CSCE 2303, 01 Project 1 Report Spring 2023

RISC-V RV32I Simulator

Omar Bahgat 900211747

Mohamed Abbas 900211252

Youssef Mansour 900212652

CSCE 2303, 01

Department of Computer Science and Engineering, AUC

Implementation Description:

Handling Input

For **instructions input handling**, we defined two main functions **'clean()'** and **'parse()'** and three helper functions **'lowercase()'**, **'removeSpaces()'**, and **'handle Parentheses()'**, that work together to handle and store a series of instructions into a format that could easily be processed and executed.

clean() function: this function takes input from a .txt file written in assembly language and it processes the file by ignoring blank lines, removing unnecessary spaces between operands and operators, and handling any operations within parentheses such as store word (sw) and load word (lw) instructions. It then returns a vector of strings where each string represents a comma-separated single instruction.

parse() function: this function takes the vector of cleaned instructions generated by the **'clean()'** function and it processes each comma-separated instruction by splitting it into individual parts such as the instruction word, destination register, and source registers. Also, if an instruction has a label, we calculate its address and store it in a map for access when jumping to the label. All the parsed instructions are then stored in a 2D vector where each instruction is represented as a 1D vector as following; ex: { [add] [t0] [t1] [t2] }

For **memory input handling**, we defined a function **'initializeMemory()'** that is responsible for initializing the computer's memory by reading values from a .txt file and storing them in a map.

initalizeMemory() function: this function takes input from a .txt file which contains pairs of integer values representing memory addresses and their corresponding values. They are then inserted as an {address,value} pair into a map called memory.

Executing Instructions

To run the program, the user enters the address of the first instruction which is validated and then the program counter is initialized to it. The main function **'runProgram()'** and the helper function **'findReg()'** are responsible for executing all instructions until a terminating instruction is encountered.

runProgram() function: this function is responsible for executing all instructions stored in the 2D instructions vector 'PI'. The function uses the value of the program counter and the initial address entered by the user to determine the index of the instruction to be executed. Depending on the instruction word such as add, xori, or beq, the function redirects the arguments to the suitable function for the instruction to be executed and the program counter is then changed accordingly. The helper function **'findReg()'** maps register names to their values to be able to access the registers when passed to the functions that execute the instruction.

Handling Output:

The **printRegisterContents()** function is a diagnostic tool that displays the contents of an array representing 32 registers. It prints a neatly formatted table that includes each register's name, number, value in decimal, and value in hexadecimal. The register's name and hexadecimal value are determined by the **regNumToName(i)** and **decimalToHex(registers[i])** functions, respectively. This function is useful for understanding the state of these registers at a given point in the execution of a program.

Bonus Features:

1. The program handles input/output in both decimal and hexadecimal formats.

Design Decisions and Assumptions:

General Program Structure:

We divided our code into two source files, one for instruction definitions and one for the rest of the code. This improved our organization as dividing the code into smaller, modular components makes it easier to understand and maintain the code as well as locate and modify it. Also, it was easier for the three of us to work on the project in parallel as each team member could work on a separate file without conflicting with another team member.

Instruction Storage Design:

We stored each instruction as a vector of strings inside a 2D vector of strings. This eliminated the need for using any extra space as we allocated memory which is just enough to store our instructions. Also, it was very easy to access any instruction within the 2D vector using the initial address entered by the user and the program counter. For example, if the initial address entered by the user is 9000 and the program counter is 9008, we can easily access the third instruction by subtracting the initial address from the program counter and then dividing by 4; (9008 - 9000)/4 = Instruction[2] which is the third instruction as we want.

Memory Storage Design:

We stored memory contents as an address-value pair inside a map. We chose to use the data structure ‘map’ to store memory contents as memory space is large (4 GBs) and we needed to use a data structure that allows us to store contents of relevant memory locations only. Also, maps allow fast retrieval of information working in O(logn).

Case-Sensitivity:

We followed the RARS case-sensitiviy conventions where register names and labels are case sensitive but instruction words are not.

- t0 != T0, x0 != X0
- loop1 != Loop1
- Add = ADD

Naming:

We followed the camelCasing writing convention to help us easily naviage and understand the codebase as well as for the code to be more readable by providing visual cues that help the user distinguish between different words in a variable or function name.

Bugs:

There aren't any bugs we have encountered so far.

