Tomasulo's Algorithm Processor Implementation

Introduction

Tomasulo's algorithm is an efficient dynamic scheduling technique that enables out-of-order execution while maintaining correct program behavior. Unlike static scheduling approaches like VLIW, Tomasulo's algorithm handles hazards at runtime using reservation stations and register renaming. This approach allows the processor to maximize instruction-level parallelism while maintaining correct execution semantics.

This report discusses the implementation of a processor simulator using Tomasulo's algorithm that handles structural and data hazards (RAW and WAW) through reservation stations and queues. When reservation stations are full, instructions are placed in queues associated with each functional unit. The simulator processes an assembly program, schedules instructions dynamically, and demonstrates the execution timeline.

Tomasulo's Algorithm Overview

Tomasulo's algorithm achieves dynamic scheduling through three key components:

- **Reservation Stations**: Each functional unit has associated reservation stations that buffer operations waiting to execute.
- **Register Renaming**: Uses virtual registers to eliminate WAR and WAW hazards.
- **Common Data Bus**: Broadcasts results to all waiting reservation stations.

Functional Units and Queues

The processor has the following functional units, each with associated reservation stations and queues:

- **Integer Adder (IADD/ISUB)**: Handles integer addition and subtraction (6 clock cycles) with a queue when reservation stations are full.

- **Integer Multiplier (IMUL)**: Handles integer multiplication (12 clock cycles) with a queue.
- **Integer Divider (IDIV)**: Handles integer division (24 clock cycles) with a queue.
- **Floating-Point Adder (FADD/FSUB)**: Handles floating-point addition and subtraction (18 clock cycles) with a queue.
- **Floating-Point Multiplier (FMUL)**: Handles floating-point multiplication (30 clock cycles) with a queue.
- **Floating-Point Divider (FDIV)**: Handles floating-point division (60 clock cycles) with a queue.
- **Logical Units**: Handle logical operations (AND, OR, XOR, NOT) in 1 clock cycle with a queue.
- **Memory Unit (LD/ST)**: Handles load and store operations in 1 clock cycle with a queue.

Architecture Constraints:

- Each functional unit has a limited number of reservation stations (typically 2 per unit)
- When reservation stations are full, instructions are placed in a queue specific to that functional unit
- Instructions are issued to reservation stations when operands become available
- Results are broadcast on a common data bus to all waiting reservation stations

Queue Behavior:

- All functional unit queues follow **First-In-First-Out (FIFO)** policy
- Queues are unbounded in simulation (can hold any number of instructions)
- **Key Insight**: The queue system effectively behaves as an infinitely large reservation station
  - Instructions wait in queues when reservation stations are full
  - When popped, they resume execution from the **EX stage** (skipping IF/ID)
  - Microinstruction status is preserved during queuing

Hazard Handling

Tomasulo's algorithm handles hazards differently than static scheduling approaches:

- **Structural Hazards**: Handled by reservation stations and queues. If all reservation stations for a functional unit are occupied, new instructions wait in the queue.
- **Queue-Based Execution**:
  - When reservation stations fill, instructions enter unit-specific queues
  - Queue processing maintains pipeline continuity:
    1. Instruction completes IF and ID stages normally
    2. If reservation station full, moves to queue (preserving operand status)
    3. When dequeued, resumes directly at EX stage
    4. Uses preserved microinstruction state to continue execution
  - Effectively creates a virtual infinite reservation station
- **RAW (Read-After-Write)**: Resolved by tracking operand availability in reservation stations. Instructions wait in reservation stations until their operands are ready.
- **WAW (Write-After-Write)**: Eliminated through register renaming. Each write creates a new virtual register.
- **WAR (Write-After-Read)**:
  - Prevented through a two-phase register protection scheme:
    1. **Read Phase Protection**:
       - When an instruction reads registers during ID stage
       - Hardware temporarily locks all source registers (Rsrc1, Rsrc2)
       - Locks are released immediately after ID stage completes
    2. **Write Phase Protection**:
       - When an instruction writes to a register (Rdst)
       - Hardware locks the destination register during ID stage
       - Lock persists until WB stage completes
  - **Why This Works**:
    - Any subsequent write must wait until previous reads complete (via read locks)
    - Register renaming provides additional protection by creating unique register versions
    - Example: If instruction 1 reads R1, instruction 2 cannot write R1 until instruction 1 finishes reading

Key Insight: The same register locking strategy used in our VLIW implementation - where source registers are only protected during ID stage and destination registers until WB - works effectively with Tomasulo's register renaming to completely eliminate WAR hazards without additional hardware.

