Report on RISC-V Simulator Implementation

Project Overview

The RISC-V Instruction Simulator is a project aimed at creating a simulated environment for running RISC-V assembly code. The simulator reads RISC-V assembly files, interprets various instruction formats (R, I, S, B, U, and J), and processes them while simulating cache memory behavior.

The project is organized into several source and header files that are stored in `source` and `header` respectively.

Project Structure

- Instruction Formats: Each format (R, I, S, B, U, and J) is implemented in separate C++ files and headers. This modular approach makes it easy to extend or modify individual instruction handling.
- Cache Simulation: The cache simulation is implemented in cache.cpp and its header file cache.hh. It reads configuration details from config.txt.
- Main Simulator: The main logic is located in lab7.cpp, which integrates all formats and cache functionality to produce simulation output.

Implementation Approach

1. Instruction format handling

The project is structured to handle multiple RISC-V instruction types. Each instruction type (R, I, S, B, U, J) is processed using dedicated functions that implement the encoding and decoding logic based on the RISC-V instruction set architecture (ISA). The following types of instructions are supported:

- R-type Instructions: For register-to-register operations (e.g., `add`, `sub`).
- I-type Instructions: For operations involving immediate values (e.g., `addi`, `lw`).
- S-type Instructions: For store instructions, such as `sw` (store word).
- B-type Instructions: For conditional branch instructions like `beq` and `bne`.
- U-type Instructions: For upper immediate instructions like `lui` and `auipc`.

- J-type Instructions: For jump instructions, such as `jal`.

  Each of these formats is handled in separate functions or classes to maintain modularity and ease of debugging.

2. Label and Instruction Separation:

The simulator allows flexibility with how labels and instructions are defined in the assembly code. Labels can appear on a different line from the instruction they reference, which helps in handling multi-line or spaced-out assembly code more effectively.

For example:

```
label l1:
    add x0, x0, x0
```

or

```
label l1: addi sp, sp, -16
```

Both cases are allowed and correctly parsed.

3. Data Type and Value Separation

In addition to labels, data declarations (such as `.word`, `.byte`) and their associated values can also be split across lines. This enhances the flexibility of the assembler, allowing the following formats:

```
.word
  10
```

or

```
.byte 0xFF
```

Both cases are handled without issue, and data is properly allocated in the simulated memory.

4. Infinite Loop Detection

One of the key features is infinite loop detection. The simulator monitors program execution and detects situations where control flow gets stuck in a loop without exit conditions. If an infinite loop is detected, the simulator halts execution and displays a warning message, ensuring that the simulation does not run indefinitely.

5. Handling Pseudo-Instructions

The simulator can handle RISC-V pseudo-instructions, which are simplified representations of more complex instructions. For example, a pseudo-

instruction like `j label` (jump to offset) is internally converted to `jal x0, label`.

6. Comment Ignoring

The simulator can ignore comments in the assembly code. Any text following a `;` symbol is treated as a comment and ignored during parsing. This allows for better readability of assembly programs without affecting functionality.

For example:

```
add x1, x2, x3 ;this is a comment

;ld x2, 0(x3) this is a commented instruction
```

7. Cache Simulation

Cache parameters, including block size and associativity, are configurable via config.txt. This component models cache operations (e.g., hits, misses) as the instructions are processed.

Testing Approach

1. Testing for Instruction Types

Every instruction type was tested using a combination of all instructions to ensure proper encoding and execution.

2. Label and Data Handling

Test programs were written where labels and instructions were on separate lines to verify proper resolution. Similarly, data type declarations were tested to ensure that values on different lines were correctly parsed and stored.

3. Whitespace and Comment Handling

Several test cases were created with varying amounts of leading/trailing white spaces and inline comments. The simulator was able to correctly parse the instructions while ignoring irrelevant spaces and comments.

4. Infinite Loop Detection Testing

Programs containing intentional infinite loops were tested to verify that the simulator correctly detected and halted execution. For example:

```
loop:
  j loop
```

The simulator successfully identified this as an infinite loop and stopped further execution.

5. Pseudo-Instruction Testing

Pseudo-instructions like `mv`, `neg`, and `bnez` were tested to ensure they were correctly translated and executed. For example, mv was tested to ensure it properly copied the value from one register to other.

6. Cache Testing

Light tests were done using the code provided in Homework-4 to verify the functionality of the simulator. Cache statistics were recorded and compared with the output from Ripes to ensure accuracy and consistency.