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

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.

2. 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.

3. Simulation Process:

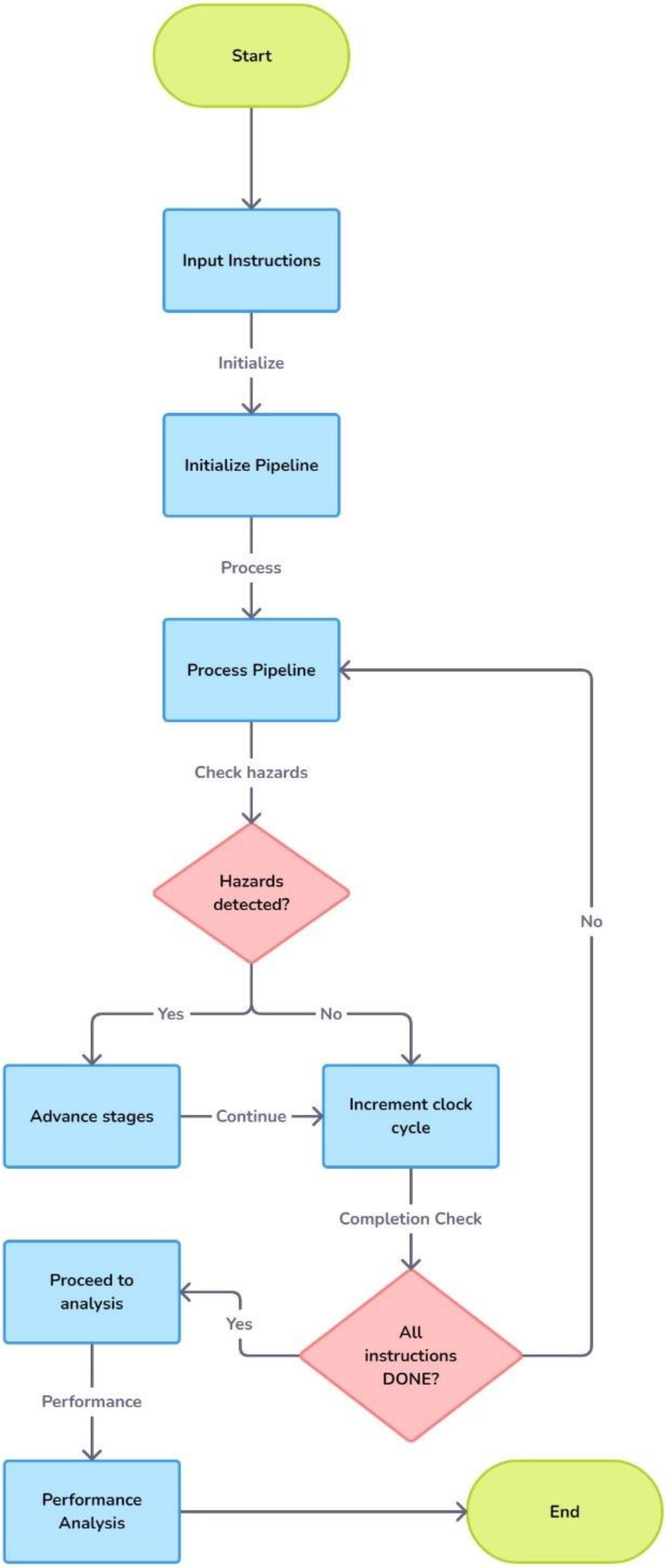
- User inputs the number of instructions and their details (operation, destination, and source registers).
- Each instruction is pushed into the pipeline, progressing one stage per clock cycle.
- Hazards are checked dynamically, and stalls are introduced if a dependency is detected.

4. Performance Metrics:

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

5. 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.


Start
Input Instructions
Initialize
Initialize Pipeline
Process
Process Pipeline
Check hazards
Hazards detected?
Yes
No
Advance stages
Continue
Increment clock cycle
Completion Check
No
All instructions DONE?
Yes
Proceed to analysis
Performance
Performance Analysis
End

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
```

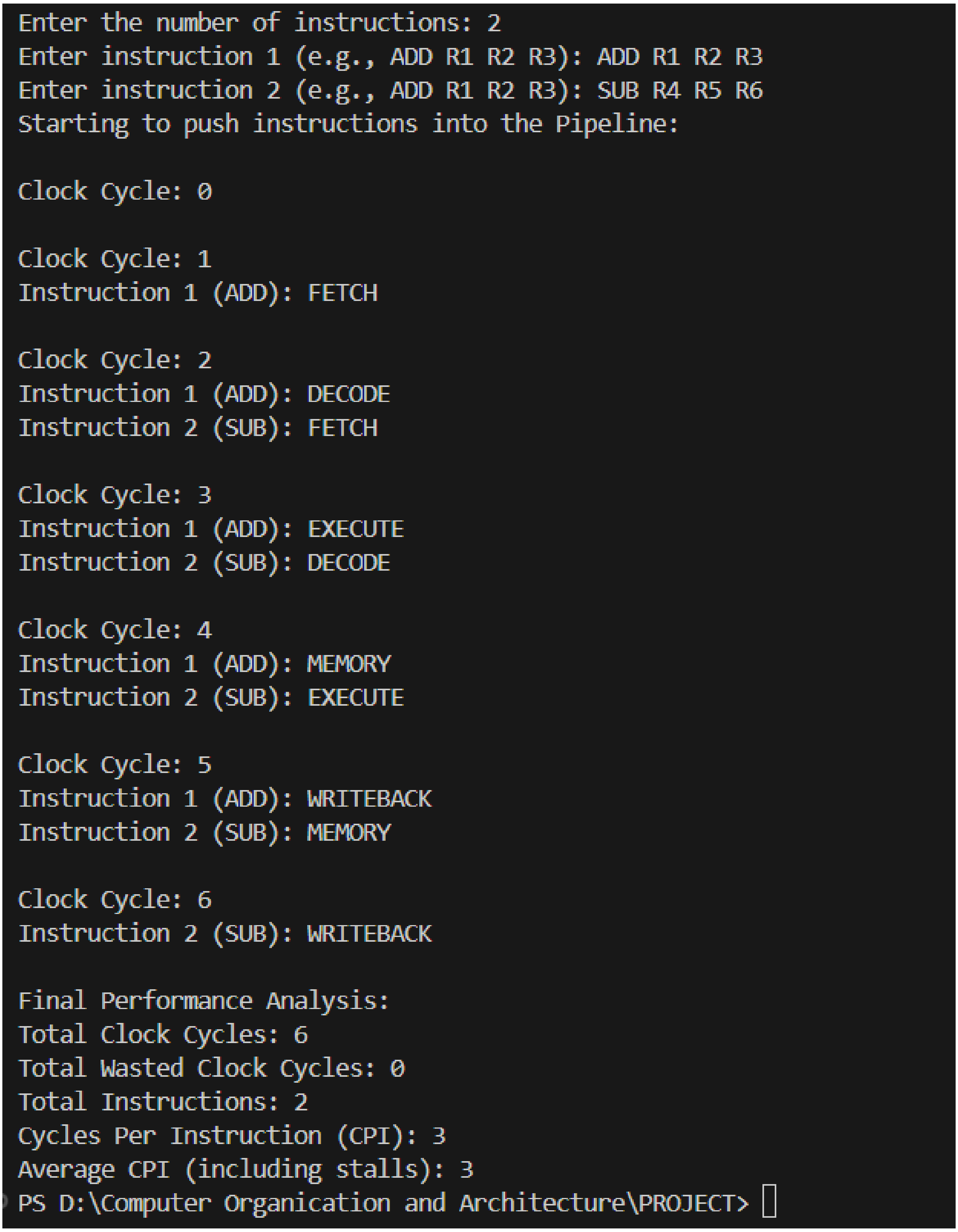
```
if(pipeline_stage[i] == DONE)
    continue;

