# Machine Simulator Part I Report

CSCI 6461 Computer System Architecture

Team: Zineb, Joseph, Fred, Max, Catherine

Due: Tuesday 07 June 9:30PM

## I. Backend Design Structure

### File Tree structure

```
.
|-- ── Registers
|-- │   ├── ALU.py
|-- │   ├── TemplateRegister.py
|-- │   ├── gpr.py
|-- │   ├── indexRegister.py
|-- │   ├── instructionRegister.py
|-- │   ├── mar.py
|-- │   ├── mbr.py
|-- │   ├── mfr.py
|-- │   ├── pc.py
|-- │   └── register_test.py
|-- ├── cpu.py
|-- ├── gui_grid.py
|-- ├── helper_functions.py
|-- ├── input_file.txt
|-- ├── instruction.py
|-- ├── instruction_test.py
`-- └── memory/
   |-- ├── memory.py
   `-- └── test_mem.py
```

### Description :

The machine simulator is constructed of multiple classes, including *Template register*, *memory*, *cpu*, *gui_grid*, *ALU*, and *instruction*. The *TemplateRegister* class is a super class from which other subclasses are created. These subclasses are *gpr*, *mar*, *mbr*, *mar*, *pc*, *indexRegister*, and *instructionRegister*. The classes and subclasses mirror the components that are needed to create a simulator capable of performing Load (LDR) and Store (STR) instructions. As shown in the class diagram below, each class consists of *get()* and *set()* functions for accessing and updating values. Our Phase I project has functionality for executing LDR, STR, and LDA instructions in memory, as well as loading and storing values into memory via the "Load" and "Store" buttons.

**Class Diagram :**
The class diagram can be viewed at the link below or in the attached pdf.
https://lucid.app/lucidchart/172ded12-dffa-4567-b0cc-fa9c3d19d5fa/edit?invitationId=inv_76012e34-6b40-4004-9c94-ba2eb22e8bee&page=0_0#

**Initialization :**
Running the GUI front panel initializes creation of a CPU class instance which then initializes memory with a default size of 2048 words as well as all required registers and the ALU.

When a signal is sent to initialize the machine (i.e. when the "Init" button is clicked), a user is prompted to upload a .txt file which will load memory with the given values where, as shown in the example below, the first hex value indicates the location in memory and the second hex value indicates the value to store.

0000 004F ← location 0 loaded with value 79
000A 0009 ← location 10 loaded with value 9

As discussed in the User Guide, instructions can be loaded via the .txt upload method or manually via the GUI load buttons.

**Instruction Cycle :**
When a signal is sent to execute an instruction (i.e. when either the "SS" or "Run" button is clicked), the instruction cycle is executed:

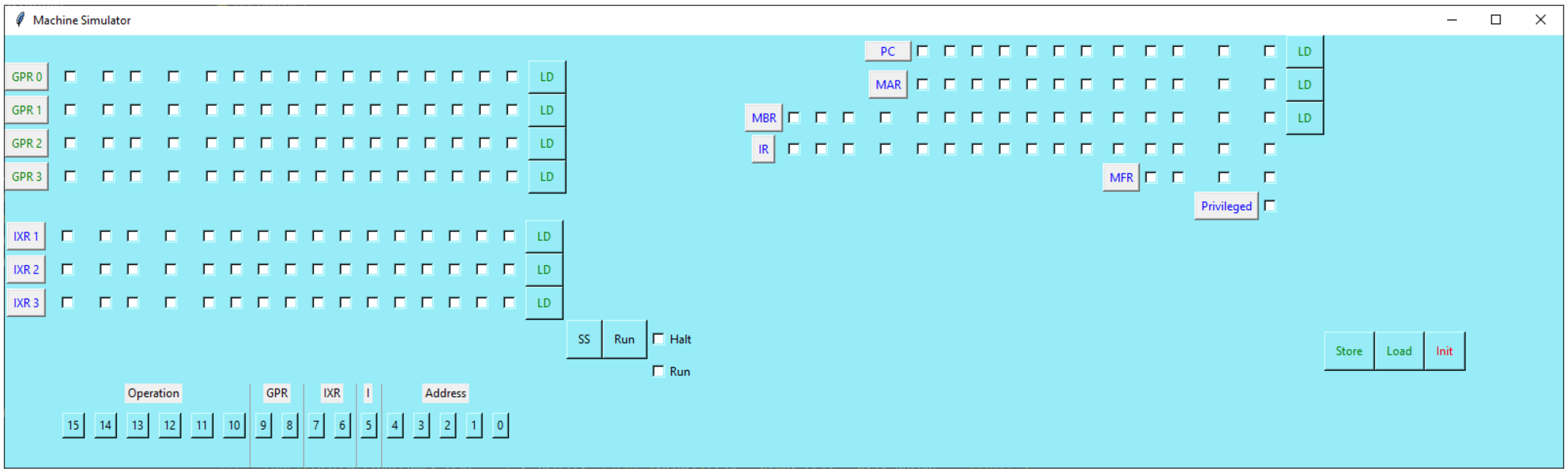
1. Copy address from the PC to the MAR
2. Increment PC
3. Load the MBR with the data from Memory[MAR]
4. Copy the MBR to the IR
5. Decode the instruction in the IR
6. Execute the decoded instruction

If the end of memory is reached via the step through button or upon completion of the run button, the machine will return the PC to 0 so the program may be run again.

