Cycle-Accurate Out-of-Order Processor Simulator in C++

Superscalar 9-stage OoO pipeline · register renaming via RMT + ROB-tag physical naming · oldest-first issue · multi-cycle pipelined FUs · NCSU ECE 563 (Microprocessor Architecture, Prof. Rotenberg) · Fall 2025 · Project 3

Topics: *OoO microarchitecture · fetch/decode/rename/dispatch/issue/execute/writeback/retire pipeline · Rename Map Table · Reorder Buffer · Issue Queue · oldest-first issue · wakeup-select logic · multi-cycle pipelined function units · structural hazards · IPC sensitivity analysis · cycle-accurate simulation in C++*

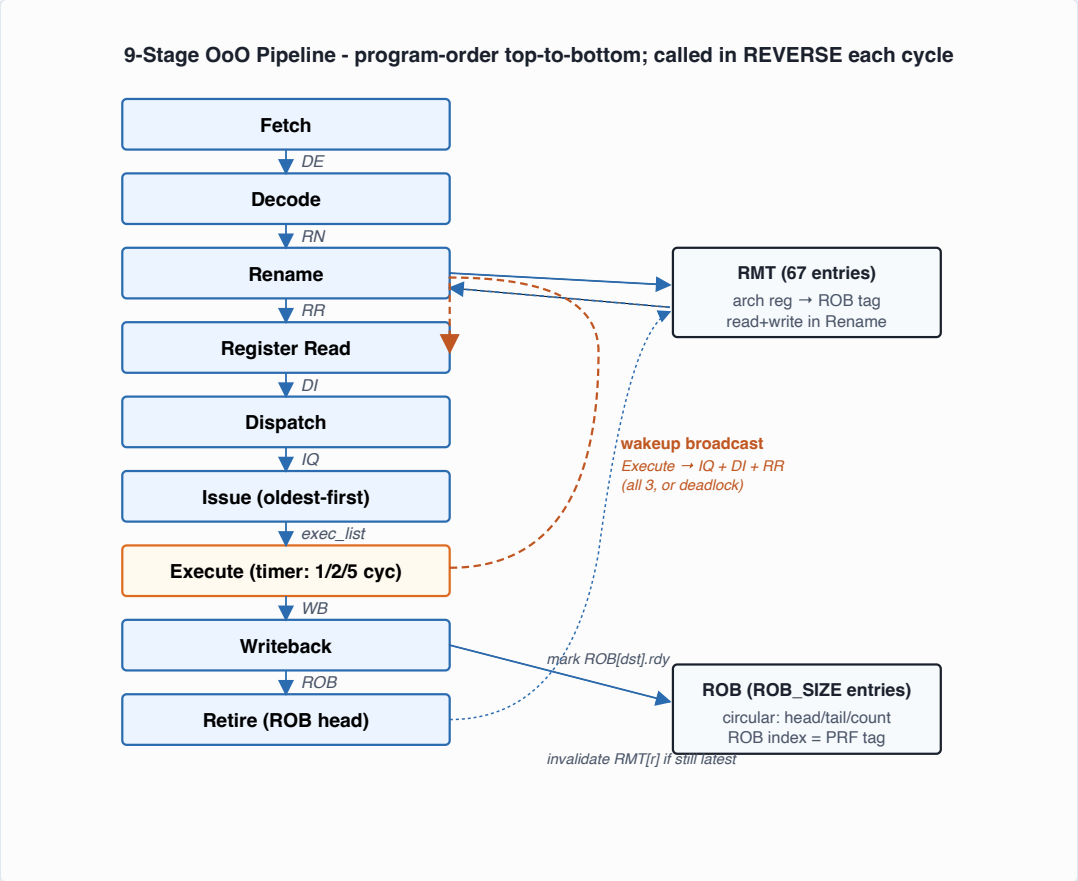
> **The one-liner:** A cycle-accurate, configurable, superscalar out-of-order processor simulator written from scratch in C++ (~648 lines, single `Processor` class). Models a 9-stage OoO pipeline - Fetch / Decode / Rename / Register-Read / Dispatch / Issue / Execute / Writeback / Retire - with a Rename Map Table, Reorder Buffer (also doubling as the physical register file), Issue Queue with oldest-first issue, WIDTH universal pipelined function units (1/2/5-cycle latencies for op types 0/1/2), and full backpressure. Passes all 8 trace-based validation runs (gcc + perl benchmarks across multiple ROB/IQ/WIDTH configurations).
>
> **Why it's interesting:** The OoO scheduling model touches dependency tracking, structural hazards, deadlock avoidance, and IPC analysis all at once, which makes it a dense slice of computer architecture. I wrote every line of this simulator from scratch - no skeleton extension, no provided base code. Worth noting: per the spec, perfect branch prediction and perfect caches are assumed, so this is a dynamic-scheduling deep-dive, not an end-to-end processor model.

