CSCI6461 Computer System Architecture Team Project

Phase 3

Vaibhav Vemula
vaibhav.vemula@gwu.edu

Rajith Ravikumar
rajith.ravikumar@gwu.edu

Gayathri Kalthi Reddy
gayathri.kalthireddy@gwu.edu

Srinith Rao Bichinepally
srinithrao.bichinepally@gwu.edu

1. Objective -

The objective of this project is to provide a comprehensive overview of the design and implementation of a basic machine architecture, with a specific focus on the development of a simple memory system and the accurate execution of Load and Store instructions. This report also covers the creation of an initial user interface for the simulator, allowing users to interact with and control the simulated architecture effectively. Additionally, the report encompasses the design and development of an assembler for the simulator, capable of encoding all instructions provided in a given text file. The assembler will take a text file as input, and its primary goal is to translate the code within this file into an output file that is identical in format to the input file used in Project I.

2. Understanding the interface

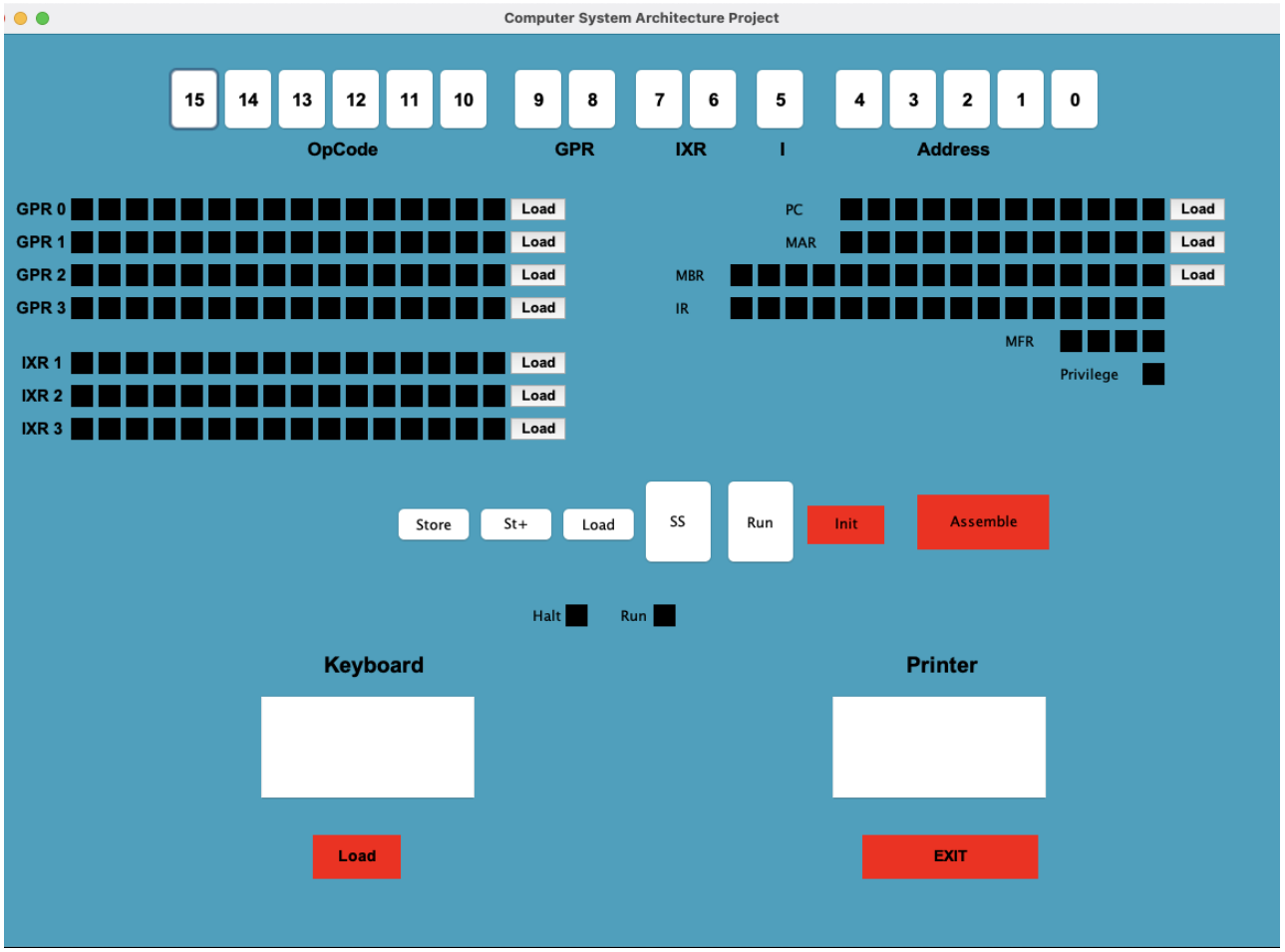

Fig 1. *Phase 3 Interface*

2.1. *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.

