TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Deployment of hardware-extended containers on a soft-core RISC-V implementation

---

Author:
Konstantinos
AMPLIANITIS

Thesis Committee:
Assoc. Sotirios IOANNIDIS
Prof. Apostolos DOLLAS
Dr. Konstantinos
GEORGOPOULOS

A thesis submitted in fulfillment of the requirements
for the diploma of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

October 3, 2023

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Deployment of hardware-extended containers on a soft-core RISC-V implementation

by Konstantinos AMPLIANITIS

With hardware designs becoming more and more complicated in order to serve the current age demands, many hardware teams tend to use reconfigurable hardware to test their designs before putting them to market as a cost-effective practice. These demands also make the hardware teams divided into sections to fulfill the requirements of the design and consequently, cloud services are becoming even more necessary. With those arguments in mind, this thesis provides a solution for a RISC-V deployment platform in order for developers or teams thereof, to upload and test their custom hardware base on a complete RISC-V processor.

Acknowledgements

Starting this thesis, I would like to thank Prof. Sotirios Ioannidis (TUC) for having trust in me in order to complete this assignment and for being my supervisor. I would also feel the need to thank Dr. Konstantinos Georgopoulos and ECE MSc Andreas Brokalakis for providing guidance and support during the work and writing of this thesis. I would also like to express my gratitude to Prof. Apostolos Dollas (TUC) for being the third member of the committee and for evaluating my thesis.

Additionally, I would like to express my gratitude to all the members of the Microprocessors and Hardware Lab (MHL) for covering all the necessary needs that this thesis required.

Finally, I would like to thank my family and friends for their support over the years. This thesis is dedicated to them.

Amplianitis Konstantinos

Technical University of Crete

Chania 2023

Contents

Abstract iii

Acknowledgements v

Contents vii

List of Figures xi

List of Tables xiii

List of Abbreviations xv

1 Introduction 1
- 1.1 Motivation and Scientific Contributions . . . 1
- 1.2 Thesis Outline . . . 1

2 Theoretical Background 3
- 2.1 FPGA . . . 3
  - 2.1.1 Overview . . . 3
  - 2.1.2 Partial Reconfiguration . . . 4
- 2.2 Soft-Core Processor . . . 6
- 2.3 RISC-V . . . 6
  - 2.3.1 Overview . . . 6
  - 2.3.2 Micro Architecture . . . 8
  - 2.3.3 I/O Connectivity of the RISC-V Platform . . . 8
- 2.4 Containers . . . 9
  - 2.4.1 Overview . . . 9
  - 2.4.2 LXC . . . 10
- 2.5 Docker . . . 10
  - 2.5.1 Overview . . . 10
  - 2.5.2 Docker Workflow . . . 11
- 2.6 Containers on RISC-V . . . 12

3 Related Work **13**
3.1 FPGA virtualization . . . 13
3.2 Docker Containers on Hardware . . . 14
3.3 RISC-V soft cores . . . 15
3.4 Thesis Approach . . . 16

4 Platforms and Tools **19**
4.1 AMD Virtex 7 Family of FPGAs . . . 19
Virtex 7 VC707 Evaluation Kit . . . 19
4.2 Xillinx Vivado Suite 2022.2 . . . 21
4.3 Vitis HLS v2020.2 . . . 22
4.4 AXI Protocol . . . 22
4.5 OS Distribution . . . 23

5 System Architecture **25**
5.1 Thesis general Workflow . . . 25
5.2 RISC-V Processor . . . 27
5.3 Initial Implementation . . . 27
5.3.1 The IO Block . . . 28
5.3.2 The DDR block . . . 28
5.4 Partial Reconfiguration . . . 29
5.4.1 Reconfigurable Modules . . . 29
5.4.2 Integration of the Reconfigurable Modules to the RISC-V Processor . . . 31
5.5 Operating System (OS) . . . 33
5.5.1 Why not bare metal . . . 33
5.5.2 Debian Distribution Setup . . . 34
5.5.3 Debian Kernel Setup . . . 34
5.5.4 Debian filesystem setup . . . 38
5.5.5 Finalizing the system and booting . . . 39
5.6 Docker . . . 40
5.6.1 Why Docker? . . . 41
5.6.2 Other available tools . . . 41
5.6.3 Docker RISC-V Setup . . . 42
5.6.4 Docker Base Image . . . 43
5.6.5 User Implementation . . . 44

6 Results **47**
6.1 CPU Comparison . . . 47

6.1.1 Assumptions . . . 47
6.1.2 RISC-V Single Core Processor . . . 48
6.1.3 RISC-V Dual Core Processor . . . 50
6.1.4 RISC-V Quad Core Processor . . . 52
6.1.5 RISC-V Octa Core Processor . . . 54
6.1.6 Summary . . . 55
6.1.7 Conclusions . . . 57
6.2 Docker Stats . . . 57
6.3 System Flow Results . . . 58
6.3.1 User Denial Of Service on Partial Region . . . 58
6.3.2 Docker Container Partial Region Execution . . . 59

7 Conclusions and Future Work **63**
7.1 Conclusions . . . 63
7.2 Future Work . . . 64

A Frequently Asked Questions **65**
A.1 How Hardware specific is the Operating System? . . . 65
A.2 Does Docker still support software containers? . . . 65

References **67**

List of Figures

2.1 Partial Reconfiguration 4
2.2 Module-based PR 5

4.1 AMD Virtex VC 707 20
4.2 AMD Virtex VC 707 Block Diagram 21

5.1 System Architecture 25
5.2 RocketChip Processor 27
5.3 IO Block 28
5.4 DDR block 29
5.5 Hardware Modules Offsets Vitis HLS 30
5.6 Block Diagram Hierarchies 31
5.7 Vivado Address Editor 31
5.8 Static Region Hierarchy 32
5.9 Partial Reconfiguration Hierarchy 32
5.10 VC 707 dip-switching mode 40
5.11 Docker service 42
5.12 Docker versioning 42
5.13 Docker Info 43
5.14 Docker Container Production Process 44
5.15 User's Container production flow 45

6.1 Single Core Processor Region on FPGA 48
6.2 Single Core Processor FPGA Utilization 49
6.3 Single Core Processor FPGA Utilization Percentage 49
6.4 Dual Core Processor Region on FPGA 50
6.5 Dual Core Processor FPGA Utilization 51
6.6 Dual Core Processor FPGA Utilization Percentage 51
6.7 Quad Core Processor Region on FPGA 52
6.8 Quad Core Processor FPGA Utilization 53
6.9 Quad Core Processor FPGA Utilization Percentage 53
6.10 Octa Core Processor Region on FPGA 54

6.11 Octa Core Processor FPGA Utilization 55
6.12 Octa Core Processor FPGA Utilization Percentage 55
6.13 Power Consumption Tempate 57
6.14 Docker Image size comparison 58
6.15 resources command help flag 58
6.16 resources pr1 already in use from user 59
6.17 resources pr1 already in use by user 2. 59
6.18 devmem2 usage 59
6.19 User interaction results on adder module 60
6.20 User interaction results on subb module 61

List of Tables

6.1 Utilization Table of all CPU implementations 56
6.2 Power(W) on CPU Implementations 56

List of Abbreviations

ALU	**A**rithmetic **L**ogic **U**nit
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BRAM	**B**lock **R**andom **A**ccess **M**emory
CPU	**C**entral **P**rocessor **U**nit
CS	**C**omputer **S**cience
DDR4	**D**ouble **D**ata **R**ate type texbf4 memory
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DSP	**D**igital **S**ignal **P**rocessor
FF	**F**lip **F**lops
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GDDR6	**G**raphics **D**ouble **D**ata **R**ate type **6** memory
GPU	**G**raphic **P**rocessor **U**nit
HBM	**H**igh **B**andwidth **M**emory
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
HPC	**H**ight **P**erformance **C**omputing
LUT	**L**ook **U**p **T**able
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
PL	**P**rogrammable **L**ogic
PS	**P**rocessing **S**ystem
RAM	**R**andom **A**ccess **M**emory
SDK	**S**oftware **D**evelopment **K**it
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SSE	**S**treaming **S**IMD **E**xtensions
SSD	**S**olid **S**tate **D**rive
TDP	**T**hermal **D**esign **P**ower
URAM	**U**ltra **R**andom **A**ccess **M**emory
USD	**U**nited **S**tates **D**ollar
Tb	**T**otal serial **B**andwidth
GT	**G**igabit **T**transceivers
GMAC	**G**iga **M**ultiply **AC**cumulate Operations

Dedicated to my family and friends…

Chapter 1

Introduction

1.1 Motivation and Scientific Contributions

Cloud applications are becoming more and more popular in this day and age. Hardware application follows the same pattern. With those situation more and more teams are considering the shift through cloud applications in order to test their implementation on environments that are more cost efficient. Docker is a tool that is used in various applications due to its feature to create environments that are universal. Docker supports also the ability to call recursively images in order to construct a final product. The above reasons led to this thesis. A similar process can be applied in hardware applications also. Hardware containers are containers which include a small partition of reconfigurable hardware that can be applied for performance or additional functionality reasons. In that way, teams or even people individually can utilize small regions of reconfigurable hardware to their own interests. When finished, the image can be uploaded into the DockerHub or a similar platform in order for other users or other members of the team to utilize a working implemenation of their work. Similarly, multi-tenancy of servers can be replaced by reconfigurable hardware, like FPGAs, with several regions communicating with several protocols (AXI) in order to implement applications based on the user's needs.

1.2 Thesis Outline

- **Chapter 2 - Theoretical Background:** This chapter provides knowledge that are considered useful for understanding about this thesis points.
- **Chapter 3 - Related Work:** Provides several papers that relate to the thesis work and presents how are they considered relevant to this work.

- **Chapter 4 - Platforms and Tools:** Analyzes the tools that have been used for the implementation of this thesis.
- **Chapter 5 - System's Architecture:** Describes the general flow of the architecture and how each of the components is implemented.
- **Chapter 6 - Results:** Presents final results of the implementation that is described in the **Chapter 5**.
- **Chapter 7 - Conclusions and Related Work:** Proposes ideas about optimizations that can be done in the future as well as ideas to enhance the functionality.

Chapter 2

Theoretical Background

2.1 FPGA

2.1.1 Overview

The generalized and programmable nature of Field Programmable Gate Arrays (FPGAs) has made them a popular choice for the implementation of digital circuits over the last decades[1]. Almost any type of digital circuit or system may be created using field programmable gate arrays (FPGAs), which are prefabricated silicon devices that can be electrically programmed. They provide low risk, low incremental cost and fast prototyping advantages. These characteristics can lead to a competitive advantage in time-to-market situations[2]. Without going through the drawn-out fabrication process necessary for ASIC designs, ideas can be tested or concepts can be confirmed on FPGAs[3] using a variety of tools that exist on the market. Nevertheless, FPGA programming and deploying are still a challenge [4]. Recently, Amazon AWS, Alibaba, and Huawei provide high-performance FPGA cloud services. High-end FPGAs are a costly resource, and with the advent of several FPGA cloud services, they become more accessible as hardware accelerators for real world applications[5].

2.1.2 Partial Reconfiguration

Reconfiguration refers to the ability of programmable devices such as FPGAs to modify custom designs by loading different configuration software[6]. As both their capabilities and sizes as well as demands have increased, FPGAs have been used in many fields, where their reprogrammability provides a distinct advantage over application-specific integrated circuit implementations (ASICs) fixed. This capability allows hardware designs to be upgraded or reused after deployment[7]. Synchronous FPGA devices provide a more sophisticated technology called Partial Reconfiguration (PR). It is the ability to reconfigure specificaly pre-defined areas of an FPGA anytime after its initial configuration. Developers can perform PR either while the design is active and the device is active, a procedure called active partial reconfiguration, or when the device is inactive in shutdown mode, a procedure known as active partial reconfiguration [8].

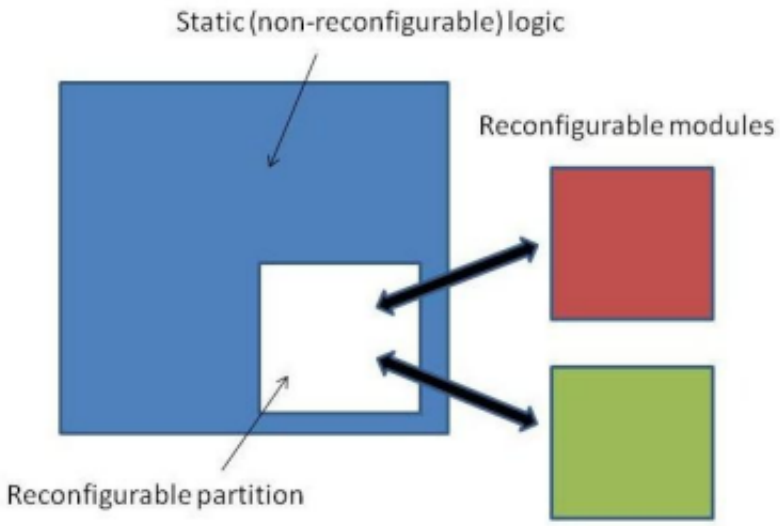

FIGURE 2.1: Partial Reconfiguration.URL

Xilinx has supported partial reconfiguration for many generations of devices. All Xilinx FPGAs support partial reconfiguration, from Virtex-4 devices to the lowest cost FPGAs like the Spartan-3/E family. There are two types of partial reconfiguration supported. Module-based and difference-based.

Module-based Partial Reconfiguration characterizes targeted blocks in the design as reconfigurable modules. Such tactic is generally used by teams as it provides the capability of multiple designers to work on the same design, creating even independent modules and implementing them to the final design. Modules that have dependencies utilize bus macros in order to achieve inter-design communications. Special bus macros allows signals to cross over a partial reconfiguration boundary. Designers should always ensure that any communication that aims to be achieved by reconfigurable signals ought to pass by a bus macro first. In any other case, there is no guarantee about the correctness of the routing in the FPGA. Module-based partial reconfiguration requires implementing a specific set of guidelines during the design specification stage. A separate bitstream is created for each reconfigurable module in your design. Such bitstreams are used to perform partial reconfiguration of FPGAs.[8]

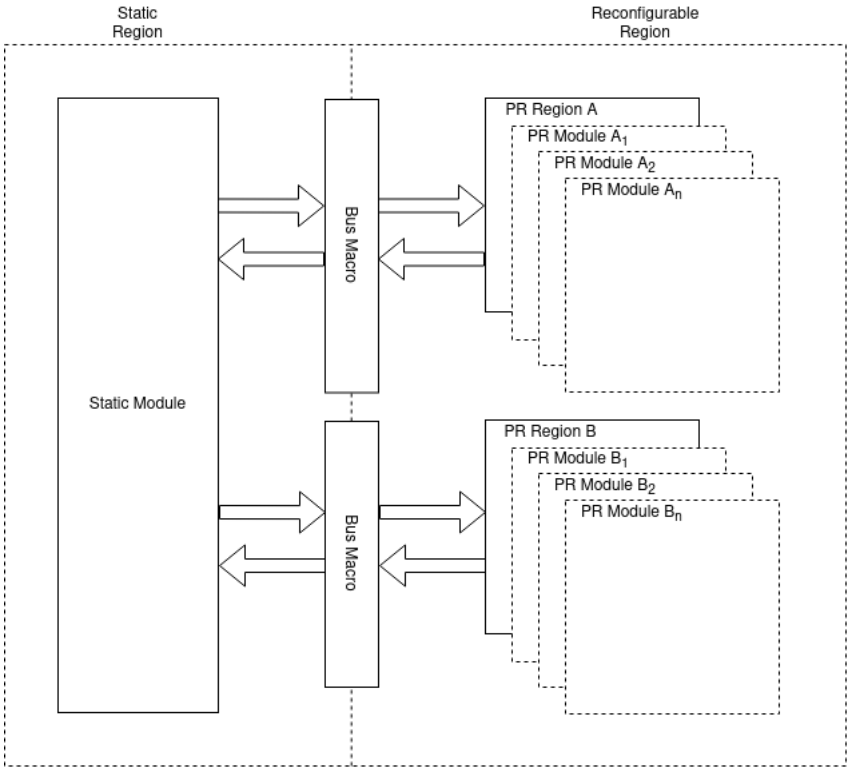

FIGURE 2.2: Module-based PR. URL

Difference-based Partial Reconfiguration is the best fit regarding small changes into the FPGA design. It is especially useful in case of changing Look-Up Table (LUT) equations or dedicated memory blocks content [8]. The bitstream that is getting generated is a product based only in the

differences of the two designs. In general this is a practice that is not suggested for small-scaled designs as it is difficult to identify the correct component and apply the changes.

