CSCI6461

Computer System Architecture

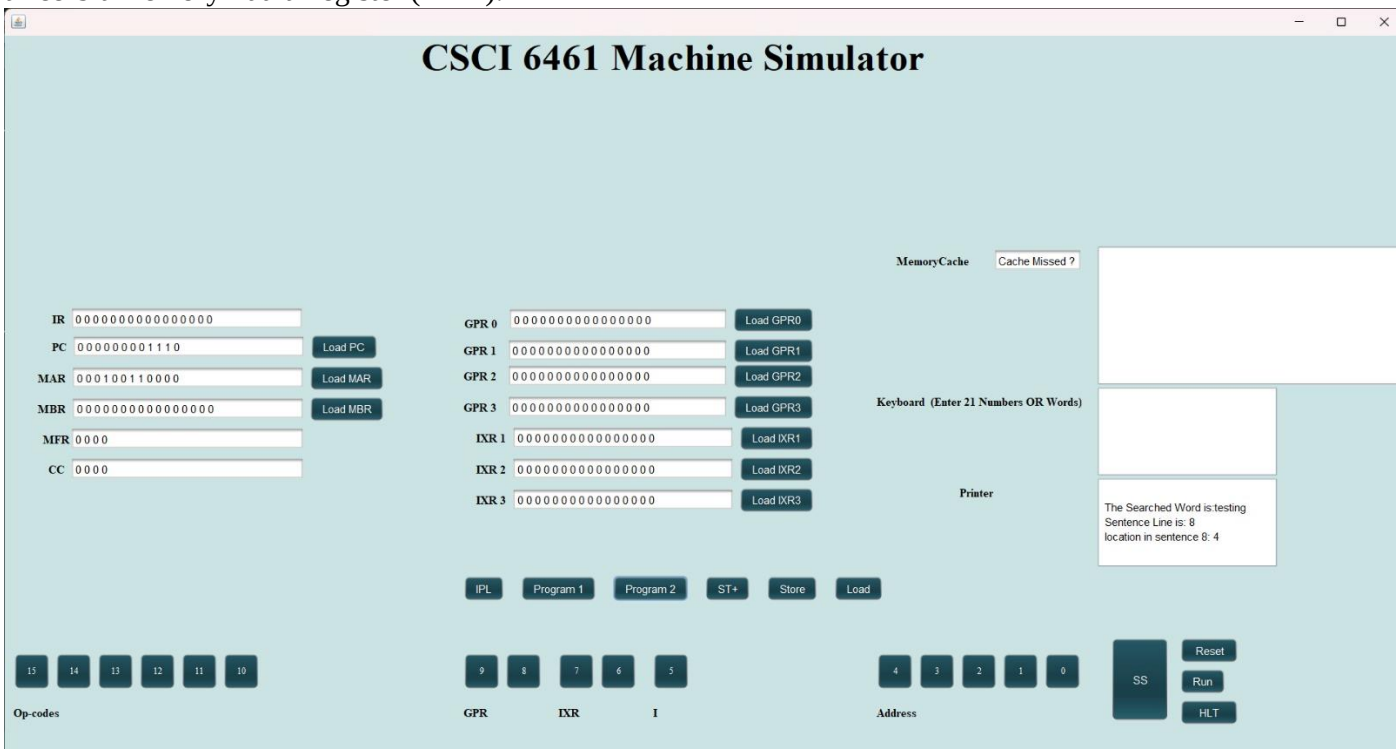
DESIGN DOCUMENT

Team - 5

- Jaiswal Nitish
- Jadhav Yash
- Jiang Xuechun

Components of the Front Panel

The user can load values into the Instruction Register (IR), Memory Buffer Register (MBR), General Purpose Registers (GPR 0-3), Index Registers (IXR 1-3), Program Counter (PC), and Memory Address Register (MAR), all of which are 12 bits in size. However, there is no load button on the four-bit Condition Code (CC) or the three-bit Memory Fault Register (MFR).



Program Counter (PC): The Program Counter holds the address of the instruction to be executed.

Instruction Register (IR): The Instruction Register holds the current instruction being executed by the CPU.

Memory Address Register (MAR): The Memory Address Register stores the address in memory from which data will be fetched or to which data will be sent.

