# Report on Program for Pipeline Stall Detector/Simulator

Student Name: Chitipotu Kushwanth

Roll Number:AI22BTECH11006

Date: November 7, 2023

## Introduction

The purpose of this report is to document the development and testing of a C program designed to analyze assembly programs, identify stalls in a pipeline, and generate stall-free assembly code. The program is expected to work both with and without data forwarding and hazard detection mechanisms.

## Problem Statement

The task at hand is to develop a program that reads an assembly program, identifies dependencies between instructions, and inserts no-operation (NOP) instructions to eliminate stalls caused by data hazards. The program should also calculate the total number of cycles required to execute the assembly code in a standard 5-stage pipeline.

The input program should follow specific conventions and assumptions:

- The program is syntactically correct.

- There is only one space between the instruction and the operands.

- Instructions are on separate lines.

- There are no blank lines, labels, or jump instructions.

- Registers can be specified using various naming conventions (e.g., x0, a0, t0).

## Approach

1. Reading Input

The program starts by reading the input assembly program from a file called "input.txt" and creates a character matrix to store the instructions. The `readFileToMatrix` function is used to read the input file, tokenize the instructions, and store them in the matrix.

Function named convertRegisterName rewrites all register names into stardard names that is of x type and are written into a file named “convertedInputs.txt.

Then we create a new matrix named temp\_matrix that reads file “convertedInputs.txt ” which contains transformed register names from other types to x type or standard type.

2. Identifying Dependencies

To identify dependencies and potential stalls in the pipeline, the program examines each instruction and its operands. It considers the data dependencies between instructions and classifies them based on their nature (load, arithmetic, or store operations) by classifier function of code.

3. Coding approach

The approach used here is the main loop takes 3 lines of inputs and verifies the classes of them by the classifier function and we take the numbers of registers from the temp\_matrix which stores the instructions in

X format and store them into arrays.Now with class comparison and these arrays we can build verification conditions as you can see in the code. This loop runs for all instructions except the last 2 for which a different set of instructions is used outside of the loop.

The loop avoids adding nop for special conditions like when the destination is x0 or where an element is added to zero and stored in the same register and the similar case for subtraction to

4. Output Generation

The program generates 4 outputs in the terminal: one for the case with no data forwarding and hazard detection named as no forwarding, another for the case with data forwarding named as with forwarding, original instructions named as matrix, and converted instructions named as matrix(temp). The output is written to "convertedInputs.txt." are the converted instructions.

The number of cycles of each case is reported below the corresponding outputs.

5. Testing

The program's correctness and functionality were verified using several test cases. These test cases include scenarios with various combinations of instructions and dependencies, as well as both simple and complex assembly code sequences. These test cases are submitted in file inputs.txt

The program was tested against various test cases to ensure its correctness and effectiveness. The test cases covered different combinations of instructions, dependencies, and the presence of data forwarding and hazard detection. These tests aimed to verify that the program correctly identifies dependencies inserts NOPs, and calculates the total number of cycles.

## Conclusion

The C program presented in this report successfully addresses the problem of identifying stalls in assembly programs and generating stall-free assembly codes. The program accommodates different register naming conventions, identifies dependencies, and inserts NOP instructions as needed to eliminate pipeline stalls. Additionally, it calculates the total number of cycles for executing the assembly code in a 5-stage pipeline.

The program has been tested against various scenarios and produces the expected output for different cases. It demonstrates the ability to handle dependencies and improve pipeline efficiency.

This report concludes the development and testing of the program, which provides a valuable tool for optimizing assembly code and understanding pipeline execution in a computer architecture context.

[End of Report]