Tomasulo Algorithm Simulation

Authors:
Saif Abd Elfattah
Adham Ali

Instructor:
Dr. Cherif Salama

Date:
7/12/2024

THE AMERICAN
UNIVERSITY IN CAIRO

Contents

1 Introduction

The Tomasulo Algorithm is a dynamic scheduling algorithm that handles instruction-level parallelism in CPUs. It was designed by Robert Tomasulo in the 1960s at IBM to optimize the performance of out-of-order execution and manage the dependencies between instructions. The key aim of this algorithm is to allow multiple instructions to execute concurrently, thereby improving the throughput and efficiency of the processor. This report explores the simulation of the Tomasulo algorithm for executing a set of instructions, highlighting its major components and their interaction.

In this report, we describe the key components of the Tomasulo algorithm, including the reservation stations, reorder buffer, execution units, and branch predictors. The focus of the simulation is on how these components work together to execute a series of instructions while handling hazards such as data, control, and structural hazards.

The goal of the simulation is to implement the Tomasulo algorithm in a computational system, enabling the analysis of performance improvements over traditional in-order execution systems and gaining insight into the behavior of instruction pipelines under different workloads.

2 Key Components

The key components of the Tomasulo Algorithm simulation implemented in the code are:

Opcode Enumeration

The `Opcode` enumeration defines various instruction opcodes that can be processed, such as ADD, ADDI, MUL, LOAD, STORE, BEQ, CALL, RET, and NAND. This allows for easier handling of different instruction types.

Instruction Class (`Inst`)

The `Inst` class represents an instruction and contains the following important attributes:

- `opcode`: Specifies the operation of the instruction (ADD, MUL, etc.).
- `rs1, rs2, rd`: Represent source and destination registers.
- `Imm`: The immediate value used in certain instructions.
- `memAdr`: The memory address for load and store operations.
- `IssueCyc, execStartCyc, execEndCyc, WriteCyc, CommCyc`: Track the cycle times for different stages of execution.
- `Spec`: A boolean flag indicating whether the instruction is speculative.

The `Inst` class also has various methods such as:

- `PARSE`: Parses a line of text and generates an `Inst` object based on the operation.
- `RegParse`: Converts a register name (e.g., $r1) to an integer.
- `ImmGet`, `rs1Get`, `rs2Get`, etc.: Getter methods for different instruction fields.
- `execEndCycSet`, `CommCycSet`, etc.: Setter methods for cycle times.

Reorder Buffer Class (`Rob`)

The `Rob` class models a reorder buffer entry, which tracks the result of executed instructions and ensures in-order commit. Key features include:

- `opcode`: The operation being performed.
- `dest`: The destination register for the result.
- `value`: The computed result of the instruction.
- `busy`: A boolean flag indicating if the entry is occupied.
- `Index`: The index of the entry in the reorder buffer.
- `Initialze`: Resets the reorder buffer entry.

Reservation Station Class

The `ReservationStation` class models a reservation station in the Tomasulo algorithm. It contains the following attributes:

- `busy`: Indicates whether the reservation station is occupied.
- `opcode`: The operation to be performed (ADD, MUL, etc.).
- `Vj, Vk`: The values of the operands.
- `Qj, Qk`: The indices of the reservation stations producing the operands.
- `Imm`: The immediate value for certain instructions.
- `EntryId`: The index of the entry in the reservation station.
- `Cycsleft`: The number of cycles remaining for the instruction to complete execution.

Methods include:

- `AssignAvalivblty`: Assigns availability to the station.
- `VjSett`, `VkSet`: Set operand values.
- `ROBentryIndexSet`: Assigns an entry in the reorder buffer.
- `CycsLeftSet`: Sets the number of remaining cycles for the operation.
- `CyscsMinusOne`: Decrements the cycle count for ongoing operations.

