CS2323-Computer Architecture. Autumn 2023

Lab-5
Pipeline Stall Detector/Simulator

Introduction

The C++ program aims to analyse a given assembly program and indicate where pipeline stalls occur, as well as output a modified assembly program with added NOPs [No-Operation] to make it stall-free. The program considers 2 cases for adding nops:

1. No Data Forwarding and No Hazard Detection are implemented.
2. Data Forwarding and Hazard Detection are implemented.

The program also calculates and outputs the total number of cycles the given assembly program will take, considering a standard 5-stage pipeline.

Note: nop= add x0, x0, x0

Code Analysis:

1. Input Handling, Output and Assumptions:

The program assumes that the input assembly program is syntactically correct, free of syntax errors, and follows specific formatting rules.

It assumes only one space between the instruction and the first operand and one space between the comma and the second operand. Blank lines and labels are not present in the input program, simplifying the parsing of the assembly code.

The code takes input from **input.txt,** the block of code and gives out an output file, **output.txt,** after inserting NOPs at appropriate places and calculating the total clock cycles. It also prints the output on the terminal.

2. Register Alias Mapping:

The function **ChangeRegisterAlias()** is defined that maps register aliases (e.g., a0, t0, s0) to their standard RISC-V register names (x0, x5, x8).

This mapping ensures consistency in register naming, making it easier to detect dependencies between instructions.

3. Pipeline Stalls and Nop Insertion:

The program identifies Data Hazards (RAW type) in the code and handles pipeline stalls by analysing dependencies between instructions. A data hazard of RAW(Read after write) type occurs when the value of a register is read before its value is written from the previous instruction.

It categorises the stalls into two cases:

- **No Data Forwarding and No Hazard Detection:** In this case, the program inserts nops to resolve dependencies between instructions, assuming no data forwarding or hazard detection. Nops are inserted for a stall, ensuring a stall-free execution. The function **NoForwarding()** is used for this purpose.
- **Data Forwarding and Hazard Detection:** In this case, the program inserts nops to account for dependencies, but it may skip some nops due to data forwarding and hazard detection mechanisms. The function **Forwarding()** is used for this purpose.

4. Number of clock cycles:

The functions **NoForwarding()** and **Forwarding()** also calculates the total number of lines(n) in the block of code after adding nops, and the total number of clock cycles(C) is calculated by adding 4 to it [C = T + n - 1, where T is 5 for 5-stage pipeline].

5. Main Function:

It reads the input assembly program from a file named "**input.txt**" and stores it in an array of strings. It also parses the registers used in each instruction.

It applies the register alias mapping using the **ChangeRegisterAlias()** function to ensure uniform naming.

6. Handling Load and Store Instructions:

The program correctly identifies and processes load and store instructions, handling dependencies and nops accordingly and ignoring load dependence on store instructions.

7. Testing and Verification:

The program provides two modes of operation: one with no data forwarding and no hazard detection and another with data forwarding and hazard detection.

It correctly handles dependencies and inserts nops as needed in both modes.

The program ensures that inserted nops do not disrupt the order of instructions in the program. The program has been tested on a variety of test cases, and some of them are listed in "**All_Input_Cases.txt.**"

8. Precautions:

The following precautions are to be kept in mind for successful execution of the program:

- If the number of lines in the block of code is more than 25, then increase the size at the beginning of the code to accommodate all the lines.
- Ensure that the "**input.txt**" does not contain blank lines, especially at the end of the file, to ensure smooth execution and no exceptions.
- The "**input.txt"** must contain one block of code for successful output generation.

9. Remarks:

While parsing the strings to get the registers, the names rd, rs1, and rs2 may not be conventionally correct but are still taken to be in that way to simplify the code and make it more readable and free from unnecessary clutter.

Conclusion:

The C++ program is a robust tool for analysing assembly programs and mitigating pipeline stalls. It handles dependencies, efficiently inserts nops, and provides valuable insights into executing instructions in a standard 5-stage pipeline. It considers scenarios with and without data forwarding and hazard detection and reports the number of clock cycles to complete the input block of code for either method. Its output is clear and informative, enhancing the understanding of pipeline execution.