JOHANNES KEPLER
UNIVERSITY LINZ

Author
Jan Rendel

Submission
Institute for Complex Systems

Thesis Supervisor
Univ.-Prof. Dr.
Daniel Große

Assistant Thesis Supervisor
DI **Simon Reitinger**

January 2026

Integration and Visualization of a RISC-V Multi-Cycle Core in SonicRV

Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Abstract

RISC-V processor architectures are prevalent in educational environments due to the free and open specification of RISC-V. To facilitate comprehension of these architectures, SonicRV, a web-based learning platform, provides interactive visualization and analysis capabilities for both single-cycle and pipelined architectures. However, it currently does not support multi-cycle processors.

To close this gap, this bachelor's thesis presents the integration and visualization of a RISC-V 32-bit multi-cycle processor within SonicRV. The thesis details the architectural characteristics of an existing RISC-V 32I multi-cycle processor and describes the systematic modifications and extensions required to integrate the processor into the existing SonicRV infrastructure. Finally, the successful integration of the multi-cycle processor is demonstrated through a comprehensive walkthrough of a sample RISC-V program executed within SonicRV.

Contents

Chapter 1

Introduction

The rise of open-source *Instruction Set Architecture* (ISA) has opened a new era in the matter of computer architectures. Among these, RISC-V [15], a *Reduced Instruction Set Computer* (RISC) architecture, has appeared as an influential ISA, gaining more and more importance over the past few years. Because of RISC-V's extensibility and modularity it is becoming an attractive alternative to proprietary ISAs like AMD's and Intel's *x86* architecture in academia and industry. Having these properties, RISC-V permits researchers and developers to experiment with novel processor design without constraints such as licensing fees or other restrictions.

A core part of understanding processor design, is the simulation combined with visualization of programs. SonicRV, a web-based education platform for simulating and visualizing small assembly programs, has been created at the *Institute for Complex Systems* (ICS) to aid in the process of understanding various RISC-V core designs, such as single-cycle and pipelined architectures. This is done by offering a platform for submitting individual RISC-V 32I (RV32I) assembly code. The program is then simulated by GHDL [2] on a *Very High Speed Integrated Circuit* (VHSIC) *Hardware Description Language* (VHDL) [6] implementation of the processor, analyzed and finally visualized by dynamically highlighting a high-level block diagram. The block diagram contains the control and data path with its components such as the *Arithmetic Logic Unit* (ALU), the *Register File* and the *Control Unit*. By stepping through the simulated instructions, the user is able to trace changes of registers, memory content and data signals, gaining great knowledge of the current state of the core. Additionally, the program specific waveform can be viewed with Surfer [11, 13], which is embedded inside the SonicRV web application.

This bachelor's thesis aims to develop an enhanced high-level visualization for a multi-cycle RV32I core, derived from the textbook [5] and implemented in [14], extending SonicRV[1]. In addition to the existing cores, the multi-cycle RV32I core is integrated into SonicRV, adapting features for the multi-cycle core. The newly added features aim to improve the understanding of the key concepts of multi-cycle cores in general, such as the *Finite State Machine* (FSM) in the *Control Unit* of the core. Additionally, the FSM is also visualized and rendered besides the main block diagram, to further aid in understanding the multi-cycle architecture. The overall goal is to further improve the educational value of SonicRV, enabling students and researchers to gain a deeper understanding of the most important principles of multi-cycle RISC-V architectures.

To demonstrate the extended visualization system for the multi-cycle core in SonicRV, an example program is simulated on the multi-cycle RISC-V core within SonicRV. The execution of the program is stepped through, instruction by instruction, showing the visualization of the main core components as well as the FSM graph. This demonstration showcases how the system can be used to illustrate fundamental concepts of multi-cycle execution, such as instruction fetching, decoding, execution, memory read, and memory write-back. The user can observe the data movement between registers, the operations performed by the ALU and the role of the *Control Unit* and its FSM.

[1] Recently a journal paper on SonicRV has been accepted [3]. It already summarizes some aspects of the developments made in this bachelor's thesis, but only from a user perspective.

The integration of the multi-cycle core extends the capabilities of SonicRV, making it an even more valuable tool for teaching and exploring RISC-V processor design. By providing a clear and interactive view of the internal workings of a multi-cycle core, this work contributes to a deeper understanding of processor design principles and promotes the adoption of RISC-V in educational and research settings. The enhanced visualization will enable users to more easily understand complex interactions within the processor, encouraging a more intuitive understanding of how a RISC-V core operates at a fundamental level.

This thesis is structured as follows: Chapter 2 reviews the preliminaries, that is SonicRV, the multi-cycle core and the *Waveform Analysis Language* (WAL). Thereafter, in Chapter 3, the actual integration of the multi-cycle core into SonicRV is described in detail. Chapter 4 demonstrates a sample program executed on the multi-cycle core in SonicRV, showing the dynamic visualization of the instruction table, core and FSM with images. Finally, in Chapter 5, the thesis is summarized and concluded.

Chapter 2

Background

This chapter covers all the prerequisites and existing tools on which this bachelor's thesis is built upon. All used tools are explained, to provide a basic understanding of their concepts and features.

2.1 SonicRV

SonicRV[1] is a publicly accessible web application that facilitates visualization of RV32I instructions, currently supporting multiple implementations of single-cycle and pipelined RISC-V core architectures. It is capable of simulating small individual RV32I programs and visualize them in a web browser on a high level block diagram of the chosen core architecture. First, a specific processor can be selected. Then a RV32I program to be simulated can be written in an embedded editor with RV32I assembly syntax highlighting, see Figure 2.1. Also, representative example programs are provided for each core.

After simulating a program, a dynamic instruction table displays all simulated RV32I instructions to the user, showing the instruction name and the *Program Counter* (PC) of the instruction, see on the left in Figure 2.2. A dynamically highlighted high level block diagram shows the current state of the processor, highlighting important components and signals with their values for the selected instruction, see center of Figure 2.2. Current register and memory contents are monitored and can be viewed on the right in Figure 2.2. Additionally, the waveform of the simulated program can be viewed using Surfer [11, 13] in the "Waveform" tab, see Figure 2.3.

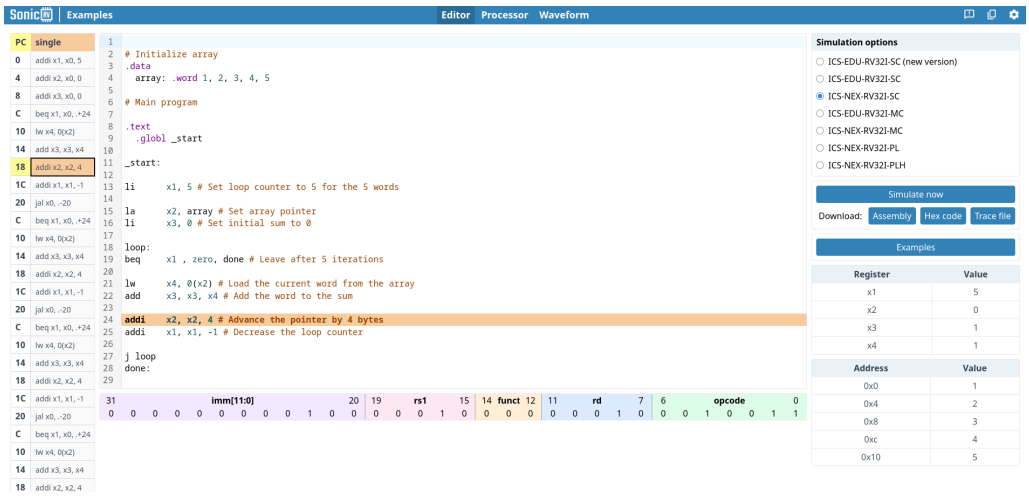

Figure 2.1: Program editor, instruction table and instruction decoder in SonicRV

[1] https://sonic-rv.ics.jku.at

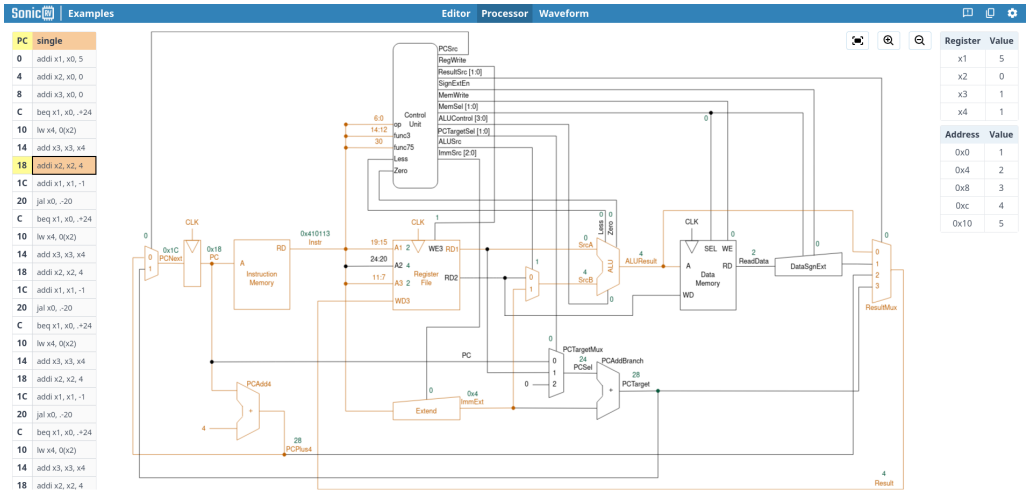

Figure 2.2: Instruction table, single cycle block diagram, memory and register contents in SonicRV

2.1.1 General Architecture

SonicRV is built on a conventional frontend and backend software architecture. The frontend's main purpose is the core selection, program submission, visualization and highlighting, whereas the backend is responsible for simulation, waveform analysis and caching.

Frontend and backend servers lie on the same machine and communicate over *Hypertext Transfer Protocol* (HTTP), as illustrated in Figure 3.1 in the order of the depicted numbers. The frontend server invokes the backend server by requesting the simulation and analysis of a program, transmitting information about the selected processor and the user program, which should be simulated. The backend parses the input parameters from the HTTP-Request and checks the submitted program for syntax and semantic correctness, using an RV32I assembler. If the program appears to be correct, the backend server then invokes GHDL [2] to simulate the program, by loading the program into a specific address in the processors memory component. Upon successful simulation, GHDL outputs a *Value Change Dump* (VCD), including signal information for every time stamp, otherwise it immediately responds to the frontend with an error message, informing the user the program execution has failed. Subsequently, the VCD is then analyzed by WAL, as described in Section 2.3. This analysis extracts the required information for the frontend and writes the output in a Javascript Object Notation (JSON) file in a predefined format. The backend then responds to the frontend with a HTTP-Response including the generated *Javascript Object Notation* (JSON) file. The frontend then parses the JSON file and provides a layout for inspecting the simulated program with an instruction table, a block diagram and memory and register contents, see Figure 2.2.

Backend

The main component of the backend is a Python Flask server, which is responsible for providing the HTTP *Application Programming Interface* (API). This API serves as the interface through which the frontend server interacts. This API provides functionality, for example, to simulate a program, specified in the frontend, in a selected core. The result of the simulation, written to a JSON file, is then retrieved by the frontend through a subsequent API call.

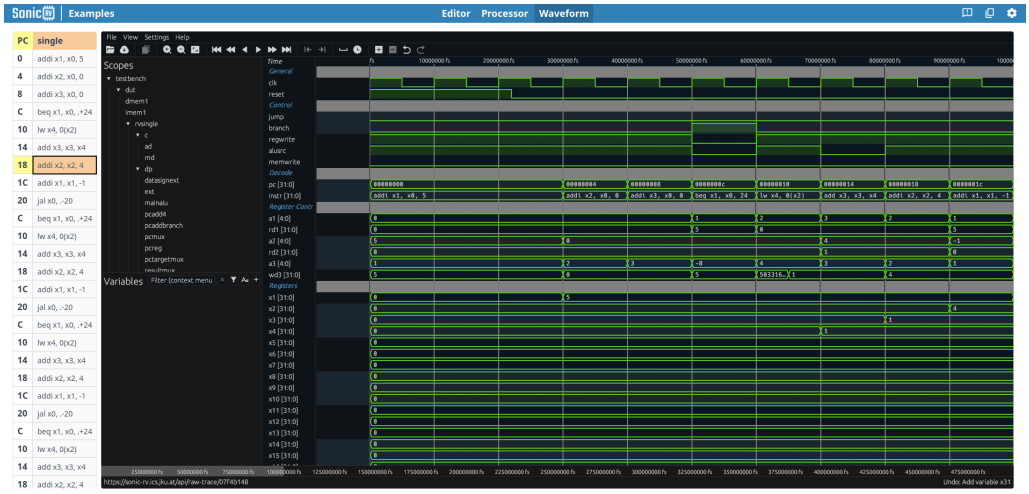

Figure 2.3: Surfer tab for a simulated program on a single cycle core

The backend's functionality is highly dependent on a well defined file structure. Each provided core has its own directory. This directory includes, among other elements, all files for the VHDL sources, the block diagram, the predefined example programs, the core-specific WAL programs and a file for the initial Surfer state. This directory structure facilitates the seamless addition, modification or removal of cores.

Frontend

The frontend operates on a NodeJS server utilizing Svelte and TypeScript, ensuring a well-structured layout and responsive user interface. To deliver essential data to the user, the frontend communicates with the backend's API. This interaction enables the frontend to present all available cores and their example programs, faciliate the submission of RV32I user programs, and visualize these programs on a selected core.

2.2 RV32I Multi-Cycle Core

In this section the general hardware design and characteristics of a RV32I multi-cycle core is explained briefly, as well as differences to RV32I single-cycle and pipelined cores. But first, let us introduce the RV32I ISA, which all cores in SonicRV have in common.

2.2.1 RV32I Instruction Set Architecture

Generally, RISC-V is an open ISA that is completely free to use [12, 15]. The base integer RISC-V ISA comes with a minimal set of integer instructions for 32 and 64 bit system architectures. This base variant of the ISA is called RV32I for 32 bit systems, and RISC-V 64I (RV64I) for 64 bit systems respectively, where the *I* denotes the integer variant. Although being only a minimal ISA, RISC-V processors can be customized to support one or more instruction set extensions, such as multiplication and division (M-extension) or floating point arithmetic (F-extension) [15]. In the field of education and processor visualization however, the RV32I minimal ISA is a good starting point to show the basic principles of different processor types.

The RV32I ISA defines 32 registers: one constant zero register and 31 general purpose registers, which are refered to as *x0* to *x31* [15]. RV32I specifies instructions for addition,

subtraction, logic operations, shifting, branching and memory operations. Those instructions are grouped into six instruction types, where every instruction type holds similar instructions:

- **R-type:** Register to register transfer instructions.
- **I-type:** Immediate/memory to register transfer instructions.
- **S-type:** Register to memory transfer instructions.
- **B-type:** Branch instructions.
- **U-type:** Upper immediate instructions.
- **J-type:** Jump and link instructions.

Each instruction type has a specific 32-bit instruction layout, providing the required information for the specific type, depicted in Figure 2.4. That is a 7-bit *opcode* field, 5-bit fields for source (*rs1*, *rs2*) and destination registers (*rd*), varying-bit and varying-position immediate (*imm*) fields, the *funct3* and the *funct7* field. Instructions within one instruction type often share the same opcode, and thus are further distinguished by their *funct3* and *funct7* fields.

