FH University of Applied Sciences
TECHNIKUM WIEN

MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Embedded Systems

Educational Processor Simulator

A graphical and educational processor simulator based on the RISC-V instruction set architecture

By: Michael Gafert

Student Number: 1910297003

Supervisors: FH-Prof. DI Dr. Martin Horauer
Dipl.-Ing. Christoph Veigl

Vienna, May 24, 2021

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Vienna, May 24, 2021 Signature

Kurzfassung

Die Landschaft aktueller Visualisierungstools für Prozessoren bietet eine große Vielfalt an Programmen in Bezug auf Informationsdetail, Visualisierungstechnik und Simulationsmöglichkeiten. Allerdings fehlt diesen Tools oft eine geführte Simulation, die Studierende helfen würde, den diskreten Informationsfluss einer CPU und den inhärenten Architekturentwurf zu verstehen. Studierende, die CPU-Architektur oder digitale Logik lernen, müssen oft auf einfache Videos mit Animationen oder statische Bilder zurückgreifen, die eine CPU auf die einfachste Weise beschreiben. Obwohl die Landschaft anderer Simulatoren mit Fokus auf den Einsatz in der Ausbildung groß ist, gibt es eine deutliche Lücke in Bezug auf geführte informationelle Simulationen. Diese Arbeit beschreibt die aktuelle Landschaft von CPU-Simulationswerkzeugen, zeigt den Bedarf für einen weiteren Simulator im Bildungsbereich auf und stellt einen entwickelten Simulator vor, der sich auf den pädagogischen Nutzen konzentriert, den er bieten kann. Der entwickelte Simulator, genannt *Apate*, beinhaltet Merkmale anderer Simulatoren und erweitert den Funktionsumfang um eine geführte Tour durch die CPU mit zusätzlichen Informationen zu jedem Schritt. Die Informationen sind direkt in den Simulator eingebettet. Apate führt den Benutzer durch die Architektur einer RISC-V basierten CPU und erweitert den Effekt um eine performante Visualisierung. Sowohl die Entwicklung, als auch die Benutzeroberfläche werden in dieser Arbeit mit einem abschließenden Usability-Test durch Lehrende und Studierende beschrieben.

Schlagworte: CPU, Simulator, RISC-V, Electron, pädagogischer Nutzen

Abstract

The landscape of current processor visualization tools offers a wide variety of programs in terms of information detail, visualization technique and simulation possibilities. However, these tools often lack a guided simulation which would help students understand the discrete information flow of a CPU and the inherent architectural design. Students learning CPU architecture or digital logic often need to resort to basic videos of animations or static images which describe a CPU in the most basic way. Although, the landscape of other simulators with focus on usage in education is big, there is a distinct gap in terms of guided information simulations. This thesis describes the current landscape of CPU simulation tools, shows the need for another simulator in the educational domain, and showcases a developed simulator that focuses on the educational benefit it can provide. The developed simulator, called *Apate*, incorporates features of other simulators and extends the feature set by a guided tour of the CPU with additional information of each step. The information is directly embedded into the simulator. Apate takes the user through a tour of the architecture of a RISC-V based CPU and enhances the effect with a performant visualization. Both development and user interface will be described in this thesis with a concluding usability test by lecturers and students.

Keywords: CPU, Simulator, RISC-V, Electron, educational

Contents

1 Introduction **1**

2 Categorization of existing simulators **3**
2.1 No GUI . . . 3
2.2 Value Level . . . 4
2.3 Structural Level . . . 7
2.4 Stage Level . . . 7
2.5 Gate Level . . . 9
2.6 Transistor Level . . . 10

3 The need for another educational CPU simulator **10**

4 A CPU from the RISC-V perspective **12**
4.1 Instruction Set Architecture . . . 14
4.2 Program Counter . . . 18
4.3 Instruction Cycle . . . 19
4.3.1 Fetch . . . 19
4.3.2 Decode (+ Operand Fetch) . . . 19
4.3.3 Execute (+ Memory Access) . . . 20
4.3.4 Write Back . . . 21
4.3.5 Advance Program Counter (Repeat) . . . 21
4.4 CPU Components . . . 22
4.4.1 Memory from the physical and logical perspective . . . 22
4.4.2 Registers and their functions . . . 24
4.4.3 Control Unit . . . 25
4.4.4 Parsing the instruction name . . . 28
4.4.5 Arithmetic Logic Unit . . . 29
4.4.6 Branch Evaluator . . . 30

5 Apate‘s Interface **31**
5.1 Welcome Screen . . . 31
5.2 Compilation Screen . . . 33
5.3 Simulation Screen . . . 33
5.3.1 Signal wires connecting components . . . 35
5.3.2 Multiplexers selecting signal wires . . . 35

5.3.3 Selective component and signal highlighting . . . 36
5.3.4 Graph subcomponent: Control Unit . . . 36
5.3.5 Graph subcomponent: Branch Evaluator . . . 37
5.3.6 Graph subcomponent: Arithmetic Logic Unit . . . 37
5.4 Guided simulation . . . 37

6 Implementation **38**
6.1 User Interface Implementation . . . 39
6.2 Central Processing Unit Implementation . . . 41
6.2.1 Synthesized HDL CPU Implementation . . . 42
6.2.2 JavaScript CPU Implementation . . . 46
6.3 ELF Parser . . . 48
6.4 Instruction Decoder . . . 51

7 Evaluation **51**
7.1 Technical Evaluation of CPU simulator with unit tests . . . 51
7.1.1 Unit testing the JavaScript ELF file parser . . . 52
7.1.2 Unit testing the instruction decoder . . . 52
7.2 Usability Test . . . 53
7.2.1 Procedure . . . 54
7.2.2 Qualitative results . . . 54
7.2.3 Quantitative results . . . 55
7.2.4 Possible improvements . . . 57

8 Conclusion **61**

Bibliography **62**

List of Figures **66**

List of Tables **68**

List of Sourcecodes **69**

1 Introduction

There are many tools which help students learn in the context of computer science. Alongside a typical presentation and the manuscript of the lecture some additional learning material is often provided. This can include books, videos, animations, websites, or downloadable games. An example tool is GIT which has many interactive tools which help students grasp the features GIT offers in a packaged form (see Katacoda [1]). Some of these tools can even be found in the form of games (see Oh My Git! [2]). In the context of computer architecture there are often simple animations or static images provided to show the inner workings of a central processing unit (CPU) including subcomponents and the flow of data. Although these animations provide a general overview, they lack the option for gaining in depth knowledge. Previous studies already showed the benefit of simulators in computer architecture and digital logic courses [3–6]. However, these simulators (described in section 2) only show the elements of a CPU with either a lack of information or a too descriptive approach of hardware accurate gates.

The targeted user groups for an educational simulator are computer science, embedded systems, electronic engineering, and digital logic students. The biggest benefit of a simulator, in contrast to animations or static images, is the interactivity it can provide. With different goals and previous knowledge, it is often hard to find the correct way of visualizing information in a compact form. An interactive simulation eliminates this problem partially as the information is provided in small pieces which cannot be addresses completely in a single image or animation. This allows students to explore the CPU architecture at their own pace and prevent them from being overwhelmed. Depending on the target group, course, and previous knowledge, the areas of interest might be different. As students can skip parts of the CPU which are not relevant for their course an educational CPU simulator can still satisfy not only their but also other student's requirements without having shortcomings. Some questions which can be answered when using an educational simulator are:

- What are the main elements of a CPU?
- How are the elements of a CPU connected and what is the function of a multiplexer?
- What are the control unit, registers, arithmetic logic unit, branch evaluator, and memory and for what are they used?
- How is the memory structured and how is it read from or written to?
- What is the instruction cycle and what are CPU stages?
- How does the program counter work and relate to the instructions?

- How is an instruction encoded and what is the instruction's respective function?
- How is an instruction executed?
- How does a function call work and what are jump instructions?
- How does compiled C code look?
- What instructions are included in a given instruction set architecture and what are they used for?

When discussing the level of detail these CPU architecture visualizations provide, one must look first at the far ends of the spectrum. One side of the spectrum is a general description of the CPU as a whole, having inputs and outputs. On the other side the CPU is shown as depicted in hardware including hardware gates enabled by nanometer sized MOSFETs. Both sides lack the answer to the question of how a CPU works and often only confuse students more or beg them to ask further questions as the information provided was unsatisfactory. A general description is too general, and the hardware level description is too specialized. A middle ground is needed where subcomponents of the CPU are described detailed enough but not too detailed to be too confusing when first seeing it. The goal of this master thesis is to provide this middle ground in the form of a CPU simulator which can be used in the context of teaching a CPU architecture. The simulator called *Apate* was developed with the only goal of being an educational tool for students of CPU architecture or digital logic classes. The implementation of this simulator, its user interface and a final evaluation is presented as part of this thesis.

The current landscape of CPU simulation tools is described in section 2. With the landscape in mind the need for another simulator is presented in section 3. The developed simulator, which is described in section 5, is based on the RISC-V instruction set architecture explained in section 4. The implementation of the simulator is further described in section 6. As the focus of this tool is the ease of use and the additional value in CPU architecture courses, section 7 provides results of both a technical and a usage evaluation using unit and usability tests.

2 Categorization of existing simulators

Before discussing the need for another CPU simulator, existing CPU simulators are described and analyzed. The term "simulator" is used in a large variety of different environments and the user's expectation is ambiguous. In this thesis "simulator" is understood as a user centered program with a visualization of the CPU's architecture and coherent information. Other classifications like functional simulations, timing simulations, full system simulators or any other classifications mentioned by Akram and Sawalha [7] are not applicable for this thesis. Educational state-of-the-art simulators have therefore been categorized into six distinct groups: No GUI, Value Level, Structural Level, Stage Level, Gate Level and Transistor Level. The proposed CPU simulator landscape is shown in figure 1. The more right on the landscape one goes, the more details of the CPU's architecture are shown. This, however, does not mean that the right most type of visualization offers the most information, as level of detail does not go hand in hand with the provided information. The six levels are described in this chapter and examples for each level are provided. Because of the sheer number of simulators only mentionable tools are stated explicitly.

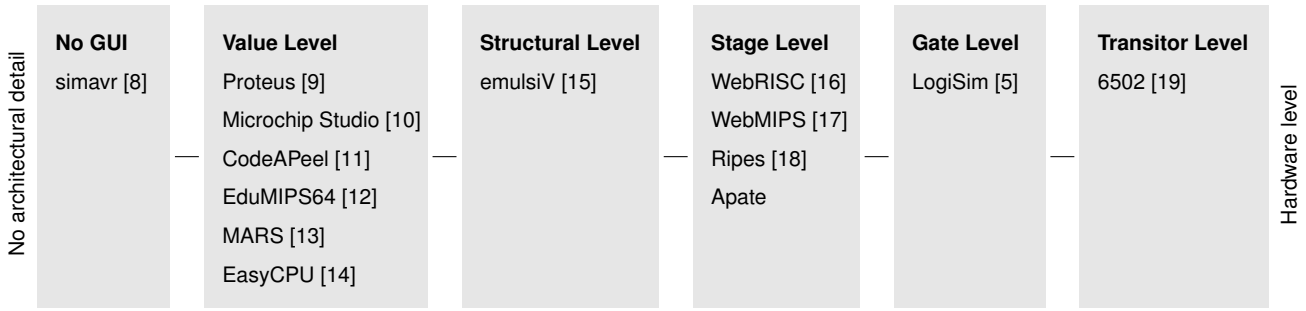

Figure 1: The CPU visualization landscape. The left side starts with simulators which show no architectural CPU details. The more right one goes the more detail will be shown until the hardware is reached.

2.1 No GUI

Simulators shown on the left in figure 1 are command line tools, backend systems or other programs which have no graphical user interface (GUI) or any type of visualization. These can be used as a backend to other simulation visualization. An example is *simavr* [8], which offers no direct GUI but can export waveforms and other information which need to be visualized and analyzed by other tools.

2.2 Value Level

The second to left category in figure 1 does not necessarily show the CPU's architecture, but some of its internal values. These include the memory, registers, special function registers (SFR), program counter and a list of compiled instructions. However, tools of this level do not show the internal state of the CPU, signals, or subcomponents. Known tools of this type are Proteus [9] and Microchip Studio [10] which are briefly described in the next sections. Other simulators of this level include CodeAPeel [11], EduMIPS64 [12] and older tools such as MARS [13] and EasyCPU [14].

Proteus

Proteus is a well-known design and simulation tool for electronics [9]. It offers the ability to design circuits, simulate them on a schematic level and export them as PCB files. Furthermore, integrated chips and microcontrollers can be added to the schematic and co-simulated. This offers students the ability to design and test schematics alongside the microcontroller dependent firmware in a virtualized environment [20]. A microcontroller as part of a schematic can be seen in figure 2. The firmware can be debugged step-by-step, whilst also simulating the circuit. This enables the user to identify not only firmware problems but also their effects on the circuit schematic. The microcontroller simulation offers instruction stepping (figure 3), a view of the contents of the microcontroller‘s memory (figure 5) and the value of the CPU's registers, including the PC and the status registers (figure 4).

As the goal of Proteus‘ co-simulation is stated as purely for use in agile development of embedded systems, it does not feature educational methods or additional information going beyond the basic needs of embedded systems development. The internal CPU architecture, stages, how an instruction is decoded or more general information about registers and instructions are therefore not visible.

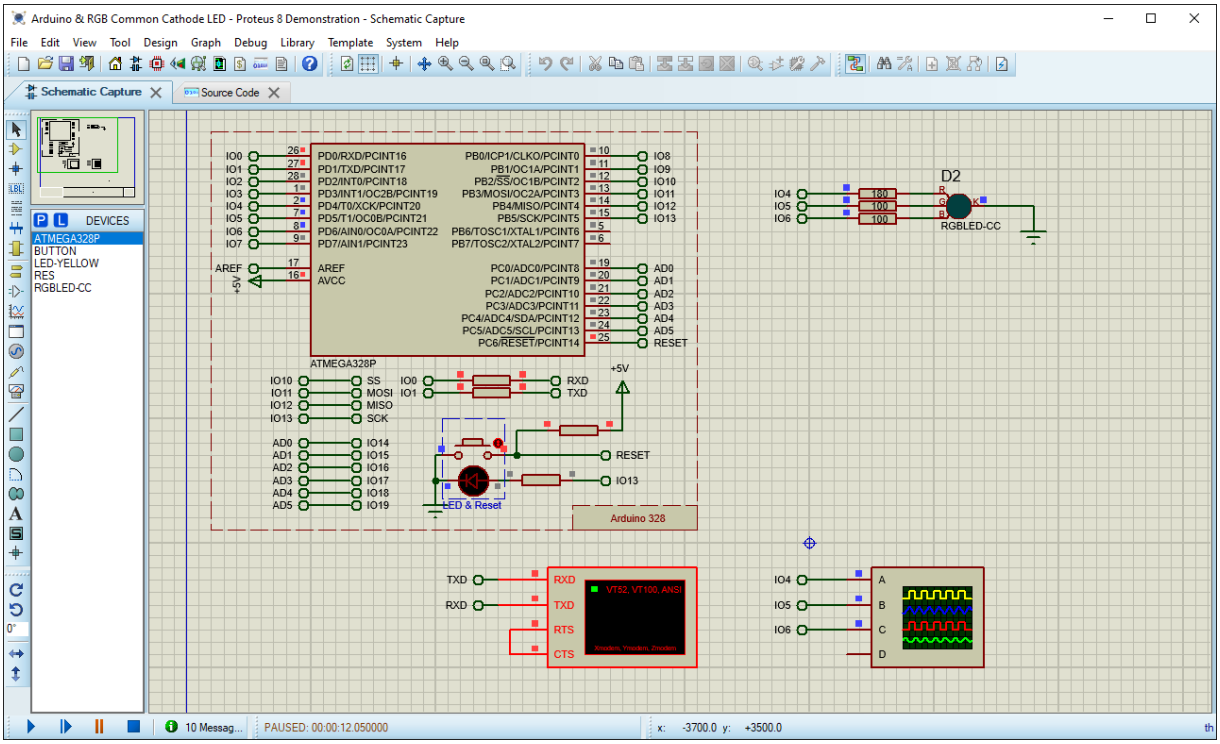

Figure 2: Proteus‘ schematic editor with electronics simulation.

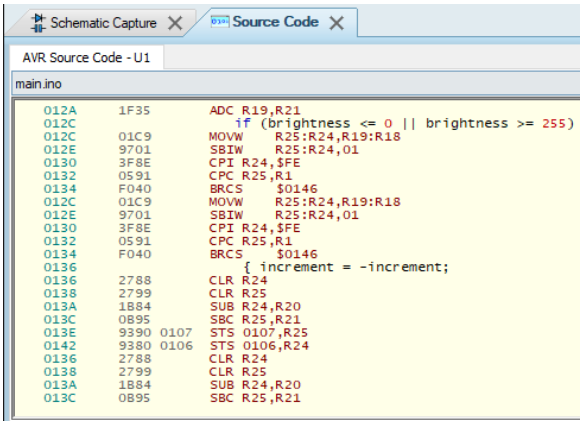

Figure 3: Proteus‘ source code view with disassemble instructions.

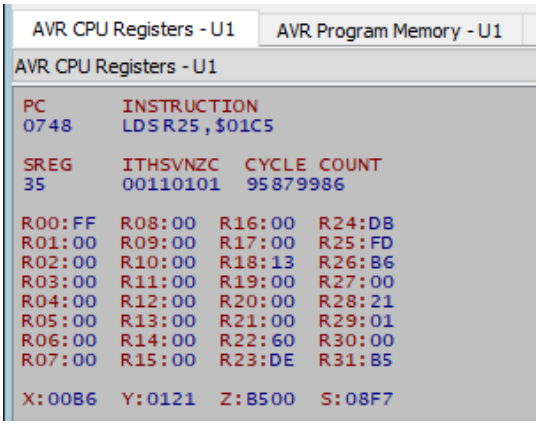

Figure 4: Registers of the AVR processor shown in Proteus.

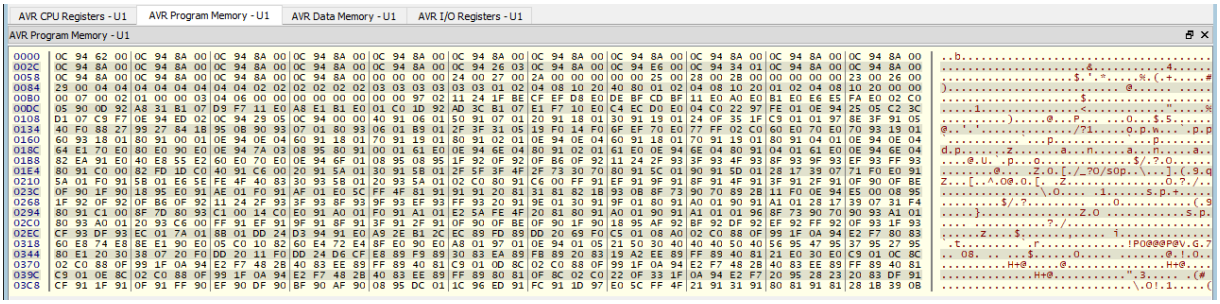

Figure 5: Memory of the CPU which is updated after each instruction shown in Proteus.

Microchip Studio

Microchip Studio, previously known as Atmel Studio, is an IDE with simulation features [10]. The interface is shown in figure 6. Although it is primarily used as an IDE, it provides a simulation of AVR and SAM microcontrollers to debug the developed code. It does not provide a schematic simulation, in contrast to Proteus, but has additional IDE features. Microchip Studio's simulation allows embedded developers to view the registers, memory and special function registers and step through instructions and the source code. A visualization of the architecture or other internal components is not provided.

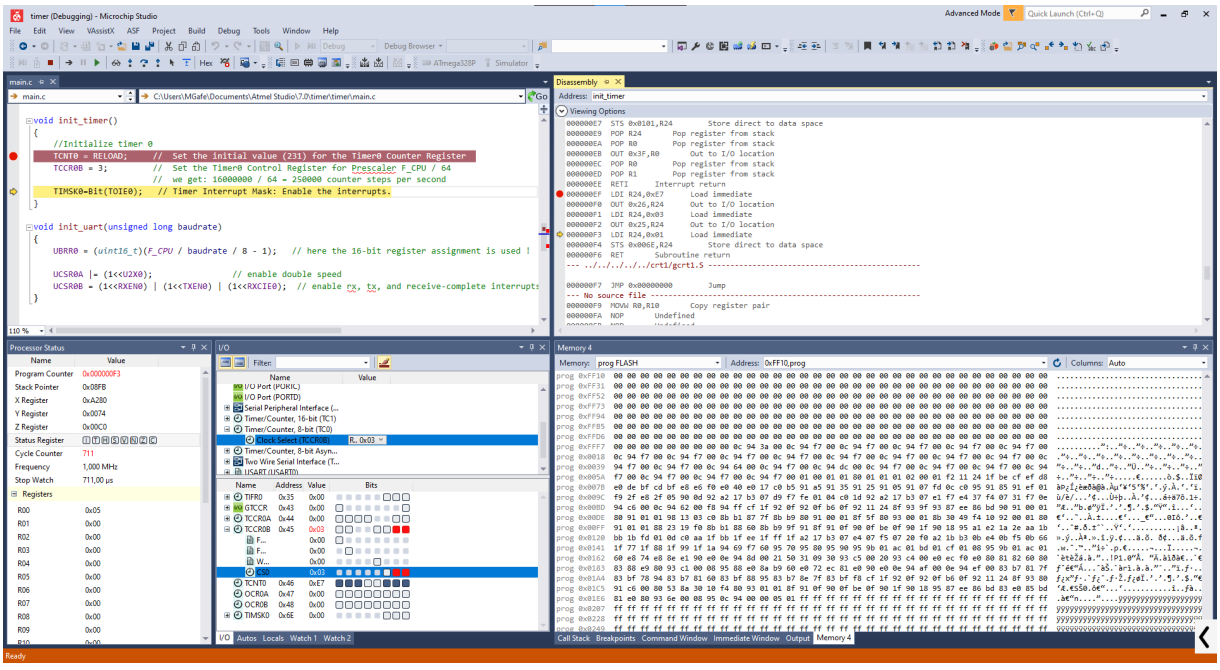

Figure 6: A simulation in Microchip Studio.

