Out-of-Order RISC-V Processor — Final Project Report

Chenhao Yang cy2822@columbia.edu
Xuepeng Han xh2718@columbia.edu
Gavin Zou cz2931@columbia.edu
Pingchuan Dong pd2827@columbia.edu
Hins Lyu ql2585@columbia.edu
Xueer Qian xq2259@columbia.edu
EECS 4340, Columbia University

***Abstract*—We built a synthesizable P6-style out-of-order RV32IM processor on top of the in-order Project 3 starter pipeline. The base machine is one instruction wide, and the live build adds 2-way superscalar fetch, decode, dispatch, and commit, plus six other features layered above the base: early tag broadcast, gshare, return address stack, store-to-load forwarding, next line prefetch, and a 2-way set-associative D-cache. All 33 test programs pass on both the RTL simulator and the synthesized gate-level netlist, and every writeback trace is byte-identical between the two. Across the suite, the seven advanced features cut cycle count by 27.46% on geomean against the OoO base, and branch prediction accuracy reaches 74.03% on geomean, an arithmetic mean lift of 8.87 percentage points over the bimodal baseline. One open item is that two endpoints in the full pipeline netlist miss the 1000 ps clock by up to 797.58 ps, even though the netlist is functionally bit-equivalent to the RTL on every program.**

I. Introduction

This processor runs RV32IM RISC-V binaries, the same RV32IM subset (the 32-bit base integer ISA plus the M extension for multiply and divide) [4] that the Project 3 in-order pipeline ran. The new part is what happens between fetch and writeback. Instructions can issue and execute out of program order, then commit back in order. In an in-order pipeline, one stalled load freezes every instruction behind it, even instructions that have nothing to do with the load's address or its result. Out-of-order execution lets independent work continue while a slow operation finishes, which is the reason for the design.

The course-supplied starter [3] is an in-order pipeline with a few pieces we kept: a pipelined multiplier from Project 2, an instruction cache, a register file, and a decoder. Everything that makes the design out-of-order is ours. We added a Reorder Buffer (ROB) that holds in-flight instructions and commits them in program order, a Reservation Station (RS) that holds dispatched instructions until their operands are ready, an embedded Register Alias Table (RAT) that handles renaming directly inside the ROB, a Load-Store Queue (LSQ) that orders memory operations, a write-back data cache, and a branch predictor with a Branch Target Buffer (BTB). The Common Data Bus (CDB) ties execution back to the ROB and the waiting consumers. On top of the base design we added seven advanced features: 2-way superscalar fetch and commit, early tag broadcast on the multiplier, a gshare direction predictor, a Return Address Stack (RAS), store-to-load forwarding, a next line prefetcher, and a 2-way set-associative data cache.

The work is mostly divided into three milestones. Milestone 1 produced the ROB and RS modules in parallel branches and stitched them into a P6-style top-level pipeline. Milestone 2 stabilized the integration and got non-memory operations running end to end. Milestone 3 added the LSQ and write-back data cache, which brought every RV32IM load and store variant online. After Milestone 3 we turned to performance features. A bimodal branch predictor was the first, then the seven advanced features were merged one branch at a time.

The rest of the report is organized so the architecture comes before the features and the numbers come after both. Section III walks through the pipeline with a top level block diagram and a paragraph per stage. Section V covers the seven advanced features one at a time, each with the problem it solves, the design we picked, the tradeoffs, and the measured result. Section VII carries the cycle counts, the per feature speedups, the branch prediction accuracy data, and the synthesis slack numbers, including the one critical path that still misses the 1000 ps clock.

II. Background and Constraints

In an in-order pipeline, instructions execute in the same order the program lists them. That is simple to reason about and simple to build, but it has a familiar weakness. Suppose a load misses the cache and stalls for the full memory latency, and the very next instruction is an add that does not touch the load's destination register. The add has nothing to wait for, but the in-order pipeline makes it wait anyway, because the load sits in front of it.