2.2 Soft-Core Processor

Given that modern designs for embedded systems are becoming more and more complex and demanding due to the progression of the technology and the demands arising. To deal with the situation the idea of using pre-designed and pre-tested intellectual property (IP) cores in designs became an attractive alternative [9]. Soft-core processors refer to a particular type of microprocessor that is designed and implemented by utilizing software[10]. All such processors can be synthesized for Application-Specific Integrated Circuit (ASIC), but the main reason for creating them is to be synthesized for FPGAs. The process of creating such a processor contains a specification of its architecture, instruction set, and functional behavior. All these sections of the process can be implemented via the utilization of high-level hardware description languages (HDLs) like VHDL or Verilog, designers can translate the software specification into a hardware representation that can be synthesized within programmable hardware devices[10].Since the processor's architecture is defined in software, it can be modified, upgraded, or tailored to meet the evolving needs of the application or system it is integrated into[9]. Another positive aspect of soft-core processors is that due to their nature, they are not subject to obsolescence as they are written in an abstraction language (such as HDL ones) and not specifically for a device. All in all, soft-core processors are an alternative that is vastly adopted by both the indistry and the research world as it provides both flexibility and stability in time.

2.3 RISC-V

2.3.1 Overview

RISC-V, an open-source hardware instruction set architecture (ISA), has emerged as a significant player in the field of computer architecture. The RISC-V ISA, based on reduced instruction set computer (RISC) principles,

was first introduced by researchers at the University of California, Berkeley, in 2010. The open-source nature of RISC-V allows for broad-based collaboration and innovation, fostering an environment conducive to rapid advancements. Recent developments in RISC-V have been driven by the increasing demand for more efficient, customizable, and cost-effective solutions in various sectors, including cloud computing, Internet of Things (IoT), and edge computing.

One of the most notable advancements in RISC-V development is the enhancement of its performance and energy efficiency. Researchers and developers have been working tirelessly to optimize the RISC-V core designs, leading to significant improvements in instruction per cycle (IPC) rates and power usage. The advent of high-performance RISC-V cores has paved the way for its adoption in high-performance computing (HPC) applications, a domain traditionally dominated by proprietary ISAs. The open-source nature of RISC-V allows for the customization of the ISA to meet specific application requirements, thereby maximizing performance and energy efficiency.

In the realm of applications, RISC-V has seen a surge in adoption in the IoT and edge computing sectors. The ability to customize the ISA allows for the development of highly specialized IoT devices with optimized power usage and performance characteristics. Furthermore, the open-source nature of RISC-V eliminates licensing costs, making it an attractive option for IoT device manufacturers. In edge computing, RISC-V's scalability and customizability have enabled the development of efficient edge servers capable of processing large amounts of data locally, thereby reducing latency and bandwidth usage.

Looking ahead, the future of RISC-V appears promising, with ongoing research and development efforts aimed at further enhancing its performance, energy efficiency, and customizability. The RISC-V community is also working towards the development of a comprehensive software ecosystem to support the growing number of RISC-V based systems. This includes the development of compilers, debuggers, and operating systems optimized for RISC-V. As the RISC-V ecosystem continues to mature, it is expected to disrupt the computer architecture landscape, offering a viable alternative to proprietary ISAs.

2.3.2 Micro Architecture

The microarchitecture of RISC-V is designed around the principles of simplicity and efficiency, which are inherent to the Reduced Instruction Set Computing (RISC) paradigm. The RISC-V ISA is modular and scalable, comprising a small set of simple and general instructions, known as the base integer ISA, which can be supplemented with optional extensions for more complex operations. This modular design allows for a high degree of flexibility, enabling the development of processors tailored to specific applications. The RISC-V microarchitecture also supports various data widths, including 32-bit, 64-bit, and 128-bit, further enhancing its versatility.

One of the key advantages of the RISC-V microarchitecture is its customizability. The open-source nature of RISC-V allows designers to modify and optimize the microarchitecture to meet specific application requirements. This is particularly beneficial in domains such as embedded systems and IoT, where devices often have unique performance, power, and area constraints. The ability to customize the microarchitecture can lead to significant improvements in system performance and energy efficiency. Another advantage of the RISC-V microarchitecture is its support for hardware-level security features. The RISC-V ISA includes provisions for hardware-enforced memory protection and user-level interrupts, which can enhance system security and reliability. The open-source nature of RISC-V also allows for transparent and collaborative security analysis, potentially leading to more secure systems. Additionally, the RISC-V community is actively working on extensions for cryptographic operations, further bolstering the security capabilities of RISC-V based systems.

2.3.3 I/O Connectivity of the RISC-V Platform

The I/O connectivity of RISC-V systems offers several advantages that stem from the open-source and modular nature of the RISC-V ISA. One of the key benefits is the ability to customize the I/O subsystem to meet specific application requirements. This is particularly relevant in domains such as embedded systems and IoT, where devices often have unique I/O needs. The flexibility of RISC-V allows designers to implement custom I/O interfaces, leading to optimized system performance and efficiency. Furthermore, the RISC-V community has developed several open-source

I/O interface standards, such as the TileLink interconnect standard, which can facilitate the design of complex SoCs. Another advantage of RISC-V in terms of I/O connectivity is its support for advanced I/O virtualization techniques. The RISC-V ISA includes provisions for hardware support for I/O virtualization, which can enhance system performance and security in virtualized environments. This is particularly beneficial in cloud computing applications, where virtualization is a key technology. The open-source nature of RISC-V also allows for transparent and collaborative development of I/O virtualization solutions, potentially leading to more efficient and secure systems. As the RISC-V ecosystem continues to mature, it is expected to further enhance the I/O connectivity capabilities of RISC-V based systems.

2.4 Containers

2.4.1 Overview

Software containers on Linux platforms have revolutionized the way applications are packaged, distributed, and deployed, offering a myriad of advantages. Containers encapsulate an application and its dependencies into a self-contained unit that can run on any Linux system, thereby eliminating the "it works on my machine" problem and facilitating consistent deployments across various environments [11]. This isolation also enhances application security by limiting the potential impact of a compromised application [12]. Containers are lightweight and have a small footprint, as they share the host system's kernel and do not require a full operating system per application like virtual machines. This leads to efficient resource utilization, enabling the deployment of more applications on a given hardware infrastructure [13]. Containers also support microservices architecture, where an application is broken down into small, loosely coupled services. This can enhance application scalability, resilience, and speed up the development process by allowing teams to work on different services independently [14]. Furthermore, containers can be easily integrated into DevOps practices, facilitating continuous integration/continuous deployment (CI/CD) workflows [15]. There are several implementations of software containers available on Linux platforms, including Docker,

which is the most popular and widely used, and Podman, which is a daemonless container engine that aims to provide a Docker-compatible command-line interface while improving on certain aspects, such as security [16].

2.4.2 LXC

Linux Containers (LXC) provide a lightweight virtualization method that runs processes in isolation by leveraging features such as cgroups, namespaces, and chroot in the Linux kernel. LXC containers offer several advantages. They are efficient and portable, allowing applications to be developed more quickly and facilitating easy porting and configuration. LXC containers also provide isolation, which enhances application security, and they are compatible with various Linux distributions, offering flexibility in deployment [17][18].

However, LXC containers also have certain disadvantages. One of the main criticisms is that they are less scalable compared to other container technologies like Docker. Moreover, while LXC images are more lightweight than traditional virtual machines, they are not as lightweight as Docker images [19]. LXC also uses cgroups to limit resource usage, but a particular problem with cgroups is its inability to limit the number of processes that can be created from within a container [20]. Despite these drawbacks, LXC remains a valuable tool for certain use cases where full operating system-level virtualization is required.

2.5 Docker

2.5.1 Overview

Docker is an open platform for developing, shipping, and running applications. It enables its users to separate their applications from the system's infrastructure, allowing them to deliver software quickly. Docker supports the infrastructure management the same ways applications are managed by the host environment. By leveraging Docker's methodologies for shipping, testing, and deploying code quickly, users can significantly reduce the delay between writing code and running it in production [21].

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow

users to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so they do not need to rely on what is currently installed on the host. You can easily share containers while you work, and be sure that everyone you share with gets the same container that works in the same way. Docker also provides tooling and a platform to manage the lifecycle of your containers. You can develop your application and its supporting components using containers. The container becomes the unit for distributing and testing your application. When you're ready, you can deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two [21].

Docker's container-based platform allows for highly portable workloads. Docker containers can run on a developer's local laptop, on physical or virtual machines in a data center, on cloud providers, or in a mixture of environments. Docker's portability and lightweight nature also make it easy to dynamically manage workloads, scaling up or tearing down applications and services as business needs dictate, in near real time. Docker is lightweight and fast. It provides a viable, cost-effective alternative to hypervisor-based virtual machines, so you can use more of your server capacity to achieve your business goals. Docker is perfect for high-density environments and for small and medium deployments where you need to do more with fewer resources [21].

2.5.2 Docker Workflow

The workflow of using Docker containers begins with the development of the application on a local machine, where the developer codes the application using their preferred language and tests it locally. The application is always developed and tested within Docker containers, regardless of the language, framework, and platform chosen. Each Docker container, which is an instance of a Docker image, includes components such as an operating system selection (e.g., a Linux distribution, Windows Nano Server, or Windows Server Core), files added during development (e.g., source code and application binaries), and configuration information (e.g., environment settings and dependencies) [22]

The development workflow for Docker container-based applications is described as an inner-loop workflow, focusing on the development work done on the developer's computer. An application is composed of the developer's own services plus additional libraries (dependencies). The basic steps usually taken when building a Docker application include coding and creating the initial application or service baseline, developing the application similar to the way an application without Docker would be developed, and deploying and testing the application or services running within Docker containers in the local environment [22]

2.6 Containers on RISC-V

Containers are an integral part of the modern software and cloud ecosystem, playing a crucial role in building applications in a reproducible way and defining standards in deployment. The use of containers on RISC-V, a free and open-source instruction set, represents a significant step towards a truly open cloud ecosystem. The combination of containers and RISC-V enables a new level of software and hardware freedom, paving the way for the future of computing design and innovation [23].

The integration of containers with RISC-V brings real openness to the future of the cloud ecosystem by providing a top-to-bottom open solution ranging from the hardware to the end-user software. This combination of technologies is expected to drive the next wave of innovation in the computing industry, offering a new level of extensibility and freedom in architecture [23].

Chapter 3

Related Work

3.1 FPGA virtualization

The increasing integration of Field-Programmable Gate Arrays (FPGAs) into diverse systems, from embedded solutions to expansive cloud infrastructures, underscores their potential as high-performance, energy-efficient accelerators. As these deployments grow in scale, the challenges of efficient application development, resource management, and system scalability become more pronounced. This has spurred interest in FPGA virtualization, aiming to address issues like multi-tenancy execution, multi-FPGA acceleration, and security.

Vaishnav et al.[24] delve into the intricacies of FPGA virtualization, offering a structured overview of the various techniques and hardware infrastructures that have been proposed. Their work categorizes these techniques, highlighting the distinctions between resource, node, and multi-node levels. They also shed light on emerging trends and potential areas for further exploration.

Zheng et al.[25] explore the potential of programmable data planes (PDPs) and introduce P4Visor, a virtualization abstraction tailored for this domain. Their work emphasizes the importance of rapid testing and deployment in the PDP ecosystem, demonstrating the capabilities of P4Visor in supporting multiple PDP programs.

Another innovative approach to FPGA virtualization is the concept of resource elasticity, as presented by Vaishnav et al.[26]. Their methodology allows for dynamic adjustments in accelerator resources, optimizing FPGA utilization and enhancing overall system performance.

Al-Aghbari and Elrabaa[27] propose a platform tailored for cloud environments, focusing on the deployment of FPGA custom computing machines. Their approach leverages FPGA virtualization to enable users to craft independent computing services on standalone FPGAs, emphasizing the platform's high level of abstraction and efficiency.

In the context of integrating FPGA virtualization with modern container technologies, Long et al.[28] offer insights into the convergence of these two domains. The paper explores the potential benefits and challenges of deploying FPGA virtualization within Docker containers, highlighting the synergies and potential applications of this combined approach.

3.2 Docker Containers on Hardware

The integration of Docker containers with hardware platforms has opened up new avenues for optimizing software deployment, scalability, and performance. Docker's lightweight and portable nature, combined with the unique capabilities of hardware platforms like FPGAs, offers a promising solution for a range of applications, from high-performance computing to edge devices.

In the paper by Xiangmeng Long et al.[28], the authors highlight the unique benefits of FPGAs in terms of low latency, energy efficiency, and reconfigurable hardware capabilities. They note that while FPGAs are increasingly being adopted in cloud computing centers and research labs, their direct operation on physical machines often leads to suboptimal resource utilization. To address this, the authors introduce a novel approach that leverages Docker and Kubernetes to virtualize FPGA resources. By encapsulating FPGA resources within Docker containers and orchestrating them with Kubernetes, they demonstrate enhanced FPGA resource sharing and utilization across multiple applications.

Preeth and colleauges[29] delve into the evaluation of Docker containers, focusing on their performance based on system resource utilization. They emphasize Docker's ability to streamline the development, deployment, and scaling of applications, irrespective of the programming language. Through a series of benchmark tests, the authors assess the performance of Docker containers in terms of file system operations, CPU utilization,

memory usage, and other system metrics. Their findings provide a comprehensive overview of Docker's efficiency and potential bottlenecks, offering valuable insights for developers and system administrators.

Lastly, a study by Lallet et al.[30] explores the potential of FPGA-based virtualization in the context of Next Generation Platform-as-a-Service (NG-PaaS). The paper delves into the challenges and opportunities of integrating FPGA virtualization in distributed systems, offering insights into potential optimizations and best practices.

3.3 RISC-V soft cores

The RISC-V instruction set architecture (ISA) has emerged as a prominent open standard for processor architectures, offering flexibility, modularity, and extensibility. Soft-cores based on the RISC-V ISA are particularly appealing for their adaptability to various application domains and their compatibility with a wide range of hardware platforms. The open-source nature of RISC-V has also fostered a vibrant ecosystem of tools, methodologies, and research initiatives aimed at optimizing and verifying RISC-V soft-cores.

A notable contribution in the realm of RISC-V soft-cores is the "GRVI Phalanx: A Massively Parallel RISC-V FPGA Accelerator" by J. Gray[31]. This work introduces the GRVI, an FPGA-efficient RISC-V RV32I soft processor, and Phalanx, a parallel processor and accelerator array framework. The research emphasizes the potential of combining multiple RISC-V cores to achieve high throughput and bandwidth, showcasing the scalability and efficiency of the proposed architecture.

Schuiki et al.[32] present an innovative approach to enhancing the compute utilization of single-issue cores through the introduction of "Stream Semantic Registers" (SSR). The SSR is a lightweight RISC-V ISA extension that implicitly encodes memory accesses, reducing the need for explicit loads/stores. The authors demonstrate the architectural speedup achieved through SSR, highlighting its potential in boosting energy efficiency and performance in multi-core clusters.

In the context of IoT processors, Bruschi et al.[33] introduce GVSoC, a highly configurable and timing-accurate event-driven simulator tailored for RISC-V based IoT processors. GVSoC combines the efficiency of C++ models with Python configuration scripts, offering a versatile platform for functional and performance analysis. The simulator aims to drive future research in the area of highly parallel and heterogeneous RISC-V based IoT processors.

Another significant contribution to the RISC-V ecosystem is Rocket Chip, an open-source digital design tailored for teaching computer architecture. Developed by a team led by Krste Asanović[34], the Rocket Chip generator produces a range of customizable open-source processors and SoC designs. These designs are not only competitive with proprietary counterparts but also serve as invaluable resources for architecture research, academic courses, and commercial RISC-V designs. The modularity of the generator allows for the creation of designs with diverse capabilities, performance metrics, and implementation technologies. Written in the Chisel hardware construction language, the Rocket Chip generator exemplifies the power of circuit generators that can produce multiple RTL implementations from a singular parameterized design.

3.4 Thesis Approach

The rapid evolution of computing paradigms has ushered in an era where traditional CPU-centric architectures are increasingly being complemented by specialized hardware accelerators. Among these, Field Programmable Gate Arrays (FPGAs) stand out for their unique blend of reconfigurability, parallelism, and energy efficiency. FPGAs offer the flexibility to design custom-tailored hardware solutions, enabling the optimization of specific computational tasks, thereby achieving performance levels that are often challenging for general-purpose processors. By harnessing the inherent parallelism and adaptability of FPGAs, this thesis aims to push the boundaries of computational efficiency and adaptability.

Building upon the foundational advantages of FPGAs, the integration of Docker container technology emerges as a transformative approach. Docker, renowned for its lightweight virtualization and portability, when deployed on FPGAs, can encapsulate hardware-accelerated applications, ensuring

consistent and scalable performance across diverse environments. By marrying the reconfigurability of FPGAs with the abstraction provided by Docker, this thesis explores a novel paradigm where hardware-accelerated applications can be rapidly developed, deployed, and scaled, all while maintaining the benefits of hardware-level optimization.