Execution Unit Class (`ExecutionUnit`)

The `ExecutionUnit` class models the execution unit in the Tomasulo algorithm. Key attributes include:

- `Regs`: A vector of registers.
- `RS`: A vector of reservation stations.
- `rob`: The reorder buffer managing instruction results.
- `Insts`: A vector of instructions to execute.
- `RegStatus`: The status of each register (whether it's available or busy).
- `mem`: The memory system for reading and writing data.
- `Cyc`: The current cycle.
- `branchBredictor`: The branch predictor used for control flow predictions.
- `busylist`: A map indicating the number of busy cycles for each opcode.
- `exec_Cycs`: A map of the number of cycles each instruction type takes to execute.

Methods include:

- `RegSet`: Sets the value of a register.
- `PutInts`: Loads a list of instructions into the execution unit.
- `Commit`: Commits the results of instructions.
- `WriteBack`: Writes back the results to registers.
- `Execution`: Handles the execution of instructions.
- `Issue`: Issues instructions to reservation stations for execution.

Branch Predictor Class (`Branch_Predict`)

The `Branch_Predict` class handles branch prediction. It includes the following:

- `AllBranches`: The total number of branches encountered.
- `mispred`: The number of mispredicted branches.
- `predict`: A method that always returns `false`, simulating a branch predictor.
- `assignPred`: Updates the misprediction statistics based on real and predicted outcomes.
- `mispredRate`: Calculates the misprediction rate.
- `print`: Prints the branch prediction statistics.

3 Performance Metrics

The performance of the processor is assessed based on the following critical metrics:

3.1 Instructions Per Cycle (IPC)

This metric measures how many instructions are successfully completed per clock cycle, offering a view of the processor's throughput.

3.2 Branch Misprediction Rate

This indicates the percentage of branch instructions that were predicted incorrectly. A lower rate is preferable as it reflects greater accuracy in branch predictions.

3.3 Execution Time

The total number of cycles needed to execute the complete sequence of instructions, which serves as a key indicator of the processor's overall efficiency.

4 User Guide

This section provides a step-by-step guide for using the simulation.

4.1 Full Simulation Example

Below is an example input file 'test.txt', containing a set of simple instructions that the processor will execute:

```
1. ADD r3 r6 r8
2. ADD r1 r0 r3
3. MUL r9 r11 r10
4. LOAD r8 4(r1)
5. STORE r8 12(r1)
6. RET
```

4.1.1 Instruction Issue

The processor issues instructions to the reservation stations, checking for any data dependencies. Operands are fetched either from the registers or forwarded from previously executed instructions. If no reservation station is available, or the instruction cannot be issued due to resource constraints or register dependencies, it is deferred to a later cycle. The issue stage is crucial as it ensures that instructions are dispatched only when all necessary operands are available and the required resources are free. This process involves placing the instruction into a reservation station, where it will wait for its operands to be computed or fetched.

4.1.2 Instruction Execution

Once operands are ready and the reservation station is prepared, the instruction begins execution. The cycle count for this step depends on the instruction's opcode. For instance, an ADD operation will take fewer cycles than a MUL operation. The execution phase involves performing the specified operation, such as addition, multiplication, or logical operations, on the operands. The result of the operation is then stored in a temporary buffer, ready to be written back in the next phase. The execution units handle multiple types of instructions, each with varying execution times.

4.1.3 Write-back

After the instruction executes, the results are written back to the registers or memory, depending on the instruction type. For arithmetic or logical operations (such as ADD, MUL, etc.), the result is typically written to a register. For memory-related operations (such as LOAD and STORE), the result is written to memory. In some cases, the results are broadcast to all relevant reservation stations, enabling dependent instructions to continue processing. This ensures that the result of an executed instruction is available for subsequent instructions that rely on it.