Out-of-order execution lets the add issue and complete while the load is still running. The pipeline tracks operand readiness instead of program position, so independent work runs whenever its inputs are available. The catch is that results now come back in a different order than the program wrote them, and the architectural state has to look as if everything still happened in program order. That is what the Reorder Buffer is for. It holds in-flight results and retires them at the head, in order, so the register file and memory only see the program-order view. Three classical hazards drive most of the design choices: Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW). Renaming handles WAR and WAW by giving each instruction a fresh destination

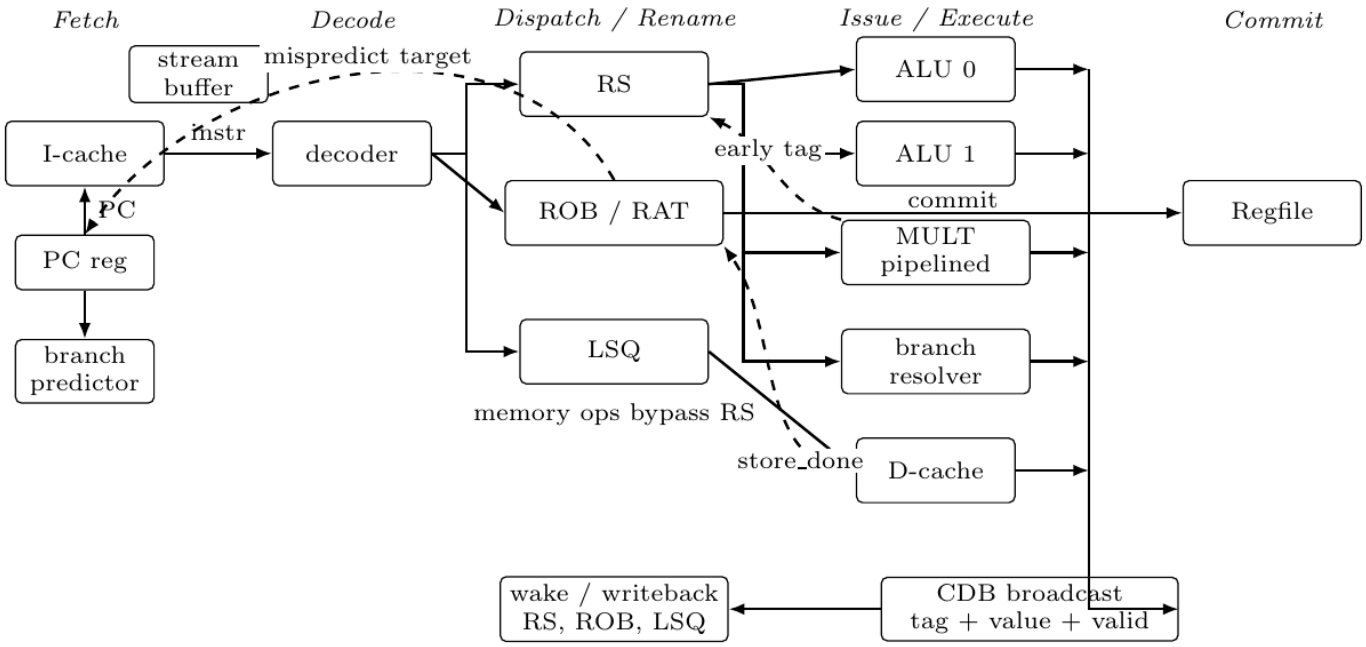

Fig. 1. Top-level pipeline block diagram. Stages, buses, and the sideband signals for early tag broadcast and store completion.

tag. The Reservation Station and CDB handle RAW by waking instructions up the moment their producers broadcast.

The ROB is the in-order checkpoint: it holds every dispatched instruction until commit and keeps the architectural state in program order. The RS is where a dispatched instruction waits for its operands and then hands itself to a functional unit once they arrive. The CDB is the shared wire that carries each completed result back to the ROB and to any waiting consumers. The LSQ is the memory-side equivalent of the ROB. It tracks loads and stores in program order and is the only path to the data cache. The branch predictor guesses the direction and target of each branch at fetch so the front end keeps moving instead of waiting for the branch to resolve at the far end of the pipeline.

Several numbers in this design were fixed by the assignment. Main memory has a 100 ns access latency. The instruction cache and the data cache are each capped at 256 bytes. The number of Common Data Buses cannot exceed the narrowest stage of the pipeline, so a one wide pipeline is allowed at most one CDB and a two wide pipeline at most two. The multiplier is the pipelined unit ported from Project 2. These constraints shape several of the tradeoffs that follow.

