Project title: ***Tomasulo Algorithm Simulation***

Course: CSCE 3301 – Computer Architecture

Semester: Fall 2025

Mohamed Ouail Slama & Yahia Kilany

Cherif Salama

December 09th, 2025

1. Introduction

Computer architecture relies on sophisticated mechanisms to exploit instruction-level parallelism and improve processor performance. Tomasulo's algorithm is a foundational dynamic scheduling technique that enables out-of-order execution while managing data hazards and hardware resource conflicts. In this project, we design and implement femTomas, a cycle-accurate simulator of a simplified 16-bit RISC processor that models Tomasulo's algorithm with speculative execution.

The goal of the simulator is to execute a given assembly program on a configurable hardware model, track the progression of each instruction through reservation stations, functional units, and a reorder buffer, and produce key performance metrics such as total execution cycles, instructions per cycle (IPC), and branch misprediction rate. The system supports a restricted instruction set, a single-issue pipeline, and a fixed set of functional units with prescribed latencies, as detailed in the project specification.

2. System Overview

This section provides a consolidated summary of the project specifications as outlined in the assignment handout. Our simulator is designed to implement a functional model of a 16-bit RISC processor employing Tomasulo's algorithm with speculative execution. The system is constrained to a single-issue pipeline and models only the four backend stages: issue, execute, write, and commit. All front-end stages (fetch and decode) are assumed to have already completed, and the instruction queue is pre-filled at the start of simulation.

The processor supports a simplified RISC instruction set including load/store, conditional branch, call/return, and arithmetic/logic operations. Memory is word-addressable with a 16-bit address space (128 KB capacity), and the register file consists of eight 16-bit general-purpose registers (R0–R7), with R0 hardwired to zero. The hardware configuration includes a fixed set of reservation stations and functional units as specified: two load units (6 cycles each), one store unit (6 cycles), two BEQ units (1 cycle), one CALL/RET unit (1 cycle), four ADD/SUB units (2 cycles each), two NAND units (1 cycle each), and one MUL unit (12 cycles). The reorder buffer (ROB) contains 8 entries, and branch prediction follows an always-not-taken policy. Several simplifying assumptions are made, including the absence of caches, interrupts, exceptions, and I/O operations, as well as a one-to-one mapping between reservation stations and functional units. The simulator records performance metrics such as total cycles, instructions per cycle (IPC), and branch misprediction percentage, and it outputs a detailed timeline of each instruction's progression through the pipeline.

3. Design & Implementation

3.1 Code Structure

The simulator is architecturally divided into distinct, modular components that closely mirror the hardware elements of a Tomasulo-based processor. The implementation is written in C++ and adopts an object-oriented design, centered around a main class named TomasuroSimulator. This class serves as the heart of the simulation, containing all the processor's dynamic state and the logic for its pipeline stages.

At the foundation are several key **data structures** defined as C++ structs. The HardwareConfig struct aggregates all user-configurable parameters, such as the number of reservation stations per functional unit type, the size of the reorder buffer, and the execution latency for each instruction class. The Instruction struct captures the static properties of an assembly instruction—its type, operand registers, immediate values, and its original text for display. To track the progress of each instruction through the pipeline, the IssuedInstruction struct records cycle-accurate timestamps for its issue, execution start and end, write-back, and commit events, along with a flag indicating if it was flushed due to a misprediction.

Modeling the processor's internal units are the ReservationStation and ROBEntry structs. A ReservationStation entry holds the state of an instruction awaiting execution, including its operands (either as ready values or tags pointing to producing ROB entries), its destination, any calculated memory address, and a countdown of remaining execution cycles. A ROBEntry represents a slot in the reorder buffer, storing the instruction's result, its readiness, its architectural destination (register or memory address), and metadata for branch resolution.