// 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)
```

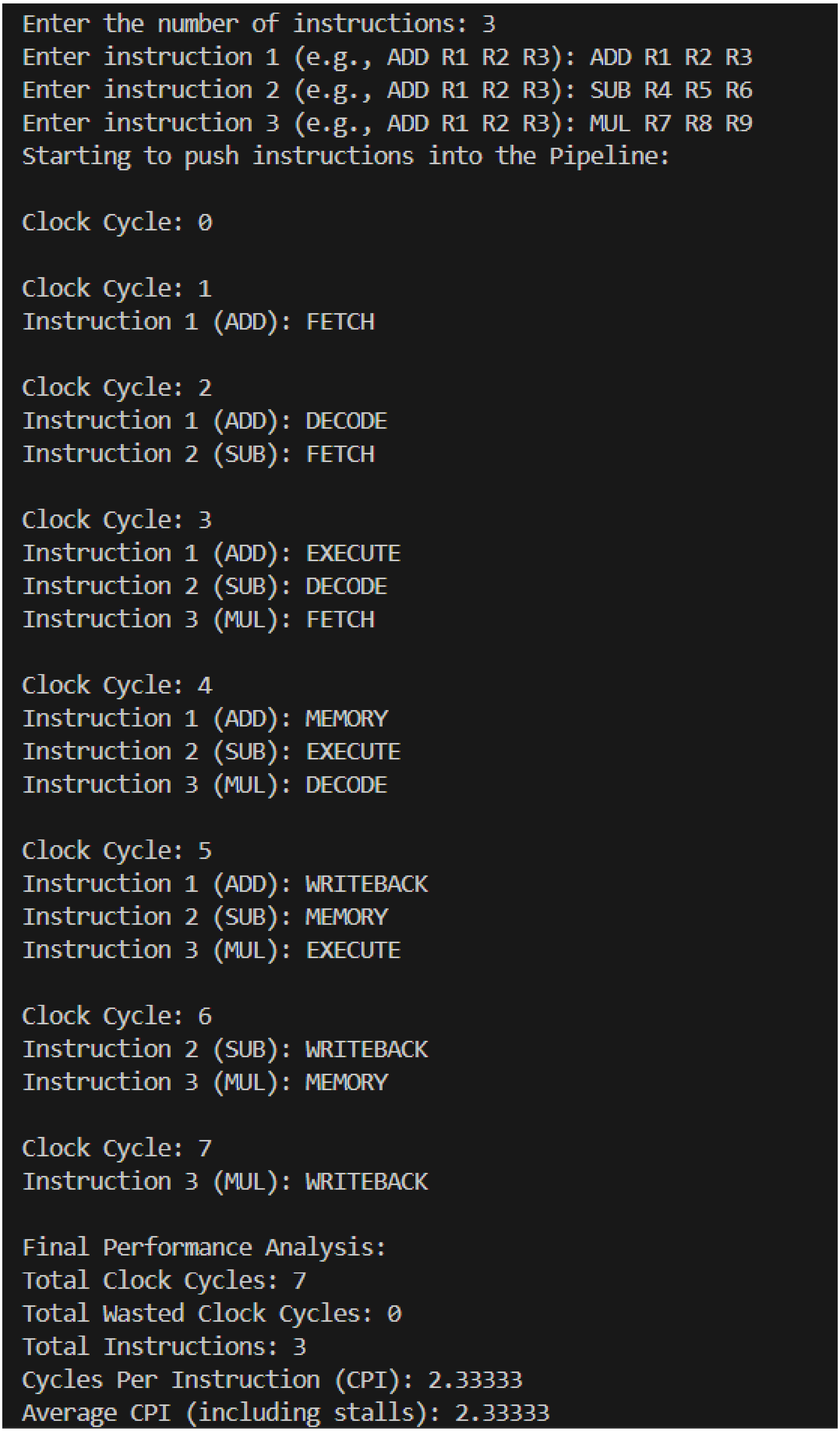
```
            {
                pipeline_stage[i] = EXECUTE;
            }
            else if(pipeline_stage[i] == EXECUTE)
            {
                pipeline_stage[i] = MEMORY;
            }
            else if(pipeline_stage[i] == MEMORY)
            {
                pipeline_stage[i] = WRITEBACK;
            }
            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:



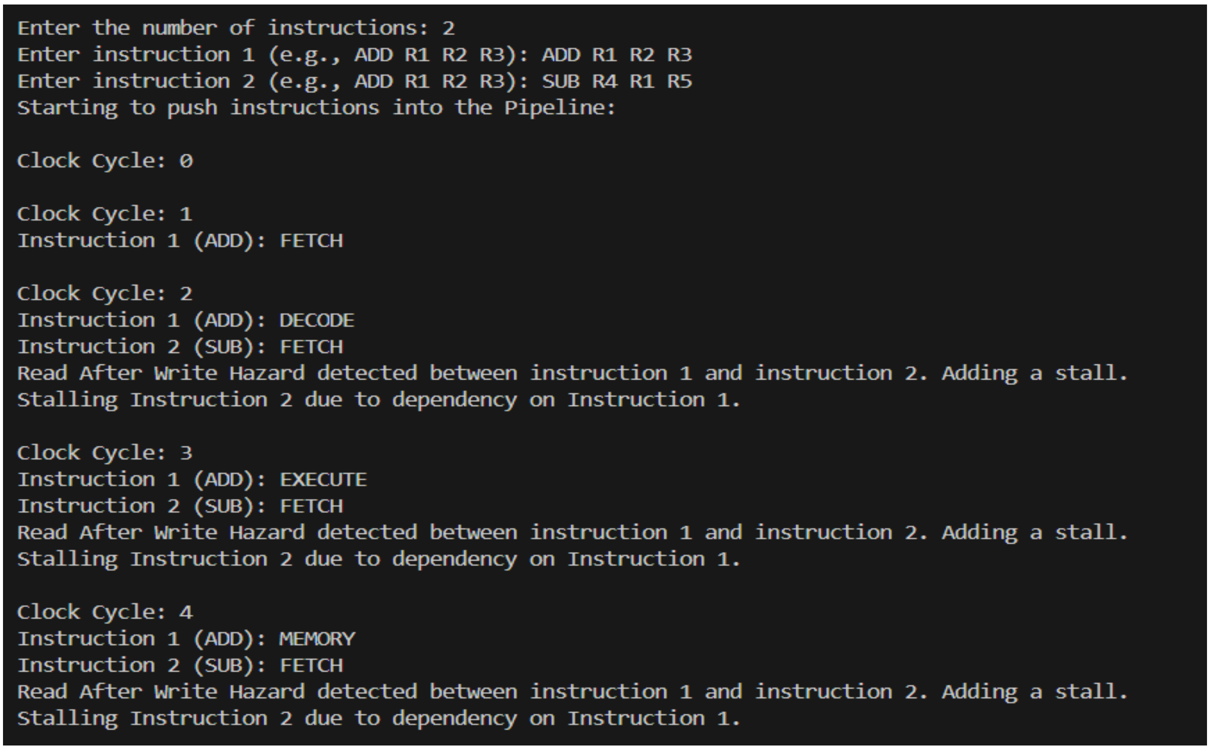
```
Enter the number of instructions: 2
Enter instruction 1 (e.g., ADD R1 R2 R3): ADD R1 R2 R3
Enter instruction 2 (e.g., ADD R1 R2 R3): SUB R4 R5 R6
Starting to push instructions into the Pipeline:

Clock Cycle: 0

Clock Cycle: 1
Instruction 1 (ADD): FETCH

Clock Cycle: 2
Instruction 1 (ADD): DECODE
Instruction 2 (SUB): FETCH

Clock Cycle: 3
Instruction 1 (ADD): EXECUTE
Instruction 2 (SUB): DECODE

Clock Cycle: 4
Instruction 1 (ADD): MEMORY
Instruction 2 (SUB): EXECUTE

Clock Cycle: 5
Instruction 1 (ADD): WRITEBACK
Instruction 2 (SUB): MEMORY

Clock Cycle: 6
Instruction 2 (SUB): WRITEBACK

Final Performance Analysis:
Total Clock Cycles: 6
Total Wasted Clock Cycles: 0
Total Instructions: 2
Cycles Per Instruction (CPI): 3
Average CPI (including stalls): 3
PS D:\Computer Organication and Architecture\PROJECT>
```



