Versal ACAP CPM DMA and Bridge Mode for PCI Express v2.1

Product Guide

Vivado Design Suite

PG347 (v2.1) May 4, 2021

XILINX®

Table of Contents

Section I

Overview

Navigating Content by Design Process

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

- **System and Solution Planning:** Identifying the components, performance, I/O, and data transfer requirements at a system level. Includes application mapping for the solution to PS, PL, and AI Engine. Topics in this document that apply to this design process include:
  - Introduction to the CPM4
  - Use Modes
- **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
    - Application Software Development
  - AXI Bridge Subsystem
    - Register Space
  - XDMA Subsystem
    - Register Space
    - Application Software Development

- **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
        - ◦ Application Software Development
    - AXI Bridge Subsystem
        - ◦ Register Space
    - XDMA Subsystem
        - ◦ Register Space
        - ◦ Application Software Development
- **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: QDMA AXI MM Interface to NoC and DDR Lab
    - XDMA Subsystem: XDMA AXI MM Interface to NoC and DDR Lab

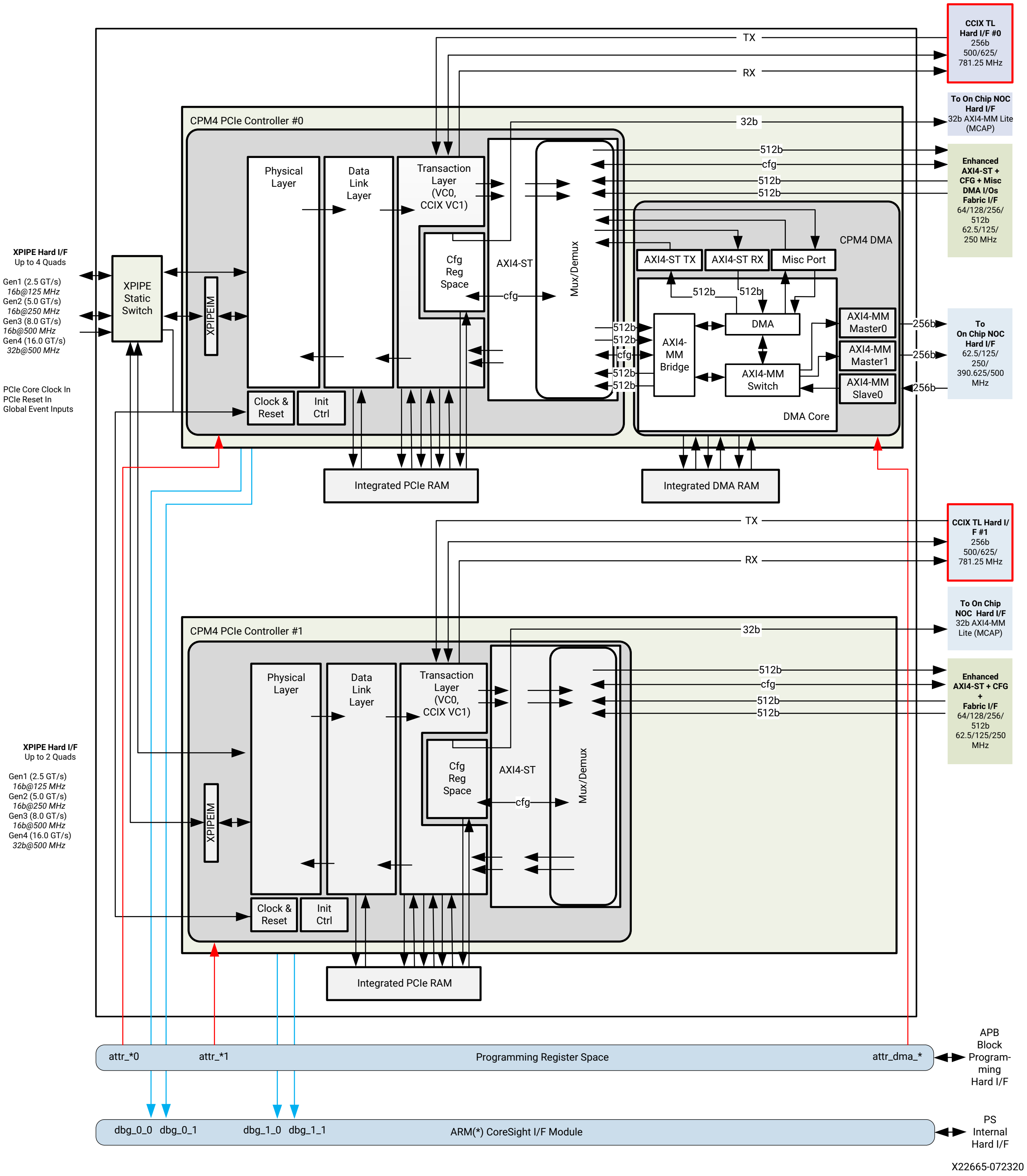
Chapter 1

Introduction

Introduction to the CPM4

The integrated block for PCIe Rev. 4.0 with DMA and CCIX Rev. 1.0 (CPM4) is shown in the following figure.

Figure 1: **CPM4 Sub-Block for PCIe Function (CPM4 PCIE)**



CPM Components

The CPM includes multiple IP cores:

- **Controllers for PCIe:** The CPM contains two instances of the Xilinx controller for PCIe: PCIE Controller 0 and PCIE Controller 1. Both controllers can have CCIX capabilities. However, only PCIE Controller 0 is capable of acting as an AXI bridge and as a DMA master. The controllers interface with the GTs through the XPIPE interface.

- **Coherent Mesh Network:** The CPM has a Coherent Mesh Network (CMN) (not shown) that forms the cache coherent interconnect block in the CPM that is based on the ARM CMN600 IP. There are two instances of L2 cache and CHI PL Interface (CPI) blocks in the CPM (also not shown).

- **DMA / AXI Bridge:** The CPM has two possible direct memory access (DMA) IP cores: DMA Subsystem for PCIe (XDMA) and Queue DMA Subsystem for PCIe (QDMA). The DMA cores are used for data transfer between the programmable logic (PL) to the host, and from the host to PL. The DMA cores can also transfer data between the host and the network on chip (NoC) which provides a high bandwidth to other NoC ports including the available DDR memory controllers (DDRMC). The CPM has an AXI Bridge Subsystem for PCIe (AXI Bridge) IP for AXI-to-host communication.

The CPM includes a clock/reset block that houses phase-locked loop (PLL) and clock dividers. The CPM also includes the system-on-a-chip (SoC) debug component for transaction-level debug. Several APB and AXI interfaces are used between blocks in the CPM for configuration.

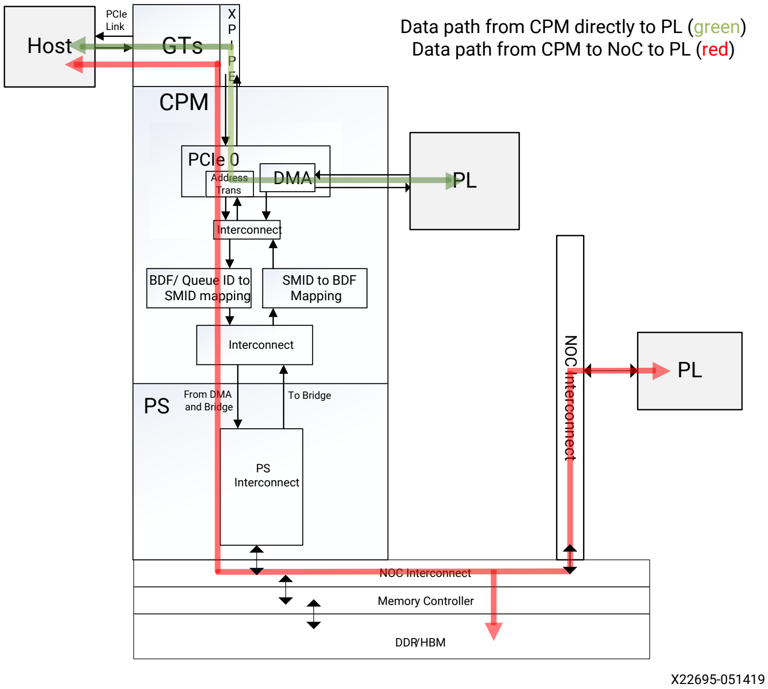
DMA Data Transfers

DMA transfers can be categorized into two different datapaths.

- **Data path from CPM to NoC to PL:** All AXI Memory Mapped signals are connected from the DMA to the AXI interconnect. These signals are then routed to the Non-Coherent interconnect in the CPM block. They then connect to the PS interconnect and the NoC. From the NoC, the signal can be directed to any block (DDR or block RAM) based on the user design. The figure below shows the datapath to NoC in red.

- **Data path from CPM directly to PL:** All AXI4-Stream signals and other side band signals, like clock and reset, are routed directly to the PL. The figure below shows the data path to the PL in green.

Figure 2: **DMA Data Paths**

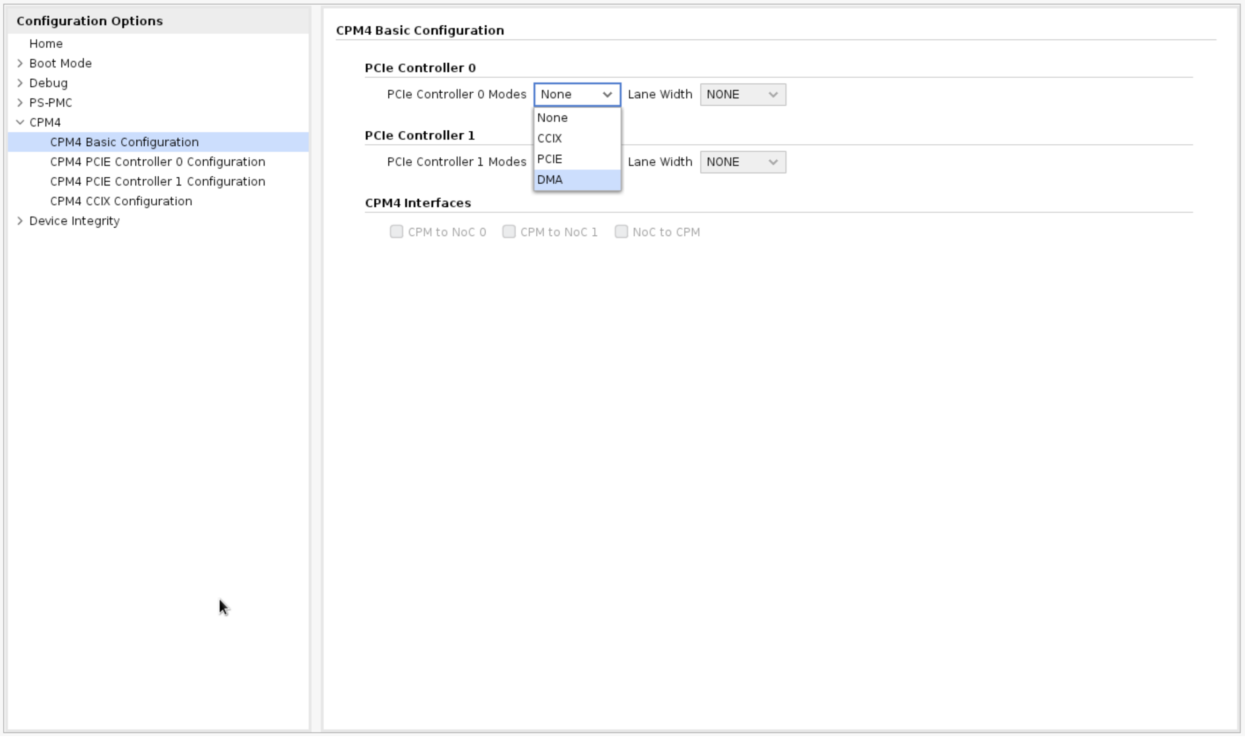


Use Modes

There are several use modes for DMA functionality in the CPM. You can select one of three options for data transport from host to programmable logic (PL), or PL to host: QDMA, AXI Bridge, and XDMA.

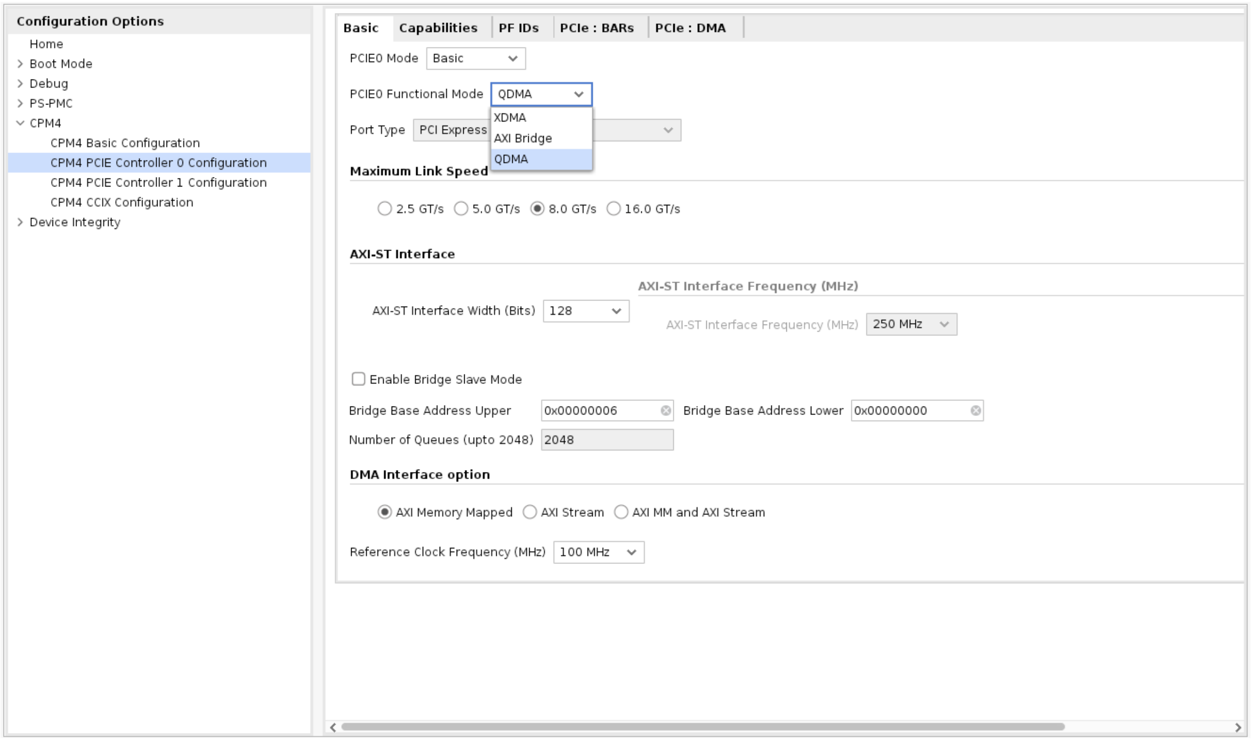
To enable DMA transfers, customize the Control, Interfaces and Processing System (CIPS) IP core as follows:

1. In the CPM4 Basic Configuration page, set the PCIe Controller 0 Mode to **DMA**.
2. Set the lane width value.



3. In the CPM4 PCIE Controller 0 Configuration page, set the PCIe Functional Mode for the desired DMA transfer mode:

   - **QDMA**
   - **AXI Bridge**
   - **XDMA**



PCIE0 Mode: Basic

PCIE0 Functional Mode: QDMA

Port Type: PCI Express

Maximum Link Speed

○ 2.5 GT/s ○ 5.0 GT/s ● 8.0 GT/s ○ 16.0 GT/s

AXI-ST Interface

AXI-ST Interface Width (Bits): 128

AXI-ST Interface Frequency (MHz)

AXI-ST Interface Frequency (MHz): 250 MHz

□ Enable Bridge Slave Mode

Bridge Base Address Upper: 0x00000006

Bridge Base Address Lower: 0x00000000

Number of Queues (upto 2048): 2048

DMA Interface option

● AXI Memory Mapped ○ AXI Stream ○ AXI MM and AXI Stream

Reference Clock Frequency (MHz): 100 MHz

The sections below explain how you can further configure and use these different functional modes for your application.

QDMA Functional Mode

QDMA mode enables the use of PCIE Controller 0 with QDMA enabled. QDMA mode provides two connectivity variants: AXI Streaming, and AXI Memory Mapped. Both variants can be enabled simultaneously.

- **AXI Streaming:** QDMA Streaming mode can be used in applications where the nature of the data traffic is streaming with source and destination IDs instead of a specific memory address location, such as network accelerators, network quality of service managers, or firewalls.
- **AXI Memory Mapped:** QDMA Memory Mapped mode can be used in applications where the nature of the data traffic is addressable memory, such as moving data between a host and a card, such as an acceleration platform.

The main difference between XDMA mode and QDMA mode is that while XDMA mode supports up to 4 independent data streams, QDMA mode can support up to 2048 independent data streams. Based on this strength, QDMA mode is typically used for applications that require many queues or data streams that need to be virtually independent from each other. QDMA mode is the only DMA mode that can support multiple functions, either physical functions or single root I/O virtualization (SR-IOV) virtual functions.

QDMA mode can be used in conjunction with AXI Bridge mode. For more details on AXI Bridge mode, which is described in the next section.

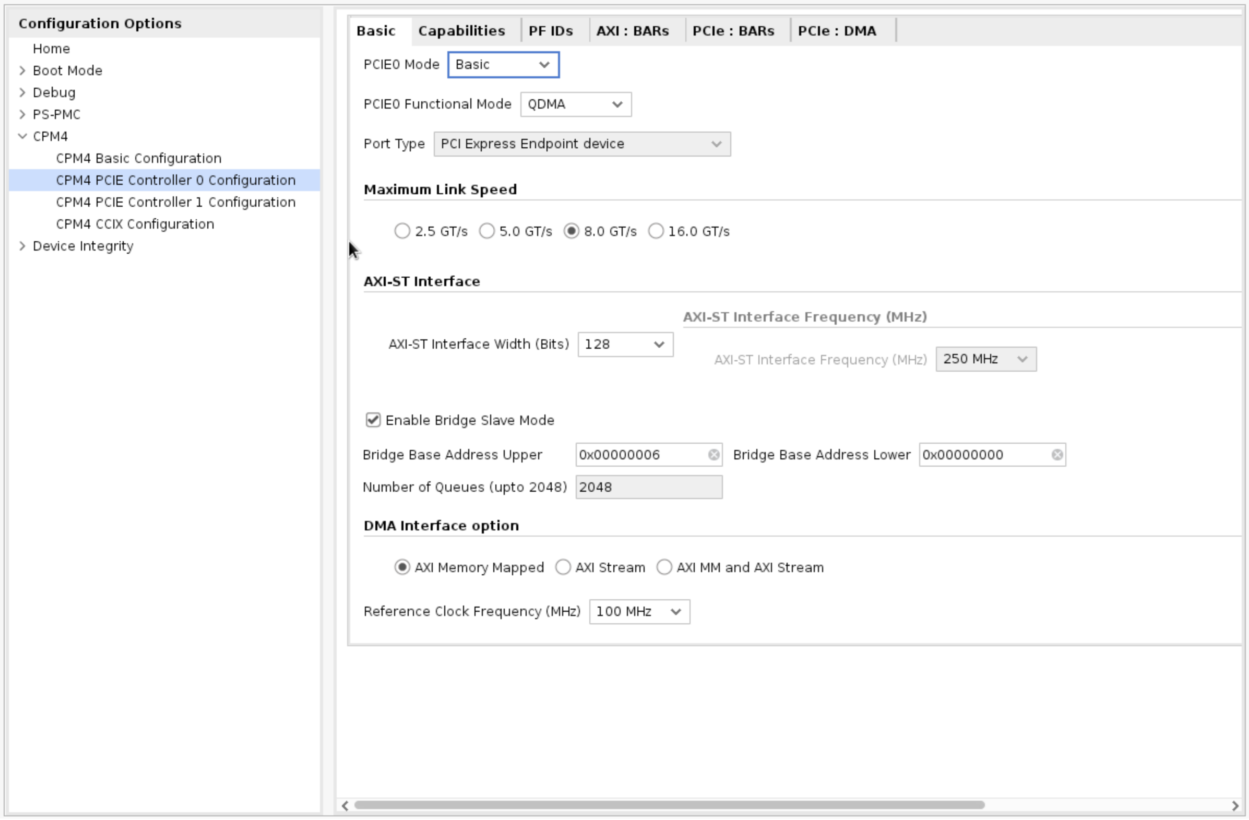
AXI Bridge Functional Mode

AXI Bridge mode enables you to interface the CPM4 PCIE Controller 0 with an AXI4 Memory Mapped domain. This use mode connects directly to the NoC which allows communication with other peripherals within the Processing System (PS) and in the Programmable Logic (PL).

AXI Bridge mode is typically used for light traffic data paths such as write to or read from Control and Status registers. AXI Bridge mode is also the only mode that can be configured for Root Port application with AXI4 Memory Mapped interface used to interface with processor, typically the PS.

AXI Bridge mode is also available in conjunction with XDMA mode or QDMA mode. To use either of these DMA modes with AXI Bridge mode, customize the core as follows:

1. In the Basic tab, set PCIE0 Functional Mode to either **XDMA** or **QDMA**.
2. Set one or both of the following options:
    - In the Basic tab, select the **Enable Bridge Slave Mode** checkbox. This option enables Slave AXI interface within the IP which you can use to generate Write or Read transaction from an AXI source peripheral to other PCIe devices.



PCIE0 Mode: Basic

PCIE0 Functional Mode: QDMA

Port Type: PCI Express Endpoint device

Maximum Link Speed

○ 2.5 GT/s ○ 5.0 GT/s ● 8.0 GT/s ○ 16.0 GT/s

AXI-ST Interface

AXI-ST Interface Width (Bits): 128

AXI-ST Interface Frequency (MHz)

AXI-ST Interface Frequency (MHz): 250 MHz

☑ Enable Bridge Slave Mode

Bridge Base Address Upper: 0x00000006 Bridge Base Address Lower: 0x00000000

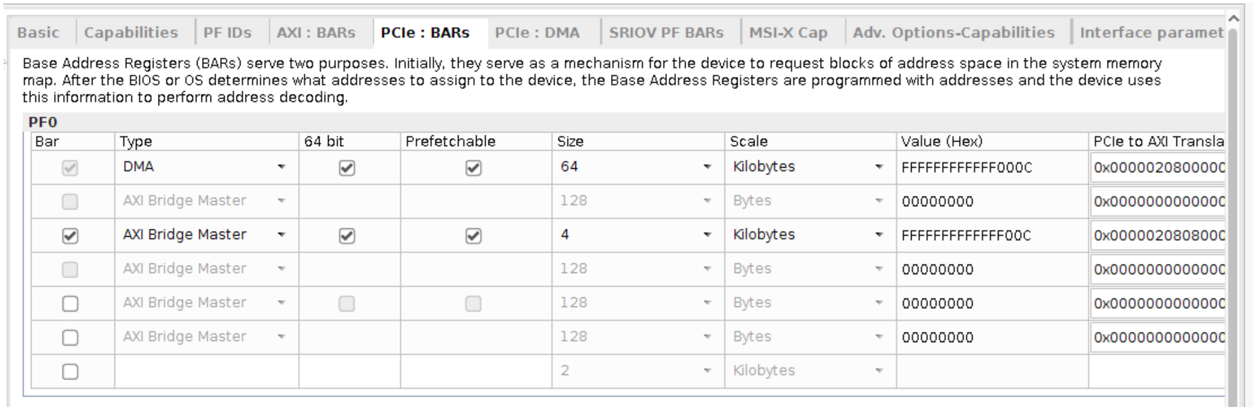
Number of Queues (upto 2048): 2048

DMA Interface option

● AXI Memory Mapped ○ AXI Stream ○ AXI MM and AXI Stream

Reference Clock Frequency (MHz): 100 MHz

    - In the PCIe: BARs tab, select the **BAR** checkbox next to AXI Bridge Master. This option enables the Master AXI interface within the IP which you can use to receive write or read transaction from a PCIe source device to AXI peripherals.



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 Transla
☑	DMA	☑	☑	64	Kilobytes	FFFFFFFFFFFF000C	0x000002080000C
□	AXI Bridge Master			128	Bytes	00000000	0x0000000000000
☑	AXI Bridge Master	☑	☑	4	Kilobytes	FFFFFFFFFFFFF00C	0x000002080800C
□	AXI Bridge Master			128	Bytes	00000000	0x0000000000000
□	AXI Bridge Master	□	□	128	Bytes	00000000	0x0000000000000
□	AXI Bridge Master			128	Bytes	00000000	0x0000000000000
□				2	Kilobytes		

XDMA Functional Mode

XDMA mode enables use of PCIE Controller 0 with the XMDA enabled. XDMA mode provides two connectivity variants: AXI Streaming, and AXI Memory Mapped. Only one variant can be enabled at a time.

- **AXI Streaming:** XDMA Streaming mode can be used in applications where the nature of the data traffic is streaming with source and destination IDs instead of a specific memory address location, such as network accelerators, network quality of service managers, or firewalls.
- **AXI Memory Mapped:** XDMA Memory mapped mode can be used in applications where the nature of the data traffic is addressable memory, such as moving data between a host and a card, such as an acceleration platform.

XDMA mode can be used in conjunction with AXI Bridge mode. For more details on AXI Bridge mode, see the AXI Bridge Functional Mode.

CPM4 Common Features

- Supports 64, 128, 256, and 512-bit data path.
- Supports x1, x2, x4, x8, or x16 link widths.
- Supports Gen1, Gen2, Gen3, and Gen4 link speeds.

***Note*:** x16 Gen4 configuration is not available in the data path from CPM directly to PL. This is only used with the CPM through AXI MM to NoC to PL data path.

QDMA Functional Mode

- 2048 queue sets
  - 2048 H2C descriptor rings.
  - 2048 C2H descriptor rings.
  - 2048 C2H Completion (CMPT) rings.

- Supports both the AXI4 Memory Mapped and AXI4-Stream interfaces per queue (AXI4-Stream not available when CPM4 configured for 16 GT/s data rate with x16 lane width).
- Supports Polling Mode (Status Descriptor Write Back) and Interrupt Mode.
- Interrupts
    - 2048 MSI-X vectors.
    - Up to 32 MSI-X vectors per PF, and 8 MSI-X vectors per VF.
    - 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.

AXI Bridge Functional Mode

AXI Bridge functional mode features are supported when AXI4 slave bridge is enabled in the XDMA or QDMA use mode.

- Supports Multiple Vector Messaged Signaled Interrupts (MSI), MSI-X interrupt, and Legacy interrupt.
- AXI4-MM Slave access to PCIe address space.
- PCIe access to AXI4-MM Master.
- Tracks and manages Transaction Layer Packets (TLPs) completion processing.
- Detects and indicates error conditions with interrupts in Root Port mode.
- Supports a single PCIe 32-bit or three 64-bit PCIe Base Address Registers (BARs) as Endpoint.
- Supports up to two PCIe 32-bit or a single PCIe 64-bit BAR as Root Port.

XDMA Functional Mode

- 64-bit source, destination, and descriptor addresses.
- Up to four host-to-card (H2C/Read) data channels.
- Up to four card-to-host (C2H/Write) data channels.
- Selectable user interface.
    - Single AXI4 memory mapped (MM) user interface.
    - AXI4-Stream user interface (each channel has its own AXI4-Stream interface; AXI4-Stream is not available when CPM4 configured for 16 GT/s data rate with x16 lane width).
- AXI4 Bridge Master interface allows for PCIe traffic to bypass the DMA engine.
- AXI4-Lite Slave interface allows access to DMA status registers.
- Scatter Gather descriptor list supporting unlimited list size.
- 256 MB max transfer size per descriptor.
- Legacy, MSI, and MSI-X interrupts.
- Block fetches of contiguous descriptors.
- Poll Mode.
- Descriptor Bypass interface.
- Arbitrary source and destination address.
- Parity check or Propagate Parity on DMA AXI interface.

Standards

The Versal ACAP CPM DMA and Bridge Mode for PCI Express adheres to the following standards:

- *AMBA AXI4-Stream Protocol Specification* (ARM IHI 0051A)
- PCI Express Base Specification v4.0 Version 1.0, and Errata updates
- PCI Local Bus Specification
- PCI-SIG® Single Root I/O Virtualization and Sharing (SR-IOV) Specification

For details, see *PCI-SIG Specifications* (https://www.pcisig.com/specifications).

Limitations

Speed Change Related Issue

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

Link Autonomous Bandwidth Status (LABS) Bit

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

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

- **Workaround:** None available.

Licensing and Ordering

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

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

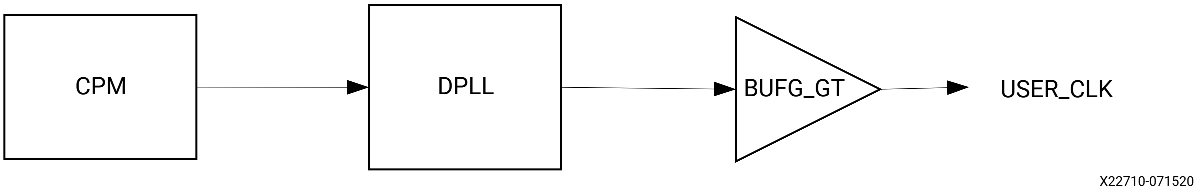
Designing with the Core

Clocking

Note**:** USER_CLK (`user_clk`) in this section refers to `pcie(n)_user_clk`, which is also described in the Clock and Reset Interface section.

The CPM 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 3: **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. The `user_clk` should be used to interface with the CPM. With the user logic, any available clocks can be used.

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 4: **Embedded System Using 100 MHz Reference Clock**

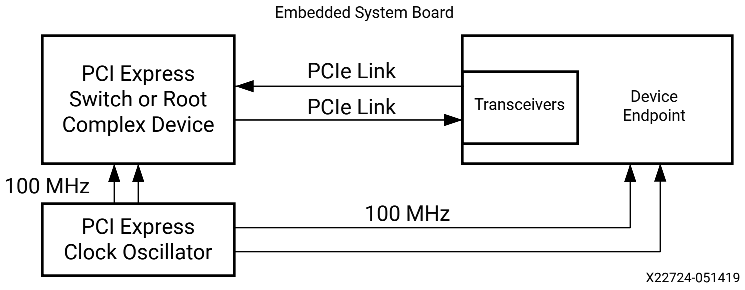
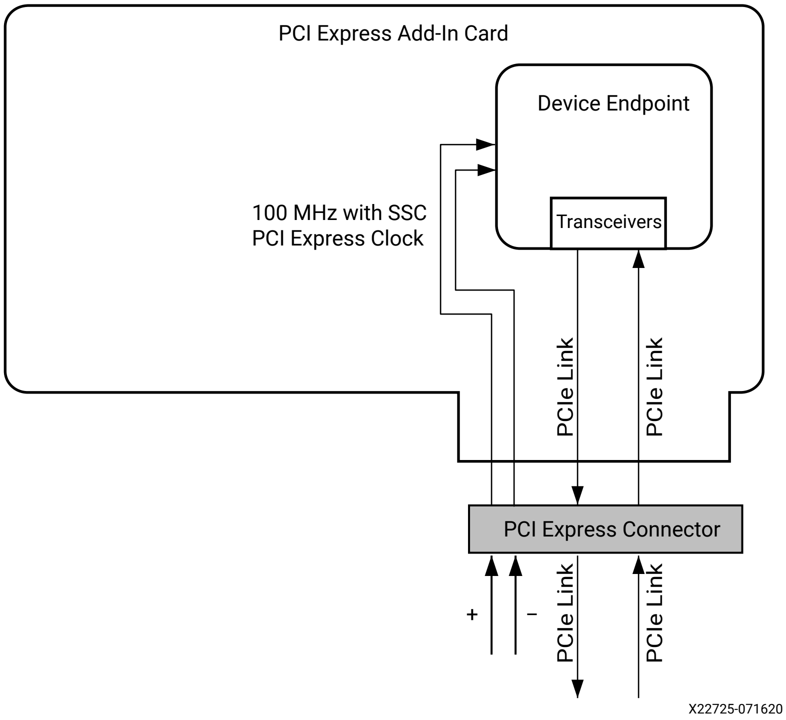


Figure 5: **Open System Add-In Card Using 100 MHz Reference Clock**



Resets

The fundamental resets for the CPM PCIe controllers and associated GTs are `perst0n` and `perst1n`. The resets are driven by the I/O inside the PS. In addition, there is a power-on-reset for CPM driven by the platform management controller (PMC). When both PS and the power-on reset from PMC are released, CPM PCIe controllers and the associated GTs will come out of reset.

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

In addition, there is a `pcie(n)_user_reset` given from the CPM PCIe controller to the user design present in the fabric logic. Whenever CPM PCIe controller goes through a reset, or there is a link down, the CPM PCIe controller issues a `pcie(n)_user_reset` to the user design in the programmable logic (PL) region. After the PCIe link is up, `pcie(n)_user_reset` is released for the user design to come out of reset.

To reset the DMA block, deassert the `dma_soft_resetn` pin. This pin is active-Low, and by default should be tied High. This will not reset the entire CPM PCIe controller but will reset only the DMA (XDMA/QDMA/AXI Bridge) block.

Section II

QDMA Subsystem

Overview

The Queue 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 functional mode 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.

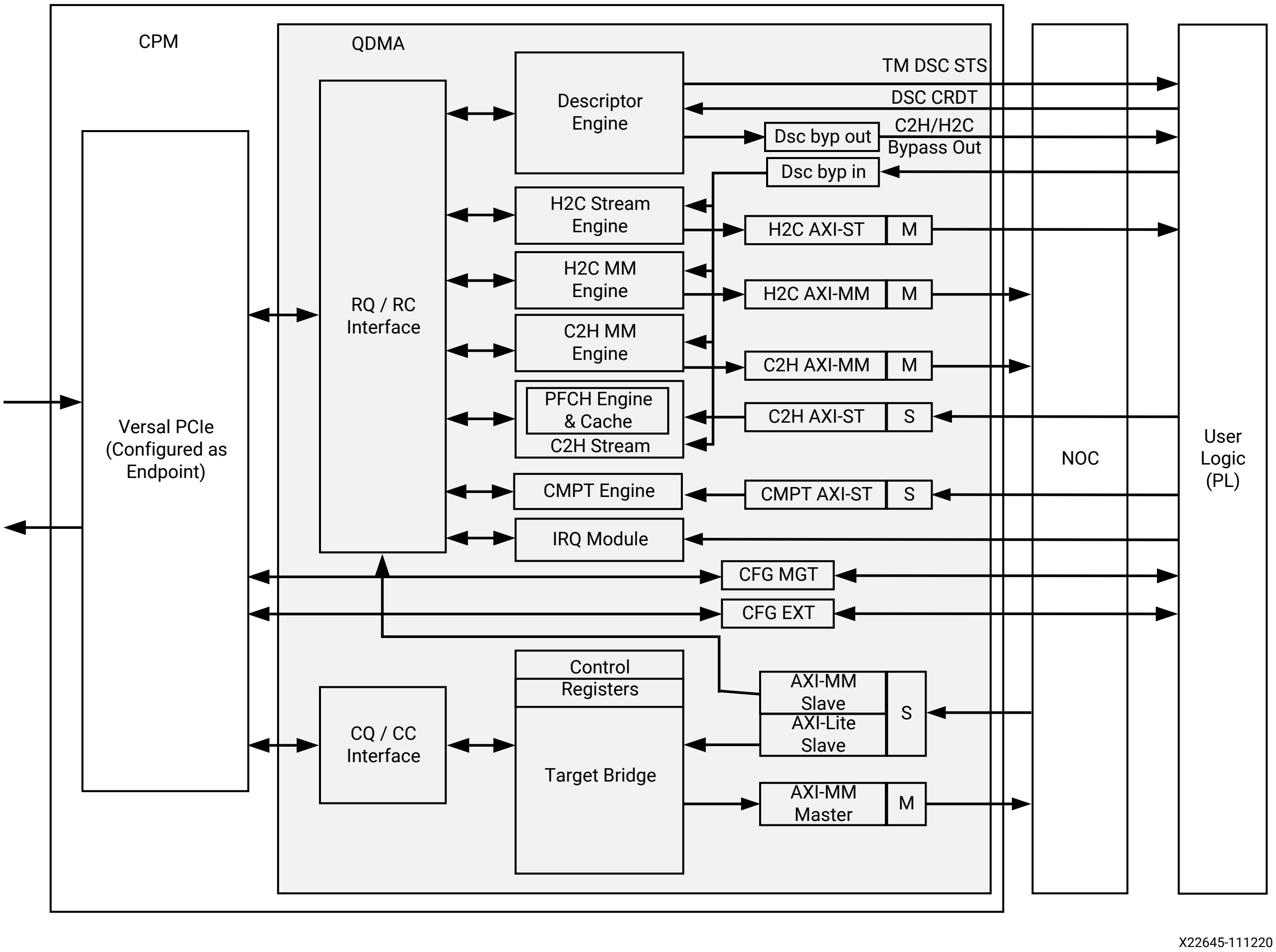
The main difference between QDMA and other DMA offerings is the concept of queues. The idea of queues is derived from the “queue set” concepts of Remote Direct Memory Access (RDMA) from high performance computing (HPC) interconnects. These queues can be individually configured by interface type, and 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.

QDMA Architecture

The following figure shows the block diagram of the QDMA.

Figure 6: **QDMA Architecture**



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 1: **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.

PCIe to AXI BARs

For each physical function, the PCIe configuration space consists of a set of five 32-bit memory BARs and one 32-bit Expansion ROM BAR. When SR-IOV is enabled, an additional five 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, 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.

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.

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 PCIE Controller operates in address aligned mode. The module uses the BAR information from the Integrated Block for IPPCIE Controller to determine where the request should be forwarded. The possible destinations for these requests are:

- DMA configuration module
- AXI4 MM Bridge interface to Network on Chip (NoC)

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 CPM Mode for PCI Express Product Guide* (PG346).

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 PCIE Controller 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 CPM Mode for PCI Express Product Guide* (PG346).

PCIe Configuration

Several factors can throttle outgoing non-posted transactions. Outgoing non-posted transactions are throttled based on flow control information from the PCIE Controller 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.

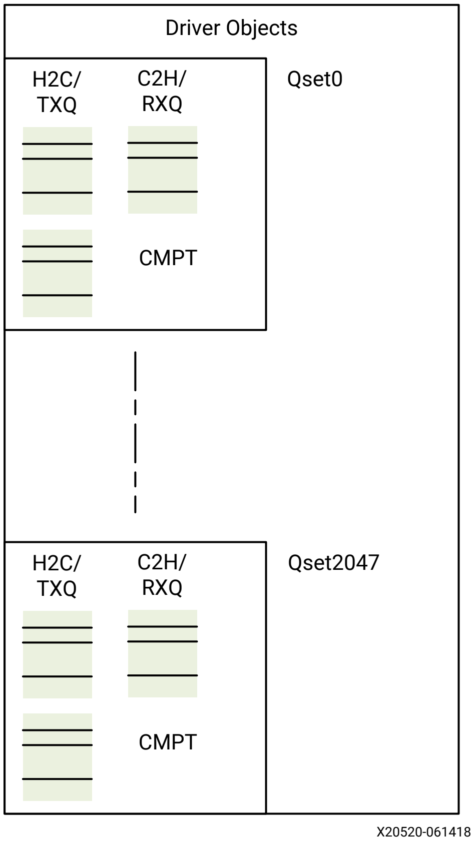
The software advertises valid descriptors for H2C and C2H queues by writing its producer index (PIDX) to the hardware. The status descriptor is the last entry of the descriptor ring, except for a C2H stream ring. The status descriptor carries the consumer index (CIDX) of the hardware so that the driver 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 7: **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.

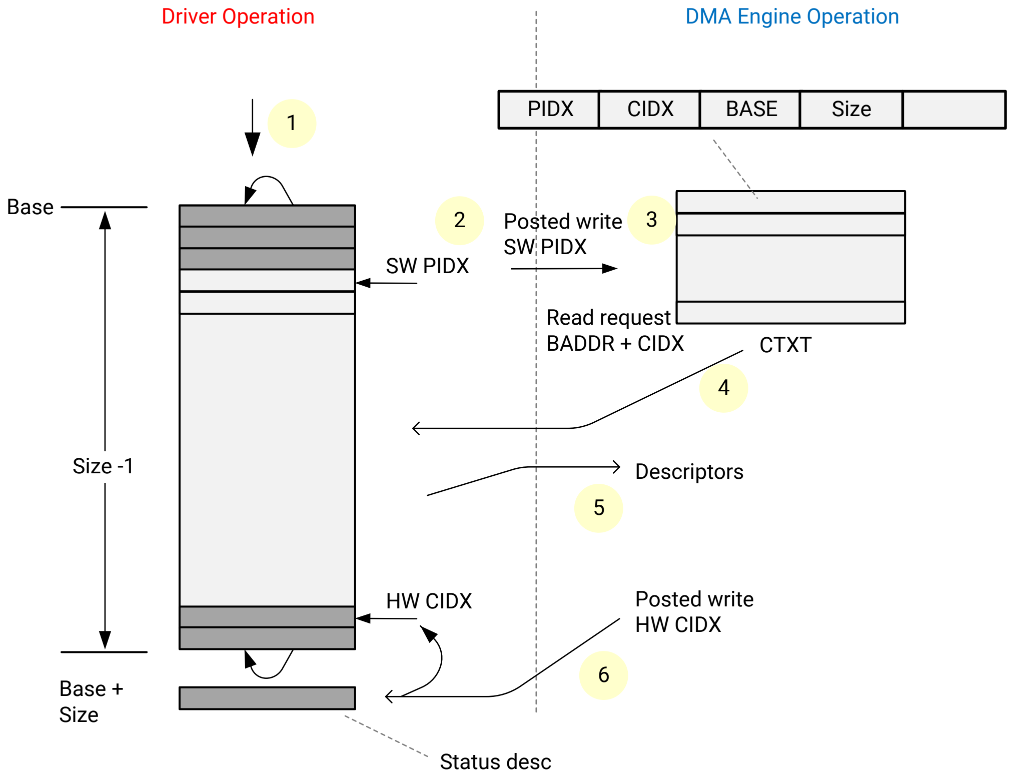
In the example above, if traffic has already started and the CIDX is 4, the maximum PIDX update is 3.

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 8: **Simple H2C and C2H Queue**



X20895-111120

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.

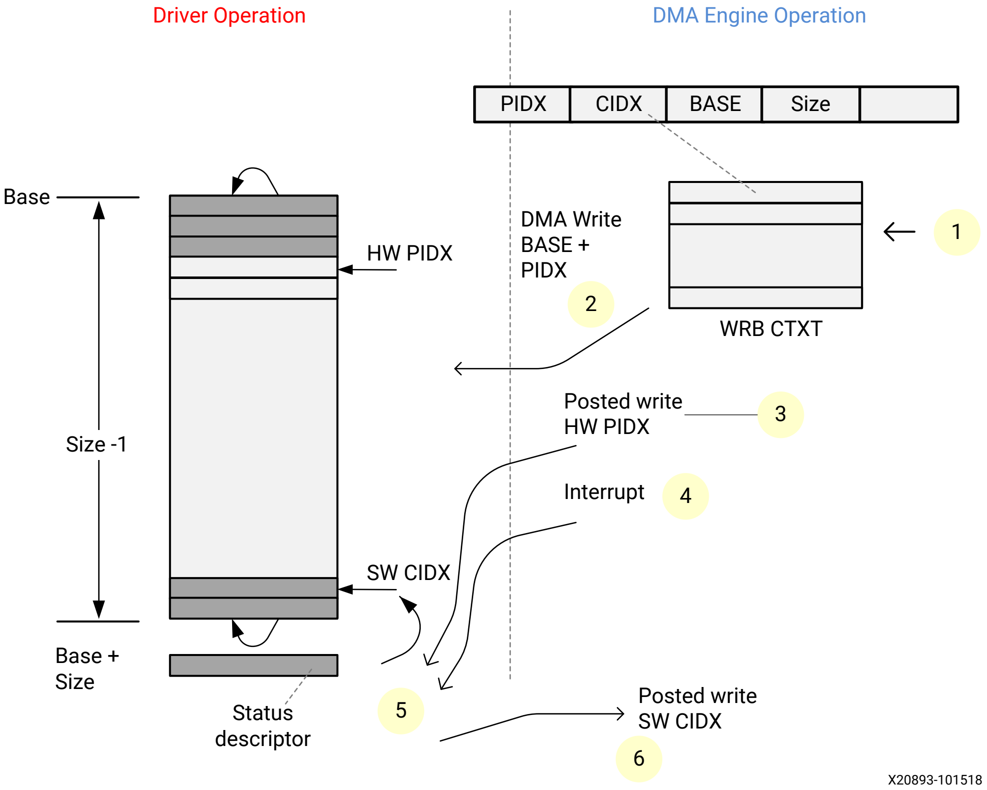
For C2H, the fetch operation is implicit through the CMPT ring.

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 9: **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.

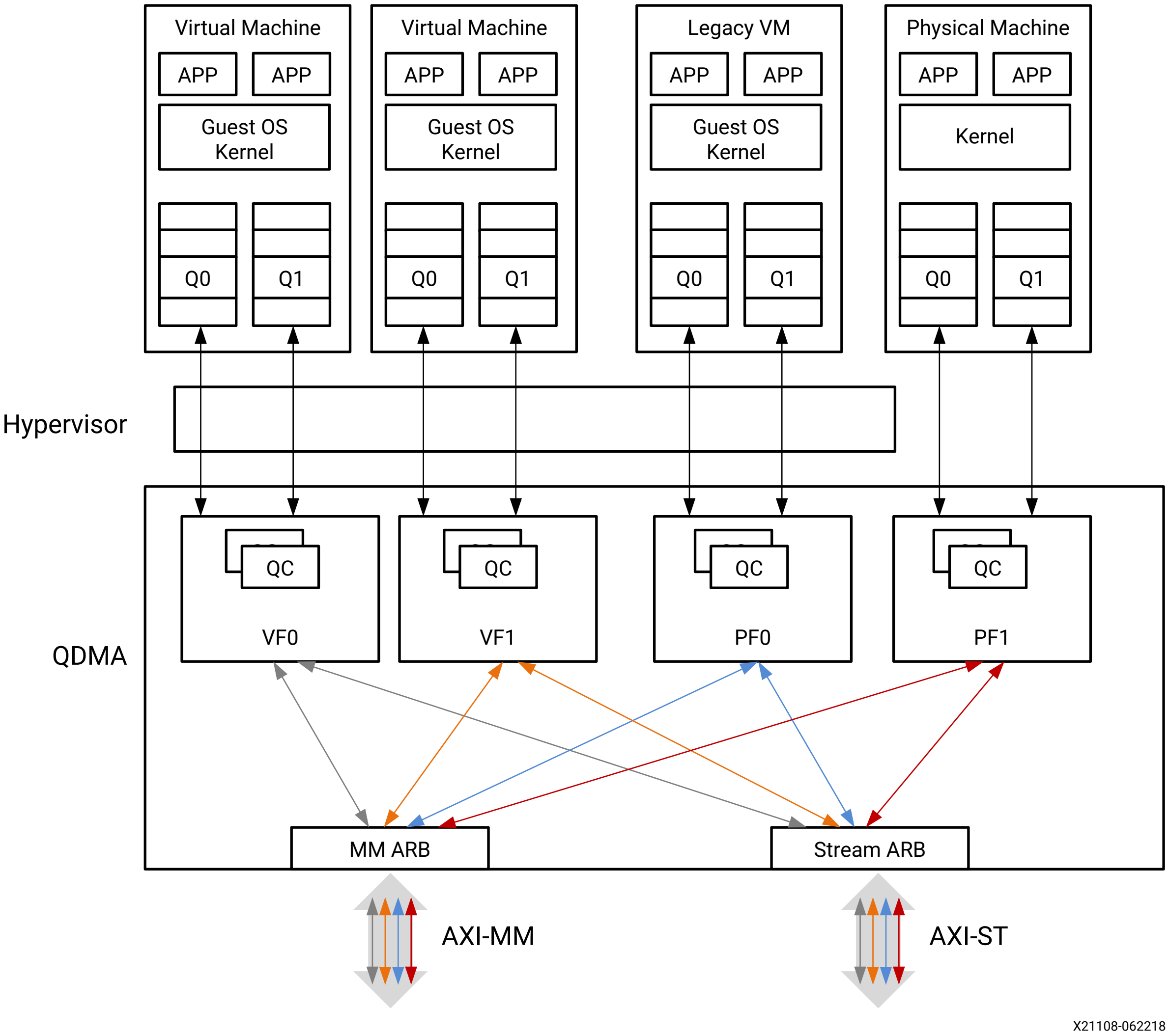
Security is an important aspect of virtualization. The QDMA Subsystem for PCI Express offers the following security functionality:

- 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 10: **QDMA in a System**



Limitations

The limitation of the QDMA is as follows:

- The DMA supports a maximum of 256 Queues on any VF function.

