IT 19 045

Examensarbete 15 hp
Juli 2019

Open Hardware: Initial Experiences with Synthesizing Open Cores

Daniel Degirmen

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Open Hardware: Initial Experiences with Synthesizing Open Cores

Daniel Degirmen

An instruction set architecture (ISA) is an abstract interface between the hardware and the lowest level software, describing the operations a processor must be able to execute. Most commercially successful ISAs are propitiatory, meaning that hardware designers are limited to the design and have to pay for a license if they wish to use it. A new open-source ISA called RISC-V has emerged in order to allow for designers to design and implement their own processors for free. This thesis explores RISC-V as well as the new open-source hardware description language Chisel.

A 32 bit RISC-V core nicknamed the "Anhyzer core" is designed and implemented, showing how one can model a RISC-V core in Chisel. The core is limited to the RV32I instruction set, and it was simulated using the hardware simulator Verilator in order to make sure it was functional. This was done by loading instructions into the instruction memory and then running the simulation. The core was able to execute the RV32I instructions, but since it a single-cycle CPU the throughput was low. Experiments showed that there was no difference in performance between theoretically slower instructions and faster ones. The core serves as a proof of concept which can be expanded upon in the future.

Handledare: Stefanos Kaxiras
Ämnesgranskare: Alexandra Jimborean
Examinator: Johannes Borgström
IT 19 045
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

When a programmer writes a computer program, it is most often done in what is called a *high-level programming language* such as Java, Python or C++. These types of programming languages provide a higher abstraction from the details of the computer, in contrast with *low-level programming languages* (assembly). A high level language allows the programmer to write human-readable code which uses elements of natural languages. However, the electronic circuitry within the computer that carries out the instructions of the written program, the *central processing unit* (CPU), can not process high-level languages directly. Instead, the CPU can execute the so called *machine code*, which has been compiled (translated) from the source code written by the programmer into binary. The part of the CPU which represents the basic computation unit is referred to as the *core*.

The CPU executes *instructions*, such as arithmetic and logical operations. These instructions are represented by a sequence of bits, and it is the *instruction set architecture* (ISA) that determines how the instructions are patterned. As such, there are several different ISAs and different systems which only understands a specific architecture. This leads to the compiler having to translate the source code into the machine code for the corresponding architecture. Thus the compiler has to know which processor or architecture that it should target.

Currently, all ISAs that are commercially popular are proprietary such as the x86 family of ISAs designed by Intel and AMD[1]. Companies such as IBM and ARM have patents on certain ”quirks” in their ISAs which prevents others from using them without a license. It may take a long time in order to acquire one, and the cost is in the range of \$5 to \$10 million USD. This rules out academia and others who may not be able to afford it. Additionally, an ARM license for example does not allow one to create own designs, but instead locks the license holder to the company's own design (some exceptions exists). This prevents competition and innovation as it stops many from designing and sharing their own ISA-compatible cores[2].

As a response to this, the architecture research group at University of California, Berkeley has created their own open source instruction set architecture called *RISC-V*. It is completely free, and allows anyone to create their own designs. Already a collaborative community exists for RISC-V and it is growing continuously.

1.1 Aim of the report

This report explores RISC-V and a new way to describe hardware, more precisely, using the *hardware description language* (HDL) *Chisel*. The report aims to answer the following questions:

- What is the state of the art regarding open hardware.
- How can we model RISC-V using Chisel?
- How does RISC-V and Chisel differ from previous work.

Another goal is to explore the possibility of designing and implementing a CPU based on open specifications. It will then be simulated using an HDL simulator (Verilator) in order to test that it functions correctly. The CPU should be able to carry out the chosen instructions to be implemented as defined in the RV32I set (see section 4). Thus it has to be designed in a way to allow for the different instructions to be carried out. In order to explore the new methodology of software-inspired hardware creation, a processor for the RISC-V architecture is designed and constructed using Chisel3. Our hope is that this thesis report can function as an introduction and a foundation for anyone who wishes to learn more about RISC-V.

2 Background

2.1 Instruction set architecture

The instruction set architecture (ISA) is an abstract interface between the hardware and the lowest level software[12]. It describes the design of a computer in terms of the basic operations it must support. The ISA defines the set of instructions that the processor understands. These instructions can vary from simple instructions that perform addition, to more complex ones such as vector instructions. Many different architectures exist on the market, such as MIPS, Power, Sparc, ARM, x86 and now more recently RISC-V.

2.2 Open-source hardware

When it comes to hardware, there have been attempts to create something similar to the way open-source software works. Open hardware can be described as "design specifications of a physical object which are licensed in such a way that said object can be studied, modified, created, and distributed by anyone"[20]. The first open-source hardware initiatives were made in 1997 by Bruce Perens[21]. Perens created the *Open Hardware Certification Program* with the goal of allowing hardware manufacturers to certify their products as open[20]. By the mid 2000s, open hardware such as the Arduino, OpenCores and other projects emerged which created a renewed interest in open-source hardware. When it comes to ISAs, RISC-V is not the first open-source instruction set architecture. Others have been developed long before it, and an initial idea for the researchers at Berkeley was to build their ISA upon an existing one[4]. However the researchers decided not to do it because of several technical reasons. Using a clean slate allows for a design that meets the goals that were set[17]. Among the other ISAs, the most popular and still actively developed is OpenRISC which was started in the year 2000 and released under the GNU general public license. An open source version of the 32- and 64 bit version of MIPS is also in development, called *MIPSOpen*[17].

3 RISC-V

RISC-V is a general purpose open source hardware instruction set architecture developed by the architecture research group at University of California, Berkeley, and was originally designed to support computer architecture research and education[3]. The project started in 2010 by researchers at the university, but many contributions to it have come from external sources not affiliated with the university[5]. Today the RISC-V foundation, which is a collaborative community of software and hardware innovators, consists of over 200 members. Their board of directors consist of world leading companies such as Google and NVIDIA.

3.1 Motivation behind RISC-V

Technology is rapidly changing the world, and one of the key reasons for this is open-source software which is facilitating software innovations[15]. Open-source software has allowed universities, well-established companies and startups to begin with an improved software infrastructure. An example of this is the LAMP-stack (Linux, Apache, MySQL, and PHP) which provides services over the web[7]. It is especially important for startups since it allows for starting the companies with much of the software already in place. Each company can then modify and build upon the existing software in order to create their own brands (for example Facebook, the popular social media platform).

While the open-source software community has been fully embraced, the same is not true for open-source hardware. Many of the benefits that open-source software provides can not be taken advantage of by hardware-software designers. The hardware aspects of systems are proprietary, meaning that a designer would need to spend time and money to license hardware components and design tools. As such, the venture capital needed to bring a hardware-software product on the market is much greater than a software-only product[7]. This effectively shuts out many from the market.

Using an existing commercial ISA provides some benefits. It has a large and widely supported software ecosystem (development tools and ported applications). Commercially successful ISAs also provide more detailed documentation and examples. However, there are several disadvantages of this which lead to the creation of RISC-V[5]. Among these are:

- **Commercial ISAs being proprietary.** Companies protect their intellectual property, and do not welcome freely available competitive implementations. This results in not being able to share register-transfer level (RTL) implementations. It also shuts out groups who do not want to trust the few providers of commercial ISA implementations.

- **Commercial ISAs being popular only in certain market areas.** An example of this is that the ARM architecture is not well supported in the server area. The Intel x86, among most other architectures, are not well supported for mobile. Others such as ARC focus on the embedded space. This leads to a market segmentation where the benefits of supporting one ISA lessens.

- **Popular ISAs are complex.** The two most popular ISAs (ARM and x86) are complex to implement in hardware due to bad/outdated design decisions. RISC-V instead opts to be a simpler ISA.

- **Commercial ISAs alone are not enough to bring up applications.** Most applications need a complete *application binary interface* (ABI) which is an interface between two binary program modules. Usually one of these modules is a library or operating systems facility and the other is a program run by the user. The ABI is reliant on libraries which are reliant on operating system support. In order to run an operating system, it requires implementing the supervisor-level ISA and device interfaces which is more complex to implement than the user-level ISA.

- **Commercial ISAs are not designed to be expanded.** The current ISAs are not designed for extensibility, and as the instruction set has expanded, the instruction encoding complexity has grown as a consequence.

- **Modifying an ISA creates a new one.** One of the main goals of RISC-V is to support architecture research which includes expanding the ISA. Even if an ISA is modified slightly, this introduces new issues as compilers have to be modified and applications have to be rebuilt to use these extensions.

The authors of RISC-V suggest that "the ISA is perhaps the most important interface in a computing system" hence there is no reason why it should be proprietary[5]. As such, in order to solve many of the problems listed above, RISC-V was created. Instead of having a proprietary ISA, RISC-V allows designers to create hardware and software designs without paying royalties to the creators. It is a frozen ISA, meaning that software can be developed once and run indefinitely on a RISC-V device[8]. This provides a stability for software developers as software written for RISC-V will be able to run on similar RISC-V cores forever[9]. While RISC-V was originally created to support research and education in computer architecture, it now aims to become a standard open architecture for industry application.

Some of the goals with RISC-V includes:

- Being an open ISA for academia and industry.
- Having support for highly parallel multi-/manycore implementations
- Being an ISA that is suitable for hardware implementation and not just simulations
- To include 32- and 64-bit address space variants for applications, operating system kernels and hardware implementations

3.2 Technical specifications

RISC-V is a load-store ISA based on *reduced instruction set computer* (RISC) principles. RISC designs aim to provide an ISA with instructions that do simple operations[4]. The simplicity of RISC designs allows the processor to execute an instruction quickly, typically taking only one clock cycle. Load-store ISAs divide instructions into two categories:

- *Memory access*, where loading and storing happens between the memory and registers. Only load and store instructions access the memory.
- *ALU-operations*, where the operands are registers.

