Bachelor's thesis

Degree Programme in Information and Communications Technology

2022

Joonas Juntunen

Performance testing of RISC-V cores using cycle-accurate software simulations

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2022 | 66 pages

Joonas Juntunen

Performance testing of RISC-V cores using cycle-accurate software simulations

With the rising costs of developing integrated-circuit designs, especially in the field of microprocessors, making use of efficient and accurate methods of testing microprocessor designs is of paramount importance. Enabling accurate testing of microprocessor designs early in the development cycle, allows for faster validation of the design for its intended use-case.

This thesis aims to demonstrate the tangible benefits of VHDL software simulations in testing microprocessor designs. This thesis takes the approach of testing the effects of customizing microprocessor designs to fulfill performance improvements on a particular workload. To achieve the objectives of the thesis, the NEORV32 RISC-V processor project was chosen as the microprocessor design that would be customized and tested. Several different customized versions of the NEORV32 processor were configured and these were then tested on a benchmark developed around the xoroshiro64** PRNG.

The goal of the thesis was achieved, with the thesis methodology and results showing that the testing of customized microprocessor designs through VHDL software simulations, gives developers an efficient and accurate method for testing performance optimizations.

Keywords:

RISC-V, VHDL, software simulations, instruction set architecture, FPGA, PRNG

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2022 | 66 sivua

Joonas Juntunen

RISC-V-ytimien suorituskyvyn testaus VHDL-ohjelmistosimulaatiolla

Integroitujen piirien suunnittelun kehittämiskustannusten kasvaessa, tehokkaiden ja tarkkojen menetelmien hyödyntäminen prosessorien testaamisessa on ensiarvoisen tärkeää. Mikroprosessorisuunnitelmien tarkka testausprosessi kehityssyklin varhaisessa vaiheessa mahdollistaa suunnittelun nopeamman validoinnin sen aiottuun käyttötarkoitukseen.

Tämän opinnäytetyön tavoitteena on osoittaa VHDL-ohjelmistosimulaatioiden konkreettiset hyödyt mikroprosessorisuunnittelun testauksessa. Opinnäytetyössä testataan suorituskyvyn optimointia erityisillä työkuormilla ja seurataan mikroprosessorisuunnittelun mukauttamisen vaikutuksia prosessoriin. Tavoitteiden saavuttamiseksi mikroprosessoriksi valittiin NEORV32 RISC-V prosessoriprojekti, jota kustomoidaan ja testataan. NEORV32-prosessorista määritettiin useita erilaisia räätälöityjä versioita, ja niitä testattiin xoroshiro64** PRNG:n ympärille kehitetyllä suorituskykytestillä.

Opinnäytetyön tavoite saavutettiin, ja opinnäytetyön metodologia ja tulokset osoittivat, että räätälöityjen mikroprosessorisuunnitelmien testaus VHDL-ohjelmistosimulaatioiden avulla antaa kehittäjille tehokkaan ja tarkan menetelmän suorituskyvyn optimoinnin testaamiseen.

Asiasanat:

RISC-V, VHDL, ohjelmistosimulaatio, käskysarja-arkkitehtuuri, FPGA, PRNG

Contents

Figures

Pictures

Tables

List of abbreviations

ALU	Arithmetic Logic Unit
ARM	Advanced RISC Machines
CISC	Complex Instruction Set Computer
CPI	Cycles Per Instruction
CPU	Central Processing Unit
CSR	Control and Status Register
DSP	Digital Signal Processing
DUT	Device Under Test
ELF	Executable and Linkable Format
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
IC	Integrated Circuit
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IoT	Internet-of-Things
ISA	Instruction Set Architecture
LUT	Lookup Table
MCU	Microcontroller unit
MUX	Multiplexer

PRNG	Pseudorandom Number Generator
RTL	Register-Transfer Level
RISC	Reduced Instruction Set Computer
SoC	System-on-a-Chip
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
XOR	Exclusive-OR logical operation

1 Introduction

With the emerging trend of Internet-of-Things (IoT) devices and the increasing number of personal computing devices in the world, we find ourselves with an ever-growing need for compact and power-efficient central processing units (CPUs) to power these devices. While the computing landscape has historically been based on Complex Instruction Set Computer (CISC) designs like the x86-family of CPUs [1], they have usually been ill-equipped in the field of power-efficiency and low-cost cores. Reduced Instruction Set Computers (RISCs) have been increasingly used over the years to supplant CISC-based designs in smaller devices partly due to their low-cost and power-efficient designs. With over 200 billion CPUs produced [2], the Advanced RISC Machines (ARM) Ltd. is the world's leading supplier of CPUs to power the IoT and personal computing devices of the world. This is the stage on to which the RISC-V instruction set architecture (ISA) aims to compete in. Although this is a major hurdle for the RISC-V ISA to overcome, it has a great number of advantages that should certainly help speed up the process. The advantages of the RISC-V ISA are in both its open-source model, meaning that it is made freely available to interested parties without any sort of restrictions, and in its small base ISA with optional modular extensions supporting a wide-array of customization [3]. This focus on implementing one base ISA specification with multiple optional extensions brings about a challenging question when customizing RISC-V cores: how to test the effects of customizing a RISC-V core efficiently and accurately. This is the question that this thesis aims to answer.

1.1 Motivation

One main of the cornerstones of any software engineering project is efficient and accurate testing during the development process [4]. This is even more important for hardware development projects and especially microprocessor development, where the rising design costs of integrated circuits (ICs) have risen from a sum of $51.3 million to $542.2 million in moving from 28nm

transistors to smaller 5nm technologies [5]. With such a high cost to the design of ICs, efficient and accurate methods for validation of IC designs enables developers a faster path to the validation of the overall design. If this validation shows any shortcomings in the design, development can then easily be pivoted to obtain the needed performance, hardware resource utilization, or power consumption requirements. This is the main objective of the thesis: to show how VHDL software simulations can allow for efficient and accurate testing of customized RISC-V CPU cores.

While several different benchmarking approaches can be considered (e.g., hardware resource utilization or power consumption benchmarks), the thesis aims to show how this testing can be used to accurately benchmark the performance of the RISC-V core under test. Also, the thesis will go over the performance implications of customizing the underlying RISC-V core with optional ISA extensions and making use of hardware resources to speed up the computation of a particular workload. Another motivation for the thesis was that while previous studies [6] exist on the customization of RISC-V cores to achieve a performance improvement on a specific workload as described in Section 1.4, none of these studies focus on the tangible benefits of testing and verification through hardware description language (HDL)-based software simulations.

1.2 Methodology

To achieve the main objectives of the thesis, the following quantitative and analytical methods were employed. First, a suitable RISC-V core was chosen for the thesis that could be customized and tested to obtain performance benchmarks on a particular workload. Special care was paid in this process to choose a core that implements the processor with the VHSIC hardware description language (VHDL). After some research, the NEORV32 RISC-V system-on-a-chip (SoC) project was chosen to be used in the thesis. Also, for testing the performance impacts of customizing the NEORV32 CPU core, a pseudo-random number generator (PRNG) was chosen as the benchmark application. The NEORV32 CPU was then progressively customized with either

an optional ISA extension or the usage of a specific hardware resource that, in theory, should improve the performance of the benchmark. The customized core was then tested through VHDL software simulations to obtain the actual performance metrics of running the benchmark. The performance improvements were then verified through the generation of VHDL waveform files. Furthermore, the customized NEORV32 SoC was then also programmed on to a Field-Programmable Gate Array (FPGA) development board. This was done to further verify that the performance results obtained from the software simulation matched results from an actual hardware implementation. Finally, the performance results from each customized version of the NEORV32 CPU were compared and analyzed to obtain the overall performance improvements between each core version.

1.3 Previous Studies

While numerous studies exist on the topic of customizing or optimizing RISC-V cores for performance gains on specific workloads, almost all of the studies are concerned with the addition of custom extensions to the ISA or the development of a completely new RISC-V core. Studies such as, “A lightweight ISE for ChaCha on RISC-V” [7] and “Extending RISC- V ISA for Accelerating the H.265/HEVC Deblocking Filter” [8] deal with the subject of adding custom extensions to the RISC-V ISA for performance gains in the ChaCha stream cipher and HEVC deblocking filter respectively; while other studies such as, “RISC-V2: A Scalable RISC-V Vector Processor” [6] develop a completely new RISC-V core for machine-learning tasks. This thesis, on the other hand, will test the performance impacts of already established and ratified RISC-V ISA extensions. Also, while numerous studies and even textbooks exist on the topic of using VHDL simulations such as, “VHDL for Logic Synthesis” [9], these are either focused on the theoretical underpinnings of VHDL simulations or a general overview of using VHDL simulations for design verification. Instead, this thesis takes a more focused look on how VHDL software simulations can enable accurate testing of RISC-V cores. The closest study in topic and format

to this thesis was found to be a 2015 study by Ritpurkar et al. [10], “Design and simulation of 32-Bit RISC architecture based on MIPS using VHDL”. Although similar in topic and scope, the study by Ritpurkar et al. [10] focused on a MIPS-based microprocessor and on testing the overall functionality of the processor through VHDL software simulations. In comparison, this thesis focuses on a RISC-V-based microprocessor and further narrows the focus to verifying performance gains through VHDL software simulations.

1.4 Thesis Structure

This thesis is structured into 7 separate chapters. Chapter 1 introduces the thesis topic and motivation for the thesis to the reader. Chapter 2 provides a brief theoretical background on the thesis topic, centered around the RISC-V ISA and the VHDL hardware description language. Following this, Chapter 3 introduces the requirements and motivations for the use of specific tools and methods in the thesis. Chapter 4 describes the methodologies used in testing the customized RISC-V cores using VHDL software simulations. Chapter 5 presents the results obtained from the testing done on the customized RISC-V cores. Chapter 6 discusses the obtained results from the tests, any limitations of the testing methodologies, and possible improvements to the thesis work. Finally, Chapter 7 summarizes the work carried out in this thesis.

2 Theoretical background

In an effort to aid the reader to better understand the act of testing RISC-V cores through VHDL software simulations, this chapter will go over the theoretical background of the concepts and technologies used in the thesis. First, a cursory overview of CPU design is introduced before further focusing the topic to the RISC-V family of CPUs. Following this are sections concerned with both the VHDL hardware description language and FPGA devices. Finally, the chapter ends with a brief look on the subject of pseudo-random number generators.

2.1 CPU and RISC-V Background

From the introduction of the first microprocessor, the Intel 4004, in 1971, CPUs have had a profound impact on every facet of modern society [11, p. 458]. Microprocessors are used in almost all digital devices that consumers interact with on a day-to-day basis including computers, mobile phones, and even in the servers that make up the backbone of the Internet. While the personal computer might be the most familiar device to normal consumers that contains a CPU, in 2016 these amounted to only about 1% of all CPU sales with the last 99% coming from the embedded market in the form of microcontroller units (MCUs) [12]. Embedded MCUs are used in the automotive industry, in industrial control applications, and in consumer electronics where a small processor is needed to fulfill one specific function [13]. This wide-ranging use of microprocessors and MCUs in the modern world has resulted in the design of thousands of differing processor designs, each built to fulfill a certain use-case. The one point of commonality between most of these designs, is the implementation of a central processing unit responsible for handling and processing data.

2.1.1 CPU and system design

The central processing unit or CPU of a general-purpose computer, as discussed previously, is the main circuit of a digital device that is responsible for the operation and data processing of the system. While the form and implementation of modern-day CPUs has greatly diverged since the early days of computing, the overall design has mostly stayed the same. The design that makes up most general-purpose CPUs, as per Elahi [14, p. 116], is comprised of three main logical units: an arithmetic logic unit (ALU), a set of registers, and a control unit. The ALU is tasked with handling the data processing capabilities of the CPU, including both arithmetical and logical operations. The registers of a CPU are used to hold data that is coming in or going out of the CPU before and after data processing on the ALU. Finally, the control unit functions to control the operations of the overall CPU and the surrounding system, including controlling input/output (I/O) operations and performing instruction execution. While the CPU is able to perform most of its arithmetical tasks by itself, it cannot fully function without an outside memory system containing the instructions and data of a stored program that the CPU can execute.

For the CPU to communicate with the memory system containing program instructions and data, a set of common connections between the two need to be made, i.e., a bus. According to Elahi [14, pp. 117–118], a CPU is commonly connected to a memory system through an address bus, data bus, and control bus. Respectively, these three busses handle the physical address of the memory being accessed, the actual data being transferred to and from the memory system, and any control signals needed to control the operation between the CPU and memory. These bus connections are typically modeled by one of two fundamental architectures, namely the von Neumann and Harvard architectures. These models mainly differ in the way that the instructions and data contained in the outside memory system are accessed by the CPU. In the von Neumann architecture, the CPU is connected to the memory system through one overall data bus that shares access to both instructions and data. This contrasts with the Harvard architecture that splits the

instruction and data accesses to their own data bus connections. Figure 1 [15] showcases the differences between the two fundamental models with the inclusion of a third model, namely the modified Harvard architecture.

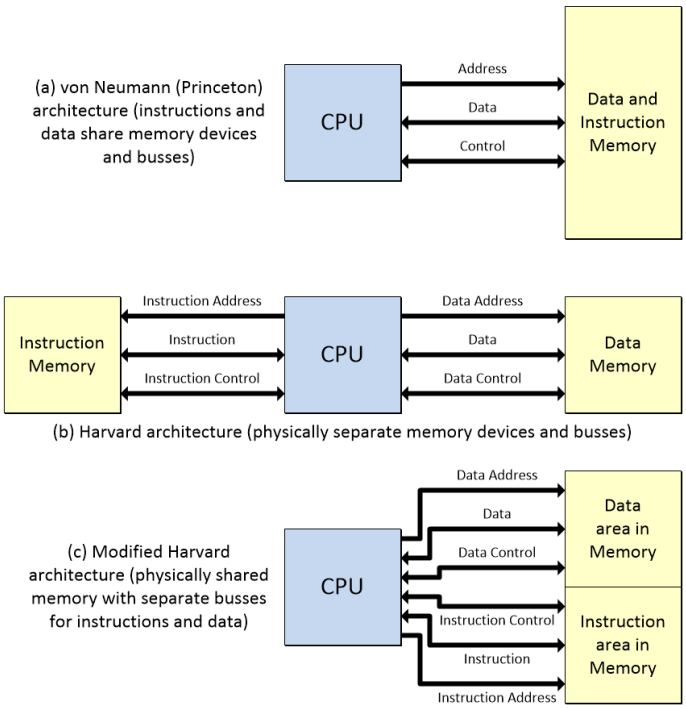


Figure 1. Computer architecture models.

