Final Assignment
An RISC-V instruction set simulator

02155 - Computer Architecture and Engineering, Autumn 2022

Group 14

Troels Lund - s161791

Technical University of Denmark
DTU Compute

Sunday, the 4th of Dec, 2022 - 23:59

Number of pages: 7

Content

1 Introduction

This report documents the implementation of a very basic RISC-V instruction set simulator. The report is the final assignment of course 02155 - Computer Architecture and Engineering at the Technical University of Denmark.

The simulator targets the Base Integer Instruction Set - RV32I, with the exceptions given in the problem description. Only the Instruction Set has been simulated, and features such as pipelining and other architectural elements have been ignored due to the scope of the project.

2 Design

The simulator is built to roughly follow the flow of a real processor and will execute a program in a *Single Cycle Processor* manner. Furthermore, the simulator is designed to operate as a little-endian system.

When the simulator starts, it sets the starting state of the machine. This includes making sure that the *PC* is set to zero. Furthermore, it allocates the space for the 32 registers in an array and *1 MB* of space for the memory. The registers consist of a 32 bit signed integer array, thus the *int* data type of Java.

This also means that every time the simulator need to handle an value from the registers as an unsigned integer a conversion have to be made.

Memory is represented as a byte array. The memory is designed to be addressed in 8-bit bytes. This means that when using a instruction as load byte (*lb*) the simulator will load the value of one position in the memory array, a half word (16-bit) two position is loaded and a word (32-bit) four position is loaded. To extract the 16-bit and 32-bit values, the 8-bit parts is combined with a bit-wise-or.

Since the program is starting at *0x0* in memory there is no reserved space segment before the text area, and the *pc* is therefore initialized to zero.

When the simulator is instantiated it is ready to execute instructions. The simulator can handle running multiple programs before shutting down. Each program will leave the simulator in a state that the next program can then work on.

When the simulator is given a program, it loads the instructions (text) into memory from the position *0* in memory, and immediately afterwards the static data.

Instructions are expected to be given in the 32-bit RISC-V instruction formats of the types *R*, *I*, *S*, *SB*, *U* and *UJ*, with instructions in little-endian order. The instructions is given as a integer array where each entry is a instruction. The loading of the programs is handled by a separate class.

The core of the simulator is the loop that handles each instruction of a given program, and every instruction is evaluated by the simulator in three steps:

1. Fetch instruction
   - The Fetch instruction step will load the 32 bit instruction from memory and represent it as a integer.
2. Decode the instruction
   - The decode step will divide the instruction into fields such as the *rd*, *intermediate func3*, *func7* ect. based on the instruction format found, looking at the *opcode*.
3. Executes the instruction
   - Mutate the state of the machine accordingly to the instruction type and fields.

In case the simulator in counters a problem, in any of the three steps, the program will raise an exception. The exception will be handled by printing the current state and if the simulator is in *debugger mode*. Then, end the program with exit code 99. This could, for instance, be when a program containing an unsupported/unimplemented instruction is attempted to be executed.

The expected way to end the program is by calling *ecall 10* by setting $a7 = 10$ when the default settings apply. The registers used for the *ecall* can be set, this was done to make sure it was compatible with test programs used for both the *Venus* and *Ripes* simulator.

The simulator is designed to incorporate a *debugger mode*. This mode will make the simulator print useful information to the console. This information includes the *program counter* both in base 10 and base 6. Furthermore the *opcode*, instruction format and the specific instruction executed. To print all the registers at each cycle, the *printReg* flag must be equal to *True*.

3 Implementation

This section will outline the implementation of the simulator. The project is written in Java (openjdk 18.0.2.1). A simple *CLI* was also developed to make it easy to run programs on the simulator.

3.1 Input and output

The program loader simply contains logic to load programs. Since test programs are provided as binary files, the core functionality of the program loader is to load binary files. The method that reads the file is called *readBinFile*. This method returns the program as a integer array. Each instruction is read one after the other in four-byte chunks. This is done by with the method *readInt* on a open *DataInputStream* associated with the specific binary file. When reading the the bytes it was discovered that the bits was stored in the file in big-endian order. Therefore, the order of bytes was reversed.

In Java the integer data type always 32-bit and therefore this will not change when running on different architectures.

Furthermore, the functionality of writing the contents of registers *x0-x31*, in a binary file has been implemented. This functionality belongs to the *DataDumper* class.

3.2 Instruction decoder

As described in the design section, the instructions have to be decoded. This is the responsibility of the *InstructionDecoder* class. This class will from the opcode map the instruction into an object, representing that specific format type of instruction.

The fields are then extracted in the constructor of each type, mostly by simply shifting the bit into place and then using a mask to take only the bits concerning that field.

All the instruction format data classes are simply used to encapsulate the data. They do all inherited from the same abstract base class *Instruction*. By doing this, the program can handle all instructions in a polymorphic way and abstract away the complexity of the different formats.

