Dynamic reconfiguration of hardware accelerators using Partial Reconfiguration

Master Thesis

Job Tijhuis

Delft University of Technology

TU Delft

Dynamic reconfiguration of hardware accelerators using Partial Reconfiguration

by

Job Tijhuis

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday, August 31, 2026 at 13:00.

Student number: 4275756

Thesis committee:	Prof. dr. H.P. Hofstee,	TU Delft, supervisor
	Dr. ir. C. Gao,	TU Delft
	Dr. ir. Z. Al-Ars,	External

Cover: An artist's rendering of time multiplexing of control signals to a quantum computer. Chalmers University of Technology | Boid (Modified)

An electronic version of this thesis is available at https://repository.tudelft.nl/.

TU Delft

Abstract

Field-programmable gate arrays (FPGAs) have become larger, and that increases the time needed to load their configuration from flash memory. This is a problem for FPGAs that communicate over PCI Express (PCIe) because the standard requires an endpoint device to respond within 120 ms of power-on.

This thesis investigates whether partial reconfiguration can offer a solution to this PCIe timing requirement. The Tandem configuration technique that is available for AMD FPGAs was tested, and its performance was evaluated using a measurement system that monitored both the PCIe link and the partial reconfiguration.

Tandem configuration, specifically Tandem PCIe, was determined to be the most effective solution to make sure the PCIe link on a large FPGA comes online within the 120 ms PCIe timing requirement, even when compared to other solutions not using partial reconfiguration. Partial reconfiguration on top of Tandem PCIe was tested and its performance was measured. The default pathway of using the Media Configuration Access Port (MCAP) to perform the partial reconfiguration was found to be too slow (under 3 MB/s) for most dynamic partial reconfiguration workloads.

A study was done to see how high partial reconfiguration rates can be pushed in hypothetical future FPGAs. It is concluded that, with careful design, reconfiguration rates of up to 400 GB/s are feasible. These rates were then applied to two case studies to see how much dynamic partial reconfiguration workloads can benefit from higher configuration rates. This analysis showed that for some workloads higher reconfiguration rates keep lowering the total runtime, but that an asymptote is reached eventually when the computation time becomes the bottleneck instead of the reconfiguration time.

Acknowledgements

I would like to take this opportunity to thank all the people who supported me along my long journey towards completing my academic career. My professors for their patience, academic guidance, and belief in my ability to get to the finish line. Philips for offering me an internship position for performing my research and being generous with their tremendous expertise. My partner and daughter for making me persist through the trials and tribulations. My friends for continuously enquiring and having the honest conversations every step of the way.

Job Tijhuis
Delft, August 2026

Contents

Glossary

Glossary

AMD Advanced Micro Devices 1, 3, 6, 15, 21–23, 28, 29, 32, 34, 35, 40

API application programming interface 21, 22, 36

ARM Advanced RISC Machines 22

ASIC application-specific integrated circuit 5

AXI Advanced eXtensible Interface 21, 23, 28, 49

BIOS Basic Input/Output System is the firmware that provides the services that an operating system (OS) and the programs that run on the OS with the ability to perform hardware initialization during the booting process after system power-on 13

BPI Byte Peripheral Interface 13, 14

BRAM Block RAM 6, 7

CDC clock domain crossing 34, 35

chipset The chipset of a computer system is usually one or more chips located on the motherboard, responsible for managing the data flow between the Central Processing Unit (CPU) and its peripherals. A chipset is usually designed to work with a specific family of CPUs and most of the time it is designed by the same company that designed the CPU 11–13

CLB configurable logic block 5–7, 10, 52

CLI command line interface 37

CPLD complex programmable logic device 5, 14, 15

CPU central processing unit 2, 11, 13, 22

CT computed tomography 1

DFX Dynamic Function eXchange 7, 15

DMA direct memory access 21, 22, 47, 59, 60

DSP digital signal processing 6, 7

EPROM erasable programmable read-only memory 5

FFT fast Fourier transform 2

FIFO first in, first out 24, 32

FPGA field-programmable gate array 1, 3–7, 58, 59

FTDI Future Technology Devices International Ltd 25, 29

GPIO general purpose input/output 21

GPU graphics processing unit 1, 11

GUI graphical user interface 37

HBM High Bandwidth Memory 19, 49, 50, 59

HDL hardware description language 21

I/O input/output 1, 11, 17, 18, 21, 23

IC integrated circuit 6, 7

ICAP Internal Configuration Access Port 8, 16, 17, 47–49, 58–60

ID identifier 12

IP intellectual property 8, 21–23, 25, 28, 29, 31, 32, 35

JTAG The JTAG standard (named after the Joint Test Action Group which developed it) is an industry standard for interacting with and verifying chip designs on a printed circuit board (PCB). A JTAG interface is a special interface added to a chip that allows for debugging and in the case of Field Programmable Gate Arrays (FPGAs) also reconfiguring of the chip. 17, 23, 25

KMDF Kernel-Mode Driver Framework 36, 37

LED light-emitting diode 22, 23

LLM large language model 54

LTSSM Link Training and Status State Machine 23, 24, 28, 31, 33, 38, 45
LUT lookup table 5, 6
MCAP Media Configuration Access Port 4, 17, 22, 25, 36, 58–60
MRI magnetic resonance imaging 1
MSBuild Microsoft Build Engine 36
MUX multiplexer 5, 6
NoC network on chip 49
NOR flash NOR flash is a type of flash where one end of each cell is connected to ground and the other end is connected to a bit line. Is is called NOR flash because it acts like a NOR gate: when the word line of a NOR cell is brought high, the corresponding storage transistor acts to pull the output bit line low 13
OOB out-of-band 19, 20
open-source Open source means the source code for a program is made freely available for possible modification and redistribution by its creators. 22
OS operating system 13, 21, 22, 25, 46, 60
PC personal computer 2, 11, 19, 22, 23, 29
PCB printed circuit board 41
PCI Peripheral Component Interconnect 11
PCI-X Peripheral Component Interconnect eXtended 11
PCIe Peripheral Component Interconnect Express 2, 3, 5, 11, 12, 19, 21, 22, 25, 42, 58
PDK process design kit 53
PR partial reconfiguration 7, 15
PROM programmable read-only memory 14–16
QDMA Queue direct memory access 22
QSPI Quad Serial Peripheral Interface 13, 14, 19, 41
RM reconfigurable module 9, 10
RP reconfigurable partition 9, 10
RR reconfigurable region 9
RTL register-transfer level 9, 21
RX receive 23, 31
SDK Software Development Kit 36
SEU single event upset 8
SMBus System Management Bus 20
SPI Serial Peripheral Interface 13
SRAM static random-access memory 1, 5, 10, 13
SSD solid-state drive 11
SSI stacked silicon interconnect 7
TX transmit 23, 31
UART Universal asynchronous receiver-transmitter 22, 23, 25
UEFI Unified Extensible Firmware Interface is a set of specifications written by the UEFI Forum. They define the architecture of the platform firmware used for booting and its interface for interaction with the operating system 13
USB Universal Serial Bus 25, 29
VCP virtual COM port 25
VSEC Vendor Specific Extended Capability 17, 36, 60
WDK Windows Driver Kit 36
XDMA Xilinx direct memory access 22
XRT Xilinx Runtime library 21, 22

1 Introduction

Field-programmable gate arrays (FPGAs) are used in many applications. At their inception, FPGAs were used mainly for glue logic or designs that require a lot of input/output (I/O) pins. Nowadays, high-end FPGAs are used mostly for fast data (pre-)processing, usually combined with high-speed transceivers to acquire and send data. Low-end FPGAs are typically used for (industrial) control systems and interface adaptation.

Most FPGAs use static random-access memory (SRAM) to store the configuration of the logic cells in the FPGA. This allows for reprogramming of the FPGA, making it a very flexible type of integrated circuit. FPGAs based on SRAM technology have the benefit that SRAM does not wear out and can be written to many times. However, SRAM is volatile and loses its contents when not powered. This means the FPGA not only needs to be configured when the configuration needs to be changed, but it also needs to be configured every time the system powers up because the configuration is lost upon power-down. This can lead to long loading times for large FPGA configurations if the configuration storage, or the reprogramming datapath, is slow. Combined with external timing requirements imposed by the interfaces that connect the FPGA to the rest of the system this creates a need to look at different approaches for configuring the FPGA at power-up.

In this thesis these challenges were analysed and a different approach for configuration loading was proposed that may scale better for loading larger FPGA configurations.

For the research in this thesis an FPGA from Advanced Micro Devices (AMD)—formerly Xilinx—was used to evaluate different solutions to these challenges. The AMD Alveo U50 accelerator card was used as the platform to do these evaluations on.

1.1. Context

This chapter starts with a general description of the department in which the project was executed. Then, an example of a data acquisition process that generates large amounts of data is described: magnetic resonance imaging (MRI). For this process, the data acquisition system is described in more detail. Then the focus turns to the preprocessing boards, which are the topic of this thesis.

1.1.1. Internship

This project was carried out as an internship within Philips Engineering Solutions, which supports other departments in the wider Philips organization with research & development, and engineering. It was carried out in the Digital Processing group which specializes in building systems that need to process data in real-time. The department uses FPGAs and graphics processing units (GPUs) for data processing.

Most projects in the Digital Processing group come from the Philips Healthcare division where medical imaging machines are designed. These medical devices—for example magnetic resonance imaging (MRI), computed tomography (CT) or ultrasound scanning machines—require image signal processing of their measurement data. To transform the measured signals to a high-quality medical image, specialized signal processing is needed.

1.1.2. Typical example of real-time data processing

There is a large quantity of data involved in creating high-quality medical images. To capture all this data in real-time, custom-made computing systems are designed to handle the image signal processing and various algorithmic image transformations.

Medical imaging requires a lot of data processing. Taking an MRI machine as an example, one can get a picture of what data processing is needed. The images generated from the MRI machine are a combination of a lot of different data points measured from the electromagnetic waves coming back from the patient. To get an image from these radio waves the data needs to be preprocessed, and techniques like fast Fourier transform (FFT) are used to construct an image. The amount of data that needs to be processed is so large that traditional computers cannot keep up with the data, or at least not efficiently.

1.1.3. Data acquisition system

An overview of an MRI system can be seen in Figure 1.1. The MRI system can be segmented into three parts: the MRI scanner, the data processing server and the control PC, which is used by an operator.

In this example the MRI scanner is taken as a singular block that sends data measured from the patient to the processing server. To eliminate electromagnetic interference from the magnet, the data cable from the MRI system is fibre optic, and the processing server is located outside the examination room. The focus of this thesis is not the physical principles of the MRI scanner, but the data preprocessing system.

The large amounts of data generated and the need for real-time processing require a specialized solution. This solution consists of FPGAs that are capable of parallel processing. The FPGAs preprocess the data, after which it is sent to the processing server where the final image processing is done on the central processing unit (CPU). The final image processing steps are mostly sequential computation, so there is no advantage to running it on an FPGA, and therefore the CPU is responsible for computing this part. Finally, the processed image is communicated using an Ethernet connection to the image display through the control PC. The control PC is used by an operator to control the MRI process, such as initiating a scan and viewing the resulting image.

1.1.4. Preprocessing board

The preprocessing boards inside the processing server are connected to the CPU through Peripheral Component Interconnect Express (PCIe). PCIe preprocessing boards provide modularity because depending on the processing needs, boards may be added (provided the server has additional slots). The use of standardized PCIe and Small Form-factor Pluggable (SFP) interfaces on the board allows for reuse in other projects if needed. For different projects, only the FPGA needs to be reprogrammed. The concept of reusing such preprocessing boards has been proven useful in practice.

Having this application developed within the healthcare domain implies that it should function ac-

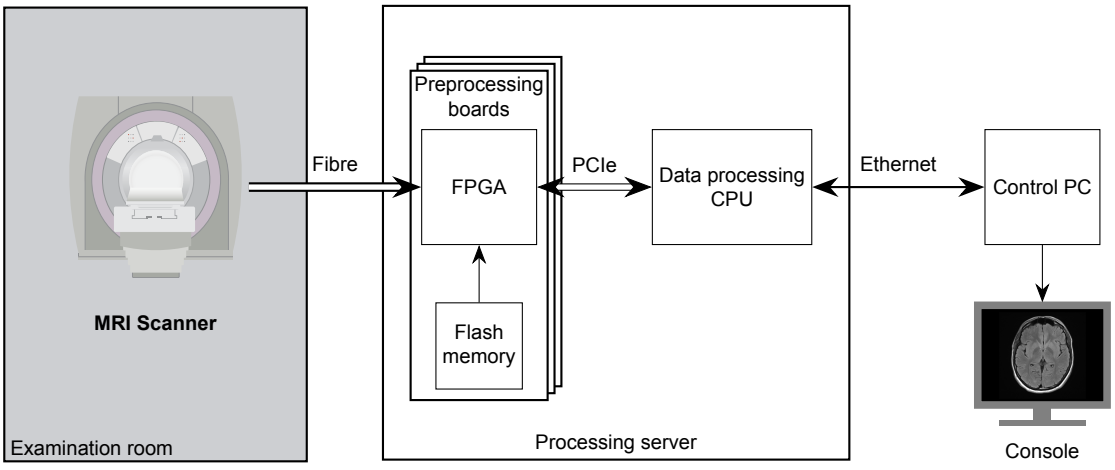


Figure 1.1: System overview of an MRI scanner that includes the scanner itself, the processing server and the imaging console; details regarding data acquisition have been left out.

cording to mission-critical domain needs, and imposes stringent requirements on the system to ensure that various constraints, such as reliability, accuracy, low-latency and quality of service, are met.

1.2. Challenges

One of the challenges of having longer configuration times is that this can cause problems in designs where there are timing constraints on the start-up of the system. In particular in designs where the FPGA communicates through PCIe because the PCIe standard dictates that the PCIe subsystem needs to be in a responsive state within 120 ms (see Section 2.3.2). To meet this timing requirement the configuration needs to be loaded into the FPGA very quickly from flash memory, but typical flash memory is only fast enough to meet this timing for smaller FPGAs, or else size restrictions must be put on the design to meet the timing constraints.

In the last decade FPGAs have become larger, containing more logic blocks. This creates the capability to contain larger configurations that have to be loaded into SRAM, leading to longer loading times. Systems with timing constraints at start-up may experience timing problems. Further, the PCIe standard requires that any subsystem is in a responsive state within 120 ms after start-up. Typical flash memory is not fast enough to meet this requirement for large FPGAs.

To address the problem of meeting the timing requirements at start-up several solutions have been proposed. Most of these solutions use the partial reconfiguration ability of FPGAs (see Section 2.2 for background). They all work on the same basic principle: An initial bitstream is loaded to partially configure the FPGA. This first stage contains the prerequisites for establishing the PCIe connection. In the second stage the remainder of the configuration is loaded. This two-step method allows loading the larger chunk of the FPGA configuration after the PCIe subsystem is already active, fulfilling the timing constraint requirement.

The different solutions provided by FPGA vendors use partial reconfiguration under the hood to solve the start-up timing requirements. However, it is also possible to use partial reconfiguration directly to configure the FPGA and create a custom solution that separates the PCIe core from the rest of the design and configures the part of the design containing the PCIe core first and then loads in the rest of the configuration. This gives more control over the end result, but requires more development time.

The solution offered by AMD is called Tandem configuration (Section 2.5). There are multiple variants of Tandem configuration, where the main differences are in the way the second stage bitstream is loaded. The second stage can be loaded from flash or through the PCIe bus because the critical path for the timing constraints only applies to the first stage bitstream. Once the first stage bitstream has initialized the PCIe bus there are no further timing constraints other than those set by the system designer.

The challenge with Tandem configuration is selecting the variant which is most beneficial because there are advantages and limitations to the different variants. With certain Tandem configuration variants it is possible to load part of the FPGA configuration through the PCIe bus. There are several methods when configuring the FPGA through PCIe, each with their own advantages and disadvantages. The main differences are: the configuration speed and the time needed to develop the solution.

1.3. Problem statement

One of the challenges encountered with the increasing capacity of FPGA devices in combination with the PCIe standard is that the PCIe standard mandates that end-point devices can start communicating within 120 ms after power on.
The main problem analysed in this thesis consists of two parts:

- Making sure the initial configuration is loaded within 120 ms to stay within the PCIe specification.
- Updating the FPGA configuration through the PCIe link from the PC without losing the PCIe connection.

1.4. Research question

The main research question is as follows. “What is the most efficient and reliable option to ensure that an FPGA connected through PCIe is detected at start-up and stays connected during dynamic partial reconfiguration while complying with the PCIe standard?”
To answer this main question, the following sub-questions have to be answered.

1.4.1. Sub-questions

1. What are the techniques available to reduce the start-up time of a large FPGA so it comes online within the 120 ms PCIe requirement?
2. How fast is dynamic partial reconfiguration on current FPGAs?
3. How can the reconfiguration time of FPGAs be reduced, and how can this benefit designs using dynamic partial reconfiguration?

1.5. Thesis outline

The remainder of this thesis is organized as follows. Chapter 2 gives background on FPGAs, partial reconfiguration and PCIe, and introduces Tandem configuration as a technique available on AMD FPGAs to meet PCIe start-up timing requirements. Chapter 3 describes the approach taken to evaluate the use of partial reconfiguration to solve the PCIe start-up problem. This includes: the target hardware platform, the available software design flows, and the design of a measurement system to characterize PCIe initialization and reconfiguration timing. Chapter 4 discusses the implementation of the measurement system, covering both the FPGA hardware design and the host software needed. Chapter 5 presents the resulting measurements of bitstream loading time, PCIe initialization time and reconfiguration speed over Media Configuration Access Port (MCAP). The measurements are used to determine whether the 120 ms timing requirement can be met. Chapter 6 then explores what higher reconfiguration rates could be offered beyond the current platform, illustrated with two case studies. Finally, Chapter 7 ends with the conclusions and recommendations.

2

Background

First some background is given on what a field-programmable gate array (FPGA) is in Section 2.1. Then partial reconfiguration of FPGAs is explained in Section 2.2. Then the fundamentals of how Peripheral Component Interconnect Express (PCIe) works are explained in Section 2.3. Some background is given on configuration flash in Section 2.4. Finally, an explanation of the Tandem configuration technique is given in Section 2.5.

2.1. Field-programmable gate array (FPGA)

The first commercial FPGA was made in 1985 by Xilinx to enable the creation of hardware devices that can be reprogrammed in the field [12]. It was an innovation combining static random-access memory (SRAM) and logic blocks in a configurable matrix. At first FPGAs were mainly used to test and verify application-specific integrated circuit (ASIC) designs.

In its simplest form, an FPGA is an array of configurable logic blocks (CLBs) with an interconnection network. Both the configurable logic blocks (CLBs) and the interconnection network are configurable. Figure 2.1 shows a schematic overview of a typical FPGA architecture.

Before FPGAs there were complex programmable logic devices (CPLDs), with the main difference between the two being that CPLDs use non-volatile memory—typically erasable programmable read-only memory (EPROM)—and FPGAs use SRAM which is volatile memory. The usage of SRAM in FPGAs has the advantage that it can be reprogrammed many times. CPLDs can only be reprogrammed a very limited number of times—typically around 1 to 1000 times—though it is common that a CPLD is only programmed once. The downside of FPGAs using volatile memory means its contents are lost when no power is applied. This also means that the configuration of the FPGA needs to be loaded in every time the FPGA powers up. For applications with a higher production volume CPLDs made more sense because of the lower per-unit cost. The extra cost of FPGAs is exacerbated by the added cost of external non-volatile memory that is needed for SRAM-based FPGAs to store the FPGA configuration. However, CPLDs are considered legacy technology now and are not widely available anymore.

2.1.1. Configurable logic blocks (CLBs)

An FPGA CLB typically contains the following three logic elements: a lookup table (LUT), a flip-flop and a multiplexer (MUX). Figure 2.2 shows the layout of a CLB. In this basic CLB the multiplexer can be configured to pass on the output of the LUT, or pass on the clocked output of the LUT through the flip-flop. When the output of the LUT is passed on without clocking, it can be combined with other CLBs to form longer and more complex logic paths. In more advanced FPGAs these elements can be arranged differently and there can be more than just one of these logic elements in a CLB; other logic elements can also be added.

2.1.2. Heterogeneous FPGAs

When FPGAs were first introduced every FPGA had its own unique internal structure. When newer models were introduced they were similarly structured to the previous model with improvements made for every new released device. The manufacturing of the FPGAs was also continually moved to more

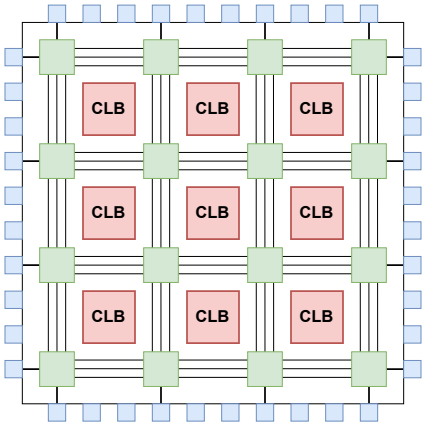


Figure 2.1: Schematic overview of the architecture of an FPGA

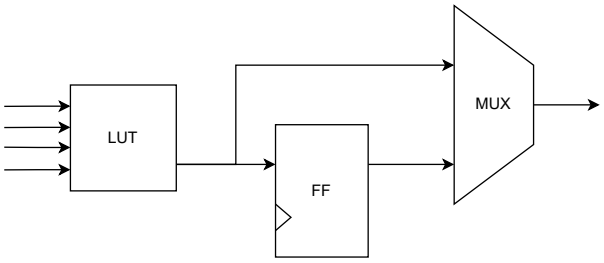


Figure 2.2: Schematic overview of a configurable logic block (CLB) containing a 4-input lookup table (LUT), a flip-flop (FF) and a multiplexer (MUX)

advanced and smaller integrated circuit (IC) technology nodes, offering higher transistor densities. With advances in IC technology moving very quickly, just putting more and more CLBs in FPGAs would have become unmanageable.

To solve this problem manufacturers switched over to heterogeneous architectures where blocks with differentiated functionality are inserted in the FPGA fabric alongside the CLBs. These blocks typically include but are not limited to: digital signal processing (DSP) blocks and Block RAMs (BRAMs). Digital signal processing (DSP) blocks contain dedicated logic for doing multiplication and addition allowing for efficient implementation of DSP algorithms on FPGAs. Block RAMs (BRAMs) are blocks of addressable memory that can be used to store large amounts of data. BRAMs are useful because one BRAM can store as much data as many CLBs; a single CLB only contains a few flip-flops.

2.1.3. FPGA architectures

The architecture of an FPGA is defined by the basic building blocks that are used to make up the FPGA fabric. This means that the CLBs, DSP blocks and BRAMs are the same between two different FPGAs of the same architecture. The amount of each of these blocks is what differentiates FPGA models. Using standardized building blocks for FPGAs with the same architecture allows for easier portability of designs between FPGAs. With each new FPGA generation these basic building blocks are improved. A new FPGA architecture is usually also designed for newer IC manufacturing technologies. Creating a new architecture then also allows for optimizations of the basic building blocks to make them map better to the specific IC technology. Newer architectures may also introduce new building blocks to support new use-cases.

2.1.4. AMD FPGAs

During the research performed in this thesis an FPGA made by Advanced Micro Devices (AMD) is used. This section gives some more information specifically about current Advanced Micro Devices (AMD) FPGA architectures. Three AMD FPGA architectures that are widely used are: the 7 series, Ultrascale and Ultrascale+ architectures. Some older FPGAs are also still available and under production, but they are not actively being supported and thus not recommended for new designs. This means the

software tooling to work with these older FPGAs does not receive updates. First the 7 series architecture is explained—this architecture predates the Ultrascale and Ultrascale+ architectures and has been around since 2010. Second the Ultrascale and Ultrascale+ architectures are explained.

7 series architecture
The AMD/Xilinx 7 series FPGAs were the first to use stacked silicon interconnect (SSI) to combine multiple dies into a single FPGA for their high-end FPGAs. This technique uses a silicon interposer to connect multiple silicon dies into a single IC. This way larger FPGAs can be created than using a monolithic die because the yield is higher for the smaller dies.

At launch the 7 series offered FPGAs in the high-end, mid-range and the upper low-end, with their Virtex-7, Kintex-7 and Artix-7 devices respectively. At first the lowest-end segments were still served by the older Spartan-6 FPGAs. With the release of the Spartan-7 family in 2017 this changed and the 7 series now offers solutions for many target applications from the very low-end to the very high-end.

Ultrascale & Ultrascale+ architecture
The Ultrascale and Ultrascale+ FPGAs were the next leap in AMD/Xilinx architecture after the 7 series. This leap offers even larger FPGAs at the high-end and more efficient FPGAs overall because of advances in technology nodes. This is combined with a new design of the programmable logic fabric that has greater flexibility for placement of logic in the CLBs. The Ultrascale+ architecture is a minor revision of the Ultrascale architecture. Ultrascale+ offers newer and faster interfaces over the Ultrascale architecture. The underlying architecture of the logic blocks is the same, but the logic is more energy efficient because of technology advances. The Ultrascale architecture was introduced in 2013 and is manufactured on a TSMC 20 nm process. The Ultrascale series of devices was only released with devices in the high-end and mid-range, Virtex Ultrascale and Kintex Ultrascale respectively. The Ultrascale+ architecture was introduced in 2016 and is manufactured on a TSMC 16 nm process. The Ultrascale+ series launched with high-end and mid-range devices just like the Ultrascale architecture. In 2021 lower-end FPGAs with high-speed transceivers were added to the series with the launch of Artix Ultrascale+ series of devices. In 2024 Spartan Ultrascale+ was added bringing the product line of Ultrascale+ in line with the 7 series.

2.1.5. Bitstreams
When an FPGA comes fresh out of the factory it does not have a configuration. For an FPGA to work the CLBs and routing blocks inside the FPGA need to have a configuration loaded into them. A design can be flashed onto the FPGA through a variety of ways, usually JTAG is used during development. The AMD FPGAs are SRAM based and SRAM does however not hold its value when power is lost. To not have to manually load a configuration into the FPGA every time it is powered on an FPGA is usually paired with a flash memory that a configuration is loaded from at power on. The FPGA will then read its configuration from this flash memory every time it is powered up.