Within the TomasuroSimulator class, these structures are instantiated as collections. Reservation stations are organized into separate vectors—one for each instruction type (e.g., load_rs, store_rs, addsub_rs). The reorder buffer is a vector of ROBEntry managed as a circular queue. The architectural register file is a simple integer array, accompanied by a register status table that tracks which ROB entry will produce the next value for each register. Memory is implemented as a map from address to value, allowing sparse initialization. A separate map stores the program's instructions keyed by their program counter for efficient fetching.

The class's public interface is straightforward, primarily consisting of a run() method that drives the simulation cycle-by-cycle. Internally, each cycle is processed by four core pipeline methods invoked in reverse order: commit(), write(), execute(), and issue().

Supporting these are helper functions like broadcastResult(), which simulates the common data bus by forwarding completed results to waiting reservation stations, and flush(), which handles the cleanup and redirection required on a branch misprediction.

Input and output are handled by auxiliary functions. The parse() function reads the structured input file, which may include an optional hardware configuration section, a starting address, the assembly program, and initial memory values. It returns a ParsedInput struct containing the fully initialized configuration and program. The printResults() method generates the required output, including a detailed table of each instruction's timing and overall performance metrics like total cycles, IPC, and branch misprediction rate.

3.2 Execution Flow and Algorithms

The simulation executes programs through a cycle-accurate model of the Tomasulo pipeline, processing instructions in four stages that run in reverse order each cycle: **Commit, Write, Execute, and Issue.** This reverse ordering ensures that results from earlier stages in the same cycle are available to later stages, accurately modeling hardware timing where writes can affect issues in the same clock edge.

Commit Stage

The commit stage begins each cycle by examining the head of the Reorder Buffer (ROB). If the ROB entry is marked as ready, the instruction is retired in program order. For arithmetic and load instructions, the result value is written to the architectural register file (unless the destination is R0, which is hardwired to zero). For store instructions, the value is written to memory at the previously calculated address. When a branch instruction is committed, its actual outcome (taken or not taken) is compared against the predictor's "always not taken" assumption. A misprediction triggers a pipeline flush: all instructions younger than the branch are invalidated, reservation stations are cleared, the ROB tail is rolled back, and the program counter is redirected to the correct target address. Call and return instructions also cause a flush and PC update upon commit.

Write Stage

During the write stage, reservation stations that have finished execution broadcast their results on the Common Data Bus (CDB). This is simulated by iterating through each reservation station bank. If a station has completed its execution cycles and is not flushed, its result value is marked as ready in the corresponding ROB entry. The result is then propagated to all other reservation stations: any station waiting for this result (indicated by matching Qj or Qk tags) receives the value, and its dependency tag is

cleared. This broadcast mechanism resolves data dependencies dynamically, allowing dependent instructions to begin execution as soon as operands become available.

Execute Stage

The execute stage processes all active reservation stations. For each station that is busy but not yet executing, the algorithm checks if its source operands are ready (both Qj and Qk are -1). Load instructions undergo an additional memory disambiguation check: they must wait for all older store instructions to have known, non-conflicting addresses. Once operands are ready and any memory hazards are resolved, execution begins. The station's cycles_remaining counter is decremented each cycle. When the counter reaches zero, the instruction's result is computed based on its type—arithmetic operations perform the corresponding calculation, branches compare register values, and memory instructions compute addresses. The result is stored in the associated ROB entry, and the station is flagged as execution-done, pending write in the next cycle.

Issue Stage

The issue stage attempts to fetch and issue the next instruction from the program counter (PC). It first checks if the ROB has a free entry and if a reservation station of the appropriate type is available. If both resources are available, a new ROB entry is allocated at the tail pointer, and a reservation station is populated. Source operand values are read from the register file if available; otherwise, dependency tags (Qj, Qk) are set to point to the ROB entry that will produce the needed value. The destination register's status is updated to point to this new ROB entry. The PC is then incremented by one, reflecting the "always not taken" branch prediction where control flow continues sequentially. If any resource is unavailable, issue stalls for that cycle.

