02155 Computer Architecture and Engineering
Fall 2021
Final Assignment - RV32I simulator

(s183926) Anders Bredgaard Thuesen

November 25, 2021

This report contains 4 pages.

1 Introduction

The purpose of this report is to describe the design and the implementation of a RISC-V simulator. The project is a simulator for the RISC-V 32-bit base instruction set (RV32I). The simulator is written in Python larger focus on readability than performance. This is in order to make the code easier to understand and reason about both for myself and for people getting into RISC-V for the first time. Therefore the project is structured in a way that the code is (relatively) easy to understand and to modify, and not necessarily comparable to an implementation in a hardware description language like VHDL or Chisel.

2 Design and implementation

The code makes use of object oriented programming as it defines the core, register file and memory as seperate classes. This makes it easier to abstract ex. reading and writing to memory, etc. The core is implemented in the `RV32ICore` class which is responsible for initializing the register file defined in the class `Registers` and allocate 1MB of memory by initializing the `Memory` class with the contents of the input program. This makes the implementation easy to test, since none of the core functionality of the simulator depends on the code that loads, executes and compares the tests with the correct answers.

The overall implementation makes heavy use of enums for representing the opcodes and funct3 codes in a human readable way. Since the `Funct3` enum has a many-to-one mapping of bit values to enum values (probability in order to optimize hardware reuse in a physical chip) we have to make sure that we check both the opcode and funct3 code when deciding what to execute.

The `RV32ICore` exposes only one method, `execute()` which executes the program starting from program counter, $PC = 0$. This is implemented using an endless `while True` loop which fetches the instruction from memory, decodes the instruction and executes the instruction. Though this might sound like the first three steps of the classic five-stage pipeline, all instruction are executed sequentially and non-overlapping. In order to extract the bitranges corresponding to ex. result registers and intermediate values, the `Bits` class is used in order to select ranges of bits easily. This makes the decoding of the instructions quite a lot simpler than using bit masks and logical shifts and easier to implement when reading the RISC-V specification. After decoding the instruction into, opcode, funct3, funct7 and intermediate values for I, S, B, U and J instruction types using the `decode_instr` function, the `op`, `funct3` and `funct7` is used in a big switch statement in order to execute the instruction. For all other instructions than the branch and jump instructions the program counter is incremented by 4 to fetch the next instruction word in the next cycle of the loop. If the instruction is decoded as an `ecall` instruction, we stop execution and return the register file.

The register file is implemented as a list of integer types representing the 32 word registers. We make sure that the `x0` register is not writable since it should be wired to zero according to the specification. This is necessary in order to make sure that ex. `jalr x0 0(x1)` instructions do not actually write to `x0`, which caused some bugs before being explicitly handled.

Implementing the `Memory` in its own class has several advantages aswell. Since RISC-V is a little endian architecture we are able to pack and unpack half words and words in a `bytearray` using python's `struct` module. This makes it easy to implement `read_halfword(address)` and `read_word(address)` methods required for `LH`, `LHU` and `LW` instructions. The same is the case for `write_halfword(address)` and `write_word(address)` and instructions `SH` and `SW`.

2.1 Testing

The simulator was built using an end-to-end test-driven approach, meaning that the code was written in several stages in order to pass each of the given tasks. This approach simplified the development since we were able to implement and test a subset of the full simulator before implementing the rest. In order for the simulator to pass a test, the register file after execution has to be exactly equal to the binary contents of the file given as the test/task name with the .res file extension. The output of the given code passes all tests:

```
================ task1 ================
shift2.bin      PASSED
shift.bin       PASSED
addpos.bin      PASSED
bool.bin        PASSED
addneg.bin      PASSED
addlarge.bin    PASSED
set.bin         PASSED
================ task2 ================
branchcnt.bin   PASSED
branchmany.bin  PASSED
branchtrap.bin  PASSED
================ task3 ================
width.bin       PASSED
recursive.bin   PASSED
loop.bin        PASSED
string.bin      PASSED
================ task4 ================
t1.bin          PASSED
t14.bin         PASSED
t3.bin          PASSED
t2.bin          PASSED
```

```
t15.bin         PASSED
t11.bin         PASSED
t6.bin          PASSED
t7.bin          PASSED
t10.bin         PASSED
t12.bin         PASSED
t5.bin          PASSED
t4.bin          PASSED
t13.bin         PASSED
t9.bin          PASSED
t8.bin          PASSED
```

3 Discussion

In this assignment we have implemented a RISC-V simulator for the RV32I base instruction set. The simulator is able to execute binary executables compiled for any processor implementing this instruction subset. This shows the importance of implementation-independent instruction set architectures like RISC-V. Besides having practical use in debugging RISC-V programs, writing a simulator has also been useful to understand the main principles when designing a instruction set. As mentioned in the previous section, this simulator implementation has from the beginning been structured to optimize for readability and not necessarily hardware efficiency. However, during the implementation it has become more apparent how the instruction set architecture aims to enable as much hardware reuse as possible ex. when decoding instructions or executing immidiate and register addressing instruction counterparts like `ADD`, `SUB` and `ADDI` or `LB` and `SB`. One might take advantage of this and make the code more reuseable by the cost of increasing complexity. In general, implementing the simulator as a combination of several subcomponents like the core, memory and register file has turned out to be a good idea since it enforces seperation of concerns and minimize the amount of moving parts to keep in mind when changing the code or fixing a bug.