stopped Reset on setting Control.Run. Set after the engine completed a descriptor with the STOP bit set if Control.IE_Descriptor_stopped is set.
0	1'b0	RO	Busy Set if the SGDMA engine is busy. Zero when it is idle.

Table 2-42: **H2C Channel Status (0x44)**

Bit Index	Default	Access Type	Description
23:1		RC	Status Clear on Read. Bit description is the same as in Table 2-41. Bit 0 cannot be cleared.

Table 2-43: **H2C Channel Completed Descriptor Count (0x48)**

Bit Index	Default	Access Type	Description
31:0	32'h0	RO	compl_descriptor_count The number of competed descriptors update by the engine after completing each descriptor in the list. Reset to 0 on rising edge of Control register, run bit.

Table 2-44: **H2C Channel Alignments (0x4C)**

Bit Index	Default	Access Type	Description
23:16	Configuration based	RO	addr_alignment The byte alignment that the source and destination addresses must align to. This value is dependent on configuration parameters.
15:8	Configuration based	RO	len_granularity The minimum granularity of DMA transfers in bytes.
7:0	Configuration based	RO	address_bits The number of address bits configured.

Table 2-45: **H2C Poll Mode Low Write Back Address (0x88)**

Bit Index	Default	Access Type	Description
31:0	0x0	RW	Pollmode_lo_wb_addr[31:0] Lower 32 bits of the poll mode writeback address.

Table 2-46: **H2C Poll Mode High Write Back Address (0x8C)**

Bit Index	Default	Access Type	Description
31:0	0x0	RW	Pollmode_hi_wb_addr[63:32] Upper 32 bits of the poll mode writeback address.

Table 2-47: **H2C Channel Interrupt Enable Mask (0x90)**

Bit Index	Default	Access Type	Description
23:19	5'h0	RW	im_desc_error[4:0] Set to 1 to interrupt when corresponding status register read_error bit is logged.
18:14	5'h0	RW	im_write_error[4:0] set to 1 to interrupt when corresponding status register write_error bit is logged.
13:9	5'h0	RW	im_read_error[4:0] set to 1 to interrupt when corresponding status register read_error bit is logged.

Table 2-47: **H2C Channel Interrupt Enable Mask (0x90)** ***(Cont'd)***

Bit Index	Default	Access Type	Description
6	1'b0	RW	im_idle_stopped Set to 1 to interrupt when the status register idle_stopped bit is logged.
5	1'b0	RW	im_invalid_length Set to 1 to interrupt when status register align_mismatch bit is logged.
4	1'b0	RW	im_magic_stopped set to 1 to interrupt when status register magic_stopped bit is logged.
3	1'b0	RW	im_align_mismatch set to 1 to interrupt when status register align_mismatch bit is logged.
2	1'b0	RW	im_descriptor_completd set to 1 to interrupt when status register descriptor_completed bit is logged.
1	1'b0	RW	im_descriptor_stopped set to 1 to interrupt when status register descriptor_stopped bit is logged.

Table 2-48: **H2C Channel Interrupt Enable Mask (0x94)**

Bit Index	Default	Access Type	Description
		W1S	Interrupt Enable Mask

Table 2-49: **H2C Channel Interrupt Enable Mask (0x98)**

Bit Index	Default	Access Type	Description
		W1C	Interrupt Enable Mask

Table 2-50: **H2C Channel Performance Monitor Control (0xC0)**

Bit Index	Default	Access Type	Description
2	1'b0	RW	Run Set to 1 to arm performance counters. Counter starts after the Control.Run (Byte offset 0x1) bit is set. Set to 0 to halt performance counters.
1	1'b0	WO	Clear Write 1 to clear performance counters.
0	1'b0	RW	Auto Automatically stop performance counters when a descriptor with the stop bit is completed. Automatically clear performance counters when the Control.Run (Byte offset 0x1) is set. Writing 1 to PerformanceControl.Run is still required to start the counters.

Table 2-51: **H2C Channel Performance Cycle Count (0xC8)**

Bit Index	Default	Access Type	Description
16	1'b0	RO	pmon_cyc_count_maxed Cycle count maximum was hit.
9:0	10'h0	RO	pmon_cyc_count [41:32] Increments once per clock while running. See PerformanceControl.Clear and PerformanceControl.Auto for clearing.

Table 2-52: **H2C Channel Performance Cycle Count (0xC4)**

Bit Index	Default	Access Type	Description
31:0	32'h0	RO	pmon_cyc_count[31:0] Increments once per clock while running. See PerformanceControl.Clear and PerformanceControl.Auto for clearing.

Table 2-53: **H2C Channel Performance Data Count (0xD0)**

Bit Index	Default	Access Type	Description
16	1'b0	RO	pmon_dat_count_maxed Data count maximum was hit
9:0	10'h0	RO	pmon_dat_count [41:32] Increments for each valid read data beat while running. See PerformanceControl.Clear and PerformanceControl.Auto for clearing.

Table 2-54: **H2C Channel Performance Data Count (0xCC)**

Bit Index	Default	Access Type	Description
31:0	32'h0	RO	pmon_dat_count[31:0] Increments for each valid read data beat while running. See PerformanceControl.Clear and PerformanceControl.Auto for clearing.

C2H Channel Registers (0x1)

The C2H channel register space is described in this section.

Table 2-55: **C2H Channel Register Space**

Address (hex)	Register Name
0x00	C2H Channel Identifier (0x00)
0x04	C2H Channel Control (0x04)
0x08	C2H Channel Control (0x08)
0x0C	C2H Channel Control (0x0C)

Table 2-55: **C2H Channel Register Space *(Cont'd)***

Address (hex)	Register Name
0x40 0x44	C2H Channel Status (0x40) C2H Channel Status (0x44)
0x48	C2H Channel Completed Descriptor Count (0x48)
0x4C	C2H Channel Alignments (0x4C)
0x88	C2H Poll Mode Low Write Back Address (0x88)
0x8C	C2H Poll Mode High Write Back Address (0x8C)
0x90 0x94 0x98	C2H Channel Interrupt Enable Mask (0x90) C2H Channel Interrupt Enable Mask (0x94) C2H Channel Interrupt Enable Mask (0x98)
0xC0	C2H Channel Performance Monitor Control (0xC0)
0xC4 0xC8	C2H Channel Performance Cycle Count (0xC4) C2H Channel Performance Cycle Count (0xC8)
0xCC 0xD0	C2H Channel Performance Data Count (0xCC) C2H Channel Performance Data Count (0xD0)

Table 2-56: **C2H Channel Identifier (0x00)**

Bit Index	Default	Access Type	Description
31:20	12'h1fc	RO	DMA Subsystem for PCIe identifier
19:16	4'h1	RO	C2H Channel Target
15	1'b0	RO	Stream 1: AXI4-Stream Interface 0: Memory Mapped AXI4 Interface
14:12	0	RO	Reserved
11:8	Varies	RO	Channel ID Target [3:0]
7:0	8'h03	RO	Version 8'h01: 2015.3 and 2015.4 8'h02: 2016.1 8'h03: 2016.2

Table 2-57: **C2H Channel Control (0x04)**

Bit Index	Default	Access Type	Description
27	0x0	RW	Disables the metadata writeback for C2H AXI4-Stream. No effect if the channel is configured to use AXI Memory Mapped.
26	0x0	RW	pollmode_wb_enable Poll mode writeback enable. When this bit is set, the DMA writes back the completed descriptor count when a descriptor with the Completed bit set, is completed.

Table 2-57: **C2H Channel Control (0x04) *(Cont'd)***

Bit Index	Default	Access Type	Description
25	1'b0	RW	non_inc_mode Non-incrementing address mode. Applies to m_axi_araddr interface only.
23:19	5'h0	RW	ie_desc_error Set to all 1s (0x1F) to enable logging of Status.Desc_error and to stop the engine if the error is detected.
13:9	5'h0	RW	ie_read_error Set to all 1s (0x1F) to enable logging of Status.Read_error and to stop the engine if the error is detected
6	1'b0	RW	ie_idle_stopped Set to 1 to enable logging of Status.Idle_stopped
5	1'b0	RW	ie_invalid_length Set to 1 to enable logging of Status.Invalid_length
4	1'b0	RW	ie_magic_stopped Set to 1 to enable logging of Status.Magic_stopped
3	1'b0	RW	ie_align_mismatch Set to 1 to enable logging of Status.Align_mismatch
2	1'b0	RW	ie_descriptor_completed Set to 1 to enable logging of Status.Descriptor_completed
1	1'b0	RW	ie_descriptor_stopped Set to 1 to enable logging of Status.Descriptor_stopped
0	1'b0	RW	Run Set to 1 to start the SGDMA engine. Reset to 0 to stop it; if it is busy it completes the current descriptor.

Table 2-58: **C2H Channel Control (0x08)**

Bit Index	Default	Access Type	Description
		W1S	Control Bit descriptions are the same as in Table 2-57.

Table 2-59: **C2H Channel Control (0x0C)**

Bit Index	Default	Access Type	Description
		W1C	Control Bit descriptions are the same as in Table 2-57.

Table 2-60: **C2H Channel Status (0x40)**

Bit Index	Default	Access Type	Description
23:19	5'h0	RW1C	descr_error[4:0] Reset (0) on setting Control.Run. Bit position 4:Unexpected completion 3: Header EP 2: Parity error 1: Completer abort 0: Unsupported request
13:9	5'h0	RW1C	read_error[4:0] Reset (0) on setting Control.Run. Bit position: 4-2: Reserved 1: Slave error 0: Decode error
6	1'b0	RW1C	idle_stopped Reset (0) on setting Control.Run. Set when the engine is idle after resetting Control.Run if Control.IE_idle_stopped is set.
5	1'b0	RW1C	invalid_length Reset on setting Control.Run. Set when the descriptor length is not a multiple of the data width of an AXI4-Stream channel and Control.ie_invalid_length is set.
4	1'b0	RW1C	magic_stopped Reset on setting Control.Run. Set when the engine encounters a descriptor with invalid magic and stopped if Control.IE_Magic_stopped is set.
3	13'b0	RW1C	align_mismatch Source and destination address on descriptor are not properly aligned to each other.
2	1'b0	RW1C	descriptor_completed Reset on setting Control.Run. Set after the engine has completed a descriptor with the COMPLETE bit set if Control.IE_Descriptor_completed is set.
1	1'b0	RW1C	descriptor_stopped Reset on setting Control.Run. Set after the engine completed a descriptor with the STOP bit set if Control.IE_Descriptor_stopped is set.
0	1'b0	RO	Busy Set if the SGDMA engine is busy. Zero when it is idle.

Table 2-61: **C2H Channel Status (0x44)**

Bit Index	Default	Access Type	Description
23:1		RC	Status Bit descriptions are the same as in Table 2-60.

Table 2-62: **C2H Channel Completed Descriptor Count (0x48)**

Bit Index	Default	Access Type	Description
31:0	32’h0	RO	compl_descriptor_count The number of competed descriptors update by the engine after completing each descriptor in the list. Reset to 0 on rising edge of Control register, run bit.

Table 2-63: **C2H Channel Alignments (0x4C)**

Bit Index	Default	Access Type	Description
23:16	varies	RO	addr_alignment The byte alignment that the source and destination addresses must align to. This value is dependent on configuration parameters.
15:8	Varies	RO	len_granularity The minimum granularity of DMA transfers in bytes.
7:0	ADDR_BITS	RO	address_bits The number of address bits configured.

Table 2-64: **C2H Poll Mode Low Write Back Address (0x88)**

Bit Index	Default	Access Type	Description
31:0	0x0	RW	Pollmode_lo_wb_addr[31:0] Lower 32 bits of the poll mode writeback address.

Table 2-65: **C2H Poll Mode High Write Back Address (0x8C)**

Bit Index	Default	Access Type	Description
31:0	0x0	RW	Pollmode_hi_wb_addr[63:32] Upper 32 bits of the poll mode writeback address.

Table 2-66: **C2H Channel Interrupt Enable Mask (0x90)**

Bit Index	Default	Access Type	Description
23:19	5’h0	RW	im_desc_error[4:0] set to 1 to interrupt when corresponding Status.Read_Error is logged.
13:9	5’h0	RW	im_read_error[4:0] set to 1 to interrupt when corresponding Status.Read_Error is logged.
6	1’b0	RW	im_idle_stopped set to 1 to interrupt when the Status.Idle_stopped is logged.
4	1’b0	RW	im_magic_stopped set to 1 to interrupt when Status.Magic_stopped is logged.

Table 2-66: **C2H Channel Interrupt Enable Mask (0x90)** ***(Cont'd)***

Bit Index	Default	Access Type	Description
2	1'b0	RW	im_descriptor_completd set to 1 to interrupt when Status.Descriptor_completed is logged.
1	1'b0	RW	im_descriptor_stopped set to 1 to interrupt when Status.Descriptor_stopped is logged.

Table 2-67: **C2H Channel Interrupt Enable Mask (0x94)**

Bit Index	Default	Access Type	Description
		W1S	Interrupt Enable Mask Bit descriptions are the same as in Table 2-66.