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

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.

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

The descriptor context is used by the descriptor engine. All descriptor rings must be aligned to the 4K address.

Table 2: **Software Descriptor Context Structure Definition**

Bit	Bit Width	Field Name	Description
[127:64]	64	dsc_base	4K aligned Base address of Descriptor Ring.
[63]	1	is_mm	This field is applicable only for internal mode. If this field is set then the descriptors will be delivered to associated H2C or C2H MM engine.
[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.
[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 pidx/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.
[50]	1	bypass	If set, the queue will operate under Bypass mode, otherwise it will be in Internal mode.

Table 2: **Software Descriptor Context Structure Definition** *(cont'd)*

Bit	Bit Width	Field Name	Description
[49:48]	2	dsc_sz	Descriptor size. 0: 8B, 1:16B; 2:32B; 3:rsv 32B is required for Memory Mapped DMA. 16B is required for H2C Stream DMA. 8B is required for C2H Stream DMA.
[47:44]	4	rng_sz	Descriptor ring size index to ring size registers.
[43:36]	8	fnc_id	Function ID. The function to which this queue belongs.
[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 QDMA_GLBL_DSC_CFG.wb_acc_int.
[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.
[31:17]	15	rsv	Reserved.
[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 3: **Hardware Descriptor Structure Definition**

Bit	Bit Width	Field Name	Description
[47:43]	5	reserved	Reserved
[42]	1	fetch_pnd	Descriptor fetch 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	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

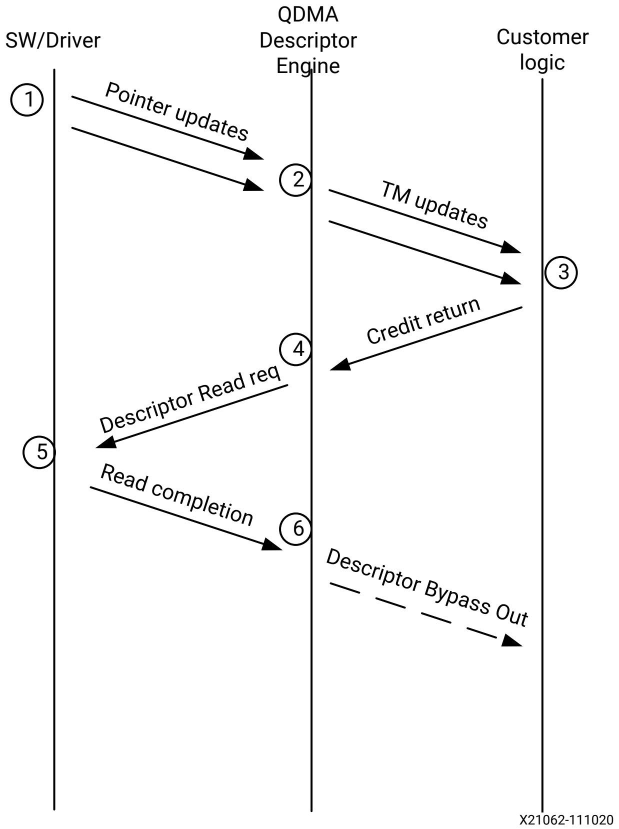
Table 4: **Credit Descriptor Context Structure Definition**

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 fence bit, credits are added only as the read request for the descriptor is generated.

Descriptor Fetch

Figure 11: **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 a ring size of -2. At any time, the software should not update the PIDX to more than a ring size of -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.wb_intvl. 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` bit 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.

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.

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 5: **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 6: **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
[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 12: **H2C Internal Mode Flow**

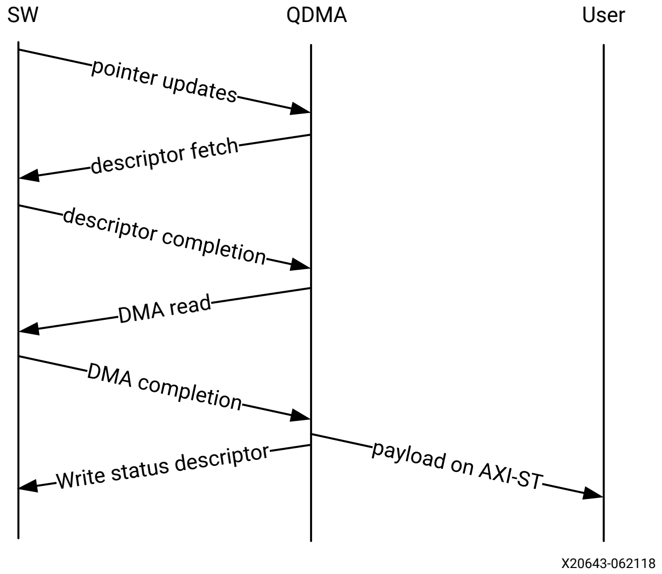
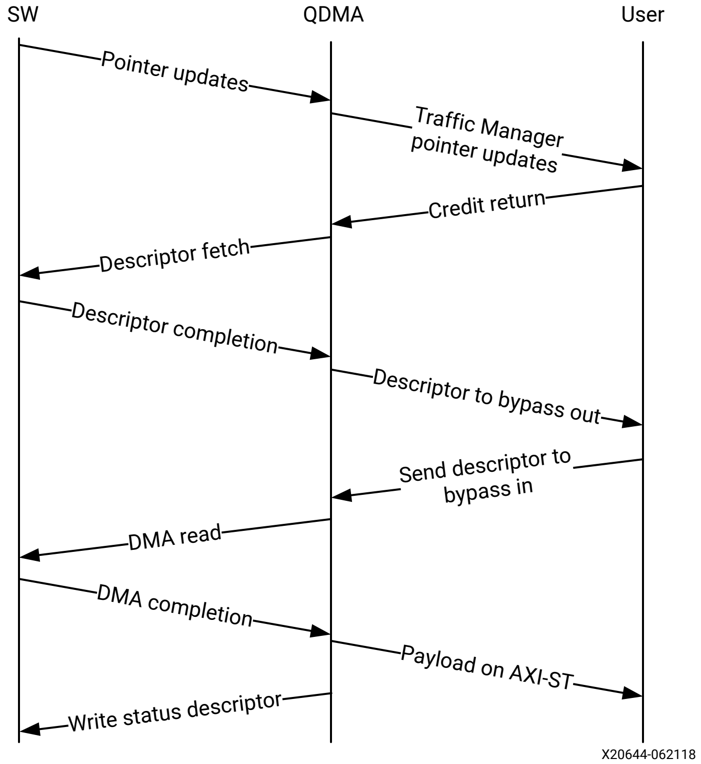


Figure 13: **H2C Bypass Mode Flow**



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 via 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 pointer 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 7: **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 8: **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

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 * descriptor size.

C2H Stream Descriptor (8B)

Table 9: **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 10: **C2H Prefetch Context Structure**

Bit	Bit Width	Field Name	Description
[45]	1	valid	Context is valid

Table 10: **C2H Prefetch Context Structure** *(cont'd)*

Bit	Bit Width	Field Name	Description
[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 cases are supported.

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

Table 11: **C2H Stream Modes**

	c2h_byp_in	desc_ctxt.desc_byp	pfch_ctxt.bypass
Simple bypass mode	simple byp in	1	1
Cache bypass mode	cache byp in	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. In most applications, one prefetch tag per host target is required. In Simple Bypass mode, the tag is not tied to any descriptor ring.

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:

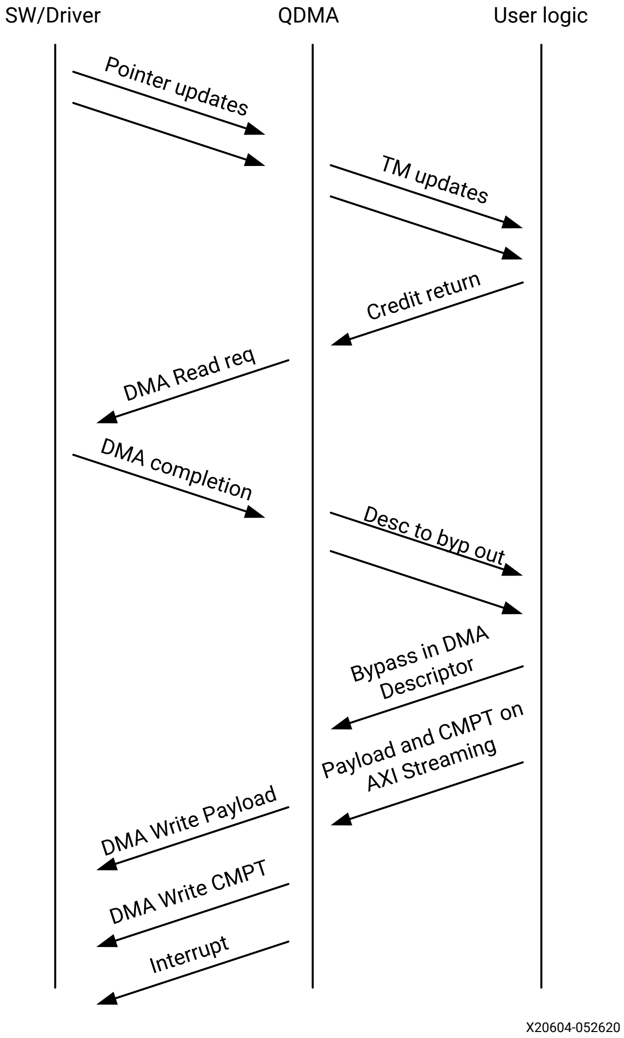
1. Software instruction:
    a. Initialize a queue (`qid`).
    b. Write to MDMA_C2H_PFCH_BYP_QID 0x1408 with valid `qid`.
    c. Read MDMA_C2H_PFCH_BYP_TAG 0x140C to obtain the prefetch tag.
    d. The prefetch tag and the `qid` that was used to fetch the tag should be used for all simple bypass packets. 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 14: **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

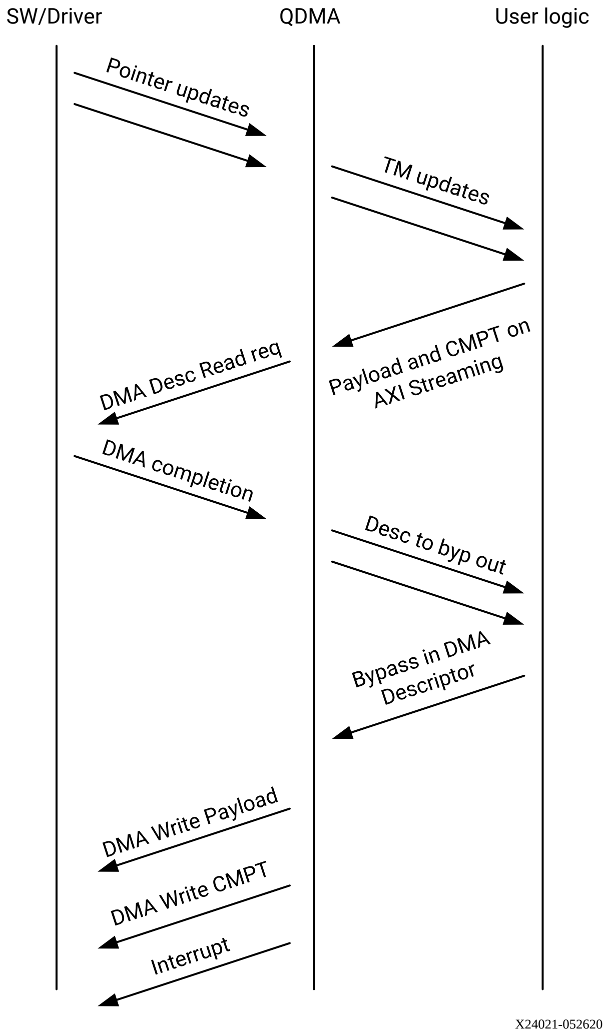
For cache bypass mode, the descriptor fetch engine sends the descriptors out on the C2H bypass output interface. The user application converts the descriptor and loops it back to the QDMA on the cache mode C2H bypass input interface. The prefetch engine sends the credits for the descriptors, and it keeps 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 bypass or 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 `c2h_byp_out_pfch_tag[6:0]` signal 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 15: **C2H Cache Bypass Mode Flow**



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

C2H Stream Packet Type

The following are some of the different C2H stream packets.

Regular Data Packet

The regular C2H data packet can be multiple bits.

- s_axis_c2h_ctrl_qid = qid of the packet
- s_axis_c2h_ctrl_len = length of the packet
- s_axis_c2h_mty = empty byte in the beat

Immediate Data Packet

The user logic can mark a data packet as 'immediate' to write to just the Completion ring without having a corresponding data packet transfer to the host. 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.

The following is the setting of the immediate data packet:

- 1 beat of data
- s_axis_c2h_ctrl_imm_data = 1'b1
- s_axis_c2h_ctrl_len = datapath width in bytes (i.e., 64 if datawidth is 512 bits)
- s_axis_c2h_mty = 0

Marker Packet

The C2H Stream Engine of the QDMA provides a way for the user logic 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 C2H Stream Descriptor Bypass Out interface. The marker is inserted by setting the `marker` bit in the C2H Stream Control input. The marker response is indicated by QDMA to the user logic by setting the `mrkr_rsp` bit on the C2H Stream Descriptor Bypass Out interface. For a marker, 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 sometimes 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 logic 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 shutdown sequence in the user logic. Although not a requirement, the marker must be sent by the user logic 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 logic:

- Sends the Completion that came along with the marker to the C2H Stream Completion Ring
- 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_mrkr' bit in the marker response is set to inform the user that an Interrupt could not be sent for this marker request. See the C2H Stream Descriptor Bypass Output interface description for details of these fields.

The following is the setting of the marker data packet:

- 1 beat of data
- s_axis_c2h_ctrl_marker = 1’b1
- s_axis_c2h_ctrl_len = data width (ex. 64 if data width is 512 bits)
- s_axis_c2h_mty = 0

The immediate data packet and the marker packet don't consume the descriptor, but 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.

Zero Length Packet

The length of the data packet can be 0. On the input, the user needs to send 1 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

Disable Completion Packet

The user can disable the completion for a specific packet. The QDMA will DMA the payload, but it will not write to the C2H Completion Ring.

The following is the setting of the disable completion packet:

- s_axis_c2h_ctrl_disable_cmp = 1

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.

C2H Completion

When the DMA write of the data payload is done, the QDMA writes the CMPT packet into the CMPT queue. Besides the user defined data, the CMPT packet also includes some other information, such as error, color, and the length. It also has a `desc_used` bit to indicate if the packet consumes a descriptor. A C2H data packet of immediate-data or marker type does not consume any descriptor.

C2H Completion Context Structure

The completion context is used by the completion engine.

Table 12: **C2H Completion Context Structure Defintion**

Bit	Bit Width	Field Name	Description
[127:126]	2	rsvd	Reserved
[125]	1	full_upd	Full Update If reset, then the Completion-CIDX-update is allowed to update only the CIDX in this context. If set, then the Completion CIDX update can update the following fields in this context: timer_ix counter_ix trig_mode en_int en_stat_desc
[124]	1	timer_running	If set, it indicates that a timer is running on this queue. This timer is for the purpose of C2H 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 HW. The SW must initialize it to 0 and then treat it as read-only.
[123]	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_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 HW. The SW must initialize it to 0 and then treat it as read-only.
[122:121]	2	err	Indicates that the C2H Completion Context is in error. This is a field written by HW. The SW 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.

Table 12: **C2H Completion Context Structure Defintion** *(cont'd)*

Bit	Bit Width	Field Name	Description
[120]	1	valid	Context is valid.
[119:104]	16	cidx	Current value of the hardware copy of the Completion Ring Consumer Index.
[103:88]	16	pidx	Completion Ring Producer Index. This is a field written by HW. The SW must initialize it to 0 and then treat it as read-only.
[87:86]	2	desc_size	Completion Entry Size: 0: 8B 1: 16B 2: 32B 3: Unknown
[85:28]	58	baddr_64	Base address of Completion ring – bit [63:6].
[27:24]	4	qsize_idx	Completion ring size index to ring size registers.
[23]	1	color	Color bit to be used on Completion.
[22:21]	2	int_st	Interrupt State: 0: ISR 1: TRIG This is a field used internally by HW. The SW must initialize it to 0 and then treat it as read-only. Because it is out of reset, the HW initializes into ISR state, it is not sensitive to trigger events. If SW desires interrupts or status writes, it must send an initial Completion CIDX update. This makes the HW move into TRIG state and as a result it becomes sensitive to any trigger conditions.
[20:17]	4	timer_idx	Index to timer register for TIMER based trigger modes.
[16:13]	4	counter_idx	Index to counter register for COUNT based trigger modes.
[12:5]	8	fnc_id	Function ID
[4:2]	3	trig_mode	Interrupt and Completion Status Write Trigger Mode: 0x0: Disabled 0x1: Every 0x2: User_Count 0x3: User 0x4: User_Timer 0x5: User_Timer_Count
[1]	1	en_int	Enable Completion interrupts.
[0]	1	en_stat_desc	Enable Completion Status writes.

C2H Completion Status Structure

The C2H 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)).

When C2H Streaming Completion is enabled, after the packet is transferred, CMPT entry and CMPT status are written to C2H Completion ring. PIDX in the Completion status can be used to indicate the currently available completion to be processed.

Table 13: **AXI4-Stream C2H Completion Status Structure**

Bit	Bit Width	Field Name	Description
[63:35]	29	Reserve	Reserved
[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

C2H Completion Entry Structure

The following is the C2H Completion ring entry structure for User format when the data format bit is set to 1’b1.

Table 14: **C2H Completion Entry User Format Structure**

Name	Size	Index
User defined bits for 32 Bytes settings	252 bits	[255:4]
User defined bits for 16 Bytes settings	124 bits	[127:4]
User defined bits for 8 Bytes settings	60 bits	[63:4]
desc_used	1	[3:3]
err	1	[2:2]
color	1	[1:1]
Data format	1	[0:0]

The following is the C2H Completion ring entry structure for Standard format when the data format bit is set to 1’b0.

Table 15: **C2H Completion Entry Standard Format Structure**

Name	Size	Index
User defined bits for 32 Bytes settings	236 bits	[255:20]
User defined bits for 16 Bytes settings	108 bits	[127:20]
User defined bits for 8 Bytes settings	44 bits	[63:20]
Len	16	[19:4]
desc_used	1	[3:3]
err	1	[2:2]
color	1	[1:1]

Table 15: **C2H Completion Entry Standard Format Structure** *(cont'd)*

Name	Size	Index
Data format	1	[0:0]

C2H Completion Input Packet

The Completion Ring entry structure is shown in C2H Stream Engine.

When the user application sends the C2H data packet, it also sends the CMPT (completion) packet to the QDMA. The CMPT packet has two formats: Standard Format and User Format.

The following is the CMPT packet from the user application in the standard format, which is when the data format bit is 1’b0.

Table 16: **CMPT Packet in Standard Format**

Name	Size	Index
User defined	44 bits-236 bits	[255:20]
rsvd	8	[19:12]
Qid	11	[11:1]
Data format	1	[0:0]

The following is the CMPT packet from the user application in the user format, which is when the data format bit is 1’b1.

Table 17: **CMPT Packet in User Format**

Name	Size	Index
User defined	61 bits-253 bits	[255:3]
rsvd	2	[2:1]
Data format	1	[0:0]

The CMPT packet has three types: 8B, 16B, or 32B. When it is 8B or 16B, it only needs one beat of the data. When it is 32B, it needs two beats of data. Each data beat is 128 bits.

C2H Interrupt/Completion Status Moderation

The QDMA provides a means to moderate the C2H 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 C2H 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 QDMA.

The C2H completion interrupt moderation is handled by the completion engine inside the C2H engine. The completion engine stores the C2H completion ring contexts of all the queues. It is possible to individually enable or disable the sending of interrupts and C2H completion status descriptors 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 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 C2H 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 descriptor, 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 C2H 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_COUNT:** This mode allows the QDMA is sensitive to either of two triggers. One of these triggers is sent by the user along with the C2H packet. The other trigger is the presence of more than a programmed threshold of unread Completion entries in the Completion Ring, as seen by the HW. This threshold 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.
- **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 C2H packet. The other trigger is the expiration of the timer that is associated with the C2H 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 C2H Timer.
- **TRIGGER_USER_TIMER_COUNT:** This mode allows the QDMA is sensitive to any of three triggers. One of these triggers is sent by the user along with the C2H packet. The second trigger is the expiration of the timer that is associated with the C2H queue. The period of the timer is driver programmable on a per-queue basis. The third trigger is the presence of more than a programmed threshold of unread Completion entries in the Completion Ring, as seen by the HW. This threshold is driver programmable on a per-queue basis. The QDMA evaluates whether or not to send an interrupt when any 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.
- **TRIGGER_DIS:** In this mode, the QDMA does not send C2H 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.

The following are the flowcharts of different modes. These flowcharts are from the point of view of the C2H 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**

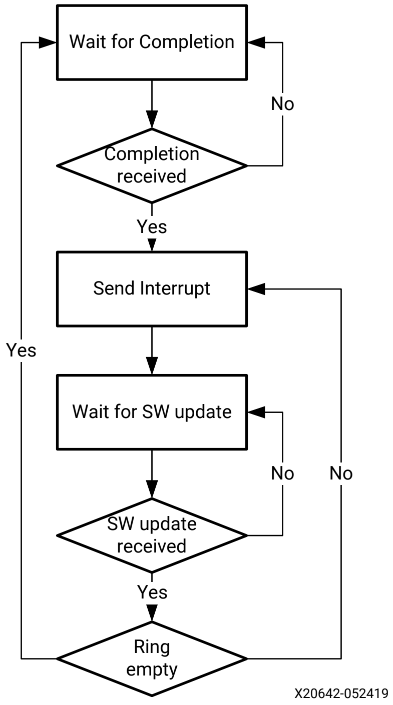


Figure 17: **Flowchart for USER Mode**

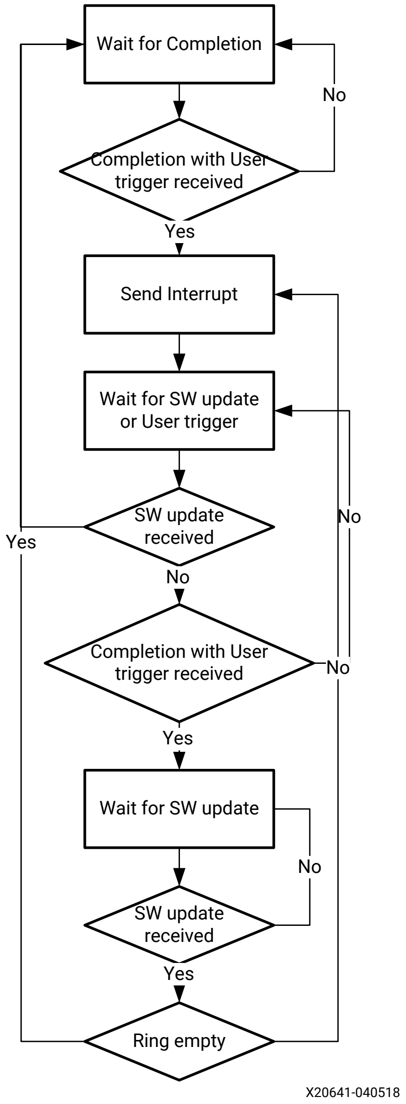


Figure 18: **Flowchart for USER_COUNT Mode**

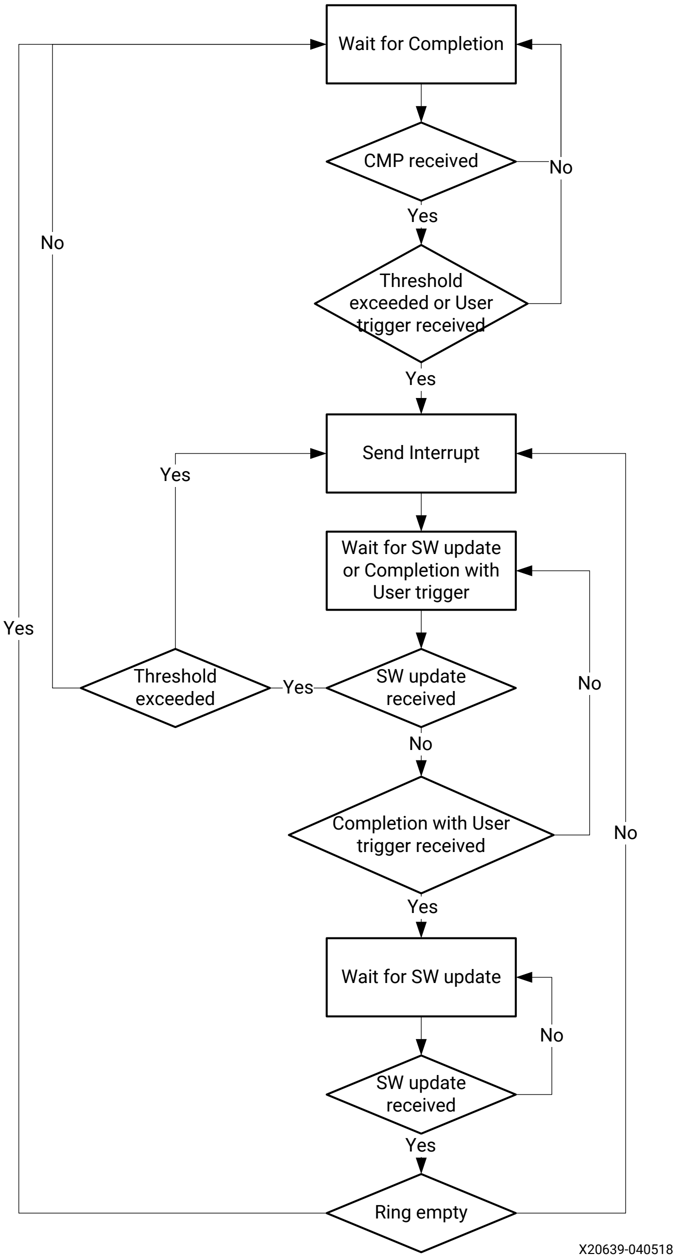
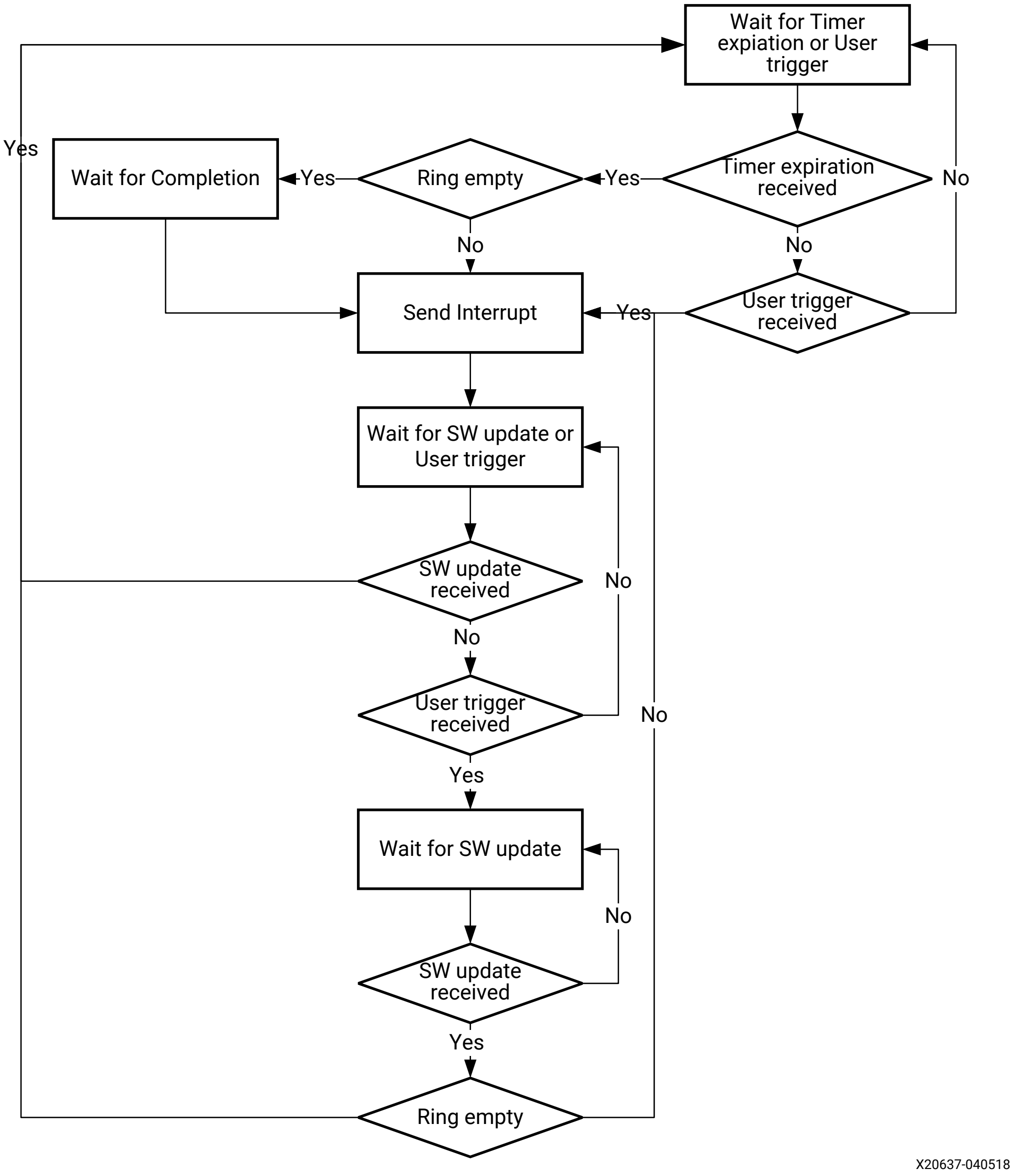


Figure 19: **Flowchart for USER_TIMER Mode**



C2H Timer

The C2H timer is a 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, then 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 for 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, sends a Status Descriptor marked with error.
- If enabled and not outstanding, sends an Interrupt.
- 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.

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

Interrupts

The QDMA supports up to 2K total MSI-X vectors. A single MSI-X vector can be used to support multiple queues.

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.

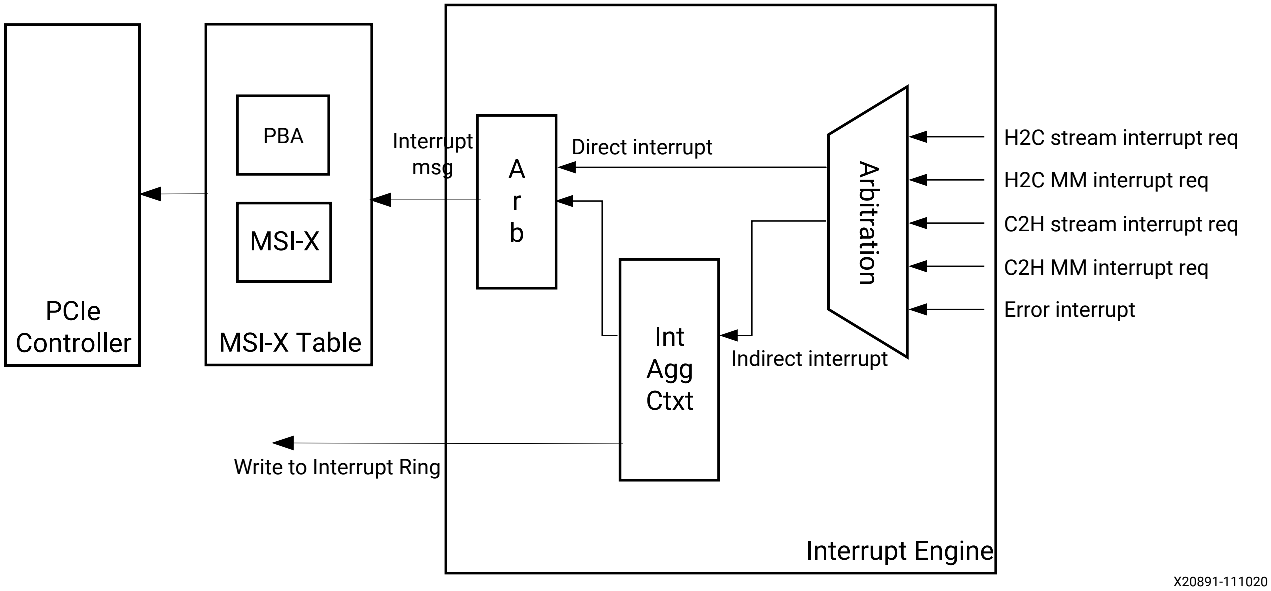
Interrupts are broadcast to all PFs, and maintain status for each PF in a queue based scheme. The queue based interrupts include the interrupts from the H2C MM, H2C stream, C2H MM, and C2H stream.

Interrupt Engine

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

This block diagram is of the Interrupt Engine.

Figure 20: **Interrupt Engine Block Diagram**



When the H2C or C2H interrupt occur, it first reads the QID to vector table. The table has 2K entries to support up to 2K queues. Each entry of the table includes two portions: one for H2C interrupts, and one for C2H interrupts. The table maps the QID to the vector, and indicates if the interrupt is direct interrupt mode or indirect interrupt mode. If it is direct interrupt mode, the vector is used to generate the PCIe MSI-X message. If it is indirect interrupt mode, the vector is the ring index, which is the index of the Interrupt Context for the Interrupt Aggregation Ring.

The following is the data in the QID to vector table.

Table 18: **QID to Vector Table**

Signal	Bit	Owner	Description
h2c_en_coal	[17:17]	Driver	1'b1: indirect interrupt mode. 1'b0: direct interrupt mode for H2C interrupt.
h2c_vector	[16:9]	Driver	For direct interrupt, it is the interrupt vector index of MSI-X table. For indirect interrupt, it is the ring index.
c2h_en_coal	[8:8]	Driver	1'b1: indirect interrupt mode. 1'b0: direct interrupt mode for C2H interrupt.
c2h_vector	[7:0]	Driver	For direct interrupt, it is the interrupt vector index of MSI-X table. For indirect interrupt, it is the ring index.

The QID to Vector table is programmed by the context access.

- Context access through QDMA_TRQ_SEL_IND:
    - QDMA_IND_CTXT_CMD.Qid = Qid
    - QDMA_IND_CTXT_CMD.Sel = MDMA_CTXT_SEL_INT_QID2VEC (0xC)
    - QDMA_IND_CTXT_CMD.Op = MDMA_CTXT_CMD_WR or MDMA_CTXT_CMD_RD (MDMA_CTXT_CMD_CLR and MDMA_CTXT_CMD_INV are not supported for Qid to Vector table)

Direct Interrupt

For direct interrupt, the QDMA processes the interrupt with the following steps.

- Look up the QID to Vector Table.
- Send out the PCIe MSI-X message.

Interrupt Aggregation Ring

For 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 same source, until software services the interrupts.

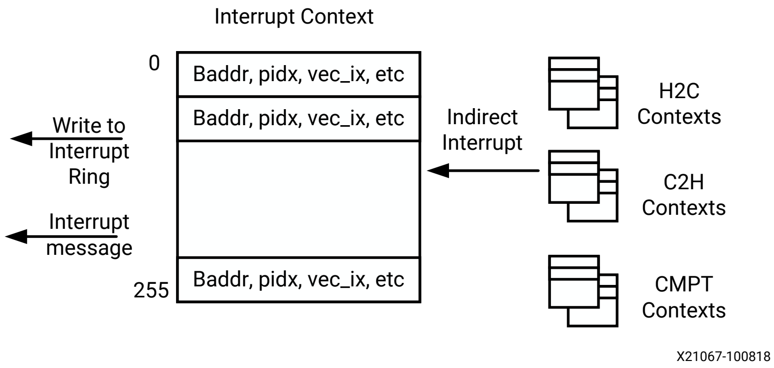
In the indirect interrupt, the QDMA processes the interrupt with the following steps.

- Look up the QID to Vector Table.
- Look up the Interrupt Context.
- Write to the Interrupt Aggregation Ring.

- Send out the PCIe MSI-X message.

This block diagram is of the indirect interrupt.

Figure 21: **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.

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

Table 19: **Interrupt Context Structure (0x8)**

Signal	Bit	Owner	Description
pidx	[75:64]	DMA	Producer Index
page_size	[63:61]	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	[60:9]	Drive	Base address of Interrupt Aggregation Ring – bit[63:12]
color	[8]	DMA	Color bit
int_st	[7]	DMA	Interrupt State: 0: WAIT_TRIGGER 1: ISR_RUNNING
reserved	[6]	NA	Reserved
vec	[5:1]	Driver	Interrupt vector index in MSI-X 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 * (number of queues + error interrupts that are mapped to this ring)

The Interrupt Context is programmed by the context access. The QDMA_IND_CTXT_CMD.Qid has the ring index, which is from the Qid to Vector Table. 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 , or

        MDMA_CTXT_CMD_INV.

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

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

Table 20: **Interrupt Aggregation Ring Entry Structure**

Signal	Bit	Owner	Description
coal_color	[63:63]	DMA	The color bit of the Interrupt Aggregation Ring. This bit inverts every time pidx wraps around on the Interrupt Aggregation Ring.
qid	[62:52]	DMA	This is from Interrupt source. Queue ID.
int_type	[51:51]	DMA	0: H2C 1: C2H
err_int	[50:50]	DMA	0: non-error interrupt 1: error interrupt
reserved	[49:39]	DMA	Reserved
stat_desc	[38:0]	DMA	This is the status descriptor of the Interrupt source.

The following is the information in the `stat_desc`.

Table 21: **stat_desc Information**

Signal	Bit	Owner	Description
error	[38:35]	DMA	This is from interrupt source: {c2h_err[1:0], 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

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 writes to the Interrupt Aggregation Ring, it reads color bit from the Interrupt Context, and writes it to the entry. When the ring (PIDX) wraps around, the hardware will flip the color bit in the Interrupt Context. In this way, when the software reads from the Interrupt Aggregation Ring, it will know which entries got written by the hardware by looking at the color bit.

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

When the `err_int` is set, it is an error interrupt. The software can then check the error status register of the Central Error Aggregator QDMA_GLBL_ERR_STAT (0x248). The register shows the error source. The software can then read the error status register of the Leaf Error Aggregator of the corresponding error.

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.

The two cases are as follows:

- 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 QDMA sends out the PCIe MSI-X message using the interrupt vector from the Interrupt Context.

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 Interrupt Aggregation 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] (0x6400). If the software `cidx` is equal to the `pidx`, this will trigger a write to the Interrupt Context 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`, it will send out another PCIe MSI-X message. Therefore, the software can read the Interrupt Aggregation Ring again. After that, the software can perform a pointer update of the interrupt source ring. For example, for a C2H stream interrupt, the software will update the pointer of the interrupt source ring, which is the C2H Completion Ring.

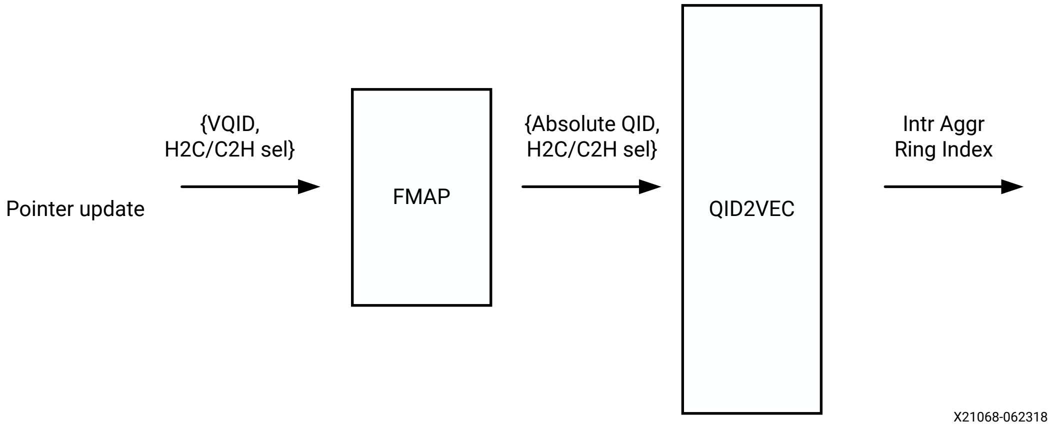
These are the steps for the software:

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, it does one dynamic point update of the software cidx using the register QDMA_DMAP_SEL_INT_CIDX[2048] (0x6400). The `Qid` in the register is the `Qid` in the last written entry. This tells hardware the pointer of the Interrupt Aggregation Ring that the software reads to.

   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.

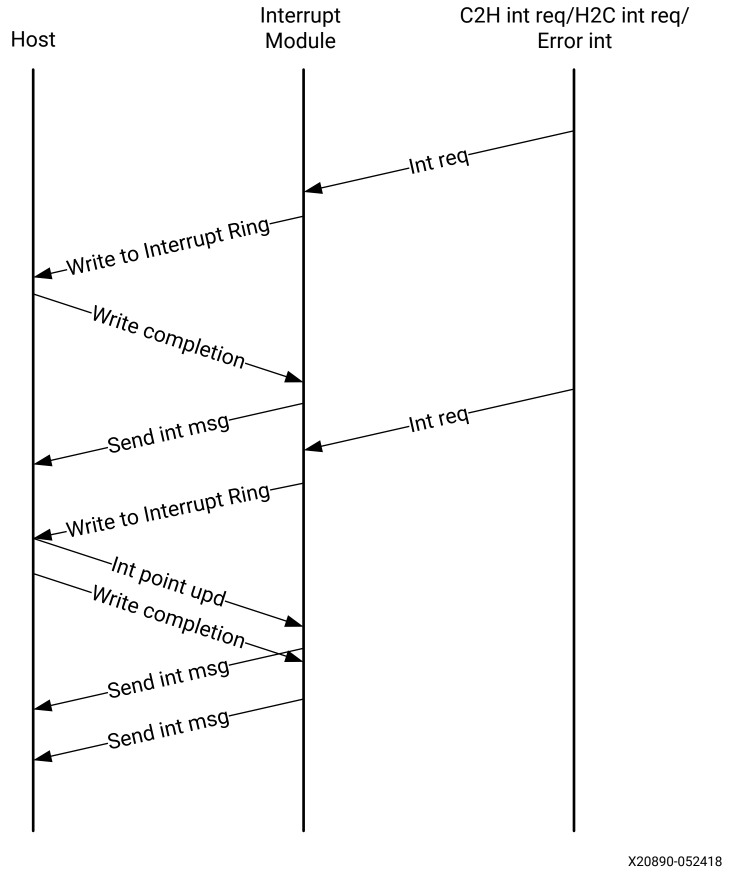
When the software performs the dynamic point update for the Interrupt Aggregation Ring using the register QDMA_DMAP_SEL_INT_CIDX[2048] (0x6400), it needs to use the virtual `qid`. The FMAP block in the hardware translates the virtual `qid` to absolute `qid`. The interrupt Engine uses the absolute `qid` when it looks up the `qid` to Vector Table.

Figure 22: **Interrupt Engine**



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.

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).

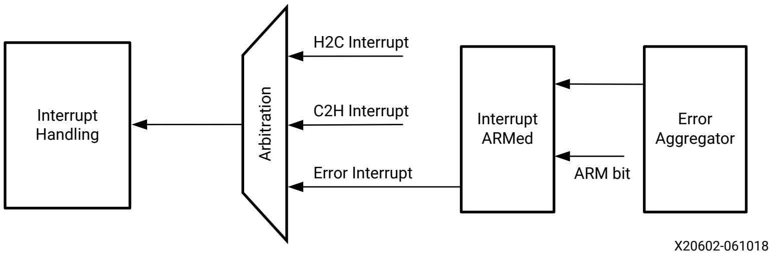
The Error Interrupt gets the vector from the error interrupt register QDMA_GLBL_ERR_INT. For the direct interrupt, the vector is the interrupt vector index of the MSI-X table.

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.

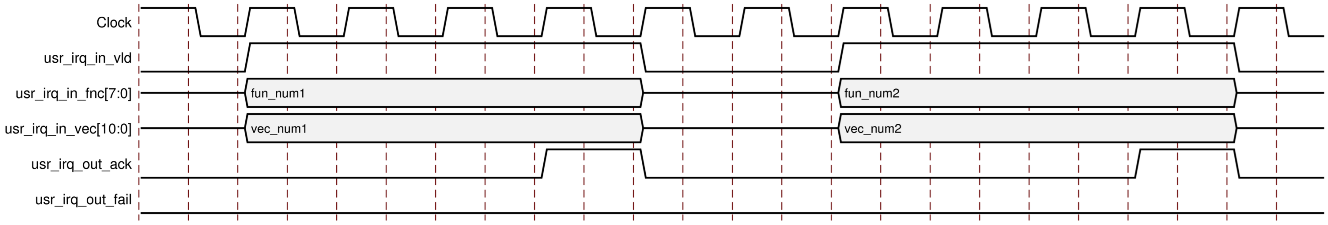
The following figure shows the error interrupt register block diagram.

Figure 24: **Error Interrupt Handling**



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.

The table can be programmed through the QDMA_TRQ_SEL_FMAP address space. Because this space only exists in the PF address map, only a physical function can modify this table.

Context Programming

- Program all mask registers to 1. They are QDMA_IND_CTXT_MASK_0 (0x824) to QDMA_IND_CTXT_MASK_7 (0x830).
- Program context values for the following registers: QDMA_IND_CTXT_DATA_0 (0x804), to QDMA_IND_CTXT_DATA_7 (0x810).
- 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.
- The context that is accessed is given in `bits [4:1]`.
- Context programing write/read does not occur when bit [0] is set.

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 22: **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.
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.

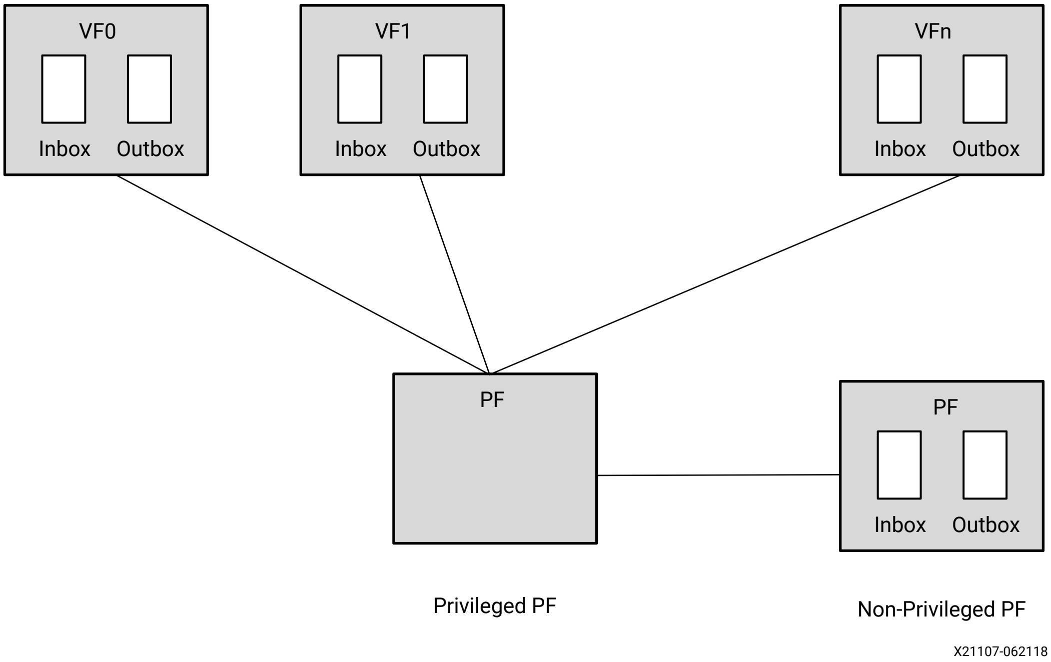
Each function (both PF and VF) has an inbox and an outbox that can fit a message size of 128B. A VF accesses its own mailbox, and a PF accesses its own mailbox and all the functions (PF or VF) associated with that PF.

