Versal ACAP DMA and Bridge Subsystem for PCI Express

Product Guide

Vivado Design Suite

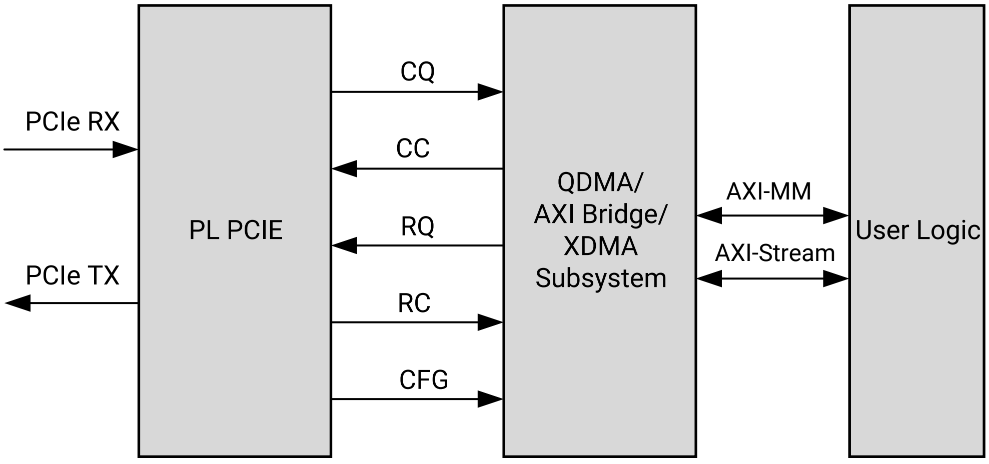
PG344 (v1.0) May 20, 2022

Table of Contents

Introduction to the DMA and Bridge Subsystems

The Versal® ACAP DMA and Bridge Subsystems for PCIe provide a rich set of options for high performance data transfer between a Versal® ACAP and other devices using the widely deployed and industry standard PCI Express system architecture.

Figure 1: **Versal® ACAP DMA and Bridge Subsystem for PCIe**



X22825-041421

These subsystems are built upon the robust and flexible programmable logic integrated block for PCI Express (PL PCIE) as shown in the figure above, and expands the integrated block capabilities through soft IP implemented in the Versal ACAP programmable logic. Three subsystems are available: QDMA Subsystem, AXI Bridge Subsystem, and XDMA Subsystem.

Modular IP Architecture

The subsystems in the Versal® ACAP DMA and Bridge Subsystem for PCIe are packaged in what is called a **modular IP architecture**. Modular IP architecture refers to the programmable logic integrated block for PCIe (PL PCIE) IP and the DMA and Bridge subsystem appearing as two separate IP that are connected in the Vivado IP integrator.

To generate the IP:

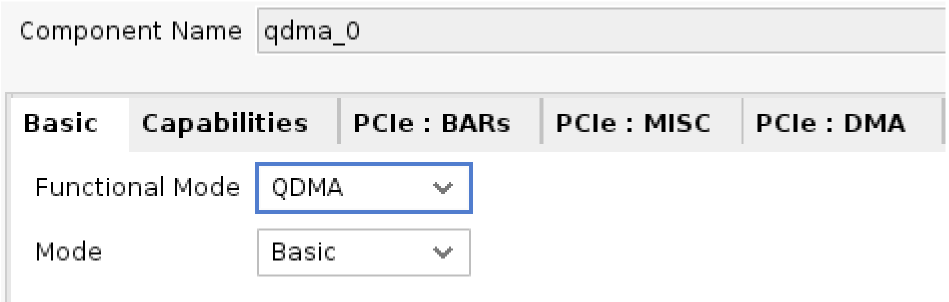
1. In the Vivado IP catalog, locate the subsystem, and add it to your design.
2. Configure the subsystem as required.
3. In the Vivado IP integrator, click **Run Block Automation**. This stitches the PL PCIE and subsystem together.

After these steps, a fully integrated subsystem is available and you can add designs as needed.

QDMA Subsystem

Locating the Subsystem

1. In the Vivado IP catalog, select **Queue DMA Subsystem for PCI Express**.
2. In the Basic tab, and set Functional Mode to **QDMA**.



Component Name qdma_0

Functional Mode QDMA

Mode Basic

Description

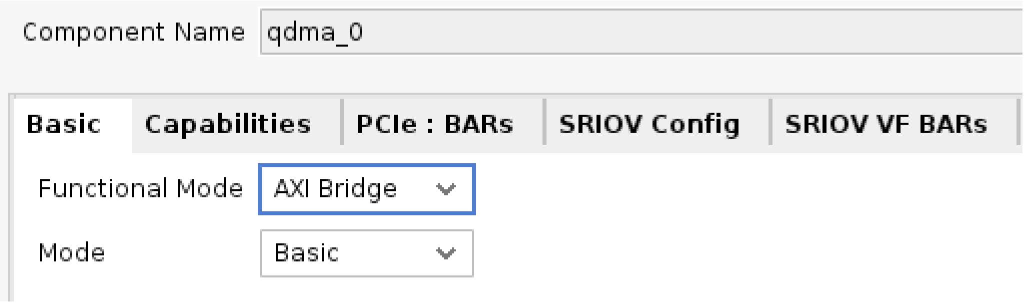
The QDMA subsystem is a queue based, configurable scatter-gather DMA implementation which provides thousands of queues, support for multiple physical/virtual functions with single-root I/O virtualization (SR-IOV), and advanced interrupt support. In this mode the IP provides AXI4-MM and AXI4-Stream user interfaces which may be configured on a per-queue basis. Based on PCIe system architecture conventions, the QDMA is highly suitable for endpoint (EP) use cases and may also be used to construct proprietary system architectures.

AXI Bridge Subsystem

Locating the Subsystem

1. In the Vivado IP catalog, select **Queue DMA Subsystem for PCI Express**.

2. In the Basic tab, and set Functional Mode to **AXI Bridge**.



Component Name qdma_0

Functional Mode AXI Bridge

Mode Basic

Description

A bridge based, configurable translation level between the PCIe system and AXI4-MM internal to the Xilinx device. In this mode the IP translates and forwards PCIe read and write accesses into AXI4-MM interface commands, and conversely translates and forwards AXI4-MM interface commands into PCIe read and write accesses. Based on PCIe system architecture conventions, the AXI Bridge is highly suitable for root port (RP) use cases as well as endpoint (EP) use cases and may also be used to construct proprietary system architectures.

XDMA Subsystem

Locating the Subsystem

In the Vivado IP catalog, select **XDMA Subsystem for PCIe**.

Description

A channel based, configurable scatter-gather DMA implementation which provides four card-to-host (C2H) channels and four host-to-card (H2C) channels with interrupt support. In this mode the IP provides either AXI4-MM or AXI4-Stream user interfaces. Based on PCIe system architecture conventions, the XDMA is highly suitable for endpoint (EP) use cases and may also be used to construct proprietary system architectures.

Features of the DMA and Bridge Subsystem

The subsystem supports these features:

- 64, 128, 256, and 512-bit data path.

- x1, x2, x4, x8, or x16 link widths.
- x1-x16 supported for Gen1-Gen3 line rates.
- x1-x8 supported for Gen4 line rates.
- x1-x4 supported for Gen5 line rates.

QDMA Subsystem

- Support for both AXI4-MM and AXI4-Stream interfaces on a per queue basis
  - 2K queue sets
  - 2K H2C Descriptor rings
  - 2K C2H Descriptor rings
  - 2K C2H Writeback rings
- Supports Polling Mode (Status Descriptor Writeback)
- C2H stream interrupt moderation, CMPT entry coalesce
- Descriptor and DMA customization through user logic
  - Custom descriptor formats
  - Traffic management
- SR-IOV support with access control, FLR, and mailboxes
- Advanced and flexible interrupt support

Related Information

Features
QDMA Subsystem Limitations

AXI Bridge Subsystem

- AXI4-MM access to PCIe address space
- PCIe access to AXI4-MM address space
- Tracks and manages Transaction Layer Packets (TLPs) completion processing
- Detects and indicates error conditions with interrupts
- Supports up to six PCIe 32-bit or three 64-bit PCIe BARs as endpoint (EP)

- Supports up to two PCIe 32-bit or a single PCIe 64-bit BAR as root port (RP)

Related Information

Feature Summary
AXI Bridge Subsystem Limitations

XDMA Subsystem

- Four host-to-card (H2C/Read) data channels
- Four card-to-host (C2H/Write) data channels
- Per channel descriptor bypass for custom descriptor creation
- Configurable user interface (AXI4-MM or AXI4-Stream)
- Advanced and flexible interrupt support

Related Information

Feature Summary
XDMA Subsystem Limitations

Maximum Supported Link Rates and Widths

Maximum Supported Link Rates and Widths with PL PCIE4:

- 2.5 GT/s, 5.0 GT/s, 8.0 GT/s at x1, x2, x4, x8, or x16
- 16 GT/s at x1, x2, x4, or x8

Maximum Supported Link Rates and Widths with PL PCIE5:

- 2.5 GT/s, 5.0 GT/s, 8.0 GT/s at x1, x2, x4, x8, or x16
- 16 GT/s at x8
- 32 GT/s at x4

IP Facts

LogiCORE™ IP Facts Table	
Subsystem Specifics	
Supported Device Family[1]	Versal® ACAP and Versal Premium ACAP
Supported User Interfaces	AXI4 Memory Map, AXI4-Lite, AXI4-Stream
Resources	**QDMA** and **AXI Bridge Subsystems**: Performance and Resource Utilization **XDMA Subsystem**: Performance and Resource Utilization
Provided with Subsystem	
Design Files	Encrypted System Verilog
Example Design	Verilog
Test Bench	Verilog
Constraints File	Xilinx Constraints File (XDC)
Simulation Model	Verilog
Supported S/W Driver[2]	**QDMA Subsystem** and **AXI Bridge Subsystem**: Linux, DPDK, and Windows Drivers **XDMA Subsystem**: Linux, and Windows Drivers
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 75397
All Vivado IP Change Logs	Master Vivado IP Change Logs: 72775
Xilinx Support web page	

Notes:

1. For a complete list of supported devices, see the Vivado® IP catalog.
2. See the "Application Software Development" chapter in the QDMA Subsystem, and XDMA Subsystem sections below.
3. For the supported versions of the tools, see the Xilinx Design Tools: Release Notes Guide.

Overview

Navigating Content by Design Process

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

- **System and Solution Planning:** Identifying the components, performance, I/O, and data transfer requirements at a system level. Includes application mapping for the solution to PS, PL, and AI Engine. Topics in this document that apply to this design process include:
  - QDMA Subsystem
  - AXI Bridge Subsystem
  - XDMA Subsystem
  - IP Facts
- **Embedded Software Development:** Creating the software platform from the hardware platform and developing the application code using the embedded CPU. Also covers XRT and Graph APIs. Topics in this document that apply to this design process include:
  - QDMA Subsystem Register Space
  - AXI Bridge Subsystem Register Space
  - XDMA Subsystem Register Space
- **Host Software Development:** Developing the application code, accelerator development, including library, XRT, and Graph API use. Topics in this document that apply to this design process include:
  - QDMA Subsystem Register Space
  - AXI Bridge Subsystem Register Space
  - XDMA Subsystem Register Space

- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the Vivado® timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:
    - QDMA Subsystem Customizing and Generating the Subsystem
    - AXI Bridge Subsystem Customizing and Generating the Subsystem
    - XDMA Subsystem Customizing and Generating the Subsystem

Unsupported Features

Fast PCI Express Endpoint Enumeration using Tandem Configuration is not supported. This use case addresses the ability to initially load a fully configurable PCI Express protocol solution from a small external ROM, so as to meet the 100 ms enumeration requirement. Support for Tandem Configuration for the PL PCIE block in Versal ACAP is not currently planned.

***Note*:** Any user design requiring fast PCIe enumeration or configuration through PCIe should use the PCIe controllers in the CPM. Not all Versal ACAP contain this particular resource. For details, see *Versal ACAP CPM Mode for PCI Express Product Guide* (PG346).

Standards for the DMA and Bridge Subsystems

Using PL PCIE4

This subsystem adheres to the following standards:

- PCI Express® Base Specification 4.0
- *AMBA AXI4-Stream Protocol Specification* (ARM IHI 0051A)

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

Using PL PCIE5

This subsystem adheres to the following standards:

- PCI Express Base Specification 5.0, and Errata/ECN updates
- *AMBA AXI4-Stream Protocol Specification* (ARM IHI 0051A)

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

Minimum Device Requirements

Table 1: **PL PCIE4 with QDMA, Bridge, or XDMA Soft IP Subsystem Maximum Configurations (Versal Prime, Versal AI Core, Versal AI Edge)**

Speed Grade	-1	-1	-2	-2	-2	-3
Voltage Grade	**L (0.70V)**	**M (0.80V)**	**L (0.70V)**	**M (0.80V)**	**H (0.88V)**	**H (0.88V)**
Gen1 (2.5 GT/s per lane)	x16	x16	x16	x16	x16	x16
Gen2 (5 GT/s per lane)	x16	x16	x16	x16	x16	x16
Gen3 (8 GT/s per lane)	x16	x16	x16	x16	x16	x16
Gen4 (16 GT/s per lane)	x8	x8	x8	x8	x8	x8

Table 2: **PL PCIE5 with QDMA or Bridge Soft IP Subsystem Maximum Configurations (Versal Premium, Versal HBM, Versal AI Core)**

Speed Grade	-1	-1	-2	-2	-2	-3
Voltage Grade	**L (0.70V)**	**M (0.80V)**	**L (0.70V)**	**M (0.80V)**	**H (0.88V)**	**H (0.88V)**
Gen1 (2.5 GT/s per lane)	x16	x16	x16	x16	x16	x16
Gen2 (5 GT/s per lane)	x16	x16	x16	x16	x16	x16
Gen3 (8 GT/s per lane)	x16	x16	x16	x16	x16	x16
Gen4 (16 GT/s per lane)	x8	x8	x8	x8	x8	x8
Gen5 (32 GT/s per lane)[1]	N/A	x4	x4	x4	x4	x4

Notes:

1. Gen5 support is available only in Versal Premium, Versal HBM, and Versal AI Core series.

Licensing and Ordering

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

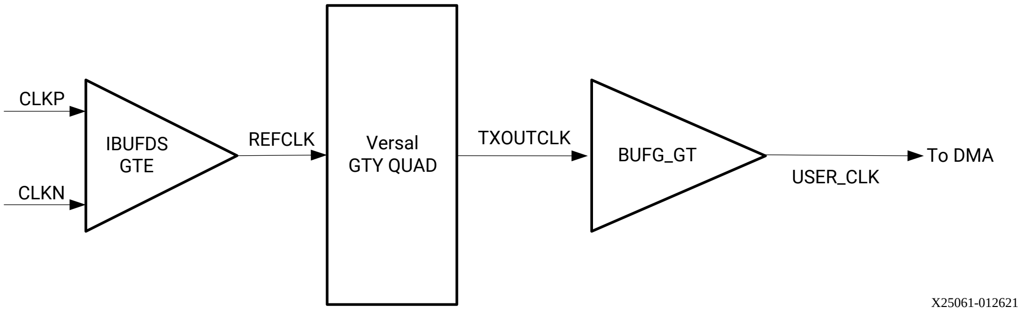
Information about other Xilinx® LogiCORE™ IP modules is available at the Xilinx Intellectual Property page. For information about pricing and availability of other Xilinx LogiCORE IP modules and tools, contact your local Xilinx sales representative.

Designing with the Subsystem

Clocking

The programmable logic integrated block for PCIe (PL PCIE) requires a 100, 125, or 250 MHz reference clock input. The following figure shows the clocking architecture. The `user_clk` clock is available for use in the fabric logic. The `user_clk` clock can be used as the system clock.

Figure 2: **USER_CLK Clocking Architecture**



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

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

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

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

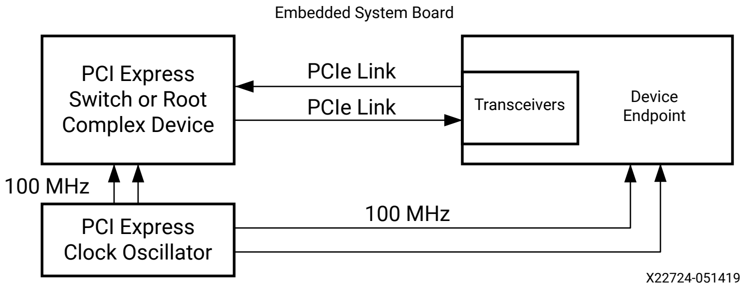
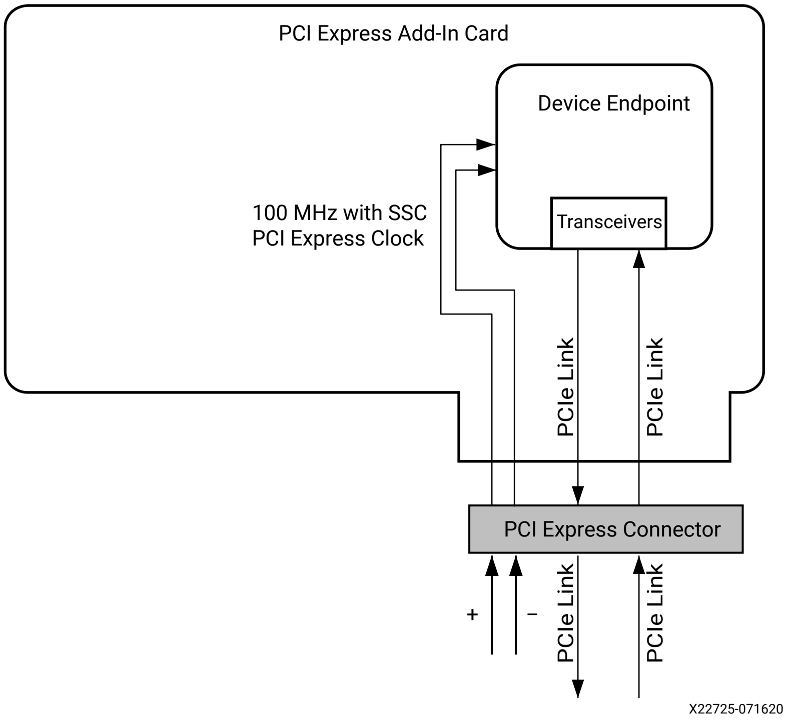


Figure 4: **Open System Add-In Card Using 100 MHz Reference Clock**



Resets

If your board is designed to use the same PCIe edge connectors to operate with CPM and PL PCIE, then we recommend using PS reset using the Control, Interface and Processing System (CIPS) IP core. For more information, see the *Versal ACAP CPM Mode for PCI Express Product Guide* (PG346).

If your board is designed to use PL PCIE in the PCIe edge connectors, then any PL reset pin can be used.

After the reset is released, the core attempts to link train and resumes normal operation.

The DMA subsystem receives the reset signal `user_reset` from the PL PCIE. The DMA subsystem uses `user_reset` as main reset for all its logic. When the PL PCIE goes through a reset or there is a link down, it issues a `user_reset` to the DMA subsystem. After the PCIe link is up, the `user_reset` is released for the DMA subsystem.

QDMA Subsystem for PL PCIE4 and PL PCIE5

Overview

The Queue-based Direct Memory Access (QDMA) subsystem is a PCI Express® (PCIe®) based DMA engine that is optimized for both high bandwidth and high packet count data transfers. The QDMA is composed of the Versal® Integrated Block for PCI Express, and an extensive DMA and bridge infrastructure that enables the ultimate in performance and flexibility.

The QDMA offers a wide range of setup and use options, many selectable on a per-queue basis, such as memory-mapped DMA or stream DMA, interrupt mode and polling. The subsystem provides many options for customizing the descriptor and DMA through user logic to provide complex traffic management capabilities.

The primary mechanism to transfer data using the QDMA is for the QDMA engine to operate on instructions (descriptors) provided by the host operating system. Using the descriptors, the QDMA can move data in both the Host to Card (H2C) direction, or the Card to Host (C2H) direction. You can select on a per-queue basis whether DMA traffic goes to an AXI4 memory map (MM) interface or to an AXI4-Stream interface. In addition, the QDMA has the option to implement both an AXI4 MM Master port and an AXI4 MM Slave port, allowing PCIe traffic to bypass the DMA engine completely.

The main difference between QDMA and other DMA offerings is the concept of queues. The idea of queues is derived from the “queue set” concepts of Remote Direct Memory Access (RDMA) from high performance computing (HPC) interconnects. These queues can be individually configured by interface type, and they function in many different modes. Based on how the DMA descriptors are loaded for a single queue, each queue provides a very low overhead option for setup and continuous update functionality. By assigning queues as resources to multiple PCIe Physical Functions (PFs) and Virtual Functions (VFs), a single QDMA core and PCI Express interface can be used across a wide variety of multifunction and virtualized application spaces.

The QDMA can be used and exercised with a Xilinx® provided QDMA reference driver, and then built out to meet a variety of application spaces.

Related Information

Port Descriptions

Modular IP Architecture

The QDMA Subsystem is packaged in what is called a **modular IP architecture**. Modular IP architecture refers to the programmable logic integrated block for PCIe (PL PCIE) IP and the QDMA subsystem appearing as two separate IP that are connected in the Vivado IP integrator.

To generate the IP:

1. In the Vivado IP catalog, locate the **QDMA Subsystem for PCI Express**, and add it to your design.
2. Configure the subsystem as required
3. In the Vivado IP integrator, click **Run Block Automation**. This stitches the PL PCIE and QDMA together.

After these steps, a fully integrated PL PCIE and QDMA is available, and you can add designs as needed.

Related Information

Customizing and Generating the Subsystem

Features

- The PCIe Integrated Block is supported in Versal® ACAP.
- Supports 64, 128, 256, and 512-bit data path.
- Supports x1, x2, x4, x8, or x16 link widths.
- Supports Gen1, Gen2, and Gen3 link speeds. Gen4 for PCIE4C block.
- Support for both the AXI4 Memory Mapped and AXI4-Stream interfaces per queue.
- 2048 queue sets
  - 2048 H2C descriptor rings.
  - 2048 C2H descriptor rings.
  - 2048 C2H Completion (CMPT) rings.
- Supports Polling Mode (Status Descriptor Write Back) and Interrupt Mode.

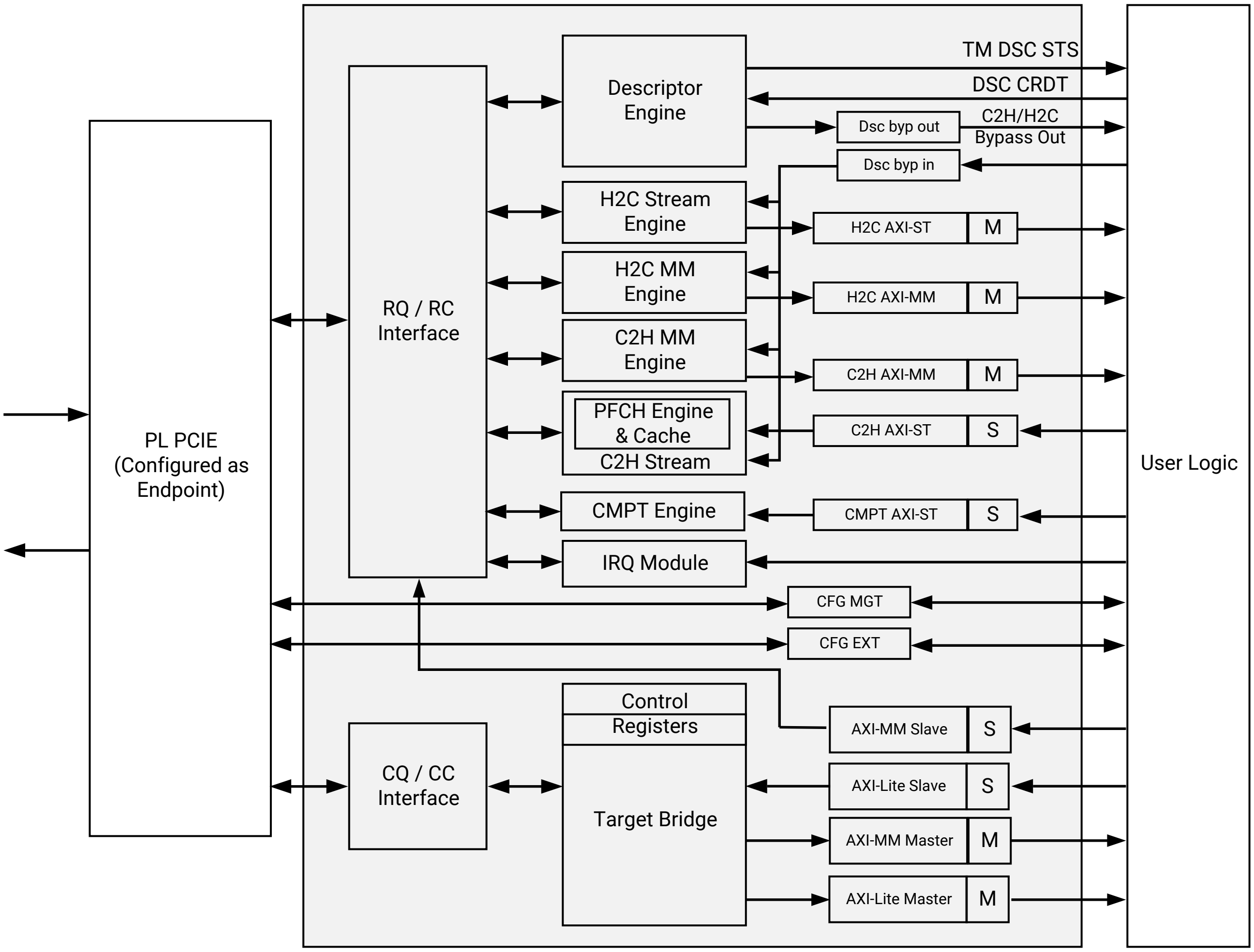
- Interrupts
  - 2048 MSI-X vectors.
  - Up to 8 MSI-X per function.

    Note**:** It is possible to assign more vectors per function. For more information, see AR 72352.
  - Interrupt aggregation.
- C2H Stream interrupt moderation.
- C2H Stream Completion queue entry coalescence.
- Descriptor and DMA customization through user logic
  - Allows custom descriptor format.
  - Traffic Management.
- Supports SR-IOV with up to 4 Physical Functions (PF) and 252 Virtual Functions (VF)
  - Thin hypervisor model.
  - QID virtualization.
  - Allows only privileged/Physical functions to program contexts and registers.
  - Function level reset (FLR) support.
  - Mailbox.
- Rich programmability on a per queue basis, such as AXI4 Memory Mapped versus AXI4-Stream interfaces.

QDMA Architecture

The following figure shows the block diagram of the QDMA.

Figure 5: **QDMA Architecture**



X22823-041421

DMA Engines

Descriptor Engine

The Host to Card (H2C) and Card to Host (C2H) descriptors are fetched by the Descriptor Engine in one of two modes: Internal mode, and Descriptor bypass mode. The descriptor engine maintains per queue contexts where it tracks software (SW) producer index pointer (PIDX), consumer index pointer (CIDX), base address of the queue (BADDR), and queue configurations for each queue. The descriptor engine uses a round robin algorithm for fetching the descriptors. The descriptor engine has separate buffers for H2C and C2H queues, and ensures it never fetches more descriptors than available space. The descriptor engine will have only one DMA read outstanding per queue at a time and can read as many descriptors as can fit in a MRRS. The descriptor engine is responsible for reordering the out of order completions and ensures that descriptors for queues are always in order.

The descriptor bypass can be enabled on a per-queue basis and the fetched descriptors, after buffering, are sent to the respective bypass output interface instead of directly to the H2C or C2H engine. In internal mode, based on the context settings the descriptors are sent to delete per H2C memory mapped (MM), C2H MM, H2C Stream, or C2H Stream engines.

The descriptor engine is also responsible for generating the status descriptor for the completion of the DMA operations. With the exception of C2H Stream mode, all modes use this mechanism to convey completion of each DMA operation so that software can reclaim descriptors and free up any associated buffers. This is indicated by the CIDX field of the status descriptor.

✓ **RECOMMENDED:** *If a queue is associated with interrupt aggregation, Xilinx recommends that the status descriptor be turned off, and instead the DMA status be received from the interrupt aggregation ring.*

To put a limit on the number of fetched descriptors (for example, to limit the amount of buffering required to store the descriptor), it is possible to turn-on and throttle credit on a per-queue basis. In this mode, the descriptor engine fetches the descriptors up to available credit, and the total number of descriptors fetched per queue is limited to the credit provided. The user logic can return the credit through the `dsc_crdt` interface. The credit is in the granularity of the size of the descriptor.

To help a user-developed traffic manager prioritize the workload, the available descriptor to be fetched (incremental PIDX value) of the PIDX update is sent to the user logic on the `tm_dsc_sts` interface. Using this interface it is possible to implement a design that can prioritize and optimize the descriptor storage.

H2C MM Engine

The H2C MM Engine moves data from the host memory to card memory through the H2C AXI-MM interface. The engine generates reads on PCIe, splitting descriptors into multiple read requests based on the MRRS and the requirement that PCIe reads do not cross 4 KB boundaries. Once completion data for a read request is received, an AXI write is generated on the H2C AXI-MM interface. For source and destination addresses that are not aligned, the hardware will shift the data and split writes on AXI-MM to prevent 4 KB boundary crossing. Each completed descriptor is checked to determine whether a writeback and/or interrupt is required.

For Internal mode, the descriptor engine delivers memory mapped descriptors straight to the H2C MM engine. The user logic can also inject the descriptor into the H2C descriptor bypass interface to move data from host to card memory. This gives the ability to do interesting things such as mixing control and DMA commands in the same queue. Control information can be sent to a control processor indicating the completion of DMA operation.

C2H MM Engine

The C2H MM Engine moves data from card memory to host memory through the C2H AXI-MM interface. The engine generates AXI reads on the C2H AXI-MM bus, splitting descriptors into multiple requests based on 4 KB boundaries. Once completion data for the read request is received on the AXI4 interface, a PCIe write is generated using the data from the AXI read as the contents of the write. For source and destination addresses that are not aligned, the hardware will shift the data and split writes on PCIe to obey Maximum Payload Size (MPS) and prevent 4 KB boundary crossings. Each completed descriptor is checked to determine whether a writeback and/or interrupt is required.

For Internal mode, the descriptor engine delivers memory mapped descriptors straight to the C2H MM engine. As with H2C MM Engine, the user logic can also inject the descriptor into the C2H descriptor bypass interface to move data from card to host memory.

For multi-function configuration support, the PCIe function number information will be provided in the `aruser` bits of the AXI-MM interface bus to help virtualization of card memory by the user logic. A parity bus, separate from the data and user bus, is also provided for end-to-end parity support.

H2C Stream Engine

The H2C stream engine moves data from the host to the H2C Stream interface. For internal mode, descriptors are delivered straight to the H2C stream engine; for a queue in bypass mode, the descriptors can be reformatted and fed to the bypass input interface. The engine is responsible for breaking up DMA reads to MRRS size, guaranteeing the space for completions, and also makes sure completions are reordered to ensure H2C stream data is delivered to user logic in-order.

The engine has sufficient buffering for up to 256 descriptor reads and up to 32 KB of data. DMA fetches the data and aligns to the first byte to transfer on the AXI4 interface side. This allows every descriptor to have random offset and random length. The total length of all descriptors put together must be less than 64 KB.

For internal mode queues, each descriptor defines a single AXI4-Stream packet to be transferred to the H2C AXI-ST interface. A packet with multiple descriptors straddling is not allowed due to the lack of per queue storage. However, packets with multiple descriptors straddling can be implemented using the descriptor bypass mode. In this mode, the H2C DMA engine can be initiated when the user logic has enough descriptors to form a packet. The DMA engine is initiated by delivering the multiple descriptors straddled packet along with other H2C ST packet descriptors through the bypass interface, making sure they are not interleaved. Also, through the bypass interface, the user logic can control the generation of the status descriptor.

C2H Stream Engine

The C2H streaming engine is responsible for receiving data from the user logic and writing to the Host memory address provided by the C2H descriptor for a given Queue.

The C2H engine has two major blocks to accomplish C2H streaming DMA, Descriptor Prefetch Cache (PFCH), and the C2H-ST DMA Write Engine. The PFCH has per queue context to enhance the performance of its function and the software that is expected to program it.

PFCH cache has three main modes, on a per queue basis, called Simple Bypass Mode, Internal Cache Mode, and Cached Bypass Mode.

- In Simple Bypass Mode, the engine does not track anything for the queue, and the user logic can define its own method to receive descriptors. The user logic is then responsible for delivering the packet and associated descriptor through the simple bypass interface. The ordering of the descriptors fetched by a queue in the bypass interface and the C2H stream interface must be maintained across all queues in bypass mode.
- In Internal Cache Mode and Cached Bypass Mode, the PFCH module offers storage for up to 512 descriptors and these descriptors can be used by up to 64 different queues. In this mode, the engine controls the descriptors to be fetched by managing the C2H descriptor queue credit on demand based on received packets in the pipeline. Pre-fetch mode can be enabled on a per queue basis, and when enabled, causes the descriptors to be opportunistically pre-fetched so that descriptors are available before the packet data is available. The status can be found in prefetch context. This significantly reduces the latency by allowing packet data to be transferred to the PCIe integrated block almost immediately, instead of having to wait for the relevant descriptor to be fetched. The size of the data buffer is fixed for a queue (PFCH context) and the engine can scatter the packet across as many as seven descriptors. In cached bypass mode descriptor is bypassed to user logic for further processing, such as address translation, and sent back on the bypass in interface. This mode does not assume any ordering descriptor and C2H stream packet interface, and the pre-fetch engine can match the packet and descriptors. When pre-fetch mode is enabled, do not give credits to IP. The pre-fetch engine takes care of credit management.

Completion Engine

The Completion (CMPT) Engine is used to write to the completion queues. Although the Completion Engine can be used with an AXI-MM interface and Stream DMA engines, the C2H Stream DMA engine is designed to work closely with the Completion Engine. The Completion Engine can also be used to pass immediate data to the Completion Ring. The Completion Engine can be used to write Completions of up to 64B in the Completion ring. When used with a DMA engine, the completion is used by the driver to determine how many bytes of data were transferred with every packet. This allows the driver to reclaim the descriptors.

The Completion Engine maintains the Completion Context. This context is programmed by the Driver and is maintained on a per-queue basis. The Completion Context stores information like the base address of the Completion Ring, PIDX, CIDX and a number of aspects of the Completion Engine, which can be controlled by setting the fields of the Completion Context.

The engine also can be configured on a per-queue basis to generate an interrupt or a completion status update, or both, based on the needs of the software. If the interrupts for multiple queues are aggregated into the interrupt aggregation ring, the status descriptor information is available in the interrupt aggregation ring as well.

The CMPT Engine has a cache of up to 64 entries to coalesce the multiple smaller CMPT writes into 64B writes to improve the PCIe efficiency. At any time, completions can be simultaneously coalesced for up to 64 queues. Beyond this, any additional queue that needs to write a CMPT entry will cause the eviction of the least recently used queue from the cache. The depth of the cache used for this purpose is configurable with possible values of 8, 16, 32, and 64.

Bridge Interfaces

AXI Memory Mapped Bridge Master Interface

The AXI MM Bridge Master interface is used for high bandwidth access to AXI Memory Mapped space from the host. The interface supports up to 32 outstanding AXI reads and writes. One or more PCIe BAR of any physical function (PF) or virtual function (VF) can be mapped to the AXI-MM bridge master interface. This selection must be made prior to design compilation. The function ID, BAR ID, VF group, and VF group offset will be made available as part of `aruser` and `awuser` of the AXI-MM interface allowing the user logic to identify the source of each memory access. The `m_axib_awuser/m_axib_aruser[54:0]` user bits mapping is listed in AXI Bridge Master Ports.

Virtual function group (VFG) refers to the VF group number. It is equivalent to the PF number associated with the corresponding VF. VFG_OFFSET refers to the VF number with respect to a particular PF. Note that this is not the FIRST_VF_OFFSET of each PF.

For example, if both PF0 and PF1 have 8 VFs, FIRST_VF_OFFSET for PF0 and PF1 is 4 and 11. Below is the mapping for VFG and VFG_OFFSET.

Table 3: **AXI-MM Interface Virtual Function Group**

Function Number	PF Number	VFG	VFG_OFFSET
0	0	0	0
1	1	0	0
4	0	0	0 (Because FIRST_VF_OFFSET for PF0 is 4, the first VF of PF0 starts at FN_NUM=4 and VFG_OFFSET=0 indicates this is the first VF for PF0)
5	0	0	1 (VFG_OFFSET=1 indicates this is the second VF for PF0)
...	...	...	...
12	1	1	0 (VFG=1 indicates this VF is associated with PF1)
13	1	1	1

Each host initiated access can be uniquely mapped to the 64 bit AXI address space through the PCIe to AXI BAR translation.

Since all functions share the same AXI Master address space, a mechanism is needed to map requests from different functions to a distinct address space on the AXI master side. An example provided below shows how PCIe to AXI translation vector is used. Note that all VFs belonging to the same PF share the same PCIe to AXI translation vector. Therefore, the AXI address space of each VF is concatenated together. Use VFG_OFFSET to calculate the actual starting address of AXI for a particular VF.

To summarize, `m_axib_awaddr` is determined as:

- For PF, `m_axib_awaddr` = `pcie2axi_vec` + `axib_offset`.
- For VF, `m_axib_awaddr` = `pcie2axi_vec` + (VFG_OFFSET + 1)*`vf_bar_size` + `axib_offset`.

Where `pcie2axi_vec` is PCIe to AXI BAR translation (that can be set when the IP core is configured from the Vivado IP Catalog).

And `axib_offset` is the address offset in the requested target space.

Related Information

AXI Bridge Master Ports

AXI4-Lite Bridge Master Interface

One or more PCIe BAR of any physical function (PF) or virtual function (VF) can be mapped to the AXI4-Lite master interface. This selection must be done at the point of configuring the IP. The function ID, BAR ID (BAR hit), VF group, and VF group offset will be made available as part of `aruser` and `awuser` of the AXI4-Lite interface to help the user logic identify the source of memory access.

The `m_axil_awuser/m_axil_aruser[54:0]` user bits mapping is listed in AXI4-Lite Master Ports.

Virtual function group (VFG) refers to the VF group number. It is equivalent to the PF number associated with the corresponding VF. VFG_OFFSET refer to the VF number with respect to a particular PF. Note that this is not the FIRST_VF_OFFSET of each PF.

For example, if both PF0 and PF1 has 8 VFs, and FIRST_VF_OFFSET for PF0 and PF1 is 4 and 11 and below is the mapping for VFG and VFG_OFFSET.

Table 4: **AXI4-Lite Interface VFG**

Function Number	PF Number	VFG	VFG_OFFSET
0	0	0	0
1	1	0	0
4	0	0	0 (Because FIRST_VF_OFFSET for PF0 is 4, the first VF of PF0 starts at FN_NUM=4 and VFG_OSSET=0 indicates this is the first VF for PF0)
5	0	0	1 (VFG_OSSET=1 indicates this is the second VF for PF0)
...	...	...	...
12	1	1	0 (VFG=1 indicates this VF is associated with PF1)
13	1	1	1

Each host initiated access can be uniquely mapped to the 64 bit AXI address space through the PCIe to AXI BAR translation.

Because all functions shares the same AXI4 master address space, a mechanism is needed to map requests from different functions to a distinct address space on the AXI master side. This below shows how PCIe to AXI translation vector is used. Note that all VFs belonging to the same PF shares the same PCIe to AXI translation vector. Therefore, the AXI address space of each VF is concatenated together. Use VFG_OFFSET to calculate the actual starting address of AXI for a particular VF.

To summarize, m_axil_awaddr is determined as:

- For PF, `m_axil_awaddr` = `pcie2axi_vec` + `axil_offset`.
- For VF, `m_axil_awaddr` = `pcie2axi_vec` + (VFG_OFFSET + 1)*`vf_bar_size` + `axil_offset`

Where `pcie2axi_vec` is PCIe to AXI BAR translation (that can be set during IP configuration.).

And `axib_offset` is the address offset in the requested target space.

Each host initiated access can be uniquely mapped to the 64 bit AXI address space. One outstanding read and one outstanding write are supported on this interface.

Expansion ROM BAR can also be mapped to AXI4-Lite interface at the IP configuration time.

Related Information

AXI4-Lite Master Ports

PCIe to AXI BARs

For each physical function, the PCIe configuration space consists of a set of six 32-bit memory BARs and one 32-bit Expansion ROM BAR. When SR-IOV is enabled, an additional six 32-bit BARs are enabled for each Virtual Function. These BARs provide address translation to the AXI4 memory mapped space capability, interface routing, and AXI4 request attribute configuration. Any pairs of BARs can be configured as a single 64-bit BAR. Each BAR can be configured to route its requests to the QDMA register space, the Bridge AXI4-Lite master interface, or the AXI MM bridge master interface.

Request Memory Type

The memory type can be set for each PCIe BAR through attributes `attr_dma_pciebar2axibar_*_cache_pf*`.

- AxCache[0] is set to 1 for modifiable, and 0 for non-modifiable.
- AxCache[1] is set to 1 for cacheable, and 0 for non-cacheable.

AXI Memory Mapped Bridge Slave Interface

The AXI-MM Bridge Slave interface is used for high bandwidth memory transfers between the user logic and the Host. AXI to PCIe translation is supported through the AXI to PCIe BARs. The interface will split requests as necessary to obey PCIe MPS and 4 KB boundary crossing requirements. Up to 32 outstanding read and write requests are supported.

AXI4-Lite Bridge Slave CSR Interface

The AXI4-Lite slave interface is used to access the AXI Bridge and QDMA internal registers. Address bit [15] indicates if the access is for QDMA registers or AXI Bridge registers.

- When `s_axil_csr_awaddr[15]` = `1'b1`, the write access is for QDMA CSR registers.
- When `s_axil_csr_awaddr[15]` = `1'b0`, the write access is for Bridge registers (When accessing Bridge Registers, access from address `0x000` to `0xDFF` will be redirected to PCIe core configuration space access and from address `0xE00` will be directed towards Bridge registers).
- When `s_axil_csr_araddr[15]` = `1'b1`, the read access is for QDMA CSR registers.
- When `s_axil_csr_araddr[15]` = `1'b0`, the read access is for Bridge registers. When accessing Bridge Registers, access from address `0x000` to `0xDFF` will be redirected to PCIe core configuration space access and from address `0xE00` will be directed towards Bridge registers.

The QDMA registers are virtualized for VFs and PFs. For example, VFs and PFs can access different parts of the address space, and each has access to its own queues. To accommodate the function specific accesses, the user logic can provide function ID on `s_axil_awuser[7:0]` for write access and `s_axil_aruser[7:0]` read access, which gives the QDMA proper internal register access. One outstanding read request and one outstanding write request are supported on the AXI4-Lite slave interface.

The AXI4-Lite slave interface is also used to generate Vendor defined messages (VDM) using the Bridge registers.

Related Information

VDM

AXI to PCIe BARs

In the Bridge Slave interface, there is one BARs which can be configured as 32 bits or 64 bits. This BAR provide address translation from AXI address space to PCIe address space. The address translation is configured through BDF table programming. Refer to Slave Bride section for BDF programming.

Related Information

Slave Bridge

Interrupt Module

The IRQ module aggregates interrupts from various sources. The interrupt sources are queue-based interrupts, user interrupts and error interrupts.

Queue-based interrupts and user interrupts are allowed on PFs and VFs, but error interrupts are allowed only on PFs. If the SR-IOV is not enabled, each PF has the choice of MSI-X or Legacy Interrupts. With SR-IOV enabled, only MSI-X interrupts are supported across all functions.

MSI-X interrupt is enabled by default. Host system (Root Complex) will enable one or all of the interrupt types supported in hardware. If MSI-X is enabled, it takes precedence.

Up to eight interrupts per function are available. To allow many queues on a given function and each to have interrupts, the QDMA offers a novel way of aggregating interrupts from multiple queues to a single interrupt vector. In this way, all 2048 queues could in principle be mapped to a single interrupt vector. QDMA offers 256 interrupt aggregation rings that can be flexibly allocated among the 256 available functions.

PCIe Block Interface

PCIe CQ/CC

The PCIe Completer Request (CQ)/Completer Completion (CC) modules receive and process TLP requests from the remote PCIe agent. This interface to the Versal Integrated Block for PCIe IP operates in address aligned mode. The module uses the BAR information from the Integrated Block for PCIe IP to determine where the request should be forwarded. The possible destinations for these requests are:

- DMA configuration module
- AXI4 MM Bridge Master interface
- the AXI4-Lite Bridge Master interface

Non-posted requests are expected to receive completions from the destination, which are forwarded to the remote PCIe agent. For further details, see the *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343).

PCIe RQ/RC

The PCIe Requester Request (RQ)/Requester Completion (RC) interface generates PCIeTLPs on the RQ bus and processes PCIe Completion TLPs from the RC bus. This interface to the Versal Integrated Block for PCIe® core operates in DWord aligned mode. With a 512-bit interface, straddling will be enabled. While straddling is supported, all combinations of RQ straddled transactions may not be implemented. For further details, see the *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343).

PCIe Configuration

Several factors can throttle outgoing non-posted transactions. Outgoing non-posted transactions are throttled based on flow control information from the PCIe® integrated block to prevent head of line blocking of posted requests. The DMA will meter non-posted transactions based on the PCIe Receive FIFO space.

General Design of Queues

The multi-queue DMA engine of the QDMA uses RDMA model queue pairs to allow RNIC implementation in the user logic. Each queue set consists of Host to Card (H2C), Card to Host (C2H), and a C2H Stream Completion (CMPT). The elements of each queue are descriptors.

H2C and C2H are always written by the driver/software; hardware always reads from these queues. H2C carries the descriptors for the DMA read operations from Host. C2H carries the descriptors for the DMA write operations to the Host.

In internal mode, H2C descriptors carry address and length information and are called gather descriptors. They support 32 bits of metadata that can be passed from software to hardware along with every descriptor. The descriptor can be memory mapped (where it carries host address, card address, and length of DMA transfer) or streaming (only host address, and length of DMA transfer) based on context settings. Through descriptor bypass, an arbitrary descriptor format can be defined, where software can pass immediate data and/or additional metadata along with packet.

C2H queue memory mapped descriptors include the card address, the host address and the length. In streaming internal cached mode, descriptors carry only the host address. The buffer size of the descriptor, which is programmed by the driver, is expected to be of fixed size for the whole queue. Actual data transferred associated with each descriptor does not need to be the full length of the buffer size.

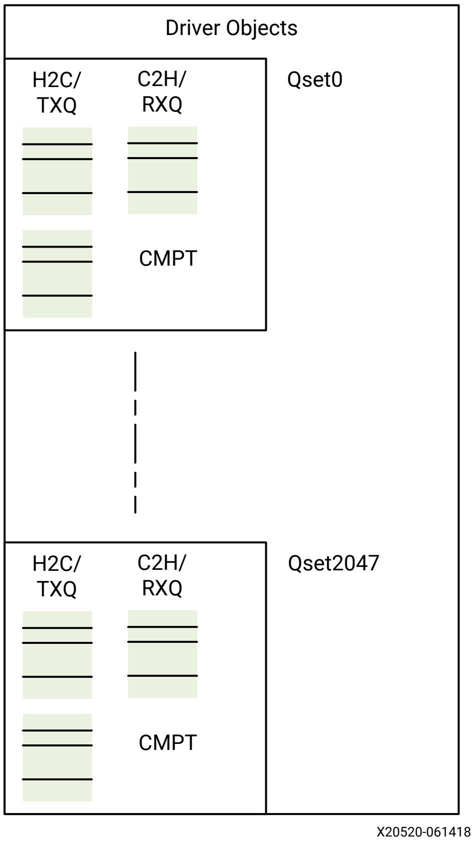
The software advertises valid descriptors for H2C and C2H queues by writing its producer index (PIDX) to the hardware. The status descriptor is the last entry of the descriptor ring, except for a C2H stream ring. The status descriptor carries the consumer index (CIDX) of the hardware so that the driver knows when to reclaim the descriptor and deallocate the buffers in the host.

For the C2H stream mode, C2H descriptors will be reclaimed based on the CMPT queue entry. Typically, this carries one entry per C2H packet, indicating one or more C2H descriptors is consumed. The CMPT queue entry carries enough information for software to claim all the descriptors consumed. Through external logic, this can be extended to carry other kinds of completions or information to the host.

CMPT entries written by the hardware to the ring can be detected by the driver using either the color bit in the descriptor or the status descriptor at the end of the CMPT ring. Each CMPT entry can carry metadata for a C2H stream packet and can also serve as a custom completion or immediate notification for the user application.

The base address of all ring buffers (H2C, C2H, and CMPT) should be aligned to a 4 KB address.

Figure 6: **Queue Ring Architecture**



The software can program 16 different ring sizes. The ring size for each queue can be selected from context programing. The last queue entry is the descriptor status, and the number of allowable entries is (queue size -1).

For example, if queue size is 8, which contains the entry index 0 to 7, the last entry (index 7) is reserved for status. This index should never be used for PIDX update, and PIDX update should never be equal to CIDX. For this case, if CIDX is 0, the maximum PIDX update would be 6.

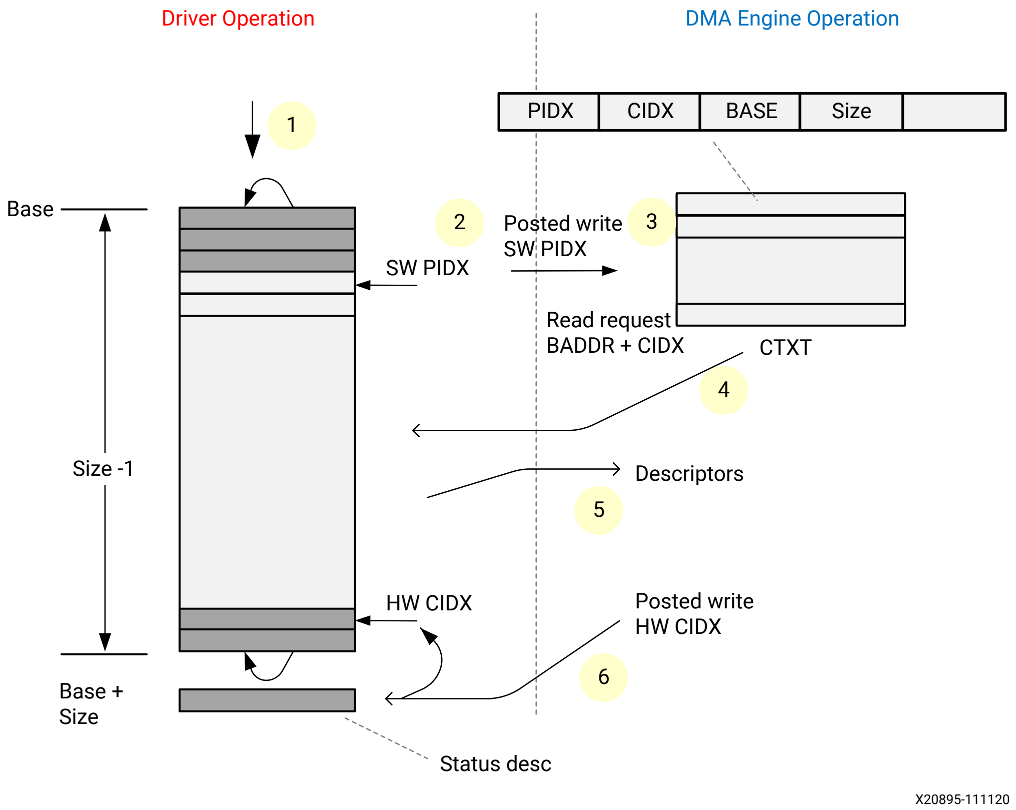
In the example above, if traffic has already started and the CIDX is 4, the maximum PIDX update is 3.

H2C and C2H Queues

H2C/C2H queues are rings located in host memory. For both type of queues, the producer is software and consumer is the descriptor engine. The software maintains producer index (PIDX) and a copy of hardware consumer index (HW CIDX) to avoid overwriting unread descriptors. The descriptor engine also maintains consumer index (CIDX) and a copy of SW PIDX, which is to make sure the engine does not read unwritten descriptors. The last entry in the queue is dedicated for the status descriptor where the engine writes the HW CIDX and other status.

The engine maintains a total of 2048 H2C and 2048 C2H contexts in local memory. The context stores properties of the queue, such as base address (BADDR), SW PIDX, CIDX, and depth of the queue.

Figure 7: **Simple H2C and C2H Queue**



The figure above shows the H2C and C2H fetch operation.

1. For H2C, the driver writes payload into host buffer, forms the H2C descriptor with the payload buffer information and puts it into H2C queue at the PIDX location. For C2H, the driver forms the descriptor with available buffer space reserved to receive the packet write from the DMA.
2. The driver sends the posted write to PIDX register in the descriptor engine for the associated Queue ID (QID) with its current PIDX value.
3. Upon reception of the PIDX update, the engine calculates the absolute QID of the pointer update based on address offset and function ID. Using the QID, the engine will fetch the context for the absolute QID from the memory associated with the QDMA.
4. The engine determines the number of descriptors that are allowed to be fetched based on the context. The engine calculates the descriptor address using the base address (BADDR), CIDX, and descriptor size, and the engine issues the DMA read request.

5. After the descriptor engine receives the read completion from the host memory, the descriptor engine delivers them to the H2C Engine or C2H Engine in internal mode. In case of bypass, the descriptors are sent out to the associated descriptor bypass output interface.
6. For memory mapped or H2C stream queues programmed as internal mode, after the fetched descriptor is completely processed, the engine writes the CIDX value to the status descriptor. For queues programmed as bypass mode, user logic controls the write back through bypass in interface. The status descriptor could be moderated based on context settings. C2H stream queues always use the CMPT ring for the completions.

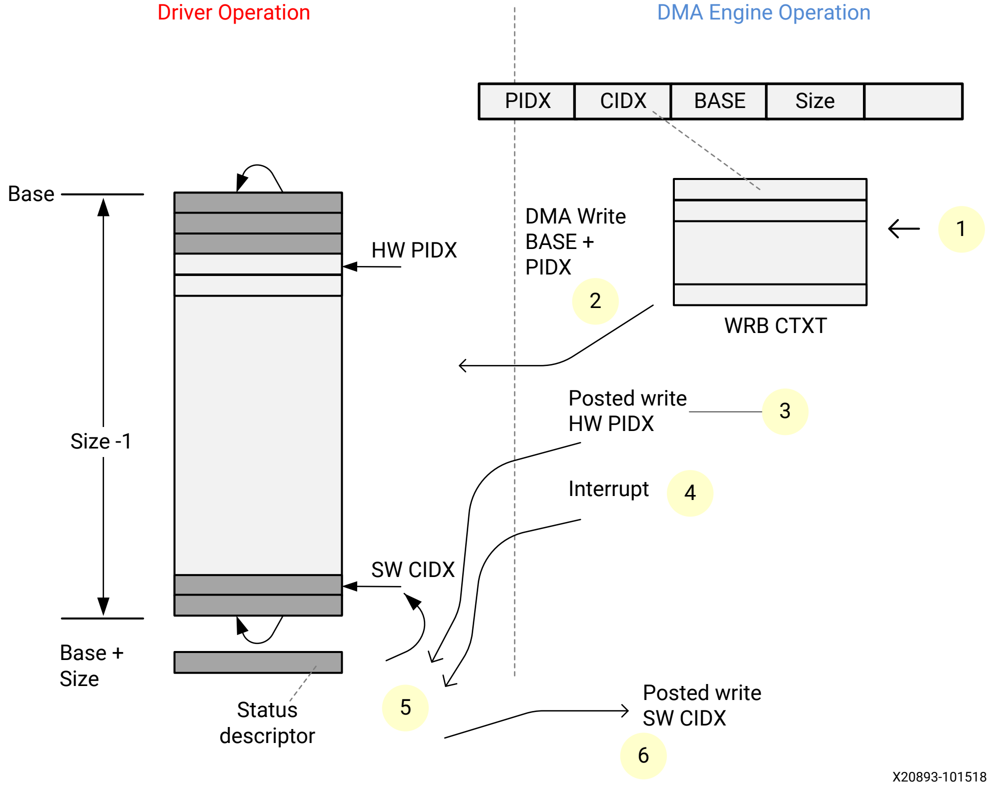
For C2H, the fetch operation is implicit through the CMPT ring.

Completion Queue

The Completion (CMPT) queue is a ring located in host memory. The consumer is software, and the producer is the CMPT engine. The software maintains the consumer index (CIDX) and a copy of hardware producer index (HW PIDX) to avoid reading unwritten completions. The CMPT engine also maintains PIDX and a copy of software consumer index (SW CIDX) to make sure that the engine does not overwrite unread completions. The last entry in the queue is dedicated for the status descriptor which is where the engine writes the hardware producer index (HW PIDX) and other status.

The engine maintains a total of 2048 CMPT contexts in local memory. The context stores properties of the queue, such as base address, SW CIDX, PIDX, and depth of the queue.

Figure 8: **Simple Completion Queue Flow**



C2H stream is expected to use the CMPT queue for completions to host, but it can also be used for other types of completions or for sending messages to the driver. The message through the CMPT is guaranteed to not bypass the corresponding C2H stream packet DMA.

The simple flow of DMA CMPT queue operation with respect to the numbering above follows:

1. The CMPT engine receives the completion message through the CMPT interface, but the QID for the completion message comes from the C2H stream interface. The engine reads the QID index of CMPT context RAM.
2. The DMA writes the CMPT entry to address BASE+PIDX.
3. If all conditions are met, optionally writes PIDX to the status descriptor of the CMPT queue with color bit.
4. If interrupt mode is enabled, the CMPT engine generates the interrupt event message to the interrupt module.
5. The driver can be in polling or interrupt mode. Either way, the driver identifies the new CMPT entry either by matching the color bit or by comparing the PIDX value in the status descriptor against its current software CIDX value.

6. The driver updates CIDX for that queue. This allows the hardware to reuse the descriptors again. After the software finishes processing the CMPT, that is, before it stops polling or leaving the interrupt handler, the driver issues a write to CIDX update register for the associated queue.

SR-IOV Support

The QDMA provides an optional feature to support Single Root I/O Virtualization (SR-IOV). The PCI-SIG® Single Root I/O Virtualization and Sharing (SR-IOV) specification (available from *PCI-SIG Specifications* (www.pcisig.com/specifications) standardizes the method for bypassing the VMM involvement in datapath transactions and allows a single endpoint to appear as multiple separate endpoints. SR-IOV classifies the functions as:

- **Physical Functions (PF)**: Full featured PCIe® functions which include SR-IOV capabilities among others.
- **Virtual Functions (VF)**: PCIe functions featuring configuration space with Base Address Registers (BARs) but lacking the full configuration resources and controlled by the PF configuration. The main role of the VF is data transfer.

Apart from PCIe defined configuration space, QDMA Subsystem for PCI Express virtualizes data path operations, such as pointer updates for queues, and interrupts. The rest of the management and configuration functionality is deferred to the physical function driver. The Drivers that do not have sufficient privilege must communicate with the privileged Driver through the mailbox interface which is provided in part of the QDMA Subsystem for PCI Express.

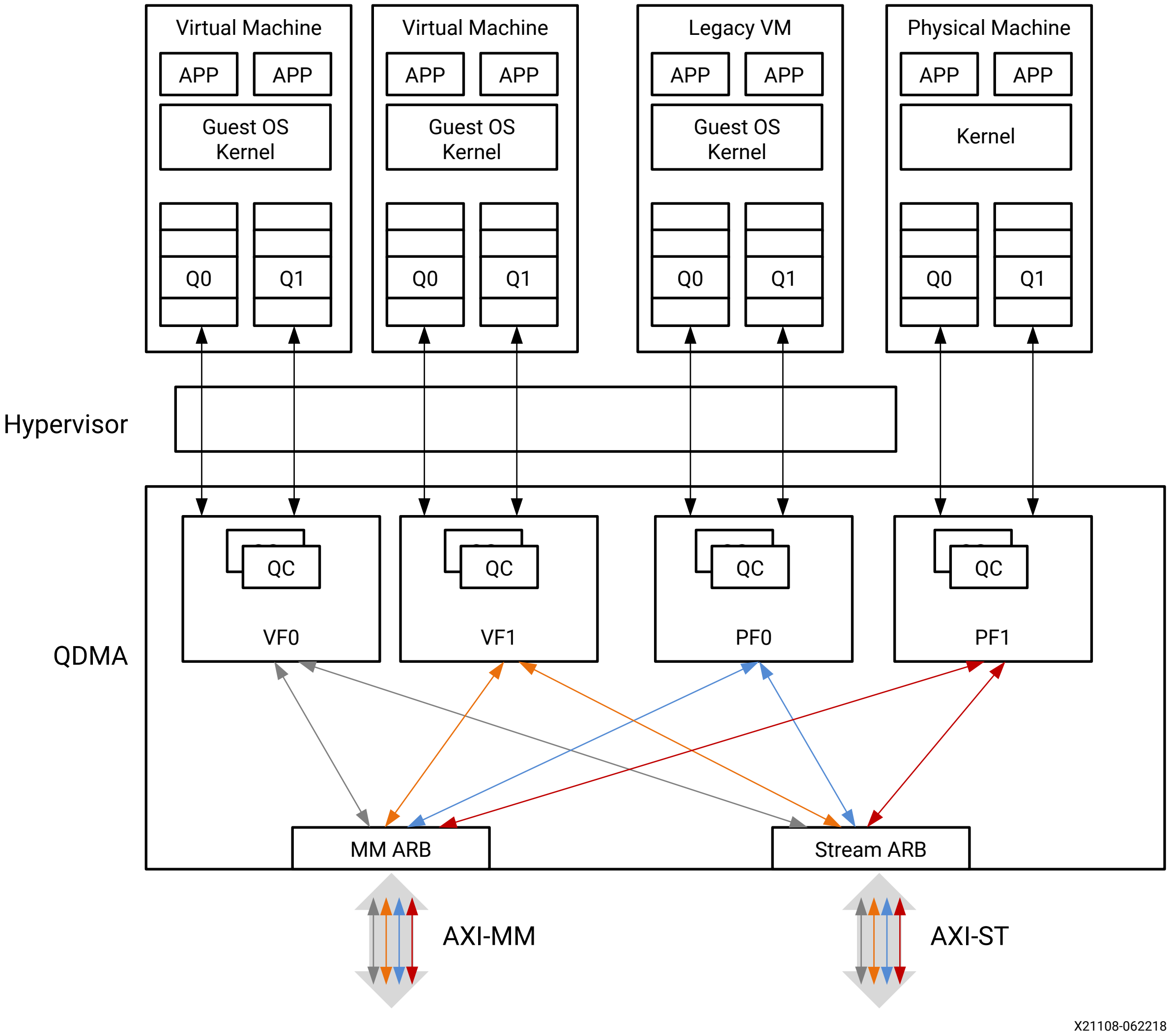
Security is an important aspect of virtualization. The QDMA Subsystem for PCI Express offers the following security functionality:

- QDMA allows only privileged PF to configure the per queue context and registers. VFs inform the corresponding PFs of any queue context programming.
- Drivers are allowed to do pointer updates only for the queue allocated to them.
- The system IOMMU can be turned on to check that the DMA access is being requested by PFs or VFs. The ARID comes from queue context programmed by a privileged function.

Any PF or VF can communicate to a PF (not itself) through mailbox. Each function implements one 128B inbox and 128B outbox. These mailboxes are visible to the driver in the DMA BAR (typically BAR0) of its own function. At any given time, any function can have one outgoing mailbox and one incoming mailbox message outstanding per function.

The diagram below shows how a typical system can use QDMA with different functions and operating systems. Different Queues can be allocated to different functions, and each function can transfer DMA packets independent of each other.

Figure 9: **QDMA in a System**



QDMA Subsystem Limitations

The limitations of the QDMA are as follows:

- The DMA supports a maximum of 256 Queues on any VF function.
- Slave Bridge AXI does not support Narrow Burst transfers.

> ✓ **RECOMMENDED:** *Use AXI SmartConnect to support Narrow Burst.*

- SRIOV is not supported in bridge mode.

Applications

The QDMA is used in a broad range of networking, computing, and data storage applications. A common usage example for the QDMA is to implement Data Center and Telco applications, such as Compute acceleration, Smart NIC, NVMe, RDMA-enabled NIC (RNIC), server virtualization, and NFV in the user logic. Multiple applications can be implemented to share the QDMA by assigning different queue sets and PCIe functions to each application. These Queues can then be scaled in the user logic to implement rate limiting, traffic priority, and custom work queue entry (WQE).

Chapter 3

Product Specification

Performance and Resource Utilization

Performance

Xilinx provides two example designs for you to experiment with. Standard example design is for functional test only. To generate a example design for performance analysis, use the following Tcl command to generate a performance example design:

```
set_property CONFIG.performance_exdes{true} [get_ips qdma_0]
```

Below are the QDMA register settings recommended by Xilinx for better performance. Performance numbers will vary based on systems and which OS is being used.

- QDMA_C2H_INT_TIMER_TICK (0xB0C) set to 25. Corresponding to 100 ns (1 tick = 4 ns for 250 MHz user clock)
- C2H trigger mode set to Counter + Timer, with counter set to 64 and timer to match round trip latency. Global register for timer should have a value of 30 for 3 µs.
- QDMA_GLBL_DSC_CFG (0x250), `max_desc_fetch` = 6, `wb_int` = 5
- QDMA_H2C_REQ_THROT (0xE24), `req_throt_en_data` = 1, `data_thresh` = 0x4000
- QDMA_C2H_PFCH_CFG (0xB08/0xA80/0xA84)
  - `evt_qcnt_th` = (QDMA_C2H_PFCH_CACHE_DEPTH/2) - 2
  - `pfch_qcnt` = QDMA_C2H_PFCH_CACHE_DEPTH/2
  - `num_pfch` = 8. A minimum of 8 is recommended. In environments with low number of active queues, programing higher values can help to boost the performance.
  - `pfch_fl_th` = 256
- QDMA_C2H_WRB_COAL_CFG (0xB50),
  - `max_buf_sz` = QDMA_C2H_CMPT_COAL_BUF_DEPTH (0xBE4)
  - `tick_val` = 25
  - `tick_cnt` = 5
- TX/RX API burst size = 64, ring depth = 2048. The driver should update TX/RX PIDX in batches of 64.
- PCIe MPS = 256 bytes, MRRS >= 512 bytes, Extended Tag Enabled, Relaxed Ordering Enabled

- The driver will update the completion CIDX in batches of 64 to reduce number of MMIO writes before updating the C2H PIDX
- The driver should update the H2C PIDX in batches of 64, and also update for the last descriptor of the scatter gather list.
- C2H context:
  - `bypass` = 0 (Internal mode)
  - `frcd_en` = 1
  - `qen` = 1
  - `wbk_en` = 1
  - `irq_en` = `irq_arm` = `int_aggr` = 0
- C2H prefetch context:
  - `pfch` = 1
  - `bypass` = 0
  - `valid` = 1
- C2H CMPT context:
  - `en_stat_desc` = 1
  - `en_int` = 0 (Poll_mode)
  - `int_aggr` = 0 (Poll mode)
  - `trig_mode` = 5
  - `counter_idx` = corresponding to 64
  - `timer_idx` = corresponding to 3 µs
  - `valid` = 1
- H2C context:
  - `bypass` = 0 (Internal mode)
  - `frcd_en` = 0
  - `fetch_max` = 0
  - `qen` = 1
  - `wbk_en` = 1
  - `wbi_chk` = 1
  - `wbi_intvl_en` = 1
  - `irq_en` = 0 (Poll mode)
  - `irq_arm` = 0 (Poll mode)
  - `int_aggr` = 0 (Poll mode)

For optimal QDMA streaming performance, packet buffers of the descriptor ring should be aligned to at least 256 bytes.

✓ **RECOMMENDED:** *Xilinx recommends that you limit the total outstanding descriptor fetch to be less than 8 KB on the PCIe. For example, limit the outstanding credits across all queues to 512 for a 16B descriptor.*

Performance in Descriptor Bypass Mode

When the design is configured in descriptor bypass mode, all the above setting apply. The following information provides recommendations to improve performance in bypass mode.

1. When bypass in `h2c_byp_in_st_sdi` ports is set, the QDMA IP generates the status write back for every packet. Xilinx recommends that this port be asserted once in 32 packets, or 64 packets. And if there are no more descriptors left then assert `h2c_byp_in_st_sdi`at the last descriptor. This requirement is per queue basis, and applies to AXI4-MM (H2C and C2H) bypass transfers and AXI4-Stream H2C transfers.
2. For AXI-Stream C2H Simple bypass mode, the `dsc_crdt_in_fence` port should be set to 1 for performance reasons. This recommendation assumes the user design already coalesced credits for each queue and sent them to the IP. In internal mode, set the `fence` bit in the QDMA_C2H_PFCH_CFG_2 (0xA84) register.

Resources Utilization

For QDMA Resource Utilization, see Resource Use web page.

QDMA Operations

Descriptor Engine

The descriptor engine is responsible for managing the consumer side of the Host to Card (H2C) and Card to Host (C2H) descriptor ring buffers for each queue. The context for each queue determines how the descriptor engine will process each queue individually. When descriptors are available and other conditions are met, the descriptor engine will issue read requests to PCIe to fetch the descriptors. Received descriptors are offloaded to either the descriptor bypass out interface (bypass mode) or delivered directly to a DMA engine (internal mode). When a H2C Stream or Memory Mapped DMA engine completes a descriptor, status can be written back to the status descriptor, an interrupt, and/or a marker response can be generated to inform software and user logic of the current DMA progress. The descriptor engine also provides a Traffic Manager Interface which notifies user logic of certain status for each queue. This allows the user logic to make informed decisions if customization and optimization of DMA behavior is desired.

Descriptor Context

The Descriptor Engine stores per queue configuration, status and control information in descriptor context that can be stored in block RAM or UltraRAM, and the context is indexed by H2C or C2H QID. Prior to enabling the queue, the hardware and credit context must first be cleared. After this is done, the software context can be programmed and the `qen` bit can be set to enable the queue. After the queue is enabled, the software context should only be updated through the direct mapped address space to update the Producer Index and Interrupt ARM bit, unless the queue is being disabled. The hardware context and credit context contain only status. It is only necessary to interact with the hardware and credit contexts as part of queue initialization in order to clear them to all zeros. Once the queue is enabled, context is dynamically updated by hardware. Any modification of the context through the indirect bus when the queue is enabled can result in unexpected behavior. Reading the context when the queue is enabled is not recommended as it can result in reduced performance.

Related Information

QDMA_DMAP_SEL_H2C_DSC_PIDX[2048] (0x18004)
QDMA_DMAP_SEL_C2H_DSC_PIDX[2048] (0x18008)

Software Descriptor Context Structure (0x0 C2H and 0x1 H2C)

The descriptor context is used by the descriptor engine.

Table 5: **Software Descriptor Context Structure Definition**

Bit	Bit Width	Field Name	Description
[255:140]	116	reserved	Reserved. Set to 0s.
[139]	1	int_aggr	If set, interrupts will be aggregated in interrupt ring.
[138:128]	[10:0]	vec	MSI-X vector used for interrupts for direct interrupt or interrupt aggregation entry for aggregated interrupts.
[127:64]	64	dsc_base	4K aligned base address of descriptor ring.
[63]	1	is_mm	This field determines if the queue is Memory Mapped or not. If this field is set, the descriptors will be delivered to associated H2C or C2H MM engine. 1: Memory Mapped 0: Stream
[62]	1	mrkr_dis	If set, disables the marker response in internal mode. Not applicable for C2H ST.
[61]	1	irq_req	Interrupt due to error waiting to be sent (waiting for irq_arm). This bit should be cleared when the queue context is initialized. Not applicable for C2H ST.
[60]	1	err_wb_sent	A writeback/interrupt was sent for an error. Once this bit is set no more writebacks or interrupts will be sent for the queue. This bit should be cleared when the queue context is initialized. Not applicable for C2H ST.

Table 5: **Software Descriptor Context Structure Definition** *(cont'd)*

Bit	Bit Width	Field Name	Description
[59:58]	2	err	Error status. Bit[1] dma – An error occurred during DMA operation. Check engine status registers. Bit[0] dsc – An error occured during descriptor fetch or update. Check descriptor engine status registers. This field should be set to 0 when the queue context is initialized.
[57]	1	irq_no_last	No interrupt was sent and the producer index (PIDX) or consumer index (CIDX) was idle in internal mode. When the irq_arm bit is set, the interrupt will be sent. This bit will clear automatically when the interrupt is sent or if the PIDX of the queue is updated. This bit should be initialized to 0 when the queue context is initialized. Not applicable for C2H ST.
[56:54]	3	port_id	Port_id The port id that will be sent on user interfaces for events associated with this queue.
[53]	1	irq_en	Interrupt enable. An interrupt to the host will be sent on host status updates. Set to 0 for C2H ST.
[52]	1	wbk_en	Writeback enable. A memory write to the status descriptor will be sent on host status updates.
[51]	1	mm_chn	Set to 0 and cannot be modified.
[50]	1	bypass	If set, the queue will operate under Bypass mode, otherwise it will be in Internal mode.
[49:48]	2	dsc_sz	Descriptor fetch size. 0: 8B, 1: 16B; 2: 32B; 3: 64B. If bypass mode is not enabled, 32B is required for Memory Mapped DMA, 16B is required for H2C Stream DMA, and 8B is required for C2H Stream DMA. If the queue is configured for bypass mode, any descriptor size can be selected. The descriptors will be delivered on the bypass output interface. It is up to the user logic to process the descriptors before they are fed back into the descriptor bypass input.
[47:44]	4	rng_sz	Descriptor ring size index. This index selects one of 16 register (offset 0x204:0x240) which has different ring sizes.
[43:41]	3	reserved	Reserved
[40:37]	4	fetch_max	Maximum number of descriptor fetches oustanding for this queue. The max outstanding is fetch_max + 1. Higher value can increase the single queue performance,
[36]	1	at	Address type of base address. 0: untranslated 1: translated This will be the address type (AT) used on PCIe for descriptor fetches and status descriptor writebacks.

Table 5: **Software Descriptor Context Structure Definition** *(cont'd)*

Bit	Bit Width	Field Name	Description
[35]	1	wbi_intvl_en	Write back/Interrupt interval. Enables periodic status updates based on the number of descriptors processed. Applicable to Internal mode. Not Applicable to C2H ST. The writeback interval is determined by register QDMA_GLBL_DSC_CFG (0x250) bits[2:0].
[34]	1	wbi_chk	Writeback/Interrupt after pending check. Enable status updates when the queue has completed all available descriptors. Applicable to Internal mode.
[33]	1	fcrd_en	Enable fetch credit. The number of descriptors fetched will be qualified by the number of credits given to this queue. Set to 1 for C2H ST.
[32]	1	qen	Indicates that the queue is enabled.
[31:25]	7	reserved	Reserved
[24:17]	8	fnc_id	Function ID
[16]	1	irq_arm	Interrupt arm. When this bit is set, the queue is allowed to generate an interrupt.
[15:0]	16	pidx	Producer index.

Hardware Descriptor Context Structure (0x2 C2H and 0x3 H2C)

Table 6: **Hardware Descriptor Structure Definition**

Bit	Bit Width	Field Name	Description
[47]	1	reserved	Reserved
[46:43]	4	fetch_pnd	Descriptor fetch pending
[42]	1	evt_pnd	Event pending
[41]	1	idl_stp_b	Queue invalid and no descriptors pending. This bit is set when the queue is enabled. The bit is cleared when the queue has been disabled (software context qen bit) and no more descriptor are pending.
[40]	1	dsc_pnd	Descriptors pending. Descriptors are defined to be pending if the last CIDX completed does not match the current PIDX.
[39:32]	8		Reserved
[31:16]	16	crd_use	Credits consumed. Applicable if fetch credits are enabled in the software context.
[15:0]	16	cidx	Consumer index of last fetched descriptor.

Credit Descriptor Context Structure

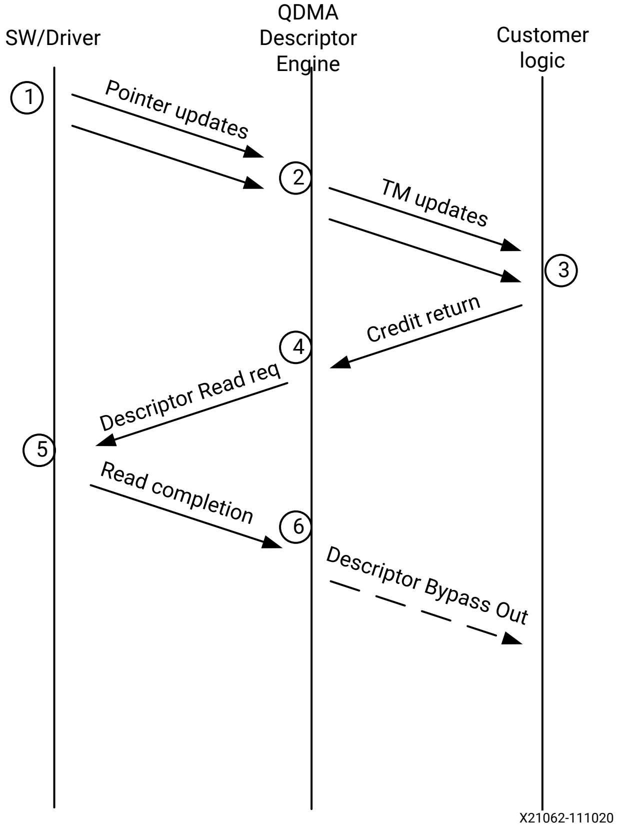
Table 7: **Credit Descriptor Context Structure Definition**

Bit	Bit Width	Field Name	Description
[31:16]	16	reserved	Reserved
[15:0]	16	credt	Fetch credits received. Applicable if fetch credits are enabled in the software context.

The credit descriptor context is for internal DMA use only and can be read from the indirect bus for debug. This context stores credits for each queue that have been received through the Descriptor Credit Interface with the CREDIT_ADD operation. If the credit operation has the `dsc_crdt_in_fence` bit set to 1, credits are added only as the read request for the descriptor is generated.

Descriptor Fetch

Figure 10: **Descriptor Fetch Flow**



1. The descriptor engine is informed of the availability of descriptors through an update to a queue's descriptor PIDX. This portion of the context is direct mapped to the QDMA_DMAP_SEL_H2C_DSC_PIDX and QDMA_DMAP_SEL_C2H_DSC_PIDX address space.
2. On a PIDX update, the descriptor engine evaluates the number of descriptors available based on the last fetched consumer index (CIDX). The availability of new descriptors is communicated to the user logic through the Traffic Manager Status Interface.
3. If fetch crediting is enabled, the user logic is required to provide a credit for each descriptor that should be fetched.
4. If descriptors are available and either fetch credits are disabled or are non-zero, the descriptor engine will generate a descriptor fetch to PCIe. The number of fetched descriptors is further qualified by the PCIe Max Read Request Size (MRRS) and descriptor fetch credits, if enabled. A descriptor fetch can also be stalled due to insufficient completion space. In each direction, C2H and H2C are allocated 256 entries for descriptor fetch completions. Each entry is the width of the datapath. If sufficient space is available, the fetch is allowed to proceed. A given queue can only have one descriptor fetch pending on PCIe at any time.
5. The host receives the read request and provides the descriptor read completion to the descriptor engine.
6. Descriptors are stored in a buffer until they can be offloaded. If the queue is configured in bypass mode, the descriptors are sent to the Descriptor Bypass Output port. Otherwise they are delivered directly to a DMA engine. Once delivered, the descriptor fetch completion buffer space is deallocated.

***Note*:** Available descriptors are always <ring size> - 2. At any time, the software should not update the PIDX to more than <ring size> - 2.

For example, if queue size is 8, which contains the entry index 0 to 7, the last entry (index 7) is reserved for status. This index should never be used for the PIDX update, and the PIDX update should never be equal to CIDX. For this case, if CIDX is 0, the maximum PIDX update would be 6.

Internal Mode

A queue can be configured to operate in Descriptor Bypass mode or Internal mode by setting the software context bypass field. In internal mode, the queue requires no external user logic to handle descriptors. Descriptors that are fetched by the descriptor engine are delivered directly to the appropriate DMA engine and processed. Internal mode allows credit fetching and status updates to the user logic for run time customization of the descriptor fetch behavior.

Internal Mode Writeback and Interrupts (AXI MM and H2C ST)

Status writebacks and/or interrupts are generated automatically by hardware based on the queue context. When `wbi_intvl_en` is set, writebacks/interrupts will be sent based on the interval selected in the register QDMA_GLBL_DSC_CFG (0x250) bits[2:0]. Due to the slow nature of interrupts, in interval mode, interrupts may be late or skip intervals. If the `wbi_chk` context bit is set, a writeback/interrupt will be sent when the descriptor engine has detected that the last descriptor at the current PIDX has completed. It is recommended the `wbi_chk` bit be set for all internal mode operation, including when interval mode is enabled. An interrupt will not be generated until the `irq_arm` bit has been set by software. Once an interrupt has been sent the `irq_arm` bit is cleared by hardware. Should an interrupt be needed when the `irq_arm` bit is not set, the interrupt will be held in a pending state until the `irq_arm` bit is set.

Descriptor completion is defined to be when the descriptor data transfer has completed and its write data has been acknowledged on AXI (H2C bresp for AXI MM, Valid/Ready of ST), or been accepted by the PCIe Controller's transaction layer for transmission (C2H MM).

Descriptor Bypass Mode

Descriptor Bypass mode also supports crediting and status updates to user logic. In addition, Descriptor Bypass mode allows the user logic to customize processing of descriptors and status updates. Descriptors fetched by the descriptor engine are delivered to user logic through the descriptor bypass out interface. This allows user logic to pre-process or store the descriptors, if desired. On the descriptor bypass out interface, the descriptors can be a custom format (adhering to the descriptor size). To perform DMA operations, the user logic drives descriptors (must be QDMA format) into the descriptor bypass input interface.

