CSCI6461 Project Part IV

Project: Basic Machine Simulator

Abhinava Phukan
abhinava.phukan@gwu.edu

Dev Shah
devshah7@gwu.edu

Natalie Rachel Jordan
nrj@gwu.edu

AlHassan Halawani
hassanhalawani@gwu.edu

April 1,2023

Abstract

To gain a deeper understanding of the design, structure and operations of a computer system, principally focusing on the ISA and how it is executed. The Simulator built here emulates small classical CISC computer which executes certain kind of instructions and how it handles memory. This allows us to understand the Macro and micro structure of the CPU.

1 Introduction

In order to run the jar file for Windows:

- Ensure You have jdk-17 or above installed in your system.
- You have Java Development Kit Binaries location put as a reference for your command line arguments (System Environment Variables)
- Run The runsim.bat fil to start the Machine Simulator.

To run the jar file using a Mac system:

- Have jdk-17 or above installed in your system.
- Run the shell script provided to run the Machine Simulator

The use of a contrived CISC computer model facilitates understanding of the complex macro- and micro-structure of the CPU, including the components not visible to the programmer. Through simulating instruction execution, the model enables learners to better comprehend the underlying mechanisms of computer operation.

2 Overview

This section covers the structure of the program, its components & features and a working demonstration of the example program.

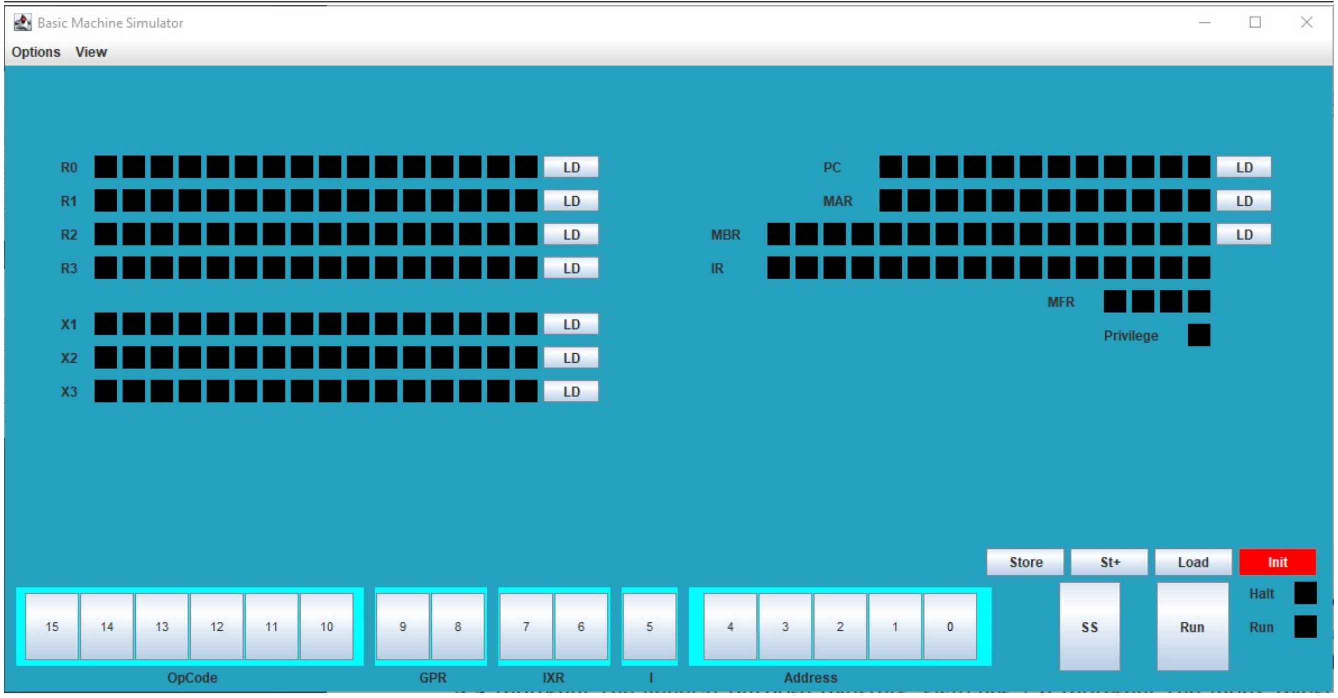


Figure 1: Gui Interface

2.1 Understanding the Interface

2.1.1 Switches

The computer simulator is designed to feature a 16-bit switch that enables the user to input binary values in a simple and intuitive manner. The switch is labeled from 0 to 15 in descending order, with each switch representing a specific bit in a binary number. The switch is segmented into separate sections that correspond to different components of a computer instruction. Switches 15-10 represent the opcode, switches 9-8 represent the general purpose registers, switches 7-6 represent the index register for addressing, and switch 5 represents the indirect bit for pointer addressing. Switches 4-0 represent the immediate address switch for accessing immediate addresses nearby.

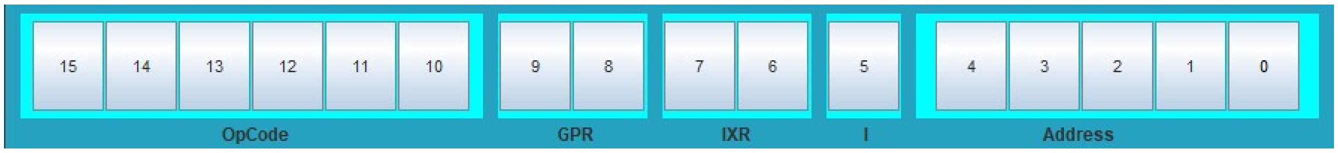


Figure 2: Switches

By toggling these switches, the user can specify the instruction to execute, the registers to use, the index register to use for addressing, and whether to access data directly or indirectly. The 16-bit switch is a versatile and powerful tool for simulating computer instructions, and its user-friendly design facilitates ease of use.

2.1.2 Load Button

The computer simulator's graphical user interface (GUI) features Load buttons that are located adjacent to each register on the interface. These buttons allow the user to input data in the form of a 16-bit binary number , which can be used for various purposes, such as specifying the value of a register or providing input for an instruction.

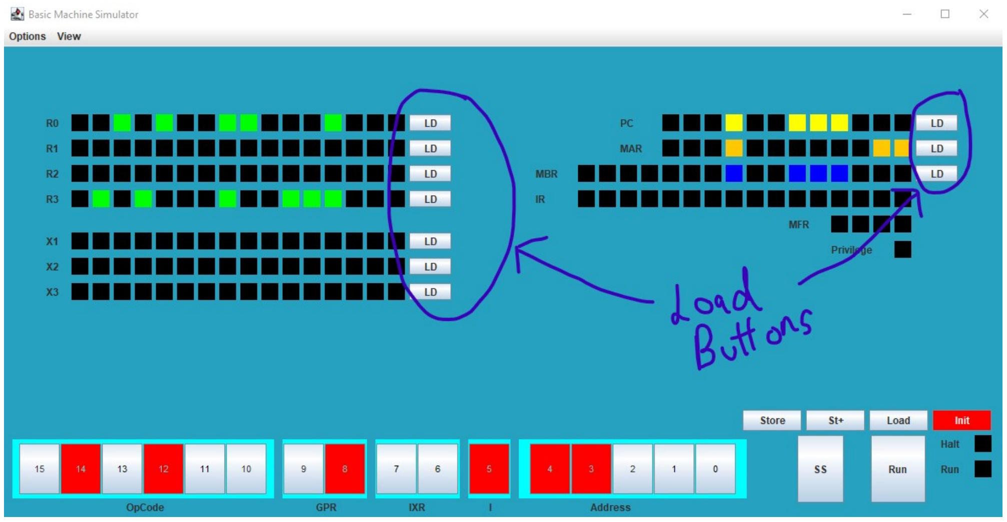


Figure 3: Load Buttons in Gui Simulator

However, it should be noted that the Instruction Register (IR), Memory Fault Register (MFR), and Privilege (1 bit) registers do not have corresponding Load buttons. This is because the IR and MFR registers are used for internal functions of the simulator, while the Privilege bit is used to determine whether a user is in a privileged mode or not.

