LAB5: Pipeline stall detection

Kartikeya Mandapati
CS22BTECH11032

November 6, 2023

1 Introduction

This report presents a program that detects the pipeline stalls of the given RISC-V assembly code. The code inserts stalls for two cases, without data forwarding and with data forwarding without hazard detection. The report outlines the program's design, implementation and practical usage.

2 Code Overview

The stall detector implemented in C++ and follows a modular structure. It can handle various types of RISC-V instructions. Both cases have individual functions for stall detection. Here's an overview of the code structure:

2.1 Main Program

The main program reads assembly-level instructions from an input file, and stores them in a vector of strings. It then iterates through each of the instruction and passes them through decodeInstruction function which decodes and calls the hazard detection function. They are iterated twice, once without data forwarding and the second time with data forwarding. The input file is set as "input.txt" in the main.

2.2 Instruction decoding

The **decodeInstruction** function takes in the instruction in the form of a string as input. It then breaks down the instruction into tokens. The first token holds the type of instruction. Based on the typr of instruction, using **isALuType** and **isLoadType** I found the other fields. As the input instructions could have both types of register conventions, I used the **getRegister** function to convert them into x{i} form, using which I can do direct string comaprision. With these as input fields it calls **checkHazard** and **checkWithForwarding** which insert nops. Based on whether a hazard is detected or not, I update the rd1,rd2,insType1 and insType2 which are the variables that store the fields of previous instructions, which is required for stall detection.
The code flow is as follows:

1. Reading instructions from file.
2. Iterating through each of them and calling decodeInstruction.
3. Tokenizing and identifying the fields and calling the stall insertion func.
4. Inserting stalls and updating the variables.
5. Repeating the same again with different stall insertion function.

2.3 Other Functions

I have made other functions for converting **isALuType** and **isLoadType** to identify the instruction type and **getRegister** to convert the register format as having separate function is a better practice.

2.4 Stall detection

The **checkHazard** and **checkWithForwarding** use the variables **rd1,rd2** which store the destination registers of previous and the one before previous. After each instruction they get updated to the next one. Based on the instruction type, it is check if rs1 or rs2 is same as the rd of the previous two instructions and nops are inserted. If the hazard is with the previous one, we need not check with the one brfore prev as the nops inserted would handle it. In the case with forwarding, only load type instructions create a max of one stall and ALU type don't create any stalls.

3 Usage

The program's usage is well-documented and provided in the README file. To use the disassembler program:

1. Prepare an input file containing assembly-level instructions.
2. Compile the C++ program.
3. Run the program, providing the input file name changed to that of yours.
4. The program will insert stalls display the resulting code along with total number of cycles.

4 Results

The stall detector program successfully inserts stalls in the given RISC-V assembly instructions. Below is a sample test:

Listing 1: Sample Input

```
addi x10, x10, 6
addi sp, sp, -16
sd x1, 8(sp)
sd x10, 0(sp)
addi x10, x10, -1
addi x10, x0, 1
addi sp, sp, 4
ld x5, 0(sp)
ld x1, 8(sp)
addi sp, sp, 16
addi x0, x0, 1
```

Listing 2: Sample output

```
Without data forwarding:
addi x10, x10, 6
addi sp, sp, -16
nop
nop
sd x1, 8(sp)
sd x10, 0(sp)
addi x10, x10, -1
addi x10, x0, 1
addi sp, sp, 4
nop
nop
ld x5, 0(sp)
ld x1, 8(sp)
addi sp, sp, 16
addi x0, x0, 1
Total: 19 cycles
```

```
With data forwarding and no hazard detection:
addi x10, x10, 6
addi sp, sp, -16
sd x1, 8(sp)
sd x10, 0(sp)
addi x10, x10, -1
addi x10, x0, 1
addi sp, sp, 4
ld x5, 0(sp)
ld x1, 8(sp)
addi sp, sp, 16
addi x0, x0, 1
Total: 15 cycles
```

5 Correctness

The correctness of the code has been verified both logically and also by testing with various inputs of over 15 test cases. I have verified with the output of RIPES simulatior and also tested edge cases and each instruction independently.

6 Conclusion

The code is easy to use and efficient. The test cases verified are attached in the zip file.