02135 Introduction to Cyber System - Assignment 2

ISA simulator in Python

Authors

Group 8
Jianan Xu - s204698
Rune Bjerre Clausen- s204702
Simona Tican - s204703

October 13, 2021

1 Introduction

This paper describes a set of tasks performed in order to implement an instruction-set architecture simulator and test it. The simulator is realized in Python and takes *.txt* files as inputs: program and data memory. There were two tests provided as a framework, supposed to test the functionality of the generated simulator. Subsequently, it was required to write a new program and test the simulator using the initialized data memory file.

2 Implementation of ISA Simulator

The Instruction Set Architecture (ISA) simulator is mainly implemented by three snippets. The first part is the function definitions of corresponding instructions in instruction-set architecture. In addition to performing the corresponding operations, the value of *program_counter* is also updated in each function. For each arithmetic and logic instruction and data transfer instruction, the *program_counter* is incremented by 1 in each function. For control and flow instructions, it depends. In `jump(r1)`, `jump_if_equal(r1, r2, r3)` and `jump_if_less(r1, r2, r3)` functions, the *program_counter* will be assigned the specific address of the next instruction. The function `nop()` performs no operation but only increments *program_counter* by 1.

The second part is a dictionary, in which the key is OPCODE and the value is the corresponding function name. It is efficient to find out the specific function based on the OPCODE by the dictionary.

The third part is a `while` loop, in which instructions in `<program_file>` are operated by matching the OPCODE with the function and executing the function. Some information for each cycle will be printed by this loop as well. The `while` loop will be executed repeatedly when *current_cycle* is less than *max_cycles*. The `while` loop terminates in two cases. One is when the maximum cycle is reached, and the other is when the `END` instruction in the `<program_file>` is operated by calling the function `end()`. In order to terminate the simulator, the *current_cycle* is updated to *max_cycles* in `end()`. Here are the three snippets and an example of the terminal output Figure 1.

```
## Part1: The function definitions of corresponding instructions
## Arithmetic and logic instructions
def addition(r1, r2, r3):
    a = registerFile.read_register(r2)
    b = registerFile.read_register(r3)
    c = a + b
    registerFile.write_register(r1, c)
    global program_counter
    program_counter += 1
...
...
## Part2: Creating a dictionary {OPCODE:function_name}
```

```
operations = {"ADD": addition,
              "SUB": subtraction,
              ...
              "NOP": nop,
              "END": end}

## Part3: Printing some information for each cycle
##        Operating instructions in <program_file>
while current_cycle < max_cycles:
    ...
```