RocketChip, a prominent RISC-V soft-core, further enriches the thesis's approach. As an open-source and highly customizable processor design, RocketChip offers a modular foundation upon which specialized computational units can be integrated. By leveraging the RocketChip's extensibility and combining it with FPGA-based acceleration and Docker's deployment advantages, this thesis charts a path towards a holistic computing platform. This platform not only bridges the gap between software and hardware but also paves the way for a new generation of efficient, scalable, and adaptable computing solutions.

Chapter 4

Platforms and Tools

4.1 AMD Virtex 7 Family of FPGAs

The Virtex family of Field Programmable Gate Arrays (FPGAs) developed by Xilinx, a part of AMD, is optimized for system performance and integration at 28nm, offering exceptional performance per watt, DSP performance, and I/O bandwidth to designs [1]. The Virtex family is used in a wide array of applications such as 10G to 100G networking, portable radar, and ASIC Prototyping. The Virtex 7 FPGAs, in particular, are designed to bring high performance and integration at 28nm, providing up to 2M logic cells, VCXO component, AXI IP, and AMS integration [35].

The Virtex 7 FPGAs offer increased system performance with up to 2.8 Tb/s total serial bandwidth with up to 96 x 13.1G GTs, up to 16 x 28.05G GTs, 5,335 GMACs, 68Mb BRAM, DDR3-1866. They also provide a significant reduction in the Bill of Materials (BOM) cost, up to 40 percent lower than a multi-chip solution, and total power reduction, up to 50 percent lower power than a multi-chip solution. The Virtex family of FPGAs also offers accelerated design productivity with a scalable optimized architecture, comprehensive tools, IP, and Targeted Design Platforms (TDPs) [35].

Virtex 7 VC707 Evaluation Kit

The Virtex 7 FPGA VC707 Evaluation Kit, developed by Xilinx, a part of AMD, is a full-featured, highly-flexible, high-speed serial base platform using the Virtex 7 XC7VX485T-2FFG1761C. The kit includes basic components of hardware, design tools, IP, and pre-verified reference designs for system designs that demand high-performance, serial connectivity, and advanced memory interfacing [36]. The VC707 Evaluation Kit is designed to

speed up development and reduce time to market for customers creating high-performance applications in a variety of markets, including communications, storage, server, video, and signal processing.

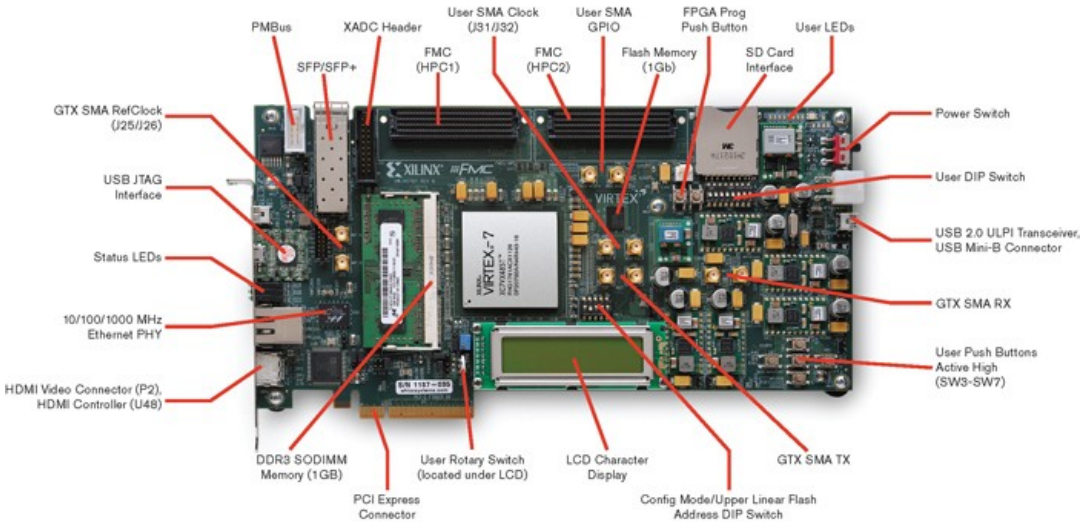

FIGURE 4.1: AMD Virtex VC 707.URL

The VC707 Evaluation Kit features a range of components that make it a versatile platform for the development and testing of FPGA designs. It includes a 1GB DDR3 SODIMM running at 800MHz / 1600Mbps, 128MB Linear BPI Flash for PCIe Configuration, and an SD Card Slot. For communication and networking, the kit offers Gigabit Ethernet GMII, RGMII and SGMII, an SFP+ transceiver connector, GTX port (TX, RX) with four SMA connectors, UART To USB Bridge, and a PCI Express x8 gen2 Edge Connector. The kit also includes a range of control and I/O features, including push buttons, DIP switches, a rotary encoder switch, and an AMS FAN Header. For display, the kit offers HDMI Video OUT, a 2 x16 LCD display, and LEDs [36].

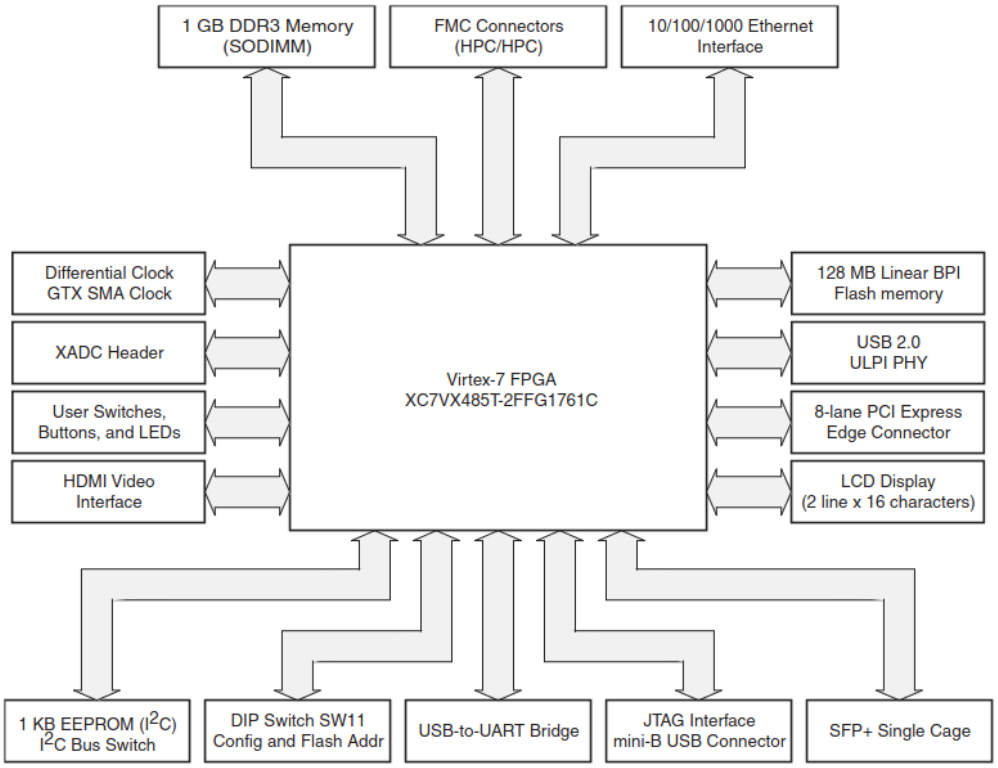

FIGURE 4.2: AMD Virtex VC 707 Block Diagram. URL

4.2 Xillinx Vivado Suite 2022.2

The Xilinx Vivado Design Suite is a revolutionary software tool for designing with Xilinx FPGAs, SoCs, and 3D ICs. It is a system-centric design environment that enables the use of C-based IP in a platform-centric workflow, creating a highly productive, ultra-high-level synthesis methodology. The Vivado suite includes a new approach to design with its IP integrator, which allows for a graphical and Tcl-based correct-by-construction design development flow. Working at the interface level, design teams can rapidly assemble complex systems that leverage IP created with the Vitis HLS tool, Vitis Model Composer, AMD IP, and Alliance Member IP, as well as your own IP. By leveraging the combination of the newly improved Vivado IPI and HLS tools, customers are saving up to 15X in development costs versus an RTL approach [37].

The Vivado suite also includes advanced machine learning algorithms that deliver the best implementation tools with significant advantages in runtime and performance. With best-in-class compilation tools for synthesis, place, route, and physical optimization, as well as AMD compiled

methodology recommendations, designers can accelerate the implementation phase of their design cycle. The Vivado ML Edition, with its advanced machine learning algorithms, delivers quality of results (QoR) improvements of up to 50 percent (average 10 percent) on complex designs, compared to the Vivado HLx Edition. New features and algorithms like ML-based logic optimization, congestion estimation, delay estimation, and intelligent design runs help automate strategies to reduce timing closure iterations [37].

4.3 Vitis HLS v2020.2

The Xilinx Vitis High-Level Synthesis (HLS) tool is a powerful software suite that enables developers to create complex FPGA algorithms by synthesizing C/C++ functions into Register Transfer Level (RTL). This tool is tightly integrated with both the Vivado Design Suite for synthesis and place & route, and the Vitis unified software platform for heterogeneous system designs and applications. By using the Vitis HLS flow, users can apply directives to the C code to create the RTL specific to a desired implementation. This allows for the creation of multiple design architectures from the C source code and enables a path for high-quality, correct-by-construction RTL. The Vitis HLS tool also features a rich set of analysis and debugging tools that facilitate design optimization [38].

The Vitis HLS tool is designed to take advantage of the benefits and characteristics offered by the architecture of AMD FPGAs. It supports parallel programming constructs in order to model a desired implementation. These constructs include HLS tasks that allow process-level concurrency, HLS vectors that allow data-level parallelism, and HLS streams that allow communication between concurrent tasks. Synthesis pragmas can be used to control the results. These pragmas include pipeline, unroll, array partitioning, and interface protocols. The output of the Vitis HLS tool is an RTL implementation that can be either packaged into a compiled object file (.xo) or exported to an RTL IP [38].

4.4 AXI Protocol

The Advanced Microcontroller Bus Architecture (AMBA) AXI (Advanced eXtensible Interface) protocol is a key component of the Xilinx Vivado and

Vitis platforms. The AXI4 interfaces supported by Vitis HLS include the AXI4-Stream interface, AXI4-Lite, and AXI4 master interfaces. These interfaces implement an adapter to manage communication according to the protocol, providing a robust infrastructure for connecting hardware accelerators to embedded CPUs. The AXI4 interfaces are the default interfaces used by Vitis HLS for the Vitis Application Acceleration Development flow, though they are also supported in the Vivado IP flow [39].

The AXI4 Memory Mapped interface can be specified on arrays and pointers, allowing for efficient memory-mapped transfers. The AXI4-Lite slave interface can be specified on any type of argument except streams, providing a lightweight interface for control register access. The AXI4-Stream interface can be specified on input or output arguments, facilitating high-speed data streaming [39].

The AXI Interconnect IP, included with the Vivado and ISE Design Suite at no additional charge, connects one or more AXI memory-mapped Master devices to one or more memory-mapped Slave devices. The AXI interfaces conform to the AMBA AXI version 4 specifications from ARM, including the AXI4-Lite control register interface subset. The Interconnect IP is intended for memory-mapped transfers only; AXI4-Stream transfers are not applicable. The AXI Interconnect IP can be used from the Vivado IP catalog as a pcore from the Embedded Development ToolKit (EDK) or as a standalone core from the CORE Generator IP catalog.

The AXI protocol on the Xilinx Vivado/Vitis platforms offers several advantages, including efficient memory-mapped transfers, lightweight control register access, and high-speed data streaming. However, it also has some limitations. For instance, the AXI4-Lite interface cannot be used with streams, and the AXI Interconnect IP is not applicable for AXI4-Stream transfers [39].

4.5 OS Distribution

Debian, often referred to as the "universal operating system," is not just limited to desktop computers and servers but also supports an impressive array of platforms, including embedded systems, single-board computers, and even mainframes. This versatility has enabled Debian to be used in a

wide range of applications, from powering small Internet of Things (IoT) devices to running critical infrastructure. The Debian community plays a vital role in the OS's continuous development. Thousands of developers and contributors from around the world work together in a decentralized manner, adhering to Debian's strict social contract and Debian Free Software Guidelines. This ensures that the software included in Debian is entirely free, empowering users with complete control over their systems and data. One of the most significant strengths of Debian is its package management system, APT, which simplifies software installation and updates. The vast Debian package repository offers an extensive selection of software, covering virtually every use case, from productivity tools and multimedia applications to development frameworks and server software. This broad selection allows users to customize their Debian installations according to their specific needs, fostering a rich and diverse user experience. Debian's stable release cycle focuses on reliability and security. The release process is thorough, involving meticulous testing to ensure that each new version meets the high standards of stability and performance that Debian is renowned for. The stable release is an excellent choice for servers and production environments, as it provides long-term support and receives regular security updates.For users seeking more up-to-date software and the latest features, Debian offers the "testing" and "unstable" branches, catering to a more adventurous audience. These branches are continuously evolving and act as a testing ground for future stable releases. However, they may be less suited for mission-critical systems due to their ever-changing nature.In conclusion, Debian's reputation as a robust, secure, and community-driven operating system has solidified its position as one of the most influential and widely used Linux distributions in the world. Its commitment to free software, extensive hardware support, and versatile package management system make it an excellent choice for a diverse range of users and use cases.

Chapter 5

System Architecture

5.1 Thesis general Workflow

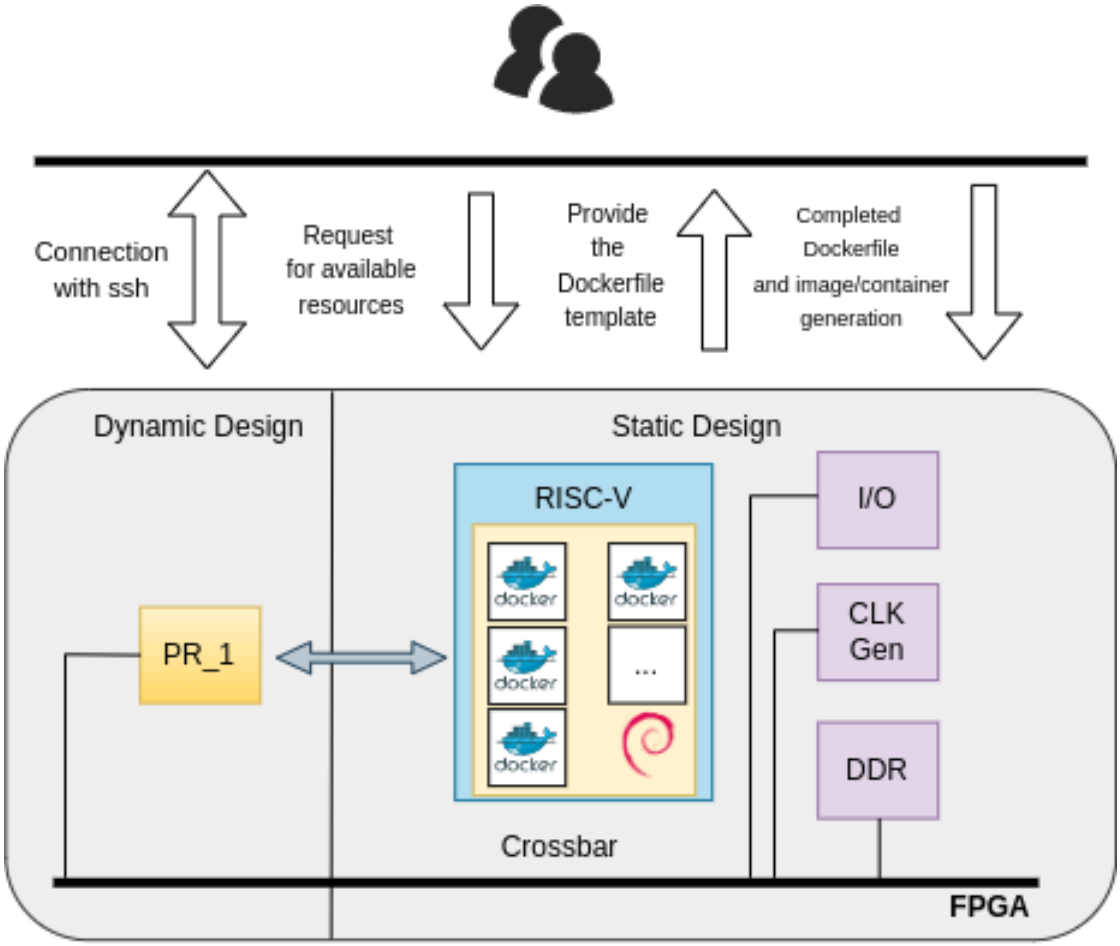

FIGURE 5.1: System Architecture.

The designed system is a user-interactive one, which allows each user to connect to the system's dedicated operating system via the SSH protocol. Once logged into their respective accounts, users can check the available

resources and regions. Based on this information, they can choose the most suitable option for their implementation.

