Microarchitectural Challenges in Developing a Dual-Issue RISC-V Tomasulo Simulator for Base and Floating-Point Extensions

The development of a high-fidelity, dual-issue superscalar simulator employing Tomasulo's algorithm necessitates meticulous attention to complex control logic and data path interactions, particularly when incorporating the RISC-V Base Integer ISA (RV32I/RV64I) alongside the Floating-Point (F/D) extensions. Dynamic scheduling enhances Instruction-Level Parallelism (ILP) by allowing instructions to execute out-of-order, but this paradigm introduces substantial hardware complexity, particularly in managing parallel instruction streams, disparate execution latencies, memory coherence, and precise speculative state recovery. The following report details the specific microarchitectural components and subtle implementation minutiae that a simulator developer must accurately model to ensure performance fidelity.

I. Foundational Dual-Issue Dynamic Scheduling: The Issue and Rename Pipeline Stages

Modern implementations of Tomasulo's algorithm use out-of-order execution enabled by register renaming and reservation stations, coupled with an in-order front-end issue pipeline.[1] Introducing dual-issue capability in the front-end—allowing two instructions, I1 and I2, to enter the system simultaneously—multiplies the complexity of resource allocation and dependency resolution that occurs during the Issue and Rename stages.

1.1. Architectural Components for Tomasulo on RISC-V

Accurate simulation requires precise modeling of the key distributed control structures.[2] The

processor state must be maintained across four critical structures:

The Reorder Buffer (ROB) and Physical Registers

The Reorder Buffer (ROB) is essential for maintaining strict program order, crucial for guaranteeing precise exceptions and accurate state commitment.[3] In sophisticated modern implementations of Tomasulo's algorithm, ROB slots often double as physical registers, holding speculative results until they are written back to the Architectural Register File (ARF) upon commitment.[5] A dual-issue pipeline requires allocating two ROB slots per cycle, which can introduce a structural hazard if the ROB capacity is saturated.[7]

Register Alias Tables (RATs)

Register renaming is fundamental to eliminating false dependencies (Write-After-Read (WAR) and Write-After-Write (WAW) hazards).[8] RISC-V's architecture, with its dedicated 32 integer registers ($x0–x31$) and 32 floating-point registers ($f0–f31$) for the F/D extensions, necessitates two parallel Register Alias Tables (RATs): one for the integer domain (IRAT) and one for the floating-point domain (FRAT).[6] These RATs map the architectural register name to the physical register tag (typically the ROB index) that will hold the newest value.[11]

The independence of the integer and floating-point register domains is generally advantageous, isolating WAW/WAR hazards to within each domain. However, RISC-V includes instructions (such as FCVT, FMV, and memory access instructions involving register transfers) that bridge these domains. These instructions introduce **cross-domain RAW hazards** that complicate the dual-issue pipeline check. For instance, if I1 is an integer operation writing to $x5$ and I2 is an FCVT reading $x5$ (and writing to an f register), I2 must receive the destination tag generated by I1 during the same issue cycle. This requires the renaming logic to simultaneously look up both RATs for both instructions and resolve potential dependencies across the boundary. The simulator must implement specialized logic to detect and forward tags between the IRAT and FRAT during the dual renaming process.

Reservation Stations (RS) and Load/Store Queue (LSQ)

Reservation Stations are distributed buffers associated with functional units (FUs). They hold

instructions, their operation codes (Op), and their source operands (Vj/Vk) or the tags (Qj/Qk) of the instructions that will produce those operands.[12] The RS autonomously monitors the Common Data Bus (CDB) and initiates execution when all operands are available.[2] The Load/Store Queue (LSQ) serves a similar purpose for memory operations, managing address generation, memory disambiguation, and store-load forwarding.[14]

1.2. Dual-Issue Front-End Logic and Structural Hazards

The dual-issue capability mandates that the front-end can fetch and decode two instructions (I1 and I2) per cycle.[7] This requires 64-bit fetch bandwidth for 32-bit instructions.[16] The primary challenge is allocating the necessary resources (ROB slots, LSQ slots, and RS slots) for both instructions simultaneously.

