CSCI6461 Section 10 Group Project Documentation: Part 2

Team Number: 09

Group Members:

1. Jugal Gajjar
2. Kamalasankari Subramaniakuppusamy
3. Karan Patel

Introduction:

Part 2 of the CSCI6461 Computer Architecture Project expands the simulator developed in Part 1 by adding a broader range of instructions and deeper functionality. This phase enhances the system's ability to handle complex operations, includes cache management, and introduces additional user interaction options. The simulator's interface has been refined to give users a detailed view of register states and memory usage, allowing for better debugging and interaction with the C6461 architecture's behavior.

Objectives of Part 2:

The objectives of Part 2 of the CS6461 Computer Architecture Project Simulator can be summarized as follows:

1. Expansion of Instruction Set: Implementation of a wider range of instructions, such as arithmetic, logical, shift, and branch operations, to more accurately simulate a computer's operational scope.
2. Enhanced Memory Management: Added methods for cache management and retrieval, adding features for memory handling.
3. Enhanced User Interface: Update to the control panel with more detailed insights into memory, error handling, and condition codes.
4. Error Handling and Debugging: Added error-handling capabilities, with clear alerts for overflow, underflow, division by zero, and memory faults.
5. Improved Execution Flow Control: Enhanced stepping and running options for detailed tracking of program flow.
6. Detailed Documentation: Thorough documentation that guides users through added functionalities and troubleshooting.

Design Structure:

1. Main Class ('Main.java'):
   - Serves as the entry point and initializes the simulator.
2. Assembler Class ('Assembler.java'):
   - Core of the assembler, responsible for translating assembly language into machine code.
   - Key Components:
     - Instruction Processing: Decodes and encodes each instruction based on the opcode and addressing modes.
     - Two-Pass Assembly: The assembler first scans to resolve memory locations and labels, then generates the final machine code.
     - Addressing Modes: Handles direct, indirect, and indexed addressing along with register and memory encoding.
     - Instruction Encoding: Encodes assembly instructions into a 16-bit binary format and converts them to octal.
   - Key Methods:
     - handleComments(): Removes comments from assembly code.
     - assembleInstruction(): Translates instructions based on opcode and parameters.
     - getOpcodeBinary(): Encodes opcodes into binary values.
3. Simulator Class ('Simulator.java')
   - Manages the graphical user interface (GUI) for interacting with the simulator, handling file input, instruction execution, and display updates.
   - Key Functions:
     - setup GUI: Creates the GUI using JTextField, JTextArea, JButton, and JLabel for registers and control buttons.
     - File Operations: Allows the user to load a program file in .txt format via a file chooser, and the instructions are processed sequentially.
     - Execution Control: Provides buttons for all instruction executions such as Add, Subtract, AND, OR, and other operations.
4. FileHandler Class ('FileHandler.java'):
   - Manages file input/output operations for the assembler.
   - Key Functions:
     - Reads assembly instructions from the input file.
     - Writes the generated machine code to the load file.
     - Creates the listing file that includes original instructions, machine code, and memory addresses.
5. Cache Class (Cache.java)
   - Implements a cache memory system with FIFO replacement policy to handle data access and storage efficiently.
   - Key Components:
     - Cache Lines: Each cache line stores tag, data array, valid, and dirty bits to maintain cache coherence and track modifications.
     - FIFO Replacement Queue: A queue that tracks cache line usage order to manage evictions.

    - Trace Logging: Logs cache activity (hits, misses, evictions) for debugging and performance analysis.
- Key Methods:
    - accessCache(int address): Checks for cache hit or miss based on tag and retrieves data if a hit, else fetches from memory.
    - fetchFromMemory(int address, CacheLine line): Fetches data from memory into the cache line and updates the FIFO queue.
    - evictCacheLine(): Removes the oldest cache line in the FIFO queue, clearing its data to make room for new entries.
    - addItemToCache(int address, int[] data): Adds new data to the cache and evicts the oldest cache line if the cache is full.
    - removeItemFromCache(int address): Clears a specific cache line if it matches the given address tag.
    - clearCache(): Resets all cache lines and clears the FIFO queue.
    - printCacheContents(): Prints the contents of all cache lines for review.
    - logTrace(String message): Logs cache events to a trace file for debugging purposes.
    - closeTrace(): Closes the trace writer to ensure all logs are saved before program termination.

Note: We have designed the cache mechanism which works using FIFO but it is yet to be integrated with the system since we are figuring out an effective way to integrate it. It will be done and submitted along with Part 3.

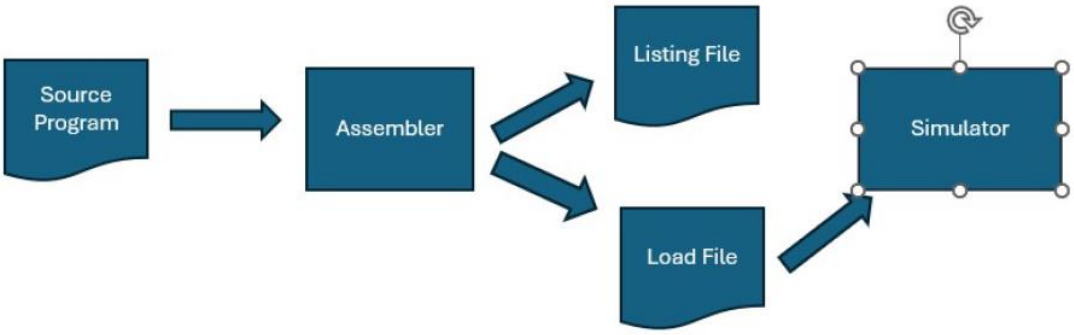


Fig. Overall Assembler Data Flow

GitHub Link:

https://github.com/JugalGajjar/CSCI6461-Computer-System-Architecture-Project/

Steps to Use:

1. Install OpenJDK 17.0.7
2. Execute .jar file. (java -jar CSCI6461.jar)
3. Place the program1.txt in an accessible directory.

