RISC-V RV32I Simulator Report

Amer Esmail Elsheikh
ID: 900196010
amer.elsheikh@aucegypt.edu

Amr Mohammed Sallam
ID: 900196011
amrmohsallam@aucegypt.edu

Zeyad Kamal Tolba
ID: 900192983
zeyadtolba8@aucegypt.edu

Implementation Brief Description:

Our implementation was mainly composed of four parts: parsing the input, executing the instructions, handling errors, producing the output.

Parsing the Input:

The input is given mainly through two files, a memory file and instructions file. These files should be used to populate three maps, a map for memory (address to value, called memory), a map from address to the instruction string (called addressToInsruction), and a map from a label to its address (called labelToAddress). The memory map is easy to populate, but for the other two, we carefully handled distinguishing between label and instruction itself using the colon ":" and we took care of the spaces that might be there in the input using function strip.

Executing the Instructions:

After populating the maps, we started from the address of the first instruction, given by the user, and then accessed the instruction and pass it to the backbone function of the code which is called "executeInstruction". The function first parses the arguments out of the instruction, keeping in mind if it had parenthesis, then it redirects these arguments to one of the 37 functions of instructions supported. These functions then do the logic behind each instruction and change the PC accordingly. Lastly, we go execute the new instruction pointed by PC until one of the three halting instructions were met.

Handling Errors:

We made sure to handle a lot of errors either by giving the user another choice of making input again or printing the error and terminating the program depending on the type of error.

If the error is regarding the first address user gives, names of the files, or any choice the user make interactively, we ask them to input correct value again.

However, if the errors exist in the logic or syntax of their assembly program or memory file, we print the error and terminate. Some of these errors handled are:

- Repeating the same label: all labels must be unique which is assumed by RARS as well.
- Labels starting by digits will generate an error which is assumed by RARS as well.

- Putting any address in the memory file that is not divisible by 4, further explained in assumptions.
- An invalid name of register used, for example “X1” as they are cases sensitive as assumed by RARS.
- Too many instructions that can not be handled by size of PC. That is not very realistic in our case, but we handled it.
- Not supported instructions.
- Shifting left or right by more than 31 bits
- Inputting an immediate that is outside of its designated range.
- Trying to load from a non-allocated address in memory
- Storing word to address not divisible by 4 and storing half words to address not divisible by 2.

Producing the Output:

Before executing each instruction, we print the PC and the instruction associated with it (that will be executed), then we execute the instruction and print the resulting register file and memory file. We print the address of every allocated place in memory with its value and we print the names of the 32 registers each with the value in it after executing the instruction. Moreover, we gave the user the choice to choose the form of the output they need as explained in bonuses.

Bonuses And List of Programs:

We extended our project to include 2 bonus features. The first one is to enable the user to choose the output format (decimal, binary, hexadecimal). Enabling the user to choose only one format will make the output readable and neat. The second bonus feature is providing 6 more test programs along with their c programs.

The following list of programs are provided with our project:

- **General Tests**: these are 3 tests with no specific logic, but they aim to only test the 40-instruction supported by the simulation. One of them “test1” includes various loops in order to check that our simulator supports loops.
- **Sum of element in an array**: the program calculates the sum of elements in an array given its head address and length. A memory file is provided for the program.
- **Fibonacci**: this program finds the nth element in the Fibonacci series given n.
- **Minimum integer in an array**: the program finds the smallest element in an array given the array head address and length. A memory file is provided for the program.
- **GCD**: the program finds the greatest common divisor between two positive numbers.
- **Sort**: the program sorts the array.
- **String Copy:** the program copies string to another one.

Design Choices and Assumptions:

Memory architecture

In out memory design, we have chosen to store instructions and data separately. Because the memory is too big to be included, we decided to implement the memory using 2 maps, one for the instructions and one for the data. These maps link between the memory address, as the key, and the stored-in data or instruction, as the value. Also, we decided to allow the user to initiate data in memory addresses that are divisible by 4, therefore make loading data from or storing data in the memory easier and guarantee the user will not write different values on the same address.

Regarding the instructions memory, we followed the concept of alignment in memory. Loading words is possible only from addresses divisible by 4, loading half words is possible only from addresses that are divisible by 2, and the loading a byte can be from any address.