Contention Resolution at Dispatch

During the Issue/Dispatch stage, I1 (the older instruction) and I2 (the younger) simultaneously request resources.[7] If the pool of available resources (e.g., reservation stations for a specific type of FU, or ROB/LSQ entries) is constrained, contention arises. A robust simulation must model prioritized arbitration based strictly on program order: I1 must be prioritized over I2.[7]

If a resource is available for I1 but not I2 (or vice versa), the logic must resolve the issue. If the core has a fixed dual-issue width, then if only one ROB slot or one specific RS slot remains, I2 might be stalled. If I1 requires two distinct resources (e.g., an Integer RS and an LSQ entry) and only one is available, I1 stalls.

Structural Hazard Cascading in Dual-Issue

A subtle but performance-critical detail is the strict requirement for in-order issue in Tomasulo's architecture to maintain control flow and renaming correctness.[1] If I1 cannot issue due to any structural hazard—such as waiting for the last available RS of its type to become free, or waiting for an available ROB slot—then I2, despite being completely independent of I1 and having its own necessary resources available, is conservatively stalled.[7]

This conservative stalling behavior in the front-end prevents the younger instruction (I2) from bypassing the older instruction (I1) into the out-of-order kernel. Consequently, an apparently minor structural hazard affecting a single instruction (I1) can result in a cycle with zero Instructions Per Cycle (IPC), severely limiting the achievable ILP. The simulation must precisely model the front-end instruction queue and its stall signaling mechanism to accurately capture this pipeline bubble propagation.

1.3. Dependency Checking and Register Renaming Complexity

Tomasulo's core mechanism dynamically resolves Read-After-Write (RAW) hazards by tracking the tags of producing instructions.[9] In a dual-issue pipeline, the dependency check must resolve RAW hazards not only against all prior in-flight instructions but also against the instruction issuing simultaneously (I1 vs. I2).

The Critical Path: Intra-cycle RAW Dependency Handling

The necessity for I2 to receive the tag generated by I1 within the same cycle defines a crucial critical path in the rename logic.[17] If I1 writes to register R_X and I2 reads R_X, the sequence of events in a single cycle must be:

1. I1 determines that R_X has an old tag (T_old) currently mapped in the RAT.
2. I1 allocates a new physical register tag (T_new) (e.g., a new ROB index) for its destination R_X.
3. I1 updates the RAT to map R_X to T_new.
4. I2 looks up its source register R_X in the RAT. Since I2 is younger and dependent on I1, it must observe the update performed by I1 within the same cycle.

If I2 reads the state of the RAT *before* I1 updates it, I2 will incorrectly receive T_old (or an actual value, if R_X was ready), leading to a logic error or a stall. Therefore, the implementation requires a dedicated, high-speed bypass path—often termed **Chained Renaming Logic**—from I1's renaming output directly to I2's source operand tag input. This logic ensures that if an older instruction (I1) generates a tag that a younger instruction (I2) needs, I2 receives that tag immediately.[17]

The stringent timing requirements imposed by this required single-cycle tag chaining makes the rename stage a highly complex combinatorial logic block, often limiting the maximum clock frequency of the processor core. A high-fidelity simulator must not abstract this away

as an instantaneous process but must rather explicitly track the internal dependency chain ($I1$ Renaming $\rightarrow I2$ Operand Look-up) and the ensuing scheduling impact.

II. Execution and Floating-Point Pipeline Modeling (F/D Extension Details)

The complexity of dynamic scheduling increases significantly with the introduction of Floating-Point (F/D) instructions due to their non-uniform, multi-cycle latencies and the increased contention they generate at the Common Data Bus (CDB).[10]

2.1. Reservation Station (RS) State Management and Dispatch

Reservation stations distribute the control of instruction execution.[12] Each reservation station monitors the CDB, waiting for the results (value and tag) that correspond to its pending source operands (Qj, Qk).[13]

Specialized Functional Units (FUs)

With the F/D extension, the reservation stations must be specialized to feed their corresponding functional units (FUs), such as separate pools for FP Add/Subtract, FP Multiply, and FP Divide/Square Root.[13]

Arbitration for Functional Unit Access

When an instruction in an RS becomes ready (all required operands are available), it attempts to issue to its functional unit.[2] If multiple instructions are ready in the same RS pool simultaneously, or if the FU is capable of accepting only one instruction per cycle (even if pipelined), a structural hazard arises.[20] The distributed control logic must enforce strict arbitration based on program order (age). If an instruction fails arbitration because an older,

ready instruction is selected to proceed, the losing instruction must stall in the RS, delaying the start of its execution cycle (EX).[20]