2.2. Partial reconfiguration
Partial reconfiguration (PR) is a technique that allows for only part of the FPGA to be reconfigured instead of configuring the entire FPGA; this can be done while the rest of the FPGA remains operational. As a concept this technique is a logical evolution because FPGAs are composed of two distinct parts: the configuration memory and the logic. The logic is the actual active hardware blocks while the configuration memory states how this logic should act. The logic includes CLBs, DSPs, or BRAMs blocks, but also the routing resources that form the connections between the different blocks. The partial reconfiguration technique takes advantage of the fact that parts of the configuration memory, and thus the configuration of the logic, can be changed without overriding the entire configuration.

For simplicity only AMD's implementation of partial reconfiguration is considered in this explanation of partial reconfiguration, but the explanation of partial reconfiguration in this section should also be applicable to FPGAs from other FPGA vendors. AMD calls their partial reconfiguration technique Dynamic Function eXchange (DFX) [27]. The term partial reconfiguration (PR) will be used when referring to the general technique, the term Dynamic Function eXchange (DFX) will be used when referring to the AMD specific implementation of partial reconfiguration.

2.2.1. Static vs. dynamic partial reconfiguration

Partial reconfiguration can be done either statically or dynamically. In the static situation the FPGA is inactive; this means all logic in the FPGA is turned off while the partial reconfiguration is being performed. In the dynamic situation the FPGA remains active while the partial reconfiguration is being performed.

There are two ways in which static partial reconfiguration can be done. The first way is to do it before the device is fully active by keeping the FPGA inactive after initial configuration. This can be useful to load different configurations at start-up depending on the desired behaviour of the device. The second way is to make the device inactive by putting it in shutdown mode. By putting the FPGA in shutdown mode all logic inside the device becomes inactive. This way a static reconfiguration can be performed. In either case the device can be made active after the partial reconfiguration is done.

Dynamic partial reconfiguration is done while the FPGA stays active. To make sure there are no glitches in the device during reconfiguration there needs to be no contention between the already active logic and the logic that will be configured with the partial reconfiguration. Contention can be avoided by making sure any resources the partial bitstream configures do not conflict with any active logic resources already in use. This is done by instructing the place & route software to place the logic in a separate region. The software ensures that the new configuration in the partial bitstream will not overlap with any configuration data that needs to stay running.

Dynamic reconfiguration is the recommended approach to use nowadays. The problem of making sure there is no contention can be handled by the software given the right design practices are followed. Static partial reconfiguration can be used if the routing resources need to be changed with partial reconfiguration since it mitigates the risk of internal contention. It is however preferable to avoid having to change the routing resources in the device.

2.2.2. Self-reconfiguration

Starting with Virtex II FPGAs AMD/Xilinx FPGAs gained the ability to reconfigure the FPGA from within the FPGA fabric itself. Self-reconfiguration is done by including the Internal Configuration Access Port (ICAP) in the design. The ICAP component can be used to read and write the configuration logic of the FPGA from within the FPGA itself. There are endless possibilities on how to use self-reconfiguration. Some examples include: dynamically adapting bus widths, self-repairing single event upset (SEU) errors in the configuration memory [21], or time multiplexing algorithms.

2.2.3. Bitstream encryption

FPGAs are also used in products that require high levels of security. To better secure the design that is loaded into the FPGA bitstream encryption was introduced with Virtex-II FPGAs. Bitstream encryption of partial bitstreams was however not supported until the Virtex-6 FPGAs. This limited the applications where partial reconfiguration could be used. It was also possible to use encrypted partial bitstreams on the Virtex-5 FPGAs by using an intellectual property (IP) block. This IP block does the decryption of the partial bitstream inside the programmable logic of the FPGA and uses the ICAP to load it into the FPGA configuration memory. Section 2.2.3 shows which AMD/Xilinx FPGAs support this.

Table 2.1: Different levels of PR support on AMD/Xilinx FPGAs

	Released	Static PR	Dynamic PR	Self-reconfiguration	Encrypted bitstream
XC6200	1995	✔	✔	✘	✘
Virtex	1998	✔	✘	✘	✘
Virtex II	2001	✔	✔	✔	✘
Virtex II Pro	2002	✔	✔	✔	✘
Virtex-4	2004	✔	✔	✔	✘
Virtex-5	2006	✔	✔	✔	✔[1]
Virtex-6	2009	✔	✔	✔	✔
Virtex-7	2010	✔	✔	✔	✔
Virtex Ultrascale	2014	✔	✔	✔	✔
Virtex Ultrascale+	2016	✔	✔	✔	✔

[1] Using an added IP block.

2.2.4. Reconfigurable regions

When designing a system using partial reconfiguration the designer has to designate regions of the FPGA that can be partially reconfigured; these regions are called reconfigurable regions (RRs). Then modules/entities in the register-transfer level (RTL) source code can be designated as partitions that can be mapped onto these RRs; these partitions are called reconfigurable partitions (RPs) [27]. Figure 2.3 shows how a design hierarchy can be split into RPs. When an entity/module is marked as an RP all logic that is part of that module will be mapped onto the same RR. Figure 2.4 shows how RPs can be assigned to different RRs; an RP can only ever be assigned to one RR.

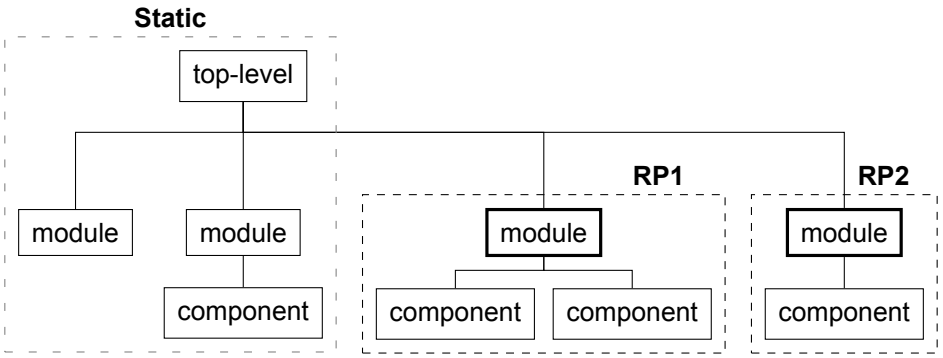


Figure 2.3: Design hierarchy containing reconfigurable partitions

These RPs can have multiple different implementations defined for the same partition in RTL source code and the different RTL instances of a reconfigurable partition will define the different implementations of the RP; these implementations are called reconfigurable modules (RMs). When generating the implementation of these RMs the designs will be constrained to the resources within the RR. Then when going through the usual implementation steps to generate a bitstream, in addition to a full bitstream, a partial bitstream is also generated for every RM. These partial bitstreams can then be loaded into their assigned RP during runtime.

Figure 2.5 shows a simplified view of RPs laid out on an FPGA where the dashed lines indicate the RRs and the bitstreams on the right are the different RMs. There are multiple bitstreams with each bitstream being a different variant, or RM, that can be loaded into a particular reconfigurable partition during runtime. The rest of the FPGA contains the static logic that is not being reconfigured at runtime, called the static region—in contrast to the reconfigurable regions.

2.2.5. Benefits

Partial reconfiguration can offer benefits to many different designs [19]. One example is using time multiplexing, which can be used in the case the design might not require all functionality at the same

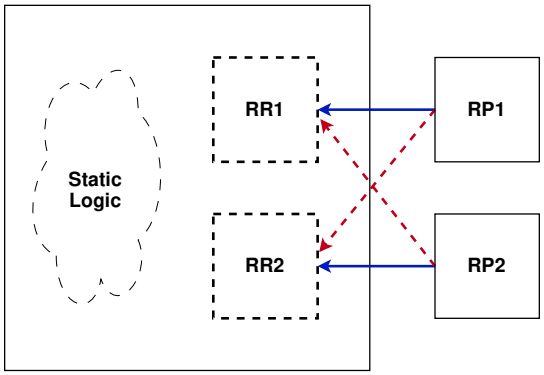


Figure 2.4: Overview of a partial reconfiguration design layout. Either the blue arrows, or the red arrows are valid assignments of RPs to the RRs.

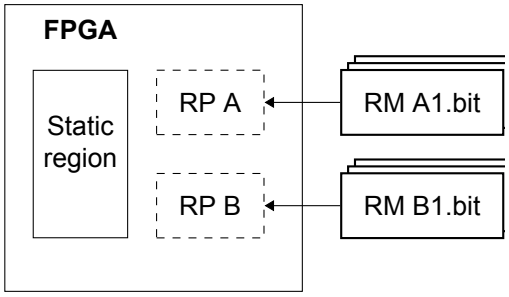


Figure 2.5: Reconfigurable regions and partitions inside an FPGA

time [31, 34]. This is done by having a RP be reloaded with different RM configurations that are never needed at the same time and can then be reconfigured when needed, for example when a device is switched into a different mode. By having a design that can take advantage of using the same region for multiple functionalities it uses less area on the FPGA and this would make it possible to make the design fit on a smaller FPGA thus saving costs.

Partial reconfiguration can of course also be used to update the FPGA after initial configuration while keeping the FPGA running. PR can also be used cleverly to incrementally load in the FPGA configuration in multiple stages, which is what is done in the Tandem configuration technique introduced later in this chapter; in Section 2.5.

2.2.6. Bitstream compression

PR implicitly means that not every part of the FPGA needs to be configured at the same time. This property is already used by all modern FPGA synthesis and bitstream generation tools in the form of bitstream compression; here they use the fact that the parts of the FPGA that do not contain any active logic do not need to be written to, such that the tools generate a bitstream that only configures the sections of the memory that are actually used [24]. This reduces the bitstream size and in consequence reduces the load time of the bitstream.

2.2.7. Challenges

One of the challenges PR has to deal with is the hardware overhead needed to enable this approach. Early FPGAs only supported reprogramming the entire configuration of the FPGA at once because supporting partial addressing of the SRAM would require extra logic, and thus more transistors which would reduce the amount of CLBs that could be put on the FPGA because of the limited numbers of transistors per chip. But with the ever-increasing transistor density of Moore's law [14] this became less of a problem, and these days FPGAs support greater and greater configuration granularity of the different logic elements on the chip. The main limit on accessing these configuration memories currently is the amount of routing resources that would be needed to make them all addressable.

Another challenge is that there is a disadvantage to the fact that the SRAM needs to be reloaded

with the configuration every time the FPGA has been powered down, but this property can also be an advantage. This is because if there is already a need to duplicate the FPGA configuration there will always be an active copy of the configuration loaded in the SRAM. This property can be used to update a device without having to write to memory that is actively being used. For example, the current configuration in the SRAM can be left intact while the flash memory can be updated to a new version that can be loaded in the next time the FPGA is reloaded, after a reboot for example. This way the change of configuration is handled in the reboot sequence that needs to account for the loading of the configuration from the flash memory to the FPGA already.

2.3. PCIe

PCIe is a high-speed serial bus expansion and communication standard used in PCs and servers to add additional input/output (I/O), storage or compute capabilities to a system. It is designed to be a successor to the older Peripheral Component Interconnect (PCI) and Peripheral Component Interconnect eXtended (PCI-X) standards.

2.3.1. Architecture

The difference between PCIe and older standards is that instead of a shared parallel bus, where all the data lines are shared between all devices on the bus, like PCI and PCI-X, PCIe uses a point-to-point topology. A typical topology is shown in Figure 2.6, where every device is connected with a dedicated serial link to either the root complex or a PCIe switch. Each link consists of multiple PCIe lanes where the number of lanes depends on the physical interface size. The root complex is the central point—root—of the hierarchy and acts as the bridge to the host processor and is responsible for managing the PCIe bus and controlling all devices connected to the PCIe bus.

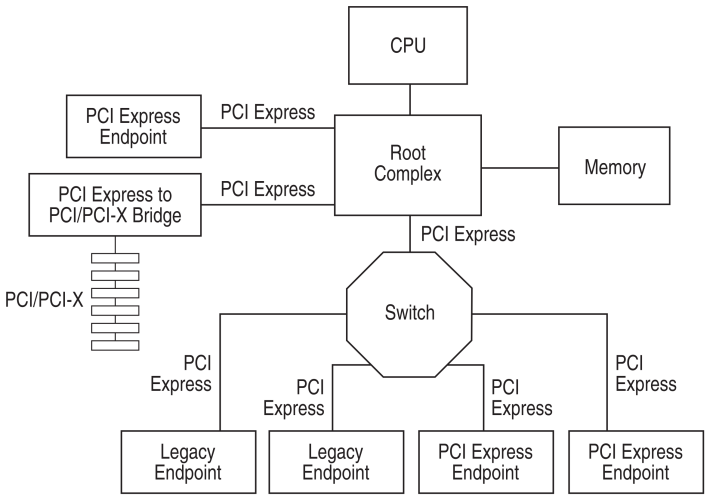


Figure 2.6: Typical PCIe topology, as illustrated in [17].

Topology

While devices can be connected directly to the root complex, in a typical PCIe topology only some endpoints are connected directly to the root complex while the rest are connected to a PCIe switch connected to the root complex. The PCIe switch is usually located in a chipset external to the central processing unit (CPU). This PCIe switch is connected to the root complex via a PCIe link on the one side and to multiple PCIe endpoints on the other side. In this switch the bandwidth is shared on a virtual bus, but it does allow for more physical devices to be connected to the PCIe bus. The devices connected directly to the root complex usually have the highest dedicated bandwidth available to them and this is why high-speed devices like graphics processing units (GPUs) or high-speed storage like solid-state drives (SSDs), or—in the case of this thesis—an FPGA processing card for data processing acceleration, are connected this way.

PCIe bandwidth sharing
Because the bandwidth is shared within a PCIe switch this means that the devices connected via the PCIe switch will not be able to use the full bandwidth all the time, but this does not need to be a problem if the devices are not high-speed peripherals that are active all the time. The type of devices connected to the PCIe switch in the chipset are commonly lower speed devices or peripherals that are active more infrequently. There do however exist dedicated PCIe switches that have much higher bandwidths and support multiple high-speed devices.

With the newer versions of the PCIe standard it is now architecturally easier to support many more high-speed peripherals. This is because the three most recent updates of the PCIe standard doubled the total throughput of each PCIe lane, where each update doubled the speed compared to the previous standard. This exponential growth means that PCIe 5.0 now has 4 times as much bandwidth as PCIe 3.0, with the newly released PCIe 6.0 standard coming in at 8 times more bandwidth than 3.0. Therefore, it is possible to support multiple previous generation PCIe devices on a single PCIe link while still having full bandwidth available for every connected endpoint.

Enumeration
One thing the PCIe root complex is responsible for is enumeration of attached devices. This is the process of identifying all attached PCIe endpoints and assigning unique Bus, Device and Function numbers to the devices so that they can be addressed and accessed by the host system. The combination of Bus, Device and Function numbers make up an identifier (ID) that uniquely identifies a specific device within a system. Enumeration usually happens at system start-up, so that is the time when new devices attached to the system will be detected, and made available to the host system.

2.3.2. Challenges of using PCIe with reconfigurable FPGAs

Because reconfiguration of the FPGA also includes configuring the PCIe blocks of the FPGA, this means that the PCIe connection is offline during reconfiguration. However, the PCIe standard [17] states that the connection needs to be online always, or at least that the connected PCIe device responds to the requests from the root complex, but during reconfiguration of the FPGA the PCIe link is completely down and unresponsive. This might lead to the root complex, which is the central hub for all the traffic on the PCIe bus, dropping the connection because the FPGA is unresponsive. Even when the FPGA is reconfigured during runtime there is still the problem that it can take a while for the FPGA to be configured when a system powers on because the initial configuration of the FPGA, which includes the PCIe block configuration, needs to be loaded. This is a problem because the PCIe standard [17] states that all devices need to be up and be responsive within 120 ms of start-up, otherwise the device might not get enumerated and detected by the system.

Enumeration only at start-up
The PCIe standard does allow for a link to be re-established when a communication error occurs. If the FPGA being reconfigured is seen as a communication error this process can be used to reconnect the FPGA to the system if the exact same PCIe link is established by the FPGA again. This will however not work if the PCIe subsystem inside the FPGA is configured differently, for example with a different link speed or width, or if the ID of the device changes, because then the PCIe root complex will see it as a new PCIe device.

This trick will however not work if there was never an initial link established and this comes up in the next challenge of using PCIe with FPGAs that can be reconfigured during runtime. For a PCIe link to be established successfully the root complex needs to enumerate the device and enumeration usually only happens at system start-up.

At start-up power is applied to all devices present in the system and a reset signal is asserted for a set period for all devices, for PCIe this is 100 ms, and after the reset signal is deasserted all devices need to be ready within 20 ms to respond to the enumeration requests of the root complex. This means that the FPGA needs to be fully configured within 100 + 20 = 120 ms after initial power is applied. FPGAs will start loading the configuration from the flash memory after power is applied, but depending on the size of the configuration this may actually take longer than 120 ms. This causes the problem that the FPGA device will never be recognized by the system when this timing is not met.

Hot-Plug

With PCIe, it is possible to trigger enumeration again after system start-up using PCIe Hot-Plug [17, p. 514] but Hot-Plug support is optional and not every system supports it. Hot-Plug also needs to be supported by both the hardware and software layers. This means that even if a system physically supports Hot-Plug, two further things are also needed: a system BIOS, or UEFI firmware on modern systems, that supports Hot-Plug and an operating system (OS) that supports Hot-Plug. On top of this comes that PCIe Hot-Plug is not necessarily intended for the use case of re-adding a device that has not actually been physically removed from the PCIe slot, which can cause problems depending on the specific implementation of PCIe Hot-Plug in both the system's Hot-Plug controller and the software implementations on top of it.

Trigger re-enumeration from software

It is sometimes possible to initiate PCIe enumeration again after system start-up from the OS or drivers, but this can be very complicated and is hard to do portably for different system configurations. It is more feasible when using newer high-end CPUs where the root complex may be part of the CPU itself and all PCIe lanes are directly connected to the CPU itself.

However, most systems in use today only have part of their PCIe lanes connected directly to the CPU and the rest are connected to the CPU via the chipset. This means that depending on which slot a device is inserted in the device is either connected straight to the CPU, or connected via the chipset. PCIe devices connected to the CPU through the chipset or other PCIe switch chips on the motherboard—the chipset acts as a PCIe switch—are thus not connected straight to the CPU. The introduction of these intermediary switches makes it more complex to re-enumerate because that depends on the PCIe switch it is connected through to support re-enumeration. It also creates an inconsistent experience depending on which slot a card is plugged into because re-enumeration might work differently if a PCIe card is connected straight to the CPU vs through a switch. However, PCIe slots on a motherboard are usually not physically different or marked clearly whether they are connected through a switch or not.

Therefore, if it is possible to solve this problem in the FPGA, there is no dependency on the specific system or CPU. Partial reconfiguration can possibly offer such a solution.

2.4. Configuration flash

Because the configuration memory of an FPGA is made up of SRAM and this is volatile memory it loses its contents every time the FPGA is powered down. This means that the configuration needs to be loaded from some type of non-volatile storage into the FPGA at power-up. The configuration is usually loaded in from an external flash storage chip located next to the FPGA on the same board, where the flash storage is able to keep information stored also without power.

2.4.1. Different types of flash storage

There are two types of flash storage chips commonly used for storage of FPGA initial configurations: Flash chips with a Byte Peripheral Interface (BPI) or a Quad Serial Peripheral Interface (QSPI). BPI flash is a parallel NOR flash and QSPI is a serial NOR flash with an option to use QSPI mode to transfer more data at once. These different interfaces achieve the same purpose and most FPGAs support one or both of these interfaces to load their configuration from. However, there are some differences between BPI and QSPI that are relevant for the challenges faced when reconfiguring FPGAs that have an active PCIe connection, like discussed in Section 2.3.2.

Differences BPI and QSPI

The main difference between BPI and QSPI is that BPI is parallel NOR flash and QSPI is serial NOR flash. BPI has a 16-bit wide data bus to read out the configuration data, this data bus can run at a speed up to $133\,\text{MHz}$ meaning it can achieve a configuration throughput of $16 \cdot 133\,\text{MHz} = 2128\,\text{Mbit/s} = 266\,\text{MB/s}$; here a Micron BPI memory [1] was taken as an example.

QSPI has an Serial Peripheral Interface (SPI) interface that is used to read out the configuration data, where SPI has been a widely used *de facto* standard for serial communication in embedded systems. In its simplest mode SPI only has three wires, one clock wire, one wire for communicating from the master to the slave, and one wire from the slave to the master. In the communication between the FPGA and the flash memory, the FPGA is the master and the flash memory is the slave. This means that there

is only a 1-bit wide data bus, since only reading data from the memory is of interest here. QSPI has the option to switch to a Quad SPI mode where 4 bits can be transferred at the same time. However, this is still not a very wide data bus for reading in the configuration quickly, this can cause problems for large FPGAs. This is why most FPGAs also implement a dual QSPI mode, in this mode two SPI flash chips are used in parallel to achieve an 8-bit wide data bus, as shown in [10]. With this 8-bit wide data bus at a speed up to 104 MHz a configuration throughput of $8 \cdot 104$ MHz $= 832$ Mbit/s $= 104$ MB/s can be achieved; here an Infineon SPI memory [20] was taken as an example.

Parallel flash availability
It can be seen that the throughput with QSPI is significantly lower than the throughput that can be achieved with BPI, this means that the loading in of the configuration with QSPI flash will take longer than when BPI flash is used. In the past it was an option to select BPI flash when faster configuration times were required, but this is not possible anymore because while BPI is still available it is considered obsolete and therefore BPI flash is not recommended for new designs [22].

For example, parallel NOR Flash from Micron is becoming obsolete and there are no new parallel NOR flashes replacing them. Micron offers parallel NOR flash that is part of their Product Longevity Program (PLP) but all their product offerings are already past their 10+ years lifecycle and are thus not suited for new products because there are no guarantees about supply.

2.4.2. Using a CPLD as an External configuration manager

It is possible to achieve higher configuration speeds even when using QSPI flash as the non-volatile storage. This can be done using an external configuration manager connected to a wider data bus on the FPGA [18]. On AMD FPGAs this can be done using the SelectMAP configuration mode, this interface has a 32-bit wide data bus to which an external CPLD can be connected to configure the FPGA [24]. The configuration data is still stored in flash memory chips, most likely QSPI chips, but the flash chips are connected to the CPLD. To achieve the higher configuration speed the CPLD multiplexes multiple flash chips—reads from them at the same time—and this can be scaled up to as many flash chips as needed to achieve the desired speed.

CPLDs discontinued
For a while CPLDs were considered an acceptable solution to the configuration speed problem, but in January 2024 AMD announced the discontinuation of its remaining CPLD families [33]. With AMD stepping out of the market as one of the major suppliers of CPLDs, CPLDs are no longer a feasible solution when designing and producing new systems.

Replacements
It is also possible to use a microcontroller as the configuration manager, but CPLDs are considered more reliable because a microcontroller has a boot process to go through that needs to be configured properly as well, while the configuration of a CPLD is fixed in place and functions like an FPGA with a non-volatile configuration memory. Another solution is using an FPGA to replace the CPLD, however this has the problem that this FPGA also needs to be configured first before it can start supplying configuration data to the main FPGA, eating into the timing budget. The same is also true for the microcontroller booting taking up time making the loading of the configuration into the FPGA longer.

Another solution is to use the partial reconfigurability of modern FPGAs to speed up the initial configuration of the PCIe block in the FPGA. This is done by decreasing the size of the bitstream that needs to be loaded from the programmable read-only memory (PROM) to get the PCIe block online faster. This technique is explained in the next section.

2.5. Tandem configuration

Tandem configuration is a solution offered by AMD/Xilinx that can solve the problem of FPGA initialization times when using PCIe [9]. This is achieved by reducing the loading time of the initial configuration by creating a minimal configuration that contains just the PCIe block. The initial configuration also includes a connection to the configuration engine to be able to load in the rest of the design after the PCIe block has been configured. This solution is available on 7 series, Ultrascale and Ultrascale+ AMD FPGAs which have integrated PCIe blocks.

Tandem configuration works by separating the bitstream into two parts that are loaded in two stages. In the first stage just the PCIe block is configured, thus allowing for the PCIe link to be initialized before the rest of the design is loaded in the second stage. Tandem configuration makes use of dynamic PR—explained in more detail in Section 2.2.1—to allow for the partial reconfiguration to take place while the PCIe block is able to stay active. Because the first stage bitstream only contains the PCIe block and a connection to the configuration engine, it is a lot smaller than a full bitstream. This makes it so that the configuration time is significantly reduced and the FPGA is able to meet the PCIe timing requirements.

There are multiple types of Tandem configuration where each type has its own benefits and drawbacks. The types of Tandem configuration currently offered by AMD, on their latest devices, are the following five: (1) Tandem PROM, (2) Tandem PCIe, (3) Tandem PCIe with Field Updates, (4) Tandem PCIe + Dynamic Function eXchange (DFX) and (5) DFX over PCIe. These types follow from the documentation of Tandem configuration as given in the integrated PCIe block product guide for Ultrascale+ devices [28]. The types of Tandem configuration can be subdivided into three sub-categories: Tandem configuration where the device can be only reconfigured once every power up, Tandem configuration where the FPGA can be updated using field updates, and Tandem configuration where DFX can be used to partially reconfigure the FPGA after the initial configuration has been loaded in on power up.

The first subcategory consists of (1) Tandem PROM and (2) Tandem PCIe. With these Tandem configuration types the FPGA can only be configured once, which presents the simplest method of Tandem configuration where the FPGA is only configured once and Tandem configuration is used solely to meet PCIe timing requirements.

The second subcategory consists of just (3) Tandem PCIe with Field Updates, this is because it is the only type that allows for field updates without having to set up a custom workflow with DFX. As an underlying technology it does use DFX to implement the field updates and the region of the second stage bitstream does need to be laid out manually on the FPGA. However, the benefit is that this can all be done within a template that is provided by AMD making it easier to implement than the general DFX solution.

The third subcategory consists of (4) Tandem PCIe + DFX and (5) DFX over PCIe. With these methods the general Dynamic Function eXchange (DFX), or partial reconfiguration technique is used directly, whereas the other Tandem configuration methods used it in the underlying implementation without the user having to explicitly use DFX. Using DFX directly means the designer is responsible for correctly configuring the reconfigurable regions (Section 2.2.4) that will make up the first and second stages.

The coming sections explain the details, requirements and potential advantages/drawbacks of the different Tandem configuration subcategories and types.

