RISC-V Simulator Implementation and Testing Report

Praneeth Chamarthy & Gona Sanjana

October 3, 2024

Contents

1 Introduction

This report outlines the implementation and testing approach for a RISC-V Simulator developed as part of **Computer Architecture - CS2323**. The simulator is designed to execute RISC-V machine code, simulating the behavior of a RISC-V processor by managing registers, memory, and executing instructions. The goal is to provide a reliable tool for understanding and analyzing RISC-V programs.

2 Implementation Approach

2.1 Code Structure

The implementation is organized into several key components, each encapsulated in its own class, to ensure modularity and ease of maintenance:

- **RegisterFile**: This component manages the 32 general-purpose registers and the program counter (PC). It provides methods to read from and write to registers, ensuring that the zero register (x0) remains constant, as per RISC-V specifications.
- **Memory**: This class simulates the system's memory, allowing for reading and writing of data at specified addresses. It includes methods for handling different data sizes (8, 16, 32, and 64 bits), ensuring accurate data manipulation.
- **Instruction**: Represents the various instructions that the simulator can execute. Each instruction type (e.g., ADD, SUB, LW) is implemented as a subclass, providing specific execution logic tailored to the instruction's semantics.
- **Simulator**: The core component that orchestrates the execution of instructions. It manages the program counter, handles input commands, and maintains the call stack and breakpoints, providing a user interface for interacting with the simulator.

2.2 Key Implementation Details

2.2.1 Instruction Decoding

The `Instruction` class includes a `decode` method that interprets machine code into specific instruction objects. This method uses the opcode and function fields to determine the correct instruction type, ensuring accurate execution of the RISC-V instruction set. The decoding process is crucial for translating binary machine code into executable instructions that the simulator can process.

2.2.2 Execution Logic

Each instruction subclass implements an `execute` method, which modifies the state of the `RegisterFile` and `Memory` based on the instruction semantics. This modular approach allows for easy extension and modification of the instruction set. The execution logic ensures that each instruction performs its intended operation, such as arithmetic calculations or memory accesses.

2.2.3 Control Flow

The simulator supports control flow instructions such as branches (e.g., BEQ, BNE) and jumps (e.g., JAL, JALR). These instructions update the program counter to alter the execution sequence, enabling complex program logic. Proper handling of control flow is

essential for executing loops, conditional statements, and function calls within RISC-V programs.

2.2.4 Data Handling

The `Memory` class provides methods to read and write data of various sizes, ensuring that memory accesses are within valid bounds. This prevents errors such as buffer overflows and ensures data integrity. The memory management system is designed to handle different data types and sizes, reflecting the flexibility of the RISC-V architecture.

3 Testing

To ensure the correctness of the implementation, a comprehensive testing strategy was employed, combining automated and manual testing techniques:

3.1 Unit Testing

Unit testing was conducted to verify the functionality of individual components such as the `RegisterFile`, `Memory`, and `Instruction` classes. Each component was tested independently to ensure that it performs its intended operations correctly. For example, tests were written to check the correct reading and writing of registers, proper memory access, and accurate execution of individual instructions. This approach helps isolate issues within specific components, making debugging more efficient.

3.2 Integration Testing

Integration testing involved testing the simulator as a whole by running a set of predefined machine code programs. These programs were designed to cover a wide range of instructions and scenarios, including arithmetic operations, memory accesses, and control flow changes. The goal was to ensure that all components work together seamlessly and that the simulator can handle complex instruction sequences without errors.

3.3 Boundary Testing

Boundary testing focused on edge cases, such as memory boundary conditions and register overflow scenarios. Tests were designed to push the limits of the simulator's capabilities, ensuring robust handling of exceptional cases. For instance, attempts to access memory beyond its allocated range or perform arithmetic operations that exceed register limits were tested to verify that the simulator handles these situations gracefully.

3.4 Debugging and Logging

The simulator includes extensive logging capabilities to trace the execution of instructions and changes in the program counter. Logging statements were used to capture detailed information about the execution flow, which facilitated debugging and verification. By analyzing log outputs, discrepancies in expected behavior could be quickly identified and addressed, improving the overall reliability of the simulator.

3.5 Manual Verification

Manual verification involved comparing the output of the simulator against expected results for a set of test programs. This process helped identify discrepancies and validate

the correctness of the implementation. By manually reviewing the simulator's output, subtle errors that automated tests might miss could be detected and corrected.

3.6 Comparison with Ripes

The simulator's output was compared with Ripes, a graphical RISC-V simulator, to ensure accuracy. Ripes provides a visual representation of the execution process, allowing for a side-by-side comparison with the simulator's output. This comparison helped identify any deviations from expected results and provided additional confidence in the simulator's correctness.

3.7 Output Comparison with `diff`

The `diff` command was used to automate the comparison of expected output with the actual output generated by the simulator. By highlighting differences between the two outputs, `diff` facilitated quick identification of errors and ensured that the simulator's behavior matched the expected results. This automated approach complemented manual verification efforts, providing a comprehensive validation strategy.

4 Challenges and Solutions

We had trouble implementing the show stack command because we had to encode and decode the instructions from the input.s file. This made it impossible for us to see the function names in the console while the show stack command was running. My implementation strategy said that it was challenging for me to locate the function name or label. Since we are encoding the instructions using assembler and must manage the data and text parts in both assembler and simulator, we also encountered challenges while handling the loading of the data section.

5 Results and Conclusion

By following this structured approach to implementation and testing, the simulator was developed to accurately model the behavior of a RISC-V-like architecture, providing a reliable tool for executing and analyzing machine code programs. This comprehensive testing strategy ensures that the simulator is robust, accurate, and ready for educational and analytical use.