Memory Buffer Register (MBR): The Memory Buffer Register temporarily holds the data being transferred to or from memory.

Memory Fault Register (MFR): The Memory Fault Register stores codes that represent specific memory-related errors or exceptions.

Condition Code (CC): The Condition Code register holds flags that reflect the outcome of operations, such as negative, zero, overflow, or carry.

General Purpose Registers (GPR 0-3): These registers are used to store intermediate data and values during program execution.

Index Registers (IXR 1-3): The Index Registers are used in addressing modes to modify the addresses of operands during instruction execution.

1. Control Buttons

2. • IPL (Initial Program Load): This button loads instructions from an external file (usually loading.txt) into memory to initialize the machine. The simulator reads the encoded instructions when this button is hit, adding the necessary information to the memory addresses. By making sure the simulator has the instructions it needs to process, this gets it ready for execution.

• Programs 1 and 2: This button enables you to upload a file and see how many words it contains.

• ST+ (Step): The user can carry out the program one instruction at a time by pressing the ST+ button. The simulator retrieves, decodes, and executes the current instruction from memory when this button is hit.

- This step-by-step execution is useful for debugging and educational purposes, as it enables users to observe how each instruction affects the state of the registers and memory.
- Store: The Store button is used to save a value from a specified register into a memory location. When the user specifies a register and a memory address, pressing this button transfers the contents of the register to the designated address in memory. This functionality is essential for storing intermediate results or data that needs to persist beyond the current execution cycle.
- Load: The Load Button in the front panel plays a crucial role in manually updating the contents of various registers or memory locations. It allows the user to input specific values directly into a register or memory address, simulating the loading of data as part of the machine's operation. After entering the desired value into the corresponding text field for a particular register, clicking the Load Button will update that specific register or memory location with the new value, simulating a "load" operation as it would occur in the machine.

- SS (Start/Stop): The SS button is a toggle that starts or stops the execution of the loaded program. When pressed, it begins executing the instructions sequentially until a halt condition is reached or the button is pressed again to stop execution. This button provides a convenient way to control the flow of the simulation, allowing users to pause the execution for analysis or adjustments.
- Reset: The Reset button reinitializes the simulator, clearing all registers and resetting the program counter and memory to their initial states. This action allows users to start fresh without any residual data or instruction states from previous runs. It's particularly useful for testing different programs or scenarios without restarting the entire application.
- Run: The Run button executes the entire loaded program continuously from start to finish until a halt condition (HLT) is encountered or until the user decides to stop the execution. This functionality allows users to simulate a full execution cycle of a program, providing insight into how a complete set of instructions operates within the system.
- HLT (Halt): The HLT button stops the execution of the program immediately. When pressed, it signals the simulator to halt all operations, preventing any further instructions from being executed. This button is crucial for ending a simulation session, particularly when an infinite loop or unexpected behaviour occurs.

