

Custom Linux Development for FPGA-Based Systems

Department Lippstadt 2

Project Work

submitted by

Mohammad Ashrafuzzaman Siddiqi

Electronic Engineering

Mat.Nr.: 2200011

mohammad-ashrafuzzaman.siddiqi@stud.hshl.de

February 23, 2025

First supervisor: Prof. Dr. Ali Hayek

Second supervisor: xyz

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Goals	2
1.3	Overview	3
2	Fundamentals	4
2.1	Definition of FPGAs	4
2.1.1	FPGA Component	4
2.1.2	FPGA Development Process	5
2.2	Operating Systems	6
2.2.1	General Purpose Operating Systems(GPOS)	6
2.2.2	Real Time Operating system	7
2.2.3	Characteristics of RTOS	7
2.2.4	Types of RTOS	7
2.2.5	RTOS Architecture	7
2.2.6	RTOS Examples	8
3	Related Work	9
3.1	A Programmable Linux-Based FPGA Platform for Audio DSP	9
3.2	A Linux-based Support for Developing Real-Time Applications on Heterogeneous Platforms with Dynamic FPGA Reconfiguration	10
3.3	RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators	10
3.4	Embedding Linux with Ability to Analyze Network Traffic on a Development Board Based on FPGA	12
4	Analysis	13
4.1	Workflow	13
5	Design and Implementation	15
5.1	Environment Setup	16
5.1.1	Tools and Dependencies	16
5.1.2	Cloning and Preparing the Repository	16
5.2	Hardware Design	17
5.2.1	Building Our Processor Core for the Target device-Virtex VC707	19
5.2.2	Validation of Design	21
5.3	Software Design	22
5.3.1	Linux Test	23
5.4	Application and System Validation Tests	25
5.4.1	Guess the Number Game on FPGA Terminal	25
5.4.2	AES Encryption/Decryption Application	26

5.4.3	Conclusion	28
5.4.4	Network Application Testing via SSH on RISC-V Linux Environment	28
5.4.5	FPGA as a Web Server for Project Dashboard	30
6	Evaluation	34
6.1	CoreMark Benchmark of Our Platform	34
6.1.1	Transferring CoreMark to the FPGA	34
6.1.2	Compiling and Running CoreMark with Make	35
6.1.3	Benchmark Results and Analysis	35
6.1.4	Conclusion	36
6.2	Linpack Benchmark of Our Platform	37
6.2.1	Procedure to Run Linpack	37
6.2.2	Results	38
6.2.3	Performance Analysis	38
6.2.4	Conclusion	39
7	Outlook and Summary	40
	References	41
A	Appendix	45
A.1	Code in detail	45

1 Introduction

This work presents a comprehensive methodology aimed at implementing a Linux distribution on a Field-Programmable Gate Array (FPGA), coupled with the development of various applications to rigorously assess the integrity and robustness of the overall system. The approach undertaken in this project underscores the multifaceted nature and inherent reconfigurability of FPGAs, offering a compelling avenue for exploring the convergence of embedded systems, operating systems, and cybersecurity.

Field-Programmable Gate Arrays (FPGAs) represent a dynamic class of programmable hardware devices that afford unparalleled flexibility and adaptability in system design. Unlike traditional fixed-function integrated circuits, FPGAs can be dynamically configured and reprogrammed to implement custom logic circuits, making them particularly well-suited for a wide range of applications, from digital signal processing to embedded system development.

The crux of this project lies in the realization of a fully functional Linux distribution on an FPGA platform, leveraging the inherent parallelism and configurability of FPGAs to emulate the functionalities of a conventional computing environment. By harnessing the power of FPGA-based emulation, the project aims to demonstrate the feasibility of running a Linux-based operating system on hardware that traditionally operates at a lower level of abstraction.

Furthermore, the project endeavors to complement the Linux distribution implementation with the development of a robust security application, designed to probe the system's vulnerabilities and evaluate its resilience against various cyber threats. This entails extensive research into diverse Linux distributions, cryptographic algorithms, and security protocols to inform the design and implementation of effective security measures tailored to the FPGA-based environment.

Key to the success of this endeavor is a thorough investigation into existing Linux distributions, including their architecture, resource requirements, and compatibility with FPGA platforms. Additionally, the project will delve into the intricacies of encryption algorithms, authentication mechanisms, and intrusion detection techniques to devise a comprehensive security framework capable of safeguarding the integrity and confidentiality of data processed within the FPGA-based system.

In summary, this project represents a concerted effort to explore the synergies between FPGA-based system design, Linux distribution implementation, and cybersecurity, with the ultimate goal of advancing the state-of-the-art in embedded computing and enhancing the security posture of FPGA-based applications.

1.1 Motivation

The primary motivation for this project is my desire to be proficient in the field of embedded technologies. I intend to work in this field after having completed my degree. This dream requires an in-depth understanding of hardware engineering and embedded systems, along with every other sub-discipline it entails. My interest was sparked by completing the advanced embedded course, where I undertook a project that involved running petalinux on a custom Microblaze processor. This served as an entry point into embedded engineering and exposed me to the fascinating world of hardware and software co-design. I believe that upon completing this project successfully, I will be even more grounded in this field and able to compete in the job market upon graduation.

Aside from the skills I will gain to enhance my career, I was also drawn to the project because of the opportunity to try my hands at advanced hardware like the Genesys board and Virtex board. I have always been especially keen on technology, even in my early days. I am excited to be able to work with industry-standard boards and technologies. Another important factor motivating this project is the possibility of not just working with open-source ecosystems but also contributing any knowledge discovered back into the open-source community or even into academia in the form of a paper.

All around, I am very excited to carry out my first complete project as an engineer. I am excited to be able to define requirements, research, analyse, implement, and evaluate this work, putting into practice everything I have learned in the previous semesters.

1.2 Goals

The objective of this project is to develop a custom Linux distribution tailored for FPGA-based systems, specifically targeting the Nexys A7 or Genesys 2 board equipped with Xilinx Artix-7 or Virtex FPGA. This distribution will be optimized to run on a soft processor core (ARM or RISC-V architecture) instantiated on the FPGA. The project will also include the development of a Python-based encryption/decryption application to showcase the custom distribution's functionality and performance on the hardware platform.

1. Hardware Setup:

- Selection of board based on availability and suitability for the project requirements.
- Configuration of the FPGA with a soft processor core to emulate an ARM or RISC-V processing environment.

2. Custom Linux Distribution Development:

- Definition of requirements for the custom distribution, focusing on minimalism and performance to suit the soft processor and FPGA constraints.
- Compilation and customization of the Linux kernel to support the soft processor architecture and the hardware peripherals available on the board.

- Selection and integration of necessary packages and drivers to support the development and execution of Python applications, specifically focusing on encryption and decryption tasks.

3. Python Application for Encryption and Decryption:

- Design and implementation of a Python-based application capable of performing encryption and decryption operations, leveraging popular cryptographic libraries for robust security measures.
- Optimization of the application to run efficiently on the custom Linux distribution, including performance benchmarking and resource management analysis.

4. Testing and Benchmarking:

- Comprehensive testing of the custom Linux distribution on the FPGA-based system, including stability, performance, and compatibility assessments.
- Performance benchmarking of the Python encryption/decryption application, focusing on execution speed, resource utilization, and scalability.

5. Documentation and Reporting:

- Detailed documentation of the system architecture, software development process.
- Compilation of testing and benchmarking results, including performance analysis and optimization strategies.

1.3 Overview

The required fundamentals of this thesis are explained in Chapter 2. These includes the fundamental basics of automata theory. Related work is discussed in Chapter 3. The detailed consideration of the influences on this work, such as existing systems to be considered, user requirements and environmental influences are analyzed in Chapter 4. Based on this analysis as well as related work, a design is presented in Chapter 5 that considers the posed requirements of this work. An evaluation of the developed approach is presented in Chapter 6. Here, the requirements set are reflected upon. Finally, a summary and outlook is presented in Chapter 7.

2 Fundamentals

In this chapter we aim to discuss and define the most important technologies and techniques fundamental to the intended design.

2.1 Definition of FPGAs

FPGAs, or Field Programmable Gate Arrays, are semiconductor devices built on a matrix of configurable logic blocks connected by programmed interconnects. The versatility of FPGAs is what makes them so popular. Unlike ASICs, which are application-specific, an FPGA can be configured for a specific function or design requirement [Xil].

From its invention, FPGAs were anticipated to have advantages over ASICs, and over time, they have indeed become increasingly competitive and advantageous in various applications. This transformation is partly attributed to Moore's law, which has enabled current FPGAs to offer significant processing power without incurring exorbitant costs [Ün21].

2.1.1 FPGA Component

1. **Configurable Logic Blocks (CLBs):** CLBs constitute the primary logic resource in FPGAs for implementing both combinational and sequential logic. A typical CLB comprises a configurable switch matrix, including elements such as a selection circuit (multiplexer), a flip-flop, and several inputs to facilitate diverse logic functions [AS].
2. **Switch Matrix:** Serving as the interconnection fabric, the switch matrix enables each CLB to tap into common wiring resources. This network is instrumental in routing signals among the CLBs and to the IOBs, which is a cornerstone for the FPGA's reconfigurability and the flexibility that is synonymous with its design [AS].
3. **Input/Output Blocks (IOBs):** As the FPGA's interface with the external environment, IOBs are crucial for establishing off-chip connections. They are strategically positioned around the FPGA's perimeter and include IO pads—integral parts of the IOBs that act as the physical conduit for signal exchange [AS].

Figure 1 illustrates a simplified FPGA architecture. The pink squares labeled 'CLB' denote the Configurable Logic Blocks, which are the principal units for logic computation, capable of performing a variety of logic functions through their internal switch matrices, multiplexers, and flip-flops. The purple lines interlinking the CLBs represent the switch matrix responsible for the flexible routing of signals within the FPGA. The 'I/O' blocks at the edges are the Input/Output Blocks that facilitate communication between the FPGA and external devices, allowing for signal transmission into and out of the chip.

