MIPS Processor Simulation

Presented by

Ashutosh Kumar (B22CS015)
Rhythm Baghel (B22CS042)
Souvik Maji (B22CS089)

Computer Architecture

Assignment:5

October 10, 2024

Contents

Chapter 1

Introduction

1.1 Problem Statement

- **Task1:** We implemented a MIPS compiler that reads and translates assembly instructions to binary machine code. The main focus was on handling R-type, I-type, and J-type instructions.
- **Task2:** The goal of Task 2 is to simulate the execution of MIPS binary instructions using a simulated MIPS processor. This includes simulating the MIPS datapath, control signals, ALU operations, memory access, and branching.
- **Task3:** Test the simulator with 5 different MIPS programs, including two provided and three that you write yourself. The aim is to challenge your simulator with complex programs and analyze the results.

1.2 Rust's Benefits

Using Rust for this task offers several benefits over other languages like C, C++, or Python:

- **Memory Safety:** Rust's ownership system prevents memory leaks and dangling pointers, critical for low-level operations like binary translation.
- **Error Handling:** Rust's type system and pattern matching help identify instruction format issues, providing clear error messages for invalid input.
- **Performance:** Rust offers near bare-metal performance, essential for efficient processing of large instruction sets.
- **Concurrency:** Rust's built-in concurrency support facilitates future extensions, like simulating multiple MIPS cores.
- **Type Safety:** Rust's strict type system prevents common bugs at compile-time, improving the reliability of the MIPS simulator.

Chapter 2

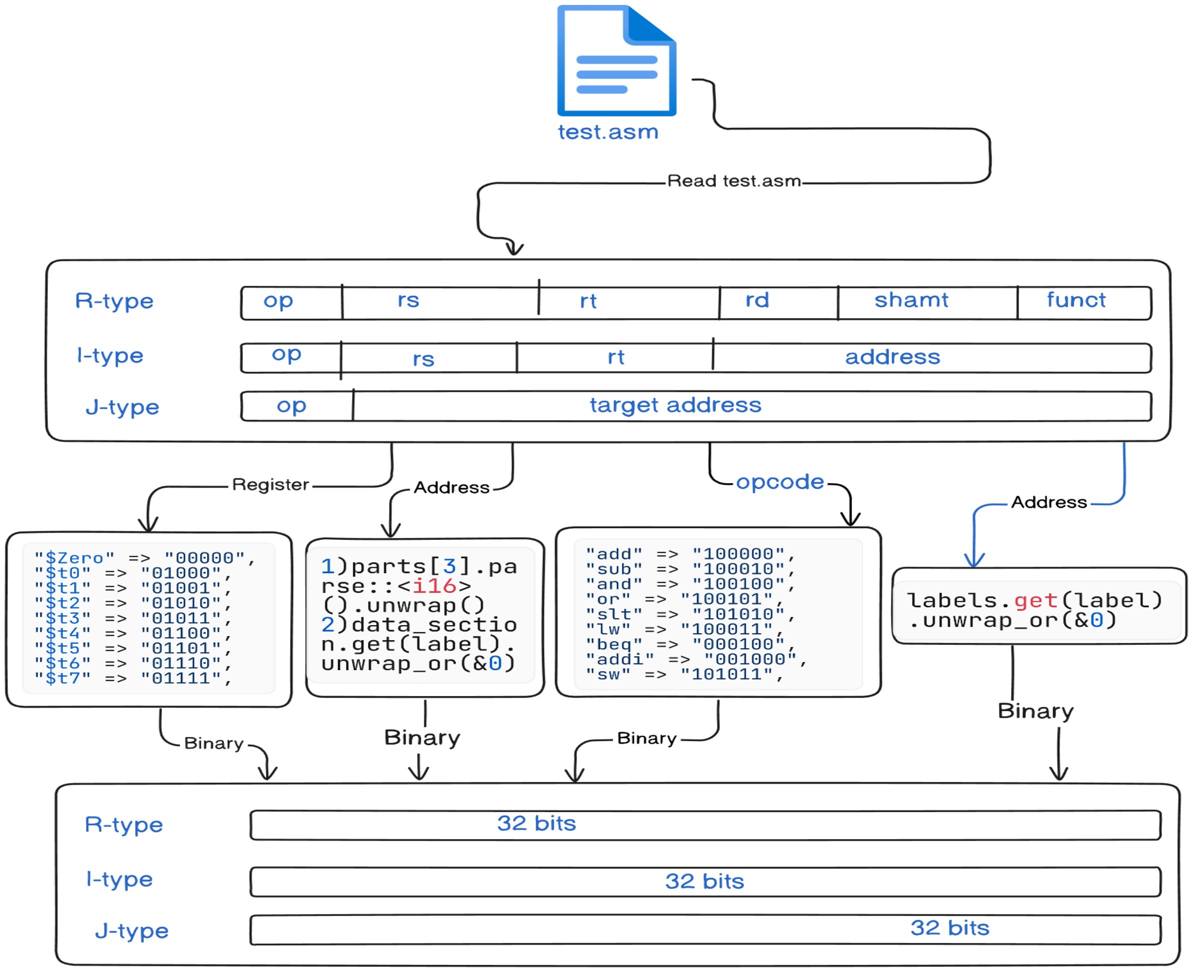
Task 1 (MIPS Compiler)

