MIPS Architecture Design

Course Name: EGC 121, Computer Architecture Processor Design

Pragyan Ojha
BC2025075

February 21, 2026

International Institute of Information Technology, Bangalore

Contents

1 Introduction

This report outlines the design and implementation of a 5-stage non-pipelined MIPS processor simulator. The processor is designed to execute 32-bit machine code generated by the MARS assembler. To demonstrate the functionality of the datapath, the `lower_bound` algorithm (commonly found in the C++ Standard Template Library) was implemented in MIPS assembly. This algorithm efficiently finds the first element in a sorted array that is greater than or equal to a target value using binary search.

2 Assembly Program

2.1 Algorithm Selection: `lower_bound`

The `lower_bound` binary search algorithm was chosen because it provides a robust test for the processor's control flow and memory access. It strictly fulfills the assignment requirements as well.

2.2 Original C++ Code

The assembly implementation is based on the following standard C++ logic for the lower bound algorithm:

```cpp
#include <iostream>
#include <vector>
using namespace std;

int lower_bound_custom(const vector<int>& arr, int target) {
    int low = 0;
    int high = arr.size();
    int ans = arr.size();

    while (low < high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] < target) {
            low = mid + 1;
        } else {
            ans = mid;
            high = mid;
        }
    }
    return ans;
}

int main() {
    vector<int> array = {1, 3, 5, 7, 9, 11, 13, 15};
    int target = 6;

    int result = lower_bound_custom(array, target);
    cout << "Lower Bound Index: " << result << endl;

    return 0;
}
```

lower_bound.cpp

2.3 Code Explanation

Instead of linear traversal, the algorithm initializes a `low` pointer to 0 and a `high` pointer to the array's length. Within a loop, the midpoint is calculated. If the element at the midpoint is strictly less than the target, the search space is narrowed to the right half. Otherwise, the current midpoint is saved as a potential answer, and the search space is narrowed to the left half. The final index is written back to memory and printed via system calls.
This implementation utilizes all required instruction categories. Memory access uses *lw* to fetch variables and array elements, and *sw* to store the final computed index. Control flow is handled by conditional branches (*bge*, *blt*) and unconditional jumps (*j*) for the binary search loop. Finally, ALU operations (*add*, *sub*, *sll*, *srl*) are used to calculate the midpoint and perform pointer arithmetic.

3 Processor Design

The simulator is implemented in Python, applying an object-oriented approach to represent the distinct hardware components.

3.1 The 5-Stage Datapath

The simulator processes instructions through five distinct stages sequentially:

1. **Instruction Fetch (IF):** Fetches the 32-bit instruction from the memory at the current Program Counter (PC) address and computes `PC + 4`.

2. **Instruction Decode (ID):** Extracts the opcode, source/target registers (`rs`, `rt`, `rd`), shift amount (`shamt`), and function code. Immediate values are sign-extended.

3. **Execute (EX):** The ALU executes the operation determined by the control unit. Branch targets and jump addresses are also calculated in this stage.

4. **Memory Access (MEM):** Handles reading from or writing to the data memory based on the `MemRead` and `MemWrite` control signals.

5. **Write Back (WB):** Writes the computed ALU result or fetched memory data back to the target register in the Register File.

3.2 Memory Architecture

The memory is strictly **byte-addressable**, segmented logically to mimic the MARS environment:

- **Text Segment:** Starts at address `0x00400000`, storing the executable machine code.

- **Data Segment:** Starts at address `0x10010000`, storing statically allocated variables such as the sorted array, target values, and string buffers.

3.3 Component Definitions

The simulator is built using a modular, object-oriented approach in Python. The hardware abstractions are defined by the following core components:

- **RegisterFile (`RegisterFile` class):** Simulates the 32 MIPS registers. It includes read and write methods, strictly enforcing that `$zero` remains immutable at 0, and applies a bitmask to ensure all stored values remain strictly 32-bit.
- **Arithmetic Logic Unit (`ALU` class):** A dedicated execution unit that handles arithmetic (ADD, SUB), logical (AND, OR), comparison (SLT), and shift operations (SLL, SRL, LUI). Like the Register File, it enforces 32-bit integer wrap-around on all computed results.
- **Memory Unit (`Memory` class):** Implements a true byte-addressable memory space using a dynamic Python dictionary mapping individual addresses to bytes. It abstracts memory access via `load_word` and `store_word` methods (handling 4-byte combinations) and includes a `load_string` method to support system calls.
- **Control & Datapath (`MIPS_Processor` class):** Acts as the primary orchestrator. It manages the Program Counter (PC), loads the `.text` and `.data` machine code segments, and sequentially steps through the 5 stages of instruction execution (IF, ID, EX, MEM, WB).

4 Assumptions and Constraints

- **Non-Pipelined Execution:** The processor completely finishes one instruction (all 5 stages) before fetching the next, meaning there are no data hazards or structural hazards to mitigate.
- **Syscall Handling:** System calls (print integer, print string, exit) are intercepted during the Execute stage for console output, mimicking MARS behavior without requiring lower-level OS simulation.
- **Memory Storage Limit:** While the addresses align with standard MIPS architecture, the memory is simulated using a Python dictionary, meaning memory is allocated dynamically only when accessed, rather than pre-allocating an entire 2^{32} byte space.

5 Execution and Results

The MIPS assembly file was first verified using the MARS assembler. The binary text dumps of the `.text` and `.data` segments were then fed into the Python simulator.