Microarchitecture being modeled

The simulator models a 9-stage OoO superscalar pipeline whose width and queue sizes are configurable per run. The trace format is `<PC> <op_type> <dst> <src1> <src2>`, with 67 architectural registers (r0-r66) per the MIPS-like ISA and op types 0/1/2 mapping to 1/2/5-cycle execute latencies.

Parameter	What it controls	Range I tested
`WIDTH`	Superscalar width - max instructions per stage per cycle; also the number of universal pipelined function units	1, 2, 4, 8
`IQ_SIZE`	Issue Queue entries	8, 16, 32, 64, 128, 256
`ROB_SIZE`	Reorder Buffer entries - also the physical register file size in this design	32, 64, 128, 256, 512

Pipeline registers between stages (DE / RN / RR / DI / IQ / execute_list / WB / ROB) hold "bundles" of up to WIDTH instructions - except IQ (sized to `IQ_SIZE`), execute_list and WB (sized to `WIDTH*5` for all in-flight FU sub-stages), and ROB (sized to `ROB_SIZE`). WIDTH universal pipelined FUs run any op type with op-type-0/1/2 latencies of 1/2/5 cycles. Per spec, perfect branch prediction + perfect caches assumed; no LSQ, no actual register values modeled (more detail in Trade-offs).



9-stage OoO pipeline with WIDTH-wide superscalar issue. Pipeline registers (DE/RN/RR/DI/IQ/exec_list/WB/ROB) buffer each inter-stage handoff. The wakeup broadcast from Execute to IQ + DI + RR (orange dashed) is the canonical deadlock-avoidance pattern; ROB doubles as the physical register file via index-as-tag.

Implementation - the Processor class

Everything lives in one file, `sim_proc.cc` (~648 lines). The Processor class owns all pipeline state and exposes one method per pipeline stage. The main loop is trivial: construct the processor with the command-line parameters, then call all 9 stages in reverse order each cycle until Advance_Cycle returns false.

Data structure	Holds
`RMT` (sized 67)	Per-arch-register: valid bit + ROB tag of the most recent producer. Read+written in Rename; invalidated during Retire when the entry is still the latest producer.
`ROB` (sized ROB_SIZE, circular with head/tail/count)	Per-entry: sequence number, dst arch register, ready bit, PC. The ROB index IS the physical register tag - no separate PRF.
`IQ` (sized IQ_SIZE, compacted via iq_count)	Renamed instr per entry: dst tag, src1/src2 tags + ren/rdy flags, op type, seq number. Compaction on issue keeps the queue contiguous from index 0.
`EX` / `WB` (each reserved WIDTH*5)	In-flight executing instructions (each with its own `ex_cycles` countdown timer) and the writeback bundle. Hits 0 → move to WB and broadcast wakeup.
`timing_info` (one per dynamic instr)	Per-stage first-cycle + duration for FE/DE/RN/RR/DI/IS/EX/WB/RT - produces the validation-output format.

The trick that makes the renaming work is that the ROB entry index is the physical-register tag. When Rename allocates ROB entry N to an instruction with dst arch register r, it sets `RMT[r] = {valid=true, rob_tag=N}`. Any later instruction that consumes r looks up `RMT[r]` and gets N - that's the producer it's waiting on. Real high-performance processors (Alpha 21264, MIPS R10K) use a separate physical register file with its own allocation pool; this design collapses the PRF into the ROB.

