UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

EXTENSION AND IMPROVEMENT OF A PCIE-BASED FPGA ENVIRONMENT FOR TESTING HPC ARCHITECTURES

ANDREA QUEROL DE PORRAS

Thesis supervisor: FILIPPO MANTOVANI (Barcelona Supercomputer Center)

Tutor: XAVIER MARTORELL BOFILL (Department of Computer Architecture)

Degree: Master Degree in Innovation and Research in Informatics (High Performance Computing)

Thesis report

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Abstract

The European Processor Initiative (EPI) is a European project that performs research to advance High-Performance Computing (HPC) through the development of European technology. EPI aims at the development of a general-purpose processor and a RISC-V-based accelerator. Barcelona Supercomputing Center (BSC) is involved in the development of a Vector Processor Unit to be connected to a RISC-V core, called Vector tile, that is part of the EPI accelerator. While the actual hardware is being produced by the silicon foundry, the project foresees the implementation of a Vector tile within an Field Programmable Gate Array (FPGA) that serves as a hardware prototype for software development. The set of hardware and software tools necessary for making the Vector tile operational as an HPC compute node is called Software Development Vehicles (SDV). The SDV functionalities rely on a Xilinx FPGA connected to a host-PC via a Peripheral Component Interconnect Express (PCIe) link. This thesis aims to study, evaluate and upgrade the current PCIe subsystem working on the EPI SDV. The main technical contribution of this work is the improvement of the PCIe link between the host-PC and the FPGAs housing the Vector tile making use of the latest Xilinx Intellectual Property (IP) and the corresponding software. This work allowed both to improve the performance of the PCIe link and to expand the portability of the SDV environment to support one more FPGA board.

Contents

Chapter 1

Introduction

High-Performance Computing (HPC) refers to the use of powerful computers and advanced algorithms to solve complex computational problems at a significantly higher speed and performance than conventional computing systems. To achieve this higher speed, HPC systems leverage accelerators (at node level) and parallelism (at system level).

RISC-V is an open-source and free Instruction Set Architecture (ISA) that offers simplicity, modularity, and scalability, enabling efficient and customizable processor designs. An open and free ISA allows having a de-facto standard interface between software and hardware, leaving hardware designers the freedom to explore, and take profit from, different implementation points.

The European Processor Initiative (EPI) is a European research project focused on advancing HPC by developing European technology, with a specific focus on creating a general-purpose processor and a RISC-V-based accelerator. Within this initiative, BSC is actively involved in the development of a Vector Processor Unit (VPU) which is designed to be connected to a RISC-V core as part of the VEC tile of the EPI accelerator.

An accelerator is a specialized hardware or co-processor (device) designed to offload specific tasks or computations from the main processor (host) to improve performance. Accelerators often have their own ISAs or rely on specialized libraries. They can take various forms like GPUs, Field Programmable Gate Arrays (FPGAs), or Application-Specific Integrated Circuitss (ASICs), and HPC applications can take advantage of them to achieve more computing power, improving the application's performance by accelerating the application partly or completely. The operation of GPUs is based on workload offloading from the host to the device, but not all accelerators follow this paradigm. In the case of the EPI accelerator, named European Processor Accelerator (EPAC), it boots Linux and can operate as a self-hosted stand-alone system.

Currently, the VEC tile of the EPAC accelerator is being tested on FPGAs. An FPGA is a board with reprogrammable logic that can be used, for example, to accelerate algorithms with specialized logic or to emulate and test the Register-Transfer Level (RTL) code of processors or ASICs before sending it to the factory.

When using an FPGA as an accelerator and/or a test platform for RTL, a communication protocol is necessary to enable communication between the host system and the

FPGA (host-device model). Typically, the Peripheral Component Interconnect Express (PCIe) protocol is used, which offers high bandwidth and low latency. For achieving a correct host-device communication, it is necessary for the host to provide software support through the device driver, and for the device (FPGA) to implement the necessary hardware for the PCIe protocol.

The FPGAs used in EPI are Xilinx and offer 2 PCIe subsystems, two implementations to manage host-device communication through PCIe. The ultimate objective of this thesis is to contribute to the EPI project by replacing the current PCIe subsystem with the latest one offered by Xilinx, which enables new avenues of development and improvement in the EPAC design on FPGAs.

1.1 Master Thesis outline

The document is structured as follows:

- Chapter 2 contextualizes the motivation behind the thesis and sets out the thesis research questions.
- Chapter 3 offers an explanation of the technical aspects needed for the development.
- Chapter 4 contains a description of PCIe and DMA, and on top of that, the explanation of the Xilinx subsystems for PCIe implementing DMA studied.
- Chapter 5 illustrates the current status of the research and industry.
- Chapter 6 explains the technical development of this thesis, altogether with its evaluation.
- Chapter 7 exposes the conclusions, answering to the research questions, and future work of this master thesis.

Chapter 2

Motivation and objectives

2.1 EPI

The European Processor Initiative (EPI) is a European project that aims to create and institute a European computing infrastructure, specifically targeting HPC. This infrastructure is based on a General Purpose Processor (GPP) and an accelerator, which is called EPAC. This project is being carried out by several partners, which are companies and research centers [1].

2.1.1 EPAC

EPAC is a collection of different accelerators all based on the RISC-V architecture. Some of the EPAC's accelerators are EPAC-VEC, EPAC-VRP, and EPAC-STX, and each one targets a specific purpose. Figure 2.1 illustrates its structure. EPAC-VEC is a RISC-V CPU connected to a RISC-V VPU with the capacity to handle vector registers of up to 256 double-precision elements per instruction [2].

Barcelona Supercomputing Center (BSC) takes part in the development of the EPAC-VEC accelerator. Some of BSC's contributions are the development of the VPU that follows the V (Vector) extension of the RISC-V ISA, of the LLVM compiler for RISC-V with support to the vector extension, and of the Software Development Vehicles (SDV) infrastructure.

The work performed in this thesis is encapsulated in the EPAC-VEC accelerator, in specific, in the SDV project.

A part of the SDV infrastructure consists of nodes with FPGAs that we use to test the EPAC-VEC accelerator. More technical details about SDV are given in Section 3.2. The FPGA board used in those nodes is the Xilinx VCU128 [3]. There are four FPGA nodes and they are used by the EPAC-VEC and SDV developers, as well as an increasing number of external researchers interested in testing EPAC-VEC on our setup. Due to the increasing number of researchers interested in testing the RISC-V vector architecture implemented in EPAC-VEC, we expect availability issues both for researchers and SDV developers.

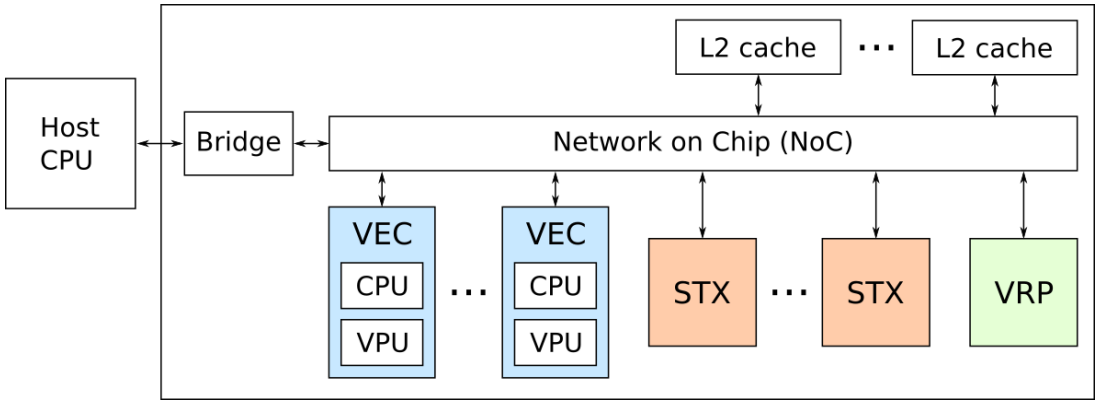


Figure 2.1: EPAC basic scheme with some of its accelerators and interconnections.

2.2 MEEP cluster

MareNostrum Experimental Exascale Platform (MEEP) is an FPGA-based cluster, whose objective is to facilitate the hardware and software development of future European technology for exascale systems [4]. It is formed by 12 nodes with 8 FPGAs per node, adding up to 96 FPGAs in total. The FPGAs used are Xilinx Alveo U55C[1]. It will be available by the end of June 2023.

2.3 Motivation

The MEEP cluster is suitable as a complement to the SDV platform, because it would allow increasing the number of SDV nodes and solve the availability issues. However, the current FPGA design used in SDV needs a change so it can work in MEEP's nodes.

The current FPGA design utilizes a specific PCIe subsystem that enables the usage of the PCIe interface. The PCIe subsystem employed is Xilinx DMA (XDMA), one of the two that Xilinx offers. The other Xilinx PCIe subsystem is Queue DMA (QDMA).

The MEEP cluster is a multiuser per node system, therefore virtualization is necessary to isolate users and their FPGAs between them. This requires using QDMA in the FPGAs because it supports Single Root I/O Virtualization (SR-IOV), which is a PCIe feature that allows virtualization and that is detailed in Section 4.1.2.

To be able to use any PCIe subsystem two pieces are required: the matching support in the FPGA design and the corresponding driver in the host machine. The QDMA FPGA part can be fully managed by the SDV team, whereas the driver needs to be agreed with the administrators of the MEEP machines in order to support the agreed Direct Memory Addressing (DMA) implementation. Hence, the available driver in the software stack will be the QDMA and the SDV design must be adapted to the QDMA requirement.

[1] Alveo U55C High Performance Compute Card https://www.xilinx.com/products/boards-and-kits/alveo/u55c.html

2.4 Research questions

For the completion of this Master's Thesis work, we considered obtaining an answer to the following research questions:

1. Does QDMA improve the performance over XDMA?
2. Which impact has QDMA over the design compared with XDMA? And over the software tools?
3. Is replacing XDMA with QDMA worth for the project?

Chapter 3

Technical background

This chapter exposes the technical information necessary for this thesis. It is explained what is an FPGA, the SDV infrastructure, the FPGA used in this thesis, and finally the software environment used.

3.1 FPGA

An FPGA is a semiconductor device formed by a pool of configurable Look-Up Tables (LUTs), registers, Digital Signal Processor (DSP) units, fast-memories, and Input/Outputs (IOs), interacting through a programmable interconnect fabric. The usage of that components is typically implemented with Hardware Description Languages (HDLs), such as VHDL or Verilog, which define the RTL. This device can be reprogrammed as many times as desired with different application or functionality requirements after manufacture [5]. Reconfigurability makes FPGAs more flexible than ASICs at the price of a lower transistor density. In addition, an FPGA offers different resources and reconfigurable logic such as DSPs, hard Intellectual Properties (IPs), etc., which will be explained in the current section.

FPGAs are often mounted on custom boards exposing several interfaces. This way the same board can serve as a "development board" or "development kit" for different purposes. Figure 3.1 shows a picture of the VCU128 development kit used for the work of this thesis. More details about this board will be explained in Section 3.3.

Figure 3.1: FPGA board, Virtex UltraScale+ HBM VCU128 FPGA Evaluation Kit. Source: [3].

FPGA's resources can be reconnected as desired, letting them to be changed as often as needed. That re-programmability capacity is FPGA's outstanding feature, as it allows that a single FPGA can be used in many ways, for different purposes.

On the one hand, an FPGA can be used as an accelerator. Its main advantage is the possibility of having as accelerator design as scientific applications, and being able to change those designs whenever a different application is being executed. On the other hand, an FPGA may be used to verify RTL code from a processor under development. That re-programmability feature allows testing each RTL modification or addition.

In the context of SDV, FPGAs are a key component in the RTL development, and therefore for the intermediate test chips and final chip. They allow verifying and testing the RTL that is being implemented in a faster way than in simulation. In other words, the frequency at which the FPGA can run is higher than the one at which a simulator can work. In the current SDV testing flow, the first verification step is done in simulation, and then more intensive testing is performed in FPGAs. Moreover, FPGAs permit testing of the software stack that is being developed in the project.

The remainder of this section introduces the main components, design flows, and concepts related to FPGA.

3.1.1 Bitstream

A bitstream is a binary file that contains all the configuration information necessary to program the FPGA. To be able to use an FPGA, the bitstream has to be loaded into the FPGA board to configure its logic and interconnects, enabling it to perform a specific task or function. Loading a bitstream file to an FPGA is denominated programming the FPGA.

Bitstream generation is not a trivial process. It is based on a complex process to synthesize the RTL code written in HDL into FPGA resources and interconnections. More details about this process are explained in section 3.4.3.

Once the bitstream is programmed onto the FPGA, the board device operates according to the logic and interconnects defined in the bitstream, performing the specific task or function for which it was designed.

3.1.2 Resources

An FPGA contains a certain number of different resources, which are used to synthesize the RTL code. The following components are tight to Xilinx FPGAs because they are the ones used in the context of the thesis. The logical ones relevant for this master thesis are:

- **Look-Up Table (LUT)**
  It is the basic, fundamental, building block of an FPGA. A LUT is a small, configurable memory unit that is capable of implementing arbitrary combinational logic functions [6]. It is a truth table that describes the output of the logic function for each possible input combination. A LUT has N inputs, which represent its size, typically for Xilinx FPGAs $N = 6$. LUTs can be used either as a function computing engine or as a data storage element. Figure 3.2a shows a LUT structure representation.

- **Registers or Flip-Flop (FF)**
  It is a type of memory element to store and synchronize digital signals [7]. The basic structure of a Flip-Flop (FF) includes: data input (D), clock input (CLK), clock enable, reset, and data output (Q). When the clock signal transitions from a low state to a high state (known as the rising edge), the data input is transferred to the output. The output then retains this value until the next rising edge of the clock signal, when the process repeats. The purpose of the clock enable pin is to allow the FF to hold a specific value for more than one clock cycle. New data inputs are only latched and passed to the data output port when both clock and clock enable are equal to one. Figure 3.2b contains a FF structure representation.

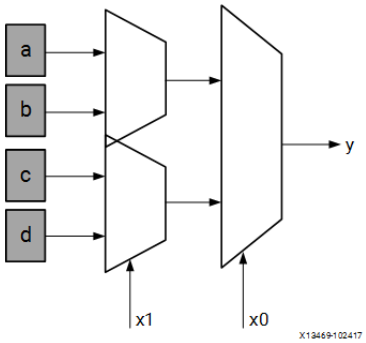
- **Digital Signal Processor (DSP)**
  It is a block dedicated to perform arithmetic operations, an Arithmetic Logic Unit (ALU) [8]. DSPs are chains of three different engines: *i)* an add/subtract unit connected to, *ii)* a multiplier attached to, *iii)* an add/subtract/accumulate block.

This implementation permits a DSP unit to fulfill functions of the form: $P = B \times (A+D)+C$ or $P+ = B \times (A+D)$. Figure 3.2c has a DSP structure representation.

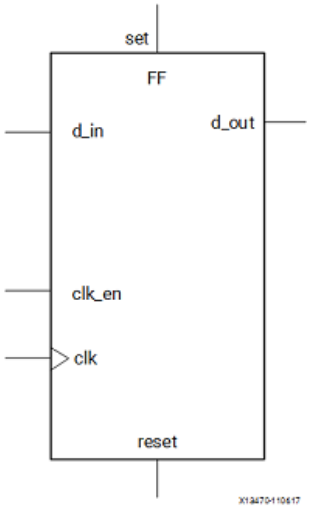
- **Block RAM (BRAM)**
  It is a dual-port RAM block that is designed to provide high-speed, low-latency, on-chip memory storage [9]. There are two Block RAM (BRAM) sizes: 18k and 36k bits. As BRAMs have two ports, they allow parallel, same-clock-cycle access to different locations. Figure 3.2d has a BRAM structure representation.

- **Ultra RAM (URAM)**
  It is a single-clock, synchronous, dual-port memory building block [9]. Its size is 288 Kb, which equals eight times the capacity of a BRAM. Each Ultra RAM (URAM) port can independently perform either one read or one write operation per clock cycle per port. URAM is available in Xilinx Ultrascale+ FPGA architecture. Figure 3.2e shows a URAM structure representation.

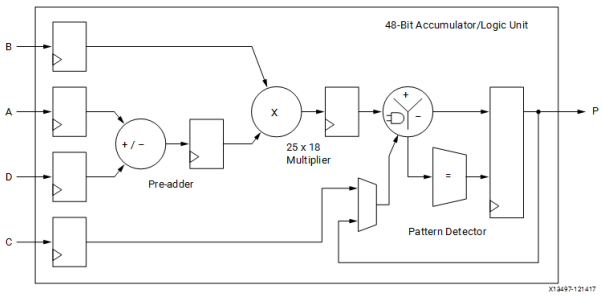
Those resources are part of the logic and routing logic of an FPGA. Figure 3.3 exhibits a representation of that structure. The FPGA provider makes available resources through building blocks that can be instantiated in the HDL, either manually or by the HDL compiler, e.g. Vivado, a program that will be explained in Section 3.4.1.



(a) LUT structure. Source: [6].

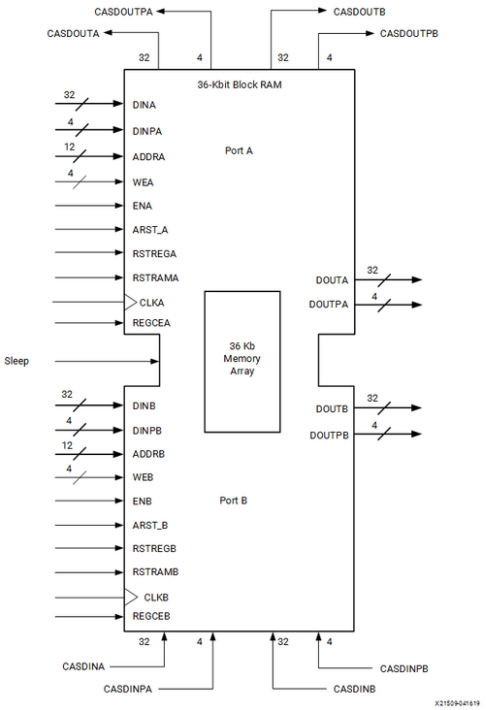


(b) FF structure. Source: [7].

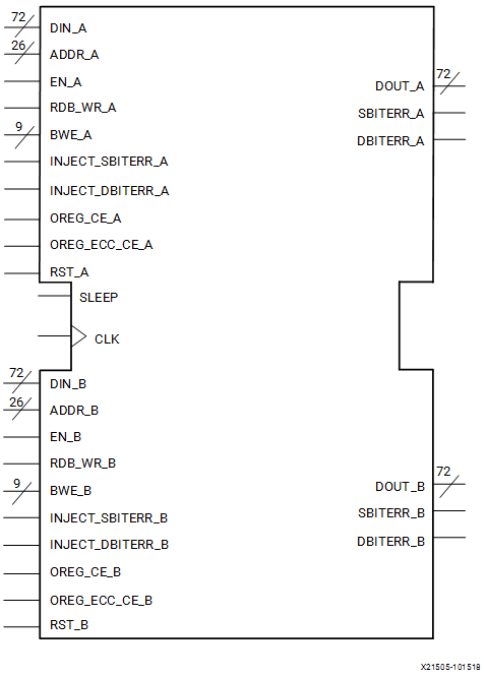


(c) DSP structure. Source: [8].

Figure 3.2: Internal structure of different FPGA resources.



(d) BRAM structure. Source: [9].



(e) URAM structure. Source: [9].

Figure 3.2: Internal structure of different FPGA resources.

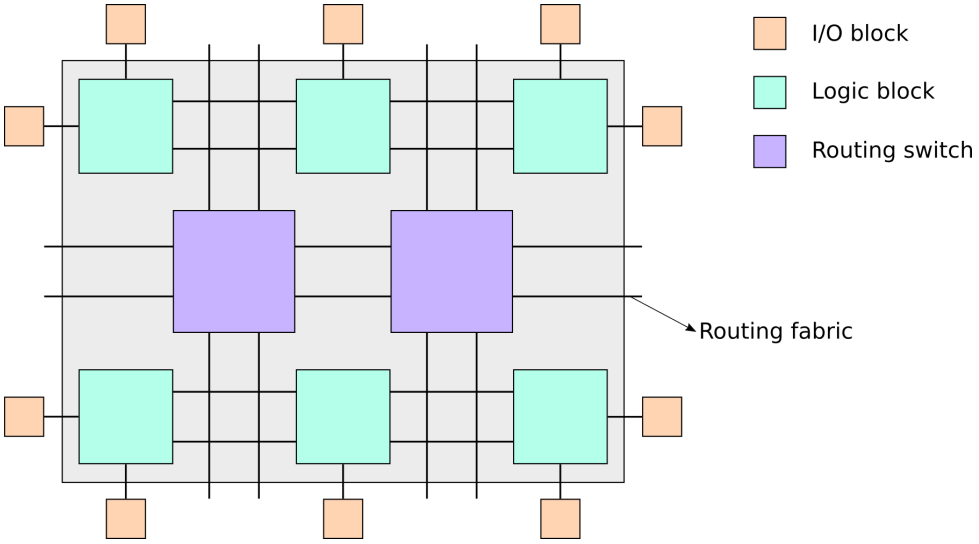


Figure 3.3: FPGA logic architecture.

3.1.3 Timing

Timing in an FPGA refers to the behavior of the digital signals within the FPGA and how they are synchronized with the clock signal [10]. It is determined by several factors, including the clock frequency, the delay through the routing resources, the delay through

the logic elements, and the setup and hold times of the FF; those factors are also known as timing requirements.

The goal is to ensure that all of the signals in the design are stable and valid when the clock edge arrives and that there is sufficient time for the signal to propagate through the FPGA resources before the next clock edge. It is critical to ensure that the design timing requirements are met to ensure that the design functions correctly and operates at the desired speed.

Developers have to define the time requirements, which will be essential during the bitstream generation process.

WNS

Worst Negative Slack (WNS) is a measure used in FPGA timing analysis. It represents the maximum amount of time by which a path in the design fails to meet the timing requirements. It is an important parameter to consider when verifying the performance of a digital design. WNS is the largest negative slack value among all the paths in the design and represents the worst-case timing violation [11].

Regarding the WNS value, there are two possible scenarios:

- WNS < 0: It means that the path is violating the timing requirements.
- WNS ≥ 0: It means that the path is meeting the timing requirements. Having positive WNS shows that there is some margin in the design, so more RTL could be added, for example.

To ensure reliable operation of a design, all timing paths must have either positive or zero slack.

Setup and hold time

The setup time and hold time are timing requirements that determine the stable and valid input conditions for a FF.

Setup time refers to the minimum amount of time that an input signal must be stable and valid before the clock edge arrives. It ensures that the input signal has settled and is stable, so that the FF can correctly capture the value of the input [12]. If the input signal changes too close to the clock edge and does not meet the setup time requirement, it can lead to incorrect data being captured by the FF, which is called setup time violation.

Hold time is the minimum amount of time that an input signal must remain stable and valid after the clock edge arrives. It ensures that the input signal remains stable during the FF's internal operation [12]. If the input signal changes too soon after the clock edge and does not meet the hold time requirement, it can lead to incorrect data being propagated through the FF, which is named hold time violation.

A visual representation of a correct setup and hold time, and a setup and hold time violation is shown in Figure 3.4.

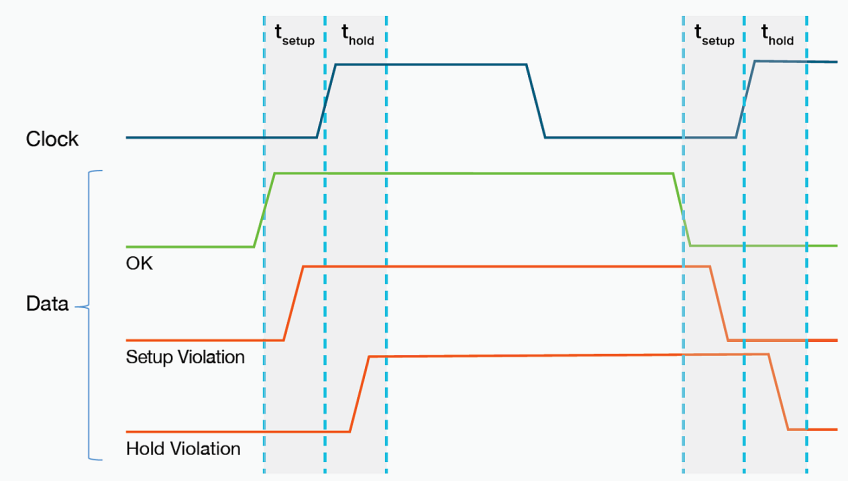


Figure 3.4: Setup and hold time violation signal diagram. Source: https://www.designnews.com/electronics-test/how-track-down-setup-and-hold-violations-mixed-signal-oscilloscope

When designing with FPGAs, it is key to meet the setup and hold time requirements to guarantee a reliable and correct functioning of the design. Failing to meet these timing requirements can lead to functional errors, timing violations, or other issues in the design.

3.1.4 Pins

The FPGA pinout is the mapping of the physical pins on an FPGA to the logical pins used in an RTL design. In other words, the pinout defines which physical pins on the FPGA are connected to which signals in the design.

It determines how the RTL can be linked to other components in the board, such as memory, PCIe, LEDs, an other interfaces [11]. The pinout also determines the routing of signals within the FPGA, which can affect the performance and power consumption of the design. In the case of Xilinx FPGA, the pins are grouped in banks, which will affect the bitstream generation.

FPGA pinouts are typically documented in the datasheet or user guide provided by the FPGA manufacturer, and example of pinout diagram is shown in Figure 3.5. That schematic view of the pin assignment corresponds to signals of the QSFP connector from the FPGA used in this project. Each FPGA has its pin mapping, which has to be kept in mind when adapting a design from one FPGA to another. Developers must define the pinout when generating the bitstream.

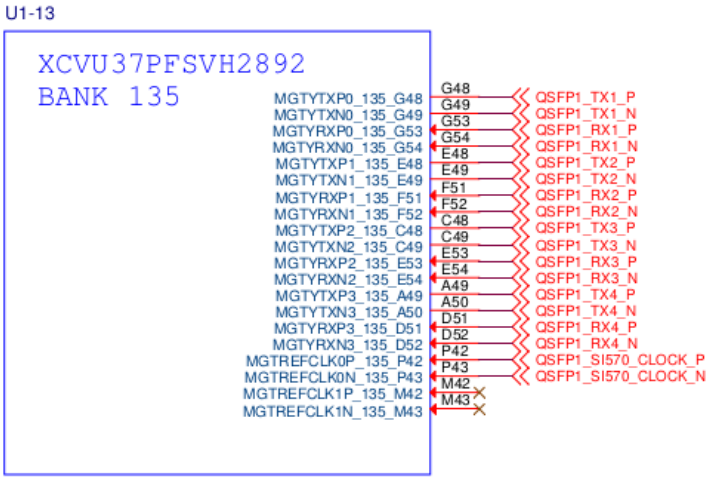


Figure 3.5: XCVU37P I/O Bank 125 Diagram, pinout. Source: [13].

3.1.5 IPs

An FPGA IP is a pre-designed and pre-verified building block that can be integrated into an FPGA design to provide a specific functionality [14]. It eases the usage of FPGA elements.

For example, a developer wants to use the DDR memory from the FPGA, but without dealing with the DDR physical protocol, which would require implementing the DDR controller from scratch. In that case, an IP can be instantiated, which offers an Advanced eXtensible Interface (AXI) interface and it internally adapts the AXI signals into the physical protocol (signals and timing) required by the DDR memory.

These IP cores are typically provided as configurable, customizable, and reusable components that can be integrated into a larger design.

Xilinx provides a wide range of IP cores that cover various functions and interfaces, like memory controllers, Ethernet, USB, PCIe, and many others. These IP cores can be used to accelerate the design process, reduce development time, and improve the performance and reliability of the overall system.

There are two IP types: hard and soft IPs. Those are defined in the sections below.

Hard IP

A hard IP is a functional block that is integrated into the FPGA as a fixed, dedicated hardware module. Those blocks are integrated into the FPGA board at the manufacturing stage, and cannot be changed or reconfigured by the user after the device has been produced. Therefore, when using hard IPs in a design, they do not require logic FPGA resources, such as LUTs.

Depending on the FPGA model, hard IPs support hardware features provided by the FPGA: for example, the FPGA board used for this work embeds 16 PCIe links, thus

allows the user to configure and instantiate the two hard IPs responsible for managing the PCIe, which are the PCIe endpoint and the PCIe clock.

Soft IP

A soft IP is a functional block that is implemented using programmable logic resources within the FPGA. As a result, its usage increases the FPGA resource utilization.

Soft IP blocks are typically provided as synthesizable RTL code, which can be integrated into a user's design. They can be customized by modifying the RTL code or by configuring the block's parameters and options.

Compared to hard IPs, soft IPs offer greater flexibility and configuration options, but may not provide the same level of performance. Soft IP blocks are also subject to the constraints and limitations of the programmable logic resources within the FPGA.

3.2 SDV insfrastructure

The objective of the Software Development Vehicles (SDV) infrastructure is providing feedback to the architects designing EPAC-VEC and to the engineers implementing the RTL. Additionaly, SDV offers porting, testing, benchmarking, and optimizing software on the new proposed hardware as early as possible [15]. In order to achieve them, a hardware platform, mainly made up of FPGAs, and software tools have been developed. This hardware and software combination is called SDVs.

The hardware platform is composed by RISC-V scalar CPU comercial boards (called Arriesgado in our setup) and FPGAs. The SDV's FPGA, referred to as FPGA@SDV from now on, is an FPGA with some "glue logic" and the EPAC design, but the only accelerator instantiated is one EPAC-VEC. The SDV hardware is placed in a cluster, called HCA, and is managed with Slurm. More details about Slurm and the SDV hardware nodes are given in the sections below. Figure 3.6 represents the cluster infrastucture.

The reason behind using SDV along with a RTL simulators, which provide detailed simulations of the design and allow detecting bugs at RTL level, is that RTL simulators are not the best choise when testing large-scale programs, like operating systems or scientific applications. For reference, the process of booting a lightweight distribution of Linux in the FPGA@SDV design consumes less than 5 minutes. Whereas, the simulation could require hours, or even days. Therefore, leveraging the FPGA@SDV not only speeds up the debugging process but also enhances the efficiency in rectifying any bug in the EPAC core.

Slurm

Slurm is an open-source workload manager and job scheduler designed for HPC clusters. It provides a centralized framework for managing and scheduling tasks, allocating computing resources, and monitoring job execution in a distributed computing environment. Slurm

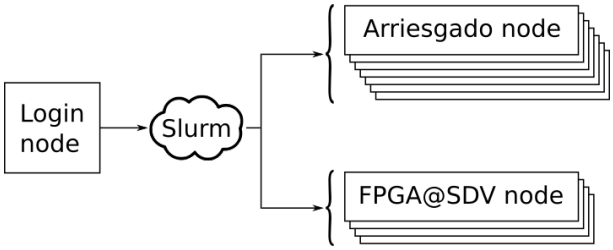


Figure 3.6: Hardware SDV infrastructure in our cluster.

enables efficient utilization of HPC resources and eases the execution of large-scale parallel and batch processing workloads [16].

Slurm offers partitions, which are logical groups of computing resources within a cluster. Each partition typically represents a subset of the cluster's computing nodes with similar capabilities, such as processor type, memory capacity, or network connectivity. By defining partitions, system administrators can effectively manage and distribute workload across different sets of resources.

Slurm is used in HCA to manage the Arriesgado and FPGA@SDV partitions.

Arriesgado

The Arriesgado partition is composed of seven distinct Arriesgado nodes. Each node is a HiFive Unmatched board[1], which is equipped with four RISC-V scalar cores operating at 1 GHz. These nodes are key in compiling and testing binaries directly in the native RISC-V architecture, so cross-compilation is not needed.

Pickle

There are four FPGA nodes known as Pickle, the so called FPGA@SDV nodes. Each Pickle node consists of an x86 CPU with a VCU128 FPGA board, interconnected through PCIe, JTAG, UART, and Ethernet. The FPGA can be reprogrammed via the x86 core, which also serves to load the Linux image and initiate a UART shell or SSH connection. More details about those interfaces are explained in Section 3.3.1.

Those Pickle nodes can be accessed through two different partitions. In other words, the 4 Pickle nodes can be allocated via 2 partitions, which share the nodes.

The first partition is `fpga-sdv`. When you request a Pickle node from that partition, the FPGA is configured by Slurm and the user gets a ready-to-use FPGA. The second partition is called `fpga` and when the users allocates it, you have to configure manually the FPGA. The `fpga-sdv` partition targets users who do not need a custom bitstream, so they use bitstreams previously built by ourselves, the SDV team. Whereas the `fpga` partition is aimed at developers who require a specific, custom bitstream build by themselves. It is mostly used by the SDV team.

[1]HiFive Unmatched https://www.sifive.com/boards/hifive-unmatched

3.3 VCU128

The FPGA board used in EPAC is the Virtex Ultrascale+ HBM VCU128 FPGA Evaluation Kit [3], which has the Xilinx Virtex UltraScale+ VU37P HBM (XCVU37P) FPGA.

An evaluation kit is a board that integrates hardware (such as RAM memory), IPs, design tools, and pre-verified reference designs, to ease the development of designs and applications [17].

The VCU128 offers different types of connections and features, the most relevant ones for this thesis and the SDV design are defined in Table 3.1.

ID	FPGA Component	Usage in FPGA@SDV
1	XCVU37P FPGA	EPAC core
2	8 GB of High Memory Bandwidth (HBM)	Tracer memory
3	4.5 GB of DDR4	Core memory
4	PCIe Gen3 x16 or Gen4 x8	Offloading of the Linux image
5	USB UART-JTAG	UART shell and ILA debuging
6	10/100/1000Mb/s Ethernet	SSH connection and access to NFS

Table 3.1: Components from the board VCU128 and their correspondant usage in the SDV design.

Figure 3.7 shows a VCU128 board with the SDV relevant components identified, following the ID numbers from Table 3.1.

3.3.1 Usage in FPGA@SDV

The FPGA@SDV design is formed by the EPAC accelerator RTL and some called "glue" logic (FPGA shell). That design is loaded into the XCVU37P FPGA. The FPGA shell makes possible: *i)* usage of memory, *ii)* communication with the FPGA/core, and *iii)* debugging RTL.

Both available memories are used in FPGA@SDV: DDR4 and HBM; but the DDR4 is the one used as memory by the EPAC accelerator, and the HBM memory is utilized for debugging purposes. The DDR4 is where the Linux image is loaded and it is accessible through DMA, a communication protocol, engine between devices (more details about DMA are provided in Section 4.2).

Communications with the core can be performed in different ways, depending on the objective. The PCIe interface is used to load, write, the Linux image from the host into the FPGA's DDR4 memory, allowing the core to boot Linux. The USB UART is utilized to open a UART shell. UART is a serial communication protocol between devices. Therefore, an interactive session can be opened in the Linux image booted at the FPGA, from the EPAC core. Finally, the last communication interface used is the Ethernet, which allows establishing an SSH connection.

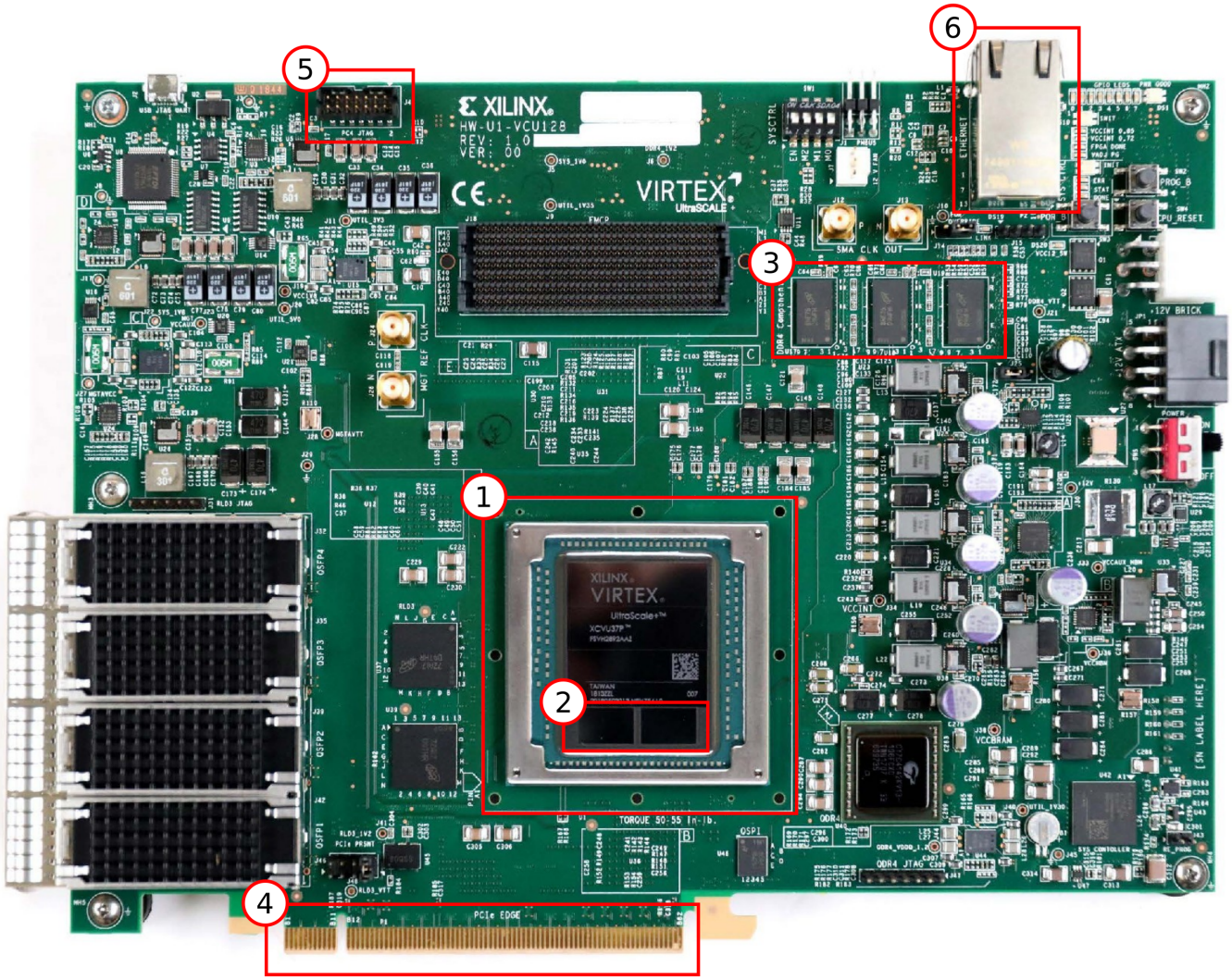


Figure 3.7: VCU128 board with key components identified.

