COMPUTER ARCHITECTURE PROJECT (TEAM – 5 – Project 1)

Team Members – Akshay Nayak, Preeti Das, Saurabh SR

1) Interface Components:

1. **General Purpose Registers (GPR):**
   - **R0, R1, R2, R3**: These are four general-purpose registers that can be loaded with data using the "Load" buttons beside each. Typically used for storing temporary data and performing arithmetic operations.
2. **Index Registers (IXR):**
   - **X1, X2, X3**: Index registers, usually utilized for indexing in memory access instructions. Like the GPRs, each has a "Load" button for loading values into them.
3. **Other Registers:**

   MSR: Machine Status Register, which holds system status information, can be loaded manually through the "Load" button.

   IR (Instruction Register): Holds the current instruction being executed. The "Load" button allows you to manually load instructions into it.

   PC (Program Counter): Stores the address of the next instruction to be executed. A "Load" button is available to input a specific address.

4. **Address Section:**

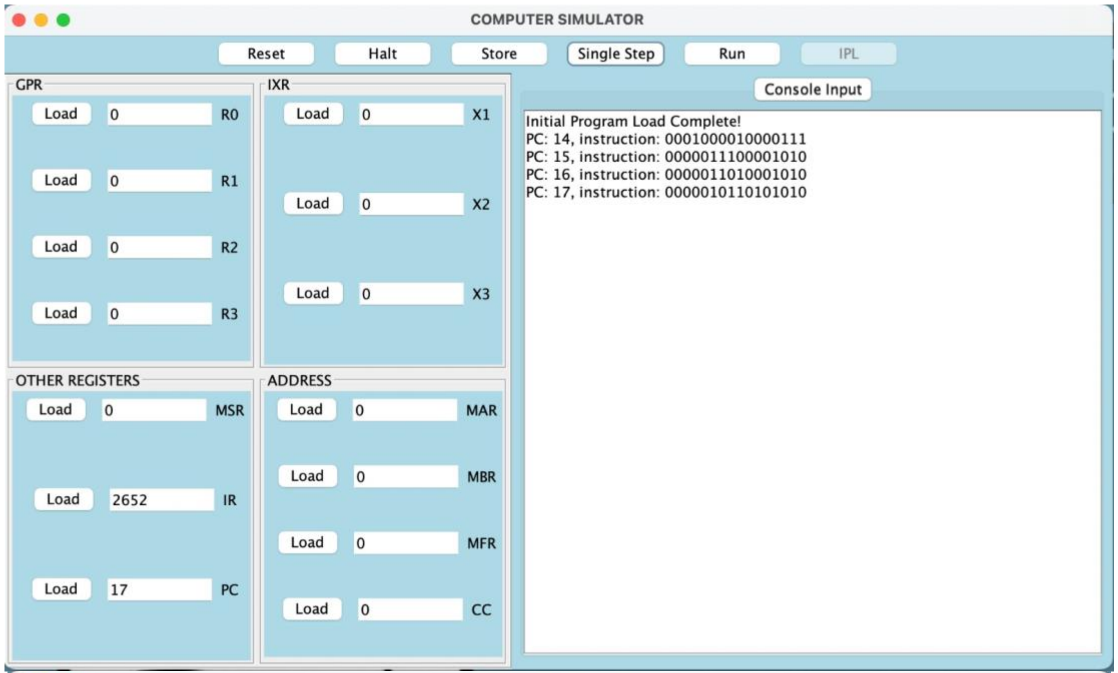
- **MAR (Memory Address Register)**: Holds the memory address being accessed. "Load" button allows setting a specific memory address.
- **MBR (Memory Buffer Register)**: Contains the data fetched from or written to memory. Can be manually loaded.
- **MFR (Memory Fault Register)**: Tracks memory access faults. Can be manually loaded.
- **CC (Condition Code Register)**: Holds condition flags resulting from arithmetic operations (e.g., zero, overflow). It also has a "Load" button.

5. **Control Buttons:**

- **Reset**: Resets the simulator.
- **Halt**: Stops the execution.
- **Store**: Stores the current state of the system.
- **Single Step**: Likely allows stepping through the program one instruction at a time.
- **Run**: Executes the program in full.
- **IPL**: Refer to Initial Program Load, used to boot or load an initial program into memory.