Table 2-68: **C2H Channel Interrupt Enable Mask (0x98)**

Bit Index	Default	Access Type	Description
		W1C	Interrupt Enable Mask Bit Descriptions are the same as in Table 2-66.

Table 2-69: **C2H Channel Performance Monitor Control (0xC0)**

Bit Index	Default	Access Type	Description
2	1'b0	RW	Run Set to 1 to arm performance counters. Counter starts after the Control.Run (Byte offset 0x1) bit is set. Set to 0 to halt performance counters.
1	1'b0	WO	Clear Write 1 to clear performance counters.
0	1'b0	RW	Auto Automatically stop performance counters when a descriptor with the stop bit is completed. Automatically clear performance counters when the Control.Run (Byte offset 0x1) is set. Writing 1 to PerformanceControl.Run is still required to start the counters.

Table 2-70: **C2H Channel Performance Cycle Count (0xC4)**

Bit Index	Default	Access Type	Description
31:0	32'h0	RO	pmon_cyc_count[31:0] Increments once per clock while running. See PerformanceControl.Clear and PerformanceControl.Auto for clearing.

Table 2-71: **C2H Channel Performance Cycle Count (0xC8)**

Bit Index	Default	Access Type	Description
16	1'b0	RO	pmon_cyc_count_maxed Cycle count maximum was hit.
9:0	10'h0	RO	pmon_cyc_count [41:32] Increments once per clock while running. See PerformanceControl.Clear and PerformanceControl.Auto for clearing.

Table 2-72: **C2H Channel Performance Data Count (0xCC)**

Bit Index	Default	Access Type	Description
31:0	32'h0	RO	pmon_dat_count[31:0] Increments for each valid read data beat while running. See PerformanceControl.Clear and PerformanceControl.Auto for clearing.

Table 2-73: **C2H Channel Performance Data Count (0xD0)**

Bit Index	Default	Access Type	Description
16	1'b0	RO	pmon_dat_count_maxed Data count maximum was hit
9:0	10'h0	RO	pmon_dat_count [41:32] Increments for each valid read data beat while running. See PerformanceControl.Clear and PerformanceControl.Auto for clearing.

IRQ Block Registers (0x2)

The IRQ Block registers are described in this section.

Table 2-74: **IRQ Block Register Space**

Address (hex)	Register Name
0x00	IRQ Block Identifier (0x00)
0x04	IRQ Block User Interrupt Enable Mask (0x04)
0x08	IRQ Block User Interrupt Enable Mask (0x08)
0x0C	IRQ Block User Interrupt Enable Mask (0x0C)
0x10	IRQ Block Channel Interrupt Enable Mask (0x10)
0x14	IRQ Block Channel Interrupt Enable Mask (0x14)
0x18	IRQ Block Channel Interrupt Enable Mask (0x18)
0x40	IRQ Block User Interrupt Request (0x40)
0x44	IRQ Block Channel Interrupt Request (0x44)
0x48	IRQ Block User Interrupt Pending (0x48)
0x4C	IRQ Block Interrupt Pending (0x4C)

Table 2-74: **IRQ Block Register Space *(Cont'd)***

Address (hex)	Register Name
0x80	IRQ Block User Vector Number (0x80)
0x84	IRQ Block User Vector Number (0x84)
0x88	IRQ Block User Vector Number (0x88)
0x8C	IRQ Block User Vector Number (0x8C)
0xA0	IRQ Block Channel Vector Number (0xA0)
0xA4	IRQ Block Channel Vector Number (0xA4)

Table 2-75: **IRQ Block Identifier (0x00)**

Bit Index	Default	Access Type	Description
31:20	12'h1fc	RO	DMA Subsystem for PCIe identifier
19:16	4'h2	RO	IRQ Identifier
15:8	8'h0	RO	Reserved
7:0	8'h03	RO	Version 8'h01: 2015.3 and 2015.4 8'h02: 2016.1 8'h03: 2016.2

Table 2-76: **IRQ Block User Interrupt Enable Mask (0x04)**

Bit Index	Default	Access Type	Description
[NUM_USR_INT-1:0]	'h0	RW	user_int_enmask User Interrupt Enable Mask

Table 2-77: **IRQ Block User Interrupt Enable Mask (0x08)**

Bit Index	Default	Access Type	Description
		W1S	user_int_enmask Bit descriptions are the same as in Table 2-76.

Table 2-78: **IRQ Block User Interrupt Enable Mask (0x0C)**

Bit Index	Default	Access Type	Description
		W1C	user_int_enmask Bit descriptions are the same as in Table 2-76.

Table 2-79: **IRQ Block Channel Interrupt Enable Mask (0x10)**

Bit Index	Default	Access Type	Description
[NUM_CHNL-1:0]	'h0	RW	channel_int_enmask Engine Interrupt Enable Mask. One bit per read or write engine. 0: Prevents an interrupt from being generated when interrupt source is asserted. The position of the H2C bits always starts at bit 0. The position of the C2H bits is the index above the last H2C index, and therefore depends on the NUM_H2C_CHNL parameter. 1: Generates an interrupt on the rising edge of the interrupt source. If the enmask bit is set and the source is already set, an interrupt is also be generated.

Table 2-80: **IRQ Block Channel Interrupt Enable Mask (0x14)**

Bit Index	Default	Access Type	Description
		W1S	channel_int_enmask Bit descriptions are the same as in Table 2-79.

Table 2-81: **IRQ Block Channel Interrupt Enable Mask (0x18)**

Bit Index	Default	Access Type	Description
		W1C	channel_int_enmask Bit descriptions are the same as in Table 2-79.

Figure 2-4 shows the packing of H2C and C2H bits.

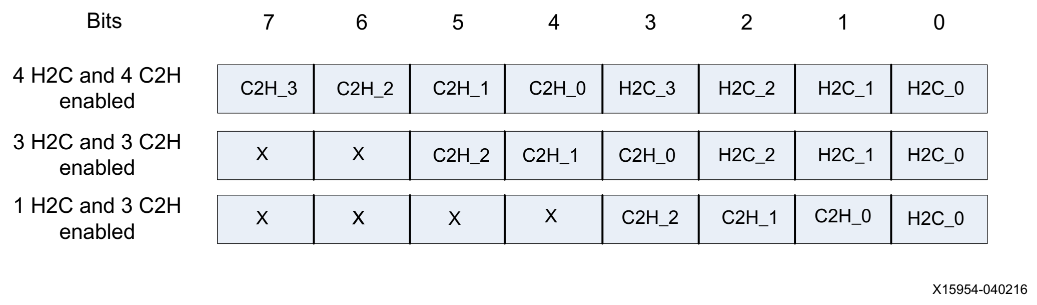

Bits	7	6	5	4	3	2	1	0
4 H2C and 4 C2H enabled	C2H_3	C2H_2	C2H_1	C2H_0	H2C_3	H2C_2	H2C_1	H2C_0
3 H2C and 3 C2H enabled	X	X	C2H_2	C2H_1	C2H_0	H2C_2	H2C_1	H2C_0
1 H2C and 3 C2H enabled	X	X	X	X	C2H_2	C2H_1	C2H_0	H2C_0

Figure 2-4: **Packing H2C and C2H**

Table 2-82: **IRQ Block User Interrupt Request (0x40)**

Bit Index	Default	Access Type	Description
[NUM_USR_INT-1:0]	'h0	RO	user_int_req User Interrupt Request This register reflects the interrupt source AND'd with the enable mask register.

Table 2-83: **IRQ Block Channel Interrupt Request (0x44)**

Bit Index	Default	Access Type	Description
[NUM_CHNL-1:0]	'h0	RO	engine_int_req Engine Interrupt Request. One bit per read or write engine. This register reflects the interrupt source AND with the enable mask register. The position of the H2C bits always starts at bit 0. The position of the C2H bits is the index above the last H2C index, and therefore depends on the NUM_H2C_CHNL parameter. Figure 2-4 shows the packing of H2C and C2H bits.

Table 2-84: **IRQ Block User Interrupt Pending (0x48)**

Bit Index	Default	Access Type	Description
[NUM_USR_INT-1:0]	'h0	RO	user_int_pend User Interrupt Pending. This register indicates pending events. The pending events are cleared by removing the event cause condition at the source component.

Table 2-85: **IRQ Block Interrupt Pending (0x4C)**

Bit Index	Default	Access Type	Description
[NUM_CHNL-1:0]	'h0	RO	engine_int_pend Engine Interrupt Pending. One bit per read or write engine. This register indicates pending events. The pending events are cleared by removing the event cause condition at the source component. The position of the H2C bits always starts at bit 0. The position of the C2H bits is the index above the last H2C index, and therefore depends on the NUM_H2C_CHNL parameter. Figure 2-4 shows the packing of H2C and C2H bits.

Table 2-86: **IRQ Block User Vector Number (0x80)**

Bit Index	Default	Access Type	Description
If MSI is enabled, this register specifies the MSI or MSI-X vector number of the MSI. In Legacy interrupts only the two LSBs of each field should be used to map to INTA, B, C, or D.			
28:24	5'h0	RW	vector 3 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[3].
20:16	5'h0	RW	vector 2 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[2].

Table 2-86: **IRQ Block User Vector Number (0x80) *(Cont'd)***

Bit Index	Default	Access Type	Description
12:8	5'h0	RW	vector 1 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[1].
4:0	5'h0	RW	vector 0 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[0].

Table 2-87: **IRQ Block User Vector Number (0x84)**

Bit Index	Default	Access Type	Description
If MSI is enabled, this register specifies the MSI or MSI-X vector number of the MSI. In Legacy interrupts only the 2 LSB of each field should be used to map to INTA, B, C, or D.			
28:24	5'h0	RW	vector 7 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[7].
20:16	5'h0	RW	vector 6 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[6].
12:8	5'h0	RW	vector 5 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[5].
4:0	5'h0	RW	vector 4 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[4].

Table 2-88: **IRQ Block User Vector Number (0x88)**

Bit Index	Default	Access Type	Description
If MSI is enabled, this register specifies the MSI or MSI-X vector number of the MSI. In Legacy interrupts only the 2 LSB of each field should be used to map to INTA, B, C, or D.			
28:24	5'h0	RW	vector 11 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[11].
20:16	5'h0	RW	vector 10 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[10].

Table 2-88: **IRQ Block User Vector Number (0x88) *(Cont'd)***

Bit Index	Default	Access Type	Description
12:8	5'h0	RW	vector 9 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[9].
4:0	5'h0	RW	vector 8 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[8].

Table 2-89: **IRQ Block User Vector Number (0x8C)**

Bit Index	Default	Access Type	Description
If MSI is enabled, this register specifies the MSI or MSI-X vector number of the MSI. In Legacy interrupts only the 2 LSB of each field should be used to map to INTA, B, C, or D.			
28:24	5'h0	RW	vector 15 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[15].
20:16	5'h0	RW	vector 14 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[14].
12:8	5'h0	RW	vector 13 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[13].
4:0	5'h0	RW	vector 12 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[12].

Table 2-90: **IRQ Block Channel Vector Number (0xA0)**

Bit Index	Default	Access Type	Description
If MSI is enabled, this register specifies the MSI vector number of the MSI. In Legacy interrupts, only the 2 LSB of each field should be used to map to INTA, B, C, or D. Similar to the other C2H/H2C bit packing clarification, see Table 2-4. The first C2H vector is the after the last H2C vector. For example, if NUM_H2C_Channel = 1, then H2C0 vector is 0xA0, bits [4:0], and C2H Channel 0 vector is at 0xA0, bits [12:8]. If NUM_H2C_Channel = 4, then H2C3 vector is at 0xA0 28:24, and C2H Channel 0 vector is at 0xA0, bits [4:0].			
28:24	5'h0	RW	vector3 The vector number that is used when an interrupt is generated by channel 3.
20:16	5'h0	RW	vector2 The vector number that is used when an interrupt is generated by channel 2.

Table 2-90: **IRQ Block Channel Vector Number (0xA0)** ***(Cont'd)***

Bit Index	Default	Access Type	Description
12:8	5'h0	RW	vector1 The vector number that is used when an interrupt is generated by channel 1.
4:0	5'h0	RW	vector0 The vector number that is used when an interrupt is generated by channel 0.

Table 2-91: **IRQ Block Channel Vector Number (0xA4)**

Bit Index	Default	Access Type	Description
If MSI is enabled, this register specifies the MSI vector number of the MSI. In Legacy interrupts, only the 2 LSB of each field should be used to map to INTA, B, C, or D. Similar to the other C2H/H2C bit packing clarification, see Table 2-4. The first C2H vector is the after the last H2C vector. For example, if NUM_H2C_Channel = 1, then H2C0 vector is 0xA4, bits [4:0], and C2H Channel 0 vector is at 0xA4, bits [12:8]. If NUM_H2C_Channel = 4, then H2C3 vector is at 0xA4 28:24, and C2H Channel 0 vector is at 0xA4, bits [4:0]			
28:24	5'h0	RW	vector7 The vector number that is used when an interrupt is generated by channel 7.
20:16	5'h0	RW	vector6 The vector number that is used when an interrupt is generated by channel 6.
12:8	5'h0	RW	vector5 The vector number that is used when an interrupt is generated by channel 5.
4:0	5'h0	RW	vector4 The vector number that is used when an interrupt is generated by channel 4.