RISC-V is defined as a base integer ISA which must be present in any implementation. Other optional extensions to the base ISA can be made. RISC-V is designed such that the base integer instructions cannot be redefined, but the base ISA can be extended with optional instructions. This allows for customization of the ISA to suit a particular need. The extensions are divided into two categories: *standard* and *non-standard* extensions. Standard extensions do not conflict with other extensions, while non-standard are either highly specialized or may conflict with (non-) standard extensions. RISC-V defines a set of standard instruction extensions in order to support software development. They provide integer multiply/divide, atomic operations and single/double precision floating point arithmetic operations. This provides modularity, allowing designers to choose which extensions to implement. The modules/extensions are:

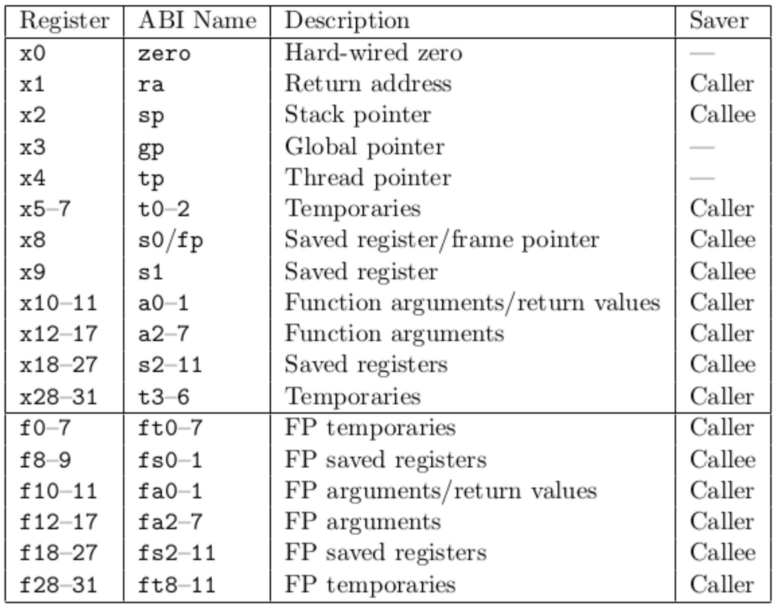
- *I* - The base integer ISA (prefixed with RV32/64, depending on the register width). It contains integer computational instructions, loads, stores and control-flow instructions.
- *M* - Provides multiplication and division. It is the standard multiply and divide extension of RISC-V.
- *A* - Atomic operations. These are atomic read, modify and memory writes for inter-process synchronization.

- *F* - Standard single precision floating point operations. This extension adds floating point registers and single precision floating point operations such as loads and stores.
- *D* - Expands the floating point registers and adds double precision floating point operations.

Note that all extensions are optional to implement, with the exception for the base integer ISA (I). Other extensions are also being worked on, such as the B extension which provides bit manipulation. Vector instructions will also be implemented in the future with the V extension.

3.2.1 Registers

RISC-V has 32 registers of which 31 are general-purpose registers (see Figure 1). There is also one additional register, *the program counter* (pc), that holds the address of the current instruction. The registers hold integer values, with register x0 (zero) hardwired to the constant 0. In RV32 the registers are 32 bits wide, and in RV64 they are 64 bits. 32 floating-point registers can be added as an extension. A 128 bit variant is also in development, but has not been frozen yet[4]. Table 1 describes the registers, as well as the calling convention of RISC-V.



Register	ABI Name	Description	Saver
`x0`	`zero`	Hard-wired zero	—
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	—
`x4`	`tp`	Thread pointer	—
`x5–7`	`t0–2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10–11`	`a0–1`	Function arguments/return values	Caller
`x12–17`	`a2–7`	Function arguments	Caller
`x18–27`	`s2–11`	Saved registers	Callee
`x28–31`	`t3–6`	Temporaries	Caller
`f0–7`	`ft0–7`	FP temporaries	Caller
`f8–9`	`fs0–1`	FP saved registers	Callee
`f10–11`	`fa0–1`	FP arguments/return values	Caller
`f12–17`	`fa2–7`	FP arguments	Caller
`f18–27`	`fs2–11`	FP saved registers	Callee
`f28–31`	`ft8–11`	FP temporaries	Caller

Table 1: RISC-V registers (and added floating point registers), with ABI names and calling convention.[4]

3.2.2 Instruction format

In the base ISA, there are 4 instruction formats R, I, S and U. These are core instruction formats that have a fixed length of 32 bits which must be aligned on a four-byte boundary. Two additional variants are the B and J instruction formats. These instructions handle immediates (constant values embedded in the instruction). *R-types* are register-register instruction. These instructions uses two registers (rs1 and rs2) as operands and then writes the result back to register rd. An example of an R-type instruction is addition (*add*). *I-types* uses register rs1 and a 12 bit immediate that is sign-extended as the operands. An example is the slli instruction which encodes a logical left shift by the immediate value. Load instructions and jump-and-link (jalr) are also encoded as I-type instructions. *S-types* are used for store instructions, where the immediate is added to register rs1 in order to get the offset. *U-types* are used for the LUI and AUIPC instructions which will be further explained later. (Unconditional) jumps are encoded as *J-type* instructions. Jump instructions add the sign extended immediate to the address from the program counter in order to get the address to jump to. *B-types* encode branch instructions, where the immediate is added to the address from the program counter. See Figure 1 for the different formats.

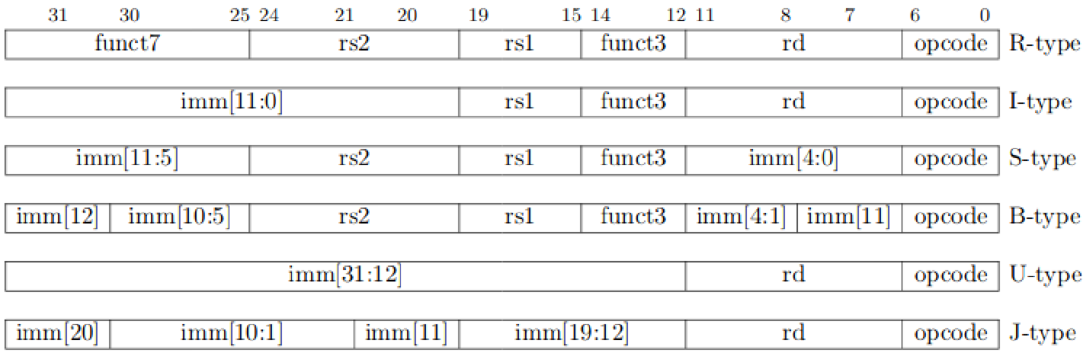

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

Figure 1: The instruction format. Each instruction is 32 bits in length. *rs* are source registers and *rd* is the destination register.[4]

3.2.3 Addressing

The address space is byte-addressable and little-endian[6]. Byte addressing means that each byte can be accessed individually, in contrast with word-addressable where a whole word (typically 4 or 8 bytes) is fetched. Loading and storing words are also possible. Endianness refers to how a sequence of bytes is stored in memory. Little endian means that the least significant byte is stored first (see Figure 2).

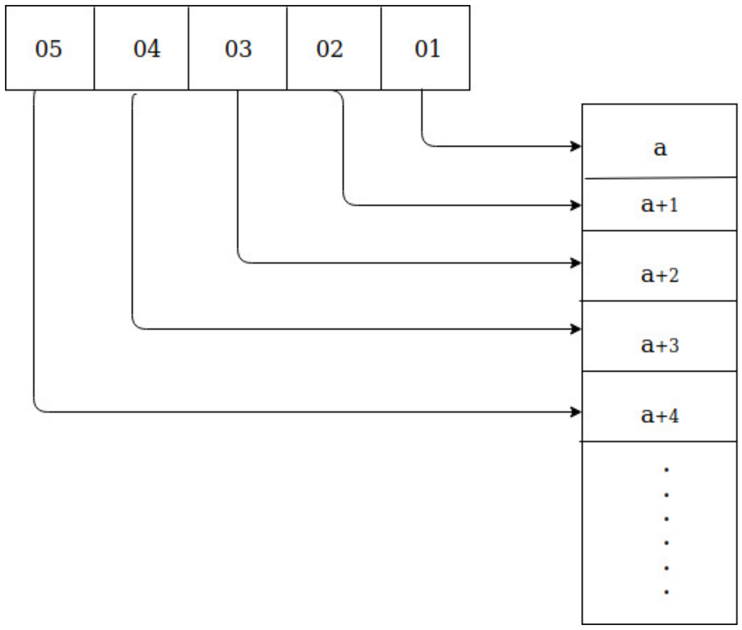
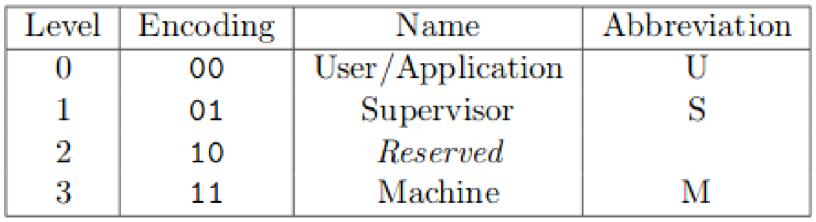


Figure 2: Little endian storage. The least significant byte (01) is stored first in memory. This in contrast with big endian where the most significant byte (05) would be stored first.

3.2.4 Privilege levels

RISC-V currently has 3 privilege levels. The privilege levels control the access of a program running on the CPU to resources such as memory[10]. The levels are the *user/application*, *supervisor* and *machine* levels (see Figure 2). The machine level is the highest priority level. It is also the only obligatory privilege level for RISC-V implementations, and code run in this mode has low-level access to the machine. As such it usually inherently trusted. User mode is the lowest level, and is used for applications while the supervisor mode is for operating system usage.



Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

Table 2: RISC-V privilege levels.

3.3 Chisel

Historically, the two most popular hardware description languages are Verilog and VHDL[16]. These languages allows hardware designers to describe the structure and behaviour of electronic circuits. Both Verilog and VHDL were originally created for hardware simulation, but were later adopted for synthesis[19]. The languages lack the abstraction facilities found in modern software languages, making it difficult to reuse components. This ultimately leads to lower designer productivity[13].