2.2. RISC-V F/D Variable Latency Modeling

Floating-point operations introduce heterogeneous execution latencies, which is the primary driver of out-of-order completion.[4] This variance must be meticulously modeled to capture performance accurately.[18]

Modeling Pipelined vs. Unpipelined Units

The simulator must distinguish between two types of functional units:

1. **Fully Pipelined Units (e.g., FP Add, FP Multiply):** These units can accept a new instruction every cycle, even while executing previous instructions.[18] Dependent instructions can quickly consume results broadcast on the CDB.
2. **Unpipelined Units (e.g., FP Divide, FP Square Root):** These units occupy the entire resource for the duration of the execution latency. For RISC-V double-precision division, this latency can be substantial (e.g., 35 or more cycles).[18] During this time, the RS entry remains allocated until the instruction writes its result, and subsequent FP division operations must stall at the issue stage if the RS is full.

The widely varying instruction latencies, particularly the long execution time of FP division, guarantee that instructions complete significantly out of program order.[21] This difference in completion time expands the necessary instruction window size (ROB capacity) to ensure the processor can find sufficient independent instructions to keep the execution units busy. The dual-issue capability exacerbates this effect by placing even more instructions into the pipeline speculatively.

The simulation parameters for F/D operations must include precise latency and throughput figures:

RISC-V F/D Latency Model (Simulation Parameters)

RISC-V F/D Operation	Functional Unit	Execution Latency (Cycles)	Issue Throughput (Cycles)

ADD.D, SUB.D	FP Adder	3	1 (Fully Pipelined)
MUL.D	FP Multiplier	8	1 (Fully Pipelined)
DIV.D	FP Divider	35	35 (Unpipelined/Blocked)
FCVT, FMV	FP/Int Converter	1-2	1

The disparity between a 3-cycle FP add and a 35-cycle FP divide significantly increases the probability that multiple results from unrelated, short-latency FUs will finish execution and attempt to write back simultaneously, leading directly to contention for the Common Data Bus.

2.3. Writeback Stage and Common Data Bus (CDB) Arbitration

The Writeback (WB) stage is where an instruction finishes execution and broadcasts its result onto the CDB. The CDB is a central innovation of Tomasulo's algorithm, acting as a shared bypass network that allows results to be immediately consumed by all waiting functional units and the ROB.[2] This snooping mechanism eliminates many RAW hazards by forwarding data before it reaches the architectural register file.[12]

CDB Arbitration Structural Hazard

In a dual-issue, dynamically scheduled core with multiple, variable-latency execution units, it is highly probable that two or more functional units (e.g., an Integer ALU and the FP Adder) will finish execution and attempt to utilize the single CDB resource in the same cycle.[2] This contention is a severe structural hazard.

If the simulator models only a single CDB (a common restriction in early Tomasulo implementations), a precise arbitration mechanism must be employed to select a single winner for the cycle. This arbitration must prioritize the oldest instruction to ensure forward progress of instructions in program order.[7]

The Ripple Effect of CDB Stall

If an instruction loses CDB arbitration, it must stall in the Writeback stage, retaining its result until the next cycle when it successfully gains control of the bus. This seemingly small delay has a performance-wide consequence: all instructions waiting in the reservation stations that are dependent on the stalled instruction cannot receive their operands, initiating a RAW stall that propagates immediately backward through the execution window.[5]

For instance, if a long-latency FP operation and a short-latency integer load address calculation both finish execution and contend for the CDB, and the integer operation loses, the Load/Store Queue (LSQ) entry waiting for that address tag stalls. This delay in memory address calculation can hold up all subsequent memory operations, including independent ones, reducing memory access parallelism. Accurately modeling this CDB contention and its ripple effect is essential, as the arbitration delay transitions from a structural hazard in the WB stage into a pipeline stall for RAW dependencies throughout the RS.

III. Memory Disambiguation and Store-Load Forwarding

Memory operations (loads and stores) present the most significant challenge to out-of-order execution, as they must maintain sequential memory semantics while executing speculatively. The Load/Store Queue (LSQ) manages this complexity by tracking addresses and data for all in-flight memory instructions.[14]

3.1. The Load/Store Queue (LSQ) Architecture

The LSQ logically comprises the Load Queue (LDQ) and the Store Queue (STQ). All memory operations allocate an LSQ entry at the issue stage, maintaining their program order relative to each other.[14]

Store Instruction Segmentation (STA/STD)

RISC-V stores (e.g., SD, SW) typically require two distinct steps, which are often modeled as separate micro-operations (UOPs) to facilitate maximal out-of-order execution [14]:

1. **Store Address Generation (STA):** Calculates the effective memory address. This UOP executes on the Integer ALU and updates the address field in the corresponding STQ entry.
2. **Store Data Generation (STD):** Waits for the data register value (which might be produced by an earlier instruction via the CDB) and buffers the store data within the STQ entry.

