CSCI 6461 Computer Architecture - Project Part 2
Design Notes and User Guide: Enhanced CPU and Cache Memory

Submitted By: *Sribalaji Annamalai Senthilkumar, Abenezer Golda, Razia Noor, Christabell Rego*

1. Overview

Part 2 of the project builds directly on the basic machine created in Part 1, transforming it into a fully functional CPU. The primary objectives of this part were threefold:

1. **Complete the Instruction Set Architecture (ISA):** Implement all remaining instructions as specified in the project description, including all arithmetic (AMR, SMR, AIR, SIR), register-to-register (MLT, DVD, TRR, AND, ORR, NOT), jump (JZ, JNE, JCC, JMA, JSR, RFS, SOB, JGE), and I/O (IN, OUT, TRAP) operations.
2. **Implement Cache Memory:** Design and integrate a 16-line, fully associative cache with a FIFO (First-In, First-Out) replacement policy to sit between the CPU and main memory.
3. **Run Program 1:** Write and successfully execute the first test program, which involves reading 20 numbers, performing calculations, and using I/O instructions to interact with the user.

This document focuses on the design and justification of the cache, its integration into the CPU, and its demonstration via the GUI.

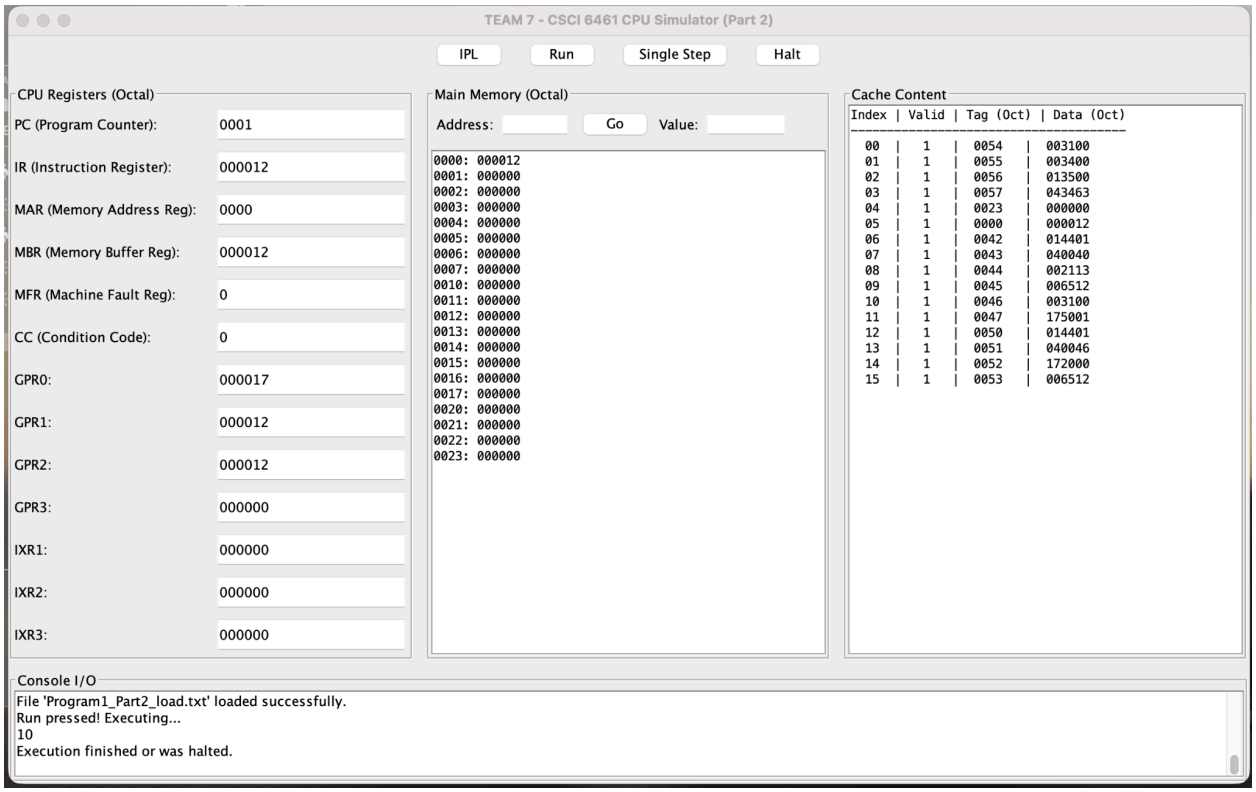
2. Design Justification

The core logic was expanded in `CPU.java` to support the new cache and full instruction set, with corresponding UI additions in `SimulatorGUI.java`.