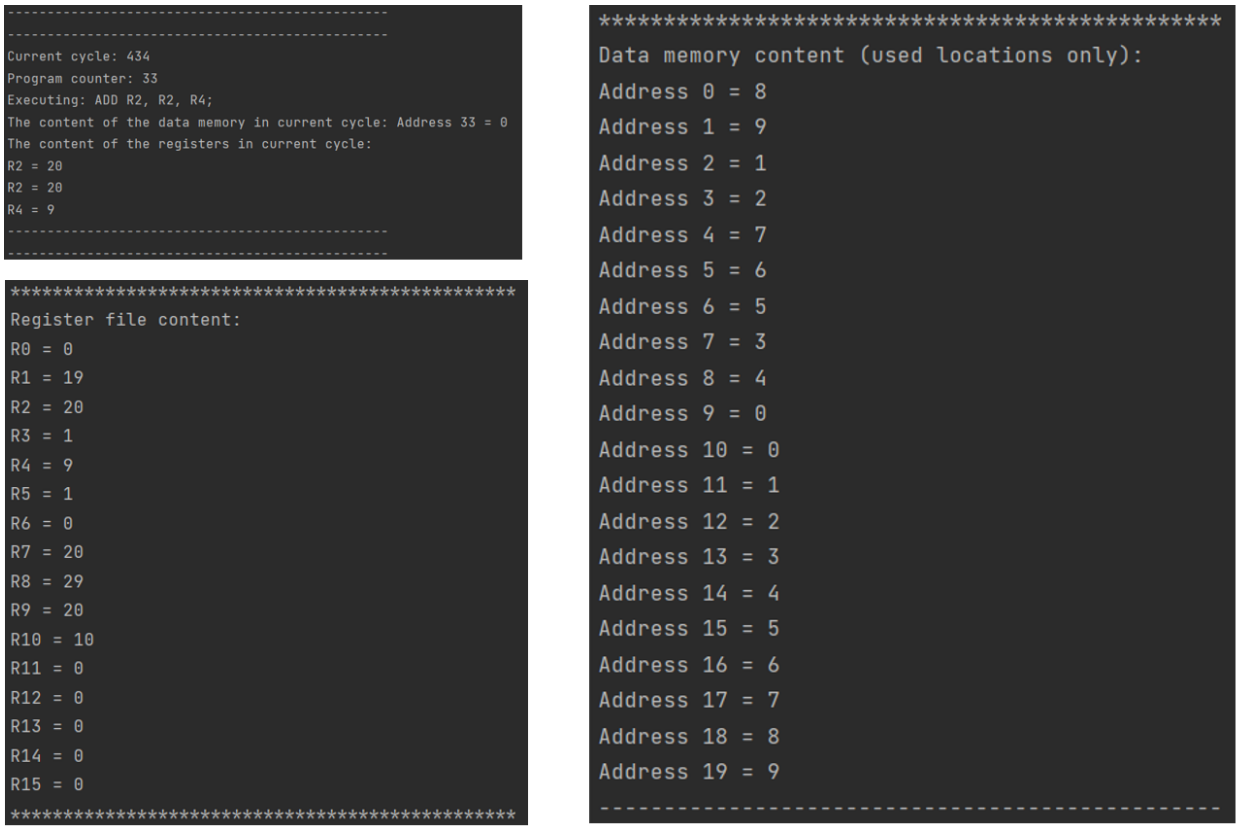

Figure 1: An example of terminal output from Test 3

3 Test 3

3.1 Insertion Sort

The third test of the ISA simulator was based on the insertion sorting algorithm. Like in test 2, the goal is to sort an array of random numbers in ascending order. An example can be seen in the figure 2 below.

Figure 2: Insertion Sorting Example

The array is divided into two parts - sorted part and unsorted part. At beginning, sorted part contains the first number of the array and the rest is unsorted part. In each round of insertion sort, algorithm takes the first number in the unsorted part as the focus number. Then it is compared to the preceding number. If it is less than the preceding number, swap them. It repeats until there is no preceding number greater than the focus number. We take the third round(3rd row) in figure 2 as an example, Here is the illustration for the third round figure 3:

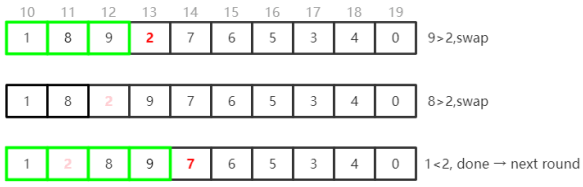

Figure 3: The third round

Above processes repeat until the unsorted subarray becomes empty. Unlike in the bubble sorting algorithm, once the focus number has been inserted into the right place in the sorted part, all numbers before the insertion place will not be compared further. Thus, it is more efficient and faster than the bubble sort. This can be seen by comparing the number of cycles for test 2 and test 3, where the same array of numbers are sorted, but at different speeds. Here the insertion sorting is done after 436 cycles, whereas the bubble sorting takes 752 cycles.

3.2 Program File

The program file simulates the instruction memory, which contains instructions for insertion sort, written in machine language. The first part is copying the array from source (data memory address $0-9$) to destination (data memory address $10-19$). It is the same as the test 2. The second part is insertion sort. There is some preparation before sorting from instruction memory address 13 to 19. In these instructions, A specific value is loaded into each register `R7`, `R8`, `R10`, `R9`, `R4`, `R1` and `R2`. The insertion sort starts at 20. If the two numbers do not swap places, the instruction jumps to 29. After the last number(stored in data memory address 19) is inserted into the right place, the instruction jumps to 35, where the program terminates. The registers `R0-R10` were used in the implementation of assembly code. The functions of `R0-R10` are shown in table 1. The contents of `R1`, `R2`, `R4`, `R5`, `R6` shown in the table are their initial values, the contents of these registers will be changed by certain instructions. But the contents of other registers are fixed.

The instructions $20-34$ is illustrated by figure 4. The instruction memory addresses are shown as 20 - 34 and the blue arrows represent the operations. The updated register contents are marked red and the sorted part of the array is outlined by green. For example, the first part of figure 4 illustrates that the instruction 20 loads the content of the data memory at the address 10(stored in the register `R1`) into the register `R5`. Similarly, 9 is loaded into register `R6`. Next, the content of `R5`(8) and the content of `R6`(9) are compared by executing instruction 22. Because 8 is less than 9, they do not have to swap places. The program jumps to the instruction memory address of "no swap" stored in register `R8`.

