RISC-V RV32I Simulator Project Report

Abdallah Abdelaziz
900196083

Abdelrahman Abdelmonem
900192706

Ghada Ahmed
900183486

Description Of the Implementation

There are essentially **four sections** in our implementation: handling the input, handling the instructions, handling the output, implementing bonuses.

Handling the input

When it comes to the **input**, the user is asked for the PC, the name of the instructions file (containing the instructions), and the memory file (containing the memory address and their values) as well as the format of output (decimal, binary, ...). All these inputs (except for format) are then saved globally so that they can be easily accessed from anywhere in the program. The datatypes of the PC, instructions, memory are stored in an unsigned int, map and a map respectively. The reason for choosing the map is because each instruction and value in the memory is associated with an address (key) and it will make it much easier when we want to access any memory or instruction later.

Handling Instructions

When it comes to **handling the instructions**. We loop over the instructions using the PC and we determine the type of each instruction by reading the first part, then using a long list of if-else statements, we call appropriate functions that will emulate the given instruction. We use the PC to got to the instruction we want to execute. This makes loops and branches easier to implement as we just need to change the value of the PC. Similarly, the registers are saved in a global array, and we can easily change the value of any register according to the instruction.

Handling output

When it comes to the **output**, we have an output function which is called after executing each instruction. The function essentially loops over the registers array and the memory map and depending on which format the user chose we print the values in the correct format, the PC is also printed.

Bonuses

We implemented **two bonuses**. The first one was outputting all values in either decimal or hexadecimal or binary. Printing all formats at once would have made the output very messy, so we chose to print one format depending on the user input. The second one was to provide a set of test programs and their equivalent C++ programs.

The programs chosen are:

- Finding the GCD of two numbers.
- Insertion sort.
- Finding the minimum integer in an array.
- Total sum of elements in an array.

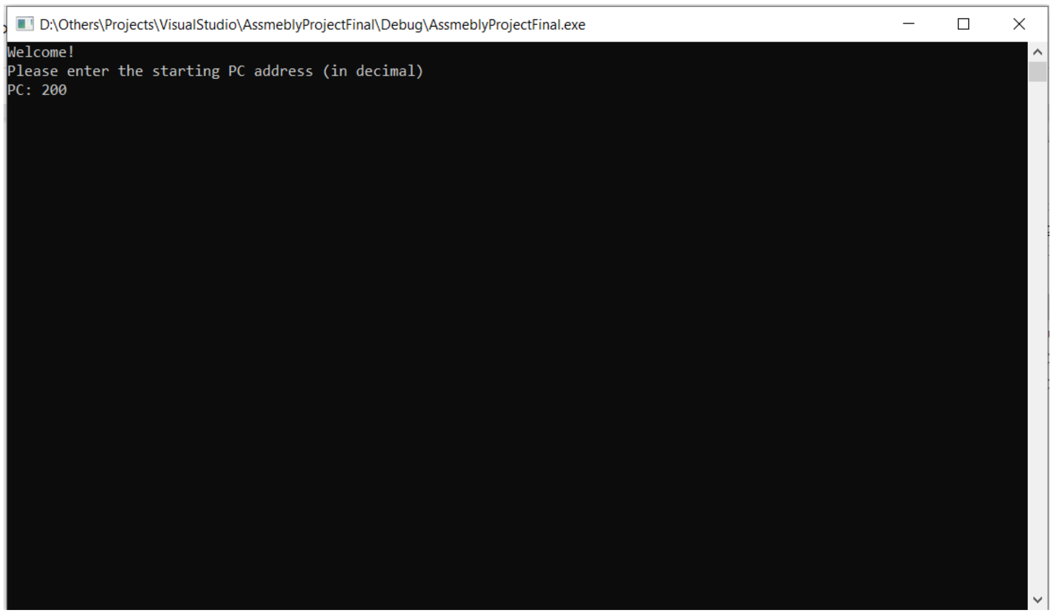
- Fibonacci term.
- String Copy.

User Guide

To run the simulator, you will need to compile the .cpp code using your preferred IDE or a Compiler. For example, you can easily copy the contents of AssemblyProject1.cpp and paste it in any C++ IDE like visual studio which will usually run the simulator after compiling.