2.5.1. Tandem PROM/Tandem PCIe

Tandem PROM and Tandem PCIe offer the same functionality when it comes to their capabilities to update the FPGA at a later date. The main difference between Tandem PROM and Tandem PCIe is that with Tandem PROM the first and second stage bitstreams are both loaded from the flash storage, or programmable read-only memory (PROM), whereas with Tandem PCIe only the first stage is loaded from the PROM and the second stage is loaded in through PCIe.

The research that Tandem configuration builds upon [8] focused on adding field update functionality to FPGAs from external sources and did not focus on the concept of separating a design into a first and second stage bitstream. They only used the concept of separating the bitstream into two stages to be able to load in an externally provided bitstream by using the self-configuration abilities provided by partial reconfiguration to field update the FPGA without intervention of an external processor or CPLD.

However, Tandem PROM takes just the concept of separating the bitstream loading into two stages and uses that to provide a solution to reduce initial FPGA configuration times while still using the traditional PROM to load the configuration data from. The same is true for Tandem PCIe in its simplest form because it also does not allow for field updates.

Tandem PROM is used as an example to explain how the two stages are loaded in Tandem configuration. Following that the working of Tandem PCIe is explained, with the main difference being that the second stage is loaded from a different location. For Tandem PCIe the bitstream partitioning is the same as in Tandem PROM, but the configuration is loaded into the FPGA through a different method.

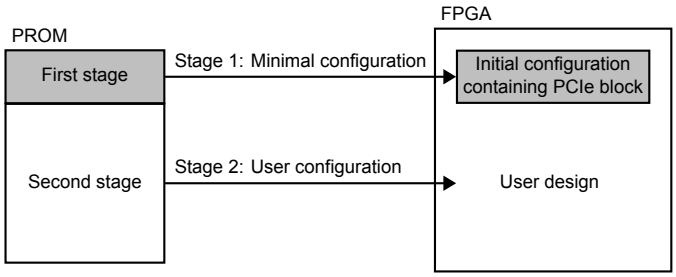


Figure 2.7: Tandem PROM bitstream loading

(1) Tandem PROM
Figure 2.7 shows how the bitstream is split up into two stages which are located in a PROM. The first stage bitstream only contains the configuration for the PCIe block and the configuration engine needed to load in the second stage. Because the first stage is an as minimal as possible configuration, the bitstream is small compared to a full bitstream and can be loaded in a lot faster. After the first bitstream is loaded in the PCIe subsystem is configured and ready for communication with the host system. Then the second stage bitstream can be loaded in while PCIe enumeration can already take place, making sure the FPGA is recognized and meets the PCIe timing requirements. Then after the second stage is fully loaded Tandem configuration is done and the FPGA works just like it would if the configuration was loaded from flash memory as a whole design.

The only difference is that when creating a design with Tandem configuration it needs to be taken into account that the regions used by the PCIe block and the configuration engine in the first stage bitstream are not available for the user design. These regions are however very small and do not use a lot of resources, and the PCIe connection is of course available in the user design.

(2) Tandem PCIe
Early research into what looks like a version of Tandem PCIe was done in the interest of being able to perform easier field updates to FPGAs, as shown in [8]. This was done by putting a configuration controller in the FPGA logic that accepts a partial bitstream from an external source utilizing the self-reconfiguration capabilities of the Virtex-II FPGA in the form of the ICAP. They demonstrated that partial reconfiguration can be used to update the FPGA while it is still running, which is what Tandem PCIe also does, albeit in a limited capacity because Tandem PCIe focuses just on reducing configuration times and does not add the ability to do field updates in its simplest form.

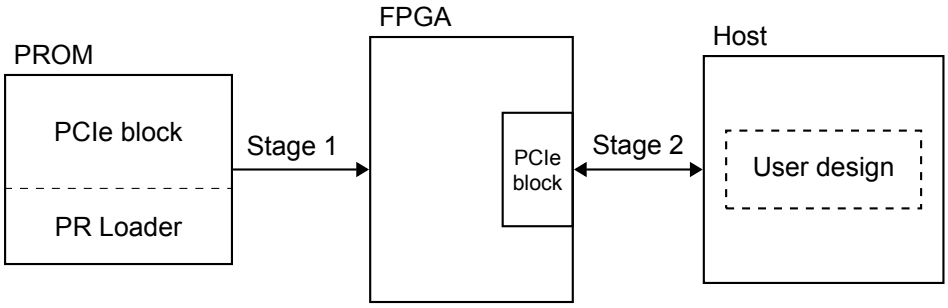


Figure 2.8: Tandem PCIe bitstream loading

Figure 2.8 shows how the two different bitstream stages are loaded in when using the Tandem PCIe configuration. The first stage bitstream is loaded in from the PROM, just like in the Tandem PROM configuration. Then after the PCIe block configured in the first stage is online and has established a PCIe connection the second stage bitstream is loaded in through PCIe from the host system. After the second stage is loaded the PCIe connection is switched back to the user design after it is done being used to load in the second stage bitstream. Tandem PCIe has the same design differences to take into account when creating a design with it, but the area taken up by the PCIe block will theoretically be slightly larger because the configuration engine now also needs to talk through PCIe and needs an extra switcher to hand the PCIe connection back to the user design after configuration is done.

Tandem PROM vs. PCIe

Tandem PROM can be preferable over Tandem PCIe because it simplifies the system design by removing the requirement for the host system to supply the second stage bitstream—this also keeps the complete configuration bitstream more localized to the FPGA device itself. Because with Tandem PROM there is no communication with the host system during initial configuration of the FPGA it has the advantage of reduced system complexity which reduces chances of errors during initial configuration of the FPGA. Using Tandem PROM instead of Tandem PCIe does come with the disadvantage that with Tandem PROM it is a lot harder to update the stored configuration. However, Tandem PROM can be used in applications where an update should only be required if there is an error in the original design, and it is acceptable to update all devices manually, through, for example, JTAG. Tandem PROM can also be used to reduce FPGA configuration times in cases where there is no PCIe connection, but there is still the requirement of the FPGA being online within a set amount of time.

Tandem PROM does have the disadvantage that the user design is not able to access the PROM for storing any extra configuration details or for storing data in the PROM—this is because the dual-purpose I/Os that connect to the PROM cannot be reconfigured in the second stage bitstream while the second stage bitstream is being loaded in from the PROM.

Tandem PCIe has the benefit of reducing flash storage requirements because a first stage bitstream can be significantly smaller than a full device bitstream and the second stage bitstream can be stored on mass storage media in the host system. This is not the case for Tandem PROM as all bitstream data is still stored in the flash memory. The total bitstream size will actually increase because splitting the design into two parts reduces the circuit optimizations that can be done and adds configuration overhead.

Tandem PCIe is in most cases more advantageous because it is more flexible, as will be elaborated in the next section, and can also provide a way of updating the FPGA through PCIe in its more advanced configurations. In more embedded applications Tandem PROM might be the better choice because of its reduced complexity compared to Tandem PCIe where a driver also needs to be included in the host system that uploads the second stage bitstream to the FPGA through PCIe at start-up.

2.5.2. (3) Tandem PCIe with Field Updates

Tandem PCIe with Field Updates is an evolution of the Tandem PCIe method and adds the ability to perform field updates on designs using Tandem configuration. It adds support for this by adding functionality to both the hardware and the tools needed to generate the bitstreams. Even though there was already work done into doing field updates at run-time on AMD/Xilinx FPGAs using PR by [16] in 2011, Tandem PCIe with Field Updates is only officially supported on the most recent two AMD FPGA families, Ultrascale and Ultrascale+.

For Tandem PCIe with Field Updates the second stage bitstream is delivered through PCIe just like in normal Tandem PCIe, but with the field updates variant of Tandem configuration it is possible to upload subsequent second stage bitstreams to the FPGA after the initial configuration. The region of the FPGA that is configured by the first stage bitstream stays the same, but the region covered by the second stage—usually all remaining area—can be updated with a compatible bitstream. This update is performed while the PCIe link remains active. However, the user design is of course not able to handle any PCIe requests while the reconfiguration is taking place, but the PCIe subsystem will respond with a correct error code making sure the link stays active.

MCAP

Instead of a one time configuration engine, like with Tandem PROM/Tandem PCIe, the PCIe block in the first stage bitstream contains the Media Configuration Access Port (MCAP) configuration interface that remains active after the initial configuration is done. This interface offers a dedicated connection to the ICAP from one specific integrated PCIe block. It can be accessed through a custom MCAP PCIe extension, or Vendor Specific Extended Capability (VSEC), that is added to the PCIe configuration space. This makes it available on the PCIe bus of the FPGA for use by the host system to perform updates without needing to interrupt regular dataflows. The only disruption it causes is that a minimal amount of traffic is generated on the PCIe bus, around 3-6 MB/s typically [32], which is relatively small compared to even the bandwidth of a single PCIe 2.0 lane at 500 MB/s.

Reconfigurable Stage Twos
With Ultrascale devices the second-stage bitstream that is loaded after the first-stage bitstream always needs to be the original design—the first version before any update—with which the initial design was made. This means that one needs to perform an extra reconfiguration after the initial loading-in to load the most recent field update. However, Ultrascale+ devices have the ability of using *Reconfigurable Stage Twos* which allows for a field update to be loaded straight away as the second-stage bitstream. This could even be used to load different versions depending on what is needed at start-up by writing a different bitstream than the default design if the situation calls for it.

2.5.3. Tandem configuration with general DFX

Unlike the other Tandem configuration methods that can be added to a design without the designer needing to know all the details of the layout on the FPGA, (4) Tandem PCIe + DFX and (5) DFX over PCIe require that the designer lays out the reconfigurable regions that constitute the different parts of the Tandem bitstream stages manually. This makes sense because one would use Tandem PCIe + DFX if the design intends to use partial reconfiguration/DFX and thus would probably want to define multiple reconfigurable regions.

(4) Tandem PCIe + DFX
As the fourth Tandem configuration type, Tandem PCIe + DFX offers the ability to load in partial bitstreams through the MCAP interface on top of offering Tandem configuration. The designer takes more manual control of the layout of the reconfigurable regions than with just Tandem PCIe, because in this method the general DFX technique is combined with Tandem configuration. It is still important to make sure that the region containing the PCIe block that includes the MCAP configuration engine is small enough to keep the loading time of the first stage bitstream low enough to meet the PCIe timing requirements.

(5) DFX over PCIe
DFX over PCIe uses the same MCAP configuration engine as the Tandem PCIe + DFX. However, in the DFX over PCIe configuration, the bitstream does not consist of two separate stages like in all the other Tandem configuration methods. As the fifth Tandem configuration type, DFX over PCIe has one single bitstream that is loaded in at start-up and potentially multiple different partial bitstreams that can be loaded in after initial configuration. The loading in of these partial bitstreams is done over PCIe through the MCAP interface. This means that it does not offer a solution to the long configuration time because the entire initial bitstream is loaded into the FPGA at once. With a big design the bitstream will still be large, therefore the timing requirements for PCIe at start-up cannot be met with this method. However, if the PCIe connection has been established successfully at start-up, DFX over PCIe still offers the possibility of updating the FPGA through DFX while the PCIe link stays active. In conclusion, while DFX over PCIe offers the ability to update the design in the form of partial bitstreams, it does not offer a solution to the PCIe timing requirements at start-up.

Differences between Tandem PCIe + DFX and Tandem PCIe with Field Updates
The main difference between Tandem PCIe + DFX and Tandem PCIe with Field Updates is that with the field updates method the designer does not need to lay out the reconfigurable regions on the FPGA manually. This is logical because choosing to use DFX gives access to many of the benefits of partial reconfiguration, but requires the designer to be knowledgeable about partial reconfiguration. Generally most FPGA designers are not concerned with the exact layout of their design on the chip and only consider I/O mapping and leave the floorplanning of the logic regions up to the automated tools.

3

Approaches

This chapter discusses what approaches were taken in the design of a system to evaluate the usage of partial reconfiguration for meeting the 120 ms timing requirement for Peripheral Component Interconnect Express (PCIe). In Section 3.1, an overview of the Alveo U50 card is given, which is the FPGA PCIe card for which the solution is designed and evaluated. Second the possible software design flows that are available for creating hardware designs on the Alveo U50 are discussed in Section 3.2. Then in Section 3.3 the architecture of the monitoring system designed for the FPGA is discussed. This monitoring system is used to evaluate configuration speed and initialization time when using partial reconfiguration. Lastly the software that was developed to run on the host system is discussed in Section 3.4. The driver and programs running on the host system are responsible for logging the measured data, as well as configuring the FPGA on-the-fly using partial reconfiguration. With the data gathered from the measuring system different performance aspects can be evaluated to characterize partial reconfiguration and the impact of deploying this technique in a design, and whether it is a suitable solution for meeting the 120 ms timing requirement for PCIe.

3.1. Alveo U50 hardware

To evaluate the different solutions for meeting the 120 ms timing requirement for PCIe, the Alveo U50 was used. The Alveo U50 is an FPGA data acceleration card that can be inserted into systems as a PCIe add-in card. It is based around a large Virtex Ultrascale+ FPGA and also has High Bandwidth Memory (HBM) chips that are located on the same package as the FPGA. The Alveo line of accelerator cards is targeted at data center servers, but they can also be used standalone like any FPGA with a PCIe connection. This is how the Alveo U50 was used in the research for this thesis, as an Ultrascale+ FPGA platform with a PCIe connection because the focus of the research was just on the PCIe connection. The added benefit of using a data acceleration card over a standard development board is that the Alveo cards are made to fit into standard computer systems, while the standard development board offered by AMD exceeds the height clearance of all but the largest PC or server cases.

Card components and connections

Figure 3.1 is a system overview of the Alveo U50 card where all the components on the card and all external connections of both the card and the FPGA are shown.
There are three main components on the board of the Alveo U50:

- The Virtex Ultrascale+ FPGA package with 8 GB of HBM memory on package.
- The Quad Serial Peripheral Interface (QSPI) flash, for storing the initial configuration that is loaded into the FPGA at power-up.
- The satellite controller (for out-of-band management).

Most functionality of the card is provided by the Virtex Ultrascale+ FPGA and the other components support the FPGA or add additional functionality. The QSPI flash stores the configuration that is loaded into the FPGA at initial start-up of the card and can be programmed with a user design if desired. The satellite controller adds out-of-band (OOB) management support to the card that can be accessed

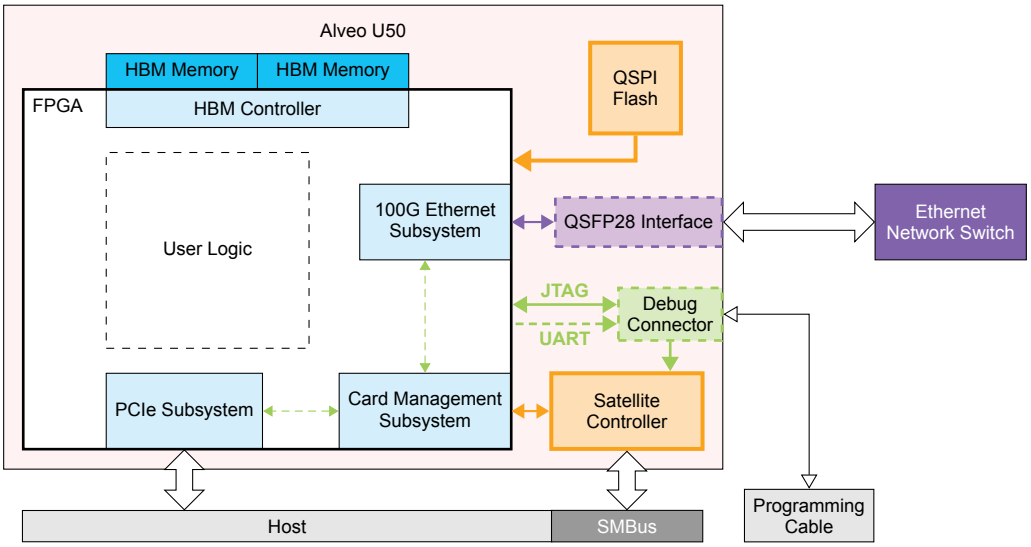


Figure 3.1: Alveo system overview

through the System Management Bus (SMBus). OOB management is typically used in server deployments to monitor the status of the hardware—for example the temperature or power usage—or it can be used to manage the card by applying firmware updates or resetting a card in case it is unresponsive. The benefit of OOB management is that it is always available because it is handled through a controller on the motherboard that is separate—out-of-band—from the host system processor.

The Alveo U50 card has four external connections:

- The PCIe bus connecting to the host system.
- The QSFP Ethernet port for connecting to a 100 Gigabit Ethernet network.
- The Debug connector: containing connections for both JTAG and UART that can be controlled through an Alveo Programming Cable [23].
- The out-of-band management SMBus (connected through pins on the PCIe connector).

The PCIe bus connects the Alveo U50 to the host system by being inserted into an open slot on the host system's motherboard. The PCIe connection is of relevance since this research is focused on the PCIe initialization requirements. The PCIe bus is the main external connection and can be used for communicating data and configurations, as well as offering a data bus with high bandwidth. The QSFP connection is located at the back of the Alveo U50 and thus also at the back of the host system. It can be connected to other cards in the same host system but also to any other card that supports 100 Gbps Ethernet located in an external system, either directly or through an Ethernet network. The QSFP Ethernet connection is not used in this research, but it is critical for utilizing the Alveo U50 to its maximum potential. With the 100 Gbps bandwidth available through the QSFP Ethernet connection it is almost 2/5th of the total bandwidth on the external connections on the Alveo U50, with the PCIe 4.0 connection being the other 3/5ths with 31.5 GB/s or 252 Gbps of throughput. The debug connector is located on the side of the Alveo U50, on the opposite side of the PCIe connector and is only connected during development; is not used when the Alveo U50 card is deployed in a production server. The debug connector is used to program the initial configuration onto the FPGA through its JTAG connection. Once the FPGA has been programmed through JTAG, the UART connection on the debug connector is used to communicate messages from within the design on the FPGA back to the host system. The SMBus connection on the Alveo U50 uses a two-wire bus that is connected through auxiliary pins on the PCIe connector. The SMBus is not used since the OOB management features are not used.

Long-term availability

The Virtex Ultrascale+ FPGA on the Alveo U50 is based on the Ultrascale+ FPGA architecture which is currently one of the most advanced FPGA architectures offered by AMD. Performing the research on such a current platform ensures that any products derived using the methods described in this thesis

can be supported for a long time. Long availability of parts is required if they are used in the design of future medical devices. This is because medical devices have a long lifetime, typically spanning over 15 years, and need to be able to be serviced even at the end of their lifetime. Using the latest generation of FPGA chips from AMD ensures long availability of these parts.

3.2. Software design flow

The Alveo U50 has two ways it can be programmed/configured: Via the traditional Vivado RTL design flow and via the Vitis/XRT flow.

3.2.1. RTL flow

The Vivado register-transfer level (RTL) flow is the same design flow used for any FPGA chip supplied by Advanced Micro Devices (AMD), where traditional hardware description languages (HDLs) like VHDL and Verilog are used to design for the FPGA. However, the workflow for Alveo cards differs as there is no custom circuit board to be designed on which the FPGA needs to be incorporated. Instead, the FPGA is already integrated on a PCIe add-in board designed by AMD. With the RTL flow the entire FPGA fabric can be programmed, but the input/output (I/O) options are limited and focused mainly on data transfer, so there are no general purpose input/output (GPIO) ports that can be used.

The designer can fully program the FPGA and has full control over the PCIe communication through the *Integrated Block for PCI Express* intellectual property (IP) block [28] in FPGA logic. When designing using the RTL flow the Alveo cards can be used in any system that supports the PCIe standard for communication between devices and the host, as long as a driver is developed for the host system.

3.2.2. Vitis/XRT flow

In the Vitis/XRT design flow the FPGA fabric is not programmed directly, instead the FPGA is pre-programmed with an image containing a Shell partition and an empty User partition. After the initial Shell partition has been loaded, the User partition can be programmed using partial reconfiguration to load in processing kernels that run on the FPGA. The Vitis/XRT design flow focuses on using FPGAs for data acceleration purposes and any designs or algorithms to be mapped to the FPGA have to be modularized in acceleration kernels. These kernels can be dynamically loaded into the FPGA and are controlled by software running on the host system.

The software running on the host is called the Xilinx Runtime library (XRT) and applications running in the host operating system (OS) can be written to use this library to offload part of their processing steps to the FPGA. The data that needs to be sent to the FPGA for processing is synced from the host to the FPGA through calls to the application programming interface (API) defined by the XRT, and the XRT driver then handles the communication and data transfer over the PCIe bus—or an Advanced eXtensible Interface (AXI) bus in embedded applications.

3.2.3. Transferring data

The RTL design flow and the Vitis/XRT design flow have different ways in which data transfers are handled, even though both methods use PCIe to transfer the data. The RTL flow allows for more direct access to the PCIe bus, while the Vitis/XRT flow uses a higher level of abstraction using predefined interfaces and an API to initiate data transfers.

Direct memory access

The more direct access to the PCIe bus in the RTL design flow makes it more flexible since it is not limited to any particular system architecture and only needs the system to be compliant with the PCIe standard, which is beneficial when custom solutions are required. However, this requires custom drivers that have to be developed which takes extra time, while in most cases a standard solution can suffice. In case only simple data transfers are required a more standardized solution like direct memory access (DMA) can be used. DMA is a feature available on almost all modern computer systems and offers a mechanism for transferring data between the host system's memory and the add-in card without intervention of the host processor. AMD has standard IP blocks available for performing DMA transfers over PCIe—including a supported driver.

Xilinx direct memory access & Queue direct memory access
AMD offers two solutions for DMA data transfer: Xilinx direct memory access (XDMA) and Queue direct memory access (QDMA). The XDMA IP block has been available for a longer time and is supported on all FPGA models that have integrated PCIe blocks since the Spartan-6. The QDMA IP block is a more recently developed product and is only available for the newest Ultrascale+ FPGAs—the different FPGA architectures are explained in Section 2.1.2. QDMA is an improved DMA implementation with additional queuing capabilities and with support for more advanced PCIe features. The queuing mechanism allows DMA access through multiple separate interfaces, allowing for designs to be separated into different sections or functions, while maintaining a high throughput.

When using the pre-provided XDMA and QDMA drivers and IP blocks from AMD one is limited to the platforms supported by the drivers. Currently, AMD only supports systems with the x86 central processing unit (CPU) architecture. However, the drivers for Linux-based OSs are open-source and can be adapted to other CPU architectures—like for example Advanced RISC Machines (ARM) CPUs, as shown in [37]. While x86 CPUs are still the industry norm, ARM processors are becoming more popular in servers and there is also the newer RISC-V architecture that is gaining more traction and could enter the server space.

Differences Vitis/XRT and RTL flow
When using the Vitis/XRT design flow for the Alveo cards, data transfers to and from the FPGA are handled by the XRT. To perform data transfers the XRT also uses XDMA/QDMA, but when designing using the Vitis/XRT flow a designer only interacts with the high-level API from the XRT and never directly with XDMA/QDMA. The XRT makes it easier to interact programmatically with the FPGA, but using the XRT means that the design can only run on systems supported by the XRT. At this time the XRT is only supported on specific versions of 3 Linux-based OS distributions, and only officially supported on x86 systems. This dependency on the XRT software, and even more broadly the whole XRT/Alveo ecosystem, is something to consider when choosing a workflow for the Alveo cards. In the end it was chosen to go with the Vivado RTL flow because it gives more control.

3.3. FPGA system design
Figure 3.2 shows the system architecture that was designed in this thesis to test the partial reconfiguration of the FPGA and measure the time it takes for the PCIe subsystem to come online and subsequently reconfigure the reconfigurable partition. The architecture is explained starting from the PCIe Subsystem that configures the reconfigurable partition through the Media Configuration Access Port (MCAP) in Section 3.3.1. The MCAP is driven through the PCIe connection by the host system—the software implementation on the host is explained further in Section 3.4, but first the hardware side is described in this section. Next in Section 3.3.2 the Change Monitor subsystem is described which monitors the PCIe Subsystem and reconfigurable partition. Then the AXI State Transmission block which is responsible for sending the detected changes by the Change Monitor over a Universal asynchronous receiver-transmitter (UART) connection is discussed in Section 3.3.3. Finally, in Section 3.3.4 the clock buffer and reset generation that support the operation of the rest of the system are discussed.

3.3.1. PCIe Subsystem & MCAP reconfiguration
The PCIe Subsystem establishes the PCIe connection to the host PC. It also contains the Media Configuration Access Port (MCAP) that is used to reconfigure the reconfigurable partition. The PCIe Subsystem is implemented using the *Integrated Block for PCI Express* [28] which is an IP block available from AMD. There are many parameters available to configure the integrated PCIe block, most importantly Tandem configuration is enabled to meet the timing requirements of PCIe at start-up. There are multiple types of Tandem configuration available, as explained in Section 2.5. The choice was made to use the *Tandem PCIe + DFX* Tandem configuration mode. Tandem PCIe + DFX was chosen because in this mode it is possible to do partial reconfiguration after the stage 2 bitstream is loaded. This will be used to measure reconfiguration rates when using MCAP. It also gives more control over the placement of the PCIe subsystem making it possible to add extra measurement logic to the stage 1 bitstream to monitor the status of the PCIe subsystem.

