Computer Architecture 02155 - Fall 2023

Final Assignment

Alexander Wang Aakersø	s223998
Georg Brink Dyvad	s224038

Group 23

November 28, 2023

This report contains 10 page(s).

Contents

Introduction

This project report presents the design, testing, and implementation of an RISC-V instruction set simulator written using Java, able to execute RISC-V programs. The simulator correctly implements a minimal subset of the RISC-V instruction set (RV32I), with only few exceptions (ignored instructions). The simulator presents all processor state registers along with a program counter (PC), and is implemented with 1MB of memory.

The design of our simulator has been made with a primary objective in mind: Staying faithful to the real-life components within a processing system. Memories store data using bytes, hence our processor uses a single byte array as memory, containing both instructions and data. Processors can either be multi-staged (pipelined) or single-staged (one-cycle). In this project, we've chosen the single-staged processor, as this type is simpler and thereby easier to implement in code, allowing the resulting simulator-design to be more in line with the real world.

As such, our simulator employs roughly the same units that a single-staged processor would. Firstly, the PC controls what instruction is fetched from our memory. Then the instruction is decoded; through logical operations, shifting, and bitmasks we extract the needed information to identify the type of instruction. Having decoded the instruction we are able to engage the needed registers from our register array, on which we perform logical and arithmetic operations (much as in the arithmetic unit of a processor) to execute the given instruction. If this instruction entails accessing the memory, this is also done here. At the very end, our PC is incremented by four, although this increment depends on what instruction is being executed, just as it is in a real processor.

In conclusion, this project demonstrates the successful development of an RISC-V instruction set simulator, fulfilling the requirements set forth in the given project description.

1 Description of Design and Implementation of Simulator

This section presents and explains the choices we've made during our project, how these choices impacted our program, and how our final program design has been implemented.

1.1 General philosophy of design

Working on projects such as this, planning is very important. As such, when we first started work, we decided on a set of design-decisions, that would make sure our program got to where we wanted it:

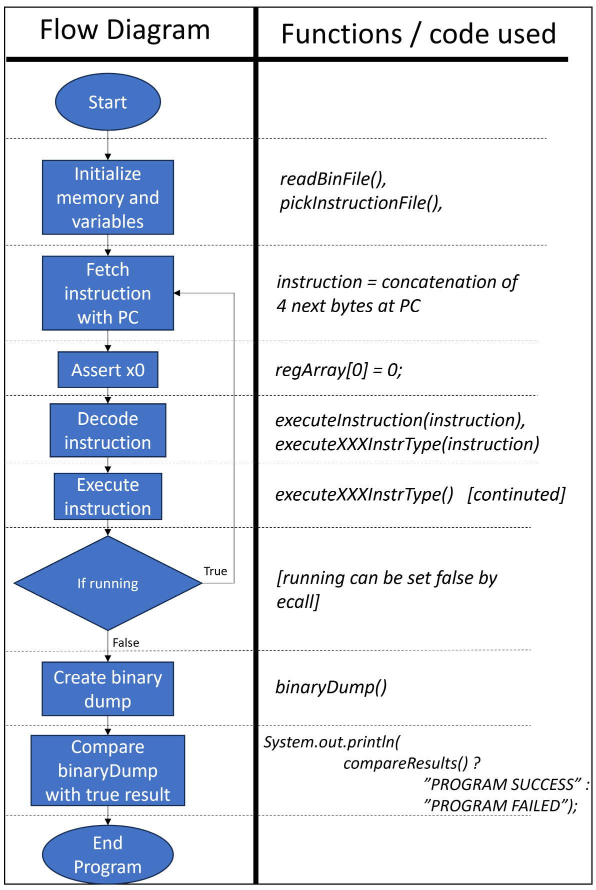
> Firstly, we wanted our simulator to be highly modular. This meaning, that the program should be easy to expand. Our structure should therefore be built gradually and segmented. The process of implementation should be to write a small segment of the simulator, test it, then upon this segment write a new segment, test it, and so on. This comes with two advantages: Isolating bugs and errors becomes easy, and thus the scope of our project becomes easier to handle. Also, Structurally designing our program to be modular allows for future upgrades to be made very easily. It is conceivable, and very likely, that we in the future would want to extend the instruction set of our simulator. For example, when we in coming courses are to implement such a processor in hardware, where having a simulator to compare with is great.
>
> Secondly, choices regarding the structure of our program should be made by weighting the importance of staying true to the structures in a real processor against the difficulty of implementing such structures in Java. It is for example clear, that operating on our instructions would be far easier if these were saved in an integer array. This however would mean using two separate memories, where real processors share one memory for both instructions and data. Staying faithful to the structure of a processor, we've chosen to only use a single memory, making our program a little more complicated.