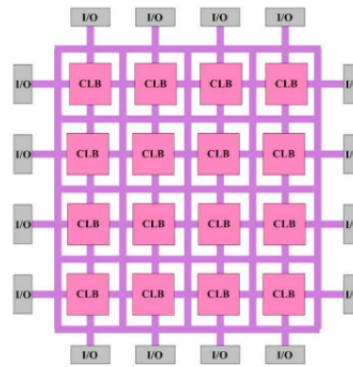


Figure 1: An FPGA Architecture Overview

Figure 2.1: FPGA Architecture

2.1.2 FPGA Development Process

FPGA programming is based on Hardware design, the designer utilizes Hardware description language in order to describe the required structure and behaviour of the design, some common steps are:

1. Concept Design: Designer describes the specification, functionality, and architecture of digital circuits intended to be implemented on FPGA.
2. HDL Design: At this level, the system is specified using hardware description languages like Verilog and VHDL. Hardware code that defines the logic and structural behaviour of digital circuits is implemented in this stage.
3. Simulation: Before the synthesis of the HDL code, simulations are run to ensure that the design meets the requirements and behaves as expected.
4. Synthesis: This step is not technology-dependent, and it involves taking HDL code and transforming it to logic gates.
5. Implementation: Once the design is synthesized, it's ready for implementation. This phase involves mapping the logic gates and interconnections onto the physical resources of the FPGA. It also includes tasks like placing and routing, where the tool determines the physical location of each gate and establishes the necessary connections.
6. Bitstream generation and programming FPGA: after completion of the implementation process, the bitstream is generated and the FPGA board is programmed.

Fig. 2.2 shows that after the Design entry (Concept and HDL design), the Design is synthesised, which brings us to the Implementation state, where translation Tech mapping and Place and route occur, and an a bit stream is formed to programme our device.

Having explored the definition and development process of FPGAs, it becomes evident that these semiconductor devices offer a unique blend of versatility and cost-effectiveness. Their ability to be configured for specific functions or design requirements, coupled with advancements driven by Moore's law, has positioned FPGAs as formidable contenders in the realm of Internet of Things. Now, let's delve into their pivotal role in the rapidly

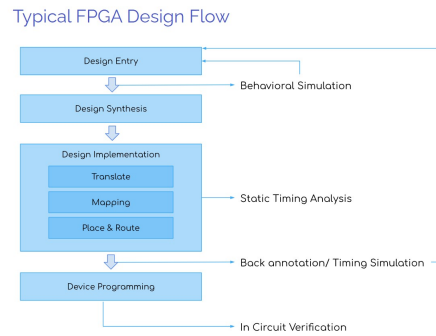


Figure 2.2: FPGA design Flow

evolving landscape of the Internet of Things (IoT), where their adaptability, processing capabilities, and other key attributes play a crucial part in shaping the future of connected devices and smart systems.

2.2 Operating Systems

An operating system is a computer programme implemented to control the computer hardware. An operating system assists in interfacing with the hardware of our computer system. It manages tasks and executes applications. An operating system contains many components and also controls our computer from system startup. Storage and input/output operations are also controlled by the operating system [GG05].

Ideally, a computer system contains hardware, CPU, memory, and I/O devices. Application programmes like Microsoft Office, browsers, and games all use the available resources in order to function. An operating system controls the distribution of these resources to complete tasks and processes. We can think of an operating system as a government that presides over computer systems and allocates resources for different tasks to ensure their success [GG05].

2.2.1 General Purpose Operating Systems(GPOS)

An operating system for general purposes, unlike real-time operating systems (RTOS), aids with hardware interface without taking specific real-time constraints into account.

A wide range of functionality, such as file systems, device drivers, networking capabilities, memory management, process management, and file systems, are often provided by these operating systems. Popular GPOS operating systems include Linux, macOS, and Microsoft Windows. In today's computing environment, GPOS are essential because they provide a user-friendly interface for computer interaction, manage hardware resources, and function as the basis for executing applications. GPOS improves the usability, scalability, and reliability of computing systems by abstracting away hardware complexity and offering a standardised environment for software development [Sta98].

2.2.2 Real Time Operating system

A real time operating system differs from a GPOS in the manner that the real time systems is deterministic, this means that the actions and outcomes are precisely predictable, given a specific initial state and a particular set of inputs, the system will always produce the same output and follow the same sequence of steps [SR04].

Real-Time Operating Systems (RTOS) are specialised operating systems that manage hardware resources, execute activities under stringent time restrictions, and provide predictable and deterministic behaviour. They are vital for applications that require timing, such as embedded systems in automotive, industrial automation, telecommunications, and aerospace.

2.2.3 Characteristics of RTOS

RTOS have several unique characteristics:

- **Deterministic Timing:** RTOS guarantee that tasks will be completed within a specified time frame allotted, known as the deadline.
- **Priority-Based Scheduling:** RTOS use priority-based task scheduling to ensure that high-priority tasks preempt lower-priority ones, ensuring critical tasks get CPU time when needed.
- **Minimal Latency:** RTOS minimize task switching and interrupt latency to ensure quick response times.
- **Resource Management:** Efficient management of system resources such as CPU, memory, and I/O to meet the stringent timing requirements of real-time applications.

2.2.4 Types of RTOS

RTOS can be categorized based on their design and implementation:

- **Hard RTOS:** Used in systems where missing a deadline could lead to catastrophic consequences (e.g., medical devices, industrial control systems).
- **Soft RTOS:** Used in systems where deadlines are important but not critical, and occasional delays can be tolerated (e.g., multimedia applications).

2.2.5 RTOS Architecture

RTOS architecture typically includes the following components:

- **Kernel:** The core part of the RTOS responsible for task scheduling, interrupt handling, and inter-task communication [LY03].
- **Task Management:** Mechanisms for creating, deleting, and managing tasks, including task states (e.g., ready, running, blocked) [LY03].

- **Inter-Process Communication (IPC):** Methods for tasks to communicate and synchronize, such as message queues, semaphores, and event flags [LY03].
- **Memory Management:** Efficient allocation and deallocation of memory to tasks while ensuring memory protection and isolation [LY03].

2.2.6 RTOS Examples

- **FreeRTOS:** An open-source RTOS widely used in embedded systems due to its simplicity and small footprint.
- **VxWorks:** A commercial RTOS known for its reliability and real-time performance, commonly used in aerospace and defense applications.
- **RTEMS (Real-Time Executive for Multiprocessor Systems):** An open-source RTOS designed for high-performance real-time applications.

3 Related Work

In this chapter, we review significant research efforts in integrating Linux with FPGA systems. These studies provide insights into challenges and solutions that inform our custom Linux distribution project for FPGA-based systems.

3.1 A Programmable Linux-Based FPGA Platform for Audio DSP

In this paper the writers present a programmable Linux-based FPGA platform specifically designed for Audio Digital Signal Processing (DSP) [CPF⁺23]. This platform leverages the flexibility of Linux and the reconfigurability of FPGA to create an environment conducive to real-time audio processing tasks. The authors highlight the use of a Xilinx FPGA, programmed using Vivado and Vitis tools, to facilitate the development of complex DSP algorithms. The platform's architecture is designed to allow seamless integration of custom IP cores with Linux, enabling the execution of audio processing algorithms directly on the FPGA fabric.

The system architecture presented is illustrated in Figure 3.1. The diagram depicts a workflow in which a Faust program is built with the Syfala compiler into numerous C++ files, which are then used to build the appropriate applications and IP cores for the FPGA. Specifically, the architecture employs the Xilinx ZYBO board, with audio processing operations conducted on a customised Audio IP core within the FPGA and control logic handled by an ARM processor running a Linux-based application. The platform is built using the Vitis, Vivado, and Faust/gcc toolchains. This design is relevant to our project because it illustrates an efficient technique for combining DSP algorithms with FPGA hardware in a Linux environment, using tools and methodologies that we plan to use.

The approach utilized by Cochard et al. is highly relevant to our work, particularly their method of integrating Linux with FPGA for real-time applications. We can replicate this by using the Xilinx Zynq platform and Vivado/Vitis toolchain to develop a Linux-based environment that supports dynamic audio processing tasks. Their emphasis on custom IP integration within the Linux environment aligns with our goal of creating a customizable Linux distribution for FPGA-based systems, where similar methods can be applied to other DSP applications beyond audio.

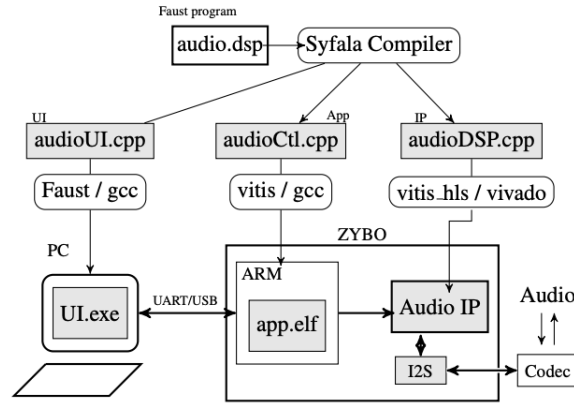


Figure 3.1: Audio DSP architecture

3.2 A Linux-based Support for Developing Real-Time Applications on Heterogeneous Platforms with Dynamic FPGA Reconfiguration

This paper explores the development of real-time applications on heterogeneous platforms through a Linux-based support system that enables dynamic FPGA reconfiguration [PBM⁺22]. Their research focusses on the integration of FPGAs with general-purpose processors (GPPs) to develop a versatile computing environment capable of adapting to changing computational demands. The authors used the Xilinx Zynq platform, which combines an ARM Cortex-A9 processor with FPGA fabric, and developed using Vivado and the Xilinx SDK.

This research is essential to our project since it proves the viability of managing real-time jobs in a Linux environment on an FPGA-based heterogeneous architecture. The dynamic reconfiguration capability outlined is especially useful for applications that require adjustable processing power, and we intend to integrate a similar architecture in our bespoke Linux version. The usage of the ARM Cortex-A9 processor in their work provides useful insights into processor selection and its impact on system performance, which helped us decide to employ a comparable CPU configuration to optimise our platform for real-time applications.

3.3 RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators

This paper introduces RIFFA 2.1, a reusable integration framework designed to facilitate the development of FPGA accelerators that can be easily integrated into general-purpose computing environments [JRHK15]. The framework abstracts the complexity of interfacing with FPGA hardware, enabling developers to focus on algorithm development. RIFFA 2.1 is compatible with a variety of FPGA platforms and can integrate with Linux-based systems, making it an adaptable tool for FPGA-based development.

