02155 - Computer architecture and Engineering
Fall 2022

Final Assignment

Group 31

Daniel F. Hauge (s201186)

This report contains 5 pages

December 4, 2022

Introduction

This report covers the final assignment of course 02155 Computer Architecture and Engineering. The objective of the assignment is to simulate the base RS32I instruction set of a RISC-V processor. The following instructions is ignored however:

- fence
- fence.i
- ebreak
- csrrw
- csrrs
- csrrc
- csrrwi
- csrrsi
- csrrci

The simulator is able to ingest a binary file, that is compiled with RISC-V RS32I as target instruction set. The ingested binary file is executed and as a result, print a correct register state (according to the program binaries).

Additionally, an output path can be specified to write the register values to little endian encoded binaries, starting from x0 to x31.

The simulator can be run on ubuntu with the following instructions:

Figure 1: Install golang (windows installer)

```
curl -O $HOME/go.tar.gz -L https://go.dev/dl/go1.19.3.linux-amd64.tar.gz
rm -rf $HOME/go && tar -C $HOME -xzf go.tar.gz
export PATH=$PATH:$HOME/go/bin
go version
```

If succesfull, go version should print something like "go version go1.19.3 linux/amd64". The simulator can then be used in the following 2 ways, given current directory is in source code root:

Figure 2: Run (Build & run)

```
go run . ./tests/binary/t1.bin ./register_dump.bin
```

Figure 3: Build executable

```
go build -o executable
./executable ./tests/binary/t1.bin ./register_dump.bin
```

Design

The design of the simulator was made with 3 main objectives.

- Extensibility
- Simplicity
- Testability

Extensibility

Although it was not planned to extend the simulator unless provided an abundance of time. Having some degree of maintainability and extensibility is relevant for this project, as with more time, the simulator would be extended with more instruction sets. To facilitate this design, code is split by their domains, like for example: Branch operations, Logical operations, Arithmetic operations, Memory operations, Decoding, Opscodes mapping, Binary handling, Tests, etc...

Simplicity

With a limited timeframe comes a limited scope. To combat potential scope creep, simplicity is choosen any time over cool or fancy stuff. There is a risk of not getting the minimal viable solution ready before deadline, if cool and fancy alleyways are visited first time around. It could be cool to boot linux, extend with all extensions (RS64I, RS32M, ...) or staged pipeline. But for this simulator, the simplest approach is taken until a more advanced approach is absolutely neccesary.

Testability

In an attempt to increase development speed and correctness, the simulator was designed with high testability in mind. Initialy tests were concieved such that a full instruction in the form of an integer is given, executed with assertions on the side effects. Although pretty granular, it was still testing too much at once with both the operation and the encoding. Another drawback of the initial approach, is that specifying integers as instructions for test cases is not very readable, and require intimate knowledge of the system.

To enable simpler tests of operations, simpler function declarations would be needed, like for example:

$$add(rd, rs1, rs2)$$

This way, a very readable and simple test could be constructed like the following:

$$a_1 \leftarrow 5$$

$$a_2 \leftarrow 3$$

$$add(a_0, a_1, a_2)$$

$$\textbf{assert } a_0 = 8$$

Decoupling decoding and operations, will make the simulator more testable, and as bonus result also make the code more readable. This pattern is extended to all parts of the simulator, such that functions are limited to a singular responsibility which makes them easier to read, determine behavior, code and test.

Implementation

Golang,

Structure

Simulator package exposing relevant functions to work with the simulator. Simulator used in main, where main facilitates file reading, memory setup ect. This structure is made to facilitate potential extensions.

Simplicity

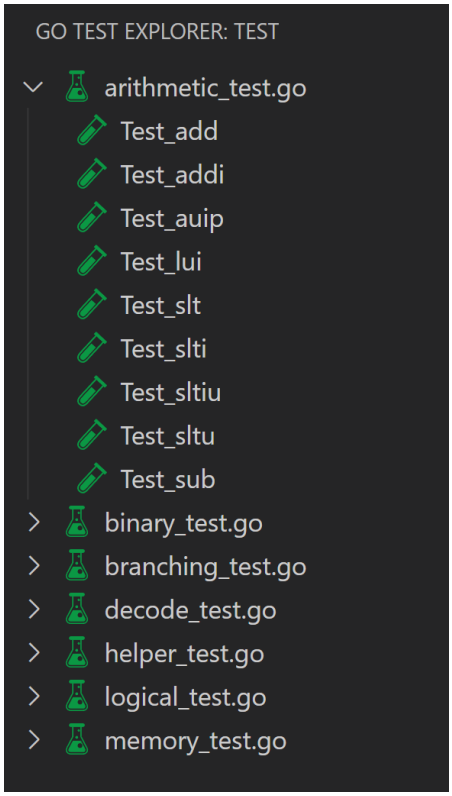
uint32 is simpler to work with, as to not think too hard about two's compliment. registers simple global static arrays of uint32 memory is simple global static arrays of uint32

Testing

Simulator package is tested with test

Some text here and her and here Some text here and her and here Some text here and her and here Some text here and her and here Some text here and her and here

```
1 func Test_add(t *testing.T) {
2     Reg[5] = 5
3     Reg[6] = 3
4     add(1, 5, 6)
5     assert(Reg[1], 8, t)
6 }
```



Discussion

- Testing was nice, but i feel into some bad habits of just debugging w/o constructing additional tests. - Testing would have been better to define tests based on exact knowledge of how RISC-V works instead of guessing from cheatsheet. - The structure was nice, and provided a good overview of the code etc. - The seperation would have made extensions easier to implement once the foundation was built. - Using uint32 made most operations trivial and predictable. - Using arrays made memory implementations very simple and trivial. - Although a more advanced virtual memory setup could be cool.