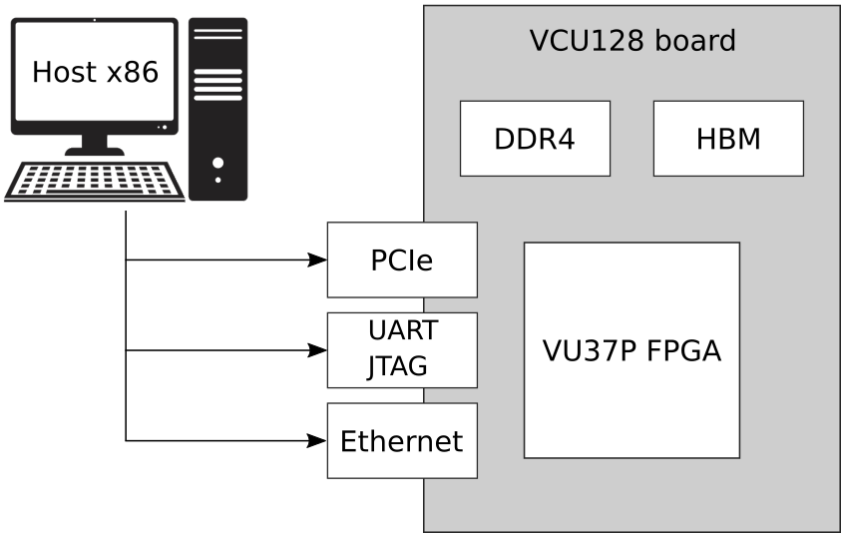


Figure 3.8: Pickle node connections between the VCU128 and the x86 host.

To debug the RTL the components utilized are HBM and Integrated Logic Analyzer (ILA). The HBM is used to store information for a partner tool called "tracer", which is used to debug the scalar core. The ILA is a logic analyzer core that allows monitoring internal signals of a design [18]. A JTAG connection is required to read the ILA outputs.

Figure 3.8 shows a schematic of a Pickle node. It can be observed the previously mentioned connections and resources.

3.4 Xilinx Software Environment

Xilinx is a company dedicated to designing and manufacturing FPGAs and other programmable logic devices. Moreover, it develops its own software tools and IP cores, to enable users to create and customize FPGA designs more easily and efficiently.

3.4.1 Vivado Design Suite

Vivado Design Suite (from now on Vivado only) is a set of software tools from Xilinx. It is used for designing, simulating, and implementing digital circuits and systems on Xilinx's FPGAs and other programmable devices. It provides an integrated development environment (IDE) with both a graphical user interface (GUI) and a command-line interface (CLI) for developing and deploying designs on Xilinx's devices [19].

Vivado supports HDLs, such as Verilog and VHDL, and high-level synthesis languages, like C and C++.

A Vivado project is a set of design files and settings used to design, implement, and program a specific FPGA. The design files include RTL, constraint, and simulation files.

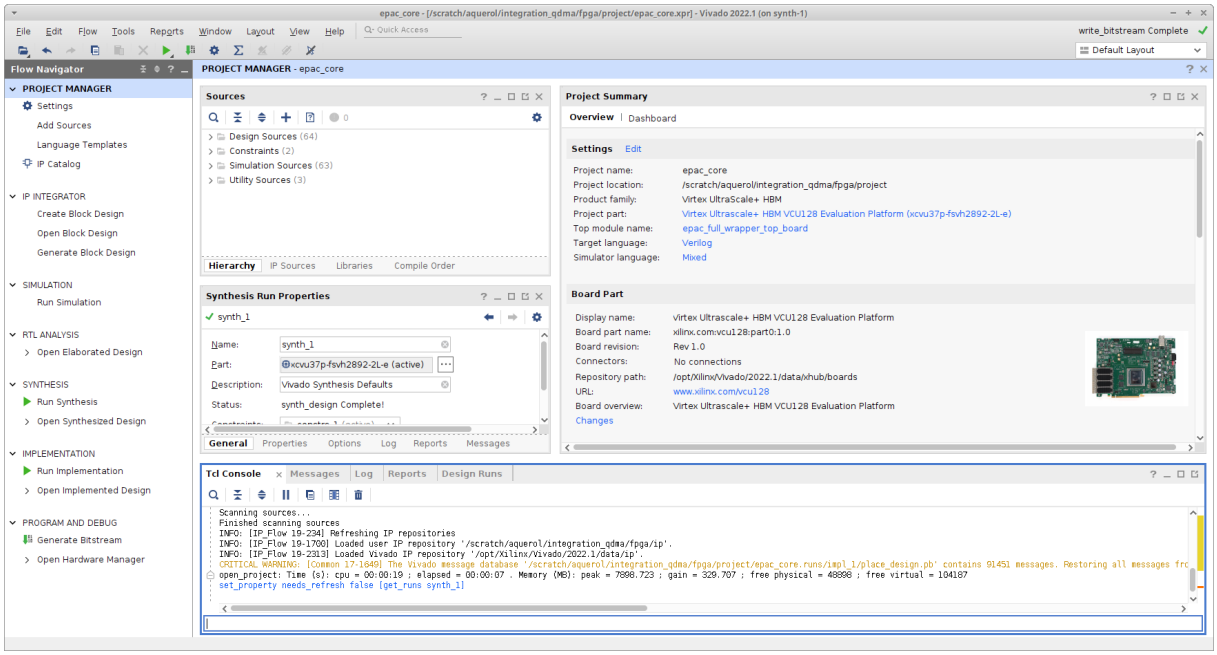


Figure 3.9: Screenshot from Vivado Design Suite GUI.

A Vivado project requires defining the target device, which can be a board (evaluation kit) or part (FPGA chip only).

Figure 3.9 shows a Vivado Design Suite GUI with a project opened. On the left, there is access to the different features and tools provided. The relevant ones for this thesis are the *IP Integrator*, *Synthesis*, *Implementation*, and *Program and Debug*.

The Vivado version used in this thesis is Vivado 2021.2.

Vivado IP Integrator

The Vivado IP Integrator is a graphical tool for integrating and customizing IP cores, which can be provided either by Xilinx or by other developers [19].

It allows the creation of Block Designs (BDs), which are graphical representations of a set of IPs and interconnections. In other words, a BD is a way to visually organize and interconnect different soft and hard IPs.

In the Vivado IP Integrator, when a BD is opened, IPs can be instantiated, customized, and connected. The graphical representation of an IP shows the input and output interface and allows configuring its parameters. In Figure 3.10 there is a Vivado IP menu: on the left, there is the IP graphical representation with its input/output signals, and on the right, there is the configuration part with different tabs.

Figure 3.11 shows an example of BD with 2 IPs interconnected and 3 of their signals marked as external. Marking a signal as external allows connecting those signals to FPGA physical pins or other signals in the RTL code. Otherwise, those signals would remain local to the BD, hence not accessible from outside the BD.

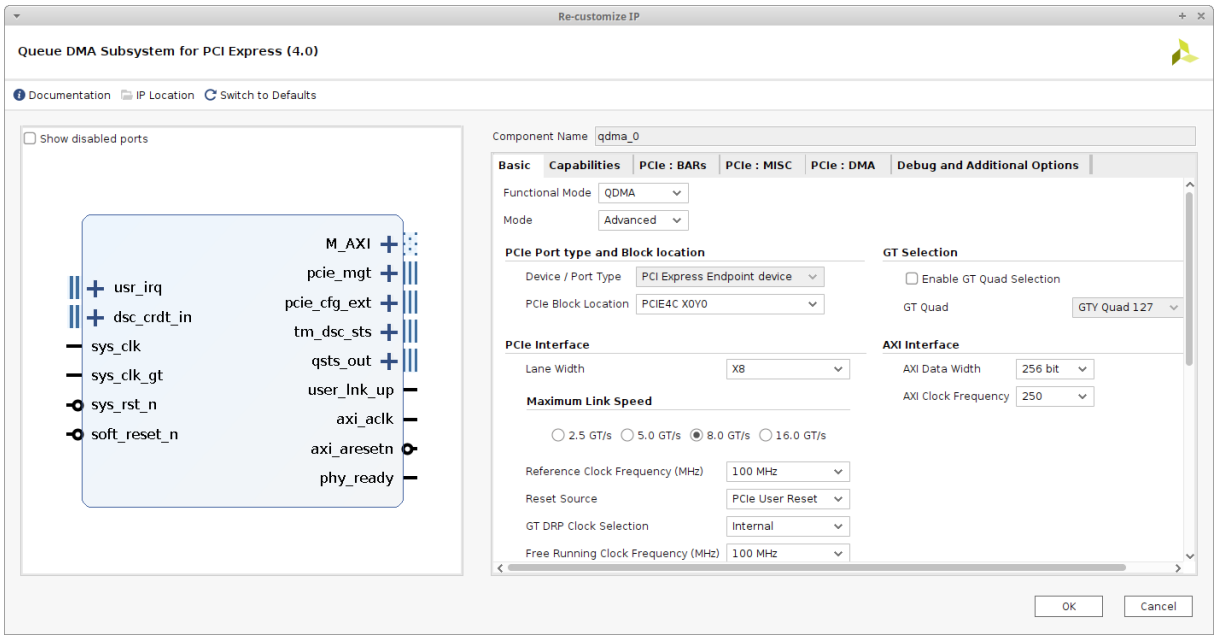


Figure 3.10: Vivado IP menu from a Xilinx QDMA IP.

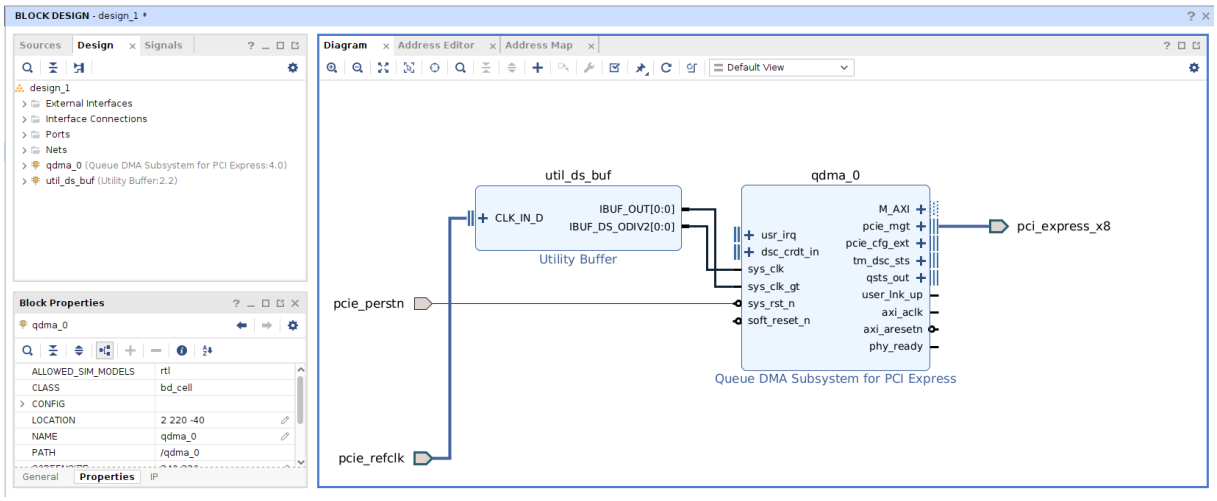


Figure 3.11: Vivado IP menu from a Xilinx QDMA IP.

3.4.2 Constraints

Xilinx has developed a format of defining timing and pin requirements for their FPGAs called Xilinx Design Constraints (XDC). XDC consists of commands defining the requirements that must be met during the bitstream generation, so the design is functional on the board [20].

The FPGA timing and pin requirements are known as constraints in the Xilinx environment. Therefore, they are defined in an XDC file.

Timing constraints

The timing constraints allow defining clocks. Listing 3.1 contains an example of a timing constraint, linking the `ddr_clk` port to the clock driven by `mmcm_clkout0`.

```
1 set ddr_clk     [get_clocks mmcm_clkout0]
```

Listing 3.1: XDC timing example.

Pinout constraints

Two properties are used to define pinout constraints: `PACKAGE_PIN` and `IOSTANDARD`.

`PACKAGE_PIN` property defines a specific assignment of a port in the logical design to a physical package pin in the FPGA [21].

`IOSTANDARD` is used to specify with which programmable IO Standard a port has to be configured [21]. An IO Standard is a predefined voltage. The possible values of `IOSTANDARD` are not relevant to this thesis.

Those properties are usually defined together and its syntax is shown in Listing 3.2. Hence, assigning an IO standard to a port means defining at which voltage that port will work, since some pins support different voltage levels. However, not all physical pins must have an IO standard defined.

```
1 set_property PACKAGE_PIN pin_name [get_ports port_name]
2 set_property IOSTANDARD value [get_ports port_name]
```

Listing 3.2: XDC `PACKAGE_PIN` and `IOSTANDARD` syntax.

Listing 3.3 contains an example of a pinout definition using both properties.

```
1 # Designates STATUS to be placed on pin B26
2 set_property PACKAGE_PIN B26 [get_ports STATUS]
3 # Sets the I/O Standard on the STATUS output to LVCMOS12
4 set_property IOSTANDARD LVCMOS12 [get_ports STATUS]
```

Listing 3.3: XDC pinout example.

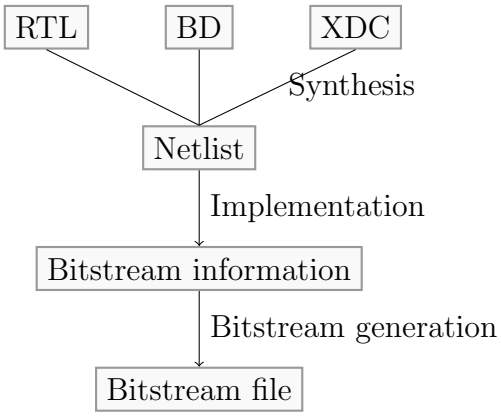


Figure 3.12: Bitstream generation flow in Vivado.

3.4.3 Bitstream generation

Bitstream generation is a complex process with different phases.

Firstly, a netlist from the design needs to be generated. A netlist is a gate-level list with all the connections between the various FPGA logical resources of a design. This step is known as synthesis and its output is the netlist.

Secondly, placement is performed using as input the netlist generated during synthesis. The placement consists of mapping the logical resources from the netlist into the specific physical locations following the physical constraints of the FPGA. This step is known as placing.

Thirdly, those placed resources are connected with the available interconnect resources on the FPGA. Taking the result of the placement phase, it is determined the best paths to connect them, so the timing is optimized and the constraints are fulfilled. This step is known as routing.

Finally, the bitstream file is generated, ending up the process with a file where all the necessary information for the FPGA is written. The bitstream is loaded onto the FPGA to configure its logic and interconnects.

The program used to generate bitstreams is Vivado, since the FPGAs used in EPI are from Xilinx. In the Vivado workflow, the first step is also called *synthesis*, while the second and third ones are grouped into a step named *implementation*. Figure 3.12 illustrates the Vivado bitstream generation flow.

Those phases need different complex algorithms (NP-Complete), which are optimized with heuristics to reduce the bitstream generation time. As they are algorithms based on heuristics, the final bitstream can change from one run to another.

3.4.4 Resource utilization

Vivado offers the possibility of extracting the number of FPGA logic resources estimated after synthesis and the actual number of resources used after implementation. It is known as resource utilization.

Name	CLB LUTs (1303680)	CLB Registers (2607360)	CARRY8 (162960)	F7 Muxes (651840)	F8 Muxes (325920)
dbg_hub (dbg_hub)	434	741	7	0	0
inst (xsdbm_v3_0_0_xsdbm)	434	741	7	0	0
BSCANID.u_xsdbm_id (xsdbm_v3_0_0_xsdbm_id)	434	741	7	0	0
CORE_XSDB.U_ICON (xsdbm_v3_0_0_icon)	16	28	0	0	0
CORE_XSDB.UUT_MASTER (xsdbm_v3_0_0_icon	262	554	6	0	0
SWITCH_N_EXT_BSCAN.bscan_inst (lt1ib_v1_0_0	0	0	0	0	0
SWITCH_N_EXT_BSCAN.bscan_switch (xsdbm_v	122	125	1	0	0

Figure 3.13: Resource utilization report from Vivado GUI.

Vivado shows a hierarchical table whose first column contains module's names and the subsequent columns hold the different FPGA resources exploited in the design. The resource usage can be expressed in absolute numbers or percentage form. Figure 3.13 shows it with absolute numbers.

Chapter 4

PCIe and DMA

This chapter contains more technical information. The reasoning for having the technical knowledge separated is that the previous chapter is composed of information that I already knew prior to this thesis, whereas this chapter's data was new for me and I had to dedicate time to look for documentation and understand it.

For this reason in this chapter, firstly, an explanation of PCIe architecture, DMA protocol and AXI protocol is provided in Section 4.1, Section 4.2, and Section 4.3, respectively.

Secondly, the Xilinx DMA subsystem for PCIe called XDMA is described, which is a Xilinx soft IP, in Section 4.4.

Thirdly, the RDMA protocol is exposed in Section 4.5, as the next Xilinx IP implements it. Remote Direct Memory Addressing (RDMA) is targeted for high-performance DMA transferences.

Fourthly, and lastly, the Xilinx PCIe subsystem implementing high-performance DMA (RDMA) is explained. It receives the name QDMA and it is detailed in Section 4.6.

All these concepts had to be fully understood to be able to proceed with the technical development of this thesis.

4.1 PCIe architecture

Peripheral Component Interconnect Express (PCIe) is a high performance IO bus standard used to interconnect peripherals devices [22]. It has applications in computing and communication platforms. PCIe is commonly used for connecting graphics cards, network cards, sound cards, FPGAs and other peripherals to the host processor.

PCIe is a faster and more flexible alternative to previous generation bus architectures, such as Peripheral Component Interconnect (PCI). It uses a serial and high-speed point-to-point interface for communication between two devices; whereas PCI uses a parallel interface. That difference is key, since a serial interface allows for higher data transfer rates than the parallel interface used by older standards.

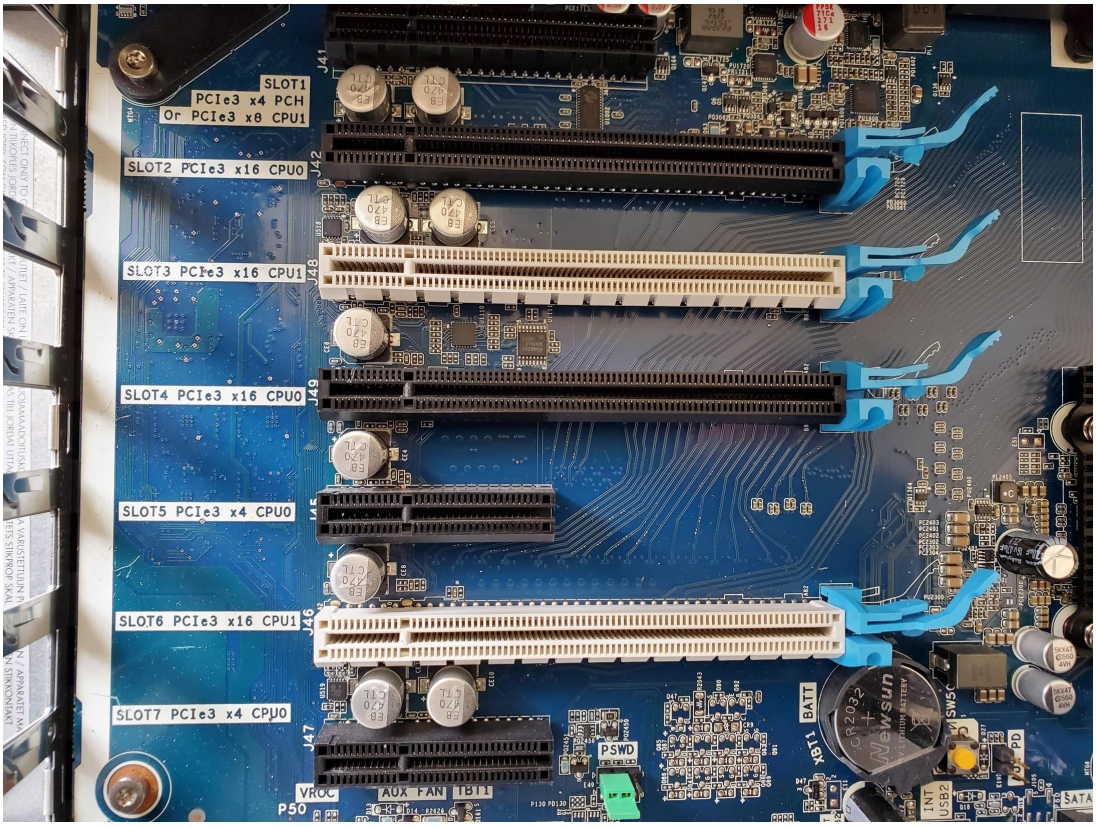


Figure 4.1: Motherboard with PCIe slots of x16 and x4 lanes. Source: https://superuser.com/questions/1541083/how-to-choose-the-right-gpu-topology-pcie-lanes-for-multiple-gpus

PCIe supports the same communication model as PCI. It supports the following transaction types: memory read/write, IO read/write, and configuration read/write.

There are several generations of PCIe, each one with higher transfer rates, hence higher bandwidth (BW). Table 4.1 shows a comparison between the different PCIe generations. All generations are backward compatible. Moreover, PCIe architecture is compatible with the PCI one.

A PCIe point-to-point interconnection is called lane. A lane consists of two pairs of differential signals: `txn, txp`, and `rxn, rxp`. The `tx` signal pair is for transmitting (writing) data. The `rx` signal pair is for receiving (reading) data. The interconnection can have x1, x2, x4, x8, x12, x16 or x32 lanes. The bandwidth of the interconnection is determined by the number of lanes and the PCIe generation.

Typically, a peripheral with a PCIe connection is plugged into a motherboard with a PCIe connector, called a PCIe slot. On the one hand, Figure 4.1 displays PCIe slots with different lane sizes. On the other hand, there is a peripheral PCIe connection shown in Figure 3.7, identified with the number 4.

Measure	PCIe Generation				
	PCIe 1.x	PCIe 2.x	PCIe 3.x	PCIe 4.x	PCIe 5.x
Raw bit rate	2.5 GT/s	5.0 GT/s	8.0 GT/s	16.0 GT/s	32 GT/s
Interconnect BW	2 Gb/s	4 Gb/s	8 Gb/s	16 Gb/s	32 Gb/s
BW Lane Direction	250 MB/s	500 MB/s	1 GB/s	2 GB/s	4 GB/s
Total BW x16	8 GB/s	16 GB/s	32 GB/s	64 GB/s	128 GB/s

Table 4.1: PCIe rate and bandwidth comparison between different PCIe generation. Source: https://blogs.synopsys.com/expressyourself/2017/08/15/1-2-3-4-5-its-official-pcie-5-0-is-announced/.

4.1.1 Interrupts

An interrupt is a signal sent by the hardware to the processor, which has to handle it [23]. In the context of PCIe, an interrupt is sent by the peripheral to notify an event to the CPU, such as the completion of a transference.

Each PCIe device has a unique interrupt number, also known as an interrupt request line. That number is then read and used by the corresponding device driver and device [23].

PCIe supports two interrupt mechanisms: legacy interrupts and Message Signal Interface (MSI). The former is from the PCI bus, and the latter is the native PCIe interrupt delivery mechanism. Both interrupt types are supported since PCIe is backward compatible with PCI.

Legacy interrupts use one of the interrupt lines to signal interrupts. The interrupt line can be `INTA, INTB, INTC`, or `INTD`, which were defined for the PCI bus. In the PCI architecture, those interrupt lines are physical pins, which are asserted and de-asserted. Whereas, in the PCIe architecture, it is optional to support legacy interruptions, and if supported, an in-band message is defined to act as virtual `INTx` wires [22].

MSI is implemented as a PCIe memory write transaction written to an Interrupt delivery address [22]. It is more efficient and allows more interrupts per card than legacy interrupts. That is because each device has a unique MSI address and can send a message directly to the CPU without having to share the interrupt line with other devices.

An extension of MSI was defined, called MSI eXtended (MSI-X), to provide more interrupt vectors. It allows signaling multiple interrupts by a single device. MSI-X uses a table to map device interrupt requests to messages and interrupt vectors. Devices supporting MSI-X feature a dynamically programmable hardware table. The MSI-X table is usually programmed by the device driver during initialization [24].

4.1.2 Functions

A PCIe function is a logical device within the PCIe device that performs a specific function, with its own set of resources [22]. A PCIe device can have one or more functions,

each one with a unique PCIe function number. In other words, in the same PCIe slot, there can be different PCIe functions, logical devices.

For example, a PCIe network card can have multiple Network Interface Cards (NICs) and each one be logically separated, reporting different functions.

Physical function

A Physical Function (PF) refers to a specific instance of a PCIe function within a device, which serves as the primary function of the device. The PF typically provides the main functionality of the device and acts as the interface to the host system. Therefore, it is responsible for handling device initialization, configuration, and communication with the host system.

Virtual function

Virtual Functions (VFs) represent additional functions instances within the device and provide additional functionality or features. VFs are created and managed by the PF.

VFs are typically used in scenarios where the device needs to be shared by multiple users or Virtual Machines (VMs). Each VF can be assigned to a specific user or virtual machine, enabling resource partitioning and isolation. That is because each VF has its own configuration space and can be independently configured and controlled.

Single Root I/O Virtualization (SR-IOV)

SR-IOV is a technology that allows a single physical PCIe device to be virtualized and shared among multiple VM or containers. It is an extension of the PCIe specification [25]. It enables direct and efficient access to the device's resources by the virtual instances, improving performance and reducing overhead.

Traditionally, when multiple VMs or containers share a physical device, the hypervisor or host Operating System (OS) manages the device access and performs IO virtualization, which introduces overhead and can limit the performance of the shared device.

With SR-IOV, a physical PCIe device is partitioned into multiple VFs and a single PF. The PF represents the primary interface to the device, and the VFs represent the virtual instances of the device. VFs appear as separate physical devices to the VMs or containers, allowing them to directly access the device's resources without involving the hypervisor or host OS.

By bypassing the virtualization layer, SR-IOV improves IO performance and reduces latency. It also enables efficient sharing of resources, as each VF can be independently allocated and managed, providing isolation between VMs or containers.

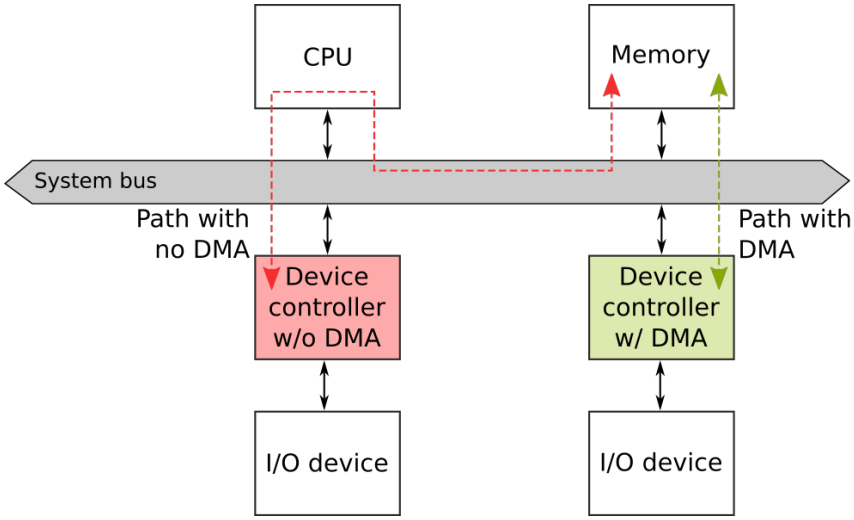


Figure 4.2: Simplified diagram of the data tranfer without DMA (in red) and with DMA (in green).

4.2 DMA protocol

Direct Memory Addressing (DMA) is a hardware mechanism that allows data to be transferred between devices without involving the CPU [23]. In other words, peripheral components can move data directly to and from CPU memory with no processor intervention.

In a typical computer system, data is transferred between devices such as hard drives or network adapters via the CPU. This means that the processor must handle all data transfers, which can be time-consuming and can limit the overall performance of the system. Therefore, the usage of DMA can *i)* improve throughput to and from a device as it reduces the computational overhead, and *ii)* increase system performance due to the reduction of the load on the processors and freeing it up to perform other tasks.

The DMA controller takes controll over the system bus and transfers data between the device and memory without involving the CPU. A simplified representation of the difference between a data transfer using DMA or not is illustrated in Figure 4.2.

DMA data transfers can be triggered via the software asking for data with functions, such as `read` or `write`, or via the device asynchronously pushing data to main memory [23]. The relevant trigger method for this thesis is the first one, hence the focus will be on that one.

The steps for a DMA read data transfer triggered by software are:

1. When a process calls read, the driver method allocates a DMA buffer and sends a request to the device to transfer its data into that buffer. Then, the process is put to sleep.

2. The device writes data to the DMA buffer, in the main memory, and raises an interrupt when it has finished.

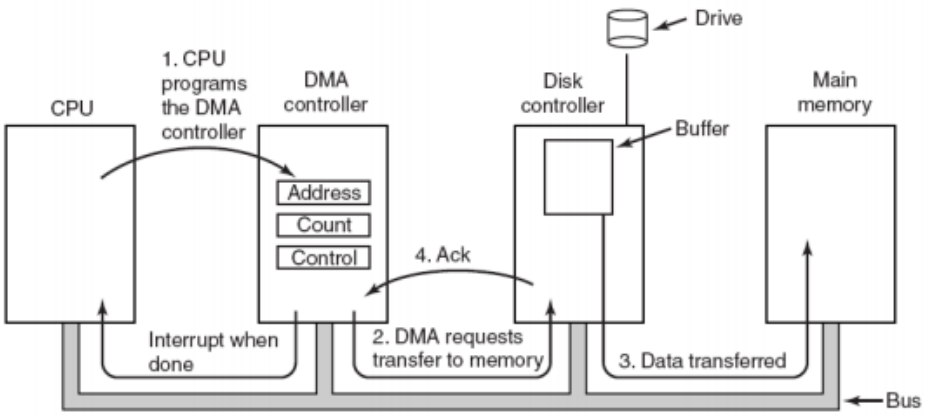


Figure 4.3: DMA read transfer issued to a memory disk. Source: https://examradar.com/direct-memory-access-questions-answers/

3. The interrupt handler gets the input data, acknowledges the interrupt, and awakens the process, which is now able to read data.

Figure 4.3 shows a DMA read transfer that is targeting a memory disk device.

The DMA write follows the same scheme as the DMA, but instead of transferring from device memory to main memory, it moves data the other way around: from main memory to device memory.

4.2.1 DMA gather/scatter transfers

In a standard DMA transfer, a device can only transfer data to or from a contiguous block of memory. This can be inefficient when data from non-contiguous memory locations need to be transferred, as it requires multiple DMA transfers to be performed (one per memory block). That would increase the overall overhead of the transfer and reduce system performance.

Therefore, instead of doing one DMA transfer per memory address, a unique DMA transfer can be issued. The DMA controller receives a "scatter/gather" buffer list and it is responsible to fetch the corresponding memory blocks. That allows issuing a single transfer, improving the overall performance.

Not all architectures support DMA gather/scatter transfers.

4.2.2 DMA support in PCIe

PCIe supports DMA, which improves the overall performance of PCIe operations.

There are two types of DMA transfers in PCIe:

- DMA read: Data is transferred from device memory to main memory.

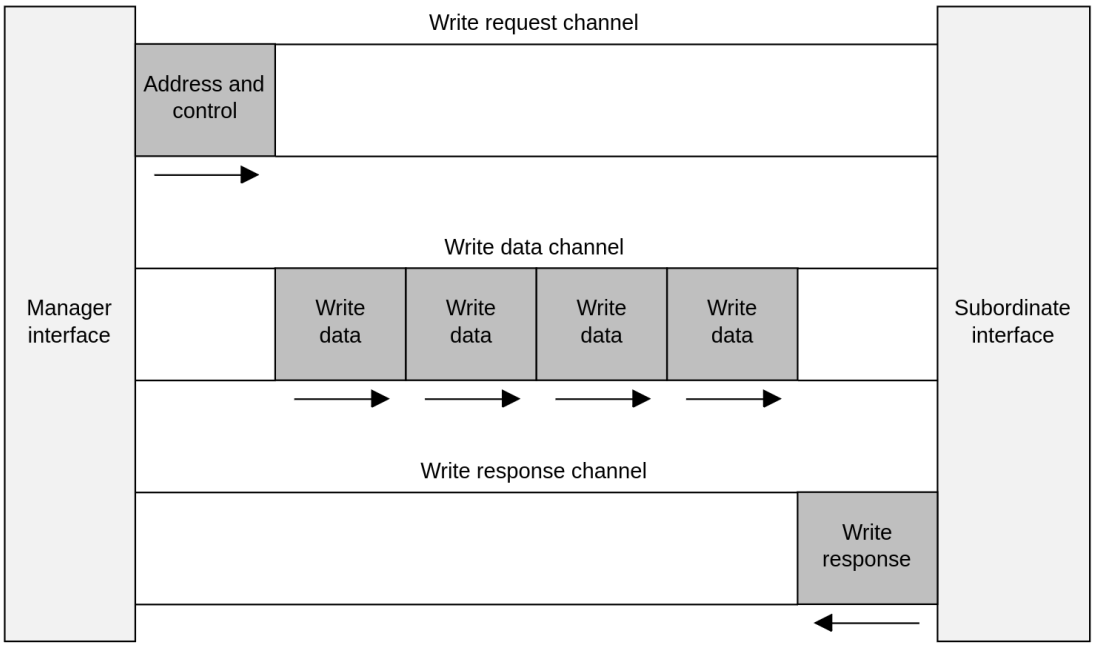


Figure 4.4: Channel architecture of write transactions. Source: [26].

- DMA write: Data is transferred from main memory to device memory.

DMA descriptors are necessary to perform them. They contain the information "describing" the data transfer with information such as the source and destination addresses.

4.3 AXI protocol

Advanced eXtensible Interface (AXI) is a high-performance, point-to-point protocol to connect Manager and Subordinate components to exchange data [26]. It is part of the AMBA (Advanced Microcontroller Bus Architecture) specification. Due to its purpose, it is useful to connect IPs in FPGA designs.

The AXI protocol offers separated address/control and data phases. It is based on using different channels for reading and writing, which provide low-cost DMA [26].

A channel is a set of signals with a defined purpose. There are three types of channels, which transfer requests, data, and responses. The specific signals of each channel are not relevant in this thesis scope. For the write transactions, the three channel types are used, as is shown in Figure 4.4. For the read transactions, only the request and data ones as Figure 4.5 illustrates.

There are different versions of the AXI protocol, each one providing a range of features and flexibility to meet the diverse requirements of different applications. The three types of AXI interfaces are:

- AXI: It is the full version of the protocol. It is also known in the industry as AXI Memory-Mapped (AXI-MM), since it requires memory addresses.