Assumptions:

1. Labels and register names are case-sensitive but instruction words are not.
2. We assume no instruction and label are written on the same line
3. Memory contents in .txt file are written in the form of "address value".
4. PC counter must fall within the interval [0:2* 10^9-1]
5. 4-GB Data segment falls within the interval [2* 10^9 : 6 * 10^9 -1]

User Guide:

1. Run the solution file then follow the instructions in the terminal as specified.
2. You must have an assembly code file in text format in inside the assembly folder.
3. If you want to store data ,you must have a text file that specifies the data.

The user must enter the instructions into separate file and the entire file name to the simulator.

```
Welcome to RV32I Simulator!
Enter the name of the assembly file with the extension (ex: assembly.txt): find_max.txt
```

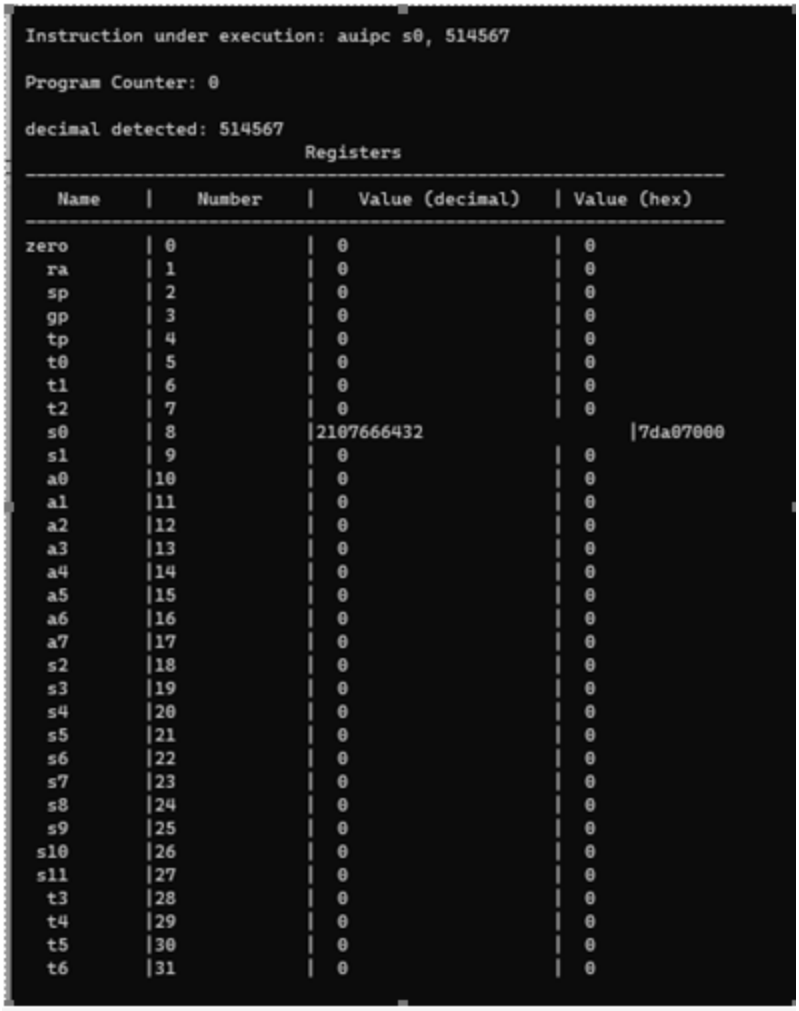
(Example)

```
auipc s0, 514567
addi t0, zero, 3
addi t1, zero, 1
addi t2, zero, 4
addi t3, zero, 9
addi t4, zero, 5
addi t5, zero, 2
sw t0, 0(s0)
sw t1, 4(s0)
sw t2, 8(s0)
sw t3, 12(s0)
sw t4, 16(s0)
sw t5, 20(s0)
addi s5, zero, 6
lw t0, 0(s0)
lw t1, 0(s0)
addi t2, zero, 1
lw t3, 0(s0)
loop:
bge t2, s5, end
slli t4, t2, 2
add t4, t4, s0
lw t3, 0(t4)
addi t2, t2, 1
blt t1, t3, update_max
bge t0, t3, update_min
beq x0,x0, loop
update_max:
add t1, t3, zero
beq x0,x0, loop
update_min:
addi t0, t3, 0
beq x0,x0, loop
end:
```