2.3 Structural Level

Structure level visualizations are often seen as simple animations with balls moving around the CPU representing data. The CPU of these visualizations often only contains an ALU, a block for the registers, a block for the memory and some paths between these. These typically do not provide more information on the subcomponents and their inner architecture. Although, their in-depth information is limited, they are a good tool for the basic overview of a CPU and their major components, if executed correctly. Simulator examples are emulsiV [15], but most structural level visualizations are video animations or still pictures.

emulsiV

An example for a structural level visualization is emulsiV [15]. It visualizes the basic components of a CPU and their data path. Each stage can be stepped through and animated values travel from one component to another.

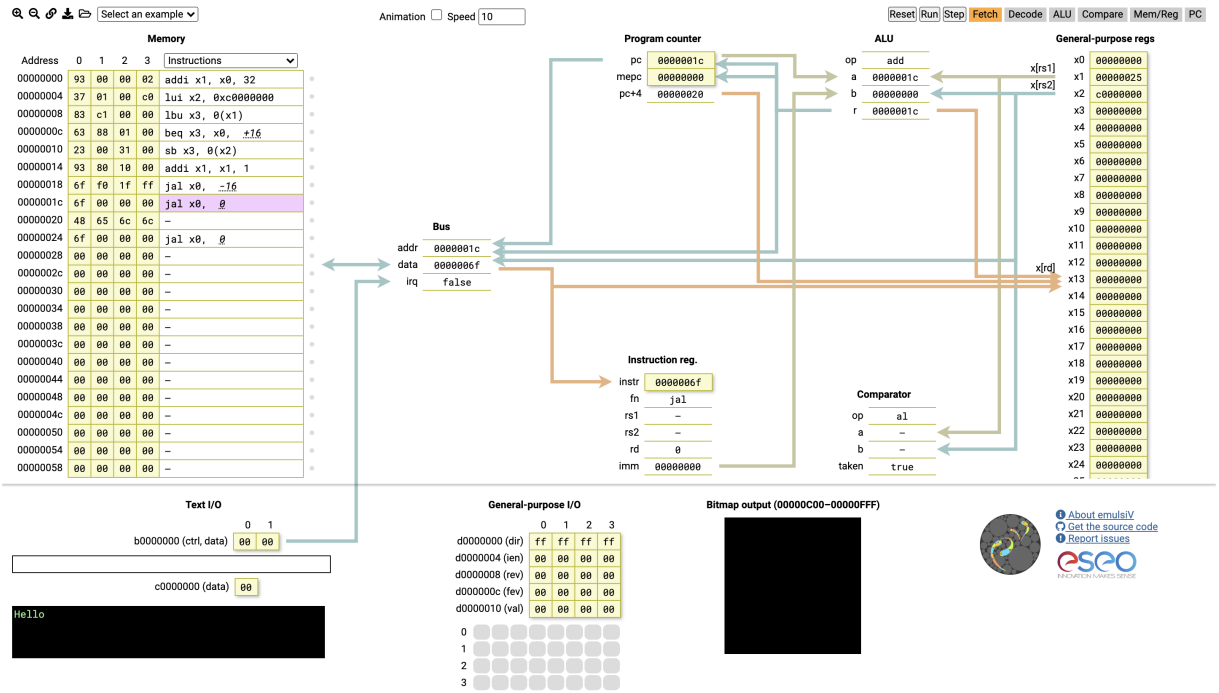

Figure 7: The web-based structural visualization of emulsiV

2.4 Stage Level

The fourth level of visualization is the stage level. This category contains all subcomponents of the CPU and the flow between them structured by CPU stages which are described in section 4.3. In comparison to the structure level these provide additional information on the chronology of events inside a CPU. The basic instruction cycle, as discussed later in detail, greatly improves

the understandability of the subcomponents, their interconnection, and tasks. Examples are WebRISC-V [16] (based on WebMIPS [17]) and Ripes [18].

WebRISC-V

WebRISC-V is a web-based tool to visualize the stages of a 5 stage pipelined RISC-V based processor [16]. Programs can be written in assembly and loaded into the processor's memory to be executed. All elements can be clicked to show more information about the element in question. With the "Step forward" button a clock cycle is executed, and the instructions shift to the next CPU stage. This allows the user to follow the instruction through the CPU.

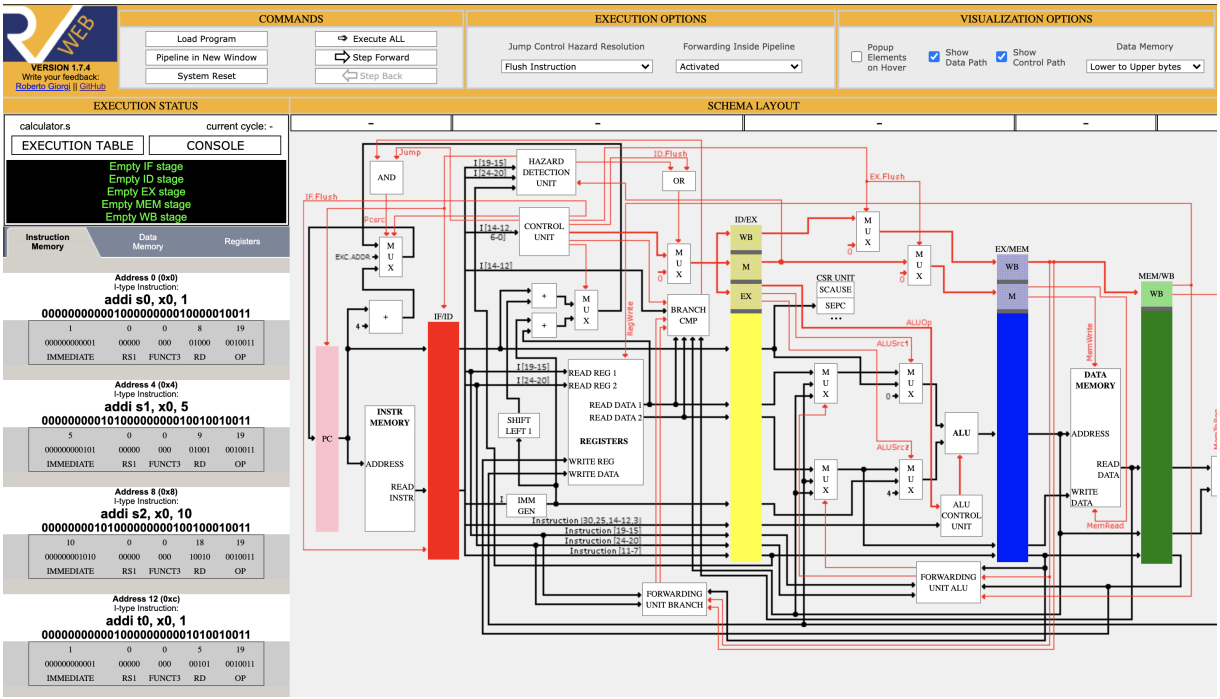

Figure 8: A simple calculator demo in WebRISC-V

Ripes

Ripes is another stage level simulator [18]. It can visualize multiple RISC-V architectures and simulate them showing registers, the memory and instructions. Information on the subcomponents, including registers and ALU, is not provided.

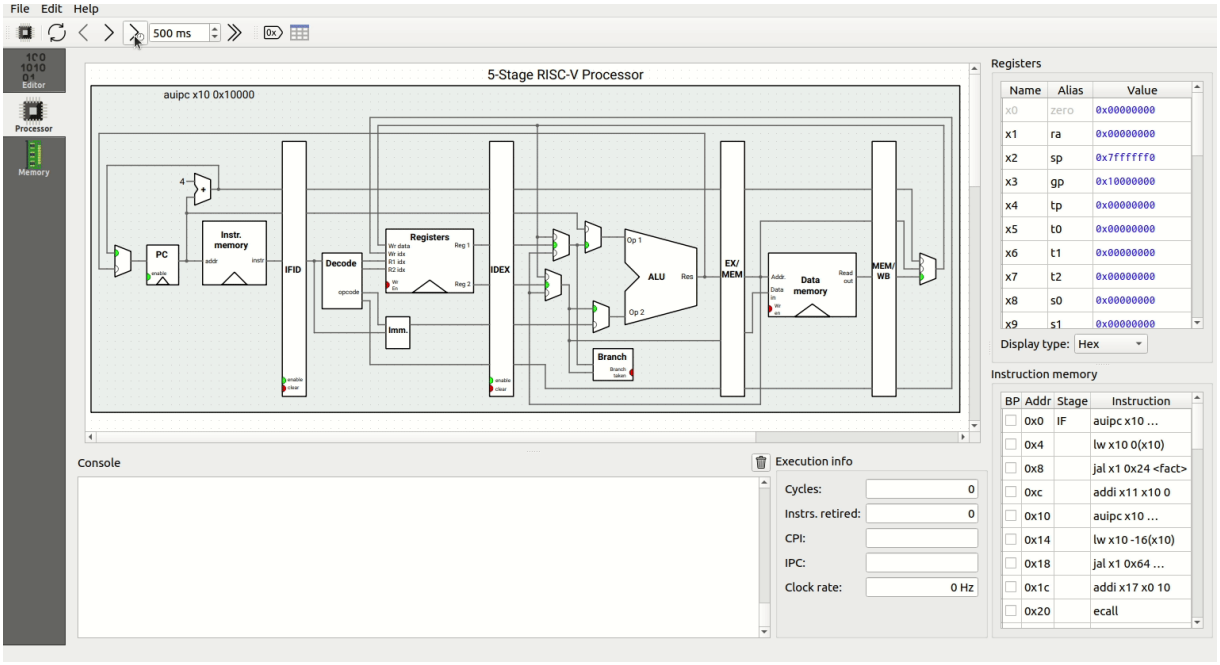

Figure 9: Simulation done in Ripes

2.5 Gate Level

The second to last category is on the gate level. Single gates are shown and connected via wires which offers the ability to gain detailed knowledge of how parts of the CPU are implemented. This knowledge is, however, very specific as the implementation differs between architectures and CPU designers. Simulators of the gate level category are often declared as logic simulators as they provide all features to build a full-fledged CPU and have been analyzed in different surveys before for use in education [3, 6]. Gate level visualizations lack a big picture of the CPU, however. A known gate level simulator is LogiSim [5].

LogiSim

LogiSim is most notably in this category and although it has not been further developed since 2011 it is still used today [5]. It provides basic gates and logic blocks to create logic circuits to design small circuits or whole CPUs, including decoding, and processing. Figure 10 shows one of many RISC-V implementations developed purely in LogiSim [21]. As mentioned, it, and other gate level simulators focus on the logic itself and not on the context of the used elements. The usefulness of LogiSim in teaching digital circuit simulations cannot be denied [6]. However, in the regard of computer architecture, other tools which are part of the structural and stage level are of greater value as they provide a more generic and overall view of the CPU.

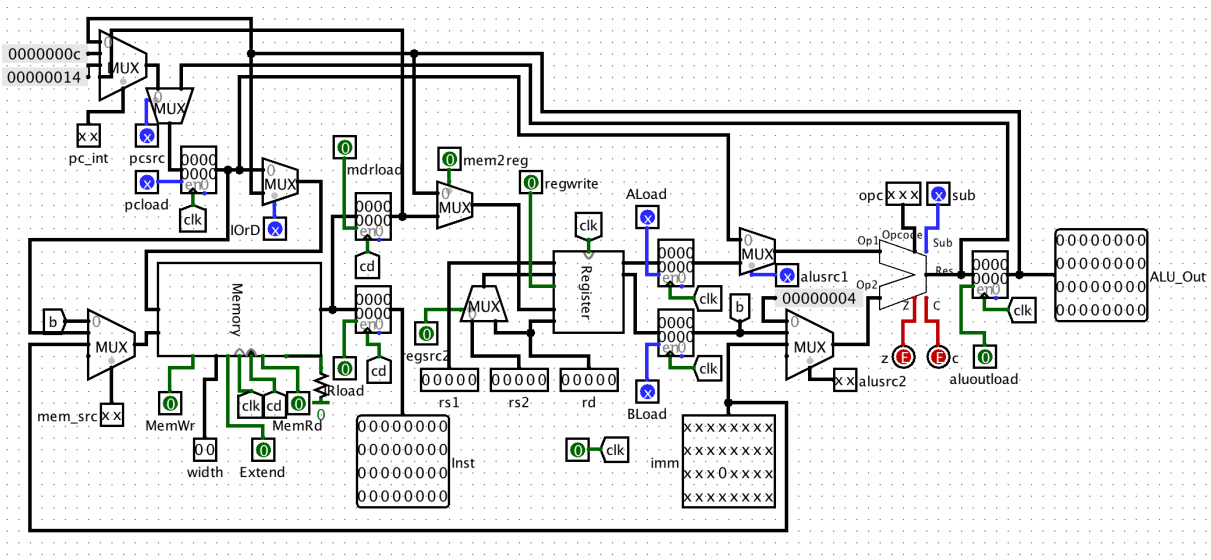

Figure 10: Logic simulation of a RISC-V CPU [21] in LogiSim

2.6 Transistor Level

Another, not often seen visualization, is the one of the hardware itself. The 6502, 6800 and ARM1 CPU have been visualized on a transistor level and can be explored on a dedicated website[1] [19]. With the goal of teaching CPU architecture on a more schematic level, this approach is not explored further in this thesis but nevertheless interesting for advanced users.

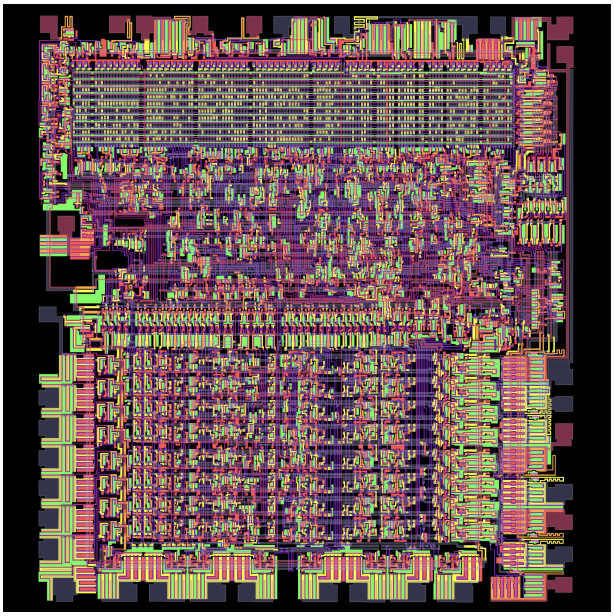

Figure 11: Transistor level simulation of the 6502 processor [19]

[1] http://www.visual6502.org/

3 The need for another educational CPU simulator

The mentioned state-of-the-art simulators show the need for another type of educational tool to help students grasp a CPU architecture even faster. All mentioned programs lack additional information which could be seamlessly integrated into the simulator. Although some provide insight into the values of the CPU, showing the memory and registers, they lack the description necessary for students to easily understand the shown element. In the example of information about instructions, most simulators provide the assembly of the instruction, but not what the instruction does, its name, values of the instruction or a mathematical formula. In the case of registers, they are declared as R0 to R31 in most cases, the register's name is not shown, however. Some, like Microchip Studio, provide the names of special function registers, which include timers, GPIO and other control registers of the SAM and AVR device family. However, this simulator lacks the inner logic of the CPU as it only shows values. Learning from other simulators the following requirements for a new kind of educational CPU simulator have been set:

- View of the memory in hexadecimal, decimal, and ASCII
- View of the registers with the corresponding name and the current value in hexadecimal, decimal, and ASCII
- A list of assembly instructions with name, a textual description, parsed values which are part of the instruction, and a definition of the registers the instruction operates on
- Visualization of the whole CPU architecture, including all relevant blocks and stages, but not too complicated to get lost
- Highlighting of the currently used elements of the CPU to allow the user to focus on the execution of the current instruction and hide not needed elements, wires, and components
- A simple and efficient way to navigate the CPU architecture while observing other values
- Additional information including descriptions, names and values of components and wires when selecting or hovering over them
- Changing values of the wires connecting the CPU components or values inside components should be baked into the visualization

- Using an existing simple instruction set to stop students from learning a new custom language
- A step-by-step guide through every relevant component the CPU uses
- The ability to write custom code with an integrated toolchain
- Simple to use on every major operating system, preferably web-based

For these reasons, a new simulator has been developed named *Apate*. It can be categorized into the stage level category described in section 2.4, as this level provides the best architectural overview of a CPU. The name is derived from the personification of deceit, as a simulator does. It provides a more in-depth educational approach, including the direct integration of additional information, a step-by-step guide through the CPU, examples, and the ability to write custom programs with an embedded C toolchain.

A simulation was preferred over animations because of the interactivity it provides. In a simulation, students have the option to step through the code and explore the architecture by themselves at their own pace. Additionally, the code and the inherent behavior can also be changed to simulate a multitude of situations which cannot be addressed by single video animations.

A non-pure web approach was chosen for multiple reasons. Even though others have shown the possible advantages of a download-free online approach [3, 16], a local installation was seen as necessary. This approach was chosen as only one browser (Chromium) and version needs to be supported, allowing for a better end user experience, as other browsers may face stability issues with not well tested cross browser JavaScript code or CSS. Furthermore, Apate offers the ability to compile custom code by the user. This requires an installed toolchain on the user's device. Apart from that, the toolchain can also be used for other tools like *objdump* or *readelf* which can be called via the user interface. A pure web-based interface would need to relocate this functionality to a server which would contrast with the otherwise possible offline experience. Another possible solution for calling locally installed applications from web pages is by registering a custom URI scheme. This would allow the interface to be a standalone web page and requires the user to install an application containing the toolchain when using these features. Another approach to embed the toolchain into a web page is recompiling the toolchain's sources to WebAssembly which has already been done with the *clang* compiler. With the use of the Electron framework a single source compiles to Windows, macOS and Linux. The simulator is therefore still compatible with all operating systems.

Apate uses the RISC-V Base Integer Instruction Set, short RV32I, which will be described in detail in the following chapter 4. RISC-V was chosen because of the gaining traction and its open-source aspect. The base integer subset was chosen to implement a simpler CPU architecture and visualization. This means that the developed CPU does not include multiplication, division, floating point operations, atomic operations, or interrupts.

4 A CPU from the RISC-V perspective

This chapter describes the RISC-V instruction set architecture which is used in the developed simulator.

Every program, large or small, consists of many simple steps which do not go beyond simple algebra and logic equations. As modern computers have evolved, the speed at which these steps are executed has greatly increased. These simple steps can be used to realize complex mathematical functions, render images to screens or drive autonomous vehicles.

A CPU (Central Processing Unit) can be described as a simple executor. It takes trivial commands, mentioned above as steps, and executes them. These commands are known as instructions. Common instructions include addition of two values, subtraction, and loading / storing data from and to the memory. To understand how basic instructions work, first the register needs to be disclosed. The register is an addressable number of descriptive flexfields (DFF). It is used as a fast volatile memory to store small values between instructions. Each element in the register has a unique address and an associated name which describes the value it holds e.g., Stack Pointer, Return Address. Functions of each register are described in section 4.4.2. The width and number of registers depends on the CPU and can go up to 64 addresses in RISC-V with a width of 32 bits.

Example of a basic instruction
Given an imagined instruction `add r0 r1 r2` which translates to "Add the value of register 1 to register 2 and save the result in register 0" equals the formula `r0 = r1 + r2`. If this instruction would be executed the values of r1 and r2 would be loaded from the registers, added together, and stored back to the register r0. In this case the `add` instruction defines what the register values are used for.

Depending on the CPU the order or names of instructions can be different. The instructions are described in-depth in section 4.1. A program typically consists of many instructions. To keep track of the current instruction the program counter is used. After every instruction the program counter is advanced by one instruction and the next instruction is executed. See figure 12. More about the program counter is described in section 4.2. To achieve this functionality of reading an instruction - which is stored as binary in the memory - to understanding it and then executing it, a CPU consists of multiple stages which are described in section 4.3.

As mentioned above, different CPU architectures often have different stages and wording. The description in the next sections focuses on the *RISC-V 32-bit base integer instruction set architecture* (ISA) short RV32I. This architecture subset supports integer addition, subtraction, and shifting. To enable multiplication or division shims[1] must be used. GCC provides these shims when compiling for an architecture which does not support a needed functionality. As

[1] A shim is a library for intercepting API calls. It handles the intercepted operation itself or redirects the functionality. Shims can be used to run programs on platforms which they were not developed for.

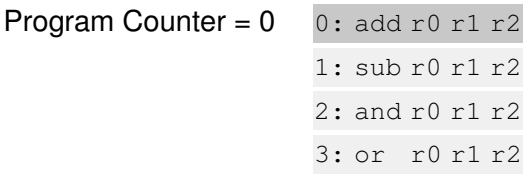

Figure 12: List of instructions addressable by the program counter. The instruction currently marked will be executed. After the execution the Program Counter will be advanced by one. (View the document in *Adobe Acrobat Reader* to see animations)

the ISA only specifies a list of instructions and their formatting, which are described in section 4.1, the CPU architecture can also differ between CPUs with the same ISA. The architecture referenced in this thesis is a specifically for this thesis developed architecture which is used as a visualization basis for the developed educational simulator. The architecture is explained in the next section 4.1.