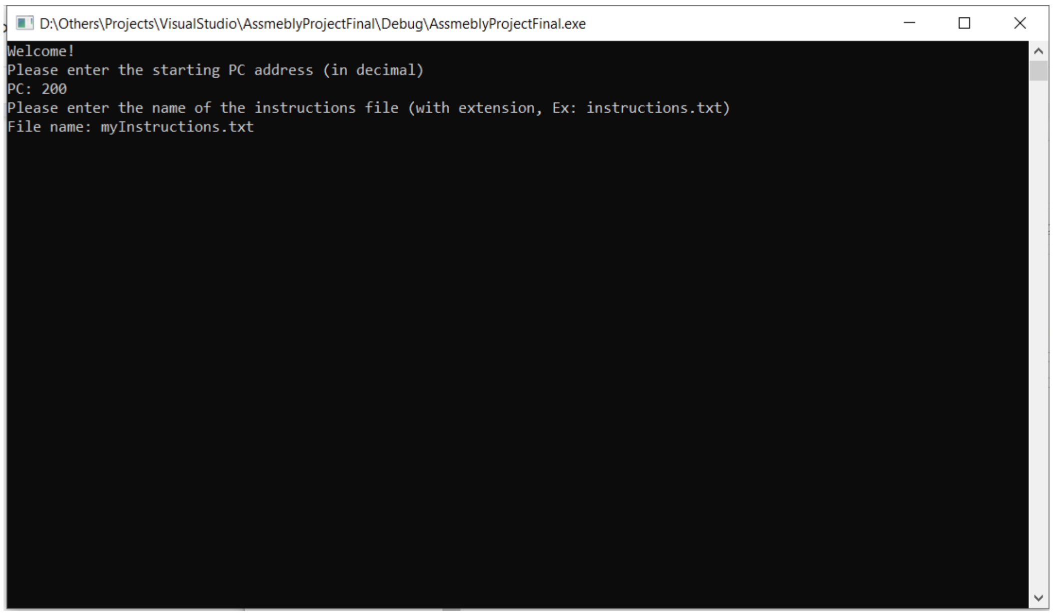
You can also use a compiler like gcc and use command lines to compile AssemblyProject.cpp and then run the executable generated by the compiler to run the simulator.

1. When you first run the program, you will be asked to input the initial PC (program counter) in decimal, after providing the PC and pressing enter. (Here, I set PC to 200)


D:\Others\Projects\VisualStudio\AssmeblyProjectFinal\Debug\AssmeblyProjectFinal.exe

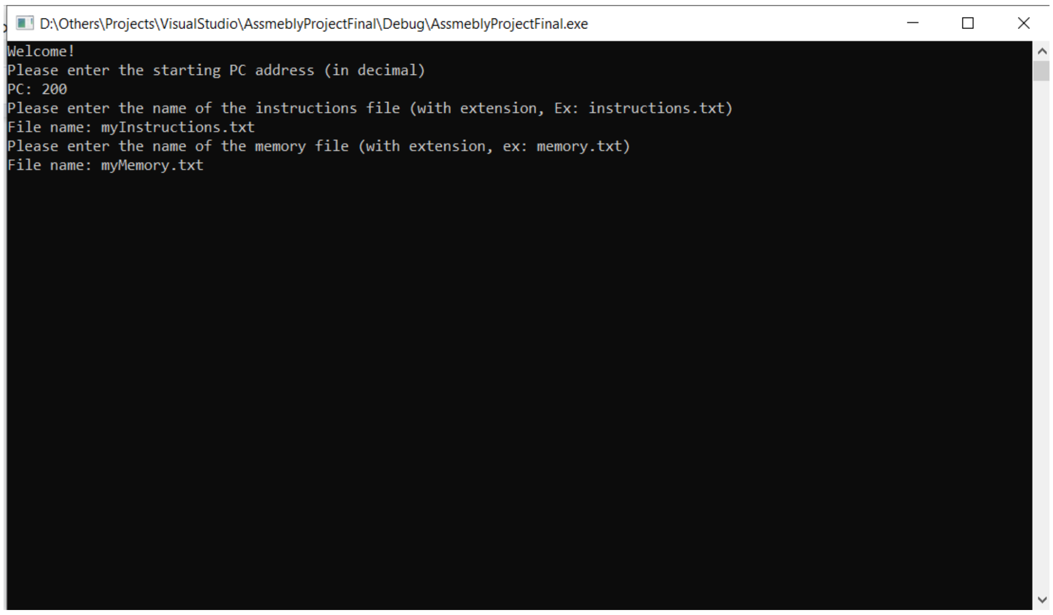
```
Welcome!
Please enter the starting PC address (in decimal)
PC: 200
```

