RISC-V Assembler Implementation and Testing Report

Praneeth Chamarthy && Gona Sanjana

September 8, 2024

Contents

1 Introduction

This report outlines the implementation and testing approach for a RISC-V assembler developed as part of **Computer Architecture - CS2323**.

2 Implementation Approach

2.1 Code Structure

The assembler is implemented in C and consists of the following main components:

- `main.c`: Contains the main program logic, file I/O, and orchestrates the two-pass assembly process.
- `parser.c` and `parser.h`: Implement parsing functions for different RISC-V instruction types and handle instruction encoding.
- `utils.c` and `utils.h`: Provide utility functions like register number lookup, label parsing, and string manipulation.
- `assembler.h`: Defines common structures, constants, and function prototypes used across the project.

2.2 Key Implementation Details

2.2.1 Two-Pass Assembly

The assembler uses a two-pass approach:

1. First Pass: Scans the input file to collect all labels and their corresponding addresses.
2. Second Pass: Processes each instruction, resolving labels to addresses and generating machine code.

2.2.2 Instruction Parsing and Encoding

- Separate parsing functions for each instruction type (R-type, I-type, S-type, B-type, U-type, J-type).
- Utilizes bitwise operations to construct the 32-bit machine code for each instruction.
- Handles immediate value parsing and encoding, including sign extension where necessary.

2.2.3 Label Handling

- Labels are stored in a symbol table during the first pass.
- The symbol table is implemented as a linked list of label structures.
- During the second pass, label references are resolved to their corresponding addresses.

2.2.4 Pseudo-instruction Support

- Implements expansion of pseudo-instructions like `mv`, `j`, etc., into their corresponding base instructions.

2.2.5 Error Handling and Reporting

- Implements robust error checking for syntax errors, invalid instructions, and out-of-range immediates.
- Provides informative error messages with line numbers for easier debugging of assembly code.
- The error will be printed to console as well as the output.hex file for more clarity and flexiblity about the errors in the input.s file

2.2.6 Modular Design

- Functions are organized into logical modules (parsing, utilities, main program logic) for better code organization and potential reusability.
- Utilizes header files to define interfaces between different parts of the program.

2.2.7 RISC-V Instruction Set Coverage

- Supports the core RISC-V RV64I instruction set.
- Includes support for common extensions like multiply/divide instructions (RV32M).

3 Testing Approach

To ensure the correctness and reliability of our RISC-V assembler, we implemented a comprehensive testing strategy that includes unit testing, integration testing, and comparison with established tools.

3.1 Unit Testing

We conducted unit tests for individual functions in `parser.c` and `utils.c`. For example:

- Testing `get_register_number()` in `utils.c` with various register names to ensure correct register number mapping.
- Testing individual instruction parsing functions (e.g., `parse_r_type()`, `parse_i_type()`, etc.) with known inputs and expected outputs.

3.2 Integration Testing

For integration testing, we created a set of small RISC-V assembly programs that use different instruction types and pseudo-instructions. These programs were assembled using our assembler, and the output machine code was verified through a bash script which compares two given files using the `diff` command

3.3 Comparison with RIPES

To further verify the correctness of our assembler, we used RIPES Our testing process involved:

1. Writing various RISC-V assembly programs covering different instruction types and edge cases.
2. Assembling these programs using both our assembler and RIPES.
3. Comparing the machine code output manually from our assembler with that of RIPES.

3.4 Edge Cases

We also tested our assembler with various edge cases, including:

- Large immediate values to ensure proper handling and error reporting.
- Forward and backward label references.
- Invalid instructions or syntax to test error handling.

3.5 Test Cases

We created a comprehensive set of test cases, organized in a separate `tests` directory:

Each test case includes an input .s file with RISC-V assembly code and an expected output .o file with the correct machine code.

4 Challenges and Solutions

Implementing Jump and Branch instructions, as well as their subsequent pseudo instructions, has presented challenges, primarily in calculating the offset that corresponds to the branch address. Eventually, I was able to resolve this offset issue, but it felt difficult when I began implementing the Li pseudo instruction, which combines the `addi` and `addiw` commands.I never understood how to know when to use `addi` and `addiw`.

5 Results and Conclusion

I encountered numerous challenges, such as segmentation faults and code crashes, when I first began testing my code based on error handling.I changed my code and used the idea of a core file to solve the problem in order to get out of this situation.
Ultimately, I found it challenging to implement the LI pseudo instruction with the assistance of `addi` and `addiw`, but I will attempt to do so in the future.