The process involves the following steps:
***User Authentication*: Each user logs in with their credentials using SSH (Secure Shell) to access the system's dedicated operating system securely.**
***Resource Check*: After logging in, the system presents the user with information about the available resources and regions. This could include hardware specifications, computing capacity, memory availability, and other relevant details.**
***Selection of Region*: With the available information, users can make an informed decision about the region that best suits their implementation requirements. This might involve considering factors like performance, availability, and location.**
***Template Provision*: Once the user selects a specific region, the system provides them with a template. This template is a starting point that outlines the basic structure of the implementation. It includes placeholders and guidelines for the user to enrich it further.**
***Enrichment of Dockerfile Template*: Users are then required to enrich the provided template with the relevant information or bitstream that they want to implement as a component in the basic RISC-V implementation. This step involves customizing the template to suit their specific needs.**
***Image and Container Creation*: After the user enriches the template, the system automatically creates an image and a container. The image contains the user's implementation with all the specified details, while the container provides a controlled environment for testing and validation.**
***Executing Implementation*: The user can then use the created image and container to test their implementation thoroughly. This testing phase helps ensure that the newly integrated component works seamlessly with the basic RISC-V implementation and meets the desired objectives.**

By following this user-centric approach, the system enables users to easily implement and test their own custom components within the RISC-V architecture. This fosters innovation and allows for a broad range of experiments and developments within the system's framework.

5.2 RISC-V Processor

The implementation RISC-V processor of system is an important part. After enough research for possible implementations that can be utilized to fill that part of the system, the best fit found was that one of *vivado-risc-v by eugene tarasov [40].*

The processor selected is a Rocker Chip implementation. It is written in the Chisel hardware construction language, which is used to describe hardware circuits at a higher level of abstraction compared to traditional hardware description languages like Verilog or VHDL. This makes it easier to develop, modify, and understand the processor's implementation. Since the Rocket Chip is open-source, it encourages collaborative development and allows researchers, hobbyists, and companies to experiment with and improve the design, making it a valuable tool in the RISC-V ecosystem. Rocket Chip's primary goal is to provide a simple, but complete, RISC-V processor core that can be easily extended and tailored for specific use cases. For that reason, it is considered to be a great fit for our implementation.[41]

5.3 Initial Implementation

The block diagram of the project can be seen in the figure 5.2

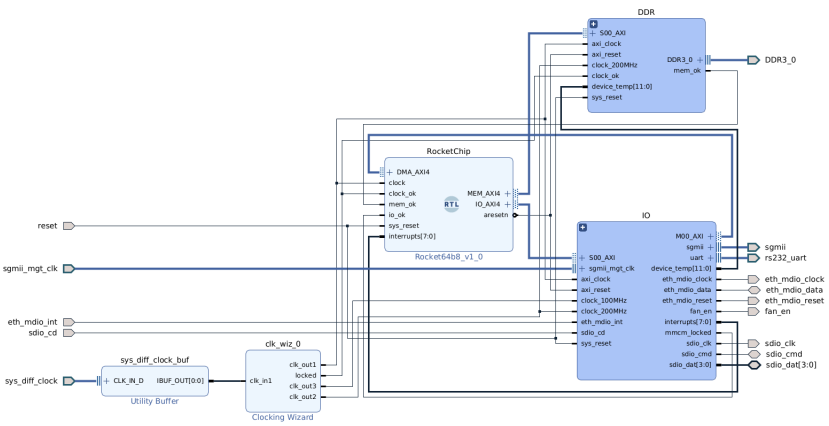

FIGURE 5.2: RocketChip Processor

The design consists of 3 basic elements. The Rocket Core which is generated by the Rocket Chip generator, the IO block for the communication with various peripherals and the DDR block for the interconnection with the DDR ram.

5.3.1 The IO Block

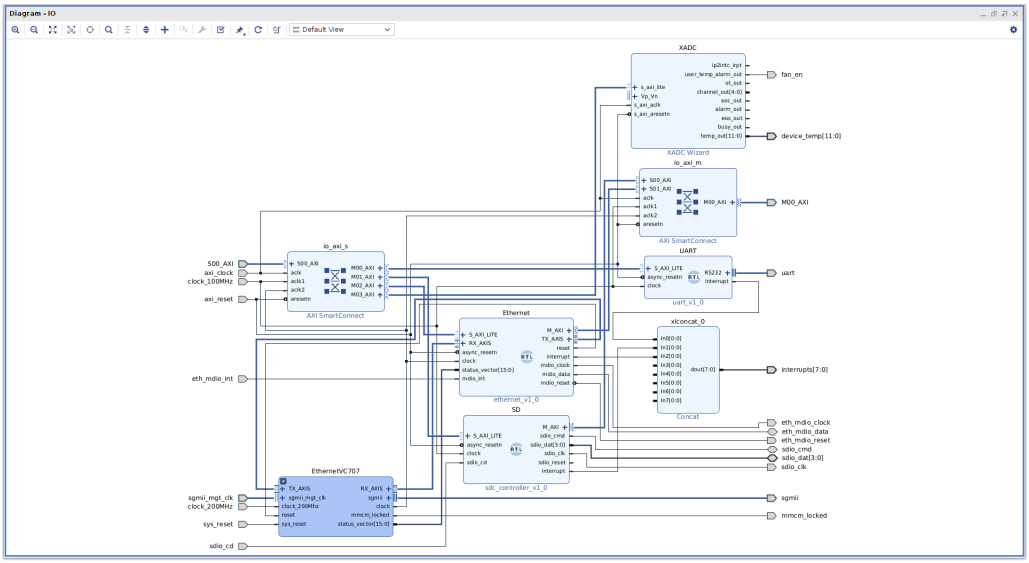

FIGURE 5.3: IO Block

The IO Block consists of various blocks about serial communication, ethernet connection, SD controller and others like AXI SmartConnect blocks. Initally it contains two modules of AXI SmartConnect in order the processor to communicate with any other IO module such as a custom adder. Furthermore it consists of a UART module for the serial communication between outsie world and the processor using AXI lite protocol. In addition it contains multiple blocks about ethernet modues in order to achieve ethernet communcation between the outside world and the processor using AXI stream protocol.

5.3.2 The DDR block

The DDR block contains three blocks for the purpose of the communication between the processor and the solid DDR memory of the FPGA. The most fundamental block is the Memory Interface Generator (MIG 7 Series)

which functions as interface with the Solid DDR memory. In addition to this, it contains a memory reset system for reseting the DDR memory as well an AXI SmartConnect module for the communication of the processor with the DDR memory. The memory provided by the RocketChip generator is 1GB. This amount of memory is considered to be enough for the Operating System that will be explained analytically in the following chapters of this thesis.

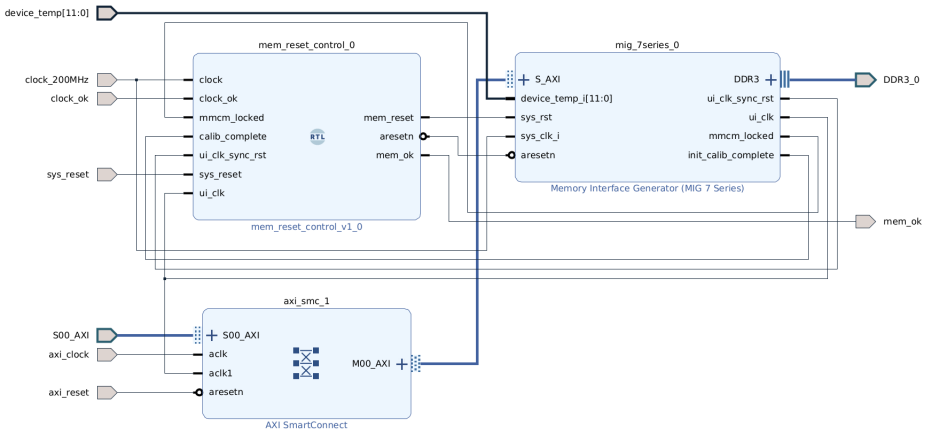

FIGURE 5.4: DDR block

The Rocket Core can be generated with multiple cores (cores=1-8). In order to select the best fit for the thesis requirements. Important note was given in Vivado measurements about LUTs, BRAM, flip-flops and DSPs.

5.4 Partial Reconfiguration

After running the initial implementation successfully, the changes regarding the partial reconfiguration have to be implemented. For the thesis purposes, the partially reconfigurable region is only one. The reconfigurable modules designed are one addition and one subtraction modules, as it is considered that these modules achieve the POC approach.

5.4.1 Reconfigurable Modules

Both the modules referred above were designed using the Vitis_HLS tool of Xilinx. The code for both modules is almost identical as the process they

do is very similar. The communication protocol that the modules operate under is AXI LITE. After synthesizing the modules, it is important to look for the offsets that have been generated by the tool. This can be found in the .h file in the impl/ip/drivers of each generated module.

```
// ==========================================================
// Vitis HLS - High-Level Synthesis from C, C++ and OpenCL v2020.2 (64-bit)
// Copyright 1986-2020 Xilinx, Inc. All Rights Reserved.
// ==========================================================
// control
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/SC)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - enable ap_done interrupt (Read/Write)
//        others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - ap_done (COR/TOW)
//        others - reserved
// 0x10 : Data signal of ap_return
//        bit 31~0 - ap_return[31:0] (Read)
// 0x18 : Data signal of A
//        bit 31~0 - A[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of B
//        bit 31~0 - B[31:0] (Read/Write)
// 0x24 : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

#define XADDER_RISC_CONTROL_ADDR_AP_CTRL   0x00
#define XADDER_RISC_CONTROL_ADDR_GIE       0x04
#define XADDER_RISC_CONTROL_ADDR_IER       0x08
#define XADDER_RISC_CONTROL_ADDR_ISR       0x0c
#define XADDER_RISC_CONTROL_ADDR_AP_RETURN 0x10
#define XADDER_RISC_CONTROL_BITS_AP_RETURN 32
#define XADDER_RISC_CONTROL_ADDR_A_DATA    0x18
#define XADDER_RISC_CONTROL_BITS_A_DATA    32
#define XADDER_RISC_CONTROL_ADDR_B_DATA    0x20
#define XADDER_RISC_CONTROL_BITS_B_DATA    32
```

FIGURE 5.5: Hardware Modules Offsets Vitis HLS.

The important addresses to keep are the 0x18, 0x20 which are the addresses where that data inputs have to be given. The return address of the implementation can be seen to be the 0x10. Finally, it is important to remember the AP_CTRL address as it is the address that will enable the module. All the other addresses concern interrupt protocols so they are not in the scope of this thesis.

5.4.2 Integration of the Reconfigurable Modules to the RISC-V Processor

The integration of the implemented modules that described previously, was made according to the Vivado Design Suite Tutorial: Dynamic Function eXchange (UG947). As can be seen in the figure below are two hierarchies. The static_region one and the rp1.

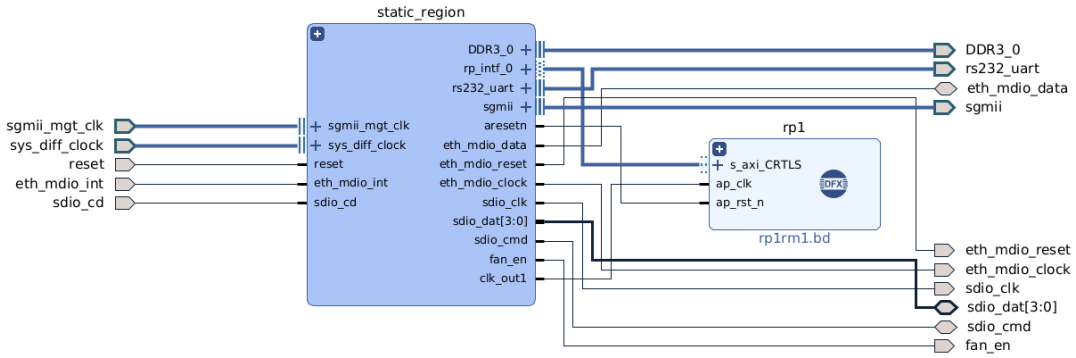

FIGURE 5.6: Block Diagram Hierarchies.

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/static_region/IO/Ethernet					
/static_region/IO/Ethernet/M_AXI (34 address bits : 16G)					
/static_region/RocketChip/DMA_AXI4	DMA_AXI4	reg0	0x0_0000_0000	16G	0x3_FFFF_FFFF
/static_region/IO/SD					
/static_region/IO/SD/M_AXI (34 address bits : 16G)					
/static_region/RocketChip/DMA_AXI4	DMA_AXI4	reg0	0x0_0000_0000	16G	0x3_FFFF_FFFF
Network 1					
/static_region/RocketChip					
/static_region/RocketChip/IO_AXI4 (31 address bits : 2G)					
/rp1/add_0/s_axi_CRTLS	s_axi_CRTLS	Reg	0x6004_0000	64K	0x6004_FFFF
/static_region/dfx_decoupler_0/s_axi_reg	s_axi_reg	Reg	0x44A0_0000	64K	0x44A0_FFFF
/static_region/IO/Ethernet/S_AXI_LITE	S_AXI_LITE	reg0	0x6002_0000	64K	0x6002_FFFF
/static_region/IO/SD/S_AXI_LITE	S_AXI_LITE	reg0	0x6000_0000	64K	0x6000_FFFF
/static_region/IO/UART/S_AXI_LITE	S_AXI_LITE	reg0	0x6001_0000	64K	0x6001_FFFF
/static_region/IO/XADC/s_axi_lite	s_axi_lite	Reg	0x6003_0000	64K	0x6003_FFFF
Network 2					
/static_region/RocketChip					
/static_region/RocketChip/MEM_AXI4 (34 address bits : 16G)					
/static_region/DDR/mig_7series_0/memmap	S_AXI	memaddr	0x0_0000_0000	16G	0x3_FFFF_FFFF

FIGURE 5.7: Component Address Editor.

The static region contains the RISC-V processor, the memory and the I/O block. A DFX Decoupler was also added into the static region. The decoupler serves as a middle layer between the static and the partially reconfigurative region. It ensures the safe interface between the two regions while

the dynamic renconfiguration is occuring. As one decoupler can support multiple interfaces including AXI4-Lite and AXI4-Stream, it is considered to be the best fit for this thesis. In a future implementation that more than one region will be supported, multiple decouplers can satisfy such requirements.

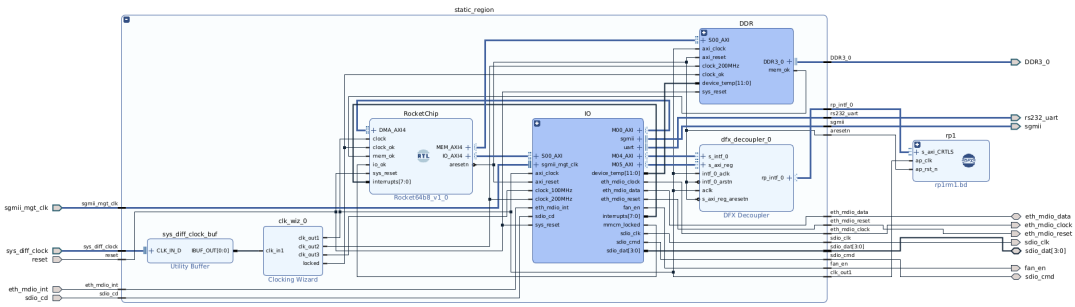

FIGURE 5.8: Static Region Hierarchy.

The pr1, or partial region, contains the module that will be placed in the region configured to be the partial reconfiguration one. Due to the vivado workflow, in order to produce the correct bitstreams, equal amount of hierarchies must be made. As mentioned in the previous sub-sections of this chapter, the modules that are used in the partial region are two. So sequentially, the hierarchies that had to be created were also two. Each hierarchy of elements was marked as one for partial reconfiguration.

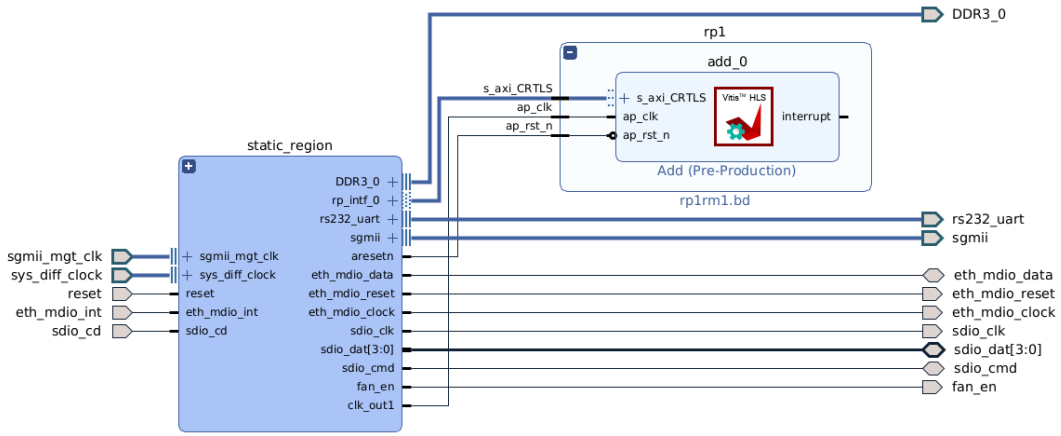

FIGURE 5.9: Partial Reconfiguration Hierarchy.