If the user logic already has descriptors, which must be in QDMA format, it can be provided directly to the DMA through the descriptor bypass ports. The user logic does not need to fetch descriptors from the host if the descriptors are already in the user logic. The user logic should not send credits through the descriptor credit interface.

Descriptor Bypass Mode Writeback/Interrupts

In bypass mode, the user logic has explicit control over status updates to the host, and marker responses back to user logic. Along with each descriptor submitted to the Descriptor Bypass Input Port for a Memory Mapped Engine (H2C and C2H) or H2C Stream DMA engine, there is a CIDX, and `sdi` field. The CIDX is used to identify which descriptor has completed in any status update (host writeback, marker response, or coalesced interrupt) generated at the completion of the descriptor. If the `sdi` field of the descriptor was input, then a writeback to the host will be generated if the context `wbk_en` bit is set. An interrupt can also be sent if the `sdi` bit is set if the context `irq_en` and `irq_arm` bits are set.

If interrupts are enabled, the user logic must monitor the traffic manager output for the `irq_arm`. After the `irq_arm` bit has been observed for the queue, a descriptor with the `sdi` bit will be sent to the DMA. Once a descriptor with the `sdi` bit has been sent, another `irq_arm` assertion must be observed before another descriptor with the `sdi` bit can be sent. If the user sets the `sdi` bit when the arm bit has not be properly observed, an interrupt may or may not be sent, and software might hang indefinitely waiting for an interrupt. When interrupts are not enabled, setting the `sdi` bit has no restriction. However excessive writeback events can severely reduce the descriptor engine performance and consume write bandwidth to the host.

Descriptor completion is defined to be when the descriptor data transfer has completed and its write data has been acknowledged on AXI4 (H2C `bresp` for AXI MM, Valid/Ready of ST), or been accepted by the PCIe Controller's transaction layer for transmission (C2H MM).

Marker Response

Marker responses can be generated for any descriptor by setting the `mrkr_req` bit. Marker responses are generated after the descriptor is completed. Similar to host writebacks, excessive marker response requests can reduce descriptor engine performance. Marker responses to the user logic can also be sent with the `sbi` bit if configured in the context. The Marker responses are sent on Queue Status ports which can be identified by the queue id.

Descriptor completion is defined as when the descriptor data transfer has completed and its write data is acknowledged on AXI (H2C `bresp` for AXI MM, Valid/Ready of ST), or is accepted by the PCIe Controller's transaction layer for transmission (C2H MM).

Traffic Manager Output Interface

The traffic manager interface provides details of a queue's status to user logic, allowing user logic to manage descriptor fetching and execution. In normal operation, for an enabled queue, each time the `irq_arm` bit is asserted or PIDX of a queue is updated, the descriptor engine asserts `tm_dsc_sts_valid`. The `tm_dsc_sts_avl` signal indicates the number of new descriptors available since the last update. Through this mechanism, user logic can track the amount of work available for each queue. This can be used for prioritizing fetches through the descriptor engine's fetch crediting mechanism or other user optimizations. On the valid cycle, the `tm_dsc_sts_irq_arm` indicates that the `irq_arm` bit was zero and was set. In bypass mode, this is essentially a credit for an interrupt for this queue. When a queue is invalidated by software or due to error, the `tm_dsc_sts_qinv` signal will be set. If this bit is observed, the descriptor engine will have halted new descriptor fetches for that queue. In this case, the contents on `tm_dsc_sts_avl` indicate the number of available fetch credits held by the descriptor engine. This information can be used to help user logic reconcile the number of credits given to the descriptor engine, and the number of descriptors it should expect to receive. Even after `tm_dsc_sts_qin` is asserted, valid descriptors already in the fetch pipeline will continue to be delivered to the DMA engine (internal mode) or delivered to the descriptor bypass output port (bypass mode).

Other fields of the `tm_dsc_sts` interface identify the queue id, DMA direction (H2C or C2H), internal or bypass mode, stream or memory mapped mode, queue enable status, queue error status, and port ID.

While the `tm_dsc_sts` interface is a valid/ready interface, it should not be back-pressured for optimal performance. Since multiple events trigger a `tm_dsc_sts` cycle, if internal buffering is filled, descriptor fetching will be halted to prevent generation of new events.

Related Information

QDMA Traffic Manager Credit Output Ports

Descriptor Credit Input Interface

The credit interface is relevant when a queue's `fcrd_en` context bit is set. It allows the user logic to prioritize and meter descriptors fetched for each queue. You can specify the DMA direction, qid, and credit value. For a typical use case, the descriptor engine uses credit inputs to fetch descriptors. Internally, credits received and consumed are tracked for each queue. If credits are added when the queue is not enabled, the credits will be returned through the Traffic Manager Output Interface with `tm_dsc_sts_qinv` asserted, and the credits in `tm_dsc_sts_avl` are not valid. Monitor `tm_dsc_sts` interface to keep an account for each queue on how many credits are consumed.

Related Information

QDMA Descriptor Credit Input Ports

Errors

Errors can potentially occur during both descriptor fetch and descriptor execution. In both cases, once an error is detected for a queue it will invalidate the queue, log an error bit in the context, stop fetching new descriptors for the queue which encountered the error, and can also log errors in status registers. If enabled for writeback, interrupts, or marker response, the DMA will generate a status update to these interfaces. Once this is done, no additional writeback, interrupts, or marker responses (internal mode) will be sent for the queue until the queue context is cleared. As a result of the queue invalidation due to an error, a Traffic Manager Output cycle will also be generated to indicate the error and queue invalidation. After the queue is invalidated, if there is an error you can determine the cause by reading the error registers and context for that queue. You must clear and remove that queue, and then add the queue back later when needed.

Although additional descriptor fetches will be halted, fetches already in the pipeline will continue to be processed and descriptors will be delivered to a DMA engine or Descriptor Bypass Out interface as usual. If the descriptor fetch itself encounters an error, the descriptor will be marked with an error bit. If the error bit is set, the contents of the descriptor should be considered invalid. It is possible that subsequent descriptor fetches for the same queue do not encounter an error and will not have the error bit set.

Memory Mapped DMA

In memory mapped DMA operations, both the source and destination of the DMA are memory mapped space. In an H2C transfer, the source address belongs to PCIe address space while the destination address belongs to AXI MM address space. In a C2H transfer, the source address belongs to AXI MM address space while the destination address belongs to PCIe address space. PCIe-to-PCIe, and AXI MM-to-AXI MM DMAs are not supported. Aside from the direction of the DMA, transfer H2C and C2H DMA behave similarly and share the same descriptor format.

Operation

The memory mapped DMA engines (H2C and C2H) are enabled by setting the `run` bit in the Memory Mapped Engine Control Register. When the `run` bit is deasserted, descriptors can be dropped. Any descriptors that have already started the source buffer fetch will continue to be processed. Reassertion of the `run` bit will result in resetting internal engine state and should only be done when the engine is quiesced. Descriptors are received from either the descriptor engine directly or the Descriptor Bypass Input interface. Any queue that is in internal mode should not be given descriptors through the Descriptor Bypass Input interface. Any descriptor sent to an MM engine that is not running will be dropped. For configurations where a mix of Internal Mode queues and Bypass Mode queues are enabled, round robin arbitration is performed to establish order.

The DMA Memory Mapped engine first generates the read request to the source interface, splitting the descriptor at alignment boundaries specific to the interface. Both PCIe and AXI read interfaces can be configured to split at different alignments. Completion space for read data is preallocated when the read is issued. Likewise for the write requests, the DMA engine will split at appropriate alignments. On the AXI interface each engine will use a single AXI ID. The DMA engine will reorder the read completion/write data to the order in which the reads were issued. Once sufficient read completion data is received the write request will be issued to the destination interface in the same order that the read data was requested. Before the request is retired, the destination interfaces must accept all the write data and provide a completion response. For PCIe the write completion is issued when the write request has been accepted by the transaction layer and will be sent on the link next. For the AXI Memory Mapped interface, the `bresp` is the completion criteria. Once the completion criteria has been met, the host writeback, interrupt and/or marker response is generated for the descriptor as appropriate.

The DMA Memory Mapped engines also support the `no_dma` field of the Descriptor Bypass Input, and zero-length DMA. Both cases are treated identically in the engine. The descriptors propagate through the DMA engine as all other descriptors, so descriptor ordering within a queue is still observed. However no DMA read or write requests are generated. The status update (writeback, interrupt, and/or marker response) for zero-length/`no_dma` descriptors is processed when all previous descriptors have completed their status update checks.

Errors

There are two primary error categories for the DMA Memory Mapped Engine. The first is an error bit that is set with an incoming descriptor. In this case, the DMA operation of the descriptor is not processed but the descriptor will proceed through the engine to status update phase with an error indication. This should result in a writeback, interrupt, and/or marker response depending on context and configuration. It will also result in the queue being invalidated. The second category of errors for the DMA Memory Mapped Engine are errors encountered during the execution of the DMA itself. This can include PCIe read completions errors, and AXI `bresp` errors (H2C), or AXI `bresp` errors and PCIe write errors due to bus master enable or function level reset (FLR), as well as RAM ECC errors. The first enabled error is logged in the DMA engine. Please refer to the Memory Mapped Engine error logs. If an error occurs on the read, the DMA write will be aborted if possible. If the error was detected when pulling write data from RAM, it is not possible to abort the request. Instead invalid data parity will be generated to ensure the destination is aware of the problem. After the descriptor which encountered the error has gone through the DMA engine, it will proceed to generate status updates with an error indication. As with descriptor errors, it will result in the queue being invalidated.

AXI Memory Mapped Descriptor for H2C and C2H (32B)

Table 8: **AXI Memory Mapped Descriptor Structure for H2C and C2H**

Bit	Bit Width	Field Name	Description
[255:192]	64	reserved	Reserved
[191:128]	64	dst_addr	Destination Address
[127:92]	36	reserved	Reserved
[91:64]	28	lengthInByte	Read length in byte
[63:0]	64	src_addr	Source Address

Internal mode memory mapped DMA must configure the descriptor queue to be 32B and follow the above descritor format. In bypass mode, the descriptor format is defined by the user logic, which must drive the H2C or C2H MM bypass input port.

AXI Memory Mapped Writeback Status Structure for H2C and C2H

The MM writeback status register is located after the last entry of the (H2C or C2H) descriptor.

Table 9: **AXI Memory Mapped Writeback Status Structure for H2C and C2H**

Bit	Bit Width	Field Name	Description
[63:48]	16	reserved	Reserved
[47:32]	16	pidx	Producer Index at time of writeback

Table 9: **AXI Memory Mapped Writeback Status Structure for H2C and C2H** *(cont'd)*

Bit	Bit Width	Field Name	Description
[31:16]	16	cidx	Consumer Index
[15:2]	14	reserved	Reserved
[1:0]	2	err	Error bit 1: Descriptor fetch error bit 0: DMA error

Stream Mode DMA

H2C Stream Engine

The H2C Stream Engine is responsible for transferring streaming data from the host and delivering it to the user logic. The H2C Stream Engine operates on H2C stream descriptors. Each descriptor specifies the start address and the length of the data to be transferred to the user logic. The H2C Stream Engine parses the descriptor and issues read requests to the host over PCIe, splitting the read requests at the MRRS boundary. There can be up to 256 requests outstanding in the H2C Stream Engine to hide the host read latency. The H2C Stream Engine implements a re-ordering buffer of 32 KB to re-order the TLPs as they come back. Data is issued to the user logic in order of the requests sent to PCIe.

If the status descriptor is enabled in the associated H2C context, the engine could additionally send a status write back to host once it is done issuing data to the user logic.

Internal and Bypass Modes

Each queue in QDMA can be programmed in either of the two H2C Stream modes: internal and bypass. This is done by specifying the mode in the queue context. The H2C Stream Engine knows whether the descriptor being processed is for a queue in internal or bypass mode.

The following figures show the internal mode and bypass mode flows.

Figure 11: **H2C Internal Mode Flow**

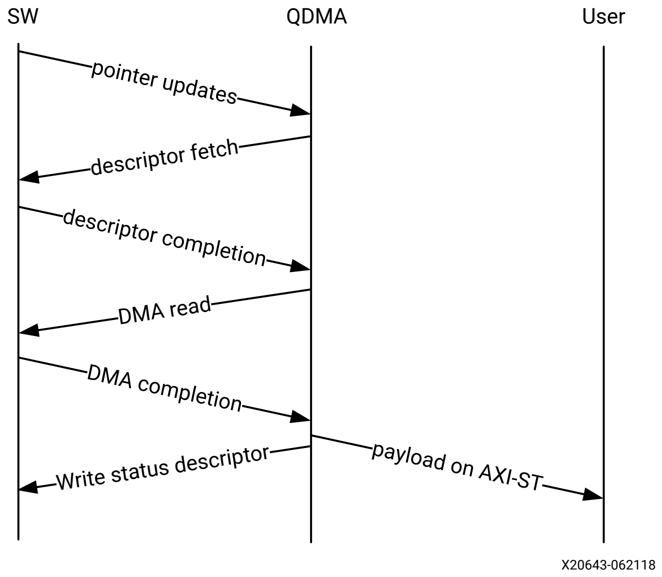
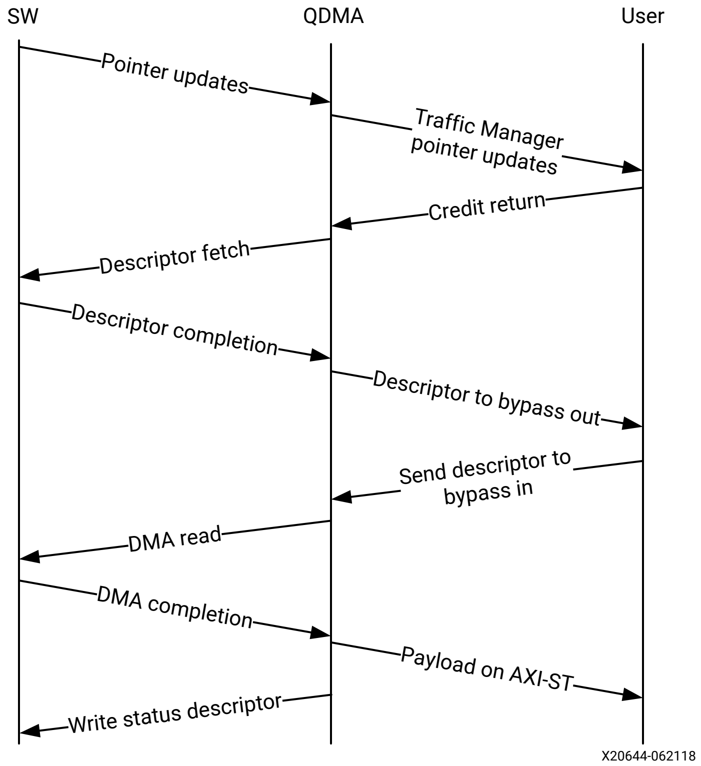


Figure 12: **H2C Bypass Mode Flow**



For a queue in **Internal mode**, after the descriptor is fetched from the host it is fed straight to the H2C Stream Engine for processing. In this case, a packet of data cannot span over multiple descriptors. Thus for a queue in internal mode, each descriptor generates exactly one AXI4-Stream packet on the QDMA H2C AXI4-Stream output. If the packet is present in host memory in non-contiguous space, then it has to be defined by more than one descriptor and this requires that the queue be programmed in bypass mode.

In the **Bypass mode**, after the descriptors are fetched from the host they are sent straight to the user logic using the QDMA bypass output port. The QDMA does not parse these descriptors at all. The user logic can store these descriptors and then send the required information from these descriptors back to QDMA using the QDMA H2C Stream descriptor bypass-in interface. Using this information, the QDMA constructs descriptors which are then fed to the H2C Stream Engine for processing.

When `fcrd_en` is enabled in the software context, DMA will wait for the user application to provide credits, "Credit return" in the picture above. When `fcrd_en` is not set, the DMA uses a pointer update, fetches descriptors and sends the descriptor out. The user application should not send in credits. "Credit return" in the above picture does not apply in this case.

The following are the advantages of using the bypass mode:

- The user logic can have a custom descriptor format. This is possible because QDMA does not parse descriptors for queues in bypass mode. The user logic parses these descriptors and provides the information required by the QDMA on the H2C Stream bypass-in interface.
- Immediate data can be passed from the software to the user logic without DMA operation.
- The user logic can do traffic management by sending the descriptors to the QDMA when it is ready to sink all the data. Descriptors can be cached in local RAM.
- Perform address translation.

There are some requirements imposed on the user logic when using the bypass mode. Because the bypass mode allows a packet to span multiple descriptors, the user logic needs to indicate to QDMA which descriptor marks the Start-Of-Packet (SOP) and which marks the End-Of-Packet (EOP). At the QDMA H2C Stream bypass-in interface, among other pieces of information, the user logic needs to provide: Address, Length, SOP, and EOP. It is required that once the user logic feeds SOP descriptor information into QDMA, it must eventually feed EOP descriptor information also. Descriptors for these multi-descriptor packets must be fed in sequentially. Other descriptors not belonging to the packet must not be interleaved within the multi-descriptor packet. The user logic must accumulate the descriptors up to the EOP descriptor, before feeding them back to QDMA. Not doing so can result in a hang. The QDMA will generate a TLAST at the QDMA H2C AXI Stream data output once it issues the the last beat for the EOP descriptor. This is guaranteed because the user is required to submit the descriptors for a given packet sequentially.

The H2C stream interface is shared by all the queues, and has the potential for a head of line blocking issue if the user logic does not reserve the space to sink the packet. Quality of service can be severely affected if the packet sizes are large. The Stream engine is designed to saturate PCIe for packet sizes as low as 128B, so Xilinx recommends that you restrict the packet size to be host page size or maximum transfer unit as required by the user application.

A performance control provided in the H2C Stream Engine is the ability to stall requests from being issued to the PCIe RQ/RC if a certain amount of data is outstanding on the PCIe side as seen by the H2C Stream Engine. To use this feature, the SW must program a threshold value in the H2C_REQ_THROT (0xE24) register. After the H2C Stream Engine has more data outstanding to be delivered to the user logic than this threshold, it stops sending further read requests to the PCIe RQ/RC. This feature is disabled by default and can be enabled with the H2C_REQ_THROT (0xE24) register. This feature helps improve the C2H Stream performance, because the H2C Stream Engine can make requests at a much faster rate than the C2H Stream Engine. This can potentially use up the PCIe side resources for H2C traffic which results in C2H traffic suffering. The H2C_REQ_THROT (0xE24) register also allows the SW to separately enable and program the threshold of the maximum number of read requests that can be outstanding in the H2C Stream engine. Thus, this register can be used to individually enable and program the thresholds for the outstanding requests and data in the H2C Stream engine.

H2C Stream Descriptor (16B)

Table 10: **H2C Descriptor Structure**

Bit	Bit Width	Field Name	Description
[127:96]	32	addr_h	Address High. Higher 32 bits of the source address in Host
[95:64]	32	addr_l	Address Low. Lower 32 bits of the source address in Host
[63:48]	16	reserved	Reserved
[47:32]	16	len	Packet Length. Length of the data to be fetched for this descriptor. This is also the packet length since in internal mode, a packet cannot span multiple descriptors. The maximum length of the packet can be 64K-1 bytes.
[31:0]	32	metadata	Metadata. QDMA passes this field on the H2C-ST TUSER along with the data on every beat. For a queue in internal mode, it can be used to pass messages from SW to user logic along with the data.

This H2C descriptor format is only applicable for internal mode. For bypass mode, the user logic can define its own format as needed by the user application.

Descriptor Metadata

Similar to bypass mode, the internal mode also provides a mechanism to pass information directly from the software to the user logic. In addition to address and length, the H2C Stream descriptor also has a 32b metadata field. This field is not used by the QDMA for the DMA operation. Instead, it is passed on to the user logic on the H2C AXI4-Stream `tuser` on every beat of the packet. Passing metadata on the `tuser` is not supported for a queue in bypass mode and consequently there is no input to provide the metadata on the QDMA H2C Stream bypass-in interface.

Zero Length Descriptor

The length field in a descriptor can be zero. In this case, the H2C Stream Engine will issue a zero byte read request on PCIe. After the QDMA receives the completion for the request, the H2C Stream Engine will send out one beat of data with `tlast` on the QDMA H2C AXI4-Stream interface. The zero byte packet will be indicated on the interface by setting the `zero_b_dma` bit in the `tuser`. The user logic must set both the SOP and EOP for a zero byte descriptor. If not done, an error will be flagged by the H2C Stream Engine.

H2C Stream Status Descriptor Writeback

When feeding the descriptor information on the bypass input interface, the user logic can request the QDMA to send a status write back to the host when it is done fetching the data from the host. The user logic can also request that a status be issued to it when the DMA is done. These behaviors can be controlled using the `sdi` and `mrkr_req` inputs in the bypass input interface.

The H2C writeback status register is located after the last entry of the H2C descriptor list.

***Note*:** The format of the H2C-ST status descriptor written to the descriptor ring is different from that written into the interrupt coalesce entry.

Table 11: **AXI4-Stream H2C Writeback Status Descriptor Structure**

Bit	Bit Width	Field Name	Description
[63:32]	32	reserved	Reserved
[47:32]	16	pidx	Producer Index
[31:16]	16	cidx	Consumer Index
[15:2]	14	reserved	Reserved (Producer Index)
[1:0]	2	error	Error 0x0 : No Error 0x1 : Descriptor or data error was encountered on this queue 0x2 and 0x3 : Reserved

Related Information

QDMA Descriptor Bypass Input Ports

H2C Stream Data Aligner

The H2C engine has a data aligner that aligns the data to zero Bytes (0B) boundary before issuing it to the user logic. This allows the start address of a descriptor to be arbitrarily aligned and still receive the data on the H2C AXI4-Stream data bus without any holes at the beginning of the data. The user logic can send a batch of descriptors from SOP to EOP with arbitrary address and length alignments for each descriptor. The aligner will align and pack the data from the different descriptors and will issue a continuous stream of data on the H2C AXI4-Stream data bus. The `tlast` on that interface will be asserted when the last beat for the EOP descriptor is being issued.

Handling Descriptors With Errors

If an error is encountered while fetching a descriptor, the QDMA Descriptor Engine flags the descriptor with error. For a queue in internal mode, the H2C Stream Engine handles the error descriptor by not performing any PCIe or DMA activity. Instead, it waits for the error descriptor to pass through the pipeline and forces a writeback after it is done. For a queue in bypass mode, it is the responsibility of the user logic to not issue a batch of descriptors with an error descriptor. Instead, it must send just one descriptor with error input asserted on the H2C Stream bypass-in interface and set the SOP, EOP, `no_dma` signal, and `sdi` or `mrkr-req` signal to make the H2C Stream Engine send a writeback to Host.

Handling Errors in Data From PCIe

If the H2C Stream Engine encounters an error coming from PCIe on the data, it keeps the error sticky across the full packet. The error is indicated to the user on the `err` bit on the H2C Stream Data Output. Once the H2C Stream sends out the last beat of a packet that saw a PCIe data error, it also sends a Writeback to the Software to inform it about the error.

C2H Stream Engine

The C2H Stream Engine DMA writes the stream packets to the host memory into the descriptor provided by the host driver through the C2H descriptor queue.

The Prefetch Engine is responsible for calculating the number of descriptors needed for the DMA that is writing the packet. The buffer size is fixed per queue basis. For internal and cached bypass mode, the prefetch module can fetch up to 512 descriptors for a maximum of 64 different queues at any given time.

The Prefetch Engine also offers low latency feature `pfch_en = 1`, where the engine can prefetch up to `qdma_c2h_pfch_cfg.num_pfch` descriptors upon receiving the packet, so that subsequent packets can avoid the PCIe latency.

The QDMA requires software to post full ring size so the C2H stream engine can fetch the needed number of descriptors for all received packets. If there are not enough descriptors in the descriptor ring, the QDMA will stall the packet transfer. For performance reasons, the software is required to post the PIDX as soon as possible to ensure there are always enough descriptors in the ring.

C2H stream packet data length is limited to 31 * C2H buffer size.

In older versions (such as 2018.3), C2H stream packet data length was limited to 7 * C2H buffer size.

C2H Stream Descriptor (8B)

Table 12: **AXI4-Stream C2H Descriptor Structure**

Bit	Bit Width	Field Name	Description
[63:0]	64	addr	Destination Address

C2H Prefetch Engine

The prefetch engine interacts between the descriptor fetch engine and C2H DMA write engine to pair up the descriptor and its payload.

Table 13: **C2H Prefetch Context Structure**

Bit	Bit Width	Field Name	Description
[45]	1	valid	Context is valid
[44:29]	16	sw_crdt	Software credit This field is written by the hardware for internal use. The software must initialize it to 0 and then treat it as read-only.
[28]	1	pfch	Queue is in prefetch This field is written by the hardware for internal use. The software must initialize it to 0 and then treat it as read-only.
[27]	1	pfch_en	Enable prefetch
[26]	1	err	Error detected on this queue During the descriptor per-fetch process, if there are any errors detected it will be logged here. This will be per queue basis.
[25:8]	18	reserved	Reserved
[7:5]	3	port_id	Port ID
[4:1]	4	buf_size_idx	Buffer size index
[0]	1	bypass	C2H is in bypass mode

C2H Stream Modes

The C2H descriptors can be from the descriptor fetch engine or C2H bypass input interfaces. The descriptors from the descriptor fetch engine are always in cache mode. The prefetch engine keeps the order of the descriptors to pair with the C2H data packets from the user. The descriptors from the C2H bypass input interfaces have one interface for both simple mode and cache mode (note that both simple bypass and cache bypass use the same interface). For simple mode, the user application keeps the order of the descriptors to pair with the C2H data packets. For cache mode, the prefetch engine keeps the order of the descriptors to pair with the C2H data packet from the user.

The prefetch context has a bypass bit. When it is `1'b1`, the user application sends the credits for the descriptors. When it is `1'b0`, the prefetch engine handles the credits for the descriptors.

The descriptor context has a bypass bit. When it is `1'b1`, the descriptor fetch engine sends out the descriptors on the C2H bypass output interface. The user application can convert it and later loop it back to the QDMA on the C2H bypass input interface. When the bypass context bit is `1'b0`, the descriptor fetch engine sends the descriptors to the prefetch engine directly.

On a per queue basis, three modes are supported.

- Cache Internal Mode
- Cache Bypass Mode
- Simple Bypass Mode

The selection between Simple Bypass Mode and Cache Bypass Mode is done by setting the bypass bits in Software Descriptor Context and C2H Prefetch Context as shown in the table below.

Note: If you already have the descriptor cached on the device, there is no need to fetch one from the host and you should follow the simple bypass mode for the C2H Stream application. In simple bypass mode, do not provide credits to fetch the descriptor, and instead, you need to send in the descriptor on the descriptor bypass interface.

Note: AXI4-Stream C2H **Simple Bypass mode** and **Cache Bypass mode** both use same bypass in ports (`c2h_byp_in_st_csh_*` ports).

Table 14: **C2H Stream Modes**

	c2h_byp_in port	desc_ctxt.desc_byp	pfch_ctxt.bypass
Simple bypass mode	c2h_byp_in_st_csh_*	1	1
Cache bypass mode	c2h_byp_in_st_csh_*	1	0
Cache internal mode	N/A	0	0

Simple Bypass Mode

For simple bypass mode, the descriptor fetch engine sends the descriptors out on the C2H bypass out interface. The user application converts the descriptor and loops it back to the QDMA on the simple mode C2H bypass input interface. The user application sends the credits for the descriptors, and it also keeps the order of the descriptors.

For simple bypass transfer to work, a prefetch tag is needed and it can be fetched from the QDMA IP.

The user application must request a prefetch tag before sending any traffic for a simple bypass queue through the C2H ST engine. Invalid queues or non-bypass queues should not request any tags using this method, since it may reduce performance by freezing tags that never get used.

The prefetch tag needs to be reserved upfront before any traffic can start. One prefetch tag per target host is required. In most applications, one prefetch tag for a host is needed. In Simple Bypass mode, the tag is not tied to any descriptor ring. For the queues that share the same prefetch tag, the data and descriptors need to come in the same order. For Simple Bypass, the data and descriptors are both controlled by the user, so they need to guarantee the order is maintained.

For example when the data stream has packets in the order of Q0, Q1, and Q2, on descriptor input, you can not send Q1, Q2, Q0 etc. The order needs to be maintained.

The user application writes to the MDMA_C2H_PFCH_BYP_QID (0x1408) register with the qid for a simple bypass queue, then reads from MDMA_C2H_PFCH_BYP_TAG (0x140C) register to retrieve the corresponding prefetch tag. This tag must be driven with all bypass_in descriptors for as long as the current qid is valid. If a current qid is invalidated, a new prefetch tag must be requested with a valid qid.

Prefetched tag must be assigned to input port `c2h_byp_in_st_csh_pfch_tag[6:0]` for all transfers. The prefetch tag points to the CAM that stores the active queues in the prefetch engine. Also the qid that was used to prefetch tag needs to be used as the qid for all simple bypass packets. Assign the qid to `s_axis_c2h_ctrl_qid`.

The steps to fetch the prefetch tag are as follows:

1. Software instruction:
    a. Initialize a queue (`qid`).
    b. Write to MDMA_C2H_PFCH_BYP_QID 0x1408 with valid `qid`.
    c. Read MDMA_C2H_PFCH_BYP_TAG 0x140C to obtain the prefetch tag.
    d. The prefetch tag and the `qid` that was used to fetch the tag should be used for all simple bypass packets. This information needs to be communicated to the user side.
2. User side:
    a. Assign the `qid` used to fetch the tag to `s_axis_c2h_ctrl_qid`.

b. Assign the actual `qid` of the packet transfer to `s_axis_c2h_cmpt_ctrl_qid`.

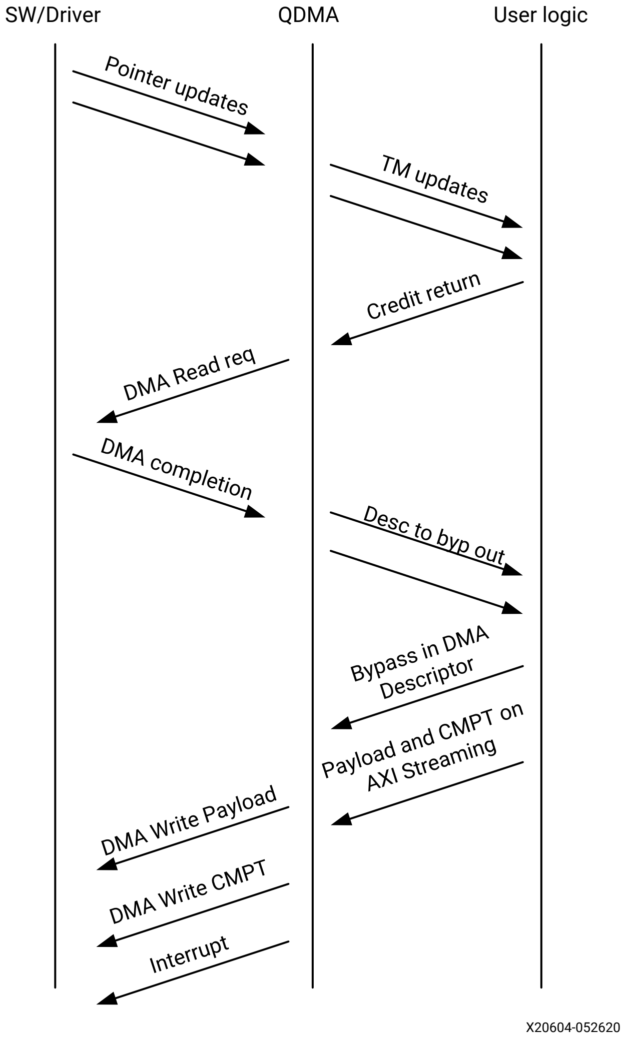
c. Assign the prefetch tag value to `c2h_byp_in_st_csh_pfch_tag`.

d. Assign the actual qid of the packet transfer to `c2h_byp_in_st_csh_qid`.

Note: The `c2h_byp_in_st_csh_pfch_tag[6:0]` port can have the same prefetch_tag for as long as the original qid is valid.

Simple bypass flow shown below does not include fetch of the "prefetch_tag".

Figure 13: **C2H Simple Bypass Mode Flow**



Note: No sequence is required between payload and completion packets.

If you already have descriptors, there is no need to update the pointers or provide credits. Instead, send the descriptors in the descriptor bypass interface, and send the data and Completion (CMPT) packets.

Cache Bypass Mode

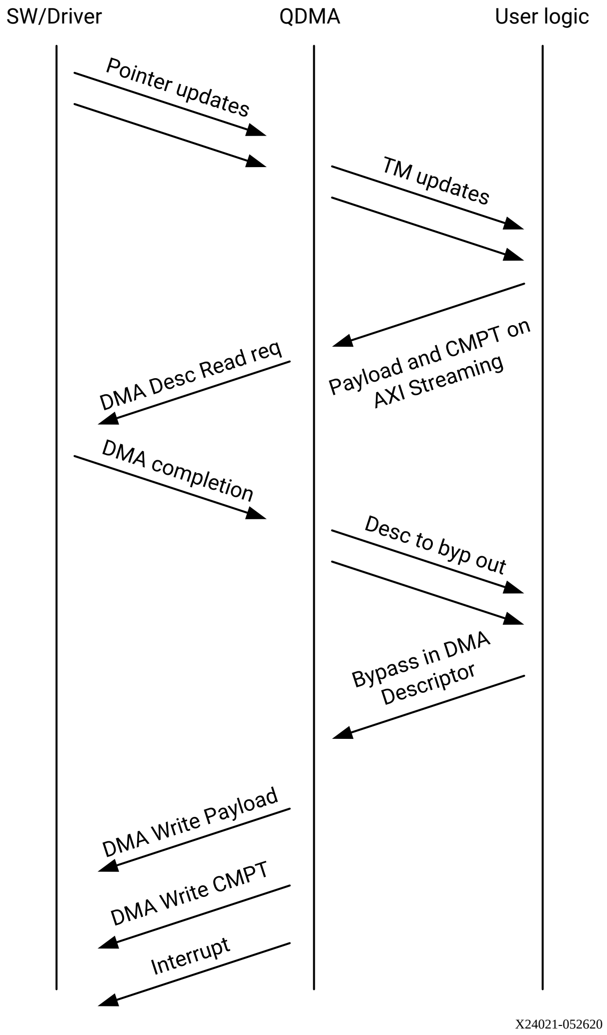
For Cache Bypass mode, the descriptor fetch engine sends the descriptors out on the C2H bypass output interface. The user application converts the descriptor and loops it back to the QDMA on the cache mode C2H bypass input interface. The prefetch engine sends the credits for the descriptors, and it keeps the order of the descriptors.

For Cache Internal mode, the descriptor fetch engine sends the descriptors to the prefetch engine. The prefetch engine sends out the credits for the descriptors and keeps the order of the descriptors. In this case, the descriptors do not go out on the C2H bypass output and do not come back on the C2H bypass input interfaces. In Cache Internal mode prefetch tag is maintained by the IP internally.

In Cache Bypass or Cache Internal mode, prefetch mode can be turned on which will prefetch the descriptor and will reduce transfer latency significantly. When prefetch mode is enabled, the user application can not send credits as input in QDMA Descriptor Credit input ports. Credits for all queues will be maintained by prefetch engine.

In cache bypass mode, prefetch tag is maintained by the IP internally. Signal `c2h_byp_out_pfch_tag[6:0]` should be looped back as an input `c2h_byp_in_st_csh_pfch_tag[6:0]`. The prefetch tag points to the cam that stores the active queues in the prefetch engine.

Figure 14: **C2H Cache Bypass Mode Flow**



***Note*:** No sequence is required between payload and completion packets.

Related Information

QDMA Descriptor Bypass Input Ports
QDMA Descriptor Bypass Output Ports

C2H Stream Packet Type

The following are some of the different C2H stream packets.

Regular Packet

The regular C2H packet has both the data packet and Completion (CMPT) packet. They are a one-to-one match.

The regular C2H data packet can be multiple beats.

- `s_axis_c2h_ctrl_qid` = C2H descriptor queue ID.
- `s_axis_c2h_ctrl_len` = length of the packet.
- `s_axis_c2h_mty` = empty byte should be set in last beat.
- `s_axis_c2h_ctrl_has_cmpt` = `1'b1`. This data packet has a corresponding CMPT packet.

The regular C2H CMPT packet is one beat.

- `s_axis_c2h_cmpt_ctrl_qid` = Completion queue ID of the packet. This can be different from the C2H descriptor QID.
- `s_axis_c2h_cmpt_ctrl_cmpt_type` = `HAS_PLD`. This completion packet has a corresponding data packet.
- `s_axis_c2h_cmpt_ctrl_wait_pld_pkt_id` = This completion packet has to wait for the data packet with this ID to be sent before the CMPT packet can be sent.

When the user application sends the data packet, it must count the packet ID for each packet. The first data packet has a packet ID of 1, and it increments for each data packets.

For the regular C2H packet, the data packet and the completion packet is a one-to-one match. Therefore, the number of data packets with `s_axis_c2h_ctrl_has_cmpt` as `1'b1` should be equal to the number of CMPT packet with `s_axis_c2h_cmpt_ctrl_cmpt_type` as `HAS_PLD`.

The QDMA has a shallow completion input FIFO of depth 2. For better performance, add FIFO for completion input as shown in the diagram below. Depth and width of the FIFO depends on the use case. Width is dependent on the largest CMPT size for the application, and depth is dependent on performance needs. For best performance for 64 Byte CMPT, a depth of 512 is recommended.

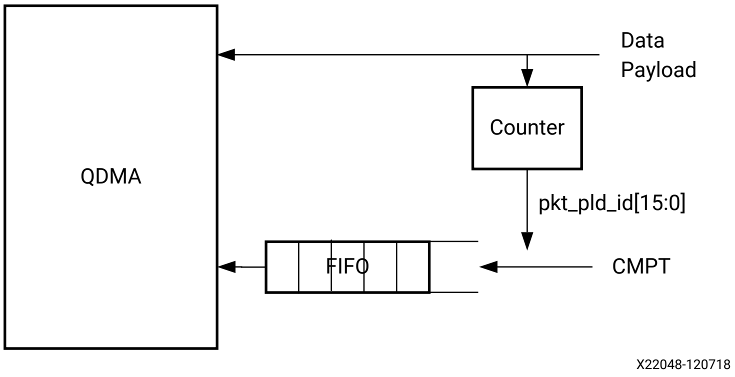
When the user application sends the data payload, it counts every packet. The first packet starts with a `pkt_pld_id` of 1. The second packet has a `pkt_pld_id` of 2, and so on. It is a 16-bits counter once the count reaches 16'hffff it wraps around to 0 and count forward.

The user application defines the CMPT type.

- If the `s_axis_c2h_cmpt_ctrl_cmpt_type` is HAS_PLD, the CMPT has a corresponding data payload. The user application must place `pkt_pld_id` of that packet in the `s_axis_c2h_cmpt_ctrl_wait_pld_pkt_id` field. The DMA will only send out this CMPT after it sends out the corresponding data payload packet.
- If the `s_axis_c2h_cmpt_ctrl_cmpt_type` is NO_PLD_NO_WAIT, the CMPT does not have any data payload, and it does not need to wait for payload. Then the DMA will send out this CMPT.

- If the s_axis_c2h_cmpt_ctrl_cmpt_type is NO_PLD_BUT_WAIT, the CMPT does not have a corresponding data payload packet. The CMPT must wait for a particular data payload packet before the CMPT is sent out. Therefore, the user application must place the pld_pkt_id of that particular data payload into the s_axis_c2h_cmpt_ctrl_wait_pld_pkt_id field. The DMA will not send out the CMPT until the data payload with that pld_pkt_id is sent out.

Figure 15: **CMPT Input FIFO**



Immediate Data Packet

The user application can have a packet that only writes to the Completion Ring without having a corresponding data packet transfer to the host. This type of packet is called immediate data packet. For the immediate data packet, the QDMA will not send the data payload, but it will write to the CMPT Queue. The immediate packet does not consume a descriptor.

For the immediate data packet, the user application only sends the CMPT packet to the DMA, and it does not send the data packet.

The following is the setting of the immediate completion packet. There is no corresponding data packet.

In some applications, the immediate completion packet does not need to wait for any data packet. But in some applications, it might still need to wait for the data payload packet. When the completion type is NO_PLD_NO_WAIT, the completion packet can be sent out without waiting for any data packet. When the completion type is NO_PLD_BUT_WAIT, the completion packet must specify the data packet ID that it needs to wait for.

- s_axis_c2h_cmpt_user_cmpt_type = NO_PLD_NO_WAIT or NO_PLD_BUT_WAIT.
- s_axis_c2h_cmpt_ctrl_wait_pld_pkt_id = Do not increment packet count.

Marker Packet

The C2H Stream Engine of the QDMA provides a way for the user application to insert a marker into the QDMA along with a C2H packet. This marker then propagates through the C2H Engine pipeline and comes out on the Queue status port interface. The marker is inserted by setting the marker bit in the C2H Stream packet. The marker response is indicated by the QDMA to the user application by setting the `qsts_out_op[7:0] = 0x0` (CMPT Marker response) bits on the Queue status ports. For a marker packet, QDMA does not send out a payload packet but still writes to the Completion Ring. Not all marker responses are generated because of a corresponding marker request. The QDMA some times generates marker responses when it encounters exceptional events. See the following section for details about when QDMA internally generates marker responses.

The primary purpose of giving the user application the ability of sending in a marker into QDMA is to determine when all the traffic prior to the marker has been flushed. This can be used in the shut down sequence in the user application. Although not a requirement, the marker can be sent by the user application with the `user_trig` bit set when sending in the marker into QDMA. This allows the QDMA to generate an interrupt and truly ensures that all traffic prior to the marker is flushed out. The QDMA Completion Engine takes the following actions when it receives a marker from the user application:

The DMA also has an option not to send completion information to completion ring during a Marker packet. Port `s_axis_c2h_cmpt_ctrl_no_wrb_marker` can be set during a marker packet. This option disables any write to completion ring when that Marker packet is generated. When this signal is set for a Maker packet, the DMA has no data or completion transfers, but will respond with maker response on `qsts_out_op[7:0]`.

- Sends the Completion that came along with the marker to the C2H Stream Completion Ring.
- Sends lower 24bits of completion data to the Queue status data port `qsts_out_data[26:3]`.
- Generates Status Descriptor if enabled (if `user_trig` was set when maker was inserted).
- Generates an Interrupt if enabled and not outstanding.
- Sends the marker response. If an Interrupt was not sent due to it being enabled but outstanding, the `retry marker_req` bit in the marker response is set to inform the user that an Interrupt could not be sent for this marker request. See the Queue status ports interface description for details of these fields.

The marker packet has both the data packet and CMPT packet. They are one-to-one match.

The following is the setting of the data packet with marker:

- 1 beat of data
- `s_axis_c2h_ctrl_marker` = `1'b1`
- `s_axis_c2h_ctrl_len` = data width (for example, 64 if data width is 512 bits)

- `s_axis_c2h_mty` = 0
- `s_axis_c2h_ctrl_has_cmpt` = `1'b1`

The following is the setting of the CMPT packet with marker:

- 1 beat of CMPT packet
- `s_axis_c2h_cmpt_ctrl_marker` = `1'b1`
- `s_axis_c2h_cmpt_ctrl_cmpt_type` = `HAS_PLD`
- `s_axis_c2h_cmpt_ctrl_wait_pld_pkt_id` = This completion packet has to wait for the data payload packet with this ID to be sent before the CMPT packet is sent.
- `s_axis_c2h_cmpt_ctrl_no_wrb_marker` = `1'b0`. If this is set to `1'b1`then no CMPT packet.

The immediate data packet and the marker packet do not consume the descriptor; instead, they write to the C2H Completion Ring. The software needs to size the C2H Completion Ring large enough to accommodate the outstanding immediate packets and the marker packets.

When a marker request is received by the DMA and the completion ring is full, the marker response will be sent out. However, because the completion ring is full, the completion entry will be dropped and the queue will be invalidated.

Zero Length Packet

The length of the data packet can be zero. On the input, the user needs to send one beat of data. The zero length packet consumes the descriptor. The QDMA will send out 1DW payload data.

The following is the setting of the zero length packet:

- 1 beat of data
- `s_axis_c2h_ctrl_len` = 0
- `s_axis_c2h_mty` = 0

***Note*:** Zero Byte packets are not supported in Internal mode and Cache bypass mode. The QDMA may hang if zero byte packets are dropped due to not available descriptor. Zero Byte Packets are supported in Simple bypass mode.

Disable Completion Packet

The user application can disable the completion for a specific packet. The QDMA provides direct memory access (DMA) to the payload, but does not write to the C2H Completion Ring. The user application only sends the data packet to the DMA, and does not send the CMPT packet.

The following is the setting of the disable completion packet:

- `s_axis_c2h_ctrl_has_cmpt` = `1'b0`

Related Information

QDMA Descriptor Bypass Output Ports

Handling Descriptors With Errors

If an error is encountered while fetching a descriptor (in pre-fetch or regular mode), the QDMA Descriptor Engine flags the descriptor with error. For a queue in internal mode, the C2H Stream Engine handles the error descriptor by not performing any PCIe or DMA activity. Instead, it waits for the error descriptor to pass through the pipeline and forces a writeback after it is done. For a queue in bypass mode, it is the responsibility of the user logic to not issue a batch of descriptors with an error descriptor. Instead, it must send just one descriptor with error input asserted on the C2H Stream bypass-in interface and set the SOP, EOP, `no_dma` signal, and `sdi` or `mrkr_req` signal to make the C2H Stream Engine send a writeback to Host.

Completion Engine

The Completion Engine writes the C2H AXI4-Stream Completion (CMPT) in the CMPT queue. The user application sends a CMPT packet and other information, such as, but not limited to, CMPT QID, and CMPT_TYPE to the QDMA. The QDMA uses this information to process the CMPT packet. The QDMA can be instructed to write the CMPT packet unchanged in the CMPT queue. Alternatively, the user application can instruct the QDMA to insert certain fields, like error and color, in the CMPT packet before writing it into the CMPT queue. Additionally, using the CMPT interface signals, the user application instructs the QDMA to order the writing of the CMPT packet in a specific way, relative to traffic on the C2H data input. Although not a requirement, a CMPT is typically used with a C2H queue. In such a case, the CMPT is used to inform the SW that a certain number of C2H descriptors have been used up by the DMA of C2H data. This allows the SW to reclaim the C2H descriptors. A CMPT can also be used without a corresponding C2H DMA operation, in which case, it is known as Immediate Data.

The user-defined portion of the CMPT packet typically needs to specify the length of the data packet transferred and whether or not descriptors were consumed as a result of the data packet transfer. Immediate and marker type packets do not consume any descriptors. The exact contents of the user-defined data are up to the user to determine.

Completion Context Structure

The completion context is used by the Completion Engine.

Table 15: **Completion Context Structure Definition**

Bit	Bit Width	Field Name	Description
[256:183]	17		Reserved. Initialize to 0.

Table 15: **Completion Context Structure Definition** *(cont'd)*

Bit	Bit Width	Field Name	Description
[182:180]	3	port_id	Port ID. The Completion Engine checks the port_id of events received at its input to the port_id configured here. If the check fails, the input is dropped, and an error is logged in the C2H_ERR_STAT register. The following are checked for port_id: 1. All events on the s_axis_c2h_cmpt interface. These include CMPTs, Immediate data, markers and VirtIO control messages. 2. CMPT CIDX pointer updates (checked only when the update is coming from the AXI side).
[179]	1		Reserved. Initialize to 0's
[178:175]	4	baddr4_low	Since the minimum alignment supported is 64B in this case, this field must be 0
[174:147]	28		Reserved. Initialize to 0's
[146]	1	dir_c2h	DMA direction is C2H. The CMPT engine can be used to manage the completion/used ring of a C2H as well as an H2C queue. 0x0: DMA direction is H2C 0x1: DMA direction is C2H
[145]	1		Reserved. Initialize to 0
[144]	1	dis_int_on_vf	Disable interrupt with VF
[143]	1	int_aggr	Interrupt Aggregration Set to configure the QID in interrupt aggregation mode
[142:132]	11	vec	Interrupt Vector Number
131	1	at	Address Translation This bit is used to determine whether the queue addresses are translated or untranslated. This information is sent to the PCIe on CMPT and Status writes. 0: Address is untranslated 1: Address is translated
130	1	ovf_chk_dis	Completion Ring Overflow Check Disable If set, then the CMPT Engine does not check whether writing a completion entry in the Completion Ring will overflow the Ring or not. The result is that QDMA invariably sends out Completions without first checking if it is going to overflow the Completion Ring and not take any actions that it normally takes when it encounters a Completion Ring overflow scenario. It is up to the software and user logic to negotiate and ensure that they do not cause a Completion Ring overflow

Table 15: **Completion Context Structure Definition** *(cont'd)*

Bit	Bit Width	Field Name	Description
[129]	1	full_upd	Full Update If reset, the all fields other than the CIDX of a Completion-CIDX-update are ignored. Only the CIDX field will be copied from the update to the context. If set, the Completion CIDX update can update the following fields in this context: • timer_ix • counter_ix • trig_mode • en_int • en_stat_desc
[128]	1	timer_running	If set, it indicates that a timer is running on this queue. This timer is for the purpose of CMPT interrupt moderation. Ideally, the software must ensure that there is no running timer on this QID before shutting the queue down. This is a field used internally by the hardware. The software must initialize it to 0 and then treat it as read-only.
[127]	1	user_trig_pend	If set, it indicates that a user logic initiated interrupt is pending to be generated. The user logic can request an interrupt through the s_axis_c2h_cmpt_ctrl_user_trig signal. This bit is set when the user logic requests an interrupt while another one is already pending on this QID. When the next Completion CIDX update is received by QDMA, this pending bit may or may not generate an interrupt depending on whether or not there are entries in the Completion ring waiting to be read. This is a field used internally by the hardware. The software must initialize it to 0 and then treat it as read-only.
[126:125]	2	err	Indicates that the Completion Context is in error. This is a field written by the hardware. The software must initialize it to 0 and then treat it as read-only. The following errors are indicated here: 0: No error. 1: A bad CIDX update from software was detected. 2: A descriptor error was detected. 3: A Completion packet was sent by the user logic when the Completion Ring was already full.
[124]	1	valid	Context is valid.
[123:108]	16	cidx	Current value of the hardware copy of the Completion Ring Consumer Index.
[107:92]	16	pidx	Completion Ring Producer Index. This is a field written by the hardware. The software must initialize it to 0 and then treat it as read-only.
[91:90]	2	desc_size	Completion Entry Size: 0: 8B 1: 16B 2: 32B 3: 64B
[89:32]	58	baddr	64B aligned base address of Completion ring – bit [63:6].

Table 15: **Completion Context Structure Definition** *(cont'd)*

Bit	Bit Width	Field Name	Description
[31:28]	4	qsize_idx	Completion ring size index. This index selects one of 16 register (offset 0x204 :0x240) which has different ring sizes.
[27]	1	color	Color bit to be used on Completion.
[26:25]	2	int_st	Interrupt State: 0: ISR 1: TRIG This is a field used internally by the hardware. The software must initialize it to 0 and then treat it as read-only. When out of reset, the hardware initializes into ISR state, and is not sensitive to trigger events. If the software needs interrupts or status writes, it must send an initial Completion CIDX update. This makes the hardware move into TRIG state and as a result it becomes sensitive to any trigger conditions.
[24:21]	4	timer_idx	Index to timer register for TIMER based trigger modes.
[20:17]	4	counter_idx	Index to counter register for COUNT based trigger modes.
[16:13]	4		Reserved. Initialize to 0
[12:5]	8	fnc_id	Function ID
[4:2]	3	trig_mode	Interrupt and Completion Status Write Trigger Mode: 0x0: Disabled 0x1: Every 0x2: reserved 0x3: User 0x4: User_Timer 0x5: reserved
[1]	1	en_int	Enable Completion interrupts.
[0]	1	en_stat_desc	Enable Completion Status writes.

Completion Status Structure

The Completion Status is located at the last location of Completion ring, that is, Completion Ring Base Address + (Size of the completion length (8,16,32) * (Completion Ring Size – 1)).

In order to make the QDMA write Completion Status to the Completion ring, Completion Status must be enabled in the Completion context. In addition to affecting Interrupts, the trigger mode defined in the Completion context also moderates the writing of Completion Statuses. Subject to Interrupt/Status moderation, a Completion Status can be written when either of the following happens:

1. A CMPT packet is written to the Completion ring.
2. A CMPT-CIDX update from the SW is received, and indicates that more Completion entries are waiting to be read.

3. The timer associated with the respective CMPT QID expires and is programmed in a timer-based trigger mode.

Table 16: **AXI4-Stream Completion Status Structure**

Bit	Bit Width	Field Name	Description
[63:37]	27		Reserved
[36:35]	2	error	Error. 0x0: No error 0x1 Bad CIDX update received 0x2: Descriptor error 0x3: CMPT ring overflow error
[34:33]	2	int_state	Interrupt State. 0: ISR 1: TRIG
[32]	1	color	Color status bit
[31:16]	16	cidx	Consumer Index (RO)
[15:0]	16	pidx	Producer Index

Completion Entry Structure

The size of a Completion (CMPT) Ring entry is 512 bits. This includes user defined data, an optional error bit, and an optional color bit. The user defined data has four size options: 8B, 16B, 32B and 64B. The bit locations of the optional error and color bits in the CMPT entry are configurable individually. This is done by specifying the locations of these fields using the Vivado® IDE IP customization options while compiling the QDMA. There are seven color bit location options and eight error bit location options. The location is specified as an offset from the LSB bit of the Completion entry.

When the user application drives a Completion packet into the QDMA, it provides a `s_axis_cmpt_ctrl_col_idx[2:0]` value and a `s_axis_cmpt_ctrl_err_idx[2:0]` value at the interface. These indices are used by the QDMA to use the correct locations of the color and error bits. For example, if `s_axis_cmpt_ctrl_col_idx[2:0] = 0` and `s_axis_cmpt_ctrl_err_idx[2:0] = 1`, then the QDMA uses the **C2H Stream Completion Color bits** position option 0 for color location, and **C2H Stream Completion Error bits** position option 1 for error location. An index of seven for color or error signals implies that the DMA will not update the corresponding color or error bits when Completion entry is updated (those fields are ignored). The C2H Stream Completions bits options are set in the PCIe DMA Tab in the Vivado® IDE.

The error and color bit location values that are used at compile time are available for the software to read from the MMIO registers. There are seven registers for this purpose, QDMA_C2H_CMPT_FORMAT (0xBC4) to QDMA_GLBL_ERR_MASK (0x24C). Each of these registers holds one color and one error bit location.

- C2H Stream Completions bits option 0 for color bit location and option 0 for error bit location are available through the QDMA_C2H_CMPT_FORMAT_0 register.
- C2H Stream Completions bits option 1 for color bit location and option 1 for error bit location are available through the QDMA_C2H_CMPT_FORMAT_1 register.
- And so on.

Table 17: **Completion Entry Structure**

Name	Size (Bits)	Index
User-defined bits for 64 Bytes settings	510-512	Depending on whether there are color and error bits present.
User-defined bits for 32 Bytes settings	254-256	Depending on whether there are color and error bits present.
User-defined bits for 16 Bytes settings	126-128	Depending on whether there are color and error bits present.
User-defined bits for 8 Bytes settings	62-64	Depending on whether there are color and error bits present.
Err		The Error bit location is defined by registers QDMA_C2H_CMPT_FORMAT_0 (0xBC4) to QDMA_C2H_CMPT_FORMAT_6 (0xBDC). These register show color bit position that is user defined during IP generation. You can index into this register based on input CMPT ports `s_axis_c2h_cmpt_ctrl_err_idx[2:0]`. You can choose not to include err bit (index value 7). In such a case, user-defined data takes up that space
Color		The Color bit location is defined by registers QDMA_C2H_CMPT_FORMAT_0 (0xBC4) to QDMA_C2H_CMPT_FORMAT_6 (0xBDC).These register show color bit position that is user defined during IP generation. You can index into this register based on input CMPT ports `s_axis_c2h_cmpt_ctrl_col_idx[2:0]`. If you do not include a color bit (index value 7), the user-defined data takes up that space.

Related Information

QDMA_CSR (0x0000)
PCIe DMA Tab

Completion Input Packet

The user application sends the CMPT packet to the QDMA.

The CMPT packet and data packet do not require a one-to-one match. For example, the immediate data packet only has the CMPT packet, and does not have the data packet. The disable completion packet only has the data packet and does not have the CMPT packet.

Each CMPT packet has a CMPT ID. It is the ID for the associated CMPT queue. Each CMPT queue has a CMPT Context. The driver sets up the mapping of the C2H descriptor queue to the CMPT queue. There also can be a CMPT queue that is not associated to a C2H queue.

The following is the CMPT packet from the user application.

Table 18: **CMPT Input Packet**

Name	Size	Index
Data	512 bits	[511:0]

The CMPT packet has four types: 8B, 16B, 32B, or 64B. It has just one pump of data with 512 bits.

Completion Status/Interrupt Moderation

The QDMA provides a means to moderate the Completion interrupts and Completion Status writes on a per queue basis. The software can select one out of five modes for each queue. The selected mode for a queue is stored in the QDMA in the Completion ring context for that queue. After a mode has been selected for a queue, the driver can always select another mode when it sends the completion ring CIDX update to the QDMA.

The Completion interrupt moderation is handled by the Completion engine. The Completion engine stores the Completion ring contexts of all the queues. It is possible to individually enable or disable the sending of interrupts and Completion Statuses for every queue and this information is present in the Completion ring context. It is worth mentioning that the modes being described here moderate not only interrupts but also Completion Status writes. Also, since interrupts and Completion Status writes can be individually enabled/disabled for each queue, these modes will work only if the interrupt/Completion Status is enabled in the Completion context for that queue.

The QDMA keeps only one interrupt outstanding per queue. This policy is enforced by QDMA even if all other conditions to send an interrupt have been met for the mode. The way the QDMA considers an interrupt serviced is by receiving a CIDX update for that queue from the driver.

The basic policy followed in all the interrupt moderation modes is that when there is no interrupt outstanding for a queue, the QDMA keeps monitoring the trigger conditions to be met for that mode. Once the conditions are met, an interrupt is sent out. While the QDMA subsystem is waiting for the interrupt to be served, it remains sensitive to interrupt conditions being met and remembers them. When the CIDX update is received, the QDMA subsystem evaluates whether the conditions are still being met. If they are still being met, another interrupt is sent out. If they are not met, no interrupt is sent out and the QDMA resumes monitoring for the conditions to be met again.

Note that the interrupt moderation modes that the QDMA subsystem provides are not necessarily precise. Thus, if the user application sends two CMPT packets with an indication to send an interrupt, it is not necessary that two interrupts will be generated. The main reason for this behavior is that when the driver is interrupted to read the Completion ring, and it is under no obligation to read exactly up to the Completion for which the interrupt was generated. Thus, the driver may not read up to the interrupting Completion, or it may even read beyond the interrupting Completion descriptor if there are valid descriptors to be read there. This behavior requires the QDMA to re-evaluate the trigger conditions every time it receives the CIDX update from the driver.

The detailed description of each mode is given below:

- **TRIGGER_EVERY:** This mode is the most aggressive in terms of interruption frequency. The idea behind this mode is to send an interrupt whenever the completion engine determines that an unread completion descriptor is present in the Completion ring.
- **TRIGGER_USER:** The QDMA provides a way to send a CMPT packet to the subsystem with an indication to send out an interrupt when the subsystem is done sending the packet to the host. This allows the user application to perform interrupt moderation when the TRIGGER_USER mode is set.
- **TRIGGER_USER_TIMER:** In this mode, the QDMA is sensitive to either of two triggers. One of these triggers is sent by the user along with the CMPT packet. The other trigger is the expiration of the timer that is associated with the CMPT queue. The period of the timer is driver programmable on a per-queue basis. The QDMA evaluates whether or not to send an interrupt when either of these triggers is detected. As explained in the preceding sections, other conditions must be satisfied in addition to the triggers for an interrupt to be sent. For more information, see Completion Timer.
- **TRIGGER_DIS:** In this mode, the QDMA does not send Completion interrupts in spite of them being enabled for a given queue. The only way that the driver can read the Completion ring in this case is when it regularly polls the ring. The driver will have to make use of the color bit feature provided in the Completion ring when this mode is set as this mode also disables the sending of any Completion Status descriptors to the Completion ring.

When a queue is programmed in TRIGGER_USER_TIMER_COUNT mode, the software can choose to not read all the Completion entries available in the Completion ring as indicated by an interrupt (or a Completion Status write). In such a case, the software can give a Completion CIDX update for the partial read. This works because the QDMA will restart the timer upon reception of the CIDX update and once the timer expires, another interrupt will be generated. This process will repeat until all the Completion entries have been read.

However, in the TRIGGER_EVERY, TRIGGER_USER and TRIGGER_USER_COUNT modes, an interrupt is sent, if at all, as a result of a Completion packet being received by the QDMA from the user logic. For every request by the user logic to send an interrupt, the QDMA sends one and only one interrupt. Thus in this case, if the software does not read all the Completion entries available to be read and the user logic does not send any more Completions requesting interrupts, the QDMA does not generate any more interrupts. This results in the residual Completions sitting in the Completion ring indefinitely. To avoid this from happening, when in TRIGGER_EVERY, TRIGGER_USER and TRIGGER_USER_COUNT mode, the software must read all the Completion entries in the Completion ring as indicated by an interrupt (or a Completion Status write).

The following are the flowcharts of different modes. These flowcharts are from the point of view of the Completion Engine. The Completion packets come in from the user logic and are written to the Completion Ring. The software (SW) update refers to the Completion Ring CIDX update sent from software to hardware.

Figure 16: **Flowchart for EVERY Mode**

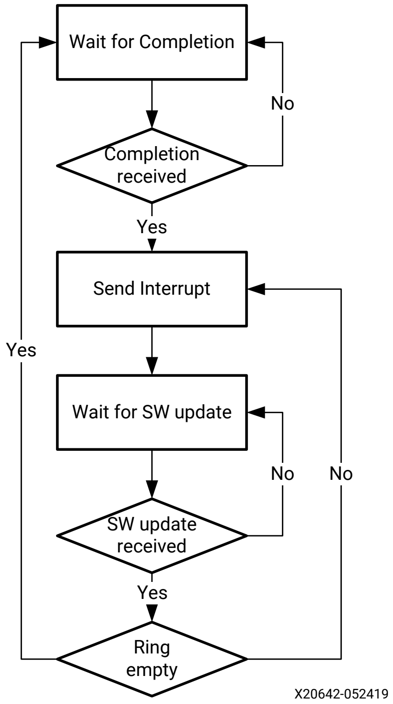


Figure 17: **Flowchart for USER Mode**

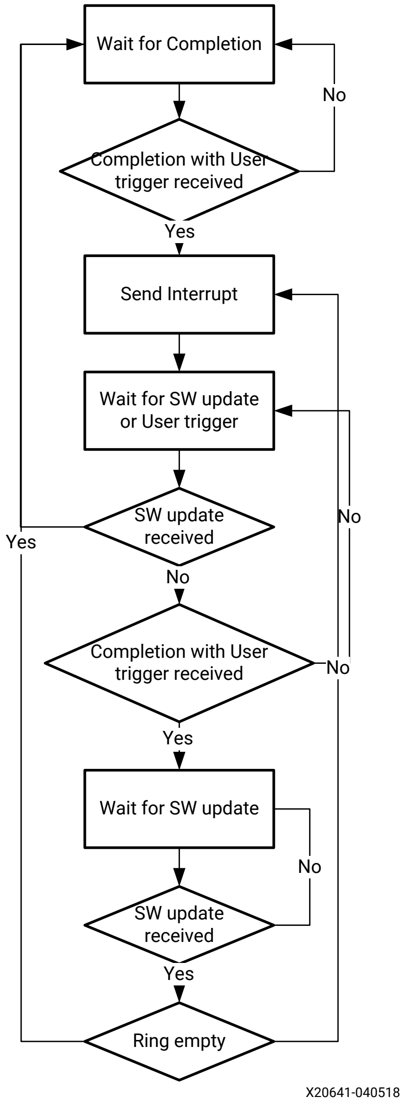


Figure 18: **Flowchart for USER_COUNT Mode**

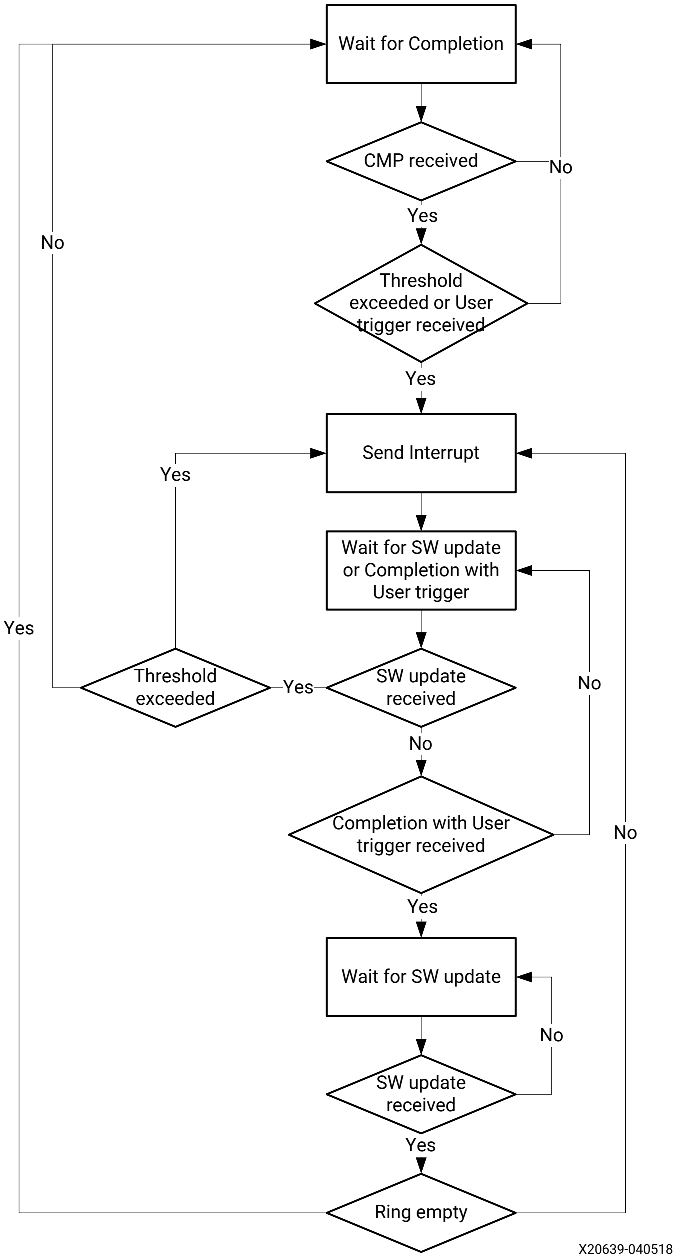


Figure 19: **Flowchart for USER_TIMER Mode**

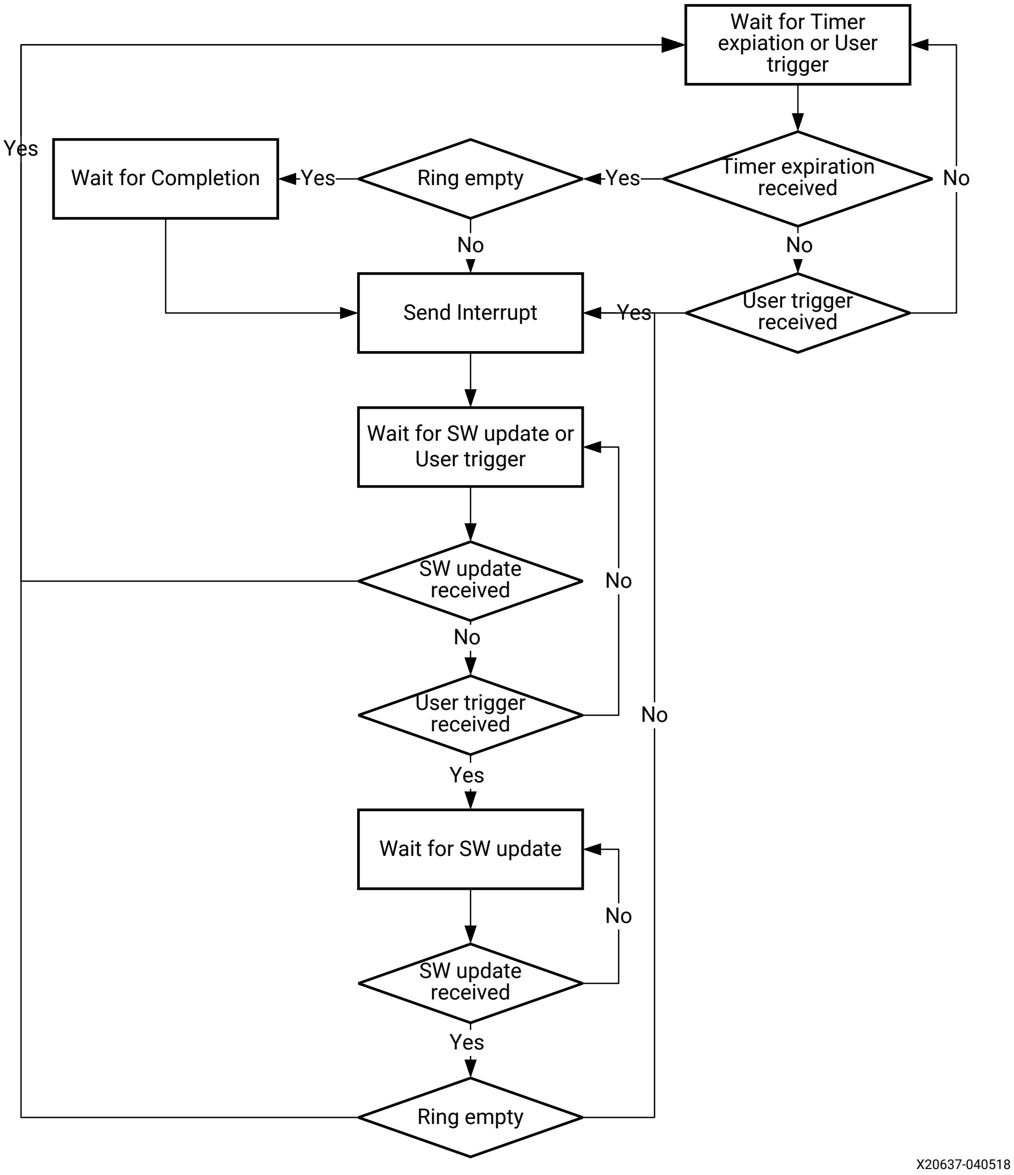
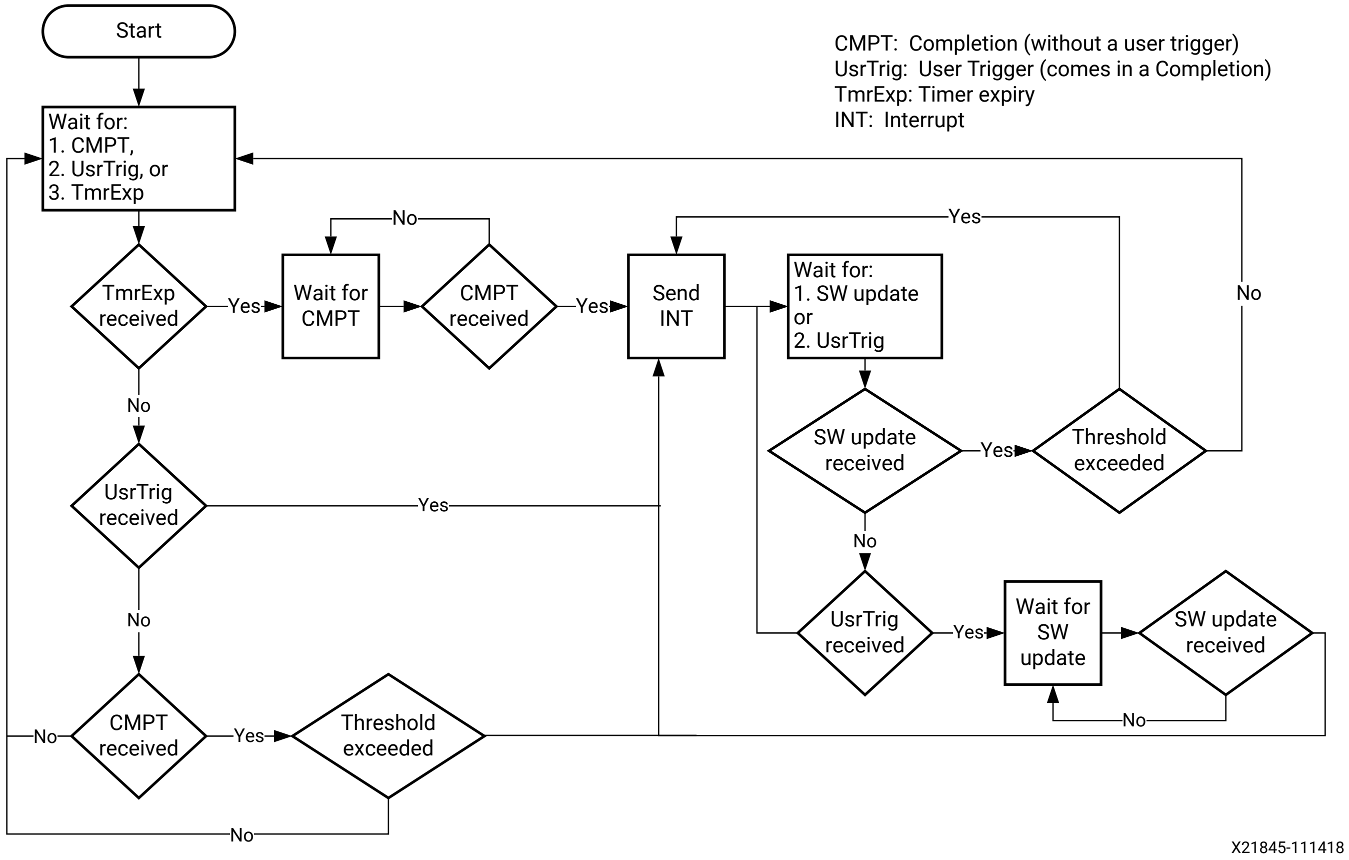


Figure 20: **Flowchart for USER_TIMER_COUNT Mode**



Completion Timer

The Completion Timer engine supports the timer trigger mode in the Completion context. It supports 2048 queues, and each queue has its own timer. When the timer expires, a timer expire signal is sent to the Completion module. If multiple timers expire at the same time, they are sent out in a round robin manner.

Reference Timer

The reference timer is based on the timer tick. The register QDMA_C2H_INT (0xB0C) defines the value of a timer tick. The 16 registers QDMA_C2H_TIMER_CNT (0xA00-0xA3c) has the timer counts based on the timer tick. The `timer_idx` in the Completion context is the index to the 16 QDMA_C2H_TIMER_CNT registers. Each queue can choose its own `timer_idx`.

Handling Exception Events

C2H Completion On Invalid Queue

When QDMA receives a Completion on a queue which has an invalid context as indicated by the Valid bit in the C2H CMPT Context, the Completion is silently dropped.

C2H Completion On A Full Ring

The maximum number of Completion entries in the Completion Ring is 2 less than the total number of entries in the Completion Ring. The C2H Completion Context has PIDX and CIDX in it. This allows the QDMA to calculate the number of Completions in the Completion Ring. When the QDMA receives a Completion on a queue that is full, QDMA takes the following actions:

- Invalidates the C2H Completion Context for that queue.
- Marks the C2H Completion Context with error.
- Drops the Completion.
- If enabled, sends a Status Descriptor marked with error.
- If enabled and not outstanding, sends an Interrupt.
- Sends a Marker Response with error.
- Logs the error in the C2H Error Status Register.

C2H Completion With Descriptor Error

When the QDMA C2H Engine encounters a Descriptor Error, the following actions are taken in the context of the C2H Completion Engine:

- Invalidates the C2H Completion Context for that queue.
- Marks the C2H Completion Context with error.
- Sends the Completion out to the Completion Ring. It is marked with an error.
- If enabled and not outstanding, sends a Status Descriptor marked with error.
- If enabled and not outstanding, sends an Interrupt. Note that the Completion Engine can only send an interrupt and/or status descriptor if not outstanding. One implication of this is that if the interrupt happens to be outstanding when the descriptor error is encountered, a queue interrupt will not be sent to the software. Despite that, the error is logged and an error interrupt is still sent, if not masked by the software
- Sends a Marker Response with error.

C2H Completion With Invalid CIDX

The C2H Completion Engine has logic to detect that the CIDX value in the CIDX update points to an empty location in the Completion Ring. When it detects such error, the C2H Completion Engine:

- Invalidates the Completion Context.
- Marks the Completion Context with error.
- Logs an error in the C2H error status register.

Port ID Mismatch

The CMPT context specifies the port over which CMPTs are expected for that CMPT queue. If the `port_id` in the incoming CMPT is not the same as the `port_id` in the CMPT context, the CMPT Engine treats the incoming CMPT as a mis-directed CMPT and drops it. It also logs an error. Note that the CMPT queue is not invalidated when a `port_id` mismatch occurs.

Bridge

The Bridge core is an interface between the AXI4 and the PCI Express integrated block. It contains the memory mapped AXI4 to AXI4-Stream Bridge, and the AXI4-Stream Enhanced Interface Block for PCIe. The memory mapped AXI4 to AXI4-Stream Bridge contains a register block and two functional half bridges, referred to as the Slave Bridge and Master Bridge.

- The slave bridge connects to the AXI4 Interconnect as a slave device to handle any issued AXI4 master read or write requests.
- The master bridge connects to the AXI4 Interconnect as a master to process the PCIe generated read or write TLPs.
- The register block contains registers used in the Bridge core for dynamically mapping the AXI4 memory mapped (MM) address range provided using the AXIBAR parameters to an address for PCIe range.

The core uses a set of interrupts to detect and flag error conditions.

Related Information

Bridge Register Space

Slave Bridge

The slave bridge provides termination of memory-mapped AXI4 transactions from an AXI4 master device (such as a processor). The slave bridge provides a way to translate addresses that are mapped within the AXI4 memory mapped address domain to the domain addresses for PCIe. Write transactions to the Slave Bridge are converted into one or more `MemWr` TLPs, depending on the configured Max Payload Size setting, which are passed to the integrated block for PCI

Express. When a remote AXI master initiates a read transaction to the slave bridge, the read address and qualifiers are captured and a MemRd request TLP is passed to the core and a completion timeout timer is started. Completions received through the core are correlated with pending read requests and read data is returned to the AXI4 master. The slave bridge can support up to 32 AXI4 write requests, and 32 AXI4 read requests.

***Note*:** If slave reads and writes are valid, IP prioritizes reads over writes. You are recommended to have proper arbitration (leave some gaps between reads so writes can pass through).

BDF Table

Address translations for AXI address is done based on BDF table programming (0x2420 to 0x2434). These BDF table entries can be programmed through the AXI4-Lite Slave CSR interface, s_axil_csr_*. There are 8 windows provided similar to 8 BARs on PCIe bus. Each entry in BDF table programming represents one window. If a user needs 2 windows then 2 entrees needs to be programmed and so on.

There are some restrictions in programming BDF table.

1. All PCIe slave bridge data transfers must be quiesced before programming the BDF table.
2. There are six registers for each BDF table entry. All six registers must be programmed to make a valid entry. Even if some registers have 0s, you need to program 0s in those registers.
3. All the six registers need to be programmed in an order for an entry to be valid. Order is listed below.
   a. 0x2420
   b. 0x2424
   c. 0x2428
   d. 0x242C
   e. 0x2430
   f. 0x2434

BDF table entree start address = 0x2420 + (0x20 * i), where i = table entree number.

Protection

Specifying protection levels for different windows within a BAR is facilitated by AXI4 prot field via Trustzones. Any access from PMC will have a*prot[1]=0 and hence will get full access.

For the BDF space the protection domain ID itself is stored in the BDF table. When a request comes in with a*rpot[1]=0, it will be allowed full access. Requests with a*prot[1]=1 will only be allowed to access BDF entries that have lower protection level.

The following table describes this behavior:

Table 19: **AXI BAR Protection Levels**

Access Type	BDF Table Value (prot[2:0])	Value in a*prot[2:0] (AXI Interface)	Action
Secure access	3'hXXX	3'hX0X (bit 1=0)	Allow
Non-secure access to secure entry	3'hX0X	3'hX1X (bit 1=1)	Do not allow
Non-secure access to less secure entry	3'hX1X	3'hX1X (bit 1=1)	Allow if bits [2] and [0] match between a*prot and BDF entry

Address Translation

Address translation can be turned off by selecting the No Address Translation option during IP configuration. When this option is selected, one full 64-bit BAR space is given for slave data transfer. You must set up any address translation if needed. If No Address Translation is *not* selected DMA will do address translation.

One 64 bit BAR space is divided into 8 (called window) and 8 window space is available for address translation. Address translation for Slave Bridge transfer are done in two steps.

1. Address translation for upper bits are programmed in GUI configuration. In the IP integrator canvas, address translation for upper bits should be edited in the Address Editor tab.
2. Address translation for lower bits are programmed in BDF table.

Slave Address Translation Examples

Example 1: BAR Size of 64 KB, with 1 Window Size 4 KB

Window 0: 4 KB with address translation of 0xF for bits [15:12].