The binary number is input by toggling the switches provided, which represent the 0s and 1s in the binary number. Once the desired binary number is displayed on the switches, the user can click the LD button provided to load the binary number into the corresponding register. This makes it easy for the user to input data into the simulator and work with the registers efficiently.

2.1.3 Control Buttons

The simulator is designed to provide users with six buttons that allow them to interact with the program directly. These buttons are crucial in storing and loading data, initializing the system, and executing instructions. Here's a breakdown of each button's functionality:

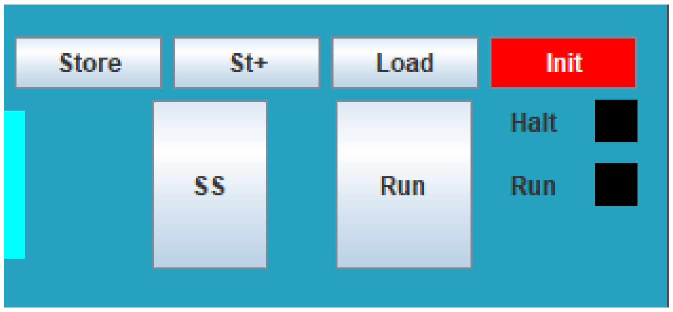


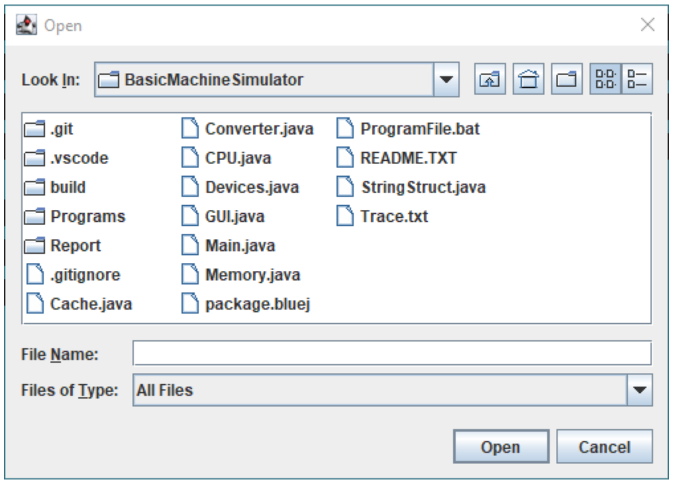
Figure 4: Control Buttons in Gui Simulator

1. Store Button: This button allows users to store data into the Data array of the system's memory. When the user clicks on the Store button, the simulator reads the value in the Memory Address Register (MAR) and the value in the Memory Buffer Register (MBR). It then uses the value stored in the MAR, which is a 12-bit index value, to locate a specific memory location in the Data array. Once the location is found, the 16-bit value in the MBR is stored in that location.
   i.e., Data[value(MAR)] = value(MBR).

2. Store+ Button: This button works similarly to the Store button but with an additional feature. After storing the data, the MAR is automatically incremented by 1, which allows users to continuously store values in sequence.

3. Load Button: This button allows users to load data from the Data array into the MBR. When the Load button is clicked, the simulator reads the value in the MAR, uses it as an index value to locate a specific memory location in the Data array, and loads the value stored in that location into the MBR.

4. Init: This button opens a file dialog that prompts the user to provide the location of the program that needs to be run on the simulator. Once the program is loaded, the instructions or other values are loaded into the Data array based on the memory location specified in the file.



.git	Converter.java	ProgramFile.bat
.vscode	CPU.java	README.TXT
build	Devices.java	StringStruct.java
Programs	GUI.java	Trace.txt
Report	Main.java	
.gitignore	Memory.java	
Cache.java	package.bluej	

Figure 5: Init button on pressing gives file dialog

5. SS: The Single Step (SS) button takes a peek at the Program Counter (PC) Register, which stores the address of the next instruction to be executed. It then loads the value in the PC into the Instruction Register (IR) and executes the opcode, allowing the user to execute instructions one by one.

6. Run: This button runs the program stored in the memory location specified by the PC. Users can manually set the entry point of the program using the Load (LD) button. The program will continue to run until it encounters a halt instruction that brings the system to a halt. These buttons provide users with an intuitive way to interact with the simulator, allowing them to perform various tasks efficiently.

2.1.4 Menubar Buttons

In addition to the buttons on the GUI interface, the menu bar in the simulator has additional features to enhance the simulation experience. The first feature, labeled as "Options", provides two options to the user. The first option is to reset the halted state of the machine, allowing the user to continue using the simulator. Once the machine is halted, the user cannot use the simulator unless the halt is reset. The second option in this menu is to reset the entire memory of the simulator, which zeros the data array and sets the register state and its bits to all zero, giving the user a fresh start.

The second feature in the menu bar, labeled as "View," provides the user with additional options to view the console monitor, console keyboard, and the cache. Additionally, users can attach virtual devices to the simulation to further enhance their experience. Overall, these features in the menu bar provide users with more control and flexibility in using the simulator, making it a powerful tool for learning and experimenting with computer architecture.

2.2 Structure of the Program

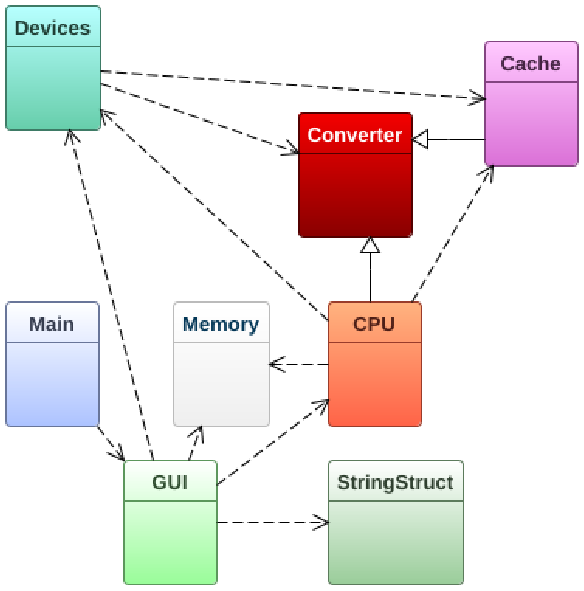


Figure 6: Class Diagram of the Simulator

- The program starts with the main class as the entry point, and upon running, it invokes the GUI from the GUI class. The GUI class is responsible for creating the user interface and is highly dependent on other classes to perform the simulation effectively.
- To run the simulation, the GUI class relies heavily on the CPU, Memory, and Devices classes. These classes are responsible for executing the instructions of the simulated machine, storing and retrieving data, and interacting with various devices that the machine may have.
- To run the simulation, the GUI class relies heavily on the CPU, Memory, and Devices classes. These classes are responsible for executing the instructions of the simulated machine, storing and retrieving data, and interacting with various devices that the machine may have.
- In addition to the core classes, the program also includes several helper classes to perform specific tasks. The Converter class is used to convert binary numbers to decimal and vice versa. This conversion is a crucial component of simulating a computer, as all data is represented in binary form.
- Similarly, the StringStruct class exists to facilitate custom string operations that may be required during the simulation. This class may be used to format data or display messages to the user.
- Another essential component of the simulator is the Cache class, which is responsible for storing simple lines of data in a first-in, first-out (FIFO) fashion. This caching mechanism helps speed up the simulation by reducing the number of times the program needs to access data from memory.
- Lastly, it is worth noting that the Converter class is extended by both the CPU and Cache classes. This extension ensures that all classes within the program can perform the necessary conversions with ease, allowing for a seamless simulation experience.

2.3 Running The Program

Following will be the instructions to run on a packaged jar file:

2.3.1 For Windows

1. Extracting the zip file and open the directory.
2. Ensure that you have java installed as per conditions provided on Introduction.
3. Look for a batch file called runsim.bat. It will run the program.
4. To run the program click on the init button that would bring the file dialogbox.