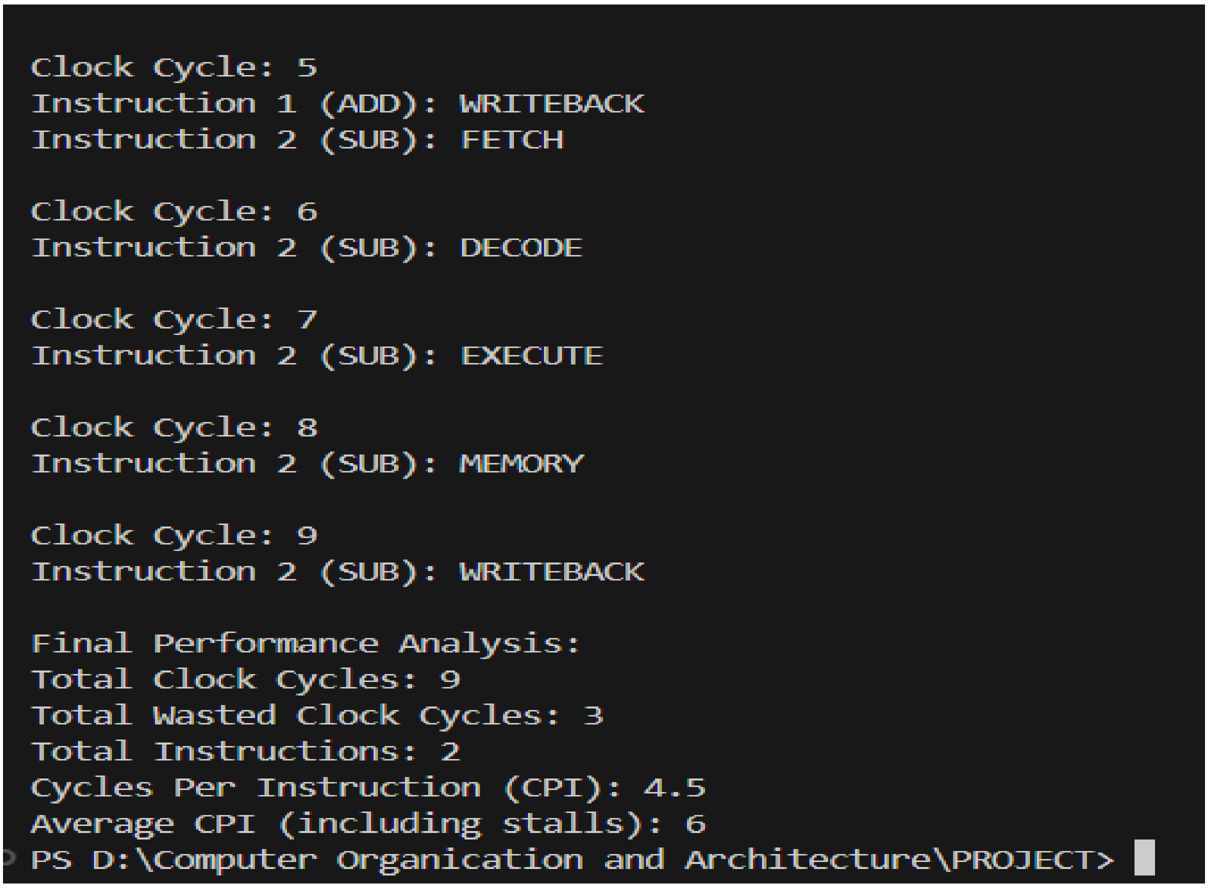
```
Enter the number of instructions: 3
Enter instruction 1 (e.g., ADD R1 R2 R3): ADD R1 R2 R3
Enter instruction 2 (e.g., ADD R1 R2 R3): SUB R4 R5 R6
Enter instruction 3 (e.g., ADD R1 R2 R3): MUL R7 R8 R9
Starting to push instructions into the Pipeline:

Clock Cycle: 0

Clock Cycle: 1
Instruction 1 (ADD): FETCH

Clock Cycle: 2
Instruction 1 (ADD): DECODE
Instruction 2 (SUB): FETCH

Clock Cycle: 3
Instruction 1 (ADD): EXECUTE
Instruction 2 (SUB): DECODE
Instruction 3 (MUL): FETCH

Clock Cycle: 4
Instruction 1 (ADD): MEMORY
Instruction 2 (SUB): EXECUTE
Instruction 3 (MUL): DECODE

Clock Cycle: 5
Instruction 1 (ADD): WRITEBACK
Instruction 2 (SUB): MEMORY
Instruction 3 (MUL): EXECUTE

Clock Cycle: 6
Instruction 2 (SUB): WRITEBACK
Instruction 3 (MUL): MEMORY

Clock Cycle: 7
Instruction 3 (MUL): WRITEBACK

Final Performance Analysis:
Total Clock Cycles: 7
Total Wasted Clock Cycles: 0
Total Instructions: 3
Cycles Per Instruction (CPI): 2.33333
Average CPI (including stalls): 2.33333
```

WITH HAZARDS:

1. READ AFTER WRITE

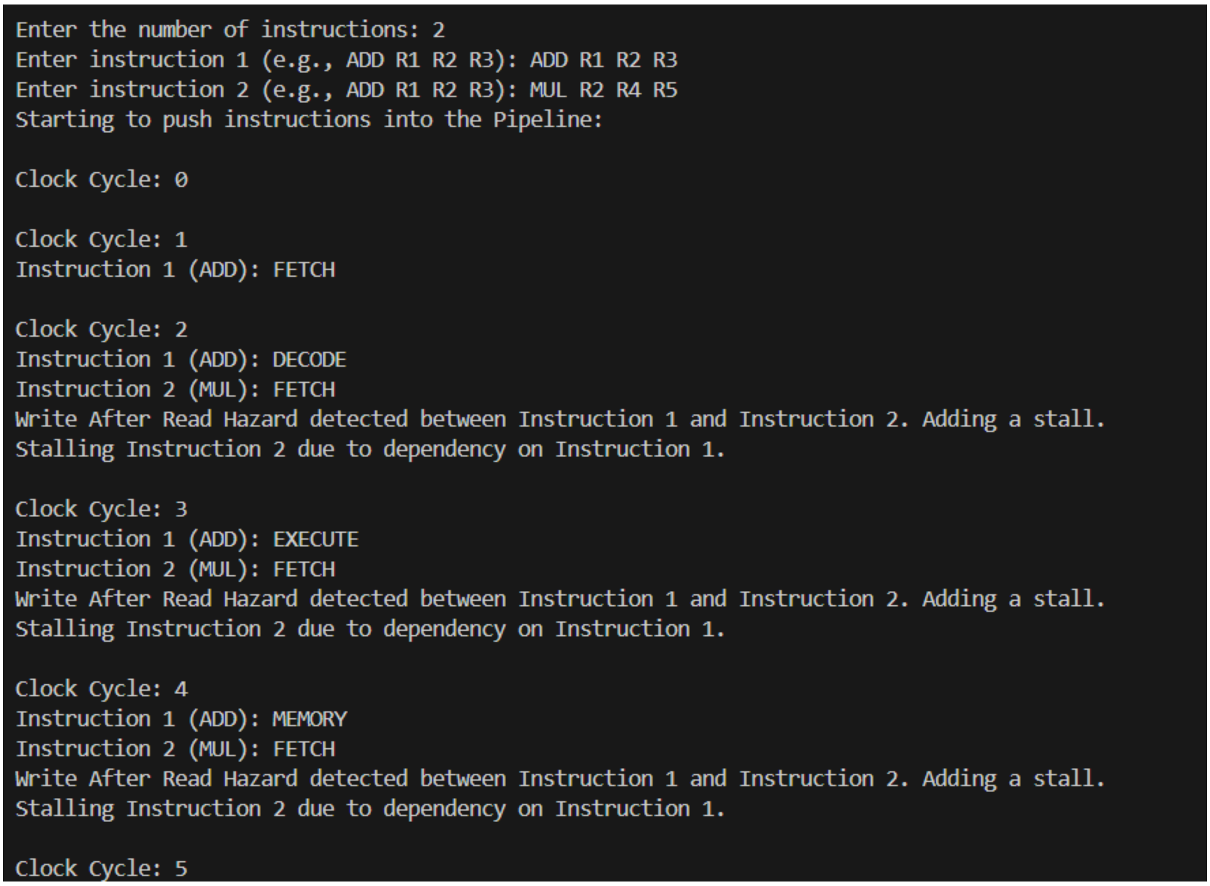


```
Enter the number of instructions: 2
Enter instruction 1 (e.g., ADD R1 R2 R3): ADD R1 R2 R3
Enter instruction 2 (e.g., ADD R1 R2 R3): SUB R4 R1 R5
Starting to push instructions into the Pipeline:

Clock Cycle: 0

Clock Cycle: 1
Instruction 1 (ADD): FETCH

Clock Cycle: 2
Instruction 1 (ADD): DECODE
Instruction 2 (SUB): FETCH
Read After Write Hazard detected between instruction 1 and instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.

Clock Cycle: 3
Instruction 1 (ADD): EXECUTE
Instruction 2 (SUB): FETCH
Read After Write Hazard detected between instruction 1 and instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.

Clock Cycle: 4
Instruction 1 (ADD): MEMORY
Instruction 2 (SUB): FETCH
Read After Write Hazard detected between instruction 1 and instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.
```