2. You will be asked to provide the name of the file that contains the instructions that will be simulated, after providing the file name and pressing enter.


D:\Others\Projects\VisualStudio\AssmeblyProjectFinal\Debug\AssmeblyProjectFinal.exe

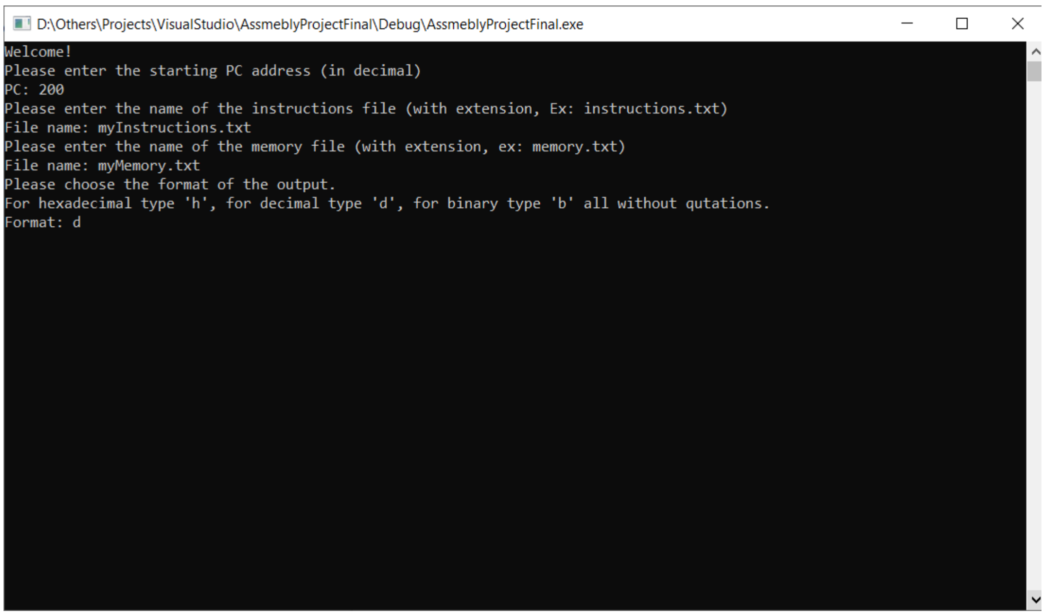
```
Welcome!
Please enter the starting PC address (in decimal)
PC: 200
Please enter the name of the instructions file (with extension, Ex: instructions.txt)
File name: myInstructions.txt
```

3. You will be asked to provide the name of the file that contains the memory addresses and their initialized values, after providing the file name and pressing enter.


D:\Others\Projects\VisualStudio\AssmeblyProjectFinal\Debug\AssmeblyProjectFinal.exe