III. Pipeline Architecture

Figure 1 traces a single instruction from one end of the machine to the other. Fetch reads the PC, looks up the I-cache, and asks the branch predictor in parallel whether this PC is a taken branch the predictor already knows about. Decode turns the raw 32-bit instruction word into the fields the rest of the pipeline expects. Dispatch allocates a ROB slot for every instruction, and either an RS slot or an LSQ slot depending on whether the instruction needs memory access. Issue picks ready instructions out of the RS and hands them to a functional unit. The LSQ head, separately, talks to the D-cache when its turn comes. Execute happens on two ALUs, one pipelined multiplier, one inline branch resolver, and one D-cache port. Results travel back on the CDB, which wakes up waiting consumers in the RS and writes the value into the right ROB entry. Commit retires the ROB head in program order, writes the architectural register file, and runs the mispredict check that compares the resolved branch outcome against what the predictor guessed at fetch.

Fetch is two-wide on the front end. The PC drives the I-cache, and on a hit the cache returns a line wide enough to hand back two instructions in a single cycle. The branch predictor reads the same PC in parallel and may redirect fetch on the same cycle if it sees a taken branch or a return. Two structures help here. The BTB caches the targets of taken branches the program has executed before, so a predicted-taken conditional or a predicted JAL goes to the right place without waiting for the branch unit far downstream. The RAS handles function returns, where the target depends on which call site invoked the function rather than on the return instruction itself. A generic indirect branch predictor would have to relearn the right target every time control passed through a different call site. The RAS sidesteps that by pushing the link address on every JAL that writes the link register and popping it on the matching return. Returns are nearly always predictable this way.

Decode is reused largely unchanged from the in-order Project 3 pipeline. It reads the raw instruction word and produces the operand selects, the ALU function code, and the small set of flags (memory access, conditional branch, unconditional branch, halt, illegal) that the dispatch stage needs to route the

instruction correctly. Reusing the decoder kept the renaming, dispatch, and ROB logic the focus of the design effort, since those are the parts an out-of-order pipeline actually changes.

Dispatch is the most distinctive design departure from the provided template. Most P6-style pipelines keep a separate map table from architectural register to physical register, alongside a separate physical register file. Our design takes a different route. The ROB acts as the physical register file, and the RAT lives inside the ROB itself. Each ROB slot holds the in-flight result of one instruction, and the RAT is a 32-entry table that maps each architectural register to whichever ROB slot will produce its next value. Dispatch allocates a ROB slot, writes the new mapping into the RAT (except for x0, which is never tracked), and either inserts the instruction into the RS or hands it to the LSQ. We picked this shape because the ROB already has the right lifetime for an in-flight result: an entry is allocated at dispatch and freed at commit, which is exactly when a physical register would need to come and go. Folding the two structures together means one less thing to maintain and verify, and the renaming logic at dispatch is simpler because the ROB tail is the new physical register tag with no free list to manage. Memory operations skip the RS entirely and go straight into the LSQ, because the LSQ already enforces program order between loads and stores. Routing memory ops through the RS would only add a structure for them to sit in.

Issue and execute are where the out-of-order design pays off. The RS scans its entries each cycle and picks the oldest one whose two source operands are both ready, then hands it to a functional unit. The two ALUs handle the simple integer operations in one cycle each. The multiplier is pipelined with a fixed latency, so a long multiply does not stall the rest of the pipeline once it enters the unit. Conditional branches resolve in their own inline unit next to the ALUs, and that same unit produces the target for JAL and JALR. Loads and stores live on the LSQ side. Only the head of the LSQ talks to the D-cache, which keeps memory ordering simple. Every value-producing result travels back on the CDB, which wakes up RS entries waiting on that tag and writes the value into the ROB. Stores are the one exception. A store does not produce a register value, so it has nothing to broadcast. It signals completion to the ROB through a sideband (`store_done`) instead, which keeps the CDB free for instructions that actually have a result to deliver. Branches do broadcast on the CDB, because their resolved target is the value other parts of the machine need, but the actual PC redirect happens at commit and not at execute.