While most CPU designs can be roughly categorized by these three models, modern CPU designs have started to incorporate ideas from all three models. For example, the ARMv8-A architecture implements a set of separate L1 caches for instruction and data akin to the Harvard model before being unified at the L2 cache to a strictly von Neumann model [16]. This inclusion of small caches of faster static random-access memory (SRAM) in between the CPU and the slower outside memory system, usually containing dynamic random-access memory (DRAM), is an effort to combat the bottleneck of transferring both instructions and data through a singular bus in the von Neumann model [11], [17].

While the previous models showcased in Figure 1 [15] give a general understanding of the way in which computer architectures are designed at the hardware level, modern advances have nonetheless brought significant changes to how CPUs are packaged and manufactured. Before the introduction of the Intel 4004 in 1971, most CPUs were housed in large “mainframes” and were constructed on large printed-circuit boards (PCBs) from numerous different chips. The Intel 4004 microprocessor was the first commercially available CPU that was able to be manufactured and housed inside of a single chip, also known as an integrated circuit (IC) [11]. The term microprocessor is widely used when referring to a CPU contained in a single IC which relies on external memory systems for program execution. As discussed previously, around 99% of all CPU sales come from the embedded market in the form of MCU units [12]. A microcontroller unit (MCU) is normally a single IC that contains one or more CPUs, memory, I/O ports, and additional functionality in the form timers and counters [15]. MCUs are typically used in the embedded space for control applications due to their low cost and low power consumption. With the rise of Internet-of-Things (IoT) devices, cellphones, and mobile computing in general, modern microprocessors are also manufactured in system-on-a-chip (SoC) designs, which in contrast to MCU chips, contain a far greater number of integrated components. For example, the A14 ARM SoC contained in the iPhone 12 integrates six CPU cores, four graphics processing units (GPUs), a dedicated sixteen core neural network processor, and up to 16GB of LPDDR4X system memory on a single SoC [18].

2.1.2 Instruction set architecture

As discussed in the previous section, for a CPU to execute any tasks, it needs to be fed with instructions and data from a stored program. For this stored program to be compatible with the CPU it is running on, the CPU needs a well-defined interface above the hardware level that programmers can utilize when writing programs for the CPU. This is interface between the hardware and low-level software is named, the instruction set architecture (ISA). The ISA abstracts

the task of commanding the underlying hardware into a model for which low-level software can be written for. As Patterson and Hennessy [19] note, this also allows for many different CPU hardware implementations to run identical software and is one of the most important abstractions in computer design. But before going further over the topic of ISAs, it may be prudent to go over the steps that a CPU goes through when executing program instructions.

According to Elahi [14, pp. 123–124], a CPU executes one instruction contained in a stored program in the following steps:

1. Fetch - Fetch the next instruction from memory for the CPU to use.
2. Decode - Decode the type of instruction with its operands and place into internal registers if applicable.
3. Execute – Execute the decoded instruction.
4. Store results – Write the results of the instruction into registers or memory.

This is a general overview of the instruction cycle in most CPU designs, though depending on the ISA and the specific hardware implementation, the phases of one instruction cycle can vary a great deal. Although the phases of the instruction cycle may vary between different implementations, one essential component of almost all CPUs that affects the instruction cycle, is the clock frequency of the processor. As almost all modern CPUs are synchronous circuits in design, they rely on an external clock signal for the CPU to work. This is due to the types of combinational logic and state elements, such as the arithmetic logic unit (ALU) and registers respectively, that are used to build a CPU [19]. Both combinational logic and state elements require the use of a clock signal to define when signals in the circuits can be written to or read from. Also, the use of synchronous circuits allows for CPUs to be highly predictable in their operations due to the reliance on an external clock signal.

Instruction set architectures (ISAs) define all of the features that are built into any processor architecture that can be utilized by programmers for the creation of low-level software. The ISA normally defines the following features for a specific processor architecture [13]:

1. Operations or Instructions
2. Types of operands
3. Memory features
4. Register files
5. Addressing modes
6. Interrupts and Exception handling

Probably the most important feature defined by the ISA, is the types of operations or instructions that the processor can execute in an instruction cycle. These can range from mathematical operations, to moving data around the system, branching operations, or I/O operations. Also, each instruction has a defined “opcode” or the actual binary representation of the instruction that the CPU can understand. Akin to the instruction features of an ISA, is the types of data or operands that the processor can use during instruction execution. This includes the size of the operands (8-bit, 16-bit, 32-bit, etc.), operand formats (binary, decimal, hexadecimal, etc.), or the use of more complex data types, such as ASCII characters or floating-point numbers. [13]

The ISA also defines how memory is organized and used in the processor. To successfully program any kind of software for the processor, one needs to know how the address space of the memory system is organized, where data is stored, and how to interface with the memory. This also extends to the registers of the processors. The ISA needs to define how registers are used (general registers or specific-data register), the number of registers contained in the architecture, and the memory size of each register (16-bit, 32-bit, 64-bit, etc.). Also, the addressing modes of the processor need to be defined in the ISA to account for how the processor can utilize operands contained in memory. This can normally be boiled down to two different models: load-store architectures or register-memory architectures. A load-store architecture only allows for

operations and instructions to modify data inside of registers. This means that any data that needs to be used in an outside memory system, needs to be first loaded into the registers of the processor before being operated upon. A register-memory architecture, on the other hand, allows for operations to be done on data contained in registers or contained within any outside memory system. Finally, the ISA also needs to define the interrupt and exception handling capabilities of the processor. [13]

Modern processor ISA developments have largely been focused on two main types of ISAs, the complex instruction set computer (CISC) ISAs and reduced instruction set computer (RISC) ISAs. While other types of ISAs exist, most commercial processors in use today are built upon the fundamentals of one of these two ISA flavors. Both terms were first coined in a seminal paper, “The Case for the Reduced Instruction Set Computer” by Patterson and Ditzel [20], which proposed a much simpler method of processor design when compared to the ISA designs of the 1980’s. CISC design philosophies, at the time, revolved around designing the complexities of the ISA into the hardware of the processors [1]. The RISC design philosophy proposed by Patterson and Ditzel [20], instead advocated for simplifying the hardware of the processor by moving the complexities of the ISA into the compiler and software side of the architecture.

CISC-style ISAs attempt to improve the performance of the CPU by using more specialized and complex instructions. The performance of the CPU can thus be increased since fewer instructions are needed to execute a specific program even though an instruction may take more than one clock cycle. RISC-style designs, on the other hand, use far simpler and generalized instructions. With simpler instructions, the CPU can execute more instructions per clock cycle. For example, the x86-64 ISA contains around 981 instructions while the RISC-V RV32G ISA contains only 122 instructions [21], [22]. This dichotomy between the CISC and RISC ISAs, also plays a part in their memory usage. One of the major design decisions in CISC ISAs of the 1970’s, was partly in due to the high cost of memory systems of the era [1], [20]. With a CISC-like design, the

memory requirements of a stored program are much more efficient since each instruction is able to execute many operations per instruction. RISC-like designs on the other hand, usually consume more memory per program since simpler instructions are needed to execute the same amount of work. Figure 2 [23] shown below, further showcases the general differences between the CISC and RISC ISA designs.

CISC	RISC
The original microprocessor ISA	Redesigned ISA that emerged in the early 1980s
Instructions can take several clock cycles	Single-cycle instructions
Hardware-centric design – the ISA does as much as possible using hardware circuitry	Software-centric design – High-level compilers take on most of the burden of coding many software steps from the programmer
More efficient use of RAM than RISC	Heavy use of RAM (can cause bottlenecks if RAM is limited)
Complex and variable length instructions	Simple, standardized instructions
May support microcode (micro-programming where instructions are treated like small programs)	Only one layer of instructions
Large number of instructions	Small number of fixed-length instructions
Compound addressing modes	Limited addressing modes

Figure 2. Differences between RISC and CISC ISAs.

2.1.3 RISC-V

RISC-V is an open standard RISC-based instruction set architecture (ISA) developed in May 2010 at UC Berkeley by Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanovi´c. This development was done as a part of the Parallel Computing Laboratory or Par Lab project funded by Intel and Microsoft to advance the field of parallel computing. The first version of the base-level ISA for RISC-V was released in May 2011 and subsequently made available through open-source licenses to the public [24]. In 2015, the RISC-V Foundation was founded with 36 founding members to oversee and steer the development of the RISC-V ISA before being reformed as the RISC-V International Association in 2020. RISC-V International is based in Switzerland as a global non-profit association for the open-source collaboration centered around the RISC-V ISA. All documents defining the RISC-V ISA are published under open-source licenses and provide unrestricted use for anyone to develop products based on the RISC-V ISA. [25]

The RISC-V architecture is a load-store architecture based on RISC principles. The ISA is designed to be simple and minimalistic to allow for efficient implementation on a variety of hardware designs. The ISA comes defined in one base integer specification for both 32-bit (RV32I) and 64-bit (RV64I) address space versions. There also exists an embedded variant for the embedded market, RV32E, which is almost equivalent to the RV32I specification and only differs by the number of included general registers (32 vs 16). The base specification includes support for integer operations, such as addition, subtraction, and shift operations. Also, support for bitwise logical operations, control flow operations, conditional branching operations, and memory access operations are contained in the base specification. Unlike many other ISAs, the RISC-V ISA is built modularly on top of the base integer specification. Numerous optional specifications can be specified when developing a RISC-V CPU entirely depending on the needs of the developer. This allows for the development of RISC-V CPUs for use in the embedded market, in general-purpose computers, or even in supercomputer designs. [26]

The following list specifies most of the common optional extensions and descriptions of their functions [26]:

- M-extension – Support for integer multiplication and division. Adds multiplication and division instructions for integers.
- A-extension – Support for atomic instructions. Adds instructions for synchronizing memory accesses atomically.
- F-extension – Support for single-precision floating-point. Adds support for 32 32-bit floating point registers and instructions for floating-point operations.
- D-extension – Support for double-precision floating-point. Adds support for 64-bit floating-point instructions.
- Zicsr-extension – Support for status and control registers. Adds support for 4096 control and status registers in a separate address space.
- C-extension – Support for compressed instructions. Adds compressed 16-bit versions of common operations.
- B-extension – Support for bit manipulation. Adds support for common bit manipulation operations.

Thus, when developing a RISC-V processor, only the required extensions needed by the implementation need to be defined. This allows for greater flexibility and simplification of the implementation design during the development phase.

The RISC-V ISA implements 32 general-purpose registers for use by the processor in both the RV32I and RV64I base specifications. In the RV32I variant, each register is 32-bits in size, while the RV64I variant widens these to 64-bits. The embedded variant, RV32E, keeps the 32-bit size of the registers while only implementing 16 registers. The memory address space that each variant is able to address is also 32-bits or 64-bits respectively. [26]

2.1.4 NEORV32

The NEORV32 processor is a RISC-V compatible system-on-a-chip (SoC) design based on the NEORV32 CPU. The processor is intended to either perform as an auxiliary processor in SoC designs or used as a stand-alone microcontroller. The processor and CPU are both written in the VHSIC Hardware Description Language (VHDL) with numerous ready to use example configurations for Field-Programmable Gate Array (FPGA) development boards. Special care is taken in the development of the NEORV32 processor to ensure that the processor always provides defined and predictable behavior during program execution. Also, the project aims to provide high-quality documentation, portability through platform independent VHDL code, and ease of use for any developer unfamiliar with RISC-V processors. [27]

According to Nolting et al. [27], the SoC design of the processor is highly configurable and provides a wide array of peripherals right out of the box. For example, for I/O operations to and from the processor, the NEORV32 project includes support for communication through the universal asynchronous receiver-transmitter (UART), serial peripheral interface (SPI), and two-wire serial interface (i2C) protocols. Also, the NEORV32 SoC includes support for timers, general-purpose input/output (GPIO) ports, and pulse-width modulation (PWM) controllers. Figure 3 [27] shown below, showcases the overall SoC design of the NEORV32 project. [27]

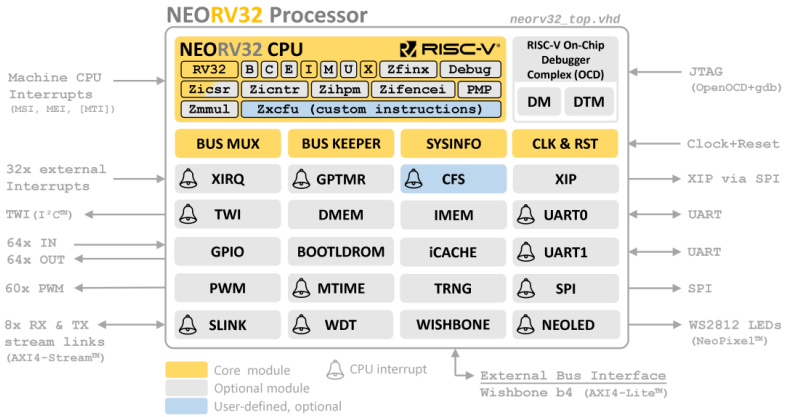


Figure 3. Overview of the NEORV32 SoC.

The NEORV32 CPU is a RISC-V compatible 32-bit (RV32I) single core CPU. Instruction cycles on the NEORV32 are implemented as a two-stage pipeline, wherein the fetch and execution of instructions is de-coupled through the implementation of a first-in first-out (FIFO) queue. This allows the fetch stage to be carried out for a new instruction while the previous instruction is still being executed in the execution stage, increasing the performance of the CPU. The NEORV32 CPU is also a von-Neumann architecture with a single internal bus to the outside memory system. [27]

2.2 VHDL Background

The Very High-Speed Integrated Circuit Hardware Description Language or VHDL, is a hardware description language that allows for describing the behavior of digital circuits through text-based models. VHDL was developed at the American Department of Defense, in the 1980’s, to employ a standard hardware description language (HDL) for documenting the behavior of application-specific integrated circuits (ASICs) received from outside procurement [9]. Although developed at the behest of the American Department of Defense, VHDL was formalized and handed into the public domain by the Institute of Electrical and Electronics Engineers (IEEE) standardization body in 1987.