The MCAP is used to configure the reconfigurable partition block. For testing purposes the reconfigurable partition is loaded with a configuration that lets a status light-emitting diode (LED) blink, with

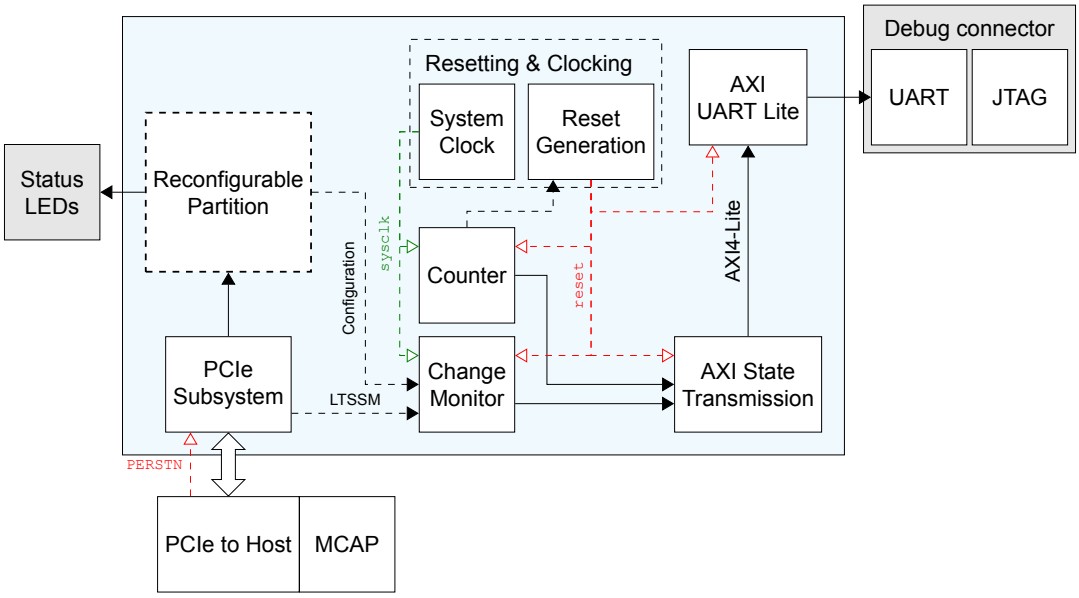


Figure 3.2: System architecture overview for reconfiguration testing and measurement

the different variants lighting up different colour LEDs to be able to visually confirm which variant is currently loaded into the partition.

3.3.2. Monitoring subsystem

The monitoring subsystem monitors the Link Training and Status State Machine (LTSSM) of the PCIe subsystem and the currently loaded configuration in the reconfigurable partition. When it detects a change in either the LTSSM state of the PCIe subsystem or the configuration of the reconfigurable partition it activates the AXI State Transmission block to send a message over UART. This message contains: the time at which the change happened, taken from the Counter block; the new state of the PCIe subsystem, or the new configuration of the reconfigurable partition; and the type of message for the receiver to be able to correctly decode the messages.

The Counter block is included to keep track of the time since the start-up of the system. The counter starts running as soon as the FPGA is configured and thus accurately tracks the time elapsed since the start-up of the FPGA. It runs on the 100 MHz system clock (`sysclk`) that is generated on the Alveo card itself because this clock signal is always stable during operation of the FPGA. Using the system clock diverges from the recommended practices for designing with the *Integrated Block for PCI Express* [28], where it is advised to use the `user_clock` that is generated by the *Integrated Block for PCIe* itself to drive any of the logic connected to the subsystem. However, from simulations it could already be seen that the `user_clock` only becomes active after most of the PCIe initialization is finished and this would not allow the PCIe subsystem states to be monitored if this clock were to be used.

3.3.3. State transmission over UART

To transmit the state changes in the system to the PC UART is used because it is available on the debug connector on the Alveo U50. The Alveo Programming Cable [23] is used to connect to the debug connector and is normally used to program the FPGA through JTAG, but it also offers a UART connection alongside the JTAG connection. UART and JTAG can be used alongside each other because the Alveo Programming Cable is a USB to serial adapter that supports multiple connections through a single USB connection at the same time. The UART connection to the PC is available within the FPGA as two I/O pins: the transmit (TX) and receive (RX) pins.

The UART TX and RX pins are mapped in the FPGA and the AXI UART Lite block is used to drive the UART transmit and receive lines. The AXI UART Lite block is an IP block from AMD that supports sending and receiving UART messages driven by an Advanced eXtensible Interface (AXI) bus. The AXI UART Lite block is an AXI slave and the AXI State Transmission block is the AXI master driving it. More details on how the AXI protocol works and how it is implemented in the AXI State Transmission

block are given in Section 4.2.3.

The UART communication is significantly slower than the changes happening within the system, especially the PCIe LTSSM state can change many times in quick succession generating many messages in a short span of time. The slow communication of the UART means that the Change Monitor block will generate messages quicker than the AXI UART Lite block will be able to send the messages over the UART connection. To make sure all generated messages do get sent the AXI State Transmission block buffers the messages internally in a first in, first out (FIFO) buffer until the AXI UART Lite block is ready to receive more messages. The reading of the messages back out of the FIFO is handled by the flow control of the AXI protocol, which ensures the messages are sent to the AXI UART Lite block when it is ready to receive them.

Transmission UART protocol

When using UART for the data transmission, data is sent one byte/character at a time. The UART protocol ensures the entire character can be decoded correctly at the receiving end by using start and stop bits to indicate at what moment in time transmission of a character starts—Figure 3.3 shows an example of a UART message. The start and stop bits are needed because UART is not clocked by a dedicated clock line and these bits are used to synchronize to the start of a transmission of a character. Having start and stop bits for every transmission ensures correct data transmission of the bits within a single character, but it does not ensure correct data ordering if larger data sets need to be sent. This causes problems when multiple bytes/characters need to be sent over UART that make up one single message, as is the case for the messages that the AXI State Transmission block generates, because in order to properly combine the bytes into a single message the receiver has to know which byte is the first byte.

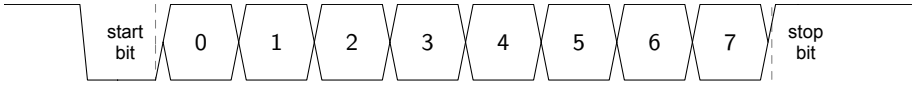


Figure 3.3: Example of a typical UART message

To solve this ordering problem a simple data protocol was implemented on top of UART that uses the first bit of each byte to indicate whether it is the first byte of a message. This allows the host system receiving the messages to determine which byte is the first character of the message and where to start decoding a message. An end bit is not needed and only a start bit is sufficient because the length of the message is known beforehand. The receiver can just count the number of received bytes to determine the end of the message. While using 1 bit in every byte to indicate the start does add a significant overhead to the messages being sent, only leaving 7 bits per character for data, it does not impact the overall system performance because messages already need to be buffered—the buffer does need to be slightly larger because of the overhead.

3.3.4. Clock network & reset generation

Unlike most development boards there is no reset button present on the Alveo U50. This is because the initial resetting of the card is handled through the PCIe reset pin `PERSTN` coming from the host system. The `PERSTN` reset pin is typically connected to the PCIe subsystem which in turn generates its own `user_reset` signal that stays high until the resetting of the PCIe subsystem is complete. The `user_reset` signal can then be used for resetting the logic that interfaces with the PCIe subsystem, since that logic should only become active after the PCIe subsystem is online. However, to be able to monitor the initialization of the PCIe subsystem logic already needs to be active before the PCIe reset is finished, and thus the `user_reset` signal cannot be used. To get around this a reset signal is manually generated instead.

To generate the reset signal the counter that is already present is used. This counter starts running as soon as the FPGA is configured and can be used as a deterministic source of time elapsed since the start-up of the system. This works because in AMD/Xilinx FPGAs memory elements can be programmed to be zero after initial configuration—even without a reset—and thus the counter is in a known state at start-up and starts counting upwards using the `sysclk`.

Clock network
The system clock, called `sysclk` in Figure 3.2, runs at 100 MHz and was chosen to drive the counter block and the reset generation block because it is online directly at start-up. The `user_clk`—coming from the PCIe subsystem—is used for the other blocks like the Change Monitor and the blocks involved in message communication over UART. The `user_clk` runs at a higher clock speed of 250 MHz which allows for faster detection of changes by the Change Monitor, but most importantly it allows for faster data transmission over UART. This is because the *AXI UART Lite* IP block supports higher baud rates when driven with a higher clock speed.

Having two clocks in the design does add complexity. However, the limited usage of the `sysclk` means there are limited clock domain crossings where a signal passes from logic driven by one clock to logic driven by a different clock, since only the reset signal and the output of the counter need to be passed from the `sysclk` clock domain to the `user_clk` clock domain.

3.4. Host system software

To complete the reconfiguration testing system the host needs to communicate in two ways with the FPGA. First the reconfiguration data needs to be sent to the FPGA over PCIe using MCAP. Second the host needs to monitor and log the messages coming back from the FPGA over UART.

3.4.1. MCAP PCIe driver

Dynamically configuring reconfigurable regions on the FPGA with MCAP is done through PCIe. While PCIe is platform independent, the way a program interacts with PCIe on the host system is OS dependent. For security and interoperability reasons programs running on the host system cannot interact directly with PCIe, instead programs can interact with PCIe devices through a driver. A driver is software that is loaded into the kernel of the operating system, where the kernel is the core of the operating system that has direct access to hardware resources, like PCIe. The purpose of a driver is to offer an interface to a particular hardware device like a PCIe card for programs that are running in user-space to be able to interact with this device.

The Media Configuration Access Port is accessible through the Vendor Specific Extended Capability (VSEC) of the PCIe core that is in the Alveo U50 FPGA. VSEC is a mechanism in the PCIe standard to expose vendor-specific registers from the device to the host. Thus, a driver is needed that interacts with the vendor-specific registers, specifically a driver that sends the configuration to the FPGA through the MCAP specific registers. AMD provides drivers to interact with MCAP through these VSEC registers for both Linux and Windows. For Linux-based operating systems there is a ready-made and up-to-date driver available [35]. However, for Windows there are only older drivers available that need modification before they can be used. Windows drivers are typically backwards compatible, but the driver binaries provided by AMD are for Ultrascale FPGAs and do not work for Ultrascale+ FPGAs. The source code is available for the Windows driver and instructions on what modifications need to be done to make the driver work with Ultrascale+ FPGAs are provided [2].

Most Philips systems use the Windows operating system, which means one of the requirements is to develop a system with the host system running Windows. So a Windows driver that works with Ultrascale+ FPGAs needed to be compiled from the provided source code.

3.4.2. Receiving UART messages

To receive the UART messages from the FPGA, the Alveo Programming Cable [23] is used to connect the Alveo U50 to the host system. The programming cable uses a serial to Universal Serial Bus (USB) chip from Future Technology Devices International Ltd (FTDI) that supports both UART and JTAG communication—both JTAG and UART can be used simultaneously. The JTAG connection is used by Vivado to program the FPGA during development, as well as for several on-chip debug methods that can be made available through JTAG. The UART connection is connected from the Alveo Programming Cable straight to the FPGA fabric where it is connected to the AXI UART Lite block that sends messages from the FPGA to the host system.

On the host system the virtual COM port (VCP) drivers [29] from FTDI make the UART connection available to user applications through a virtual serial port. The user program running on the host system then reads out the UART messages by communicating with the virtual serial port using a cross-platform serial library that supports both Windows and Linux operating systems. The readout program first

configures the virtual COM port to the correct baud rate and then continuously reads out the messages coming from the FPGA and logs them. These messages consist of multiple parts and are read one character at a time and combined into complete messages once all parts of a single message are received and decoded.

Decoding of multipart messages

To correctly decode the multipart messages sent over the UART connection, a transmission protocol is used as explained already in Section 3.3.3. The first bit of every byte/character is used to indicate which character is the first part of a message in a sequence of characters. With the start of the message known it can be decoded once all characters that make up the different parts of the message have been received.

The message consists of 3 parts in the following order: message type, event time and the new state. The first character contains the message type, which indicates what type of change the message is communicating, for example whether it is a state change of the PCIe subsystem or a change of the reconfigurable partition. Second, the event time is transmitted; the event time indicates the time elapsed (in clock cycles) since FPGA start-up, as taken from the counter. The event time is split up across 5 characters because the used counter is 32 bits and only 7 bits can be sent every character. With the last character, the new state that has been observed is transmitted. This can be either the new state of the PCIe subsystem or the new configuration of the reconfigurable region depending on the message type. Figure 3.4 shows a full message with all the different parts.

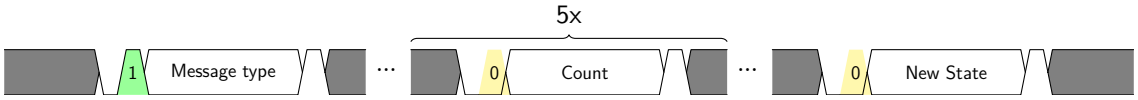


Figure 3.4: Full UART message coming from FPGA. Time progressing from left to right.

4

Implementation

This chapter discusses the implementation of the approaches laid out in the previous chapter. The implementation consists of two main parts: the hardware and the software. First in Section 4.1 a top-level overview is given that shows how the system is segmented into the different parts and what the boundaries are between the hardware and the software. Next in Section 4.2, the hardware implementation is discussed and a detailed description of how the information flows through the subsystems is given. Then in Section 4.3, the implementation of the software is discussed and the details of how the software initiates actions in the hardware are explained and also how data is read back from the hardware.

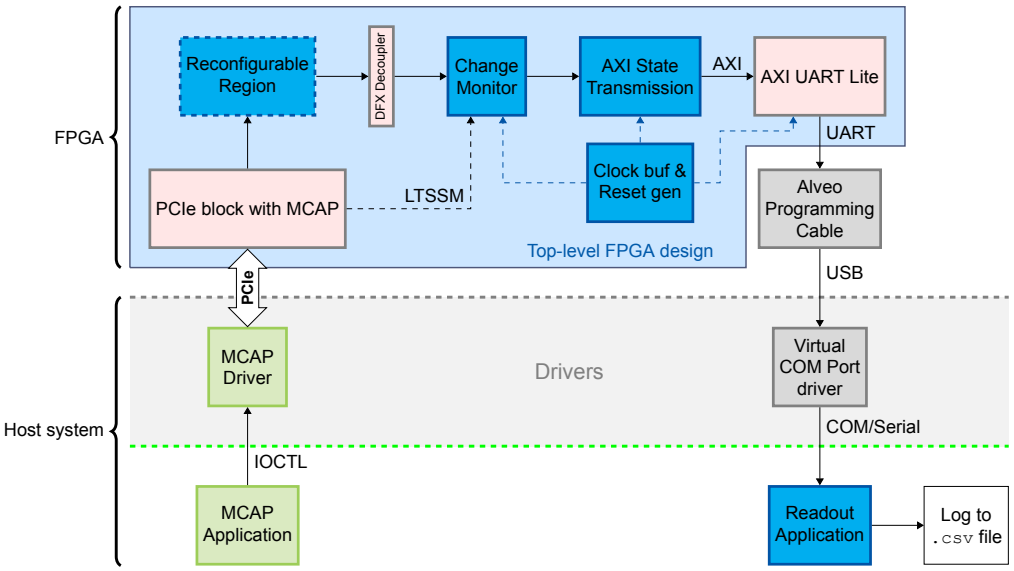


Figure 4.1: Top-level overview. The blocks marked in blue were developed as part of this thesis. This includes the top-level FPGA design. The blocks marked in green were modified to make them work properly.

4.1. Top-level overview

A top-level overview of the system is shown in Figure 4.1. The overview focuses on the way the data flows through the system. Starting at the MCAP Application, where the process of partial reconfiguration starts, and ending at the readout application, where the data observed from the partial reconfiguration process is logged to a `.csv` file. Going through this data flow equals one measurement and can be done multiple times in sequence to generate repeated measurements.

Analysing the timing behaviour of using partial reconfiguration together with PCIe starts with the partial reconfiguration itself. The partial reconfiguration is initiated from software running on the host system. This is done by the MCAP application that queries the MCAP driver for the connected FPGA

cards and then starts communicating with the FPGA using the MCAP driver. The MCAP driver communicates with the FPGA through PCIe using the MCAP port and sends, for example, reconfiguration instructions combined with reconfiguration data over PCIe to the FPGA. There were already examples and source code for an MCAP driver and application provided by AMD, but they needed modification before they could work correctly on modern systems, specifically Windows systems as required by the use-case of Philips. Section 4.3.1 explains the modifications and debugging that were needed to get MCAP working.

PCIe subsystem

On the hardware side of the system—inside the FPGA—the MCAP port receives MCAP requests over the PCIe bus. The MCAP port is integrated into the PCIe intellectual property (IP) block provided by Advanced Micro Devices (AMD). Inside the PCIe block the MCAP port is connected to the reconfiguration resources to be able to dynamically reconfigure parts of the FPGA. The MCAP port receives configuration data over PCIe and then uses the reconfiguration resources to reconfigure the reconfigurable region the received data encodes for. In Section 4.2.1 the details on how the PCIe block is configured to use MCAP are discussed.

Change Monitor

When a new configuration is loaded into the reconfigurable region the Change Monitor will detect this change and initiate a message to be sent to log the change that occurred. The Change Monitor activates the AXI State Transmission block to send the message and the AXI State Transmission block takes care of sending the message over the Advanced eXtensible Interface (AXI) bus. The Change Monitor also monitors the Link Training and Status State Machine (LTSSM) state of the PCIe block and sends a message when the LTSSM state changes. However, the LTSSM in the PCIe block only changes at initial start-up or reconfiguration of the PCIe block itself, meaning almost all messages related to LTSSM state changes will happen at the initial start-up of the system. Reconfiguration of the PCIe block can happen either through reconfiguration of the entire FPGA, or through the dynamic reconfiguration port of the PCIe block—only reconfigurations of the entire FPGA are applicable in this research. The implementation of how the changes are monitored is discussed in Section 4.2.2.

DFX decoupler

In between the reconfigurable region and the Change Monitor is a DFX Decoupler block. The DFX Decoupler block is a block from the AMD IP library that disconnects signals when DFX, or partial reconfiguration, takes place. This is needed because the signals coming out of the reconfigurable region are not stable while reconfiguration is happening. The DFX Decoupler decouples the signals coming out of the reconfigurable region while it is being reconfigured. This is done when the `mcap_design_switch`, coming from the PCIe block, indicates a reconfiguration through MCAP is happening. More details about the DFX Decoupler block are given in Section 4.2.4.

AXI State Transmission

When the Change Monitor initiates the sending of a message the AXI State Transmission block buffers all data that makes up a message. After the data is buffered the AXI State Transmission block will format the message in smaller parts according to the UART protocol described in Section 3.3.3. Finally, the AXI State Transmission block will send the message parts over the AXI bus to the AXI UART Lite block, to be sent over the UART connection. The implementation of the AXI State Transmission block is discussed in Section 4.2.3.

AXI UART Lite

The AXI UART Lite block is a standard block available from the AMD/Xilinx IP library. It receives messages through write transactions on the AXI bus and sends these messages as characters over a UART connection. A more detailed description of the inner workings of the AXI UART Lite block is given in Section 4.2.3. The section discusses how the AXI State Transmission interacts with the AXI UART Lite block to write messages over UART.

Clock buffer & reset generation

The clock buffering and reset generation block is not directly in the dataflow path, but it does serve an important role in the functioning of the entire system. The clock signal coming from outside the FPGA is buffered to generate the `sysclk` signal. A reset signal is also generated that stays high for a number of clock cycles of the `sysclk` clock. Section 4.2.4 discusses how this is implemented.

Alveo Programming Cable

The UART messages transmitted by the AXI UART Lite block end up at the Alveo Programming Cable, which connects to the FPGA through the debug connector on the Alveo U50 card. The Alveo

Programming Cable is normally used to program the FPGA over JTAG, but alongside the JTAG connection it also has a UART connection that connects straight to the FPGA. The Alveo Programming Cable acts as a serial to USB bridge between the FPGA and the host system—this is needed since most modern PCs do not have a native serial interface.

Virtual COM Port driver

To receive UART messages from the Alveo Programming Cable—which acts as a USB-to-serial bridge—a Universal Serial Bus (USB) driver is needed. The Alveo Programming Cable uses a Future Technology Devices International Ltd (FTDI) chip so the Virtual COM Port driver from FTDI [29] is used. The Virtual COM Port driver is an off-the-shelf driver that offers a simple abstraction for applications to interact with FTDI USB-to-serial bridge devices. Through this driver applications can interact with the USB-to-serial device—the Alveo Programming Cable in this case—as if the application were interacting directly with a serial port. The decoding of the UART messages already happens in the Alveo Programming Cable and the data is actually sent over USB in larger chunks, but with the Virtual COM Port driver the messages are still read out one character at a time by the readout application like would be done for a real serial port. The Virtual COM Port driver is used by the readout application running on the host and some more details about the Virtual COM Port driver are given when discussing the readout application in Section 4.3.2.

Readout application

The readout application is the final block in the chain and this program combines the characters received over UART into complete messages. The readout application does this according to the transmission UART protocol described in Section 3.3.3. This protocol dictates that the first bit of each character indicates whether a character is the first character of a message. The implementation of the readout application is discussed in Section 4.3.2.

4.2. Hardware implementation

For the hardware implementation a top-level FPGA design was created that includes both newly developed blocks, and the necessary standard IP blocks from AMD. Figure 4.2 shows an overview of the top-level FPGA design and the blocks that make up the top-level design. The hardware implementation is discussed in four parts, where each part goes into detail on a specific action happening in the hardware and how each part contributes to the dataflow of monitoring partial reconfiguration speed. First in Section 4.2.1 the configuration of the PCIe subsystem is discussed and how the MCAP functionality of the PCIe core is used to configure the reconfigurable region. Then in Section 4.2.2 the way changes in the system are monitored and detected is discussed. After that in Section 4.2.3 the implementation of a custom AXI master that uses the AXI UART Lite block to send messages over UART is discussed. Lastly in Section 4.2.4 the setup of the clock network is discussed, as well as how the reset signals in the system are generated.

4.2.1. PCIe MCAP implementation

When the host system initiates a reconfiguration of the reconfigurable region in the FPGA it does this through the MCAP driver, which in turn communicates with the FPGA through the MCAP protocol. To be able to do reconfiguration of the FPGA through MCAP it needs to be enabled in the *Integrated Block for PCI Express* [28] IP block from AMD. MCAP is enabled by configuring the PCIe block's Tandem configuration/partial reconfiguration setting to *Tandem PCIe + DFX*—using MCAP is also possible in other modes. Enabling this setting adds the ability to the PCIe block to access the reconfiguration resources of the FPGA and make them available over PCIe through MCAP.

First it was tried to use the PCIe block in *DFX over PCIe* mode because the flexibility of a DFX design was needed to implement the test system. But afterwards it was discovered that this results in the initial bitstream being too big to configure within 120 ms. Looking at the implementation it was determined that the extra logic added for the measurement was not causing the large amount of area to be used. Instead, the PCIe core was taking up a larger area than needed. This is because in *DFX over PCIe* the PCIe core is not constrained to a specific area of the chip, this is done because by not constraining the PCIe core the user design can share some of the logic with the PCIe core and allow for more optimal placement and routing of the logic. The PCIe logic was however spilling over a small part of its logic into the adjacent configuration blocks, which meant that these configuration blocks now needed to be configured as well.

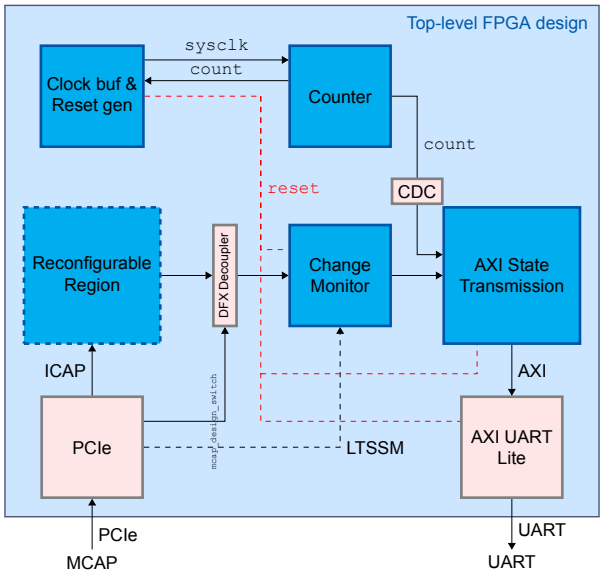


Figure 4.2: Overview of the hardware architecture

Xilinx does not include proper constraint files (.xdc) for the XCU50 chip that is used on the Alveo U50. After some research into this and looking at how the bitstream manipulation tool Byteman [11] handles the XCU50 chip, it was found that the XCU50 chip is a rebatch of the XCVU35P chip. This conclusion seems to match very well because the XCVU35P chip also includes HBM memory and has a similar amount of logic cells. Because of this the IP integrator from AMD would not actually create a PCIe block that properly supports Tandem configuration. To address this the generated PCIe block was edited to include the constraint files from the XCVU35P chip, which were compatible because they had the PCIe block that could be used for Tandem configuration configured to be in the same location as when using a PCIe block with Tandem configuration on the XCU50. It was missing the constraints for which area of the FPGA the stage 1 configuration should be constrained to.

PCIe block configuration
In addition to setting the Tandem configuration/partial reconfiguration mode, some more settings were set. The PCIe lane width is set to x16, to use the maximum lane width of the Alveo U50. The PCIe link speed is 8.0 GT/s, which is the speed for PCIe 3.0. The Alveo U50 also has support for PCIe 4.0, but not over the entire x16 width natively. With PCIe 4.0 the full bandwidth of the Alveo U50 can only be used through two bifurcated x8 lanes, this is why PCIe 3.0 speeds are used instead of PCIe 4.0. In this configuration the Alveo can also be used in a PCIe 2.0 test setup because PCIe is backwards compatible, but setting the speed to PCIe 3.0 ensures the highest speed available is always used.

MCAP
To start an MCAP operation the host system sends a request to the FPGA by asserting the *request for MCAP* bit in the MCAP control register. The FPGA then needs to grant access to the host system to perform MCAP operations by deasserting the *request for release* bit in the MCAP status register. Before the FPGA grants access it should safely finish active communication on the interfaces of any reconfigurable regions. Since the test design in the reconfigurable region is simple and does not need to safely shut off any external interfaces it should grant access to the host system straight away. The host system then polls the MCAP status register until access has been granted, or times out when the FPGA does not grant access. When the host system has been granted control by the FPGA it can perform MCAP operations. As soon as the host system is done with its operations, it deasserts the *request for MCAP* bit in the MCAP control register, which will allow the FPGA to resume normal operation.

PCIe timing requirements
The Tandem configuration mode of the PCIe block was configured to *Tandem PCIe + DFX*, which allows for the dynamic reconfiguration. This Tandem configuration mode makes sure that a minimal stage 1

bitstream is created. It might however be needed to add user logic to the stage 1 either because it needs to be available on start-up or the resources are within the same regions as the PCIe block and cannot be assigned in the stage 2 bitstream. When adding logic to the stage 1 bitstream care must be taken that it is still small enough to meet the 120 ms timing requirement; this needs to be verified manually.

