

MIPS Processor simulator

Computer architecture project

Ahmed Alaa Ahmed | CSE115 | 16P6071

Submitted to :

Dr. Cherif Salama

Eng.Ahmed Fathy

# Description

The program is a simple MIPS single cycle simulator that emulates a bit-level implementation of the MIPS processor and keeps track of the instructions , Registers, Data memory and every wire in the data path.

The program is written in Java and offers a user friendly GUI made with JavaFX and FXML to provide and easy and responsive interface.

The program contains a fairly easy to use assembler where the user can write assembly directly into it or load a file directly into the text area , then the program assembles the code into a real MIPS machine code stored in the instruction memory to be used in the data path.

Every single component of the MIPS single cycle data path was made into a class/object to be able to easily track all the hardware level changes through each cycle in various instructions going through the processor.

Each wire is made to be a JavaFX property to be bound with labels that are placed on the data path view in the GUI to make it easier for the user to track all the wires in their place.

The Register file is an array of objects of type Register that can hold a specific value , number , and binary code for tracking and checking the data on a binary level.

The registers data , current instruction , number of cycles and error reporting currently appear in a console area below the main Assembly input area but should be implemented on the data path image view in the program.

Some of the bonus features included:

* Easy to use GUI
* Interactive diagram of the data path
* A fairly east to use assembler (lacks directives and some pseudoinstructions)
* Some of the bonus instructions implemented (not all due to lack of time):
* Arithmetic : sub
* Load / Store : lh , lhu , sh
* Logic : srl , and
* Comparison : sltu , sltui

## Data path

The Program implements a tweaked data path , shown in Figure 1 , that has added some extensions to the original data path to be able to support some of the required and bonus instructions (A better picture will be provided in the program submission).