Instruction Set

The processor supports the following instructions, each mapped to specific functional units with their associated queues:

Instruction	**Description**	**Functional Units and Pipeline Stages**
IADD Rdst, Rsrc1, Rsrc2	Integer Addition	`IF` (1), `ID` (1), `RS` (wait), `ADD` (6), `WB` (1)
ISUB Rdst, Rsrc1, Rsrc2	Integer Subtraction	`IF` (1), `ID` (1), `RS` (wait), `ADD` (6), `WB` (1)
IMUL Rdst, Rsrc1, Rsrc2	Integer Multiplication	`IF` (1), `ID` (1), `RS` (wait), `MUL` (12), `WB` (1)
IDIV Rdst, Rsrc1, Rsrc2	Integer Division	`IF` (1), `ID` (1), `RS` (wait), `DIV` (24), `WB` (1)
FADD Rdst, Rsrc1, Rsrc2	Floating-Point Addition	`IF` (1), `ID` (1), `RS` (wait), `FADD` (18), `WB` (1)
FSUB Rdst, Rsrc1, Rsrc2	Floating-Point Subtraction	`IF` (1), `ID` (1), `RS` (wait), `FADD` (18), `WB` (1)
FMUL Rdst, Rsrc1, Rsrc2	Floating-Point Multiplication	`IF` (1), `ID` (1), `RS` (wait), `FMUL` (30), `WB` (1)
FDIV Rdst, Rsrc1, Rsrc2	Floating-Point Division	`IF` (1), `ID` (1), `RS` (wait), `FDIV` (60), `WB` (1)
AND Rdst, Rsrc1, Rsrc2	Logical AND	`IF` (1), `ID` (1), `RS` (wait), `AND` (1), `WB` (1)
OR Rdst, Rsrc1, Rsrc2	Logical OR	`IF` (1), `ID` (1), `RS` (wait), `OR` (1), `WB` (1)
XOR Rdst, Rsrc1, Rsrc2	Logical XOR	`IF` (1), `ID` (1), `RS` (wait), `XOR` (1), `WB` (1)
NOT Rdst, Rsrc1	Logical NOT	`IF` (1), `ID` (1), `RS` (wait), `NOT` (1), `WB` (1)
LD Rdst, [Mem]	Load from Memory	`IF` (1), `ID` (1), `RS` (wait), `MEM` (1), `WB` (1)
ST Rsrc, [Mem]	Store to Memory	`IF` (1), `ID` (1), `RS` (wait), `MEM` (1), `WB` (1)
NOP	No Operation	`IF` (1)

Code Explanation

The simulator is implemented in Python and consists of the following key components:

- **Reservation Station Management**: Tracks available reservation stations and queues instructions when stations are full.

- **Register Renaming**: Handles WAW and WAR hazards by dynamically allocating registers.
- **Instruction Issue**: Issues instructions to reservation stations when operands are available.
- **Execution Timeline**: Tracks the progress of each instruction through the pipeline.

Key Functions

- `parse_assembly`: Parses the assembly code into a list of instructions.
- `handle_reservation_stations`: Manages reservation station allocation and queueing.
- `handle_waw_and_war`: Implements register renaming to eliminate WAW and WAR hazards.
- `display_clock_cycle_execution`: Displays the execution timeline for each instruction.
- `display_functional_units`: Shows which functional units are active each cycle.
- `plot_pipeline_gantt`: Visualizes the pipeline execution.

Input and Output

Input Assembly Code

The input assembly program performs basic arithmetic operations:

```
LD R1, [1000]
LD R2, [2000]
IADD R3, R1, R2
ISUB R4, R1, R2
IADD R4, R4, R2
IMUL R5, R1, R2
IDIV R6, R1, R2
ST R3, [3000]
ST R4, [4000]
ST R5, [5000]
ST R6, [6000]
```

Output Explanation

The simulator outputs the execution timeline and reservation station usage:

Updated Code with Register Renaming

The algorithm automatically renames registers to eliminate WAW hazards:

```
LD R1, [1000]
LD R2, [2000]
IADD R3, R1, R2
ISUB R4, R1, R2
IADD R0, R4, R2  // R4 renamed to R0
IMUL R5, R1, R2
IDIV R6, R1, R2
ST R3, [3000]
ST R0, [4000]    // Uses renamed register
ST R5, [5000]
ST R6, [6000]
```

Total Clock Cycles

The program takes **35 clock cycles** to execute.