Chisel, an acronym for **C**onstructing **H**ard-ware **I**n a **S**cala **E**mbedded **L**anguage, is an open-source HDL created by the architecture research group at University of California, Berkeley. It is embedded inside the Scala programming language, and provides a high level of hardware design abstraction[13]. As such, when a user writes a Chisel program, they are in fact writing a Scala program that creates a hardware graph[14]. Chisel provides programming language concepts such as object orientation, functional programming, parameterized types, and type inference. It can be compiled into Verilog code, and the design can then be simulated in software or synthesized on an *field programmable gate array* (FPGA) or an *application-specific integrated circuit* (ASIC). Chisel can also be used to generate a software simulation based on C++.

By being embedded in Scala, Chisel can take advantage of many features present in Scala such as libraries or the fact that it compiles to the Java virtual machine. A key reason for embedding Chisel in Scala is also to support highly parameterized circuit generators which are not present in traditional HDLs. The abstraction found in Chisel allows designers to reuse objects and functions as well as define their own data types. It also abstracts away features such as clocks and resets, which must be explicitly defined in languages such as Verilog. Instead these signals are global signals. Since these components are so common, Chisel automatically creates them for the user whenever they are needed (for example when creating a register). As such Chisel makes the power of modern software programming languages available for hardware design[13].

3.3.1 Differences

Chisel allows basic data types to be aggregated into bundles in order to create new data types or ease usage of bundled signals. These bundles can be subclassed to create a hierarchical structure, allowing for code reuse. Bundles can be used in interfaces which can be connected making writing busses and/or whole interfaces easy[18]. VHDL provides similar functionality but it does not support subclassing. Chisel and VHDL support operator overloading, but Chisel automatically infers bit width at compile time.

Functions exist in both Chisel and VHDL, but Chisel allows for parameterized functions. These functions can handle all data of a super class, and this polymorphism is not available in VHDL. Instead the hardware designer must write the same function for each of the data types that are used. Chisel also allows for parameterized classes, making them more reusable[20]. This type of polymorphism has not been seen before in hardware design and is very powerful, as it allows for creation of generic entities. Parameterization also allows for easy changes to a design, effectively creating a new version of it.

In Verilog, one can use the `readmemh` function to instantiate memories by reading hex-values from a file. Chisel currently lacks similar functionality, but an experimental function called `loadMemoryFromFile` exists and is based on readmemh. Since it is experimental it is not as stable as the Verilog function and is susceptible to changes or even removal.

4 CPU development

4.1 Design

In order to implement the base integer instruction set for RISC-V, the RV32I set, a design that allows for all instructions to be implemented was created (see Figure 3). Nicknamed the "Anhyzer core", the design takes inspiration from the book *Computer Organization and Design - The Hardware Software Interface: RISC-V Edition* by David A. Patterson[12], using the design as a foundation that was expanded upon. This design is a *single-cycle* design, meaning that only one instruction can be processed every clock cycle. Each instruction begins execution on one clock edge, and finishes on the next. The following sections describes how it is designed and implemented, as well as how one can simulate it.

4.1.1 Design choices

The choice for the design was based on a number of factors. As a modern CPU is extremely complicated, the design was simplified in order to keep the complexity to a minimum. This was deliberately made so that anyone with a basic understanding of hardware design can understand how the CPU works. The choice for the design was also made so that it functions as a base that can be expanded upon. The CPU is modular so that each component handles a specific task. For example, the *arithmetic logical unit* (ALU) does not handle branches or jumps. This was instead split up into different units which handle branches and jumps separately. As the RISC-V instruction set keeps expanding, if more instructions were to be added, the modularity allows for easier expansion of the supported instructions. This was inspired by the way software is designed in the modular programming paradigm.

The Anhyzer core uses two separate models for memory, one for the instructions and one for the data. While this is not necessary in a single-cycled design, since there are no structural hazards, the choice to split up the entities was still made. The reason for this was that in a pipelined design, the memories are split up since the instruction fetch stage accesses the instruction memory and the memory read/write stage accesses the data memory. As such if one were to pipeline the Anhyzer core, this would be easier as the memories are already separated. Another reason was because of the software design principle of *separation of concerns*. As each module is designed to handle one task (i.e store/load data or instructions), modifying one of them will not affect the other.

To sum up the design choices, some of the advantages of the chosen design includes:

- Simple (reduced complexity).
- No data/structural or control hazards which one might get if the CPU is pipelined.

- Serves as a base that can be further expanded to suit different needs.

Some of the disadvantages are:

- Low throughput since only one instruction can be executed each cycle.
- Clock cycle must be stretched to accommodate the slowest instruction.

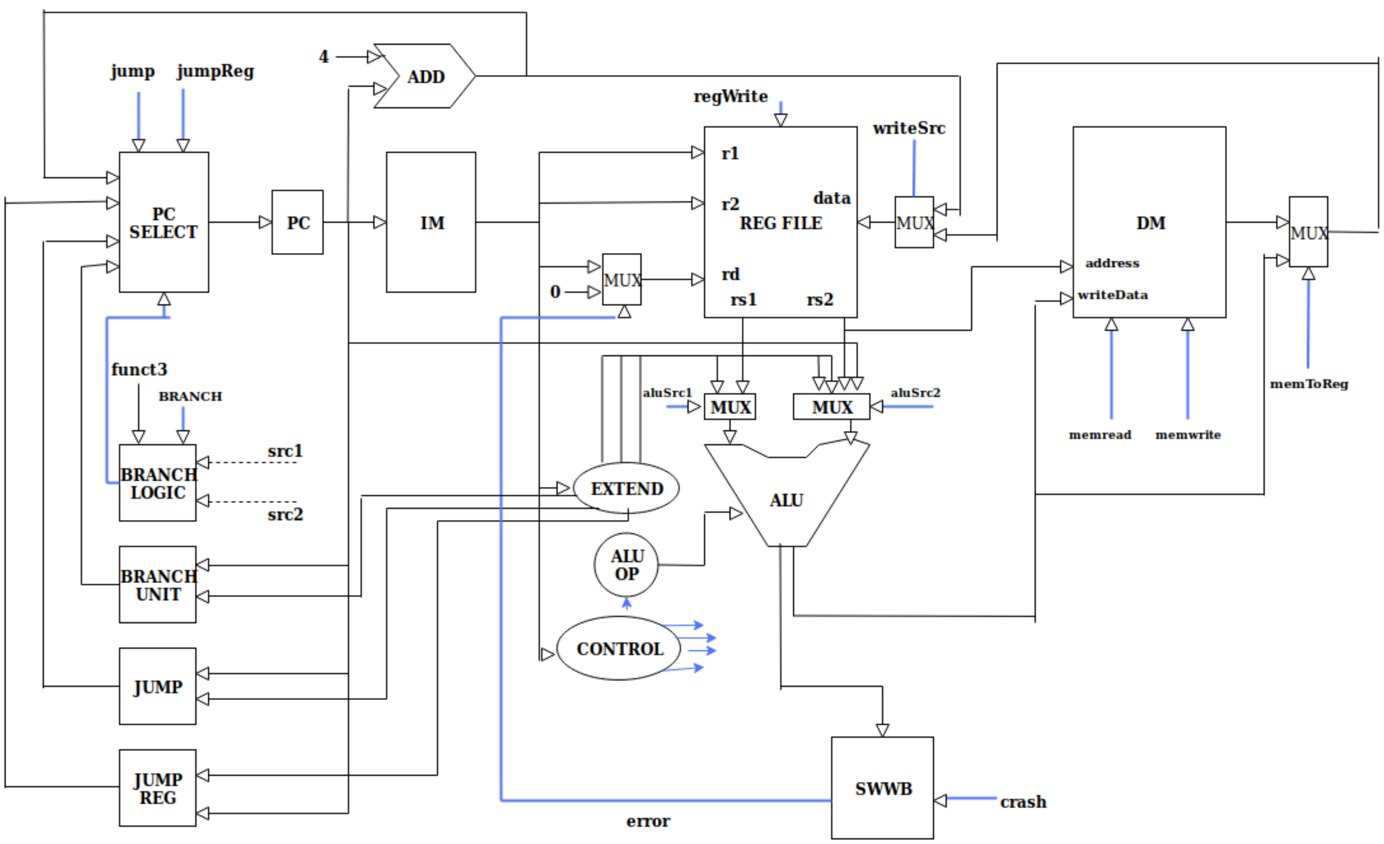


Figure 3: The Anhyzer core. Control paths are blue, while data paths are black. Inputs are marked with an arrow, while outputs are straight lines.

4.2 Overview

Following is an overview of how the components in the Anhyzer core (a RISC-V RV32I CPU) are defined and functions.

4.2.1 Instruction memory

The instruction memory (IM) stores the instructions as 32-bit binaries. The PC outputs the address of the instruction to be loaded from the IM which is then outputted. The instruction is then split up such that the bits corresponding to registers are sent to the register file, and the opcode to the control unit. The funct7 and funct3 bits are also sent to the control unit. Another set of bits are sent to the extension unit where they are sign extended.

4.2.2 Register file

The register file contains 32 register, each with a width of 32 bits. The registers stores integer values.

4.2.3 Data memory

The data memory is responsible for storing data. Data can then be written to or read from the memory. As RISC-V is little endian, the least significant byte is stored first.

4.2.4 ALU

The *arithmetic logic unit* carries out all the arithmetic and logical instruction such as adding two operands or executing a bitwise AND. A basic error detection was implemented such that the ALU can detect overflow and illegal instructions. In the first case this is done by creating a 33 bit value and checking if the most significant bit is 1. If that is the case, the error output is asserted. Any illegal instruction will also cause the error output to be asserted. The ALU has a switch-statement that checks for all supported instructions. If it falls through, then the error output is asserted.