Both STA and STD UOPs can execute out-of-order relative to each other and relative to other instructions in the pipeline, allowing the address calculation to proceed early, even if the data is delayed.[14] Crucially, the actual write to the cache hierarchy only occurs when the store instruction reaches the head of the ROB and commits, preserving precise state.[22]

3.2. Program Order and Memory Dependence Checking

The LSQ's primary role is to enforce memory ordering. A younger load can execute before an older store only if the LSQ can guarantee there is no address conflict between them.

Memory Disambiguation Logic

When a load instruction (L.D) is ready to execute, the LSQ searches the STQ for all older, uncommitted stores. The load can proceed only if one of the following is true:

1. The addresses of all older stores are known, and none of them overlap with the load's address.[23]
2. An older store exists at the exact same address, and its data is available for forwarding (Store-Load Forwarding, Section 3.3).

Handling Unknown Address Hazards

The most common performance bottleneck in the LSQ is the conservative stall enforced by unknown addresses.[24] If a load reaches the LSQ, and an older store (or sequence of stores) has not yet completed its Store Address (STA) UOP, the system cannot guarantee safety. Lacking definitive address information, the load must conservatively stall, assuming a potential dependency.[23]

The conservative stall can introduce significant latency, potentially masking the performance benefits of out-of-order execution.[25] The duration of this stall is directly dependent on how long it takes for the older store's address generation instruction to execute and broadcast its result. The simulator must precisely track the age (program order) of STA UOPs and the waiting status of subsequent loads.

3.3. Store-Load Forwarding Implementation Details

Store-Load Forwarding is essential for high performance. It allows a younger load to bypass the latency of writing data to and then reading it from the L1 cache by directly retrieving the data from an older, pending store entry within the STQ.[26]

Address Overlap Detection

The core challenge of forwarding is determining if the memory regions accessed by the load and the store overlap.[27] The load’s effective address and size must be compared against the address and size of all older, address-known stores in the STQ.[28]

Byte-Level Granularity and Forwarding Failure

The most minute implementation detail lies in handling unaligned or partially overlapping memory accesses, which are common in RISC-V due to its support for various data sizes (byte, half, word, double) and optional unaligned access support for F/D operations.[24]

Forwarding succeeds efficiently only if the load size and address exactly match the store size

and address.[26]

- **Unaligned/Partial Overlap Logic:** If a store operation writes a smaller amount of data than a subsequent load reads, or if the addresses are aligned such that only partial byte overlap exists, complex byte-masking and merging logic is required in the LSQ to construct the correct load data from the buffered store data.[29]
- **Forwarding Failure:** If the overlap is complex, such as a narrow store followed by a wide load that crosses cache lines, or if the load requires merging data from multiple prior stores, the hardware often defaults to a conservative failure state.[26]
    - When forwarding fails, the load must stall until the corresponding store instruction commits to the L1 cache. This incurs a significant multi-cycle penalty (often 10-15 cycles or more).[26]

Multicycle Forwarding and Ordering Reconstruction

If a load address matches multiple older stores (e.g., three sequential byte stores followed by a word load from the same address range), the LSQ forwarding logic must reconstruct the full word by combining the data from the individual STQ entries, respecting program order.[29] This requires sequential access to the STQ entries and data synthesis logic, confirming that memory access, even when forwarded, may not be a single-cycle operation within the LSQ.[29] The simulator must model the latency overhead incurred by this multi-cycle reconstruction process to accurately reflect real-world performance penalties.

IV. Commitment and Speculative Recovery (Branch Resolution Focus)

The final architectural stages involve committing state to the architectural registers and memory, which must occur in program order to maintain correctness. This is managed by the Reorder Buffer (ROB) and is critical for recovering from speculation errors, particularly branch mispredictions.