After finishing creating the constraints file that are auto-generated by the Xilinx Vivado tool, the generate bitstream option provides four different bitstreams. Two of them contain both the static region and a reconfigurable module. These can be used as "initial" bitstreams, meaning they are produced to bring the system into an initial state. The other two bitsreams contain only the modules. These two are useful only for the partial reconfiguration. In general a user needs also the static design to create their individual implementation and be able to apply it correctly.

5.5 Operating System (OS)

The RISC-V architecture has emerged as a promising and versatile platform for various computing applications, giving rise to a diverse range of operating systems (OS) tailored to its specifications. From the traditional Linux-based distributions, such as Debian RISC-V and Fedora RISC-V, to light-weight and embedded-focused options such as FreeRTOS and Zephyr, the availability of OS for RISC-V demonstrates its adaptability across different domains. In addition to Linux-based distributions, RISC-V supports bare-metal programming, allowing developers to directly control hardware resources without relying on an underlying OS. Additionally, real-time operating systems (RTOS) tailored for RISC-V provide deterministic and low-latency execution for time-critical applications, like industrial control systems and robotics. The challenges in OS development highlight the need for continuous research and innovation to fully harness the potential of the RISC-V architecture in shaping the future of computing.

5.5.1 Why not bare metal

Due to the fact that the thesis implementation demands a tool like Docker, a bare metal implementation approach is not considered to be the correct one. This point is justified by the fact that Docker on bare-metal does not provide the same level of isolation. The lack of this feature is a blocking factor for such an implementation, as it is designed to support more than one user at the time.

5.5.2 Debian Distribution Setup

From the plethora of Operating Systems the can work on RISC-V platforms, the Debian distribution was selected for familiarity reasons. In order to create an Operating System capable of hosting a tool like Docker, the Kernel had to be created accordingly. In other words, there is a need for a kernel that provides all the kernel modules required both for Docker and its respective tools. The Operating system that has been created provides the ability to have a full implementation of Docker, not only the parts needed to serve the purposes of this thesis. More specifically, Docker can fully support software containers, networks etc.
The creation of the Debian OS is divided in two steps. The Debian Kernel setup and the Debian filesystem setup. Both the steps will be described thoroughly in the sections below.

5.5.3 Debian Kernel Setup

The kernel is a fundamental component of an operating system. It serves as the bridge between the hardware and software, allowing them to communicate and interact with each other. Its primary role is to manage the system's resources and provide essential services to applications and user processes. In these days the kernel building process is much simpler than before but yet remains a challenge. The first step that needs to be made is to decide what are the needed kernel modules for Docker. In order to to do that, a bash-script [42] that is approved by Docker was used. The script contains both the required modules that need to be installed and optional ones. All the modules are listed below:

```
Generally Necessary:
    - cgroup hierarchy
    - CONFIG_NAMESPACES
    - CONFIG_NET_NS
    - CONFIG_PID_NS
    - CONFIG_IPC_NS
    - CONFIG_UTS_NS
    - CONFIG_CGROUPS
    - CONFIG_CGROUP_CPUACCT
    - CONFIG_CGROUP_DEVICE
    - CONFIG_CGROUP_FREEZER
    - CONFIG_CGROUP_SCHED
```

- CONFIG_CPUSETS
- CONFIG_MEMCG
- CONFIG_KEYS
- CONFIG_VETH
- CONFIG_BRIDGE
- CONFIG_BRIDGE_NETFILTER
- CONFIG_NF_NAT_IPV4
- CONFIG_IP_NF_FILTER
- CONFIG_IP_NF_TARGET_MASQUERADE
- CONFIG_NETFILTER_XT_MATCH_ADDRTYPE
- CONFIG_NETFILTER_XT_MATCH_CONNTRACK
- CONFIG_NETFILTER_XT_MATCH_IPVS
- CONFIG_IP_NF_NAT
- CONFIG_NF_NAT
- CONFIG_NF_NAT_NEEDED
- CONFIG_POSIX_MQUEUE

Optional Features:

- CONFIG_USER_NS
- CONFIG_SECCOMP
- CONFIG_CGROUP_PIDS
- CONFIG_MEMCG_SWAP
- CONFIG_MEMCG_SWAP_ENABLED
- CONFIG_BLK_CGROUP
- CONFIG_BLK_DEV_THROTTLING
- CONFIG_IOSCHED_CFQ
- CONFIG_CFQ_GROUP_IOSCHED
- CONFIG_CGROUP_PERF
- CONFIG_CGROUP_HUGETLB
- CONFIG_NET_CLS_CGROUP
- CONFIG_CGROUP_NET_PRIO
- CONFIG_CFS_BANDWIDTH
- CONFIG_FAIR_GROUP_SCHED
- CONFIG_RT_GROUP_SCHED
- CONFIG_IP_NF_TARGET_REDIRECT
- CONFIG_IP_VS
- CONFIG_IP_VS_NFCT
- CONFIG_IP_VS_PROTO_TCP

```
    - CONFIG_IP_VS_PROTO_UDP
    - CONFIG_IP_VS_RR
    - CONFIG_EXT4_FS
    - CONFIG_EXT4_FS_POSIX_ACL
    - CONFIG_EXT4_FS_SECURITY
- Network Drivers:
  - "overlay":
    - CONFIG_VXLAN
      Optional (for encrypted networks):
      - CONFIG_CRYPTO
      - CONFIG_CRYPTO_AEAD
      - CONFIG_CRYPTO_GCM
      - CONFIG_CRYPTO_SEQIV
      - CONFIG_CRYPTO_GHASH
      - CONFIG_XFRM
      - CONFIG_XFRM_USER
      - CONFIG_XFRM_ALGO
      - CONFIG_INET_ESP
      - CONFIG_INET_XFRM_MODE_TRANSPORT
  - "ipvlan":
    - CONFIG_IPVLAN
  - "macvlan":
    - CONFIG_MACVLAN
    - CONFIG_DUMMY
  - "ftp,tftp client in container":
    - CONFIG_NF_NAT_FTP
    - CONFIG_NF_CONNTRACK_FTP
    - CONFIG_NF_NAT_TFTP
    - CONFIG_NF_CONNTRACK_TFTP
- Storage Drivers:
  - "aufs":
    - CONFIG_AUFS_FS
  - "btrfs":
    - CONFIG_BTRFS_FS
    - CONFIG_BTRFS_FS_POSIX_ACL
  - "devicemapper":
    - CONFIG_BLK_DEV_DM
    - CONFIG_DM_THIN_PROVISIONING
```

```
  - "overlay":
    - CONFIG_OVERLAY_FS
  - "zfs":
    - /dev/zfs
    - zfs command
    - zpool command
Limits:
    - /proc/sys/kernel/keys/root_maxkeys
```

In order to create a Kernel that fulfills the above requirements, a specific procedure was followed. This procedure is described below.

First thing that needs to be done is to install all the necessary libraries for RISC-V in order to complete the cross-compilation process. Qt5 libraries are also vital for the xconfig to work.
In order to create the kernel easier, the provided Makefile from the *vivado-risc-v of eugene tarasov*[40] was used. In the repository folder in the linux-stable directory the following lines of commands were executed:

```
sudo make ARCH=riscv mrproper
sudo make ARCH=riscv xconfig
```

The first command will clear the previously auto-generated files in order to avoid possible conflicts. If it is the first time creating the kernel this command will not make any changes but it is considered overall a good practice. The second command will open an graphical environment displaying a list of all the kernel modules. When checking all the modules needed it is preferable for them to be added as a built in module and not as extrenals. The reason behind that is external modules should be reloaded manually each time the system is initializing.

After completing the process, a linux.config file should be created and saved in the proper directory in order for the kernel generator to work properly. The path is ./vivado-riscv/patches where ./ symbolizes the directory that the repository was downloaded in.

5.5.4 Debian filesystem setup

Before running the script to generate the image of the final Debian system, the file system needs to be created. The reason behind creating the filesystem was to make it as light as possible, since the work case described already demands a substantial amount of the FPGA's resources. In order to create the filesystem, the following procedure was executed:

Firstly, similar to the Kernel generation, all the needed libraries need to be installed. Those libraries involve *debootstrap, dpkg-cross, debian-ports* etc. After the installation, the minimal bootstrap root filesystem has to be generated. In order for this to be done, the following command was run:

```
sudo debootstrap --arch=riscv64 --foreign
--keyring /usr/share/keyrings/debian-ports-archive-keyring.gpg
--include=debian-ports-archive-keyring sid ./temp-rootfs
http://deb.debian.org/ debian-ports
```

After the execution of the command, a directory will appear in which the filesystem is stored. Every change regarding the rootfs must be done inside the directory. First thing to do is to update the list of the debian sources. In this implementation the following sources were used:

```
cat >/etc/apt/sources.list <<EOF
deb http://ftp.ports.debian.org/debian-ports sid main
deb http://ftp.ports.debian.org/debian-ports unstable main
deb http://ftp.ports.debian.org/debian-ports unreleased main
deb http://ftp.ports.debian.org/debian-ports experimental main
EOF
```

The next step is to create the network config in order for the FPGA to have a static ip. This was an important step as the connectivity through ssh is considered to be vital. Without static ip the process of each user to connect to the machine would be much more difficult. In order to config the networking service of the system the following commands were used:

```
cat >/etc/network/interfaces <<EOF
auto lo
iface lo inet loopback

auto eth0
```

```
iface eth0 inet static
address 192.168.0.77
netmask 255.255.255.0
gateway 192.168.0.1
```

Finally the creation of the fstab is done in order for the system to mount automatically and easier and the set of the hostname for presentation reasons.

```
cat >/etc/fstab <<EOF
LABEL=rootfs / ext4 user_xattr,errors=remount-ro 0 1
EOF

echo "debian-riscv" > /etc/hostname
```

Last thing to be done is to compress the rootfs as it is desired to be packed in a zip form.

5.5.5 Finalizing the system and booting

After preparing both the Kernel and the rootfs, the system image can be generated with the help of the Makefile that was referred earlier. The script will generate a .img file which then will be burned into an SD card. A crucial step in order for the FPGA to boot the system is to set in in the correct boot mode.

Table 1-2: **VC707 Board FPGA Configuration Modes**

Configuration Mode	SW13 DIP switch Settings (M[2:0])	Bus Width	CCLK Direction
Master BPI	010	x8, x16	Output
JTAG	101	x1	Not Applicable

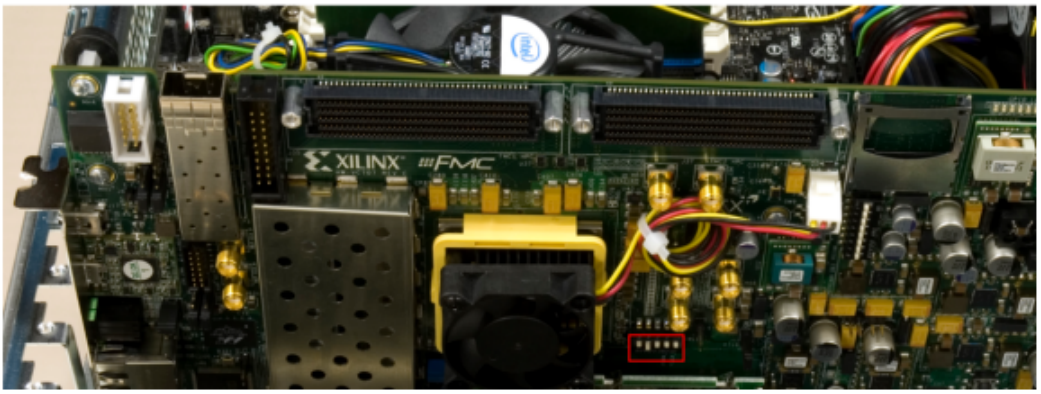

FIGURE 5.10: VC 707 dip-switching mode. URL

It is important for the booting process to be done through the serial port. In order for the system to boot the admin need to have access through serial port (Putty/Screen).

5.6 Docker

Docker is a widely-used platform that simplifies the process of developing, deploying, and managing applications within containerized environments. Containers are lightweight and isolated units that package an application and all its dependencies, ensuring consistency across various environments. Docker allows developers to create, share, and run containers effortlessly, making it an efficient solution for application deployment and scaling. By utilizing containerization, Docker enables developers to isolate applications from the underlying infrastructure, which enhances portability and minimizes potential compatibility issues. Additionally, Docker facilitates collaboration among teams, as it streamlines the process of sharing code and environments, leading to more efficient development

workflows. Its popularity is attributed to its versatility, efficiency, and ability to address challenges related to application deployment in both development and production environments.[43]

5.6.1 Why Docker?

Docker is the most popular tool among the others mainly because of its ease of use. Docker uses commands that are similar with those used in most of Linux systems. Docker's advantage is that it provides users with a consistent and isolated environment. It takes the responsibility of isolating and segregating your apps and resources in such a way that each container becomes able to access all the required resources in an isolated manner i.e., without disturbing or depending on another container. It eventually allows you to run multiple containers simultaneously on the same host[44]. Another great reason to use Docker is its portability. Images can either transferred and run on every platform or can be uploaded to the Docker Hub and be accessible to everyone.

5.6.2 Other available tools

During this thesis execution, some alternatives of Docker were examined. Among others, the most popular tools were LXC Linux Containers and Podman. Both tools were rejected for different reasons. LXC Linux Containers are considered to be much harder to use than Docker and do not provide the portability Docker does. Also, despite the fact that LXC is a Linux tool, the community that exists around it is not as big as the Docker one. Podman on the other hand, is a vastly growing alternative for Docker. The blocking factor for using the specific tool was that podman is not as user-friendly as Docker is and the documentation about it is very limited in comparison. Additionally, some third-party tools and services have better integration and support for Docker compared to Podman due to Docker's widespread adoption.

All in all, Docker seems to be the best option for an implementation like the one described in this thesis, as it provides both a wide variety of application and it is very friendly to the users.

5.6.3 Docker RISC-V Setup

One big setback of the system's implementation was that, according to the official Docker page, there was no support for Docker on RISC-V platforms. In order to successfully install the tool in the system and overcome the architectural barrier, a Debian executable file was found and packaged for the RISC-V platform. After some research was done on the Internet, there were some implementations provided and, after installing the latest package that had been found, the docker service is working properly as it can be seen in the screenshot below.

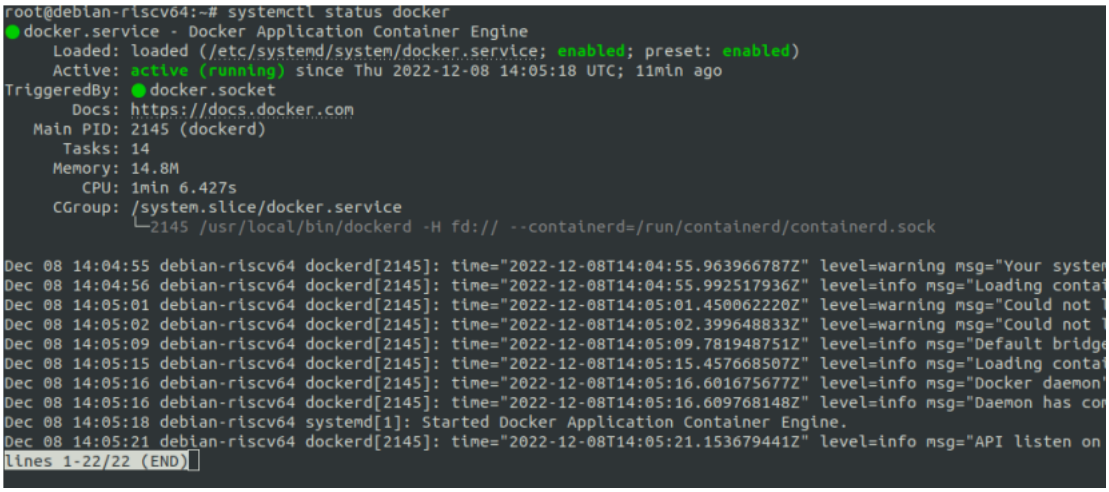

FIGURE 5.11: Docker service.

The Docker version of the package is 20.10.2. Due to the fact that the packaging of the tool was not done by its official provider it is important to mention that when typing the *docker –version* command, the version that will return as unknown.This can be seen in the figure *5.4*.