Stage-by-stage - and why they're called in reverse

The main loop calls all 9 stages in **reverse** order every cycle:

```
do {
  processor.Retire();      // ROB head → free
  processor.Writeback();   // WB → mark ROB ready
  processor.Execute();     // ex_cycles--, hit 0 → WB + wakeup
  processor.Issue();       // IQ → execute_list (oldest-first, WIDTH per cycle)
  processor.Dispatch();    // DI → IQ
  processor.RegRead();     // RR → DI
  processor.Rename();      // RN → RR (alloc ROB, rename via RMT)
  processor.Decode();      // DE → RN
  processor.Fetch(FP);     // trace → DE (WIDTH per cycle)
} while(processor.Advance_Cycle());
```

The reverse order is critical and is the canonical way to simulate hardware pipeline stages in sequential code. Each stage reads from the pipeline register written by the next-younger stage and writes the pipeline register read by the next-older stage. Calling in reverse guarantees that every stage operates on the values from the *previous* cycle - not the current cycle - which matches the physical reality of register-to-register hardware flow.

Each stage uses the same backpressure pattern: "if my output pipeline register is empty AND my input has something, advance." Concretely:

Stage	Guard before advancing	What happens on advance
`Fetch`	DE empty AND trace not exhausted	Read up to WIDTH instructions from the trace file, populate DE, create timing_info entries with FE = num_cycles.
`Decode`	RN empty AND DE not empty	Copy bundle DE → RN, clear DE.
`Rename`	RR empty AND RN not empty AND ROB has space for the full RN bundle	For each instr in RN: allocate ROB entry at tail, rename src1/src2 via `RMT` (keep arch reg if not renamed yet), rename dst by writing `RMT[dst] = rob_tail`, advance to RR. Critically, the *all-or-nothing* ROB-space check preserves bundle atomicity.
`RegRead`	DI empty AND RR not empty	For each instr: if a src is renamed, check if its producer's ROB entry is already ready (it might have written back during this same cycle's earlier Execute call) and mark src_rdy accordingly; if not renamed, src is trivially ready. Advance to DI.
`Dispatch`	DI not empty AND IQ has space for full DI bundle	Copy DI bundle into IQ at iq_count..iq_count+|DI|-1, increment iq_count.
`Issue`	IQ has at least one ready entry	Run up to WIDTH passes: each pass scans IQ for the *oldest* (lowest seq number) entry with both src_rdy bits set, removes it via shift-down compaction, pushes into execute_list with `ex_cycles` = 1/2/5 by op type.
`Execute`	always	Decrement `ex_cycles` on every entry in execute_list. When it hits 0: wakeup dependents in IQ/DI/RR (set src_rdy if their renamed src tag matches this dst), push to WB, remove from execute_list.
`Writeback`	always (operates on whatever is in WB)	For each instr in WB: mark ROB[dst].rdy = true. Clear WB.
`Retire`	ROB head ready	Retire up to WIDTH consecutive ready entries from ROB head. For each: if `RMT[dst].rob_tag == rob_head` (this entry is still the latest producer), invalidate `RMT[dst]`. Advance head, decrement count.

The **stale-rename check at Retire** (the `RMT[dst].rob_tag == rob_head` guard) is the subtle correctness bit. If a later instruction also wrote to the same arch register, `RMT[dst]` was already overwritten to point to the *newer* ROB entry - invalidating it here would lose that producer. The guard ensures only the latest-producer-at-retire-time gets its RMT entry cleared back to "read from ARF."

Wakeup + issue - the two hardest correctness puzzles

Oldest-first issue is implemented with multiple passes through the IQ. Each pass: scan all `iq_count` entries, track the minimum seq number among entries with both src_rdy bits set, then issue that entry. Repeat up to WIDTH times per cycle. Complexity is O(WIDTH × IQ_SIZE) per cycle - fine for the IQ_SIZE range tested (8-256). The alternative (age-sorted insertion) trades insert-time work for issue-time work; for these queue sizes the multi-pass approach is cleaner.