4.1. Reorder Buffer (ROB) Functionality and State Tracking

The ROB guarantees sequential consistency.[3] Instructions complete their out-of-order execution, write their results back to their ROB slot via the CDB (Writeback stage), and are marked 'Ready'.[9]

ROB Head Advancement

Instructions commit only when they reach the head of the ROB and are marked 'Ready.' In a dual-issue design, the ROB must support retiring up to two instructions per cycle.[7]

Store Commitment

Store instructions (S.D, S.W, etc.) wait at the ROB head until commitment. Unlike ALU operations, stores do not write to the architectural state until this point. Once committed, the store is permitted to write its data (which has been buffered in the STQ) into the L1 cache.[22] This in-order store retirement prevents the memory system from seeing speculative writes.

To support precise state recovery, the ROB entry must track all necessary information to undo speculative changes. Specifically for the RISC-V F/D simulator, this includes:

Reorder Buffer (ROB) Entry State Fields

Field	Purpose	Critical Role in Speculation
Destination Tag	Physical Register/Architectural Register Index	Identifies the register to be updated upon commit [6]
Exception/FCSR Payload	Pending FP status flags (NV, OF, INX, etc.)	Ensures precise update of the Floating-Point Control and Status Register [2]
Branch Metadata	PC, Predicted Target, Actual Target	Used to verify prediction accuracy at the time of

		commit [31]
Old Map Tag	Previous tag mapped to the destination register	Essential for restoring the Register Alias Table (RAT) upon misprediction [6]

4.2. Branch Resolution at ROB Retirement (Commit Stage)

The user specified that branch resolution occurs at the ROB retirement stage. This design choice guarantees that speculation is managed strictly in program order, simplifying recovery mechanisms but potentially increasing latency penalties.[4]

Verification and Misprediction Detection

When a branch instruction reaches the ROB head and is ready to commit, the ROB compares the outcome calculated during the Execution stage against the outcome predicted by the front-end (Branch Prediction Unit).[31] If the branch was mispredicted, a misprediction signal is generated, initiating the pipeline squash.[32]

Pipeline Recovery Penalty

A critical factor impacting performance is the variable latency before recovery can start.[32] If a long-latency instruction, such as a cache-missing load or a 35-cycle FP divide, precedes the mispredicted branch in the ROB, the branch must wait for that instruction to complete and commit before it can reach the ROB head and detect the misprediction.[31]

The time taken to retire instructions preceding a mispredicted branch directly influences the branch misprediction penalty. If the branch is delayed by a long-latency load or FP operation, the time spent executing on the incorrect path increases significantly. The simulator must accurately track the dependency chain and completion status of all instructions older than the branch to determine the exact cycle the misprediction signal is generated.

4.3. State Restoration Protocol

Upon misprediction, the processor must instantly discard all effects of instructions executed on the incorrect path and restore the architectural state to the point *before* the branch.[4]

Pipeline Flushing

A squash or kill signal must be immediately broadcast throughout the pipeline structures for all instructions younger than the mispredicted branch [31]:

- The Fetch Unit must be redirected to the correct target address.
- Reservation Stations must release their allocated entries and invalidate source tags for speculative instructions.
- The LSQ must invalidate speculative loads and stores.
- The ROB must mark all younger entries as invalid and release their physical registers.

RAT/Map Table Recovery

To achieve fast recovery, the processor must restore the architectural map (RAT state) instantly. The required mechanism involves recovering the map to the state it held immediately before the mispredicted branch was issued.[6] This recovery is facilitated by the **Old Map Tag** stored in the ROB entry of the mispredicted branch. By restoring the mapping from the ROB's saved state, the processor can instantly discard the results of all instructions on the wrong path, avoiding a time-consuming sequential walk through the ROB.

4.4. Precise Handling of the FCSR (F/D Requirement)

The Floating-Point Control and Status Register (FCSR) is used in RISC-V (as part of the F/D extensions) to hold control bits (rounding mode) and exception flags (e.g., inexact, overflow, underflow).[2] Since FP instructions execute out-of-order, handling FCSR updates precisely is

crucial.