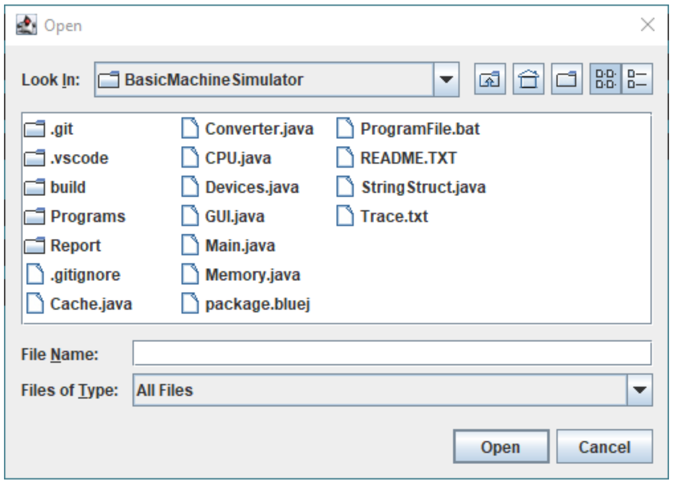


Figure 7: Activating File Dialog Box

5. Navigate the Program folder and choose which program you would like to run.

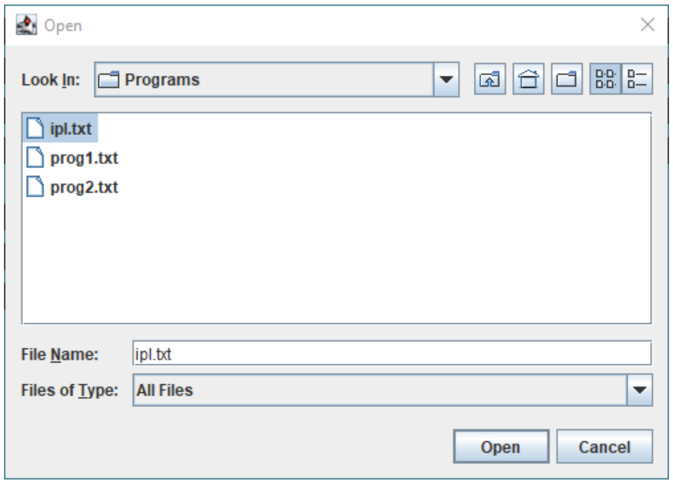
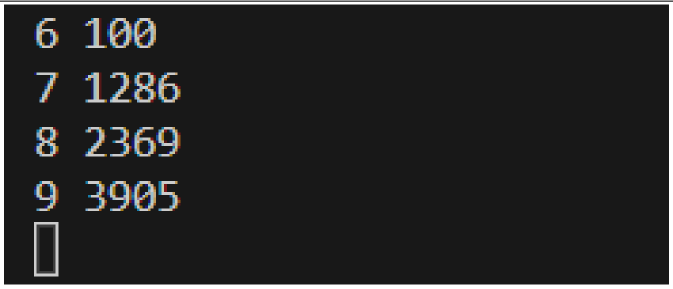


Figure 8: Choosing ipl.txt file as demo

6. On Accepting the file click ok then you will see in the console the lines of code that has been loaded into the memory.



```
6 100
7 1286
8 2369
9 3905
```

Figure 9: Loading Code

7. Press those switch buttons to set the value where entry point of the program is located.

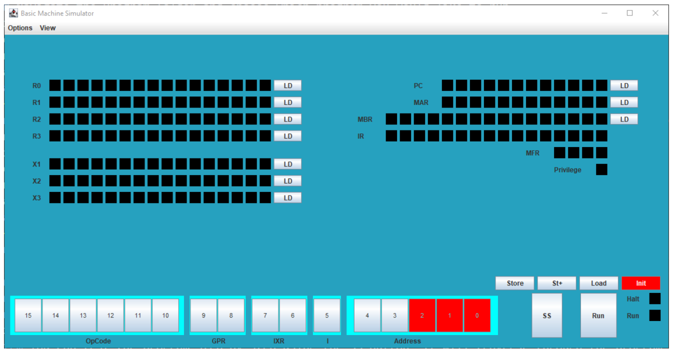


Figure 10: Pressing the switch

8. Load the value in the switch to the Program Counter (PC) by pressing the LD button adjacent to it.
9. Once the Value is loaded it would turn on the simulated LEDs.

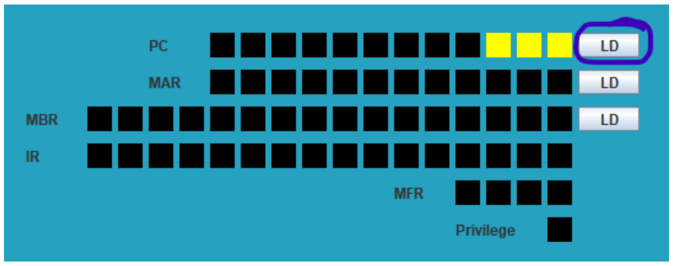


Figure 11: Load the entry point into PC

10. Here you have to options:
    - You can test the program step by step using the SS button. It will traverse through the program normally as the Program Counter is incremented.

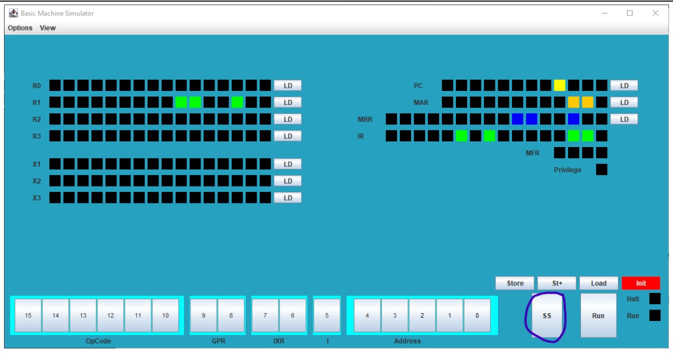
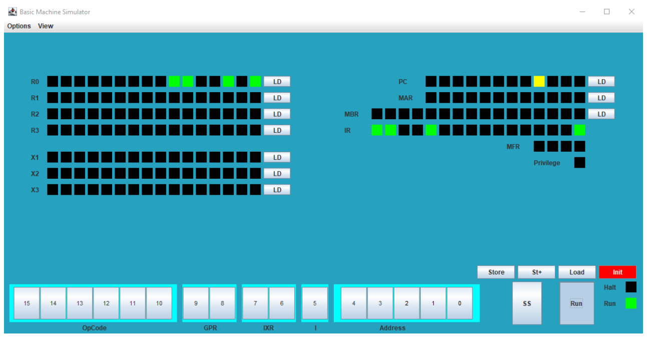
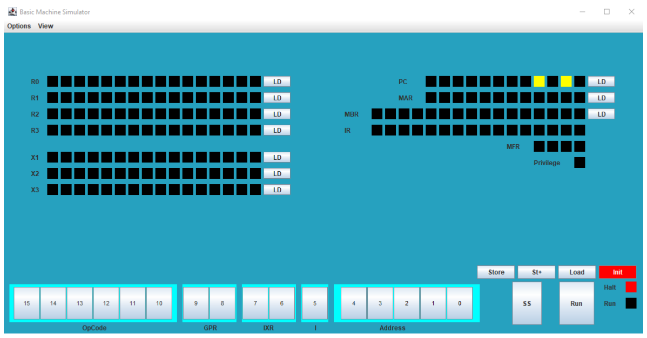


Figure 12: After Pressing the SS Button on ipl.txt

- You can run the program directly by using the run Button. As soon as the Run button is pressed, the PC will increment or jump to other instructions on its own and the Run indicator will turn GREEN. Once the program comes to a halt, the halt LED indicator will turn RED and the Run indicator will turn BLACK. After this no more programs can be run unless it is reset in the options menu.



(a) Running the prog2.txt.



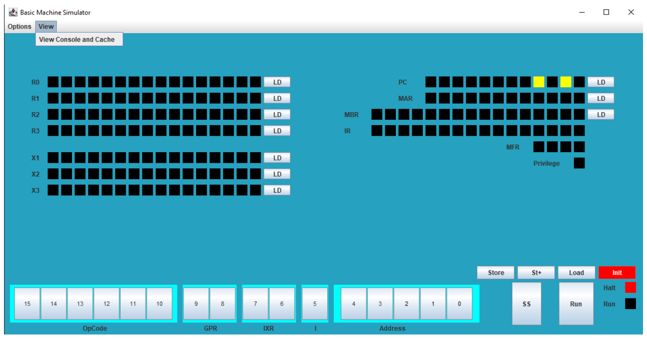
(b) Prog2.txt halting after completion.