4.2.2. Change monitoring system

The main purpose of the change monitor is to detect a change in the system and generate a message of when this change happened. The change monitoring subsystem monitors two aspects of the system: (1) the PCIe LTSSM states and (2) the reconfigurable region variant currently loaded—both aspects are monitored in the same way. The change monitoring subsystem detects changes by keeping track of the previous state of the signals of interest in a local register. When one of the tracked signals differs from the value in the local register the change monitor will initiate a message to be sent by the AXI State Transmission block by sending a pulse on the `init_send` signal line. At the same time it outputs the newly detected state of the system on the `send_state` signal. The data set on the `send_state` signal will be included in the message being sent. When it regards a change in the LTSSM state `send_state` is set to the new LTSSM state, and when it regards a change in the reconfigurable region variant `send_state` is set to the new variant.

The main purpose of the messages the change monitor generates is logging of events in-time. This is done by including the time elapsed since the start-up of the FPGA in the message. This time is tracked by a counter that starts running as soon as the FPGA is configured. The current value of the counter is included in the message being sent when `init_send` is pulsed high. Combining the `change_type` the `count` value and the `send_state` data gives a full message representing one single change event in the system.

4.2.3. AXI UART transmission

The transmission of UART messages is implemented using two blocks: the AXI State Transmission block and the AXI UART Lite block. The AXI State Transmission block is an AXI master that generates 8 bit messages on the AXI bus. The AXI State Transmission block is designed to be connected to the AXI UART Lite block, which is an AXI slave that receives messages from the AXI bus and sends them over UART to the host system. First details on the AXI UART Lite block are given and then the implementation of how the AXI State Transmission block interacts with the AXI UART Lite block is discussed.

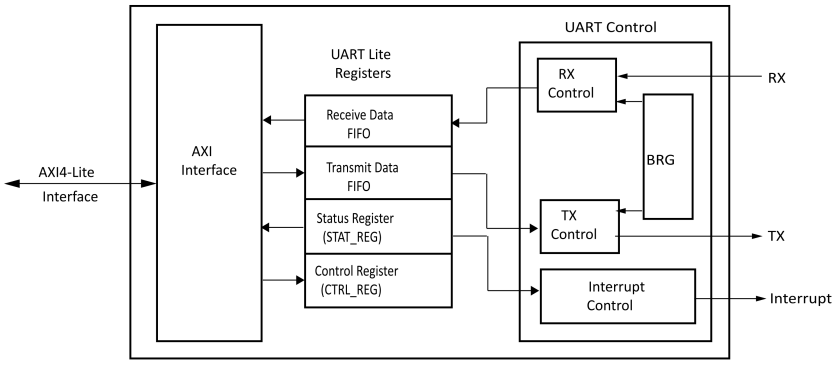


Figure 4.3: Block diagram of AXI UART Lite as given by AMD/Xilinx [4]

AXI UART Lite

The AXI UART Lite IP block from AMD has an AXI4-Lite interface by which data can be sent and received, and by which the status and control registers can be accessed. Figure 4.3 shows a block diagram of the AXI UART Lite block. The AXI4-Lite interface is set up as an AXI slave and all communication is initiated from the AXI master. The receive (RX) and transmit (TX) lines are connected to external UART pins, to which the AXI UART Lite block will send UART characters, and also receive

UART characters from. The outgoing characters are sent from the transmit FIFO which is filled through write transactions from the AXI bus and the incoming characters are written to the receive FIFO which can be read out through the AXI bus.

AXI4-Lite is the simplest variant of the AXI4 interfaces meant for low-throughput memory-mapped communication. The memory-mapped part of AXI is not used for the AXI UART Lite block because every read and every write happens at the same address. Using the AXI4 bus for relatively simple communication like the single 8 bit characters of UART communication might add overhead, but the benefit is that it is a standardized interface. The AXI bus is used throughout all IP blocks offered by AMD, this lowers the burden of designers having to learn many protocols. The cut-down nature of the AXI4-Lite variant means that it can be implemented with relatively little logic in cases where the use case is simple. This is the case because messages only need to be sent over the AXI bus and there is no need to receive messages.

The AXI UART Lite block is configured to use the maximum data width of 8 bits without any parity bits. The baud rate is set to 921600 which is the highest baud rate available, as noted in Section 3.3.4 the baud rate supported by the AXI UART Lite block depends on the given clock frequency. The AXI UART Lite block has two FIFO buffers for buffering characters, one for characters to be transmitted and one for characters that were received. Both the transmit and receive first in, first out (FIFO) buffers can hold up to 16 characters, the size of the buffers is not configurable. The AXI UART Lite block also has a status register that offers extra error information and the status of the FIFOs and also a control register that can be used to reset the FIFOs, both of these registers can be accessed through the AXI bus.

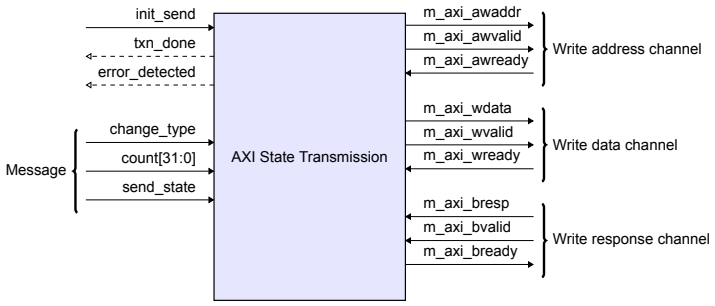


Figure 4.4: Schematic diagram of the AXI State Transmission block

AXI State Transmission

The AXI State Transmission block is an AXI master block connected to the AXI UART Lite slave. Figure 4.4 shows a schematic drawing of the AXI State Transmission block, with the inputs and related feedback on the left and the AXI4-Lite master outputs on the right. When the `init_send` signal is pulsed the AXI State transmission block buffers all message components. The message is then split into multiple characters that match the data width of the AXI UART Lite block—the data width is 8 bits or one byte wide. The message is then sent one byte at a time over the AXI bus to the AXI UART Lite block.

The AXI State transmission block is based on a standard design for an AXI master transceiver from the AMD IP Integrator. The standard transceiver design was simplified, and the reading functionality was removed because the AXI State Transmission block only needs to send messages over the AXI bus and does not need to receive messages. Another simplification was made around the *write address channel* because the AXI UART Lite block only has a single write address which means the write address is constant and does not need to be updated. The *write response channel* implementation is also simplified. The *write response channel* sends responses from the AXI slave to the AXI master to indicate whether a write to the AXI slave was successful. Since there is no way to recover from any write errors that occur the errors are ignored, and it was chosen to consider these characters or messages to be lost and unrecoverable. If any error appears on the *write response channel* the `error_detected` bit is set which can be monitored when debugging the design, but is not used during normal operation.

There are three main parts to sending a message after the `init_send` signal is pulsed: (1) buffering the message data in a register, (2) saving the data of the message in a FIFO buffer, and (3) sending the data of the message over the AXI bus.

(1) Buffering the message in a register

As soon as the `init_send` signal is asserted the message is buffered in a register, this is done because the parts that make up the message are only guaranteed to contain the correct data while `init_send` is high—and a register can save the message in a single clock cycle. The design includes two registers that can quickly buffer up to two messages. Having multiple registers is needed because the Change Monitor block can actually detect changes—and initiate the sending of another message—faster than the AXI State Transmission block can process the message already in the register. If there is still a message in the first message register when `init_send` is pulsed the new message will be saved in the second register. More registers can be added if needed, but it was found that one extra register is enough—mostly based on how fast the LTSSM states of the PCIe block change.

(2) Save the message in the FIFO buffer

As soon as there is a message saved in the register the process of saving the message into a larger FIFO buffer is started. This larger FIFO buffer is included in the design because there can be many messages that need to be sent and the UART connection is slow in transmitting these messages. The saving of the message into FIFO takes multiple clock cycles because the message is stored in smaller parts in the FIFO. The size of these parts already matches the data width of the AXI UART Lite block. The data width of the AXI UART Lite block is only 8 bits or one byte wide and thus the FIFO buffer is also only 8 bits wide.

Before the message parts are saved in the FIFO buffer they are encoded with a 'start bit'. In the first message part the 'start bit' will be set to '1'. This first part is in turn the first character sent over the UART connection and the start bit will help to identify the start of a message on the receiving end. On all other parts of the message the start bit is set to '0'.

Including the start bits one message is 56 bits or 7 bytes long, Figure 4.5 shows the makeup of a message. The first byte is the `change_type` which indicates the type of message that is being sent; the next 5 bytes contain the `count` value, or time, at which the detected event happened; and the last byte contains the `send_state` value, this is additional information that depends on the type of message being sent. Even though the `count` value being sent is only 32 bits, 5 characters are used to transmit the `count` value. This leaves some bits unused that could have been used to transmit a larger `count` value, but the count value is limited to 32 bits by the clock domain crossing macro as explained in the Clock domain crossings Section later on.

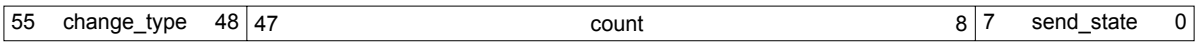


Figure 4.5: Composition of a complete message, including start bits. The numbers represent bit positions of the beginning and endings of the data parts.

To write the message parts from the register to the FIFO buffer a multiplexer combined with a counter is used. The output of the multiplexer selects one message part—which is 7 bits—at a time and the transaction counter determines which part of the message the multiplexer selects. The `wr_en` signal on the FIFO is asserted to enable writing to the FIFO and the transaction counter is incremented each cycle until all parts of the message have been selected and written. The output of the multiplexer is combined with a start bit to make up the full byte that is written to the FIFO. The start bit is '1' for the first part and '0' for the other parts of a message. Once all message parts have been written to the FIFO the `wr_en` signal of the FIFO is turned off until another message needs to be written to the FIFO. The `full` signal of the FIFO buffer is not used during normal operation since the size of the FIFO is dimensioned in such a way that it is large enough to not have any overflows of the buffer.

(3) Send the message over the AXI bus

The last part to transmitting a message out of the AXI State Transmission block is sending it in parts to the AXI UART Lite block over the AXI bus. The parts of the message to be sent are already stored in the correct format in the FIFO buffer. The FIFO buffer operates in first-word fall-through mode which means the next word to be read is already present on data out signal—`dout`—of the FIFO before the read enable signal—`rd_en`—is asserted. The `dout` signal is connected straight to the `m_axi_wdata` output and the `m_axi_wvalid` is used to signal when there is data ready to be transferred over the AXI bus. The AXI protocol uses handshaking to ensure data is transferred correctly over the bus. This is done by the AXI UART Lite block setting `s_axi_wready` high when it has read the data, letting the AXI State Transmission block know it can offer more data to be read.

4.2.4. Clock network & reset generation

There are two more blocks that are not directly involved in data handling, but these blocks are important for the functioning of the top-level hardware design. These two blocks are the clock network and the reset generation blocks. They provide the necessary clock and reset signals that allow the other blocks already discussed in this chapter to function correctly.

Reset generation

For the system to function correctly all blocks need to be reset to a known condition after configuration of the FPGA. The way to achieve this in a typical design where the PCIe subsystem from AMD is used is to use the `user_reset` signal generated by the PCIe block. However, the `user_reset` is not suitable because the initialization of the PCIe subsystem needs to be monitored, and the `user_reset` is only deasserted after the PCIe subsystem is already fully initialized. Another reset signal that is available is the `pcie_perstn` signal which comes from the PCIe connector that could be used as a reset signal, but this signal is unsuitable for the same reason. To monitor the PCIe subsystem correctly the monitoring subsystems need to be online before the PCIe subsystem becomes active, and if the `pcie_perstn` signal were used as a reset signal the PCIe subsystem and the monitoring subsystems would become active at the same time. A custom reset signal is generated from the counter that runs on the `sysclk`, this way it is independent of the `pcie_perstn` signal.

Clock network

The clock network contains two clocks: the `sysclk` and the `user_clk`. The `sysclk` is a 100 MHz clock signal coming from the clock generation chip on the Alveo card which is external to the FPGA. To use the `sysclk` signal a clock buffer is added to the design that converts the signal from the external low-voltage differential signalling (LVDS) interface to a clock signal that can be used inside the FPGA. The `user_clk` is a 250 MHz clock signal coming from the PCIe subsystem block and is generated by the PCIe block as a clock signal to be used by any logic that interfaces with the PCIe subsystem.

The `sysclk` clock signal is guaranteed to be running as soon as the FPGA starts up, this clock signal is used for blocks that need to be online straight away—for example to monitor initialization. Since the `user_clk` clock signal is generated by the PCIe block it is only guaranteed to be running after the initialization of the PCIe block is finished. The `user_clk` clock signal is used for blocks that benefit from having a higher speed clock—for example the AXI UART Lite block.

Clock domain crossings

Since there are two clock signals in the system regions of logic that are driven by one clock and regions driven by the other clock—with a different frequency—need to be considered as separate regions. These regions are called clock domains and care needs to be taken when signals from one clock domain cross to another clock domain. When a signal passes from one clock domain to another clock domain this is called a clock domain crossing (CDC) and such a crossing needs synchronization to prevent possible hold time violations on synchronous logic in the destination clock domain—otherwise it cannot be guaranteed that data signals are read out correctly in the destination clock domain.

There are two clocks in the system and thus two clock domains: the `sysclk` clock domain and the `user_clk` clock domain. There are only two signals that need to cross from the `sysclk` clock domain to the `user_clk` clock domain: the reset signal that is generated, and the `count` value from the counter.

Synchronous reset

The reset signal is derived from the counter and this counter runs on the `sysclk` clock and because of this the logic that generates the reset signal also runs on the `sysclk` clock.

To pass the global reset signal from the `sysclk` clock domain to the rest of the design—which uses the `user_clk` clock domain—the `XPM_CDC_SYNC_RST` CDC block is used. This is a predefined CDC macro from the AMD library that ensures correct synchronous transfer of the reset signal from the `sysclk` clock domain to the `user_clk` clock domain. This simple macro takes the source clock and the destination clock as inputs and uses a series of flip-flops to synchronize the incoming reset signal to the destination clock domain. Having two or more flip-flops in series prevents any metastabilities causing incorrect or non-synchronous resetting of logic.

Gray encoding CDC counter

Since the `count` signal consists of multiple bits this signal is synchronized differently from the reset signal which is just a single bit. The multiple bits in the signal also relate to each other in such a way

that all bits need to be transferred at the same time. This requires a different solution than just putting many synchronizer circuits in parallel because the bits represent a single number and these parallel synchronizers are not guaranteed to have the same delay.

The `count` signal is however a special case because it is a counter and the value of the counter only increases by 1 every clock cycle. This property can be used to transfer the `count` signal to another clock domain without needing to do handshaking—which otherwise would be needed when transferring multi-bit signals across clock domains. That the next value of the counter only differs by 1 each time it is updated allows it to be encoded as a Gray code. Gray codes have the special property that the next value in the sequence always only differs by one bit and this property can be used to make sure the signal is transferred correctly across clock domains. The encoding of the `count` signal into Gray codes and back is handled by the `XPM_CDC_GRAY` CDC macro from AMD. The `XPM_CDC_GRAY` block then ensures that the `count` value is correctly transferred from the `sysclk` clock domain to the `user_clk` clock domain. The `XPM_CDC_GRAY` macro is limited to a maximum of 32 bits and this does limit the timespan that can be tracked using the `count` signal.

DFX Decoupler

The DFX Decoupler that is between the reconfigurable region and the Change Monitor is an AMD IP block. The DFX Decoupler decouples signals in the reconfigurable region from the static region during partial reconfiguration. The Change Monitor connects to the signal which indicates the current variant of the reconfigurable region and this signal needs to be disconnected during partial reconfiguration. Decoupling is needed because signals from the reconfigurable region might glitch during reconfiguration, or they can be driven to `'1'` while the interconnect inside the region is being configured. The DFX Decoupler IP has support for predefined interfaces, for example an AXI4-Lite, but the signal that needs decoupling is a simple bit vector and just the data width needs to be configured in the DFX decoupler block. The DFX Decoupler is switched by the `mcap_design_switch` signal coming from the PCIe block, this signal is asserted when MCAP configuration is initiated.

4.3. Software implementation

From the top-level overview in Section 4.1 it can be seen that the dataflow for a measurement starts and ends by actions from the host system. These two separate actions are: initiating the reconfiguration of the FPGA, and reading out the messages from the UART connection—and they are done by two pieces of software. Figure 4.6 shows a sub-overview of just the software on the host system and how they interact with each other. At the top it also shows the PCIe and UART connections where it connects to the hardware.

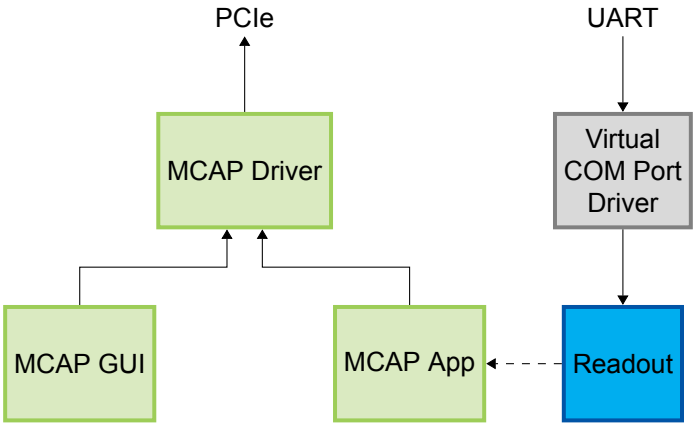


Figure 4.6: Overview of the software architecture

To reconfigure the FPGA, the MCAP protocol is used; Section 4.3.1 discusses how this is achieved. The MCAP protocol communicates through PCIe with the FPGA and requires a driver and an associated application to implement this on the host system. Both the driver and application were modified and recompiled to make them work on the Windows system that is the target host system for this thesis.

To read out the messages from the UART connection a readout program was developed that continuously monitors the Virtual COM Port Driver for new characters. In Section 4.3.2 the implementation of this program is discussed. The readout program decodes and combines the characters received on the UART connection into messages and logs them for further analysis.

4.3.1. MCAP driver & application

To send MCAP data to the FPGA a driver and an application were needed. This section describes the work that was needed to develop these.

Updating driver to work with Ultrascale+

As noted in [2] the base address of the Media Configuration Access Port (MCAP) Vendor Specific Extended Capability (VSEC) is added to the PCIe configuration space at 0x340 Ultrascale devices, and 0x350 for Ultrascale+ devices. The precompiled binaries from the files included with [2] are for Ultrascale devices and because of the differing VSEC offset the precompiled driver does not work with Ultrascale+ devices. Thus, the drivers need to be recompiled from the source code also included with the files supplied by AMD in [2]. The VSEC offset inside the MCAP driver is hardcoded as a single value that needed to be changed to make the driver work with Ultrascale+ devices. Then the driver needs to be recompiled into new binary files that can be loaded in by Windows. The recompiling of the MCAP driver posed some challenges as discussed in the next section.

Recompiling the MCAP Windows driver

To compile applications for the Windows operating system the Windows Software Development Kit (SDK) is needed, and to compile Windows drivers the Windows Driver Kit (WDK) is also needed. The WDK that is installed needs to match the Windows SDK version it is designed for—this means every Windows SDK version has one associated WDK version. Windows kernel-mode drivers are developed using the Kernel-Mode Driver Framework (KMDF), and so is the MCAP driver—the WDK is used to compile drivers built using the KMDF.

The source code files provided by AMD/Xilinx for the MCAP driver are however designed for an older version of the WDK. The application programming interface (API) of the headers files from the WDK have not changed, but the WDK has moved to a new build system and the MCAP driver needed to be migrated to the new build system. The MCAP driver still used the Windows Build Utility (`build.exe`) to build the drivers, but starting in WDK 8 the Windows Build Utility was replaced with the Microsoft Build Engine (MSBuild) build system. This meant that the source files for the MCAP driver would not compile—as is—for Windows versions newer than Windows 8. To compile the MCAP driver for the target platform—which runs Windows 10—the driver was ported over from the `build.exe` build tool to the new MSBuild build tool.

Incompatible WDK versions

After converting the build system from the `build.exe` utility to MSBuild the MCAP driver could be compiled and run. The driver did however not work and crashed at runtime when it was compiled with the latest Windows SDK and associated WDK version. The driver was debugged with the WinDbg driver debugger, but the cause of the crashes could not be found and the problem was most likely not within the MCAP driver code itself. After some research online and trying older versions of the Windows SDK it was found that Windows SDK version 10.0.17763.1 did work and newer versions of the SDK did not—no further time was spent to determine the cause of this incompatibility.

Compiling with older WDK versions

The installed WDK includes the header and library files to build drivers developed with the KMDF, but when installing an older WDK version one critical step in the installation cannot be executed, which causes the compilation of KMDF drivers to fail. After installing the WDK a VSIX package needs to be installed that contains a Visual Studio extension that integrates the WDK into Visual Studio/MSBuild. The WDK Visual Studio extension lets MSBuild know where to find the headers and libraries needed to compile a KMDF based driver. The VSIX package is a `.vsix` file that contains all the data and metadata needed to install the Visual Studio extension.

VSIX packages are however restricted to specific Visual Studio versions and the `.vsix` files supplied with older WDKs cannot be installed in newer versions of Visual Studio. The installation of an older WDK version does succeed, but the installation of the Visual Studio extension fails. Without the

extension Visual Studio does have the header and library files needed to build drivers with the WDK, but it does not know where to find these files and which toolchains they require to be built.

The `.vsix` file is however a simple archive and renaming it to a `.zip` file or instructing an extraction tool to extract it as a ZIP archive allows for manual installation of the files required to make the WDK compilation work. The files of importance from the VSIX package are the `.props` and `.targets` files which instruct MSBuild on how to compile KMDF drivers. These files are already in the proper file structure and just need to be copied into the appropriate Visual Studio installation folders. The `.props` and `.targets` files need to be copied to the `VCTargetsPath` of the Visual Studio version that is in use. Inside the `.vsix` archive the files that need to be installed are either in the `$MSBuild\Microsoft\VC\<version>\` or `$VCTargets` directory—depending on the Visual Studio version the WDK was targeting. The `VCTargetsPath` is the path where Visual Studio stores configuration for all supported targets and toolchains, and for Visual Studio 2022 and 2019 the `VCTargetsPath` is set to `%VSINSTALLDIR%\MSBuild\Microsoft\VC\<version>\` and for older versions this path is `%VSINSTALLDIR%\Common7\IDE\VC\VCTargets\`.

Compiling with older toolset

The WDK version 10.0.17763.1 targets the older *MSVC v141 - Visual Studio 2017 C++* build tools. To use this toolset the `/p:VisualStudioVersion=15.0` argument is passed to MSBuild—version 15.0 corresponds with Visual Studio 2017. The MSBuild tool does however not set the `VSTargetsPath` correctly which means the most recent headers and libraries are used when compiling programs and not the Visual Studio 2017 headers. To fix this the `VSTargetsPath` is set manually by passing: `/p:VCTargetsPath=%VSINSTALLDIR%\MSBuild\Microsoft\VC\v150` to MSBuild, where `%VSINSTALLDIR%` is substituted for the installation directory of Visual Studio. This allows for KMDF drivers to be compiled with Visual Studio 2019 and 2022—this problem does of course not present itself when using Visual Studio 2017.

MCAP applications

Included with the MCAP driver are two applications that can be used to initiate MCAP actions, such as sending a bitstream to the FPGA to reconfigure a reconfigurable region. Both applications serve the same purpose, but one has a graphical user interface (GUI) and the other one has a command line interface (CLI). The CLI application gives the user more granular control over how the MCAP actions are executed, while the GUI offers a more easy to use interface for reading the status of the FPGA and uploading bitstreams.

The MCAP applications were supplied with old `.vcproj` project files for building the applications. These `.vcproj` files do not work with the newer MSBuild build system that was introduced with Visual Studio 2010, this meant the applications also had to be migrated to the new MSBuild build system which uses `.vcxproj` project files instead of `.vcproj` files for building application projects. This migration can be handled automatically by Visual Studio as long as the source code is still compatible with the new Visual Studio toolset, which is the case for both MCAP applications.

Improving the MCAP driver

During usage of the MCAP driver it was noticed that the MCAP control register kept being reset every time a bitstream was written to the FPGA. The MCAP control register is a 32 bit register with different bits in the register activating certain MCAP operations. To write to the control register the driver has to write to the entire register at once, but this can overwrite bits in the register that were set by previously executed actions. To prevent this from happening the previous value of the register needs to be read out and be restored to the control register after the MCAP operation is finished. The MCAP driver already reads out the current value of the control register and uses that as a basis for layering the new MCAP operation on top of. This means the bits already set in the control register will not be overwritten by the new MCAP operation, but after the MCAP operation is done the MCAP driver does not use the saved register value to reset the control register to its original state, instead it uses a default value to reset the control register.

In the recompiled driver this error was corrected by using the already saved value of the control register prior to the MCAP transaction to reset the control register to its original state after any MCAP transaction is finished. This solved some issues that were encountered in situations where reading status data through MCAP would cause the driver to disable the MCAP configuration bit—which enables MCAP to have control over the configuration logic. This overriding happens because reading data

through MCAP also uses the MCAP control register and thus when a read operation is started the MCAP read enable bit is set, which caused the other bits set to be overwritten.

4.3.2. UART readout program

To read the UART data out of the virtual COM port driver, a readout application was developed. This readout application continuously reads a single character, or byte, from the COM port—the COM port is a serial port—and decodes full messages according to the protocol described in Section 3.3.3.

Cross-platform serial library

To read out the UART characters coming from the FPGA the readout program needs to talk to the Virtual COM Port driver over serial. To achieve this an open-source cross-platform serial library is used [30], this library supports multiple operating systems. This allows the readout program to be used on for example either a Windows-based host system, or a Linux-based host system.

Program state logic

Figure 4.7 shows a state diagram of how the readout program handles reading the UART characters and decoding them. The first part of the protocol is detecting a character where the start bit is set to `'1'`. The readout program waits for new characters to be available on the COM port, and when there are new characters available it starts checking them one by one until a character with the start bit set to `'1'` is found, indicating the start of a message. While waiting for characters the readout program is in a blocked state until it receives a character, however the program times out every second to let the user know it is still monitoring the UART and to check whether the user wants to exit the program.