With this philosophy in mind, the following sections detail how we implemented our simulator:

1.2 Top-level of our design: Main

In this project, we've decided to use the class-like nature of Java to define static variables which we can change from anywhere in the class. Such static variables include (not limited to) the register array [integer array of length 32], our program counter[integer], and our memory[byte array of length 0x100000 (1MB)]. These together show the state and memory of our processor and must therefore be within the scope of our functions.

Below we introduce a flow diagram which describes this top-level of our simulator.



```
readBinFile(),
pickInstructionFile(),

instruction = concatenation of
4 next bytes at PC

regArray[0] = 0;

executeInstruction(instruction),
executeXXXInstrType(instruction)

executeXXXInstrType()   [continued]

[running can be set false by
ecall]

binaryDump()

System.out.println(
        compareResults() ?
                "PROGRAM SUCCESS" :
                "PROGRAM FAILED");
```

Figure 1: Flow diagram of our top level and important code-snippets used in each section.

This top level is executed in main(). Notice that we at the start of our program read the entire given binary file - the execution of the instructions in this binary file is however done individually, using a while-loop, which is controlled using our program counter (PC). Beginning in a while loop, we fetch instructions using the integer coded address of the program counter. Next we assert the constant value of the x0 register, as this should always be 0. We then call our executeInstruction() function which both decodes and executes the current instruction.

31 27 26 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: Core instruction formats of the RISC-V instruction set [2]. It can be seen that there are six different formats, of which all are used in the RV32I base set. Also notice how for all six formats, the opcode is stored in the same 7 bits.

We first identify the opcode to determine in which way to decode the rest of the instruction. We then execute the instruction in a separate instruction-format specific function based on the decoded parameters. After instruction execution, our program counter is incremented, and the next instruction is executed. After execution of the last instruction, the while loop breaks. Post program execution we create a binary file containing the value stored in our registers and compare this to the expected output, given as part of the project description.

The following sections dive into the implementation of the components seen in our flow-diagram.

1.3 Fetching instructions from binary file

The user is upon opening the program greeted by a message, allowing them to enter the name of a binary file (without the .bin identifier). Our function readBinFile() uses Java's file-reading capabilities in the form of Fileinputstream and Datainputstream to write the given binary values directly into our memory, the first byte being placed in 0x000000.

1.4 Decoding instructions

Before executing the fetched instructions we first need to decode them. To do so, we use our program counter to index our memory, fetching the current byte and the next three bytes in succession. These four bytes together must describe the complete instruction. We concatenate these into an integer, using bitshifting, bitmasking, and arithmetic or-operations.

Referring to the RISC-V card [2], there are six different instruction formats we need to implement and ten different opcodes among them. We chose to differentiate between instructions based on their opcode by making Java functions for each opcode. To decipher the opcode, we simply left bit-shift the instruction 25 bits, then unsigned right shift by 25 bits, leaving the 7 LSB, which in the RISC-V structure describes the instruction opcode. This opcode is then used as a switch in a switch-case structure, which calls the next level of decoding.

Knowing the opcode, we also know the instruction type, which allows us to decode the rest of the instruction (like funct3, immediate, rs1, rs2, rd and funct7). To do so, we use bit-masking and bit-shifting. When extracting the immediate parameter as an integer it's important to extend its signage beyond the 12-20 bits which are given in the instruction. Using the funct3 field, we again utilize a switch-case statement to execute our instruction.

1.5 Executing instructions

After constructing the initial structure for instruction decoding it was easy to fill out the cases of each instruction. Again referring to the RISC-V card [2] the included C code can be rewritten in Java without problems. Most instructions can be written in one line, but others are more challenging. When writing instructions interacting with the memory, the bit-wise operation statements can become rather tedious and there are many pitfalls to avoid. Proper handling of the signage of integers cast from bytes is very important to avoid long debugging sessions.

1.6 Memory

We kept the structure of our memory true to a real processor by representing it with a byte array. This allows for straightforward implementations of the memory related instructions. Seeing as a real processor uses a shared instruction-data-memory, we ended up doing the same. Compared to a real processor our memory has no reserved unusable sections making it easier to work with in the assembly code.

2 Discussion of Design

In this section we explain and discuss our final design. Important choices, such as what programming language was used, how the group collaborated on the project, and certain changes that we in retrospect could have made to better our program.

2.1 Why Java?

Choosing the language for our project, we quickly came to the conclusion that we had two options. Use C or use Java. We did not consider Python to be viable, as it did not introduce anything that we felt could be of use to our program, while it's very high abstraction level removes the possibility of interacting with memory and datatypes on a lower-level, which could be useful.

Choosing between Java and C came down to preference. C has a lot of functions, which Java does not (such as pointers and dynamic memory-allocation), however we did not feel the need for using these. Being more practised in Java, having previously used it for Chisel, we decided to go this way.

Also, using Java allowed us to use the code editor IntelliJ. One of the best features of IntelliJ is "Code Together" which allows for the entire group to work on a project concurrently, eliminating the need for creating a GIT-repository.