The modular design of RIFFA 2.1 is consistent with our project’s goal of creating a flexible and extendable Linux distribution for FPGA-based systems. By adding RIFFA 2.1 concepts, we can improve our platform’s capacity to support a variety of FPGA accelerators, guaranteeing that it can handle a wide range of applications. The framework’s compatibility with Linux is very useful because it allows us to expedite the integration process and use existing Linux infrastructure to efficiently manage FPGA resources.

The architecture of RIFFA 2.1, as illustrated in Figure 5.1, showcases a comprehensive framework for integrating FPGA accelerators into general-purpose computing environments. The diagram highlights key components, including the RX and TX engines responsible for managing data flow between the FPGA and the host system via PCI Express. The use of scatter-gather techniques for efficient data handling and the reordering queues for managing non-posted requests ensure high-performance communication between the user IP cores and the host application. This design simplifies the process of deploying FPGA-based accelerators by providing a robust interface that abstracts the complexities of PCIe communication, which is crucial for our intended project.

RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators

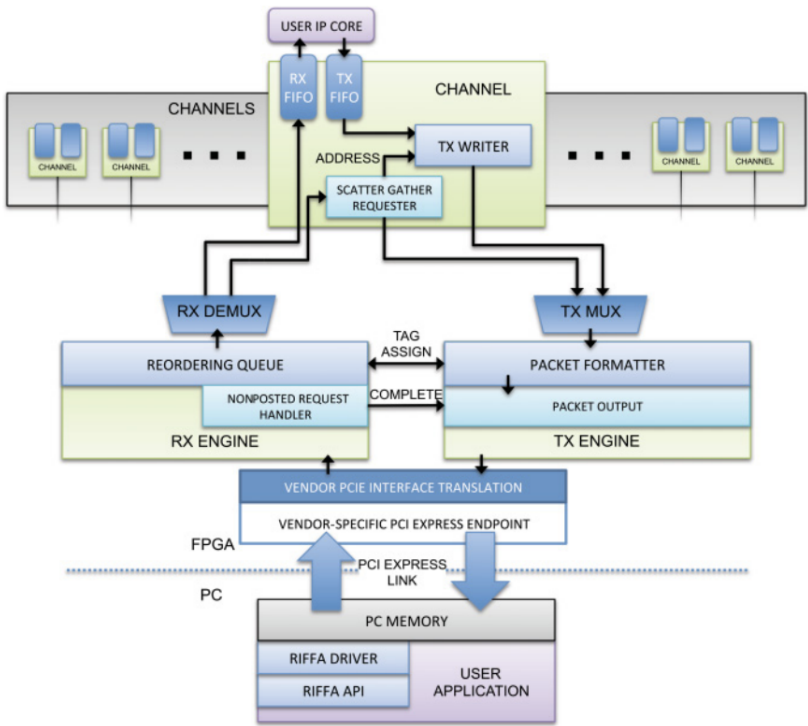


Fig. 3. RIFFA architecture.

3. ARCHITECTURE

Figure 3.2: RIFFA Architecture

3.4 Embedding Linux with Ability to Analyze Network Traffic on a Development Board Based on FPGA

This research paper discusses the implementation of a Linux based system on an FPGA development board with the capability to analyze network traffic [LCCM13]. Their work focuses on embedding Linux onto a Xilinx Spartan-6 FPGA, using a soft processor core to handle network traffic analysis tasks. The system is designed to monitor and analyze network packets in real-time, demonstrating the potential of FPGA-based systems in network security applications.

The methodology described provides a foundation for our work in embedding Linux onto FPGA platforms for security-related applications. Their use of a soft processor core within the FPGA fabric offers insights into optimizing processor selection and resource allocation for network traffic analysis. By replicating their approach, we aim to extend the capabilities of our custom Linux distribution to include real-time network monitoring and analysis, enhancing its utility in security applications.

4 Analysis

This analysis chapter builds on the comprehensive review undertaken in the Related Work chapter (see 3), which investigated a range of existing implementations relevant to this project work. These components are essential to the design and integration of linux on an FPGA for varied applications.

4.1 Workflow

To implement this project, a proper workflow is needed to ensure efficient and coordinated development among all parts. It is crucial to identify the different components of this project. Embedded development usually cuts across varied processes. Based on research conducted and previous experience in this field, the important parts needed to achieve Linux running on an FPGA are:

- **Development Environment:** The environment in which we build the required system is the first consideration. Ubuntu is the clear choice for me, partly because I completed a smaller version of this project using Ubuntu, gaining familiarity with its extensive FPGA development tools and robust command-line interface. Additionally, numerous online resources, methodologies, and even official Xilinx walkthroughs recommend using a Linux environment like Ubuntu. While it is possible to use the Windows operating system or the Windows Subsystem for Linux, using Ubuntu from the start avoids many potential issues that may occur on Windows.
- **Hardware Development:** This involves selecting the processor, determining the necessary GPIOs and input/output blocks, generating the bitstream, and flashing the board. This stage is crucial because the hardware characteristics significantly influence both the software and application development.
- **Software Development:** Our target software is a full-scale Linux operating system, so determining how to build and run it on our hardware is a major concern. There are various options, such as the Vitis SDK, which can automate most of the process, but it may not support every processor type. For CPUs like RISC-V, we would have to build Linux manually using tools like U-Boot and Buildroot. The CPU architecture chosen will greatly affect how Linux is constructed.
- **Benchmarking:** Benchmarking is essential for evaluating our system. Several benchmarks, such as Dhrystone and Whetstone, will be used to gain a comprehensive understanding of the system's performance and limitations.
- **Application:** The goal is to develop both complex and simple applications for our system. We want to test it with everyday computing tasks to evaluate how feasible

it is to use an FPGA for daily computing needs. Given its reconfigurable nature, a successful outcome would be quite significant.

5 Design and Implementation

This chapter covers the implementation of our project. After an extensive introduction to the project, its goals, and scope in Chapter 1, we moved on to the project fundamentals in Chapter 2. This chapter describes the necessary concepts and background understanding needed to fully grasp the project in all its vigor. Chapter 3 details various related research and projects that provide valuable information on our procedure and greatly influence our design and implementation approach. The preliminary stages of this research work are concluded by analysis in Chapter 4, where a description of the development environment and its selection criteria, the hardware development process, software development process, and benchmarking are highlighted in detail.

The preceding chapters bring us to this point, where we will now carry out the implementation of our project with all the acquired knowledge. It is important to explain the general development process we intend to employ.

The V-Model was employed as the guiding design framework for this project due to its emphasis on systematic development and validation. Each phase of the development process, from the definition of system requirements to detailed design and implementation, was directly aligned with the corresponding verification activities, such as module testing, system integration, and validation. This approach ensured that the custom Linux distribution, FPGA configuration, and security application were rigorously tested at every stage, enabling a robust and reliable final system that meets the defined project goals.

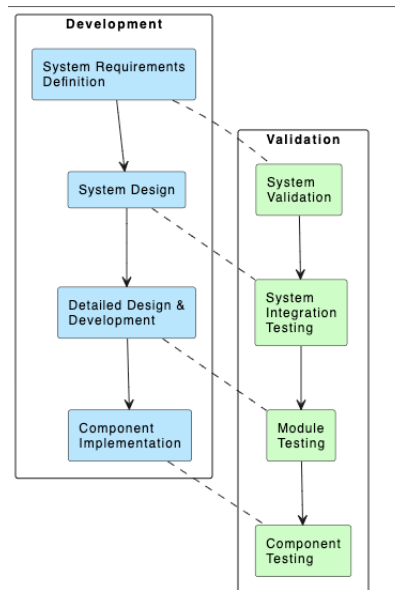


Figure 5.1: V model

5.1 Environment Setup

The environment setup is a critical aspect of this project, ensuring that the tools and configurations are optimized for developing and testing the FPGA-based system. This section outlines the steps undertaken to establish the development environment, heavily inspired by the *Vivado RISC-V* repository [Tar23].

5.1.1 Tools and Dependencies

The following tools and dependencies were used to set up the development environment:

- **Xilinx Vivado** (Version 22.1): Used for synthesizing and programming the FPGA hardware.
- **RISC-V GCC Toolchain**: Required for compiling software to run on the soft processor.
- **Buildroot**: Utilized for creating a custom Linux distribution tailored to the FPGA environment.
- **Python**: Employed for developing and testing the encryption and decryption application, machine learning, and other applications, chosen for easy adaptability with Linux.

In addition to these tools, the following commands were executed to install crucial dependencies and ensure the development environment was fully configured:

1. Install essential system utilities:

```
sudo apt-get install iproute2 make libncurses5-dev wget tar gzip
```

2. Set up development tools:

```
sudo apt-get install gcc-multilib build-essential automake g++ libtool flex bison x
```

3. Configure RISC-V toolchain support:

```
sudo apt-get install python3-pip python3-jinja2 python3-pexpect libssl-dev zlib1g-d
```

4. Install additional libraries for compatibility:

```
sudo apt-get install libncurses5-dev libgtk2.0-0:i386 libstdc++6:i386
```

These commands ensured that the environment was equipped to support FPGA hardware design, Linux kernel compilation, and application development. This setup forms the backbone of the development process, enabling seamless integration of the Vivado toolchain and the custom Linux distribution.

5.1.2 Cloning and Preparing the Repository

The *Vivado RISC-V* GitHub repository [Tar23] served as the foundation for this project. The repository was cloned, and the specific branch corresponding to the desired configuration was selected. The following steps were undertaken:

1. Clone the repository using the command:

```
git clone https://github.com/eugene-tarassov/vivado-risc-v.git
```

2. Checkout the appropriate branch that matches the FPGA hardware (e.g., `branch-name`).
3. Verify all dependencies and submodules were correctly initialized.

This is done by running the following commands:

```
sudo apt install git make
git clone https://github.com/eugene-tarassov/vivado-risc-v.git
cd vivado-risc-v
make apt-install
make update-submodules
```

5.2 Hardware Design

In order to implement this section with the utmost accuracy, several considerations needed to be made. First, it was necessary to evaluate the actual hardware that the design would be implemented upon. During the initial stages of project planning, the Genesys board was selected as the hardware platform. However, due to incompatibilities with our desired processor and workflow, we had to reconsider this choice. After further analysis, we opted for the more advanced Virtex VC707 board, as it better aligned with our project requirements.