1. Selections in Vivado IP configuration in the AXI BARs tab are as follows:
   - AXI BAR size 64K: **0xFFFF bits [15:0]**
   - Address translation for bits[63:16] can be set in GUI. In this example [63:16] = 0x0
   - Set Aperture Base Address: **0x0000_0000_0000_0000**
   - Set Aperture High Address: **0x0000_0000_0000_FFFF**
2. The BDF table programming:
   - Program 1 entries for 1 window
   - Window Size = AXI BAR size/8 = 64K / 8 = 0x1FFF = 8 KB (13 bits). Each window max size is 8 KB.
   - In this example for window size of 4K, 0x1 is programmed at 0x2430 bits [25:0].
   - Address translation for bits [15:13] are programmed at 0x2420 and 0x2424.
   - In this example, address translation for bits [15:13] are set to 0x7.

Table 20: **BDF Table Programming**

Program Value	Registers
0x0000_E000	Address translation value Low
0x0	Address translation value High
0x0	PASID/ Reserved
0x0	[11:0]: Function Number
0xC0000001	[31:30] Read/Write Access permission [29] : R0 access Error [28:26] Protection ID [25:0] Window Size ([25:0]*4K = actual size of the window)
0x0	reserved

For this example Slave address 0x0000_0000_0000_0100 will be address translated to 0x0000_0000_0000_E100.

Example 2: BAR Size of 64 KB, with 1 Window 8 KB

Window 0: 8 KB with address translation of 0x6 ('b110) for bits [15:13].

1. Selections in Vivado IP configuration in the AXI BARs tab are as follows:
   - AXI BAR size 64K: **0xFFFF bits [15:0]**
   - Address translation for bits [63:16] can be done in GUI. In this example [63:16] = 0x0
   - Set Aperture Base Address: **0x0000_0000_0000_0000**
   - Set Aperture High Address: **0x0000_0000_0000_FFFF**
2. The BDF table programming:
   - Program 1 entries for 1 window.
   - Window Size = AXI BAR size/8 = 64K / 8 = 0x1FFF = 8 KB (13 bits). Each window max size is 8 KB.
   - In this example for window size of 8K, 0x2 is programmed at 0x2430 bits [25:0].
   - Address translation for bits [15:13] are programmed at 0x2420 and 0x2424.
   - In this example, address translation for bits [15:13] are set to 0x6 ('b110).

Table 21: **BDF Table Programming**

Offset	Program Value	Registers
0x2420	0x0000_C000	Address translation value Low
0x2424	0x0	Address translation value High
0x2428	0x0	PASID/ Reserved
0x242C	0x0	[11:0]: Function Number

Table 21: **BDF Table Programming** *(cont'd)*

Offset	Program Value	Registers
0x2430	0xC0000002	[31:30] Read/Write Access permission [29] : R0 access Error [28:26] Protection ID [25:0] Window Size ([25:0]*4K = actual size of the window)
0x2434	0x0	reserved

For this example, the Slave address 0x0000_0000_0000_0100 will be address translated to 0x0000_0000_0000_C100.

Example 3: BAR Size of 32 GB, and 4 Windows of Various Sizes

- Window 0: 4 KB with address translation of 0xAAAAA for bits [31:12].
- Window 1: 4 GB with no address translation on window.
- Window 2: 64 KB with address translation of 0xBBBB for bits [31:16].
- Window 3: 1 GB with address translation of 11111 for bits [34:30].

1. Selections in Vivado IP configuration in the AXI BARs tab are as follows:
    - AXI BAR size 32G: **0x7_FFFF_FFFF bits [34:0]**.
    - Address translation for bits [63:35] can be programmed in GUI. In this example [63:36] = 0xAB.
    - Set Aperture Base Address: **0x0000_0AB0_0000_0000**.
    - Set Aperture High Address: **0x0000_0AB7_FFFF_FFFF**.
2. The BDF table programming:
    - Window Size = AXI BAR size/8 = 32 GB / 8 = 0xFFFF_FFFF = 4 GB (32 bits). Each window max size is 4 GB.
    - Program 4 entries for 4 windows:
        - BDF entry 0 table starts at 0x2420.
        - BDF entry 1 table starts at 0x2440.
        - BDF entry 2 table starts at 0x2460.
        - BDF entry 3 table starts at 0x2480.
    - Window 0 size 4 KB.
        - Program 0x1 to 0x2430 bits [25:0].
        - Address translation for bits [34:32] are programmed at 0x2420 and 0x2424.

  - Program 0x0000_0000 to 0x2420.
  - Program 0x0000_0007 to 0x2424
- Window 1 size 4 GB.
  - Program 0x10_0000 to 0x2450 bits [25:0].
  - No address translation because all address bits will be used by the window.
  - Program 0x0000_0000 to 0x2440.
  - Program 0x0000_0000 to 0x2444
- Window 2 size 64 KB.
  - Program 0x10 to 0x2470 bits [25:0].
  - Address translation for bits [34:32] are programmed at 0x2460 and 0x2464.
  - Program 0x0000_0000 to 0x2460
  - Program 0x0000_00005 to 0x2464
- Window 3 size 1 GB.
  - Program 0x4_0000 to 0x2490 bits [25:0].
  - Address translation for bits [34:30] are programmed at 0x2480 and 0x2484.
  - Program 0x0000_0000 to 0x2480.
  - Program 0x0000_0003 to 0x2484

Table 22: **BDF Table Programming Entry 0**

Offset	Program Value	Registers
0x2420	0x0000_0000	Address translation value Low
0x2424	0x7	Address translation value High
0x2428	0x0	PASID/ Reserved
0x242C	0x0	[11:0]: Function Number
0x2430	0xC0000001	[31:30] Read/Write Access permission [29] : R0 access Error [28:26] Protection ID [25:0] Window Size ([25:0]*4K = actual size of the window)
0x2434	0x0	reserved

Table 23: **BDF Table Programming Entry 1**

Offset	Program Value	Registers
0x2440	0x0000	Address translation value Low

Table 23: **BDF Table Programming Entry 1** *(cont'd)*

Offset	Program Value	Registers
0x2444	0x0	Address translation value High
0x2448	0x0	PASID/ Reserved
0x244C	0x0	[11:0]: Function Number
0x2450	0xC010_0000	[31:30] Read/Write Access permission [29] : R0 access Error [28:26] Protection ID [25:0] Window Size ([25:0]*4K = actual size of the window)
0x2454	0x0	reserved

Table 24: **BDF Table Programming Entry 2**

Offset	Program Value	Registers
0x2460	0x_0000	Address translation value Low
0x2464	0x05	Address translation value High
0x2468	0x0	PASID/ Reserved
0x246C	0x0	[11:0]: Function Number
0x2470	0xC000_00010	[31:30] Read/Write Access permission [29] : R0 access Error [28:26] Protection ID [25:0] Window Size ([25:0]*4K = actual size of the window)
0x2474	0x0	reserved

Table 25: **BDF Table Programming Entry 3**

Offset	Program Value	Registers
0x2480	0x0000_0000	Address translation value Low
0x2484	0x3	Address translation value High
0x2488	0x0	PASID/ Reserved
0x248C	0x0	[11:0]: Function Number
0x2490	0xC004_0000	[31:30] Read/Write Access permission [29] : R0 access Error [28:26] Protection ID [25:0] Window Size ([25:0]*4K = actual size of the window)
0x2494	0x0	reserved

For this example

the Slave address 0x0000_0000_0000_0100 will be address translated to 0x0000_0AB7_0000_0100.

the Slave address 0x0000_0001_0000_0100 will be address translated to 0x0000_0AB0_0000_0100.

the Slave address 0x0000_0002_0000_0100 will be address translated to 0x0000_0AB5_0000_0100.

the Slave address 0x0000_0003_0000_0100 will be address translated to 0x0000_0AB3_0000_0100.

The slave bridge does not support narrow burst AXI transfers. To avoid narrow burst transfers, connect the AXI smart-connect module which will convert narrow burst to full burst AXI transfers.

Master Bridge

The master bridge processes both PCIe `MemWr` and `MemRd` request TLPs received from the integrated block for PCI Express and provides a means to translate addresses that are mapped within the address for PCIe domain to the memory mapped AXI4 address domain. Each PCIe `MemWr` request TLP header is used to create an address and qualifiers for the memory mapped AXI4 bus and the associated write data is passed to the addressed memory mapped AXI4 Slave. The Master Bridge can support up to 32 active PCIe `MemWr` request TLPs. PCIe `MemWr` request TLPs support is as follows:

Each PCIe `MemRd` request TLP header is used to create an address and qualifiers for the memory mapped AXI4 bus. Read data is collected from the addressed memory mapped AXI4 bridge slave and used to generate completion TLPs which are then passed to the integrated block for PCI Express. The Master Bridge in AXI Bridge mode can support up to 32 active PCIe `MemRd` request TLPs with pending completions for improved AXI4 pipe-lining performance.

Interrupts

The QDMA supports up to 2K total MSI-X vectors. A single MSI-X vector can be used to support multiple queues. Each function can support up to 8 vectors (8 *256 function = 2K vectors). When SR-IOV functions are not enabled, each PF function can support up to 32 vectors. Contact Xilinx Support if more than 8 vectors are needed for each function.

The QDMA supports Interrupt Aggregation. Each vector has an associated Interrupt Aggregation Ring. The QID and status of queues requiring service are written into the Interrupt Aggregation Ring. When a PCIe® MSI-X interrupt is received by the Host, the software reads the Interrupt Aggregation Ring to determine which queue needs service. Mapping of queues to vectors is programmable vector number provided in the queue context. It supports MSI-X interrupt modes for SR-IOV and non-SR-IOV.

Asynchronous and Queue Based Interrupts

The QDMA supports both asynchronous interrupts and queue-based interrupts.

The asynchronous interrupts are used for capturing events that are not synchronous to any DMA operations, namely, errors, status, and debug conditions.

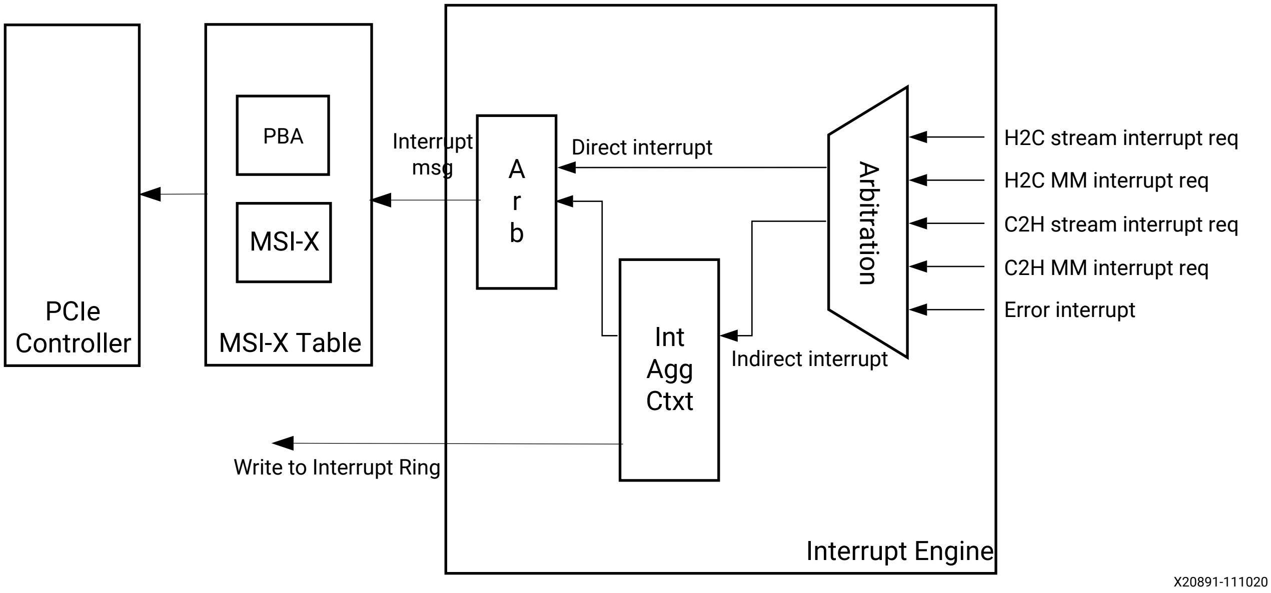
Interrupts are broadcast to all PFs, and maintain status for each PF in a queue based scheme. The queue based interrupts include the interrupts from the H2C MM, H2C stream, C2H MM, and C2H stream.

Interrupt Engine

The Interrupt Engine handles the queue based interrupts and the error interrupt.

The following figure shows the Interrupt Engine block diagram.

Figure 21: **Interrupt Engine Block Diagram**



The Interrupt Engine gets the interrupts from H2C MM, H2C stream, C2H MM, C2H stream, or error interrupt.

It handles the interrupts in two ways: direct interrupt or indirect interrupt. The interrupt sources has the information that shows if it is direct interrupt or indirect interrupt. It also has the information of the vector. If it is direct interrupt, the vector is the interrupt vector that is used to generate the PCIe MSI-X message (the interrupt vector `indix` of the MSIX table). If it is indirect interrupt, the vector is the ring index of the Interrupt Aggregation Ring. The interrupt source gets the information of interrupt type and vector from the Descriptor Software Context, the Completion Context, or the error interrupt register.

Direct Interrupt

For direct interrupt, the Interrupt Engine gets the interrupt vector from the source, and it then sends out the PCIe MSI-X message directly.

Interrupt Aggregation Ring

For the indirect interrupt, it does interrupt aggregation. The following are some restrictions for the interrupt aggregation.

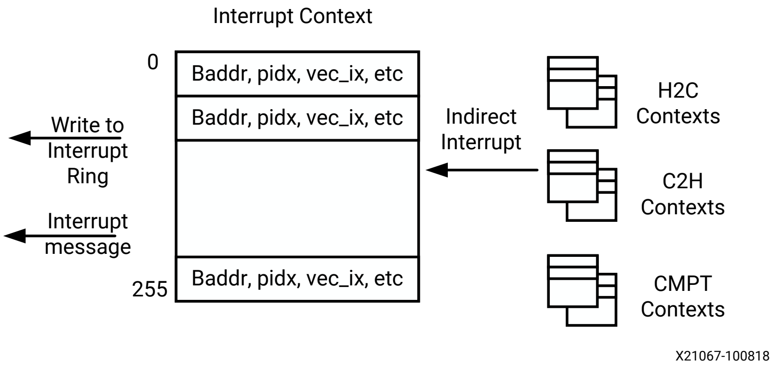
- Each Interrupt Aggregation Ring can only be associated with one function. But multiple rings can be associated with the same function.
- The interrupt engine supports up to three interrupts from the same source, until the software services the interrupts.

The Interrupt Engine processes the indirect interrupt with the following steps.

- Interrupt source provides the index to which interrupt ring it belongs too.
- Reads interrupt context for that queue.
- Writes to the Interrupt Aggregation Ring.
- Sends out the PCIe MSI-X message.

This following figure shows the indirect interrupt block diagram.

Figure 22: **Indirect Interrupt**



The Interrupt Context includes the information of the Interrupt Aggregation Ring. It has 256 entries to support up to 256 Interrupt Aggregation Rings.

Color bit is added so software does not read more entries that what it should read. When the software allocates the memory space for the Interrupt Aggregation Ring, the `coal_color` starts with `1'b0`. The software needs to initialize the color bit of the Interrupt Context to be `1'b1`. When the hardware completes the entire ring and flips to first entry in the next pass, it also flips the color value to 0 and starts writing 0 in color bit space. The software does the same after it completes the last entry with a color value 1, and goes to the first entry in the second pass and expects a color value 0. If the software does not see a color value 0,which indicates an old entry, it waits for new entry with a color value 0.

The software reads the Interrupt Aggregation Ring to get the `Qid`, and the `int_type` (H2C or C2H). From the `Qid`, the software can identify whether the queue is stream or MM.

The `stat_desc` in the Interrupt Aggregation Ring is the status descriptor from the Interrupt source. When the status descriptor is disabled, the software can get the status descriptor information from the Interrupt Aggregation Ring.

There can be two cases:

- The interrupt source is C2H stream. Then it is the status descriptor of the C2H Completion Ring. The software can read the `pidx` of the C2H Completion Ring.
- The interrupt source is others (H2C stream, H2C MM, C2H MM). Then it is the status descriptor of that source. The software can read the `cidx`.

Finally, the Interrupt Engine sends out the PCIe MSI-X message using the interrupt vector from the Interrupt Context. When there is an interrupt from any source, the interrupt engine updates PIDX and check for `int_st` of that interrupt context. If `int_st` is 0 (WAITING_TRIGGER) then the interrupt engine will send a interrupt. If `int_st` is 1 (ISR_RUNNING), the interrupt engine will not send interrupt. If the interrupt engine sends interrupt it will update `int_sts` to 1 and once software updated CIDX and CIDX matches PIDX `int_sts` will be cleared. The process is explained below.

When the PCIe MSI-X interrupt is received by the Host, the software reads the Interrupt Aggregation Ring to determine which queue needs service. After the software reads the ring, it will do a dynamic pointer update for the software CIDX to indicate the cumulative pointer that the software reads to. The software does the dynamic pointer update using the register QDMA_DMAP_SEL_INT_CIDX[2048] (0x18000). If the software CIDX is equal to the PIDX, this will trigger a write to the Interrupt Context to clear `int_st`on the interrupt state of that queue. This is to indicate the QDMA that the software already reads all of the entries in the Interrupt Aggregation Ring. If the software CIDX is not equal to the PIDX, the interrupt engine will send out another PCIe MSI-X message. Therefore, the software can read the Interrupt Aggregation Ring again. After that, the software can do a pointer update of the interrupt source ring. For example, if it is C2H stream interrupt, the software will update pointer of the interrupt source ring, which is the C2H Completion Ring.

These are the steps for the software:

1. After the software gets the PCIe MSI-X message, it reads the Interrupt Aggregation Ring entries.
2. The software uses the `coal_color` bit to identify the written entries. Each entry has `Qid` and `Int_type` (H2C or C2H). From the `Qid` and `Int_type`, the software can check if it is stream or MM. This points to a corresponding source ring. For example, if it is C2H stream, the source ring is the C2H Completion Ring. The software can then read the source ring to get information, and do a dynamic pointer update of the source ring after that.
3. After the software finishes reading of all written entries in the Interrupt Aggregation Ring, it does one dynamic pointer update of the software `cidx` using the register QDMA_DMAP_SEL_INT_CIDX[2048] (0x18000). This communicates to the hardware of the Interrupt Aggregation Ring pointer used by the software.

   If the software `cidx` is not equal to the `pidx`, the hardware will send out another PCIe MSI-X message, so that the software can read the Interrupt Aggregation Ring again.

When the software does the dynamic pointer update for the Interrupt Aggregation Ring using the register QDMA_DMAP_SEL_INT_CIDX[2048] (0x18000), it sends the ring index of the Interrupt Aggregation Ring.

The following diagram shows the indirect interrupt flow. The Interrupt module gets the interrupt requests. It first writes to the Interrupt Aggregation Ring. Then it waits for the write completions. After that, it sends out the PCIe MSI-X message. The interrupt requests can keep on coming, and the Interrupt module keeps on processing them. In the meantime, the software reads the Interrupt Aggregation Ring, and it does the dynamic pointer update. If the software CIDX is not equal to the PIDX, it will send out another PCIe MSI-X message.

Interrupt Context Structure

The following is the Interrupt Context Structure (0x8).

Table 26: **Interrupt Context Structure (0x8)**

Signal	Bit	Owner	Description
rsvd	[255:126]	Driver	Reserved. Initialize to 0s
func	[125:114]	Driver	Function number
rsvd	[113:83]	Driver	Reserved. Initialize to 0s
at	[82]	Driver	1'b0: un-translated address 1'b1: translated address
pidx	[81:70]	DMA	Producer Index, updated by DMA IP.

Table 26: **Interrupt Context Structure (0x8)** *(cont'd)*

Signal	Bit	Owner	Description
page_size	[69:67]	Driver	Interrupt Aggregation Ring size: 0: 4 KB 1: 8 KB 2: 12 KB 3: 16 KB 4: 20 KB 5: 24 KB 6: 28 KB 7: 32 KB
baddr_4k	[66:15]	Drive	Base address of Interrupt Aggregation Ring – bit [63:12]
color	[14]	DMA	Color bit
int_st	[13]	DMA	Interrupt State: 0: WAIT_TRIGGER 1: ISR_RUNNING
Rsvd	[12]	NA	Reserved
vec	[11:1]	Driver	Interrupt vector index in msix table
valid	[0]	Driver	Valid

The software needs to size the Interrupt Aggregation Ring appropriately. Each source can send up to three messages to the ring. Therefore, the size of the ring needs satisfy the following formula.

Number of entry ≥ 3 x number of queues

The Interrupt Context is programmed by the context access. The QDMA_IND_CTXT_CMD.Qid has the ring index, which is from the interrupt source. The operation of MDMA_CTXT_CMD_CLR can clear all of the bits in the Interrupt Context. The MDMA_CTXT_CMD_INV can clear the valid bit.

- Context access through QDMA_TRQ_SEL_IND:
  - QDMA_IND_CTXT_CMD.Qid = Ring index
  - QDMA_IND_CTXT_CMD.Sel = MDMA_CTXT_SEL_INT_COAL (0x8)
  - QDMA_IND_CTXT_CMD.cmd.Op =

    MDMA_CTXT_CMD_WR

    MDMA_CTXT_CMD_RD

    MDMA_CTXT_CMD_CLR

    MDMA_CTXT_CMD_INV

After the interrupt engine looks up the Interrupt Context, the interrupt engine writes to the Interrupt Aggregation Ring. The interrupt engine also updates the Interrupt Context with the new PIDX, color, and the interrupt state.

Interrupt Aggregation Entry

This is the Interrupt Aggregation Ring entry structure. It has 8B data.

Table 27: **Interrupt Aggregation Ring Entry Structure**

Signal	Bit	Owner	Description
Coal_color	[63]	DMA	The color bit of the Interrupt Aggregation Ring. This bit inverts every time pidx wraps around on the Interrupt Aggregation Ring.
Qid	[62:39]	DMA	This is from Interrupt source. Queue ID.
Int_type	[38:38]	DMA	0: H2C 1: C2H
Rsvd	[37:37]	DMA	Reserved
Stat_desc	[36:0]	DMA	This is the status descriptor of the Interrupt source.

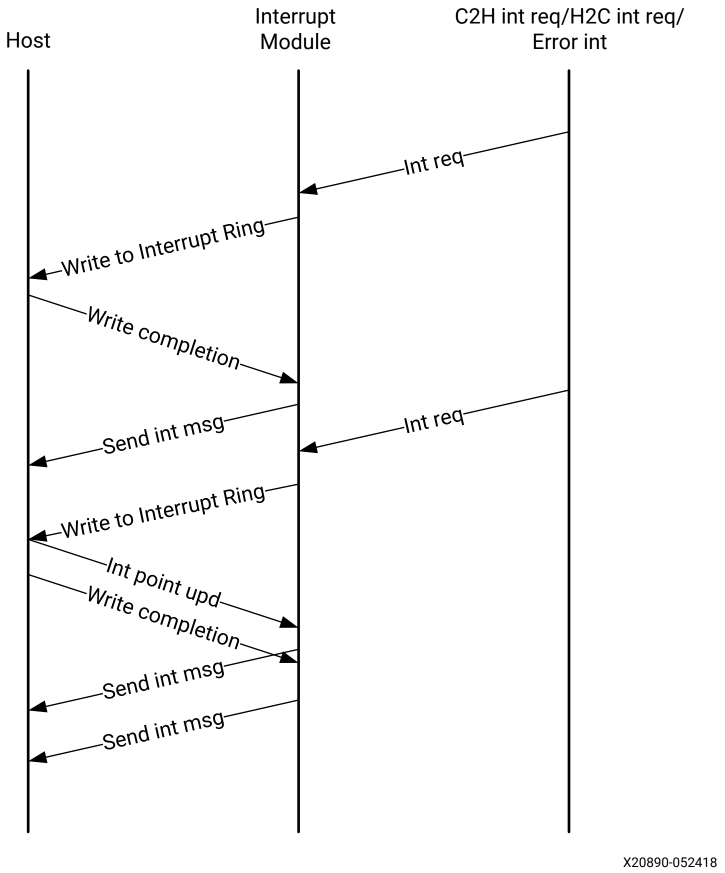
The following is the information in the `stat_desc`.

Table 28: **stat_desc Information**

Signal	Bit	Owner	Description
Error	[36:35]	DMA	This is from interrupt source: c2h_err[1:0], or h2c_err[1:0].
Int_st	[34:33]	DMA	This is from Interrupt source. Interrupt state. 0: WRB_INT_ISR 1: WRB_INT_TRIG 2: WRB_INT_ARMED
Color	[32:32]	DMA	This is from Interrupt source. This bit inverts every time pidx wraps around and this field gets copied to color field of descriptor.
Cidx	[31:16]	DMA	This is from Interrupt source. Cumulative consumed pointer.
Pidx	[15:0]	DMA	This is from Interrupt source. Cumulative pointer of total interrupt Aggregation Ring entry written.

Interrupt Flow

Figure 23: **Interrupt Flow**



Error Interrupt

There are Leaf Error Aggregators in different places. They log the errors and propagate the errors to the Central Error Aggregator. Each Leaf Error Aggregator has an error status register and an error mask register. The error mask is enable mask. Irrespective of the enable mask value, the error status register always logs the errors. Only when the error mask is enabled, the Leaf Error Aggregator will propagate the error to the Central Error Aggregator.

The Central Error Aggregator aggregates all of the errors together. When any error occurs, it can generate an Error Interrupt if the `err_int_arm` bit is set in the error interrupt register QDMA_GLBL_ERR_INT (0B04). The `err_int_arm` bit is set by the software and cleared by the hardware when the Error Interrupt is taken by the Interrupt Engine. The Error Interrupt is for all of the errors including the H2C errors and C2H errors. The Software must set this `err_int_arm` bit to generate interrupt again.

The Error Interrupt supports the direct interrupt only. Register QDMA_GLBL_ERR_INT bit[23], `en_coal` must always be programmed to 0 (direct interrupt).

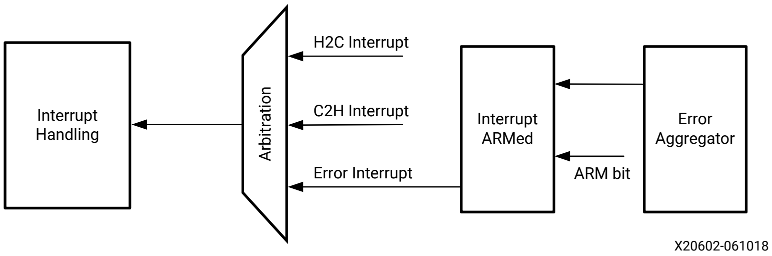
The Error Interrupt gets the vector from the error interrupt register QDMA_GLBL_ERR_INT. For the direct interrupt, the vector is the interrupt vector index of the MSI-X table.

Here are the processes of the Error Interrupt.

1. Reads the Error Interrupt register QDMA_C2H_GLBL_INT (0B04) to get function and vector numbers.
2. Sends out the PCIe MSI-X message.

The following figure shows the error interrupt register block diagram.

Figure 24: **Error Interrupt Handling**



Legacy Interrupt

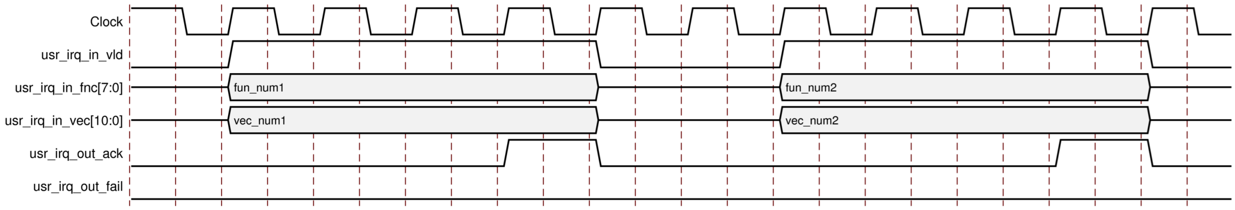
The QDMA supports the legacy interrupt for physical function, and it is expected that the single queue will be associated with interrupt.

To enable the legacy interrupt, the software needs to set the `en_lgcy_intr` bit in the register QDMA_GLBL_GLBL_INTERRUPT_CFG (0x2C4). When `en_lgcy_intr` is set, the QDMA will not send out MSI-X interrupt.

When the legacy interrupt wire INTA, INTB, INTC, or INTD is asserted, the QDMA hardware sets the `lgcy_intr_pending` bit in the QDMA_GLBL_GLBL_INTERRUPT_CFG (0x2C4) register. When the software receives the legacy interrupt, it needs to clear the `lgcy_intr_pending` bit. The hardware will keep the legacy interrupt wire asserted until the software clears the `lgcy_intr_pending` bit.

User Interrupt

Figure 25: **Interrupt**



Queue Management

Function Map Table

The Function Map Table is used to allocate queues to each function. The index into the RAM is the function number. Each entry contains the base number of the physical QID and the number of queues allocated to the function. It provides a function based, queue access protection mechanism by translating and checking accesses to logical queues (through QDMA_TRQ_SEL_QUEUE_PF and QDMA_TRQ_SEL_QUEUE_VF address space) to their physical queues. Direct register accesses to queue space beyond what is allocated to the function in the table will be canceled and an error will be logged.

Function map can be accessed through the indirect context register space QMDA_IND_CTXT_CMD registers, with QDMA_IND_CTXT_CMD.sel = 0xC. When accessed through indirect context register space, the context structure is defined by the Function Map Context Structure table.

1. Program Function Map Context structure in QDMA_IND_CTXT_DATA (0x844 - 0x820) registers as listed in the below table.
2. Program QDMA_IND_CTXT_CMD registers
    a. [19:7] : function number
    b. [6:5] : 2'h1 (write a context data)
    c. [4:1] : 4'hC (FMAP)

For more information on QDMA_IND_CTXT_CMD (0x844), *Versal ACAP Register Reference* (AM012).

Because these spaces exist only in the PF address map, only a physical function can modify this table.

Table 29: **Function Map Context Structure (0xC)**

Bits	Bit Width	Field Name	Description
[255:44]			Reserved. Set to 0.
[43:32]	12	Qid_max	Maximum number of queues this function has.
[31:11]			Reserved. Set to 0.
[10:0]	11	Qid_base	The base queue ID for the function.

Context Programming

- Program all mask registers to 1. They are QDMA_IND_CTXT_MASK_0 (0x824), to QDMA_IND_CTXT_MASK_7 (0x840).

- Program context values on to the following registers: QDMA_IND_CTXT_DATA_0 (0x804) to QDMA_IND_CTXT_DATA_7 (0x820).
- A Host Profile table context needs to be programmed before any context settings QDMA_CTXT_SELC_HOST_PROFILE. Select 0xA in QDMA_IND_CTXT_CMD (0x844), and write all data field to 0s and program context. All other values are reserved.
- Refer to 'Software Descriptor Context Structure', 'C2H Prefetch Context Structure' and 'C2H Prefetch Context Structure' to program the context data registers.
- Program any context to corresponding Queue in the following context command register: QDMA_IND_CTXT_CMD (0x844).

Note:

- Qid is given in `bits [17:7]`.
- Opcode `bits [6:5]` selects what operations must be done.
  - QDMA_CTXT_CLR : All content of context is zeroed out. Qinv will be sent out on tm_dsc_sts
  - QDMA_CTXT_WR : Write context
  - QDMA_CTXT_RD : Read context
  - QDMA_CTXT_INV : Qen in set to zero and other context values are intact. Qinv will be sent out on tm_dsc_sts and unused credits will be sent out.
- The context that is accessed is given in `bits [4:1]`.
- Context programing write/read does not occur when bit [0] is set.

Related Information

QDMA_CSR (0x0000)

Queue Setup

- Clear Descriptor Software Context.
- Clear Descriptor Hardware Context.
- Clear Descriptor Credit Context.
- Set-up Descriptor Software Context.
- Clear Prefetch Context.
- Clear Completion Context.

- Set-up Completion Context.
  - If interrupts/status writes are desired (enabled in the Completion Context), an initial Completion CIDX update is required to send the hardware into a state where it is sensitive to trigger conditions. This initial CIDX update is required, because when out of reset, the hardware initializes into an unarmed state.
- Set-up Prefetch Context.

Queue Teardown

Queue Tear-down (C2H Stream):

- Send Marker packet to drain the pipeline.
- Wait for Marker completion.
- Invalidate/Clear Descriptor Software Context.
- Invalidate/Clear Prefetch Context.
- Invalidate/Clear Completion Context.
- Invalidate Timer Context (clear cmd is not supported).

Queue Tear-down (H2C Stream & MM):

- Invalidate/Clear Descriptor Software Context.

Virtualization

QDMA implements SR-IOV passthrough virtualization where the adapter exposes a separate virtual function (VF) for use by a virtual machine (VM). A physical function (PF) can be optionally made privileged with full access to QDMA registers and resources, but only VFs implement per queue pointer update registers and interrupts. VF drivers must communicate with the driver attached to the PF through the mailbox for configuration, resource allocation, and exception handling. The QDMA implements function level reset (FLR) to enable operating system on VM to reset the device without interfering with the rest of the platform.

Table 30: **Privileged Access**

Type	Notes
Queue context/other control registers	Registers for Context access only controlled by PFs (All 4 PFs).
Status and statistics registers	Mainly PF only registers. VFs need to coordinate with a PF driver for error handling. VFs need to communicate through the mailbox with driver attached to PF.
Data path registers	Both PFs and VFs must be able to write the registers involved in data path without needing to go through a hypervisor. Pointer update for H2C/C2H Descriptor Fetch can be done directly by VF or PF for the queues associated with the function using its own BAR space. Any pointer updates to queue that do not belong to the function will be dropped with error logged.

Table 30: **Privileged Access** *(cont'd)*

Type	Notes
Other protection recommendations	Turn on IOMMU to protect bad memory accesses from VMs.
PF driver and VF driver communication	The VF driver needs to communicate with the PF driver to request operations that have global effect. This communication channel needs this ability to pass messages and generate interrupts. This communication channel utilizes a set of hardware mailboxes for each VF.

Mailbox

In a virtualized environment, the driver attached to a PF has enough privilege to program and access QDMA registers. For all the lesser privileged functions, certain PFs and all VFs must communicate with privileged drivers using the mailbox mechanism. The communication API must be defined by the driver. The QDMA IP does not define it.

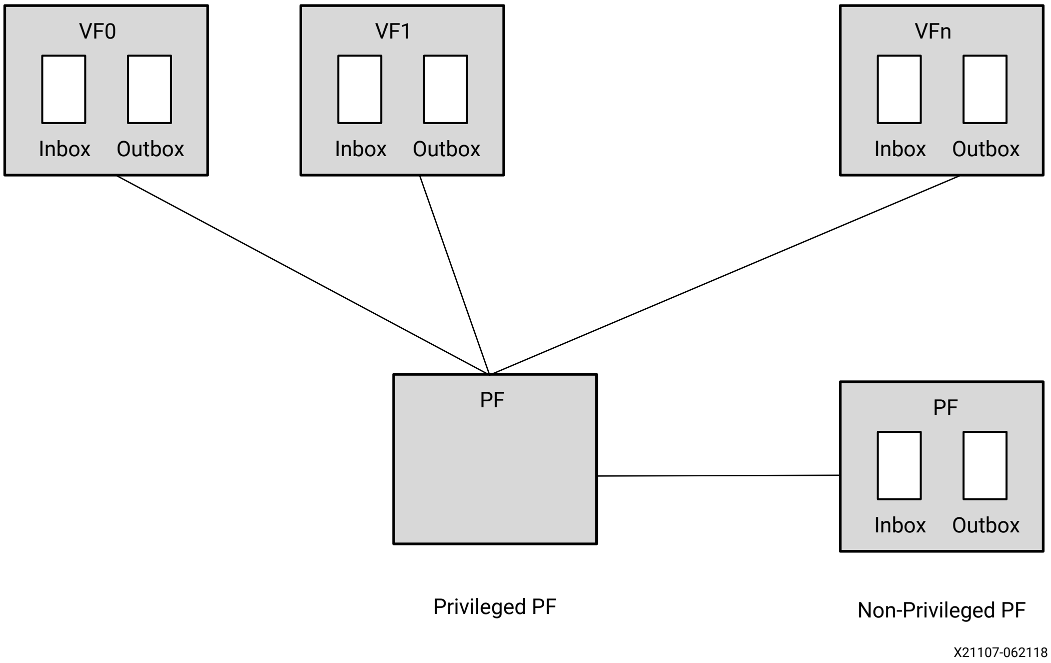
Each function (both PF and VF) has an inbox and an outbox that can fit a message size of 128B. A VF accesses its own mailbox, and a PF accesses its own mailbox and all the functions (PF or VF) associated with that PF.

***Note*:** Enabling mailbox will increase PL utilization.

The QDMA mailbox allows the following access:

- From a VF to the associated PF.
- From a PF to any VF belonging to its own virtual function group (VFG).
- From a PF (typically a driver that does not have access to QDMA registers) to another PF.

Figure 26: **Mailbox**

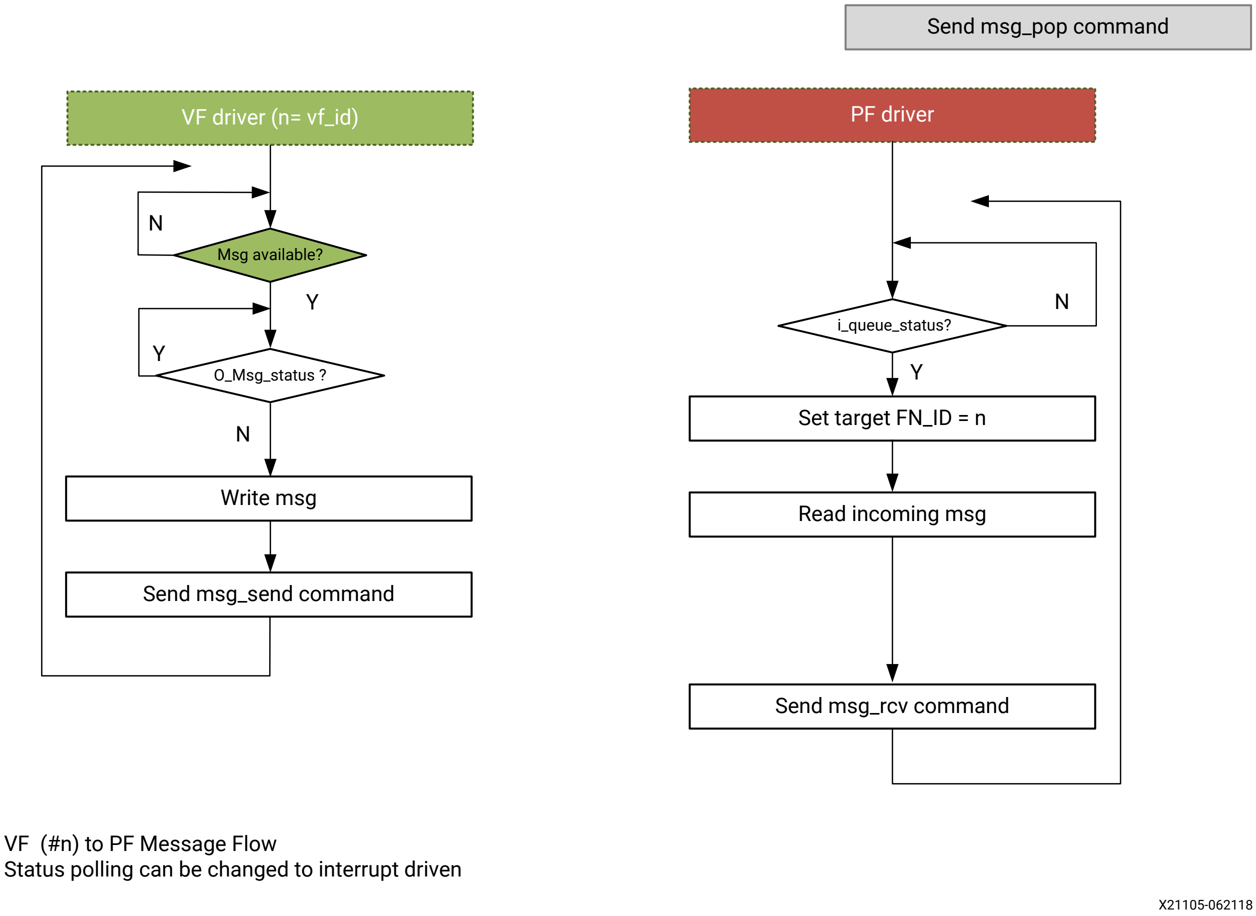


VF To PF Messaging

A VF is allowed to post one message to a target PF mailbox until the target function (PF) accepts it. Before posting the message the source function should make sure its `o_msg_status` is cleared, then the VF can write the message to its Outgoing Message Registers. After finishing message writing, the VF driver sends `msg_send` command through write 0x1 at the control/status register (CSR) address 0x5004. The mailbox hardware then informs the PF driver by asserting `i_msg_status` field.

The function driver should enable the periodic polling of the `i_msg_status` to check the availability of incoming messages. At a PF side, `i_msg_status = 0x1` indicates one or more message is pending for the PF driver to pick up. The `cur_src_fn` in the Mailbox Status Register gives the function ID of the first pending message. The PF driver should then set the Mailbox Target Function Register to the source function ID of the first pending message. Then access to a PF's Incoming Message Registers is indirectly, which means the mailbox hardware will always return the corresponding message bytes sent by the Target function. Upon finishing the message reading, the PF driver should also send `msg_rcv` command through write 0x2 at the CSR address. The hardware will deassert the `o_msg_status` at the source function side. The following figure illustrates the messaging flow from a VF to a PF at both the source and destination sides.

Figure 27: **VF to PF Messaging Flow**



PF To VF Messaging

The messaging flow from a PF to the VFs that belong to its VFG is slightly different than the VF to PF flow because:

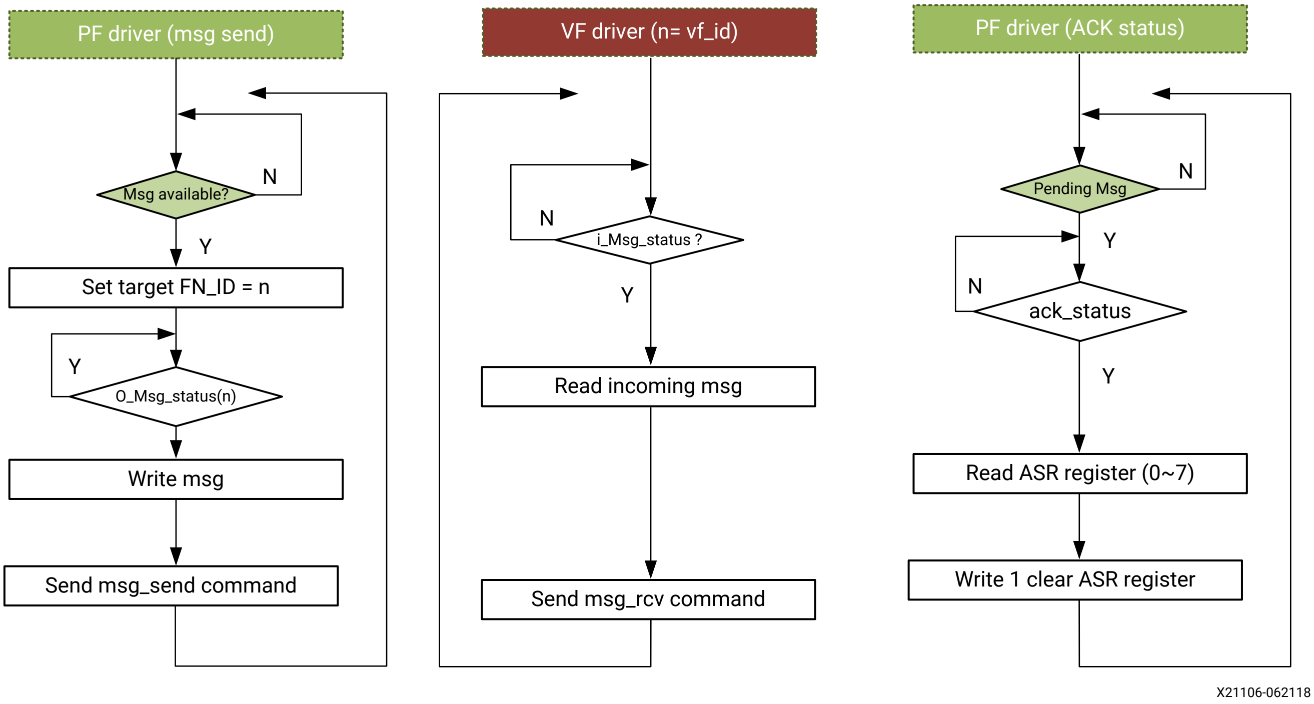
A PF can send messages to multiple destination functions, therefore, it may receives multiple acknowledgments at the moment when checking the status. As illustrated in the following figure, a PF driver must set Mailbox Target Function Register to the destination function ID before doing any message operation; for example, checking the incoming message status, write message, or send the command. At the VF side (receiving side), whenever a VF driver get the `i_msg_status = 0x1`, the VF driver should read its Incoming Message Registers to pick up the message. Depending on the application, the VF driver can send the `msg_rcv` immediately after reading the message or after the corresponding message being processed.

To avoid one-by-one polling of the status of outgoing messages, the mailbox hardware provides a set of Acknowledge Status Registers (ASR) for each PF. Upon the mailbox receiving the `msg_rcv` command from a VF, it deasserts the `o_msg_status` field of the source PF and it also sets the corresponding bit in the Acknowledge Status Registers. For a given VF with function ID <N>, acknowledge status is at:

- Acknowledge Status Register address: <N> / 32 + <0x22420 Register Address>
- Acknowledge Status bit location: <N> / 32

The mailbox hardware asserts the `ack_status` filed in the Status Register (0x22400) when there is any bit was asserted in the Acknowledge Status Register (ASR). The PF driver can poll the `ack_status` before actually reading out the Acknowledge status registers. The PF driver may detect multiple completions through one register access. After being processed, the PF driver should also write the value back to the same register address to clear the status.

Figure 28: **PF to VF Messaging Flow**



Mailbox Interrupts

The mailbox module supports interrupt as the alternative event notification mechanism. Each mailbox has an Interrupt Control Register (at the offset 0x22410 for a PF, or at the offset 0x5010 for a VF). Set 1 to this register to enable the interrupt. Once the interrupt is enabled, the mailbox will send the interrupt to the QDMA given there is any pending event for the mailbox to process, namely, any incoming message pending or any acknowledgment for the outgoing messages. Configure the interrupt vector through the Function Interrupt Vector Register (0x22408 for a FP, or 0x5008 for a VF) according to the driver configuration.

Enabling the interrupt does not change the event logging mechanism, which means the user must check the pending events through reading the Function Status Registers. The first step to respond to an interrupt request is disabling the interrupt. It is possible that the actual number of the pending events is more than the number of the events at the moment when the mailbox is sent the interrupt.

> ✓ **RECOMMENDED:** *Xilinx recommends that the user application interrupt handler process all the pending events that present in the status register. Upon finishing the interrupt response, the user application re-enables the interrupt.*

The mailbox will check its event status at the time the interrupt control change from disabled to enabled. If there is any new events that arrived the mailbox between reading the interrupt status and the re-enabling the interrupt, the mailbox will generate a new interrupt request immediately.

Related Information

QDMA_PF_MAILBOX (0x22400)
QDMA_VF_MAILBOX (0x5000)

Function Level Reset

The function level reset (FLR) mechanism enables the software to quiesce and reset Endpoint hardware with function-level granularity. When a VF is reset, only the resources associated with this VF are reset. When a PF is reset, all resources of the PF, including that of its associated VFs, are reset. Since FLR is a privileged operation, it must be performed by the PF driver running in the management system.

Use Mode

- Hypervisor requests for FLR when a function is attached and detached (i.e., power on and off).
- You can request FLR as follows:

  ```
  echo 1 > /sys/bus/pci/devices/$BDF/reset
  ```

  where `$BDF` is the bus device function number of the targeted function.

FLR Process

A complete FLR process involves of three major steps.

1. Pre-FLR: Pre-FLR resets all QDMA context structure, mailbox, and user logic of the target function.
   - Each function has a register called MDMA_PRE_FLR_STATUS, which keeps track of the pre-FLR status of the function. The offset is calculated as MDMA_PRE_FLR_STATUS_OFFSET = MB_base + 0x100, which is located at offset 0x100 from the mailbox memory space of the function. Note that PF and VF have different MB_base. The definition of MDMA_PRE_FLR_STATUS is shown in the table below.

- The software writes 1 to MDMA_PRE_FLR_STATUS[0] (bit 0) of the target function to initiate pre-FLR. Hardware will clear MDMA_PRE_FLR_STATUS[0] when pre-FLR completes. The software keeps polling on MDMA_PRE_FLR_STATUS[0], and only proceeds to the next step when it returns 0.

Table 31: **MDMA_PRE_FLR_STATUS Register**

Offset	Field	R/W Type	Width	Default	Description
0x100	pre_flr_st	RW	32	0	[31:1] reserved. [0]: 1 Initiates pre-FLR. [0]: 0 pre-FLR done. bit[0] is set by the driver and cleared by the hardware.

2. Quiesce: The software must ensure all pending transaction is completed. This can be done by polling the Transaction Pending bit in the Device Status register (in PCIe Configuration Space), until it is cleared or times out after a certain period of time.
3. PCIe-FLR: PCIe-FLR resets all resources of the target function in the PCIe controller.

   ***Note*:** Initiate Function Level Reset bit (bit 15 of PCIe Device Control Register) of the target function should be set to 1 to trigger FLR process in PCIe.

OS Support

If the PF driver is loaded and alive (i.e., use mode 1), all three steps aforementioned are performed by the driver. However, for Versal, if a user wants to perform FLR before loading the PF driver (as defined in Use Mode above), an OS kernel patch is provided to allow OS to perform the correct FLR sequence through functions defined in `//…/source/drivers/pci/quick.c`.

Port ID

Port ID is the categorization of some queues on the FPGA side. When the DMA is shared by more than one user application, the port ID provides indirection to QID so that all the interfaces can be further demuxed with lower cost. However, when used by a single application, the port ID can be ignored and drive the port id inputs to 0s.

Host Profile

Host profile must be programmed to represent Root Port host. Host profile can be programmed through Context programming. Select QDMA_CTXT_SELC_HOST_PROFILE (4'hA) in QDMA_IND_CTXT_CMD. Host profile context structure is given in below table.

Table 32: **Host Profile Context Structure**

Bit	Bit Width	Field Name	Description
[255:188]	68	Reserved	Reserved

Table 32: **Host Profile Context Structure** *(cont'd)*

Bit	Bit Width	Field Name	Description
[187:186]	2		H2C AXI4-MM write awprot
[185:182]	4		H2C AXI4-MM write awcache
[181:178]	4		H2C AXIMM steering
[177:104]	74	Reserved	Reserved
[103:102]	2		C2H AXI4-MM read arprot
[101:98]	4		C2H AXI4-MM read awcache
[97:94]	4		C2H AXIMM steering
[0:93]	94	Reserved	Reserved

H2C AXI4-MM Steering bit and C2H AXI4-MM Steering bits should set to 0's .if not DMA AXI4-MM transfers will not work. For most cases Host profile context structure will be all 0s, and Host profile must still be program to represent a host.

System Management

Resets

The QDMA supports all the PCIe defined resets, such as link down, reset, hot reset, and function level reset (FLR) (supports only Quiesce mode).

Soft Reset

Reset the QDMA logic through the `soft_reset_n` port. This port needs to be held in reset for a minimum of 100 clock cycles (`axi_aclk` cycles).

This does not reset PCIe hard block. It resets only the DMA portion of logic. This reset can be asserted if there is a DMA hang or some error condition.

Soft Reset Use Cases

The uses cases that prompt the use of `soft_reset` include:

- DMA hangs and user is not getting proper values.
- DMA transfer have errors, but the PCIe links are good.
- DMA records some asynchronous error

After `soft_reset`, you must reinitialize the queues and program all queue context.

VDM

Vendor Defined Messages (VDMs) are an expansion of the existing messaging capabilities with PCI Express. PCI Express Specification defines additional requirements for Vendor Defined Messages, header formats and routing information. For details, see *PCI-SIG Specifications* (https://www.pcisig.com/specifications).

QDMA allows the transmission and reception of VDMs. To enable this feature, select **Enable Bridge Slave Mode** in the Vivado Customize IP dialog box. This enables the `st_rx_msg` interface.

RX Vendor Defined Messages are stored in shallow FIFO before they are transmitted to the output port. When there are many back-to-back VDM messages, FIFO will overflow and these message will be dropped. So it is better to repeat VDM messages at regular intervals.

Throughput for VDMs depend on several factors: PCIe speed, data width, message length, and the internal VDM pipeline.

Internal VDM pipelines must be replaced with the Internal RX VDM FIFO interface for network on chip (NoC) access, which has a shallow buffer of 64B.

***Note*:** New VDM messages will be dropped if more than 64B of VDM are received before the FIFO is serviced through NoC.

Internal RX VDM FIFO interface cannot handle back-to-back messages. Pipeline throughput can only handle one in every four accesses, which is about 25% efficiency from the host access.

IMPORTANT! *Do not use back-to-back VDM access.*

RX Vendor Defined Messages:

1. When QDMA receives a VDM, the incoming messages will be received on the `st_rx_msg` port.
2. The incoming data stream will be captured on the `st_rx_msg_data` port (per-DW).
3. The user application needs to drive the `st_rx_msg_rdy` to signal if it can accept the incoming VDMs.
4. Once `st_rx_msg_rdy` is High, the incoming VDM is forwarded to the user application.
5. The user application needs to store this incoming VDMs and track of how many packets were received.

TX Vendor Defined Messages:

1. To enable transmission of VDM from QDMA, program the TX Message registers in the Bridge through the AXI4-Lite Slave interface.

2. Bridge has TX Message Control, Header L (bytes 8-11), Header H (bytes 12-15) and TX Message Data registers as shown in the PCIe TX Message Data FIFO Register (TX_MSG_DFIFO).
3. Issue a Write to offset 0xE64 through AXI4-Lite Slave interface for the TX Message Header L register.
4. Program offset 0xE68 for the required VDM TX Header H register.
5. Program up to 16DW of Payload for the VDM message starting from DW0 – DW15 by sending Writes to offset 0xE6C one by one.
6. Program the `msg_routing`, `msg_code`, data length, requester function field and `msg_execute` field in the TX_MSG_CTRL register in offset 0xE60 to send the VDM TX packet.
7. The TX Message Control register also indicates the completion status of the message in bit 23. User needs to read this bit to confirm the successful transmission of the VDM packet.
8. All the fields in the registers are RW except bit 23 (`msg_fail`) in TX Control register which is cleared by writing a 1.
9. VDM TX packet will be sent on the AXI-ST RQ transmit interface.

Related Information

VDM Ports
Bridge Register Space

Config Extend

PCIe extended interface can be selected for more configuration space. When the Configuration Extend Interface is selected, you are responsible for adding logic to extend the interface to make it work properly.

Expansion ROM

If selected, the Expansion ROM is activated and can be a value from 2 KB to 4 GB. According to the PCI Local Bus Specification (*PCI-SIG Specifications* (https://www.pcisig.com/specifications)), the maximum size for the Expansion ROM BAR should be no larger than 16 MB. Selecting an address space larger than 16 MB can result in a non-compliant core.

Errors

Bridge Errors

Slave Bridge Abnormal Conditions

Slave bridge abnormal conditions are classified as: Illegal Burst Type and Completion TLP Errors. The following sections describe the manner in which the Bridge handles these errors.

Illegal Burst Type

The slave bridge monitors AXI read and write burst type inputs to ensure that only the INCR (incrementing burst) type is requested. Any other value on these inputs is treated as an error condition and the Slave Illegal Burst (SIB) interrupt is asserted. In the case of a read request, the Bridge asserts SLVERR for all data beats and arbitrary data is placed on the `s_axi_rdata` bus. In the case of a write request, the Bridge asserts SLVERR for the write response and all write data is discarded.

Completion TLP Errors

Any request to the bus for PCIe (except for a posted Memory write) requires a completion TLP to complete the associated AXI request. The Slave side of the Bridge checks the received completion TLPs for errors and checks for completion TLPs that are never returned (Completion Timeout). Each of the completion TLP error types are discussed in the subsequent sections.

Unexpected Completion

When the slave bridge receives a completion TLP, it matches the header RequesterID and Tag to the outstanding RequesterID and Tag. A match failure indicates the TLP is an Unexpected Completion which results in the completion TLP being discarded and a Slave Unexpected Completion (SUC) interrupt strobe being asserted. Normal operation then continues.

Unsupported Request

A device for PCIe might not be capable of satisfying a specific read request. For example, if the read request targets an unsupported address for PCIe, the completer returns a completion TLP with a completion status of `0b001 - Unsupported Request`. The completer that returns a completion TLP with a completion status of `Reserved` must be treated as an unsupported request status, according to the PCI Express Base Specification v3.0. When the slave bridge receives an unsupported request response, the Slave Unsupported Request (SUR) interrupt is asserted and the DECERR response is asserted with arbitrary data on the AXI4 memory mapped bus.

Completion Timeout

A Completion Timeout occurs when a completion (Cpl) or completion with data (CplD) TLP is not returned after an AXI to PCIe memory read request, or after a PCIe Configuration Read/Write request. For PCIe Configuration Read/Write request, completions must complete within the `C_COMP_TIMEOUT` parameter selected value from the time the request is issued. For PCIe Memory Read request, completions must complete within the value set in the Device Control 2 register in the PCIe Configuration Space register. When a completion timeout occurs, an OKAY response is asserted with all 1s data on the memory mapped AXI4 bus.

Poison Bit Received on Completion Packet

An Error Poison occurs when the completion TLP EP bit is set, indicating that there is poisoned data in the payload. When the slave bridge detects the poisoned packet, the Slave Error Poison (SEP) interrupt is asserted and the SLVERR response is asserted with arbitrary data on the memory mapped AXI4 bus.

Completer Abort

A Completer Abort occurs when the completion TLP completion status is `0b100` - Completer Abort. This indicates that the completer has encountered a state in which it was unable to complete the transaction. When the slave bridge receives a completer abort response, the Slave Completer Abort (SCA) interrupt is asserted and the SLVERR response is asserted with arbitrary data on the memory mapped AXI4 bus.

Table 33: **Slave Bridge Response to Abnormal Conditions**

Transfer Type	Abnormal Condition	Bridge Response
Read	Illegal burst type	SIB interrupt is asserted. SLVERR response given with arbitrary read data.
Write	Illegal burst type	SIB interrupt is asserted. Write data is discarded. SLVERR response given.
Read	Unexpected completion	SUC interrupt is asserted. Completion is discarded.
Read	Unsupported Request status returned	SUR interrupt is asserted. DECERR response given with arbitrary read data.
Read	Completion timeout	SCT interrupt is asserted. SLVERR response given with arbitrary read data.
Read	Poison bit in completion	Completion data is discarded. SEP interrupt is asserted. SLVERR response given with arbitrary read data.
Read	Completer Abort (CA) status returned	SCA interrupt is asserted. SLVERR response given with arbitrary read data.

PCIe Error Handling

Master Bridge Abnormal Conditions

The following sections describe the manner in which the master bridge handles abnormal conditions.

AXI DECERR Response

When the master bridge receives a DECERR response from the AXI bus, the request is discarded and the Master DECERR (MDE) interrupt is asserted. If the request was non-posted, a completion packet with the Completion Status = Unsupported Request (UR) is returned on the bus for PCIe.

AXI SLVERR Response

When the master bridge receives a SLVERR response from the addressed AXI slave, the request is discarded and the Master SLVERR (MSE) interrupt is asserted. If the request was non-posted, a completion packet with the Completion Status = Completer Abort (CA) is returned on the bus for PCIe.

Max Payload Size for PCIe, Max Read Request Size

When the master bridge receives a SLVERR response from the addressed AXI slave, the request is discarded and the Master SLVERR (MSE) interrupt is asserted. If the request was non-posted, a completion packet with the Completion Status = Completer Abort (CA) is returned on the bus for PCIe.

Completion Packets

When the `MAX_READ_REQUEST_SIZE` is greater than the `MAX_PAYLOAD_SIZE`, a read request for PCIe can ask for more data than the master bridge can insert into a single completion packet. When this situation occurs, multiple completion packets are generated up to `MAX_PAYLOAD_SIZE`, with the Read Completion Boundary (RCB) observed.

Poison Bit

When the poison bit is set in a transaction layer packet (TLP) header, the payload following the header is corrupted. When the master bridge receives a memory request TLP with the poison bit set, it discards the TLP and asserts the Master Error Poison (MEP) interrupt strobe.

Zero Length Requests

When the master bridge receives a read request with the Length = `0x1`, FirstBE = `0x00`, and LastBE = `0x00`, it responds by sending a completion with `Status = Successful Completion`.

When the master bridge receives a write request with the Length = `0x1`, FirstBE = `0x00`, and LastBE = `0x00` there is no effect.

Table 34: **Master Bridge Response to Abnormal Conditions**

Transfer Type	Abnormal Condition	Bridge Response
Read	DECERR response	MDE interrupt strobe asserted. Completion returned with Unsupported Request status.
Write	DECERR response	MDE interrupt strobe asserted.
Read	SLVERR response	MSE interrupt strobe asserted. Completion returned with Completer Abort status.
Write	SLVERR response	MSE interrupt strobe asserted.
Write	Poison bit set in request	MEP interrupt strobe asserted. Data is discarded.
Read	DECERR response	MDE interrupt strobe asserted. Completion returned with Unsupported Request status.
Write	DECERR response	MDE interrupt strobe asserted.

Linkdown Errors

If the PCIe link goes down during DMA operations, transactions may be lost and the DMA may not be able to complete. In such cases, the AXI4 interfaces will continue to operate. Outstanding read requests on the C2H Bridge AXI4 MM interface receive correct completions or completions with a slave error response. The DMA will log a link down error in the status register. It is the responsibility of the driver to have a timeout and handle recovery of a link down situation.

Data Path Errors

Data protection is supported on the primary data paths. CRC error can occur on C2H streaming, H2C streaming. Parity error can occur on Memory Mapped, Bridge Master and Bridge Slave interfaces. Error on Write payload can occur on C2H streaming, Memory Mapped and Bridge Slave. Double bit error on write payload and read completions for Bridge Slave interface causes parity error. Parity errors on requests to the PCIe are dropped by the core, and a fatal error is logged by the PCIe. Parity errors are not recoverable and can result in unexpected behavior. Any DMA during and after the parity error should be considered invalid. If there is a parity error and transfer hangs or stops, the DMA will log the error. You must investigate and fix the parity issues. Once the issues are fixed, clear that queue and reopen the queue to start a new transfer.

DMA Errors

All DMA errors are logged in their respective error status register. Each block has error status and error mask register so error can be passed on to higher level and eventually to QDMA_GLBL_ERR_STAT register.

Errors can be fatal error based on register settings. If there is an fatal error DMA will stop the transfer and will send interrupt if enabled. After debug and analysis, you must invalidate and restart the queue to start the DMA transfer.

Error Aggregator

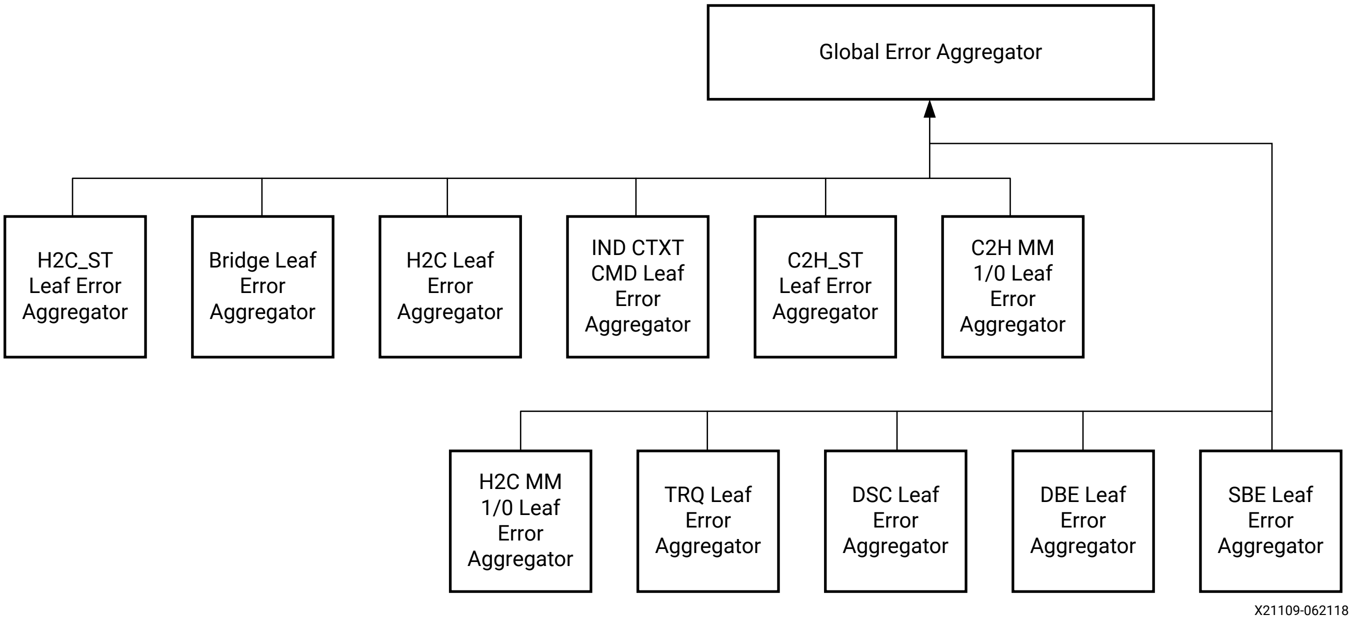
There are Leaf Error Aggregators in different places. They log the errors and propagate them to the central place. The Central Error Aggregator aggregates the errors from all of the Leaf Error Aggregators.

The QDMA_GLBL_ERR_STAT register is the error status register of the Central Error Aggregator. The bit fields indicate the locations of Leaf Error Aggregators. Then, look for the error status register of the individual Leaf Error Aggregator to find the exact error.

The register QDMA_GLBL_ERR_MASK is the error mask register of the Central Error Aggregator. It has the mask bits for the corresponding errors. When the mask bit is set to `1'b1`, it will enable the corresponding error to be propagated to the next level to generate an Interrupt. The detail information of the error generated interrupt is described in the interrupt section. Error interrupt is controlled by the register QDMA_GLBL_ERR_INT (0xB04).

Each Leaf Error Aggregator has an error status register and an error mask register. The error status register logs the error. The hardware sets the bit when the error happens, and the software can write `1'b1` to clear the bit if needed. The error mask register has the mask bits for the corresponding errors. When the mask bit is set to `1'b1`, it will enable the propagation of the corresponding error to the Central Error Aggregator. The error mask register does not affect the error logging to the error status register.

Figure 29: **Error Aggregator**


Global Error Aggregator
H2C_ST Leaf Error Aggregator
Bridge Leaf Error Aggregator
H2C Leaf Error Aggregator
IND CTXT CMD Leaf Error Aggregator
C2H_ST Leaf Error Aggregator
C2H MM 1/0 Leaf Error Aggregator
H2C MM 1/0 Leaf Error Aggregator
TRQ Leaf Error Aggregator
DSC Leaf Error Aggregator
DBE Leaf Error Aggregator
SBE Leaf Error Aggregator
X21109-062118

The error status registers and the error mask registers of the Leaf Error Aggregators are as follows.

C2H Streaming Error

QDMA_C2H_ERR_STAT (0xAF0): This is the error status register of the C2H streaming errors.

QDMA_C2H_ERR_MASK (0xAF4): This the error mask register. The software can set the bit to enable the corresponding C2H streaming error to be propagated to the Central Error Aggregator.

QDMA_C2H_FIRST_ERR_QID (0xB30): This is the Qid of the first C2H streaming error.

C2H MM Error

QDMA_C2H MM Status (0x1040)

C2H MM Error Code Enable Mask (0x1054)

C2H MM Error Code (0x1058)

C2H MM Error Info (0x105C)

QDMA H2C0 MM Error

H2C0 MM Status (0x1240)

H2C MM Error Code Enable Mask (0x1254)

H2C MM Error Code (0x1258)

H2C MM Error Info (0x125C)

TRQ Error

QDMA_GLBL_TRQ_ERR_STS (0x264): This is the error status register of the Trq errors.

QDMA_GLBL_TRQ_ERR_MSK (0x268): This is the error mask register.

QDMA_GLBL_TRQ_ERR_LOG_A (0x26C): This is the error logging register. It shows the select, function and the address of the access when the error happens.

Descriptor Error

QDMA_GLBL_DSC_ERR_STS (0x254)

QDMA_GLBL_DSC_ERR_MSK (0x258): This is the error logging register. It has the QID, DMA direction, and the consumer index of the error.

QDMA_GLBL_DSC_ERR_LOG0 (0x25C)

QDMA_GLBL_TRQ_ERR_STS (0x264): This is the error status register of the TRQ errors.

RAM Double Bit Error

QDMA_RAM_DBE_STS_A (0xFC)

QDMA_RAM_DBE_MSK_A (0xF8)

RAM Single Error

QDMA_RAM_SBE_STS_A (0xF4)
QDMA_RAM_SBE_MSK_A (0xF0)

Related Information

Register Space

C2H Streaming Fatal Error Handling

- QDMA_C2H_FATAL_ERR_STAT (0xAF8): The error status register of the C2H streaming fatal errors.
- QDMA_C2H_FATAL_ERR_MASK (0xAFC): The error mask register. The SW can set the bit to enable the corresponding C2H fatal error to be sent to the C2H fatal error handling logic.
- QDMA_C2H_FATAL_ERR_ENABLE (0xB00): This register enables two C2H streaming fatal error handling processes:
  1. Stop the data transfer by disabling the *write request* from the C2H DMA Write Engine by setting `enable_wrq_dis` bit [0] to 1.
  2. Invert the *write payload parity* on the data transfer by setting `enable_wpl_par_inv` bit [1] to 1.

Port Descriptions

***Note*:** The Versal ACAP DMA and Bridge Subsystem for PCIe IP is implemented in a modular IP architecture. This means that GTs, PCIe IP, and the subsystem IP are implemented separately. The interface signals between GTs and PCIe IP going to a subsystem IP are not listed in this guide. These interface signals are found in *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343). The signals below apply to the subsystem only.

The QDMA connects directly to the PCIe Integrated Block. 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. Ports associated with this core are described below.

Table 35: **Parameters**

Parameter Name	Description
PL_LINK_CAP_MAX_LINK_WIDTH	Phy lane width
C_M_AXI_ADDR_WIDTH	AXI4 Master interface Address width
C_M_AXI_ID_WIDTH	AXI4 Master interface id width
C_M_AXI_DATA_WIDTH	AXI4 Master interface data width 64 or 128 or 256 or 512 bits

Table 35: **Parameters** *(cont'd)*

Parameter Name	Description
C_S_AXI_ID_WIDTH	AXI4 Bridge Slave interface id width
C_S_AXI_ADDR_WIDTH	AXI4 Bridge Slave interface Address width
C_S_AXI_DATA_WIDTH	AXI4 Bridge Slave interface data width 64 or 128 or 256 or 512 bits
C_S_AXI_ID_WIDTH	AXI4 Bridge Slave interface id width
AXI_DATA_WIDTH	AXI4 DMA transfer data width. Example 64 or 128 or 256 or 512 bits

Related Information

QDMA Architecture

QDMA Global Ports

Table 36: **QDMA Global Port Descriptions**

Port Name	I/O	Description
axi_aclk	O	User clock out. PCIe derived clock output for all interface signals output from QDMA, and input to QDMA. Use this clock to drive inputs and gate outputs from QDMA.
axi_aresetn	O	User reset out. AXI reset signal synchronous with the clock provided on the axi_aclk output. This reset should drive all corresponding AXI interconnect aresetn signals.
soft_reset_n	I	Soft reset (active-Low). Use this port to assert reset and reset the DMA logic. This will reset only the DMA logic. You should assert and de-assert this port.
phy_rdy_out_sd	I	Active-High signal that indicates when Phy is ready. This signal is from the Phy block.
user_lnk_up_sd	I	Active-High identifies that the PCI Express core is linked up with a host device. This signal is from the PCIe block
user_clk_sd	I	User clock from the PCIe block. All of the QDMA blocks use this clock
user_reset_sd	I	Active-High user reset signals from PCIe block.
csr_prog_done	O	This port is enabled only when the AXI-Lite CSR Slave Interface option is selected in Basic tab during IP customization. This port indicates whether you can access the AXI Lite CSR interface. 1'b0: The AXI Lite CSR Slave interface is not accessible. 1'b1: The AXI-Lite CSR Slave interface is accessible.

All AXI interfaces are clocked out and in by the `axi_aclk` signal. You are responsible for using `axi_aclk` to driver all signals into the DMA.

QDMA Global Ports

Table 37: **QDMA Global Port Descriptions**

Port Name	I/O	Description
sys_clk	I	Should be driven by the ODIV2 port of reference clock IBUFDS_GTE4. See the Versal Integrated Block for PCI Express LogiCORE IP Product Guide (PG343) .
sys_clk_gt	I	PCIe reference clock. Should be driven from the port of reference clock IBUFDS_GTE4. See the Versal Integrated Block for PCI Express LogiCORE IP Product Guide (PG343) .
sys_rst_n	I	Reset from the PCIe edge connector reset signal.
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.
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.
user_lnk_up	O	Output active-High identifies that the PCI Express core is linked up with a host device.
axi_aclk	O	User clock out. PCIe derived clock output for for all interface signals output from and input to QDMA. Use this clock to drive inputs and gate outputs from QDMA.
axi_aresetn	O	User reset out. AXI reset signal synchronous with the clock provided on the axi_aclk output. This reset should drive all corresponding AXI Interconnect aresetn signals.
soft_reset_n	I	Soft reset (active-Low). Use this port to assert reset and reset the DMA logic. This will reset only the DMA logic. User should assert and de-assert this port.
phy_ready	O	Phy ready out status.
csr_prog_done	O	This port is enabled only when the AXI-Lite CSR Slave Interface option is selected in the Basic tab in the IP customization GUI. This port indicates whether access to AXI Lite CSR interface is available. 1'b0: The AXI Lite CSR Slave interface is not accessible. 1'b1: The AXI Lite CSR Slave interface is accessible.

All AXI interfaces are clocked out and in by the `axi_aclk` signal. You are responsible for using `axi_aclk` to driver all signals into the DMA.

AXI Bridge Master Ports

Table 38: **AXI4 Memory Mapped Master Bridge Read Address Interface Port Descriptions**

Signal Name	I/O	Description
m_axib_araddr [C_M_AXI_ADDR_WIDTH-1:0]	O	This signal is the address for a memory mapped read to the user logic from the host.

Table 38: **AXI4 Memory Mapped Master Bridge Read Address Interface Port Descriptions** *(cont'd)*

Signal Name	I/O	Description
m_axib_arid [C_M_AXI_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	Master read protection type.
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	Master read lock type.
m_axib_arcache[3:0]	O	Master read memory type.
m_axib_arburst[1:0]	O	Master read address burst type.
m_axib_aruser[28:0]	O	Master read user bits. m_axib_aruser[7:0] = function number m_axib_aruser[15:8] = reserved m_axib_aruser[18:16] = bar id m_axib_aruser[26:19] = vf offset m_axib_aruser[28:27] = vf id

Table 39: **AXI4 Memory Mapped Master Bridge Read Interface Port Descriptions**

Signal Name	I/O	Description
m_axib_rdata [C_M_AXI_DATA_WIDTH-1:0]	I	Master read data.
m_axib_ruser [C_M_AXI_DATA_WIDTH/8-1:0]	I	m_axib_ruser[C_M_DATA_WIDTH/8-1:0] = read data odd parity, per byte.
m_axib_rid [C_M_AXI_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.

Table 40: **AXI4 Memory Mapped Master Bridge Write Address Interface Port Descriptions**

Signal Name	I/O	Description
m_axib_awaddr [C_M_AXI_ADDR_WIDTH-1:0]	O	This signal is the address for a memory mapped write to the user logic from the host.
m_axib_awid [C_M_AXI_ID_WIDTH-1:0]	O	Master write address ID.
m_axib_awlen[7:0]	O	Master write address length.

Table 40: **AXI4 Memory Mapped Master Bridge Write Address Interface Port Descriptions** *(cont'd)*

Signal Name	I/O	Description
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	Master write protection type.
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	Master write lock type.
m_axib_awcache[3:0]	O	Master write memory type.
m_axib_awuser[28:0]	O	Master write user bits. m_axib_aruser[7:0] = function number m_axib_aruser[15:8] = reserved m_axib_aruser[18:16] = bar id m_axib_aruser[26:19] = vf offset m_axib_aruser[28:27] = vf id

Table 41: **AXI4 Memory Mapped Master Bridge Write Interface Port Descriptions**

Signal Name	I/O	Description
m_axib_wdata [C_M_AXI_DATA_WIDTH-1:0]	O	Master write data.
m_axib_wuser [C_M_AXI_DATA_WIDTH/8-1:0]	O	m_axib_wuser [C_M_AXI_DATA_WIDTH/8-1:0] = write data odd parity, per byte.
m_axib_wlast	O	Master write last.
m_axib_wstrb [C_M_AXI_DATA_WIDTH/8-1:0]	O	Master write strobe.
m_axib_wvalid	O	Master write valid.
m_axib_wready	I	Master write ready.

Table 42: **AXI4 Memory Mapped Master Bridge Write Response Interface Port Descriptions**

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

AXI Bridge Slave Ports

Table 43: **AXI4 Bridge Slave Write Address Interface Port Descriptions**

Port Name	I/O	Description
s_axib_awid [C_S_AXI_ID_WIDTH-1:0]	I	Slave write address ID.
s_axib_awaddr [C_S_AXI_ADDR_WIDTH-1:0]	I	Slave write address.
s_axib_awuser[7:0]	I	s_axib_awuser[7:0] indicates function_number.
s_axib_awregion[3:0]	I	Slave write region decode.
s_axib_awlen[7:0]	I	Slave write burst length.
s_axib_awsize[2:0]	I	Slave write burst size.
s_axib_awburst[1:0]	I	Slave write burst type. Only the INCR burst type is supported.
s_axib_awvalid	I	Slave address write valid.
s_axib_awready	O	Slave address write ready.

Table 44: **AXI4 Bridge Slave Write Interface Port Descriptions**

Port Name	I/O	Description
s_axib_wdata [C_S_AXI_DATA_WIDTH-1:0]	I	Slave write data.
s_axib_wstrb [C_S_AXI_DATA_WIDTH/8-1:0]	I	Slave write strobe.
s_axib_wlast	I	Slave write last.
s_axib_wvalid	I	Slave write valid.
s_axib_wready	O	Slave write ready.
s_axib_wuser [C_S_AXI_DATA_WIDTH/8-1:0]	I	s_axib_wuser [C_S_AXI_DATA_WIDTH/8-1:0] = write data odd parity, per byte.

Table 45: **AXI4 Bridge Slave Write Response Interface Port Descriptions**

Port Name	I/O	Description
s_axib_bid [C_S_AXI_ID_WIDTH-1:0]	O	Slave response ID.
s_axib_bresp[1:0]	O	Slave write response.
s_axib_bvalid	O	Slave write response valid.
s_axib_bready	I	Slave response ready.

Table 46: **AXI4 Bridge Slave Read Address Interface Port Descriptions**

Port Name	I/O	Description
s_axib_arid [C_S_AXI_ID_WIDTH-1:0]	I	Slave read address ID.

Table 46: **AXI4 Bridge Slave Read Address Interface Port Descriptions** *(cont'd)*

Port Name	I/O	Description
s_axib_araddr [C_S_AXI_ADDR_WIDTH-1:0]	I	Slave read address.
s_axib_arregion[3:0]	I	Slave read region decode.
s_axib_arlen[7:0]	I	Slave read burst length.
s_axib_arsize[2:0]	I	Slave read burst size.
s_axib_arburst[1:0]	I	Slave read burst type. Only the INCR burst type is supported.
s_axib_arvalid	I	Slave read address valid.
s_axib_arready	O	Slave read address ready.

Table 47: **AXI4 Bridge Slave Read Interface Port Descriptions**

Port Name	I/O	Description
s_axib_rid [C_S_AXI_ID_WIDTH-1:0]	O	Slave read ID tag.
s_axib_rdata [C_S_AXI_ID_WIDTH-1:0]	O	Slave read data.
s_axib_ruser [C_S_AXI_DATA_WIDTH/8-1:0]	O	s_axib_aruser[C_S_AXI_ID_WIDTH/8-1:0] = read data odd parity, per byte.
s_axib_rresp[1:0]	O	Slave read response.
s_axib_rlast	O	Slave read last.
s_axib_rvalid	O	Slave read valid.
s_axib_rready	I	Slave read ready.

AXI4-Lite Master Ports

Table 48: **Config AXI4-Lite Memory Mapped Write Master Interface Port Descriptions**

Signal Name	I/O	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	Protection type.
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_acil_awuser [29:0]	O	m_axib_awuser[7:0] = function number m_axib_awuser[15:8] = reserved m_axib_awuser[18:16] = bar id m_axib_awuser[26:19] = vf offset m_axib_awuser[28:27] = vf id
m_axil_wdata[31:0]	O	Master write data.
m_axil_wstrb[3:0]	O	Master write strobe.
m_axil_wvalid	O	Master write valid.

Table 48: **Config AXI4-Lite Memory Mapped Write Master Interface Port Descriptions** *(cont'd)*

Signal Name	I/O	Description
m_axil_wready	I	Master write ready.
m_axil_bvalid	I	Master response valid.
m_axil_bresp[1:0]	I	
m_axil_bready	O	Master response valid.