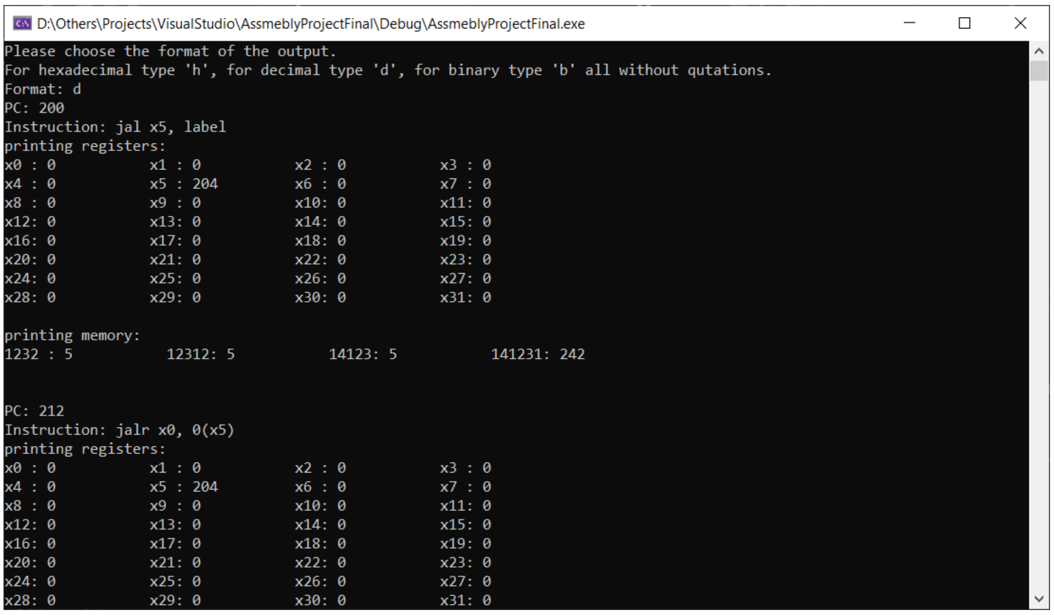
```
Welcome!
Please enter the starting PC address (in decimal)
PC: 200
Please enter the name of the instructions file (with extension, Ex: instructions.txt)
File name: myInstructions.txt
Please enter the name of the memory file (with extension, ex: memory.txt)
File name: myMemory.txt
```

4. You will be asked to provide the preferred format of the output, For hexadecimal type 'h', for decimal type 'd', for binary type 'b' all without quotations.



```
D:\Others\Projects\VisualStudio\AssmeblyProjectFinal\Debug\AssmeblyProjectFinal.exe
Welcome!
Please enter the starting PC address (in decimal)
PC: 200
Please enter the name of the instructions file (with extension, Ex: instructions.txt)
File name: myInstructions.txt
Please enter the name of the memory file (with extension, ex: memory.txt)
File name: myMemory.txt
Please choose the format of the output.
For hexadecimal type 'h', for decimal type 'd', for binary type 'b' all without qutations.
Format: d
```

5. After pressing enter the simulator will run and will output each instruction in the file and the registers and memory address after that instruction.

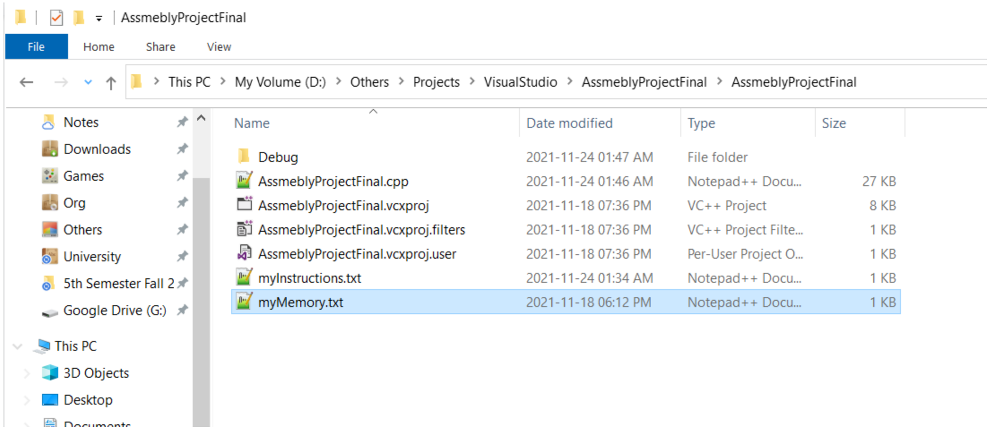


```
D:\Others\Projects\VisualStudio\AssmeblyProjectFinal\Debug\AssmeblyProjectFinal.exe
Please choose the format of the output.
For hexadecimal type 'h', for decimal type 'd', for binary type 'b' all without qutations.
Format: d
PC: 200
Instruction: jal x5, label
printing registers:
x0 : 0          x1 : 0          x2 : 0          x3 : 0
x4 : 0          x5 : 204        x6 : 0          x7 : 0
x8 : 0          x9 : 0          x10: 0          x11: 0
x12: 0          x13: 0          x14: 0          x15: 0
x16: 0          x17: 0          x18: 0          x19: 0
x20: 0          x21: 0          x22: 0          x23: 0
x24: 0          x25: 0          x26: 0          x27: 0
x28: 0          x29: 0          x30: 0          x31: 0

printing memory:
1232 : 5          12312: 5          14123: 5          141231: 242


PC: 212
Instruction: jalr x0, 0(x5)
printing registers:
x0 : 0          x1 : 0          x2 : 0          x3 : 0
x4 : 0          x5 : 204        x6 : 0          x7 : 0
x8 : 0          x9 : 0          x10: 0          x11: 0
x12: 0          x13: 0          x14: 0          x15: 0
x16: 0          x17: 0          x18: 0          x19: 0
x20: 0          x21: 0          x22: 0          x23: 0
x24: 0          x25: 0          x26: 0          x27: 0
x28: 0          x29: 0          x30: 0          x31: 0
```