According to Rushton [9], VHDL is intended for every part of the design cycle in creating electronic systems. In general, the design process for VHDL is taken in three stages: the modeling/specification phase, register-transfer level (RTL) phase, and the final netlist phase [9]. The first phase of the design process deals with modeling the electronic system using VHDL based on the requirements set out at the beginning of development. This VHDL is used to run simulations on the model to check that it exhibits the correct functionality. The second phase of the design process transforms the VHDL model written in the first phase into an RTL-level design for further synthesis in phase three. An RTL-level design is a model of the system built from sets of registers containing data and the transfer functions that describe the flow of data between the

registers [9]. The third phase of the design process then takes the RTL-level design and synthesizes a netlist of the actual hardware components contained in the design with their connections to each other.

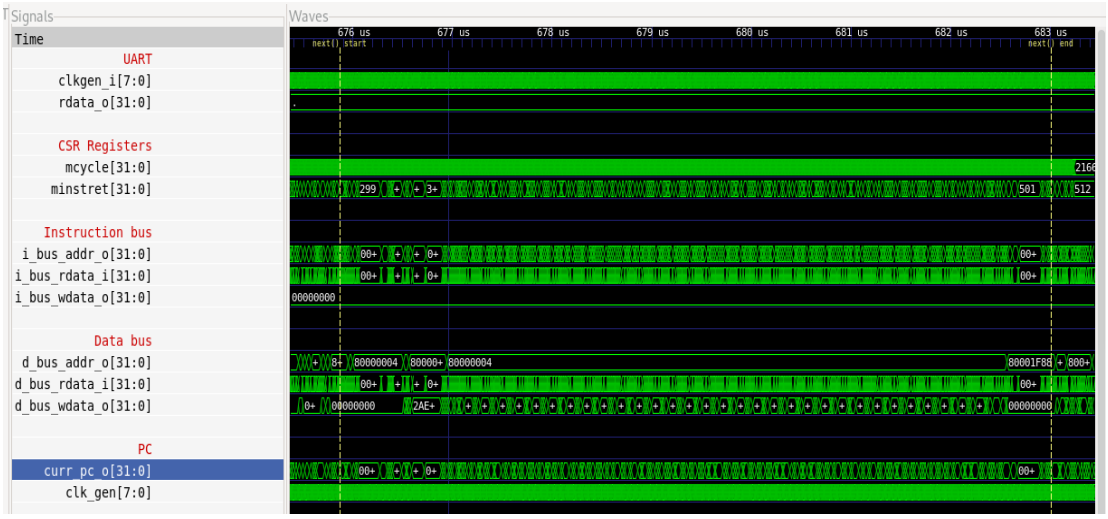
2.2.1 VHDL software simulations

As discussed previously, the first phase of the design process for any new electronic system is the modeling of the behavior and functionality of the system in VHDL [9]. One of the most important parts of this phase is the verification of the model through software simulations. One can also ascertain the importance placed by the IEEE on simulations through the VHDL languages reference manual [28]. This is the standard by which the VHDL language is defined and includes definitions on how software simulators should implement the language. These definitions make sure that all VHDL simulators share the same behavior between different simulator implementations [9].

All VHDL simulators are event-driven in their method of operation. Event-driven simulators simulate the state of the circuit only during times when changes are occurring in the circuit. This helps to minimize the amount of work needed to be done by the simulator when simulating a circuit [29]. Although before simulation of the circuit can occur, the VHDL files that make up the design must be analyzed and compiled into a model that can be simulated by the simulator [29]. Once the simulation model is compiled, the actual simulation can be started. VHDL simulations run through an overall simulation time that is made up of smaller discrete time steps. VHDL simulations run through two different phases called the event processing phase and the process execution phase [9]. During each time step the elements or signals of the simulated model are updated if the signal is due to change in the current time step. After all signals have been updated, the process execution phase of the simulation can begin. A process in VHDL is the core statement by which sequential logic can be defined for design elements. Each process triggered by the change of signals is run until no processes remain to be run. If any of the signals are due to be changed by a

triggered process, the simulator will add new signal events to be computed during the next event processing phase. [9], [29]

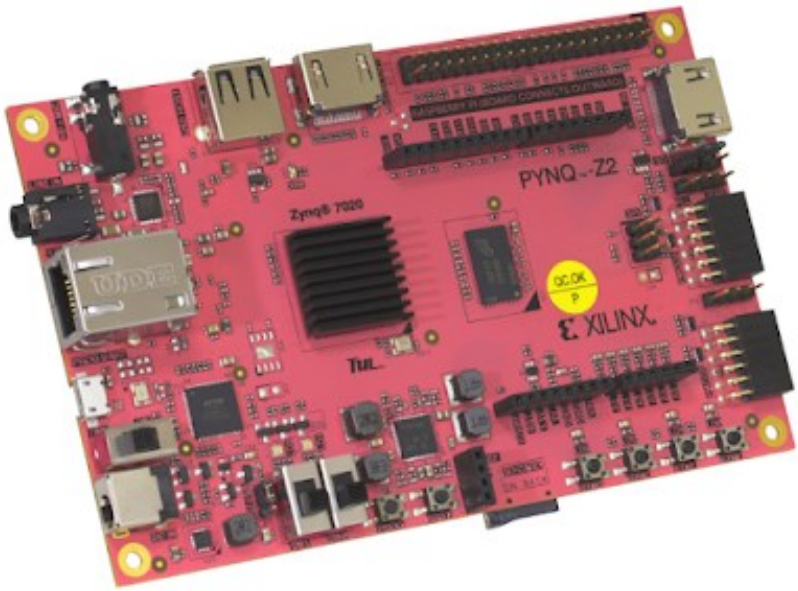
To fully simulate and verify any VHDL model, a testbench must be developed to apply test stimuli to the elements of the model. For example, if an element of a VHDL model requires a clock signal at a certain frequency, without this signal the simulation of the system would not work correctly. Testbenches consist of VHDL language statements that generate the input stimuli to the system and instantiate an instance of the VHDL design, also known as a device-under test (DUT). During the simulation of a testbench, the sequence of transactions undertaken during the simulation can be output as waveform files. Using waveform files allows developers to see the signal states of all elements in the DUT and verify correct functionality of all design components. Picture 1 shows the waveform output from a testbench simulation of the NEORV32 processor done during this thesis.



Picture 1. Example of waveform output from a testbench simulation.

2.3 FPGA Background

Field-Programmable Gate Arrays (FPGAs) are integrated circuits (ICs) made up of programmable logic devices which can emulate any kind of digital circuit. FPGAs are made up of distinct logic elements, I/O elements, and the wiring interconnects between these elements [30]. The programmability of FPGAs comes from the reconfigurability of all three elements to emulate everything from simple logic gates to complex digital circuits. In general, logic elements are made up of lookup tables (LUTs) and the surrounding control logic made up of flip-flops and multiplexers (MUXs) [30]. Most FPGAs also contain a sizeable amount of random-access memory (RAM) for storage of larger amounts of data than can be held in simple LUT elements. Also, since most the logic elements contained in FPGAs are synchronous in nature, most FPGA development boards also contain clock signal generators. In addition to the basic logic elements, modern FPGA boards also contain dedicated hardware for digital-signal processing (DSP) tasks. Picture 2 [31] shows an example of an FPGA board, the PYNQ-Z2, which is also used in this thesis for implementing and testing the NEORV32 processor.



Picture 2. Example of an FPGA board.

2.4 PRNG Background

Pseudo-random number generators (PRNGs) are computational algorithms for the generation of a sequence of random numbers. As can be determined from the name, the sequence of numbers generated is not truly random in a statistical sense. This is due to the generated sequence being completely determined by the initial value or the seed of the PRNG [32]. A PRNG algorithm is computed in successive rounds with the result from obtained from the previous round. After a certain number of iterations, known as the period, all PRNG algorithms will start to repeat [32]. Although the results from PRNG algorithms are not truly random, PRNG use is nonetheless important in several computing fields due to their reproducibility and speed. For example, generating random numbers through a linear feedback shift register (LSFR) only requires a single right-shift operation and an exclusive-OR operation (XOR) to generate a suitably random stream of numbers [33].

2.4.1 Xoroshiro64** PRNG

The xoshiro/xoroshiro family of PRNGs were developed in 2021 by David Blackman and Sebastiano Vigna in the paper, “Scrambled Linear Pseudorandom Number Generators” [34]. The xoshiro/xoroshiro PRNGs are akin to the LSFR-type number generators, in that the algorithm is computed using bit-shifting and XOR operations. The xoroshiro64** PRNG is a 32-bit number generator using a combination of the “xoroshiro” linear engine and the “**” non-linear output scrambler [34]. The “xoroshiro” linear engine computes the transformation of the PRNG state by the combination of a XOR, shift, and two rotation operations. The PRNG state is further scrambled with the “**” non-linear scrambler consisting of a multiply-rotate-multiply sequence. The intended use of the xoroshiro64** PRNG, due in part to its small state and 32-bit output, is in embedded hardware applications [35].

3 Requirements, tools, and techniques

3.1 Requirements

The main objective of this thesis, as discussed in Chapter 1, is to show how VHDL software simulations can be used to efficiently verify performance improvements on customized RISC-V cores. As such, a rigorous set of requirements should be laid out in the pursuit of the thesis objective.

The MoSCoW method is a methodology for the prioritization of requirements needed to complete a specific project [36]. The requirements are ranked into four categories based on the priority of the requirement and how critical it is to the project's success. Table 1 below describes the requirements set out for the thesis as prioritized by the MoSCoW method.

Table 1. Moscow table of requirements

Priority	**Requirements**
Must Have	32-bit RISC-V core Base Integer specification M-extension support Xoroshiro64** PRNG benchmark
Should Have	B-extension support (Zbb subset) Waveform generation from simulations Speed optimizations using FPGA resources Verification on FPGA hardware
Could Have	64-bit RISC-V core C-extension support E-extension support
Won't Have	A, F, D, etc., extension support

3.2 Tools and techniques

After having a set of requirements laid out, a suitable number of tools and techniques need to be employed to fulfill the requirements for the thesis. The main tools and techniques that were chosen for this thesis were the following:

- 32-bit RISC-V processor or core
    - The NEORV32 RISC-V SoC project [37] was chosen due to its ease of use, extensive documentation, and implementation in the VHDL hardware description language. The NEORV32 SoC also includes compatibility with all of the ISA extensions listed in the requirements. Also, the NEORV32 includes performance optimizations to specific workloads using FPGA hardware resources.
- PRNG-based testing benchmark
    - The xoroshiro64** PRNG [35] was chosen as the workload for testing the performance improvements of different customized versions of the NEORV32 core. This PRNG was chosen due to the main algorithm employing certain computational operations (exclusive-OR, shift, and rotate operations) that can be sped up with the inclusion of specific RISC-V ISA extensions.
- VHDL-based software simulator
    - The GHDL software simulator [38] was chosen due to it being used as the default VHDL analyzer, compiler, and simulator in the NEORV32 project. Also, GHDL can output VHDL waveform files for verification of the performance improvements on the PRNG workload.
- FPGA hardware
    - The Pynq-Z2 FPGA [31] board was chosen as the FPGA hardware for further verification of the PRNG benchmark results. This development board was mainly chosen due to the authors previous experience in its use during university courses.

4 Methodology

In the previous chapter, the requirements, tools, and techniques that were used in this thesis were introduced to the reader. Also, a cursory overview of the motivations for choosing these tools and techniques was also presented. In this chapter, the methodology of the thesis work is presented.

4.1 Tested NEORV32 cores and their selection criteria

Before going over the overall work that was done during the thesis, it seems prudent to go over the NEORV32 core versions that were tested. For fulfilling the thesis objective, several different customized NEORV32 cores were chosen. Each version of the NEORV32 core, needed to implement some sort of performance improvement on the PRNG benchmark. This list of core versions is critical to know in the subsequent sections of this chapter. For example, in the installation phase, the supported ISA extensions need to be configured into the RISC-V toolchain during compilation. Table 2 below outlines the complete list of tested NEORV32 core versions used in the thesis with their optimizations.

Table 2. Tested NEORV32 core versions

Tested cores	Description	Type
RV32I	32-bit RISC-V core with the Base Integer extension support	RISC-V ISA
RV32IM	Addition of hardware multiplication and division support with the M-extension	RISC-V ISA
RV32IMB	Addition of hardware support of bitwise operations with the B-extension	RISC-V ISA
RV32IMB_FM	Faster multiplication operations using DSP slices on FPGA hardware	NEORV32 feature
RV32IMB_FMFS	Faster bit-shifting operations using barrel shifters on FPGA hardware	NEORV32 feature

In selecting the NEORV32 core versions, the xoroshiro64** algorithm was first evaluated regarding the computational operations it contains. The xoroshiro64** algorithm is made up of the following operations:

- Multiplication - Two operations in the PRNG.
- Rotate left operations - Three operations in the PRNG.
- Bit-shift operations – Three operations in the PRNG.

The rotate left operation used in the algorithm is used to rotate the bit values of a 32-bit integer by a specified amount. Each bit value in the number is moved to the left and the left-most bit value is moved to the end of the number [39]. All in all, there are 3 different numerical operations that can be optimized to improve the performance of the PRNG benchmark.

The RV32I version of the NEORV32 core, is the base version that was tested with the PRNG benchmark. This version should, in theory, be the slowest of all the different versions that were tested. The next version, RV32IM, should optimize the multiplication operations in the PRNG benchmark. The RV32IMB version adds hardware support for common bitwise operations, including the rotate left operation. This should, again in theory, optimize the performance of the rotate left operations used in the algorithm. Without hardware support for the rotate left operation, the algorithm emulates the operation using two bit-shift operations, a logical OR operation, and a subtraction operation. The RV32IMB_FM further optimizes the performance of the multiplication operations using digital-signal processing (DSP) blocks on the FPGA hardware. Finally, the RV32IMB_FMFS optimizes the three bit-shift operations also using FPGA hardware.

4.2 Program installation

To build the necessary testing framework, several required software packages needed to be installed. First, to compile the xoroshiro64** PRNG benchmark or basically any programs for the RISC-V processor architecture, a cross-compiling GCC toolchain [40] needed to be installed and configured. Next, the

source code for the NEORV32 [37] project was downloaded and compiled to build a working implementation of the NEORV32 SoC. The Xilinx Vivado integrated development environment (IDE) [41] was also installed to port the NEORV32 processor to the PYNQ-Z2 FPGA development board. Finally, both GHDL [38] and GTKWave [42] were installed to allow for the software simulation of the NEORV32 processors and to view the VHDL waveform files of the testing runs respectively.