2.1 Objective:

The primary objective of Task 1 is to implement a MIPS compiler capable of reading MIPS assembly instructions and converting them into binary machine code. This task requires the handling of different instruction types (R-type, I-type, J-type) and ensuring the correct translation based on the MIPS architecture.

Steps Involved:

- **Reading Assembly Input:** The program reads a text file containing MIPS assembly instructions. This input includes both the .data and .text sections, which encompass memory allocation and executable instructions.
- **Parsing Instructions:** The compiler identifies the type of each instruction (R-type, I-type, J-type) by parsing the MIPS assembly code. For each instruction, it extracts the operation code (opcode), function code (for R-type), registers, and any immediate values involved.
- **Conversion to Binary:** The program then translates each parsed instruction into the corresponding binary machine code. This translation adheres to the MIPS instruction format, which includes:
  - R-type: Opcode (6 bits), Source and Destination Registers, Function code, Shift amount (if needed)
  - I-type: Opcode, Source, Destination Registers, and Immediate Value
  - J-type: Opcode and target address
- **Output:** The output is a binary representation of the MIPS assembly program, suitable for execution in a MIPS-like processor or simulator.



2.2 Coding:

- **Code file (without cargo):** main.rs
- **Command for run rust file in linux:**

```
rustc main.rs
./main
```

- Referance
- **1. Input and Output:**
  - **Code file (without cargo):**
  - **Input:** test1.asm

```
.data
    num1: .word 10
    num2: .word 20
.text
    lw $t0, num1
    lw $t1, num2
    add $t2, $t0, $t1
    sub $t3, $t1, $t0
```

```
    and $t4, $t0, $t1
    or  $t5, $t0, $t1
    slt $t6, $t0, $t1
    sw  $t6, num1
```

- **Output:**

```
PC: 0, Instruction: lw $t0, num1
Binary: 100011 00000 01000 10
PC: 1, Instruction: lw $t1, num2
Binary: 100011 00000 01001 20
PC: 2, Instruction: add $t2, $t0, $t1
Binary: 000000 01000 01001 01010 00000 100000
PC: 3, Instruction: sub $t3, $t1, $t0
Binary: 000000 01001 01000 01011 00000 100010
PC: 4, Instruction: and $t4, $t0, $t1
Binary: 000000 01000 01001 01100 00000 100100
PC: 5, Instruction: or  $t5, $t0, $t1
Binary: 000000 01000 01001 01101 00000 100101
PC: 6, Instruction: slt $t6, $t0, $t1
Binary: 000000 01000 01001 01110 00000 101010
PC: 7, Instruction: sw  $t6, num1
Binary: 101011 00000 01110 10
```

- **2. Input and Output:**
  - **Code file (without cargo):**
  - **Input:** test2.asm

```
.data
    val1: .word 5
    val2: .word 5
    result: .word 0
.text
    lw $t0, val1
    lw $t1, val2
    addi $t2, $t0, 10
    beq $t0, $t1, equal_case
    sub $t3, $t0, $t1
    sw  $t3, result
    j end
equal_case:
    add $t3, $t0, $t1
    sw  $t3, result
end:
```

