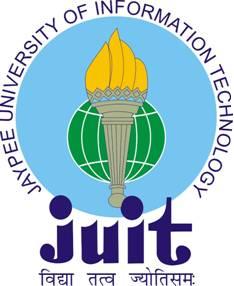
****

**A Project report on**

**PIPELINE SCHEDULING SIMULATOR**

**For the course:**

**COMPUTER ORGANIZATION AND ARCHITECTURE LAB**

**SUBMITTED BY:**

**Anmol Sharma & Vanshika Sharma**

**221030285 221030331**

**SUBMITTED TO:**

**Dr. Pardeep Garg**

**Table of Contents**

* **Abstract**
* **Introduction**
* **Methodology**
* **Code**
* **Results**
* **Conclusions**

**ABSTRACT**

Instruction pipelining is a crucial technique in modern processors, designed to improve throughput by overlapping the execution of multiple instructions. However, the presence of hazards such as Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW) can significantly hinder performance. This project aims to simulate a simplified instruction pipeline, focusing on hazard detection and resolution. The implemented simulation identifies dependencies between instructions, introduces stalls where necessary, and tracks the impact on pipeline performance. Through this, the project provides insights into how hazards disrupt instruction flow and the strategies to mitigate their effects. Additionally, the simulation calculates critical metrics such as total clock cycles, wasted cycles due to stalls, and cycles per instruction (CPI). The results highlight the interplay between pipeline design and performance, showcasing the challenges and solutions in modern processor architecture. This project serves as a practical exploration of pipelining concepts and their implementation in computer systems.

**INTRODUCTION**

PROPOSED MODEL

The ever-increasing demand for faster and more efficient processors has driven the evolution of techniques like instruction pipelining. Pipelining divides instruction execution into discrete stages, allowing multiple instructions to be processed simultaneously. This overlap significantly improves performance by maximizing hardware utilization. However, real-world implementations of pipelining are not without challenges. Dependencies between instructions, referred to as hazards, can disrupt the smooth progression of instructions through the pipeline. These hazards can lead to incorrect results if not addressed properly.

This project simulates an instruction pipeline to explore the behavior of hazards and their impact on performance. By focusing on three types of hazards—RAW, WAR, and WAW—the simulation dynamically identifies dependencies, introduces stalls where necessary, and calculates performance metrics. The implementation is structured around a five-stage pipeline: Fetch, Decode, Execute, Memory Access, and Writeback. The results provide a detailed trace of the pipeline’s state during execution and offer insights into how stalls affect overall efficiency.

This work is not only a practical demonstration of pipelining but also a foundation for understanding the design considerations in processor architecture. By analyzing hazard management strategies and their outcomes, the project bridges theoretical concepts and practical implementation.

**METHODOLOGY**

1. PipeLine Stages:

The pipeline consists of five stages: Fetch (FETCH), Decode (DECODE), Execute (EXECUTE), Memory Access (MEMORY), and Writeback (WRITEBACK). Each instruction progresses through these stages sequentially.

1. Hazard Detection:

* RAW Hazard: Occurs when an instruction depends on the result of a previous instruction.
* WAR Hazard: Occurs when an earlier instruction writes to a register after a subsequent instruction reads it.
* WAW Hazard: Occurs when multiple instructions write to the same register.  
  These hazards are identified by comparing destination and source registers between consecutive instructions.

