Final Assignment

- 02155 Computer Architecture and Engineering -

Cristian Botezatu
s164571

Signature

01/12/2018

Date

Simon Daniel Eiriksson
s180722

Signature

01/12/2018

Date

Contents

1 Introduction **1**
1.1 Using the simulator . . . 1

2 Design and implementation **2**
2.1 Data structures and data types . . . 2
2.2 Decoding the instruction . . . 2
2.2.1 The use of an instruction dictionary . . . 3
2.3 Instruction execution . . . 4
2.3.1 Handling the signed and unsigned integers in instruction execution . . . 4
2.3.2 Handling memory out of bounds in instruction execution . . . 4
2.3.3 Introducing the ”hardwareErrorLevel” . . . 5
2.3.4 The program counter (PC) . . . 5
2.3.5 Dealing with register x0 . . . 6

3 Simulator flow **6**

4 Discussion **7**

1 Introduction

The goal of this assignment is to implement a RISC-V simulator, which implements the RV32I instruction set. It should be testable on a x64 machine running Ubuntu. The RV32I instruction set, is the minimal RISC-V instruction set, comprising of arithmetic, logical, control flow (branch), and data load & store instructions, computed on 32-bit registers. The assignment does not include the implementation of FENCE, FENCE.I, EBREAK, CSRRW, CSRRS, CSRRC, CSRRWI, CSRRSI and CSRRCI.
We have chosen to implement our RISC-V processor simulator in Python. The main components of the simulator are:

- A dictionary containing the computer state
- An instruction decoder
- An instruction dictionary, which links the decoded instructions to their name, type and execution function
- Instruction execution, which manipulates the state dictionary

The code has been structured in:

- **main.py**: a small script which handles command line arguments and calls the main simulator.
- **run_all_bin.py**: an auxiliary script which calls the main simulator on all .bin files in a specified directory.
- **RISC_V_simulator.py**: The main file containing the core code
- **InstructionExecutions.py**: comprising the functions that execute the simulated instructions
- **HelperFunctions.py**: where the following auxiliary help functions are stored
  - bits_slice (slices a string of bits from a given instruction),
  - sign_extender (sign extends a value from a shorter bit-string to 32 bit signed integer. Used for sign extending immediate values)
  - bitmask (creates a integer with a sequence of 0 and 1s).

In the report, we are going to describe our RISC-V simulator with regards to the chosen design and implementation.

1.1 Using the simulator

The **main.py** Python script may be run from a terminal with the following command line:

```
python main.py addlarge.bin
```

This will run a simulation of the **addlarge.bin** file, assuming that it is situated in the same directory as the **main.py** file. Calling the simulator in this way will result in a text output to the terminal, revealing if the simulated program ran successfully or not (defined in section 2.3.1 via the "hardwareErrorLevel"). Further, the program output two files: **addlarge.test_res** and **addlarge.log**.

- The file **addlarge.test_res** contains the value of the 32-bit RISC-V registers, after the execution has completed.
- The file **addlarge.log** contains an elaborate output of the decoded instructions and the computer state before and after each instruction has been executed. This is used for debugging purposes.

2 Design and implementation

In this sections, the various components in the simulator will be presented. The main focus is on design issues and implementation.

2.1 Data structures and data types

For the RISC-V simulator, we have chosen to use a Python dictionary structure to contain the state of the machine with the following definition:

```
state = {
        "memorySize": memorySize,
        "memoryArray": np.zeros(memorySize, dtype=np.uint8),
        "registerArray": np.zeros(32, dtype=np.int32),
        "PC": np.int32(0),
        "hardwareErrorLev": np.int32(0)
        }
```

We have used the Numpy library for our data types, as Python's standard built-in data types do not match the processor level data types well. On the contrary, Numpy's integer types are treated as "normal" integers, in the sense that arithmetic operations are done according to their type, and casting between signed and unsigned integers are done in a bitwise manner (that is: if a signed integer is cast to an unsigned integer, it does not preserve the value of the integer, but rather keeps the binary representation).

