02155 - Computer architecture and Engineering
Fall 2022

Final Assignment

Group 31

Daniel F. Hauge (s201186)

This report contains 8 pages

December 4, 2022

Contents

1 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
go version go1.19.3 linux/amd64
```

If succesfull, go version should be printed. 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
```

2 Design

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

- Extensibility
- Simplicity
- Testability

2.1 Extensibility

Although extensions was not planned for the simulator from the beginning. 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...

2.2 Simplicity

With a limited timeframe comes a limited scope. To combat potential scope creep, simplicity was choosen any time over cool or fancy stuff. The risk of not getting the minimal viable solution ready before deadline was too likely and consequential, 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 was taken until a more advanced approach was absolutely neccesary.

2.3 Testability

In an attempt to increase development speed and correctness, the simulator was built 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:

$$a_1 \leftarrow 5$$

$$a_2 \leftarrow 3$$

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

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

A looser coupling, like with the decoupling of decoding and operations, will make the simulator more testable. The looser coupling also as a bonus result 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.

3 Implementation

The simulator is implementated in Golang, also known as just Go. Go is an opensource programming language supported by google, that boasts an impressive performance. Choosing Go instead of any other programming language was very much just a roll of the dice. Even if Go is choosen from chance, it does however, help implement the design quite well.

3.1 Structure

Go operates with packages that by convention is structured by folders. To facilitate an extensible and maintainable simulator, the simulator is build as a RS32I package. The simulator package is then used in a main package that handles loading binary and displaying outputs et cetera.

```
src
├── main.go
└── simulator (RS32I)
    ├── arithmetic.go
    ├── arithmetic_test.go
    ├── binary.go
    └── (etc..)
```

With this structure, new extensions could be implemented and used in the main package.

3.2 Simplicity

Registers and memory is implemented by plain global arrays. The program counter is also a global variable. To simplify the implementation, registers are using uint32 to make using Go's standard "encoding/binary" library easier.

```
var (
    Pc  int32
    Reg []uint32
    Mem []byte
)
```

Running a simulation of a binary file is then simply implemented by loading binary file into Mem array, then execute instructions in a while loop.

```
for s.Pc < programLength && !s.ECALL {
    instrBs := s.Mem[s.Pc : s.Pc+4] // Fetch instructions from memory
    inst := s.Decode(instrBs)       // Decode instruction from binary
    inst.Execute()                  // Execute operation from instruction
    s.Pc += 4                       // Increment program counter
}
```

3.3 Testing

Golang provides a testing framework as part of the standard language toolchain. By convention, tests are automatically picked up for files postfixed with "_test". The test as described in the Testability design section 2.3 can be created as follows.

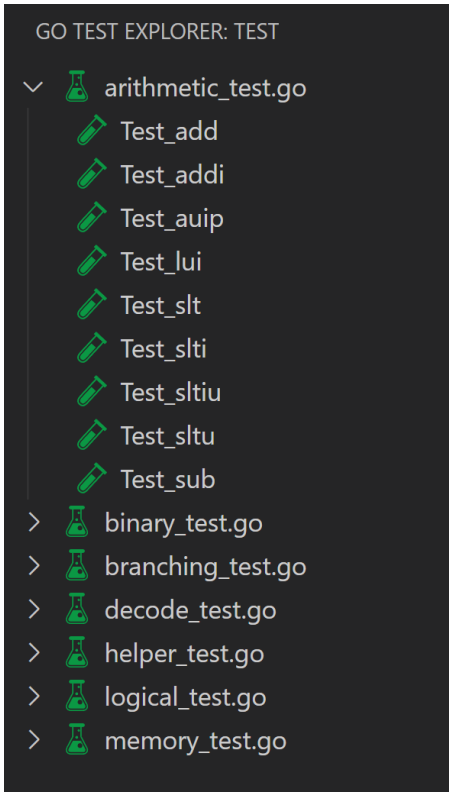
Given the *Add* function exists in **arithmetic.go**, then a test file is created named: **arithmetic_test.go**. The following test code is then put in the test file:

```
func Test_add(t *testing.T) {
    Reg[11] = 5
    Reg[12] = 3
    add(10, 11, 12)
    assert(Reg[10], 8, t)
}
```

Using command line, the tests can be executed by the following command (given source root as working directory):

```
$ go test ./simulator/
ok      ./simulator 0.246s
```

Besides the command line, there is various integrations that exists. Like the visual studio code extension view shown in the screenshot to the right.



4 Discussion

Having made the simulator as a package, extensions for other instruction sets seem not too difficult to integrate. Furthermore, using global arrays simplified the memory management and handling of registers.

Designing for testability was quite comfortable. Whenever some defect or unwanted behavior was detected, a test was not difficult to write. Having unit tests also made identifying root causes of issues found easier.

In the demo, the purpose of some of the assignment material was clarified. The files ending with ".res" is a binary dump of the expected values of the registers. With the simulators functionality to do a binary dump too, automated End-to-end tests could have been made like the following: With all the ".res" files in an **expected** directory located in the source root. Then a script could be doing the following: Foreach binary program provided, run the simulator and dump registers to an **actual** directory. Then use a compare tool like one of the gnu diffutils: *cmp* or *diff*, for example:

```
$ ls tests/binary/ | sed 's/....$//' | xargs -I {} go run . tests/binary/{}.bin actual/{}.res
$ ls expected/ | xargs -I {} diff -s ./actual/{} ./expected/{}
Files ./actual/addlarge.res and ./expected/addlarge.res are identical
Files ./actual/addneg.res and ./expected/addneg.res are identical
Files ./actual/addpos.res and ./expected/addpos.res are identical
Files ./actual/bool.res and ./expected/bool.res are identical
Files ./actual/shift.res and ./expected/shift.res are identical
Files ./actual/shift2.res and ./expected/shift2.res are identical
.........
```

From the *diff* man page:

> Exit status is 0 if inputs are the same, 1 if different, 2 if trouble.