The start character also encodes for the type of message being sent containing the `change_type` data in the character. The program checks if it is a message type that is recognized by the readout program before reading out the rest of the message—if it is not a known message type something went wrong during transmission. There are two message types: `'p'` indicating a PCIe state change, and `'s'` indicating a change in the reconfigurable region.

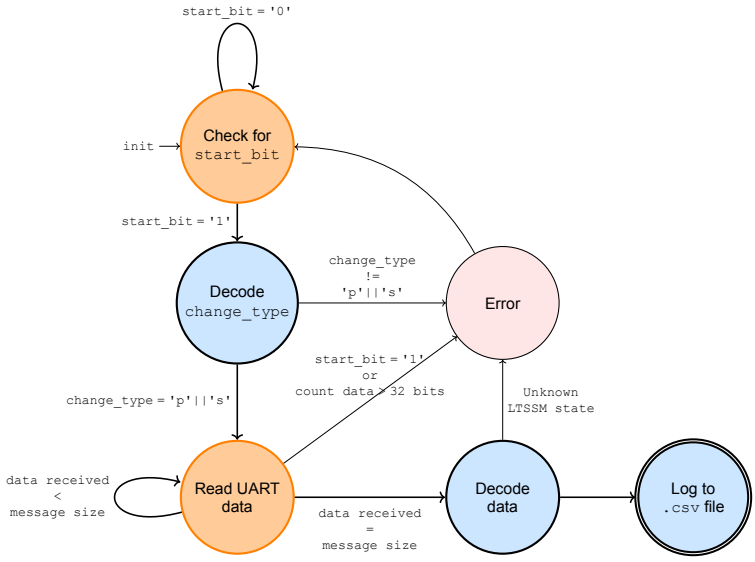


Figure 4.7: Finite state machine of readout program. In the orange states, new UART characters are read in.

After determining the start of a valid message the readout program reads out characters until it has read out one complete message. The program reads out characters one by one checking each character for errors. Once a complete message has been received it is decoded. The decoding consists of two parts, the first part is combining the `count` data into a single 32-bit integer using bit shifts and the second part involves decoding the `send_state` data. For state changes related to the reconfigurable region no decoding is applied since it is just a simple number. However, for PCIe related state changes the LTSSM state contained in the `send_state` data is decoded into a human-readable format by mapping the integer value to the associated names given in the PCIe standard [17].

The final step after successfully receiving and decoding a message is to log it to a file for further manual analysis. This is done to a `.csv` file for easy reading in of the data in any analysis scripts. One line containing the `change_type`, `count` and decoded `send_state` is added to the `.csv` for each message. The program generates a new `.csv` file, with a timestamp in the name of the file, every time the program is run.

5 Results

Using the measurement system described in Chapter 3 and Chapter 4 the PCIe start-up time of the FPGA can be measured. The PCIe initialization speed combined with the loading time of the bitstream gives the total time that needs to be lower than the 120 ms PCIe start-up requirement. Thus, to see if the PCIe start-up requirement is met the bitstream loading time is needed. The bitstream loading time is determined in Section 5.1.

5.1. Bitstream loading time

The measurement system cannot measure the bitstream loading time because nothing is loaded in the FPGA at that time. On the Alveo U50 there is no way to measure when configuration starts and when it finishes because the `DONE` and `INIT_B` pins [24] are not broken out on the PCB. The theoretical bitstream loading time can however be calculated because the model of flash memory used on the Alveo U50 is known. The configuration speed is thus also known. With the configuration speed the loading time can be calculated based on the size of the bitstream.

5.1.1. Theoretical configuration rate

To calculate the theoretical configuration rate the specifications of the flash chip on the Alveo U50 are looked up.
Specifications of the flash storage chip on the Alveo U50:

- Micron
- SPI NOR
- 1.7-2.0V
- 1 Gb (128 MB)
- SPI bus width: x4

The maximum configuration clock frequency is determined by multiple factors that all follow from the FPGA board being used. These factors include the length of the traces between the FPGA and the flash memory chip, the specific make and model of FPGA used, as well as the timing requirements of the SPI connection as stated in the flash memory's data sheet.

The maximum frequency for the SPI communication clock is calculated according to Equation (5.1). This equation is taken from AMD/Xilinx documentation [15]. The maximum clock frequency is limited by three factors: T_{SPITCO}: SPI Clock Low to Output Valid, T_{SPIDCC}: FPGA Data Setup Time, and T_{TPD}: Board Trace Propagation Delay. The equation for the maximum clock frequency is given in Equation (5.1).

$$F_{CCLK,MAX} \leq \frac{1}{T_{SPITCO} + T_{SPIDCC} + T_{TPD}} \tag{5.1}$$

T_{SPITCO} is determined by the flash memory used and is given in the data sheet. For the Alveo U50 this is the Micron MT25QU01G [13]. For this Micron SPI flash memory it is given that $T_{SPITCO} = 5\,\text{ns}$ when there is only a single load connected to the flash memory. T_{SPIDCC} is the FPGA data setup time

taken from the Virtex Ultrascale+ data sheet [6]. T_{SPIDCC} is called T_{DCCK} in the FPGA data sheet. For the FPGA used on the Alveo U50 it is given that $T_{SPIDCC} = 3$ ns. T_{TPD} is determined by the length and geometry of the traces between the flash memory and the FPGA.

However, no value for T_{TPD} is provided by AMD, nor are the design files for the printed circuit board (PCB) of the Alveo U50, which would allow T_{TPD} to be calculated from the trace length. Since the configuration frequency calculation is only a first order estimation, the worst case propagation delay is used instead. For this the propagation delay of the trace is calculated based on an estimation of the trace length and trace delay per unit length. For the length of the trace half the length of the Alveo U50 card is taken—the data sheet [7] states the Alveo U50 is 6.60 inch or 16.76 cm long. For the propagation delay the delay per unit length for the reference PCB stackup [26] from the AMD PCB design guidelines is taken. This is 169.5 ps/in or 66.73 ps/cm. Plugging in these numbers gives $T_{TPD} = 169.5\,\text{ps/in} \cdot (2 * 3.30\,\text{in}) = 1.12\,\text{ns}$. The total trace length is two times the length of the trace because the traces are traversed twice in one data transfer. This is because the clock is driven from the FPGA side, so for one transfer the clock signal goes from the FPGA to the flash memory, then the data is sent from the flash memory back to the FPGA over the same length trace. Using a very rough approximation for T_{TPD} has limited effect on the configuration frequency estimation because T_{TPD} is relatively small compared to T_{SPITCO} and T_{SPIDCC}. $F_{CCLK,MAX}$ can then be calculated and this gives:

$$F_{CCLK,MAX} \leq \frac{1}{T_{SPITCO} + T_{SPIDCC} + T_{TPD}} = \frac{1}{5\,\text{ns} + 3\,\text{ns} + 1.12\,\text{ns}} = 109.65\,\text{MHz}$$

To get the actual maximum configuration rate the frequency tolerance of the clock signal generated by the FPGA has to be taken into account. For the FPGA used on the Alveo U50 the frequency tolerance ($F_{MCCLKTOL}$) is 15%. Equation (5.2) shows how the final configuration rate is calculated.

$$CR \leq \frac{F_{CCLK,MAX}}{1 + F_{MCCLKTOL}} \tag{5.2}$$

This results in a maximum configuration rate of $CR \leq F_{CCLK,MAX}/(1 + F_{MCCLKTOL}) = 109.65/(1 + 0.15) = 95.35$ MHz. Vivado only gives the following discrete options for the configuration rate:

2.7, 5.3, 8.0, 10.6, 21.3, 31.9, 36.4, 51.0, 56.7, 63.8, 72.9, 85.0, 102.0, 127.5 MHz

So the configuration rate for the Alveo U50 is 85 MHz because that is the highest configuration rate that is below 95.35 MHz. 85 MHz also matches the default configuration rate that is used for the Alveo U50 in the Vivado software. The flash memory could support the 127.5 MHz configuration rate, but because an internally generated clock signal from the FPGA is used this rate cannot be achieved. This is because clock signals generated internally in the FPGA are not extremely accurate and thus limit the configuration rate. To get higher configuration rates an external clock that is more accurate can be used—with a lower $F_{MCCLKTOL}$.

5.1.2. Configuration load time

With the configuration rate known the time it takes to fully configure the FPGA can be calculated. To determine the configuration time Equation (5.3) is used. In this equation T is the configuration time in seconds, S is the bitstream size in bits, CR is the configuration clock frequency in Hz, and W is the databus width in bits. The Alveo U50 has one flash memory connected using a Quad Serial Peripheral Interface (QSPI) bus, so the bus size is 4. Note that AMD Ultrascale+ FPGAs support up to two QSPI flash memories in dual QSPI configuration.

$$T = \frac{S}{CR \cdot W} \tag{5.3}$$

For the Alveo U50 the maximum (uncompressed) bitstream size is around 60 MB, or 480 Mbit. To load the full configuration bitstream would then take $T = 480\,\text{Mbit}/(85\,\text{MHz} \cdot 4) \approx 1.4\,\text{s}$. This is way too long to meet the PCIe start-up requirement. Bitstream compression also cannot help here, because taking a very optimistic compression ratio of 10 to 1 would give a configuration time of $T = 48\,\text{Mbit}/(85\,\text{MHz} \cdot 4) \approx 141\,\text{ms}$, which is still longer than the 120 ms requirement. Bitstream compression also relies heavily on parts of the design being empty and containing no configuration, so the larger a design is the less effective compression is.

5.1.3. Full configuration time

When calculating the configuration time it is not sufficient to only consider the loading time of the bitstream. Before the FPGA can start loading any bitstream it first has to complete its power on reset sequence, which takes T_{POR}. The integrated PCIe block product guide [28] states the requirement as Equation (5.4), where T_{FPGA_PWR} is the time it takes for the FPGA power supplies to ramp up and become stable. The exact value for T_{FPGA_PWR} is not known for the Alveo U50 and is assumed to be 2 ms, the same as in the example in [28].

$$T_{POR} + T_{FPGA\_PWR} + T_{stage\ 1} < 120\,\text{ms} \tag{5.4}$$

For Virtex UltraScale+ devices T_{POR} is 65 ms for standard power supply ramp rates and 15 ms for fast ramp rates [28].

5.2. Using Tandem configuration

Tandem configuration (see Section 2.5) can be used to meet the 120 ms start-up requirement for PCIe. Tandem configuration splits the to be programmed configuration into two stages. Stage 1 contains only the PCIe core and stage 2 contains the rest of the design.

5.2.1. Low-level implementation of Tandem configuration

On the lower level Tandem configuration is implemented using partial reconfiguration. Two reconfigurable regions are automatically created for the two stages. The logic elements these reconfigurable regions cover are defined by Pblock constraints. Pblock constraints can also be defined manually for the purpose of custom partial reconfiguration. Pblocks are typically aligned to clock regions of the FPGA and thus lay claim to all resources in that clock region. Certain logic elements can be excluded from being included in the Pblock but by default all logic in that clock region will be included in the reconfigurable region.

5.2.2. Stage 1 Pblock size

One would expect the stage 1 Pblock to be as small as possible, but the AMD tooling always generates the same Pblock spanning 4 clock regions. This happens for all lane widths: x1, x2, x4, x8 or x16. The reason for this is the transceiver layout of the XCU50 FPGA. There is limited documentation available of the XCU50, but since it is a rebatch of the XCVU35P FPGA its documentation can be used. Figure 5.1 shows the transceiver layout of the XCVU35P. The four GTY Quad transceivers (224-227) in the bottom right corner are the ones routed to the 16 PCIe lanes on the Alveo U50. Each GTY Quad transceiver can support up to 4 lanes.

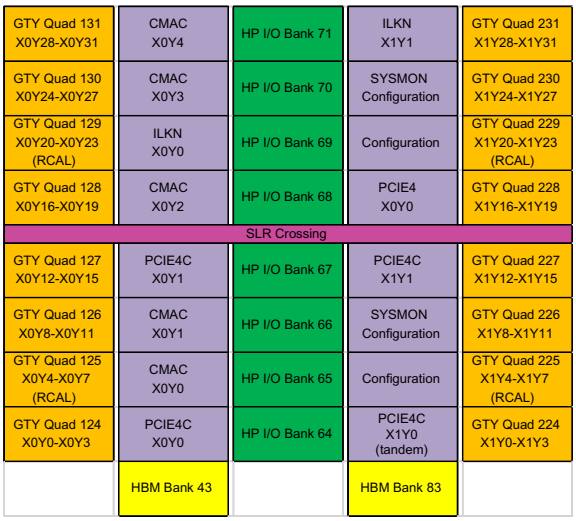


Figure 5.1: Bank diagram of the XCVU35P as given by AMD/Xilinx [25]

There are two PCIe cores available that can be configured together with the transceivers in this corner to implement PCIe functionality. Since Tandem configuration is required the `PCIE4C X1Y0`

core needs to be used—as stated in the brackets in Figure 5.1. There are also additional requirements on which transceiver should be selected as the PCIe lane 0 GT Quad, as stated in the integrated PCIe block product guide [28]. Table 5.1 is an excerpt from the PCIe block guide, and it states that for the PCIe block `PCIE4C X1Y0` the GTY Quad 227 needs to be used to be able to support the maximum link width of x16. Since the Alveo U50 is a x16 PCIe card lane 0 is routed on the PCB to the GTY Quad 227 on the FPGA. This means that this transceiver needs to always be selected as lane 0 when generating the PCIe core in Vivado. This is because PCIe link detection is sent on lane 0 and if the root port sees no receiver on this lane the PCIe card will not be detected. The exception to this is if the host supports PCIe bifurcation, because with PCIe bifurcation the bifurcated lanes are all seen as separate devices. For example if the x16 link is bifurcated into 4 times x4 links each one of those links would have its own lane 0. Note that the Alveo U50 only supports 2 times x8 bifurcation.

Table 5.1: GTY quads that can be used by each PCIe block of the XCVU35P, as given by AMD/Xilinx [28]

PCIe block	GTY quad supporting a maximum link width of		
	x16	x8	x4
`PCIE4C X0Y0`	GTY Quad 127	GTY Quad 126, GTY Quad 125	GTY Quad 124
`PCIE4C X0Y1`	GTY Quad 127	GTY Quad 126, GTY Quad 125	GTY Quad 124
`PCIE4C X1Y1`	GTY Quad 227	GTY Quad 226, GTY Quad 225	GTY Quad 224
`PCIE4C X1Y0`	GTY Quad 227	GTY Quad 226, GTY Quad 225	GTY Quad 224
`PCIE4 X0Y0`	GTY Quad 231	GTY Quad 230, GTY Quad 229	GTY Quad 228

While the bank diagram in Figure 5.1 is not a direct mapping of where these blocks are located on the FPGA they do give an indication of where the transceiver banks are relative to the PCIe cores driving them. The PCIe blocks are actually located in the same clock region as the transceiver to the right/left of them. The stage 1 Pblock needs to encompass both the transceiver and the PCIe block. The Pblock needs to span four clock regions from the bottom right all the way up to the clock region containing GTY Quad 227. It is created as a continuous Pblock because discontinuous Pblocks are strongly discouraged because they make routing very difficult. This means that even when the PCIe link is only x4 wide four clock regions of logic are configured in the stage 1 bitstream even though only one transceiver is needed and it could fit in one clock region.

This is significantly smaller than configuring the whole FPGA. For example, the FPGA on the Alveo U50 has 64 clock regions, so the stage 1 bitstream will only configure $4/64 \cdot 100 = 6.25\%$ of the FPGA.

5.2.3. Reducing bitstream size

The stage 1 bitstream is already smaller allowing for faster configuration, but for the Alveo U50 it is still 4015 KiB because the Pblock covers 4 clock regions. This results in the following configuration time:

$$T = \frac{S}{CR \cdot W} = \frac{4015 \cdot 1024 \cdot 8}{85\,\text{MHz} \cdot 4} \approx 94.5\,\text{ms}$$

Even with fast power supply ramp rates and a T_{POR} of 15 ms this would only barely meet the requirement from Equation (5.4).

Most of the logic in the stage 1 Pblock is unused and not configured. To make use of this sparsity bitstream compression can be used to reduce the size of the stage 1 bitstream, see Section 2.2.6. The compressed bitstream is 2265 KiB. This results in the following configuration time:

$$T = \frac{S}{CR \cdot W} = \frac{2265 \cdot 1024 \cdot 8}{85\,\text{MHz} \cdot 4} \approx 53.3\,\text{ms}$$

With the fast power supply ramp rates this meets the requirement from Equation (5.4) with more margin. It should be noted that this bitstream does include extra measurement logic as seen in Figure 5.2 and is thus larger than normal; Section 5.3 explains what is measured using this logic.

5.2.4. Increasing configuration rate

The configuration rate of the Alveo U50 is not as high as it can be. The Alveo U50 is a reference platform to perform measurements on and a higher configuration rate can be achieved when designing a new

board. In Equation (5.1) T_{SPITCO} and T_{SPIDCC} cannot be optimized because these are fixed values from the QSPI flash memory and the FPGA respectively, and a T_{SPITCO} of 5 ns is already the lowest out of all available QSPI NOR flash memory. The upper limit of the configuration frequency is $1/(5\,\text{ns} + 3\,\text{ns}) = 125\,\text{MHz}$ (assuming no propagation delay), combined with a clock tolerance ($F_{MCCLKTOL}$) of $\pm 15\%$ this gives a maximum configuration rate of $108.7\,\text{MHz}$.

The low clock tolerance can be resolved by using an external master configuration clock (EMCCLK). AMD does not publish the accuracy of the crystal oscillator that is used on the Alveo U50, so it cannot be hypothesized what the configuration rate would have been if an EMCCLK was hooked up. Since the Alveo U50 is a fixed function data center card the clock architecture can not be modified anyway. Assuming a clock with a tolerance of $\pm 0.005\%$, which is a typical accuracy for system clocks already used for FPGAs, the theoretical maximum becomes $124.99\,\text{MHz}$—according to Equation (5.4).

That leaves T_{TPD} as the last limiting factor for the configuration rate. The XCVU35P FPGA in the FSVH2892 package will be taken as an example of how optimal this value can realistically be made. Combining the fact that the FPGA package is 55x55 mm and that in the worst case where the configuration pins are in the middle of the package, $55/2 = 27.5\,\text{mm}$ is needed to route them to the edge of the FPGA. For the Micron MT25QU01G the package size is 6x8mm, and thus $8/2 = 4\,\text{mm}$ is needed to route the pins to the edge. Assuming that the flash memory chip can be placed within $3\,\text{cm}$ of the FPGA this would give the following propagation delay:

$$T_{TPD} = 2 * (27.5 + 4 + 30) * 66.73 \approx 821\,\text{ps}$$

This then results in the following maximum configuration frequency:

$$F_{CCLK,MAX} \leq \frac{1}{T_{SPITCO} + T_{SPIDCC} + T_{TPD}} = \frac{1}{5\,\text{ns} + 3\,\text{ns} + 0.82\,\text{ns}} \approx 113.38\,\text{MHz}$$

Applying Equation (5.4) then gives:

$$CR \leq \frac{F_{CCLK,MAX}}{1 + F_{MCCLKTOL}} = \frac{113.38\,\text{MHz}}{1 + 0.00005} \approx 113.37\,\text{MHz}$$

The $66.73\,\text{ps/cm}$ value for the propagation delay per unit length is taken from the reference PCB stackup guide from AMD [26], but the actual value depends on the PCB manufacturer. This linear propagation delay estimation is a good upper bound of the maximum propagation delay when designing a PCB, but the actual propagation delay will be lower. One could perform a more elaborate analysis or simulation of the exact trace geometry to determine the actual propagation delay, which is most likely lower than the rough estimation.

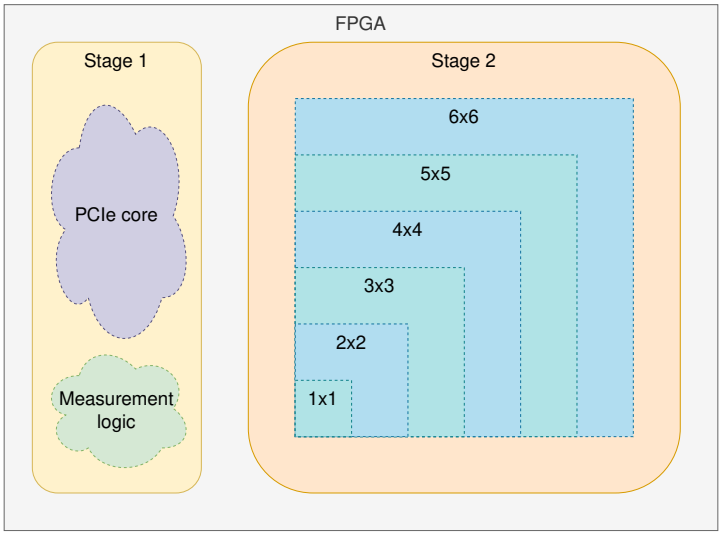


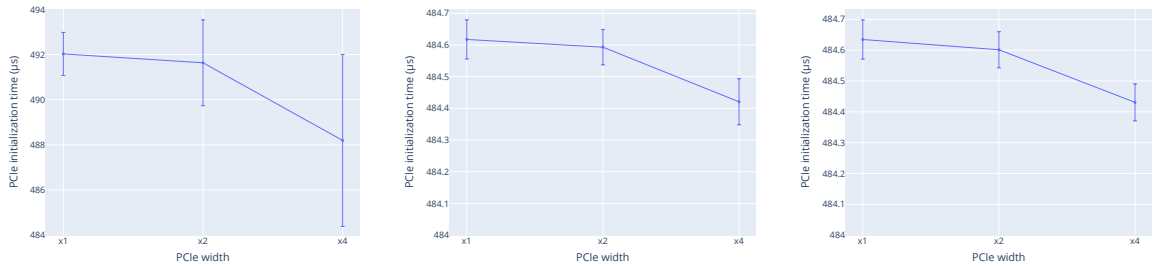
Figure 5.2: Basic overview of Tandem configuration test setup.

5.3. Configuration time measurement

To measure PCIe initialization time and MCAP programming speed a bitstream with the architecture in Figure 5.2 was created. Measurement logic was added to the stage 1 Pblock that monitors the PCIe state and reports any changes in state over UART to a PC for logging. This was used to measure PCIe initialization time. The start and end of partial reconfiguration through MCAP was also measured. Based on this the MCAP configuration speed was determined.

5.3.1. PCIe initialization

To measure PCIe initialization time the PCIe Link Training and Status State Machine (LTSSM) states were monitored. The `Detect.Active` LTSSM state is taken as the start of PCIe initialization and the `L0` LTSSM state is taken as the end of PCIe initialization. Figure 5.3 shows PCIe initialization times for the different PCIe versions. The time from detection to completing link training and being ready to receive PCIe messages was less than 500 µs. The PCIe specification states that the PCIe core needs to be ready within 20 ms of the Fundamental Reset ending. 500 µs is thus a negligible amount of time of this process.

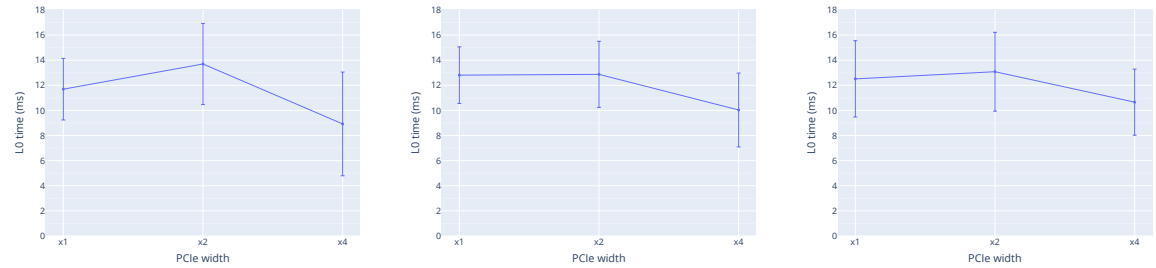


(a) PCIe 1.0 **(b)** PCIe 2.0 **(c)** PCIe 3.0

Figure 5.3: PCIe initialization time

The important measurement is the absolute time it takes to get the PCIe link online. The PCIe link is online when the LTSSM state enters `L0`. Figure 5.4 shows how long it takes to enter the L0 state for the different PCIe versions and lane widths. The L0 time includes the PCIe initialization time and is significantly higher than just the initialization time, but still within the 20 ms PCIe requirement. The longer L0 time is because it includes the `Detect` stage and the `Detect.Active` part of the `Detect` stage typically only starts after a 12 ms timeout.

It should be noted that here the L0 time is taken as the moment the PCIe core is considered ready, but this is an intentionally conservative choice since the PCIe 3.0 base specification only requires that the PCIe device is able to go into the `Detect` LTSSM state within 20 ms of the Fundamental Reset ending. This choice was made because the L0 time could be more consistently be measured.



(a) PCIe 1.0 **(b)** PCIe 2.0 **(c)** PCIe 3.0

Figure 5.4: PCIe L0 time

5.3.2. Stage 2 loading

Once the stage 1 bitstream is loaded in the PCIe core is online and ready to respond to configuration requests. The first requests are enumeration requests where the Root Port detects which PCIe devices are present in the system. Enumeration typically happens shortly after the system is powered on. The PCIe configuration in the stage 1 bitstream already contains all the configuration and address mappings that will be connected to the logic that is in stage 2. But until stage 2 is loaded in it will respond with Unsupported Requests (UR) for transactions to these address regions.

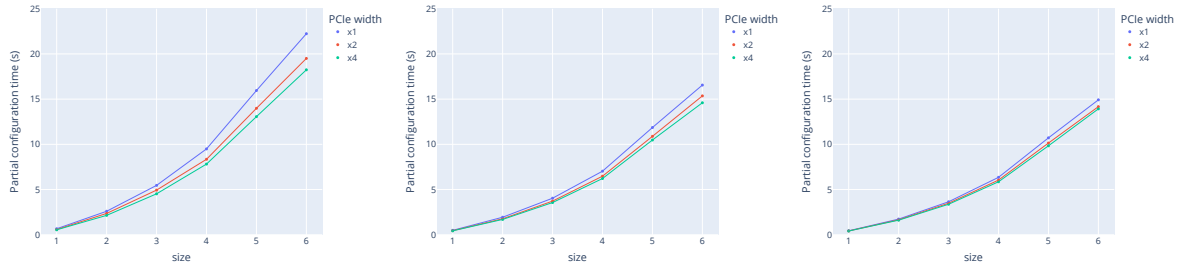
Next the operating system (OS) will load in drivers that will finish the configuration of the PCIe device. When using Tandem PCIe the first driver that needs to be loaded is the MCAP driver that is used to load in the stage 2 bitstream. Because stage 1 already gets the FPGA to a state where it responds to PCIe requests there is no time limit anymore on the loading in of stage 2. After stage 2 is successfully loaded in the MCAP driver needs to write to the design switch register to disable isolation muxing of the PCIe core.