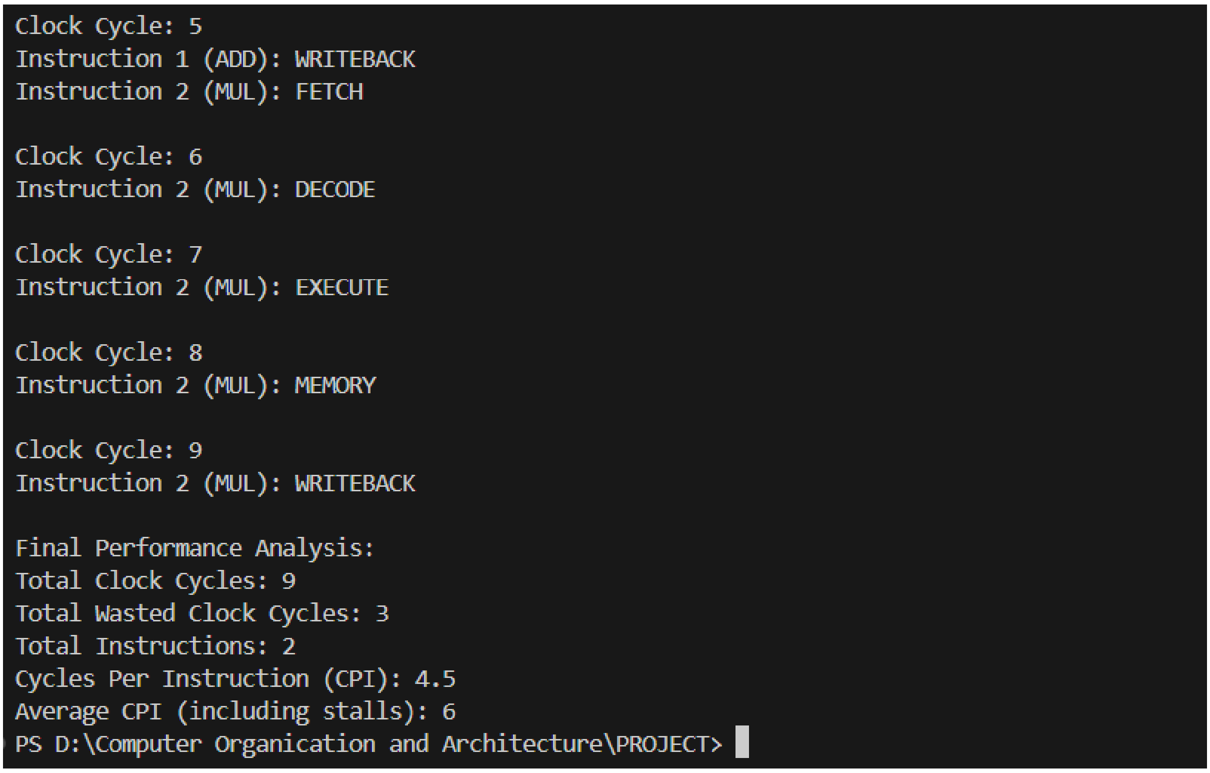


```
Clock Cycle: 5
Instruction 1 (ADD): WRITEBACK
Instruction 2 (SUB): FETCH

Clock Cycle: 6
Instruction 2 (SUB): DECODE

Clock Cycle: 7
Instruction 2 (SUB): EXECUTE

Clock Cycle: 8
Instruction 2 (SUB): MEMORY

Clock Cycle: 9
Instruction 2 (SUB): WRITEBACK

Final Performance Analysis:
Total Clock Cycles: 9
Total Wasted Clock Cycles: 3
Total Instructions: 2
Cycles Per Instruction (CPI): 4.5
Average CPI (including stalls): 6
PS D:\Computer Organication and Architecture\PROJECT>
```

2. WRITE AFTER READ

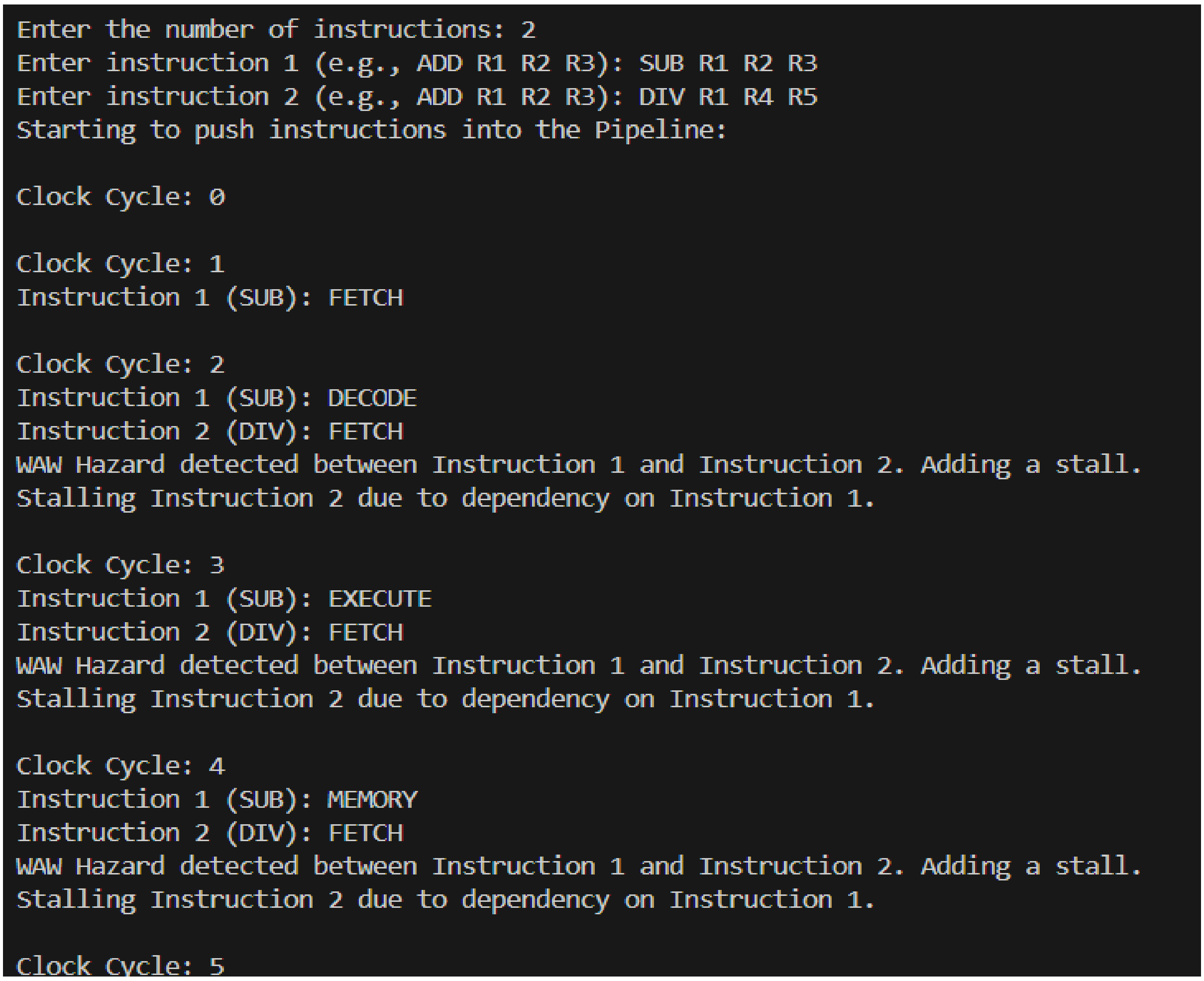


```
Enter the number of instructions: 2
Enter instruction 1 (e.g., ADD R1 R2 R3): ADD R1 R2 R3
Enter instruction 2 (e.g., ADD R1 R2 R3): MUL R2 R4 R5
Starting to push instructions into the Pipeline:

Clock Cycle: 0

Clock Cycle: 1
Instruction 1 (ADD): FETCH

Clock Cycle: 2
Instruction 1 (ADD): DECODE
Instruction 2 (MUL): FETCH
Write After Read Hazard detected between Instruction 1 and Instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.

Clock Cycle: 3
Instruction 1 (ADD): EXECUTE
Instruction 2 (MUL): FETCH
Write After Read Hazard detected between Instruction 1 and Instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.

Clock Cycle: 4
Instruction 1 (ADD): MEMORY
Instruction 2 (MUL): FETCH
Write After Read Hazard detected between Instruction 1 and Instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.

Clock Cycle: 5
```