Table 49: **Config AXI4-Lite Memory Mapped Read Master Interface Port Descriptions**

Signal Name	I/O	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_aruser[28:0]	O	m_axib_aruser[7:0] = function number m_axib_aruser[15:8] = reserved m_axib_aruser[18:16] = bar id m_axib_aruser[26:19] = vf offset m_axib_aruser[28:27] = vf id
m_axil_arprot[2:0]	O	Protection type.
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[1:0]	I	Master read response.
m_axil_rvalid	I	Master read valid.
m_axil_rready	O	Master read ready.

AXI4-Lite Slave Ports

AXI4-Lite Slave ports can be used to access QDMA Queue space registers (QDMA_TRQ_SEL_QUEUE_PF (0x18000) and QDMA_TRQ_SEL_QUEUE_VF (0x3000)).

Table 50: **Config AXI4-Lite Memory Mapped Write Slave Interface Signals**

Signal Name	I/O	Description
s_axil_awaddr[31:0]	I	This signal is the address for a memory mapped write to the DMA Queue space registers 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_awuser[12:0]	I	[12:8]: Reserved [7:0]: Function number
s_axil_awprot[2:0]	I	Protection type. This port is not being used.
s_axil_awready	O	Slave write address ready.

Table 50: **Config AXI4-Lite Memory Mapped Write Slave Interface Signals** *(cont'd)*

Signal Name	I/O	Description
s_axil_wdata[31:0]	I	Slave write data.
s_axil_wstrb[3:0]	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_bresp[1:0]	O	Slave write response.
s_axil_bready	I	Save response ready.

Table 51: **Config AXI4-Lite Memory Mapped Read Slave Interface Signals**

Signal Name	I/O	Description
s_axil_araddr[31:0]	I	This signal is the address for a memory mapped read to the DMA Queue space from the user logic.
s_axil_arprot[2:0]	I	Protection type. This port is not being used.
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_aruser[12:0]	I	[12:8]: Reserved [7:0]: Function number
s_axil_arready	O	Slave read address ready.
s_axil_rdata[31:0]	O	Slave read data.
s_axil_rresp[1:0]	O	Slave read response.
s_axil_rvalid	O	Slave read valid.
s_axil_rready	I	Slave read ready.

AXI4-Lite Slave CSR Ports

Table 52: **Config AXI4-Lite Memory Mapped Write Slave CSR Interface Signals**

Signal Name	I/O	Description
s_axil_csr_awaddr[31:0]	I	This signal is the address for a memory mapped write to the DMA from the user logic. s_axil_csr_awaddr[15]: 1'b1 – QDMA CSR register 1'b0 – Bridge register
s_axil_csr_awvalid	I	The assertion of this signal means there is a valid write request to the address on s_axil_csr_awaddr.
s_axil_csr_awprot[2:0]	I	Protection type. This port is not being used.
s_axil_csr_awready	O	Slave write address ready.
s_axil_csr_wdata[31:0]	I	Slave write data.
s_axil_csr_wstrb[3:0]	I	Slave write strobe.
s_axil_csr_wvalid	I	Slave write valid.

Table 52: **Config AXI4-Lite Memory Mapped Write Slave CSR Interface Signals** *(cont'd)*

Signal Name	I/O	Description
s_axil_csr_wready	O	Slave write ready.
s_axil_csr_bvalid	O	Slave write response valid.
s_axil_csr_bresp[1:0]	O	Slave write response.
s_axil_csr_bready	I	Save response ready.

Table 53: **Config AXI4-Lite Memory Mapped Read Slave CSR Interface Signals**

Signal Name	I/O	Description
s_axil_csr_araddr[31:0]	I	This signal is the address for a memory mapped read to the DMA from the user logic. s_axil_csr_araddr[15]: 1'b1 – QDMA register 1'b0 – Bridge register
s_axil_csr_arprot[2:0]	I	Protection type. This port is not being used.
s_axil_csr_arvalid	I	The assertion of this signal means there is a valid read request to the address on s_axil_csr_araddr.
s_axil_csr_arready	O	Slave read address ready.
s_axil_csr_rdata[31:0]	O	Slave read data.
s_axil_csr_rresp[1:0]	O	Slave read response.
s_axil_csr_rvalid	O	Slave read valid.
s_axil_csr_rready	I	Slave read ready.

AXI4 Memory Mapped DMA Ports

Table 54: **AXI4 Memory Mapped DMA Read Address Interface Signals**

Signal Name	Direction	Description
m_axi_araddr [C_M_AXI_ADDR_WIDTH-1:0]	O	This signal is the address for a memory mapped read to the user logic from the DMA.
m_axi_arid [3:0]	O	Standard AXI4 description, which is found in the AXI4 Protocol Specification *AMBA AXI4-Stream Protocol Specification* (ARM IHI 0051A).
m_axi_aruser[31:0]	O	m_axi_aruser[18:0] = reserved m_axi_aruser[31:19] = queue number
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	Protection type.
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	Lock type.
m_axi_arcache[3:0]	O	Memory type.

Table 54: **AXI4 Memory Mapped DMA Read Address Interface Signals** *(cont'd)*

Signal Name	Direction	Description
m_axi_arburst[1:0]	O	Master read burst type.

Table 55: **AXI4 Memory Mapped DMA Read Interface Signals**

Signal Name	Direction	Description
m_axi_rdata [C_M_AXI_DATA_WIDTH-1:0]	I	Master read data.
m_axi_rid [3:0]	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 [C_M_AXI_DATA_WIDTH/8-1:0]	I	Master read odd data parity, per byte. This port is enabled only in Data Protection mode.

Table 56: **AXI4 Memory Mapped DMA Write Address Interface Signals**

Signal Name	Direction	Description
m_axi_awaddr [C_M_AXI_ADDR_WIDTH-1:0]	O	This signal is the address for a memory mapped write to the user logic from the DMA.
m_axi_awid[3:0]	O	Master write address ID.
m_axi_awuser[31:0]	O	m_axi_awuser[18:0] = reserved m_axi_awuser[31:19] = queue number
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	Protection type.
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	Lock type.
m_axi_awcache[3:0]	O	Memory type.

Table 57: **AXI4 Memory Mapped DMA Write Interface Signals**

Signal Name	Direction	Description
m_axi_wdata [C_M_AXI_DATA_WIDTH-1:0]	O	Master write data.
m_axi_wlast	O	Master write last.
m_axi_wstrb[31:0]	O	Master write strobe.

Table 57: **AXI4 Memory Mapped DMA 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 [C_M_AXI_DATA_WIDTH/8-1:0]	O	Master write user. m_axi_wuser[C_M_AXI_DATA_WIDTH/8-1:0] = write data odd parity, per byte. This port is enabled only in Data Protection mode.

Table 58: **AXI4 Memory Mapped DMA 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[3:0]	I	Master response ID.
m_axi_bready	O	Master response ready.

AXI4-Stream H2C Ports

Table 59: **AXI4-Stream H2C Port Descriptions**

Port Name	I/O	Description
m_axis_h2c_tdata [AXI_DATA_WIDTH-1:0]	O	Data output for H2C AXI4-Stream.
m_axis_h2c_tcrc [31:0]	O	32-bit CRC value for that beat. IEEE 802.3 CRC-32 Polynomial
m_axis_h2c_tuser_qid[10:0]	O	Queue ID
m_axis_h2c_tuser_port_id[2:0]	O	Port ID
m_axis_h2c_tuser_err	O	If set, indicates the packet has an error. The error could come from the PCIe, or the error could be in the DMA transfer. Xilinx recommends that you look at the error registers and context for details. When the DMA first detects the error, the error bit will be set to 1. And the error bit will be set for the remainder of that packet. The error bit will be reset to 0 for the next packet if there are no errors in that packet.
m_axis_h2c_tuser_mdata[31:0]	O	Metadata In internal mode, QDMA passes the lower 32 bits of the H2C AXI4-Stream descriptor on this field.
m_axis_h2c_tuser_mty[5:0]	O	The number of bytes that are invalid on the last beat of the transaction. This field is 0 for a 64B transfer.
m_axis_h2c_tuser_zero_byte	O	When set, it indicates that the current beat is an empty beat (zero bytes are being transferred).
m_axis_h2c_tvalid	O	Valid
m_axis_h2c_tlast	O	Indicates that this is the last cycle of the packet transfer
m_axis_h2c_tready	I	Ready

AXI4-Stream C2H Ports

Table 60: **AXI4-Stream C2H Port Descriptions**

Port Name	I/O	Description
s_axis_c2h_tdata [AXI_DATA_WIDTH-1:0]	I	It supports 4 data widths: 64 bits, 128 bits, 256 bits, and 512 bits. Every C2H data packet has a corresponding C2H completion packet.
s_axis_c2h_tcrc [31:0]	I	32 bit CRC value for that beat. IEEE 802.3 CRC-32 Polynomial IP samples CRC value only when s_axis_c2h_tlast is asserted.
s_axis_c2h_ctrl_len [15:0]	I	Length of the packet. For ZERO byte write, the length is 0. C2H stream packet data length is limited to 31 * c2h buffer size. In older versions (such as 2018.3), C2H stream packet data length was limited to 7 * C2H buffer size. ctrl_len is in bytes and should be valid in first beat of the packet.
s_axis_c2h_ctrl_qid [10:0]	I	Queue ID.
s_axis_c2h_ctrl_has_cmpt	I	1'b1: The data packet has a completion. 1'b0: The data packet doesn't have a completion.
s_axis_c2h_ctrl_marker	I	Marker message used for making sure pipeline is completely flushed. After that, you can safely do queue invalidation.
s_axis_c2h_ctrl_port_id [2:0]	I	Port ID.
s_axis_c2h_ctrl_ecc[6:0]	I	Sideband protection for C2H control signals. Output of the Xilinx® Error Correction Code (ECC) core. ECC IP input is described below.
s_axis_c2h_mty [5:0]	I	Empty byte should be set in last beat.
s_axis_c2h_tvalid	I	Valid.
s_axis_c2h_tlast	I	Indicate last packet.
s_axis_c2h_tready	O	Ready.

To generate ECC signals for C2H control bus s_axis_c2h_ctrl_ecc[6:0], use Xilinx Error Correction Code IP. Signals that are used are listed below.

Input to ECC IP using ecc_gen_datain[56:0]

```
assign ecc_gen_datain[56:0] = { 24'h0, //reserved
                    s_axis_c2h_ctrl_has_cmpt_int, //has compt
                    s_axis_c2h_ctrl_marker_int, //marker
                    s_axis_c2h_ctrl_port_id, //port_id
                    1'b0, // reserved should be set to 0.
                    s_axis_c2h_ctrl_qid_int, // Qid
                    s_axis_c2h_ctrl_len_int}; //length
```

AXI4-Stream C2H Completion Ports

Table 61: **AXI4-Stream C2H Completion Port Descriptions**

Port Name	I/O	Description
s_axis_c2h_cmpt_tdata[511:0]	I	Completion data from the user application. This contains information that is written to the completion ring in the host.
s_axis_c2h_cmpt_size [1:0]	I	00: 8B completion. 01: 16B completion. 10: 32B completion. 11: 64B completion
s_axis_c2h_cmpt_dpar [15:0]	I	Odd parity computed as bit per 32b. s_axis_c2h_cmpt_dpar[0] is parity over s_axis_c2h_cmpt_tdata[31:0]. s_axis_c2h_cmpt_dpar[1] is parity over s_axis_c2h_cmpt_tdata[63:31] and so on.
s_axis_c2h_cmpt_ctrl_qid[10:0]	I	Completion queue ID.
s_axis_c2h_cmpt_ctrl_marker	I	Marker message used for making sure pipeline is completely flushed. After that, you can safely do queue invalidation.
s_axis_c2h_cmpt_ctrl_user_trig	I	User can trigger the interrupt and the status descriptor write if they are enabled.
s_axis_c2h_cmpt_ctrl_cmpt_type[1:0]	I	2’b00: NO_PLD_NO_WAIT. The CMPT packet does not have a corresponding payload packet, and it does not need to wait. 2’b01: NO_PLD_BUT_WAIT. The CMPT packet does not have a corresponding payload packet; however, it still needs to wait for the payload packet to be sent before sending the CMPT packet. 2’b10: RSVD. 2’b11: HAS_PLD. The CMPT packet has a corresponding payload packe, and it needs to wait for the payload packet to be sent before sending the CMPT packet.
s_axis_c2h_cmpt_ctrl_wait_pld_pkt_id[15:0]	I	The data payload packet ID that the CMPT packet needs to wait for before it can be sent.
s_axis_c2h_cmpt_ctrl_port_id[2:0]	I	Port ID.
s_axis_c2h_cmpt_ctrl_col_idx[2:0]	I	Color index that defines if the user wants to have the color bit in the CMPT packet and the bit location of the color bit if present.
s_axis_c2h_cmpt_ctrl_err_idx[2:0]	I	Error index that defines if the user wants to have the error bit in the CMPT packet and the bit location of the error bit if present.
s_axis_c2h_cmpt_ctrl_no_wrb_marker	I	Disables CMPT packet during Marker transfer. 1'b0 : CMPT packets are sent to CMPT ring 1'b1 : CMPT packets are not sent to CMPT ring.
s_axis_c2h_cmpt_tvalid	I	Valid. s_axis_c2h_cmpt_tvalid must be asserted until s_axis_c2h_cmpt_tready is asserted.
s_axis_c2h_cmpt_tready	O	Ready.

AXI4-Stream Status Ports

Table 62: **AXI-ST C2H Status Port Descriptions**

Port Name	I/O	Description
axis_c2h_status_valid	O	Valid per descriptor.

Table 62: **AXI-ST C2H Status Port Descriptions** *(cont'd)*

Port Name	I/O	Description
axis_c2h_status_qid [10:0]	O	QID of the packet.
axis_c2h_status_drop	O	The QDMA drops the packet if it does not have enough descriptors to transfer the full packet to the host. This bit indicates if the packet was dropped or not. A packet that is not dropped is considered as having been accepted. 0: Packet is not dropped. 1: Packet is dropped.
axis_c2h_status_last	O	Last descriptor.
axis_c2h_status_cmp	O	0: Dropped packet or C2H packet with has_cmpt of 1'b0. 1: C2H packet that has completions.
axis_c2h_status_error	O	When axis_c2h_status_error is set to 1, the descriptor fetched has an error. When set to 0, there is no error.

AXI4-Stream C2H Write Completion Ports

Table 63: **AXI-ST C2H Write Completion Port Descriptions**

Port Name	I/O	Description
axis_c2h_dmawr_cmp	O	This signal is asserted when the last data payload write request of the packet gets the write completion. It is one pulse per packet.

VDM Ports

Table 64: **VDM Port Descriptions**

Port Name	I/O	Description
st_rx_msg_valid	O	Valid
st_rx_msg_data[31:0]	O	Beat 1: {REQ_ID[15:0], VDM_MSG_CODE[7:0], VDM_MSG_ROUTING[2:0], VDM_DW_LENGTH[4:0]} Beat 2: VDM Lower Header [31:0] or {(Payload_length=0), VDM Higher Header [31:0]} Beat 3 to Beat <n>: VDM Payload
st_rx_msg_last	O	Indicate the last beat
st_rx_msg_rdy	I	Ready. ***Note*:** When this interface is not used, Ready must be tied-off to 1.

RX Vendor Defined Messages are stored in shallow FIFO before they are transmitted to output ports. When there are many back to back VDM messages, FIFO overflows and these messages are dropped. It is best to repeat VDM messages at regular intervals.

Configuration Extend Interface Ports

The Configuration Extend interface allows the core to transfer configuration information with the user application when externally implemented configuration registers are implemented.

Table 65: **Configuration Extend Interface Port Descriptions**

Port Name	I/O	Width	Description
cfg_ext_read_received	O	1	Configuration Extend Read Received The core asserts this output when it has received a configuration read request from the link. Set when PCI Express Extended Configuration Space Enable is selected in the user defined configuration Capabilities tab in in the Vivado® IDE. • All received configuration reads with cfg_ext_register_number in the range of 0xb0-0xbf is considered to be PCIe Legacy Extended Configuration Space. • All received configuration reads with cfg_ext_register_number in the range of 0x120-13F is considered to be PCIe Extended Configuration Space. • All received configuration reads regardless of their address will be indicated by 1 cycle assertion of cfg_ext_read_received. Valid data is driven on cfg_ext_register_number and cfg_ext_function_number. • Only received configuration reads within the two aforementioned ranges need to be responded by the user application outside of the IP.
cfg_ext_write_received	O	1	Configuration Extend Write Received The core asserts this output when it has received a configuration write request from the link. Set when PCI Express Extended Configuration Space Enable is selected in Capabilities tab in the Vivado IDE. • Data corresponding to all received configuration writes with cfg_ext_register_number in the range 0xb0-0xbf is presented on cfg_ext_register_number, cfg_ext_function_number, cfg_ext_write_data and cfg_ext_write_byte_enable. • All received configuration writes with cfg_ext_register_number in the range 0x120-13F is presented on cfg_ext_register_number, cfg_ext_function_number, cfg_ext_write_data and cfg_ext_write_byte_enable.
cfg_ext_register_number	O	10	Configuration Extend Register Number The 10-bit address of the configuration register being read or written. The data is valid when cfg_ext_read_received or cfg_ext_write_received is High.
cfg_ext_function_number	O	8	Configuration Extend Function Number. The 8-bit function number corresponding to the configuration read or write request. The data is valid when cfg_ext_read_received or cfg_ext_write_received is High.
cfg_ext_write_data	O	32	Configuration Extend Write Data Data being written into a configuration register. This output is valid when cfg_ext_write_received is High.

Table 65: **Configuration Extend Interface Port Descriptions** *(cont'd)*

Port Name	I/O	Width	Description
cfg_ext_write_byte_enable	O	4	Configuration Extend Write Byte Enable Byte enables for a configuration write transaction.
cfg_ext_read_data	I	32	Configuration Extend Read Data You can provide data from an externally implemented configuration register to the core through this bus. The core samples this data on the next positive edge of the clock after it sets cfg_ext_read_received High, if you have set cfg_ext_read_data_valid.
cfg_ext_read_data_valid	I	1	Configuration Extend Read Data Valid The user application asserts this input to the core to supply data from an externally implemented configuration register. The core samples this input data on the next positive edge of the clock after it sets cfg_ext_read_received High. The core expects the assertions of this signal within 262144 ('h4_0000) clock cycles of user clock after receiving the read request on cfg_ext_read_received signal. If no response is received by this time, the core will send auto-response with 'h0 payload, and the user application must discard the response and terminate that particular request immediately

FLR Ports

Table 66: **FLR Port Descriptions**

Port Names	I/O	Description
usr_flr_fnc [7:0]	O	Function The function number of the FLR status change.
usr_flr_set	O	Set Asserted for 1 cycle indicating that the FLR status of the function indicated on usr_flr_fnc[7:0] is active.
usr_flr_clr	O	Clear Asserted for 1 cycle indicating that the FLR status of the function indicated on usr_flr_fnc[7:0] is completed.
usr_flr_done_fnc [7:0]	I	Done Function The function for which FLR has been completed by user logic.
usr_flr_done_vld	I	Done Valid Assert for one cycle to signal that FLR for the function on usr_flr_done_fnc[7:0] has been completed.

QDMA Descriptor Bypass Input Ports

Table 67: **QDMA H2C-Streaming Bypass Input Port Descriptions**

Port Name	I/O	Description
h2c_byp_in_st_addr [63:0]	I	64-bit starting address of the DMA transfer.
h2c_byp_in_st_len [15:0]	I	The number of bytes to transfer.

Table 67: **QDMA H2C-Streaming Bypass Input Port Descriptions** *(cont'd)*

Port Name	I/O	Description
h2c_byp_in_st_sop	I	Indicates start of packet. Set for the first descriptor. Reset for the rest of the descriptors.
h2c_byp_in_st_eop	I	Indicates end of packet. Set for the last descriptor. Reset for the rest of the descriptors.
h2c_byp_in_st_sdi	I	H2C Bypass In Status Descriptor/Interrupt If set, it is treated as an indication from the user application to the QDMA to send the status descriptor to host, and to generate an interrupt to host when the QDMA has fetched the last byte of the data associated with this descriptor. The QDMA honors the request to generate an interrupt only if interrupts have been enabled in the H2C SW context for this QID and armed by the driver. This can only be set for an EOP descriptor. QDMA will hang if the last descriptor without h2c_byp_in_st_sdi has an error. This results in a missing writeback and hw_ctxt.dsc_pend bit that are asserted indefinitely. The workaround is to send a zero length descriptor to trigger the Completion (CMPT) Status. **RECOMMENDED:** *For performance reasons, Xilinx recommends that this port be asserted once in 32 or 64 descriptors and assert at the last descriptor if there are no more descriptors left.*
h2c_byp_in_st_mrkr_req	I	H2C Bypass In Marker Request When set, the descriptor passes through the H2C Engine pipeline and once completed, produces a marker response on the H2C Streaming Bypass-Out interface. This can only be set for an EOP descriptor.
h2c_byp_in_st_no_dma	I	H2C Bypass In No DMA When sending in a descriptor through this interface with this signal asserted, it informs the QDMA to not send any PCIe requests for this descriptor. Because no PCIe request is sent out, no corresponding DMA data is issued on the H2C Streaming output interface. This is typically used in conjunction with h2c_byp_in_st_sdi to cause Status Descriptor/Interrupt when the user logic is out of the actual descriptors and still wants to drive the h2c_byp_in_st_sdi signal. If h2c_byp_in_st_mrkr_req and h2c_byp_in_st_sdi are reset when sending in a no-DMA descriptor, the descriptor is treated as a NOP and is completely consumed inside the QDMA without any interface activity. If h2c_byp_in_st_no_dma is set, then both h2c_byp_in_st_sop and h2c_byp_in_st_eop must be set. If h2c_byp_in_st_no_dma is set, the QDMA ignores the address and length fields of this interface.
h2c_byp_in_st_qid [10:0]	I	The QID associated with the H2C descriptor ring.
h2c_byp_in_st_error	I	This bit can be set to indicate an error for the queue. The descriptor will not be processed. Context will be updated to reflect an error in the queue.
h2c_byp_in_st_func [7:0]	I	PCIe function ID
h2c_byp_in_st_cidx [15:0]	I	The CIDX that will be used for the status descriptor update and/or interrupt (aggregation mode). Generally the CIDX should be left unchanged from when it was received from the descriptor bypass output interface.
h2c_byp_in_st_port_id [2:0]	I	QDMA port ID

Table 67: **QDMA H2C-Streaming Bypass Input Port Descriptions** *(cont'd)*

Port Name	I/O	Description
h2c_byp_in_st_vld	I	Valid. High indicates descriptor is valid, one pulse for one descriptor.
h2c_byp_in_st_rdy	O	Ready to take in descriptor

Table 68: **QDMA H2C-MM Descriptor Bypass Input Port Descriptions**

Port Name	I/O	Description
h2c_byp_in_mm_radr[63:0]	I	The read address for the DMA data.
h2c_byp_in_mm_wadr[63:0]	I	The write address for the dma data.
h2c_byp_in_mm_no_dma	I	H2C Bypass In No DMA When sending in a descriptor through this interface with this signal asserted, this signal informs the QDMA to not send any PCIe requests for this descriptor. Because no PCIe request is sent out, no corresponding DMA data is issued on the H2C MM output interface. This is typically used in conjunction with h2c_byp_in_mm_sdi to cause Status Descriptor/Interrupt when the user logic is out of the actual descriptors and still wants to drive the h2c_byp_in_mm_sdi signal. If h2c_byp_in_mm_mrkr_req and h2c_byp_in_mm_sdi are reset when sending in a no-DMA descriptor, the descriptor is treated as a No Operation (NOP) and is completely consumed inside the QDMA without any interface activity. If h2c_byp_in_mm_no_dma is set, the QDMA ignores the address. The length field should be set to 0.
h2c_byp_in_mm_len[27:0]	I	The DMA data length. The upper 12 bits must be tied to 0. Thus only the lower 16 bits of this field can be used for specifying the length.
h2c_byp_in_mm_sdi	I	H2C-MM Bypass In Status Descriptor/Interrupt If set, it is treated as an indication from the User to QDMA to send the status descriptor to host and generate an interrupt to host when the QDMA has fetched the last byte of the data associated with this descriptor. The QDMA will honor the request to generate an interrupt only if interrupts have been enabled in the H2C ring context for this QID and armed by the driver. QDMA will hang if the last descriptor without h2c_byp_in_mm_sdi has an error. This results in a missing writeback and hw_ctxt.dsc_pend bit that are asserted indefinitely. The workaround is to send a zero length descriptor to trigger the Completion (CMPT) Status. **RECOMMENDED:** *For performance reasons, Xilinx recommends that this port be asserted once in 32 or 64 descriptors and be asserted at the last descriptor if there are no more descriptors left.*
h2c_byp_in_mm_mrkr_req	I	H2C-MM Bypass In Marker Request Indication from the User that the QDMA must send a completion status to the User once the QDMA has completed the data transfer of this descriptor.
h2c_byp_in_mm_qid [10:0]	I	The QID associated with the H2C descriptor ring.
h2c_byp_in_mm_error	I	This bit can be set to indicate an error for the queue. The descriptor will not be processed. Context will be updated to reflect and error in the queue.

Table 68: **QDMA H2C-MM Descriptor Bypass Input Port Descriptions** *(cont'd)*

Port Name	I/O	Description
h2c_byp_in_mm_func [7:0]	I	PCIe function ID
h2c_byp_in_mm_cidx [15:0]	I	The CIDX that will be used for the status descriptor update and/or interrupt (aggregation mode). Generally the CIDX should be left unchanged from when it was received from the descriptor bypass output interface.
h2c_byp_in_mm_port_id [2:0]	I	QDMA port ID
h2c_byp_in_mm_vld	I	Valid. High indicates descriptor is valid, one pulse for one descriptor.
h2c_byp_in_mm_rdy	O	Ready to take in descriptor

Table 69: **QDMA C2H-Streaming Bypass Input Port Descriptions**[1]

Port Name	I/O	Description
c2h_byp_in_st_csh_addr [63:0]	I	64 bit address where DMA writes data.
c2h_byp_in_st_csh_qid [10:0]	I	The QID associated with the C2H descriptor ring.
c2h_byp_in_st_csh_error	I	This bit can be set to indicate an error for the queue. The descriptor will not be processed. Context will be updated to reflect and error in the queue.
c2h_byp_in_st_csh_func [7:0]	I	PCIe function ID
c2h_byp_in_st_csh_port_id[2:0]	I	QDMA port ID
c2h_byp_in_st_csh_pfch_tag[6:0]	I	Prefetch tag. The prefetch tag points to the cam that stores the active queues in the prefetch engine. In Cache Bypass mode, you must loop back `c2h_byp_out_pfch_tag[6:0]` to `c2h_byp_in_st_csh_pfch_tag[6:0]`. In Simple Bypass mode, used need to pass in the Prefetch tag value from MDMA_C2H_PFCH_BYP_TAG (0x140C) register.
c2h_byp_in_st_csh_vld	I	Valid. High indicates descriptor is valid, one pulse for one descriptor.
c2h_byp_in_st_csh_rdy	O	Ready to take in descriptor.

Notes:

1. AXI-Stream C2H Simple Bypass mode and Cache Bypass mode both use the same bypass ports, c2h_byp_in_st_csh_*.

Table 70: **QDMA C2H-MM Descriptor Bypass Input Port Descriptions**

Port Name	I/O	Description
c2h_byp_in_mm_raddr [63:0]	I	The read address for the DMA data.
c2h_byp_in_mm_wadr[63:0]	I	The write address for the DMA data.

Table 70: **QDMA C2H-MM Descriptor Bypass Input Port Descriptions** *(cont'd)*

Port Name	I/O	Description
c2h_byp_in_mm_no_dma	I	C2H Bypass In No DMA When sending in a descriptor through this interface with this signal asserted, this signal informs the QDMA to not send any PCIe requests for this descriptor. Because no PCIe request is sent out, no corresponding DMA data is read from C2H MM interface. This is typically used in conjunction with c2h_byp_in_mm_sdi to cause Status Descriptor/Interrupt when the user logic is out of the actual descriptors and still wants to drive the c2h_byp_in_mm_sdi signal. If c2h_byp_in_mm_mrkr_req and c2h_byp_in_mm_sdi are reset when sending in a no-DMA descriptor, the descriptor is treated as a NOP and is completely consumed inside the QDMA without any interface activity. If c2h_byp_in_mm_no_dma is set, the QDMA ignores the address. The length field should be set to 0.
c2h_byp_in_mm_len[27:0]	I	The DMA data length
c2h_byp_in_mm_sdi	I	C2H Bypass In Status Descriptor/Interrupt If set, it is treated as an indication from the User to QDMA to send the status descriptor to host, and generate an interrupt to host when the QDMA has fetched the last byte of the data associated with this descriptor. The QDMA will honor the request to generate an interrupt only if interrupts have been enabled in the C2H ring context for this QID and armed by the driver. **RECOMMENDED:** *For performance reasons, Xilinx recommends that this port be asserted once in 32 or 64 descriptors, and asserted at the last descriptor if there are no more descriptors left.*
c2h_byp_in_mm_mrkr_req	I	C2H Bypass In Marker Request Indication from the User that the QDMA must send a completion status to the User once the QDMA has completed the data transfer of this descriptor.
c2h_byp_in_mm_qid [10:0]	I	The QID associated with the C2H descriptor ring
c2h_byp_in_mm_error	I	This bit can be set to indicate an error for the queue. The descriptor will not be processed. Context will be updated to reflect and error in the queue.
c2h_byp_in_mm_func [7:0]	I	PCIe function ID
c2h_byp_in_mm_cidx [15:0]	I	The User must echo the CIDX from the descriptor that it received on the bypass-out interface.
c2h_byp_in_mm_port_id[2:0]	I	QDMA port ID
c2h_byp_in_mm_vld	I	Valid. High indicates descriptor is valid, one pulse for one descriptor.
c2h_byp_in_mm_rdy	O	Ready to take in descriptor.

QDMA Descriptor Bypass Output Ports

Table 71: **QDMA H2C Descriptor Bypass Output Port Descriptions**

Port Name	I/O	Description
h2c_byp_out_dsc [255:0]	O	The H2C descriptor fetched from the host. For H2C AXI-MM, the subsystem uses all 256 bits, and the structure of the bits are the same as this table. For H2C AXI-ST, the subsystem uses [127:0] bits, and the structure of the bits are the same as this table.
h2c_byp_out_st_mm	O	Indicates whether this is a streaming data descriptor or memory-mapped descriptor. 0: streaming 1: memory-mapped
h2c_byp_out_dsc_sz [1:0]	O	Descriptor size. This field indicates size of the descriptor. 0: 8B 1: 16B 2: 32B 3: 64B - 64B descriptors will be transferred with two valid/ready cycles. The first cycle has the least significant 32 bytes. The second cycle has the most significant 32 bytes. CIDX and other queue information is valid only on the second beat of a 64B descriptor .
h2c_byp_out_qid [10:0]	O	The QID associated with the H2C descriptor ring.
h2c_byp_out_error	O	Indicates that an error was encountered in descriptor fetch or execution of a previous descriptor.
h2c_byp_out_func [7:0]	O	PCIe function ID
h2c_byp_out_cidx [15:0]	O	H2C Bypass Out Consumer Index The ring index of the descriptor fetched. The User must echo this field back to QDMA when submitting the descriptor on the bypass-in interface.
h2c_byp_out_port_id [2:0]	O	QDMA port ID
h2c_byp_out_fmt[2:0]	O	Format Tthe encoding for this field is as follows. 0x0: Standard descriptor 0x1 - 0x7: Reserved
h2c_byp_out_vld	O	Valid. High indicates descriptor is valid, one pulse for one descriptor.
h2c_byp_out_rdy	I	Ready. When this interface is not used, Ready must be tied-off to 1.

Table 72: **QDMA C2H Descriptor Bypass Output Port Descriptions**

Port Name	I/O	Description
c2h_byp_out_dsc [255:0]	O	The C2H descriptor fetched from the host. For C2H AXI-MM, the subsystem uses all 256 bits, and the structure of the bits is the same as this table. For C2H AXI-ST, the subsystem uses [63:0] bits, and the structure of the bits is the same as this table. The remaining bits are ignored.

Table 72: **QDMA C2H Descriptor Bypass Output Port Descriptions** *(cont'd)*

Port Name	I/O	Description
c2h_byp_out_st_mm	O	Indicates whether this is a streaming data descriptor or memory-mapped descriptor. 0: streaming 1: memory-mapped
c2h_byp_out_dsc_sz [1:0]	O	Descriptor size. This field indicates the amount of valid descriptor information on h2c_byp_out_dsc. 0: 8B 1: 16B 2: 32B 3: 64B - 64B descriptors will be transferred with two valid/ready cycles. The first cycle has the least significant 32 bytes. The second cycle has the most significant 32 bytes. CIDX and other queue information is valid only on the second beat of a 64B descriptor.
c2h_byp_out_qid [10:0]	O	The QID associated with the H2C descriptor ring.
c2h_byp_out_error	O	Indicates that an error was encountered in descriptor fetch or execution of a previous descriptor.
c2h_byp_out_func [7:0]	O	PCIe function ID.
c2h_byp_out_cidx [15:0]	O	C2H Bypass Out Consumer Index The ring index of the descriptor fetched. The User must echo this field back to QDMA when submitting the descriptor on the bypass-in interface.
c2h_byp_out_port_id [2:0]	O	QDMA port ID
c2h_byp_out_pfch_tag[6:0]	O	Prefetch tag. The prefetch tag points to the cam that stores the active queues in prefetch engine
c2h_byp_out_fmt[2:0]	O	Format The encoding for this field is as follows. 0x0 : Standard descriptor 0x1 - 0x7 : Reserved
c2h_byp_out_vld	O	Valid. High indicates descriptor is valid, one pulse for one descriptor.
c2h_byp_out_rdy	I	Ready. When this interface is not used, Ready must be tied-off to 1.

It is common for `h2c_byp_out_vld` or `c2h_byp_out_vld` to be asserted with the CIDX value; this occurs when the Descriptor bypass mode option is not set in the context programming selection. You must set the Descriptor bypass mode during QDMA IP core customization in the Vivado® IDE to see descriptor bypass output ports. When Descriptor bypass option is selected in the Vivado® IDE but the descriptor bypass bit is not set in context programming, you will see valid signals getting asserted with CIDX updates.

QDMA Descriptor Credit Input Ports

Table 73: **QDMA Descriptor Credit Input Port Descriptions**

Port Name	I/O	Description
dsc_crdt_in_vld	I	Valid. When asserted the user must be presenting valid data on the bus and maintain the bus values until both valid and ready are asserted on the same cycle.
dsc_crdt_in_rdy	O	Ready. Assertion of this signal indicates the DMA is ready to accept data from this bus.
dsc_crdt_in_dir	I	Indicates whether credits are for H2C or C2H descriptor ring. 0: H2C 1: C2H
dsc_crdt_in_fence	I	If the fence bit is set, the credits are not coalesced, and the queue is guaranteed to generate a descriptor fetch before subsequent credit updates are processed. The fence bit should only be set for a queue that is enabled, and has both descriptors and credits available, otherwise a hang condition might occur.
dsc_crdt_in_qid [10:0]	I	The QID associated with the descriptor ring for the credits are being added.
dsc_crdt_in_crdt [15:0]	I	The number of descriptor credits that the user application is giving to QDMA to fetch descriptors from the host.

QDMA Traffic Manager Credit Output Ports

Table 74: **QDMA TM Credit Output Port Descriptions**

Port Name	I/O	Description
tm_dsc_sts_vld	O	Valid. Indicates valid data on the output bus. Valid data on the bus is held until tm_dsc_sts_rdy is asserted by the user.
tm_dsc_sts_rdy	I	Ready. Assertion indicates that the user logic is ready to accept the data on this bus. When this interface is not used, Ready must be tied-off to 1. ***Note*:** When this interface is not used, Ready must be tied-off to 1.
tm_dsc_sts_byp	O	Shows the bypass bit in the SW descriptor context
tm_dsc_sts_dir	O	Indicates whether the status update is for a H2C or C2H descriptor ring. 0: H2C 1: C2H
tm_dsc_sts_mm	O	Indicates whether the status update is for a streaming or memory-mapped queue. 0: streaming 1: memory-mapped
tm_dsc_sts_qid [10:0]	O	The QID of the ring
tm_dsc_sts_avl [15:0]	O	If tm_dsc_sts_qinv is set, this is the number of credits available in the descriptor engine. If tm_dsc_sts_qinv is not set this is the number of new descriptors that have been posted to the ring since the last time this update was sent.

Table 74: **QDMA TM Credit Output Port Descriptions** *(cont'd)*

Port Name	I/O	Description
tm_dsc_sts_qinv	O	If set, it indicates that the queue has been invalidated. This is used by the user application to reconcile the credit accounting between the user application and QDMA.
tm_dsc_sts_qen	O	The current queue enable status.
tm_dsc_sts_irq_arm	O	If set, it indicates that the driver is ready to accept interrupts
tm_dsc_sts_error	O	Set to 1 if the PIDX update is rolled over the current CIDX of associated queue.
tm_dsc_sts_pidx[15:0]	O	PIDX of the Queue
tm_dsc_sts_port_id [2:0]	O	The port id associated with the queue from the queue context.

User Interrupts

Table 75: **User Interrupts Port Descriptions**

Port Name	I/O	Description
usr_irq_in_vld	I	Valid An assertion indicates that an interrupt associated with the vector, function, and pending fields on the bus should be generated to PCIe. Once asserted, Usr_irq_in_vld must remain high until usr_irq_out_ack is asserted by the DMA.
usr_irq_in_vec [11:0]	I	Vector The MSIX vector to be sent. Vector starts from 0 to 7. Vector 0 is the first vector.
usr_irq_in_fnc [7:0]	I	Function The function of the vector to be sent.
usr_irq_out_ack	O	Interrupt Acknowledge An assertion of the acknowledge bit indicates that the interrupt was transmitted on the link the user logic must wait for this pulse before signaling another interrupt condition.
usr_irq_out_fail	O	Interrupt Fail An assertion of fail indicates that the interrupt request was aborted before transmission on the link.

Eight vectors is the maximum allowed per function.

Queue Status Ports

Table 76: **Queue Status Ports**

Port Name	I/O	Description
qsts_out_op[7:0]	O	Opcode This indicates the type of packet being issued. Encoding of this field is as follows. 0x0: CMPT Marker Response 0x1: H2C-ST Marker Response 0x2: C2H-MM Marker Response 0x3: H2C-MM Marker Response 0x4-0xff: reserved
qsts_out_data[63:0]	O	The data field for the individual opcodes are defined in the tables below.
qsts_out_port_id[2:0]	O	Port ID
qsts_out_qid[11:0]	O	Queue ID
qsts_out_vld	O	Queue status valid
qsts_out_rdy	I	Queue status ready. Ready must be tied to 1 so status output will not be blocked. Even if this interface is not used, the ready port must be tied to 1.

Table 77: **Queue status data**

qsts_out_data	Field	Description
[1:0]	err	Error code reported by the CMPT Engine. 0: No error 1: SW gave bad Completion CIDX update 2: Descriptor error received while processing the C2H packet 3: Completion dropped by the C2H Engine because Completion Ring was full
[2]	retry_marker_req	An Interrupt could not be generated in spite of being enabled. This happens when an Interrupt is already outstanding on the queue when the marker request was received. The user logic must wait and retry the marker request again if an Interrupt is desired to be sent.
[25:3]	marker_cookie	When the CMPT Engine sends a marker to the Queues status port interface, it sends the lower 23 bits of the CMPT as part of the marker response on the Queues status port interface. Thus the user logic can place a 23-bit value in the CMPT when making the marker request and it will receive the same 23 bits with the marker response. When the marker is generated as a result of an error that the CMPT Engine encountered (as opposed to a marker request made by the user logic), then this 23-bit field is not valid. ***Note*:** Even if the user has enabled stamping of error and/or color bits in the CMPT writes to the host, the marker_cookie does not contain them. It is exactly the lower 23 bits of the CMPT that the user logic provided to the QDMA when making the marker request.

Table 77: **Queue status data** *(cont'd)*

qsts_out_data	Field	Description
[26]	is_mrkr_rsp	This bit will be set to 1 if the marker response is based on marker request. If this bit is set to '0' marker response is based on errors.
[63:27]	rsv	Reserved

Register Space

This section provides register space information for the QDMA.

In register space descriptions, configuration register attributes are defined as follows:

- **NA:** Reserved
- **RO:** Read-Only - Register bits are read-only and cannot be altered by the software.
- **RW:** Read-Write - Register bits are read-write and are permitted to be either Set or Cleared by the software to the desired state.
- **RW1C:** Write-1-to-clear-status - Register bits indicate status when read. A Set bit indicates a status event which is Cleared by writing a 1b. Writing a 0b to RW1C bits has no effect.
- **W1C:** Non-readable-write-1-to-clear-status - Register will return 0 when read. Writing 1b Clears the status for that bit index. Writing a 0b to W1C bits has no effect.
- **W1S:** Non-readable-write-1-to-set - Register will return 0 when read. Writing 1b Sets the control set for that bit index. Writing a 0b to W1S bits has no effect.

QDMA PF Address Register Space

All the physical function (PF) registers are listed in the `qdma_v4_0_pf_registers.csv` available in the Register Reference File.

TIP: *When you generate the IP in default mode, not all registers are exposed. For example, debug registers will be missing. Refer to the* `qdma_v4_0_pf_registers.csv` *file to identify the debug registers. To expose all registers, use the following tcl command during IP generation:*

```
set_property CONFIG.debug_mode {DEBUG_REG_ONLY} [get_ips qdma_0]
```

Table 78: **QDMA PF Address Register Space**

Register Name	Base (Hex)	Byte Size (Dec)	Register List and Details
QDMA_CSR	0x0000	9216	QDMA Configuration Space Register (CSR) found in `qdma_v4_0_pf_registers.csv`.
QDMA_TRQ_SEL_QUEUE_PF	0x18000	32768	Also found in QDMA_TRQ_SEL_QUEUE_PF (0x18000).

Table 78: **QDMA PF Address Register Space** *(cont'd)*

Register Name	Base (Hex)	Byte Size (Dec)	Register List and Details
QDMA_PF_MAILBOX	0x22400	16384	Also found in QDMA_PF_MAILBOX (0x22400).
QDMA_TRQ_MSIX	0x30000	32768	Also found in QDMA_TRQ_MSIX (0x30000).

QDMA_CSR (0x0000)

QDMA Configuration Space Register (CSR) descriptions are accessible in `qdma_v4_0_pf_registers.csv` available in the Register Reference File.

QDMA_TRQ_SEL_QUEUE_PF (0x18000)

Table 79: **QDMA_TRQ_SEL_QUEUE_PF (0x18000) Register Space**

Register	Address	Description
QDMA_DMAP_SEL_INT_CIDX[2048] (0x18000)	0x18000-0x1CFF0	Interrupt Ring Consumer Index (CIDX)
QDMA_DMAP_SEL_H2C_DSC_PIDX[2048] (0x18004)	0x18004-0x1CFF4	H2C Descriptor Producer index (PIDX)
QDMA_DMAP_SEL_C2H_DSC_PIDX[2048] (0x18008)	0x18008-0x1CFF8	C2H Descriptor Producer Index (PIDX)
QDMA_DMAP_SEL_CMPT_CIDX[2048] (0x1800C)	0x1800C-0x1CFFC	C2H Completion Consumer Index (CIDX)

There are 2048 Queues, each Queue will have more than four registers. All these registers can be dynamically updated at any time. This set of registers can be accessed based on the Queue number.

> Queue number is absolute *Qnumber* [0 to 2047].
> Interrupt CIDX address = 0x18000 + Qnumber*16
> H2C PIDX address = 0x18004 + Qnumber*16
> C2H PIDX address = 0x18008 + Qnumber*16
> Write Back CIDX address = 0x1800C + Qnumber*16

For Queue 0:

> 0x18000 correspond to QDMA_DMAP_SEL_INT_CIDX
> 0c18004 correspond to QDMA_DMAP_SEL_H2C_DSC_PIDX
> 0x18008 correspond to QDMA_DMAP_SEL_C2H_DSC_PIDX
> 0x1800C correspond to QDMA_DMAP_SEL_CMPT_CIDX

For Queue 1:

> 0x18010 correspond to QDMA_DMAP_SEL_INT_CIDX
> 0c18014 correspond to QDMA_DMAP_SEL_H2C_DSC_PIDX
> 0x18018 correspond to QDMA_DMAP_SEL_C2H_DSC_PIDX

0x1801C correspond to QDMA_DMAP_SEL_CMPT_CIDX

For Queue 2:

0x18020 correspond to QDMA_DMAP_SEL_INT_CIDX
0c18024 correspond to QDMA_DMAP_SEL_H2C_DSC_PIDX
0x18028 correspond to QDMA_DMAP_SEL_C2H_DSC_PIDX
0x1802C correspond to QDMA_DMAP_SEL_CMPT_CIDX

QDMA_DMAP_SEL_INT_CIDX[2048] (0x18000)

Table 80: **QDMA_DMAP_SEL_INT_CIDX[2048] (0x18000)**

Bit	Default	Access Type	Field	Description
[31:24]	0	NA	Reserved	Reserved
[23:16]	0	RW	ring_idx	Ring index of the Interrupt Aggregation Ring
[15:0]	0	RW	sw_cdix	Software Consumer index (CIDX)

QDMA_DMAP_SEL_H2C_DSC_PIDX[2048] (0x18004)

Table 81: **QDMA_DMAP_SEL_H2C_DSC_PIDX[2048] (0x18004)**

Bit	Default	Access Type	Field	Description
[31:17]	0	NA	Reserved	Reserved
[16]	0	RW	irq_arm	Interrupt arm. Set this bit to 1 for next interrupt generation.
[15:0]	0	RW	h2c_pidx	H2C Producer Index

QDMA_DMAP_SEL_C2H_DSC_PIDX[2048] (0x18008)

Table 82: **QDMA_DMAP_SEL_C2H_DSC_PIDX[2048] (0x18008)**

Bit	Default	Access Type	Field	Description
[31:17]	0	NA	Reserved	Reserved
[16]	0	RW	irq_arm	Interrupt arm. Set this bit to 1 for next interrupt generation.
[15:0]	0	RW	c2h_pidx	C2H Producer Index

QDMA_DMAP_SEL_CMPT_CIDX[2048] (0x1800C)

Table 83: **QDMA_DMAP_SEL_CMPT_CIDX[2048] (0x1800C)**

Bit	Default	Access Type	Field	Description
[31:29]	0	NA	Reserved	Reserved
[28]	0	RW	irq_en_wrb	Interrupt arm. Set this bit to 1 for next interrupt generation.
[27]	0	RW	en_sts_desc_wrb	Enable Status Descriptor for CMPT
[26:24]	0	RW	trigger_mode	Interrupt and Status Descriptor Trigger Mode: 0x0: Disabled 0x1: Every 0x2: User_Count 0x3: User 0x4: User_Timer 0x5: User_Timer_Count
[23:20]	0	RW	c2h_timer_cnt_index	Index to QDMA_C2H_TIMER_CNT
[19:16]	0	RW	c2h_count_threshhold	Index to QDMA_C2H_CNT_TH
[15:0]	0	RW	wrb_cidx	CMPT Consumer Index (CIDX)

QDMA_PF_MAILBOX (0x22400)

Table 84: **QDMA_PF_MAILBOX (0x22400) Register Space**

Register	Address	Description
Function Status Register (0x22400)	0x22400	Status bits
Function Command Register (0x22404)	0x22404	Command register bits
Function Interrupt Vector Register (0x22408)	0x22408	Interrupt vector register
Target Function Register (0x2240C)	0x2240C	Target Function register
Function Interrupt Vector Register (0x22410)	0x22410	Interrupt Control Register
RTL Version Register (0x22414)	0x22414	RTL Version Register
PF Acknowledgment Registers (0x22420-0x2243C)	0x22420-0x2243C	PF acknowledge
FLR Control/Status Register (0x22500)	0x22500	FLR control and status
Incoming Message Memory (0x22C00-0x22C7C)	0x22C00-0x22C7C	Incoming message (128 bytes)
Outgoing Message Memory (0x23000-0x2307C)	0x23000-0x2307C	Outgoing message (128 bytes)

Mailbox Addressing

- **PF addressing:** `Addr = PF_Bar_offset + CSR_addr`
- **VF addressing:** `Addr = VF_Bar_offset + VF_Start_offset + VF_offset + CSR_addr`

Function Status Register (0x22400)

Table 85: **Function Status Register (0x22400)**

Bit	Default	Access Type	Field	Description
[31:12]	0	NA	Reserved	Reserved
[11:4]	0	RO	cur_src_fn	This field is for PF use only. The source function number of the message on the top of the incoming request queue.
[2]	0	RO	ack_status	This field is for PF use only. The status bit will be set when any bit in the acknowledgment status register is asserted.
[1]	0	RO	o_msg_status	For VF: The status bit will be set when VF driver write msg_send to its command register. When The associated PF driver send acknowledgment to this VF, the hardware clear this field. The VF driver is not allow to update any content in its outgoing mailbox memory (OMM) while o_msg_status is asserted. Any illegal write to the *OMM* will be discarded (optionally, this can cause an error in the AXI Lite response channel). For PF: The field indicated the message status of the target FN which is specified in the *Target FN Register*. The status bit will be set when PF driver sends msg_send command. When the corresponding function driver send acknowledgment by sending msg_rcv, the hardware clear this field. The PF driver is not allow to update any content in its outgoing mailbox memory (OMM) while o_msg_status(target_fn_id) is asserted. Any illegal write to the *OMM* will be discarded (optionally, case an error in the AXI4L response channel).
[0]	0	RO	i_msg_status	For VF: When asserted, a message in the VF's incoming Mailbox memory is pending for process. The field will be cleared once the VF driver write msg_rcv to its command register. For PF: When asserted, the messages in the incoming Mailbox memory are pending for process. The field will be cleared only when the event queue is empty.

Function Command Register (0x22404)

Table 86: **Function Command Register (0x22404)**

Bit	Default	Access Type	Field	Description
[31:3]	0	NA	Reserved	Reserved
[2]	0	RO	Reserved	Reserved
[1]	0	RW	msg_rcv	For VF: VF marks the message in its Incoming Mailbox Memory as received. Hardware asserts the acknowledgement bit of the associated PF. For PF: PF marks the message send by target_fn as received. The hardware will refresh the i_msg_status of the PF, and clear the o_msg_status of the target_fn.

Table 86: **Function Command Register (0x22404)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[0]	0	RW	msg_send	For VF: VF marks the current message in its own Outgoing Mailbox as valid. For PF: • Current target_fn_id belongs to a VF: PF finished writing a message into the Incoming Mailbox memory of the VF with target_fn_id. The hardware sets the i_msg_status field of the target FN's status register. • Current target_fn_id belongs to a PF: PF finished writing a message into its own outgoing Mailbox memory. Hardware will push the message to the event queue of the PF with target_fn_id.

Function Interrupt Vector Register (0x22408)

Table 87: **Function Interrupt Vector Register (0x22408)**

Bit	Default	Access Type	Field	Description
[31:5]	0	NA	Reserved	Reserved
[4:0]	0	RW	int_vect	5-bit interrupt vector assigned by the driver.

Target Function Register (0x2240C)

Table 88: **Target Function Register (0x2240C)**

Bit	Default	Access Type	Field	Description
[31:8]	0	NA	Reserved	Reserved
[7:0]	0	RW	target_fn_id	This field is for PF use only. The FN number which the current operation is targeting at.

Function Interrupt Vector Register (0x22410)

Table 89: **Function Interrupt Vector Register (0x22410)**

Bit	Default	Access Type	Field	Description
[31:1]	0	NA	Reserved	Reserved
[0]	0	RW	int_en	Interrupt enable.

RTL Version Register (0x22414)

Table 90: **RTL Version Register (0x22414)**

Bit	Default	Access Type	Field	Description
[31:16]	0x1fd3	RO		QDMA ID
[15:0]	0	RO		Vivado versions 0x10: Vivado version 2019.2

PF Acknowledgment Registers (0x22420-0x2243C)

Table 91: **PF Acknowledgment Registers (0x22420-0x2243C)**

Register	Addr	Default	Access Type	Field	Width	Description
Ack0	0x22420	0	RW		32	Acknowledgment from FN 31~0
Ack1	0x22424	0	RW		32	Acknowledgment from FN 63~32
Ack2	0x22428	0	RW		32	Acknowledgment from FN 95~64
Ack3	0x2242C	0	RW		32	Acknowledgment from FN 127~96
Ack4	0x22430	0	RW		32	Acknowledgment from FN 159~128
Ack5	0x22434	0	RW		32	Acknowledgment from FN 191~160
Ack6	0x22438	0	RW		32	Acknowledgment from FN 223~192
Ack7	0x2243C	0	RW		32	Acknowledgment from FN 255~224

FLR Control/Status Register (0x22500)

Table 92: **FLR Control/Status Register (0x22500)**

Bit	Default	Access Type	Field	Description
[31:1]	0	NA	Reserved	Reserved
[0]	0	RW	Flr_status	Software write 1 to initiate the Function Level Reset (FLR) for the associated function. The field is kept asserted during the FLR process. After the FLR is done, the hardware de-asserts this field.

Incoming Message Memory (0x22C00-0x22C7C)

Table 93: **Incoming Message Memory (0x22C00-0x22C7C)**

Register	Addr	Default	Access Type	Field	Width	Description
i_msg_i	0x22C00 + i*4	0	RW		32	The *i*th word of the incoming message (0 ≤ I < 128).

Outgoing Message Memory (0x23000-0x2307C)

Table 94: **Outgoing Message Memory (0x23000-0x2307C)**

Register	Addr	Default	Access Type	Field	Width	Description
o_msg_i	0x23000 + i *4	0	RW		32	The *i*th word of the outgoing message (0 ≤ I < 128).

QDMA_TRQ_MSIX (0x30000)

Table 95: **QDMA_TRQ_MSIX (0x30000)**

Byte Offset	Bit	Default	Access Type	Field	Description
0x30000	[31:0]	0	RW	addr	MSI-X vector0 message lower address. MSIX_Vector0_Address[63:32]
0x30004	[31:0]	0	RW	addr	MSI-X vector0 message upper address. MSIX_Vector0_Address[63:32]
0x30008	[31:0]	0	RW	data	MSIX_Vector0_Data[31:0] MSI-X vector0 message data.
0x3000C	[31:0]	0	RW	control	MSIX_Vector0_Control[31:0] MSI-X vector0 control. Bit Position: 31:1: Reserved. 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.

***Note*:** The table above represents one MSI-X table entry 0. There are 2K MSI-X table entries for the QDMA.

QDMA VF Address Register Space

All the virtual function (VF) registers are listed in the `qdma_v4_0_vf_registers.csv` available in the Register Reference File.

Table 96: **QDMA VF Address Register Space**

Target Name	Base (Hex)	Byte Size (Dec)	Notes
QDMA_TRQ_SEL_QUEUE_VF (0x3000)	00003000	4096	VF Direct QCSR (16B per Queue, up to max of 256Queue per function)
QDMA_TRQ_MSIX_VF (0x4000)	00004000	4096	Space for 32 MSIX vectors and PBA
QDMA_VF_MAILBOX (0x5000)	00005000	8192	Mailbox address space

QDMA_TRQ_SEL_QUEUE_VF (0x3000)

VF functions can access direct update registers per queue with offset (0x3000). The description for this register space is the same as QDMA_TRQ_SEL_QUEUE_PF (0x18000).

This set of registers can be accessed based on Queue number. Queue number is relative Qnumber for that VF.

Interrupt CIDX address = 0x3000 + Qnumber*16
H2C PIDX address = 0x3004 + Qnumber*16
C2H PIDX address = 0x3008 + Qnumber*16
Completion CIDX address = 0x300C + Qnumber*16

For Queue 0:

0x3000 correspond to QDMA_DMAP_SEL_INT_CIDX
0x3004 correspond to QDMA_DMAP_SEL_H2C_DSC_PIDX
0x3008 correspond to QDMA_DMAP_SEL_C2H_DSC_PIDX
0x300C correspond to QDMA_DMAP_SEL_WRB_CIDX

For Queue 1:

0x3010 correspond to QDMA_DMAP_SEL_INT_CIDX
0x3014 correspond to QDMA_DMAP_SEL_H2C_DSC_PIDX
0x3018 correspond to QDMA_DMAP_SEL_C2H_DSC_PIDX
0x301C correspond to QDMA_DMAP_SEL_WRB_CIDX

QDMA_TRQ_MSIX_VF (0x4000)

VF functions can access the MSIX table with offset (0x0000) from that function. The description for this register space is the same as QDMA_TRQ_MSIX (0x30000).

QDMA_VF_MAILBOX (0x5000)

Table 97: **QDMA_VF_MAILBOX (0x05000) Register Space**

Registers (Address)	Address	Description
Function Status Register (0x5000)	0x5000	Status register bits
Function Command Register (0x5004)	0x5004	Command register bits
Function Interrupt Vector Register (0x5008)	0x5008	Interrupt vector register
Target Function Register (0x500C)	0x500C	Target Function register
Function Interrupt Control Register (0x5010)	0x5010	Interrupt Control Register
RTL Version Register (0x5014)	0x5014	RTL Version Register
Incoming Message Memory (0x5800-0x587C)	0x5800-0x587C	Incoming message (128 bytes)
Outgoing Message Memory (0x5C00-0x5C7C)	0x5C00-0x5C7C	Outgoing message (128 bytes)

Function Status Register (0x5000)

Table 98: **Function Status Register (0x5000)**

Bit Index	Default	Access Type	Field	Description
[31:12]	0	NA	Reserved	Reserved
[11:4]	0	RO	cur_src_fn	This field is for PF use only. The source function number of the message on the top of the incoming request queue.
[2]	0	RO	ack_status	This field is for PF use only. The status bit will be set when any bit in the acknowledgement status register is asserted.
[1]	0	RO	o_msg_status	For VF: The status bit will be set when VF driver write msg_send to its command register. When the associated PF driver sends acknowledgement to this VF, the hardware clears this field. The VF driver is not allow to update any content in its outgoing mailbox memory (OMM) while o_msg_status is asserted. Any illegal writes to the OMM are discarded (optionally, case an error in the AXI4-Lite response channel). For PF: The field indicated the message status of the target FN which is specified in the Target FN Register. The status bit is set when PF driver sends the msg_send command. When the corresponding function driver sends acknowledgement through msg_rcv, the hardware clears this field. The PF driver is not allow to update any content in its outgoing mailbox memory (OMM) while o_msg_status(target_fn_id) is asserted. Any illegal writes to the OMM are discarded (optionally, case an error in the AXI4L response channel).

Table 98: **Function Status Register (0x5000)** *(cont'd)*

Bit Index	Default	Access Type	Field	Description
[0]	0	RO	i_msg_status	For VF: When asserted, a message in the VF's incoming Mailbox memory is pending for process. The field is cleared after the VF driver writes msg_rcv to its command register. For PF: When asserted, the messages in the incoming Mailbox memory are pending for process. The field is cleared only when the event queue is empty.

Function Command Register (0x5004)

Table 99: **Function Command Register (0x5004)**

Bit Index	Default	Access Type	Field	Description
[31:3]	0	NA	Reserved	Reserved
[2]	0	RO	Reserved	Reserved
[1]	0	RW	msg_rcv	For VF: VF marks the message in its Incoming Mailbox Memory as received. The hardware asserts the acknowledgement bit of the associated PF. For PF: PF marks the message send by target_fn as received. The hardware refreshes the i_msg_status of the PF, and clears the o_msg_status of the target_fn.
[0]	0	RW	msg_send	For VF: VF marks the current message in its own Outgoing Mailbox as valid. For PF: Current target_fn_id belongs to a VF: PF finished writing a message into the Incoming Mailbox memory of the VF with target_fn_id. The hardware sets the i_msg_status field of the target FN's status register. Current target_fn_id belongs to a PF: PF finished writing a message into its own outgoing Mailbox memory. The hardware pushes the message to the event queue of the PF with target_fn_id.

Function Interrupt Vector Register (0x5008)

Table 100: **Function Interrupt Vector Register (0x5008)**

Bit Index	Default	Access Type	Field	Description
[31:5]	0	NA	Reserved	Reserved
[4:0]	0	RW	int_vect	5-bit interrupt vector assigned by the driver software.

Target Function Register (0x500C)

Table 101: **Target Function Register (0x500C)**

Bit Index	Default	Access Type	Field	Description
[31:8]	0	NA	Reserved	Reserved
[7:0]	0	RW	target_fn_id	This field is for PF use only. The FN number that the current operation is targeting.

Function Interrupt Control Register (0x5010)

Table 102: **Function Interrupt Control Register (0x5010)**

Bit Index	Default	Access Type	Field	Description
[31:1]	0	NA	res	Reserved
[0]	0	RW	int_en	Interrupt enable.

RTL Version Register (0x5014)

Table 103: **RTL Version Register (0x5014)**

Bit	Default	Access Type	Field	Description
[31:16]	0x1fd3	RO	.	QDMA ID
[15:0]	0	RO	.	Vivado versions 0x10: Vivado version 2019.2

Incoming Message Memory (0x5800-0x587C)

Table 104: **Incoming Message Memory (0x5800-0x587C)**

Register	Addr	Default	Access Type	Field	Width	Description
i_msg_i	0x5800 + i*4	0	RW		32	The *i*th word of the incoming message (i < 128).

Outgoing Message Memory (0x5C00-0x5C7C)

Table 105: **Outgoing Message Memory (0x5C00-0x5C7C)**

Register	Addr	Default	Access Type	Field	Width	Description
o_msg_i	0x5C00 + i *4	0	RW		32	The *i*th word of the outgoing message (i < 128).

AXI4-Lite Slave CSR Register Space

The Bridge register space and DMA register space are accessible through the AXI4-Lite Slave CSR interface. This interface is accessible only when `csr_prog_done` port is set to 1. You must wait until `csr_prog_done` port it set.

Table 106: **AXI4-Lite Slave CSR Register Space**

Register Space	AXI4-Lite Slave CSR Interface	Details
Bridge registers	AXI4-Lite Slave CSR Address **bit [15]** is set to 0	Found in `qdma_v4_0_bridge_registers.csv` available in the Register Reference File.
DMA registers	AXI4-Lite Slave CSR Address **bit [15]** is set to 1	Described in QDMA PF Address Register Space and QDMA VF Address Register Space. ***Note***: Through this interface, only the DMA CSR register can be accessed. The DMA Queue space register can only be accessed through AXI4-Lite Slave.

Bridge Register Space

Bridge register addresses start at 0xE00. Addresses from 0x00 to 0xE00 are directed to the PCIe Core configuration register space.

QDMA Bridge register descriptions are found in `qdma_v4_0_bridge_registers.csv` available in the Register Reference File.

DMA Register Space

The DMA register space is described in the following sections:

- QDMA PF Address Register Space
- QDMA VF Address Register Space

AXI4-Lite Slave Register Space

DMA queue space registers can be accessed through the AXI4-Lite Slave interface.

QDMA Queue space PF register addresses and QDMA Queue space VF register addresses are described in QDMA_TRQ_SEL_QUEUE_PF (0x18000) and QDMA_TRQ_SEL_QUEUE_VF (0x3000).

Note: Through this interface, only the DMA Queue space registers can be accessed. DMA CSR register can be accessed only through AXI4-Lite Slave CSR interface.

Chapter 4

Design Flow Steps

This section describes customizing and generating the subsystem, constraining the subsystem, and the simulation, synthesis, and implementation steps that are specific to this IP subsystem. 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 subsystem in the Vivado® Design Suite.

If you are customizing and generating the subsystem 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 subsystem using the following steps:

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

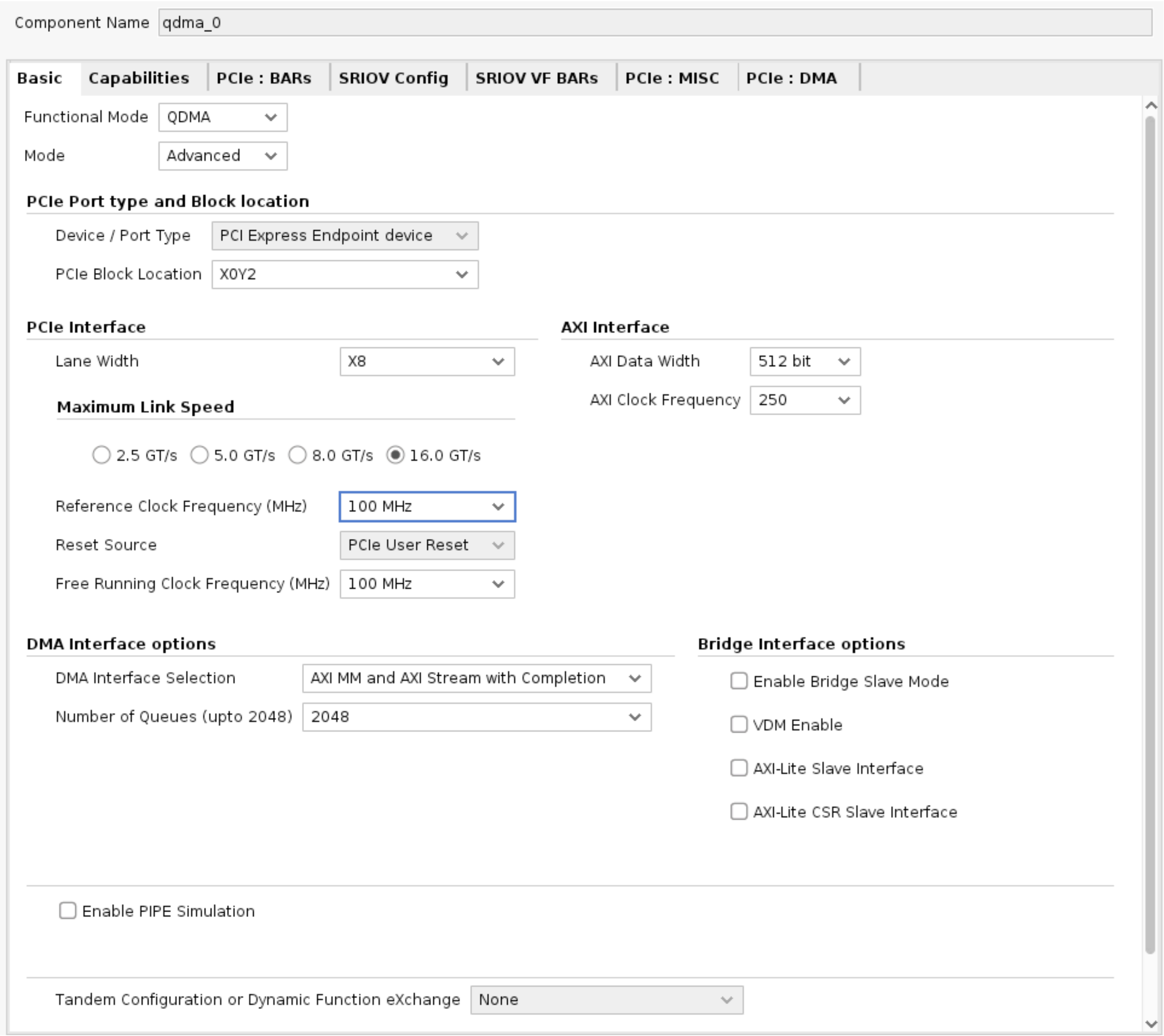
For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) and the *Vivado Design Suite User Guide: Getting Started* (UG910).

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

Basic Tab

The Basic tab is shown in the following figure.

Figure 30: **Basic Tab**



Component Name: qdma_0

Functional Mode: QDMA
Mode: Advanced

PCIe Port type and Block location
Device / Port Type: PCI Express Endpoint device
PCIe Block Location: X0Y2

PCIe Interface
Lane Width: X8

Maximum Link Speed
○ 2.5 GT/s ○ 5.0 GT/s ○ 8.0 GT/s ● 16.0 GT/s

Reference Clock Frequency (MHz): 100 MHz
Reset Source: PCIe User Reset
Free Running Clock Frequency (MHz): 100 MHz

AXI Interface
AXI Data Width: 512 bit
AXI Clock Frequency: 250

DMA Interface options
DMA Interface Selection: AXI MM and AXI Stream with Completion
Number of Queues (upto 2048): 2048

Bridge Interface options
□ Enable Bridge Slave Mode
□ VDM Enable
□ AXI-Lite Slave Interface
□ AXI-Lite CSR Slave Interface

□ Enable PIPE Simulation

Tandem Configuration or Dynamic Function eXchange: None

- **Functional Mode:** Option to select between QDMA and AXI Bridge.
- **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.
- **GT Selection/Enable GT Quad Selection:** Select the Quad in which lane 0 is located.

- **Lane Width:** The core requires the selection of the initial lane width. The *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343) defines the available widths and associated generated core. Wider lane width cores can train down to smaller lane widths if attached to a smaller lane-width device. Options are 4, 8, or 16 lanes.
- **Maximum Link Speed:** The core allows you to select the Maximum Link Speed supported by the device. The *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343) defines the lane widths and link speeds supported by the device. Higher link speed cores are capable of training to a lower link speed if connected to a lower link speed capable device. The default option is Gen3.
- **Reference Clock Frequency:** The default is 100 MHz.
- **Reset Source:** You can choose one of:
  - **PCIe User Reset:** The user reset comes from PCIe core after the link is established. When the PCIe link goes down, the user reset is asserted and the core goes to reset mode. And when the link comes back up, the user reset is deasserted.
  - **Phy Ready:** When selected, the core is not affected by PCIe link status.
- **AXI Data Width:** Select 128, 256 bit, or 512 bit. The core allows you to select the Interface Width, as defined in the *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343). The default interface width set in the Customize IP dialog box is the lowest possible interface width.
- **AXI Clock Frequency:** 250 MHz depending on the lane width/speed.
- **DMA Interface Option:** You can select one of these options:
  - AXI Memory Mapped and AXI Stream with Completion
  - AXI Memory Mapped only
  - AXI Stream with Completion
- **Number of Queues (up to 2048):** Selects maximum number of queues. Options are 512 (default), 1024 and 2048.
- **Enable Bridge Slave Mode:** Select to enable the AXI-MM Slave interface.
- **VDM Enable:** Select to enable Vendor Define Messages.
- **AXI Lite Slave Interface:** Select to enable the AXI4-Lite slave interface, which can access DMA queue space.
- **AXI Lite CSR Slave Interface:** Select to enable the AXI4-Lite CSR slave interface, which can access DMA Configuration Space Register or Bridge registers.

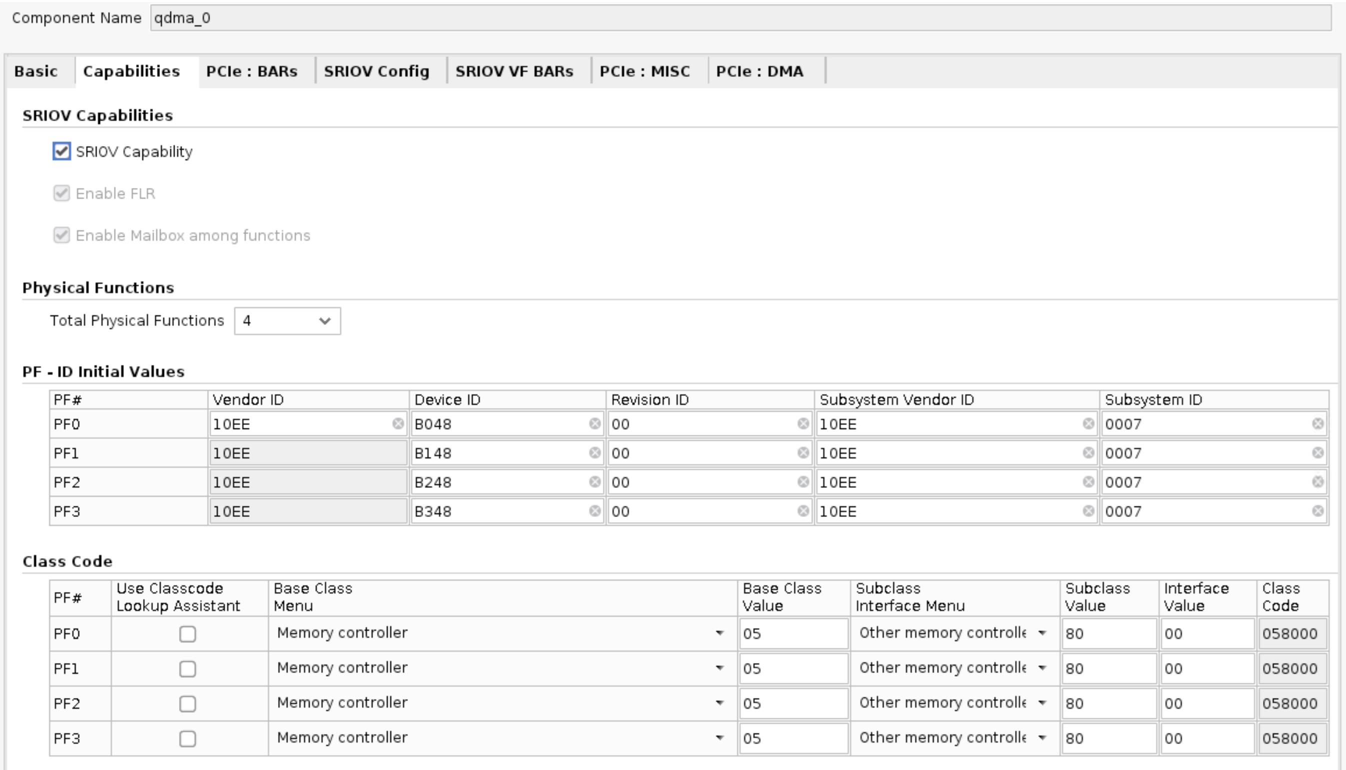
- **Enable PIPE Simulation:** When selected, this option enables an external third-party bus functional model (BFM) to connect to the PIPE interface of integrated block for PCIe. For details, see *PIPE Mode Simulation Using Integrated Endpoint PCI Express Block in Gen3 x8 and Gen2 x8 Configurations* (XAPP1184). Refer to these designs to connect the External PIPE Interface ports of the Versal® ACAP core to third-party BFMs. Enable pipe simulation for faster simulation. This is used only for simulation.

- **Tandem Configuration or Dynamic Function eXchange:** There is no dedicated support for Tandem Configuration or Dynamic Function eXchange using the PL-based PCIe hard block.

Capabilities Tab

The Capabilities Tab is shown in the following figure.

Figure 31: **Capabilities Tab**



Component Name qdma_0

SRIOV Capabilities

☑ SRIOV Capability

☑ Enable FLR

☑ Enable Mailbox among functions

Physical Functions

Total Physical Functions 4

PF - ID Initial Values

PF#	Vendor ID	Device ID	Revision ID	Subsystem Vendor ID	Subsystem ID
PF0	10EE	B048	00	10EE	0007
PF1	10EE	B148	00	10EE	0007
PF2	10EE	B248	00	10EE	0007
PF3	10EE	B348	00	10EE	0007

Class Code

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

- **SRIOV Capability:** Enables Single Root Port I/O Virtualization (SR-IOV) capabilities. The integrated block implements extended SR-IOV PCIe. When this is enabled, SR-IOV is implemented on all selected physical functions. When SR-IOV capabilities are enabled only MSI-X interrupt is supported.

  - **Enable Mailbox among functions:** This is a Mailbox system to communicate between different functions. When **SR-IOV Capability** (above) is enabled, this option is enabled by default. Mailbox can be selected independently of the SR-IOV Capability selection.

  - **Enable FLR:** Enables the functionl level reset port. When SR-IOV capability (above) is enabled, this option is enabled by default.

- **Physical Functions:** A maximum of four Physical Functions can be enabled.

- **PF - ID Initial Values:**

  - **Vendor ID:** Identifies the manufacturer of the device or application. Valid identifiers are assigned by the PCI Special Interest Group to guarantee that each identifier is unique. The default value, `10EEh`, is the Vendor ID for Xilinx. Enter a vendor identification number here. `FFFFh` is reserved.

  - **Device ID:** A unique identifier for the application; the default value, which depends on the configuration selected, is 70h. This field can be any value; change this value for the application.

    The Device ID parameter is evaluated based on:

    - The device family: B for Versal.
    - EP or RP mode
    - Link width
    - Link speed

    For example, Device ID `B03F` represents Device ID for Versal B, `3` for Gen3, and `F` for X16 (width).

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

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

  - **Revision ID:** Indicates the revision of the device or application; an extension of the Device ID. The default value is `00h`; enter values appropriate for the application.

  - **Subsystem Vendor ID:** Further qualifies the manufacturer of the device or application. Enter a Subsystem Vendor ID here; the default value is 10EEh. Typically, this value is the same as Vendor ID. Setting the value to 0000h can cause compliance testing issues.

  - **Subsystem ID:** Further qualifies the manufacturer of the device or application. This value is typically the same as the Device ID; the default value depends on the lane width and link speed selected. Setting the value to 0000h can cause compliance testing issues.

- **Class Code:** The Class Code identifies the general function of a device.

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

- **Base Class:** Broadly identifies the type of function performed by the device.

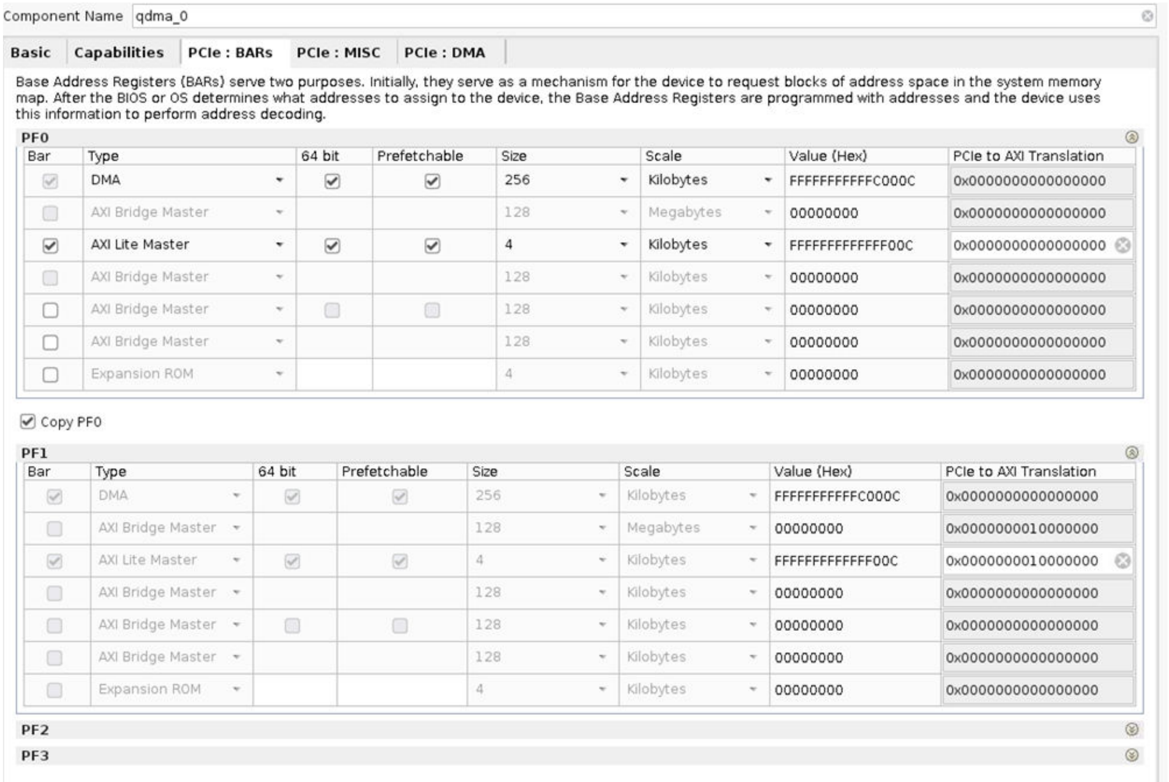
- **Subclass:** More specifically identifies the device function.

- **Interface:** Defines a specific register-level programming interface, if any, allowing device-independent software to interface with the device.

PCIe BARs Tab

The PCIe BARs tab is shown in the following figure.

Figure 32: **PCIe BARs Tab**



Component Name qdma_0

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

PF0

Bar	Type	64 bit	Prefetchable	Size	Scale	Value (Hex)	PCIe to AXI Translation
☑	DMA	☑	☑	256	Kilobytes	FFFFFFFFFFFC000C	0x0000000000000000
□	AXI Bridge Master			128	Megabytes	00000000	0x0000000000000000
☑	AXI Lite Master	☑	☑	4	Kilobytes	FFFFFFFFFFFFF00C	0x0000000000000000
□	AXI Bridge Master			128	Kilobytes	00000000	0x0000000000000000
□	AXI Bridge Master	□	□	128	Kilobytes	00000000	0x0000000000000000
□	AXI Bridge Master			128	Kilobytes	00000000	0x0000000000000000
□	Expansion ROM			4	Kilobytes	00000000	0x0000000000000000

☑ Copy PF0

PF1

Bar	Type	64 bit	Prefetchable	Size	Scale	Value (Hex)	PCIe to AXI Translation
☑	DMA	☑	☑	256	Kilobytes	FFFFFFFFFFFC000C	0x0000000000000000
□	AXI Bridge Master			128	Megabytes	00000000	0x0000000010000000
☑	AXI Lite Master	☑	☑	4	Kilobytes	FFFFFFFFFFFFF00C	0x0000000010000000
□	AXI Bridge Master			128	Kilobytes	00000000	0x0000000000000000
□	AXI Bridge Master	□	□	128	Kilobytes	00000000	0x0000000000000000
□	AXI Bridge Master			128	Kilobytes	00000000	0x0000000000000000
□	Expansion ROM			4	Kilobytes	00000000	0x0000000000000000

PF2

PF3

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

  - **32-bit BARs:** The address space can be as small as 128 bytes or as large as 2 gigabytes. Used for DMA, AXI Lite Master or AXI Bridge Master.

- **64-bit BARs:** The address space can be as small as 128 bytes or as large as 8 Exabytes. Used for DMA, AXI Lite Master or AXI Bridge Master.

All BAR register share these options.