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

Figure 2.4: RV32I instruction formats

As an example, the binary representation of the *or x4, x5, x6* instruction (R-type) is illustrated in Figure 2.5. This instruction executes a bitwise OR between the source registers *x5* (rs1) and *x6* (rs2), storing the result in the destination register *x4* (rd). The *funct7*, *funct3*, and *opcode* fields are defined by the RV32I specification, indicating that this is an *or* instruction. The fields *rs1*, *rs2*, and *rd* specify the relevant source and destination registers utilized in this instruction. If the instruction were changed to *or x1, x2, x3*, the *funct7*, *funct3*, and *opcode* fields would remain unchanged, while the *rs1*, *rs2*, and *rd* fields would change accordingly.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
0000000	00110	00101	110	00100	0110011	or x4, x5, x6

Figure 2.5: Binary representation of the *or x4, x5, x6* instruction

Having explained the principles of the RV32I ISA , the existing hardware implementation of the multi-cycle core shall be briefly discussed and compared to the single-cycle and pipelined architectures.

2.2.2 Hardware Implementation and Differences to Other Cores

Every processor type, that is single-cycle, multi-cycle and pipelined, is designed to run machine code defined by its ISA. To execute instructions, the processor utilizes dedicated hardware components to perform arithmetic, logic, memory and other operations.

A single-cycle processor yields the least complex design [5]. As the name already suggests, a single-cycle processor executes every instruction within exactly one clock cycle. The downside of this is a long critical path[2], which forces slow clock speeds, because the minimal possible cycle duration must accommodate the longest operation.

A multi-cycle processor tries to mitigate this issue, by dividing the execution paths into multiple steps [5]. As a result, multi-cycle processors execute each instructions within multiple consecutive clock cycles to shorten the critical path in the architecture, permitting faster clock speeds. Note that not all instructions need the same exact amount of clock cycles to complete, since not every instruction utilizes every hardware component inside the processor. The exact amount of clock cycles (or steps) needed to complete an instruction is determined by an additional hardware component, a hardware-implemented Finite State Machine (FSM), which is further discussed in Section 2.2.3.

In comparison to single-cycle and multi-cycle cores, which execute instructions purely sequentially, pipelined cores introduce parallelism, allowing to execute multiple instructions at once, by overlapping execution stages [5]. This increases throughput, but requires additional control logic to handle hazards and maintain correct behavior when specific instructions interact.

SonicRV already provides implementations and visualizations for both single-cycle and pipelined cores. The missing part, the multi-cycle core, is integrated and visualized in this thesis. For clarity, the most unique part of the multi-cycle processor, the FSM, which determines the state flow, is described in more detail in the following section.

2.2.3 State Flow of a Multi-Cycle Core

The state flow of a multi-cycle processor is determined by an FSM which is integrated into the *Control Unit*. This FSM defines the steps involved in executing instructions, which is crucial in the processor design [5]. It specifies the sequence and number of steps (or clock cycles) required to execute each instruction. Each state of the FSM sets the control signals that drive the appropriate hardware components and guide the signal flow. When a state completes, the FSM selects the next state in the execution sequence until the instruction itself is completed.

The FSM of the multi-cycle processor that is integrated into SonicRV in this thesis, is depicted in Figure 2.6, based on [5, 14]. This diagram illustrates the states (represented by circles) and transitions (represented by arrows), with corresponding descriptions of signal changes that are then distributed by the *Control Unit* to the hardware components. The FSM defines 15 individual states, including the *Reset* state. This state is the initial state upon every processor (re-)start or error. The *S0 - Fetch* state is always executed after the *Reset* state and is the first state of any instruction.

For a concrete example, the state flow of the already familiar *or x4, x5, x6* is explained step by step. We begin at the *S0 - Fetch* state, either coming from the *Reset* state or from the termination of a previous instruction. Here, the signals in the *Control Unit* are set to fetch the instruction from memory and increment the *PC*. Then the processor transitions to the *S1 - Decode* state, where the instruction itself is decoded by the *Control Unit*. Up until this point every instruction behaves similarly.

However, the next step depends on the instructions *opcode*. Given that the *or* instruction is an R-type instruction, the FSM transitions to the *S6 - ExecuteR* (Execute R-type) state. In this state the actual bitwise OR operation of registers *x5* and *x6* is performed by the *ALU*, as indicated by the *ALU-Signals* in the state.

Subsequently, the FSM transitions to the *S7 - ALUWB* (ALU Writeback) state. Here, the result from the previous state is stored back in register *x4*. Finally, the FSM reverts

[2]Longest continuous, most time consuming combinational path of the hardware design

to the *S0 - Fetch* state, indicating the completion of the current instruction and allowing the next one to be fetched.

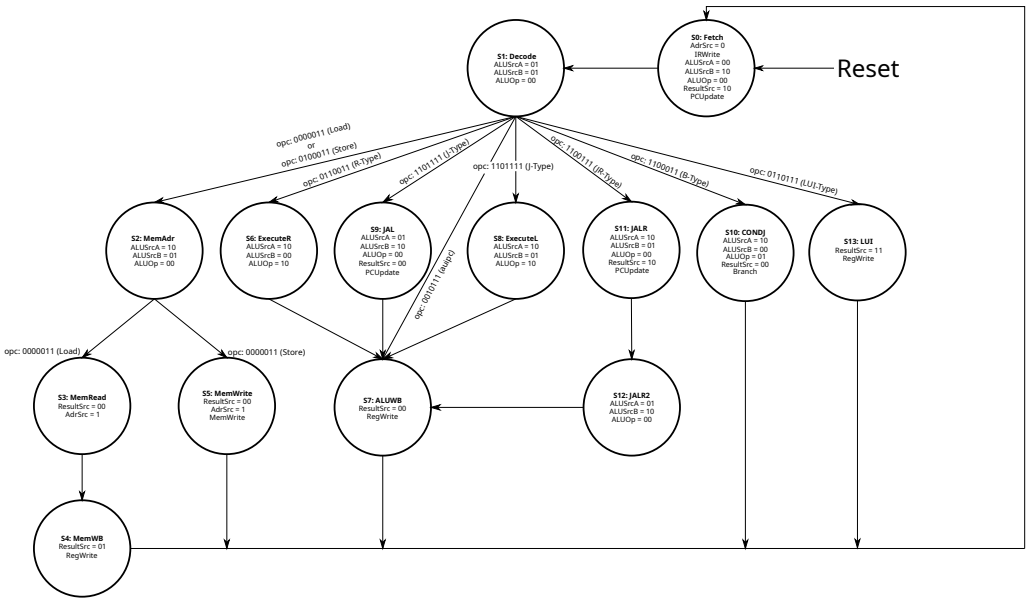

Figure 2.6: FSM of a multi-cycle core

The design of this FSM allows the processor to transition through any instruction of RV32I ISA, while remaining deterministic, complete and efficient.

2.3 WAL

SonicRV's visualization system is strongly dependent on the waveform of a given simulated program. The waveform is generated by GHDL that compiles the hardware design sources of a given processor into machine code, executes it, and outputs a VCD [2]. The VCD typically contains not only every signal change with its corresponding time stamp, but also the hierarchical structure of modules that make up the hardware design [10]. Therefore, SonicRV utilizes the *Waveform Analysis Language* (WAL) programming extract relevant data from the VCD after the simulation.

WAL is a *Domain Specific Language* (DSL) with a Lisp-like syntax, offering a tool for automated waveform analysis [7, 8, 9, 10]. Using this DSL, separate programs can be implemented for any type of processor to filter out relevant signal changes and export the results. WAL can also be imported as a python module and therefore WAL programs may be directly invoked from python.

Utilizing WAL enables the backend of SonicRV to automatically and efficiently filter and store only relevant data in a common format. Furthermore, it serves as a crucial component for integrating different processor architectures, as it facilitates adaptation to the differing data structures of the cores.

Having covered all prerequisites for this thesis, the detailed integration of the multi-cycle processor will be addressed in the following chapter.

Chapter 3

Integration of a Multi-Cycle Core in SonicRV

This chapter will explain all the implementation steps necessary for integrating an existing RV32I multi-cycle core into SonicRV. Before this work, SonicRV supported only single-cycle and pipelined cores. This thesis integrates the missing piece, namely a multi-cycle core into SonicRV. In this thesis, two cores were actually integrated:

1. *ics-edu-rv32i-mc*: A 32 bit multi-cycle core for easy understanding, based on [5]. It lacks *U-* and *J-type* instructions and implements only the *beq* instruction out of all the *B-type* instructions. It does not, and should not reveal every aspect of a fully working RV32I core.
2. *ics-nex-rv32i-mc*: A 32 bit multi-cycle core implementing all RV32I instructions, offering all details of the implementation and hardware design. The VHDL implementation has been developed in [14].

For the sake of brevity, only the integration of the *ics-nex-rv32i-mc* core is described, since it is generally more complex and also covers the main aspects of the integration of the *ics-edu-rv32i-mc* core. The following sections describe the process of the integration in a chronological implementation order, for both, backend and frontend, where each section covers one or more *subsystems* of Figure 3.1. All *subsystems* are indicated with numbers from 1 to 6. Those numbers also represent the order of execution when a program is simulated. However, before we start, some constraints shall be briefly described.

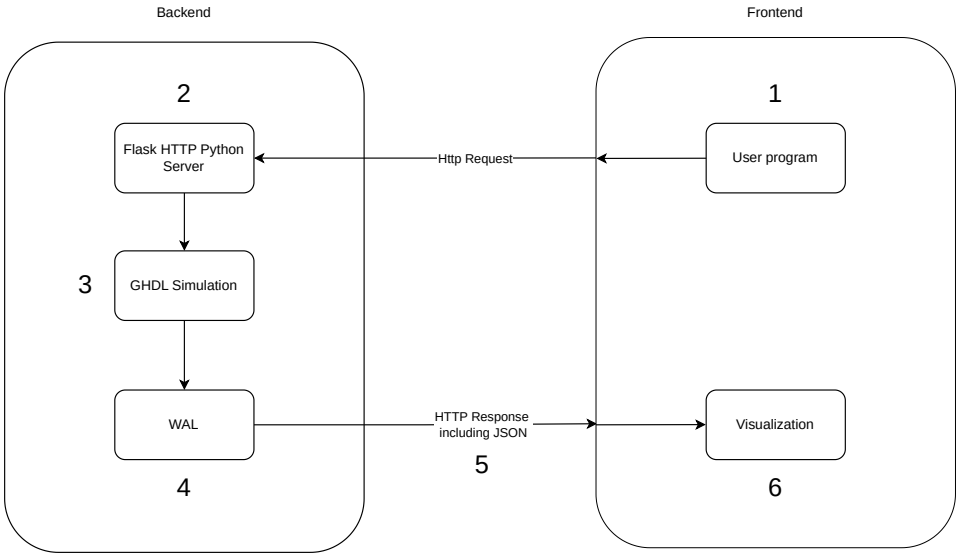

Figure 3.1: Simplified architecture of SonicRV's simulation/visualization process

3.1 Constraints

As with any established project, the introduction of new functionality is subject to inherent constraints. In the case of SonicRV, the integration of an additional core, possessing distinct characteristics and operational behavior, must be accomplished without altering the existing structure, functionality, or visual representations of the other cores.

During the integration of the *ics-nex-rv32i-mc* core, certain components had to be necessarily modified and extended. Nonetheless, any modification to the backend API or to frontend data structures shared across cores requires careful consideration. When such modifications are unavoidable, corresponding adjustments must be implemented in the remaining cores to ensure their functionality remains consistent with the state before the modifications.

3.2 Backend Integration

This section focuses on the backend part of SonicRV (left part in Figure 3.1). First of all the necessary backend file hierarchy is explained, whereas the remaining of the section focuses on simulation data retrieval, refinement and structuring.

3.2.1 File hierarchy

To integrate a new core into SonicRV, adherence to a predefined file structure is required in order for the backend to correctly identify and incorporate the core. For the provision of all necessary information and fully functional simulation and visualization, this structure must include at minimum:

1. a description file of the core (info.json), which includes information such as the core name, a description, supported instructions, etc...
2. an SVG image of the core (<image>.svg), resembling the base image for dynamic visualizations
3. the VHDL sources for the hardware design (simulation/src/*.vhd). These must be compilable with GHDL. Additionally, support of loading a RV32I binary to an arbitrary memory address as well as loading register file contents is necessary
4. a WAL script that analyzes and exports the relevant signals (main.wal)
5. example programs in RISC-V assembly (examples/*.s)
6. a simulation script that simulates a program on the core (sim.sh)
7. files for the initial memory and register file contents (files/*.txt)
8. a configuration file on where the program should be loaded into RAM (config.env)
9. files for hover, that show up in the frontend when a VHDL entity is selected (hover/*.vhd)
10. a file for the initial state of Surfer (riscv.ron)

These files are required so that the backend server loads the core and enables it to be picked by the user in the web application. In the course of this chapter, the purpose and contents as well as modification and extension of these files are shown and explained.

As a reference for the structure of the *ics-nex-rv32i-mc* core, the file hierarchy for this core is shown in Figure 3.2, with the numbers in parenthesis indicating the files listed in the previous enumeration.

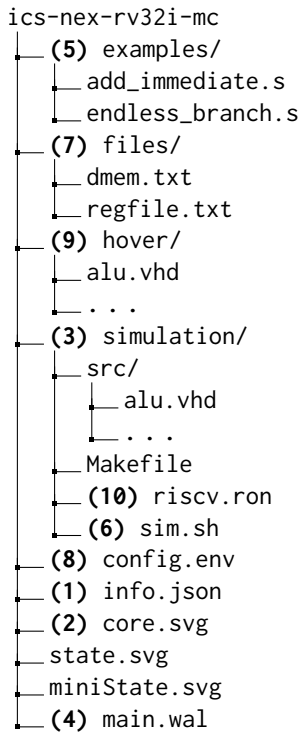

Figure 3.2: File hierarchy of the *ics-nex-rv32i-mc* core.

3.2.2 VHDL Adaptions

This section covers *subsystem 3* in the backend of Figure 3.1. Here the focus lies on the default GHDL signal data exposure and the necessary VHDL adaptions.

In SonicRV, solely a correct hardware specification of a core does not suffice and requires adaptions to function, since the default GHDL VCD export lacks required information. Thus, to integrate and visualize the *ics-nex-rv32i-mc* core, the existing VHDL source from [14] had to be extended to achieve the following requirements, to export all necessary information and data to the VCD.

Requirement 1: Relevant signals of the processor need to be exported to the waveform file. However, by default, GHDL only exports some primitive types to the waveform file, but no arrays of any type [2], which are commonly used for data structures like memory cells or register files. In this particular case, the register file, memory and FSM needed additional signal exposure. To solve this, primitive dummy signals for every array cell are introduced in every entity of interest. These dummy signals carry the same data type as the corresponding array, and are re-assigned at every clock cycle, to hold the same data as the array. In this way, array data is mapped to primitive signals and GHDL is now able to export these dummy signals.

Listing 3.1 shows the concrete solution of exporting a VHDL array structure to the VCD. It shows the behavioral architecture of the *RegFile* (Register File) entity in the *ics-nex-rv32i-mc* core, responsible for the storage and signal output of 32 registers. Internally, the *RegFile* uses a custom type *ramtype*, defined as an array of 32 word sized vectors (Line 2). This array is used in the sequential logic in Line 10 to store and output register values. We use an array here, because it allows for a generic process implementation, which is not shown here.