Config Block Registers (0x3)

The Config Block registers are described in this section.

Table 2-92: **Config Block Register Space**

Address (hex)	Register Name
0x00	Config Block Identifier (0x00)
0x04	Config Block BusDev (0x04)
0x08	Config Block PCIE Max Payload Size (0x08)
0x0C	Config Block PCIE Max Read Request Size (0x0C)
0x10	Config Block System ID (0x10)
0x14	Config Block MSI Enable (0x14)
0x18	Config Block PCIE Data Width (0x18)
0x1C	Config PCIE Control (0x1C)

Table 2-92: **Config Block Register Space *(Cont'd)***

Address (hex)	Register Name
0x40	Config AXI User Max Payload Size (0x40)
0x44	Config AXI User Max Read Request Size (0x44)
0x60	Config Write Flush Timeout (0x60)

Table 2-93: **Config Block Identifier (0x00)**

Bit Index	Default	Access Type	Description
31:20	12'h1fc	RO	DMA Subsystem for PCIe identifier
19:16	4'h3	RO	Config Identifier
15:8	8'h0	RO	Reserved
7:0	8'h03	RO	Version 8'h01: 2015.3 and 2015.4 8'h02: 2016.1 8'h03: 2016.2

Table 2-94: **Config Block BusDev (0x04)**

Bit Index	Default	Access Type	Description
[15:0]	PCIe IP	RO	bus_dev Bus, device, and function

Table 2-95: **Config Block PCIE Max Payload Size (0x08)**

Bit Index	Default	Access Type	Description
[2:0]	PCIe IP	RO	pcie_max_payload Maximum write payload size. This is the lesser of the PCIe IP MPS and DMA Subsystem for PCIe parameters. 3'b000: 128 bytes 3'b001: 256 bytes 3'b010: 512 bytes 3'b011: 1024 bytes 3'b100: 2048 bytes 3'b101: 4096 bytes

Table 2-96: **Config Block PCIE Max Read Request Size (0x0C)**

Bit Index	Default	Access Type	Description
[2:0]	PCIe IP	RO	pcie_max_read Maximum read request size. This is the lesser of the PCIe IP MRRS and DMA Subsystem for PCIe parameters. 3'b000: 128 bytes 3'b001: 256 bytes 3'b010: 512 bytes 3'b011: 1024 bytes 3'b100: 2048 bytes 3'b101: 4096 bytes

Table 2-97: **Config Block System ID (0x10)**

Bit Index	Default	Access Type	Description
[15:0]	16'hff01	RO	system_id DMA Subsystem for PCIe system ID

Table 2-98: **Config Block MSI Enable (0x14)**

Bit Index	Default	Access Type	Description
[0]	PCIe IP	RO	MSI_en MSI Enable
[1]	PCIe IP	RO	MSI-X Enable

Table 2-99: **Config Block PCIE Data Width (0x18)**

Bit Index	Default	Access Type	Description
[2:0]	C_DAT_WIDTH	RO	pcie_width PCIe AXI4-Stream Width 0: 64 bits 1: 128 bits 2: 256 bits

Table 2-100: **Config PCIE Control (0x1C)**

Bit Index	Default	Access Type	Description
[0]	1'b1	RW	Relaxed Ordering PCIe read request TLPs are generated with the relaxed ordering bit set.

Table 2-101: **Config AXI User Max Payload Size (0x40)**

Bit Index	Default	Access Type	Description
6:4	3'h5	RO	user_eff_payload The actual maximum payload size issued to the user application. This value might be lower than user_prg_payload due to IP configuration or datapath width. 3'b000: 128 bytes 3'b001: 256 bytes 3'b010: 512 bytes 3'b011: 1024 bytes 3'b100: 2048 bytes 3'b101: 4096 bytes
2:0	3'h5	RW	user_prg_payload The programmed maximum payload size issued to the user application. 3'b000: 128 bytes 3'b001: 256 bytes 3'b010: 512 bytes 3'b011: 1024 bytes 3'b100: 2048 bytes 3'b101: 4096 bytes

Table 2-102: **Config AXI User Max Read Request Size (0x44)**

Bit Index	Default	Access Type	Description
6:4	3'h5	RO	user_eff_read Maximum read request size issued to the user application. This value may be lower than user_max_read due to PCIe configuration or datapath width. 3'b000: 128 bytes 3'b001: 256 bytes 3'b010: 512 bytes 3'b011: 1024 bytes 3'b100: 2048 bytes 3'b101: 4096 bytes
2:0	3'h5	RW	user_prg_read Maximum read request size issued to the user application. 3'b000: 128 bytes 3'b001: 256 bytes 3'b010: 512 bytes 3'b011: 1024 bytes 3'b100: 2048 bytes 3'b101: 4096 bytes

Table 2-103: **Config Write Flush Timeout (0x60)**

Bit Index	Default	Access Type	Description
4:0	5'h0	RW	Write Flush Timeout Applies to AXI4-Stream C2H channels. This register specifies the number of clock cycles a channel waits for data before flushing the write data it already received from PCIe. This action closes the descriptor and generates a writeback. A value of 0 disables the timeout. The timeout value in clocks = 2^{value}.

H2C SGDMA Registers (0x4)

The H2C SGDMA registers are described in this section.

Table 2-104: **H2C SGDMA Register Space**

Address (hex)	Register Name
0x00	H2C SGDMA Identifier (0x00)
0x80	H2C SGDMA Descriptor Low Address (0x80)
0x84	H2C SGDMA Descriptor High Address (0x84)
0x88	H2C SGDMA Descriptor Adjacent (0x88)

Table 2-105: **H2C SGDMA Identifier (0x00)**

Bit Index	Default	Access Type	Description
31:20	12'h1fc	RO	DMA Subsystem for PCIe identifier
19:16	4'h4	RO	H2C DMA Target
15	1'b0	RO	Stream 1: AXI4-Stream Interface 0: Memory Mapped AXI4 Interface
14:12	3'h0	RO	Reserved
11:8	Varies	RO	Channel ID Target [3:0]
7:0	8'h03	RO	Version 8'h01: 2015.3 and 2015.4 8'h02: 2016.1 8'h03: 2016.2

Table 2-106: **H2C SGDMA Descriptor Low Address (0x80)**

Bit Index	Default	Access Type	Description
31:0	32'h0	RW	dsc_adr[31:0] Lower bits of start descriptor address. Dsc_adr[63:0] is the first descriptor address that is fetched after the Ctrl.run bit is set.

Table 2-107: **H2C SGDMA Descriptor High Address (0x84)**

Bit Index	Default	Access Type	Description
31:0	32'h0	RW	dsc_adr[63:32] Upper bits of start descriptor address. Dsc_adr[63:0] is the first descriptor address that is fetched after the Ctrl.run bit is set.

Table 2-108: **H2C SGDMA Descriptor Adjacent (0x88)**

Bit Index	Default	Access Type	Description
5:0	6'h0	RW	dsc_adj[5:0] Number of extra adjacent descriptors after the start descriptor address.

C2H SGDMA Registers (0x5)

The C2H SGDMA registers are described in this section.

Table 2-109: **C2H SGDMA Register Space**

Address (hex)	Register Name
0x00	C2H SGDMA Identifier (0x00)
0x80	C2H SGDMA Descriptor Low Address (0x80)
0x84	C2H SGDMA Descriptor High Address (0x84)
0x88	C2H SGDMA Descriptor Adjacent (0x88)

Table 2-110: **C2H SGDMA Identifier (0x00)**

Bit Index	Default	Access Type	Description
31:20	12'h1fc	RO	DMA Subsystem for PCIe identifier
19:16	4'h5	RO	C2H DMA Target
15	1'b0	RO	Stream 1: AXI4-Stream Interface 0: Memory Mapped AXI4 Interface
14:12	3'h0	RO	Reserved
11:8	Varies	RO	Channel ID Target [3:0]
7:0	8'h03	RO	Version 8'h01: 2015.3 and 2015.4 8'h02: 2016.1 8'h03: 2016.2

Table 2-111: **C2H SGDMA Descriptor Low Address (0x80)**

Bit Index	Default	Access Type	Description
31:0	32'h0	RW	dsc_adr[31:0] Lower bits of start descriptor address. Dsc_adr[63:0] is the first descriptor address that is fetched after the Ctrl.run bit is set.

Table 2-112: **C2H SGDMA Descriptor High Address (0x84)**

Bit Index	Default	Access Type	Description
31:0	32'h0	RW	dsc_adr[63:32] Upper bits of start descriptor address. Dsc_adr[63:0] is the first descriptor address that is fetched after the Ctrl.run bit is set.

Table 2-113: **C2H SGDMA Descriptor Adjacent (0x88)**

Bit Index	Default	Access Type	Description
5:0	6'h0	RW	dsc_adj[5:0] Number of extra adjacent descriptors after the start descriptor address.

SGDMA Common Registers (0x6)

The SGDMA Common host are described in this section.

Table 2-114: **SGDMA Common Register Space**

Address (hex)	Register Name
0x00	SGDMA Identifier Registers (0x00)
0x10	SGDMA Descriptor Control Register (0x10)
0x14	SGDMA Descriptor Control Register (0x14)
0x18	SGDMA Descriptor Control Register (0x18)

Table 2-115: **SGDMA Identifier Registers (0x00)**

Bit Index	Default	Access Type	Description
31:20	12'h1fc	RO	DMA Subsystem for PCIe identifier
19:16	4'h6	RO	SGDMA Target
15:8	8'h0	RO	Reserved
7:0	8'h03	RO	Version 8'h01: 2015.3 and 2015.4 8'h02: 2016.1 8'h03: 2016.2

Table 2-116: **SGDMA Descriptor Control Register (0x10)**

Bit Index	Default	Access Type	Description
19:16	4'h0	RW	c2h_dsc_halt[15:0] One bit per C2H channel. Set to one to halt descriptor fetches for corresponding channel.
3:0	4'h0	RW	h2c_dsc_halt[15:0] One bit per H2C channel. Set to one to halt descriptor fetches for corresponding channel.

Table 2-117: **SGDMA Descriptor Control Register (0x14)**

Bit Index	Default	Access Type	Description
		W1S	Bit descriptions are the same as in Table 2-116.

Table 2-118: **SGDMA Descriptor Control Register (0x18)**

Bit Index	Default	Access Type	Description
		W1C	Bit descriptions are the same as in Table 2-116.

MSI-X Vector Table and PBA (0x8)

The MSI-X Vector table and PBA are described in Table 2-119.

Table 2-119: **MSI-X Vector Table and PBA (0x00–0xFE0)**

Byte Offset	Bit Index	Default	Access Type	Description
0x00	31:0	32'h0	RW	MSIX_Vector0_Address[31:0] MSI-X vector0 message lower address.
0x04	31:0	32'h0	RW	MSIX_Vector0_Address[63:32] MSI-X vector0 message upper address.
0x08	31:0	32'h0	RW	MSIX_Vector0_Data[31:0] MSI-X vector0 message data.
0x0C	31:0	32'h0	RW	MSIX_Vector0_Control[31:0] MSI-X vector0 control. Bit Position: • 31:1: Reserved. • 0: Mask. When set to one, this MSI-X vector is not used to generate a message.
0x1F0	31:0	32'h0	RW	MSIX_Vector31_Address[31:0] MSI-X vector31 message lower address.
0x1F4	31:0	32'h0	RW	MSIX_Vector31_Address[63:32] MSI-X vector31 message upper address.
0x1F8	31:0	32'h0	RW	MSIX_Vector31_Data[31:0] MSI-X vector31 message data.

Table 2-119: **MSI-X Vector Table and PBA (0x00–0xFE0) *(Cont'd)***

Byte Offset	Bit Index	Default	Access Type	Description
0x1FC	31:0	32'h0	RW	MSIX_Vector31_Control[31:0] MSI-X vector31 control. Bit Position: • 31:1: Reserved. • 0: Mask. When set to one, this MSI-X vector is not used to generate a message.
0xFE0	31:0	32'h0	RW	Pending_Bit_Array[31:0] MSI-X Pending Bit Array. There is one bit per vector. Bit 0 corresponds to vector0, etc.