On a further note, all of the work done on this thesis was done on a x86-64 Fedora 35 Linux operating system [43]. As such, some commands and installation directions used in the thesis, may only be applicable to the Fedora-family of Linux distributions.

4.2.1 RISC-V toolchain installation

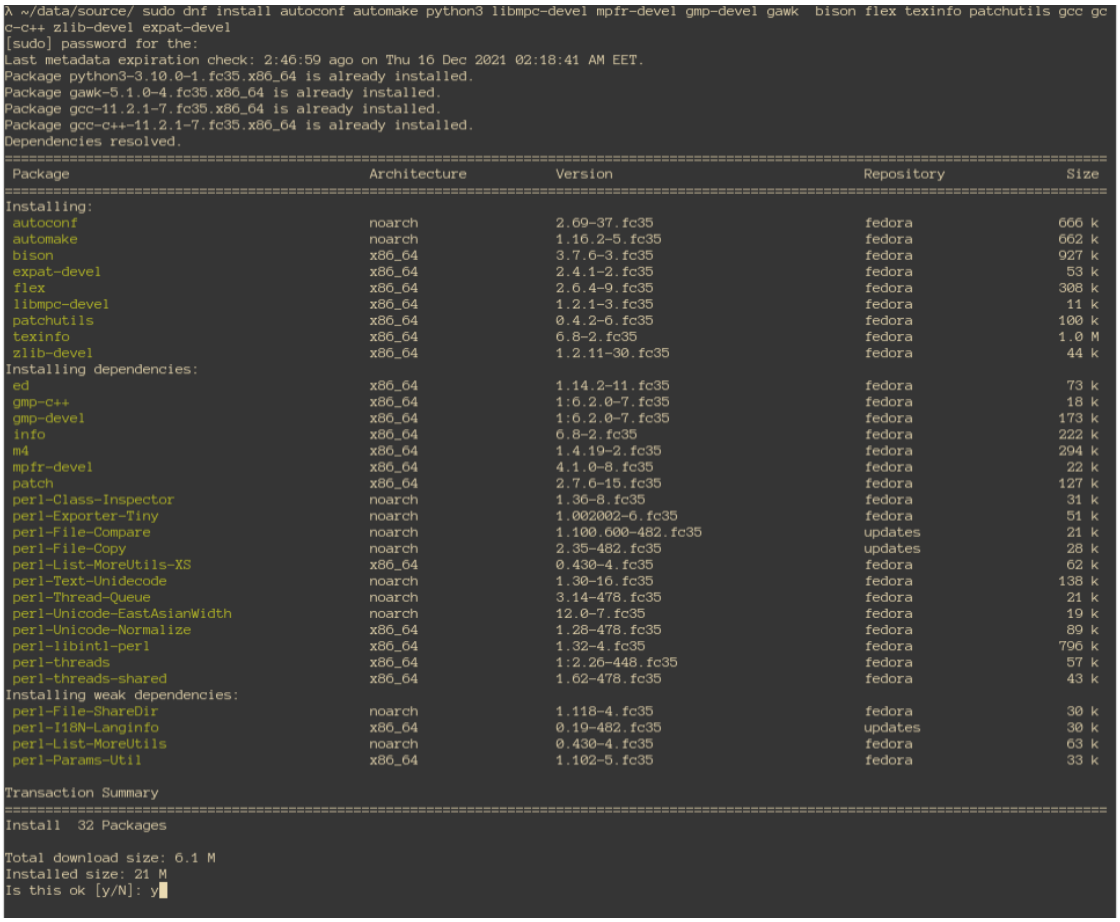
To build the PRNG benchmark for the RISC-V architecture, a compatible cross-compiling toolchain needed to be installed. A cross-compiling toolchain contains all of the necessary tools for compiling programs on the target processor architecture when the underlying systems architecture differs. The RISC-V GNU compiler toolchain [40] was chosen for this thesis due to its use in the NEORV32 project. A suitable LLVM-based cross-compiler could also be used, but the usage of this was not researched in this thesis.

To install the RISC-V toolchain, the source code of the toolchain was downloaded from its official GitHub repository using the Git version-control program [44]. The following command was used download the RISC-V toolchain source code:

- git clone https://github.com/riscv/riscv-gnu-toolchain

After downloading the source code from the GitHub repository, several prerequisite programs were installed through the Fedora package manager, DNF. These prerequisites are detailed in the RISCV toolchains GitHub page

[40]. Picture 3 below shows the installation of the prerequisite programs needed by the RISC-V toolchain before compilation.



Picture 3. Installation of prerequisite programs through the Fedora package manager.

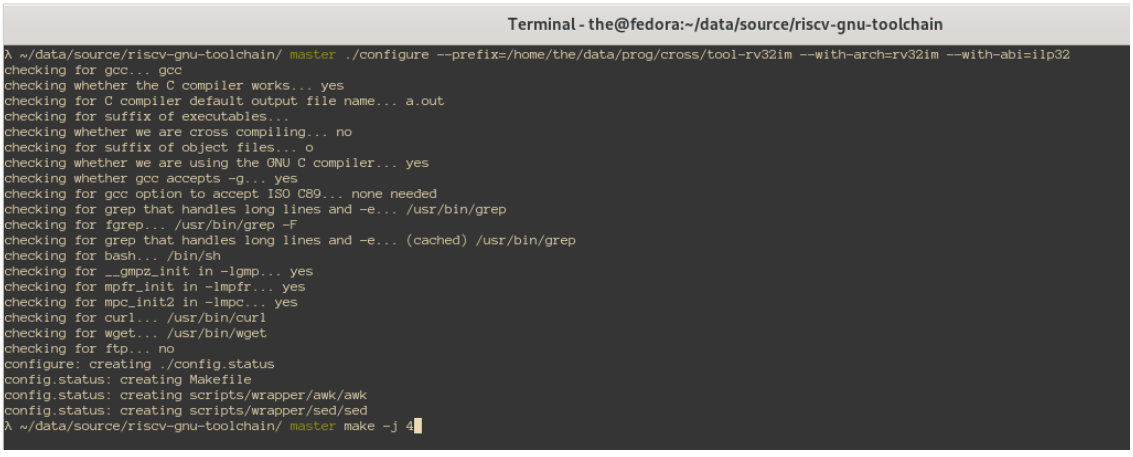
Once all the prerequisite programs were installed, the RISC-V toolchains source code needed to be configured. To successfully compile the PRNG benchmark for the customized NEORV32 processor variants, two features needed to be considered when configuring the RISC-V toolchain: the supported RISC-V ISA extensions and the C standard library used for the program executable.

To compile the PRNG benchmark for the different NEORV32 core versions, the RISC-V toolchain needed to be configured with the corresponding ISA extension support. To compile for all of the targets listed in Table 2 at the start of this chapter, the toolchain should be configured to support the I, M, and B

ISA extensions. As of the writing of this thesis, the RISC-V toolchain does not include support for generating instructions from the B-extension or any of its subsets [45]. This is due to the RISC-V toolchain including GCC version 11.1 [40] while preliminary support for the B-extension has been added to version 12.1 of GCC [46]. This thesis uses a different method for generating the correct instructions and opcodes contained in the B-extension and will be further explained in Section 4.3.1. Excluding the B-extension, both the I and M-extensions are supported in version 11.1 and were thus configured into the toolchain before compilation.

The RISC-V toolchain supports two different C standard library types: Newlib and the GNU C library [40]. The NEORV32 project precludes the use of the Newlib library-based toolchain [47], so this is the version that was configured into the toolchain before compilation.

Before the RISC-V toolchain can be successfully compiled, the toolchain needed to be configured with the correct ISA extension support and C standard library version. Picture 4 below shows the configuration command executed in the root folder of the RISC-V toolchain and the “make” command [48] used for compiling the toolchain after configuration.



Picture 4. Configuration and compilation of the RISC-V toolchain.

When the compilation had finished, the toolchain was ready to use. The final process was to make sure that the RISC-V toolchain executables could be found by the operating system. This was done by adding the location of the toolchains “/bin” folder to the Linux “PATH” environment variable using the following command executed in the terminal [47]:

- export PATH=$PATH:/home/the/data/prog/cross/tool-rv32im/bin

4.2.2 NEORV32 installation

The first task in installing the NEORV32 project, was to download the projects GitHub repository with the following command:

- git clone https://github.com/stnolting/neorv32.git

After the download had finished, the NEORV32 user guide documentation [47] recommended testing that the RISC-V toolchain had been successfully installed and was working as expected. Executing the following commands in the NEORV32 root folder, made sure that the toolchain and the NEORV32 project were working as intended:

- cd sw/example/blink_led/ - Move to the example program folder.
- make check – Will compile the example program while checking the status of the toolchain and the NEORV32 project.

With the RISC-V toolchain and the NEORV32 project installed correctly, the terminal output a “Toolchain ok” message indicating that both were in fact installed correctly.

4.2.3 Xilinx Vivado installation

The Xilinx Vivado integrated development environment (IDE) was installed by downloading the required version from the Xilinx website [41]. The 2018.2 version of Vivado was used during this thesis. In order to download the Vivado installation program, the creation of a Xilinx.com account is required. Once a Xilinx account had been created and the installer downloaded, the IDE was installed from the web-installer onto the operating system. The only installation option that was necessary to configure was which edition of the IDE to install, as some editions require paid-licenses to fully use. The free Vivado HL WebPACK edition of the IDE was chosen to be used in this thesis. Picture 5 below shows the installation of the Xilinx Vivado IDE.

Picture 5. Xilinx Vivado IDE installation.

After the Vivado IDE had been installed, the device drivers that would interface with the FPGA boards supported by Xilinx Vivado needed to be installed. Normally on a Windows operating system, this phase is done by the Vivado installer without manual intervention from the user. On the Linux version of Vivado, these need to be installed separately by the end-user. The following commands executed in the Vivado root folder, installed all of the required device drivers:

- cd data/xicom/cable_drivers/lin64/install_script/install_drivers/ - Move to the Vivado device driver folder.
- ./install_drivers – Install all device drivers for supported FPGA boards.

To add in support for the PYNQ-Z2 board in Vivado, the board files for the PYNQ-Z2 were downloaded from the PYNQ documentations site [49]. These were added to the “/data/boards/board_files/” directory in the root folder of the Vivado installation. Also, the ".xdc” constraints file, from the same site, was saved for later inclusion in to the Vivado project of the NEORV32 processor.

4.2.4 Supplementary software installation

Finally, the GHDL software simulator and the GTKWave VHDL waveform viewer could be installed from the Fedora package manager. Picture 6 below shows the installation process for both GHDL and GTKWave.

Picture 6. Example of supplementary program installation.

4.3 Testing setup

Once all the necessary programs were installed, the testing setup for the thesis was undertaken. In order to test the five different versions of the customized NEORV32 core, a suitable benchmark built upon the xoroshiro64** PRNG needed to be developed. Also, each of the five different core versions needed to be configured appropriately for the software simulation testing. This meant configuring the VHDL testbenches that would be used in the GHDL software simulator. Finally, to verify the benchmark on the PYNQ-Z2 FPGA board, each core version was built and compiled in the Xilinx Vivado IDE for the PYNQ-Z2.

4.3.1 Benchmark program

To obtain a suitable benchmark based on the xoroshiro64** PRNG, a number of auxiliary functions needed to be developed. The benchmark included the source code for the xoroshiro64** PRNG, functions used to calculate the number of cycles and instructions taken during program execution, and I/O functions used to output the benchmark data through a UART interface.

The source code for the main algorithm used in the xoroshiro64** PRNG could be directly imported into the benchmark without any modifications [35].

The NEORV32 project defines several intrinsic functions that enabled querying the status and control (CSR) registers of the processor. The CSR registers that were used in the benchmark are as follows [27]:

- mcycle – 64-bit register holding the number of clock cycles since the start of the processor.
- minstret – 64-bit register holding the number of instructions executed since the start of the processor.
- mcountinhibit – Enable or disable automatic incrementation of CSR counters.

The NEORV32 project defines a core library that includes helper functions for the reading and setting of the CSR registers. The following helper functions were used in the benchmark for the CSR register usage:

- neorv32_cpu_get_cycle() – Queries the mcycle register for the cycle count.
- neorv32_cpu_get_instret() – Queries the minstret register for the instruction count.
- neorv32_cpu_csr_write([CSR register], [value]) – Sets the corresponding CSR register to the specified value.

The PRNG benchmark starts the cycle and instruction counters at the start of each round of the PRNG algorithm. Once one round has been completed, the counters are stopped until the start of the next round. This makes sure that the cycle and instruction counters are only updated during the actual execution of the PRNG and not during execution of any other auxiliary code. The one round cycle and instruction CSR values are then output to the I/O system. They are further saved in variables for calculating the total number of cycles and instructions spent calculating all PRNG rounds at the end of the benchmark. The complete benchmark runs through exactly 25 rounds of the algorithm.

To output data from the benchmark during the execution of the PRNG, any data that needs to be written is output through the universal asynchronous receiver-transmitter (UART) interface. UART is a serial interface for serial communication between two different computing devices. The NEORV32 SoC includes two different UART interfaces that can be enabled for outputting data from the processor [27]. The NEORV32 core library includes helper functions for printing character data from either UART interface. The following function enables communication through the first UART interface:

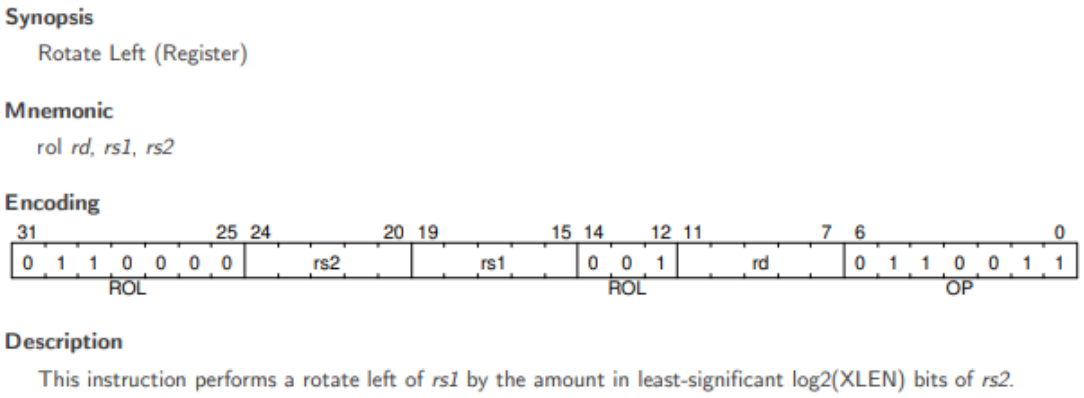
- neorv32_uart0_printf() – Formatted printing function for output through the first UART interface.