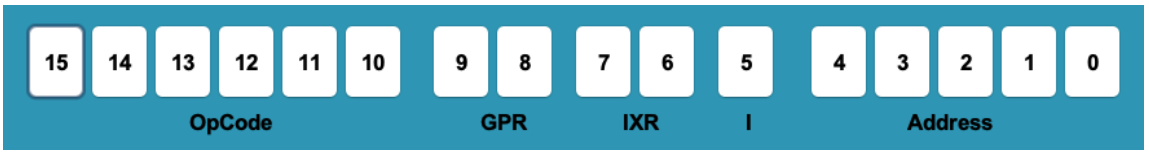

Fig 2. *Switches*

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.

2.2. 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.

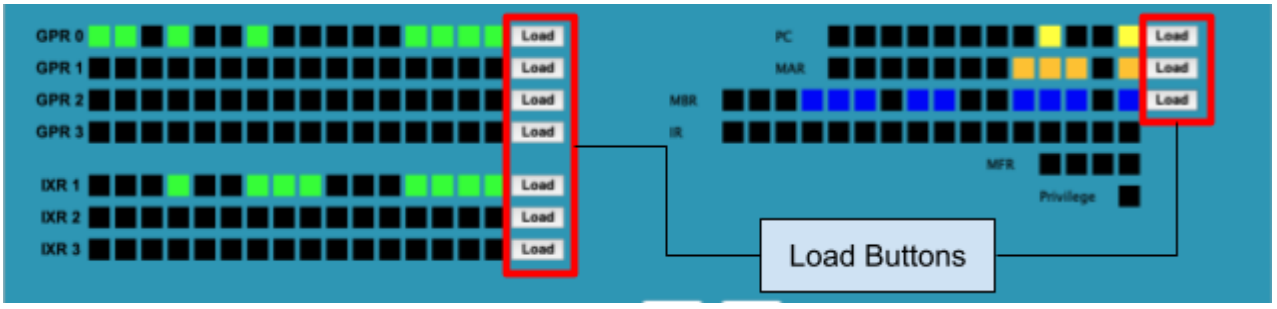

Fig. 3. *Load Buttons*

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.

2.3. 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.