Designing with the Core

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

Clocking

For information about clocking, see the applicable PCIe™ integrated block product guide:

- *Virtex-7 FPGA Integrated Block for PCI Express Product Guide* (PG023) [Ref 3]
- *UltraScale Architecture Gen3 Integrated Block for PCI Express Product Guide* (PG156) [Ref 4]

Resets

For information about resets, see the applicable PCIe integrated block product guide:

- *Virtex-7 FPGA Integrated Block for PCI Express Product Guide* (PG023) [Ref 3]
- *UltraScale Architecture Gen3 Integrated Block for PCI Express Product Guide* (PG156) [Ref 4]

Tandem Configuration

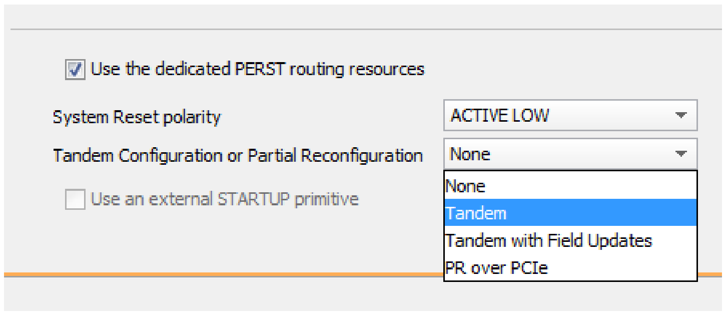
Tandem Configuration features are available for the Xilinx DMA Subsystem for PCI Express® core for all UltraScale™ devices. Tandem Configuration uses a two-stage methodology that enables the IP to meet the configuration time requirements indicated in the PCI Express Specification. Multiple use cases are supported with this technology:

- **Tandem PROM**: Load the single two-stage bitstream from the flash.
- **Tandem PCIe**: Load the first stage bitstream from flash, and deliver the second stage bitstream over the PCIe link to the MCAP.

- **Tandem with Field Updates**: After a Tandem PROM or Tandem PCIe initial configuration, update the entire user design while the PCIe link remains active. The update region (floorplan) and design structure are predefined, and Tcl scripts are provided.
- **Tandem + Partial Reconfiguration**: This is a more general case of Tandem Configuration followed by Partial Reconfiguration (PR) of any size or number of PR regions.
- **Partial Reconfiguration over PCIe**: This is a standard configuration followed by PR, using the PCIe/MCAP as the delivery path of partial bitstreams.

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

1. Change the **Mode** to **Advanced**.
2. Change the **Tandem Configuration or Partial Reconfiguration** option according to your particular case:
   - **Tandem** for Tandem PROM, Tandem PCIe or Tandem + Partial Reconfiguration use cases.
   - **Tandem with Field Updates** ONLY for the predefined Field Updates use case.
   - **PR over PCIe** to enable the MCAP link for Partial Reconfiguration, without enabling Tandem Configuration.


[✓] Use the dedicated PERST routing resources

System Reset polarity: ACTIVE LOW

Tandem Configuration or Partial Reconfiguration: None

[ ] Use an external STARTUP primitive

None

Tandem

Tandem with Field Updates

PR over PCIe

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

For complete information about Tandem Configuration, including required PCIe block locations, design flow examples, requirements, restrictions and other considerations, see Tandem Configuration in the *UltraScale Architecture Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide (PG156)* [Ref 4]. For information on Partial Reconfiguration, see the *Vivado Design Suite User Guide: Partial Reconfiguration (UG909)* [Ref 10].

Design Flow Steps

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

- *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [Ref 5]
- *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 6]
- *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 7]
- *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 9]

Customizing and Generating the Core

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

If you are customizing and generating the core in the Vivado IP integrator, see the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [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, run the `validate_bd_design` command in the Tcl console.

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

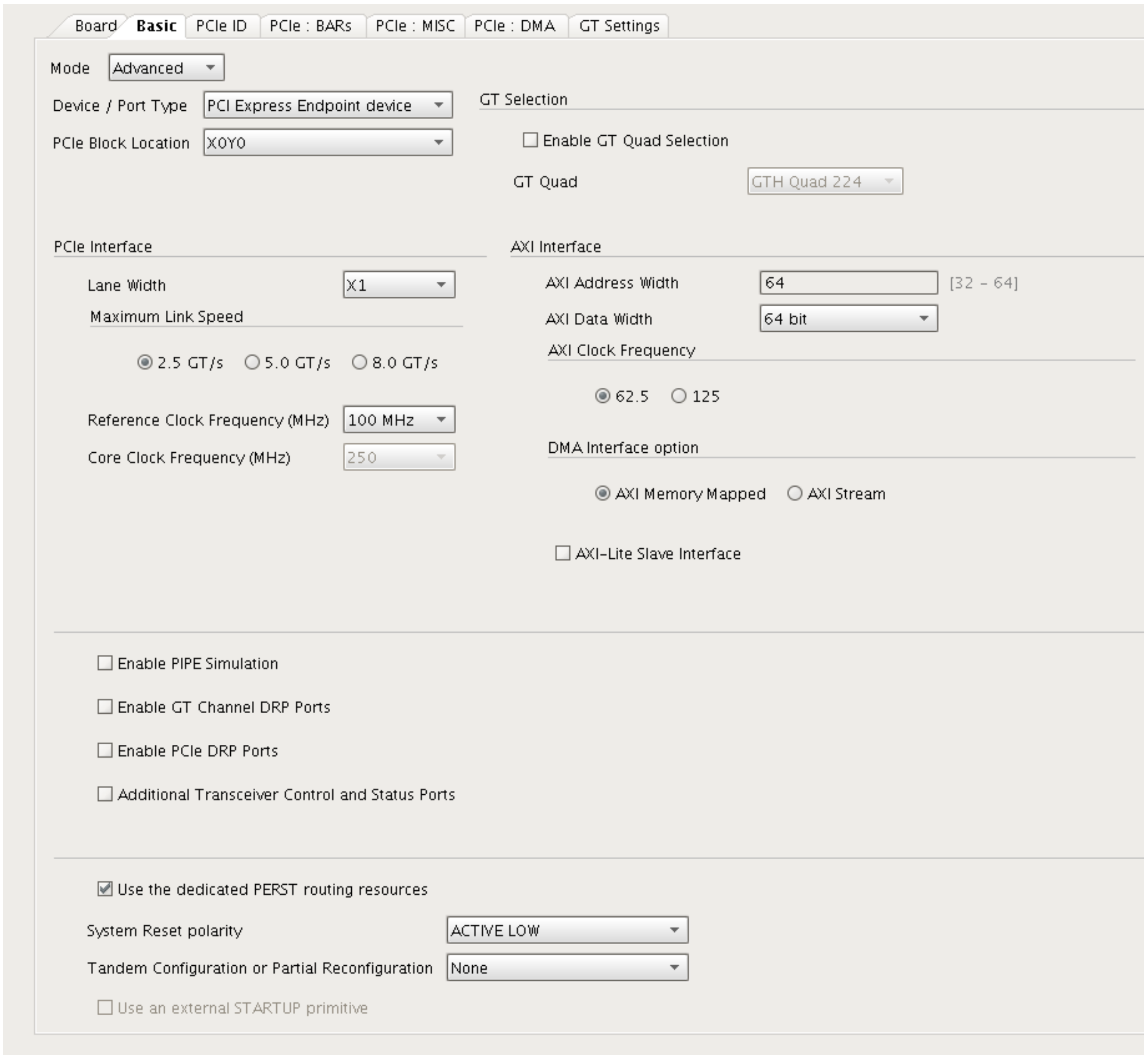
1. Select the IP from the Vivado IP catalog.
2. Double-click the selected IP or select the **Customize IP** command from the toolbar or right-click menu.

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 6] and the *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 7].

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

Basic Tab

The Basic tab is shown in Figure 4-1.



Board | **Basic** | PCIe ID | PCIe : BARs | PCIe : MISC | PCIe : DMA | GT Settings

Mode: Advanced

Device / Port Type: PCI Express Endpoint device

PCIe Block Location: X0Y0

GT Selection

☐ Enable GT Quad Selection

GT Quad: GTH Quad 224

PCIe Interface

Lane Width: X1

Maximum Link Speed

[✓ 2.5 GT/s] 5.0 GT/s 8.0 GT/s

Reference Clock Frequency (MHz): 100 MHz

Core Clock Frequency (MHz): 250

AXI Interface

AXI Address Width: 64 [32 - 64]

AXI Data Width: 64 bit

AXI Clock Frequency

[✓ 62.5] 125

DMA Interface option

[✓ AXI Memory Mapped] AXI Stream

☐ AXI-Lite Slave Interface

☐ Enable PIPE Simulation

☐ Enable GT Channel DRP Ports

☐ Enable PCIe DRP Ports

☐ Additional Transceiver Control and Status Ports

☑ Use the dedicated PERST routing resources

System Reset polarity: ACTIVE LOW

Tandem Configuration or Partial Reconfiguration: None

☐ Use an external STARTUP primitive

Figure 4-1: **Basic Tab**

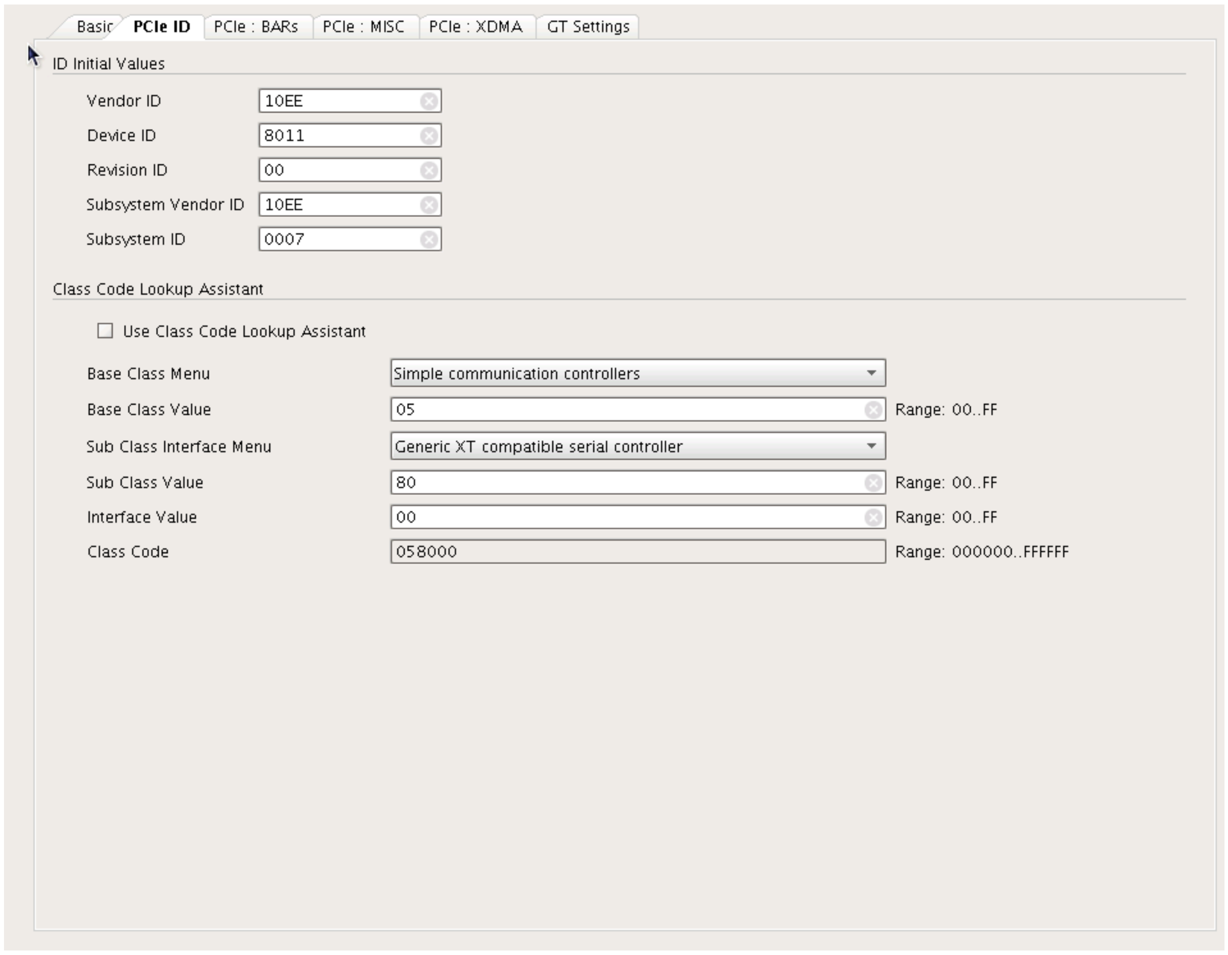
The options are defined as follows:

- **Mode**: Allows you to select the **Basic** or **Advanced** mode of the configuration of core.
- **Device /Port Type**: Only PCI Express® Endpoint device mode is supported.
- **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.