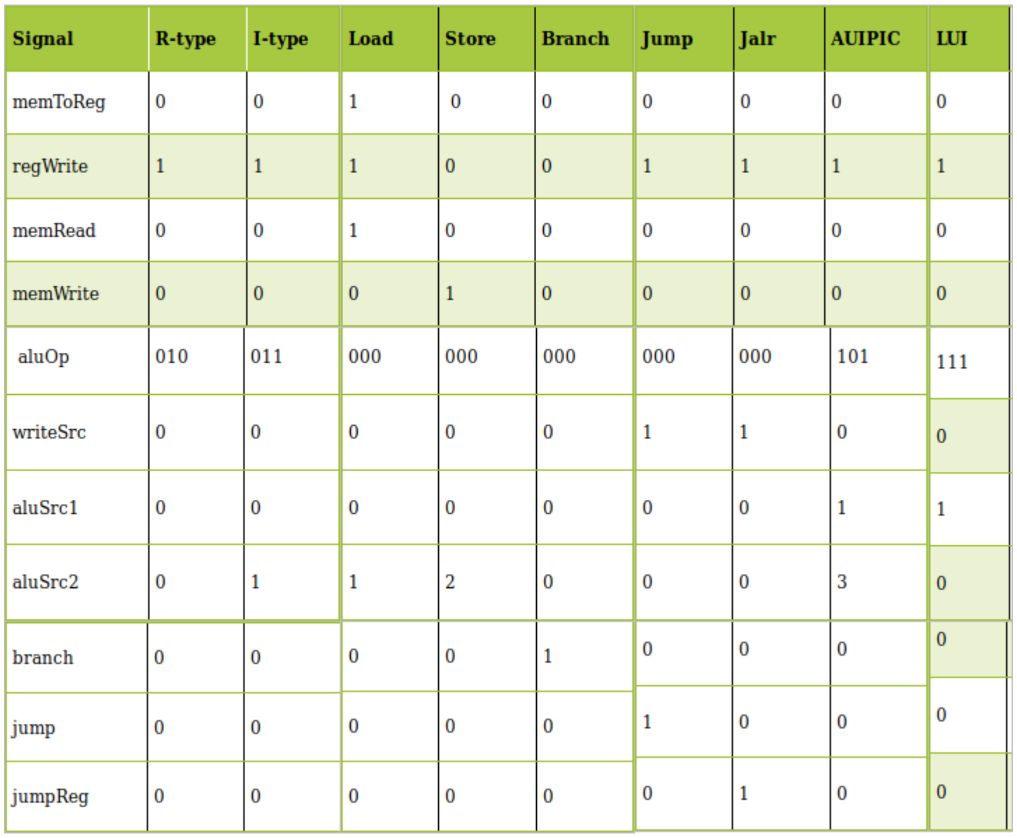
4.2.5 Control unit

The control unit is the main unit responsible for making sure the correct data is used, deciding which operation to perform etc. It takes the *opcode* as input and uses it to decide which signals to turn on. The opcode is the part of the instruction (the first 7 bits) that decides which operation should be performed. Basically, the control unit decides which input of the multiplexers should be outputted, as well as which operation the ALU should perform. In the case of the ALU, the control unit outputs a certain aluOp code which the ALU control unit then decodes together with the funct7 and funct3 fields. See Table 3 for an explanation of each signal. The truth table for the different instruction types, as well as the aluOps can be seen in Table 4.

aluSrc2 can take on the values 0, 1 and 2. If it is 0, then register rs2 is used as an operand. If it is 1, then the second operand is the I-type extended value. Finally if the value is 2, the B-type extended value is used.

Signals		
Signal name	Effect when asserted	Effect when not asserted
memToReg	data to register is from memory	data to register is from ALU
regWrite	write to register rd	do not write to register
memRead	read value at specified address	None
memWrite	write value at writeData port to memory	None
writeSrc	value to register writeData port is PC+4	value to register comes from ALU/memory
aluSrc1	use register rs1 as operand	use U-type extended value as operand
branch	take branch if branch instruction returns true	do not branch
jump	input to the PC is the address calculated by the jump unit	None
jumpReg	input to PC is address calculated by the jumpReg unit	None

Table 3: Signal types for the control unit.



Signal	R-type	I-type	Load	Store	Branch	Jump	Jalr	AUIPIC	LUI
memToReg	0	0	1	0	0	0	0	0	0
regWrite	1	1	1	0	0	1	1	1	1
memRead	0	0	1	0	0	0	0	0	0
memWrite	0	0	0	1	0	0	0	0	0
aluOp	010	011	000	000	000	000	000	101	111
writeSrc	0	0	0	0	0	1	1	0	0
aluSrc1	0	0	0	0	0	0	0	1	1
aluSrc2	0	1	1	2	0	0	0	3	0
branch	0	0	0	0	1	0	0	0	0
jump	0	0	0	0	0	1	0	0	0
jumpReg	0	0	0	0	0	0	1	0	0

Table 4: Truth table for the control unit

4.2.6 ALU control

The ALU control takes the funct7, funct3 and aluOp values and then uses these values to decode the correct operation for the ALU. It checks the aluOp first to see what type of instruction is in question (R-type, I-type etc.). The funct7 and funct3 fields are then used to decide which of the different operations the ALU should execute.

4.2.7 Sign extension unit

This unit handles the sign extension of instructions. For example, the add immediate instruction (addi) takes the 12 bit immediate and then extends it to a 32 bit value by padding it with the sign value. As such a 12 bit value such as `011111111111` would be extended to `00000000000000000000011111111111`. Figure 4 shows the different extended immediate values.

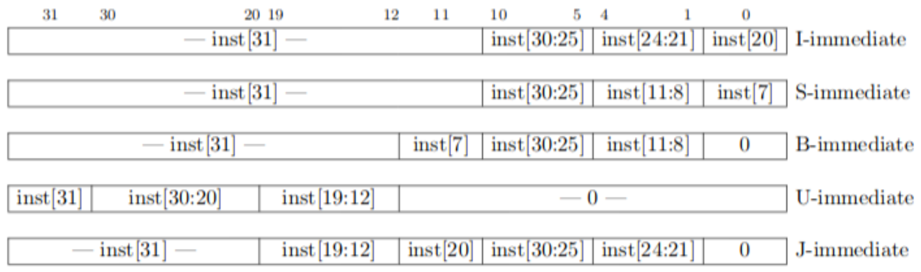


Figure 4: Types of immediates produced by the sign extension unit.[4]

4.2.8 Something went wrong box

The *Something went wrong box* is the Anhyzer core's error handler. It takes as input the error signal from the ALU and a crash signal. When the error input to it is asserted, it asserts the error output. This output controls the multiplexer which decides which register to write to. As such when the error output is asserted, the destination register is set to x0, effectively turning the instruction that caused an exception into a no operation (NOP), as writes to register x0 are ignored. If the crash signal is asserted (which is manually done), then instead of turning the exception into a NOP the CPU crashes the program.

4.2.9 PC-select

The PC-select outputs the value that the PC should store and output. It has 3 control inputs: the jump-, jalr- and takeBranch-signal. When the corresponding signal is asserted the next value of the PC will be the values calculated by the branch-, jump- or jumpReg unit. If none of the signals are asserted then the value will be the current PC value + 4.

4.2.10 Branch logic

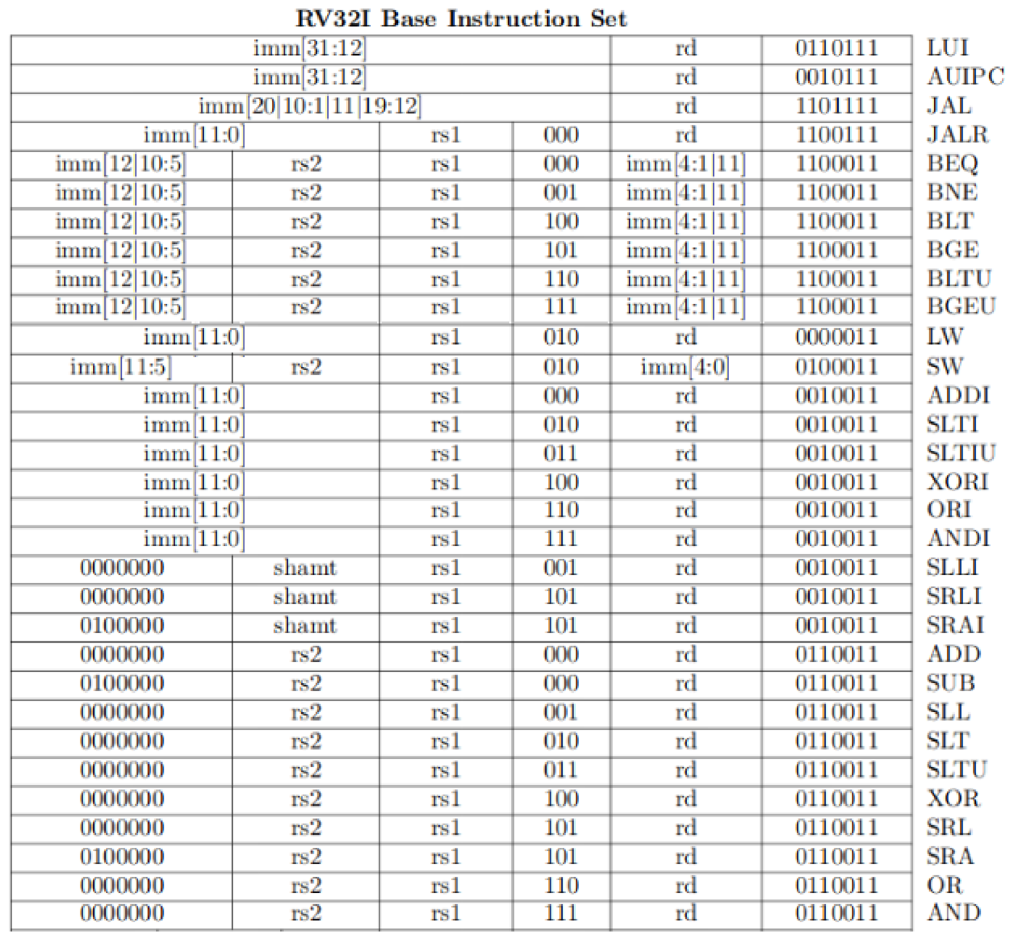
The branch logic is responsible for deciding whether a branch should be taken or not. It takes the funct3 field as input to decide which of the branch instructions it should execute (BEQ, BNE, etc.). The two registers are then used as operands and if the operation returns 1, then it checks to see that the branch signal is also asserted. If both conditions are true, the signal to take the branch is asserted and the PC-select will output the value calculated by the branch unit.