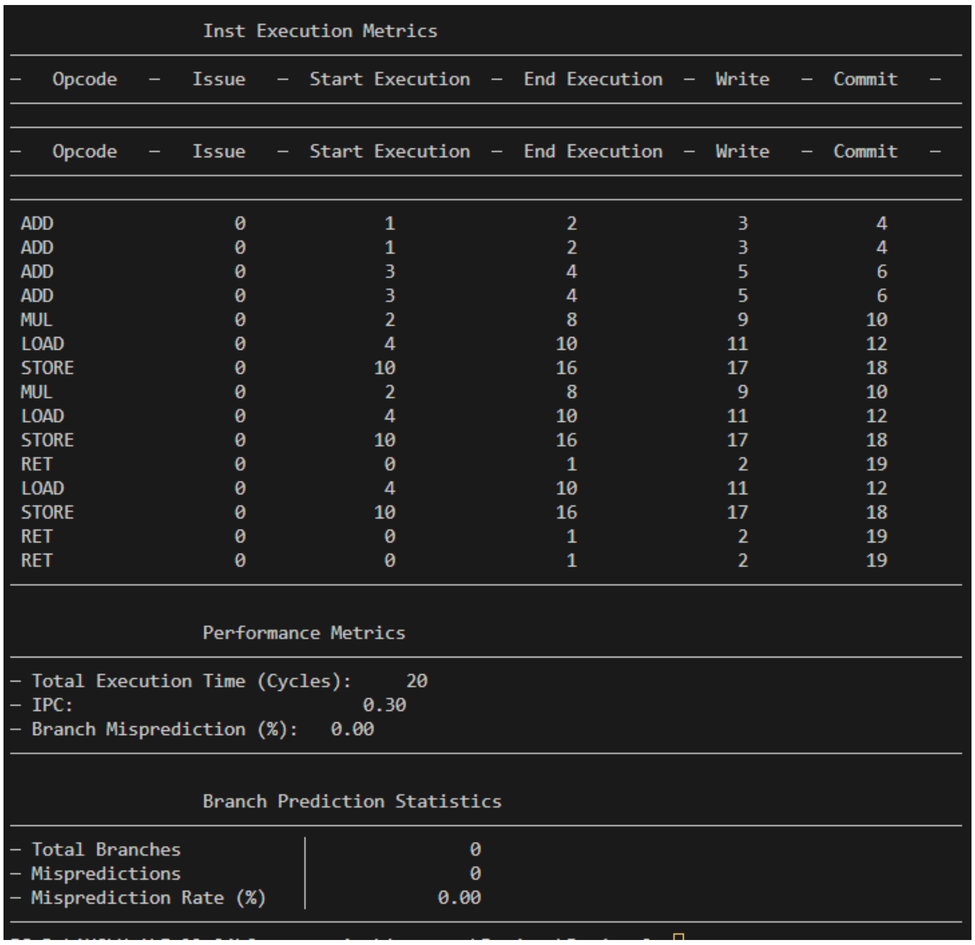
4.1.4 Commit

Once an instruction has been fully executed and its results written back, it is committed from the reorder buffer (ROB) to the register file. This marks the instruction as complete, and any associated resources, such as reservation stations, are freed for future use. Committing an instruction signifies that it is now permanently stored and can be used by subsequent instructions. This step ensures the correctness and consistency of the processor's state and provides the mechanism for completing the execution of instructions in a specified order.

4.1.5 Branch Prediction and Flushing

The processor employs a branch predictor to forecast the outcome of branch instructions (e.g., BEQ). The branch predictor predicts the likely direction of branches, reducing the penalty of branch mispredictions. In the event of a misprediction, the pipeline is flushed, and all instructions following the branch are discarded. This forces the pipeline to reissue the instructions based on the correct branch outcome, ensuring that the processor's execution is accurate. The flushing mechanism helps maintain the integrity of the instruction flow by ensuring that only correct instructions are committed, particularly after branch instructions where the outcome could significantly alter the sequence of execution.