- **Lane Width**: The core requires the selection of the initial lane width. Table 4-1 in the *UltraScale Architecture Gen3 Integrated Block for PCI Express Product Guide* (PG156) [Ref 4] 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.
- **Maximum Link Speed**: The core allows you to select the Maximum Link Speed supported by the device. Table 4-2 in the *UltraScale Architecture Gen3 Integrated Block for PCI Express Product Guide* (PG156) [Ref 4] 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. Select Gen1, Gen2, or Gen3.
- **Reference Clock Frequency**: The default is 100 MHz, but 125 and 250 MHz are also supported
- **GT Selection/Enable GT Quad Selection**: Select the Quad in which lane 0 is located.
- **AXI Address Width**: Currently, only 64-bit width is supported.
- **AXI Data Width**: Select 64, 128, or 256 bit. The core allows you to select the Interface Width, as defined in Table 4-3 in the *UltraScale Architecture Gen3 Integrated Block for PCI Express Product Guide* (PG156) [Ref 4]. The default interface width set in the **Customize IP** dialog box is the lowest possible interface width.
- **AXI Clock Frequency**: Select 62.5 MHz, 125 MHz or 250 MHz depending on the lane width/speed.
- **DMA Interface Option**: Select AXI4 Memory Mapped and AXI4-Stream.
- **AXI Lite Slave Interface**: Select to enable the AXI4-Lite slave Interface.
- **Tandem Configuration or Partial Reconfiguration**: Select the tandem configuration or partial reconfiguration feature, if application to your design. See Tandem Configuration for details.

PCIe ID Tab

The PCIe ID tab is shown in Figure 4-2.



ID Initial Values

Vendor ID	10EE
Device ID	8011
Revision ID	00
Subsystem Vendor ID	10EE
Subsystem ID	0007

Class Code Lookup Assistant

☐ Use Class Code Lookup Assistant

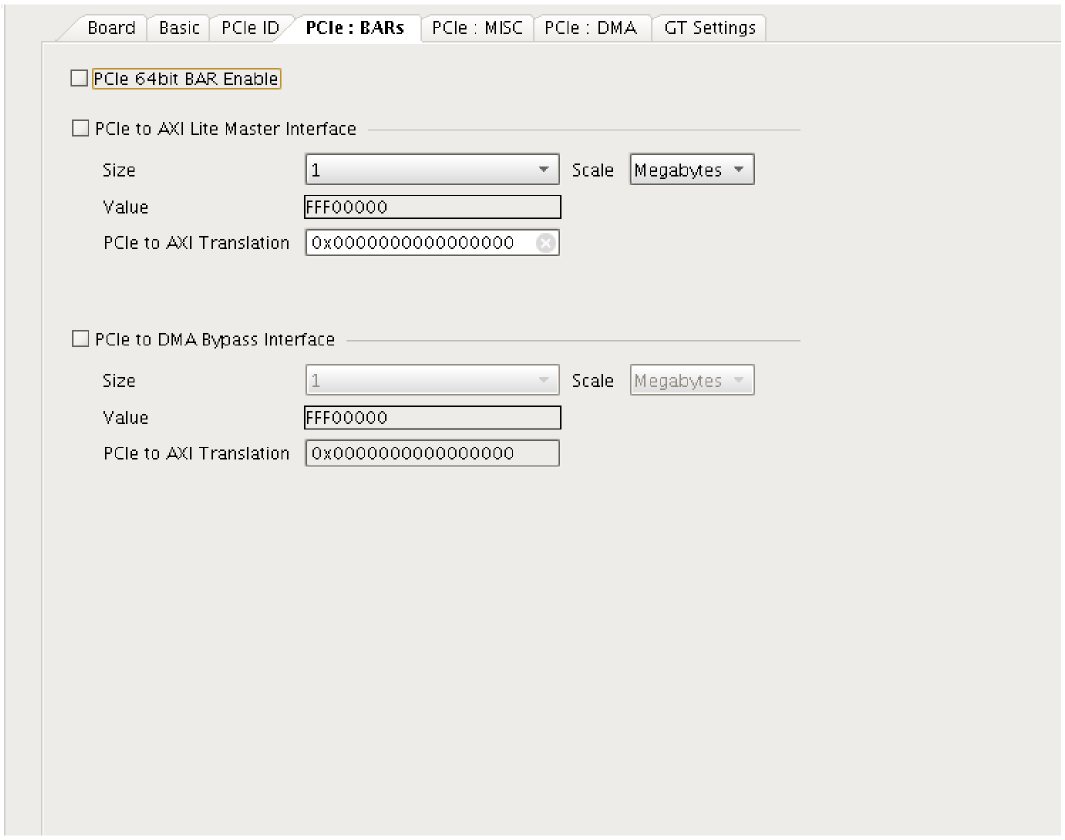
Base Class Menu	Simple communication controllers	
Base Class Value	05	Range: 00..FF
Sub Class Interface Menu	Generic XT compatible serial controller	
Sub Class Value	80	Range: 00..FF
Interface Value	00	Range: 00..FF
Class Code	058000	Range: 000000..FFFFFF

Figure 4-2: **PCIe ID Tab**

For a description of these options, see Chapter 4, "Design Flow Steps" in the *Virtex-7 FPGA Integrated Block for PCI Express Product Guide* (PG023) [Ref 3], or *UltraScale Architecture Gen3 Integrated Block for PCI Express Product Guide* (PG156) [Ref 4].

PCIe: BARs Tab

The PCIe BARs tab is shown in Figure 4-3.



☐ PCIe 64bit BAR Enable

☐ PCIe to AXI Lite Master Interface

Size	1	Scale	Megabytes
Value	FFF00000		
PCIe to AXI Translation	0x0000000000000000		

☐ PCIe to DMA Bypass Interface

Size	1	Scale	Megabytes
Value	FFF00000		
PCIe to AXI Translation	0x0000000000000000		

Figure 4-3: **PCIe BARs Tab**

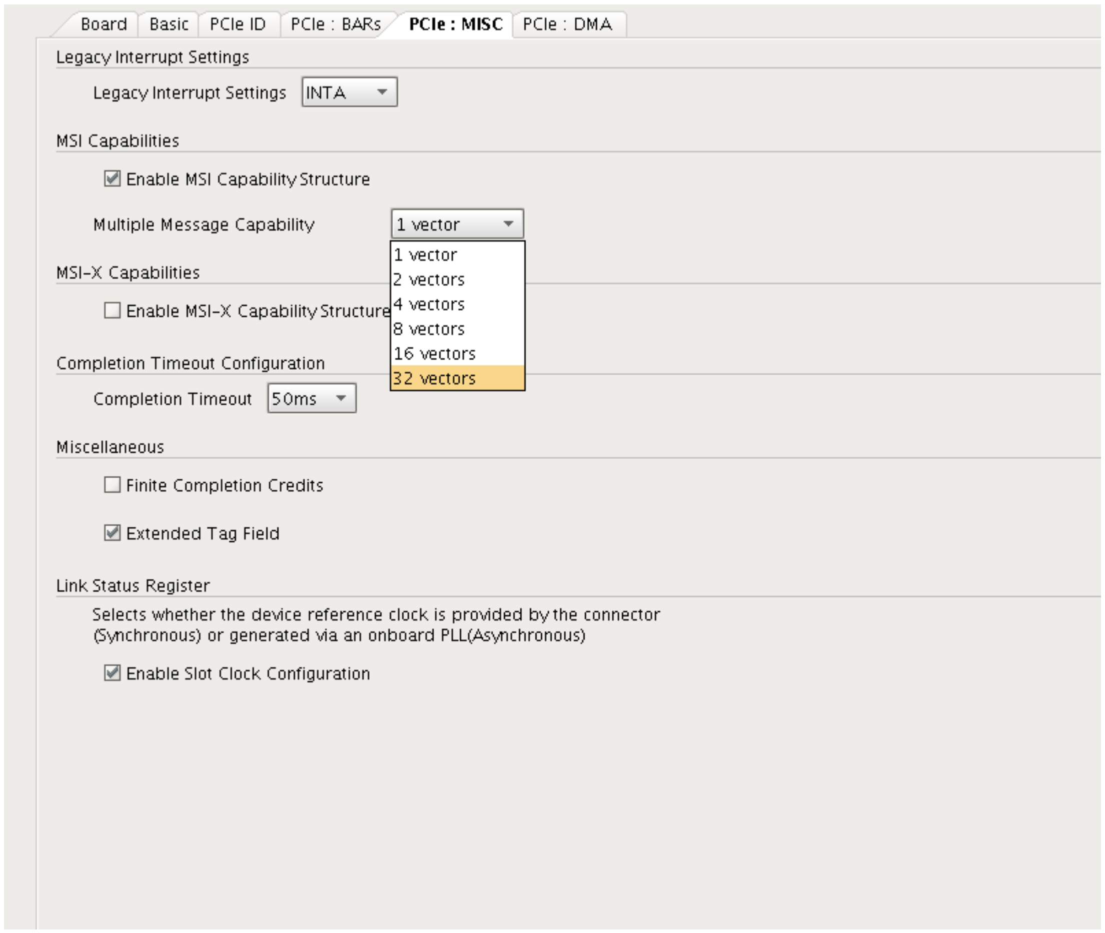
PCIe 64 bit BAR Enable: Enables the 64-bit non-prefetchable address space for PCIe to AXI-Lite Master, PCIe to DMA, and PCIE to DMA bypass interfaces.

PCIe to AXI Lite Master Interface: You can optionally enable **PCIe to AXI-Lite Interface** BAR space. The size, scale, and address translation are configurable.

PCIe to DMA Bypass Interface: You can optionally enable **PCIe to DMA Bypass Interface** BAR space. The size, scale and address translations are also configurable.

PCIe: Misc Tab

The PCIe Miscellaneous tab is shown in Figure 4-4.



Legacy Interrupt Settings

Legacy Interrupt Settings: INTA

MSI Capabilities

☑ Enable MSI Capability Structure

Multiple Message Capability: 1 vector

MSI-X Capabilities

☐ Enable MSI-X Capability Structure

Completion Timeout Configuration

Completion Timeout: 50ms

Miscellaneous

☐ Finite Completion Credits

☑ Extended Tag Field

Link Status Register

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

☑ Enable Slot Clock Configuration

Figure 4-4: **PCIe Misc Tab**

Legacy Interrupt Settings: Select one of the Legacy Interrupts: INTA, INTB, INTC, or INTD.

MSI Capabilities: By default, MSI Capabilities is enabled, and 1 vector is enabled. You can choose up to 32 vectors. In general, Linux uses only 1 vector for MSI. This option can be disabled.

MSI-X Capabilities: Select a MSI-X event. For more information, see MSI-X Vector Table and PBA (0x8).

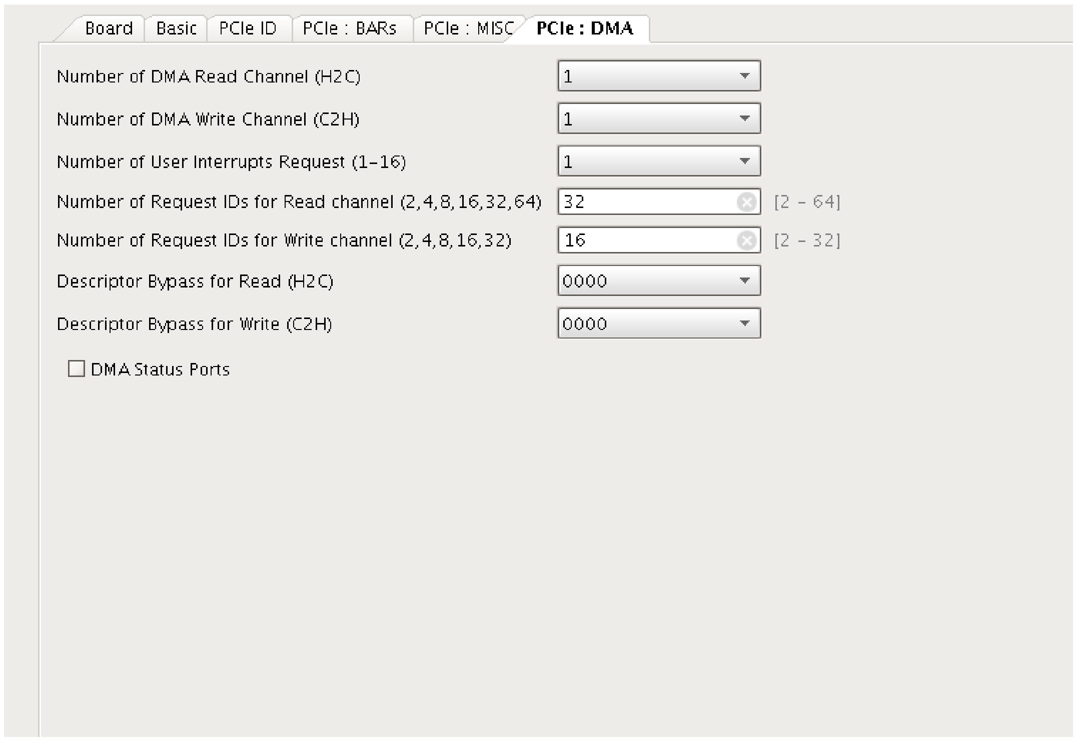
Completion Timeout Configuration: By default, completion timeout is set to 50 ms. Option of 50us is also available.

Finite Completion Credits: On systems which support finite completion credits, this option can be enabled for better performance.