> **IMPORTANT!** *The DMA requires a large amount of space to support functions and queues. By default, 64-bit BAR space is selected for the DMA BAR. This applies for PF and VF bars. You must calculate your design needs first before selecting between 64-bit and 32-bit BAR space.*

BAR selections are configurable. By default DMA is at BAR 0 (64 bit), AXI-Lite Master is at BAR 2 (64 bit). These selections can be changed according to user needs.

- **BAR:** Click the checkbox to enable the BAR. Deselect the checkbox to disable the BAR.
- **Type:** Select from **DMA** (by default in BAR0), **AXI Lite Master** (by default in BAR1, if enabled), or **AXI Bridge Master** (by default in BAR2, if enabled). All other BARs, you can select between AXI List Master and AXI Bridge Master. Expansion ROM can be enabled by selecting BAR6

  For 64-bit BAR (default selection), **DMA** (by default in BAR0), **AXI Lite Master** (by default in BAR2, if enabled), and **AXI Bridge Master** (by default in BAR4, if enabled). Expansion ROM can be enabled by selection BAR6.

  - **DMA:** DMA by default is assigned to BAR0 space and for all PFs. DMA option can be selected in any available BAR (only one BAR can have DMA option). If you select **DMA Mailbox Management** rather than DMA; however, DMA Mailbox Management will not allow you to perform any DMA operations. After selecting the DMA Mailbox Management option, the host has access to the extended Mailbox space. For details about this space, see the QDMA_PF_MAILBOX (0x22400) register space.
  - **AXI Lite Master:** Select the AXI Lite Master interface option for any BAR space. The Size, scale and address translation are configurable.
  - **AXI Bridge Master:** Select the AXI Bridge Master interface option for any BAR space. The Size, scale and address translation are configurable.
  - **Expansion ROM:** When enabled, this space is accessible on the AXI4-Lite Master. This is a read-only space. The size, scale, and address translation are configurable.
- **Size:** The available Size range depends on the 32-bit or 64-bit bar selected. The DMA requires 256 Kbytes of space, which is the fixed default selection. Other BAR size selections are available, but must be specified.
- **Scale:** Select between Byte, Kilobytes and Megabytes.
- **Value:** The value assigned to the BAR based on the current selections.

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

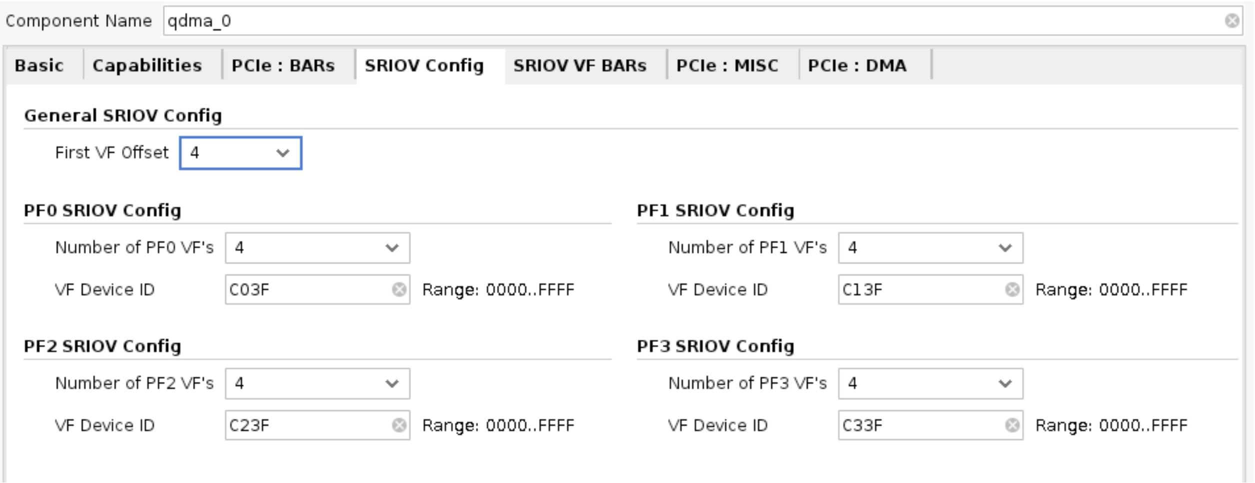
SRIOV Config Tab

The SRIOV Config tab allows you to specify the SR-IOV capability for a physical function (PF). The information is used to construct the SR-IOV capability structure. Virtual functions do not exist on power-on. It is the function of the system software to discover and enable VFs based on system capability. The VF support is discovered by scanning the SR-IOV capability structure for each PF.

***Note*:** When **SRIOV Capability** is selected in Capabilities Tab, the SRIOV Config tab appears.

The SRIOV Config Tab is shown in the following figure.

Figure 33: **SRIOV Config Tab**



Component Name: qdma_0

General SRIOV Config

First VF Offset: 4

PF0 SRIOV Config

Number of PF0 VF's: 4

VF Device ID: C03F Range: 0000..FFFF

PF1 SRIOV Config

Number of PF1 VF's: 4

VF Device ID: C13F Range: 0000..FFFF

PF2 SRIOV Config

Number of PF2 VF's: 4

VF Device ID: C23F Range: 0000..FFFF

PF3 SRIOV Config

Number of PF3 VF's: 4

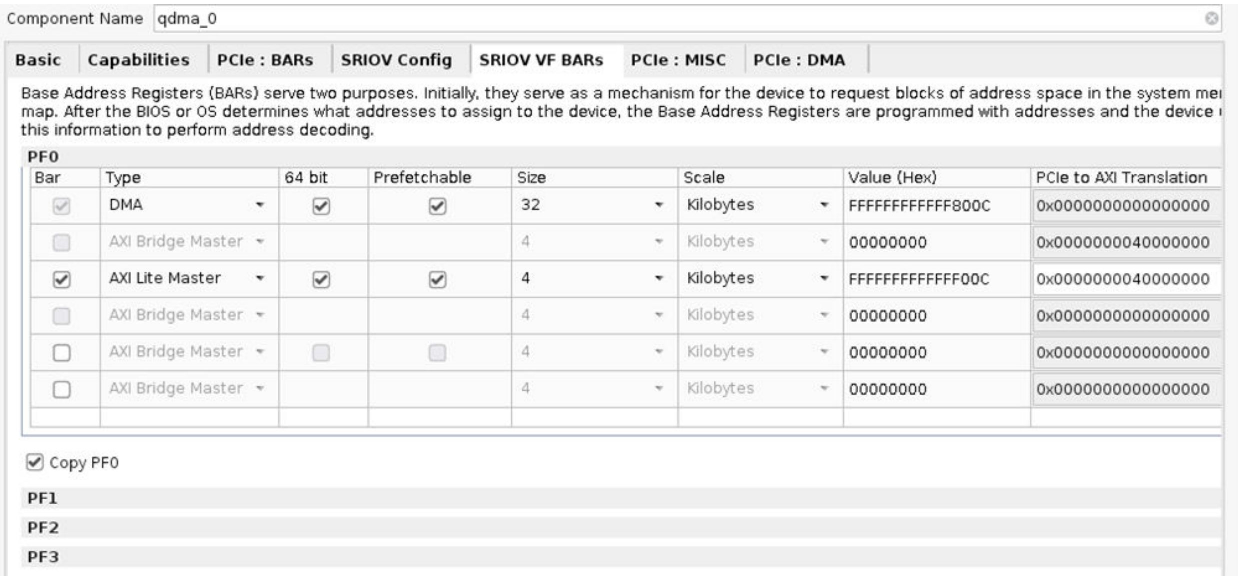
VF Device ID: C33F Range: 0000..FFFF

- **General SRIOV Config:** This value specifies the offset of the first PF with at least one enabled VF. When ARI is enabled, allowed value is 'd4 or 'd64, and the total number of VF in all PFs plus this field must not be greater than 256. When ARI is disabled, this field will be set to 1 to support 1PFplus 7VF non-ARI SRIOV configurations only.
- **Number of PFx VFs:** Indicates the number of virtual functions associated to the physical function. A total of 252 virtual functions are available that can be flexibly used across the four physical functions.
- **VF Device ID:** Indicates the 16-bit Device ID for all virtual functions associated with the physical function.

SRIOV VF BARs Tab

The SRIOV VF BARs tab is shown in the following figure.

Figure 34: **SRIOV VF BARs Tab**



Component Name qdma_0

Basic | Capabilities | PCIe : BARs | SRIOV Config | SRIOV VF BARs | PCIe : MISC | PCIe : DMA

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

PF0

Bar	Type	64 bit	Prefetchable	Size	Scale	Value (Hex)	PCIe to AXI Translation
☑	DMA	☑	☑	32	Kilobytes	FFFFFFFFFFFF800C	0x0000000000000000
□	AXI Bridge Master			4	Kilobytes	00000000	0x0000000040000000
☑	AXI Lite Master	☑	☑	4	Kilobytes	FFFFFFFFFFFFF00C	0x0000000040000000
□	AXI Bridge Master			4	Kilobytes	00000000	0x0000000000000000
□	AXI Bridge Master	□	□	4	Kilobytes	00000000	0x0000000000000000
□	AXI Bridge Master			4	Kilobytes	00000000	0x0000000000000000

☑ Copy PF0

PF1

PF2

PF3

The SRIOV VF BARs tab enables you to configure the base address registers (BARs) for all virtual function (VFs) within a virtual function group (VFG). All the VFs within the same VFG share the same BASE ADDRESS Registers (BARS) configurations. Each Virtual Function supports up to six 32-bit BARs or three 64-bit BARs. Virtual Function BARs can be configured without any dependency on the settings of the associated Physical Functions BARs.

IMPORTANT! *The DMA requires a large amount of space to support functions and queues. By default, 64-bit BAR space is selected for the DMA BAR. This applies for PF and VF bars. You must calculate your design needs first before selecting between 64-bit and 32-bit BAR space.*

BAR selections are configurable. By default DMA is at BAR 0 (64 bit), AXI-Lite Master is at BAR 2 (64 bit). These selections can be changed according to user needs.

- **BAR:** Select applicable BARs using the checkboxes.
- **Type:** Select the relevant option:
  - **DMA:** Is fixed to BAR0 space.
  - **AXI Lite Master:** Is fixed to BAR1 space.
  - **AXI Bridge Master:** Is fixed to BAR2 space.For all other bars, select either AXI Lite Master or AXI Bridge Master.

    Note: The current IP supports a maximum of one DMA BAR (or a management BAR given only mailbox is required) for one VF. The other BARs can be configured as AXI Lite Master to access the assigned memory space through the AXI4-Lite bus. Virtual Function BARs do not support I/O space and must be configured to map to the appropriate memory space.
- **64-bit:** VF BARs can be either 64-bit or 32-bit. The default is 64-bit BAR.

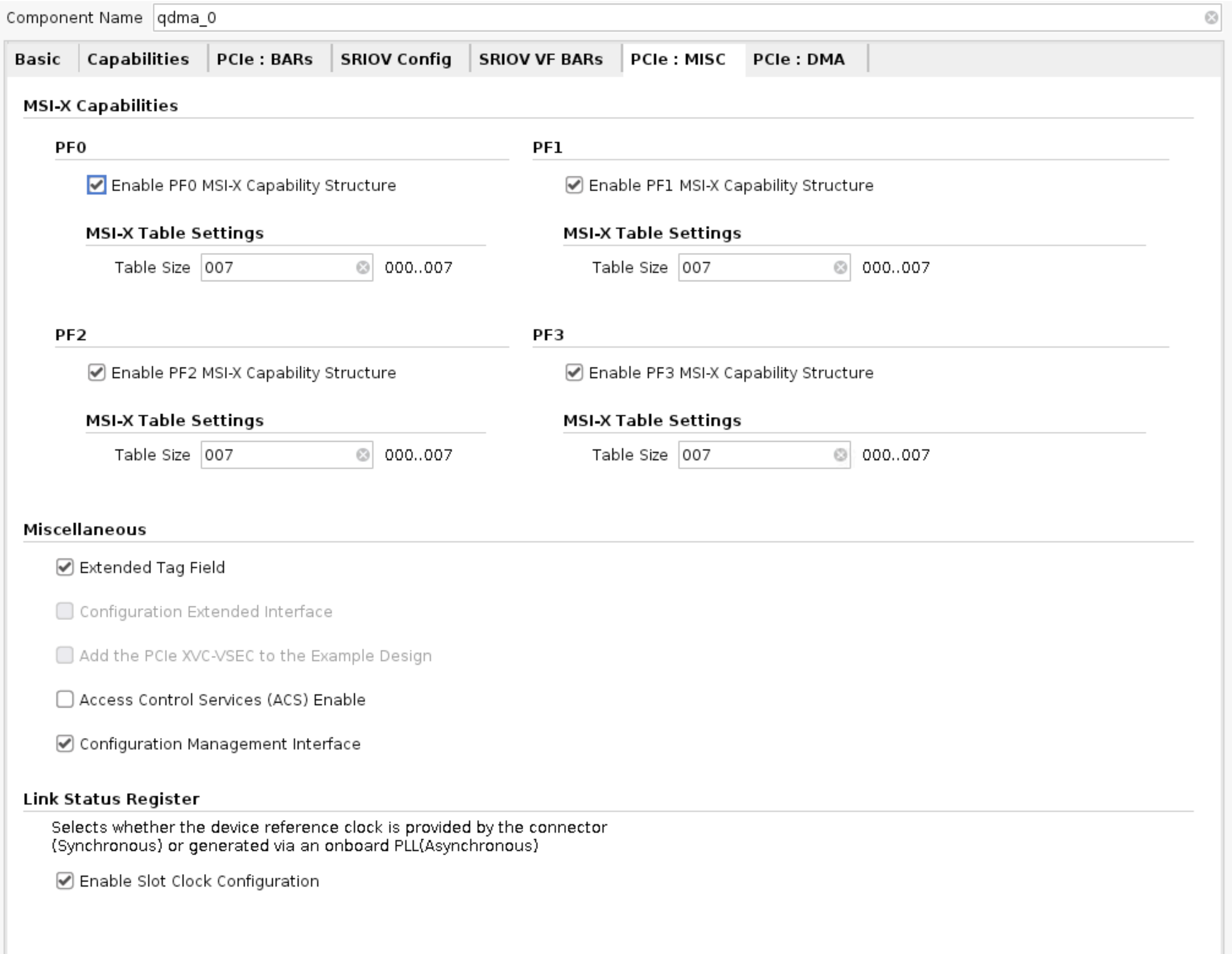
- 64-bit addressing is supported for the DMA BAR.
- When a BAR is set as 64 bits, it uses the next BAR for the extended address space and makes the next BAR inaccessible.

- **Size:** The available Size range depends on the 32-bit or 64-bit BAR selected. The Supported Page Sizes field indicates all the page sizes supported by the PF and, as required by the SR-IOV specification. Based on the Supported Page Size field, the system software sets the System Page Size field which is used to map the VF BAR memory addresses. Each VF BAR address is aligned to the system page boundary.By default, DMA space is 32 Kbytes. With this much space allocated, the user logic can access 256 queues for a VF function.
- **Value:** The value assigned to the BAR based on the current selections.

PCIe MISC Tab

The PCIe Miscellaneous Tab is shown in the following figure.

Figure 35: **PCIe MISC Tab**



Component Name qdma_0

MSI-X Capabilities

PF0

☑ Enable PF0 MSI-X Capability Structure

MSI-X Table Settings

Table Size 007 000..007

PF1

☑ Enable PF1 MSI-X Capability Structure

MSI-X Table Settings

Table Size 007 000..007

PF2

☑ Enable PF2 MSI-X Capability Structure

MSI-X Table Settings

Table Size 007 000..007

PF3

☑ Enable PF3 MSI-X Capability Structure

MSI-X Table Settings

Table Size 007 000..007

Miscellaneous

☑ Extended Tag Field

☐ Configuration Extended Interface

☐ Add the PCIe XVC-VSEC to the Example Design

☐ Access Control Services (ACS) Enable

☑ Configuration Management Interface

Link Status Register

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

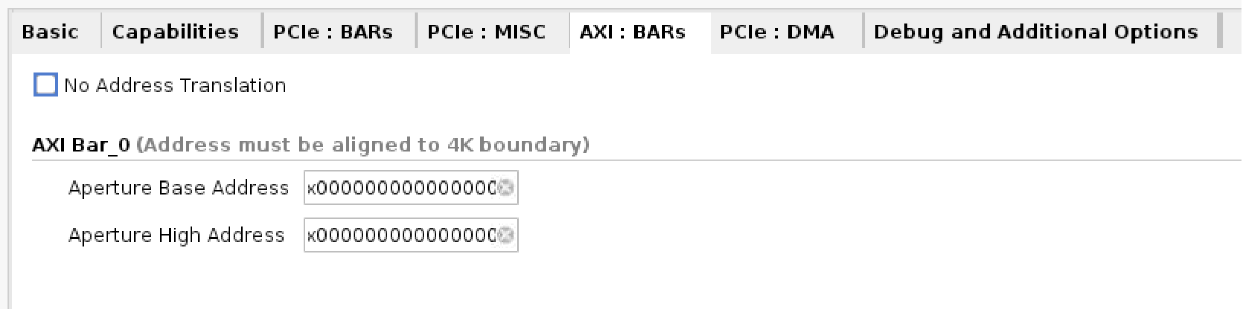
☑ Enable Slot Clock Configuration

- **MSI-X Capabilities:** MSI-X is enabled by default.The MSI-X settings for different physical functions can be set as required.
- **MSI-X Table Settings:** Defines the MSI-X Table Structure.
  - **Table Size:** Specifies the MSI-X Table size. The default is 8 (8 interrupt vectors per function).
- **Extended Tag Field:** By default for Versal® devices the Extended Tab option gives 256 tags. If Extended Tag option is not selected, the DMA uses 32 tags.
- **Configuration Extended Interface:** The PCIe extended interface can be selected for more configuration space. When Configuration Extended Interface is selected user is responsible for adding logic to extend the interface to make it work properly.
- **Access Control Server (ACS) Enable:** ACS is selected by default.
- **Configuration Management Interface:** This interface is used to read and write to the configuration space registers.
- **Link Status Register:** By default, Enable Slot Clock Configuration is selected. This means that the slot configuration bit is enabled in the link status register.

AXI BARs Tab

The SRIOV VF BARs tab is shown in the following figure.

Figure 36: **AXI BARs Tab**



☐ No Address Translation

AXI Bar_0 (Address must be aligned to 4K boundary)

Aperture Base Address x0000000000000000

Aperture High Address x0000000000000000

Slave Bridge Address Translation

- **No Address Translation:** When this option is selected, the DMA will not do any address translation. One full 64-bit BAR space is provided, and you are responsible for any address translation, if required.When address translation is required by DMA, do not select this option.
- **AXI Bar_0 Address Translation: Aperture Base Address** and **Aperture High Address** can be programmed with desired values. This provides the AXI bar size. Address translation for higher bits (above BAR size) can be programmed as desired.

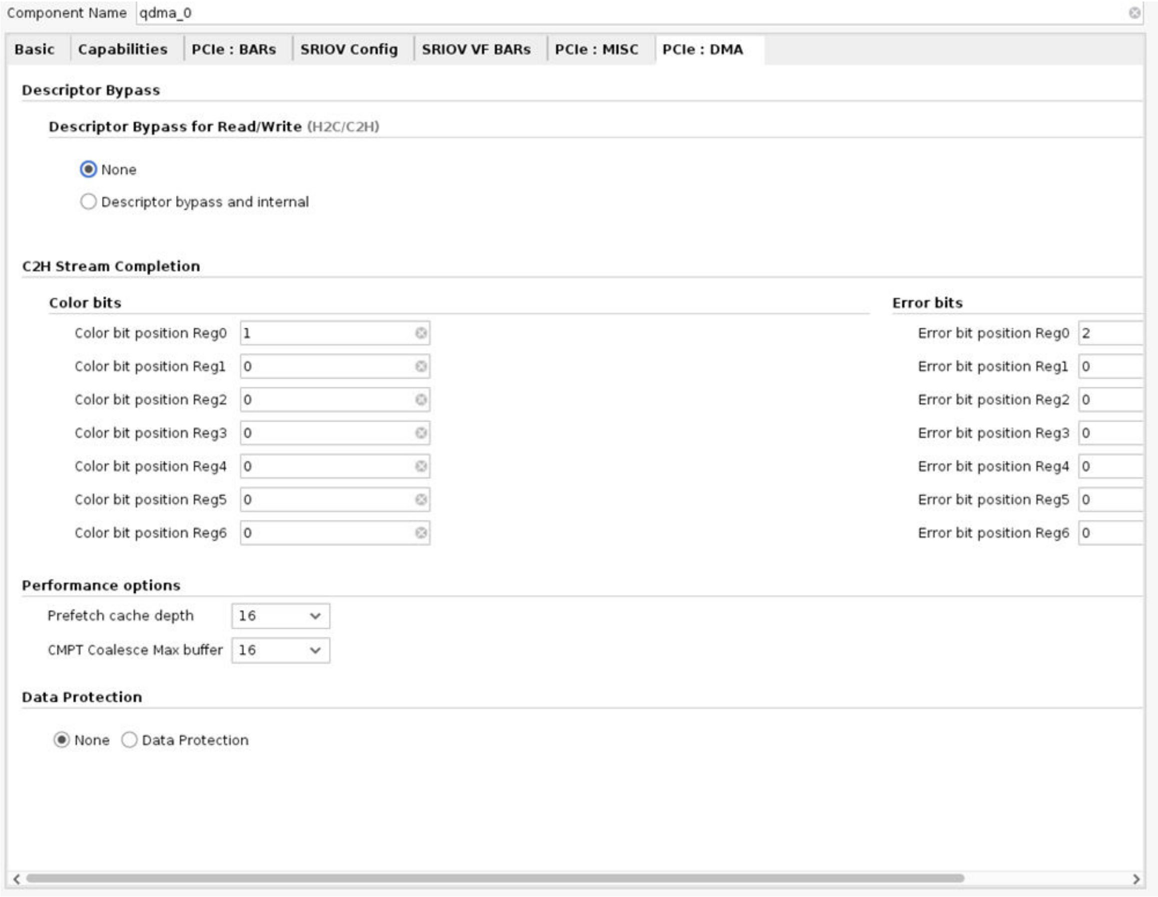
Related Information

Slave Bridge

PCIe DMA Tab

The PCIe DMA Tab is shown in the following figure.

Figure 37: **PCIe DMA Tab**



Component Name qdma_0

Descriptor Bypass

Descriptor Bypass for Read/Write (H2C/C2H)

● None
○ Descriptor bypass and internal

C2H Stream Completion

Color bits		Error bits	
Color bit position Reg0	1	Error bit position Reg0	2
Color bit position Reg1	0	Error bit position Reg1	0
Color bit position Reg2	0	Error bit position Reg2	0
Color bit position Reg3	0	Error bit position Reg3	0
Color bit position Reg4	0	Error bit position Reg4	0
Color bit position Reg5	0	Error bit position Reg5	0
Color bit position Reg6	0	Error bit position Reg6	0

Performance options

Prefetch cache depth 16
CMPT Coalesce Max buffer 16

Data Protection

● None ○ Data Protection

- **Descriptor Bypass for Read/Write (H2C/C2H):** Two options to select from.

  ***Note*:** In this mode (Internal mode) DMA will not bypass any H2C or C2H descriptors.

  - **Descriptor bypass and Internal:** In this mode descriptor ports for bypass out and bypass in are both enabled. Based on the context settings H2C or C2H descriptors can be sent out on descriptor bypass out. User can send in descriptors on Descriptor bypass in ports.

- **C2H Stream Completion:**

- **C2H Stream Completion Color bits:** Completion Color bit position in completion entry. There are seven registers available to program, from bit 0 to 511 (for 64 bytes completion). You can program the bits, and generate a BIT file. During the DMA transfer, the input pins `s_axis_c2h_cmpt_ctrl_color_idx[2:0]` determine which Color bit position to use. Default bit position 1 is selected in register 0.
- **C2H Stream Completion Error bits:** Completion Error bit position in completion entry. There are seven registers available to program, from bit 0 to 511 (for 64 bytes completion). You can program the bits, and generate a BIT file. During a DMA transfer, the input pins `s_axis_c2h_cmpt_ctrl_err_idx[2:0]` determine which Error bit position to use. Default bit position 2 is selected in register 0.

- **Performance options:**
  - **Pre-fetch cache depth:** The Prefetch cache supports up to 64 Queues. Select one of 16 or 64 (default 16). The Prefetch cache can support that many active queues at any given time. When one active queue finishes fetch and delivers all the descriptors for the packets of that queue, it then releases cache entry for other active queues. A larger cache size supports more active queues, but the area will also increase.
  - **CMPT Coalesce Max buffer:** Completion (CMPT) Coalesce Max buffer supports up to 64 buffers. Select one of 16 or 32 (default 16). Each entry of the CMPT Coalesce Buffer coalesces multiple Completions (up to 64B) to form a single queue before writing to the host to improve bandwidth utilization. A deeper CMPT Coalesce Buffer allows coalescing within more queues, but will increase the area as a downside.
- **Data Protection:** Parity Checking and end to end data protection. By default, data protection is not enabled.

  When **Data Protection** is not enabled:

  - You do not need to give any CRC/ECC values on C2H data and the control interface.
  - This will not log any Error and will not drop any packet.
  - User should ground the ECC and CRC ports.
  - CMPT parity check is not affected by this parameter.

    ***Note*:** You must always give the parity on CMPT.

  When **Data Protection** is enabled:

  - You must send CRC/ECC values on C2H data and the control interface.
  - If there is any ECC or CRC error, error bits will be logged and data packet will be sent to host.
  - If error interrupt is enabled, an interrupt will be sent to host.
  - FATAL error can be enabled in the QDMA_C2H_FATAL_ERR_ENABLE register.
    - QDMA_C2H_FATAL_ERR_ENABLE[0]: If this bit is set, all packets are dropped after an error occurs.

- QDMA_C2H_FATAL_ERR_ENABLE[1]: If this bit is set, parity is inverted and an error packet is sent to PCIe.

User Parameters

Additional core customizing options are available. For details, see AR 72352.

Output Generation

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

Constraining the Subsystem

Required Constraints

The QDMA subsystem 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_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 for PCIe 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.

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

Clock Frequencies

For detailed information about clock requirements, see the *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343).

Clock Management

For detailed information about clock requirements, see the *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343).

Clock Placement

For detailed information about clock requirements, see the *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343).

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 the XDC constraint file and must be called from the Tcl command file or typed directly into the Tcl Console.

Simulation

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

Basic Simulation

Simulation models for the AXI-MM and AXI-ST options can be generated and simulated. The simple simulation model options enable you to develop complex designs.

AXI-MM Mode

The example design for the AXI4 Memory Mapped (AXI-MM) mode has 512 KB block RAM on the user side, where data can be written to the block RAM, and read from block RAM to the Host.

After the Host to Card (H2C) transfer is started, the DMA reads data from the Host memory, and writes to the block RAM. After the transfer is completed, the DMA updates the write back status and generates an interrupt (if enabled). Then, the Card to Host (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 set up with one descriptor each, and the total transfer size is 128 bytes.

AXI-ST Mode

The example design for the AXI4-Stream (AXI-ST) mode has a data check that checks the data from the H2C transfer, and has a data generator that generates the data for C2H transfer.

After the H2C transfer is started, the DMA engine reads data from the Host memory, and writes to the user side. After the transfer is completed, the DMA updates write back status and generates an interrupt (if enabled). The data checker on the user side checks for a predefined data to be present, and the result is posted in a predefined address for the user application to read.

After the C2H transfer is started, the data generator generates predefined data and associated control signals, and sends them to the DMA. The DMA transfers data to the Host, updates the completion (CMPT) ring entry/status, and generates an interrupt (if enabled).

H2C and C2H are set up with one descriptor each, and the total transfer size is 128 bytes.

Related Information

Reference Software Driver Flow

PIPE Mode Simulation

The QDMA 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 tab 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.

Synthesis and Implementation

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

Example Design

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

💡 **TIP:** *The PCIe reset pin for PL PCIE designs can be connected to any compatible single ended PL I/O pin location. If your board is compatible for either CPM4 or PL PCIE usage, you can use the CPM4 pin MIO38 to route the `sys_rst_n`. When this is done, the PL PCIE can use the reset as routed to the PL.*

Before opening the example design, set the following Tcl property to use the reset on the MIO38 pin:

```
set_property CONFIG.insert_cips {true} [get_ips pcie_versal_0]
```

Available Example Designs

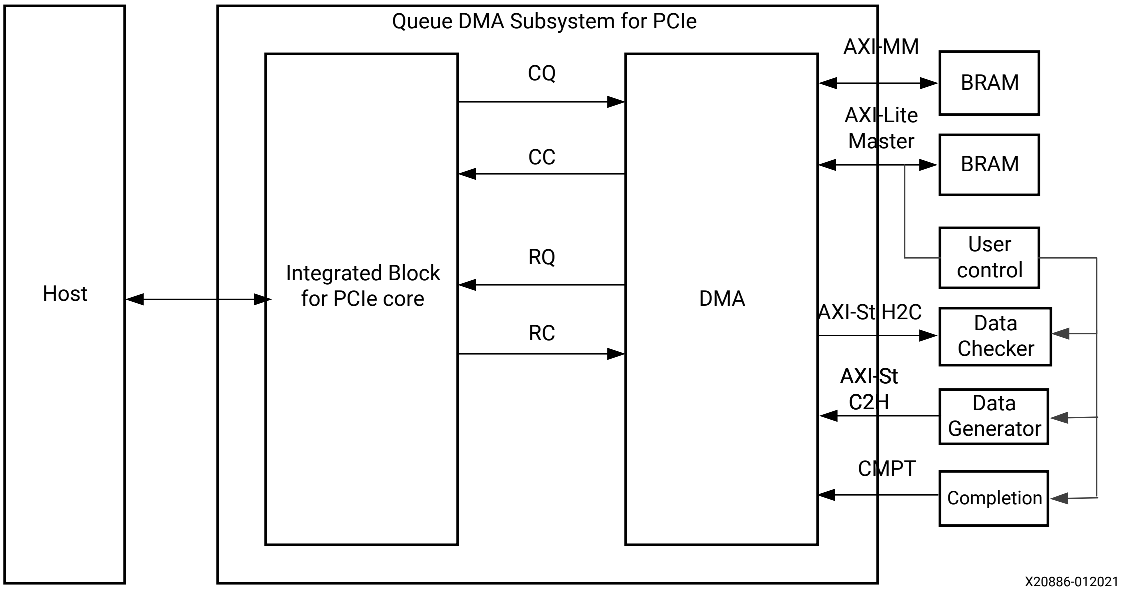
The example designs are as follows:

- AXI Memory Mapped and AXI4-Stream With Completion Default Example Design
- AXI Memory Mapped Example Design
- AXI Stream with Completion Example Design
- Example Design with Descriptor Bypass In/Out Loopback

AXI Memory Mapped and AXI4-Stream With Completion Default Example Design

The following is an example design generated when the DMA Interface Selection option is set to **AXI Memory Mapped and AXI4-Stream with Completion** option in the Basic tab.

Figure 38: **Default Example Design**



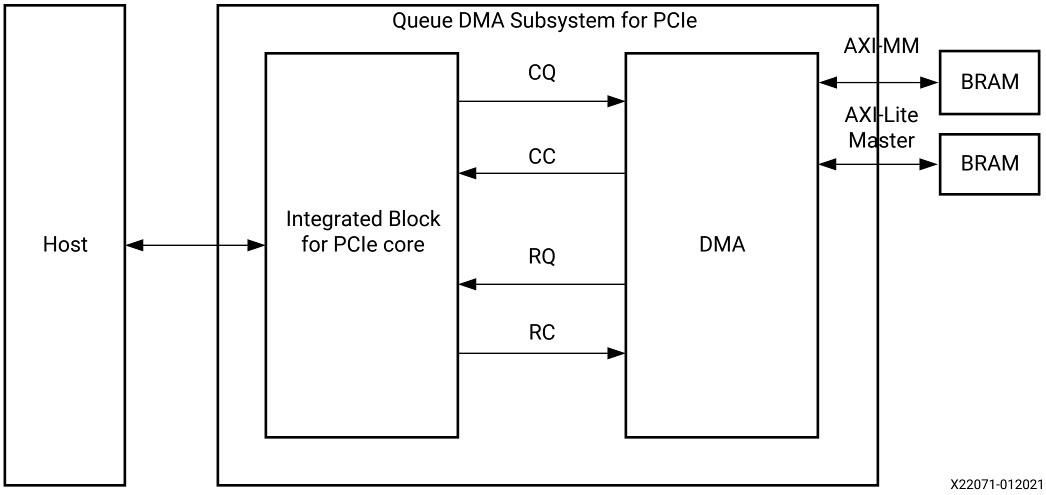
The generated example design provides blocks to interface with the AXI Memory Mapped and AXI4-Stream interfaces.

- The AXI MM interface is connected to 512 KB of block RAM.
- The AXI4-Stream interface is connected to custom data generator and data checker module.
- The CMPT interface is connected to the Completion block generator.
- The data generator and checker works only with predefined pattern, which is a 16-bit incremental pattern starting with 0. This data file is included in driver package.

The pattern generator and checker can be controlled using the registers found in the Example Design Registers. These registers can only be controlled through the AXI4-Lite Master interface. To test the QDMA's AXI4-Stream interface, ensure that the AXI4-Lite Master interface is present.

AXI Memory Mapped Example Design

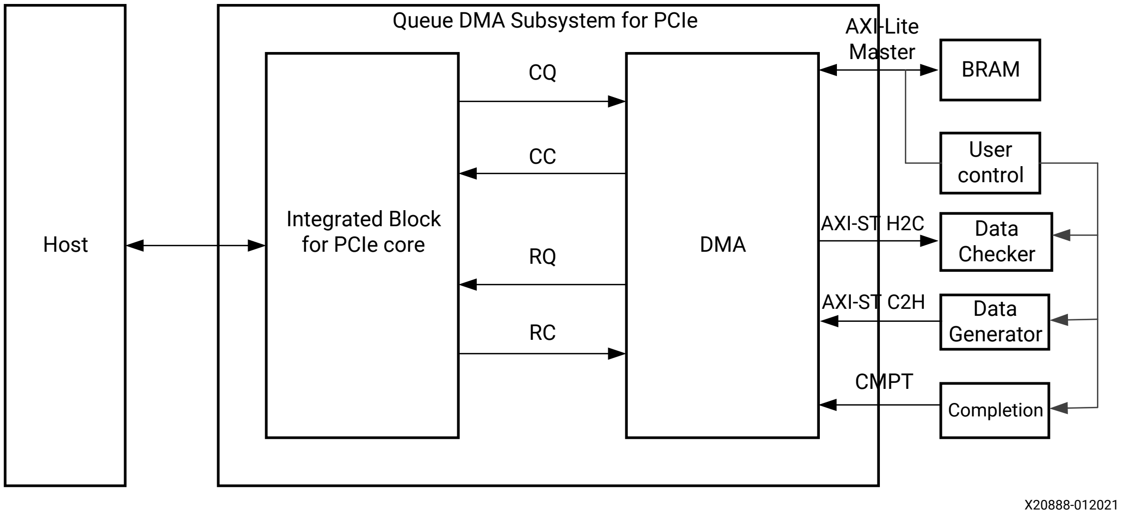
Figure 39: **AXI Memory Map Example Design**



The example design above is generated when the DMA Interface Selection option is set to **AXI-MM only** in the Basic tab. In this mode, the AXI MM interface is connected to a 512 KB block RAM. The diagram above shows that AXI4-Lite Master is connected to a 4 KB block RAM. For Host to Card (H2C) transfers, the DMA reads data from the Host and writes to the block RAM. For Card to Host (C2H) transfers, the DMA reads data from the block RAM and writes to the Host memory.

AXI Stream with Completion Example Design

Figure 40: **AXI4-Stream Example Design**

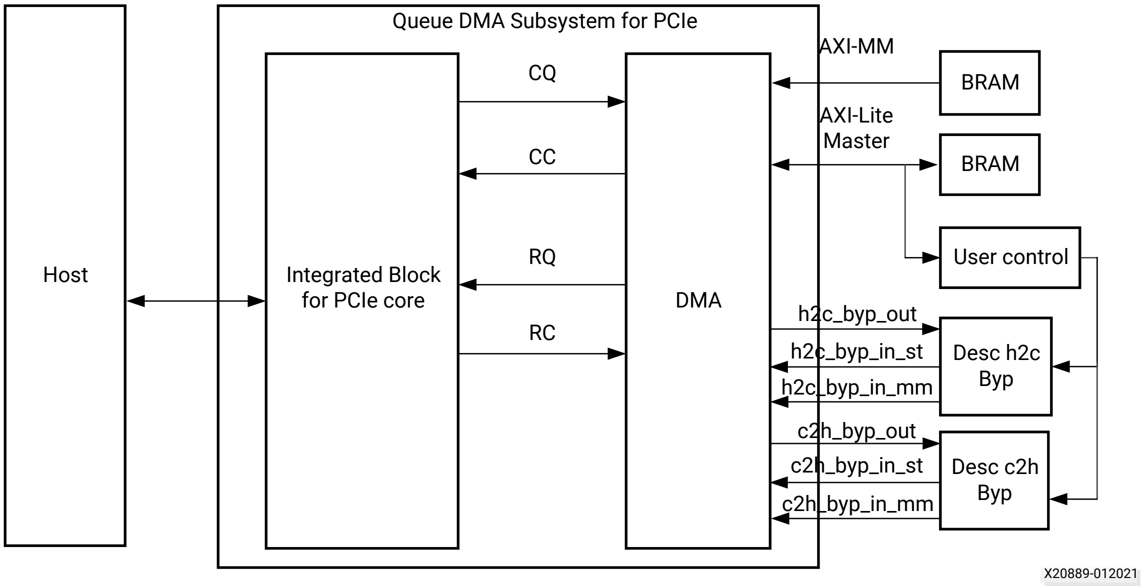


The example design above is generated when the DMA Interface Selection option is set to **AXI Stream with Completion** in the Basic tab. In this mode, the AXI-ST H2C interface is connected to a data checker, and the AXI-ST C2H interface is connected to data generator and CMPT interface is connected to Completion generator module. The diagram shows AXI4-Lite Master is connected to the 4 KB block RAM and the User Control logic. The software can control data checker and data generator though the AXI4-Lite Master interface. The data generator and checker work only with a predefined pattern, which is a 16-bit incremental pattern starting with 0. This data file is included in the driver package.

The pattern generator and checker can be controlled using the registers found in the Example Design Registers.

Example Design with Descriptor Bypass In/Out Loopback

Figure 41: **AXI Memory Map and Descriptor Bypass Example Design**



The example design above is generated when **Descriptor Bypass for Read (H2C)** and **Descriptor Bypass for Write (C2H)** options are selected in the PCIe DMA tab. These options can be selected with any of the DMA Interface Options in the Basic tab:

- AXI Memory Mapped and AXI4-Stream with Completion
- AXI Memory Mapped only
- AXI Stream with Completion
- AXI Memory Mapped with Completion

The Descriptor Bypass in/out loopback is controlled by the AXI4-Lite Master by writing to the Example Design Register DESCRIPTOR_BYPASS (0x090) bit[0] and bit[1].

C2H Stream Simple Bypass Mode Transfer

To set up a QDMA to data transfer in simple bypass mode

1. Write the active qid to register 0x1408 (MDMA_C2H_PFCH_BYP_QID).
2. Read the tag value from 0x140C (MDMA_c2H_PFCH_BYP_TAG).
3. Write the tag value and qid that was used to fetch the tag in the example design register C3H_PREFETCH_TAG 0x24. The qid bits are [26:16] and tag bits are [6:0].
4. Set up the simple bypass descriptor loopback by writing register DESCRIPTOR_BYPASS 0x90 bits [2:0] = `3'b100`.

After the setup initial C2H stream data transfer, the prefetch tag is valid until the qid is valid. When the current qid becomes invalid, you must generate a new tag.

Example Design Registers

Table 107: **Example Design Registers**

Registers	Address	Description
C2H_ST_QID (0x000)	0x000	AXI-ST C2H Queue id
C2H_ST_LEN (0x004)	0x004	AXI-ST C2H transfer length
C2H_CONTROL_REG (0x008)	0x008	AXI-ST C2H pattern generator control
H2C_CONTROL_REG (0x00C)	0x00C	AXI-ST H2C Control
H2C_STATUS (0x010)	0x010	AXI-ST H2C Status
C2H_STATUS (0x018)	0x018	AXI-ST C2H Status
C2H_PACKET_COUNT (0x020)	0x020	AXI-ST C2H number of packets to transfer
C2H_COMPLETION_DATA_0 (0x030) to C2H_COMPLETION_DATA_7 (0x04C)	0x4C-0x030	AXI-ST C2H completion data
C2H_COMPLETION_SIZE (0x050)	0x050	AXI-ST completion data type
SCRATCH_REG0 (0x060)	0x060	Scratch register 0
SCRATCH_REG1 (0x064)	0x064	Scratch register 1
C2H_PACKETS_DROP (0x088)	0x088	AXI-ST C2H Packets drop count
C2H_PACKETS_ACCEPTED (0x08C)	0x08C	AXI-ST C2H Packets accepted count
DESCRIPTOR_BYPASS (0x090)	0x090	C2H and H2C descriptor bypass loopback
USER_INTERRUPT (0x094)	0x094	User interrupt, vector number, function number
USER_INTERRUPT_MASK (0x098)	0x098	User interrupt mask
USER_INTERRUPT_VECTOR (0x09C)	0x09C	User interrupt vector
DMA_CONTROL (0x0A0)	0x0A0	DMA control
VDM_MESSAGE_READ (0x0A4)	0x0A4	VDM message read

C2H_ST_QID (0x000)

Table 108: **C2H_ST_QID (0x000)**

Bit	Default	Access Type	Field	Description
[31:11]	0	NA		Reserved
[10:0]	0	RW	c2h_st_qid	AXI4-Stream C2H Queue ID

C2H_ST_LEN (0x004)

Table 109: **C2H_ST_LEN (0x004)**

Bit	Default	Access Type	Field	Description
[31:16]	0	NA		Reserved
[15:0]	0	RW	c2h_st_len	AXI4-Stream packet length

C2H_CONTROL_REG (0x008)

Table 110: **C2H_CONTROL_REG (0x008)**

Bit	Default	Access Type	Field	Description
[31:6]	0	NA		Reserved
[5]	0	RW		C2H Stream Marker request C2H Stream Marker response will be registered at address 0x18, bit [0].
[4]	0	NA		reserved
[3]	0	RW		Disable completion. For this packet, there will not be any completion.
[2]	0	RW		Immediate data. When set, the data generator sends immediate data. This is a self-clearing bit. Write 1 to initiate transfer.
[1]	0	RW		Starts AXI-ST C2H transfer. This is a self-clearing bit. Write 1 to initiate transfer.
[0]	0	RW		Streaming loop back. When set, the data packet from H2C streaming port in the Card side is looped back to the C2H streaming ports.

For Normal C2H stream packet transfer, set address offset 0x08 to 0x2.

For C2H immediate data transfer, set address offset 0x8 to 0x4.

For C2H/H2C stream loopback, set address offset 0x8 to 0x1.

H2C_CONTROL_REG (0x00C)

Table 111: **H2C_CONTROL_REG (0x00C)**

Bit	Default	Access Type	Field	Description
[31:30]	0	NA		Reserved
[0]	0	RW		Clear match bit for H2C transfer.

H2C_STATUS (0x010)

Table 112: **H2C_STATUS (0x010)**

Bit	Default	Access Type	Field	Description
[31:15]	0	NA		Reserved
[14:4]	0	R		H2C transfer Queue ID
[3:1]	0	NA		Reserved
[0]	0	R		H2C transfer match

C2H_STATUS (0x018)

Table 113: **C2H_STATUS (0x018)**

Bit	Default	Access Type	Field	Description
[31:30]	0	NA		Reserved
[0]	0	R		C2H Marker response

C2H_PACKET_COUNT (0x020)

Table 114: **C2H_PACKET_COUNT (0x020)**

Bit	Default	Access Type	Field	Description
[31:10]	0	NA		Reserved
[9:0]	0	RW		AXI-ST C2H number of packet to transfer

C2H_PREFETCH_TAG(0x024)

Table 115: **C2H_PREFETCH_TAG (0x024)**

Bit	Default	Access Type	Field	Description
[31:27]	0	NA		Reserved
[26:16]	0	RW		Qid for prefetch tag
[15:7]	0	NA		Reserved
[6:0]	0	RW		Prefetch tag value

C2H_COMPLETION_DATA_0 (0x030)

Table 116: **C2H_COMPLETION_DATA_0 (0x030)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [31:0]

C2H_COMPLETION_DATA_1 (0x034)

Table 117: **C2H_COMPLETION_DATA_1 (0x034)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [63:32]

C2H_COMPLETION_DATA_2 (0x038)

Table 118: **C2H_COMPLETION_DATA_2 (0x038)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [95:64]

C2H_COMPLETION_DATA_3 (0x03C)

Table 119: **C2H_COMPLETION_DATA_3 (0x03C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [127:96]

C2H_COMPLETION_DATA_4 (0x040)

Table 120: **C2H_COMPLETION_DATA_4 (0x040)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [159:128]

C2H_COMPLETION_DATA_5 (0x044)

Table 121: **C2H_COMPLETION_DATA_5 (0x044)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [191:160]

C2H_COMPLETION_DATA_6 (0x048)

Table 122: **C2H_COMPLETION_DATA_6 (0x048)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [223:192]

C2H_COMPLETION_DATA_7 (0x04C)

Table 123: **C2H_COMPLETION_DATA_7 (0x04C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [255:224]

C2H_COMPLETION_SIZE (0x050)

Table 124: **C2H_COMPLETION_SIZE (0x050)**

Bit	Default	Access Type	Field	Description
[31:13]	0	NA		Reserved
[12]	0	RW		Completion Type. 1'b1: NO_PLD_BUT_WAIT 1'b0: HAS PLD
[10:8]	0	RW		s_axis_c2h_cmpt_ctrl_err_idx[2:0] Completion Error Bit Index. 3'b000: Selects 0th register. 3'b111: No error bit is reported.
[6:4]	0	RW		s_axis_c2h_cmpt_ctrl_col_idx[2:0] Completion Color Bit Index. 3'b000: Selects 0th register. 3'b111: No color bit is reported.
[3]	0	RW		s_axis_c2h_cmpt_ctrl_user_trig Completion user trigger

Table 124: **C2H_COMPLETION_SIZE (0x050)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[1:0]	0	RW		AXI4-Stream C2H completion data size. 00: 8 Bytes 01: 16 Bytes 10: 32 Bytes 11: 64 Bytes

SCRATCH_REG0 (0x060)

Table 125: **SCRATCH_REG0 (0x060)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW		Scratch register

SCRATCH_REG1 (0x064)

Table 126: **SCRATCH_REG1 (0x064)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW		Scratch register

C2H_PACKETS_DROP (0x088)

Table 127: **C2H_PACKETS_DROP (0x088)**

Bit	Default	Access Type	Field	Description
[31:0]	0	R		The number of AXI-ST C2H packets (descriptors) dropped per transfer

Each AXI-ST C2H transfer can contain one or more descriptors depending on transfer size and C2H buffer size. This register represents how many of the descriptors were dropped in the current transfer. This register will reset to 0 in the beginning of each transfer.

C2H_PACKETS_ACCEPTED (0x08C)

Table 128: **C2H_PACKETS_ACCEPTED (0x08C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	R		The number of AXI-ST C2H packets (descriptors) accepted per transfer

Each AXI-ST C2H transfer can contain one or more descriptors depending on the transfer size and C2H buffer size. This register represents how many of the descriptors were accepted in the current transfer. This register will reset to 0 at the beginning of each transfer.

DESCRIPTOR_BYPASS (0x090)

Table 129: **Descriptor Bypass (0x090)**

Bit	Default	Access Type	Field	Description
[31:3]	0	NA		Reserved
[2:1]	0	RW	c2h_dsc_bypass	C2H descriptor bypass loopback. When set, the C2H descriptor bypass-out port is looped back to the C2H descriptor bypass-in port. 2'b00: No bypass loopback. 2'b01: C2H MM desc bypass loopback and C2H Stream cache bypass loopback. 2'b10: C2H Stream Simple descriptor bypass loopback. 2'b11: H2C stream 64 byte descriptors are looped back to Completion interface.
[0]	0	RW	h2c_dsc_bypass	H2C descriptor bypass loopback. When set, the H2C descriptor bypass-out port is looped back to the H2C descriptor bypass-in port. 1'b1: H2C MM and H2C Stream descriptor bypass loopback 1'b0: No descriptor loopback

USER_INTERRUPT (0x094)

Table 130: **User Interrupt (0x094)**

Bit	Default	Access Type	Field	Description
[31:20]	0	NA		Reserved
[19:12]	0	RW	usr_irq_in_fun	User interrupt function number
[11:9]	0	NA		Reserved
[8:4]	0	RW	usr_irq_in_vec	User interrupt vector number

Table 130: **User Interrupt (0x094)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[3:1]	0	NA		Reserved
[0]	0	RW	usr_irq	User interrupt. When set, the example design generates a user interrupt.

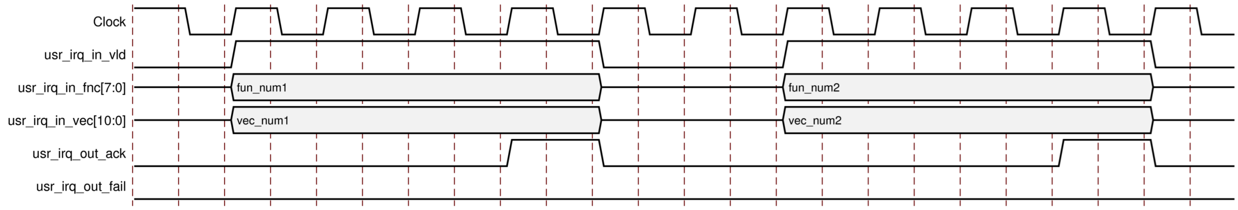
To generate a user interrupt:

1. Write the function number at bits [19:12]. This corresponds to the function that generates the `usr_irq_in_fnc` user interrupt.
2. Write MSI-X Vector number at bits [8:4]. This corresponds to the entry in the MSI-X table that is set up for `usr_irq_in_vec` user interrupt.
3. Write 1 to bit [0] to generate user interrupt. This bit clears itself after `usr_irq_out_ack` from the DMA is generated.

All three above steps can be done at the same time, with a single write.

The user interrupt timing diagram is shown below:

Figure 42: **Interrupt**



USER_INTERRUPT_MASK (0x098)

Table 131: **User Interrupt Mask (0x098)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW		User Interrupt Mask

USER_INTERRUPT_VECTOR (0x09C)

Table 132: **User Interrupt Vector (0x09C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW		User Interrupt Vector

The `user_interrupt_mask[31:0]` and `user_interrupt_vector[31:0]` registers are provided as an example design for user interrupt aggregation that can generate a user interrupt for a function. The `user_interrupt_mask[31:0]` is anded (bitwire and) with `user_interrupt_vector[31:0]` and a user interrupt is generated. The `user_interrupt_vector[31:0]` is clear on read register.

To generate a user interrupt:

1. Write the function number at `user_interrupt[19:12]`. This corresponds to which function generates the `usr_irq_in_fnc` user interrupt.
2. Write the MSI-X Vector number at `user_interrupt[8:4]`. This corresponds to which entry in MSI-X table is set up for the `usr_irq_in_vec` user interrupt.
3. Write mask value in the `user_interrupt_mask[31:0]` register.
4. Write the interrupt vector value in the `user_interrupt_vector[31:0]` register.

This generates a user interrupt to the DMA block.

There are two way to generate user interrupt:

- Write to `user_interrupt[0]`, or
- Write to the `user_interrupt_vector[31:0]` register with mask set.

DMA_CONTROL (0x0A0)

Table 133: **DMA Control (0x0A0)**

Bit	Default	Access Type	Field	Description
[31:1]		NA		Reserved
[0]	0	RW	gen_qdma_reset	When soft_reset is set, generates a soft reset to the DMA block. This bit is cleared after 100 cycles.

Writing a 1 to `DMA_control[0]` generates a soft reset on `soft_reset_n` (active-Low). A reset is asserted for 100 cycles, and following which of the signals will be deasserted.

VDM_MESSAGE_READ (0x0A4)

Table 134: **VDM Message Read (0x0A4)**

Bit	Default	Access Type	Field	Description
[31:0]		RO		VDM message read

Vendor Defined Message (VDM) messages, `st_rx_msg_data`, are stored in fifo in the example design. A read to this register (0x0A4) will pop out one 32-bit message at a time.

Customizing and Generating the Example Design

In the Customize IP dialog box, use the default core parameter values for the IP example design.

After reviewing the core parameters:

1. Right-click the component name.
2. Select **Open IP Example Design**.

This opens a separate example design.

Chapter 6

Test Bench

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

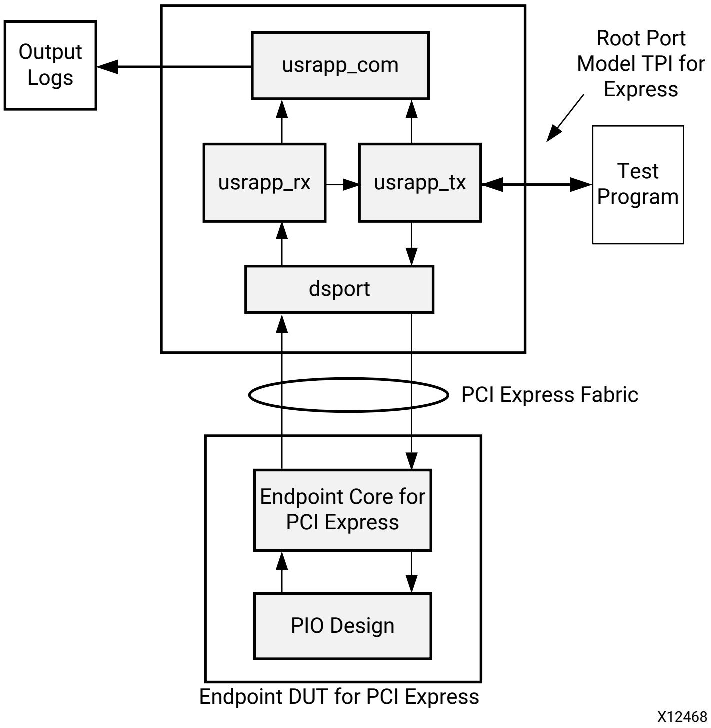
Source code for the Root Port Model is included to provide the model for a starting point for your test bench. All the significant work for initializing the core configuration space, creating TLP transactions, generating TLP logs, and providing an interface for creating and verifying tests is complete.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

The Root Port Model consists of:

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

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

Figure 43: **Root Port Model and Top-Level Endpoint**



Architecture

The Root Port Model consists of these blocks:

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

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

The `usrapp_tx` block sends TLPs to the `dsport` block for transmission across the PCI Express Link to the Endpoint DUT. In turn, the Endpoint DUT device transmits TLPs across the PCI Express Link to the `dsport` block, which are subsequently passed to the `usrapp_rx` block. The `dsport` and core are responsible for the data link layer and physical link layer processing when communicating across the PCI Express logic. Both `usrapp_tx` and `usrapp_rx` use the `usrapp_com` block for shared functions, for example, TLP processing and log file outputting.

Transaction sequences or test programs are initiated by the `usrapp_tx` block to stimulate the Endpoint device fabric interface. TLP responses from the Endpoint device are received by the `usrapp_rx` block. Communication between the `usrapp_tx` and `usrapp_rx` blocks allow the `usrapp_tx` block to verify correct behavior and act accordingly when the `usrapp_rx` block has received TLPs from the Endpoint device.

Scaled Simulation Timeouts

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

Test Selection

All simulation test cases are based on the provided example designs. Simulation tasks perform writes and reads to specific example design registers for setup and checking. These simulation tasks may not work for all customer designs.

Available Tests

The following table describes the tests provided for simulation. These tests are selected based on the QDMA IP configuration. For example, if the AXI4-MM only option is selected, the `qdma_mm_test0` test case is selected and will be executed during simulation.

Table 135: **Test Case**

Option	Test Name	Language	Description
AXI4-MM only	qdma_mm_test0	Verilog	1. The test bench initializes the queue and performs the AXI4-MM transfer in the H2C direction. 2. Then, the test bench initializes the queue and performs the AXI4-MM transfer in the C2H direction. The test bench compares the write data with the read data for correctness.
AXI4-ST only	qdma_st_test0	Verilog	1. The test bench initializes the queue, performs the AXI4-ST transfer in H2C direction, and then checks for data correctness. 2. The test bench initializes the queue, performs the AXI4-ST transfer in the C2H direction, and then check data for correctness.

Table 135: **Test Case** *(cont'd)*

Option	Test Name	Language	Description
AXI4-MM and AXI4-ST with completion	qdma_mm_st_test0	Verilog	1. The test bench initializes the queue and performs the AXI4-MM transfer in the H2C direction. Then, the test bench initializes the queue, performs AXI4-MM in C2H direction, and compares the data for correctness. 2. The test bench initializes the queue, performs the AXI4-ST transfer in the H2C direction, and then checks data for correctness. 3. Then, the test bench initializes the and performs AXI4-ST in the C2H direction and check data for correctness. This test is a combination of test cases `qdma_mm_test0` and `qdma_st_test0`.

Verilog Test Selection

You can change the test case by editing the `usr_pci_exp_usrapp_tx.v` file. Look for the "testname" assignment and modify the test case.

Waveform Dumping

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

Verilog Flow

The model provides a mechanism for outputting the simulation waveform to a file using the `+dump_all` command line parameter.

Output Logging

The simulation process creates three output log files. They are `tx.dat`, `rx.dat`, and `error.dat`. The `rx.dat` and `tx.dat` files each contain a detailed record of every TLP that was received and transmitted, respectively.

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

The `error.dat` file is used in conjunction with the expectation tasks. Only PCIe protocol errors will be listed in `error.dat`. DMA data mismatch or transfer errors will be printed out in the log file.

Test Description

The model provides a Test Program Interface (TPI). The TPI provides the means to create tests by invoking a series of Verilog tasks. All tests should follow these steps:

1. Perform conditional comparison of a unique test name.
2. Wait for reset and link-up.
3. Initialize the queue context for that queue.
4. Transmit packet for the queue.
5. Verify that the test succeeded.

Model Task List

PCIe-Related Tasks

For all the PCIe-related tasks, refer to *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213).

DMA Tasks

Table 136: **DMA Tasks**[1]

Name	Input(s)		Output	Description
TSK_QDMA_MM_H2C_TEST	qid, dsc_bypass irq_en	10:0 - -		This task will do a H2C AXI-MM transfer on Queue ID "qid". if "dsc_bypass" is given task will do descriptor bypass If irq_en is giver, task will setup MSI-X table and will send interrupt once transfer is completed.
TSK_QDMA_MM_C2H_TEST	qid, dsc_bypass irq_en	10:0 - -		This task will do a C2H AXI-MM transfer on Queue ID "qid". if "dsc_bypass" is given task will do descriptor bypass If irq_en is giver, task will setup MSI-X table and will send interrupt once transfer is completed.

Table 136: **DMA Tasks**[1] *(cont'd)*

Name	Input(s)		Output	Description
TSK_QDMA_ST_C2H_TEST	qid, dsc_bypass	10:0 -		This task will do a C2H AXI-ST transfer on Queue ID "qid". if "dsc_bypass" is given task will do descriptor bypass
TSK_QDMA_ST_H2C_TEST	qid, dsc_bypass	10:0 -		This task will do a H2C AXI-ST transfer on Queue ID "qid". if "dsc_bypass" is given task will do descriptor bypass

Notes:

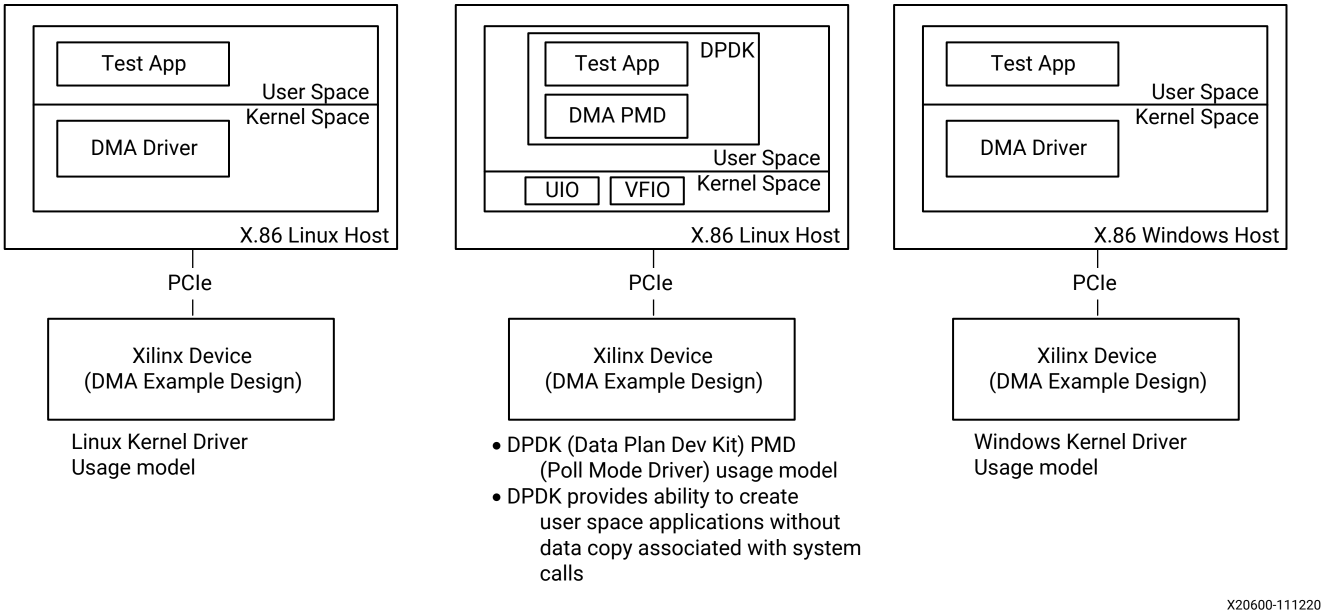
1. The DMA tasks in this table are recommended for testing purposes only.

Chapter 7

Application Software Development

Device Drivers

Figure 44: **Device Drivers**

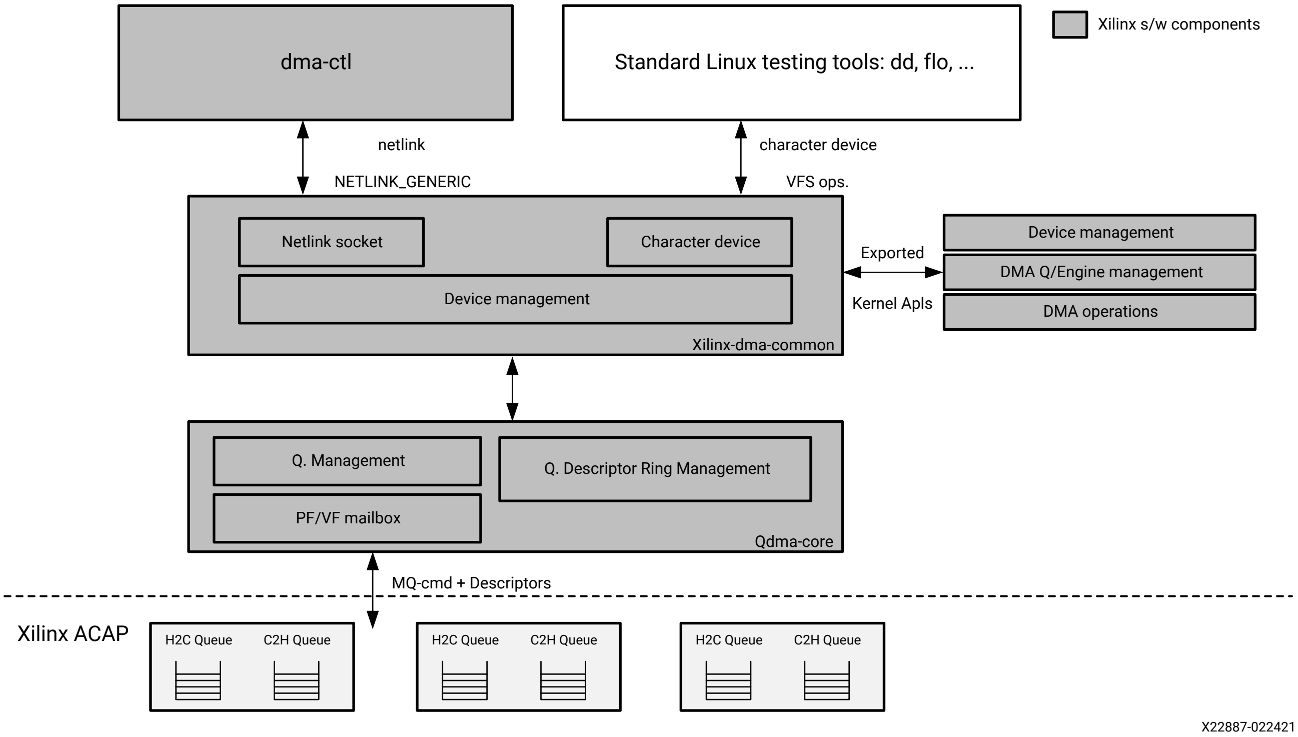


The above figure shows the usage model of Linux and Windows QDMA software drivers. The QDMA example design is implemented on a Xilinx® ACAP, which is connected to an X86 host through PCI Express®.

- In the first use mode, the QDMA driver in kernel space runs on Linux, whereas the test application runs in user space.
- In the second use mode, the Data Plane Dev Kit (DPDK) is used to develop a QDMA Poll Mode Driver (PMD) running entirely in the user space, and use the UIO and VFIO kernel framework to communicate with the ACAP.
- In the third usage mode, the QDMA driver runs in kernel space on Windows, whereas the test application runs in the user space.

Linux QDMA Software Architecture (PF/VF)

Figure 45: **Linux DMA Software Architecture**



The QDMA driver consists of the following three major components:

- **Device control tool**: Creates a netlink socket for PCIe device query, queue management, reading the context of a queue, etc.
- **DMA tool**: Is the user space application to initiate a DMA transaction. You can use standard Linux utility `dd` or `fio`, or use the example application in the driver package.
- **Kernel space driver**: Creates the descriptors and translates the user space function into low-level command to interact with the ACAP.

Using the Drivers

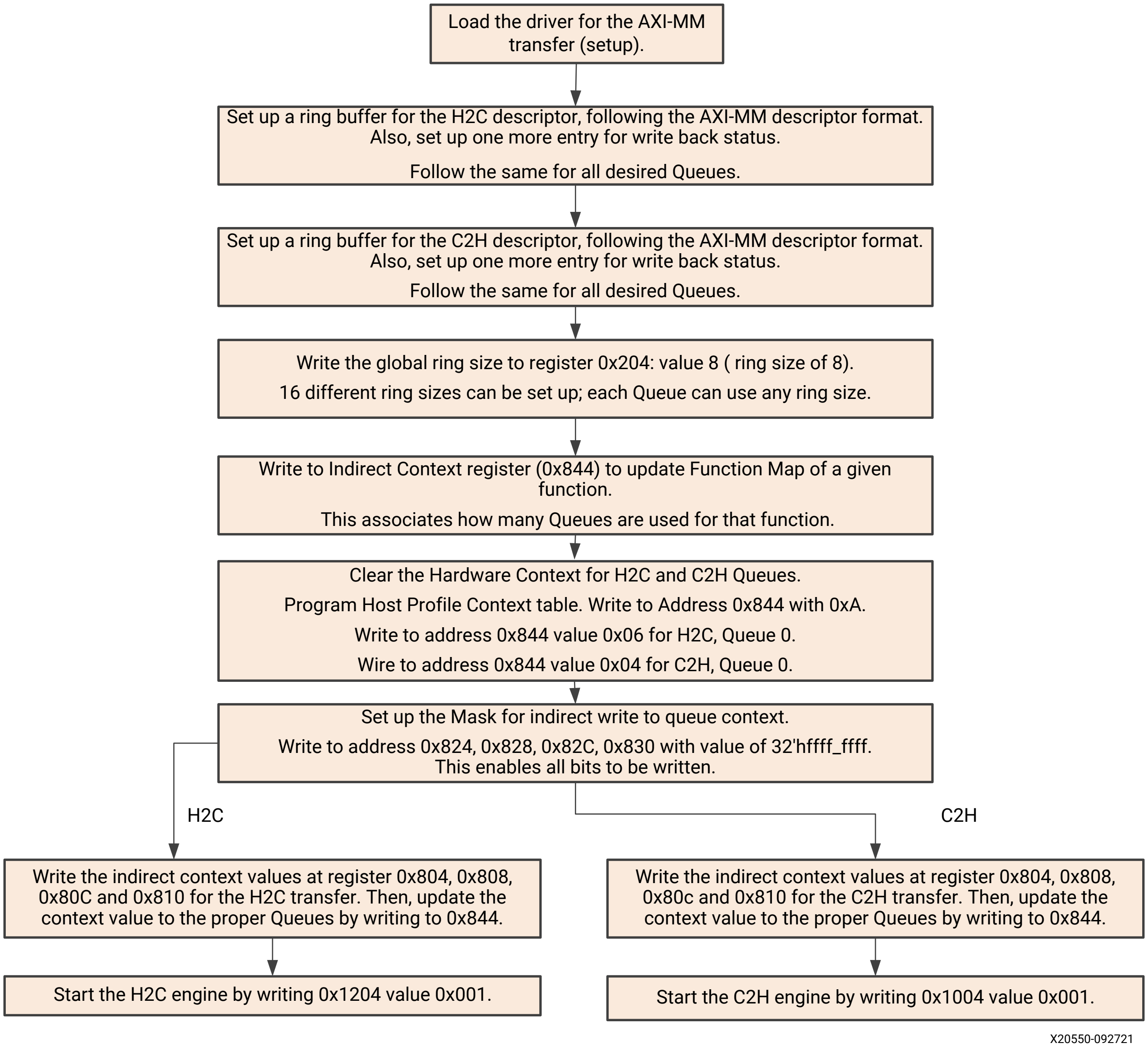
Linux, DPDK and Windows drivers and the corresponding documentation are available at Xilinx DMA IP Drivers.

***Note*:** Starting from 2022.1 release of the Linux driver for QDMA, if a design is using streaming queues, they must be explicitly enabled via API as they are not configured at module load. If a design is using tandem PCIe methodology at power-on, the enablement must occur after Stage 2 is loaded to the device.