However, as array types are not exported to the VCD by GHDL, 32 dummy signals are introduced in Lines 5 to 8, which are of the primitive type *std_logic_vector*. Those can

then be assigned to the corresponding value stored in the *ramtype* array, see Lines 15 to 20, whenever changed. In this way, every register value change is exported to the VCD by GHDL.

```
1  architecture behave of RegFile is
2    type ramtype is array (31 downto 0) of std_logic_vector(31 downto 0);
3
4    -- dummy signals
5    signal  x0,  x1,  x2,  x3,  x4,  x5, x6, x7,
6            x8,  x9, x10, x11, x12, x13, x14, x15,
7           x16, x17, x18, x19, x20, x21, x22, x23,
8           x24, x25, x26, x27, x28, x29, x30, x31: std_logic_vector(31 downto
      0);
9  begin
10   process(clk) begin
11       -- register file logic
12   end process;
13
14   -- combinatorial assignment of dummy signals
15    x0 <= (31 downto 0 => '0');
16    x1 <= (reg(1));
17    x2 <= (reg(2));
18    ...
19    x30 <= (reg(30));
20    x31 <= (reg(31));
21
22 end;
```

Listing 3.1: *Register File* hardware design and exporting arrays to the VCD

Requirement 2: State Signals of the FSM need to be generated and exported. This is needed to visualize dynamic state changes in the dynamic state machine visualization in the frontend. The FSM internally works with an internal register that holds the current state, defined as an *enum* type, consisting of 15 individual states, see Section 2.2. Since it is an *enum* type, again GHDL will not export this state signal into the waveform file. As in requirement 1, this is again solved with primitive dummy signals. A 4-bit dummy vector signal (that yields 16 possible states) is created and combinatorially assigned with the position of the current state in the enum. In this way, the current state for every clock cycle can be exported to the resulting waveform file, the VCD.

Requirement 3: Functionality for pre-loading the *Register File* and data-memory with arbitrary hexadecimal data from a text file at startup is required for educational purposes. Especially the *ics-edu-rv32i-mc* core benefits from this feature. To achieve this behavior, global GHDL string arguments where added to the VHDL design at the top level. These strings describe the paths of the files, where the static pre-loading content is located, see the file-structure in Section 3.2.1. The path information is propagated down to the desired entity, where a pre-loading function is implemented and called before the simulation to populate the memory cells and registers with the desired values.

Requirement 4: Require loading and running programs from arbitrary memory addresses. This requirement aims to establish a common memory layout, imitating the layout of existing cores. In SonicRV, per default, 16 kilobytes (or 4096 words in RV32I) are used as data section for every core. Code section starts at address 0x1000 (4096 word addresses). This is solved by loading the binary file consisting of the RV32I user program to the static address of 0x1000. Before the simulation starts, also the *PC* has to be assigned to this address.

With these requirements implemented, we have a multi-cycle hardware specification that is able to export all necessary information and data via GHDL to a VCD.

3.2.3 Waveform Analysis of a Multi-cycle Core

The fundamental prerequisite for achieving a clear and accurate visualization of the core and its states of the FSM is a well-structured, non-redundant data source. This section covers the process by which the resulting VCD from the GHDL hardware simulation is transformed into usable input for the frontend visualization and is located in *subsystem 4* in Figure 3.1.

After the modifications explained in Section 3.2.2, the VCD, generated during the GHDL simulation of the user program, contains all relevant information regarding the signals exposed in the VHDL description of the multi-cycle core. However, not all of the information contained in the VCD is required for the visualization, and thus would lead to a large object with irrelevant data to transfer to the client.

To extract the necessary subset of information, WAL is used. WAL is capable of filtering the contents of the VCD programmatically. Existing cores already use WAL programs to retrieve and filter signals of all VHDL entities, as well as *Register File* and memory contents. Additionally, instruction-level information, such as the instruction name and operands are also already extracted from the waveform by WAL.

```
(defmacro stage args
  (define name (first args))
  (define info '())
  ...

  (defmacro log [logid expr] `(logUnquoted ',logid ',expr))
  (defun logUnquoted [logid expr] (set! info (append info `(',logid ,expr)))
    )

  (when (> (length args) 1)
    (for [arg (rest args)]
         (eval arg)))
   ...)
```

Listing 3.2: Parts of the *stage* macro

As an example, Listing 3.2 shows parts of the existing WAL macro called *stage*[1], that can also be used for the integration of the multi-cycle core, to filter out signals. The *stage* macro defines a named analysis stage and collects signal filter expressions into a shared list. The first argument is the stage name, the remaining arguments are forms executed inside the stage context to configure VCD signal logging. An empty list *info* is created to accumulate signal entries for later processing.

Inside the stage, a helper macro *log* is introduced. It takes an identifier and a signal expression and expands to a call to *logUnquoted*. The *logUnquoted* function records the pair by appending it to the *info* list. This allows any call to *log* inside the *stage* macro body to register a signal without dealing with quoting or list construction. After defining these helpers, all arguments after the stage name are evaluated. This means any logging calls or other filter expressions in the stage body are executed immediately, populating *info* for the current stage.

```
(stage multi
       (log adr testbench.rvmulti.dp.adrmux.y)

       (for [i (range 0 32)]
            (logUnquoted (symbol-add 'x i)
                (symbol-add 'testbench.rvmulti.dp.registerfile.x i))))
```

[1]This macro bears the name "stage", because of the pipelined cores. The macro is designed to be called multiple times with different names for the different pipeline stages. In the case of the multi-cycle processor, the *stage* macro is only defined once, because we only have one stage.

Listing 3.3: Usage of the *stage* macro

Listing 3.3 shows an example usage of the *stage* macro of the multi-cycle core. The call *(stage multi ...)* creates a stage named "multi". Inside the body, a signal "*adr*" is logged directly using the *log* macro (Line 2). Here we define the signal *testbench.rvmulti.dp.adrmux.y* (GHDL waveform path) to be logged to the name *adr*. A loop then logs all 32 register signals (Line 4). It iterates over the register indices xi of the GHDL waveform path and uses *symbol-add* to generate both the log IDs and the corresponding GHDL signal names, calling logUnquoted for each entry.

The usage of this existing WAL macro *stage*, enables us to filter and log all relevant signals, registers, memory content, etc. of the multi-cycle core, by filtering all desired signals in the *stage* macro. However, since the multi-cycle core has a unique component, the FSM, a novel WAL program is required to filter and log the signals of the FSM. Remember, the FSM keeps track of the current step when executing an instruction. An instruction is broken down into multiple steps, which are performed in consecutive clock-cycles. To analyze the different transitions through the FSM, a WAL algorithm is designed, and further explained in the following section.

3.2.4 Waveform Analysis of an FSM

This section also covers the *subsystem 4* in Figure 3.1 and focuses solely on the waveform analysis of the FSM, rather than on the general waveform analysis in the previous section.

To dynamically highlight an FSM state diagram in the frontend it is necessary to have information about all states within each instruction. We will later see that this can be easily extracted using WAL. Steps in the FSM shall further be called states, since each step within an instruction, represents a unique state in the executed instruction, also represented by the FSM.

The *ics-nex-rv32i-mc* core, for example, takes a minimum of 3 and a maximum of 5 states to execute a given instruction, depending on the specific instruction. For example the *beq* instruction is completed within 3 consecutive clock cycles and therefore has 3 states S0 - Fetch, S1 - Decode and S10 - CONDJ, while the *lw* instruction runs through 5 consecutive clock cycles and 5 micro-steps S0 - Fetch, S1 - Decode, S2 - MemAdr, S3 - MemRead and S4 - MemWB, see Figure 2.6.

For a clear representation and visualization of the instruction table[2] and state diagram in the frontend, an entry for every state shall be extracted. This guarantees a complete view of the simulation results.

For every state, also a human readable state name is required. To achieve this, a WAL macro, named *fsm*, is responsible for determining the state number of the FSM waveform signal and map it to a short and long state name, corresponding to the state diagram in Figure 2.6.

```
1 (defmacro stage args
2   ...
3   (defmacro fsm expr
4     (define name expr[0])
5     (define signal expr[1])
6     (define mappings (for/list [id (rest (rest expr))] (list id[0] `(list ,
      id[1] ,id[2]))))
7     `(logUnquoted ',name
8       `(case ,signal
9          ,@mappings)))
```

[2]Table in the frontend with columns for PC address, instruction and state information

```
  ...
)
```

Listing 3.4: Implementation of the fsm macro to map state names with raw state numbers

Listing 3.4 shows the implementation of the *fsm* macro, that converts state numbers into state names from a defined state-number to name map. Note that this macro is defined within the *stage* macro, already shown in Listing 3.2. Firstly, the *fsm* macro is defined and takes an argument consisting of a name, a signal expression, and a list of state rows, where each row provides a state number, a short name and a long name. The first two elements of the input form are bound locally as name and signal. The remaining elements are then processed into mappings where each row is transformed into a pair whose first element is the state number, and whose second element is a quoted list of the short and long names.

The macro then expands into a *logUnquoted* call, which was already defined in Listing 3.2, that contains a case expression on the provided signal. All the entries from the mapping are expanded into a switch case statement, producing one branch per given state. At expansion time, this generates a lookup that maps each state number to its corresponding short and long name while keeping the structure explicit and adaptable in the code.

```
(stage multi
  ...
  (fsm fsm_state testbench.rvmulti.cu.fsm.current_state_view
     [0 "S0" "Fetch"]
     [1 "S1" "Decode"]
     [2 "S2" "MemAdr"]
     [3 "S3" "MemRead"]
     ;; More states ...
     )
   ...
)
```

Listing 3.5: Usage of the WAL function that maps state names with the current exported state number in the waveform

Listing 3.5 shows an explanatory snippet of the usage of the *fsm* macro. The *expr* argument must consist of a name, that should be exported (logged) to the frontend, see Section 3.2.5, in this case "fsm_state", the corresponding signal that holds the state number, here "testbench.rvmulti.cu.fsm.current_state_view", and some rows with three columns, representing the state id, a short name and a long name. Note that arbitrarily many state mappings can be defined, enabling us to reuse the *fsm* macro for different multi-cycle processors with different FSMs.

With the use of this macro, FSM state signals of the simulation can now be mapped to short and long names. This means that we are finally able to extract all relevant data that we need in the frontend.

3.2.5 Adaption of Data Structures for Communication

This section covers the last missing piece of the backend integration, the communication part to the frontend. With a quick reference to Figure 3.1, this is *subsystem 5* and focuses on the general data format as well as the data exchange and communication from backend to frontend.

The latter is made possible through an existing API in SonicRV. This API is responsible for transmitting all the necessary information and generated data structures from the

backend to the frontend. This includes, for example, core and state diagram images, signal changes between instructions or program examples.

One of the main data structures is a JSON object that includes all the current signal values for every instruction for the frontend visualization. This data structure is generated by the WAL program of Section 3.2.3, that analyses all exported signals, after the simulation of the user program.

```
{
  "stages": {
    "single": [
      {
        "instruction": "addi x1, x0, 5",
        "kind": "i",
        "memory": { "8192": 10 },
        "pc": 4096,
        "signals": {
          "pc": "4096",
          ...
        },
        "realtime": 25000000
      },
      {
        "instruction": "andi x1, x1, 255",
        "kind": "i",
        "memory": { "8192": 10 },
        "pc": 4100,
        "signals": {
          "pc": "4100",
          ...
        },
        "realtime": 35000000
      }
    ]
  }
}
```

Listing 3.6: JSON object for simulated instructions and their signals

Listing 3.6 shows the basic structure of the common JSON object of an existing single-cycle core. This object provides information for different stages at the root level of the object. This state information is only used for the pipelined cores, and will be just a single stage for the single-cycle and multi-cycle cores. Right beneath is an array with all the simulated instructions, where each instruction gets its own object in the array. Every object includes key value pairs with information about the exact RV32I assembly instruction and the instruction kind (see Section 2.2.1), the current memory contents after the instruction was executed, the current *PC*, an array with all the signal values for the current instruction, and the simulation time of the instruction in nanoseconds.

For the integration of the multi-cycle core, the overall structure of this JSON object has to stay the same to avoid complex checks for the current core and structure of the JSON in the frontend at runtime. Information for all states per instruction is additionally generated by a multi-cycle WAL program, see Section 3.2.4, and added to the JSON, leaving the general structure unmodified. In the frontend, only relevant information of the JSON object for the chosen core is taken into account for visualization. This renders runtime checks for the correct data simple and adaptable.

When considering a similar JSON object for the *ics-nex-rv32i-mc* core the following data has to be at least included:

- Information for every extracted state from the FSM

- Signal changes for each state
- A human readable name for each state

With the help of the waveform analysis, explained in Section 3.2.3 and Section 3.2.4, we already obtained all the required data. The remaining task is to store this data in a similar JSON structure as the single-cycle and pipelined processor already use.

For this, the idea is to store individual consecutive states as array objects, rather than whole instructions like the single-cycle JSON object, as in Listing 3.6. As an example of how this will look like, Listing 3.7 shows the adapted JSON object for the *ics-nex-rv32i-mc* core. Here, every object of the *multi* array represents an individual state, rather than a whole instruction, keeping the structure of the object the same. Additionally, each state object stores data of its corresponding short and long name, which we generated in Section 3.2.4 with WAL. For instance, in Listing 3.7, the first array object is the *S0 - Fetch* state of the *addi x1, x0, 5* instruction, whereas the second entry represents the *S1 - Decode* state of the exact same instruction. In this way, the JSON structure stays the same, while having signal change information for each individual state.

```
{
  "stages": {
    "multi": [
      {
        "instruction": "addi x1, x0, 5",
        "memory": { "8192": 10 },
        "pc": 4096,
        "signals": { ... },
        "fsm_state": [ "S0", "Fetch" ]
      },
      {
        "instruction": "addi x1, x0, 5",
        "memory": { "8192": 10 },
        "pc": 4096,
        "signals": { ... },
        "fsm_state": [ "S1", "Decode" ]
      },
      ...
      {
        "instruction": "andi x1, x1, 255",
        "memory": { "8192": 10 },
        "pc": 4100,
        "kind": "i",
        "fsm_state": [ "S0", "Fetch" ]
      },
      {
        "instruction": "andi x1, x1, 255",
        "memory": { "8192": 10 },
        "pc": 4100,
        "signals": {
        "fsm_state": [ "S1", "Decode" ]
      },
      ...
    ]
  }
}
```

Listing 3.7: JSON object including new multi-cycle specific structure and data

With modifications to the main JSON object, the backbone for the multi-cycle integration is now built, allowing us to use all necessary information required for a complete visualization in the frontend.

3.3 Frontend Integration

The goal for the frontend part was to provide a visualization of the *ics-nex-rv32i-mc* core, that blends in with the existing cores in SonicRV, reusing as much code as possible to avoid adapting the other cores. However, still a huge portion of the work ran into the correct design and highlighting mechanisms of the SVG images for the core itself and the corresponding state machine. Mechanisms for state machine highlighting and the simultaneous visualization of the core and state image had to implemented from scratch.

Additionally, the instruction table was adapted to dynamically highlight instructions for a multi-cycle core in a clean manner, introducing a new column for the current state name a block highlighting feature to better recognize the current selected instruction.

The following subsections cover the process of the frontend integration in a chronological manner, starting with SVG highlighting and design, and after that, the layout and instruction table adaptions. As another reference to Figure 3.1, in this section, we mainly cover the right side and there especially *subsystem 6*, the visualization.