Cycle Loop and Termination

These four stages are repeated each cycle within the main run() loop. The simulation terminates when all instructions have been committed and no active entries remain in the ROB or reservation stations, or when a safety limit of cycles is exceeded to prevent infinite loops in faulty programs. Throughout execution, the simulator maintains detailed timing records for each instruction instance and aggregates performance statistics, including total cycles, instructions committed, and branch misprediction counts, which

are reported at the end of the simulation.

3.3 Bonus Features

A significant enhancement to the base simulator is the implementation of fully customizable hardware configuration, which allows users to modify the microarchitectural parameters of the simulated processor. This feature transforms the simulator from a fixed model into a flexible experimental platform, enabling detailed analysis of how various hardware resources impact performance. Users can adjust the number of reservation stations per functional unit, change the size of the reorder buffer, and modify execution latencies for different instruction types, all through a simple structured input format.

The customization is seamlessly integrated into the simulator's input parsing system. At the beginning of an input file, users may optionally include a CONFIG section containing key-value pairs that specify hardware parameters. These parameters include reservation station counts for each instruction class—such as LOAD_RS, STORE_RS, BEQ_RS, ADDSUB_RS, and others—as well as the number of ROB entries and the execution cycle counts for all supported operations. If the configuration section is omitted, the simulator gracefully defaults to the original project specifications, ensuring backward compatibility with standard test cases.

Internally, the configuration data is captured in a dedicated HardwareConfig structure, which is passed to the simulator during initialization. The simulator uses these values to dynamically resize its internal data structures: reservation station vectors are allocated according to the specified counts, the ROB is sized accordingly, and execution latency maps are populated with the provided cycle values. This design ensures that the entire pipeline—from issue logic to commit timing—adapts to the custom hardware model without requiring code changes or recompilation.

4. AI Usage Documentation

Throughout the development of the simulator, we leveraged AI tools—primarily Perplexity AI—as a collaborative aid to streamline certain aspects of the project. Our use was focused on problem-solving and refinement rather than on generating core simulation logic. The AI served as a debugging assistant, a brainstorming partner, and a documentation aid, while all final design decisions, implementation, and validation were carried out manually by our team.

We engaged the AI with prompts such as "What's wrong with my code?" when encountering runtime errors or logical inconsistencies in the pipeline; "Write me a

parsing function that turns the following input format into instructions stored in memory," which helped bootstrap the input module; "Generate an outline for the project report," used to structure our documentation; and "Proof-read/rewrite it in a better way," applied to improve the clarity and professionalism of our written sections. We also used the AI to generate sample assembly programs that cover all instruction types and include loops, which we then verified and adapted to our simulator's syntax for testing purposes.

Our workflow was iterative and human-centric. We began by asking the AI for high-level suggestions on project structure and design, then adapted and expanded those ideas with our own architectural understanding. When stuck on specific issues—such as incorrect ROB management or reservation station tagging—we pasted relevant code snippets and asked the AI to identify potential problems. Each suggestion was carefully reviewed, tested, and integrated only after we fully understood its operation and implications. For the written report, the AI helped polish paragraphs and ensure consistent technical phrasing, but the content and technical accuracy remained entirely our own.

5. User Guide

This guide explains the input format for the simulator so that users can create their own test files. Input is provided via a plain text file structured into four main sections: an optional hardware configuration, the program starting address, the list of assembly instructions, and initial memory values. All sections must appear in this order.

5.1 Input Format

The input file may optionally begin with a CONFIG section that allows customization of the processor's hardware parameters. This section starts with the line CONFIG and ends with END_CONFIG. Between these lines, users can specify key-value pairs to set the number of reservation stations for each functional unit type, the size of the reorder buffer, and the execution latency for each instruction class. Any parameter not specified will use its default value. For example:

```
text
CONFIG
ROB_ENTRIES 12
LOAD_RS 4
MUL_CYCLES 8
END_CONFIG
```

If the CONFIG section is omitted entirely, all hardware parameters are set to the default values given in the project specification.