4.2.11 Jump and Branch units

These units calculate the value to which to jump/branch to. In the case of the branch and jump unit, this is done by adding the current PC value to the extended B-immediate for branches and the extended J-immediate for jumps. The jumpReg is different as it takes register rs1 and the I-immediate, adds them together and sets the least significant bit to 0. The outputs from these units are then sent to the PC-select unit.

4.2.12 Instructions

The following instructions (figure 5) are supported by the Anhyzer core.



RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

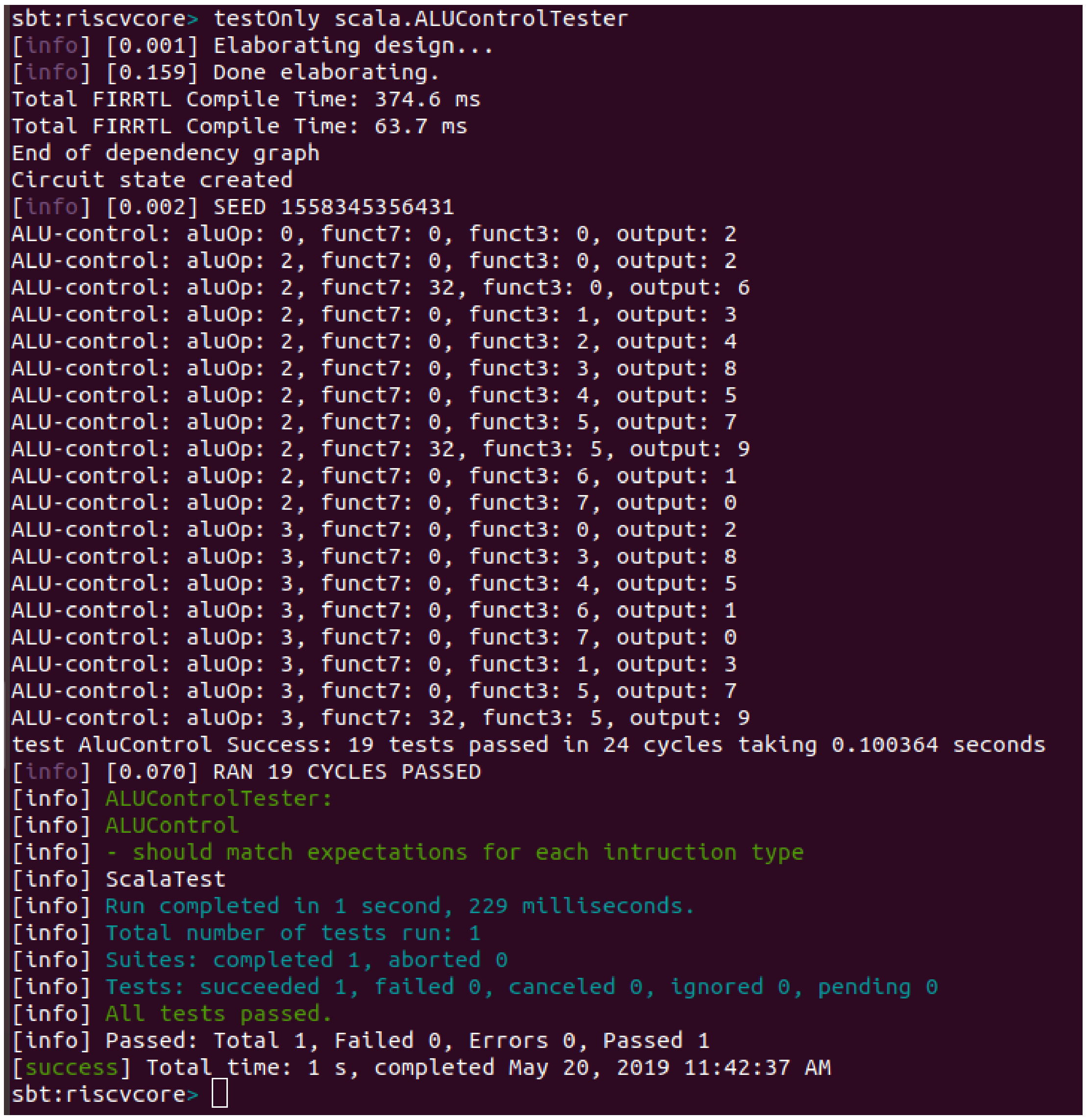
Figure 5: The instructions implemented in the Anhyzer core.[4]

Brief explanations of the instructions:

- `ADD/ADDI:` performs addition using registers rs1, rs2 or rs1 and the sign extended 12 bit immediate.
- `SUB:` subtracts rs2 from rs1.
- `OR/ORI:` bitwise OR instruction.
- `AND/ANDI:` bitwise AND instruction.
- `XOR/XORI:` bitwise XOR instruction.
- `LW/SW:` Loads/Stores a word (32 bit/8 byte) in the data memory.
- `SLT(U)/SLTI(U):` puts the value 1 in register rd if rs1 is less than rs2/the sign extended immediate. SLT/SLTI are signed while SLTU/SLTIU are unsigned operations.
- `SLL(I)/SLR(I)/SRA(I):` left-/right logical shift and right arithmetic shift of register rs1 with the lower 5 bits of rs2/shift amount field (shamt).
- `BEQ, BNE, BLT, BGE:` branch if rs1 and rs2 are equal, not equal, rs1 is less than rs2, and rs1 is greater than or equal to rs2. These operations are signed with BLTU and BGEU being unsigned.
- `JALR:` jump-and-link register (indirect jump). Adds the extended I-immediate to register rs1, and writes the instruction following it (PC+4) to register rd.
- `JAL:` jump-and-link (unconditional jump). Adds the extended J-immediate to the PC and then writes PC+4 to register rd.
- `AUIPC:` add upper immediate to PC. Used to build PC-relative addresses. Writes the result to register rd.
- `LUI:` load upper immediate. Used to build 32-bit constants by taking the upper 20 bits and then filling the lowest 12 bits with zeroes. Writes the result to register rd.

4.3 Simulation

The Anhyzer core was simulated in order to make sure that it functions correctly. Chisel provides many ways to test out a design. A common method is to compile the Chisel code into Verilog and then writing test harnesses in for example C++. Chisel also provides self-contained testers which is what was used to simulate the Anhyzer core. In order to make sure that each component was working correctly, the *PeekPokeTester* was used. It allows for unit-testing of the separate components by ”poking”, in which a value to the input is set. One can then ”peek” to assert that the correct value is output.



```
sbt:riscvcore> testOnly scala.ALUControlTester
[info] [0.001] Elaborating design...
[info] [0.159] Done elaborating.
Total FIRRTL Compile Time: 374.6 ms
Total FIRRTL Compile Time: 63.7 ms
End of dependency graph
Circuit state created
[info] [0.002] SEED 1558345356431
ALU-control: aluOp: 0, funct7: 0, funct3: 0, output: 2
ALU-control: aluOp: 2, funct7: 0, funct3: 0, output: 2
ALU-control: aluOp: 2, funct7: 32, funct3: 0, output: 6
ALU-control: aluOp: 2, funct7: 0, funct3: 1, output: 3
ALU-control: aluOp: 2, funct7: 0, funct3: 2, output: 4
ALU-control: aluOp: 2, funct7: 0, funct3: 3, output: 8
ALU-control: aluOp: 2, funct7: 0, funct3: 4, output: 5
ALU-control: aluOp: 2, funct7: 0, funct3: 5, output: 7
ALU-control: aluOp: 2, funct7: 32, funct3: 5, output: 9
ALU-control: aluOp: 2, funct7: 0, funct3: 6, output: 1
ALU-control: aluOp: 2, funct7: 0, funct3: 7, output: 0
ALU-control: aluOp: 3, funct7: 0, funct3: 0, output: 2
ALU-control: aluOp: 3, funct7: 0, funct3: 3, output: 8
ALU-control: aluOp: 3, funct7: 0, funct3: 4, output: 5
ALU-control: aluOp: 3, funct7: 0, funct3: 6, output: 1
ALU-control: aluOp: 3, funct7: 0, funct3: 7, output: 0
ALU-control: aluOp: 3, funct7: 0, funct3: 1, output: 3
ALU-control: aluOp: 3, funct7: 0, funct3: 5, output: 7
ALU-control: aluOp: 3, funct7: 32, funct3: 5, output: 9
test AluControl Success: 19 tests passed in 24 cycles taking 0.100364 seconds
[info] [0.070] RAN 19 CYCLES PASSED
[info] ALUControlTester:
[info] ALUControl
[info] - should match expectations for each intruction type
[info] ScalaTest
[info] Run completed in 1 second, 229 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 1, Failed 0, Errors 0, Passed 1
[success] Total time: 1 s, completed May 20, 2019 11:42:37 AM
sbt:riscvcore>
```

Figure 6: Unit test for the aluOP unit. Values are ”poked” to the input of the component. In this case the ”poked” values are the funct3 and funct7 fields as well as the aluOp. The output is then ”peeked” in order to assert the correct value, in this case the output.

This was done for every component in order to make sure that they worked as intended. See Figure 6 for an example of the result of the test of the aluOp unit.

The *SteppedHWIOTester* was used to instantiate and test the design which can then be stepped through using the `step()` function. This causes the simulation to advance one clock cycle. The underlying simulator is the open-source HDL simulator Verilator. Instructions were loaded into the instruction memory and the simulation was then run. To make sure that everything worked as intended, the output of the simulation was observed to make sure values were correct. Following is an example of a simulation of the Anhyzer core (figure 7). To get an overview, the instructions loaded into the instruction memory are:

`addi x2, 3, x7` (add the immediate value 3 to register x2 and then store the value in x2 it in destination register x7).

`sw x7, 0(x0)` (store the word in register x7 at memory address of the value in x0 with the offset 0 (i.e at address 0 of the data memory)).