4.1 Instruction Set Architecture

The length, layout and encoding of instructions are dependent on the ISA (Instruction Set Architecture). The ISA defines the necessary non ambiguous definition to parse an instruction. Typically, an instruction consists of an operation code (opcode) and some operand. The opcode is a defined bit sequence representing the function of an instruction. The function will be executed with the given operands which can be a value directly encoded in the instruction or an address to a register or memory. However, as the ISA only defines the length, layout and encoding of the instruction the implementation of how this instruction is executed is not defined. This allows multiple CPU architectures to execute the same instructions. Take AMD and Intel CPUs as an example. Both execute x86 instructions but are internally different. This allows a separation between physical implementation of the CPU and the software running on it. Code compiled to one ISA can therefore run on every CPU which implements the given ISA. However, as some microcontrollers / System-on-Chips offer different ISA extensions across different manufacturers, code sometimes has to be complied again with different compilers or arguments for these architectures to utilize these extended instructions. See for example the multiplication instructions extension (RV32M) of the RISC-V ISA [22] which enables the `mul` (multiplication), `div` (division) instruction and more. The instruction extension can be utilized in a RISC-V based CPU architecture implementing the multiply extension by setting the `-march=rv32im` flag while compiling [23].

How the binary representation can be parsed depends on the ISA. Table 2 shows the encoding for RISC-V instructions. Every instruction is 32 bits long and consists of the opcode, from bit 0 to 6, and other values depending on the instruction type which can be differentiated by the opcode. The possible values are shown in table 1.

A list of all opcodes (field: Opcode) and their corresponding instruction type (field: Type)

Short name	Long name	Description
opcode	**Operation Code**	Specifies what instruction type the instruction is.
funct3	**Function 3**	Further specifies the instruction by differentiating between instructions with the same opcode.
funct7	**Function 7**	Specifies the instruction even further if the opcode and funct3 are the same. See `add` and `sub` which both have the same opcode and funct3 and can only be differentiated by funct7.
rd	**Register Destination**	The register destination address where the result of the calculation is stored. This is only an index to the register.
rs1	**Register Source 1**	One of the two register sources. These are only indices to the register. To get the value stored in the register the given index needs to be looked up in the register.
rs2	**Register Source 2**	The second register source used for instructions which require two values (*OP* and *BRANCH* instructions).
imm	**Immediate Value**	A value encoded directly into the instruction. This value is used for example when writing the following C code: `int a = b + 5` which translates to assembly `addi r1 r2 5`. The value five is directly encoded in the `addi` instruction.

Table 1: All parsable values in an RISC-V instruction and their description

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type (Register)
imm[11:0]			rs1	funct3	rd	opcode	I-type (Immediate)
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type (Store)
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type (Branch)
imm[31:12]					rd	opcode	U-type (Upper Immediate)
imm[20|10:1|11|19:12]					rd	opcode	J-type (Jump)

Table 2: RISC-V ISA encoding for each instruction type [22]. The instruction types correspond to the type listed in table 3.

can be found in table 3. The table shows the instructions (1) with its name in assembly `add`, `sub`, etc., (2) a longer name of that instruction, (3) the aforementioned type which defines the encoding, (4) the opcode which defines the type, (5) funct3 and (6) funct7 for differentiating between instructions of the same type and opcode, and (7) an description which depicts the mathematical function of the instruction.

Table 3: RISC-V base integer instructions [22]. Bit selectors are depicted as for example [0:3], selecting bit 0 to 3. The letter M depicts the program memory.

Inst	Name	Type	Opcode	funct3	funct7	Description (C)	Note
OP Instructions							
`add`	ADD	R	`0110011`	`0x0`	`0x00`	`rd = rs1 + rs2`	
`sub`	SUB	R	`0110011`	`0x0`	`0x20`	`rd = rs1 - rs2`	
`xor`	XOR	R	`0110011`	`0x4`	`0x00`	`rd = rs1 ^ rs2`	
`or`	OR	R	`0110011`	`0x6`	`0x00`	`rd = rs1 | rs2`	
`and`	AND	R	`0110011`	`0x7`	`0x00`	`rd = rs1 & rs2`	
`sll`	Shift Left Logical	R	`0110011`	`0x1`	`0x00`	`rd = rs1 << rs2`	
`srl`	Shift Right Logical	R	`0110011`	`0x5`	`0x00`	`rd = rs1 >> rs2`	
`sra`	Shift Right Arith*	R	`0110011`	`0x5`	`0x20`	`rd = rs1 >> rs2`	msb-extends
`slt`	Set Less Than	R	`0110011`	`0x2`	`0x00`	`rd = (rs1 < rs2)?1:0`	
`sltu`	Set Less Than (U)	R	`0110011`	`0x3`	`0x00`	`rd = (rs1 < rs2)?1:0`	zero-extends
OP-IMM Instructions							
`addi`	ADD Immediate	I	`0010011`	`0x0`		`rd = rs1 + imm`	
`xori`	XOR Immediate	I	`0010011`	`0x4`		`rd = rs1 ^ imm`	
`ori`	OR Immediate	I	`0010011`	`0x6`		`rd = rs1 | imm`	
`andi`	AND Immediate	I	`0010011`	`0x7`		`rd = rs1 & imm`	
`slli`	Shift Left Logical Imm	I	`0010011`	`0x1`	`imm[5:11]=0x00`	`rd = rs1 << imm[0:4]`	
`srli`	Shift Right Logical Imm	I	`0010011`	`0x5`	`imm[5:11]=0x00`	`rd = rs1 >> imm[0:4]`	
`srai`	Shift Right Arith Imm	I	`0010011`	`0x5`	`imm[5:11]=0x20`	`rd = rs1 >> imm[0:4]`	msb-extends
`slti`	Set Less Than Imm	I	`0010011`	`0x2`		`rd = (rs1 < imm)?1:0`	
`sltiu`	Set Less Than Imm (U)	I	`0010011`	`0x3`		`rd = (rs1 < imm)?1:0`	zero-extends
LOAD Instructions							
`lb`	Load Byte	I	`0000011`	`0x0`		`rd = M[rs1+imm][0:7]`	
`lh`	Load Half	I	`0000011`	`0x1`		`rd = M[rs1+imm][0:15]`	

`lw`	Load Word	I	`0000011`	`0x2`		`rd = M[rs1+imm][0:31]`	
`lbu`	Load Byte (U)	I	`0000011`	`0x4`		`rd = M[rs1+imm][0:7]`	zero-extends
`lhu`	Load Half (U)	I	`0000011`	`0x5`		`rd = M[rs1+imm][0:15]`	zero-extends
STORE Instructions							
`sb`	Store Byte	S	`0100011`	`0x0`		`M[rs1+imm][0:7] = rs2[0:7]`	
`sh`	Store Half	S	`0100011`	`0x1`		`M[rs1+imm][0:15] = rs2[0:15]`	
`sw`	Store Word	S	`0100011`	`0x2`		`M[rs1+imm][0:31] = rs2[0:31]`	
BRANCH Instructions							
`beq`	Branch ==	B	`1100011`	`0x0`		`if(rs1 == rs2) PC += imm`	
`bne`	Branch !=	B	`1100011`	`0x1`		`if(rs1 != rs2) PC += imm`	
`blt`	Branch <	B	`1100011`	`0x4`		`if(rs1 <  rs2) PC += imm`	
`bge`	Branch $\geq$	B	`1100011`	`0x5`		`if(rs1 >= rs2) PC += imm`	
`bltu`	Branch < (U)	B	`1100011`	`0x6`		`if(rs1 <  rs2) PC += imm`	zero-extends
`bgeu`	Branch $\geq$ (U)	B	`1100011`	`0x7`		`if(rs1 >= rs2) PC += imm`	zero-extends
JAL Instructions							
`jal`	Jump And Link	J	`1101111`			`rd = PC+4; PC += imm`	
JALR Instructions							
`jalr`	Jump And Link Reg	I	`1100111`	`0x0`		`rd = PC+4; PC = rs1 + imm`	
LUI Instructions							
`lui`	Load Upper Imm	U	`0110111`			`rd = imm << 12`	
AUIPC Instructions							
`auipc`	Add Upper Imm to PC	U	`0010111`			`rd = PC + (imm << 12)`	
SYSTEM Instructions							
`ecall`	Environment Call	I	`1110011`	`0x0`	`imm=0x0`	`Transfer control to OS`	
`ebreak`	Environment Break	I	`1110011`	`0x0`	`imm=0x1`	`Transfer control to debugger`	

Example of parsing an instruction
Given the raw instruction binary `0b00000000001000001000000100110011`. The first 7 bits (0-6) starting at the least significant bit are the opcode `0110011` which indicate according to table 3 the instruction type R. Type-R consists of three operands: rd (Register Destination), rs1 (Register Source Address 1) and rs2 (Register Source Address 2). Both funct3 and funct7 are only used, alongside the opcode, to differentiate between different instructions. The instruction binary can therefore be separated, according to table 2, to `0b 0000000 00010 00001 000 00010 0110011`: opcode = `0110011`, rd = `00000`, funct3 = `000`, rs1 = `00001`, rs2 = `00010`, funct7 = `0000000`. Looking up the opcode, funct3 and funct7 in table 3, this instruction equals `add rd rs1 rs2`. When also inserting the parsed operands (rd, rs1, rs2) the assembly resembles `add r0 r1 r2` ('r0' meaning register with address zero). In the description field of the table the formula can be seen. The implementation in the CPU of this instruction is not defined by an ISA.

4.2 Program Counter

The PC (Program Counter) keeps track of the next instruction location in the memory to load and execute. As shown in figure 12 the program counter is advanced after every instruction execution. In a single threaded CPU, there is one program counter for the current process or program running. In contrast to a multi-threaded CPU, which has multiple program counters for each process currently running at the same time [24]. As the developed architecture for *Apate* should be simple, only one program counter was implemented.

The program counter is stored as an internal CPU register. This should not be mistaken with the CPU register described in section 4.4.2. The program counter register is not part of the ISA defined registers and therefore cannot be changed. It is solely controlled by the CPU itself in the *Advance Program Counter* stage which is introduced in section 4.3.

As the program counter points to the memory location where the next instruction is stored it may not be advanced by one but by the byte length of the instruction. In the case of the RISC-V ISA an instruction is 32 bits long which equals four bytes. Each instruction is therefore four bytes (or memory locations) long. The first instruction is at memory address zero. The second at memory address four and the third starts at address eight.

The program counter can also be changed by *BRANCH*, *JAL* - Jump and Link, *JALR* - Jump and Link Register and *AUIPC* - Add Upp Imm to PC instructions. In these cases, the program counter is advanced by the immediate value which is encoded in these instructions. Their function can be seen in table 3.

4.3 Instruction Cycle

Each instruction passes through different stages in the CPU to achieve its execution goal. To have a general understanding of the instruction cycle each stage is listed in figure 13 and described in later subsections.

Fetch → Decode → Operand Fetch → Execute → Memory Access → Write Back → Repeat

Figure 13: Instruction Cycle

The instruction cycle is also known as the fetch-decode-execute cycle. Some CPU architectures have either combined some stages, split, or renamed them. The main concept, however, is the same for every CPU. The instruction is (1) fetched from the memory, (2) decoded so that the CPU knows what the binary data means, (3) the operands are fetched from the register, (4) with the operands a result is calculated which is either stored in the (5) memory or in the (6) register and lastly (7) the cycle is repeated by advancing the Program Counter.

Clocked by the CPU clock the data is passed sequentially from the first stage to the last stage. Each stage therefore takes one clock cycle to finish. The less stages a CPU has the less CPIs (clock cycles per instruction) are needed. As some instructions may not need all stages the CPI is dependent on the instruction. In the case of the *ATmega328P* an `add` instruction takes one cycle to execute and a `ret` (Subroutine return) takes four cycles [25].

In a non-pipelined CPU, each instruction must be finished before another one is loaded. This means, every instruction must pass through every stage before the next instruction is loaded. A pipelined CPU on the other hand loads a new instruction on every clock cycle. This improves the speed greatly by a factor of the number of stages, but also introduces more complexity.

To prevent more complexity, the developed CPU architecture consists of 5 stages and is not pipelined. Each instruction takes 5 clock cycles and the next instruction is only loaded once the last instruction reaches the *Advance Program Counter* stage.

4.3.1 Fetch

For the instruction to be executed it first needs to be loaded from the memory. The address of the instruction in the memory is set by the current program counter. In the next clock cycle the unparsed 32-bit binary instruction is sent to the decode stage.

4.3.2 Decode (+ Operand Fetch)

In the decode stage the unparsed instruction data is decoded into signals the CPU can understand. The decoding itself is accomplished by the control unit (green in figure 14). The control unit gets the raw binary of the instruction as an input and returns signals which control the CPU. These include control signals (blue in figure 14) which represent the instruction and its type and the data path which contains the parsed operands.

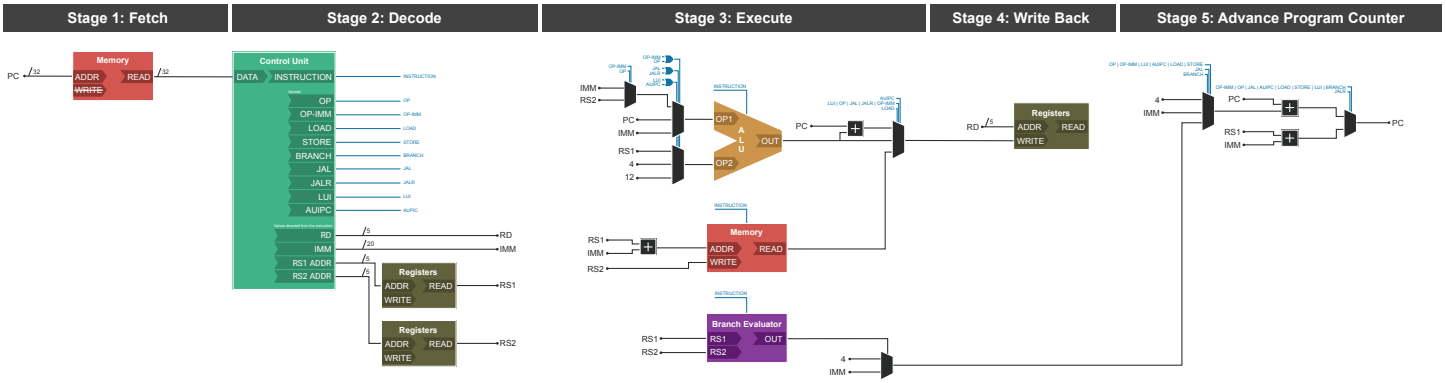

Figure 14: The developed non-pipelined 5 stage CPU architecture of *Apate*. The control path and program counter are marked blue. Depending on the instruction type and the instruction string (blue), which are parsed by the control unit (green), the multiplexers forward different signals.

In the given architectural figure 14, the instruction is represented as a string. To save physical space this would normally be encoded in one-hot encoding or a different method without strings where each instruction has a unique wire which controls parts of the CPU. For simplicity the instruction name is represented as a string. The instruction type of the current instruction (see section 4.1 for all instruction types) are represented as nine unique signals (*OP, OP-IMM, LOAD, STORE, BRANCH, JAL, JALR, LUI, AUIPC*). The parsed operands include the register destination (rd), immediate value (imm) and register source 1 and 2 (rs1 and rs2). The register sources, parsed by the control unit, are only addresses and not the register´s values. To get the given value of the register they need to be looked up first. In this CPU architecture the register lookup, also called *Operand Fetch*, is included in the decode stage. See the register components which use the signals of the control unit and return the values which are stored in the registers. How the instruction is parsed in the control unit is described in section 4.4.3.

4.3.3 Execute (+ Memory Access)

Once the instruction is parsed by the control unit the signals which control the CPU are set. These control the CPU elements via multiplexers (MUX). Multiplexers select one signal from multiple input signals. As shown in figure 14 the execution stage consists of three blocks. The arithmetic logic unit (ALU), memory and branch evaluator. Depending on the instruction one of these blocks is used. ALU instructions include the following instruction types: *OP, OP-IMM, LUI, JAL, JALR, AUIPC*. All of these instructions require functions performed by the ALU. Depending on the instruction used the ALU needs different values. The following example shows the forwarding of the signals to the ALU if the instruction is of type *OP-IMM*.

Example of the operator selection for the ALU with an *OP-IMM* instruction
In the execution stage the top left multiplexer selects between the immediate value (IMM) and register source 2 (RS2). In case of the IMM instruction the immediate value is passed along. The multiplexer to the right selects between the previously selected value, PC, and IMM. In case of the IMM instruction the previously selected value, which was the immediate value, is passed to OP1. OP2 will be set to the value of RS1. Depending on the instruction name the ALU performs different arithmetic or logical functions which are described in the ALU section 4.4.5. The result is passed from the ALU´s OUT port to the next stage to be saved in the register.

In case of an AUIPC instruction the result of the ALU is furthermore added to the PC. The formula for AUIPC is `rd = PC + (imm 12)`. The left shift by 12 is performed inside the ALU and the addition afterwards.

The memory and branch evaluator do not have a such a selection of signals as the ALU. The memory block always used rs1, imm and rs2. It is used by *LOAD* and *STORE* instructions. In the case of *LOAD*, the read value is further passed to the *Write Back* stage. The branch evaluator always compares rs1 and rs2 and is used in *BRANCH* instructions.

4.3.4 Write Back

Once the execution stage is complete a value either from the ALU or a loaded value from the memory needs to be saved in the register. The register address at which the value must be written to is set by the register destination (RD) signal. This signal was also encoded in the instruction. Once this stage was executed the value has been written at the corresponding register location.

4.3.5 Advance Program Counter (Repeat)

The last stage needs to make sure that the PC is advanced. The PC is normally advanced by four, according to the RISC-V ISA, as every instruction is four bytes long. In the case of branches or function calls the next instruction resides at a different location in memory. Therefore, the PC needs to be advanced by more than four. In the case of a function call the `jal` instruction jumps to another location in the program by setting the program counter directly. The formula of this instruction looks like: `rd = PC + 4; PC += imm`. The instruction which would be executed next (PC+4) is saved to a register which is typically called *Return Address* (ra). In the RISC-V ISA the ra register address is r1. The location of the next instruction needs to be saved temporarily to allow the CPU to remember at which point the jump was performed so that a jump back is possible later. Figure 15 shows an example of how and when the program counter is changed and saved. Instruction types which change the program counter directly are

JAL, JALR, and `BRANCH`. After the program counter is advanced the next instruction can be read. The instruction cycle is hereby completed and the next *FETCH* stage is executed.

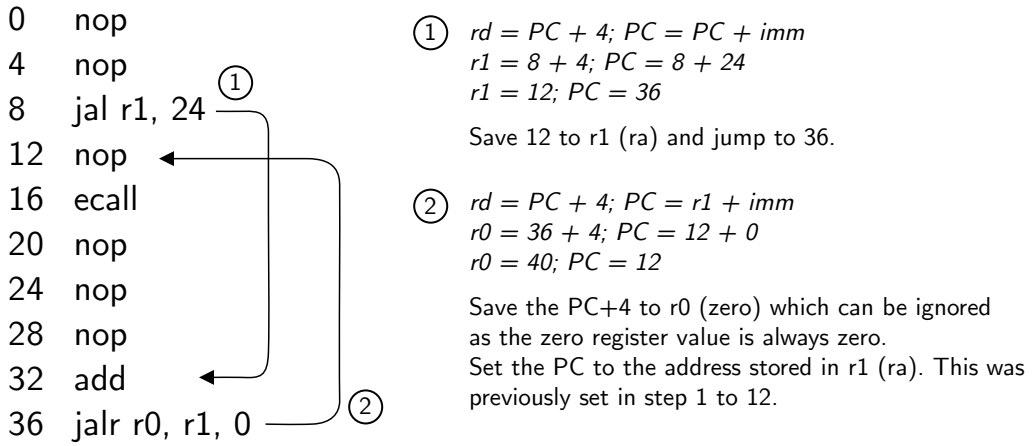

Figure 15: Example jump with `jal` and `jalr`. (1) saves the PC to r1 and jumps to 24. (2) loads the PC from r1 to jump back.

4.4 CPU Components

This section discusses the main components inside a CPU based on the architecture shown in figure 14. These include registers, memory, control unit, arithmetic logic unit and branch evaluator.

4.4.1 Memory from the physical and logical perspective

The memory can be viewed from the physical and logical perspective. From the physical perspective a modern computer typically has multiple memories shown in figure 16. These range from small and fast to large and slow. Registers are an internal core component of the CPU and can be accessed directly by the CPU. The physical memory on the other hand, is located outside of the CPU and needs to be accessed by *LOAD* or *STORE* instructions. Internally the CPU then sends a command to the memory controller which fetches or stores the data to and from the memory. From the instruction perspective it does however not inform the program where the fetched data was stored. It could have been on the RAM or on an external hard drive.