**Design Choices**

- When "Run" is clicked, the program runs through all memory addresses and executes the corresponding instruction. Then, the machine is returned to its initial state (with all registers set to 0) so that the program can be executed again.

## II. Front Panel Design



## III. User Guide

**To run GUI program:** `python gui_grid.py`

**Dependencies**: python3, tkinter

**GUI functionality**

*Button definitions*:

a) Init - prompts for a program file to be loaded into memory
b) SS - machine will iterate a single step by executing the instruction at the current PC address
c) Run - machine will iterate through all of memory and execute all instructions
d) LD - loads corresponding register with the value of the switches selected by the user
e) Instruction Switches: all buttons are representative of binary values (clicked turns value to '1') where 0 is representative of the least significant bit
f) Store - stores the value in the MBR into Memory[MAR]
g) Load - Loads the MBR with the value at Memory[MAR]

*Program run*

Users can load initial values in the program in one of two ways:

1) Load program files (.txt) via the init button in the GUI
    a) In order to properly load values into the program, you must ensure values' desired address and desired instructions are formatted in two-part hexadecimal as follows:

0000 004F
000A 0009

2) Manually load instructions or values to the machine via setting the instruction switches (located at the bottom left corner of the GUI) and pressing the "LD" buttons for the target register to be loaded. Clicking "SS" will execute the instruction stored at the address of the PC.
    a) Example: To load the value '7' at address '1'

## IV. Test Program

**test_program.txt** ← loading this input file via the "Init" button will load the following

| **Memory Location** ***(hex)*** | **Value** ***(hex)*** |
|---|---|
| *0000* = 0<br>0000 0000 0000 0000 | *0001* = 1<br>0000 0000 0000 0001 (no op) |
| *0001* = 1<br>0000 0000 0000 0001 | *0011* = 17<br>0000 0000 0001 0001 (no op) |
| *0002* = 2<br>0000 0000 0000 0010 | *0009* = 9<br>0000 0000 0000 1001 (no op) |
| *0003* = 3<br>0000 0000 0000 0011 | *0100* = 256<br>0000 0001 0000 0000 (no op) |
| *0004* = 4<br>0000 0000 0000 0100 | 0603 = 1539<br>000001 10 00 0 00011<br>LDR GPR2, IXR0, 0, 3<br>Load GPR2 with value at mem location 3<br><br>At this state, GPR2 should display decimal value 256/binary value 0000 0001 0000 0000 |
| 0005 = 5<br>0000 0000 0000 0101 | 0520 = 1312<br>000001 01 00 1 00000<br>LDR GPR1, IXR0, 1, 0<br>Load GPR1 with the value at the memory address held in memory location 0<br><br>At this state, GPR1 should display decimal value 17/binary value 0000 0000 0001 0001 |
| 0006 = 6<br>0000 0000 0000 0110 | 0C20 = 3104<br>000011 00 00 1 00000<br>LDA GPR0, IXR0, 1, 0 |

| | |
|---|---|
| | Load GPR0 with the effective address<br><br>At this state, GPR0 should display decimal value 1/binary value 0000 0000 0000 0001 |
| 0007 = 7<br>0000 0000 0000 0111 | 0A09 = 2569<br>000010 10 00 0 01001<br>STR GPR2, IX0, 0, 9<br>Store value at GPR2 to mem location 9<br><br>At this state, memory location 9 should have the value from GPR2 which is decimal value 256/binary value 0000 0001 0000 0000 |

## V. Work Distribution

**Zineb**

- Initial template code for gui which was completed by Cat and Max after
- Register classes
- Instruction class
- Testing instruction and register classes
- Pair programmed with Max to code load and store methods in instruction class
- Suggested the initial OO design for classes, attributes and the required setters and getters that was validated by the team
- Diagram class with Fred

**Max.**

- Helped Zineb and Catherine to code logic and fix bugs when needed
- Pair programmed with Zineb to code load and store methods in instruction class and getting the EA
- Briefly created the ALU class and added its functionalities
- Front end gui buttons (checkboxes, LD buttons)
- Front end Instruction switches to be operational and being displayed in MBR, MAR and PC
- Helped Zineb with the instruction.py back end
- Changed gui into class for the executable
- Paired programming for testing with Cat

**Catherine.**

- Memory class
- Functionality for “Init” button to load program into memory
- Functionality for “Load” and “Store” buttons (worked with Max)

- Functionality for “LD” buttons for GPRs and IXRs
- Functionality for “SS” and “Run” buttons to execute instruction cycle (Pair programmed with Joe)
- Memory access logic for LDR, LDA, STR
- Helped Zineb and Max to debug Load and Store methods
- Helped Max debug IR and MBR GUI display
- Documentation (worked with Zineb, Fred)
- Test program

**Joe.**

- Functionality for “SS” and “Run” buttons to execute instruction cycle (Pair programmed with Catherine)
- Gui update on step through for PC, MAR, MBR, IR GPR#, IXR#
- Initial work on executable and package structure for project

**Fred.**

- Worked on class diagram with Zineb
- Worked on the OO concepts/design that are needed for the project with Zineb
- Worked on program counter
- Documentation (worked with Zineb, Cat)