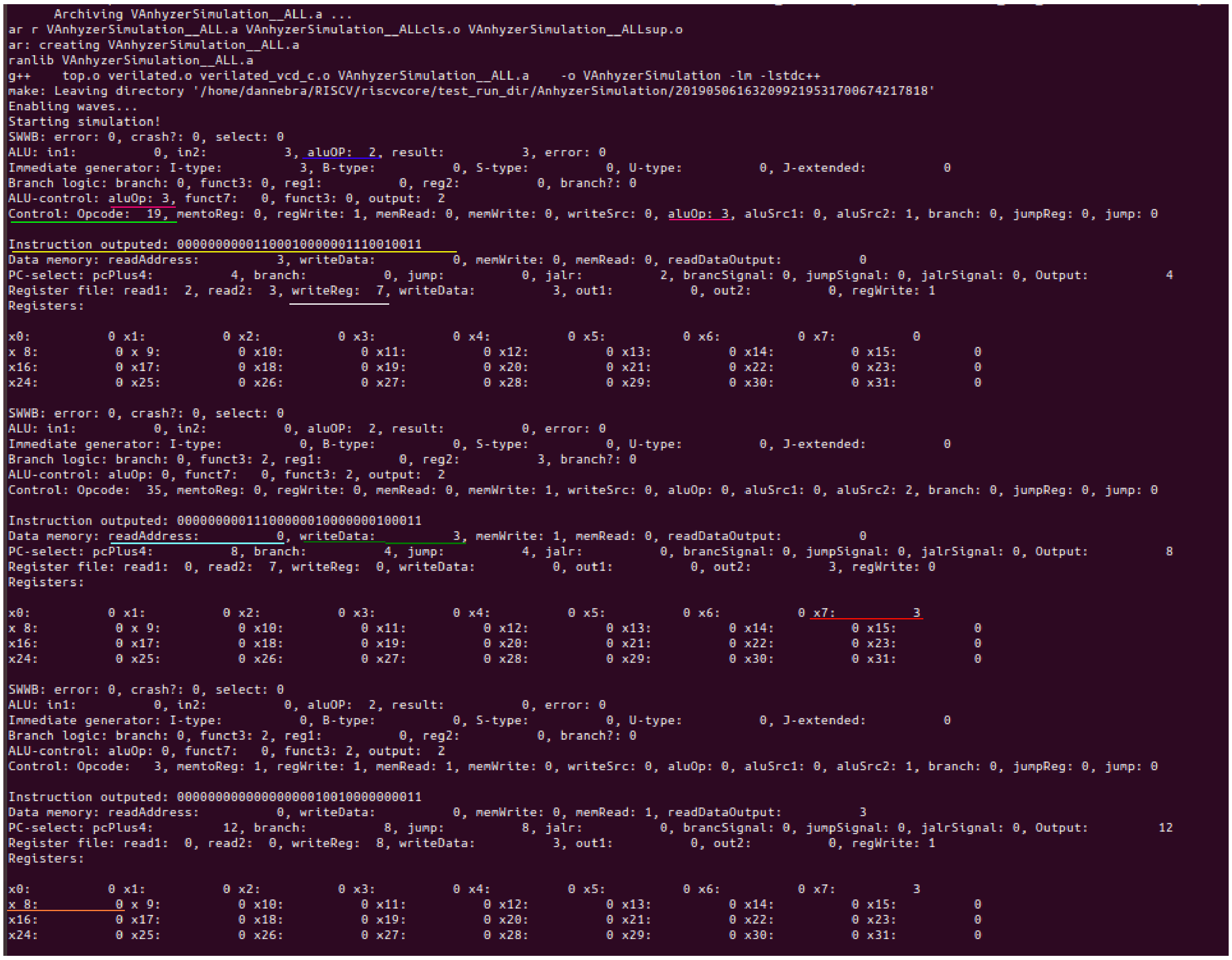
`lw x8, 0(x0)` (load the word at memory address 0 and store it in register x8).

In this example, the Chisel code is first compiled into Verilog code which is the used by Verilator. Following are the steps of the computation:

1. The add immediate instruction is output from the instruction memory (yellow line). The control unit receives the opcode from the instruction and sets the correct outputs. In this case it detects an I-type instruction (opcode 0010011 (19)) marked with the light green line. As such it sets the regWrite signal to 1 since it writes to register 7 (white). aluSrc2 is set to 1 since it uses the extended I-type immediate. The ALU-control receives the funct7 and funct3 fields, as well as the aluOp (011/3) from the main control unit (pink). The ALU-control then outputs the value 2 to the ALU (blue), signaling that this is an add instruction. The add instruction is carried out and the ALU outputs the value 3. Since no error has been detected, the error output is not asserted leading to all of the SWWB's values being 0. Since all registers are empty from the start, the values in each register is 0. As no jumps or branches are made, the pcSelect outputs 4 to output the instruction immediately following the previous one.

2. The store instruction is output and the address to where to store the value is calculated (teal). For both load and store instructions the ALU will execute an add instruction to calculate the address. It is done by taking the value in the register (x0 in this case) and adding the offset to it. Since both values are 0 in the example, the ALU outputs 0. The control unit now receives the opcode 0100011 (35) which means that it is a store instruction. Each of the signals corresponding to the store instruction is set accordingly. The value 3 can now be seen in register x7 (red). The value in x7 is stored in the data memory at address 0 (readAddress). As such the value 3 is set on the writeData input (dark green). Note also that pcSelect is 8 now, since no branches was taken this time either.

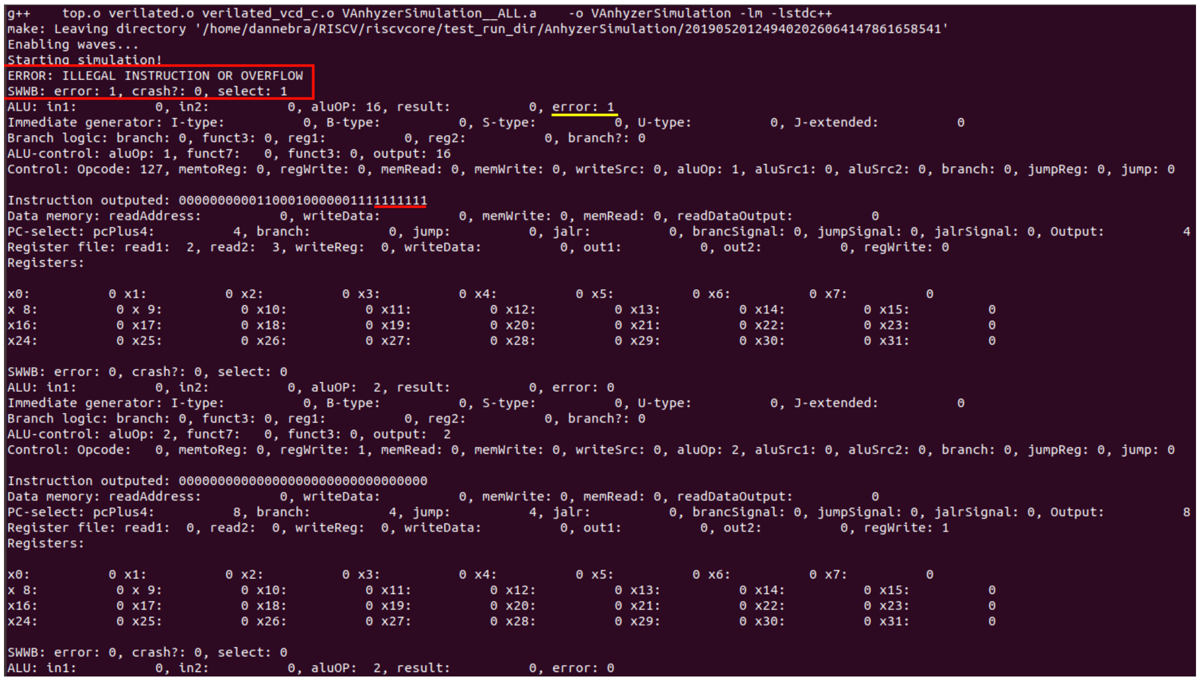
3. The final instruction is the load instruction which has the opcode 0000011 (3). memRead is set to 1, and the address to where to load the value from is calculated. Note that the value in x8 is still 0 (orange line). This is because the value is updated at the end of the cycle, meaning that it will show up in the next cycle (similar to how the value 3 in x7, marked with the red line, is not shown in the first stage). After this instruction, the program terminates.



```
      Archiving VAnhyzerSimulation__ALL.a ...
ar r VAnhyzerSimulation__ALL.a VAnhyzerSimulation__ALLcls.o VAnhyzerSimulation__ALLsup.o
ar: creating VAnhyzerSimulation__ALL.a
ranlib VAnhyzerSimulation__ALL.a
g++    top.o verilated.o verilated_vcd_c.o VAnhyzerSimulation__ALL.a    -o VAnhyzerSimulation -lm -lstdc++
make: Leaving directory '/home/dannebra/RISCV/riscvcore/test_run_dir/AnhyzerSimulation/2019050616320992195317006742178l8'
Enabling waves...
Starting simulation!
SWWB: error: 0, crash?: 0, select: 0
ALU: in1:          0, in2:          3, aluOP:  2, result:          3, error: 0
Immediate generator: I-type:          3, B-type:          0, S-type:          0, U-type:          0, J-extended:          0
Branch logic: branch: 0, funct3: 0, reg1:          0, reg2:          0, branch?: 0
ALU-control: aluOp: 3, funct7:   0, funct3: 0, output:  2
Control: Opcode:  19, memtoReg: 0, regWrite: 1, memRead: 0, memWrite: 0, writeSrc: 0, aluOp: 3, aluSrc1: 0, aluSrc2: 1, branch: 0, jumpReg: 0, jump: 0

Instruction outputed: 00000000001100010000001110010011
Data memory: readAddress:          3, writeData:          0, memWrite: 0, memRead: 0, readDataOutput:          0
PC-select: pcPlus4:          4, branch:          0, jump:          0, jalr:          2, brancSignal: 0, jumpSignal: 0, jalrSignal: 0, Output:          4
Register file: read1:  2, read2:  3, writeReg:  7, writeData:          3, out1:          0, out2:          0, regWrite: 1
Registers:

x0:          0 x1:          0 x2:          0 x3:          0 x4:          0 x5:          0 x6:          0 x7:          0
x 8:          0 x 9:          0 x10:          0 x11:          0 x12:          0 x13:          0 x14:          0 x15:          0
x16:          0 x17:          0 x18:          0 x19:          0 x20:          0 x21:          0 x22:          0 x23:          0
x24:          0 x25:          0 x26:          0 x27:          0 x28:          0 x29:          0 x30:          0 x31:          0

SWWB: error: 0, crash?: 0, select: 0
ALU: in1:          0, in2:          0, aluOP:  2, result:          0, error: 0
Immediate generator: I-type:          0, B-type:          0, S-type:          0, U-type:          0, J-extended:          0
Branch logic: branch: 0, funct3: 2, reg1:          0, reg2:          3, branch?: 0
ALU-control: aluOp: 0, funct7:   0, funct3: 2, output:  2
Control: Opcode:  35, memtoReg: 0, regWrite: 0, memRead: 0, memWrite: 1, writeSrc: 0, aluOp: 0, aluSrc1: 0, aluSrc2: 2, branch: 0, jumpReg: 0, jump: 0

Instruction outputed: 00000000011100000010000000100011
Data memory: readAddress:          0, writeData:          3, memWrite: 1, memRead: 0, readDataOutput:          0
PC-select: pcPlus4:          8, branch:          4, jump:          4, jalr:          0, brancSignal: 0, jumpSignal: 0, jalrSignal: 0, Output:          8
Register file: read1:  0, read2:  7, writeReg:  0, writeData:          0, out1:          0, out2:          3, regWrite: 0
Registers:

x0:          0 x1:          0 x2:          0 x3:          0 x4:          0 x5:          0 x6:          0 x7:          3
x 8:          0 x 9:          0 x10:          0 x11:          0 x12:          0 x13:          0 x14:          0 x15:          0
x16:          0 x17:          0 x18:          0 x19:          0 x20:          0 x21:          0 x22:          0 x23:          0
x24:          0 x25:          0 x26:          0 x27:          0 x28:          0 x29:          0 x30:          0 x31:          0

SWWB: error: 0, crash?: 0, select: 0
ALU: in1:          0, in2:          0, aluOP:  2, result:          0, error: 0
Immediate generator: I-type:          0, B-type:          0, S-type:          0, U-type:          0, J-extended:          0
Branch logic: branch: 0, funct3: 2, reg1:          0, reg2:          0, branch?: 0
ALU-control: aluOp: 0, funct7:   0, funct3: 2, output:  2
Control: Opcode:   3, memtoReg: 1, regWrite: 1, memRead: 1, memWrite: 0, writeSrc: 0, aluOp: 0, aluSrc1: 0, aluSrc2: 1, branch: 0, jumpReg: 0, jump: 0

Instruction outputed: 00000000000000000010010000000011
Data memory: readAddress:          0, writeData:          0, memWrite: 0, memRead: 1, readDataOutput:          3
PC-select: pcPlus4:         12, branch:          8, jump:          8, jalr:          0, brancSignal: 0, jumpSignal: 0, jalrSignal: 0, Output:         12
Register file: read1:  0, read2:  0, writeReg:  8, writeData:          3, out1:          0, out2:          0, regWrite: 1
Registers:

x0:          0 x1:          0 x2:          0 x3:          0 x4:          0 x5:          0 x6:          0 x7:          3
x 8:          0 x 9:          0 x10:          0 x11:          0 x12:          0 x13:          0 x14:          0 x15:          0
x16:          0 x17:          0 x18:          0 x19:          0 x20:          0 x21:          0 x22:          0 x23:          0
x24:          0 x25:          0 x26:          0 x27:          0 x28:          0 x29:          0 x30:          0 x31:          0
```