To Execute Other Instructions:

1. Execute the jar file to start the simulator.
2. Press IPL button to initialize the simulator.
3. Load the register and index register for the instruction execution. (To load values in them, enter octal value in the octal field and convert it to binary number. Then, press the 'Load' button corresponding to the register to load value in them.)
4. Enter instruction parameters in the console input separated by comma(s).
5. Press the corresponding instruction button to execute the instruction.

To Execute Program 1:

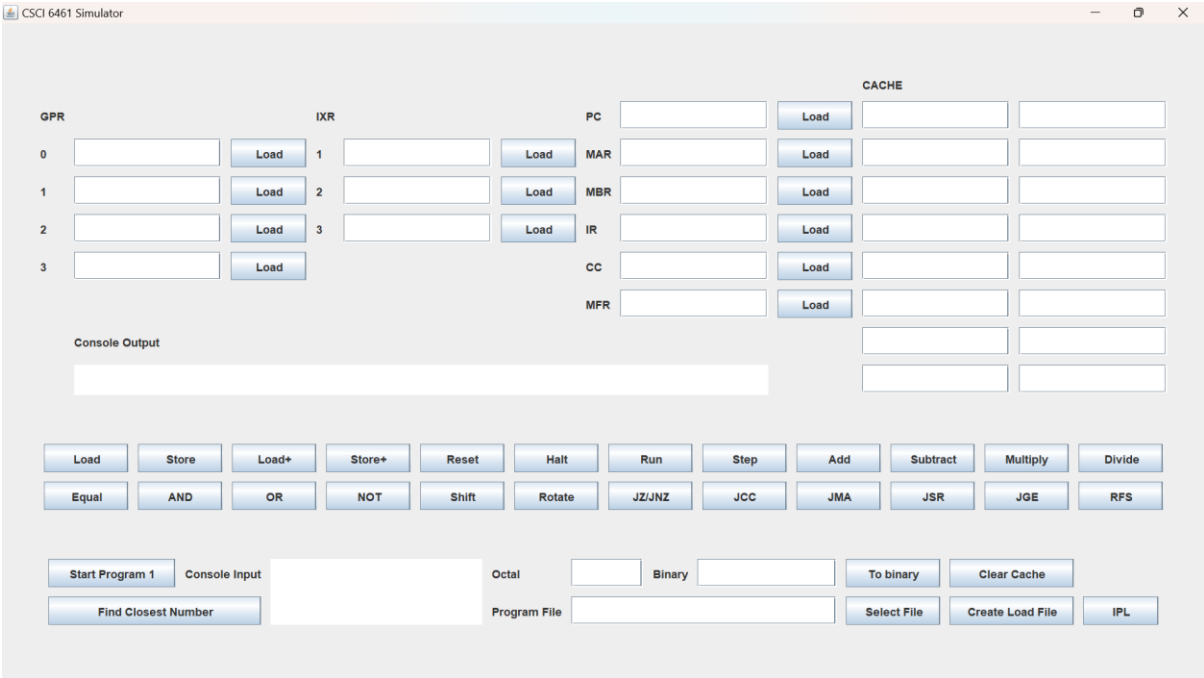
1. Execute the jar file to start the simulator.
2. Press IPL button to initialize the simulator.
3. Enter 20 numbers separated by comma in the console input and press 'Start Program 1' button.
4. Enter the number whose closest number we have to find in the console input and press 'Find Closest Number' button.

The closest number will be internally stored in memory location 1000 from where the program will access it and display result in output area.

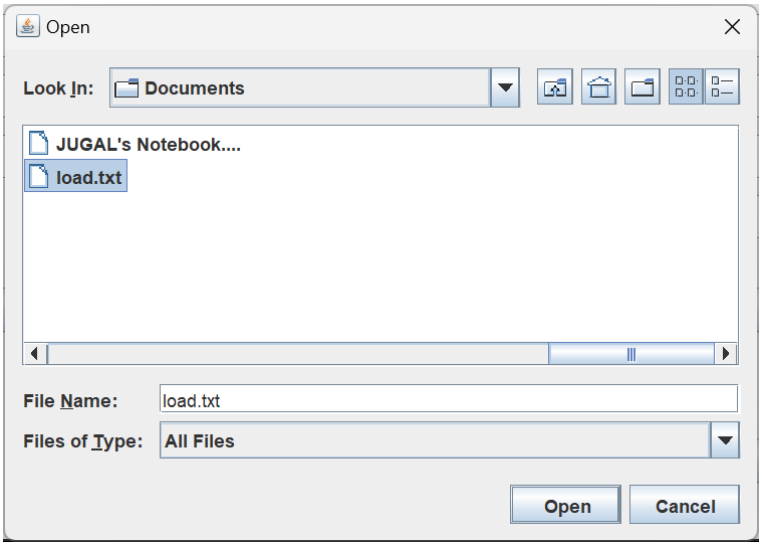
Important Note:

The program1.txt will be dynamically modified by the data entered in the console input and corresponding load file will be generated automatically once all data values has been filled in.