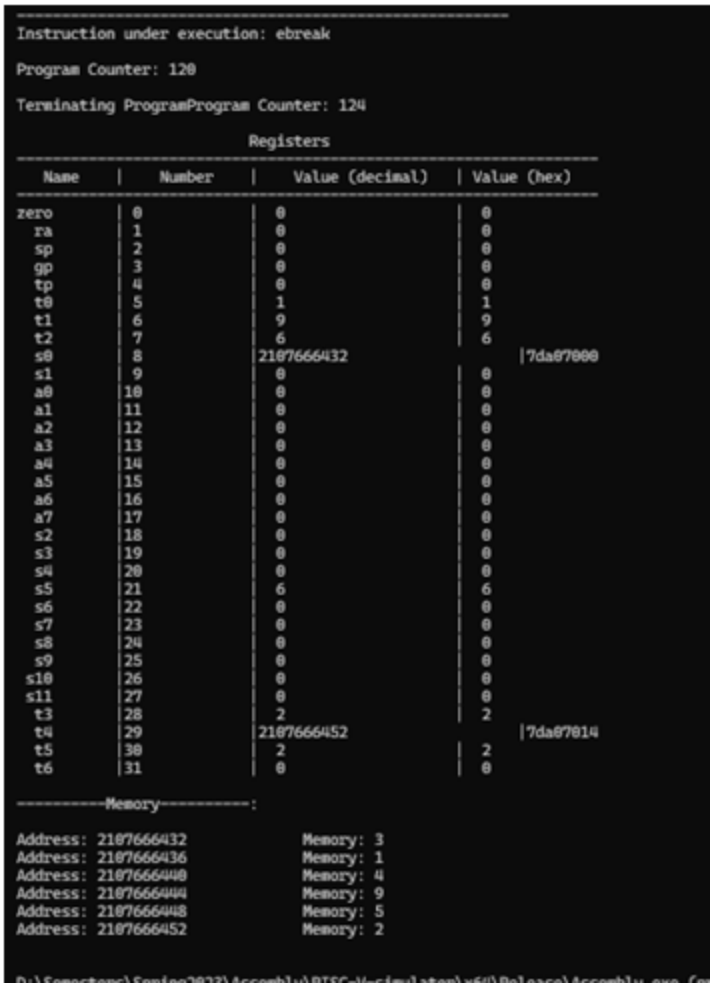
2) The program will ask the user to enter the initial address for the first instruction to be executed and ask whether you need to initialize data or not.

```
Enter the address of the first instruction (non-negative number): 0
To load data into the memory press 1, else press 2: 2
```

3) The simulator will excute each instruction and print the instruction itself, register contents, and memory contents.



4) At the end of the simulation, the registers content will be printed.



```
--------------------------------------------------
Instruction under execution: ebreak

Program Counter: 120

Terminating ProgramProgram Counter: 124

                        Registers
-------------------------------------------------------------
   Name    |    Number    |    Value (decimal)   | Value (hex)
-------------------------------------------------------------
zero       | 0            |  0                   |  0
  ra       | 1            |  0                   |  0
  sp       | 2            |  0                   |  0
  gp       | 3            |  0                   |  0
  tp       | 4            |  0                   |  0
  t0       | 5            |  1                   |  1
  t1       | 6            |  9                   |  9
  t2       | 7            |  6                   |  6
  s0       | 8            |2107666432                   |7da07000
  s1       | 9            |  0                   |  0
  a0       |10            |  0                   |  0
  a1       |11            |  0                   |  0
  a2       |12            |  0                   |  0
  a3       |13            |  0                   |  0
  a4       |14            |  0                   |  0
  a5       |15            |  0                   |  0
  a6       |16            |  0                   |  0
  a7       |17            |  0                   |  0
  s2       |18            |  0                   |  0
  s3       |19            |  0                   |  0
  s4       |20            |  0                   |  0
  s5       |21            |  6                   |  6
  s6       |22            |  0                   |  0
  s7       |23            |  0                   |  0
  s8       |24            |  0                   |  0
  s9       |25            |  0                   |  0
 s10       |26            |  0                   |  0
 s11       |27            |  0                   |  0
  t3       |28            |  2                   |  2
  t4       |29            |2107666452                   |7da07014
  t5       |30            |  2                   |  2
  t6       |31            |  0                   |  0

----------Memory----------:

Address: 2107666432             Memory: 3
Address: 2107666436             Memory: 1
Address: 2107666440             Memory: 4
Address: 2107666444             Memory: 9
Address: 2107666448             Memory: 5
Address: 2107666452             Memory: 2
```