1. Simulation Process:

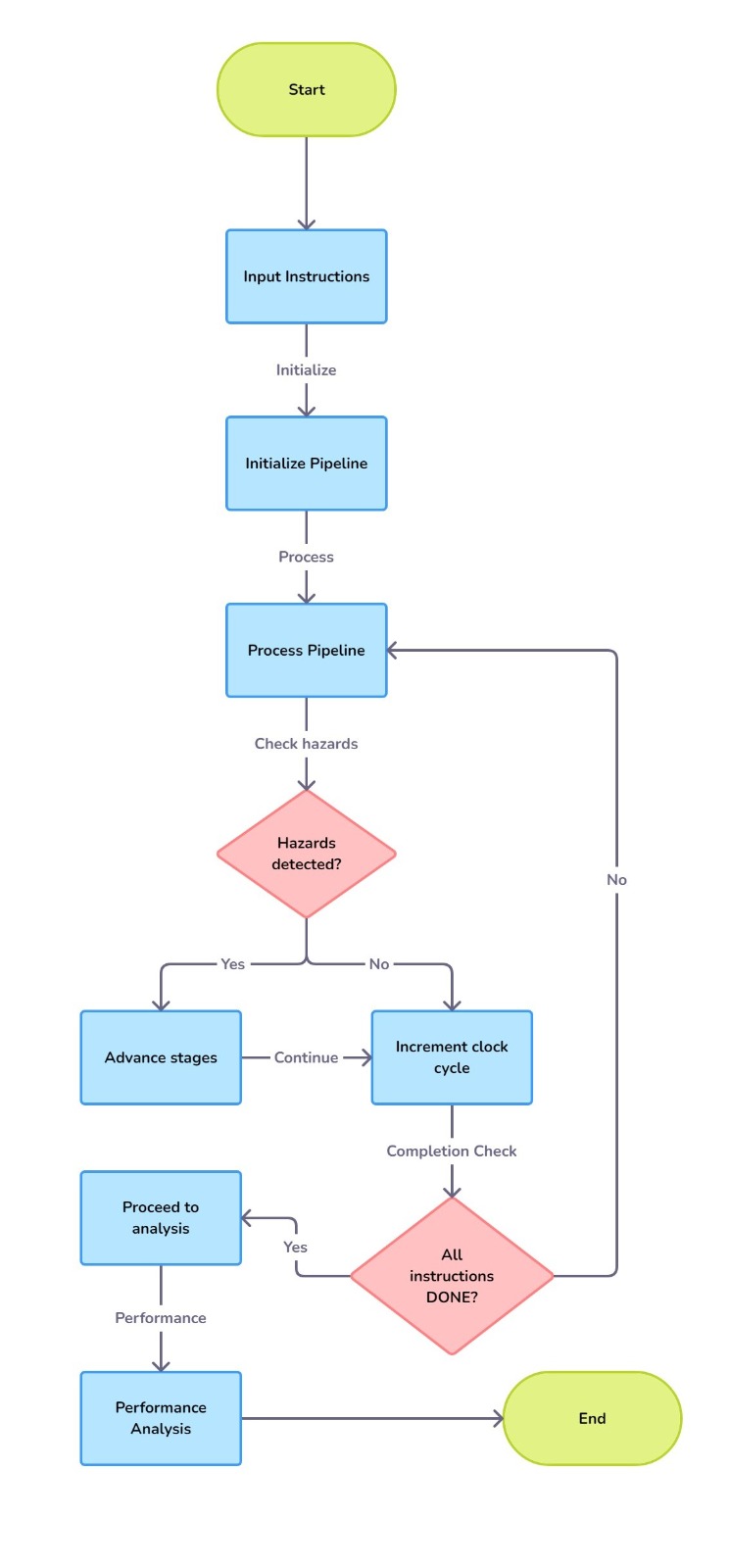
* User inputs the number of instructions and their details (operation, destination, and source registers).
* Each instruction is pushed into the pipeline, progressing one stage per clock cycle.
* Hazards are checked dynamically, and stalls are introduced if a dependency is detected.

1. Performance Metrics:

* Total clock cycles and wasted cycles due to stalls are calculated.
* Cycles Per Instruction (CPI) is computed to measure pipeline efficiency.

1. Implementation Details:

* The code is written in C++ using arrays for instruction data and an enum for pipeline stages.
* The simulation loops until all instructions reach the final stage.

****

**CODE:**

#include <iostream>

#include <string>

using namespace std;

const int MAX\_INSTRUCTIONS = 100;

int num\_instructions;                    // Number of instructions

string operations[MAX\_INSTRUCTIONS];     // Operation

string dest[MAX\_INSTRUCTIONS];           // Destination register

string src1[MAX\_INSTRUCTIONS];           // Source register 1

string src2[MAX\_INSTRUCTIONS];           // Source register 2

int pipeline\_stage[MAX\_INSTRUCTIONS];    // To represent the current pipeline stage for every instruction

int instruction\_counter = 0;             // To track the clock cycle for which instruction should enter fetch stage

int clock\_cycle = 0;                     // To store total number of clock cycles

int wasted\_cycle = 0;                    // To store the number of wasted clock cycles due to hazards

enum Pipeline\_Stage                      // Enum to store stages of pipeline

{

    FETCH,

    DECODE,

    EXECUTE,

    MEMORY,

    WRITEBACK,

    DONE

};

void get\_input()

{

    cout << "Enter the number of instructions: ";

    cin >> num\_instructions;

    for (int i = 0; i < num\_instructions; i++)

    {

        cout << "Enter instruction " << i + 1 << " (e.g., ADD R1 R2 R3): ";

        cin >> operations[i] >> dest[i] >> src1[i] >> src2[i];

        pipeline\_stage[i] = DONE;  // Set all instructions to DONE initially

    }

}

bool check\_hazard(int current, int previous)

{

    bool hazard\_detected = false;

    // For Read after Write Hazard

    if(src1[current] == dest[previous] || src2[current] == dest[previous] )