Reservation Station Activity

The simulator shows detailed reservation station allocation and queueing behavior:

- **Cycle 5**:
  - Instruction 3 (IADD) acquires ADD reservation station
  - ADD stations available: 1/2
- **Cycle 6**:
  - ADD station released (now 2/2 available)
  - Instruction 4 (ISUB) acquires ADD station
  - ADD stations available: 1/2
- **Cycle 7**:
  - Instruction 5 (IADD) acquires last ADD station
  - ADD stations full (0/2 available)
- **Cycle 10-11**:
  - Instructions 8-9 (ST) acquire MEM stations
  - MEM stations fill (0/2 available by cycle 11)
- **Cycle 12**:
  - ADD station released (1/2 available)

  - MEM stations full - Instruction 10 pushed to MEM queue
- **Cycle 13**:
  - MEM station released (1/2 available)
  - Instruction 10 popped from MEM queue and acquires station
  - Instruction 11 pushed to MEM queue (stations full again)
- **Cycle 19**:
  - ADD stations fully released (2/2 available)
- **Cycle 26**:
  - MEM station released (1/2 available)
  - Instruction 11 popped from MEM queue and acquires station
- **Cycle 27**:
  - MEM station released (1/2 available)
- **Cycle 34**:
  - MEM stations fully released (2/2 available)
- **Final Status**:
  - All reservation stations available (2/2)
  - All queues empty
  - All functional units idle

Key Observations:

- **ADD Unit Contention**:
  - All 2 stations occupied cycles 5-7
  - No queuing needed due to fast instruction completion
- **MEM Unit Operations**:
  - Required queueing for stores during cycles 10-26
  - Maintained correct execution order (FIFO)
- **System Behavior**:
  - All instructions completed by cycle 35
  - Queue design ensured forward progress (no deadlock)
    * Guaranteed by: (1) finite execution latency for all instructions, and
    * (2) strict FIFO processing of queued instructions

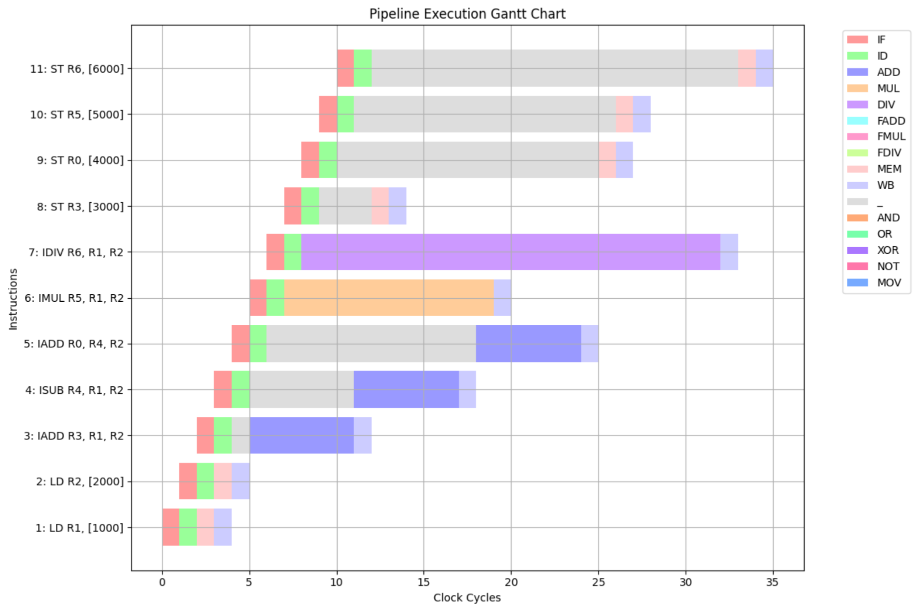

Figure 1: Pipeline Execution Gantt Chart showing dynamic scheduling with Tomasulo's algorithm.

Execution Timeline

The Gantt chart shows the pipeline stages for each instruction:
Key observations:

- Instructions begin execution as soon as operands are available
- Multiple instructions can be in flight simultaneously
- Long latency operations (like IDIV) overlap with other computations
- Memory operations proceed as soon as reservation stations are available

Comparison with VLIW Implementation

The same assembly program was executed on both VLIW and Tomasulo processor simulators, revealing significant differences in performance and implementation approach:

Performance Comparison

- **VLIW Execution Time**: 47 clock cycles
- **Tomasulo Execution Time**: 35 clock cycles (25.53% faster)

- **NOP Instructions**:
  - VLIW required 33 NOP instructions (74.47% of total instructions)
  - Tomasulo required no NOPs (dynamic scheduling eliminates need)

