# DMA/Bridge Subsystem for PCI Express v4.1

## *Product Guide*

**Vivado Design Suite**

**PG195 (v4.1) September 21, 2020**

# Table of Contents


**Chapter 1: Introduction** .......... **5**
- Features .......... 5
- IP Facts .......... 6

**Chapter 2: Overview** .......... **7**
- Feature Summary .......... 9
- Applications .......... 9
- Unsupported Features .......... 10
- Limitations .......... 10
- Licensing and Ordering .......... 11

**Chapter 3: Product Specification** .......... **12**
- Standards .......... 12
- Performance and Resource Utilization .......... 12
- Minimum Device Requirements .......... 12
- Configurable Components of the Core .......... 13
- DMA Operations .......... 19
- Port Descriptions .......... 30
- Register Space .......... 43

**Chapter 4: Designing with the Core** .......... **77**
- Clocking and Resets .......... 77
- Tandem Configuration .......... 78

**Chapter 5: Design Flow Steps** .......... **83**
- Customizing and Generating the Subsystem .......... 83
- DMA/Bridge Subsystem for PCI Express® .......... 84
- Output Generation .......... 94
- Constraining the Subsystem .......... 94
- Simulation .......... 96
- Synthesis and Implementation .......... 99

**Chapter 6: Example Design** ..... 100
AXI4 Memory Mapped Default Example Design ..... 101
AXI4 Memory Mapped with PCIe to AXI4-Lite Master and PCIe to DMA Bypass Example Design ..... 102
AXI4 Memory Mapped with AXI4-Lite Slave Interface Example Design ..... 103
AXI4-Stream Example Design ..... 104
AXI4-Memory Mapped with Descriptor Bypass Example ..... 105
Vivado IP Integrator-Based Example Design ..... 106
User IRQ Example Design ..... 109

**Chapter 7: Test Bench** ..... 112
Root Port Model Test Bench for Endpoint ..... 112

**Appendix A: Application Software Development** ..... 120
Linux Device Driver ..... 120
Interrupt Processing ..... 121
Example H2C Flow ..... 122
Example C2H Flow ..... 122

**Appendix B: Upgrading** ..... 124
New Parameters ..... 124
New Ports ..... 124

**Appendix C: Debugging** ..... 127
Finding Help on Xilinx.com ..... 127
Debug Tools ..... 128
Hardware Debug ..... 129

**Appendix D: Using the Xilinx Virtual Cable to Debug** ..... 131
Overview ..... 131
Host PC XVC-Server Application ..... 132
Host PC XVC-over-PCIe Driver ..... 132
XVC-over-PCIe Enabled FPGA Design ..... 133
Using the PCIe-XVC-VSEC Example Design ..... 139

**Appendix E: Additional Resources and Legal Notices** ..... 148
Xilinx Resources ..... 148
Documentation Navigator and Design Hubs ..... 148
References ..... 148

Revision History........................................................................................................149
Please Read: Important Legal Notices.................................................................. 153

# Introduction

The Xilinx® DMA/Bridge Subsystem for PCI Express® (PCIe®) implements a high performance, configurable Scatter Gather DMA for use with the PCI Express® 2.1 and 3.x Integrated Block. The IP provides a choice between an AXI4 Memory Mapped or AXI4-Stream user interface.

This IP optionally also supports a PCIe AXI Bridge mode which is enabled for only UltraScale+™ devices. For details about PCIe AXI Bridge mode operation, see *AXI Bridge for PCI Express Gen3 Subsystem Product Guide* (PG194).

This document covers DMA mode operation only.

## Features

- Supports UltraScale+™, UltraScale™, Virtex®-7 XT Gen3 (Endpoint), and 7 series 2.1 (Endpoint) Integrated Blocks for PCIe. 7A15T and 7A25T are not supported
- Support for 64, 128, 256, 512-bit datapath (64, and 128-bit datapath only for 7 series Gen2 IP)
- 64-bit source, destination, and descriptor addresses
- Up to four host-to-card (H2C/Read) data channels (up to two for 7 series Gen2 IP)
- Up to four card-to-host (C2H/Write) data channels (up to two for 7 series Gen2 IP)
- Selectable user interface
  - Single AXI4 memory mapped (MM) user interface
  - AXI4-Stream user interface (each channel has its own AXI4-Stream interface)
- AXI4 Master and AXI4-Lite Master optional interfaces allow for PCIe traffic to bypass the DMA engine
- AXI4-Lite Slave to access DMA status registers
- Scatter Gather descriptor list supporting unlimited list size
- 256 MB max transfer size per descriptor
- Legacy, MSI, and MSI-X interrupts
- Block fetches of contiguous descriptors

- Poll Mode
- Descriptor Bypass interface
- Arbitrary source and destination address
- Parity check or Propagate Parity on AXI bus (not available for 7 series Gen2 IP)

# IP Facts

| LogiCORE™ IP Facts Table | |
|---|---|
| **Subsystem Specifics** | |
| Supported Device Family[1] | UltraScale+, UltraScale, 7 series Gen2 devices |
| Supported User Interfaces | AXI4 MM, AXI4-Lite, AXI4-Stream |
| Resources | See Resource Utilization web page. |
| **Provided with Subsystem** | |
| Design Files | Encrypted System Verilog |
| Example Design | Verilog |
| Test Bench | Verilog |
| Constraints File | XDC |
| Simulation Model | Verilog |
| Supported S/W Driver | Linux and Windows Drivers[2] |
| **Tested Design Flows[3]** | |
| Design Entry | Vivado® Design Suite |
| Simulation | For supported simulators, see the Xilinx Design Tools: Release Notes Guide. |
| Synthesis | Vivado synthesis |
| **Support** | |
| Release Notes and Known Issues | Master Answer Record: AR 65443 |
| All Vivado IP Change Logs | Master Vivado IP Change Logs: 72775 |
| 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 details, see Appendix A: Application Software Development and AR 65444.
3. For the supported versions of the tools, see the Xilinx Design Tools: Release Notes Guide.

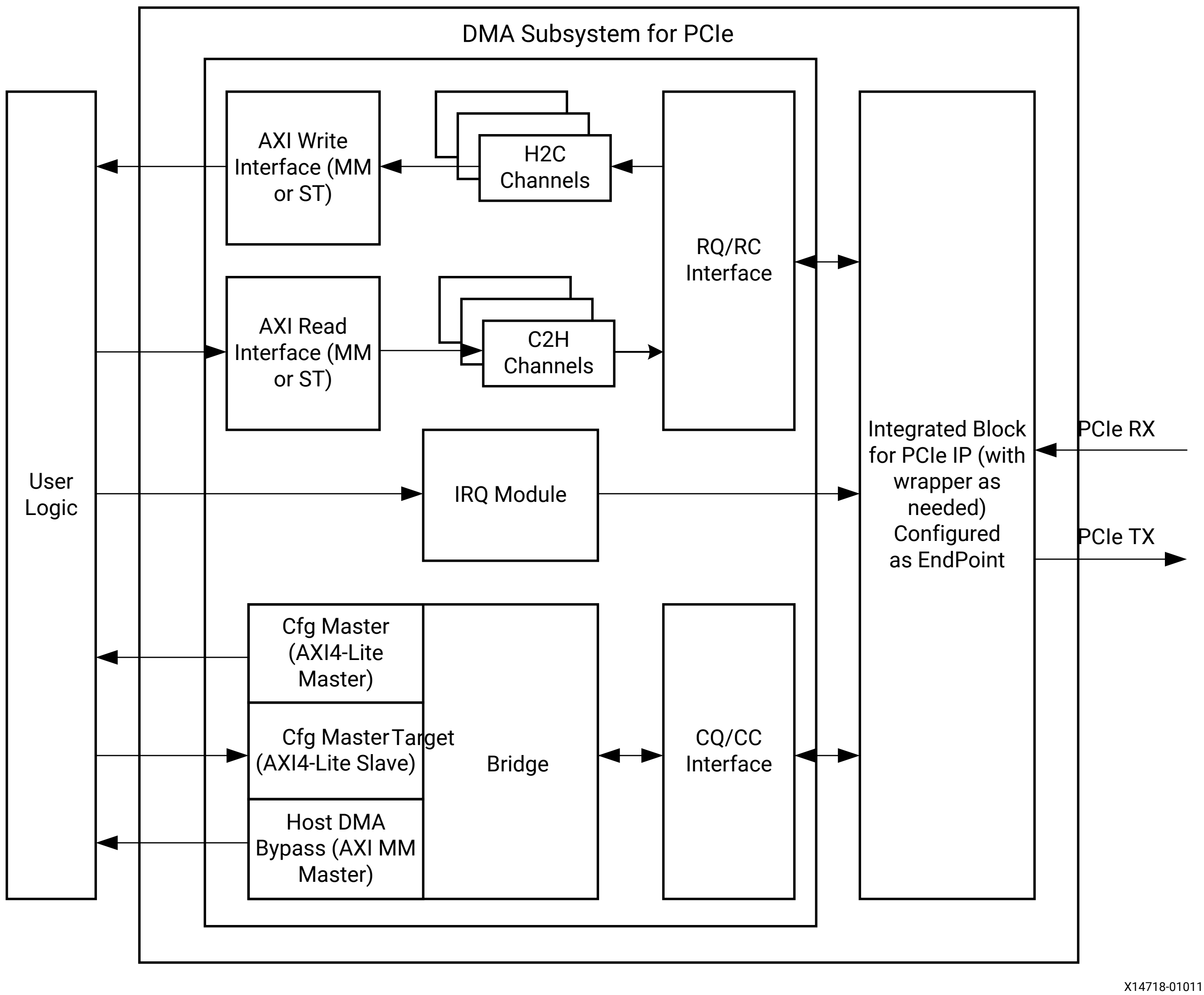
# Overview

The DMA/Bridge Subsystem for PCI Express® (PCIe®) can be configured to be either a high-performance direct memory access (DMA) data mover or a bridge between the PCI Express and AXI memory spaces.

- **DMA Data Mover:** As a DMA, the core can be configured with either an AXI (memory mapped) interface or with an AXI streaming interface to allow for direct connection to RTL logic. Either interface can be used for high performance block data movement between the PCIe address space and the AXI address space using the provided character driver. In addition to the basic DMA functionality, the DMA supports up to four upstream and downstream channels, the ability for PCIe traffic to bypass the DMA engine (Host DMA Bypass), and an optional descriptor bypass to manage descriptors from the FPGA fabric for applications that demand the highest performance and lowest latency.
- **Bridge Between PCIe and AXI Memory:** When configured as a PCIe Bridge, received PCIe packets are converted to AXI traffic and received AXI traffic is converted to PCIe traffic. The bridge functionality is ideal for AXI peripherals needing a quick and easy way to access a PCI Express subsystem. The bridge functionality can be used as either an Endpoint or as a Root Port. PCIe Bridge functionality is only supported for UltraScale+™ devices. Non UltraScale+ devices have specific a PCIe Bridge IP available in the Vivado® IP catalog. For details about PCIe Bridge mode operation, see *AXI Bridge for PCI Express Gen3 Subsystem Product Guide* (PG194).

This document covers DMA mode operation only.

*Figure 1:* **DMA/Bridge Subsystem for PCI Express® Overview**




This diagram refers to the Requester Request (RQ)/Requester Completion (RC) interfaces, and the Completer Request (CQ)/Completer Completion (CC) interfaces. For more information about these, see the *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213).

## Feature Summary

The DMA/Bridge Subsystem for PCI Express® allows for the movement of data between Host memory and the DMA subsystem. It does this by operating on 'descriptors' that contain information about the source, destination and amount of data to transfer. These direct memory transfers can be both in the Host to Card (H2C) and Card to Host (C2H) transfers. The DMA can be configured to have a single AXI4 Master interface shared by all channels or one AXI4-Stream interface for each channel enabled. 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 up to 16 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/Bridge Subsystem for PCIe internal configuration and status registers through an AXI4-Lite Slave Configuration interface. Requests that are mastered on this interface 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 features of the standard are not supported by this core:

- Tandem Configuration solutions (Tandem PROM, Tandem PCIe, Tandem with Field Updates, PR over PCIe) are not supported for Virtex®-7 XT and 7 series Gen2 devices
- Tandem Configuration is not yet supported for Bridge mode in UltraScale+ devices.
- SR-IOV
- ECRC
- Example design not supported for all configurations
- Narrow burst (not supported on the master interface)
- BAR translation for DMA addresses to the AXI4 Memory Mapped interface

# Limitations

## PCIe Transaction Type

The PCIe® transactions that can be generated are those that are compatible with the AXI4 specification. The following table lists the supported PCIe transaction types.

*Table 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 (only PCIe 3.x core)

### Others

- Only supports the INCR burst type. Other types result in a 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)
- For 7 series Gen2 IP, PCIe access from the Host system must be limited to 1DW (4 Bytes) transaction only.

***Note***: Both AXI Bypass and Register access are limited by this restriction.

#### *PCIe to DMA Bypass Master*

- Only issues the INCR burst type
- Only issues the data, non-secure, and unprivileged protection type
- For 7 series Gen2 IP, limited to 1DW (4 Bytes) transaction only

#### *User Interrupt in MSI-X Mode*

Users need to program a different vector number for each user interrupts in the IRQ Block User Vector Number register to generate `acks` for all user interrupts. This generates `acks` for all user interrupts when there are simultaneous interrupts. When all vector numbers are pointing to the same MSI-X entry, there is only one `ack`.

## Licensing and Ordering

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.

*Chapter 3*

# Product Specification

The DMA/Bridge 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.

## Standards

The DMA/Bridge Subsystem for PCIe is compliant with the following specifications:

- *AMBA AXI4-Stream Protocol Specification* (ARM IHI 0051A)
- PCI Express Base Specification v2.1 and v3.0

## Performance and Resource Utilization

For DMA Perfomance data, see AR 68049.

For DMA Resource Utilization, see Resource Utilization web page.

## Minimum Device Requirements

The following table lists the link widths and supported speed for a given speed grade.

*Table 2:* **Minimum Device Requirements**

| Capability Link Speed | Capability Link Width | Supported Speed Grades |
|---|---|---|
| **UltraScale+™ Architecture (PCIE4)** | | |
| Gen1/Gen2 | x1, x2, x4, x8, x16 | -1, -1L, -1LV, -2, -2L, -2LV, -3[1] |
| Gen3 | x1, x2, x4 | -1, -1L, -1LV, -2, -2L, -2LV, -3[1] |
| | x8 | -1, -2, -2L, -2LV, -3[1] |
| | x16 | -1, -2, -2L, -3[1] |

*Table 2:* **Minimum Device Requirements** *(cont'd)*

| Capability Link Speed | Capability Link Width | Supported Speed Grades |
|---|---|---|
| **Virtex® UltraScale+™ Devices with HBM (PCIE4C)[2]** | | |
| Gen1/Gen2 | x1, x2, x4, x8, x16 | -1, -2, -2L, -2LV, -3 |
| Gen3 | x1, x2, x4 | -1, -2, -2L, -2LV, -3 |
| | x8 | -1, -2, -2L, -2LV, -3 |
| | x16 | -1, -2, -2L, -2LV, -3 |
| Gen4[6] | x1, x2, x4, x8 | -2, -2L, -3 |
| **UltraScale™ Devices** | | |
| Gen1 | x1, x2, x4, x8 | -1, -1L, -1LV, -1H, -1HV, -2, -3[3] |
| Gen2 | x1, x2, x4, x8 | -1, -1L, -1LV, -1H, -1HV, -2, -3[3] |
| Gen3 | x1, x2, x4 | -1, -1L, -1LV, -1H, -1HV, -2, -3[3, 4] |
| Gen3 | x8 | -2, -3 |
| **7 series Gen3 Devices** | | |
| Gen1 | x1, x2, x4, x8 | -1, -1M, -1I, -2, -2L, -2G, -2I, -3 |
| Gen2 | x1, x2, x4, x8 | -1, -1M, -1I, -2, -2L, -2G, -2I, -3 |
| Gen3 | x1, x2, x4, x8 | -2, -2L, -2G, -2I, -3 |
| **7 series Gen2 Devices** | | |
| Gen1 | x1, x2, x4, x8 | -1[5], -2[5], -3 |
| Gen2 | x1, x2, x4 | -1[5], -2[5], -3 |
| | x8 | -2[5], -3 |

**Notes:**

1. -1L(0.95V), -1LV(0.90V), -2L(0.85V), -2LV(0.72V).
2. Virtex® UltraScale+™ devices with high bandwidth memory (HBM) contain both PCIE4 and PCIE4C blocks. Only the PCIE4C blocks support Gen3 x16 in the -2LV speed grade.
3. -1L(0.95V), -1LV(0.90V), -1H(1.0V), -1HV(0.95V).
4. The Core Clock Frequency option must be set to 250 MHz for -1, -1LV, -1L, -1H and -1HV speed grades.
5. Available -1 speed grades are -1M, -1I, -1Q depending on family selected. Available -2 speed grades are -2, -2G, -2I, -2IL, -2L depending on the family selected.
6. For Gen4 mode restrictions, see *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213).

# Configurable Components of the Core

Internally, the core can be configured to implement up to eight independent physical DMA engines (up to four H2C and four C2H). 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/Bridge Subsystem for PCI Express® generates requests and expected completions. The AXI4-Stream interface is data-only.

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/Bridge 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 `M_AXI_BYPASS` 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.

When multiple channels for H2C and C2H are enabled, transactions on the AXI4 Master interface are interleaved between all selected channels. Simple round robin protocol is used to service all channels. Transactions granularity depends on host Max Payload Size (MPS), page size, and other host settings.

## 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.

In the following tables, the PCIe BARs selection corresponds to the options set in the PCIe BARs Tab in the Vivado® Integrated Design Environment (IDE).

*Table 3:* **32-Bit BARs**

| PCIe BARs Selection During IP Customization | BAR0 (32-bit) | BAR1 (32-bit) | BAR2 (32-bit) |
|---|---|---|---|
| Default | DMA | | |
| **PCIe to AXI Lite Master** enabled | PCIe to AXI4-Lite Master | DMA | |
| **PCIe to AXI Lite Master** and **PCIe to DMA Bypass** enabled | PCIe to AXI4-Lite Master | DMA | PCIe to DMA Bypass |
| **PCIe to DMA Bypass** enabled | DMA | PCIe to DMA Bypass | |

*Table 4:* **64-Bit BARs**

| PCIe BARs Selection During IP Customization | BAR0 (64-bit) | BAR2 (64-bit) | BAR4 (64-bit) |
|---|---|---|---|
| Default | DMA | | |
| **PCIe to AXI Lite Master** enabled | PCIe to AXI4-Lite Master | DMA | |
| **PCIe to AXI Lite Master** and **PCIe to DMA Bypass** enabled | PCIe to AXI4-Lite Master | DMA | PCIe to DMA Bypass |
| **PCIe to DMA Bypass** enabled | DMA | PCIe to DMA Bypass | |

Different combinations of BARs can be selected. The tables above list only 32-bit selections and 64-bit selections for all BARs as an example. You can select different combinations of BARs based on your requirements.

### Related Information

PCIe BARs Tab

## H2C Channel

The previous tables represents PCIe to AXI4-Lite Master, DMA, and PCIe to DMA Bypass for 32-bit and 64-bit BAR selections. Each space can be individually selected for 32-bits or 64-bits BAR.

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.

When multiple channels are enabled, transactions on the AXI4 Master interface are interleaved between all selected channels. Simple round robin protocol is used to service all channels. Transactions granularity depends on host Max Payload Size (MPS), page size, and other host settings.

## 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 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.

When multiple channels are enabled, transactions on the AXI4 Master interface are interleaved between all selected channels. Simple round robin protocol is used to service all channels. Transactions granularity depends on host MaxPayload Size (MPS), page size, and other host settings.

## 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 to AXI4-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 only. You cannot access the PCIe integrated block register through this interface. This interface does not generate requests to the host.

## 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.

## IRQ Module

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

***Note***: The Host can enable one or more interrupt types from the specified list of supported interrupts during IP configuration. The IP only generates one interrupt type at a given time even when there are more than one enabled. MSI-X interrupt takes precedence over MSI interrupt, and MSI interrupt take precedence over Legacy interrupt. The Host software must not switch (either enable or disable) an interrupt type while there is an interrupt asserted or pending.

### *Legacy Interrupts*

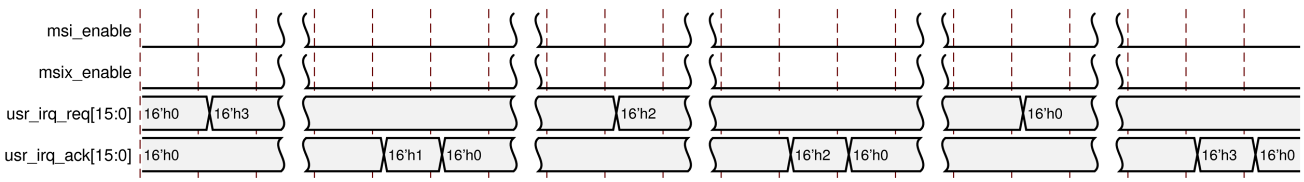
Asserting one or more bits of `usr_irq_req` when legacy interrupts are enabled causes the DMA to issue a legacy interrupt over PCIe. Multiple bits may be asserted simultaneously but each bit must remain asserted until the corresponding usr_irq_ack bit has been asserted. After a `usr_irq_req` bit is asserted, it must remain asserted until the corresponding `usr_irq_ack` bit is asserted and the interrupt has been serviced and cleared by the Host. The `user_irq_ack` assertion indicates the requested interrupt has been sent to the PCIe block. This will ensure interrupt pending register within the IP remains asserted when queried by the Host's Interrupt Service Routine (ISR) to determine the source of interrupts. You must implement a mechanism in the user application to know when the interrupt routine has been serviced. This detection can be done in many different ways depending on your application and your use of this interrupt pin. This typically involves a register (or array of registers) implemented in the user application that is cleared, read, or modified by the Host software when an interrupt is serviced.

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`, and `INTD` through the configuration registers. The following figure shows the legacy interrupts.

***Note***: This figure shows only the handshake between `usr_irq_req` and `usr_irq_ack`. Your application might not clear or service the interrupt immediately, in which case, you must keep `usr_irq_req` asserted past `usr_irq_ack`.

*Figure 2:* **Legacy Interrupts**




## *MSI and MSI-X Interrupts*

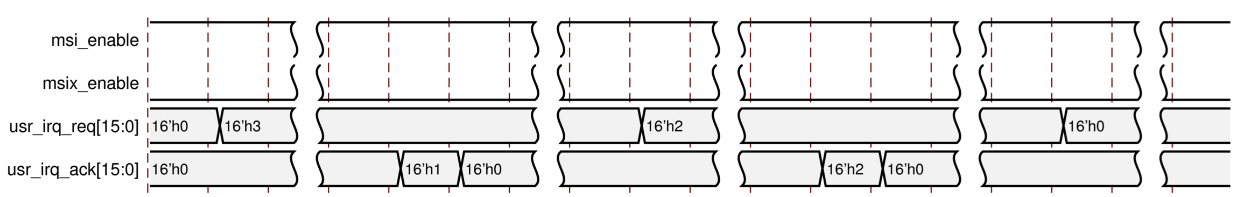
Asserting one or more bits of `usr_irq_req` causes the generation of an MSI or MSI-X interrupt if MSI or MSI-X is enabled. If both MSI and MSI-X capabilities are enabled, an MSI-X interrupt is generated.

After a `usr_irq_req` bit is asserted, it must remain asserted until the corresponding `usr_irq_ack` bit is asserted and the interrupt has been serviced and cleared by the Host. The `user_irq_ack` assertion indicates the requested interrupt has been sent to the PCIe block. This will ensure the interrupt pending register within the IP remains asserted when queried by the Host's Interrupt Service Routine (ISR) to determine the source of interrupts. You must implement a mechanism in the user application to know when the interrupt routine has been serviced. This detection can be done in many different ways depending on your application and your use of this interrupt pin. This typically involves a register (or array of registers) implemented in the user application that is cleared, read, or modified by the Host software when an Interrupt is serviced.

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. The following figure shows the MSI interrupt.

***Note***: This figure shows only the handshake between `usr_irq_req` and `usr_irq_ack`. Your application might not clear or service the interrupt immediately, in which case, you must keep `usr_irq_req` asserted past `usr_irq_ack`.

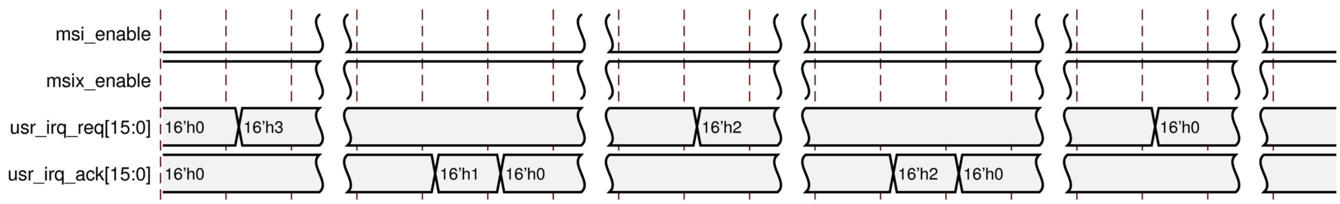
*Figure 3:* **MSI Interrupts**




The following figure shows the MSI-X interrupt.

***Note***: This figure shows only the handshake between `usr_irq_req` and `usr_irq_ack`. Your application might not clear or service the interrupt immediately, in which case, you must keep `usr_irq_req` asserted past `usr_irq_ack`.

*Figure 4:* **MSI-X Interrupts**




For more details, see Interrupt Processing.

## 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/Bridge Subsystem for PCIe. This configuration information can be read through register reads to the appropriate register offset within the config module.

# DMA Operations

## Quick Start

At the most basic level, the PCIe® DMA engine typically moves data between host memory and memory that resides in the FPGA which is often (but not always) on an add-in card. When data is moved from host memory to the FPGA memory, it is called a Host to Card (H2C) transfer or System to Card (S2C) transfer. Conversely, when data is moved from the FPGA memory to the host memory, it is called a Card to Host (C2H) or Card to System (C2S) transfer.

These terms help delineate which way data is flowing (as opposed to using read and write which can get confusing very quickly). The PCIe DMA engine is simply moving data to or from PCIe address locations.

In typical operation, an application in the host must to move data between the FPGA and host memory. To accomplish this transfer, the host sets up buffer space in system memory and creates descriptors that the DMA engine use to move the data.

The contents of the descriptors will depend on a number of factors, including which user interface is chosen for the DMA engine. If an AXI4-Stream interface is selected, C2H transfers do not use the source address field and H2C fields do not use the destination address. This is because the AXI4-Stream interface is a FIFO type interface that does not use addresses.

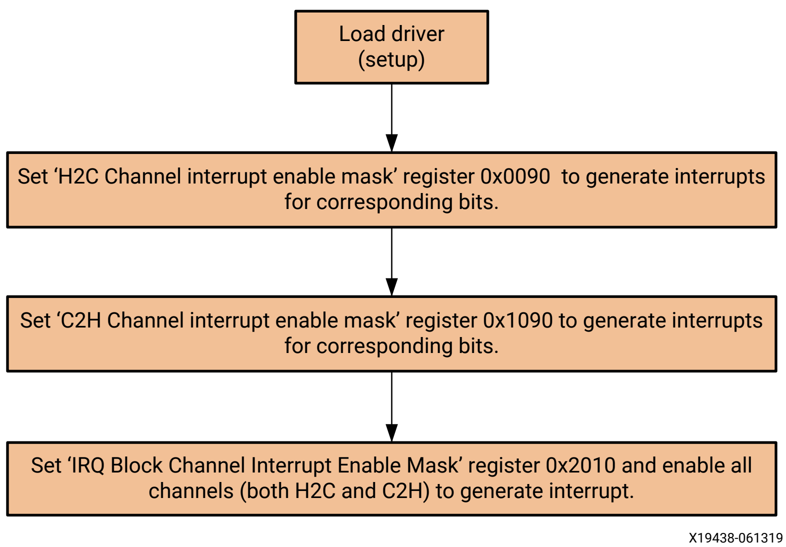
If an AXI Memory Mapped interface is selected, then a C2H transfer has the source address as an AXI address and the destination address is the PCIe address. For a H2C transfer, the source address is a PCIe address and the destination address is an AXI address.

The following flow charts show typical transfers for both H2C and C2H transfers when the data interface is selected during IP configuration for an AXI Memory Mapped interface.

## Initial Setup For H2C and C2H Transfers

The following figure shows the initial setup for both H2C and C2H transfers.

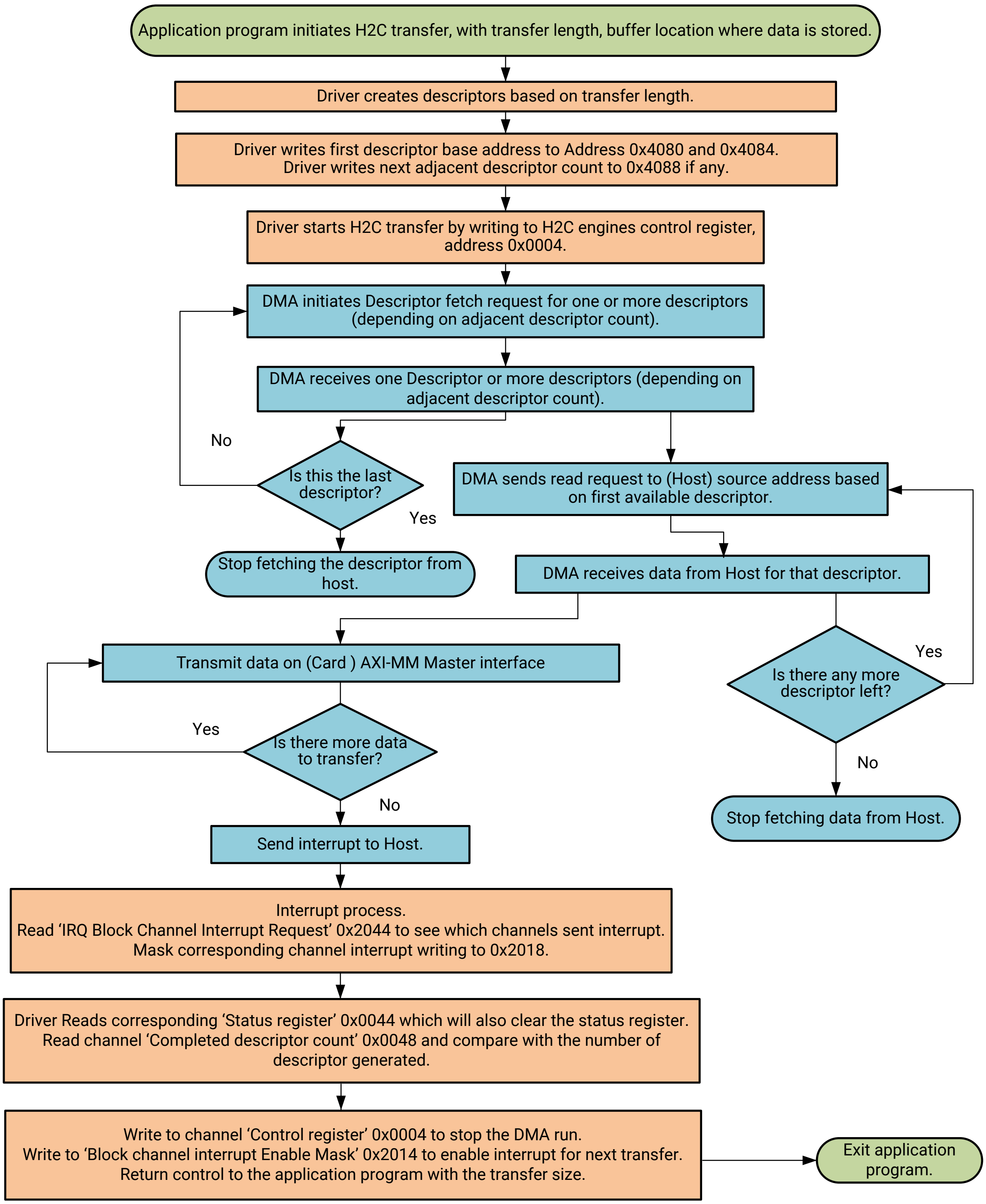
*Figure 5:* **Setup**




## AXI-MM Transfer For H2C

The following figure shows a basic flow chart that explains the data transfer for H2C. The flow chart color coding is as follows: Green is the application program; Orange is the driver; and Blue is the hardware.