Following the configuration (or directly at the start of the file if no configuration is used), the starting address of the program must be provided as a single integer on its own line. This address determines where the first instruction will be placed in the simulated memory.

The next section contains the assembly instructions, one per line, written in the supported simplified RISC assembly syntax. Valid instructions include LOAD, STORE, ADD, SUB, NAND, MUL, BEQ, CALL, and RET. Each line should follow the exact format shown in the project description—for instance, LOAD R1, 0(R2) or BEQ R1, R2, -5. The instruction block is terminated by a line containing only the word END.

Finally, the memory initialization section consists of lines each containing an address and a 16-bit value, separated by a space. These lines set the initial contents of memory before execution begins. The list is terminated by a line containing -1 -1. Only addresses that are explicitly initialized or later written by the program will appear in the final memory dump.

Example File

```
CONFIG
ROB_ENTRIES 10
LOAD_RS 3
STORE_RS 2
MUL_CYCLES 10
END_CONFIG
0
LOAD R1, 0(R0)
LOAD R2, 1(R0)
ADD R3, R1, R2
STORE R3, 2(R0)
END
0 100
1 200
-1 -1
```

When the simulator is executed, the user chooses between manual input and file input. If file input is selected, the program reads the specified text file, configures the hardware accordingly, loads the instructions and memory, and runs the simulation. The output includes a detailed cycle-by-cycle timing table for each instruction, overall performance metrics (total cycles, IPC, branch statistics), and the final state of registers and memory. This format allows users to easily create and test a wide variety of programs and

hardware configurations, facilitating exploration of Tomasulo's algorithm under different microarchitectural parameters.

5.2 Program Walkthrough

The program loads two values from memory (addresses 0 and 1), calls a subroutine at PC 6, executes a branch, performs arithmetic, and returns. Below is the assembly listing with corresponding PC values:

```
PC 0: LOAD R1, 0(R0)     ; R1 = Mem[0] = 10
PC 1: LOAD R2, 1(R0)     ; R2 = Mem[1] = 20
PC 2: CALL 6             ; Jump to PC 6, store return address in R1
PC 3: BEQ R0, R0, 4      ; Always taken branch (R0 == R0)
PC 4: ADD R4, R3, R1     ; R4 = R3 + R1
PC 5: STORE R4, 3(R0)    ; Mem[3] = R4
PC 6: ADD R3, R1, R2     ; R3 = R1 + R2 = 30
PC 7: RET                ; Return to address in R1
PC 8: ADD R3, R1, R2     ; R3 = R1 + R2 (post-return)
```

Memory is initialized with Mem[0]=10 and Mem[1]=20.

Simulation Output Analysis

```
=== Hardware Configuration ===
ROB Entries: 8
Reservation Stations:
  LOAD: 2
  STORE: 1
  BEQ: 2
  CALL/RET: 1
  ADD/SUB: 4
  NAND: 2
  MUL: 1
```

The simulator output begins by displaying the loaded hardware configuration—default values with 8 ROB entries, 2 load reservation stations, 1 CALL/RET station, etc.

```
=== RUNNING SIMULATION ===
  [Cycle 11] Misprediction! Flushing, redirecting to PC 6
  [Cycle 16] Misprediction! Flushing, redirecting to PC 3
  [Cycle 19] Misprediction! Flushing, redirecting to PC 8
```

During execution, three misprediction flushes occur, triggered by the CALL at PC 2, the BEQ at PC 3, and the RET at PC 7.

```
=== SIMULATION RESULTS ===

Instruction Timing Table (All Issued Instances):
  PC                  Instruction     #   Issue   ExecS   ExecE   Write  Commit    Status
-------------------------------------------------------------------------------------------
   0               LOAD R1, 0(R0)     0       1       2       7       8       9        OK
   1               LOAD R2, 1(R0)     0       2       3       8       9      10        OK
   2                      CALL 6     0       3       4       4       5      11        OK
   3                BEQ R0, R0, 4     0       4       5       5       6      -1   FLUSHED
   4               ADD R4, R3, R1     0       5       6       7       8      -1   FLUSHED
   5              STORE R4, 3(R0)     0       6       8      -1      -1      -1   FLUSHED
   6               ADD R3, R1, R2     0       7       9      10      -1      -1   FLUSHED
   7                          RET     0       8       9       9      10      -1   FLUSHED
   8               ADD R3, R1, R2     0       9      10      -1      -1      -1   FLUSHED
   6               ADD R3, R1, R2     1      11      12      13      14      15        OK
   7                          RET     1      12      13      13      14      16        OK
   8               ADD R3, R1, R2     1      13      14      15      -1      -1   FLUSHED
   3                BEQ R0, R0, 4     1      16      17      17      18      19        OK
   4               ADD R4, R3, R1     1      17      18      -1      -1      -1   FLUSHED
   5              STORE R4, 3(R0)     1      18      -1      -1      -1      -1   FLUSHED
   8               ADD R3, R1, R2     2      19      20      21      22      23        OK
```

The Instruction Timing Table shows all 16 issued instruction instances (some instructions are issued multiple times due to flushes). Each row includes the PC, instruction text, instance number, and cycle times for issue (Iss), execution start (ExS), execution end (ExE), write (Wr), and commit (Cm), along with a status (OK or FLUSHED).

Performance Metrics

```
=== PERFORMANCE METRICS ===
Total execution time: 24 cycles
Instructions issued: 16
Instructions committed: 7
IPC: 0.292
Branches encountered: 2
Branch mispredictions: 1
Branch misprediction rate: 50.00%

=== FINAL REGISTER STATE ===
R0: 0  R1: 3  R2: 20  R3: 23
R4: 0  R5: 0  R6: 0  R7: 0

=== FINAL MEMORY STATE ===
Mem[0]: 10
Mem[1]: 20
```

The simulation completed in 24 cycles, with 16 instructions issued but only 7 committed, yielding an IPC of 0.292. Two branches were encountered (the CALL is treated as a control-flow instruction, and the BEQ is a conditional branch), with one misprediction (the BEQ), giving a branch misprediction rate of 50%. The final register state shows R1=10, R2=20, R3=30 (result of the final ADD), and other registers unchanged. Memory remains as initialized, since the store instruction was flushed and never committed.

6. Test Programs

6.1 Test Case 1: Hardware Configuration Customization

To verify that the hardware customization feature functions correctly, we created a dedicated test file, hardware_config_test.txt. This program includes a mix of instruction types and is executed under a modified hardware configuration that differs from the project defaults. The purpose of this test is to ensure that the simulator properly reads and applies custom hardware parameters, and that execution behavior adjusts accordingly.

Program Description:

This test program loads three values from memory, performs a multiplication, two arithmetic operations, a conditional branch, a load, a NAND operation, and a store. It is designed to exercise multiple functional units and to stress the reservation station and ROB allocation under custom limits.

```
0: LOAD R1, 0(R0)   ; R1 = Mem[0] = 5
1: LOAD R2, 1(R0)   ; R2 = Mem[1] = 3
2: LOAD R3, 2(R0)   ; R3 = Mem[2] = 1
3: MUL R4, R1, R2   ; R4 = 5 * 3 = 15
4: ADD R1, R1, R3   ; R1 = 5 + 1 = 6
5: SUB R2, R2, R3   ; R2 = 3 - 1 = 2
6: BEQ R2, R0, 2    ; Branch not taken (R2 ≠ 0)
7: ADD R5, R4, R1   ; R5 = 15 + 6 = 21
8: STORE R5, 3(R0)  ; Mem[3] = 21
9: LOAD R6, 3(R0)   ; R6 = Mem[3] = 21
10: NAND R7, R6, R3 ; R7 = ~(21 & 1) = 0xFFFE
```

Initial Memory Values:
Address 0: 5
Address 1: 3
Address 2: 1

Custom Hardware Configuration:
The test uses a CONFIG section that increases capacity and reduces latencies relative to the defaults:

- ➔ ROB increased to 16 entries
- ➔ Reservation stations increased for all units (e.g., 3 LOAD stations, 6 ADD/SUB stations)
- ➔ Load/store latency reduced from 6 to 4 cycles
- ➔ Multiplication latency reduced from 12 to 8 cycles
- ➔ Most ALU operations set to 1 cycle

Purpose:
This test validates that:

- ➔ The parser correctly reads all CONFIG parameters.
- ➔ The simulator resizes internal structures (ROB, RS vectors) accordingly.
- ➔ Execution latencies are updated (e.g., MUL finishes earlier).
- ➔ Increased resources allow more instructions to be in flight simultaneously, potentially improving performance.

Output Summary:

```
=== Hardware Configuration ===
ROB Entries: 16
Reservation Stations:
  LOAD: 3
  STORE: 2
  BEQ: 3
  CALL/RET: 2
  ADD/SUB: 6
  NAND: 3
  MUL: 2
```

The simulation completes successfully, showing adjusted cycle counts in the timing table that reflect the reduced latencies. No structural hazards occur despite the longer instruction sequence, confirming that the increased reservation station counts are effective. The final register and memory states match the expected values computed from the program logic.

6.2 Test Case 2: all instructions

Input **:**

```
0
LOAD R1, 0(R0)
LOAD R2, 4(R0)
ADD R3, R1, R2
SUB R4, R2, R1
NAND R5, R1, R2
MUL R6, R1, R2
STORE R3, 8(R0)
STORE R4, 12(R0)
STORE R5, 16(R0)
STORE R6, 20(R0)
END
0 3
4 7
8 0
12 0
16 0
20 0
```

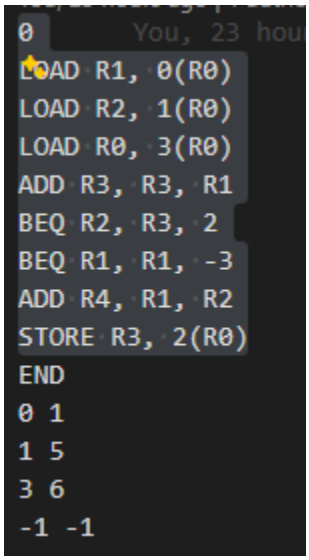
Output :

```
Choose input method:
1. Manual input
2. Read from file
Enter choice (1 or 2): 2
Enter filename: test_all_instructions.txt
Successfully loaded 10 instructions from file.

=== RUNNING SIMULATION ===

=== SIMULATION RESULTS ===

Instruction Timing Table (All Issued Instances):
  PC                  Instruction     #   Issue   ExecS   ExecE   Write  Commit    Status
-------------------------------------------------------------------------------------------
   0                LOAD R1, 0(R0)     0       1       2       7       8       9        OK
   1                LOAD R2, 4(R0)     0       2       3       8       9      10        OK
   2                ADD R3, R1, R2     0       3       9      10      11      12        OK
   3                SUB R4, R2, R1     0       4       9      10      11      13        OK
   4               NAND R5, R1, R2     0       5       9       9      10      14        OK
   5                MUL R6, R1, R2     0       6       9      20      21      22        OK
   6               STORE R3, 8(R0)     0       7      11      16      17      23        OK
   7              STORE R4, 12(R0)     0      17      18      23      24      25        OK
   8              STORE R5, 16(R0)     0      24      25      30      31      32        OK
   9              STORE R6, 20(R0)     0      31      32      37      38      39        OK

=== PERFORMANCE METRICS ===
Total execution time: 40 cycles
Instructions issued: 10
Instructions issued: 10
Instructions committed: 10
IPC: 0.250
Branches encountered: 0
Branch mispredictions: 0

=== FINAL REGISTER STATE ===
R0: 0  R1: 3  R2: 7  R3: 10
R4: 4  R5: 65532  R6: 21  R7: 0

=== FINAL MEMORY STATE ===
Mem[0]: 3
Mem[4]: 7
Mem[8]: 10
Mem[12]: 4
Mem[16]: 65532
Mem[20]: 21
```

Description :

This test case is designed to verify the correct functionality of all implemented instruction types. The initial register values are R1 = 3 and R2 = 7, and a sequence of arithmetic and logical operations is applied to them. The resulting values are written into R3, R4, R5, and R6, and eventually stored to memory, confirming correct dataflow and execution ordering. The results also highlight the behavior of the Reorder Buffer (ROB): although the store to address 8 (STORE R3, 8(R0)) completes its write at cycle 17, it does not commit until cycle 23 because the ROB enforces *in-order commit*, requiring the older instruction (MUL R6, R1, R2) to commit first at cycle 22. Additionally, the test exposes the multiple store instructions experience long issue-stage stalls, waiting for the store RS to become free.

6.3 Test Case 3 Simple loop:



Output:

```
Instruction Timing Table (All Issued Instances):
  PC                  Instruction     #   Issue   ExecS   ExecE   Write  Commit    Status
-------------------------------------------------------------------------------------------
   0               LOAD R1, 0(R0)     0       1       2       7       8       9        OK
   1               LOAD R2, 1(R0)     0       2       3       8       9      10        OK
   2               LOAD R0, 3(R0)     0       8       9      14      15      16        OK
   3               ADD R3, R3, R1     0       9      10      11      12      17        OK
   4                BEQ R2, R3, 2     0      10      12      12      13      18        OK
   5               BEQ R1, R1, -3     0      11      12      12      13      19        OK
   6               ADD R4, R1, R2     0      12      13      14      15      -1   FLUSHED
   7              STORE R3, 2(R0)     0      13      14      -1      -1      -1   FLUSHED
   3               ADD R3, R3, R1     1      19      20      21      22      23        OK
   4                BEQ R2, R3, 2     1      20      22      22      23      24        OK
   5               BEQ R1, R1, -3     1      21      22      22      23      25        OK
   6               ADD R4, R1, R2     1      22      23      24      -1      -1   FLUSHED
   7              STORE R3, 2(R0)     1      23      24      -1      -1      -1   FLUSHED
   3               ADD R3, R3, R1     2      25      26      27      28      29        OK
   4                BEQ R2, R3, 2     2      26      28      28      29      30        OK
   5               BEQ R1, R1, -3     2      27      28      28      29      31        OK
   6               ADD R4, R1, R2     2      28      29      30      -1      -1   FLUSHED
   7              STORE R3, 2(R0)     2      29      30      -1      -1      -1   FLUSHED
   3               ADD R3, R3, R1     3      31      32      33      34      35        OK
   4                BEQ R2, R3, 2     3      32      34      34      35      36        OK
   5               BEQ R1, R1, -3     3      33      34      34      35      37        OK
   6               ADD R4, R1, R2     3      34      35      36      -1      -1   FLUSHED
   7              STORE R3, 2(R0)     3      35      36      -1      -1      -1   FLUSHED
   3               ADD R3, R3, R1     4      37      38      39      40      41        OK
   4                BEQ R2, R3, 2     4      38      40      40      41      42        OK
   5               BEQ R1, R1, -3     4      39      40      40      41      -1   FLUSHED
   6               ADD R4, R1, R2     4      40      41      -1      -1      -1   FLUSHED
   7              STORE R3, 2(R0)     4      41      -1      -1      -1      -1   FLUSHED
   7              STORE R3, 2(R0)     5      42      43      48      49      50        OK

=== PERFORMANCE METRICS ===
Total execution time: 51 cycles
Instructions issued: 29
Instructions committed: 18
IPC: 0.353
Branches encountered: 10
Branch mispredictions: 5
Branch misprediction rate: 50.00%

=== FINAL REGISTER STATE ===
R0: 0  R1: 1  R2: 5  R3: 5
R4: 0  R5: 0  R6: 0  R7: 0
```

Description

This test case is a loop that increments R3 by 1 until it reaches 5. It tests mispredictions and pipeline recovery. It also includes a test in load R0, 3(R0) to show that R0 is always 0. We see multiple flushing happen due to the BEQ R1, R1, -3. This continues until 5 iterations of the loop, then after R3 reaches 5, the exit condition BEQ, R2, R3, 2. Becomes true and skips the second branch and stores R3 in memory safely. The instruction ADD R4, R1, R2 is there to showcase the flushing of values; R4 remains

unchanged at 0, which means that it never executes.

6.4 Test Case 4: all instructions

Input:

```
0
LOAD R1, 0(R0)
LOAD R2, 1(R0)
CALL 6
BEQ R0, R0, 4
ADD R4, R3, R1
STORE R4, 3(R0)
ADD R3, R1, R2
RET
ADD R3, R1, R2
END
0 10
1 20
-1 -1
```

Output

```
Instruction Timing Table (All Issued Instances):
  PC                 Instruction     #   Issue   ExecS   ExecE   Write  Commit    Status
-------------------------------------------------------------------------------------------
   0              LOAD R1, 0(R0)     0       1       2       7       8       9        OK
   1              LOAD R2, 1(R0)     0       2       3       8       9      10        OK
   2                     CALL 6      0       3       4       4       5      11        OK
   3               BEQ R0, R0, 4     0       4       5       5       6      -1   FLUSHED
   4              ADD R4, R3, R1     0       5       6       7       8      -1   FLUSHED
   5             STORE R4, 3(R0)     0       6       8      -1      -1      -1   FLUSHED
   6              ADD R3, R1, R2     0       7       9      10      -1      -1   FLUSHED
   7                         RET     0       8       9       9      10      -1   FLUSHED
   8              ADD R3, R1, R2     0       9      10      -1      -1      -1   FLUSHED
   6              ADD R3, R1, R2     1      11      12      13      14      15        OK
   7                         RET     1      12      13      13      14      16        OK
   8              ADD R3, R1, R2     1      13      14      15      -1      -1   FLUSHED
   3               BEQ R0, R0, 4     1      16      17      17      18      19        OK
   4              ADD R4, R3, R1     1      17      18      -1      -1      -1   FLUSHED
   5             STORE R4, 3(R0)     1      18      -1      -1      -1      -1   FLUSHED
   8              ADD R3, R1, R2     2      19      20      21      22      23        OK
```

Description:

The purpose of this test is to show the control flow of the CALL and RET. We see that when CALL commits at cycle 11, all younger instructions in ROB are flushed and then it saves 3 in R1, and pc goes to 6 to get the instruction ADD R3, R1, R2. Then, when RET commits, the

younger instructions are flushed, and it loads R1 into pc and goes to the instruction BEQ R0, R0, 4, which redirects us to the last instruction in the program ADD R3, R1, R2

7. Conclusion

This project implemented a cycle-accurate simulator of a Tomasulo-based 16-bit RISC processor with speculative execution. The simulator models out-of-order execution, register renaming, dynamic scheduling, and branch recovery through reservation stations, functional units, and a reorder buffer. A detailed cycle-driven pipeline, the system correctly handles data hazards, memory dependencies, and control-flow changes while producing precise timing tables and performance metrics.

The completed simulator supports configurable hardware parameters, enabling experimentation with different microarchitectural setups. The test cases demonstrate correct functionality across all instruction types, proper speculation and misprediction handling, and the expected behavior of the ROB and reservation stations. Overall, the project successfully replicates the core mechanisms of Tomasulo’s algorithm and provides a flexible platform for analyzing instruction-level parallelism and dynamic scheduling.