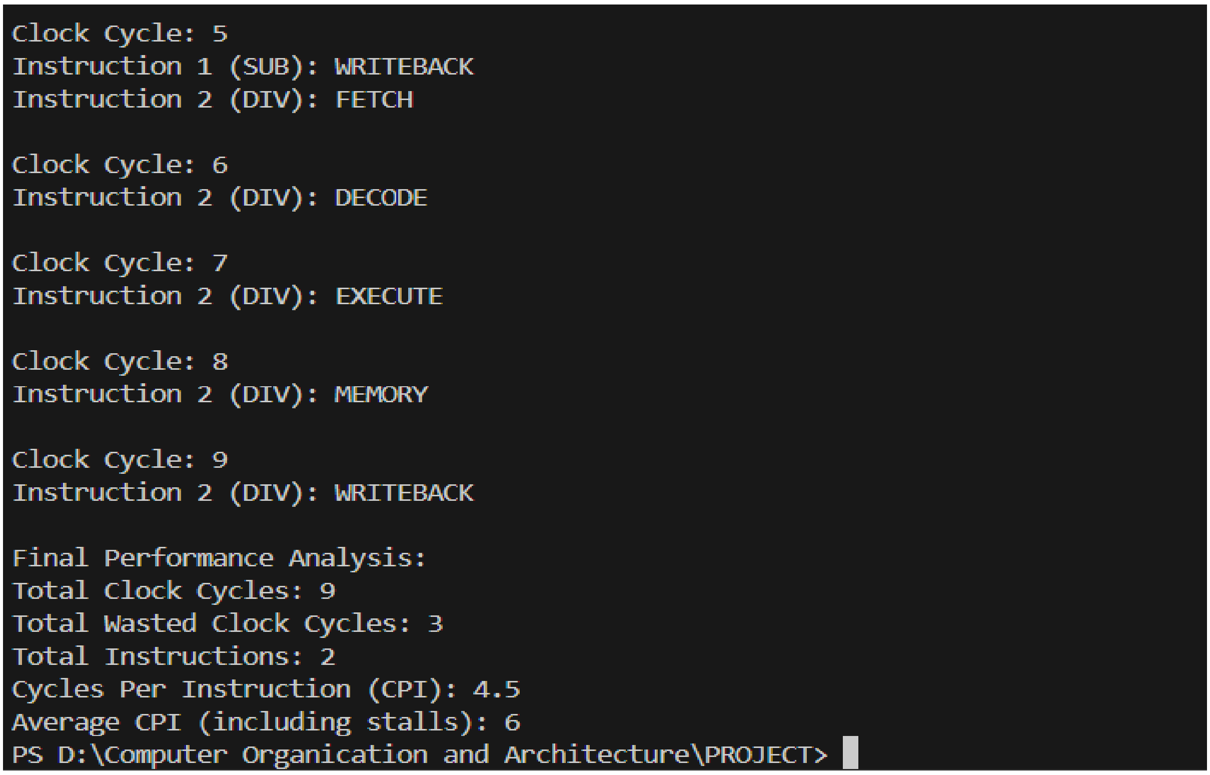


```
Clock Cycle: 5
Instruction 1 (ADD): WRITEBACK
Instruction 2 (MUL): FETCH

Clock Cycle: 6
Instruction 2 (MUL): DECODE

Clock Cycle: 7
Instruction 2 (MUL): EXECUTE

Clock Cycle: 8
Instruction 2 (MUL): MEMORY

Clock Cycle: 9
Instruction 2 (MUL): WRITEBACK

Final Performance Analysis:
Total Clock Cycles: 9
Total Wasted Clock Cycles: 3
Total Instructions: 2
Cycles Per Instruction (CPI): 4.5
Average CPI (including stalls): 6
PS D:\Computer Organication and Architecture\PROJECT>
```

3. WRITE AFTER WRITE

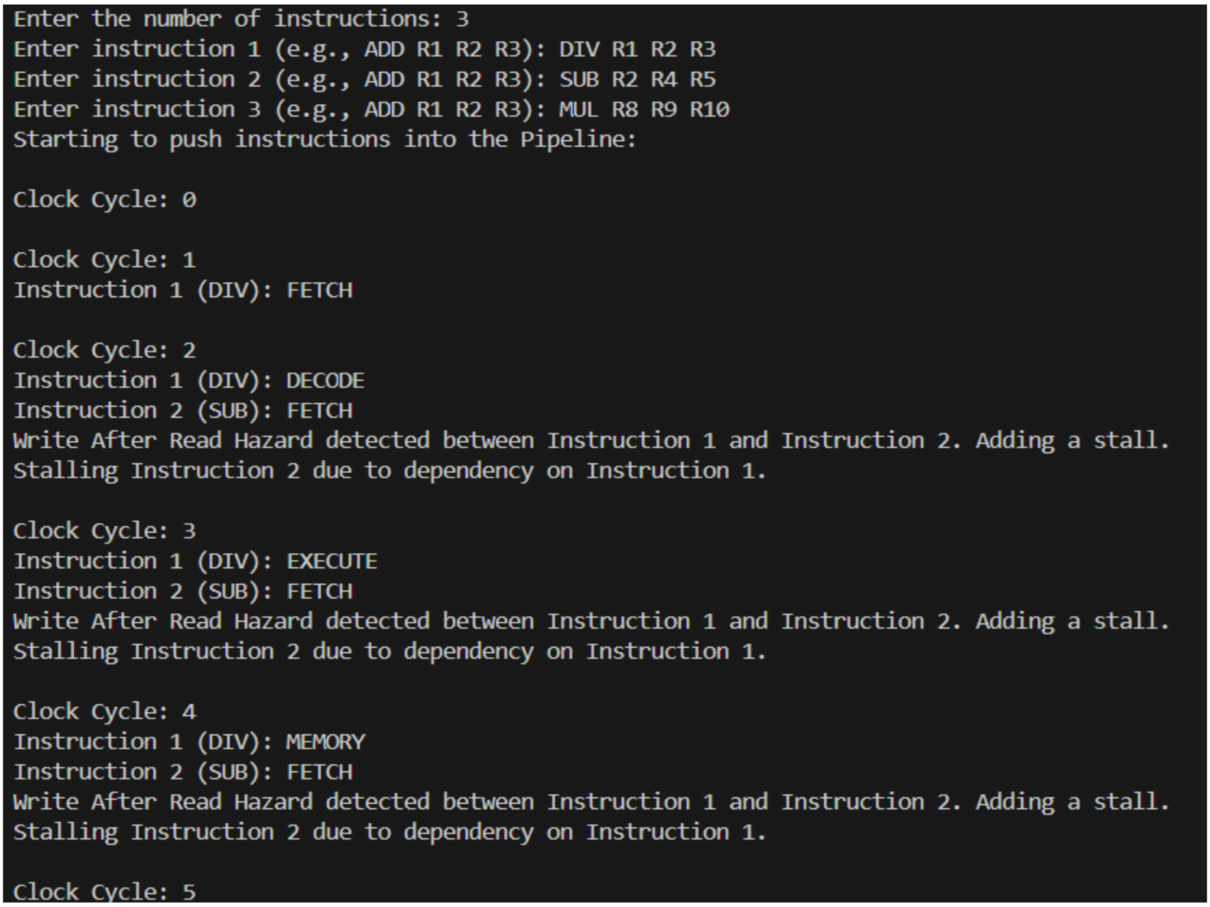


```
Enter the number of instructions: 2
Enter instruction 1 (e.g., ADD R1 R2 R3): SUB R1 R2 R3
Enter instruction 2 (e.g., ADD R1 R2 R3): DIV R1 R4 R5
Starting to push instructions into the Pipeline:

Clock Cycle: 0

Clock Cycle: 1
Instruction 1 (SUB): FETCH

Clock Cycle: 2
Instruction 1 (SUB): DECODE
Instruction 2 (DIV): FETCH
WAW Hazard detected between Instruction 1 and Instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.

Clock Cycle: 3
Instruction 1 (SUB): EXECUTE
Instruction 2 (DIV): FETCH
WAW Hazard detected between Instruction 1 and Instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.

Clock Cycle: 4
Instruction 1 (SUB): MEMORY
Instruction 2 (DIV): FETCH
WAW Hazard detected between Instruction 1 and Instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.

Clock Cycle: 5
```