Modularity

We implemented every instruction using a separate function holding its word, such as **lw()** and **sll()**, which enabled us to better handle the huge number of code lines and make our program more modular.

Case sensitivity

We followed the case sensitivity assumptions adopted by RARS; naming labels is case sensitive (“loop” is not the same as “Loop”), but the instruction words, such as “lw”, are insensitive.

Overloading

We noticed that some instruction can be provided in two different syntax, so we decided to use overloading to overcome this challenge; instructions, such as “jal”, can be provided either by specifying a label or an offset. We overloaded the **jal** function by specifying two different signatures, one for label and one for offset.

Register file

The 32 registers are represented by an array of length 32. All registers are initialized to zero at the beginning of the program, then modified according to the instructions given. While executing each instruction, we check whether the destination register is **x0** or not. If it is **x0,** we prevent writing into it and so we preserve its value of zero.

Error handling

Our program is designed to include a lot of error handling conditions that validates everything before continuing the program execution.

Bugs:

The simulator output is exactly as expected, so no bugs exist. However, the behaviour of our simulator is built on some assumptions, that if not adopted, may result in unexpected output.

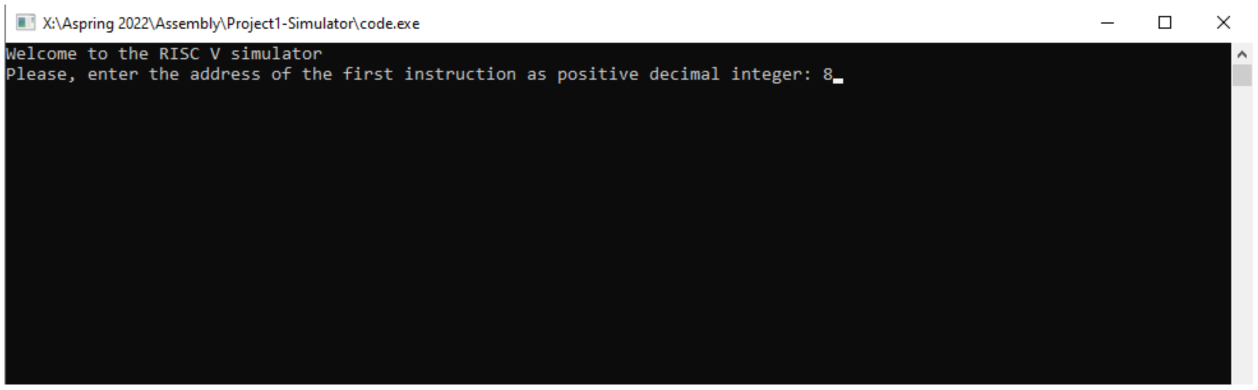
Assumptions:

1. The memory format for the data is as follows "address value". We assume that the data is represented in this format and the program will read the data incorrectly or will neglect it if the data is not in this format.
2. Comments are not supported, so we assume that no comments exist in the input files.
3. Labels are case sensitive, but instruction words are not.
4. Memory access follows the alignment concept and if not followed, an error will be thrown.
5. The memory addresses specified by the user for the data is divisible by 4. If one of them is not divisible by 4, an error will be thrown.

User Guide:

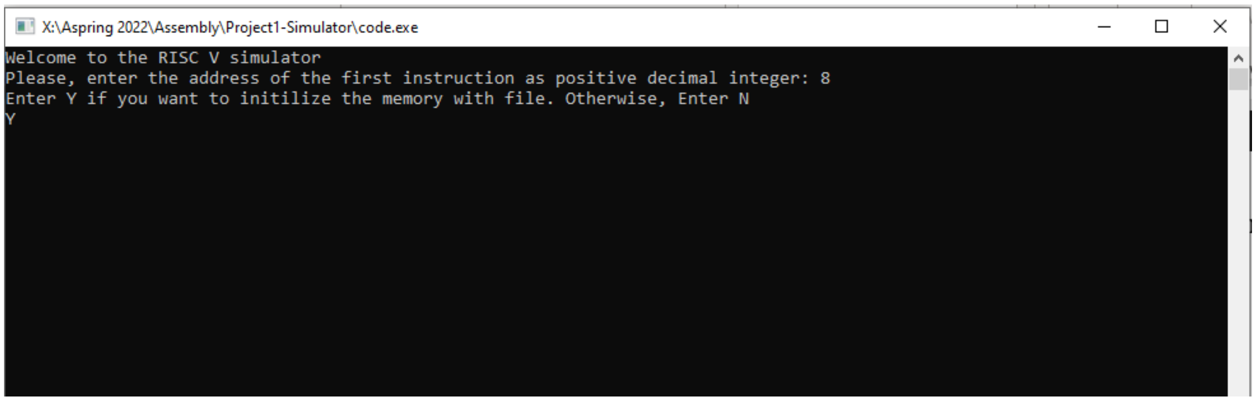
The project is written in C++, so you can compile it using any C++ compiler. We used mingw-w64 g++ compiler to compile and generate the .exe file. After compiling, you can easier use your IDE interface to run the program or simply use the generated .exe file to do that (Our .exe is provided in the submission). After that, make sure your assembly file and memory file, if you have any, in the same directory of the exe file if your will run it (or in the same directory of the cpp file if you will run the code from your IDE). The program will follow as:

1. The first thing is that you will be asked to provide the starting address of the program as a decimal integer fitting in unsigned 32 bits number and press Enter. For example, I provided 8 here.



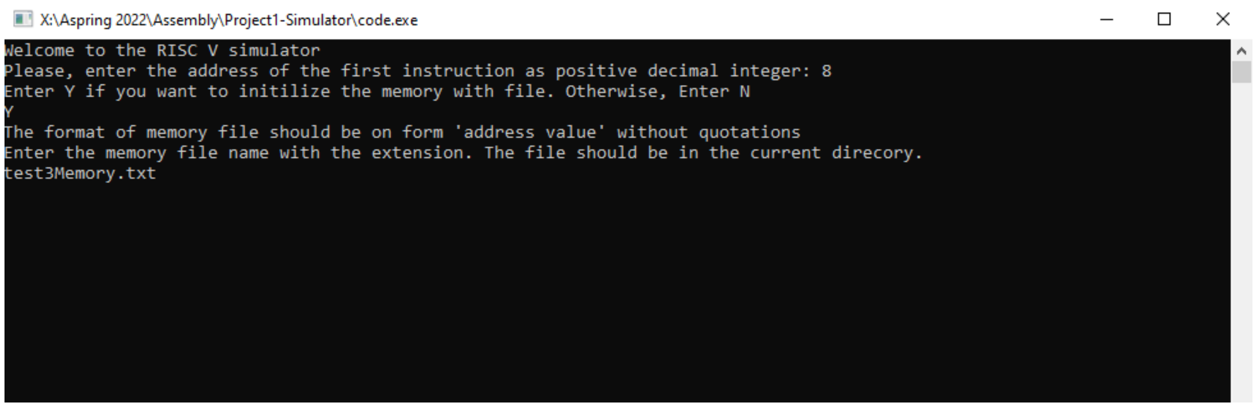
```
Welcome to the RISC V simulator
Please, enter the address of the first instruction as positive decimal integer: 8
```

2. Then, you will be asked if you would like to provide initial memory values through memory file. If you would like to do so, put the file in the same directory as described and type Y and press Enter. (Or if you want them memory to be free initially, just type N)



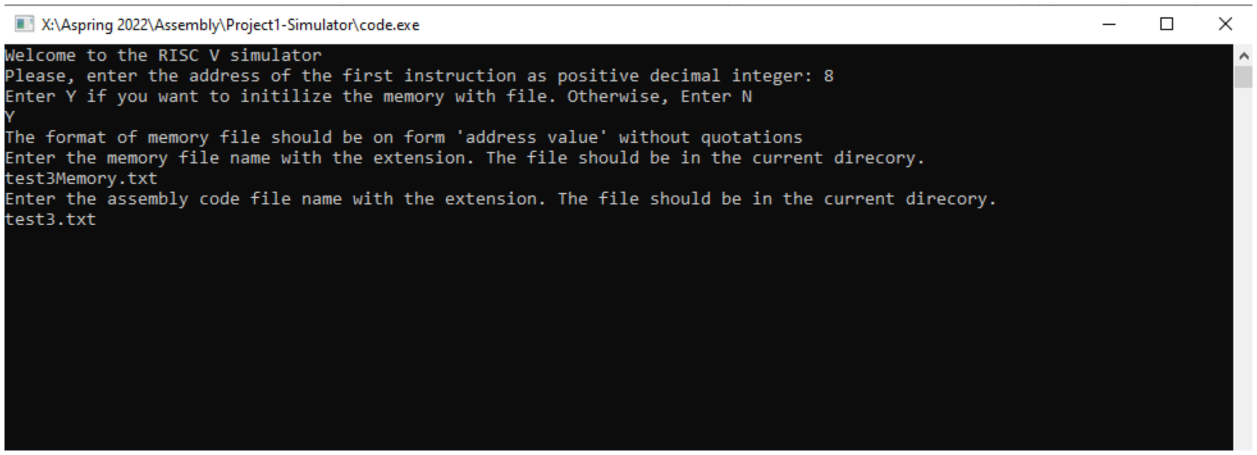
```
X:\Aspring 2022\Assembly\Project1-Simulator\code.exe
Welcome to the RISC V simulator
Please, enter the address of the first instruction as positive decimal integer: 8
Enter Y if you want to initilize the memory with file. Otherwise, Enter N
Y
```

