Tomasulo Algorithm Simulator Development Report

1. Introduction

This project implements a comprehensive GUI-based simulator for the Tomasulo algorithm using JavaFX. The simulator accurately models out-of-order execution with reservation stations, register renaming, and a write-back cache system. It handles all types of data hazards (RAW, WAR, WAW), address clashes, branch instructions, and complex scenarios including loops and simultaneous instruction completions.

Key Features

- **Complete Tomasulo Pipeline:** Issue, Execute, and Write-Back stages.
- **Reservation Stations:** Separate stations for FP Add/Sub, FP Mul/Div, Integer ALU, Load, and Store operations.
- **Register File:** 32 integer registers (R0-R31) and 32 floating-point registers (F0-F31).
- **Direct-Mapped Write-Back Cache:** Configurable cache size, block size, hit latency, and miss penalty.
- **Address Clash Detection:** Issue-time and execute-time checks for memory hazards.
- **Branch Handling:** Stall-on-branch with no prediction.
- **Simultaneous Completion Handling:** FIFO-based write-back when multiple instructions complete in the same cycle.
- **GUI Visualization:** Real-time display of all pipeline components.

---

2. Development Approach

2.1 Development Methodology

The project was developed using an iterative, component-based approach:

1. **Core Architecture Design:** Established the main simulator class and component interfaces.
2. **Component Implementation:** Built each component (Reservation Stations, Cache, Register File, etc.) independently.
3. **Integration:** Connected components with proper data flow and state management.
4. **Hazard Handling:** Implemented address clash detection and register tag management.
5. **Edge Case Resolution:** Addressed simultaneous completions, branch loops, and premature termination.
6. **GUI Development:** Created JavaFX tables for visualization.
7. **Testing and Debugging:** Validated with provided test cases and fixed edge cases.

2.2 Technology Stack

- **Language:** Java 11+
- **GUI Framework:** JavaFX
- **Build System:** Gradle
- **IDE:** Compatible with IntelliJ IDEA, Eclipse, VS Code

2.3 Project Structure

```
Micro/
├── src/main/java/com/example/tomasulo/
│   ├── core/
│   │   ├── TomasuloSimulator.java       # Main simulator engine
│   │   ├── Instruction.java             # Instruction representation
│   │   ├── InstructionParser.java       # Parses assembly instructions
│   │   └── ExecutionState.java          # Tracks simulation state
│   ├── components/
│   │   ├── ReservationStationManager.java  # Manages all RS
│   │   ├── ReservationStation.java         # Individual RS
│   │   ├── RegisterFile.java               # Register file with tags
│   │   ├── Cache.java                      # Direct-mapped write-back cache
│   │   ├── CacheBlock.java                 # Cache block representation
│   │   ├── LoadStoreBuffer.java            # Load/Store buffer
│   │   └── BranchUnit.java                 # Branch instruction handler
│   ├── gui/
│   │   ├── TomasuloGUI.java                # Main GUI window
│   │   ├── ReservationStationTable.java    # RS display table
│   │   ├── LoadStoreBufferTable.java       # L/S Buffer display
│   │   ├── RegisterFileTable.java          # Register file display
│   │   ├── CacheTable.java                 # Cache display
│   │   └── InstructionListTable.java       # Instruction queue display
│   └── utils/
│       ├── InstructionType.java            # Instruction type enum
│       ├── RegisterType.java               # Register type enum
│       └── Constants.java                  # Default constants
├── test_case_1.txt                         # Sequential test case
├── test_case_2.txt                         # Sequential test case
├── test_case_3.txt                         # Loop test case
└── build.gradle                            # Gradle build configuration
```

---

3. Code Structure

3.1 Main Simulator Class: TomasuloSimulator.java

The core simulator orchestrates all pipeline stages and component interactions.

Key Responsibilities:

- **Cycle Management:** Increments cycle counter and coordinates stage execution.
- **Pipeline Stages:** Executes Issue → Execute → Write-Back in order.
- **State Tracking:** Maintains instruction pointer, completion status, and execution log.
- **Simultaneous Completion Groups:** Tracks instructions that finish execution simultaneously.