This function is used to print the cycle and instruction information gathered during the execution of the benchmark.

As discussed previously in Section 4.2.1, the RISC-V toolchain does not include support for generating instructions and opcodes from the B-extension. In order to generate a rotate left instruction needed for testing the RV32IMB versions, the benchmark instead uses the intrinsic functions contained in the NEORV32 core libraries. These functions can be used for generating custom RISC-V instructions in program executables. The following function will generate the rotate left instruction used in the RV32IMB benchmark:

- CUSTOM_INSTR_R2_TYPE(0b0110000, a5, a0, 0b001, a0, 0b0110011) – Custom instruction function used to generate the rotate left instruction contained in the B-extension.

To decode the bit values used in the function call, Picture 7 shows the encoding table from the RISC-V Bitmanip ISA extension documentation [45, p. 38].



Picture 7. Encoding table for the rotate left operation.

The encoding table defines the format that the instruction needs to follow in order for a compatible RISC-V processor to execute the instruction. The instruction uses two registers, rs1 and rs2, to define the value that will be rotated and the amount that it needs to be rotated by. After the instruction has been computed, the result will be stored in register rd. The custom function coded for the benchmark uses the registers a0 and a5, which are defined as function argument registers in the RISC-V calling conventions [26, p. 137].

Since the custom function can only generate a single RISC-V assembly instruction, this cannot be included in normal C source code. The benchmark instead defines sets of RISC-V assembly code blocks contained in “asm volatile” statements [50] that contain the rest of the PRNG algorithm code that can be directly compiled through the RISC-V toolchain. The substance of the assembly code blocks matches the RV32IM benchmark assembly code exactly, excluding the custom instructions which are input into the final program through the custom function calls.

4.3.2 Simulation setup

In setting up the different test versions of the NEORV32 core, the VHDL testbench included with the NEORV32 project needed to be modified. The VHDL testbench, “neorv32_tb.vhd”, is contained in the “sim” folder at the root of the NEORV32 project folder. For the software simulations, the default testbench included with the NEORV32 project was chosen for use in this thesis, since it already defined everything that is needed for simulating the processor. The modifications that were needed in the testbench, were to configure the NEORV32 core with the correct ISA extensions and any FPGA-specific features for a particular test version. For example, the RV32IM version needed to be configured with the I and M-extensions enabled in the testbench. The NEORV32 project also includes ready-made scripts for starting the simulation through GHDL. The scripts also copy the executable program that is being simulated into the processor’s internal memory map. The executable is then directly booted when the simulation starts in GHDL without needing any input from the user.

4.3.3 Hardware setup

For verifying the performance improvements on actual FPGA hardware, the NEORV32 cores were also ported to the PYNQ-Z2 FPGA board. To start, a new VHDL project was created inside of Vivado 2018.2 that would contain the

VHDL files from the NEORV32 project. After the project had been setup, the default library name of the project needed to be set in the project settings menu of Vivado to "neorv32". If the default library name used in the project was different, the compilation of the VHDL files for the NEORV32 project would fail automatically due to linking errors. Next, the following steps needed to be taken in order to successfully import the NEORV32 VHDL files:

- Import the “.xdc” constraints file for the PYNQ-Z2 board downloaded during the installation phase.
- Import all files inside of the “neorv32/rtl/core/” folder, excluding the “mem” folder, to the project.
- Import all “*.default.vhd” files from the “mem” folder into the project separately.
- Add the “neorv32_testsetup_bootloader.vhd” file from “rtl/test_setups/” into the project and set it as the top design file.

Once all of the required files were imported into the Vivado project, both the top design file, “neorv32_testsetup_bootloader.vhd”, and the “.xdc” constraints file needed to be modified to build the project for the PYNQ-Z2 board. Inside of the top design file, the “CLOCK_FREQUENCY” generic was modified to fit the PYNQ boards natural system clock frequency of 125MHz. Also, the reset buttons, “rstn_i” signal, contained in the NEORV32 source code is triggered on a low signal state. On the other hand, the buttons on the PYNQ-Z2 are enabled on a high signal state. To make sure that the reset buttons to the NEORV32 processor work correctly, the reset signal was inverted in the top design file. Finally, the “.xdc” constraints file was modified to map the internal NEORV32 signals to the correct signals on the PYNQ-Z2 board.

The following changes were made in the constraints file for the NEORV32 processor to work:

- Set the PYNQs internal 125MHz clock to the NEORV32 “clk_i” signal.
- Set the first button on the PYNQ to the NEORV32 “rstn_i” signal.
- Set the first four “gpio_o” output signals of the NEORV32 to the PYNQ LEDs.
- Set the “uart0_txd_o” and “uart0_rxd_i” signals of the NEORV32 to the PmodB output of the PYNQ.

Once all of the required files were modified, the synthesis, implementation, and bitstream generation were executed. Once these were all completed, the compiled bitstream was then uploaded to the PYNQ-Z2 device.

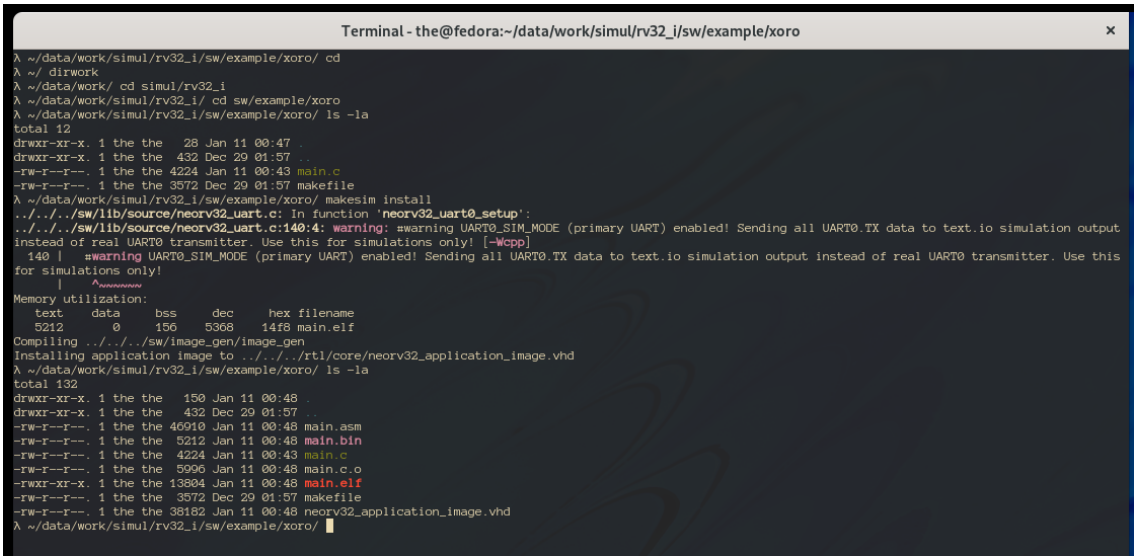
4.4 Testing runs

Once all of the necessary setup steps were taken, the actual NEORV32 core versions were ready to be tested in both the GHDL simulator and on the PYNQ-Z2 board. The testing workflow went about in the following way:

1. Compile the benchmark for the NEORV32 core being tested.
2. Simulate the benchmark on the modified testbench through GHDL.
3. Verify the performance improvements in the assembly code output of the compiled benchmark.
4. Verify the performance improvements through the generated VHDL waveform file.
5. Execute the benchmark on the NEORV32 core being tested contained on the PYNQ-Z2 board.
6. Verify that the output from the hardware and software tests match.

4.4.1 Simulation test runs

In order to simulate the PRNG benchmark through the GHDL simulator, the benchmark source code was first compiled through the RISC-V toolchain to obtain an executable program for the NEORV32 processor. This was done using the included “make” scripts of the NEORV32 project. The scripts will automatically compile the program executable and install it to the internal memory of the simulation testbench. Picture 8 below shows the compilation of the benchmark program through the RISC-V toolchain.



Picture 8. Compilation of the benchmark program.

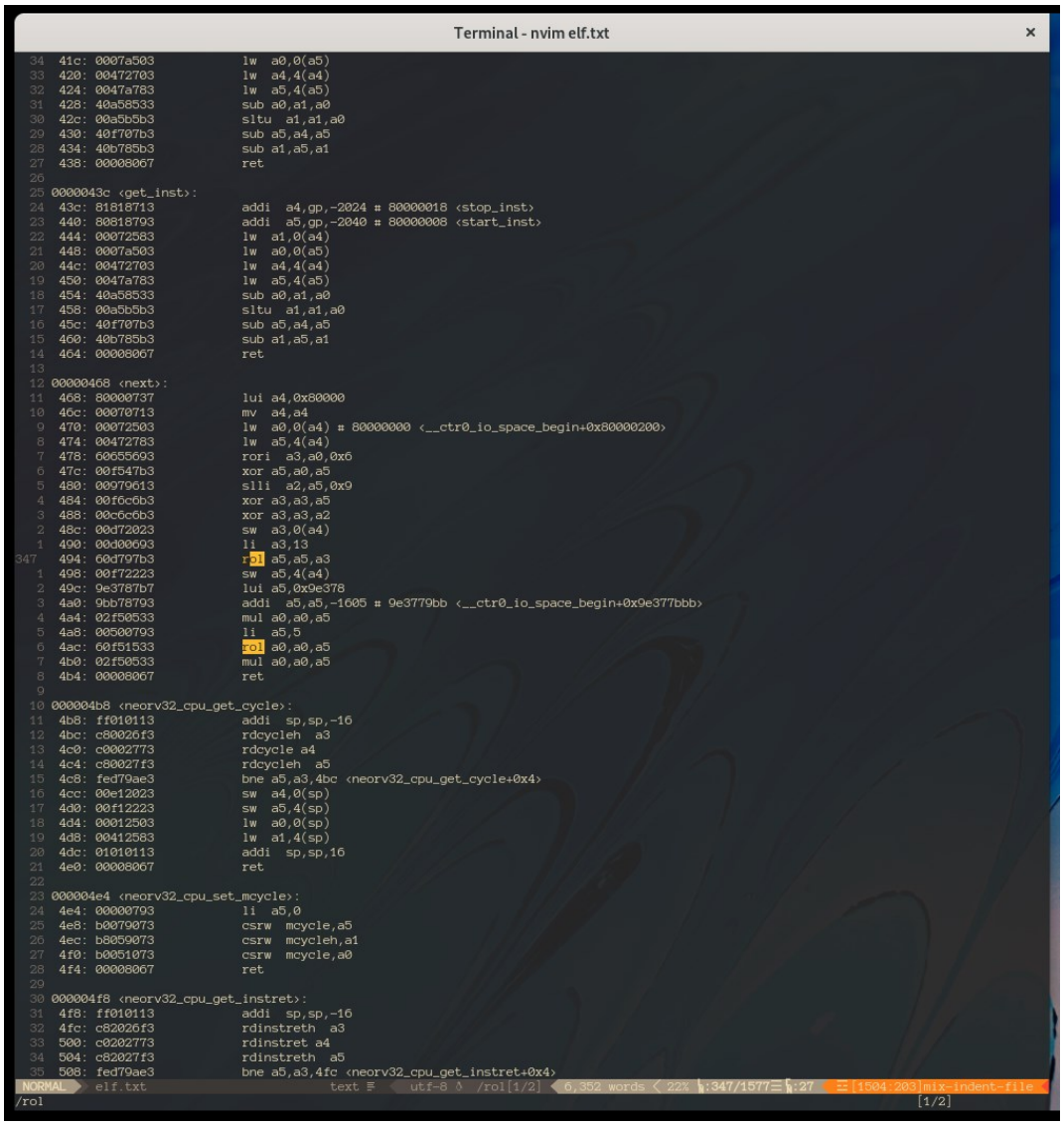
Once the compilation of the benchmark was finished, the simulation run could then be started. The NEORV32 codebase contains two bash script files for running the GHDL simulations automatically with the compiled executable program. The “.ghdl.setup.sh” script needs to be run first, as this will import all of the required VHDL files and synthesize them through GHDL. Once this command has finished, the “.ghdl.run.sh” script can be run to start the actual simulation of the benchmark through the NEORV32 testbench. Picture 9 below shows the output from the GHDL simulation running the PRNG benchmark.

Picture 9. Simulation of the PRNG benchmark.

4.4.2 Verifying simulation runs

The verification of the simulation runs was done in two different phases. First, the assembly code of the benchmark program was dumped from the “.elf” executable obtained through the RISC-V toolchain. The executable and linkable format (ELF) is a common file format used for executable programs in Linux-based operating systems. The ELF file of the benchmark was dumped through the RISC-V toolchains “objdump” program, which outputs the assembly code

listing of a compiled executable. Picture 10 below shows the assembly code listing dumped from the RV32IMB benchmarks executable ELF file.



Picture 10. Dumped assembly code listing of the PRNG benchmark.