2.3.2 For Mac Systems

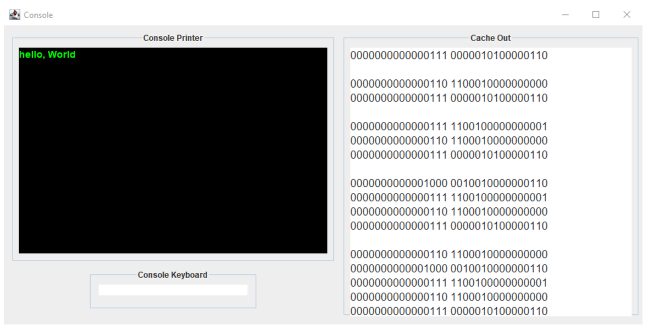
1. Extract the zip file and open the extracted directory.
2. Ensure your java is setup before continuing as per instructions provided on Introduction.
3. Run the runsim.sh file to start the file.
4. Follow the steps followed on the Windows section when the program is activated and run.

2.4 Viewing the cache

Since the Console printer and keyboard is attached alongside the cache, Go to view in the menu and view the cache and the console Option. The figure below shows the cache trace. More about the additional features will be discussed in the next section.



(a) Going to View Console and Cache.



```
hello, World
```

```
0000000000000111 0000010100000110

0000000000000110 1100010000000000
0000000000000111 0000010100000110

0000000000000111 1100100000000001
0000000000000110 1100010000000000
0000000000000111 0000010100000110

0000000000001000 0010010000000110
0000000000000111 1100100000000001
0000000000000110 1100010000000000
0000000000000111 0000010100000110

0000000000000110 1100010000000000
0000000000001000 0010010000000110
0000000000000111 1100100000000001
0000000000000110 1100010000000000
0000000000000111 0000010100000110
```

(b) Console and Cache.

3 Design Notes

3.1 CPU class structure

3.1.1 CPU Structure

- PC
- CC
- IR
- MAR
- MBR
- MFR
- R0,R1,R2,R3 (GPR)
- X1,X2,X3 (IXR)
- dev (for devices)
- cache (for cachelines)

3.1.2 Instructions Implemented

- HLT (0x00), LDR (0x01), STR (0x02), LDA (0x03), LDX (0x21), STX (0x22).
- AMR (0x04), SMR (0x05), AIR (0x06), SIR (0x07).
- JZ (0x08), JNE (0x09), JCC (0x0A), JMA (0x0B).
- JSR (0x0C), RFS (0x0D), SOB (0x0E), JGE (0x0F).
- MLT (0x10), DVD (0x11), TRR (0x12), AND (0x13), ORR (0x14), NOT (0x15).
- TRAP(0x18), SRC (0x19), RRC (0x1A), FADD(0x1B), FSUB(0x1C), VADD(0x1D).
- VSUB(0x1E), CNVRT(0x1F), LDFR(0x28), STFR(0x29), FIN (0x31), OUT (0x32), CHK (0x33).

3.2 Cache Implementation

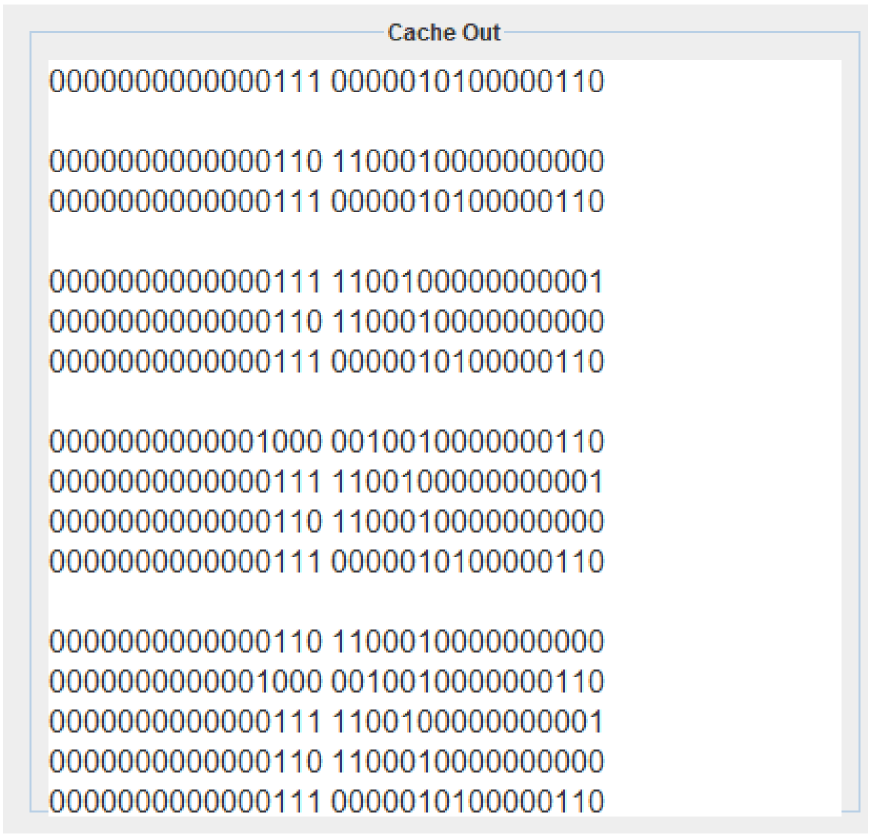
- Cache is implemented by developing a Node structure for cache lines containing two members:
  - Key: Contains the address of the Program Counter
  - Val: Contains the Value of the current IR.
- Each Cache line is stored in form of an array following a FIFO structure.

The implementation of the structure is as follows:

Listing 1: Cache Struct Implementation Code

```
class CacheData{
    public short key;
    public short val;
}
public CacheData[] lines;
```

Here is the sample below for an output for cache:



```
0000000000000111 0000010100000110

0000000000000110 1100010000000000
0000000000000111 0000010100000110

0000000000000111 1100100000000001
0000000000000110 1100010000000000
0000000000000111 0000010100000110

0000000000001000 0010010000000110
0000000000000111 1100100000000001
0000000000000110 1100010000000000
0000000000000111 0000010100000110

0000000000000110 1100010000000000
0000000000001000 0010010000000110
0000000000000111 1100100000000001
0000000000000110 1100010000000000
0000000000000111 0000010100000110
```

Figure 15: Sample Cache Output

3.3 Devices Implementation

The default main components added to the Simulator is the Console keyboard and the console printer:

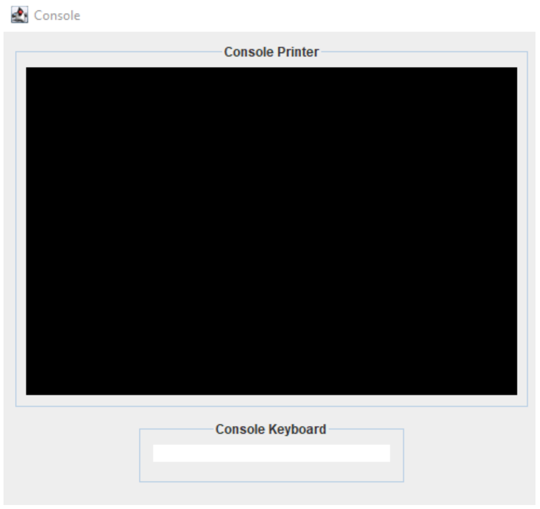


Figure 16: Console Printer and Console Keyboard

3.4 Implementing MFR and Trap Code Phase III

3.4.1 Reserved Memory

The Following are the memory locations that are reserved for this project:

1. 0 Reserved for TRAP instruction
2. 1 Reserved for Machine Fault
3. 2 Store PC for TRAP
4. 3 Not used
5. 4 Store PC for Machine Fault
6. 5 Not used
7. 6 Reserved for temporarily storing indirect address
8. 7 Store R0 when JSR invoked
9. 8 Store R3 for return value
10. 9 Additional Storage Feature

3.4.2 MFR and TRAP implementation

The memory locations are reserved for certain functions, such as trap instruction and machine faults. The trap instruction is reserved at location 0, while location 1 is reserved for machine faults, which is caused by an erroneous condition in the machine. The simulator will trap to memory address 1 when a machine fault occurs, and the MFR (Machine Fault Register) will be set to a binary value corresponding to the type of fault that occurred. Predefined machine faults include illegal memory address to reserved locations, illegal trap code, illegal operation code, and illegal memory address beyond 2048. When a trap instruction or machine fault occurs, the current PC and MSR (Machine Status Register) contents are saved to reserved locations, and the address from location 0 or 1 is fetched into the PC to execute the first instruction of a routine that handles the trap or machine fault.