Core Component: `CPU.java`
A. Instruction Set Implementation

- **Fetch-Decode-Execute:** The `executeInstruction()` method was expanded with a `switch` statement to handle the opcodes for all new instructions.
- **Decoding:** Additional bitwise decoding logic was added to parse fields for register-to-register operations, I/O device IDs, and shift/rotate counts.
- **I/O Handling:** The `IN` and `OUT` instructions are handled using Java Swing components. `OUT` appends text to the `consoleOutputArea`. `IN` required a thread-safe implementation

using SwingUtilities.invokeAndWait to pause the SwingWorker (run) thread while showing a JOptionPane on the main GUI thread, preventing concurrency errors.



B. Cache Memory Implementation

To meet the project requirement for a 16-line, fully associative cache with FIFO replacement, the following design was implemented:

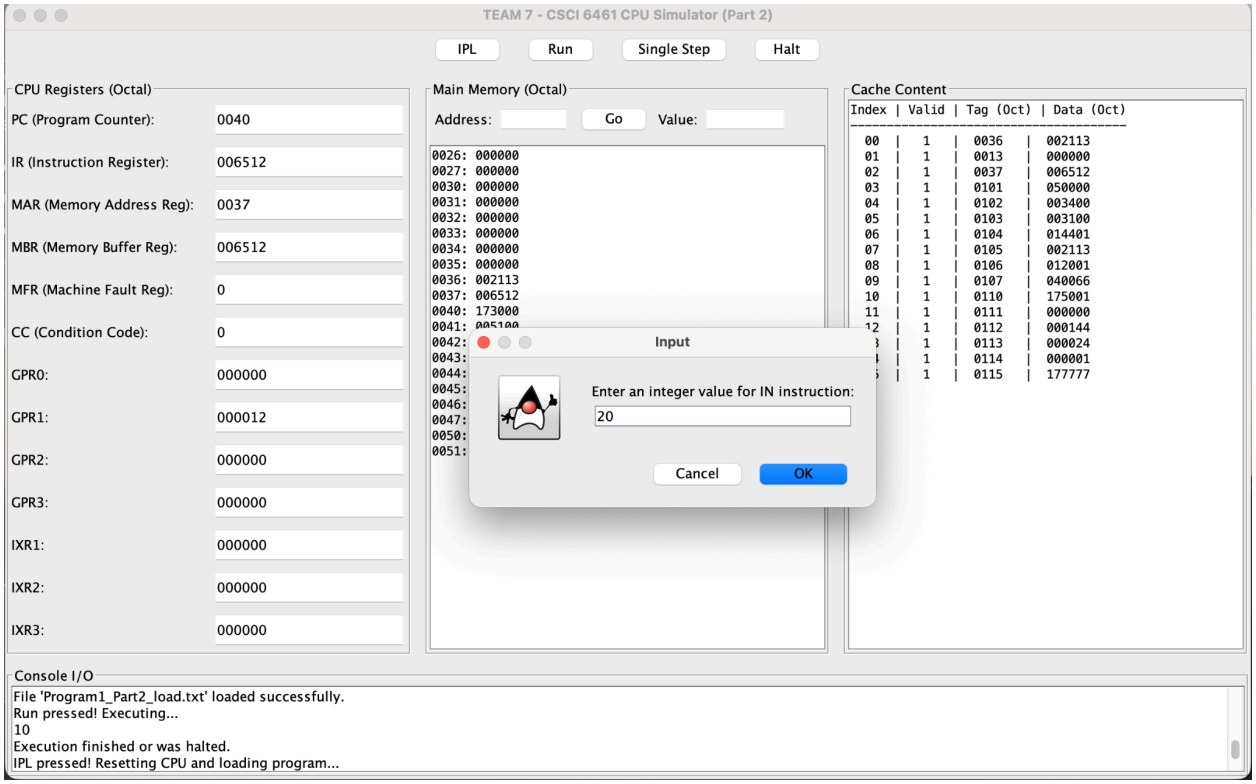
- **Data Structures:**
  - **CacheLine Inner Class:** A private inner class, CacheLine, was created inside CPU.java to represent a single line in the cache. It contains three fields: int tag (the main memory address), int data (the 16-bit value at that address), and boolean valid (to indicate if the line holds active data).
  - **Cache Array:** A 16-element array, private final CacheLine[] cache = new CacheLine[CACHE_SIZE], serves as the cache itself.
  - **FIFO Queue:** A java.util.LinkedList<Integer> fifoQueue is used to manage the replacement policy. It stores the *indices* (0-15) of the cache array.