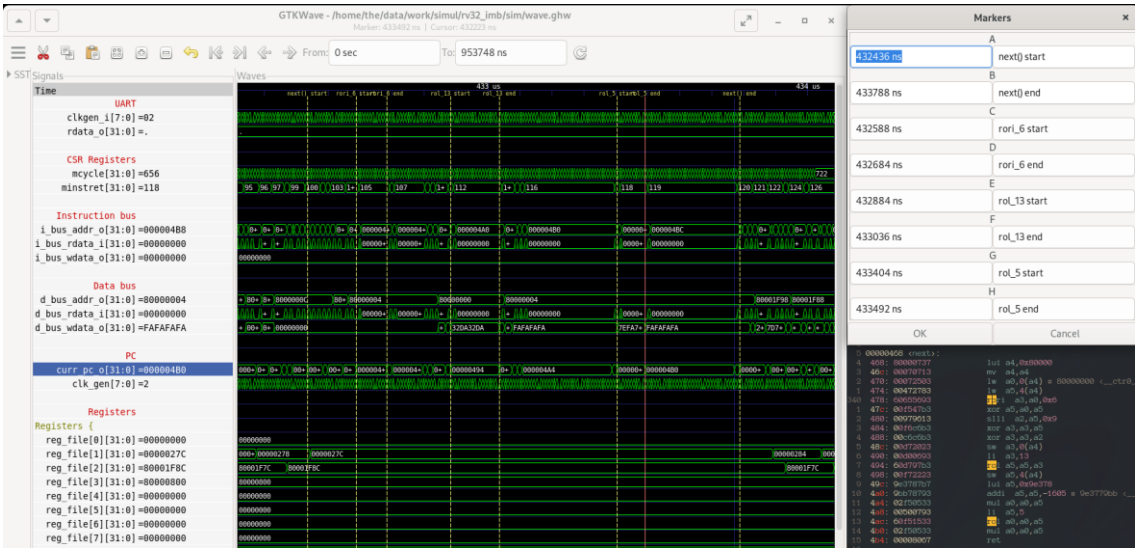
This assembly code listing was used to verify that the customized ISA extensions were reflected in the assembly code. For example, Picture 10 shows the output from the RV32IMB benchmark which should implement the rotate left (rol) instruction optimization, which is highlighted in yellow in the assembly code. After the performance optimization was verified through the assembly code listing of the benchmark, the next method of verification was through the generated VHDL waveform files.

The VHDL waveform files contain all of the operations that the NEORV32 processor executed during the simulation run through GHDL. Using the assembly code listing dumped in the previous part, the memory address for the start of the PRNG algorithm can be found. For example, in Picture 11, the memory address of the PRNG function starts at the memory address value of “0x00000468”.

```
15 00000468 <next>:
14  468: 80000737           lui a4,0x80000
13  46c: 00070713           mv  a4,a4
12  470: 00072503           lw  a0,0(a4) # 80000000
11  474: 00472783           lw  a5,4(a4)
10  478: 60655693           rori  a3,a0,0x6
 9  47c: 00f547b3           xor a5,a0,a5
 8  480: 00979613           slli  a2,a5,0x9
 7  484: 00f6c6b3           xor a3,a3,a5
 6  488: 00c6c6b3           xor a3,a3,a2
 5  48c: 00d72023           sw  a3,0(a4)
 4  490: 00d00693           li  a3,13
 3  494: 60d797b3           rol a5,a5,a3
 2  498: 00f72223           sw  a5,4(a4)
```

Picture 11. Memory location of the PRNG algorithm.

Using the search functions in GTKWave, it was possible to search for all occurrences of the PRNG functions starting memory address. Thus, the total number of cycles, instructions, and time spent computing the PRNG benchmark can be further verified through the VHDL waveform file. Picture 12 below shows the waveform file of the RV32IMB version, and the calculated time spent executing one round of the PRNG.



Picture 12. Verification of performance optimizations through a VHDL waveform file.

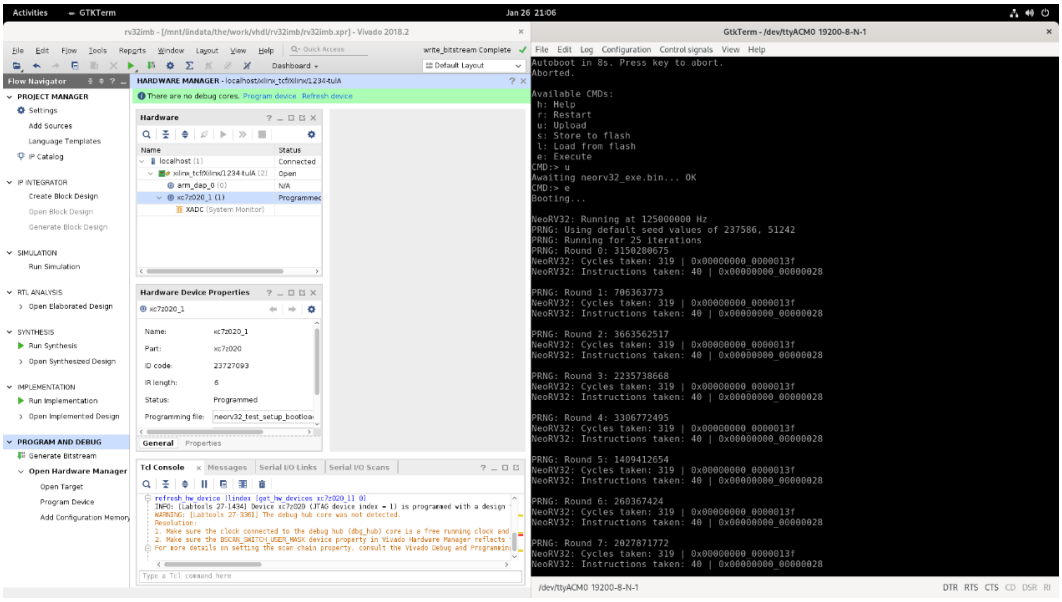
4.4.3 Hardware test runs

Once the simulation run for a certain NEORV32 core version was finished, the compiled benchmark was then run on the PYNQ-Z2 version of the NEORV32 core. In the simulation of the NEORV32 core, the benchmark executable is copied into the processors internal memory from which it can start to execute the program. The PYNQ-Z2 version of the NEORV32 processor, on the other hand, uses a small bootloader from which the program executable can be installed. While the same method of copying the executable to the processors internal memory could also be used on the PYNQ-Z2, it was deemed to be easier to use the bootloader version in the hardware testing runs. This is because the benchmark can be started manually once the serial communication to the PYNQ-Z2 is established. In order to communicate with the bootloader on the NEORV32 processor, a suitable UART receiver needed to be connected to the PmodB inputs of the PYNQ that contain the UART signals from the NEORV32. During the thesis work, a Teensy 4.0 microcontroller was used as the UART receiver, but any UART or FTDI-based receiver would work. The Teensy routes the UART serial transmission through its Universal Serial Bus (USB) port to a host computer, where a compatible serial terminal needs to be

installed. The GTKTerm program was used for interfacing with PYNQ-Z2 FPGA board during this thesis. Also, the following serial interface settings were configured in GTKTerm to match the serial UART output of the NEORV32 core:

- 19200 Baud
- 8 data bits
- 1 stop bit
- no parity bits
- no transmission/flow control protocol

Once the connection was established to the PYNQ-Z2, the bootloader of the NEORV32 core was visible on GTKTerm. Next, the benchmark program executable was uploaded to the bootloader of the NEORV32 processor. This was done through the GTKTerm “Send raw file” file transfer operation. Once the file transfer finished, the benchmark could be executed through the NEORV32 bootloader. After the benchmark had completed, the output could be verified to the output from the software simulations. Picture 13 below shows the output of the benchmark running on the PYNQ-Z2 board.



Picture 13. PRNG benchmark running on the PYNQ-Z2.

5 Results

In the previous chapter, the overall methodology for completing the objectives of the thesis were laid out. This chapter will compare the performance metrics obtained for each version of the NEORV32 processor during execution of the PRNG benchmark.

5.1 Simulation results

To accurately gauge the performance optimizations that were gained in each version of the NEORV32 core, the following performance metrics were calculated for each version

- Cycle and instruction count per one round of the PRNG.
- Total cycle and instruction count for the complete benchmark.
- Change (%) per core version compared to the base RV32I version.
- CPI per round and CPI change (%).
- CPU time spent per round and CPU time change (%).
- Hardware utilization metrics when ported to the PYNQ-Z2.

On a further note, this chapter will not include graphs on the performance metrics of the hardware runs. Since both the simulation and FPGA versions of the NEORV32 core are built from the same VHDL code, the output values from both should match perfectly. This was verified to be the case during the testing phase when comparing all NEORV32 core versions.

5.1.1 Cycle and instruction count results

The cycle and instruction performance metrics, show how many overall clock cycles and instructions were consumed by the NEORV32 core during the benchmark execution. The cycle count shows how many clock cycles each version of the NEORV32 core spent in computing the benchmark. The

instruction counts shows how many RISC-V instructions were needed to compute the benchmark. Figure 4 below shows the cycle and instruction counts per one round of the PRNG benchmark.

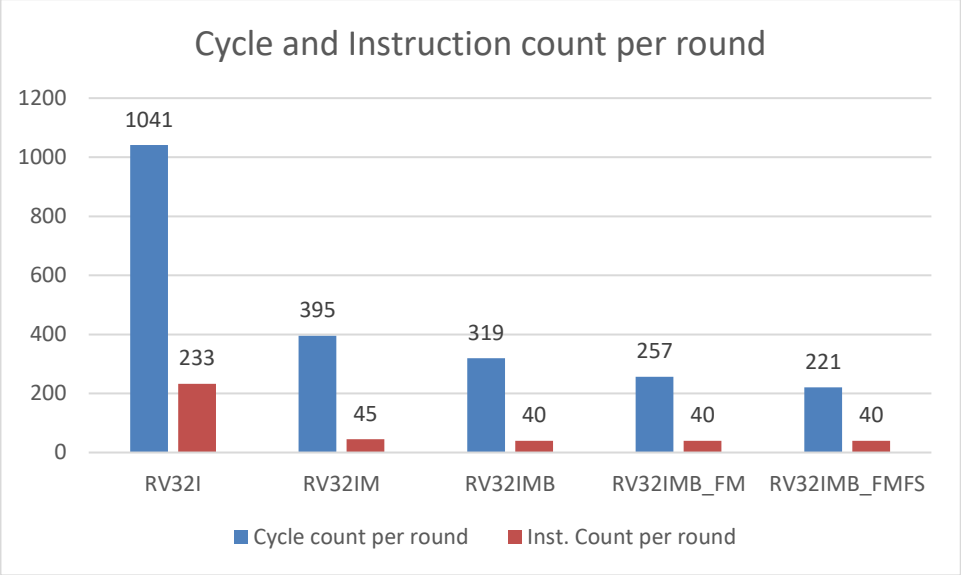


Figure 4. Graph of the cycle and instruction counts per PRNG round.

Looking at Figure 4, a large drop in the cycle and instruction counts between the RV32I version and the RV32IM version can be seen. This is mainly because to compute the multiplication operations contained in the benchmark, the RV32I version must emulate multiplication using only bit-shifting operations. Subsequent versions of the NEORV32 core all contain a small but measurable performance improvement. The instruction counts from the RV32IMB version onward are all equal, since the final two versions only enable optimizations based on FPGA hardware resources and not ISA modifications.

Figure 5 below shows the total number of cycles and instructions taken for running the whole benchmark or 25-rounds of the PRNG.

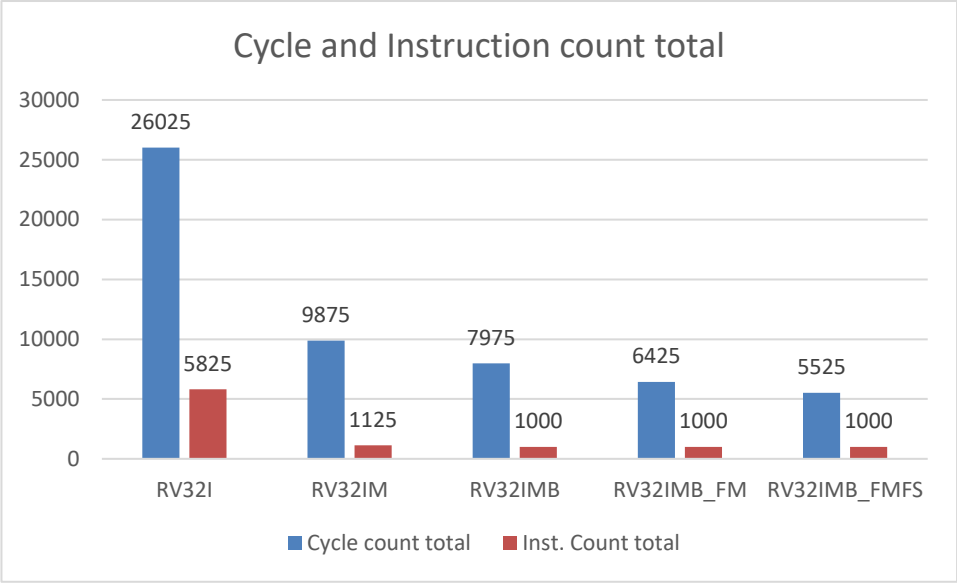


Figure 5. Graph of the cycle and instruction counts per PRNG benchmark.

Figure 6 below shows the change in performance per NEORV32 core compared to the base RV32I core version. The biggest drop is again from the RV32I core to the RV32IM core, with a cycle reduction of around 62% and an instruction count reduction of 81%.

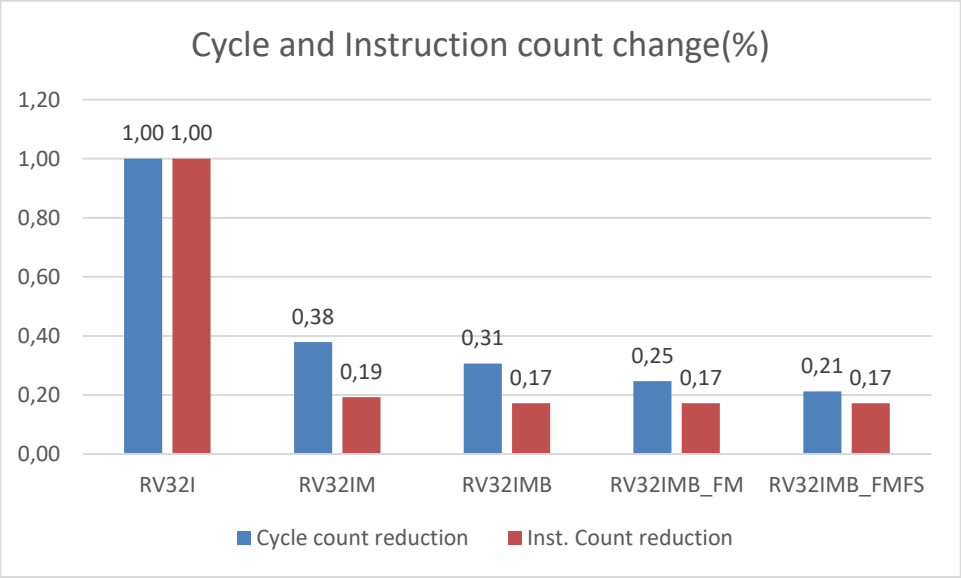


Figure 6. Graph of the change (%) in cycle and instruction counts.