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**

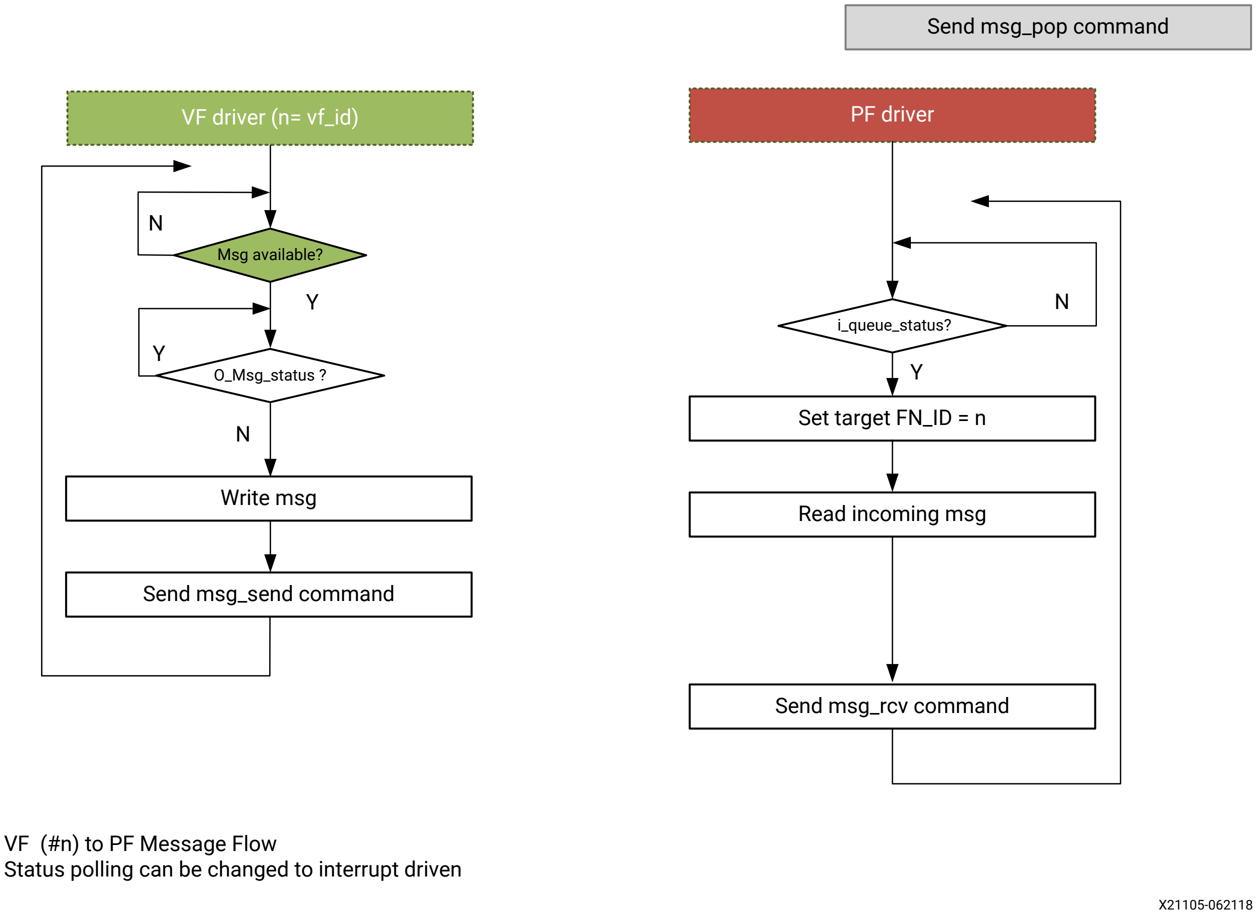


VF To PF Messaging

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:

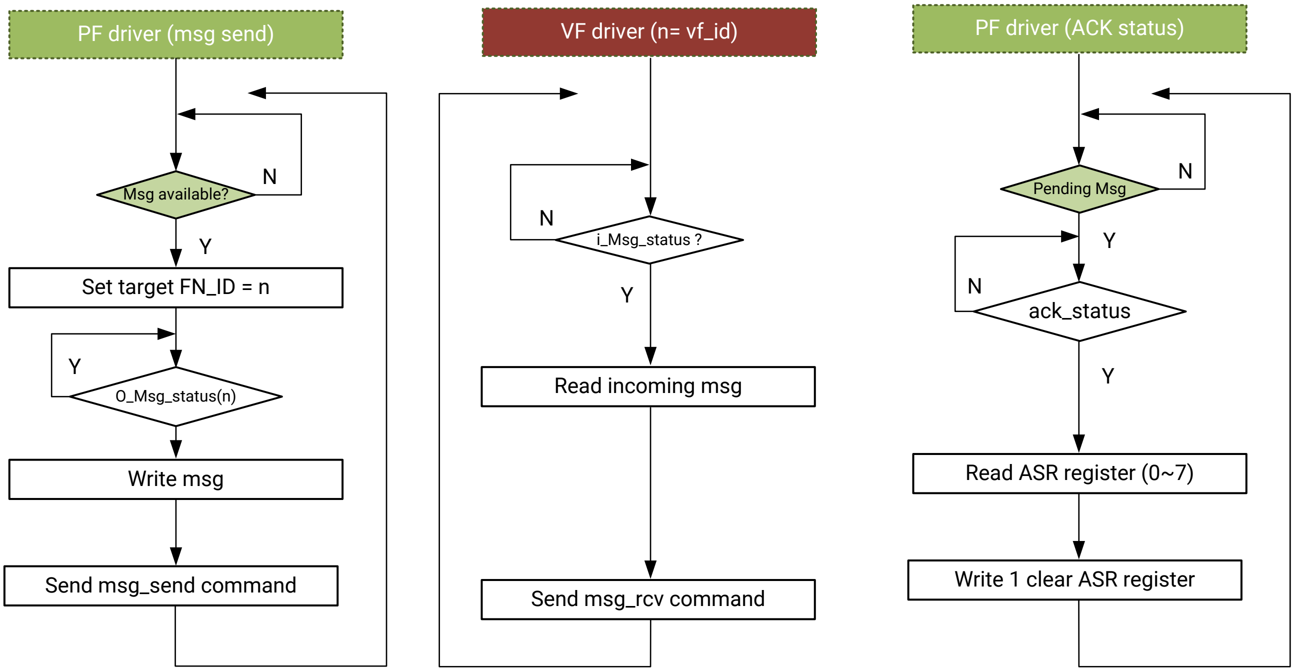
A PF can send messages to multiple destination functions, therefore, it may receives multiple acknowledgments at the moment when checking the status. As illustrated in the following figure, a PF driver must set Mailbox Target Function Register to the destination function ID before doing any message operation; 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**



X21106-062118

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.

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 23: **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.

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).

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 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 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.

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

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

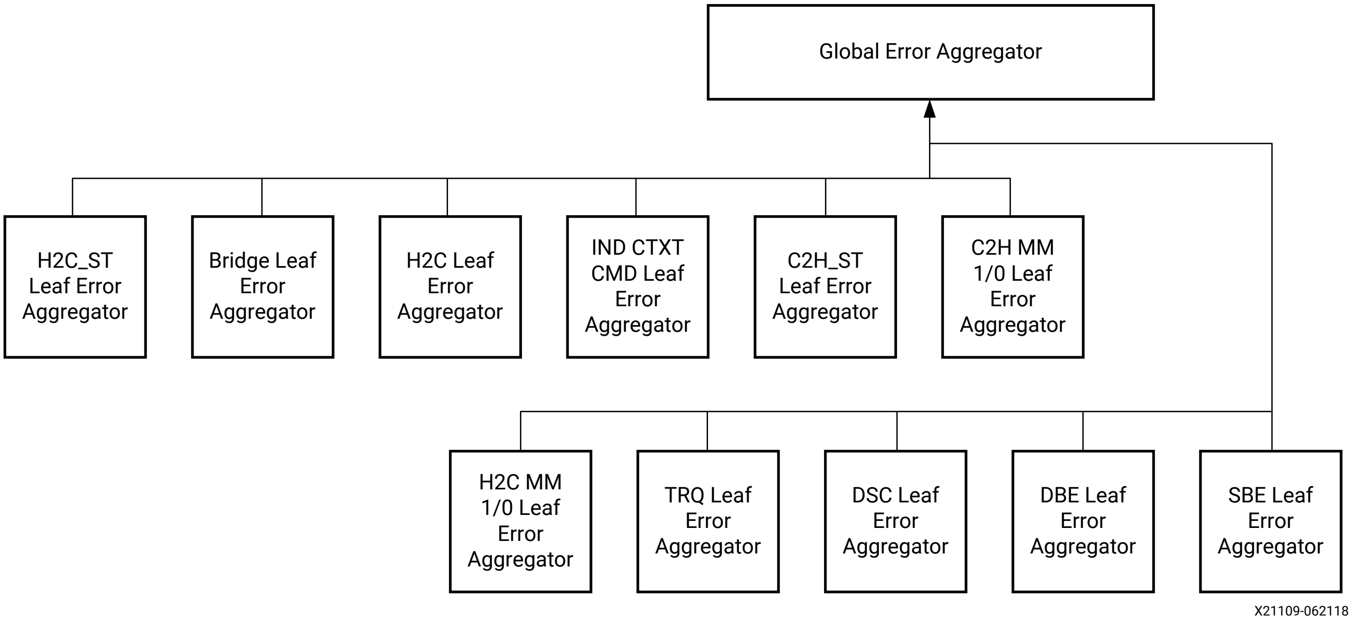
There are Leaf Error Aggregators in different places. They log the errors and propagate them to the central place. The Central Error Aggregator aggregates the errors from all of the Leaf Error Aggregators.

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)

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 WRQ from the C2H DMA Write Engine.
2. Invert the WPL parity on the data transfer.

Port Descriptions

QDMA Global Signals

Table 24: **QDMA Global Port Descriptions**

Port Name	I/O	Description
gt_refclk0_p/gt_refclk0_n	I	GT reference clock
pci_gt_txp/pci_gt_txn [PL_LINK_CAP_MAX_LINK_WIDTH-1:0]	O	PCIe TX serial interface.
pci_gt_rxp/pci_gt_rxn [PL_LINK_CAP_MAX_LINK_WIDTH-1:0]	I	PCIe RX serial interface.
pcie0_user_lnk_up	O	Output active-High identifies that the PCI Express core is linked up with a host device.
pcie0_user_clk	O	User clock out. PCIe derived clock output for all interface signals output/input to the QDMA. Use this clock to drive inputs and gate outputs from QDMA.
dma0_axi_aresetn	O	User reset out. AXI reset signal synchronous with the clock provided on the pcie0_user_clk output. This reset should drive all corresponding AXI Interconnect aresetn signals.
dma0_soft_resetn	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.

All AXI interfaces are clocked out and in by the `pcie0_user_clk` signal. You are responsible for using `pcie0_user_clk` to drive all signals into the CPM.

`pcie0_user_clk` should be used to interface with the CPM. In the user logic, any available clocks can be used.

AXI Slave Interface

AXI Bridge Slave ports are connected from the Versal ACAP Network on Chip (NoC) to the CPM DMA internally. For Slave Bridge AXI-MM details, see the *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313).

To access QDMA registers, you must follow the protocols outlined in the AXI Slave Bridge Register Limitations section.

Related Information

Slave Bridge Registers Limitations

AXI4 Memory Mapped Interface

AXI4 Memory Mapped (MM) Master ports are connected from the CPM to the Versal ACAP Network on Chip (NoC) internally. For details, see the *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313). The AXI4 MM Master interface can be connected to the DDR memory, or to the PL user logic, depending on the NoC configuration.

AXI4-Lite Master Interface

AXI4-Lite Master ports are connected from the CPM to the Versal ACAP Network on Chip (NoC) internally. For details, see the *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313).

Use the SmartConnect IP to connect the NoC to the AXI4-Lite Master interface. For details, see the *SmartConnect LogiCORE IP Product Guide* (PG247).

AXI4-Stream H2C Interface

Table 25: **AXI4-Stream H2C Interface Descriptions**

Port Name	I/O	Description
dma0_m_axis_h2c_tdata [AXI_DATA_WIDTH-1:0]	O	Data output for H2C AXI4-Stream.
dma0_m_axis_h2c_dpar [AXI_DATA_WIDTH/8-1 : 0]	O	Odd parity calculated bit-per-byte over m_axis_h2c_tdata. m_axis_h2c_dpar[0] is parity calculated over m_axis_h2c_tdata[7:0]. m_axis_h2c_dpar[1] is parity calculated over m_axis_h2c_tdata[15:8], and so on.
dma0_m_axis_h2c_tuser_qid[10:0]	O	Queue ID
dma0_m_axis_h2c_tuser_port_id[2:0]	O	Port ID
dma0_m_axis_h2c_tuser_err	O	If set, indicates the packet has an error. The error could be coming from the PCIe, or the QDMA might have encountered a double bit error.
dma0_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.
dma0_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.
dma0_m_axis_h2c_tuser_zero_byte	O	When set, it indicates that the current beat is an empty beat (zero bytes are being transferred).
dma0_m_axis_h2c_tvalid	O	Valid
dma0_m_axis_h2c_tlast	O	Indicates that this is the last cycle of the packet transfer.
dma0_m_axis_h2c_tready	I	Ready

AXI4-Stream C2H Interface

Table 26: **AXI4-Stream C2H Interface Descriptions**

Port Name	I/O	Description
dma0_s_axis_c2h_tdata [AXI_DATA_WIDTH-1:0]	I	Supports 4 data widths: 64 bits, 128 bits, 256 bits, and 512 bits. Every C2H data packet has a corresponding C2H completion packet.
dma0_s_axis_c2h_dpar [AXI_DATA_WIDTH/8-1 : 0]	I	Odd parity computed as bit per byte.
dma0_s_axis_c2h_ctrl_len [15:0]	I	Length of the packet. For 0 (zero) byte write, the length is 0. C2H stream packet data length is limited to 7 * descriptor size.
dma0_s_axis_c2h_ctrl_qid [10:0]	I	Queue ID.
dma0_s_axis_c2h_ctrl_imm_data	I	Immediate data. This allows only the completion and no DMA on the data payload.
dma0_s_axis_c2h_ctrl_dis_cmp	I	Disable completion
dma0_s_axis_c2h_ctrl_marker	I	Marker message used for making sure pipeline is completely flushed. After that, you can safely perform queue invalidation.
dma0_s_axis_c2h_ctrl_port_id [2:0]	I	Port ID.
dma0_s_axis_c2h_ctrl_user_trig	I	User trigger. This can trigger the interrupt and the status descriptor write if they are enabled.
dma0_s_axis_c2h_mty [5:0]	I	Empty byte should be set in last beat.
dma0_s_axis_c2h_tvalid	I	Valid.
dma0_s_axis_c2h_tlast	I	Indicate last packet.
dma0_s_axis_c2h_tready	O	Ready.
dma0_s_axis_c2h_cmpt_tdata[127:0]	I	Completion data from the user application. This contains information that is written to the completion ring in the host. This information includes the length of the packet transferred in bytes, error, color bit, and user data. Based on completion size, this could be 1 or 2 beats. Every C2H completion packet has a corresponding C2H data packet.
dma0_s_axis_c2h_cmpt_size[1:0]	I	00: 8B completion. 01: 16B completion. 10: 32B completion. 11: unknown.
dma0_s_axis_c2h_dmpt_dpar[3:0]	I	Odd parity computed as bit per word. 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.
dma0_s_axis_c2h_cmpt_tvalid	I	Valid
dma0_s_axis_c2h_cmpt_tlast	I	Indicates the end of the completion data transfer.
dma0_s_axis_c2h_cmpt_tready	O	Ready

AXI4-Stream C2H Completion Interface

Table 27: **AXI4-Stream C2H Completion Interface Descriptions**

Port Name	I/O	Description
dma0_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.
dma0_s_axis_c2h_cmpt_size [1:0]	I	00: 8B completion. 01: 16B completion. 10: 32B completion. 11: 64B completion
dma0_s_axis_c2h_cmpt_dpar [15:0]	I	Odd parity computed as bit per 32b. dma0_s_axis_c2h_cmpt_dpar[0] is parity over dma0_s_axis_c2h_cmpt_tdata[31:0]. dma0_s_axis_c2h_cmpt_dpar[1] is parity over dma0_s_axis_c2h_cmpt_tdata[63:31], and so on.
dma0_s_axis_c2h_cmpt_ctrl_qid[10:0]	I	Completion queue ID.
dma0_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.
dma0_s_axis_c2h_cmpt_ctrl_user_trig	I	Triggers the interrupt and the status descriptor write when enabled.
dma0_s_axis_c2h_cmpt_ctrl_cmpt_type[1:0]	I	2’b00: NO_PLD_NO_WAIT. The Completion (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 packet, and it needs to wait for the payload packet to be sent before sending the CMPT packet.
dma0_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.
dma0_s_axis_c2h_cmpt_ctrl_port_id[2:0]	I	Port ID.
dma0_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.
dma0_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.
dma0_s_axis_c2h_cmpt_tvalid	I	Valid.
dma0_s_axis_c2h_cmpt_tready	O	Ready.

AXI4-Stream Status Interface

Table 28: **AXI-ST C2H Status Interface Descriptions**

Port Name	I/O	Description
dma0_axis_c2h_status_valid	O	Valid per descriptor.
dma0_axis_c2h_status_qid[10:0]	O	QID of the packet.

Table 28: **AXI-ST C2H Status Interface Descriptions** *(cont'd)*

Port Name	I/O	Description
dma0_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.

AXI4-Stream C2H Write Cmp Interface

Table 29: **AXI-ST C2H Write Cmp Interface Descriptions**

Port Name	I/O	Description
dma0_axis_c2h_dmawr_cmp	O	This signal is asserted when the last data payload Wrq of the packet gets the completion of Wcp. It is one pulse per packet.

VDM Interface

Table 30: **VDM Port Descriptions**

Port Name	I/O	Description
dma0_st_rx_msg_valid	O	Valid
dma0_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
dma0_st_rx_msg_last	O	Indicates the last beat
dma0_st_rx_msg_rdy	I	Ready. ***Note*:** When this interface is not used, Ready must be tied-off to 1.

RECOMMENDED: *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, the FIFO overflows and these messages are dropped. It is best to repeat VDM messages at regular intervals.*

FLR Interface

Table 31: **FLR Port Descriptions**

Port Names	I/O	Description
dma0_usr_flr_fnc [7:0]	O	Function The function number of the FLR status change.
dma0_usr_flr_set	O	Set Asserted for 1 cycle indicating that the FLR status of the function indicated on dma0_usr_flr_fnc[7:0] is active.
dma0_usr_flr_clr	O	Clear Asserted for 1 cycle indicating that the FLR status of the function indicated on dma0_usr_flr_fnc[7:0] is completed.
dma0_usr_flr_done_fnc [7:0]	I	Done Function The function for which FLR has been completed.
dma0_usr_flr_done_vld	I	Done Valid Assert for one cycle to signal that FLR for the function on dma0_usr_flr_done_fnc[7:0] has been completed.

QDMA Descriptor Bypass Input Interface

Table 32: **QDMA H2C-Streaming Bypass Input Interface Descriptions**

Port Name	I/O	Description
dma0_h2c_byp_in_st_addr[63:0]	I	64-bit starting address of the DMA transfer.
dma0_h2c_byp_in_st_len[15:0]	I	The number of bytes to transfer.
dma0_h2c_byp_in_st_sop	I	Indicates start of packet. Set for the first descriptor. Reset for the rest of the descriptors.
dma0_h2c_byp_in_st_eop	I	Indicates end of packet. Set for the last descriptor. Reset for the rest of the descriptors
dma0_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.
dma0_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 interface. This can only be set for an EOP descriptor.

Table 32: **QDMA H2C-Streaming Bypass Input Interface Descriptions** *(cont'd)*

Port Name	I/O	Description
dma0_h2c_byp_in_st_no_dma	I	H2C Bypass In No DMA When sending a descriptor through the 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 signal 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 dma0_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 dma0_h2c_byp_in_st_no_dma is set, both dma0_h2c_byp_in_st_sop and dma0_h2c_byp_in_st_eop must be set. If dma0_h2c_byp_in_st_no_dma is set, the QDMA ignores the address and length fields of this interface.
dma0_h2c_byp_in_st_qid[10:0]	I	The QID associated with the H2C descriptor ring.
dma0_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
dma0_h2c_byp_in_st_func[7:0]	I	PCIe function ID
dma0_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.
dma0_h2c_byp_in_st_port_id[2:0]	I	QDMA port ID
dma0_h2c_byp_in_st_vld	I	Valid. High indicates descriptor is valid. One pulse for one descriptor.
dma0_h2c_byp_in_st_rdy	O	Ready to take in descriptor

Table 33: **QDMA H2C-MM Descriptor Bypass Input Port Descriptions**

Port Name	I/O	Description
dma0_h2c_byp_in_mm_radr[63:0]	I	The read address for the DMA data.
dma0_h2c_byp_in_mm_wadr[63:0]	I	The write address for the dma data.
dma0_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.

Table 33: **QDMA H2C-MM Descriptor Bypass Input Port Descriptions** *(cont'd)*

Port Name	I/O	Description
dma0_h2c_byp_in_mm_sdi	I	H2C-MM Bypass In Status Descriptor/Interrupt If set, the signal is treated as an indication from the user logic to the QDMA to send the status descriptor to the host and generate an interrupt to the 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 dma0_h2c_byp_in_mm_sdi has an error. This results in a missing writeback, and the hw_ctxt.dsc_pend bit is asserted indefinitely. The workaround is to send a zero length descriptor to trigger the Completion (CMPT) Status.
dma0_h2c_byp_in_mm_mrkr_req	I	H2C-MM Bypass In Completion Request Indication from the user logic that the QDMA must send a completion status to the user logic after the QDMA has completed the data transfer of this descriptor.
dma0_h2c_byp_in_mm_qid[10:0]	I	The QID associated with the H2C descriptor ring.
dma0_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.
dma0_h2c_byp_in_mm_func[7:0]	I	PCIe function ID
dma0_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.
dma0_h2c_byp_in_mm_port_id[2:0]	I	QDMA port ID
dma0_h2c_byp_in_mm_vld	I	Valid. High indicates descriptor is valid, one pulse for one descriptor.
dma0_h2c_byp_in_mm_rdy	O	Ready to take in descriptor

Table 34: **QDMA C2H-Streaming Cache Bypass Input Port Descriptions**

Port Name	I/O	Description
dma0_c2h_byp_in_st_csh_addr [63:0]	I	64 bit address where DMA writes data.
dma0_c2h_byp_in_st_csh_qid [10:0]	I	The QID associated with the C2H descriptor ring.
dma0_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.
dma0_c2h_byp_in_st_csh_func [7:0]	I	PCIe function ID
dma0_c2h_byp_in_st_csh_port_id[2:0]	I	QDMA port ID
dma0_c2h_byp_in_st_csh_vld	I	Valid. High indicates descriptor is valid, one pulse for one descriptor.
dma0_c2h_byp_in_st_csh_rdy	O	Ready to take in descriptor.

Table 35: **QDMA C2H-Streaming Simple Bypass Input Port Descriptions**

Port Name	I/O	Description
dma0_c2h_byp_in_st_sim_addr [63:0]	I	64-bit address where DMA writes data.
dma0_c2h_byp_in_st_sim_qid [10:0]	I	The QID associated with the C2H descriptor ring.
dma0_c2h_byp_in_st_sim_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.
dma0_c2h_byp_in_st_sim_func [7:0]	I	PCIe function ID
dma0_c2h_byp_in_st_sim_port_id[2:0]	I	QDMA port ID
dma0_c2h_byp_in_st_sim_vld	I	Valid. High indicates descriptor is valid. One pulse for one descriptor.
dma0_c2h_byp_in_st_sim_rdy	O	Ready to take in descriptor.

Table 36: **QDMA C2H-MM Descriptor Bypass Input Port Descriptions**

Port Name	I/O	Description
dma0_c2h_byp_in_mm_raddr [63:0]	I	The read address for the DMA data.
dma0_c2h_byp_in_mm_wadr[63:0]	I	The write address for the DMA data.
dma0_c2h_byp_in_mm_len[27:0]	I	The DMA data length.
dma0_c2h_byp_in_mm_sdi	I	C2H Bypass In Status Descriptor/Interrupt If set, it is treated as an indication from the user logic to the 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.
dma0_c2h_byp_in_mm_mrkr_req	I	C2H Bypass In Marker Request Indication from the user logic that the QDMA must send a completion status to the user logic after the QDMA has completed the data transfer of this descriptor.
dma0_c2h_byp_in_mm_qid [10:0]	I	The QID associated with the C2H descriptor ring.
dma0_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.
dma0_c2h_byp_in_mm_func [7:0]	I	PCIe function ID
dma0_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.
dma0_c2h_byp_in_mm_port_id[2:0]	I	QDMA port ID
dma0_c2h_byp_in_mm_vld	I	Valid. High indicates descriptor is valid. One pulse for one descriptor.
dma0_c2h_byp_in_mm_rdy	O	Ready to take in descriptor.

QDMA Descriptor Bypass Output Interface

Table 37: **QDMA H2C Descriptor Bypass Output Interface Descriptions**

Port Name	I/O	Description
dma0_h2c_byp_out_dsc[255:0]	O	The H2C descriptor fetched from the host. For Streaming descriptor, use the lower 64b of this field as the address. The remaining bits can be ignored. For H2C AXI-MM, the QDMA uses all 256 bits, and the structure of the bits are the same as this table. For H2C AXI-ST, the QDMA uses [127:0] bits, and the structure of the bits are the same as this table.
dma0_h2c_byp_out_st_mm	O	Indicates whether this is a streaming data descriptor or memory-mapped descriptor. 0: Streaming 1: Memory-mapped
dma0_h2c_byp_out_dsc_sz[1:0]	O	Descriptor size. This field indicates the amount of valid descriptor information on dma0_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 .
dma0_h2c_byp_out_qid[10:0]	O	The QID associated with the H2C descriptor ring.
dma0_h2c_byp_out_error	O	Indicates that an error was encountered in descriptor fetch or execution of a previous descriptor.
dma0_h2c_byp_out_func[7:0]	O	PCIe function ID
dma0_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.
dma0_h2c_byp_out_port_id[2:0]	O	QDMA port ID
dma0_h2c_byp_out_mrkr_rsp	O	Indicates completion status in response to h2c_byp_in_st_mrkr_req (Stream) or h2c_byp_in_mm_mrkr_req (MM).
dma0_h2c_byp_out_vld	O	Valid. High indicates descriptor is valid, one pulse for one descriptor.
dma0_h2c_byp_out_rdy	I	Ready. When this interface is not used, Ready must be tied-off to 1.

Table 38: **QDMA C2H Descriptor Bypass Output Port Descriptions**

Port Name	I/O	Description
dma0_c2h_byp_out_dsc[255:0]	O	The C2H descriptor fetched from the host. For C2H AXI-MM, the QDMA uses all 256 bits, and the structure of the bits is the same as this table. For C2H AXI-ST, the QDMA uses [63:0] bits, and the structure of the bits is the same as this table. The remaining bits are ignored.

Table 38: **QDMA C2H Descriptor Bypass Output Port Descriptions** *(cont'd)*

Port Name	I/O	Description
dma0_c2h_byp_out_st_mm	O	Indicates whether this is a streaming data descriptor or memory-mapped descriptor. 0: streaming 1: memory-mapped
dma0_c2h_byp_out_dsc_sz[1:0]	O	Descriptor size. This field indicates the amount of valid descriptor information on dma0_h2c_byp_out_dsc. 0: 8B 1: 16B 2: 32B 3: 64B to 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.
dma0_c2h_byp_out_qid[10:0]	O	The QID associated with the H2C descriptor ring.
dma0_c2h_byp_out_error	O	Indicates that an error was encountered in descriptor fetch or execution of a previous descriptor.
dma0_c2h_byp_out_func[7:0]	O	PCIe function ID.
dma0_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.
dma0_c2h_byp_out_port_id[2:0]	O	QDMA port ID
dma0_c2h_byp_out_mrkr_rsp	O	Indicates completion status in response to s_axis_c2h_ctrl_marker (Stream) or c2h_byp_in_mm_mrkr_req (MM). For the completions status for `dma0_s_axis_c2h_ctrl_marker` (Stream), the details are given in the table below.
dma0_c2h_byp_out_vld	O	Valid. High indicates descriptor is valid, one pulse for one descriptor.
dma0_c2h_byp_out_rdy	I	Ready. When this interface is not used, Ready must be tied-off to 1.

Table 39: **QDMA C2H Descriptor Bypass out Marker Response Description**

Field Name	location	Description
err[1:0]	[1:0]	Error code reported by the C2H 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
retry_marker_req	[2]	The marker request could not be completed because 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.
rsv	[255:3]	Reserved

It is common for `dma0_h2c_byp_out_vld` or `dma0_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 the 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 Interface

Table 40: **QDMA Descriptor Credit Input Port Descriptions**

Port Name	I/O	Description
dma0_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.
dma0_dsc_crdt_in_rdy	O	Ready. Assertion of this signal indicates the DMA is ready to accept data from this bus.
dma0_dsc_crdt_in_dir	I	Indicates whether credits are for H2C or C2H descriptor ring. 0: H2C 1: C2H
dma0_dsc_crdt_in_qid [10:0]	I	The QID associated with the descriptor ring for the credits are being added.
dma0_dsc_crdt_in_crdt [15:0]	I	The number of descriptor credits that the user application is giving to the QDMA to fetch descriptors from the host.

QDMA Traffic Manager Credit Output Interface

Table 41: **QDMA TM Credit Output Port Descriptions**

Port Name	I/O	Description
dma0_tm_dsc_sts_vld	O	Valid. Indicates valid data on the output bus. Valid data on the bus is held until dma0_tm_dsc_sts_rdy is asserted by the user.
dma0_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.
dma0_tm_dsc_sts_byp	O	Shows the bypass bit in the SW descriptor context.
dma0_tm_dsc_sts_dir	O	Indicates whether the status update is for a H2C or C2H descriptor ring. 0: H2C 1: C2H
dma0_tm_dsc_sts_mm	O	Indicates whether the status update is for a streaming or memory-mapped queue. 0: Streaming 1: Memory-mapped
dma0_tm_dsc_sts_qid [10:0]	O	The QID of the ring

Table 41: **QDMA TM Credit Output Port Descriptions** *(cont'd)*

Port Name	I/O	Description
dma0_tm_dsc_sts_avl [15:0]	O	If dma0_tm_dsc_sts_qinv is set, this is the number of credits available in the descriptor engine. If dma0_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.
dma0_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.
dma0_tm_dsc_sts_qen	O	The current queue enable status.
dma0_tm_dsc_sts_irq_arm	O	If set, it indicates that the driver is ready to accept interrupts.
dma0_tm_dsc_sts_error	O	Set to 1 if the PIDX update is rolled over the current CIDX of associated. queue.
dma0_tm_dsc_sts_port_id [2:0]	O	The port id associated with the queue from the queue context.

User Interrupts

Table 42: **User Interrupts Port Descriptions**

Port Name	I/O	Description
dma0_usr_irq_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, dma0_usr_irq_in_vld must remain high until dma0_usr_irq_ack is asserted by the DMA.
dma0_usr_irq_vec [4:0]	I	Vector The MSI-X vector to be sent.
dma0_usr_irq_fnc [7:0]	I	Function The function of the vector to be sent.
dma0_usr_irq_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.
dma0_usr_irq_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 number allowed per function.

Register Space

QDMA PF Address Register Space

All the physical function (PF) registers are found in `cpm-qdma-v2-1-registers.csv` available in the register map files.

To locate the register space information:

1. Download the register map files from the Xilinx website.
2. Extract the ZIP file contents into any write-accessible location.
3. Refer to `cpm-qdma-v2-1-registers.csv`.

Table 43: **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 `cpm-qdma-v2-1-registers.csv`.
QDMA_TRQ_MSIX	0x2000	512	Also found in QDMA_TRQ_MSIX (0x2000). Maximum of 32 vectors per function.
QDMA_PF_MAILBOX	0x2400	16384	Also found in QDMA_PF_MAILBOX (0x2400).
QDMA_TRQ_SEL_QUEUE_PF	0x6400	32768	Also found in QDMA_TRQ_SEL_QUEUE_PF (0x6400).

QDMA_CSR (0x0000)

QDMA Configuration Space Register (CSR) descriptions are found in `cpm-qdma-v2-1-registers.csv`. See above for details.

QDMA_TRQ_MSIX (0x2000)

Table 44: **QDMA_TRQ_MSIX (0x2000)**

Byte Offset	Bit	Default	Access Type	Field	Description
0x2000	[31:0]	0	NA	addr	MSIX_Vector0_Address[63:32] MSI-X vector0 message lower address.
0x2004	[31:0]	0	RO	addr	MSIX_Vector0_Address[63:32] MSI-X vector0 message upper address.
0x2008	[31:0]	0	RO	data	MSIX_Vector0_Data[31:0] MSI-X vector0 message data.

Table 44: **QDMA_TRQ_MSIX (0x2000)** *(cont'd)*

Byte Offset	Bit	Default	Access Type	Field	Description
0x200C	[31:0]	0	RO	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.

The MSI-X table PBA offset is at 0x1400.

***Note*:** The table above represents one MSI-X table entry 0. Each function can only support up to 32 vectors.

QDMA_PF_MAILBOX (0x2400)

Table 45: **QDMA_PF_MAILBOX (0x2400) Register Space**

Register	Address	Description
Function Status Register (0x2400)	0x2400	Status bits
Function Command Register (0x2404)	0x2404	Command register bits
Function Interrupt Vector Register (0x2408)	0x2408	Interrupt vector register
Target Function Register (0x240C)	0x240C	Target Function register
Function Interrupt Vector Register (0x2410)	0x2410	Interrupt Control Register
RTL Version Register (0x2414)	0x2414	RTLVersion Register
PF Acknowledgment Registers (0x2420-0x243C)	0x2420-0x243C	PF acknowledge
FLR Control/Status Register (0x2500)	0x2500	FLR control and status
Incoming Message Memory (0x2C00-0x2C7C)	0x2C00-0x2C7C	Incoming message (128 bytes)
Outgoing Message Memory (0x3000-0x307C)	0x3000-0x307C	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 (0x2400)

Table 46: **Function Status Register (02400)**

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 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 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, this can cause an error in the AXI4-Lite 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 (0x2404)

Table 47: **Function Command Register (0x2404)**

Bit	Default	Access Type	Field	Description
[31:3]	0	NA	Reserved	Reserved
[2]	0	RO	Reserved	Reserved

Table 47: **Function Command Register (0x2404)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[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.
[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 (0x2408)

Table 48: **Function Interrupt Vector Register (0x2408)**

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 (0x240C)

Table 49: **Target Function Register (0x240C)**

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.

Function Interrupt Vector Register (0x2410)

Table 50: **Function Interrupt Vector Register (0x2410)**

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

RTL Version Register (0x2414)

Table 51: **RTL Version Register (0x2414)**

Bit	Default	Access Type	Field	Description
[31:16]	0x1fd3	RO		QDMA ID
[15:0]	0	RO		Vivado versions 0x1000: CPM QDMA Vivado version 2020.1.

PF Acknowledgment Registers (0x2420-0x243C)

Table 52: **PF Acknowledgment Registers (0x2420-0x243C)**

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 (0x2500)

Table 53: **FLR Control/Status Register (0x2500)**

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 (0x2C00-0x2C7C)

Table 54: **Incoming Message Memory (0x2C00-0x2C7C)**

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 (0x3000-0x307C)

Table 55: **Outgoing Message Memory (0x3000-0307C)**

Register	Addr	Default	Access Type	Field	Width	Description
o_msg_i	0x3000 + i *4	0	RW		32	The *i*th word of the outgoing message (0 ≤ I < 128).

QDMA_TRQ_SEL_QUEUE_PF (0x6400)

Table 56: **QDMA_TRQ_SEL_QUEUE_PF (0x6400) Register Space**

Register	Address	Description
QDMA_DMAP_SEL_INT_CIDX[2048] (0x6400)	0x6400-0xB3F0	Interrupt Ring Consumer Index (CIDX)
QDMA_DMAP_SEL_H2C_DSC_PIDX[2048] (0x6404)	0x6404-0xB3F4	H2C Descriptor Producer index (PIDX)
QDMA_DMAP_SEL_C2H_DSC_PIDX[2048] (0x6408)	0x6408-0xB3F8	C2H Descriptor Producer Index (PIDX)
QDMA_DMAP_SEL_CMPT_CIDX[2048] (0x640C)	0x640C-0xB3FC	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 = 0x6400 + Qnumber*16
> H2C PIDX address = 0x6404 + Qnumber*16

C2H PIDX address = 0x6408 + Qnumber*16
Write Back CIDX address = 0x640C + Qnumber*16

For Queue 0:

0x6400 correspond to QDMA_DMAP_SEL_INT_CIDX
0x6404 correspond to QDMA_DMAP_SEL_H2C_DSC_PIDX
0x6408 correspond to QDMA_DMAP_SEL_C2H_DSC_PIDX
0x640C correspond to QDMA_DMAP_SEL_WRB_CIDX

For Queue 1:

0x6410 correspond to QDMA_DMAP_SEL_INT_CIDX
0x6414 correspond to QDMA_DMAP_SEL_H2C_DSC_PIDX
0x6418 correspond to QDMA_DMAP_SEL_C2H_DSC_PIDX
0x641C correspond to QDMA_DMAP_SEL_WRB_CIDX

For Queue 2:

0x6420 correspond to QDMA_DMAP_SEL_INT_CIDX
0x6424 correspond to QDMA_DMAP_SEL_H2C_DSC_PIDX
0x6428 correspond to QDMA_DMAP_SEL_C2H_DSC_PIDX
0x642C correspond to QDMA_DMAP_SEL_WRB_CIDX

QDMA_DMAP_SEL_INT_CIDX[2048] (0x6400)

Table 57: **QDMA_DMAP_SEL_INT_CIDX[2048] (0x6400)**

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] (0x6404)

Table 58: **QDMA_DMAP_SEL_H2C_DSC_PIDX[2048] (0x6404)**

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] (0x6408)

Table 59: **QDMA_DMAP_SEL_C2H_DSC_PIDX[2048] (0x6408)**

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] (0x640C)

Table 60: **QDMA_DMAP_SEL_CMPT_CIDX[2048] (0x640C)**

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 VF Address Register Space

Table 61: **QDMA VF Address Register Space**

Target Name	Base (Hex)	Byte Size (Dec)	Notes
QDMA_TRQ_SEL_QUEUE_VF (0x3000)	00003000	32768	VF Direct QCSR (16B per Queue, up to max of 2048 Queue per function)
QDMA_TRQ_MSIX_VF (0x400)	00004000	4096	Space for 32 MSI-X vectors and PBA
QDMA_VF_MAILBOX (0x1000)	00001000	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 (0x6400).

This set of registers can be accessed based on Queue number. Queue number is absolute Qnumber, [0 to 2047].

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 (0x400)

VF functions can access the MSI-X table with offset (0x0000) from that function. The description for this register space is the same as QDMA_TRQ_MSIX (0x2000).

QDMA_VF_MAILBOX (0x1000)

Table 62: **QDMA_VF_MAILBOX (0x0100) Register Space**

Registers (Address)	Address	Description
Function Status Register (0x1000)	0x1000	Status register bits
Function Command Register (0x1004)	0x1004	Command register bits
Function Interrupt Vector Register (0x1008)	0x1008	Interrupt vector register
Target Function Register (0x100C)	0x100C	Target Function register
Function Interrupt Control Register (0x1010)	0x1010	Interrupt Control Register
RTL Version Register (0x1014)	0x1014	RTL Version Register
Incoming Message Memory (0x1800-0x187C)	0x1800-0x187C	Incoming message (128 bytes)
Outgoing Message Memory (0x1C00-0x1C7C)	0x1C00-0x1C7C	Outgoing message (128 bytes)

Function Status Register (0x1000)

Table 63: **Function Status Register (0x1000)**

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, this can cause 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, this can cause an error in the AXI4-Lite 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 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 (0x1004)

Table 64: **Function Command Register (0x1004)**

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.

Table 64: **Function Command Register (0x1004)** *(cont'd)*

Bit Index	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. The hardware pushes the message to the event queue of the PF with target_fn_id.

Function Interrupt Vector Register (0x1008)

Table 65: **Function Interrupt Vector Register (0x1008)**

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 (0x100C)

Table 66: **Target Function Register (0x100C)**

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 (0x1010)

Table 67: **Function Interrupt Control Register (0x1010)**

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

RTL Version Register (0x1014)

Table 68: **RTL Version Register (0x1014)**

Bit	Default	Access Type	Field	Description
[31:16]	0x1fd3	RO		QDMA ID
[15:0]	0	RO		Vivado versions 0x1000: CPM QDMA Vivado version 2020.1.

Incoming Message Memory (0x1800-0x187C)

Table 69: **Incoming Message Memory (0x1800-0x187C)**

Register	Addr	Default	Access Type	Field	Width	Description
i_msg_i	0x1800 + i*4	0	RW		32	The *i*th word of the incoming message (i < 128).

Outgoing Message Memory (0x1C00-0x1C7C)

Table 70: **Outgoing Message Memory (0x1C00-0x1C7C)**

Register	Addr	Default	Access Type	Field	Width	Description
o_msg_i	0x1C00 + i *4	0	RW		32	The *i*th word of the outgoing message (i < 128).

AXI Slave Register Space

DMA register space can be accessed using AXI Slave interface. When AXI Slave Bridge mode is enabled (based on GUI settings) user can also access Bridge registers and can also access Host memory space.

Table 71: **AXI4 Slave Register Space**

Register Space	AXI Slave Interface Address range	Details
Bridge registers	0x6_0000_0000	Described in Bridge register space CSV file. See Bridge Register Space for details.
DMA registers	0x6_1000_0000	Described in QDMA PF Address Register Space and QDMA VF Address Register Space.
Slave Bridge access to Host memory space	0xE001_0000 - 0xEFFF_FFFF 0x6_1100_0000 - 0x7_FFFF_FFFF 0x80_0000_0000 - 0xBF_FFFF_FFFF	Address range for Slave bridge access is set during IP customization in the Address Editor tab of the Vivado IDE.

Bridge Register Space

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

All the bridge registers are listed in the `cpm-bridge-v2-1-registers.csv` available in the register map files.

To locate the register space information:

1. Download the register map files from the Xilinx website.
2. Extract the ZIP file contents into any write-accessible location.
3. Refer to `cpm-bridge-v2-1-registers.csv`.

DMA Register Space

The DMA register space is described in the following sections:

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

Chapter 4

Design Flow Steps

This section describes customizing and generating the functional mode, constraining the functional mode, and the simulation, synthesis, and implementation steps that are specific to this IP functional mode. 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)

QDMA AXI MM Interface to NoC and DDR Lab

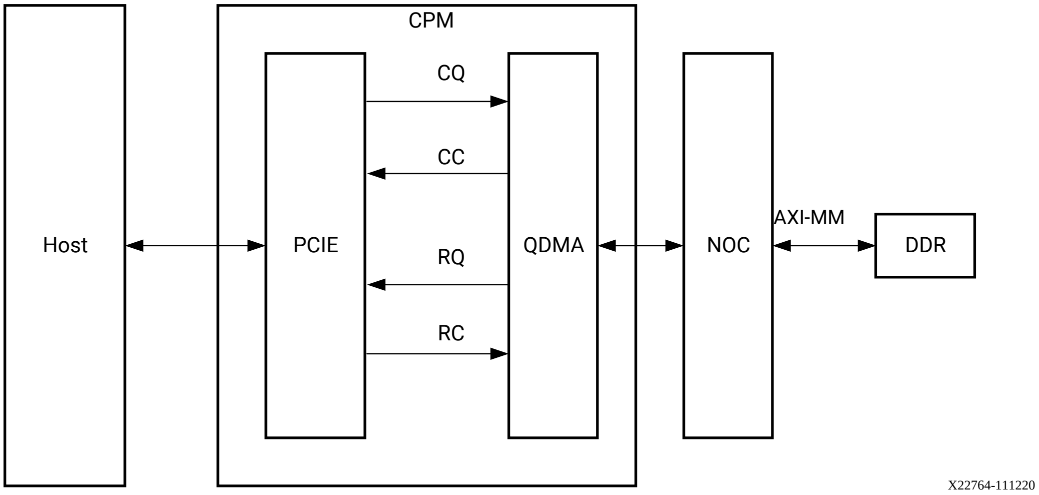
This lab describes the process of generating a Versal™ ACAP QDMA design with AXI4 Memory Mapped interface connected to network on chip (NoC) IP and DDR memory. This design has the following configurations:

- AXI4 memory mapped (AXI MM) connected to DDR through the NoC IP
- Gen3 x 16
- 4 physical functions (PFs) and 252 virtual functions (VFs)
- MSI-X interrupts

This lab provides step by step instructions to configure a Control, Interfaces and Processing System (CIPS) QDMA design and network on chip (NoC) IP. The following figure shows the AXI4 Memory Mapped (AXI-MM) interface to DDR using the NoC IP. At the end of this lab, you can synthesize and implement the design, and generate a Programmable Device Image (PDI) file. The PDI file is used to program the Versal ACAP and run data traffic on a system. For the AXI-MM interface host to chip (H2C) transfers, data is read from Host and sent to DDR memory. For chip to host (C2H) transfers, data is read from DDR memory and written to host.

This lab targets a xcvc1902-vsvd1760-1LP-e-S-es1 part on a VCK5000 board. This lab connects to DDR memory found outside the ACAP. A constraints file is provided and added to the design during the lab. The constraints file lists all DDR pins and their placement. You can modify the constraint file based on your requirements and DDR part selection.

Figure 30: **AXI4 Memory Mapped to DDR Design**



Tutorial Design File

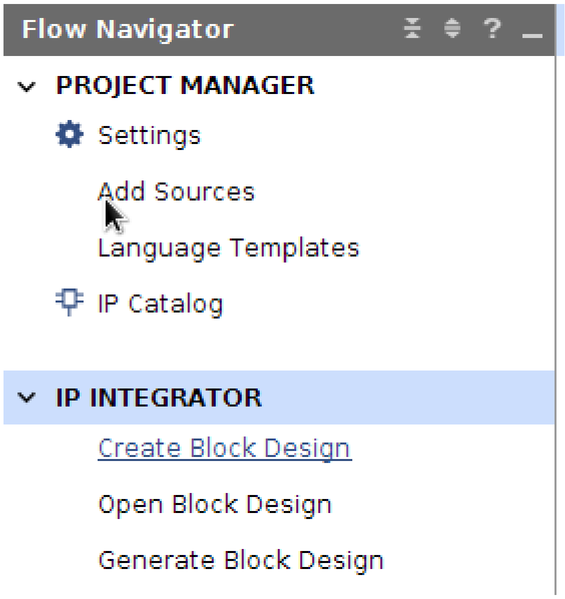
Before running the lab, download the `top_impl.xdc` constraints file available in the reference design file. To do so:

1. Download the reference design file from the Xilinx website.
2. Extract the ZIP file contents into any write-accessible location.
3. Locate the `top_impl.xdc` constraints file.

The provided `top_impl.xdc` constraints file contains the needed DDR pins and their placement for this tutorial lab. The constraints file can be modified as needed for later use.

Start the Vivado Design Suite

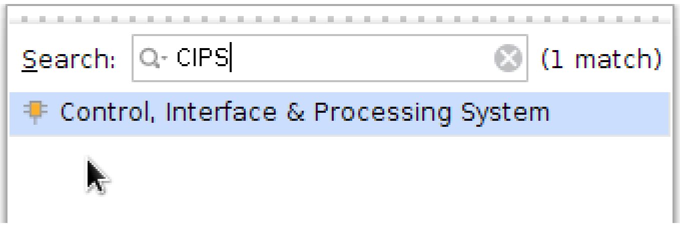
1. Open the Vivado® Design Suite.
2. Click **Create Project** from the Quick Start Menu.
3. Step through the popup menus to access the Default Part page.
4. In the Default Part page, search for and select: **xcvc1902-vsvd1760-1LP-e-S-es1**.
5. Continue to the Finish stage to create the new project and open Vivado.
6. In the Vivado Flow Navigator, click **IP Integrator → Create Block Design**. A popup dialog displays to create the block design.