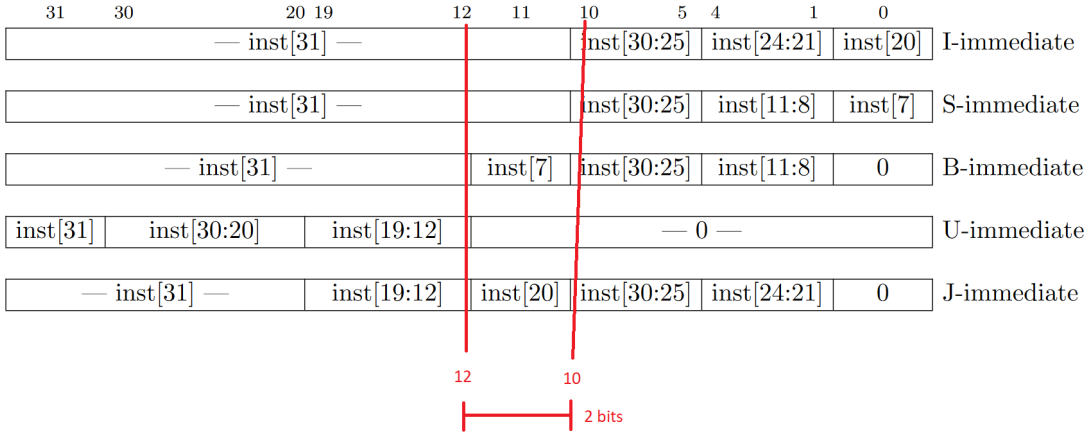
Then assert all exit codes of *diff* or *cmp* is 0, or manually add -s flag to diff and observe output like above.

This clarification could have been very helpfull if known earlier in the development and testing of the simulator. This approach to testing could have been pretty good to ensure correct behavior from an end-to-end perspective.

From hindsight, designing for testability is nice, but writing unit tests just to apporach 100% test coverage, is probably not neccesary or super usefull. Another approach could be to design with testability in mind, but only write crucial unit tests up front, and defer unit tests for when defect are found. Having end-to-end tests that test the system as a whole and most common use cases, helps alot to determine correct behavior as a whole. And when an end-to-end test fails to pass, unit tests could then be written to help remedy the incorrect behavior (ie. until end-to-end test pass).

5 Remediation of demo

In the demostration of the simulator, a defect was found which ultimately prevented one of the tests to pass because of runtime error. The test in question was "recursive.bin". The problem was with the decoding of the J immediate format. The problem's root cause was a misinterpretation of the specification. On page 17 of the official RISC-V RS32I specification, it was misinterpreted that instr[20] should be extended, in a similar manner to instr[31]. The following illustration visualize how the misinterpretation was made with a view in a hasty manner.



This misinterpretation caused an incorrect decoding implementation, which caused incorrect behavior from instructions using the J format (in this case only jal). After becomming aware of the misunderstanding, the J immediate decoding was then fixed resulting in a similar behavior to that of ripes running the same binary.

To be extra sure, "recursive.bin" was manually downloaded from the assignment repository. Simulating "recursive.bin" will similarly to ripes, get stuck in an endless loop. The screenshot prints debug information of the instructions being executed in the endless loop. The screenshot captures 2 iterations of the loop.

```
11111110110001000010011110000011 (36):  lw x15 x8 0xffffffec
00000000111100000100011001100011 (40):  blt x0 x15 0xc
00000000000010000000001110010011 (44):  addi x15 x0 0x1
00000001110000000000000001101111 (48):  jal x0 0x28
Hov! Assigning x0 -> will ignore
11111110110001000010011110000011 (52):  lw x15 x8 0xffffffec
11111111111101111000011110010011 (56):  addi x15 x15 0xffffffff
00000000000001111000010100010011 (60):  addi x10 x15 0x0
11111101000111111111000011101111 (64):  jal x1 0x4294967248
00000000000100010011011000100011 (20):  sw x2 x1 0x1c
00000001000000010010110000100011 (24):  sw x2 x8 0x18
00000010000000010000010000010011 (28):  addi x8 x2 0x20
11111110101001000010011000100011 (32):  sw x8 x10 0xffffffec
11111110110001000010011110000011 (36):  lw x15 x8 0xffffffec
00000000111100000100011001100011 (40):  blt x0 x15 0xc
00000000000010000000001110010011 (44):  addi x15 x0 0x1
00000001110000000000000001101111 (48):  jal x0 0x28
Hov! Assigning x0 -> will ignore
11111110110001000010011110000011 (52):  lw x15 x8 0xffffffec
11111111111101111000011110010011 (56):  addi x15 x15 0xffffffff
00000000000001111000010100010011 (60):  addi x10 x15 0x0
11111101000111111111000011101111 (64):  jal x1 0x4294967248
00000000000100010011011000100011 (20):  sw x2 x1 0x1c
00000001000000010010110000100011 (24):  sw x2 x8 0x18
00000010000000010000010000010011 (28):  addi x8 x2 0x20
11111110101001000010011000100011 (32):  sw x8 x10 0xffffffec
11111110110001000010011110000011 (36):  lw x15 x8 0xffffffec
```

Looking at the code in recursive.c, it should return with exit code 101. Hence, recompiling recursive.c instead of using binary file from assignment repository and running it succesfully terminates, but not with same results as from the assignment repository. I suspect that the compilation of the recursive.c has been done with different methods that has descrepencies.