5.1 MARS Assembler Verification

Figure 1 shows the successful assembly of the code, confirming there are no syntax errors.

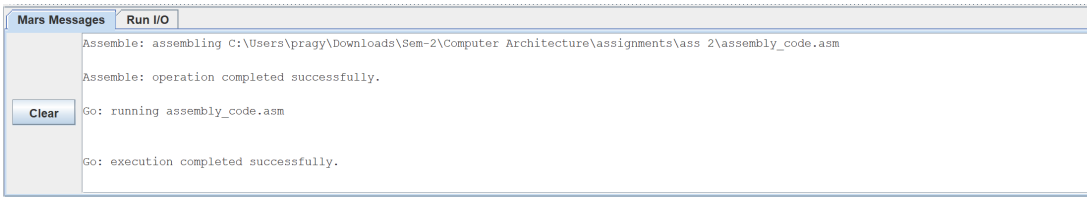

Figure 1: Successful assembly of the code in MARS

Figure 2 shows the `Run I/O` terminal after execution, confirming the logic is correct and executes without errors in the MARS Assembler.

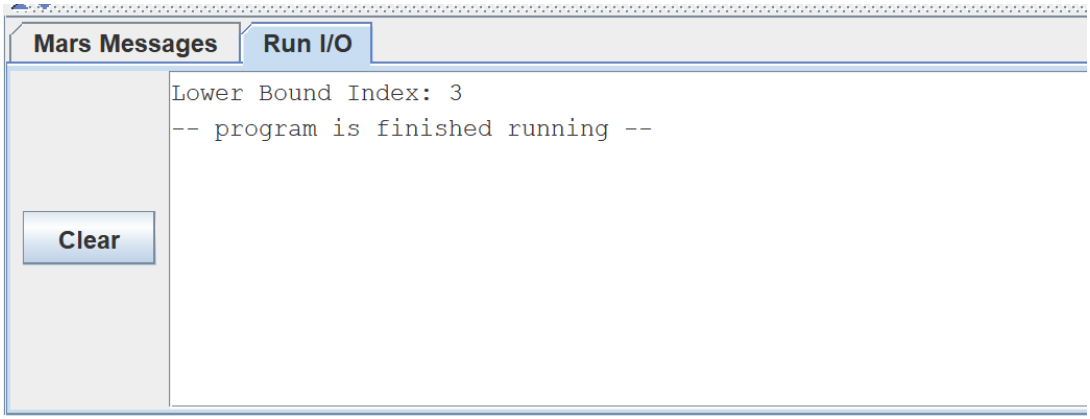

Figure 2: Successful execution and output in the MARS Run I/O tab

5.2 Simulator Execution Snapshots

The simulator successfully parses the machine code and matches the MARS output. The debug trace showcases the changing state of the Program Counter and ALU results across the execution cycles.

```
--- Cycle 6 | PC: 0x400014 ---
[IF]  Fetched Inst: 0x8c320024
[ID]  Opcode: 35, rs: 1, rt: 18, rd: 0, imm: 36
[EX]  ALU Op: ADD, Result: 268501028
[MEM] Read value 6 from addr 0x10010024
[WB]  Wrote value 6 to R18

--- Cycle 7 | PC: 0x400018 ---
[IF]  Fetched Inst: 0x24080000
[ID]  Opcode: 9, rs: 0, rt: 8, rd: 0, imm: 0
[EX]  ALU Op: ADD, Result: 0
[MEM] No memory access
[WB]  Wrote value 0 to R8

--- Cycle 8 | PC: 0x40001c ---
[IF]  Fetched Inst: 0x116021
[ID]  Opcode: 0, rs: 0, rt: 17, rd: 12, imm: 24609
[EX]  ALU Op: ADD, Result: 8
[MEM] No memory access
[WB]  Wrote value 8 to R12

--- Cycle 9 | PC: 0x400020 ---
[IF]  Fetched Inst: 0x111082a
[ID]  Opcode: 0, rs: 8, rt: 17, rd: 1, imm: 2090
[EX]  ALU Op: SLT, Result: 1
[MEM] No memory access
[WB]  Wrote value 1 to R1

--- Cycle 10 | PC: 0x400024 ---
[IF]  Fetched Inst: 0x1020000d
[ID]  Opcode: 4, rs: 1, rt: 0, rd: 0, imm: 13
[EX]  Branch not taken
[MEM] No memory access
[WB]  No write back

--- Cycle 11 | PC: 0x400028 ---
[IF]  Fetched Inst: 0x2284822
[ID]  Opcode: 0, rs: 17, rt: 8, rd: 9, imm: 18466
[EX]  ALU Op: SUB, Result: 8
[MEM] No memory access
[WB]  Wrote value 8 to R9
```

Figure 3: Python Simulator Cycle-by-Cycle trace

```
========================================
          SIMULATION FINISHED

[FINAL PROGRAM OUTPUT]
>>> Lower Bound Index: 3

[Register Dump]
R00-R03: [0, 268500992, 10, 0]
R04-R07: [3, 0, 0, 0]
R08-R11: [3, 3, 268501004, 7]
R12-R15: [3, 0, 0, 0]
R16-R19: [268500992, 3, 6, 0]
R20-R23: [0, 0, 0, 0]
R24-R27: [0, 0, 0, 0]
R28-R31: [0, 0, 0, 0]
PS C:\Users\pragy\Downloads\Sem-2\Computer Architecture\assignments\ass 2>
```

Figure 4: Final Output and Final Register File dump after execution