7. Click **OK**. An empty block design diagram canvas opens.

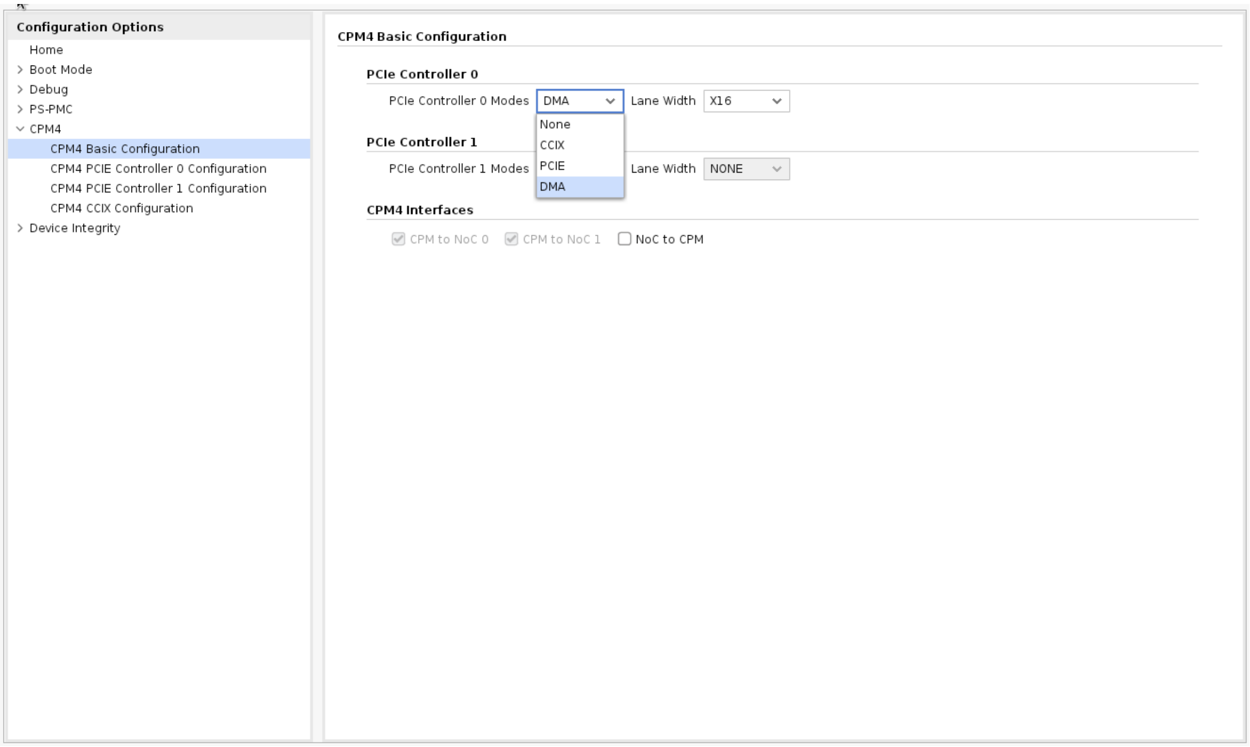
Instantiate the CIPS IP

1. Right-click on the block design canvas, and from the context menu select **Add IP**.
2. The IP catalog pops up. In the Search field type `CIPS` to filter to the list of IP.



3. From the filtered list, double-click the **Control, Interface, and Processing System** IP core to instantiate the IP on the block design canvas.
4. This adds the Versal CIPS IP to the canvas. Double-check **Versal CIPS IP**.
5. The configuration dialog box for the Control, Interfaces and Processing System IP core displays. In the Configuration Options pane, expand **CPM**, and click **CPM Configuration**.
6. Set the PCIe0 Modes to **DMA**, and set the lane width to **X16**.

   Available lane widths are X4, X8 and X16. X1 and X2 are not supported.



CPM4 Basic Configuration

PCIe Controller 0

PCIe Controller 0 Modes: DMA; Lane Width: X16

PCIe Controller 1

PCIe Controller 1 Modes; Lane Width: NONE

CPM4 Interfaces

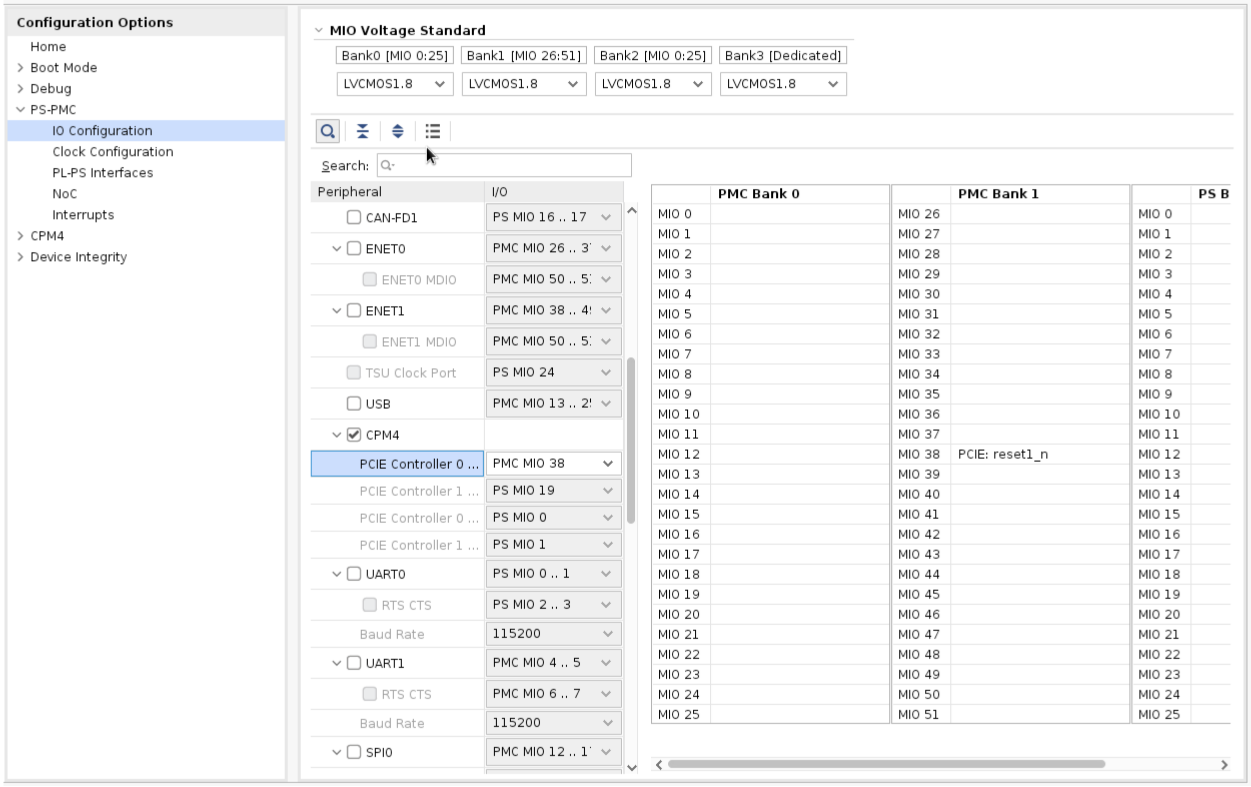
☑ CPM to NoC 0 ☑ CPM to NoC 1 □ NoC to CPM

7. In the Configuration Options pane, expand **PS-PMC**, and click **IO Configuration**.
8. The IO Configuration page displays with a list of options to configure the CPM-PCIe functional mode. In the Peripheral column, select the **PCIe Reset** checkbox.

   Notice that only A0 End Point is selectable in the I/O column.

   Notice also that the multi-use I/O (MIO) pin selected in PCIe reset is automatically connected to the PCIe Reset I/O, in this case MIO 38.
9. Next to A0 End Point, select **PS MIO 38**, which is the MIO pin that matches the MIO pin is connected in your board.

   Available MIO pin selections are PS MIO 18, PMC MIO 24, and PMC MIO 38.



Configuration Options

- Home
- Boot Mode
- Debug
- PS-PMC
  - IO Configuration
  - Clock Configuration
  - PL-PS Interfaces
  - NoC
  - Interrupts
- CPM4
- Device Integrity

MIO Voltage Standard

Bank0 [MIO 0:25]	Bank1 [MIO 26:51]	Bank2 [MIO 0:25]	Bank3 [Dedicated]
LVCMOS1.8	LVCMOS1.8	LVCMOS1.8	LVCMOS1.8

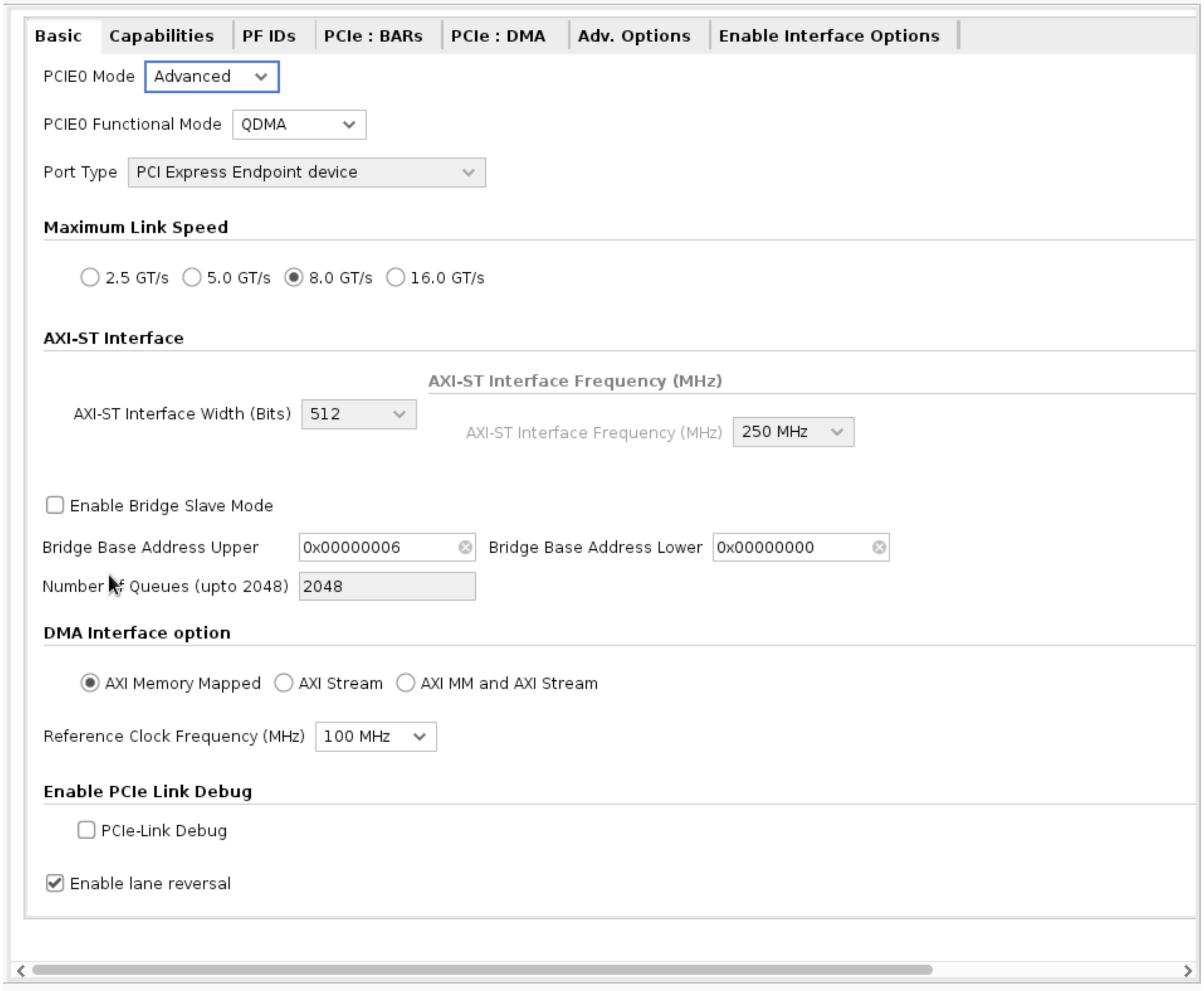
Search:

Peripheral	I/O
□ CAN-FD1	PS MIO 16 .. 17
□ ENET0	PMC MIO 26 .. 3
□ ENET0 MDIO	PMC MIO 50 .. 5
□ ENET1	PMC MIO 38 .. 4
□ ENET1 MDIO	PMC MIO 50 .. 5
□ TSU Clock Port	PS MIO 24
□ USB	PMC MIO 13 .. 2
☑ CPM4	
PCIE Controller 0 ...	PMC MIO 38
PCIE Controller 1 ...	PS MIO 19
PCIE Controller 0 ...	PS MIO 0
PCIE Controller 1 ...	PS MIO 1
□ UART0	PS MIO 0 .. 1
□ RTS CTS	PS MIO 2 .. 3
Baud Rate	115200
□ UART1	PMC MIO 4 .. 5
□ RTS CTS	PMC MIO 6 .. 7
Baud Rate	115200
□ SPI0	PMC MIO 12 .. 1

PMC Bank 0		PMC Bank 1		PS B	
MIO 0		MIO 26		MIO 0	
MIO 1		MIO 27		MIO 1	
MIO 2		MIO 28		MIO 2	
MIO 3		MIO 29		MIO 3	
MIO 4		MIO 30		MIO 4	
MIO 5		MIO 31		MIO 5	
MIO 6		MIO 32		MIO 6	
MIO 7		MIO 33		MIO 7	
MIO 8		MIO 34		MIO 8	
MIO 9		MIO 35		MIO 9	
MIO 10		MIO 36		MIO 10	
MIO 11		MIO 37		MIO 11	
MIO 12		MIO 38	PCIE: reset1_n	MIO 12	
MIO 13		MIO 39		MIO 13	
MIO 14		MIO 40		MIO 14	
MIO 15		MIO 41		MIO 15	
MIO 16		MIO 42		MIO 16	
MIO 17		MIO 43		MIO 17	
MIO 18		MIO 44		MIO 18	
MIO 19		MIO 45		MIO 19	
MIO 20		MIO 46		MIO 20	
MIO 21		MIO 47		MIO 21	
MIO 22		MIO 48		MIO 22	
MIO 23		MIO 49		MIO 23	
MIO 24		MIO 50		MIO 24	
MIO 25		MIO 51		MIO 25	

CPM Configuration

1. In the Configuration Options pane, expand **CPM**, and click **PCIE0 Configuration** to customize the PCIe Port 0.
2. In the Basic tab, set the following options:
   - CPM Modes: **Advanced**.
   - PCIE0 Functional Mode: **QDMA**.
   - Maximum Link Speed: **8.0 GT/s** (Gen3).
   - DMA Interface option: **AXI Memory Mapped**.



PCIE0 Mode: Advanced

PCIE0 Functional Mode: QDMA

Port Type: PCI Express Endpoint device

Maximum Link Speed

○ 2.5 GT/s ○ 5.0 GT/s ● 8.0 GT/s ○ 16.0 GT/s

AXI-ST Interface

AXI-ST Interface Width (Bits): 512

AXI-ST Interface Frequency (MHz)

AXI-ST Interface Frequency (MHz): 250 MHz

□ Enable Bridge Slave Mode

Bridge Base Address Upper: 0x00000006 Bridge Base Address Lower: 0x00000000

Number of Queues (upto 2048): 2048

DMA Interface option

● AXI Memory Mapped ○ AXI Stream ○ AXI MM and AXI Stream

Reference Clock Frequency (MHz): 100 MHz

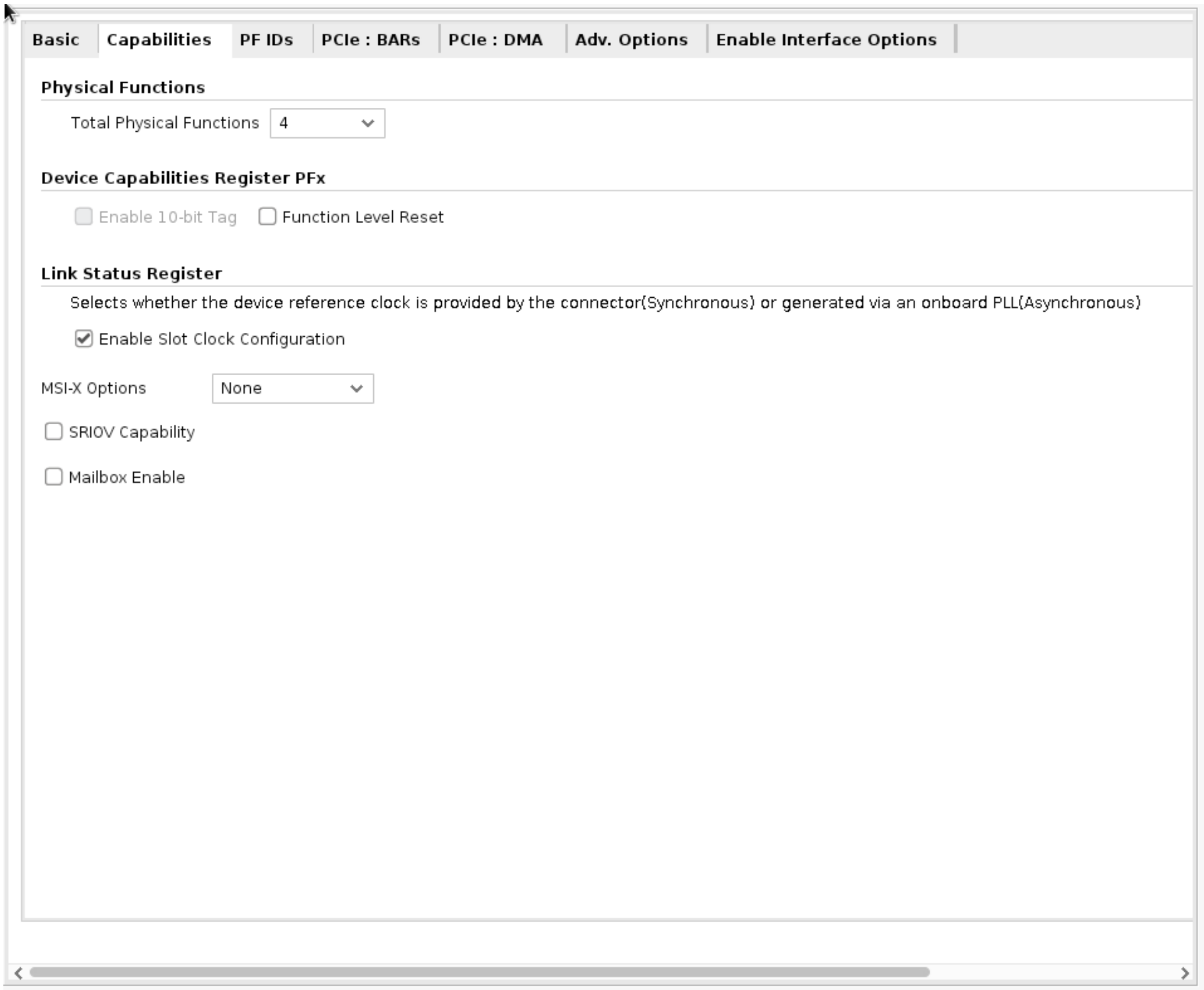
Enable PCIe Link Debug

□ PCIe-Link Debug

☑ Enable lane reversal

3. In the Capabilities tab, set the following option:
   - Total Physical Functions: **4**
   - MSI-X Options: **MSI-X Internal**

     This option enables the CPM QDMA in MSI-X internal mode.



Physical Functions

Total Physical Functions 4

Device Capabilities Register PFx

□ Enable 10-bit Tag □ Function Level Reset

Link Status Register

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

☑ Enable Slot Clock Configuration

MSI-X Options None

□ SRIOV Capability

□ Mailbox Enable

4. In the PF ID tab, there are 4 PFs listed with device ID. Based on your need, you can modify the device ID. For this lab we will keep the default device ID.

5. In the PCIe: BAR tab, set the following options:

   First row (for BAR0):

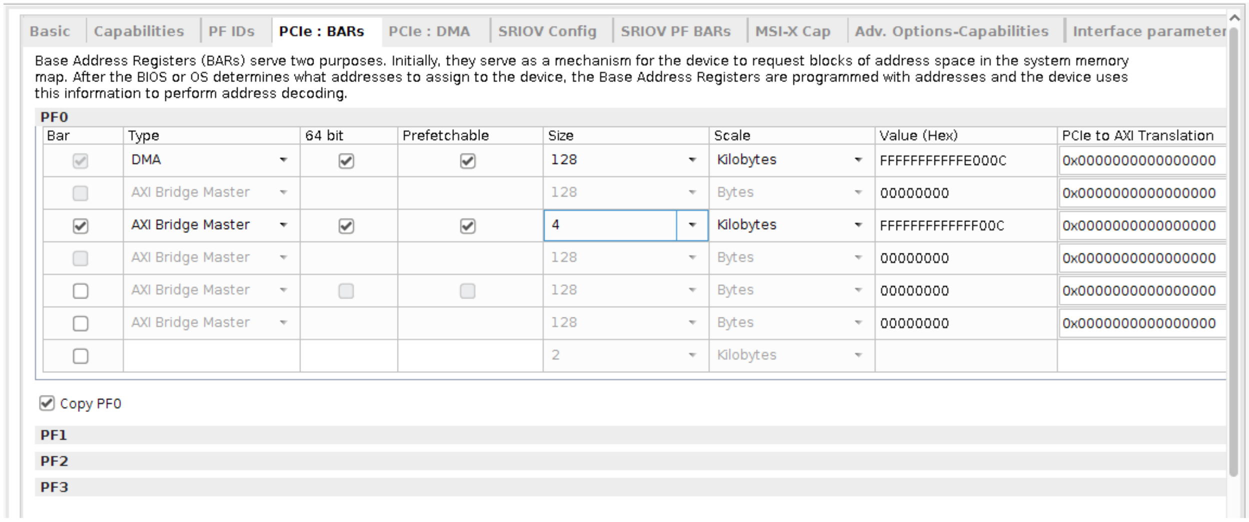
   - Select the **Bar** checkbox.
   - Set type to **DMA**.
   - Select the **64 bit** checkbox.
   - Select the **Prefetchable** checkbox.
   - Set size to **128 Kilobytes**.

   Second row (for BAR2):

   - Select the **Bar** checkbox.
   - Set type to **AXI Bridge Master**.

- Select the **64 bit** checkbox.
- Select the **Prefetchable** checkbox.
- Set size to **4 Kilobytes**.

The same Bar options can be copied for all 4 PFs. Depending on your needs, you can modify the BAR selection for all PFs. For this lab, we will copy PF10 selection to all 3 PFs. To do so, click **Copy PF0**.



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	☑	☑	128	Kilobytes	FFFFFFFFFFFE000C	0x0000000000000000
□	AXI Bridge Master			128	Bytes	00000000	0x0000000000000000
☑	AXI Bridge Master	☑	☑	4	Kilobytes	FFFFFFFFFFFFF00C	0x0000000000000000
□	AXI Bridge Master			128	Bytes	00000000	0x0000000000000000
□	AXI Bridge Master	□	□	128	Bytes	00000000	0x0000000000000000
□	AXI Bridge Master			128	Bytes	00000000	0x0000000000000000
□				2	Kilobytes		

☑ Copy PF0

PF1

PF2

PF3

6. In the PCIe: DMA tab keep all default selections.
7. Click **OK** to generate the CIPS QDMA IP.

NoC Configuration

Next you will add and configure a Network on Chip (NoC) IP core for the DDR connection.

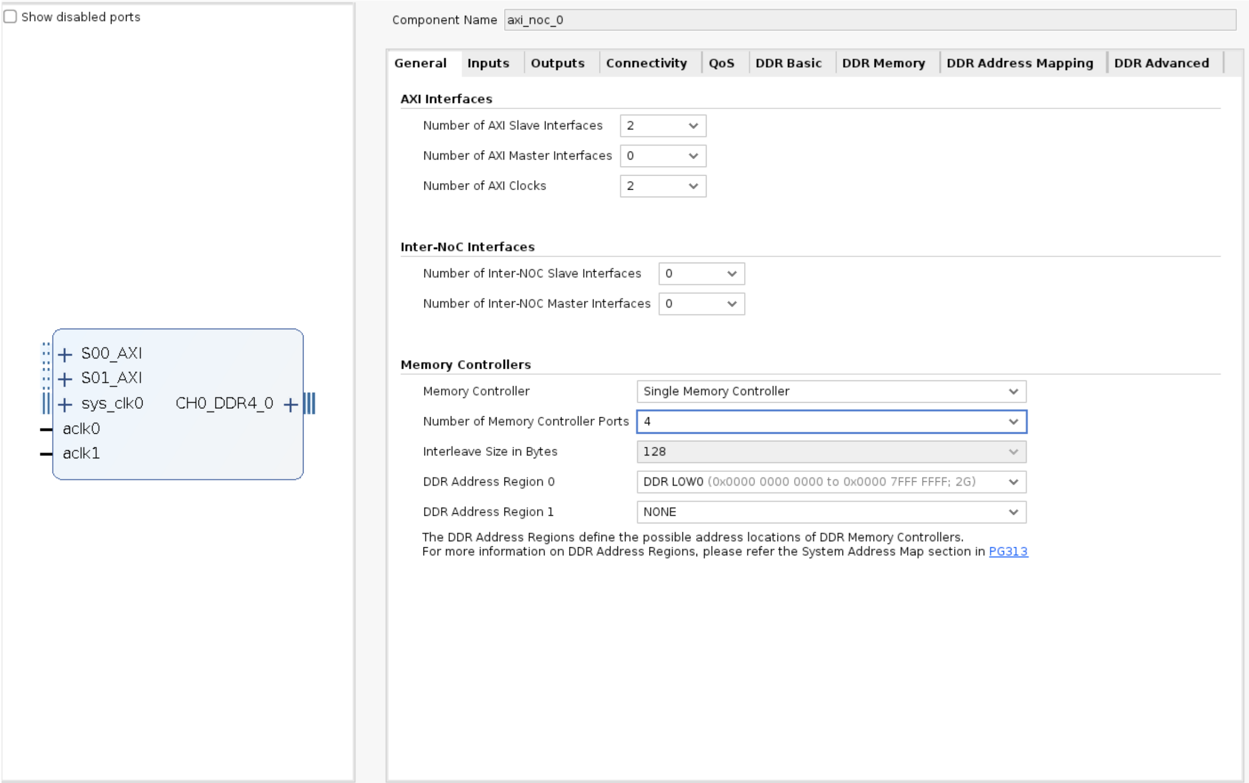
1. Right-click on the block design canvas and from the context menu select **Add IP**.
2. The IP catalog pops up. In the Search field type `AXI NoC` to filter a list of IP.
3. From the filtered list, double-click the **AXI NoC** IP core to instantiate the IP on the block design canvas.

   Customize the IP as follows:

4. In the General tab, set the following options:
   - Number of AXI Slave Interfaces: **2**.
   - Number of AXI Master Interfaces: **0**.
   - Number of AXI Clocks: **2**.

     The number of AXI clocks is set to two because there are two clocks needed for the AXI Slave input, and none needed for AXI Master output.

- Memory Controller: **Single Memory Controller**.
- Number of Memory Controller Port: **4**.
- All others options use the default settings.



☐ Show disabled ports

Component Name axi_noc_0

AXI Interfaces

Number of AXI Slave Interfaces 2
Number of AXI Master Interfaces 0
Number of AXI Clocks 2

Inter-NoC Interfaces

Number of Inter-NOC Slave Interfaces 0
Number of Inter-NOC Master Interfaces 0

Memory Controllers

Memory Controller Single Memory Controller
Number of Memory Controller Ports 4
Interleave Size in Bytes 128
DDR Address Region 0 DDR LOW0 (0x0000 0000 0000 to 0x0000 7FFF FFFF; 2G)
DDR Address Region 1 NONE

The DDR Address Regions define the possible address locations of DDR Memory Controllers.
For more information on DDR Address Regions, please refer the System Address Map section in PG313

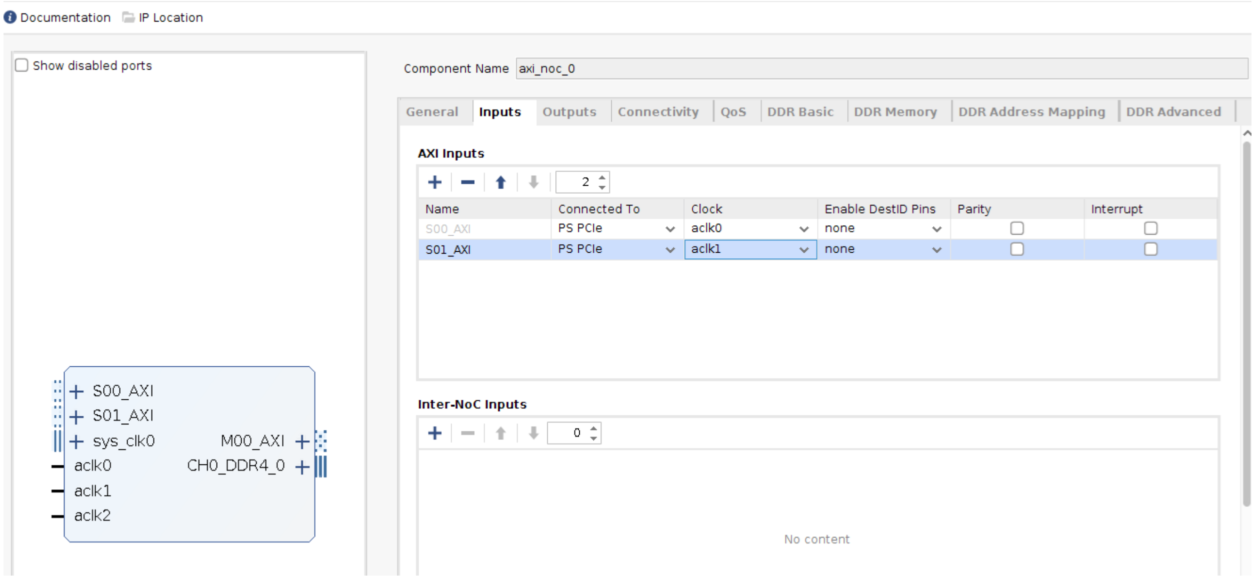
5. In the Inputs tab, set the following options.

   First row (for S00_AXI):

   - Connected To: **PS PCIe**.
   - Clock: **aclk0** (input clock).
   - All other options use default settings.

   Second row (for S01_AXI):

   - Connected To: **PS PCIe**.
   - Clock: **aclk1** (input clock).
   - All other options use default settings.



Component Name axi_noc_0

AXI Inputs

2

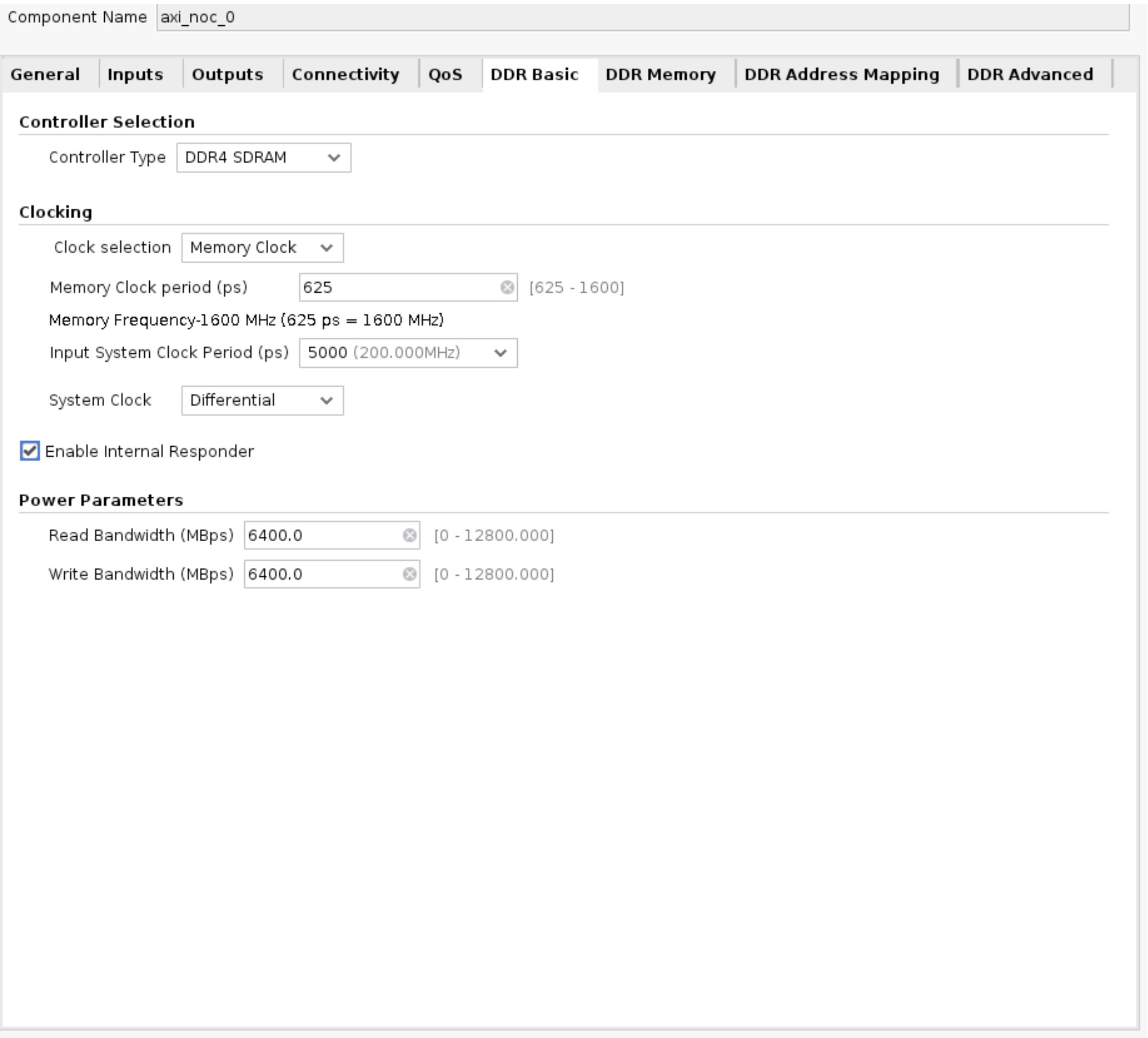
Name	Connected To	Clock	Enable DestID Pins	Parity	Interrupt
S00_AXI	PS PCIe	aclk0	none	☐	☐
S01_AXI	PS PCIe	aclk1	none	☐	☐

Inter-NoC Inputs

0

No content

6. In the Connectivity tab, set the NoC connectivity as follows:
   - For S00_AXI, select the **MC Port 0** checkbox.
   - For S01_AXI, select the **MC Port 0** checkbox.
   - All others options use the default settings.
7. In the DDR Basic tab, set the following options:
   - Input System clock period (ps): **5000 (200.000 MHz)**.
   - Select the **Enable Internal Responder** checkbox.
   - All others options use the default settings.



Component Name axi_noc_0

Controller Selection

Controller Type: DDR4 SDRAM

Clocking

Clock selection: Memory Clock

Memory Clock period (ps): 625 [625 - 1600]

Memory Frequency-1600 MHz (625 ps = 1600 MHz)

Input System Clock Period (ps): 5000 (200.000MHz)

System Clock: Differential

☑ Enable Internal Responder

Power Parameters

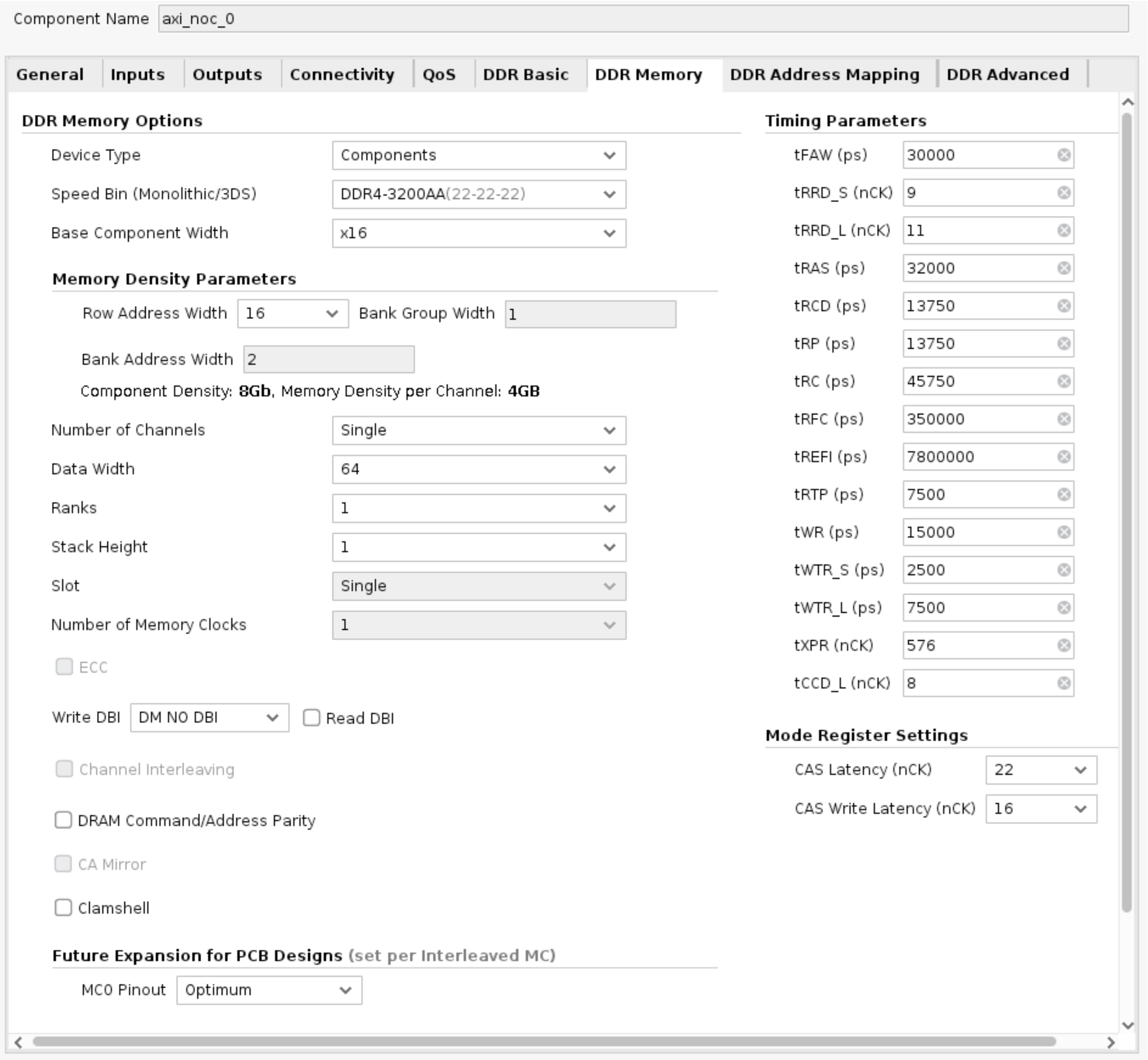
Read Bandwidth (MBps): 6400.0 [0 - 12800.000]

Write Bandwidth (MBps): 6400.0 [0 - 12800.000]

***Note*:** This is a sample configuration. Your DDR configuration and frequencies should be based on your design requirements.

8. In the DDR Memory tab, set the following options:

   - Memory Device Type: **Components**.
   - Memory Speed Grade: **DDR4-3200AA(22-22-22)**.
   - Base Component Width: **x16**.
   - All others options use the default settings.



Component Name: axi_noc_0

DDR Memory Options

Device Type: Components
Speed Bin (Monolithic/3DS): DDR4-3200AA(22-22-22)
Base Component Width: x16

Memory Density Parameters

Row Address Width: 16
Bank Group Width: 1
Bank Address Width: 2
Component Density: **8Gb**, Memory Density per Channel: **4GB**

Number of Channels: Single
Data Width: 64
Ranks: 1
Stack Height: 1
Slot: Single
Number of Memory Clocks: 1

□ ECC
Write DBI: DM NO DBI
□ Read DBI
□ Channel Interleaving
□ DRAM Command/Address Parity
□ CA Mirror
□ Clamshell

Future Expansion for PCB Designs (set per Interleaved MC)

MC0 Pinout: Optimum

Timing Parameters

tFAW (ps)	30000
tRRD_S (nCK)	9
tRRD_L (nCK)	11
tRAS (ps)	32000
tRCD (ps)	13750
tRP (ps)	13750
tRC (ps)	45750
tRFC (ps)	350000
tREFI (ps)	7800000
tRTP (ps)	7500
tWR (ps)	15000
tWTR_S (ps)	2500
tWTR_L (ps)	7500
tXPR (nCK)	576
tCCD_L (nCK)	8

Mode Register Settings

CAS Latency (nCK): 22
CAS Write Latency (nCK): 16

9. Click **OK** to generate a NoC IP with DDR.

Generate the Clock for the NoC IP

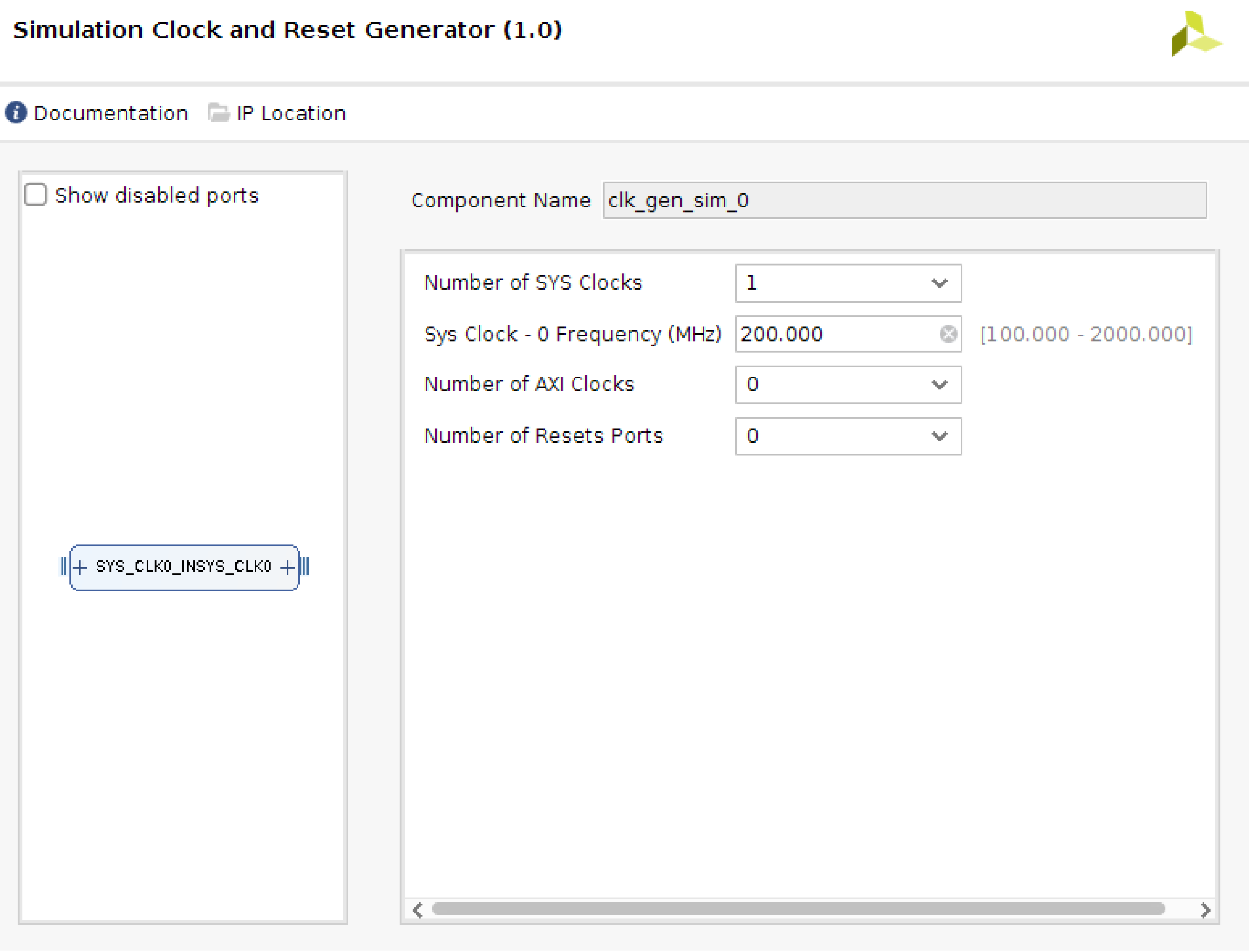
Next, generate a clock source for the NoC module. To do this, you will configure and generate the Simulation Clock and Reset Generator IP core.

1. Click **Add IP**, and search for `Simulation Clock and Reset Generator`.
2. From the filtered list, double-click the **Simulation Clock and Reset Generator** IP core to instantiate the IP on the block design canvas.

   Configure the core as follows:

3. For Number of SYS clocks, select **1**.

4. For Sys Clock 0 Frequency (MHz), enter **200**.
5. For Number of AXI Clocks, select **0**.
6. For Number of Resets Ports, select **0**.
7. Click **OK** to generate IP.



Simulation Clock and Reset Generator (1.0)

Component Name: clk_gen_sim_0

Number of SYS Clocks: 1

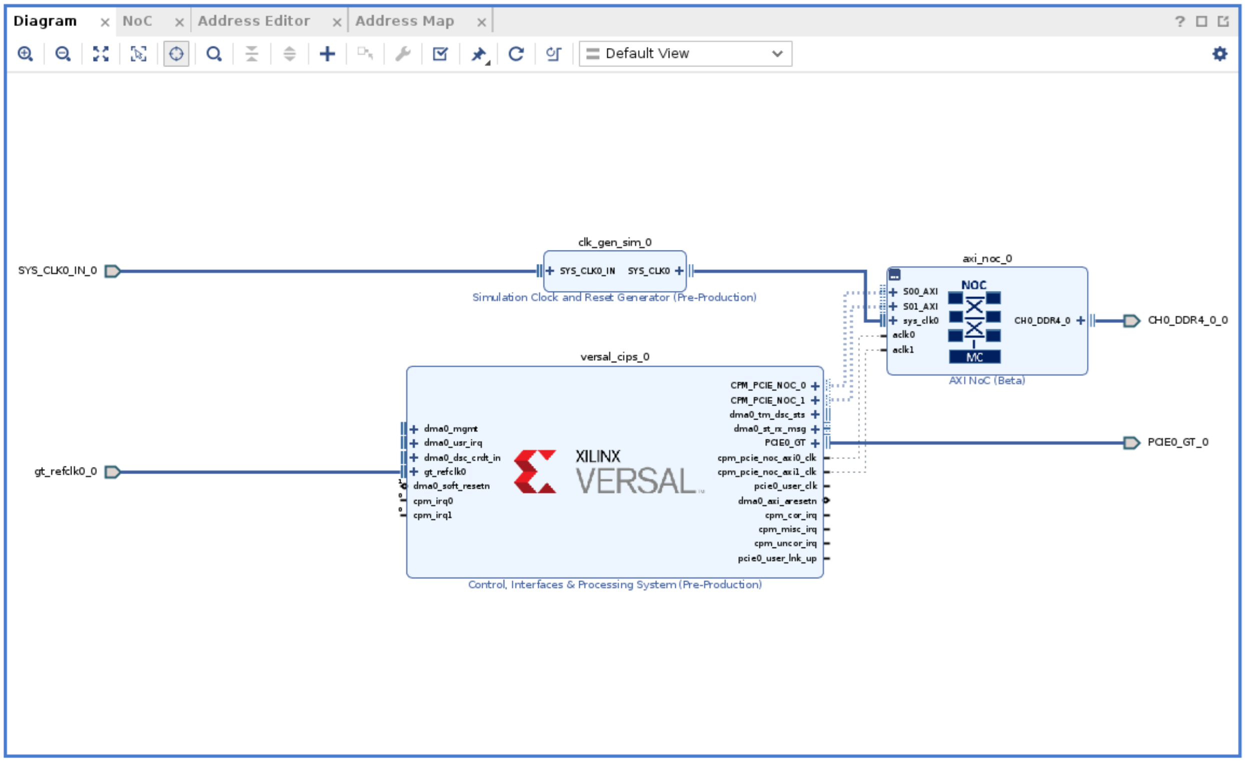
Sys Clock - 0 Frequency (MHz): 200.000 [100.000 - 2000.000]

Number of AXI Clocks: 0

Number of Resets Ports: 0

IP Configuration

1. Make the connections between the IP cores as shown in following figure.
2. Set `GT_REFCLK_D`, `GT_PCIEA0_RX`, `GT_PCIEA0_TX`,`SYS_CLK0_IN`, and `CH0_DDR4_0` as primary ports. To do so:
   a. Select pins `gt_refclk0`, and `PCIE0_GT` of versal_cips_0, `SYS_CLK0_IN` of clk_gen_sim_0, and `CH0_DDR4_0` of axi_noc_0 by pressing **Ctrl+click**.
   b. Click the **Make External (Ctrl + T)** icon in the toolbar at the top of the canvas.
3. Add a Constant IP, and configure the IP to generate a constant value of logic 1.