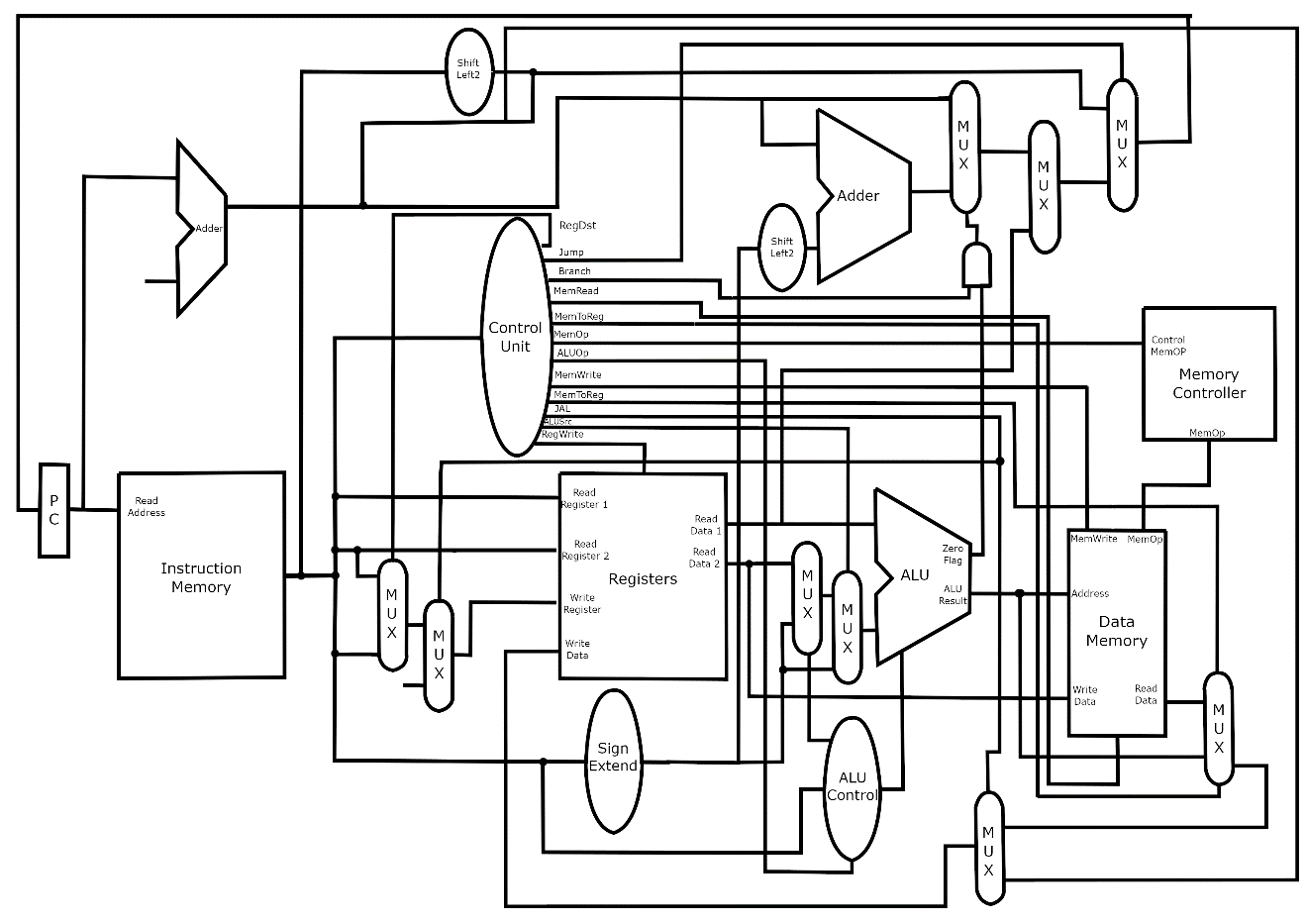


Figure 1 (Data path to the MIPS simulator program)

Some extensions include : Extra 4 Multiplexors for to support the flow of the instructions , Extra output on the Control unit (refer to control unit logic diagram Figure 2) and a Memory controller unit to support the extra memory load and store instructions.

The ALU control also has an extra outputs that control multiplexors responsible for the pathing of shifting amount and the JR instruction.

## Control unit

The control unit in my implementation got some extra outputs for supporting the extra instructions over the MIPS single cycle implementation ,including extending the ALUOP to be 3 bits instead of 2 to support extra arithmetic instructions.

Below is a Logic Diagram of the control unit (Figure 2) with inputs of the Opcodes from the instruction(0 to 5) and outputs of the control flags.