Commit is what makes the architectural state look in-order even though execution was not. The ROB head retires once it is busy and ready, writes its value into the architectural register file, and then runs the mispredict check on any branch that is committing. If the resolved direction or target does not match what the predictor said at fetch, the ROB raises a single cycle redirect signal that flushes the RS, flushes the LSQ, kills any multiply still in flight, and steers the PC to the correct target on the next cycle. In-order commit is what lets the machine speculate aggressively in the front end and still recover cleanly when the speculation turns out to be wrong. Without it, a mispredicted branch would leave the register file in some partially updated state that does not correspond to any valid program point.

We chose the P6 style, with the RAT embedded in the ROB and the ROB doubling as the physical register file, over the R10K style, which keeps a separate map table, a separate free list, and a unified physical register pool larger than the architectural register count. R10K makes more sense once a machine widens out and the unified pool starts to pay for itself, because instructions can rename freely without being bottlenecked by ROB allocation. At a one-wide base, though, the R10K bookkeeping costs more than it returns: a separate map table to read and write every cycle, a free list to keep consistent with allocation and reclaim, and a register pool with its own sizing and freeing policy. The advanced features we planned (superscalar, early tag broadcast, branch prediction enhancements, store-to-load forwarding, and cache improvements) do not need the unified pool to work, so the simpler P6 shape gave us the same correctness guarantees with less hardware to build and less RTL to verify. The tradeoff would tilt the other way past two-wide, but at the width we settled on, P6 was the cleaner fit.

IV. Base Implementation Details

A single `dispatch_fire` signal allocates a ROB slot and either an RS or LSQ slot in the same cycle, so the two structures never disagree about which instructions are in flight. The RAT lives inside the ROB as a 32-entry table that says, for each architectural register r, which ROB slot will produce its next value. Two correctness corners bit us at least once. The first is the stale-RAT clear at commit. When the ROB commits a slot that wrote r, the RAT entry for r is cleared only if it still points at that slot. A younger instruction may have already re-renamed r, and its mapping has to survive. Clearing unconditionally would erase a newer producer. The second is the JAL/JALR commit value override. JAL is the unconditional jump-and-link and JALR is its register-indirect cousin, and both write the next-PC (NPC, the address of the instruction after the jump) into a link register so the callee can return. The branch resolver computes the resolved target, which is what travels on the CDB and is the right value for the PC redirect. But for any branch whose destination register is non-zero, the link register has to hold the return address, not the target, so the ROB at commit overrides the broadcast value with NPC for those instructions. We found this the hard way: as soon as programs made function calls, every recursive return came back to the wrong PC because CDB consumers had already latched zero into the link register.

The RS issue selector reads the *registered* `src_ready` bits, not a combinational version that ORs in the current cycle's CDB wakeup. The earlier combinational form looked fine on paper but created a feedback loop. Issue picks an entry. Its tag drives the CDB, which wakes up a different entry whose ready bit was off a moment ago. The selector flips to that entry. The new selection broadcasts a different tag, the previously-woken entry goes back to sleep, and the selector flips back.

The simulator sat inside that delta cycle and never advanced. The simulator did not crash. The program just stopped making progress. `mult_no_lsq` froze at cycle 2192 every run, and roughly a dozen other tight-loop programs froze near the same horizon for the same reason. The fix is to use the registered ready bit, which costs one extra cycle when an operand arrives on the same cycle's CDB. The CDB-bypass mux still wires through to the issued entry's *value* output, so correctness holds.

The base design has one CDB with a fixed priority order: MULT first, then an LSQ load, then the ALU. MULT is pipelined and multi-cycle, so once a multiply is in flight its value pops out at a fixed time and asking it to come back later is awkward. ALU operations are one-cycle and replay cheaply, so they sit at the bottom and retry next cycle if they lose. Loads sit in between. Stores never use the CDB at all because they have no register value to deliver. When a store finishes its cache handshake, the LSQ raises a one-cycle `store_done` sideband to the ROB, and the store stays parked in the LSQ until the ROB commits it. Branches do broadcast on the CDB, because the resolved target is a value the rest of the machine needs, but the PC redirect happens at commit. That keeps mispredict recovery clean: only one branch is ever oldest in the ROB, so there is exactly one place the redirect can fire.