3.3.1 SVG Highlighting

Before designing the SVG for the core and the FSM, the existing dynamic SVG highlighting mechanisms shall be discussed to better understand later decisions in the SVG design. SonicRV already makes use of dynamic SVG highlighting for the single-cycle and pipeline cores, by coloring signals and values of interest, when a specific instruction (and pipeline stage, for the pipelined core) from the instruction table is selected.

The file format of an SVG is an XML structure, providing a hierarchy of objects e.g. lines, rectangles or text, with CSS style information [1]. This style information is used to render the objects e.g. in a specific color, thickness or opacity.

Because XML is natively parsable into a *Document Object Model* (DOM) tree and modifiable by JavaScript in all established browsers [4], existing data, like CSS styles or text contents in the loaded SVG can be added, deleted or modified. This characteristic can be used, to add, modify or remove style information with the desired loaded and parsed SVG file, based on the currently selected instruction in the instruction table.

For SonicRV, two use cases, regarding SVG highlighting can be found, which are described in the remaining section.

SVG Value Updates

Dynamic SVG values are used to show current numerical signal values for data paths and entity in- and outputs and are bound to the processor's signals of the simulation. Those can be seen in Figure 2.1 in the main SVG in the center, marked in green.

In order to specify dynamic SVG values, text elements in the SVG have to be defined, wherever needed. Those text elements have to be annotated with additional XML attributes to bind the element to a signal.

For instance, Listing 3.8 shows the dynamic signal value element of the *AluResult* path of the *ics-nex-rv32i-mc* core in the main SVG. The *aluresult* signal is bound to this element by specifying an *id* attribute in the form of *id="cpu-signal-<signal name>"*. Here the same signal names as of the WAL analysis in Section 3.2.3 have to be used. Additionally, a data format may be specified. In this case, a hexadecimal representation is chosen, specified by the attribute *data-number-format="hex"*, although also *decimal* and *binary* representations are supported. Lastly a placeholder can be specified, which in SonicRV is usually just an underscore character. This placeholder will be displayed when no signal data is received, otherwise it is overwritten with the actual signal value in the specified representation.

```
<text
    id="cpu-signal-aluresult"
    class="cpu-signal"
    data-number-format="hex">
    _ <!-- Placeholder -->
</text>
```

Listing 3.8: Example of a processor signal binding in the multi-cycle core SVG

SVG Path Highlighting

SVG path highlighting is used to emphasize entities and connecting data paths of interest. For instance, in Figure 2.1 path highlighting of a single-cycle *add* instruction is highlighted in orange. Desired paths and entities are highlighted to point out the data flow between hardware components.

To configure SVG path highlighting for the *ics-nex-rv32i-mc* core, we have to apply specific style information to the desired signals and values in the core SVG, attributes within each contained object in the SVG file have to be statically defined in the attributes of the individual SVG elements. This is done by specifying the attribute *data-highlight* in each element, or group of elements. This attribute can be supplied with a condition. This condition should evaluate to true when the element should be highlighted.

A parser in the frontend is invoked for each of the specified conditions in the *data-highlight* attribute. The parser evaluates each condition whenever a different row in the instruction table is selected. Through this small DSL, it is possible to highlight SVG elements based on conditions. This DSL is capable of evaluating following types of expressions:

- Boolean operators: AND (*), OR (+), NOT (~)
- RV32I instruction type comparison with the current instruction
- Signal value comparison
- State name comparison

The usage of such dynamic highlighting conditions is best described by a small example, shown in Listing 3.9. The example shows a dynamic SVG highlighting condition for the *ALU Result* path of the *ics-nex-rv32i-mc* core. The attribute *data-highlighting* provides a condition, on when the path should be highlighted. The value of the *data-highlight* attribute, evaluates to true when the *pcwrite* or the *regwrite* are set to a logical 1, or the *adrsrc* signal is a logical 1 and the core is not in State 4.

```
...
<g
  id="ALUResult"
  data-highlight="pcwrite=1 + regwrite=1 + (adrsrc=1 * ~S4)">
  <path ... />
  <text ... >ALU Result</text >
  ...
</g>
...
```

Listing 3.9: Example of a statically defined highlighting condition in the multi-cycle core SVG.

3.3.2 SVG Design

In SonicRV, dynamic SVG highlighting is used to emphasize events of interest. This includes, for example, signal and state changes of the core. The way to define conditional highlights and value changes in SVG images, has been discussed in Section 3.3.1. With this information, required SVGs for the *ics-nex-rv32i-mc* core can be designed.

For the *ics-nex-rv32i-mc* core three SVGs are necessary to provide a clear user experience: one for the main processor's block diagram, one for the detailed state diagram and one minimal state diagram. The minimal state diagram should be simultaneously rendered besides the block diagram to keep track of the core and the states. Thus, following SVGs have been designed and are presented with their actual file names:

- **core.svg:** A block diagram of the core, which shows a high level view of all VHD components and paths in between them, see Figure 3.3.
- **state.svg:** A detailed image of the FSM of the core, showing all states with detailed information, such as the state name and important signal changes for each state, see Figure 3.4.
- **miniState.svg:** A simplified image of the FSM, showing solely state numbers and connections, see Figure 3.5.

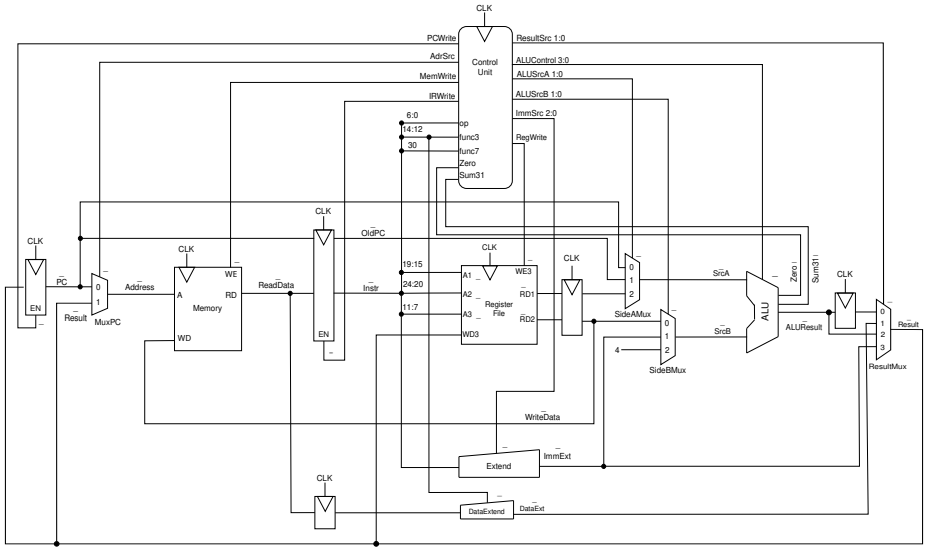

Figure 3.3: Multi-cycle core SVG

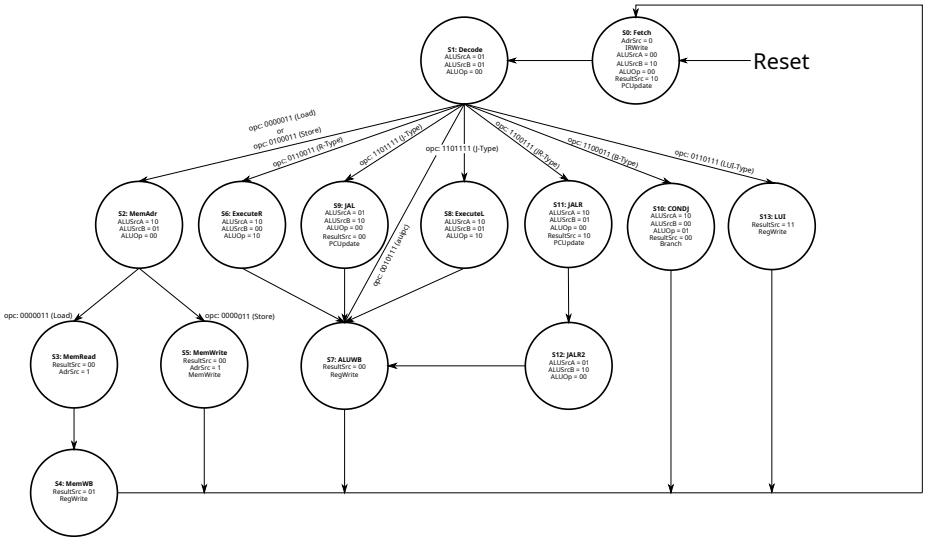

Figure 3.4: Multi-cycle state SVG

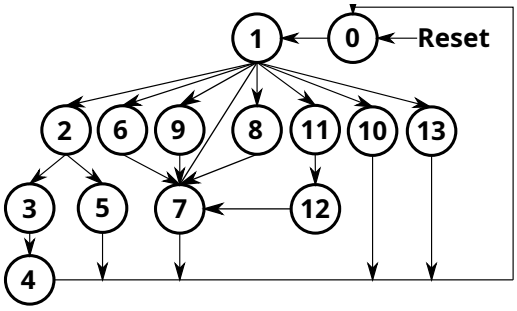

Figure 3.5: Multi-cycle miniState FSM SVG

The core and state SVGs had to be drawn from scratch, according to the actual core and state diagram layout, provided by [14]. The core image is also designed to comply to some design decisions of existing core images. For example, the overall layout of the core, component placement and font and stroke styles, are designed in a way, to provide a similar look and feel compared to the existing core

In the following, the design of the core and state SVGs is explained in more detail.

Dynamic Highlighting of a Multi-Cycle Core

When designing a dynamically highlightable SVG for the *ics-nex-rv32i-mc* core, a few aspects have to be addressed, in order to cover the main characteristics of a multi-cycle core.

First of all, the core image should have a high level of abstraction, making it easy to understand while covering all relevant technical aspects of the core for the user. This was achieved, by hiding technical details inside the main components and only provide in- and output signals of those to the user. For example, the FSM, which controls the state transition logic of the core at runtime, is hidden inside the *Control Unit* component in the SVG. The final abstracted core SVG contains components for various registers of different sizes, the *Control Unit*, *ALU*, *Register File*, memory and immediate- and data extension units. This layout provides a clear structure, but also all relevant components to the user, see Figure 3.3.

An additional aspect is to include every relevant signal value in the SVG, such as displaying corresponding signal values for every in- and output signal of any component. This is especially convenient for registers, that store data throughout one clock cycle, see registers with a *CLK* signal in Figure 3.3 This was solved, by adding dynamic signal values, as described in Section 3.3.2, to each in- and output signal of each component, where needed. Each individual dynamic signal shown in the SVG, needs a corresponding binding to a signal from the simulation by GHDL. For this, all used signals have to be added to the WAL program, exporting the signals to the frontend, see Section 3.2.3.

Furthermore, since we are dealing with a multi-cycle core, where instructions are executed throughout a varying amount of consecutive clock cycles, highlighting should not only be based on the current instruction type, but also the current FSM state and current signal values. By using the dynamic path-highlighting approach, explained in Section 3.3.1, this can be achieved by carefully going through each possible instruction type and state combination and applying conditional highlighting rules. For example, when the core executes a *jal* (*J-type*) instruction, and is in state *S7 - ALUWB*, highlighting has to be adapted, depending on the instruction type, current state and signal values. Given this information, the SVG is highlighted differently whether the jump is taken or not. For a visual representation of the core highlighting, refer to chapter 4.

Finally, to minimize the impact on the client's resources of the highlighting logic, the logic for dynamic path-highlighting was optimized to reduce the number of checks. For instance, when conditionally highlighting components, common patterns were identified and the highlighting logic was applied at a higher hierarchical level in the SVG elements. Additionally, large checks were logically simplified as much as possible.

Dynamic Highlighting of an FSM

For the dynamic highlighting of the *state* and *miniState* SVGs the same principles of simplicity and clarity as in the core SVG highlighting are of importance. However, the FSM highlighting solely depends on the state information, rather than instruction types and signal values, which keeps the overall highlighting logic generally simpler.

The implementation for the FSM highlighting consists of two requirements:

- **Requirement 1:** Highlight the current state of the core.
- **Requirement 2:** Highlight all previous states up the current selected state in a slightly different style than the current state, to allow backtracing states in the current instruction. For example, see Figure 3.4 and imagine we are in state *S7 - ALUWB*, all previous states until *S0 - Decode* should also be highlighted.

To implement those requirements, two new static CSS classes are introduced in the frontend, which are applied once the SVGs are loaded. One for the current state and one for the states in the backtracing path.

Using the dynamic SVG path-highlighting approach, explained in Section 3.3.1, renders the task of highlighting the current state quite trivial. Because conditional state name matching is already directly supported in SonicRV, providing a *data-highlight="S<state index>"*, for every state object in the SVG is sufficient to fulfill requirement 1.

However, for requirement 2, additional logic had to be implemented, as State 7 in the FSM, has more than one predecessor, see Figure 3.4. Given this aspect, there are multiple possible ways to reach State 7 from State 0, so we need additional logic to algorithmically backtrace the states to State 0.

To address this issue, a new SVG attribute was introduced: *data-highlight-previous="PS<state index>"*. This attribute is statically applied to each state object in the SVG, corresponding to the relevant index, similar to the previously introduced *data-highlight* attribute. The attribute was then made recognizable by the attribute parser. Now, whenever a new state is selected in the instruction table, a function in the frontend retrieves the backtrace from the current state to State 0. This is possible because the complete information of every state processed by the core during simulation is available through the waveform data structure, see Section 3.2.5. Using this data, the function stores all previous states in a list within the current instruction. The list is then iterated through, and each previous state, up to State 0, is highlighted accordingly. The solution of both requirements can now be applied to the state and miniState SVGs, as they have the exact same structure.

As the presented SVGs are now compliant with the highlighting system of SonicRV and the paths and signals are highlighted and updated regarding the selected instruction, the SVGs can finally be integrated. The actual integration of those in the layout of SonicRV is discussed in the next section.

3.3.3 SVG Integration and Layout Adaptions

This section covers the integration of all the designed SVGs in the previous sections into SonicRV and the resulting changes to the layout in the *Processor* tab of the simulator, see Figure 2.2.

Before this work, the layout of the *Processor* tab included following components:

1. **Instruction Table:** On the left-hand side, the instruction table displays all instructions of a submitted program. Each row provides information about the currently selected instruction, its PC, and, for the pipelined core, also the pipeline stages. Users can select any row (or individual pipeline stage) to update the content of other components accordingly.
2. **Main SVG:** In the center, the SVG of the selected core, which dynamically highlights paths of interest and changes signal values based on the selected instruction in the instruction table.
3. **Sidebar**: On the right-hand side, an extra column displays additional information. This includes two separate tables for the current register and memory contents at the time of the selected instruction in the instruction table.

With the multi-cycle core all of these components needed adjustment, to fit to the core's unique characteristics. Specifically, adaptions of components 2 and 3 are described here, while the adaption of component 1 is described in Section 3.3.4.

The main adaption was providing support to render multiple dynamically highlighted SVGs simultaneously. Since the multi-cycle core added two new SVGs, the layout now had to support three SVGs instead. To integrate those into SonicRV's frontend following requirements were determined:

- **Switchable Main SVG:** A button should allow the user to toggle between the core SVG and state SVG. Theses images are referred to as the "main images".
- **Constantly Visible FSM image:** A smaller, less detailed version of the state SVG, the "miniState SVG" or "sidebar image", should always be visible. This image should help track the FSM when the core image is displayed of instead of the main state image.

To meet those requirements, several changes were necessary for both the backend and the frontend.