5.4. Partial reconfiguration speed

To measure the MCAP configuration speed partial bitstreams of different sizes were generated. Bitstreams of six different sizes were generated by changing the size of the reconfigurable region inside the FPGA. This was done by creating several designs with a Pblock that covers a range of clock regions. First only one clock region, then for the next step-up 2 by 2 clocks regions, continuing all the way up to 6 by 6. Figure 5.2 shows an approximation of the area that these reconfigurable regions take up inside the design.

5.4.1. MCAP configuration speed

With the measurement setup described in Chapter 4 the partial reconfiguration speed over MCAP was measured. The configuration speed was measured for all PCIe versions supported by the PCIe core and three different PCIe widths. The host system supports up to PCIe 4.0, the PCIe version of each test ran at was limited by configuring the PCIe core inside the FPGA to only support a lower PCIe version. Figure 5.5 shows the partial reconfiguration time for the different size bitstreams. The configuration rate at which the bitstreams were transferred over MCAP were similar for the same PCIe version and width, regardless of the bitstream size. Figure 5.6 shows the configuration rates for the different PCIe versions and lane widths.



(a) PCIe 1.0 **(b)** PCIe 2.0 **(c)** PCIe 3.0

Figure 5.5: MCAP partial reconfiguration time

	PCIe 1.0		PCIe 2.0		PCIe 3.0	
PCIe width	Speed (MB/s)	Speedup	Speed (MB/s)	Speedup	Speed (MB/s)	Speedup
x1	1.55	1.00x	2.10	1.35x	2.33	1.50x
x2	1.75	1.13x	2.28	1.47x	2.44	1.57x
x4	1.89	1.21x	2.38	1.53x	2.51	1.61x

Table 5.2: Configuration rate speedup for different PCIe versions and lane widths

Table 5.2 shows the effect different PCIe versions and lane widths have on the MCAP configuration

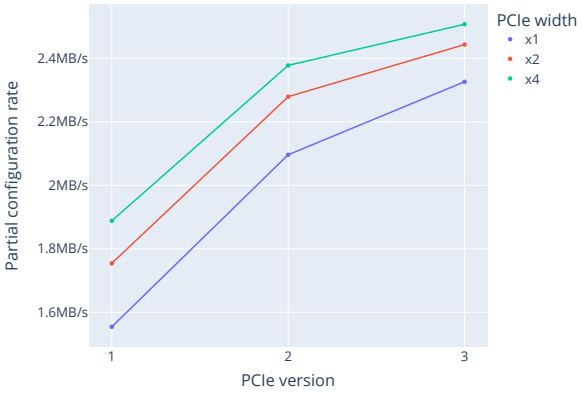


Figure 5.6: MCAP configuration rate

rate. It can be seen that the PCIe version has a large impact on the configuration speed. While the transfer speed of PCIe 2.0 (5.0 GT/s) is twice as big as PCIe 1.0 (2.5 GT/s), this does not result in a 2x speedup of the MCAP configuration speed. This is most likely because of the inherent overhead of the MCAP configuration. The bitstream is written in chunks of 32 bits to the MCAP register. This is done serially and the MCAP Windows driver that was developed does not do any kind of write queuing.

Having a wider PCIe width also consistently increases the configuration speed, but it does not have as big of an effect. The speedup comes from the write to the MCAP register being byte striped across the PCIe lanes, where multiple bytes of the message are sent simultaneously across PCIe lanes and reconstructed at the receiving end. Byte striping is more beneficial for larger transfers, but since the MCAP messages are small it only gives a small speedup.

5.5. Partial reconfiguration rate limits

Some small optimizations could still be done in the MCAP driver to increase the configuration speed when using MCAP. The MCAP protocol is fundamentally bandwidth limited to 3-6 MB/s because it uses PCIe configuration messages which are only 32 bits per transfer. To increase the configuration rate further the bitstream needs to be transferred to the FPGA through higher bandwidth interfaces.

5.5.1. Available configuration bandwidth

The partial reconfiguration rate of MCAP is significantly slower than the maximum rate that the FPGA is capable of. All partial reconfiguration inside an AMD FPGA happens through the Internal Configuration Access Port (ICAP). Even MCAP internally connects to the ICAP to perform its partial reconfigurations. The ICAP has a 32 bit interface that runs at 125 MHz. This frequency is specific to the XCVU35P FPGA, different maximum configuration rates apply for different FPGA speed grades and families. The maximum configuration rate then becomes $125\,\text{MHz} \cdot 32/8 = 500\,\text{MB/s}$.

5.5.2. Partial reconfiguration using DMA

To transfer data faster over the PCIe link direct memory access (DMA) can be used. DMA supports larger data transfers and thus makes better use of all the available PCIe bandwidth. A white paper from AMD describes how to implement such a partial reconfiguration over PCIe DMA solution [32]. This solution uses XDMA (Section 3.2.3) for the DMA transfers and requires custom logic in the user design to connect one of the DMA channels from the PCIe core to the ICAP. Using a PCIe 1.0 x2 connection the ICAP configuration rate can almost be maxed out already. PCIe 1.0 runs at $2.5\,\text{GT/s}$ per lane and uses 8b/10b encoding which has 20% overhead, this results in a usable line rate of $2.5\,\text{GT/s} \cdot 8/10 \div 8 = 250\,\text{MB/s}$ for each lane. For PCIe 1.0 x2 this makes the theoretical throughput $500\,\text{MB/s}$, the actual throughput will be lower because of PCIe protocol overhead. It should be noted that the solution of using DMA to transfer bitstreams to the FPGA can only be used for partial reconfiguration after the stage 2 bitstream is loaded in, so it cannot be used to speed up stage 2 load times when Tandem configuration is enabled.

5.5.3. Partial reconfiguration from device memory

Instead of loading the bitstream over PCIe every time the bitstream can also be placed into device memory and be loaded in from there. The memory inside the FPGA is usually not big enough to hold a bitstream, but most FPGAs have DRAM attached to them of which a part could be reserved for this purpose. The bandwidth to the DRAM is even higher than the PCIe bandwidth, but the main limiting factor for configuration time is still the maximum throughput of the ICAP. There are two benefits to loading the bitstream from device memory instead of PCIe. The first benefit is that the latency between starting the partial reconfiguration and the bitstream actually being transferred to the ICAP is lower than doing partial reconfiguration over DMA. This is because the DMA transfer of the bitstream needs to be started from the host, so there first needs to be a signal send to the host to start the partial reconfiguration. When the bitstream is in the device memory the FPGA itself has control over starting the partial reconfiguration directly. The second benefit is that the reconfiguration time is more predictable because the bitstream is not transferred over a PCIe link that might be contested with other data transfers. There can still be contention on the access to the device memory, but since both the partial reconfiguration circuitry and any other logic accessing the device memory are part of the user design this can be taken into account.

6

Future potential

At the end of the previous chapter loading the bitstreams used for partial reconfiguration from device memory attached to the FPGA was proposed. Using this approach there is a lot of bandwidth available to load bitstreams even faster, for example the High Bandwidth Memory (HBM) memory on the Alveo U50 has 316 GB/s of memory bandwidth. The bottleneck is however the configuration bandwidth available in the FPGA. For the FPGA on the Alveo U50 the configuration rate maxes out at 500 MB/s in the current design. In this chapter the possibilities are explored on how higher (partial) reconfiguration rates can be used if one were to design a new FPGA with such higher rates.

6.1. Higher reconfiguration rates

The amount of bandwidth available through HBM is enormous compared to the reconfiguration bandwidth available in the Ultrascale+ FPGA used in the Alveo U50. To make reconfiguration that much faster in one step would be a big leap. Thus, the analysis of higher reconfiguration rates is split up into multiple smaller throughput increases, based on current state-of-the-art developments and hypothetical future scenarios. Four different scenarios are discussed on how higher reconfiguration rates can be achieved.

6.1.1. Maximizing Ultrascale+ throughput (800 MB/s)

Higher reconfiguration rates can be achieved already on current FPGA architectures. The Ultrascale+ architecture FPGA that the Alveo U50 uses actually supports a reconfiguration rate up to 200 MHz $\cdot$ $32/8 = 800$ MB/s. To achieve this rate special consideration needs to be taken into account in the design that drives the Internal Configuration Access Port (ICAP). Large FPGAs from AMD—like the one on the Alveo U50—are actually constructed from multiple silicon dies connected together through a silicon interposer. To make all silicon dies programmable through the ICAP one of the dies is designated as the master and that ICAP will pass on any configuration data destined for the other dies to the ICAP of those dies. This does however impose a speed limit on the configuration chain because the configuration data now needs to pass through chip-to-chip interconnects that can only run at speeds up to 125 MHz. To get around this speed limit and to be able to achieve the 800 MB/s configuration speed the ICAP on every silicon die needs to be driven locally from within that die.

6.1.2. Versal configuration throughput (6.4 GB/s)

Versal FPGAs are the latest FPGA architecture from AMD. These FPGAs have a revised configuration architecture which allows for higher reconfiguration rates of up to 6.4 GB/s. This is currently the state-of-the-art configuration throughput in commercially available FPGAs. Configuration on Versal FPGAs is frame-based, but it has not been published what data is contained in these frames. A simplified understanding of how this works is that to configure an area of the FPGA a certain amount of frames is needed that contain all the configuration data for that area. To distribute configuration frames throughout the FPGA there is one main configuration frame unit (CFU). The CFU receives configuration data over Advanced eXtensible Interface (AXI) through the network on chip (NoC) and translates this configuration data into frames. These frames are then transferred over the configuration frame interface (CFI)

bus to all the configuration frame (CFrame) engines. The configuration frames are sent to all CFrame engines on the bus and each engine decodes whether that frame is addressed to it. Figure 6.1 shows how the different parts are connected to each other in the Versal configuration architecture. The CFI bus is 128 bits wide and runs at a frequency of 400 MHz. Because the CFrame engines are only addressable through the CFI bus it is the limiting factor for configuration throughput and thus the maximum configuration rate is 6.4 GB/s.

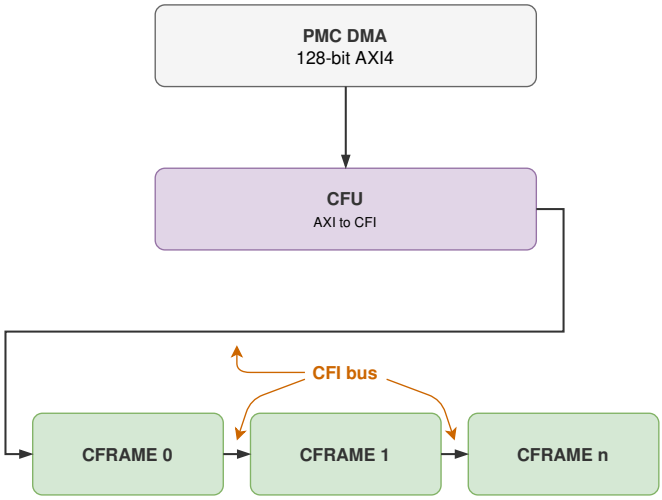


Figure 6.1: Versal configuration architecture

It is unclear from the Versal documentation whether the CFI bus was dimensioned to match the configuration bandwidth available in all CFrame engines or whether the CFrame engines have more bandwidth available. However, considering that the amount of CFrame engines can vary between FPGAs and the amount of them depends on the size of the FPGA there probably is more configuration bandwidth available if a wider bus was used.

6.1.3. DDR4 throughput (40 GB/s)

If one can design a new FPGA with a significantly higher reconfiguration rate then the bottleneck moves from the configuration interface to the memory bandwidth available on the FPGA. For this scenario it is assumed that the FPGA uses conventional DRAM using the DDR4 interface as its memory. One channel of DDR4 typically has a bandwidth of 20 GB/s. Having two channels of DDR4 on this hypothetical FPGA would allow for 40 GB/s of reconfiguration bandwidth. It is assumed that to achieve this configuration rate, power draw and heat dissipation will not be a problem yet. For higher configuration rates this will be taken into account, as is discussed in the next section.

6.1.4. HBM throughput (400 GB/s)

For the next step-up in reconfiguration bandwidth the memory bandwidth available through HBM is taken as the upper limit. There are already FPGAs available with HBM, including the Alveo U50 used throughout this thesis. HBM has a much higher bandwidth than DDR4 because of its very wide bus connecting it to the FPGA—a single HBM2 stack has a 1024 bit-wide bus. With a typical per-pin data rate of 1.6 Gbit/s this gives a bandwidth of $1024\,\text{bit} \cdot 1.6\,\text{Gbit/s} \div 8 = 204.8$ GB/s for a single stack. Most FPGAs equipped with HBM have at least two of these stacks giving a combined memory bandwidth of $2 \cdot 204.8\,\text{GB/s} = 409.6\,\text{GB/s}$.

For convenience the round number of 400 GB/s is targeted for the reconfiguration rate. Configuring the FPGA at such a high configuration rate introduces new challenges. There are two physical constraints that need to be considered: the power budget available to toggle configuration bits, and the heat generated by toggling all those bits. On top of those constraints the extra chip area that is needed for the configuration backbone which distributes the configuration data throughout the FPGA also needs to be taken into account. All these factors are examined in the next sections.

Power limit

To determine how fast an FPGA can be reprogrammed before running into a power limit two things are needed: the maximum amount of power available in the FPGA, and the amount of power consumed to toggle a single configuration bit. The XCVU35P FPGA is taken as an example to see how quickly the power limit is an issue. For the XCVU35P the maximum current is $I_{CCINT} = 135\,\text{A}$ with a voltage of $V_{CCINT} = 0.850\,\text{V}$. V_{CCINT} is the main power rail inside the FPGA and these values are taken from the AMD Power Design Manager (PDM) software. The maximum power budget is then $114.75\,\text{W}$.

To get an estimate of the power consumption per toggled bit the power consumed by LUTs configured as shift registers is taken. Ultrascale+ FPGAs use 6-input LUTs that are implemented using 64-bit SRAMs. On these FPGAs the LUTs inside SLICEM slices can be configured as shift registers. Under the hood this shift register is implemented by using 32 bits of the 64-bit SRAM of the LUT as the actual shift register. Because these shift registers use the configuration SRAMs directly they are a good estimation of how much power is consumed configuring them.

The XCVU35P has 403200 LUTs that can be configured as shift registers. Configuring all these LUTs to be driven by a $200\,\text{MHz}$ clock with a toggle rate of 100% in the PDM software gives a power consumption of $24.059\,\text{W}$—routing calculations were turned off to get only LUT power consumption. Given the amount of LUTs and the fact that 32 bits are used in every LUT the power consumed per bit is calculated as follows:

$$P_{bit} = \frac{24.059}{403200 \cdot 32} = 1.8647\,\mu\text{W}$$

Using the clock frequency the amount of energy per bit toggle is calculated:

$$E_{bit} = \frac{P_{bit}}{200\,\text{MHz}} = 9.323\,\text{fJ}$$

To estimate how much power is consumed when performing configuration a simplified model is used where the configuration memory is assumed to be one big scan chain. This scan chain can then be split up into a number of parallel scan chains to create the desired amount of configuration throughput. Because of the nature of a scan chain every bit that is configured needs to pass through the entire chain to be loaded in. This means that the power consumption scales with the total bitstream size because the bitstream is shifted over by one bit every cycle the scan chain advances. A way to reduce power consumption is by running the scan chain at a lower frequency because the power consumption of the scan chain scales linearly with the frequency it needs to run at. There are also other ways to reduce power consumption of scan chains that will be discussed in the next section.

In Table 6.1 the DDR4 scenario with a desired configuration throughput of $40\,000\,\text{MB/s}$ is illustrated. The number of scan chains is the number of parallel scan chains and this value can be increased to have more scan chains in parallel. The scan chain frequency is calculated based on the desired configuration rate and the number of parallel scan chains; this is the frequency that the scan chains need to run at to meet the configuration throughput requirement. The power consumption is then calculated based on this frequency and the bitstream size.

Table 6.1: Different scan chain configurations for a configuration rate of $40\,000\,\text{MB/s}$ with their estimated power consumption.

Bitstream size (MB)	Configuration rate (MB/s)	# Scan chains	Scan chain frequency (MHz)	Power (W)
1	40,000	1,600	200	1.8647
10	40,000	1,600	200	18.6469
20	40,000	1,600	200	37.2938
40	40,000	1,600	200	74.5877
60	40,000	1,600	200	111.8815
60	40,000	3,200	100	55.9408

It can be seen that the power consumption scales very quickly with the bitstream size. It should be noted that this is a worst-case scenario and that in real configuration architectures some form of addressability is typically used which significantly lowers power consumption. The table does however show that even in the worst case a configuration rate of $40\,000\,\text{MB/s}$ is still within the power budget of $114\,\text{W}$ with only

1600 parallel scan chains, or 50 32-bit buses. If more breathing room in the power budget is desired the number of scan chains can be increased, but that can be costly.

For the targeted configuration rate of $400\,\mathrm{GB/s}$ increasing the number of parallel scan chains to stay within the power budget quickly becomes infeasible, especially when configuring larger bitstreams. To solve this the scan chain can be divided into smaller chains by introducing taps into each chain. Segmenting the scan chains like this makes it possible to turn off the parts of the chain that are not being programmed. One simple way to implement the segmenting is by broadcasting the input of the scan chain to all segments and adding a clock gate before each segment. The clock enable for each segment is then used to turn them on or off. One 2:1 mux at the output of each segment is also needed to support reading back of the scan chains. Adding these clock gates and muxes also adds silicon area overhead; this will be addressed in a later section.

Table 6.2 shows how the adding of segments gives another design variable to reduce power consumption. If required the amount of scan chains can be reduced while keeping power consumption the same. This is done by increasing the number of scan chain segments by the same factor as the number of scan chains is reduced. Having shorter scan chain segments lowers power consumption while at the same time potentially allowing those segments to be clocked at higher frequencies. This can be helpful when the overhead of routing all the scan chains is taking up too many resources.

Table 6.2: Different scan chain configurations for a configuration rate of $40\,000\,\mathrm{MB/s}$ with their estimated power consumption and area overhead.

Bitstream size (MB)	Configuration rate (MB/s)	# Scan chains	# Scan chain segments	Scan chain frequency (MHz)	Power (W)	Area overhead %
1	400,000	8,000	2	400	1.8647	3.71
10	400,000	8,000	2	400	18.6469	0.32
10	400,000	8,000	8	400	4.6617	1.27
20	400,000	8,000	4	400	18.6469	0.32
40	400,000	8,000	4	400	37.2938	0.16
60	400,000	8,000	8	400	27.9704	0.21
60	400,000	4,000	16	800	27.9704	0.21

The design parameters chosen in Table 6.2 are relatively conservative for the available power budget scenario of $114\,\mathrm{W}$. This is because the power consumed by configuring segments of scan chains is not evenly spread throughout the whole FPGA. The simple case where the scan chains are laid out from the top to the bottom and parallel to each other from left to right on the chip is considered. When no segmentation is used all bits in all scan chains are active and the power usage is spread out evenly across the entire chip. With the scan chain divided into segments only one of the segments is active at a time. For example if there are 4 segments and they are activated sequentially from the top to the bottom only the top quarter of the chip will be consuming all the power in the first step, moving downwards with every step. The power delivery network on the FPGA is usually designed to deliver power spread out throughout the entire chip not only a part of the FPGA—this is why it was chosen to stay under one quarter of the power budget. This localized power consumption also has implications for the thermal limit, which will be addressed in the next section.

Thermal limit

When trying to achieve these higher reconfiguration rates the thermal limits also need to be accounted for. The XCVU35P FPGA has a recommended maximum junction temperature of $T_J = 100\,°\mathrm{C}$—a typical value for most commercial chips. The amount of heat produced on a chip is directly related to the amount of power that is consumed. It is assumed that if the power limits are respected the temperature within the FPGA will not get too high. It is also assumed that the thermal conductivity of the FPGA silicon is high enough to transfer the generated heat effectively to the heat spreader on top of the FPGA. This is a relatively safe assumption since the configurable logic blocks (CLBs) inside an FPGA are located quite far apart from each other because of the routing that needs to be in-between them—meaning there is a lot of silicon mass around the active logic to transfer the heat. Given these assumptions it can be concluded that if the FPGA is adequately cooled by a cooling system achieving

higher configuration rates will not run into thermal limits given that the power usage is evenly spread throughout the FPGA.

However, as stated in the previous section the power consumption, and thus heat production, is not spread out evenly throughout the FPGA if the scan chains segments are simply driven sequentially from top to bottom. This can be optimized to spread the power usage out more across the FPGA. The problem is considered as a grid of M scan chains wide by N segments high where M segment activations need to be placed on this grid as far apart from each other as possible. Since the grid is much wider than it is tall the requirement that only one segment can be active per column does not restrict finding optimal placement. The most optimal placement of activations on this grid is a hexagonal packing arrangement of circles. The diameter of the circle is analogous for the distance between activations. A circle packed in a hexagonal arrangement has an area of: $A_C = \sqrt{3}/2 \cdot D^2$ and the total grid area is: $A_G = M \cdot N$. To satisfy the requirement that each column needs to have one activation Equation (6.1) needs to hold. Rewriting this into Equation (6.2) gives the minimum distance that each activation needs to be apart; this is the upper bound.

$$\frac{A_G}{A_C} = \frac{M \cdot N}{\frac{\sqrt{3}}{2} \cdot D^2} = M \tag{6.1}$$

$$D_{min} = \sqrt{\frac{2N}{\sqrt{3}}} \tag{6.2}$$

To maximize the distance between points on the grid a pattern needs to found that has these properties. As an example the formula in Equation (6.3) is used to generate such a pattern. This formula states in which row of a column a point needs to be placed, g is a multiplier that determines how far down the column is shifted, n is the modulus that determines when the pattern repeats, and c is the cycle offset that shifts the pattern over by one. The cycle offset starts at zero and goes up to N to shift the pattern over and make sure all points are hit. The modulus is taken to be equal to the height of the grid $n = N$. The pattern created with this formula repeats after n columns.

$$R(x) = (g \cdot x + c) \bmod n \tag{6.3}$$

For efficient distribution of points it is good to choose a value for g that is coprime with the value of n. This will ensure that all rows are hit before the pattern repeats itself. Choosing non-coprime values for g will leave certain rows unused. Choosing a coprime number for g is not always the most optimal solution for minimizing the distance between all points, the minimum distance can sometimes increase by leaving rows empty, but it was chosen that distribution among rows would best distribute power consumption. There are other values that still create reasonable separation between points, for example choosing $g = \sqrt{n}$ creates a grid that is more regular but still spreads out the points. Given the design directions one can go into to spread the power consumption, and thus heat production, it is concluded that the challenge of thermal hotspots can be mitigated.

Silicon area overhead

For every segment added to the scan chains an extra tap is added which introduces an extra clock gate and mux per tap. This added logic means more area on the silicon will be used for the scan chain. To give an estimate of the area overhead of this logic the size of such logic cells was looked up in the SkyWater Technology Foundry 130 nm process node (SKY130) process design kit (PDK) [3]. SKY130 is an open-source PDK that contains all information needed to design a chip for that process node, which includes simulation models and layouts of the standard cells available on that process node. Open-source PDKs are one of the few places where actual numbers about cell sizes are available because most chip foundries keep that information private for commercial reasons. The sizes in Table 6.3 were used to estimate the area overhead—it should be noted that the smallest size SRAM cell was used from the library; in reality such density may not always be achieved. First the size of the configuration memory was calculated by multiplying the bitstream size with the area of an SRAM cell. Then the area used for the added taps is calculated by multiplying the amount of scan chains by the number of taps, and multiplying that again with the area used by a clock gate and a 2:1 mux.

The overhead percentage of the extra tap logic compared to the configuration memory area is shown in the last column of Table 6.2. It should be noted that the logic needed to distribute the configuration

Table 6.3: SKY130 standard cell areas

Circuit	Cell name	Area (μm^2)
Clock gate	`sdlclkp`	18.768
2:1 mux	`mux2`	11.261
6T SRAM	`sram_sp_cell`	1.896

data to all scan chains and the routing that is needed for that is not taken into account. So this area estimation only evaluates the area added by the introduction of segmentation. The routing for the clock enable signals of segment clock gates is also not taken into account. It can be seen that at small bitstream sizes the cost of adding taps (3.7%) to the scan chains is quite costly without it having significant benefits because the power consumption is already not so high. This can mostly be explained by the fact that logic is added to every scan chain and the ratio between the number of scan chains and the amount of configuration bits is already quite high. As bitstream sizes get bigger and the introduction of taps becomes a necessity the cost of the extra area can be amortized more easily because of the larger configuration memory.

6.2. Case study: PD-Swap

To see what the benefits of having a higher reconfiguration speed bring in practice a case study was done using the PD-Swap paper [36]. PD-Swap uses dynamic partial reconfiguration to swap between prefill- and decode-optimized logic during large language model (LLM) inference on an edge FPGA. Prefill and decode are distinct stages performed once per request and can thus be time-multiplexed. Since the swapping happens for every request, and the reconfiguration is thus on the critical path, it is a good illustration of how faster reconfiguration affects performance.

6.2.1. Baseline reconfiguration throughput

PD-Swap only states how long the partial reconfiguration takes, but does not specify the bitstream size or the reconfiguration throughput. The reconfiguration time is known and to get the reconfiguration throughput the size of the bitstream is needed. The bitstream size is estimated based on the amount of resources used for the dynamic region, which is available in the paper. It is stated that the resources that are in contention are LUT, BRAM and URAM; so only those resources are considered in the estimation of the dynamic region. If a semi-uniform distribution of the resources throughout the FPGA is assumed the size of the dynamic region is determined by the resource of which the highest percentage is used. In reality the distribution of these resources is not uniform, but it is a repeating pattern across the FPGA fabric so over larger regions it averages out. Table 6.4 shows the resource usage of the dynamic region, and it can be seen that BRAM is the resource that is most used at 56.3% usage.