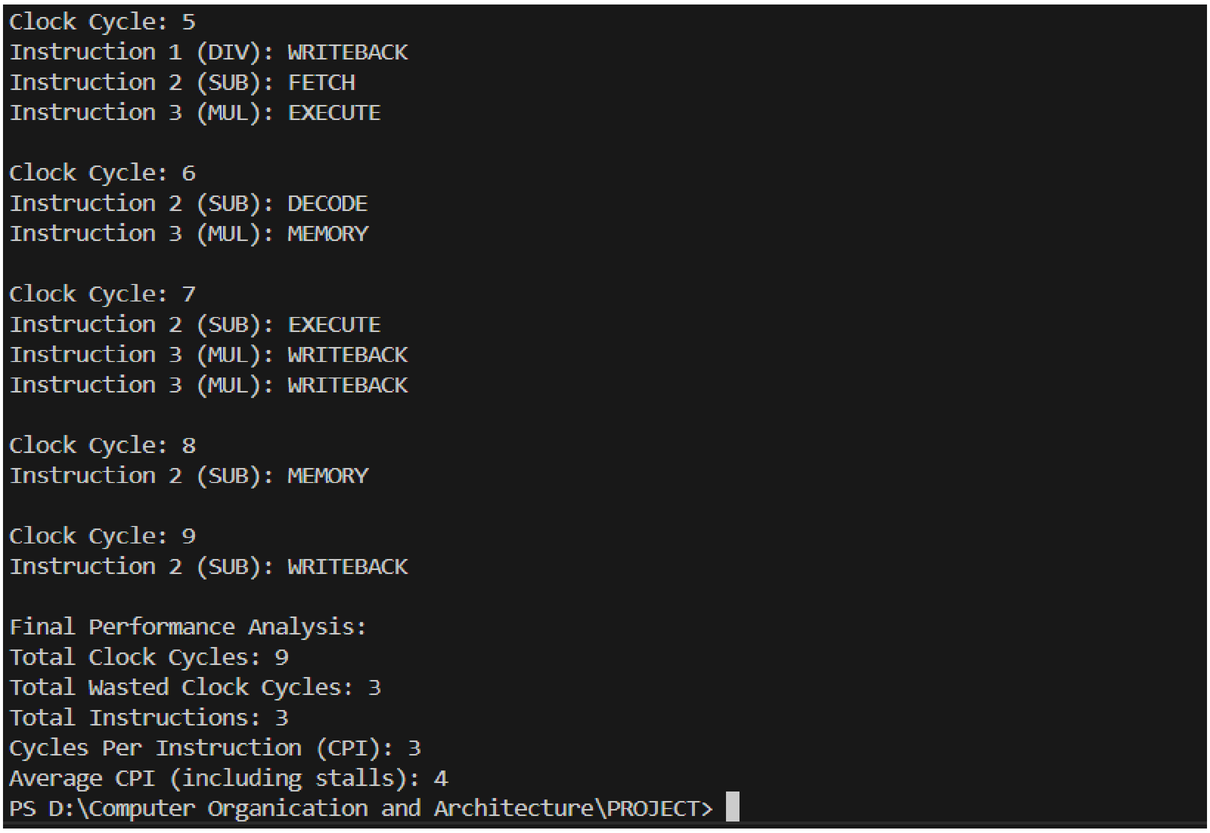


```
Clock Cycle: 5
Instruction 1 (SUB): WRITEBACK
Instruction 2 (DIV): FETCH

Clock Cycle: 6
Instruction 2 (DIV): DECODE

Clock Cycle: 7
Instruction 2 (DIV): EXECUTE

Clock Cycle: 8
Instruction 2 (DIV): MEMORY

Clock Cycle: 9
Instruction 2 (DIV): WRITEBACK

Final Performance Analysis:
Total Clock Cycles: 9
Total Wasted Clock Cycles: 3
Total Instructions: 2
Cycles Per Instruction (CPI): 4.5
Average CPI (including stalls): 6
PS D:\Computer Organication and Architecture\PROJECT> █
```

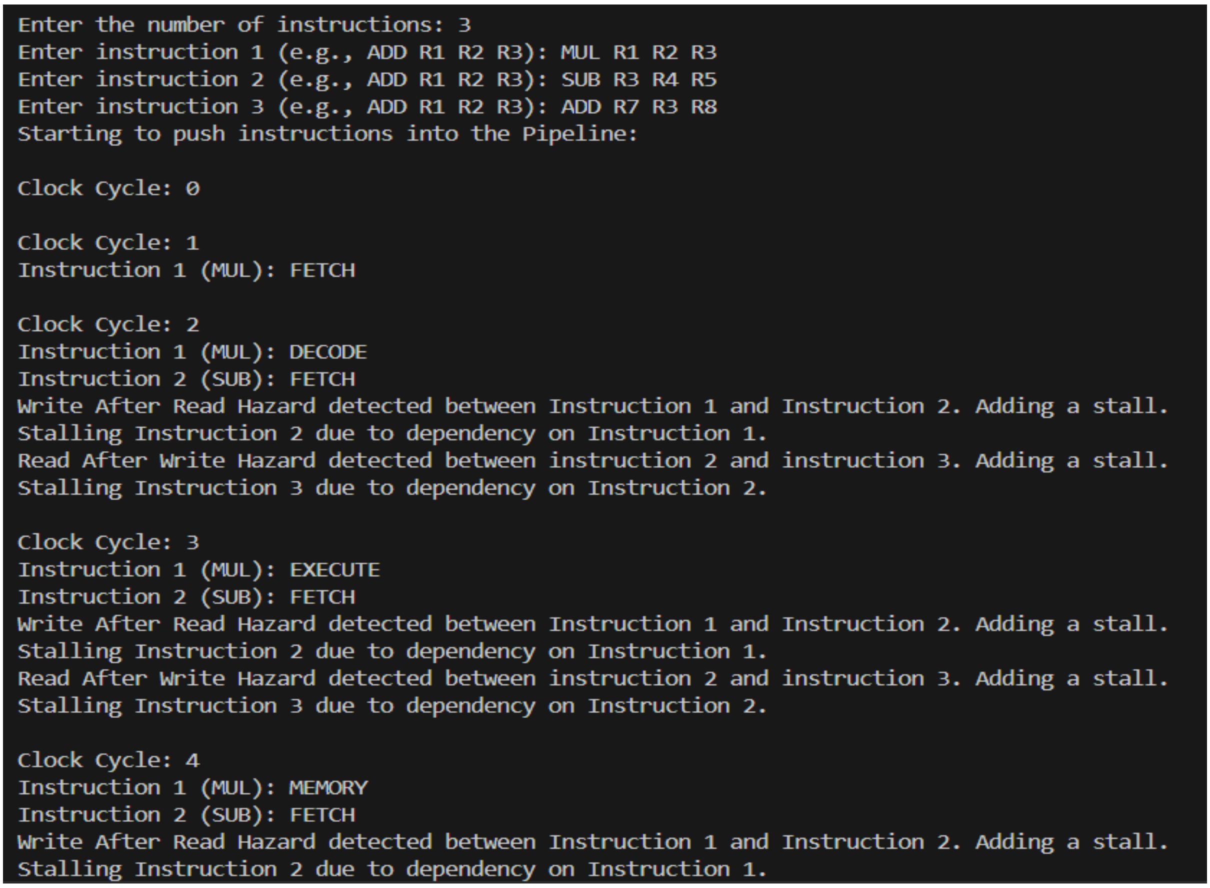
WITH HAZARDS AND 3 INSTRUCTIONS



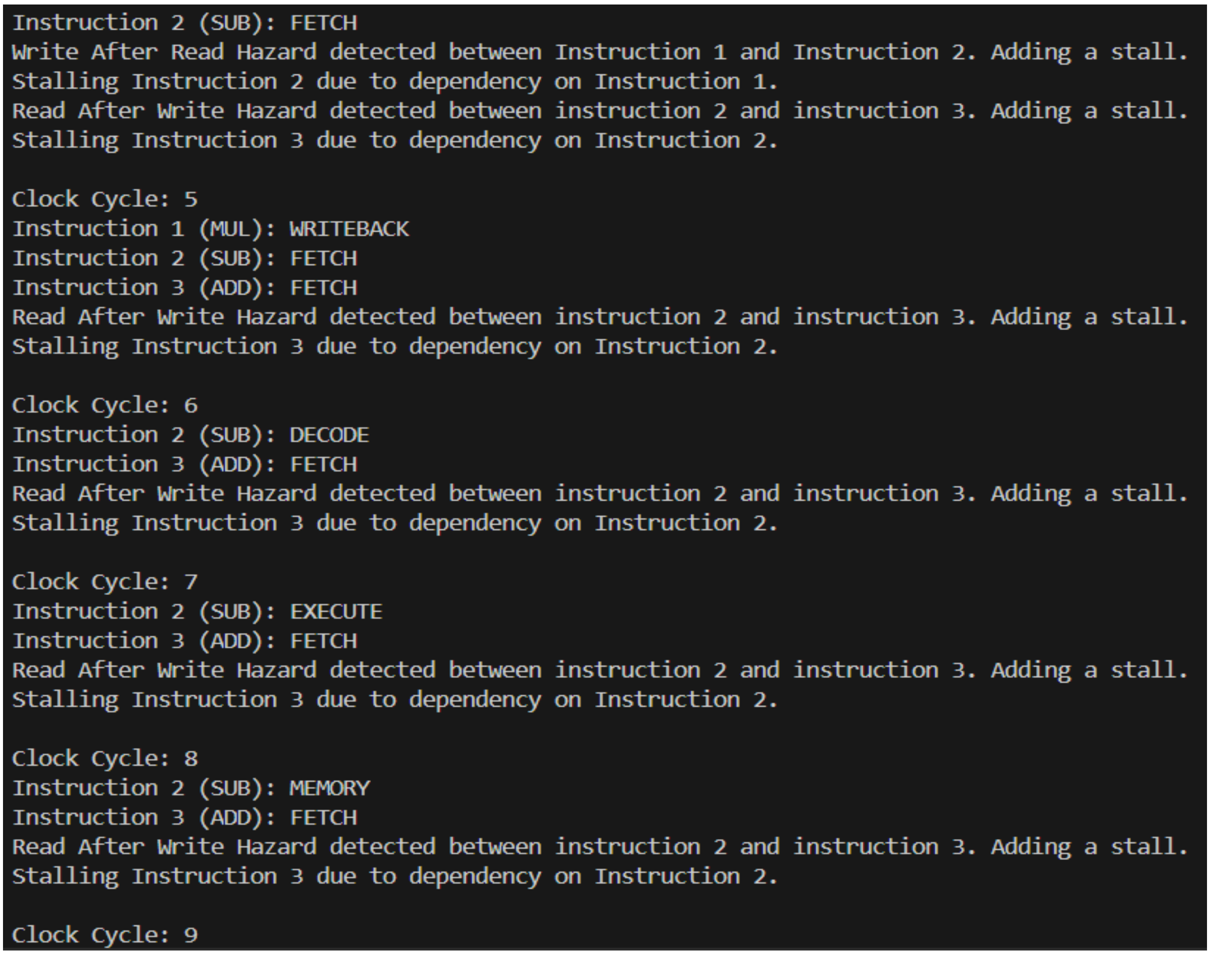
```
Enter the number of instructions: 3
Enter instruction 1 (e.g., ADD R1 R2 R3): DIV R1 R2 R3
Enter instruction 2 (e.g., ADD R1 R2 R3): SUB R2 R4 R5
Enter instruction 3 (e.g., ADD R1 R2 R3): MUL R8 R9 R10
Starting to push instructions into the Pipeline:

Clock Cycle: 0

Clock Cycle: 1
Instruction 1 (DIV): FETCH

Clock Cycle: 2
Instruction 1 (DIV): DECODE
Instruction 2 (SUB): FETCH
Write After Read Hazard detected between Instruction 1 and Instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.

Clock Cycle: 3
Instruction 1 (DIV): EXECUTE
Instruction 2 (SUB): FETCH
Write After Read Hazard detected between Instruction 1 and Instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.

Clock Cycle: 4
Instruction 1 (DIV): MEMORY
Instruction 2 (SUB): FETCH
Write After Read Hazard detected between Instruction 1 and Instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.

Clock Cycle: 5
```