    {

        cout << "Read After Write Hazard detected between instruction " <<previous + 1 << " and instruction " << current + 1 << ". Adding a stall.\n";

        hazard\_detected = true;

    }

    // For Write After Read Hazard

    if( src1[previous] == dest[current] || src2[previous] == dest[current] )

    {

        cout << "Write After Read Hazard detected between Instruction " << previous + 1 << " and Instruction " << current + 1 << ". Adding a stall.\n";

        hazard\_detected = true;

    }

    // For Write After Write Hazard

    if( dest[previous] == dest[current] )

    {

        cout << "WAW Hazard detected between Instruction " << previous + 1 << " and Instruction " << current + 1 << ". Adding a stall.\n";

        hazard\_detected = true;

    }

    return hazard\_detected;

}

void pipline()

{

    int completed\_instructions = 0;

    cout << "Starting to push instructions into the Pipeline: \n";

    while(completed\_instructions < num\_instructions)

    {

        cout << "\nClock Cycle: " << clock\_cycle << "\n";  // To print the state of pipeline for this clock cycle

        // Process each instruction (one instruction moves forward per clock cycle)

        for(int i = 0; i < num\_instructions; i++)

        {

            // Skip instructions that are DONE

            if(pipeline\_stage[i] == DONE)

                continue;

            // Print the instruction state

            if(pipeline\_stage[i] != DONE) {

                cout << "Instruction " << i + 1 << " (" << operations[i] << "): ";

                switch (pipeline\_stage[i])

                {

                    case FETCH:

                        cout << "FETCH";

                        break;

                    case DECODE:

                        cout << "DECODE";

                        break;

                    case EXECUTE:

                        cout << "EXECUTE";

                        break;

                    case MEMORY:

                        cout << "MEMORY";

                        break;

                    case WRITEBACK:

                        cout << "WRITEBACK";

                        break;

                }

                cout << "\n";

            }

            // Check for hazards with the previous instruction

            // Check for hazards with the previous instruction

            if (i > 0)

            {

                while (pipeline\_stage[i - 1] != DONE && check\_hazard(i, i - 1))

                {

                    wasted\_cycle++; // Increase wasted cycle count due to hazard

                    cout << "Stalling Instruction " << i + 1 << " due to dependency on Instruction " << i << ".\n";

                    i++;

                }

            }

            // Move instruction to the next stage if not DONE

            if(pipeline\_stage[i] == FETCH)

            {

                pipeline\_stage[i] = DECODE;  // First instruction starts at FETCH and moves to DECODE

            }

            else if(pipeline\_stage[i] == DECODE)

            {

                pipeline\_stage[i] = EXECUTE;

            }

            else if(pipeline\_stage[i] == EXECUTE)

            {

                pipeline\_stage[i] = MEMORY;

            }

            else if(pipeline\_stage[i] == MEMORY)

            {

                pipeline\_stage[i] = WRITEBACK;

            }

            else if(pipeline\_stage[i] == WRITEBACK)

            {

                pipeline\_stage[i] = DONE;  // After Writeback, instruction is completed

                completed\_instructions++;  // Increment completed instruction count

            }

        }

        // Manage which instruction should enter the FETCH stage

        if(instruction\_counter < num\_instructions)

        {

            pipeline\_stage[instruction\_counter] = FETCH;

            instruction\_counter++;  // Move the counter to the next instruction

        }

        clock\_cycle++;  // Increment clock cycle after processing all instructions

    }

}

void final\_performance\_analysis()

{

    cout << "\nFinal Performance Analysis:\n";

    cout << "Total Clock Cycles: " << clock\_cycle - 1 << "\n";

    cout << "Total Wasted Clock Cycles: " << wasted\_cycle << "\n";

    cout << "Total Instructions: " << num\_instructions << "\n";

    cout << "Cycles Per Instruction (CPI): " << static\_cast<float>(clock\_cycle - 1) / num\_instructions << "\n";

    cout << "Average CPI (including stalls): " << static\_cast<float>((clock\_cycle - 1) + wasted\_cycle) / num\_instructions << "\n";

}

int main()

{

    get\_input();

    pipline();