2.2 Modular design

Our simulator design is very modular in the sense that it can easily be expanded to include more parts of the RISC-V complete instruction set. Any instructions can be added, simply by writing an opcode-specific function and then calling this function in executeInstruction() of which a new case must be added.

Looking at our design, we chose to sort our instructions based on their opcode. In retrospect, it could also have been a possibility to group instructions based on the instruction format-type such as R, I, etc. Many of the instructions with different opcodes share the same format-type, meaning the decryption of these are all done in the same way. The consequence of differentiating by opcode is repeating the same lines used for decoding our format-types for each different opcode, which if the program was to be expanded, could become very many lines. But on the plus-side we only have to decode and check the opcode once. If we split instructions into groups based on format-type we would have to make an additional switch case on the opcode in each of these functions. Thus there are advantages and disadvantages to each instruction grouping approach.

2.3 Simulator accuracy to processor structures

One of our main-design philosophies were to create a simulator, which kept itself faithful to the structures employed by real processors. Looking at our structures, we would say that this has been fulfilled. Our registers are of the type integer (32 bits). Our program counter is an integer, which counts bytes, just as it does in a real processor (we increment it by 4 to move to the next instruction). Our memory is constructed using a byte-array, each storing up to 8 bits, which fits with the memory of a processor, being able to index using bytes. Also, instructions are saved in the same memory as data, which means that our simulator only uses a single memory, as a real processor often does.

2.4 Lessons learned in the process of implementing our design

At first when designing the memory for our simulator we chose to separate instructions from the rest of the memory. Previously we had an integer array as the memory for our instructions and a byte array for our general memory. This is reflected in the way our instruction decoding operates on integers instead of bytes. In the end this proved to be the better decoding strategy even after switching to a single memory.

Late in the process of designing our simulator we made this switch from two separate memories to a single one. Ultimately the advantages of a single memory outweighed our desire to keep our program as was. Because of our modular program structure it was surprisingly easy to make the change and even reduced the total number of code-lines in our project. Practically there was no difference between the functionality of the two simulator memories, but in niche cases not explored in this course like self-modifying it could have made a difference.

As mentioned in the modular design section we chose to go with opcode grouping of instructions rather than format-type grouping as the method to split instruction execution into smaller functions. Format-type instruction grouping might scale a little better for potential expansions into other parts of the RISC-V instruction set but for our particular minimal implementation it was the easiest to go for opcode grouping.

2.5 Testing and debugging our simulator

The simulator passes every given test in the GitHub [1]. These tests allowed us to test different sets of functionality of our simulator during the process of writing it. Following the recommended strategy of first implementing basic logical and arithmetic instructions we could use the tests in task1 to identify bugs before moving on to the next phase. This reduced the overall time spent debugging as it is much easier to debug a program with two errors rather than twenty.

We also wrote a few functions with the sole purpose of easing the process of testing and debugging. We used our printOutput() function to monitor the values of the registers after the execution of each instruction. This was helpful to identify the instructions that were bugged. We also wrote a compareResult() function to complement the required binaryDump() function. This allowed us to instantly check whether our simulator passed a test without having to manually go through the value of each register, by crosschecking our binary dump of the final register values with the expected result-files.

3 Conclusion

In this project we have successfully designed, implemented, and verified a working simulator of the RV32I RISC-V instruction set. To do so, we took offset in a design-philosophy, which focuses on our program being highly modular and faithful to the workings of a real single-cycle processor. This philosophy has been expressed in the format of our code, the choice of structures, data types, and the general flow of our simulator. We conclude that our processor fulfills all requirements stated in the project assignment and that it has passed all available tests.

While working on this project we have gained a deeper understanding of the RISC-V ISA and the architecture of a computer processor (single cycle). It has been very rewarding to gradually implement more instructions and upgrade our simulator with the ability to pass more and more of the given tests. This project has inspired us to continue with more courses in digital electronics and we look forward to doing the 3-week course on an FPGA implementation of a RISC-V processor.

4 Bibliography

1. Schoeberl, M. (2023). *Lab Material for Computer Architecture.* [online] GitHub. Available at: `https://github.com/schoeberl/cae-lab` [Accessed 28 Nov. 2023].

2. SFU Computing Science. *RISC-V Reference RISC-V Instruction Set.* [online] Available at: `https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/notebooks/RISCV/RISCV_CARD.pdf` [Accessed 28 Nov. 2023].

3. JetBrains (2019). *IntelliJ IDEA.* [online] JetBrains. Available at: `https://www.jetbrains.com/idea/`.

4. Petersen, M.B. (2023). *Ripes.* [online] GitHub. Available at: `https://github.com/mortbopet/Ripes` [Accessed 28 Nov. 2023].

5 Appendix A: Table of Figures

Figure	Description	Source
1	Flow diagram of simulator	Georg Brink Dyvad
2	Core RISC-V instruction formats	[2]