From the logical perspective the memory can be divided into two categories. The data memory and the instruction memory. One stores the instructions of the program and the latter the data. Although both instruction words and data words look the same on the memory, as they are all 32 bit long binary representations in the case of 32-bit systems, they serve different purposes. In the *von Neumann architecture* the instruction and data memory reside on the same memory and are controlled by the same memory management controller. In the

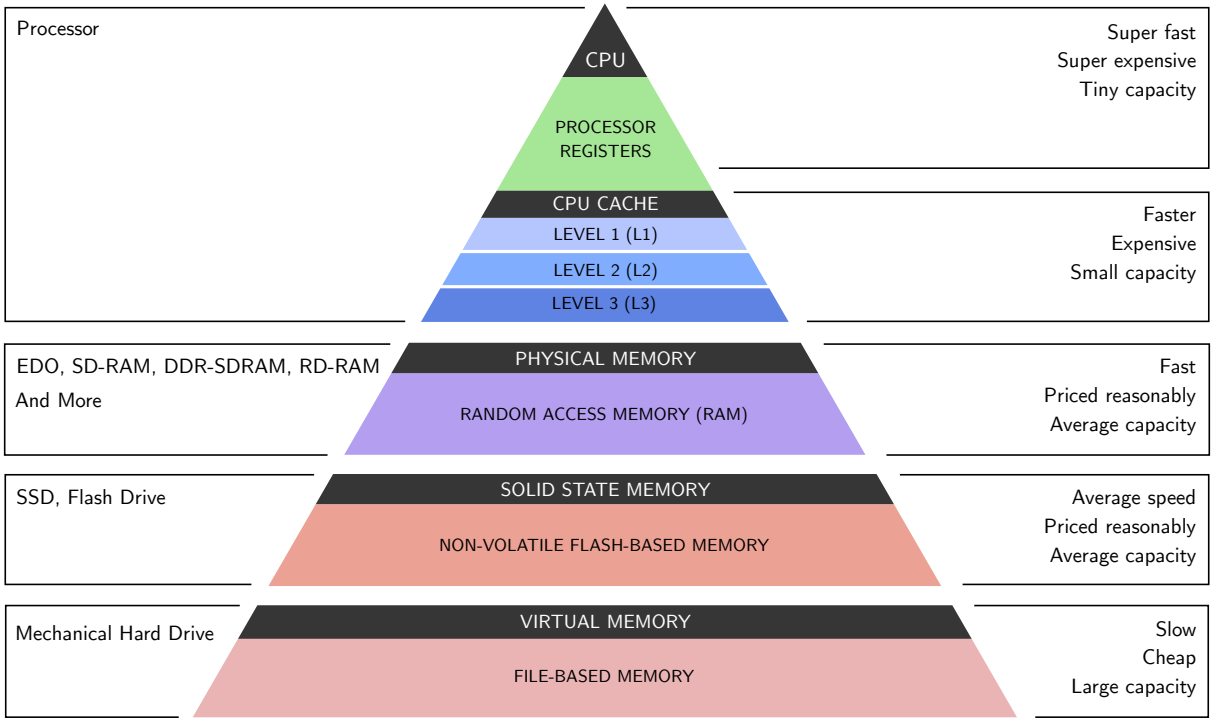

Figure 16: Simplified memory hierarchy based on an illustration by Ryan J. Leng [27].

Harvard architecture instruction and data memory are accessed through different management controllers as they reside in different address spaces. This allows for instructions and data to be fetched at the same time [26]. The *Harvard architecture* is used for example by the *ATmega328p* along other AVR based microcontrollers [25]. The *von Neumann architecture* is used by many x86 processors. *Apate* also uses the latter one and has only one memory containing instructions and data.

The memory is typically separated into multiple sections shown in figure 17. The application data is located beginning with address 0 (top). The application data holds all instructions. This section is also called *.text* when linking. Below the application data the static data (*.bss, .data*) holds initialized or uninitialized global variables which were defined in the source code. This also includes prefilled and allocated arrays or variables. The heap contains static data and can be allocated at runtime. With `malloc()` data can be allocated and freed with `free()`. Starting at the highest memory address (bottom) the stack begins. It contains local variables of functions. If the function returns local variables are overwritten by the next function. Both heap and stack can grow at runtime. Notice that application data and static data sizes are fixed after compilation. If the free memory space between them gets too close and both memory locations touch, the system is out of memory and a stack overflow exception is thrown if the memory management controller provides this feature [24].

To access the memory in *Apate* the CPU has a memory access component called memory which is marked red in figure 18. The component takes in an address and if required a value which should be stored on the memory. Depending on the location of the component and

the instruction which can be `sb` - Store Byte, `sh` - Store Half, `sw` - Store Word, `lb` - Load Byte, `lh` - Load Half, `lw` - Load Word, `lbu` - Load Byte Unsigned, `lhu` - Load Half Unsigned, the component performs different memory actions on the internal memory. The output of the memory component is the loaded bits which are passed as a signal to the next component if a *LOAD* instruction was used. In the *FETCH* stage the CPU only fetches the instruction from the memory, therefore the memory always reads a word containing 4 bytes.

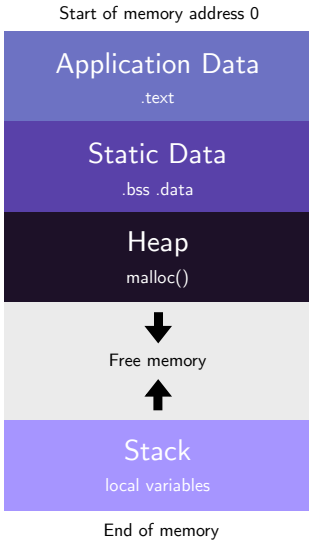

Figure 17: Memory layout

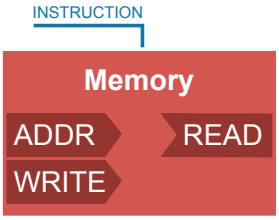

Figure 18: Memory component of the proposed CPU architecture

4.4.2 Registers and their functions

Nearly all instructions operate on registers. The RISC-V ISA defines 32 registers listed in table 4. Although all registers can be directly written to and read from at any time (except x0 which is always zero), every register serves a special purpose. The purpose of each register and their main function can be seen in the description column of table 4. Alongside the register address in column register, some registers have special abbreviations. These help differentiating them when reading assembly for example. How the registers are used is defined by the compiler which transforms the non CPU specific code to CPU dependent instructions. While compiling the compiler selects the registers depending on the use case. In case of a function call a `jal` instruction will therefore most probably use the ra register to store the next program counter to which the CPU shall return later. For an `addi` instruction, function arguments a0 and a1 will most likely be used. The selection of registers also depends on the compiler used, *gcc* and *clang* have for example different outputs.

The registers can be addressed by an instruction in three ways. The register source 1, register source 2 or the register destination. All these fields can be part of an instruction if said instruction needs to use them. Depending on the instruction these fields are set to a register address the compiler defines. The CPU parses these fields and finds the register which should

be used. In case of the register sources the value first needs to be read from the registers. The register destination is used after the execution to set the destination register of the calculated value in the write back stage.

In figure 19 the component to access the registers in *Apate* is shown. One input defines the address which is a value between 0 and 31. In the *DECODING* stage of the CPU a value needs to be read from the registers. The value will be output by the read port and passed to the data path. In the *WRITE BACK* stage a value needs to be written to the registers. The signal containing the value is connected to the write port and the address is set by the register destination (rd).

Register	ABI Name	Description
x0	zero	Zero constant
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporaries
x8	s0 / fp	Saved / frame pointer
x9	s1	Saved register
x10-x11	a0-a1	Fn args/return values
x12-x17	a2-a7	Fn args
x18-x27	s2-s11	Saved registers
x28-x31	t3-t6	Temporaries

Table 4: RISC-V defined registers

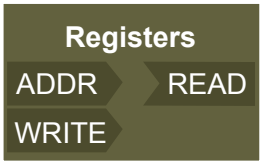

Figure 19: Register component of the proposed CPU architecture

4.4.3 Control Unit

The control unit, as the name suggests, controls the CPU via control signals. The control signals are marked blue in figure 14. These control the signal paths of the CPU via multiplexers and the components via the instruction name which is represented as a string. To control the CPU the control unit needs to parse the instruction. This section discusses the parsing of an unparsed 32-bit instruction into its opcode (*OP*, *BRANCH*, etc.), the instruction name (`add`, `sh`, etc.) shown in table 3, its operators including the immediate value (imm), the return destination (rd) and register source 1 and 2 (rs1, rs2). Figure 14 shows the control unit in green with the mentioned outputs and the unparsed instruction as the input.

Parsing the opcode

The first stage to parsing the instruction is to determine the instruction type for further parsing of the operators and the opcode for controlling the CPU. The instruction type depends on the given opcode and can be of type R (Register), I (Immediate), S (Store), B (Branch), U (Upper

immediate) and J (Jump). Depending on the opcode the encoding of the instruction is different between the types which are shown in table 2.

To get the type the first seven bits of the instruction must be compared with predefined opcode values which can be seen in the instruction list. A schematic of this comparison is shown in figure 20 step 1 on the left. In physical CPUs the hardware to accomplish the comparison is defined by a hardware design language. An example source code 1 shows the comparison of the first 7 bits in *Verilog*, a well-known hardware description language (HDL).

```
1  is_op_imm    <= unparsed_instruction[6:0] == 7'b0010011;
2  is_imm       <= unparsed_instruction[6:0] == 7'b0110011;
3  is_branch    <= unparsed_instruction[6:0] == 7'b1100011;
4  is_load      <= unparsed_instruction[6:0] == 7'b0000011;
5  is_store     <= unparsed_instruction[6:0] == 7'b0100011;
6  is_lui       <= unparsed_instruction[6:0] == 7'b0110111;
7  is_auipc     <= unparsed_instruction[6:0] == 7'b0010111;
8  is_jal       <= unparsed_instruction[6:0] == 7'b1101111;
9  is_jalr      <= unparsed_instruction[6:0] == 7'b1100111;
```

Sourcecode 1: Verilog example for comparing the 7 first bits to predefined opcode bit sequences.

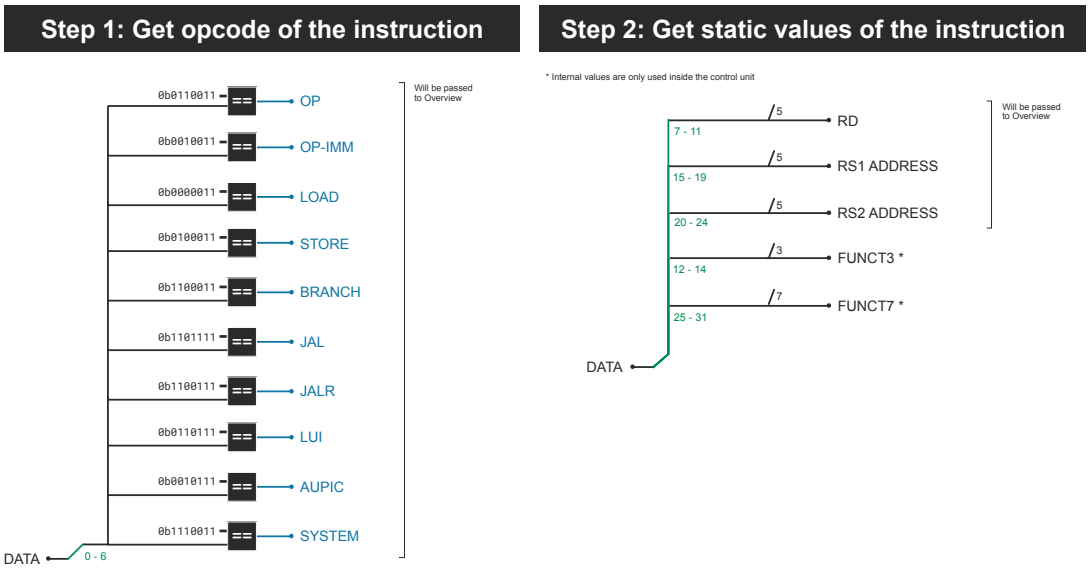

Figure 20: Control unit schematic of step 1 and 2. Parsing of the opcode on the left and parsing of the operands on the right. The green wire represents a bit selection. On the left bit 0 to 6 is selected from the unparsed instruction. On the right the bits per value are selected according to the encoding of table 2. These values are always at the same location in contrast to the immediate value which is parsed in step 3.

Parsing the operands including funct3 and funct7

The operands are extracted from the instruction via bit selections. At what bit locations the values are located is described in section 4.1 and in table 2. All values except the immediate

value are located at the same bit locations and can therefore be extracted from the instruction. For the immediate value a special differentiation must be made between instruction types, which is described in the next section. The bit selection is marked green and selects bits from the 32 bit long unparsed instruction and saves the binary value in one of the operands.

Although all values are located at the same bit location some of them might not be used. See for example funct3 which is only used in an R-type instruction. It will still be parsed, as the parallel logic has no performance impact on CPUs, but its value, which is wrong in the case of funct3 and an instruction which is not of type R, will not be used anywhere in the CPU.

Parsing the immediate value

The only value which is not located at the same location for every instruction is the immediate value. Its bit combination differs between instruction types. In this case, first the instruction type must be determined. In step 1 the opcode has already been decoded. This can be used to determine the instruction type. The types can also be seen in table 3. Instruction type R consists of *OP* instructions. Instruction type I consists of *OP-IMM, LOAD, JALR* and *SYSTEM* instructions. Instruction type S consists of *STORE* instructions. Instruction type B consists of *BRANCH* instructions. Instruction type J consists of the *JAL* instruction and instruction type U consists of *LUI* and *AUIPC*. A schematic of the selection can be seen in figure 21. The Verilog code representation is shown in source code 2. The *OP* instruction is not included as it does not have an immediate value.

```
1  (* parallel_case *)
2  case (1'b1)
3    is_jal:
4      decoded_imm <= $signed({unparsed_instruction[31], unparsed_instruction[19:12],
       ↪  unparsed_instruction[20], unparsed_instruction[30:21]});
5    |{is_lui, is_auipc}:
6      decoded_imm <= $signed({unparsed_instruction[31:12]});
7    |{is_jalr, is_load, is_op_imm, is_system}:
8      decoded_imm <= unparsed_instruction[31:20];
9    is_branch:
10     decoded_imm <= $signed({unparsed_instruction[31], unparsed_instruction[7],
       ↪  unparsed_instruction[30:25], unparsed_instruction[11:8], 1'b0});
11   is_store:
12     decoded_imm <= $signed({unparsed_instruction[31:25], unparsed_instruction[11:7]});
13   default:
14     decoded_imm <= 1'bx;
15 endcase
```

Sourcecode 2: Verilog example for decoding the immediate value and selecting the correct value depending on the opcode. Code adopted from PicoRV32 [28].

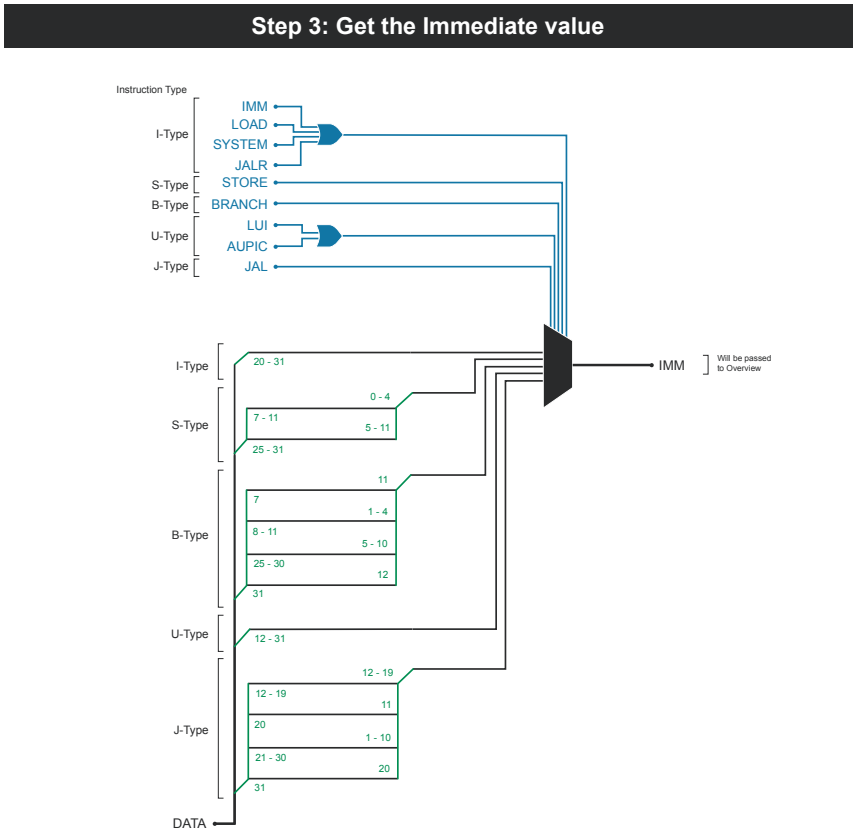

Figure 21: Parsing the immediate value in step 3 of the control unit

4.4.4 Parsing the instruction name

In most CPU architectures the instruction name is not decoded explicitly as *opcode*, *funct3* and *funct7* are used directly to control parts of the CPU (see implementations of [21, 28]) . An example of how *funct3* can be used directly is later shown in figure 25 of section 4.4.6. The process of parsing is often not necessary but can still be established by additional logic in the CPU. This logic uses the *opcode*, *funct3* and *funct7* to create a signal which only corresponds to one instruction of the RV32I instruction set. Figure 22 shows the parsing logic as visualized in *Apate*. Each *opcode* has a distinct schema on how the instruction is decoded (see the encoding in table 3).

Example parsing the name of the `add` instruction

opcode, *funct3* and *funct7* have already been parsed in a previous step. The *opcode* for an `add` instruction is of type *OP*. The possible instructions are limited to this *opcode* type. Next, depending on *funct3* an instruction is chosen. In this case *funct3* equals 0x0 for both `add` and `sub` instruction. Therefore, also *funct7* is parsed and compared, which equals 0x0. Now the instruction can be confirmed as an `add` instruction. An `add` instruction is therefore applicable if the *opcode* equals 0b0110011, *funct3* equals 0x0 and *funct7* equals 0x0. The resulting instruction name is stored in a signal called instruction. This signal is used throughout the CPU to control multiplexers and subcomponents.

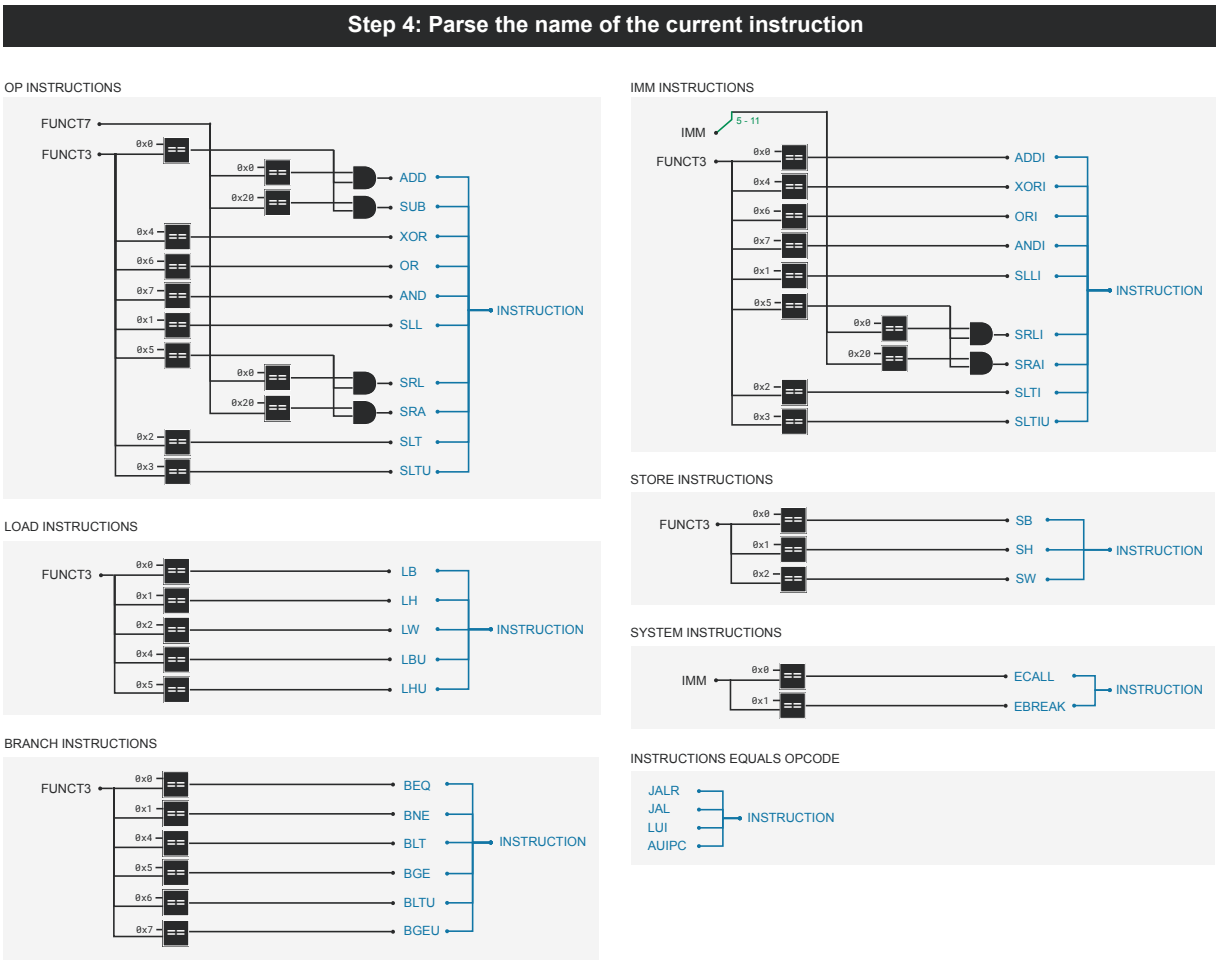

Figure 22: Parsing the instruction name in step 4 of the control unit.