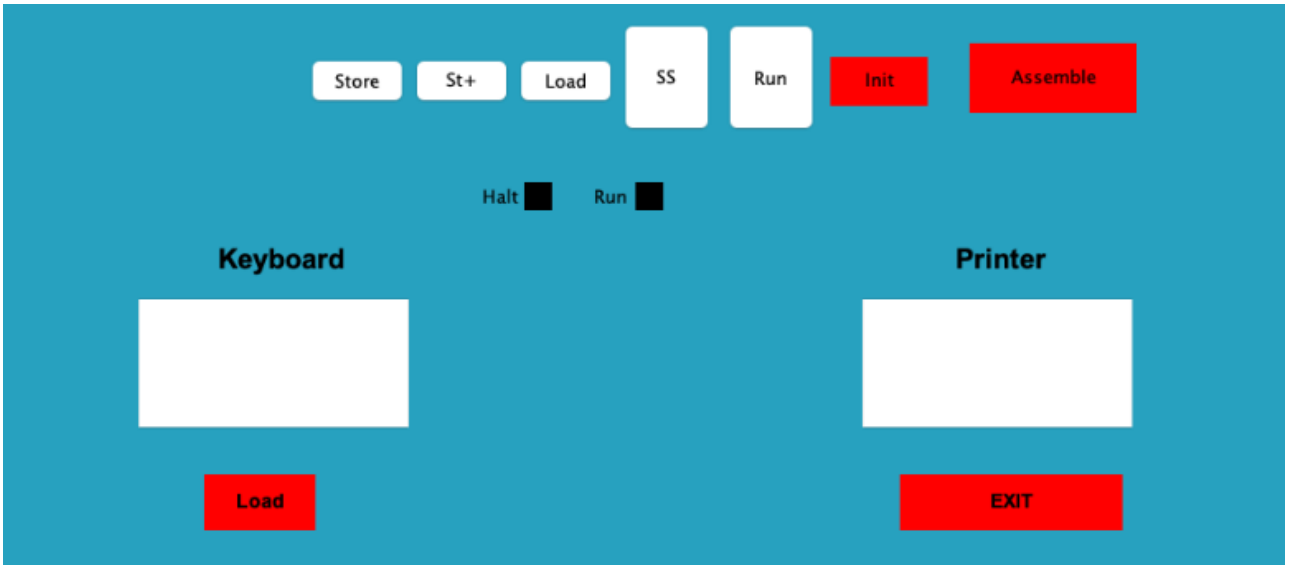

Fig.4. *Control Buttons in Gui Simulator*

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. St+ 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 (IPL Button)

   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.

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.

2.4. Assembler Button

The assembler is vital for translating assembly instructions into machine code, mirroring the input file format from Project I, employing hexadecimal locations. It follows a two-pass approach:
Two-Pass Assembler Workflow:

First Pass: Label Processing

1. Scan through files contents to compute the location of all labels present in the code.
2. For each label encountered, the label name and its corresponding location value are stored.

Second Pass: Code Translation

1. Each assembly instruction is processed and translated into the appropriate machine code instruction.
2. The assembler maps mnemonic opcodes to their respective numerical representations.
3. A dedicated function is designed to take the text-based op code part of an instruction and return the corresponding hexadecimal representation.

2.5 Load Button

The load button beneath the keyboard allows users to transfer the entered data onto the register or memory.

2.6 Exit Button

Press the exit button the exit simulator.

3. Design Notes

3.1. CPU Structure

- PC
- IR
- MAR
- MBR
- MFR
- R0,R1,R2,R3 (GPRs)
- X1,X2,X3 (IXRs)

3.2. Instructions Implemented

- LDR (OpCode - 01)
- STR (OpCode - 02)
- LDA (OpCode - 03)
- LDX (OpCode - 41)
- STX (OpCode - 42)
- JZ (OpCode - 10)
- JNE (opcode - 11)
- JCC (opcode - 12)
- JMA (opcode - 13)
- JSR (opcode - 14)
- RFS (opcode - 15)
- JGE (opcode - 17)
- AMR (opcode - 4)
- SMR (opcode - 5)
- AIR (opcode - 6)
- SIR (opcode - 7)

3.3. Code Structure

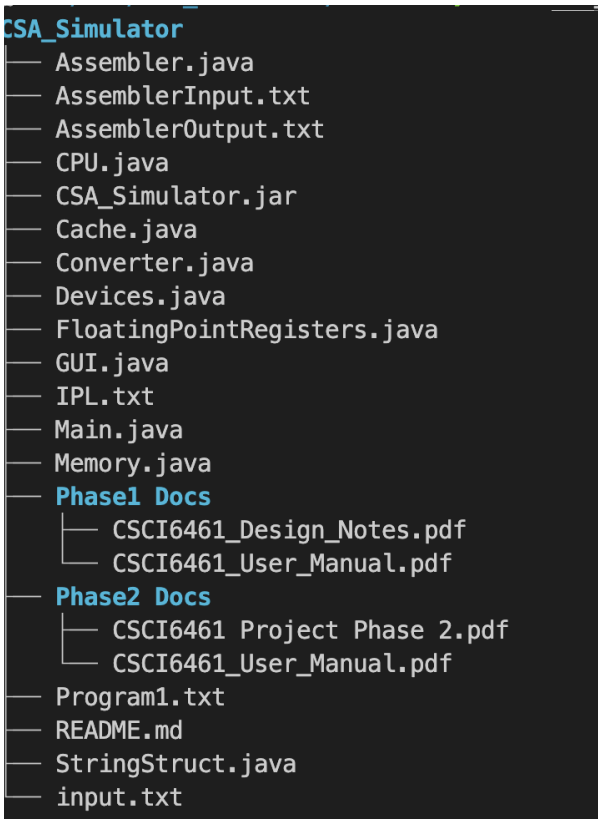

***Fig.5**. Folder Structure*

3.3.1. IPL.txt

The machine's front interface includes an Initial Program Load (init) button, which restarts the computer and loads the file IPL.txt into memory. The file is in hexadecimal format, with two hexadecimal numbers each line, the first being a machine address and the second being the content at that address.