Pipeline Stage Order (per cycle):

1. Execute stage (try to start execution for ready instructions)
2. Issue stage (issue new instructions if not stalled)
3. Write-back stage (write back completed instructions)
4. Tick (decrement cycles for executing instructions)
5. Check execution end (mark instructions that finished execution)
6. Check completion (determine if simulation is complete)

Key Data Structures:

- `simultaneousCompletionGroups`: Maps executeEndCycle → Set of RS names
- `incompleteGroupMembers`: RS names in incomplete simultaneous completion groups
- `incompleteGroupMembersWrittenBackThisCycle`: RS that wrote back this cycle
- `dependentRSWaitingForIncompleteGroup`: Dependent RS waiting for incomplete groups

3.2 Instruction Processing: Instruction.java

Represents a single instruction with all metadata:

- Instruction type (ADD.D, MUL.D, L.D, S.D, etc.)
- Source and destination registers
- Immediate values and base registers
- Pipeline stage timestamps (Issue, Exec Start, Exec End, Write-Back)
- Completion status

3.3 Instruction Parser: InstructionParser.java

Parses MIPS assembly instructions from text files:

- Supports FP operations: ADD.D, SUB.D, MUL.D, DIV.D
- Supports integer operations: ADDI, DADDI, SUBI, DSUBI
- Supports memory operations: L.D, L.S, LW, LD, S.D, S.S, SW, SD
- Supports branches: BEQ, BNE
- Handles labels and immediate values

---

4. Core Components Implementation

4.1 Reservation Station Manager

File: `ReservationStationManager.java`

Manages all reservation stations organized by instruction category:

- **FP Add/Sub Stations:** For ADD.D, SUB.D operations
- **FP Mul/Div Stations:** For MUL.D, DIV.D operations
- **Integer ALU Stations:** For ADDI, SUBI, DADDI, DSUBI operations
- **Load Stations:** For load operations
- **Store Stations:** For store operations

Operand Update Logic: When a reservation station writes back, `updateOperands()` is called to:

1. Find all RS waiting for that result (Qj or Qk matches the station name).
2. Clear the tag (set Qj/Qk to null).
3. Update the value (Vj/Vk) with the result.
4. For loads/stores: Calculate address from base register value.

4.2 Reservation Station

File: `ReservationStation.java`

Represents a single reservation station with unique identifiers, busy flags, operation mnemonics, operand values (Vj, Vk), tags (Qj, Qk), and destination tracking.

4.3 Register File

File: `RegisterFile.java`

Manages 32 integer and 32 floating-point registers.

- **Storage:** `LinkedHashMap` to preserve insertion order (R0-R31, F0-F31).
- **R0 Hardcoding:** R0 is always 0.0 and cannot be modified.
- **Tag Tracking:** Each register has an associated tag (RS name) indicating the current producer.
- **Value Storage:** Stores register values as doubles (integers stored as doubles).

4.4 Load/Store Buffer

File: `LoadStoreBuffer.java`

Manages memory operations with address clash detection.

Address Clash Detection:

- **Issue-Time Check (`hasAddressClashAtIssue`):** Performed when the base register is ready at issue time. Checks against earlier, incomplete memory operations.
- **Execute-Time Check (`hasAddressClash`):** Performed when the address is calculated during execution. Checks against earlier operations (by issue cycle).
- **Clash Rules:**
    - LOAD vs STORE: Load must wait for earlier store to same address.
    - STORE vs LOAD: Store must wait for earlier load to same address.
    - STORE vs STORE: Store must wait for earlier store to same address.

4.5 Branch Unit

File: `BranchUnit.java`

Handles branch instruction execution using a **Stall-on-Branch** strategy (no prediction).

- **Stall Mechanism:** When a branch issues, the `branchStall` flag is set, preventing new instructions from issuing.
- **Evaluation:** Condition evaluated at write-back using current register values.
- **Resolution:** If taken, IP is updated to the target address. If the branch causes a loop back, the completion flag is reset.

---

5. Cache Implementation

5.1 Cache Architecture

File: `Cache.java`

The cache is implemented as a **direct-mapped, write-back cache** with the following characteristics:

- **Mapping Formula:** `index = (address / blockSize) % numBlocks`
- **Write-Back Policy:** A dirty bit indicates modification. Main memory is only updated upon eviction of a dirty block.
- **Dynamic Block Base Address:** Each cache block stores its starting address (`baseAddress`), allowing blocks to start at any address rather than fixed aligned boundaries.

5.2 Cache Block Structure

File: `CacheBlock.java`

Each block contains the data (byte array), tag, base address, valid bit, and dirty bit.

5.3 Cache Operations

Load Operation:

1. Calculate cache index.
2. If **HIT**: Read data, return hit latency.
3. If **MISS**: Write back dirty block (if needed), load new block from memory, read data, return miss penalty.

Store Operation:

1. Calculate cache index.
2. If **HIT**: Write data, mark dirty, return hit latency.
3. If **MISS**: Write back dirty block (if needed), load new block, write data, mark dirty, return miss penalty.

5.4 Data Representation

Values are stored as **unsigned integers in little-endian byte order**.