Reference Software Driver Flow

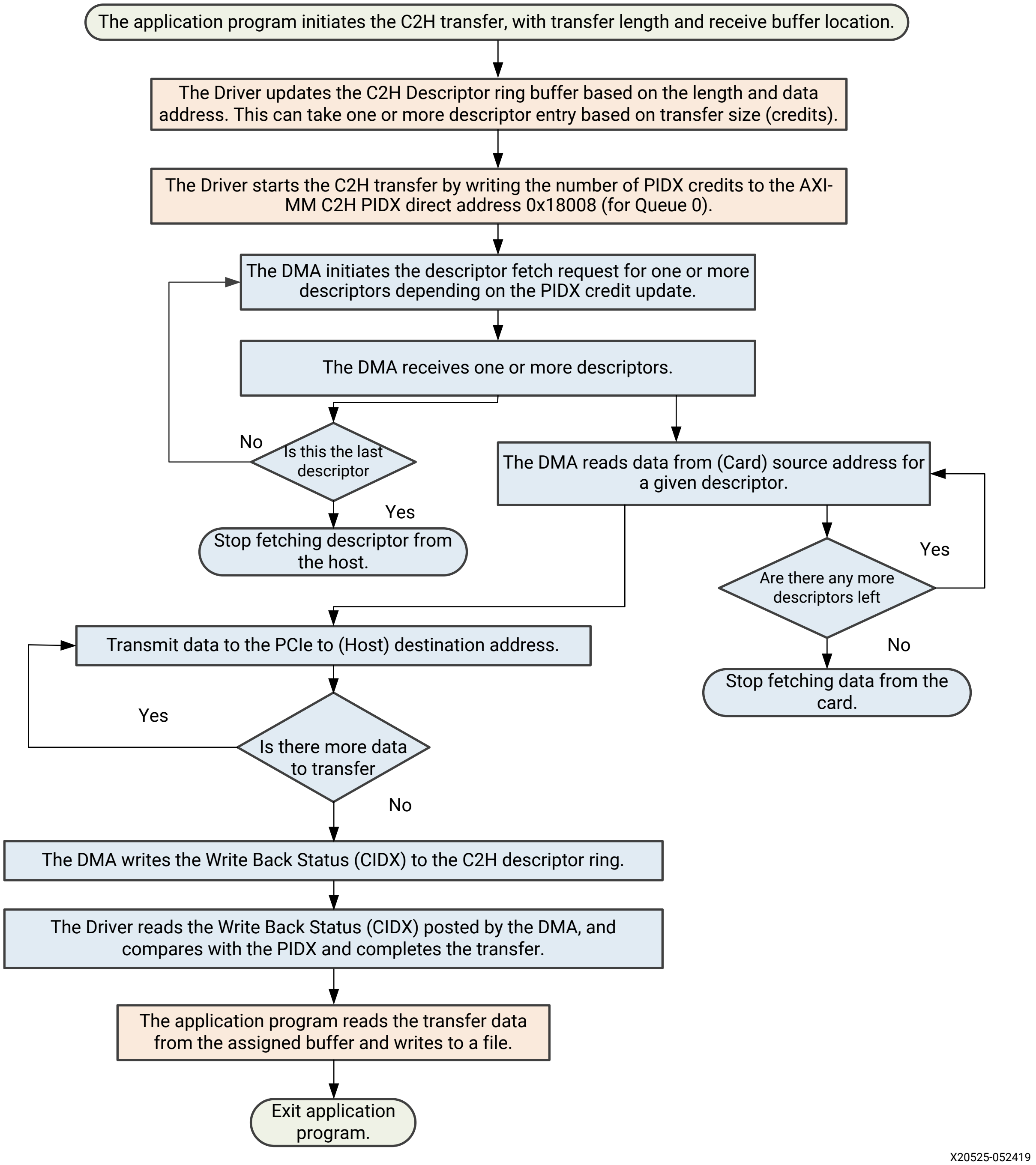
AXI4 Memory Map Flow Chart

Figure 46: **AXI4 Memory Map Flow Chart**


Load the driver for the AXI-MM transfer (setup).
Set up a ring buffer for the H2C descriptor, following the AXI-MM descriptor format. Also, set up one more entry for write back status.
Follow the same for all desired Queues.
Set up a ring buffer for the C2H descriptor, following the AXI-MM descriptor format. Also, set up one more entry for write back status.
Follow the same for all desired Queues.
Write the global ring size to register 0x204: value 8 (ring size of 8).
16 different ring sizes can be set up; each Queue can use any ring size.
Write to Indirect Context register (0x844) to update Function Map of a given function.
This associates how many Queues are used for that function.
Clear the Hardware Context for H2C and C2H Queues.
Program Host Profile Context table. Write to Address 0x844 with 0xA.
Write to address 0x844 value 0x06 for H2C, Queue 0.
Wire to address 0x844 value 0x04 for C2H, Queue 0.
Set up the Mask for indirect write to queue context.
Write to address 0x824, 0x828, 0x82C, 0x830 with value of 32'hffff_ffff. This enables all bits to be written.
H2C
C2H
Write the indirect context values at register 0x804, 0x808, 0x80C and 0x810 for the H2C transfer. Then, update the context value to the proper Queues by writing to 0x844.
Write the indirect context values at register 0x804, 0x808, 0x80c and 0x810 for the C2H transfer. Then, update the context value to the proper Queues by writing to 0x844.
Start the H2C engine by writing 0x1204 value 0x001.
Start the C2H engine by writing 0x1004 value 0x001.
X20550-092721

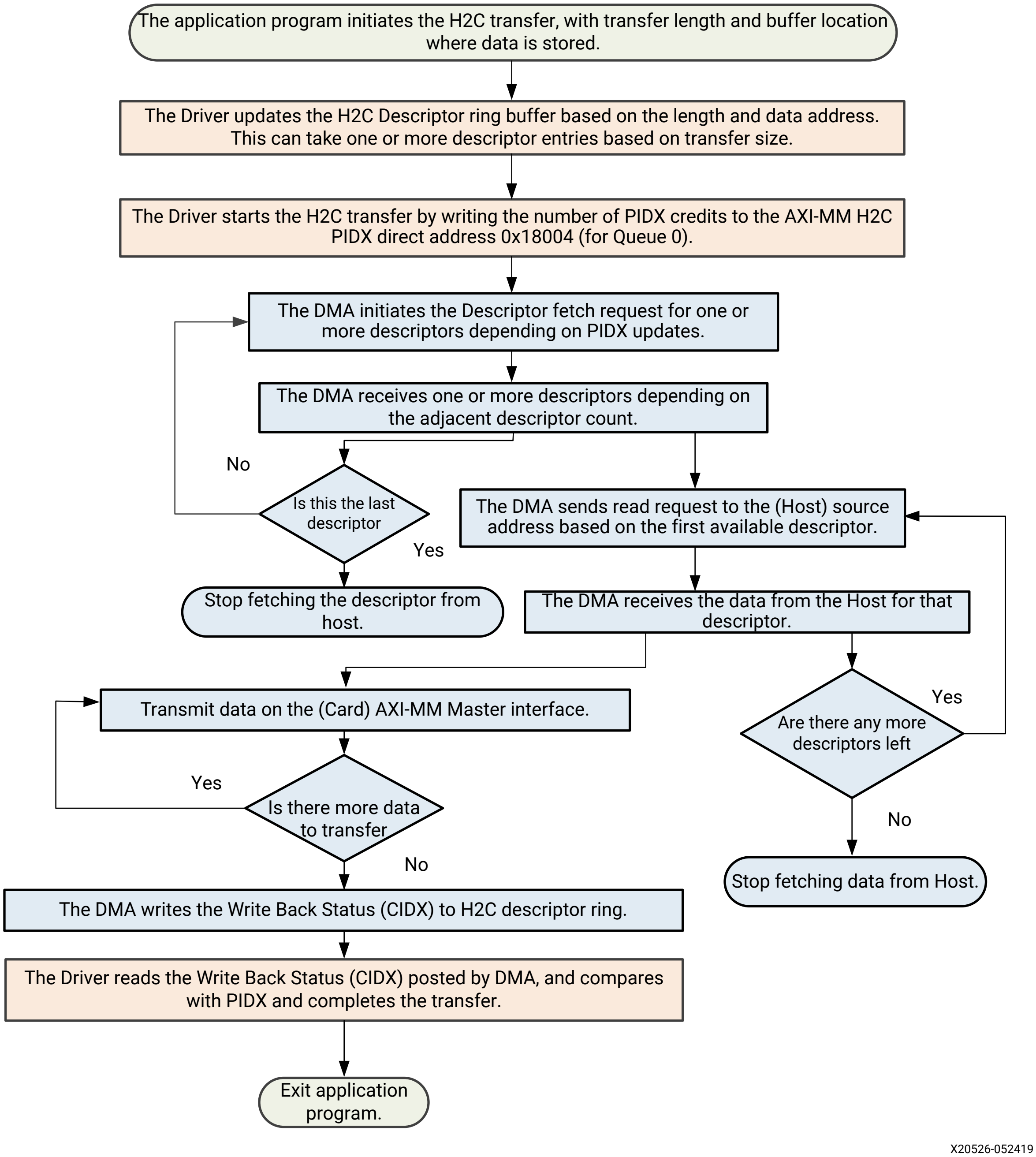
AXI4 Memory Mapped C2H Flow

Figure 47: **AXI4 Memory Mapped Card to Host (C2H) Flow Diagram**


The application program initiates the C2H transfer, with transfer length and receive buffer location.
The Driver updates the C2H Descriptor ring buffer based on the length and data address. This can take one or more descriptor entry based on transfer size (credits).
The Driver starts the C2H transfer by writing the number of PIDX credits to the AXI-MM C2H PIDX direct address 0x18008 (for Queue 0).
The DMA initiates the descriptor fetch request for one or more descriptors depending on the PIDX credit update.
The DMA receives one or more descriptors.
Is this the last descriptor
No
Yes
Stop fetching descriptor from the host.
The DMA reads data from (Card) source address for a given descriptor.
Are there any more descriptors left
Yes
No
Stop fetching data from the card.
Transmit data to the PCIe to (Host) destination address.
Is there more data to transfer
Yes
No
The DMA writes the Write Back Status (CIDX) to the C2H descriptor ring.
The Driver reads the Write Back Status (CIDX) posted by the DMA, and compares with the PIDX and completes the transfer.
The application program reads the transfer data from the assigned buffer and writes to a file.
Exit application program.
X20525-052419

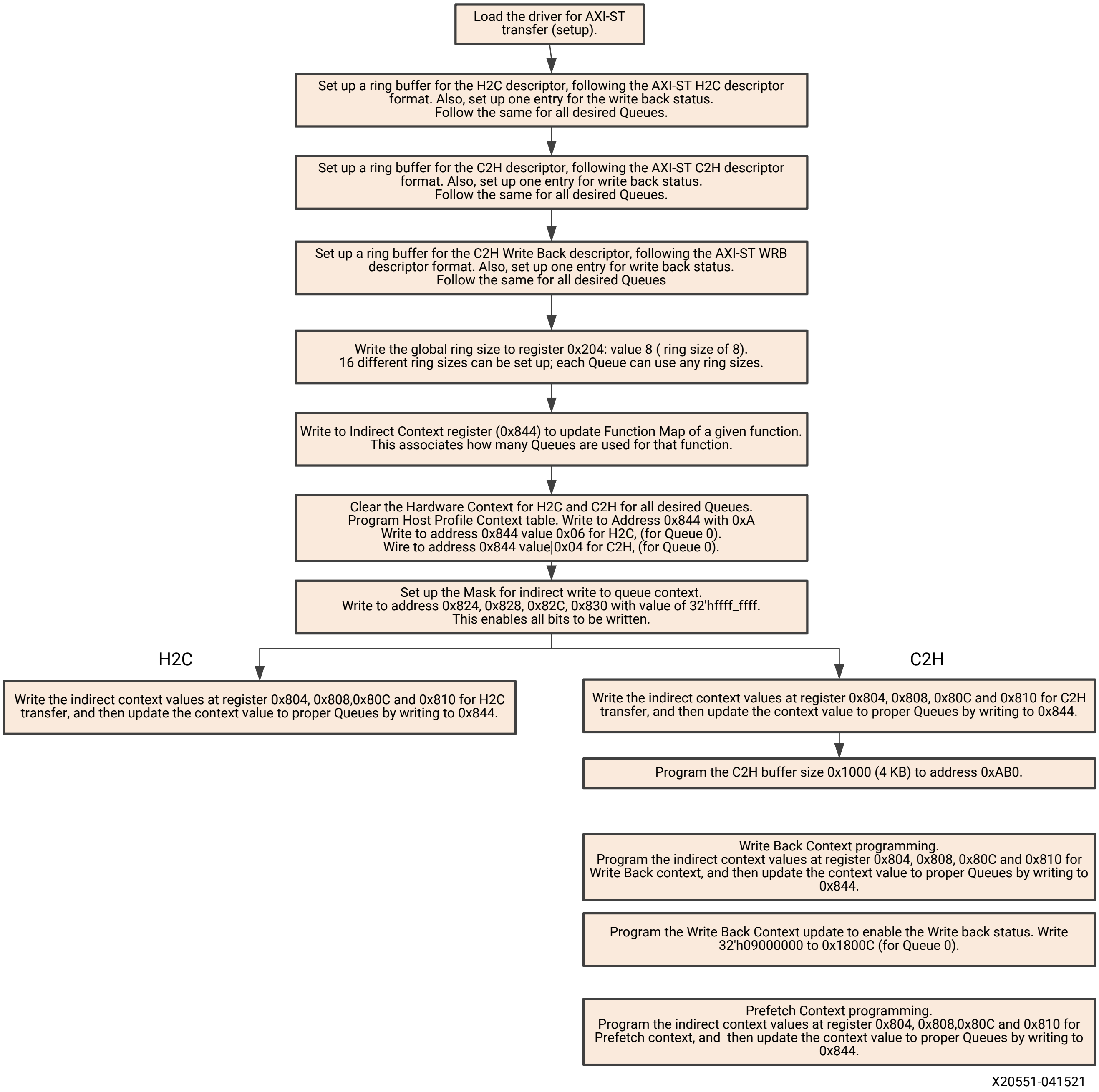
AXI4 Memory Mapped H2C Flow

Figure 48: **AXI4 Memory Mapped Host to Card (H2C) Flow Diagram**


The application program initiates the H2C transfer, with transfer length and buffer location where data is stored.
The Driver updates the H2C Descriptor ring buffer based on the length and data address. This can take one or more descriptor entries based on transfer size.
The Driver starts the H2C transfer by writing the number of PIDX credits to the AXI-MM H2C PIDX direct address 0x18004 (for Queue 0).
The DMA initiates the Descriptor fetch request for one or more descriptors depending on PIDX updates.
The DMA receives one or more descriptors depending on the adjacent descriptor count.
No
Is this the last descriptor
Yes
Stop fetching the descriptor from host.
The DMA sends read request to the (Host) source address based on the first available descriptor.
The DMA receives the data from the Host for that descriptor.
Transmit data on the (Card) AXI-MM Master interface.
Yes
Is there more data to transfer
No
Yes
Are there any more descriptors left
No
Stop fetching data from Host.
The DMA writes the Write Back Status (CIDX) to H2C descriptor ring.
The Driver reads the Write Back Status (CIDX) posted by DMA, and compares with PIDX and completes the transfer.
Exit application program.
X20526-052419

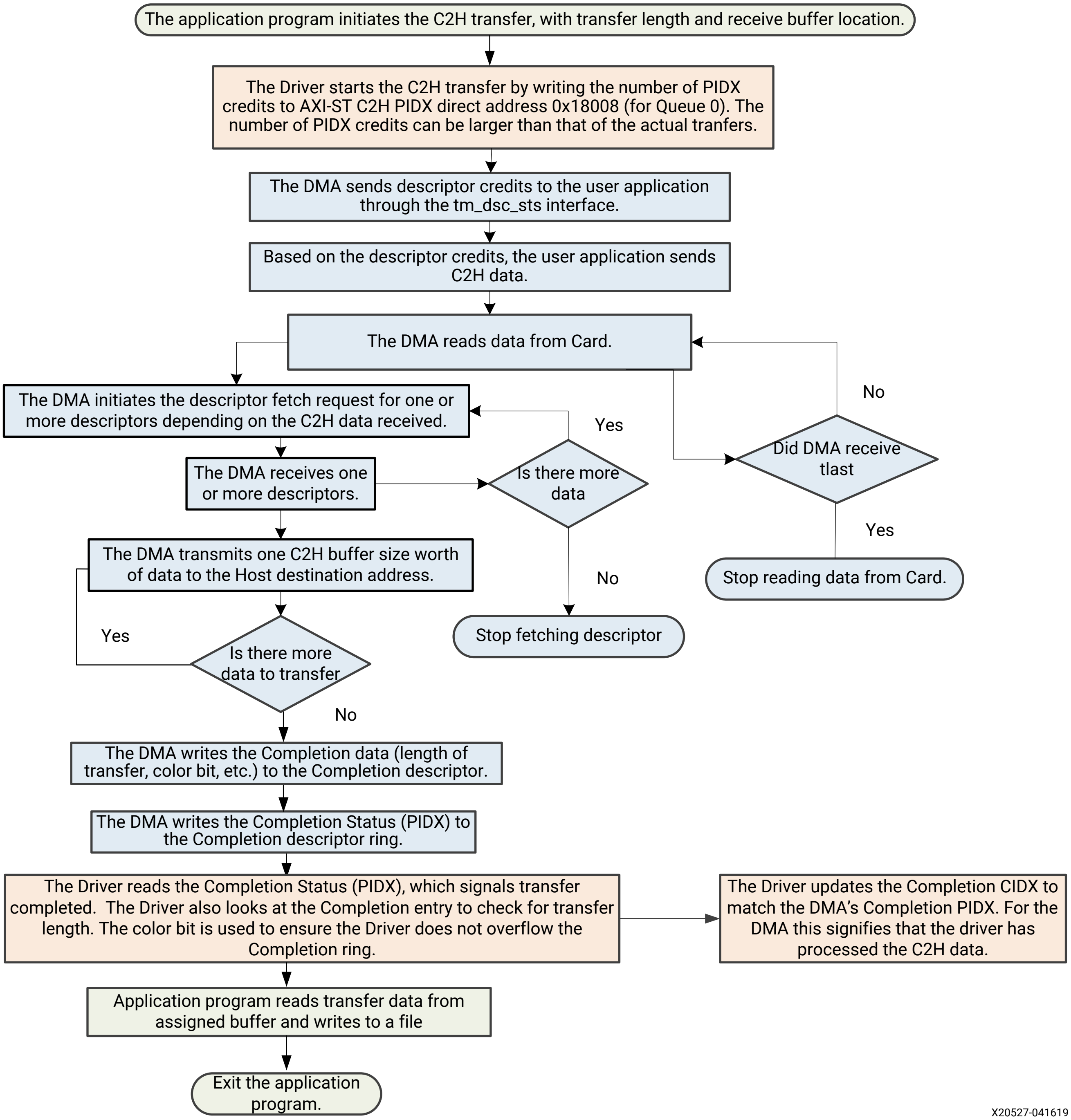
AXI4-Stream Flow Chart

Figure 49: **AXI4-Stream Flow Chart**


Load the driver for AXI-ST transfer (setup).
Set up a ring buffer for the H2C descriptor, following the AXI-ST H2C descriptor format. Also, set up one entry for the write back status. Follow the same for all desired Queues.
Set up a ring buffer for the C2H descriptor, following the AXI-ST C2H descriptor format. Also, set up one entry for write back status. Follow the same for all desired Queues.
Set up a ring buffer for the C2H Write Back descriptor, following the AXI-ST WRB descriptor format. Also, set up one entry for write back status. Follow the same for all desired Queues
Write the global ring size to register 0x204: value 8 (ring size of 8). 16 different ring sizes can be set up; each Queue can use any ring sizes.
Write to Indirect Context register (0x844) to update Function Map of a given function. This associates how many Queues are used for that function.
Clear the Hardware Context for H2C and C2H for all desired Queues. Program Host Profile Context table. Write to Address 0x844 with 0xA Write to address 0x844 value 0x06 for H2C, (for Queue 0). Wire to address 0x844 value 0x04 for C2H, (for Queue 0).
Set up the Mask for indirect write to queue context. Write to address 0x824, 0x828, 0x82C, 0x830 with value of 32'hffff_ffff. This enables all bits to be written.
H2C
C2H
Write the indirect context values at register 0x804, 0x808,0x80C and 0x810 for H2C transfer, and then update the context value to proper Queues by writing to 0x844.
Write the indirect context values at register 0x804, 0x808, 0x80C and 0x810 for C2H transfer, and then update the context value to proper Queues by writing to 0x844.
Program the C2H buffer size 0x1000 (4 KB) to address 0xAB0.
Write Back Context programming. Program the indirect context values at register 0x804, 0x808, 0x80C and 0x810 for Write Back context, and then update the context value to proper Queues by writing to 0x844.
Program the Write Back Context update to enable the Write back status. Write 32'h09000000 to 0x1800C (for Queue 0).
Prefetch Context programming. Program the indirect context values at register 0x804, 0x808,0x80C and 0x810 for Prefetch context, and then update the context value to proper Queues by writing to 0x844.
X20551-041521

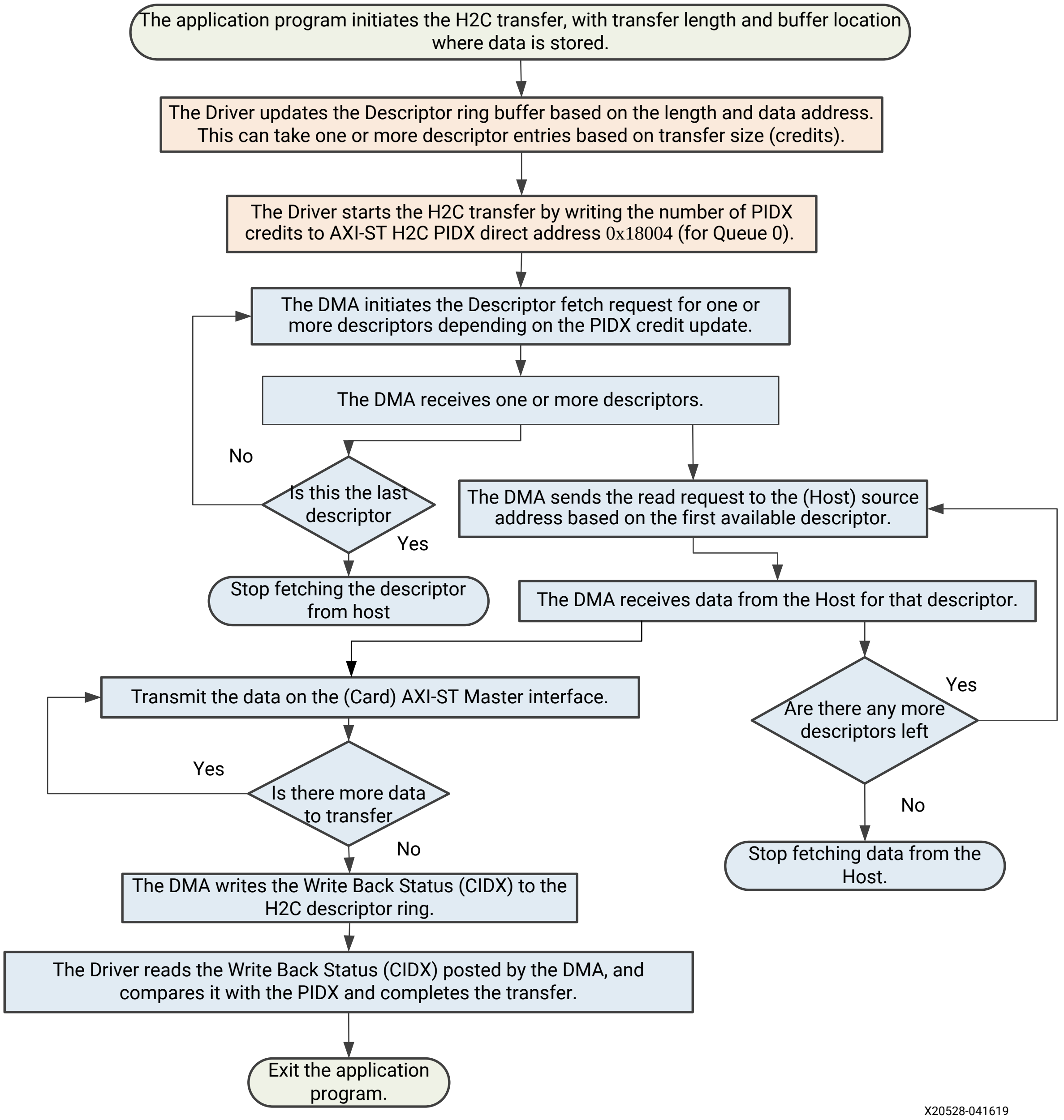
AXI4-Stream C2H Flow

Figure 50: **AXI4-Stream C2H Flow Diagram**


The application program initiates the C2H transfer, with transfer length and receive buffer location.
The Driver starts the C2H transfer by writing the number of PIDX credits to AXI-ST C2H PIDX direct address 0x18008 (for Queue 0). The number of PIDX credits can be larger than that of the actual tranfers.
The DMA sends descriptor credits to the user application through the tm_dsc_sts interface.
Based on the descriptor credits, the user application sends C2H data.
The DMA reads data from Card.
The DMA initiates the descriptor fetch request for one or more descriptors depending on the C2H data received.
The DMA receives one or more descriptors.
Is there more data
Yes
No
Stop fetching descriptor
Did DMA receive tlast
No
Yes
Stop reading data from Card.
The DMA transmits one C2H buffer size worth of data to the Host destination address.
Is there more data to transfer
Yes
No
The DMA writes the Completion data (length of transfer, color bit, etc.) to the Completion descriptor.
The DMA writes the Completion Status (PIDX) to the Completion descriptor ring.
The Driver reads the Completion Status (PIDX), which signals transfer completed. The Driver also looks at the Completion entry to check for transfer length. The color bit is used to ensure the Driver does not overflow the Completion ring.
The Driver updates the Completion CIDX to match the DMA's Completion PIDX. For the DMA this signifies that the driver has processed the C2H data.
Application program reads transfer data from assigned buffer and writes to a file
Exit the application program.
X20527-041619

AXI4-Stream H2C Flow

Figure 51: **AXI4-Stream H2C Flow Diagram**


The application program initiates the H2C transfer, with transfer length and buffer location where data is stored.
The Driver updates the Descriptor ring buffer based on the length and data address. This can take one or more descriptor entries based on transfer size (credits).
The Driver starts the H2C transfer by writing the number of PIDX credits to AXI-ST H2C PIDX direct address 0x18004 (for Queue 0).
The DMA initiates the Descriptor fetch request for one or more descriptors depending on the PIDX credit update.
The DMA receives one or more descriptors.
No
Is this the last descriptor
Yes
Stop fetching the descriptor from host
The DMA sends the read request to the (Host) source address based on the first available descriptor.
The DMA receives data from the Host for that descriptor.
Yes
Are there any more descriptors left
No
Stop fetching data from the Host.
Transmit the data on the (Card) AXI-ST Master interface.
Yes
Is there more data to transfer
No
The DMA writes the Write Back Status (CIDX) to the H2C descriptor ring.
The Driver reads the Write Back Status (CIDX) posted by the DMA, and compares it with the PIDX and completes the transfer.
Exit the application program.
X20528-041619

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 subsystem, 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 subsystem. 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.

Debug Guide

For more information on PCIe debug, see PCIe Debug K-Map.

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 subsystem 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 Subsystem

AR 75397.

QDMA Debugging Answer Record

AR 000033516.

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.

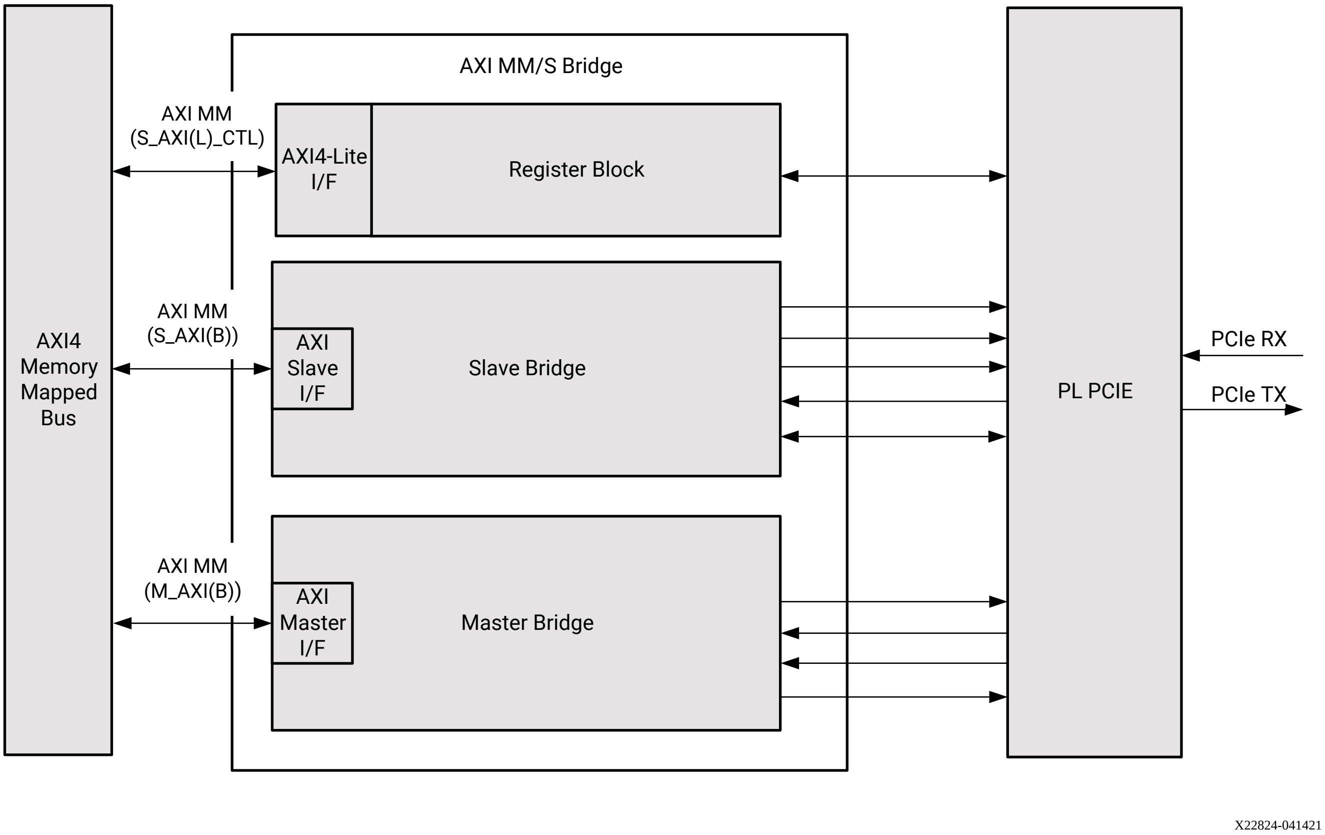
AXI Bridge Subsystem for PL PCIE4 and PL PCIE5

Overview

The AXI Bridge Subsystem is designed for the Vivado® IP integrator in the Vivado® Design Suite. The AXI Bridge Subsystem provides an interface between an AXI4 user logic interface and PCI Express® using the Versal® Integrated Block for PCI Express. The AXI Bridge subsystem provides the translation level between the AXI4 embedded system to the PCI Express system. The AXI Bridge subsystem translates the AXI4 memory read or writes to PCI™ Transaction Layer Packets (TLP) packets and translates PCIe memory read and write request TLP packets to AXI4 interface commands.

The architecture of the AXI Bridge is shown in the following figure.

Figure 52: **High-Level Bridge Architecture**



Modular IP Architecture

The QDMA subsystem configured as AXI Bridge is packaged in what is called a **modular IP architecture**. Modular IP architecture refers to the programmable logic integrated block for PCIe (PL PCIE) IP and the AXI Bridge subsystem appearing as two separate IP that are connected in the Vivado IP integrator.

To generate the IP:

1. In the Vivado IP catalog, locate the **QDMA Subsystem for PCI Express**, and add it to your design.
2. Configure the subsystem as **AXI Bridge**.
3. In the Vivado IP integrator, click **Run Block Automation**. This stitches the PL PCIE and QDMA configured as AXI Bridge together.

Related Information

Customizing and Generating the Subsystem

Feature Summary

The AXI Bridge subsystem is an interface between the AXI4 bus and PCI Express®. The core contains the memory mapped AXI4 to AXI4-Stream Bridge and the AXI4-Stream Enhanced Interface Block for PCIe. The memory-mapped AXI4 to AXI4-Stream Bridge contains a register block and two functional half bridges, referred to as the Slave Bridge and Master Bridge. The slave bridge connects to the AXI4 Interconnect as a slave device to handle any issued AXI4 master read or write requests. The master bridge connects to the AXI4 interconnect as a master to process the PCIe generated read or write TLPs. The core uses a set of interrupts to detect and flag error conditions.

The AXI Bridge subsystem supports both Root Port and Endpoint configurations.

- When configured as an Endpoint, the AXI Bridge subsystem supports up to six 32-bit or three 64-bit PCIe Base Address Registers (BARs).
- When configured as a Root Port, the core supports up to two 32-bit or a single 64-bit PCIe BAR.

The AXI Bridge subsystem is compliant with the *PCI-SIG Specifications* (https://www.pcisig.com/specifications) and with the *AMBA AXI and ACE Protocol Specification* (ARM IHI0022E).

AXI Bridge Subsystem Limitations

1. For this subsystem, the bridge master and bridge slave cannot achieve more than 128 Gb/s.
2. Bridge will be compliant with all MPS and MRRS settings; however, all traffic initiated from the Bridge will be limited to 256 Bytes (max).
3. AXI address width is limited to 48 bits.

PCIe Capability

Only the following PCIe capabilities are supported because of the AXI4 specification:

- 4 PFs
- MSI-X
- PM
- Advanced error reporting (AER)

Chapter 9

Product Specification

The Register block contains registers used in the AXI Bridge subsystem for dynamically mapping the AXI4 memory mapped (MM) address range provided using the AXIBAR parameters to an address for PCIe® range.

The slave bridge provides termination of memory-mapped AXI4 transactions from an AXI master device (such as a processor). The slave bridge provides a way to translate addresses that are mapped within the AXI4 memory mapped address domain to the domain addresses for PCIe. Write transactions to the Slave Bridge are converted into one or more `MemWr` TLPs, depending on the configured Max Payload Size setting, which are passed to the integrated block for PCI Express. The slave bridge can support up to 32 active AXI4 Write requests. When a remote AXI master initiates a read transaction to the slave bridge, the read address and qualifiers are captured and a MemRd request TLP is passed to the core and a completion timeout timer is started. Completions received through the core are correlated with pending read requests and read data is returned to the AXI master. The slave bridge can support up to 32 active AXI4 Read requests with pending completions.

The master bridge processes both PCIe MemWr and MemRd request TLPs received from the Integrated Block for PCI Express and provides a means to translate addresses that are mapped within the address for PCIe domain to the memory mapped AXI4 address domain. Each PCIe `MemWr` request TLP header is used to create an address and qualifiers for the memory mapped AXI4 bus and the associated write data is passed to the addressed memory mapped AXI4 Slave. The Master Bridge can support up to 32 active AXI4 Read requests and AXI4 Write requests.

Each PCIe `MemRd` request TLP header is used to create an address and qualifiers for the memory-mapped AXI4 bus. Read data is collected from the addressed memory mapped AXI4 slave and used to generate completion TLPs which are then passed to the integrated block for PCI Express. The Master Bridge can support up to 32 active PCIe `MemRd` request TLPs with pending completions for improved AXI4 pipelining performance.

The instantiated AXI4-Stream Enhanced PCIe block contains submodules including the Requester/Completer interfaces to the AXI bridge and the Register block. The Register block contains the status, control, interrupt registers, and the AXI4-Lite interface.

Performance and Resource Utilization

Because the AXI Bridge Subsystem is based on the QDMA Subsystem (configured as AXI Bridge), refer to the QDMA Subsystem Performance and Resource Utilization section for AXI Bridge data.

Related Information

Performance and Resource Utilization

AXI Bridge Operations

AXI Transactions for PCIe

The following tables are the translation tables for AXI4-Stream and memory-mapped transactions.

Table 137: **AXI4 Memory-Mapped Transactions to AXI4-Stream PCIe TLPs**

AXI4 Memory-Mapped Transaction	AXI4-Stream PCIe TLPs
INCR Burst Read of AXIBAR	MemRd 32 (3DW)
INCR Burst Write to AXIBAR	MemWr 32 (3DW)
INCR Burst Read of AXIBAR	MemRd 64 (4DW)
INCR Burst Write to AXIBAR	MemWr 64 (4DW)

Table 138: **AXI4-Stream PCIe TLPs to AXI4 Memory Mapped Transactions**

AXI4-Stream PCIe TLPs	AXI4 Memory-Mapped Transaction
MemRd 32 (3DW) of PCIEBAR	INCR Burst Read
MemWr 32 (3DW) to PCIEBAR	INCR Burst Write
MemRd 64 (4DW) of PCIEBAR	INCR Burst Read
MemWr 64 (4DW) to PCIEBAR	INCR Burst Write

For PCIe® requests with lengths greater than 1 Dword, the size of the data burst on the Master AXI interface will always equal the width of the AXI data bus even when the request received from the PCIe link is shorter than the AXI bus width.

The `s_axi_wstrb` signal can be used to facilitate data alignment to an address boundary. `s_axi_wstrb` may equal 0 in the beginning of a valid data cycle and will appropriately calculate an offset to the given address. However, the valid data identified by `s_axi_wstrb` must be continuous from the first byte enable to the last byte enable.

Transaction Ordering for PCIe

The AXI Bridge subsystem conforms to PCIe® transaction ordering rules. See the PCI-SIG Specifications for the complete rule set. The following behaviors are implemented in the AXI Bridge subsystem to enforce the PCIe transaction ordering rules on the highly-parallel AXI bus of the bridge.

- The `bresp` to the remote (requesting) AXI4 master device for a write to a remote PCIe device is not issued until the `MemWr` TLP transmission is guaranteed to be sent on the PCIe link before any subsequent TX-transfers.
- If Relaxed Ordering bit is not set within the TLP header, then a remote PCIe device read to a remote AXI slave is not permitted to pass any previous remote PCIe device writes to a remote AXI slave received by the AXI Bridge subsystem. The AXI read address phase is held until the previous AXI write transactions have completed and `bresp` has been received for the AXI write transactions. If the Relaxed Ordering attribute bit is set within the TLP header, then the remote PCIe device read is permitted to pass.
- Read completion data received from a remote PCIe device are not permitted to pass any remote PCIe device writes to a remote AXI slave received by the AXI Bridge subsystem prior to the read completion data. The `bresp` for the AXI write(s) must be received before the completion data is presented on the AXI read data channel.

***Note*:** The transaction ordering rules for PCIe might have an impact on data throughput in heavy bidirectional traffic.

BAR and Address Translation

BDF Table

Address translations for AXI address is done based on BDF table programming (0x2420 to 0x2434). These BDF table entries can be programmed through the AXI4-Lite Slave CSR interface, `s_axil_csr_*`. There are 8 windows provided similar to 8 BARs on PCIe bus. Each entry in BDF table programming represents one window. If a user needs 2 windows then 2 entrees needs to be programmed and so on.

There are some restrictions in programming BDF table.

1. All PCIe slave bridge data transfers must be quiesced before programming the BDF table.
2. There are six registers for each BDF table entry. All six registers must be programmed to make a valid entry. Even if some registers have 0s, you need to program 0s in those registers.
3. All the six registers need to be programmed in an order for an entry to be valid. Order is listed below.
    a. 0x2420
    b. 0x2424

c. 0x2428
d. 0x242C
e. 0x2430
f. 0x2434

BDF table entree start address = 0x2420 + (0x20 * i), where i = table entree number.

Address Translations

Address translation can be turned off by selecting the No Address Translation option during IP configuration. When this option is selected, one full 64-bit BAR space is given for slave data transfer. You must set up any address translation if needed. If No Address Translation is *not* selected DMA will do address translation.

One 64 bit BAR space is divided into 8 (called window) and 8 window space is available for address translation. Address translation for Slave Bridge transfer are done in two steps.

1. Address translation for upper bits are programmed in GUI configuration. In the IP integrator canvas, address translation for upper bits should be edited in the Address Editor tab.
2. Address translation for lower bits are programmed in BDF table.

Address Translation Examples

Example 1: BAR Size of 64 KB, with 1 Window Size 4 KB

Window 0: 4 KB with address translation of 0xF for bits [15:12].

1. Selections in Vivado IP configuration in the AXI BARs tab are as follows:
   - AXI BAR size 64K: **0xFFFF bits [15:0]**
   - Address translation for bits[63:16] can be set in GUI. In this example [63:16] = 0x0
   - Set Aperture Base Address: **0x0000_0000_0000_0000**
   - Set Aperture High Address: **0x0000_0000_0000_FFFF**
2. The BDF table programming:
   - Program 1 entries for 1 window
   - Window Size = AXI BAR size/8 = 64K / 8 = 0x1FFF = 8 KB (13 bits). Each window max size is 8 KB.
   - In this example for window size of 4K, 0x1 is programmed at 0x2430 bits [25:0].
   - Address translation for bits [15:13] are programmed at 0x2420 and 0x2424.
   - In this example, address translation for bits [15:13] are set to 0x7.

Table 139: **BDF Table Programming**

Program Value	Registers
0x0000_E000	Address translation value Low
0x0	Address translation value High
0x0	PASID/ Reserved
0x0	[11:0]: Function Number
0xC0000001	[31:30] Read/Write Access permission [29] : R0 access Error [28:26] Protection ID [25:0] Window Size ([25:0]*4K = actual size of the window)
0x0	reserved

For this example Slave address 0x0000_0000_0000_0100 will be address translated to 0x0000_0000_0000_E100.

Example 2: BAR Size of 64 KB, with 1 Window 8 KB

Window 0: 8 KB with address translation of 0x6 ('b110) for bits [15:13].

1. Selections in Vivado IP configuration in the AXI BARs tab are as follows:
   - AXI BAR size 64K: **0xFFFF bits [15:0]**
   - Address translation for bits [63:16] can be done in GUI. In this example [63:16] = 0x0
   - Set Aperture Base Address: **0x0000_0000_0000_0000**
   - Set Aperture High Address: **0x0000_0000_0000_FFFF**
2. The BDF table programming:
   - Program 1 entries for 1 window.
   - Window Size = AXI BAR size/8 = 64K / 8 = 0x1FFF = 8 KB (13 bits). Each window max size is 8 KB.
   - In this example for window size of 8K, 0x2 is programmed at 0x2430 bits [25:0].
   - Address translation for bits [15:13] are programmed at 0x2420 and 0x2424.
   - In this example, address translation for bits [15:13] are set to 0x6 ('b110).

Table 140: **BDF Table Programming**

Offset	Program Value	Registers
0x2420	0x0000_C000	Address translation value Low
0x2424	0x0	Address translation value High
0x2428	0x0	PASID/ Reserved
0x242C	0x0	[11:0]: Function Number

Table 140: **BDF Table Programming** *(cont'd)*

Offset	Program Value	Registers
0x2430	0xC0000002	[31:30] Read/Write Access permission [29] : R0 access Error [28:26] Protection ID [25:0] Window Size ([25:0]*4K = actual size of the window)
0x2434	0x0	reserved

For this example, the Slave address 0x0000_0000_0000_0100 will be address translated to 0x0000_0000_0000_C100.

Example 3: BAR Size of 32 GB, and 4 Windows of Various Sizes

- Window 0: 4 KB with address translation of 0xAAAAA for bits [31:12].
- Window 1: 4 GB with no address translation on window.
- Window 2: 64 KB with address translation of 0xBBBB for bits [31:16].
- Window 3: 1 GB with address translation of 11111 for bits [34:30].

1. Selections in Vivado IP configuration in the AXI BARs tab are as follows:
    - AXI BAR size 32G: **0x7_FFFF_FFFF bits [34:0]**.
    - Address translation for bits [63:35] can be programmed in GUI. In this example [63:36] = 0xAB.
    - Set Aperture Base Address: **0x0000_0AB0_0000_0000**.
    - Set Aperture High Address: **0x0000_0AB7_FFFF_FFFF**.
2. The BDF table programming:
    - Window Size = AXI BAR size/8 = 32 GB / 8 = 0xFFFF_FFFF = 4 GB (32 bits). Each window max size is 4 GB.
    - Program 4 entries for 4 windows:
        - BDF entry 0 table starts at 0x2420.
        - BDF entry 1 table starts at 0x2440.
        - BDF entry 2 table starts at 0x2460.
        - BDF entry 3 table starts at 0x2480.
    - Window 0 size 4 KB.
        - Program 0x1 to 0x2430 bits [25:0].
        - Address translation for bits [34:32] are programmed at 0x2420 and 0x2424.

    - Program 0x0000_0000 to 0x2420.
    - Program 0x0000_0007 to 0x2424
- Window 1 size 4 GB.
    - Program 0x10_0000 to 0x2450 bits [25:0].
    - No address translation because all address bits will be used by the window.
    - Program 0x0000_0000 to 0x2440.
    - Program 0x0000_0000 to 0x2444
- Window 2 size 64 KB.
    - Program 0x10 to 0x2470 bits [25:0].
    - Address translation for bits [34:32] are programmed at 0x2460 and 0x2464.
    - Program 0x0000_0000 to 0x2460
    - Program 0x0000_00005 to 0x2464
- Window 3 size 1 GB.
    - Program 0x4_0000 to 0x2490 bits [25:0].
    - Address translation for bits [34:30] are programmed at 0x2480 and 0x2484.
    - Program 0x0000_0000 to 0x2480.
    - Program 0x0000_0003 to 0x2484

Table 141: **BDF Table Programming Entry 0**

Offset	Program Value	Registers
0x2420	0x0000_0000	Address translation value Low
0x2424	0x7	Address translation value High
0x2428	0x0	PASID/ Reserved
0x242C	0x0	[11:0]: Function Number
0x2430	0xC0000001	[31:30] Read/Write Access permission [29] : R0 access Error [28:26] Protection ID [25:0] Window Size ([25:0]*4K = actual size of the window)
0x2434	0x0	reserved

Table 142: **BDF Table Programming Entry 1**

Offset	Program Value	Registers
0x2440	0x0000	Address translation value Low

Table 142: **BDF Table Programming Entry 1** *(cont'd)*

Offset	Program Value	Registers
0x2444	0x0	Address translation value High
0x2448	0x0	PASID/ Reserved
0x244C	0x0	[11:0]: Function Number
0x2450	0xC010_0000	[31:30] Read/Write Access permission [29] : R0 access Error [28:26] Protection ID [25:0] Window Size ([25:0]*4K = actual size of the window)
0x2454	0x0	reserved

Table 143: **BDF Table Programming Entry 2**

Offset	Program Value	Registers
0x2460	0x_0000	Address translation value Low
0x2464	0x05	Address translation value High
0x2468	0x0	PASID/ Reserved
0x246C	0x0	[11:0]: Function Number
0x2470	0xC000_00010	[31:30] Read/Write Access permission [29] : R0 access Error [28:26] Protection ID [25:0] Window Size ([25:0]*4K = actual size of the window)
0x2474	0x0	reserved

Table 144: **BDF Table Programming Entry 3**

Offset	Program Value	Registers
0x2480	0x0000_0000	Address translation value Low
0x2484	0x3	Address translation value High
0x2488	0x0	PASID/ Reserved
0x248C	0x0	[11:0]: Function Number
0x2490	0xC004_0000	[31:30] Read/Write Access permission [29] : R0 access Error [28:26] Protection ID [25:0] Window Size ([25:0]*4K = actual size of the window)
0x2494	0x0	reserved

For this example

the Slave address 0x0000_0000_0000_0100 will be address translated to 0x0000_0AB7_0000_0100.

the Slave address 0x0000_0001_0000_0100 will be address translated to 0x0000_0AB0_0000_0100.

the Slave address 0x0000_0002_0000_0100 will be address translated to 0x0000_0AB5_0000_0100.

the Slave address 0x0000_0003_0000_0100 will be address translated to 0x0000_0AB3_0000_0100.

The slave bridge does not support narrow burst AXI transfers. To avoid narrow burst transfers, connect the AXI smart-connect module which will convert narrow burst to full burst AXI transfers.

Malformed TLP

The integrated block for PCI Express® detects a malformed TLP. For the IP configured as an Endpoint core, a malformed TLP results in a fatal error message being sent upstream if error reporting is enabled in the Device Control register.

Abnormal Conditions

This section describes how the Slave side and Master side (see the following tables) of the AXI Bridge subsystem handle abnormal conditions.

Slave Bridge Abnormal Conditions

Slave bridge abnormal conditions are classified as: Illegal Burst Type and Completion TLP Errors. The following sections describe the manner in which the Bridge handles these errors.

Illegal Burst Type

The slave bridge monitors AXI read and write burst type inputs to ensure that only the INCR (incrementing burst) type is requested. Any other value on these inputs is treated as an error condition and the Slave Illegal Burst (SIB) interrupt is asserted. In the case of a read request, the Bridge asserts SLVERR for all data beats and arbitrary data is placed on the `s_axi_rdata` bus. In the case of a write request, the Bridge asserts SLVERR for the write response and all write data is discarded.

Completion TLP Errors

Any request to the bus for PCIe (except for a posted Memory write) requires a completion TLP to complete the associated AXI request. The Slave side of the Bridge checks the received completion TLPs for errors and checks for completion TLPs that are never returned (Completion Timeout). Each of the completion TLP error types are discussed in the subsequent sections.

Unexpected Completion

When the slave bridge receives a completion TLP, it matches the header RequesterID and Tag to the outstanding RequesterID and Tag. A match failure indicates the TLP is an Unexpected Completion which results in the completion TLP being discarded and a Slave Unexpected Completion (SUC) interrupt strobe being asserted. Normal operation then continues.

Unsupported Request

A device for PCIe might not be capable of satisfying a specific read request. For example, if the read request targets an unsupported address for PCIe, the completer returns a completion TLP with a completion status of `0b001 - Unsupported Request`. The completer that returns a completion TLP with a completion status of `Reserved` must be treated as an unsupported request status, according to the PCI Express Base Specification v3.0. When the slave bridge receives an unsupported request response, the Slave Unsupported Request (SUR) interrupt is asserted and the DECERR response is asserted with arbitrary data on the AXI4 memory mapped bus.

Completion Timeout

A Completion Timeout occurs when a completion (Cpl) or completion with data (CplD) TLP is not returned after an AXI to PCIe memory read request, or after a PCIe Configuration Read/Write request. For PCIe Configuration Read/Write request, completions must complete within the `C_COMP_TIMEOUT` parameter selected value from the time the request is issued. For PCIe Memory Read request, completions must complete within the value set in the Device Control 2 register in the PCIe Configuration Space register. When a completion timeout occurs, an OKAY response is asserted with all 0s and 1s data on the memory mapped AXI4 bus.

Poison Bit Received on Completion Packet

An Error Poison occurs when the completion TLP EP bit is set, indicating that there is poisoned data in the payload. When the slave bridge detects the poisoned packet, the Slave Error Poison (SEP) interrupt is asserted and the SLVERR response is asserted with arbitrary data on the memory mapped AXI4 bus.

Completer Abort

A Completer Abort occurs when the completion TLP completion status is `0b100` - Completer Abort. This indicates that the completer has encountered a state in which it was unable to complete the transaction. When the slave bridge receives a completer abort response, the Slave Completer Abort (SCA) interrupt is asserted and the SLVERR response is asserted with arbitrary data on the memory mapped AXI4 bus.

Table 145: **Slave Bridge Response to Abnormal Conditions**

Transfer Type	Abnormal Condition	Bridge Response
Read	Illegal burst type	SIB interrupt is asserted. SLVERR response given with arbitrary read data.
Write	Illegal burst type	SIB interrupt is asserted. Write data is discarded. SLVERR response given.
Read	Unexpected completion	SUC interrupt is asserted. Completion is discarded.
Read	Unsupported Request status returned	SUR interrupt is asserted. DECERR response given with arbitrary read data.
Read	Completion timeout	SCT interrupt is asserted. SLVERR response given with arbitrary read data.
Read	Poison bit in completion	Completion data is discarded. SEP interrupt is asserted. SLVERR response given with arbitrary read data.
Read	Completer Abort (CA) status returned	SCA interrupt is asserted. SLVERR response given with arbitrary read data.

Master Bridge Abnormal Conditions

The following sections describe the manner in which the master bridge handles abnormal conditions.

AXI DECERR Response

When the master bridge receives a DECERR response from the AXI bus, the request is discarded and the Master DECERR (MDE) interrupt is asserted. If the request was non-posted, a completion packet with the Completion Status = Unsupported Request (UR) is returned on the bus for PCIe.

AXI SLVERR Response

When the master bridge receives a SLVERR response from the addressed AXI slave, the request is discarded and the Master SLVERR (MSE) interrupt is asserted. If the request was non-posted, a completion packet with the Completion Status = Completer Abort (CA) is returned on the bus for PCIe.

Max Payload Size for PCIe, Max Read Request Size or 4K Page Violated

It is the responsibility of the requester to ensure that the outbound request adheres to the Max Payload Size, Max Read Request Size, and 4 Kb Page Violation rules. If the master bridge receives a request that violates one of these rules, the bridge processes the invalid request as a valid request, which can return a completion that violates one of these conditions or can result in the loss of data. The master bridge does not return a malformed TLP completion to signal this violation.

Completion Packets

When the `MAX_READ_REQUEST_SIZE` is greater than the `MAX_PAYLOAD_SIZE`, a read request for PCIe can ask for more data than the master bridge can insert into a single completion packet. When this situation occurs, multiple completion packets are generated up to `MAX_PAYLOAD_SIZE`, with the Read Completion Boundary (RCB) observed.

Poison Bit

When the poison bit is set in a transaction layer packet (TLP) header, the payload following the header is corrupted. When the master bridge receives a memory request TLP with the poison bit set, it discards the TLP and asserts the Master Error Poison (MEP) interrupt strobe.

Zero Length Requests

When the master bridge receives a read request with the Length = `0x1`, FirstBE = `0x00`, and LastBE = `0x00`, it responds by sending a completion with `Status = Successful Completion`.

When the master bridge receives a write request with the Length = `0x1`, FirstBE = `0x00`, and LastBE = `0x00` there is no effect.

Table 146: **Master Bridge Response to Abnormal Conditions**

Transfer Type	Abnormal Condition	Bridge Response
Read	DECERR response	MDE interrupt strobe asserted. Completion returned with Unsupported Request status.
Write	DECERR response	MDE interrupt strobe asserted.
Read	SLVERR response	MSE interrupt strobe asserted. Completion returned with Completer Abort status.
Write	SLVERR response	MSE interrupt strobe asserted.
Write	Poison bit set in request	MEP interrupt strobe asserted. Data is discarded.
Read	DECERR response	MDE interrupt strobe asserted. Completion returned with Unsupported Request status.

Table 146: **Master Bridge Response to Abnormal Conditions** *(cont'd)*

Transfer Type	Abnormal Condition	Bridge Response
Write	DECERR response	MDE interrupt strobe asserted.

Link Down Behavior

The normal operation of the AXI Bridge subsystem is dependent on the integrated block for PCIe establishing and maintaining the point-to-point link with an external device for PCIe. If the link has been lost, it must be re-established to return to normal operation.

When a Hot Reset is received by the AXI Bridge subsystem, the link goes down and the PCI Configuration Space must be reconfigured.

Initiated AXI4 write transactions that have not yet completed on the AXI4 bus when the link goes down have a SLVERR response given and the write data is discarded. Initiated AXI4 read transactions that have not yet completed on the AXI4 bus when the link goes down have a SLVERR response given, with arbitrary read data returned.

Any `MemWr` TLPs for PCIe that have been received, but the associated AXI4 write transaction has not started when the link goes down, are discarded.

Endpoint

When configured to support Endpoint functionality, the AXI Bridge subsystem fully supports Endpoint operation as supported by the underlying block. There are a few details that need special consideration. The following subsections contain information and design considerations about Endpoint support.

Interrupts

Multiple interrupt modes can be configured during IP configuration, however only one interrupt mode is used at runtime. If multiple interrupt modes are enabled by the host after PCI bus enumeration at runtime, MSI-X interrupt takes precedence over Legacy interrupt. All of these interrupt modes are sent using the same `usr_irq_*` interface and the core automatically picks the best available interrupt mode at runtime.

MSI-X Interrupts

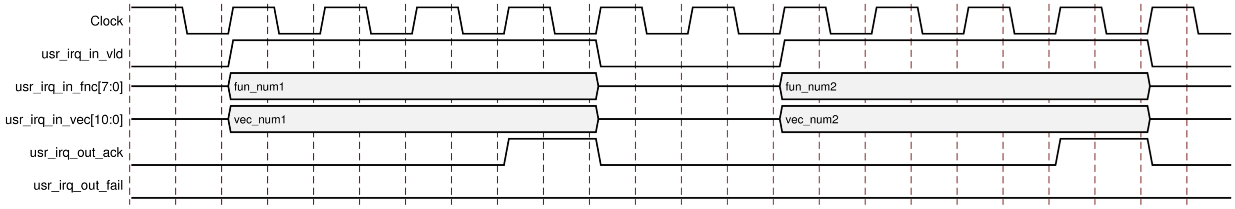
Asserting one or more bits of `usr_irq_req` causes the generation of an MSI-X interrupt if MSI-X is enabled. If both MSI-X capabilities are enabled, an MSI-X interrupt is generated. The MSI-X interrupts mode is enabled when you set the MSI-X Implementation Location option to Internal in the PCIe Misc Tab.

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 `usr_irq_ack` assertion indicates the requested interrupt has been sent on 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-X vectors. The clock signal (Clock) is `axi_clk`.For MSI-X support, there is also a vector table and PBA table.

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 53: **Interrupt**



Root Port

When configured to support Root Port functionality, the AXI Bridge subsystem fully supports Root Port operation as supported by the underlying block. There are a few details that need special consideration. The following subsections contain information and design considerations about Root Port support.

Enhanced Configuration Access Memory Map

When the subsystem is configured as a Root Port, configuration traffic is generated by using the PCI Express enhanced configuration access mechanism (ECAM). ECAM functionality is available only when the core is configured as a Root Port. Reads and writes to a certain memory aperture are translated to configuration reads and writes, as specified in the PCI Express Base Specification (v3.0), §7.2.2.

The address breakdown is defined in the following table. ECAM is used in conjunction with Bridge Register Memory Map only when used in both AXI Bridge for PCIe Gen3 core as well as DMA/Bridge Subsystem for PCIe in AXI Bridge mode core. DMA/Bridge Subsystem for PCIe Register Memory Map does not have ECAM functionality.

When an ECAM access is attempted to the primary bus number, which defaults as bus 0 from reset, then access to the type 1 PCI Configuration Header of the integrated block in the Enhanced Interface for PCIe is performed. When an ECAM access is attempted to the secondary bus number, then type 0 configuration transactions are generated. When an ECAM access is attempted to a bus number that is in the range defined by the secondary bus number and subordinate bus number (not including the secondary bus number), then type 1 configuration transactions are generated. The primary, secondary, and subordinate bus numbers are written by Root Port software to the type 1 PCI Configuration Header of the Enhanced Interface for PCIe in the beginning of the enumeration procedure.

When an ECAM access is attempted to a bus number that is out of the bus_number and subordinate bus number range, the bridge does not generate a configuration request and signal `SLVERR` response on the AXI4-Lite bus. When the Bridge is configured for EP (`PL_UPSTREAM_FACING = TRUE`), the underlying Integrated Block configuration space and the core memory map are available at the beginning of the memory space. The memory space looks like a simple PCI Express® configuration space. When the Bridge is configured for RC (`PL_UPSTREAM_FACING = FALSE`), the same is true, but it also looks like an ECAM access to primary bus, Device 0, Function 0.

When the subsystem is configured as a Root Port, the reads and writes of the local ECAM are Bus 0. Because the ACAP only has a single Integrated Block for PCIe core, all local ECAM operations to Bus 0 return the ECAM data for Device 0, Function 0.

Configuration write accesses across the PCI Express bus are non-posted writes and block the AXI4-Lite interface while they are in progress. Because of this, system software is not able to service an interrupt if one were to occur. However, interrupts due to abnormal terminations of configuration transactions can generate interrupts. ECAM read transactions block subsequent Requester read TLPs until the configuration read completions packet is returned to allow unique identification of the completion packet.

Table 147: **ECAM Addressing**

Bits	Name	Description
1:0	Byte Address	Ignored for this implementation. The s_axi_ctl_wstrb[3:0] signals define byte enables for ECAM accesses.
7:2	Register Number	Register within the configuration space to access.
11:8	Extended Register Number	Along with Register Number, allows access to PCI Express Extended Configuration Space.
14:12	Function Number	Function Number to completer.
19:15	Device Number	Device Number to completer.
27:20	Bus Number	Bus Number to completer.

Power Limit Message TLP

The AXI Bridge subsystem automatically sends a Power Limit Message TLP when the Master Enable bit of the Command Register is set. The software must set the Requester ID register before setting the Master Enable bit to ensure that the desired Requester ID is used in the Message TLP.

Root Port Configuration Read

When an ECAM access is performed to the primary bus number, self-configuration of the integrated block for PCIe is performed. A PCIe configuration transaction is not performed and is not presented on the link. When an ECAM access is performed to the bus number that is equal to the secondary bus value in the Enhanced PCIe Type 1 configuration header, then Type 0 configuration transactions are generated.

When an ECAM access is attempted to a bus number that is in the range defined by the secondary bus number and subordinate bus number range (not including secondary bus number), then Type 1 configuration transactions are generated. The primary, secondary and subordinate bus numbers are written and updated by Root Port software to the Type 1 PCI Configuration Header of the AXI Bridge subsystem in the enumeration procedure.

When an ECAM access is attempted to a bus number that is out of the range defined by the secondary bus_number and subordinate bus number, the bridge does not generate a configuration request and signal a SLVERR response on the AXI4-Lite bus.

When a Unsupported Request (UR) response is received for a configuration read request, all ones are returned on the AXI4-Lite bus to signify that a device does not exist at the requested device address. It is the responsibility of the software to ensure configuration write requests are not performed to device addresses that do not exist. However, the AXI Bridge subsystem asserts SLVERR response on the AXI4-Lite bus when a configuration write request is performed on device addresses that do not exist or a UR response is received.

Root Port BAR

Root Port BAR does not support packet filtering (all TLPs received from PCIe link are forwarded to the user logic), however Address Translation can be configured to enable or disable, depending on the IP configuration.

During core customization in the Vivado® Design Suite, when there is no BAR enabled, RP passes all received packets to the user application without address translation or address filtering.

When BAR is enabled, by default the BAR address starts at `0x0000_0000` unless programmed separately. Any packet received from the PCIe® link that hits a BAR is translated according to the PCIE-to-AXI Address Translation rules.

Note: The IP must not receive any TLPs outside of the PCIe BAR range from the PCIe link when RP BAR is enabled. If this rule cannot be enforced, it's recommended that the PCIe BAR is disabled and do address filtering and/or translation outside of the IP.

The Root Port BAR customization options in the Vivado Design Suite are found in the PCIe BARs Tab.

AXI BAR in Root Port Mode

In Root Port mode, if the address translation is needed you must program the BDF table.

Related Information

BAR and Address Translation

Configuration Transaction Timeout

Configuration transactions are non-posted transactions. The AXI Bridge subsystem has a timer for timeout termination of configuration transactions that have not completed on the PCIe link. `SLVERR` is returned when a configuration timeout occurs. Timeouts of configuration transactions are flagged by an interrupt as well.

Abnormal Configuration Transaction Termination Responses

Responses on AXI4-Lite to abnormal terminations to configuration transactions are shown in the following table.

Table 148: **Responses of Bridge to Abnormal Configuration Terminations**

Transfer Type	Abnormal Condition	Bridge Response
Config Read or Write	Bus number not in the range of primary bus number through subordinate bus number.	SLVERR response is asserted.
Config Read or Write	Valid bus number and completion timeout occurs.	SLVERR response is asserted.
Config Read or Write	Completion timeout.	SLVERR response is asserted.
Config Write	Bus number in the range of secondary bus number through subordinate bus number and UR is returned.	SLVERR response is asserted.

Receiving Interrupts

MSI-X Interrupt

All MSI-X interrupts must be decoded by the user application externally to the IP. To do this, set all of their Endpoints to use an MSI-X address that falls outside of the range of the 4Kb window from the base address programmed in the Root Port MSI Base Register 1 and Root Port MSI Base Register 2. All MSI-X interrupts will be forwarded to the M_AXI(B) interface.

All TLPs forwarded to M_AXI(B) interface are subject to the PCIe-to-AXI Address translation.

MSI Interrupt

The IP decodes the MSI interrupt based on the value programmed in Root Port MSI Base Register 1 and Root Port MSI Base Register 2. Any Memory Write TLPs received from the link with an address that falls within the 4 Kb window from the base address programmed in those registers will be treated as MSI interrupt, and will not be forwarded to the M_AXI(B) interface.

***Note*:** MSI Message Data [5:0] will always be decoded as MSI Message vector regardless of how many vectors are enabled at your Endpoint.

When an MSI interrupt is received, the MSI Decode 31-0 register or the MSI Decode 63-32 register is set. If the MSI Decode 31-0 register or the MSI Decode 63-32 register is also set, the `interrupt_out_msi_vec*` pins are asserted. `interrupt_out_msi_vec0to31` corresponds to MSI vector 0 - 31, and `interrupt_out_msi_vec32to63` corresponds to MSI vector 32 - 63. After receiving this interrupt, the user application must follow this procedure to service the interrupt:

1. Optional: Write 0 to the MSI Decode 31-0 register or the MSI Decode 63-32 register to deassert the `interrupt_out_msi_vec*` pins while the interrupt is being serviced.
2. Read the MSI Decode 31-0 register or the MSI Decode 63-32 register to check which interrupt vector is asserted.
3. Write 1 to the MSI Decode 31-0 register or the MSI Decode 63-32 register to clear the MSI interrupt bit.
4. If step 1 was executed, write 1 to the MSI Decode 31-0 register or the MSI Decode 63-32 register bit to re-enable the `interrupt_out_msi_vec*` pins for future MSI interrupts.

Legacy INTx Interrupt

When the IP has received an INTx interrupt, the Interrupt Decode 2 register is set. If the Interrupt Mask 2 register is also set, the `interrupt_out` pin is asserted. After receiving this interrupt, the user application must follow this procedure to service the interrupt:

1. Optional: Write 0 to the Interrupt Mask 2 register to deassert an interrupt line while the interrupt is being serviced.
2. Read the Interrupt Decode 2 register to check which interrupt line is currently asserted.

3. Repeat step 2 until all interrupt lines are deasserted. The interrupt line is automatically cleared when the IP receives the INTx deassert message corresponding to that interrupt line.
4. If step 1 was executed, write 1 to the Interrupt Mask 2 register to re-enable an interrupt line for future INTx interrupt.

AXI Bridge Port Descriptions

***Note*:** The Versal ACAP DMA and Bridge Subsystem for PCIe IP is implemented in a modular IP architecture. This means that GTs, PCIe IP, and the subsystem IP are implemented separately. The interface signals between GTs and PCIe IP going to a subsystem IP are not listed in this guide. These interface signals are found in *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343). The signals below apply to the subsystem only.

The interface signals for the subsystem are described in the following tables.

Global Signals

The interface signals for the Bridge are described in the following table.

Table 149: **Global Signals**

Signal Name	I/O	Description
axi_aresetn	O	In Endpoint configuration, reset signal for the S_AXIB and M_AXIB interfaces. axi_aresetn deasserts after a function has transitioned into D0_active power state (configured and enabled). In Root Port configuration, axi_aresetn deasserts after GT transceivers are initialized (assertion of Phy Ready, independent from PCIe link status).
axi_ctl_aresetn	O	Reset signal for the S_AXIL interface. In Endpoint configuration, axi_ctl_aresetn deasserts after a function has transitioned into D0_active power state (configured and enabled). In Root Port configuration, axi_ctl_aresetn deasserts after GT transceivers are initialized (assertion of Phy Ready, independent from PCIe link status).
axi_aclk	O	PCIe derived clock output for M_AXIB, S_AXIB, and S_AXIL 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.
interrupt_out	O	Interrupt signal. It is asserted for as long as there exists at least one bit asserted in the Interrupt Decode register and is not masked in the Interrupt Mask register, and/or asserted in the Interrupt Decode 2 register and is not masked in the Interrupt Decode 2 Mask register.

Table 149: **Global Signals** *(cont'd)*

Signal Name	I/O	Description
interrupt_out_msi_vec0to31	O	Interrupt signal. It is asserted for as long as there exists at least one bit asserted in the MSI Decode 31-0 register and is not masked in the Interrupt Mask 1 register. Only available in Root Port configuration with Interrupt Decode mode.
interrupt_out_msi_vec32to63	O	Interrupt signal. It is asserted for as long as there exists at least one bit asserted in the MSI Decode 63-32 register and is not masked in the Interrupt Mask 2 register. Only available in Root Port configuration with Interrupt Decode mode.
soft_reset_n	I	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 has occurred. It is not required to be toggled during initial link up operation. 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). 0: Resets all internal Bridge engines and registers as well as asserts axi_aresetn and axi_ctl_aresetn signals while maintaining PCIe link up. 1: Normal operation.
sys_rst_n	I	Reset from the PCIe edge connector reset signal.
phy_rdy_out_sd	I	Active-High signal that indicates when Phy is ready. This signal is from the Phy block.
user_lnk_up_sd	I	Active-High identifies that the PCI Express core is linked up with a host device. This signal is from the PCIe block
user_clk_sd	I	User clock from the PCIe block. All of the QDMA blocks use this clock
user_reset_sd	I	Active-High user reset signals from PCIe block.

AXI Slave Interface

Table 150: **AXI Slave Interface Signals**

Signal Name	I/O	Description
s_axib_awid[c_s_axi_id_width-1:0]	I	Slave write address ID
s_axib_awaddr[axi_addr_width-1:0]	I	Slave write address
s_axib_awregion[3:0]	I	Slave write region decode
s_axib_awlen[7:0]	I	Slave write burst length
s_axib_awsize[2:0]	I	Slave write burst size
s_axib_awburst[1:0]	I	Slave write burst type
s_axib_awvalid	I	Slave address write valid
s_axib_awready	O	Slave address write ready
s_axib_wdata[axi_data_width-1:0]	I	Slave write data

Table 150: **AXI Slave Interface Signals** *(cont'd)*

Signal Name	I/O	Description
s_axib_wstrb[axi_data_width/8-1:0]	I	Slave write strobe
s_axib_wlast	I	Slave write last
s_axib_wvalid	I	Slave write valid
s_axib_wready	O	Slave write ready
s_axib_bid[c_s_axi_id_width-1:0]	O	Slave response ID
s_axib_bresp[1:0]	O	Slave write response
s_axib_bvalid	O	Slave write response valid
s_axib_bready	I	Slave response ready
s_axib_arid[c_s_axi_id_width-1:0]	I	Slave read address ID
s_axib_araddr[axi_addr_width-1:0]	I	Slave read address
s_axib_arregion[3:0]	I	Slave read region decode
s_axib_arlen[7:0]	I	Slave read burst length
s_axib_arsize[2:0]	I	Slave read burst size
s_axib_arburst[1:0]	I	Slave read burst type
s_axib_arvalid	I	Slave read address valid
s_axib_arready	O	Slave read address ready
s_axib_rid[c_s_axi_id_width-1:0]	O	Slave read ID tag
s_axib_rdata[axi_data_width-1:0]	O	Slave read data
s_axib_rresp[1:0]	O	Slave read response
s_axib_rlast	O	Slave read last
s_axib_rvalid	O	Slave read valid
s_axib_rready	I	Slave read ready

AXI4-Lite Slave CSR Interface

Table 151: **AXI4-Lite Slave CSR Signals**

Signal Name	I/O	Description
s_axil_csr_awaddr[31:0]	I	This signal is the address for a memory mapped write to the DMA from the user logic. s_axil_csr_awaddr[15]: 1'b1 – QDMA CSR register 1'b0 – Bridge register
s_axil_csr_awvalid	I	The assertion of this signal means there is a valid write request to the address on s_axil_csr_awaddr.
s_axil_csr_awprot[2:0]	I	Protection type.(unused)
s_axil_csr_awready	O	Slave write address ready.
s_axil_csr_wdata[31:0]	I	Slave write data.
s_axil_csr_wstrb[3:0]	I	Slave write strobe.
s_axil_csr_wvalid	I	Slave write valid.

Table 151: **AXI4-Lite Slave CSR Signals** *(cont'd)*

Signal Name	I/O	Description
s_axil_csr_wready	O	Slave write ready.
s_axil_csr_bvalid	O	Slave write response valid.
s_axil_csr_bresp[1:0]	O	Slave write response.
s_axil_csr_bready	I	Save response ready.

Interrupt Signals

The interface signals for the Bridge are described in the following table.

Table 152: **Interrupt Signals**

Signal Name	I/O	Description
usr_irq_in_vld	I	Valid An assertion indicates that an interrupt associated with the vector, function, and pending fields on the bus should be generated to PCIe. Once asserted, Usr_irq_in_vld must remain high until usr_irq_out_ack is asserted by the DMA.
usr_irq_in_vec [10:0]	I	Vector The MSIX vector to be sent.
usr_irq_in_fnc [7:0]	I	Function The function of the vector to be sent.
usr_irq_out_ack	O	Interrupt Acknowledge An assertion of the acknowledge bit indicates that the interrupt was transmitted on the link the user logic must wait for this pulse before signaling another interrupt condition.
usr_irq_out_fail	O	Interrupt Fail An assertion of fail indicates that the interrupt request was aborted before transmission on the link.

Register Space

Bridge register addresses start at 0xE00. Addresses from 0x00 to 0xE00 are directed to the PCIe Core configuration register space.

QDMA Bridge register descriptions are found in `qdma_v4_0_bridge_registers.csv` available in the Register Reference File.

AXI4-Lite Slave CSR Register Space

The Bridge register space and DMA register space are accessible through the AXI4-Lite Slave CSR interface. This interface is accessible only when `csr_prog_done` port is set to 1. You must wait until `csr_prog_done` port it set.

Table 153: **AXI4-Lite Slave CSR Register Space**

Register Space	AXI4-Lite Slave CSR Interface	Details
Bridge registers	AXI4-Lite Slave CSR Address **bit [15]** is set to 0	Found in `qdma_v4_0_bridge_registers.csv` available in the Register Reference File.
DMA registers	AXI4-Lite Slave CSR Address **bit [15]** is set to 1	Described in QDMA PF Address Register Space and QDMA VF Address Register Space. ***Note*:** Through this interface, only the DMA CSR register can be accessed. The DMA Queue space register can only be accessed through AXI4-Lite Slave.

Bridge Register Space

Bridge register addresses start at 0xE00. Addresses from 0x00 to 0xE00 are directed to the PCIe Core configuration register space.

QDMA Bridge register descriptions are found in `qdma_v4_0_bridge_registers.csv` available in the Register Reference File.

DMA Register Space

The DMA register space is described in the following sections:

- QDMA PF Address Register Space
- QDMA VF Address Register Space

AXI4-Lite Slave Register Space

DMA queue space registers can be accessed through the AXI4-Lite Slave interface.

QDMA Queue space PF register addresses and QDMA Queue space VF register addresses are described in QDMA_TRQ_SEL_QUEUE_PF (0x18000) and QDMA_TRQ_SEL_QUEUE_VF (0x3000).

***Note*:** Through this interface, only the DMA Queue space registers can be accessed. DMA CSR register can be accessed only through AXI4-Lite Slave CSR interface.

Design Flow Steps

This section describes customizing and generating the subsystem, constraining the subsystem, and the simulation, synthesis, and implementation steps that are specific to this IP subsystem. 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 subsystem in the Vivado® Design Suite.

If you are customizing and generating the subsystem 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 subsystem using the following steps:

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

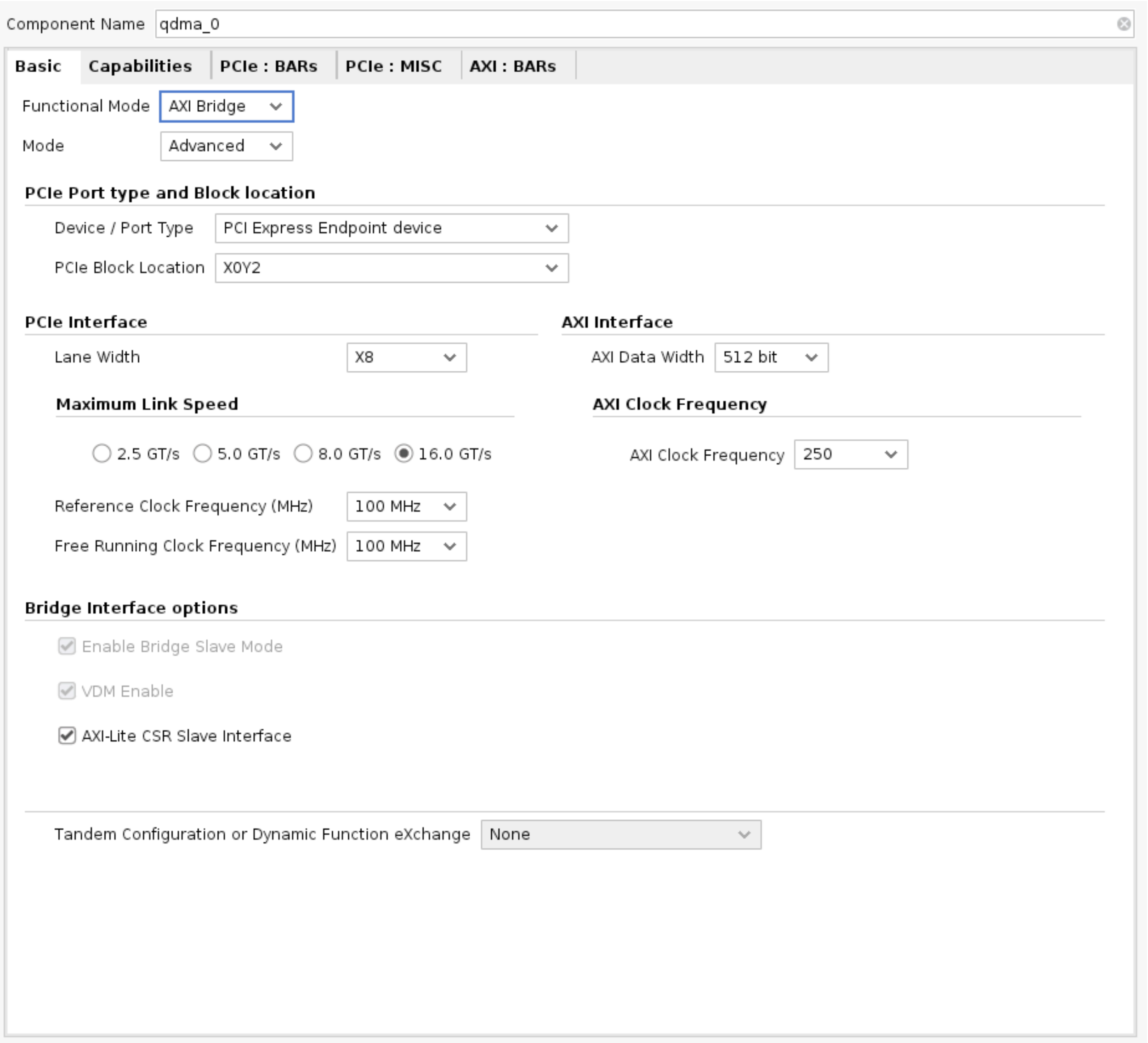
For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) and the *Vivado Design Suite User Guide: Getting Started* (UG910).

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

Basic Tab

The Basic tab options for the AXI Bridge mode (Functional Mode option) are shown in the following figure.

Figure 54: **Basic Tab**



Component Name: qdma_0

Functional Mode: AXI Bridge

Mode: Advanced

PCIe Port type and Block location

Device / Port Type: PCI Express Endpoint device

PCIe Block Location: X0Y2

PCIe Interface

Lane Width: X8

Maximum Link Speed

○ 2.5 GT/s ○ 5.0 GT/s ○ 8.0 GT/s ● 16.0 GT/s

Reference Clock Frequency (MHz): 100 MHz

Free Running Clock Frequency (MHz): 100 MHz

AXI Interface

AXI Data Width: 512 bit

AXI Clock Frequency

AXI Clock Frequency: 250

Bridge Interface options

☑ Enable Bridge Slave Mode

☑ VDM Enable

☑ AXI-Lite CSR Slave Interface

Tandem Configuration or Dynamic Function eXchange: None

The options are defined as follows:

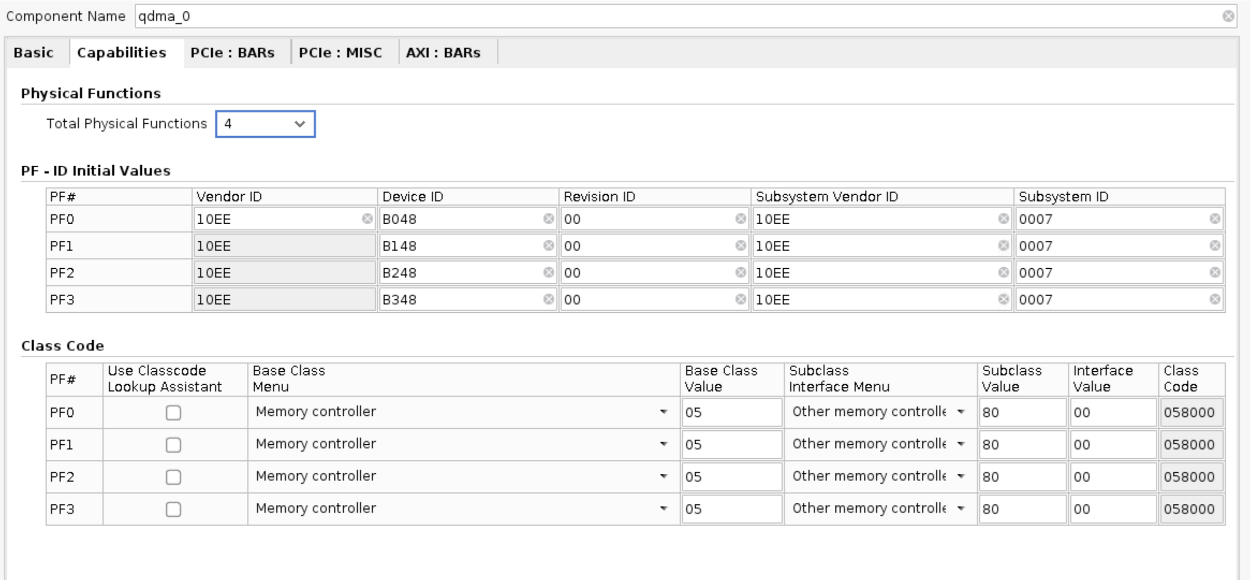
- **Functional Mode:** For this subsystem, select **AXI Bridge** option.
- **Mode:** Allows you to select the Basic or Advanced mode of the configuration of core.
- **Device /Port Type:** Option to select between PCI Express® Endpoint device and PCI Express® Root Complex mode.

- **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. The *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343) defines the available widths and associated generated core. Wider lane width cores can train down to smaller lane widths if attached to a smaller lane-width device. Options are 4, 8, or 16 lanes.
- **Maximum Link Speed:** The core allows you to select the Maximum Link Speed supported by the device. The*Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343) defines the lane widths and link speeds supported by the device. Higher link speed cores are capable of training to a lower link speed if connected to a lower link speed capable device. The default option is Gen3.
- **Reference Clock Frequency:** The default is 100 MHz.
- **Reset Source:**
  - **PCIe User Reset:** The user reset comes from PCIe core after the link is established. When the PCIe link goes down, the user reset is asserted and the core goes to reset mode. And when the link comes back up, the user reset is deasserted.
- **AXI Data Width:** Select 128, 256 bit, or 512 bit. The core allows you to select the Interface Width, as defined in the *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343). The default interface width set in the Customize IP dialog box is the lowest possible interface width.
- **AXI Clock Frequency:** 250 MHz depending on the lane width/speed.
- **Enable Bridge Slave Mode:** Selected by default.
- **VDM Enable:** Selected by default.
- **AXI Lite Slave Interface:** Select to enable the AXI4-Lite slave interface, which can access DMA queue space.
- **AXI Lite CSR Slave Interface:** Selected by default. AXI4-Lite CSR slave interface, which can access DMA Configuration Space Register or Bridge registers.

Capabilities Tab

The Capabilities Tab is shown in the following figure.

Figure 55: **Capabilities Tab**



Component Name qdma_0

Physical Functions

Total Physical Functions 4

PF - ID Initial Values

PF#	Vendor ID	Device ID	Revision ID	Subsystem Vendor ID	Subsystem ID
PF0	10EE	B048	00	10EE	0007
PF1	10EE	B148	00	10EE	0007
PF2	10EE	B248	00	10EE	0007
PF3	10EE	B348	00	10EE	0007

Class Code

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

- **Total Physical Functions:** A maximum of four Physical Functions can be enabled.
- **PF - ID Initial Values:**
  - **Vendor ID:** Identifies the manufacturer of the device or application. Valid identifiers are assigned by the PCI Special Interest Group to guarantee that each identifier is unique. The default value, `10EEh`, is the Vendor ID for Xilinx. Enter a vendor identification number here. `FFFFh` is reserved.
  - **Device ID:** A unique identifier for the application; the default value, which depends on the configuration selected, is 70h. This field can be any value; change this value for the application.

    The Device ID parameter is evaluated based on:

    - The device family: B for Versal.
    - EP or RP mode
    - Link width
    - Link speed

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

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

  - **Revision ID:** Indicates the revision of the device or application; an extension of the Device ID. The default value is `00h`; enter values appropriate for the application.

  - **Subsystem Vendor ID:** Further qualifies the manufacturer of the device or application. Enter a Subsystem Vendor ID here; the default value is 10EEh. Typically, this value is the same as Vendor ID. Setting the value to 0000h can cause compliance testing issues.

  - **Subsystem ID:** Further qualifies the manufacturer of the device or application. This value is typically the same as the Device ID; the default value depends on the lane width and link speed selected. Setting the value to 0000h can cause compliance testing issues.

- **Class Code:** The Class Code identifies the general function of a device.

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

  - **Base Class:** Broadly identifies the type of function performed by the device.

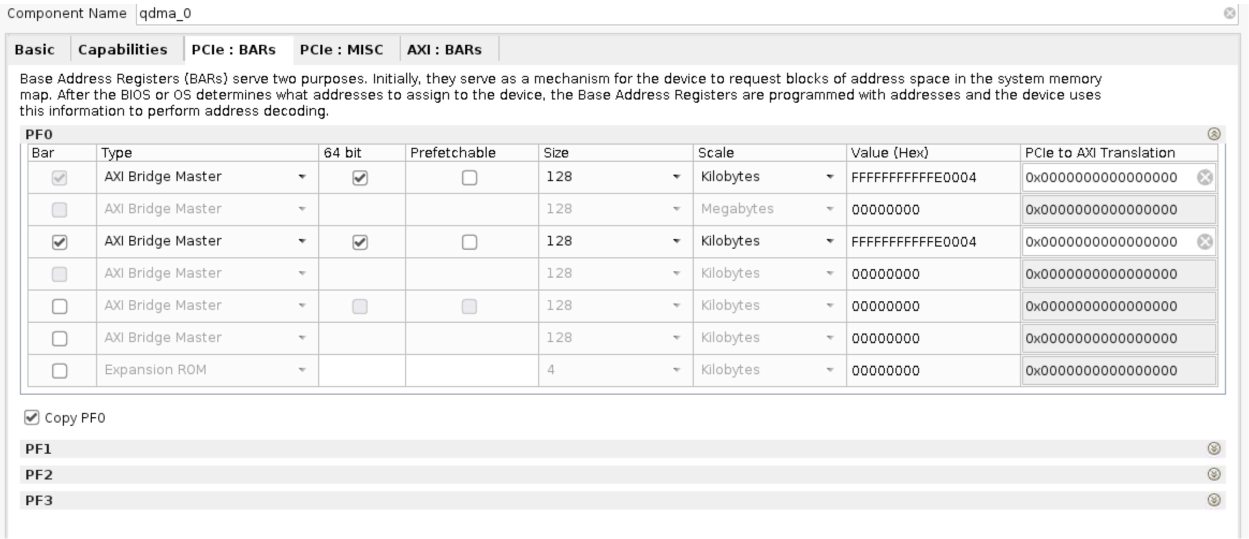
  - **Subclass:** More specifically identifies the device function.

  - **Interface:** Defines a specific register-level programming interface, if any, allowing device-independent software to interface with the device.

PCIe BARs Tab

The PCIe BARs tab options for the AXI Bridge mode (Functional Mode option) is shown in the following figure.

Figure 56: **PCIe BARs Tab**



Component Name qdma_0

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

PF0

Bar	Type	64 bit	Prefetchable	Size	Scale	Value (Hex)	PCIe to AXI Translation
☑	AXI Bridge Master	☑	□	128	Kilobytes	FFFFFFFFFFFE0004	0x0000000000000000
□	AXI Bridge Master			128	Megabytes	00000000	0x0000000000000000
☑	AXI Bridge Master	☑	□	128	Kilobytes	FFFFFFFFFFFE0004	0x0000000000000000
□	AXI Bridge Master			128	Kilobytes	00000000	0x0000000000000000
□	AXI Bridge Master	□	□	128	Kilobytes	00000000	0x0000000000000000
□	AXI Bridge Master			128	Kilobytes	00000000	0x0000000000000000
□	Expansion ROM			4	Kilobytes	00000000	0x0000000000000000

☑ Copy PF0

PF1

PF2

PF3

- **Base Address Register Overview:** In Endpoint configuration, the core supports up to six 32-bit BARs or three 64-bit BARs, and the Expansion read-only memory (ROM) BAR. BARs can be one of two sizes:
  - **32-bit BARs:** The address space can be as small as 128 B or as large as 2 GB. Used for DMA, AXI Lite Master or AXI Bridge Master.
  - **64-bit BARs:** The address space can be as small as 128 B or as large as 8 Exabytes. Used for DMA, AXI Lite Master or AXI Bridge Master.

All BAR register share these options.

> **IMPORTANT!** *The DMA requires a large amount of space to support functions and queues. By default, 64-bit BAR space is selected for the DMA BAR. This applies for PF and VF bars. You must calculate your design needs first before selecting between 64-bit and 32-bit BAR space.*

BAR selections are configurable. By default DMA is at BAR 0 (64 bit), AXI-Lite Master is at BAR 2 (64 bit). These selections can be changed according to user needs.

- **BAR:** Click the checkbox to enable the BAR. Deselect the checkbox to disable the BAR.
- **Type:** Select from **DMA** (by default in BAR0), **AXI Lite Master** (by default in BAR1, if enabled), or **AXI Bridge Master** (by default in BAR2, if enabled). All other BARs, you can select between AXI List Master and AXI Bridge Master. Expansion ROM can be enabled by selecting BAR6

  For 64-bit BAR (default selection), **DMA** (by default in BAR0), **AXI Lite Master** (by default in BAR2, if enabled), and **AXI Bridge Master** (by default in BAR4, if enabled). Expansion ROM can be enabled by selection BAR6.

  - **DMA:** DMA by default is assigned to BAR0 space and for all PFs. DMA option can be selected in any available BAR (only one BAR can have DMA option). If you select **DMA Mailbox Management** rather than DMA; however, DMA Mailbox Management will not allow you to perform any DMA operations. After selecting the DMA Mailbox Management option, the host has access to the extended Mailbox space. For details about this space, see the QDMA_PF_MAILBOX (0x22400) register space.
  - **AXI Lite Master:** Select the AXI Lite Master interface option for any BAR space. The Size, scale and address translation are configurable.
  - **AXI Bridge Master:** Select the AXI Bridge Master interface option for any BAR space. The Size, scale and address translation are configurable.
  - **Expansion ROM:** When enabled, this space is accessible on the AXI4-Lite Master. This is a read-only space. The size, scale, and address translation are configurable.
- **Size:** The available Size range depends on the 32-bit or 64-bit bar selected. The DMA requires 256 KB of space, which is the fixed default selection. Other BAR size selections are available, but must be specified.
- **Scale:** Select between Byte, Kilobytes and Megabytes.