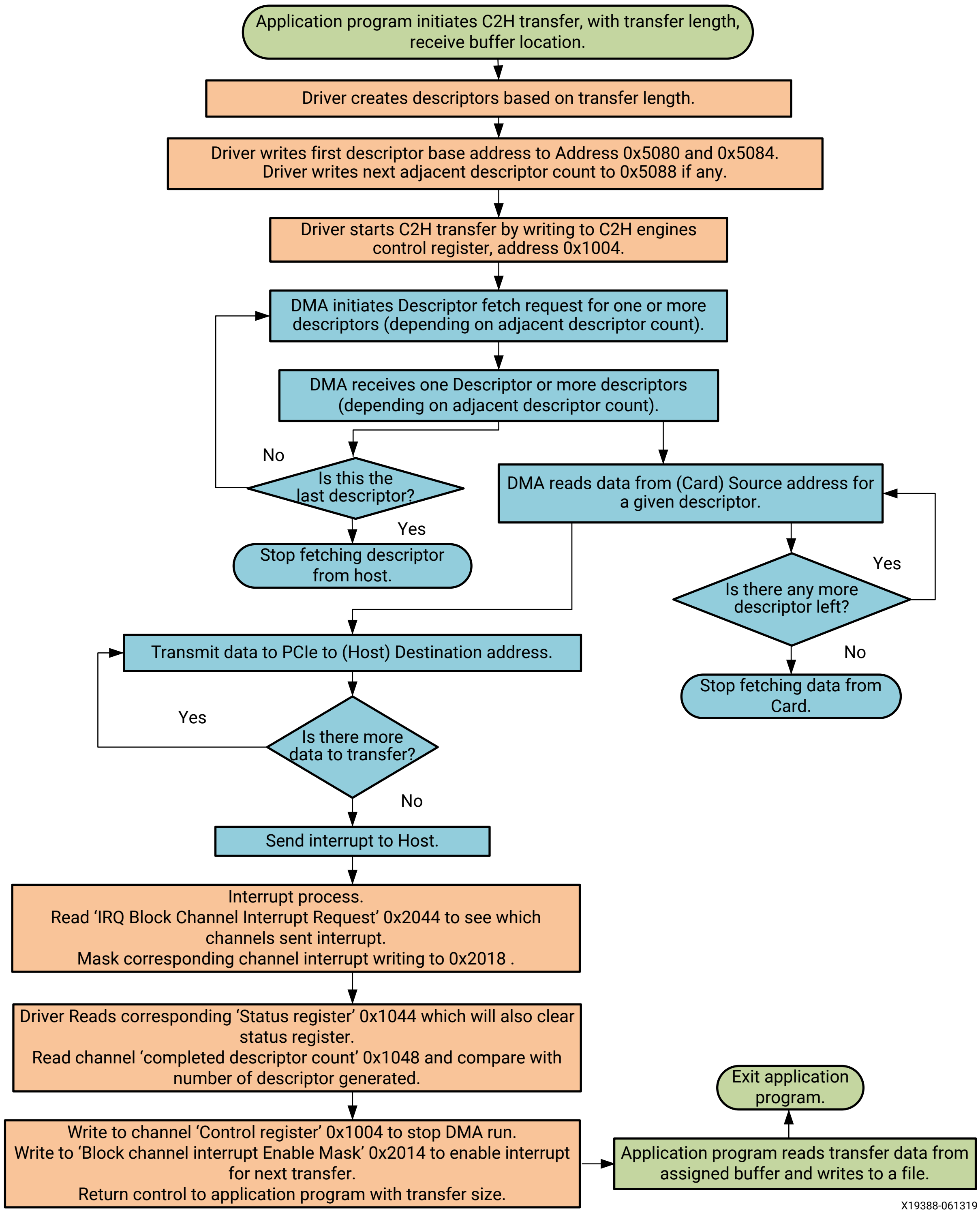
*Figure 6:* **DMA H2C Transfer Summary**



### AXI-MM Transfer For C2H

The following figure shows a basic flow chart that explains the data transfer for C2H. The flow chart color coding is as follows: Green is the application program; Orange is the driver; and Blue is the hardware.

*Figure 7:* **DMA C2H Transfer Summary**



## Descriptors

The DMA/Bridge Subsystem for PCI Express® 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 describe the memory transfers that the DMA/Bridge 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.

***Note*:** Because MRRS in most host systems is 512 bytes or 1024 bytes, having more than 32 adjacent descriptors is not allowed on a single request. However, the design will allow a maximum 64 descriptors in a single block of adjacent descriptors if needed.

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.

When using an AXI4 memory mapped interface, DMA addresses to the card are not translated. If the Host does not know the card address map, the descriptor must be assembled in the user logic and submitted to the DMA using the descriptor bypass interface.

*Table 5:* **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] | | | |

*Table 5:* **Descriptor Format** *(cont'd)*

| Offset | Fields |
|---|---|
| 0x10 | Dst_adr[31:0] |
| 0x14 | Dst_adr[63:32] |
| 0x18 | Nxt_adr[31:0] |
| 0x1C | Nxt_adr[63:32] |

*Table 6:* **Descriptor Fields**

| Offset | Field | Bit Index | Sub Field | Description |
|---|---|---|---|---|
| 0x0 | Magic | 15:0 | | 16'had4b. Code to verify that the driver generated descriptor is valid. |
| 0x0 | | 1:0 | | Reserved set to 0's |
| 0x0 | Nxt_adj | 5:0 | | The number of additional adjacent descriptors after the descriptor located at the next descriptor address field.<br>A block of adjacent descriptors cannot cross a 4k boundary. |
| 0x0 | Control | 5, 6, 7 | | Reserved |
| 0x0 | | 4 | EOP | End of packet for stream interface. |
| 0x0 | | 2, 3 | | Reserved |
| 0x0 | | 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 H2C/C2H Channel control register. |
| 0x0 | | 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. |
| 0x04 | Length | 31:28 | | Reserved set to 0's |
| 0x04 | | 27:0 | | Length of the data in bytes. |

*Table 6:* **Descriptor Fields** *(cont'd)*

| Offset | Field | Bit Index | Sub Field | Description |
|---|---|---|---|---|
| 0x0C-0x8 | Src_adr | 63:0 | | Source address for H2C and memory mapped transfers.<br>Metadata writeback address for C2H transfers. |
| 0x14-0x10 | Dst_adr | 63:0 | | Destination address for C2H and memory mapped transfers. Not used for H2C stream. |
| 0x1C-0x18 | Nxt_adr | 63:0 | | Address of the next descriptor in the list. |

The DMA has `Bit_width` * 512 deep FIFO to hold all descriptors in the descriptor engine. This descriptor FIFO is shared with all selected channels.

- For Gen3x16 with 4H2C and 4C2H design, AXI bit width is 512 bits. FIFO depth is $512\ \text{bit} * 512 = 64\ \text{Bytes} * 512 = 32\ \text{KBytes}$ (1K descriptors). This FIFO is shared by 8 DMA engines.
- For Gen3x8 with 2H2C and 2C2H design, AXI bit width is 256 bits. FIFO depth is $256\ \text{bit} * 512 = 32\ \text{Bytes} * 512 = 16\ \text{KBytes}$ (512 descriptors). This FIFO is shared by 4 DMA engines.

## 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 Control register Run bit must be set. The NextDescriptorAddress and NextAdjacentCount, and Magic descriptor fields are not used when descriptors are bypassed. The `ie_descriptor_stopped` bit in Control register 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 poll host memory to determine when the DMA is complete instead of waiting for an interrupt.

For a given DMA engine, the completed descriptor count writeback occurs when the DMA completes a transfer for a descriptor, and `ie_descriptor_completed` and `Pollmode_wb_enable` are 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 7:* **Completed Descriptor Count Writeback Format**

| Offset | Fields | | |
|---|---|---|---|
| 0x0 | Sts_err | 7'h0 | Compl_descriptor_count[23:0] |

*Table 8:* **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.

Data delivered to the AXI4-Stream interface will be packed for each descriptor. tkeep is all 1s except for the last cycle of a data transfer of the descriptor if it is not a multiple of the datapath width. The DMA does not pack data across multiple descriptors.

# 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 pre-defined WB Magic value `16'h52b4` (Table 10: C2H Stream Writeback Fields), 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.

***Note***: C2H Channel Writeback information is different then Poll mode updates. C2H Channel Writeback information provides the driver current length status of a particular descriptor. This is different from `Pollmode_*`, as is described in Poll Mode.

The `tkeep` bits for transfers for all except the last data transfer of a packet must be all 1s. On the last transfer of a packet, when `tlast` is asserted, you can specify a `tkeep` that is not all 1s to specify a data cycle that is not the full datapath width. The asserted `tkeep` bits need to be packed to the lsb, indicating contiguous data.

The length of a C2H Stream descriptor (the size of the destination buffer) must always be a multiple of 64 bytes.

*Table 9:* **C2H Stream Writeback Format**

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

*Table 10:* **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 11:* **Address Alignment**

| Interface Type | Datapath Width | Address Restriction |
|---|---|---|
| AXI4 MM | 64, 128, 256, 512 | None |
| AXI4-Stream | 64, 128, 256, 512 | None |
| AXI4 MM fixed address[1] | 64 | Source_addr[2:0] == Destination_addr[2:0] == 3’h0 |
| AXI4 MM fixed address[1] | 128 | Source_addr[3:0] == Destination_addr[3:0] == 4’h0 |
| AXI4 MM fixed address[1] | 256 | Source_addr[4:0] == Destination_addr[4:0] == 5’h0 |
| AXI4 MM fixed address[1] | 512 | Source_addr[5:0] == Destination_addr[5:0]==6'h0 |

**Notes:**

1. For fixed address mode, you must set bit [25] in the control registers. Details are found in H2C Channel Control (0x04) and C2H Channel Control (0x04).

### Length Granularity

*Table 12:* **Length Granularity**

| Interface Type | Datapath Width | Length Granularity Restriction |
|---|---|---|
| AXI4 MM | 64, 128, 256, 512 | None |
| AXI4-Stream | 64, 128, 256, 512 | None[1] |
| AXI4 MM fixed address | 64 | Length[2:0] == 3'h0 |
| AXI4 MM fixed address | 128 | Length[3:0] == 4'h0 |
| AXI4 MM fixed address | 256 | Length[4:0] == 5'h0 |
| AXI4 MM fixed address | 512 | Length[5:0] == 6'h0 |

**Notes:**

1. Each C2H descriptor must be sized as a multiple of 64 Bytes. However, there are no restrictions to the total number of Bytes in the actual C2H transfer.

### Parity

Parity checking occurs one of two ways. Set the **Parity Checking** option in the PCIe DMA Tab in the Vivado® IDE during core customization:

When **Check Parity** is enabled, the DMA/Bridge Subsystem for PCIe checks for parity on read data from PCIe, and generates parity for write data to the PCIe.

When **Propagate Parity** is enabled, the DMA/Bridge Subsystem for PCIe propagates parity to the user AXI interface. You are responsible for checking and generating parity in the AXI Interface. Parity is valid every clock cycle when a data valid signal is asserted, and parity bits are valid only for valid data bytes. Parity is calculated for every byte; total parity bits are DATA_WIDTH/8.

- Parity information is sent and received on `*_tuser` ports in AXI4-Stream (AXI_ST) mode.
- Parity information is sent and received on `*_ruser` and `*_wuser` ports in AXI4 Memory Mapped (AXI-MM) mode.

Odd parity is used for parity checking. By default, parity checking is not enabled.

#### Related Information

PCIe DMA Tab

# Port Descriptions

**IMPORTANT!** *This document covers only DMA mode port descriptions. For AXI Bridge mode, see the AXI Bridge for PCI Express Gen3 Subsystem Product Guide (PG194).*

The DMA/Bridge Subsystem for PCI Express® connects directly to the integrated block for PCIe. The datapath interfaces to the PCIe integrated block IP are 64, 128, 256 or 512-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 the following tables.

## Top-Level Interface Signals

*Table 13:* **Top-Level Interface Signals**

| Signal Name | Direction | Description |
|---|---|---|
| sys_clk | I | 7 series Gen2 and Virtex-7 Gen3: PCIe reference clock. Should be driven from the O port of reference clock IBUFDS_GTE2.<br>UltraScale: DRP clock and internal system clock (Half the frequency of sys_clk_gt if PCIe Reference Clock is 250 MHz, otherwise same frequency as sys_clk_gt frequency). Should be driven by the ODIV2 port of reference clock IBUFDS_GTE3. |
| sys_clk_gt | I | UltraScale only: PCIe reference clock. Should be driven from the O port of reference clock IBUFDS_GTE3. See the *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156), or *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213). |
| sys_rst_n | I | Reset from the PCIe edge connector reset signal |
| axi_aclk | O | PCIe derived clock output for m_axi* and s_axi* interfaces. axi_aclk is a derived clock from the TXOUTCLK pin from the GT block; it is not expected to run continuously while axi_aresetn is asserted. |
| axi_aresetn | O | AXI reset signal synchronous with the clock provided on the axi_aclk output. This reset should drive all corresponding AXI Interconnect aresetn signals. |
| dma_bridge_resetn | I | Optional pin and available only when SOFT_RESET_EN parameter is set to TRUE. This pin is intended to be user driven reset when link down, Function Level Reset, Dynamic Function eXchange, or another error condition defined by user occurs. It is not required to be toggled during initial link up operation.<br>When used, all PCIe traffic must be in quiesce state. The signal must be asserted for longer than the Completion Timeout value (typically 50 ms).<br>• 0: Resets all internal Bridge engines and registers as well as asserts the axi_aresetn signal while maintaining PCIe link up.<br>• 1: Normal operation.<br>See Clocking and Resets for further instruction on using this signal. |
| user_lnk_up | O | Output Active-High Identifies that the PCI Express core is linked up with a host device. |

*Table 13:* **Top-Level Interface Signals** *(cont'd)*

| Signal Name | Direction | Description |
|---|---|---|
| msi_enable | O | Indicates when MSI is enabled. |
| msi_vector_width[2:0] | O | Indicates the size of the MSI field (the number of MSI vectors allocated to the device). |
| msix_enable | O | Indicates when MSI-X is enabled. |

### Related Information

Clocking and Resets

## PCIe Interface Signals

*Table 14:* **PCIe Interface Signals**

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

## H2C Channel 0-3 AXI4-Stream Interface Signals

*Table 15:* **H2C Channel 0-3 AXI4-Stream Interface Signals**

| Signal Name[1] | Direction | Description |
|---|---|---|
| m_axis_h2c_tready_*x* | I | 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* | O | 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* [DATA_WIDTH-1:0] | O | Transmit data from the DMA to the user logic. |
| m_axis_h2c_tvalid_*x* | O | The DMA asserts this whenever it is driving valid data on m_axis_h2c_tdata. |
| m_axis_h2c_tuser_*x* [DATA_WIDTH/8-1:0] | O | Parity bits. This port is enabled only in Propagate Parity mode. |

**Notes:**

1. _x in the signal name changes based on the channel number 0, 1, 2, and 3. For example, for channel 0 use the m_axis_h2c_tready_0 port, and for channel 1 use the m_axis_h2c_tready_1 port.

## C2H Channel 0-3 AXI4-Stream Interface Signals

*Table 16:* **C2H Channel 0-3 AXI4-Stream Interface Signals**

| Signal Name[1] | Direction | Description |
|---|---|---|
| s_axis_c2h_tready_*x* | O | Assertion of this signal indicates that the DMA is ready to accept data. Data is transferred across the interface when s_axis_c2h_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* | I | The user logic asserts this signal to indicate the end of the DMA packet. |
| s_axis_c2h_tdata_*x* [DATA_WIDTH-1:0] | I | Transmits data from the user logic to the DMA. |
| s_axis_c2h_tvalid_*x* | I | The user logic asserts this whenever it is driving valid data on s_axis_c2h_tdata. |
| m_axis_c2h_tuser_*x* [DATA_WIDTH/8-1:0] | I | Parity bits. This port is enabled only in Propagate Parity mode. |

**Notes:**

1. _x in the signal name changes based on the channel number 0, 1, 2, and 3. For example, for channel 0 use the m_axis_c2h_tready_0 port, and for channel 1 use the m_axis_c2h_tready_1 port.

## AXI4 Memory Mapped Read Address Interface Signals

*Table 17:* **AXI4 Memory Mapped Read Address Interface Signals**

| Signal Name | Direction | Description |
|---|---|---|
| m_axi_araddr [AXI_ADR_WIDTH-1:0] | O | This signal is the address for a memory mapped read to the user logic from the DMA. |
| m_axi_arid [ID_WIDTH-1:0] | O | Standard AXI4 description, which is found in the AXI4 Protocol Specification. |
| m_axi_arlen[7:0] | O | Master read burst length. |
| m_axi_arsize[2:0] | O | Master read burst size. |
| m_axi_arprot[2:0] | O | 3’h0 |
| m_axi_arvalid | O | The assertion of this signal means there is a valid read request to the address on m_axi_araddr. |
| m_axi_arready | I | Master read address ready. |
| m_axi_arlock | O | 1’b0 |
| m_axi_arcache[3:0] | O | 4’h0 |
| m_axi_arburst | O | Master read burst type. |

## AXI4 Memory Mapped Read Interface Signals

*Table 18:* **AXI4 Memory Mapped Read Interface Signals**

| Signal Name | Direction | Description |
|---|---|---|
| m_axi_rdata [DATA_WIDTH-1:0] | I | Master read data. |
| m_axi_rid [ID_WIDTH-1:0] | I | Master read ID. |
| m_axi_rresp[1:0] | I | Master read response. |
| m_axi_rlast | I | Master read last. |
| m_axi_rvalid | I | Master read valid. |
| m_axi_rready | O | Master read ready. |
| m_axi_ruser [DATA_WIDTH/8-1:0] | I | Parity ports for read interface. This port is enabled only in Propagate Parity mode. |

## AXI4 Memory Mapped Write Address Interface Signals

*Table 19:* **AXI4 Memory Mapped Write Address Interface Signals**

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

## AXI4 Memory Mapped Write Interface Signals

*Table 20:* **AXI4 Memory Mapped Write Interface Signals**

| Signal Name | Direction | Description |
|---|---|---|
| m_axi_wdata [DATA_WIDTH-1:0] | O | Master write data. |
| m_axi_wlast | O | Master write last. |
| m_axi_wstrb | O | Master write strobe. |

*Table 20:* **AXI4 Memory Mapped Write Interface Signals** *(cont'd)*

| Signal Name | Direction | Description |
|---|---|---|
| m_axi_wvalid | O | Master write valid. |
| m_axi_wready | I | Master write ready. |
| m_axi_wuser [DATA_WIDTH/8-1:0] | O | Parity ports for read interface. This port is enabled only in Propagate Parity mode. |

## AXI4 Memory Mapped Write Response Interface Signals

*Table 21:* **AXI4 Memory Mapped Write Response Interface Signals**

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

## AXI4 Memory Mapped Master Bypass Read Address Interface Signals

*Table 22:* **AXI4 Memory Mapped Master Bypass Read Address Interface Signals**

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

## AXI4 Memory Mapped Master Bypass Read Interface Signals

**AXI4 Memory Mapped Master Bypass Read Interface Signals**

| Signal Name | Direction | Description |
|---|---|---|
| m_axib_rdata [DATA_WIDTH-1:0] | I | Master read data. |
| m_axib_rid [ID_WIDTH-1:0] | I | Master read ID. |
| m_axib_rresp[1:0] | I | Master read response. |
| m_axib_rlast | I | Master read last. |
| m_axib_rvalid | I | Master read valid. |
| m_axib_rready | O | Master read ready. |
| m_axib_ruser [DATA_WIDTH/8-1:0] | I | Parity ports for read interface. This port is enabled only in Propagate Parity mode. |

## AXI4 Memory Mapped Master Bypass Write Address Interface Signals

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

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

## AXI4 Memory Mapped Master Bypass Write Interface Signals

**AXI4 Memory Mapped Master Bypass Write Interface Signals**

| Signal Name | Direction | Description |
|---|---|---|
| m_axib_wdata [DATA_WIDTH-1:0] | O | Master write data. |
| m_axib_wlast | O | Master write last. |
| m_axib_wstrb | O | Master write strobe. |
| m_axib_wvalid | O | Master write valid. |
| m_axib_wready | I | Master write ready. |
| m_axib_wuser [DATA_WIDTH/8-1:0] | O | Parity ports for read interface. This port is enabled only in Propagate Parity mode. |

## AXI4 Memory Mapped Master Bypass Write Response Interface Signals

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

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

## Config AXI4-Lite Memory Mapped Write Master Interface Signals

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

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

*Table 25:* **Config AXI4-Lite Memory Mapped Write Master Interface Signals** *(cont'd)*

| Signal Name | Direction | Description |
|---|---|---|
| m_axil_wready | I | Master write ready. |
| m_axil_bvalid | I | Master response valid. |
| m_axil_bready | O | Master response ready. |

# Config AXI4-Lite Memory Mapped Read Master Interface Signals

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

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

# Config AXI4-Lite Memory Mapped Write Slave Interface Signals

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

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

## Config AXI4-Lite Memory Mapped Read Slave Interface Signals

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

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

## Interrupt Interface

*Table 29:* **Interrupt Interface**

| Signal Name | Direction | Description |
|---|---|---|
| usr_irq_req[NUM_USR_IRQ-1:0] | I | Assert to generate an interrupt. Maintain assertion until interrupt is serviced. |
| usr_irq_ack[NUM_USR_IRQ-1:0] | O | Indicates that the interrupt has been sent on PCIe. Two acks are generated for legacy interrupts. One ack is generated for MSI interrupts. |

Each bits in `usr_irq_req`bus corresponds to the same bits in `usr_irq_ack`. For example, `usr_irq_ack[0]` represents an ack for `usr_irq_req[0]`.

## Channel 0-3 Status Ports

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

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

## Configuration Extend Interface Port Descriptions

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

***Note*:** This interface is not available for 7 series Gen2 IP.

*Table 31:* **Configuration Extend Interface Port Descriptions**

| Port | Direction | Width | Description |
|---|---|---|---|
| cfg_ext_read_received | O | 1 | Configuration Extend Read Received<br>The core asserts this output when it has received a configuration read request from the link. When neither user-implemented legacy or extended configuration space is enabled, receipt of a configuration read results in a one-cycle assertion of this signal, together with valid cfg_ext_register_number and cfg_ext_function_number.<br>When user-implemented legacy, extended configuration space, or both are enabled, for the `cfg_ext_register_number` falling below mentioned ranges, this signal is asserted and the user logic must present the cfg_ext_read_data and cfg_ext_read_data_valid.<br>Legacy Space:<br>0xB0-0xBF<br>Extended Configuration space:<br>0xE80 - 0xFFF (UltraScale+ HBM PCIe4C cores only)<br>0x480 - 0x4FF |
| cfg_ext_write_received | O | 1 | Configuration Extend Write Received<br>The core generates a one-cycle pulse on this output when it has received a configuration write request from the link. |
| cfg_ext_register_number | O | 10 | Configuration Extend Register Number<br>The 10-bit address of the configuration register being read or written. The data is valid when cfg_ext_read_received or cfg_ext_write_received is High. |
| cfg_ext_function_number | O | 8 | Configuration Extend Function Number<br>The 8-bit function number corresponding to the configuration read or write request. The data is valid when cfg_ext_read_received or cfg_ext_write_received is High. |
| cfg_ext_write_data | O | 32 | Configuration Extend Write Data<br>Data being written into a configuration register. This output is valid when cfg_ext_write_received is High. |
| cfg_ext_write_byte_enable | O | 4 | Configuration Extend Write Byte Enable Byte enables for a configuration write transaction. |
| cfg_ext_read_data | I | 32 | Configuration Extend Read Data<br>You can provide data from an externally implemented configuration register to the core through this bus. The core samples this data on the next positive edge of the clock after it sets cfg_ext_read_received High, if you have set cfg_ext_read_data_valid. |
| cfg_ext_read_data_valid | I | 1 | Configuration Extend Read Data Valid<br>The user application asserts this input to the core to supply data from an externally implemented configuration register. The core samples this input data on the next positive edge of the clock after it sets cfg_ext_read_received High. |

## Configuration Management Interface Ports

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

*Table 32:* **Configuration Management Interface Ports**

| Port | Direction | Width | Description |
|---|---|---|---|
| cfg_mgmt_addr | I | 19 | Read/Write Address.<br>Configuration Space Dword-aligned address |
| cfg_mgmt_byte_enable | I | 4 | Byte Enable<br>Byte Enable for write data, where cfg_mgmt_byte_enable[0] corresponds to cfg_mgmt_write_data[7:0] and so on |
| cfg_mgmt_read_data | O | 32 | Read Data Out<br>Read data provides the configuration of the Configuration and Management registers |
| cfg_mgmt_read | I | 1 | Read Enable<br>Asserted for a read operation. Active-High |
| cfg_mgmt_read_write_done | O | 1 | Read/Write Operation Complete<br>Asserted for 1 cycle when operation is complete. Active-High |
| cfg_mgmt_write_data | I | 32 | Write data<br>Write data is used to configure the Configuration and Management registers |
| cfg_mgmt_write | I | 1 | Write Enable<br>Asserted for a write operation. Active-High |

## Descriptor Bypass Mode

If in the PCIe DMA Tab in the Vivado IDE either **Descriptor Bypass for Read (H2C)** or **Descriptor Bypass for Write (C2H)** are selected, these ports are present. Each binary bit corresponds to a channel: LSB correspond to Channel 0. Value 1 in bit positions means the corresponding channel descriptor bypass is enabled.

*Table 33:* **H2C 0-3 Descriptor Bypass Port**

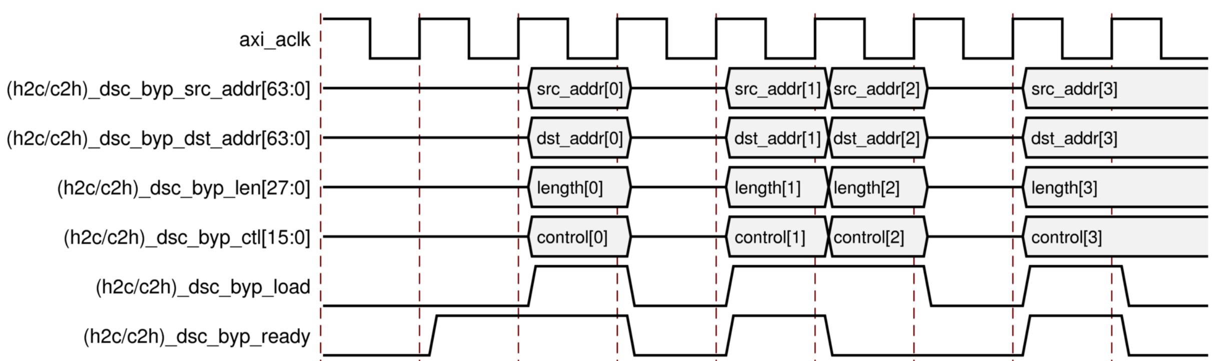
| Port | Direction | Description |
|---|---|---|
| h2c_dsc_byp_ready | O | Channel is ready to accept new descriptors. After h2c_dsc_byp_ready is deasserted, one additional descriptor can be written. The Control register 'Run' bit must be asserted before the channel accepts descriptors. |
| h2c_dsc_byp_load | I | Write the descriptor presented at h2c_dsc_byp_data into the channel's descriptor buffer. |
| h2c_dsc_byp_src_addr[63:0] | I | Descriptor source address to be loaded. |
| h2c_dsc_byp_dst_addr[63:0] | I | Descriptor destination address to be loaded. |
| h2c_dsc_byp_len[27:0] | I | Descriptor length to be loaded. |
| h2c_dsc_byp_ctl[15:0] | I | Descriptor control to be loaded.<br>[0]: Stop. Set to 1 to stop fetching next descriptor.<br>[1]: Completed. Set to 1 to interrupt after the engine has completed this descriptor.<br>[3:2]: Reserved.<br>[4]: EOP. End of Packet for AXI-Stream interface.<br>[15:5]: Reserved.<br>All reserved bits can be forced to 0s. |

*Table 34:* **C2H 0-3 Descriptor Bypass Ports**

| Port | Direction | Description |
|---|---|---|
| c2h_dsc_byp_ready | O | Channel is ready to accept new descriptors. After c2h_dsc_byp_ready is deasserted, one additional descriptor can be written. The Control register 'Run' bit must be asserted before the channel accepts descriptors. |
| c2h_dsc_byp_load | I | Descriptor presented at c2h_dsc_byp_* is valid. |
| c2h_dsc_byp_src_addr[63:0] | I | Descriptor source address to be loaded. |
| c2h_dsc_byp_dst_addr[63:0] | I | Descriptor destination address to be loaded. |
| c2h_dsc_byp_len[27:0] | I | Descriptor length to be loaded. |
| c2h_dsc_byp_ctl[15:0] | I | Descriptor control to be loaded.<br>[0]: Stop. Set to 1 to stop fetching next descriptor.<br>[1]: Completed. Set to 1 to interrupt after the engine has completed this descriptor.<br>[3:2]: Reserved.<br>[4]: EOP. End of Packet for AXI-Stream interface.<br>[15:5]: Reserved.<br>All reserved bits can be forced to 0s. |

The following timing diagram shows how to input the descriptor in descriptor bypass mode. When `dsc_byp_ready` is asserted, a new descriptor can be pushed in with the `dsc_byp_load` signal.

*Figure 8:* **Timing Diagram for Descriptor Bypass Mode**




**IMPORTANT!** *Immediately after `dsc_byp_ready` is deasserted, one more descriptor can be pushed in. In the above timing diagram, a descriptor is pushed in when `dsc_byp_ready` is deasserted.*

## Related Information

PCIe DMA Tab

# Register Space

***Note:*** This document covers only DMA mode register space. For AXI Bridge mode, see the *AXI Bridge for PCI Express Gen3 Subsystem Product Guide* (PG194).

Configuration and status registers internal to the DMA/Bridge Subsystem for PCI Express® 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 Target Bridge.

## PCIe to AXI-Lite Master (BAR0) Address Map

Transactions that hit the PCIe to AXI4-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 AXI4-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 PCIeDMA/Bridge Subsystem for PCI Express® internal configuration register bus. This bus supports 32 bits of address space and 32-bit read and write requests.

DMA/Bridge 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 35:* **PCIe to DMA Address Format**

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

*Table 36:* **PCIe to DMA Address Field Descriptions**

| Bit Index | Field | Description |
|---|---|---|
| 15:12 | Target | The destination submodule within the DMA<br>4’h0: H2C Channels<br>4’h1: C2H Channels<br>4’h2: IRQ Block<br>4’h3: Config<br>4’h4: H2C SGDMA<br>4’h5: C2H SGDMA<br>4’h6: SGDMA Common<br>4'h8: MSI-X |

*Table 36:* **PCIe to DMA Address Field Descriptions** *(cont'd)*

| Bit Index | Field | Description |
|---|---|---|
| 11:8 | Channel ID[3:0] | This field is only applicable for H2C Channel, C2H Channel, H2C SGDMA, and C2H SGDMA Targets. This field indicates which engine is being addressed for these Targets. 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 37:* **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 Registers (0x0)

The H2C channel register space is described in this section.