4.4.5 Arithmetic Logic Unit

The arithmetic logic unit (ALU) uses two operands and an instruction to calculate an output. The RISC-V base integer instruction set does not define the functions required by the ALU as it does not define the architecture. However, from the instructions which should be executable

by a RV32I CPU the following ALU operators can be derived: Addition, subtraction, bitwise or, bitwise and, bitwise exclusive or, shift left, shift right, set less than. Multiplication, division, and binary operations are not supported by this ALU. All possible operations need to be combinatorial, meaning they need to be executed without a clock signal. This is required because the execution should happen in one single stage and therefore one clock cycle. The operators shift left and shift right are therefore barrel shifters which shift the operator one by the amount of operator two in a single cycle. The "set less than" operation is used by the "Set Less Than (Immediate)" instructions `slt`, `sltu`, `slti`, `sltiu` and are part of the ALU in the proposed architecture. The "set less than" operation sets the output of the ALU to one, if operator one is smaller than operator two. All operations can be seen in figure 23. The used operation is set by the used instruction. The multiplexer shown on the right site selects the relevant output. All other operations are executed nonetheless as the operations are combinatorial.

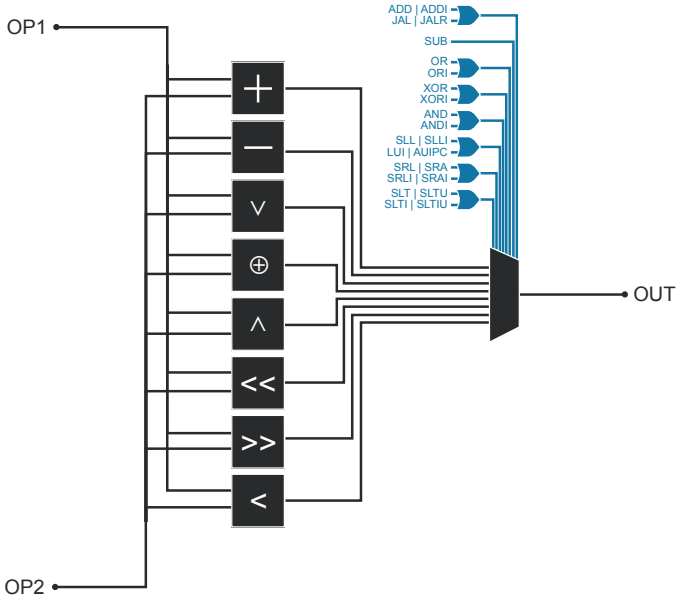

Figure 23: Internal layout of the arithmetic logic unit of the proposed RV32I CPU architecture of Apate. Operations from top to bottom: Addition, subtraction, bitwise and, bitwise xor, bitwise or, shift left by OP2, shift right by OP2, set one if OP1 is less than OP2. All operations are performed combinatorial at the same time. The necessary result is selected by the multiplexer controlled by the instruction signals (blue).

4.4.6 Branch Evaluator

The branch evaluator subcomponent uses two inputs and an instruction and performs different comparison operations depending on the instruction. The RV32I instruction set defines four comparison operations which always compare the register source 1 and register source 2. These instructions are: branch equals `beq`, branch not equals `bne`, branch less than `blt`, branch greater equal `bge` and respective unsigned versions of `bge` and `blt` which extend the result

with zeroes. The branch evaluator in the proposed architecture outputs either zero or one if the comparison is fulfilled. The output of the branch evaluator is later used to either perform a jump by adding the immediate to the current program counter or not.

The design of the branch evaluator is shown (simplified) in figure 24. The comparison elements are shown as simple blocks. This layout does not reflect the actual hardware design and is simplified to be understood more easily. Figure 25 shows an example of a more realistic approach using two comparator logic blocks and the *funct3* parameter.

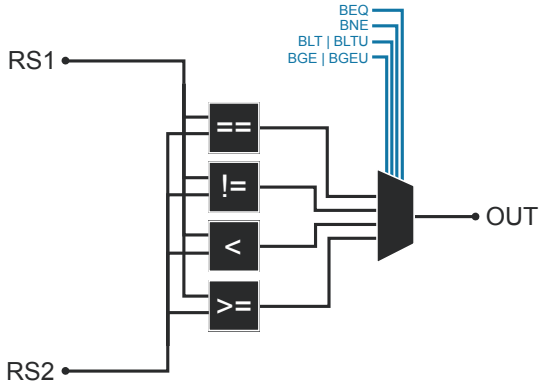

Figure 24: Simplified form of the branch evaluator as shown in Apate. The underlying logic of the comparison is not shown.

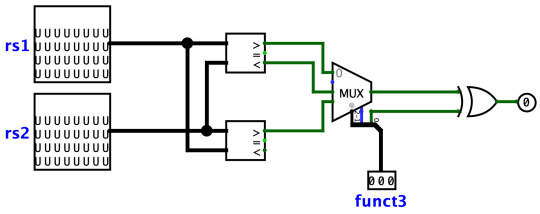

Figure 25: Branch evaluator example using two comparators from a RISC-V CPU designed in LogiSim [21].

5 Apate‘s Interface

The developed simulation tool, called *Apate*, consists of three main screens: the welcome screen, compilation screen and simulation screen. This chapter describes all three screens in detail.

5.1 Welcome Screen

In the welcome screen (see figure 26) the user can select one of three different options. If the user wants to compile his or her own C files, he or she can use the left most selection to create a new project or load an existing one.

To compile source files first the compiler needs to be set or downloaded. A message will tell the user if the compiler has not been set yet and links to the settings. In the settings the

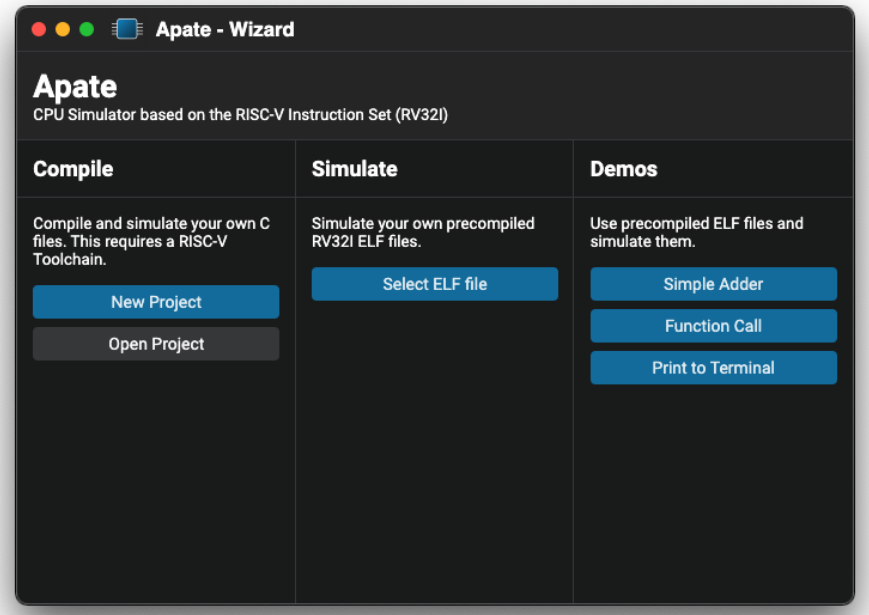

Figure 26: The welcome screen of *Apate* with three options.

compiler path can be set. It also offers the option to download a precompiled RISC-V toolchain for Linux, macOS and Windows.

A new project is filled with a basic *main.c, riscv.ld, sim.h, start.s* and *config.json*. The *main.c* only contains a basic main function returning 0. Defines are part of the *sim.h* and define addresses like the memory location to which shall be written to when printing to the program output tab. Also, a basic `void print(char *)` function is provided. *start.s* defines the program's entry point and sets the stack pointer to the highest memory address which *Apate* supports. It then calls the `int main()` function of the *main.c*. For linking a basic *riscv.ld* linker script is provided. It only provides basic functionalities and therefore does not support the standard library. Setting the source files and gcc flags can be done in the *config.json*, which provides one field for defining source files and one for providing gcc flags. The default gcc flags are *-O0 -march=rv32i -mabi=ilp32 -Triscv.ld -lgcc -nostdlib -o main.elf -g* which disable optimization (*-O0*), compile for the RV32I ISA (*-march=rv32i -mabi=ilp32*), use the provided linker script (*-Triscv.ld -lgcc*), disable the standard library (*-nostdlib*), output an ELF file (*-o main.elf*), and include debug information (*-g*) which can be used to display the source C code when using *objdump*. An already existing folder containing a *config.json* and the necessary source files can be loaded as a project.

Already compiled ELF files can also be loaded from the welcome screen. These must be valid RV32I ELF files with the correct stack pointer set. Loading ELF files can be used when wanting to simulate a project from another person or providing specific examples to students.

Demo projects with sources and already compiled ELF files can be loaded if the user is using

the simulator for the first time. These provide some basic demos including addition, function calls and printing characters to the built-in program output tab. These will be emphasized when first starting *Apate* and can be dismissed if the user does not want to load an example.

5.2 Compilation Screen

If users see the need for editing or writing custom source files in assembly or C, they can use the compilation screen shown in figure 27. This part of the interface offers a project files overview and a text editor for editing text documents which include source files, linker scripts, the *config.json* and more. By selecting one of the files in the file explorer it will be shown in the editor. Changes will be saved automatically.

Once the source files have been edited and new files have been added to the sources field in the *config.json*, the project can be compiled. The compile button will call *riscv64-unknown-elf-gcc* with the given compiler flags to generate an ELF file. The compiler output, including potential errors, is shown in the terminal. The generated ELF file will be shown in the file explorer and can be opened as a hex dump, the output of *objdump* or the output of *readelf*. The default flags for *objdump* and *readelf* can be changed in the settings. The default flags for *objdump* are *–section .text.init –section .text –section .data –full-contents –disassemble –syms –source -z* to only view the instruction and data section disassembled with inlined source lines. The default flags for *readelf* are *-a* which displays all ELF information.

5.3 Simulation Screen

Once a valid ELF is compiled, selected in the welcome screen or inside the project folder it can be used in the simulation shown in figure 28. The simulation consists of three areas. Left the instructions are displayed. These are parsed directly from the ELF file and are not used by the simulator. Their main purpose is to help users see all instructions and which one is currently executed by the CPU. The current instruction is marked blue and corresponds to the current program counter. Each instruction contains additional information including the long name, formula, description, and the values of rs1, rs2, rd and imm. Sections and symbols representing functions are also shown in the list of instructions as separators.

On the right some values of components of the CPU can be viewed. The register tab shows all 32 registers of the CPU with address, name, and value. The value can be viewed as decimal, hexadecimal, or ASCII. If the register is used by the instruction as part of rs1, rs2 or rd the name of the register is marked accordingly.

The memory tab shows the entire internal memory. Its values and addresses can be selected to be represented as decimal or hexadecimal. An ASCII representation of the value is always shown. If the memory is used by the current stage the corresponding address is marked. The

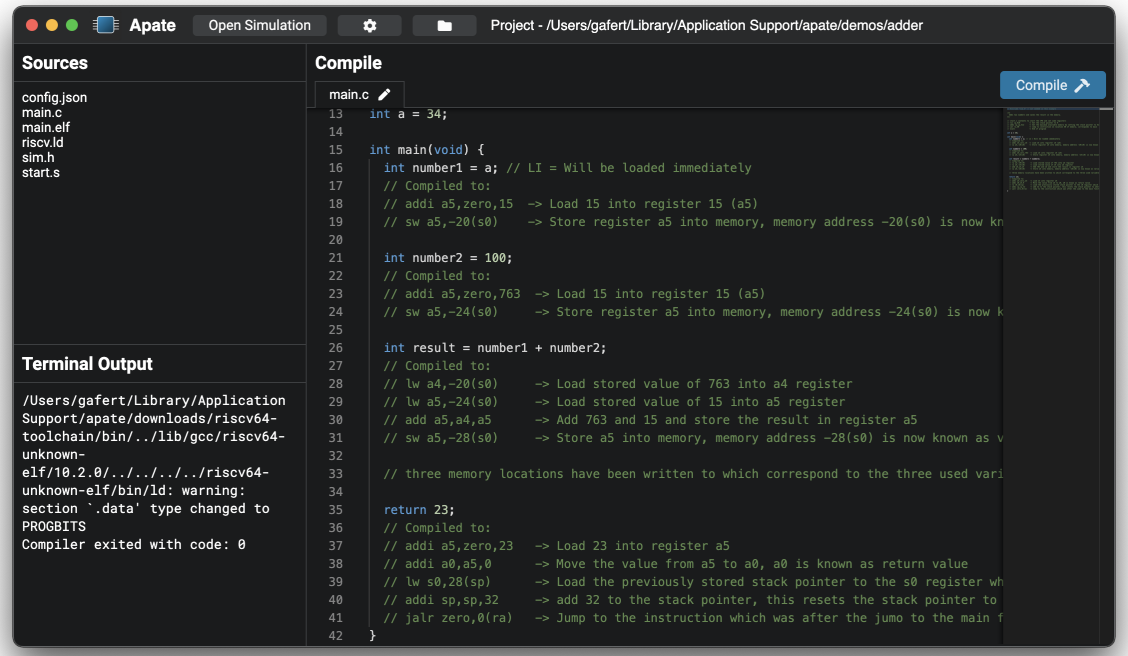

Figure 27: The compilation screen of *Apate* with the folder structure on the left and the text editor on the right.

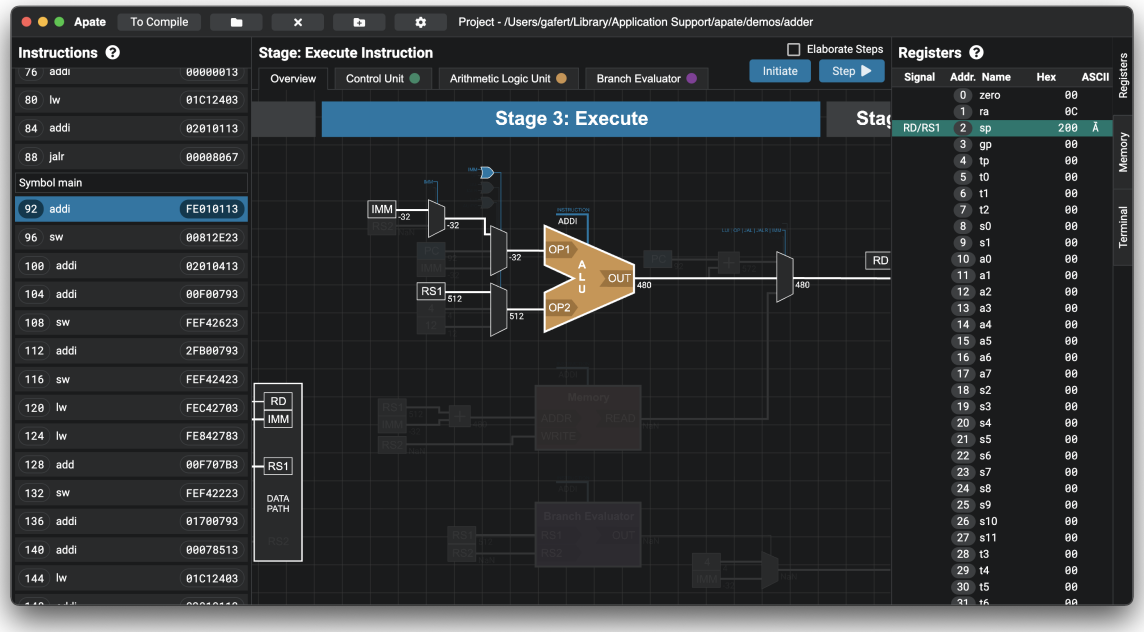

Figure 28: The simulation screen. Highlighting the currently used wires and components depending on the instruction. The currently used instruction is marked on the left in blue. The center shows the CPU architecture. On the right the registers are shown. (View the document in *Adobe Acrobat Reader* to see animations)

currently used address and store or load instruction are also shown at the top of the memory tab.

The program output tab shows the serial output of the loaded program. When writing a character to memory location *0d23456* the data will not be written to the memory but instead printed to this terminal. The memory address can be accessed with the `#define SERIAL (*((volatile uint32_t *) 23456))` definition in *sim.h*.

In the center of the simulation screen the CPU graph is shown. It consists of multiple tabs representing different components of the CPU. The overview represents the instruction cycle discussed in section 4.3. The control unit, ALU and branch evaluator contain logic which cannot be shown in this overview. When clicking these components their internal logic is shown by switching to one of the other tabs. The memory and register components are only represented as values as the real logic of a memory is too large and complicated to show. Values of these components can be viewed in the right area mentioned earlier. Each of the components of the graph can also be hovered over with the mouse to reveal more information about the component.

The graph is often simplified in terms of real CPU behavior to allow a more understandable representation. Typically, the instruction name is often represented as one hot encoding and not a string. Instructions are often grouped together in more complex ways than depicted here. Also, the logic is typically more complex to allow less stages and therefore faster CPUs. However, even if the architecture is simplified all necessary logic is represented in the graph (except memory and register logic which would be too big to show).

5.3.1 Signal wires connecting components

The components are connected via signal wires. There are control wires marked blue and data wires marked white. Both can hold one or multiple bits which can be viewed as a bus wire. If a wire holds multiple bits the bit length is shown above the respective wire. Most wires also show their current value as decimal or a string. The current value and name of the wire can also be seen when hovering directly over it.

5.3.2 Multiplexers selecting signal wires

As not all wires or components are needed for every instruction a choice must be made. Multiplexers (MUXs) can select between different input data and forward the result to the connected output. A multiplexer has multiple data inputs and one data output. Depending on the blue control signals one of the inputs is forwarded to the output. The truth table is shown when hovering over the multiplexer but can also be deducted from the control wires going into the multiplexer. The topmost control signal will forward the topmost input signal when activated. Most of the time the signal controlling the multiplexer is one of the single bit instruction type signals. Sometimes a binary representation of one signal is used to either forward the first or

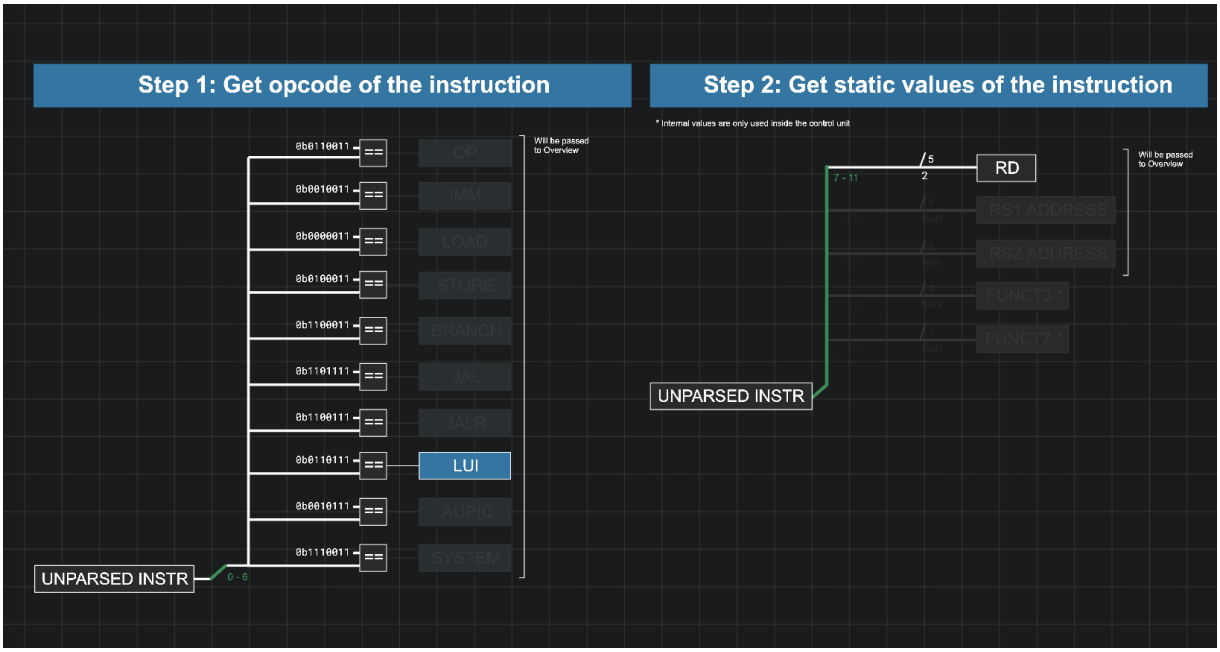

Figure 29: Inner logic of step 1 and step 2 of the control unit as shown in *Apate* with highlighted elements which are activated once the opcode is parsed

the second input signal. The respective behavior will always be described in the information text and truth table shown when hovering over the multiplexer. Examples for multiplexers are shown in later figures.

5.3.3 Selective component and signal highlighting

Depending on the currently executed instruction not all components and wires are used as their results are not selected by the multiplexers. In this case not used components are grayed out in the graph so that the user can focus on the important components which are currently used. Figure 28 shows the grayed out wires and components in the execution stage.

5.3.4 Graph subcomponent: Control Unit

The control unit component has one 32-bit binary instruction input and several outputs. If the component is clicked in the overview its internal logic will be shown. The logic consists of four areas which can be seen in figures 20, 21 and 22 in detail. Some logic will be hidden once the opcode is parsed in the first step as the unused logic is known by then. An example of the inner control logic in *Apate* is shown in figure 29.