4.1.6 Simulation Output



```
                  Inst Execution Metrics
 ───────────────────────────────────────────────────────────────────────────────────
 ─   Opcode   ─   Issue   ─  Start Execution  ─  End Execution  ─  Write   ─  Commit   ─
 ───────────────────────────────────────────────────────────────────────────────────

 ───────────────────────────────────────────────────────────────────────────────────
 ─   Opcode   ─   Issue   ─  Start Execution  ─  End Execution  ─  Write   ─  Commit   ─
 ───────────────────────────────────────────────────────────────────────────────────

 ───────────────────────────────────────────────────────────────────────────────────
  ADD               0            1               2              3            4
  ADD               0            1               2              3            4
  ADD               0            3               4              5            6
  ADD               0            3               4              5            6
  MUL               0            2               8              9           10
  LOAD              0            4              10             11           12
  STORE             0           10              16             17           18
  MUL               0            2               8              9           10
  LOAD              0            4              10             11           12
  STORE             0           10              16             17           18
  RET               0            0               1              2           19
  LOAD              0            4              10             11           12
  STORE             0           10              16             17           18
  RET               0            0               1              2           19
  RET               0            0               1              2           19
 ───────────────────────────────────────────────────────────────────────────────────

                  Performance Metrics
 ───────────────────────────────────────────────────────────────────────────────────
 - Total Execution Time (Cycles):     20
 - IPC:                           0.30
 - Branch Misprediction (%):   0.00
 ───────────────────────────────────────────────────────────────────────────────────

                  Branch Prediction Statistics
 ───────────────────────────────────────────────────────────────────────────────────
 - Total Branches          │          0
 - Mispredictions          │          0
 - Misprediction Rate (%)  │       0.00
 ───────────────────────────────────────────────────────────────────────────────────
```

Figure 1: Simulation Output

5 Bonus Features

5.1 Graphical User Interface (GUI)

A user-friendly interface was built using Python's Tkinter library, providing a visual and interactive way to explore and operate the simulation.

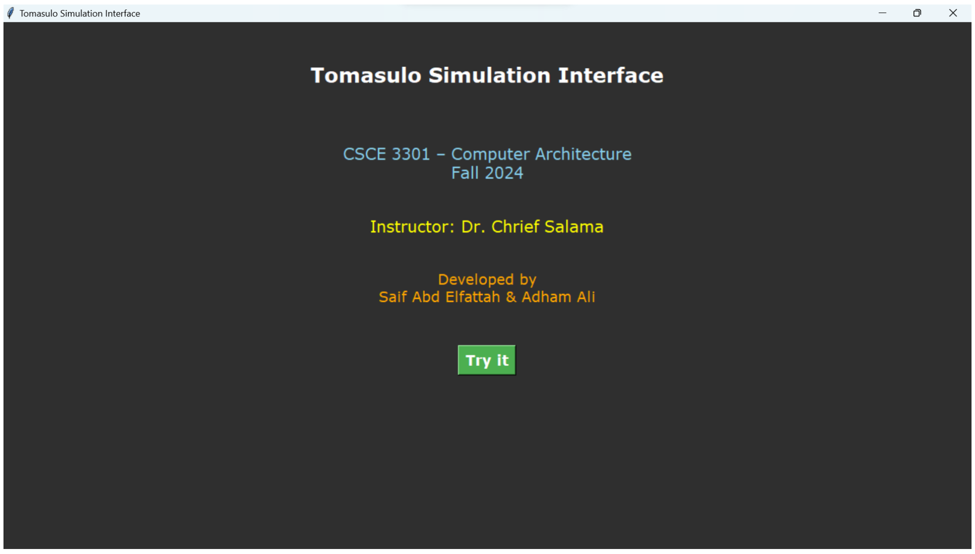

Tomasulo Simulation Interface
Tomasulo Simulation Interface
CSCE 3301 – Computer Architecture
Fall 2024
Instructor: Dr. Chrief Salama
Developed by
Saif Abd Elfattah & Adham Ali
Try it

Figure 2: GUI

5.2 Input Parsing from File

The simulation includes a feature that allows users to import instruction sets directly from a file, simplifying the process of supplying data to the system.

6 Conclusion

The simulation effectively models the functioning of a pipelined processor utilizing Tomasulo's algorithm with speculative execution, showcasing the benefits of out-of-order execution, branch prediction, and reservation stations. It emphasizes key concepts such as instruction scheduling, hazard detection, and execution, which are fundamental to modern processor architecture. Furthermore, the addition of a graphical user interface and file input parsing improves the simulation's user experience, enabling users to easily test various instruction sets and assess performance.