*Table 38:* **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 (0x0C) |
| 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) |

*Table 38:* **H2C Channel Register Space** *(cont'd)*

| Address (hex) | Register Name |
|---|---|
| 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) |
| 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) |

## H2C Channel Identifier (0x00)

*Table 39:* **H2C Channel Identifier (0x00)**

| Bit Index | Default Value | Access Type | Description |
|---|---|---|---|
| 31:20 | 12’h1fc | RO | Core identifier |
| 19:16 | 4’h0 | RO | H2C Channel Target |
| 15 | 1’b0 | RO | Stream<br>1: AXI4-Stream Interface<br>0: AXI4 Memory Mapped Interface |
| 14:12 | 0 | RO | Reserved |
| 11:8 | Varies | RO | Channel ID Target [3:0] |
| 7:0 | 8'h04 | RO | Version<br>8'h01: 2015.3 and 2015.4<br>8'h02: 2016.1<br>8'h03: 2016.2<br>8'h04: 2016.3<br>8'h05: 2016.4<br>8'h06: 2017.1 to current release |

## H2C Channel Control (0x04)

*Table 40:* **H2C Channel Control (0x04)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:28 | | | Reserved |
| 27 | 1’b0 | RW | When set write back information for C2H in AXI-Stream mode is disabled, default write back is enabled. |

*Table 40:* **H2C Channel Control (0x04)** *(cont'd)*

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 26 | 0x0 | RW | pollmode_wb_enable<br>Poll mode writeback enable.<br>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<br>Non-incrementing address mode. Applies to m_axi_araddr interface only. |
| 24 | | | Reserved |
| 23:19 | 5'h0 | RW | ie_desc_error<br>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<br>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<br>Set to all 1s (0x1F) to enable logging of Status.Read_error and to stop the engine if the error is detected. |
| 8:7 | | | Reserved |
| 6 | 1'b0 | RW | ie_idle_stopped<br>Set to 1 to enable logging of Status.Idle_stopped |
| 5 | 1'b0 | RW | ie_invalid_length<br>Set to 1 to enable logging of Status.Invalid_length |
| 4 | 1'b0 | RW | ie_magic_stopped<br>Set to 1 to enable logging of Status.Magic_stopped |
| 3 | 1'b0 | RW | ie_align_mismatch<br>Set to 1 to enable logging of Status.Align_mismatch |
| 2 | 1'b0 | RW | ie_descriptor_completed<br>Set to 1 to enable logging of Status.Descriptor_completed |
| 1 | 1'b0 | RW | ie_descriptor_stopped<br>Set to 1 to enable logging of Status.Descriptor_stopped |
| 0 | 1'b0 | RW | Run<br>Set to 1 to start the SGDMA engine. Reset to 0 to stop transfer; if the engine is busy it completes the current descriptor. |

**Notes:**

1. The ie_* register bits are interrupt enabled. When these bits are set and the corresponding condition is met, status will be logged in the H2C Channel Status (0x40). When the proper interrupt masks are set (per H2C Channel Interrupt Enable Mask (0x90)), the interrupt will be generated.

## H2C Channel Control (0x08)

*Table 41:* **H2C Channel Control (0x08)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:28 | | | Reserved |

*Table 41:* **H2C Channel Control (0x08)** *(cont'd)*

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 27:0 | | W1S | Control<br>Bit descriptions are the same as in H2C Channel Control (0x04). |

## H2C Channel Control (0x0C)

*Table 42:* **H2C Channel Control (0x0C)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 27:0 | | W1C | Control<br>Bit descriptions are the same as in H2C Channel Control (0x04). |

## H2C Channel Status (0x40)

*Table 43:* **H2C Channel Status (0x40)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:24 | | | Reserved |
| 23:19 | 5’h0 | RW1C | descr_error[4:0]<br>Reset (0) on setting the Control register Run bit.<br>4: Unexpected completion<br>3: Header EP<br>2: Parity error<br>1: Completer abort<br>0: Unsupported request |
| 18:14 | 5’h0 | RW1C | write_error[4:0]<br>Reset (0) on setting the Control register Run bit.<br>Bit position:<br>4-2: Reserved<br>1: Slave error<br>0: Decode error |
| 13:9 | 5’h0 | RW1C | read_error[4:0]<br>Reset (0) on setting the Control register Run bit.<br>Bit position<br>4: Unexpected completion<br>3: Header EP<br>2: Parity error<br>1: Completer abort<br>0: Unsupported request |
| 8:7 | | | Reserved |
| 6 | 1’b0 | RW1C | idle_stopped<br>Reset (0) on setting the Control register Run bit. Set when the engine is idle after resetting the Control register Run bit if the Control register ie_idle_stopped bit is set. |

*Table 43:* **H2C Channel Status (0x40)** *(cont'd)*

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 5 | 1’b0 | RW1C | invalid_length<br>Reset on setting the Control register Run bit. Set when the descriptor length is not a multiple of the data width of an AXI4-Stream channel and the Control register ie_invalid_length bit is set. |
| 4 | 1’b0 | RW1C | magic_stopped<br>Reset on setting the Control register Run bit. Set when the engine encounters a descriptor with invalid magic and stopped if the Control register ie_magic_stopped bit is set. |
| 3 | 1’b0 | RW1C | align_mismatch<br>Source and destination address on descriptor are not properly aligned to each other. |
| 2 | 1’b0 | RW1C | descriptor_completed<br>Reset on setting the Control register Run bit. Set after the engine has completed a descriptor with the COMPLETE bit set if the Control register ie_descriptor_stopped bit is set. |
| 1 | 1’b0 | RW1C | descriptor_stopped<br>Reset on setting Control register Run bit. Set after the engine completed a descriptor with the STOP bit set if the Control register ie_descriptor_stopped bit is set. |
| 0 | 1’b0 | RO | Busy<br>Set if the SGDMA engine is busy. Zero when it is idle. |

## H2C Channel Status (0x44)

*Table 44:* **H2C Channel Status (0x44)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 23:1 | | RC | Status<br>Clear on Read. Bit description is the same as in H2C Channel Status (0x40).<br>Bit 0 cannot be cleared. |

## H2C Channel Completed Descriptor Count (0x48)

*Table 45:* **H2C Channel Completed Descriptor Count (0x48)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:0 | 32’h0 | RO | compl_descriptor_count<br>The number of competed descriptors update by the engine after completing each descriptor in the list.<br>Reset to 0 on rising edge of Control register Run bit (H2C Channel Control (0x04). |

### H2C Channel Alignments (0x4C)

*Table 46:* **H2C Channel Alignments (0x4C)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 23:16 | Configuration based | RO | addr_alignment<br>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<br>The minimum granularity of DMA transfers in bytes. |
| 7:0 | Configuration based | RO | address_bits<br>The number of address bits configured. |

### H2C Poll Mode Low Write Back Address (0x88)

*Table 47:* **H2C Poll Mode Low Write Back Address (0x88)**

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

### H2C Poll Mode High Write Back Address (0x8C)

*Table 48:* **H2C Poll Mode High Write Back Address (0x8C)**

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

### H2C Channel Interrupt Enable Mask (0x90)

*Table 49:* **H2C Channel Interrupt Enable Mask (0x90)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 23:19 | 5’h0 | RW | im_desc_error[4:0]<br>Set to 1 to interrupt when corresponding status register read_error bit is logged. |
| 18:14 | 5’h0 | RW | im_write_error[4:0]<br>set to 1 to interrupt when corresponding status register write_error bit is logged. |
| 13:9 | 5’h0 | RW | im_read_error[4:0]<br>set to 1 to interrupt when corresponding status register read_error bit is logged. |
| 8:7 | | | Reserved |
| 6 | 1’b0 | RW | im_idle_stopped<br>Set to 1 to interrupt when the status register idle_stopped bit is logged. |

*Table 49:* **H2C Channel Interrupt Enable Mask (0x90)** *(cont'd)*

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 5 | 1'b0 | RW | im_invalid_length<br>Set to 1 to interrupt when status register invalid_length bit is logged. |
| 4 | 1'b0 | RW | im_magic_stopped<br>set to 1 to interrupt when status register magic_stopped bit is logged. |
| 3 | 1'b0 | RW | im_align_mismatch<br>set to 1 to interrupt when status register align_mismatch bit is logged. |
| 2 | 1'b0 | RW | im_descriptor_completd<br>set to 1 to interrupt when status register descriptor_completed bit is logged. |
| 1 | 1'b0 | RW | im_descriptor_stopped<br>set to 1 to interrupt when status register descriptor_stopped bit is logged. |

## H2C Channel Interrupt Enable Mask (0x94)

*Table 50:* **H2C Channel Interrupt Enable Mask (0x94)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| | | W1S | Interrupt Enable Mask |

## H2C Channel Interrupt Enable Mask (0x98)

*Table 51:* **H2C Channel Interrupt Enable Mask (0x98)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| | | W1C | Interrupt Enable Mask |

## H2C Channel Performance Monitor Control (0xC0)

*Table 52:* **H2C Channel Performance Monitor Control (0xC0)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 2 | 1'b0 | RW | Run<br>Set to 1 to arm performance counters. Counter starts after the Control register Run bit is set.<br>Set to 0 to halt performance counters. |
| 1 | 1'b0 | WO | Clear<br>Write 1 to clear performance counters. |

*Table 52:* **H2C Channel Performance Monitor Control (0xC0)** *(cont'd)*

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 0 | 1'b0 | RW | Auto<br>Automatically stop performance counters when a descriptor with the stop bit is completed. Automatically clear performance counters when the Control register Run bit is set. Writing 1 to the Performance Monitor Control register Run bit is still required to start the counters. |

## H2C Channel Performance Cycle Count (0xC4)

*Table 53:* **H2C Channel Performance Cycle Count (0xC4)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:0 | 32'h0 | RO | pmon_cyc_count[31:0]<br>Increments once per clock while running. See the Performance Monitor Control register (0xC0) bits Clear and Auto for clearing. |

## H2C Channel Performance Cycle Count (0xC8)

*Table 54:* **H2C Channel Performance Cycle Count (0xC8)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 16 | 1'b0 | RO | pmon_cyc_count_maxed<br>Cycle count maximum was hit. |
| 9:0 | 10'h0 | RO | pmon_cyc_count [41:32]<br>Increments once per clock while running. See the Performance Monitor Control register (0xC0) bits Clear and Auto for clearing. |

## H2C Channel Performance Data Count (0xCC)

*Table 55:* **H2C Channel Performance Data Count (0xCC)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:0 | 32'h0 | RO | pmon_dat_count[31:0]<br>Increments for each valid read data beat while running. See the Performance Monitor Control register (0xC0) bits Clear and Auto for clearing. |

## H2C Channel Performance Data Count (0xD0)

*Table 56:* **H2C Channel Performance Data Count (0xD0)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 16 | 1'b0 | RO | pmon_dat_count_maxed<br>Data count maximum was hit |

*Table 56:* **H2C Channel Performance Data Count (0xD0)** *(cont'd)*

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 15:10 | | | Reserved |
| 9:0 | 10'h0 | RO | pmon_dat_count [41:32]<br>Increments for each valid read data beat while running. See the Performance Monitor Control register (0xC0) bits Clear and Auto for clearing. |

## *C2H Channel Registers (0x1)*

The C2H channel register space is described in this section.

*Table 57:* **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) |
| 0x40 | C2H Channel Status (0x40) |
| 0x44 | 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 | C2H Channel Interrupt Enable Mask (0x90) |
| 0x94 | C2H Channel Interrupt Enable Mask (0x94) |
| 0x98 | C2H Channel Interrupt Enable Mask (0x98) |
| 0xC0 | C2H Channel Performance Monitor Control (0xC0) |
| 0xC4 | C2H Channel Performance Cycle Count (0xC4) |
| 0xC8 | C2H Channel Performance Cycle Count (0xC8) |
| 0xCC | C2H Channel Performance Data Count (0xCC) |
| 0xD0 | C2H Channel Performance Data Count (0xD0) |

### C2H Channel Identifier (0x00)

*Table 58:* **C2H Channel Identifier (0x00)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:20 | 12'h1fc | RO | Core identifier |
| 19:16 | 4'h1 | RO | C2H Channel Target |

*Table 58:* **C2H Channel Identifier (0x00)** *(cont'd)*

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 15 | 1’b0 | RO | Stream<br>1: AXI4-Stream Interface<br>0: AXI4 Memory Mapped Interface |
| 14:12 | 0 | RO | Reserved |
| 11:8 | Varies | RO | Channel ID Target [3:0] |
| 7:0 | 8'h04 | RO | Version<br>8'h01: 2015.3 and 2015.4<br>8'h02: 2016.1<br>8'h03: 2016.2<br>8'h04: 2016.3<br>8'h05: 2016.4<br>8'h06: 2017.1 to current release |

## C2H Channel Control (0x04)

*Table 59:* **C2H Channel Control (0x04)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:28 | | | Reserved |
| 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<br>Poll mode writeback enable.<br>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<br>Non-incrementing address mode. Applies to m_axi_araddr interface only. |
| 23:19 | 5’h0 | RW | ie_desc_error<br>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<br>Set to all 1s (0x1F) to enable logging of Status.Read_error and to stop the engine if the error is detected |
| 8:7 | | | Reserved |
| 6 | 1’b0 | RW | ie_idle_stopped<br>Set to 1 to enable logging of Status.Idle_stopped |
| 5 | 1’b0 | RW | ie_invalid_length<br>Set to 1 to enable logging of Status.Invalid_length |
| 4 | 1’b0 | RW | ie_magic_stopped<br>Set to 1 to enable logging of Status.Magic_stopped |
| 3 | 1’b0 | RW | ie_align_mismatch<br>Set to 1 to enable logging of Status.Align_mismatch |

*Table 59:* **C2H Channel Control (0x04)** *(cont'd)*

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 2 | 1'b0 | RW | ie_descriptor_completed<br>Set to 1 to enable logging of Status.Descriptor_completed |
| 1 | 1'b0 | RW | ie_descriptor_stopped<br>Set to 1 to enable logging of Status.Descriptor_stopped |
| 0 | 1'b0 | RW | Run<br>Set to 1 to start the SGDMA engine. Reset to 0 to stop the transfer, if the engine is busy it completes the current descriptor. |

1. The ie_* register bits are interrupt enabled. When these bits are set and the corresponding condition is met, the status will be logged in the C2H Channel Status (0x40). When proper interrupt masks are set (per C2H Channel Interrupt Enable Mask (0x90) ), the interrupt will be generated.

## C2H Channel Control (0x08)

*Table 60:* **C2H Channel Control (0x08)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| | | W1S | Control<br>Bit descriptions are the same as in C2H Channel Control (0x04). |

## C2H Channel Control (0x0C)

*Table 61:* **C2H Channel Control (0x0C)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| | | W1C | Control<br>Bit descriptions are the same as in C2H Channel Control (0x04). |

## C2H Channel Status (0x40)

*Table 62:* **C2H Channel Status (0x40)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 23:19 | 5'h0 | RW1C | descr_error[4:0]<br>Reset (0) on setting the Control register Run bit.<br>Bit position:<br>4:Unexpected completion<br>3: Header EP<br>2: Parity error<br>1: Completer abort<br>0: Unsupported request |

*Table 62:* **C2H Channel Status (0x40)** *(cont'd)*

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 13:9 | 5'h0 | RW1C | read_error[4:0]<br>Reset (0) on setting the Control register Run bit.<br>Bit position:<br>4-2: Reserved<br>1: Slave error<br>0: Decode error |
| 8:7 | | | Reserved |
| 6 | 1'b0 | RW1C | idle_stopped<br>Reset (0) on setting the Control register Run bit. Set when the engine is idle after resetting the Control register Run bit if the Control register ie_idle_stopped bit is set. |
| 5 | 1'b0 | RW1C | invalid_length<br>Reset on setting the Control register Run bit. Set when the descriptor length is not a multiple of the data width of an AXI4-Stream channel and the Control register ie_invalid_length bit is set. |
| 4 | 1'b0 | RW1C | magic_stopped<br>Reset on setting the Control register Run bit. Set when the engine encounters a descriptor with invalid magic and stopped if the Control register ie_magic_stopped bit is set. |
| 3 | 13'b0 | RW1C | align_mismatch<br>Source and destination address on descriptor are not properly aligned to each other. |
| 2 | 1'b0 | RW1C | descriptor_completed<br>Reset on setting the Control register Run bit. Set after the engine has completed a descriptor with the COMPLETE bit set if the Control register ie_descriptor_completed bit is set. |
| 1 | 1'b0 | RW1C | descriptor_stopped<br>Reset on setting the Control register Run bit. Set after the engine completed a descriptor with the STOP bit set if the Control register ie_magic_stopped bit is set. |
| 0 | 1'b0 | RO | Busy<br>Set if the SGDMA engine is busy. Zero when it is idle. |

## C2H Channel Status (0x44)

*Table 63:* **C2H Channel Status (0x44)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 23:1 | | RC | Status<br>Bit descriptions are the same as in C2H Channel Status (0x40). |

## C2H Channel Completed Descriptor Count (0x48)

*Table 64:* **C2H Channel Completed Descriptor Count (0x48)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:0 | 32'h0 | RO | compl_descriptor_count<br>The number of competed descriptors update by the engine after completing each descriptor in the list.<br>Reset to 0 on rising edge of Control register, run bit (C2H Channel Control (0x04)). |

## C2H Channel Alignments (0x4C)

*Table 65:* **C2H Channel Alignments (0x4C)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 23:16 | varies | RO | addr_alignment<br>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<br>The minimum granularity of DMA transfers in bytes. |
| 7:0 | ADDR_BITS | RO | address_bits<br>The number of address bits configured. |

## C2H Poll Mode Low Write Back Address (0x88)

*Table 66:* **C2H Poll Mode Low Write Back Address (0x88)**

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

## C2H Poll Mode High Write Back Address (0x8C)

*Table 67:* **C2H Poll Mode High Write Back Address (0x8C)**

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

## C2H Channel Interrupt Enable Mask (0x90)

*Table 68:* **C2H Channel Interrupt Enable Mask (0x90)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 23:19 | 5'h0 | RW | im_desc_error[4:0]<br>set to 1 to interrupt when corresponding Status.Read_Error is logged. |
| 13:9 | 5'h0 | RW | im_read_error[4:0]<br>set to 1 to interrupt when corresponding Status.Read_Error is logged. |
| 8:7 | | | Reserved |
| 6 | 1'b0 | RW | im_idle_stopped<br>set to 1 to interrupt when the Status.Idle_stopped is logged. |
| 4 | 1'b0 | RW | im_magic_stopped<br>set to 1 to interrupt when Status.Magic_stopped is logged. |
| 2 | 1'b0 | RW | im_descriptor_completd<br>set to 1 to interrupt when Status.Descriptor_completed is logged. |
| 1 | 1'b0 | RW | im_descriptor_stopped<br>set to 1 to interrupt when Status.Descriptor_stopped is logged. |
| 0 | | | Reserved |

## C2H Channel Interrupt Enable Mask (0x94)

*Table 69:* **C2H Channel Interrupt Enable Mask (0x94)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| | | W1S | Interrupt Enable Mask<br>Bit descriptions are the same as in C2H Channel Interrupt Enable Mask (0x90). |

## C2H Channel Interrupt Enable Mask (0x98)

*Table 70:* **C2H Channel Interrupt Enable Mask (0x98)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| | | W1C | Interrupt Enable Mask<br>Bit Descriptions are the same as in C2H Channel Interrupt Enable Mask (0x90). |

## C2H Channel Performance Monitor Control (0xC0)

*Table 71:* **C2H Channel Performance Monitor Control (0xC0)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 2 | 1’b0 | RW | Run<br>Set to 1 to arm performance counters. Counter starts after the Control register Run bit is set.<br>Set to 0 to halt performance counters. |
| 1 | 1’b0 | WO | Clear<br>Write 1 to clear performance counters. |
| 0 | 1’b0 | RW | Auto<br>Automatically stop performance counters when a descriptor with the stop bit is completed. Automatically clear performance counters when the Control register Run bit is set. Writing 1 to the Performance Monitor Control register Run bit is still required to start the counters. |

## C2H Channel Performance Cycle Count (0xC4)

*Table 72:* **C2H Channel Performance Cycle Count (0xC4)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:0 | 32’h0 | RO | pmon_cyc_count[31:0]<br>Increments once per clock while running. See the Performance Monitor Control register (0xC0) bits Clear and Auto for clearing. |

## C2H Channel Performance Cycle Count (0xC8)

*Table 73:* **C2H Channel Performance Cycle Count (0xC8)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 16 | 1’b0 | RO | pmon_cyc_count_maxed<br>Cycle count maximum was hit. |
| 15:10 | | | Reserved |
| 9:0 | 10’h0 | RO | pmon_cyc_count [41:32]<br>Increments once per clock while running. See the Performance Monitor Control register (0xC0) bits Clear and Auto for clearing. |

### C2H Channel Performance Data Count (0xCC)

*Table 74:* **C2H Channel Performance Data Count (0xCC)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:0 | 32’h0 | RO | pmon_dat_count[31:0]<br>Increments for each valid read data beat while running. See the Performance Monitor Control register (0xC0) bits Clear and Auto for clearing. |

### C2H Channel Performance Data Count (0xD0)

*Table 75:* **C2H Channel Performance Data Count (0xD0)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 16 | 1’b0 | RO | pmon_dat_count_maxed<br>Data count maximum was hit |
| 15:10 | | | Reserved |
| 9:0 | 10’h0 | RO | pmon_dat_count [41:32]<br>Increments for each valid read data beat while running. See the Performance Monitor Control register (0xC0) bits Clear and Auto for clearing. |

## *IRQ Block Registers (0x2)*

The IRQ Block registers are described in this section.

*Table 76:* **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 Channel Interrupt Pending (0x4C) |
| 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) |

*Table 76:* **IRQ Block Register Space** *(cont'd)*

| Address (hex) | Register Name |
|---|---|
| 0xA0 | IRQ Block Channel Vector Number (0xA0) |
| 0xA4 | IRQ Block Channel Vector Number (0xA4) |

Interrupt processing registers are shared between AXI Bridge and AXI DMA. In AXI Bridge mode when **MSI-X Capabilities** is selected, 64 KB address space from the BAR0 is reserved for the MSI-X table. By default, register space is allocated in BAR0. You can select register space in a different BAR, from BAR1 to BAR5, by using the `CONFIG.bar_indicator {BAR0}` Tcl command. This option is valid only when **MSI-X Capabilities** option is selected. There is no allocated space for other interrupt options.

## IRQ Block Identifier (0x00)

*Table 77:* **IRQ Block Identifier (0x00)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:20 | 12’h1fc | RO | Core identifier |
| 19:16 | 4’h2 | RO | IRQ Identifier |
| 15:8 | 8’h0 | RO | Reserved |
| 7:0 | 8'h04 | RO | Version<br>8'h01: 2015.3 and 2015.4<br>8'h02: 2016.1<br>8'h03: 2016.2<br>8'h04: 2016.3<br>8'h05: 2016.4<br>8'h06: 2017.1 to current release |

## IRQ Block User Interrupt Enable Mask (0x04)

*Table 78:* **IRQ Block User Interrupt Enable Mask (0x04)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| [NUM_USR_INT-1:0] | 'h0 | RW | user_int_enmask<br>User Interrupt Enable Mask<br>0: Prevents an interrupt from being generated when the user interrupt source is asserted.<br>1: Generates an interrupt on the rising edge of the user interrupt source. If the Enable Mask is set and the source is already set, a user interrupt will be generated also. |

## IRQ Block User Interrupt Enable Mask (0x08)

*Table 79:* **IRQ Block User Interrupt Enable Mask (0x08)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| | | W1S | user_int_enmask<br>Bit descriptions are the same as in IRQ Block User Interrupt Enable Mask (0x04). |

## IRQ Block User Interrupt Enable Mask (0x0C)

*Table 80:* **IRQ Block User Interrupt Enable Mask (0x0C)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| | | W1C | user_int_enmask<br>Bit descriptions are the same as in IRQ Block User Interrupt Enable Mask (0x04). |

## IRQ Block Channel Interrupt Enable Mask (0x10)

*Table 81:* **IRQ Block Channel Interrupt Enable Mask (0x10)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| [NUM_CHNL-1:0] | 'h0 | RW | channel_int_enmask<br>Engine Interrupt Enable Mask. One bit per read or write engine.<br>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.<br>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. |

## IRQ Block Channel Interrupt Enable Mask (0x14)

*Table 82:* **IRQ Block Channel Interrupt Enable Mask (0x14)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| | | W1S | channel_int_enmask<br>Bit descriptions are the same as in IRQ Block Channel Interrupt Enable Mask (0x10). |

## IRQ Block Channel Interrupt Enable Mask (0x18)

*Table 83:* **IRQ Block Channel Interrupt Enable Mask (0x18)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| | | W1C | channel_int_enmask<br>Bit descriptions are the same as in IRQ Block Channel Interrupt Enable Mask (0x10). |

The following figure shows the packing of H2C and C2H bits.

*Figure 9:* **Packing H2C and C2H**

| 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 |

X15954-010115

## IRQ Block User Interrupt Request (0x40)

*Table 84:* **IRQ Block User Interrupt Request (0x40)**

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

## IRQ Block Channel Interrupt Request (0x44)

*Table 85:* **IRQ Block Channel Interrupt Request (0x44)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| [NUM_CHNL-1:0] | 'h0 | RO | engine_int_req<br>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. The previous figure shows the packing of H2C and C2H bits. |

## IRQ Block User Interrupt Pending (0x48)

*Table 86:* **IRQ Block User Interrupt Pending (0x48)**

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

## IRQ Block Channel Interrupt Pending (0x4C)

*Table 87:* **IRQ Block Channel Interrupt Pending (0x4C)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| [NUM_CHNL-1:0] | 'h0 | RO | engine_int_pend<br>Engine Interrupt Pending.<br>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.<br>The previous figure shows the packing of H2C and C2H bits. |

## IRQ Block User Vector Number (0x80)

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.

*Table 88:* **IRQ Block User Vector Number (0x80)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 28:24 | 5'h0 | RW | vector 3<br>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<br>The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[2]. |
| 12:8 | 5'h0 | RW | vector 1<br>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<br>The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[0]. |

## IRQ Block User Vector Number (0x84)

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.

*Table 89:* **IRQ Block User Vector Number (0x84)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 28:24 | 5'h0 | RW | vector 7<br>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<br>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<br>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<br>The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[4]. |

## IRQ Block User Vector Number (0x88)

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.

*Table 90:* **IRQ Block User Vector Number (0x88)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 28:24 | 5'h0 | RW | vector 11<br>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<br>The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[10]. |
| 12:8 | 5'h0 | RW | vector 9<br>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<br>The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[8]. |

## IRQ Block User Vector Number (0x8C)

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.

*Table 91:* **IRQ Block User Vector Number (0x8C)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 28:24 | 5'h0 | RW | vector 15<br>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<br>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<br>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<br>The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[12]. |

## IRQ Block Channel Vector Number (0xA0)

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 the previous figure. The first C2H vector is after the last H2C vector. For example, if NUM_H2C_Channel = 1, then H2C0 vector is at 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 0xA4, bits [4:0].

*Table 92:* **IRQ Block Channel Vector Number (0xA0)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 28:24 | 5'h0 | RW | vector3<br>The vector number that is used when an interrupt is generated by channel 3. |
| 20:16 | 5'h0 | RW | vector2<br>The vector number that is used when an interrupt is generated by channel 2. |
| 12:8 | 5'h0 | RW | vector1<br>The vector number that is used when an interrupt is generated by channel 1. |
| 4:0 | 5'h0 | RW | vector0<br>The vector number that is used when an interrupt is generated by channel 0. |

## IRQ Block Channel Vector Number (0xA4)

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 the previous figure. The first C2H vector is after the last H2C vector. For example, if NUM_H2C_Channel = 1, then H2C0 vector is at 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 0xA4, bits [4:0].

*Table 93:* **IRQ Block Channel Vector Number (0xA4)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 28:24 | 5'h0 | RW | vector7<br>The vector number that is used when an interrupt is generated by channel 7. |
| 20:16 | 5'h0 | RW | vector6<br>The vector number that is used when an interrupt is generated by channel 6. |
| 12:8 | 5'h0 | RW | vector5<br>The vector number that is used when an interrupt is generated by channel 5. |
| 4:0 | 5'h0 | RW | vector4<br>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 94:* **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) |
| 0x40 | Config AXI User Max Payload Size (0x40) |
| 0x44 | Config AXI User Max Read Request Size (0x44) |
| 0x60 | Config Write Flush Timeout (0x60) |

### Config Block Identifier (0x00)

*Table 95:* **Config Block Identifier (0x00)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:20 | 12'h1fc | RO | Core identifier |

*Table 95:* **Config Block Identifier (0x00)** *(cont'd)*

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 19:16 | 4'h3 | RO | Config identifier |
| 15:8 | 8'h0 | RO | Reserved |
| 7:0 | 8'h04 | RO | Version<br>8'h01: 2015.3 and 2015.4<br>8'h02: 2016.1<br>8'h03: 2016.2<br>8'h04: 2016.3<br>8'h05: 2016.4<br>8'h06: 2017.1 to current release |

## Config Block BusDev (0x04)

*Table 96:* **Config Block BusDev (0x04)**

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

## Config Block PCIE Max Payload Size (0x08)

*Table 97:* **Config Block PCIE Max Payload Size (0x08)**

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

## Config Block PCIE Max Read Request Size (0x0C)

*Table 98:* **Config Block PCIE Max Read Request Size (0x0C)**

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

## Config Block System ID (0x10)

*Table 99:* **Config Block System ID (0x10)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| [15:0] | 16’hff01 | RO | system_id<br>Core system ID |

## Config Block MSI Enable (0x14)

*Table 100:* **Config Block MSI Enable (0x14)**

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

## Config Block PCIE Data Width (0x18)

*Table 101:* **Config Block PCIE Data Width (0x18)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| [2:0] | C_DAT_WIDTH | RO | pcie_width<br>PCIe AXI4-Stream Width<br>0: 64 bits<br>1: 128 bits<br>2: 256 bits<br>3: 512 bits |

## Config PCIE Control (0x1C)

*Table 102:* **Config PCIE Control (0x1C)**

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

## Config AXI User Max Payload Size (0x40)

*Table 103:* **Config AXI User Max Payload Size (0x40)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 6:4 | 3'h5 | RO | user_eff_payload<br>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.<br>3'b000: 128 bytes<br>3'b001: 256 bytes<br>3'b010: 512 bytes<br>3'b011: 1024 bytes<br>3'b100: 2048 bytes<br>3'b101: 4096 bytes |
| 3 | | | Reserved |
| 2:0 | 3'h5 | RW | user_prg_payload<br>The programmed maximum payload size issued to the user application.<br>3'b000: 128 bytes<br>3'b001: 256 bytes<br>3'b010: 512 bytes<br>3'b011: 1024 bytes<br>3'b100: 2048 bytes<br>3'b101: 4096 bytes |

## Config AXI User Max Read Request Size (0x44)

*Table 104:* **Config AXI User Max Read Request Size (0x44)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 6:4 | 3'h5 | RO | user_eff_read<br>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.<br>3'b000: 128 bytes<br>3'b001: 256 bytes<br>3'b010: 512 bytes<br>3'b011: 1024 bytes<br>3'b100: 2048 bytes<br>3'b101: 4096 bytes |

*Table 104:* **Config AXI User Max Read Request Size (0x44)** *(cont'd)*

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 3 | | | Reserved |
| 2:0 | 3’h5 | RW | user_prg_read<br>Maximum read request size issued to the user application.<br>3'b000: 128 bytes<br>3'b001: 256 bytes<br>3'b010: 512 bytes<br>3'b011: 1024 bytes<br>3'b100: 2048 bytes<br>3'b101: 4096 bytes |

### Config Write Flush Timeout (0x60)

*Table 105:* **Config Write Flush Timeout (0x60)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 4:0 | 5’h0 | RW | Write Flush Timeout<br>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)*

*Table 106:* **H2C SGDMA Registers (0x4)**

| 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) |
| 0x8C | H2C SGDMA Descriptor Credits (0x8C) |

### H2C SGDMA Identifier (0x00)

*Table 107:* **H2C SGDMA Identifier (0x00)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:20 | 12’h1fc | RO | Core identifier |
| 19:16 | 4’h4 | RO | H2C DMA Target |

*Table 107:* **H2C SGDMA Identifier (0x00)** *(cont'd)*

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 15 | 1’b0 | RO | Stream<br>1: AXI4-Stream Interface<br>0: AXI4 Memory Mapped Interface |
| 14:12 | 3’h0 | RO | Reserved |
| 11:8 | Varies | RO | Channel ID Target [3:0] |
| 7:0 | 8'h04 | RO | Version<br>8'h01: 2015.3 and 2015.4<br>8'h02: 2016.1<br>8'h03: 2016.2<br>8'h04: 2016.3<br>8'h05: 2016.4<br>8'h06: 2017.1 to current release |

## H2C SGDMA Descriptor Low Address (0x80)

*Table 108:* **H2C SGDMA Descriptor Low Address (0x80)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:0 | 32’h0 | RW | dsc_adr[31:0]<br>Lower bits of start descriptor address. Dsc_adr[63:0] is the first descriptor address that is fetched after the Control register Run bit is set. |

## H2C SGDMA Descriptor High Address (0x84)

*Table 109:* **H2C SGDMA Descriptor High Address (0x84)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:0 | 32’h0 | RW | dsc_adr[63:32]<br>Upper bits of start descriptor address.<br>Dsc_adr[63:0] is the first descriptor address that is fetched after the Control register Run bit is set. |

## H2C SGDMA Descriptor Adjacent (0x88)

*Table 110:* **H2C SGDMA Descriptor Adjacent (0x88)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 5:0 | 6’h0 | RW | dsc_adj[5:0]<br>Number of extra adjacent descriptors after the start descriptor address. |

### H2C SGDMA Descriptor Credits (0x8C)

*Table 111:* **H2C SGDMA Descriptor Credits (0x8C)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 9:0 | 10'h0 | RW | h2c_dsc_credit[9:0]<br>Writes to this register will add descriptor credits for the channel. This register will only be used if it is enabled via the channel's bits in the Descriptor Credit Mode register.<br>Credits are automatically cleared on the falling edge of the channels Control register Run bit or if Descriptor Credit Mode is disabled for the channel. The register can be read to determine the number of current remaining credits for the channel. |

## *C2H SGDMA Registers (0x5)*

The C2H SGDMA registers are described in this section.