PCI Extended Tag: By default, 6-bit completion tags are used.

PCIe: DMA Tab

The DMA tab is shown in Figure 4-5.



Number of DMA Read Channel (H2C) 1
Number of DMA Write Channel (C2H) 1
Number of User Interrupts Request (1-16) 1
Number of Request IDs for Read channel (2,4,8,16,32,64) 32 [2 - 64]
Number of Request IDs for Write channel (2,4,8,16,32) 16 [2 - 32]
Descriptor Bypass for Read (H2C) 0000
Descriptor Bypass for Write (C2H) 0000
☐ DMA Status Ports

Figure 4-5: **DMA Tab**

Number of Read/ Write Channels: Available selection is from 1 to 4.

Number of User Interrupt Request: Up to 16 user interrupt requests can be selected.

Number of Request IDs for Read channel: Select the max number of outstanding request per channel. Available selection is from 2 to 64.

Number of Request IDs for Write channel: Select max number of outstanding request per channel. Available selection is from 2 to 32.

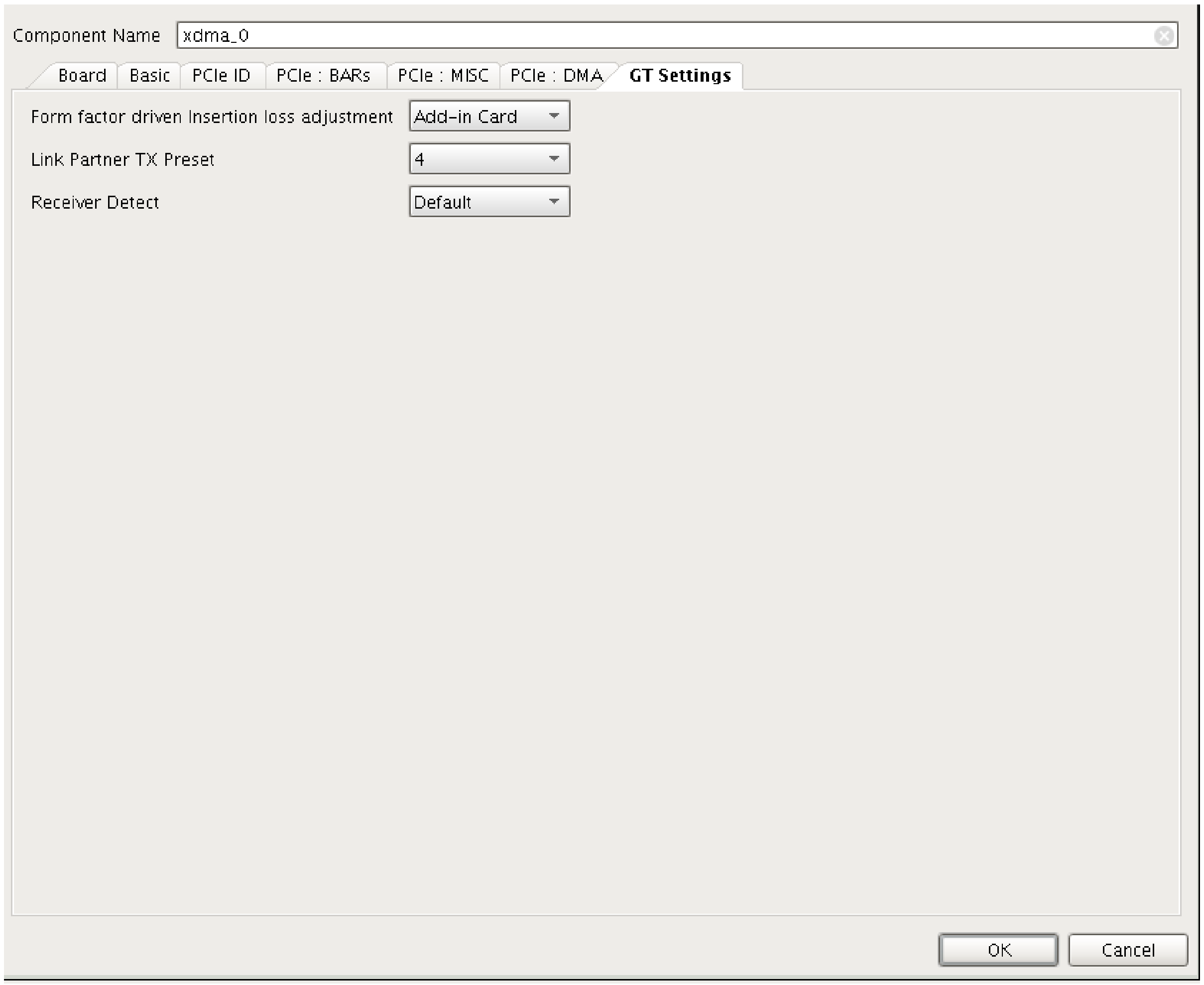
Descriptor Bypass for Read (H2C): Available for all selected read channels. Each binary digits corresponds to a channel. LSB corresponds to Channel 0. Value of one in bit position means corresponding channels has Descriptor bypass enabled.

Descriptor Bypass for Write (C2H): Available for all selected write channels. Each binary digits corresponds to a channel. LSB corresponds to Channel 0. Value of one in bit position means corresponding channels has Descriptor bypass enabled.

DMA Status port: DMA status ports are available for all channels

GT Settings Tab

The GT Settings tab is shown in Figure 4-6.


Component Name: xdma_0

Form factor driven Insertion loss adjustment	Add-in Card
Link Partner TX Preset	4
Receiver Detect	Default

Figure 4-6: **GT Settings Tab**

For a description of these options, see Chapter 4, "Design Flow Steps" in the *UltraScale Architecture Gen3 Integrated Block for PCI Express Product Guide* (PG156) [Ref 4].

Output Generation

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 6].

Constraining the Core

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

Required Constraints

The DMA Subsystem for PCIe requires the specification of timing and other physical implementation constraints to meet specified performance requirements for PCI Express. These constraints are provided in a Xilinx Design Constraints (XDC) file. Pinouts and hierarchy names in the generated XDC correspond to the provided example design.

IMPORTANT: *If the example design top file is not used, copy the IBUFDS_GTE3 instance for the reference clock, IBUF Instance for sys_rst and also the location and timing constraints associated with them into your local design top.*

To achieve consistent implementation results, an XDC containing these original, unmodified constraints must be used when a design is run through the Xilinx tools. For additional details on the definition and use of an XDC or specific constraints, see *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 8].

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

For detailed information about clock requirements, see *Virtex-7 FPGA Integrated Block for PCI Express Product Guide* (PG023) [Ref 3], or *UltraScale Architecture Gen3 Integrated Block for PCI Express Product Guide* (PG156) [Ref 4].

Clock Management

For detailed information about clock requirements, see *Virtex-7 FPGA Integrated Block for PCI Express Product Guide* (PG023) [Ref 3], or *UltraScale Architecture Gen3 Integrated Block for PCI Express Product Guide* (PG156) [Ref 4].

Clock Placement

For detailed information about clock requirements, see *Virtex-7 FPGA Integrated Block for PCI Express Product Guide* (PG023) [Ref 3], or *UltraScale Architecture Gen3 Integrated Block for PCI Express Product Guide* (PG156) [Ref 4].

Banking

This section is not applicable for this IP core.

Transceiver Placement

This section is not applicable for this IP core.

I/O Standard and Placement

This section is not applicable for this IP core.

Relocating the Integrated Block Core

By default, the IP core-level constraints lock block RAMs, transceivers, and the PCIe block to the recommended location. To relocate these blocks, you must override the constraints for these blocks in the XDC constraint file. To do so:

1. Copy the constraints for the block that needs to be overwritten from the core-level XDC constraint file.
2. Place the constraints in the user XDC constraint file.
3. Update the constraints with the new location.

The user XDC constraints are usually scoped to the top-level of the design; therefore, ensure that the cells referred by the constraints are still valid after copying and pasting them. Typically, you need to update the module path with the full hierarchy name.

Note: If there are locations that need to be swapped (that is, the new location is currently being occupied by another module), there are two ways to do this.

- If there is a temporary location available, move the first module out of the way to a new temporary location first. Then, move the second module to the location that was occupied by the first module. Next, move the first module to the location of the second module. These steps can be done in XDC constraint file.

- If there is no other location available to be used as a temporary location, use the `reset_property` command from Tcl command window on the first module before relocating the second module to this location. The `reset_property` command cannot be done in XDC constraint file and must be called from the Tcl command file or typed directly into the Tcl Console.

Simulation

This section contains information about simulating IP in the Vivado Design Suite.

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 9].

PIPE Mode Simulation

The DMA Subsystem for PCIe supports the PIPE mode simulation where the PIPE interface of the core is connected to the PIPE interface of the link partner. This mode increases the simulation speed.

Use the **Enable PIPE Simulation** option on the **Basic** page of the **Customize IP** dialog box to enable PIPE mode simulation in the current Vivado Design Suite solution example design, in either Endpoint mode or Root Port mode. The External PIPE Interface signals are generated at the core boundary for access to the external device. Enabling this feature also provides the necessary hooks to use third-party PCI Express VIPs/BFMs instead of the Root Port model provided with the example design.

Table 4-1 and Table 4-2 describe the PIPE bus signals available at the top level of the core and their corresponding mapping inside the EP core (`pcie_top`) PIPE signals.

IMPORTANT: *A new file, `xil_sig2pipe.v`, is delivered in the simulation directory, and the file replaces `phy_sig_gen.v`. BFM/VIPs should interface with the xil_sig2pipe instance in `board.v`.*

Table 4-1: **Common In/Out Commands and Endpoint PIPE Signals Mappings**

In Commands	Endpoint PIPE Signals Mapping	Out Commands	Endpoint PIPE Signals Mapping
common_commands_in[25:0]	not used	common_commands_out[0]	pipe_clk[1]
		common_commands_out[2:1]	pipe_tx_rate_gt[2]
		common_commands_out[3]	pipe_tx_rcvr_det_gt
		common_commands_out[6:4]	pipe_tx_margin_gt
		common_commands_out[7]	pipe_tx_swing_gt
		common_commands_out[8]	pipe_tx_reset_gt
		common_commands_out[9]	pipe_tx_deemph_gt
		common_commands_out[16:10]	not used[3]

Notes:

1. `pipe_clk` is an output clock based on the core configuration. For Gen1 rate, `pipe_clk` is 125 MHz. For Gen2 and Gen3, `pipe_clk` is 250 MHz.
2. `pipe_tx_rate_gt` indicates the pipe rate (2'b00-Gen1, 2'b01-Gen2, and 2'b10-Gen3).
3. The functionality of this port has been deprecated and it can be left unconnected.

Table 4-2: **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_elec_idle_gt	pipe_tx_0_sigs[34]	pipe_tx0_elec_idle_gt
pipe_rx_0_sigs[35]	pipe_rx0_data_valid_gt	pipe_tx_0_sigs[35]	pipe_tx0_data_valid_gt
pipe_rx_0_sigs[36]	pipe_rx0_start_block_gt	pipe_tx_0_sigs[36]	pipe_tx0_start_block_gt
pipe_rx_0_sigs[38:37]	pipe_rx0_syncheader_gt	pipe_tx_0_sigs[38:37]	pipe_tx0_syncheader_gt
pipe_rx_0_sigs[83:39]	not used	pipe_tx_0_sigs[39]	pipe_tx0_polarity_gt
		pipe_tx_0_sigs[41:40]	pipe_tx0_powerdown_gt
		pipe_tx_0_sigs[69:42]	not used[1]

Notes:

1. The functionality of this port has been deprecated and it can be left unconnected.

Synthesis and Implementation

For details about synthesis and implementation, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 6].

Example Design

This chapter contains information about the example designs provided in the Vivado® Design Suite. The example designs are as follows:

- AXI4 Memory Mapped Default Example Design
- AXI4 Memory Mapped with PCIe to AXI-Lite Master and PCIe to DMA Bypass Example Design
- AXI4 Memory Mapped with AXI4-Lite Slave Interface Example Design
- AXI4-Stream Example Design

AXI4 Memory Mapped Default Example Design

Figure 5-1 shows the AXI4 Memory Mapped (AXI-MM) interface as the default design. The example design gives 4 kilobytes (KB) block RAM on user design with AXI4 MM interface. For H2C transfers, the DMA Subsystem for PCIe reads data from host and writes to block RAM in the user side. For C2H transfers, the DMA Subsystem for PCIe reads data from block RAM and writes to host memory.