Backend Changes

In the backend, the actual names of the images to be loaded to the frontend have to be specified. This was accomplished by providing the SVG file names in an array within the *info.json* file, see Section 3.2.1.

Listing 3.10 shows the layout of the adapted *ics-nex-rv32i-mc info.json* file. An arbitrary amount of main images (*images* array) and sidebar images (*sidebarImages* array) can be specified. Prior to this change the arrays in the *info.json* file did not exist, as solely one SVG image (*core.svg*) is present for existing cores.

```
1  {
2    "name": "ICS-NEX-RV32I-MC",
3    ...
4    "images" : [
5        "core",
6        "state"
7    ],
8    "sidebarImages": [
9        "miniState"
10   ],
11   ...
12 }
```

Listing 3.10: *info.json* file with SVG image defintions

Altering the structure of the *info.json* for the *ics-nex-rv32i-mc* core, lead to also bring the *info.json* files of all other cores in SonicRV to the same structure. Otherwise frontend logic would be needed to differentiate the structure between multi-cycle core and other types.

In order to be able to load an arbitrary amout of SVG files in the frontend, the backend API was adapted to accept a file name as a query parameter, allowing it fetch a specific image by file name. The API call was changed from */api/image/<core>* to */api/image/<core>/<image>*, enabling us to specify a core name as well as an image name we want to receive.

Frontend Changes

Once all *info.json* files were brought to the same structure, the frontend must be adapted to this new file structure. The frontend is now required to retrieve the SVG image file names from the *info.json* file and fetch the images one by one using the adapted API from the previous section.

Listing 3.11 illustrates how all images associated with a selected core are loaded. When a core is selected in the frontend, the function *loadCoreInfo* (Line 1) is invoked with the core name as its parameter. This function fetches the corresponding *info.json* file via the backend API and parses its JSON contents.

Based on the parsed data, *loadCoreInfo* retrieves both the main images and the sidebar images by calling the helper function *fetchImages* (Lines 7 and 8). The function *fetchImages* (Line 15) iterates over the list of image identifiers and fetches each image individually using the previously introduced API endpoint */api/image/<core>/<image>* in Line 20. The fetched image data is stored in an array indexed by the image name (Line 16).

Finally, the populated image arrays are returned to *loadCoreInfo* (Line 28), where they are assembled into the resulting *CoreInfo* object and returned to the caller (Line 12). This *CoreInfo* object is stored in the context of each simulation session in the frontend and holds all necessary information, such as the core name and all related images.

```
1  async function loadCoreInfo(core: string): Promise<CoreInfo> {
2    const responseCoreInfo = await fetch('/api/info/' + core);
3    const coreInfoJson = await responseCoreInfo.json();
4
5    const data = {
6      ...
7      images: await fetchImages(coreInfoJson['images'], core),
8      sidebarImages: await fetchImages(coreInfoJson['sidebarImages'], core),
9      ...
10   };
11
12   return data;
13 }
14
15 async function fetchImages(images: any, core: string): Promise<Images> {
16   const fetchedImages: Images = {};
17
18   if (images) {
19     for (const image of images) {
20       const response = await fetch(`/api/image/${core}/${image}`);
21       if (!response.ok) {
22           // error handling
23       } else {
24         fetchedImages[image] = await response.text();
25       }
26     }
27   }
28   return fetchedImages;
29 }
```

Listing 3.11: TypeScript functions for fetching processor ressources

When a program is simulated and the user navigates to the *Processor* tab, the first image of the main image array is shown as the main image in the center. All provided sidebar images are shown in the sidebar component stacked below each other, beneath the memory content table. A button, which becomes available when there are multiple images provided by the core, allows the user to switch between all the main images. Pressing the button rotates the index pointing to the shown main image of the array by one, displaying the next image while hiding the other ones. This design allows for flexibility, enabling any number of main and sidebar images to be loaded for each core. Furthermore, this approach leaves room for future expansions, where other cores could also benefit from additional images.

After implementing those adaptions, SonicRV is now capable of rendering multiple SVGs, while simultaneously highlighting them. This was done in a way, to leave the overall layout the same, but also provide features to all the other cores, for potential future features. For a demonstration of this feature, refer to chapter 4.

3.3.4 Instruction Table Highlighting

Last but not least, the instruction table was adapted for the multi-cycle core.

The instruction table in SonicRV internally consists of a plain HTML table. It is highlighted by predefined CSS classes, dynamically applied to each independent cell. This dynamic styling depends principally on the column and the current selected row. Selected rows are additionally emphasized in the main color of the corresponding column, see Figure 3.6.

Generally, the instruction table holds similar data among all core types. However, the instruction table of the multi-cycle core holds rows of states, rather than whole instructions as of the single-cycle and pipelined cores. That means, that for every instruction, three to five row entries, depending on the specific instruction FSM depth, are generated. Without any adaptions, this design was unpleasant and hard to navigate through.

To solve this issues, two new features were added to the instruction table when selecting a multi-cycle core:

PC	multi	state name
1000	add x0, x1, x2	S0 - Fetch
1000	add x0, x1, x2	S1 - Decode
1000	add x0, x1, x2	S6 - ExecuteR
1000	add x0, x1, x2	S7 - ALUWB
1004	sub x0, x1, x2	S0 - Fetch
1004	sub x0, x1, x2	S1 - Decode
1004	sub x0, x1, x2	S6 - ExecuteR
1004	sub x0, x1, x2	S7 - ALUWB
1008	slt x0, x1, x2	S0 - Fetch

Figure 3.6: Multi-cycle instruction table

State Name Column

Firstly, an additional column for the state name has been added. This column shows the name and number of the of the state, that corresponds to the sub-instruction. Remember that each instruction is split into its states, where each of these states get a unique row in the instruction table. With the help of this extra column, the FSM state flow of each instruction can be inspected at a glance.

Block Highlighting

Secondly, block highlighting was introduced to visually help to keep track to which instruction the current selected state belongs. This block highlighting features, highlights the current selected state and PC, and emphasizes all other rows that corresponds to that instruction. To provide an example, Figure 3.6 shows the instruction table with the new state column and the current selected *sub* instruction in state *S1 - Decode*. Note that the whole *sub* instruction with PC 1004 from state *S0* to *S7* is also emphasized in the same color, but less opaque.

To accomplish the block highlighting feature, the already familiar dynamic CSS styling mechanism was extended. For this, a predefined CSS class and an additional function in the frontend instruction table logic was implemented, shown in Listing 3.12. This function decides whether a cell should be highlighted or not, i.e. if the cell is in the current block, it should be highlighted. It is called for all cells that are in range of the maximum FSM

depth, rather than all existing cells, to save computational resources. The FSM depth is calculated by another simple algorithm implemented in this thesis, which is not shown here.

```
1  function blockHighlight(
2    waveform: Waveform,
3    selection: StepSelection,
4    cell: StepSelection
5  ): boolean {
6    let selectedPC: string = getPC(waveform, selection.index);
7    let runner: number = cell.index;
8    let direction: number = Math.sign(selection.index - cell.index);
9
10   while (getPC(waveform, runner) == selectedPC) {
11     if (runner == selection.index) {
12       return true;
13     }
14
15     // Needed for the case of the PC always being the same e.g.:
16     // loop:
17     //     beq x0, x0, loop
18     if (
19       (direction > 0 && wfGetStateIndex(runner + 1, waveform) == 0) ||
20       (direction < 0 && wfGetStateIndex(runner, waveform) == 0)
21     ) {
22       return false;
23     }
24     runner += direction;
25   }
26   return false;
27 }
```

Listing 3.12: blockHighlighting function that decides whether a cell is within an instruction.

The *blockHighlight()* function accepts three parameters:

1. **waveform:** Holds data of all simulated and exported instructions by WAL of the current program.
2. **selection:** Holds the row index of the current selected instruction.
3. **cell:** The cell of interest we want to check if it should be highlighted.

The function returns a *boolean*, being true if the cell of parameter 3 should be highlighted, otherwise false.

The basic principle of this function is to start from the cell of interest and try to reach the currently selected cell in the instruction table without detecting a PC change. Usually neighboring instructions have different PCs. This means, that if the PC changes during the traversal of the instructions, the current cell under observation can not be in the same instruction, as the user-selected cell, and thus should not be emphasized.

This behavior is implemented by a running index and a direction. Firstly, we store the actual *PC* value of the selected cell in the instruction table (Line 6). Then in Line 7, the *runner* variable is assigned with the cell index of the cell we want to start reaching the selected cell. Thereafter in Line 8 the direction is calculated which is either -1 to search up, or 1 to search down in the instruction table.

For the actual algorithm a *while* loop checks if the cell at the index of *runner* still has the same *PC* value as the selected *PC*. After every iteration of the *while* loop the *runner* is incremented or decremented based on the *direction* (Line 24). If we eventually reach the

index of the selected cell from the cell of interest we return *true* in Line 11 (cell should be emphasized), otherwise we return false (cell should not be emphasized).

In corner cases, e.g. if we immediately jump to the same instruction again, the *PC* does not change between those instruction. For this case an extra check had to be implemented in Line 18. This condition checks, based on the direction, if we ran over the very first state (*S0 - Fetch*), i.e. we encountered a different instruction. In this case the cell should not be emphasized and we return false.

With this last step accomplished, the *ics-nex-rv32i-mc* core is now fully implemented and functional within SonicRV. For a demonstration of all the explained integrations, refer to the next chapter.

Chapter 4

Demonstration

This chapter covers the basic usage of SonicRV, accompanied by a small example program simulated on the newly integrated *ics-nex-rv32i-mc* core. The demonstration will showcase preconfiguration of memory and register contents, as well as the visualization of the core, the FSM and the instruction table.

4.1 User Interface Overview

The frontend in SonicRV provides three main tabs to guide the workflow from selecting and modifying example program, simulating it on a specific core type and finally visualize instructions, signals and the waveform:

The *Editor* tab provides functionality for selecting one of the cores and write a program for later simulation, see Figure 4.2. The *editor* also features RV32I syntax highlighting, for easier editing. By pressing the *Simulate now* button, the program is simulated on the selected core. The user may then download files for the raw assembly, the hex code in *ASCII* format and a VCD trace file.

The *Processor* tab, see Figure 4.5, provides an overview of the simulation, which is only available after a successful simulation within the *Editor* tab. On the left side an *Instruction Table* is displayed. This instruction table tabulates all simulated instructions and can be navigated through with the arrow keys on the keyboard. During navigation, the main block diagram in the center is constantly updated and highlighted according to the selected instruction in the *Instruction Table*. On the right side, tables with the current register and memory contents are displayed. Both are updated when another instruction in the *Instruction Table* is selected.

The *Waveform* tab, as depicted in Figure 4.4, renders the corresponding waveform of the simulation. Within this tab an embedded Surfer app will open, displaying the predefined Surfer signals in the backend's *riscv.ron* file, see Section 2.1.1. New signals can be added and viewed by navigating through the Surfer interface.

4.2 Example Program

Once we understood how the basic navigation in SonicRV works, a small example program shall be defined and briefly explained before demonstrating it in SonicRV. Note that the following program, as well as the core preconfigurations of the *ics-nex-rv32i-mc* core have been setup only for demonstration purposes, and are not part of production SonicRV. Preconfigurations are only used for the *ics-edu-rv32i-mc* core.

```
1 # Configured preconditions
2 #   x5 = 6
3 #   x9 = 1028 (0x404)
4 #   M[0x400] = 10
5
```

```
6  L6:
7      lw x6, -4(x9)    # x6 = M[x9 + (-4)]
8      sw x6, 8(x9)     # M[x9 + 8] = x6
9      or x4, x5, x6    # x4 = x5 OR x6
10     jal x4, L6
```

Listing 4.1: endless_jump example program of the *ics-nex-rv32i-mc* core.

Listing 4.1 shows the example program, we want to demonstrate, called *endless_jump*, which shall be explained briefly before we will see this program in SonicRV. The comments in the first line inform the user that this core is using preconfigured memory and register contents. In this case register *x5* has value 6, register *x9* has value 0x404 or 1028 in decimal, and memory address 0x400 has value 10, even before the first instruction is executed.

Firstly, the label *L6* is defined to which we will later jump back to. Then, the register *x6* is loaded with the preconfigured memory value 10 at address 0x400, using the *lw* instruction and relative addressing. Afterwards the content of register *x6* is stored back with the *sw* instruction to memory address 0x40c, again with the help of *x9* and relative addressing. Thereafter, the bitwise OR of register *x5* and *x6* is computed with the *or* instruction and the result stored in *x4*. Finally, we use the *jal* instruction to jump back to the previously defined label *L6* in Line 6 and store the address after the *jal* instruction in *x4*.