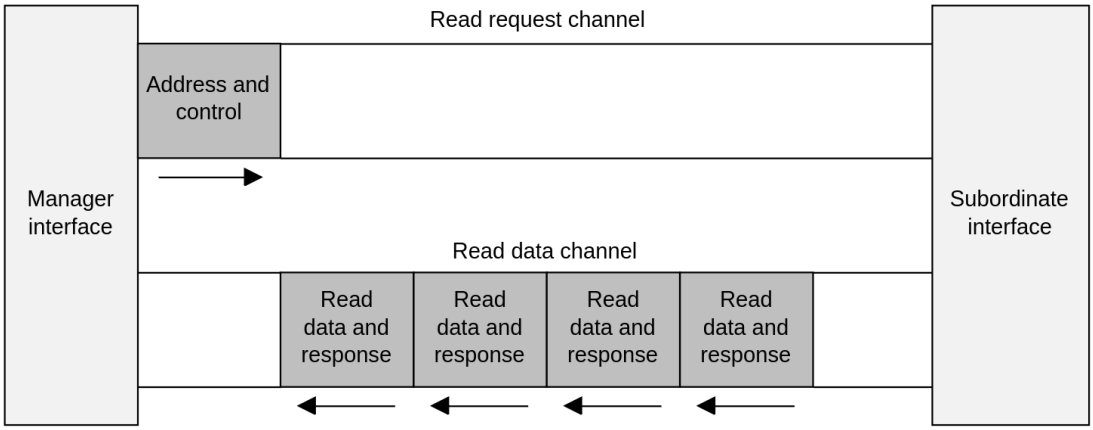


Figure 4.5: Channel architecture of read transactions. Source: [26].

- AXI-Lite: It is a simplified version of the AXI. It supports a subset of features, making it simpler but with lower throughput.
- AXI4-Stream (AXI-ST): It is a specialized version of the AXI targeting streaming data applications [27]. It supports a continuous stream of data without addressing (without address channels) and requires fewer resources than the AXI.

4.4 XDMA Xilinx IP

The Xilinx DMA (XDMA) Xilinx IP is a PCIe DMA subsystem [28]. It is a high-performance DMA engine integrated into Xilinx FPGAs with PCIe connectivity. The XDMA IP provides a scalable, high-bandwidth, and low-latency data transfer solution between the host computer and the FPGA board.

The features that XDMA offers and important for this thesis are:

- Up to 4 host-to-card (H2C) data channels and up to 4 card-to-host (C2H) channels. The H2C channels are for the write operations from the host, and the C2H channels for the read ones.
- A user interface that can be AXI-MM or AXI-ST.
- Interrupts can be legacy, MSI, or MSI-X.

However, it does not support SR-IOV. In addition to the limitations of this IP, it only supports 1 PCIe PF. These implies that the design with XDMA for the MEEP cluster cannot work, as XDMA does not support virtualization.

The XDMA IP exploited in FPGA@SDV is configured to be able to use all 4 channels per direction and the AXI-MM user interface.

The XDMA subsystem allows moving data between the host and FPGA memory by operating on DMA descriptors. Those descriptors carry information about the source and

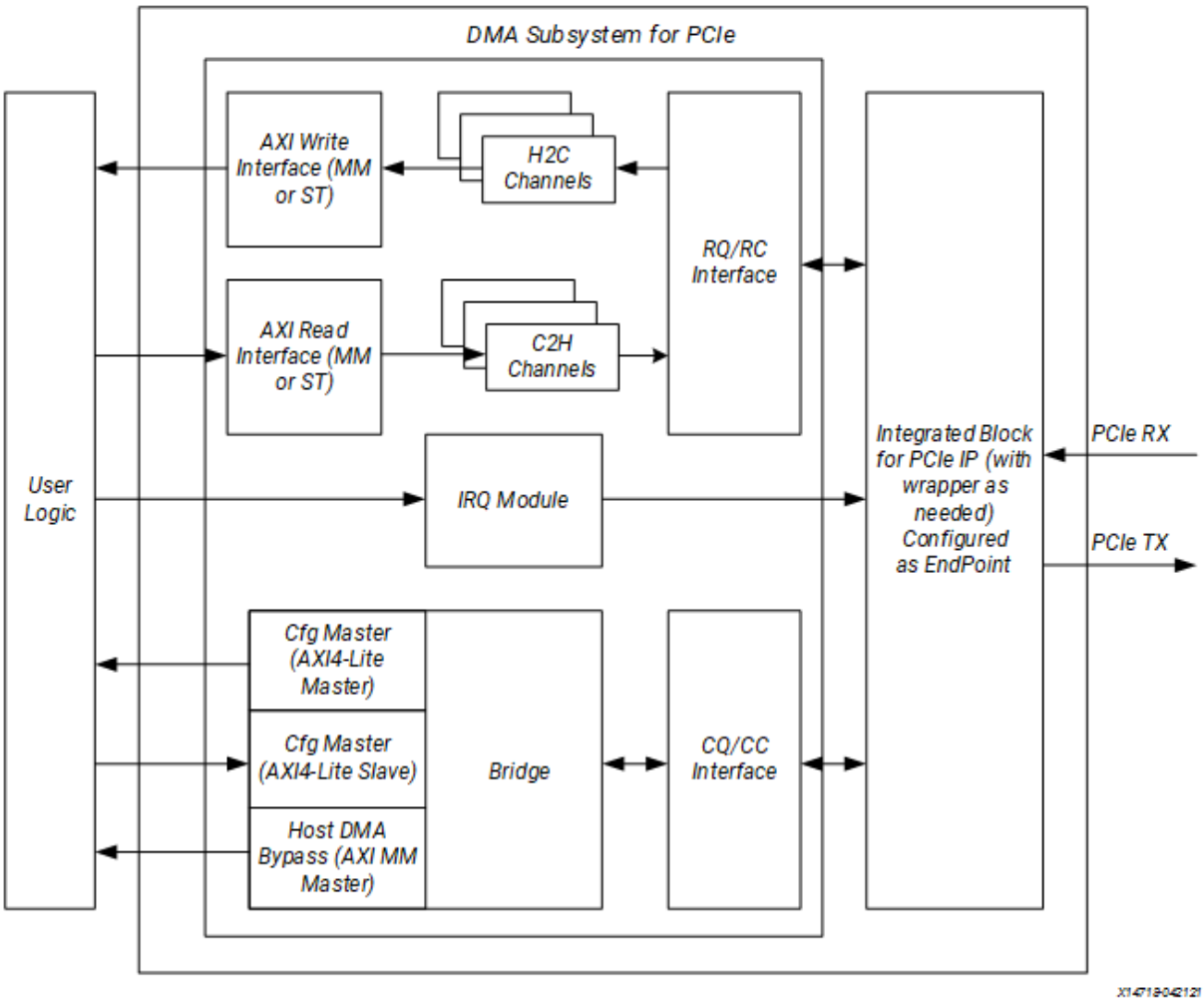


Figure 4.6: XDMA diagram. Source: [28]

destination addresses, and the amount of data to transfer. The transfers can be in both H2C and C2H directions.

Figure 4.6 contains the IP diagram. The arrows represent the bus connections between entities inside and outside the XDMA IP, which can be uni or bi-directional. The user logic block (on the left) represents the RTL connected to the IP design, i.e the logic that an RTL developer has to connect to the IP. The big block named *DMA Subsystem for PCIe* represents the XDMA IP with its engines. The focus of this section will be on *AXI Write Interface*, *AXI Read Interface*, and *IRQ Module*. On the right, there are two buses, *PCIe RX* and *PCIe TX*, that correspond to the physical PCIe connection.

4.4.1 Components

As could be seen previously, the XDMA IP is formed by different engines. Each channel is a DMA engine, so 8 DMA engines share the AXI-MM user interface.

From a high-level point of view, the H2C channel reads from the PCIe physical interface, *PCIe RX*, and sends what it has received to the user logic. The C2H channel behaves similarly, it reads from the user application and sends that data to the PCIe physical interface *PCIe TX*.

H2C Channel

The H2C channel takes care of DMA transfers from the host to the card. When it receives a transfer:

1. The DMA engine issues reads to the PCIe Requester reQuest (RQ) block, which means sending read requests to the host.
   A transfer is previously split into requests depending on the maximum request size defined.
2. The data from the host is received through the Requester Completion (RC) block.
3. H2C block issues write requests to the user interface.
4. After completing a transfer, the DMA engine issues an interruption or writeback to notify the host.

Figure 4.7 illustrates the previous steps.

C2H Channel

The C2H channel handles DMA transfers from the card to the host. The transfer processing consists in:

1. The DMA engine issues read requests to the user logic.

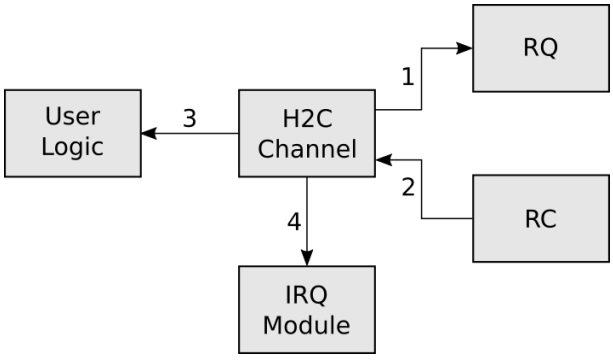


Figure 4.7: Diagram of a H2C transfer. The numbers correspond to the ones for the steps explained in the corresponding section.

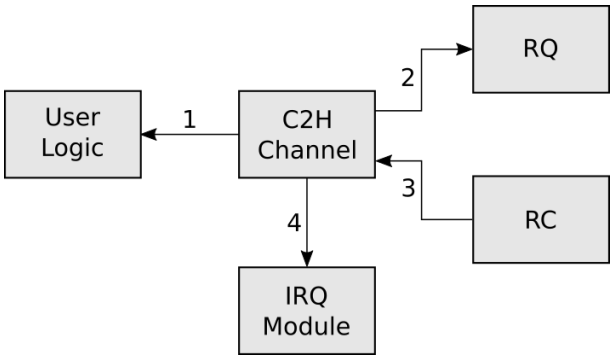


Figure 4.8: Diagram of a C2H transfer. The numbers correspond to the ones for the steps explained in the corresponding section.

2. Once the data is received, C2H engines issue a write request via the RQ block.
3. The RC block delivers the write request completion to the C2H block.
4. After completing a transfer, the DMA engine issues an interruption or writeback to notify the host.

Figure 4.8 illustrates the previous explanation.

IRQ Module

The IRQ module is resposible for generating interruptions over the PCIe link and receiving interrupts from the user application and from each DMA channel. It supports legacy, MSI, and MSI-X interruptions.

4.4.2 Operations

The DMA operations must move data between the host and FPGA memory. The host is responsible for allocating a buffer in its system memory and preparing the DMA descriptors. Those tasks are performed via a driver.

In an H2C transfer, the source address is a PCIe address and the destination is an AXI address. Whereas in a C2H transfer, the source is an AXI address and the destination a PCIe address.

Prior to any DMA transfer, the driver in the host has to be set up, which means being loaded and configured accordingly to how was the XDMA IP configured.

AXI-MM transfer for H2C

Figure 4.9 offers a detailed flow chart of an H2C transfer. It starts at the application level, issuing an H2C transfer (a write) to the driver. Then, the driver initiates the DMA transfer to the FPGA. Once the H2C channel has performed all the necessary steps, it sends an interrupt to the driver, which then notifies the user application that its transfer has finished.

AXI-MM transfer for C2H

Figure 4.10 shows a well-explained flow chart of a C2H transfer. It starts at the application level, issuing a C2H transfer (a read) to the driver. Then, the driver initiates the DMA transfer to the FPGA. Once the C2H channel has performed all the necessary steps, it sends an interrupt to the driver, which then notifies the user application that its transfer has finished.

4.4.3 Port description

Next, the interfaces and signals used in the FPGA@SDV design and relevant to this thesis are going to be described. Figure 4.11 shows the XDMA graphical representation in a BD from the FPGA@SDV Vivado project design.

Global signals

Table 4.2 contains the signals utilized in FPGA@SDV design.

Signal name	Type	Description
`sys_clk`	Input	Internal system clock
`sys_clk_gt`	Input	PCIe reference clock
`sys_rst_n`	Input	PCIe reset
`axi_aclk`	Output	PCIe output clock for AXI signals
`axi_aresetn`	Output	AXI reset signal syncronous to AXI clock

Table 4.2: Top-level signals name, type (input or output) and description. Source: [28].

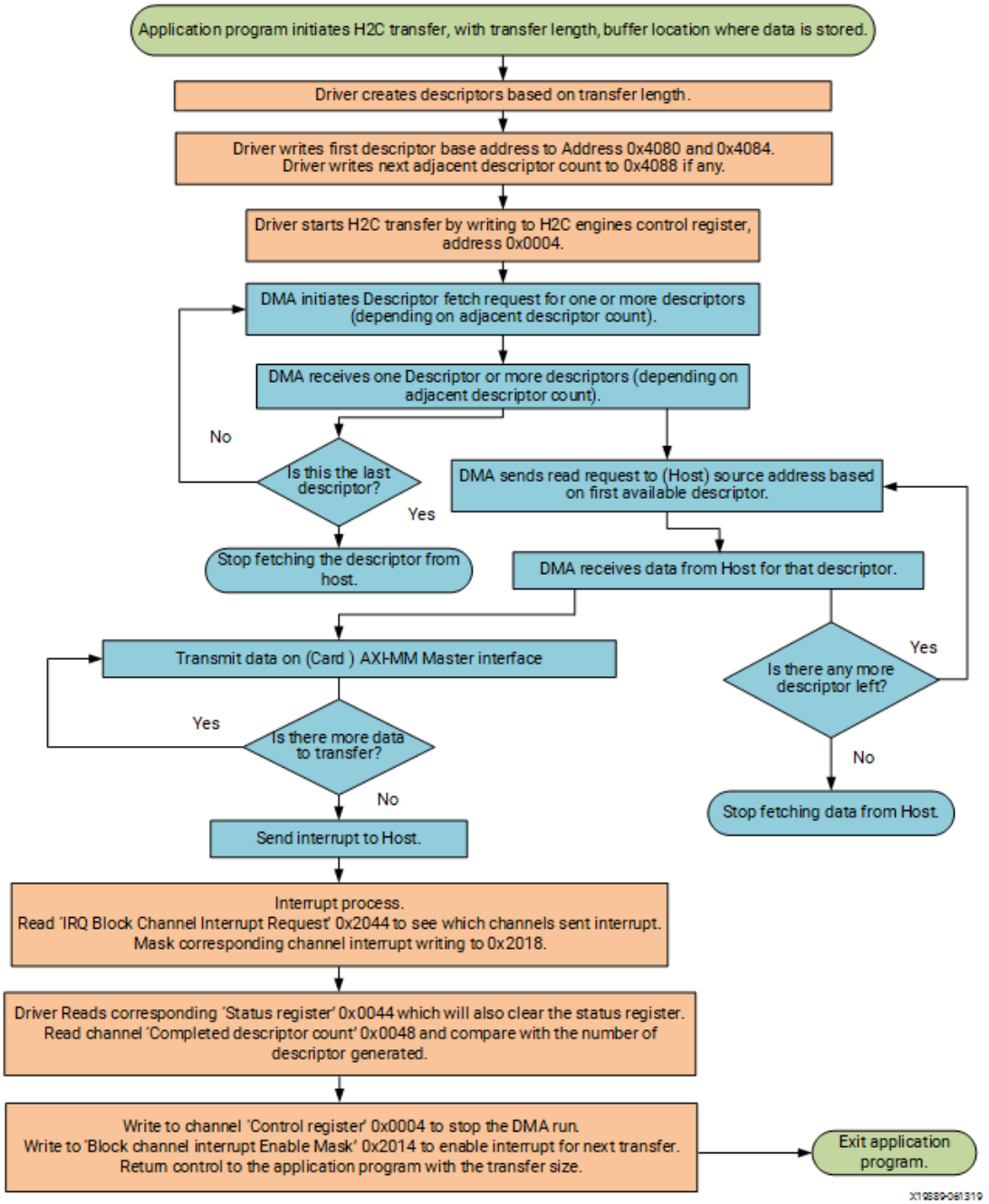


Figure 4.9: XDMA DMA H2C Transfer flow chart, where green is the application program, orange is the driver, and blue is the hardware. Source: [28].

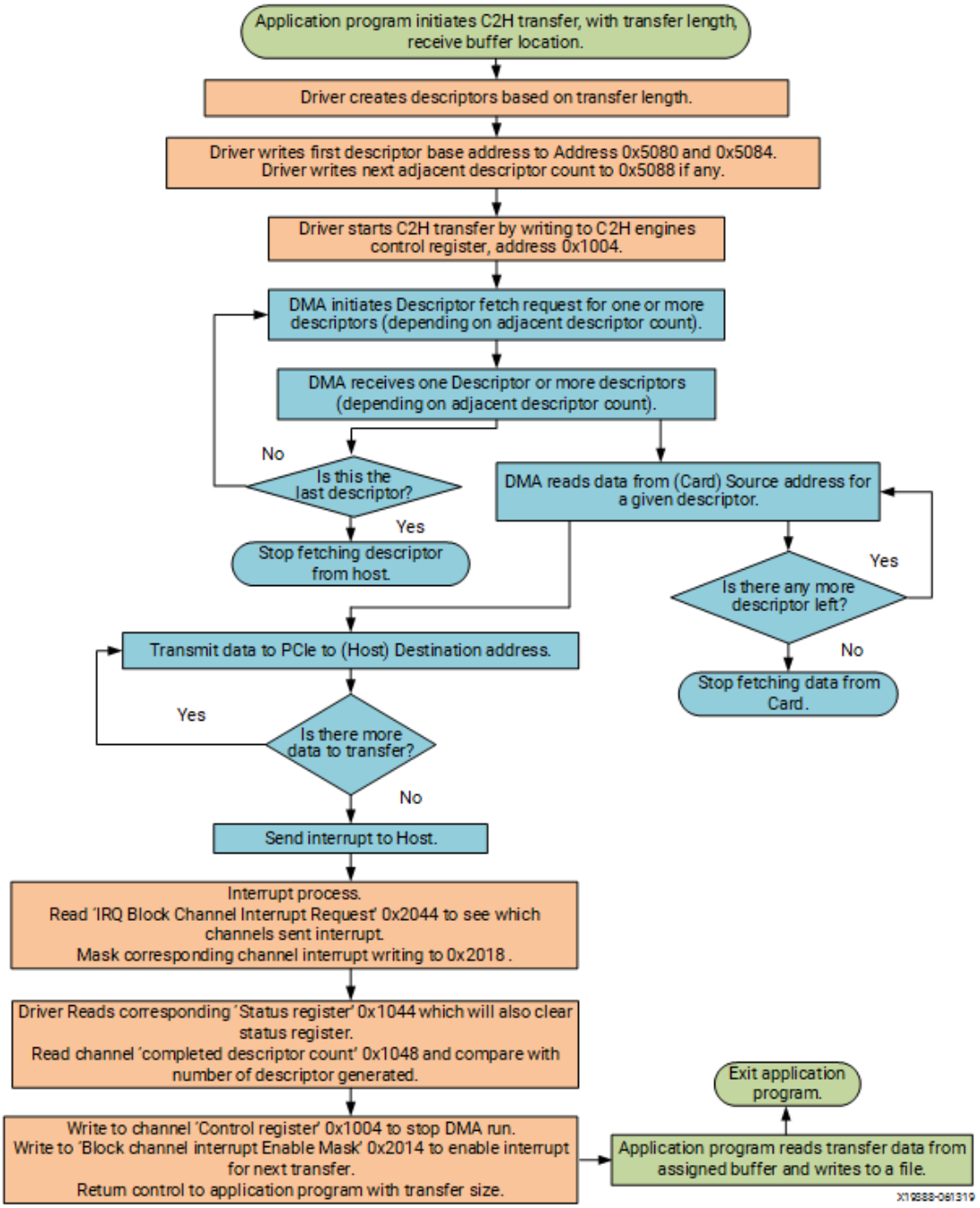


Figure 4.10: XDMA DMA C2H Transfer flow chart, where green is the application program, orange is the driver, and blue is the hardware. Source: [28].

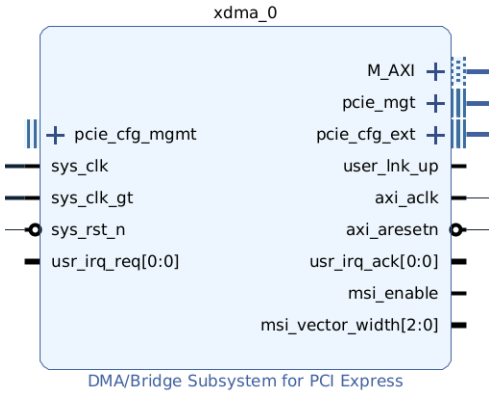


Figure 4.11: XDMA graphical representation in a BD from the Vivado GUI.

PCIe interface

Table 4.3 exhibits the signals used in FPGA@SDV design. All those signals are grouped in Figure 4.11 under the name interface `pcie_mgt`.

Signal name	Type	Description
`pci_exp_rxp`	Input	PCIe RX serial
`pci_exp_rxn`	Input	PCIe RX serial
`pci_exp_txp`	Output	PCIe TX serial
`pci_exp_txn`	Output	PCIe TX serial

Table 4.3: PCIe interface signals name, type (input or output) and description. Source: [28].

4.4.4 Driver

Xilinx provides a device driver for the XDMA IP that allows the host computer to access the FPGA and perform data transfers using the DMA engine. The driver runs at kernel space. Figure 4.12 represents the driver usage model in a Linux OS, which is the OS of the host CPU in Pickle nodes.

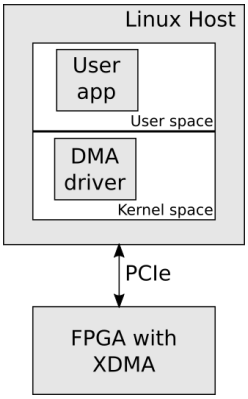


Figure 4.12: Linux kernel device driver usage model.

4.5 RDMA

The Remote Direct Memory Addressing (RDMA) protocol provides a DMA application to network protocols. It offers read and write services to user space applications and makes it possible to transfer data without intermediate data copies [29]. That allows data to be transferred between systems without involving the CPU, hence reducing the overhead and latency associated with traditional network protocols.

On the one hand, in traditional network protocols, such as TCP/IP, data must be copied multiple times between system memory and network buffers as it moves between layers. This copying process, handled by the CPU, can be time-consuming and introduce latency and performance overhead. On the other hand, in RDMA, the RDMA controller allows applications to directly access hardware and zero-copy data movement, which means that there is no CPU intervention.

RDMA is common in HPC, where fast and efficient data transfer is critical. For example, in many data centers, the principal interconnection protocol between nodes is InfiniBand[1], which uses RDMA technology, instead of Ethernet.

Although this protocol is network-oriented, Xilinx has developed a soft IP and device driver for FPGAs with PCIe that take advantage of some RDMA concepts.

4.5.1 Concepts

The most important concepts necessary for this thesis are the following:

- **Local Peer**
  The local entity, local end of the connection, in the description of a data transference between two nodes [29].
- **Remote Peer**
  The remote entity, opposite end of the connection, in the description of a data transference between two nodes [29].
- **Data sink**
  Peer receiving a data payload [29].
- **Data source**
  Peer sending a data payload [29].
- **RDMA Message**
  Data transfer technique to perform an RDMA Operation [29].
- **RDMA Operation**
  The sequence of RDMA messages needed to transfer data from a Data Source to a Data sink [29].

[1]NVIDIA InfiniBand https://www.nvidia.com/en-us/networking/products/infiniband/

- **RDMA Completion**
  Process of informing the user application that a certain RDMA operation has finished [29].
- **RDMA Write**
  RDMA operation that transfers data from the Local Peer (Data Source) to the Remote Peer (Data Sink) [29]. It is initiated by the Local Peer.
- **RDMA Read**
  RDMA operation that transfers data from the Remote Peer (Data Source) to the Local Peer (Data Sink) [29]. It is initiated by the Local Peer.
- **Queue**
  Basic logical element used to manage the different RDMA messages and events.

RDMA protocol provides access to different RDMA Operations, but the relevant ones are RDMA Write and RDMA Read.

4.5.2 Queues

A Queue Pair (QP) is a primary architectural element formed by two work queues: a Send work queue (outbound) and a Receive work queue (inbound) [30]. In other words, these two work queues establish a Queue Pair (QP). Each peer has its own QP, and data can be transferred in both directions simultaneously, allowing for bidirectional data transfers.

A connection is based on the bond between two RDMA peers with QPs, a local QP linked to a remote QP. A connection enables the exchange of RDMA operations.

The QP is identified by a QP number and it is independently configured, QPs are independent of each other.

4.5.3 RDMA Write operation

RDMA protocol provides the user application access to the RDMA Write operation.

In an RDMA Write, the Local Peer acts as Data Source by transferring data to the Remote Peer, the Data Sink.

Figure 4.13 illustrates the Message Sequence Chart (MSC) of an RDMA Write. The RDMA Write operation is formed by an RDMA Write Message, sent by the Local Peer to the Remote Peer. That message contains the data and where it has to be written in the Data Sink, among other information.

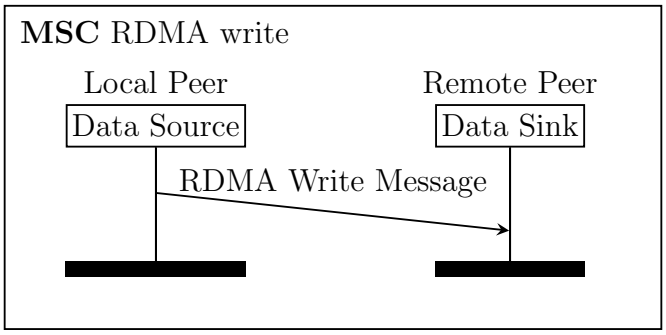


Figure 4.13: Message sequence diagram of an RDMA Write.

4.5.4 RDMA Read operation

RDMA protocol provides the user application access to the RDMA Write operation.

In an RDMA Read, the Local Peer acts as Data Sink by requesting data from the Remote Peer, which is the Data Source.

Figure 4.14 illustrates the MSC of an RDMA Read. The RDMA Read operation is formed by an RDMA Read Request, sent by the Local Peer, and an RDMA Read Response, sent by the Remote Peer.

The RDMA Read Request Message contains the Data Sink address from which the Remote Peer has to read and the Data Source address where data has to be transferred, among other information.

The RDMA Read Response Message holds the data from the Remote Peer, as part of other information.

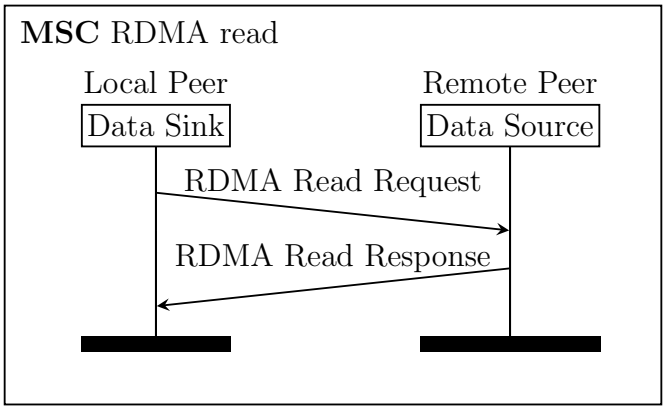


Figure 4.14: Message sequence diagram of an RDMA read.

4.6 QDMA Xilinx IP

The Queue DMA (QDMA) Xilinx IP is a PCIe DMA subsystem that implements the RDMA protocol with the concept of multiple queues, which is different than the XDMA

IP as that IP uses channels [31]. The QDMA IP provides a high-bandwidth and low-latency data transfer solution between the FPGA and other devices such as NICs, storage devices, and other PCIe-enabled devices.

QDMA is designed to move data between devices and memory by using a queue-based approach, an idea derived from the RDMA concept queue set. The main mechanism to perform transfers is through descriptors provided by the host OS. The data can be moved in the H2C direction (write) and the C2H direction (read).

The features that QDMA offers and important for this thesis are:

- Up to 2048 queues can be used.
- Support for both AXI-MM and AXI-ST interfaces per queue.
  Each queue can be configured individually by the driver to use AXI-MM or AXI-ST interface. In other words, the QDMA IP supports both AXI interfaces at the same time.
- Interrupts can be legacy, MSI, or MSI-X.
- Supports SR-IOV with up to 4 PFs and 252 VFs.

The possibility of assigning queues as resources to multiple PFs and VFs for a single QDMA block, hence a single FPGA board, allows different multifunction and virtualized application spaces.

The main limitation of the IP is that it supports a maximum of 256 queues on any VF.

Figure 4.15 shows the IP diagram. The arrows represent the bus connections between entities inside and outside the QDMA IP, which can be uni or bidirectional. Note that this diagram is flipped compared to the XDMA one (Figure 4.6), the user logic block is on the right and the physical PCIe connections are on the left. The QDMA IP is formed by the *Ultrascale+ PCIe Integrated block*, the rectangle on the left, and the light gray square, which has the QDMA engines. The focus will be on the blocks highlighted in blue: *descriptor engine*, *H2C MM engine* and *H2C AXI-MM* interface, *C2H MM engine* and *C2H AXI-MM* interface, and *IRQ module*. On the left, the two buses correspond to the physical PCIe connection *PCIe RX* and *PCIe TX*, the top and bottom arrows, respectively.

4.6.1 Architecture

At first sight, the diagram of the QDMA IP is more complex than the XDMA one. The H2C engines manage all the H2C queues, as the C2H engines do with the C2H queues.

Descriptor Engine

The descriptor engine fetches the H2C and C2H descriptors and maintains per-queue contexts with a series of pointers.

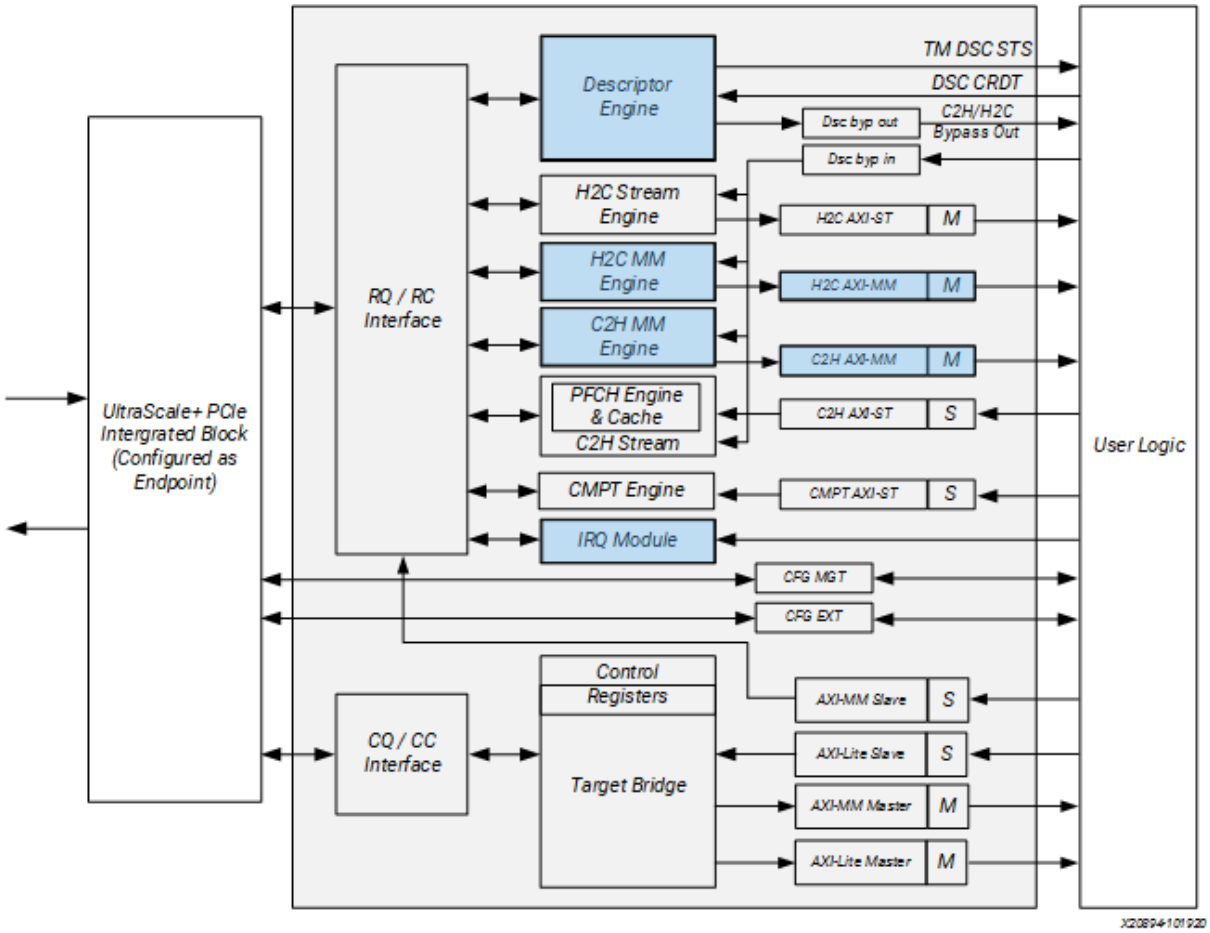


Figure 4.15: QDMA diagram with highlighted interfaces and blocks. Source: [31]

It has separate buffers for each queue type and can send them directly to the H2C and C2H engines, both Memory-Mapped (MM) and Stream ones.

H2C MM Engine

The H2C MM engine transfers data from host memory to FPGA memory via the H2C AXI-MM interface [31].

When it receives a descriptor, it is responsible for generating PCIe read requests to host memory. Once data is received through the completion of a PCIe read request, an AXI write is generated on the H2C AXI-MM interface with the content from the PCIe read.

C2H MM Engine

The C2H MM engine transfers data from FPGA memory to host memory via the C2H AXI-MM interface [31].

When it receives a descriptor, it is responsible for generating AXI read requests to FPGA memory on the C2H AXI-MM interface. Once data is received through the completion of an AXI read request, a PCIe write to the host is generated with the data from the previous AXI read.

Completion engine

The Completion (CMPT) engine is used to write to the completion queues. The completions are used by the driver to determine the number of bytes that were transferred [31]. It is mainly used along with the C2H Stream engine.

Interrupt Module

The Interrupt (IRQ) module aggregates interrupts from different sources, which are queue-based, user and error interrupts [31]. The queue-based interrupts include interrupts from H2C MM and C2H MM.

Depending if the SR-IOV is enabled or not, the available interrupt types changes. With SR-IOV not enabled, each PF can have either legacy or MSI-X interrupts. Whereas if SR-IOV is enabled, the only interruptions supported across all functions are MSI-X.

Queues design

The multi-queue PCIe subsystem uses the RDMA model queue pair. Each queue set is formed by H2C, C2H, and C2H Stream CMPT queues. The elements of each queue are descriptors [31].

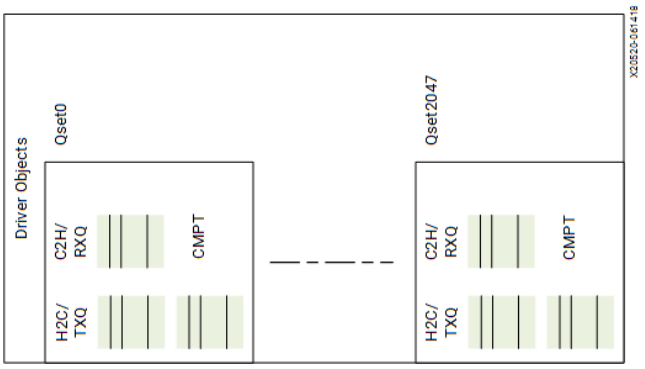


Figure 4.16: Queue ring architecture. Source: [31]

Those descriptors are written by the driver to H2C and C2H queues, and the engines read from those queues. Valid descriptors are advertised by writing their index to queues. Descriptors carry the host address, card address and length of DMA transfer.

Queues are rings located in host memory. A ring is a memory region dedicated to having certain data. Figure 4.16 illustrates a queue ring architecture in the host memory. For H2C and C2H queues, the producer is the driver and the consumer is the descriptor engine.

4.6.2 Operations

Descriptor fetch

The H2C and C2H fetch operation consists in the following steps:

1. The driver prepares the descriptor along with the payload buffer information for the H2C transaction or with reserved buffer space to receive the data for the C2H transaction. Then, it is placed in the corresponding queue with an associated producer index (producer memory address).
2. The driver sends the producer index to the descriptor engine.
3. The descriptor engine issues a DMA read request to gather the descriptor.
4. Once the descriptor engine receives the read completion from the host, meaning that all descriptors information has been delivered, the engine delivers them to the corresponding H2C engine or C2H engine.

Figure 4.17 illustrates the fetch operation, where the numbers correspond to the previous explanation.