The state dictionary contains:

- the memory size, which is used to test for memory out-of-bound errors
- the memory array, which is saved as unsigned 8-bit integers
- the register array, which is saved as 32-bit signed integers
- the program counter (PC), also saved as a 32-bit signed integer
- a variable containing a hardware error level. This was introduced to save the reason for any disruption of code execution or errors, such as memory out of bounds and others.

The first part of the simulator sets the memory size and stack pointer to the desired values, and loads the program which is to be simulated into the memory array, starting the code at memory address 0.
The simulator then starts the program counter at 0, and begins running through the program, as described in the following sections.

2.2 Decoding the instruction

When decoding an instruction, the RISC-V Instruction Set Manual Volume I: User-Level ISA[1] has been thoroughly analyzed and followed. All instructions in RV32I are 32-bit and are grouped in 6 types - "R", "I", "S", "B", "U" and "J". Based on the grouping, bit fields for the instruction's components: "opcode", "rd', "funct3", "rs1", "rs2", "funct7" and "imm" is found. Moreover, as presented in the below table[1], the relevant components specific to each type are considered.

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

For example, if considering the I-type instructions ("JALR", "LB", "LH", "LW", "LBU", "LHU", "ADDI", "SLTI", "SLTIU", "XORI", "ORI", "ANDI", "SLLI", "SRLI", "SRAI"), adding the number of the registers ("rs1" and "rd")

and the immediate fields for the decoded instructions are extracted as follows:

```
    ...
    if instr_dec["instr_type"] in ("R", "I", "U", "UJ"):
        rd = bits_slice(instruction, 7, 11)
        decodedInstruction['rd'] = rd

    if instr_dec["instr_type"] in ("R", "I", "S", "SB"):
        rs1 = bits_slice(instruction, 15, 19)
        decodedInstruction['rs1'] = rs1
        decodedInstruction['funct3'] = funct3

    if instr_dec["instr_type"] == "I":
        decodedInstruction['immediate'] = bits_slice(instruction, 20, 31)
        decodedInstruction['immediate_length'] = 12
        decodedInstruction['immediate_bin'] = bin(decodedInstruction['immediate'])
        decodedInstruction['immediate_ext'] = sign_extender(decodedInstruction["immediate"],
                                                            decodedInstruction['immediate_length'])
    ...
```

In the above case, the immediate is of length 12 and comprises bit 20 to bit 31 from the 32-bit instruction. Following the definition of the Integer Register-Immediate Instruction from the RISC-V Manual[1], we sign_extend the immediate value.
In the implementation of the simulator, the RISC-V Manual was used and the components of all instructions were identified. After tackling the implementation, we can proceed with analysing the instruction dictionary.

2.2.1 The use of an instruction dictionary

As described above, each instruction comprises of several bit fields. Some of them (the "opcode", "funct3", "funct7" fields) identifies the instruction to be executed.
The following table displays 6 instructions from the RV32I ISA, from which it appears what parts of the instruction are necessary for identifying the specific instruction:

RV32I Base Instruction Set

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20|10:1|11|19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
0000000		rs2	rs1	000	rd	0110011	ADD

When creating the simulator, we have defined an instruction dictionary, which maps each binary code instruction to its name, type and execution function.

The instruction dictionary uses an unique instruction identifier as key. It comprises of a binary value derived from the opcode, funct3 and funct7 fields together with a short string identifying which elements are included in the binary value.

Thus, the RV32I ISA has been encoded into the instruction dictionary as follows:

```
instr_dict = {
    ("o", 0b0110111): {"instr": "LUI", "instr_type": "U", "execution": exec_lui},
    ("o", 0b0010111): {"instr": "AUIPC", "instr_type": "U", "execution": exec_auipc},
    ("o", 0b1101111): {"instr": "JAL", "instr_type": "UJ", "execution": exec_jal},
    ("o3", 0b0001100111): {"instr": "JALR", "instr_type": "I", "execution": exec_jalr},
    ("o3", 0b0001100011): {"instr": "BEQ", "instr_type": "SB", "execution": exec_beq},
    ...
    ("o37", 0b00000000000110011): {"instr": "ADD", "instr_type": "R", "execution": exec_add},
    ...
    }
```