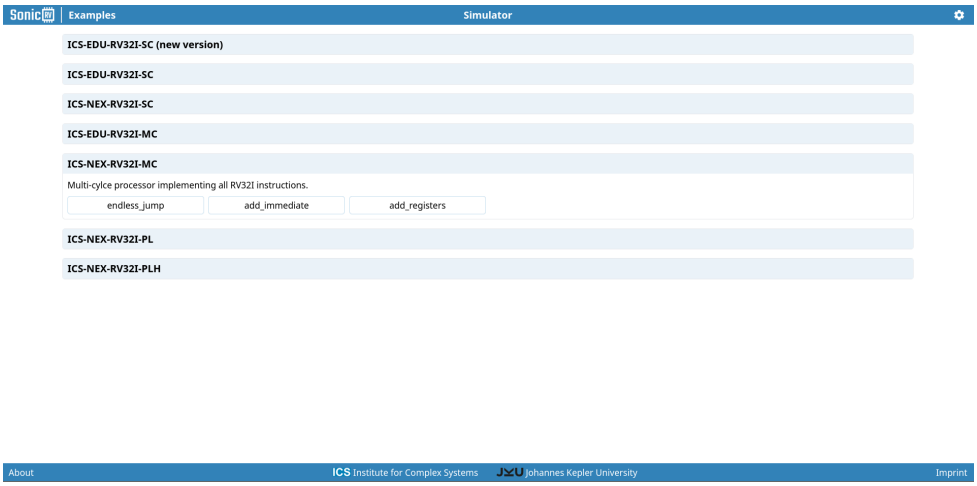

Figure 4.1: Example screen for choosing cores and example programs

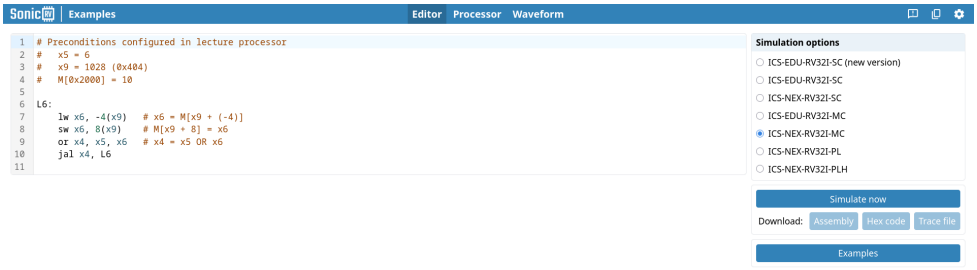

Figure 4.2: *Editor* tab with a loaded program

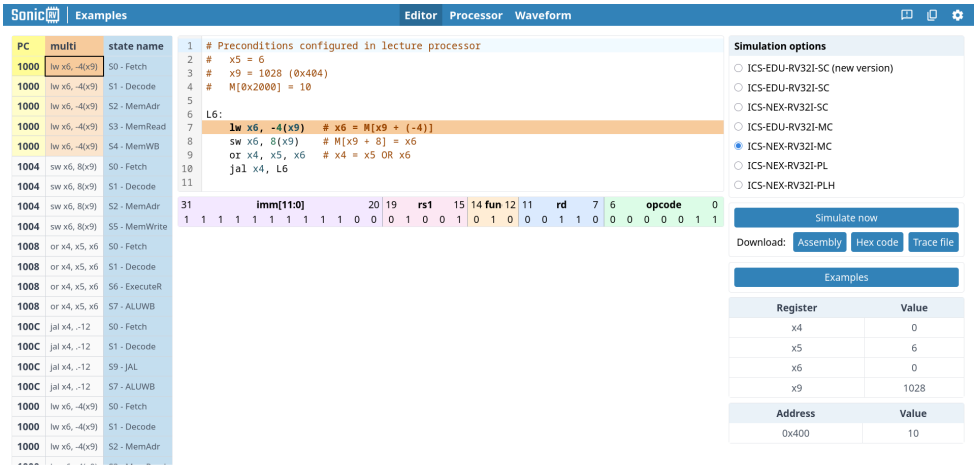

Figure 4.3: Simulated program in the editor tab

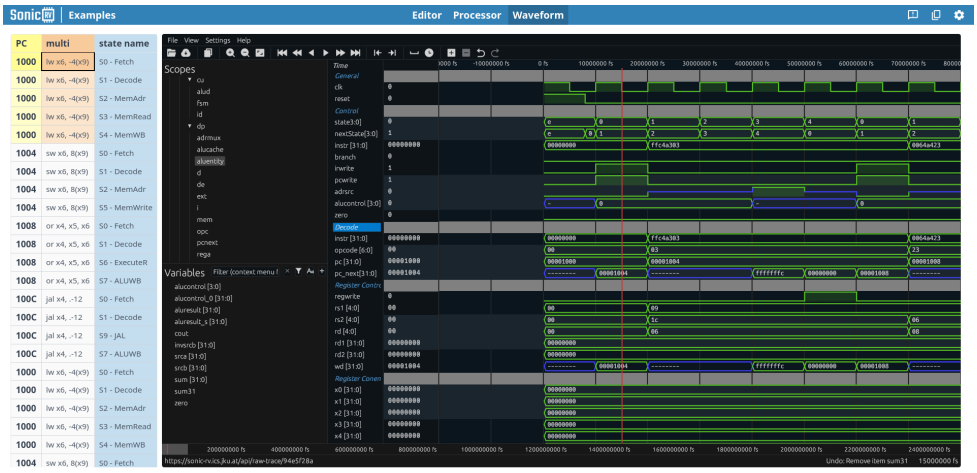

Figure 4.4: Waveform tab showing signals in Surfer

4.3 Example Program in SonicRV

After the test execution of the example program, we can now simulate and demonstrate it with the help of the integrated multi-cycle core and the extended features in SonicRV. In the following sections, the program is first loaded and simulated, then stepped through instruction by instruction while showing the most important highlighted signals and paths for each instruction and state.

4.3.1 Loading and Simulating the Example Program

The example program *endless_jump* is selected in the *Examples* screen of SonicRV in the *ics-nex-rv32i-mc* core, see Figure 4.1. Upon selection, SonicRV automatically navigates to the *Editor* tab, see Figure 4.2. The program is then simulated by the *Simulate now* button on the right.

After the simulation has successfully finished and the frontend received all information from the backend, the *Editor* switches from an editing state to a simulation state. Figure 4.3 shows the *Editor* tab in the simulation state with the instruction table on the left, source code and instruction formats in the center, and memory and register contents on the right.

The instruction table tabulates all executed steps from the multi-cycle core and the simulated *endless_jump* program with information about the *PC* (yellow column), instruction (orange column) and state name (blue column). The currently selected step is highlighted with black borders around the center table cell.

In the center of Figure 4.3, the source code of the simulated example program is displayed. Additionally, the current selected instruction of the instruction table is highlighted in the source code in orange. Furthermore, the RV32I binary instruction format, see Section 2.2.1 of the currently selected instruction in the instruction table is displayed beneath the source code. In this case the Figure shows the binary instruction for the selected instruction *lw x6, -4(x9)*.

On the right of Figure 4.3 register and memory contents are displayed in a table. This table represents the current memory and register state after the selected step in the instruction table was executed. Note that the registers and memory cells have already been loaded with the preconfigured values of the core.

We can also switch to the *Waveform* tab as shown in Figure 4.4, when a program has been simulated. It shows the actual waveform with predefined signals in the Surfer waveform viewer. GHDL signals can be added, modified or removed via Surfer's interface by the user. Additionally, Surfer will accordingly move to the time stamp of the selected cell in the instruction table. The current time stamp is indicated by the vertical red line.

Finally, in the *Processor* tab, the processor and FSM visualizations can be viewed, as depicted in Figure 4.5. Here, the main processor SVG is displayed in the center. In addition, also the instruction table, memory and register contents, as well as the *MiniState* SVG, below the register and memory table is rendered. All those components are dynamically updated according to the chosen cell in the instruction table. The main SVG view can be toggled to show the detailed FSM diagram instead, by pressing the blue button right above the main SVG.

The next sections will now cover the demonstration of the *endless_jump* program, step-by-step, where each Subsection corresponds to an instruction of the example program. For the actual demonstration we will only stay in the *Processor* tab.

4.3.2 *lw*-Instruction

The first instruction of the *endless_jump* program is *lw x6, -4(x9)*. Figure 4.5 illustrates the initial state, *S0 - Fetch*, with a *PC* value of `0x1000`, as indicated by both the in-

struction table and the *MiniState*. The instruction table highlights all states within the same instruction, shown in the first five rows, using the newly introduced block-highlight feature.

In the *S0 - Fetch* state, the behavior of the *lw* instruction is dynamically visualized in the central SVG. Key signals are highlighted, showing their values: The memory component is accessed at address 0x1000 (via the *Adress* signal), and the instruction at this address is loaded into a temporary register by the *ReadData* signal. Simultaneously, the *PC* is incremented by 4 in the *SideBMux* using the *ALU* and stored in a temporary *PC* register. The *MiniState* is updated to reflect the executed state.

Moving to the second state, *S2 - Decode*, the instruction is decoded. This is visualized in Figure 4.6. The instruction, loaded into the temporary register in the *Fetch* state, is decoded by the *Instr* signal, which is passed to the *Register File* component. Here, the source register *x9* (indicated by *A1*) and the destination register *x6* (indicated by *A3*) are selected. The value of *x9* is immediately loaded by the *Register File* and stored in a temporary register for the next state.

In the subsequent state, *S2 - MemAdr* (see Figure 4.7), the memory address is calculated by adding the value from register *x9* (via *SideAMux*) to the immediate value *-4* from the instruction (via *SideBMux*). This calculation results in 0x400, which is stored in a temporary register to be used in the next state.

The next state, *S3 - MemRead*, is shown in Figure 4.8. The *ALUResult* from the previous state is used to access the main memory at address 0x400. The value stored at this address, 10 (0xA), is retrieved from memory, propagated by the *ReadData* signal, and eventually stored in a temporary register for the following state.

Finally, in the last state, *S5 - MemWB* (Memory Writeback), as shown in Figure 4.9, the value 10 obtained from memory is written into register *x6*, as determined in the *Decode* state. However, the register table has not yet updated, since this only happens when the instruction is fully finished and the next one has started.

To explore the state diagram of the FSM, users can switch between the main SVG and the state SVG. The blue button above the SVG component allows toggling between these views. Figure 4.9 shows the detailed FSM state diagram, where the entire instruction's backtrace is visible, just as in the *MiniState* SVG. The state diagram illustrates the precise states traversed during the execution of the current instruction, by highlighting them. Specifically, the states visited include *S0 - Fetch*, *S1 - Decode*, *S2 - MemAdr*, *S3 - MemRead*, and *S4 - MemWB*, with the latter highlighted differently to indicate the current state of execution. For each state, the respective *Control Unit* signals are also displayed.

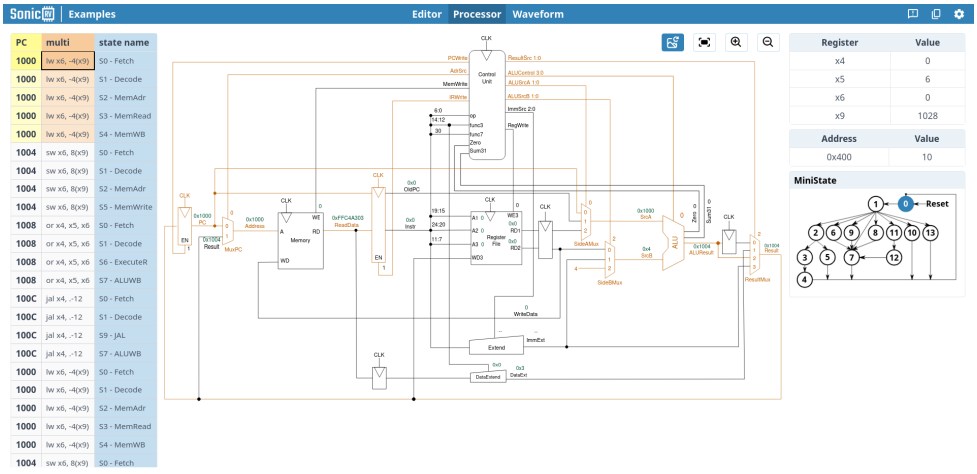

Figure 4.5: First cycle of the *lw* instruction

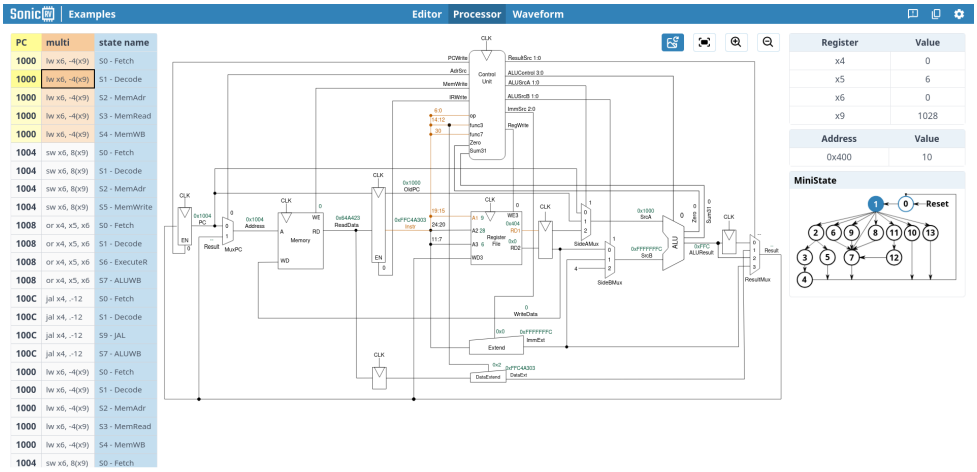

Figure 4.6: Second cycle of the *lw* instruction

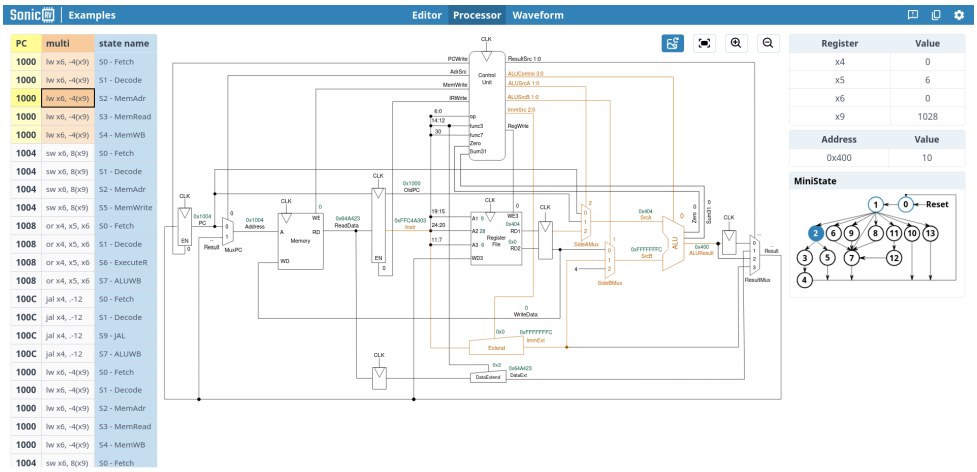

Figure 4.7: Third cycle of the *lw* instruction

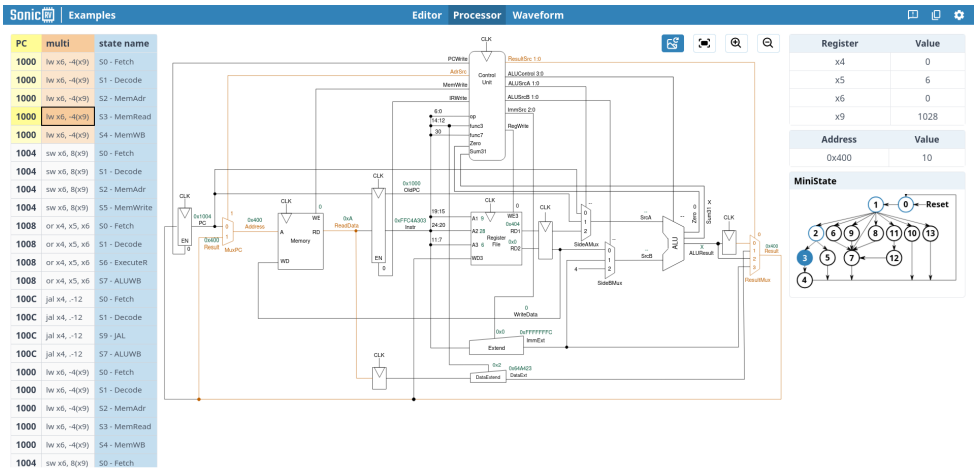

Figure 4.8: Fourth cycle of the *lw* instruction

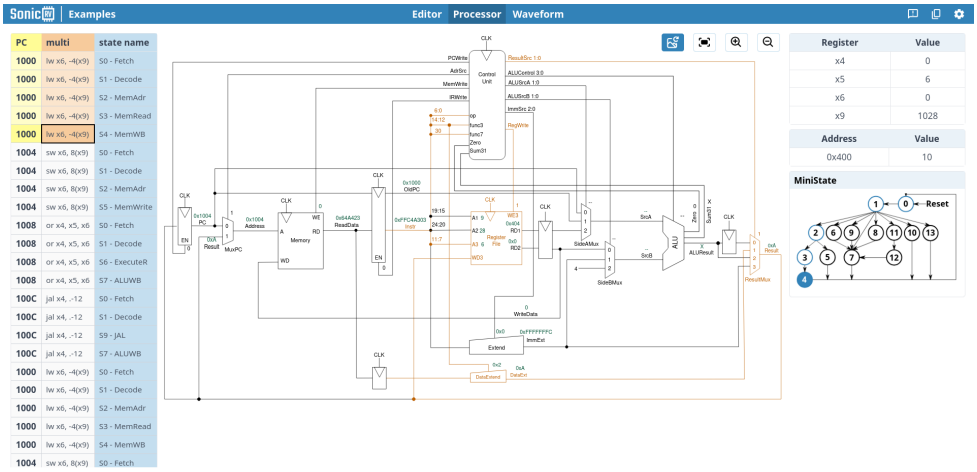

Figure 4.9: Last cycle of the *lw* instruction

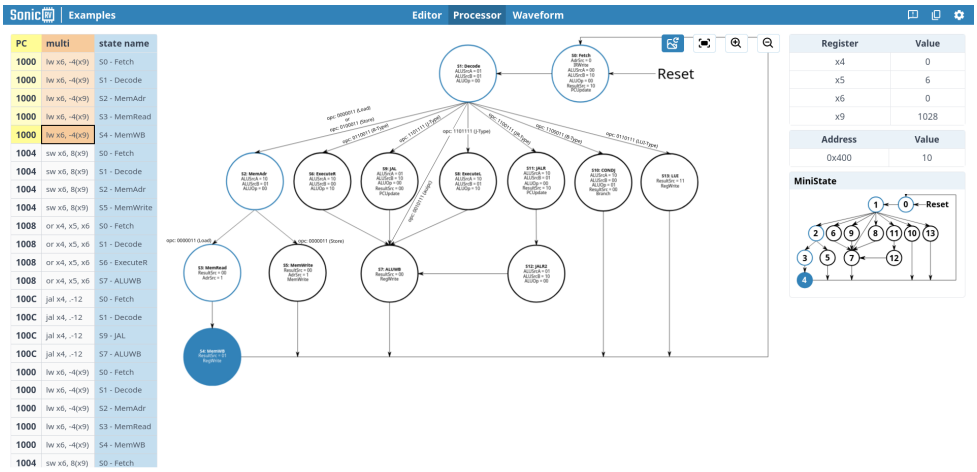

Figure 4.10: FSM view of the last cycle of the *lw* instruction

4.3.3 *sw*-Instruction

Moving on to the next instruction, *sw x6, 8(x9)*, the processor again enters the *S0 – Fetch* state, as shown in Figure 4.11. The register table now reflects the updated register contents produced by the preceding *lw* instruction. Since the *Fetch* state is identical for all instructions, the highlighted datapaths in this state are not discussed further.