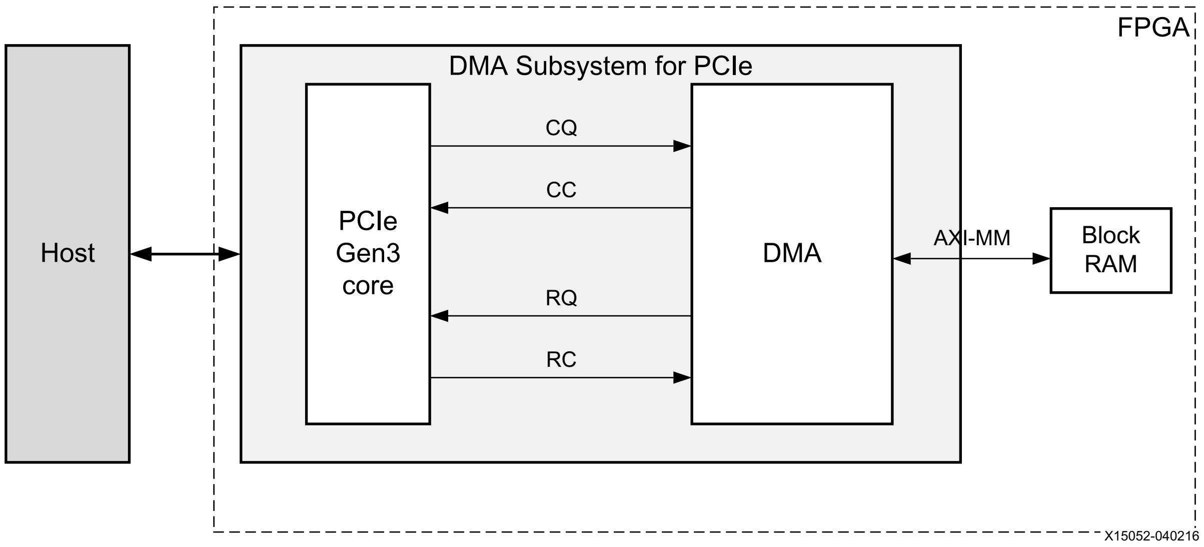


Figure 5-1: **AXI-MM Default Example Design**

AXI4 Memory Mapped with PCIe to AXI-Lite Master and PCIe to DMA Bypass Example Design

Figure 5-2 shows a system where the PCIe to AXI-Lite Master (BAR0) and PCIe to DMA Bypass (BAR2) are selected. 4K block RAM is connected to the PCIe to DMA Bypass interfaces. The host can use **DMA Bypass interface** to read/write data to the user space using the AXI4 MM interface. This interface bypasses DMA engines. The host can also use the PCIe to AXI-Lite Master (BAR0 address space) to write/read user logic. The example design connects 4K block RAM to the PCIe to AXI-Lite Master interface so the user application can perform read/writes.

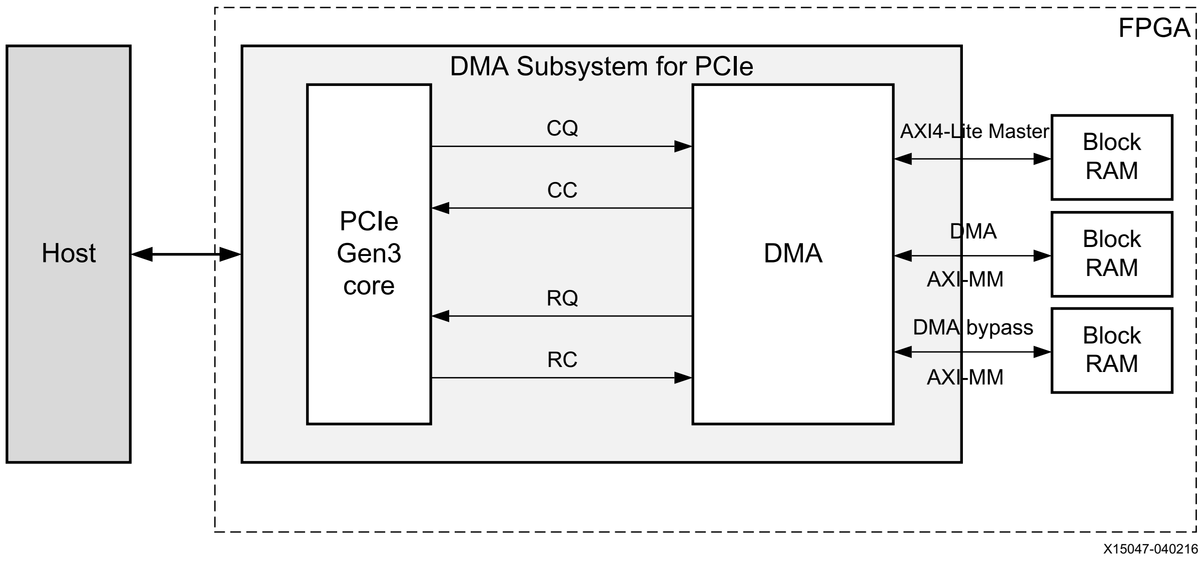


Figure 5-2: **AXI-MM Example with PCIe to DMA Bypass Interface and PCIe to AXI-Lite Master Enabled**

AXI4 Memory Mapped with AXI4-Lite Slave Interface Example Design

When the PCIe to AXI-Lite master and AXI4-Lite slave interface are enabled, the generated example design (shown in Figure 5-3) has a loopback from AXI4-Lite master to AXI4-Lite slave. Typically, the user logic can use a AXI4-Lite slave interface to read/write DMA Subsystem for PCIe registers. With this example design, the host can use **PCIe to AXI-Lite Master** (BAR0 address space) and read/write DMA Subsystem for PCIe registers, which is the same as using **PCIe to DMA** (BAR1 address space). This example design also shows **PCIe to DMA bypass Interface** (BAR2) enabled.

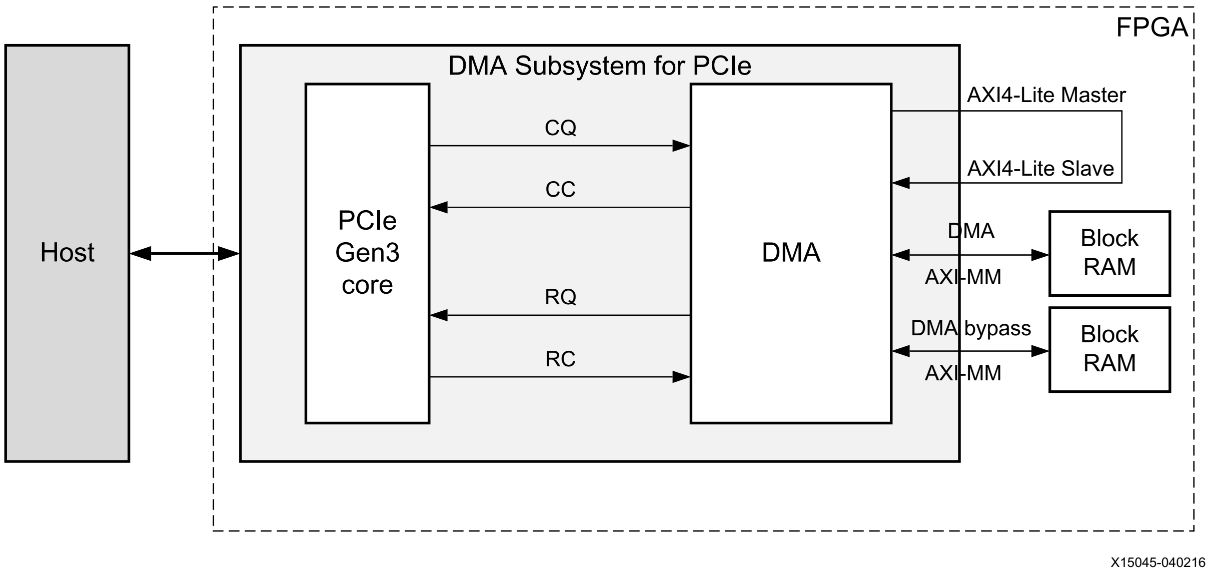


Figure 5-3: **AXI-MM Example with AXI4-Lite Slave Enabled**

AXI4-Stream Example Design

When the AXI4-Stream interface is enabled, each H2C streaming channels is looped back to C2H channel. Shown in Figure 5-4, the example design gives a loopback design for AXI4 streaming. The limitation is that you need to select an equal number of H2C and C2H channels for proper operation. This example design also shows **PCIe to DMA bypass interface** and **PCIe to AXI-Lite Master** selected.

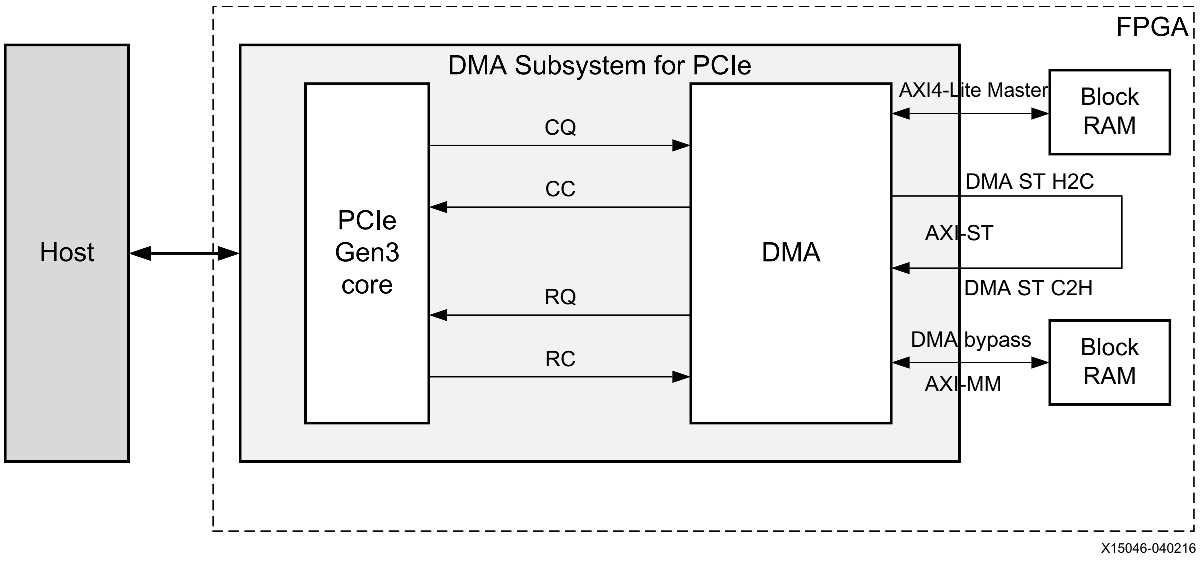


Figure 5-4: **AXI4 Stream Example with PCIe to DMA Bypass Interface and PCIe to AXI-Lite Master Enabled**

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 shows the Root Port Module with DMA Subsystem for PCIe.

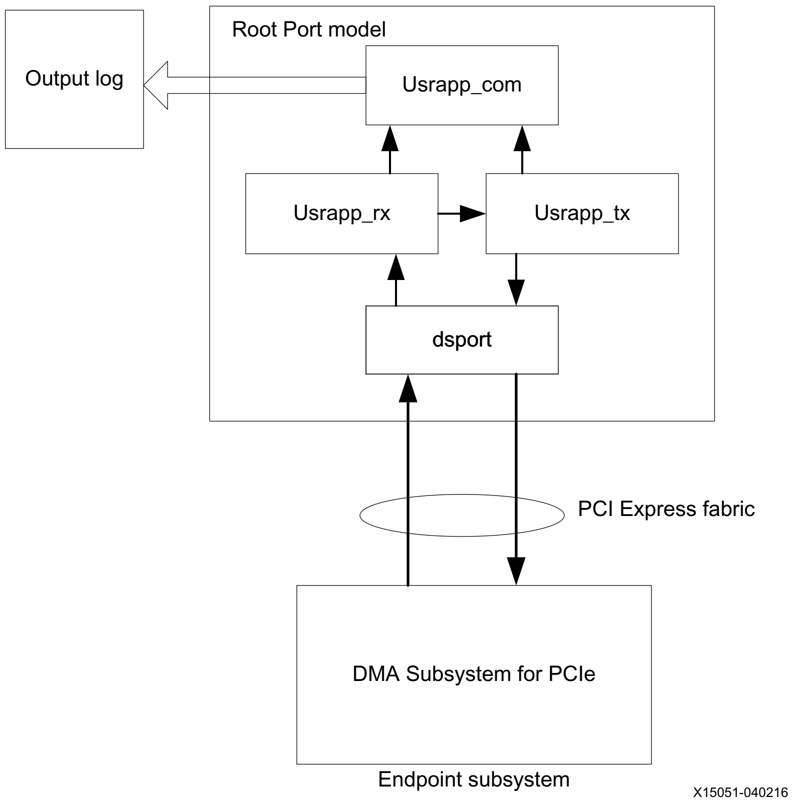


Figure 6-1: **Root Port Module with DMA Subsystem for PCIe**

Architecture

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

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

The `usrapp_tx` and `usrapp_rx` blocks interface with the dsport block for transmission and reception of TLPs to/from the EndPoint DUT. The Endpoint DUT consists of the DMA Subsystem for PCIe.

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.

PIO write and read are initiated by `usrapp_tx`.