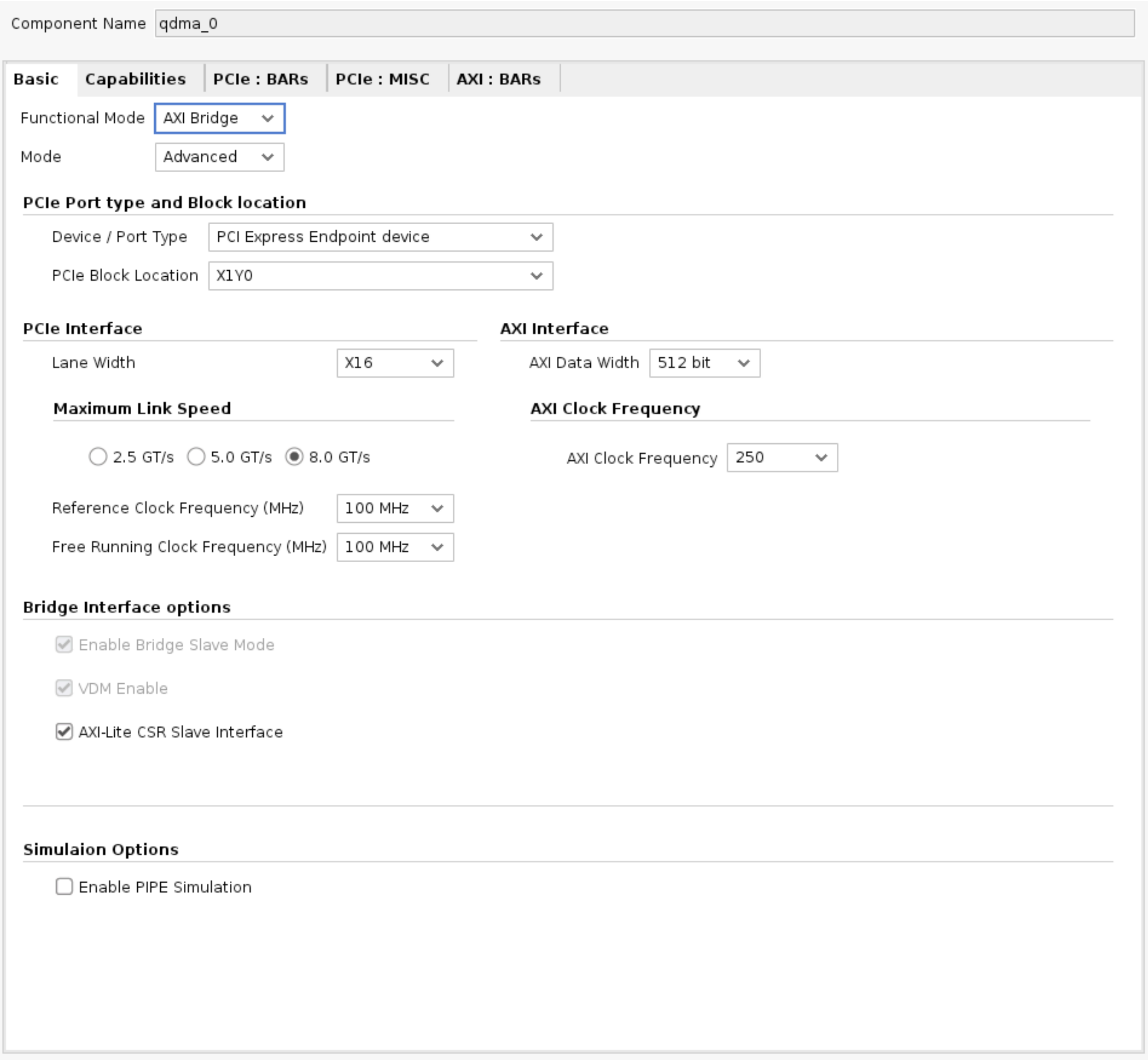
- **Value:** The value assigned to the BAR based on the current selections.
- **PCIe to AXI Translation:** Configures the translation mapping between PCI Express® and AXI address space. You should edit this parameter to fit design requirements

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

PCIe MISC Tab

The PCIe Miscellaneous tab options for the AXI Bridge subsystem are shown in the following figure.

Figure 57: **PCIe Miscellaneous Tab**



Component Name qdma_0

Functional Mode AXI Bridge

Mode Advanced

PCIe Port type and Block location

Device / Port Type PCI Express Endpoint device

PCIe Block Location X1Y0

PCIe Interface

Lane Width X16

Maximum Link Speed

○ 2.5 GT/s ○ 5.0 GT/s ● 8.0 GT/s

Reference Clock Frequency (MHz) 100 MHz

Free Running Clock Frequency (MHz) 100 MHz

AXI Interface

AXI Data Width 512 bit

AXI Clock Frequency

AXI Clock Frequency 250

Bridge Interface options

☑ Enable Bridge Slave Mode

☑ VDM Enable

☑ AXI-Lite CSR Slave Interface

Simulaion Options

□ Enable PIPE Simulation

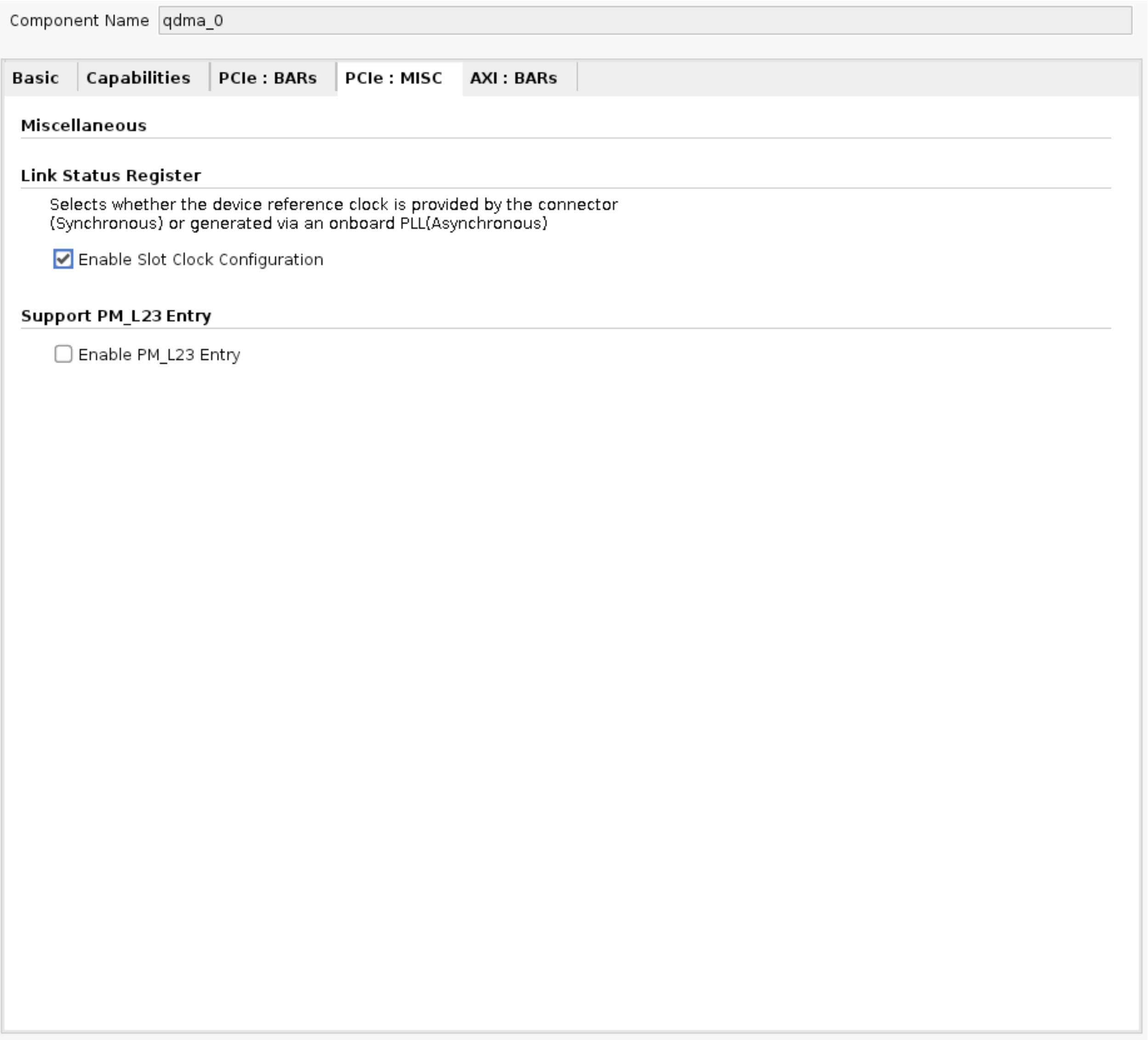
The options are defined as follows:

- **Legacy Interrupt Settings:** Select one of the Legacy Interrupts: INTA, INTB, INTC, or INTD.
- **MSI-X Capabilities:** MSI-X is enabled by default.The MSI-X settings for different physical functions can be set as required.section can be done for all select PF's.
- **MSI-X Table Settings:** Defines the MSI-X Table Structure.
    - **Table Size:** Specifies the MSI-X Table size. The default is 8 (8 interrupt vectors per function). Adding more vectors to a function is possible; contact Xilinx for support.
    - **Table Offset:** Specifies the offset from the Base address Register (BAR) in DMA configuration space used to map function in MSI-X Table onto memory space. MSI-X table space is fixed at offset 0x30000.PBA table is fixed at offset 0x34000
- **Access Control Server (ACS) Enable:** ACS is selected by default.
- **Link Status Register:** By default, Enable Slot Clock Configuration is selected. This means that the slot configuration bit is enabled in the link status register.

PCIe MISC Tab in Root Port Mode

The PCIe Miscellaneous tab options for the AXI Bridge subsystem are shown in the following figure.

Figure 58: **PCIe Miscellaneous Tab**



Component Name: qdma_0

Miscellaneous

Link Status Register

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

☑ Enable Slot Clock Configuration

Support PM_L23 Entry

□ Enable PM_L23 Entry

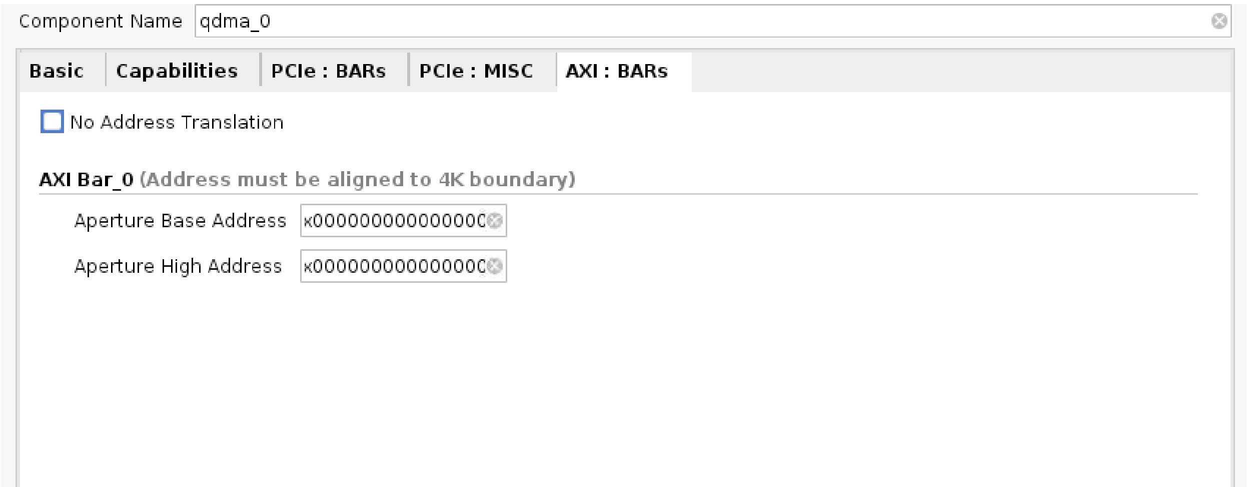
The options are defined as follows:

- **Link Status Register:** By default, Enable Slot Clock Configuration is selected. This means that the slot configuration bit is enabled in the link status register.
- **Enable PM_L23 Entry:** .

AXI BARs Tab

The AXI BARs tab options for the AXI Bridge mode (Functional Mode option) are shown in the following figure.

Figure 59: **AXI BARs Tab**



Component Name qdma_0

□ No Address Translation

AXI Bar_0 (Address must be aligned to 4K boundary)

Aperture Base Address x00000000000000C

Aperture High Address x00000000000000C

The options are defined as follows:

- **No Address Translation:** When this option is selected user will get a one 64-bit BAR with out any translation. User is responsible for setting up address translation.When this option is *not* selected DMA will setup a address transitional based the entries that is selected in below options. And DMA will have fixed 8 window for the BAR selected below. Descriptions for how to set each of the 8 window is explained in Salve Bridge section.
- **Aperture Base Address:** Sets the base address for the address range of BAR. You should edit this parameter to fit design requirements. When adding the IP in a block design, this option is not available. Instead, set this option in Address Editor tab. When you add the IP to a block design, this option is not available. In that case, you must set this option in the Address Editor tab.
- **Aperture High Address:** Sets the upper address threshold for the address range of BAR. You should edit this parameter to fit design requirements. When you add the IP to a block design, this option is not available. In that case, you must set this option in the Address Editor tab.

  ***Note*:** In addition, you must also program the BDF table.

Related Information

BAR and Address Translation

Output Generation

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

Example Design

This chapter contains information about the example design provided with the generation of the IP in the Vivado® Design Suite.

> **TIP:** *The PCIe reset pin for PL PCIE designs can be connected to any compatible single ended PL I/O pin location. If your board is compatible for either CPM4 or PL PCIE usage, you can use the CPM4 pin MIO38 to route the* `sys_rst_n`*. When this is done, the PL PCIE can use the reset as routed to the PL.*

Before opening the example design, set the following Tcl property to use the reset on the MIO38 pin:

```
set_property CONFIG.insert_cips {true} [get_ips pcie_versal_0]
```

Endpoint Configuration

The example simulation design for the Endpoint configuration of the AXI PCIe® block consists of two parts.

- Root Port Model: a test bench that generates, consumes, and checks PCI Express® bus traffic
- AXI Block RAM Controller

Xilinx AXI Verification IP Attached to the AXI Slave Interface

AXI Verification IP allows you to initiate AXI transfer from an Endpoint configured bridge or generates a larger more complex AXI transfer from a Root Port configured bridge. For more details, see the *AXI Verification IP LogiCORE IP Product Guide* (PG267).

To enable AXI Verification IP example design option:

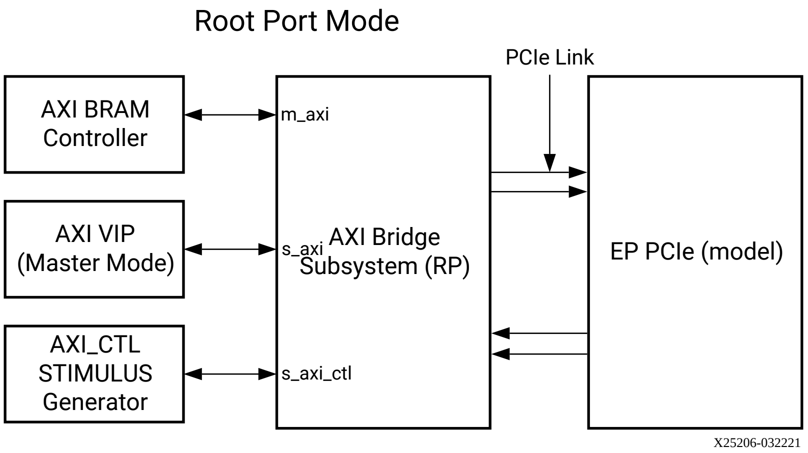
1. Add the IP to the design.
2. Run the following Tcl command in Vivado® console:

   ```
   set_property CONFIG.axi_vip_in_exdes true [get_ips <ip_name>]
   ```

3. Open the IP Example Design.

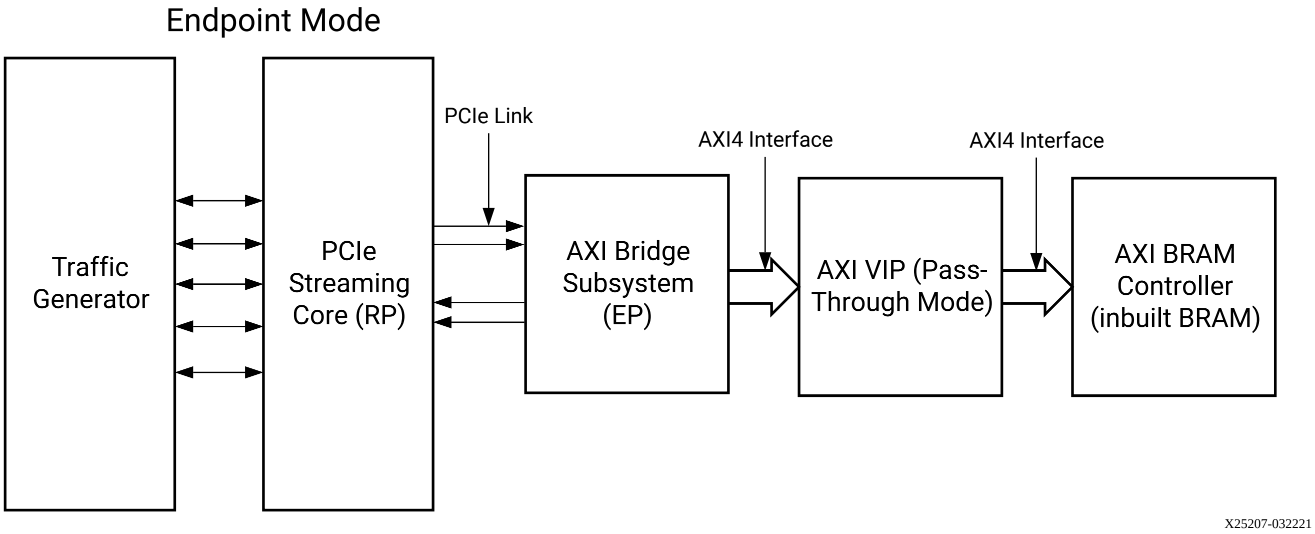
The following figure shows the IP in Root Port configuration.

Figure 60: **Root Port Configuration**



The following figure shows the AXI Bridge Subsystem for PCIe IP in Endpoint configuration.

Figure 61: **Endpoint Configuration**



Customizing and Generating the Example Design

In the Customize IP dialog box, use the default core parameter values for the IP example design. In particular:

- In the PCIE:Basics tab, the example design supports only an Endpoint (EP) device.
- In the AXI:BARS tab, the Base Address, High Address, and AXI to PCIe® Translation default values are used.

After reviewing the core parameters:

1. Right-click the component name.
2. Select **Open IP Example Design**.

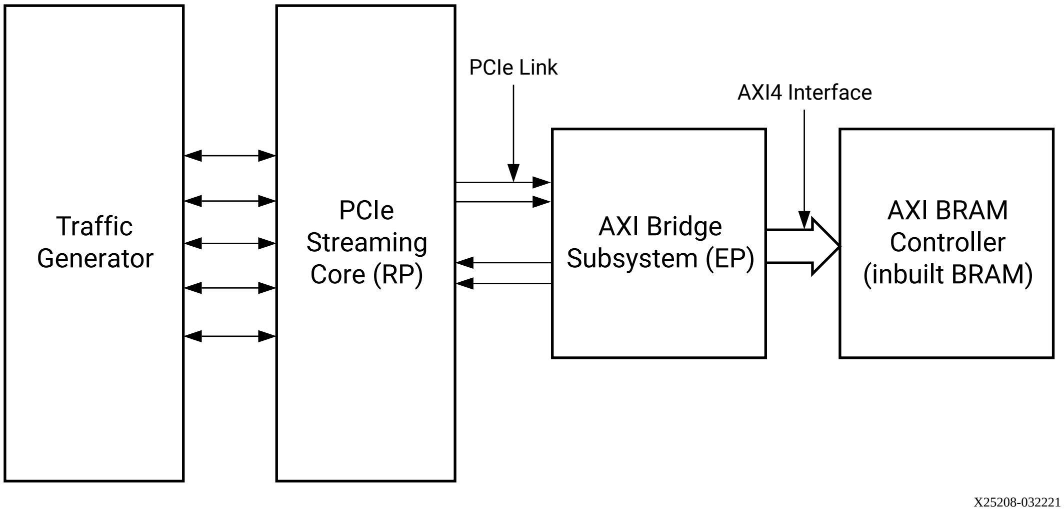
This opens a separate example design.

Simulation Design Overview

For the simulation design, transactions are sent from the Root Port Model to the Bridge core configured as an Endpoint and processed inside the AXI Block RAM controller design.

The following figure illustrates the simulation design provided with the AXI Bridge subsystem.

Figure 62: **Example Design Block Diagram**



The example design supports Verilog as the target language.

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).

Implementation Design Overview

For implementation design, the AXI Block RAM controller can be used as a scratch pad memory to write and read to Block RAM locations.

Example Design Elements

The core wrapper includes:

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

Debugging

This appendix provides information for using the resources available on the Xilinx® Support website, debug tools, and other step-by-step processes for debugging designs that use the AXI Bridge for PCIe core.

Finding Help on Xilinx.com

To help in the design and debug process when using the AXI Bridge subsystem, the Xilinx Support web page contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support.

Documentation

This product guide is the main document associated with the 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, see the online help after installation.

Solution Centers

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

The PCI Express® Solution Center is located at Xilinx Solution Center for PCI Express. Extensive debugging collateral is available in AR: 56802.

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 are listed below, and can be located by using the Search Support box on the main Xilinx support web page. To maximize your search results, use proper keywords, such as:

- the product name
- tool messages
- summary of the issue encountered

Master Answer Record for the Subsystem

AR 75397.

AXI Bridge Debugging Answer Record

AR 000033516.

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 Bridge design issues. It is important to know which tools are useful for debugging various situations.

Third-Party Tools

This section describes third-party software tools that can be useful in debugging.

LSPCI (Linux)

LSPCI is available on Linux platforms and allows you to view the PCI Express® device configuration space. LSPCI is usually found in the `/sbin` directory. LSPCI displays a list of devices on the PCI buses in the system. See the LSPCI manual for all command options. Some useful commands for debugging include:

- `lspci -x -d [<vendor>]:[<device>]`

  This displays the first 64 bytes of configuration space in hexadecimal form for the device with vendor and device ID specified (omit the -d option to display information for all devices). The default Vendor/Device ID for Xilinx cores is 10EE:6012. Here is a sample of a read of the configuration space of a Xilinx device:

  ```
  > lspci -x -d 10EE:6012
  81:00.0 Memory controller: Xilinx Corporation: Unknown device 6012
  00: ee 10 12 60 07 00 10 00 00 00 80 05 10 00 00 00
  10: 00 00 80 fa 00 00 00 00 00 00 00 00 00 00 00 00
  20: 00 00 00 00 00 00 00 00 00 00 00 00 ee 10 6f 50
  30: 00 00 00 00 40 00 00 00 00 00 00 00 05 01 00 00
  ```

  Included in this section of the configuration space are the Device ID, Vendor ID, Class Code, Status and Command, and Base Address Registers.

- `lspci -xxxx -d [<vendor>]:[<device>]`

  This displays the extended configuration space of the device. It can be useful to read the extended configuration space on the root and look for the Advanced Error Reporting (AER) registers. These registers provide more information on why the device has flagged an error (for example, it might show that a correctable error was issued because of a replay timer timeout).

- `lspci -k`

  Shows kernel drivers handling each device and kernel modules capable of handling it (works with kernel 2.6 or later).

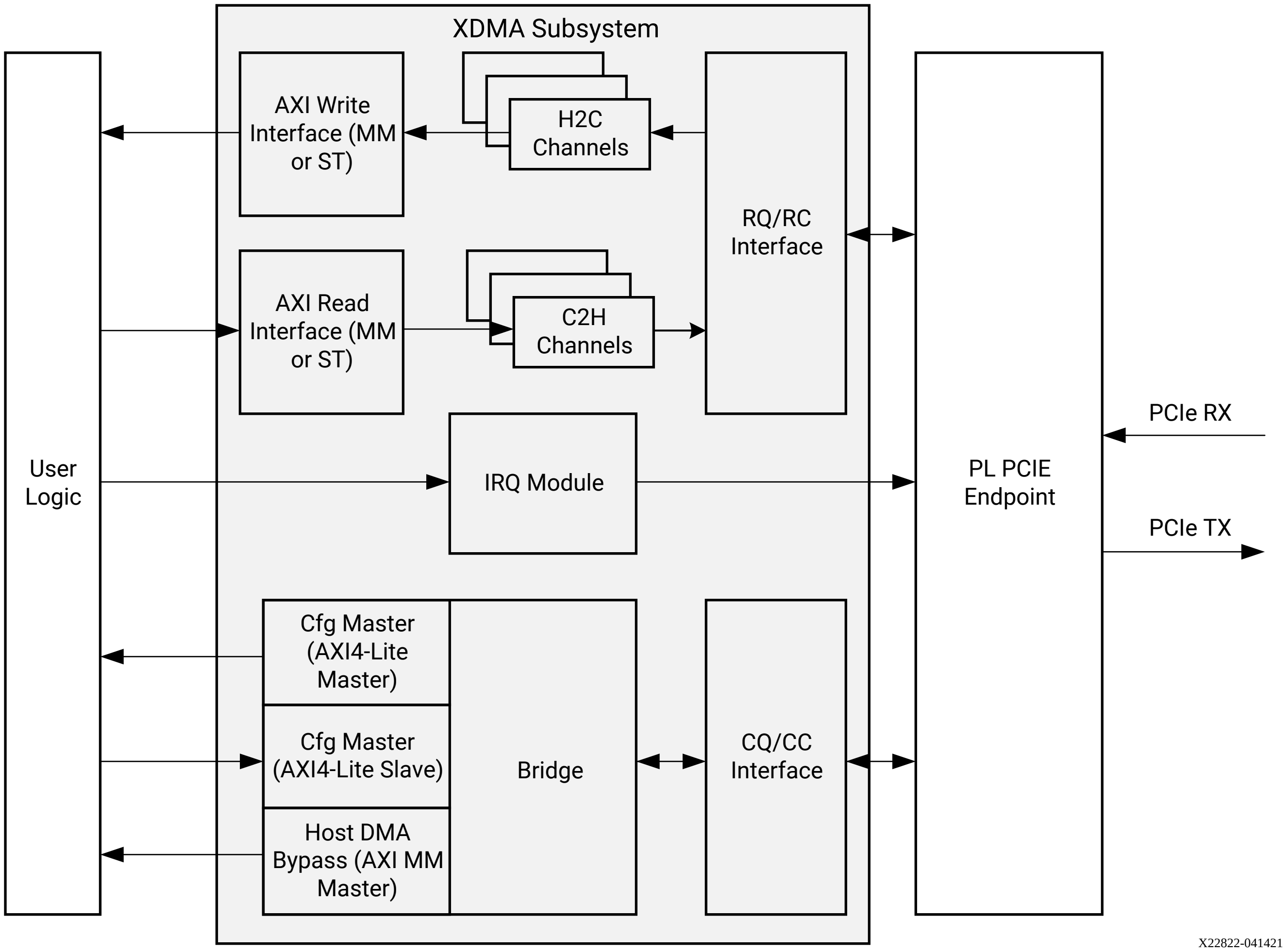
XDMA Subsystem for PL PCIE4

Overview

The XDMA Subsystem can be configured as a high performance direct memory access (DMA) data mover between the PCI Express® and AXI memory spaces. As a DMA, the subsystem 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 Versal® ACAP for applications that demand the highest performance and lowest latency.

IMPORTANT! *This feature is not available in PL PCIE5.*

Figure 63: **XDMA Subsystem Overview**


XDMA Subsystem
User Logic
AXI Write Interface (MM or ST)
H2C Channels
AXI Read Interface (MM or ST)
C2H Channels
RQ/RC Interface
IRQ Module
PL PCIE Endpoint
PCIe RX
PCIe TX
Cfg Master (AXI4-Lite Master)
Cfg Master (AXI4-Lite Slave)
Host DMA Bypass (AXI MM Master)
Bridge
CQ/CC Interface
X22822-041421

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 *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343).

Modular IP Architecture

The XDMA Subsystem is packaged in what is called a **modular IP architecture**. Modular IP architecture refers to the programmable logic integrated block for PCIe (PL PCIE) IP and the XDMA subsystem appearing as two separate IP that are connected in the Vivado IP integrator.

To generate the IP:

1. In the Vivado IP catalog, locate the **XDMA Subsystem** and add it to your design.
2. Configure the subsystem as required

3. In the Vivado IP integrator, click **Run Block Automation**. This stitches the PL PCIE and XDMA together.

After these steps, a fully integrated PL PCIE and XDMA is available, and you can add other IP to your design as needed.

Related Information

Customizing and Generating the Subsystem

Feature Summary

The XDMA Subsystem 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 XDMA 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

XDMA Subsystem Limitations

PCIe Capability

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

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

Chapter 13

Product Specification

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

Performance and Resource Utilization

For DMA Perfomance data, see AR 68049.

For DMA Resource Utilization, see Resource Utilization web page.

Configurable Components of the Subsystem

Internally, the subsystem 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 XDMA Subsystem generates requests and expected completions. The AXI4-Stream interface is data-only.

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

- A 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 XDMA 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 IP Configuration GUI.

Table 154: **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 155: **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

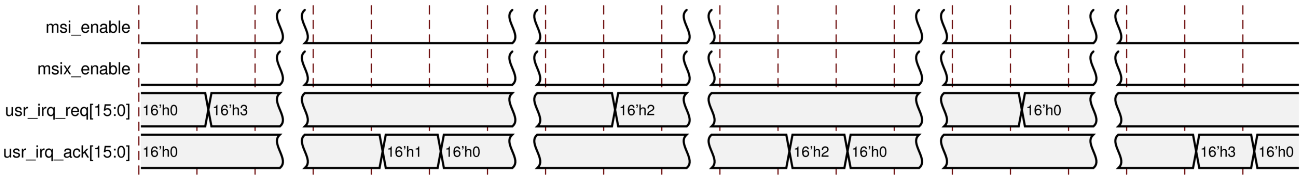
Asserting one or more bits of `usr_irq_req` when legacy interrupts are enabled causes the DMA to issue a legacy interrupt over PCIe. Multiple bits may be asserted simultaneously but each bit must remain asserted until the corresponding `usr_irq_ack` bit has been asserted. After a `usr_irq_req` bit is asserted, it must remain asserted until both corresponding `usr_irq_ack` bit is asserted and the interrupt is serviced and cleared by the Host. This ensures interrupt pending register within the IP remains asserted when queried by the Host's Interrupt Service Routine (ISR) to determine the source of interrupts. The `usr_irq_ack` assertion indicates the requested interrupt has been sent to the PCIe block. 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 `usr_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 `xdma0_usr_irq_req` wire can be reasserted to generate another legacy interrupt.

The `xdma0_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 `xdma0_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`. The figure below shows one possible scenario where `usr_irq_ack`is deasserted at the same cycle for both requests[1:0], which might not be the case in other situations.

Figure 64: **Legacy Interrupts**



MSI and MSI-X Interrupts

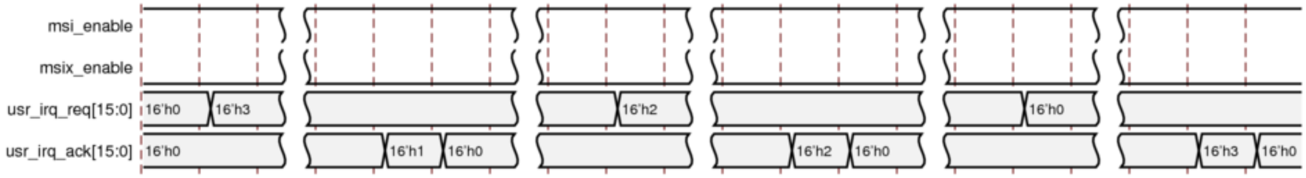
Asserting one or more bits of `usr_irq_req` causes the generation of an MSI or MSI-X interrupt if MSI or MSI-X is enabled. If both MSI and MSI-X capabilities are enabled, an MSI-X interrupt is generated.

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 `usr_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`.

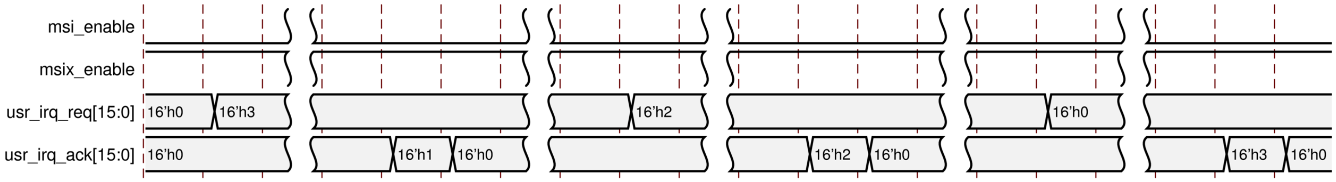
Figure 65: **MSI Interrupts**



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 66: **MSI-X Interrupts**



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 XDMA. This configuration information can be read through register reads to the appropriate register offset within the config module.

XDMA Operations

Quick Start

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

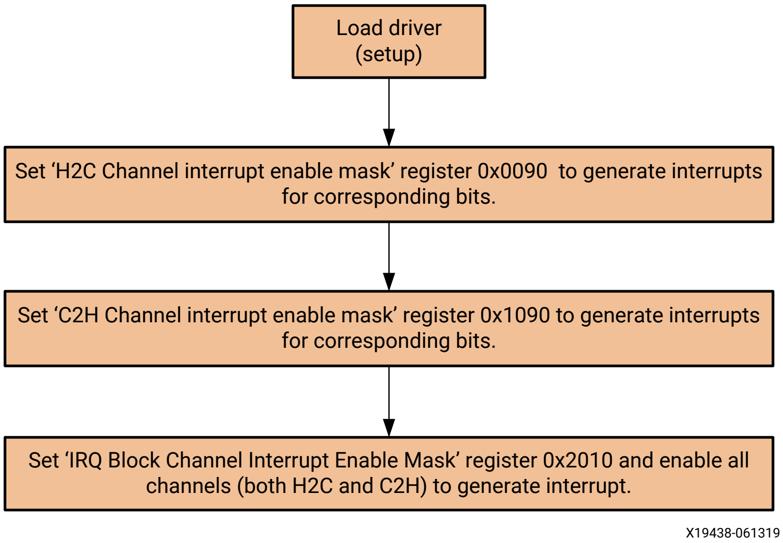
The following flow charts show typical transfers for both H2C and C2H transfers when the data interface is selected during IP configuration for an AXI Memory Mapped interface.

Note: In the following flow charts, the table numbers refer to tables found in *DMA/Bridge Subsystem for PCI Express Product Guide* (PG195). The flow charts will be updated in a future release.

Initial Setup For H2C and C2H Transfers

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

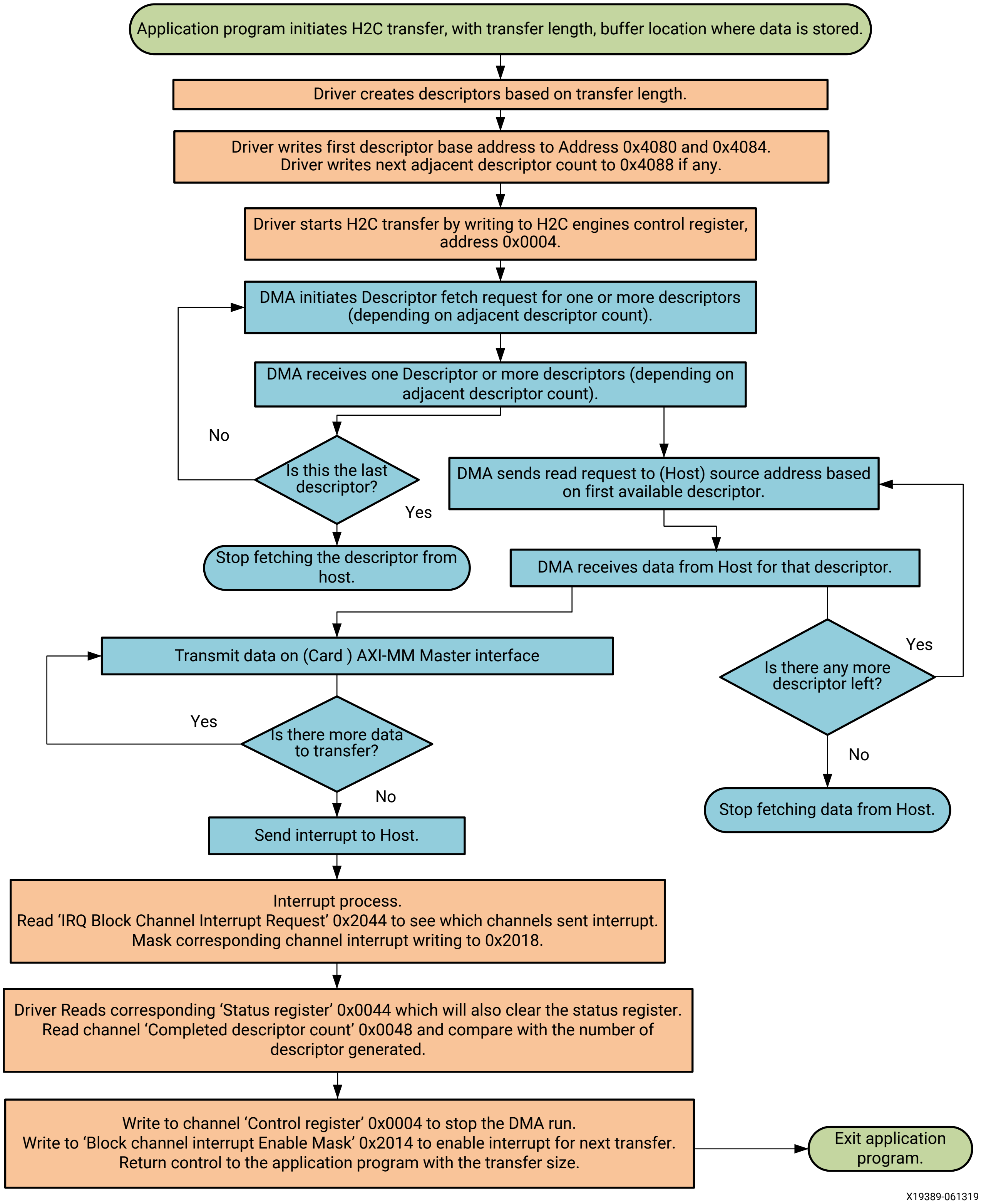
Figure 67: **Setup**


Load driver (setup)
Set 'H2C Channel interrupt enable mask' register 0x0090 to generate interrupts for corresponding bits.
Set 'C2H Channel interrupt enable mask' register 0x1090 to generate interrupts for corresponding bits.
Set 'IRQ Block Channel Interrupt Enable Mask' register 0x2010 and enable all channels (both H2C and C2H) to generate interrupt.
X19438-061319

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.

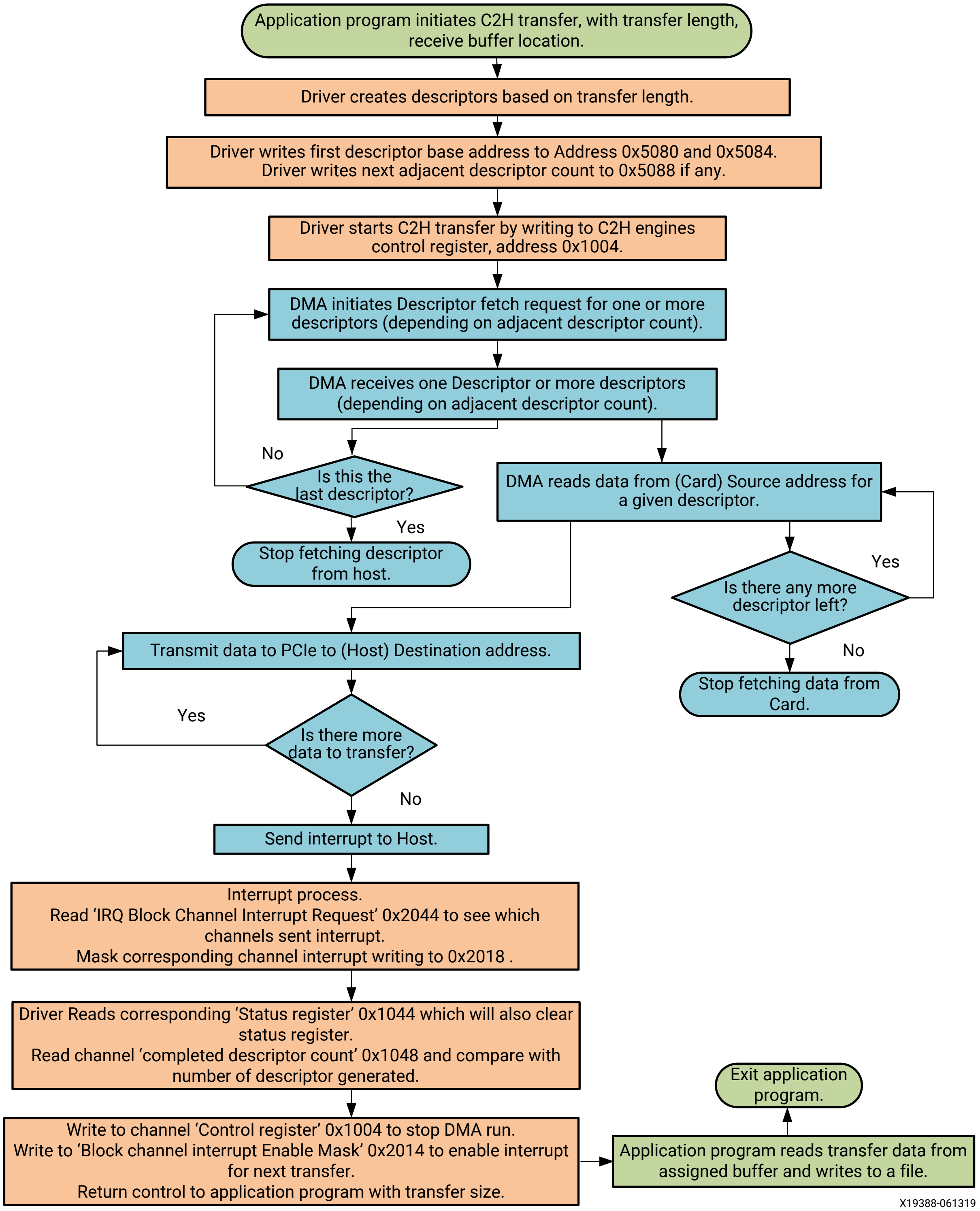
Figure 68: **DMA H2C Transfer Summary**


Application program initiates H2C transfer, with transfer length, buffer location where data is stored.
Driver creates descriptors based on transfer length.
Driver writes first descriptor base address to Address 0x4080 and 0x4084.
Driver writes next adjacent descriptor count to 0x4088 if any.
Driver starts H2C transfer by writing to H2C engines control register, address 0x0004.
DMA initiates Descriptor fetch request for one or more descriptors (depending on adjacent descriptor count).
DMA receives one Descriptor or more descriptors (depending on adjacent descriptor count).
No
Is this the last descriptor?
Yes
Stop fetching the descriptor from host.
DMA sends read request to (Host) source address based on first available descriptor.
DMA receives data from Host for that descriptor.
Transmit data on (Card) AXI-MM Master interface
Yes
Is there any more descriptor left?
No
Stop fetching data from Host.
Yes
Is there more data to transfer?
No
Send interrupt to Host.
Interrupt process.
Read 'IRQ Block Channel Interrupt Request' 0x2044 to see which channels sent interrupt.
Mask corresponding channel interrupt writing to 0x2018.
Driver Reads corresponding 'Status register' 0x0044 which will also clear the status register.
Read channel 'Completed descriptor count' 0x0048 and compare with the number of descriptor generated.
Write to channel 'Control register' 0x0004 to stop the DMA run.
Write to 'Block channel interrupt Enable Mask' 0x2014 to enable interrupt for next transfer.
Return control to the application program with the transfer size.
Exit application program.
X19389-061319

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 69: **DMA C2H Transfer Summary**


Application program initiates C2H transfer, with transfer length, receive buffer location.
Driver creates descriptors based on transfer length.
Driver writes first descriptor base address to Address 0x5080 and 0x5084. Driver writes next adjacent descriptor count to 0x5088 if any.
Driver starts C2H transfer by writing to C2H engines control register, address 0x1004.
DMA initiates Descriptor fetch request for one or more descriptors (depending on adjacent descriptor count).
DMA receives one Descriptor or more descriptors (depending on adjacent descriptor count).
No
Is this the last descriptor?
Yes
Stop fetching descriptor from host.
DMA reads data from (Card) Source address for a given descriptor.
Yes
Is there any more descriptor left?
No
Stop fetching data from Card.
Transmit data to PCIe to (Host) Destination address.
Yes
Is there more data to transfer?
No
Send interrupt to Host.
Interrupt process.
Read 'IRQ Block Channel Interrupt Request' 0x2044 to see which channels sent interrupt.
Mask corresponding channel interrupt writing to 0x2018 .
Driver Reads corresponding 'Status register' 0x1044 which will also clear status register.
Read channel 'completed descriptor count' 0x1048 and compare with number of descriptor generated.
Write to channel 'Control register' 0x1004 to stop DMA run.
Write to 'Block channel interrupt Enable Mask' 0x2014 to enable interrupt for next transfer.
Return control to application program with transfer size.
Application program reads transfer data from assigned buffer and writes to a file.
Exit application program.
X19388-061319

Descriptors

The XDMA Subsystem 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 XDMA 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 156: **Descriptor Format**

<table>
<tr><th>Offset</th><th colspan="4">Fields</th></tr>
<tr><td>0x0</td><td>Magic[15:0]</td><td>Rsv[1:0]</td><td>Nxt_adj[5:0]</td><td>Control[7:0]</td></tr>
<tr><td>0x04</td><td colspan="4">4’h0, Len[27:0]</td></tr>
<tr><td>0x08</td><td colspan="4">Src_adr[31:0]</td></tr>
<tr><td>0x0C</td><td colspan="4">Src_adr[63:32]</td></tr>
<tr><td>0x10</td><td colspan="4">Dst_adr[31:0]</td></tr>
</table>

Table 156: **Descriptor Format** *(cont'd)*

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

Table 157: **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. 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.
0x0C-0x8	Src_adr	63:0		Source address for H2C and memory mapped transfers. Metadata writeback address for C2H transfers.

Table 157: **Descriptor Fields** *(cont'd)*

Offset	Field	Bit Index	Sub Field	Description
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 and Gen4x8 with 4H2C and 4C2H design, AXI bit width is 512 bits. FIFO depth is $512\ \mathrm{bit} * 512 = 64\ \mathrm{B} * 512 = 32\ \mathrm{KB}$ (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 $256\ \mathrm{bit} * 512 = 32\ \mathrm{B} * 512 = 16\ \mathrm{KB}$ (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 158: **Completed Descriptor Count Writeback Format**

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

Table 159: **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 161: 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 160: **C2H Stream Writeback Format**

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

Table 161: **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.

***Note*:** C2H Streaming writeback address cannot cross 4K boundary.

Address Alignment

Table 162: **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	64	Source_addr[2:0] == Destination_addr[2:0] == 3’h0
AXI4 MM fixed address	128	Source_addr[3:0] == Destination_addr[3:0] == 4’h0
AXI4 MM fixed address	256	Source_addr[4:0] == Destination_addr[4:0] == 5’h0
AXI4 MM fixed address	512	Source_addr[5:0] == Destination_addr[5:0]==6'h0

Length Granularity

Table 163: **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

Table 163: **Length Granularity** *(cont'd)*

Interface Type	Datapath Width	Length Granularity Restriction
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

Set the **Propagate Parity** option in the PCIe DMA Tab in the Vivado® IDE to check for parity. Otherwise, no parity checking occurs.

When **Propagate Parity** is enabled, the XDMA 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

***Note*:** The Versal ACAP DMA and Bridge Subsystem for PCIe IP is implemented in a modular IP architecture. This means that GTs, PCIe IP, and the subsystem IP are implemented separately. The interface signals between GTs and PCIe IP going to a subsystem IP are not listed in this guide. These interface signals are found in *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343). The signals below apply to the subsystem only.

The XDMA Subsystem 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 subsystem are described in the following tables.

XDMA Global Ports

Table 164: **Top-Level Interface Signals**

Signal Name	Direction	Description
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.
user_lnk_up_sd	I	Active-High Identifies that the PCI Express core is linked up with a host device. This signal is from the integrated block for PCIe.
phy_rdy_out_sd	I	Active-High signal that indicates when Phy is ready. This signal is from the Phy block.
user_clk_sd	I	User clock from the PCIe block. All of the QDMA blocks use this clock
user_reset_sd	I	Active-High user reset signals from the PCIe block.

H2C Channel 0-3 AXI4-Stream Interface Signals

Table 165: **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.
m_axis_h2c_tkeep_*x* [DATA_WIDTH/8-1:0]	O	The tkeep signal specifies how many bytes are valid when tlast is asserted.

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 166: **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.
m_axis_c2h_tkeep_*x* [DATA_WIDTH/8-1:0]	I	The tkeep signal tells how many bytes are valid for each beat. It must be asserted for all beets except when tlast is asserted. You must specify how many bytes are valid when tlast is asserted.

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 167: **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 168: **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 169: **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 170: **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 170: **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 171: **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 172: **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

Table 173: **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 174: **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

Table 175: **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 176: **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 177: **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 177: **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 178: **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 179: **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 180: **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 181: **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 182: **Channel 0-3 Status Ports**

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

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.

Table 183: **Configuration Extend Interface Port Descriptions**

Port	Direction	Width	Description
cfg_ext_read_received	O	1	Configuration Extend Read Received The core asserts this output when it has received a configuration read request from the link. When neither user-implemented legacy or extended configuration space is enabled, receipt of a configuration read results in a one-cycle assertion of this signal, together with valid cfg_ext_register_number and cfg_ext_function_number. When user-implemented legacy, extended configuration space, or both are enabled, for the `cfg_ext_register_number` 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. Legacy Space: 0xB0-0xBF Extended Configuration space: 0x480 - 0x4FF

Table 183: **Configuration Extend Interface Port Descriptions** *(cont'd)*

Port	Direction	Width	Description
cfg_ext_write_received	O	1	Configuration Extend Write Received The core generates a one-cycle pulse on this output when it has received a configuration write request from the link.
cfg_ext_register_number	O	10	Configuration Extend Register Number The 10-bit address of the configuration register being read or written. The data is valid when cfg_ext_read_received or cfg_ext_write_received is High.
cfg_ext_function_number	O	8	Configuration Extend Function Number The 8-bit function number corresponding to the configuration read or write request. The data is valid when cfg_ext_read_received or cfg_ext_write_received is High.
cfg_ext_write_data	O	32	Configuration Extend Write Data Data being written into a configuration register. This output is valid when cfg_ext_write_received is High.
cfg_ext_write_byte_enable	O	4	Configuration Extend Write Byte Enable Byte enables for a configuration write transaction.
cfg_ext_read_data	I	32	Configuration Extend Read Data You can provide data from an externally implemented configuration register to the core through this bus. The core samples this data on the next positive edge of the clock after it sets cfg_ext_read_received High, if you have set cfg_ext_read_data_valid.
cfg_ext_read_data_valid	I	1	Configuration Extend Read Data Valid The user application asserts this input to the core to supply data from an externally implemented configuration register. The core samples this input data on the next positive edge of the clock after it sets cfg_ext_read_received High.

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 184: **Configuration Management Interface Ports**

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

Table 184: **Configuration Management Interface Ports** *(cont'd)*

Port	Direction	Width	Description
cfg_mgmt_read_write_done	O	1	Read/Write Operation Complete Asserted for 1 cycle when operation is complete. Active-High
cfg_mgmt_write_data	I	32	Write data Write data is used to configure the Configuration and Management registers
cfg_mgmt_write	I	1	Write Enable Asserted for a write operation. Active-High

Descriptor Bypass Mode

In the PCIe DMA tab of the Vivado IDE, if **Descriptor Bypass for Read (H2C)** or **Descriptor Bypass for Write (C2H)** is 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 185: **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. [0]: Stop. Set to 1 to stop fetching next descriptor. [1]: Completed. Set to 1 to interrupt after the engine has completed this descriptor. [3:2]: Reserved. [4]: EOP. End of Packet for AXI-Stream interface. [15:5]: Reserved. All reserved bits can be forced to 0s. control port "h2c_dsc_byp_ctl[4:0]" are same as in descriptor control, refer to Table 6.

Table 186: **C2H 0-3 Descriptor Bypass Ports**

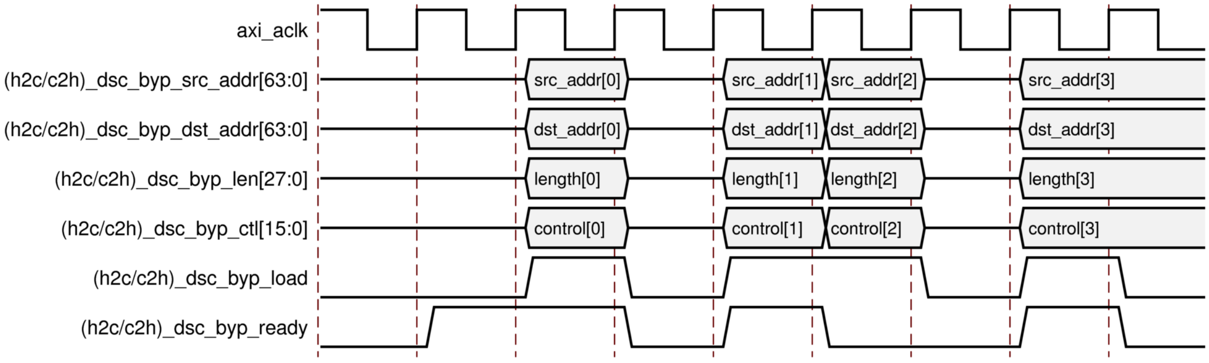
Port	Direction	Description
c2h_dsc_byp_ready	O	Channel is ready to accept new descriptors. After c2h_dsc_byp_ready is deasserted, one additional descriptor can be written. The Control register 'Run' bit must be asserted before the channel accepts descriptors.
c2h_dsc_byp_load	I	Descriptor presented at c2h_dsc_byp_* is valid.

Table 186: **C2H 0-3 Descriptor Bypass Ports** *(cont'd)*

Port	Direction	Description
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. [0]: Stop. Set to 1 to stop fetching next descriptor. [1]: Completed. Set to 1 to interrupt after the engine has completed this descriptor. [3:2]: Reserved. [4]: EOP. End of Packet for AXI-Stream interface. [15:5]: Reserved. All reserved bits can be forced to 0s. control port "h2c_dsc_byp_ctl[4:0]" are same as in descriptor control, refer to Table 6.

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 70: **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

Configuration and status registers internal to the XDMA Subsystem 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 Bridge Master Address Map

Transactions that hit the PCIe to AXI Bridge Master are routed to the AXI4 Memory Mapped user interface.

PCIe to DMA Address Map

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

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

PCIe to DMA Address Format

Table 187: **PCIe to DMA Address Format**

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

Table 188: **PCIe to DMA Address Field Descriptions**

Bit Index	Field	Description
15:12	Target	The destination submodule within the DMA 4’h0: H2C Channels 4’h1: C2H Channels 4’h2: IRQ Block 4’h3: Config 4’h4: H2C SGDMA 4’h5: C2H SGDMA 4’h6: SGDMA Common 4'h8: MSI-X
11:8	Channel ID[3:0]	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.

Table 188: **PCIe to DMA Address Field Descriptions** *(cont'd)*

Bit Index	Field	Description
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 189: **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 190: **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)
0x8C	H2C Poll Mode High Write Back Address (0x8C)
0x90	H2C Channel Interrupt Enable Mask (0x90)

Table 190: **H2C Channel Register Space** *(cont'd)*

Address (hex)	Register Name
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 191: **H2C Channel Identifier (0x00)**

Bit Index	Default Value	Access Type	Description
31:20	12’h1fc	RO	Subsystem identifier
19:16	4’h0	RO	H2C Channel Target
15	1’b0	RO	Stream 1: AXI4-Stream Interface 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 8'h01: 2015.3 and 2015.4 8'h02: 2016.1 8'h03: 2016.2 8'h04: 2016.3 8'h05: 2016.4 8'h06: 2017.1 to current release

H2C Channel Control (0x04)

Table 192: **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.
26	0x0	RW	pollmode_wb_enable Poll mode writeback enable. When this bit is set the DMA writes back the completed descriptor count when a descriptor with the Completed bit set, is completed.

Table 192: **H2C Channel Control (0x04)** *(cont'd)*

Bit Index	Default	Access Type	Description
25	1'b0	RW	non_inc_mode Non-incrementing address mode. Applies to m_axi_araddr interface only.
24			Reserved
23:19	5'h0	RW	ie_desc_error Set to all 1s (0x1F) to enable logging of Status.Desc_error and to stop the engine if the error is detected.
18:14	5'h0	RW	ie_write_error Set to all 1s (0x1F) to enable logging of Status.Write_error and to stop the engine if the error is detected.
13:9	5'h0	RW	ie_read_error Set to all 1s (0x1F) to enable logging of Status.Read_error and to stop the engine if the error is detected.
8:7			Reserved
6	1'b0	RW	ie_idle_stopped Set to 1 to enable logging of Status.Idle_stopped
5	1'b0	RW	ie_invalid_length Set to 1 to enable logging of Status.Invalid_length
4	1'b0	RW	ie_magic_stopped Set to 1 to enable logging of Status.Magic_stopped
3	1'b0	RW	ie_align_mismatch Set to 1 to enable logging of Status.Align_mismatch
2	1'b0	RW	ie_descriptor_completed Set to 1 to enable logging of Status.Descriptor_completed
1	1'b0	RW	ie_descriptor_stopped Set to 1 to enable logging of Status.Descriptor_stopped
0	1'b0	RW	Run Set to 1 to start the SGDMA engine. Reset to 0 to stop 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 193: **H2C Channel Control (0x08)**

Bit Index	Default	Access Type	Description
31:28			Reserved
27:0		W1S	Control Bit descriptions are the same as in H2C Channel Control (0x04).

H2C Channel Control (0x0C)

Table 194: **H2C Channel Control (0x0C)**

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

H2C Channel Status (0x40)

Table 195: **H2C Channel Status (0x40)**

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

Table 195: **H2C Channel Status (0x40)** *(cont'd)*

Bit Index	Default	Access Type	Description
3	1’b0	RW1C	align_mismatch Source and destination address on descriptor are not properly aligned to each other.
2	1’b0	RW1C	descriptor_completed Reset on setting 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 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 Set if the SGDMA engine is busy. Zero when it is idle.

H2C Channel Status (0x44)

Table 196: **H2C Channel Status (0x44)**

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

H2C Channel Completed Descriptor Count (0x48)

Table 197: **H2C Channel Completed Descriptor Count (0x48)**

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

H2C Channel Alignments (0x4C)

Table 198: **H2C Channel Alignments (0x4C)**

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

Table 198: **H2C Channel Alignments (0x4C)** *(cont'd)*

Bit Index	Default	Access Type	Description
7:0	Configuration based	RO	address_bits The number of address bits configured.

H2C Poll Mode Low Write Back Address (0x88)

Table 199: **H2C Poll Mode Low Write Back Address (0x88)**

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

H2C Poll Mode High Write Back Address (0x8C)

Table 200: **H2C Poll Mode High Write Back Address (0x8C)**

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

H2C Channel Interrupt Enable Mask (0x90)

Table 201: **H2C Channel Interrupt Enable Mask (0x90)**

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

Table 201: **H2C Channel Interrupt Enable Mask (0x90)** *(cont'd)*

Bit Index	Default	Access Type	Description
3	1’b0	RW	im_align_mismatch set to 1 to interrupt when status register align_mismatch bit is logged.
2	1’b0	RW	im_descriptor_completd set to 1 to interrupt when status register descriptor_completed bit is logged.
1	1’b0	RW	im_descriptor_stopped set to 1 to interrupt when status register descriptor_stopped bit is logged.

H2C Channel Interrupt Enable Mask (0x94)

Table 202: **H2C Channel Interrupt Enable Mask (0x94)**

Bit Index	Default	Access Type	Description
		W1S	Interrupt Enable Mask

H2C Channel Interrupt Enable Mask (0x98)

Table 203: **H2C Channel Interrupt Enable Mask (0x98)**

Bit Index	Default	Access Type	Description
		W1C	Interrupt Enable Mask

H2C Channel Performance Monitor Control (0xC0)

Table 204: **H2C Channel Performance Monitor Control (0xC0)**

Bit Index	Default	Access Type	Description
2	1’b0	RW	Run Set to 1 to arm performance counters. Counter starts after the Control register Run bit is set. Set to 0 to halt performance counters.
1	1’b0	WO	Clear Write 1 to clear performance counters.
0	1’b0	RW	Auto Automatically stop performance counters when a descriptor with the stop bit is completed. Automatically clear performance counters when the Control 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 205: **H2C Channel Performance Cycle Count (0xC4)**

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

H2C Channel Performance Cycle Count (0xC8)

Table 206: **H2C Channel Performance Cycle Count (0xC8)**

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

H2C Channel Performance Data Count (0xCC)

Table 207: **H2C Channel Performance Data Count (0xCC)**

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

H2C Channel Performance Data Count (0xD0)

Table 208: **H2C Channel Performance Data Count (0xD0)**

Bit Index	Default	Access Type	Description
16	1’b0	RO	pmon_dat_count_maxed Data count maximum was hit
15:10			Reserved
9:0	10’h0	RO	pmon_dat_count [41:32] 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 209: **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 210: **C2H Channel Identifier (0x00)**

Bit Index	Default	Access Type	Description
31:20	12’h1fc	RO	Subsystem identifier
19:16	4’h1	RO	C2H Channel Target
15	1’b0	RO	Stream 1: AXI4-Stream Interface 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 8'h01: 2015.3 and 2015.4 8'h02: 2016.1 8'h03: 2016.2 8'h04: 2016.3 8'h05: 2016.4 8'h06: 2017.1 to current release

C2H Channel Control (0x04)

Table 211: **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 Poll mode writeback enable. When this bit is set, the DMA writes back the completed descriptor count when a descriptor with the Completed bit set, is completed.
25	1’b0	RW	non_inc_mode Non-incrementing address mode. Applies to m_axi_araddr interface only.
23:19	5’h0	RW	ie_desc_error Set to all 1s (0x1F) to enable logging of Status.Desc_error and to stop the engine if the error is detected.
13:9	5’h0	RW	ie_read_error Set to all 1s (0x1F) to enable logging of Status.Read_error and to stop the engine if the error is detected
8:7			Reserved
6	1’b0	RW	ie_idle_stopped Set to 1 to enable logging of Status.Idle_stopped
5	1’b0	RW	ie_invalid_length Set to 1 to enable logging of Status.Invalid_length
4	1’b0	RW	ie_magic_stopped Set to 1 to enable logging of Status.Magic_stopped
3	1’b0	RW	ie_align_mismatch Set to 1 to enable logging of Status.Align_mismatch
2	1’b0	RW	ie_descriptor_completed Set to 1 to enable logging of Status.Descriptor_completed
1	1’b0	RW	ie_descriptor_stopped Set to 1 to enable logging of Status.Descriptor_stopped
0	1’b0	RW	Run Set to 1 to start the SGDMA engine. Reset to 0 to stop the 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, 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 212: **C2H Channel Control (0x08)**

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

C2H Channel Control (0x0C)

Table 213: **C2H Channel Control (0x0C)**

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

C2H Channel Status (0x40)

Table 214: **C2H Channel Status (0x40)**

Bit Index	Default	Access Type	Description
23:19	5’h0	RW1C	descr_error[4:0] Reset (0) on setting the Control register Run bit. Bit position: 4:Unexpected completion 3: Header EP 2: Parity error 1: Completer abort 0: Unsupported request
13:9	5’h0	RW1C	read_error[4:0] Reset (0) on setting the Control register Run bit. Bit position: 4-2: Reserved 1: Slave error 0: Decode error
8:7			Reserved
6	1’b0	RW1C	idle_stopped 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 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.

Table 214: **C2H Channel Status (0x40)** *(cont'd)*

Bit Index	Default	Access Type	Description
4	1’b0	RW1C	magic_stopped 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 Source and destination address on descriptor are not properly aligned to each other.
2	1’b0	RW1C	descriptor_completed 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 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 Set if the SGDMA engine is busy. Zero when it is idle.

C2H Channel Status (0x44)

Table 215: **C2H Channel Status (0x44)**

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

C2H Channel Completed Descriptor Count (0x48)

Table 216: **C2H Channel Completed Descriptor Count (0x48)**

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

C2H Channel Alignments (0x4C)

Table 217: **C2H Channel Alignments (0x4C)**

Bit Index	Default	Access Type	Description
23:16	varies	RO	addr_alignment The byte alignment that the source and destination addresses must align to. This value is dependent on configuration parameters.

Table 217: **C2H Channel Alignments (0x4C)** *(cont'd)*

Bit Index	Default	Access Type	Description
15:8	Varies	RO	len_granularity The minimum granularity of DMA transfers in bytes.
7:0	ADDR_BITS	RO	address_bits The number of address bits configured.

C2H Poll Mode Low Write Back Address (0x88)

Table 218: **C2H Poll Mode Low Write Back Address (0x88)**

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

C2H Poll Mode High Write Back Address (0x8C)

Table 219: **C2H Poll Mode High Write Back Address (0x8C)**

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

C2H Channel Interrupt Enable Mask (0x90)

Table 220: **C2H Channel Interrupt Enable Mask (0x90)**

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

C2H Channel Interrupt Enable Mask (0x94)

Table 221: **C2H Channel Interrupt Enable Mask (0x94)**

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

C2H Channel Interrupt Enable Mask (0x98)

Table 222: **C2H Channel Interrupt Enable Mask (0x98)**

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

C2H Channel Performance Monitor Control (0xC0)

Table 223: **C2H Channel Performance Monitor Control (0xC0)**

Bit Index	Default	Access Type	Description
2	1’b0	RW	Run Set to 1 to arm performance counters. Counter starts after the Control register Run bit is set. Set to 0 to halt performance counters.
1	1’b0	WO	Clear Write 1 to clear performance counters.
0	1’b0	RW	Auto Automatically stop performance counters when a descriptor with the stop bit is completed. Automatically clear performance counters when the Control 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 224: **C2H Channel Performance Cycle Count (0xC4)**

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

C2H Channel Performance Cycle Count (0xC8)

Table 225: **C2H Channel Performance Cycle Count (0xC8)**

Bit Index	Default	Access Type	Description
16	1'b0	RO	pmon_cyc_count_maxed Cycle count maximum was hit.
15:10			Reserved
9:0	10'h0	RO	pmon_cyc_count [41:32] 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 226: **C2H Channel Performance Data Count (0xCC)**

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

C2H Channel Performance Data Count (0xD0)

Table 227: **C2H Channel Performance Data Count (0xD0)**

Bit Index	Default	Access Type	Description
16	1'b0	RO	pmon_dat_count_maxed Data count maximum was hit
15:10			Reserved
9:0	10'h0	RO	pmon_dat_count [41:32] 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 228: **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)

Table 228: **IRQ Block Register Space** *(cont'd)*

Address (hex)	Register Name
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)
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 229: **IRQ Block Identifier (0x00)**

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

IRQ Block User Interrupt Enable Mask (0x04)

Table 230: **IRQ Block User Interrupt Enable Mask (0x04)**

Bit Index	Default	Access Type	Description
[NUM_USR_INT-1:0]	'h0	RW	user_int_enmask User Interrupt Enable Mask 0: Prevents an interrupt from being generated when the user interrupt source is asserted. 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 231: **IRQ Block User Interrupt Enable Mask (0x08)**

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

IRQ Block User Interrupt Enable Mask (0x0C)

Table 232: **IRQ Block User Interrupt Enable Mask (0x0C)**

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

IRQ Block Channel Interrupt Enable Mask (0x10)

Table 233: **IRQ Block Channel Interrupt Enable Mask (0x10)**

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

IRQ Block Channel Interrupt Enable Mask (0x14)

Table 234: **IRQ Block Channel Interrupt Enable Mask (0x14)**

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

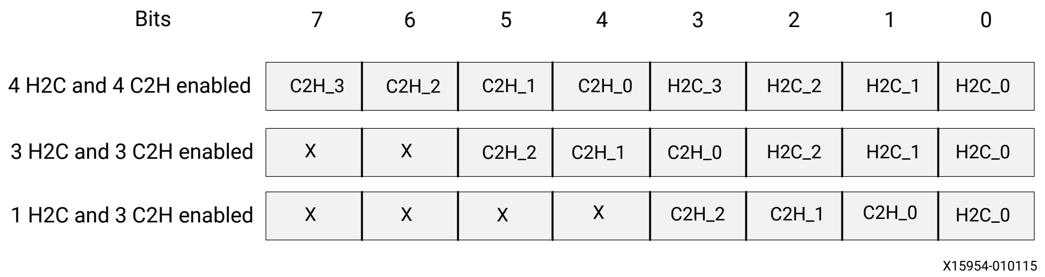
IRQ Block Channel Interrupt Enable Mask (0x18)

Table 235: **IRQ Block Channel Interrupt Enable Mask (0x18)**

Bit Index	Default	Access Type	Description
		W1C	channel_int_enmask 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 71: **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

IRQ Block User Interrupt Request (0x40)

Table 236: **IRQ Block User Interrupt Request (0x40)**

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

IRQ Block Channel Interrupt Request (0x44)

Table 237: **IRQ Block Channel Interrupt Request (0x44)**

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

IRQ Block User Interrupt Pending (0x48)

Table 238: **IRQ Block User Interrupt Pending (0x48)**

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

IRQ Block Channel Interrupt Pending (0x4C)

Table 239: **IRQ Block Channel Interrupt Pending (0x4C)**

Bit Index	Default	Access Type	Description
[NUM_CHNL-1:0]	'h0	RO	engine_int_pend Engine Interrupt Pending. One bit per read or write engine. This register indicates pending events. The pending events are cleared by removing the event cause condition at the source component. The position of the H2C bits always starts at bit 0. The position of the C2H bits is the index above the last H2C index, and therefore depends on the NUM_H2C_CHNL parameter. 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 240: **IRQ Block User Vector Number (0x80)**

Bit Index	Default	Access Type	Description
28:24	5’h0	RW	vector 3 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[3].
20:16	5’h0	RW	vector 2 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[2].
12:8	5’h0	RW	vector 1 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[1].
4:0	5’h0	RW	vector 0 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[0].

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 241: **IRQ Block User Vector Number (0x84)**

Bit Index	Default	Access Type	Description
28:24	5’h0	RW	vector 7 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[7].
20:16	5’h0	RW	vector 6 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[6].
12:8	5’h0	RW	vector 5 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[5].
4:0	5’h0	RW	vector 4 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[4].

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 242: **IRQ Block User Vector Number (0x88)**

Bit Index	Default	Access Type	Description
28:24	5’h0	RW	vector 11 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[11].

Table 242: **IRQ Block User Vector Number (0x88)** *(cont'd)*

Bit Index	Default	Access Type	Description
20:16	5’h0	RW	vector 10 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[10].
12:8	5’h0	RW	vector 9 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[9].
4:0	5’h0	RW	vector 8 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[8].

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 243: **IRQ Block User Vector Number (0x8C)**

Bit Index	Default	Access Type	Description
28:24	5’h0	RW	vector 15 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[15].
20:16	5’h0	RW	vector 14 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[14].
12:8	5’h0	RW	vector 13 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[13].
4:0	5’h0	RW	vector 12 The vector number that is used when an interrupt is generated by the user IRQ usr_irq_req[12].

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 244: **IRQ Block Channel Vector Number (0xA0)**

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

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 245: **IRQ Block Channel Vector Number (0xA4)**

Bit Index	Default	Access Type	Description
28:24	5'h0	RW	vector7 The vector number that is used when an interrupt is generated by channel 7.
20:16	5'h0	RW	vector6 The vector number that is used when an interrupt is generated by channel 6.
12:8	5'h0	RW	vector5 The vector number that is used when an interrupt is generated by channel 5.
4:0	5'h0	RW	vector4 The vector number that is used when an interrupt is generated by channel 4.

Config Block Registers (0x3)

The Config Block registers are described in this section.

Table 246: **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 247: **Config Block Identifier (0x00)**

Bit Index	Default	Access Type	Description
31:20	12’h1fc	RO	Subsystem identifier
19:16	4’h3	RO	Config identifier
15:8	8’h0	RO	Reserved
7:0	8'h04	RO	Version 8'h01: 2015.3 and 2015.4 8'h02: 2016.1 8'h03: 2016.2 8'h04: 2016.3 8'h05: 2016.4 8'h06: 2017.1 to current release

Config Block BusDev (0x04)

Table 248: **Config Block BusDev (0x04)**

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

Config Block PCIE Max Payload Size (0x08)

Table 249: **Config Block PCIE Max Payload Size (0x08)**

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

Config Block PCIE Max Read Request Size (0x0C)

Table 250: **Config Block PCIE Max Read Request Size (0x0C)**

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

Config Block System ID (0x10)

Table 251: **Config Block System ID (0x10)**

Bit Index	Default	Access Type	Description
[15:0]	16’hff01	RO	system_id Core system ID

Config Block MSI Enable (0x14)

Table 252: **Config Block MSI Enable (0x14)**

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

Config Block PCIE Data Width (0x18)

Table 253: **Config Block PCIE Data Width (0x18)**

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

Config PCIE Control (0x1C)

Table 254: **Config PCIE Control (0x1C)**

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

Config AXI User Max Payload Size (0x40)

Table 255: **Config AXI User Max Payload Size (0x40)**

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

Config AXI User Max Read Request Size (0x44)

Table 256: **Config AXI User Max Read Request Size (0x44)**

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

Config Write Flush Timeout (0x60)

Table 257: **Config Write Flush Timeout (0x60)**

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

H2C SGDMA Registers (0x4)

Table 258: **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 259: **H2C SGDMA Identifier (0x00)**

Bit Index	Default	Access Type	Description
31:20	12’h1fc	RO	Subsystem identifier
19:16	4’h4	RO	H2C DMA Target
15	1’b0	RO	Stream 1: AXI4-Stream Interface 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 8'h01: 2015.3 and 2015.4 8'h02: 2016.1 8'h03: 2016.2 8'h04: 2016.3 8'h05: 2016.4 8'h06: 2017.1 to current release

H2C SGDMA Descriptor Low Address (0x80)

Table 260: **H2C SGDMA Descriptor Low Address (0x80)**

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

H2C SGDMA Descriptor High Address (0x84)

Table 261: **H2C SGDMA Descriptor High Address (0x84)**

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

H2C SGDMA Descriptor Adjacent (0x88)

Table 262: **H2C SGDMA Descriptor Adjacent (0x88)**

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

H2C SGDMA Descriptor Credits (0x8C)

Table 263: **H2C SGDMA Descriptor Credits (0x8C)**

Bit Index	Default	Access Type	Description
9:0	10'h0	RW	h2c_dsc_credit[9:0] 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. 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 264: **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 265: **C2H SGDMA Identifier (0x00)**

Bit Index	Default	Access Type	Description
31:20	12’h1fc	RO	Subsystem identifier
19:16	4’h5	RO	C2H DMA Target
15	1’b0	RO	Stream 1: AXI4-Stream Interface 0: AXI4 Memory Mapped Interface

Table 265: **C2H SGDMA Identifier (0x00)** *(cont'd)*

Bit Index	Default	Access Type	Description
14:12	3’h0	RO	Reserved
11:8	Varies	RO	Channel ID Target [3:0]
7:0	8'h04	RO	Version 8'h01: 2015.3 and 2015.4 8'h02: 2016.1 8'h03: 2016.2 8'h04: 2016.3 8'h05: 2016.4 8'h06: 2017.1 to current release

C2H SGDMA Descriptor Low Address (0x80)

Table 266: **C2H SGDMA Descriptor Low Address (0x80)**

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

C2H SGDMA Descriptor High Address (0x84)

Table 267: **C2H SGDMA Descriptor High Address (0x84)**

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

C2H SGDMA Descriptor Adjacent (0x88)

Table 268: **C2H SGDMA Descriptor Adjacent (0x88)**

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

C2H SGDMA Descriptor Credits (0x8C)

Table 269: **C2H SGDMA Descriptor Credits (0x8C)**

Bit Index	Default	Access Type	Description
9:0	10'h0	RW	c2h_dsc_credit[9:0] 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. 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 270: **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 271: **SGDMA Identifier Registers (0x00)**

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

SGDMA Descriptor Control Register (0x10)

Table 272: **SGDMA Descriptor Control Register (0x10)**

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

SGDMA Descriptor Control Register (0x14)

Table 273: **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 274: **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 275: **SGDMA Descriptor Credit Mode Enable (0x20)**

Bit Index	Default	Access Type	Description
3:0	0x0	RW	h2c_dsc_credit_enable [3:0] 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] 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 276: **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 277: **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 278: **MSI-X Vector Table and PBA (0x00–0xFE0)**

Byte Offset	Bit Index	Default	Access Type	Description
0x00	31:0	32’h0	RW	MSIX_Vector0_Address[31:0] MSI-X vector0 message lower address.
0x04	31:0	32’h0	RW	MSIX_Vector0_Address[63:32] MSI-X vector0 message upper address.
0x08	31:0	32’h0	RW	MSIX_Vector0_Data[31:0] MSI-X vector0 message data.
0x0C	31:0	32'hFFFFFFFF	RW	MSIX_Vector0_Control[31:0] MSI-X vector0 control. Bit Position: 31:1: Reserved. 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] MSI-X vector31 message lower address.
0x1F4	31:0	32’h0	RW	MSIX_Vector31_Address[63:32] MSI-X vector31 message upper address.
0x1F8	31:0	32’h0	RW	MSIX_Vector31_Data[31:0] MSI-X vector31 message data.

Table 278: **MSI-X Vector Table and PBA (0x00–0xFE0)** *(cont'd)*

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

Design Flow Steps

This section describes customizing and generating the subsystem, constraining the subsystem, and the simulation, synthesis, and implementation steps that are specific to this IP subsystem. 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 subsystem in the Vivado® Design Suite.

If you are customizing and generating the subsystem 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 subsystem using the following steps:

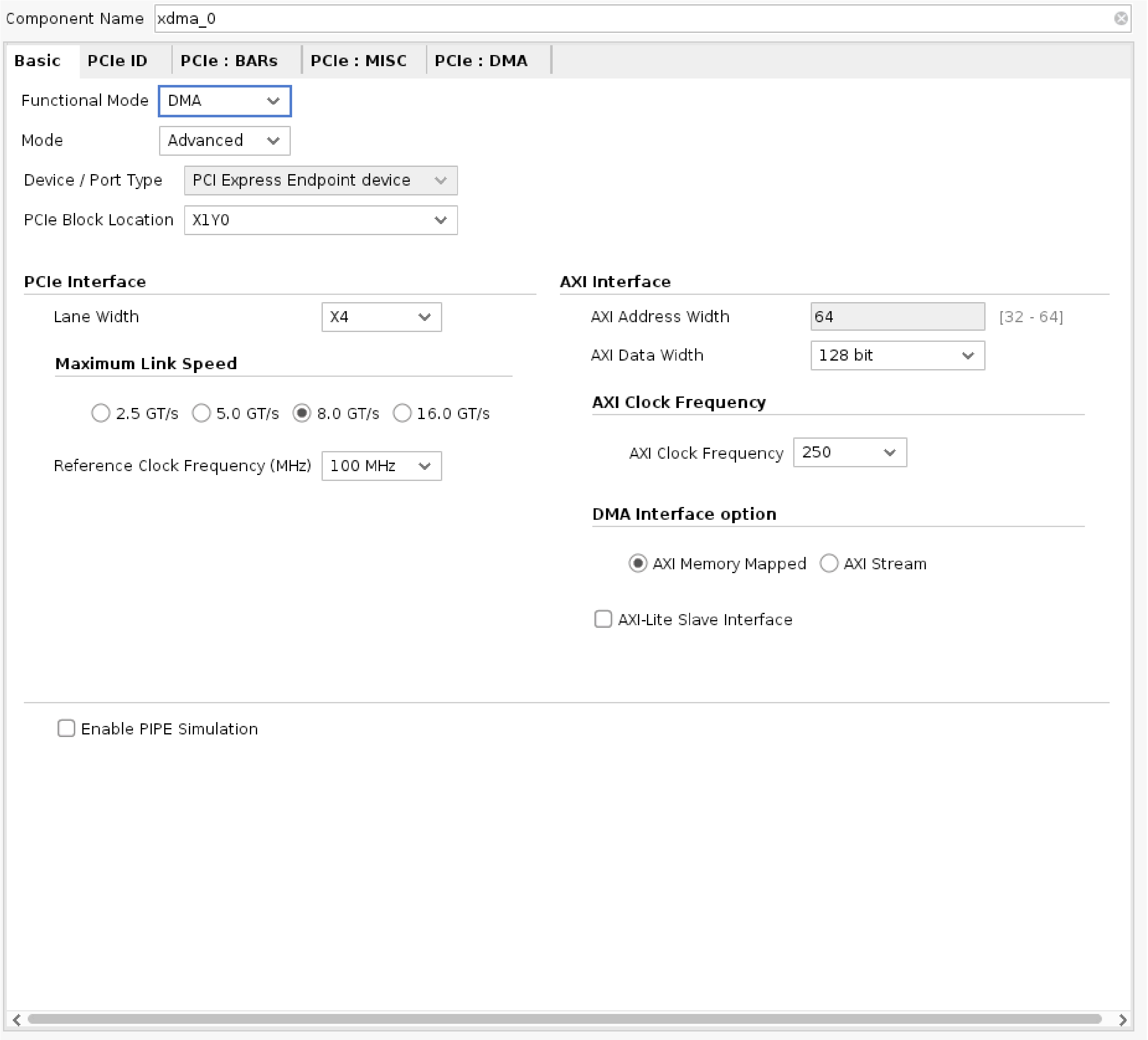
1. Select the IP from the IP catalog.
2. Double-click the selected IP or select the Customize IP command from the toolbar or right-click menu.