The processor then transitions to the *S1 – Decode* state (see Figure 4.12). Unlike the *lw* instruction, the *sw* instruction specifies two source registers. Therefore also the dynamic SVG highlighting is different. Register *x9* (selected via *A1*) provides the base address, while register *x6* (selected via *A2*) provides the data to be stored. The value of *x9* (0x404) is loaded and stored in a temporary register for subsequent address computation.

Next, the *S2 – MemAdr* state is executed (see Figure 4.13). In this state, the effective memory address is calculated by adding the sign-extended immediate value to the previously stored contents of register *x9*. The resulting address is stored in a temporary register. The required datapath activity is controled by the control signals generated by the *Control Unit*.

Up to this point, the FSM follows the same execution path as for the *lw* instruction. However, in the final step the *sw* instruction diverges and enters the *S5 – MemWrite* state, as indicated in the *MiniState* diagram in Figure 4.14. In the main block diagram, the value of register *x6*, previously latched on the *WriteData* signal, is written to memory. The destination address 0x40C, computed in the preceding state, is forwarded via the *Result* and subsequently the *Address* signal to the memory address input.

After this state, execution of the *sw* instruction is complete. The updated memory contents become visible at the beginning of the next instruction, which is demonstrated in the following instruction.

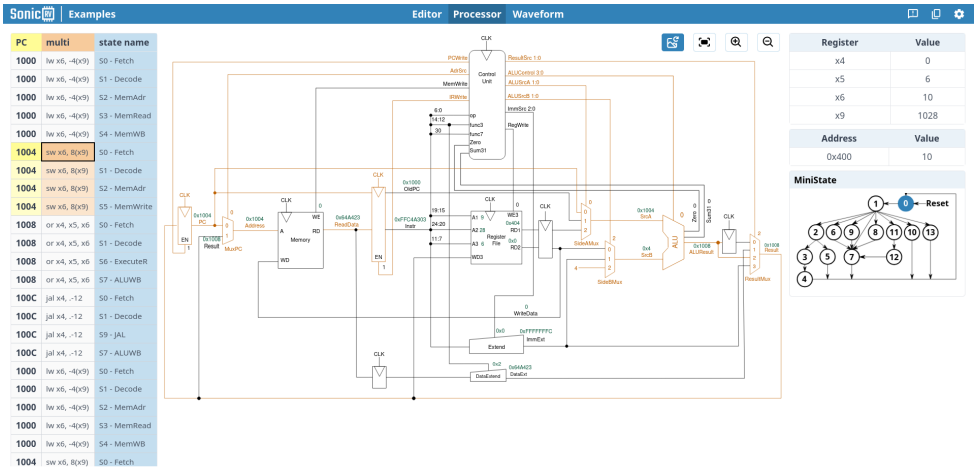

Figure 4.11: First cycle of the *sw* instruction

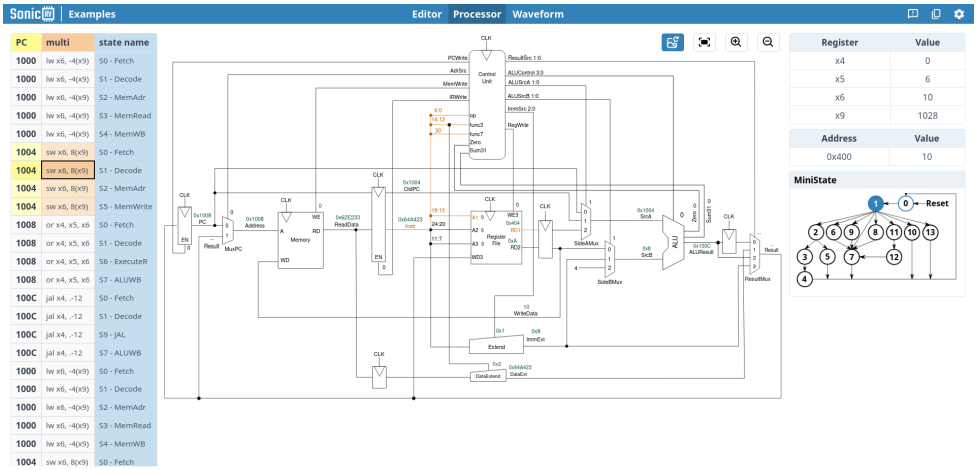

Figure 4.12: Second cycle of the *sw* instruction

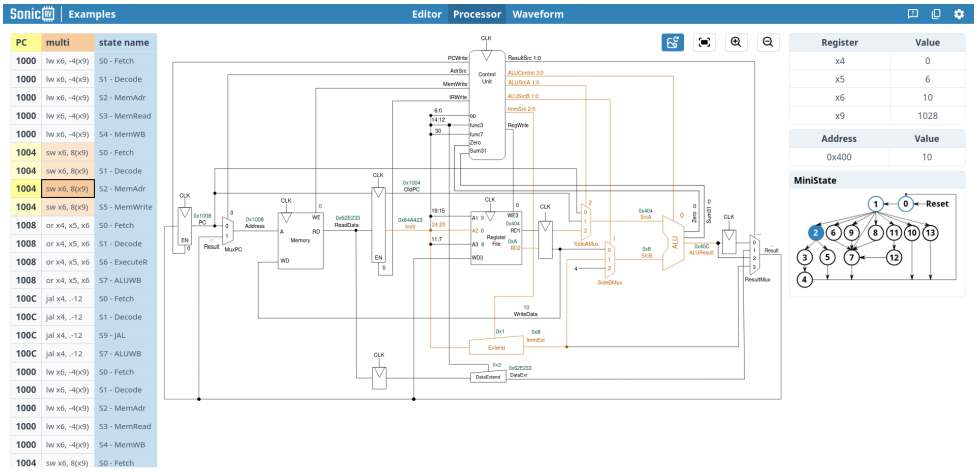

Figure 4.13: Third cycle of the *sw* instruction

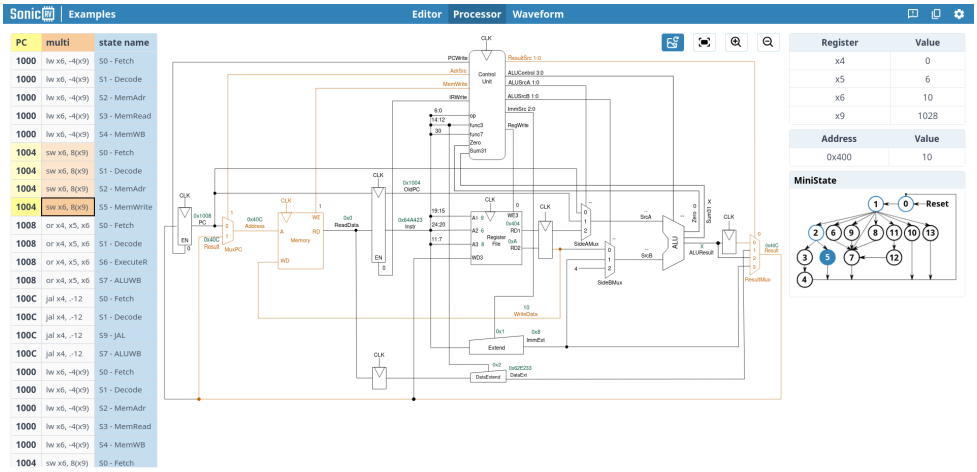

Figure 4.14: Fourth cycle of the *sw* instruction

4.3.4 *or*-Instruction

The next instruction in the example program is the *or x4, x5, x6* instruction. As an *R-Type* instruction, it computes the bitwise OR of the contents of registers *x5* and *x6* and writes the result to register *x4*. After completion of the instruction, the register table is expected to reflect this updated value.

As with all instructions, execution begins in the *S0 – Fetch* state, shown in Figure 4.15. In this state, the instruction is fetched from memory and the *PC* is incremented. The memory table now reflects the effect of the preceding *sw* instruction: the value of register *x6* has been stored at memory address `0x40C`, as previously described in Section 4.2. Since the *Fetch* state is identical across all instructions, it is not discussed in further detail.

The processor then enters the *S1 – Decode* state (see Figure 4.16). Being an *R-type* instruction, three registers are involved. The source registers *x5* (selected via *A1*) and *x6* (selected via *A2*) are read from the *Register File* and stored in temporary registers. At the same time, the destination register *x4* is selected via *A3*. In this state also both operand values are prepared for execution by storing them in the register next to the *Register File* on the right.

Next, the FSM transitions to the *S6 – ExecuteR* state (Execute *R-rype*), as depicted in Figure 4.17. In this state, the *Control Unit* routes the previously stored operand values to the inputs of the *ALU*. The *ALUControl* signal selects the bitwise OR operation (control value 3), while the signals *SrcA* with a value of 0x6 and *SrcB* with a value of 0xA are routed into the *ALU*. This results in an output of 0xE through the bitwise OR operation, shown by the *ALUResult* signal. The resulting value is subsequently stored in a temporary register for later write-back.

The *ALUControl* signal selects the bitwise OR operation (control value 3) and the signals *SrcA* with value 0x6 and *SrcB* with value 0xA are propagated into the *ALU*, resulting in 0xE by the bitwise OR operation. This value is then stored in a temporary register for subsequent write-back, propagated by the *ALUResult* signal.

Finally, the instruction reaches the *S7 – ALUWB* (ALU Writeback) state. As shown in Figure 4.18, the stored *ALUResult* is forwarded to the *Register File* via the *Result* path. Simultaneously, the *RegWrite* signal is asserted, enabling the value present on input *WD3* to be written to the destination register specified by *A3*, namely *x4*.

After completion of the *or* instruction, the value x5|x6 = 14 is expected to be visible in the *Register File*. This updated register content is observed at the beginning of the subsequent instruction.