*Table 112:* **C2H SGDMA Registers (0x5)**

| 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) |
| 0x8C | C2H SGDMA Descriptor Credits (0x8C) |

### C2H SGDMA Identifier (0x00)

*Table 113:* **C2H SGDMA Identifier (0x00)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:20 | 12’h1fc | RO | Core identifier |
| 19:16 | 4’h5 | RO | C2H DMA Target |
| 15 | 1’b0 | RO | Stream<br>1: AXI4-Stream Interface<br>0: AXI4 Memory Mapped Interface |
| 14:12 | 3’h0 | RO | Reserved |
| 11:8 | Varies | RO | Channel ID Target [3:0] |
| 7:0 | 8'h04 | RO | Version<br>8'h01: 2015.3 and 2015.4<br>8'h02: 2016.1<br>8'h03: 2016.2<br>8'h04: 2016.3<br>8'h05: 2016.4<br>8'h06: 2017.1 to current release |

## C2H SGDMA Descriptor Low Address (0x80)

*Table 114:* **C2H SGDMA Descriptor Low Address (0x80)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:0 | 32'h0 | RW | dsc_adr[31:0]<br>Lower bits of start descriptor address. Dsc_adr[63:0] is the first descriptor address that is fetched after the Control register Run bit is set. |

## C2H SGDMA Descriptor High Address (0x84)

*Table 115:* **C2H SGDMA Descriptor High Address (0x84)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:0 | 32'h0 | RW | dsc_adr[63:32]<br>Upper bits of start descriptor address.<br>Dsc_adr[63:0] is the first descriptor address that is fetched after the Control register Run bit is set. |

## C2H SGDMA Descriptor Adjacent (0x88)

*Table 116:* **C2H SGDMA Descriptor Adjacent (0x88)**

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

## C2H SGDMA Descriptor Credits (0x8C)

*Table 117:* **C2H SGDMA Descriptor Credits (0x8C)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 9:0 | 10'h0 | RW | c2h_dsc_credit[9:0]<br>Writes to this register will add descriptor credits for the channel. This register is only used if it is enabled through the channel's bits in the Descriptor Credit Mode register.<br>Credits are automatically cleared on the falling edge of the channels Control register Run bit or if Descriptor Credit Mode is disabled for the channel. The register can be read to determine the number of current remaining credits for the channel. |

### SGDMA Common Registers (0x6)

*Table 118:* **SGDMA Common Registers (0x6)**

| 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) |
| 0x20 | SGDMA Descriptor Credit Mode Enable (0x20) |
| 0x24 | SG Descriptor Mode Enable Register (0x24) |
| 0x28 | SG Descriptor Mode Enable Register (0x28) |

#### SGDMA Identifier Registers (0x00)

*Table 119:* **SGDMA Identifier Registers (0x00)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 31:20 | 12’h1fc | RO | Core identifier |
| 19:16 | 4’h6 | RO | SGDMA Target |
| 15:8 | 8’h0 | RO | Reserved |
| 7:0 | 8'h04 | RO | Version<br>8'h01: 2015.3 and 2015.4<br>8'h02: 2016.1<br>8'h03: 2016.2<br>8'h04: 2016.3<br>8'h05: 2016.4<br>8'h06: 2017.1 to current release |

#### SGDMA Descriptor Control Register (0x10)

*Table 120:* **SGDMA Descriptor Control Register (0x10)**

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

### SGDMA Descriptor Control Register (0x14)

*Table 121:* **SGDMA Descriptor Control Register (0x14)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| | | W1S | Bit descriptions are the same as in SGDMA Descriptor Control Register (0x10). |

### SGDMA Descriptor Control Register (0x18)

*Table 122:* **SGDMA Descriptor Control Register (0x18)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| | | W1C | Bit descriptions are the same as in SGDMA Descriptor Control Register (0x10). |

### SGDMA Descriptor Credit Mode Enable (0x20)

*Table 123:* **SGDMA Descriptor Credit Mode Enable (0x20)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| 3:0 | 0x0 | RW | h2c_dsc_credit_enable [3:0]<br>One bit per H2C channel. Set to 1 to enable descriptor crediting. For each channel, the descriptor fetch engine will limit the descriptors fetched to the number of descriptor credits it is given through writes to the channel's Descriptor Credit Register. |
| 15:4 | | | Reserved |
| 19:16 | 0x0 | RW | c2h_dsc_credit_enable [3:0]<br>One bit per C2H channel. Set to 1 to enable descriptor crediting. For each channel, the descriptor fetch engine will limit the descriptors fetched to the number of descriptor credits it is given through writes to the channel's Descriptor Credit Register. |

### SG Descriptor Mode Enable Register (0x24)

*Table 124:* **SG Descriptor Mode Enable Register (0x24)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| | | W1S | Bit descriptions are the same as in SGDMA Descriptor Credit Mode Enable (0x20). |

### SG Descriptor Mode Enable Register (0x28)

*Table 125:* **SG Descriptor Mode Enable Register (0x28)**

| Bit Index | Default | Access Type | Description |
|---|---|---|---|
| | | W1C | Bit descriptions are the same as in SGDMA Descriptor Credit Mode Enable (0x20). |

## *MSI-X Vector Table and PBA (0x8)*

The MSI-X Vector table and PBA are described in the following table. MSI-X table offsets start at 0x8000. The table below shows two MSI-X vector entries (MSI-X table has 32 vector entries). PBA address offsets start at 0x8FE0. Address offsets are fixed values.

***Note*:** The MSI-X enable in configuration control register should be asserted before writing to MSI-X table. If not, the MSI-X table will not work as expected.

*Table 126:* **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]<br>MSI-X vector0 message lower address. |
| 0x04 | 31:0 | 32’h0 | RW | MSIX_Vector0_Address[63:32]<br>MSI-X vector0 message upper address. |
| 0x08 | 31:0 | 32’h0 | RW | MSIX_Vector0_Data[31:0]<br>MSI-X vector0 message data. |
| 0x0C | 31:0 | 32'hFFFFFFFF | RW | MSIX_Vector0_Control[31:0]<br>MSI-X vector0 control.<br>Bit Position:<br>31:1: Reserved.<br>0: Mask. When set to 1, this MSI-X vector is not used to generate a message. When reset to 0, this MSI-X Vector is used to generate a message. |
| 0x1F0 | 31:0 | 32’h0 | RW | MSIX_Vector31_Address[31:0]<br>MSI-X vector31 message lower address. |
| 0x1F4 | 31:0 | 32’h0 | RW | MSIX_Vector31_Address[63:32]<br>MSI-X vector31 message upper address. |
| 0x1F8 | 31:0 | 32’h0 | RW | MSIX_Vector31_Data[31:0]<br>MSI-X vector31 message data. |
| 0x1FC | 31:0 | 32'hFFFFFFFF | RW | MSIX_Vector31_Control[31:0]<br>MSI-X vector31 control.<br>Bit Position:<br>31:1: Reserved.<br>0: Mask. When set to one, this MSI-X vector is not used to generate a message. When reset to 0, this MSI-X Vector is used to generate a message. |
| 0xFE0 | 31:0 | 32’h0 | RW | Pending_Bit_Array[31:0]<br>MSI-X Pending Bit Array. There is one bit per vector. Bit 0 corresponds to vector0, etc. |

Chapter 4

# Designing with the Core

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

## Clocking and Resets

### Clocking

The `axi_aclk` output is the clock used for all AXI interfaces and should drive all corresponding AXI Interconnect `aclk` signals. `axi_aclk`is not a free running clock. This is a derived clock and will be valid after signal `axi_aresetn` is de-asserted

***Note***: The `axi_aclk` output should not be used for the system clock for your design. The `axi_aclk` is not a free-run clock output. As noted, `axi_aclk` may not be present at all times.

### Resets

For the DMA/ Bridge Subsystem for PCIe in AXI Bridge mode, there is an optional `dma_bridge_resetn` input pin which allows you to reset all internal Bridge engines and registers as well as all AXI peripherals driven by `axi_aresetn` pin. When the following parameter is set, `dma_bridge_resetn` does not need to be asserted during initial link up operation because it will be done automatically by the IP. You must terminate all transactions before asserting this pin. After being asserted, the pin must be kept asserted for a minimum duration of at least equal to the Completion Timeout value (typically 50 ms) to clear any pending transfer that may currently be queued in the data path. To set this parameter, type the following command at the Tcl command line:

```
set_property -dict [list CONFIG.soft_reset_en {true} [get_ips <ip_name>]
```

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

- *7 Series FPGAs Integrated Block for PCI Express LogiCORE IP Product Guide* (PG054)
- *Virtex-7 FPGA Integrated Block for PCI Express LogiCORE IP Product Guide* (PG023)
- *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156)

- *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213)

# Tandem Configuration

Tandem Configuration features are available for the Xilinx® DMA Subsystem for PCI Express® core for all UltraScale™ and most 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 (UltraScale only) or Tandem PCIe initial configuration, update the entire user design while the PCIe link remains active. The update region (floorplan) and design structure are predefined, and Tcl scripts are provided.
- **Tandem + Dynamic Function eXchange:** This is a more general case of Tandem Configuration followed by Dynamic Function eXchange (DFX) of any size or number of dynamic regions.
- **Dynamic Function eXchange over PCIe:** This is a standard configuration followed by DFX, using the PCIe/MCAP as the delivery path of partial bitstreams.

For information on Dynamic Function eXchange, see the *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909).

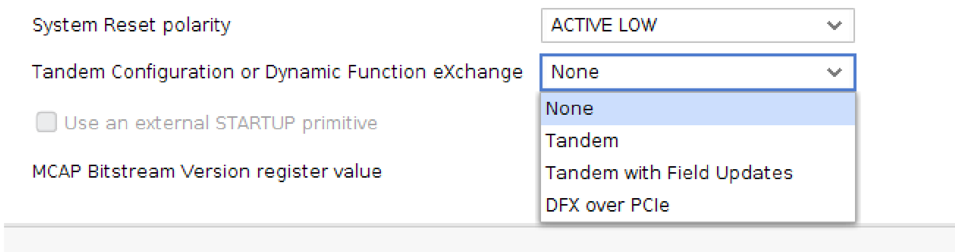
## Customizing the Core for Tandem Configuration

### *UltraScale Devices*

To enable any of the Tandem Configuration capabilities for UltraScale™ devices, select the appropriate Vivado® IP catalog option when customizing the core. In the Basic tab:

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

*Figure 10:* **Tandem Configuration or Dynamic Function eXchangeOptions for UltraScale Devices**



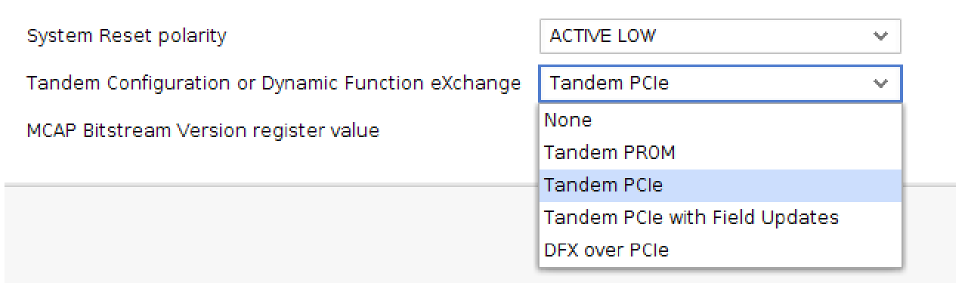

For complete information about Tandem Configuration, including required PCIe block locations, design flow examples, requirements, restrictions and other considerations, see Tandem Configuration in the *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156).

## *UltraScale+ Devices*

To enable any of the Tandem Configuration capabilities for UltraScale+™ devices, select the appropriate IP catalog option when customizing the core. In the Basic tab:

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

*Figure 11:* **Tandem Configuration or Dynamic Function eXchange Option**




**IMPORTANT!** *Tandem Configuration is currently supported for DMA mode, and is not supported for Bridge mode in UltraScale+ devices.*

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+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213).

# Supported Devices

The DMA/Bridge Subsystem for PCIe® core and Vivado® tool flow support implementations targeting Xilinx® reference boards and specific part/package combinations. Tandem configuration supports the configurations found in the following tables.

## *UltraScale Devices*

The following table lists the Tandem PROM/PCIe supported configurations for UltraScale™ devices.

| **HDL** | Verilog Only | | | | |
|---|---|---|---|---|---|
| **PCIe Configuration** | All configurations (max: X8Gen3) | | | | |
| **Xilinx Reference Board Support** | KCU105 Evaluation Board for Kintex® UltraScale™ FPGA<br>VCU108 Evaluation Board for Virtex® UltraScale™ FPGA | | | | |
| **Device Support** | **Part**[1] | **PCIe Block Location** | **PCIe Reset Location** | **Tandem Configuration** | **Tandem with Field Updates** |
| Kintex UltraScale | XCKU025 | PCIE_3_1_X0Y0 | IOB_X1Y103 | Production | Production |
| | XCKU035 | PCIE_3_1_X0Y0 | IOB_X1Y103 | Production | Production |
| | XCKU040 | PCIE_3_1_X0Y0 | IOB_X1Y103 | Production | Production |
| | XCKU060 | PCIE_3_1_X0Y0 | IOB_X2Y103 | Production | Production |
| | XCKU085 | PCIE_3_1_X0Y0 | IOB_X2Y103 | Production | Production |
| | XCKU095 | PCIE_3_1_X0Y0 | IOB_X1Y103 | Production | Production |
| | XCKU115 | PCIE_3_1_X0Y0 | IOB_X2Y103 | Production | Production |
| Virtex UltraScale | XCVU065 | PCIE_3_1_X0Y0 | IOB_X1Y103 | Production | Production |
| | XCVU080 | PCIE_3_1_X0Y0 | IOB_X1Y103 | Production | Production |
| | XCVU095 | PCIE_3_1_X0Y0 | IOB_X1Y103 | Production | Production |
| | XCVU125 | PCIE_3_1_X0Y0 | IOB_X1Y103 | Production | Production |
| | XCVU160 | PCIE_3_1_X0Y1 | IOB_X1Y363 | Production | Production |
| | XCVU190 | PCIE_3_1_X0Y2 | IOB_X1Y363 | Production | Production |
| | XCVU440 | PCIE_3_1_X0Y2 | IOB_X1Y363 | Production | Production |

**Notes:**

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

## *UltraScale+ Devices*

The following table lists the Tandem PROM/PCIe supported configurations for UltraScale+ devices.

| **HDL** | Verilog Only | | | |
|---|---|---|---|---|
| **PCIe Configuration** | All configurations (max: X16Gen3 or X8Gen4) | | | |
| **Xilinx Reference Board Support** | KCU116 Evaluation Board for Kintex UltraScale+ FPGA<br>VCU118 Evaluation Board for Virtex UltraScale+ FPGA | | | |
| **Device Support** | **Part**[1] | **PCIe Block Location** | **Tandem Configuration** | **Tandem PCIe with Field Updates** |
| Kintex UltraScale+ | KU3P | PCIE40E4_X0Y0 | Production | Production |
| | KU5P | PCIE40E4_X0Y0 | Production | Production |
| | KU11P | PCIE40E4_X1Y0 | Production | Production |
| | KU15P | PCIE40E4_X1Y0 | Production | Production |
| Virtex UltraScale+ | VU3P | PCIE40E4_X1Y0 | Production | Production |
| | VU5P | PCIE40E4_X1Y0 | Production | Production |
| | VU7P | PCIE40E4_X1Y0 | Production | Production |
| | VU9P | PCIE40E4_X1Y2 | Production | Production |
| | VU11P | PCIE40E4_X0Y0 | Production | Production |
| | VU13P | PCIE40E4_X0Y1 | Production | Production |
| | VU23P | PCIE40E4_X0Y0 | Not yet supported | Not yet supported |
| | VU27P | PCIE40E4_X0Y0 | Production | Production |
| | VU29P | PCIE40E4_X0Y0 | Production | Production |
| | VU31P | PCIE4CE4_X1Y0 | Production | Production |
| | VU33P | PCIE4CE4_X1Y0 | Production | Production |
| | VU35P | PCIE4CE4_X1Y0 | Production | Production |
| | VU37P | PCIE4CE4_X1Y0 | Production | Production |
| | VU45P | PCIE4CE4_X1Y0 | Production | Production |
| | VU47P | PCIE4CE4_X1Y0 | Production | Production |
| Zynq® UltraScale+™ MPSoC | ZU4CG/EG/EV | PCIE40E4_X0Y1 | Production | Production |
| | ZU5CG/EG/EV | PCIE40E4_X0Y1 | Production | Production |
| | ZU7CG/EG/EV | PCIE40E4_X0Y1 | Production | Production |
| | ZU11EG | PCIE40E4_X1Y0 | Production | Production |
| | ZU17EG | PCIE40E4_X1Y0 | Production | Production |
| | ZU19EG | PCIE40E4_X1Y0 | Production | Production |

| | | | | |
|---|---|---|---|---|
| | ZU21DR | PCIE40E4_X0Y0 | Not yet supported | Not supported |
| | ZU25DR | PCIE40E4_X0Y0 | Not yet supported | Not supported |
| | ZU27DR | PCIE40E4_X0Y0 | Not yet supported | Not supported |
| | ZU28DR | PCIE40E4_X0Y0 | Not yet supported | Not supported |
| | ZU29DR | PCIE40E4_X0Y0 | Not yet supported | Not supported |
| | ZU39DR | PCIE4CE4_X0Y0 | Beta (Tandem PROM only) | Not supported |
| Zynq® UltraScale+™ RFSoC[2] | ZU43DR | PCIE4CE4_X0Y0 | Not yet supported | Not supported |
| | ZU45DR | PCIE4CE4_X0Y0 | Beta (Tandem PROM only) | Not supported |
| | ZU47DR | PCIE4CE4_X0Y0 | Beta (Tandem PROM only) | Not supported |
| | ZU48DR | PCIE4CE4_X0Y0 | Beta (Tandem PROM only) | Not supported |
| | ZU49DR | PCIE4CE4_X0Y0 | Beta (Tandem PROM only) | Not supported |

**Notes:**

1. Only production silicon is officially supported. Bitstream generation is disabled for all engineering sample silicon (ES1, ES2) devices.
2. Zynq RFSoC devices do not have MCAP-enabled PCIe block locations. Because of this, support will be limited to the Tandem PROM solution only.

# Design Flow Steps

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

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

## Customizing and Generating the Subsystem

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

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

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

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

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

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

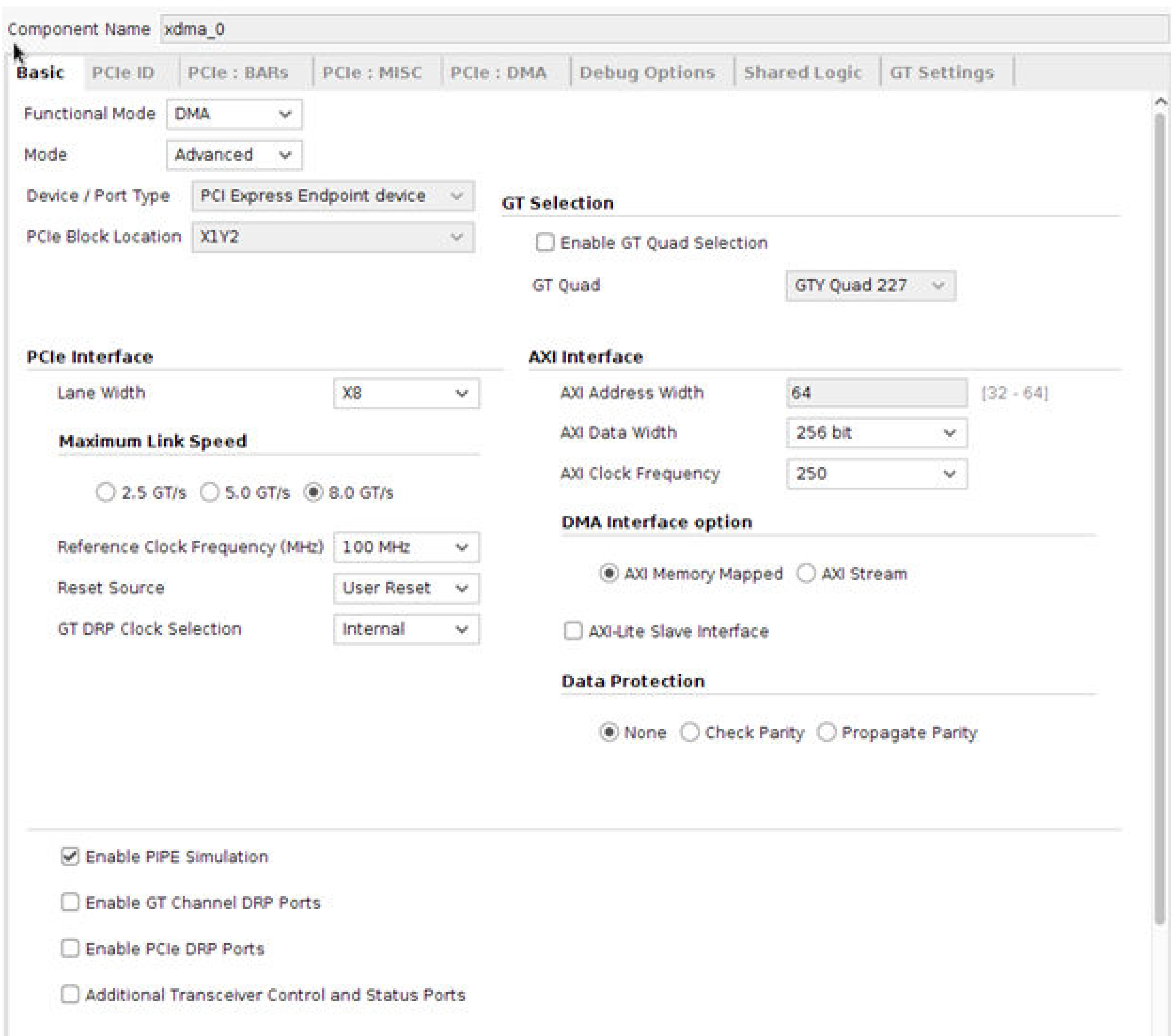
# DMA/Bridge Subsystem for PCI Express®

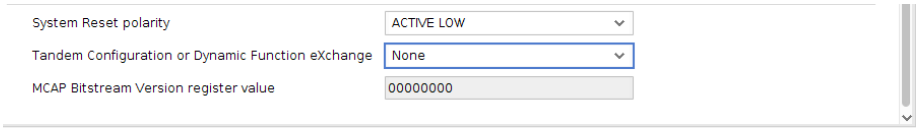
This section shows the configuration options that are available when the Functional Mode is set to **DMA**.

## Basic Tab

The Basic tab for the DMA mode (Functional Mode option) is shown in the following figure.

*Figure 12:* **Basic Tab for DMA Functional Mode**






The options are defined as follows:

- **Functional Mode:** Allows you to select between the following:
  - DMA (DMA Subsystem for PCIe).
  - AXI Bridge (AXI Bridge Subsystem for PCIe). The AXI Bridge option is valid only for UltraScale+™ devices. For details about PCIe Bridge mode operation, see *AXI Bridge for PCI Express Gen3 Subsystem Product Guide* (PG194). This document covers DMA mode operation only.
- **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. For supported lane widths and link speeds, see the *7 Series FPGAs Integrated Block for PCI Express LogiCORE IP Product Guide* (PG054), *Virtex-7 FPGA Integrated Block for PCI Express LogiCORE IP Product Guide* (PG023), *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156), or the *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213) Higher link speed cores are capable of training to a lower link speed if connected to a lower link speed capable device.
- **Reference Clock Frequency:** The default is 100 MHz, but 125 MHz and 250 MHz are also supported.
- **Reset Source:** You can choose between User Reset and Phy ready.
  - User reset comes from the PCIe core once link is established. When PCIe link goes down, User Reset is asserted and XDMA goes to reset mode. And when the link comes back up, User Reset is deasserted.
  - When the Phy ready option is selected, XDMA is not affected by PCIe link status.
- **GT DRP Clock Selection:** Select either internal clock (default) or external clock.
- **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, 256-bit, or 512-bit (only for UltraScale+). The core allows you to select the Interface Width, as defined in the *7 Series FPGAs Integrated Block for PCI Express LogiCORE IP Product Guide* (PG054), *Virtex-7 FPGA Integrated Block for PCI Express LogiCORE IP Product Guide* (PG023), *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156), or the *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213)
- **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.
- **AXI4-Lite Slave Interface:** Select to enable the AXI4-Lite slave Interface.
- **Data Protection:** By default parity checking is disabled.
  - When **Check Parity** is enabled, XDMA checks for parity on read data from the PCIe and generates parity for write data to the PCIe.
  - When **Propagate Parity** is enabled, XDMA propagates parity to the user AXI interface. The user is responsible for checking and generating parity on the user AXI interface.
- **Tandem Configuration or Dynamic Function eXchange:** Select the tandem configuration or Dynamic Function eXchange feature, if application to your design.

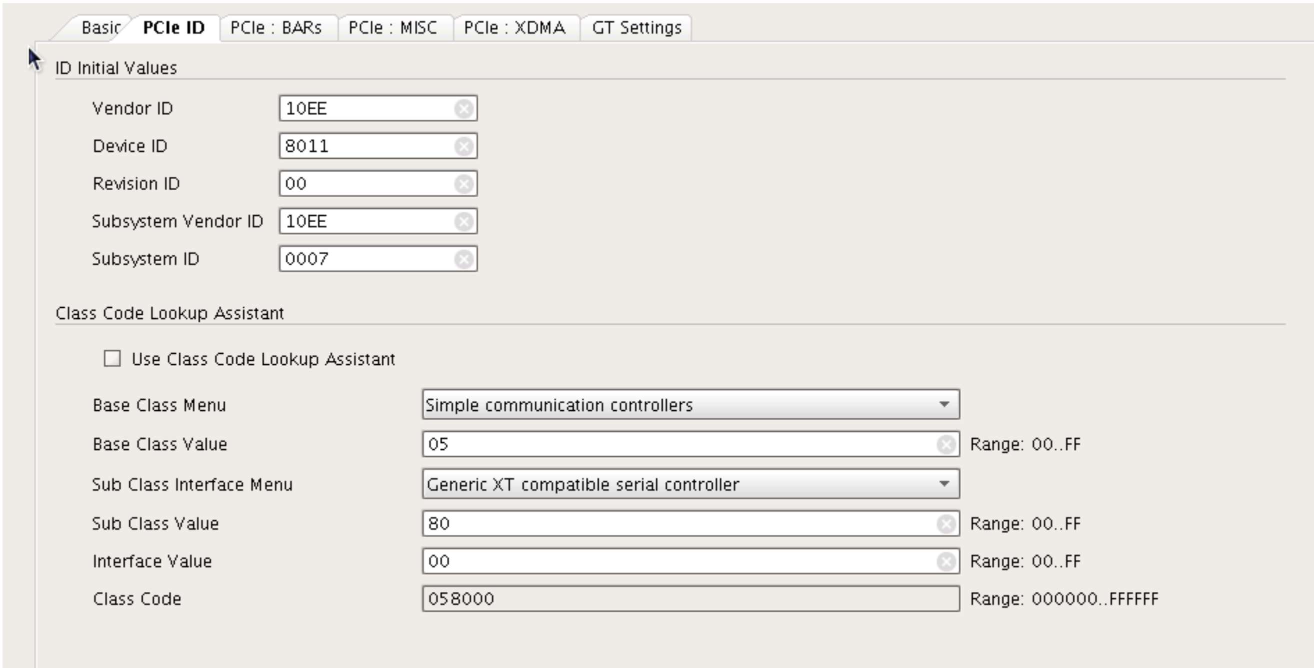
#### Related Information

Tandem Configuration

## PCIe ID Tab

The PCIe ID tab is shown in the following figure.