- **Conversion:** `double` → `long` → little-endian `byte[]`.
- **Example:** Value 100 with size 8 → `[0x64, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00]`

---

6. Hazard Handling and Edge Cases

6.1 Address Clash Handling

Problem: An instruction should not issue if it conflicts with an earlier, incomplete memory operation. **Solution (Issue-Time):**

Java
```
// For LOAD instructions
if (loadAddress != null) {
    if (loadStoreBuffer.hasAddressClashAtIssue(loadAddress, loadSize,
            LoadStoreBuffer.LoadStoreType.LOAD, inst.getIssueCycle())) {
        state.getTrace().remove(state.getTrace().size() - 1);
        return; // Do not issue
    }
}
```

6.2 Register Tag Management (Race Condition)

Problem: When an older instruction writes back, it might clear the tag of a register that a newer instruction is also targeting. **Solution:** Clear tag only if the writing RS is still the current producer.

Java
```
String currentTag = registerFile.getTag(dest);
if (currentTag != null && currentTag.equals(rs.getName())) {
    registerFile.clearTag(dest);
}
```

6.3 Simultaneous Completion Handling

Problem: Multiple instructions finishing execution in the same cycle can cause logic errors if dependent instructions start before the write-back is fully processed. **Solution:**

1. **Group Detection:** Identify instructions ready for write-back with the same `executeEndCycle`.
2. **Dependent Delay:** Prevent dependent instructions from starting execution if their dependency belongs to an incomplete group.

6.4 Branch Loop Handling

Problem: When a branch causes a loop back, the simulation might mark itself as complete prematurely. **Solution:** Reset the completion flag when the branch is taken and IP loops back. Ensure the Load/Store Buffer is empty before marking simulation as complete.

6.5 FIFO Write-Back

Solution: When multiple instructions are ready for write-back, sort by issue cycle (FCFS) and write back only one per cycle to simulate bus conflict resolution.

---

7. GUI Implementation

File: `TomasuloGUI.java`

The main window provides a comprehensive dashboard:

- **Control Panel:** Load File, Step, Run, Reset buttons.
- **Reservation Station Table:** Displays Name, Busy, Op, Vj, Vk, Qj, Qk, Dest, Cycles.
- **Load/Store Buffer Table:** Shows pending memory operations, addresses, and values.
- **Register File Table:** Ordered display (R0-R31, F0-F31) with values and tags.
- **Cache Table:** Displays Index, Valid, Dirty, Base Address, and Hex values.
- **Instruction Queue:** Shows timestamps for Issue, Exec Start, Exec End, and Write Back.

Update Mechanism: Tables are refreshed after every simulation step (`step()`) to reflect the current state of the `ReservationStationManager`, `RegisterFile`, and `Cache`.

---

8. Test Cases

8.1 Test Case 1: Sequential Code with Address Clash

Key Scenario: `S.D F6, 8(R2)` must wait for `L.D F2, 8(R2)` to write back due to address overlap. **Result:** ✅ Address clash correctly prevents S.D from issuing until L.D writes back. All dependencies are correctly tracked.

8.2 Test Case 2: Sequential Code with Cache Operations

Key Scenario: Validating cache hits, misses, and the dirty bit write-back policy. **Result:** ✅ Cache blocks correctly track dynamic base addresses and dirty bits are set on stores.

8.3 Test Case 3: Loop Code with Branch

Key Scenario: A `BNE` instruction loops execution back to a label (`LOOP`). **Result:** ✅ Loop executes correctly without premature termination. Branch resolution uses current register values, and simultaneous completions (MUL.D and S.D) are handled correctly.

---

9. Running and Testing

9.1 Building the Project

Bash
Using Gradle wrapper
./gradlew build

9.2 Running the Simulator

Option 1: `run_simulator.bat` (Windows Batch Script) **Option 2:** `./gradlew run`
Option 3: `java -jar build/libs/javafx-test-1.0-SNAPSHOT.jar`

9.3 Usage Guide

1. **Load Instructions:** Click "Load File" and select a test case.
2. **Configure:** Adjust cache/latency parameters if necessary.
3. **Run:** Use "Step" for cycle-by-cycle analysis or "Run" for full execution.
4. **Verify:** Check the "Execution Log" tab for detailed event tracking.

---

10. Conclusion

This project successfully implements a robust Tomasulo algorithm simulator.

- **Achievements:** Complete pipeline execution, accurate hazard handling, dynamic cache system, and effective GUI visualization.
- **Key Highlights:** The implementation successfully manages complex edge cases such as simultaneous instruction completion and register tag race conditions.
- **Future Enhancements:** Potential additions include branch prediction, sophisticated cache replacement policies, and performance metrics.