- **Cache Logic (Methods):**
    - **`readMemory(int address)`:** This method was modified to implement the cache logic.
        1. **Cache Hit:** It iterates through all 16 `cache` array entries. If it finds a line where `valid == true` and `tag == address`, it returns the data from that cache line immediately.
        2. **Cache Miss:** If no matching tag is found, it proceeds to read the data from `memory[address]`. It then calls `addToCache(address, data)` to store the new data before returning it.

    - **`writeMemory(int address, int value)`:** This method was modified to implement a **Write-Through** and **Write-Allocate** policy.

        1. **Write-Through:** The data is *always* written directly to main memory at `memory[address] = MBR`.
        2. **Cache Update (Hit):** After writing to memory, it checks if the `address` is currently in the cache. If so, it updates the `cache[i].data` with the new value.
        3. **Cache Update (Miss - Write-Allocate):** If the address is not in the cache, it calls `addToCache(address, value)` to load the newly written value into the cache.

    - **`addToCache(int address, int data)`:** This private method handles the cache population and FIFO replacement.
        1. **Invalid Line:** It first searches for an invalid line (`valid == false`). If one is found, it sets the `tag`, `data`, and `valid` fields, and adds the line's `index` to the end of the `fifoQueue`.
        2. **FIFO Replacement:** If all lines are valid, it uses `fifoQueue.poll()` to remove the *oldest* index from the front of the queue. This `replaceIndex` is used to overwrite the oldest cache line with the new `tag` and `data`. The `replaceIndex` is then added to the *end* of the `fifoQueue` with `fifoQueue.add()`, marking it as the newest entry.



3. User Guide: How to Observe the Cache

The SimulatorGUI.java class was updated with a new panel to visualize the state of the cache in real-time.

- **New GUI Panel:** A third column, **"Cache Content,"** was added to the simulator's main window. This JTextArea is populated by calling a new public method in the CPU, getCacheContents().
- **updateGUI():** The updateGUI() method in SimulatorGUI.java is now called after every instruction (on Single Step or during the Run loop). This method calls cpu.getCacheContents() and sets the text of the cacheDisplayArea, so you can see the cache state change on every cycle.

Demonstrating the Cache with Program 1:

1. **Launch** the simulator JAR.
2. Click **IPL** and load the Program1_Part2_load.txt file.
3. **Observe the Cache:** Initially, the cache is empty (all "Valid" bits are 0). After loading, the cache panel will show the 16 most recently loaded lines of the program.
4. Click **Run**.

5. As you enter numbers for Program 1, you will see the cache lines change dynamically.
    - When the `STR 2,1,0` instruction writes your input to memory (e.g., at address `0100`, `0101`, etc.), you will see those addresses and their data appear in the cache.
    - When the program loops, it re-fetches instructions (`LDR`, `SOB`, etc.). If these instructions are still in the cache (a **hit**), memory is not accessed. If they have been replaced (a **miss**), they will be re-loaded from memory and added back to the cache, forcing the oldest line out.

This visualization directly demonstrates the cache's operation, showing its contents (Tag and Data), its valid status, and the FIFO replacement policy in action as the program runs.