FCSR Speculative Update and Payload Tracking

FP instructions calculate their exception flags speculatively during the Execute stage. These flags must not update the architectural FCSR directly. Instead, they are bundled as a status payload with the instruction's computed result and stored in the instruction's corresponding ROB entry.[2]

FCSR Commitment as a Serializing Bottleneck

The architectural FCSR must only be updated when the FP instruction commits at the head of the ROB, ensuring that the cumulative flag state reflects sequential program execution.[2]

The consequence of this precision requirement is that the FCSR becomes a serialization point. Any subsequent instruction that reads or modifies the FCSR (e.g., using a FCLAS or FSRM instruction) must effectively stall its commitment until all prior FP instructions that might affect the status flags have committed and updated the FCSR. This forces a synchronization point that overrides the benefits of out-of-order scheduling for FP-heavy code that frequently checks status flags, creating a significant, unavoidable performance bottleneck that must be accurately modeled in the simulator.

V. Conclusions and Synthesis of Microarchitectural Interactions

Developing a dual-issue RISC-V simulator based on Tomasulo's algorithm requires modeling intricate interactions between highly parallel, distributed structures. The simultaneous issuance of two instructions, coupled with the variable latencies introduced by the F/D extensions, creates unique pressure points in the microarchitecture.

The analysis demonstrates that performance bottlenecks are frequently caused not by a lack of parallelism in the execution units, but by structural hazards and control dependencies in the distributed management hardware:

1. **Front-End Serial Dependencies:** The need for **intra-cycle RAW dependency checking** (I1 providing a tag to I2) introduces a critical timing constraint on the renaming logic, potentially limiting clock frequency.
2. **Back-End Congestion:** The **CDB arbitration structural hazard**, amplified by the latency mismatch of F/D operations, generates cascading RAW hazards throughout the Reservation Stations when results are delayed from writing back.
3. **Memory Access Latency:** The requirement for **conservative stalling** when an older store address is unknown significantly delays loads, overriding the benefits of out-of-order execution in memory-intensive code sections. Furthermore, complex **multicycle forwarding** logic is necessary to reconstruct partially overwritten data in the LSQ, confirming that LSQ operations are not uniformly single-cycle.
4. **Control Serialization:** The need for **precise FCSR commitment** introduces a synchronization barrier at the ROB head, forcing sequential execution for status register operations. This fundamentally limits ILP exploitation in scenarios where floating-point status flags are actively monitored.
5. **Recovery Complexity:** The performance penalty of branch misprediction is not constant; it is highly variable based on the latency of instructions preceding the branch in the ROB. Efficient state recovery depends on sophisticated techniques like RAT checkpointing to instantly restore architectural state upon misprediction detection.

Accurate simulation of this dual-issue superscalar core relies heavily on modeling these minute details—the capacity limits of the structural queues (ROB, RS, LSQ), the strict age-based arbitration protocols (for RS dispatch, CDB access, and LSQ ordering), and the cycle costs associated with complex operations like memory forwarding and branch recovery. Abstracting these specific control mechanisms will inevitably lead to an overestimation of achievable Instructions Per Cycle (IPC).

Works cited

1. CS 5513 Lecture 5, accessed November 23, 2025, https://people.engr.tamu.edu/djimenez/utsa/cs5513/lecture5.ppt
2. Tomasulo's algorithm - Wikipedia, accessed November 23, 2025, https://en.wikipedia.org/wiki/Tomasulo%27s_algorithm
3. Instruction-Level Parallelism - Jyotiprakash's Blog, accessed November 23, 2025, https://blog.jyotiprakash.org/instruction-level-parallelism
4. Out-of-order execution - Wikipedia, accessed November 23, 2025, https://en.wikipedia.org/wiki/Out-of-order_execution
5. PPT, accessed November 23, 2025, https://www.engineering.iastate.edu/~zzhang/courses/cpre585-f04/slides/lecture7.ppt
6. The Rename Stage - RISCV-BOOM documentation, accessed November 23, 2025, https://docs.boom-core.org/en/latest/sections/rename-stage.html
7. Solved 1- Tomasulo's Algorithm with Dual Issue and | Chegg.com, accessed November 23, 2025,