Wakeup broadcasts to three places, not one. When an instruction's last execution cycle completes in Execute(), its dst ROB tag is broadcast to wake up dependent source operands in:

1. **The Issue Queue** - the obvious one. Any IQ entry whose renamed src1 or src2 matches the producer's dst gets src_rdy set.
2. **The Dispatch bundle (DI)** - instructions that were freshly renamed and read this cycle but haven't been dispatched into the IQ yet.
3. **The Register-Read bundle (RR)** - instructions that just finished Rename and are waiting for the RegRead stage.

The spec explicitly requires all three to avoid deadlock. Why? Consider a producer instruction whose `ex_cycles` hits 0 on cycle *N*. A consumer dependent on it was just renamed on cycle *N-1* and currently sits in DI or RR - its src tags point to the producer's ROB index, src_rdy=false. If wakeup only broadcasts to the IQ, the consumer will eventually move from DI → IQ on cycle *N+1*, but by then the wakeup signal is gone - the producer's broadcast has already happened. The consumer's src_rdy stays false forever and the simulator deadlocks. Broadcasting to all three buffers catches the consumer wherever it happens to be at the moment of wakeup.

A subtler same-cycle pattern: RegRead also re-checks producer readiness directly (`ROB[src].rdy`). Because stages are called in reverse and Execute / Writeback run before RegRead in each cycle, by the time RegRead executes, the ROB ready bit reflects writebacks that completed earlier in *this same cycle* - so a producer finishing execution on cycle *N* correctly wakes a consumer entering RR on cycle *N*.

Experiments - IPC sensitivity to IQ_SIZE and ROB_SIZE

Two benchmark traces (gcc, perl). Two experiment sweeps.

Experiment A - Large ROB, sweep IQ_SIZE. Fix ROB_SIZE = 512 (so ROB never bottlenecks), sweep IQ_SIZE ∈ {8, 16, 32, 64, 128, 256}, plot IPC for WIDTH ∈ {1, 2, 4, 8}. Both gcc and perl show the same shape: WIDTH=1 asymptotes at ~0.95 IPC immediately (every IQ_SIZE works). WIDTH=2 asymptotes at ~1.93. WIDTH=4 reaches ~3.85 once IQ_SIZE ≥ 32. WIDTH=8 reaches ~7.7 (gcc) / ~7.5 (perl) but needs IQ_SIZE ≥ 128 to get there.

Optimized IQ_SIZE per WIDTH (smallest IQ_SIZE achieving within 6% of the IPC at IQ_SIZE=256):

WIDTH	gcc	perl
1	8	8
2	16	16
4	32	64
8	64	128

Two readings worth noting:

- **Larger WIDTH demands a larger IQ.** To keep WIDTH execution lanes busy each cycle, the issue logic has to find that many independent ready instructions - which means looking deeper into the dynamic instruction window. A small IQ doesn't hold enough in-flight instructions for the issue logic to find enough independent ones.
- **Perl needs a larger IQ than gcc at WIDTH=8.** Plausibly because perl has more data-dependent instructions per dynamic window (the issue logic must look farther to find the same number of independent ones), and/or perl has more long-latency op-type-2 instructions clogging FUs and forcing the IQ to absorb more in-flight pressure.

Experiment B - Sweep ROB_SIZE at optimized IQ_SIZE. For each WIDTH, use that WIDTH's optimized IQ_SIZE from the table above, sweep ROB_SIZE ∈ {32, 64, 128, 256, 512}. WIDTH=1/2/4 saturate by ROB_SIZE = 128. WIDTH=8 keeps climbing through 256 (gcc reaches ~7.3, perl ~7.0) and 512 (gcc ~7.3, perl ~7.4). The takeaway: ROB becomes the binding constraint exactly when the IQ is sized correctly - the deeper the IQ can look, the more in-flight ROB entries it consumes, and a small ROB starts stalling Rename.

Design decisions and trade-offs

Three limitations of the simulator, and the reasoning behind each.