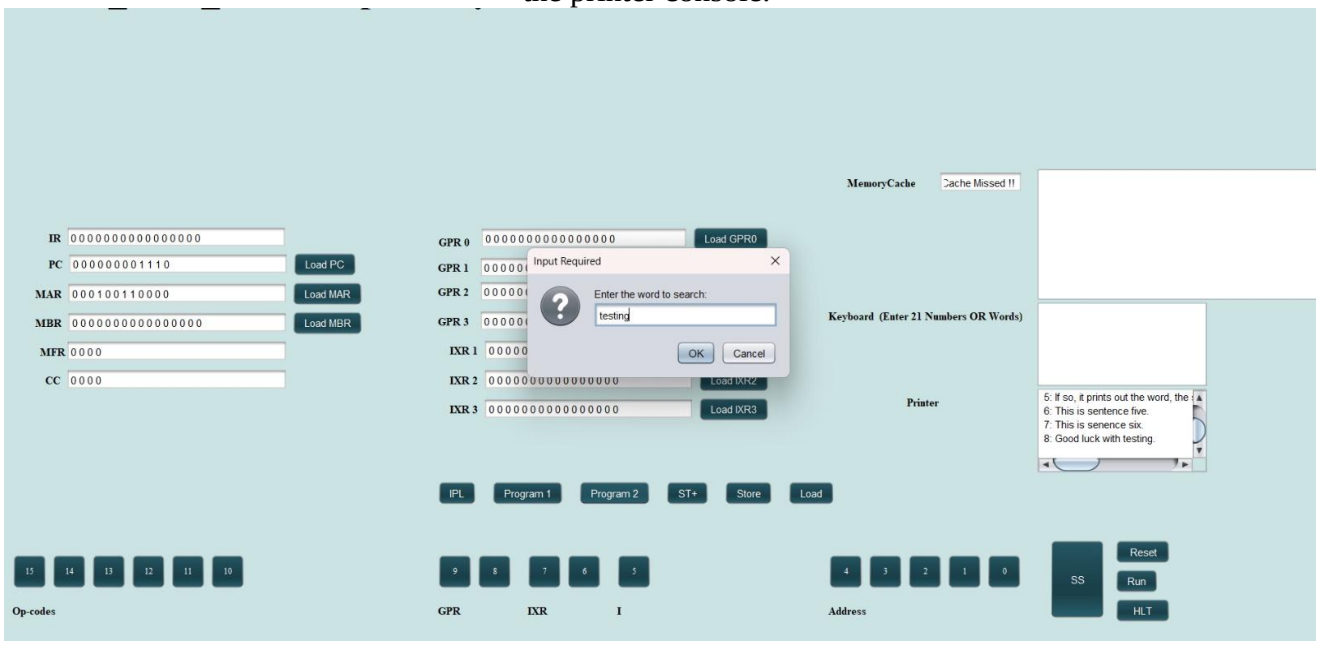
3. Cache Console

This cache console displays the most recent MAR (Memory Address Register) instruction executed. If you run the same instruction again, it results in a cache hit; otherwise, a cache miss occurs, and the console updates to store the latest instruction. The cache can hold up to 16 key-value pairs, and once it exceeds this limit, it uses a FIFO (First-In-First-Out) policy to free up space for new instructions.

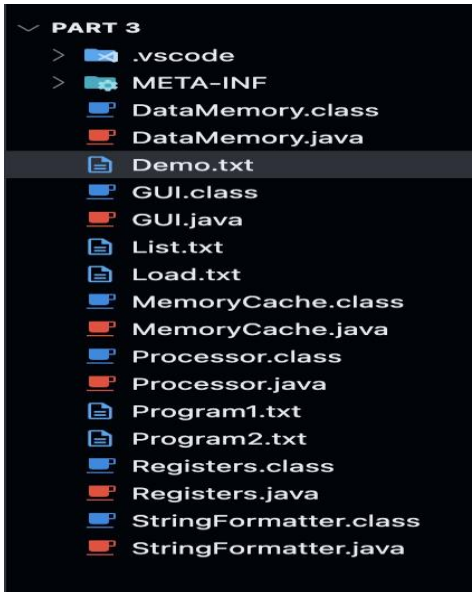
4. Keyboard and printer

This keyboard and printer setup is used to input and display values for running Program

Enter input through the keyboard area, and the output appears in the printer console.



Code Structure



1) Processor.java

This Processor.java file defines the structure and functionality of a basic Processor emulator. It creates keycomponents of the Processor, such as registers and memory, and implements logic for executing instructions.

Key Components:

- Registers: Several registers like PC, CC, IR, MAR, MBR, MFR, GPR, and IX are initialized, each with a specific size (bit length). These hold key data for execution.
- Memory: A memory object (dataMemory) is created to store values during instruction execution.

Important Functions:

1. execute(String type): This function simulates the Processor executing an instruction. It reads the program counter (PC), fetches the instruction from memory, and decodes and executes it. It handles opcodes like LDR, STR, LDA, LDX, STX, and HLT.
2. decodeOPCode(int[] binary_OPCode): Decodes the binary opcode from the instruction to determine the operation (LDR, STR, etc.).
3. getRegVals(String register) and setRegVals(String register, int[] value): These functions retrieve and set values in Processor registers.
4. getMemVal(int row) and setMemVal(int row, int[] value): These handle memory access, retrieving and storing values at specified memory locations.
5. computeEA(int[] instruction): Computes the effective address (EA) for an instruction, taking into account indirect and indexed addressing modes.
6. loadIPLFile(String path): Loads a program into memory from a file, initializing the memory contents based on input.
7. openFileChooser(): Provides a file chooser UI for selecting a file to load into memory.