The LSQ is a first-in, first-out (FIFO) queue. Loads and stores enter at the tail in dispatch order, only the head talks to the cache, and stores hold their data until commit. Memory operations never enter the RS. The simplicity is deliberate. A one-port head-only LSQ already enforces what the rubric requires, which is that loads and stores retire in program order. Routing memory ops through the RS as well would give two structures overlapping responsibility for memory ordering, and any bug in one would be a bug in both.

The base D-cache is write-back, write-allocate, with byte-granular dirty and valid masks for sub-word stores. RV32IM has byte, half-word, and word load and store variants, so the cache has to track which bytes within a line are valid and which are dirty. Writebacks are always full doublewords. A partial-byte store is absorbed into the cache and only line eviction writes to memory. Write-back over write-through because memory latency is fixed at 100 ns and writing a doubleword on every byte store would dominate runtime. Write-allocate because most stores are followed by reads to the same address.

The base branch predictor is a bimodal direction predictor (a table of 2-bit saturating counters) plus a small BTB that caches taken-branch targets. Both lookups are combinational on the fetch PC, so a predicted-taken hit redirects fetch on the same cycle the PC is read. This is the base predictor the rubric requires and the comparison baseline used in Sec. V.C. The final build replaces the bimodal table with gshare and adds the RAS while keeping the BTB. We keep the bimodal version as the baseline so every speedup claim about gshare and the RAS is measured against the same starting point.

A short RV32 sequence makes the out-of-order's advantage clear:

```
lw x10, 0(x5) # load from memory; latency depends
    on the cache
addi x11, x12, 1 # independent of the load, ready
    immediately
add x13, x10, x11 # depends on the load result in
    x10
```

An in-order pipeline would freeze the `addi` behind the load even though the `addi`'s sources are already in the register file. Our RS dispatches all three, sees the `addi` is ready, and issues it to an ALU while the load is still talking to the D-cache, so the `addi` finishes and broadcasts on the CDB ahead of the load. The dependent `add` waits in the RS for the load's tag and only issues once that tag appears on the CDB.

V. Advanced Features

We layered seven advanced features on top of the base out-of-order pipeline: two from the difficult tier and five from the simpler tier. The assignment asks for at least one difficult feature alongside other simpler ones, so the count works out. Each of the seven gets its own subsection below, and Table I is the index for tracking which subsection covers which spec category.

Each subsection opens with the problem the feature solves, in plain language that does not assume a hardware background, then describes the design we built, the tradeoffs we accepted (area, timing, complexity, or a small loss elsewhere in the pipeline), and finally the result we measured. The two difficult features get dedicated subsections of their own (Sec. V.A for 2-way superscalar and Sec. V.B for early tag broadcast). The five simpler features are grouped under shared parents to keep related design decisions together: branch prediction enhancements in Sec. V.C, D-cache enhancements in Sec. V.D, and store-to-load forwarding in Sec. V.E. Each leaf still gets its own four-part treatment even when two leaves share a parent.

A. 2-way Superscalar

A one-instruction-wide pipeline retires at most one instruction per cycle. The usual way to talk about that ceiling is instructions per cycle (IPC), the average count of instructions a machine completes per clock tick. Real programs often spend long stretches with two unrelated instructions sitting ready in the Reservation Station, both waiting for an issue slot that will only ever serve one of them per cycle. That is wasted instruction-level parallelism (ILP), the technical name for the fact that a program's instructions are not all dependent on each other and could in principle run side by side [1]. Going from one issue per cycle to two roughly doubles the ceiling.

The widening runs from fetch through commit. The I-cache returns an 8-byte line on a hit, so a single fetch already carries two instructions on most cycles, and decode is two parallel decoders working on the two halves of that line. Dispatch allocates two ROB slots and either two RS slots or, if one of the pair is a memory operation, one RS slot and one LSQ slot, in lockstep so the two structures never disagree about what is in flight. Two ALUs at the end of the issue stage let two independent integer operations finish in the same cycle. The CDB widens to two slots as well, because the spec rule is that the number of CDBs cannot exceed the issue width

TABLE I
ADVANCED FEATURES IMPLEMENTED IN THIS DESIGN AND HOW EACH ONE MAPS ONTO THE SPEC'S SEC. 4.2 CATEGORIES.