  - **Output:**

```
PC: 0, Instruction: lw $t0, val1
Binary: 100011 00000 01000 5
PC: 1, Instruction: lw $t1, val2
Binary: 100011 00000 01001 5
PC: 2, Instruction: addi $t2, $t0, 10
Binary: 001000 01000 01010 0000000000001010
PC: 3, Instruction: beq $t0, $t1, equal_case
Binary: 000100 01000 01001 7
```

```
PC: 4, Instruction: sub $t3, $t0, $t1
Binary: 000000 01000 01001 01011 00000 100010
PC: 5, Instruction: sw  $t3, result
Binary: 101011 00000 01011 0
PC: 6, Instruction: j end
Binary: 000010 00000000000000000000001001
PC: 7, Instruction: add $t3, $t0, $t1
Binary: 000000 01000 01001 01011 00000 100000
PC: 8, Instruction: sw  $t3, result
Binary: 101011 00000 01011 0
```

2.3 Some Links:

- **With Cargo :** cargo
  - command for run cargo:

    ```
    cargo run
    ```

- **Without Cargo :** rustc
  - command for run main.rs:

    ```
    rustc main.rs
    ./main
    ```

Chapter 3

Task 2 (MIPS Execution Simulation)

3.1 Objective

The objective of this task is to develop a MIPS simulator in Rust that can:

- Simulate the execution of MIPS binary instructions.
- Accurately model the MIPS datapath and control signals.
- Implement instruction fetch, decode, execution, memory access, and write-back stages.
- Support ALU operations, memory access (`lw`, `sw`), and branch instructions (`beq`, `j`).
- Simulate the program counter (PC) and 32 general-purpose registers.
- Ensure correct reads and writes to registers and memory.

3.2 Steps Involved

3.2.1 Initialization and Data Structures

- **Define `DecodedInstruction` Struct:** Holds fields like `opcode`, `rs`, `rt`, `rd`, `funct`, `immediate`, and `address`.
- **Define `MipsSimulator` Struct:** Contains registers array, memory map, labels map, program counter (`pc`), program instructions, binary program, and register mappings.
- **Initialize Simulator:** Instantiate `MipsSimulator` with default values. Create `reg_map` and `reg_map_rev` for register name to index mapping.

3.2.2 Loading the Assembly Program

- **Implement `load_program_from_file` Method:**
  - Reads the assembly file line by line.
  - Handles directives like `.data` and `.text`.
  - Parses labels and associates them with addresses.
  - Parses instructions and stores them in the `program` vector.
  - Manages data storage in memory and updates `next_free_address`.

3.2.3 Assembling the Program

- **Implement `assemble_program` Method:**
  - Iterates over the `program` vector.
  - **Instruction Decoding:** Uses `instruction_decode_assembly` to split instructions into opcode and operands.
  - **Instruction Assembly:** Uses `assemble_instruction` to convert instructions into binary format.
  - Handles different instruction types (R-type, I-type, J-type).
  - Manages label addresses and immediate values.
  - **Printing Binary Instructions:** Uses `print_binary_instruction` to output formatted binary code.
  - Stores the binary instructions in `binary_program`.

3.2.4 Instruction Fetch and Decode

- **Implement `instruction_fetch` Method:**
  - Fetches the next instruction based on the program counter (`pc`).
- **Implement `instruction_decode` Method:**
  - Decodes the fetched binary instruction into its components.
  - Populates a `DecodedInstruction` instance.

3.2.5 Executing Instructions

- **Implement `execute_instruction` Method:**
  - Determines instruction type based on opcode and funct fields.
  - **ALU Operations:**
    - Implements methods like `execute_add`, `execute_sub`, `execute_and`, `execute_or`, `execute_slt`.
    - Performs arithmetic and logical operations.
  - **Immediate Operations:**

        - Implements `execute_addi` for immediate addition.
    - **Memory Access:**
        - Implements `execute_lw` and `execute_sw`.
        - Calculates effective addresses and performs memory reads/writes.
    - **Branching:**
        - Implements `execute_beq` for branch equal.
        - Updates the program counter if the branch condition is met.
    - **Jumping:**
        - Implements `execute_j` for unconditional jumps.
        - Directly updates the program counter to the target address.
    - **Program Counter Management:**
        - Increments `pc` unless modified by branch or jump instructions.
        - Returns a flag indicating whether `pc` was modified during execution.

3.2.6 Simulating Registers and Memory

- **Registers:**
    - Simulates 32 MIPS registers using an array.
    - Provides methods to retrieve register indices and names.
    - Ensures proper handling of register writes and reads.
- **Memory:**
    - Uses a `HashMap` to simulate main memory.
    - Manages data storage and retrieval during memory access instructions.
    - Associates labels with memory addresses for data sections.