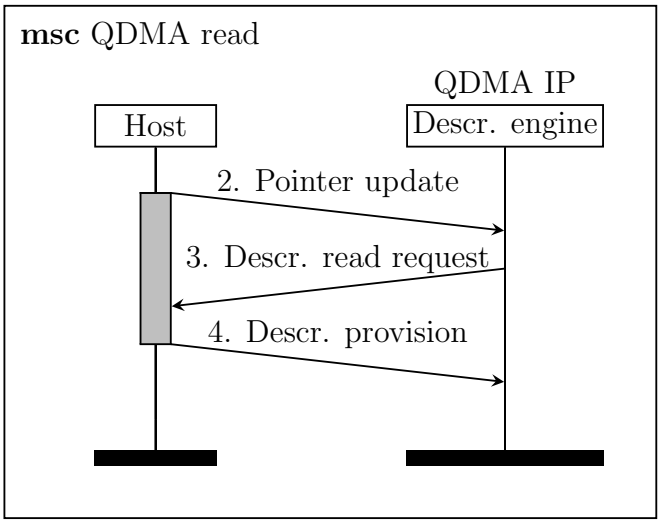


Figure 4.17: Message sequence diagram of a QDMA descriptor fetch.

Memory Mapped DMA

Memory-mapped DMA operations have source and destination memory-mapped spaces. For the H2C transfer, the source address is from the PCIe address space and the destination address belongs to the AXI-MM address space; and vice versa for the C2H transfer. Both transfers have similar behavior and share the descriptor format [31].

The operation for either H2C or C2H transfers follows the same general procedure:

1. Fetch descriptors as detailed previously, in Section 4.6.2.
2. Generate a read request to the source interface to get the data.
   - **H2C**: through PCIe to reach the host memory.
   - **C2H**: through C2H AXI-MM interface to reach the FPGA memory.
3. Write data in the destination interface.
   - **H2C**: to H2C AXI-MM interface to reach the FPGA memory.
   - **C2H**: to PCIe to reach the host memory.
4. Receive the write completion from the destination to the source.
   - **H2C**: an AXI `bresp` from H2C AXI-MM interface to H2C MM engine.
   - **C2H**: a PCIe write request accepted from the host to the C2H MM engine.
5. Once the completion criterion is met, send an interrupt.

Figures 4.18 and 4.19 ilustrate the previous process for H2C transfers and C2H transfers, respectively. The last step, sending an interrupt, is not included in the diagram because it is dependent on the type of interrupt used.

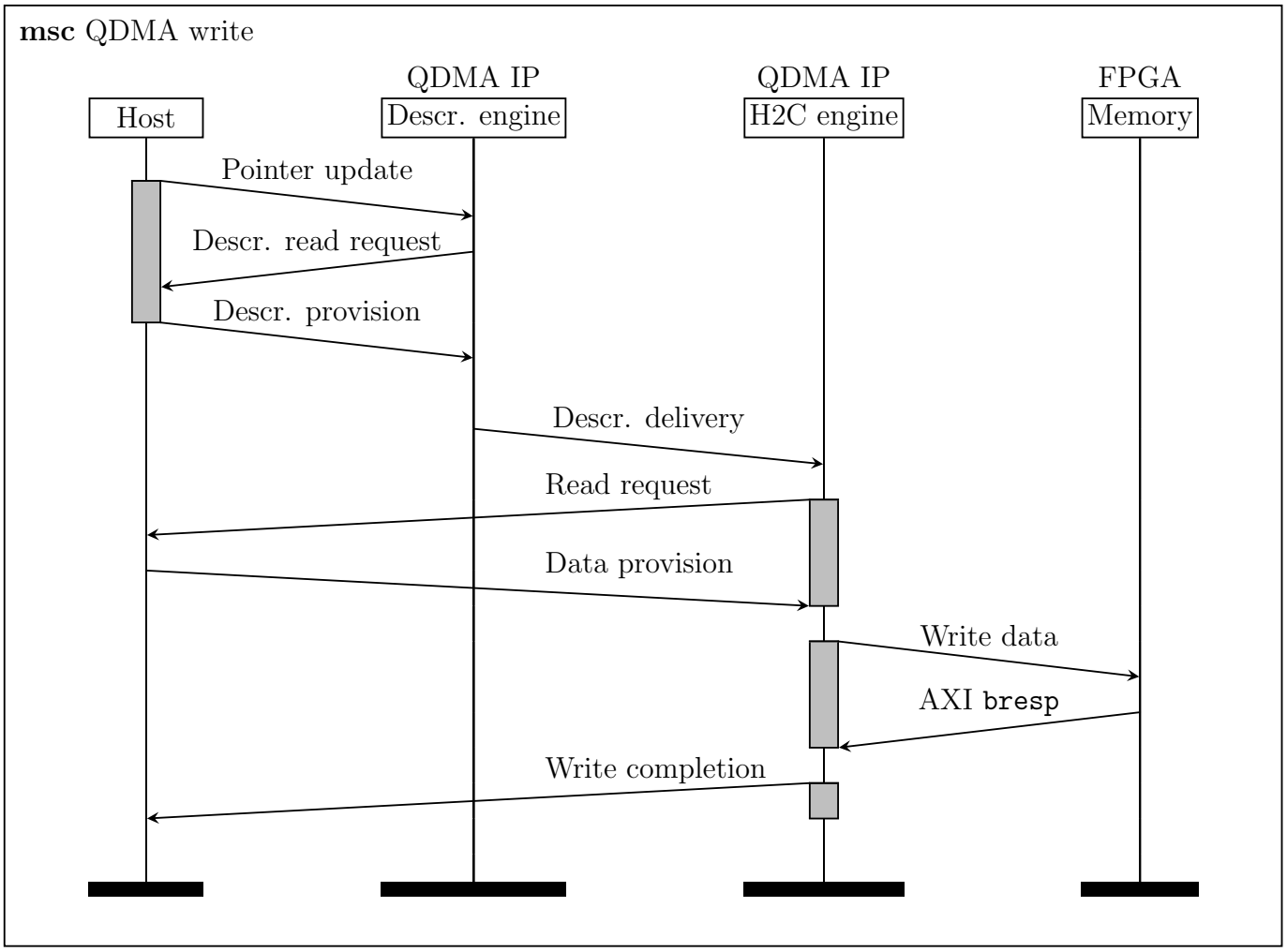


Figure 4.18: Message sequence diagram of a QDMA write (H2C tranfer).

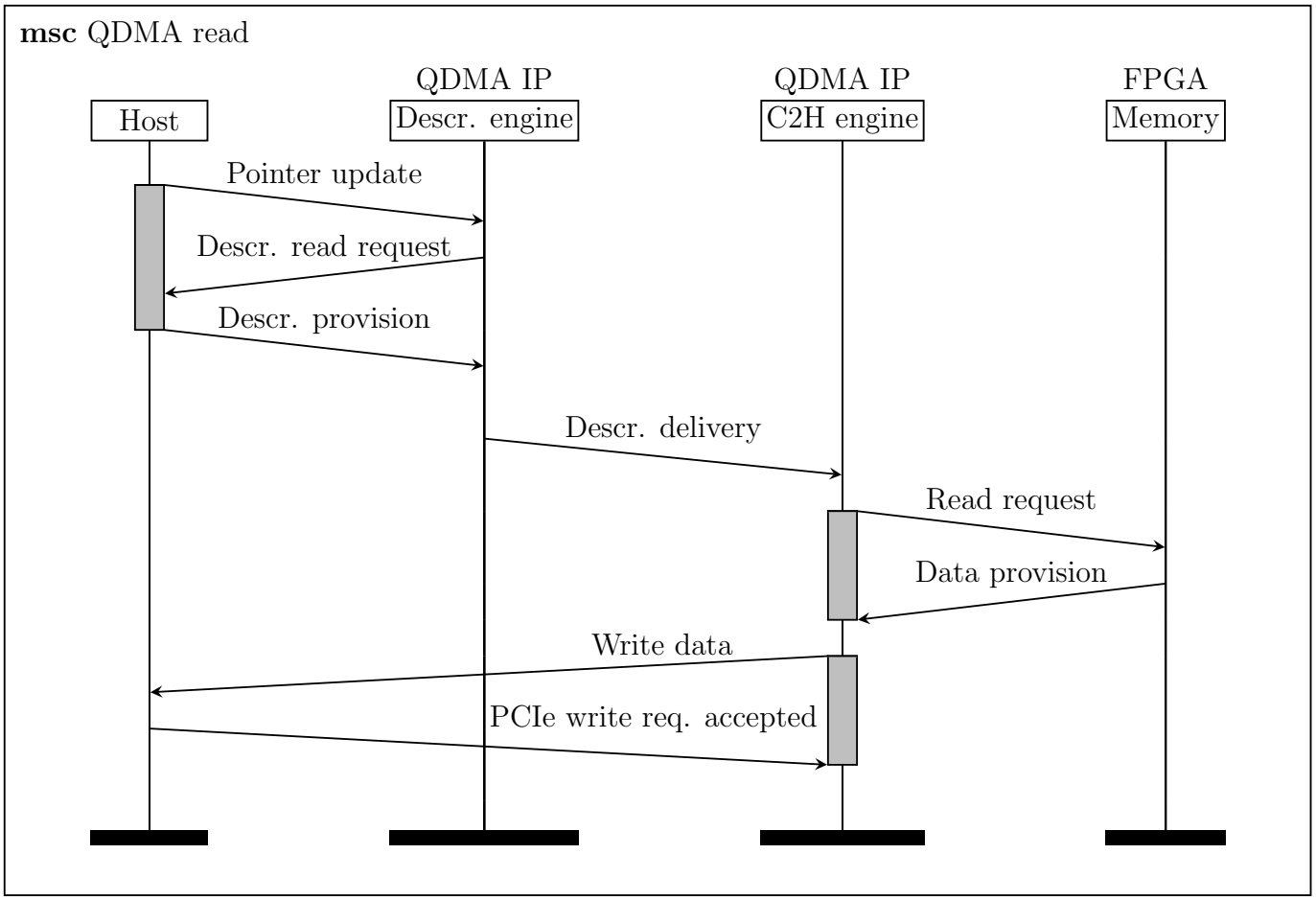


Figure 4.19: Message sequence diagram of a QDMA read (C2h transfer).

4.6.3 Port description

The relevant ports and descriptions in QDMA IP are the same as for XDMA, which are explained in Section 4.4.3.

4.6.4 Driver

As for the XDMA IP, Xilinx provides a software driver for the QDMA IP that allows the host to access the FPGA and perform DMA transfers. It also runs in kernel space and Figure 4.12 also applies to the QDMA driver.

The QDMA driver package is formed of 3 main components [31]:

- Device driver: Creates the descriptors and translates the user space functions into low-level commands in order to interact with the QDMA device.
- Device control tools: Creates the QDMA queues, manages them, and other functions.

- DMA tools: User space applications to perform example DMA transfers.

Device driver

Xilinx provides 2 different drivers depending on the function type: PF driver for Physical Function and VF driver for Virtual Function [32]. The user has to decide which driver to use based on how the QDMA IP was configured.

PF and VF drivers can be inserted in different modes. The module (driver) parameter `mode` specifies how the completions must be processed:

- **Poll Mode**
  Driver polls on the status descriptor write-back for completions.
- **Direct Interrupt Mode**
  A single interrupt vector is assigned to each queue. An interrupt is raised by the hardware (FPGA) upon receiving the completions and the driver reads the completion status.
- **Indirect Interrupt Mode**
  Each vector has an associated Interrupt Aggregation Ring. When a PCIe MSI-X interrupt is received by the driver, it reads the Interrupt Aggregation Ring to determine which queue needs service.
- **Legacy Interrupt Mode**
  Driver processes the status descriptor write-back using legacy interrupts.

Device control tool

The so-called `dma-ctl` tool is an application that provides a set of commands to configure and manage QDMA queues in the system. It runs in user space.

It offers a collection of functions:

- Show the list of PCIe functions bonded to the QDMA driver. Hence, the functions that have a QDMA IP.
- Query queue control and configuration: list queues, add/configure new queues on a device, and start, stop or delete them.
- Access to registers: read and write a register, and dumb the QDMA configuration registers.
- Display queues' parameters and entries from different rings.

Chapter 5

State-of-the-art

This chapter is dedicated to analyzing the state-of-the-art of various RISC-V implementations for FPGAs and evaluating the status of the FPGA@SDV design at the beginning of this thesis.

5.1 RISC-V implementations in FPGA

As RISC-V is an open ISA it has allowed many companies to develop processors without having to conceive an ISA from scratch or paying royalties (such as ARM ISA). For this reason, there are different RISC-V implementations in the form of ASICs or FPGA's soft-cores. On the one hand, ASIC-based designs implement System-on-Chips (SoCs) composed of the processor, a cache hierarchy and several peripherals such as Ethernet and PCIe. On the other hand, soft-cores are soft IP cores developed to be used in FPGAs, so they are written in an HDL and are synthesizable. Moreover, more than one soft-core can be instantiated in an FPGA, which allows the creation of multi-core systems.

Due to the scope of this thesis, which is centered on FPGAs, commercial RISC-V soft-cores have been studied.

5.1.1 Rocket Chip

Rocket chip is an open-source SoC generator that creates RTL code. It allows instantiating a general-purpose RISC-V CPU, which can be Rocket or BOOM [33]. It is developed by the University of California, Berkeley.

It is written using Chisel, which is an HDL embedded in Scala. Chisel allows designers to describe and generate hardware designs using a high-level programming language.

BOOM

The Berkeley Out-of-Order Machine (BOOM) is an open-source soft-core implementation of the RISC-V ISA [34]. It is based on the RISC-V RV64GC variant, also known as

RV64IMAFDC: **R**ISC-**V** **64**-bit **I**nteger, **M**ultiply and divide, **A**tomic, **F**loating point single and **D**ouble precision and **C**ompressed instruction set support. The BOOM soft-core was developed at the University of California, Berkeley.

The implementation of BOOM is written using Chisel.

BOOM is designed to be synthesizable, meaning that it can be transformed into an HDL representation and synthesized into an actual hardware design. It is also parameterizable, enabling customization of various design aspects such as cache size, issue width, and other performance-related parameters.

BOOM does not provide built-in support for peripherals. It focuses primarily on the CPU core itself and its associated functionality. Peripheral support, such as input/output interfaces, external memory controllers, and other hardware components, would need to be added separately to create a complete system using BOOM.

The core is capable of successfully booting the Linux OS. This implies that it satisfies the requirements and supports the necessary functionalities to initiate and run the Linux kernel, enabling the execution of Linux-based applications and software on the BOOM soft-core.

Rocket

Rocket is an in-order soft-core that implements RV32G and RV64G and it is developed by Berkeley. It is also written in Chisel.

Rocket is synthesizable, as well as BOOM, and is compatible with FPGA designs through the Rocket chip integrator. It is capable of booting Linux.

FPGA integration

Rocket chip offers support to some FPGA boards. The Xilinx boards supported are: Arty FPGA Evaluation Kit, VC707 FPGA Evaluation Kit and VCU118 Evaluation Kit. Their FPGA shell includes JTAG, UART and **XDMA**.

5.1.2 Nios V processor

The Nios V processor is a RISC-V soft processor core developed by Intel. It is specifically designed to be used with Intel FPGA devices.

The Nios V processor has two distinct variants available. The Nios V/m variant is designed as a microcontroller and implements RV32IA. The 32I extension stands for 32-bit Integer. The Nios V/g variant is a general-purpose processor that implements RV32IMA extensions.

To integrate the Nios V processor into an FPGA design, it can be instantiated using Intel Quartus. Intel Quartus is a software development tool provided by Intel that enables designers to create, configure, and program FPGA designs, like Vivado.

The Nios V processor core itself does not include any built-in peripheral interfaces or connection logic to external devices. It focuses primarily on the CPU functionality. To interact with peripherals and external devices, additional logic and interfaces need to be added separately to the FPGA design, connecting them to the Nios V core.

The Nios V processor does not have native support for booting the Linux operating system.

5.1.3 PULP platform

Parallel Ultra-Low-Power (PULP) is an open-source platform developed by ETH Zurich and the University of Bologna. Its architecture includes a RISC-V core as the main core and support to different IO peripherals. The two possible RISC-V cores are RI5CY and zero-riscy [35].

RI5CY is an in-order core that implements RV32IMCF RISC-V extensions and it has been designed to target ultra-low-power constraints. Whereas the other core, zero-riscy, is also an in-order core, but it implements RV32IMVE (E stands for the reduced number of registers extension) and it targets ultra-low-power and ultra-low-area constraints. RI5CY is developed in System Verilog, an HDL, and it does not boot Linux.

As it was mentioned before, the PULP platform offers support to IO peripherals, which means that it does not instantiate those IPs, but it provides interfaces to those peripherals so the users can connect FPGA-dependant IPs.

5.1.4 Ariane

CVA6, formerly known as Ariane, is an in-order RISC-V core that implements RV64IAMC and it is implemented by OpenHW Group [36]. It is written in System Verilog.

Ariane can boot Linux as it implements three privilege levels to fully support a Unix-like OS.

It is parametrized and offers a different separated FPGA emulation platform, although it only provides support for the Xilinx Genesys 2 board. That FPGA emulation platform is called CVA6 APU and it offers support to differnet interfaces such as UART and Ethernet.

5.2 Initial state of SDV design

In the SDV design for FPGAs, also known as FPGA@SDV, the PCIe subsystem used is XDMA. The FPGAs supported initially with the FPGA@SDV design are shown in Table 5.1.

FPGA model	PCIe Subsystem
Xilinx VCU128	XDMA
Xilinx U55C	XDMA

Table 5.1: FPGA supported with the EPAC design.

5.3 Comparison

Rocket chip is the only open-source RISC-V implementation studied that provides PCIe support, specifically the XDMA IP. The developers do not provide information about the PCIe performance and synthesizing one of their cores and FPGA shells would not be feasible because non of their supported FPGA boards are available in our systems and only adapting the shell to the Xilinx VCU128 would be by itself a three-month task. Therefore, comparing FPGA@SDV againts other RISC-V soft-cores is a complex task that cannot be performed due to the thesis time limit.

Chapter 6

Replacement of XDMA with QDMA

After studying and understanding all the concepts from the previous chapters, the technical work could be started.

This chapter is focused on the development performed in order to achieve a functional SDV design in the VCU128 using the QDMA PCIe subsystem, instead of the XDMA one. The final goal is to be able to boot the Linux image with QDMA keeping the same functionalities and same tools as with XDMA, and to evaluate both IPs.

6.1 Example design

Xilinx provides example designs for some of their IPs. An example design is a Vivado project with all the necessary sources to be able to test the IP's functionality.

An example design for the QDMA IP can be generated, as the QDMA documentation stated [31]. First of all, a QDMA IP was instantiated in a new Vivado project, in a BD. Then, that IP was configured with the same basic PCIe parameters as in XDMA. That is because the documentation states that the example design can be generated and adapted to the parameters set in the IP [31].

After that, the configured IP example design was produced and studied. The example design is written in Verilog and SystemVerilog, two HDL languages. It contains a wrapper for the QDMA IP and a *QDMA app*. The *QDMA app* is an RTL code that acts as the user application.

The interesting part and motivation to investigate the example design was to observe the simulation behavior of the IP. It would be useful being able to compare the expecte QDMA behaviour in simulation agains the QDMA in the SDV design, if it does not work as expected.

Unfortunately, running the simulation was not possible. When launching it in Vivado, it reported the error shown in Listing 6.1. The first error redirected to a log file, which was consulted and the error in Listing 6.2 was reported. The error reports that signal `s_axil_araddr` is not declared before using it as input for another module. That error had an easy fix, but before performing it, I realized that there were more signals undeclared,

and, more importantly, that those input signals were not being generated anywhere in the code. No module had those signals as output, and as I was not the designer of that RTL code, it was not straightforward knowing where they had to be generated.

```
1 ERROR: [USF-XSim-62] 'compile' step failed with error(s). Please check
     the Tcl console output or '/home/aquerol/tfm/qdma_example/qdma_0_ex/
     qdma_0_ex.sim/sim_1/behav/xsim/xvlog.log' file for more information.
2 ERROR: [Vivado 12-4473] Detected error while running simulation. Please
     correct the issue and retry this operation.
3 ERROR: [Common 17-39] 'launch_simulation' failed due to earlier errors.
```

Listing 6.1: Errors reported by Vivado lauching the example design simulation.

```
1 ERROR: [VRFC 10-2989] 's_axil_araddr' is not declared [/home/aquerol/tfm
     /qdma_example/qdma_0_ex/imports/qdma_app.sv:456]
2 ERROR: [VRFC 10-8530] module 'qdma_app' is ignored due to previous
     errors [/home/aquerol/tfm/qdma_example/qdma_0_ex/imports/qdma_app.sv
     :58]
```

Listing 6.2: Errors reported in the log file when running the example desing simulation.

After spending some time on it, it became clear that keep working with a non-functional example design was not worth the time. Hence, it was decided to **perform the replacement of IPs in the SDV design directly**.

6.2 Replacement in SDV design

After deciding to continue directly with the SDV design, the following steps were foreseen:

- In the hardware side:
  1. Replace the XDMA IP in the BD with the QDMA IP.
  2. Adapt the design constraints.
  3. Generate the *QDMA bitstream*.
- In the software side:
  1. Compile the QDMA driver and tools provided by Xilinx.
  2. Load the driver with the proper parameters.
  3. Configure a queue.
  4. Execute the data word test (explained in Section 6.2.8).
  5. Adapt the current EPAC tools and check that the EPAC behavior remains unchanged.
- Run the performance tests and extract different metrics.

Sections below do not strictly follow the previous steps, since technical difficulties arose during the developement of those.

6.2.1 Change of IPs

IP settings

The first step was changing the XDMA IP with the QDMA IP in the BD. To do so, a QDMA IP was instantiated in the BD. Then, each QDMA IP tab was configured, keeping the common settings with the same values and adjusting the remaining ones as close as possible to the previous XDMA configuration or in the most optimal manner possible for the project.

The following figures contain the XDMA configuration in the firsts subfigures and the new QDMA configuration in the last subfigure, because some XDMA configuration tabs are merged into a unique QDMA IP. Each figure groupes similar settings in both IPs. The relevant settings for this thesis will be explained below.

Figure 6.1 shows the configuration from the Basic tab, which has almost the same parameters in both IPs. The QDMA configuration, Figure 6.1c, has the additional setting *Number of Queues* that was set with the minimum number of queues available (512), because only one queue was necessary for the SDV tools. The other QDMA parameters were defined with the same XDMA values. The *PCIe Block Location* affects the placement of the soft IP in the FPGA, the *Lane width* is set to 8x (8 lanes), and the *Maximum Link speed* sets the PCIe generation, hence the 5.0 GT/s refers to PCIe Gen 2.

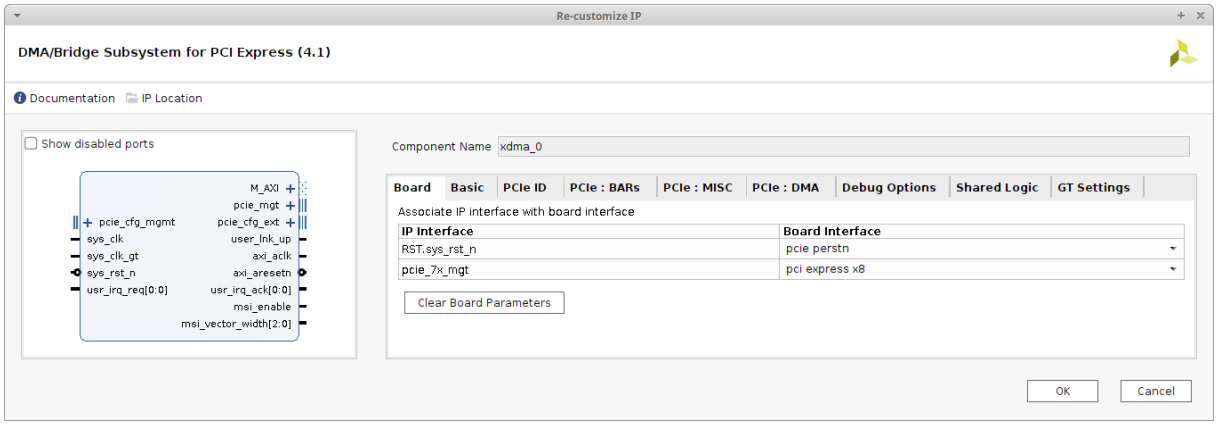
The tabs shown in Figure 6.2 contain the PCIe identification, which has the default value set, so the driver does not have to be changed. Figure 6.2b shows additional settings related to SR-IOV and the number of PFs. SR-IOV has not been activated as Pickle nodes do not have virtualization, therefore the number of PFs required is only 1.

The next tab manages the output AXI buses, in Figure 6.3. The only AXI interface enabled is the DMA one. Figure 6.3b shows that the QDMA IP offers more configurable parameters for this interface, which have been set with the default values.

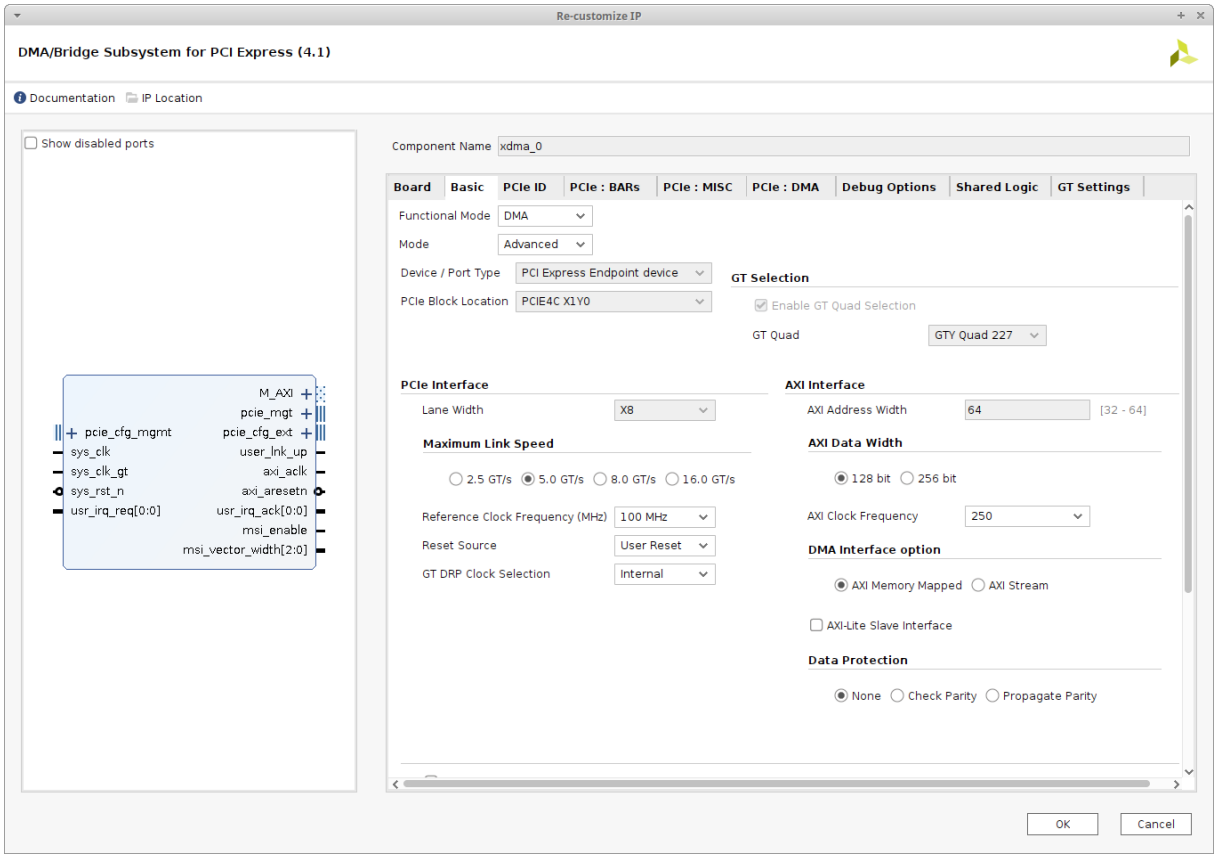
The MISC tab in Figure 6.4 has the interrupt configuration. The XDMA IP offers more settings for the legacy and MSI interrupts than for the MSI-X ones, whereas the QDMA settings are the other way around: more configuration options for MSI-X interrupts than the other ones. Legacy interrupts are enabled in the QDMA IP.

Figure 6.5 contains the DMA settings, which are different for each IP. The first two options from the XDMA IP, Figure 6.5a, indicate that 4 H2C and C2H channels are being used. This option would correspond in QDMA to the *Number of Queues* set in the Basic tab. In addition, the XDMA descriptor bypass is set per channel, whereas the QDMA descriptor bypass is set globally. In both cases, it is deactivated, so the IP is the one that manages the descriptors.

The remaining XDMA tabs (Figures 6.6a to 6.6c) are all grouped in the same QDMA tab (Figure 6.6d).

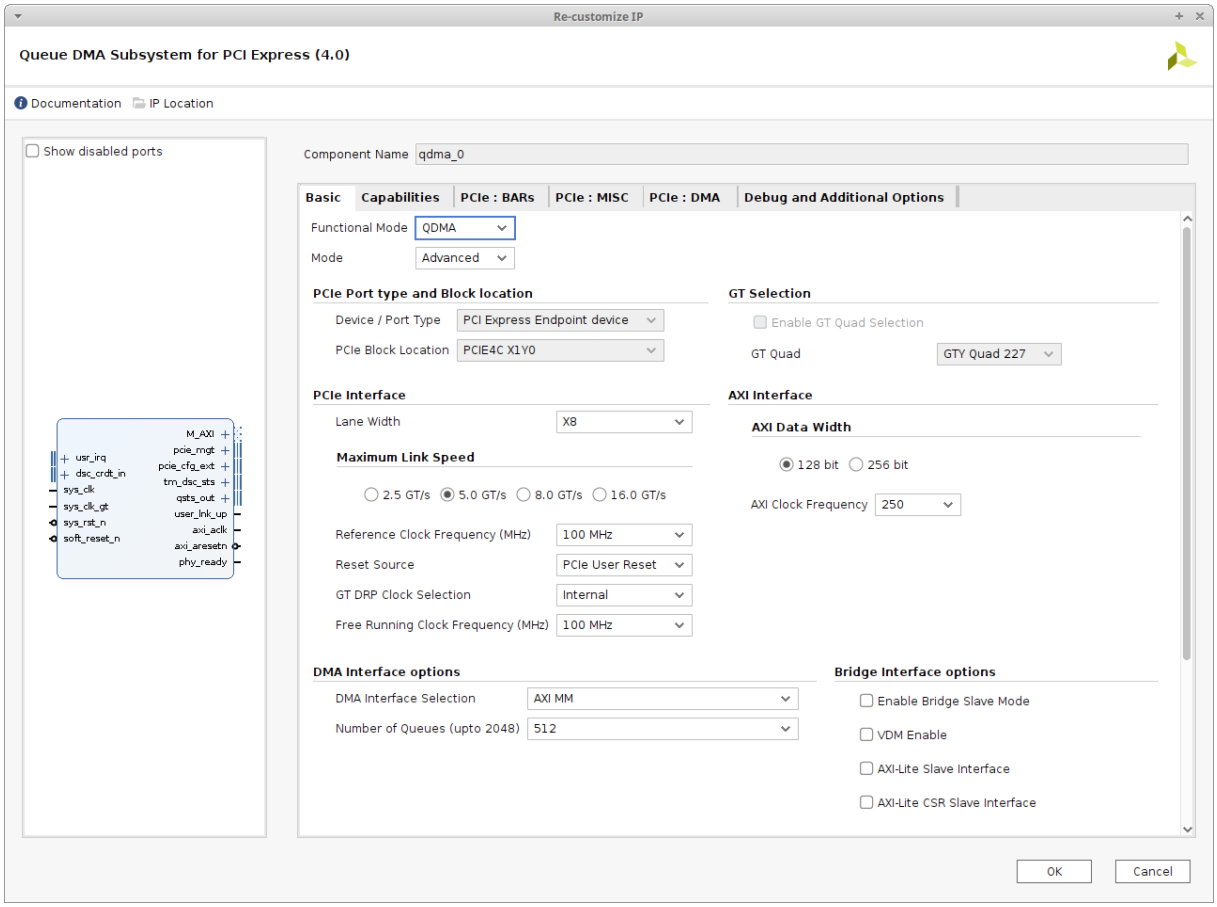


(a) XDMA



(b) XDMA

Figure 6.1: Basic tab



(c) QDMA

Figure 6.1: Basic tab

Block Design connection

The next step was connecting the input and output signals from XDMA IP to the QDMA IP, as Figure 6.7 illustrates. Moreover, the QDMA IP had to be connected to the board PCIe hard IP.

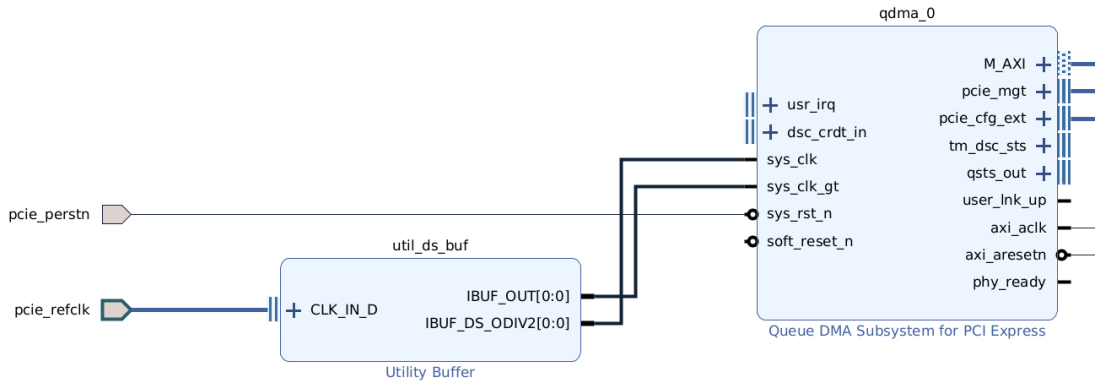
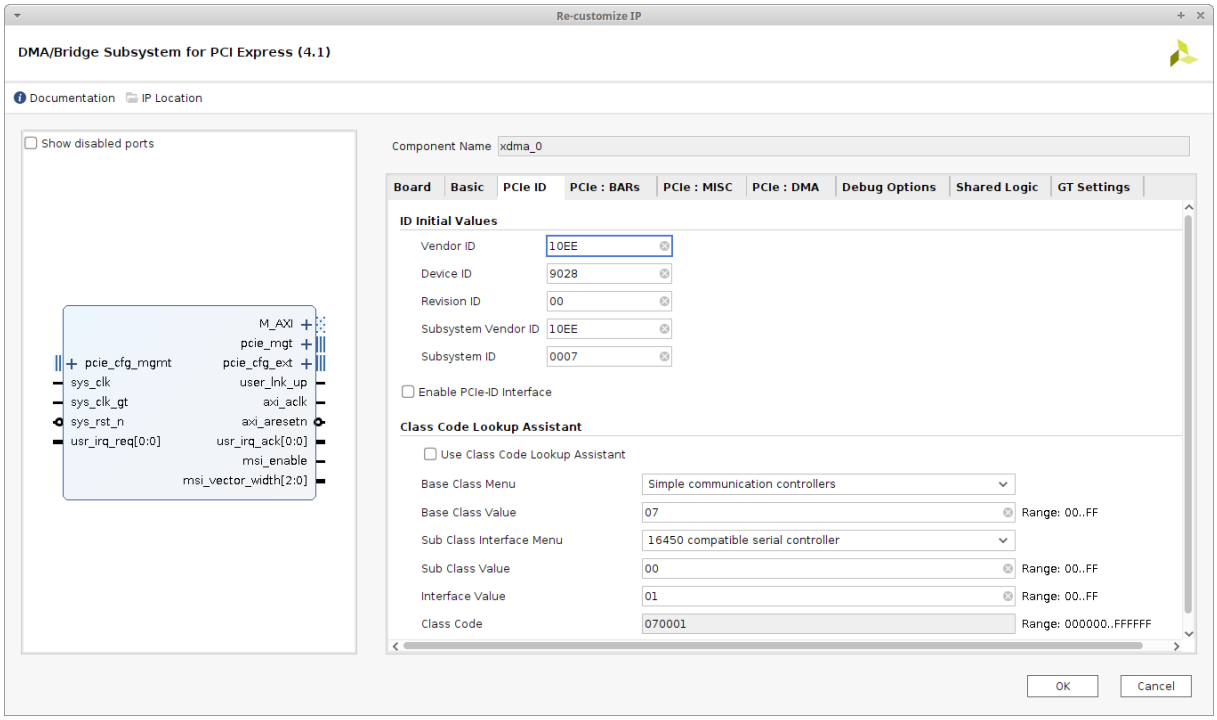
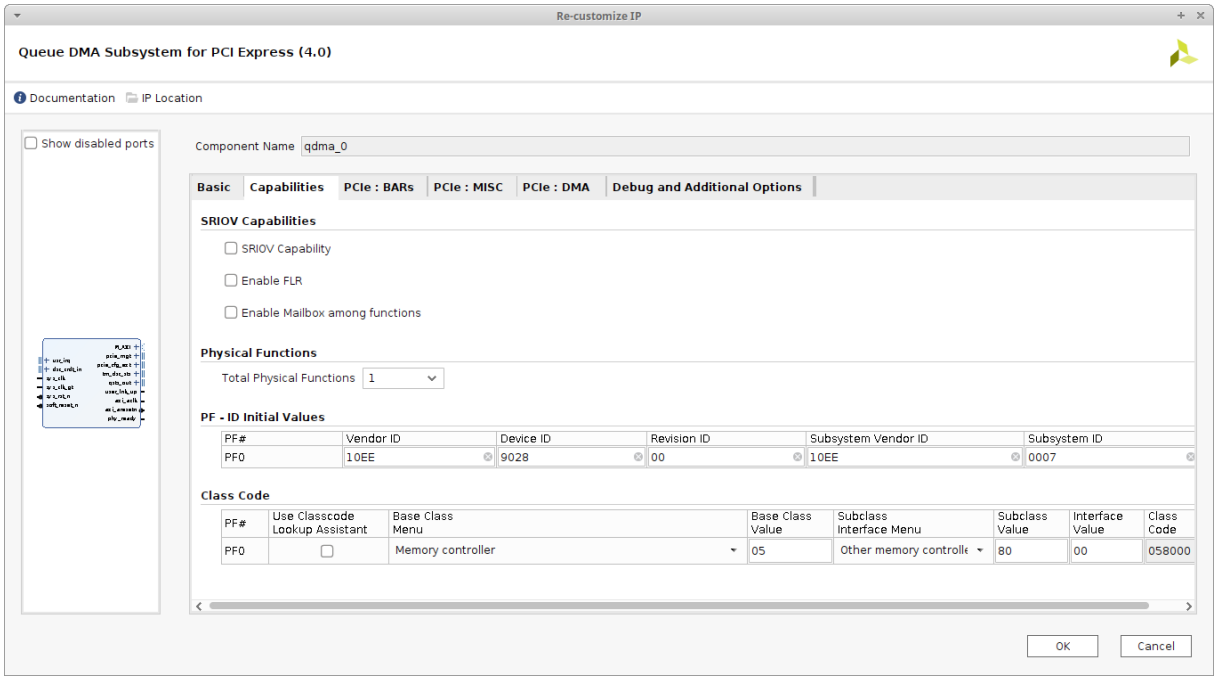


Figure 6.7: BD with the pins and connections for the QDMA IP.