For example:

- We have encoded "LUI" as 0b0110111 because it is a U-type instruction and only the opcode field (7 bits = 0b0110111) is necessary for its identification. In this case, the unique identifier is set to ("o", 0b0110111), the "o" meaning that it only comprise of the opcode.
- On the other hand, we have encoded "ADD" as 0b00000000000110011, because it is an "R"-type instruction, and thus is identified by the 7-bit funct7 field (0b0000000), 3-bit funct3 field (0b000) and the 7-bit opcode field (0b0110011). In this case, the unique identifier is set to ("o37", 0b00000000000110011), the "o37" meaning that the instrucion value comprise of the funct7, funct3 and opcode values concatenated.

2.3 Instruction execution

For each instruction in the RV32I ISA, we have defined a function in our Python script, which accepts references to the decoded instruction and the computer state as input.
By having each instruction as a function that takes the state and the decoded instruction as its arguments, the specific instruction's components can be used for computations, while updating the state of the machine (with regards to "memorySize", "registerArray", "Program Counter (PC)", "memoryArray" and "hardwareErrorLev").

2.3.1 Handling the signed and unsigned integers in instruction execution

For example the I-type operation SRLI is tackled the following way:

```
...
# SRLI
def exec_srli(decodedInstruction, state):
    a = bits_slice(decodedInstruction["immediate"], 0, 4)
    state["registerArray"][decodedInstruction["rd"]] =
            np.int32(np.uint32(state["registerArray"][decodedInstruction["rs1"]]) >> a)
...
```

SRLI is a representative instruction for showing how we have managed the signed and unsigned bit representation of instructions.

The bits_slice function chops out the lowest five bits of the immediate value and saves it into the temporary variable "a". The aim of the instruction is to right-shift the "rs1" register by "a", but without sign-extension. That is, we should treat the "rs1" register as an unsigned 32-bit integer. The purpose of casting "rs1" as unsigned is to allow the right shifting of the unsigned 32-bit representation of the register "rs1". Further, we cast the result as a 32-bit signed integer, saving it into "rd", which is a signed 32-bit integer. These bit manipulations, assure that the right bit-value is stored in the destination register - "rd".

2.3.2 Handling memory out of bounds in instruction execution

If an instruction tries to access memory which is outside of the memory space of the machine, the simulator raises "hardwareErrorLev" = 5, which will cause the simulator to stop.
In the below code snippet, a memory address is calculated from the sign extended immediate value and the rs1 register. If the value is within a range that allows for the four-byte chunk of memory to be accessed, the memory values are loaded from the memory, left-shifted as necessary, and fitted into the rd register.

If the memory address is outside of the valid range, the "hardwareErrorLev" = 5 is triggered (as presented above) and nothing is loaded in the memory.

```
# LW
def exec_lw(decodedInstruction, state):
    a = decodedInstruction['immediate_ext'] + state["registerArray"][decodedInstruction["rs1"]]
    if a < 0 or a > state["memorySize"] - 3:
        state["hardwareErrorLev"] = 5
    else:
        state["registerArray"][decodedInstruction["rd"]] =
            state["memoryArray"][a] + (state["memoryArray"][a + 1] << 8) +
            (state["memoryArray"][a + 2] << 16) + (state["memoryArray"][a + 3] << 24)
```

2.3.3 Introducing the "hardwareErrorLevel"

We have chosen to implement our computer state dictionary with a "hardwareErrorLevel", which keeps the simulator informed of any incoherence in the program and tells if the simulated program is terminated by an ecall.