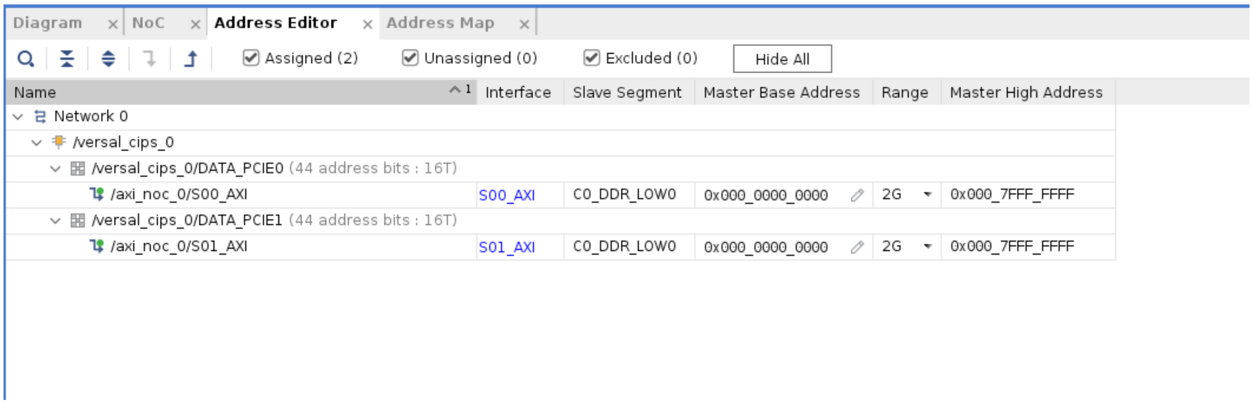


Address Settings

Next, set the necessary address settings for the NoC IP.

1. Open the **Address Editor** tab as shown in the following figure. Expand the tree by clicking the down-arrow on **versal_cips_0**. Expand **DATA_PCIE0**, and expand **DATA_PCIE1**.
2. For S00_AXI, right-click in the Master Base Address cell, and select **Assign** from the context menu.
3. And similarly for S01_AXI, right-click in the Master Base Address cell, and select **Assign** from the context menu.

   Note that the address `0x00000` is assigned to the DDR.



Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/versal_cips_0					
/versal_cips_0/DATA_PCIE0 (44 address bits : 16T)					
/axi_noc_0/S00_AXI	S00_AXI	C0_DDR_LOW0	0x000_0000_0000	2G	0x000_7FFF_FFFF
/versal_cips_0/DATA_PCIE1 (44 address bits : 16T)					
/axi_noc_0/S01_AXI	S01_AXI	C0_DDR_LOW0	0x000_0000_0000	2G	0x000_7FFF_FFFF

Validate the Block Design

1. To validate the design, open the Diagram tab, and click the **Validate Design** icon ☑, or right-click anywhere in the canvas and, from the context menu, select **Validate Design**.

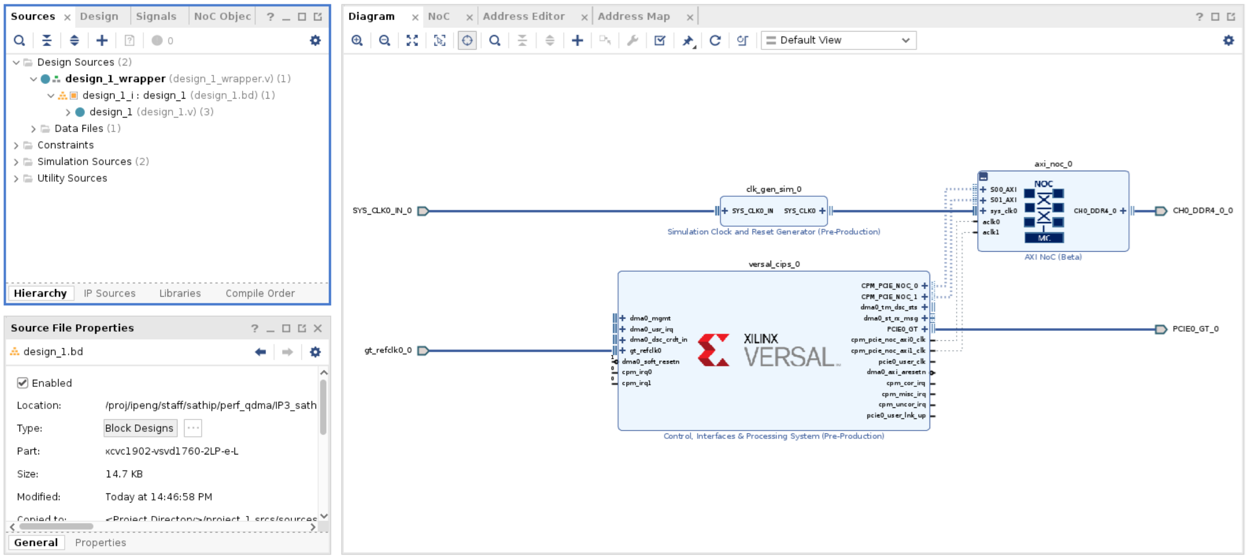
   After validation, confirmation of the successful validation displays in a pop up window.

Create a Design Wrapper

After validation, create a design wrapper. A design wrapper file enables you to add any needed logic. For this lab, additional logic is not needed.

1. In the Vivado IDE Sources window, right-click on **design_1 (design_1.bd)**.
2. From the context menu, select **Create HDL Wrapper** to generate a wrapper file.

   A `design_1_wrapper` file is added to the Sources window as shown in the following figure.



Design Sources (2)
- design_1_wrapper (design_1_wrapper.v) (1)
  - design_1_i : design_1 (design_1.bd) (1)
    - design_1 (design_1.v) (3)
  - Data Files (1)
- Constraints
- Simulation Sources (2)
- Utility Sources

Source File Properties

design_1.bd

☑ Enabled

Location: /proj/ipeng/staff/sathip/perf_qdma/IP3_sath

Type: Block Designs

Part: xcvc1902-vsvd1760-2LP-e-L

Size: 14.7 KB

Modified: Today at 14:46:58 PM

Copied to: <Project Directory>/project_1.srcs/sources

Synthesize and Implement the Design

After the wrapper file is created, you will add the constraints file `top_impl.xdc`, which is provided with this guide, to your design in Vivado. The constraints file constrains DDR pin placement. Then, you can run synthesis and implementation, which generates a PDI (Programmable Device Image) file.

***Note*:** To locate the `top_impl.xdc` constraints file, first download the reference design file file and extract its contents. For details, see Tutorial Design File.

1. In the Flow Navigator window, click **Add Sources**, click **Add or create Constraints**, and add the `top_impl.xdc` file.

2. In the Flow Navigator, click **Synthesis and Implementation** to implement the project design and generate a PDI file.

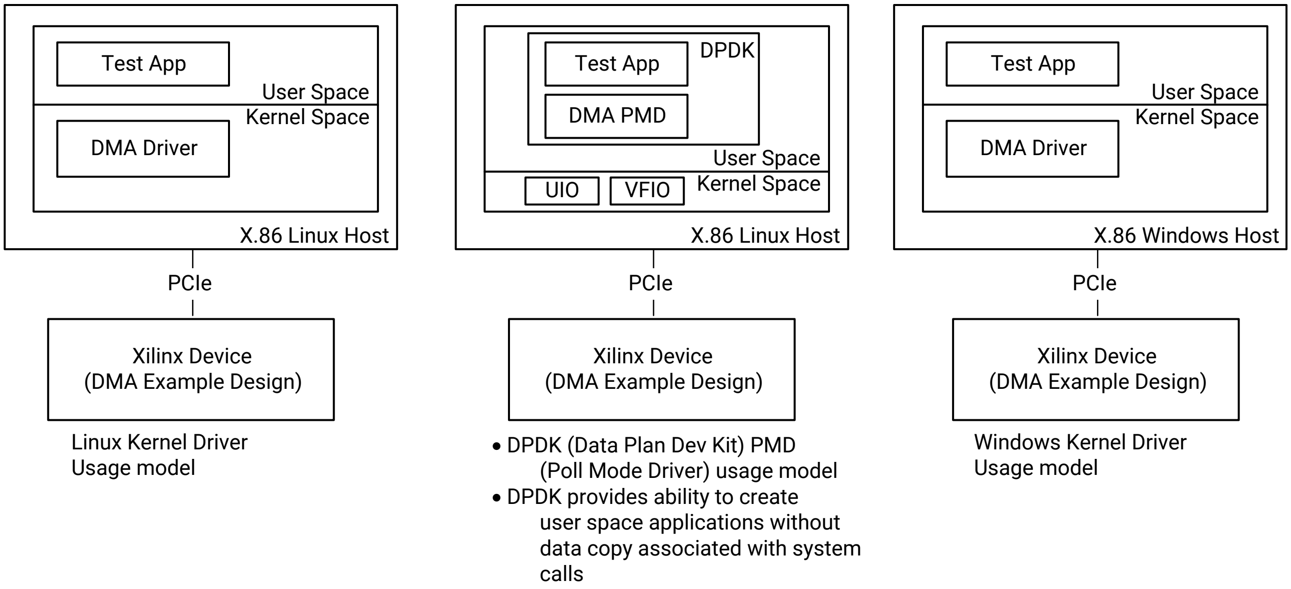
***Note*:** The Tandem critical warning `HD.TANDEM` can be ignored in this release.

Chapter 5

Application Software Development

Device Drivers

Figure 31: **Device Drivers**



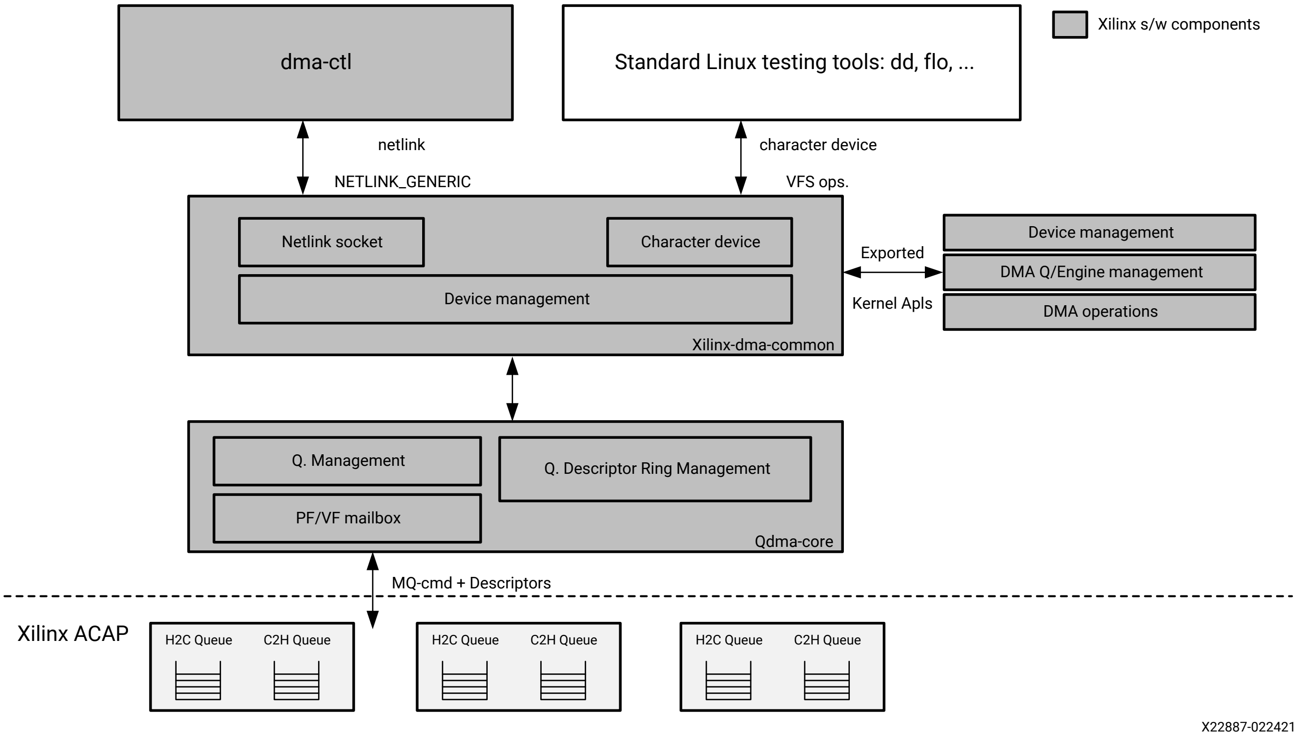
X20600-111220

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 DMA Software Architecture (PF/VF)

Figure 32: **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.

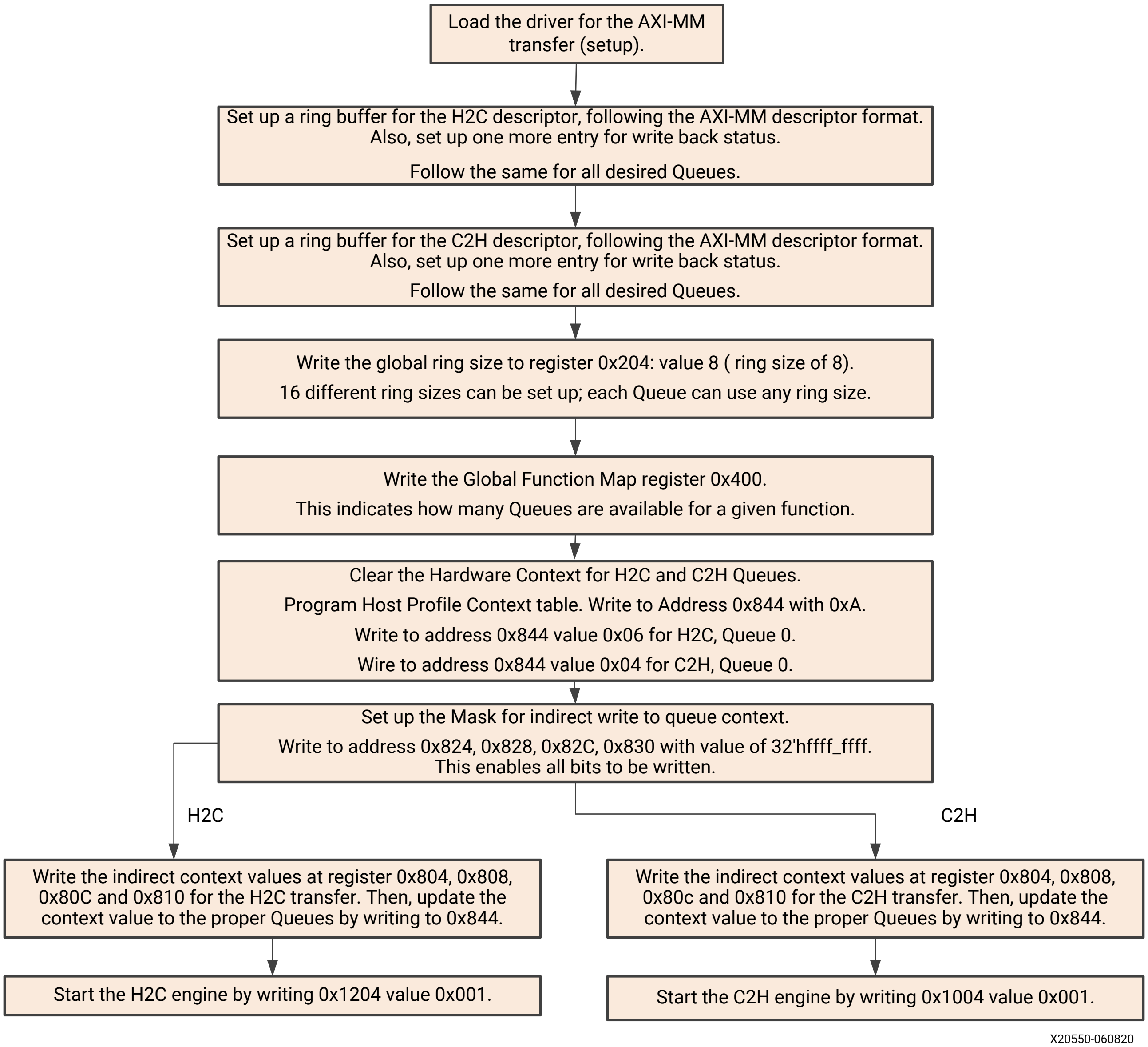
Using the Drivers

Linux, DPDK and Windows drivers and the corresponding documentation are available at Xilinx DMA IP Drivers.

Reference Software Driver Flow

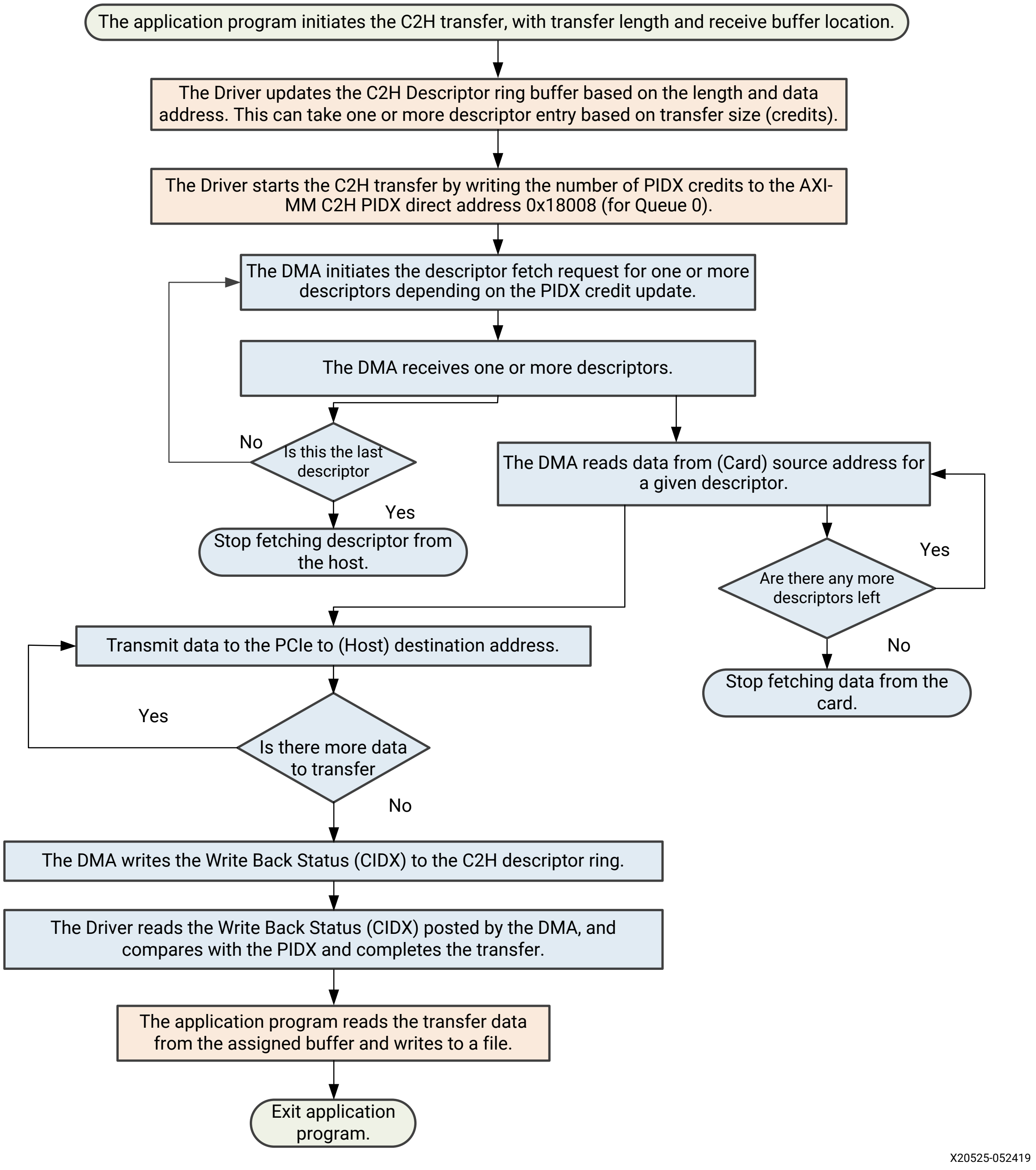
AXI4 Memory Map Flow Chart

Figure 33: **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 the Global Function Map register 0x400.
This indicates how many Queues are available for a given 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.
X20550-060820

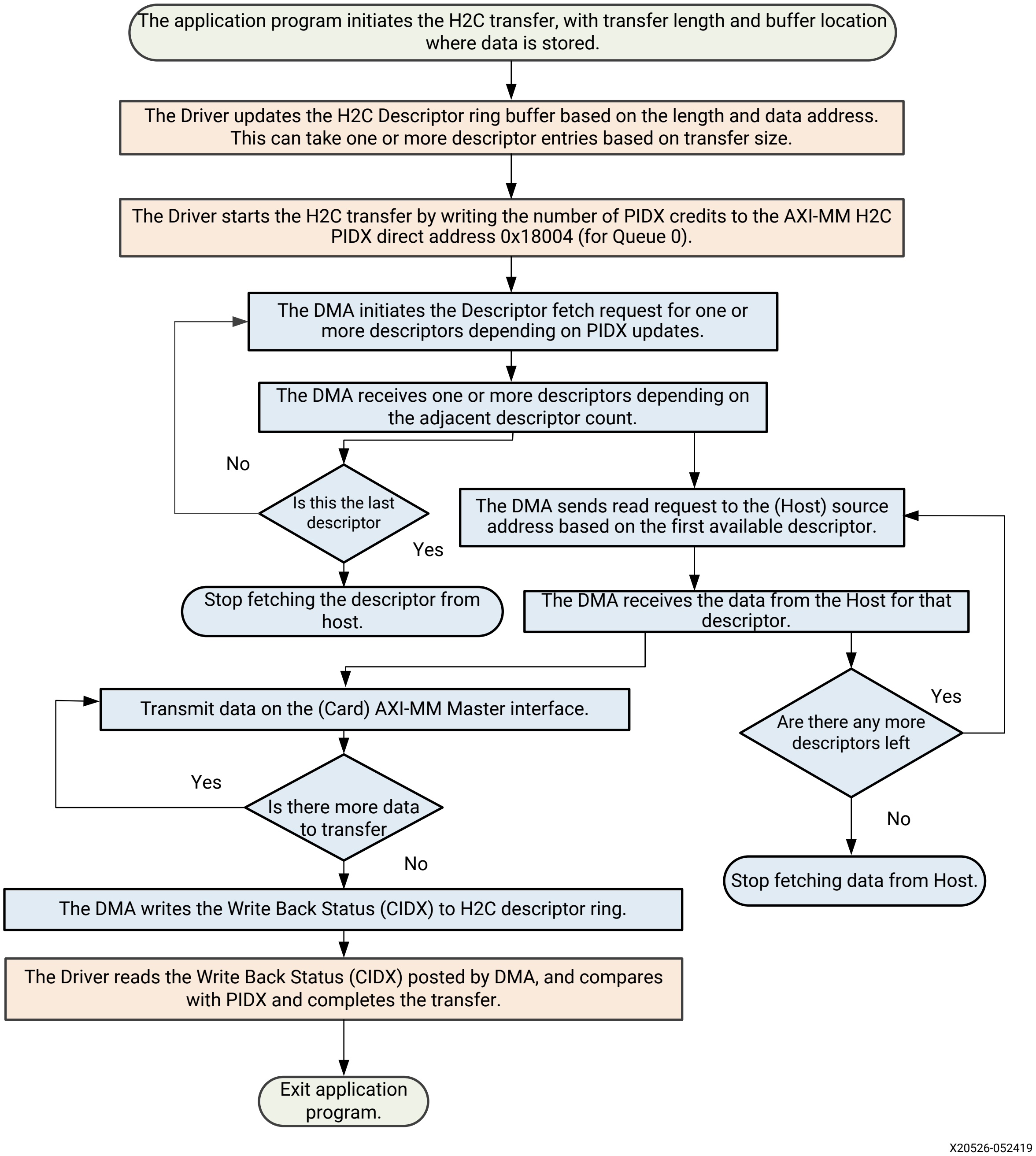
AXI4 Memory Mapped C2H Flow

Figure 34: **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.
No
Is this the last descriptor
Yes
Stop fetching descriptor from the host.
The DMA reads data from (Card) source address for a given descriptor.
Yes
Are there any more descriptors left
No
Stop fetching data from the card.
Transmit data to the PCIe to (Host) destination address.
Yes
Is there more data to transfer
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

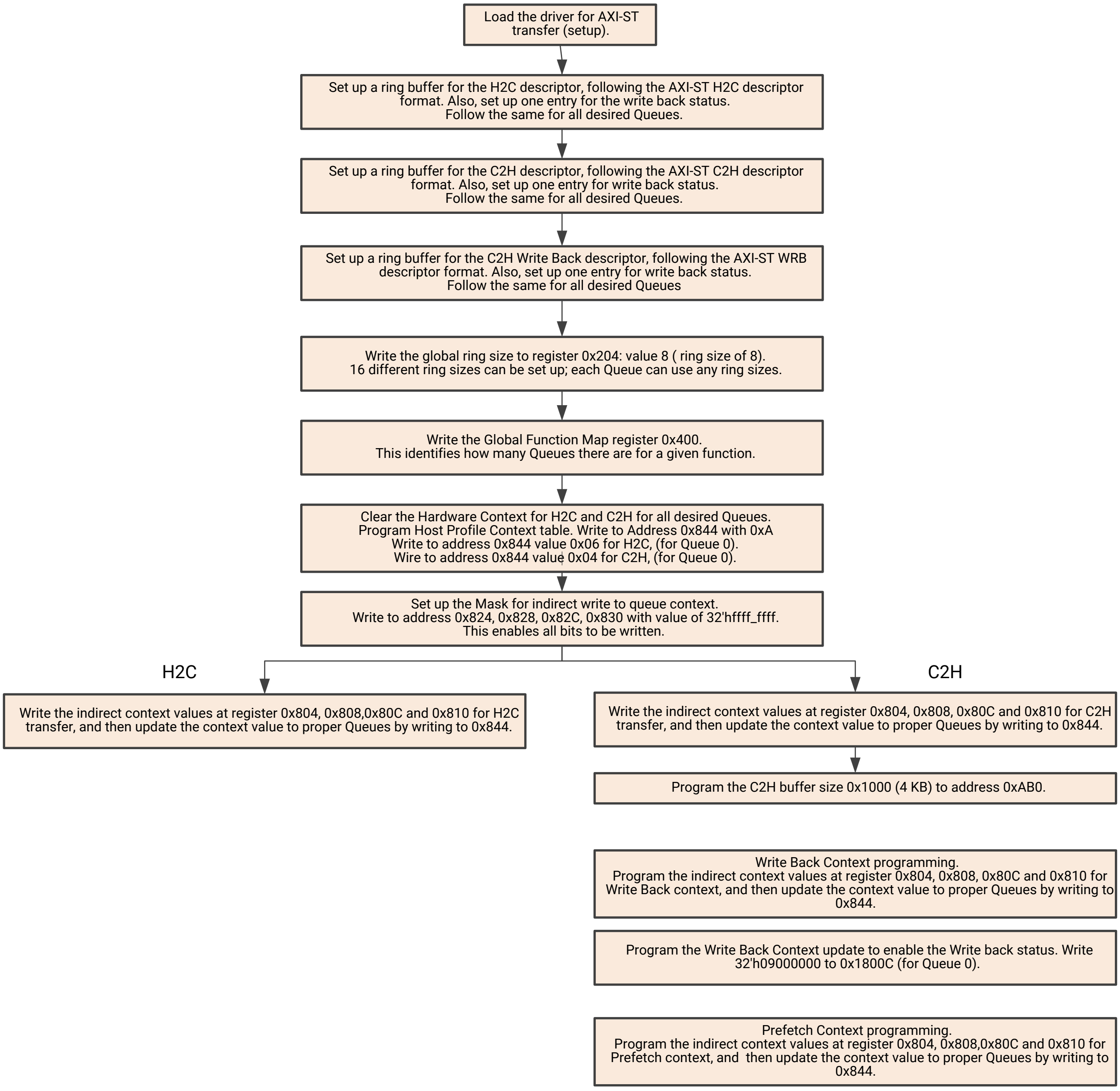
AXI4 Memory Mapped H2C Flow

Figure 35: **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.
Yes
Are there any more descriptors left
No
Stop fetching data from Host.
Transmit data on the (Card) AXI-MM Master interface.
Yes
Is there more data to transfer
No
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

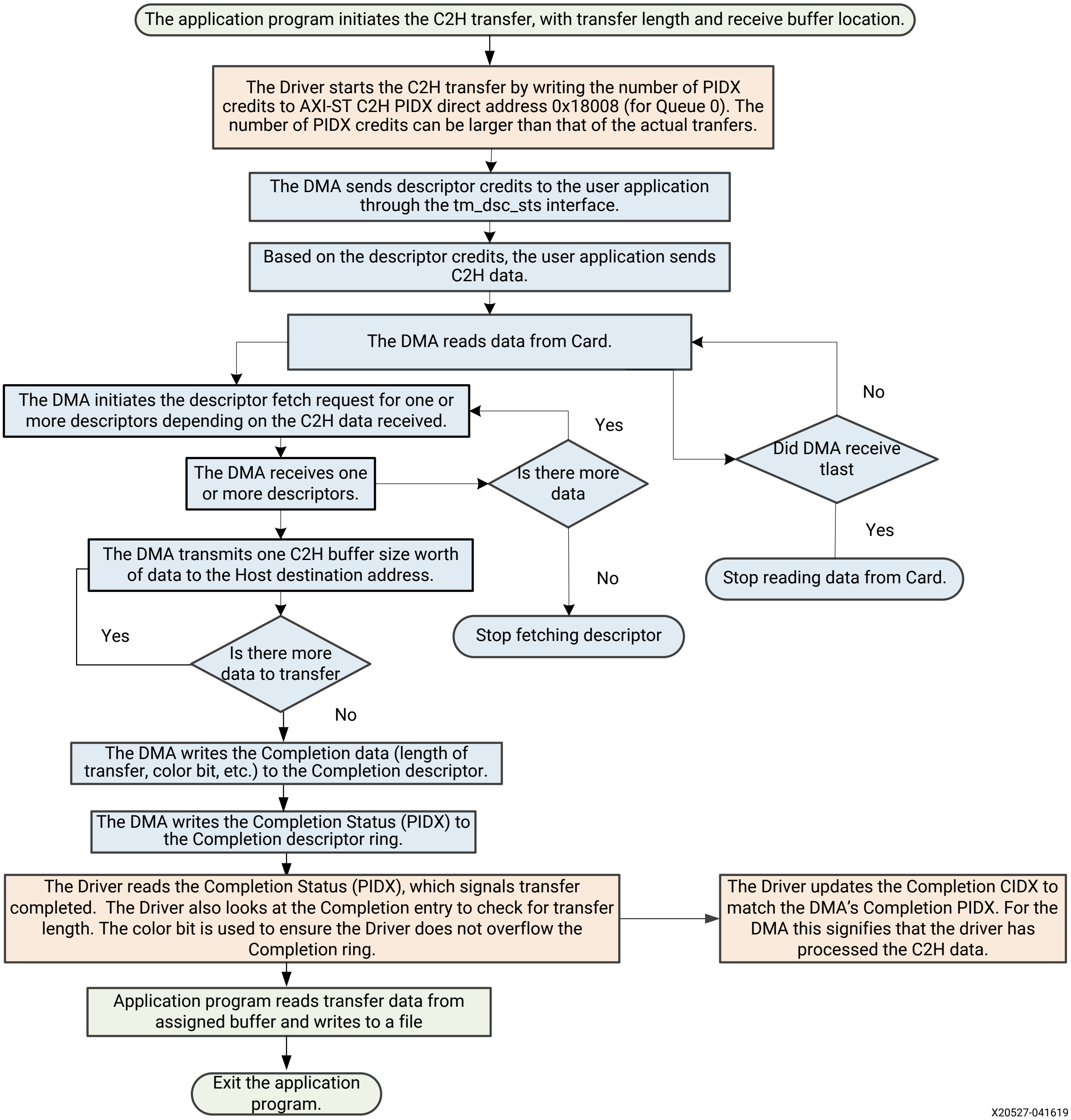
Figure 36: **AXI4-Stream Flow Chart**


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 the Global Function Map register 0x400. This identifies how many Queues there are for a given 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

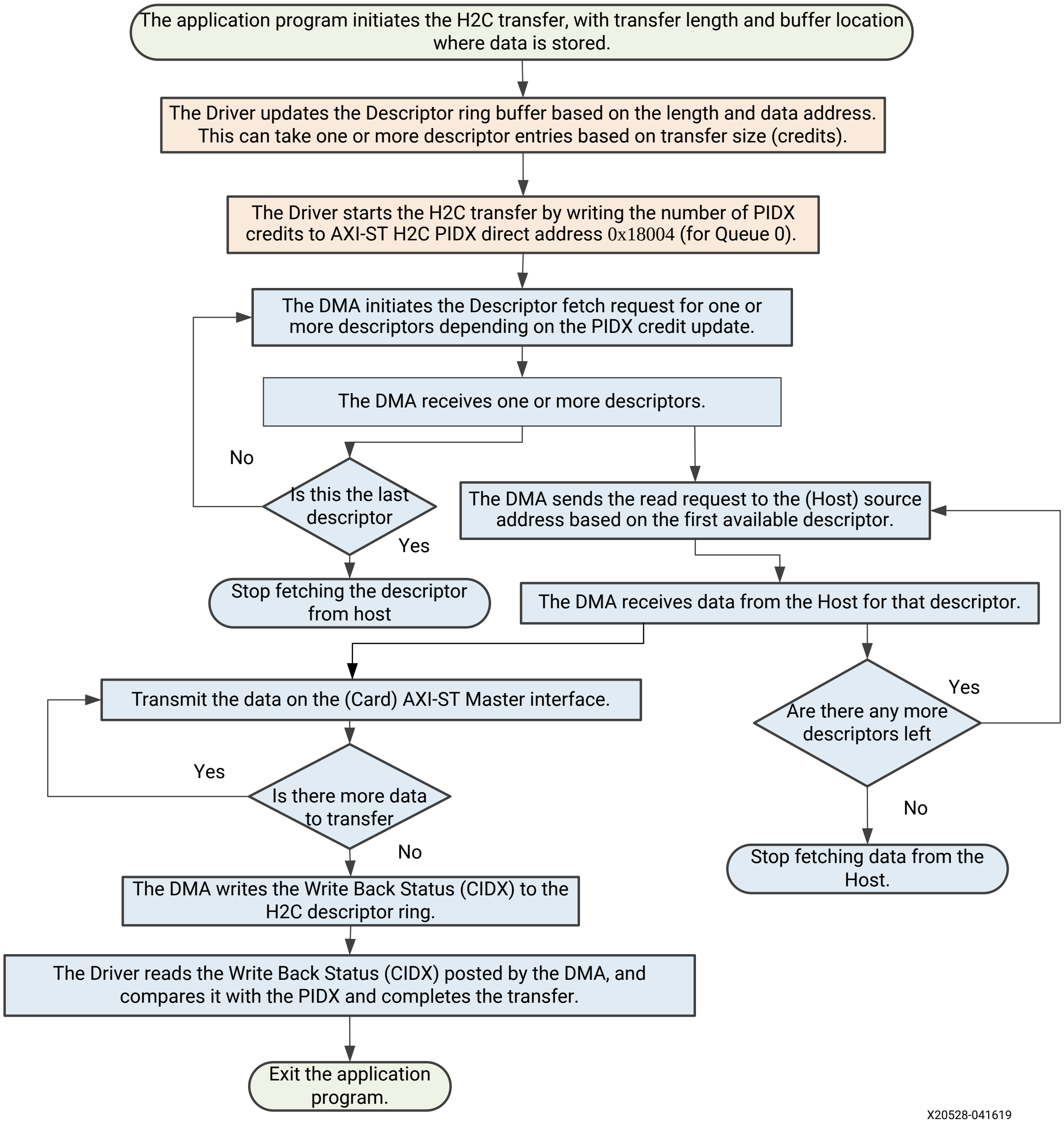
AXI4-Stream C2H Flow

Figure 37: **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 38: **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

Chapter 6

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

Solution Centers

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

The Solution Center specific to the QDMA is 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 functional mode can be located by using the Search Support box on the main Xilinx support web page. To maximize your search results, use keywords such as:

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

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

Master Answer Record for the Core

AR 75396.

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.

Soft Reset

Reset the QDMA logic through the `dma0_soft_reset_n` port. This port needs to be held in reset for a minimum of 100 clock cycles (`pcie0_user_clk` cycles).

This signal resets only the DMA portion of logic. It does not reset the PCIe hard block.

Soft Reset Use Cases

The uses cases that prompt the use of `dma0_soft_reset` include:

- DMA does not respond, and the user application is not getting proper values.
- DMA transfer has errors, but the PCIe links are good.
- DMA records some asynchronous errors.

After `dma0_soft_reset`, you must reinitialize the queues and program all queue context.

Registers

A complete list of registers and attributes for the QDMA Subsystem is available in the *Versal ACAP Register Reference* (AM012). Reviewing the registers and attributes might be helpful for advanced debugging.

***Note*:** The attributes are set during IP customization in the Vivado IP catalog. After core customization, attributes are read-only.

Chapter 7

Upgrading

This appendix is not applicable for the first release of the functional mode.

Section III

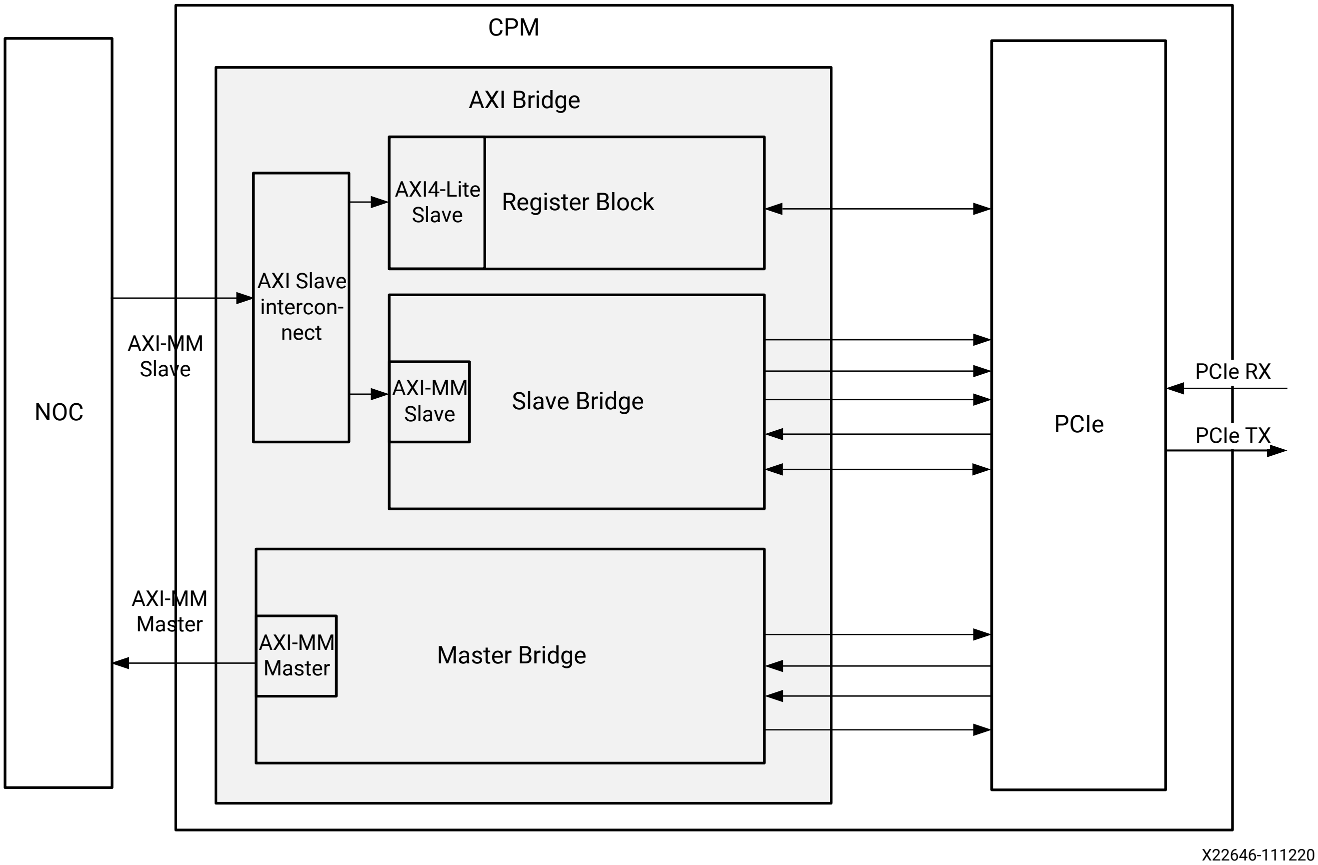
AXI Bridge Subsystem

Overview

The AXI Bridge Subsystem is designed for the Vivado® IP integrator in the Vivado® Design Suite. The AXI Bridge functional mode provides an interface between an AXI4 customer user interface and PCI Express® using the Versal™ Integrated Block for PCI Express. The AXI Bridge functional mode provides the translation level between the AXI4 embedded system to the PCI Express system. The AXI Bridge functional mode 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 Bridge is shown in the following figure.

Figure 39: **High-Level AXI Bridge Architecture**



Limitations

For this functional mode, the bridge master and bridge slave cannot achieve more than 128 Gb/s.

Chapter 8

Product Specification

The Register block contains registers used in the AXI Bridge functional mode 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 PCIe `MemWr` request TLPs. PCIe `MemWr` request TLPs support is as follows:

- 4 for 64-bit AXI data width
- 8 for 128-bit AXI data width
- 16 for 256-bit AXI data width
- 32 for 512-bit AXI data width

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 in 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, and interrupt registers.

AXI Bridge Operations

AXI Transactions for PCIe

The following tables are the translation tables for AXI4-Stream and memory-mapped transactions.

Table 72: **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 73: **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.

`s_axi_wstrb` 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 functional mode 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 functional mode 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 functional mode. 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 functional mode 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

BAR Addressing

`C_AXIBAR_n` and `C_AXIBAR_HIGHADDR_n` are used to calculate the size of the AXI BAR *n* and during address translation to PCIe address.

- `C_AXIBAR_n` provides the low address where AXI BAR *n* starts and will be regarded as address offset 0x0 when the address is translated.
- `C_AXIBAR_HIGHADDR_n` is the high address of the last valid byte address of AXI BAR *n*. (For more details on how the address gets translated, see Address Translation.)

The difference between the two parameters is your AXI BAR *n* size. These parameters must be set accordingly such that the AXI BAR *n* size is a power of two and must have at least 4K.

When a packet is sent to the core (outgoing PCIe packets), the packet must have an address that is in the range of `C_AXIBAR_n` and `C_AXIBAR_HIGHADDR_n`. Any packet that is received by the core that has an address outside of this range will be responded to with a SLVERR. When the IP integrator is used, these parameters are derived from the Address Editor tab within the IP integrator. The Address Editor sets the AXI Interconnect as well as the core so the address range matches, and the packet is routed to the core only when the packet has an address within the valid range.

Address Translation

The address space for PCIe® is different than the AXI address space. To access one address space from another address space requires an address translation process. On the AXI side, the bridge supports mapping to PCIe on up to six 32-bit or 64-bit AXI base address registers (BARs). The generics used to configure the BARs follow.

C_AXIBAR_NUM, C_AXIBAR_n, C_AXIBAR_HIGHADDR_n, and C_AXIBAR2PCIEBAR_n

where *n* represents an AXIBAR number from 0 to 5. The bridge for supports mapping on up to six 32-bit BARs or three 64-bit BARs for PCIe. The generics used to configure the BARs are:

PCIEBAR_NUM, C_PCIEBAR2AXIBAR_n and PF0_BARn_APERTURE_SIZE

where *n* represents a particular BAR number for PCIe from 0 to 5.

AXIBAR2PCIEBAR_n translation vectors can be changed by using software by writing to AXI Base Address Translation Configuration Registers.

Four examples follow:

- Example 1 (32-bit PCIe Address Mapping) demonstrates how to set up three AXI BARs and translate the AXI address to a 32-bit address for PCIe.
- Example 2 (64-bit PCIe Address Mapping) demonstrates how to set up three AXI BARs and translate the AXI address to a 64-bit address for PCIe.
- Example 3 demonstrates how to set up two 64-bit PCIe BARs and translate the address for PCIe to an AXI address.
- Example 4 demonstrates how to set up a combination of two 32-bit AXI BARs and two 64 bit AXI BARs, and translate the AXI address to an address for PCIe.

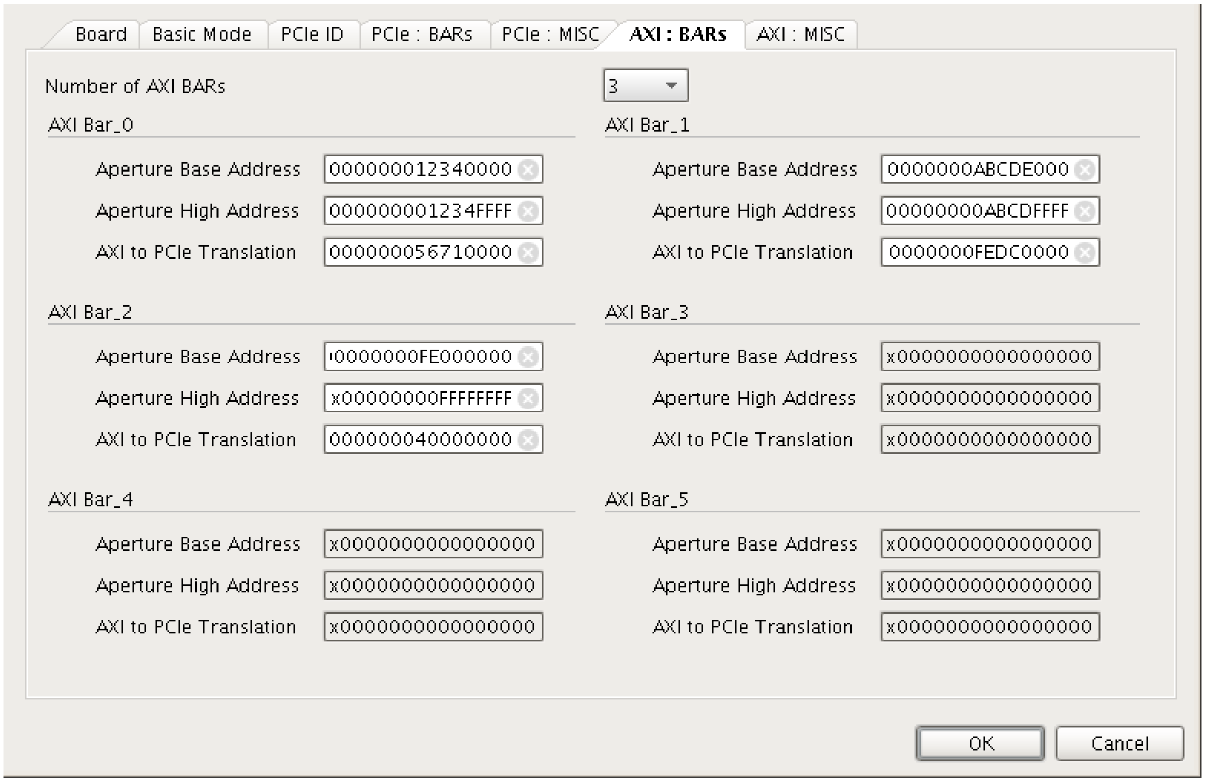
Example 1 (32-bit PCIe Address Mapping)

This example shows the generic settings to set up three independent AXI BARs and address translation of AXI addresses to a remote 32-bit address space for PCIe. This setting of AXI BARs does not depend on the BARs for PCIe in the AXI Bridge functional mode.