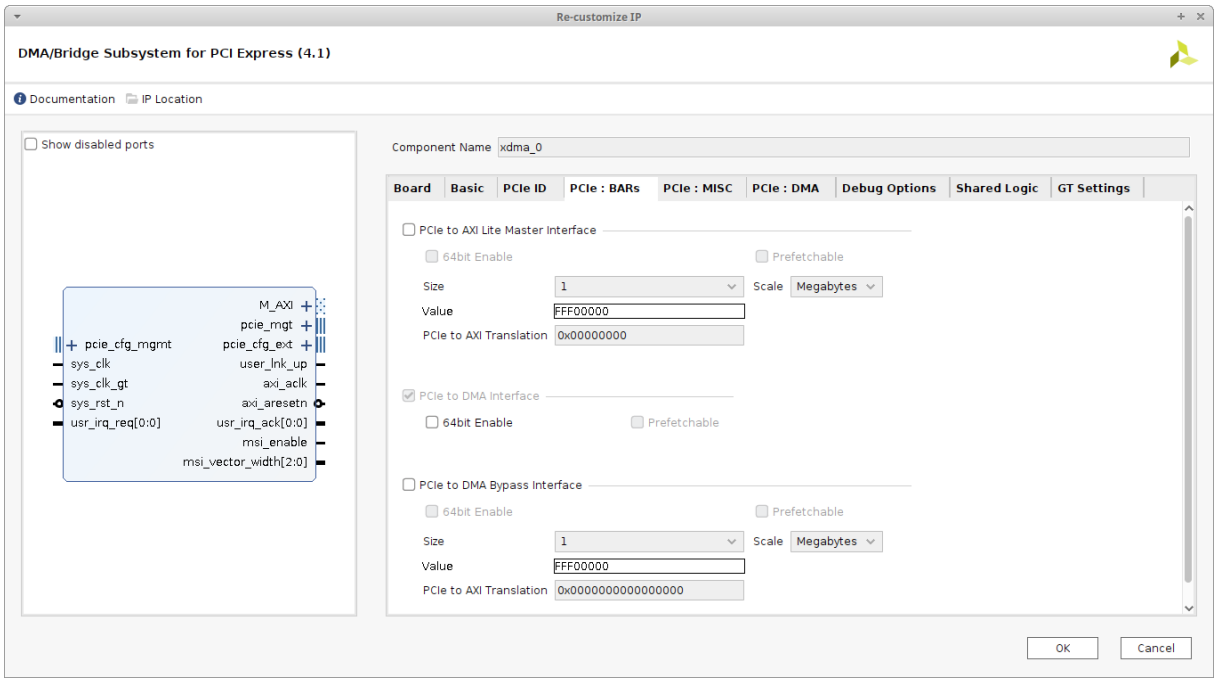


(a) XDMA

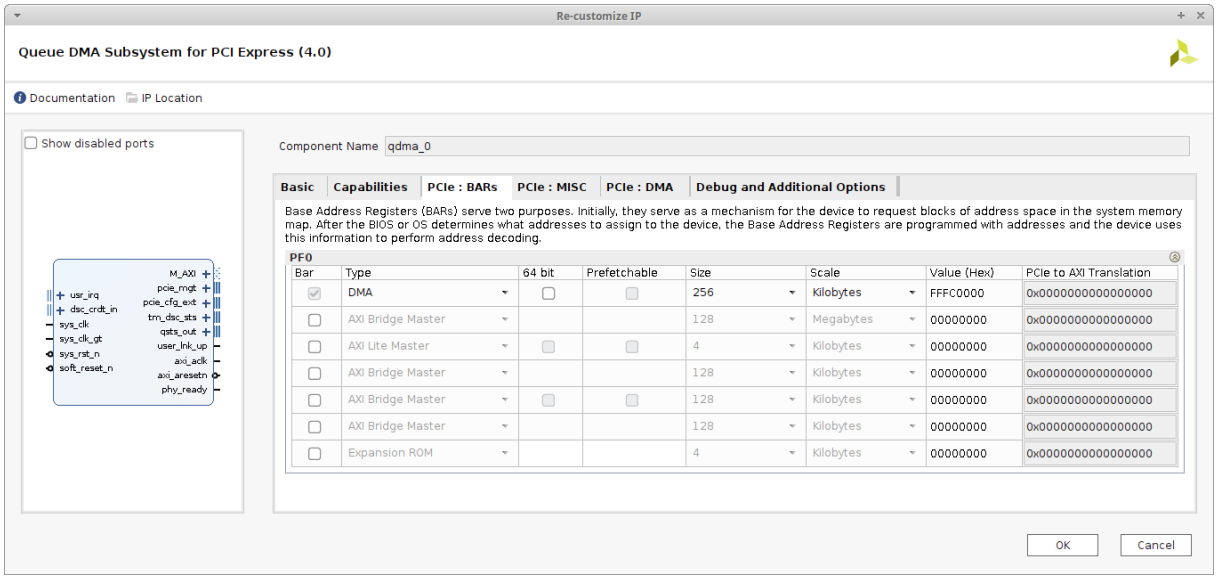


(b) QDMA

Figure 6.2: PCIe ID and Capabilities tab

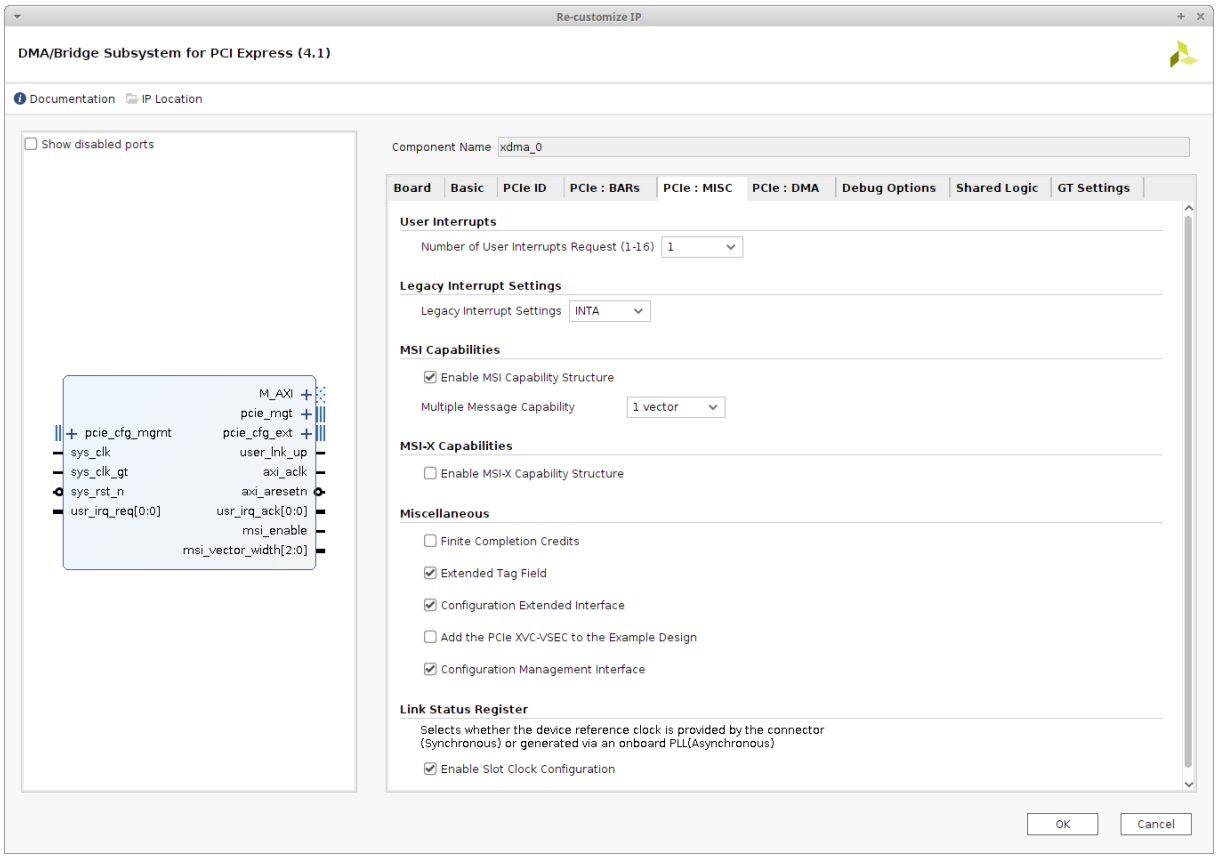


(a) XDMA

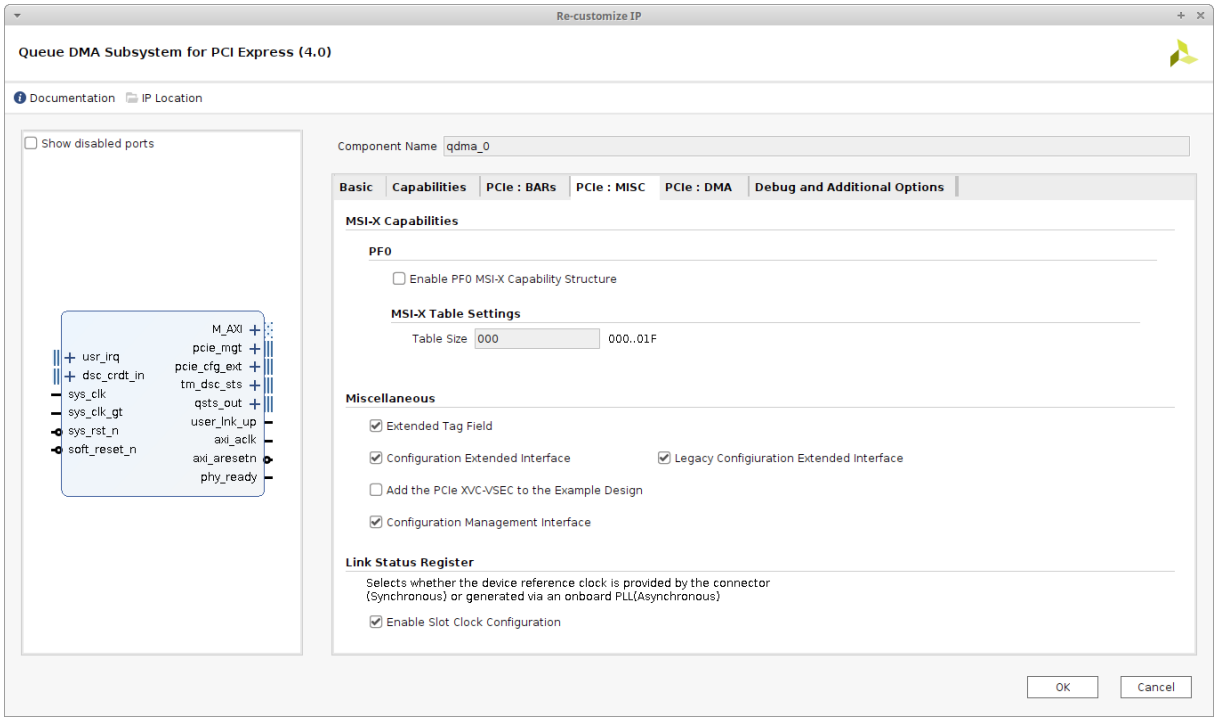


(b) QDMA

Figure 6.3: Bars tab

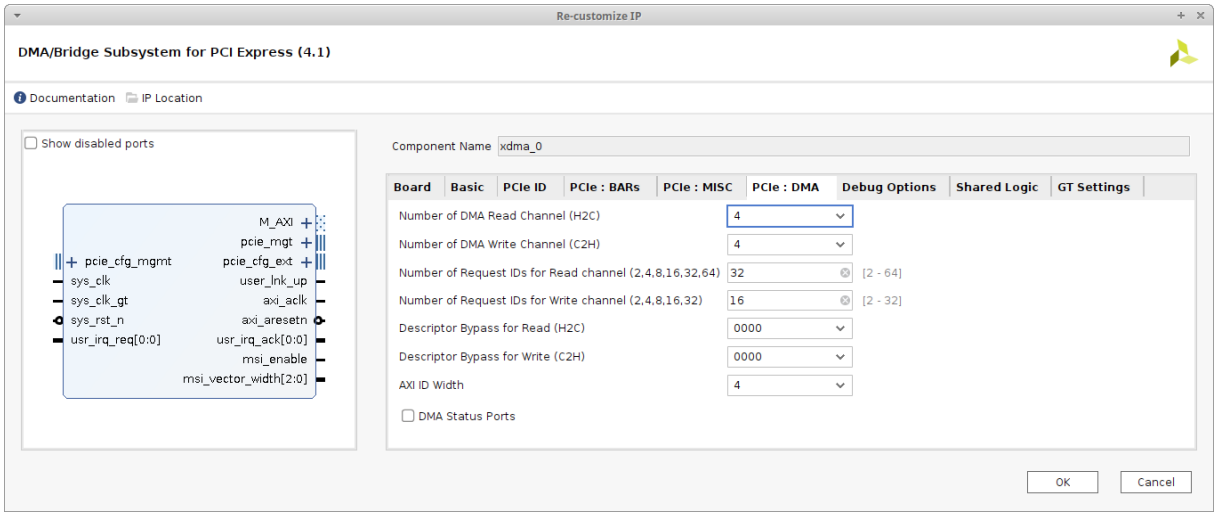


(a) XDMA

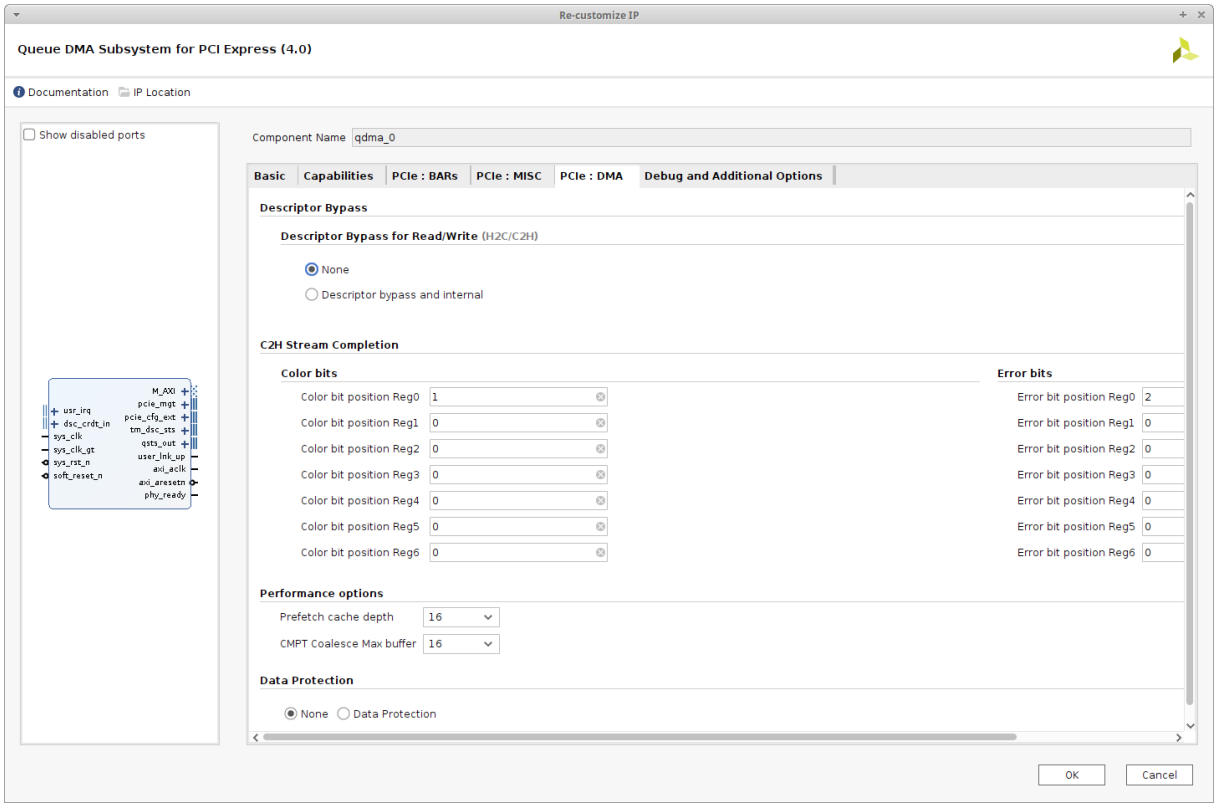


(b) QDMA

Figure 6.4: MISC tab

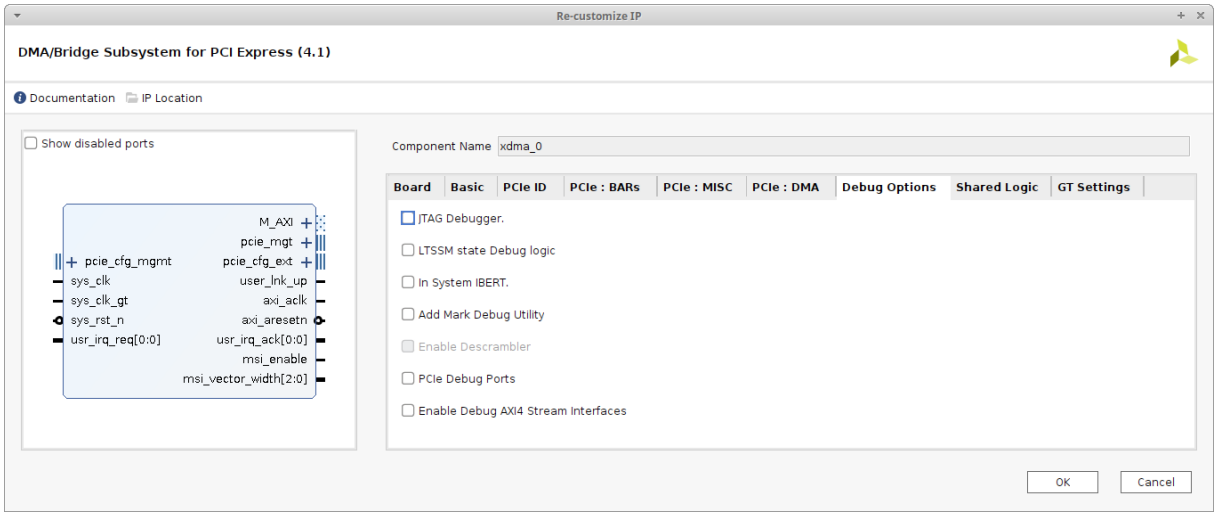


(a) XDMA

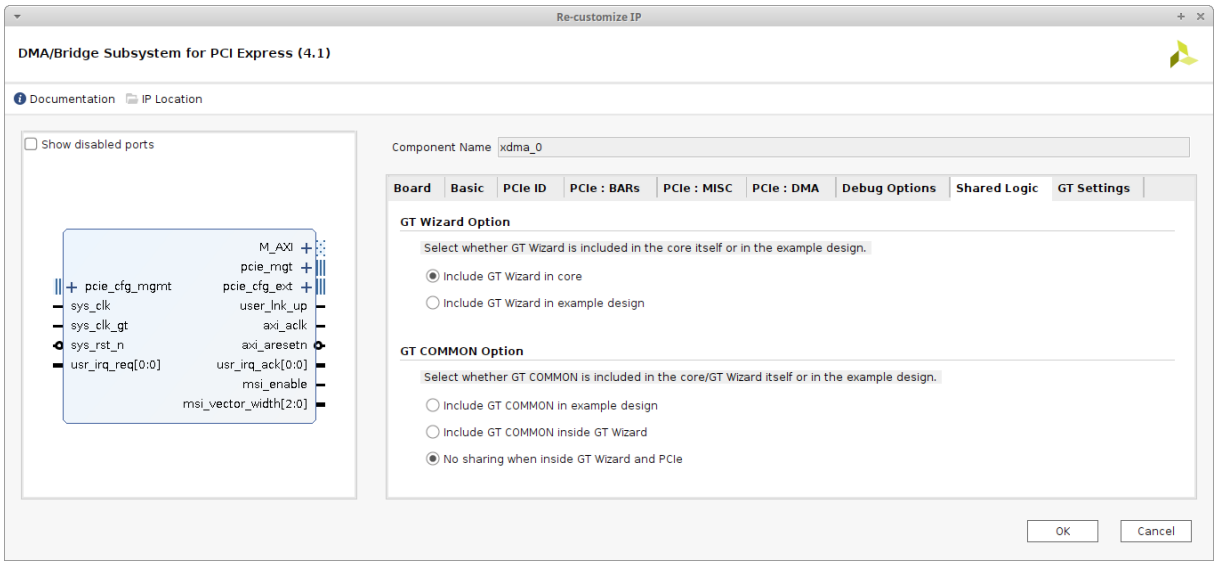


(b) QDMA

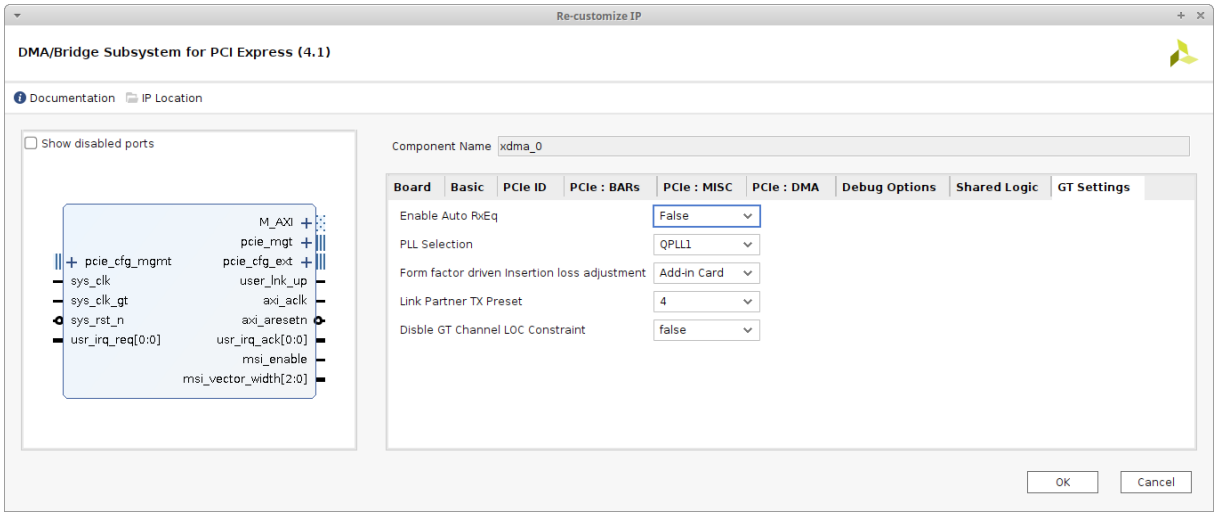
Figure 6.5: DMA tab



(a) XDMA debug options

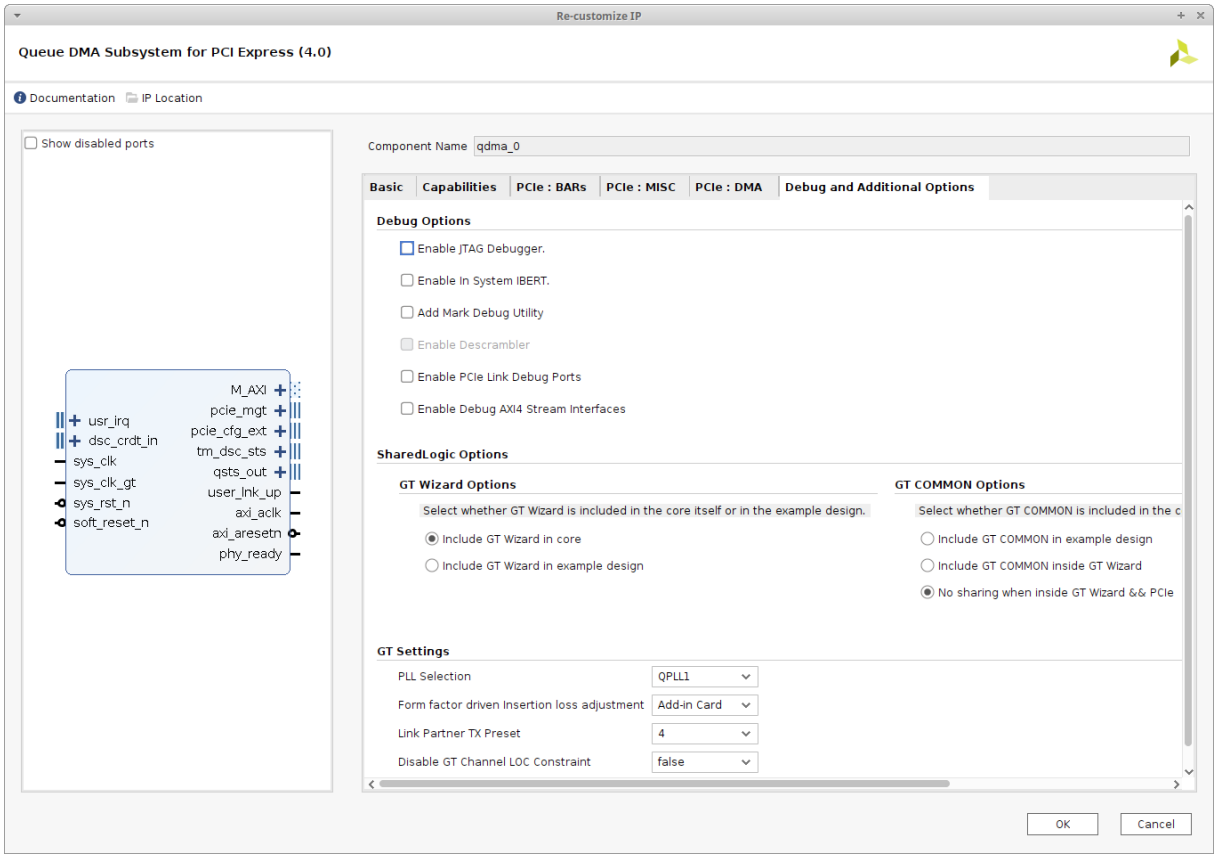


(b) XDMA shared logic



(c) XDMA GT settings

Figure 6.6: Debug and additional options tabs



(d) QDMA

Figure 6.6: Debug and additional options tabs

6.2.2 Constraints

The next step was adapting the time and pinout constraints.

The time XDC file only needed a change in the path to the AXI signal. Listing 6.3 shows the modified line.

```
1 set pcie_clk    [get_clocks -of_objects [get_pins design_1_i/qdma_0/
    axi_aclk]]
```

Listing 6.3: Modified line in the time XDC file.

Regarding the pinout constraints, a curious fact was observed: there were no pins assigned for the PCIe signals. Hence, it was decided to proceed with the bitstream generation without defining them, as for XDMA they were not required and the bitstream worked correctly.

6.2.3 Bitstream generation failure

The final step of the hardware development was generating the bitstream running the synthesis and implementation in Vivado.

This part is crucial since it is when the majority of problems can arise, as there are many things that can go wrong. It is important to keep in mind that the vanilla FPGA@SDV design already occupies a big percentage of LUTs, which results in limiting the Vivado routing possibilities to meet the time constraints. The main concerns were *i)* that the new IP would use more LUTs than we can *afford*, ending up in not closing timing; and *ii)* the uncertainty with the PCIe pins, to check if it would keep working without defining them.

Vivado offers the possibility to check how many FPGA resources are estimated to be used post-synthesis. It is useful as a *rule of thumb* because if the number of LUTs required is close to or above 90%, it is very likely that Vivado will struggle during the routing phase in the implementation and with a high chance of not meeting the time.

Therefore, the resource utilization after synthesis was checked and it was below 80%, meaning that Vivado would be very likely to close timing. After running the implementation, the WNS was positive (> 0). However, when generating the bitstream file, the errors shown in Listing 6.4 arose.

```
1 [DRC NSTD-1] Unspecified I/O Standard: 31 out of 162 logical ports use I
    /O standard (IOSTANDARD) value 'DEFAULT', instead of a user assigned
    specific value. This may cause I/O contention or incompatibility with
     the board power or connectivity affecting performance, signal
    integrity or in extreme cases cause damage to the device or the
    components to which it is connected. To correct this violation,
    specify all I/O standards. This design will fail to generate a
    bitstream unless all logical ports have a user specified I/O standard
     value defined. To allow bitstream creation with unspecified I/O
    standard values (not recommended), use this command: set_property
    SEVERITY {Warning} [get_drc_checks NSTD-1].  NOTE: When using the
    Vivado Runs infrastructure (e.g. launch_runs Tcl command), add this
    command to a .tcl file and add that file as a pre-hook for
    write_bitstream step for the implementation run. Problem ports:
    pci_express_x8_txp[7], pci_express_x8_txp[6], pci_express_x8_txp[5],
    pci_express_x8_txp[4], pci_express_x8_txp[3], pci_express_x8_txp[2],
    pci_express_x8_txp[1], pci_express_x8_txp[0], pci_express_x8_txn[7],
    pci_express_x8_txn[6], pci_express_x8_txn[5], pci_express_x8_txn[4],
    pci_express_x8_txn[3], pci_express_x8_txn[2], pci_express_x8_txn[1],
    pci_express_x8_txn[0], pci_express_x8_rxp[7], pci_express_x8_rxp[6],
    pci_express_x8_rxp[5], pci_express_x8_rxp[4], pci_express_x8_rxp[3],
    pci_express_x8_rxp[2], pci_express_x8_rxp[1], pci_express_x8_rxp[0],
    pci_express_x8_rxn[7], pci_express_x8_rxn[6], pci_express_x8_rxn[5],
    pci_express_x8_rxn[4], pci_express_x8_rxn[3], pci_express_x8_rxn[2],
    pci_express_x8_rxn[1], pci_express_x8_rxn[0], pcie_perstn,
    pcie_refclk_clk_p, and pcie_refclk_clk_n.
2
3 [DRC UCIO-1] Unconstrained Logical Port: 31 out of 162 logical ports
    have no user assigned specific location constraint (LOC). This may
    cause I/O contention or incompatibility with the board power or
```

```
connectivity affecting performance, signal integrity or in extreme
cases cause damage to the device or the components to which it is
connected. To correct this violation, specify all pin locations. This
 design will fail to generate a bitstream unless all logical ports
have a user specified site LOC constraint defined.  To allow
bitstream creation with unspecified pin locations (not recommended),
use this command: set_property SEVERITY {Warning} [get_drc_checks
UCIO-1].  NOTE: When using the Vivado Runs infrastructure (e.g.
launch_runs Tcl command), add this command to a .tcl file and add
that file as a pre-hook for write_bitstream step for the
implementation run.  Problem ports: pci_express_x8_txp[7],
pci_express_x8_txp[6], pci_express_x8_txp[5], pci_express_x8_txp[4],
pci_express_x8_txp[3], pci_express_x8_txp[2], pci_express_x8_txp[1],
pci_express_x8_txp[0], pci_express_x8_txn[7], pci_express_x8_txn[6],
pci_express_x8_txn[5], pci_express_x8_txn[4], pci_express_x8_txn[3],
pci_express_x8_txn[2], pci_express_x8_txn[1], pci_express_x8_txn[0],
pci_express_x8_rxp[7], pci_express_x8_rxp[6], pci_express_x8_rxp[5],
pci_express_x8_rxp[4], pci_express_x8_rxp[3], pci_express_x8_rxp[2],
pci_express_x8_rxp[1], pci_express_x8_rxp[0], pci_express_x8_rxn[7],
pci_express_x8_rxn[6], pci_express_x8_rxn[5], pci_express_x8_rxn[4],
pci_express_x8_rxn[3], pci_express_x8_rxn[2], pci_express_x8_rxn[1],
pci_express_x8_rxn[0], pcie_perstn, pcie_refclk_clk_p, and
pcie_refclk_clk_n.
```

Listing 6.4: Vivado errors reported during the bitstream generation phase.

6.2.4 Pinout problem

Listing 6.4 contains two errors that express the same root cause, the PCIe pins have not been assigned correctly.

The 31 reported pins are classified and explained in Table 6.1.

It was noted that all signals except `pcie_perstn` had the double of pins to their signal length. It is because those signals need to be connected to differential pins.

Differential pins in an FPGA are pairs of pins that are used to transmit signals differentially. In differential signaling, a pair of signals is transmitted with opposite polarity to each other, such that one signal represents the logical state (positive pin) and the other represents the logical complement (negative pin). The difference between the two signals is then used to carry the information. By transmitting signals differentially, it is possible to achieve higher signal integrity and reduce the effects of noise and interference on the

Signal name	Number of pins	Usage
`pci_express_x8_tx`	14	PCIe signals C2H
`pci_express_x8_rx`	14	PCIe signals H2C
`pcie_refclk_clk`	2	PCIe clock
`pcie_perstn`	1	PCIe reset

Table 6.1: PCIe pins used in the FPGA@SDV design.

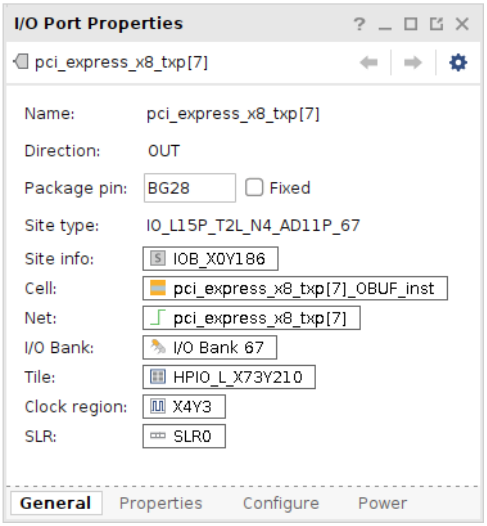


Figure 6.8: I/O port properties of pin `pci_express_x8_txp[7]` from Vivado.

signal. In an FPGA, differential pins are often used for high-speed signaling applications, such as high-speed data transmission or clock distribution.

Therefore, each signal bit requires two pins, which are referred to as `<signal_name>_p` and `<signal_name>_n` in Vivado.

As those 31 pins were not specified in any XDC file, Vivado automatically assigns them to General Purpose I/O (GPIO) pins, as can be seen in Figure 6.8. That auto-assignation ends up producing Design Rule Checks (DRCs) violations because those signals should be connected to differential pins instead of GPIO pins.

After understanding the previous concepts and Vivado flow, the solution was realized: defining those pins in the pinout XDC file.