5.3.5 Graph subcomponent: Branch Evaluator

The branch evaluator component has two inputs, one control signal and one output. Inputs are register source 1 and register source 2. These two values will be compared by the following functions, which are selected by the respective branch instruction: Equal, not equal, less than, greater equal. Each comparison is executed in parallel, and the output is selected by the multiplexer. The output of the branch evaluator is either one or zero. The result of the branch evaluator is used further along in the CPU to select between the correct value (4 or immediate value) which will be added to the program counter to enable a jump instruction which is needed when branching to another instruction. Figure 30 shows the branch evaluator in *Apate* with highlighted elements.

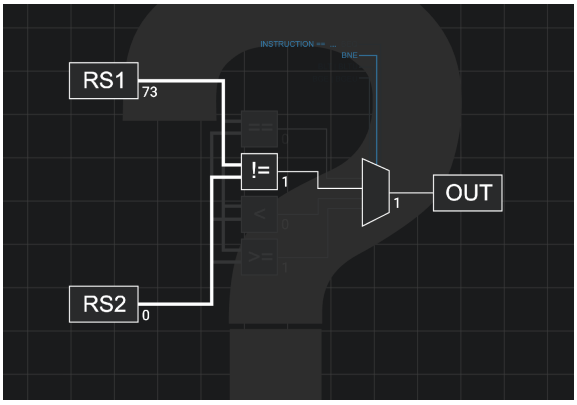

Figure 30: Inner logic of the branch evaluator as shown in *Apate* with highlighted elements depending on the current `lui` instruction

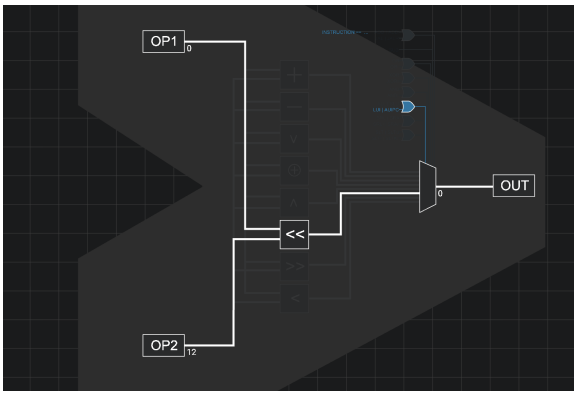

Figure 31: Inner logic of the arithmetic logic unit as shown in *Apate* with highlighted elements depending on the current `bne` instruction

5.3.6 Graph subcomponent: Arithmetic Logic Unit

The ALU has two inputs, one control signal and one output. The two inputs are called operators. These were selected depending on the instruction from either register source 1 or 2, immediate value, 4 or 12. The operand for the operators is set by the control signal which is the instruction name. Operands include addition, subtraction, bitwise and, bitwise xor, bitwise or, shift left by OP2, shift right by OP2, set one if OP1 is less than OP2. Figure 23 shows the schematic for the ALU and figure 31 shows the schematic as part of *Apate* with highlighted elements.

5.4 Guided simulation

The main feature differentiating *Apate* from other simulators is the guided simulation. Each step of the simulation is described, and additional information is linked. The guided information depends on the current instruction and the state of the CPU. The information is shown in a

popup in the right corner of the simulation. The guided steps automatically zoom on the currently described component and wires. The internal logic tabs are also automatically opened and closed.

Example of step-by-step information

The following items show an example for the information shown when executing an `addi` instruction starting at the *FETCH* stage. The bold text describes what elements are marked, focused and what area of the graph the user is zoomed into. The other text is the information the user is provided with within the popup. The list is advanced once the user presses the step button. Only the information provided by the first few steps is provided in this example.

1. **The overview is shown at the beginning.**
2. **Zooming on the fetch stage.**
   You reached the fetch stage. The next instruction is read from the memory.
3. **The program counter signal is marked.**
   Pass the current program counter (PC) as the address to the memory. You can see what the memory does in the memory tab on the right.
4. **The memory subcomponent is marked.**
   Loaded a new instruction from the memory at the address of the program counter (PC).
5. **Zooming on the fetch and decode stage. The instruction data signal going from the memory to the control unit marked.**
   Pass the unparsed 32 bit long instruction data into the control unit to be decoded.
6. **Zooming on the decode stage.**
   You reached the decode stage. The unparsed instruction will be parsed, and all the signals will be filled to control the CPU. For now, the signals are unknown as they are not parsed yet.
7. **Going into the control unit component.**
   In the control unit the instruction is decoded into different signals used throughout the CPU. These values include...

6 Implementation

This chapter describes the implementation of *Apate*. First, the user interface implementation, shown in the previous chapter, is described in section 6.1. This section also informs about other user interface implementation options and the reasons why these have not been pursued. The needs for the underlying CPU simulator implementation are stated in section 6.2. The different implementation options and pitfalls of using hardware design language designed CPUs are discussed in section 6.2.1. The used JavaScript based CPU implementation is described in section 6.2.2. Finally, other components used in the program, including the ELF parser and the instruction decoder, are explained.

6.1 User Interface Implementation

The user interface of a simulator typically contains two primary elements. A graphical visualization displaying a map of a CPU architecture and a surrounding user interface containing lists, text and images. This requires a multipurpose engine or framework to display both interface types. An assessment of multiple game engines and frameworks has been made. This includes Unity [29], PyGame [30], PyQT [31], libGDX [32], Electron [33], JavaFX [34], and .NET [35] based applications. The three requirements are listed below.

1. **Cross Platform**
   With the requirement of the simulator to work on every major operating system independent of previously installed tools none could be eliminated as all provided a method for providing executables.

2. **Advanced UI Development**
   The next requirement to easily build responsive user interfaces eliminated multiple contestants. Although Unity provides UI layout methods they are not as easy to implement as .NET or web-based interfaces. This also applies to PyGame and libGDX.

3. **Performant Rendering**
   With the need to also embed a performant map like visualization at the heart of the simulation a web-based approach with Electron was chosen. With web-based applications a multitude of matured user interface frameworks can be chosen (Angular, React, Vue, Svelte, etc.) and with the help of webGL all visualizations previously only possible in game engines can also be pursued in web-based applications. Many applications are utilizing the web's capabilities on desktop applications including WhatsApp Desktop, Visual Studio Code, Atom, Discord and many more [33]. JavaFX and PyQT also do not hold up in the regard of rendering anything different than generic user interfaces from the author's perspective and experience.

As the whole simulator is an Electron based application the user interface can be developed with any web-based framework. For Apate, Angular 10 was chosen [36]. Angular allows for dynamic interfaces developed with HTML based template layouts and corresponding TypeScript modules. Updates of variables are directly forwarded to the corresponding HTML element and updated nearly instantaneous. To allow a fast development cycle the interface was first designed in Adobe XD[1]. Once the concept design was completed the layout was developed in Angular.

For the CPU architecture map, Three was used [37]. Three is a JavaScript library which utilizes the built-in webGL renderer of browsers. Three allows for advanced 2D and 3D visualizations which are independent of the Angular based interface. There are multiple ways the CPU architecture can be visualized. A common practice is the use of SVG graphics embedded into the web page. With the increasing number of SVG elements this can lead to slow and sluggish interactions with the vector graphic, however. Another method for drawing elements on a web page utilizes packages like p5.js which use the HTML5 canvas [38]. A canvas element can be drawn onto directly by simple JavaScript code. However, the better version is webGL, as it is more performant than canvas-based rendering when dealing with a large number of objects. The CPU architecture itself was designed in Adobe XD and exported as a scalable vector graphic (SVG). The SVG document and its elements are parsed by an SVG loader into simple paths and text elements. This SVG loader is part of the Three examples and is slightly modified to be able to parse additional SVG elements like `<text>` and `<tspan>`. The resulting paths are reconstructed with 3D meshes so they could be displayed by Three on the webGL renderer. These meshes correspond to the given SVG element and are either rectangles, lines, or splines. The text embedded in the SVG document with the tag `<text>Example</text>` was rendered using multi-channel signed distance fields to allow for sharper text [39]. These text elements scale better than generic canvas based texts and prevent the text from looking pixelated.

The underlying CPU code only interacts with the user interface and CPU architecture map over a list of variables in an external file. The CPU updates the variables, and the interface reacts to these updates. The interaction is shown in figure 32.

[1] Adobe XD is a vector graphics based user interface design development program by Adobe.

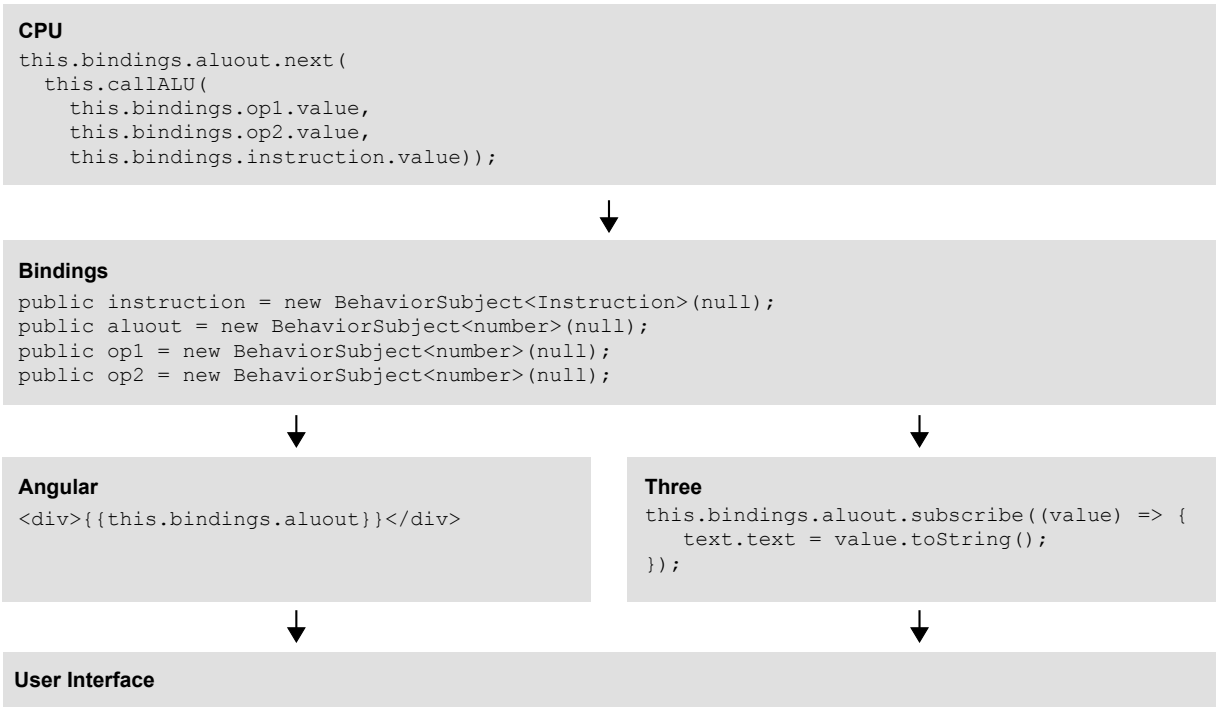

Figure 32: The interaction between CPU and the user interface. The CPU only updates values in a separate bindings file. Angular templates and Three subscribe to these values to update the corresponding elements of the user interface.

6.2 Central Processing Unit Implementation

The CPU of Apate is developed as a TypeScript module. TypeScript was chosen for a multitude of reasons which are discussed in this section. When first thinking of a possible CPU implementation the architecture needs to be defined. For this simulator RISC-V was chosen. There are three possible CPU implementations.

- Using existing "No GUI" backend simulators and build a frontend for them
- Using a CPU designed in Hardware Design Language and compiling it to a shared library
- Writing a custom CPU simulator implementation

There are already a few RISC-V "No GUI" backend simulators which simulate a full CPU. These could simply be incorporated into a frontend to visualize the CPU. However, most of these simulators only output a final file containing the result of an output register. The internal architecture is hidden from the user. As Apate wants to show the simulation in real time step-by-step these "No GUI" simulators cannot be utilized.

Another possible solution is using a CPU core designed in a Hardware Design Language[2] (HDL) and using tools to create an executable version of the designed CPU. This allows for a

[2]Hardware Design Languages are languages like VHDL and Verilog. With the help of special tools these designs

"real" CPU implementation as the resulting executable is an exact description of the logic of a real CPU. This solution is described briefly in section 6.2.1, but was ultimately not chosen as the possible CPU implementations do not focus on a simple architecture design, but on either speed or size which is the general goal of an HDL core.

The last solution is to write a custom CPU simulator. This is the most flexible solution as the whole architecture is custom and can be designed for the best educational experience. As Apate uses JavaScript and TypeScript, the CPU was also written as a TypeScript module. The incorporation of the CPU into the simulator is described above.

6.2.1 Synthesized HDL CPU Implementation

This chapter describes a possible solution for having a HDL CPU in the simulator. It was analyzed over a few months but ultimately disregarded. However, as it was a necessary step in the development of Apate, the process of utilizing an HDL core in a JavaScript based environment will be described here.

To have a “real” CPU it would be advisable to not depend on simulations, but on real hardware designs written in a so-called Hardware Design Language (HDL). These IP cores are readily available on the web, see GitHub [40], OpenCores [41], or other code sharing sites. The HDL design should fit the following requirements.

- Simple and well-defined ISA
- An assembler or a compiler for the defined ISA
- No or small periphery
- Soft IP Cores, meaning no FPGA or tool specific code
- Designed in Verilog or VHDL

To compile these HDL designs tools like GHDL [42] or Verilator [43] can be used to create an executable and use it as a generic shared library. More information on these tools can be found in the next chapters. These tools also provide interfaces, like VHPIDIRECT, VPI or Verilators direct interface, so other languages can tap into the HDL design’s signals and registers.

Typically, HDL designs are synthesized into netlists and further into FPGA gate logic descriptions. These designs describe the CPU’s internal structure and are no code by themselves. The difference between GHDL and Verilator and other HDL simulators like ModelSim [44] is that GHDL and Verilator generate native machine code from the design and not a netlist. This native code running the design executes much faster than a simulated netlist. Timing- and routing delays are however, not accounted for. As both timing- and routing delays are not crucial in this specific use case as a CPU simulator, both tools provide the necessary requirements.

---

can be synthesized to silicon circuits or FPGA (Field Programmable Gate Array) logic definitions. These design languages are primarily used to design real hardware.

VHDL with GHDL

GHDL [42] is a command line tool for analyzing, compiling, and executing VHDL code. As it is not a testing framework, a so-called testbench, which is the starter file in this example, needs to be written in VHDL. With the use of llvm or gcc, GHDL can generate an executable of the VHDL Code. It uses C and C++ as an intermediary language to compile directly to machine code. The executable can be run without GHDL installed and is much faster than a simulated netlist. The typical workflow for using GHDL can be seen in source code 3 where a VHDL file (code.vhd) is compiled into an executable called *code_tb*. To incorporate GHDL with other languages it provides two interfaces: VHPIDIRECT and VPI.

```
1  ghdl -a --work=work code.vhd // Analyze the design
2  ghdl -e --work=work code_tb // Elaborate the top entity
3  ./code_tb // Run like any other program
```

Sourcecode 3: Building and running a VHDL design.

VHPIDIRECT

VHPIDIRECT provides a way to link C code with the compiled VHDL code. C functions can be called directly from VHDL and pass simple data types. More complex data types like arrays are passed as a fat pointer. As the "fat pointer"[3] is poorly documented it is hard to understand what exactly is passed. Callbacks and signal changes are not supported. For these features, GHDL recommends the other interface called VPI which is described in the next section but has even less documentation.

Example of using VHPIDIRECT

A short example uses the neo430 [45], a msp420 based CPU by Stephan Nolting. For demonstration purposes a hard coded application memory, already containing a simple program, is used. To enable VHPIDIRECT, external procedures need to be defined in VHDL, shown in source code 4. There are two procedures (functions) called `read_reg` and `write_reg`. These VHDL procedures are called by the neo430 design whenever a register is read from or written to. The procedures are defined as `VHPIDIRECT read_reg` and `VHPIDIRECT write_reg`. When compiled with GHDL, these will be linked to C functions with the respective name. The C functions are part of source code 5, where a print function prints the accessed register's current value. Each time the neo430 reads or writes from its internal register the VHDL procedure will be called which will then call the C function and print the register. To combine the neo430, the binding procedures and the C functions, first the VHDL code needs to be elaborated. Then the C file needs to be compiled and finally

[3] A "fat pointer" is a pointer which not only contains an address to memory but also additional information.

the compiled C file and the VHDL executable named *neo430_tb* need to be linked with GHDL. The *neo430_tb* executable can then be executed to show the accessed registers.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package bindings is
  procedure read_reg(name: integer; value: integer);
    attribute foreign of read_reg :
      procedure is "VHPIDIRECT read_reg";
  procedure write_reg(name: integer; value: integer);
    attribute foreign of write_reg :
      procedure is "VHPIDIRECT write_reg";
end bindings;

package body bindings is
  procedure read_reg(name: integer; value: integer) is
  begin
    assert false report "VHPI" severity failure;
  end read_reg;
  procedure write_reg(name: integer; value: integer) is
  begin
    assert false report "VHPI" severity failure;
  end write_reg;
end bindings;
```

Sourcecode 4: neo430_c_bindings.vhd

```
#include <stdio.h>
#include "ghdl.h"

extern int ghdl_main (int argc, char **argv);

int main(int argc, char const *argv[]) {
        printf("Started execution\n");
        const char*  args[3] = {"--stop-time=20ms", "--ieee-asserts=disable",
        ↪  "--assert-level=error"};
        ghdl_main(3,args);
        return 0;
}

void write_reg(int r, int v) {
        printf("W -> R%d: %x\n", r, v);
}

void read_reg(int r, int v) {
        printf("R -> R%d: %x\n", r, v);
}
```

Sourcecode 5: vhpidirect.c

```
ghdl -a --work=neo430 $srcdir_core/neo430_c_bindings.vhd
ghdl -a --work=neo430 $srcdir_core/neo430_package.vhd
ghdl -a --work=neo430 $srcdir_core/[...].vhd

gcc -g -c vhpidirect.c -o vhpidirect.o
ghdl -e --work=neo430 -Wl,vhpidirect.o neo430_tb // Link the C code with VHDL

./neo430_tb // Call like any other program
[...]
W -> R0: c
R -> R0: c
R -> R8: c000
R -> R1: 800
R -> R1: 800
W -> R1: c800
R -> R1: c800
R -> R0: c
[...]
```

Sourcecode 6: Building VHDL with VHPIDIRECT C program

This example shows that it is possible to compile a VHDL design and access its values from a C executable. However, this requires setting up callbacks for each signal, which requires a deep knowledge of the neo430 and its signals. Another possible HDL core is the lxp32 [46]. As the lxp32 has a custom ISA there is no compiler available, this means that a requirement of the simulator for writing custom C code cannot be fulfilled. For these reasons the VHPIDIRECT interface was not used.

VPI (Verilog Procedural Interface) / PLI 2.0

GHDL, even though it is a VHDL compiler, also includes VPI, formally known as PLI 2.0, a Verilog specific interface. However, even though callbacks and signal assignments should be possible, there are no simple demonstrations provided by GHDL. As only VPI and VHPIDIRECT can be used with VHDL designs, the search for VHDL solutions was discontinued.

Verilog with Verilator

Verilator is GHDL's counterpart for Verilog designs. Verilator compiles Verilog designs to intermediary C code, called "verilated" code. This code can be used as a base for the testbench which can be entirely written in C. Every wire and register can be accessed directly via pointers `top -> top__DOT__module__DOT__register`. This provides a fast and easy way to access every part of the Verilog design. The generated intermediary code can be extended and then compiled with gcc to an executable or shared library to be used by other programs. Verilator also provides the VPI interface but recommends the direct access via pointers mentioned above. The following example shows the usage of Verilator.

Example of using verilated code

First the Verilog design needs to be verilated into the intermediary C code representation. Verilator provides a header file to allow for further integration of the design into custom C++ files. Source code 7 shows a C file where the generated header *Vpico_testbench.h* is used. The used HDL core is the PicoRV32 [28]. Once the design is verilated, it can then be used like any other C++ class. `top -> clk = !top -> clk;` combined with `top -> eval();` simulates a clock cycle.