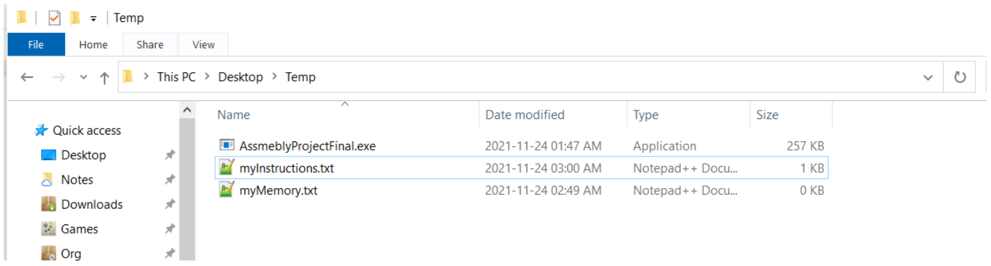
In case you use the IDE, you need to make sure that you have your two files containing the instructions and the memory values in the same directory as the .cpp file.



This PC > My Volume (D:) > Others > Projects > VisualStudio > AssmeblyProjectFinal > AssmeblyProjectFinal

Name	Date modified	Type	Size
Debug	2021-11-24 01:47 AM	File folder	
AssmeblyProjectFinal.cpp	2021-11-24 01:46 AM	Notepad++ Docu...	27 KB
AssmeblyProjectFinal.vcxproj	2021-11-18 07:36 PM	VC++ Project	8 KB
AssmeblyProjectFinal.vcxproj.filters	2021-11-18 07:36 PM	VC++ Project Filte...	1 KB
AssmeblyProjectFinal.vcxproj.user	2021-11-18 07:36 PM	Per-User Project O...	1 KB
myInstructions.txt	2021-11-24 01:34 AM	Notepad++ Docu...	1 KB
myMemory.txt	2021-11-18 06:12 PM	Notepad++ Docu...	1 KB

In case you run the program using executable, you need to make sure that you have your files containing the instructions and the memory values in the same directory as the .exe file



This PC > Desktop > Temp

Name	Date modified	Type	Size
AssmeblyProjectFinal.exe	2021-11-24 01:47 AM	Application	257 KB
myInstructions.txt	2021-11-24 03:00 AM	Notepad++ Docu...	1 KB
myMemory.txt	2021-11-24 02:49 AM	Notepad++ Docu...	0 KB

Design Decisions and Assumptions

Memory Architecture

In this design we decided to have data and instructions stored in different places; hence, managed independently. In other words, in this design we can have an integer stored in address 12 and at the same time have and instruction in address 12. Due to the size of the memory, we decided to use a **hash table** to represent the memory. The table has the address of the array as the key and the content of the memory with this address as the value. This applies to both the data memory and the instructions memory. Accessing the contents of each memory depends on instruction executed. To illustrate, the load and store instructions deal with the data memory, and the jump and branch instructions usually involve accessing instructions memory. We also assumed in our design that the code uses **valid base addresses and offsets**. A valid address depends on the instruction. For words, the sum of the base address and the offset needs to be a multiple of 4. When dealing with half words, the address plus the offset needs to be a multiple of 2. Is applies the concept of alignment in memory. In this sense, we cannot access different words nor different half words at the same time. We also assumed that the user will always enter valid addresses and data in the data file. If the user inputs negative number, we cast them to unsigned numbers.