The Virtex VC707 is a high-performance FPGA development board, equipped with the Xilinx Virtex-7 series FPGA. Key features of this board include:

- **High Logic Density:** The Virtex-7 FPGA offers significantly higher logic cell density compared to the Genesys board, enabling more complex designs and advanced processing capabilities.
- **High-Speed Interfaces:** It supports multiple high-speed I/O standards, including PCIe and DDR3, which are crucial for modern embedded applications.
- **Scalability:** The VC707 is designed to handle a wide range of applications, from digital signal processing to advanced system-on-chip (SoC) designs.
- **Versatile Connectivity:** The board features multiple connectivity options such as FMC (FPGA Mezzanine Card), Ethernet, and USB, enhancing its adaptability for diverse use cases.

Another key reason for selecting the Virtex VC707 is its compatibility with the project procedure and the *Vivado RISC-V* GitHub repository. Unlike the Genesys board, the VC707 is directly supported by the repository and provides the additional capability to implement more advanced processors. This alignment ensures seamless integration with the workflow while offering the flexibility to explore more complex configurations and applications.

The development process involves configuring the FPGA using the Vivado Design Suite, generating the bitstream, and integrating the soft processor core with the custom Linux

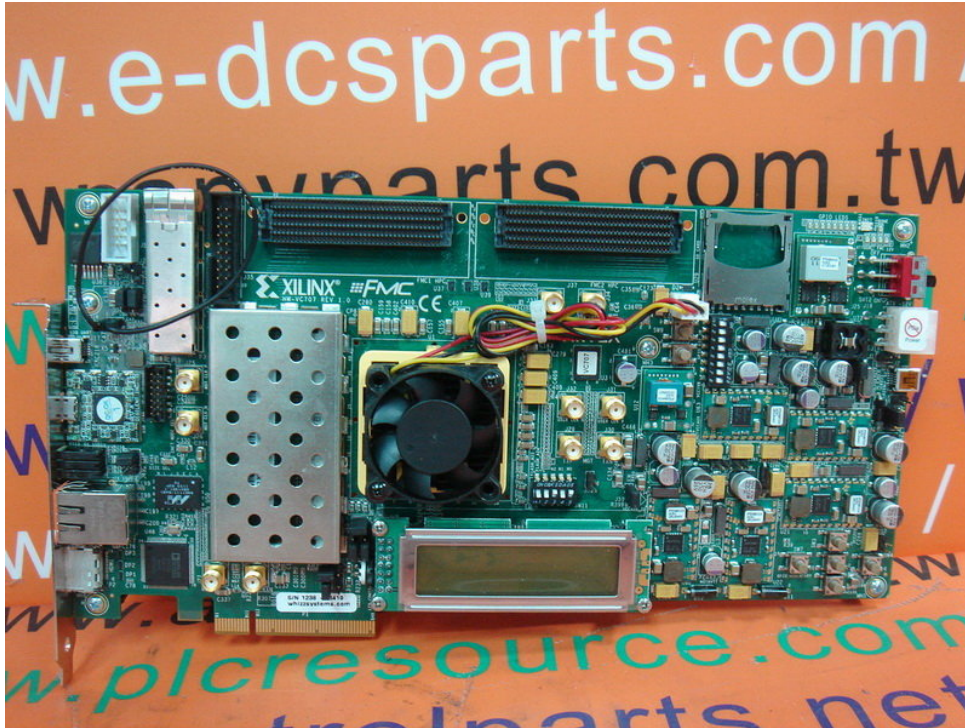


Figure 5.2: VIRTEX VC707

distribution. The flexibility and scalability of the chosen FPGA platform ensure that the system can handle both current project requirements and future enhancements.

5.2.1 Building Our Processor Core for the Target device-Virtex VC707

To implement the RISC-V processor core on the Virtex VC707, the following process was carried out:

5.2.1.1 Introduction to the Build Process

The process of building a processor core for the Virtex VC707 involves configuring the Vivado RISC-V repository to align with the board's hardware capabilities. This ensures optimal performance and compatibility with our project requirements. A single-core configuration was chosen for this project to balance resource utilization and performance.

5.2.1.2 Environment Preparation

Before building the processor core, the development environment was prepared:

```
source /opt/Xilinx/Vivado/2024.2/settings64.sh
```

This command ensures that the Xilinx Vivado environment is correctly sourced, providing access to all required tools and libraries.

```
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
sudo apt install python
Reading package lists... Done
Building dependency tree
Reading state information... Done
Note, selecting 'python-is-python2' instead of 'python'
python-is-python2 is already the newest version (2.7.17-4).
The following packages were automatically installed and are no longer required:
  chrntun-codex-ffmpeg-extra gstreamer1.0-vaapi
  libgstreamer-plugins-bad1.0-0 libva-wayland2
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
sh@xf86shra7-Voge-S11n-7-54ARE05:~/Vivado-rlsc-v$ make update-submodules
rm -rf workspace/patch-*.done
git submodule sync --recursive
Synchronizing submodule url for 'ethernet/verilog-ethernet'
Synchronizing submodule url for 'generators/gemini'
Synchronizing submodule url for 'generators/gemini/software/gemini-roc-tests'
Synchronizing submodule url for 'generators/gemini/software/libgemini'
Synchronizing submodule url for 'generators/gemini/software/onnxruntime-riscv'
Synchronizing submodule url for 'generators/riscv-boom'
Synchronizing submodule url for 'generators/sifive-cache'
Synchronizing submodule url for 'generators/testchipip'
Synchronizing submodule url for 'linux-stable'
Synchronizing submodule url for 'opencbi'
Synchronizing submodule url for 'rocket-chip'
Synchronizing submodule url for 'rocket-chip/cde'
Synchronizing submodule url for 'rocket-chip/hardfloat'
Synchronizing submodule url for 'rocket-chip/hardfloat/berkeley-softfloat-3'
Synchronizing submodule url for 'rocket-chip/hardfloat/berkeley-testfloat-3'
Synchronizing submodule url for 'rocket-chip/torture'
Synchronizing submodule url for 'u-boot'
git -c submodule.torture.update=none -c submodule.software/gemini-roc-tests.update=none -c submodule.software/onnxruntime-riscv.update=none submodule update --init --force --recursive
Submodule path 'ethernet/verilog-ethernet': checked out 'baac5f8d811d4385d5d9d69957975ead8bbed088'
Submodule path 'generators/gemini': checked out '799bc56b6dd859f2b1a9027a96a0b5be0ad7ed6'
Skipping submodule './.../generators/gemini/'
Submodule path 'generators/gemini/software/libgemini': checked out 'd873aa8bf39a01bca225044970745632816ce3d'
Submodule path 'generators/riscv-boom': checked out '18c48bb41ac3c64f6c65ef51db2e165d546679d'
Submodule path 'generators/sifive-cache': checked out 'dd1cafc0c7573a7056ef0946750af31a30a97c1'
Submodule path 'generators/testchipip': checked out '1952231509c939a935e47fazeef8108405d0130d'
Submodule path 'linux-stable': checked out '9c3a72fbc90d029f0161da4a4932c3cd4e0503f'
Submodule path 'opencbi': checked out 'a2b255b88918715173942f2c5e1f97ac9e90c877'
Submodule path 'rocket-chip': checked out 'dcb06a6e1c76d1129cb6d26494932a34c37185'
Skipping submodule './rocket-chip/'
Submodule path 'rocket-chip/cde': checked out '52768c97a27b254c0cc0ac9401feb55b29e18c28'
Submodule path 'rocket-chip/hardfloat': checked out 'd93aa570806013dea479a92ba9bb33d1f2d4f09f'
Submodule path 'rocket-chip/hardfloat/berkeley-softfloat-3': checked out '5c06db33f1e2130f67c045327b0ec949032df1d'
Submodule path 'rocket-chip/hardfloat/berkeley-testfloat-3': checked out '00b20075dd3c1a5d0d0d07a93043282832221612'
Submodule path 'u-boot': checked out 'd637294e264ed0eb29f390dfc393166f64d41b17'
sh@xf86shra7-Voge-S11n-7-54ARE05:~/Vivado-rlsc-v$
```

Figure 5.3: Command Line Interface for Building the processor

5.2.1.3 Processor Core Configuration

The Makefile provided in the Vivado RISC-V repository was used to specify the processor core configuration and the target board:

```
make CONFIG=rocket64b1 BOARD=vc707 bitstream
```

In this command:

- `CONFIG=rocket64b1`: Specifies a single 64-bit Linux-capable core.
- `BOARD=vc707`: Sets the target board to the Virtex VC707.

5.2.1.4 Build Process

The Makefile automates the process of generating the hardware description, synthesizing the design, and creating the FPGA bitstream. This streamlined workflow ensured that the processor core was correctly implemented on the VC707 platform. While running this command our build was completed on the first try and the hardware description files needed to program the FPGA were successfully generated although we performed all operations through the command line interface and were able to load our design in the Vivado GUI using this command:

```
source d:/vivado-risc-v/workspace/rocket64b2/system-vc707.tcl
```


5.2.2 Validation of Design

This section highlights images of the designed hardware system to show the details of our now built hardware platform.