*Figure 13:* **PCIe ID Tab**



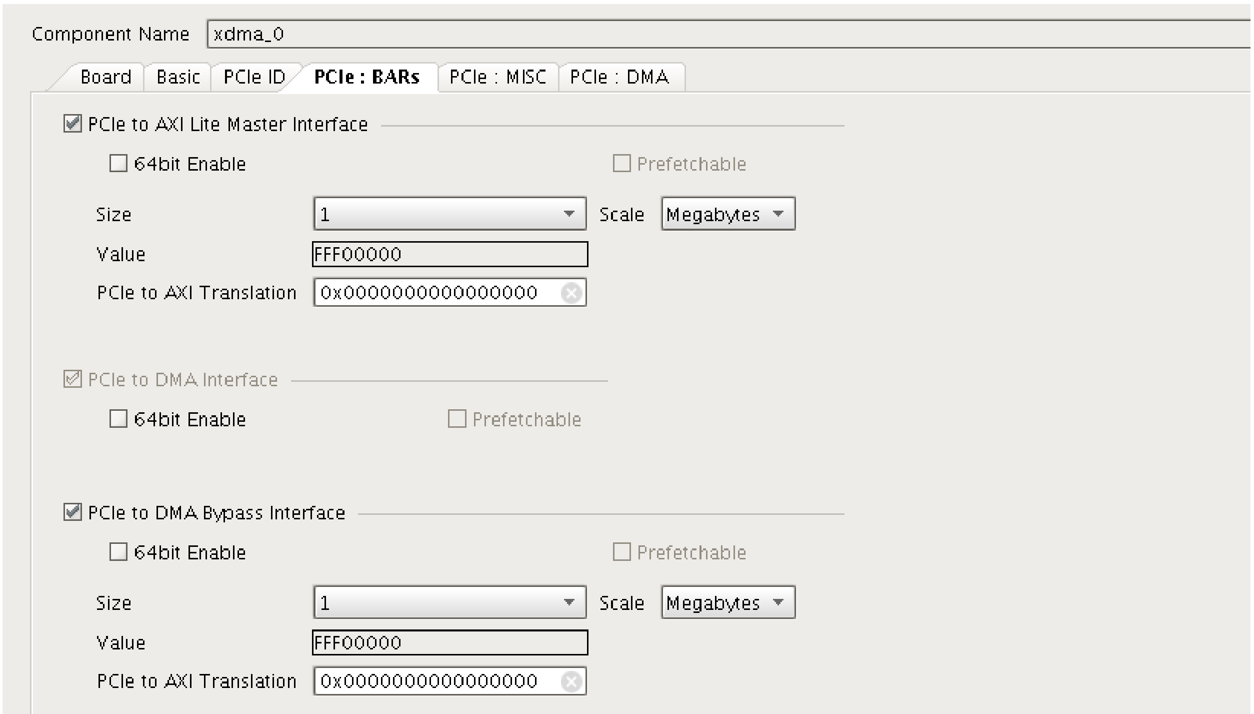

For a description of these options, see the “Design Flow Steps” chapter in the respective product guide listed below:

- *7 Series FPGAs Integrated Block for PCI Express LogiCORE IP Product Guide* (PG054)
- *Virtex-7 FPGA Integrated Block for PCI Express LogiCORE IP Product Guide* (PG023)
- *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156)
- *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213)

## PCIe BARs Tab

The PCIe BARs tab is shown in the following figure.

*Figure 14:* **PCIe BARs Tab**



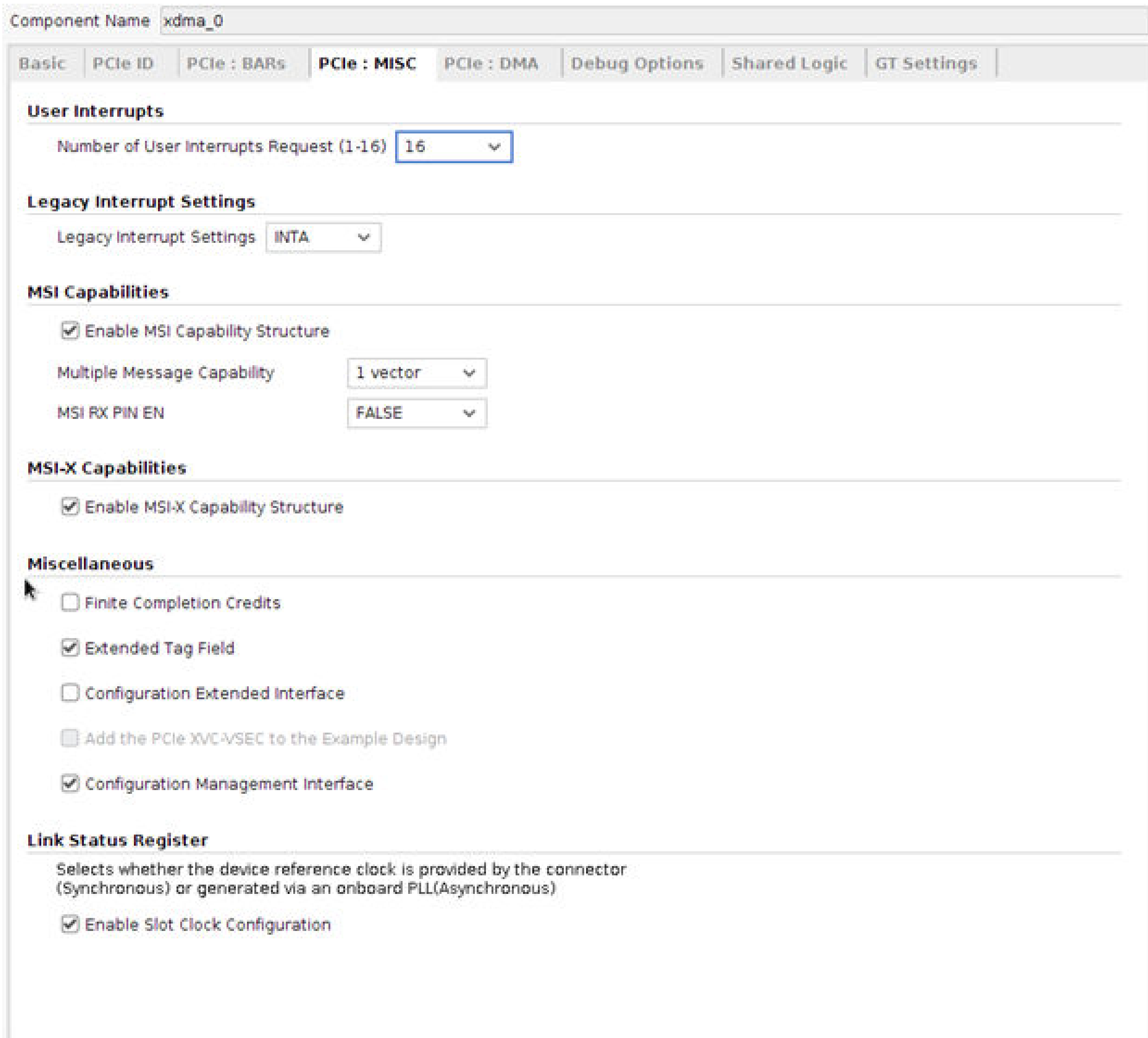

- **PCIe to AXI Lite Master Interface:** You can optionally enable to AXI-Lite Interface BAR space. The size, scale, and address translation are configurable.
- **PCIe to XDMA Interface:** This options is always selected.
- **PCIe to DMA Bypass Interface:** You can optionally enable PCIe to DMA Bypass Interface BAR space. The size, scale and address translations are also configurable.

Each BAR space can be individually selected for 64 bit options. And each 64 bit BAR space can be selected for Prefetchable or not.

## PCIe Misc Tab

The PCIe Miscellaneous tab is shown in the following figure.

*Figure 15:* **PCIe Misc Tab**



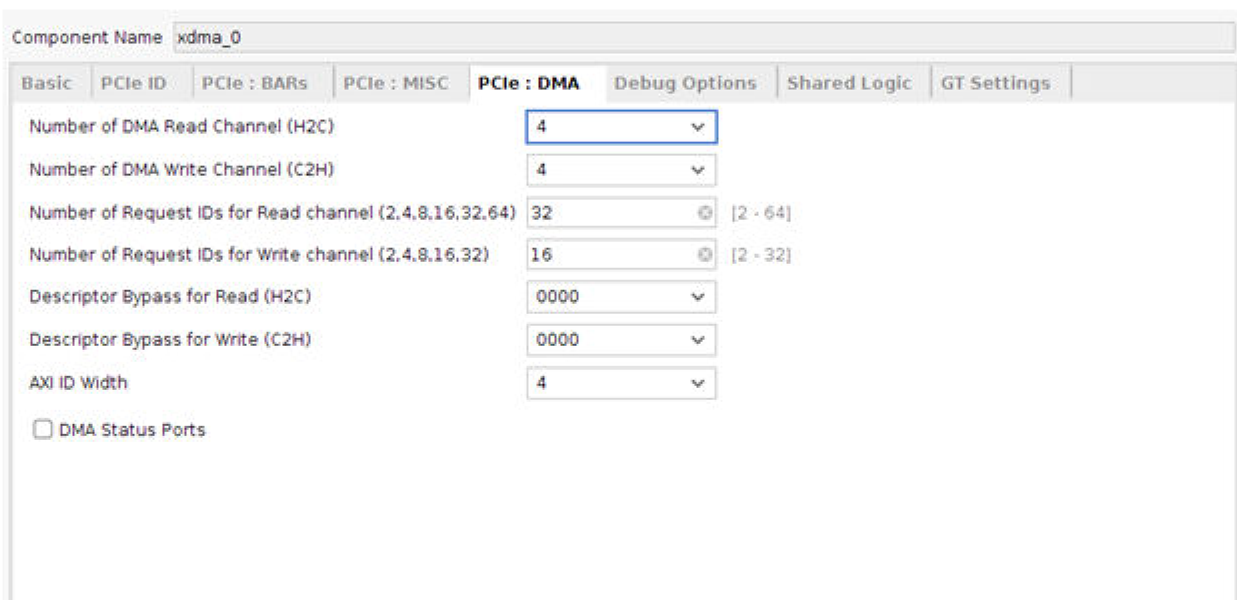

- **Legacy Interrupt Settings:** Select one of the Legacy Interrupts: INTA, INTB, INTC, or INTD.
- **Number of User Interrupt Request:** Up to 16 user interrupt requests can be selected.
- **MSI Capabilities:** By default, MSI Capabilities is enabled, and 1 vector is enabled. You can choose up to 32 vectors. In general, Linux uses only 1 vector for MSI. This option can be disabled.
- **MSI RX PIN EN:** This option is valid only in AXI Bridge Root Port mode.
- **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 50 µs 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. For UltraScale™ and Virtex®-7 devices, the Extended Tag option gives 64 tags. For UltraScale+™ devices, the Extended Tag option gives 256 tags. If the Extended Tag option is not selected, DMA uses 32 tag for all devices.
- **Configuration Extend Interface:** PCIe extended interface can be selected for more configuration space. When Configuration Extend Interface is selected, you are responsible for adding logic to extend the interface to make it work properly.
- **Configuration Management Interface:** PCIe configuration Management interface can be brought to the top level when this options is selected.

## PCIe DMA Tab

The PCIe DMA tab is shown in the following figure.

*Figure 16:* **PCIe DMA Tab**



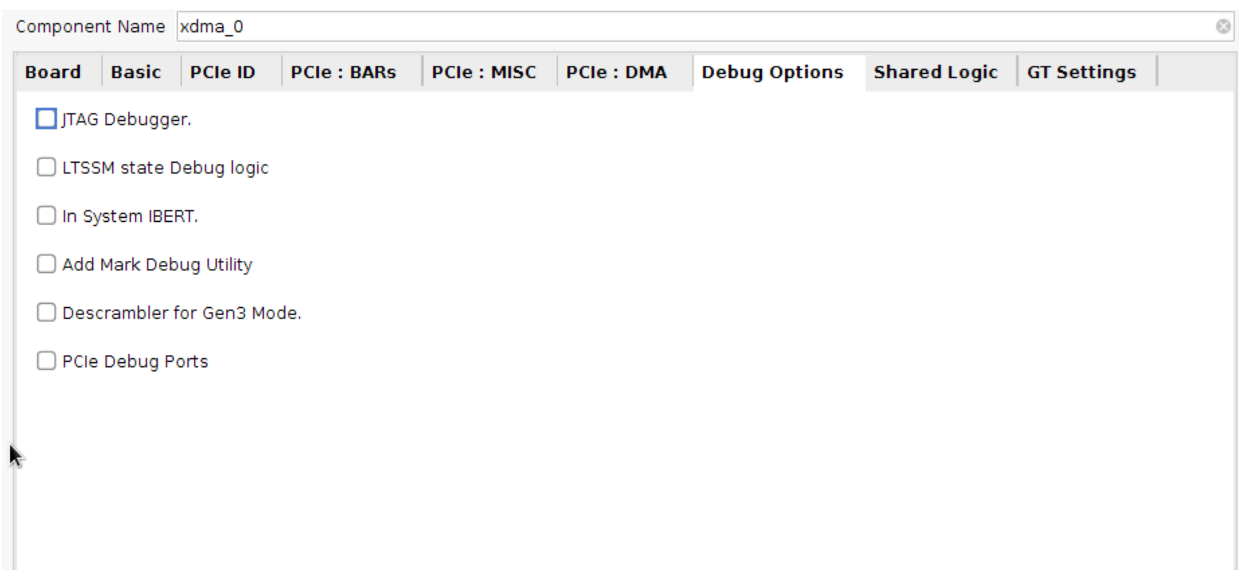

- **Number of Read/ Write Channels:** Available selection is from 1 to 4.
- **Number of Request IDs for Read channel:** Select the max number of outstanding request per channel. Available selection is from 2 to 64.
- **Number of Request IDs for Write channel:** Select max number of outstanding request 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.
- **AXI ID Width:** The default is 4-bit wide. You can also select 2 bits.
- **DMA Status port:** DMA status ports are available for all channels.

## Debug Options Tab

The Debug Options tab is shown in the following figure.

*Figure 17:* **Debug Options Tab**




- **JTAG Debugger:** This option enables JTAG debugging.
- **LTSSM State Debug Logic:** This option shows all the LTSSM state transitions that have been made starting from link up.
- **In System IBERT:** This option is used to check and see the eye diagram of the serial link at the desired link speed. For more information on In System IBERT, refer to *In-System IBERT LogiCORE IP Product Guide* (PG246).

> **IMPORTANT!** *This option is used mainly for hardware debug purposes. Simulations are not supported when this option is used.*

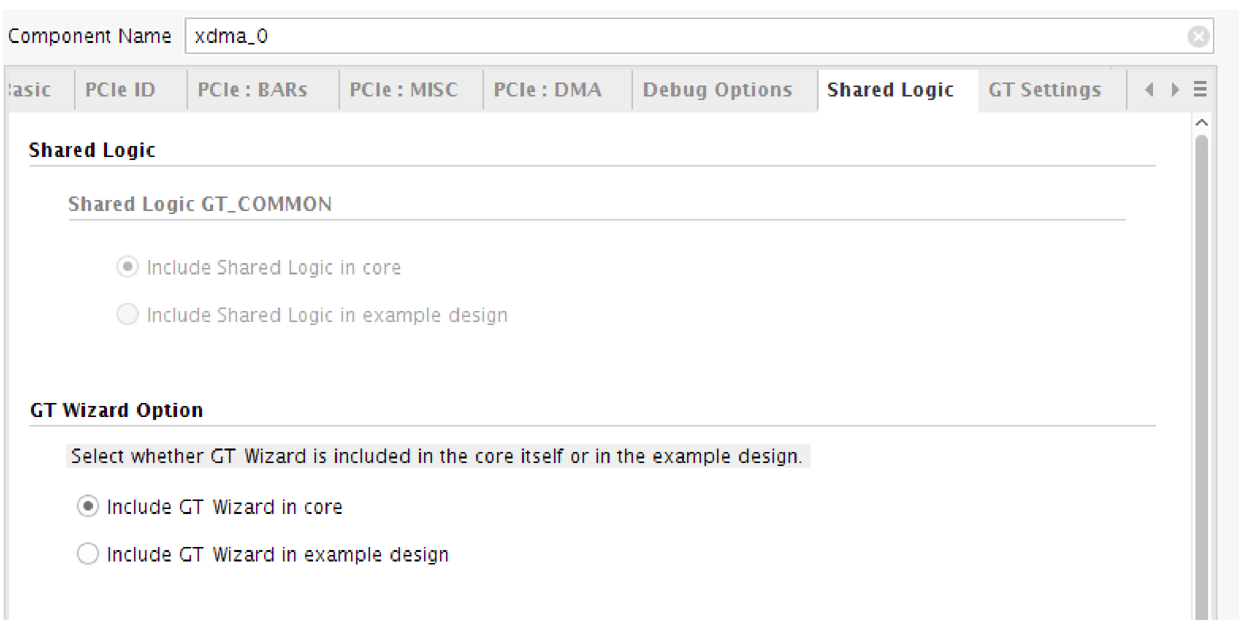
- **Add Mark Debug Utility:** This option adds predefined PCIe signals to with mark_debug attribute so these signals can be added in ILA for debug purpose.

- **Descrambler for Gen3 Mode:** This option integrates encrypted version of the descrambler module inside the PCIe core, which will be used to descrambler the PIPE data to/from PCIe integrated block in Gen3 link speed mode.
- **PCIe Debug Ports:** With this option enabled, the following ports are available:
  - `cfg_negotiated_width: cfg_negotiated_width_o`
  - `cfg_current_speed: cfg_current_speed_o`
  - `cfg_ltssm_state: cfg_ltssm_state_o`
  - `cfg_err_cor: cfg_err_cor_o`
  - `cfg_err_fatal: cfg_err_fatal_o`
  - `cfg_err_nonfatal: cfg_err_nonfatal_o`
  - `cfg_local_error: cfg_local_error_o`
  - `cfg_local_error_valid: cfg_local_error_valid_o`

## Shared Logic Tab

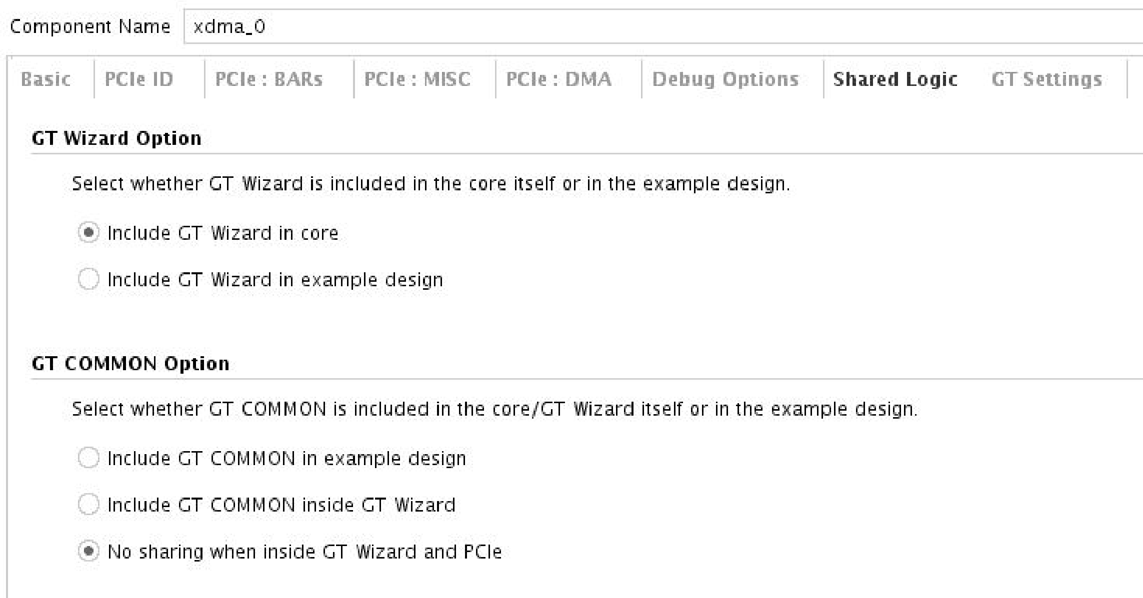
The Shared Logic tab for IP in an UltraScale™ device is shown in the following figure.

*Figure 18:* **Shared Logic (UltraScale Devices)**




The Shared Logic tab for IP in an UltraScale+™ device is shown in the following figure.

*Figure 19:* **Shared Logic (UltraScale+ Devices)**



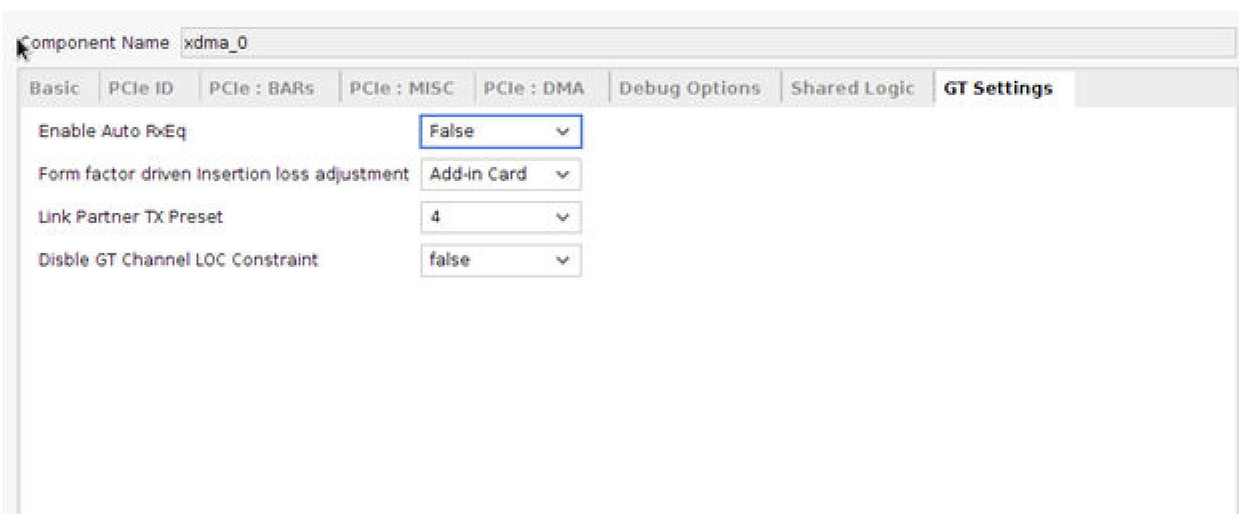

For a description of these options, see Chapter 4, "Design Flow Steps" in the respective product guide listed below:

- *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156)
- *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213)

## GT Settings Tab

The GT Settings tab is shown in the following figure.

*Figure 20:* **GT Settings Tab**



For a description of these options, see Chapter 4, "Design Flow Steps" in the respective product guide listed below:

- *7 Series FPGAs Integrated Block for PCI Express LogiCORE IP Product Guide* (PG054)
- *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156)
- *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213)

# Output Generation

For details, see *Vivado Design Suite User Guide: Designing with IP* (UG896).

# Constraining the Subsystem

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

### Required Constraints

The DMA/Bridge Subsystem for PCI Express® 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 (for UltraScale+ IBUFDS_GTE4) 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).

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 Gen2 and 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, Clock Management, and Clock Placement

For detailed information about clock requirements, see the respective product guide listed below:

- *7 Series FPGAs Integrated Block for PCI Express LogiCORE IP Product Guide* (PG054)
- *Virtex-7 FPGA Integrated Block for PCI Express LogiCORE IP Product Guide* (PG023)
- *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156)
- *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213)

### Banking

This section is not applicable for this IP subsystem.

### Transceiver Placement

This section is not applicable for this IP subsystem.

### I/O Standard and Placement

This section is not applicable for this IP subsystem.

## 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).

## Basic Simulation

Simulation models for AXI-MM and AXI-ST options can be generates and simulated. These are very basic simulation model options on which you can develop complicated designs.

### *AXI-MM Mode*

The example design for the AXI4 Memory Mapped (AXI-MM) mode has 4 KB block RAM on the user side, so data can be written to the block RAM and read from block RAM to the Host. The first H2C transfer is started and the DMA reads data from the Host memory and writes to the block RAM. Then, the C2H transfer is started and the DMA reads data from the block RAM and writes to the Host memory. The original data is compared with the C2H write data.

H2C and C2H are setup with one descriptor each, and the total transfer size is 64 bytes.

### *AXI-ST Mode*

The example design for the AXI4-Stream (AXI_ST) mode is a loopback design. On the user side the H2C ports are looped back to the C2H ports. First, the C2H transfer is started and the C2H DMA engine waits for data on the user side. Then, the H2C transfer is started and the DMA engine reads data from the Host memory and writes to the user side. Because it is a loopback, design data from H2C is directed to C2H and ends up in the host destination address.

H2C and C2H are setup with one descriptor each, and the total transfer size is 64 bytes.

Interrupts are not used in Vivado® Design Suite simulations. Instead, descriptor completed count register is polled to determine transfer complete.

### Descriptor Bypass

Simulation models for the descriptor bypass mode is available only for channel 0. This design can be expanded to support other channels.

## PIPE Mode Simulation

The DMA/Bridge Subsystem for PCI Express® 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. See also *PIPE Mode Simulation Using Integrated Endpoint PCI Express Block in Gen3 x8 and Gen2 x8 Configurations Application Note* (XAPP1184).

The following tables describe the PIPE bus signals available at the top level of the core and their corresponding mapping inside the EP core (`pcie_top`) PIPE signals.

> **IMPORTANT!** *The* `xil_sig2pipe.v` *file 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`*.*

PIPE mode simulations are not supported for this core when VHDL is the selected target language.

*Table 129:* **Common In/Out Commands and Endpoint PIPE Signals Mappings**

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

*Table 129:* **Common In/Out Commands and Endpoint PIPE Signals Mappings** *(cont'd)*

| In Commands | Endpoint PIPE Signals Mapping | Out Commands | Endpoint PIPE Signals Mapping |
|---|---|---|---|
| | | common_commands_out[16:10] | not used[3] |

**Notes:**

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

*Table 130:* **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.

## *Parameters for Custom PIPE Simulation*

For PIPE simulation, certain parameters are required, and might need to be manually set. These required parameters are provided in the example design. When you generate an example design from the Vivado IP catalog, all required parameters are set, and no additional action is required. However, custom designs will require that you add the following parameters to your design test bench file.

```
defparam board.AXI_PCIE_EP.xdma_0_i.inst.pcie4_ip_i.inst.PL_SIM_FAST_LINK_TRAINING=2'h3;
localparam EXT_PIPE_SIM = "TRUE";
defparam board.AXI_PCIE_EP.xdma_0_i.inst.pcie4_ip_i.inst.EXT_PIPE_SIM = EXT_PIPE_SIM;
defparam board.RP.pcie_4_0_rport.pcie_4_0_int_inst.EXT_PIPE_SIM = "TRUE";
defparam board.RP.EXT_PIPE_SIM = "TRUE";
```

# Synthesis and Implementation

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

Chapter 6

# Example Design

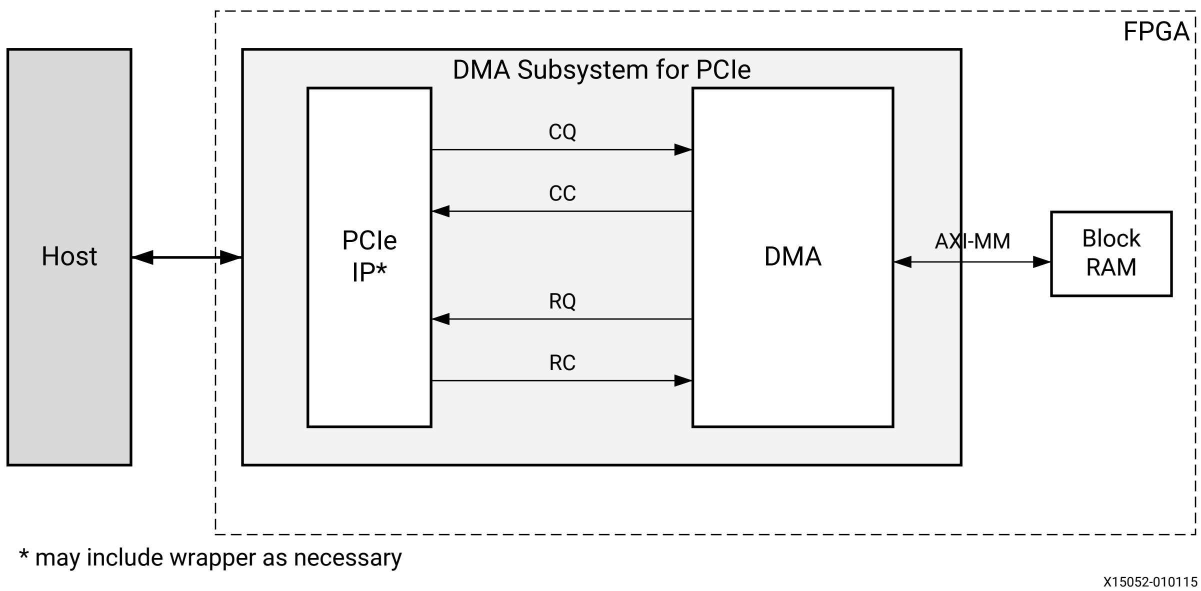
This chapter contains information about the example designs provided in the Vivado® Design Suite. The example designs are as follows:

- AXI4 Memory Mapped Default Example Design
- AXI4 Memory Mapped with PCIe to AXI4-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 with Descriptor Bypass Example
- Vivado IP Integrator-Based Example Design
- User IRQ Example Design

# AXI4 Memory Mapped Default Example Design

The following figure 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/Bridge Subsystem for PCI Express® reads data from host and writes to block RAM in the user side. For C2H transfers, the DMA/Bridge Subsystem for PCI Express® reads data from block RAM and writes to host memory. The example design from the IP catalog has only 4 KB block RAM; you can regenerate the core for larger block RAM size, if wanted.

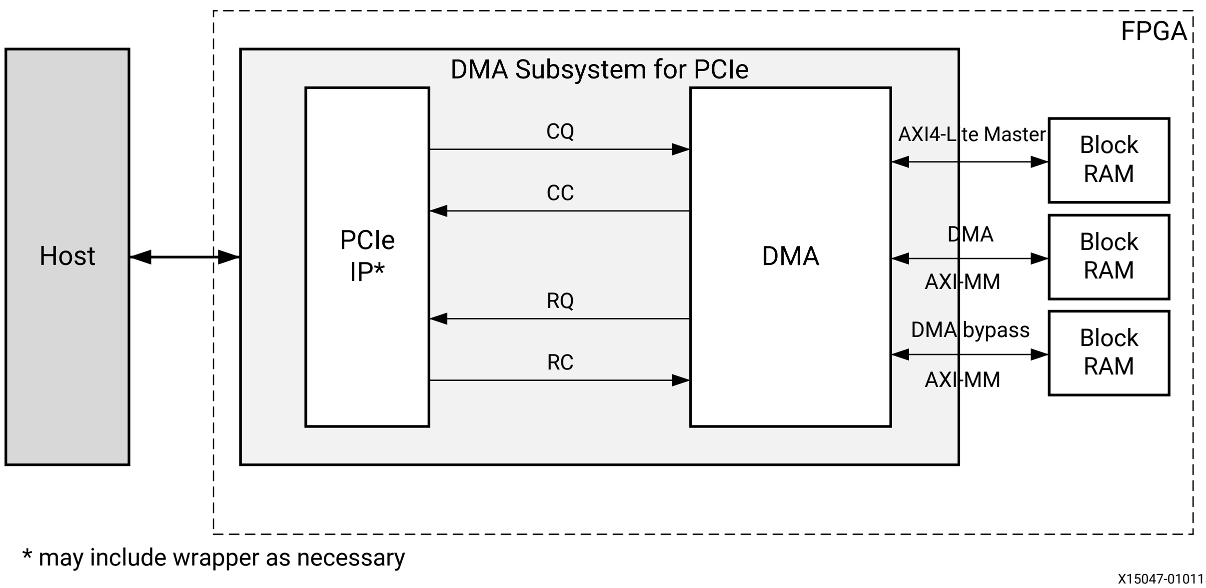
*Figure 21:* **AXI-MM Default Example Design**



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

The following figure shows a system where the PCIe to AXI4-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 AXI4-Lite Master (BAR0 address space) to write/read user logic. The example design connects 4K block RAM to the PCIe to AXI4-Lite Master interface so the user application can perform read/writes.