In this example, where C_AXIBAR_NUM=3, the following assignments for each range are made:

```
AXI_ADDR_WIDTH=48

C_AXIBAR_0=0x00000000_12340000
C_AXI_HIGHADDR_0=0x00000000_1234FFFF (64 Kbytes)
C_AXIBAR2PCIEBAR_0=0x00000000_56710000 (Bits 63-32 are zero in order to
produce a
32-bit PCIe TLP. Bits 15-0 must be zero based on the AXI BAR aperture size.
Non-zero
values in the lower 16 bits are invalid translation values.)

C_AXIBAR_1=0x00000000_ABCDE000
C_AXI_HIGHADDR_1=0x00000000_ABCDFFFF (8 Kbytes)
C_AXIBAR2PCIEBAR_1=0x00000000_FEDC0000 (Bits 63-32 are zero in order to
produce a
32-bit PCIe TLP. Bits 12-0 must be zero based on the AXI BAR aperture size.
Non-zero
values in the lower 13 bits are invalid translation values.)

C_AXIBAR_2=0x00000000_FE000000
C_AXI_HIGHADDR_2=0x00000000_FFFFFFFF (32 Mbytes)
C_AXIBAR2PCIEBAR_2=0x00000000_40000000 (Bits 63-32 are zero in order to
produce a
32-bit PCIe TLP. Bits 24-0 must be zero based on the AXI BAR aperture size.
Non-zero
values in the lower 25 bits are invalid translation values.)
```

Figure 40: **Example 1 Settings**



Number of AXI BARs: 3

AXI Bar_0
Aperture Base Address: 000000012340000
Aperture High Address: 000000001234FFFF
AXI to PCIe Translation: 000000056710000

AXI Bar_1
Aperture Base Address: 0000000ABCDE000
Aperture High Address: 00000000ABCDFFFF
AXI to PCIe Translation: 0000000FEDC0000

AXI Bar_2
Aperture Base Address: 0000000FE000000
Aperture High Address: x00000000FFFFFFFF
AXI to PCIe Translation: 000000040000000

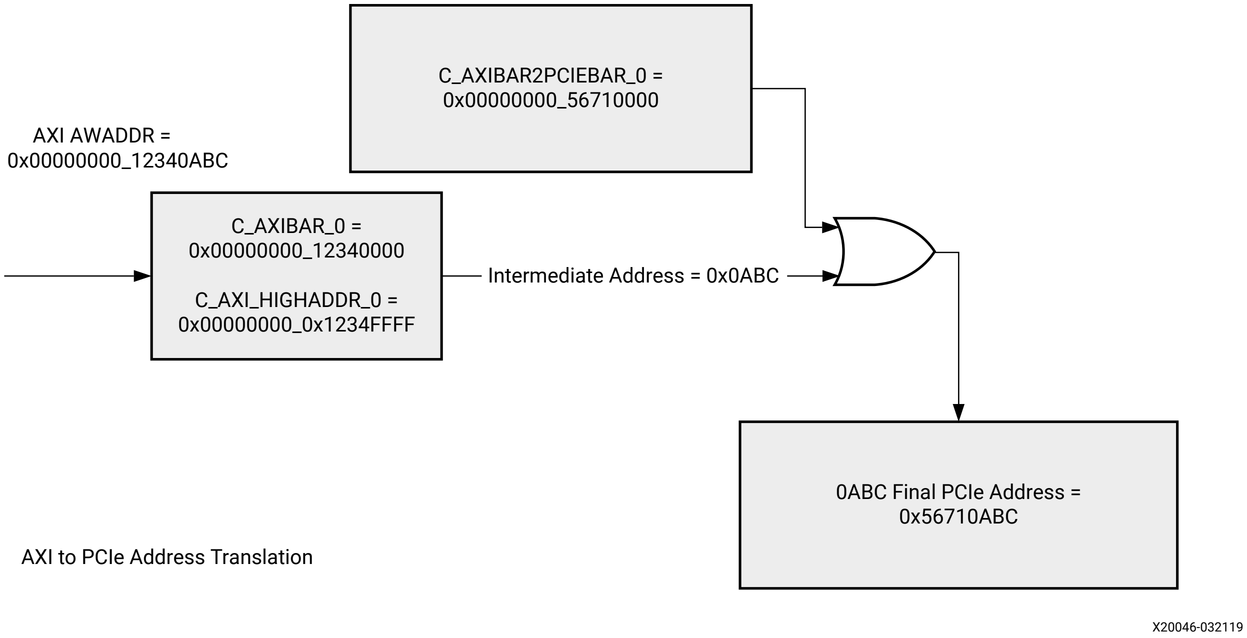
AXI Bar_3
Aperture Base Address: x0000000000000000
Aperture High Address: x0000000000000000
AXI to PCIe Translation: x0000000000000000

AXI Bar_4
Aperture Base Address: x0000000000000000
Aperture High Address: x0000000000000000
AXI to PCIe Translation: x0000000000000000

AXI Bar_5
Aperture Base Address: x0000000000000000
Aperture High Address: x0000000000000000
AXI to PCIe Translation: x0000000000000000

- Accessing the Bridge `AXIBAR_0` with address `0x0000_12340ABC` on the AXI bus yields `0x56710ABC` on the bus for PCIe.

Figure 41: **AXI to PCIe Address Translation**



- Accessing the Bridge AXIBAR_1 with address `0x0000_ABCDF123` on the AXI bus yields `0xFEDC1123` on the bus for PCIe.
- Accessing the Bridge AXIBAR_2 with address `0x0000_FFEDCBA` on the AXI bus yields `0x41FEDCBA` on the bus for PCIe.

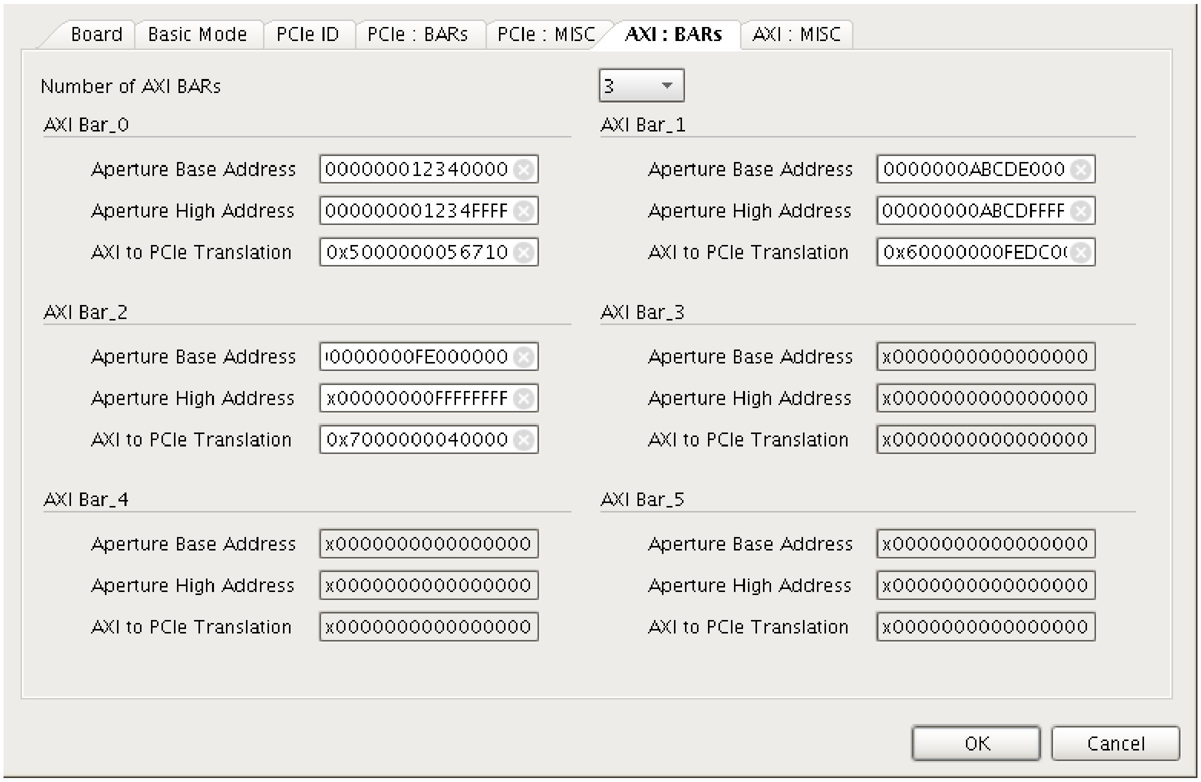
Example 2 (64-bit PCIe Address Mapping)

This example shows the generic settings to set up to three independent AXI BARs and address translation of AXI addresses to a remote 64-bit address space for PCIe. This setting of AXI BARs does not depend on the BARs for PCIe within the Bridge.

In this example, where C_AXIBAR_NUM=3, the following assignments for each range are made:

```
AXI_ADDR_WIDTH=48

C_AXIBAR_0=0x00000000_12340000
C_AXI_HIGHADDR_0=0x00000000_1234FFFF (64 Kbytes)
C_AXIBAR2PCIEBAR_0=0x5000000056710000 (Bits 63-32 are non-zero in order to 
produce a 
64-bit PCIe TLP. Bits 15-0 must be zero based on the AXI BAR aperture size. 
Non-zero 
values in the lower 16 bits are invalid translation values.)

C_AXIBAR_1=0x00000000_ABCDE000
C_AXI_HIGHADDR_1=0x00000000_ABCDFFFF (8 Kbytes)
C_AXIBAR2PCIEBAR_1=0x60000000_FEDC0000 (Bits 63-32 are non-zero in order to 
produce
a 64-bit PCIe TLP. Bits 12-0 must be zero based on the AXI BAR aperture
size. Non-zero
values in the lower 13 bits are invalid translation values.)

C_AXIBAR_2=0x00000000_FE000000
C_AXI_HIGHADDR_2=0x00000000_FFFFFFFF (32 Mbytes)
C_AXIBAR2PCIEBAR_2=0x7000000040000 (Bits 63-32 are non-zero in order to
produce a
64-bit PCIe TLP. Bits 24-0 must be zero based on the AXI BAR aperture size.
Non-zero
values in the lower 25 bits are invalid translation values.)
```

Figure 42: **Example 2 Settings**



Number of AXI BARs: 3

	AXI Bar_0	AXI Bar_1
Aperture Base Address	000000012340000	0000000ABCDE000
Aperture High Address	00000001234FFFF	00000000ABCDFFFF
AXI to PCIe Translation	0x5000000056710	0x60000000FEDC0

	AXI Bar_2	AXI Bar_3
Aperture Base Address	0000000FE000000	x0000000000000000
Aperture High Address	x00000000FFFFFFFF	x0000000000000000
AXI to PCIe Translation	0x7000000040000	x0000000000000000

	AXI Bar_4	AXI Bar_5
Aperture Base Address	x0000000000000000	x0000000000000000
Aperture High Address	x0000000000000000	x0000000000000000
AXI to PCIe Translation	x0000000000000000	x0000000000000000

- Accessing the Bridge AXIBAR_0 with address `0x0000_12340ABC0x500000056710ABC` on the bus for PCIe.
- Accessing the Bridge AXIBAR_1 with address `0x0000_ABCDF123` on the AXI bus yields on the AXI bus yields `0x60000000FEDC1123` on the bus for PCIe.
- Accessing the Bridge AXIBAR_2 with address `0x0000_FFFEDCBA` on the AXI bus yields `0x7000000041FEDCBA` on the bus for PCIe.

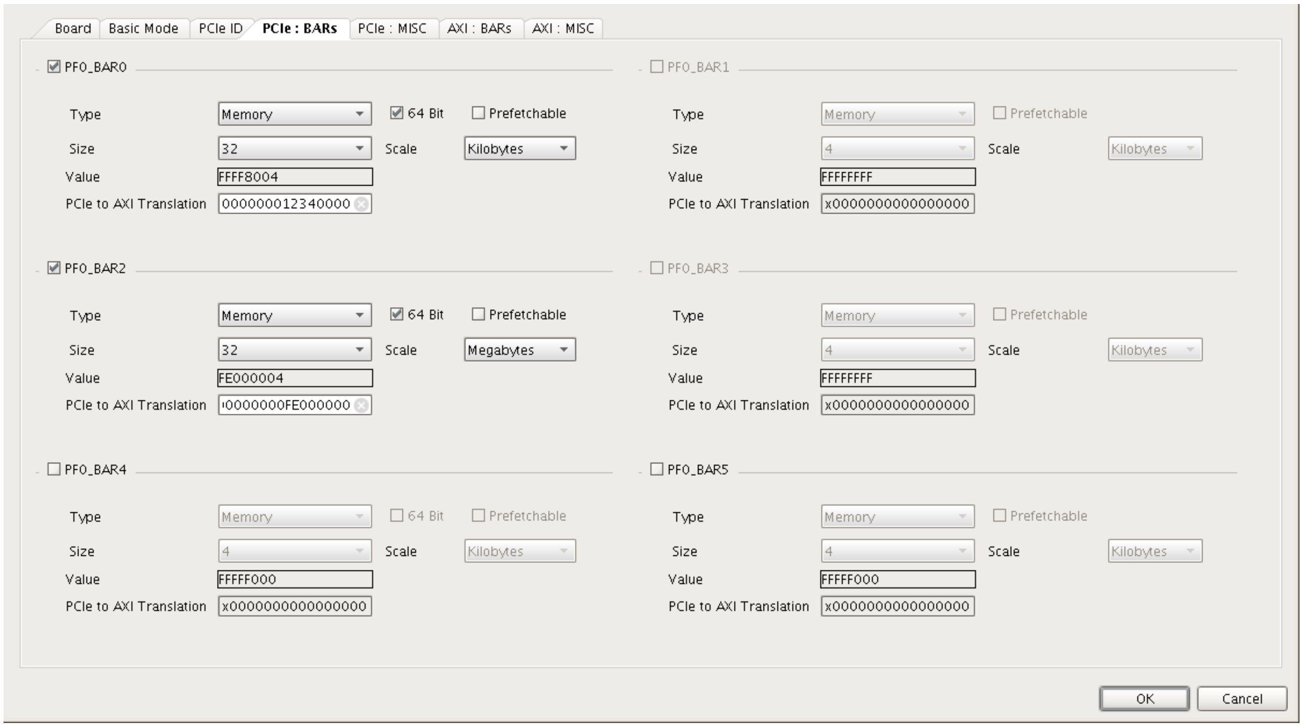
Example 3

This example shows the generic settings to set up two independent BARs for PCIe® and address translation of addresses for PCIe to a remote AXI address space. This setting of BARs for PCIe does not depend on the AXI BARs within the bridge.

In this example, where C_PCIEBAR_NUM=2, the following range assignments are made:

```
AXI_ADDR_WIDTH=48

BAR 0 is set to 0x20000000_ABCD8000 by the Root Port. (Since this is a 64-
bit BAR
PCIe, BAR1 is disabled.)
PF0_BAR0_APERTURE_SIZE=0x08 (32 Kbytes)
C_PCIEBAR2AXIBAR_0=0x00000000_12340000 (Because the AXI address is 48-bits
wide,
bits 63-48 should be zero. Base on the PCIe Bar Size bits 14-0 should be
zero.
Non-zero values in these ranges are invalid.)

BAR 2 is set to 0xA000000012000000 by Root Port. (Since this is a 64-bit
BAR PCIe BAR3
is disabled.)
PF0_BAR0_APERTURE_SIZE=0x12 (32 Mbytes)
C_PCIEBAR2AXIBAR_2=0x00000000_FE000000 (Because the AXI address is 48-bits
wide,
bits 63-48 should be zero. Base on the PCIe Bar Size bits 24-0 should be
zero.
Non-zero values in these ranges are invalid.)
```

Figure 43: **Example 3 Settings**



☑ PF0_BAR0
Type: Memory ☑ 64 Bit □ Prefetchable
Size: 32 Scale: Kilobytes
Value: FFFF8004
PCIe to AXI Translation: 000000012340000

□ PF0_BAR1
Type: Memory □ Prefetchable
Size: 4 Scale: Kilobytes
Value: FFFFFFFF
PCIe to AXI Translation: x0000000000000000

☑ PF0_BAR2
Type: Memory ☑ 64 Bit □ Prefetchable
Size: 32 Scale: Megabytes
Value: FE000004
PCIe to AXI Translation: ı0000000FE000000

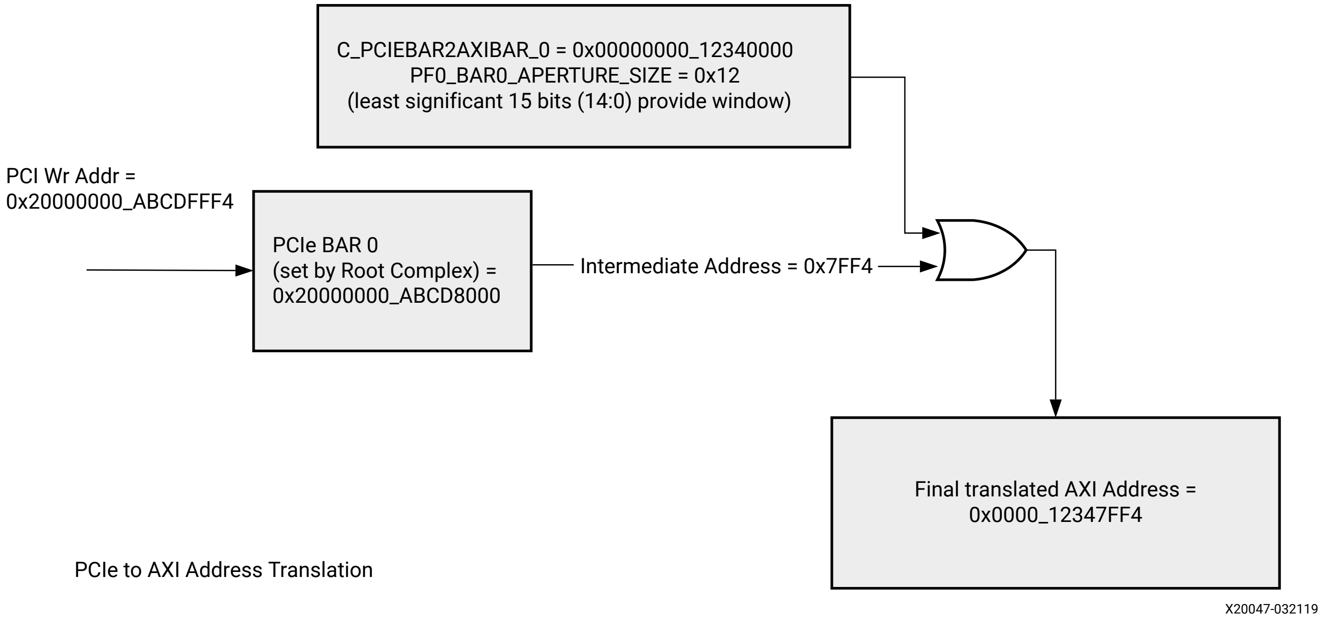
□ PF0_BAR3
Type: Memory □ Prefetchable
Size: 4 Scale: Kilobytes
Value: FFFFFFFF
PCIe to AXI Translation: x0000000000000000

□ PF0_BAR4
Type: Memory □ 64 Bit □ Prefetchable
Size: 4 Scale: Kilobytes
Value: FFFFF000
PCIe to AXI Translation: x0000000000000000

□ PF0_BAR5
Type: Memory □ Prefetchable
Size: 4 Scale: Kilobytes
Value: FFFFF000
PCIe to AXI Translation: x0000000000000000

- Accessing the Bridge PCIEBAR_0 with address `0x20000000_ABCDFFF4` on the bus for PCIe yields `0x0000_12347FF4` on the AXI bus.

Figure 44: **PCIe to AXI Translation**


C_PCIEBAR2AXIBAR_0 = 0x00000000_12340000
PF0_BAR0_APERTURE_SIZE = 0x12
(least significant 15 bits (14:0) provide window)
PCI Wr Addr =
0x20000000_ABCDFFF4
PCIe BAR 0
(set by Root Complex) =
0x20000000_ABCD8000
Intermediate Address = 0x7FF4
Final translated AXI Address =
0x0000_12347FF4
PCIe to AXI Address Translation
X20047-032119

- Accessing Bridge PCIEBAR_2 with address `0xA00000001235FEDC` on the bus for PCIe yields `0x0000_FE35FEDC` on the AXI bus.

Example 4

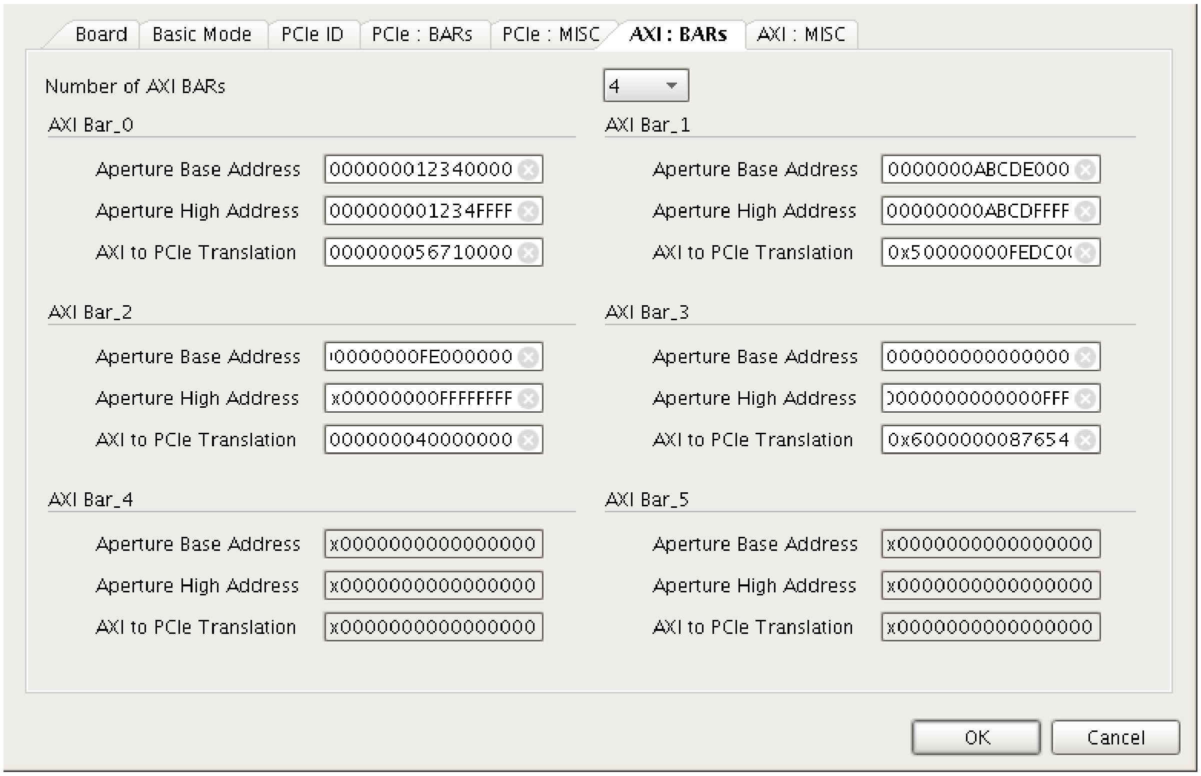
This example shows the generic settings of four AXI BARs and address translation of AXI addresses to a remote 32-bit and 64-bit addresses for PCIe®. This setting of AXI BARs do not depend on the BARs for PCIe within the Bridge.

In this example, where C_AXIBAR_NUM=4, the following assignments for each range are made:

```
AXI_ADDR_WIDTH=48

C_AXIBAR_0=0x00000000_12340000
C_AXI_HIGHADDR_0=0x00000000_1234FFFF (64 KB)
C_AXIBAR2PCIEBAR_0=0x00000000_56710000 (Bits 63-32 are zero to produce a 32-
bit PCIe
TLP. Bits 15-0 must be zero based on the AXI BAR aperture size. Non-zero
values in
the lower 16 bits are invalid translation values.)

C_AXIBAR_1=0x00000000_ABCDE000
C_AXI_HIGHADDR_1=0x00000000_ABCDFFFF (8 KB)
C_AXIBAR2PCIEBAR_1=0x50000000_FEDC0000 (Bits 63-32 are non-zero to produce
a 64-bit
PCIe TLP. Bits 12-0 must be zero based on the AXI BAR aperture size. Non-
zero values
in the lower 13 bits are invalid translation values.)
```

```
C_AXIBAR_2=0x00000000_FE000000
C_AXI_HIGHADDR_2=0x00000000_FFFFFFFF (32 MB)
C_AXIBAR2PCIEBAR_2=0x00000000_40000000 (Bits 63-32 are zero to produce a 32-
bit PCIe
TLP. Bits 24-0 must be zero based on the AXI BAR aperture size. Non-zero
values in
the lower 25 bits are invalid translation values.)

C_AXIBAR_3=0x00000000_00000000
C_AXI_HIGHADDR_3=0x00000000_00000FFF (4 KB)
C_AXIBAR2PCIEBAR_3=0x60000000_87654000 (Bits 63-32 are non-zero to produce
a 64-bit
PCIe TLP. Bits 11-0 must be zero based on the AXI BAR aperture size. Non-
zero values
in the lower 12 bits are invalid translation values.)
```

Figure 45: **Example 4 Settings**



Number of AXI BARs: 4

AXI Bar_0
- Aperture Base Address: 000000012340000
- Aperture High Address: 00000001234FFFF
- AXI to PCIe Translation: 000000056710000

AXI Bar_1
- Aperture Base Address: 0000000ABCDE000
- Aperture High Address: 00000000ABCDFFFF
- AXI to PCIe Translation: 0x50000000FEDC0(

AXI Bar_2
- Aperture Base Address: 0000000FE000000
- Aperture High Address: x00000000FFFFFFFF
- AXI to PCIe Translation: 000000040000000

AXI Bar_3
- Aperture Base Address: 000000000000000
- Aperture High Address: 000000000000FFF
- AXI to PCIe Translation: 0x600000087654

AXI Bar_4
- Aperture Base Address: x0000000000000000
- Aperture High Address: x0000000000000000
- AXI to PCIe Translation: x0000000000000000

AXI Bar_5
- Aperture Base Address: x0000000000000000
- Aperture High Address: x0000000000000000
- AXI to PCIe Translation: x0000000000000000

- Accessing the Bridge AXIBAR_0 with address `0x0000_12340ABC` on the AXI bus yields `0x56710ABC` on the bus for PCIe.
- Accessing the Bridge AXIBAR_1 with address `0x0000_ABCDF123` on the AXI bus yields `0x50000000FEDC1123` on the bus for PCIe.
- Accessing the Bridge AXIBAR_2 with address `0x0000_FFFEDCBA` on the AXI bus yields `0x41FEDCBA` on the bus for PCIe.

- Accessing the Bridge AXIBAR_3 with address `0x0000_00000071` on the AXI bus yields `0x6000000087654071` on the bus for PCIe.

Addressing Checks

When setting the following parameters for PCIe® address mapping, `C_PCIEBAR2AXIBAR_n` and `PF0_BARn_APERTURE_SIZE`, be sure these are set to allow for the addressing space on the AXI system. For example, the following setting is illegal and results in an invalid AXI address.

```
C_PCIEBAR2AXIBAR_n=0x00000000_FFFFF000
PF0_BARn_APERTURE_SIZE=0x06 (8 KB)
```

For an 8 Kilobyte BAR, the lower 13 bits must be zero. As a result, the `C_PCIEBAR2AXIBAR_n` value should be modified to be `0x00000000_FFFFE0000`. Also, check for a larger value on `PF0_BARn_APERTURE_SIZE` compared to the value assigned to the `C_PCIEBAR2AXIBAR_n` parameter. And example parameter setting follows.

```
C_PCIEBAR2AXIBAR_n=0xFFFF_E000
PF0_BARn_APERTURE_SIZE=0x0D (1 MB)
```

To keep the AXIBAR upper address bits as `0xFFFF_E000` (to reference bits [31:13]), the `PF0_BARn_APERTURE_SIZE` parameter must be set to `0x06` (8 KB).

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 functional mode 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 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 74: **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 75: **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.

Link Down Behavior

The normal operation of the AXI Bridge functional mode 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 functional mode, 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 functional mode 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

The Interrupt modes in the following section applies to AXI Bridge mode only.

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 MSI interrupt, and MSI interrupt takes precedence over Legacy interrupt. All of these interrupt modes are sent using the same `xdma0_usr_irq_*` interface and the core automatically picks the best available interrupt mode at runtime.

Legacy Interrupts

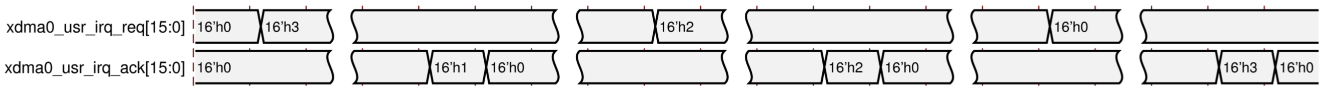
Asserting one or more bits of `xdma0_usr_irq_req` when legacy interrupts are enabled causes the IP to issue a legacy interrupt over PCIe. Multiple bits may be asserted simultaneously but each bit must remain asserted until the corresponding `xdma0_usr_irq_ack` bit has been asserted. After a `xdma0_usr_irq_req` bit is asserted, it must remain asserted until the corresponding `xdma0_usr_irq_ack` bit is asserted and the interrupt has been serviced and cleared by the Host. The `xdma0_usr_irq_ack` assertion indicates the requested interrupt has been sent on the PCIe block. This will ensure interrupt pending register within the IP remains asserted when queried by the Host's Interrupt Service Routine (ISR) to determine the source of interrupts. You must implement a mechanism in the user application to know when the interrupt routine has been serviced. This detection can be done in many different ways depending on your application and your use of this interrupt pin. This typically involves a register (or array of registers) implemented in the user application that is cleared, read, or modified by the Host software when an interrupt is serviced.

After the `xdma0_usr_irq_req` bit is deasserted, it cannot be reasserted until the corresponding `xdma0_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 `xdma0_usr_irq_ack` occurred, the `xdma0_usr_irq_req` wire can be reasserted to generate another legacy interrupt.

The `xdma0_usr_irq_req` bit can be mapped to legacy interrupt INTA, INTB, INTC, INTD through the configuration registers. The following figure shows the legacy interrupts.

This figure shows only the handshake between `xdma0_usr_irq_req` and `xdma0_usr_irq_ack`. The user application might not clear or service the interrupt immediately, in which case, you must keep `xdma0_usr_irq_req` asserted past `xdma0_usr_irq_ack`.

Figure 46: **Legacy Interrupts**



MSI and Internal MSI-X Interrupts

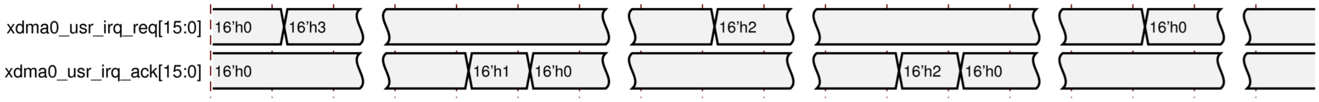
Asserting one or more bits of `xdma0_usr_irq_req` causes the generation of an MSI or MSI-X interrupt if MSI or MSI-X is enabled. If both MSI and MSI-X capabilities are enabled, an MSI-X interrupt is generated. The Internal MSI-X interrupts mode is enabled when you set the MSI-X Implementation Location option to Internal in the PCIe Misc Tab.

After a `xdma0_usr_irq_req` bit is asserted, it must remain asserted until the corresponding `xdma0_usr_irq_ack` bit is asserted and the interrupt has been serviced and cleared by the Host. The `xdma0_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 `xdma0_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.

This figure shows only the handshake between `xdma0_usr_irq_req` and `xdma0_usr_irq_ack`. Your application might not clear or service the interrupt immediately, in which case, you must keep `xdma0_usr_irq_req` asserted past `xdma0_usr_irq_ack`.

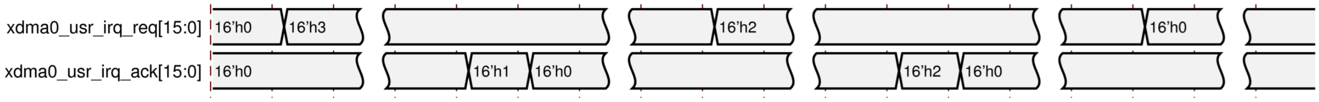
Figure 47: **MSI Interrupts**



The following figure shows the MSI-X interrupt.

This figure shows only the handshake between `xdma0_usr_irq_req` and `xdma0_usr_irq_ack`. Your application might not clear or service the interrupt immediately, in which case, you must keep `xdma0_usr_irq_req` asserted past `xdma0_usr_irq_ack`.

Figure 48: **MSI-X Interrupts**



Root Port

When configured to support Root Port functionality, the AXI Bridge functional mode 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.

Power Limit Message TLP

The AXI Bridge functional mode 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 functional mode 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 functional mode 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.

Configuration Transaction Timeout

Configuration transactions are non-posted transactions. The AXI Bridge functional mode 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 76: **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.

MSI Interrupt

The IP will decode 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 TLP received from the link with an address that falls within a 4 Kb window from the base address programmed in those registers will be treated as an MSI interrupt, and will not be forwarded to the M_AXI(B) interface. When an MSI interrupt is received, the Interrupt Decode register bit[17] will be set. If the Interrupt Mask register bit[17] is also set, the `interrupt_out` pin is asserted. After receiving this interrupt, the user application must follow the following procedure to service the interrupt:

1. Optional: Write 0 to the Interrupt Mask register bit [17] to deassert the `interrupt_out` pin while the interrupt is being serviced.
2. Read the Root Port Status/Control Register bit [18] to check if it is not empty.
3. Read the Root Port Status/Control Register bit [19] to check if it has overflowed.
4. If the interrupt FIFO is not empty, read the Root Port Interrupt FIFO Read Register 2 to check MSI Message Data from the received MSI interrupt. This is used by the user application to determine the interrupt vector number and can also be used to determine the source of the interrupt.
5. Write 1 to the Root Port Interrupt FIFO Read Register 1 bit [31] to remove the interrupt user has just read from the FIFO.
6. Repeat from step 2 until the FIFO is indicated as empty.
7. If at any time during this process, the FIFO was indicated as overflowed (status from step 2), the user application must check any unserviced interrupt vectors to check for any pending interrupts on that line. Failure to do this before continuing can leave some interrupt vector unserviced.
8. Write 1 to the Interrupt Decode Register bit [17] to clear the MSI interrupt bit.
9. If step 1 was executed, write 1 to the Interrupt Mask Register bit [17] to re-enable the `interrupt_out` pin for future MSI 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.

Interrupt Decode Mode

Legacy INTx Interrupt

When the IP has received an INTx interrupt, the Root Port Interrupt Decode 2 register is set. If the Root Port Interrupt Decode 2 Mask 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 Decode 2 Mask register to deassert an interrupt line while the interrupt is being serviced.
2. Read the Root Port 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 Decode 2 Mask register to re-enable an interrupt line for future INTx interrupt.

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 Root Port MSI Interrupt Decode 1 or Root Port MSI Interrupt Decode 2 register is set. If the Root Port MSI Interrupt Decode 1 or Root Port MSI Interrupt Decode 2 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 Root Port MSI Interrupt Decode 1 or 2 Mask register to deassert the `interrupt_out_msi_vec*` pins while the interrupt is being serviced.
2. Read the Root Port MSI Interrupt Decode 1 or 2 register to check which interrupt vector is asserted.
3. Write 1 to the Root Port MSI Interrupt Decode 1 or 2 register to clear the MSI interrupt bit.
4. If step 1 was executed, write 1 to the Root Port MSI Interrupt Decode 1 or 2 Mask register bit to re-enable the `interrupt_out_msi_vec*` pins for future MSI interrupts.

MSI-X Interrupt

All MSI-X interrupts must be decoded by the user application externally to the IP. To do this, the user application must set all 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 the Root Port MSI Base Register 2. All MSI-X interrupts are forwarded to the M_AXI(B) interface.

All TLPs forwarded to M_AXI(B) interface are subject to PCIe-to-AXI Address translation.

Port Description

Global Signals

The interface signals for the Bridge are described in the following table.

Table 77: **Global Signals**

Signal Name	I/O	Description
gt_refclk0_p/gt_refclk0_n	I	GT reference clock.
pci_gt_txp/pci_gt_txn [PL_LINK_CAP_MAX_LINK_WIDTH-1:0]	O	PCIe TX serial interface.
pci_gt_rxp/pci_gt_rxn [PL_LINK_CAP_MAX_LINK_WIDTH-1:0]	I	PCIe RX serial interface.
pcie0_user_lnk_up	O	Output active-High identifies that the PCI Express core is linked up with a host device.
pcie0_user_clk	O	User clock out. PCIe derived clock output for all interface signals output/input to AXI Bridge. Use this clock to drive inputs and gate outputs from AXI Bridge.
dma0_user_reset	O	User reset out. AXI reset signal synchronous with the clock provided on the pcie0_user_clk output. This reset should drive all corresponding AXI Interconnect signals.
cpm_cor_irq	O	Reserved
cpm_misc_irq	O	Reserved
cpm_uncor_irq	O	Reserved
cpm_irq0	I	Reserved
cpm_irq1	I	Reserved

AXI Slave Interface

AXI Bridge Slave ports are connected from the Versal™ ACAP programmable Network on Chip (NoC) to the CPM DMA internally. For slave bridge AXI-MM details and configuration, see *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313).

AXI Master Interface

AXI4 Memory Mapped (MM) Master ports are connected from the Versal ACAP Network on Chip (NoC) to the CPM DMA internally. For details, see *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313). The AXI4 MM Master interface can be connected to the DDR or the PL, depending on the NoC configuration.

AXI4-Lite Master Interface

AXI4-Lite Master ports are connected from the CPM to the Versal ACAP Network on Chip (NoC) internally. For details, see *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313).

Use the SmartConnect IP to connect the NoC to the AXI4-Lite Master interface. For details, see *SmartConnect LogiCORE IP Product Guide* (PG247).

AXI Bridge for PCIe Interrupts

Table 78: **AXI Bridge for PCIe Interrupts**

Signal Name	I/O	Description
xdma0_usr_irq_req[NUM_USR_IRQ-1:0]	I	User interrupt request. Asset to generate an interrupt and maintain assertion until interrupt is serviced.
xdma0_usr_irq_ack[NUM_USR_IRQ-1:0]	O	User interrupt acknowledge. Indicates that the interrupt has been set on PCIe. Two acks are generated for legacy interrupt. One ack is generated for MSI/MSI-X interrupts.
xdma0_usr_irq_fnc[7:0]	I	Function The function of the vector to be sent.

***Note*:** The `xdma0_` prefix in the above signal names will be changed to `dma0_*` in a future release.

NUM_USR_IRQ is selectable and it ranges from 0 to 15. Each bits in `xdma0_usr_irq_req` bus corresponds to the same bits in `xdma0_usr_irq_ack`. For example, `xdma0_usr_irq_ack[0]` represents an `ack` for `xdma0_usr_irq_req[0]`.

Register Space

Bridge register space can be accessed using AXI Slave interface and user can also access Host memory space.

Table 79: **AXI Slave Bridge Register Space**

Register Space	AXI Slave Interface Address Range	Details
Bridge registers	0x6_0000_0000	Described in Bridge register space CSV file. See Bridge Register Space for details.
Slave Bridge access to Host memory space	0xE001_0000 - 0x EFFF_FFFF 0x6_1100_0000 - 0x7_FFFF_FFFF 0x80_0000_0000 - 0xBF_FFFF_FFFF	Address range for Slave bridge access is set during IP customization in the Address Editor tab of the Vivado IDE.

Bridge register descriptions are found in `cpm-bridge-v2-1-registers.csv` available in the register map files.

To locate the register space information:

1. Download the register map files from the Xilinx website.
2. Extract the ZIP file contents into any write-accessible location.
3. Refer to the `cpm-bridge-v2-1-registers.csv` file.

Slave Bridge Registers Limitations

The Register Space mentioned in this document can also be accessible through the AXI4 Memory Mapped Slave interface. All accesses to these registers will be based on the following AXI Base Addresses:

- For QDMA registers: Base Address = `0x6_1000_0000`
- For XDMA registers: Base Address = `0x6_1002_0000`
- For Bridge registers: Base Address = `0x6_0000_0000`

The offsets within each register space are the same as listed for the PCIe BAR accesses.

Please make sure that all transactions targeting these register spaces have AWCACHE[1] and ARCACHE[1] set to `1'b0` (Non-Modifiable) and only access it in 4 Bytes transactions.

- All transactions originating from Programmable Logic (PL) region, must have an AXI Master that sets AxCACHE[1] = `1'b0` before it enters the AXI NOC.
- All transactions originating from the APU or RPU must be defined by a Memory Attribute `nGnRnE` or `nGnRE` to ensure AxCACHE[1] = `1'b0`.

- All transactions originating from PPU has no additional requirement necessary.

Chapter 9

Design Flow Steps

This section describes customizing and generating the functional mode, constraining the functional mode, and the simulation, synthesis, and implementation steps that are specific to this IP functional mode. 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)

AXI Bridge Lab

An AXI Bridge tutorial lab will be added in a future release.

Chapter 10

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

Solution Centers

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

The Solution Center specific to the AXI Bridge is 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 functional mode can be located by using the Search Support box on the main Xilinx support web page. To maximize your search results, use keywords such as:

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

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

Master Answer Record for the Core

AR 75396.

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.

Registers

A complete list of registers and attributes for the AXI Bridge Subsystem is available in the *Versal ACAP Register Reference* (AM012). Reviewing the registers and attributes might be helpful for advanced debugging.

***Note*:** The attributes are set during IP customization in the Vivado IP catalog. After core customization, attributes are read-only.

Chapter 11

Upgrading

This appendix is not applicable for the first release of the functional mode.

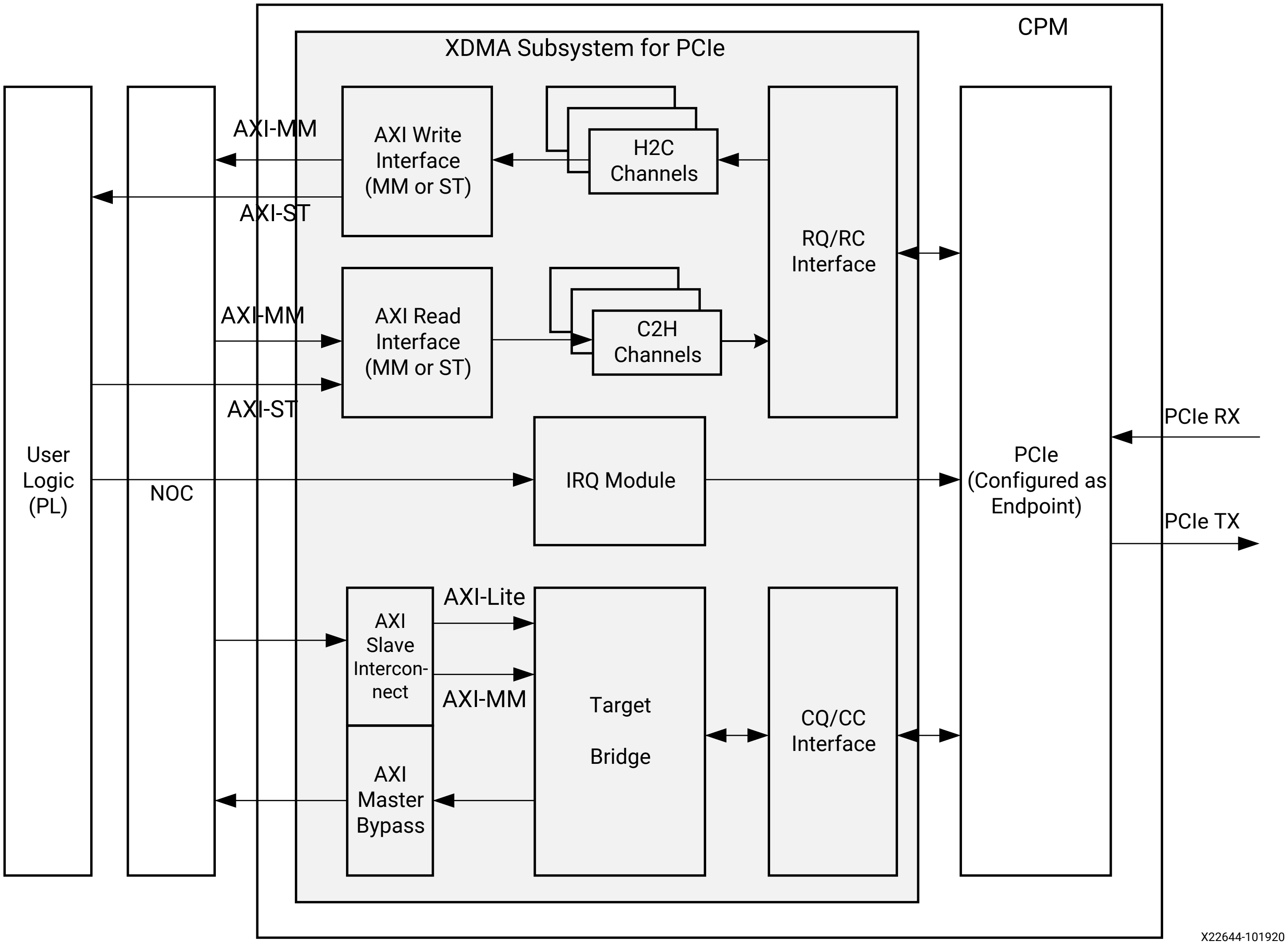
Section IV

XDMA Subsystem

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 functional mode 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.

Figure 49: **XDMA Subsystem**



This diagram refers to the Requester Request (RQ)/Requester Completion (RC) interfaces, and the Completer Request (CQ)/Completer Completion (CC) interfaces.

Limitations

The limitations of the XDMA are as follows:

- SR-IOV
- Example design not supported for all configurations
- Narrow burst (not supported on the master interface)

Architecture

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 registers, or the CQ bypass port. After the downstream user logic has returned data for a non-posted request, the target bridge generates a read completion TLP and sends it to the PCIe IP over the CC bus.

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

H2C Channel

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

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

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.

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

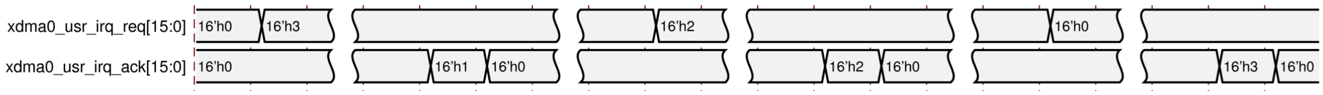
Asserting one or more bits of xdma0_usr_irq_req when legacy interrupts are enabled causes the DMA to issue a legacy interrupt over PCIe. Multiple bits may be asserted simultaneously but each bit must remain asserted until the corresponding xdma0_usr_irq_ack bit has been asserted. After a xdma0_usr_irq_req bit is asserted, it must remain asserted until the corresponding xdma0_usr_irq_ack bit is asserted and the interrupt has been serviced and cleared by the Host. The xdma0_usr_irq_ack assertion indicates the requested interrupt has been sent to the PCIe block. This will ensure interrupt pending register within the IP remains asserted when queried by the Host's Interrupt Service Routine (ISR) to determine the source of interrupts. You must implement a mechanism in the user application to know when the interrupt routine has been serviced. This detection can be done in many different ways depending on your application and your use of this interrupt pin. This typically involves a register (or array of registers) implemented in the user application that is cleared, read, or modified by the Host software when an interrupt is serviced.

After the xdma0_usr_irq_req bit is deasserted, it cannot be reasserted until the corresponding xdma0_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 xdma0_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 xdma0_usr_irq_ack. Your application might not clear or service the interrupt immediately, in which case, you must keep xdma0_usr_irq_req asserted past xdma0_usr_irq_ack.

Figure 50: **Legacy Interrupts**



MSI and MSI-X Interrupts

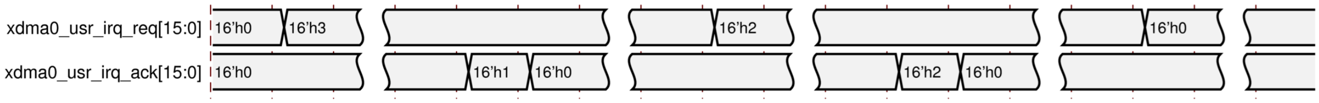
Asserting one or more bits of `xdma0_usr_irq_req` causes the generation of an MSI or MSI-X interrupt if MSI or MSI-X is enabled. If both MSI and MSI-X capabilities are enabled, an MSI-X interrupt is generated.

