COMP3005

Computer Organization

MIPS Simulator Project Report

Furkan FİDAN - 212010020002
Rabia Leyla KIYAK - 212010020010
Batuhan Orkun İNCE - 212010020015
Seyfullah ADIGÜZEL - 212010020102

1. Project Objectives

The primary objective of this project is to design and implement a simulator for a subset of the MIPS 32-bit architecture. The goals include:

- Understanding MIPS instruction formats and encoding.
- Implementing a functional instruction execution cycle.
- Developing skills in digital system simulation and GUI development.
- Gaining hands-on experience with computer architecture concepts.

2.Architecture Overview

2.1 Core Components

1. **Registers**:

32 general-purpose registers *($zero, t0−t0−t9, s0−s0−s7, etc.).*

Symbolic and numeric names for easier referencing.

Registers are implemented as an array.

2. **Memory**:

512 bytes each for instruction memory (IM) and data memory (DM).

Memory access is validated for out-of-bounds errors.

3. **Instruction Set**:

- **R-Format**: add , sub , and , or , slt , sll , srl
- **I-Format**: addi , lw , sw , beq , bne
- **J-Format**: j , jal , jr

4. **Program Counter (PC)**:

Tracks the address of the current instruction.

Updated based on the instruction type (e.g., jumps or branches).

2.2 User Interface

- The GUI is implemented using PyQt5 and features:
- An assembly code editor for inputting programs.
- Displays for registers, memory, machine code, and execution trace.
- Control buttons for running, stepping through, and resetting the program.

2.3 Execution Flow

The simulator follows a standard fetch-decode-execute cycle:

- **Fetch**: Retrieve the instruction from memory using the PC.
- **Decode**: Parse the instruction to determine the operation and operands.
- **Execute**: Perform the operation and update the state of the simulator.

3. Implementation Details

3.1 Instruction Handling

Each instruction is implemented in Python as part of the *execute_instruction* method. The simulator:

- Validates instructions for correct format and operands.
- Handles arithmetic, logical, and memory operations based on MIPS specifications.

3.2 Machine Code Generation

Instructions are converted to binary machine code using the *generate_machine_code* method. This ensures:

- Accurate representation of instructions.
- Mapping between assembly and binary formats for debugging purposes.

3.3 Error Handling

The simulator includes:

- Warnings for unsupported or incorrectly formatted instructions.
- Validation for memory accesses to prevent out-of-bounds errors.

4. Challenges Faced

1. **Instruction Decoding**:

- Ensuring correct parsing of instructions, especially with labels and offsets.

Solution: A two-pass approach to handle labels.

2. **Memory Management**:

- Implementing bounds checking for memory operations.

Solution: Validate memory addresses before access.

3. **GUI Synchronization**:

- Keeping register and memory displays in sync with the internal state.

Solution: Update displays after every execution step.

5. Performance Analysis

5.1 Execution Time

The simulator performs efficiently for small to medium programs. For larger instruction sets, performance can be optimized by reducing UI update frequency.

5.2 Memory Usage

Memory usage is minimal due to the fixed size of instruction and data memory.

6. Known Limitations

The simulator does not support floating-point instructions.

Limited to 512 bytes of memory, which restricts program size.

Error messages could be more descriptive for complex input errors.

7. Future Enhancements

Add support for floating-point and additional MIPS instructions.

Include file I/O for saving and loading assembly programs.

Enhance debugging tools with breakpoints and watch variables.

8. Conclusion

This project successfully demonstrates the functionality of a MIPS simulator. It provides a foundation for understanding computer architecture and instruction execution. The simulator can be further expanded to support more complex features and additional MIPS instructions.