Figure 7: Example simulation of the Anhyzer core. Print statements have been added to see the different states of each component.

For the error handling, a couple of simple experiments were made where illegal instructions were loaded into memory. Any instruction with an opcode that is not recognized will trigger the ALU's error output (see figure 8).



```
g++    top.o verilated.o verilated_vcd_c.o VAnhyzerSimulation__ALL.a    -o VAnhyzerSimulation -lm -lstdc++
make: Leaving directory '/home/dannebra/RISCV/riscvcore/test_run_dir/AnhyzerSimulation/2019052012494020260641478616585 41'
Enabling waves...
Starting simulation!
ERROR: ILLEGAL INSTRUCTION OR OVERFLOW
SWWB: error: 1, crash?: 0, select: 1
ALU: in1:          0, in2:          0, aluOP: 16, result:          0, error: 1
Immediate generator: I-type:          0, B-type:          0, S-type:          0, U-type:          0, J-extended:          0
Branch logic: branch: 0, funct3: 0, reg1:          0, reg2:          0, branch?: 0
ALU-control: aluOp: 1, funct7:   0, funct3: 0, output: 16
Control: Opcode: 127, memtoReg: 0, regWrite: 0, memRead: 0, memWrite: 0, writeSrc: 0, aluOp: 1, aluSrc1: 0, aluSrc2: 0, branch: 0, jumpReg: 0, jump: 0

Instruction outputed: 00000000011000100000001111111111
Data memory: readAddress:          0, writeData:          0, memWrite: 0, memRead: 0, readDataOutput:          0
PC-select: pcPlus4:          4, branch:          0, jump:          0, jalr:          0, brancSignal: 0, jumpSignal: 0, jalrSignal: 0, Output:          4
Register file: read1:  2, read2:  3, writeReg:  0, writeData:          0, out1:          0, out2:          0, regWrite: 0
Registers:

x0:          0 x1:          0 x2:          0 x3:          0 x4:          0 x5:          0 x6:          0 x7:          0
x 8:          0 x 9:          0 x10:          0 x11:          0 x12:          0 x13:          0 x14:          0 x15:          0
x16:          0 x17:          0 x18:          0 x19:          0 x20:          0 x21:          0 x22:          0 x23:          0
x24:          0 x25:          0 x26:          0 x27:          0 x28:          0 x29:          0 x30:          0 x31:          0

SWWB: error: 0, crash?: 0, select: 0
ALU: in1:          0, in2:          0, aluOP:  2, result:          0, error: 0
Immediate generator: I-type:          0, B-type:          0, S-type:          0, U-type:          0, J-extended:          0
Branch logic: branch: 0, funct3: 0, reg1:          0, reg2:          0, branch?: 0
ALU-control: aluOp: 2, funct7:   0, funct3: 0, output:  2
Control: Opcode:   0, memtoReg: 0, regWrite: 1, memRead: 0, memWrite: 0, writeSrc: 0, aluOp: 2, aluSrc1: 0, aluSrc2: 0, branch: 0, jumpReg: 0, jump: 0

Instruction outputed: 00000000000000000000000000000000
Data memory: readAddress:          0, writeData:          0, memWrite: 0, memRead: 0, readDataOutput:          0
PC-select: pcPlus4:          8, branch:          4, jump:          4, jalr:          0, brancSignal: 0, jumpSignal: 0, jalrSignal: 0, Output:          8
Register file: read1:  0, read2:  0, writeReg:  0, writeData:          0, out1:          0, out2:          0, regWrite: 1
Registers:

x0:          0 x1:          0 x2:          0 x3:          0 x4:          0 x5:          0 x6:          0 x7:          0
x 8:          0 x 9:          0 x10:          0 x11:          0 x12:          0 x13:          0 x14:          0 x15:          0
x16:          0 x17:          0 x18:          0 x19:          0 x20:          0 x21:          0 x22:          0 x23:          0
x24:          0 x25:          0 x26:          0 x27:          0 x28:          0 x29:          0 x30:          0 x31:          0

SWWB: error: 0, crash?: 0, select: 0
ALU: in1:          0, in2:          0, aluOP:  2, result:          0, error: 0
```

Figure 8: Example of an illegal instruction triggering the SWWB. The opcode 1111111 (underlined with red) is not valid. As such the ALU's error output is asserted (yellow line) which causes the SWWB to turn the instruction into a NOP (red box). Following the illegal instruction is a NOP, showing that nothing has changed.

5 Results

The Anhyzer core is able to simulate the RV32I instruction set as well as provide some error handling. This was validated from the experiments run in the previous section (4.3). Bugs might still exist, but from the observations the core outputs the correct values. This shows that the design is functional as a proof of concept and works as a foundation if one were to create a more complex RISC-V CPU with more functionality or instructions.

A benchmark was performed on the Anhyzer core, where instructions were loaded into memory and then simulated. Two different types of instructions were loaded into memory, add instructions (addi x2 0 x7) and load instructions (lw x8 0(x0)) in order to see if there was any difference in performance between the two types. The benchmark takes into account the overhead of creating the circuit. The result of the benchmark can be seen in Figure 9.

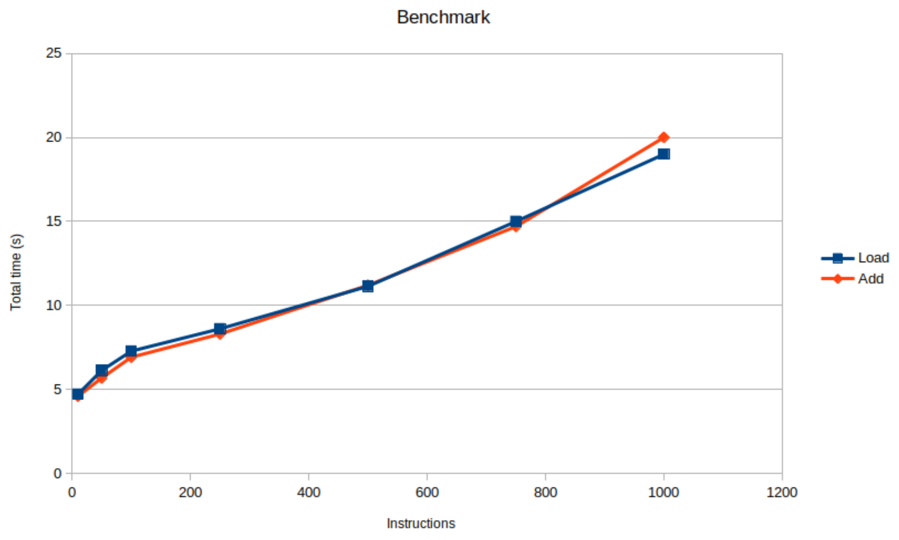


Figure 9: Benchmark for the Anhyzer core. The number of instructions run is on the x-axis, while the time in seconds is on the y-axis. The experiment was carried out using Linux Ubuntu 18.04 64 bit on an Intel i5-4210M of 3.20 GHz, with a processor of 4 cores, with a 8 GB RAM and an 3 MB L3 cache.