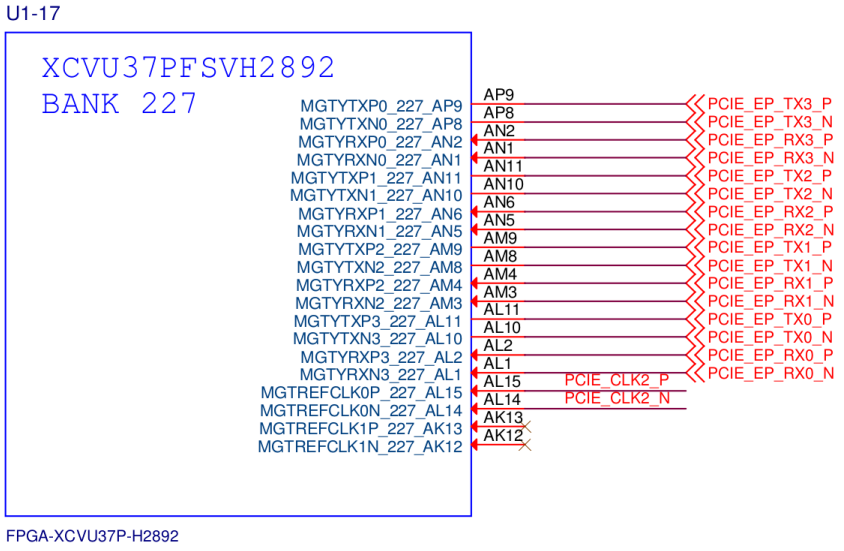
Although it was not necessary for the XDMA, we have guessed that as the XDMA IP is older than the QDMA one, the XDMA had been fine-tuned to automatically assign correctly the pins. Whereas for QDMA, that point is still not reached as it is a newer IP and much more complex than the XDMA one.

6.2.5 Pinout assignment

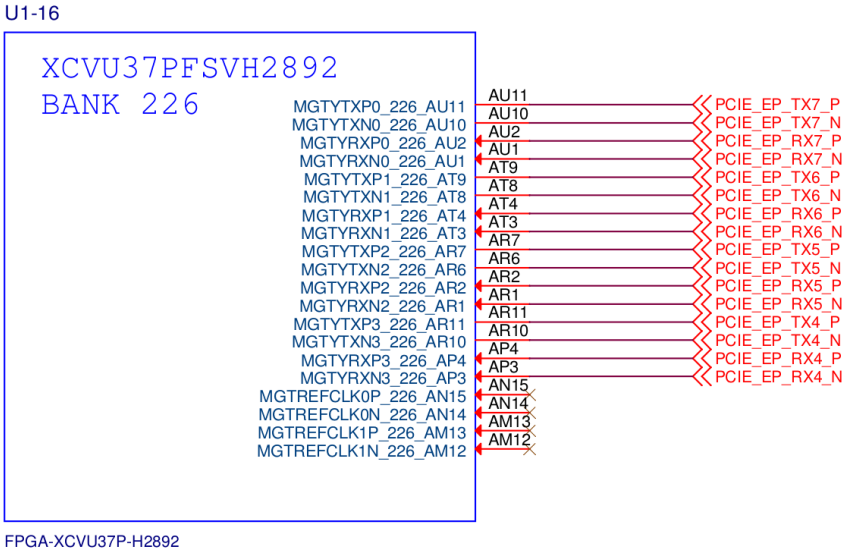
In order to be able to assign the correct PCIe pins, the VCU128 files with the board schematic were downloaded and studied.

The pins are grouped in banks. The PCIe pins are located in banks 224, 225, 226 and 227 [13]. Each bank contains 18 pins grouped in 9 pairs of differential pins. Those pairs correspond to: 1 PCIe clock, 4 PCIe `rx`, and 4 PCIe `tx`. A graphical representation of a bank and its pins is shown in Figure 6.9a. Those banks with differential pins are also referred to as *quads*.

Figure 6.9 shows the two selected banks for this thesis: bank 227 and bank 226. To know which banks had to be chosen, the documentation of the IP from which both XDMA



(a) VCU128 pinout schematic from bank 227. Source: [13].



(b) VCU128 pinout schematic from bank 226. Source: [13].

Figure 6.9: VCU128 banks with its corresponding pins.

and QDMA are built on top (Integrated Block for PCI Express (PCIe) solution IP core [37]) was consulted.

Each PCIe quad is formed by 4 lanes, as can be extracted from Figure 6.9. Hence, Xilinx recommends using consecutive lanes to improve the design place, route, and timing [37]. That means assigning consecutive quads when the lane width is > 4.

Usually, PCIe lane 0 is placed at the topmost quad and the consecutive ones are assigned vertically down. Figure 6.10 illustrates where the XCVU37P pins are located physically. The topmost quad inside the red square corresponds to quad 227 and there is where lane 0 is recommended to be placed. Therefore, the PCIe pins for our FPGA@SDV design should be placed in quad 227 and 226, the pins that are colored pastel green and baby blue, respectively. Keeping that in mind, the PCIe pinout manual assignment is the one shown in Listing 6.5.

```
1  set_property PACKAGE_PIN BF41     [get_ports pcie_perstn]
2  set_property IOSTANDARD  LVCMOS12 [get_ports pcie_perstn]
3
4  set_property PACKAGE_PIN AL15   [get_ports pcie_refclk_clk_p]
5  set_property PACKAGE_PIN AL14   [get_ports pcie_refclk_clk_n]
6
7  set_property PACKAGE_PIN AU2   [get_ports {pci_express_x8_rxp[7]}]
8  set_property PACKAGE_PIN AU1   [get_ports {pci_express_x8_rxn[7]}]
9  set_property PACKAGE_PIN AT4   [get_ports {pci_express_x8_rxp[6]}]
10 set_property PACKAGE_PIN AT3   [get_ports {pci_express_x8_rxn[6]}]
11 set_property PACKAGE_PIN AR2   [get_ports {pci_express_x8_rxp[5]}]
12 set_property PACKAGE_PIN AR1   [get_ports {pci_express_x8_rxn[5]}]
13 set_property PACKAGE_PIN AP4   [get_ports {pci_express_x8_rxp[4]}]
14 set_property PACKAGE_PIN AP3   [get_ports {pci_express_x8_rxn[4]}]
15 set_property PACKAGE_PIN AN2   [get_ports {pci_express_x8_rxp[3]}]
16 set_property PACKAGE_PIN AN1   [get_ports {pci_express_x8_rxn[3]}]
17 set_property PACKAGE_PIN AN6   [get_ports {pci_express_x8_rxp[2]}]
18 set_property PACKAGE_PIN AN5   [get_ports {pci_express_x8_rxn[2]}]
19 set_property PACKAGE_PIN AM4   [get_ports {pci_express_x8_rxp[1]}]
20 set_property PACKAGE_PIN AM3   [get_ports {pci_express_x8_rxn[1]}]
21 set_property PACKAGE_PIN AL2   [get_ports {pci_express_x8_rxp[0]}]
22 set_property PACKAGE_PIN AL1   [get_ports {pci_express_x8_rxn[0]}]
23
24 set_property PACKAGE_PIN AU11  [get_ports {pci_express_x8_txp[7]}]
25 set_property PACKAGE_PIN AU10  [get_ports {pci_express_x8_txn[7]}]
26 set_property PACKAGE_PIN AT9   [get_ports {pci_express_x8_txp[6]}]
27 set_property PACKAGE_PIN AT8   [get_ports {pci_express_x8_txn[6]}]
28 set_property PACKAGE_PIN AR7   [get_ports {pci_express_x8_txp[5]}]
29 set_property PACKAGE_PIN AR6   [get_ports {pci_express_x8_txn[5]}]
30 set_property PACKAGE_PIN AR11  [get_ports {pci_express_x8_txp[4]}]
31 set_property PACKAGE_PIN AR10  [get_ports {pci_express_x8_txn[4]}]
32 set_property PACKAGE_PIN AP9   [get_ports {pci_express_x8_txp[3]}]
33 set_property PACKAGE_PIN AP8   [get_ports {pci_express_x8_txn[3]}]
34 set_property PACKAGE_PIN AN11  [get_ports {pci_express_x8_txp[2]}]
35 set_property PACKAGE_PIN AN10  [get_ports {pci_express_x8_txn[2]}]
36 set_property PACKAGE_PIN AM9   [get_ports {pci_express_x8_txp[1]}]
37 set_property PACKAGE_PIN AM8   [get_ports {pci_express_x8_txn[1]}]
38 set_property PACKAGE_PIN AL11  [get_ports {pci_express_x8_txp[0]}]
39 set_property PACKAGE_PIN AL10  [get_ports {pci_express_x8_txn[0]}]
```

Listing 6.5: PCIe pinout defined in the file `pinout.xdc`.

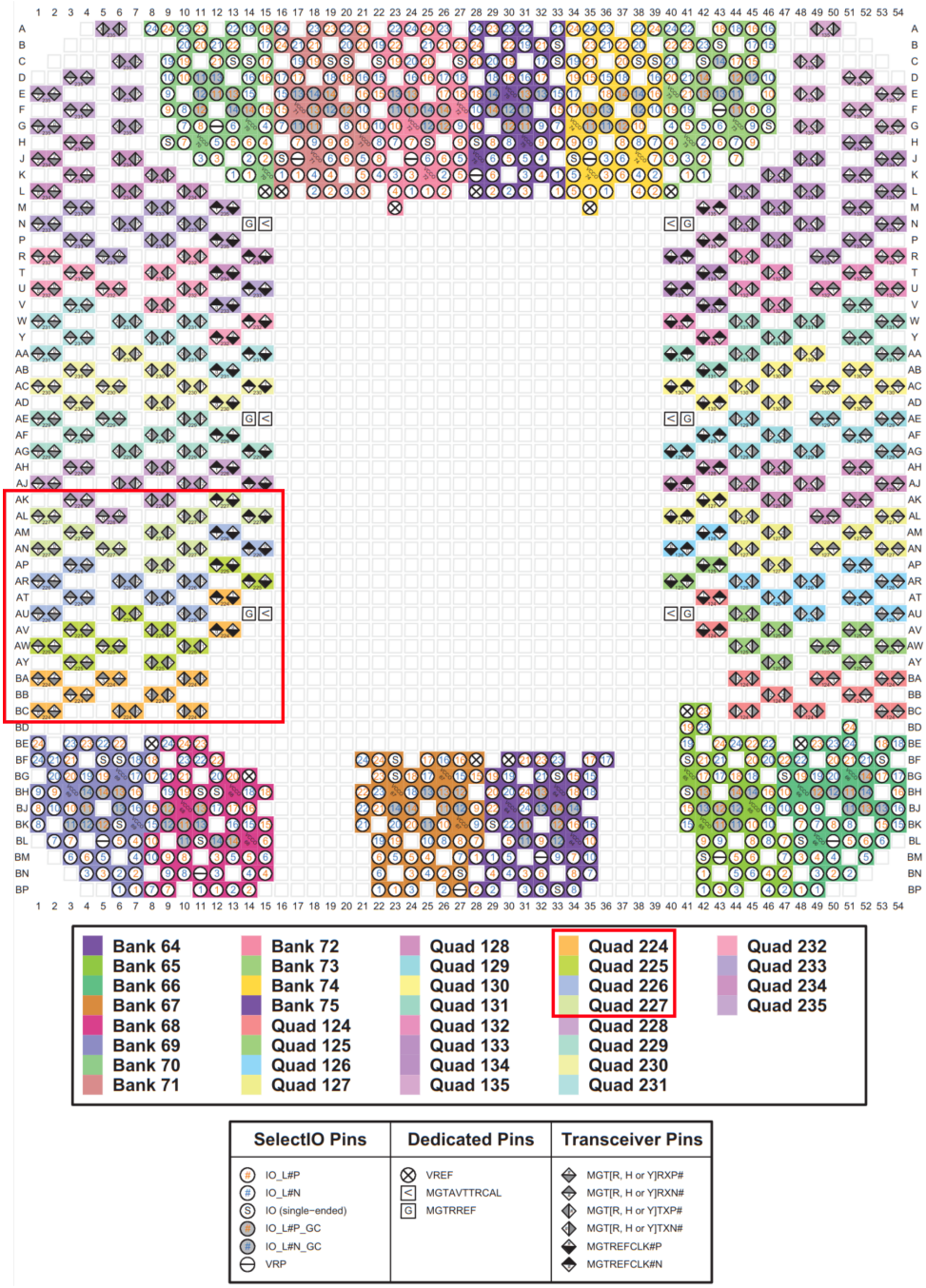


Figure 6.10: XCVU37P physical representation of its pinout with PCIe quads marked.

6.2.6 New pinout problem

Once the pinout definition was added and both synthesis and implementation were reset, Vivado was launched again to generate the bitstream. Unexpectedly, right after the

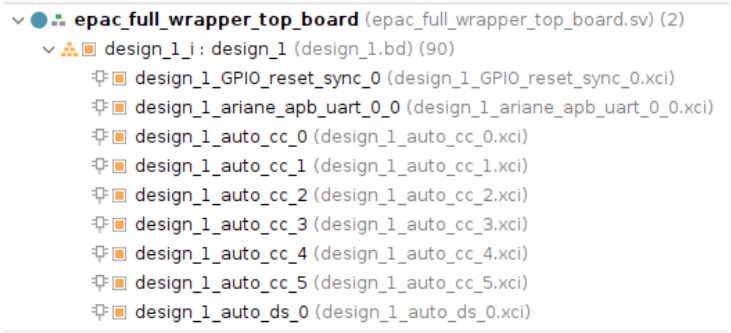
implementation, the same errors as in Listing 6.4 appeared. There were DRC violations in the PCIe pins, but this time only the `tx` pins were reported, as is shown in Listing 6.6. Although only failed the pins [7, 1], the `pci_express_x8_tx[0]` was well-defined.

```
1 [DRC NSTD-1] Unspecified I/O Standard: 14 out of 162 logical ports use I
     /O standard (IOSTANDARD) value 'DEFAULT', instead of a user assigned
     specific value. This may cause I/O contention or incompatibility with
      the board power or connectivity affecting performance, signal
     integrity or in extreme cases cause damage to the device or the
     components to which it is connected. To correct this violation,
     specify all I/O standards. This design will fail to generate a
     bitstream unless all logical ports have a user specified I/O standard
      value defined. To allow bitstream creation with unspecified I/O
     standard values (not recommended), use this command: set_property
     SEVERITY {Warning} [get_drc_checks NSTD-1].  NOTE: When using the
     Vivado Runs infrastructure (e.g. launch_runs Tcl command), add this
     command to a .tcl file and add that file as a pre-hook for
     write_bitstream step for the implementation run. Problem ports:
     pci_express_x8_txp[7], pci_express_x8_txp[6], pci_express_x8_txp[5],
     pci_express_x8_txp[4], pci_express_x8_txp[3], pci_express_x8_txp[2],
     pci_express_x8_txp[1], pci_express_x8_txn[7], pci_express_x8_txn[6],
     pci_express_x8_txn[5], pci_express_x8_txn[4], pci_express_x8_txn[3],
     pci_express_x8_txn[2], and pci_express_x8_txn[1].
2
3
4 [DRC UCIO-1] Unconstrained Logical Port: 14 out of 162 logical ports
     have no user assigned specific location constraint (LOC). This may
     cause I/O contention or incompatibility with the board power or
     connectivity affecting performance, signal integrity or in extreme
     cases cause damage to the device or the components to which it is
     connected. To correct this violation, specify all pin locations. This
      design will fail to generate a bitstream unless all logical ports
     have a user specified site LOC constraint defined.  To allow
     bitstream creation with unspecified pin locations (not recommended),
     use this command: set_property SEVERITY {Warning} [get_drc_checks
     UCIO-1].  NOTE: When using the Vivado Runs infrastructure (e.g.
     launch_runs Tcl command), add this command to a .tcl file and add
     that file as a pre-hook for write_bitstream step for the
     implementation run.  Problem ports: pci_express_x8_txp[7],
     pci_express_x8_txp[6], pci_express_x8_txp[5], pci_express_x8_txp[4],
     pci_express_x8_txp[3], pci_express_x8_txp[2], pci_express_x8_txp[1],
     pci_express_x8_txn[7], pci_express_x8_txn[6], pci_express_x8_txn[5],
     pci_express_x8_txn[4], pci_express_x8_txn[3], pci_express_x8_txn[2],
     and pci_express_x8_txn[1].
```

Listing 6.6: Vivado errors reported during the bitstream generation phase.

It was very strange since the other PCIe pins were correctly assigned in the post-implementation design.

The first possibility was that `tx` pins were not properly defined. They were checked and they were already well-defined. Just to be completely sure, I cloned a vanilla XDMA project and replaced the pinout file with the one modified, with the PCIe pins explicitly



(a) BD files in a clean Vivado project.

(b) BD files post-synthesis in a Vivado project.

Figure 6.11: Source files describing a BD in Vivado.

specified. The previous errors did not appear and the bitstream was generated. Therefore, the source of the errors had to be somewhere else.

The other major change was in the BD, as the PCIe IP had been. The parameter from both XDMA and QDMA IPs were compared via the Vivado GUI, with the emergent window that appears to change an IP configuration. There were no differences in the parameters that could affect the IP's output port `pcie_mgt`.

After that verification, what was left to check were the intermediate files generated during the Vivado bitstream generation flow. It was realized that new files appeared post-synthesizing the BD design, as can be seen in Figure 6.11. The generated Verilog files are automatically created by Vivado. In the Verilog BD top file, `design.v`, the PCIe signals were searched.

Listing 6.7 contains the relevant lines with the PCIe signals `tx` and `rx`. It is noticed that the `pci_express_x8_rx` signals are declared as 8-bit signals, whereas the `tx` signals are only 1-bit wide. That explains why the least significant bit from `pci_express_x8_tx` was correctly assigned, it was because it was declared.

```
1 //[...]
2
3 // signal declaration
4 (* X_INTERFACE_INFO = "xilinx.com:interface:pcie_7x_mgt:1.0
     pci_express_x8 rxn" *) input [7:0]pci_express_x8_rxn;
5 (* X_INTERFACE_INFO = "xilinx.com:interface:pcie_7x_mgt:1.0
     pci_express_x8 rxp" *) input [7:0]pci_express_x8_rxp;
6 (* X_INTERFACE_INFO = "xilinx.com:interface:pcie_7x_mgt:1.0
     pci_express_x8 txn" *) output pci_express_x8_txn;
```

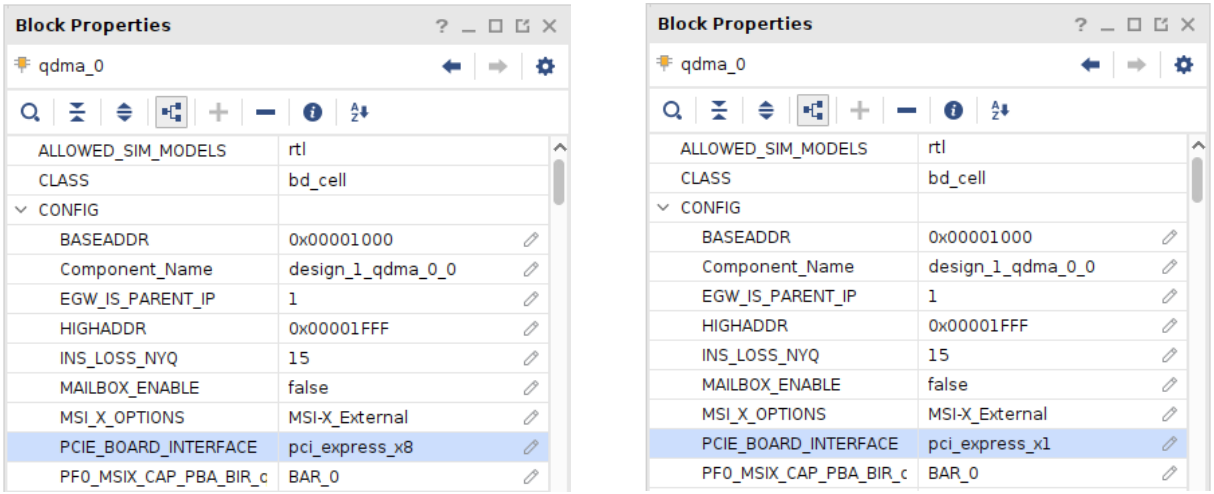
```
7  (* X_INTERFACE_INFO = "xilinx.com:interface:pcie_7x_mgt:1.0
       pci_express_x8 txp" *) output pci_express_x8_txp;
8
9  //[...]
10
11 // signal assignment
12 assign pci_express_x8_txn = qdma_0_pcie_mgt_txn[0];
13 assign pci_express_x8_txp = qdma_0_pcie_mgt_txp[0];
14
15 assign qdma_0_pcie_mgt_rxn = pci_express_x8_rxn[7:0];
16 assign qdma_0_pcie_mgt_rxp = pci_express_x8_rxp[7:0];
17
18 //[...]
```

Listing 6.7: Relevant PCIe lines from `design.v` file.

Subsequently, the root issue had to be found, since the `design.v` file is being generated by Vivado automatically and it was well-produced for the XDMA project. It became clear that something in the QDMA IP was off, and the settings were checked again. Although this time they were verified in a different Vivado window, the *Block Properties* window. Figure 6.12 contains the Block properties for each IP. It can be seen that the highlighted property, `PCIE_BOARD_INTERFACE`, is different. That could be the cause for not having the correct signal sizes.

The `PCIE_BOARD_INTERFACE` value in the QDMA project was fixed manually. Then it was verified that the other parameters were correct and the bitstream generation flow was run again.

Finally, **the QDMA bitstream was created correctly** with no errors.



(a) XDMA properties. (b) QDMA properties.

Figure 6.12: IP properties shown in the Block Properties window from Vivado GUI.

6.2.7 Driver and tools compilation

Xilinx provides two QDMA drivers: PF module and VF module, as it was explained in Section 4.6.4. The PF module is the one that had to be used for this thesis, since the QDMA IP inside the FPGA is set to use only a PCIe PF. The source code is in a public GitHub repository called `dma_ip_drivers`[1].

The PF module was compiled in the Pickle x86 processor with a simple `make driver MODULE=mod_pf` command. A `.ko` file was generated named `qdma_pf.ko`.

The compilation of the tools was also done in the host machine executing `make apps`. It generated all 5 tool's binaries.

6.2.8 Data word test

The data word test is a C program that:

1. Opens the FPGA device,
2. Writes a word (4 bytes) to an FPGA memory direction,
3. Reads from the same memory direction a word, and
4. Compares if the written word is equal to the read word.

The code written to perform this test is at appendix A.1, in Listing A.1.

The only part of the code dependent on the type of PCIe subsystem used is the first one, opening the FPGA device. It opens the XDMA H2C and C2H 0 channels, or the QDMA H2C and C2H 0 queues, depending on what is passed as an argument: `xdma` or `qdma`. The rest of the code, and steps, are independent of the PCIe subsystem. The data word test was first tested in the XDMA environment (driver and bitstream).

The objectives of this test are *i)* checking if the H2C and C2H channels/queues can be accessed properly with a C program and *ii)* verifying that data flows in both directions correctly. In other words, make sure that the QDMA IP and the driver and queues are properly configured.

This code was first tested with the XDMA IP and driver to validate its functionality with regard to verifying QDMA. Listing 6.8 contains the output from the execution of that test and it can be seen in the last line that the test was successful.

```
1 $ ./data_test xdma
2 Stating data test for XDMA:
3 Writing data 0xabcdef12 to address 0x800000000000
4 Writing into FPGA
5 Written bytes: 4
6 Reading from FPGA
7 Read bytes: 4
```

[1] https://github.com/Xilinx/dma_ip_drivers

```
8 Read bytes: 4
9 Read word abcdef12 is EQUAL than the test word abcdef12
```

Listing 6.8: Output from the terminal after executing the data word test binary.

6.2.9 Loading the driver

The compiled module file had to be inserted in the kernel to be able to use it. The Linux installed in the host machine provides different commands to manage kernel modules, which are shown in Table 6.2.

Module Function	Commands
Insert module	`modprobe`, `insmod`
Remove module	`modprobe`, `rmmod`
List current modules	`lsmod`
Show module information	`modinfo`

Table 6.2: Linux commands to manage kernel modules.

Firstly, it was checked if the XDMA module was installed with `lsmod | grep xdma` command. It reported that the XDMA driver was loaded, so it was removed with `sudo /sbin/rmmod xdma`. After removing it, the QDMA driver had to be loaded.

Prior to inserting the QDMA module, its parameters were checked with the `modinfo` command. Listing 6.9 contains some of the `modinfo` output and it can be seen that the `mode` parameter is the one that needs to be configured. This parameter is defined with the bus number, the PF number and the mode, which was detailed in Section 4.6.4. The bus number depends on the PCIe slot where the FPGA is plugged and how Linux identifies it, hence it was identified with the command `lspci`, which lists all PCI devices, including the PCIe ones. That command reports both bus number and PF, although the PF was already known from the QDMA IP configuration, so their value is `08` and `0`, respectively. The mode depends on the interrupt type set in the QDMA IP, so the mode was `4` as is the one for legacy interrupts. Therefore, the QDMA driver parameter was `mode=0x08:0:4`.

```
1  aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel/bin
2  \$ modinfo qdma-pf.ko
3  filename:       /home/aquerol/tfm/dma_ip_drivers/QDMA/linux-kernel/bin/
       qdma-pf.ko
4  license:        Dual BSD/GPL
5  version:        2023.1.0.0.
6  description:    Xilinx QDMA PF Reference Driver
7  author:         Xilinx, Inc.
8  srcversion:     A9E058C9EAFC72CF9E4DD21
9  [...]
10 name:           qdma_pf
11 vermagic:       5.4.0-139-generic SMP mod_unload modversions
12 parm:           mode:Load the driver in different modes, dflt is auto
       mode, format is "<bus_num>:<pf_num>:<mode>" and multiple comma
       separated entries can be specified (string)
```

```
13 parm:           config_bar:specify the config bar number, dflt is 0,
       format is "<bus_num>:<pf_num>:<bar_num>" and multiple comma separated
        entries can be specified (string)
14 parm:           master_pf:specify the master_pf, dflt is 0, format is "<
       bus_num>:<master_pf>" and multiple comma separated entries can be
       specified (string)
15 parm:           num_threads:Number of threads to be created each for
       request and writeback processing (uint)
```

Listing 6.9: Console output from the `modinfo` command

First, the `modprobe` command was tried, but it did not work because it looks for modules in the module directory under `/lib/modules/`, and not in a path given by the user. Therefore, the `insmod` command had to be used as it allows inserting modules from a given path. The command line executed was `sudo /sbin/insmod bin/qdma_pf.ko mode=0x08:0:4`. Afterward, it was checked if the driver was loaded with the command `lsmod`.

After that validation, the bitstream was programmed into the FPGA. Then, the device control tool named `dma-ctl` (explained in Section 4.6.4) was used to show devices with QDMA. However, it reported nothing. Therefore, the output of the command `dmesg` was checked. This command displays all kernel messages and it showed the contents from Listing 6.10. It can be observed that the QDMA module failed to initialize the QDMA device.

```
1  [17:02] aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel
2  \$ dmesg
3  [...]
4  [955775.545957] qdma_pf:qdma_mod_init: Xilinx QDMA PF Reference Driver
       v2023.1.0.0.
5  [955775.546301] qdma_pf:probe_one: 0000:08:00.0: func 0x0, p/v 0/0,0
       x0000000000000000.
6  [955775.546305] qdma_pf:probe_one: Configuring '08:00:0' as master pf
7  [955775.546306] qdma_pf:probe_one: Driver is loaded in legacy interrupt
       (4) mode
8  [955775.546307] qdma_pf:qdma_device_open: qdma-pf, 08:00.00, pdev 0
       x000000007e662b7b, 0x10ee:0x9028.
9  [955775.546438] Device Type: Soft IP
10 [955775.546440] IP Type: EQDMA4.0 Soft IP
11 [955775.546440] Vivado Release: vivado 2020.2
12 [955775.546450] qdma_pf:qdma_device_attributes_get: qdma08000-p0000
       :08:00.0: num_pfs:1, num_qs:512, flr_present:0, st_en:0, mm_en:1,
       mm_cmpt_en:0, mailbox_en:0, mm_channel_max:1, qid2vec_ctx:0,
       cmpt_ovf_chk_dis:1, mailbox_intr:1, sw_desc_64b:1, cmpt_desc_64b:1,
       dynamic_bar:1, legacy_intr:1, cmpt_trig_count_timer:1
13 [955775.546452] qdma_pf:qdma_device_open: Vivado version = vivado 2020.2
14 [955775.546454] qdma_dev_entry_create: Created the dev entry
       successfully
15 [955776.151830] hw_monitor_reg: Reg read=5 Expected=0, err:-3
16 [955776.157313] eqdma_indirect_reg_clear: hw_monitor_reg failed with err
       :-3
17 [955776.164013] qdma_pf:qdma_device_init: init ctxt write failed, err -3
18 [955776.170451] qdma_pf:qdma_device_online: qdma_init failed -16.
```

```
19 [955776.170451] qdma_pf:qdma_device_open: Failed to set the dma device
       online, err = -16
20 [955776.170530] qdma-pf: probe of 0000:08:00.0 failed with error -16
```

Listing 6.10: Console output from the `dmesg` command

Some research indicated that it was because there were 3 signals that needed to be set to 1. Those signals are `tm_dsc_sts_rdy`, `qsts_out_rdy` and `soft_reset_n`. The first two signals belong to the two AXI buses not used in the QDMA IP and they are *ready* signals, which indicate to the IP that everything is ready to start. The `soft_reset_n` signal corresponds to the DMA reset for the IP. To set them to 1, a constant IP was instantiated in the BD and configured as Figure 6.13 shows and its output was connected to those signals as Figure 6.14 displays. With those additions performed, a new bitstream was generated.

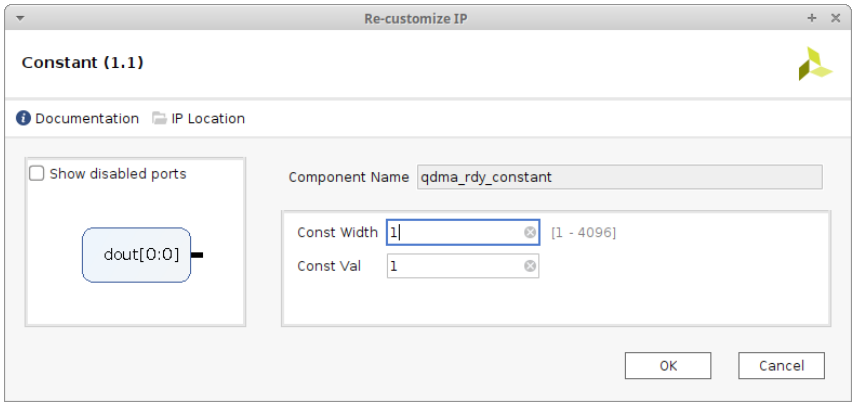


Figure 6.13: Constant IP settings.

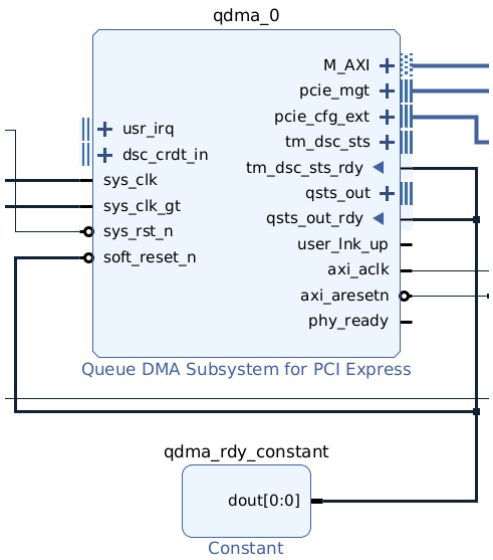


Figure 6.14: BD with the ready and soft reset signals from QDMA connected to a Constant IP.

6.2.10 Queue configuration

Once the bitstream was generated, it was programmed into the FPGA and the `dmesg` output was checked again. As Listing 6.11 shows, this time the QDMA module was loaded successfully.

```
1  [1795969.178347] qdma_pf:qdma_mod_init: Xilinx QDMA PF Reference Driver
       v2023.1.0.0.
2  [1795969.178745] qdma_pf:probe_one: 0000:08:00.0: func 0x0, p/v 0/0,0
       x0000000000000000.
3  [1795969.178750] qdma_pf:probe_one: Configuring '08:00:0' as master pf
4  [1795969.178751] qdma_pf:probe_one: Driver is loaded in legacy interrupt
       (4) mode
5  [1795969.178754] qdma_pf:qdma_device_open: qdma-pf, 08:00.00, pdev 0
       x00000000058486ee, 0x10ee:0x9028.
6  [1795969.178916] Device Type: Soft IP
7  [1795969.178918] IP Type: EQDMA4.0 Soft IP
8  [1795969.178919] Vivado Release: vivado 2020.2
9  [1795969.178930] qdma_pf:qdma_device_attributes_get: qdma08000-p0000
       :08:00.0: num_pfs:1, num_qs:512, flr_present:0, st_en:0, mm_en:1,
       mm_cmpt_en:0, mailbox_en:0, mm_channel_max:1, qid2vec_ctx:0,
       cmpt_ovf_chk_dis:1, mailbox_intr:1, sw_desc_64b:1, cmpt_desc_64b:1,
       dynamic_bar:1, legacy_intr:1, cmpt_trig_count_timer:1
10 [1795969.178932] qdma_pf:qdma_device_open: Vivado version = vivado
       2020.2
11 [1795969.178934] qdma_dev_entry_create: Created the dev entry
       successfully
12 [1795969.184503] qdma_pf:qdma_device_open: 0000:08:00.0, 08000, pdev 0
       x00000000058486ee, xdev 0x000000002e667a75, ch 1, q 0, vf 0.
```

Listing 6.11: Console output from the `dmesg` command

The following stage was configuring the queues with the `dma-ctl` tool as Listing 6.12 shows. First, the device name was obtained with the command from Line 2. Its output also reports the maximum number of Queue Pairs (QP) that can be assigned to a given QDMA device. By default, `max QP` is set to 0, so a configuration file has to be modified with the value desired, as Line 10 exposes. Line 17 shows the `dma-ctl` device list with the modified `max QP`. A queue needs to be created and then started so it is functional. Line 21 created the queue for the QDMA device with index 0, in Memory-Mapped (MM) mode and for the host-to-card (H2C) direction. It was checked that no errors arose with `dmesg`, in Line 24. Finally, the queue in the opposite direction, card-to-host (C2H), had to be created as Line 28 exhibits. However, a `dmesg` error message appeared (Line 31).

```
1  [11:40] aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel/bin
2  $ ./dma-ctl dev list
3  qdma08000   0000:08:00.0    max QP: 0, -~-
4
5  [11:41] aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel/bin
6  $ cat /sys/bus/pci/devices/0000:08:00.0/qdma/qmax
7  0
8
9  [11:57] aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel/bin
10 $ echo 8 > /sys/bus/pci/devices/0000:08:00.0/qdma/qmax
```

```
11
12 [12:00] aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel/bin
13 $ cat /sys/bus/pci/devices/0000:08:00.0/qdma/qmax
14 8
15
16 [12:00] aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel/bin
17 $ ./dma-ctl dev list
18 qdma08000   0000:08:00.0    max QP: 8, 0~7
19
20 [12:02] aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel/bin
21 $ ./dma-ctl qdma08000 q add idx 0 mode mm dir h2c
22
23 [12:02] aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel/bin
24 $ dmesg
25 [1811004.776719] qdma_pf:intr_legacy_clear: un-registering legacy
      interrupt from qdma08000
26
27 [12:03] aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel/bin
28 $ ./dma-ctl qdma08000 q add idx 0 mode mm dir c2h
29
30 [12:03] aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel/bin
31 \$ dmesg
32 [1811031.554648] qdma_pf:xnl_q_add: xpdev_queue_add() failed: -22
```

Listing 6.12: Commands and output from the console.

Researching in the QDMA driver source code, it was found the following comment: `/** support only 1 queue in legacy interrupt mode */`, in file `QDMA/linux-kernel/driver/libqdma/libqdma_export.c`. Therefore, having one queue per direction was not feasible with legacy interrupts and it was decided to use MSI interrupts.

6.2.11 MSI interruptions

In order to change the interrupt type from legacy to MSI, the *Legacy Configuration Extended interface* option has to be deactivated, as can be observed in Figure 6.15. The bitstream was regenerated.