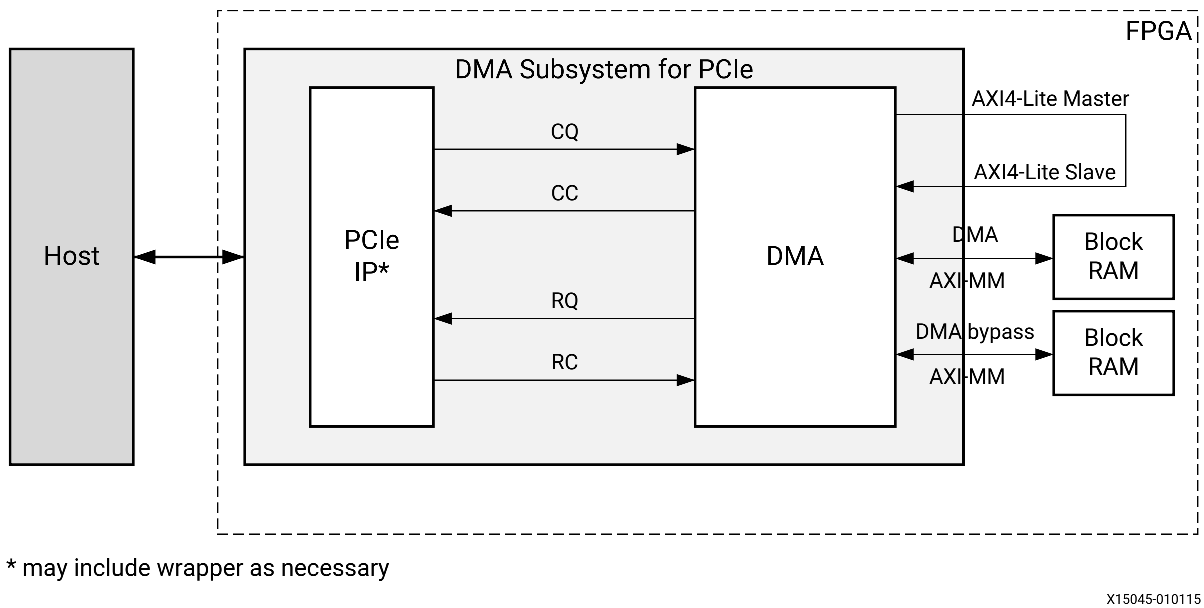
*Figure 22:* **AXI-MM Example with PCIe to DMA Bypass Interface and PCIe to AXI-Lite Master Enabled**



# AXI4 Memory Mapped with AXI4-Lite Slave Interface Example Design

When the PCIe® to AXI4-Lite master and AXI4-Lite slave interface are enabled, the generated example design (shown in the following figure) 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/Bridge Subsystem for PCI Express® registers. With this example design, the host can use PCIe to AXI4-Lite Master (BAR0 address space) and read/write DMA/Bridge Subsystem for PCI Express® 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.

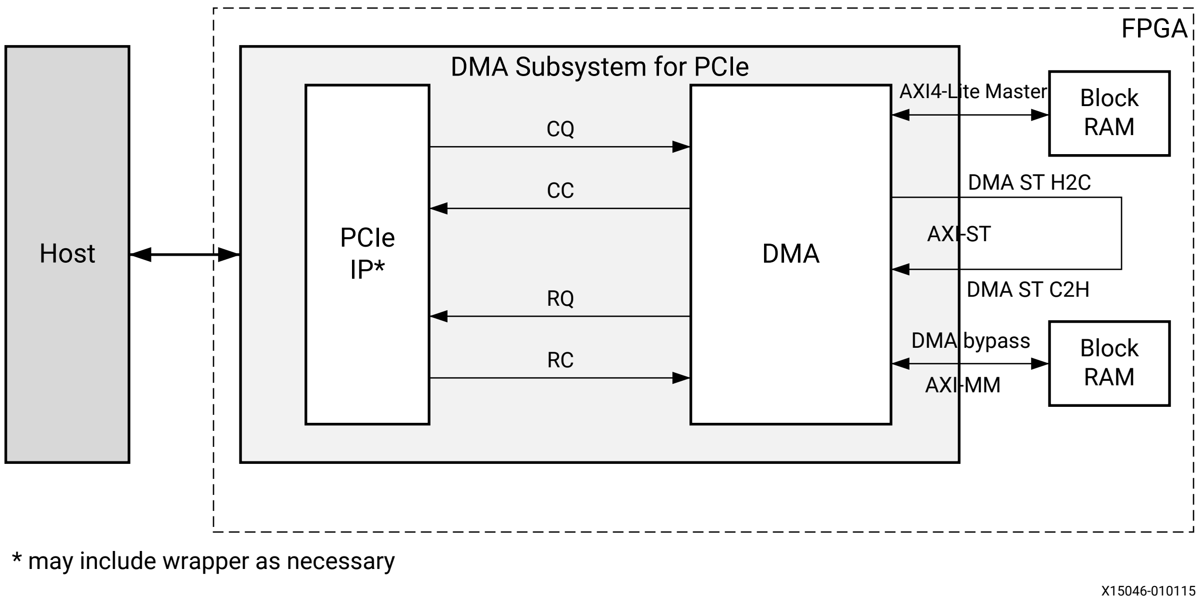
*Figure 23:* **AXI-MM Example with AXI-Lite Slave Enabled**



# AXI4-Stream Example Design

When the AXI4-Stream interface is enabled, each H2C streaming channels is looped back to C2H channel. As shown in the following figure, 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 24:* **AXI4-Stream Example with PCIe to DMA Bypass Interface and PCIe to AXI-Lite Master Enabled**

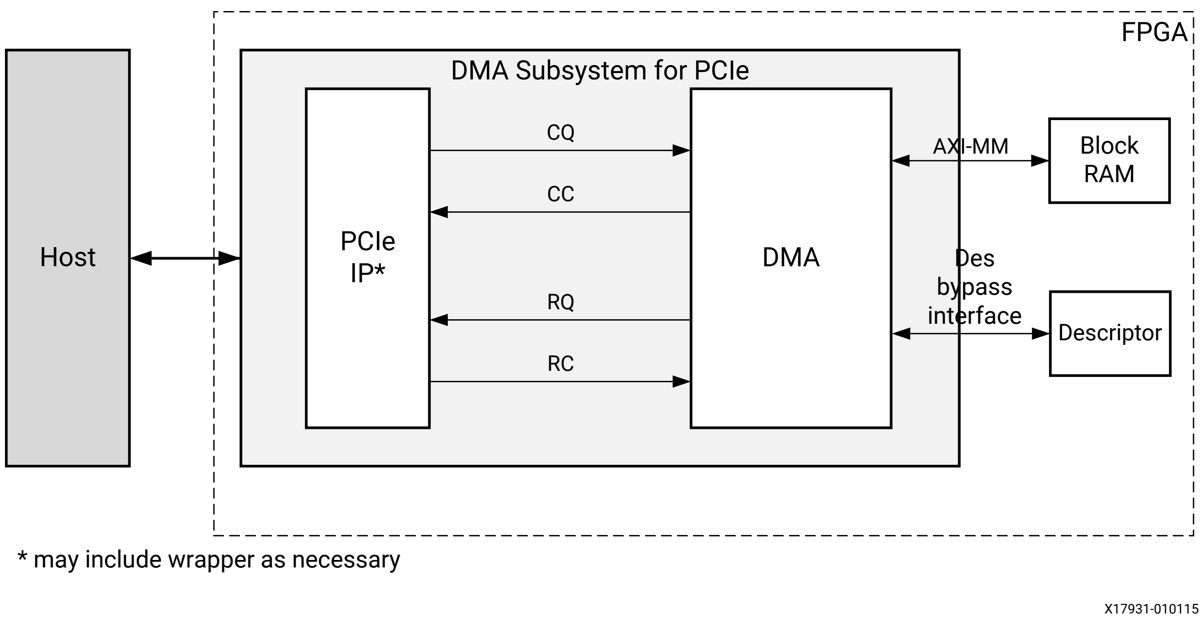


# AXI4-Memory Mapped with Descriptor Bypass Example

When Descriptor bypass mode is enabled, the user logic is responsible for making descriptors and transferring them in descriptor bypass interface. The following figure shows AXI4-Memory Mapped design with descriptor bypass mode enabled. You can select which channels will have descriptor bypass mode. When Channel 0 of H2C and Channel 0 C2H are selected for Descriptor bypass mode, the generated Vivado® example design has descriptor bypass ports of H2C0 and C2H0 connected to logic that will generate only one descriptor of 64bytes. The user is responsible for developing codes for other channels and expanding the descriptor itself.

The following figure shows the AXI-MM example with Descriptor Bypass Mode enabled.

*Figure 25:* **AXI-MM Example With Descriptor Bypass Mode Enabled**

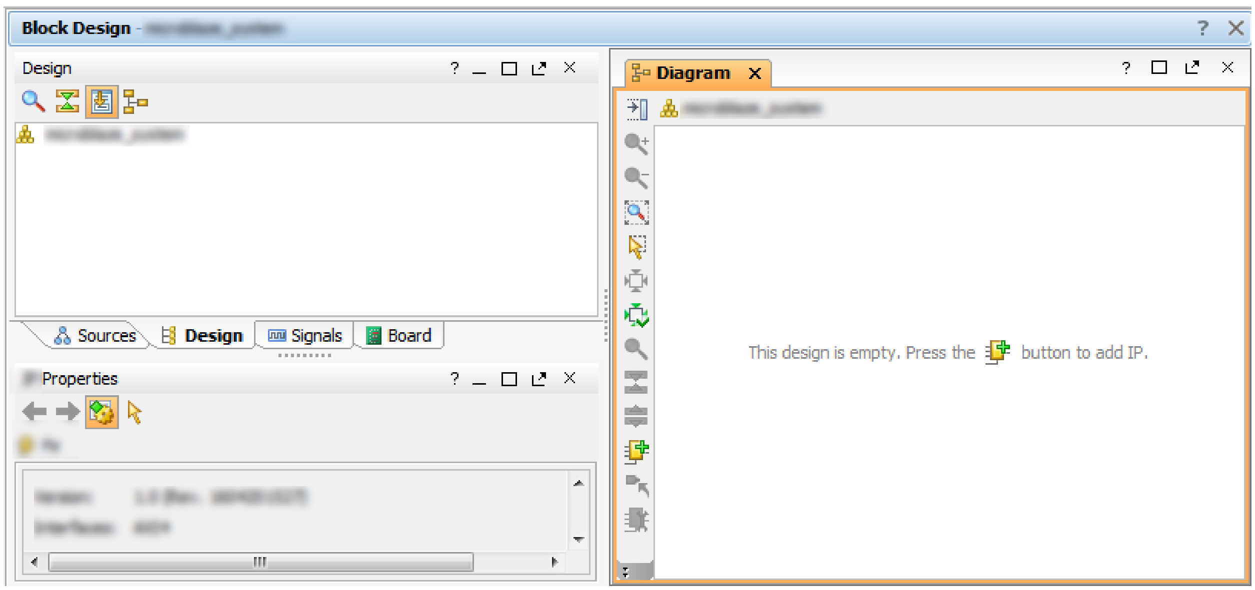


# Vivado IP Integrator-Based Example Design

In addition to the RTL-based example designs, the IP also supports a Vivado® IP integrator-based example design. To use the example design:

1. Create an IP integrator block diagram.
2. Open the IP integrator workspace, as shown in the following figure.

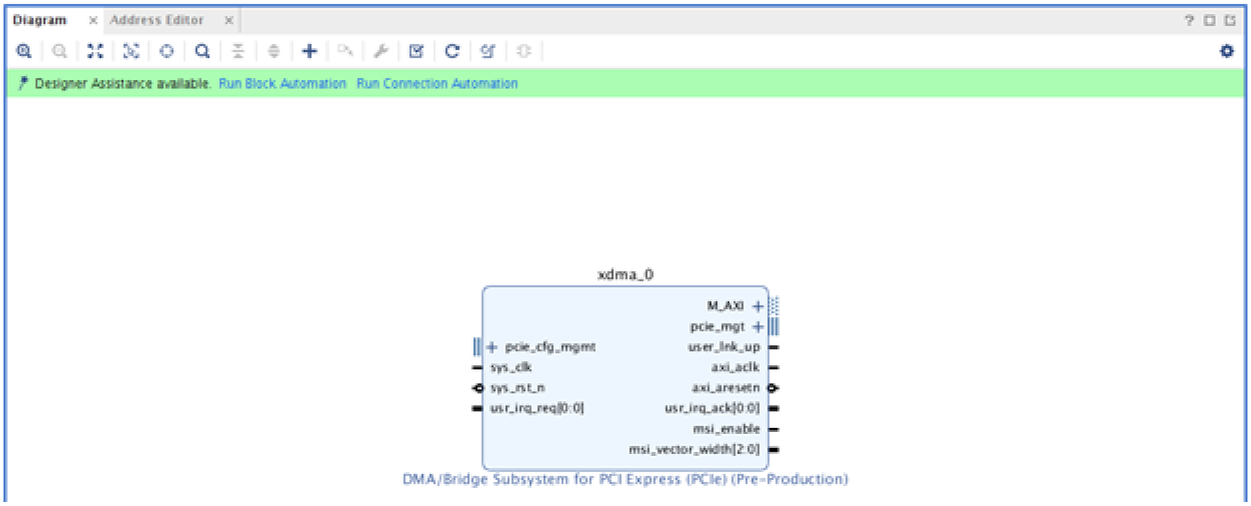
*Figure 26:* **Initial View of the Vivado IP Integrator Showing an Informational Message**




3. In order to add the DMA/Bridge IP to the canvas, search for DMA/Bridge (xdma) IP in the IP catalog.

   After adding the IP to the canvas, the green Designer Assistance information bar appears at the top of the canvas.

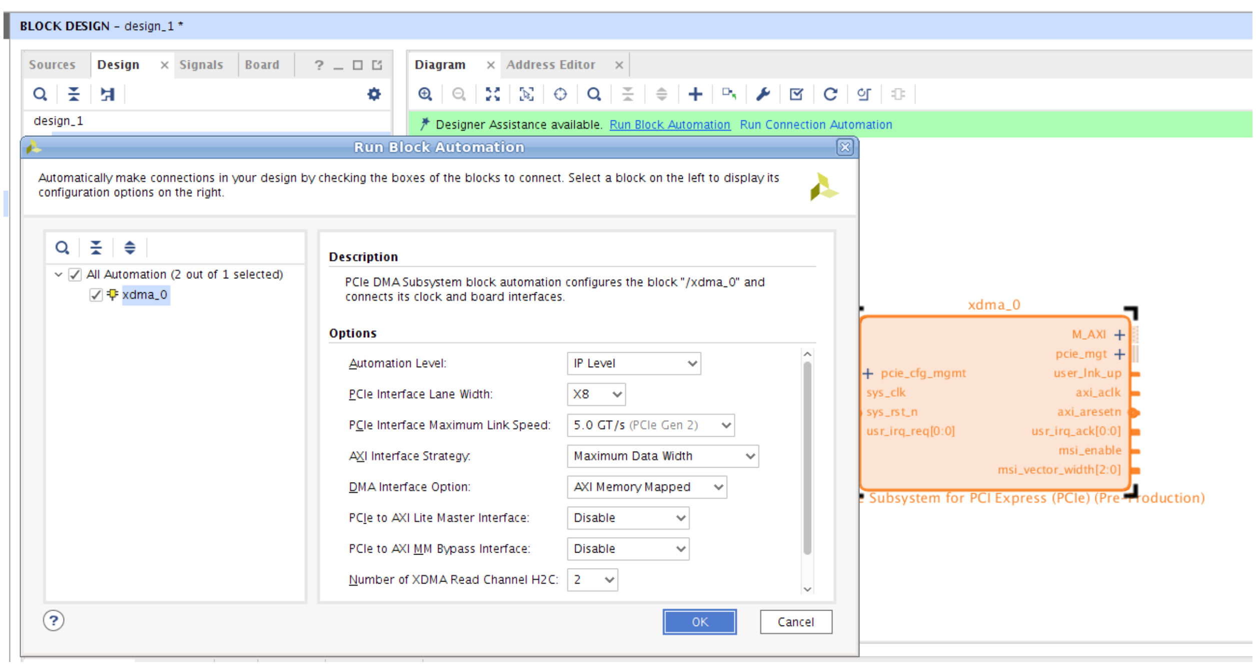
*Figure 27:* **Designer Assistance Offering Block Automation**




4. Click **Run Block Automation** from the Designer Assistance information bar.

 This opens a Run Block Automation dialog box (shown in the following figure) which lists all the IP currently in the design eligible for block automation (left pane), and any options associated with a particular automation (right pane). In this case, there is only the XDMA IP in the hierarchy in the left pane. The right pane has a description and options available. The Options can be used to configure the IP as well as decide the level of automation for block automation.

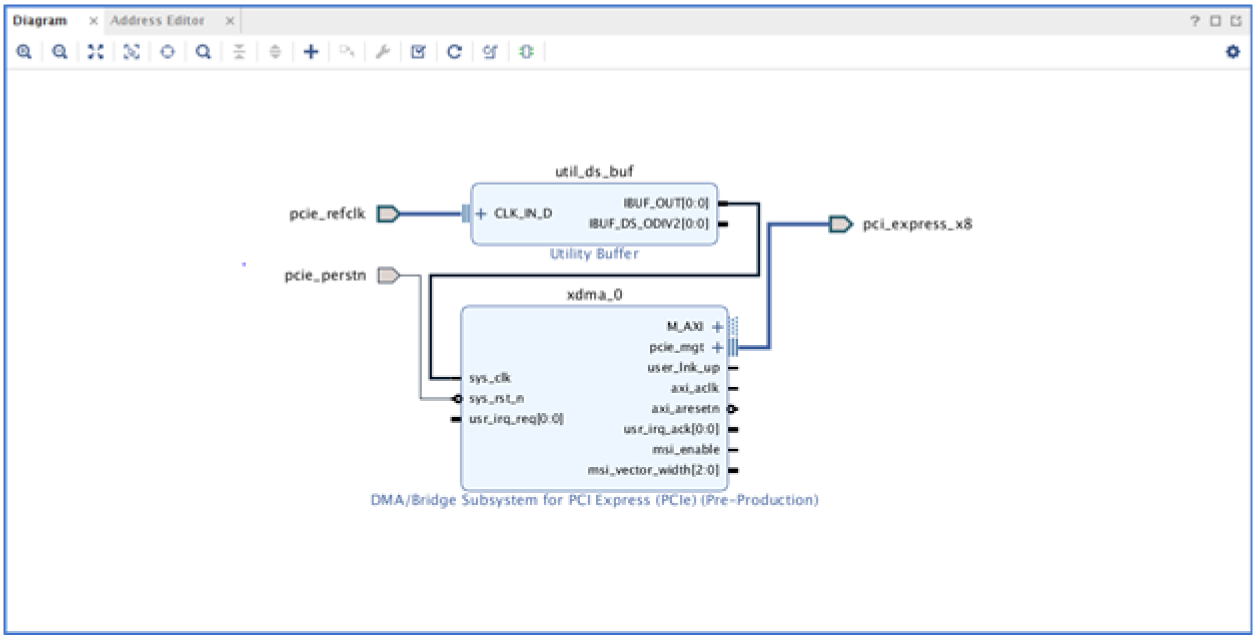
*Figure 28:* **Run Block Automation Dialog Box**



The Run Block Automation dialog box has an Automation Level option, which can be set to IP Level or Subsystem Level.

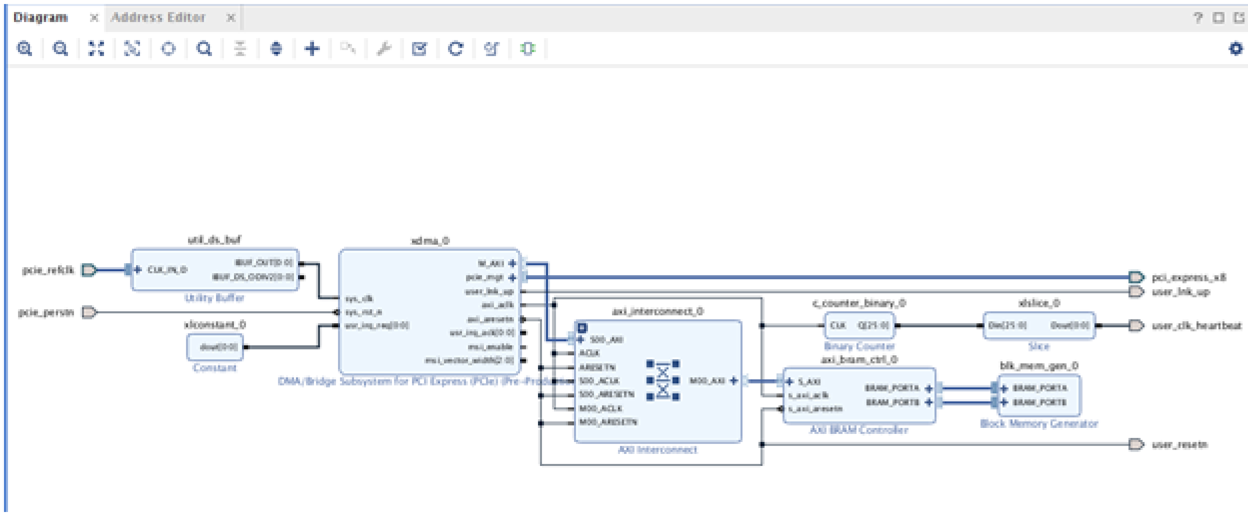
- **IP Level:** When you select IP level automation, the Block Automation inserts the utility buffer for the `sys_clk` input, connects the `sys_rst_n` input and `pcie_mgt` output interface for the XDMA IP, as shown in the following figure.

*Figure 29:* **IP Level Block Automation**




- **Subsystem Level:** When you select subsystem level automation, the Block Automation inserts the necessary sub IPs on the canvas and makes the necessary connections. In addition to connecting the `sys_clk` and `sys_rst_n` inputs it also connects the pcie_mgt output interface and `user_lnk_up`, `user_clk_heartbeat` and `user_resetn` outputs. It inserts the AXI interconnect to connect the Block Memory with the XDMA IP through the AXI Bram controller. The AXI interconnect has one master interface and multiple slave interfaces when the AXI4-Lite master and AXI-MM Bypass interfaces are enabled in the Run Block Automation dialog box. The block automation also inserts Block Memories and AXI Bram Controllers when the AXI4-Lite master and AXI-MM Bypass interfaces are enabled.

*Figure 30:* **Subsystem Level Block Automation**



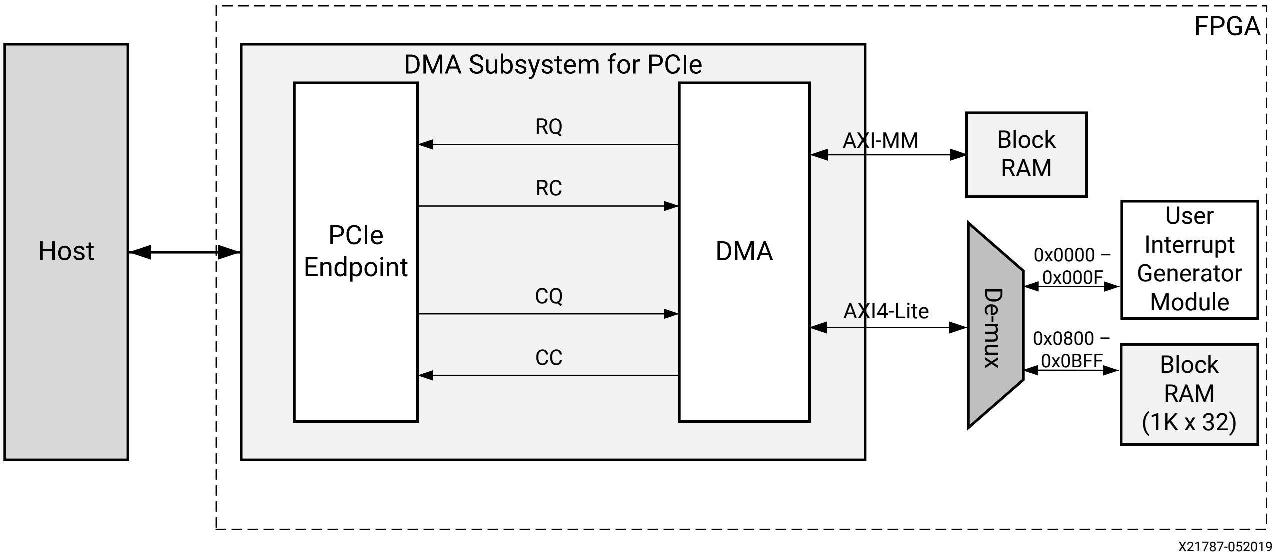
# User IRQ Example Design

The user IRQ example design enables the host to connect to the AXI4-Lite Master interface along with the default DMA/Bridge Subsystem for PCI Express® example design. In the example design, the User Interrupt generator module and an external block RAM is integrated on this AXI4-Lite interface. The host can use this interface to generate the user IRQ by writing to the register space of the User Interrupt generator module and can also read/write to the external 1K block RAM. The following figure shows the example design.

The example design can be generated using the following Tcl command.

```
set_property -dict [list CONFIG.usr_irq_exdes {true}] [get_ips <ip_name>]
```

*Figure 31:* **User IRQ Example Design**




The register description is found in the following table.

*Table 131:* **Example Design Registers**

| Register Offset | Register Name | Access Type | Description |
|---|---|---|---|
| 0x00 | Scratch Pad | RW | Scratch Pad |
| 0x04 | DMA BRAM Size | RO | User Memory Size connected to XDMA.<br>Memory size = (2[7:4]) ([3:0]Byte)<br>[7:4] – denotes the size in powers of 2.<br>0 – 1<br>1 – 2<br>2 – 4<br>…<br>8 – 256<br>9 – 512<br>[3:0] – denotes unit.<br>0 – Byte<br>1 – KByte<br>2 – MByte<br>3 – GByte<br>For example, if the register value is 21, the size is 4 KB. If the register value is 91, the size is 512 KB. |
| 0x08 | Interrupt Control Register | RW | Interrupt control register (write 1 to generate interrupt).<br>Interrupt Status register corresponding bit must be 1 (ready) to generate interrupt. Also, reset the corresponding bit after ISR is served. |

*Table 131:* **Example Design Registers** *(cont'd)*

| Register Offset | Register Name | Access Type | Description |
|---|---|---|---|
| 0x0C | Interrupt Status Register | RO | Interrupt Status.<br>1: ready<br>0: Interrupt generation in progress |

***Note***: In case of Legacy interrupt, the Interrupt Control Register (0x08) value for the corresponding interrupt bit should only be cleared after the ISR is served as this can be used by the host to determine the interrupt source.

Chapter 7

# 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 basic 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.

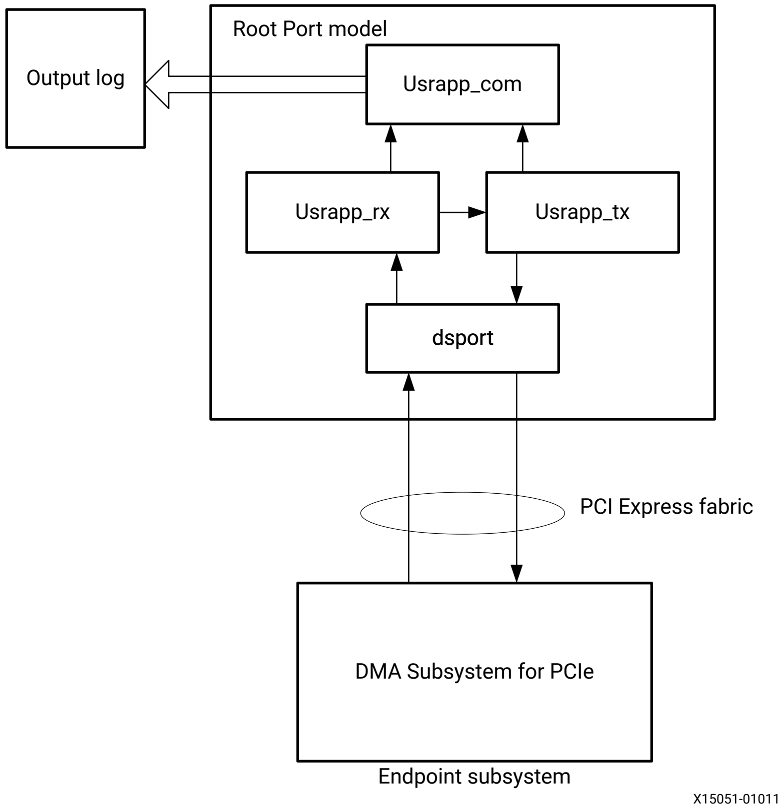
Source code for the Root Port Model is included to provide the model for a starting point for your test bench. All the significant work for initializing the core configuration space, creating TLP transactions, generating TLP logs, and providing an interface for creating and verifying tests are complete,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

The Root Port Model consists of:

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

The following figure shows the Root Port Module with DMA Subsystem for PCIe.

*Figure 32:* **Root Port Module with DMA Subsystem for PCIe**




## Architecture

The Root Port Model, illustrated in the previous figure, 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 7 series Gen2 Integrated Block for PCIe, the 7 series Gen3 Integrated Block for PCIe, the UltraScale™ Devices Gen3 Integrate Block for PCIe, and the UltraScale+™ Devices Integrate Block for PCIe. See the “Test Bench” chapter in the *7 Series FPGAs Integrated Block for PCI Express LogiCORE IP Product Guide* (PG054), *Virtex-7 FPGA Integrated Block for PCI Express LogiCORE IP Product Guide* (PG023), *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156), or *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213) 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 132:* **Test Case Descriptions**

| Test Case Name | Description |
|---|---|
| Dma_test0 | AXI4 Memory Mapped interface simulation. Reads data from host memory and writes to block RAM (H2C). Then, reads data from block RAM and write to host memory (C2H). The test case at the end compares data for correctness. |
| Dma_stream0 | AXI4-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

Simulation is set up to transfer one descriptor in H2C and one descriptor in C2H direction. Transfer size is set to 128 bytes in each descriptor. For both AXI-MM and AXI-Stream, data is read from Host and sent to Card (H2C). Then data is read from Card and sent to Host (C2H). Data read from Card is compared with original data for data validity.

Limitations:

- Simulation does not support Interrupts. Test case just reads status and complete descriptor count registers to decide if transfer is completed.
- Simulations are done only for Channel 0. In a future release, multi channels simulations will be enabled.
- Transfer size is limited to 128 bytes and only one descriptor.
- Root port simulation model is not a complete BFM. Simulation supports one descriptor transfer which shows a basic DMA procedure.