#	Feature	Spec category (Sec. 4.2)	Tier
1	2-way superscalar	Superscalar execution	difficult
2	Early tag broadcast	Early tag broadcast (L7)	difficult
3	gshare predictor	Fetch, sophisticated branch predictors †	simpler
4	Return Address Stack	Fetch, return address stack	simpler
5	Store-to-load forwarding	Memory hierarchy, load/store forwarding	simpler
6	Next-line prefetch (stream buffer)	Memory hierarchy, prefetching †	simpler
7	2-way set-associative D-cache	Memory hierarchy, associative caches †	simpler

and we used the maximum allowed. Anything narrower would let two ALU results queue behind a single broadcast slot and turn the second ALU into a stall waiting its turn on the bus. Commit retires up to two ROB entries per cycle in program order so the back end keeps pace with the front end. A few units stayed single. There is still one MULT functional unit, because multiply is rare relative to ALU operations on the suite and a second pipelined multiplier would burn a lot of area for very little return. The queue of waiting consumers sits in front of the multiplier, not behind it, so one MULT does not become a serializing bottleneck the way a single ALU would. The LSQ also stays single-port. A second LSQ port and a second cache port would have been a much larger surgery than the rest of the widening combined, so two adjacent loads still serialize at the cache.

Going wide means rebuilding port logic everywhere. The RS now picks two ready entries each cycle instead of one, the ROB allocates and commits two slots per cycle, and the RAT has to handle the case where slot 1 reads an architectural register that slot 0 in the same cycle is renaming. Synthesis area went up and timing slack went down. That is a real cost, not a free one. The single-port LSQ is the most visible IPC limit, since memory operations are common enough that any program with a tight memory-stream inner loop will see one of its two issue slots sitting idle on those cycles.

Across the 33-program suite, the widening produces a 30 to 50 percent reduction in cycle count on most programs, with the largest wins on ALU-bound code such as alexnet and dft and the smallest on programs whose inner loops are dominated by serial dependences or by the LSQ. The per program breakdown is in the performance table in Sec. VII.

B. Early Tag Broadcast

A consumer that depends on a multiply normally waits for the multiplier's result to land on the CDB before it can issue. The multiplier's latency is fixed and known the moment a multiply enters the unit, though, so a dependent could in principle wake up earlier and be ready to issue on the cycle the value actually arrives. The cycle the consumer would otherwise spend sitting in the RS, watching the CDB, and only then flipping its registered ready bit is the cycle Early Tag Broadcast (ETB) tries to recover.

The mechanism is a single extra wire. When a multiply enters the multiplier's first stage, the unit drives the destination tag onto a dedicated early-tag sideband that runs alongside the CDB but is not part of it. Every RS entry watches that wire the same way it watches the CDB tag. If the early tag matches one of an entry's source tags, that source's registered ready bit flips to 1, and the entry becomes a candidate for the issue selector on the next cycle. The operand value itself does not ride this wire. The value still arrives on the CDB at the original cycle, and the CDB-bypass mux feeds it into the issued entry's value port the way it always did. ETB is purely an issue-eligibility wakeup and does not bypass the value path. Figure 5 shows the timing.

A unit-test scenario in `rs_test.sv` guards the split between issue eligibility and value delivery. The selector still reads the registered `src_ready`, never a combinational form that ORs in the early tag, because the registered-ready rule from Sec. IV holds here too. If the early tag fed `issue_found` combinationally, the same feedback loop that froze the simulator on tight loop programs in the base design would come back. The early tag has to stay a registered wakeup.

ETB only works on functional units whose latency is fixed and whose results never replay. The pipelined multiplier qualifies, because once a multiply enters stage 0 it will produce a result a known number of cycles later. Loads do not qualify, because a cache miss can extend their latency without warning. The other limit is structural. The wakeup arrives one cycle earlier, but for that early issue to fire, the dependent has to find an issue slot on the new cycle, and on the same cycle the producing multiply is broadcasting on the CDB. With one CDB, an ALU consumer the selector picks on the early cycle is held by the rule that gates non-MULT issue when a multiply is broadcasting, because the ALU's combinational result would collide with the multiply on the bus. The consumer issues one cycle later anyway. ETB's per program win is gated on the second CDB that arrives with the superscalar pass in Sec. V.A.