```
1   #include "Vpico_testbench.h" // Use the generated header
2   #include "verilated.h"
3
4   void main(void) {
5       Verilated::commandArgs(0, (char **) "");
6       top = new Vpico_testbench; // The top entity is called pico_testbench
7       top->clk = 0;
8
9       while (!Verilated::gotFinish()) {
10          top->clk = !top->clk; // Toggle the clock signal
11          top->eval(); // Evaluation executes the design
12      }
13  }
```

Sourcecode 7: Initiating the testbench for the Vpico_testbench top module in C.

Incoperating verilated designs into JavaScript

The verilated code can be extended by functions which return the current value of signals in the design. When compiling the verilated and extended code to a shared library, the library containing the CPU can be accessed by other programs with foreign function interfaces. Electron utilizes the Node runtime which can load dynamic libraries with a package called *ffi-napi*. Functions of the shared library can then be called like any other JavaScript functions. The loading of the shared library is shown in source code 8. A header file and the library loading can be automated with a python script exposing all signals and registers of the CPU core to the JavaScript application.

However, this means that all functions returning the values must be called after every clock cycle to allow for updates in the application. This results in heavy CPU usage. Furthermore, the selected PicoRV32 CPU design is optimized for size and not simple architecture. These reasons result in problems when trying to visualize this CPU and were the reason this solution was not further followed.

6.2.2 JavaScript CPU Implementation

With the goal of Apate to visualize the internal architecture of a CPU, the implementation itself needs to be aligned with the visualization. This section describes the JavaScript / TypeScript

```
import {Library} from 'ffi-napi';

// .dylib on macOS is equivalent to .so or .dll
let simLib = new Library('libVpico_testbench.dylib', {
  'advanceSimulation': ['void', []], // Expose the functions of the library
  'startSimulation': ['void', []],
  'getPC': ['int', []]
});

simLib.startSimulation(); // Call like any other function
```

Sourcecode 8: Loading a dynamic shared library into Node with ffi-napi.

based implementation of designed CPU architecture.

The CPU was structured into 5 stages as described in section 4.3. The stages are controlled by a state machine. Most logic of typical HDL CPU is combinatorial logic, including ALU, branch evaluations and simple additions. Although these cannot be handled combinatorially in JavaScript, because of JavaScript's inherent sequential nature, they are regarded as not state machine relevant and called regardless of the current state. This allows the visualization to show results of other signals which are not in the current stage. Other logic which should only be called when necessary in a specific stage, include memory read, instruction decoding, memory write, register write and advancing the program counter are part of the state machine. The state machine is advanced by the `advanceSimulationClock()` function which will be called once the user reaches the next stage in the interface.

The state of the CPU is part of the aforementioned *bindingSubjects* service. This allows for every other element of the simulator to get updates once the CPU changes signal (variable) values. Memory and registers are simple arrays stored in the *bindingSubjects* service. These will be accessed by the CPU. There is no external memory module outside the CPU. The memory will be filled once the `init(pathToElf: string)` function is called, which loads an ELF file with the `parseElf(elf: Buffer)` function provided by the ELF parser which is described in the next section 6.3. Once the ELF is loaded into a dedicated object the program section is loaded into the memory.

When the program is executed the first four bytes will be loaded from the memory into a new variable. These bytes will then be parsed by a `parseInstruction(bytes): Instruction` function which returns an object of type *Instruction*. This function is discussed in section 6.4. The *Instruction* object contains all values a decoding unit may decode in the process. The object will be separated and stored in several variables of the *bindingSubjects* service. Once the variables are updated the user interface and graph will update accordingly. The same procedure also applies for the other stages where variables are updated.

6.3 ELF Parser

The ELF parser provides a single function called `parseELF(elf: Buffer) : ELF` returning an object of type *ELF*. The passed value is the raw bytes of the ELF file. The returned object contains a parsed version of the ELF file, including ELF headers and the program section. Figure 33 shows an overview of an ELF file and its format. An ELF heavily utilizes links to where the actual data is stored. Offsets can be read from the header which lead to another location of the ELF where another offset to the data can be read. For all required data to be parsed one must first gather all required offsets. The offsets of values within a section, for example the header, are defined by the ELF standard. All information about the ELF can be found in the ELF(5) man page [47]. During the development the online tool *elfy.io* [48] and the command line tool *readelf* were used to display the contents of the file. An example output of *readelf* is shown in source code 9.

The parsing of an ELF file will be described in the following points. A *parsedElf* object will be filled with data from an ELF file. To read the data the function `read2BytesLittleEndian(elf, address): number` is used which uses the elf file and the address of the required data within the ELF file. This function reads two bytes starting at the value of parameter address. Another function which reads four bytes is also used. The address is typically a combination of offsets.

- To identify that the raw binary data is indeed an ELF file four numbers can be read at the beginning of the header: 0x7F, 0x45, 0x4c, 0x46. These translate in ASCII to ".ELF".
- To identify the target instruction set architecture of the ELF file the *e_machine* field can be read from the header. This should read as 0xF3 for RISC-V executables. The corresponding code is `parsedElf.e_machine = read2BytesLittleEndian(elf, 0x12);`.
- To get the program bytes the program offset *p_offset*, and the program size *p_memsz* can be read from the program header table. The program header table itself is also offset in the ELF by *e_phoff*. To read the program bytes first the program header table offset must be read with `parsedElf.e_phoff = read4BytesLittleEndian(elf, 0x1c);` which can then be used to read *p_offset* with `parsedElf.p_offset = read4BytesLittleEndian(elf, parsedElf.e_phoff + 0x04);` and *p_memsz* with `parsedElf.p_memsz = read4BytesLittleEndian(elf, parsedElf.e_phoff + 0x14);`. To get the program the bytes can be sliced from the ELF file with `parsedElf.program = elf.slice(parsedElf.p_offset, parsedElf.p_offset + parsedElf.p_memsz);`. This also includes the data section used by the program.
- The program can now be loaded into the CPU's memory. However, there is more information included in an ELF file apart from the program itself. This includes debug information and the names of functions. These values are encoded into sections. First the section header offset *e_shoff* (`parsedElf.e_shoff = read4BytesLittleEndian(elf,`

`0x20);`) and the number of sections *e_shnum* must be parsed. With the section header offset and the number of sections the section header table can be iterated by the number of sections to get the name offset, type, size and offset of the sections. To get the name string the name offset has to be used with `section_header.name = readString(elf, sectionHeaderNames.sh_offset + section_header.sh_name);`

- To get the name of functions and sections the string table section *.strtab* can be used. The string table contains a list of strings for section names and symbols. A section can contain multiple symbols. There are for example multiple functions in the program. Therefore, there are also multiple symbols naming each function. The number of symbols in a section can be calculated by the size of the section *sh_size* by the size of one entry (symbol) *sh_entsize*. Each name of the symbol is again represented as an offset *st_name*. The offset is relative to the *.strtab* section. The name of a symbol can therefore be parsed with `elfSymbol.name = readString(elf, parsedElf.section_headers[stringTableSectionId].sh_offset + elfSymbol.st_name);` where the variable *stringTableSectionId* is the index of the section named *.strtab* and *elfSymbol* being one of the symbols of a section with a name.

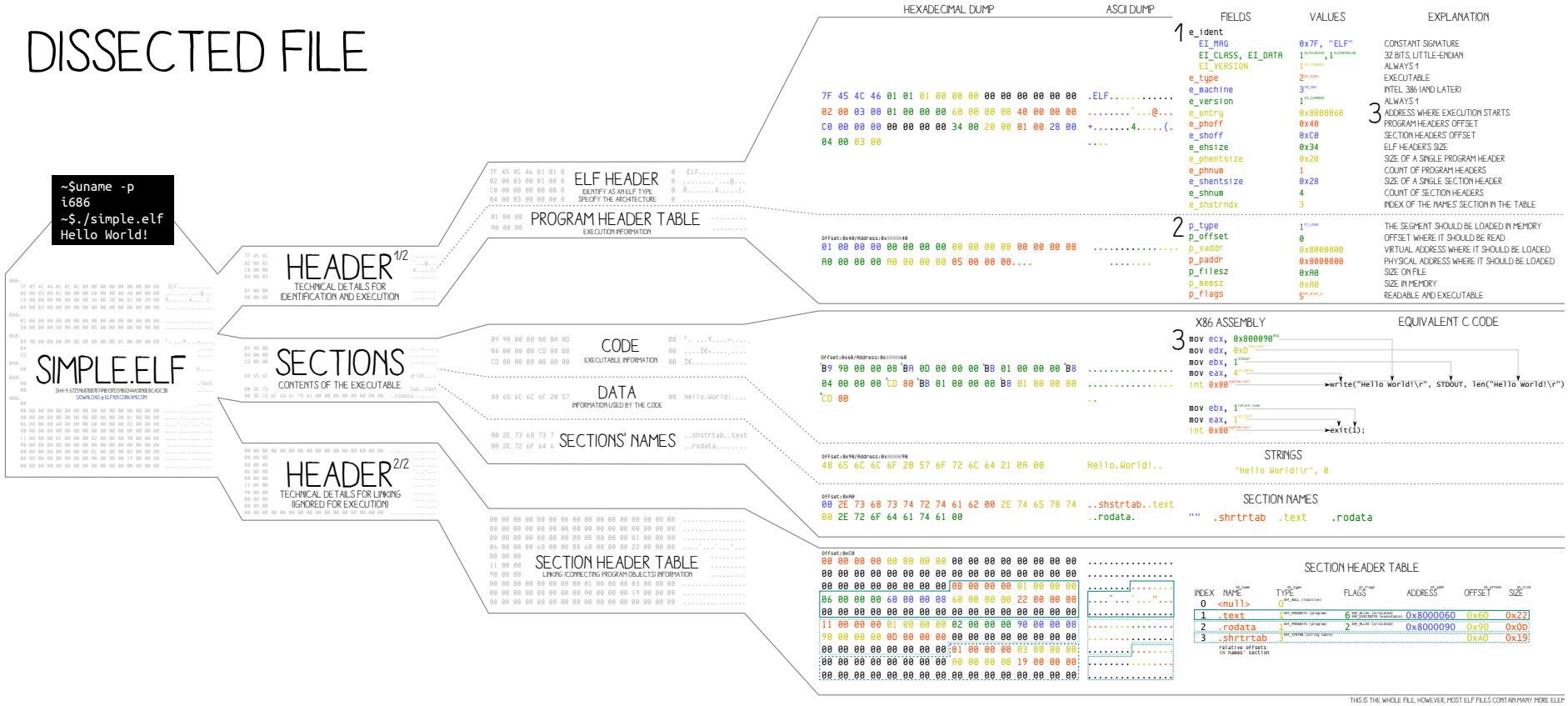

Figure 33: A simple depiction of the ELF format showing the different sections by Ange Albertini titled: ELF 101 a Linux executable walkthrough.

6.4 Instruction Decoder

The instruction decoding is provided by an additional `parseInstruction(instruction, addr = 0): Instruction` function which returns an object of type *Instruction*. It is called by the CPU once in the decoding stage and once when the ELF is first loaded to show the instructions in the instruction panel of the UI. Although the CPU only needs the instruction name, registers and the opcode, the *Instruction* object also contains additional information such as the raw bytes of the instruction, the instruction type (R, I, S, B, U, J), the corresponding instruction format (opcode) (*OP, OP-IMM, BRANCH, LOAD, STORE, SYSTEM, JAL, JALR, AUIPC, LUI*), an instruction description including the instruction short name (`addi`, `add`, `lui`, etc.), the long name (Add Immediate), a textual description, an equation (rd = rs1 + imm), possible immediate values, the correct immediate values depending on the instruction format, the destination register, register source 1 and register source 2. The assembly is also provided and represented like the output of *objdump*. However, this requires an additional second parameter, the instruction's address in memory, when calling `parseInstruction(instruction, addr = 0): Instruction` to correctly display jumps. Most values are stored by the CPU in the *bindingsSubject* service where they are used throughout the visualization.

7 Evaluation

Every newly created program or system requires an evaluation. This chapter describes the measures taken in the terms of unit tests used during development to evaluate the implemented CPU described in the previous chapter. The unit test implementation and results are described in section 7.1. Apart from the technical evaluation a primary part of evaluating a user interface are usability tests by possible users. The usability tests are described in section 7.2. The methodology, procedure and results from the test and an additional interview are described and further analyzed in both qualitative and quantitative means. Concluding the usability test a list including improvements, deducted from the usability test, is presented.

7.1 Technical Evaluation of CPU simulator with unit tests

This chapter describes the implemented unit tests for Apate. The elf parser and instruction decoder have been unit tested to prevent erroneous simulation execution.

7.1.1 Unit testing the JavaScript ELF file parser

The JavaScript based ELF parser was validated via unit tests which compared the output of the official *riscv64-unknown-elf-readelf* with the flag *-a*, which displays all information, against the output generated by JavaScript. First *riscv64-unknown-elf-readelf* was used to generate the information about the ELF. The output can be seen in source code 9. Then the JavaScript based parser was executed with the same input ELF file. With the help of regular expressions, the necessary information was gathered from *riscv64-unknown-elf-readelf*'s output and then compared against the developed parser. One of the used tests can be seen in source code 10. Test for machine type, start of section headers, start of program header, section header size, section header number, section header string table index and the contents of each section header including name, address and size was performed on three different ELF files.

```
1  ELF Header:
2    Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
3    Class:                             ELF32
4    [...]
5    Size of this header:               52 (bytes)
6    Size of program headers:           32 (bytes)
7    Number of program headers:         1
8    Size of section headers:           40 (bytes)
9    Number of section headers:         13
10   Section header string table index: 12
11
12 Section Headers:
13   [Nr] Name              Type      Addr     Off    Size   ES Flg Lk Inf Al
14   [ 0]                   NULL      00000000 000000 000000 00      0   0  0
15   [ 1] .text.init        PROGBITS  00000000 001000 000010 00  AX  0   0  1
16   [ 2] .text             PROGBITS  00000010 001010 00008e 00  AX  0   0  4
17   [ 3] .debug_line       PROGBITS  00000000 00109e 0001b1 00      0   0  1
18   [ 4] .debug_info       PROGBITS  00000000 00124f 000127 00      0   0  1
19   [ 5] .debug_abbrev     PROGBITS  00000000 001376 0000af 00      0   0  1
20   [ 6] .debug_aranges    PROGBITS  00000000 001428 000040 00      0   0  8
21   [...]
```

Sourcecode 9: Concatenated output of *riscv64-unknown-elf-readelf -a main.elf*.

7.1.2 Unit testing the instruction decoder

To test the instruction decoder a unit test was used which takes a hexadecimal representation of an instruction and the instruction's address as an optional parameter to parse an instruction. The parsed instruction object returned by the `parseInstruction()` function contains a field called *assembly* which contains the instruction as an assembly string. The resulting string is compared with a predefined string which equals the output of *objdump*. A code snippet of the corresponding unit test can be seen in source code 11.

```
const elfPath = 'main_renderer/static/demos/adder/main.elf';

it('elf section headers size', () => {
  // Get the stdout of riscv64-unknown-elf-readelf
  const readElf = execSync(`riscv64-unknown-elf-readelf -a ${elfPath}`).toString()

  // Initiate the CPUService which contains the parsing in the unit test
  const service: CPUService = TestBed.get(CPUService);
  service.init(elfPath);

  // Use regex to extract the number
  const sectionHeaderSize = Number(/(?:Size of section headers:)(?:[
  ↪  ]+)(\d+)/.exec(readElf)[1]);

  // Compare the data
  expect(service.parsedElf.e_shentsize).toEqual(sectionHeaderSize);

  console.log('Size of section headers equals to ' + sectionHeaderSize);
});
```

Sourcecode 10: Unit test comparing the size of the section headers parsed by JavaScript and *riscv64-unknown-elf-readelf*.