1. Block Design (RocketChip Integration)

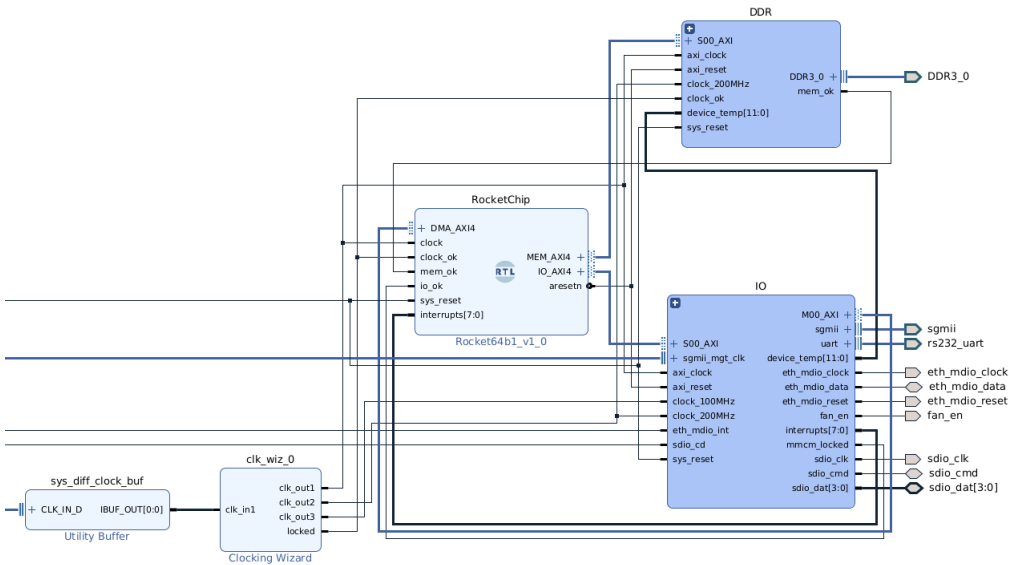


Figure 5.4: RocketChip Integration in the Hardware Platform

The block design shows the integration of a RocketChip processor with external DDR memory and various IO interfaces. The key components include the RocketChip as the central processing block connected to memory and IO through AXI interfaces, DDR memory for high-speed access, and a Cloning Wizard for distributing system clocks. This design demonstrates a complete hardware architecture essential for stable SoC operation.

2. Design Runs Results

Design Runs									
Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	
✓ synth_1 (active)	constrs_1	synth_design Complete!							
✓ impl_1	constrs_1	write_bitstream Complete!	0.133	0.000	0.045	0.000	4.373	0.000	
Out-of-Context Module Runs									
> ✓ riscv		Submodule Runs Complete							

Figure 5.5: Design Runs Result

This figure captures the synthesis and implementation results from the Vivado tool. Successful synthesis and implementation without timing violations confirm a robust design. Key metrics such as WNS (Worst Negative Slack) of 0.133 ns and TPWS (Total Positive Slack) of 0 ns indicate that timing closure was achieved, ensuring operational reliability at the target frequencies.

3. IO Block Design (Ethernet and AXI Integration)

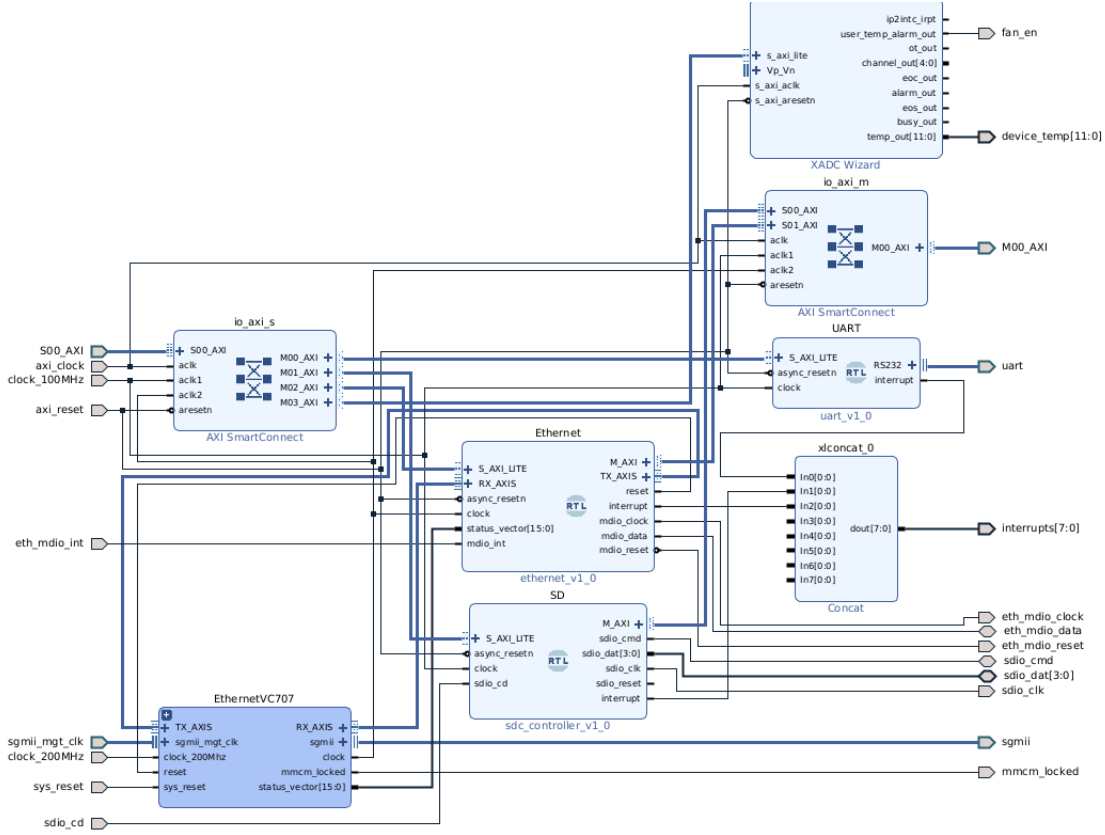


Figure 5.6: IO Integration with Ethernet and UART

This image focuses on the integration of various communication interfaces, including Ethernet and UART, through AXI interconnects. Key components include an AXI Smart Connect for efficient interconnections, an Ethernet core for high-speed communication, an XADC Wizard for system parameter monitoring, and a UART interface for debugging. This design ensures comprehensive communication capabilities and scalability for the hardware platform.

5.3 Software Design

Upon completion of the hardware design phase of the project, the focus now shifts toward building software to run on top of this hardware platform. This will enable us to test more advanced applications and programs.

Once again, the repository by [Tar23] proved to be highly useful for this purpose. It includes a Makefile that utilizes Buildroot and several other technologies, allowing us to load an SD card with a full-scale Debian Linux distribution. This setup is ideal for testing the limits of our system. The SD card is inserted into the SD card reader on our FPGA, marking the beginning of the process.

The Linux build process took approximately one hour and a few minutes to complete. I estimate that a system with greater RAM capacity, such as 32 GB, could reduce this process to about 15 minutes.

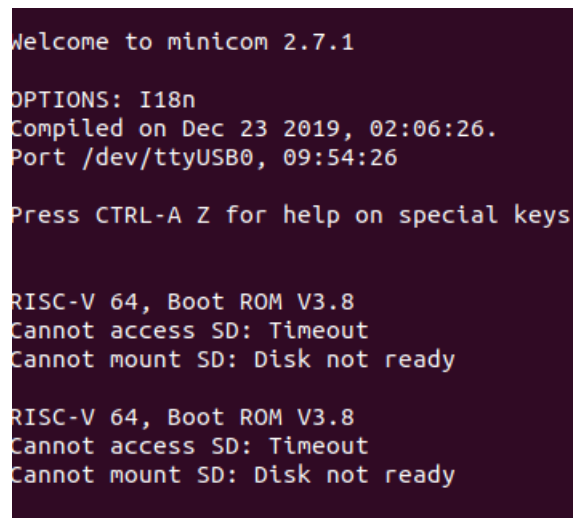
5.3.1 Linux Test

After completing the hardware setup and validation, the next step involves testing the software environment on the FPGA. This section describes the process of connecting to the board via UART using Minicom, troubleshooting SD card issues, booting Linux, and successfully logging into the system.

1. Connecting to the Board via UART

To interact with the FPGA board, we connected to its UART interface using Minicom, a terminal emulator for serial communication. Upon establishing the connection, the system attempted to boot and provided console feedback, but no Linux environment was initially available.

2. Troubleshooting SD Card Issues

A screenshot of a Minicom terminal window with a black background and white text. The text shows the Minicom welcome message, options, and compilation details. It then displays two identical error messages: 'RISC-V 64, Boot ROM V3.8', 'Cannot access SD: Timeout', and 'Cannot mount SD: Disk not ready'.

```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Dec 23 2019, 02:06:26.
Port /dev/ttyUSB0, 09:54:26

Press CTRL-A Z for help on special keys

RISC-V 64, Boot ROM V3.8
Cannot access SD: Timeout
Cannot mount SD: Disk not ready

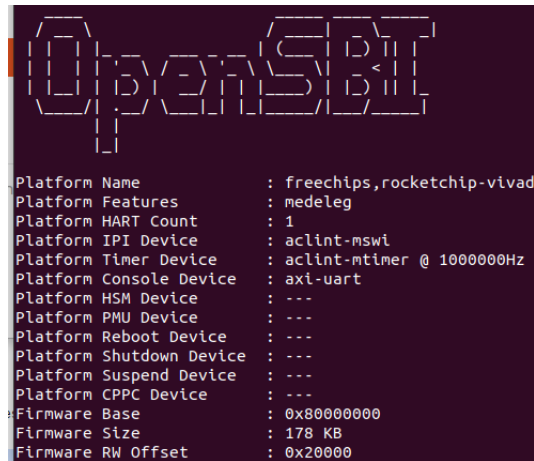
RISC-V 64, Boot ROM V3.8
Cannot access SD: Timeout
Cannot mount SD: Disk not ready
```

Figure 5.7: Error message indicating the failure to mount the SD card due to a timeout.

During the initial testing phase, the system displayed an error message indicating that it failed to mount the SD card due to a timeout. This error typically occurs when the SD card is either missing or lacks a valid Linux image. As a result, the system was restricted to running bare-metal programs.

After inserting an SD card containing a properly prepared Linux image, the boot process continued without issues.