The measured effect on the 33-program suite is small. Disabling ETB at the all features on operating point raises geomean cycle count by about 0.10%, with the largest single program regression on `outer_product` at 1.07%. That program runs a tight back to back multiply loop and is the workload most directly aligned with what ETB optimizes. Everywhere else, the multi-cycle multiply path is not what the program is waiting on, so an earlier wakeup does not change much. The implementation is correct. It wakes RS consumers on the

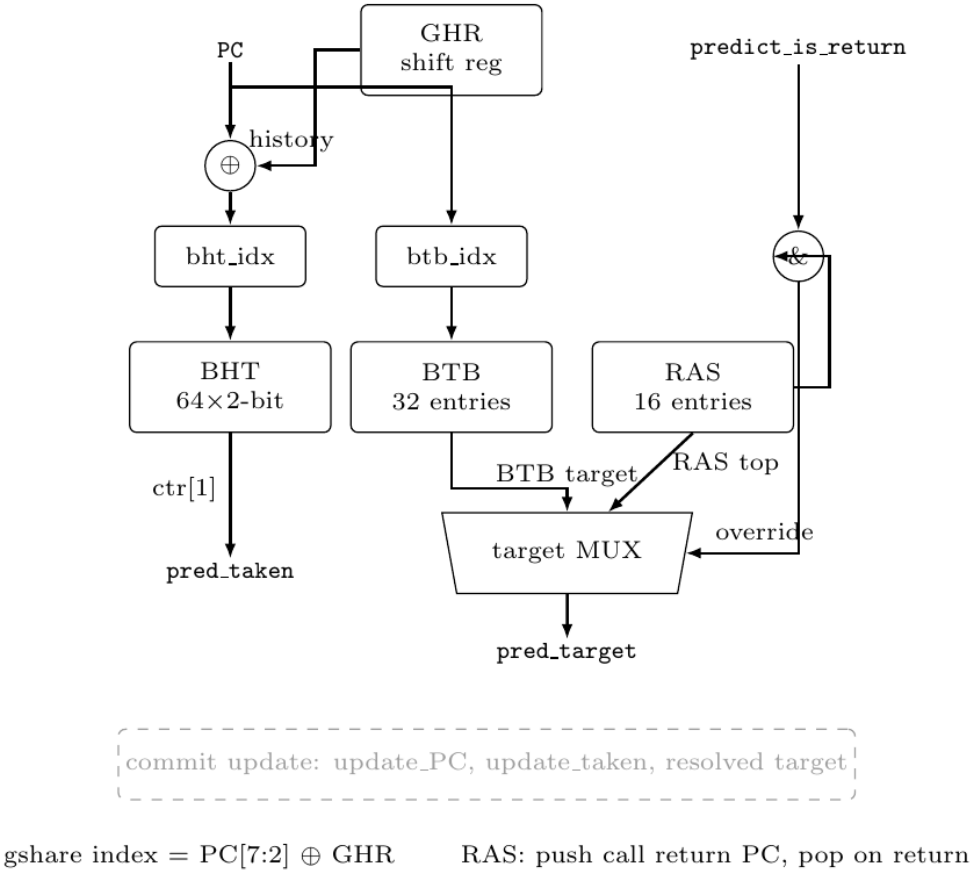

Fig. 2. gshare predictor with Branch Target Buffer and Return Address Stack.

cycle the multiply enters its final stage, and the unit-test guard against bypassing the value path stays green in every regression run. The marginal contribution at the all-on operating point reflects how rarely the multiplier sits on the critical path of this workload, not a defect in the wiring. The full per program breakdown is in Sec. VII.

C. Branch-Prediction Enhancements

Two predictor improvements went into the same `branch_predictor.sv` module that held the bimodal baseline. Both share the same fetch-PC input, the same Branch Target Buffer (BTB) lookup, and the same single-update-per-commit budget, but they fix different things. gshare goes after pattern aliasing on conditional branches, where two unrelated branches whose program counters share the table-index bits end up sharing a counter. The Return Address Stack handles a problem the bimodal table cannot help with at all: function returns are predictable from where the function was called, not from the return instruction itself. We present them as two leaves under one parent because they share a module, but each is its own feature with its own four-part write-up. Figure 2 shows both pieces alongside the BTB.