To calculate the partial bitstream size the size of a full bitstream is needed. PD-Swap uses the XCZU5EV FPGA and for this FPGA a full bitstream is 7.82 MB—this number was obtained by generating a bitstream for the XCZU5EV with bitstream compression disabled. Multiplying this with 56.3% gives a partial bitstream size estimate of 4.40 MB. Combined with a configuration time of 45 ms this gives a reconfiguration throughput of $4.40\,\text{MB}/45\,\text{ms} = 97.8\,\text{MB/s}$. This value is around what the typical reconfiguration throughput is of the PCAP reconfiguration port that PD-Swap uses. The maximum configuration throughput for PCAP is measured to be $190.8\,\text{MB/s}$ in [5], but typically a lower speed is reached because of driver overhead, thus $97.8\,\text{MB/s}$ is within the expected bandwidth.

Table 6.4: Resource usage of the dynamic region in PD-Swap [36].

	Total available	**Used in dynamic region**	**Used percentage**
LUT	117,120	32,140	27.4
BRAM	144	81	56.3
URAM	64	10	15.6

6.2.2. Reconfiguration overhead

The time it takes to process one sequence of tokens is not stated in the PD-Swap paper, but it can be calculated from the tokens/s (TK/s) numbers and the given time it takes to complete the projection and FFN computations. The computation time for the projection and FFN computations is only given for a sequence length of 128, so all computations will be for that case. At a sequence length of 128 tokens the throughput for the decode stage is 27.2 TK/s, which means that the decode stage took $128/27.2 = 4.71$ s. The prefill time is given as 0.91 s. Combining this with 31 ms for the projection and FFN computations the total processing time for a 128 sequence is $4.71 + 0.91 + 31$ ms $= 5.65$ s.

With a reconfiguration time of 45 ms this gives a reconfiguration overhead of only 0.8%. The actual reconfiguration overhead is even lower because PD-Swap overlaps the reconfiguration with the projection and FFN computations in the static region to reduce reconfiguration overhead. This reduces the reconfiguration time overhead from 45 ms to 14 ms or 0.2%. This is however only the reconfiguration overhead when a single sequence is processed. For continuous processing of sequences the dynamic region needs to be reconfigured back to the prefill optimized attention logic to prefill tokens from the next sequence. This is extra overhead not mentioned in the PD-Swap paper. Taking this into account the reconfiguration overhead is 59 ms or 1%.

Considering the prefill and decode performance that is gained compared to the TeLLMe baseline implementation—which takes $(128/23.5\,\text{TK/s}) + 0.97 = 6.42$ s—the performance is improved by 0.77 seconds. The reconfiguration overhead reduces this gain by 7.7%.

6.2.3. Speed-up from higher configuration rate

Applying the maximum reconfiguration rate from Section 6.1.1 to the PD-Swap case study can reduce the reconfiguration time to $4.40\,\text{MB}/800\,\text{MB/s} = 5.5$ ms. This would allow the reconfiguration that happens between the prefill and decode stages to be fully overlapped with the computations in the static region, leaving only 5.5 ms in-between every sequence. The performance gained is then only reduced 0.7% by the reconfiguration overhead. The reconfiguration overhead is then only $5.5\,\text{ms}/5.65\,\text{s} = 0.1\%$, which is already so low that there are not many benefits from even higher reconfiguration rates. The total processing time of a complete sequence is very long, and thus the reconfiguration time is already marginal compared to the advantages gained by using partial reconfiguration. The main limitation for the PD-Swap design is the processing time of the decode stage which is bandwidth limited, and this can only be sped up by using an FPGA with higher bandwidth memory. The prefill stage is mainly limited by processing power and can probably benefit from a larger FPGA.

6.3. Case study: Multi-Hash-Chain

For a second case study a system that uses partial reconfiguration to implement the 16XR multi-hash-chain proof-of-work algorithm [31] was analysed. To be able to say something about how much the performance of the system described in [31] can be improved by higher reconfiguration rates the time that is spent doing reconfiguration needs to be determined first.

To determine the average time that is spent doing reconfiguration it first needs to be determined how often the dynamic region is reconfigured. In the worst case sixteen reconfigurations need to be performed to process one batch of X16R hashes, but if a hash algorithm is repeated multiple times in a row no reconfiguration needs to happen between those calculations. This needs to be taken into account when calculating the average time spent doing reconfiguration because as stated in the paper the chance of having no repetitions at all is quite low.

6.3.1. Repetitions distribution

The >N Repetitions line in Table 6.5 shows the distribution of repetitions as given in [31]. It is given that a batch of 62.9 million hashes takes 2.28 seconds to process. Assuming that 2.28 seconds is the average processing time calculated over many batches where the number of repetitions varies according to the distribution of repetitions as given in Table 6.5 the amount of time spent doing partial reconfiguration needs to be calculated based on the average number of repetitions. The number of repetitions reduces the amount of reconfigurations that need to be done.

To calculate the average amount of repetitions the given distribution is first converted to a distribution per number of repetitions: the =N Repetitions line in Table 6.5. The =N Repetitions line in the table is constructed by setting the maximum number of repetitions to 5 so the N>5 percentage of 0.40%

Table 6.5: Distribution of consecutive repetitions of the same hash algorithm

N	0	1	2	3	4	5
>N Repetitions		99.90%	80.50%	23.10%	3.70%	0.40%
=N Repetitions	0.10%	19.40%	57.40%	19.40%	3.30%	0.40%

becomes the percentage of N=5 repetitions. The N=4 repetitions then becomes the N>4 repetitions minus the 0.40% that is already accounted for in the N=5 repetitions value. This is repeated for all values. When N=0 repetitions is reached all the previous percentages add up to 99.90% and the leftover 0.10% is assigned to N=0. The actual probability of having zero repetitions is much lower in reality, so this value is quite conservative, but the remaining 0.10% needs to be assigned for it to be a valid distribution.

6.3.2. Reconfiguration time

The average number of repetitions is the expected value of the =N Repetitions distribution:

$$E[N] = \sum_{n=0}^{5} n \cdot P(N = n) = 2.076$$

The average number of reconfigurations then becomes: $16 - 2.076 = 13.924$. Since the partial bitstreams are streamed in directly from HBM memory to the ICAP it can be assumed that the maximum throughput of $200\,\text{MHz} \cdot 32/8 = 800\,\text{MB/s}$ can be achieved since the HBM bandwidth is larger than the ICAP reconfiguration bandwidth. The combined size of all partial bitstreams of the sixteen different hash algorithms is $352\,\text{MB}$, so on average the reconfiguration of a single hash algorithm takes:

$$\frac{352\,\text{MB}}{800\,\text{MB/s}} \div 16 = 27.5\,\text{ms}$$

The average amount of time spent doing partial reconfiguration then becomes $13.924 \cdot 27.5\,\text{ms} = 382.91\,\text{ms}$. With a total runtime of 2.28 seconds this means that the average reconfiguration overhead is 16.79%.

6.3.3. Speed-up from higher reconfiguration rate

With a reconfiguration overhead of 16.79% speeding up reconfiguration can significantly impact the total system performance. To see how faster reconfiguration rates scale all four scenarios from Section 6.1 are applied to see their impact. Since the system already has optimized the reconfiguration rate and is running on an Ultrascale+ FPGA the scenario from Section 6.1.1 is already the baseline.

To perform the 16XR proof-of-work algorithm a 16 long random sequence of different hash algorithms needs to be calculated in succession. The system in [31] batches the hash calculations for each set in the sequence. First a set of hashes is calculated for the first hash algorithm in the sequence and the results from this are stored in memory. Then the hash algorithm is changed, if needed, using partial reconfiguration and the next hashes in the sequence are calculated. The 16XR proof-of-work algorithm switches up the random sequence of hash algorithms every 60 seconds. This means that any batch that was still in the process of being calculated needs to be thrown away. A relatively large batch size of 62.9 million hashes was chosen mainly to reduce reconfiguration overhead, but this also means a larger amount of work is thrown away every time the sequence switches.

The results of having higher reconfiguration rates are shown in Table 6.6. The most important columns to look at are the reconfiguration and block drop overhead percentages and how lowering these affects the resulting hash rate. The block overhead is what time is wasted by throwing away a batch when the sequence changes after 60 seconds; the shown hash rate accounts for this overhead. The reduce factor is by how much the batch size was reduced to lower the block drop overhead—this of course also increases the reconfiguration overhead again. Jumping to a reconfiguration rate of $6.4\,\text{GB/s}$ already significantly reduces reconfiguration overhead. Halving the batch size with that same reconfiguration rate also reduces the block drop overhead, but ends up with a lower hash rate than only reducing the reconfiguration time. Reducing time spent on reconfiguration increases the hash

rate further, and as soon as the block drop overhead is higher than the reconfiguration overhead the batch size can be reduced further.

Table 6.6: Hash rates for different reconfiguration rates and batch sizes.

Reduce factor	Hashes (million)	Computation time (s)	Runtime (s)	Block drop overhead	Config rate (GB/s)	Config time (ms)	Reconfig overhead	Hash rate (MH/s)	Memory usage (MB)
1	62.90	1.897	2.280	3.80%	0.8	382.9	16.79%	26.54	3750
1	62.90	1.897	1.945	3.24%	6.4	47.9	2.46%	31.29	3750
2	31.45	0.949	0.996	1.66%	6.4	47.9	4.80%	31.04	1875
2	31.45	0.949	0.956	1.59%	40	7.7	0.80%	32.37	1875
4	15.73	0.474	0.482	0.80%	40	7.7	1.59%	32.37	938
4	15.73	0.474	0.475	0.79%	400	0.8	0.16%	32.84	938
8	7.86	0.237	0.238	0.40%	400	0.8	0.32%	32.92	469
16	3.93	0.119	0.119	0.20%	400	0.8	0.64%	32.88	234

Because reducing the batch size increases the hash rate, but also increases the reconfiguration overhead it makes an interesting case where the increased reconfiguration rate can be used to increase performance. However, for higher reconfiguration rates an asymptote is reached where the computation time of the hashes becomes the main limiting factor for the hash rate. An added benefit of reducing the batch size is that less memory is needed to store the intermediate hashes.

7 Conclusion

This thesis investigated how partial reconfiguration, in the form of Tandem configuration, can be used to keep the configuration time of a large field-programmable gate array (FPGA) within the 120 ms start-up requirement of the Peripheral Component Interconnect Express (PCIe) standard. A measurement system was designed (Chapter 3) and implemented (Chapter 4) on an Alveo U50 accelerator card. This was used to measure both the PCIe initialization time and the Media Configuration Access Port (MCAP) reconfiguration speed in Chapter 5. Chapter 6 then explored how much higher reconfiguration rates can be pushed in potential newer devices and what could be gained from this. How this answers the research questions posed in Section 1.4 is summarized in this chapter, including recommendations for future work.

7.1. Research questions

First a summary is given on how the three sub-questions are answered, then the main research question is addressed.

7.1.1. Reducing start-up time

What are the techniques available to reduce the start-up time of a large FPGA so it comes online within the 120 ms PCIe requirement?

Many techniques were evaluated that could reduce the initial start-up time of large FPGAs, specifically AMD FPGAs, and Tandem configuration (Section 2.5) was selected as the most beneficial. Many of the other techniques are either not available anymore or are too costly compared to Tandem configuration. Tandem configuration offers a better solution because it can be implemented on existing FPGAs without additional hardware. Tandem configuration achieves earlier initialization of the PCIe link, to meet the 120 ms PCIe start-up requirement, by taking advantage of the partial reconfiguration capabilities of the FPGA to split up the configuration or bitstream into two stages. Stage 1 is a minimal bitstream containing only the PCIe core, with stage 2 containing the rest of the design. The stage 1 bitstream can be loaded much faster because it is a lot smaller, and thus meets the 120 ms PCIe start-up requirement.

The initialization time of the PCIe core itself was measured to see how much impact that has on the total initialization time. The PCIe initialization time was measured as less than $500\,\mu s$, which is relatively small compared to the 120 ms requirement. The timeout before detection of the PCIe link started was already longer at 12 ms.

7.1.2. Dynamic partial reconfiguration speed

How fast is dynamic partial reconfiguration on current FPGAs?

With the same measurement system partial reconfiguration rates over MCAP were measured. The results showed that the configuration rates over MCAP were very low, under $3\,MB/s$, compared to the maximum achievable rate of $500\,MB/s$ through the Internal Configuration Access Port (ICAP). This limitation is because MCAP transports the bitstream to the FPGA through PCIe configuration messages which have a maximum size of 32 bits. The MCAP configuration rate increased slightly with higher PCIe

versions and wider PCIe link width, but never crossed the 3 MB/s threshold. To get around this a higher bandwidth channel can be established to the FPGA to transfer the bitstream more efficiently to the ICAP. Doing it this way the higher reconfiguration rate of 500 MB/s can be achieved—this was not tested in this thesis, but has been proven in the literature.

7.1.3. Higher reconfiguration rates

How can the reconfiguration time of FPGAs be reduced, and how can this benefit designs using dynamic partial reconfiguration?

To analyse the benefits of higher reconfiguration rates four scenarios were outlined, each with higher reconfiguration rates, in Chapter 6.

In the first scenario the configuration throughput of the ICAP on Ultrascale+ FPGAs is maxed out. There are many ways to saturate the configuration throughput of the ICAP because FPGAs have many different high-speed interfaces, but two methods are most commonly used. The first is to set up a direct memory access (DMA) interface over PCIe, which has a lot more bandwidth than the MCAP method of using configuration messages. The second is to load the bitstream in through DRAM memory connected to the FPGA. One added benefit of this method is that the FPGA itself now has control over when to start the reconfiguration because it does not have to wait for the host to transfer the bitstream over PCIe to the FPGA, removing the host from the critical path.

The second scenario looks beyond what is possible on Ultrascale+ FPGAs and considers the state-of-the-art Versal FPGAs, also from AMD, which have a reconfiguration rate of 6.4 GB/s. Versal FPGAs achieve this much higher reconfiguration rate through a completely revised configuration architecture.

The third scenario assumes that for even higher reconfiguration rates the interface to the FPGA becomes the bottleneck. A 40 GB/s limit matching the typical throughput of two DDR4 channels was chosen for this scenario. This is a hypothetical scenario as there are currently no commercial FPGAs available that have such a high reconfiguration rate.

The fourth scenario looks at a hypothetical FPGA that has High Bandwidth Memory (HBM) memory attached to it with a memory bandwidth of 400 GB/s. To see if a configuration rate of 400 GB/s is even feasible, an analysis was done, and it was concluded that it is possible.

Case studies

To see the benefit of these four scenarios two case studies were done in Chapter 6. For the first case study (PD-Swap) it was determined that the reconfiguration overhead was already quite low (1%) and that by using the reconfiguration rate from the first scenario (800 MB/s) this already lowers to 0.1%—there was no more benefit to be had with higher configuration rates. For the second case study a multi-hash-chain proof-of-work system was analysed. In that case the benefits of higher configuration rates stretched much further and even the highest configuration rate of scenario four (400 GB/s) improved the performance of the system—albeit with diminishing returns. From this it can be concluded that for a dynamic enough workload that effectively makes use of partial reconfiguration there is a place for higher reconfiguration rates.

7.1.4. Main research question

What is the most efficient and reliable option to ensure that an FPGA connected through PCIe is detected at start-up and stays connected during dynamic partial reconfiguration while complying with the PCIe standard?

For AMD FPGAs Tandem configuration is the most reliable and flexible option available to ensure large FPGAs comply with the PCIe standard. Combining it with partial reconfiguration after the stage 2 bitstream has been loaded also gives many options for updating the FPGA while the PCIe connection stays online. Doing all of this over MCAP is however not the most efficient solution, so if reconfiguration speed is important other ways to transfer the bitstream to the configuration engine need to be considered.

7.2. Future work

There is more work that can be done to build on the work in this thesis. Some of the points noted in this section are technical improvements to what was already developed and others are avenues for further exploration.

7.2.1. MCAP Windows driver

The Windows MCAP driver that was developed based on the source code from AMD can be improved in a few ways.

The registers that the MCAP driver needs to write to are part of the Vendor Specific Extended Capabilities (VSECs). Currently, the offset to the MCAP VSEC is hardcoded, but the VSECs can be decoded by enumerating them and determining which one is the VSEC that has the MCAP capability ID. This would also make it possible to have a driver that supports both Ultrascale and Ultrascale+ devices at the same time, since they are at different offsets.

The architecture of the driver, as designed by AMD, could also be improved. The driver currently uses a spin lock to wait for the transfer data over PCIe to complete. Such a spin lock claims the whole processing core until it is released, preventing other drivers and processes from completing any work. It was found that this causes instability of the operating system (OS); this is because the MCAP driver is a kernel-mode driver and runs at the same level as OS drivers.

7.2.2. DMA-based reconfiguration

The DMA based reconfiguration path discussed in Chapter 5 was not implemented or measured in this thesis. Implementing it on the same measurement system would make it possible to verify the predicted gain over MCAP directly, and to measure how close to the ICAP limit such a path can get in practice.

7.2.3. Design space exploration

A more thorough design space exploration (DSE) of an FPGA with higher configuration rates can also be done. The analysis done in this thesis on the actual cost of implementing an FPGA in silicon is very rough and could use more refinement. Such a DSE would also need to include some representative dynamic partial reconfiguration benchmarks because the benefits of higher reconfiguration rates depend a lot on the design that is run on the FPGA.

References

[1] *256Mb, 512Mb, 1Gb StrataFlash Memory Features*. Datasheet. Micron, Mar. 2015, p. 121.

[2] *64761 - Bitstream Loading across the PCI Express Link in UltraScale and UltraScale+ Devices for Tandem PCIe and Partial Reconfiguration*. URL: https://support.xilinx.com/s/article/64761 (visited on 25/12/2021).

[3] Tim Ansell and Mehdi Saligane. 'The Missing Pieces of Open Design Enablement: A Recent History of Google Efforts'. In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '20)*. 2nd Nov. 2020. DOI: 10.1145/3400302.3415736.

[4] *AXI UART Lite v2.0 Product Guide (PG142)*. Product Guide. San Jose, California, U.S.: Xilinx, 5th May 2017, p. 26.

[5] Alex R. Bucknall, Shanker Shreejith and Suhaib A. Fahmy. 'Build Automation and Runtime Abstraction for Partial Reconfiguration on Xilinx Zynq UltraScale+'. In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. 2020.

[6] *DS923 - Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics*. Data Sheet. San Jose, California, U.S.: Xilinx, 23rd June 2021, p. 81.

[7] *DS965 - Alveo U50 Data Center Accelerator Card Data Sheet*. Data Sheet. San Jose, California, U.S.: AMD, 2023.

[8] R.J. Fong, S.J. Harper and P.M. Athanas. 'A Versatile Framework for FPGA Field Updates: An Application of Partial Self-Reconfiguration'. In: *14th IEEE International Workshop on Rapid Systems Prototyping, 2003. Proceedings.* 14th IEEE International Workshop on Rapid Systems Prototyping, 2003. Proceedings. June 2003, pp. 117–123. DOI: 10.1109/IWRSP.2003.1207038.

[9] Sunita Jain, Mrinal Sarmah and David Dye. *XAPP1179 - Using Tandem Configuration for PCIe in the Kintex-7 Connectivity TRD*. Application Note. San Jose, California, U.S.: Xilinx, 25th Oct. 2013, p. 8.

[10] Takahiro Kuwano. *AN98507 - Connecting Cypress SPI Serial Flash to Configure Xilinx FPGAs*. Application Note. Cypress, 6th Feb. 2022, p. 9.

[11] Kristiyan Manev et al. 'Byteman: A Bitstream Manipulation Framework'. In: *2022 International Conference on Field-Programmable Technology (ICFPT)*. 2022 International Conference on Field-Programmable Technology (ICFPT). Dec. 2022, pp. 1–9. DOI: 10.1109/ICFPT56656.2022.9974549.

[12] Clive Maxfield. *Who Made the First PLD?* EE Times. 20th Sept. 2011. URL: https://www.eetimes.com/who-made-the-first-pld/ (visited on 22/01/2023).

[13] *Micron Serial NOR Flash Memory*. 23rd Aug. 2023.

[14] Gordon E Moore et al. 'Progress in Digital Integrated Electronics'. In: *Electron Devices Meeting*. Vol. 21. Washington, DC. 1975, pp. 11–13.

[15] Matt Nielson and Ryan Rumsey. *XAPP1233 - SPI Configuration and Flash Programming in UltraScale FPGAs*. San Jose, California, U.S.: Xilinx, 20th Oct. 2017, p. 43.

[16] Patrick S. Ostler, Michael J. Wirthlin and Joshua E. Jensen. 'FPGA Bootstrapping on PCIe Using Partial Reconfiguration'. In: *2011 International Conference on Reconfigurable Computing and FPGAs*. 2011 International Conference on Reconfigurable Computing and FPGAs. Nov. 2011, pp. 380–385. DOI: 10.1109/ReConFig.2011.40.

[17] *PCI Express® Base Specification Revision 3.0*. Standard. PCI-SIG. Beaverton, Oregon, U.S., 10th Nov. 2010.

[18] Mike Peattie. *XAPP502 - Using a Microprocessor to Configure Xilinx FPGAs via Slave Serial or SelectMAP Mode*. Application Note. Version 1.6.1. San Jose, California, U.S.: Xilinx, 24th Aug. 2009, p. 17.

[19] Luca Pezzarossa et al. ‘Using Dynamic Partial Reconfiguration of FPGAs in Real-Time Systems’. In: *Microprocessors and Microsystems* 61 (Sept. 2018), pp. 198–206. DOI: 10.1016/j.micpro.2018.05.017. URL: https://doi.org/10.1016/j.micpro.2018.05.017.

[20] *S25FL128S, S25FL256S, 128 Mb (16 MB)/256 Mb (32 MB) FL-S Flash SPI Multi-I/O, 3.0V*. Infineon, 10th June 2022, p. 165.

[21] Aaron Gerald Stoddard. ‘Configuration Scrubbing Architectures for High-Reliability FPGA Systems’. MA thesis. Brigham Young University, 2015.

[22] Stephanie Tapp and Ryan Rumsey. *XAPP1220 - UltraScale FPGA BPI Configuration and Flash Programming*. Application Note. San Jose, California, U.S.: AMD Xilinx, 16th Mar. 2022.

[23] *UG1377 - Alveo Programming Cable User Guide*. San Jose, California, U.S.: AMD/Xilinx, 1st Mar. 2023.

[24] *UG570 - UltraScale Architecture Configuration User Guide*. User Guide. San Jose, California, U.S.: Xilinx, 14th Jan. 2022, p. 226.

[25] *UG575 - UltraScale Device Packaging and Pinouts*. User Guide. Version 1.22. San Jose, California, U.S.: AMD, 10th Mar. 2026. URL: https://docs.amd.com/r/en-US/ug575-ultrascale-pkg-pinout (visited on 26/07/2026).

[26] *UG583 - UltraScale Architecture PCB Design User Guide*. User Guide. San Jose, California, U.S.: AMD, 19th May 2023, p. 359.

[27] *UG909 - Vivado Design Suite User Guide Dynamic Function eXchange*. User Guide. San Jose, California, U.S.: Xilinx, 17th Dec. 2021, p. 250. URL: https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2021_2/ug909-vivado-partial-reconfiguration.pdf.

[28] *UltraScale+ Devices Integrated Block for PCI Express v1.3 Product Guide (PG213)*. Product Guide. San Jose, California, U.S.: AMD Xilinx, 20th Nov. 2025. URL: https://docs.amd.com/r/en-US/pg213-pcie4-ultrascale-plus (visited on 26/07/2026).

[29] *VCP Drivers - FTDI*. 13th July 2020. URL: https://ftdichip.com/drivers/vcp-drivers/,%20https://ftdichip.com/drivers/vcp-drivers/ (visited on 06/05/2023).

[30] William Woodall. *Serial Communication Library*. 5th June 2023. URL: https://github.com/wjwwood/serial (visited on 05/06/2023).

[31] Tong Wu and Oliver Diessel. ‘Leveraging FPGA Runtime Reconfigurability to Implement Multi-Hash-Chain Proof-of-Work’. In: *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). New York City, NY, USA: IEEE, May 2022, pp. 1–5. DOI: 10.1109/FCCM53951.2022.9786081. URL: https://ieeexplore.ieee.org/document/9786081.

[32] *XAPP1338 - Fast Partial Reconfiguration Over PCI Express*. Application Note. San Jose, California, U.S.: Xilinx, 11th Mar. 2019, p. 15. URL: https://www.xilinx.com/content/dam/xilinx/support/documentation/application_notes/xapp1338-fast-partial-reconfiguration-pci-express.pdf.

[33] *XCN23009 - Product Discontinuation Notice for XC9500XL, CoolRunner XPLA 3, CoolRunner II, Spartan II, and Spartan 3, 3A, 3AN, 3E, 3ADSP Product Families*. Product Discontinuation Notice. Version 1.1. Santa Clara, California, U.S.: AMD, 29th Jan. 2024, p. 7. URL: https://docs.amd.com/v/u/en-US/XCN23009.

[34] Yuhua Xu, Jie Luo and Wei Sun. ‘Flare: An FPGA-Based Full Precision Low Power CNN Accelerator with Reconfigurable Structure’. In: *Sensors* 24.7 (31st Mar. 2024), p. 2239. DOI: 10.3390/s24072239. URL: https://www.mdpi.com/1424-8220/24/7/2239.

[35] *XVSEC Linux Driver*. Xilinx XVSEC Software. 11th Aug. 2022. URL: https://xilinx.github.io/dma_ip_drivers/master/XVSEC/xvsec_linux/index.html (visited on 05/05/2023).

[36] Yifan Zhang et al. *PD-Swap: Prefill-Decode Logic Swapping for End-to-End LLM Inference on Edge FPGAs via Dynamic Partial Reconfiguration*. 12th Dec. 2025. arXiv: 2512.11550 [cs.AR]. URL: https://arxiv.org/abs/2512.11550.

[37] Zhonghao Zhang and Zhengxiang Li. 'ARM and FPGA Heterogeneous Accelerated Processing System Based on HLS and PCIe'. In: *2021 4th International Conference on Information and Computer Technologies (ICICT)*. 2021 4th International Conference on Information and Computer Technologies (ICICT). Mar. 2021, pp. 300–304. DOI: 10.1109/ICICT52872.2021.00055.