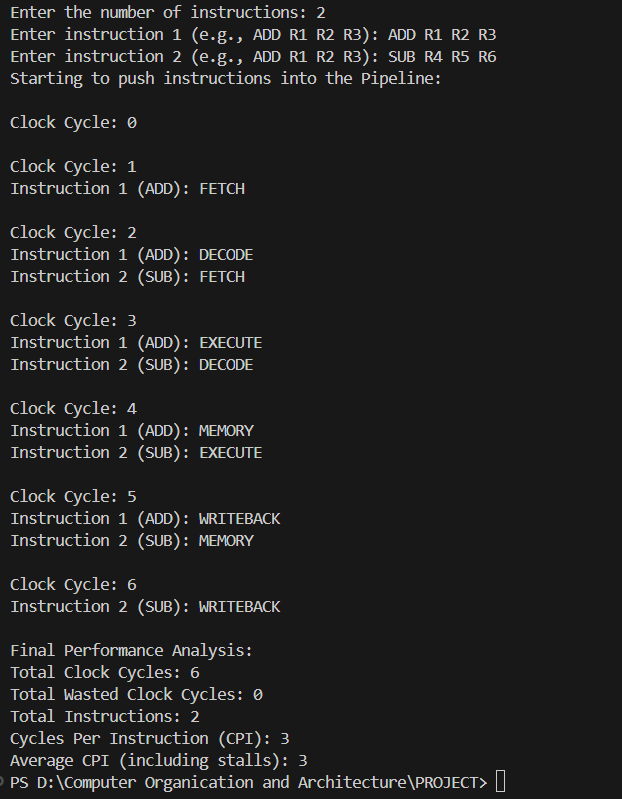
    final\_performance\_analysis();

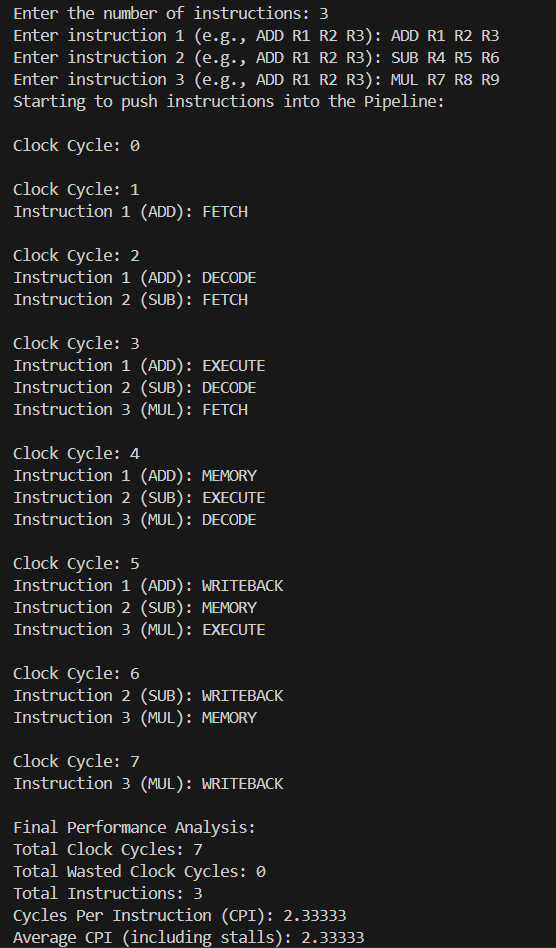
    return 0;

}

**RESULTS:**

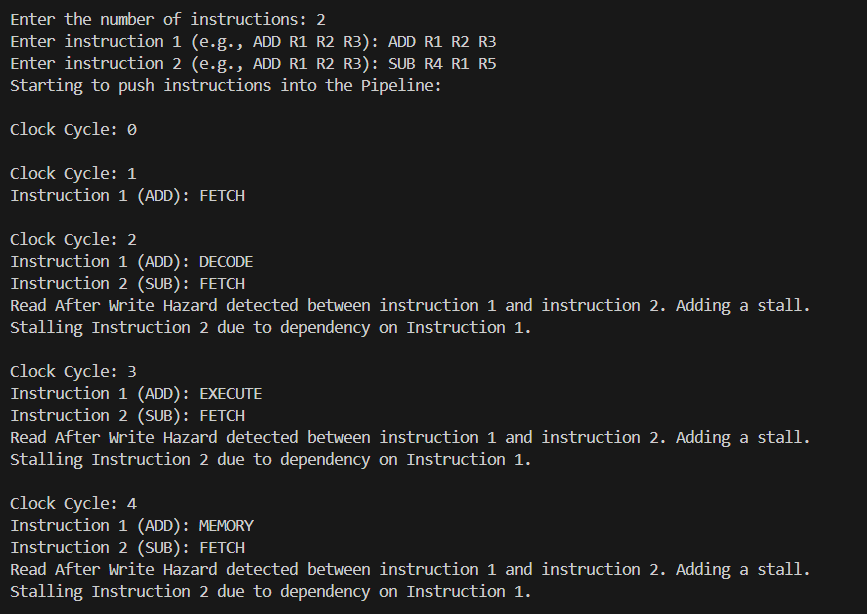
WITHOUT HAZARDS:

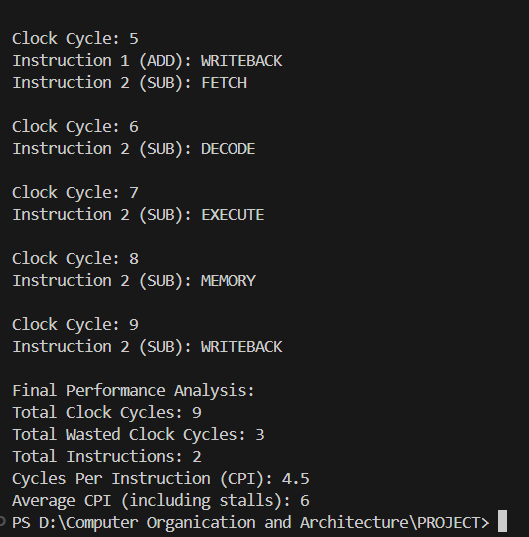
****

****

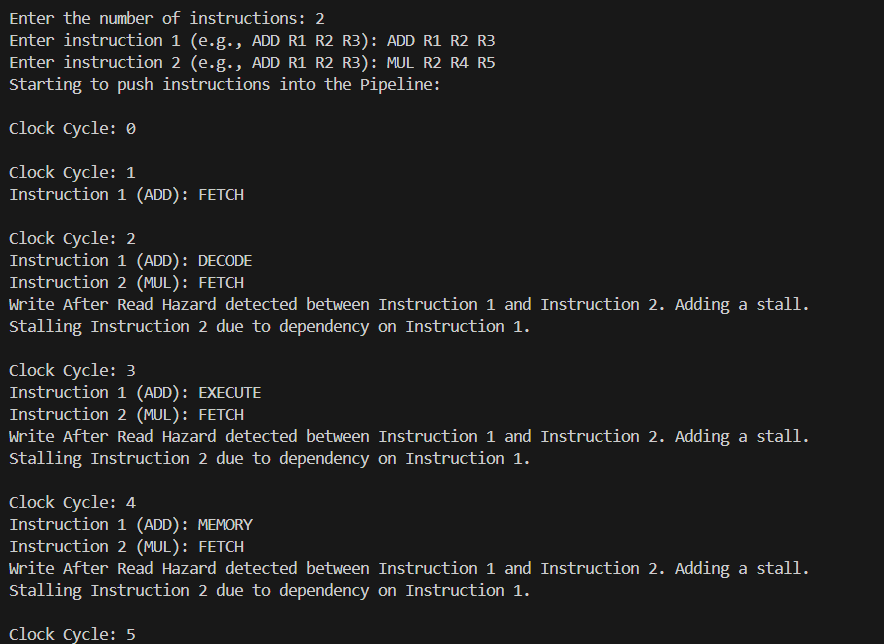
WITH HAZARDS:

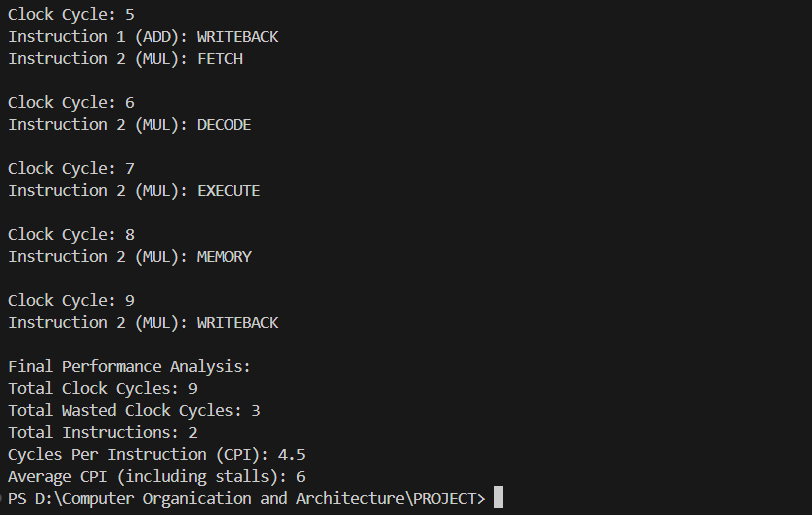
1. READ AFTER WRITE



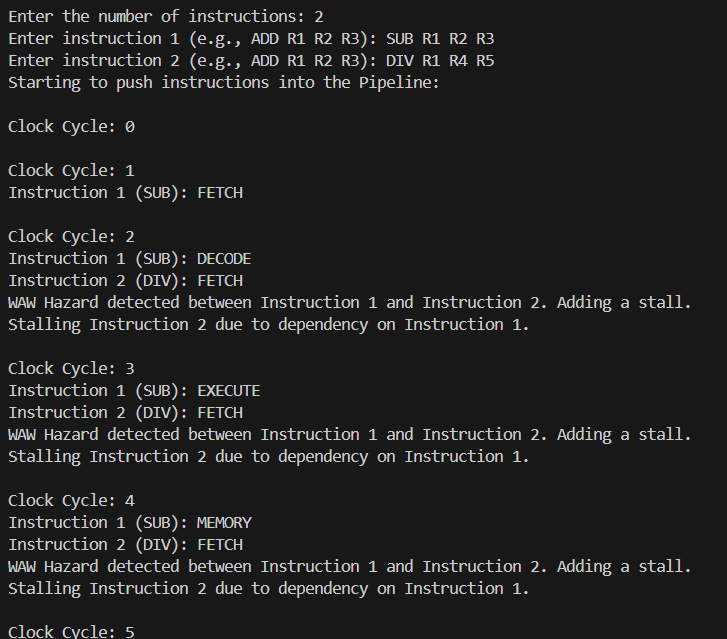


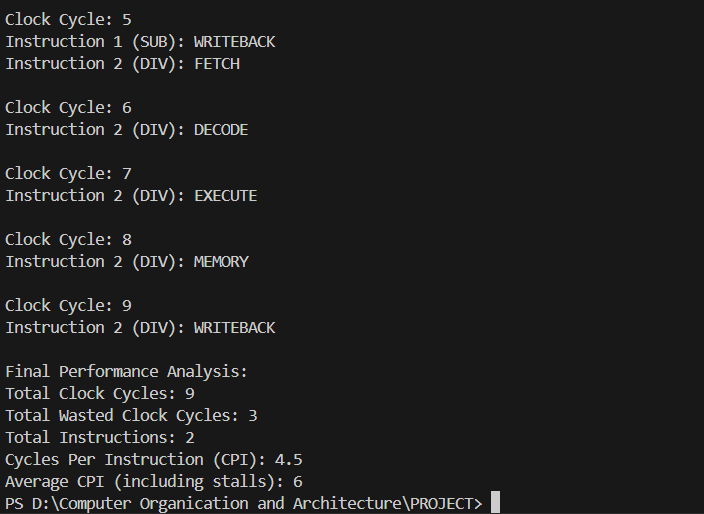
2. WRITE AFTER READ



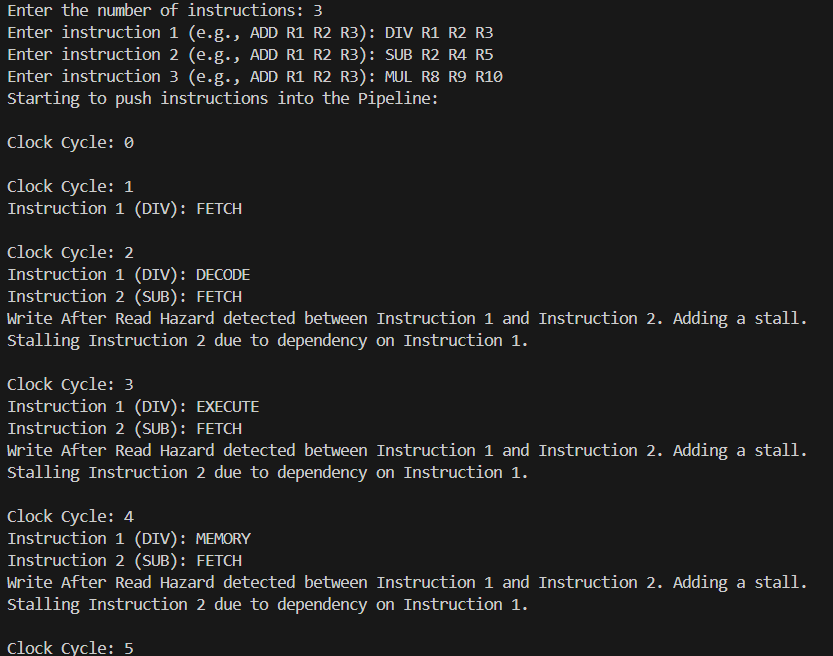


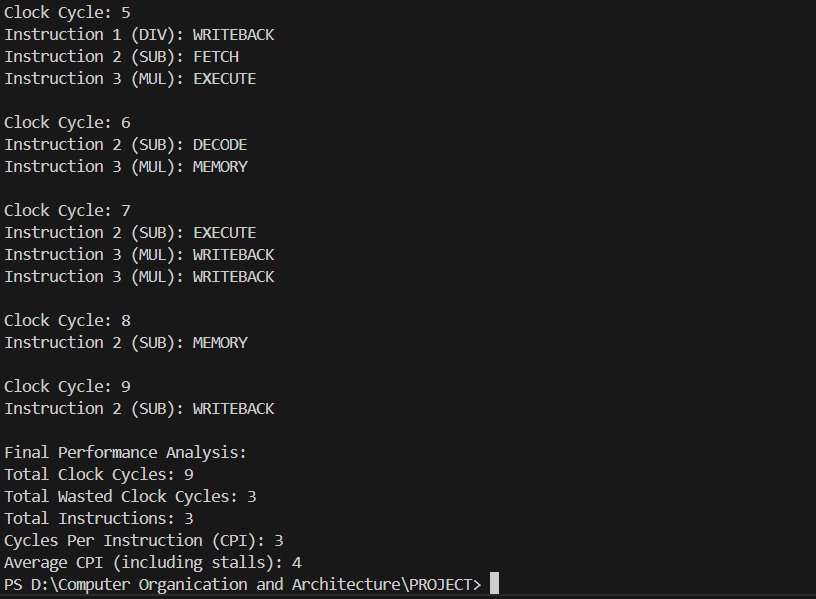
3. WRITE AFTER WRITE

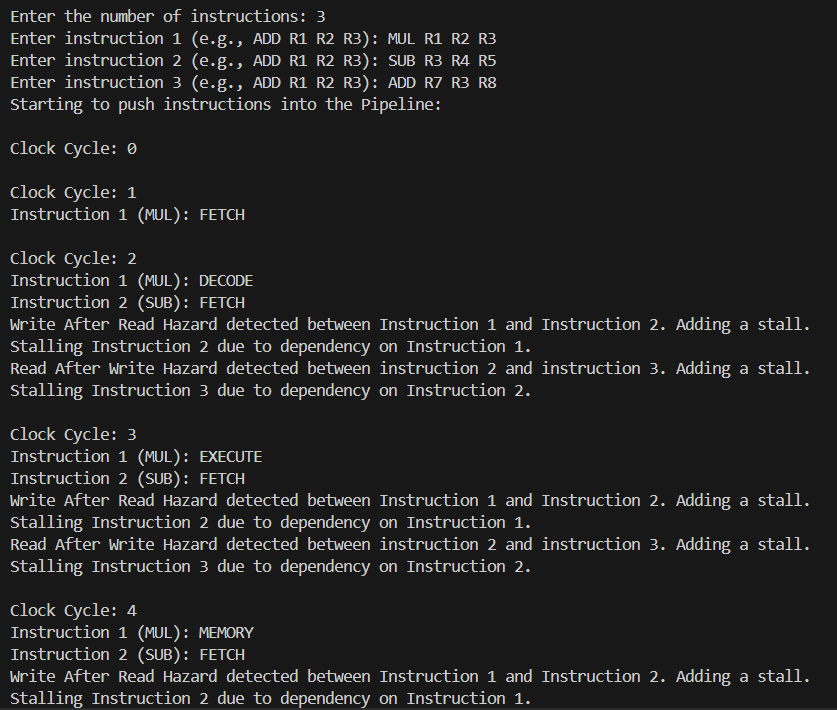


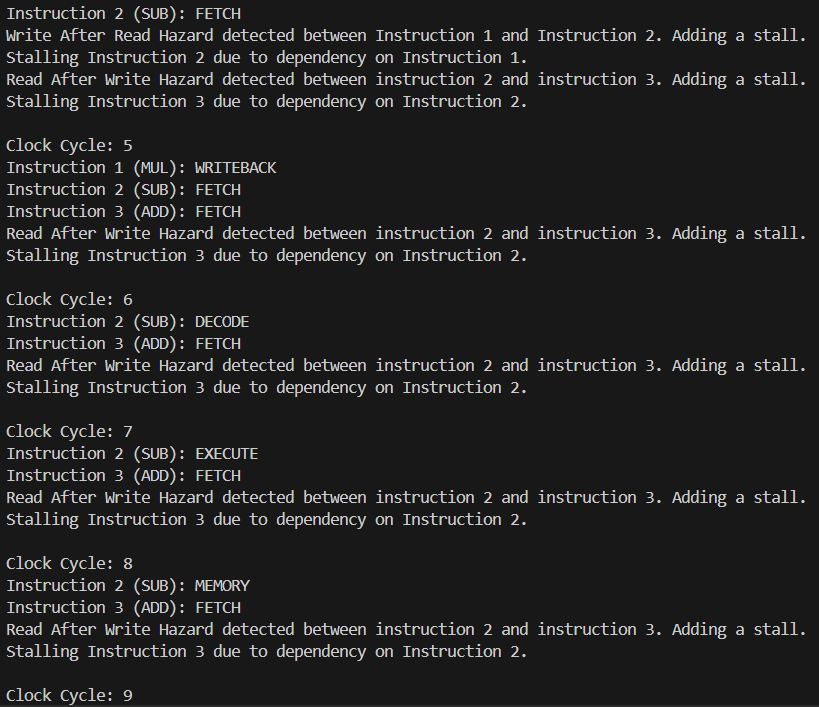