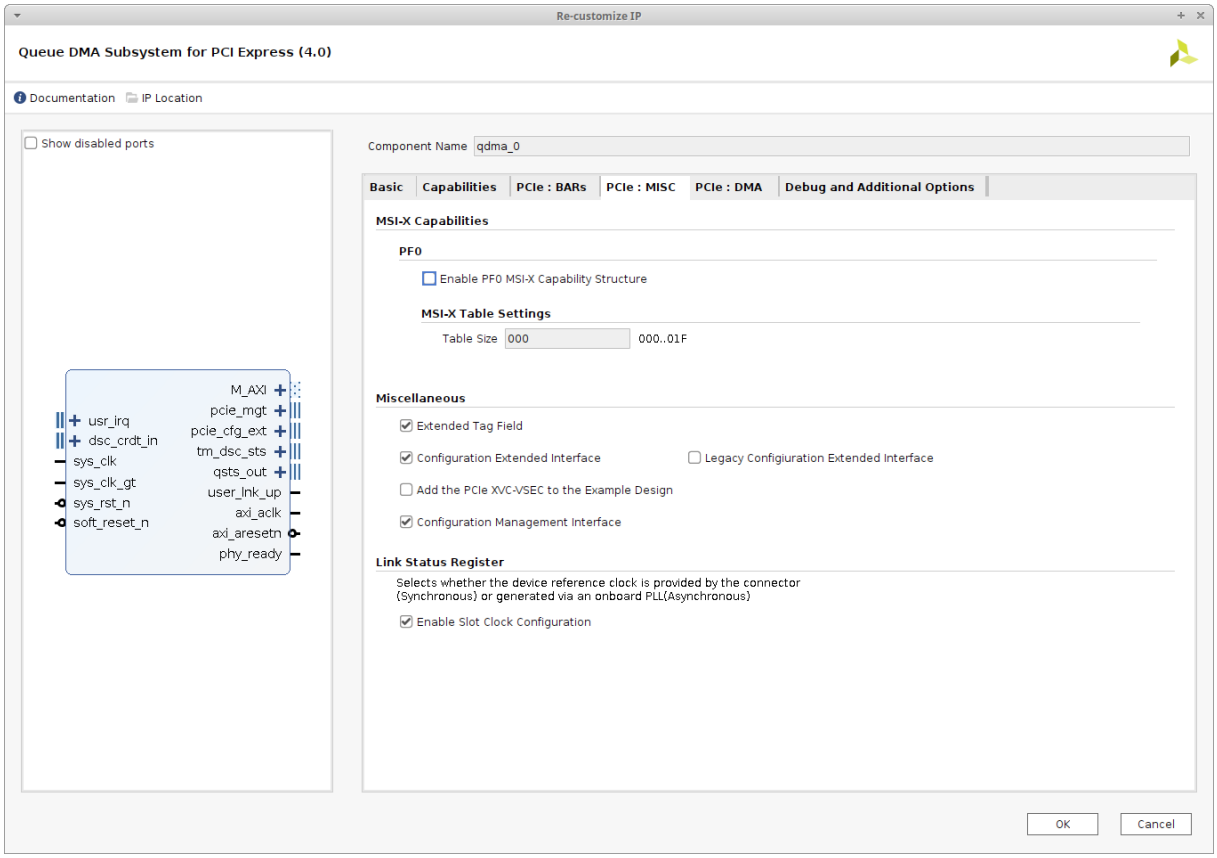


Figure 6.15: QDMA MISC tab with MSI interruptions enabled.

The driver had to be removed and loaded with a different mode, which was the Direct Interrupt mode (2). Therefore, the module parameter was `mode=0x08:0:2`. After loading the QDMA driver and reprogramming the FPGA, the queues were added and initialized as can be observed in Listing 6.13. Line 5 adds a bi-directional MM QP, in other words, with a single command the two directions from the QP are added. Line 8 starts this queue in both directions. The command `dmesg` reported some errors related to AXI completion (`ST or MM cmpt not supported`), which can be ignored because the completion is managed automatically by the QDMA IP; and other errors related to AXI-ST (`ST is not supported`), which is a not-used interface in our design, so it is expected to report an AXI-ST error. Finally, the QDMA device appeared, as can be seen in Line 22.

```
1  [16:34] aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel/bin
2  $ echo 8 > /sys/bus/pci/devices/0000:08:00.0/qdma/qmax
3
4  [16:35] aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel/bin
5  $ ./dma-ctl qdma08000 q add idx 0 mode mm dir bi
6
7  [16:35] aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel/bin
8  $ ./dma-ctl qdma08000 q start idx 0 dir bi
9
10 [16:35] aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel/bin
11 $ dmesg
```

```
12 [5620394.605029] qdma_global_writeback_interval_read: ST or MM cmpt not
      supported, err:-7
13 [5620394.612939] qdma_pf:qdma_csr_read: Hardware Feature not supported
14 [5620394.612944] qdma_read_global_buffer_sizes: ST is not supported, err
      :-7
15 [5620394.619642] qdma_pf:qdma_csr_read: Hardware Feature not supported
16 [5620394.619647] qdma_read_global_timer_count: ST or MM cmpt not
      supported, err:-7
17 [5620394.626952] qdma_pf:qdma_csr_read: Hardware Feature not supported
18 [5620394.626956] qdma_read_global_counter_threshold: ST or MM cmpt not
      supported, err:-7
19 [5620394.634778] qdma_pf:qdma_csr_read: Hardware Feature not supported
20
21 [16:36] aquerol@pickle-5 ~/tfm/dma_ip_drivers/QDMA/linux-kernel/bin
22 \$ ls /dev/qdma*
23 /dev/qdma08000-MM-0
```

Listing 6.13: Commands and output from the console.

The data word test was used to check the QDMA environment. It did not work because it could not open the QDMA device. As `dmesg` did not report any error, the driver was reloaded with a different mode, the Indirect Interrupt Mode with value 3. The queues were configured again and the data word test encountered the same error. Consequently, it was decided to test a bitstream with MSI-X interrupts.

6.2.12 MSIx interruptions

To disable MSI and configure MSI-X interrupts, the *Enable PF0 MSI-X Capability Structure* checkbox was enabled and the MSI-X table size was set with the default value. The bitstream was generated and programmed into the FPGA.

As the driver was already loaded with the Indirect Interrupt Mode, it was not necessary to reload it because after each bitstream reprogram the PCIe devices are rescanned and the driver can initialize automatically the new QDMA-compatible device. The MM queues were configured as in the previous section and the data word test could not open the QDMA device anyway.

One possibility was that the QDMA IP was not well configured because, while researching the error related to signals `tm_dsc_sts_rdy`, `qsts_out_rdy` and `soft_reset_n` (explained previously in Section 6.2.9), two main different possibilities were equally explained in the official Xilinx forums. One was the solution implemented previously: the three signals set to 1, while the other was setting the two ready signals to 1 and connecting the soft reset signal to the same source as the PCIe reset (`sys_rst_n`). Hence, the second possibility was implemented changing the soft reset source, as can be observed in Figure 6.17.

The newly generated bitstream was tested, but the problem remained unsolved. Once at this point, a new approach was taken: trying to track down the reason why XDMA was working while QDMA was not. This perspective led to discovering that the XDMA device had different permissions than the QDMA one. That was due to a *udev* rule that was defined for XDMA. An *udev* rule is a Linux configuration file that specifies how the system

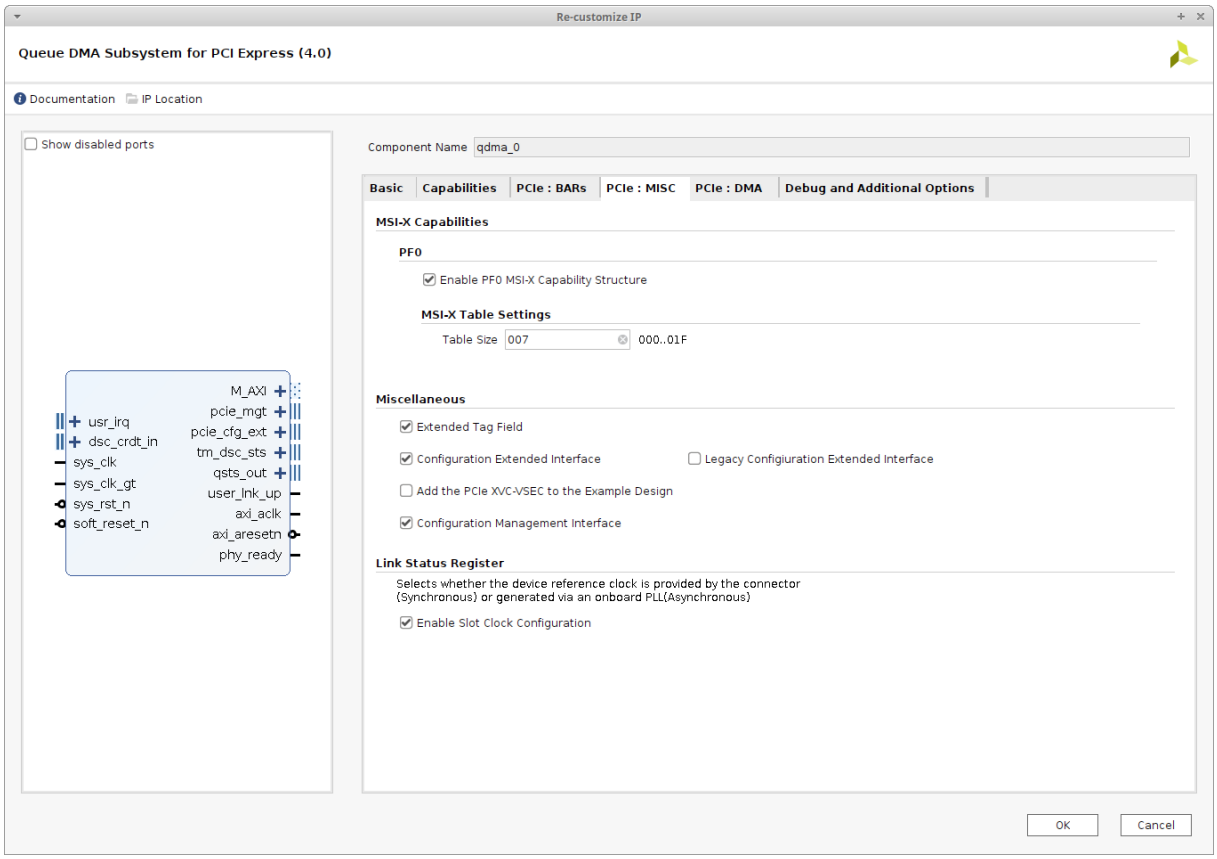


Figure 6.16: QDMA MISC tab with MSIx interruptions enabled.

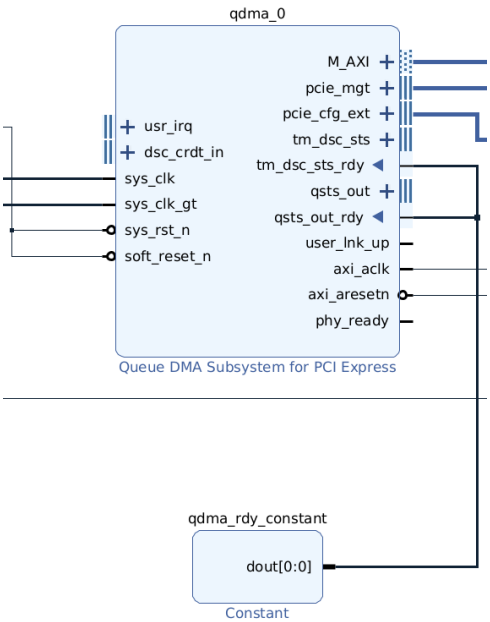


Figure 6.17: BD with the `soft_reset_n` signal connected to the same source as `sys_reset_n`.

should handle specific devices or events, which allows customization and automation of device-related tasks. For this reason, a new *udev* rule was required for QDMA devices. It was asked to the system administrator of the cluster. As soon as it was created, the data word test was executed and it **finished successfully**.

With this successful test, it was clear that the same write and read code could be used indistinctively of the PCIe subsystem underneath. That fact was key so the final part of the technical development could be performed, which was changing the SDV tools and checking out if the Linux image could still be loaded.

6.2.13 Adapt SDV tools

The SDV tools only required changing the devices from the XDMA ones to the QDMA new ones, as the same write and read mechanisms used for the XDMA driver and IP could be reused. It was a simple change because only two lines had to be replaced. On the one hand, Listing 6.14 contains the XDMA device opening, and on the other hand, Listing 6.15 shows those lines for the QDMA devices.

```
1 h2c_fd = open("/dev/xdma0_h2c_0"
      , O_RDWR);
2 // [...]
3 c2h_fd = open("/dev/xdma0_c2h_0"
      , O_RDWR);
```

Listing 6.14: Device-related lines in the C code used in the SDV tools for XDMA.

```
1 c2h_fd = open("/dev/qdma08000-MM
      -0", O_RDWR);
2 // [...]
3 c2h_fd = open("/dev/qdma08000-MM
      -0", O_RDWR);
```

Listing 6.15: Device-related lines modified in the C code used in the SDV tools for QDMA.

After changing the devices, the final tests were performed: offloading the Linux image and checking if EPAC boots properly. The tool that offloads the Linux images was executed, followed by opening a UART shell to verify that the boot process finishes properly. After the boot procedure, the login appeared and **the FPGA@SDV could be used perfectly as usual**, as the vanilla version with XDMA.

6.2.14 Process automation

At this point, a functional bitstream, a properly loaded and configured driver, and FPGA@SDV tools adaptation were achieved. However, became obvious that the workflow to be able to use the FPGA@SDV was more complex with QDMA than with XDMA.

Assuming the driver is already loaded, the XDMA workflow consists of reprogramming the FPGA and booting Linux, a two-step procedure. Whereas, the QDMA workflow requires reprogramming the FPGA, changing the QMAX file, creating the queue, starting the queue, and finally booting Linux, which implies 3 more steps for the users.

Those 3 extra actions may seem trivial once you know what to do, but the SDVs are used by many users with different backgrounds and setting up the FPGA@SDV should not be more complex. On the contrary, the process should be equal, or even better, easier.

Moreover, it was checked if the XDMA and QDMA drivers could coexist without having to remove and load them each time. The answer was that both drivers can exist side-by-side without problems, which avoids reconfiguring modules from one user to another.

Pickle nodes can be allocated from two different partitions: `fpga` and `fpga-sdv`, as was detailed in Section 3.2. For the users of the first partition, a Bash script was created to omit the explicit QDMA setup, which is shown in Listing 6.16. For the `fpga-sdv` partition, the necessary add and start queue commands were included in the Slurm configuration so users can request the QDMA release. Figure 6.18 exhibits a Slurm allocation of a `fpga-sdv` node with the QDMA release and how it can be properly accessed. This way, both target users are covered and the underlying PCIe subsystem is transparent to them.

```
1  #!/bin/bash
2
3  # 1. Change QMAX value
4  echo 8 >  /sys/bus/pci/devices/0000\:08\:00.0/qdma/qmax
5  echo "QMAX value set to 8"
6
7  # 2. Add queue
8  $qdma_path/dma-ctl qdma08000 q add idx 0 mode mm dir bi
9  echo "Bi-directional Queue 0 added"
10
11 # 3. Start queue
12 $qdma_path/dma-ctl qdma08000 q start idx 0 dir bi
13 echo "Queue 0 started"
```

Listing 6.16: Bash script to automate setting up QDMA driver and queues.

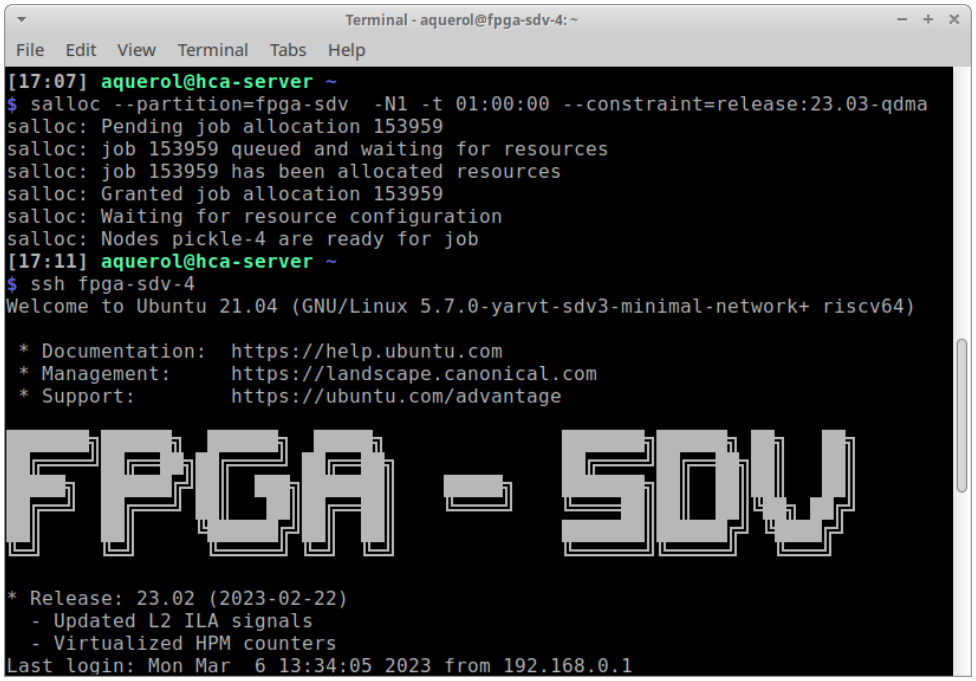


Figure 6.18: Console screenshot allocationg a Pickle node in `fpga-sdv` mode with the QDMA configuration.

6.3 Evaluation

In order to evaluate the improvement or the worsening of the PCIe subsystem IP replacement, a series of tests and metrics have been extracted from both XDMA and QDMA IPs.

The performance tests have been executed with the XDMA vanilla bitstreams and the final QDMA bitstream (with MSI-X and the soft reset connected to the PCIe reset signal). Therefore, both functional designs can be compared in performance terms: bandwidth and time to offload the Linux image.

Moreover, metrics from Vivado have been extracted to not only evaluate its performance but its implications in the design. Those metrics are resource utilization, the synthesis and implementation time required, and the WNS. They have been obtained for all the available implemented versions, although not all of them have worked in the FPGA. The reason behind comparing all of them is to be able to understand better the differences between the XDMA vanilla design and the final QDMA one, since the other QDMA designs have had a change at a time. All bitstreams were generated with the same synthesis and implementation settings, in the same machine and with the same Vivado version.

Table 6.3 contains the nomenclature for the different generated versions. Those abbreviated names will be used from now on.

Abbreviation name	Full version name
XDMA vanilla	Starting point of FPGA@SDV design with XDMA
QDMA Legacy + no rdy	QDMA with Legacy interrupts without ready and soft reset connected
QDMA Legacy + rdy	QDMA with Legacy interrupts with ready and soft reset connected
QDMA MSI	QDMA with MSI interrupts
QDMA MSIx	QDMA with MSIx interrupts
QDMA MSIx + sft rst	QDMA with MSIx interrupts and soft reset connected to PCIe reset

Table 6.3: Abbreviated and full name from the different versions tested and generated in this thesis.

6.3.1 Data burst performance

A benchmark was created to evaluate the actual performance of both PCIe subsystems obtaining the transfer time and bandwidth.

The base of this benchmark is the data word test explained in Section 6.2.8. That test has been expanded to send a buffer of N elements instead of a single word of 4 bytes and it is also written in C.

The input arguments for the test are the type of PCIe subsystem (`xdma` or `qdma`) and the number of buffer elements. Hence, the number of bytes sent and received is N *elements* $\times$ 4 *bytes/element*. The tests measures the time dedicated reading and writing only, not the whole execution. Listing 6.17 contains the execution and output of the Data Burst test binary for QDMA and 32 4-byte elements.

```
[13:34] aquerol@pickle-5 ~/fpga-platforms-portability/data_test
$ ./data_buffer_test qdma 32
Stating data test for QDMA:
Allocating and initializing test_buffer of 32 elements, 128 bytes
test 6b8b4567, test read 0
Writing 128 bytes to address 0x800000000000
Writing into FPGA
Written bytes: 128
Reading from FPGA
Read bytes: 128
Total unmatched word 0

Time_write 80 us

Time_read 36 us
```

Listing 6.17: Execution and output from the data buffer test in Pickle-5.

The code written to perform this test is in Listing A.2 found in Appendix A.2, together with a Bash script, in Listing A.3. That script was developed to execute the Data Burst test with different buffer sizes and extract the write and read time in CSV format by parsing the test output. The performance was obtained with different buffer sizes, from 2^1 elements (4 bytes) to 2^{28} elements (1 GB), incrementing by 1 the exponent each time.

Knowing the transfer time and bytes sent, the bandwidth (BW) is trivial to extract, since it is a derived metric. The formula to obtain it is $B = D/t$, where B is the bandwidth measured in *MB/s*, D is the data size (or buffer size) measured in *MB* (megabytes), and t is the time in s (seconds) measured for transferring the data through the PCIe.

The Data Burst test was executed 5 times with each buffer size for both XDMA and QDMA. As the standard deviation is less than 5%, the error bars are not shown in the plots below. Then, the extracted data was processed and the resulting data is shown in Table 6.4 for the XDMA design and in Table 6.5 for the QDMA one.

Plots with write-read time, bandwidth and latency were generated from that data. All plots contain in the X-axis the buffer size, which has been modified from the vanilla number (as in tables) to a more readable number, for example, from 131072 to 128K. Moreover, the color code is purple for write transfers and orange for read transfers, and a less saturated color for XDMA and more saturated for QDMA.

Buffer size (B)	Write		Read	
	Time (us)	BW (MB/s)	Time (us)	BW (MB/s)
4	54.4	0.1	19.6	0.207
8	47.8	0.2	21	0.383
16	60.0	0.3	24.6	0.679
32	55.4	0.6	38.2	0.904
64	56.8	1.3	36	1.979
128	54.6	2.57	31.6	4.30
256	68.0	4.21	40.8	6.69
512	63.2	8.55	40.2	13.68
1024	60.0	17.42	38.8	27.45
2048	92.8	22.55	59.0	35.25
4096	105.8	39.12	53.2	79.32
8192	171.8	47.95	92.6	91.22
16384	284.6	58.02	145.6	117.63
32768	481.8	68.06	218.4	150.73
65536	913.8	71.72	397.0	165.09
131072	1769.0	74.10	774.4	169.29
262144	3650.4	72.08	1647.8	162.46
524288	6944.4	75.54	3041.0	173.34
1048576	13654.8	76.79	5834.4	179.73
2097152	27311.6	76.79	11596.0	180.87
4194304	54525.8	76.92	23040.8	182.06
8388608	108719.2	77.16	45723.8	183.47
16777216	217246.2	77.23	91310.6	183.74
33554432	435208.2	77.10	183107.0	183.25
67108864	869991.0	77.14	365705.4	183.51
134217728	1738817.2	77.19	727934.8	184.38
268435456	3478165.6	77.18	1456436.0	184.31
536870912	6950200.2	77.25	2900827.6	185.08
1073741824	13895425.2	77.27	5801329.0	185.09

Table 6.4: XDMA data buffer results.

Buffer size (B)	Write			Read		
	Time (us)	BW (MB/s)	Speedup	Time (us)	BW (MB/s)	Speedup
4	26.2	0.168	2.08	15.20	0.27	1.29
8	18.6	0.4	2.57	13.80	0.58	1.52
16	18.8	0.9	3.19	13.00	1.23	1.89
32	21.0	1.5	2.64	14.60	2.22	2.62
64	20.8	3.1	2.73	13.80	4.64	2.61
128	22.4	5.7	2.44	14.60	8.79	2.16
256	22.8	11.3	2.98	14.60	17.55	2.79
512	25.6	20.1	2.47	16.20	31.67	2.48
1024	33.0	31.1	1.82	18.40	55.69	2.11
2048	46.8	43.8	1.98	23.20	89.19	2.54
4096	73.0	56.1	1.45	33.40	123.01	1.59
8192	123.2	66.5	1.39	55.60	147.50	1.67
16384	224.8	72.9	1.27	98.80	166.01	1.47
32768	430.8	76.1	1.12	184.60	177.53	1.18
65536	840.8	77.9	1.09	354.80	184.74	1.12
131072	1662.0	78.9	1.06	696.20	188.29	1.11
262144	3306.8	79.3	1.10	1377.60	190.29	1.20
524288	6599.0	79.4	1.05	2749.60	190.68	1.11
1048576	13183.8	79.5	1.04	5506.80	190.42	1.06
2097152	26358.6	79.6	1.04	10993.40	190.77	1.05
4194304	52673.2	79.6	1.04	21924.20	191.31	1.05
8388608	105261.2	79.7	1.03	43878.80	191.18	1.04
16777216	210475.2	79.7	1.03	87531.60	191.68	1.04
33554432	421516.2	79.6	1.03	175418.20	191.29	1.04
67108864	843725.0	79.5	1.03	352837.50	190.20	1.04
134217728	1688657.8	79.5	1.03	701372.20	191.36	1.04
268435456	3378207.0	79.5	1.03	1400786.40	191.63	1.04

Table 6.5: QDMA data buffer results and QDMA speedup respect the XDMA time results.

Time

Figure 6.19 contains the plot with write and read required time data. The Y-axis has time expressed in micro-seconds (μs) and it is in logarithmic scale. In this plot, the lower the time, the better.

Firstly, we can observe that reading is faster than writing, with both XDMA and QDMA. This behavior was predictable because of the write and read implementation, which was detailed in Section 4.6.2. Figure 4.18 and Figure 4.19 show the MSC diagram for write and read transfers, respectively. At first sight, it is noticeable that write transfers require one more message than read transfers. However, the key for the time difference is that write transfers are composed of two memory accesses by the IP, one to the host memory and another one to the FPGA memory, whereas for read transfers, the IP only accesses the FPGA memory.

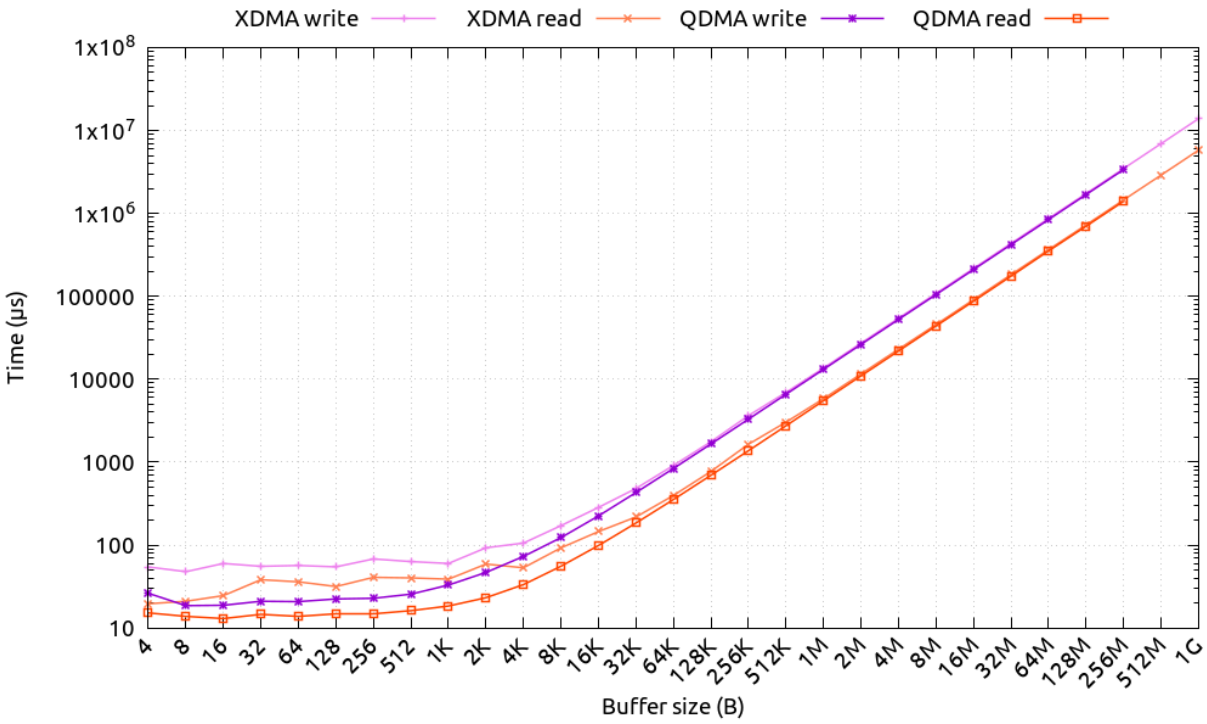


Figure 6.19: Plot comparing the time for write (purple) and read (orange) with XDMA and QDMA. The lower, the better.

Moreover, it is observed that QDMA transfers take less time than XDMA ones. The QDMA IP is newer than the XDMA IP, which could be the reason behind QDMA being faster. As IPs' source code is private, it is not possible to evaluate the difference at RTL level.

Bandwidth

Figure 6.20 contains the plot with write and read bandwidth data. The Y-axis has bandwidth expressed in megabytes per second (MB/s). We can see that QDMA read transfers are the ones with higher bandwidth, as expected from the previous time results analysis.

Additionally, the distance between an XDMA transfer and a QDMA one, for both write and read, is bigger during the part of the curve with a higher slope. This will soon be analyzed in more detail.

All four lines end up stabilizing to a constant bandwidth, which shows that the maximum bandwidth that the IP and driver can offer is reacheable. Table 6.6 contains the maximum bandwidth range and at which buffer size it is reached, in other words, the minimum buffer size needed to get the maximum bandwidth. It is observed that the QDMA subsystem obtains a higher bandwidth earlier than the XDMA one.

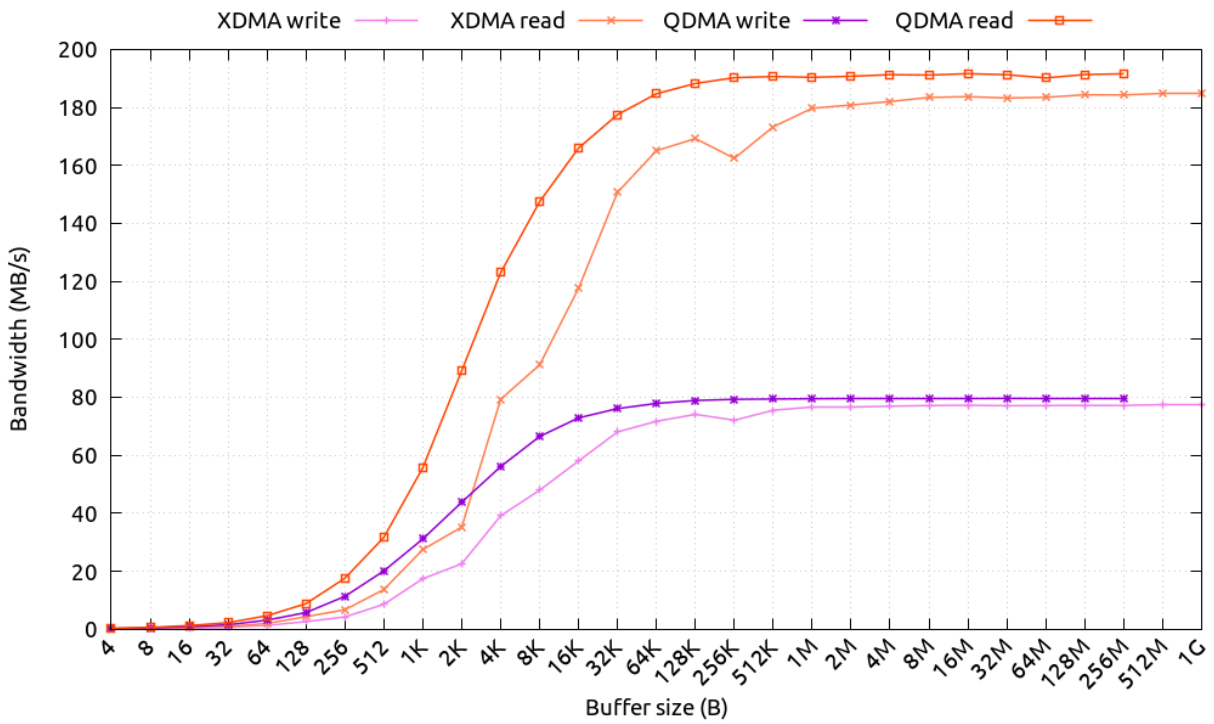


Figure 6.20: Plot comparing the bandwidth for write (purple) and read (orange) with XDMA and QDMA. The higher, the better.

Transfer	Size (B)	Bandwidth range (MB/s)
XDMA write	128K	74-77.3
QDMA write	64K	78-79.5
XDMA read	4M	182-185
QDMA read	128K	188-192

Table 6.6: Minimum buffer size to reach the maximum bandwidth range per transfer type and PCIe subsystem.

Latency

Figure 6.21 contains the plot with write and read time data for buffer sizes from 4 to 1024. Its objective is to study the latency of the PCIe subsystems. The Y-axis has time expressed in micro-seconds (μs).

The plot shows that there is more time variability per transfer type with the XDMA transfers, whereas QDMA transfers are more stable. Moreover, the XDMA read requires more time than the QDMA write. That is curious since the global trend is that reads are faster than writes, as has been observed previously. Hence, ti seems Xilinx has improved the performance of small transfers in its newer PCIe subsystem (QDMA).

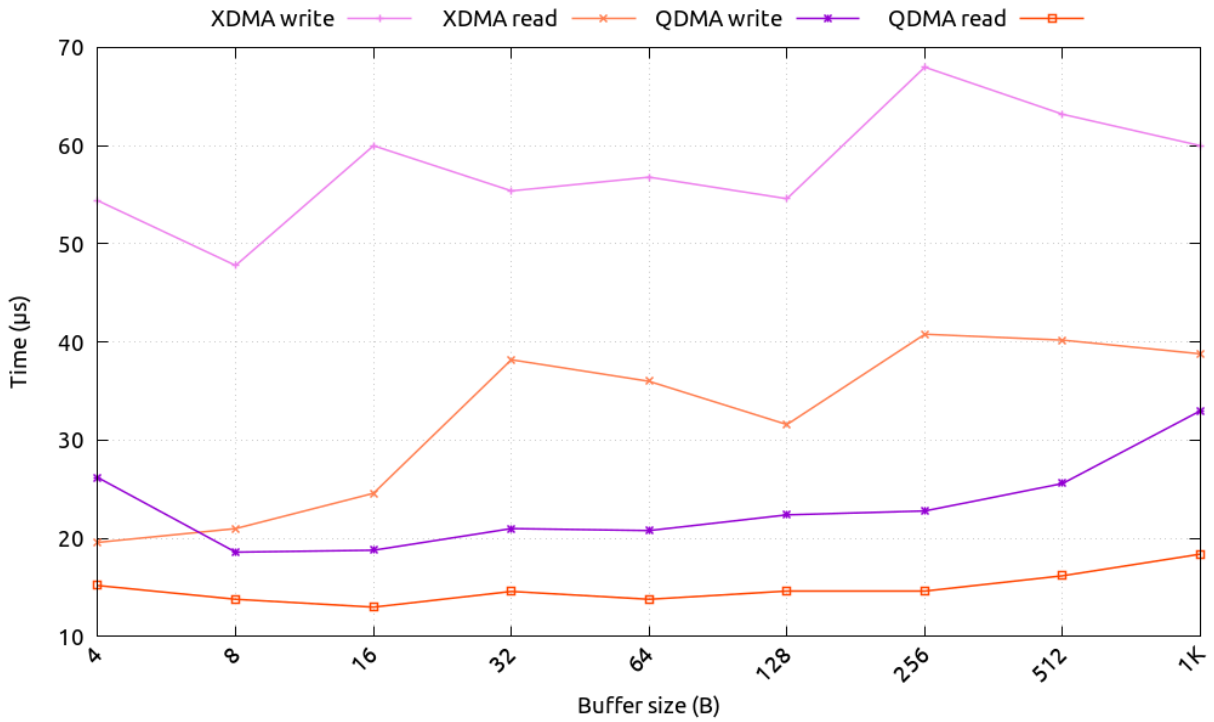


Figure 6.21: Plot comparing the latency (expressed with time) for write (purple) and read (orange) with XDMA and QDMA. The lower, the better.

Speedup

The *speedup* columns from Table 6.5, the third column of each transfer type, contain the XDMA/QDMA speedup. It is computed by dividing the XDMA time results by the QDMA time ones. We can observe that for write transfers the maximum speedup is 3.19x, at 16 B, and for read transfers, it is 2.79x, at 256 B. Moreover, the speedup is higher with smaller buffer sizes, which could be because Xilinx had optimized the QDMA IP for smaller buffer sizes.

Preliminary conclusions

The QDMA IP has a better performance than the XDMA one, with a minimum speedup of 1.04x for bigger data transfers and up to 3.19x with smaller data transfers. In addition, based on the latency analysis, it seems QDMA is improved especially for smaller buffer sizes. Even though this is a supposition substantiated by the previous results because the Xilinx IP code is privative.

Independently of the PCIe subsystem, read transfers are faster than write transfers.

6.3.2 Boot time

The next and final performance test was booting Linux with the EPAC tool, to offload the Linux image into the FPGA DDR4 memory. The time that takes the tool to offload the OS binaries was measured with the `time` command. `time` runs the program specified afterwards and when it finishes, `time` displays the real, user and system execution time of the program ran [38].

The same image was used for both XDMA and QDMA bitstreams. The Linux binaries were sent 10 times for each bitstream and since the variability is below 2% I ignored the study of the errors of these measurements. The average time was computed, as can be seen in Table 6.7.

As could be expected from the results of the previous performance tests, the QDMA design needs less time to offload the Linux image. QDMA design is 13% faster than XDMA design.