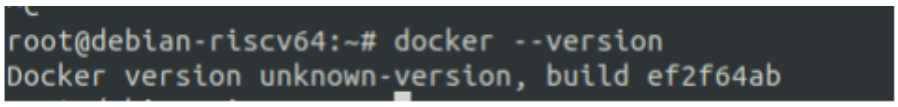

FIGURE 5.12: Docker versioning.

A detailed info view can be seen in the figure *5.5*, provided by running the command *docker info*. As it can be seen Docker uses overlay2 as primary filesystem and all the options provided are set as they were on normal platforms.

```
root@debian-riscv64:~# docker info
Client:
 Debug Mode: false

Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 0
 Server Version: dev
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
 Swarm: inactive
 Runtimes: runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version: d76c121f76a5fc8a462dc64594aea72fe18e1178
 runc version: e79e4de4ac16da0ce48777afb72c6241de870525
 init version: 3c53686 (expected: fec3683)
 Security Options:
  cgroupns
 Kernel Version: 5.19.16-dirty
 Operating System: Debian GNU/Linux bookworm/sid
 OSType: linux
 Architecture: riscv64
 CPUs: 8
 Total Memory: 968MiB
 Name: debian-riscv64
 ID: 2RUF:BS4O:HYSA:MF55:4ORX:WSE5:2LXG:FBZN:M3TK:5ZJK:DL4B:CQSG
 Docker Root Dir: /var/lib/docker
 Debug Mode: false
 Registry: https://index.docker.io/v1/
 Labels:
 Experimental: false
 Insecure Registries:
  127.0.0.0/8
 Live Restore Enabled: false

WARNING: No kernel memory limit support
WARNING: No kernel memory TCP limit support
```

FIGURE 5.13: Docker Info.

5.6.4 Docker Base Image

Each Dockerfile that can be created must be originated by principle by another Image. The creation of custom base Images in Docker projects is a very good practice that is adopted by a lot of companies nowadays. The idea behind that is to provide an image that contains as wide an array of tools and libraries that can be used in any possible use case. In that way, when the user creates an Image for in this case test a bitstream, they can do that providing the custom base Image that is already in the system. In that

way, everything that is needed as a general rule is already packaged and only the personalized changes need to be applied. As a result the Image creation times reduces vastly.

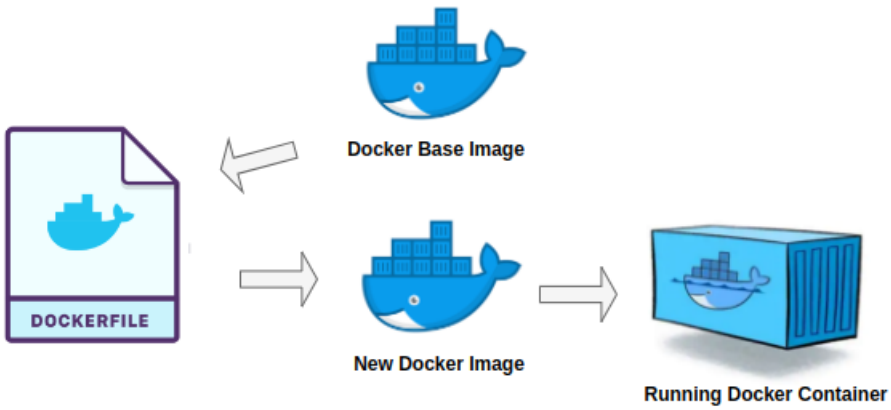

FIGURE 5.14: Docker Container Production Process.

The base Docker image implementation for this thesis is the following:

```
FROM riscv64/debian:sid
COPY id_rsa /home/
COPY test_bash.sh /home/
WORKDIR  /home/
RUN chmod 600 id_rsa
RUN chmod 700 program_bitsream.sh
```

As it can be seen from the code above, the base image used for the custom base image is the debian:sid one. This selection was made due to the lack of available docker images existing. Besides that, the Image is very simple as it only copies needed files as a key to verify the integrity of the bitstream programming source and the script that programs the bitstream itself.

5.6.5 User Implementation

As described in the *Overall Idea of the System*, the user input plays a great part in the system's workflow. While connecting to their respective account

the user can find in their home directory a *Dockerfile* that must be used as a template. In other words, the user must fill in a given template of a Dockerfile that exists in their home folder. The Dockerfile template has as origin the base Image that got created as described in the *Docker Base Image* subsection. The only thing that the user is called to add is a *COPY* command, in order to copy their bitstream into their dedicated container. After running the filled Dockerfile a new image is created. This helps in terms of both portability and reusability. As the Image stays in the Docker Images section it can be used by the user as many times it is required to and can also be transferred easily to a different platform or even get uploaded into the Docker Hub.

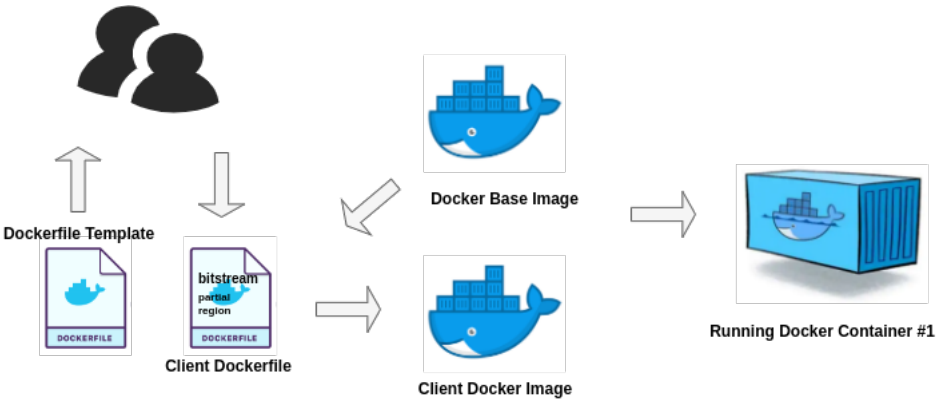

FIGURE 5.15: User's Container production flow.

After the image generation, the user runs the container and executes the program_bitstream.sh file that is already there due to the base image. This bash file, sends the imported bitsream to the gateway in which the board is connected. The process is done through scp (SECURE COPY). This also explains the id_rsa file which is a private RSA generated key. When the file reaches the appropriate location in the gateway, with the help of a CLI tool called xsdb, the partial reconfiguration is achieved.

Chapter 6

Results

6.1 CPU Comparison

6.1.1 Assumptions

RocketChip generator can create both 32 bit and 64 bit architectures. x32 bit CPUs are lacking the ability to support Linux Distributions. This specific problem is crucial, due to the Docker need of an Operating System. Therefore, they were not taken into account in the following comparison.

In order for the comparison of the Processors to be fair across the different implementations, the partial region is considered to be one and of specific/ static size. This assumption is considered valid as it complies with the specification of the POC that has been explained earlier. With that in mind, this thesis will present firstly the characteristics of each implementation and then, finally, provide a summary and conclusions.

It is important to mention at this point, that all the measurements that are depicted in the following sections are generated in Xilinx Vivado 2022.2.

6.1.2 RISC-V Single Core Processor

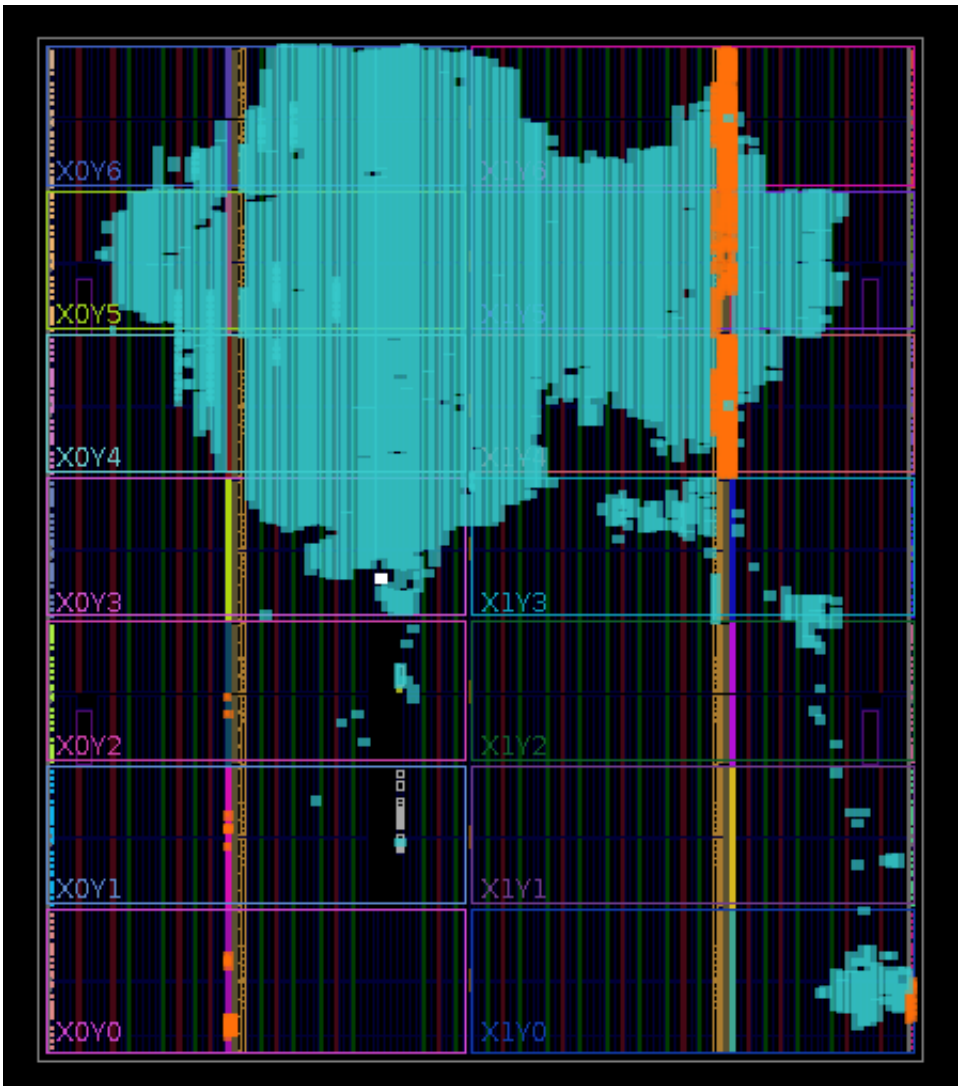

FIGURE 6.1: Single Core Processor Region on FPGA.

Resource	Utilization	Available	Utilization %
LUT	58593	303600	19.30
LUTRAM	6270	130800	4.79
FF	38732	607200	6.38
BRAM	27	1030	2.62
DSP	15	2800	0.54
IO	133	700	19.00
GT	1	28	3.57
BUFG	12	32	37.50
MMCM	4	14	28.57
PLL	1	14	7.14

FIGURE 6.2: Single Core Processor FPGA Utilization.

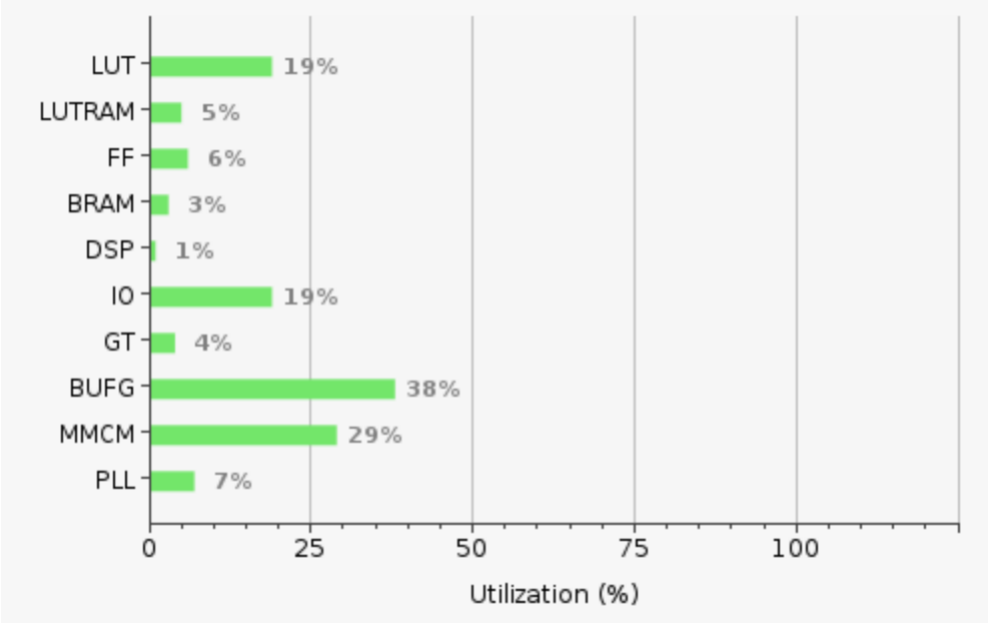

FIGURE 6.3: Single Core Processor FPGA Utilization (%).

6.1.3 RISC-V Dual Core Processor

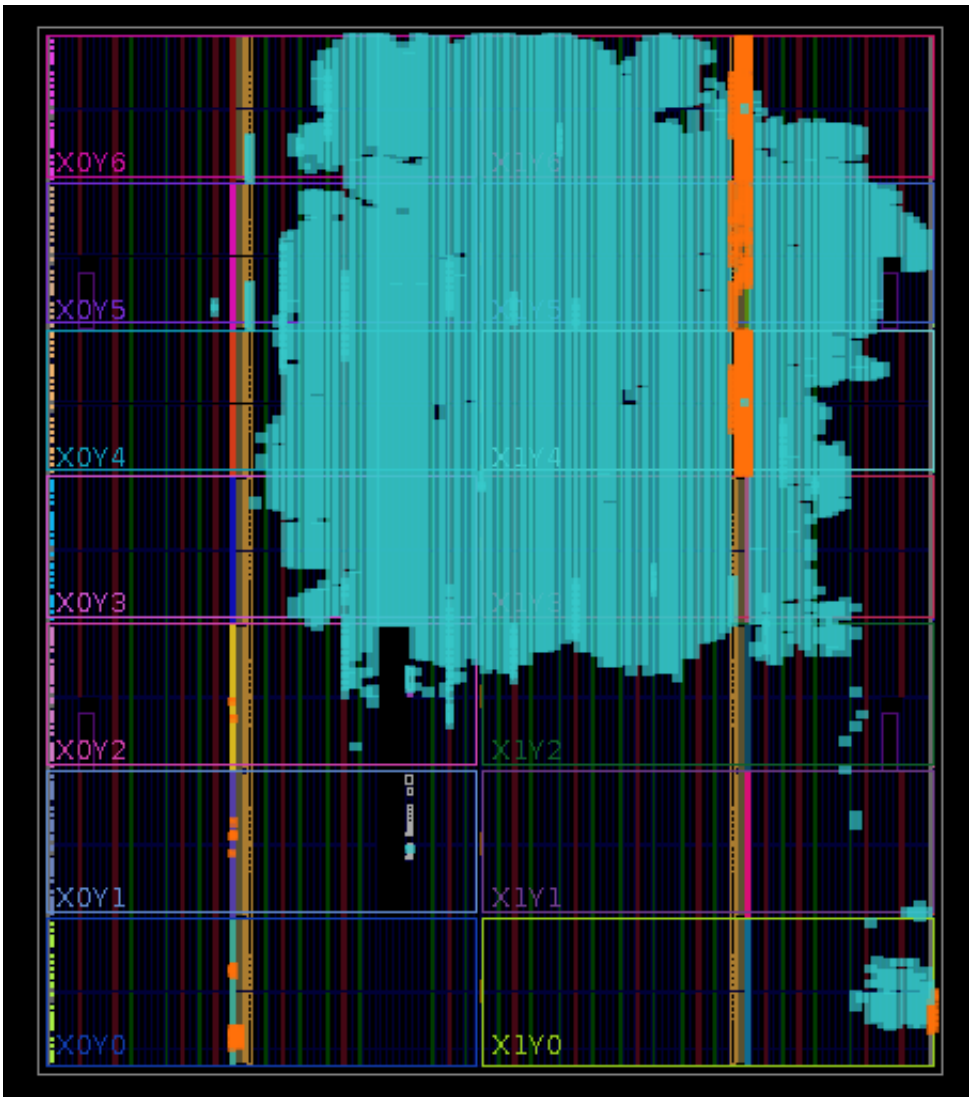

FIGURE 6.4: Dual Core Processor Region on FPGA.

Resource	Utilization	Available	Utilization %
LUT	85459	303600	28.15
LUTRAM	6805	130800	5.20
FF	52148	607200	8.59
BRAM	51	1030	4.95
DSP	30	2800	1.07
IO	133	700	19.00
GT	1	28	3.57
BUFG	12	32	37.50
MMCM	4	14	28.57
PLL	1	14	7.14

FIGURE 6.5: Dual Core Processor FPGA Utilization.