Reading from files

In the project we read two files, the first is the instructions file and the second is the data file. Reading the instructions is quite challenging as it labels and instructions. Furthermore, instructions need to be parsed in a specific way depending on its type. Finally, we need to map registers that have the same name such as “zero” and “x0” to the same element in the array. This is done through multiple utility functions. The functions trim, lower, and parse the text according to the type of the instruction.

Reading from the data file is simple under the assumption that data is valid and that it is provided such that every line in the file contains an address and the value in this address separated by a space.

Labels

To handle labels which are essential for the branching and jumping, we created another hash table that maps labels to the address of the next instruction. In this part, we assumed that labels are case **insensitive**. We also support having the label in the same line of the instruction or in the line before it.

Register File

The registers are stored in an array of size 32, each register is of type int and hence of size 32 bits. The registers are all initialized to 0 at the beginning of the program and are modified as the program runs according to the instructions. In all instructions that modify the values of registers, we check if this register is x0. This achieves the hard wiring of x0 to 0 and prevents it from being changed when the program is running.

Summary of Assumptions

1. The input is true: we don’t validate the input coming from files; we assume addresses are valid, data is valid, instructions are free of typos, and registers used are valid ones. Instructions are also logically valid; we do not detect infinite loops nor recursive calls that do not have base cases.
2. In each line of the data file, there is an address and a value separated by a space. It is also assumed that addresses are multiples of 4.
3. We do not support comments: if the program has comments, the behavior is not well defined. In principle, if comments are in the same line as the instruction, this should not cause any problems. However, if comments are in separate lines, the program will not work. So, it is recommended to eliminate comments before running the program.
4. File names do not include spaces. For example, a file like “insertion sort.txt” will not be read properly.
5. Instructions and labels are not case sensitive.
6. As mentioned, we do not support memory access unless it is aligned. However, we do not terminate if this happens. The result will not be valid though.

Known Bugs or Issues

Under the assumptions mentioned in the previous section, the simulator behaves as expected, and no bugs were observed when testing the program.

However, if assumptions are not followed, the simulation will not give warning. It will either quit because of an exception or will give a wrong output. One of these cases is when accessing half word that starts at address 1 for example, the program will execute the instruction and will not terminate, so the user needs to verify the logic and the syntax of the code before simulating it.

List of Programs

Generic tests

We heavily tested the simulator on a variety of programs. The first set of programs are "test1.txt", "test2.txt", and "test3.txt" which are generic with no specific purpose. They aim to test all 40 instructions. To run "test3.txt" the memory file should be "test3_memory.txt".

Finding the GCD of two numbers

This program in "GCD.txt" finds the greatest common divisor between two positive integers. This program doesn't require memory files. (Just add any file the respects the assumptions about data file)

Insertion sort

This program in "sort.txt" sorts an array that is stored in the memory. The address of the array is passed in a0, and the length is passed in a1. The program runs insertion sort on the array. When the program terminates elements will be sorted in the array. For this program, the memory file is "sort_memory.txt".

Finding the minimum integer in an array

The program in "minimum.txt" finds the minimum in an array given its base address and its length. The array needs to be stored in the memory. For this program load "minimum_memory.txt" as the memory file.

Total sum of elements in an array

The program "sum.txt" finds the sum of an array given its base address and its length. The array needs to be stored in the memory as well. The memory file for this program is "sum_memory.txt".

Fibonacci term

The program "fib.txt" find the Fib_n given n. We don't need to write a memory file.

String Copy

The program copies one string into the other. The instructions are stored in "stringCopy.txt", and the memory file for this program is called "stringCopy_memory.txt". This illustrates the use of unsigned byte load instruction and unsigned byte save instruction.

Each assembly program has its C code provided with the same file name.

Resources

- The code for the trim function was written following this article.
- Some the programs used to test the simulator were obtained from our solutions to Assignment 2 and code snippets from lecture slides.