5.1.2 CPI and CPU time results

Cycles per instruction (CPI), is the average number of cycles executed per instruction [51]. On a theoretical processor, if all of the instructions are executed in the exact same number of cycles, the CPI of the processor would equal to one. However, for most processors the number of cycles taken for different instructions varies between the instruction. For example, in the NEORV32 CPU, an addition operation will normally consume two clock cycles, while a multiplication operation will normally consume 36 cycles [27]. As such, the CPI can inform the user on the efficiency of the CPU implementation. Figure 7 below shows the calculated CPI per one round of the PRNG in blue and the change (%) of the CPI value in orange.

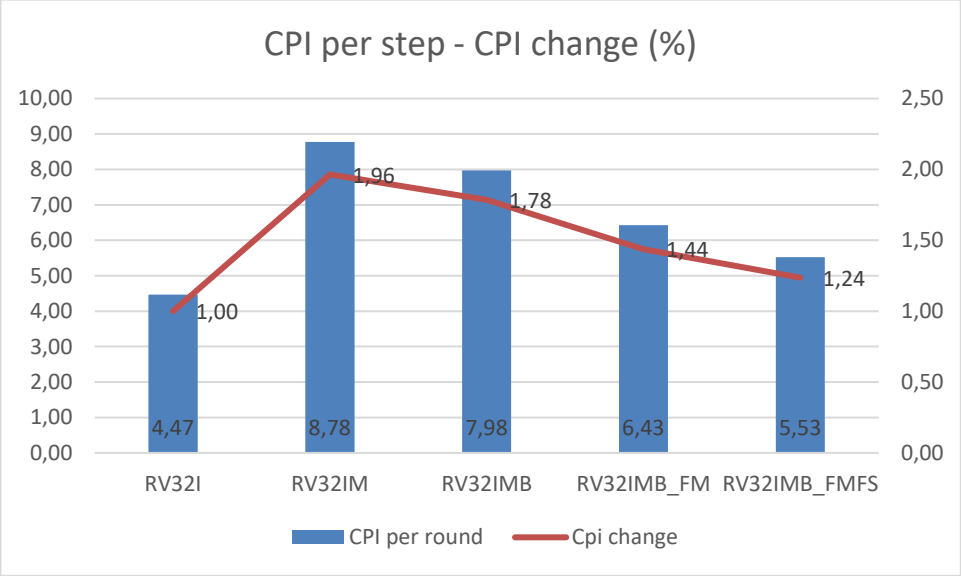


Figure 7. Graph of the calculated CPI values.

Looking at the graph in Figure 7, the RV32I core has the smallest CPI value of all versions tested. This is mainly because the RV32I core only implements the most basic instructions which all take a small number of cycles to execute. There is a large jump in the CPI values from the RV32I and the RV32IM version since the RV32IM version implements the multiplication operation, which on its own will consume 36 cycles for one instruction.

Figure 8 below shows the actual CPU time that was taken during the execution of one round of the PRNG.

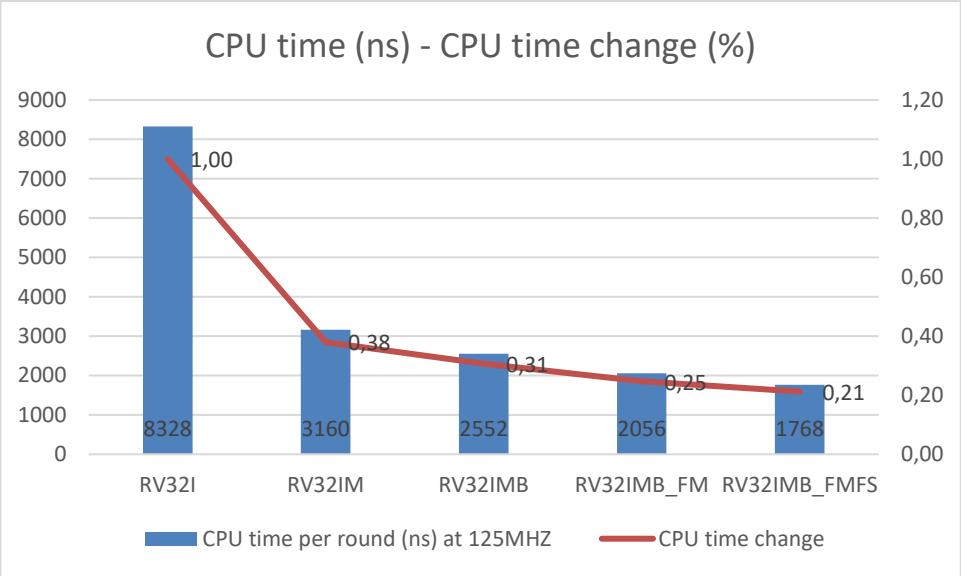


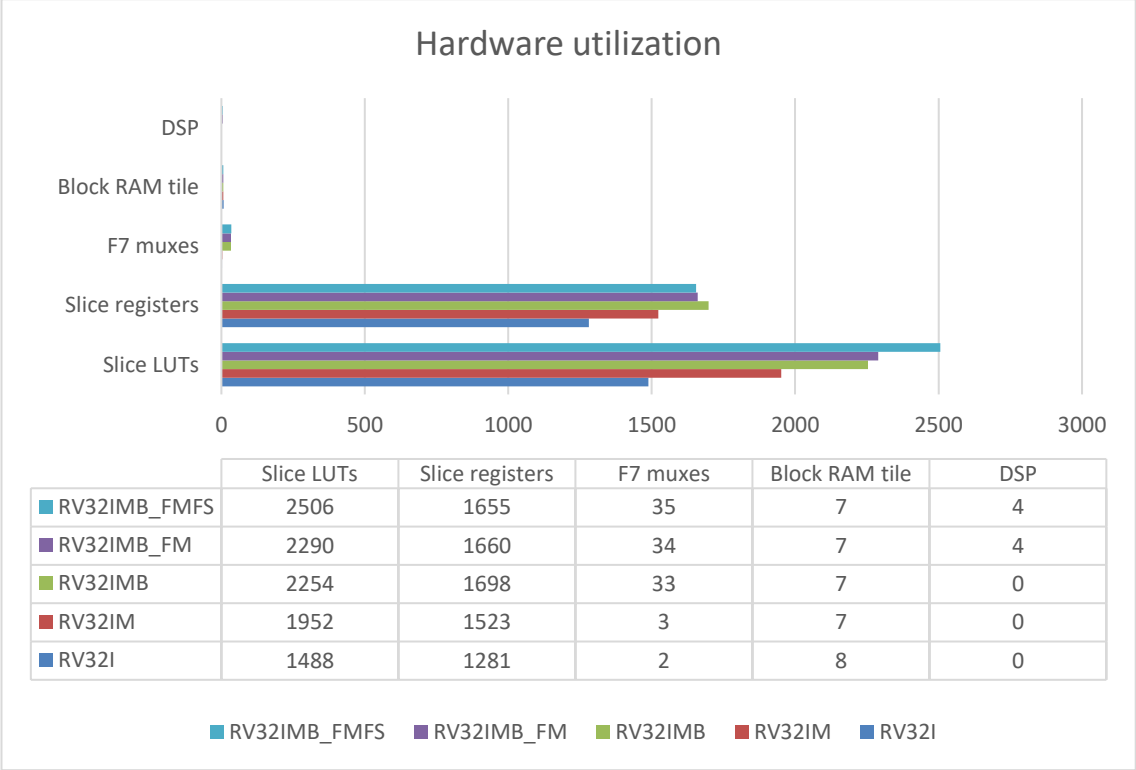
Figure 8. Graph of the calculated CPU time.

The CPU time graph shown in Figure 8, shows the number of nanoseconds each different version of the NEORV32 core took in running one round of the benchmark. The base RV32I version took 8.328 microseconds to compute one round, while the RV32IMB_FMFS core version only took 1.768 microseconds. This is almost an 80% drop in the amount of time taken when running one round of the PRNG benchmark.

5.2 Hardware results

In addition to comparing the performance metrics for each version of the NEORV32 core, it would also be prudent to see how the hardware utilization metrics stack up for each version. As each optional ISA extension, that is implemented in the processor, will consume a certain amount of hardware resources on the FPGA hardware, knowing the amount of performance improvements gained vs. the actual hardware cost of the implementation is a key metric. This information will advise the on-going development process on

the fulfillment of any key requirements in the microprocessor design. Figure 9 below shows the hardware utilization of each different version of the NEORV32 core on the PYNQ-Z2 FPGA board.



	Slice LUTs	Slice registers	F7 muxes	Block RAM tile	DSP
RV32IMB_FMFS	2506	1655	35	7	4
RV32IMB_FM	2290	1660	34	7	4
RV32IMB	2254	1698	33	7	0
RV32IM	1952	1523	3	7	0
RV32I	1488	1281	2	8	0

Figure 9. Graph of the hardware utilization of each NEORV32 version.

Looking at Figure 9, the hardware utilization figures jump the most when moving from the RV32I version to the RV32IM version. Slice look-up tables (LUTs) grow by around 31% and slice registers increase by around 19% between the RV32I and RV32IM version. F7 multiplexers (MUXs) grow from a value of three MUXs to a value of 33 MUXs when moving from the RV32IM version to the RV32IMB version. Finally, the RV32IMB_FM and RV32IMB_FMFS core versions, are the only versions to utilize DSP slices on the FPGA hardware, due to the faster multiplication optimization enabled for these two cores.

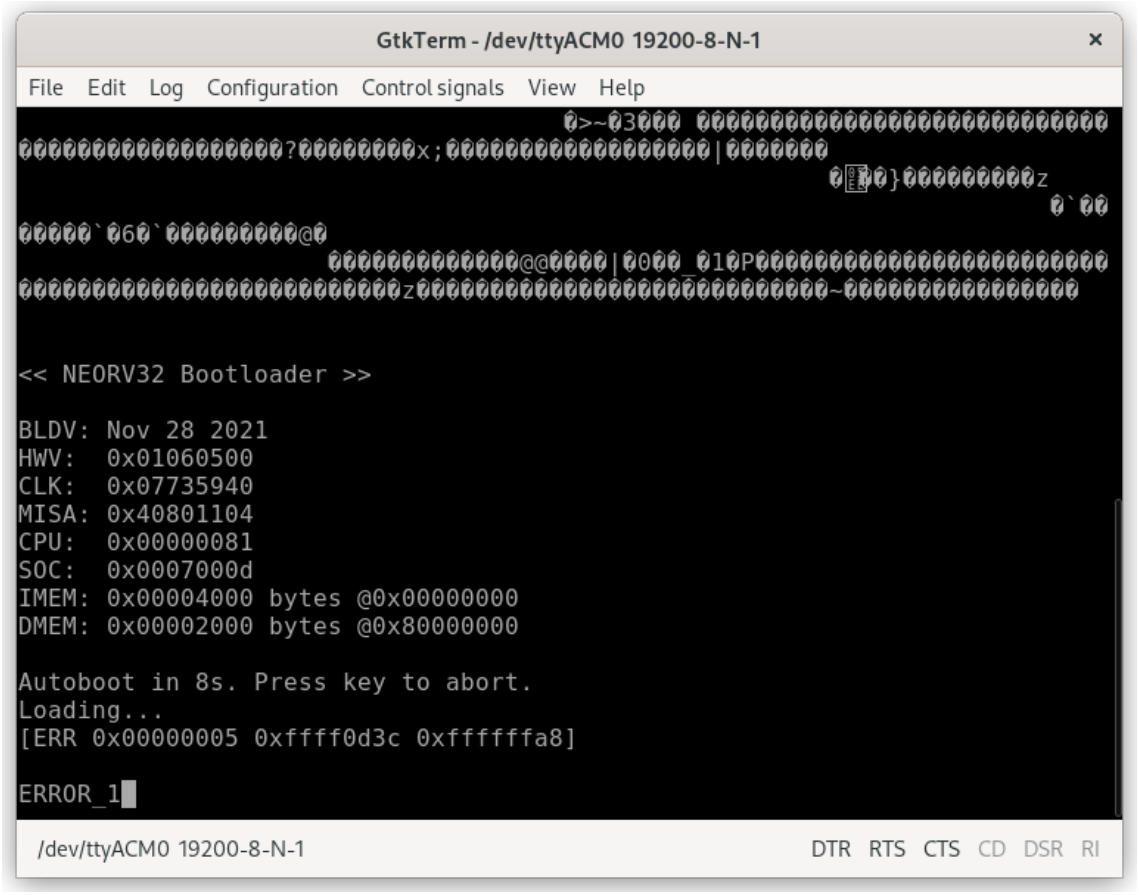
6 Discussion

Based on the findings of this thesis work, the testing methodology employed in this thesis shows how VHDL software simulations enable efficient and accurate testing of customized RISC-V cores. Employing software simulations in the development workflow for customizing microprocessors, allows for verifying performance optimizations without the need for implementation on actual hardware. This was shown to be the case in Chapter 5, where all of the performance metrics obtained from the PRNG benchmark were identical between the software simulation and the FPGA hardware implementation. The performance optimizations that were chosen for the different NEORV32 core versions, all show a successive improvement in performance of the PRNG benchmark when simulated through GHDL. A cycle count improvement of 79% and an instruction count improvement of 83%, was seen when comparing the base unoptimized NEORV32 core to the fully optimized version. The use of VHDL software simulations during performance testing will certainly allow for quicker turn-around times on achieving certain performance thresholds in microprocessor implementations.

In carrying out the testing methodologies, only one significant problem arose that affected the thesis work. Since the RISC-V toolchain does not yet include the newer GCC versions that support the B-extension, this thesis is unable to test the output compiled by the RISC-V toolchain on the RV32IMB version. The workaround used in the thesis was to implement the PRNG algorithm in RISC-V assembly language. As this code is hard coded into the benchmark program, there is no way to allow GCC to attempt to optimize the PRNG algorithm on the RV32IMB, RV32IMB_FM, and RV32IMB_FMFS core versions. This may have left out some small performance improvements to the last three core versions.