- By default, post-synthesis simulation is not supported for the example design. To enable post-synthesis simulation, generate the IP using the following Tcl command:

```
set_property -dict [list CONFIG.post_synth_sim_en {true}] [get_ips
<ip_name>]
```

**Note**: With this feature, functional simulation time increases to approximately 2.5 ms.

## *AXI4 Memory Mapped Interface*

First, the test case starts the H2C engine. The H2C engine reads data from host memory and writes to block RAM on user side. Then, the test case starts the C2H engine. The C2H engine reads data from block RAM 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 is created in the Host memory. The H2C descriptor gives data length 128 bytes, source address (host), and destination address (Card).
3. The test case writes data (incremental 128 bytes of 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 128 bytes, source address (Card), and destination address (host).
6. Write H2C descriptor starting address to register (`0x4080` and `0x4084`).
7. Write to H2C control register to start H2C transfer address (`0x0004`). Bit 0 is set to 1 to start the transfer. For details of control register, refer to H2C Channel Control (0x04).
8. The DMA transfer takes the data host source address and sends to the block RAM destination address.
9. The test case then starts the C2H transfer.
10. Write C2H descriptor starting address to register (`0x5080` and`0x5084`).
11. Write to C2H control register to start the C2H transfer (`0x1004`). Bit 0 is set to 1 to start the transfer. For details of control the register, see C2H Channel Control (0x04).
12. The DMA transfer takes data from the block RAM source address and sends data 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).
15. The test case then disables transfer by deactivating the Run bit (bit0) in the Control registers (`0x0004` and `0x1004`) for the H2C and C2H engines.

### *AXI4-Stream Interface*

For AXI4-Stream, the example design is a loopback design. Channel H2C_0 is looped back to C2H_0 (and so on) for all other channels. 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 is created in the Host memory. The H2C descriptor gives the data length 128 bytes, Source address (host), and Destination address (Card).
3. The test case writes data (incremental 128 bytes of data) in the Host source address space.
4. The test case also sets up one descriptor for the C2H engine in Host memory.
5. The C2H descriptor gives data length 128 bytes, source address (Card), and destination address (host).
6. Write C2H descriptor starting address to register (`0x5080` and `0x5084`).
7. Write 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. Write H2C descriptor starting address to register (`0x4080` and `0x4084`).
10. Write to the H2C control register to start H2C 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 read data from the Card and writes it back to the Host memory destination address.
14. The test case checks for the H2C and C2H descriptor completed count (value of 1).
15. The test case then compares the data for correctness.
16. The test case then disables transfer by deactivating the Run bit (bit 0) in the Control registers `0x0004` and `0x1004` for the H2C and C2H engines.

## Descriptor Bypass Mode

Simulation for Descriptor bypass mode is possible when Channel 0 of both H2C and C2H are selected for descriptor bypass option. The example design generated has one descriptor ready to pump in the Descriptor bypass mode interface.

### *AXI-MM Descriptor Bypass Mode Simulation*

1. The example design has a predefined descriptor for the H2C and C2H engine.
2. The H2C descriptor has 128 bytes of data, source address (Host) and destination address (Card).
3. The C2H descriptor has 128 bytes of data, source address (Card) and destination address (Host).
4. The test case writes incremental 128 bytes of data to the Host memory source address.
5. The PIO writes to the H2C engine Control register to start the transfer (`0x0004`).
6. The DMA reads data from the Host address and sends it to the Card block RAM destination address.
7. The PIO writes to the C2H engine Control register to start the transfer (`0x1004`).
8. The DMA reads data from the Card block RAM source address and sends it to the Host destination address.
9. The test case compares data for correctness.
10. The test case checks for the H2C and C2H descriptor completed count (value of 1).
11. The test case then disables the transfer by deasserting the Run bit (bit 0) in the Control register for the H2C and C2H engine (`0x0004` and `0x1004`).

### *AXI-Stream Descriptor Bypass Mode Simulation with Loopback Design*

1. The example design has a predefined descriptor for the H2C and C2H engine.
2. The H2C descriptor has 128 bytes of data, source address (Host) and destination address (Card).
3. The C2H descriptor has 128 bytes of data, source address (Card) and destination address (Host).
4. The test case writes incremental 128 bytes of data to Host memory source address.
5. The PIO writes to the C2H engine Control register to start the transfer (`0x1004`).
6. The C2H engine starts the DMA transfer but waits for data (loopback design).
7. The PIO writes to the H2C engine Control register to start the transfer (`0x0004`).
8. The H2C engine reads data from the Host address and sends it to Card.
9. The data is looped back to the C2H engine.
10. The C2H engine reads data from the Card and sends it to the Host destination address.
11. The test case compares data for correctness.
12. The test case checks for the H2C and C2H descriptor completed count (value of 1).

13. The test case then disables the transfer by deasserting the Run bit (bit 0) in the Control register for the H2C and C2H engine (`0x0004` and `0x1004`).

When the transfer is started, one H2C and one C2H descriptor are transferred in Descriptor bypass interface and after that DMA transfers are performed as explained in above section. Descriptor is setup for 64 bytes transfer only.

## *Simulation Updates*

Following is an overview of how existing root port tasks can be modified to exercise multi-channels, and multi descriptor cases.

### Multi-Channels Simulation, Example Channel 1 H2C and C2H

1. Create an H2C Channel 1 descriptor in the Host memory address that is different than the H2C and C2H Channel 0 descriptor.
2. Create a C2H Channel 1 descriptor in the Host memory address that is different than the H2C and C2H Channel 0 and H2C Channel 1 descriptor.
3. Create transfer data (128 Bytes) for the H2C Channel 1 transfer in the Host memory which does not overwrite any of the 4 descriptors in the Host memory (H2C and C2H Channel 0 and Channel 1 descriptors), and H2C Channel 0 data.
4. Also make sure the H2C data in the Host memory does not overlap the C2H data transfer space for both C2H Channel 0 and 1.
5. Write the descriptor starting address to H2C Channel 0 and 1.
6. Enable multi-channel transfer by writing to control register (bit 0) of H2C Channel 0 and 1.
7. Enable multi-channel transfer by writing to control register (bit 0) of C2H Channel 0 and 1.
8. Compare the data for correctness.

The same procedure applies for AXI-Stream configuration. Refer to the above section for detailed explanation of the AXI-Stream transfer.

### Multi Descriptor Simulation

1. Create a transfer of 256 bytes data (incremental or any data). Split the data into two 128 bytes of data section. First, the data starts at address S1, and second, 128 bytes starts at address S2.
2. Create a new descriptor (named DSC_H2C_1) in the Host memory address at DSC1.
3. The DSC_H2C_1 descriptor has 128 bytes for DMA transfer, Host address S1 (source) and destination address D1 (card).
4. Create a new descriptor (named DSC_H2C_2) in the Host memory at address DSC2 that is different from DSC_H2C_1 Descriptor.

5. The DSC_H2C_2 descriptor has 128 bytes for DMA transfer, Host address S2 (source) and destination address D2 (card).
6. Link these two descriptors by adding next descriptor address in DSC_H2C_1. Write DSC2 in next descriptor field.
7. Wire the descriptor starting address to H2C Channel 0.
8. Enable DMA transfer for H2C Channel 0 by writing the Run bit in Control register `0x0004`.

## *Test Tasks*

*Table 133:* **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. |
| TSK_XDMA_FIND_BAR | This task finds XDMA configuration space between different enabled BARs (BAR0 to BAR6). |

For other PCIe-related tasks, see the “Test Bench” chapter in the *7 Series FPGAs Integrated Block for PCI Express LogiCORE IP Product Guide* (PG054), *Virtex-7 FPGA Integrated Block for PCI Express LogiCORE IP Product Guide* (PG023), *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156), or *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213).

# Appendix A

# Application Software Development

This section provides details about the Linux device driver and the Windows driver lounge that is provided with the core. For additional information about the driver, see Xilinx DMA IP Reference drivers.

## Linux Device Driver

The Linux device driver has the following character device interfaces:

- User character device for access to user components.
- Control character device for controlling DMA/Bridge Subsystem for PCI Express® 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/Bridge Subsystem for PCI Express® 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 IRQ Block Channel Vector Number (0xA0)) and IRQ Block User Vector (see IRQ Block User Vector Number (0x80)). 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, which consists of a maximum of 16 usable DMA interrupt vectors and a maximum of 16 usable user interrupt vectors. 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 IRQ Block Channel Vector Number (0xA0)) and in the IRQ Block User Vector (see Table IRQ Block User Vector Number (0x80)) 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, which consists of a maximum of 16 usable DMA interrupt vectors and a maximum of 16 usable user interrupt vectors. The DMA has 32 MSI-X tables, one for each source (see Table MSI-X Vector Table and PBA (0x00–0xFE0)). For MSI-X channel interrupt processing the driver should use the Engine's Interrupt Enable Mask for H2C and C2H (see Table H2C Channel Interrupt Enable Mask (0x90) or Table C2H Channel Interrupt Enable Mask (0x90)) to disable and enable interrupts.

## User Interrupts

The user logic must hold `usr_irq_req` active-High even after receiving `usr_irq_ack` (acks) to keep the interrupt pending register asserted. This enables the Interrupt Service Routine (ISR) within the driver to determine the source of the interrupt. Once the driver receives user interrupts, the driver or software can reset the user interrupts source to which hardware should respond by deasserting `usr_irq_req`.

# 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 follows:

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 drive returns transfer byte length to 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.

# Upgrading

This appendix contains information about upgrading to a more recent version of the IP core.

## New Parameters

The following new parameters are added in the IP in the current release.

*Table 134:* **New Parameters**

| Name | Display Name | Description | Default Value |
|---|---|---|---|
| Shared_Logic_Gtc_7xG2 | Include Shared Logic (Transceiver GT_COMMON) in example design | When selected, includes GT_COMMON block in example design. | False |
| Shared_Logic_Clk_7xG2 | Include Shared Logic (Clocking) in example design | When selected, includes Clock block in example design. | False |
| Shared_Logic_Both_7xG2 | Include Shared Logic in core | When selected, includes both GT_COMMON and Clock blocks in core. | False |
| ecc_en | Enable ECC | Enables ECC. Requires one of the Parity option to be turned on. | False |
| aspm_support | ASPM Support optionality | Indicates ASPM support is Enabled or Disabled. | No_ASPM |

## New Ports

The ports in the following table appear at the boundary when the Internal Shared `GT_COMMON` and Clocking option is selected in the Shared Logic tab for 7 series Gen2 devices.

*Table 135:* **Ports For Shared Logic (Internal Shared GT_COMMON and Clocking Option)**

| Name | Direction | Width |
|---|---|---|
| Int_dclk_out | O | 1 bit |

*Table 135:* **Ports For Shared Logic (Internal Shared GT_COMMON and Clocking Option)** *(cont'd)*

| Name | Direction | Width |
|---|---|---|
| Int_oobclk_out | O | 1 bit |
| Int_pclk_sel_slave | I | 1 bit |
| Int_pclk_out_slave | O | 1 bit |
| Int_pipe_rxusrclk_out | O | 1 bit |
| Int_qplllock_out | O | 2 bits |
| Int_qplloutclk_out | O | 2 bits |
| Int_qplloutrefclk_out | O | 2 bits |
| Int_rxoutclk_out | O | 1 bit |
| Int_userclk1_out | O | 1 bit |
| Int_userclk2_out | O | 1 bit |

The ports in the following table appear at the boundary when the Shared `GT_COMMON` option is selected in the Share Logic tab for 7 series Gen2 devices.

*Table 136:* **Ports For Shared Logic (Shared GT_COMMON Option)**

| Name | Direction | Width |
|---|---|---|
| Qpll_drp_crscode | I | 12 bits |
| Qpll_drp_fsm | I | 18 bits |
| Qpll_drp_done | I | 2 bits |
| Qpll_drp_reset | I | 2 bits |
| Qpll_qplllock | I | 2 bits |
| Qpll_qplloutclk | I | 2 bits |
| Qpll_qplloutrefclk | I | 2 bits |
| Qpll_qplld | O | 1 bit |
| Qpll_qpllreset | O | 2 bits |
| Qpll_drp_clk | O | 1 bit |
| Qpll_drp_rst_n | O | 1 bit |
| Qpll_drp_ovrd | O | 1 bit |
| Qpll_drp_gen3 | O | 1 bit |
| Qpll_drp_start | O | 1 bit |

The ports in the following table appear at the boundary when the Shared Clocking option is selected in the Share Logic tab for 7 series Gen2 devices.

*Table 137:* **Ports For Shared Logic (Shared Clocking Option)**

| Name | Direction | Width |
|---|---|---|
| Pipe_pclk_in | I | 1 bit |

*Table 137:* **Ports For Shared Logic (Shared Clocking Option)** *(cont'd)*

| Name | Direction | Width |
|---|---|---|
| Pipe_rxusrclk_in | I | 1 bit |
| Pipe_rxoutclk_in | I | 1 bit |
| Pipe_dclk_in | I | 1 bit |
| Pipe_userclk1_in | I | 1 bit |
| Pipe_userclk2_in | I | 1 bit |
| Pipe_oobclk_in | I | 1 bit |
| Pipe_mmcm_lock_in | I | 1 bit |
| Pipe_mmcm_rst_n | I | 1 bit |
| Pipe_txoutclk_out | O | 1 bit |
| Pipe_rxoutclk_out | O | 1 bit |
| Pipe_pclk_sel_out | O | 1 bit |
| Pipe_gen3_out | O | 1 bit |

The following table shows the new port added in this version of the IP. This port is available at the boundary when MSI-X feature is enabled and the device type is PCIe Endpoint.

*Table 138:* **New Port**

| Name | Direction | Width |
|---|---|---|
| msix_en | O | 1 bit |

*Appendix C*

# Debugging

This appendix includes details about resources available on the Xilinx® Support website and debugging tools.

## Finding Help on Xilinx.com

To help in the design and debug process when using the core, the Xilinx Support web page contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support. The Xilinx Community Forums are also available where members can learn, participate, share, and ask questions about Xilinx solutions.

### Documentation

This product guide is the main document associated with the core. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page or by using the Xilinx® Documentation Navigator. Download the Xilinx Documentation Navigator from the Downloads page. For more information about this tool and the features available, open the online help after installation.

### Solution Centers

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

See the Xilinx Solution Center for PCI Express for the DMA/Bridge Subsystem for PCIe.

### Answer Records

Answer Records include information about commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a Xilinx product. Answer Records are created and maintained daily ensuring that users have access to the most accurate information available.

Answer Records for this core can be located by using the Search Support box on the main Xilinx support web page. To maximize your search results, use keywords such as:

- Product name
- Tool message(s)
- Summary of the issue encountered

A filter search is available after results are returned to further target the results.

### *Master Answer Record for the DMA/Bridge Subsystem for PCIe*

AR 65443

## Technical Support

Xilinx provides technical support on the Xilinx Community Forums for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support if you do any of the following:

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

To ask questions, navigate to the Xilinx Community Forums.

# Debug Tools

There are many tools available to address DMA/Bridge Subsystem for PCIe 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 to interact with the logic debug LogiCORE 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).

## Reference Boards

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

- 7 series FPGA evaluation boards
  - VC709
  - KC705
- UltraScale™ FPGA Evaluation boards
  - KCU105
  - VCU108
- UltraScale+™
  - KCU116
  - VCU118
  - ZCU106

# 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.
- If your outputs go to 0, check your licensing.

## Initial Debug of the DMA/Bridge Subsystem for PCI Express®

Status bits out of each engine can be used for initial debug of the subsystem. Per channel interface provides important status to the user application. See also Table Channel 0-3 Status Ports.

*Table 139:* **Initial Debug of the DMA/Bridge Subsystem for PCI Express®**

| 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_Stop | Descriptor_Done and Stop bit set in the descriptor. |
| 1 | Descriptor_Completed | Descriptor_Done and Completed bit set in the descriptor. |
| 0 | Busy | Channel descriptor buffer is not empty or DMA requests are outstanding. |

# Appendix D

# Using the Xilinx Virtual Cable to Debug

The Xilinx® Virtual Cable (XVC) allows the Vivado® Design Suite to connect to FPGA debug cores through non-JTAG interfaces. The standard Vivado® Design Suite debug feature uses JTAG to connect to physical hardware FPGA resources and perform debug through Vivado. This section focuses on using XVC to perform debug over a PCIe® link rather than the standard JTAG debug interface. This is referred to as XVC-over-PCIe and allows for Vivado ILA waveform capture, VIO debug control, and interaction with other Xilinx debug cores using the PCIe link as the communication channel.

XVC-over-PCIe should be used to perform FPGA debug remotely using the Vivado Design Suite debug feature when JTAG debug is not available. This is commonly used for data center applications where the FPGA is connected to a PCIe Host system without any other connections to the hardware device.

Using debug over XVC requires software, driver, and FPGA hardware design components. Since there is an FPGA hardware design component to XVC-over-PCIe debug, you cannot perform debug until the FPGA is already loaded with an FPGA hardware design that implements XVC-over-PCIe and the PCIe link to the Host PC is established. This is normally accomplished by loading an XVC-over-PCIe enabled design into the configuration flash on the board prior to inserting the card into the data center location. Since debug using XVC-over-PCIe is dependent on the PCIe communication channel this should not be used to debug PCIe link related issue.

> **IMPORTANT!** *XVC only provides connectivity to the debug cores within the FPGA. It does not provide the ability to program the device or access device JTAG and configuration registers. These operations can be performed through other standard Xilinx interfaces or peripherals such as the PCIe MCAP VSEC and HWICAP IP.*

## Overview

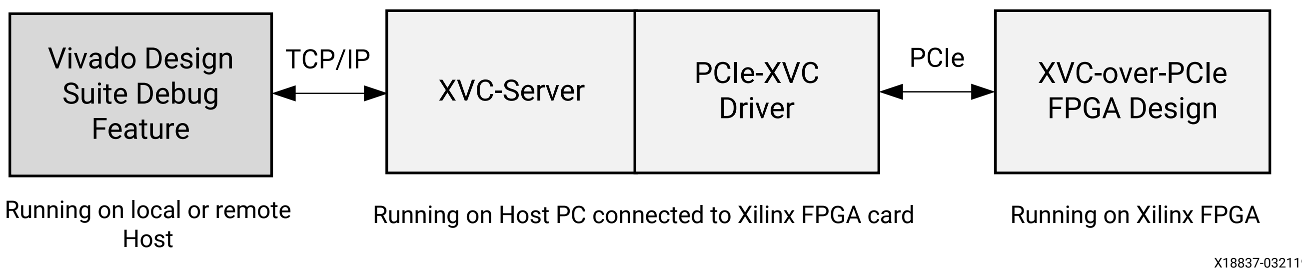
The main components that enable XVC-over-PCIe debug are as follows:

- Host PC XVC-Server application
- Host PC PCIe-XVC driver

- XVC-over-PCIe enabled FPGA design

These components are provided as a reference on how to create XVC connectivity for Xilinx FPGA designs. These three components are shown in the following figure and connect to the Vivado Design Suite debug feature through a TCP/IP socket.

*Figure 33:* **XVC-over-PCIe Software and Hardware Components**




# Host PC XVC-Server Application

The `hw_server` application is launched by Vivado Design Suite when using the debug feature. Through the Vivado IDE you can connect hw_server to local or remote FPGA targets. This same interface is used to connect to local or remote PCIe-XVC targets as well. The Host PCIe XVC-Server application connects to the Xilinx `hw_server` using TCP/IP socket. This allows Vivado (using `hw_server`) and the XVC-Server application to be running on the same PC or separate PCs connected through Ethernet. The XVC-Server application needs to be run on a PC that is directly connected to the FPGA hardware resource. In this scenario the FPGA hardware is connected through PCIe® to a Host PC. The XVC-Server application connects to the FPGA hardware device through the PCIe-XVC driver that is also running on the Host PC.

# Host PC XVC-over-PCIe Driver

The XVC-over-PCIe driver provides connectivity to the PCIe enabled FPGA hardware resource that is connected to the Host PC. As such this is provided as a Linux kernel mode driver to access the PCIe hardware device, which is available in the following location, `<Vivado_Installation_Path>/data/xicom/driver/pcie/xvc_pcie.zip`. The necessary components of this driver must be added to the driver that is created for a specific FPGA platform. The driver implements the basic functions needed by the XVC-Server application to communicate with the FPGA via PCIe.

# XVC-over-PCIe Enabled FPGA Design

Traditionally Vivado® debug is performed over JTAG. By default, Vivado tool automation connects the Xilinx debug cores to the JTAG BSCAN resource within the FPGA to perform debug. In order to perform XVC-over-PCIe debug, this information must be transmitted over the PCIe link rather than over the JTAG cable interface. The Xilinx Debug Bridge IP allows you to connect the debug network to PCIe through either the PCIe extended configuration interface (PCIe-XVC-VSEC) or through a PCIe BAR via an AXI4-Lite Memory Mapped interface (AXI-XVC).

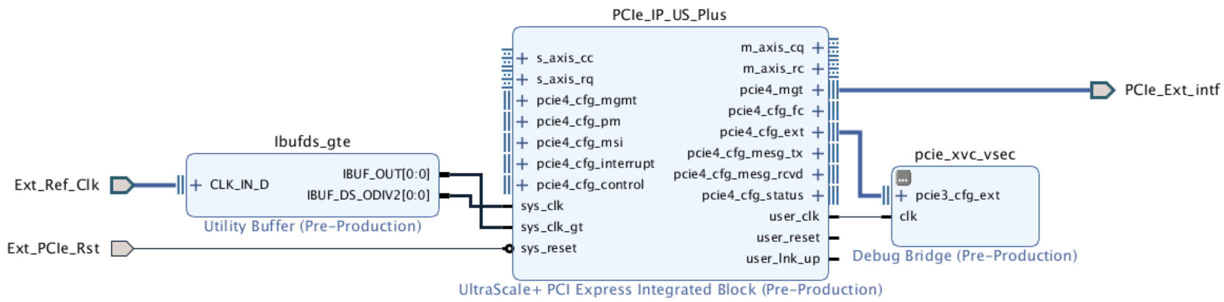
The Debug Bridge IP, when configured for **From PCIe to BSCAN** or **From AXI to BSCAN**, provides a connection point for the Xilinx® debug network from either the PCIe Extended Capability or AXI4-Lite interfaces respectively. Vivado tool automation connects this instance of the Debug Bridge to the Xilinx debug cores found in the design rather than connecting them to the JTAG BSCAN interface. There are design trade-offs to connecting the debug bridge to the PCIe Extended Configuration Space or AXI4-Lite. The following sections describe the implementation considerations and register map for both implementations.

## XVC-over-PCIe Through PCIe Extended Configuration Space (PCIe-XVC-VSEC)

Using the PCIe-XVC-VSEC approach, the Debug Bridge IP uses a PCIe Vendor Specific Extended Capability (VSEC) to implement the connection from PCIe to the Debug Bridge IP. The PCIe extended configuration space is set up as a linked list of extended capabilities that are discoverable from a Host PC. This is specifically valuable for platforms where one version of the design implements the PCIe-XVC-VSEC and another design implementation does not. The linked list can be used to detect the existence or absence of the PCIe-XVC-VSEC and respond accordingly.

The PCIe Extended Configuration Interface uses PCIe configuration transactions rather than PCIe memory BAR transactions. While PCIe configuration transactions are much slower, they do not interfere with PCIe memory BAR transactions at the PCIe IP boundary. This allows for separate data and debug communication paths within the FPGA. This is ideal if you expect to debug the datapath. Even if the datapath becomes corrupt or halted, the PCIe Extended Configuration Interface can remain operational to perform debug. The following figure describes the connectivity between the PCIe IP and the Debug Bridge IP to implement the PCIe-XVC-VSEC.

*Figure 34:* **XVC-over-PCIe with PCIe Extended Capability Interface**

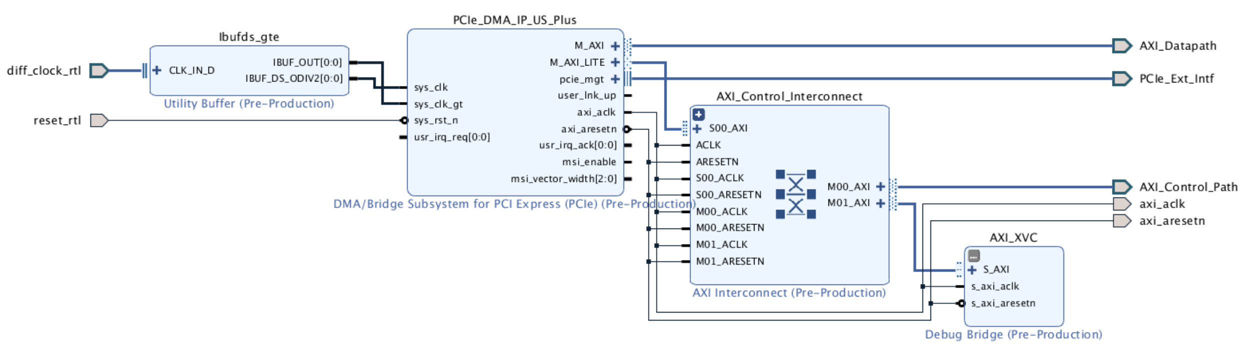



***Note*****:** Although the previous figure shows the UltraScale+™ Integrated Block for PCIe IP, other PCIe IP (that is, the UltraScale™ Integrated Block for PCIe, AXI Bridge for PCIe, or PCIe DMA IP) can be used interchangeably in this diagram.

# XVC-over-PCIe Through AXI (AXI-XVC)

Using the AXI-XVC approach, the Debug Bridge IP connects to the PCIe IP through an AXI Interconnect IP. The Debug Bridge IP connects to the AXI Interconnect like other AXI4-Lite Slave IPs and similarly requires that a specific address range be assigned to it. Traditionally the `debug_bridge` IP in this configuration is connected to the control path network rather than the system datapath network. The following figure describes the connectivity between the DMA Subsystem for PCIe IP and the Debug Bridge IP for this implementation.

*Figure 35:* **XVC over PCIe with AXI4-Lite Interface**




***Note*****:** Although the previous figure shows the PCIe DMA IP, any AXI-enabled PCIe IP can be used interchangeably in this diagram.

The AXI-XVC implementation allows for higher speed transactions. However, XVC debug traffic passes through the same PCIe ports and interconnect as other PCIe control path traffic, making it more difficult to debug transactions along this path. As result the AXI-XVC debug should be used to debug a specific peripheral or a different AXI network rather than attempting to debug datapaths that overlap with the AXI-XVC debug communication path.

## XVC-over-PCIe Register Map

The PCIe-XVC-VSEC and AXI-XVC have a slightly different register map that must be taken into account when designing XVC drivers and software. The register maps in the following tables show the byte-offset from the base address.

- The PCIe-XVC-VSEC base address must fall within the valid range of the PCIe Extended Configuration space. This is specified in the Debug Bridge IP configuration.
- The base address of an AXI-XVC Debug Bridge is the offset for the Debug Bridge IP peripheral that was specified in the Vivado Address Editor.

The following tables describe the register map for the Debug Bridge IP as an offset from the base address when configured for the **From PCIe-Ext to BSCAN** or **From AXI to BSCAN** modes.

*Table 140:* **Debug Bridge for XVC-PCIe-VSEC Register Map**

| Register Offset | Register Name | Description | Register Type |
|---|---|---|---|
| 0x00 | PCIe Ext Capability Header | PCIe defined fields for VSEC use. | Read Only |
| 0x04 | PCIe VSEC Header | PCIe defined fields for VSEC use. | Read Only |
| 0x08 | XVC Version Register | IP version and capabilities information. | Read Only |
| 0x0C | XVC Shift Length Register | Shift length. | Read Write |
| 0x10 | XVC TMS Register | TMS data. | Read Write |
| 0x14 | XVC TDIO Register | TDO/TDI data. | Read Write |
| 0x18 | XVC Control Register | General control register. | Read Write |
| 0x1C | XVC Status Register | General status register. | Read Only |

*Table 141:* **Debug Bridge for AXI-XVC Register Map**

| Register Offset | Register Name | Description | Register Type |
|---|---|---|---|
| 0x00 | XVC Shift Length Register | Shift length. | Read Write |
| 0x04 | XVC TMS Register | TMS data. | Read Write |
| 0x08 | XVC TDI Register | TDI data. | Read Write |
| 0x0C | XVC TDO Register | TDO data. | Read Only |
| 0x10 | XVC Control Register | General control register. | Read Write |
| 0x14 | XVC Status Register | General status register. | Read Only |
| 0x18 | XVC Version Register | IP version and capabilities information. | Read Only |

## PCIe Ext Capability Header

This register is used to identify the PCIe-XVC-VSEC added to a PCIe design. The fields and values in the PCIe Ext Capability Header are defined by PCI-SIG and are used to identify the format of the extended capability and provide a pointer to the next extended capability, if applicable. When used as a PCIe-XVC-VSEC, the appropriate PCIe ID fields should be evaluated prior to interpretation. These can include PCIe Vendor ID, PCIe Device ID, PCIe Revision ID, Subsystem Vendor ID, and Subsystem ID. The provided drivers specifically check for a PCIe Vendor ID that matches Xilinx (`0x10EE`) before interpreting this register. The following table describes the fields within this register.

*Table 142:* **PCIe Ext Capability Header Register Description**

| Bit Location | Field | Description | Initial Value | Type |
|---|---|---|---|---|
| 15:0 | PCIe Extended Capability ID | This field is a PCI-SIG defined ID number that indicates the nature and format of the Extended Capability. The Extended Capability ID for a VSEC is `0x000B` | 0x000B | Read Only |
| 19:16 | Capability Version | This field is a PCI-SIG defined version number that indicates the version of the capability structure present. Must be `0x1` for this version of the specification. | 0x1 | Read Only |
| 31:20 | Next Capability Offset | This field is passed in from the user and contains the offset to the next PCI Express Capability structure or 0x000 if no other items exist in the linked list of capabilities. For Extended Capabilities implemented in the PCIe extended configuration space, this value must always be within the valid range of the PCIe Extended Configuration space. | 0x000 | Read Only |

## PCIe VSEC Header (PCIe-XVC-VSEC only)

This register is used to identify the PCIe-XVC-VSEC when the Debug Bridge IP is in this mode. The fields are defined by PCI-SIG, but the values are specific to the Vendor ID (0x10EE for Xilinx). The PCIe Ext Capability Header register values should be qualified prior to interpreting this register.

*Table 143:* **PCIe XVC VSEC Header Register Description**

| Bit Location | Field | Description | Initial Value | Type |
|---|---|---|---|---|
| 15:0 | VSEC ID | This field is the ID value that can be used to identify the PCIe-XVC-VSEC and is specific to the Vendor ID (0x10EE for Xilinx). | 0x0008 | Read Only |
| 19:16 | VSEC Rev | This field is the Revision ID value that can be used to identify the PCIe-XVC-VSEC revision. | 0x0 | Read Only |
| 31:20 | VSEC Length | This field indicates the number of bytes in the entire PCIe-XVC-VSEC structure, including the PCIe Ext Capability Header and PCIe VSEC Header registers. | 0x020 | Read Only |

### *XVC Version Register (PCIe-XVC-VSEC only)*

This register is populated by the Xilinx tools and is used by the Vivado Design Suite to identify the specific features of the Debug Bridge IP that is implemented in the hardware design.

### *XVC Shift Length Register*

This register is used to set the scan chain shift length within the debug scan chain.

### *XVC TMS Register*

This register is used to set the TMS data within the debug scan chain.

### *XVC TDO/TDI Data Register(s)*

This register is used for TDO/TDI data access. When using PCIePCI-XVC-VSEC, these two registers are combined into a single field. When using AXI-XVC, these are implemented as two separate registers.

### *XVC Control Register*

This register is used for XVC control data.

### *XVC Status Register*

This register is used for XVC status information.

## XVC Driver and Software

Example XVC driver and software has been provided with the Vivado Design Suite installation, which is available at the following location: `<Vivado_Installation_Path>/data/xicom/driver/pcie/xvc_pcie.zip`. This should be used for reference when integrating the XVC capability into Xilinx FPGA platform design drivers and software. The provided Linux kernel mode driver and software implement XVC-over-PCIe debug for both PCIe-XVC-VSEC and AXI-XVC debug bridge implementations.

When operating in PCIe-XVC-VSEC mode, the driver will initiate PCIe configuration transactions to interface with the FPGA debug network. When operating in AXI-XVC mode, the driver will initiate 32-bit PCIe Memory BAR transactions to interface with the FPGA debug network. By default, the driver will attempt to discover the PCIe-XVC-VSEC and use AXI-XVC if the PCIe-XVC-VSEC is not found in the PCIe configuration extended capability linked list.