6 Related work

Many RISC-V designs have been created with the most noticeable ones being the SODOR and Berkeley out of order Machine (BOOM). These cores are created by University of California, Berkely. The SODOR cores are used as educational cores and provide a 2,3 and 5 stage pipelined core. The BOOM core is more sophisticated, being a 64-bit parameterized and synthesizable core. Both cores are written in Chisel and can be downloaded from the university's Github page. Compared to the Anhyzer core, the SODOR and BOOM cores have a better performance since they are pipelined.

7 Discussion

RISC-V is very interesting in many ways. It now opens up open-source processor design, where previously hardware designers would have to pay for a license for proprietary ISAs. Even then, designers were not able to freely create designs. To test out the RISC-V ISA, the Anhyzer core was implemented in Chisel and simulated to make sure it works. It is currently able to run the RV32I instructions. The CPU is somewhat primitive, and there are some problems with it. The experimental function to instantiate memory was not really working. At times it kept loading nothing but zeroes into the memory, so instructions were instead hard coded into the instruction memory. Whether this is because of Chisel or because of something else is uncertain. The Anhyzer core's error handling can also be improved. It is only able to detect arithmetic overflow and illegal instructions, which might not be enough. Currently, the designer has to decide whether or not the CPU should crash the program when an exception occurs. The default is to transform an instruction that causes an exception into a NOP which is problematic as the user does not know that an error has occurred. Instead this might lead to undefined behavior, as the program might expect certain values. Thus it might be better to always have the error signal asserted which crashes the program if a problem occurs.

When it comes to performance, the Anhyzer core is not the most efficient core. Since it is a single-cycle CPU, the throughput will be low. From the benchmark, one can see that it takes a while to instantiate the core and the time it takes to execute the instructions is also somewhat long. This is most likely because of overhead from the simulation, so it might be faster if it was run on a better computer, or when synthesised. There was also no major differences in run time for the load and add instructions, meaning that the simulation seems to adapt to the slowest instruction type. As the goal was not to create the most efficient core, long run times were expected.

One problem with the simulation was also that it timed out and then crashed if too many instructions were loaded into memory. The simulation would throw a "java.lang.OutOfMemoryError: GC overhead limit exceeded" exception if one tried to simulate for example 10 000 instructions. It would have been interesting to see how the core handles millions of instructions, but this was limited by the provided tool.

7.1 Problems encountered with Chisel

Most of the problems that occurred during the implementation of the CPU was because of problems with Chisel. Chisel is a very new language, and as such it is also very volatile. Many things are susceptible to changes as the language keeps being developed. This was very noticeable as much of the information on the internet regarding Chisel that is not older than a few years are outdated. This made it a bit difficult, as a beginner to Chisel and hardware description languages, to learn it. Many of the features that other HDLs have are experimental or non-existent in Chisel (such as instantiating memory). Even the BOOM CPU, which is written by the researchers who develop Chisel have many functions that are deprecated. Exceptions such as "Internal Error! Please file an issue at https://github.com/ucb-bar/firrtl/issues" were common, giving an impression that not everything is in place. If one were to use Chisel professionally then it might lead to problems as the language is changing so quickly. As such I believe that Chisel is not a hardware description language that is really ready to hit the market just yet. Even when developing something that is not meant to be used commercially, Chisel still feels like an incomplete language.

The creators claim that Chisel increases developer productivity, but in my opinion it will only do so when the language is more mature and stable. Having to relearn the syntax or having experimental functions removed is not something that makes a developer more productive. Instead it creates other problems that needs to be fixed. With languages such as Verilog and VHDL being much older and stable and with much more information available, switching to Chisel might take some convincing. Still, I believe that in the future Chisel might become a popular HDL since it provides many features that makes it easier to develop hardware, that other languages do not have.

8 Summary

8.1 Conclusion

This thesis have explored RISC-V and the new hardware description language Chisel which was inspired by software development. A 32 bit RISC-V CPU nicknamed the Anhyzer core, implementing the RV32I instruction set, was designed and simulated. RISC-V now provides many interesting new possibilities when designing open cores, as it is open-source and completely free to use. Chisel also provides many features inspired from software, but still lacks some of the stability needed for professional use. However, with the interest in RISC-V from companies such as Google, it seems like it might be the future of processor design.

8.2 Future work

As RISC-V and Chisel keeps changing rapidly, it would be interesting to see what features are implemented, and how RISC-V is able to compete with the more traditional ISAs. The Anhyzer core can also be expanded upon, implementing more instructions. The error handling can also be made more sophisticated. To increase throughput, the core can also be pipelined. However, many of these things require a greater knowledge in hardware design.

Another topic that would be interesting to look into in more detail would be if changing the parameters of a core written in Chisel affects the performance. The BOOM core is parameterized, and can thus be modified in order to see what happens. In order to do this, the fetch/decode width can be changed from 1 to 2 and 3. The number of re-order buffer (ROB) entries can be changed from 64 to 96, 128 and possibly 256. Other things to look at would be to change the number of (floating point) physical registers to 96, 128 or 180. Finally, one could look at how enabling pre-fetching would affect the core's performance.

References

[1] *Waterman A.*, Design of the RISC-V Instruction Set Architecture, Technical Report No. UCB/EECS-2016-1, January 3, 2016.
Available at: `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.pdf`

[2] *Asanovic K., A. Patterson D.*, Instruction Sets Should Be Free: The Case For RISC-V, Technical Report No. UCB/EECS-2014-146, August 6, 2014.
Available at: `https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf`

[3] *Waterman A., Yunsup L., A. Patterson D., Asanovic K.*, The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0, Technical Report No. UCB/EECS-2014-54, May 6, 2014.
Available at: `https://apps.dtic.mil/dtic/tr/fulltext/u2/a605735.pdf`

[4] *Page D.*, A Practical Introduction to Computer Architecture, p. 199, Springer-Verlag London Limited, ISBN: 978-1-84882-255-9, 2009.

[5] *Waterman A., Asanovic K.*, The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2, May 7, 2017.
Available at: `https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`

[6] *Kanter D.*, RISC-V Offers Simple, Modular IS, Microprocessor report, March 28, 2016.
Available at:
`https://www.cslab.pepperdine.edu/warford/cosc425/RISC-V-Offers-Simple-Modular-ISA.pdf`

[7] *Ceze L., Hill M. D., Sankaralingam K., Wenisch T. F.* (2017) Democratizing Design for Future Computing Platforms
Available at:
`https://cra.org/ccc/resources/ccc-led-whitepapers/`

[8] *Marena, T.*, RISC-V: Opening a New Era of Innovation for Embedded Design, June 27 2018. Available at:
`https://www.allaboutcircuits.com/industry-articles/risc-v-opening-a-new-era-of-innovation-for-embedded-design/`
Accessed: 2019-05-22.

[9] Official RISC-V web page, Why RISC-V?, 2019.
Available at: `https://riscv.org/why-risc-v/`
Accessed: 2019-05-22.

[10] *Silberschatz A., B. Galvin P., Gagne G.*, Operating System Concepts, 8th Edition. John Wiley Sons, Inc. 2009. p. 22, ISBN: 9780470128725.

[11] *Waterman A., Asanovic K.*, The RISC-V Instruction Set Manual Volume II: Privileged Architecture, May 7, 2017.
Available at: `https://content.riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf`

[12] *A. Patterson D.*, Computer Organization and Design RISC-V Edition - The Hardware Software Interface: RISC-V Edition, Morgan Kaufmann (Elsevier Inc.), 2018, ISBN: 9780128122754.

[13] *Bachrach J., Vo H., Richards B., Lee Y., Waterman A., Avižienis R, Wawrzynek J., Asanovic K.*, Chisel: Constructing Hardware in a Scala Embedded Language, ACM, June 2012.
Available at: `https://chisel.eecs.berkeley.edu/chisel-dac2012.pdf`

[14] Free chips project, Chisel introduction/tutorial, May 2, 2017.
Available at: `https://github.com/freechipsproject/chisel3/wiki/Chisel-Introduction`
Accessed: 2019-05-22.

[15] *D. Hill M., Christie D., A. Patterson D., Yi J. J., Chiou D., Sendag R.*, Proprietary versus Open Instruction Sets, IEEE Computer Society, July/August 2016.
Available at: `http://research.cs.wisc.edu/multifacet/papers/ieeemicro16_card_isa.pdf`

[16] *Ferdjallah M.*, Introduction to Digital Systems : Modeling, Synthesis, and Simulation Using VHDL, Wiley, 2011, p. 47, ISBN: 9780470900550.

[17] Official RISC-V web page, Frequently Asked Questions about RISC-V, 2019.
Available at: `https://riscv.org/faq/`
Accessed: 2019-05-22.

[18] The MIPS Open initiative, MIPS Open, 2019.
Available at: `https://www.mips.com/mipsopen/`
Accessed: 2019-05-22.

[19] *Heilmann F., Brugger C., Wehn N.*, White Paper - Investigate the high-level HDL Chisel, Microelectronics Research Group - University Kaiserslautern Kaiserslautern, Germany, 18/10 2013.
Available at: `https://kluedo.ub.uni-kl.de/frontdoor/deliver/index/docId/3638/file/_Chisel+Report.pdf`

[20] Open source, What is open hardware? 2019.
Available at: `https://opensource.com/resources/what-open-hardware`
Accessed: 2019-05-22.

[21] Wikipedia, Open-source hardware
Available at: `https://en.wikipedia.org/wiki/Open-source_hardware`
Accessed: 2019-05-22.