After a `xdma0_usr_irq_req` bit is asserted, it must remain asserted until the corresponding `xdma0_usr_irq_ack` bit is asserted and the interrupt has been serviced and cleared by the Host. The `xdma0_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 `xdma0_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 `xdma0_usr_irq_req` and `xdma0_usr_irq_ack`. Your application might not clear or service the interrupt immediately, in which case, you must keep `xdma0_usr_irq_req` asserted past `xdma0_usr_irq_ack`.

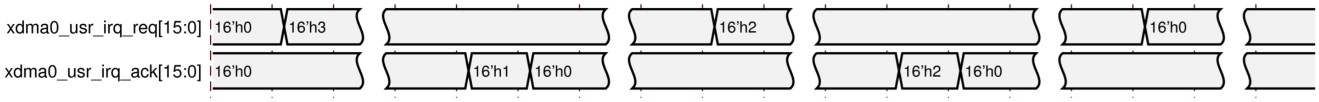
Figure 51: **MSI Interrupts**



The following figure shows the MSI-X interrupt.

Note: This figure shows only the handshake between `xdma0_usr_irq_req` and `xdma0_usr_irq_ack`. Your application might not clear or service the interrupt immediately, in which case, you must keep `xdma0_usr_irq_req` asserted past `xdma0_usr_irq_ack`.

Figure 52: **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.

Chapter 12

Product Specification

DMA Operations

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 80: **Descriptor Format**

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

Table 81: **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.

Table 81: **Descriptor Fields** *(cont'd)*

Offset	Field	Bit Index	Sub Field	Description
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.
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 (512 * 512) 32 KB deep FIFO to hold all descriptors in the descriptor engine. This descriptor FIFO is shared with all selected channels.

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 82: **Completed Descriptor Count Writeback Format**

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

Table 83: **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 85: 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 84: **C2H Stream Writeback Format**

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

Table 85: **C2H Stream Writeback Fields**

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

Address Alignment

Table 86: **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 87: **Length Granularity**

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

Notes:

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

Parity

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.

Port Description

Global Signals

Table 88: **Global Signals**

Signal Name	Direction	Description
gt_refclk0_p/gt_refclk0_n	I	GT reference clock
pci_gt_txp/pci_gt_txn [PL_LINK_CAP_MAX_LINK_WIDTH-1:0]	O	PCIe TX serial interface.
pci_gt_rxp/pci_gt_rxn [PL_LINK_CAP_MAX_LINK_WIDTH-1:0]	I	PCIe RX serial interface.
pcie0_user_lnk_up	O	Output active-High identifies that the PCI Express core is linked up with a host device.
pcie0_user_clk	O	User clock out. PCIe derived clock output for all interface signals output/input to QDMA. Use this clock to drive inputs and gate outputs from QDMA.
dma0_axi_aresetn	O	User reset out. AXI reset signal synchronous with the clock provided on the pcie0_user_clk output. This reset should drive all corresponding AXI Interconnect aresetn signals.
dma0_soft_resetn	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 deassert this port.

AXI Slave Interface

AXI Bridge Slave ports are connected from the Versal ACAP Network on Chip (NoC) to the CPM DMA internally. For Slave Bridge AXI-MM details, see *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313).

To access XDMA registers, you must follow the protocols outlined in the AXI Slave Bridge Register Limitations section.

Related Information

Slave Bridge Registers Limitations

AXI4 Memory Mapped Interface

AXI4 Memory Mapped (MM) Master ports are connected from the CPM to the Versal ACAP Network on Chip (NoC) internally. For details, see *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313). The AXI4 MM Master interface can be connected to DDR or to the PL user logic, depending on the NoC configuration.

AXI4-Lite Master Interface

AXI4-Lite Master ports are connected from the CPM to the Versal ACAP Network on Chip (NoC) internally. For details, see *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313).

Use the SmartConnect IP to connect the NoC to the AXI4-Lite Master interface. For details, see *SmartConnect LogiCORE IP Product Guide* (PG247).

H2C Channel 0-3 AXI4-Stream Interface Signals

Table 89: **H2C Channel 0-3 AXI4-Stream Interface Signals**

Signal Name[1]	Direction	Description
dma0_m_axis_h2c_*x*_tready	I	Assertion of this signal by the user logic indicates that it is ready to accept data. Data is transferred across the interface when dma0_m_axis_h2c_tready and dma0_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.
dma0_m_axis_h2c_*x*_tlast	O	The DMA asserts this signal in the last beat of the DMA packet to indicate the end of the packet.
dma0_m_axis_h2c_*x*_tdata [DATA_WIDTH-1:0]	O	Transmit data from the DMA to the user logic.
dma0_m_axis_h2c_*x*_tkeep [DATA_WIDTH/8-1:0]	O	tkeep will be all 1s except when `dma0_m_axis_h2c_x_tlast` is asserted.
dma0_m_axis_h2c_*x*_tvalid	O	The DMA asserts this whenever it is driving valid data on dma0_m_axis_h2c_tdata.
dma0_m_axis_h2c_tuser [DATA_WIDTH/8-1:0]	O	Parity bits. This port is enabled only in Propagate Parity mode.

Notes:

1. _x in the signal name changes based on the channel number 0, 1, 2, and 3. For example, for channel 0 use the dma0_m_axis_h2c_tready_0 port, and for channel 1 use the dma0_m_axis_h2c_tready_1 port.

C2H Channel 0-3 AXI4-Stream Interface Signals

Table 90: **C2H Channel 0-3 AXI4-Stream Interface Signals**

Signal Name[1]	Direction	Description
dma0_s_axis_c2h_*x*_tready	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.
dma0_s_axis_c2h_*x*_tlast	I	The user logic asserts this signal to indicate the end of the DMA packet.
dma0_s_axis_c2h_*x*_tdata [DATA_WIDTH-1:0]	I	Transmits data from the user logic to the DMA.
dma0_s_axis_*x*_tkeep [DATA_WIDTH/8-1:0]	I	tkeep must all be 1s for all cycles except when dma0_s_axis_c2h_x_tlastis asserted. The asserted tkeep bits need to be packed to the lsb, indicating contiguous data.
dma0_s_axis_c2h_*x*_tvalid	I	The user logic asserts this whenever it is driving valid data on s_axis_c2h_tdata.
dma0_s_axis_c2h_*x*_tuser [DATA_WIDTH/8-1:0]	I	Parity bits. This port is enabled only in Propagate Parity mode.

Notes:

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

Interrupt Interface

Table 91: **Interrupt Interface**

Signal Name	Direction	Description
dma0_usr_irq_req[NUM_USR_IRQ-1:0]	I	Assert to generate an interrupt. Maintain assertion until interrupt is serviced.
dma0_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.
dma0_usr_irq_func[7:0]	I	In most cases these signals are tied to 0s for function 0.

NUM_USR_IRQ is selectable and it ranges from 0 to 15. Each bits in `dma0_usr_irq_req`bus corresponds to the same bits in `dma0_usr_irq_ack`. For example, `dma0_usr_irq_ack[0]` represents an `ack` for `dma0_usr_irq_req[0]`.

Channel 0-3 DMA Status Interface

Table 92: **Channel 0-3 DMA Status Interface**

Signal Name	Direction	Description
dma0_h2c_sts_x [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
dma0_c2h_sts_x [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

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 `dma0_c2h_sts_0` port, and for channel 1 use the `dma0_c2h_sts_1` port.

Descriptor Bypass Interface

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

Table 93: **H2C 0-3 Descriptor Bypass Interface description**

Port	Direction	Description
dma0_h2c_dsc_byp_x_ready	O	Channel is ready to accept new descriptors. After dma0_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.
dma0_h2c_dsc_byp_x_load	I	Write the descriptor presented at dma0_h2c_dsc_byp_data into the channel's descriptor buffer.
dma0_h2c_dsc_byp_src_x_addr[63:0]	I	Descriptor source address to be loaded.
dma0_h2c_dsc_byp_dst_x_addr[63:0]	I	Descriptor destination address to be loaded.
dma0_h2c_dsc_byp_x_len[27:0]	I	Descriptor length to be loaded.

Table 93: **H2C 0-3 Descriptor Bypass Interface description** *(cont'd)*

Port	Direction	Description
dma0_h2c_dsc_byp_x_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.

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 `dma0_h2c_dsc_byp_0_ctl[15:0]` port, and for channel 1 use the `dma0_h2c_dsc_byp_1_ctl[15:0]` port.

Table 94: **C2H 0-3 Descriptor Bypass Ports**

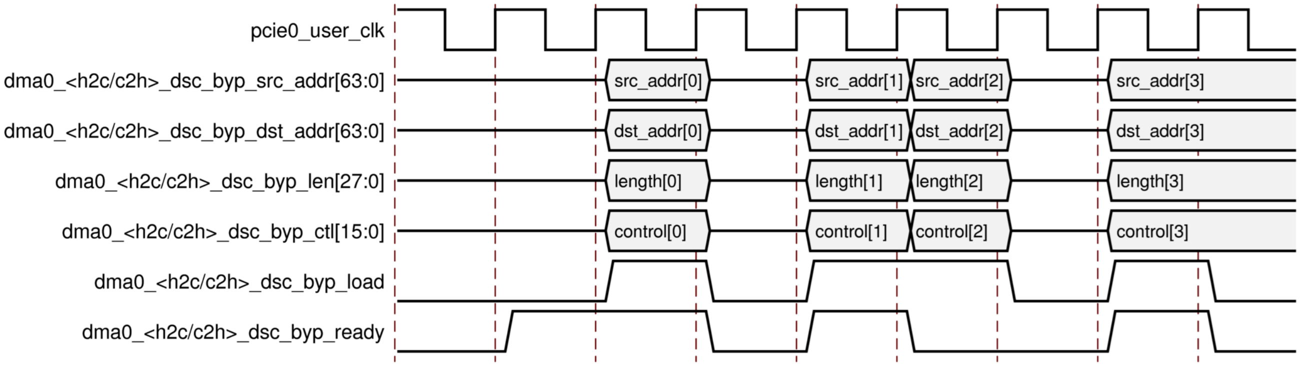
Port	Direction	Description
dma0_c2h_dsc_byp_x_ready	O	Channel is ready to accept new descriptors. After dma0_c2h_dsc_byp_ready is deasserted, one additional descriptor can be written. The Control register 'Run' bit must be asserted before the channel accepts descriptors.
dma0_c2h_dsc_byp_x_load	I	Descriptor presented at dma0_c2h_dsc_byp_* is valid.
dma0_c2h_dsc_byp_src_x_addr[63:0]	I	Descriptor source address to be loaded.
dma0_c2h_dsc_byp_dst_x_addr[63:0]	I	Descriptor destination address to be loaded.
dma0_c2h_dsc_byp_x_len[27:0]	I	Descriptor length to be loaded.
dma0_c2h_dsc_byp_x_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.

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 `dma0_c2h_dsc_byp_0_ctl[15:0]` port, and for channel 1 use the `dma0_c2h_dsc_byp_1_ctl[15:0]` port.

The following timing diagram shows how to input the descriptor in descriptor bypass mode. When `dma0_<h2c|c2h>_dsc_byp_ready` is asserted, a new descriptor can be pushed in with the `dma0_<h2c|c2h>_dsc_byp_load` signal.

Figure 53: **Timing Diagram for Descriptor Bypass Mode**


pcie0_user_clk
dma0_<h2c/c2h>_dsc_byp_src_addr[63:0]
dma0_<h2c/c2h>_dsc_byp_dst_addr[63:0]
dma0_<h2c/c2h>_dsc_byp_len[27:0]
dma0_<h2c/c2h>_dsc_byp_ctl[15:0]
dma0_<h2c/c2h>_dsc_byp_load
dma0_<h2c/c2h>_dsc_byp_ready
src_addr[0]
src_addr[1]
src_addr[2]
src_addr[3]
dst_addr[0]
dst_addr[1]
dst_addr[2]
dst_addr[3]
length[0]
length[1]
length[2]
length[3]
control[0]
control[1]
control[2]
control[3]

> **IMPORTANT!** *Immediately after `dma0_<h2c/c2h>_dsc_byp_ready` is deasserted, one more descriptor can be pushed in. In the above timing diagram, a descriptor is pushed in when `dma0_<h2c/c2h>_dsc_byp_ready` is deasserted.*

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.

XDMA Address Register Space

All the registers are found in `cpm-xdma-v2-1-registers.csv` available in the register map files.

To locate the register space information:

1. Download the register map files from the Xilinx website.

2. Extract the ZIP file contents into any write-accessible location.

3. Refer to the `cpm-xdma-v2-1-registers.csv` file.

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 95: **PCIe to DMA Address Format**

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

Table 96: **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.
7:0	Byte Offset	The byte address of the register to be accessed within the target. Bits[1:0] must be 0.

AXI Slave Register Space

DMA register space can be accessed using AXI Slave interface. When AXI Slave Bridge mode is enabled (based on GUI settings) user can also access Bridge registers and can also access Host memory space.

Table 97: **AXI4 Slave Register Space**

Register Space	AXI Slave Interface Address Range	Details
Bridge registers	0x6_0000_0000	Described in Bridge register space CSV file. See Bridge Register Space for details.

Table 97: **AXI4 Slave Register Space** *(cont'd)*

Register Space	AXI Slave Interface Address Range	Details
DMA registers	0x6_1002_0000	Described in XDMA Address Register Space.
Slave Bridge access to Host memory space	0xE001_0000 - 0xEFFF_FFFF 0x6_1100_0000 - 0x7_FFFF_FFFF 0x80_0000_0000 - 0xBF_FFFF_FFFF	Address range for Slave bridge access is set during IP customization in the Address Editor tab of the Vivado IDE.

Bridge Register Space

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

All the bridge registers are listed in the `cpm-bridge-v2-1-registers.csv` available in the register map files.

To locate the register space information:

1. Download the register map files from the Xilinx website.
2. Extract the ZIP file contents into any write-accessible location.
3. Refer to the `cpm-bridge-v2-1-registers.csv` file.

DMA Register Space

The DMA register space is described in XDMA Address Register Space.

Chapter 13

Design Flow Steps

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

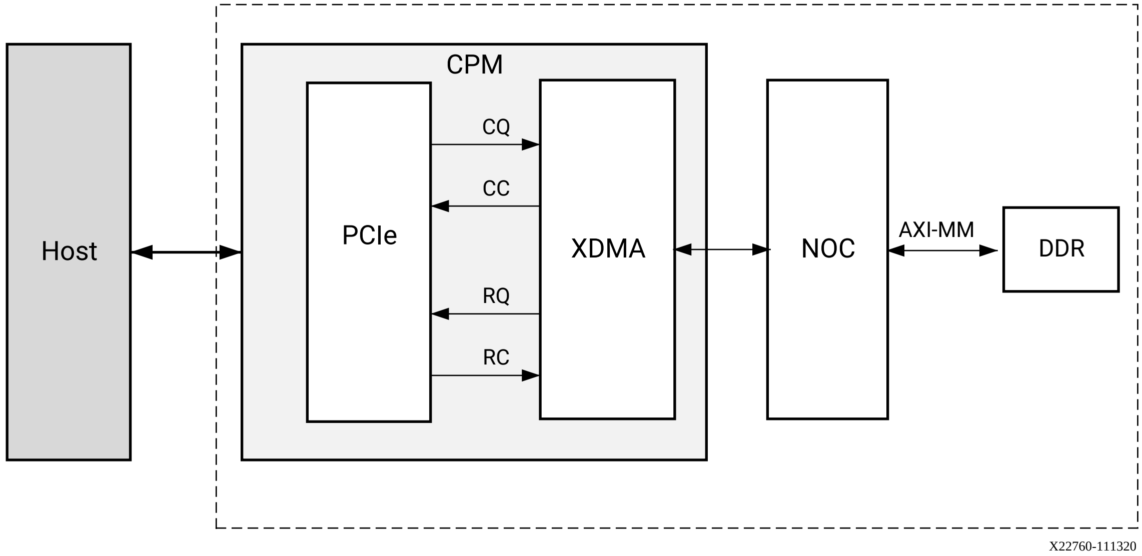
- *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
- *Vivado Design Suite User Guide: Designing with IP* (UG896)
- *Vivado Design Suite User Guide: Getting Started* (UG910)
- *Vivado Design Suite User Guide: Logic Simulation* (UG900)

XDMA AXI MM Interface to NoC and DDR Lab

This lab describes the process of generating a Versal™ ACAP XDMA design with AXI4 Memory Mapped interface connecting to DDR memory. This lab explains a step by step procedure to configure a Control, Interfaces and Processing System (CIPS) XDMA design and network on chip (NoC) IP. The following figure shows the AXI4 Memory Mapped (AXI-MM) interface to DDR using the NoC IP. At the end of this lab, you can synthesize and implement the design, and generate a Programmable Device Image (PDI) file. The PDI file is used to program the Versal ACAP and run data traffic on a system. For host to chip (H2C) transfers, data is read from Host, and sent to DDR memory. For chip to host (C2H) transfers, data is read from DDR memory and written to Host. Transfer can be initiated on all 4 channels.

This lab targets a xcvc1902-vsvd1760-1LP-e-S-es1 part on a VCK5000 board. This lab connects to DDR found outside the ACAP. A constraint file is provided for use with this lab. The constraints file lists all DDR pins and their placement. You can modify the constraint file based on your requirement and DDR part selection.

Figure 54: **AXI-MM Default Example Design**



Tutorial Design File

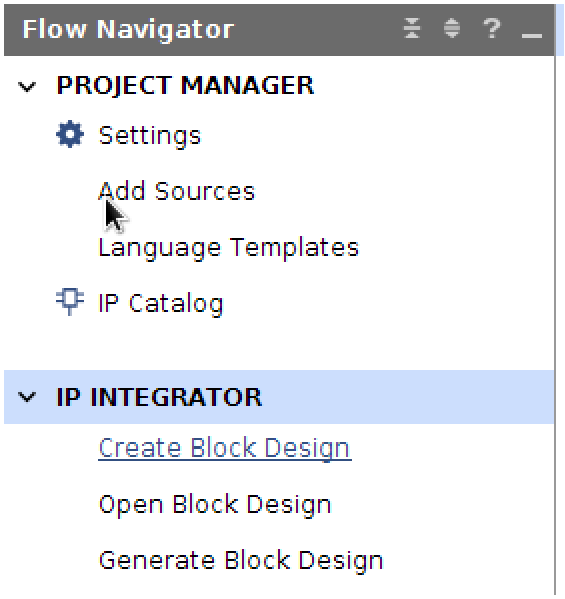
Before running the lab, download the `top_impl.xdc` constraints file available in the reference design file. To do so:

1. Download the reference design file from the Xilinx website.
2. Extract the ZIP file contents into any write-accessible location.
3. Locate the `top_impl.xdc` constraints file.

The provided `top_impl.xdc` constraints file contains the needed DDR pins and their placement for this tutorial lab. The constraints file can be modified as needed for later use.

Start the Vivado Design Suite

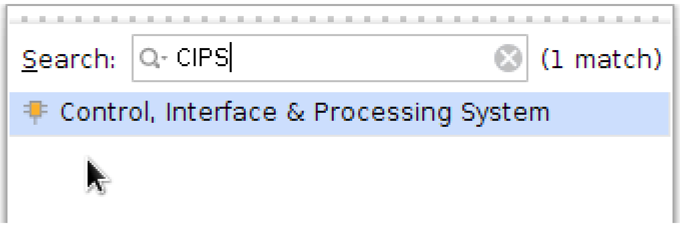
1. Open the Vivado® Design Suite.
2. Click **Create Project** from the Quick Start Menu.
3. Step through the popup menus to access the Default Part page.
4. In the Default Part page, search for and select: **xcvc1902-vsvd1760-1LP-e-S-es1**.
5. Continue to the Finish stage to create the new project and open Vivado
6. In the Vivado Flow Navigator, click **IP Integrator** → **Create Block Design**. A popup dialog displays to create the block design.



7. Click **OK**. An empty block design diagram canvas opens.

Instantiate the CIPS IP

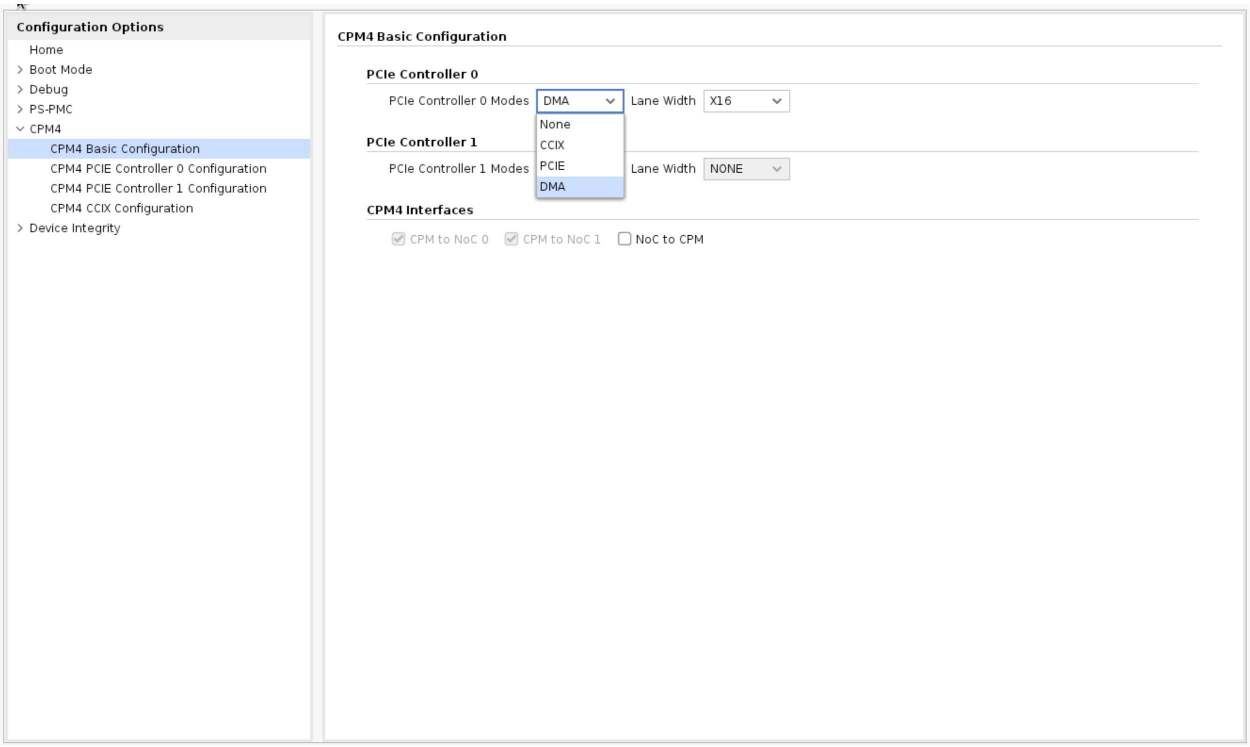
1. Right-click on the block design canvas, and from the context menu select **Add IP**.
2. The IP catalog pops up. In the Search field type `CIPS` to filter to the list of IP.



Search: CIPS (1 match)

3. From the filtered list, double-click the **Control, Interface, and Processing System** IP core to instantiate the IP on the block design canvas.
4. This adds the Versal CIPS IP to the canvas. Double-click the **Versal CIPS IP**.
5. The configuration dialog box for the Control, Interfaces and Processing System IP core displays. In the Configuration Options pane, expand **CPM**, and click **CPM Configuration**.
6. Set the PCIe0 Modes to **DMA**, and set the lane width to **X16**.

   Available lane widths are X4, X8 and X16. X1 and X2 are not supported.



CPM4 Basic Configuration

PCIe Controller 0

PCIe Controller 0 Modes: DMA   Lane Width: X16

PCIe Controller 1

PCIe Controller 1 Modes   Lane Width: NONE

CPM4 Interfaces

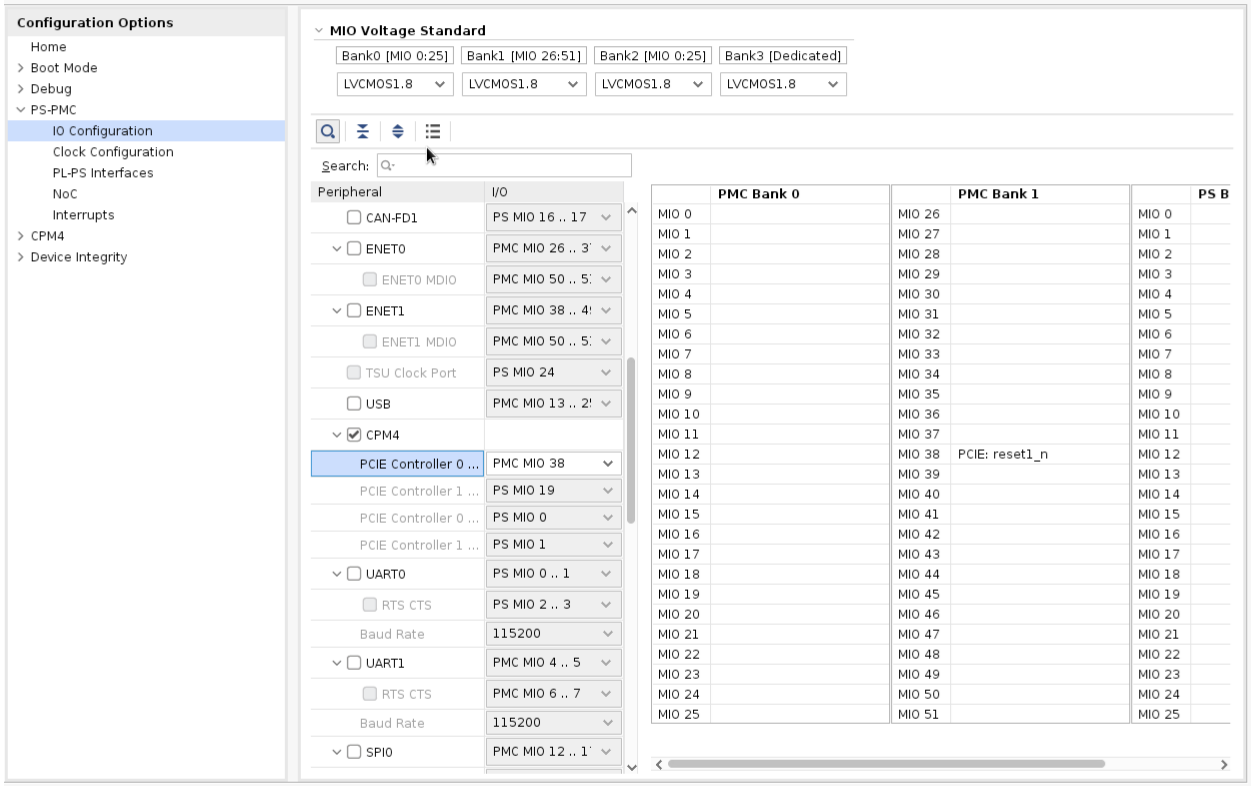
☑ CPM to NoC 0   ☑ CPM to NoC 1   □ NoC to CPM

7. In the Configuration Options pane, expand **PS-PMC**, and click **IO Configuration**.
8. The IO Configuration page displays with a list of options to configure the CPM-PCIe functional mode. In the Peripheral column, select the **PCIe Reset** checkbox.

   Notice that only A0 End Point is selectable in the I/O column.

   Notice also that the multi-use I/O (MIO) pin selected in PCIe reset is automatically connected to the PCIe Reset I/O, in this case MIO 38.
9. Next to A0 End Point, select **PS MIO 38**, which is the MIO pin that matches the MIO pin is connected in your board.

   Available MIO pin selections are PS MIO 18, PMC MIO 24, and PMC MIO 38.



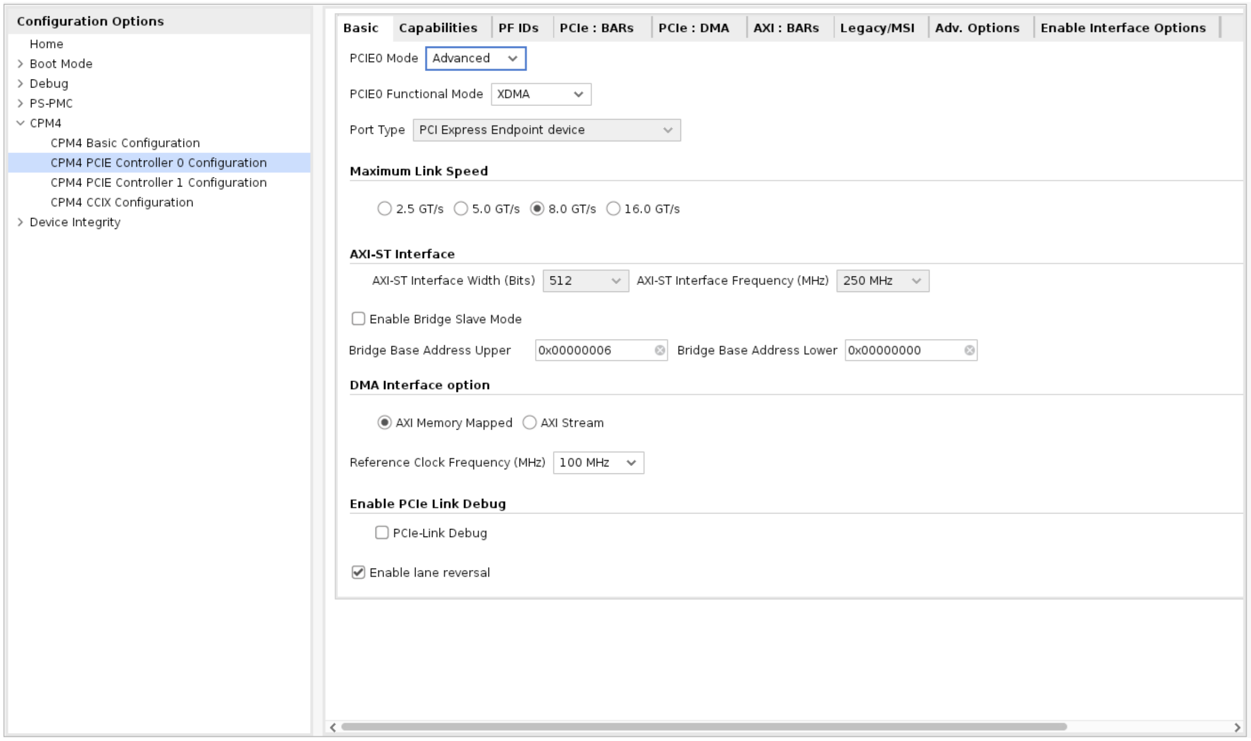
Bank0 [MIO 0:25]	Bank1 [MIO 26:51]	Bank2 [MIO 0:25]	Bank3 [Dedicated]
LVCMOS1.8	LVCMOS1.8	LVCMOS1.8	LVCMOS1.8

Peripheral	I/O
☐ CAN-FD1	PS MIO 16 .. 17
☐ ENET0	PMC MIO 26 .. 3'
☐ ENET0 MDIO	PMC MIO 50 .. 5:
☐ ENET1	PMC MIO 38 .. 4!
☐ ENET1 MDIO	PMC MIO 50 .. 5:
☐ TSU Clock Port	PS MIO 24
☐ USB	PMC MIO 13 .. 2!
☑ CPM4	
PCIE Controller 0 ...	PMC MIO 38
PCIE Controller 1 ...	PS MIO 19
PCIE Controller 0 ...	PS MIO 0
PCIE Controller 1 ...	PS MIO 1
☐ UART0	PS MIO 0 .. 1
☐ RTS CTS	PS MIO 2 .. 3
Baud Rate	115200
☐ UART1	PMC MIO 4 .. 5
☐ RTS CTS	PMC MIO 6 .. 7
Baud Rate	115200
☐ SPI0	PMC MIO 12 .. 1'

PMC Bank 0		PMC Bank 1		PS B
MIO 0		MIO 26		MIO 0
MIO 1		MIO 27		MIO 1
MIO 2		MIO 28		MIO 2
MIO 3		MIO 29		MIO 3
MIO 4		MIO 30		MIO 4
MIO 5		MIO 31		MIO 5
MIO 6		MIO 32		MIO 6
MIO 7		MIO 33		MIO 7
MIO 8		MIO 34		MIO 8
MIO 9		MIO 35		MIO 9
MIO 10		MIO 36		MIO 10
MIO 11		MIO 37		MIO 11
MIO 12		MIO 38	PCIE: reset1_n	MIO 12
MIO 13		MIO 39		MIO 13
MIO 14		MIO 40		MIO 14
MIO 15		MIO 41		MIO 15
MIO 16		MIO 42		MIO 16
MIO 17		MIO 43		MIO 17
MIO 18		MIO 44		MIO 18
MIO 19		MIO 45		MIO 19
MIO 20		MIO 46		MIO 20
MIO 21		MIO 47		MIO 21
MIO 22		MIO 48		MIO 22
MIO 23		MIO 49		MIO 23
MIO 24		MIO 50		MIO 24
MIO 25		MIO 51		MIO 25

CPM Configuration

1. In the Configuration Options pane, expand **CPM**, and click **PCIE0 Configuration** to customize the PCIe Port 0.
2. In the Basic tab, set the following options:
   - CPM Modes: **Advanced**
   - PCIE0 Functional Mode: **XDMA**
   - Maximum Link Speed: **8.0 GT/s** (Gen3)
   - DMA Interface option: **AXI Memory Mapped**



Configuration Options

- Home
- › Boot Mode
- › Debug
- › PS-PMC
- ⌄ CPM4
  - CPM4 Basic Configuration
  - CPM4 PCIE Controller 0 Configuration
  - CPM4 PCIE Controller 1 Configuration
  - CPM4 CCIX Configuration
- › Device Integrity

Basic | Capabilities | PF IDs | PCIe : BARs | PCIe : DMA | AXI : BARs | Legacy/MSI | Adv. Options | Enable Interface Options

PCIE0 Mode: Advanced

PCIE0 Functional Mode: XDMA

Port Type: PCI Express Endpoint device

Maximum Link Speed

○ 2.5 GT/s ○ 5.0 GT/s ● 8.0 GT/s ○ 16.0 GT/s

AXI-ST Interface

AXI-ST Interface Width (Bits): 512 AXI-ST Interface Frequency (MHz): 250 MHz

□ Enable Bridge Slave Mode

Bridge Base Address Upper: 0x00000006 Bridge Base Address Lower: 0x00000000

DMA Interface option

● AXI Memory Mapped ○ AXI Stream

Reference Clock Frequency (MHz): 100 MHz

Enable PCIe Link Debug

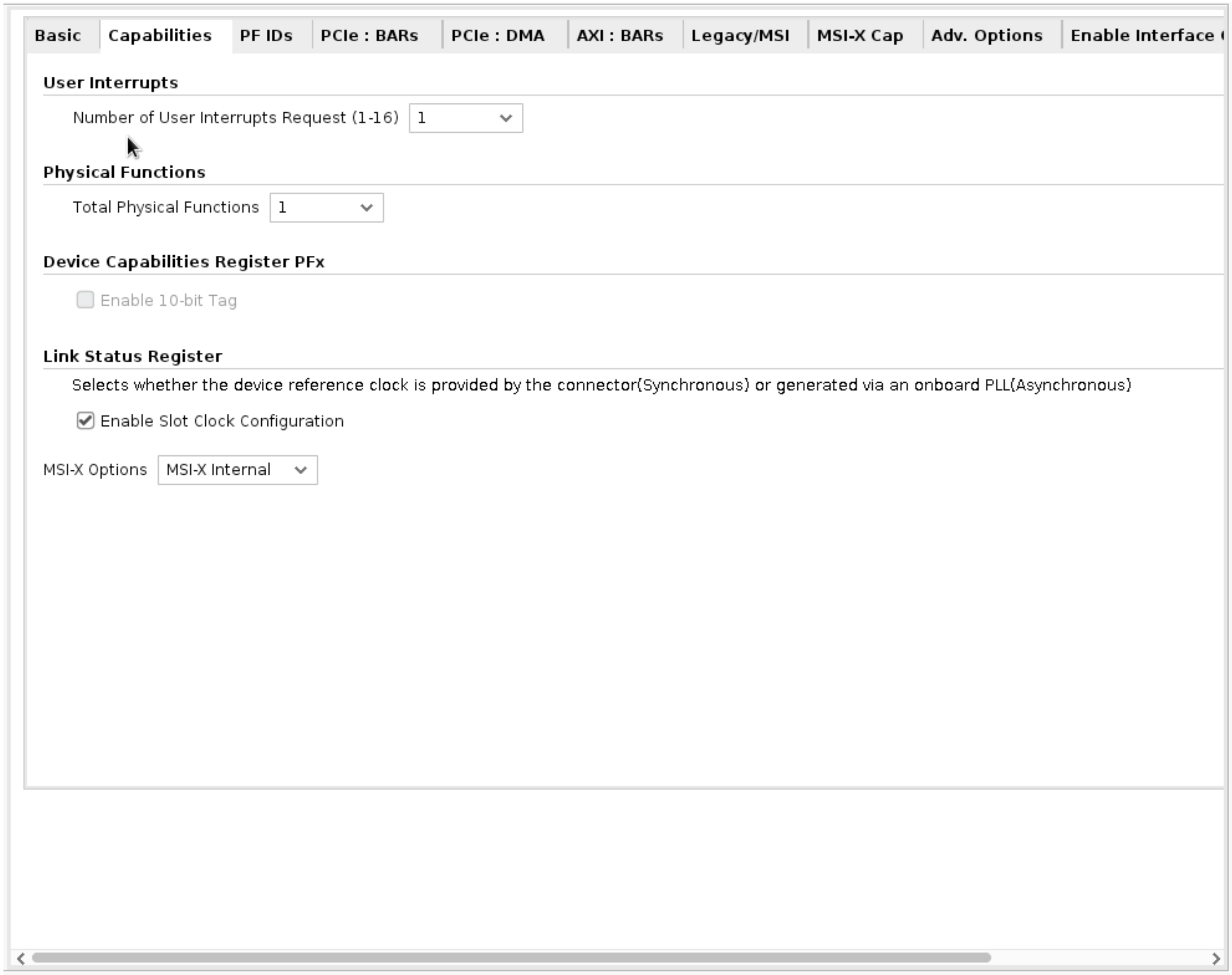
□ PCIe-Link Debug

☑ Enable lane reversal

3. In the Capabilities tab, set the following option:

   - MSI-X- Options: **MSI-X Internal**

   This option enables the CPM XDMA in MSI-X internal mode.



User Interrupts

Number of User Interrupts Request (1-16): 1

Physical Functions

Total Physical Functions: 1

Device Capabilities Register PFx

□ Enable 10-bit Tag

Link Status Register

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

☑ Enable Slot Clock Configuration

MSI-X Options: MSI-X Internal

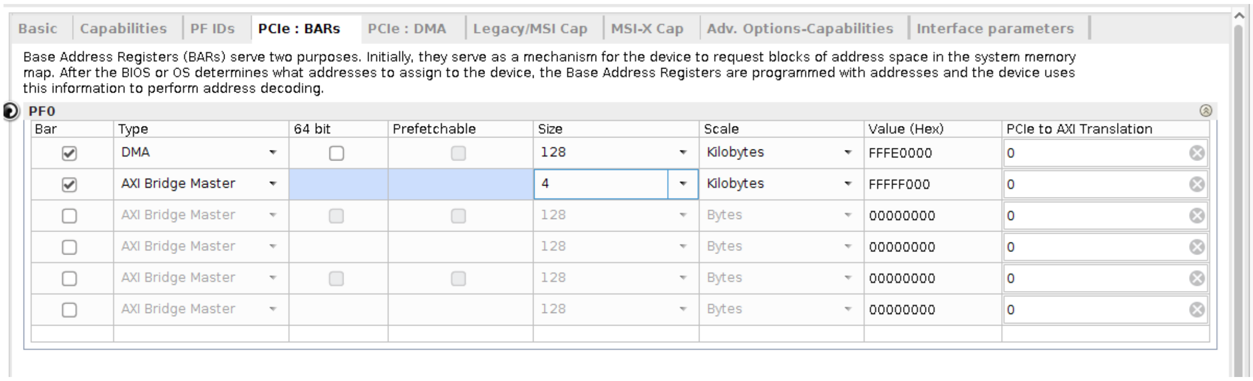
4. In the PCIe: BARs tab, set the following options:

   First row (for BAR0):

   - Select the **Bar** checkbox.
   - Set type to **DMA**.
   - Set size to **128** Kilobytes.

   Second row (for BAR1):

   - Select the **Bar** checkbox.
   - Set type to **AXI Bridge Master**.
   - Set size to **4** Kilobytes.

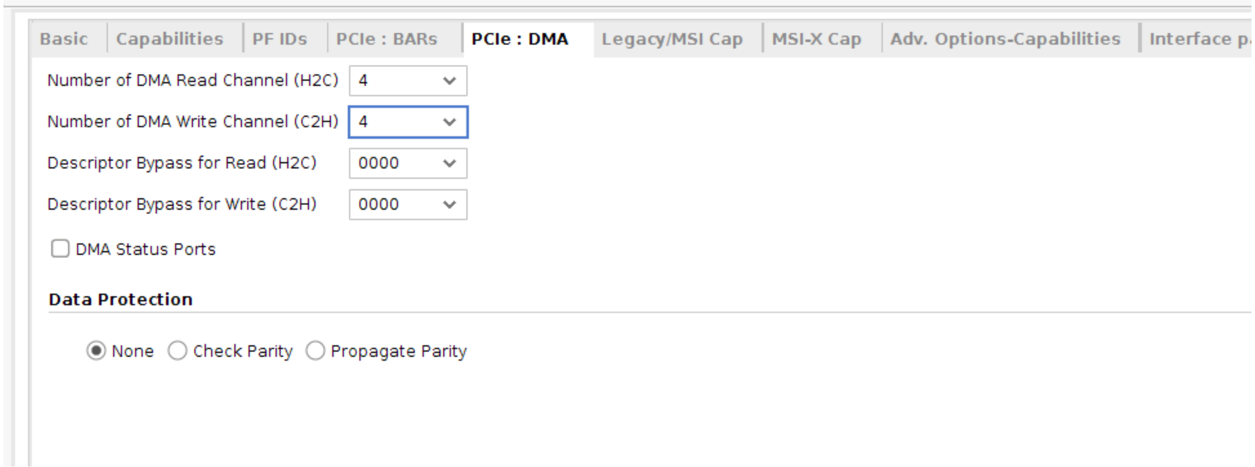


Base Address Registers (BARs) serve two purposes. Initially, they serve as a mechanism for the device to request blocks of address space in the system memory map. After the BIOS or OS determines what addresses to assign to the device, the Base Address Registers are programmed with addresses and the device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

PF0

Bar	Type	64 bit	Prefetchable	Size	Scale	Value (Hex)	PCIe to AXI Translation
☑	DMA	☐	☐	128	Kilobytes	FFFE0000	0
☑	AXI Bridge Master			4	Kilobytes	FFFFF000	0
☐	AXI Bridge Master	☐	☐	128	Bytes	00000000	0
☐	AXI Bridge Master			128	Bytes	00000000	0
☐	AXI Bridge Master	☐	☐	128	Bytes	00000000	0
☐	AXI Bridge Master			128	Bytes	00000000	0

5. In the PCIe: DMA tab, set the following options:
   - Number of DMA Read Channel (H2C): **4** (for 4 channels).
   - Number of DMA Write Channel (C2H): **4** (for 4 channels).



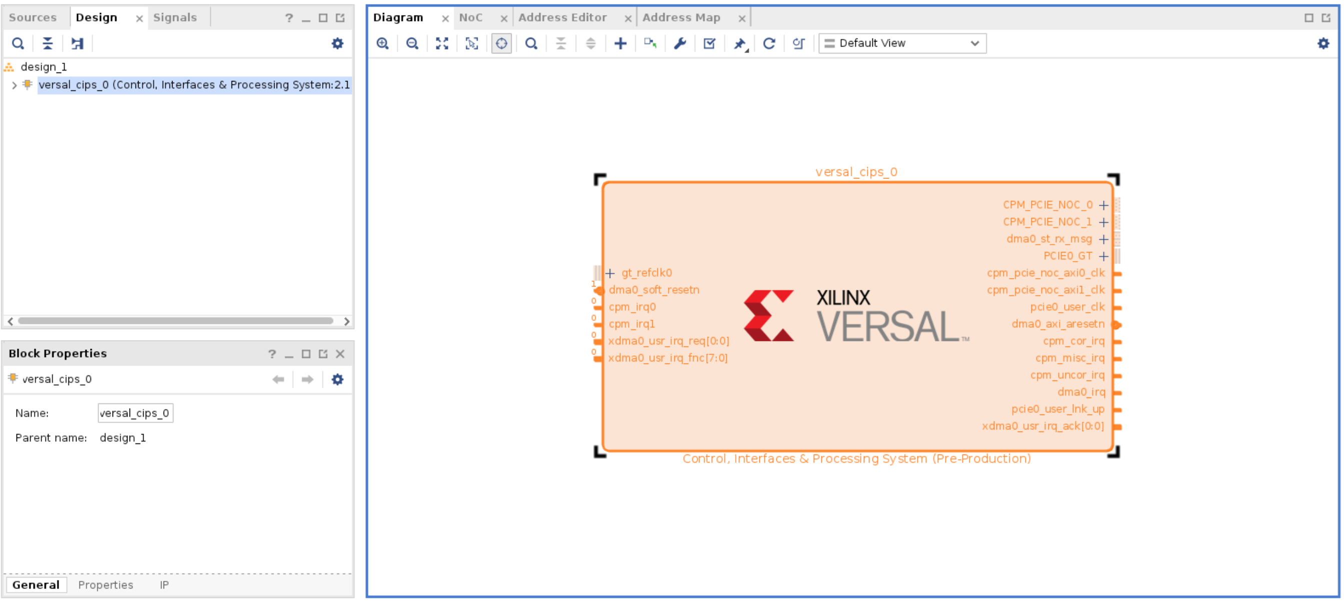
Number of DMA Read Channel (H2C): 4
Number of DMA Write Channel (C2H): 4
Descriptor Bypass for Read (H2C): 0000
Descriptor Bypass for Write (C2H): 0000
☐ DMA Status Ports

Data Protection

● None ○ Check Parity ○ Propagate Parity

6. In all other tabs, keep the default settings.
7. Click **OK** to generate the CIPS XDMA IP.

The generated IP core displays in the Diagram tab.



NoC Configuration

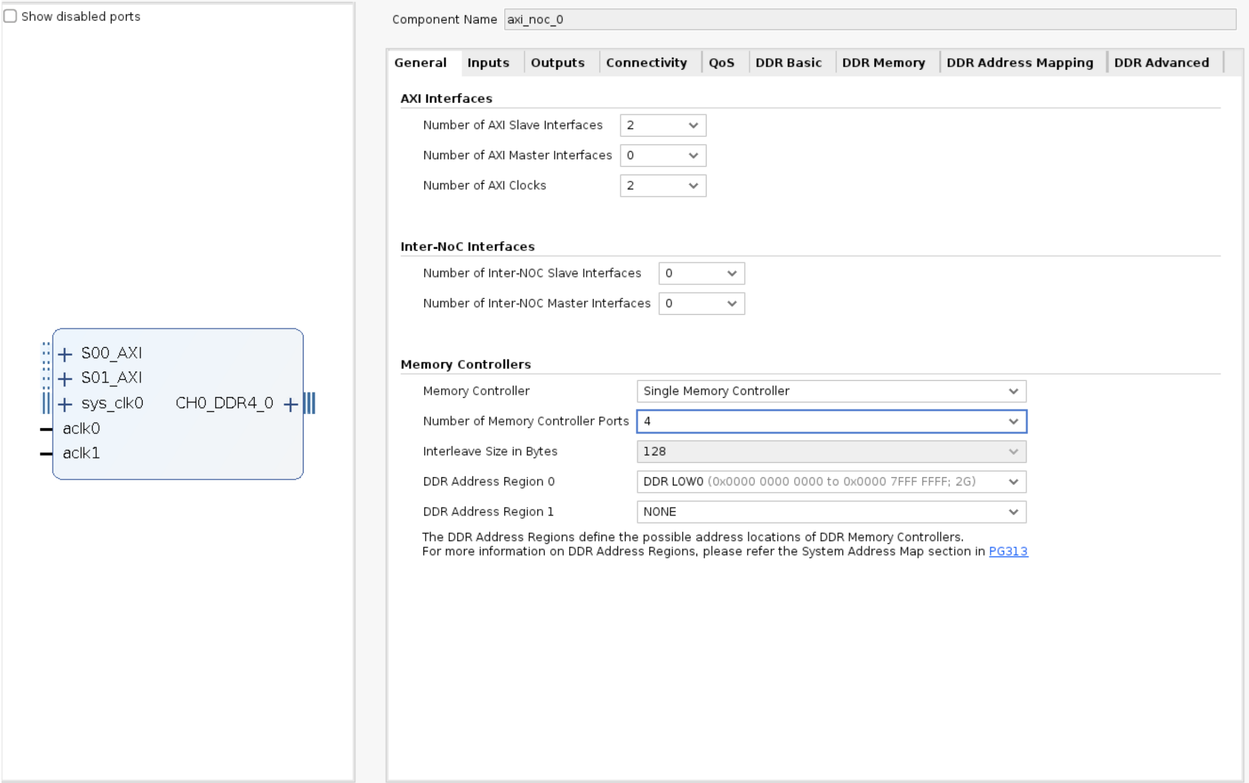
Next you will add and configure a Network on Chip (NoC) IP core for the DDR connection.