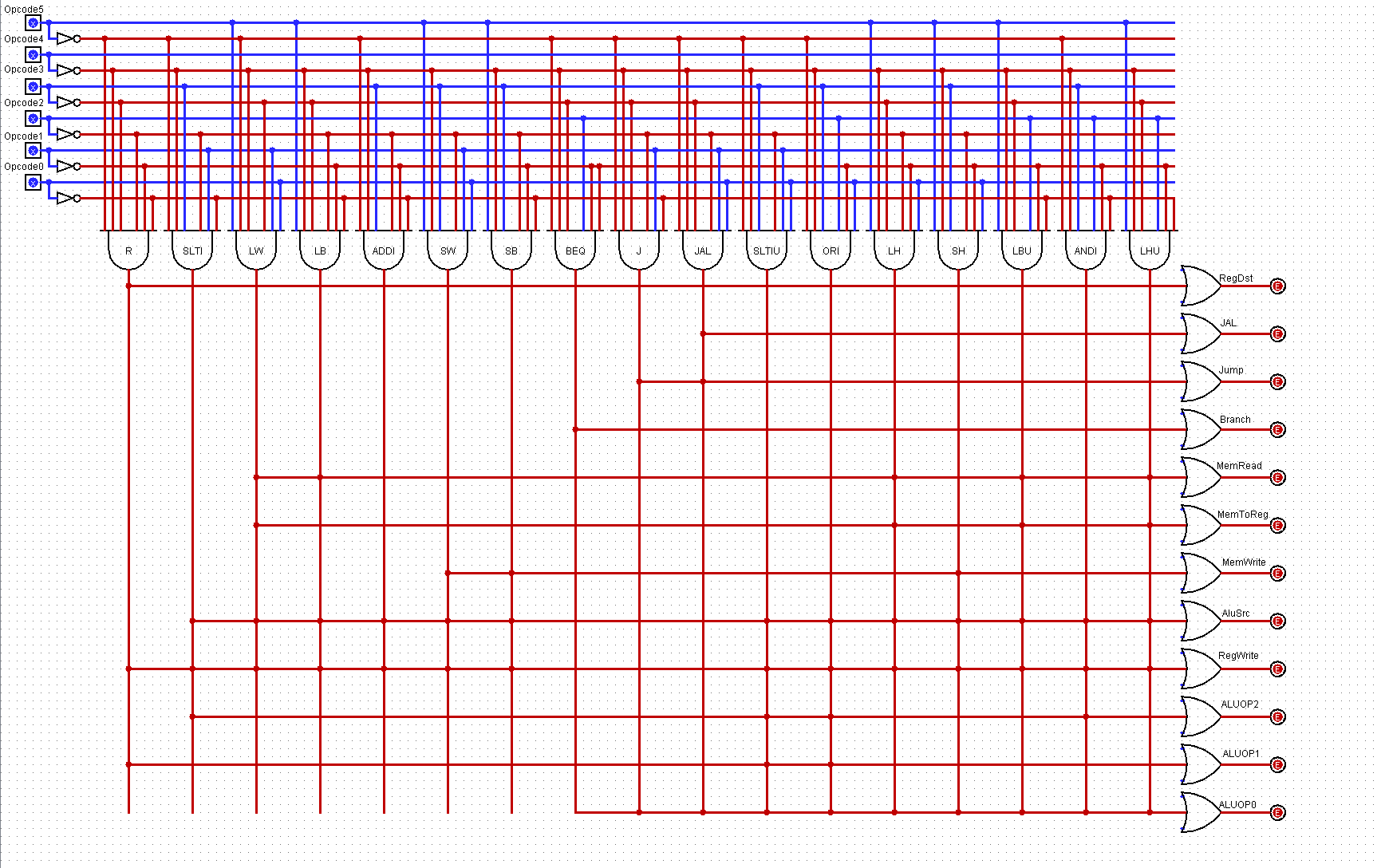


Figure 2 (Control unit's logic diagram)

The logic diagram was drawn using Logisim , the same program was also used to test some of the various inputs, however the implementation of the control unit in java wasn’t on a bit level as that would’ve costed so much time, instead it was implemented as a series of simple IF statements and Boolean outputs that are accessible by the rest of the processor (each to it’s target).

Table(1) : Truth table for the Control unit

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | RFormat | slti | Lw | lb | addi | sw | sb | beq | j | jal | sltiu | ori | lh | sh | lbu | andi | lhu |
| RegDst | 1 | 0 | 0 | 0 | 0 | x | x | x | x | x | 0 | 0 | 0 | x | 0 | 0 | 0 |
| Jal | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Jump | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Branch | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | x | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MemRead | x | x | 1 | 1 | x | x | 1 | x | x | x | x | x | 1 | x | 1 | x | 1 |
| MemToReg | 0 | 0 | 1 | 1 | 0 | x | 1 | x | x | x | 0 | 0 | 1 | x | 1 | 0 | 1 |
| MemWrite | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| AluSrc | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | x | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| RegWrite | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| ALUOP2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| ALUOP1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| ALUOP0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

Above is a quickly made truth table for the control unit it shows what are the expected output of the control unit in case of every single instruction that is supported and manipulated by the control unit.

## Assumptions

In this implementation, manipulating the memory into loading specific bytes or half words was a bit tricky , so I introduced a new component (Memory controller) similar to the ALU control unit which takes the first 6 bits in the instruction(31,26) and decodes them into a specific string of bits which controls the memory , it was made outside of the control unit to stay away from being complex and to keep the data path as clear and easy to understand as possible.

To accommodate for that , the data memory also has an extra input that takes in the MemOp code to perform the specified task in the instruction.

ALU control also takes an extra bit in this implementation because the 2 bits of the original ALUOP weren’t enough to include all the instructions without adding an extra component so passing an extra bit to the ALU control unit gave extra space for many more instructions.

Data memory is assumed to be infinite with a stack pointer pointing at ox7FFFFFF and can be decremented freely , because both the instruction and data memory are implemented using a HashMap rather than an array of bits ,just to make things a bit easier.

## User guide

The usage of the program is fairly simple , the user can just straight up write assembly code into a text area located in the program as shown in Figure 3, or load a file from outside the program with the extension (\*.txt or \*.asm) as shown below in Figure 4,5.