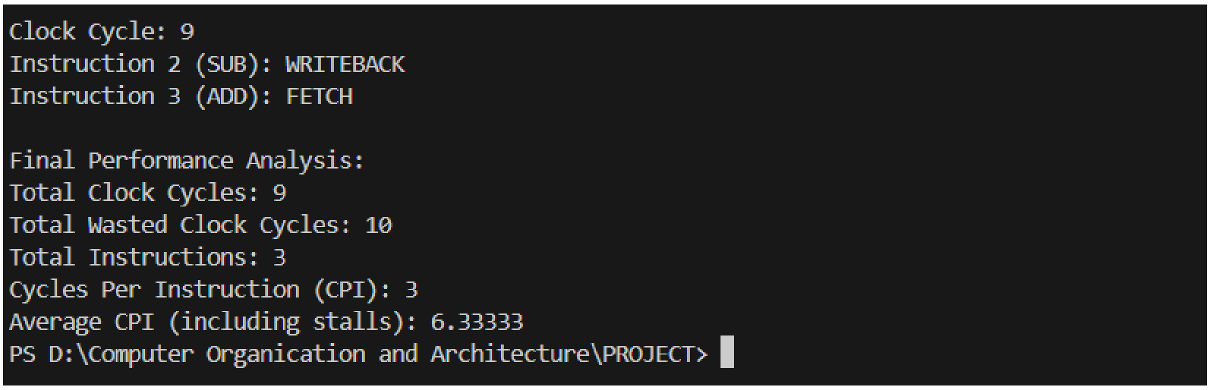
```
Clock Cycle: 5
Instruction 1 (DIV): WRITEBACK
Instruction 2 (SUB): FETCH
Instruction 3 (MUL): EXECUTE

Clock Cycle: 6
Instruction 2 (SUB): DECODE
Instruction 3 (MUL): MEMORY

Clock Cycle: 7
Instruction 2 (SUB): EXECUTE
Instruction 3 (MUL): WRITEBACK
Instruction 3 (MUL): WRITEBACK

Clock Cycle: 8
Instruction 2 (SUB): MEMORY

Clock Cycle: 9
Instruction 2 (SUB): WRITEBACK

Final Performance Analysis:
Total Clock Cycles: 9
Total Wasted Clock Cycles: 3
Total Instructions: 3
Cycles Per Instruction (CPI): 3
Average CPI (including stalls): 4
PS D:\Computer Organication and Architecture\PROJECT>
```



```
Enter the number of instructions: 3
Enter instruction 1 (e.g., ADD R1 R2 R3): MUL R1 R2 R3
Enter instruction 2 (e.g., ADD R1 R2 R3): SUB R3 R4 R5
Enter instruction 3 (e.g., ADD R1 R2 R3): ADD R7 R3 R8
Starting to push instructions into the Pipeline:

Clock Cycle: 0

Clock Cycle: 1
Instruction 1 (MUL): FETCH

Clock Cycle: 2
Instruction 1 (MUL): DECODE
Instruction 2 (SUB): FETCH
Write After Read Hazard detected between Instruction 1 and Instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.
Read After Write Hazard detected between instruction 2 and instruction 3. Adding a stall.
Stalling Instruction 3 due to dependency on Instruction 2.

Clock Cycle: 3
Instruction 1 (MUL): EXECUTE
Instruction 2 (SUB): FETCH
Write After Read Hazard detected between Instruction 1 and Instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.
Read After Write Hazard detected between instruction 2 and instruction 3. Adding a stall.
Stalling Instruction 3 due to dependency on Instruction 2.

Clock Cycle: 4
Instruction 1 (MUL): MEMORY
Instruction 2 (SUB): FETCH
Write After Read Hazard detected between Instruction 1 and Instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.
```

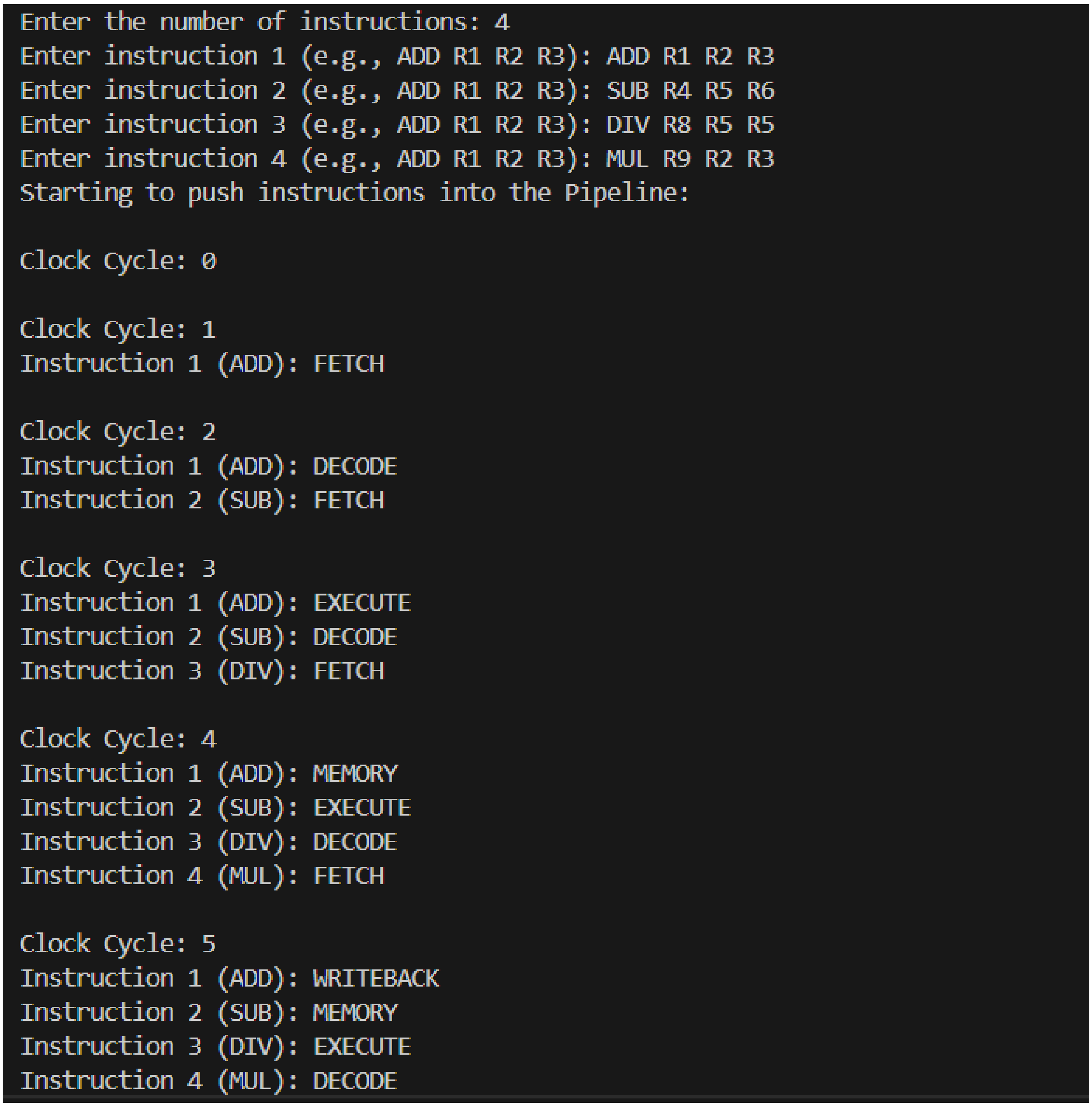


```
Instruction 2 (SUB): FETCH
Write After Read Hazard detected between Instruction 1 and Instruction 2. Adding a stall.
Stalling Instruction 2 due to dependency on Instruction 1.
Read After Write Hazard detected between instruction 2 and instruction 3. Adding a stall.
Stalling Instruction 3 due to dependency on Instruction 2.

Clock Cycle: 5
Instruction 1 (MUL): WRITEBACK
Instruction 2 (SUB): FETCH
Instruction 3 (ADD): FETCH
Read After Write Hazard detected between instruction 2 and instruction 3. Adding a stall.
Stalling Instruction 3 due to dependency on Instruction 2.

Clock Cycle: 6
Instruction 2 (SUB): DECODE
Instruction 3 (ADD): FETCH
Read After Write Hazard detected between instruction 2 and instruction 3. Adding a stall.
Stalling Instruction 3 due to dependency on Instruction 2.

Clock Cycle: 7
Instruction 2 (SUB): EXECUTE
Instruction 3 (ADD): FETCH
Read After Write Hazard detected between instruction 2 and instruction 3. Adding a stall.
Stalling Instruction 3 due to dependency on Instruction 2.

Clock Cycle: 8
Instruction 2 (SUB): MEMORY
Instruction 3 (ADD): FETCH
Read After Write Hazard detected between instruction 2 and instruction 3. Adding a stall.
Stalling Instruction 3 due to dependency on Instruction 2.

Clock Cycle: 9
```

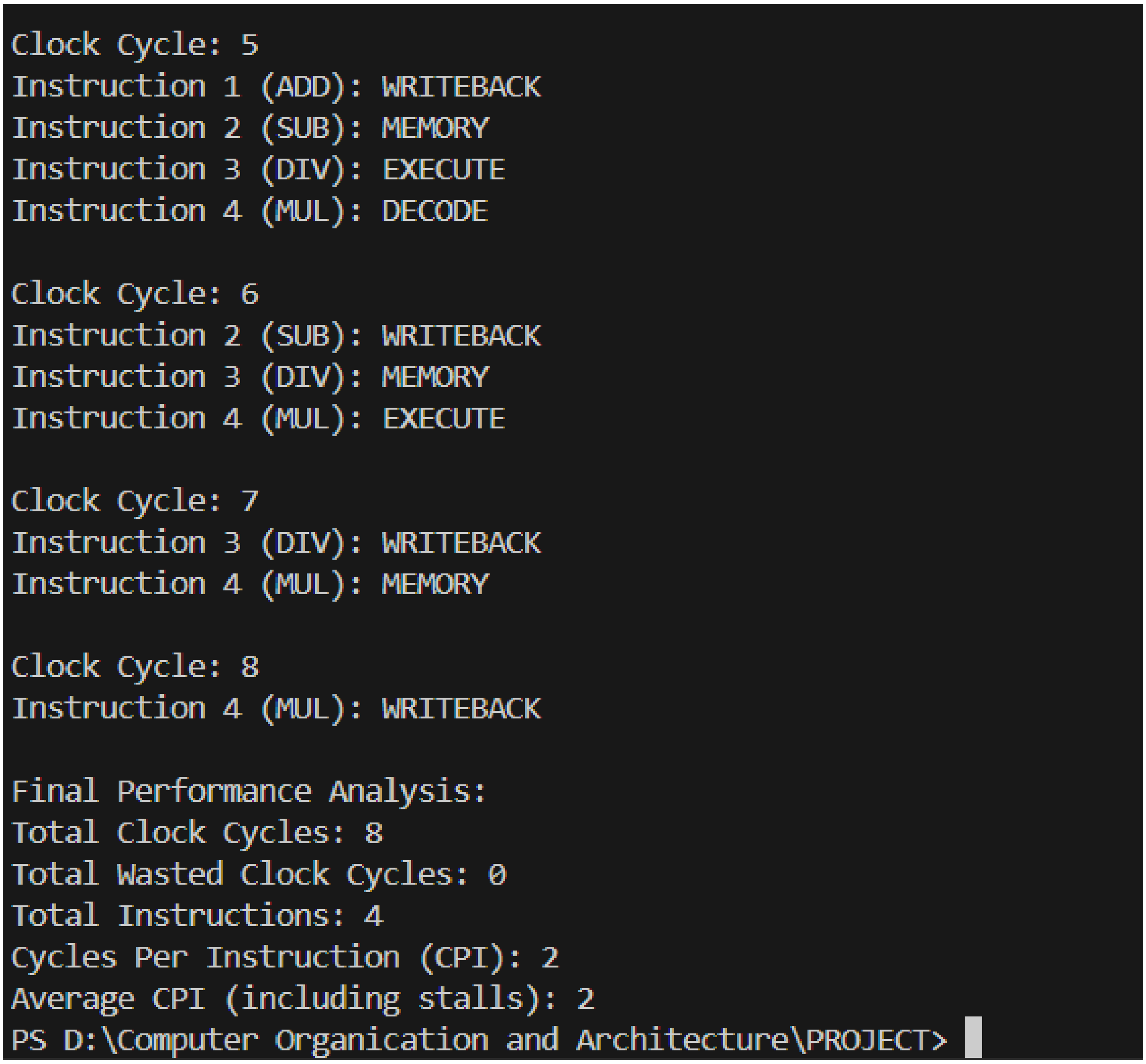


```
Clock Cycle: 9
Instruction 2 (SUB): WRITEBACK
Instruction 3 (ADD): FETCH

Final Performance Analysis:
Total Clock Cycles: 9
Total Wasted Clock Cycles: 10
Total Instructions: 3
Cycles Per Instruction (CPI): 3
Average CPI (including stalls): 6.33333
PS D:\Computer Organication and Architecture\PROJECT>
```

4 INSTRUCTIONS WITHOUT HAZARDS:



```
Enter the number of instructions: 4
Enter instruction 1 (e.g., ADD R1 R2 R3): ADD R1 R2 R3
Enter instruction 2 (e.g., ADD R1 R2 R3): SUB R4 R5 R6
Enter instruction 3 (e.g., ADD R1 R2 R3): DIV R8 R5 R5
Enter instruction 4 (e.g., ADD R1 R2 R3): MUL R9 R2 R3
Starting to push instructions into the Pipeline:

Clock Cycle: 0

Clock Cycle: 1
Instruction 1 (ADD): FETCH

Clock Cycle: 2
Instruction 1 (ADD): DECODE
Instruction 2 (SUB): FETCH

Clock Cycle: 3
Instruction 1 (ADD): EXECUTE
Instruction 2 (SUB): DECODE
Instruction 3 (DIV): FETCH

Clock Cycle: 4
Instruction 1 (ADD): MEMORY
Instruction 2 (SUB): EXECUTE
Instruction 3 (DIV): DECODE
Instruction 4 (MUL): FETCH

Clock Cycle: 5
Instruction 1 (ADD): WRITEBACK
Instruction 2 (SUB): MEMORY
Instruction 3 (DIV): EXECUTE
Instruction 4 (MUL): DECODE
```



```
Clock Cycle: 5
Instruction 1 (ADD): WRITEBACK
Instruction 2 (SUB): MEMORY
Instruction 3 (DIV): EXECUTE
Instruction 4 (MUL): DECODE

Clock Cycle: 6
Instruction 2 (SUB): WRITEBACK
Instruction 3 (DIV): MEMORY
Instruction 4 (MUL): EXECUTE

Clock Cycle: 7
Instruction 3 (DIV): WRITEBACK
Instruction 4 (MUL): MEMORY

Clock Cycle: 8
Instruction 4 (MUL): WRITEBACK

Final Performance Analysis:
Total Clock Cycles: 8
Total Wasted Clock Cycles: 0
Total Instructions: 4
Cycles Per Instruction (CPI): 2
Average CPI (including stalls): 2
PS D:\Computer Organication and Architecture\PROJECT>
```

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.