Report

Name: Saksham
Roll No. : AI22BTECH11024

Introduction

This report details the design and implementation of a C program developed to analyse assembly programs and insert NOP (No-Operation) instructions to eliminate pipeline stalls. The program also calculates the total number of cycles required for program execution in a 5-stage pipeline architecture.

The program has been developed in accordance with the given requirements and is based on the logic that revolves around detecting data hazards that lead to pipeline stalls and inserting the necessary NOP instructions to resolve them. The program gives the output for both data forwarding and without data forwarding.

Code Structure and Logic:

Data Hazard Detection and NOP Insertion

The core logic of the code focuses on identifying dependencies between instructions, regardless of their types(load and store required special care), and mitigating pipeline stalls by inserting NOP (No-Operation) instructions. Here is a comprehensive explanation of the logic:

Data Hazard Detection:

1. The code identifies dependencies by comparing the destination register of the current instruction with the source registers of the preceding instructions.
 - If the destination register of the current instruction matches the source register of a previous instruction, a data hazard is detected.
 - Furthermore, the code checks if the destination register of the current instruction matches the destination register of the previous store instruction, which may lead to a hazard.

2. Data hazards are detected for all instructions, and the code keeps track of which instruction is affected by a hazard and how many NOPs need to be inserted for each instruction.

3. Special care was taken for store type instructions.

NOP Insertion:

1. When a data hazard is identified for an instruction, NOP instructions are inserted into the pipeline. The number of NOPs inserted is according to whether data forwarding is being implemented or not:

- In the absence of data forwarding (flag = 0), two NOPs are inserted to manage the hazard. These NOPs represent the fetch and decode stages in the pipeline.

- In the case of data forwarding (flag = 1), only one NOP is inserted, as data forwarding can resolve hazards more efficiently.

Additional Considerations:

- The code accommodates scenarios in which register aliases are used (e.g., "s0" instead of "x8") by converting these aliases to their architectural register names. This ensures accurate data hazard detection.

- The total number of cycles is calculated by considering the number of instructions and the inserted NOPs. Each NOP contributes one cycle.

- The code properly addresses edge cases where instructions depend on multiple preceding instructions.

Through this logic, the code effectively identifies data hazards and inserts the necessary NOPs to eliminate stalls, ensuring the efficient execution of the assembly program in a 5-stage pipeline architecture. The user's choice of data forwarding or non-data forwarding mode determines the number of NOPs inserted.

Testing and Validation:

The code has undergone rigorous testing to ensure correctness and effectiveness. Various test cases have been employed to validate the program's behaviour, including scenarios with different combinations of instructions and their dependencies.

Furthermore, the code has been tested in both data forwarding (flag = 1) and non-data forwarding (flag = 0) modes to verify that it correctly inserts the appropriate number of NOPs based on the selected mode.

A sample_inputs.txt file is also provided containing some among the many sample inputs on which the code was tested.

Conclusion:

The C program effectively achieves the objective of identifying data hazards in assembly programs and inserting NOP instructions to eliminate pipeline stalls. This adaptability ensures that the code can handle a wide range of assembly programs while complying with the requirements specified in the assignment.

The code's logic and structure are thoroughly documented, and it follows a systematic approach to manage different instruction types and detect hazards. Extensive testing has demonstrated the accuracy and efficiency of the code in identifying and resolving data hazards.