3.2.7 Running the Simulator

- **Implement `run` Method:**
    - Loops through the `binary_program` using the program counter.
    - Fetches, decodes, and executes instructions.
    - Handles program termination when the end of the program is reached.
- **Output Results:**
    - **Print Registers:** Implements `print_registers` to display the contents of all registers.
    - **Print Memory:** Implements `print_memory` to display memory contents, including labels.

3.2.8 Main Function Execution

- Instantiates the `MipsSimulator`.
- Loads the assembly program from a file (e.g., `"test_3_final.asm"`).
- Assembles the program into binary instructions.
- Runs the simulator.
- Prints the final state of registers and memory.

3.3 Conclusion

The Rust code successfully implements a MIPS simulator that can:

- Load and parse MIPS assembly code.
- Assemble instructions into binary format.
- Simulate the execution of MIPS instructions across all pipeline stages.
- Handle various instruction types, including arithmetic, logical, memory access, and control flow instructions.
- Maintain and display the state of registers and memory after execution.

The simulator follows the standard MIPS instruction pipeline, effectively simulating the datapath and control signals necessary for instruction execution.

3.4 Usage Example

Listing 3.1: Main Function Execution

```
fn main() {
    let mut simulator = MipsSimulator::new();

    simulator.load_program_from_file("test_3_final.asm");

    simulator.assemble_program();

    simulator.run();

    simulator.print_registers();
    simulator.print_memory();
}
```

Note: Replace `"test_3_final.asm"` with the path to your MIPS assembly file.

3.5 Key Implementation Highlights

- **Instruction Handling:**
  - Supports R-type, I-type, and J-type instructions.
  - Properly encodes and decodes instructions based on MIPS instruction formats.
  - Manages immediate values and sign extension for appropriate instructions.
- **Control Signal Simulation:**
  - Control signals are simulated through method calls and execution logic.
  - The flow of instruction execution mimics the control flow in a real MIPS processor.
- **Label Management:**
  - Labels are stored in a `labels` map for easy retrieval during assembly and execution.
  - Facilitates branch and jump address calculations.
- **Error Handling:**
  - Provides error messages for unknown opcodes, funct codes, registers, or labels.
  - Ensures robust parsing and execution by handling exceptions.
- **Program Counter (`pc`):**
  - Accurately updates the `pc` during execution.
  - Accounts for branching and jumping instructions that modify the flow of execution.
- **Register Mapping:**
  - Uses `reg_map` and `reg_map_rev` for easy conversion between register names and indices.
  - Simplifies instruction parsing and execution.
- **Memory Alignment and Addressing:**
  - Handles memory addresses correctly, considering word alignment.
  - Calculates effective addresses for `lw` and `sw` instructions.

3.6 Conclusion

By following the above steps, the simulator provides a comprehensive platform for simulating MIPS instruction execution, suitable for educational purposes and understanding the inner workings of a MIPS processor.

Chapter 4

Task 3 (MIPS Testing and Reporting)

4.1 Objective

The main aim of this task is to -

- Create 5 MIPS Program .
- The program be fairly complex which can challenge our simulator.
- Test the simulator on these files .
- Mention the results of all the programs.
- Compare and analyse the results obtained above.

4.2 Results

Below you can find our results. You can find the corresponding test cases **HERE**.

```
souvikmaji@coder24:~/Downloads/CA_Rust$ cargo run
   Compiling CA_Rust v0.1.0 (/home/souvikmaji/Downloads/CA_Rust)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.20s
     Running `target/debug/CA_Rust`
PC: 0, Instruction: lw $t0, num1
Binary: 100011 00000 01000 10000000000010000000000000000
PC: 1, Instruction: lw $t1, num2
Binary: 100011 00000 01001 10000000000010000000000000100
PC: 2, Instruction: add $t2, $t0, $t1
Binary: 000000 01000 01001 01010 00000 100000
PC: 3, Instruction: sub $t3, $t1, $t0
Binary: 000000 01001 01000 01011 00000 100010
PC: 4, Instruction: and $t4, $t0, $t1
Binary: 000000 01000 01001 01100 00000 100100
PC: 5, Instruction: or  $t5, $t0, $t1
Binary: 000000 01000 01001 01101 00000 100101
PC: 6, Instruction: slt $t6, $t0, $t1
Binary: 000000 01000 01001 01110 00000 101010
souvikmaji@coder24:~/Downloads/CA_Rust$
```