The reserved memory locations and machine faults are crucial components that ensure proper operation of the machine. These features provide a way to handle errors and unexpected events that may occur during simulation.

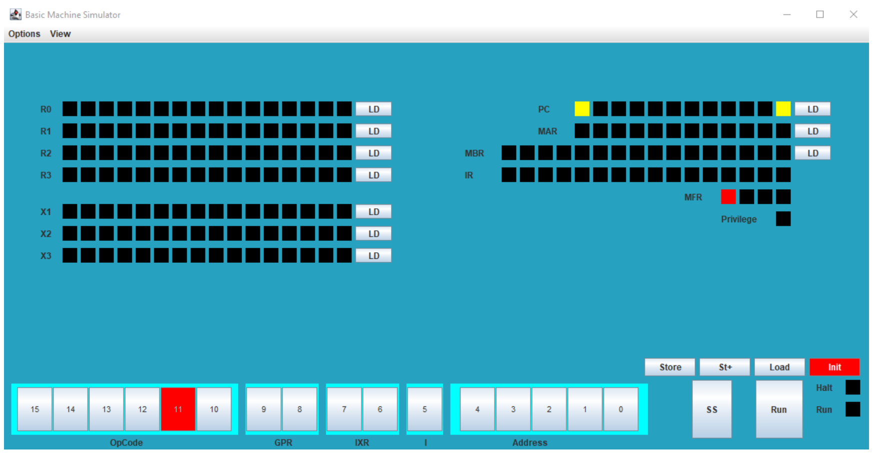


Figure 17: MFR showing illegal out of bounds access

4 Running Program 2

In order to Run the Program 2 certain steps have to be setup before running the program:

1. Load the GUI. You should see a new Menu option called File.

2. Manually Set the Switches to the value 1000. Load that value into the MAR.

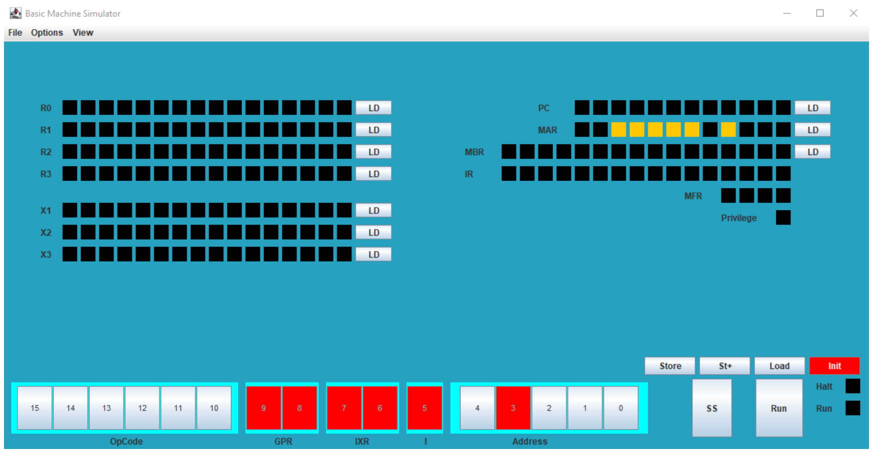


Figure 18: MAR set to 1000 with switches and LD Button.

3. Go to File -¿ Open File. This triggers a file opening dialouge box.

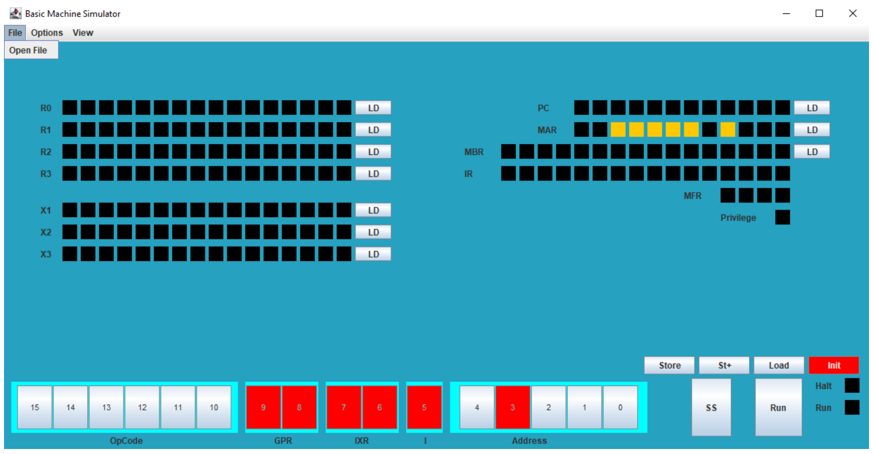
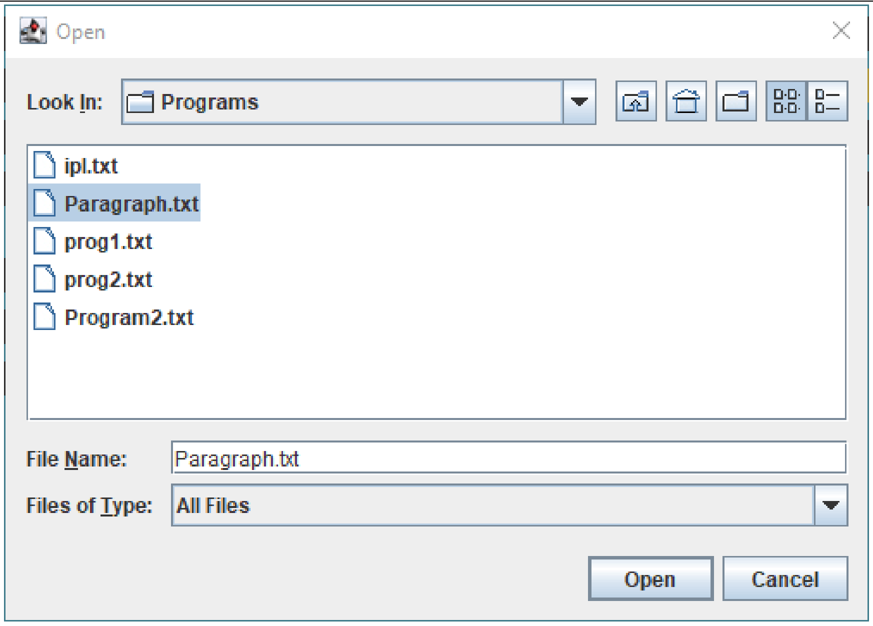


Figure 19: Open File.

4. Locate the paragraph.txt file.



ipl.txt
Paragraph.txt
prog1.txt
prog2.txt
Program2.txt

File Name: Paragraph.txt
Files of Type: All Files

Figure 20: Paragraph.txt File

5. Once loaded, It will successfully load the file. The contents of the file will be stored at location 1000 pointing to the head of the file in memory.