1. Right-click on the block design canvas and from the context menu select **Add IP**.
2. The IP catalog pops up. In the Search field type `AXI NoC` to filter a list of IP.
3. From the filtered list, double-click the **AXI NoC** IP core to instantiate the IP on the block design canvas.

   Customize the IP as follows:

4. In the General tab, set the following options:
   - Number of AXI Slave Interfaces: **2**.
   - Number of AXI Master Interfaces: **0**.
   - Number of AXI Clocks: **2**.

     The number of AXI clocks is set to two because there are two clocks needed for the AXI Slave input, and none needed for AXI Master output.
   - Memory Controller: **Single Memory Controller**.
   - Number of Memory Controller Port: **4**.
   - All others options use the default settings.



□ Show disabled ports

Component Name axi_noc_0

General | Inputs | Outputs | Connectivity | QoS | DDR Basic | DDR Memory | DDR Address Mapping | DDR Advanced

AXI Interfaces

Number of AXI Slave Interfaces	2
Number of AXI Master Interfaces	0
Number of AXI Clocks	2

Inter-NoC Interfaces

Number of Inter-NOC Slave Interfaces	0
Number of Inter-NOC Master Interfaces	0

Memory Controllers

Memory Controller	Single Memory Controller
Number of Memory Controller Ports	4
Interleave Size in Bytes	128
DDR Address Region 0	DDR LOW0 (0x0000 0000 0000 to 0x0000 7FFF FFFF; 2G)
DDR Address Region 1	NONE

The DDR Address Regions define the possible address locations of DDR Memory Controllers.
For more information on DDR Address Regions, please refer the System Address Map section in PG313

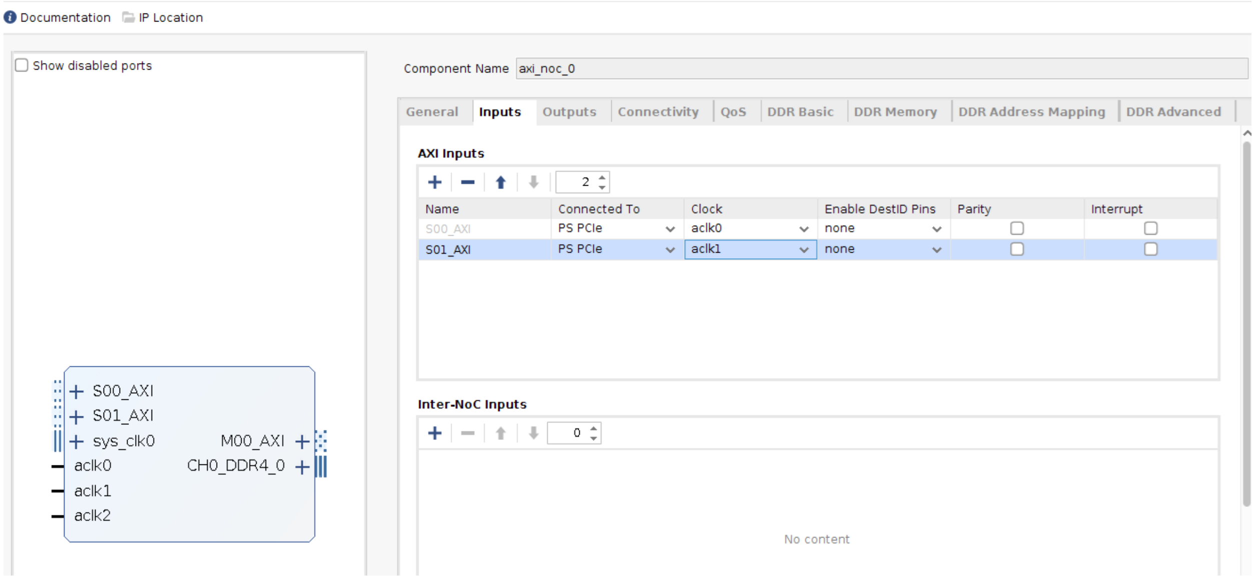
5. In the Inputs tab, set the following options.

   First row (for S00_AXI):

   - Connected To: **PS PCIe**.
   - Clock: **aclk0** (input clock).
   - All other options use default settings.

   Second row (for S01_AXI):

   - Connected To: **PS PCIe**.
   - Clock: **aclk1** (input clock).
   - All other options use default settings.



Component Name axi_noc_0

AXI Inputs

2

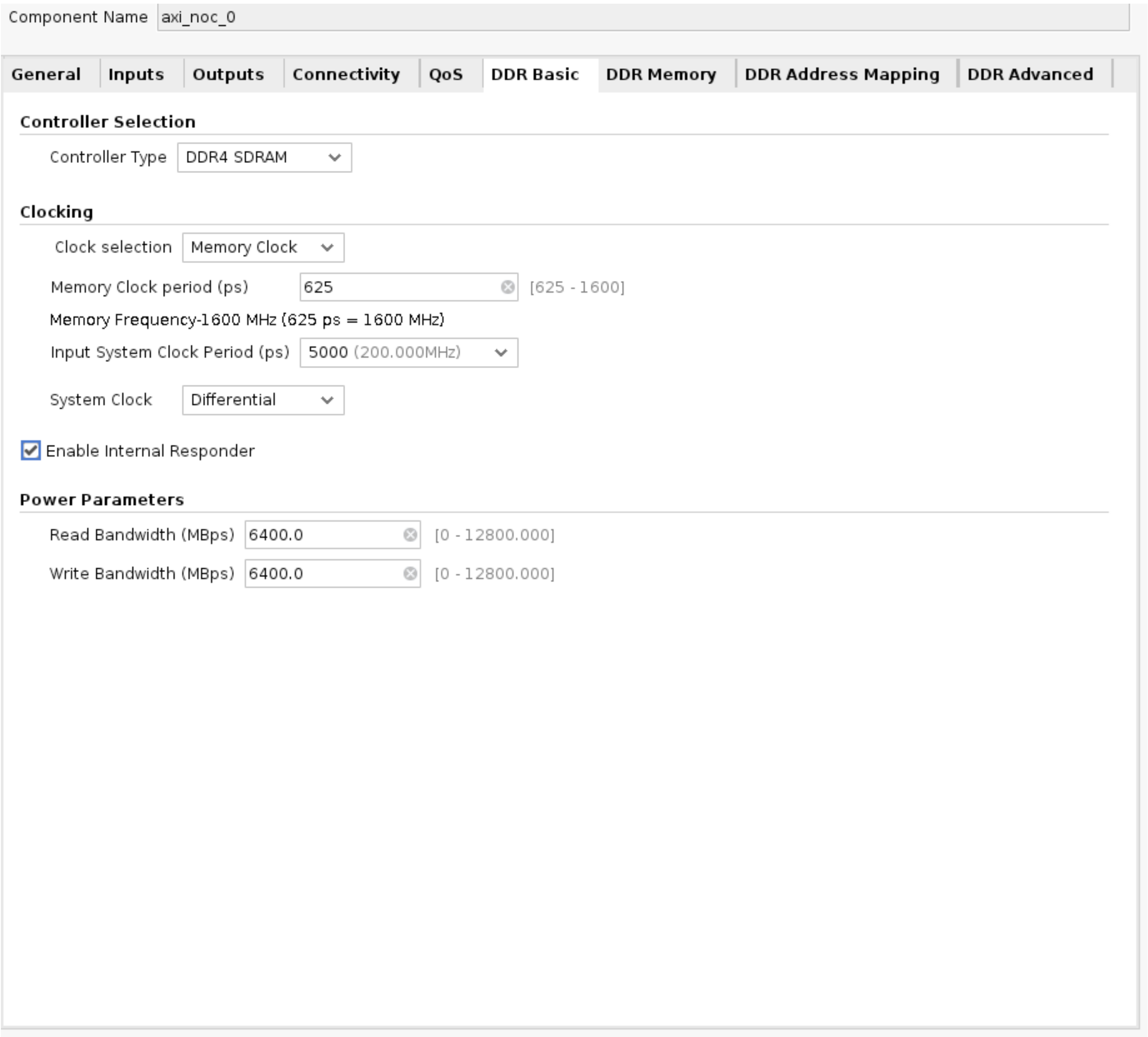
Name	Connected To	Clock	Enable DestID Pins	Parity	Interrupt
S00_AXI	PS PCIe	aclk0	none	□	□
S01_AXI	PS PCIe	aclk1	none	□	□

Inter-NoC Inputs

0

No content

6. In the Connectivity tab, set the NoC connectivity as follows:
   - For S00_AXI, select the **MC Port 0** checkbox.
   - For S01_AXI, select the **MC Port 0** checkbox.
   - All others options use the default settings.
7. In the DDR Basic tab, set the following options:
   - Input System clock period (ps): **5000 (200.000 MHz)**.
   - Select the **Enable Internal Responder** checkbox.
   - All others options use the default settings.



Component Name axi_noc_0

Controller Selection

Controller Type DDR4 SDRAM

Clocking

Clock selection Memory Clock

Memory Clock period (ps) 625 [625 - 1600]

Memory Frequency-1600 MHz (625 ps = 1600 MHz)

Input System Clock Period (ps) 5000 (200.000MHz)

System Clock Differential

☑ Enable Internal Responder

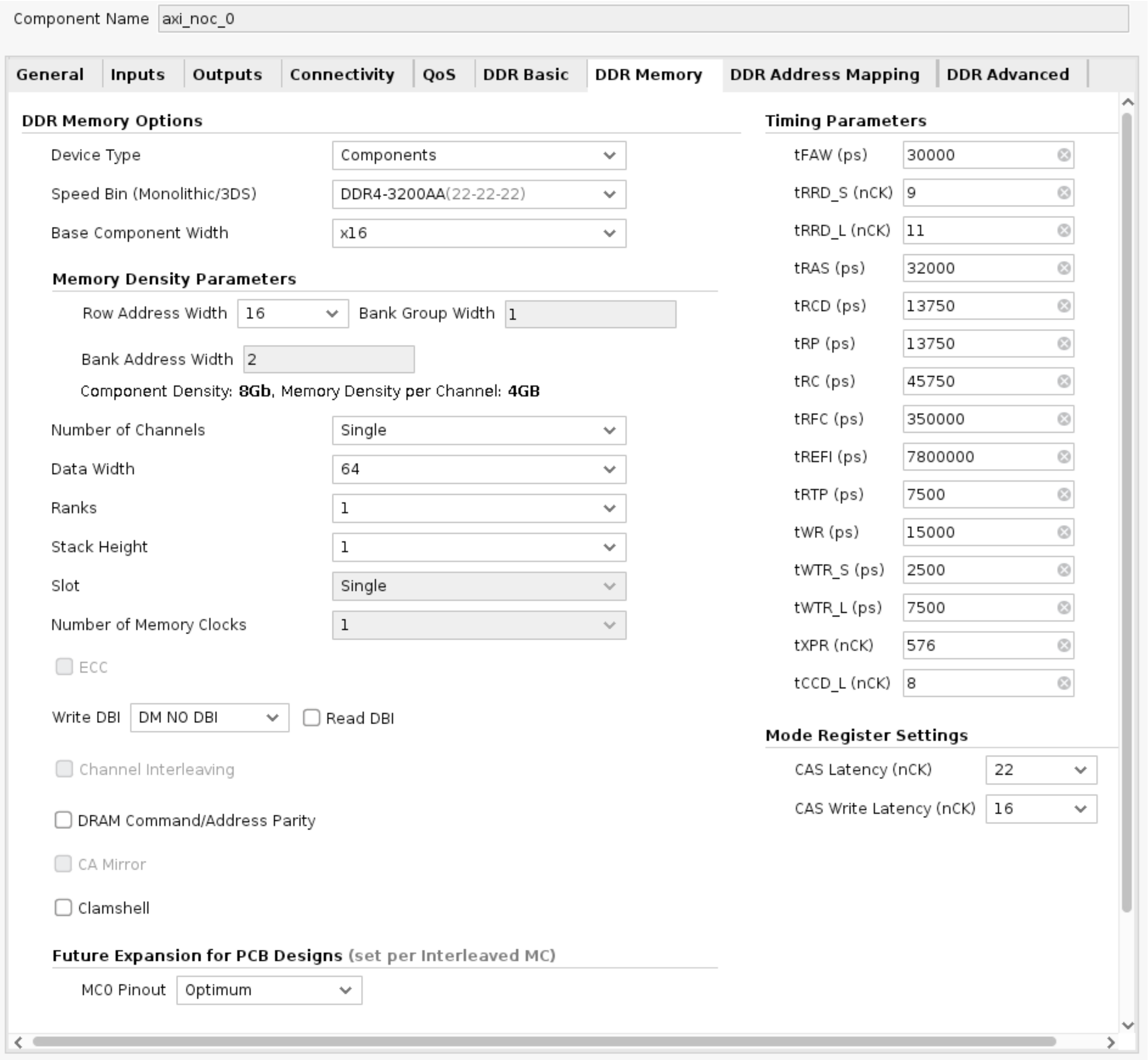
Power Parameters

Read Bandwidth (MBps) 6400.0 [0 - 12800.000]

Write Bandwidth (MBps) 6400.0 [0 - 12800.000]

***Note*:** This is a sample configuration. Your DDR configuration and frequencies should be based on your design requirements.

8. In the DDR Memory tab, set the following options:
    - Memory Device Type: **Components**.
    - Memory Speed Grade: **DDR4-3200AA(22-22-22)**.
    - Base Component Width: **x16**.
    - All others options use the default settings.



Component Name axi_noc_0

DDR Memory Options

Device Type: Components
Speed Bin (Monolithic/3DS): DDR4-3200AA(22-22-22)
Base Component Width: x16

Memory Density Parameters

Row Address Width: 16
Bank Group Width: 1
Bank Address Width: 2
Component Density: **8Gb**, Memory Density per Channel: **4GB**

Number of Channels: Single
Data Width: 64
Ranks: 1
Stack Height: 1
Slot: Single
Number of Memory Clocks: 1

☐ ECC

Write DBI: DM NO DBI ☐ Read DBI

☐ Channel Interleaving

☐ DRAM Command/Address Parity

☐ CA Mirror

☐ Clamshell

Future Expansion for PCB Designs (set per Interleaved MC)

MC0 Pinout: Optimum

Timing Parameters

tFAW (ps)	30000
tRRD_S (nCK)	9
tRRD_L (nCK)	11
tRAS (ps)	32000
tRCD (ps)	13750
tRP (ps)	13750
tRC (ps)	45750
tRFC (ps)	350000
tREFI (ps)	7800000
tRTP (ps)	7500
tWR (ps)	15000
tWTR_S (ps)	2500
tWTR_L (ps)	7500
tXPR (nCK)	576
tCCD_L (nCK)	8

Mode Register Settings

CAS Latency (nCK): 22
CAS Write Latency (nCK): 16

9. Click **OK** to generate a NoC IP with DDR.

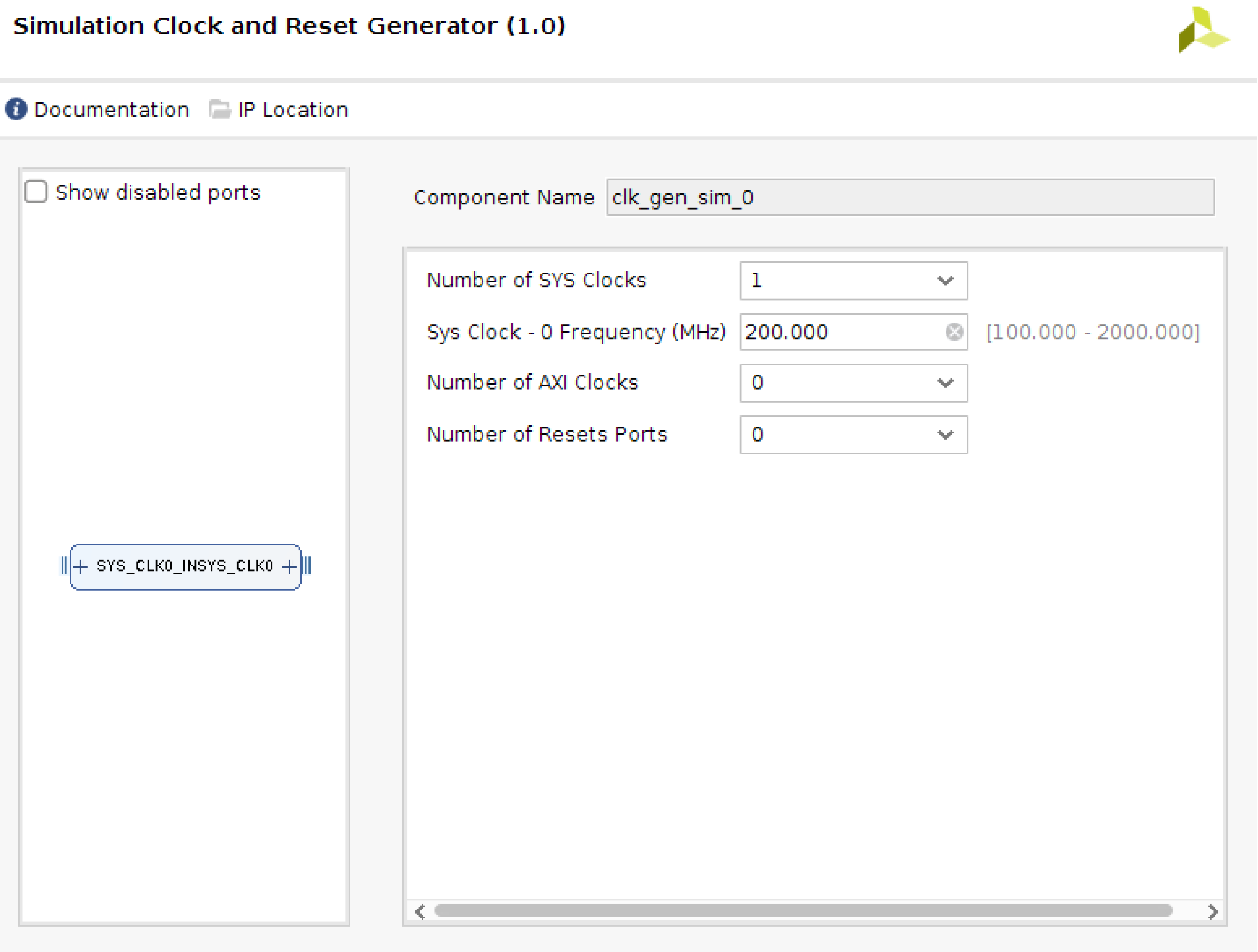
Generate the Clock For the NoC IP

Next, generate a clock source for the NoC module. To do this, you will configure and generate the Simulation Clock and Reset Generator IP core.

1. Click **Add IP**, and search for `Simulation Clock and Reset Generator`.
2. From the filtered list, double-click the **Simulation Clock and Reset Generator** IP core to instantiate the IP on the block design canvas.

   Configure the core as follows:
3. For Number of SYS clocks, select **1**.

4. For Sys Clock 0 Frequency (MHz), enter **200**.
5. For Number of AXI Clocks, select **0**.
6. For Number of Resets Ports, select **0**.
7. Click **OK** to generate IP.



Simulation Clock and Reset Generator (1.0)

Component Name: clk_gen_sim_0

Number of SYS Clocks: 1

Sys Clock - 0 Frequency (MHz): 200.000 [100.000 - 2000.000]

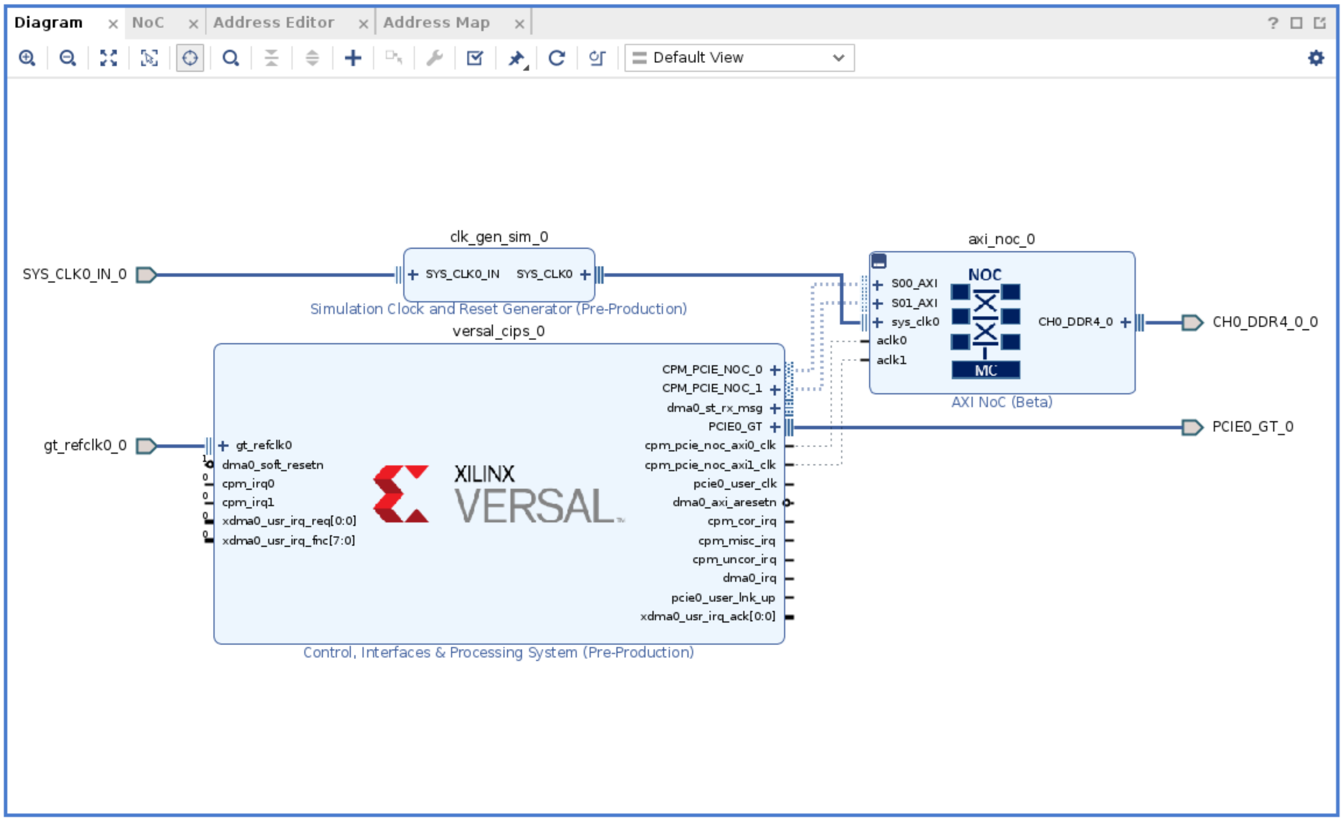
Number of AXI Clocks: 0

Number of Resets Ports: 0

IP Connections

Next, add signal connections between the IP in the Vivado IP integrator.

1. Make the connections between the IP cores as shown in following figure.
2. Set `GT_REFCLK_D`, `GT_PCIEA0_RX`, `GT_PCIEA0_TX`, `SYS_CLK0_IN` and `CH0_DDR4_0` as primary ports. To do so:
   a. Select pins `GT_REFCLK_D`, `GT_PCIEA0_RX`and `GT_PCIEA0_TX` of versal_cips_0, `SYS_CLK0_IN` of clk_gen_sim_0, and `CH0_DDR4_0` of axi_noc_0 by pressing **Ctrl+click**.
   b. Click the **Make External (Ctrl + T)** icon in the toolbar at the top of the canvas.

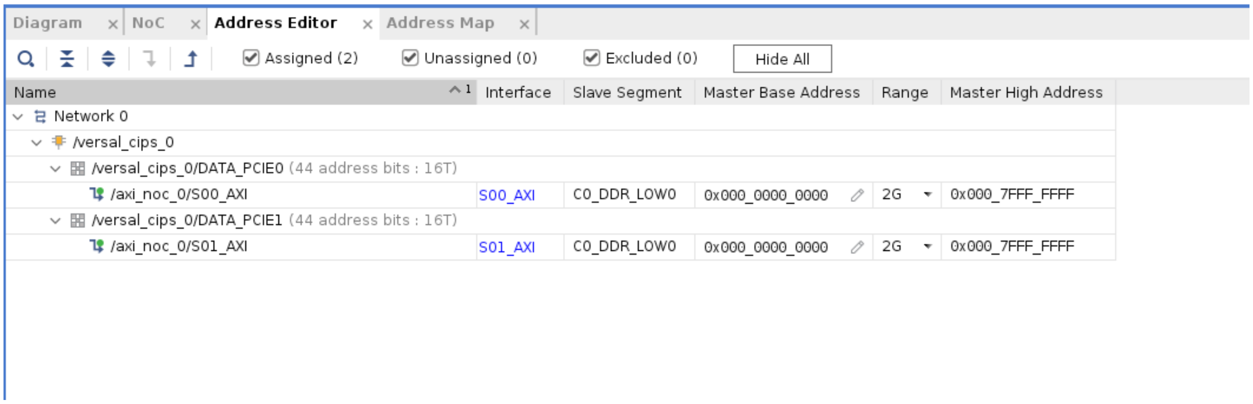


Address Settings

Next, set the necessary address settings for the NoC IP.

1. Open the **Address Editor** tab as shown in the following figure. Expand the tree by clicking the down-arrow on **versal_cips_0**. Expand **DATA_PCIE0**, and expand **DATA_PCIE1**.
2. For S00_AXI, right-click in the Master Base Address cell, and select **Assign** from the context menu.
3. And similarly for S01_AXI, right-click in the Master Base Address cell, and select **Assign** from the context menu.

   Note that the address `0x00000` is assigned to the DDR.



Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/versal_cips_0					
/versal_cips_0/DATA_PCIE0 (44 address bits : 16T)					
/axi_noc_0/S00_AXI	S00_AXI	C0_DDR_LOW0	0x000_0000_0000	2G	0x000_7FFF_FFFF
/versal_cips_0/DATA_PCIE1 (44 address bits : 16T)					
/axi_noc_0/S01_AXI	S01_AXI	C0_DDR_LOW0	0x000_0000_0000	2G	0x000_7FFF_FFFF

Validate the Block Design

1. To validate the design, open the Diagram tab, and click the **Validate Design** icon ☑, or right-click anywhere in the canvas and, from the context menu, select **Validate Design**.

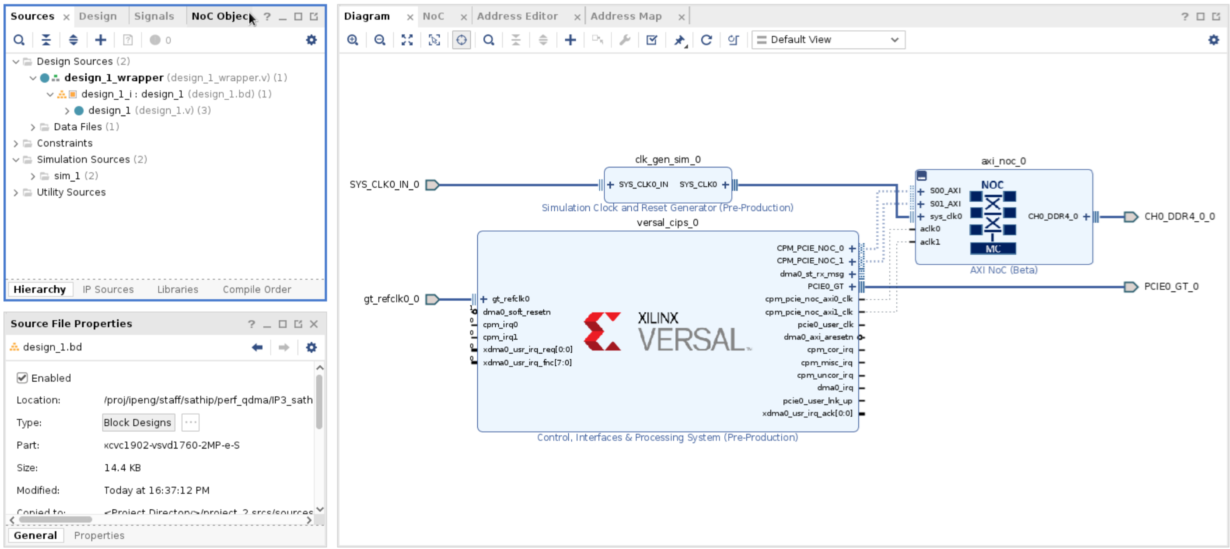
   After validation, confirmation of the successful validation displays in a pop up window.

Create a Design Wrapper

After validation, create a design wrapper. A design wrapper file enables you to add any needed logic. For this lab, additional logic is not needed.

1. In the Vivado IDE Sources window, right-click on **design_1 (design_1.bd)**.
2. From the context menu, select **Create HDL Wrapper** to generate a wrapper file.

   A `design_1_wrapper` file is added to the Sources window as shown in the following figure.



- Design Sources (2)
  - design_1_wrapper (design_1_wrapper.v) (1)
    - design_1_i : design_1 (design_1.bd) (1)
      - design_1 (design_1.v) (3)
  - Data Files (1)
- Constraints
- Simulation Sources (2)
  - sim_1 (2)
- Utility Sources

design_1.bd

☑ Enabled

Location:	/proj/ipeng/staff/sathip/perf_qdma/IP3_sath
Type:	Block Designs
Part:	xcvc1902-vsvd1760-2MP-e-S
Size:	14.4 KB
Modified:	Today at 16:37:12 PM
Copied to:	<Project Directory>/project_2.srcs/sources

Synthesize and Implement the Design

After the wrapper file is created, you will add the `top_impl.xdc` constraints file, which is provided with this guide, to your design in Vivado. The constraints file constrains DDR pin placement. Then, you can run synthesis and implementation, which generates a PDI (Programmable Device Image) file.

Note: To locate the `top_impl.xdc` constraints file, you need to download the `pg347-ea2-labs.zip` file and extract its contents. For details, see Tutorial Design File.

1. In the Flow Navigator window, click **Add Sources**, click **Add or create Constraints**, and add the `top_impl.xdc` file.
2. In the Flow Navigator, click **Synthesis and Implementation** to implement the project design, and generate a PDI file.

Note: The Tandem critical warning `HD.TANDEM` can be ignored this release.

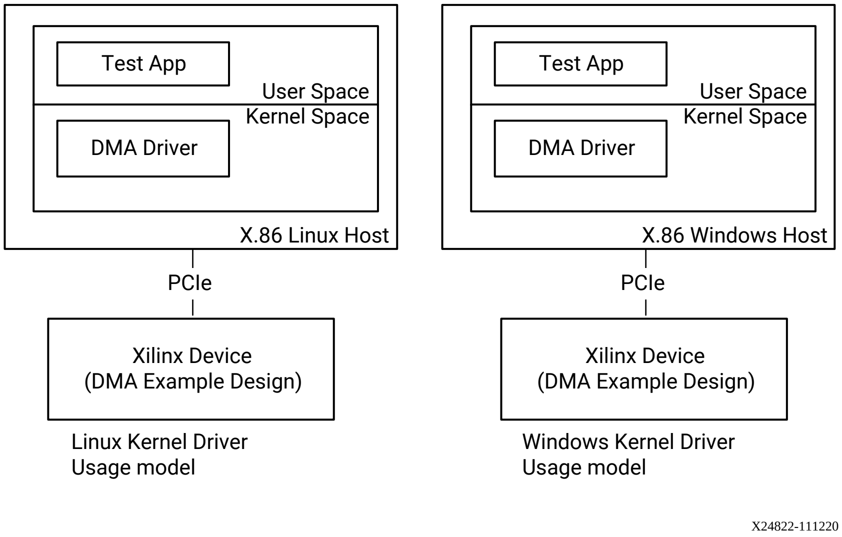
Chapter 14

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 55: **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 follow:

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

Chapter 15

Debugging

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

Documentation

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

Solution Centers

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

The Solution Center specific to the functional mode is listed below.

Xilinx Solution Center for PCI Express

Answer Records

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

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

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

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

Master Answer Record for the Core/Subsystem

AR 75396.

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.

Registers

A complete list of registers and attributes for the XDMA Subsystem is available in the *Versal ACAP Register Reference* (AM012). Reviewing the registers and attributes might be helpful for advanced debugging.

***Note*:** The attributes are set during IP customization in the Vivado IP catalog. After core customization, attributes are read-only.

Chapter 16

Upgrading

This appendix is not applicable for the first release of the functional mode.

Appendix A

GT Selection and Pin Planning

This appendix provides guidance on gigabit transceiver (GT) selection for Versal™ devices and some key recommendations that should be considered when selecting the GT locations. The GT locations for Versal devices can be customized through the IP customization wizard. This appendix provides guidance for CPM, PL PCIe and PHY IP based solutions. In this guide, the CPM PL PCIe related guidance is of primary importance, while the other related guidance might be relevant and is provided for informational purposes.

A GT Quad is comprised of four GT lanes. When selecting GT Quads for the PL PCIe-based solution CPM PCIE controller solution, Xilinx® recommends that you use the GT Quad most adjacent to the integrated block. While this is not required, it will improve place, route, and timing for the design.

- Link widths of x1, x2, and x4 require one bonded GT Quad and should not split lanes between two GT Quads.
- A link width of x8 requires two adjacent GT Quads that are bonded and are in the same SLR.
- A link width of x16 requires four adjacent GT Quads that are bonded and are in the same SLR.
- PL PCIe blocks should use GTs adjacent to the PCIe block where possible.
- CPM has a fixed connectivity to GTs based on the CPM configuration.

For GTs on the **left side of the device**, PCIe lane 0 is placed in the bottom-most GT of the bottom-most GT Quad. Subsequent lanes use the next available GTs moving vertically up the device as the lane number increments. This means that the highest PCIe lane number uses the top-most GT in the top-most GT Quad that is used for PCIe.

For GTs on the **right side of the device**, PCIe lane 0 is placed in the top-most GT of the top-most GT Quad. Subsequent lanes use the next available GTs moving vertically down the device as the lane number increments. This means that the highest PCIe lane number uses the bottom-most GT in the bottom-most GT Quad that is used for PCIe.

The PCIe reference clock uses GTREFCLK0 in the PCIe lane 0 GT Quad for x1, x2, x4, and x8 configurations. For x16 configurations the PCIe reference clock should use GTREFCLK0 on a GT Quad associated with lanes 8-11. This allows the clock to be forwarded to all 16 PCIe lanes.

The PCIe reset pins for CPM designs must connect to one of specified pins for each of the two PCIe controllers. The PCIe reset pin for PL PCIe and PHY IP designs can be connected to any compatible PL pin location, or the CPM PCIe reset pins when the corresponding CPM PCIe controller is not in use. This is summarized in the table below.

Table 98: **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.

CPM4 GT Selection

The CPM block within Versal devices has a fixed set of GTs that can be used for each of the two PCIe controllers. These GTs are shared between the two PCIe controllers and High Speed Debug Port (HSDP) as such x16 link widths are only supported when a single PCIe controller is in use and HSDP is disabled. When two CPM PCIe controllers or one PCIe controller and HSDP are enabled each link will be limited to a x8 link width. GT Quad allocation for CPM happens at GT Quad granularity and must include all GT Quads from the most adjacent to the CPM to the top-most GT Quad that is in use by the CPM. GT Quads that are used or between GT Quads that are used by the CPM (for either PCIe or HSDP) cannot be shared with PL resources even if GTs within the quad are not in use.

CPM in Single Controller Mode

When a single PCIe controller in the CPM is being used and HSDP is disabled, PCIe x1, x2, x4, x8, and x16 link widths are supported. PCIe lane0 is places at the bottom-most GT of the bottom-most GT Quad that is directly above the CPM. Subsequent lanes use the next available GTs moving vertically up the device as the lane number increments. This means the highest PCIe lane number uses the top-most GT in the top-most GT Quad that is used for PCIe. Because the GT locations and lane ordering for CPM is fixed it cannot be modified through IP customization.

As stated previously GT Quad allocation happens at GT Quad granularity and cannot share unused GT Quad resources with the PL. This means that CPM PCIe controller 0 configurations that use x1 or x2 link widths will not use all the GTs within the Quad and that these GTs cannot be used in the PL for additional GT connectivity. Unused GT Quads in this configuration can be used by the PL to implement PL GT based solutions.

When CPM PCIe controller 0 and High Speed Debug Port (HSDP) is enabled, a PCIe link width of x16 cannot be used and the CPM will use all three GT Quads that are directly above the CPM regardless of PCIe link width. In this configuration, these GT Quads are allocated to CPM and cannot be shared with PL resources. CPM PCIe lanes 0-7 will be unchanged in their GT selection and lane ordering. HSDP will use the bottom-most GT that is the third GT Quad away from CPM. This corresponds to the same location as PCIe lane 8 for a x16 link configuration. The fourth GT Quad in this configuration is not use by CPM and can be used to implement PL GT based solutions.

CPM in Dual Controller Mode

When the CPM is configured to use two PCIe controllers, High Speed Debug Port (HSDP) cannot be used because it shares GTs with the two PCIe controllers. Each PCIe controller can support x1, x2, x4 and x8 link widths in this configuration. This configuration will use at least the bottom three GT Quads closest to the CPM. These GT Quads cannot be used by PL resources. If CPM PCIe controller 1 is using a link width of x1, x2, or x4; then CPM uses three GT Quads. In this case the fourth GT Quad can be used by PL resources to implement GT based solutions. If CPM PCIe controller 1 is using a x8 link width, all four GT Quads will be used by the CPM and cannot be used by PL resources.

CPM PCIe controller 0 lane0 is placed at the bottom-most GT of the bottom-most GT Quad that is directly above the CPM. Subsequent lanes use the next available GTs moving vertically up the device as the lane number increments. CPM PCIe controller 0 lane7 connects to the top-most GT in the second GT Quad away from the CPM.

CPM PCIe controller 1 lane0 is places at the bottom-most GT of third GT Quad above the CPM. Subsequent lanes use the next available GTs moving vertically up the device as the lane number increments. CPM PCIe controller 1 lane7 connects to the top-most GT in the fourth GT Quad away from the CPM.

High Speed Debug Port (HSDP) Only Modes

When the CPM is configured to use the High Speed Debug Port (HSDP) without enabling either PCIe controller, the bottom-most GT in the bottom-most GT Quad closest to CPM should be used. This will allow the CPM to use only one GT Quad and allow the next three GT Quads to be used by PL resources.

HSDP can also be enabled for the bottom-most GT in the third GT Quad up from CPM. In this scenario CPM will use three GT Quads and only use one GT. The remaining unused GTs cannot be used or shared by PL resources. As result typically HSDP will not be used in this configuration.

CPM4 Additional Considerations

To facilitate migration from UltraScale™ or UltraScale+™ designs, boards may be designed to use either CPM4 or PL PCIe integrated blocks to implement PCIe solutions. When designing a board to use either CPM4 or the PL PCIe hardblock, the CPM4 pin selection and planning guidelines should be followed because they are more restrictive. By doing this a board can be designed that will work for either a CPM4 or PL PCIe implementation. To route the PCIe reset from the CPM4 to the PL for use with a PL PCIe implementation the following parameter must be set in Vivado prior to customizing the CIPS IP.

```
set_param pcw.enplpciereset 1
```

When this parameter is enabled the PCIe reset for each disabled CPM4 PCIe controller will be routed to the PL. The same CPM4 pin selection limitations will apply and the additional PCIe reset output pins will be exposed at the boundary of the CIPS IP. If the CPM4 PCIe controller is enabled, the PCIe reset will be used internal to the CPM4 and will not be routed to the PL for connectivity to PL PCIe controllers.

GT Locations

Assigning GT Locations

Unlike in UltraScale+ and previous devices where direct assignment of GTs are not possible in the user constraints, in Versal the GT locations assignment can be done in the user constraints, while changing GT locations in GT customization IP is not available. Below is an example of assigning GT locations in a user constraint file.

Note: The gt_quad instances should be assigned contiguously.

```
set_property LOC GTY_QUAD_X0Y6   [get_cells $gt_quads -filter NAME=~*/
gt_quad_3/*]
set_property LOC GTY_QUAD_X0Y5   [get_cells $gt_quads -filter NAME=~*/
gt_quad_2/*]
set_property LOC GTY_QUAD_X0Y4   [get_cells $gt_quads -filter NAME=~*/
gt_quad_1/*]
set_property LOC GTY_QUAD_X0Y3   [get_cells $gt_quads -filter NAME=~*/
gt_quad_0/*]
```

GT Quad Locations

The following table identifies the PCIe lane0 GT Quad(s) that can be used for each PCIe controller location. The Quad shown in bold is the most adjacent or suggested GT Quad for each PCIe lane0 location.

Table 99: **GT Locations**

Device	Package	PCIe Blocks	GT QUAD (X16)	GT QUAD (X8)	GT QUAD (X4 and Below)
XCVC1902	VIVA1596	CPM Controller 0	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**
		CPM Controller 1	N/A	**GTY_QUAD_X0Y5**	**GTY_QUAD_X0Y5**
		X0Y2	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3
		X0Y1	**GTY_QUAD_X0Y3**	GTY_QUAD_X0Y5 GTY_QUAD_X0Y4 **GTY_QUAD_X0Y3**	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3
		X1Y2	**GTY_QUAD_X1Y5**	**GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3	**GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3 GTY_QUAD_X1Y2
		X1Y0	**GTY_QUAD_X1Y5**	GTY_QUAD_X1Y5 GTY_QUAD_X1Y4 **GTY_QUAD_X1Y3**	GTY_QUAD_X1Y5 GTY_QUAD_X1Y4 GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2**
XCVC1902	VSVA2197	CPM Controller 0	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**
		CPM Controller 1	N/A	**GTY_QUAD_X0Y5**	**GTY_QUAD_X0Y5**
		X0Y2	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3
		X0Y1	**GTY_QUAD_X0Y3**	GTY_QUAD_X0Y5 GTY_QUAD_X0Y4 **GTY_QUAD_X0Y3**	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3
		X1Y2	**GTY_QUAD_X1Y6**	**GTY_QUAD_X1Y6** GTY_QUAD_X1Y5 GTY_QUAD_X1Y4	GTY_QUAD_X1Y6 **GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3
		X1Y0	**GTY_QUAD_X1Y3**	**GTY_QUAD_X1Y3** GTY_QUAD_X1Y2 GTY_QUAD_X1Y1	GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2** GTY_QUAD_X1Y1 GTY_QUAD_X1Y0

Table 99: **GT Locations** *(cont'd)*

Device	Package	PCIe Blocks	GT QUAD (X16)	GT QUAD (X8)	GT QUAD (X4 and Below)
XCVC1902	VSVD1760	CPM Controller 0	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**
		CPM Controller 1	N/A	**GTY_QUAD_X0Y5**	**GTY_QUAD_X0Y5**
		X0Y2	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3
		X0Y1	**GTY_QUAD_X0Y3**	GTY_QUAD_X0Y5 GTY_QUAD_X0Y4 **GTY_QUAD_X0Y3**	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3
		X1Y2	N/A	**GTY_QUAD_X1Y4**	**GTY_QUAD_X1Y4** GTY_QUAD_X1Y3
		X1Y0	N/A	**GTY_QUAD_X1Y4**	GTY_QUAD_X1Y4 **GTY_QUAD_X1Y3**
XCVM1802	VFVC1760	CPM Controller 0	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**
		CPM Controller 1	N/A	**GTY_QUAD_X0Y5**	**GTY_QUAD_X0Y5**
		X0Y2	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3
		X0Y1	**GTY_QUAD_X0Y3**	GTY_QUAD_X0Y5 GTY_QUAD_X0Y4 **GTY_QUAD_X0Y3**	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3
		X1Y2	**GTY_QUAD_X1Y6**	**GTY_QUAD_X1Y6** GTY_QUAD_X1Y5 GTY_QUAD_X1Y4	GTY_QUAD_X1Y6 **GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3
		X1Y0	**GTY_QUAD_X1Y3**	**GTY_QUAD_X1Y3** GTY_QUAD_X1Y2 GTY_QUAD_X1Y1	GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2** GTY_QUAD_X1Y1 GTY_QUAD_X1Y0

Table 99: **GT Locations** *(cont'd)*

Device	Package	PCIe Blocks	GT QUAD (X16)	GT QUAD (X8)	GT QUAD (X4 and Below)
XCVM1802	VSVA2197	CPM Controller 0	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**
		CPM Controller 1	N/A	**GTY_QUAD_X0Y5**	**GTY_QUAD_X0Y5**
		X0Y2	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3
		X0Y1	**GTY_QUAD_X0Y3**	GTY_QUAD_X0Y5 GTY_QUAD_X0Y4 **GTY_QUAD_X0Y3**	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3
		X1Y2	**GTY_QUAD_X1Y6**	**GTY_QUAD_X1Y6** GTY_QUAD_X1Y5 GTY_QUAD_X1Y4	GTY_QUAD_X1Y6 **GTY_QUAD_X1Y5** GTY_QUAD_X1Y4 GTY_QUAD_X1Y3
		X1Y0	**GTY_QUAD_X1Y3**	**GTY_QUAD_X1Y3** GTY_QUAD_X1Y2 GTY_QUAD_X1Y1	GTY_QUAD_X1Y3 **GTY_QUAD_X1Y2** GTY_QUAD_X1Y1 GTY_QUAD_X1Y0
XCVM1802	VSVD1760	CPM Controller 0	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y3**
		CPM Controller 1	N/A	**GTY_QUAD_X0Y5**	**GTY_QUAD_X0Y5**
		X0Y2	**GTY_QUAD_X0Y3**	**GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3	GTY_QUAD_X0Y6 **GTY_QUAD_X0Y5** GTY_QUAD_X0Y4 GTY_QUAD_X0Y3
		X0Y1	**GTY_QUAD_X0Y3**	GTY_QUAD_X0Y5 GTY_QUAD_X0Y4 **GTY_QUAD_X0Y3**	GTY_QUAD_X0Y6 GTY_QUAD_X0Y5 **GTY_QUAD_X0Y4** GTY_QUAD_X0Y3
		X1Y2	N/A	**GTY_QUAD_X1Y4**	**GTY_QUAD_X1Y4** GTY_QUAD_X1Y3
		X1Y0	N/A	**GTY_QUAD_X1Y4**	GTY_QUAD_X1Y4 **GTY_QUAD_X1Y3**

Appendix B

Migrating

For information about migrating UltraScale+ device QDMA, AXI Bridge, and XDMA IP designs to the Versal ACAP CPM DMA and Bridge Mode for PCI Express, see AR 75396.

Appendix C

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. *Control, Interface and Processing System LogiCORE IP Product Guide* (PG352)
2. *Versal ACAP DMA and Bridge Subsystem for PCI Express Product Guide* (PG344)
3. *Versal ACAP CPM Mode for PCI Express Product Guide* (PG346)
4. *Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide* (PG343)
5. *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313)
6. *SmartConnect LogiCORE IP Product Guide* (PG247)
7. *QDMA Subsystem for PCI Express Product Guide* (PG302)
8. *DMA/Bridge Subsystem for PCI Express Product Guide* (PG195)
9. *AXI Bridge for PCI Express Gen3 Subsystem Product Guide* (PG194)
10. *Versal ACAP Register Reference* (AM012)
11. *PCI-SIG Specifications* (https://www.pcisig.com/specifications)
12. *AMBA AXI4-Stream Protocol Specification* (ARM IHI 0051A)
13. *Vivado Design Suite User Guide: Designing with IP* (UG896)
14. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
15. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
16. *Vivado Design Suite User Guide: Getting Started* (UG910)
17. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
05/04/2021 Version 2.1	
Limitations	Added known issues for the release.
QDMA Functional Mode and XDMA Functional Mode	Added clarifying details regarding AXI4-Stream interface data rate support.

Section	Revision Summary
12/04/2020 Version 2.1	
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