Project Report: RISC-V Assembler and Simulator

1. Abstract

This project involves the development of a **RISC-V Assembler and Simulator** using C++. The primary objective is to bridge the gap between human-readable assembly language and machine-readable binary code. The system accepts RISC-V assembly code (RV32I Base Integer Instruction Set) as input, processes it through a two-pass assembly process, and outputs the corresponding 32-bit hexadecimal machine code. Additionally, the project includes simulation capabilities to model the execution of these instructions on a virtualized RISC-V architecture.

2. Introduction

RISC-V is an open-standard Instruction Set Architecture (ISA) based on established Reduced Instruction Set Computer (RISC) principles. Unlike legacy ISAs, RISC-V is modular and extensible. To understand low-level computing and computer architecture, building an assembler is a fundamental exercise.

This tool performs two critical functions:

1. **Translation:** Converting mnemonics (e.g., add, lw) and operands into binary/hexadecimal instruction words.
2. **Simulation:** (As per design goals) Modeling the behavior of registers and memory to execute the generated code.

3. System Architecture

The system is built using C++ and utilizes the Standard Template Library (STL) for efficient data management. The architecture is defined by the following components:

3.1 Data Structures

- **Register Map (registers):** Stores the 5-bit binary representation of the 32 general-purpose registers (x0 to x31).
- **Alias Map (register_alias):** Maps Application Binary Interface (ABI) names (e.g., sp, ra, a0) to their corresponding integer register indices, allowing for user-friendly assembly coding.
- **Instruction Map:** A lookup table storing opcodes, funct3, and funct7 codes for supported instructions, categorized by format (R, I, S, B, J, U).
- **Label Map:** A symbol table used to resolve memory addresses for branching and

jumping.

3.2 The Assembly Process (Two-Pass Algorithm)

1. **Pass One (Symbol Resolution):**
   - The assembler scans the input.txt file.
   - It identifies labels (strings ending with :) and records their line numbers in the labels map.
   - This step is crucial for calculating relative offsets for Branch (beq, bne) and Jump (jal) instructions where the target label might appear later in the code.
2. **Pass Two (Translation):**
   - The assembler re-reads the input line by line.
   - It tokenizes the string to separate the mnemonic, registers, and immediates.
   - Based on the mnemonic, the appropriate format handler is invoked.

4. Implementation Details

The core logic is divided based on the RISC-V instruction formats. The C++ implementation utilizes bitset for binary manipulation and stringstream for parsing.

4.1 Instruction Formats Supported

The assembler handles the standard RV32I formats:

- **R-Format (Register-Register):**
  - *Instructions:* add, sub, xor, or, and, sll, srl, sra, slt, sltu.
  - *Logic:* Constructs the machine code using funct7 | rs2 | rs1 | funct3 | rd | opcode.
- **I-Format (Immediate & Loads):**
  - *Instructions:* addi, lw, lb, jalr, etc.
  - *Logic:* Handles 12-bit signed immediates. It specifically parses memory addressing syntax like offset(base) (e.g., 4(sp)).
- **S-Format (Store):**
  - *Instructions:* sw, sh, sb.
  - *Logic:* Splits the 12-bit immediate into two parts (imm[11:5] and imm[4:0]) to fit the split-immediate architecture of RISC-V stores.
- **B-Format (Branch):**
  - *Instructions:* beq, bne, blt, etc.
  - *Logic:* Calculates the **PC-relative offset**.
  - *Formula:* Jump Offset = (Label_Line - Current_Line) * 4.
  - Checks that the offset fits within the 13-bit range and is 2-byte aligned.
- **J-Format (Jump):**
  - *Instructions:* jal.
  - *Logic:* Handles 20-bit jumps with bit shuffling to match the specific J-immediate encoding structure of RISC-V.
- **U-Format (Upper Immediate):**

    - *Instructions:* lui, auipc.
    - *Logic:* Processes 20-bit upper immediate values.

4.2 Error Handling and Validation

The system implements robust error checking to ensure valid machine code generation:

- **Syntax Validation:** Checks for correct comma placement and token counts.
- **Register Validation:** Ensures registers are within x0-x31 or valid ABI names.
- **Immediate Bounds:** Verifies that immediate values fit within their respective bit-widths (e.g., -2048 to 2047 for I-Type).
- **Label Existence:** Flags errors if a branch targets a non-existent label.
- **Alignment:** Ensures branch targets are aligned to instruction boundaries.

5. Simulation & Data Path

While the code provided focuses on the Assembler, the project design includes a simulation aspect. The simulator models the processor state:

- **Program Counter (PC):** Tracks execution flow.
- **Register File:** An array representing the 32 registers.
- **Memory:** A byte-addressable array to simulate RAM.
- **Execution Loop:** Fetch -> Decode -> Execute -> Memory Access -> Write Back.

6. Results and Testing

6.1 Test Case: Arithmetic

Input (Assembly):

```
addi t0, x0, 5   ; Load 5 into t0
addi t1, x0, 10  ; Load 10 into t1
add t2, t0, t1   ; t2 = t0 + t1
```

Output (Hex Machine Code):
The assembler successfully parses these lines, resolves t0 to x5, t1 to x6, etc., and outputs the correct 32-bit hex strings.

6.2 Test Case: Branching and Labels

Input:

```
    beq x0, x0, target
    addi x1, x1, 1
target:
    sub x1, x1, x2
```

Result:
The populate_lables function correctly identifies target at line 3. The bformat function calculates the offset of 8 bytes (2 instructions) and encodes it into the B-type format.

7. Conclusion

This project successfully implements a functional RISC-V Assembler capable of parsing complex assembly syntax, resolving labels, and generating accurate machine code. The modular design allows for easy extension to include additional instructions (such as the M-extension for multiplication). The rigorous error handling ensures that invalid assembly code is caught before machine code generation, making it a robust tool for learning and development.