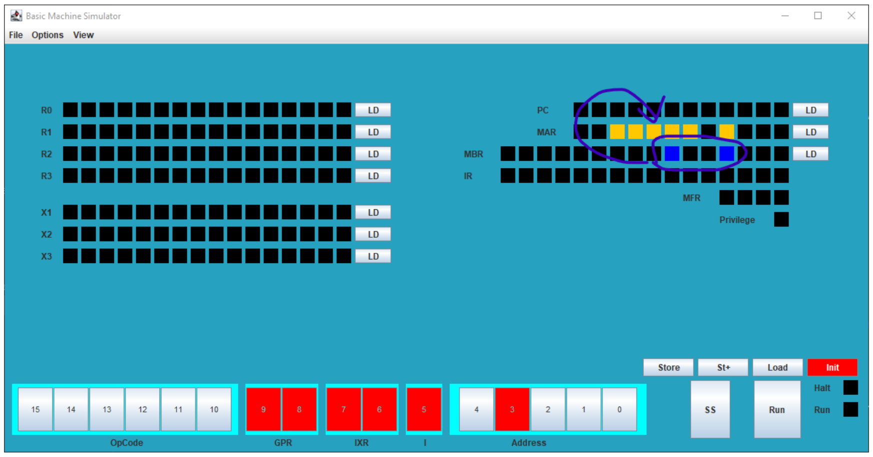
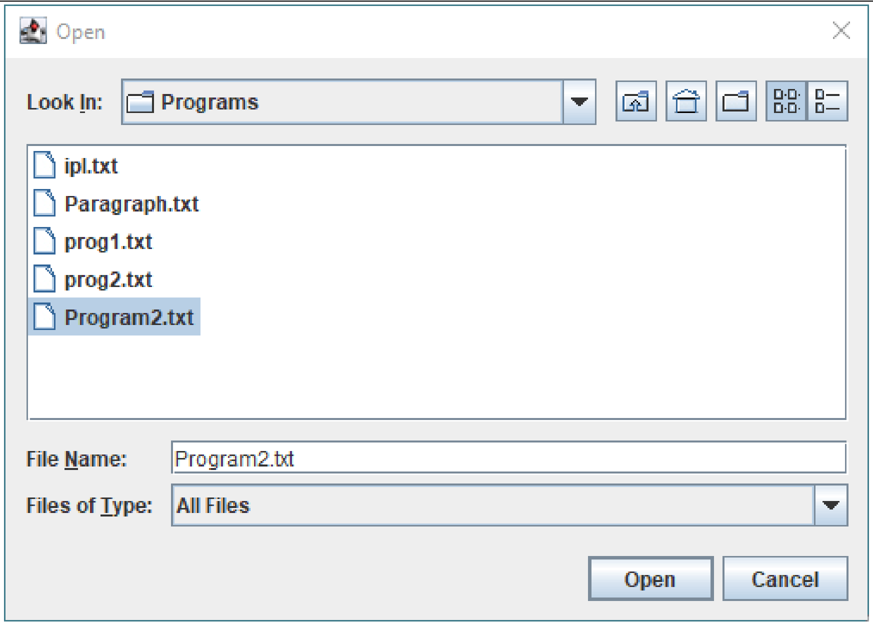


Figure 21: Head of Paragraph.txt in memory

6. Click the init button to trigger the file dialog.

7. Locate Program2.txt file and Open it. It will load the Program2.txt file on the memory.



Look In: Programs

ipl.txt
Paragraph.txt
prog1.txt
prog2.txt
Program2.txt

File Name: Program2.txt
Files of Type: All Files

Figure 22: Program2.txt Location

8. Note that the Entry Point of the Program Booting is at memory location 12.
9. Set the switch value to 12 and load that value in the PC using the LD button.