3. Then, if you wanted to provide an initial memory file, you will be asked about its name along with the extension. For example, here I put “test3Memory.txt”



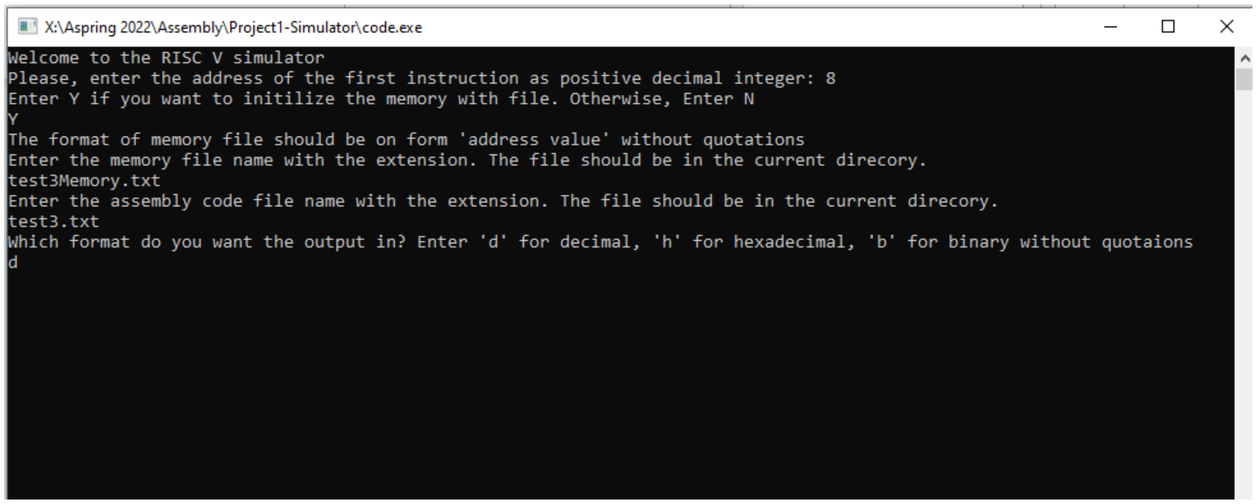
```
X:\Aspring 2022\Assembly\Project1-Simulator\code.exe
Welcome to the RISC V simulator
Please, enter the address of the first instruction as positive decimal integer: 8
Enter Y if you want to initilize the memory with file. Otherwise, Enter N
Y
The format of memory file should be on form 'address value' without quotations
Enter the memory file name with the extension. The file should be in the current direcory.
test3Memory.txt
```

4. Then, you will be asked to provide the name of the assembly file with its extension. For example, here I put “test3Memory.txt”



```
X:\Aspring 2022\Assembly\Project1-Simulator\code.exe
Welcome to the RISC V simulator
Please, enter the address of the first instruction as positive decimal integer: 8
Enter Y if you want to initilize the memory with file. Otherwise, Enter N
Y
The format of memory file should be on form 'address value' without quotations
Enter the memory file name with the extension. The file should be in the current direcory.
test3Memory.txt
Enter the assembly code file name with the extension. The file should be in the current direcory.
test3.txt
```