The DMA Subsystem for PCIe uses the UltraScale™ Architecture Gen3 Integrate Block for PCIe, and the 7 series Gen3 Integrated Block for PCIe. See the "Test Bench" chapter in the *Virtex-7 FPGA Integrated Block for PCI Express Product Guide* (PG023) [Ref 3] and *UltraScale Architecture Gen3 Integrated Block for PCI Express Product Guide* (PG156) [Ref 4], respectively.

Test Case

The DMA Subsystem for PCIe can be configured as AXI4 Memory Mapped (AXI-MM) or AXI4-Stream (AXI_ST) interface. The simulation test case reads configuration register to determine if a AXI4 Memory Mapped or AXI4-Stream configuration. The test case, based on the AXI settings, performs simulation for either configuration.

Table 6-1: **Test Case Descriptions**

Test Case Name	Description
Dma_test0	AXI4 Memory Mapped interface simulation. Reads data from host memory and writes to block BRAM (H2C). Then, reads data from BRAM and write to host memory (C2H). The test case at the end compares data for correctness.
Dma_stream0	AXI-Stream interface simulation. Reads data from host memory and sends to AXI4-Stream user interface (H2C), and the data is looped back to host memory (C2H).

Simulation

AXI4 Memory Mapped Interface

First, the test case starts the H2C engine. The H2C engine reads data from host memory and writes to BRAM on user side. Then, the test case starts the C2H engine. The C2H engine reads data from BRAM and writes to host memory. The following are the simulation steps:

1. The test case sets up one descriptor for the H2C engine.
2. The H2C descriptor gives the data length 64 bytes, source address (host), and destination address (Card).
3. Writes data (incremental data) in the source address space.
4. The test case also sets up one descriptor for the C2H engine.
5. The C2H descriptor gives data length 64 bytes, source address (Card), and destination address (host).
6. The PIO writes to H2C descriptor starting register.
7. The PIO writes to H2C control register to start H2C transfer.

8. The DMA transfer takes data from the host source address to the block RAM destination address.
9. The test case then starts the C2H transfer.
10. The PIO writes to the C2H descriptor starting register.
11. The PIO writes to the C2H control register to start the C2H transfer.
12. The DMA transfer takes data from the block RAM source address to the host destination address.
13. The test case then compares the data for correctness.
14. The test case checks for the H2C and C2H descriptor completed count (value of 1).

AXI4-Stream Interface

For AXI4-Stream, the example design is a loopback design. First, the test case starts the C2H engine. The C2H engine waits for data that is transmitted by the H2C engine. Then, the test case starts the H2C engine. The H2C engine reads data from host and sends to the Card, which is looped back to the C2H engine. The C2H engine then takes the data, and writes back to host memory. The following are the simulation steps:

1. The test case sets up one descriptor for the H2C engine.
2. The H2C descriptor gives the data length 64 bytes, Source address (host), and Destination address (Card).
3. Writes data (incremental data) in the source address space.
4. The test case also sets up one descriptor for the C2H engine.
5. The C2H descriptor gives data length 64 bytes, source address (Card), and destination address (host).
6. The PIO writes to the C2H descriptor starting register.
7. The PIO writes to the C2H control register to start the C2H transfer first.
8. The C2H engine starts and waits for data to come from the H2C ports.
9. The PIO writes to the H2C descriptor starting register.
10. The PIO writes to the H2C control register to start C2H transfer.
11. The H2C engine takes data from the host source address to the Card destination address.
12. The data is looped back to the C2H engine.
13. The C2H engine takes data from the Card, then sends and writes to the host memory destination address.
14. The test case checks for the H2C and C2H descriptor completed count (value of 1).

Test Tasks

Table 6-2: **Test Tasks**

Name	Description
TSK_INIT_DATA_H2C	This task generates one descriptor for H2C engine and initializes source data in host memory.
TSK_INIT_DATA_C2H	This task generates one descriptor for C2H engine.
TSK_XDMA_REG_READ	This task reads the DMA Subsystem for PCIe register.
TSK_XDMA_REG_WRITE	This task writes the DMA Subsystem for PCIe register.
COMPARE_DATA_H2C	This task compares source data in the host memory to destination data written to block RAM. This task is used in AXI4 Memory Mapped simulation.
COMPARE_DATA_C2H	This task compares the original data in the host memory to the data C2H engine writing to host. This task is used in AXI4 Memory Mapped simulation

For other PCIe related tasks, see the “Test Bench” chapter in *Virtex-7 FPGA Integrated Block for PCI Express Product Guide* (PG023) [Ref 3], or *UltraScale Architecture Gen3 Integrated Block for PCI Express Product Guide* (PG156) [Ref 4].

Device Driver

This chapter provide details about the Linux device driver that is provided with the core. For additional information about the driver, see AR 65444.

Overview

The Linux device driver has the following character device interfaces:

- User character device for access to user components.
- Control character device for controlling DMA Subsystem for PCIe components.
- Events character device for waiting for interrupt events.
- SGDMA character devices for high performance transfers.

The user accessible devices are as follows:

- **XDMA0_control**: Used to access DMA Subsystem for PCIe registers.
- **XDMA0_user**: Used to access AXI-Lite master interface.
- **XDMA0_bypass**: Used to access DMA Bypass interface.
- **XDMA0_events_***: Used to recognize user interrupts.

Interrupt Processing

Legacy Interrupts

There are four types of legacy interrupts: A, B, C and D. You can select any interrupts in the **PCIe: MISC** tab under **Legacy Interrupt Settings**. You must program the corresponding values for both IRQ Block Channel Vector (see Table 2-90) and IRQ Block User Vector (see Table 2-86). Values for each legacy interrupts are A = 0, B = 1, C = 2 and D = 3. The host recognizes interrupts only based on these values.

MSI Interrupts

For MSI interrupts, you can select from 1 to 32 vectors in the **PCIe:MISC** tab under **MSI Capabilities**. The Linux operating system (OS) supports only 1 vector. Other operating systems might support more vectors and you can program different vectors values in the IRQ Block Channel Vector (see Table 2-90) and in the IRQ Block User Vector (see Table 2-86) to represent different interrupt sources. The Xilinx Linux driver supports only 1 MSI vector.

MSI-X Interrupts

The DMA supports up to 32 different interrupt source for MSI-X. The DMA has 32 entire tables, one for each source (see Table 2-119). For MSI-X channel interrupt processing the driver should use the Engine Interrupt Enable Mask for H2C and C2H (see Table 2-47 or Table 2-66) to disable and enable interrupts.

User interrupts

The user logic must hold `usr_irq_req` active-High even after receiving `usr_irq_ack` (acks) for the user interrupt to work properly. This enables the driver to determine the source of the interrupt. Once the driver receives user interrupts, the driver or software can reset the user interrupts (`usr_irq_req`). This is the same for MSI and Legacy Interrupts. For MSI-X interrupts, the MSI-X table can be programmed with user interrupt information. Once the host receives the interrupt, the driver or software is aware of the interrupt source so the user logic can deassert `usr_irq_req` after receiving ack.

Example H2C Flow

In the example H2C flow, `loaddriver.sh` loads devices for all available channels. The `dma_to_device` user program transfers data from host to Card.

The example H2C flow sequence is as follow:

1. Open the H2C device and initialize the DMA.
2. The user program reads the data file, allocates a buffer pointer, and passes the pointer to write function with the specific device (H2C) and data size.
3. The driver creates a descriptor based on input data/size and initializes the DMA with descriptor start address, and if there are any adjacent descriptor.
4. The driver writes a control register to start the DMA transfer.
5. The DMA reads descriptor from the host and starts processing each descriptor.
6. The DMA fetches data from the host and sends the data to the user side. After all data is transferred based on the settings, the DMA generates an interrupt to the host.

7. The ISR driver processes the interrupt to find out which engine is sending the interrupt and checks the status to see if there are any errors. It also checks how many descriptors are processed.
8. After the status is good, the driver returns the transfer byte length to the user side so it can check for the same.

Example C2H Flow

In the example C2H flow, `loaddriver.sh` loads the devices for all available channels. The `dma_from_device` user program transfers data from Card to host.

The example C2H flow sequence is as follow:

1. Open device C2H and initialize the DMA.
2. The user program allocates buffer pointer (based on size), passes pointer to read function with specific device (C2H) and data size.
3. The driver creates descriptor based on size and initializes the DMA with descriptor start address. Also if there are any adjacent descriptor.
4. The driver writes control register to start the DMA transfer.
5. The DMA reads descriptor from host and starts processing each descriptor.
6. The DMA fetches data from Card and sends data to host. After all data is transferred based on the settings, the DMA generates an interrupt to host.
7. The ISR driver processes the interrupt to find out which engine is sending the interrupt and checks the status to see if there are any errors and also checks how many descriptors are processed.
8. After the status is good, the drive returns transfer byte length to user side so it can check for the same.

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 PCIe DMA, the Xilinx Support web page contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support.

Documentation

This product guide is the main document associated with the PCIe DMA. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page or by using the Xilinx Documentation Navigator.

Download the Xilinx Documentation Navigator from the Downloads page. For more information about this tool and the features available, open the online help after installation.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

See the Xilinx Solution Center for PCI Express for the PCIe DMA.

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 PCIe DMA

AR 65443

Technical Support

Xilinx provides technical support at the Xilinx Support web page for this IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support if you do any of the following:

- Implement the solution in devices that are not defined in the documentation.
- Customize the solution beyond that allowed in the product documentation.
- Change any section of the design labeled DO NOT MODIFY.

To contact Xilinx Technical Support, navigate to the Xilinx Support web page.

Debug Tools

There are many tools available to address PCIe DMA design issues. It is important to know which tools are useful for debugging various situations.

Vivado Design Suite Debug Feature

The Vivado® Design Suite debug feature inserts logic analyzer and virtual I/O cores directly into your design. The debug feature also allows you to set trigger conditions to capture application and integrated block port signals in hardware. Captured signals can then be analyzed. This feature in the Vivado IDE is used for logic debugging and validation of a design running in Xilinx devices.

The Vivado logic analyzer is used with the logic debug IP cores, including:

- ILA 2.0 (and later versions)
- VIO 2.0 (and later versions)

See the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 12].

Reference Boards

Various Xilinx development boards support the PCIe DMA. These boards can be used to prototype designs and establish that the core can communicate with the system.

- 7 series FPGA evaluation boards
  - VC709
- UltraScale FPGA Evaluation boards
  - KCU105
  - VCU108

Hardware Debug

Hardware issues can range from link bring-up to problems seen after hours of testing. This section provides debug steps for common issues. The Vivado debug feature is a valuable resource to use in hardware debug. The signal names mentioned in the following individual sections can be probed using the debug feature for debugging the specific problems.

General Checks

Ensure that all the timing constraints for the core were properly incorporated from the example design and that all constraints were met during implementation.

- Does it work in post-place and route timing simulation? If problems are seen in hardware but not in timing simulation, this could indicate a PCB issue. Ensure that all clock sources are active and clean.
- If using MMCMs in the design, ensure that all MMCMs have obtained lock by monitoring the `locked` port.

Initial Debug of the DMA Subsystem for PCIe

The register in Table B-1 can be used for initial debug of the subsystem. Per channel interface provides important status to the user application.

Table B-1: **Initial Debug of the DMA Subsystem for PCIe**

Bit Index	Field	Description
6	Run	Channel control register run bit.
5	IRQ_Pending	Asserted when the channel has interrupt pending.
4	Packet_Done	On an AXIST interface this bit indicates the last data indicated by the EOP bit has been posted.
3	Descriptor_Done	A descriptor has finished transferring data from the source and posted it to the destination.
2	Descriptor_Completed	Descriptor_Done and "Stop" bit set in the descriptor.
1	Descriptor_Stop	Descriptor_Done and "Completed" bit set in the descriptor.
0	Busy	Channel descriptor buffer is not empty or DMA requests are outstanding.

Appendix C

Additional Resources and Legal Notices

Xilinx Resources

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

References

These documents provide supplemental material useful with this product guide:

1. *AMBA AXI4-Stream Protocol Specification*
2. PCI-SIG Documentation (www.pcisig.com/specifications)
3. *Virtex-7 FPGA Integrated Block for PCI Express LogiCORE IP Product Guide* (PG023)
4. *UltraScale Architecture Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156)
5. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
6. *Vivado Design Suite User Guide: Designing with IP* (UG896)
7. *Vivado Design Suite User Guide: Getting Started* (UG910)
8. *Vivado Design Suite User Guide: Using Constraints* (UG903)
9. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
10. *Vivado Design Suite User Guide: Partial Reconfiguration* (UG909)
11. *ISE to Vivado Design Suite Migration Guide* (UG911)
12. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
13. *Vivado Design Suite User Guide: Implementation* (UG904)
14. *LogiCORE IP AXI Interconnect Product Guide* (PG059)

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/08/2016	2.0	• Identifier Version update • AXI4-Stream Writeback Disable Control bit documented
04/06/2016	2.0	Initial Xilinx release.

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos.

© Copyright 2016 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.