(a) Test 1

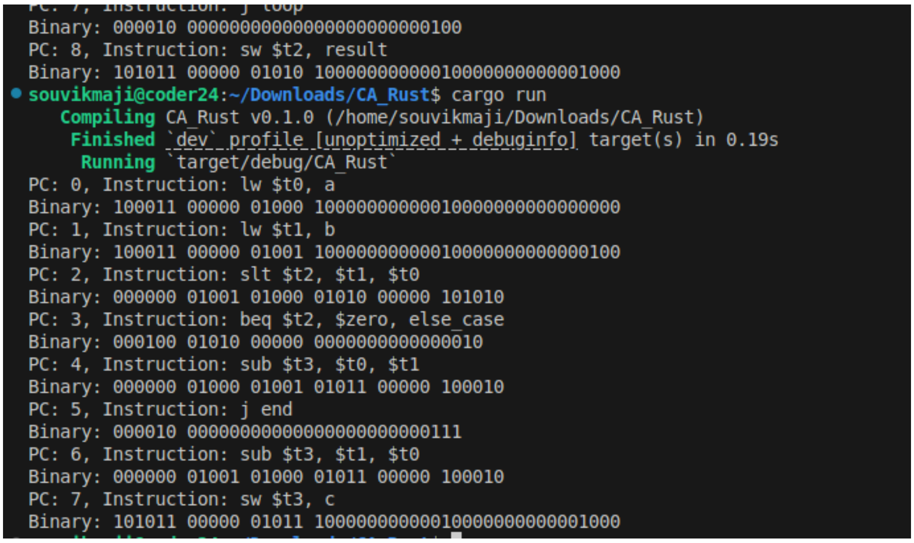
```
PC: 5, Instruction: or  $t5, $t0, $t1
Binary: 000000 01000 01001 01101 00000 100101
PC: 6, Instruction: slt $t6, $t0, $t1
Binary: 000000 01000 01001 01110 00000 101010
souvikmaji@coder24:~/Downloads/CA_Rust$ cargo run
   Compiling CA_Rust v0.1.0 (/home/souvikmaji/Downloads/CA_Rust)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.20s
     Running `target/debug/CA_Rust`
PC: 0, Instruction: lw $t0, val1
Binary: 100011 00000 01000 10000000000010000000000000000
PC: 1, Instruction: lw $t1, val2
Binary: 100011 00000 01001 10000000000010000000000000100
PC: 2, Instruction: addi $t2, $t0, 10
Binary: 001000 01000 01010 0000000000001010
PC: 3, Instruction: beq $t0, $t1, equal_case
Binary: 000100 01000 01001 0000000000000010
PC: 4, Instruction: sub $t3, $t0, $t1
Binary: 000000 01000 01001 01011 00000 100010
PC: 5, Instruction: j end
Binary: 000010 00000000000000000000000111
PC: 6, Instruction: add $t3, $t0, $t1
Binary: 000000 01000 01001 01011 00000 100000
souvikmaji@coder24:~/Downloads/CA_Rust$
0  Connect  Live Share  rust-analyzer
```

(b) Test 2

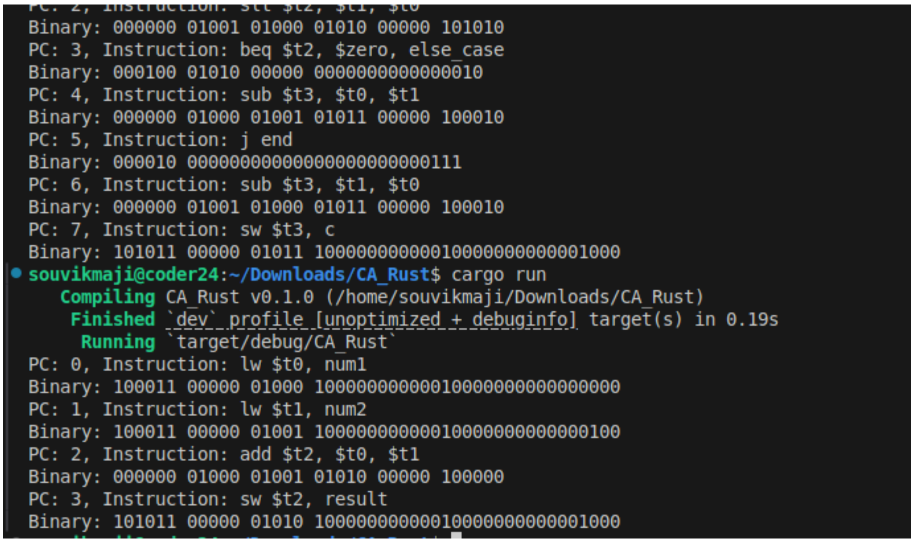
```
PC: 6, Instruction: add $t3, $t0, $t1
Binary: 000000 01000 01001 01011 00000 100000
souvikmaji@coder24:~/Downloads/CA_Rust$ cargo run
   Compiling CA_Rust v0.1.0 (/home/souvikmaji/Downloads/CA_Rust)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.18s
     Running `target/debug/CA_Rust`
PC: 0, Instruction: lw $t0, num1
Binary: 100011 00000 01000 10000000000010000000000000000
PC: 1, Instruction: lw $t1, num2
Binary: 100011 00000 01001 10000000000010000000000000100
PC: 2, Instruction: addi $t2, $zero, 0
Binary: 001000 00000 01010 0000000000000000
PC: 3, Instruction: addi $t3, $zero, 0
Binary: 001000 00000 01011 0000000000000000
PC: 4, Instruction: beq $t3, $t1, end
Binary: 000100 01011 01001 0000000000000011
PC: 5, Instruction: add $t2, $t2, $t0
Binary: 000000 01010 01000 01010 00000 100000
PC: 6, Instruction: addi $t3, $t3, 1
Binary: 001000 01011 01011 0000000000000001
PC: 7, Instruction: j loop
Binary: 000010 00000000000000000000000100
PC: 8, Instruction: sw $t2, result
Binary: 101011 00000 01010 10000000000010000000000001000
```

