THE AMERICAN UNIVERSITY IN CAIRO

100 YEARS

CSCE3301 - Computer Architecture

# Tomasulo Algorithm Simulation

Names: Jana Elfeky, Kareem Elnaghy

Dr. Cherif Salama

## Introduction

This project aims to design and implement a simulator for a simplified 16-bit RISC processor using Tomasulo's algorithm with speculation. The simulator focuses on simulating an out-of-order execution pipeline while adhering to the Ridiculously Simple Computer (RiSC-16) instruction set. The primary objective is to assess the performance of the processor by recording key metrics, including instruction throughput, branch prediction accuracy, and cycle count.

The simulator supports the execution of various programs, handling instruction dependencies dynamically using reservation stations, a reorder buffer (ROB), and functional units for arithmetic, logic, branching, and memory operations. Additionally, speculative execution is implemented with an always-not-taken branch prediction strategy, allowing instructions to execute beyond predicted branches.

The implementation comprises modules for the instruction pipeline, memory, register file, Tomasulo's reservation stations, and functional units. Bonus features include a parser for directly reading assembly programs, as well as a configurable hardware setup that enables testing with varying numbers of reservation stations and ROB entries.

This report details the simulator's architecture, implementation strategy,  and results from testing diverse assembly programs.

## Bonuses

1. We implemented a parser in our simulator to allow users to input programs using proper assembly language, including labels for jump targets and function calls. The parser supports both direct input and reading from text files, handling label resolution, offset calculations, and instruction validation. This feature streamlines program testing by automating translation into the simulator's format, ensuring accuracy and usability while providing helpful error feedback for debugging.

2. We implemented support for variable hardware configuration, allowing users to customize the simulator's hardware setup. This includes specifying the number of reservation stations for each instruction class, the number of ROB entries, and the cycle latency for each functional unit type. This flexibility ensures the simulator can adapt to various use cases and performance testing scenarios, enhancing its versatility.

## Structure

The Tomasulo Simulator consists of several classes that simulate key components of Tomasulo's algorithm. These classes coordinate to execute instructions efficiently by leveraging reservation stations, functional units, and a reorder buffer for out-of-order execution and proper result handling.

1. **CommonDataBus**
   Facilitates communication between components by transmitting results and tags to ensure dependency resolution. It supports single and dual-value writes and manages a busy state to prevent conflicts.

2. **FunctionalUnit**
   Represents a processing unit (e.g., ADD, MUL) that performs arithmetic or logical operations. It tracks operation type, latency, and operand values and transitions through execution cycles until completion.

3. **Instruction**
   Encapsulates the details of a single instruction, including its operation, operands, and relevant execution cycle timings (issue, start execution, finish, write, commit).

4. **ReorderBuffer**
   Tracks instructions for in-order retirement after execution. It maintains tags, destinations, and values while ensuring data consistency and readiness for write-back.

5. **ReservationStation**
   Buffers instructions awaiting execution. It handles operand readiness, assigns functional units, and transitions through various states (issued, executing, writing, committing) for each instruction.

6. **TomasuloSimulator**
   The core class that manages the simulation. It handles instruction parsing, coordinates reservation stations, allocates functional units, and manages the reorder buffer for out-of-order execution. It ensures instructions are executed correctly with proper dependency resolution.

These modules interact dynamically to manage instruction flow, resolve data hazards, and maximize throughput in the Tomasulo algorithm's pipeline.

# User Guide

## Files and Input

### Input Instructions File

The instructions are read from a text file specified in the parseInstructions function. The file must contain a list of instructions, one per line. Each instruction should follow this format:

<operation> <destination> <source1> <source2> <offset>

Where the operation can be one of the following:

- **LOAD**: Loads data from memory.
- **STORE**: Stores data to memory.
- **ADDI**: Adds an immediate value to a register.
- **BEQ**: Conditional branch if equal.
- **CALL**: Procedure call.
- **RET**: Procedure return.
- **ADD**, **MUL**, **NAND**: Basic arithmetic operations.

Example:

```
LOAD R1,0(R2)
ADD R3,R1,R2
BEQ R1,R2,4
```

### User Input Configuration

The program asks the user for the number of reservation stations and latencies for each instruction type. These include:

- ADD/ADDI
- MUL
- NAND
- BEQ
- LOAD
- STORE
- CALL/RET

### Memory Setup

A predefined memory array is used for simulation, and initial values are placed at specific addresses in the memory (e.g., memory[0] = 10).

## Simulation Process

### 1. Instruction Parsing:

- The program parses the instructions from the file using the **parseInstructions** function. Each instruction is converted into an **Instruction** object.

### 2. User Configuration:

- The user inputs the number of reservation stations and latencies for each instruction class. This information helps set up the Tomasulo simulator's configuration.

### 3. Simulation:

- The simulator is initialized with the parsed instructions, memory, and configuration parameters. The simulation runs using the **simulate()** function of the **TomasuloSimulator** class, which processes the instructions and simulates their execution.

## Steps for Running the Simulation

### 1. Prepare Input File

Create a text file with a list of instructions in the format described above. Place this file in the same directory as the simulator or specify the full file path.

### 2. Set Up the Configuration

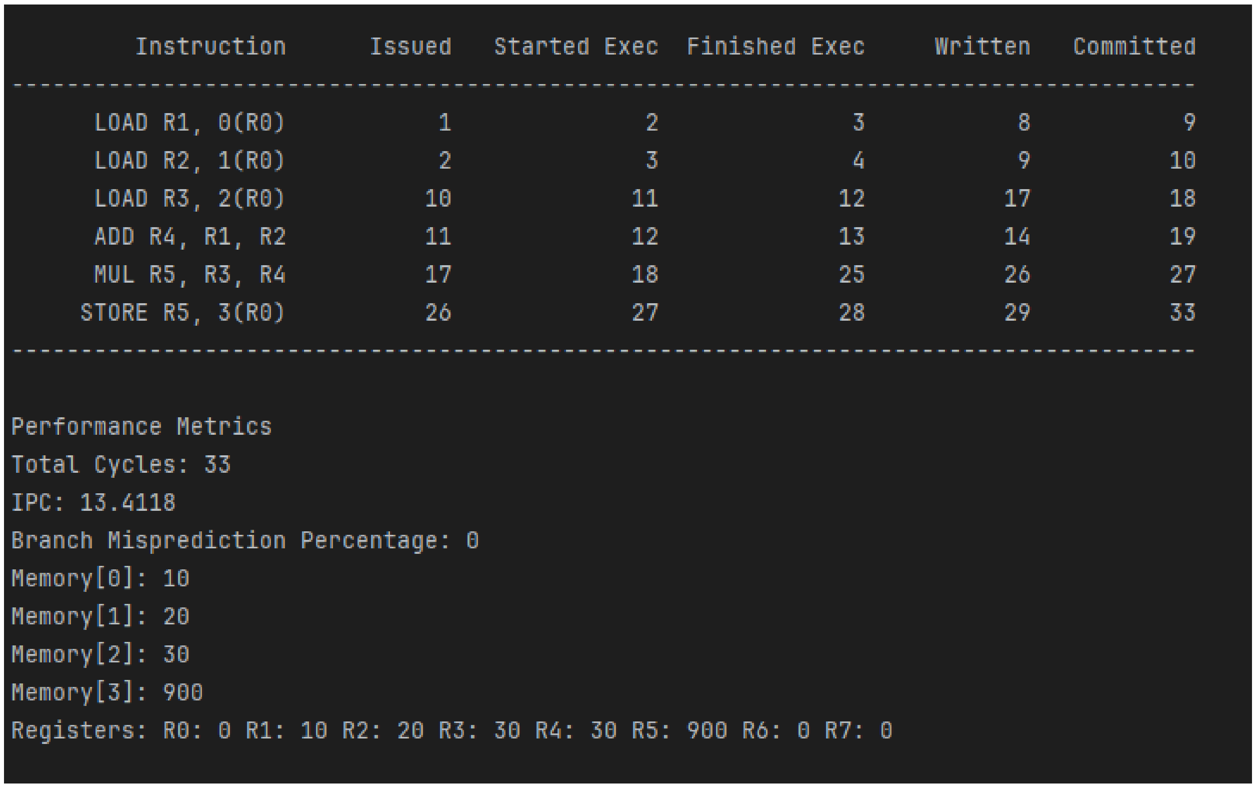
When running the simulator, the program will prompt you for the number of reservation stations and the latencies for each instruction type. Enter appropriate values based on your simulation needs.

## 3. Start the Simulation

Once the configuration is set, the simulator will start processing the instructions based on the Tomasulo algorithm. It will simulate the execution of each instruction, using the reservation stations, functional units, and memory.

## 4. View Output

The simulation will display detailed results and logs based on the **simulate()** function implementation. The output should look like the following:



```
         Instruction      Issued   Started Exec  Finished Exec     Written   Committed
--------------------------------------------------------------------------------------
      LOAD R1, 0(R0)           1              2              3           8           9
      LOAD R2, 1(R0)           2              3              4           9          10
      LOAD R3, 2(R0)          10             11             12          17          18
      ADD R4, R1, R2          11             12             13          14          19
      MUL R5, R3, R4          17             18             25          26          27
     STORE R5, 3(R0)          26             27             28          29          33
--------------------------------------------------------------------------------------


Performance Metrics
Total Cycles: 33
IPC: 13.4118
Branch Misprediction Percentage: 0
Memory[0]: 10
Memory[1]: 20
Memory[2]: 30
Memory[3]: 900
Registers: R0: 0 R1: 10 R2: 20 R3: 30 R4: 30 R5: 900 R6: 0 R7: 0
```

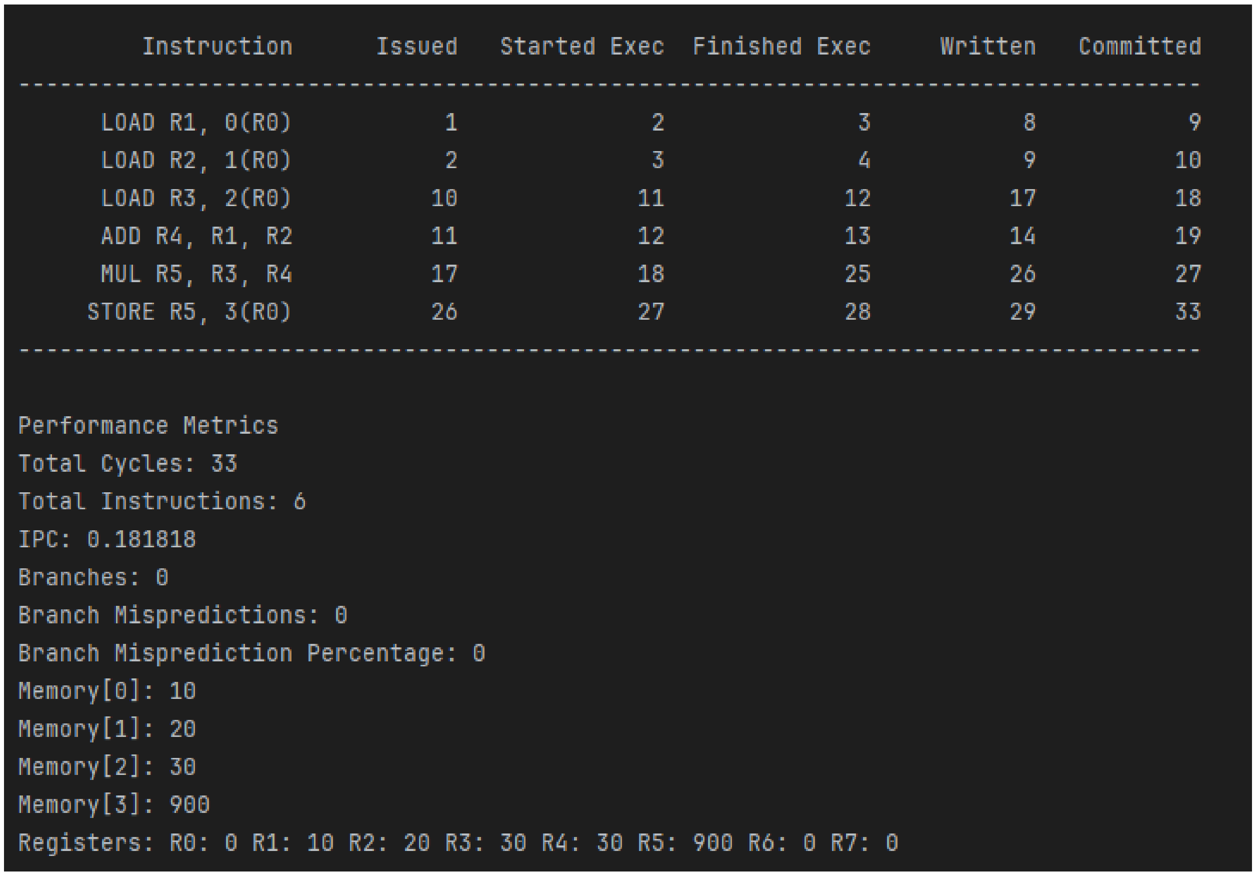
## Test Cases

### Test Case 1:

Input Program

```
LOAD R1, 0(R0)
LOAD R2, 1(R0)
LOAD R3, 2(R0)
ADD R4, R1, R2
MUL R5, R3, R4
STORE R5, 3(R0)
```

Output



```
        Instruction      Issued   Started Exec  Finished Exec     Written   Committed
-------------------------------------------------------------------------------------
      LOAD R1, 0(R0)          1             2              3           8           9
      LOAD R2, 1(R0)          2             3              4           9          10
      LOAD R3, 2(R0)         10            11             12          17          18
      ADD R4, R1, R2         11            12             13          14          19
      MUL R5, R3, R4         17            18             25          26          27
     STORE R5, 3(R0)         26            27             28          29          33
-------------------------------------------------------------------------------------


Performance Metrics
Total Cycles: 33
Total Instructions: 6
IPC: 0.181818
Branches: 0
Branch Mispredictions: 0
Branch Misprediction Percentage: 0
Memory[0]: 10
Memory[1]: 20
Memory[2]: 30
Memory[3]: 900
Registers: R0: 0 R1: 10 R2: 20 R3: 30 R4: 30 R5: 900 R6: 0 R7: 0
```

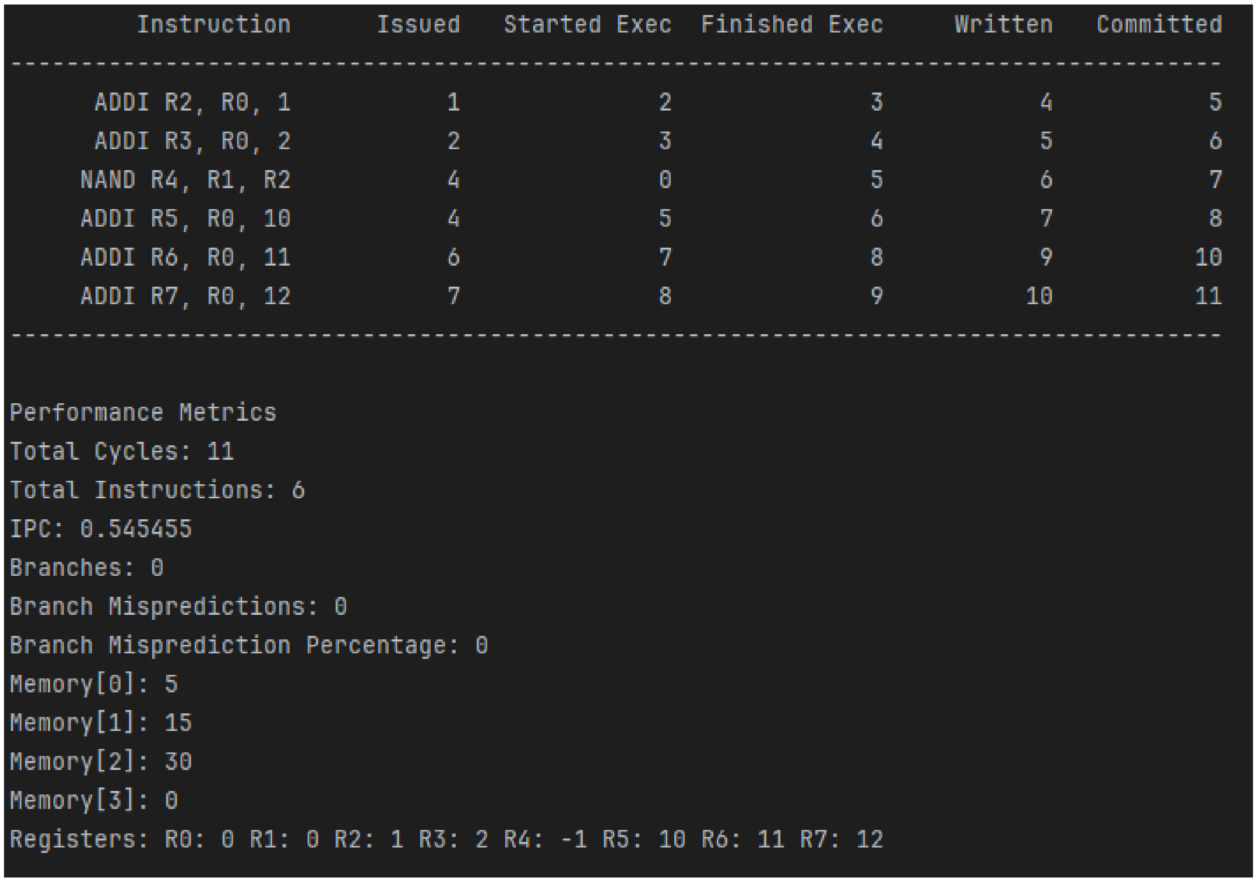
- Execution Timeline:
  - Instructions were issued sequentially (cycles 1 to 26) with out-of-order execution.
  - Dependencies were handled effectively (e.g., ADD waited for LOAD, MUL waited for ADD).
- Performance Metrics:
  - Total cycles: 33; IPC: ~0.18.
  - No branch mispredictions (0%), as no branching instructions were present.
- Memory and Register Updates:
  - Memory[3]: Correctly stored result 900 from MUL.
  - Registers: Updated accurately (R4 = 30, R5 = 900).

**Test Case 2:**

Input Program

```
ADDI R2, R0, 1
ADDI R3, R0, 2
NAND R4, R1, R2
ADDI R5, R0, 10
ADDI R6, R0, 11
ADDI R7, R0, 12
```

Output



```
        Instruction      Issued   Started Exec  Finished Exec     Written   Committed
-------------------------------------------------------------------------------------
      ADDI R2, R0, 1          1             2              3           4           5
      ADDI R3, R0, 2          2             3              4           5           6
     NAND R4, R1, R2          4             0              5           6           7
     ADDI R5, R0, 10          4             5              6           7           8
     ADDI R6, R0, 11          6             7              8           9          10
     ADDI R7, R0, 12          7             8              9          10          11
-------------------------------------------------------------------------------------

Performance Metrics
Total Cycles: 11
Total Instructions: 6
IPC: 0.545455
Branches: 0
Branch Mispredictions: 0
Branch Misprediction Percentage: 0
Memory[0]: 5
Memory[1]: 15
Memory[2]: 30
Memory[3]: 0
Registers: R0: 0 R1: 0 R2: 1 R3: 2 R4: -1 R5: 10 R6: 11 R7: 12
```

Execution Timeline:

- Instructions executed sequentially, with no pipeline stalls or delays.
- Dependent instruction (NAND) started after ADDI R2 and ADDI R3 completed, ensuring correct operand values.

Performance Metrics:

- Total cycles: 11; IPC: 0.55
- No branch mispredictions (0% misprediction rate).

Memory and Register States:

- Registers updated accurately:
  - R2 = 1, R3 = 2, R4 = -1 (result of NAND), R5 = 10, R6 = 11, R7 = 12.

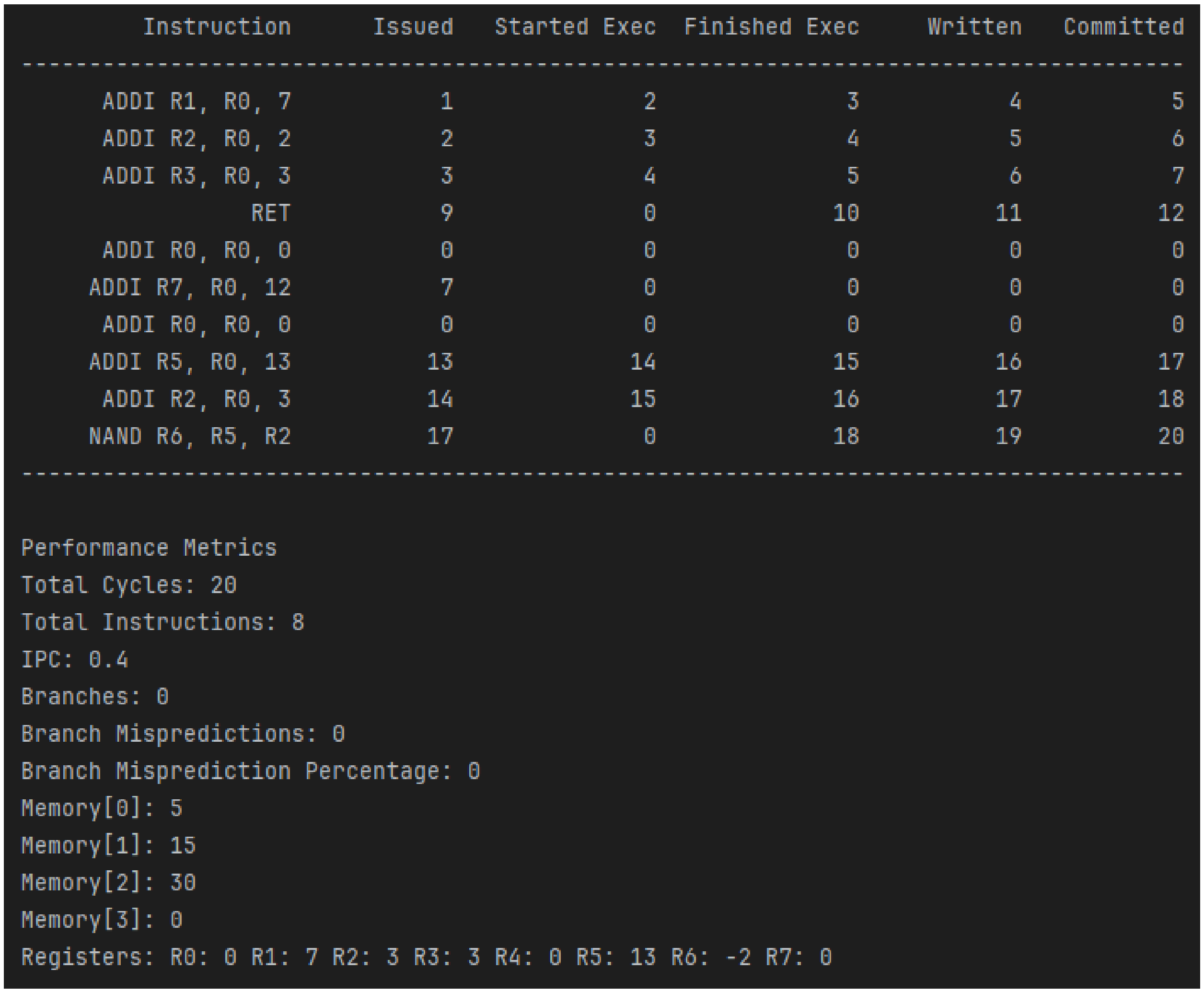
- Memory state remained unchanged, as no memory instructions were involved.

**Test Case 3:**

Input Program

```
ADDI R1, R0, 7
ADDI R2, R0, 2
ADDI R3, R0, 3
RET
ADDI R0, R0, 0
ADDI R7, R0, 12
ADDI R0, R0, 0
ADDI R5, R0, 13
ADDI R2, R0, 3
NAND R6, R5, R2
```

Output



```
          Instruction      Issued   Started Exec  Finished Exec     Written   Committed
---------------------------------------------------------------------------------------
       ADDI R1, R0, 7           1              2              3           4           5
       ADDI R2, R0, 2           2              3              4           5           6
       ADDI R3, R0, 3           3              4              5           6           7
                  RET           9              0             10          11          12
       ADDI R0, R0, 0           0              0              0           0           0
      ADDI R7, R0, 12           7              0              0           0           0
       ADDI R0, R0, 0           0              0              0           0           0
      ADDI R5, R0, 13          13             14             15          16          17
       ADDI R2, R0, 3          14             15             16          17          18
      NAND R6, R5, R2          17              0             18          19          20
---------------------------------------------------------------------------------------


Performance Metrics
Total Cycles: 20
Total Instructions: 8
IPC: 0.4
Branches: 0
Branch Mispredictions: 0
Branch Misprediction Percentage: 0
Memory[0]: 5
Memory[1]: 15
Memory[2]: 30
Memory[3]: 0
Registers: R0: 0 R1: 7 R2: 3 R3: 3 R4: 0 R5: 13 R6: -2 R7: 0
```

- Execution Timeline:
  - Instructions executed sequentially, with a RET function causing a jump that skips the ADDI instruction writing to R7.

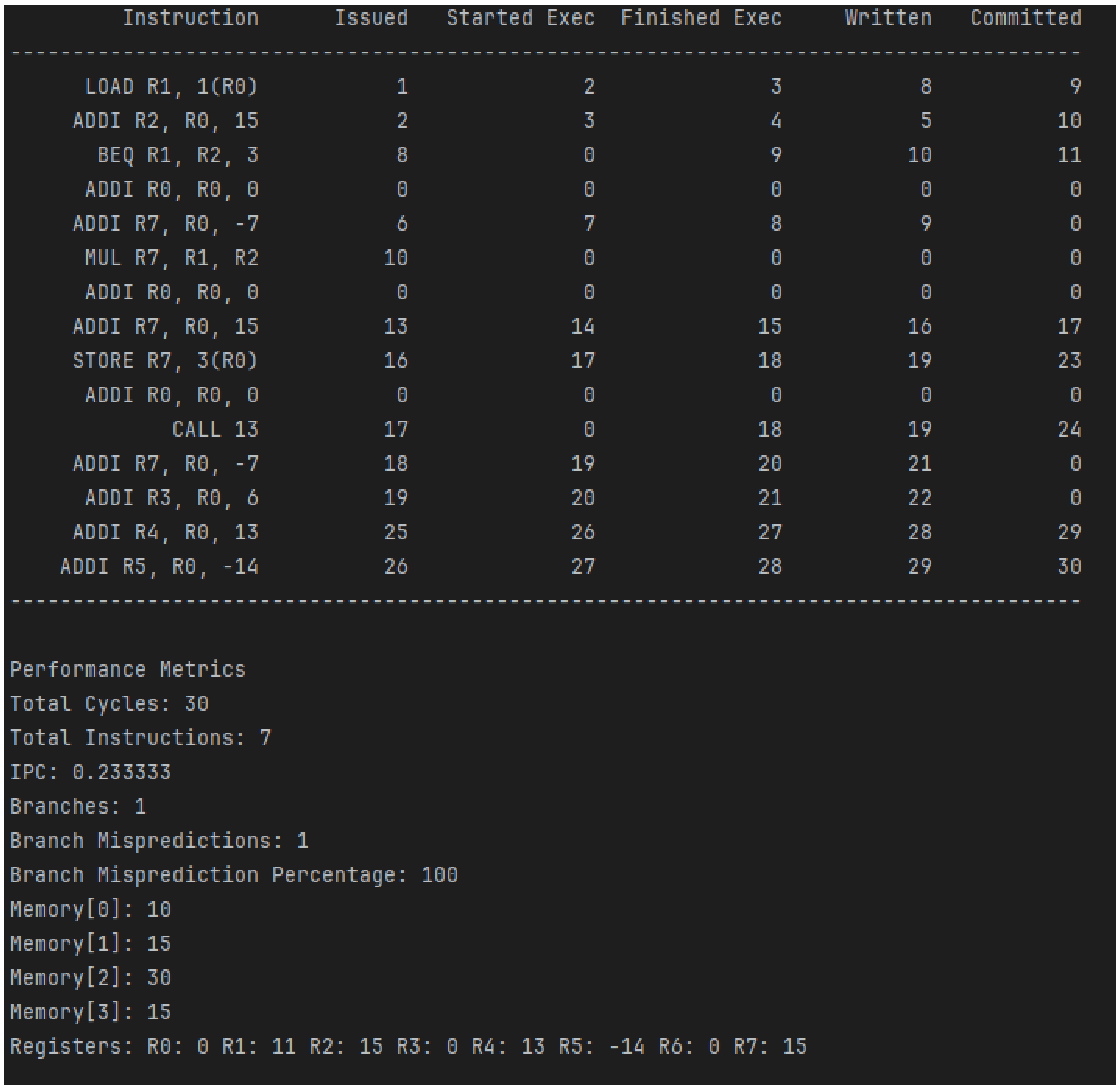
    - NOP operations (i.e. operations that write to R0 are not issued)
- Performance Metrics:
    - Total cycles: 20; IPC: 0.4
    - No branch mispredictions, as no branches were involved.
- Memory and Register States:
    - Registers updated as expected:
        - R1 = 7, R2 = 3 (overwritten), R3 = 3, R5 = 13, R6 = -2, R7 = 0
    - Memory values remain unchanged due to the absence of memory operations.

**Test Case 4:**

Input Program

```
LOAD R1, 1(R0)
ADDI R2, R0, 15
BEQ R1, R2, 3
ADDO R0, R0, 0
ADDI R7, R0, -7
MUL R7, R1, R2
ADDI R0, R0, 0
ADDI R7, R0, 15
STORE R7, 3(R0)
ADDI R0, R0, 0
CALL 13
ADDI R7, R0, -7
ADDI R3, R0, 6
ADDI R4, R0, 13
ADDI R5, R0, -14
```

Output



```
          Instruction      Issued   Started Exec  Finished Exec     Written   Committed
-------------------------------------------------------------------------------------
       LOAD R1, 1(R0)           1              2              3           8           9
      ADDI R2, R0, 15           2              3              4           5          10
        BEQ R1, R2, 3           8              0              9          10          11
       ADDI R0, R0, 0           0              0              0           0           0
      ADDI R7, R0, -7           6              7              8           9           0
       MUL R7, R1, R2          10              0              0           0           0
       ADDI R0, R0, 0           0              0              0           0           0
      ADDI R7, R0, 15          13             14             15          16          17
      STORE R7, 3(R0)          16             17             18          19          23
       ADDI R0, R0, 0           0              0              0           0           0
              CALL 13          17              0             18          19          24
      ADDI R7, R0, -7          18             19             20          21           0
       ADDI R3, R0, 6          19             20             21          22           0
      ADDI R4, R0, 13          25             26             27          28          29
     ADDI R5, R0, -14          26             27             28          29          30
-------------------------------------------------------------------------------------


Performance Metrics
Total Cycles: 30
Total Instructions: 7
IPC: 0.233333
Branches: 1
Branch Mispredictions: 1
Branch Misprediction Percentage: 100
Memory[0]: 10
Memory[1]: 15
Memory[2]: 30
Memory[3]: 15
Registers: R0: 0 R1: 11 R2: 15 R3: 0 R4: 13 R5: -14 R6: 0 R7: 15
```

- Execution Timeline:
  - We load values to registers through LOAD and ADDI.
  - Then a BEQ is taken which skips a few instructions and jumps to PC = 6, which is the NOP Instruction.
  - Then the next instructions execute until the CALL function which will store the value of PC+1 in R1 and then jump to PC = 13.
  - Then finally the final instruction is the only instruction executed due to the jump.
- Performance Metrics:
  - Total cycles: 30; IPC: 0.233, reflecting moderate efficiency.
  - 1 Branch was taken and thus we have a 100 Percent branch prediction percentange.
- Memory and Register States:
  - Registers updated as expected
  - Memory[3] = 15 due to the final STORE instruction.