Registers	Contents	Functions
R0	0	R0 is read only and always contains 0.
R1	10	R1 is the read/write pointer 1
R2	11	R2 is the read/write pointer 2
R3	1	R3 is the unit to increment or decrement
R4	0	R4 is an accumulator. Every time the program jumps to no swap (instruction memory 29), R4 is incremented by 1. After R1 and R2 are reset to their initial value, they are incremented by R4. It makes sure that in the new round of insertion sort, R1 and R2 point to the first two numbers in the unsorted part of the array.
R5	0	R5 is the temporary buffer used to copy the content of R1
R6	0	R6 is the temporary buffer used to copy the content of R2
R7	20	R7 stores the instruction memory address of insertion sort
R8	29	R7 stores the instruction memory address of no swap
R9	20	R9 stores the lower boundary of the replicated array
R10	10	R9 stores the upper boundary of the replicated array

Table 1: The contents and functions of Registers R1-R10

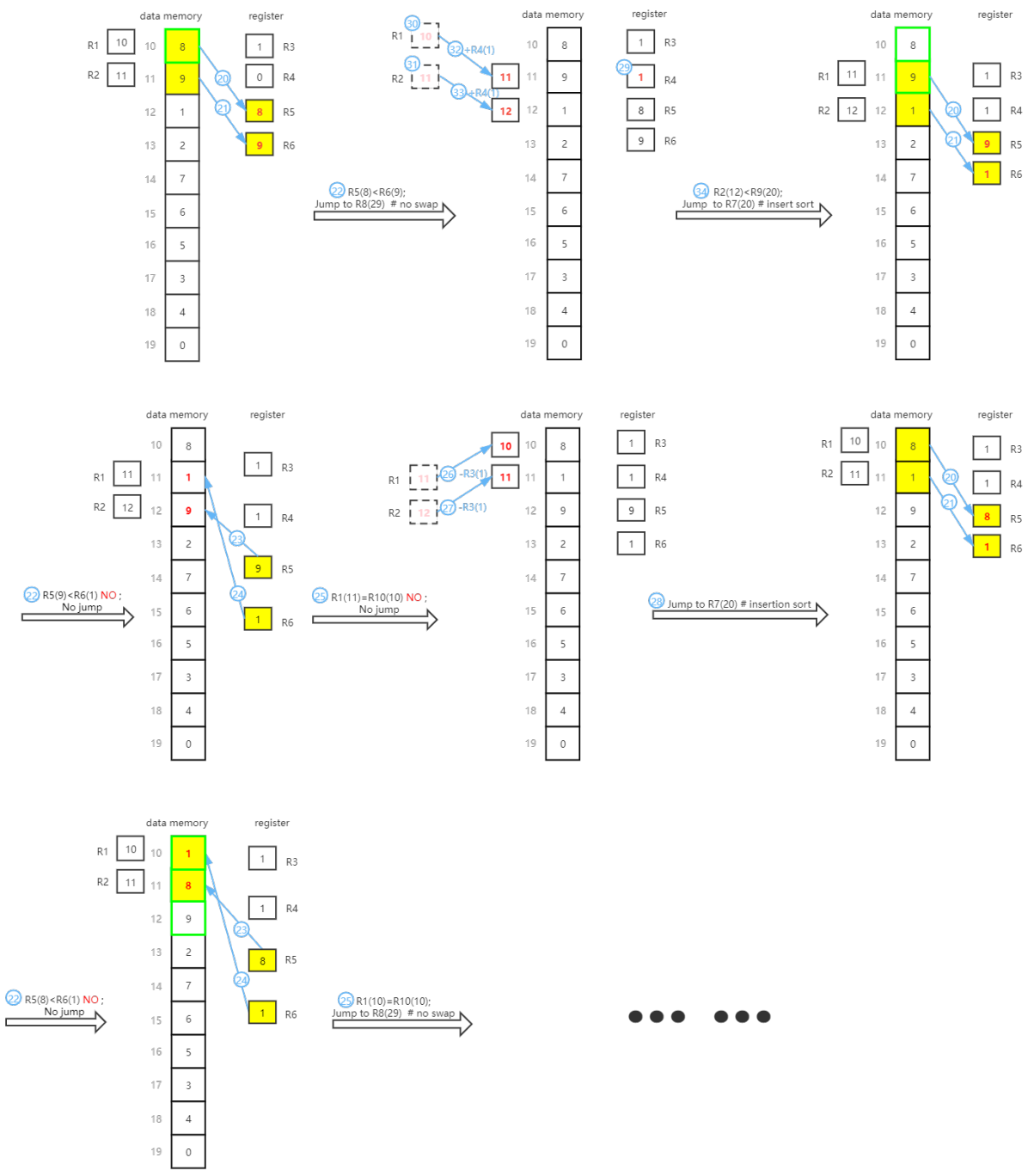

Figure 4: The illustration of instructions 20-34

4 Conclusion

Instruction-set architecture is an essential communication tool between a human being and a machine. The tasks performed in this assignment yield to a good understanding of the usage of this tool. Creating a new program based on a short list of instruction set was engaging and demanding at the same time.