The simulator runs in a while-loop as long as the "hardwareErrorLev" = 0, and at various points in the script, the value may be changed, causing the simulation to terminate before the next instruction is fetched. The following values for "hardwareErrorLev" has been defined:

```
...
errormessages = {
    0: "Everything is good",
    1: "PC out of bounds",
    2: "The program has run succesfully",
    3: "Ends the program with return code: {}".format(state["registerArray"][11]),
    4: "Instruction not found in the dictionary",
    5: "Trying to load/store in memory out of bounds"
}
...
```

2.3.4 The program counter (PC)

After executing each instruction, if the instruction is *not* a jump/branch instruction, the program counter is incremented by 4, to let it point to the next instructions as seen below.

```
...
 # If the instruction was NOT a jump/branch instruction, we increment the PC:
if not(decodedInstruction["instrText"] in ("JAL", "JALR", "BEQ", "BNE", "BLT", "BGE", "BLTU",
                                           "BGEU")):
    state["PC"] = state["PC"] + 4
...
```

For jump/branch, each instruction itself changes the PC. For example, the J-type "JAL" instruction's execution function proceeds as follows:

```
...
# JAL
def exec_jal(decodedInstruction, state):
    state["registerArray"][decodedInstruction["rd"]] = state["PC"] + 4
    state["PC"] = state["PC"] + 2 * decodedInstruction['immediate_ext']
...
```

In this case, the PC is updated to a new values determined by its former value and the value of the immediate field of the instruction.

Before continuing to the next instruction, the simulator tests whether the PC is not out of bounds of the memory space. This is implemented by making use of the "hardwareErrorLev" that is updated from 0 to 1 - triggering the error "PC out of bounds".

```
...
if 0 > state["PC"] > memorySize - 3:
    state["hardwareErrorLev"] = 1
...
```

2.3.5 Dealing with register x0

According to section 2.1 in the RISC-V manual[1], register x0 is hard-wired to value 0. The simulator has been built such that it allows the manipulation of register x0. Thus, it is mandatory to reset it to 0 if it has been changed to another value. An alternative implementation could have been to keep it fixed to 0, so that it would not have been possible to change it.

```
...
if instr_dec["instr_type"] in ("R", "I", "U", "UJ"):
    if decodedInstruction["rd"] == 0:
        state["registerArray"][0] = 0
...
```

3 Simulator flow

In sum, when the simulator runs through the binary code, it carries out the following tasks:

- Check if the error level is 0. If not, leave the loop and print an error message.
- Fetch the 32-bit instructions, byte-wise from the memory array.
- Slice out the bits for the funct3, funct7 and opcode fields.
- Concatenate the instruction components (funct3, funct7 and opcode) into three sets of instruction values, which uniquely identify the relevant instruction.
- Match the newly derived instruction identifier to the instruction dictionary. If the instruction identifier does not match anything in the instruction dictionary, the simulator sets the "hardWareErrorLev" to 4 and breaks the loop.
- Decoding the rest of the bit fields which are relevant for the instruction type (immediate value and destination and source registers) as described above
- Execute the relevant instruction with the decoded instruction parts and the machine state as parameters.
- Increment the program counter if relevant.
- Test if the PC has a legit value (no memory out of bounds). If it does not, raise "hardWareErrorLev" to 1.
- return to the top

[1] https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

4 Discussion

In conclusion, we have successfully managed to implement a RISC-V simulator. We have tested the simulator on all test files in the course test repository[2] and all the RISC-V RV32I instruction test files provided by the author of the following repository[3].

Along the process, due to our design choices, we have faced several difficulties.

First of all, opting for Python as the programming language forced us to deal with numerous errors. Integer types have no native support in Python and, thus, we used the Numpy library's integer types. With this regards, to control integer overflow, manual type casting was needed in some cases.

Secondly, after implementing the instruction dictionary, we encountered issues with invalid instructions being interpreted as valid. In order to handle it, a short string was added to the instruction identifier. Doing so, for decoding purposes, we check the binary representation of the instruction of interest only by considering the relevant instruction's componnets ("o" - opcode, "o3" - opcode and funct3, "o37" - opcode, funct3 and fucnt7).

[2]https://github.com/schoeberl/cae-lab/tree/master/finasgmt/tests
[3]https://github.com/TheAIBot/RISC-V_Sim/tree/master/RISC-V_Sim/InstructionTests