The driver is provided in the data directory of the Vivado installation as a `.zip` file. This `.zip` file should be copied to the Host PC connected through PCIe to the Xilinx FPGA and extracted for use. `README.txt` files have been included; review these files for instructions on installing and running the XVC drivers and software.

## Special Considerations for Tandem or Dynamic Function eXchange Designs

Tandem Configuration and Dynamic Function eXchange (DFX) designs may require additional considerations as these flows partition the physical resources into separate regions. These physical partitions should be considered when adding debug IPs to a design, such as VIO, ILA, MDM, and MIG-IP. A Debug Bridge IP configured for **From PCIe-ext to BSCAN** or **From AXI to BSCAN** should only be placed into the static partition of the design. When debug IPs are used inside of a DFX or Tandem Field Updates region, an additional debug BSCAN interface should be added to the dynamic region module definition and left unconnected in the dynamic region module instantiation.

To add the BSCAN interface to the Reconfigurable Partition definition the appropriate ports and port attributes should be added to the Reconfigurable Partition definition. The sample Verilog provided below can be used as a template for adding the BSCAN interface to the port declaration.

```
...
// BSCAN interface definition and attributes.
// This interface should be added to the DFX module definition
// and left unconnected in the DFX module instantiation.
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN drck" *)
(* DEBUG="true" *)
input S_BSCAN_drck,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN shift" *)
(* DEBUG="true" *)
input S_BSCAN_shift,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN tdi" *)
(* DEBUG="true" *)
input S_BSCAN_tdi,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN update" *)
(* DEBUG="true" *)
input S_BSCAN_update,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN sel" *)
(* DEBUG="true" *)
input S_BSCAN_sel,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN tdo" *)
(* DEBUG="true" *)
output S_BSCAN_tdo,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN tms" *)
(* DEBUG="true" *)
input S_BSCAN_tms,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN tck" *)
(* DEBUG="true" *)
input S_BSCAN_tck,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN runtest" *)
(* DEBUG="true" *)
input S_BSCAN_runtest,
```

```
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN reset" *)
(* DEBUG="true" *)
input S_BSCAN_reset,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN capture" *)
(* DEBUG="true" *)
input S_BSCAN_capture,
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN bscanid_en" *)
(* DEBUG="true" *)
input S_BSCAN_bscanid_en,
....
```

When `link_design` is run, the exposed ports are connected to the static portion of the debug network through tool automation. The ILAs are also connected to the debug network as required by the design. There might also be an additional `dbg_hub` cell that is added at the top level of the design. For Tandem with Field Updates designs, the `dbg_hub` and tool inserted clock buffer(s) must be added to the appropriate design partition. The following is an example of the Tcl commands that can be run after `opt_design` to associate the `dbg_hub` primitives with the appropriate design partitions.

```
# Add the inserted dbg_hub cell to the appropriate design partition.
set_property HD.TANDEM_IP_PBLOCK Stage1_Main [get_cells dbg_hub]
# Add the clock buffer to the appropriate design partition.
set_property HD.TANDEM_IP_PBLOCK Stage1_Config_IO [get_cells
dma_pcie_0_support_i/
pcie_ext_cap_i/vsec_xvc_inst/vsec_xvc_dbg_bridge_inst/inst/bsip/ins
t/USE_SOFTBSCAN.U_TAP_TCKBUFG]
```

# Using the PCIe-XVC-VSEC Example Design

The PCIe-XVC-VSEC has been integrated into the PCIe example design as part of the Advanced settings for the UltraScale+™ Integrated Block for PCIe IP. This section provides instruction of how to generate the PCIe example design with the PCIe-XVC-VSEC, and then debug the FPGA through PCIe using provided XVC drivers and software. This is an example for using XVC in customer applications. The FPGA design, driver, and software elements will need to be integrated into customer designs.

## Generating a PCIe-XVC-VSEC Example Design

The PCIe-XVC-VSEC can be added to the UltraScale+™ PCIe example design by selecting the following options.

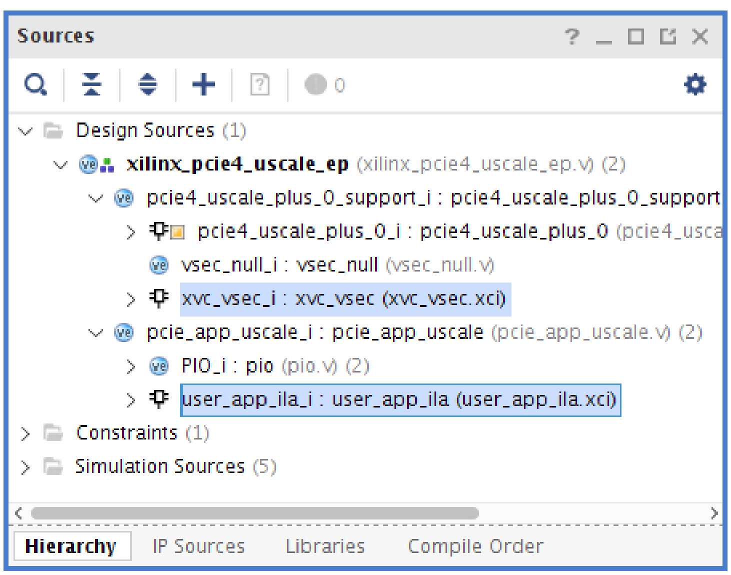
1. Configure the core to the desired configuration.
2. On the Basic tab, select the **Advanced** Mode.

3. On the Adv. Options-3 tab:
    a. Select the **PCI Express Extended Configuration Space Enable** checkbox to enable the PCI Express extended configuration interface. This is where additional extended capabilities can be added to the PCI Express core.
    b. Select the **Add the PCIe-XVC-VSEC to the Example Design** checkbox to enable the PCIe-XVC-VSEC in the example design generation.
4. Verify the other configuration selections for the PCIe IP. The following selections are needed to configure the driver for your hardware implementation:
    - PCIe Vendor ID (`0x10EE` for Xilinx)
    - PCIe Device ID (dependent on user selection)
5. Click **OK** to finalize the selection and generate the IP.
6. Generate the output products for the IP as desired for your application.
7. In the Sources window, right-click the IP and select **Open IP Example Design**.
8. Select a directory for generating the example design, and select **OK**.

    After being generated, the example design shows that:

    - the PCIe IP is connected to `xvc_vsec` within the support wrapper, and
    - an ILA IP is added to the user application portion of the design.

    This demonstrates the desired connectivity for the hardware portion of the FPGA design. Additional debug cores can be added as required by your application.




***Note*:** Although the previous figure shows to the UltraScale+ Integrated Block for PCIe IP, the example design hierarchy is the same for other PCIe IPs.

9. Double-click the Debug Bridge IP identified as `xvc_vsec` to view the configuration option for this IP. Make note of the following configuration parameters because they will be used to configure the driver.

   - PCIe XVC VSEC ID (default `0x0008`)
   - PCIe XVC VSEC Rev ID (default `0x0`)

   > **IMPORTANT!** *Do not modify these parameter values when using a Xilinx Vendor ID or provided XVC drivers and software. These values are used to detect the XVC extended capability. (See the PCIe specification for additional details.)*

10. In the Flow Navigator, click **Generate Bitstream** to generate a bitstream for the example design project. This bitstream will be then be loaded onto the FPGA board to enable XVC debug over PCIe.

After the XVC-over-PCIe hardware design has been completed, an appropriate XVC enabled PCIe driver and associated XVC-Server software application can be used to connect the Vivado Design Suite to the PCIe connected FPGA. Vivado can connect to an XVC-Server application that is running local on the same Machine or remotely on another machine using a TCP/IP socket.

## System Bring-Up

The first step is to program the FPGA and power on the system such that the PCIe link is detected by the host system. This can be accomplished by either:

- Programming the design file into the flash present on the FPGA board, or
- Programming the device directly via JTAG.

If the card is powered by the Host PC, it will need to be powered on to perform this programming using JTAG and then re-started to allow the PCIe link to enumerate. After the system is up and running, you can use the Linux `lspci` utility to list out the details for the FPGA-based PCIe device.

## Compiling and Loading the Driver

The provided PCIe drivers and software should be customized to a specific platform. To accomplish this, drivers and software are normally developed to verify the Vendor ID, Device ID, Revision ID, Subsystem Vendor ID, and Subsystem ID before attempting to access device-extended capabilities or peripherals like the PCIe-XVC-VSEC or AXI-XVC. Because the provided driver is generic, it only verifies the Vendor ID and Device ID for compatibility before attempting to identify the PCIe-XVC-VSEC or AXI-XVC peripheral.

The XVC driver and software are provide as a ZIP file included with the Vivado Design Suite installation.

1. Copy the ZIP file from the Vivado install directory to the FPGA connected Host PC and extract (unzip) its contents. This file is located at the following path within the Vivado installation directory.

   ```
   XVC Driver and SW Path: …/data/xicom/driver/pcie/xvc_pcie.zip
   ```

   The `README.txt` files within the `driver_*` and `xvcserver` directories identify how to compile, install, and run the XVC drivers and software, and are summarized in the following steps. Follow the following steps after the driver and software files have been copied to the Host PC and you are logged in as a user with root permissions.

2. Modify the variables within the `driver_*/xvc_pcie_user_config.h` file to match your hardware design and IP settings. Consider modifying the following variables:

   - **PCIE_VENDOR_ID:** The PCIe Vendor ID defined in the PCIe® IP customization.
   - **PCIE_DEVICE_ID:** The PCIe Device ID defined in the PCIe® IP customization.
   - **Config_space:** Allows for the selection between using a PCIe-XVC-VSEC or an AXI-XVC peripheral. The default value of AUTO first attempts to discover the PCIe-XVC-VSEC, then attempts to connect to an AXI-XVC peripheral if the PCIe-XVC-VSEC is not found. A value of CONFIG or BAR can be used to explicitly select between PCIe®-XVC-VSEC and AXI-XVC implementations, as desired.
   - **config_vsec_id:** The PCIe XVC VSEC ID (default `0x0008`) defined in the Debug Bridge IP when the Bridge Type is configured for From PCIE to BSCAN. This value is only used for detection of the PCIe®-XVC-VSEC.
   - **config_vsec_rev:** The PCIe XVC VSEC Rev ID (default `0x0`) defined in the Debug Bridge IP when the Bridge Type is configured for From PCIe to BSCAN. This value is only used for detection of the PCIe-XVC-VSEC.
   - **bar_index:** The PCIe BAR index that should be used to access the Debug Bridge IP when the Bridge Type is configured for From AXI to BSCAN. This BAR index is specified as a combination of the PCIe IP customization and the addressable AXI peripherals in your system design. This value is only used for detection of an AXI-XVC peripheral.
   - **bar_offset:** PCIe BAR Offset that should be used to access the Debug Bridge IP when the Bridge Type is configured for From AXI to BSCAN. This BAR offset is specified as a combination of the PCIe IP customization and the addressable AXI peripherals in your system design. This value is only used for detection of an AXI-XVC peripheral.

3. Move the source files to the directory of your choice. For example, use:

   ```
   /home/username/xil_xvc or /usr/local/src/xil_xvc
   ```

4. Make sure you have root permissions and change to the directory containing the driver files.

   ```
   # cd /driver_*/
   ```

5. Compile the driver module:

   ```
   # make install
   ```

   The kernel module object file will be installed as:

```
/lib/modules/[KERNEL_VERSION]/kernel/drivers/pci/pcie/Xilinx/
xil_xvc_driver.ko
```

6. Run the `depmod` command to pick up newly installed kernel modules:

```
# depmod -a
```

7. Make sure no older versions of the driver are loaded:

```
# modprobe -r xil_xvc_driver
```

8. Load the module:

```
# modprobe xil_xvc_driver
```

If you run the `dmesg` command, you will see the following message:

```
kernel: xil_xvc_driver: Starting…
```

***Note***: You can also use `insmod` on the kernel object file to load the module:

```
# insmod xil_xvc_driver.ko
```

However, this is not recommended unless necessary for compatibility with older kernels.

9. The resulting character file, `/dev/xil_xvc/cfg_ioc0`, is owned by user root and group root, and it will need to have permissions of `660`. Change permissions on this file if it does not allow the application to interact with the driver.

```
# chmod 660 /dev/xil_xvc/cfg_ioc0
```

10. Build the simple test program for the driver:

```
# make test
```

11. Run the test program:

```
# ./driver_test/verify_xil_xvc_driver
```

You should see various successful tests of differing lengths, followed by the following message:

```
"XVC PCIE Driver Verified Successfully!"
```

## Compiling and Launching the XVC-Server Application

The XVC-Server application provides the connection between the Vivado HW server and the XVC enabled PCIe device driver. The Vivado Design Suite connects to the XVC-Server using TCP/IP. The desired port number will need to be exposed appropriately through the firewalls for your network. The following steps can be used to compile and launch the XVC software application, using the default port number of `10200`.

1. Make sure the firewall settings on the system expose the port that will be used to connect to the Vivado Design Suite. For this example, port `10200` is used.

2. Make note of the host name or IP address. The host name and port number will be required to connect Vivado to the `xvcserver` application. See the OS help pages for information regarding the firewall port settings for your OS.
3. Move the source files to the directory of your choice. For example, use:

   `/home/username/xil_xvc` or `/usr/local/src/xil_xvc`

4. Change to the directory containing the application source files:

   ```
   # cd ./xvcserver/
   ```

5. Compile the application:

   ```
   # make
   ```

6. Start the XVC-Server application:

   ```
   # ./bin/xvc_pcie -s TCP::10200
   ```

   After the Vivado Design Suite has connected to the XVC-server application you should see the following message from the XVC-server.

   ```
   Enable verbose by setting VERBOSE evn var.
   Opening /dev/xil_xvc/cfg_ioc0
   ```

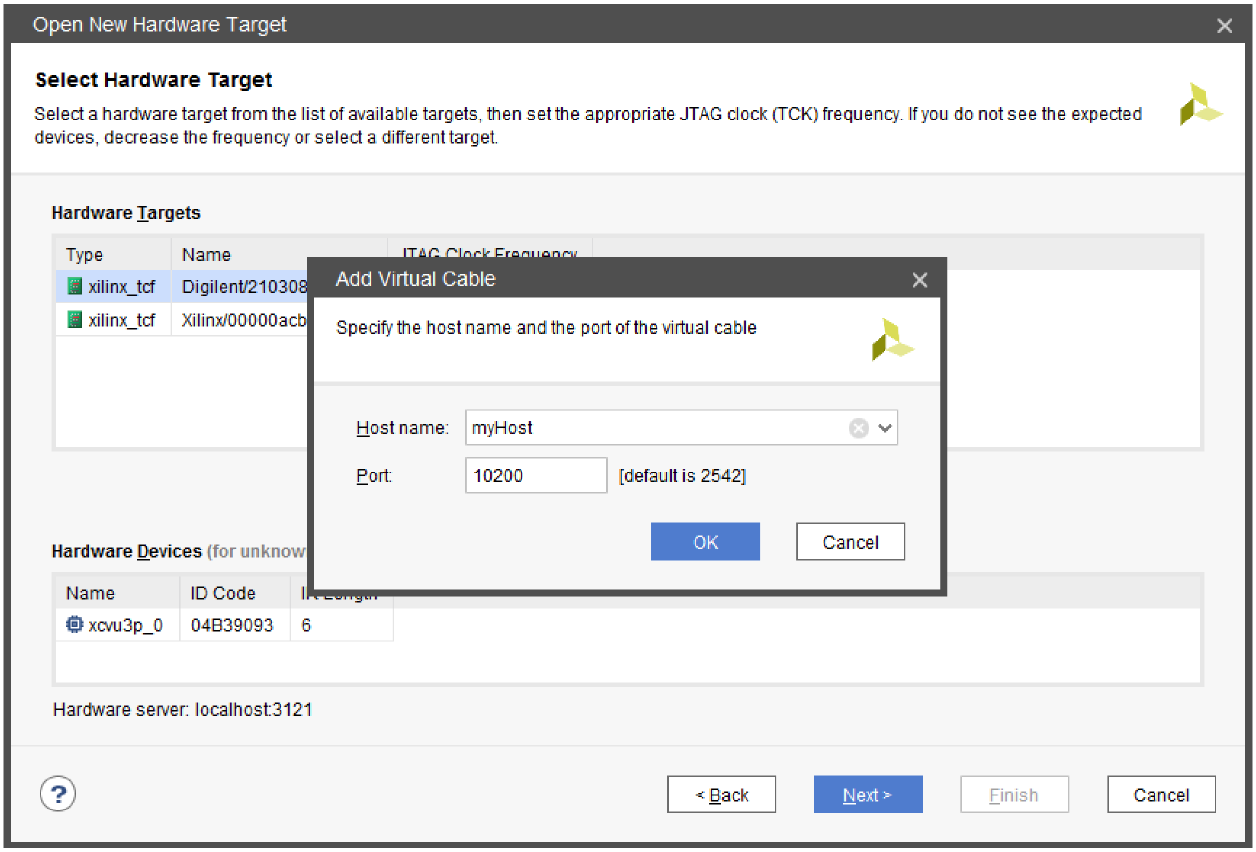
## Connecting the Vivado Design Suite to the XVC-Server Application

The Vivado Design Suite can be run on the computer that is running the XVC-server application, or it can be run remotely on another computer that is connected over an Ethernet network. The port however must be accessible to the machine running Vivado. To connect Vivado to the XVC-Server application follow the steps should be used and are shown using the default port number.

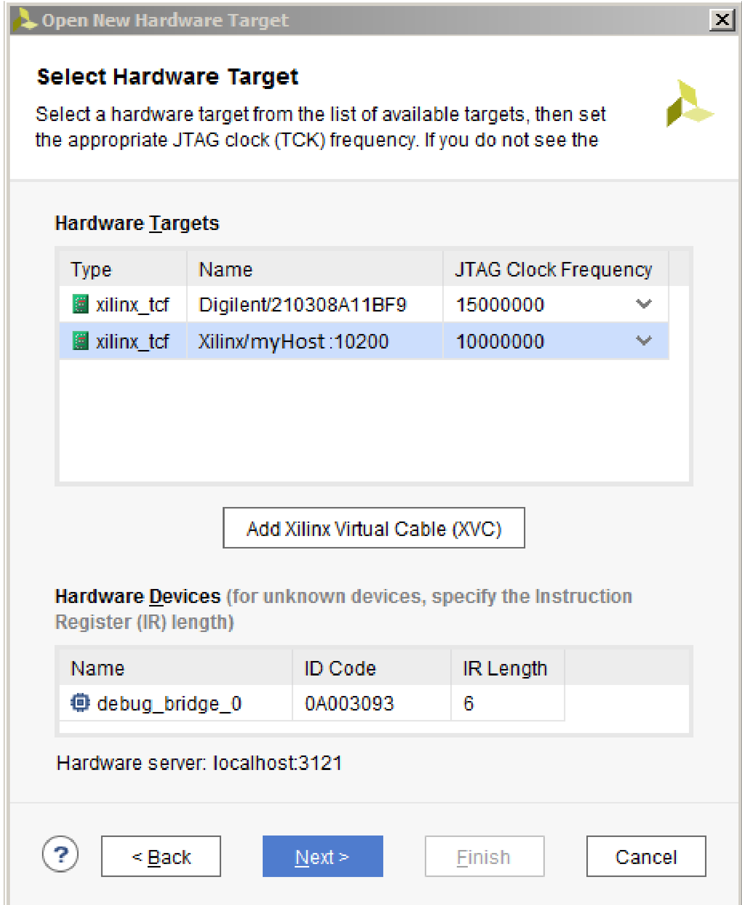
1. Launch the Vivado Design Suite.
2. Select **Open HW Manager**.
3. In the Hardware Manager, select **Open target → Open New Target**.
4. Click **Next**.
5. Select **Local server**, and click **Next**.

   This launches `hw_server` on the local machine, which then connects to the `xvcserver` application.

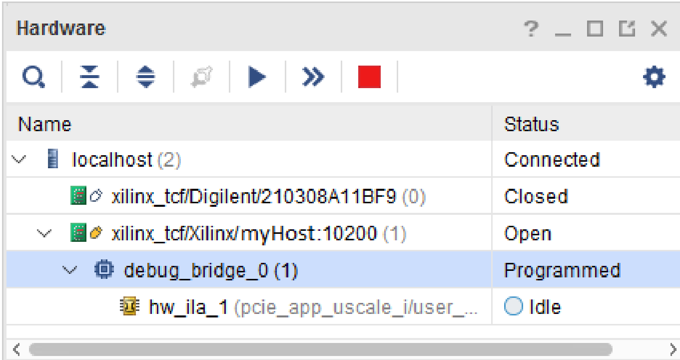
6. Select **Add Xilinx Virtual Cable (XVC)**.
7. In the Add Virtual Cable dialog box, type in the appropriate Host name or IP address, and Port to connect to the `xvcserver` application. Click **OK**.

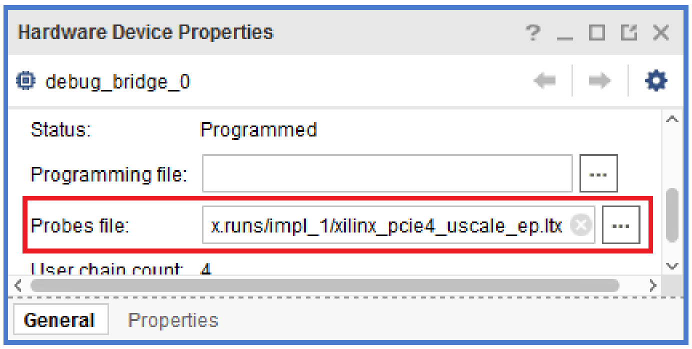



8. Select the newly added XVC target from the Hardware Targets table, and click **Next**.




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




Vivado now recognizes your debug cores and debug signals, and you can debug your design through the Vivado hardware tools interface using the standard debug approach.

This allows you to debug Xilinx FPGA designs through the PCIe connection rather than JTAG using the Xilinx Virtual Cable technology. You can terminate the connection by closing the hardware server from Vivado using the right-click menu. If the PCIe connection is lost or the XVC-Server application stops running, the connection to the FPGA and associated debug cores will also be lost.

## Run Time Considerations

The Vivado connection to an XVC-Server Application should not be running when a device is programmed. The XVC-Server Application along with the associated connection to Vivado should only be initiated after the device has been programmed and the hardware PCIe interface is active.

For DFX designs, it is important to terminate the connection during DFX operations. During a DFX operation where debug cores are present inside the dynamic region, a portion of the debug tree is expected to be reprogrammed. Vivado debug tools should not be actively communicating with the FPGA through XVC during a DFX operation.

*Appendix E*

# Additional Resources and Legal Notices

## Xilinx Resources

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

## Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

***Note*:** For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

## References

These documents provide supplemental material useful with this guide:

1. *AMBA AXI4-Stream Protocol Specification* (ARM IHI 0051A)
2. PCI-SIG Documentation (www.pcisig.com/specifications)
3. *Vivado Design Suite: AXI Reference Guide* (UG1037)
4. *AXI Bridge for PCI Express Gen3 Subsystem Product Guide* (PG194)
5. *7 Series FPGAs Integrated Block for PCI Express LogiCORE IP Product Guide* (PG054)
6. *Virtex-7 FPGA Integrated Block for PCI Express LogiCORE IP Product Guide* (PG023)
7. *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156)
8. *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213)
9. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
10. *Vivado Design Suite User Guide: Designing with IP* (UG896)
11. *Vivado Design Suite User Guide: Getting Started* (UG910)
12. *Vivado Design Suite User Guide: Using Constraints* (UG903)
13. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
14. *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909)
15. *ISE to Vivado Design Suite Migration Guide* (UG911)
16. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
17. *Vivado Design Suite User Guide: Implementation* (UG904)
18. *AXI Interconnect LogiCORE IP Product Guide* (PG059)

# Revision History

The following table shows the revision history for this document.

| Section | Revision Summary |
|---|---|
| **09/21/2020 Version 4.1** | |
| General Updates | Made clarifications throughout. |
| Tandem Configuration | Updated Partial Reconfiguration references to Dynamic Function eXchange. |
| Debug Options Tab | Added new debug options. |
| Parameters for Custom PIPE Simulation | Added guidance for required PIPE Simulation parameters. |
| Appendix A: Application Software Development | Updated link for additional driver information. |
| **11/22/2019 Version 4.1** | |
| Tandem Configuration | Updated Supported Devices. |

| Section | Revision Summary |
|---|---|
| MSI-X Vector Table and PBA (0x8) | Added MSI-X table offset and PBA table offset values. |
| **06/20/2019 Version 4.1** | |
| Tandem Configuration | Updated Supported Devices. |
| DMA C2H Stream | Clarified that C2H Stream descriptor length size must be a multiple of 64 Bytes. |
| IRQ Block Registers (0x2) | Clarified MSI-X Interrupt register description for AXI Bridge mode. |
| Customizing and Generating the Subsystem | Updated screen captures. |
| Basic Tab | Added GT DRP Clock Selection, and Data Protection options. |
| PCIe Misc Tab | Added MSI RX PIN EN option. |
| PCIe DMA Tab | Removed Parity Checking option (moved to Basic tab). |
| Appendix A: Application Software Development | Appendix renamed from Device Driver to Application Software Development. |
| **12/05/2018 Version 4.1** | |
| Chapter 3: Product Specification | In the Minimum Device Requirements table:<br>• Added -2 supported speed grade for Gen3 x16 for architecture devices (PCIE4).<br>• Added Gen4 link speed details for Virtex UltraScale+™ devices with high bandwidth memory (HBM).<br>• Added information about behavior for multiple channels to the H2C Channel section.<br>• Added information about access restrictions to the AXI4-Lite Slave.<br>• Updated the cfg_ext_read_received description. |
| Chapter 4: Designing with the Core | Updated the production support details in the Tandem PROM/PCIeSupported Configurations (UltraScale+ Devices) table. |
| Chapter 7: Test Bench | • Updated the Descriptor Bypass Mode description to reflect that H2C and C2H descriptors have 128 bytes of data.<br>• Added the Tcl command for Post-Synthesis simulation for the example design. |
| Chapter 6: Example Design | Added the User IRQ Example Design |
| **04/04/2018 Version 4.1** | |
| General Updates | Clarified that Tandem Configuration is not yet supported for Bridge mode in UltraScale+ devices. |
| Chapter 2: Overview | Added limitation: For 7 series, PCIe access from Host system must be limited to 1DW (4 Bytes) transaction only. |

| Section | Revision Summary |
|---|---|
| Chapter 3: Product Specification | Added clarifying text to the IRQ Module configuration component (Legacy Interrupts, and MSI and MSI-X Interrupts sections).<br>Editorial updates in the H2C Channel 0-3 AXI4-Stream Interface Signals tables, and C2H Channel 0-3 AXI4-Stream Interface Signals tables.<br>Added the dma_bridge_resetn signal to the Top-Level Interface Signals table.<br>Updated Register Name: IRQ Block Channel Interrupt Pending (0x4C)<br>Added Virtex UltraScale+ Devices with HBM (PCIE4C) minimum device requirements information. |
| Chapter 4: Designing with the Core | Added Virtex UltraScale+ parts to the Tandem PROM/PCIe Supported Configurations (UltraScale+ Devices) table.<br>Added Shared Logic support for 7 series Gen2 family devices. |
| Device Driver Appendix | Added clarifying text to the MSI Interrupt, MSI-X Interrupts, and User Interrupts sections. |
| **12/20/2017 Version 4.0** | |
| General Updates | Updated Minimum Device Requirements table for Gen 3 x8 support.<br>Added detail to the h2c_dsc_byp_ctl[15:0], and c2h_dsc_byp_ctl[15:0] port descriptions.<br>Added Timing Diagram for Descriptor Bypass mode.<br>Updated description for 11:8 bit index (Channel ID[3:0] field) in the PCIe to DMA Address Field Descriptions table.<br>Added new c_s_axi_supports_narrow_burst parameter to the “Upgrading” appendix. |
| **10/04/2017 Version 4.0** | |
| General Updates | PCIe AXI Bridge mode operation removed from this guide, and moved to *AXI Bridge for PCI Express Gen3 Subsystem Product Guide* (PG194). This document (PG195) only covers DMA mode operation.<br>In the Tandem Configuration section, added instruction and device support information for UltraScale+ devices, and added device support information for UltraScale devices.<br>Updated the “Upgrading” appendix according to port and parameter changes for this version of the core.<br>Added Appendix D, “Using Xilinx Virtual Cable to Debug”. |
| **06/07/2017 Version 3.1** | |
| General Updates | Updated the [NUM_USR_INT-1:0] bit description details.<br>Updated the PCI Extended Tag parameter description.<br>Added a quick start for DMA C2H and H2C transfers in the Product Specification chapter. |

| Section | Revision Summary |
| --- | --- |
| **04/05/2017 Version 3.1** | |
| General Updates | Updated driver support, Windows driver is in pre-production.<br>Updated Identifier Version.<br>Added new GUI parameters: Reset Source, MSI-X Implementation<br>Location, and AXI outstanding transactions.<br>Added Vivado IP integrator-based example design.<br>Updated the Simulation and Descriptor Bypass Mode sections in the Test Bench chapter.<br>Added new parameters and ports to the Upgrading appendix. |
| **02/21/2017 Version 3.0** | |
| General Updates | Updated supported UltraScale+ device speed grades in Minimum Device Requirements table. |
| **11/30/2016 Version 3.0** | |
| General Updates | Updated the core name to reflect two core functional modes: AXI Bridge Subsystem for PCIe (UltraScale+ only), and DMA Subsystem for PCIe (all other supported devices).<br>Organized the Customizing and Generating the Core section (Chapter 4) according to the options available for the two functional modes.<br>Added Debug Options tab in the Vivado IDE to enable debugging options in the core.<br>Updated Identifier Version. |
| **10/12/2016 Version 3.0** | |
| General Updates | Added Artix®-7 and Zynq-7000 SoC device restriction that 7A15T and 7A25T are the only ones not supported. |

| Section | Revision Summary |
|---|---|
| **10/05/2016 Version 3.0** | |
| General Updates | Added additional device family support.<br>Add support for use with the Xilinx Gen2 integrated block for PCIe core.<br>Added performance data to an Answer Record on the web.<br>Updated datapath width and restriction in the Address Alignment and<br>Length Granularity tables in the DMA Operations section.<br>Updated Port Descriptions:<br>• Added support for Parity ports.<br>• Added support for the Configuration Extend ports.<br>Updated Register Space descriptions:<br>• Updated Identifier Version.<br>• Added H2C SGDMA Descriptor Credits (0x8C), C2H SGDMA Descriptor Credits (0x8C0, SGDMA Descriptor Credit Mode Enable (0x20), SG Descriptor Mode Enable Register (0x24), SG Descriptor Mode Enable Register (0x28).<br>Updated Vivado IP catalog description (2016.3):<br>• Updated PCIe: BARs tab, PCIe: Misc tab, and PCIe: DMA tab.<br>• Added Shared Logic tab.<br>Added Basic Vivado Simulation section.<br>Added AXI-MM Example with Descriptor Bypass Mode section.<br>Added additional supported 7 series evaluation board in Debugging appendix). |
| **06/08/2016 Version 2.0** | |
| General Updates | Identifier Version update<br>AXI4-Stream Writeback Disable Control bit documented |
| **04/06/2016 Version 2.0** | |
| Initial Xilinx release. | N/A |

# 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 https://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 https://www.xilinx.com/legal.htm#tos.

### AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

### Copyright

© Copyright 2020 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries. All other trademarks are the property of their respective owners.