```
  it('jal', function() {
    // Provide address of this instruction to allow relative to absolute address
    ↪  conversion
    const instr = parseInstruction(0x0200006f, 0x20);
    expect(instr.assembly).toEqual('jal zero,40')
  });
```

Sourcecode 11: Unit test comparing the result of the JavaScript based instruction parser and a predefined string.

7.2 Usability Test

This section describes the social acceptance of the developed simulator by presenting a methodology and the results of seven usability tests with an included interview.

To gain feedback of *Apate* and evaluate the simulator seven usability tests have been conducted. Typically, these usability tests are held in person where a potential user is using a software to complete given tasks. During the usability test the participant is asked to think aloud. The goal of a usability test is to identify different types of problems the participant encounters ranging from cosmetic to severe usability problems. Due to COVID-19, such in person usability tests were not possible and instead remote synchronous tests were conducted which provide virtually the same results as conventional in person tests [49]. Additionally, to the usability tests a short 10 minute interview was held to gain quantifiable data which can be analyzed further. The results of both usability test and interview are shown below.

7.2.1 Procedure

The participants were asked to share their screen while completing the following task: Use the provided simulator, called *Apate*, to get to know the tool itself (1) and understand the basics of a RISC-V based architecture (2). There were no time constraints, but the participants were encouraged to try to get to know *Apate* by themselves without the help of the facilitator. While completing the tasks the participants were asked to think aloud. After the participants felt like they completed the tasks and understood the simulator, the session was concluded with an around 10-minute-long structured interview.

7.2.2 Qualitative results

This section describes insights which have been gathered during the usability tests. The most valued elements of the simulation are the detailed description of each instruction, the performant visualization and the description shown when hovering over the components.

Identification of the simulator's goals

One question asked the participants to describe the simulator in one sentence. This answer should prove that the participants identified the simulator's features correctly and categorize it as a graphical simulator for use in education. The participants answered the following for the question "Please describe Apate in one sentence". The answers have been translated from German.

> "RISC-V simulator with focus on simple usage and graphical visualization"
>
> "Graphical appealing tool to visualize all individual processes of a CPU"
>
> "A very performant program to visualize how a CPU works"
>
> "A very helpful tool to capture the processes of a CPU in a stepwise and clear manner"
>
> "Graphical CPU simulator for RISC-V"
>
> "Easy to use, easy to understand"
>
> "The simulator allows me to execute C programs in assembly step-by-step. I can view which registers and memory cells are written to or read from."

The answers show that the majority identified the simulator as a graphical simulator for CPU architecture. The educational effect was not part of the participants' description, although it can be deducted from the simplicity mentioned.

Usage in university courses

Furthermore, all participants were asked to imagine a scenario where they are lecturing in a university course named "CPU architecture" and if they would use Apate as a main tool or as an additional tool. A main tool would be the basis of the course. Other course material would build upon the simulator and most of the examples shown would be conducted within the simulator.

Additonal tool All participants would use Apate as an additional tool in their imagined "CPU architecture" university course.

Main tool Three of seven participants would use the simulator as a main tool in their university course. Three other participants were unsure, and one participant chose not to use Apate as a main tool as there cannot be one single tool for a topic as broad as CPU architecture. However, all participants who chose to use Apate as a main tool mentioned that it heavily depends on the exact content of the course. The students would furthermore require previous knowledge of the C language if Apate would be used.

7.2.3 Quantitative results

After the participants used Apate a short interview was conducted where the participants needed to rank the following questions shown in figure 34 from (1) knowing nothing or not approving at all to (5) knowing a lot or approving. The first question asked the participants about their current knowledge about CPU architectures. This was used to create possible regressions of later questions. Most participants categorized themselves as 4 ($\bar{x}_M\ (modus) = 4,\ \bar{x}\ (mean) = 3.43,\ \tilde{x}\ (median) = 4,\ \sigma\ (std.\ deviation) = 1.27$) which indicates a bias of the participants towards a group educated about CPU architecture. Questions 2 to 10 are general questions about the simulator. Later questions are about specific elements and if the simulator provided enough information. Specific results are described in the next chapters.

Previous knowledge effects

The majority of participants had classified themselves as knowing a bit about CPU architectures. One participant stands out who regarded him- or herself as knowing very little about CPU architecture. The results show that the effect of previous knowledge is not insignificant as this participant answered generally lower ($\bar{x} = 3.65$) than the other participants ($\bar{x} = 4.63$). This must be considered when discussing the target group for the simulator.

Conceptual information

Questions 11 to 18 asked the participants on concepts shown in the simulator and if they think they understood them. The most noticeable information missing to understand certain concepts

Figure 34: Results of the survey questions asked in the interview after using Apate. The abbreviation A.u.A. means "After using Apate". Question 1: (1) I don't know anything - (5) I know a lot. Question 2 to 18: (1) Disagree - (5) Agree completely.

are the memory ($\bar{x} = 3.86$, question 14) and second the decoding of an instruction ($\bar{x} = 4.43$, question 13) along with understanding the RISC-V architecture ($\bar{x} = 4.43$, question 11) The memory was not explicitly described in the simulator and only shown as an information shown when hovering over the heading of the memory and therefore matches the given results. The decoding of the instruction is by far the most complicated part of the architecture and still needs work to be described as simply as the ALU ($\bar{x} = 5.00$) or the registers ($\bar{x} = 5.00$).

CPU architecture

The answers provided indicate that the architecture is mostly understood but can be improved further. Further improvements are described in a later section (7.2.4). This includes the signal paths not being understood by some and the instruction decoding being too complicated.

Good impression and future use

Although, there is still desire for improvement, most participants would use Apate again ($\bar{x} = 4.86$) and had a good overall impression ($\bar{x} = 4.71$).

7.2.4 Possible improvements

Problems which arose have been noted while the test was conducted. These are the most mentioned problems the participants identified, arranged from most severe (1) to least severe (26). Some problems have already been addressed in the current version of Apate but are also mentioned here as they have been regarded as worth mentioning.

1. **Getting started document**
   A getting started document is necessary to explain the general layout of the simulator including the different tabs, the navigation, how to load an example, how to edit custom code and how the guided simulation works. This requirement has already been fulfilled after the first interview with an around 10-minute-long video explaining Apate. The video explains the interface in a walkthrough, starting with the Adder example, describing each element of the simulator briefly and finally editing and compiling a custom project.

2. **Show previous information**
   There is currently no way to display the information shown in a previous step. This is felt as difficult for the participants as the information was sometimes only skimmed over and not fully comprehended. The simulator requires the user to understand the information provided and builds upon the shown information. This requires users to go back one step. A better approach would be to go back a whole step and show the highlighted elements again additionally to the information.

3. **Complicated control path**
   The control path is currently marked blue and starts at the second stage, the decode stage, after the control unit. The control unit outputs the control signals in an element called control path. This was initially meant as a tunnel. The signals going into the control path would leave this tunnel after reaching the location where they are needed. The output of these signals was described by their name and not associated with the control path. It was difficult for some participants to understand where the blue control signals are coming from. A direct connection between the control unit and the required location would be preferred. However, this might result in an overwhelming net of signals.

4. **Complicated data path**
   The same problem occurring with the control path also translates to the data path. The source of signals used in the CPU architecture is not immediately noticed. Every data path signal comes from the control unit, as does every signal from the control path. A direct connection is preferred, likewise to the control path.

5. **Not immediately noticeable reason for static ISA defined numbers**
   The information why RISC-V uses 12 and 4 as static numbers for some instructions is not provided clearly enough. For the respective instructions the information should be described in more detail.

6. **General procedure of instruction decoding**
   The instruction decoding should not only be displayed in the architecture, but on a conceptual level where it can be seen from everywhere. The bit selection (green in the architecture) was not understood completely as there is no information describing the bit wise structure of an instruction as shown in figure 2. This information should be provided in the graph or in the instructions list.

7. **On-demand information**
   The step-by-step information is only shown when the respective step which has the information is reached. Some participants wanted to read information after the fact but were not able to locate it due to the on-demand behavior. All information should be accessible at any step.

8. **Different instructions between objdump and instruction list**
   When using objdump in the compilation screen, a list of instructions will be printed. Some of these instructions are represented by default as pseudo instructions. These include *mv* (move), *nop* (no operation), etc. This confused a participant when comparing the two sets. Adding the tag *-M no-aliases* to objdump removes these pseudo instructions and only prints valid RISC-V instructions which equal the ones shown in the instruction list of the simulation screen.

9. **Start with desaturation**
   At the beginning of the simulation the architecture should be desaturated and only saturated when the element is used. This has already been implemented.

10. **Incorrect description**
    Register Destination (RD) is sometimes regarded as Return Register and should be changed.

11. **Refocus on current step**
    There is currently no way to focus back on the element which was focused automatically. Once the user moves the view he or she can go everywhere in the architecture and maybe cannot remember what elements had the focus. There should be a refocus button to enable this behavior.

12. **Subcomponent back button**
    There needs to be an additional button to go back to the overview once in a subcomponent's tab. All users had problems figuring out that the tabs need to be used to choose between subcomponents.

13. **Hex representation**
    The program counter and the values in the architecture are only shown in decimal. Some users wanted to know the respective hex value. It should be possible to select between decimal and hexadecimal representation.

14. **Infinite zooming**
    It should not be possible to zoom out further then the subcomponent or overview. This prevents users from seeing the other components which are rendered on the same plane but at a different location.

15. **Additional ALU information**
    One student wanted to know how the shown logic blocks (addition, subtraction, etc.) are typically implemented in digital logic. The hover information could be used to display this additional information when hovering over the logic block.

16. **Inline C source code**
    To aid the concept of C compilation and the resulting instructions the corresponding C source code should be embedded into the instruction list. This equals the behavior of objdump when providing the *–source* parameter, which inlines the C source among the instructions

17. **Consistent naming convention**
    The instruction bytes passed to the control unit are sometimes referred to as “raw instruction”, “unparsed” or “encoded”, these should be renamed to a consistent “encoded” which is the more concrete term. The decoded instruction was equally sometimes named different. One user thought these are different values.

18. **Program halt**
    There is no information when the program is complete or halted and cannot be stepped further. A popup has been implemented which states that the program reached an *ecall* instruction and completed or is stuck in a loop.

19. **Editor autosave**
    There is no information on if the file was automatically saved in the editor of the compilation screen. Users wanted to save the file by themselves but couldn’t find a save option.

20. **Memorized scrollbar position**
    The editor and other scrollable elements should memorize their scrolling position when the compile and simulation screen are reopened to allow for a better experience.

21. **Scrollbars**
    There should be scrollbars in the memory and instruction views (already implemented) to enable faster scrolling than possible with a mouse wheel.

22. **Hotkeys**
    There should be hotkeys to control the guided simulation.

23. **Simulate the whole program**
    The whole program should be executable with a single button press. This has already

been implemented partially via a feature which allows the program to execute until a certain instruction is reached. This feature can be accessed by using “Continue to” when right clicking on an instruction.

24. **Step whole instruction**
    Currently the user can only step through the stages of an instruction. There is no feature which allows the user to step over one function at a time. Although, the “Continue to” feature can be used here.

25. **Instructions embedded into memory**
    Instructions exist both in the instruction list on the left and in the memory on the right. When comparing the hexadecimal values, the connection can be seen. To aid the connection between memory and instruction list the instructions could be embedded into the memory or connected with the instruction in some other way.

26. **Different cursor icons**
    The cursor should be different depending on whether the element is clickable, draggable or elsewise.

8 Conclusion

The results of the usability test indicate a significant benefit of the developed simulator in the educational domain. The current landscape of simulators shows a gap for a more informational and guided approach for CPU simulation tools. This gap has been narrowed by the developed simulator which focuses only on the educational aspects necessary in a classroom. Even though there are improvements to be made, the usability test participants agreed on the fact that the simulator can be utilized in a CPU architecture course and even be used as a main course pillar. The requirements for such a special use simulator have been stated in the first chapters and have in large parts been fulfilled by Apate. The only requirement not fulfilled is the web-based approach other papers stated as a beneficiary element. This can still be further improved upon in a later version of Apate.

However, only real usage in a university course will provide the necessary feedback which is needed to state that the indicated gap has been closed by Apate. With the goal of using the simulator in real university courses, this feedback will be gathered at a larger scale at a later point in time. For now, the small sample size of seven users is one step in the right direction, however.

During development, the most prominent takeaway of creating a CPU simulator is the flexibility, simplicity and clarity needed by the backend simulation. Different approaches using HDL were used, but in the end only a custom CPU architecture with a JavaScript based backend offered the necessary features. While still being separated from the user interface code the backend simulation offered the simplicity needed to generate performant and rich visualizations of the internal components of a CPU and their interconnecting data paths.

Comparing Apate to other simulators, listed in the state of art section in the beginning, a clear visual distinction can be made. Although these simulators also focus on the visualization it can be objectively stated that these interfaces are less performant and appealing to look at and to use. Future interfaces for simulators which also focus on the educational effect should take the quality-of-experience into account which can only be achieved by a state-of-the-art toolkit and a user experience approach in mind.

Bibliography

[1] O'Reilly Media, Inc, *Katacoda - Interactive Learning and Training Platform for Software Engineers*, 2021. [Online]. Available: https://www.katacoda.com/ (visited on May 11, 2021).

[2] bleeptrack and blinry, *Oh My Git! - An open source game about learning Git!* 2021. [Online]. Available: https://ohmygit.org/ (visited on May 11, 2021).

[3] Bosko Nikolic, Zaharije Radivojevic, Jovan Djordjevic, and Veljko Milutinovic, "A survey and evaluation of simulators suitable for teaching courses in computer architecture and organization," *IEEE Transactions on Education*, vol. 52, no. 4, pp. 449–458, 2009. DOI: 10.1109/TE.2008.930097.

[4] P. W. C. Prasad, Abeer Alsadoon, Azam Beg, and Anthony Chan, "Using simulators for teaching computer organization and architecture," *Computer Applications in Engineering Education*, vol. 24, no. 2, pp. 215–224, 2016. DOI: https://doi.org/10.1002/cae.21699. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.21699. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cae.21699.

[5] Carl Burch, "Logisim: A graphical system for logic circuit design and simulation," *J. Educ. Resour. Comput.*, vol. 2, no. 1, pp. 5–16, Mar. 2002, ISSN: 1531-4278. DOI: 10.1145/545197.545199. [Online]. Available: https://doi.org/10.1145/545197.545199.

[6] Vanja Luković, Radojka Krneta, Ana Vulović, Christos Dimopoulos, Konstantinos Katzis, and Maria Meletiou-Mavrotheris, "Using logisim educational software in learning digital circuits design," in *Proceedings of 3rd International Conference on Electrical, Electronic and Computing Engineering IcETRAN*, 2016, pp. 5–1.

[7] Ayaz Akram and Lina Sawalha, "A survey of computer architecture simulation techniques and tools," *IEEE Access*, vol. 7, pp. 78 120–78 145, 2019. DOI: 10.1109/ACCESS.2019.2917698.

[8] Michel Pollet (aka. buserror), *simavr*, 2020. [Online]. Available: https://github.com/buserror/simavr (visited on Feb. 17, 2021).

[9] Labcenter Electronics, *PCB Design and Circuit Simulator Software - Proteus*, 2021. [Online]. Available: https://www.labcenter.com/ (visited on Feb. 17, 2021).

[10] Free Software Foundation, Inc., *Microchip Studio for AVR and SAM Devices*, 2021. [Online]. Available: https://www.microchip.com/en-us/development-tools-tools-and-software/microchip-studio-for-avr-and-sam-devices (visited on Apr. 29, 2021).

[11] A. Yavuz Oruc, Abdullah Atmaca, Y. Nevzat Sengun, and A. Semi Yeniyol, “Codeapeel: An integrated and layered learning technology for computer architecture courses,” *CoRR*, vol. abs/2104.09502, 2021. arXiv: 2104.09502. [Online]. Available: https://arxiv.org/abs/2104.09502.

[12] Andrea Spadaccini, Davide Patti, Maurizio Palesi, and Fabrizio Fazzino, “Edumips64, a visual cpu simulator for teaching computer architecture,” *submitted for publication*, 2011.

[13] Kenneth Vollmar and Pete Sanderson, “Mars: An education-oriented mips assembly language simulator,” *SIGCSE Bull.*, vol. 38, no. 1, pp. 239–243, Mar. 2006, ISSN: 0097-8418. DOI: 10.1145/1124706.1121415. [Online]. Available: https://doi.org/10.1145/1124706.1121415.

[14] Cecile Yehezkel, William Yurcik, Murray Pearson, and Dean Armstrong, “Three simulator tools for teaching computer architecture: Little man computer, and rtlsim,” *J. Educ. Resour. Comput.*, vol. 1, no. 4, pp. 60–80, Dec. 2001, ISSN: 1531-4278. DOI: 10.1145/514144.514732. [Online]. Available: https://doi.org/10.1145/514144.514732.

[15] Guillaume Savaton, *emulsiV*, 2021. [Online]. Available: https://github.com/Guillaume-Savaton-ESEO/emulsiV (visited on Apr. 30, 2021).

[16] Roberto Giorgi and Gianfranco Mariotti, “WebRISC-V: A web-based education-oriented risc-v pipeline simulation environment,” in *ACM Workshop on Computer Architecture Education (WCAE-19)*, Phoenix, AX, (USA), Jun. 2019, pp. 1–6, ISBN: 978-1-4503-6842-1/19/06. DOI: 10.1145/3338698.3338894. [Online]. Available: http://www.dii.unisi.it/~giorgi/papers/Giorgi19-wcae.pdf.

[17] Irina Branovic, Roberto Giorgi, and Enrico Martinelli, “Webmips: A new web-based mips simulation environment for computer architecture education,” in *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, 2004, 19–es.

[18] Morten Borup Petersen, *Ripes*, 2021. [Online]. Available: https://github.com/mortbopet/Ripes (visited on Feb. 17, 2021).

[19] Greg James, Barry Silverman, and Brian Silverman, “Visualizing a classic cpu in action: The 6502,” in *ACM SIGGRAPH 2010 Talks*, ser. SIGGRAPH ’10, Los Angeles, California: Association for Computing Machinery, 2010, ISBN: 9781450303941. DOI: 10.1145/1837026.1837061. [Online]. Available: https://doi.org/10.1145/1837026.1837061.

[20] Bo Su and Li Wang, “Application of proteus virtual system modelling (vsm) in teaching of microcontroller,” in *2010 International Conference on E-Health Networking Digital Ecosystems and Technologies (EDT)*, vol. 2, 2010, pp. 375–378. DOI: 10.1109/EDT.2010.5496343.

[21] Tom Scherlis, *logisim RISC-V CPU*, 2021. [Online]. Available: https://github.com/Toms42/logisim-RISC-V-CPU (visited on Apr. 29, 2021).

[22] RISC-V International, *RISC-V ISA Specification*, 2021. [Online]. Available: https://github.com/riscv/riscv-isa-manual (visited on Mar. 4, 2021).

[23] *RISC-V Options*, 2021. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/RISC-V-Options.html (visited on Mar. 4, 2021).

[24] Abraham Silberschatz, Peter B. Galvin, Greg Gagne, and John Wiley, *Operating system concepts*. Wiley, Copyright, 2018. [Online]. Available: https://www.wiley.com/en-us/Operating+System+Concepts%2C+10th+Edition-p-9781119320913 (visited on Mar. 5, 2021).

[25] Atmel, *ATmega328P Datasheet*, 2021. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf (visited on Mar. 4, 2021).

[26] Noam Nisan and Shimon Schocken, *The Elements of Computing Systems - Building a Modern Computer from First Principles*, 2016. [Online]. Available: https://mitpress.mit.edu/books/elements-computing-systems (visited on Mar. 6, 2021).

[27] Ryan J. Leng, *The Secrets of PC Memory: Part 1*, 2007. [Online]. Available: https://www.bit-tech.net/reviews/tech/memory/the_secrets_of_pc_memory_part_1/3/ (visited on Mar. 6, 2021).

[28] Claire Xenia Wolf, *PicoRV32 - A Size-Optimized RISC-V CPU*, 2021. [Online]. Available: https://github.com/cliffordwolf/picorv32 (visited on Feb. 1, 2021).

[29] Unity Technologies, *Unity Real-Time Development Platform*, 2021. [Online]. Available: https://unity.com/ (visited on Apr. 30, 2021).

[30] *PyGame*, 2021. [Online]. Available: https://www.pygame.org/ (visited on Apr. 30, 2021).

[31] Riverbank Computing Limited, *pyQt*, 2021. [Online]. Available: https://www.riverbankcomputing.com/software/pyqt/ (visited on Apr. 30, 2021).

[32] *libGDX*, 2021. [Online]. Available: https://libgdx.com/ (visited on Apr. 30, 2021).

[33] OpenJS Foundation, *Electron*, 2021. [Online]. Available: https://www.electronjs.org/ (visited on Apr. 30, 2021).

[34] OpenJDK, *JavaFX*, 2021. [Online]. Available: https://openjfx.io/ (visited on Apr. 30, 2021).

[35] Microsoft, *.NET*, 2021. [Online]. Available: https://dotnet.microsoft.com/ (visited on Apr. 30, 2021).

[36] Google, *Angular*, 2021. [Online]. Available: https://angular.io/ (visited on Apr. 30, 2021).

[37] mrdoob, *Three.js - JavaScript 3D library*, 2021. [Online]. Available: https://threejs.org/ (visited on Apr. 30, 2021).

[38] *p5.js*, 2021. [Online]. Available: https://p5js.org/ (visited on Apr. 30, 2021).

[39] Viktor Chlumský, "Shape Decomposition for Multi-channel Distance Fields," *Czech Technical University in Prague*, 2015. [Online]. Available: https://github.com/Chlumsky/msdfgen.

[40] GitHub, Inc., *GitHub*, 2021. [Online]. Available: https://github.com (visited on Feb. 1, 2021).

[41] OpenCores.org, *OpenCores Processors*, 2021. [Online]. Available: https://opencores.org/projects?expanded=Processor (visited on Feb. 1, 2021).

[42] GHDL, *GHDL*, 2021. [Online]. Available: https://github.com/ghdl/ghdl (visited on Feb. 1, 2021).

[43] Verilator, *Verilator*, 2021. [Online]. Available: https://github.com/verilator/verilator (visited on Feb. 1, 2021).

[44] Siemens Digital Industries Software, *ModelSim*, 2021. [Online]. Available: https://www.mentor.com/products/fv/modelsim/ (visited on Feb. 1, 2021).

[45] Stehan Nolting, *neo430*, 2021. [Online]. Available: https://github.com/stnolting/neo430 (visited on Feb. 1, 2021).

[46] Alex I. Kuznetsov, *lxp32*, 2021. [Online]. Available: https://lxp32.github.io/ (visited on Feb. 1, 2021).

[47] *ELF(5) - Linux Programmer's Manual*, 2017. [Online]. Available: https://manpages.debian.org/stretch/manpages/elf.5.en.html (visited on May 2, 2021).

[48] *elfy.io | Online ELF editor*, 2021. [Online]. Available: https://elfy.io/ (visited on May 2, 2021).

[49] Andreas Holzinger, "Usability engineering methods for software developers," *Commun. ACM*, vol. 48, no. 1, pp. 71–74, Jan. 2005, ISSN: 0001-0782. DOI: 10.1145/1039539.1039541. [Online]. Available: https://doi.org/10.1145/1039539.1039541.

List of Figures

Figure 1 The CPU visualization landscape. The left side starts with simulators which show no architectural CPU details. The more right one goes the more detail will be shown until the hardware is reached. 3
Figure 2 Proteus' schematic editor with electronics simulation. 5
Figure 3 Proteus' source code view with disassemble instructions. 5
Figure 4 Registers of the AVR processor shown in Proteus. 5
Figure 5 Memory of the CPU which is updated after each instruction shown in Proteus. 6
Figure 6 A simulation in Microchip Studio. 6
Figure 7 The web-based structural visualization of emulsiV 7
Figure 8 A simple calculator demo in WebRISC-V 8
Figure 9 Simulation done in Ripes 9
Figure 10 Logic simulation of a RISC-V CPU [21] in LogiSim 10
Figure 11 Transistor level simulation of the 6502 processor [19] 10
Figure 12 List of instructions addressable by the program counter. The instruction currently marked will be executed. After the execution the Program Counter will be advanced by one. (View the document in *Adobe Acrobat Reader* to see animations) 14
Figure 13 Instruction Cycle 19
Figure 14 The developed non-pipelined 5 stage CPU architecture of *Apate*. The control path and program counter are marked blue. Depending on the instruction type and the instruction string (blue), which are parsed by the control unit (green), the multiplexers forward different signals. 20
Figure 15 Example jump with `jal` and `jalr`. (1) saves the PC to r1 and jumps to 24. (2) loads the PC from r1 to jump back. 22
Figure 16 Simplified memory hierarchy based on an illustration by Ryan J. Leng [27]. . . 23
Figure 17 Memory layout 24
Figure 18 Memory component of the proposed CPU architecture 24
Figure 19 Register component of the proposed CPU architecture 25
Figure 20 Control unit schematic of step 1 and 2. Parsing of the opcode on the left and parsing of the operands on the right. The green wire represents a bit selection. On the left bit 0 to 6 is selected from the unparsed instruction. On the right the bits per value are selected according to the encoding of table 2. These values are always at the same location in contrast to the immediate value which is parsed in step 3. 26

Figure 21 Parsing the immediate value in step 3 of the control unit 28
Figure 22 Parsing the instruction name in step 4 of the control unit. 29
Figure 23 Internal layout of the arithmetic logic unit of the proposed RV32I CPU architecture of Apate. Operations from top to bottom: Addition, subtraction, bitwise and, bitwise xor, bitwise or, shift left by OP2, shift right by OP2, set one if OP1 is less than OP2. All operations are performed combinatorial at the same time. The necessary result is selected by the multiplexer controlled by the instruction signals (blue). 30
Figure 24 Simplified form of the branch evaluator as shown in Apate. The underlying logic of the comparison is not shown. 31
Figure 25 Branch evaluator example using two comparators from a RISC-V CPU designed in LogiSim [21]. 31
Figure 26 The welcome screen of *Apate* with three options. 32
Figure 27 The compilation screen of *Apate* with the folder structure on the left and the text editor on the right. 34
Figure 28 The simulation screen. Highlighting the currently used wires and components depending on the instruction. The currently used instruction is marked on the left in blue. The center shows the CPU architecture. On the right the registers are shown. (View the document in *Adobe Acrobat Reader* to see animations) . 34
Figure 29 Inner logic of step 1 and step 2 of the control unit as shown in *Apate* with highlighted elements which are activated once the opcode is parsed 36
Figure 30 Inner logic of the branch evaluator as shown in *Apate* with highlighted elements depending on the current `lui` instruction 37
Figure 31 Inner logic of the arithmetic logic unit as shown in *Apate* with highlighted elements depending on the current `bne` instruction 37
Figure 32 The interaction between CPU and the user interface. The CPU only updates values in a separate bindings file. Angular templates and Three subscribe to these values to update the corresponding elements of the user interface. . . . 41
Figure 33 A simple depiction of the ELF format showing the different sections by Ange Albertini titled: ELF 101 a Linux executable walkthrough. 50
Figure 34 Results of the survey questions asked in the interview after using Apate. The abbreviation A.u.A. means "After using Apate". Question 1: (1) I don‘t know anything - (5) I know a lot. Question 2 to 18: (1) Disagree - (5) Agree completely. 56

List of Tables

Table 1 All parsable values in an RISC-V instruction and their description 15
Table 2 RISC-V ISA encoding for each instruction type [22]. The instruction types correspond to the type listed in table 3. 16
Table 3 RISC-V base integer instructions [22]. Bit selectors are depicted as for example [0:3], selecting bit 0 to 3. The letter M depicts the program memory. 16
Table 4 RISC-V defined registers 25

List of Sourcecodes

1 Verilog example for comparing the 7 first bits to predefined opcode bit sequences. . . 26
2 Verilog example for decoding the immediate value and selecting the correct value depending on the opcode. Code adopted from PicoRV32 [28]. 27
3 Building and running a VHDL design. 43
4 neo430_c_bindings.vhd 44
5 vhpidirect.c 44
6 Building VHDL with VHPIDIRECT C program 45
7 Initiating the testbench for the Vpico_testbench top module in C. 46
8 Loading a dynamic shared library into Node with ffi-napi. 47
9 Concatenated output of *riscv64-unknown-elf-readelf -a main.elf*. 52
10 Unit test comparing the size of the section headers parsed by JavaScript and *riscv64-unknown-elf-readelf*. 53
11 Unit test comparing the result of the JavaScript based instruction parser and a predefined string. 53

List of Abbreviations

CPU Central Processing Unit

RISC Reduced Instruction Set Computer

ISA Instruction Set Architecture

PC Program Counter

UI User Interface

HDL Hardware Design Language

RISC-V A RISC ISA spelled "riscv five"

RV32I RISC-V 32-bit base integer instruction set architecture

RV32M RISC-V 32-bit ISA multiplication extension

r0 Register with address zero. Analogue to r1, r2, etc. or x0, x1, etc.