(c) Test 3

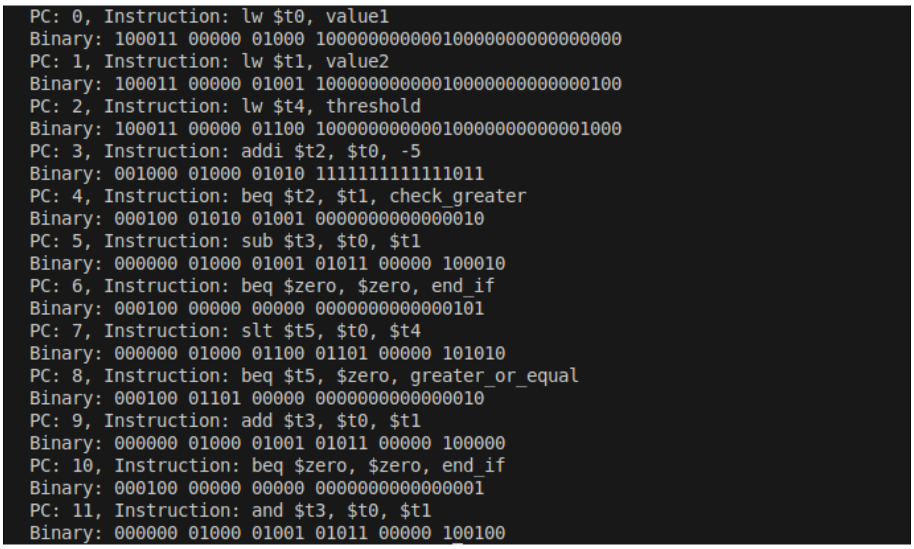
Figure 4.1: Results for Task 1



(a) Test 4

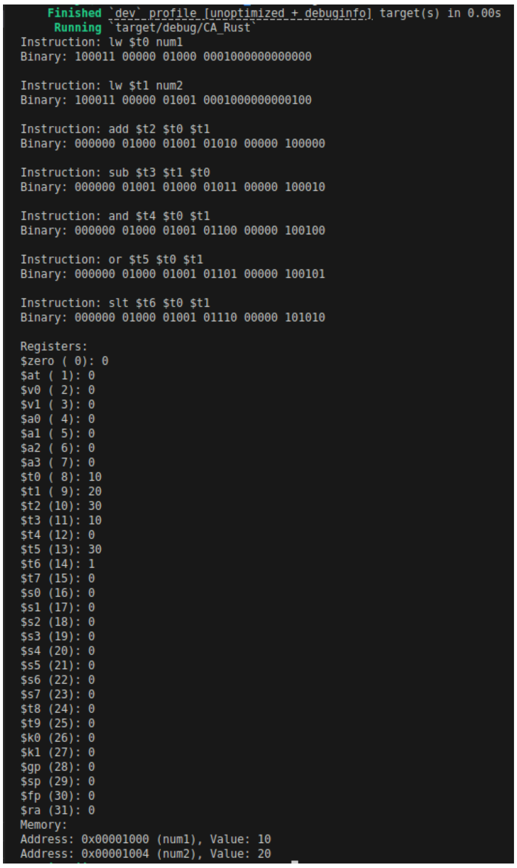


(b) Test 5

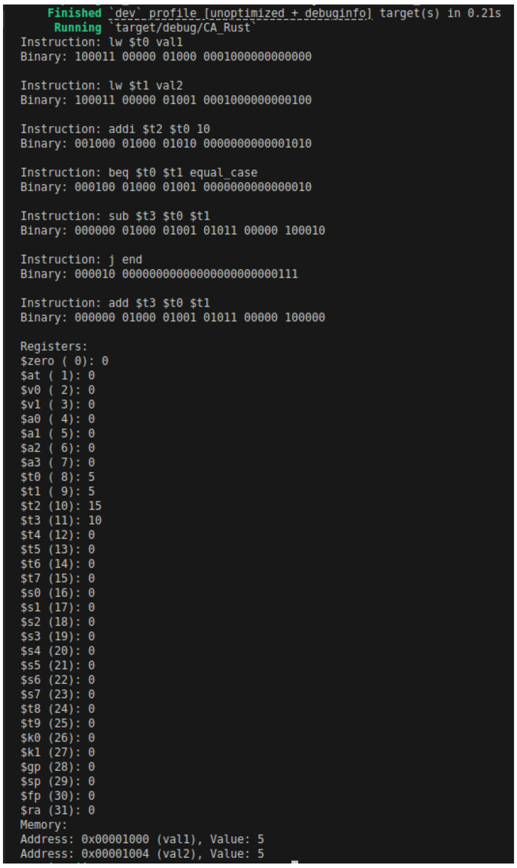


(c) Test 6

Figure 4.2: Results for Task 1

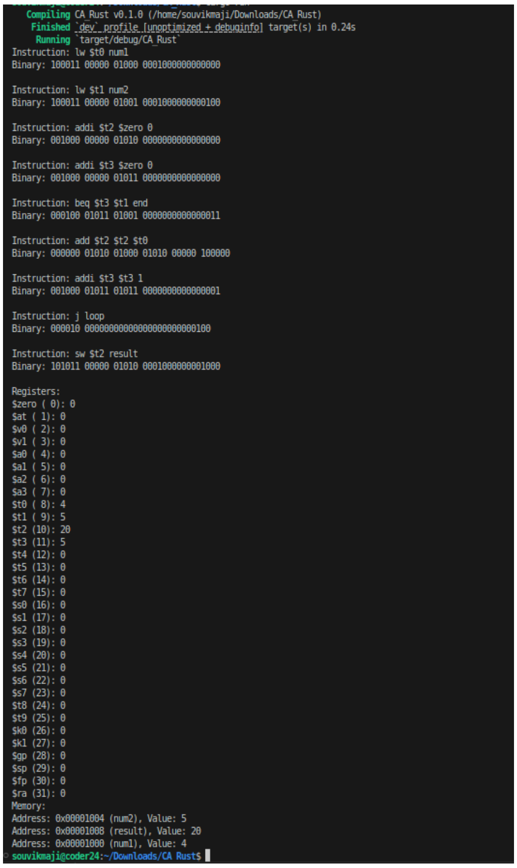


(a) Test 1

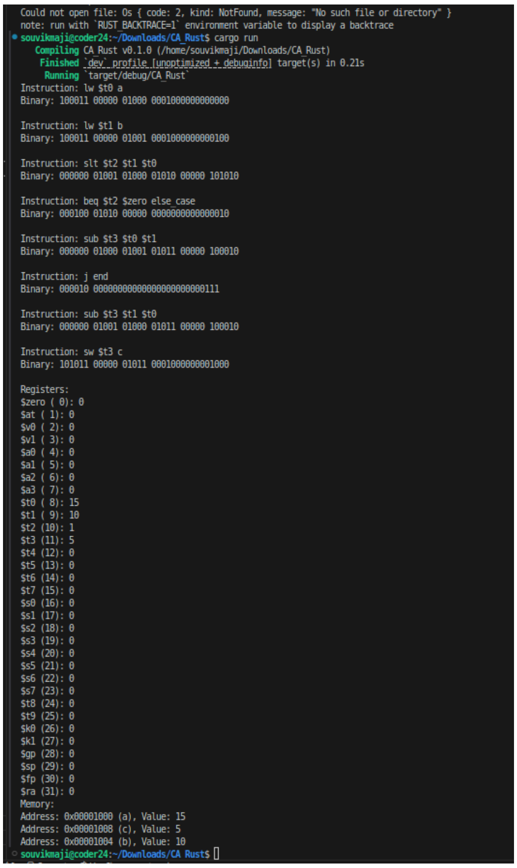


(b) Test 2

Figure 4.3: Results for Task 2

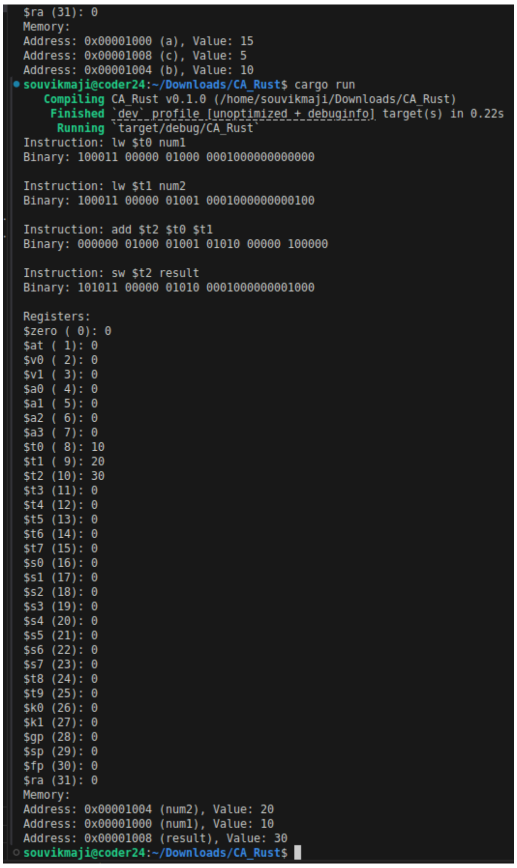


(a) Test 3

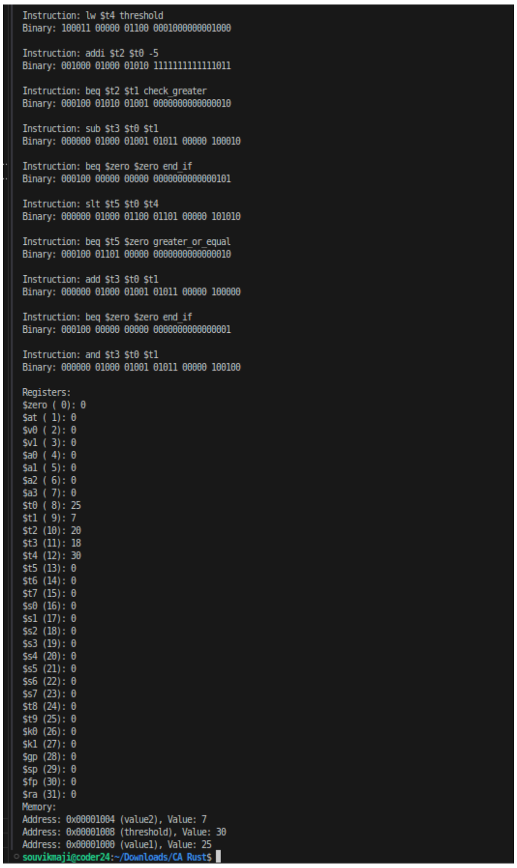


(b) Test 4

Figure 4.4: Results for Task 2



(a) Test 5



(b) Test 6

Figure 4.5: Results for Task 2

Chapter 5

Overview

5.1 Challenges and Solutions:

- **Parsing Instructions:** Handled complex instruction formats by breaking down and systematically converting them to binary.
- **Control Signals and Dataflow:** Simulated proper control signals by carefully following MIPS pipeline stages.
- **Branching and PC Updates:** Implemented logic to update the program counter accurately in branch instructions.
- **Memory and Registers:** Ensured correct memory and register access by modularizing the components.

5.2 Learning:

I gained a deep understanding of MIPS architecture, instruction flow, and processor simulation. Implementing this in Rust enhanced my skills in handling low-level programming with performance and safety benefits.

5.3 Individual contribution:

Member	Works
Ashutosh kumar(B22CS015)	Task 1,Task 3
Rhythm Baghel (B22CS042)	Task 1,Task 3
Souvik Maji (B22CS089)	Task 2, Task 3

5.4 Links:

- Github Link
- Reference