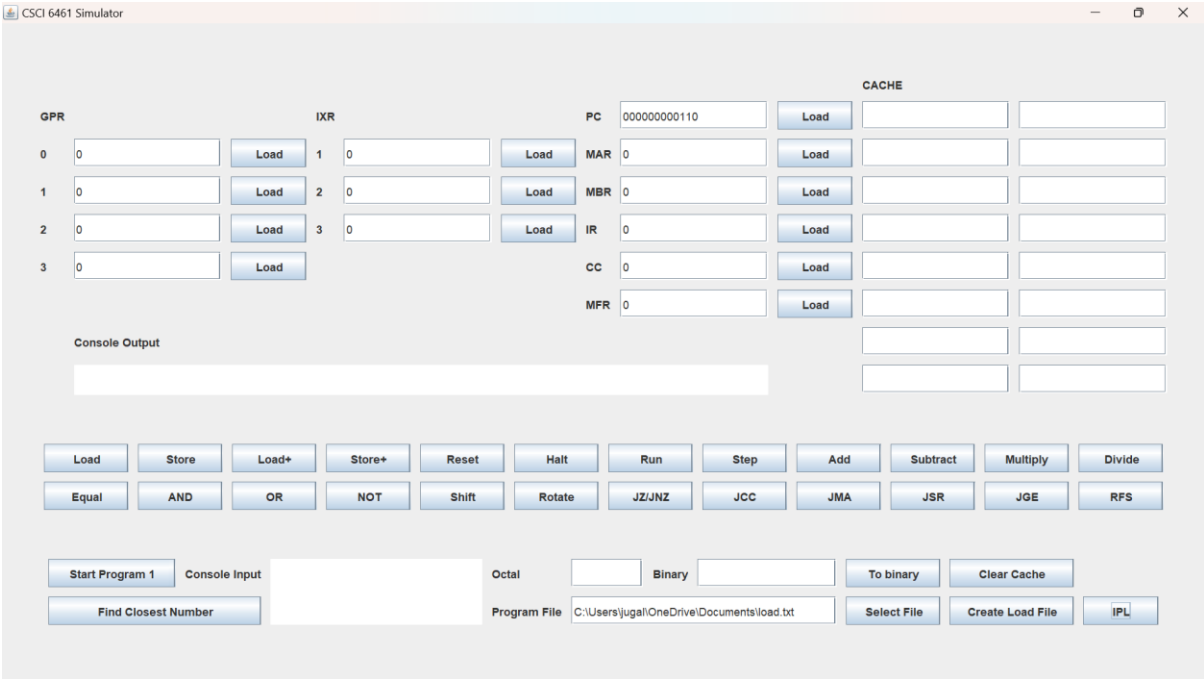
Interface Components:



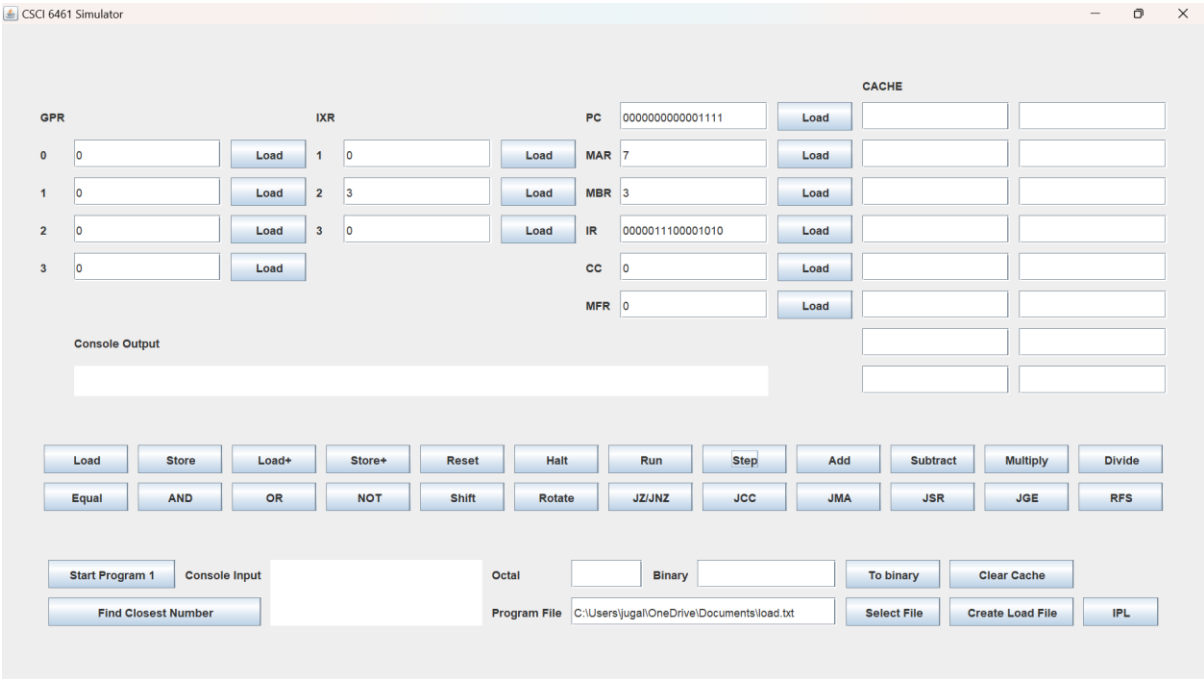
Console



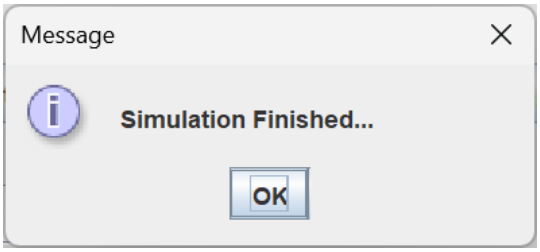
File Selector



Initialized Simulator

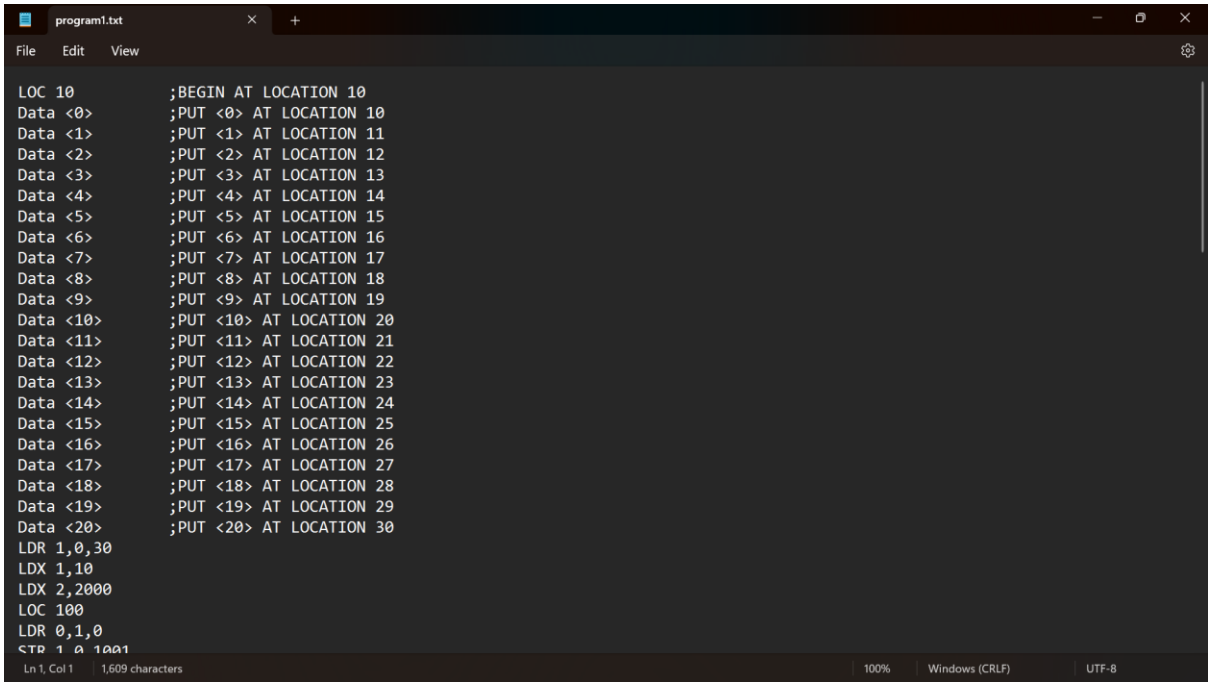


Some intermediate step with using ‘Step’



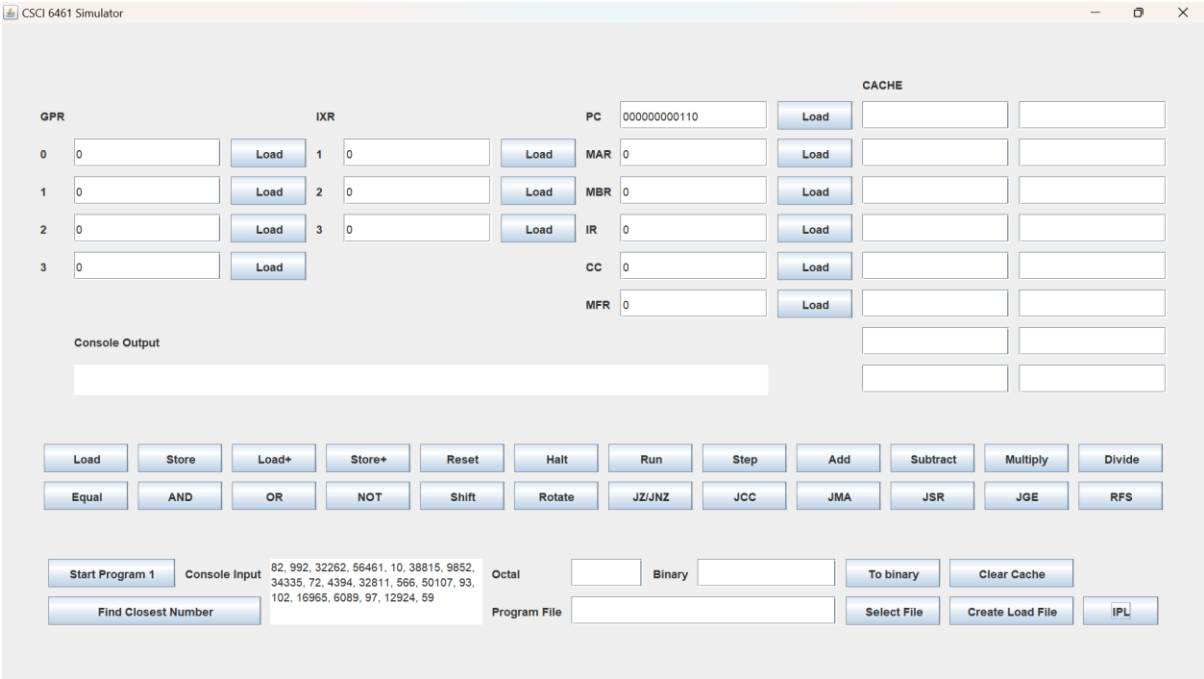
Message received upon successful 'Run'