Although 1.13x of speedup may not seem a lot compared with the results of the previous section, it is important keeping in mind that here the whole boot offloading process is measured, whereas in the previous section, only the read and write times are measured and not the whole program execution.

PCIe subsystem	Average Offloading time (s)	Speedup
XDMA	3.054	1.00x
QDMA	2.703	1.13x

Table 6.7: Average time needed to offload the Linux image.

6.3.3 Resource utilization

The first metric extracted from Vivado is resource utilization. It has been gathered after the implementation and from all the available versions with a bitstream generated. This metric is important because LUTs are the critical resource in the EPAC design and as EPI developers add microarchitectural features, it is going to keep increasing and difficulting closing timing. Therefore, if an important increase in LUTs was observed, it would imply less room for future features and probably more problems closing timing.

The data was extracted from one execution because *i)* it takes a significant amount of time and machine resources to generate each bitstream and *ii)* I observed from my previous experience with Vivado that the resource utilization variation from one run to another is not significant ($< 0.5\%$). Therefore, it was decided that it was not necessary performing various runs from the same version, so no machine resources were wasted. This reasoning also applies to the next Vivado metrics, as all of them do not present a significant variance between runs with the same design.

The following resource utilization tables are similar to Vivado resource tables (explained in Section 3.4.4). The main difference between them is that the resource total number and percentage of each module are both expressed one next to the other.

Before proceeding to the resource usage analysis, Table 6.8 contains the configuration reference for each version.

Version	Table reference
XDMA	Table 6.9
QDMA Legacy + no rdy	Table 6.10
QDMA Legacy + rdy	Table 6.11
QDMA MSI	Table 6.12
QDMA MSIx	Table 6.13
QDMA MSIx + soft rst as rst	Table 6.14

Table 6.8: Table reference for each tested version.

Tables are composed by the resource usage of the whole BD (`design_1`) and by the PCIe subsystem IP and its submodules. Both PCIe subsystems are formed by a PCIe IP (`pcie4c_ip`) and an RTL wrapper (`udma_wrapper` for XDMA and `rtl_wrapper` for QDMA). Moreover, the XDMA IP has a `ram_top` instance that QDMA does not have.

Name	CLB LUTs		CLB Registers		BRAM		URAM		DSPs	
	Num	%	Num	%	Num	%	Num	%	Num	%
design_1	86159	6.61%	116596	4.47%	104.5	5.18%	0	0.00%	3	0.03%
↦ xdma_0	30401	2.33%	36786	1.41%	50	2.48%	0	0.00%	0	0.00%
⟼ pcie4c_ip	2685	0.21%	7582	0.29%	22	1.09%	0	0.00%	0	0.00%
⟼ udma_wrapper	27716	2.13%	29203	1.12%	4	0.20%	0	0.00%	0	0.00%
⟼ ram_top	1	0.00%	1	0.00%	24	1.19%	0	0.00%	0	0.00%

Table 6.9: Resource utilization of the BD and XDMA IP in absolute and percentage number.

Name	CLB LUTs		CLB Registers		BRAM		URAM		DSPs	
	Num	%	Num	%	Num	%	Num	%	Num	%
design_1	90670	6.95%	121189	4.65%	106.5	5.28%	5	0.52%	3	0.03%
↦ qdma_0	35046	2.69%	41340	1.59%	52	2.58%	5	0.52%	0	0.00%
⟼ pcie4c_ip	2694	0.21%	6139	0.24%	20	1.09%	0	0.00%	0	0.00%
⟼ rtl_wrapper	32353	2.48%	35201	1.35%	30	1.49%	5	0.52%	0	0.00%

Table 6.10: Resource utilization of the BD and QDMA IP with legacy interrupts in absolute and percentage number.

Name	CLB LUTs		CLB Registers		BRAM		URAM		DSPs	
	Num	%	Num	%	Num	%	Num	%	Num	%
design_1	91029	6.98%	121188	4.65%	106.5	5.28%	5	0.52%	3	0.03%
↦ qdma_0	35102	2.69%	41342	1.59%	52	2.58%	5	0.52%	0	0.00%
⟼ pcie4c_ip	2699	0.21%	6139	0.24%	22	1.09%	0	0.00%	0	0.00%
⟼ rtl_wrapper	32403	2.49%	35203	1.35%	30	1.49%	5	0.52%	0	0.00%

Table 6.11: Resource utilization of the BD and QDMA IP with legacy interrupts and connected ready signals in absolute and percentage number.

Name	CLB LUTs		CLB Registers		BRAM		URAM		DSPs	
	Num	%	Num	%	Num	%	Num	%	Num	%
design_1	90866	6.97%	121193	4.65%	106.5	5.28%	5	0.52%	3	0.03%
↦ qdma_0	35054	2.69%	41336	1.59%	52	2.58%	5	0.52%	0	0.00%
⟼ pcie4c_ip	2691	0.21%	6139	0.24%	22	1.09%	0	0.00%	0	0.00%
⟼ rtl_wrapper	32363	2.48%	35197	1.35%	30	1.49%	5	0.52%	0	0.00%

Table 6.12: Resource utilization of the BD and QDMA IP with MSI interrupts in absolute and percentage number.

Name	CLB LUTs		CLB Registers		BRAM		URAM		DSPs	
	Num	%	Num	%	Num	%	Num	%	Num	%
design_1	90965	6.98%	121195	4.65%	106.5	5.28%	5	0.52%	3	0.03%
↦ qdma_0	35017	2.69%	41344	1.59%	52	2.58%	5	0.52%	0	0.00%
⟼ pcie4c_ip	2690	0.21%	6139	0.24%	22	1.09%	0	0.00%	0	0.00%
⟼ rtl_wrapper	32328	2.48%	35205	1.35%	30	1.49%	5	0.52%	0	0.00%

Table 6.13: Resource utilization of the BD and QDMA IP with MSIx interrupts in absolute and percentage number.

Name	CLB LUTs		CLB Registers		BRAM		URAM		DSPs	
	Num	%	Num	%	Num	%	Num	%	Num	%
design_1	91051	6.98%	121171	4.65%	106.5	5.28%	5	0.52%	3	0.03%
↦ qdma_0	35045	2.69%	41338	1.59%	52	2.58%	5	0.52%	0	0.00%
⟼ pcie4c_ip	2695	0.21%	6139	0.24%	22	1.09%	0	0.00%	0	0.00%
⟼ rtl_wrapper	32350	2.48%	35199	1.35%	30	1.49%	5	0.52%	0	0.00%

Table 6.14: Resource utilization of the BD and QDMA IP with MSIx interrupts and soft reset connected to PCIe reset in absolute and percentage number.

Analyzing those tables, a comparison between XDMA and QDMA IPs is performed. The fewer resources are required, the better, as it potentially leaves more room for the implementation algorithms to close timing.

The first noticeable fact was that both QDMA and XDMA IP instances operate without DSPs. The QDMA IP incorporates 5 URAMs, while XDMA has none. Furthermore, QDMA design utilizes 52 BRAMs, slightly more than XDMA's 50 BRAMs.

Regarding `pcie4c_ip`, this IP uses fewer FFs with QDMA than with XDMA, 0.24% and 0.29% respectively. The LUT usage is stable at 0.21% for both IPs.

The main difference between these 2 IPs is the wrapper. The XDMA one utilizes 2.13% LUTs and 1.12% FFs, while the QDMA wrapper uses 2.48% LUTs and 1.35%.

Comparing the different QDMA versions, the LUT difference is at most 0.01%. The LUT usage range is from 32328 (MSI-X) to 32403 (Legacy with ready signals). The final QDMA version, MSI-X with soft reset, is the second QDMA design with fewer LUTs utilization.

Therefore, XDMA IP requires fewer resources in general than QDMA. However, keeping in mind that LUTs are the only critical FPGA resource, QDMA does not need so much more LUTs than XDMA, as the difference is 0.35% LUTs.

LUT utilization of each version ordered by less to more usage:

$$\text{XDMA} < \text{MSIx} < \text{MSIx + sft rst} < \text{Legacy + no rdy} < \text{Legacy + rdy} < \text{MSI}$$

6.3.4 Synthesis and implementation time

The next Vivado metric is the time that Vivado needed to perform the synthesis and implementation. This metric has been taken because the XDMA vanilla bitstream already requires a significant amount of time to be generated. Hence, a big increase in the bitstream generation time would not be ideal.

Table 6.15 holds the synthesis, implementation and total time dedicated to generating a bitstream. The table has these data for the XDMA vanilla version and each QDMA version. As a reminder, the lower the time, the better.

PCIe subsystem	Version	Synth time	Impl time	Total time
XDMA	Vanilla	0:44:18	3:44:39	4:28:57
QDMA	Legacy + no rdy	0:53:03	2:36:37	3:29:40
	Legacy + rdy	1:03:18	3:54:41	4:57:59
	MSI	0:44:52	3:53:44	4:38:36
	MSIx	1:00:15	3:55:31	4:55:46
	MSIx + soft rst as rst	1:00:15	4:00:50	5:01:05

Table 6.15: Vivado synthesis, implementation and total time required by XDMA and each QDMA version.

Comparing the XDMA and QDMA Legacy + no rdy designs, the QDMA total time is lower than the XDMA. Although XDMA synthesis time is lower by $\tilde{9}$ minutes, there is a significant difference in the implementation time of 1 hour and 8 minutes. That is probably because of the seed used in the Vivado heuristic algorithms. The seed choice procedure is not known by us, but from previous experience, it could be design dependent. That means that the seed could be computed by, for example, performing a hash over the code, hence, a change in the code implies a change in the seed.

Regarding the QDMA designs, it can be noted that the fastest one is the first version, legacy interrupts without the ready and soft reset signals connected. In the next version, where the mentioned signals are connected to a block that generates a constant, the total time increases by around 40% with respect to the previous design. It is probably due to having to connect 3 signals to a fixed value, increasing the netlist and resources, which rises the complexity in the synthesis and implementation phases. The other QDMA versions have a similar implementation time, but the synthesis is faster for MSI interrupts.

The total time difference between the XDMA vanilla design and the final QDMA one is around 32 minutes, an increase of 11.95% . It can be a consequence of the slight resource usage increase, in other words, a raising in the design complexity is translated into more synthesis and implementation time.

Total time of each version ordered by less to more time:

Legacy + no rdy < XDMA < MSI < MSIx < Legacy + rdy < MSIx + sft reset

6.3.5 WNS

The last Vivado metric is the WNS. It is key because currently closing EPAC timing can be a problem with some configurations, so having a lower value would not be the ideal case.

Table 6.16 shows the different WNS reported by Vivado. Just as a reminder, the higher the value, the better, as it means that the tool has a higher scope of action in the implementation phase.

PCIe subsystem	Version	WNS (ns)
XDMA	Vanilla	0.093
QDMA	Legacy + no rdy	0.094
	Legacy + rdy	0.074
	MSI	0.058
	MSIx	0.071
	MSIx + soft rst as rst	0.098

Table 6.16: WNS of the XDMA vanilla design and each QDMA design version.

The best WNS is from the final QDMA version, which is even better than the XDMA design by 5.38%. The worst WNS is from the MSI design (0.058).

Regarding the QDMA legacy versions, WNS gets worse with the ready signals connected, from 0.094 to 0.074.

Observing the QDMA MSI-X designs, WNS improves by having the soft reset connected to the PCIe reset. That could be because the placing and routing internal constraints of connecting signals to a constant could be strict, therefore having to connect 2 signals to a constant instead than 3, loosens up the constraints and helps closing timing.

WNS of each version ordered by more to less value:

MSIx sft reset > Legacy + no rdy > XDMA > Legacy + rdy > MSIx > MSI

Chapter 7

Conclusions

At the end of this thesis, it is important answering the research questions laid out in Section 2.4.

Question 1. Does QDMA improve the performance over XDMA?

The response to the first question is yes, QDMA improves performance over XDMA. This was observed analyzing the data buffer performance and the boot time, in Section 6.3.1 and Section 6.3.2, respectively. QDMA has a better bandwidth and it seems to be optimized specially for smaller packet sizes, which is the usual use case in our project.

Question 2. Which impact has QDMA over the design compared with XDMA? And over the software tools?

Regarding the first part of the second question, the QDMA impact on the design is that it is worth because QDMA improves the WNS although it occupies 0.36% more LUTs and requires more bitstream generation time. The main problem we are currently facing in the SDV project is to close timing as the EPAC design uses a lot of LUTs and achieving a positive WNS is becoming more difficult. Therefore, having a new design that improves the WNS eases the timing issue we are facing.

With respect to the second part of the question, the change in the software tools used is minimal and transparent to the user as they can continue using the boot and reprogram tools. However, the QDMA driver load is more complex than the XDMA one and QDMA needs configuring queues, which is a tedious process. These matters have been targeted by creating a script that prepares the environment and by including the QDMA design in the `fpga-sdv` partition, which ends up with a fully functional FPGA ready to be used (Section 6.2.14).

Question 3. Is replacing XDMA with QDMA worth for the project?

Finally, the third question reply is yes, replacing XDMA with QDMA is beneficial for the FPGA@SDV project because it is better from both the performance and Vivado perspective.

Apart from the answers to the research questions, this thesis has allowed me to learn the whole process of understanding, changing, debugging and evaluating an IP. Moreover, I have been able to make use of the theoretical driver concepts I had, learning the driver

load and configuration procedure and the different Linux commands that allow you to manage a driver. I have expanded my limited knowledge about DMA and learnt how it can be applied with PCIe, apart from getting to know RDMA, which I had not heard before this thesis. Furthermore, it has been an opportunity to work with two state-of-the-art FPGA boards commercially available: VCU128, which targets RTL development as it provides more interfaces, and Alveo U55C, which aims data centers and offers less interfaces. Working with them has given me an insight of the difficulties of managing powerful FPGAs with large amount of resources, having the ability to make really complex designs.

7.1 Future work

Having a functional and stable QDMA design opens the possibility of implemeting various improvements on the FPGA@SDV design.

7.1.1 QDMA at Xilinx Alveo U55C

As was explained in Section 2.2, the MEEP cluster is going to be available by the end of June 2023 and we will be able to upgrade the current U55C FPGA@SDV design from using XDMA to QDMA. The experience obtained in this thesis with a known board and environment will allow for a faster transition from XDMA to QDMA in a new FPGA (Alveo U55C).

7.1.2 Ethernet over PCIe

One opportunity is to apply the work in progress of Ethernet over PCIe. It consists of sending Ethernet packets via PCIe, rather than through the Ethernet physical interface. It is currently being developed by The OmpSs Programming Model group at BSC[1], led by Xavier Martorell. They have implemented two drivers: one for the host side and another for the processor emulated in the FPGA.

The main prerequisite to be able to use their Ethernet over PCIe implementation is having QDMA as PCIe subsystem. That is because the host driver uses the QDMA queue system to send Ethernet packets to the FPGA.

Therefore, with the XDMA design the possibility of utilizing their work in SDV was nonexistent and now it is the next objective.

7.1.3 Custom ILA

Another opportunity is implementing ourselves an ILA for the FPGA@SDV design. Currently, the ILA size is limited by the timing issues that we encounter occasionally. At the

[1] https://www.bsc.es/research-development/research-areas/programming-models/the-ompss-programming-model/people

moment, the ILA is used to debug and extract performance metrics from benchmarks and scientific applications that are executed at FPGA@SDV. One request we usually receive from the users who gather those metrics is being able to trace more execution time, and that is determined by the ILA window.

By implementing a custom ILA we want to resolve that petition. The idea is to collect the necessary information from signals or events and store it in either BRAMs or HBM memory. Once that storage space is full, it would be flushed, written, to the host memory via the QDMA. This could be implemented with XDMA, but as has been seen, QDMA IP offers more bandwidth specially for smaller transfers, which is critical for this idea. Bandwidth is key because it determines how long it takes to offload the information, which affects the buffer size needed between the custom ILA storage and the PCIe subsystem to avoid losing information from the execution that might not be stopped.

Acronyms

ALU Arithmetic Logic Unit.

ASIC Application-Specific Integrated Circuits.

AXI Advanced eXtensible Interface.

AXI-MM AXI Memory-Mapped.

AXI-ST AXI4-Stream.

BD Block Design.

BRAM Block RAM.

BSC Barcelona Supercomputing Center.

C2H card-to-host.

DMA Direct Memory Addressing.

DRC Design Rule Check.

DSP Digital Signal Processor.

EPAC European Processor Accelerator.

EPI European Processor Initiative.

FF Flip-Flop.

FPGA Field Programmable Gate Array.

GPIO General Purpose I/O.

H2C host-to-card.

HBM High Memory Bandwidth.

HDL Hardware Description Language.

HPC High-Performance Computing.

ILA Integrated Logic Analyzer.

IO Input/Output.

IP Intellectual Property.

ISA Instruction Set Architecture.

LUT Look-Up Table.

MEEP MareNostrum Experimental Exascale Platform.

MM Memory-Mapped.

MSC Message Sequence Chart.

MSI Message Signal Interface.

MSI-X MSI eXtended.

NIC Network Interface Card.

OS Operating System.

PCI Peripheral Component Interconnect.

PCIe Peripheral Component Interconnect Express.

PF Physical Function.

QDMA Queue DMA.

QP Queue Pair.

RC Requester Completion.

RDMA Remote Direct Memory Addressing.

RQ Requester reQuest.

RTL Register-Transfer Level.

SDV Software Development Vehicles.

SoC System-on-Chip.

SR-IOV Single Root I/O Virtualization.

URAM Ultra RAM.

VF Virtual Function.

VM Virtual Machine.

VPU Vector Processor Unit.

WNS Worst Negative Slack.

XDC Xilinx Design Constraints.

XDMA Xilinx DMA.

Appendices

Appendix A

Dummy tests

This appendix contains the source code for the data tests, both word and buffer, and the Makefile to compile them. I have written all the codes shown below.

A.1 Dummy word

Listing A.1 contains the source code from the data word test.

```
#include <inttypes.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>


int write_word (int card_fd, uint32_t data, int size, uint64_t addr) {
    int retval = 0;

    retval = lseek(card_fd, addr, SEEK_SET);
    if (retval < 0) {
        perror("Write lseek error");
        return -1;
    }

    retval = write(card_fd, &data, size);
    if (retval != size) {
        perror("Write error");
        return -1;
    }

    return retval;
}

```

```
30 int read_word (int card_fd, uint32_t * data, int size, uint64_t addr) {
31     int retval = 0;
32 
33     retval = lseek(card_fd, addr, SEEK_SET);
34     if (retval < 0) {
35         perror("Read lseek error");
36         return -1;
37     }
38 
39     retval = read(card_fd, data, size);
40     printf("Read bytes: %d\n", retval);
41     if (retval != size) {
42         perror("Read error");
43         return -1;
44     }
45 
46     return retval;
47 
48 }
49 
50 int main (int argc, char *argv[]) {
51 
52     if (argc != 2) {
53         printf("Usage: ./dummy_word <xdma | qdma>\n");
54         return -1;
55     }
56 
57     /*
58      ******************************************
59      * Open devices                           *
60      ******************************************
61      */
62     char *device_h2c;
63     char *device_c2h;
64     if (strcmp(argv[1], "xdma") == 0) {
65         printf("Stating dummy test for XDMA:\n");
66 
67         // XDMA h2c and c2h channel 0
68         device_h2c = "/dev/xdma0_h2c_0";
69         device_c2h = "/dev/xdma0_c2h_0";
70     }
71     else if (strcmp(argv[1], "qdma") == 0) {
72         printf("Stating dummy test for QDMA:\n");
73 
74         // QDMA h2c and c2h channel 0
75         device_h2c = "/dev/qdma08000-MM-0";
76         device_c2h = "/dev/qdma08000-MM-0";
77     }
78     else {
79         printf("Non of the available devices specified: xdma or qdma\n")
               ;
80         return 1;
81     }
82 
83     int fpga_write_fd = open(device_h2c, O_RDWR);
84     int fpga_read_fd = open(device_c2h, O_RDWR);
```

```
85
86      if (fpga_write_fd < 0) {
87          printf("H2C channel device could not be opened\n");
88          perror("Opening H2C device");
89          exit(-1);
90      }
91      if (fpga_read_fd < 0) {
92          printf("C2H channel device could not be opened\n");
93          perror("Opening C2H device");
94          exit(-1);
95      }
96
97      /*
98      ******************************************
99      * Writing a word                         *
100     ******************************************
101     */
102     uint32_t test_word = 0xABCDEF12;
103     int size = sizeof(uint32_t);
104
105     int written_bytes;
106     /* uint64_t axi_addr = 0x44A20008; */
107     uint64_t axi_addr = 0x800000000000;
108
109     fprintf(stdout, "Writing data 0x%x to address 0x%lx\n", test_word,
            axi_addr);
110     printf("Writing into FPGA\n");
111
112     written_bytes = write_word(fpga_write_fd, test_word, size, axi_addr)
            ;
113
114     printf("Written bytes: %d\n", written_bytes);
115
116     if (written_bytes < 0)
117         goto exit;
118
119     /*
120     ******************************************
121     * Reading a word                         *
122     ******************************************
123     */
124
125     uint64_t read_bytes;
126     uint32_t read_test_word;
127     printf("Reading from FPGA\n");
128     read_bytes = read_word(fpga_read_fd, &read_test_word, size, axi_addr
            );
129
130     printf("Read bytes: %ld\n", read_bytes);
131     if (read_bytes < 1)
132         goto exit;
133
134
135     int ret_val = 0;
136     if (read_test_word != test_word) {
```

```
137         printf("Read word %x is DIFFERENT than the test word %x \n",
                read_test_word, test_word);
138         ret_val = -1;
139     }
140     else
141         printf("Read word %x is EQUAL than the test word %x \n",
                read_test_word, test_word);
142
143
144
145
146 exit:
147     close(fpga_write_fd);
148     close(fpga_read_fd);
149     return ret_val;
150 }
```

Listing A.1: Data word test source code written in C.

A.2 Dummy buffer

Listing A.2 shows the source code from the data buffer test.

```
1  #include <inttypes.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <stdint.h>
6  #include <stdio.h>
7  #include <errno.h>
8  #include <fcntl.h>
9  #include <sys/time.h>
10 #include <time.h>
11 #include <string.h>
12
13
14 #define RW_MAX_SIZE 0x7ffff000
15
16 int write_buffer (int card_fd, uint32_t *data, uint64_t size, uint64_t
       addr) {
17     int64_t ret_val = 0;
18     uint64_t count = 0;
19     uint32_t *buffer = data;
20     uint64_t offset = addr;
21
22     while (count < size) {
23         uint64_t bytes = size - count;
24
25         if (bytes > RW_MAX_SIZE)
26             bytes = RW_MAX_SIZE;
27
28         ret_val = lseek(card_fd, offset, SEEK_SET);
```

```
29          if (ret_val == -1) {
30              perror("Write lseek error");
31              return -1;
32          }
33
34          ret_val = write(card_fd, buffer, bytes);
35          if (ret_val != bytes) {
36              fprintf(stderr, "Write 0x%lx @ 0x%lx failed %ld.\n", bytes,
                    offset, ret_val);
37              perror("Write file");
38              return -1;
39          }
40
41          count += ret_val;
42          buffer += bytes;
43          offset += bytes;
44      }
45
46      return count;
47  }
48
49  int read_buffer (int card_fd, uint32_t * data, uint64_t size, uint64_t
        addr) {
50      int64_t ret_val = 0;
51      uint64_t count = 0;
52      uint32_t *buffer = data;
53      uint64_t offset = addr;
54
55      while (count < size) {
56          uint64_t bytes = size - count;
57
58          if (bytes > RW_MAX_SIZE)
59              bytes = RW_MAX_SIZE;
60
61          ret_val = lseek(card_fd, offset, SEEK_SET);
62          if (ret_val == -1) {
63              perror("Read lseek error");
64              return -1;
65          }
66
67          ret_val = read(card_fd, buffer, bytes);
68          if (ret_val != bytes) {
69              fprintf(stderr, "Read 0x%lx @ 0x%lx failed %ld.\n", bytes,
                    offset, ret_val);
70              perror("Read file");
71              return -1;
72          }
73
74          count += ret_val;
75          buffer += bytes;
76          offset += bytes;
77      }
78
79      return count;
80  }
81
```

```
82  int main (int argc, char *argv[]) {
83
84      if (argc != 3) {
85          printf("Usage: ./dummy_buffer <xdma | qdma> <buffer_elements>\n"
                );
86          return -1;
87      }
88
89      /*
90       ****************************************
91       * Open devices                         *
92       ****************************************
93       */
94      char *device_h2c;
95      char *device_c2h;
96      if (strcmp(argv[1], "xdma") == 0) {
97          printf("Stating dummy test for XDMA:\n");
98
99          // XDMA h2c and c2h channel 0
100         device_h2c = "/dev/xdma0_h2c_0";
101         device_c2h = "/dev/xdma0_c2h_0";
102     }
103     else if (strcmp(argv[1], "qdma") == 0) {
104         printf("Stating dummy test for QDMA:\n");
105
106         // QDMA h2c and c2h channel 0
107         device_h2c = "/dev/qdma08000-MM-0";
108         device_c2h = "/dev/qdma08000-MM-0";
109     }
110     else {
111         printf("Non of the available devices specified: xdma or qdma\n")
                ;
112         return 1;
113     }
114
115     int fpga_write_fd = open(device_h2c, O_RDWR);
116     int fpga_read_fd = open(device_c2h, O_RDWR);
117
118     if (fpga_write_fd < 0) {
119         printf("H2C channel device could not be opened\n");
120         perror("Opening H2C device");
121         exit(-1);
122     }
123     if (fpga_read_fd < 0) {
124         printf("C2H channel device could not be opened\n");
125         perror("Opening C2H device");
126         exit(-1);
127     }
128
129     int ret_val = 0;
130     struct timeval t0, t1;
131     uint64_t time_write, time_read;
132
133
134     /*
135     ****************************************
```

```
136     * Allocate buffers                          *
137     *********************************************
138     */
139     uint64_t num_elem = strtoull(argv[2], NULL, 10);
140     int32_t *test_buffer, *test_buffer_read;
141 
142     uint64_t size = num_elem * sizeof(int32_t);
143 
144     printf("Allocating and initializing test_buffer of %lu elements, %lu
            bytes\n", num_elem, size);
145 
146     /* posix_memalign((void **)&buffer, 4096 /*alignment *1/ , size +
           4096); */
147     test_buffer = malloc(size);
148     test_buffer_read = malloc(size);
149 
150     for (int i = 0; i < num_elem; i++) {
151         test_buffer[i] = rand();
152     }
153     printf("test %x, test read %x\n", test_buffer[0], test_buffer_read
           [0]);
154 
155     /*
156     *********************************************
157     * Writing a buffer                           *
158     *********************************************
159     */
160     int64_t written_bytes;
161     /* uint64_t axi_addr = 0x44A20008; */
162     uint64_t axi_addr = 0x800000000000;
163 
164     fprintf(stdout, "Writing %lu bytes to address 0x%lx\n", size,
           axi_addr);
165     printf("Writing into FPGA\n");
166 
167     gettimeofday(&t0, NULL);
168     written_bytes = write_buffer(fpga_write_fd, test_buffer, size,
           axi_addr);
169     gettimeofday(&t1, NULL);
170 
171     time_write = (t1.tv_sec-t0.tv_sec)*1000000 + t1.tv_usec-t0.tv_usec;
172 
173     printf("Written bytes: %ld\n", written_bytes);
174 
175     if (written_bytes < 0) {
176         ret_val = -1;
177         goto exit;
178     }
179 
180     /*
181     *********************************************
182     * Reading a buffer                           *
183     *********************************************
184     */
185 
186     int64_t read_bytes;
```

```
187     printf("Reading from FPGA\n");
188     gettimeofday(&t0, NULL);
189     read_bytes = read_buffer(fpga_read_fd, test_buffer_read, size,
            axi_addr);
190     gettimeofday(&t1, NULL);
191
192     time_read = (t1.tv_sec-t0.tv_sec)*1000000 + t1.tv_usec-t0.tv_usec;
193
194     printf("Read bytes: %ld\n", read_bytes);
195     if (read_bytes < 0) {
196         ret_val = -1;
197         goto exit;
198     }
199
200     int unmatched_words = 0;
201     for (int i = 0; i < num_elem; i++) {
202         /* printf("test_buffer[%d] = %x; test_buffer_read[%d] = %x\n", i
                , test_buffer[i], i, test_buffer_read[i]); */
203         if (test_buffer[i] != test_buffer_read[i]) {
204             unmatched_words++;
205             printf("In position %d -> test_buffer = %x !=
                    test_buffer_read = %x\n", i, test_buffer[i],
                    test_buffer_read[i]);
206         }
207     }
208
209     printf("Total unmatched word %d\n", unmatched_words);
210
211     printf("\nTime_write %lu us\n", time_write);
212     printf("\nTime_read %lu us\n", time_read);
213
214     ret_val = unmatched_words ? -1 : 0;
215
216 exit:
217     close(fpga_write_fd);
218     close(fpga_read_fd);
219     free(test_buffer);
220     free(test_buffer_read);
221     return ret_val;
222 }
```

Listing A.2: Dummy buffer test source code written in C.

Listing A.3 contains the Bash script to execute the data buffer test with different buffer sizes and extract the time results.

```
1  #!/bin/bash
2
3  echo "buf_elems, buf_size(B), run, write_time(us), read_time(us)"
4
5  # max_elems=$(( 2 << 29 ))
6
7  for j in {0..29}
8  do
9      buf_elems=$(( 2 ** $j ))
```

```
10
11      for i in {1..5}
12      do
13          tmp=`./dummy_buffer_test $buf_elems`
14          # ./dummy_buffer_test $buf_elems
15
16          if [ $? -ne 0 ]; then
17              echo "Fail"
18              continue
19          fi
20
21          buf_size=$(( $buf_elems * 4 ))
22
23          write_time=$(awk '/Time_write/ {print $2}' <<< $tmp)
24          read_time=$(awk '/Time_read/ {print $2}' <<< $tmp)
25
26          echo $buf_elems, $buf_size, $i, $write_time, $read_time
27      done
28  done
```

Listing A.3: Bash script to execute and extract in CSV form the time results from the data buffer tests.

A.3 Makefile

Listing A.4 contains the Makefile written to compile the two data tests.

```
1  CC ?= gcc
2  CFLAGS+="-D_FILE_OFFSET_BITS=64 -D_GNU_SOURCE -D_LARGE_FILE_SOURCE"
3
4  .PHONY: all clean
5
6  all: data_word_test data_buffer_test
7
8
9  data_word_test: data_word.c
10     $(CC) $(CFLAGS) -o $@ $<
11
12 data_buffer_test: data_buffer.c
13     $(CC) $(CFLAGS) -o $@ $<
14
15 clean:
16     rm -rf data_test data_word_test data_buffer_test
```

Listing A.4: Makefile to compile the 2 data tests.

Bibliography

[1] European Processor Initiative. European processor iniciative - epi. https://www.european-processor-initiative.eu/, 2023. Accessed on May 24, 2023. 2.1

[2] European Processor Initiative. Accelerator processor stream. https://www.european-processor-initiative.eu/accelerator/, 2023. Accessed on May 24, 2023. 2.1.1

[3] Inc. Xilinx. Virtex ultrascale+ hbm vcu128 fpga evaluation kit. https://www.xilinx.com/products/boards-and-kits/vcu128.html, 2023. Accessed on March 22, 2023. 2.1.1, 3.1, 3.3

[4] Barcelona Supercomputer Center. Meep: Marenostrum experimental exascale platform. https://www.bsc.es/research-and-development/projects/meep-marenostrum-experimental-exascale-platform, 2023. Accessed on June 14, 2023. 2.2

[5] Inc. Xilinx. What is an fpga? https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html. Accessed on March 22, 2023. 3.1

[6] Inc. Xilinx. Lut. https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/yeo1504034293627.html, 2018. Accessed on March 25, 2023. 3.1.2, 3.2a

[7] Inc. Xilinx. Flip flop. https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/ksg1504034293914.html, 2018. Accessed on March 25, 2023. 3.1.2, 3.2b

[8] Inc. Xilinx. Dsp48 block. https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/uwa1504034294196.html, 2018. Accessed on March 25, 2023. 3.1.2, 3.2c

[9] Inc. Xilinx. *Versal ACAP Memory Resources Architecture Manual (AM007)*, 2020. 3.1.2, 3.2d, 3.2e

[10] Alchitry. Fpga timing. https://alchitry.com/fpga-timing-verilog. Accessed on March 26, 2023. 3.1.3

[11] Inc. Xilinx. *UltraFast Design Methodology Guide for FPGAs and SoCs (UG949)*, 2022. 3.1.3, 3.1.4

[12] NandLand. What is setup and hold time in an fpga? https://nandland.com/lesson-12-setup-and-hold-time/. Accessed on June 11, 2023. 3.1.3

[13] Inc. Xilinx. *VCU128 Evaluation Board HW-U1-VCU128*, 2018. 3.5, 6.2.5, 6.9a, 6.9b

[14] Inc. Xilinx. Intellectual property. https://www.xilinx.com/products/intellectual-property.html. Accessed on March 26, 2023. 3.1.5

[15] Filippo Mantovani, Pablo Vizcaino, Fabio Banchelli, Marta Garcia-Gasulla, Roger Ferrer, Giorgos Ieronymakis, Nikos Dimou, Vassilis Papaefstathiou, and Jesus Labarta. Software development vehicles to enable extended and early co-design: a risc-v and hpc case of study. *arXiv preprint arXiv:2306.01797*, 2023. 3.2

[16] SchedMD. Slurm. https://slurm.schedmd.com/overview.html, 2023. Accessed on May 23, 2023. 3.2

[17] Inc. Xilinx. Boards. https://www.xilinx.com/products/boards-and-kits.html. Accessed on March 22, 2023. 3.3

[18] Inc. Xilinx. *LogiCORE IP Product Guide Integrated Logic Analyzer v6.1 (PG172)*, 2016. 3.3.1

[19] Inc. Xilinx. *Vivado Design Suite User Guide (UG910)*, 2021. 3.4.1, 3.4.1

[20] Inc. Xilinx. *Vivado Design Suite User Guide, Using Constraints (UG903)*, 2022. 3.4.2

[21] Inc. Xilinx. *Vivado Design Suite Properties Reference Guide (UG912)*, 2022. 3.4.2

[22] Ravi Budruk Tom Shanley, Don Anderson. *PCI Express System Architecture*. MindShare, Inc, 2003. 4.1, 4.1.1, 4.1.2

[23] Alessandro Rubini Jonathan Corbet, Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly, 2005. 4.1.1, 4.2, 4.2

[24] Intel. Msi-x. https://www.intel.com/content/www/us/en/docs/programmable/683268/21-1-4-0-0/msi-x.html. Accessed on April 06, 2023. 4.1.1

[25] VMware. Single root i/o virtualization (sr-iov). https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.networking.doc/GUID-CC021803-30EA-444D-BCBE-618E0D836B9F.html, 2019. Accessed on May 22, 2023. 4.1.2

[26] ARM Limited. Amba axi protocol specification. Protocol Specification ARM IHI 0022, ARM Limited, Cambridge, England, March 2023. Accessed: 14 April. 4.4, 4.3, 4.5

[27] ARM Limited. Amba4 axi4-stream protocol specification. Protocol Specification ARM IHI 0051A, ARM Limited, Cambridge, England, 2010. Accessed: 14 April. 4.3

[28] Inc. Xilinx. *DMA/Bridge Subsystem for PCI Express Product Guide (PG195)*, 2022. 4.4, 4.6, 4.2, 4.9, 4.10, 4.3

[29] Renato J. Recio, Paul R. Culley, Dave Garcia, Bernard Metzler, and Jeff Hilland. A Remote Direct Memory Access Protocol Specification. RFC 5040, October 2007. 4.5, 4.5.1

[30] Cisco. Infiniband concepts. https://www.cisco.com/c/en/us/td/docs/server_nw_virtual/2-10-0_release/element_manager/user_guide/appA.html, 2007. Accessed on April 11, 2023. 4.5.2

[31] Inc. Xilinx. *QDMA Subsystem for PCI Express Product Guide (PG302)*, 2022. 4.6, 4.15, 4.6.1, 4.6.1, 4.6.1, 4.6.1, 4.6.1, 4.16, 4.6.2, 4.6.4, 6.1

[32] Inc. Xilinx. Xilinx qdma linux driver. https://xilinx.github.io/dma_ip_drivers/master/QDMA/linux-kernel/html/index.html, 2023. Accessed on May 10, 2023. 4.6.4

[33] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016. 5.1.1

[34] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. May 2020. 5.1.1

[35] Antonio Pullini, Davide Rossi, Igor Loi, Giuseppe Tagliavini, and Luca Benini. Mr.wolf: An energy-precision scalable parallel ultra low power soc for iot edge processing. *IEEE Journal of Solid-State Circuits*, 54(7):1970–1981, 2019. 5.1.3

[36] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019. 5.1.4

[37] Inc. Xilinx. *UltraScale+ Devices Integrated Block for PCI Express Product Guide (PG213)*, 2022. 6.2.5

[38] David MacKenzie. TIME(1). Man page. Section 1. 6.3.2