Dynamic Instruction Scheduling Using Tomasulo Algorithm

Ashwinth Anbu M (CS22B2055)

Abishek Chakravarthy (CS22B2054)

March 25, 2025

Contents

1 Introduction

Dynamic instruction scheduling is a crucial technique in modern CPU design that allows for increased instruction-level parallelism. The Tomasulo algorithm, developed by Robert Tomasulo at IBM in 1967, is a hardware algorithm for dynamic scheduling that allows out-of-order execution while handling data dependencies efficiently. This report details the implementation of a simulator for a processor that uses the Tomasulo algorithm with in-order issue, out-of-order execution, and out-of-order commit capabilities.

2 System Architecture

2.1 Processor Specifications

The simulated processor has the following specifications:

- 64-bit processor architecture
- 32 general-purpose registers, each 64 bits wide
- In-order instruction issue
- Out-of-order execution
- Out-of-order commit using a Reorder Buffer (ROB)

2.2 Functional Units

The processor includes multiple functional units with various execution latencies:

Functional Unit	**Execution Cycles**
Integer Addition/Subtraction (IADD)	6
Integer Multiplication (IMUL)	12
Floating Point Addition/Subtraction (FADD)	18
Floating Point Multiplication (FMUL)	30
Load (LD)	1
Store (ST)	1
Logic Unit (AND/OR/XOR)	1
No Operation (NOP)	1

Table 1: Functional Units and Their Execution Latencies

2.3 Instruction Set

The processor supports the following instruction types:

- `IADD Rdst, Rsrc1, Rsrc2` - Integer addition
- `IMUL Rdst, Rsrc1, Rsrc2` - Integer multiplication
- `FADD Rdst, Rsrc1, Rsrc2` - Floating-point addition
- `FMUL Rdst, Rsrc1, Rsrc2` - Floating-point multiplication
- `LD Rdst, Mem` - Load from memory
- `ST Mem, Rsrc` - Store to memory
- `AND/OR/XOR Rdst, Rsrc1, Rsrc2` - Logical operations
- `NOP` - No operation

3 Implementation Details

3.1 Data Structures

The simulator implements the following key data structures:

3.1.1 Instruction

Represents an instruction in the program with its type, source/destination registers, execution status, and timing information:

```
struct Instruction {
    InstructionType type;
    int dest;
    int src1;
    int src2;
    int memAddr;
    int cycles;
    int issueTime;
    int execStartTime;
    int execCompTime;
    int commitTime;
    bool isIssued;
    bool isExecStarted;
    bool isExecCompleted;
    bool isCommitted;
    // Constructor and methods...
};
```

3.1.2 Reservation Station

Holds information about an instruction being executed, including its operands and dependencies:

```
struct ReservationStation {
    bool busy;
    InstructionType op;
    int Vj, Vk;      // Values of operands
    int Qj, Qk;      // ROB entries that will produce operands
    int dest;        // Destination ROB entry
    int cycles;      // Remaining execution cycles
    int instrIndex; // Index in the instructions array
    // Constructor...
};
```

3.1.3 Reorder Buffer Entry

Tracks instructions in program order for in-order commit:

```
struct ROBEntry {
    bool ready;       // Execution completed
    int instrIndex;   // Index in the instructions array
    int dest;          // Destination register
    int value;         // Result value
    // Constructor...
};
```

3.1.4 Register Status

Tracks which ROB entry will update each register:

```
struct RegisterStatus {
    int robIndex;     // -1 if register is ready
    // Constructor...
};
```

3.2 Pipeline Stages

The simulator implements the following pipeline stages:

3.2.1 Issue Stage

Instructions are issued in-order from the program counter (PC). For each instruction:

1. Check if ROB is full; if so, stall
2. Find an available reservation station; if none, stall
3. Add instruction to reservation station
4. Collect available operands and identify dependencies
5. Add entry to ROB
6. Update register status for destination register
7. Update instruction status and timing
8. Increment PC

3.2.2 Execute Stage

Instructions execute out-of-order when their operands are ready:

1. For each busy reservation station, check if all operands are ready
2. If ready and not started, mark as started and record start time
3. Decrement remaining execution cycles
4. When execution completes, mark ROB entry as ready and free the RS

3.2.3 Commit Stage

Instructions commit in-order from the ROB:

1. Check if the instruction at ROB head is ready
2. If ready, update register file with result
3. Update register status if needed
4. Mark instruction as committed and record commit time
5. Increment ROB head pointer

4 Sample Program Execution

The simulator was tested with the following assembly program:

```
LD R1, M10
LD R2, M20
IMUL R3, R1, R2
IADD R4, R1, R3
FADD R5, R2, R4
ST M30, R5
```

4.1 Instruction Packet Generation

When an instruction is fetched, it is decoded into an instruction packet containing:

- Instruction type (opcode)
- Source register numbers
- Destination register number
- Memory address (for load/store)
- Execution cycles required

4.2 Execution Trace

Based on the simulation results, below is the execution trace showing how instructions progress through the pipeline stages:

Instruction	Issue	Exec Start	Exec Complete	Commit
LD R1, M10	0	1	1	2
LD R2, M20	1	2	2	3
IMUL R3, R1, R2	2	4	15	16
IADD R4, R1, R3	3	16	21	22
FADD R5, R2, R4	4	22	39	40
ST M30, R5	5	40	40	41

Table 2: Execution Trace (Clock Cycles)

4.3 Sample Cycle Analysis

Below is a detailed snapshot of the processor state at Clock Cycle 36:

```
Clock Cycle: 36
Instruction Status:
Instruction 0: Issued at 0, Exec started at 0, Exec completed
    at 0, Committed at 0
Instruction 1: Issued at 1, Exec started at 1, Exec completed
    at 1, Committed at 1
Instruction 2: Issued at 2, Exec started at 2, Exec completed
    at 13, Committed at 13
Instruction 3: Issued at 3, Exec started at 14, Exec completed
    at 19, Committed at 19
Instruction 4: Issued at 4, Exec started at 20
Instruction 5: Issued at 5

Reservation Stations:
FADD[0]: Busy, Instr: 4, Qj: -1, Qk: -1, Vj: 0, Vk: 0, Dest: 4,
     Cycles left: 1
MEM[0]: Busy, Instr: 5, Qj: 4, Qk: -1, Vj: 0, Vk: 0, Dest: 5,
    Cycles left: 1
```

At this cycle:

- Instructions 0-3 have completed execution and been committed
- Instruction 4 (FADD) is in its final execution cycle
- Instruction 5 (ST) is waiting for Instruction 4 to complete (data dependency)
- Most reservation stations are free, indicating no resource contention at this point

5 Performance Analysis

5.1 Performance Metrics

Based on the simulation results, the following performance metrics were obtained:

Metric	**Value**
Instructions Per Cycle (IPC)	0.142857
Average Instruction Latency	18.1667 cycles
Structural Hazard Stalls	0
Data Hazard Stalls	4
Total Execution Time	42 cycles

Table 3: Performance Metrics Summary

5.2 Stall Analysis

The simulation detected 3 data hazard stalls but no structural hazards. This indicates that:

- The processor has sufficient hardware resources (reservation stations, reorder buffer entries) for this program
- True data dependencies are the primary limiting factor for performance
- The Tomasulo algorithm effectively handled register renaming, but was limited by the inherent dependencies in the code

6 Conclusion

The implemented Tomasulo algorithm simulator successfully models a processor with in-order issue, out-of-order execution, and out-of-order commit capabilities. The performance analysis of the sample program execution reveals several key insights:

- The achieved IPC of 0.153846 is limited primarily by data dependencies in the program rather than hardware resource constraints
- The processor successfully handled register renaming and data hazard detection
- No structural hazards were encountered, indicating sufficient hardware resources

- The average instruction latency of 15.5 cycles reflects both the inherent execution latencies of instructions and waiting periods due to dependencies

These results demonstrate that the Tomasulo algorithm provides benefits even for programs with heavy data dependencies, but the performance improvements are ultimately bounded by the inherent parallelism available in the program. For this specific test program, a critical path of dependent instructions limits the potential speedup.