Execution Timeline Comparison

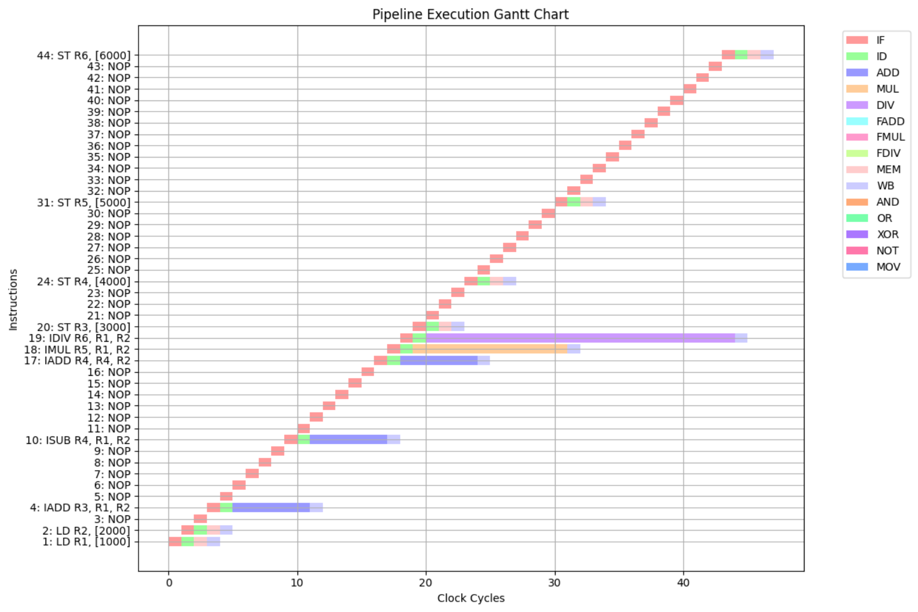

Figure 2: VLIW Pipeline Execution Gantt Chart showing static scheduling with compiler-inserted NOPs.

Key observations from the VLIW execution:

- Extensive NOP insertion (33 NOPs visible in the chart)
- Fixed instruction bundles with explicit stalls
- Long latency operations (IDIV) block subsequent instructions
- Memory operations must wait for all dependencies to resolve

Metric	**VLIW**	**Tomasulo**	**Improvement**
Total Clock Cycles	47	35	25.53% faster
NOP Instructions	33	0	100% reduction
Memory Operations Completed	Cycle 47	Cycle 35	12 cycles earlier
Longest Dependency Chain	IDIV (24 cycles)	IDIV (24 cycles)	Equal

Architectural Differences

VLIW Approach	**Tomasulo Approach**
Static scheduling at compile time	Dynamic scheduling at runtime
Compiler inserts NOPs for hazards	Hardware handles hazards via reservation stations
Fixed instruction bundles	Flexible instruction issue
Explicit WAW hazard handling	Implicit WAW elimination via register renaming
Simple hardware, complex compiler	Complex hardware, simpler compiler

Key Advantages of Tomasulo

- **Better Resource Utilization**:
  - Dynamic scheduling keeps units busy by finding independent instructions
- **Hazard Handling**:
  - Complete elimination of NOP instructions (33 to 0)
  - Uses reservation stations to buffer operations
  - Register renaming eliminates WAW/WAR hazards
- **Memory Operations**:
  - Memory operations complete 12 cycles earlier
  - Can overlap memory operations when no true dependencies exist

Why Tomasulo Performs Better

The 12-cycle improvement (25.53%) comes from several factors:

- **Elimination of NOPs**: Complete removal of 33 pipeline bubbles
- **Register Renaming**: Enables more instructions in flight
- **Memory Operation Overlap**: Stores begin earlier when operands ready
- **Queue Efficiency**:
  - The unbounded queue implementation guarantees:
    * No structural hazards from limited reservation stations
    * Instructions never stall waiting for station availability
    * Maintains correct execution order (FIFO per functional unit)
  - In real hardware, this would require large physical queues

- **Limitation**: The current implementation only detects register spills but does not automatically resolve them through spilling to memory. When register pressure becomes too high, the simulator reports "Register Spill" and terminates rather than implementing spill code.

Tradeoffs

- **VLIW Advantages**:
  - Simpler hardware design
  - Predictable timing (important for real-time systems)
  - Power efficient (no complex scheduling logic)
- **Tomasulo Advantages**:
  - Better performance for complex dependency patterns
  - Handles unpredictable latencies better
  - No need for sophisticated compiler scheduling

This comparison demonstrates how dynamic scheduling with Tomasulo's algorithm provides better performance (25.53% faster) and efficiency than static VLIW scheduling for programs with complex dependency patterns, at the cost of more complex hardware implementation.

Conclusion

This report compared two processor architectures:

- **VLIW**: Static scheduling (47 cycles) with compiler-inserted NOPs. Simple hardware but complex compiler.
- **Tomasulo**: Dynamic scheduling (35 cycles, 25.53% faster) using reservation stations. Complex hardware but simpler compiler. Currently detects but doesn't resolve register spills.

Key advantages of Tomasulo:

- Eliminates all NOPs (33 to 0)
- 25.53% faster execution
- Better performance for complex dependencies

While VLIW offers simplicity and predictability, Tomasulo provides superior performance. The optimal choice depends on application needs and compiler capabilities. Future work should implement register spill resolution.