3. Role of OpenSBI in Booting Linux



```

OpenSBI

Platform Name      : freechips,rocketchip-vivado
Platform Features  : medeleg
Platform HART Count : 1
Platform IPI Device : acliint-mswi
Platform Timer Device : acliint-mtimer @ 1000000Hz
Platform Console Device : axi-uart
Platform HSM Device : ---
Platform PMU Device : ---
Platform Reboot Device : ---
Platform Shutdown Device : ---
Platform Suspend Device : ---
Platform CPPC Device : ---
Firmware Base      : 0x80000000
Firmware Size      : 178 KB
Firmware RW Offset : 0x20000
  
```

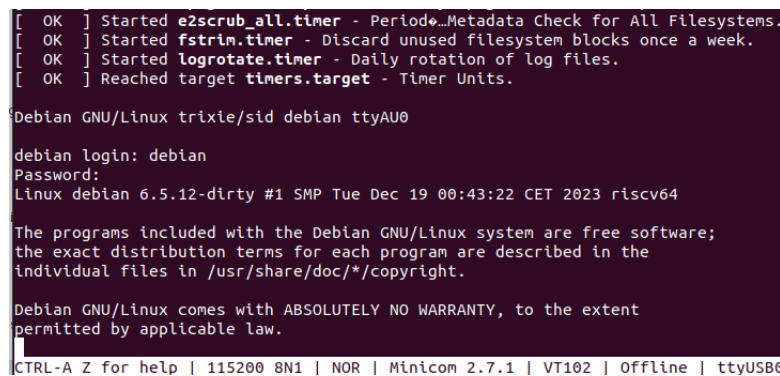
Figure 5.8: OpenSBI output showing platform details, including firmware base address, size, and hardware initialization status.

OpenSBI (Open Supervisor Binary Interface) plays a critical role in the boot process by initializing the hardware and preparing the system for loading the operating system. Upon successful SD card insertion, OpenSBI initialized the platform and displayed key information, including:

- **Platform Name:** freechips,rocketchip-vivado
- **Platform HART Count:** 1 (indicating a single-core processor)
- **Firmware Base:** 0x80000000
- **Firmware Size:** 178 KB

OpenSBI successfully initialized the hardware and passed control to the Linux kernel.

4. Successful Linux Execution on FPGA and Login



```

[ OK ] Started e2scrub_all.timer - Periodic Metadata Check for All Filesystems.
[ OK ] Started fstrim.timer - Discard unused filesystem blocks once a week.
[ OK ] Started logrotate.timer - Daily rotation of log files.
[ OK ] Reached target timers.target - Timer Units.

Debian GNU/Linux trixie/sid debian ttyAUG0

debian login: debian
Password:
Linux debian 6.5.12-dirty #1 SMP Tue Dec 19 00:43:22 CET 2023 riscv64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Offline | ttyUSB0
  
```

Figure 5.9: Login screen after the successful boot process. The default username and password are both set to `debian`.

After resolving the SD card issue and ensuring proper initialization via OpenSBI, the system successfully booted into the Debian Linux environment. The default login credentials (`debian` for both username and password) allowed access to the system. This confirmed that the custom Linux distribution was fully functional on the FPGA, running on the RISC-V processor.

5.4 Application and System Validation Tests

To demonstrate the operational capability of our custom RISC-V Linux environment on the FPGA, various application tests were conducted. These tests validate the system's performance and ensure that the hardware and software components are functioning as intended.

5.4.1 Guess the Number Game on FPGA Terminal

This test involved creating and running an interactive "Guess the Number" game on the FPGA, using Python to handle the game logic and provide terminal-based interaction. The purpose was to demonstrate Python's capabilities for interactive applications on the FPGA's Debian Linux environment.

5.4.1.1 System Setup and Access via SSH

The FPGA system was accessed remotely from a macOS workstation using Secure Shell (SSH). The following command was executed to connect to the FPGA:

```
ssh debian@<ip_address>
```

Upon entering the password, successful remote access was established, and the Debian shell environment was displayed.

5.4.1.2 Game Code Implementation

The "Guess the Number" game was created by writing a Python script. The following steps were performed to create and run the game:

```
touch guess_the_number.py
nano guess_the_number.py
```

The code was written in the `guess_the_number.py` file, implementing the random number generation and feedback mechanism. Once the code was saved, the game was run using:

```
python3 guess_the_number.py
```

5.4.1.3 Game Interaction and Execution

The game prompts the user to guess a random number between 1 and 100. The system provides feedback on whether the guess is too low, too high, or correct. Below is a screenshot of the terminal output during the game:

```
debian@debian:~$ touch guess_the_number.py
debian@debian:~$ nano guess_the_number.py
debian@debian:~$ python3 guess_the_number.py
Welcome to Guess the Number!
I am thinking of a number between 1 and 100. Can you guess it?
Enter your guess: 55
Too low! Try again.
Enter your guess: 79
Too high! Try again.
Enter your guess: 60
Too low! Try again.
Enter your guess: 65
Too low! Try again.
Enter your guess: 70
Too high! Try again.
Enter your guess: 69
Congratulations! You've guessed the number in 6 attempts.
debian@debian:~$
```

Figure 5.10: Terminal output showing the interaction with the "Guess the Number" game.

5.4.1.4 Conclusion

The successful execution of the game confirmed the following:

- The Python script correctly implemented the logic for generating a random number and providing feedback on guesses.
- The game ran smoothly on the FPGA, demonstrating Python's capability for interactive terminal applications.
- The user was able to successfully guess the number within a reasonable number of attempts.

This interactive game provided a simple yet effective demonstration of the FPGA's computational and user-interaction capabilities.

5.4.2 AES Encryption/Decryption Application

This section documents the development and functionality of an interactive AES encryption and decryption application. The tool is implemented in Python using built-in modules and simple cryptographic techniques. It leverages a passphrase-derived key (using SHA-256) to perform AES-like block-wise XOR encryption with PKCS#7 padding. In addition, the application provides dynamic functionality for storing encrypted messages in a JSON file, listing stored messages, and later decrypting them using the correct passphrase.

5.4.2.1 Overview and Implementation

The application offers the following features:

- **Encrypt New Text:** The user enters a text message along with a passphrase. A 16-byte key is derived from the passphrase using SHA-256, and the text is encrypted using block-wise XOR operations. The encrypted message is then stored (as a hexadecimal string) along with a unique ID and timestamp.
- **Decrypt a Stored Message:** The user can retrieve and decrypt any previously stored message by providing its unique ID and the corresponding passphrase.

- **List Stored Messages:** Displays all stored messages with their IDs and timestamps.

The source code implements these functionalities and presents a menu-driven interface for an interactive user experience. The application does not require any additional package installations on Debian, as it uses only Python's built-in modules.

5.4.2.2 Procedure

The following steps outline how the application is used:

1. **Running the Application:** The application is executed via the command:

```
python3 aes_interactive.py
```

2. **Encrypting a Message:** The user selects option 1 from the menu, then inputs the text to encrypt and a passphrase. The application derives a 16-byte key from the passphrase, encrypts the text, and stores the result along with a unique ID and timestamp.
3. **Decrypting a Message:** To decrypt a message, the user selects option 2 and provides the unique ID of the stored message and the correct passphrase. The application then retrieves the message, decrypts it using the derived key, and displays the original text.
4. **Listing Stored Messages:** Option 3 displays all stored messages along with their IDs and timestamps, allowing the user to identify messages for later decryption.

5.4.2.3 Demonstration of Results

Below is a sample run of the application. The output demonstrates encrypting a message, storing it, and then retrieving and decrypting it.

```
=== AES Encryption/Decryption Menu ===
1. Encrypt new text
2. Decrypt a stored message
3. List stored messages
4. Quit
Enter your choice (1-4): 1
Enter text to encrypt: Fantastic 4
Enter a passphrase for encryption: 6214
Encrypted data (hex): ff35d23eabde6f932048ab6045222779
Message stored with ID: 1738511713

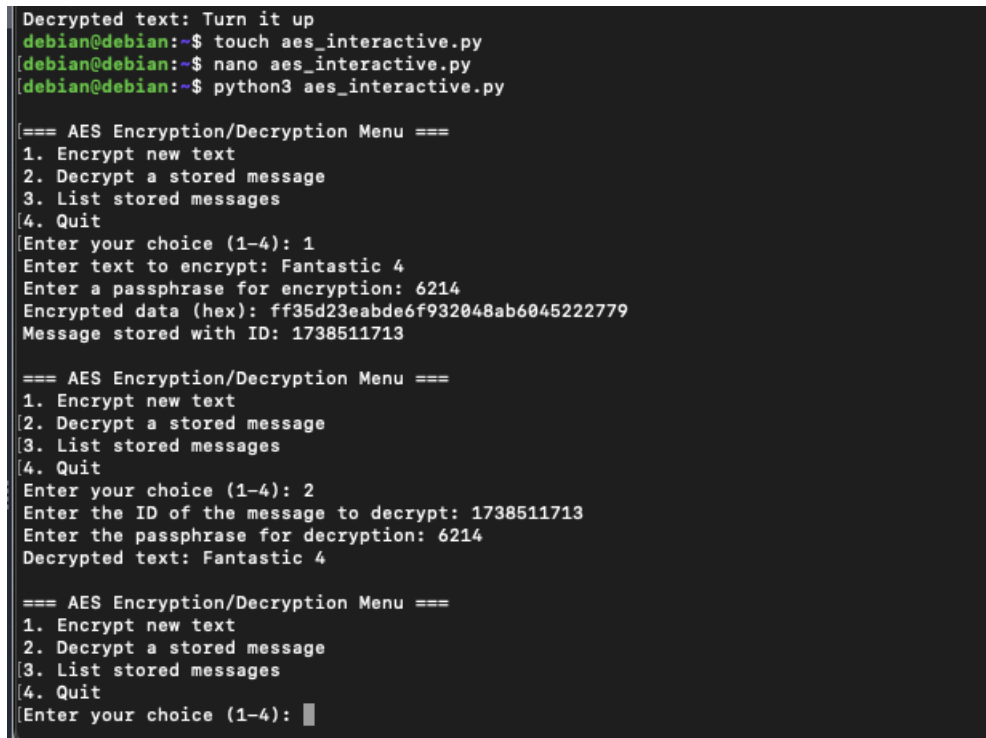
=== AES Encryption/Decryption Menu ===
1. Encrypt new text
2. Decrypt a stored message
3. List stored messages
4. Quit
Enter your choice (1-4): 2
```

Enter the ID of the message to decrypt: 1738511713
Enter the passphrase for decryption: 6214
Decrypted text: Fantastic 4

=== AES Encryption/Decryption Menu ===

1. Encrypt new text
2. Decrypt a stored message
3. List stored messages
4. Quit

Enter your choice (1-4):



```
Decrypted text: Turn it up
debian@debian:~$ touch aes_interactive.py
debian@debian:~$ nano aes_interactive.py
debian@debian:~$ python3 aes_interactive.py

=== AES Encryption/Decryption Menu ===
1. Encrypt new text
2. Decrypt a stored message
3. List stored messages
4. Quit
Enter your choice (1-4): 1
Enter text to encrypt: Fantastic 4
Enter a passphrase for encryption: 6214
Encrypted data (hex): ff35d23eabde6f932048ab6045222779
Message stored with ID: 1738511713

=== AES Encryption/Decryption Menu ===
1. Encrypt new text
2. Decrypt a stored message
3. List stored messages
4. Quit
Enter your choice (1-4): 2
Enter the ID of the message to decrypt: 1738511713
Enter the passphrase for decryption: 6214
Decrypted text: Fantastic 4

=== AES Encryption/Decryption Menu ===
1. Encrypt new text
2. Decrypt a stored message
3. List stored messages
4. Quit
Enter your choice (1-4): █
```

Figure 5.11: Screenshot of the AES Encryption/Decryption Application in Action

5.4.3 Conclusion

The interactive AES encryption/decryption application demonstrates how a secure messaging tool can be implemented using built-in Python libraries. The key derivation from a user-supplied passphrase ensures that the encryption is personalized and secure. This tool provides a dynamic interface for encrypting, storing, and later decrypting messages, making it a robust solution for applications requiring lightweight cryptographic operations on a custom Linux distribution.