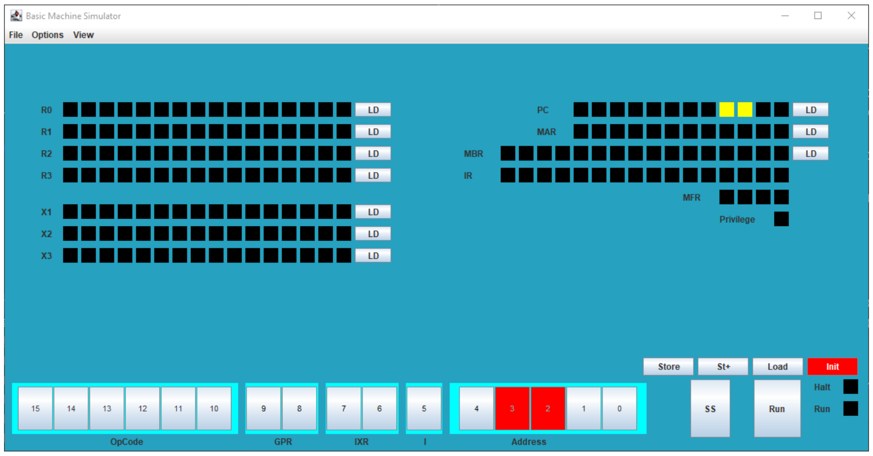


Figure 23: Setting Program Boot at Location 12 in PC

10. Then go to the menu bar and look for View.
11. Go to View -¿ View Console and Cache.

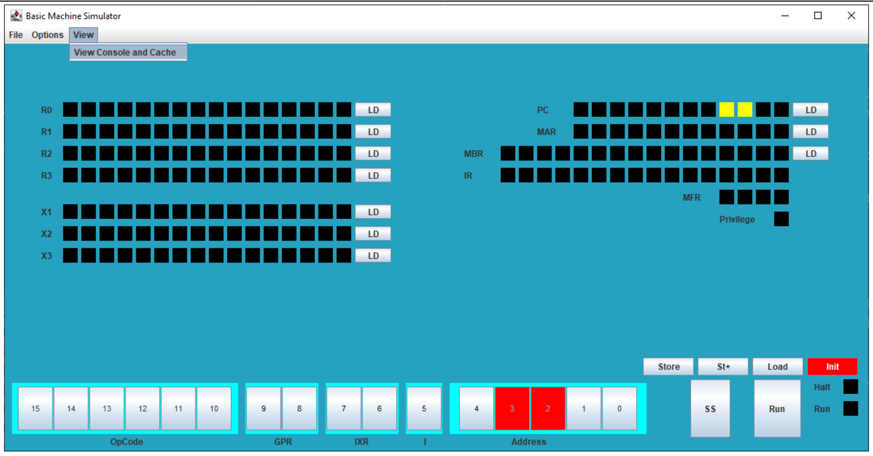


Figure 24: View Console and Cache

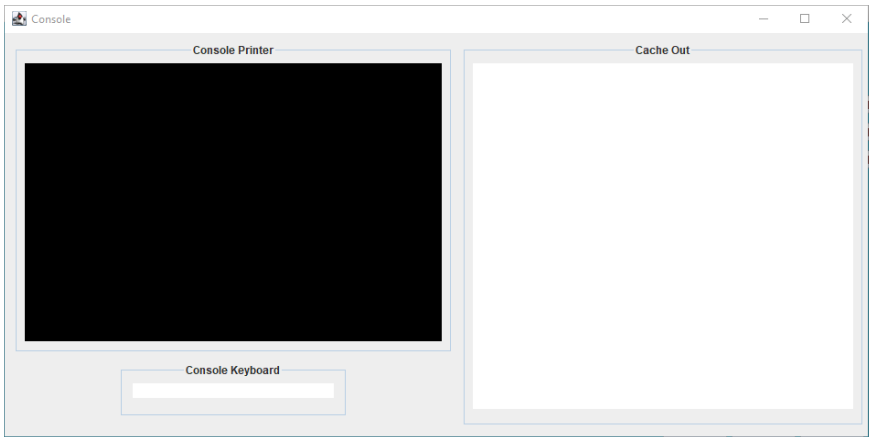


Figure 25: Opening Console and Cache GUI

12. Go to the Console Keyboard and Enter the string you want to search.

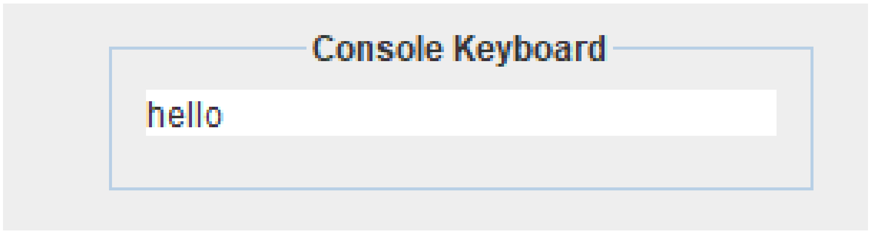


Figure 26: Enter Input in Console Keyboard

13. Now you can either press Run to run the Program or do a single step to test the program step by step.

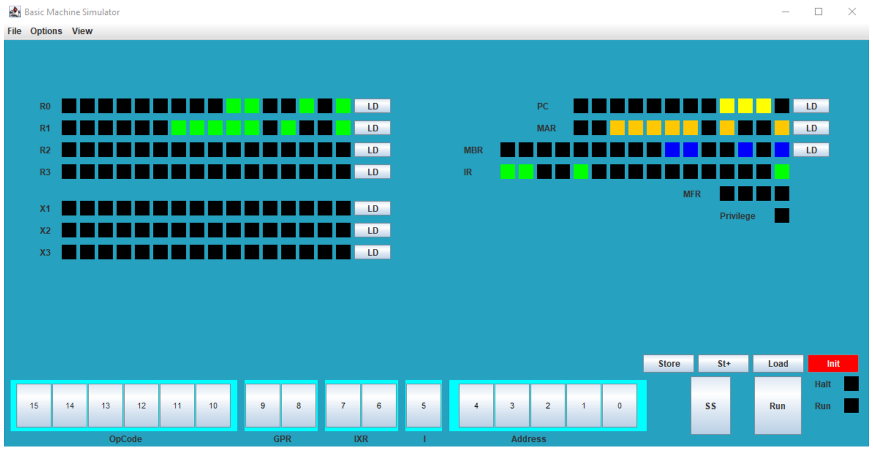
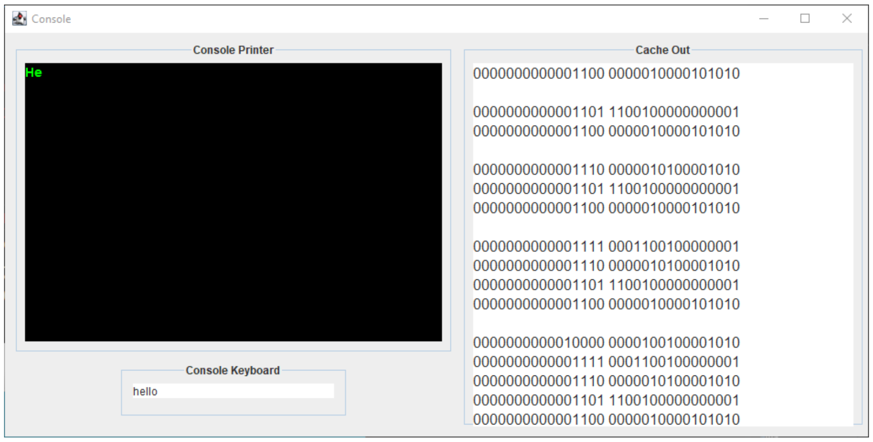


Figure 27: Single Step running the Program


Console

Console Printer

```
He
```

Console Keyboard

```
hello
```

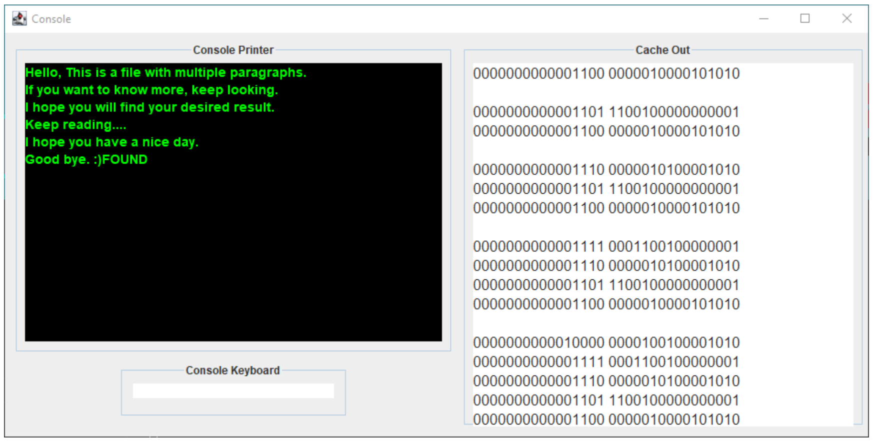
Cache Out

```
0000000000001100 0000010000101010

0000000000001101 1100100000000001
0000000000001100 0000010000101010

0000000000001110 0000010100001010
0000000000001101 1100100000000001
0000000000001100 0000010000101010

0000000000001111 0001100100000001
0000000000001110 0000010100001010
0000000000001101 1100100000000001
0000000000001100 0000010000101010

0000000000010000 0000100100001010
0000000000001111 0001100100000001
0000000000001110 0000010100001010
0000000000001101 1100100000000001
0000000000001100 0000010000101010
```

Figure 28: Output generated from console and cache.

4.1 Running the Program Test

- Sucessful Match


Console

Console Printer

```
Hello, This is a file with multiple paragraphs.
If you want to know more, keep looking.
I hope you will find your desired result.
Keep reading....
I hope you have a nice day.
Good bye. :)FOUND
```

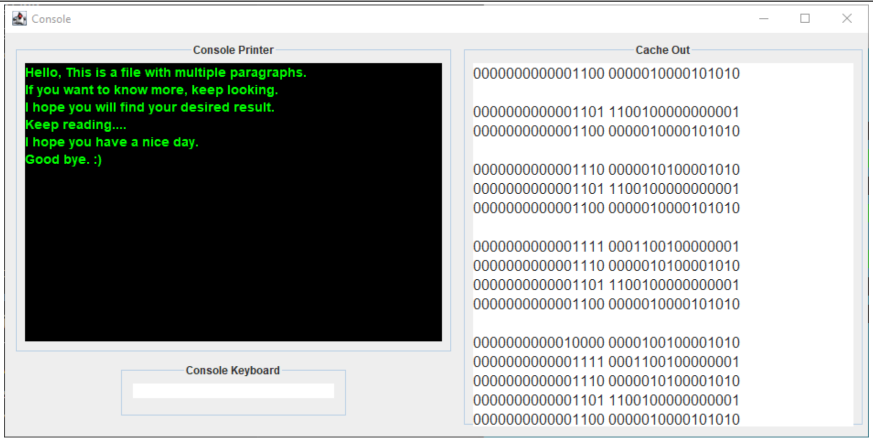
Console Keyboard

Cache Out

```
0000000000001100 0000010000101010

0000000000001101 1100100000000001
0000000000001100 0000010000101010

0000000000001110 0000010100001010
0000000000001101 1100100000000001
0000000000001100 0000010000101010

0000000000001111 0001100100000001
0000000000001110 0000010100001010
0000000000001101 1100100000000001
0000000000001100 0000010000101010

0000000000010000 0000100100001010
0000000000001111 0001100100000001
0000000000001110 0000010100001010
0000000000001101 1100100000000001
0000000000001100 0000010000101010
```

Figure 29: Console Showing Match Found with the Message "FOUND"

- Unsuccessful Match



```
Hello, This is a file with multiple paragraphs.
If you want to know more, keep looking.
I hope you will find your desired result.
Keep reading....
I hope you have a nice day.
Good bye. :)
```

```
0000000000001100 0000010000101010

0000000000001101 1100100000000001
0000000000001100 0000010000101010

0000000000001110 0000010100001010
0000000000001101 1100100000000001
0000000000001100 0000010000101010

0000000000001111 0001100100000001
0000000000001110 0000010100001010
0000000000001101 1100100000000001
0000000000001100 0000010000101010

0000000000010000 0000100100001010
0000000000001111 0001100100000001
0000000000001110 0000010100001010
0000000000001101 1100100000000001
0000000000001100 0000010000101010
```

Figure 30: Console Showing Nothing with Failed Match

4.2 Program 2 Source Code

Listing 2: Program 2 Source Code

```
0008 002B
0009 0218
000A 03E8
000C 042A
000D C801
000E 050A
000F 1901
0010 090A
0011 240C
0012 2C16
0014 01F4
0015 0060
0016 8454
0017 8495
0018 0E40
0019 0A84
001A C700
001B 0BA4
001C 1A01
001D 2719
001E 0A85
001F 8485
0020 2C28
0013 0028
01FC 0000
0028 03E8
0029 01F4
002A 01F4
002B 84D3
002C 04C0
002D 05C1
002E 07E1
002F 2329
0030 06E0
0031 4AC0
0032 2BD7
```

```
0033 0588
0034 25CF
0035 1801
0036 08C0
0037 058A
0038 0988
0039 05C2
003A 07E2
003B 22D5
003C 27C8
003D 0000
003F 1801
0040 1901
0041 08C0
0042 09C1
0043 0588
0044 1901
0045 0988
0046 05C1
0047 27C6
0048 0000
0218 0FDE
0219 CB01
021A 1B09
021B CB01
021C 1B06
021D CB01
021E 1F07
021F CB01
0220 1F0A
0221 CB01
```

5 Vector Operations and Floating Point Operations Part IVa

Two Floating Point Registers has been added where for a 16bit format we have followed a format where 1 bit is used as the sign bit, 7 bits for the exponents between the range(-63 to 64) and remaining bits for the mantissa.