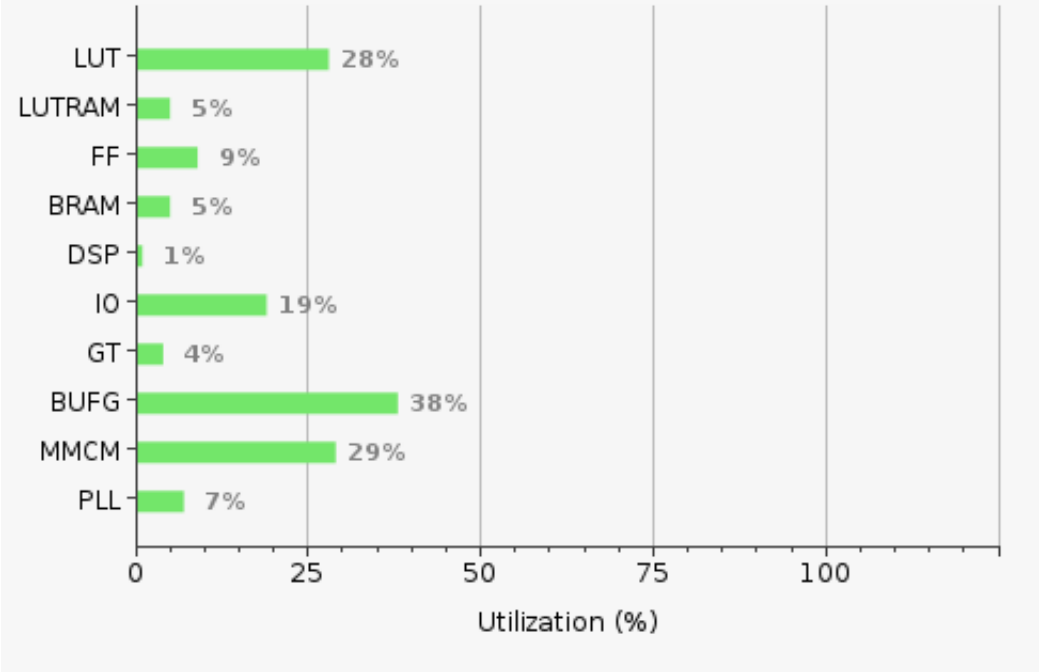

FIGURE 6.6: Dual Core Processor FPGA Utilization (%).

6.1.4 RISC-V Quad Core Processor

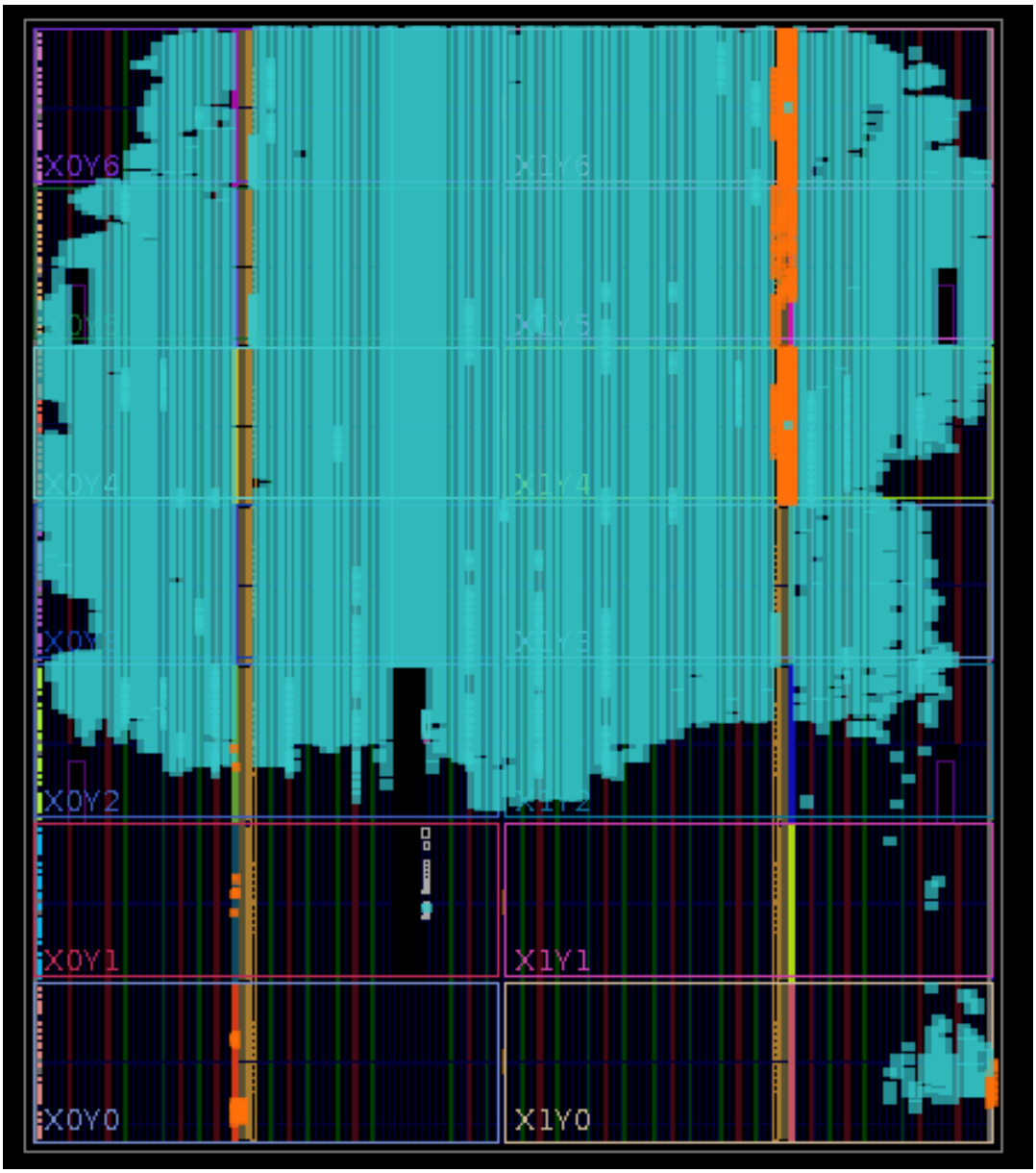

FIGURE 6.7: Quad Core Processor Region on FPGA.

Resource	Utilization	Available	Utilization %
LUT	139351	303600	45.90
LUTRAM	7877	130800	6.02
FF	79810	607200	13.14
BRAM	99	1030	9.61
DSP	60	2800	2.14
IO	133	700	19.00
GT	1	28	3.57
BUFG	12	32	37.50
MMCM	4	14	28.57
PLL	1	14	7.14

FIGURE 6.8: Quad Core Processor FPGA Utilization.

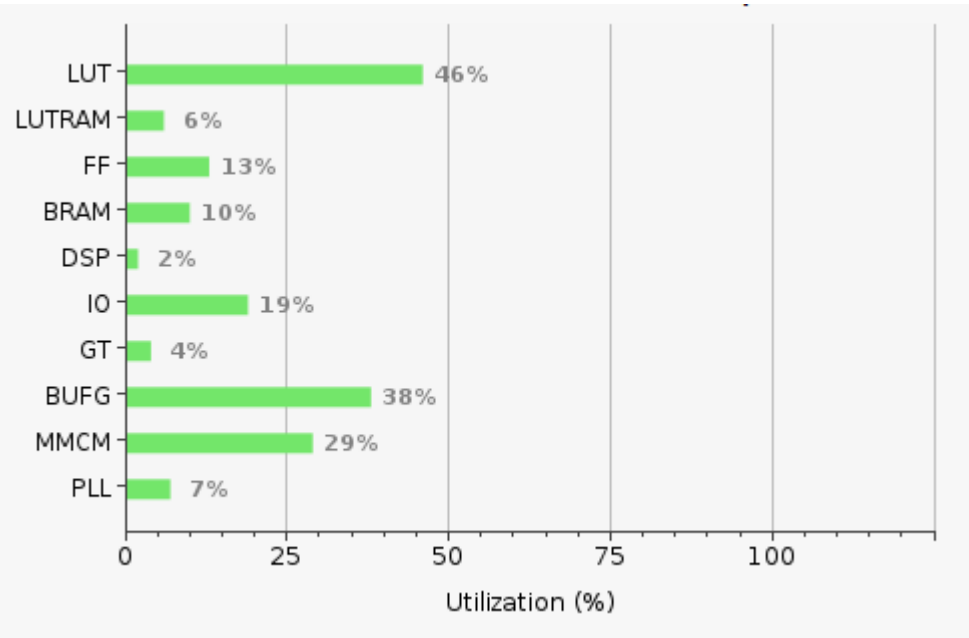

FIGURE 6.9: Quad Core Processor FPGA Utilization (%).

6.1.5 RISC-V Octa Core Processor

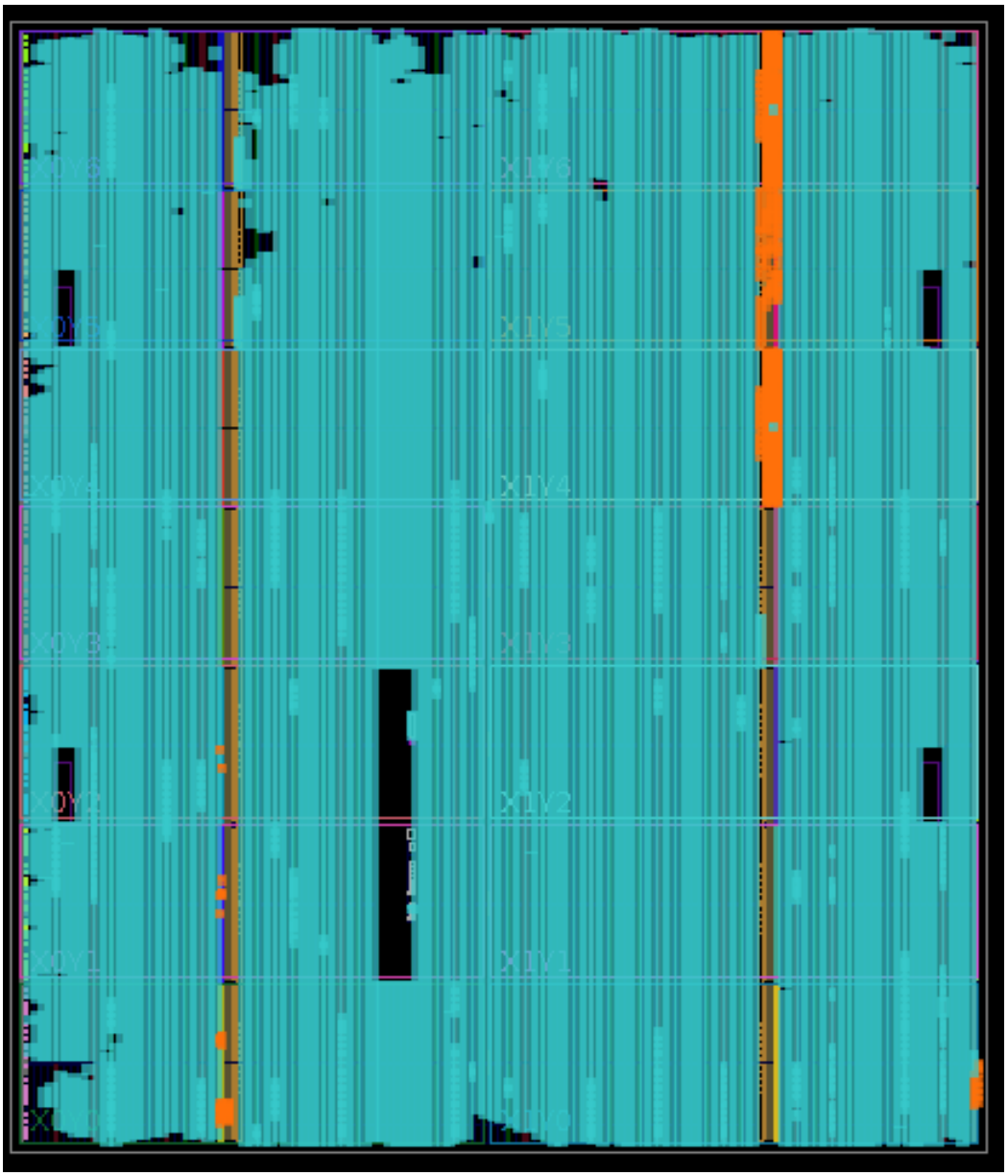

FIGURE 6.10: Octa Core Processor Region on FPGA.

Resource	Utilization	Available	Utilization %
LUT	246018	303600	81.03
LUTRAM	9545	130800	7.30
FF	132875	607200	21.88
BRAM	195	1030	18.93
DSP	120	2800	4.29
IO	133	700	19.00
GT	1	28	3.57
BUFG	12	32	37.50
MMCM	4	14	28.57
PLL	1	14	7.14

FIGURE 6.11: Octa Core Processor FPGA Utilization.

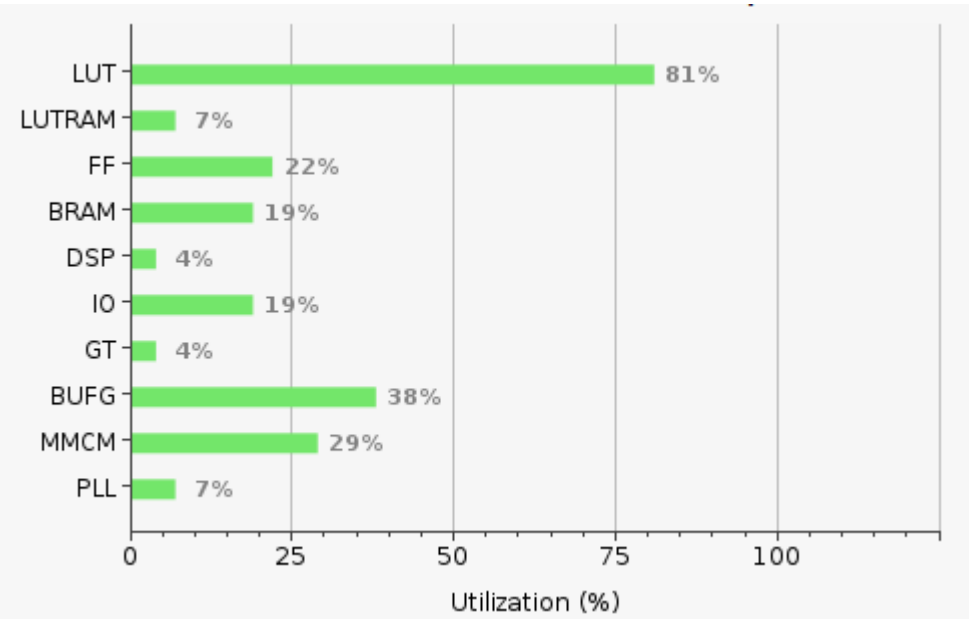

FIGURE 6.12: Octa Core Processor FPGA Utilization (%).

6.1.6 Summary

After presenting all the necessary info about the utilization of each CPU implementation, a summary of those number is presented bellow in the form of a table:

	LUT	LUTRAM	FF	BRAM	DSP	IO	GT	BUFG	MMCM	PLL
Single-Core Processor	58593	6270	38732	27	15	133	1	12	4	1
Dual-Core Processor	85459	6805	52148	51	30	133	1	12	4	1
Quad-Core Processor	139351	7877	79810	99	60	133	1	12	4	1
Octa-Core Processor	246018	9545	13285	195	120	133	1	12	4	1

TABLE 6.1: Utilization Table of all CPU implementations

As can be seen from the table 6.1, there is an almost linear increase of Flip-Flops and BRAM. DSP utilization is increasing linearly also. All of these values are expected and the deviations that occur are a product of Vivado optimizations. LUT utilization increases about one and a half times per row. This increase is also considered to be within reasonable numbers, due to the existence of Task and Data Parallelism. I/O subsystem in each core remains the same in each implementation so its utilization is expected to be the same across all CPUs. Every other value that is presented in the table (GT, BUFG, MMCM, PLL) is about the Clocks that are present in each CPU implementation and since they are set to be equal across all the implementations that are shown, the reasons that their respective values are equal are justified.

In addition to the above summary, Power Consumption is considered to be a very important metric. Therefore, the table presenting the power utilization of each CPU implementation is shown below (Table 6.2).

	Overall	GTX	Dynamic	Device's Static
Single-Core Processor	3.407	0.237	2.883	0.287
Dual-Core Processor	3.678	0.237	3.150	0.291
Quad-Core Processor	3.979	0.237	3.444	0.298
Octa-Core Processor	4.38	0.237	3.832	0.311

TABLE 6.2: Power(W) on CPU Implementations

Vivado also analyzes the Dynamic Power Consumption in categories via this graph.

