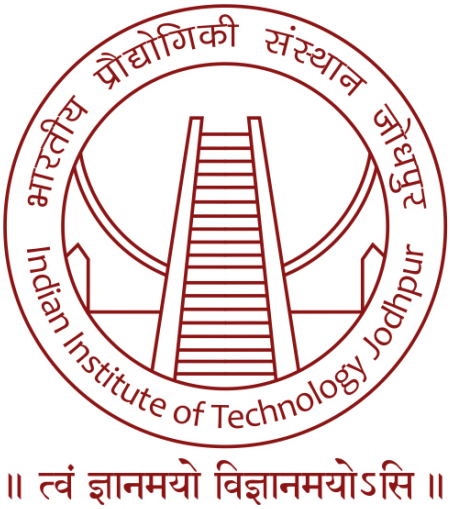


MIPS Processor Simulation

Presented by

Dhyey Findoriya (B22EE024)
Keshika Sharma (B22EE040)
Mansi Choudhary (B22EE045)

Computer Architecture

Assignment:5

October 21, 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

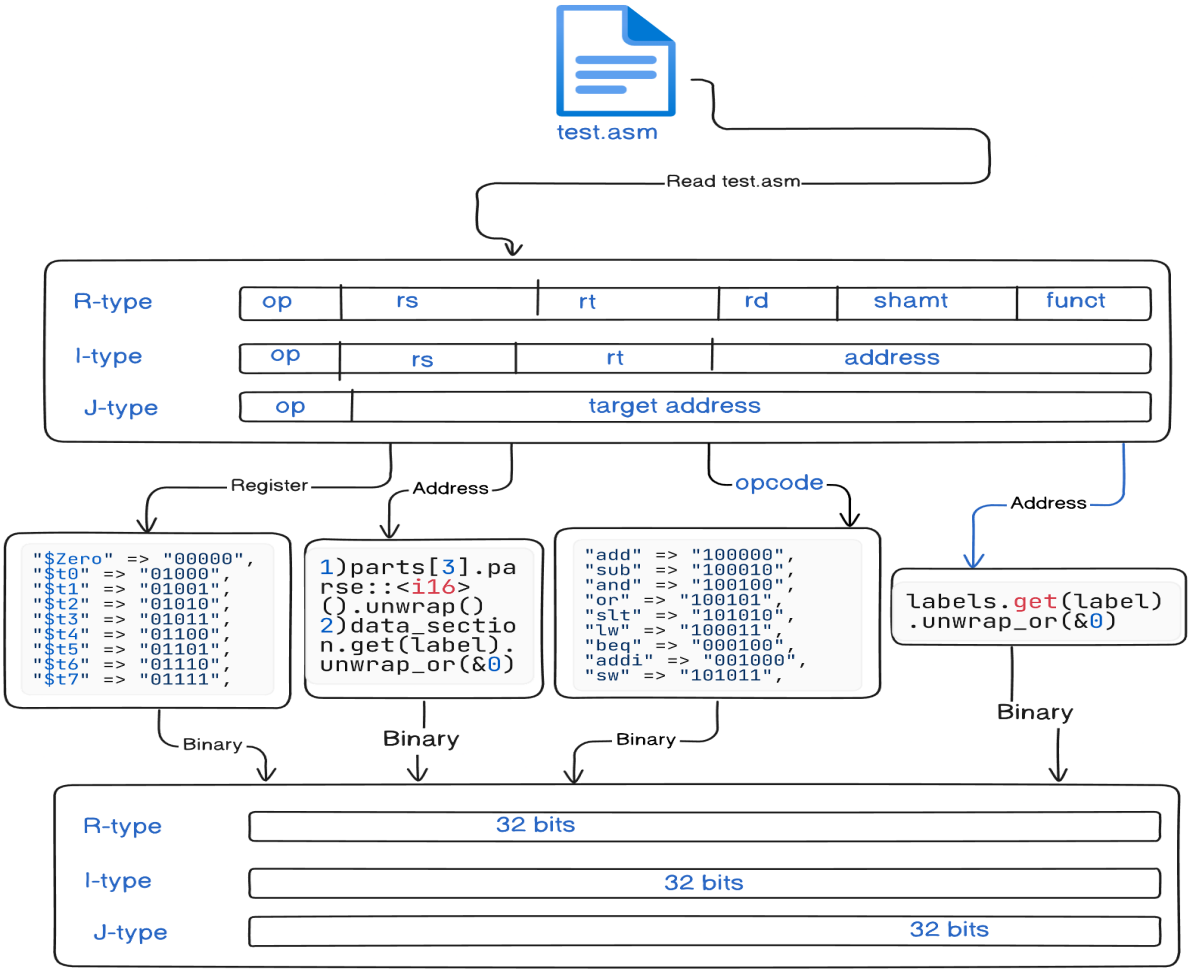
Task 1 (MIPS Compiler)

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 ()

3.1 Analysis

Documentation of the analysis or specification...

3.2 Design

3.2.1 Problem Statement

3.2.2 Requirement Analysis

3.2.3 Use cases

3.2.4 Design Diagrams

Chapter 4

Overview

4.1 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.

4.2 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.

4.3 Individual contribution:

-

Member	Works
Dhyey Findoriya(B22EE024)	Task 1 And Lab Report
Keshika Sharma (B22EE040)	Task 3 And Lab Report
Mansi Choudhary(B22EE045)	Task 2 And Lab Report

4.4 Links:

- Github Link
- Referance