Step-by-Step Screenshots of Program 1 Execution:

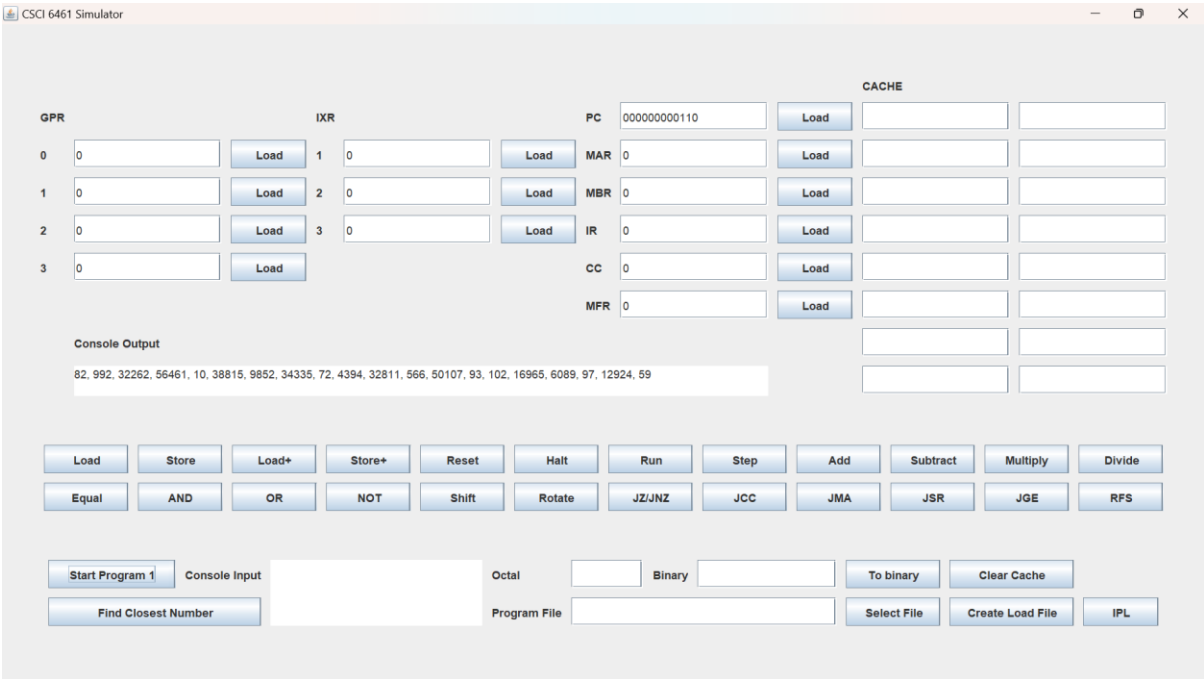


```
LOC 10          ;BEGIN AT LOCATION 10
Data <0>        ;PUT <0> AT LOCATION 10
Data <1>        ;PUT <1> AT LOCATION 11
Data <2>        ;PUT <2> AT LOCATION 12
Data <3>        ;PUT <3> AT LOCATION 13
Data <4>        ;PUT <4> AT LOCATION 14
Data <5>        ;PUT <5> AT LOCATION 15
Data <6>        ;PUT <6> AT LOCATION 16
Data <7>        ;PUT <7> AT LOCATION 17
Data <8>        ;PUT <8> AT LOCATION 18
Data <9>        ;PUT <9> AT LOCATION 19
Data <10>       ;PUT <10> AT LOCATION 20
Data <11>       ;PUT <11> AT LOCATION 21
Data <12>       ;PUT <12> AT LOCATION 22
Data <13>       ;PUT <13> AT LOCATION 23
Data <14>       ;PUT <14> AT LOCATION 24
Data <15>       ;PUT <15> AT LOCATION 25
Data <16>       ;PUT <16> AT LOCATION 26
Data <17>       ;PUT <17> AT LOCATION 27
Data <18>       ;PUT <18> AT LOCATION 28
Data <19>       ;PUT <19> AT LOCATION 29
Data <20>       ;PUT <20> AT LOCATION 30
LDR 1,0,30
LDX 1,10
LDX 2,2000
LOC 100
LDR 0,1,0
STR 1,0,1001
```