3.3 Simulator core

This class named *ISASimulator* is the core of the simulator. The *c*lass contains the state of the machine and all logic for mutating the state.

The state consists of the *program counter* as an integer, the 32 registers as an integer array, and the memory. The memory is a byte array and have the length of *0x100000*. This makes the total size of memory 1 MB. This is done because the test programs expect that this amount of memory will be allocated to run correctly. Since Java's byte data type is signed it is implemented so the data is stored as signed bytes and then when loaded is converted to an integer with the unsigned value. To do these "conversions", small auxiliary functions are used to improve readability. These auxiliary functions are used to convert from signed to unsigned for both bytes and integers and do sign-extensions.

The *ISASimulator* class contains the *runProgram* method. This method takes as an input a program, and optionally a name of the program. The *runProgram* will execute the program and contains the fetch-decode-execute loop described in design. In practice, it is implemented as a while loop that will run with the condition of *!end*. The *end* variable is a Boolean value used to stop the program.

This can happen in three ways. Two of those is ordinary valid was to end the program and one is in case of an error. The two valid ways are that there are no more instructions to execute or that the instruction *ecall 10* is executed.

As described before will the *InstructionDecoder* provide a Instruction object with fields extracted. This object is then handled by the *exeInstr* method that will based on the instruction type to call a method that contains all the business logic for that type. Here the *opcode* and *func* fields are used to resolve witch action to preform.

After each cycle, the *x0* register is set to zero. This was the easiest way to ensure that a store on the zero register has no effect on future execution. In the case that *x0* is set to an other value, it will be zero again before the next cycle and therefore will keep the program consistent.

In each loop, the *instrCount* variable is incremented; this variable has no effect on the execution and is only there for debugging purposes.

Arithmetic and logical instructions were the first implemented since these are very simple to implement. This group of instructions simulates the basic computations that in a real processor take place in the *ALU*.

These instructions operate on data from the registers. This could, for example, be the arithmetic instruction *addi*. *addi* will take the a destination register - *rd*, a source register - *rs1* and an intermediate value *imm*. The source register and the intermediate value values will be added together and then written to the destination register.

In some cases the signed integers' values need to be handled as unsigned integers. This for instance happens when using the logical instruction *sltiu* both the *rs1* and *imm* are treated as unsigned numbers. In those cases, a conversion is done.

Branch instructions enables the program to deviate from the strictly sequential order of instructions. This is done by changing the *program counter* according to the specific instruction used.

Since the *program counter* is incremented by 4 bytes after execution in each cycle. All branching instructions need to take this in to a counter in the implementation, this is simply done by subtracting 4 bytes from the new *program counter*.

3.4 CLI

To make it easier to use the simulator, a simple CLI has been made. The project is set up to produce the artifact *rv32i.jar* that contains the simulator itself and some code that allows the user to use the simulator as a CLI tool.

The produced *jar* file can be used as follows:

```
java -cp "picocli-4.7.0.jar:rv32i.jar" RV32I <file path> <flags>
```

More information can be found in the *readme.md* file included in the project.

4 Discussion

The implementation of the simulator has a very basic structure and was not designed to be extended much further and, therefore, would need a substantial refactoring. An example of this is that all the logic for executing the instructions and the machine state is located in one class. Furthermore the implementation not in any way optimized for performance but rather readability and simplicity.

Furthermore, the implementation could have been implemented to closely match hardware components. Instead a higher abstraction model has been chosen for this implementation.

Choosing a language such as *C* to implement the simulator could have yielded multiple benefits. For instance a higher performance, this would be beneficial in case the simulator should be extended, to simulate more sophisticated programs.

Furthermore *C* could have made it possible to avoid the adaptation of types, since the *C* language have a much richer variations of types.

The test cases was implemented very early in the development process. This made the process much easier, since the test could be done frequently to ensure the correctness of the implementation.

5 Tests

All test cases given in as part of the assignment have been used to evaluate the correctness of the simulator. To make this process easier a unit testing framework (*JUnit*) was set up at the start of development.

Test converge was *91%*, measured in percentages of lines tested in the core *ISASimulator* class.

Must of the code not tested is the code that handles errors, it would therefore be beneficial to add some negative test cases.

To simplify the process of writing each test, a testing utility that loads the program *.bin* binary file and the register results *.res* file. Then the program contained in the *.bin* file is executed on an instance of the simulator. The test is then evaluated by comparing all the registers with the register values stored in the corresponding *.res* binary files.

All test cases given for *task 1*, *task 2*, *task 3* and *task 4* was passed.

Furthermore the register file dump functionality is tested to make sure this feature work correctly. The dump-files can therefore also be used to test the correctness of the simulator.

6 Conclusion

A basic RISC-V instruction set simulator was built that implements all RV32I instructions. It successfully passed all tests given for the project, and meets the requirements of the given assignment.

Implementing the simulator have been a both fun and joyful learning experience, that have me more familiar with the RISC-V ISA.