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

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

Basic Tab

Figure 72: **Basic Tab for DMA Functional Mode**



Component Name: xdma_0

Functional Mode: DMA

Mode: Advanced

Device / Port Type: PCI Express Endpoint device

PCIe Block Location: X1Y0

PCIe Interface

Lane Width: X4

Maximum Link Speed

○ 2.5 GT/s ○ 5.0 GT/s ● 8.0 GT/s ○ 16.0 GT/s

Reference Clock Frequency (MHz): 100 MHz

AXI Interface

AXI Address Width: 64 [32 - 64]

AXI Data Width: 128 bit

AXI Clock Frequency

AXI Clock Frequency: 250

DMA Interface option

● AXI Memory Mapped ○ AXI Stream

□ AXI-Lite Slave Interface

□ Enable PIPE Simulation

The options are defined as follows:

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

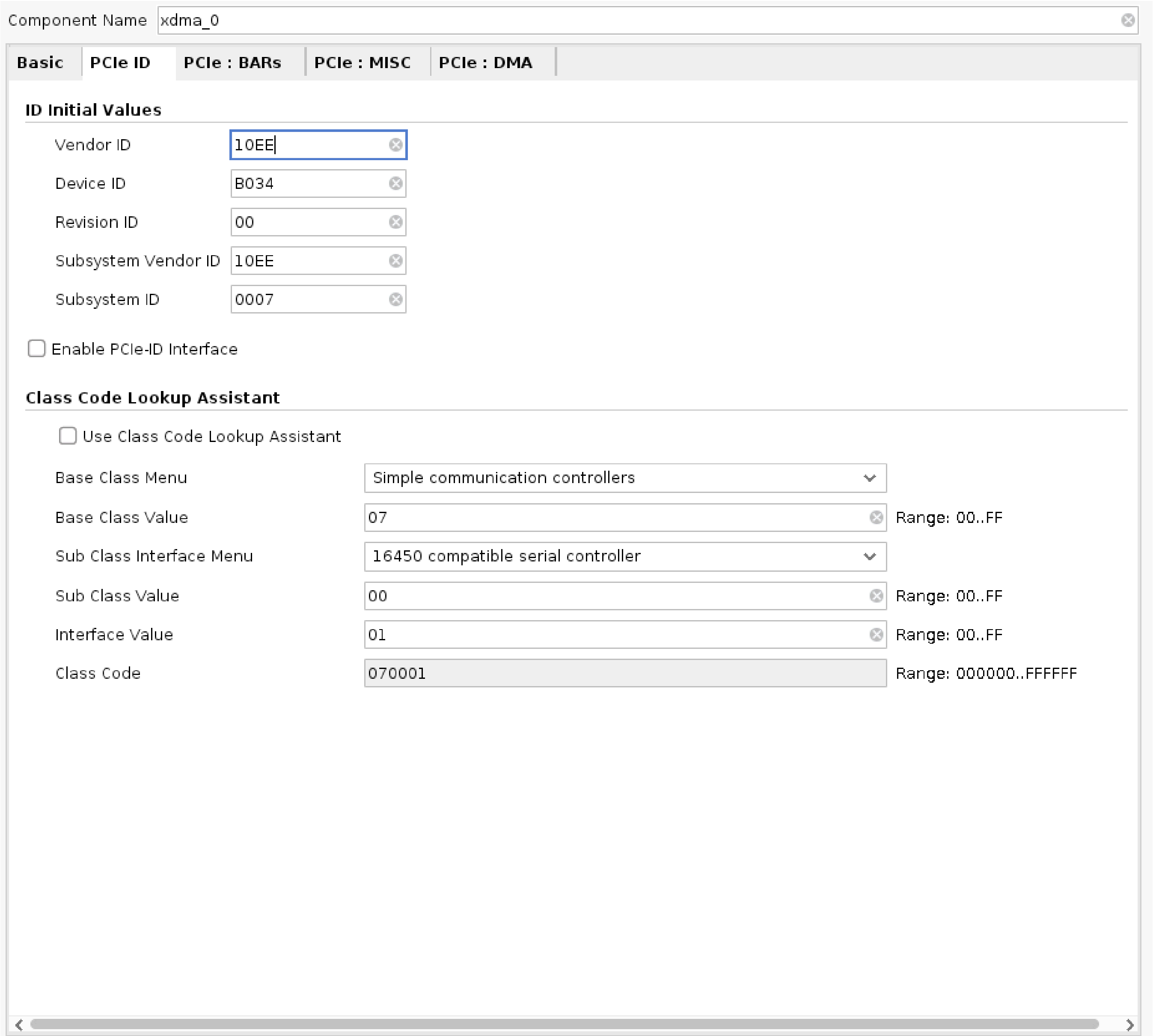
- **PCIe Block Location:** Selects from the available integrated blocks to enable generation of location-specific constraint files and pinouts. This selection is used in the default example design scripts. This option is not available if a Xilinx Development Board is selected.
- **Lane Width:** The subsystem requires the selection of the initial lane width. For supported lane widths and link speeds, see the *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343). Higher link speed cores are capable of training to a lower link speed if connected to a lower link speed capable device.
- **Maximum Link Speed:** The subsystem requires the selection of the PCIe Gen speed.
- **Reference Clock Frequency:** The default is 100 MHz, but 125 MHz and 250 MHz are also supported.
- **AXI Address Width:** Currently, only 64-bit width is supported.
- **AXI Data Width:** Select 64, 128, 256-bit, or 512-bit. The subsystem allows you to select the Interface Width, as defined in the *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343).
- **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.
- **Enable PIPE Simulation:** When selected, this option enables an external third-party bus functional model (BFM) to connect to the PIPE interface of the integrated block for PCIe.

RECOMMENDED: *Xilinx recommends that you select the correct GT starting quad before setting the link rate and width. Selecting line rate and width prior to selecting GT quad can have adverse effects.*

PCIe ID Tab

The PCIe ID tab is shown in the following figure.

Figure 73: **PCIe ID Tab**



Component Name: xdma_0

ID Initial Values

Vendor ID	10EE
Device ID	B034
Revision ID	00
Subsystem Vendor ID	10EE
Subsystem ID	0007

□ Enable PCIe-ID Interface

Class Code Lookup Assistant

□ Use Class Code Lookup Assistant

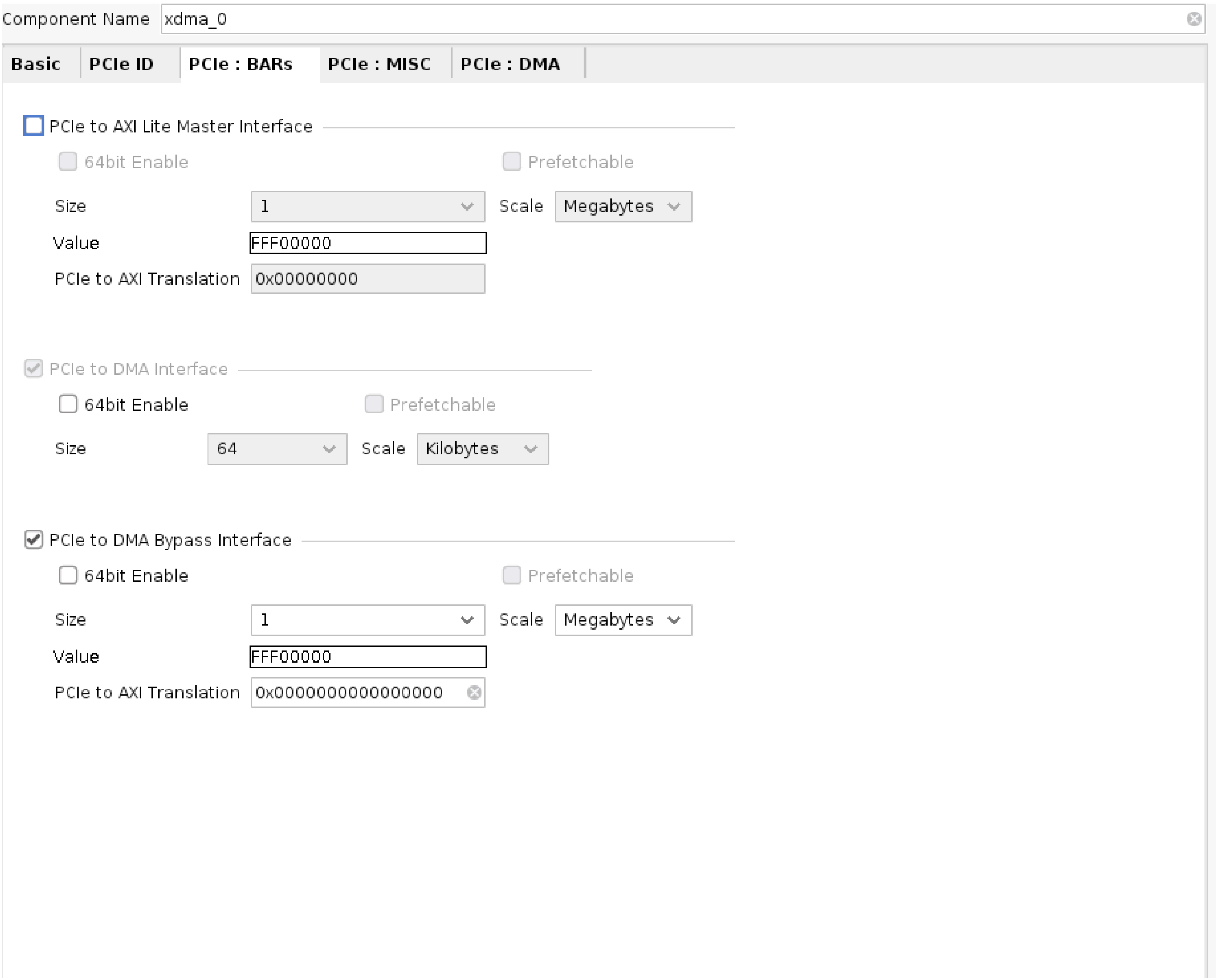
Base Class Menu	Simple communication controllers	
Base Class Value	07	Range: 00..FF
Sub Class Interface Menu	16450 compatible serial controller	
Sub Class Value	00	Range: 00..FF
Interface Value	01	Range: 00..FF
Class Code	070001	Range: 000000..FFFFFF

For a description of these options, see the *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343).

PCIe BARs Tab

The PCIe BARs tab is shown in the following figure.

Figure 74: **PCIe BARs Tab**



Component Name: xdma_0

□ PCIe to AXI Lite Master Interface

□ 64bit Enable □ Prefetchable

Size: 1 Scale: Megabytes

Value: FFF00000

PCIe to AXI Translation: 0x00000000

☑ PCIe to DMA Interface

□ 64bit Enable □ Prefetchable

Size: 64 Scale: Kilobytes

☑ PCIe to DMA Bypass Interface

□ 64bit Enable □ Prefetchable

Size: 1 Scale: Megabytes

Value: FFF00000

PCIe to AXI Translation: 0x0000000000000000

- **Base Address Register Overview:** In Endpoint configuration, the core supports up to six 32-bit BARs or three 64-bit BARs, and the Expansion read-only memory (ROM) BAR. BARs can be one of two sizes:
  - **32-bit BARs:** The address space can be as small as 128 bytes or as large as 2 GB. Used for DMA, AXI Lite Master or AXI Bridge Master.
  - **64-bit BARs:** The address space can be as small as 128 bytes or as large as 8 Exabytes. Used for DMA, AXI Lite Master or AXI Bridge Master.

All BAR register share these options.

> **IMPORTANT!** *The DMA requires a large amount of space to support functions and queues. By default, 64-bit BAR space is selected for the DMA BAR. This applies for PF and VF bars. You must calculate your design needs first before selecting between 64-bit and 32-bit BAR space.*

BAR selections are configurable. By default DMA is at BAR 0 (64 bit), AXI-Lite Master is at BAR 2 (64 bit). These selections can be changed according to user needs.

- **BAR:** Click the checkbox to enable the BAR. Deselect the checkbox to disable the BAR.
- **Type:** Select from **DMA** (by default in BAR0), **AXI Lite Master** (by default in BAR1, if enabled), or **AXI Bridge Master** (by default in BAR2, if enabled). All other BARs, you can select between AXI List Master and AXI Bridge Master. Expansion ROM can be enabled by selecting BAR6

  For 64-bit BAR (default selection), **DMA** (by default in BAR0), **AXI Lite Master** (by default in BAR2, if enabled), and **AXI Bridge Master** (by default in BAR4, if enabled). Expansion ROM can be enabled by selection BAR6.

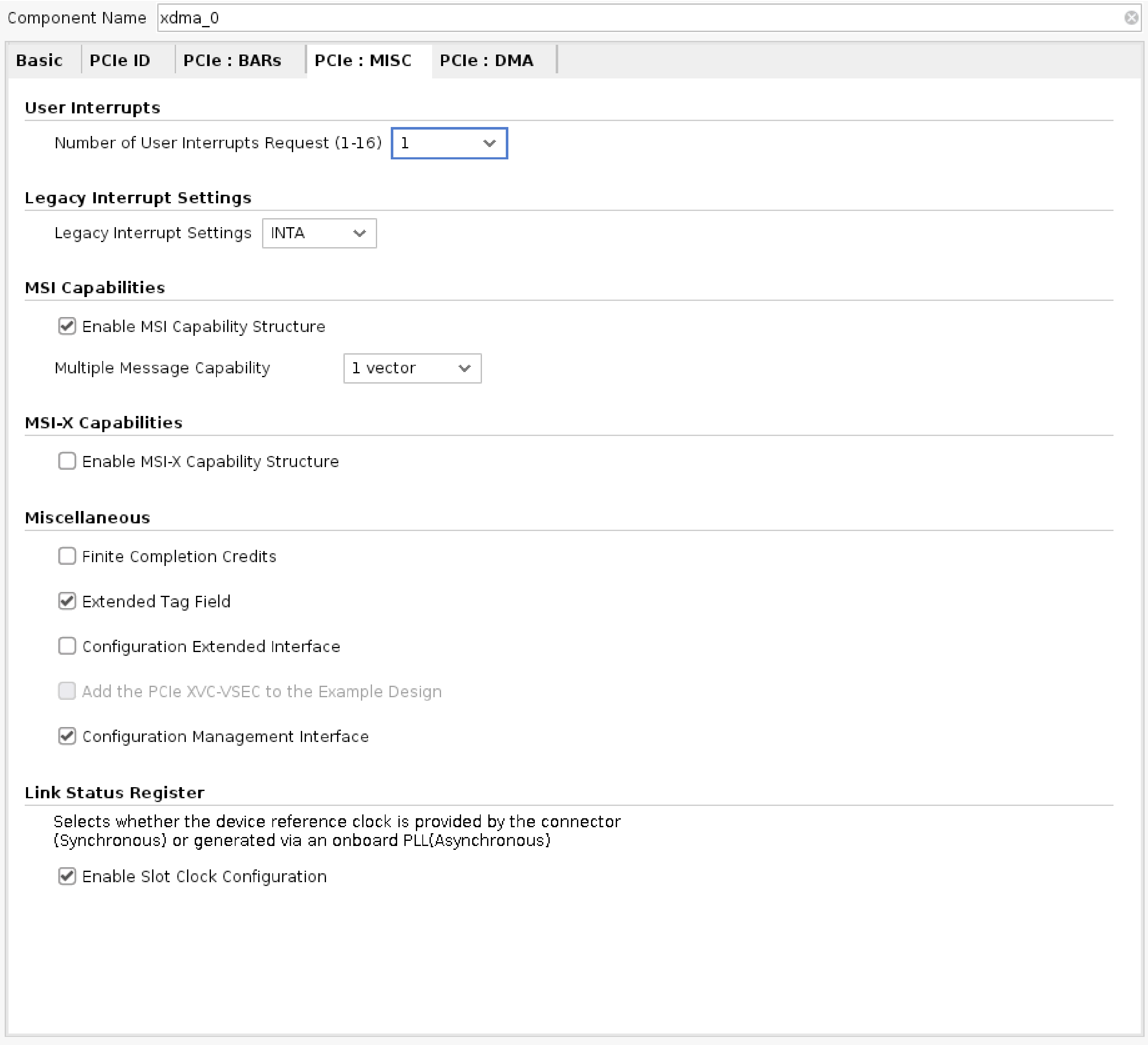
  - **DMA:** DMA by default is assigned to BAR0 space and for all PFs. DMA option can be selected in any available BAR (only one BAR can have DMA option). If you select **DMA Mailbox Management** rather than DMA; however, DMA Mailbox Management will not allow you to perform any DMA operations. After selecting the DMA Mailbox Management option, the host has access to the extended Mailbox space. For details about this space, see the QDMA_PF_MAILBOX (0x22400) register space.
  - **AXI Lite Master:** Select the AXI Lite Master interface option for any BAR space. The Size, scale and address translation are configurable.
  - **AXI Bridge Master:** Select the AXI Bridge Master interface option for any BAR space. The Size, scale and address translation are configurable.
  - **Expansion ROM:** When enabled, this space is accessible on the AXI4-Lite Master. This is a read-only space. The size, scale, and address translation are configurable.
- **Size:** The available Size range depends on the 32-bit or 64-bit bar selected. The DMA requires 256 KB of space, which is the fixed default selection. Other BAR size selections are available, but must be specified.
- **Scale:** Select between Byte, Kilobytes and Megabytes.
- **Value:** The value assigned to the BAR based on the current selections.

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

PCIe MISC Tab

The PCIe Miscellaneous tab is shown in the following figure.

Figure 75: **PCIe Misc Tab**



Component Name xdma_0

User Interrupts

Number of User Interrupts Request (1-16) 1

Legacy Interrupt Settings

Legacy Interrupt Settings INTA

MSI Capabilities

☑ Enable MSI Capability Structure

Multiple Message Capability 1 vector

MSI-X Capabilities

□ Enable MSI-X Capability Structure

Miscellaneous

□ Finite Completion Credits

☑ Extended Tag Field

□ Configuration Extended Interface

□ Add the PCIe XVC-VSEC to the Example Design

☑ Configuration Management Interface

Link Status Register

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

☑ Enable Slot Clock Configuration

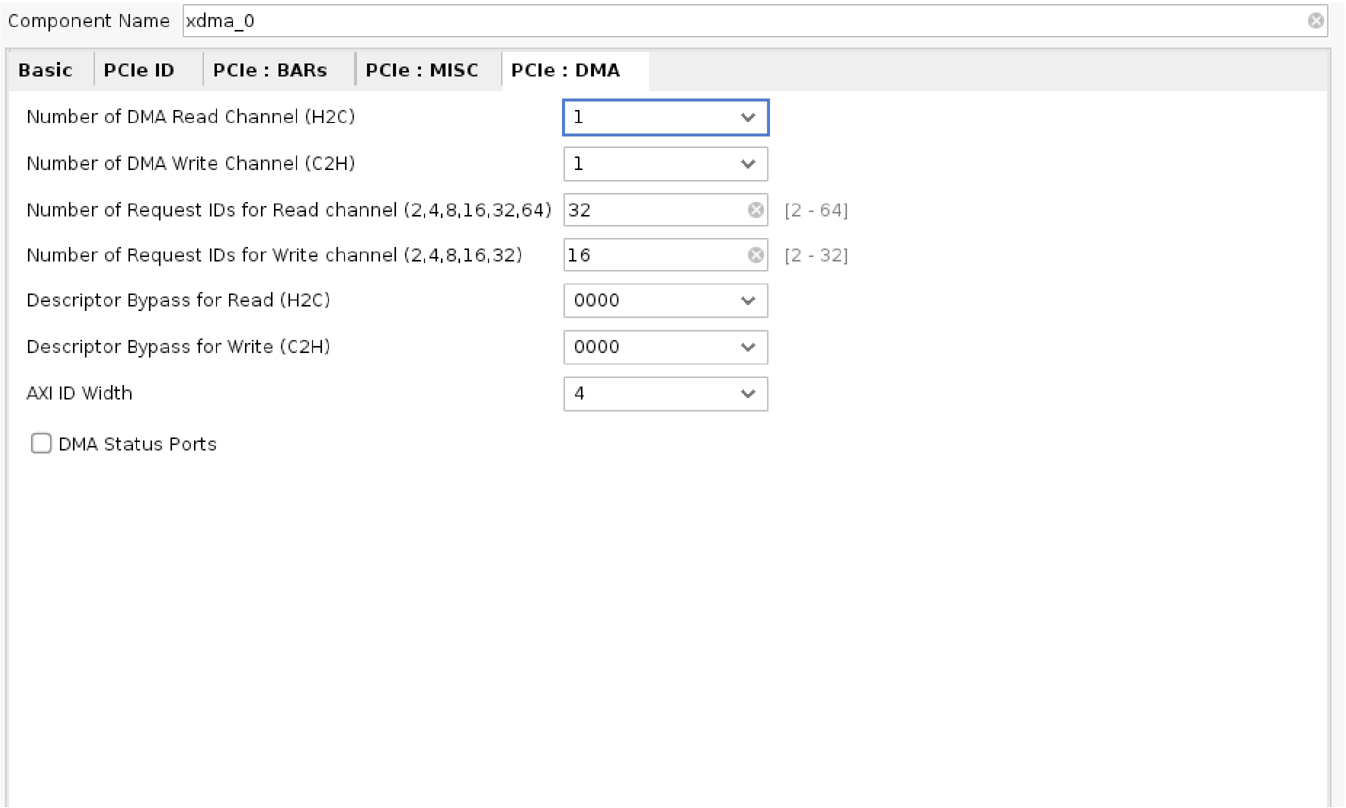
- **Number of User Interrupt Request:** Up to 16 user interrupt requests can be selected.
- **Legacy Interrupt Settings:** Select one of the Legacy Interrupts: INTA, INTB, INTC, or INTD.
- **MSI Capabilities:** By default, MSI Capabilities is enabled, and 1 vector is enabled. You can choose up to 32 vectors. In general, Linux uses only 1 vector for MSI. This option can be disabled.
- **MSI-X Capabilities:** Select a MSI-X event. For more information, see MSI-X Vector Table and PBA (0x8).

- **Finite Completion Credits:** On systems which support finite completion credits, this option can be enabled for better performance.
- **Extended Tag Field:** 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.
- **Link Status Register:** By default, Enable Slot Clock Configuration is selected. This means that the slot configuration bit is enabled in the link status register.

PCIe DMA Tab

The PCIe DMA tab is shown in the following figure.

Figure 76: **PCIe DMA Tab**



Component Name: xdma_0

Number of DMA Read Channel (H2C): 1

Number of DMA Write Channel (C2H): 1

Number of Request IDs for Read channel (2,4,8,16,32,64): 32 [2 - 64]

Number of Request IDs for Write channel (2,4,8,16,32): 16 [2 - 32]

Descriptor Bypass for Read (H2C): 0000

Descriptor Bypass for Write (C2H): 0000

AXI ID Width: 4

☐ DMA Status Ports

- **Number of DMA Read Channels:** Available selection is from 1 to 4.
- **Number of DMA 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.

Output Generation

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

Example Design

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

> **TIP:** *The PCIe reset pin for PL PCIE designs can be connected to any compatible single ended PL I/O pin location. If your board is compatible for either CPM4 or PL PCIE usage, you can use the CPM4 pin MIO38 to route the `sys_rst_n`. When this is done, the PL PCIE can use the reset as routed to the PL.*

Before opening the example design, set the following Tcl property to use the reset on the MIO38 pin:

```
set_property CONFIG.insert_cips {true} [get_ips pcie_versal_0]
```

Available Example Designs

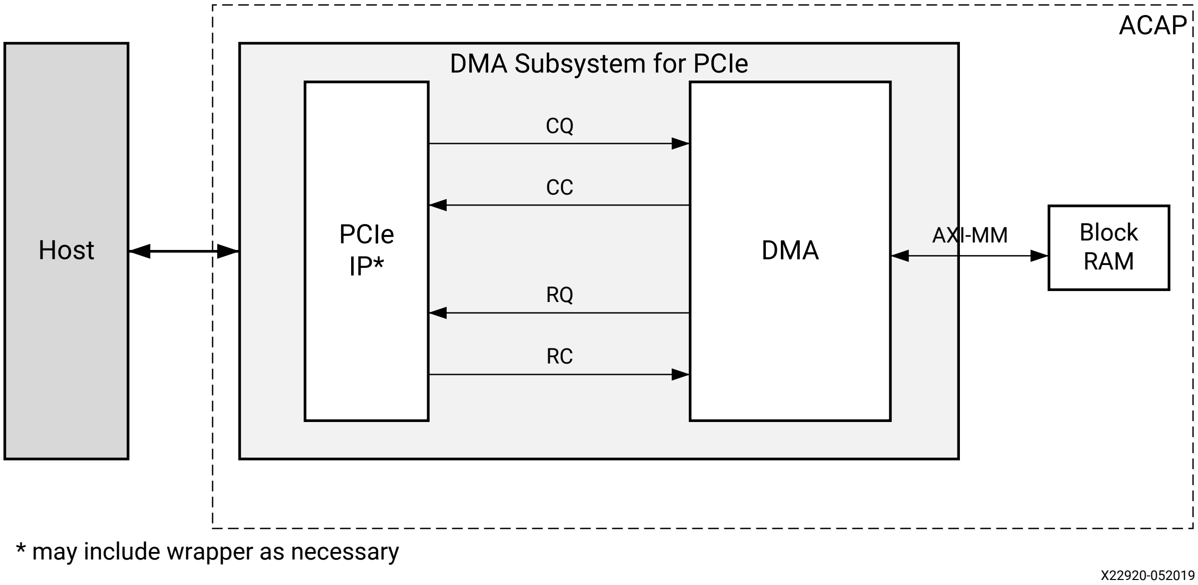
The example designs are as follows:

- 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
- 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 XDMA Subsystem reads data from host and writes to block RAM in the user side. For C2H transfers, the XDMA Subsystem 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 subsystem for larger block RAM size, if wanted.

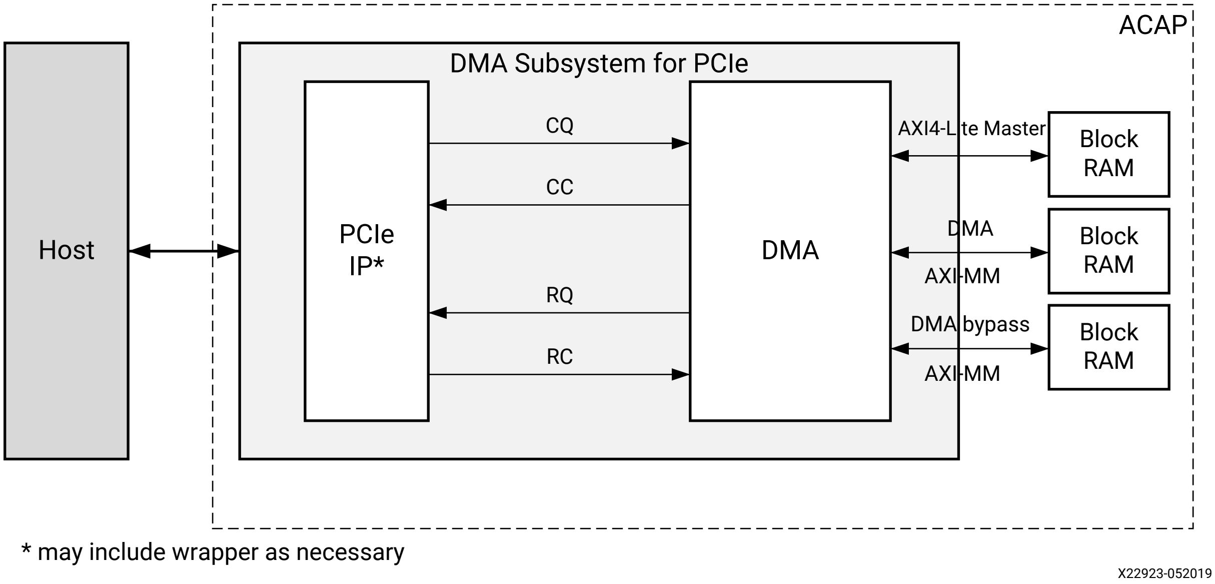
Figure 77: **AXI-MM Default Example Design**



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.

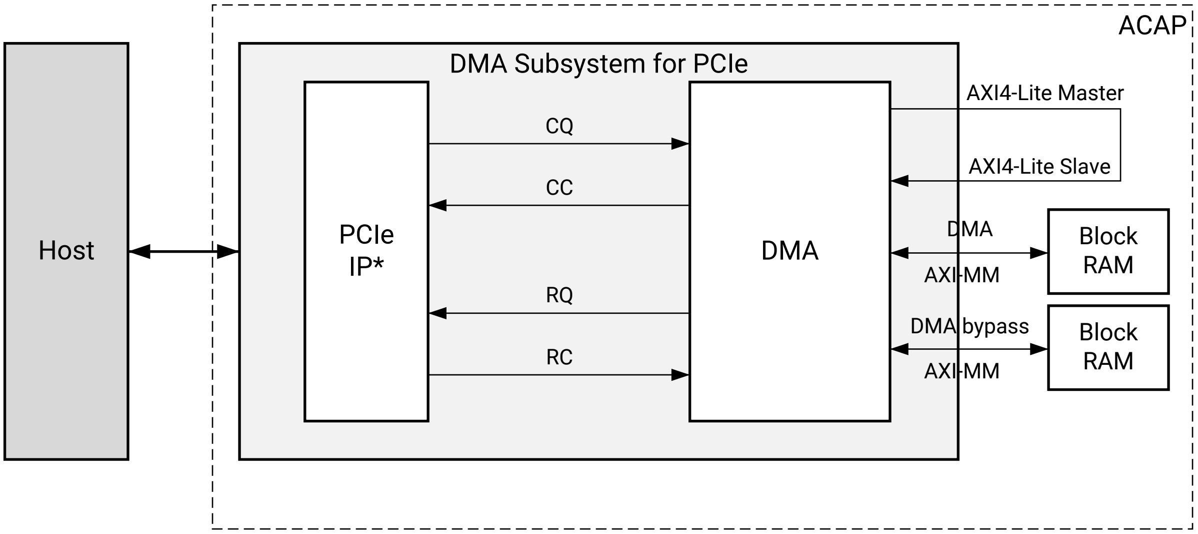
Figure 78: **AXI-MM Example with PCIe to DMA Bypass Interface and PCIe to AXI-Lite Master Enabled**



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 XDMA Subsystem registers. With this example design, the host can use PCIe to AXI4-Lite Master (BAR0 address space) and read/write XDMA Subsystem 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 79: **AXI-MM Example with AXI-Lite Slave Enabled**



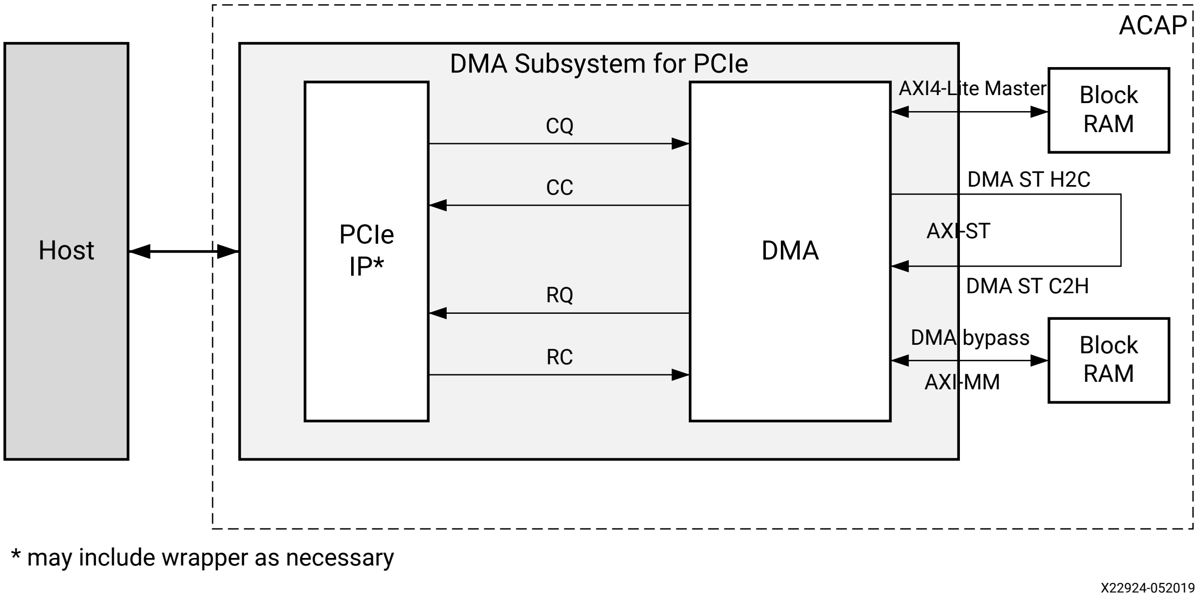
* may include wrapper as necessary

X22922-052019

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

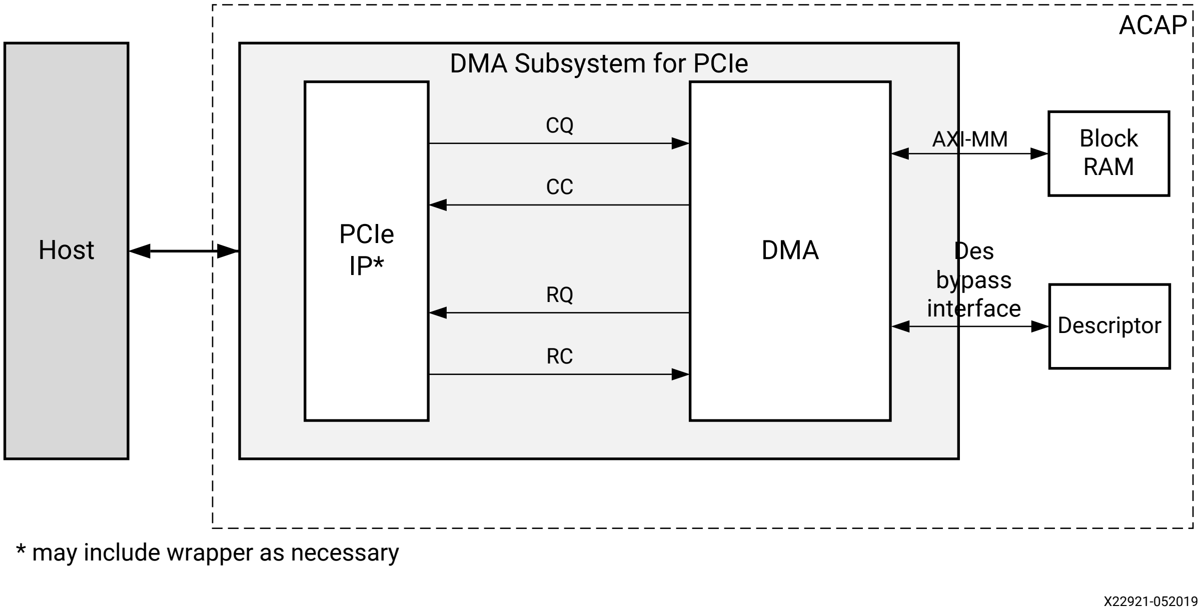


AXI4 Memory Mapped with Descriptor Bypass Example

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 64 bytes. 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 81: **AXI-MM Example With Descriptor Bypass Mode Enabled**



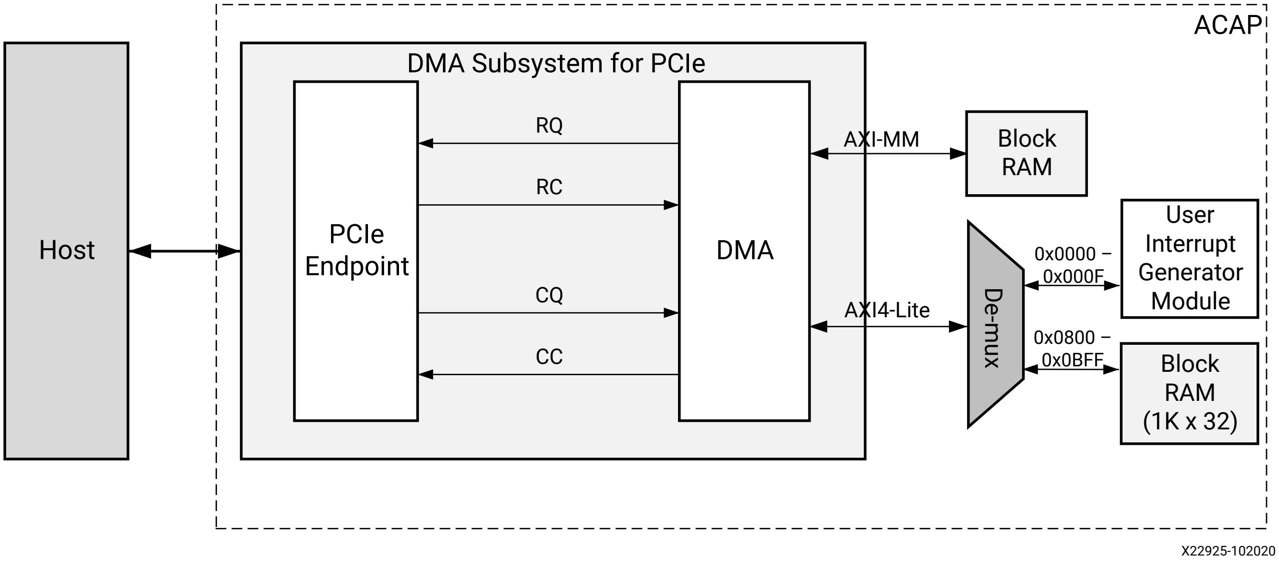
User IRQ Example Design

The user IRQ example design enables the host to connect to the AXI4-Lite Master interface along with the default XDMA Subsystem 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 82: **User IRQ Example Design**



The register description is found in the following table.

Table 279: **Example Design Registers**

Register Offset	Register Name	Access Type	Description
0x00	Scratch Pad	RW	Scratch Pad

Table 279: **Example Design Registers** *(cont'd)*

Register Offset	Register Name	Access Type	Description
0x04	DMA BRAM Size	RO	User Memory Size connected to XDMA. Memory size = (2[7:4]) ([3:0]Byte) [7:4] – denotes the size in powers of 2. 0 – 1 1 – 2 2 – 4 ... 8 – 256 9 – 512 [3:0] – denotes unit. 0 – Byte 1 – KB 2 – MB 3 – GB 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). Interrupt Status register corresponding bit must be 1 (ready) to generate interrupt. Also, reset the corresponding bit after ISR is served.
0x0C	Interrupt Status Register	RO	Interrupt Status. 1: ready 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.

Customizing and Generating the Example Design

In the Customize IP dialog box, use the default parameter values for the IP example design.

After reviewing the IP parameters:

1. Right-click the component name.
2. Select **Open IP Example Design**.

This opens a separate example design.

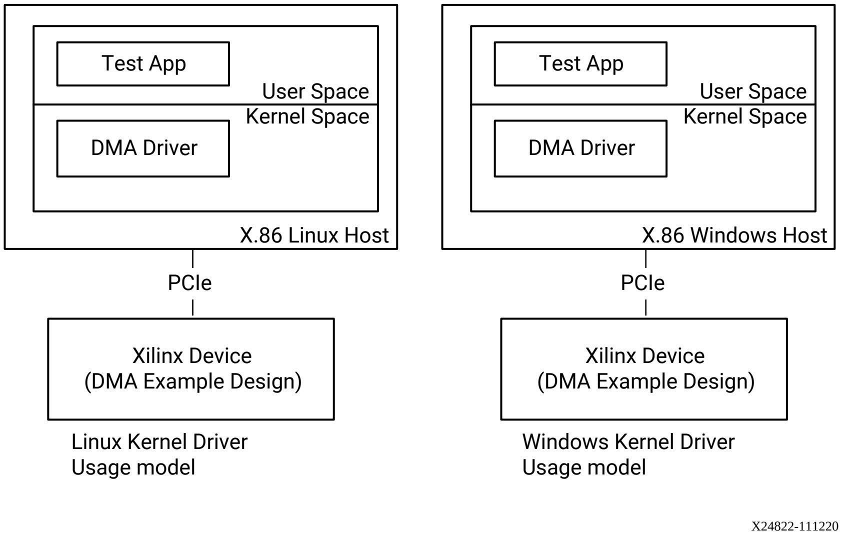
Chapter 16

Application Software Development

This section provides details about the Linux device driver and the Windows driver lounge that is provided with the core.

Device Drivers

Figure 83: **Device Drivers**



The above figure shows the usage model of Linux and Windows XDMA software drivers. The XDMA example design is implemented on a Xilinx® ACAP, which is connected to an X86 host through PCI Express.

- In the first use mode, the XDMA driver in kernel space runs on Linux, whereas the test application runs in user space.
- In the second use mode, the XDMA driver runs in kernel space on Windows, whereas the test application runs in the user space.

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 XDMA Subsystem 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 XDMA Subsystem 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.

Using the Driver

The XDMA drivers can be downloaded from the Xilinx DMA IP Drivers page.

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 the IRQ Block Channel Vector and the IRQ Block User Vector. 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 and in the IRQ Block User Vector 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. For MSI-X channel interrupt processing the driver should use the Engine's Interrupt Enable Mask for H2C and C2H 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 follows:

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.

Chapter 17

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 subsystem, 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 subsystem. 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.

Debug Guide

For more information on PCIe debug, see PCIe Debug K-Map.

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.

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 subsystem 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 Subsystem

AR 75397.

XDMA Debugging Answer Record

AR 000033516.

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.

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 XDMA

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.

Table 280: **Initial Debug of the Subsystem**

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 A

Upgrading

Migrating from Other Device Cores

For information on migrating from UltraScale+™ Devices DMA/Bridge Subsystems for PCIe to Versal ACAP DMA and Bridge Subsystem for PL PCIE4, see AR 000033502.

For information on migrating from UltraScale+™ Devices DMA/Bridge Subsystems for PCIe to Versal ACAP DMA and Bridge Subsystem for PL PCIE5, see AR 000033503.

Upgrading

This section is not applicable for the first release of the subsystem.

Appendix B

Limitations

Speed Change Related Issue

- **Description:** Repeated speed changes can result in the link not coming up to the intended targeted speed.
- **Workaround:** A follow-on attempt should bring the link back. In extremely rare scenarios, a full reboot might be required.

Link Autonomous Bandwidth Status (LABS) Bit

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

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

- **Workaround:** Ensure the software / application ignores the LABS bit as this is an informational bit and does not impact functionality.

  Note: For any application, Xilinx recommends that you make sure the link is quiesced and no transactions are pending before performing any link rate changes.

AXI Bridge

1. For this subsystem, the bridge master and bridge slave cannot achieve more than 128 Gb/s.
2. Bridge will be compliant with all MPS and MRRS settings; however, all traffic initiated from the Bridge will be limited to 256 Bytes (max)
3. AXI address width is limited to 48 bits.

PCIe Transaction Type

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

Table 281: **Supported PCIe Transaction Types**

TX	RX
MRd32	MRd32
MRd64	MRd64

Table 281: **Supported PCIe Transaction Types** *(cont'd)*

TX	RX
MWr32	MWr32
MWr64	MWr64
Msg	Msg
Cpl	Cpl
CplD	CplD
Cfg Type0/1 (For Root Port)	

AXI Slave

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

AXI Master

- Only issues the INCR burst type
- Only issues the data, non-secure, and unprivileged protection type

Related Information

QDMA Subsystem Limitations
AXI Bridge Subsystem Limitations
XDMA Subsystem Limitations

GT Selection and Pin Planning

This appendix provides guidance on gigabit transceiver (GT) selection for applicable Versal® devices and some key recommendations that should be considered when selecting the GT locations. This appendix provides guidance for CPM, PL PCIe® and PHY IP based solutions. In this guide, the IP related guidance is of primary importance, while the other related guidance might be relevant and is provided for informational purposes.

A GT Quad is comprised of four GT lanes. GT Quad and ref clock locations for CPM4 are in fixed locations depending on the desired link configuration (see GT Quad Locations). When selecting GT Quads for the PHY IP based solution with Xilinx PCIe MAC, Xilinx recommends that you use the GT Quads most adjacent to the Xilinx PCIe macro. While this is not required, it improves place, route, and timing for the design.

- Link widths of x1, x2, and x4 require one bonded GT Quad and should not split lanes between two GT Quads.
- A link width of x8 requires two adjacent GT Quads that are bonded and are in the same SLR.
- A link width of x16 requires four adjacent GT Quads that are bonded and are in the same SLR.
- PL PCIe blocks should use GTs adjacent to the PCIe block where possible.
- CPM has a fixed connectivity to GTs based on the CPM configuration.

For GTs on the **left side of the device**, PCIe lane 0 is placed in the bottom-most GT of the bottom-most GT Quad. Subsequent lanes use the next available GTs moving vertically up the device as the lane number increments. This means that the highest PCIe lane number uses the top-most GT in the top-most GT Quad that is used for PCIe.

For GTs on the **right side of the device**, PCIe lane 0 is placed in the top-most GT of the top-most GT Quad. Subsequent lanes use the next available GTs moving vertically down the device as the lane number increments. This means that the highest PCIe lane number uses the bottom-most GT in the bottom-most GT Quad that is used for PCIe.

The PCIe reference clock uses GTREFCLK0 in the PCIe lane 0 GT Quad for x1, x2, x4, and x8 configurations. For x16 configurations the PCIe reference clock should use GTREFCLK0 on a GT Quad associated with lanes 8-11. This allows the clock to be forwarded to all 16 PCIe lanes.

The PCIe reset pins for CPM designs must connect to one of specified pins for each of the two PCIe controllers. The PCIe reset pin for PL PCIe and PHY IP designs can be connected to any compatible PL pin location, or the CPM PCIe reset pins when the corresponding CPM PCIe controller is not in use. This is summarized in the following table:

Table 282: **PCIe Controller Reset Pin Locations**

Versal PCIe Controller	Versal Reset Pin Location
CPM PCIe Controller 0	PS MIO 18
	PMC MIO 24
	PMC MIO 38
CPM PCIe Controller 1	PS MIO 19
	PMC MIO 25
	PMC MIO 39
PL PCIe Controllers	Any compatible single-ended PL I/O pin.
Versal ACAP PHY IP	Any compatible single-ended PL I/O pin.

PCIe PHY IP has two Vivado tcl parameters. lane_reversal with values true or false {Default). lane_order with values Bottom (Default) or Top. For example in a x2 design, by default PIPE signals of the PCIe MAC[1:0] connects to PIPE signals of the GT QUAD[1:0]. When we apply lane_reversal {true} then PIPE signals of the PCIe MAC[1:0] connects to PIPE signals of the GT QUAD[0:1]. When we apply lane_order {Top} then PIPE signals of the PCIe MAC[1:0] connects to PIPE signals of the GT QUAD[3:2].

PL PCIe GT Selection

For PL PCIe blocks the most adjacent GTs should be used and connected to the PCIe solution IP where possible. The PL PCIe block supports x1, x2, x4, x8, and x16 link widths. This will provide the best place, route and timing result for the PCIe solution.

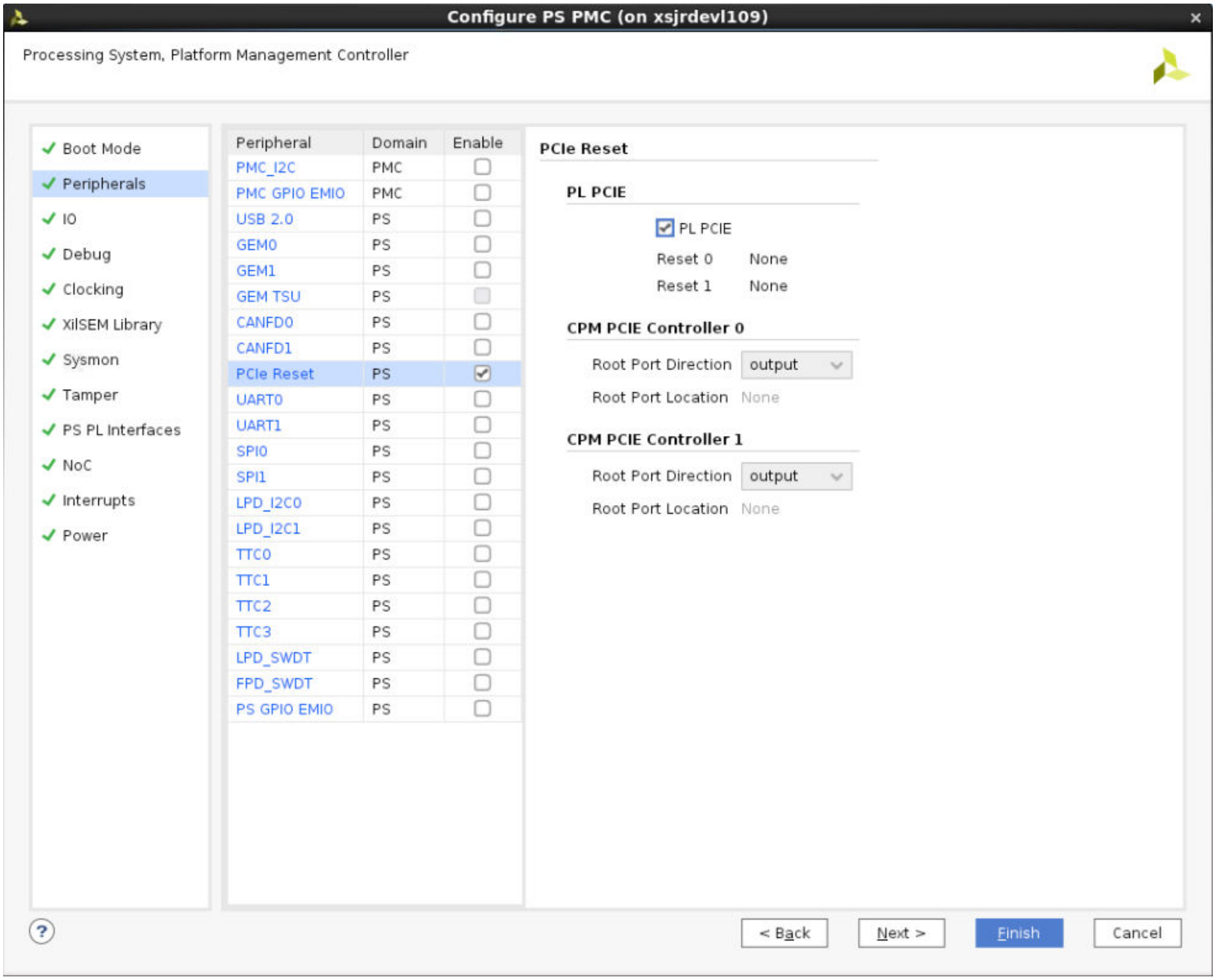
For GTs on the **left side of the device**, PCIe lane 0 is placed in the bottom-most GT of the bottom-most GT Quad. Subsequent lanes use the next available GTs moving vertically up the device as the lane number increments. This means that the highest PCIe lane number uses the top-most GT in the top-most GT Quad that is used for PCIe.

For GTs on the **right side of the device**, PCIe lane 0 is placed in the top-most GT of the top-most GT Quad. Subsequent lanes use the next available GTs moving vertically down the device as the lane number increments. This means that the highest PCIe lane number uses the bottom-most GT in the bottom-most GT Quad that is used for PCIe.

CPM4 Additional Considerations

To facilitate migration from UltraScale™ or UltraScale+™ designs, boards may be designed to use either CPM4 or PL PCIe integrated blocks to implement PCIe solutions. When designing a board to use either CPM4 or the PL PCIe hardblock, the CPM4 pin selection and planning guidelines should be followed because they are more restrictive. By doing this a board can be designed that will work for either a CPM4 or PL PCIe implementation. To route the PCIe reset from the CPM4 to the PL for use with a PL PCIe implementation the following option will need to be enabled under PS-PMC in the CIPS IP customization GUI.

Figure 84: **Configuring the PS PMC**



Peripheral	Domain	Enable
PMC_I2C	PMC	□
PMC GPIO EMIO	PMC	□
USB 2.0	PS	□
GEM0	PS	□
GEM1	PS	□
GEM TSU	PS	□
CANFD0	PS	□
CANFD1	PS	□
PCIe Reset	PS	☑
UART0	PS	□
UART1	PS	□
SPI0	PS	□
SPI1	PS	□
LPD_I2C0	PS	□
LPD_I2C1	PS	□
TTC0	PS	□
TTC1	PS	□
TTC2	PS	□
TTC3	PS	□
LPD_SWDT	PS	□
FPD_SWDT	PS	□
PS GPIO EMIO	PS	□

PCIe Reset

PL PCIE

☑ PL PCIE

Reset 0 None

Reset 1 None

CPM PCIE Controller 0

Root Port Direction output

Root Port Location None

CPM PCIE Controller 1

Root Port Direction output

Root Port Location None

When this option is enabled the PCIe reset for each disabled CPM4 PCIe controller will be routed to the PL. The same CPM4 pin selection limitations will apply and the additional PCIe reset output pins will be exposed at the boundary of the CIPS IP. If the CPM4 PCIe controller is enabled, the PCIe reset is used internal to the CPM4 and is not routed to the PL for connectivity to PL PCIe controllers.

GT Locations

Assigning GT Locations

Unlike in UltraScale+™ and previous devices implementations where direct assignment of GTs are not possible in the user constraints, in Versal ACAP implementations the GTs are external to the PCIe® IP and hence the GT assignment can be done in the user constraints.

Versal ACAP GT location assignment can be done in the user constraints file, IO planner, or hard block planner. For more information on how to assign GT locations in IO planner or hard block planner, see Synthesizing and Implementing the Design section in *Versal ACAP Transceivers Wizard LogiCORE IP Product Guide* (PG331). The GT locations are assigned in Quad granularity and not per lane. The lane 0 location and lane ordering are predetermined by whether the GTs are on the left or right side of the device. For details of lane 0 placement and lane ordering, see Appendix C: GT Selection and Pin Planning. Changing the lane 0 location or lane ordering is not supported other than the default setting described in GT Selection and Pin Planning. Following is an example of assigning GTYP locations in a user constraint file.

Note: The gt_quad instances should be assigned contiguously.

```
set_property LOC GTY_QUAD_X0Y6   [get_cells $gt_quads -filter NAME=~*/
gt_quad_3/*]
set_property LOC GTY_QUAD_X0Y5   [get_cells $gt_quads -filter NAME=~*/
gt_quad_2/*]
set_property LOC GTY_QUAD_X0Y4   [get_cells $gt_quads -filter NAME=~*/
gt_quad_1/*]
set_property LOC GTY_QUAD_X0Y3   [get_cells $gt_quads -filter NAME=~*/
gt_quad_0/*]
```

GT Quad Locations

The following table identifies the PCIe lane0 GT Quad(s) that can be used for each PCIe controller location. The Quad shown in bold is the most adjacent or suggested GT Quad for each PCIe lane0 location.

Table 283: **PCIE4 GTY Locations**

Device	Package	Left Side PCIE4 Blocks	Left Side Suggested GT QUAD	Right Side PCIE4 Blocks	Right Side Suggested GT QUAD
XCVC1902	VIVA1596	X0Y2	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	X1Y2	GTY_QUAD_X1Y6 **GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3
		X0Y1	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3	X1Y0	GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2**
XCVC1902	VSVA2197	X0Y2	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	X1Y2	GTY_QUAD_X1Y6 **GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3
		X0Y1	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3	X1Y0	GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2** GTY_QUAD_X1Y1 GTY_QUAD_X1Y0
XCVC1902	VSVD1760	X0Y2	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	X1Y2	**GTY_QUAD_X1Y4** GTY_QUAD_X1Y3
		X0Y1	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3	X1Y0	**GTY_QUAD_X1Y3**
XCVM1802	VFVC1760	X0Y2	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	X1Y2	GTY_QUAD_X1Y6 **GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3
		X0Y1	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3	X1Y0	GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2** GTY_QUAD_X1Y1 GTY_QUAD_X1Y0
XCVM1802	VSVA2197	X0Y2	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	X1Y2	GTY_QUAD_X1Y6 **GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3
		X0Y1	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	X1Y0	GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2** GTY_QUAD_X1Y1 GTY_QUAD_X1Y0
XCVM1802	VSVD1760	X0Y2	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	X1Y2	**GTY_QUAD_X1Y4** GTY_QUAD_X1Y3
		X0Y1	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3	X1Y0	**GTY_QUAD_X1Y3**

Table 283: **PCIE4 GTY Locations** *(cont'd)*

Device	Package	Left Side PCIE4 Blocks	Left Side Suggested GT QUAD	Right Side PCIE4 Blocks	Right Side Suggested GT QUAD
XCVE1752	VSVA2197	X0Y2	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	X1Y2	GTY_QUAD_X1Y6 **GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3
		X0Y1	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3	X1Y0	GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2** GTY_QUAD_X1Y1 GTY_QUAD_X1Y0
XCVC1702	VSVA2197	X0Y2	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	X1Y2	GTY_QUAD_X1Y6 **GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3
		X0Y1	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3	X1Y0	GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2** GTY_QUAD_X1Y1 GTY_QUAD_X1Y0
XCVC1802	VIVA1596	X0Y2	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	X1Y2	**GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3
		X0Y1	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3	X1Y0	GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2** GTY_QUAD_X1Y1
XCVC1802	VSVA2197	X0Y2	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	X1Y2	GTY_QUAD_X1Y6 **GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3
		X0Y1	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3	X1Y0	GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2** GTY_QUAD_X1Y1 GTY_QUAD_X1Y0
XCVC1802	VSVD1760	X0Y2	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	X1Y2	**GTY_QUAD_X1Y4** GTY_QUAD_X1Y3
		X0Y1	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3	X1Y0	**GTY_QUAD_X1Y3**
XCVM1502	VFVC1760	X0Y2	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	X1Y2	GTY_QUAD_X1Y6 **GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3
		X0Y1	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3	X1Y0	GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2** GTY_QUAD_X1Y1 GTY_QUAD_X1Y0

Table 283: **PCIE4 GTY Locations** *(cont'd)*

Device	Package	Left Side PCIE4 Blocks	Left Side Suggested GT QUAD	Right Side PCIE4 Blocks	Right Side Suggested GT QUAD
XCVM1402	VSVD1760	X0Y1	**GTY_QUAD_X0Y3** GTY_QUAD_X0Y2	N/A	N/A
				N/A	N/A
		X0Y0	GTY_QUAD_X0Y3 **GTY_QUAD_X0Y2** GTY_QUAD_X0Y1 GTY_QUAD_X0Y0	N/A	N/A
				N/A	N/A
XCVM1402	NBVB1024	X0Y1	**GTY_QUAD_X0Y3** GTY_QUAD_X0Y2	N/A	N/A
				N/A	N/A
		X0Y0	GTY_QUAD_X0Y3 **GTY_QUAD_X0Y2** GTY_QUAD_X0Y1 GTY_QUAD_X0Y0	N/A	N/A
				N/A	N/A
XCVM1302	VSVD1760	X0Y1	**GTY_QUAD_X0Y3** GTY_QUAD_X0Y2	N/A	N/A
				N/A	N/A
		X0Y0	GTY_QUAD_X0Y3 **GTY_QUAD_X0Y2** GTY_QUAD_X0Y1 GTY_QUAD_X0Y0	N/A	N/A
				N/A	N/A
XCV20	NBVB1024	X0Y1	**GTY_QUAD_X0Y3** GTY_QUAD_X0Y2	N/A	N/A
				N/A	N/A
		X0Y0	GTY_QUAD_X0Y3 **GTY_QUAD_X0Y2** GTY_QUAD_X0Y1 GTY_QUAD_X0Y0	N/A	N/A
				N/A	N/A

Appendix D

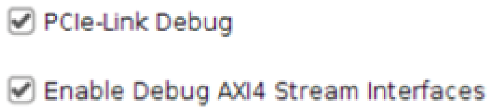
PCIe Link Debug Enablement

The customization provides an option to enable PCIe® Link Debug. Enabling this option will insert a debug core inside the IP core that will be recognized by the Vivado® Hardware Manager and provide PCIe specific debug information and view. The debug view provides information relating to the current link speed, current link width, and LTSSM state transitions, which can facilitate debug of PCIe link related issues.

Enabling PCIe Link Debug

Use this guide to enable and connect PCIe Link Debug in a Vivado IP integrator design. This section only describes the additional connections that should be added to enable PCIe Link Debug in a design. It does not discuss how to properly connect the PCIe enabled IPs to create a working design. Block automation can be used, or the connectivity and connections described below should be added to an existing design and IP configuration.

1. Enable this option in the core customization wizard, and select the options in the customization GUI, as shown below. The CPM PCIe cores are customized through the CIPS IP and for PL PCIe cores are customized through the Versal ACAP Integrated Block for PCIe IP.

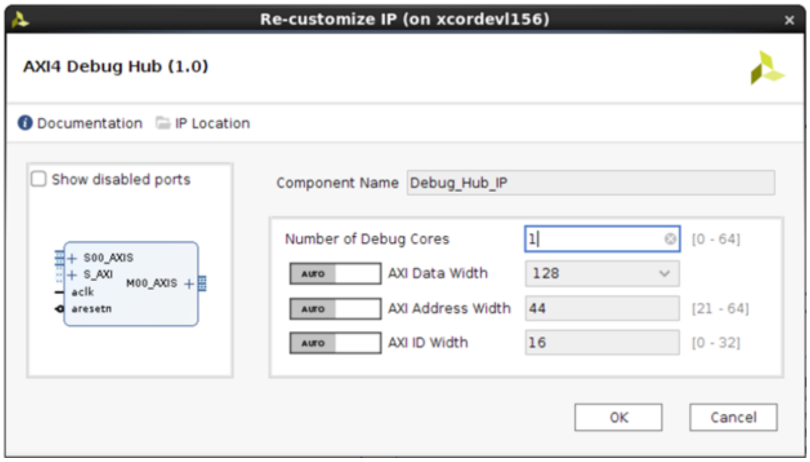
   

   ☑ PCIe-Link Debug

   ☑ Enable Debug AXI4 Stream Interfaces

   This adds the PCIe debug core to the PCIe IP and exposes the debug AXI4-Stream interfaces and ports. The debug AXI4-Stream and interface ports should be connected to a Debug Hub IP within the Versal design to enable debug for the design. The PCIe example design provides one implementation of how the Debug Hub IP can be connected in Versal designs. This is also detailed in the description below.

2. Add the Debug Hub IP to the design and use the following configuration options to enable the Debug Hub AXI Memory Mapped interface along with one set of AXI4-Stream interfaces. Additional AXI4-Stream interfaces can be enabled and connected in your design as desired.



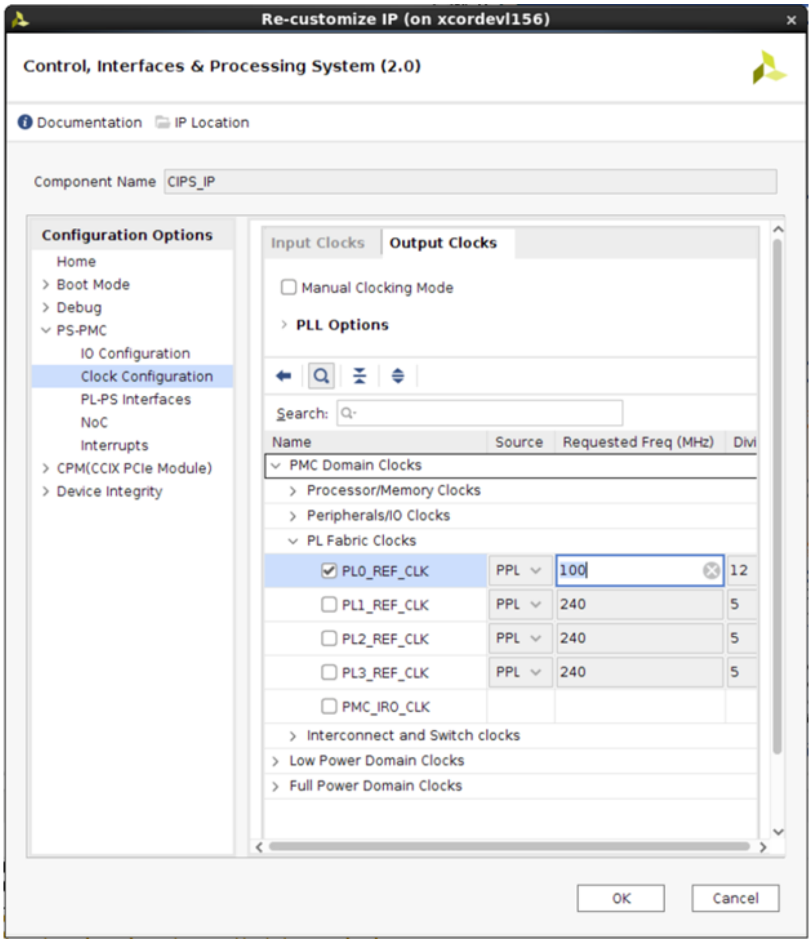
AXI4 Debug Hub (1.0)

Component Name: Debug_Hub_IP

Number of Debug Cores	1	[0 - 64]
AXI Data Width	128	
AXI Address Width	44	[21 - 64]
AXI ID Width	16	[0 - 32]

3. Add the CIPS IP to the design or configure the existing CIPS IP and include the following configuration options. These options will enable an AXI Master, clock, and reset that can be connected to the Debug Hub IP. To do so:

   a. Select **PS-PMC→ Clock Configuration→Output Clocks→PMC Domain Clocks→PL Fabric Clocks** selection enable a 100 MHz or similar output clock.

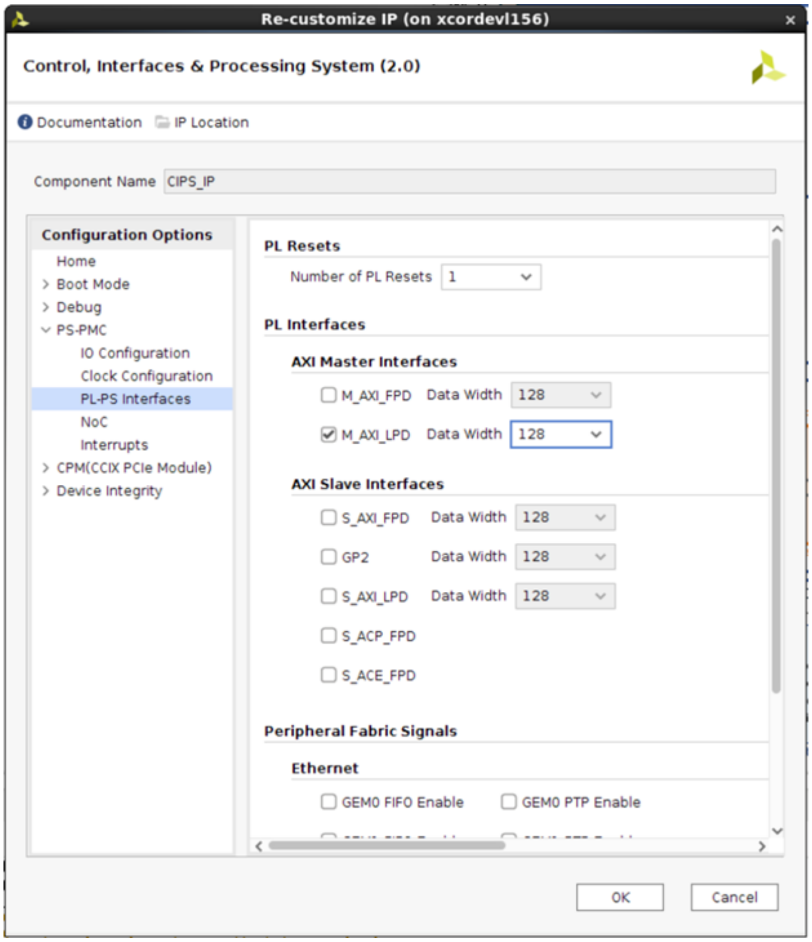


Control, Interfaces & Processing System (2.0)

Component Name: CIPS_IP

Name	Source	Requested Freq (MHz)	Divi
PMC Domain Clocks			
Processor/Memory Clocks			
Peripherals/IO Clocks			
PL Fabric Clocks			
☑ PL0_REF_CLK	PPL	100	12
□ PL1_REF_CLK	PPL	240	5
□ PL2_REF_CLK	PPL	240	5
□ PL3_REF_CLK	PPL	240	5
□ PMC_IRO_CLK			
Interconnect and Switch clocks			
Low Power Domain Clocks			
Full Power Domain Clocks			

b. Select **PS-PMC→PL-PS Interfaces**, and enable at least one PL reset in **Number of PL Resets**, and the M_AXI_LPD AXI master.



Re-customize IP (on xcordevl156)

Control, Interfaces & Processing System (2.0)

Component Name CIPS_IP

PL Resets

Number of PL Resets 1

PL Interfaces

AXI Master Interfaces

☐ M_AXI_FPD Data Width 128

☑ M_AXI_LPD Data Width 128

AXI Slave Interfaces

☐ S_AXI_FPD Data Width 128

☐ GP2 Data Width 128

☐ S_AXI_LPD Data Width 128

☐ S_ACP_FPD

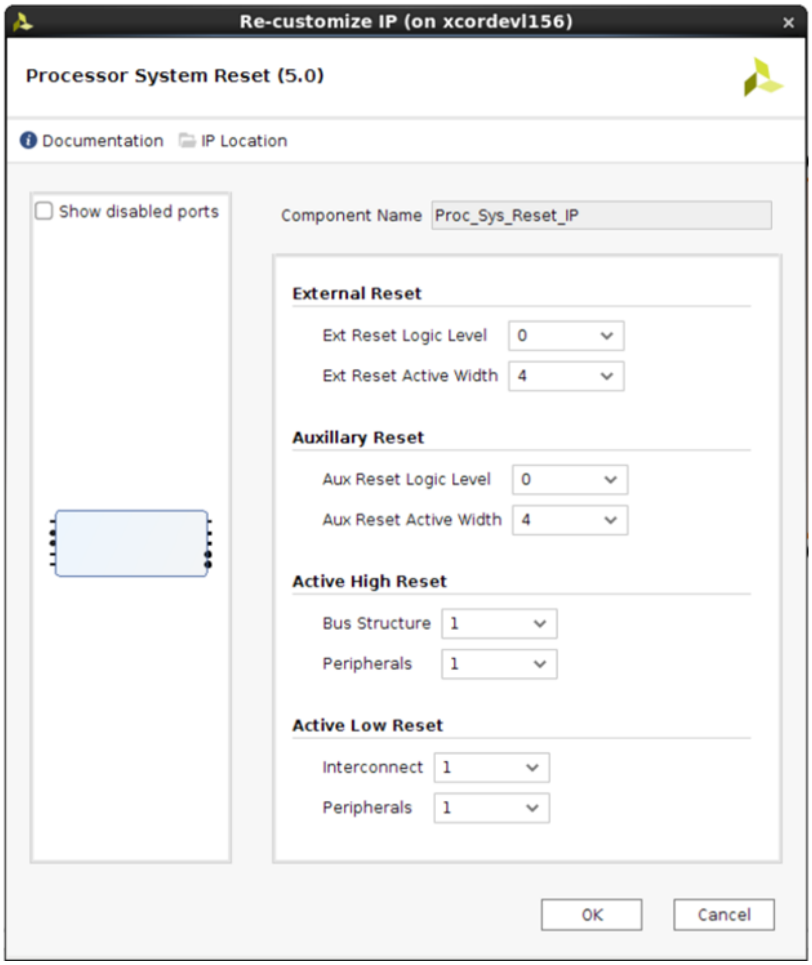
☐ S_ACE_FPD

Peripheral Fabric Signals

Ethernet

☐ GEM0 FIFO Enable ☐ GEM0 PTP Enable

4. Add and configure the Processor System Reset IP.


Re-customize IP (on xcordevl156)

Processor System Reset (5.0)

Documentation   IP Location

□ Show disabled ports

Component Name: Proc_Sys_Reset_IP

External Reset

Ext Reset Logic Level: 0

Ext Reset Active Width: 4

Auxillary Reset

Aux Reset Logic Level: 0

Aux Reset Active Width: 4

Active High Reset

Bus Structure: 1

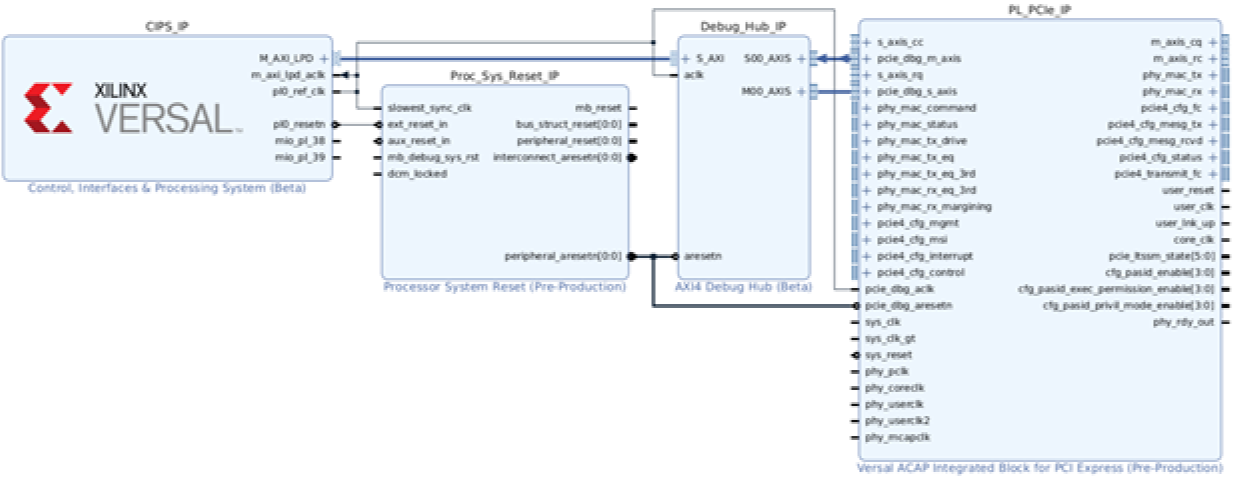
Peripherals: 1

Active Low Reset

Interconnect: 1

Peripherals: 1

5. Connect the IPs as shown in the following figures. This may need to be customized to fit with existing design connectivity.



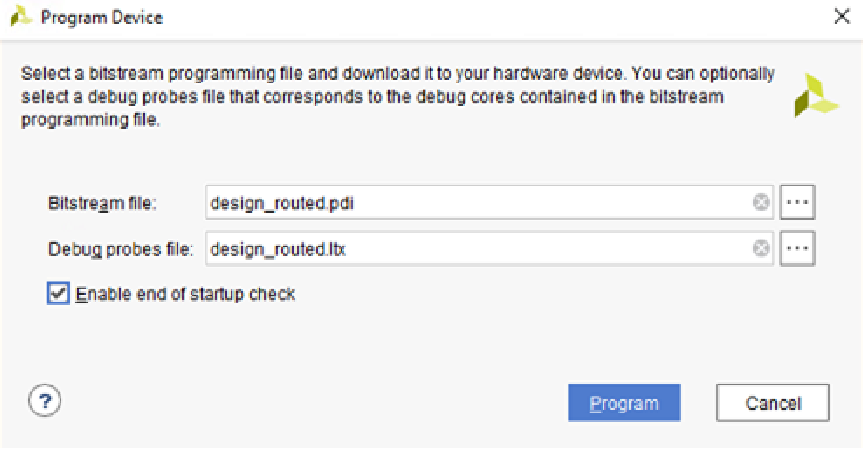
After the debug connections have been added to an Vivado IP integrator design, as shown above, PCIe Link debug is enabled in the generated `.pdi` image. The connections shown above should be added to a full design and are not sufficient to create a working design alone. The PCIe IP ports and the remainder of the design must be created and configured as per the desired operation of the PCIe-enabled IP.

Connecting to PCIe Link Debug in Vivado

Use the following steps to connect Vivado Hardware Manager to the FPGA device and associated PCIe Link Debug enabled design.

1. Open the Hardware Manager.
2. Select the device from the **Tools → Program Device...** drop-down menu.
3. Select the `.pdi` and `.ltx` files for programming the device, and select **Program**.

   Note: You should not load the `.ltx` file and refresh the target until after the `.pdi` file has been programmed.



Program Device

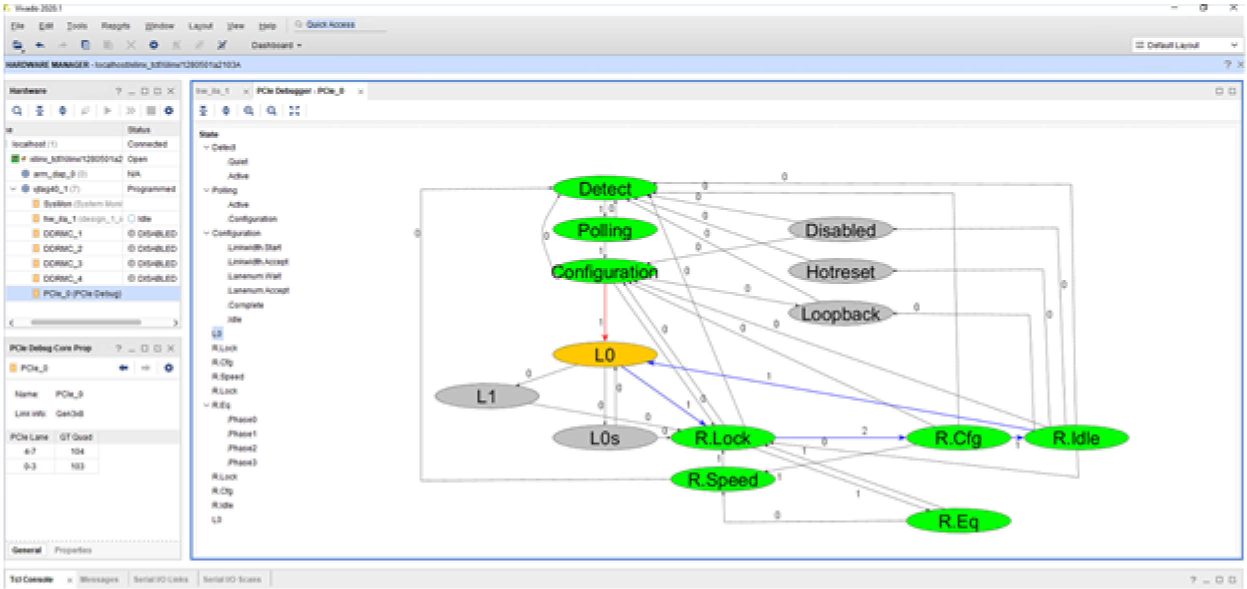
Select a bitstream programming file and download it to your hardware device. You can optionally select a debug probes file that corresponds to the debug cores contained in the bitstream programming file.

Bitstream file: design_routed.pdi

Debug probes file: design_routed.ltx

☑ Enable end of startup check

4. Select the PCIe Debug core in the Hardware window. You will see three main views that include the PCIe Debug Core Properties, PCIe Link LTSSM State Trace, and the PCIe Link LTSSM State Diagram with transitions.



Name: PCIe_0

Link Info: Gen3x8

PCIe Lane	GT Quad
4-7	104
0-3	103

Using this view, you can observe the active PCIe link status and state transitions. In the PCIe Debug Core Properties window, you can see the name of the PCIe debug core (PCIe_0), the current link status (Gen3x8), and the connected GTs (Quads 103 and 104). The PCIe LTSSM State Trace view shows a hierarchical view of the PCIe LTSSM state machine transitions. The PCIe LTSSM State Diagram provides a graphical display of the PCIe LTSSM states transitions that were traversed during the PCIe link up process. Visited LTSSM states are shown in green, the final or current LTSSM state is shown in yellow and the number of times each transition was traversed is identified on the arcs between states.

In addition to the graphical display, the `report_hw_pcie` command can be used to generate a console text report that contains the PCIe debug information. This information can be shared with others to aid in debugging PCIe Link issues. For this example, the name of the debug core is PCIe_0, and is inserted into the command.

```
report_hw_pcie PCIe_0
```

This information helps determine where in the PCIe link-up process an issue occurred and can guide further debug of link related issues.

Appendix 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. *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343)
2. *QDMA Subsystem for PCI Express Product Guide* (PG302)
3. *DMA/Bridge Subsystem for PCI Express Product Guide* (PG195)
4. *Versal ACAP CPM Mode for PCI Express Product Guide* (PG346)
5. *Versal ACAP CPM DMA and Bridge Mode for PCI Express Product Guide* (PG347)
6. *PCI-SIG Specifications* (https://www.pcisig.com/specifications)
7. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
8. *Vivado Design Suite User Guide: Designing with IP* (UG896)
9. *Vivado Design Suite User Guide: Getting Started* (UG910)
10. *Vivado Design Suite User Guide: Logic Simulation* (UG900)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
05/20/2022 Version 1.0	
AXI4-Stream C2H Ports	Updated AXI4-Stream C2H Port Descriptions table.
Function Map Table	Updated.
AXI4-Lite Master Ports	Updated Config AXI4-Lite Memory Mapped Read Master Interface Port Descriptions table.
AXI4-Stream C2H Ports	Updated AXI4-Stream C2H Port Descriptions table.
QDMA VF Address Register Space	Updated QDMA VF Address Register Space table.
Descriptor Bypass Mode	Updated.
04/26/2022 Version 1.0	
General updates	Updated for Versal Premium ACAP support.
Debug Guide	New section.
12/20/2021 Version 1.0	
Minimum Device Requirements	Added new section.
C2H Channel 0-3 AXI4-Stream Interface Signals H2C Channel 0-3 AXI4-Stream Interface Signals	Added m_axis_h2c_tkeep_*x*.

Section	Revision Summary
Basic Tab	Added recommendations regarding selecting the correct GT starting quad before lane rate and width.
QDMA Global Ports	Added csr_prog_done.
AXI4-Stream C2H Ports	Updated s_axis_c2h_ctrl_ecc[6:0].
User Interrupts	Update usr_irq_in_vec.
PCIe MISC Tab in Root Port Mode	New content added.
Debugging chapters	Added link to debugging answer record.
Appendix A: Upgrading	Updated link for migration information.
Appendix B: Limitations	Updated content, and moved to an appendix.
04/27/2021 Version 1.0	
Initial 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