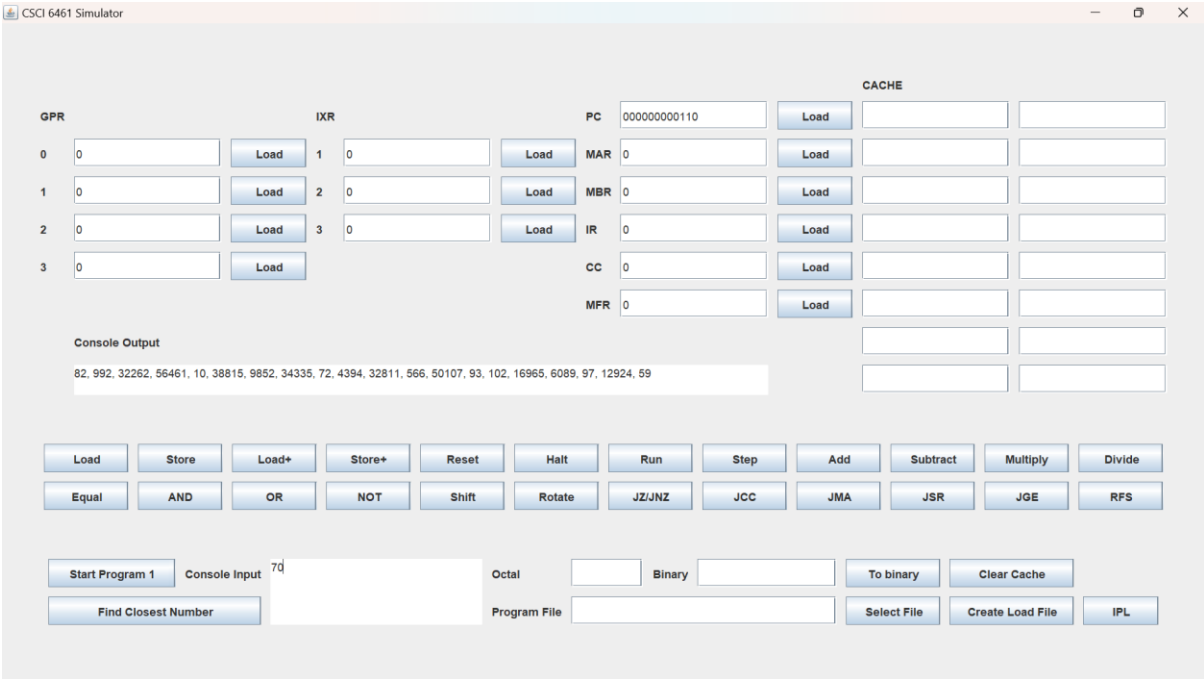
Program1.txt in which the placeholders will get dynamically updated in memory when user enters the input values



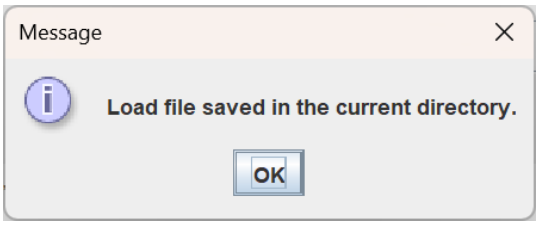
Entering 20 input numbers



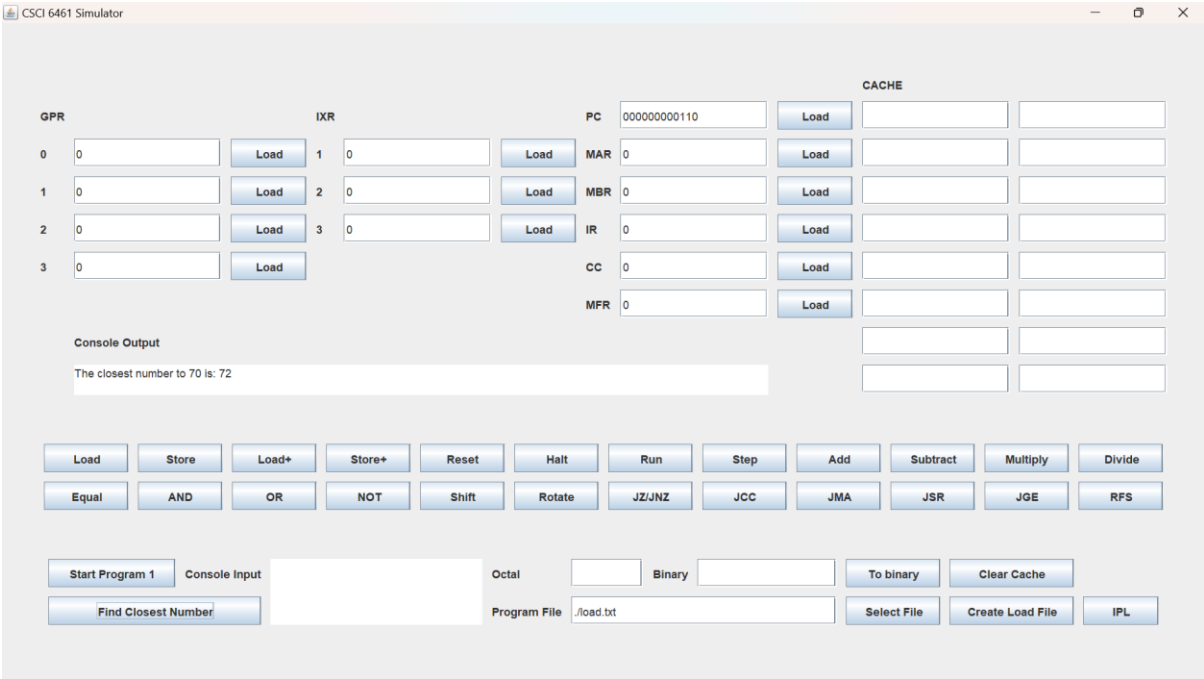
Input values printed on the console output