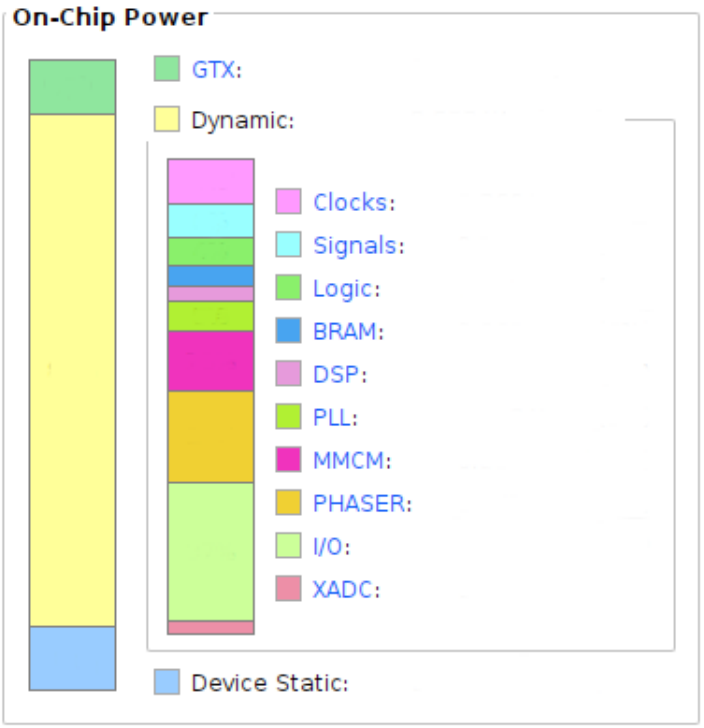

FIGURE 6.13: Power Consumption Template(**W**).

6.1.7 Conclusions

Due to the fact that the Octa-Core CPU implementation had about 81% of LUT usage which is considered enough space to fulfill our POC requests about partial reconfiguration region space. Since Power Consumption was considered to be out of the scope of this thesis, the main criterion about the CPU implementation that was taken into account was purely performance. Since the cores, provide more parallelism and sequentially better performance, the Octa-Core implementation was considered the best fit.

6.2 Docker Stats

An important stat that is considered important to be displayed is the amount of the image each user deploys. An example of this can be seen in the following picture:

```
debian@debian:~$ docker image ls
REPOSITORY          TAG         IMAGE ID       CREATED          SIZE
pr1_test            latest      d9f0b434600e   2 hours ago      99.6MB
base_image          latest      596bbb4b07dc   2 months ago     99.3MB
```

FIGURE 6.14: Docker Image size comparison.

The difference between sizes equals to 0.3MB. This amount is the exact amount of the partial bitstream. In other words, each time a user tries to build a new implementation as an image, the only action that takes place into the docker file is the copy of the user's bitstream to the container.

6.3 System Flow Results

The results of the implementation are divided in three categories. The responses that the user is taking when the region they want to utilize is not available, the process during the reconfiguration of the respective region and the execution of a task on that region and finally, the stats that are provided by Docker about the utilization of the system's resources during that process.

6.3.1 User Denial Of Service on Partial Region

While it is important for the user to be able to utilize the desired region, it is vital for the system's integrity to have a mechanism for preventing users to deploy a container that takes advantage over a region already in use.
To achieve that, a bash command has been created, named "resources". In order for that command to work properly, there is an acceptance that has to be made. All users are obliged to name their containers/images in a "PR<number>_*" convention.

The results of the command are the following:

```
debian@debian:~$ resources -h
USAGE: resources -r <region_name>
Regions available: PR1
```

FIGURE 6.15: resources command help flag.

For this thesis implementation there is only one partial reconfiguration region, with name PR1. So, when typing the resources command with that region the results are the following:

```
debian@debian:~$ docker run --name pr1_test -it pr1_test:latest bash
root@26c7c45699f4:/home#
```

FIGURE 6.16: resources pr1 already in use from user.

```
debian2@debian:~$ resources -r pr1
pr1 region is already in use
debian2@debian:~$
```

FIGURE 6.17: resources pr1 already in use from user 2.

6.3.2 Docker Container Partial Region Execution

In order to present the container execution results, there is a need to provide some initial background. As described in the previous chapter, the bitstreams provided for this implementation were two. An add and a subtract module. The user achieves the interaction with these two partialy configurable modules through a tool named devmem2. Devmem2 provides an easier interaction with the /dev/mem in order to read and write data from the addresses that the modules are configured. The way of usage can be seen in the picture bellow:

```
debian@debian:~$ ./devmem2

Usage:  ./devmem2 { address } [ type [ data ] ]
        address : memory address to act upon
        type    : access operation type : [b]yte, [h]alfword, [w]ord
        data    : data to be written
```

FIGURE 6.18: devmem2 usage.

The usage of the partial reconfiguration modules can be seen in the following pictures:

```
root@debian-riscv64:~# docker run --privileged -it testimage:latest bash
root@6b5b7b7d8b3a:/app# ls /dev/m
mapper/    mem        mmcblk0    mmcblk0p1  mmcblk0p2  mqueue/
root@6b5b7b7d8b3a:/app# ls /dev/m
mapper/    mem        mmcblk0    mmcblk0p1  mmcblk0p2  mqueue/
root@6b5b7b7d8b3a:/app# ls /dev/m
mapper/    mem        mmcblk0    mmcblk0p1  mmcblk0p2  mqueue/
root@6b5b7b7d8b3a:/app# ./devmem2 0x60040010
/dev/mem opened.
Memory mapped at address 0x3f8cc5f000.
Value at address 0x60040010 (0x3f8cc5f010): 0x0
root@6b5b7b7d8b3a:/app# ./devmem2 0x60040020 w 2
/dev/mem opened.
Memory mapped at address 0x3f894ec000.
Value at address 0x60040020 (0x3f894ec020): 0x0
Written 0x2; readback 0x2
root@6b5b7b7d8b3a:/app# ./devmem2 0x60040018 w 2
/dev/mem opened.
Memory mapped at address 0x3faa417000.
Value at address 0x60040018 (0x3faa417018): 0x0
Written 0x2; readback 0x2
root@6b5b7b7d8b3a:/app# ./devmem2 0x60040000 b 1
/dev/mem opened.
Memory mapped at address 0x3f90a8a000.
Value at address 0x60040000 (0x3f90a8a000): 0x4
Written 0x1; readback 0x6
root@6b5b7b7d8b3a:/app# ./devmem2 0x60040010
/dev/mem opened.
Memory mapped at address 0x3fa3086000.
Value at address 0x60040010 (0x3fa3086010): 0x4
root@6b5b7b7d8b3a:/app#
```

FIGURE 6.19: User interaction results on adder module.

```
root@e4b63b689c66:/home# ./devmem2 0x60040018 w 4
/dev/mem opened.
Memory mapped at address 0x3f91232000.
Value at address 0x60040018 (0x3f91232018): 0x2
Written 0x4; readback 0x4
root@e4b63b689c66:/home# ./devmem2 0x60040020 w 1
/dev/mem opened.
Memory mapped at address 0x3faf7c6000.
Value at address 0x60040020 (0x3faf7c6020): 0x4
Written 0x1; readback 0x1
root@e4b63b689c66:/home# ./devmem2 0x60040000 b 1
/dev/mem opened.
Memory mapped at address 0x3fa413a000.
Value at address 0x60040000 (0x3fa413a000): 0x4
Written 0x1; readback 0x6
root@e4b63b689c66:/home# ./devmem2 0x60040010
/dev/mem opened.
Memory mapped at address 0x3fb8e1a000.
Value at address 0x60040010 (0x3fb8e1a010): 0x3
root@e4b63b689c66:/home#
```

FIGURE 6.20: User interaction results on subb module.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

With modern hardware applications becoming more and more complex and teams becoming larger in terms of human numbers, more and more techniques are adopted in order to test any possible implementation before putting it into production. Also research around RISC-V is vastly growing. There are a lot of projects regarding improvements in efficiency, in security and other fields. More and more teams use online platforms like Amazon Web Services (AWS) or Microsoft Azure. The problem that arises with those platforms is that, not only the extension the team works on has to be set up, but also the rest of the design. They provide "empty" FPGAs with their main use to be bare-metal applications. This thesis proposal could help teams test their implementation on various accelerators of a RISC-V processor as long as the communication requirements are fulfilled. Finaly, due to the protability and reusability that Docker engine provides, teams can share their work in the form of images that can be deployed in any similar platform. All in all, it is an application can contribute to the community in various ways.

7.2 Future Work

This thesis implementation is a proof of concept. Therefore, there are a lot of improvements that can be done. First of all, provided a larger FPGA or even a series of FPGAs, there could be more partial regions that will consequently result in more users that could be served. Additionally, given more resources, some of the procedures (e.g. the execution of the Dockerfiles, or the running of the Docker containers) could be faster. The ssh connection should be replaced by a graphical interface that will make the application more user-friendly and will remove the need of technical knowledge of this part. Also, the addition of a scheduler that will keep time-based incoming request is an idea that can be implemented and be very useful to the project. Another suggestion could be that drivers should be created for the partially re-configurable regions in order for the users to not be obliged to access their implementations by direct access to the physical addresses (/dev/mem). Finally, privilege hierarchy can be implemented for security reasons. User's should not be able to have interaction with docker features in any way except through system's executed scripts or, later, through the graphical interface. This will help avoid situations where users ignore the system's feedback and e.g. spawn a container in a region that are not allowed. All of these changes proposed are considered to be feasible to be done and will improve the current system by a significant amount.

Appendix A

Frequently Asked Questions

A.1 How Hardware specific is the Operating System?

Although the operating system was designed for the VC-707 board, the Operating System is functional on any RISC-V based platform with the only adjustment needed being the alteration of the device tree with changes appropriate to the hardware that is to be used.

A.2 Does Docker still support software containers?

Even though minimal changes have been made to the Docker package, its functionality has not been affected. The only limitation existing is that there aren't as many images built for the RISC-V architecture as for other architectures. Network creation functionality has also not been affected.

References

[1] U. F. Habib Mehrez Zied Marrakchi, "Fpga architectures: An overview," no. 1, p. 7, Jan. 2012. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4614-3594-5_2.

[2] S. M. Trimberger, "Field-programmable gate array technology," no. 1, p. 7, Jan. 2012. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4614-3594-5_2.

[3] C. W. Winser Alexander, "Digital signal processing systems design," pp. 519–542, 2017. [Online]. Available: https://www.sciencedirect.com/book/9780128045473/digital-signal-processing#book-info.

[4] M. C. Lucas Bragança, W. C. Jeronimo Penha, and R. F. José Augusto M. Nacif, "An open source custom k-means generator for aws cloud fpga accelerators," 2012. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9628301.

[5] F. L. X. Wang Y. Niu and Z. Xu, "When fpga meets cloud: A first look at performance," 2020. [Online]. Available: https://fangmingliu.github.io/files/TCC20-FPGA-Cloud.pdf.

[6] Z. L. Ming Liu Wolfgang Kuehn and A. Jantsch, "Run-time partial reconfiguration speed investigation and architectural design space exploration," no. 29, Sep. 2009. [Online]. Available: https://ieeexplore.ieee.org/document/5272463.

[7] K. Vipin and S. A. Fahmy, "Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications," vol. 51, no. 4, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3193827.

[8] C. Kao, "Benefits of partial reconfiguration," 2005. [Online]. Available: https://islab.soe.uoguelph.ca/sareibi/TEACHING_dr/ENG6530_RCS_html_dr/outline_W2017/docs/PAPER_REVIEW_dr/RTR_RCS_dr/xc_reconfig55.pdf.

[9] J. G. Tong, I. D. L. Anderson, and M. A. S. Khalid, "Soft-core processors for embedded systems," pp. 170–173, 2006.

[10] F. Plavec, "Soft-core processor design," pp. 2–11, 2004.

[11] D. Inc., "What is a container?," 2021. [Online]. Available: https://www.docker.com/resources/what-container.

[12] T. Combe, A. Martin, and R. D. Pietro, "To docker or not to docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, 2016.

[13] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[14] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[15] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops," *IEEE Software*, vol. 33, no. 3, pp. 94–100, 2016.

[16] I. Red Hat, "Podman: A more secure way to run containers," 2021. [Online]. Available: https://www.redhat.com/en/topics/containers/what-is-podman.

[17] IONOS, "Lxc: Features, pros, and cons of linux containers," 2021. [Online]. Available: https://www.ionos.com/digitalguide/server/know-how/what-is-lxc-linux-containers/.

[18] Atatus, "Getting started with linux containers: A beginner's guide," 2021. [Online]. Available: https://www.atatus.com/blog/linux-containers-beginners-guide/.

[19] E. Blog, "Lxc vs docker: Which container platform is right for you?," 2021. [Online]. Available: https://earthly.dev/blog/lxc-vs-docker/.

[20] Quora, "What are the downsides of using linux containers (lxd or lxc)?," 2021. [Online]. Available: https://www.quora.com/What-are-the-downsides-of-using-Linux-containers-LXD-or-LXC.

[21] D. Documentation, "Docker overview," 2023. [Online]. Available: https://docs.docker.com/get-started/overview/.

[22] M. Learn, "Development workflow for docker apps," 2023. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/docker-application-development-process/docker-app-development-workflow.

[23] C. Eduardo, "Docker containers on risc-v architecture," 2023. [Online]. Available: https://carlosedp.medium.com/docker-containers-on-risc-v-architecture-5bc45725624b.

[24] A. Vaishnav, K. D. Pham, and D. Koch, "A survey on FPGA virtualization," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, Aug. 2018. DOI: 10.1109/fpl.2018.00031.

[25] P. Zheng, T. Benson, and C. Hu, "P4visor," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, ACM, Dec. 2018. DOI: 10.1145/3281411.3281436.

[26] A. Vaishnav, K. D. Pham, D. Koch, and J. Garside, "Resource elastic virtualization for FPGAs using OpenCL," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, Aug. 2018. DOI: 10.1109/fpl.2018.00028.

[27] A. A. Al-Aghbari and M. E. S. Elrabaa, "Cloud-based FPGA custom computing machines for streaming applications," *IEEE Access*, vol. 7, pp. 38 009–38 019, 2019. DOI: 10.1109/access.2019.2906910.

[28] X. Long, B. Liu, F. Jiang, Q. Zhang, and X. Zhi, "FPGA virtualization deployment based on docker container technology," in *2020 5th International Conference on Mechanical, Control and Computer Engineering (ICMCCE)*, IEEE, Dec. 2020. DOI: 10.1109/icmcce51767.2020.00109.

[29] N. Preeth E, F. J. P. Mulerickal, B. Paul, and Y. Sastri, "Evaluation of docker containers based on hardware utilization," in *2015 International Conference on Control Communication &amp Computing India (ICCC)*, IEEE, Nov. 2015. DOI: 10.1109/iccc.2015.7432984.

[30] J. Lallet, A. Enrici, and A. Saffar, "FPGA-based system for the acceleration of cloud microservices," in *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, IEEE, Jun. 2018. DOI: 10.1109/bmsb.2018.8436912.

[31] J. Gray, "GRVI phalanx: A massively parallel RISC-v FPGA accelerator accelerator," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, May 2016. DOI: 10.1109/fccm.2016.12.

[32] F. Schuiki, F. Zaruba, T. Hoefler, and L. Benini, "Stream semantic registers: A lightweight RISC-v ISA extension achieving full compute utilization in single-issue cores," *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 212–227, Feb. 2021. DOI: 10.1109/tc.2020.2987314.

[33] N. Bruschi, G. Haugou, G. Tagliavini, F. Conti, L. Benini, and D. Rossi, "GVSoC: A highly configurable, fast and accurate full-platform simulator for RISC-v based IoT processors," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*, IEEE, Oct. 2021. DOI: 10.1109/iccd53106.2021.00071.

[34] K. Asanović, R. Avizienis, J. Bachrach, *et al.*, "The rocket chip generator," 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:5364470.

[35] Xilinx, "Virtex 7 fpga family," 2023. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html.

[36] Xilinx, "Amd virtex 7 fpga vc707 evaluation kit," 2023. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html.

[37] Xilinx, "Vivado ml overview," 2023. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html.

[38] Xilinx, "Vitis hls," 2023. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html.

[39] Xilinx, "Axi adapter interface protocols," 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/AXI-Adapter-Interface-Protocols.

[40] E. Tarasov, *Vivado-risc-v*. [Online]. Available: https://github.com/eugene-tarassov/vivado-risc-v.

[41] K. A. et al, *The rocket chip generator*. [Online]. Available: http://palms.ee.princeton.edu/system/files/RocketChipGenerator.pdf.

[42] G. Fichtner, "Verify your linux kernel for container compatibility," Apr. 2019. [Online]. Available: https://blog.hypriot.com/post/verify-kernel-container-compatibility/.

[43] Docker, *Docker official documentation*. [Online]. Available: https://docs.docker.com/.

[44] madhur912, *Why should you use docker*. [Online]. Available: https://www.geeksforgeeks.org/why-should-you-use-docker-7-major-reasons/.