The class simulates core Processor behaviors, including fetch-decode-execute cycles and basic memory management.

2) Registers.java

Register class simulates a Processor register, encapsulating its value and size while providing methods for interaction. Here are the key components and their functionalities:

Key Components:

1. Instance Variables:
    - private int[] registerValue: An array that holds the current value of the register.
    - private int registerSize: An integer representing the size (in bits) of the register.
2. Constructor:
    - public Register(int size): Initializes the register with a specified size by calling the initRegisterSize(size) method.
3. Getter Methods:

- public int[] getRegVals(): Returns the current value stored in the register.

4. private void init(int size): Returns the size of the register.
5. Setter Method:
    - public void setRegVals(int[] newValue): Sets the value of the register if thesize is not zero. If the register size is zero, it throws an IllegalStateException. The new value is copied into the register using Arrays.copyOf to ensure that modifications to the input array do not affect the register's internal state.

3) DataMemory.java

The DataMemory class simulates a memory structure for a Processor, consisting of 2048 words, each 16 bits. It provides methods to retrieve (getMemVal) and set (setMemVal) memory values while ensuring data integrity through array copying. The class encapsulates the memory representation and allows safe access to its contents, making it a fundamental component in Processor simulation.

4) StringFormatter.java

StringFormatter class provides utility functions for converting binary arrays to integers and hexadecimal strings to integers. It introduces a private helper method, hexToBinArray, which accepts a specified length for the binary array, streamlining the conversion of hexadecimal values to both 16-bit and 12-bit binary arrays through public wrapper methods. This enhances code reusabilityand maintainability while keeping the original functionality intact.

5) load.txt

The machine's front interface includes an Initial Program Load (init) button, which restarts the computer and loads the file loading.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.

6) Demo.txt

It contains the demo text which is used in the assembler to check for word search and also track the line of the sentence where the word is present.

7) GUI.java

This file contains the UI of the entire simulator, buttons, switches, registers display. It implements the on click functionality of the all the components, updates memory and runs the instructions. Basically, it’s the powerhouse of the entire project.

Main functions in this file are-

- **loadST() –** loads the memory[MAR] =MBR
- **StorePlusButtonClick -** loads the memory[MAR] =MBR and automatically increases MAR value
- **HaltButtonClick –** stops the program running
- **initButtonClick –** loads the pre-program into the memory
- **UltimateLoadAction –** used to run instructions stored at memory[PC]
- **ResetButtonClick –** resets the entire memory

8) MemoryCache.java

The `MemoryCache` class is a straightforward, fixed-size cache using a Least Recently Used (LRU) strategy. Here's a detailed explanation of its components and purpose:
The cache aims to store a limited number of key-value pairs. When it reaches its maximum capacity, the oldest entry is removed to accommodate new data, following the LRU policy. This ensures that frequently accessed data remains available in memory, enhancing data retrieval efficiency.
1. Fields:
- `cache`: A `ConcurrentHashMap` that stores key-value pairs. It is thread-safe, allowing multiple threads to interact with it without extra synchronization.
- `capacity`: The maximum number of entries the cache can hold.
- `keysOrder`: A `Deque` (double-ended queue) that maintains the order of keys as they are added. This helps track the least recently used items.
2. Constructor:
- Initializes the `cache`, `capacity`, and `keysOrder`. The `capacity` parameter sets the maximum capacity of the cache.
3. Methods:
- `updateKeyOrder(String key)`: A private helper method that adds the key to the end of `keysOrder`. If the cache exceeds its maximum size, it removes the oldest key from both `keysOrder` and `cache`.
- `retrieveKeysOrder()`: Returns all keys in `keysOrder` as a comma-separated string. This is useful for displaying or logging the current cache content.
- `insertValue(String key, String value)`: Adds a key-value pair to the cache or updates the value if the key already exists. It calls `updateKeyOrder` to ensure the cache size stays within its limit.
- `removeValue(String key)`: Removes a key and its associated value from both `cache` and `keysOrder`.
- `fetchValue(String key)`: Retrieves the value associated with the given key. If the key is not in the
4. cache, it returns `null`.