1. **Perfect branch prediction and perfect caches are assumed.** Per the spec, this project deliberately scopes out everything except dynamic scheduling. There is no BTB, no conditional branch predictor, no I-cache/TLB, no D-cache/TLB, no Load-Store Queue. Real OoO processors spend a huge fraction of

their complexity on those pieces - a Skylake-class core might be 30% scheduling, 70% memory + branch infrastructure. The most natural next addition is an LSQ for memory dependencies.

2. **ROB doubles as the physical register file - there's no separate PRF.** The ROB entry index IS the physical-register tag. This is a teaching simplification. Real high-performance processors (Alpha 21264, MIPS R10K, modern Intel/AMD) use a separate physical register file with its own allocation pool - that way, the number of in-flight rename mappings is decoupled from the number of in-flight instructions awaiting retirement. In this design, those two numbers are tied together at ROB_SIZE.
3. **Register values are not modeled.** Only operation type (which determines execute latency) and register specifiers (which determine dependencies) are tracked. There's no ALU, no functional results, no memory image. The simulator's output is timing - cycle counts and IPC - not data correctness. This is per spec: the model tracks scheduling latency, not data values, so there is no ALU because functional results never affect timing here.

Common questions about this project

Question	Short answer
Why call the pipeline stages in reverse order each cycle?	Each stage reads from the pipeline register written by the next-younger stage. Calling in reverse guarantees every stage operates on the *previous* cycle's values, which matches register-to-register hardware flow. Forward order would let stages see this cycle's downstream advances and double-pump instructions.
How is oldest-first issue implemented?	Sequence numbers assigned at Fetch. Issue makes up to WIDTH passes through the IQ each cycle. Each pass scans all entries with src1_rdy AND src2_rdy, tracks the minimum seq number, then issues that entry. After issue, the IQ is compacted by sliding entries down so iq_count-1 is always the last filled slot. O(WIDTH × IQ_SIZE) per cycle.
Why broadcast wakeup to DI and RR, not just the IQ?	Deadlock avoidance. A consumer instruction freshly renamed last cycle might still be in DI or RR when its producer completes execution. If we only wake up IQ entries, the consumer would move from DI → IQ next cycle with src_rdy still false, having missed the wakeup signal that already fired. It would sit forever waiting for a broadcast that's never coming.
What is in each ROB entry?	Sequence number, dst architectural register, ready bit, PC. No value field - the ROB index itself acts as the physical register tag. Real designs would also have an exception bit, a mispredict bit, and (in PRF designs) a pointer back to the physical register holding the result.
How does the stale-rename check at Retire work?	When retiring ROB entry at head with dst arch register r, I check `RMT[r].rob_tag == rob_head`. If true, I'm the latest producer - invalidate `RMT[r]` so future consumers read from the ARF. If false, a later in-flight instruction also wrote r and `RMT[r]` points to that newer ROB entry - leave it alone. Without this check, retiring an early producer would clobber a still-valid mapping to a later producer.
Where would a Load-Store Queue plug in?	Between Issue and Execute for loads/stores. Memory-op instructions would issue into the LSQ rather than directly to a function unit; address generation happens, then memory disambiguation (does this load alias an older store still pending?), then the actual memory access. Stores commit from the LSQ at Retire. The IQ no longer tracks memory ops once they enter the LSQ; the LSQ has its own oldest-first issue for matured loads.

Question	Short answer
How would branch misprediction handling be added?	Three pieces: a branch predictor in Fetch (BTB + direction predictor) that lets Fetch continue speculatively; rename-time checkpointing of the RMT so a misprediction can roll the rename state back; squash-from-ROB-tail logic that flushes all instructions younger than the mispredicting branch from IQ / DI / RR / DE / RN / execute_list / WB / ROB. The ECE 721 follow-on project covers exactly this on Prof. Rotenberg's full 721sim base.
What was the hardest part?	Getting wakeup right in DI and RR, not just IQ. The first version only broadcast to IQ and ran fine on small traces, then deadlocked on the larger validation runs where same-cycle producer-consumer pairs landed in adjacent pipeline registers. Took a careful re-read of the spec note ("required to avoid deadlock") to realize the broadcast had to fan out to all three buffers. Lesson: when the spec emphasizes something with "required to avoid deadlock," treat it as a hard constraint, not advice.