5.4.4 Network Application Testing via SSH on RISC-V Linux Environment

This test aimed to validate the network connectivity and computational capability of the system by running a Python-based application involving HTTP requests.

5.4.4.1 System Setup and Access via SSH

The RISC-V Linux environment was accessed from a macOS workstation using Secure Shell (SSH). The following command was executed from the terminal to establish the connection:

```
ssh debian@<ip_address>
```

Upon entering the password, the terminal displayed the Debian environment running on the FPGA, confirming successful remote access over Ethernet.

5.4.4.2 Dependency Installation

Before running the network application, we installed the necessary dependencies. Given the managed Python environment in Debian, the ‘python3-requests’ package was installed using the system package manager:

```
sudo apt update
sudo apt install python3-requests -y
```

5.4.4.3 Network Test Application

The following Python script was created on the FPGA using the ‘nano’ editor:

```
nano network_test.py
```

The script content is as follows:

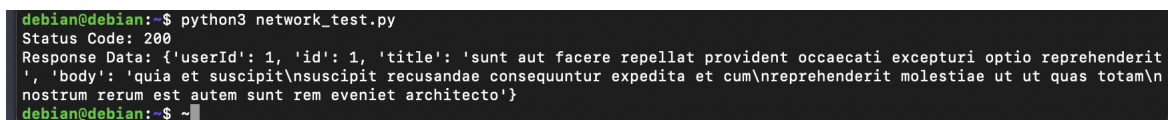
```
import requests

url = "https://jsonplaceholder.typicode.com/posts/1"
try:
    response = requests.get(url)
    print("Status Code:", response.status_code)
    print("Response Data:", response.json())
except Exception as e:
    print(f"Network request failed: {e}")
```

5.4.4.4 Execution and Results

The script was executed using the following command:

```
python3 network_test.py
```



```
debian@debian:~$ python3 network_test.py
Status Code: 200
Response Data: {'userId': 1, 'id': 1, 'title': 'sunt aut facere repellat provident occaecati excepturi optio reprehenderit', 'body': 'quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto'}
debian@debian:~$
```

Figure 5.12: Output of the network test application, demonstrating successful HTTP request execution.

This successful execution verified the following:

- The Ethernet connection was functional and stable, enabling communication with external servers.
- Python applications involving HTTP requests were executed without issues.
- Dependencies were correctly installed and operational.

5.4.4.5 Conclusion

The successful completion of this test confirmed that the custom Linux environment on the RISC-V processor could handle real-world network operations. This capability demonstrates the platform's readiness for application development involving network interactions, thereby validating the integrity and performance of the Ethernet setup.

5.4.5 FPGA as a Web Server for Project Dashboard

This test aimed to deploy and access a custom webpage hosted on the FPGA running the Debian Linux environment. The goal was to demonstrate the use of the FPGA as a web server, presenting project information and enabling future interactions via the web interface.

5.4.5.1 System Setup and Access via SSH

The system was accessed remotely from a macOS workstation using Secure Shell (SSH). The following command was executed to connect to the FPGA:

```
ssh debian@<ip_address>
```

Upon entering the password, successful remote access was established, confirmed by the display of the Debian shell environment.

5.4.5.2 Web Server Configuration

A Python-based web server script, `web_server.py`, was deployed on the FPGA. The script listens for HTTP requests and serves static web pages, including the project's dashboard. The server was started using the following command:

```
python3 web_server.py
```

5.4.5.3 Webpage Creation and Content

The following HTML code was created using the `nano` editor to serve as the project's main webpage:

```
nano index.html
```

The key features of the webpage include:

- Project Overview: Describes the FPGA configuration, operating system, and objectives.

- Technical Specifications: Details on the RISC-V architecture and development environment.
- Modular Sections: Space for future system monitoring and data visualization.

The HTML content for the page is presented below for reference:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>FPGA Project Dashboard</title>
  <style>
    /* Styling for an intuitive interface */
    body { background-color: #eef4fb; font-family: Arial, sans-serif; }
    h1 { color: #1e90ff; }
  </style>
</head>
<body>
  <header>
    <h1>FPGA Project Dashboard</h1>
  </header>
  <main>
    <section>
      <h2>System Overview</h2>
      <p>The Virtex VC707 FPGA runs a custom RISC-V processor with Debian Lin
    </section>
  </main>
</body>
</html>
```

5.4.5.4 Accessing the Webpage from a Browser

After launching the web server on the FPGA, the webpage was accessed by navigating to the following URL on the browser:

`http://<fpga_ip_address>:8000`

5.4.5.5 Execution and Results

```
[debian@debian:~$ python3 web_server.py
Serving HTTP on port 8000...
192.168.178.86 - - [02/Feb/2025 11:33:55] "GET / HTTP/1.1" 200 -
192.168.178.86 - - [02/Feb/2025 11:43:23] "GET / HTTP/1.1" 304 -
192.168.178.86 - - [02/Feb/2025 11:43:23] "GET / HTTP/1.1" 304 -
```

Figure 5.13: Terminal output showing the web server running and serving the index.html file.

The successful browser-based display of the webpage is shown in Figure 5.14.



Figure 5.14: Browser output showing the project's main dashboard served by the FPGA web server.

This confirmed the following:

- The FPGA successfully hosted and served the custom webpage.

- The web server operated correctly, handling incoming HTTP requests without errors.
- Access to the webpage from a remote browser was functional and stable.

5.4.5.6 Conclusion

This successful deployment demonstrated the capability of the FPGA as a functional web server. The project dashboard served by the web server provided a clear and modular platform for future expansion, including data visualization and system monitoring. Additionally, the seamless browser-based access validated the system's suitability for remote operations and user interactions.

6 Evaluation

In this chapter, our goal is to employ well-known benchmarks to evaluate the developed system. A brief overview of the project, as well as the process of executing research and designing it, will also be discussed.

6.1 CoreMark Benchmark of Our Platform

CoreMark is a widely used benchmark designed to measure the performance of microprocessor cores. It provides a simple and effective way to evaluate the CPU's performance in tasks that are representative of real-world workloads, including list processing, matrix manipulation, and state machine management. CoreMark tests the core's ability to handle a variety of operations such as integer and floating-point arithmetic, control flow, and memory handling. The benchmark is essential because it provides a lightweight, portable means of evaluating the raw performance of processors in terms of iterations per second, which is critical when assessing the capabilities of embedded systems or custom FPGA platforms.

In this study, we utilized the CoreMark benchmark to evaluate the performance of our platform, which is based on a Field Programmable Gate Array (FPGA). We ran the benchmark using the 'make' utility to compile and load the application, after transferring the necessary files to the FPGA using SCP (Secure Copy Protocol). This section describes the process of running the CoreMark benchmark on our FPGA and presents the results obtained from the benchmark tests.

6.1.1 Transferring CoreMark to the FPGA

To execute the CoreMark benchmark on the FPGA, we first had to transfer the benchmark files to the FPGA. This was done using the SCP command, which securely copies files from our local machine to the FPGA system. The relevant files, including the source code and executables, were placed in the appropriate directories on the FPGA. Below is an example of the SCP command used for this transfer:

```
scp -r /path/to/coremark/ debian@<FPGA_IP>:/home/debian/coremark/
```

After the files were successfully transferred, we moved on to compiling and running the benchmark.

6.1.2 Compiling and Running CoreMark with Make

Once the CoreMark files were transferred to the FPGA, we used the ‘make’ tool to compile and run the benchmark. The ‘make’ utility simplifies the process of building projects by automatically handling dependencies and compilation. The command below was used to build and run the benchmark in performance mode:

```
make XCFLAGS=" -DPERFORMANCE_RUN=1" load run1.log
```

This command triggers the compilation process, loads the benchmark onto the FPGA, and starts running it. It also logs the results of the run in the ‘run1.log’ file. A similar command was used for the validation run:

```
make XCFLAGS=" -DVALIDATION_RUN=1" load run2.log
```

After execution, the results were logged in the respective files (‘run1.log’ and ‘run2.log’).

6.1.3 Benchmark Results and Analysis

The results of the two benchmark runs are presented below.

6.1.3.1 Performance Run Results (run1.log)

From the performance run, we obtained the following data:

```
CoreMark Size      : 666
Total ticks        : 14228
Total time (secs): 14.228000
Iterations/Sec     : 210.851841
Iterations         : 3000
Compiler version   : GCC13.2.0
Compiler flags     : -O2 -DPERFORMANCE_RUN=1 -lrt
Memory location    : Heap
seedcrc           : 0xe9f5
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0xcc42
Correct operation validated.
CoreMark 1.0 : 210.851841 / GCC13.2.0 -O2 -DPERFORMANCE_RUN=1 -lrt / Heap
```

This output indicates the core’s performance during the benchmark, showing a throughput of approximately 210.85 iterations per second. The benchmark ran for 14.23 seconds, completing 3000 iterations. The results also include CRC checksums to ensure the correctness of the operation.

6.1.3.2 Validation Run Results (run2.log)

The validation run yielded similar results:

```
CoreMark Size      : 666
Total ticks        : 14307
Total time (secs): 14.307000
Iterations/Sec     : 209.687566
Iterations         : 3000
Compiler version   : GCC13.2.0
Compiler flags     : -O2 -DPERFORMANCE_RUN=1 -lrt
Memory location    : Heap
seedcrc           : 0x18f2
[0]crclist        : 0xe3c1
[0]crcmatrix      : 0x0747
[0]crcstate       : 0x8d84
[0]crcfinal       : 0x2717
Correct operation validated.
CoreMark 1.0 : 209.687566 / GCC13.2.0 -O2 -DPERFORMANCE_RUN=1 -lrt / Heap
```

The validation run showed a throughput of 209.69 iterations per second, with a total execution time of 14.31 seconds. Although the performance and validation runs were slightly different, the results were close, indicating consistent behavior across both tests.

6.1.4 Conclusion

The CoreMark benchmark provides a valuable insight into the raw processing power of our FPGA-based platform. The performance and validation runs both showed iteration rates exceeding 200 per second, which is suitable for many embedded system tasks. While there were small differences between the performance and validation runs, these are expected due to variations in system conditions during execution.

By using SCP to transfer files and ‘make’ to compile and execute the benchmark, we were able to easily test the performance of our platform on the FPGA. This process also validated the correctness of the benchmark with CRC checksums, ensuring the results were accurate. These results form a baseline for further optimizations or comparisons with other platforms in future evaluations.

6.2 Linpack Benchmark of Our Platform

The *Linpack* benchmark is a widely-used performance test that focuses on solving systems of linear equations. It is often employed to assess the floating-point computational capabilities of processors, as it measures the number of floating-point operations per second (FLOP/s) a system can perform. The benchmark provides a good approximation of performance for applications that involve large matrix computations, which are common in scientific computing and simulations.

We used the Linpack benchmark to evaluate the computational performance of our platform. The benchmark is written in C, and its primary task involves solving a set of linear algebra equations using double precision floating-point operations. The program measures both the time taken to solve the system and the resulting performance in terms of kFLOPS (thousands of floating-point operations per second).

6.2.1 Procedure to Run Linpack

To run the Linpack benchmark on our platform, the following steps were executed:

1. **Clone or Navigate to the Linpack Directory:** The first step is to navigate to the directory where Linpack is located. In our case, this directory is `~/linpack`:

```
cd ~/linpack
```

2. **Build the Benchmark:** We checked if the system was ready for running Linpack by ensuring that the necessary components were in place. The build process was initiated using the `make` command:

```
make
```

The make system did not need any further adjustments, and the build process completed successfully, showing the message "Nothing to be done for 'all'."

3. **Run the Benchmark:** After building the executable, the benchmark was run with the following command:

```
./linpack
```

This executed the Linpack benchmark, which carried out a series of linear algebra computations and produced the results.

6.2.2 Results

The benchmark produced the following results, which are based on solving a matrix of size 200x200 using double precision:

Memory required: 315K.

Machine precision: 15 digits.

Array size: 200 X 200.

The benchmark executed in multiple repetitions with varying levels of work, and the output presented is summarized in the following table:

Reps	Time (s)	DGEFA (%)	DGESL (%)	Overhead (%)	KFLOPS
4	0.70	89.62	2.40	7.99	8579.26
8	1.39	89.58	2.43	7.99	8581.89
16	2.78	89.51	2.58	7.91	8573.90
32	5.57	89.45	2.50	8.04	8585.04
64	11.12	89.55	2.50	7.95	8583.52

Table 6.1: Linpack Benchmark Results (200x200 Array)

The key output metrics are as follows:

- **Reps:** Number of repetitions for each execution cycle.
- **Time (s):** Total time taken in seconds for the benchmark to run.
- **DGEFA:** The percentage of time spent in the LU decomposition (factorization) phase.
- **DGESL:** The percentage of time spent solving the system of equations.
- **Overhead:** The percentage of time spent on operations unrelated to the main computations (e.g., setup).
- **KFLOPS:** The performance measured in thousands of floating-point operations per second.

6.2.3 Performance Analysis

From the results above, we can observe the following trends:

- The **KFLOPS** values remain relatively stable across the different repetition counts, ranging from approximately 8573 to 8585 kFLOPS. This suggests that the platform's performance remains consistent with increasing repetitions.
- The **time** required for each run grows linearly with the number of repetitions, as expected.
- The **DGEFA** and **DGESL** percentages are also quite stable, indicating that the system is well-optimized for both matrix factorization and solving operations.

This performance is satisfactory, and we can confidently conclude that the platform performs well for matrix-based computations, which are critical in scientific applications.

6.2.4 Conclusion

In this section, we demonstrated the process of setting up and running the Linpack benchmark on our platform. The benchmark results provide valuable insight into the floating-point processing capabilities of our system. The stable performance across different runs confirms that the system can handle matrix operations efficiently, making it suitable for tasks involving scientific computing and simulations.

7 Outlook and Summary

This project was both a challenging and captivating experience for me, representing a deeper dive into embedded engineering and development—a concept I became familiar with during my Special Emphasis B course titled Advanced Embedded Engineering. The initial stages involved extensive research and trial and error. Defining the scope of the project was indeed challenging, but it became a challenge I was ultimately grateful for.

In-depth research, particularly when selecting the platform and its components, is something I look back on fondly. Various processor architectures, such as ARM, were considered, but RISC-V was chosen due to its compatibility with the project requirements. The cornerstone of my successful implementation was the Xilinx GitHub repository by Tarassov [Tar23], which provided the necessary toolset to build the underlying hardware architecture. This resource also enabled further evaluation of the architecture and allowed me to experiment with interesting new applications.

Based on my implementation and the evaluation results from various benchmarks, as well as the execution of multiple real-world computing use cases—including a web server, game application, and encryption/decryption tasks—I believe we have established that RISC-V and the FPGA can successfully prototype a robust computing system. Due to the modularity of RISC-V, it can lead to more specialized computing devices tailored to specific requirements.

For instance, the RISC-V processor can be built with varying levels of complexity, even scaling up to 16-core, 32-core, or 64-core processors capable of handling advanced computing tasks. Conversely, it is possible to design smaller systems suitable for use cases like smartwatches and IoT devices. These systems can serve as platforms to run any type of Linux or even RTOS software, fulfilling diverse application needs. I believe further research in this area can lead to even more optimized systems with greater computational benefits and applications.

Personally, I am deeply content with the level of self-improvement I have achieved over the past year while handling this project. I am even happier with the subject matter and the area in which I have had the opportunity to work and grow.

Bibliography

- [AS] CoQube Analytics and Services. Fpga design flow. Gives an introduction to FPGA design flow.
- [CPF⁺23] Pierre Cochard, Maxime Popoff, Antoine Fraboulet, Tanguy Risset, Stéphane Letz, and Romain Michon. A programmable linux-based fpga platform for audio dsp. In *Sound and Music Computing Conference*, pages 110–116. Bresin, R., & Falkenberg, K., 2023.
- [GG05] PETER BAER GALVIN and GREG GAGNE. Abraham silberschatz. 2005.
- [JRHK15] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. Riffa 2.1: A reusable integration framework for fpga accelerators. *ACM Transactions on Reconfigurable Technology and Systems (TRETs)*, 8(4):1–23, 2015.
- [LCCM13] Andres M Leiva-Cochachin and Fredy Chalco-Mendoza. Embedding linux with ability to analyze network traffic on a development board based on fpga. *International Journal of Computer Applications*, 77(17), 2013.
- [LY03] Qing Li and Caroline Yao. *Real-time concepts for embedded systems*. CRC press, 2003.
- [PBM⁺22] Marco Pagani, Alessandro Biondi, Mauro Marinoni, Lorenzo Molinari, Giuseppe Lipari, and Giorgio Buttazzo. A linux-based support for developing real-time applications on heterogeneous platforms with dynamic fpga reconfiguration. *Future Generation Computer Systems*, 129:125–140, 2022.
- [SR04] John A Stankovic and Raj Rajkumar. Real-time operating systems. *Real-Time Systems*, 28(2-3):237–253, 2004.
- [Sta98] William Stallings. *Operating systems internals and design principles*. Prentice-Hall, Inc., 1998.
- [Tar23] Eugene Tarassov. Vivado risc-v. <https://github.com/eugene-tarassov/vivado-risc-v>, 2023. Accessed: December 20, 2024.
- [Xil] Xilinx. What is an fpga? Basis of my FPGA definition. Accessed on 2023-10-28.
- [Ün21] M. Ünlerşen. *Field Programmable Gate Array (FPGA)*. 2021. Book on FPGA that gives deep insights into components of an FPGA.

List of Figures

2.1	FPGA Architecture	5
2.2	FPGA design Flow	6
3.1	Audio DSP architecture	10
3.2	RIFFA Architecture	11
5.1	V model	15
5.2	VIRTEX VC707	18
5.3	Command Line Interface for Building the processor	19
5.4	RocketChip Integration in the Hardware Platform	21
5.5	Design Runs Result	21
5.6	IO Integration with Ethernet and UART	22
5.7	Error message indicating the failure to mount the SD card due to a timeout.	23
5.8	OpenSBI output showing platform details, including firmware base address, size, and hardware initialization status.	24
5.9	Login screen after the successful boot process. The default username and password are both set to debian	24
5.10	Terminal output showing the interaction with the "Guess the Number" game.	26
5.11	Screenshot of the AES Encryption/Decryption Application in Action	28
5.12	Output of the network test application, demonstrating successful HTTP request execution.	29
5.13	Terminal output showing the web server running and serving the index.html file.	32
5.14	Browser output showing the project's main dashboard served by the FPGA web server.	32

List of Tables

6.1 Linpack Benchmark Results (200x200 Array) 38

Affidavit

I Mohammad Ashrafuzzaman Siddiqi herewith declare that I have composed the present paper and work by myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The paper and work in the same or similar form has not been submitted to any examination body and has not been published. This paper was not yet, even in part, used in another examination or as a course performance. .

Lippstadt, February 23, 2025

Mohammad Ashrafuzzaman Siddiqi

A Appendix

A.1 Code in detail