https://www.chegg.com/homework-help/questions-and-answers/1-tomasulo-s-algorithm-dual-issue-speculation-consider-following-architectural-assumptions-q115547029
8. Register renaming, accessed November 23, 2025, https://cs.ijs.si/courses/processor/Chapter3/tsld029.htm
9. SOLUTIONS - People @EECS, accessed November 23, 2025, https://people.eecs.berkeley.edu/~kubitron/courses/cs252-S10/handouts/oldquiz/sp07-quiz1_soln.pdf
10. How exactly are RISC-V extensions like F implemented in a pipelined processor, accessed November 23, 2025, https://stackoverflow.com/questions/77902728/how-exactly-are-risc-v-extensions-like-f-implemented-in-a-pipelined-processor
11. Register renaming - Wikipedia, accessed November 23, 2025, https://en.wikipedia.org/wiki/Register_renaming
12. Out-of-Order Execution - Washington, accessed November 23, 2025, https://courses.cs.washington.edu/courses/cse471/15sp/lectures/tomasuloStudents.pdf
13. L11 Tomasulo, accessed November 23, 2025, https://cseweb.ucsd.edu/classes/fa99/cse240/ilp2.pdf
14. The Load/Store Unit (LSU) - RISCV-BOOM documentation, accessed November 23, 2025, https://docs.boom-core.org/en/latest/sections/load-store-unit.html
15. What exactly is a dual-issue processor? - Stack Overflow, accessed November 23, 2025, https://stackoverflow.com/questions/8014739/what-exactly-is-a-dual-issue-processor
16. Multiple Issue Superscalar Processors, accessed November 23, 2025, https://people.cs.pitt.edu/~childers/CS2410/slides/lect-multiple-issue.pdf
17. How efficient is register renaming? - Computer Science Stack Exchange, accessed November 23, 2025, https://cs.stackexchange.com/questions/156576/how-efficient-is-register-renaming
18. Lecture 6: Tomasulo Algorithm (II), accessed November 23, 2025, https://home.engineering.iastate.edu/zzhang/courses-bak/cpre585-f03/slides/lecture6.pdf
19. The Execute Pipeline - RISCV-BOOM documentation, accessed November 23, 2025, https://docs.boom-core.org/en/latest/sections/execution-stages.html
20. Description of Tomasulo Based Machine, accessed November 23, 2025, https://cseweb.ucsd.edu/classes/fa07/cse240a/Papers/tomasulo.pdf
21. Lecuture 6: Tomasulo Scheduling - People @EECS, accessed November 23, 2025, https://people.eecs.berkeley.edu/~kubitron/courses/cs252-F00/lectures/lec06-dynasched.ppt
22. 2. Out-of-Order Core Microarchitecture — MARSS-RISCV 4.1a documentation, accessed November 23, 2025, https://marss-riscv-docs.readthedocs.io/en/latest/sections/oocore-microarch.html

23. Memory disambiguation - Wikipedia, accessed November 23, 2025, https://en.wikipedia.org/wiki/Memory_disambiguation
24. How does store to load forwarding happens in case of unaligned memory access?, accessed November 23, 2025, https://stackoverflow.com/questions/42210733/how-does-store-to-load-forwarding-happens-in-case-of-unaligned-memory-access
25. Out-of-Order Memory Accesses Using a Load Wait Buffer - Carnegie Mellon University, accessed November 23, 2025, https://users.ece.cmu.edu/~schen1/18-741_final_report.pdf
26. Store forwarding by example. | Easyperf, accessed November 23, 2025, https://easyperf.net/blog/2018/03/09/Store-forwarding
27. What's the most efficient way to test if two ranges overlap? - Stack Overflow, accessed November 23, 2025, https://stackoverflow.com/questions/3269434/whats-the-most-efficient-way-to-test-if-two-ranges-overlap
28. Late-Binding: Enabling Unordered Load-Store Queues - Computer Science at Columbia University, accessed November 23, 2025, http://www.cs.columbia.edu/~simha/cal/pubs/pdfs/ulsq.pdf
29. Late-Binding: Enabling Unordered Load-Store Queues - UT Computer Science, accessed November 23, 2025, https://www.cs.utexas.edu/~skeckler/pubs/isca07.pdf
30. Auto-Vectorization and Store-to-Load-Forwarding | Emanuel's Blog - GitHub Pages, accessed November 23, 2025, https://eme64.github.io/blog/2024/06/24/Auto-Vectorization-and-Store-to-Load-Forwarding.html
31. How are branch mispredictions handled before a hardware interrupt - Stack Overflow, accessed November 23, 2025, https://stackoverflow.com/questions/54422950/how-are-branch-mispredictions-handled-before-a-hardware-interrupt
32. MOWER : A NEW DESIGN FOR NON-BLOCKING MISPREDICTION RECOVERY - Digital Commons @ Michigan Tech, accessed November 23, 2025, https://digitalcommons.mtu.edu/cgi/viewcontent.cgi?article=1916&context=etds