Floating Operations/Vector OpCodes are implemented and demonstrated in program 3 and program 4 respectively. For this part we have implemented the Vector Operations like this below:

- Memory to Memory Operations performed.
- Either FR0 or FR1 is used to obtain the size of the vector operations to be taken.

5.1 Demonstration of Vector Operations/Floating Operations using Program 3

1. Run the Script to launch the Simulator.
2. Click on the Init Button.
3. Locate Program3.txt
4. Insert 100 into the MAR and press the load button to see the original button
5. Load the File.
6. Load the Value to 15 using the switches to the PC section.
7. Do single steps running.

8. After Encountering the OpCode VADD.
9. Load the Value 100 into the MAR.
10. Press the Load Button to see the addition.
11. After doing single step operation again check the value stored in address 100 with VSUB.

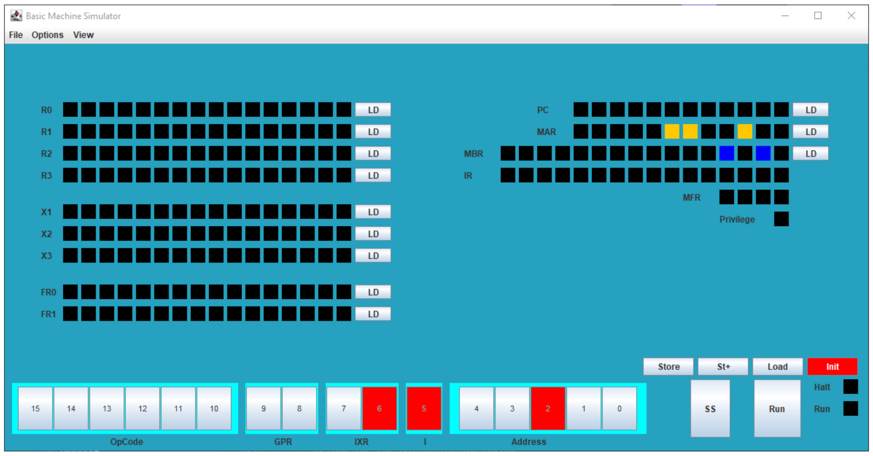


Figure 31: V1 value before VADD.

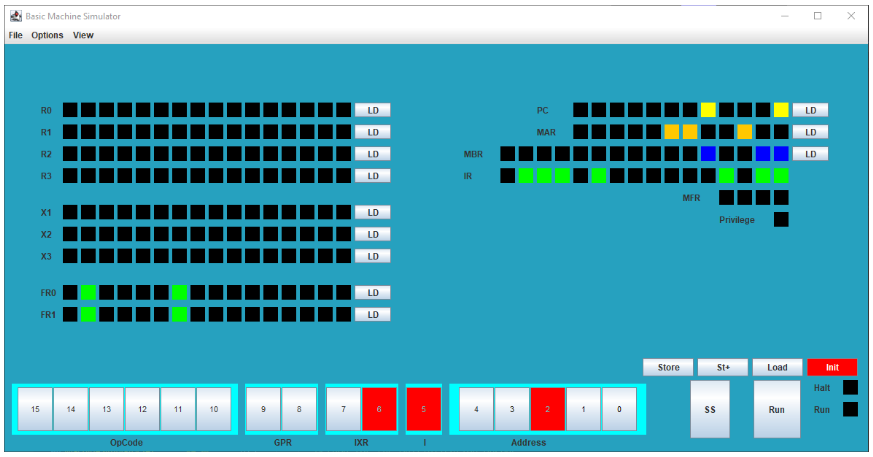


Figure 32: V1 value after VADD.

5.2 Program 3 Source Code

Listing 3: Program 3 Source Code

```
000B 0064
000C 00C8
000D 4200
000E 4200
000F A00D
0064 000A
0065 0015
0066 0020
0067 002B
0068 0036
0069 0041
```

```
006A 004C
006B 0057
00C8 0009
00C9 0012
00CA 001B
00CB 0024
00CC 002D
00CD 0036
00CE 003F
00CF 0048
0010 740B
0011 780B
0012 0000
```

5.3 Program 4 for ConvertOpCode

1. Follow the same steps provided for Program3.txt
2. Do the Single Step Operation from PC counter Location
3. observe the values stored.
4. After the STFR Operation, check the value stored in The memory location inside the MAR by Loading the Value at location 20.
5. We would obtain the resultant value stored in memory location.

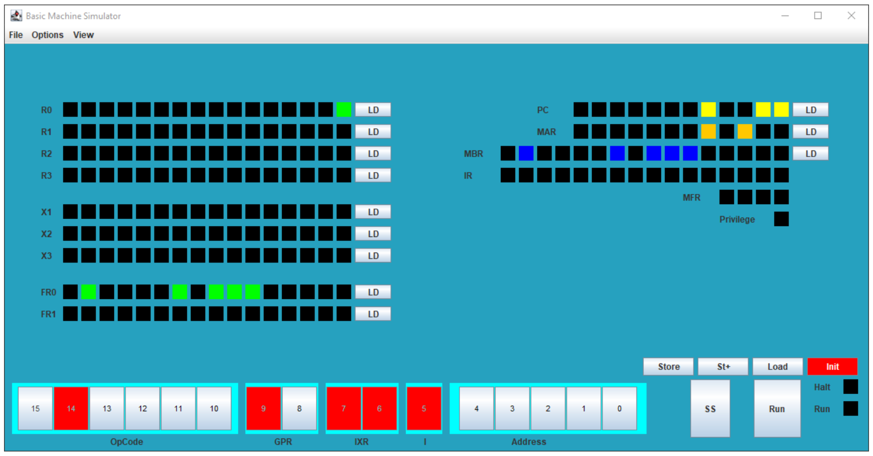


Figure 33: Floating Point Addition of Numbers and Then Storing it in memory location 20

Listing 4: Program 4 Source Code

```
000B 42B8
000C 3F40
000D 0001
000E 040D
000F 7C0B
0010 6C0C
0011 A414
```

6 Conclusion

Through this project, we learned about the macro and microstructures of CPUs, various components of memory, and the execution of instructions. We were able to apply our understanding of Java programming to develop a functional simulator that emulated the behavior of a classical CISC computer. By simulating machine faults, traps, and memory addressing, we gained a deeper understanding of the importance of error handling and fault tolerance in real-world computer systems.

Contributions:

- Abhinava Phukan:
  - Implemented OpCode 0x0-0x3,0x18,0x21,0x22,0x10-0x15,0x19,0x1A,0x31-0x33
  - Device and Cache imeplementation and GUI
  - Program 2 and other Bug Fixes
  - Gui Development
  - MFR and TRAP Code Implementation
  - Report
  - Structure Design and implementation.
  - Implmented Floating Point Operations.
  - Bug Fixing for Program 3 and 4.
- Dev Shah:
  - OpCode Implemented: AMR,SMR,AIR,SIR
  - Console UI Design
  - Main Gui Design
  - Bug Fixes
  - Program 2 Development Collaborator
  - Program 3 Developer.
  - Program 4 Developer.
- Natalie Jordan:
  - OpCode Implemented: JZ,JNE,JCC,JMA
  - Program for Testing (prog1.txt)
  - Report Writing
  - GUI Develpoment
  - GUI Developement for Floating Point Registers
- AlHassan Halawani:
  - OpCode Implemented: JSR, RFS, SOB, JGE
  - Report Writing
  - Structure Implementation.
  - Floating Point Registers