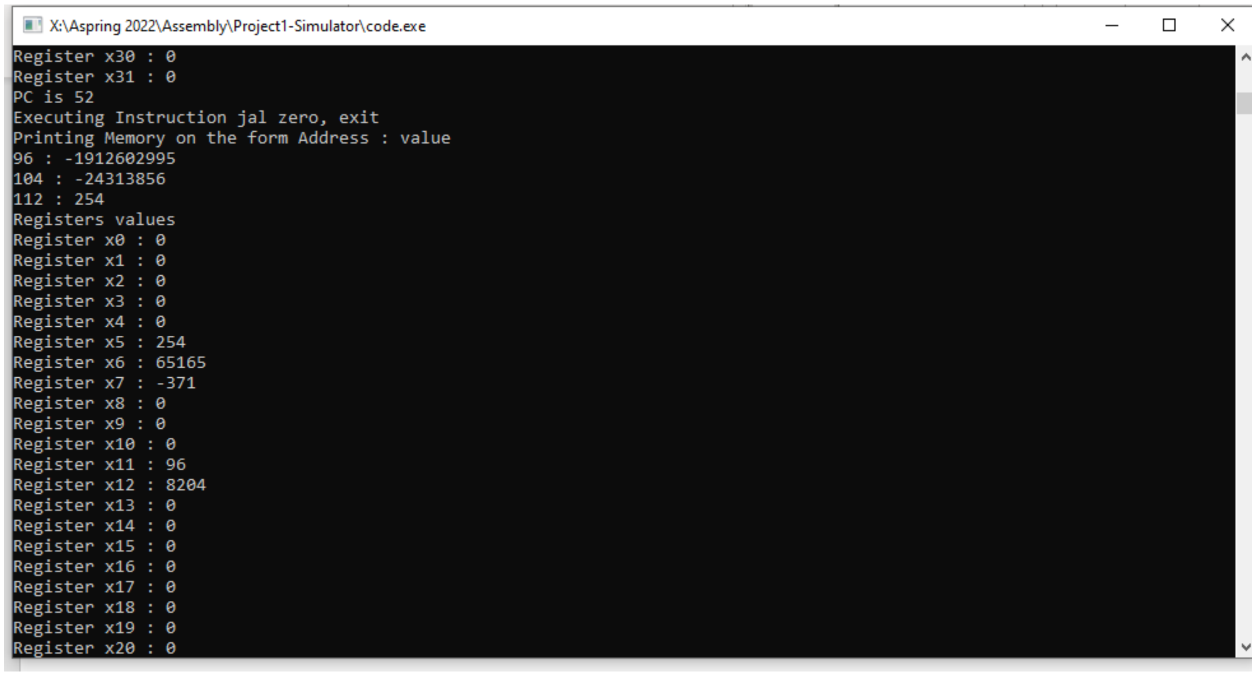
5. Lastly, you will be asked to enter which format do you want the output in. Here, we chose ‘d’



X:\Aspring 2022\Assembly\Project1-Simulator\code.exe

```
Welcome to the RISC V simulator
Please, enter the address of the first instruction as positive decimal integer: 8
Enter Y if you want to initilize the memory with file. Otherwise, Enter N
Y
The format of memory file should be on form 'address value' without quotations
Enter the memory file name with the extension. The file should be in the current direcory.
test3Memory.txt
Enter the assembly code file name with the extension. The file should be in the current direcory.
test3.txt
Which format do you want the output in? Enter 'd' for decimal, 'h' for hexadecimal, 'b' for binary without quotaions
d
```

And, now you should have the output as specified for every single instruction that looks like this



X:\Aspring 2022\Assembly\Project1-Simulator\code.exe

```
Register x30 : 0
Register x31 : 0
PC is 52
Executing Instruction jal zero, exit
Printing Memory on the form Address : value
96 : -1912602995
104 : -24313856
112 : 254
Registers values
Register x0 : 0
Register x1 : 0
Register x2 : 0
Register x3 : 0
Register x4 : 0
Register x5 : 254
Register x6 : 65165
Register x7 : -371
Register x8 : 0
Register x9 : 0
Register x10 : 0
Register x11 : 96
Register x12 : 8204
Register x13 : 0
Register x14 : 0
Register x15 : 0
Register x16 : 0
Register x17 : 0
Register x18 : 0
Register x19 : 0
Register x20 : 0
```