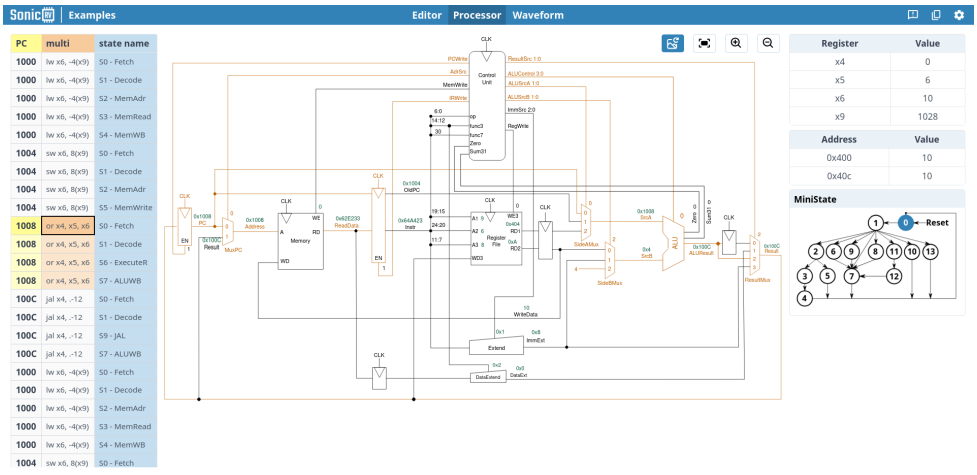

Figure 4.15: First cycle of the *or* instruction

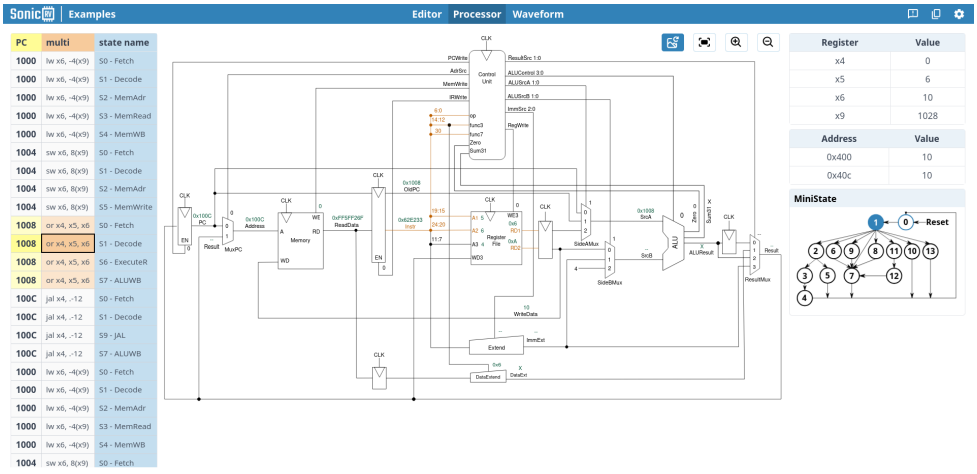

Figure 4.16: Second cycle of the *or* instruction

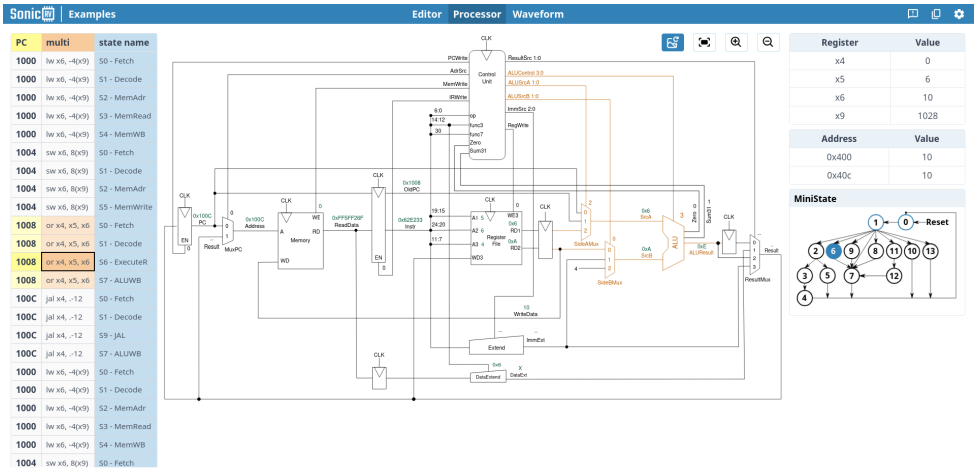

Figure 4.17: Third cycle of the *or* instruction

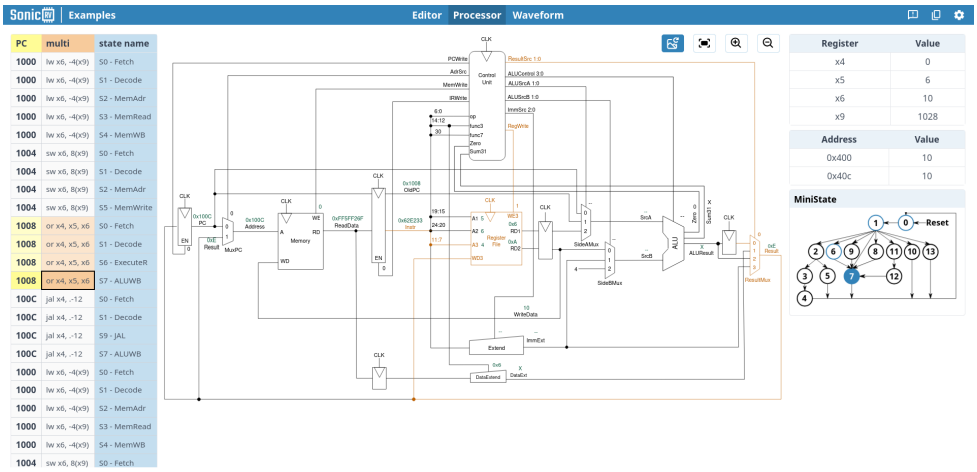

Figure 4.18: Fourth cycle of the *or* instruction

4.3.5 *jal*-Instruction

Finally, the *jal x4, L6* instruction is executed to perform an unconditional jump to the label *L6*. As with all instructions, execution begins in the *S0 – Fetch* state, shown in Figure 4.19. The instruction is fetched and the *PC* is incremented. At this point, the register table reflects the result of the preceding *or* instruction, with register *x4* holding the value 14. Since the datapath activity in the *Fetch* state is identical for all instructions, it is not discussed further. After completion of the *jal* instruction, the processor is expected to write the return address i.e., the *PC* of the subsequent sequential instruction—to register *x4*. In this case, the expected value is 0x1010 or 4112 in decimal representation.

The processor then transitions to the *S1 – Decode* state. In contrast to previously discussed instructions, only the destination register *x4* is selected (via *A3*). Simultaneously, the jump target address is computed by adding the sign-extended immediate offset encoded in the instruction to the current *PC*. This offset was produced by the RV32I assembler and embedded directly in the instruction. Consequently, the symbolic instruction *jal x4, L6* appears as *jal x4, -12* in the instruction table. The value −12 corresponds to a backward jump of three instructions, each occupying 4 bytes.

Next, the *S3 – JAL* state is executed, as shown in Figure 4.21. In this state, the original *PC* value is incremented by 4 and stored temporarily as the return address. At the same time, the previously computed jump target address is written to the *PC* register, thereby redirecting instruction fetch to the target label *L6*. This address determines the *PC* of the next instruction, which is the *lw* instruction at the jump target.

Finally, the instruction reaches the *S7 – ALUWB* (ALU Writeback) state, depicted in Figure 4.22. In this state, the stored return address value (0x1010) is written to the destination register *x4*. This is achieved by forwarding the value via the *Result* path to input *WD3* of the *Register File* while asserting the *RegWrite* signal.

To conclude the demonstration, the subsequent instruction is shown in Figure 4.23. The processor has returned to the *lw x6, -4(x9)* instruction, now in its initial instruction. This final step confirms the correct operation of the example program in SonicRV, when comparing it to the test execution in Section 4.2. This walk-through and detailed step-by-step demonstration suggests, that the implemented features, such as the state diagrams, instruction table block highlighting, signal highlighting work correctly, at least for this example.

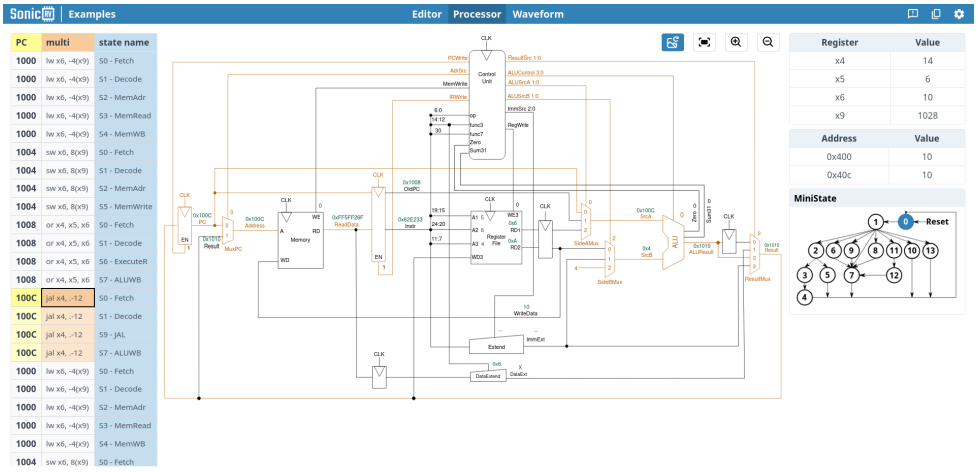

Figure 4.19: First cycle of the *jal* instruction

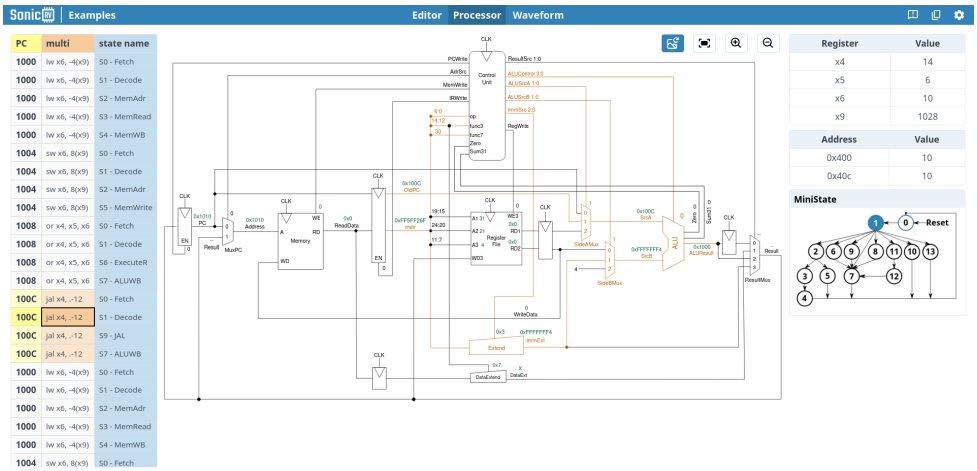

Figure 4.20: Second cycle of the *jal* instruction

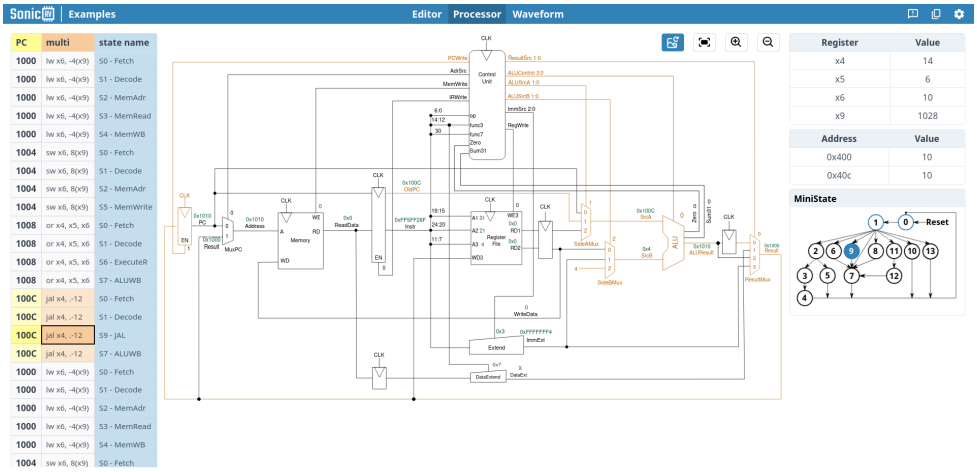

Figure 4.21: Third cycle of the *jal* instruction

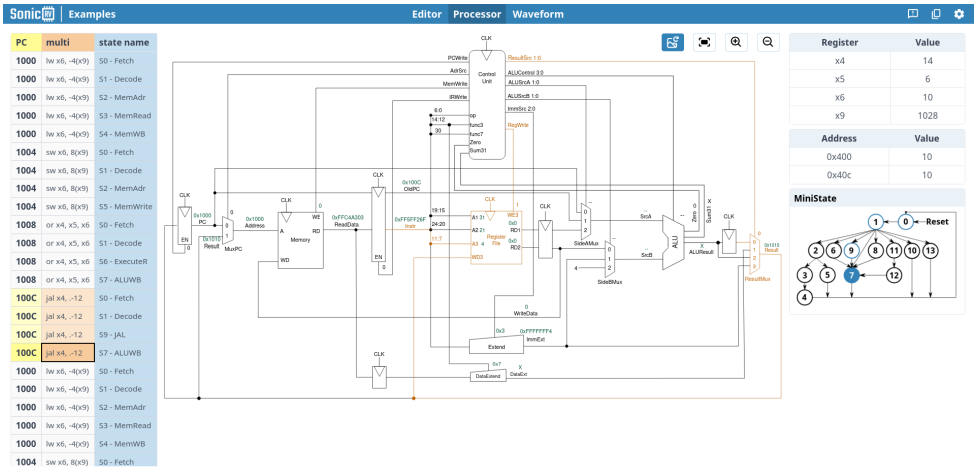

Figure 4.22: Fourth cycle of the *jal* instruction

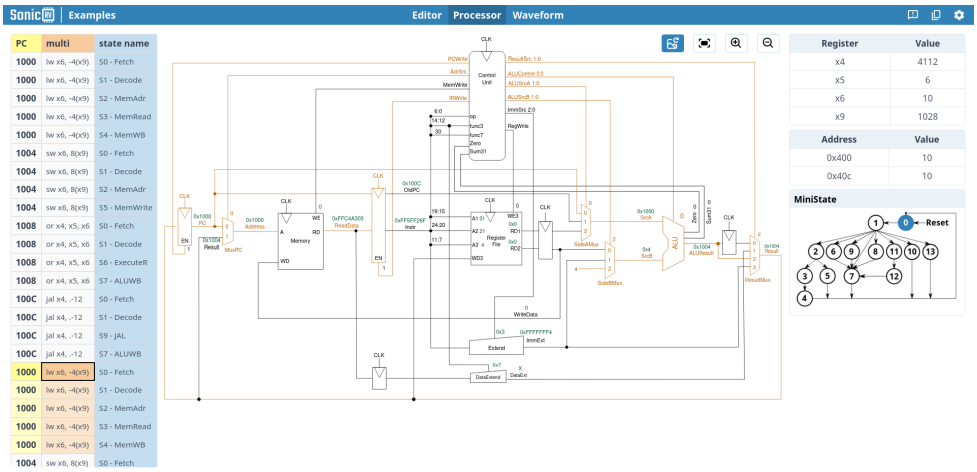

Figure 4.23: Second cycle of the *lw* instruction in the second iteration

Chapter 5

Conclusions

In this thesis, we presented the integration of an existing RV32I multi-cycle processor into SonicRV, an educational web-based application. SonicRV is used in the Computer Science bachelor courses for learning processor architectures. We began by outlining the current state of SonicRV and introducing its fundamental technologies and used tools. Following this, we provided a detailed explanation of the integration process for the multi-cycle core, offering insights into the existing framework and the necessary adaptations for the successful integration. A step-by-step demonstration of a RV32I sample program illustrated the newly implemented features, including block highlighting, state diagrams, and dynamic highlighting, emphasizing SonicRV's role in enhancing the understanding of hardware architecture and the design of multi-cycle cores.

As a result of this thesis, SonicRV now supports all three primary processor architectures (single-cycle, multi-cycle, and pipelined) enabling researchers and students to simulate, visualize, and compare these architectures within one single web application.

Bibliography

[1] Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. 2000. *Scalable vector graphics (SVG) 1.0 specification*. iuniverse Bloomington.

[2] Tristan Gingold. 2024. GHDL: VHDL 2008/93/87 simulator. (2024). http://ghdl.free.fr/ghdl/index.html.

[3] Daniel Große, Simon Reitinger, and Lucas Klemmer. 2026. SonicRV: An Educational Platform for Web-Based Simulation and Visualization of RISC-V Processor Architectures. *IEEE Access*. (early access). DOI: 10.1109/ACCESS.2026.3656022.

[4] Alan Grosskurth and Michael W Godfrey. 2005. A reference architecture for web browsers. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, pp. 661–664.

[5] Sarah Harris and David Harris. 2021. *Digital Design and Computer Architecture, RISC-V Edition*. Elsevier.

[6] 2019. IEEE Standard for VHDL Language Reference Manual. *IEEE Std 1076-2019*, 1–673. DOI: 10.1109/IEEESTD.2019.8938196.

[7] Lucas Klemmer and Daniel Große. 2024. Towards a Highly Interactive Design-Debug-Verification Cycle. In *ASP Design Automation Conf.* Pp. 692–697.

[8] Lucas Klemmer and Daniel Große. 2025. *Versatile Hardware Analysis Techniques – From Waveform-based Analysis to Formal Verification*. Springer. DOI: 10.1007/978-3-031-83093-8.

[9] Lucas Klemmer and Daniel Große. 2022. WAL: A Novel Waveform Analysis Language for Advanced Design Understanding and Debugging. In *ASP Design Automation Conf.* Pp. 358–364.

[10] Lucas Klemmer and Daniel Große. 2024. WAVING Goodbye to Manual Waveform Analysis in HDL Design with WAL. *IEEE Transactions on Computer Aided Design of Circuits and Systems*, 43, 10, 3198–3211.

[11] Lucas Klemmer, Frans Skarman, Oscar Gustafsson, and Daniel Große. 2024. Surfer: A Waveform Viewer as Dynamic as RISC-V. In *RISC-V Summit Europe*.

[12] David A. Patterson and Andrew Waterman. 2017. *The RISC-V Reader - An Open Architecture Atlas*. Strawberry Canyon LLC. ISBN: 978-0-999-24911-6.

[13] Frans Skarman, Lucas Klemmer, Daniel Große, Oscar Gustafsson, and Kevin Laeufer. 2025. Surfer – An Extensible Waveform Viewer. In *International Conference on Computer Aided Verification (CAV)*, pp. 392–404. DOI: 10.1007/978-3-031-98685-7_19.

[14] Oliver Trost. 2024. *RISC-V Multi-Cycle Processor Implementation in VHDL*. Bachelor thesis. Johannes Kepler University Linz.

[15] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley.