

Project 2 Report

Sara Naseef, Lina Serry

900202369, 900203355

Department of Computer Science and Engineering

CSCE 3301 - Computer Architecture

Cherif Salama

Fall 2023

**Introduction**

The primary goal of this project is to create a robust simulator that accurately assesses the performance of the proposed 16-bit RISC processor. This simulator will help in understanding the processor's efficiency, speed, and overall performance under various conditions. This report outlines the development of an architectural simulator designed to evaluate the performance of a simplified out-of-order 16-bit RISC processor. The simulator leverages Tomasulo’s algorithm, excluding speculation, to emulate the processor's behavior.

**Our Implementation Logic**

**Reservation Stations:**

* The Reservation Stations are used to hold instructions waiting to be executed. We initialized them in a struct. Each station contains information about the operation, the operands, and the status of the instruction.
* Attributes: Each ReservationStation structure includes flags for busyness, operation type, operand values (Vj, Vk), source operand reservation stations (Qj, Qk), address calculation (A), execution cycles, instruction count, and branching information.

**Instruction Parsing:**

* Functionality: The parseInstruction function takes a string representing an instruction and decodes it into operation codes and registers. This information is crucial for the simulator to understand and process each instruction.

**Reservation Station Allocation:**

* Mechanism: Based on the type of instruction, the simulator allocates an appropriate reservation station. This is managed by a function, which assigns instructions to specific reservation stations based on their operation codes.

**Simulation Process:**

Stages: The simulation process comprises three main stages: issuing, executing, and writing.

* Issuing Stage: Instructions are fetched and queued in the appropriate reservation stations.
* Executing Stage: The simulator checks for operand readiness and executes instructions when ready.
* Writing Stage: Results of computations are written back, and reservation stations are cleared for new instructions.

**Branch Prediction and Handling:**

* Approach: The simulator implements mechanisms for branch prediction and handling, which are crucial for simulating out-of-order execution.

**Clock Cycle and Performance Metrics:**

* The simulation tracks the number of clock cycles required for each instruction and overall program execution. This data is used to calculate performance metrics like CPI (Cycles Per Instruction) and branch miss rate.

**Memory and Registers:**

* Memory (mem) and registers (registers) are simulated as vectors, storing values as the program executes.

**Instruction Queue:**

* Instructions are stored and processed sequentially, with additional handling for jumps and branches.

Tables and Queues:

* Various tables and queues for write queue and load/store queue, manage the flow and execution status of instructions.

**Execution Flow:**

* Initialization: Reservation stations are initialized based on user input, and instructions are parsed and loaded.
* Simulation Loop: The main loop of the simulation (simulateTomasulo function) iterates over the instructions, processing each stage in order.
* Branch and Jump Handling: Special mechanisms handle branch and jump instructions, updating the instruction pointer as necessary.
* Performance Calculation: Upon completion, the simulator outputs performance metrics including the total clock cycles, CPI, and branch miss rate.

**Bonus Features:**

We implemented bonus feature 3. The Feature is responsible for making the user specify the number of reservation stations for each class of instructions and the number of cycles needed by each functional unit type.

**Test Cases:**

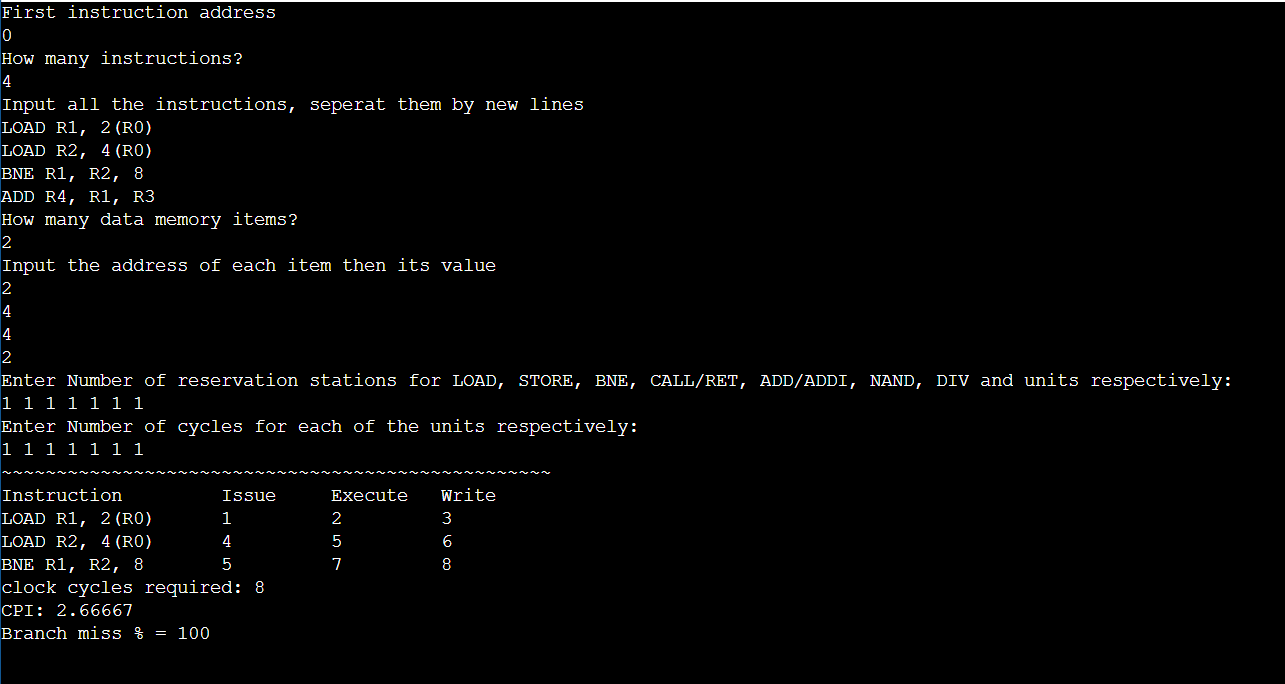
**Test Case 1:** Branching

LOAD R1, 2(R0)

LOAD R2, 4(R0)

BNE R1, R2, 8

ADD R4, R1, R3



After loading in R1-> 4 and R2-> 2, then comparing them, there is a branch miss, and the program doesn’t execute ADD R4, R1, R3. So, the program is branching correctly.

**Test Case 2:** Multiple types of instructions

LOAD R1, 2(R0)

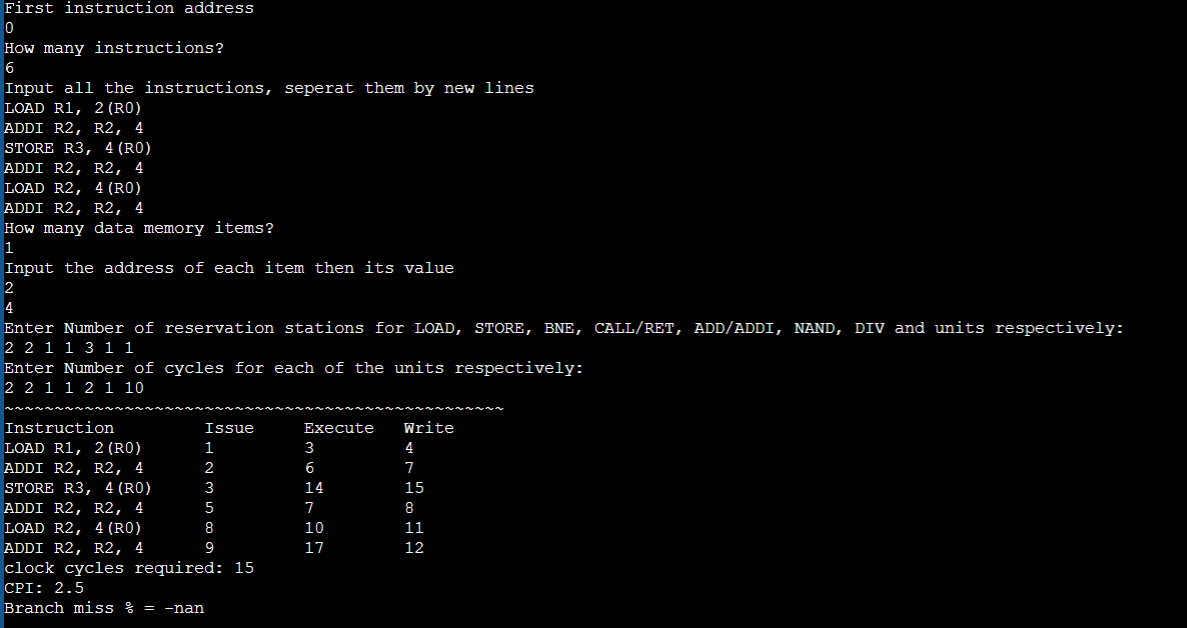
ADDI R2, R2, 4

STORE R3, 4(R0)

ADDI R2, R2, 4

LOAD R2, 4(R0)

ADDI R2, R2, 4



**Test Case 3:** Multiple types

ADD R1, R1, R2

NAND R3, R4, R5

DIV R5, R2, R1