COMPUTER ARCHITECTURE PROJECT (TEAM – 5 – Project 1)

Team Members – Akshay Nayak, Preeti Das, Saurabh SR



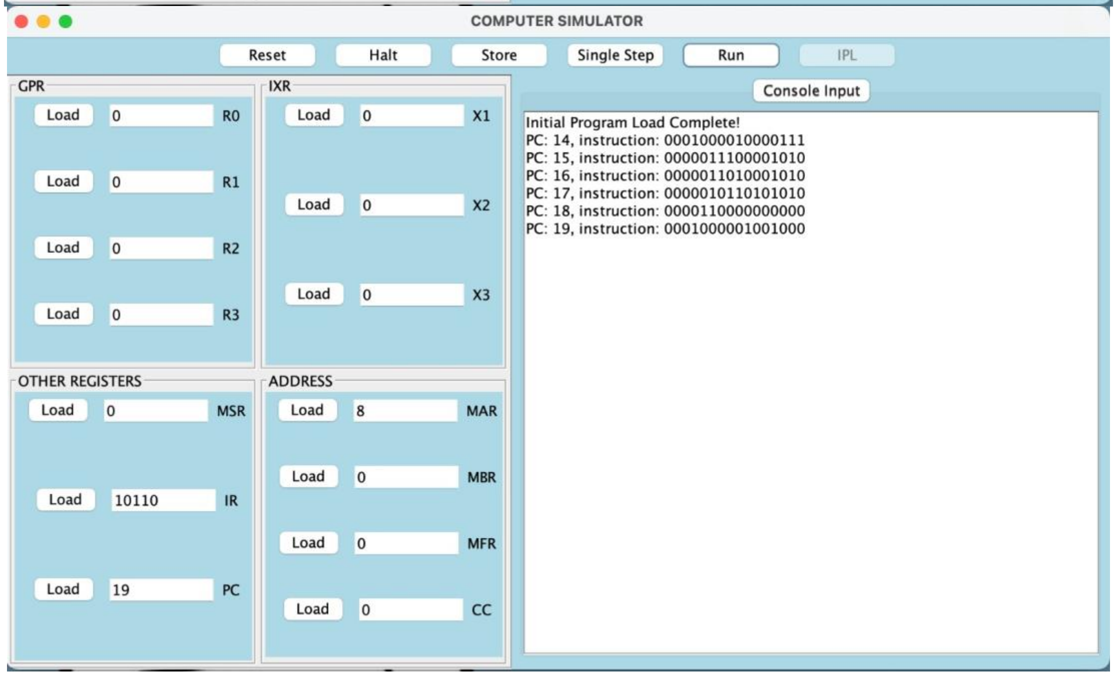
GPR: R0 0, R1 0, R2 0, R3 0

IXR: X1 0, X2 0, X3 0

OTHER REGISTERS: MSR 0, IR 2652, PC 17

ADDRESS: MAR 0, MBR 0, MFR 0, CC 0

```
Initial Program Load Complete!
PC: 14, instruction: 0001000010000111
PC: 15, instruction: 0000011100001010
PC: 16, instruction: 0000011010001010
PC: 17, instruction: 0000010110101010
```



GPR: R0 0, R1 0, R2 0, R3 0

IXR: X1 0, X2 0, X3 0

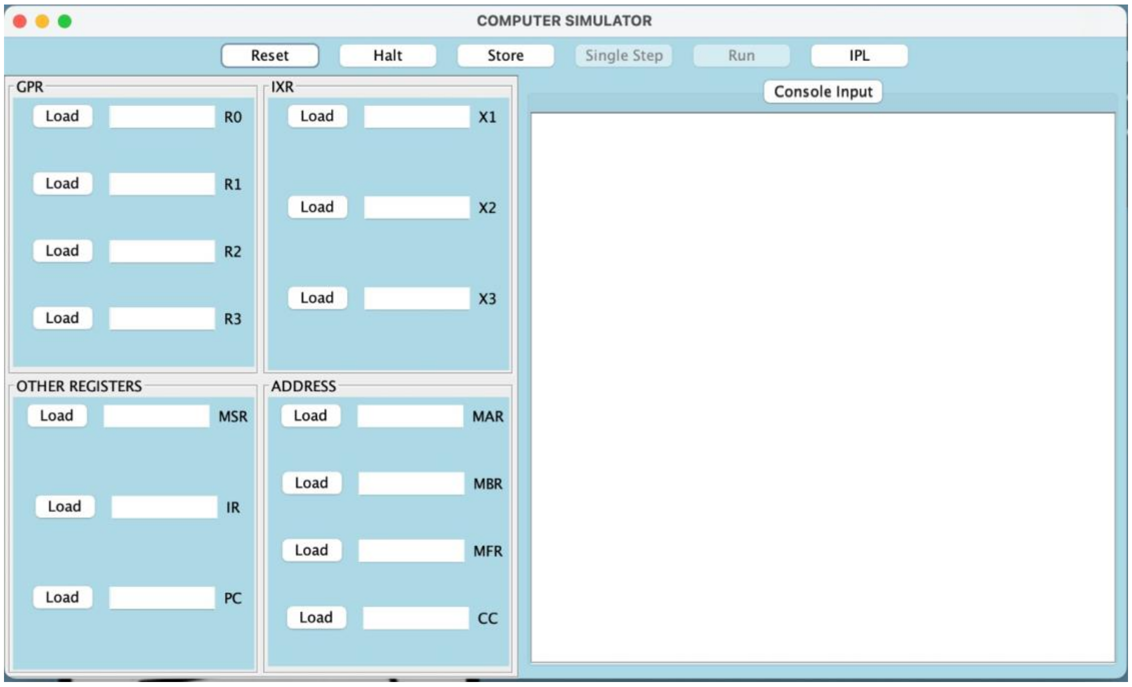
OTHER REGISTERS: MSR 0, IR 10110, PC 19

ADDRESS: MAR 8, MBR 0, MFR 0, CC 0

```
Initial Program Load Complete!
PC: 14, instruction: 0001000010000111
PC: 15, instruction: 0000011100001010
PC: 16, instruction: 0000011010001010
PC: 17, instruction: 0000010110101010
PC: 18, instruction: 0000110000000000
PC: 19, instruction: 0001000001001000
```