Entering number whose closest value we intend to find

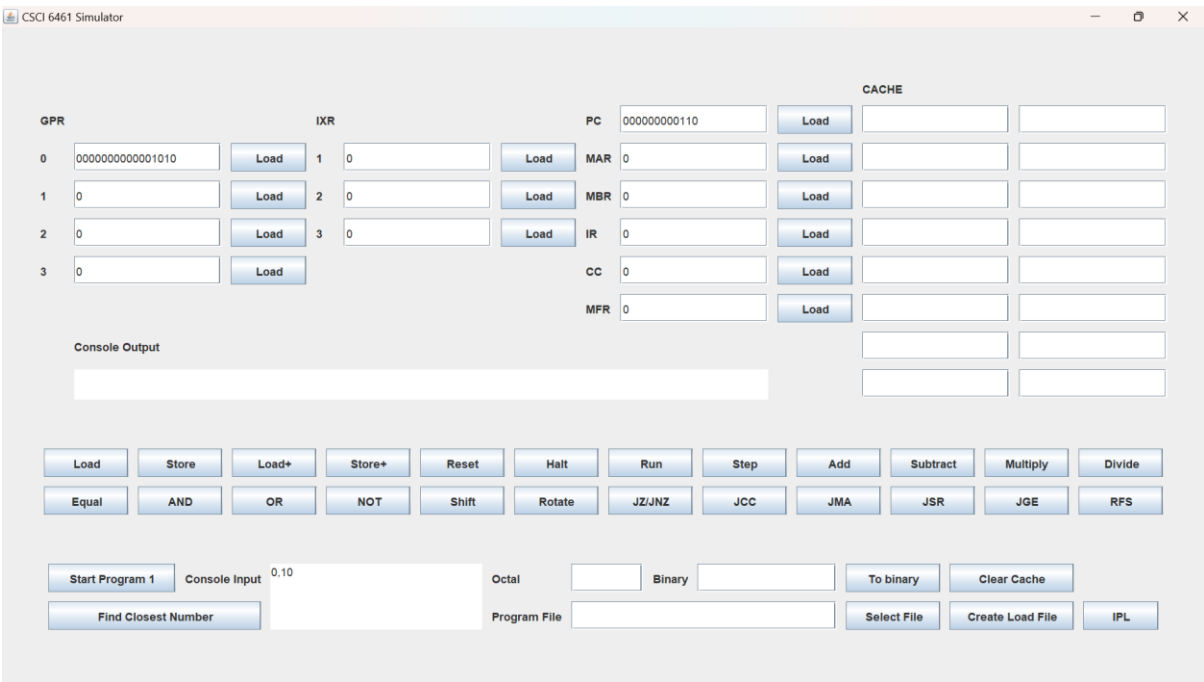


Successful dynamic creation of load file for program1

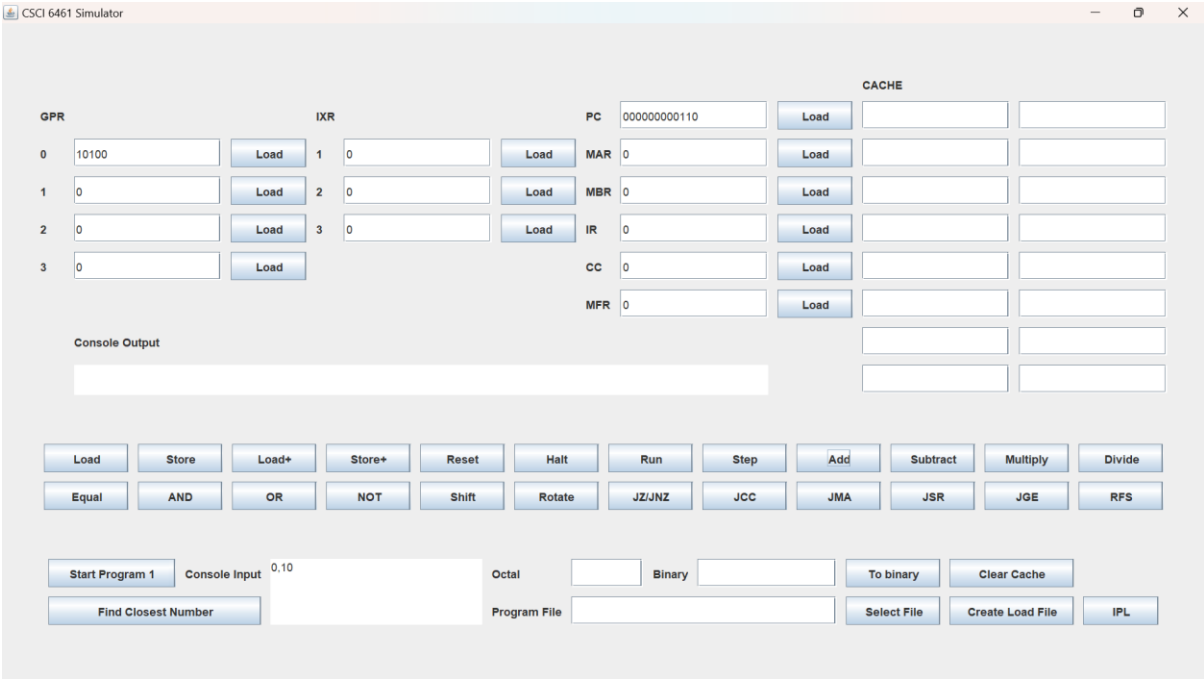


Final output on the console

Demonstrating use of ADD instruction as POC (Proof of Concept):