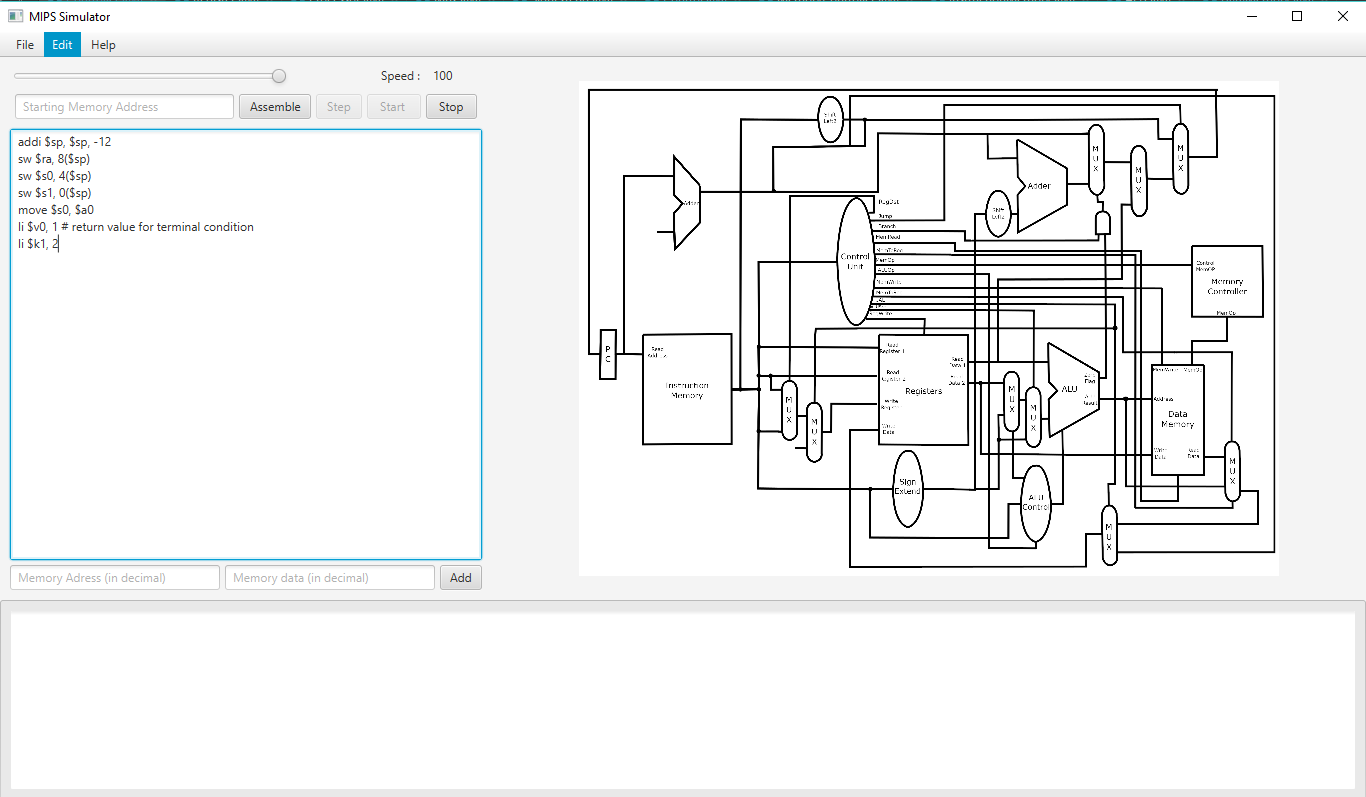


Figure 3 Main interface

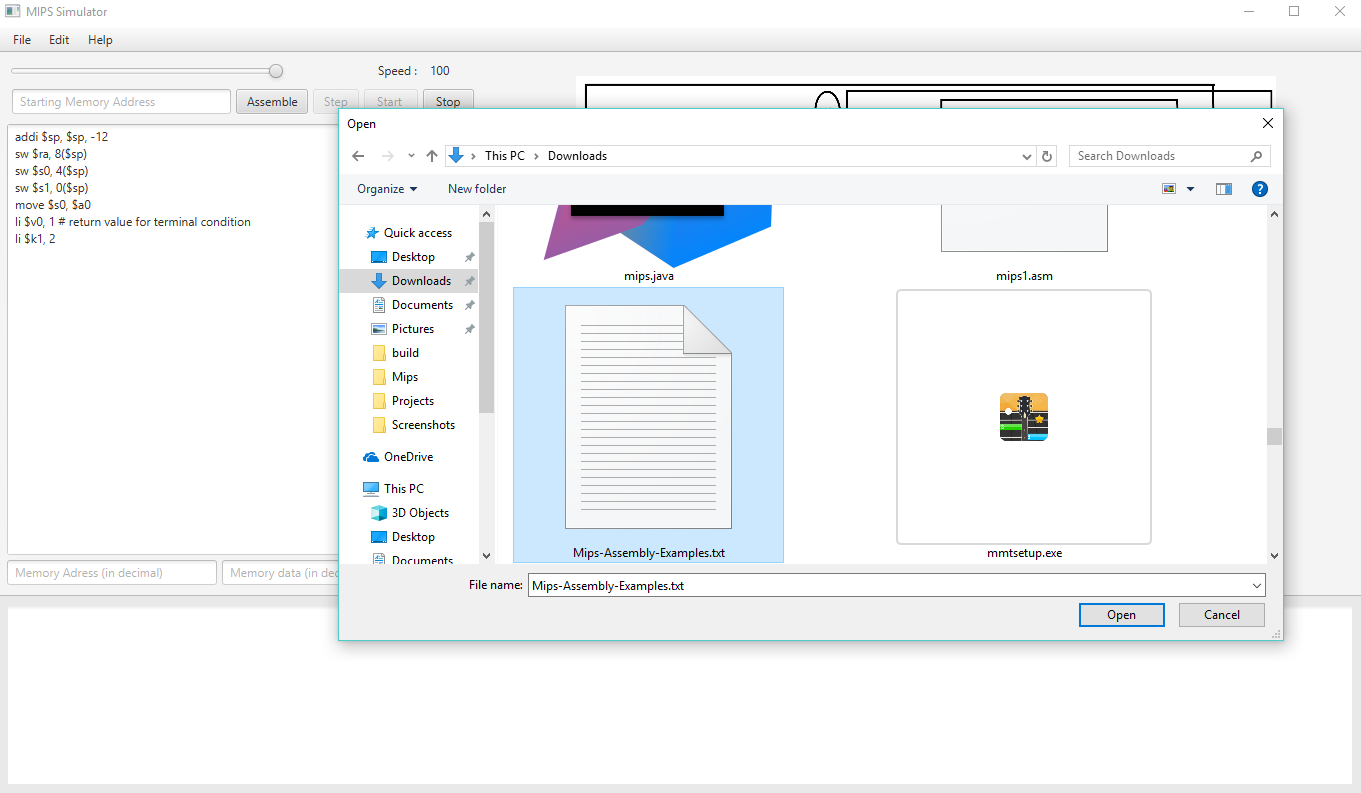


Figure 4 Loading a file

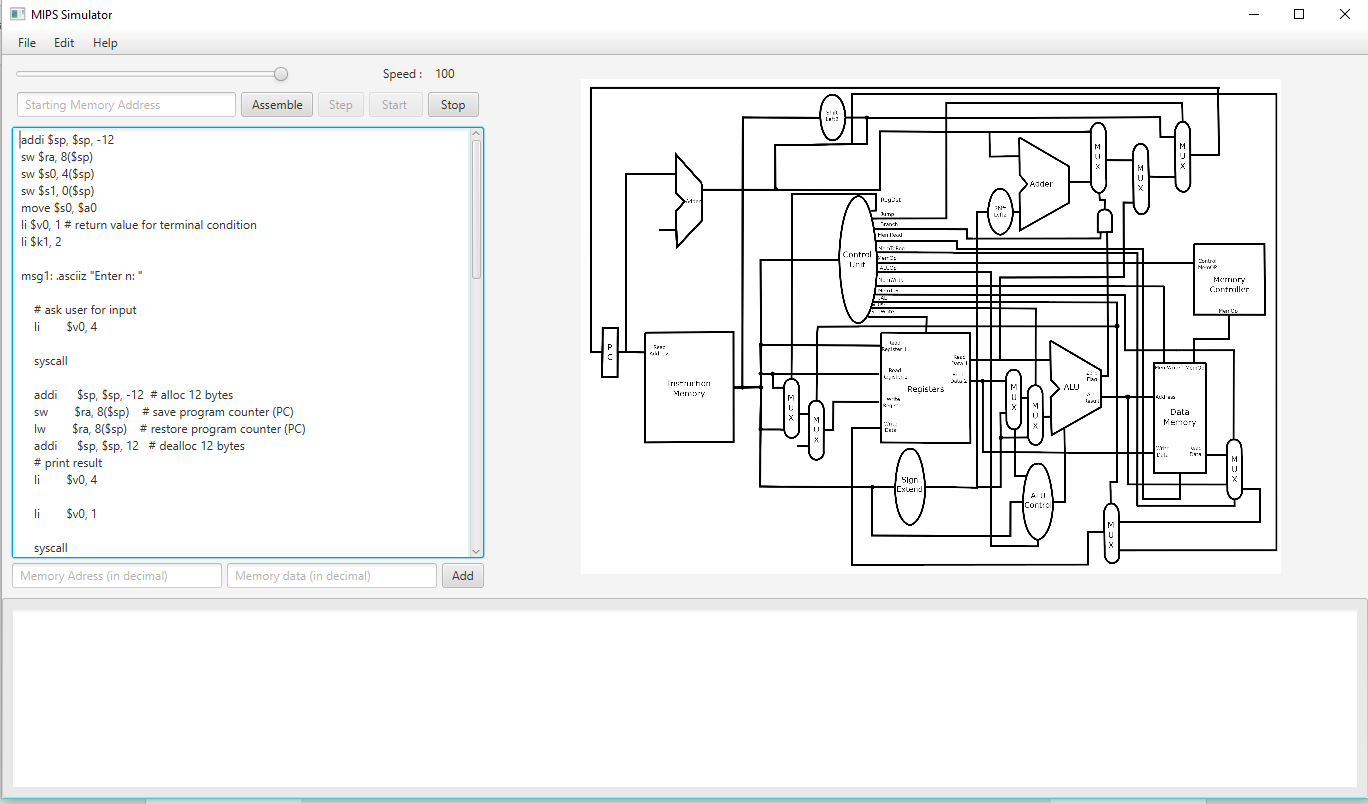


Figure 5 Loaded file in assembly area

The user can then click “Assemble” button for the program to assemble the code and enable running it , the user may then choose to either run the program step by step by clicking the “Step” button to monitor each and every change in the code or to run the whole program using the “Start button” as shown in Figure 6.

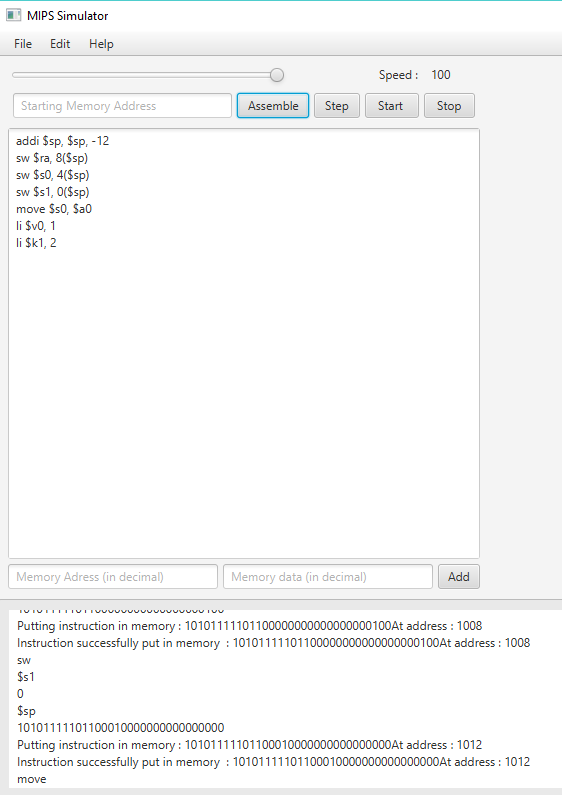


Figure 6 The starting buttons are enabled

Furthermore, the user may choose to run the program automatically step by step but with setting a specific speed at which the program runs instead of a loop that runs non stop ,this way the program can change values of the wires slowly while the user can watch and monitor each step as shown in Figure 7.

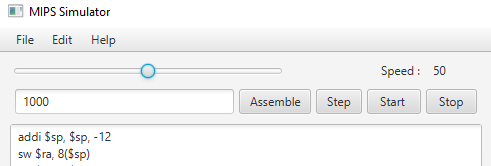


Figure 7 Speed slider changes execution speed

An optional way to load data into memory before the execution of the program is provided through 2 text fields that allow the user to load any type of data into the memory at specific addresses through clicking the “Add” button , it could help in loading extra words or bytes into memory before the execution (has to be in decimal though).



Figure 8 inserting extra data into the memory

At the execution of the program the data path shows the exact values of each wire through the execution on the wire itself (at the time of writing the report this feature is buggy so instead each wire’s value is printed out in the console located below the assembly text area along side with the register values.

# Programs tested

I haven’t had much time to include or test many big programs of my own instead I was focusing on getting everything to be functional first , so instead I checked my code using lecture codes and test cases provided thankfully by the TA.

Programs will be provided in a separate text file with the program.