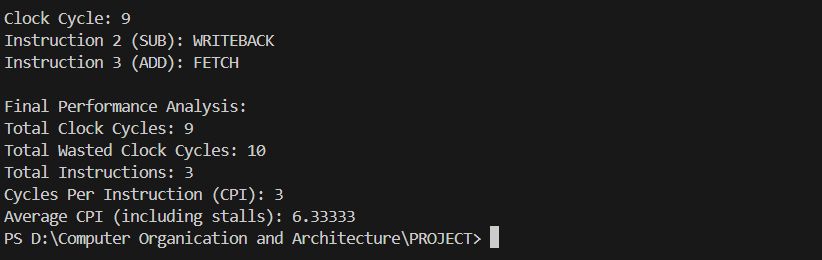
WITH HAZARDS AND 3 INSTRUCTIONS



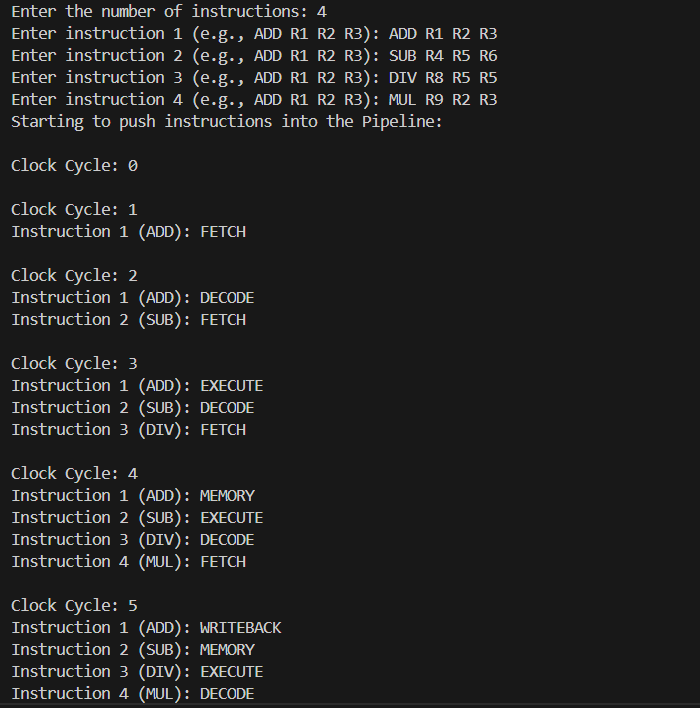


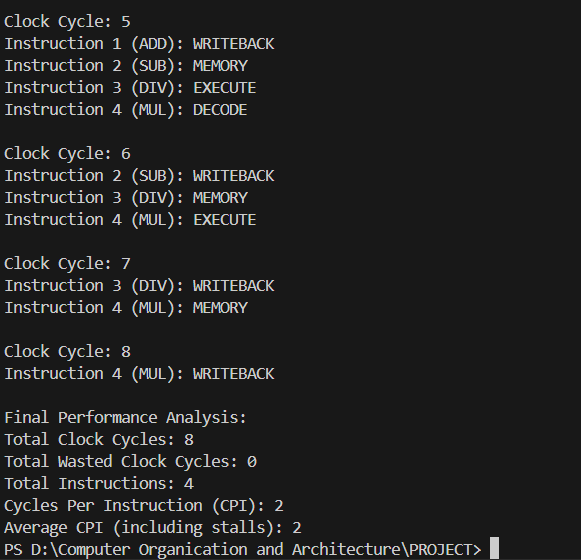






4 INSTRUCTIONS WITHOUT HAZARDS:





**Conclusion:**

The project successfully demonstrates how pipeline hazards impact instruction execution and performance. The simulation shows that managing these hazards is crucial for optimizing the pipeline's efficiency. The calculated CPI provides insights into the pipeline's performance under various workloads. This simulation serves as a foundational tool for understanding pipeline behavior and designing strategies to mitigate hazards in real-world processors.