One small setback that was encountered, was during the implementation of the NEORV32 processor the PYNQ-Z2 FPGA board. After having compiled the NEORV32 processor in the Vivado IDE and uploaded it to the PYNQ-Z2 board, the bootloaders output was sent through the UART interface as unrecognizable

characters in GTKTerm. Picture 14 below shows the output from the broken UART transmissions.



Picture 14. An example of garbled UART data output from the NEORV32 processor.

The cause was found to be that the projects top design file was compiled with the default clock frequency of the NEORV32 processor, which is 100MHz. The natural clock frequency of the FPGA fabric on the PYNQ-Z2, on the other hand, is 125MHz. Although the problem was fixed after changing the clock frequency of the project to 125MHz, the exact cause for the garbled output was never fully identified. The most likely culprit might be a timing problem between the UART of the NEORV32 core and the receiving GTKTerm terminal. Since the UART protocol has no synchronized clock signal in use between the transmitter and receiver, the receiver tries to synchronize its internal clock by way of the transmitted signals start and stop bits. With a 25% difference between the

internal clock signals, the receiving terminal might be reading the wrong bits of the UART message, leading to the garbled output.

Some of the work that could still be done on this thesis, were laid out in Chapter 3 when setting out the requirements in Table 1. The same testing methodologies used on the NEORV32 project could easily be translated to a 64-bit RISC-V processor project. Furthermore, the testing of the C and E ISA extensions could be added to the thesis to gauge their effect on the performance metrics of the NEORV32 processor. The C-extension would add support for 16-bit long instruction formats that should speed up the fetching of multiple instructions from memory. This was left out of the thesis at this point, due to a prioritization on the ISA extensions that were theorized to have the greatest improvement on the performance of the PRNG benchmark. The E-extension or the Embedded ISA extension, lowers the number of general registers in the NEORV32 from 32 registers to 16 registers. This would yield a smaller processor footprint when implementing the processor on smaller hardware targets. This extension could, in theory, slow down the performance of the PRNG benchmark, if the number of registers needed to compute the algorithm efficiently exceeds the 16 set out in the E-extension. This was chosen not to be included since the E-extension is still a work-in-progress, and the results obtained from the version now available could change dramatically once the extension is fully ratified. Finally, in regard to the PRNG benchmark that was chosen for the performance benchmark, a more rigorous benchmark could have been chosen that would have stressed the processor more during its execution. This could have yielded more accurate performance metrics for each of the different core versions. For example, the NEORV32 contains a port of the Coremark CPU benchmarking suite and is used in the NEORV32 documentation for outlining CPU performance [27]. This was not implemented in the thesis though, due to one of the main motivations being performance gains on a specific type of workload. A pseudo-random number generator (PRNG) was chosen as the singular workload that would be improved throughout each NEORV32 core version.

7 Conclusion

As discussed in Chapter 1 of this thesis, one of the main cornerstones of software engineering is efficient and accurate testing during the development process [4]. This thesis has explored the use of VHDL software simulations in enabling efficient and accurate testing of customized RISC-V cores. The testing of customized RISC-V cores was taken from the perspective of implementing performance optimizations for a particular workload. The xoroshiro64** PRNG was chosen as this workload and a suitable benchmark program was then developed around this PRNG workload. The performance optimizations on the NEORV32 processor were developed in response to the workload that was chosen for the thesis. Different versions of the NEORV32 processor were customized with the inclusion of the I, M, and B ISA extensions. Also, two versions of the NEORV32 processor were customized to take advantage of the FPGA hardware on the PYNQ-Z2. The thesis is able to demonstrate a continuous performance improvement from the base NEORV32 core to the fully customized core. These improvements were in the neighborhood of an 80% performance gain on the PRNG workload. The performance metrics from each simulated NEORV32 core were then further verified with an actual hardware implementation. In each of the five core versions, the performance metrics from the simulated core matched exactly to the metrics from the hardware implementation. Overall, the results showcased in this thesis, support the main objectives set out at the start of Chapter 1: to show how VHDL software simulations can allow for efficient and accurate testing of customized RISC-V CPU cores.

References

[1] J. Hruska, “RISC vs. CISC Is the Wrong Lens for Comparing Modern x86, ARM CPUs,” *ExtremeTech*, 2021. https://www.extremetech.com/computing/323245-risc-vs-cisc-why-its-the-wrong-lens-to-compare-modern-x86-arm-cpus (accessed Feb. 25, 2022).

[2] ARM Blueprint, “Arm Partners Have Shipped 200 Billion Chips,” 2021. https://www.arm.com/blogs/blueprint/200bn-arm-chips (accessed Feb. 07, 2022).

[3] S. Matteson and M. Himelstein, “RISC-V: What it is, and what benefits it can provide to your organization ,” *TechRepublic*, 2020. https://www.techrepublic.com/article/risc-v-what-it-is-and-what-benefits-it-can-provide-to-your-organization/ (accessed Feb. 25, 2022).

[4] IBM, “What is Software Testing and How Does it Work?,” *IBM*, 2020. https://www.ibm.com/topics/software-testing (accessed Feb. 25, 2022).

[5] M. Lapedus, “Big Trouble At 3nm,” *Semiconductor Engineering*, 2018. https://semiengineering.com/big-trouble-at-3nm/ (accessed Feb. 26, 2022).

[6] K. Patsidis, C. Nicopoulos, G. C. Sirakoulis, and G. Dimitrakopoulos, “RISC-V2: A Scalable RISC-V Vector Processor,” in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5. doi: 10.1109/ISCAS45731.2020.9181071.

[7] B. Marshall, D. Page, and T. Hung Pham, “A lightweight ISE for ChaCha on RISC-V,” in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2021, pp. 25–32. doi: 10.1109/ASAP52443.2021.00011.

[8] M. Alizadeh and M. Sharifkhani, “Extending RISC- V ISA for Accelerating the H.265/HEVC Deblocking Filter,” in *2018 8th International Conference on Computer and Knowledge Engineering (ICCKE)*, 2018, pp. 126–129. doi: 10.1109/ICCKE.2018.8566467.

[9] A. Rushton, *VHDL for Logic Synthesis*. New York, UNITED KINGDOM: John Wiley & Sons, Incorporated, 2011. [Online]. Available: http://ebookcentral.proquest.com/lib/turkuamk-ebooks/detail.action?docID=675308

[10] S. P. Ritpurkar, M. N. Thakare, and G. D. Korde, “Design and simulation of 32-Bit RISC architecture based on MIPS using VHDL,” *ICACCS 2015 - Proceedings of the 2nd International Conference on Advanced Computing and Communication Systems*, Nov. 2015, doi: 10.1109/ICACCS.2015.7324067.

[11] D. Harris, S. L. Harris, D. Harris, and S. L. Harris, *Digital Design and Computer Architecture : From Gates to Processors*. Burlington, UNITED STATES: Elsevier Science & Technology, 2007. [Online]. Available: http://ebookcentral.proquest.com/lib/turkuamk-ebooks/detail.action?docID=404196

[12] J. Ganssle, “The shape of the MCU market ,” *Embedded.com*, 2016. https://www.embedded.com/the-shape-of-the-mcu-market/ (accessed Mar. 05, 2022).

[13] T. Noergaard, *Embedded Systems Architecture : A Comprehensive Guide for Engineers and Programmers*. Burlington, UNITED STATES: Elsevier Science & Technology, 2005. [Online]. Available: http://ebookcentral.proquest.com/lib/turkuamk-ebooks/detail.action?docID=294593

[14] A. Elahi, *Computer systems: Digital design, fundamentals of computer architecture and assembly language*. Springer International Publishing, 2017. doi: 10.1007/978-3-319-66775-1.

[15] M. Maxfield, “What the FAQ are CPUs, MPUs, MCUs, and GPUs?,” *EEJournal*, 2019. https://www.eejournal.com/article/what-the-faq-are-cpus-mpus-mcus-and-gpus/ (accessed Mar. 05, 2022).

[16] ARM Developer, “ARM Cortex-A Series Programmer’s Guide for ARMv8-A,” *ARM Developer*, 2022. https://developer.arm.com/documentation/den0024/a/Caches/Cache-terminology (accessed May 09, 2022).

[17] J. P. Mueller and L. Massaron, “The Von Neumann Bottleneck’s Affect on Artificial Intelligence,” *Dummies*, 2018. https://www.dummies.com/article/technology/information-technology/ai/general-ai/von-neumann-bottlenecks-affect-artificial-intelligence-254223/ (accessed May 09, 2022).

[18] R. Ritchie, “Apple A14 Bionic Explained — From iPad Air to iPhone 12 | iMore,” *iMore*, Sep. 28, 2020. https://www.imore.com/apple-a14-bionic-explained-ipad-air-iphone-12 (accessed May 10, 2022).

[19] D. A. Patterson and J. L. Hennessy, *Computer organization and design RISC-V edition : the hardware/software interface*, 1. ed. Morgan Kaufmann/Elsevier, 2018.

[20] D. A. Patterson and D. R. Ditzel, “The Case for the Reduced Instruction Set Computer,” 1981.

[21] S. Heule, “How Many x86-64 Instructions Are There Anyway?,” Mar. 07, 2016. https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/ (accessed May 13, 2022).

[22] Devopedia, “RISC-V Instruction Sets,” *Devopedia*, 2018. https://devopedia.org/risc-v-instruction-sets (accessed May 13, 2022).

[23] S. Thornton, “RISC vs. CISC Architectures: Which one is better?,” *Microcontroller Tips*, 2018. https://www.microcontrollertips.com/risc-vs-cisc-architectures-one-better/ (accessed May 13, 2022).

[24] A. Waterman, Y. Lee, D. Patterson, and K. Asanovi, “The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA The RISC-V Instruction Set Manual Volume I: Base User-Level ISA Version 1.0,” 2011, Accessed: May 13, 2022. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html

[25] RISC-V International, “History,” 2022. https://riscv.org/about/history/ (accessed May 13, 2022).

[26] RISC-V Foundation, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, 2019.

[27] S. Nolting *et al.*, “The NEORV32 RISC-V Processor: Datasheet,” *GitHub pages*, 2022. https://stnolting.github.io/neorv32/ (accessed Jan. 27, 2022).

[28] IEEE, “IEEE Standard VHDL Language Reference Manual,” *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pp. 1–640, 2009, doi: 10.1109/IEEESTD.2009.4772740.

[29] M. Zwoliński, *Digital system design with VHDL*, 2nd ed. Prentice Hall, 2004.

[30] H. Amano, *Principles and Structures of FPGAs*. Springer Singapore, 2018.

[31] Pynq, “PYNQ - Python productivity for Zynq - Board,” *Pynq.io*, 2022. http://www.pynq.io/board.html (accessed May 14, 2022).

[32] A. Arobelidze, “Random Number Generator: How Do Computers Generate Random Numbers?,” *Freecodecamp*, 2020. https://www.freecodecamp.org/news/random-number-generator/ (accessed May 14, 2022).

[33] Maxim Integrated, “Random Number Generation Using LFSR ,” *Maxim Integrated*, 2010. https://www.maximintegrated.com/en/design/technical-documents/app-notes/4/4400.html (accessed May 14, 2022).

[34] D. Blackman and S. Vigna, “Scrambled Linear Pseudorandom Number Generators,” *ACM Trans. Math. Softw.*, vol. 47, no. 4, Sep. 2021, doi: 10.1145/3460772.

[35] D. Blackman and S. Vigna, “xoshiro/xoroshiro generators and the PRNG shootout,” 2022. https://prng.di.unimi.it/ (accessed Feb. 02, 2022).

[36] D. Haughey, “MoSCoW Prioritisation Method,” *Projectsmart*, 2021. https://www.projectsmart.co.uk/tools/moscow-method.php (accessed May 14, 2022).

[37] S. Nolting *et al.*, “The NEORV32 RISC-V Processor,” *GitHub repository*. GitHub, 2022. Accessed: Jan. 27, 2022. [Online]. Available: https://github.com/stnolting/neorv32

[38] GHDL, “GHDL: VHDL 2008/93/87 simulator.” Github, 2022. Accessed: Feb. 01, 2022. [Online]. Available: https://github.com/ghdl/ghdl

[39] B. Bockholt, “Left rotation,” *Dictionary of Algorithms and Data Structures*, 2020. https://xlinux.nist.gov/dads/HTML/leftrotation.html (accessed May 15, 2022).

[40] RISC-V Software Collaboration, “riscv-gnu-toolchain: GNU toolchain for RISC-V, including GCC,” *Github*, 2022. https://github.com/riscv-collab/riscv-gnu-toolchain (accessed Feb. 04, 2022).

[41] Xilinx, “Downloads,” *Xilinx.com*, 2022. https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools.html (accessed May 15, 2022).

[42] GTKWave, “GTKWave,” *Sourceforge*, 2022. http://gtkwave.sourceforge.net/ (accessed Feb. 04, 2022).

[43] Fedora, “Fedora 35 User Documentation,” *Fedora Docs*, 2021. https://docs.fedoraproject.org/en-US/fedora/f35/ (accessed May 15, 2022).

[44] Git, “Git,” 2022. https://git-scm.com/ (accessed May 15, 2022).

[45] RISC-V International, “RISC-V Bit-Manipulation ISA-extensions,” Version 1.0.0-38-g865e7a7, 2021.

[46] GCC Team, “GCC 12 Release Series — Changes, New Features, and Fixes - GNU Project,” *GNU Project*, 2022. https://gcc.gnu.org/gcc-12/changes.html (accessed May 15, 2022).

[47] S. Nolting *et al.*, “The NEORV32 RISC-V Processor: User guide,” *GitHub pages*, 2022. https://stnolting.github.io/neorv32/ug/ (accessed Jan. 27, 2022).

[48] GNU Project, “Make,” 2022. https://www.gnu.org/software/make/ (accessed Feb. 04, 2022).

[49] Pynq, “PYNQ Introduction — Python productivity for Zynq,” *Pynq Documentation*, 2022. https://pynq.readthedocs.io/en/latest/index.html (accessed May 16, 2022).

[50] GCC Team, “Basic Asm (Using the GNU Compiler Collection (GCC)),” *GCC documentation*, 2022. https://gcc.gnu.org/onlinedocs/gcc/Basic-Asm.html#Basic-Asm (accessed May 16, 2022).

[51] W. Stallings, *Computer organization and architecture : designing for performance*, 8th ed. Pearson Education, 2010.