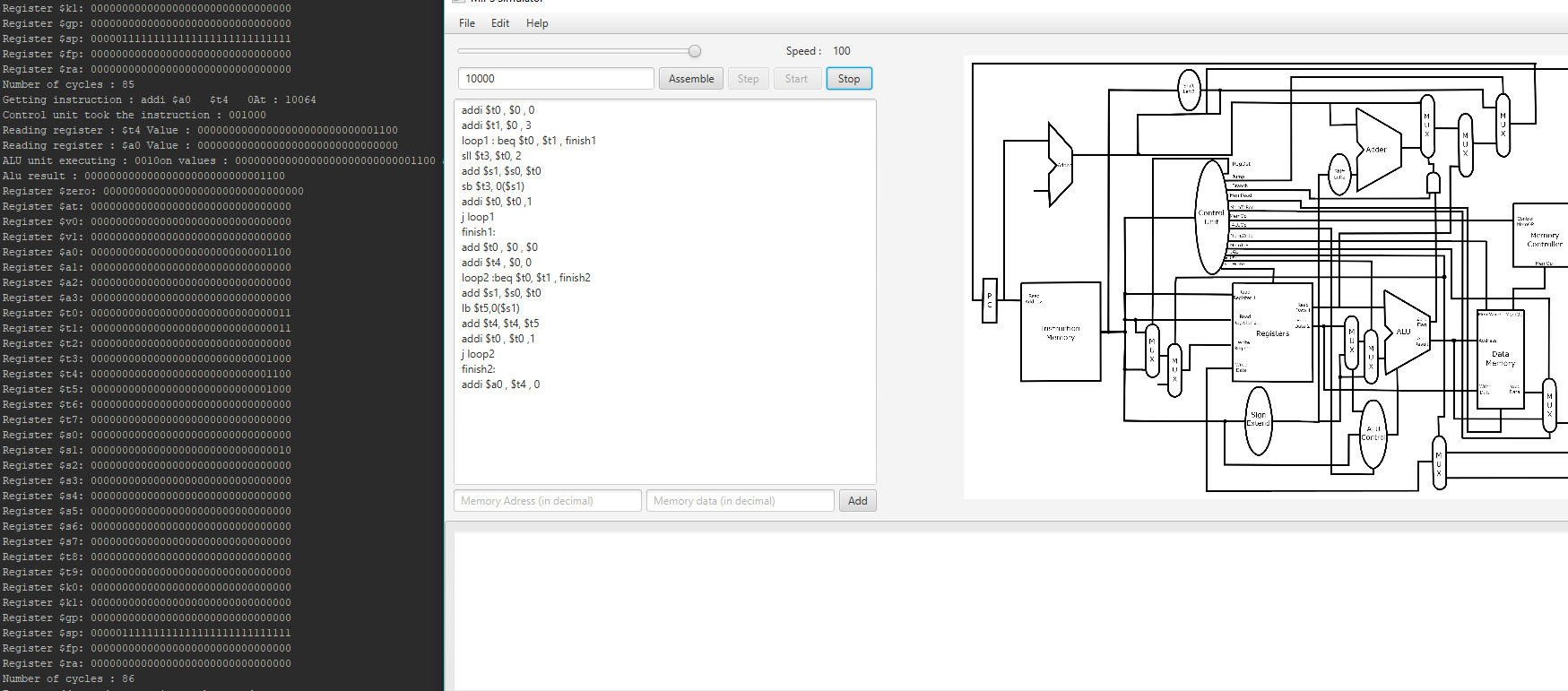


Figure 9 Program 1

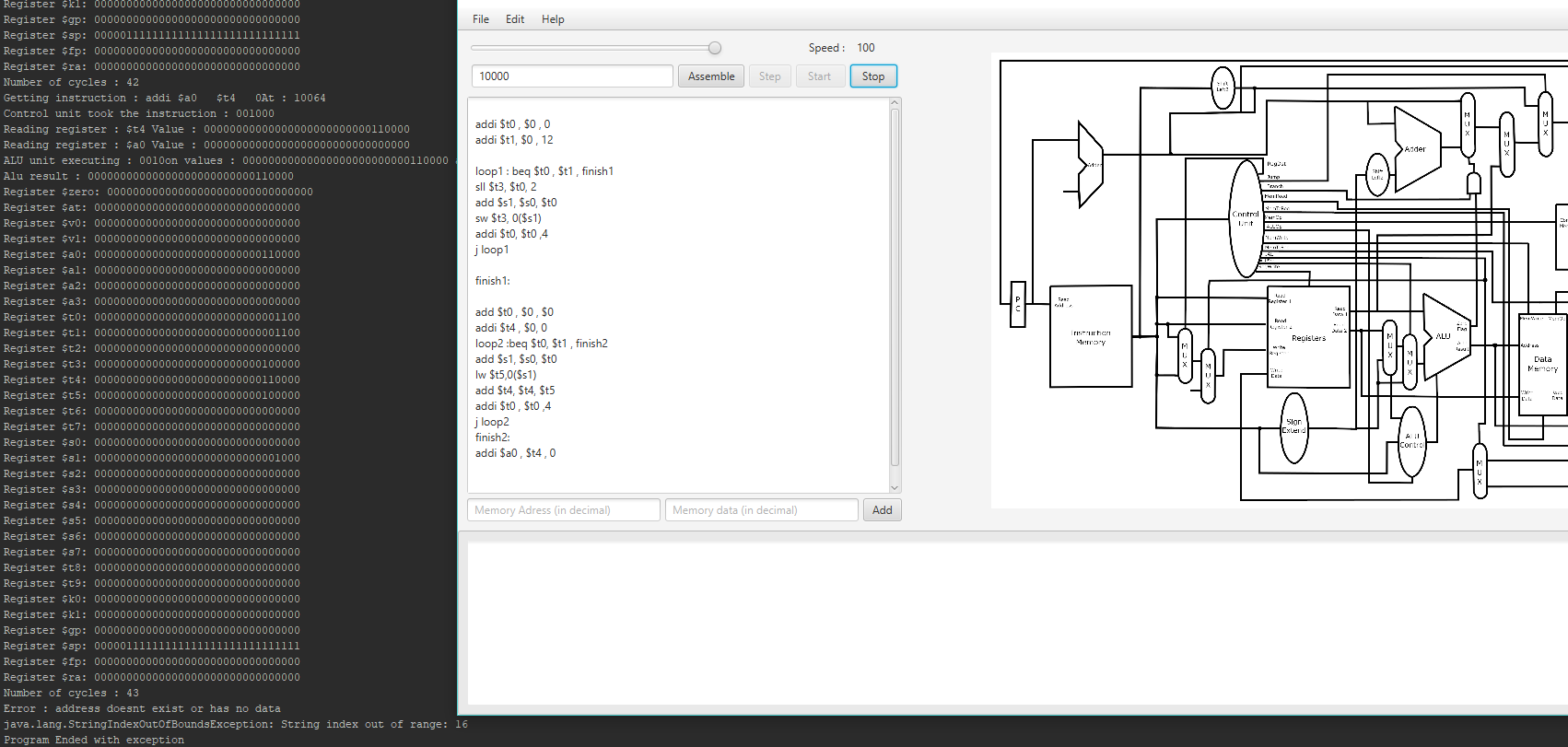


Figure 10 Program 2

Note : when I took these snapshots I hadn’t implemented the console below the Assembly area but it should be working by the submission time.

## Work split

I didn’t work on this project with a team I worked alone , mainly to challenge myself and see how much I can achieve , and to force myself to learn more and study the MIPS architecture further instead of focusing on one thing , it wasn’t easy and the outcome wasn’t as good or as close I wanted it to be but it’s working (mostly) with the required instructions , later I’ll add more features into it and keep updating it.

So there was no team to split the work among , but my commit history should show how the workflow went from the beginning [: https://github.com/InEdited/MIPS-Simulator](https://engasuedu-my.sharepoint.com/personal/16p6071_eng_asu_edu_eg/Documents/:%20https:/github.com/InEdited/MIPS-Simulator)

Note : The repository was kept private till the time of submission so that no other team can read my implementation , after submission the repository will be public so you can check it and check the commit history and make sure of my progress through it.

## Work Experience (optional)

As I mentioned earlier I have worked on this project completely alone , it was hard but I might argue that it was worth it.

(I’ll be writing this section day by day so things might change till the day of submission)

I tried to make this project a web based simulator using javaScript and ReactJS framework to make it easier and to achieve the second bonus requirement by making it an animated datapath , which I almost succeeded in I even could put the picture of the data path on a local website and made it interactive with mouse hovering but the further I would go into that project implementing the processor itself the harder it would become so I found out that if I keep going I won’t be able to make it in time of the project submission for the grades.

Moved on to a second idea I started out with python and pyQt as I have an acceptable knowledge in python and it was gonna work out well but I found out that it would be kind of too easy to be implemented and wont teach me almost anything new.

I started to go back into making it in java again , but I decided that I need to learn something new as well so I decided to use JavaFX and FXML to make the GUI and interface , it turned out pretty nice even though it was hard because there’s hardly any sources to learn JavaFX online so I had to dig in a lot.

I drew the data path from scratch to add my own extensions which had me to use GIMP which was hard to use at first and burned out much time but it turned out pretty nicely.

This had to be the hardest project I have worked on because of the hardware implementation of it , also the second longest and I’m willing to keep adding into it more whenever I’m free and keep the github repository updated.

This project was coded was lots of soda and pizza, screaming at the screen often too.