3.3.2. CPU.java

The most important file in this project is this one, which provides the code needed to run the simulator. It has a number of functions for managing the front end and controlling the simulation.

Few of the important functions are listed below -

1. **private void StoreRegister() -** This is an internal function that loads the value into the specified register.
2. **private void LoadIndexRegister() -** This is an internal function that loads the value from memory into the specified index register.
3. **private void RStoretoMem()** - This is an internal function to store memory from a general purpose register.

4. **private void IXStoretoMem()** - This is an internal function to store memory from an index register.
5. **private short FetchEA()** -This is an internal function that fetches the Effective address effectively.
6. **private void StoreRegisterwithEA()** - This is an internal function that loads the effective address value into the specified register.
7. **public void Reset()** - Function used to reset the machine state.
8. **public void Execute()** - Function used to execute the instructions according to the memory.

3.3.3. Converter.java

This class contains functions to perform numerical conversions.

1. **BinaryToDecimal()** - Function to convert Binary number to Decimal number in an array.
2. **DecimalToBinary()** - Function to convert Decimal number to Binary number in an array.
3. **HexToDecimal()** - Function to convert Hex number to Decimal number in an array.

3.3.4. Devices.java

This file contains instructions for printing and connecting to input and output devices while the simulator is operating.

3.3.5. GUI.java

This file contains all of the front panel interface's GUI Components and event handlers. This file is also used for all input and output actions.

Few of the important functions are listed below -

1. **GUI() - constructor** - This constructor sets the location and creates the PC, MAR, MBR, IR, MFR, and Privilege labels to the left of the panels, and sets the label color to black. This creates a backbone structure of the graphical interface.
2. **RefreshLeds()** - This function handles the case when every time the internal of the registers are updated, the LEDs are updated as well.
3. **LoadButtonAction()** - This function contains loops which use a switch case to be activated once the LOAD button is pressed for the corresponding panel.
4. **switchAction()** - This function is used to change the color of the switches at the bottom of the screen once one of them is pressed.
5. **Store()** - This function will store the memory and print to the screen that the store was successful.
6. **StorePlus()** - This function will store the memory and print to the screen that the store was successful MAR is incremented here after storing

7. **LoadValue()** - This function will load the memory and print to the screen for the user that the load was successful in the MBR. If the memory was out of bounds, it will print an error message dialog to the screen.
8. **LoadFileIntoMemory()** - This function is used to load files from the local system to memory (IPL.txt).
9. **loadFile()** - This function is used to load files from the local system.
10. **ProcessFile()** - This function is used to process the file uploaded.
11. **execCode()** -This function is used to execute the code from a CPU.java file.
12. **LoadGui()** - This function is to render the GUI.

3.3.6. Main.java

Main file which contains the main function to set the environment and run the simulator and load the GUI.

3.3.7. Memory.java

This file is to define the memory block. This file is required because Java Does not support unsigned primitive type so we are expanding with int to cover max unsigned value that can be stored in a largest 16 bit number.

3.3.8. AssemblerInput.txt

Each line of "AssemblerInput.txt" contains a single assembly instruction or label declaration. This file uses the decimal representation for the instruction sequence. Labels, which serve as markers for specific locations in the program, are declared using a name followed by a colon (e.g., label:). These labels enable program control flow and are referenced in subsequent instructions.

4. Cache Implementation

Following image shows the implementation of cache.

```
class CacheData{
    public short key;
    public short val;
}
public class Cache extends Converter{
    public CacheData[] lines;
    public int front,rear;

    public Cache(){
        this.front = this.rear = 0;
        lines = new CacheData[16];
        for(int i=0;i<16;i++)
            lines[i] = new CacheData();
    }
    public void push(short key,short val){
        if(this.rear == 16){
            for(int i=0;i<15;i++){
                lines[i].key=lines[i+1].key;
                lines[i].val=lines[i+1].val;
            }
            lines[15].key = key;
            lines[15].val = val;
            return ;
        }
        if(rear==0){
            lines[rear].key=key;
            lines[rear].val=val;
            rear++;
        }else{
            lines[rear].key = key;
            lines[rear].val = val;
            rear++;
        }
    }
}
```