Before ADD Immed (AIR) -- R0 = 10 (in binary), Immed = 10 (Decimal)



After ADD Immed (AIR) -- 10 + 10 = 20 -- R0 = 20 (in binary)

1. **General Purpose Registers (GPR)**
   - There are four registers (labeled 0–3), each capable of holding a 16-bit binary value.
   - Each register includes a yellow button that allows users to load values directly from the Binary input field.
2. **Index Registers (IXR)**
   - Comprising three index registers (1–3), each also supports a 16-bit binary value.
   - Like the GPRs, each index register features a yellow button for value loading.
3. **Essential Registers**
   - **Program Counter (PC)**: This register points to the address of the next instruction to be executed. When updated, it retrieves the relevant instruction from memory and places it in the Instruction Register.
   - **Memory Address Register (MAR)**: This register contains the address of the word that will be accessed in memory.
   - **Memory Buffer Register (MBR)**: This stores the last word fetched from memory or the word that was just written back.
   - **Instruction Register (IR)**: This displays the instruction ready for execution. Each time the PC updates, the corresponding instruction appears in the IR and executes when either the Step or Run button is pressed.
   - **Condition Code (CC)**: This will be implemented with arithmetic and logical operations and includes four 1-bit flags: O (overflow), U (underflow), D (division by zero), and E (indicating equality).

- **Machine Fault Register (MFR)**: This shows the machine fault code if an error occurs, with complete functionality expected in later updates.

Buttons and Controls

1. **Load (under GPR, IXR, PC, MAR, MBR, IR, CC, MFR)**: Transfers the current input value into the respective register, enabling manual input into each specific register.
2. **Console Output**: Displays messages and output from the simulator, helping users see results of operations or debug information in real-time.
3. **Arithmetic Operations**:
   - Add: Adds values in specified registers or memory locations.
   - Subtract: Subtracts values from registers or memory locations.
   - Multiply: Multiplies values in registers.
   - Divide: Divides values between registers.
4. **Logic Operations**:
   - Equal: Compares values in registers or memory locations.
   - AND: Performs a logical AND operation on values.
   - OR: Executes a logical OR operation on values.
   - NOT: Applies a logical NOT to the selected register value.
5. **Shift and Rotate**:
   - Shift: Shifts the bits in the specified register left or right.
   - Rotate: Rotates bits in the selected register.
6. **Control Operations**:
   - Reset: Resets all registers and memory to their default values.
   - Halt: Stops the current execution in the simulator.
   - Run: Executes the program continuously until it completes or hits a halt.
   - Step: Executes one instruction at a time, helpful for debugging.
7. **Conditional Jumps**:
   - JZ/JNZ: Jumps based on zero (JZ) or non-zero (JNZ) conditions.
   - JCC: Conditional jump based on the condition code.
   - JMA: Unconditional jump to a specified memory address.
   - JSR: Jumps to subroutine.
   - JGE: Jumps if the register is greater than or equal to a value.
   - RFS: Return from subroutine.
8. **File and Program Controls**:
   - Start Program 1: Initiates Program 1 in memory.
   - Console Input: Field to enter user commands or data for the program.
   - Program File: Field displaying the selected file path for program loading.

- Select File: Allows users to choose a program file to load into the simulator.
- Create Load File: Creates a file from the current setup for reloading later.

9. **Cache Controls**:
    - Clear Cache: Clears the contents of the cache, resetting it to its initial state.
10. **Numerical Conversion**:
    - Octal Input/Conversion: Allows entry of octal values. The To binary button converts octal to binary.
    - Binary Input: Accepts binary values for direct entry into registers.
11. **Find Closest Number**: Searches for and displays the closest number based on the given input values.

Memory and Register Management:

- **GPR (General Purpose Registers)**: Four registers (0 to 3) used for general operations and data storage.
- **IXR (Index Registers)**: Three index registers (1 to 3) used for addressing modes and calculations.
- **PC (Program Counter)**: Stores the address of the next instruction to execute.
- **MAR (Memory Address Register)**: Holds the address for memory read/write operations.
- **MBR (Memory Buffer Register)**: Holds data to be transferred to/from memory.
- **IR (Instruction Register)**: Stores the currently executing instruction.
- **CC (Condition Code Register)**: Contains flags for conditions like zero, overflow, etc.
- **MFR (Machine Fault Register)**: Stores error codes or fault information from memory and execution errors.

Conclusion:

The CSCI 6461 simulator is a tool that replicates the functioning of the C6461 computer architecture by executing machine-level instructions. It processes program files, decodes opcodes, and performs operations on registers and memory, providing a real-time view of how each instruction is executed. The simulator not only supports various instruction types but also facilitates debugging through its step-by-step execution feature. By offering an interactive interface, this simulator enables users to gain a deeper understanding of how the C6461 system manages memory, processes instructions, and handles control flow. Overall, this project serves as a foundation for experimenting with low-level operations and understanding the internal workings of computer architecture.