COMPUTER ARCHITECTURE PROJECT (TEAM – 5 – Project 1)

Team Members – Akshay Nayak, Preeti Das, Saurabh SR



2) FrontPanel Class –

The FrontPanel class is responsible for simulating a CPU's front panel interface, handling user inputs (via buttons), program execution (single-step or run), and display of register and memory states. Key components include listeners, CPU operations, memory handling, and GUI layout.

Key Functions:

1. **addListeners()**:
   - Adds MouseListener to buttons (SingleStep, Run, Reset, IPL).
   - Handles program loading, instruction execution, and reset logic.
2. **runInstruction()**:
   - Decodes and executes a CPU instruction by loading the class corresponding to the opcode dynamically.
   - Updates registers, cache, and console output after instruction execution.
3. **clearAll()**:
   - Resets the panel's state, clearing registers, memory mappings, and console output.
   - Disables buttons except IPL, re-enabling only after a new program is loaded.
4. **refreshCacheTable()**:
   - Updates the cache table in the UI based on current memory cache states.
5. **refreshRegistersPanel()**:
   - Refreshes all displayed register values based on the internal CPU state.
6. **createBoxPanel()**:

- Creates grouped register panels (GPR, IXR, etc.) for display in the UI.

7. **addTabbedPane()**:
    - Adds a tabbed console panel, allowing user input with prompt formatting.
8. **addButtons()**:
    - Adds control buttons (Reset, Halt, Store, Single Step, Run, IPL) to the panel with configured background color.
9. **addCache()**:
    - Configures and displays a memory cache table, showing tag and data values.
10. **setEnableForPanel()**:
    - Enables or disables all components in a given JPanel based on execution states.

Helper Methods:

- **getInstructionInstance()**: Dynamically loads the instruction class based on the opcode.
- **handleMachineFault()**: Handles exceptions by updating the Machine Fault Register and displaying error messages.
- **pushConsoleBuffer()**: Updates the console with content from printer/keyboard buffers.

This class manages the entire front panel interaction, combining CPU instruction execution with a real-time interface update for a simulator/debugging environment.

3) Backend Functionality Overview: -

The FrontPanel class backend manages core components like instruction execution, memory management, register updates, and user interactions with the simulated CPU. Below are the key backend processes handled by the class:

Instruction Execution

- **runInstruction(String instruction, Registers registers, MCU mcu)**
    - The function processes binary instructions fetched from memory.
    - Extracts opcode from the instruction (first 6 bits) and dynamically loads the appropriate class.
    - Each instruction is executed via its execute() method, which updates the registers and memory.
    - After execution, it refreshes the cache table and console buffer.
- **getInstructionInstance(String className)**
    - Dynamically loads the class for the given opcode using reflection (Class.forName()).
    - Returns an instance of AbstractInstruction that defines the instruction's behavior.

CPU and Register State Management

- **clearAll()**
  - Resets all registers and flags.
  - Clears program instructions from memory and resets the console.
  - Resets CPU state, including the **Program Counter (PC)** and other registers.
- **refreshRegistersPanel()**
  - Updates the visible state of CPU registers in the UI by fetching their current values from the Registers class.
  - It supports various registers (GPR, IXR, MAR, MBR, PC, IR, MSR, etc.).
- **initCPU()**
  - Initializes the CPU components, such as loading registers, clearing memory, and setting default values for critical registers (e.g., PC, IR).

Program Control (Loading and Execution)

- **addListeners()**
  - Implements the control flow for buttons:
    - **Single Step**: Executes a single instruction.
    - **Run**: Continuously executes instructions until the program ends.
    - **Reset**: Clears memory, registers, and halts execution.
    - **IPL (Initial Program Load)**: Loads a program from a file into memory and initializes the CPU for execution.
- **loadProgramFile() (from MCU class)**
  - Loads program instructions from a file (input.txt) into memory for execution.

4) Backend Classes and Structure

- **Registers**: Manages the state of CPU registers, including the **Program Counter (PC)**, **Instruction Register (IR)**, **General-Purpose Registers (GPR)**, and **Index Registers (IXR)**. It provides methods like getPC(), setPC(), getIR(), etc.
- **`AbstractInstruction`**: Base class for CPU instructions. Each instruction (e.g., `LoadInstruction`, `StoreInstruction`) extends this class and implements the `execute()` method to perform specific operations.
- **`Constants`**: Contains constant values, including opcode mappings and fault codes for validating and executing instructions.