V.C.1. gshare. The bimodal predictor stored a 2-bit saturating counter at every Branch History Table (BHT) entry and indexed the BHT with PC bits alone. A 2-bit counter is a small piece of state that remembers whether a branch tends to be taken or not, and resists single-cycle exceptions to its prevailing direction. The problem is the indexing: two branches whose PCs land on the same six-bit slice share a counter even when they go in opposite directions. A loop-control branch that is almost always taken and a data-dependent branch that flips with input both train the same counter and pollute each other's prediction. Loop-control and data-dependent branches tend to sit near each other in compiled code, so this collision is common rather than rare.

The gshare design [2], which builds on the two-level adaptive predictor [5], indexes the BHT by `PC[7:2] XOR GHR`, where the GHR (Global History Register) is a shift register holding the taken/not-taken outcomes of recent committed conditional branches. The BHT has 64 entries, and the GHR width matches the BHT index width so every history bit can affect the result. A single hot branch with a "TNTNTNT…" pattern now lands on six different counters depending on which history led into it, instead of fighting itself on one. GHR width is a balance. Too narrow and patterns re-alias under different histories the way the bimodal table aliased on PC. Too wide and the predictor takes longer to specialize on a hot branch because more history bits must settle before a counter accumulates evidence. Tying the width to the BHT index split the difference cleanly.

The marginal contribution at the all features on operating point is small. Disabling gshare raises geomean cycle count by about 0.19% across the 33-program suite. The largest single program regression is `fib_rec` at 9.65%, the workload pattern gshare was built for: tight recursion where the same call site sees a short, regular taken/not-taken history that the XOR fold can specialize on but the bimodal table cannot. Programs whose inner loops are not predictor-bound see no change. The full per program breakdown is in Sec. VII.

V.C.2. Return Address Stack. Function returns are nearly 100% predictable in principle, because the right target is always "go back to the call site that wrote the return address into the link register." A generic indirect-branch predictor cannot exploit that. The BTB caches one target per branch, so a recursive function whose return sees a different call site on every frame leaves the BTB constantly relearning the wrong target. Every switch between frames mispredicts.

The Return Address Stack is a 16-entry hardware stack that tracks call sites the way software tracks them. When the front end sees a JAL that writes to the link register, the next-PC of that JAL is pushed. When the front end sees a return-shaped JALR (`x0, ra, 0`), the top of the stack supplies the predicted target and the entry is popped. On returns, the RAS overrides the BTB. When the stack is empty, the override does not fire and the predictor falls back to the BTB, so cold-start behavior matches the pre-RAS design. The 16-entry depth is generous for this benchmark suite, where the deepest recursion observed is in the single digits, but a 16-entry table costs little area and removes saturation as a concern during testing.

The RAS contribution at the all-on operating point is small, like gshare. Disabling the RAS raises geomean cycle count by about 0.10%, and the largest single program regression is `basic_malloc` at 0.52%. The benchmark suite does not have enough deep recursion in its hot paths to make the RAS a dominant feature at the geomean level. Where it does help, it replaces a steady stream of BTB mispredicts with a stack lookup that gets the target right on every call frame. The full per program breakdown is in Sec. VII.

D. D-cache Enhancements

The D-cache picked up two separate improvements that share enough infrastructure to belong in the same parent section even though they target different miss patterns. Both depend on the bus arbitration mask in `pipeline.sv` that gives each cache a view of `mem2proc_response` masked to zero on cycles when that cache did not actually drive the bus. Without the mask, one cache would latch the other's response and silently corrupt a fill. The next-line prefetcher also reuses the same `stream_buffer.sv` module on the I-cache side that the D-cache uses on the data side, so the prefetch logic itself is one piece of code serving two clients. The set-associative geometry attacks conflict misses, where two hot addresses fight over a single cache slot. The prefetcher attacks cold misses on sequential walks, where the program is about to reference a line that nothing has loaded yet. Figure 3 shows both pieces alongside the bus arbitration.

V.D.1. 2-way set-associative D-cache. A direct-mapped cache picks a slot for an address by taking a slice of the address bits as the index and storing the line there. A 256 byte D-cache with 8 byte lines has 32 slots, and any two addresses whose index bits collide have to share one slot. If a program walks two arrays whose strides happen to fold onto the same slot, every reference evicts the other one even though the rest of the cache is sitting empty. Sort-style inner loops do this regularly: an array element, a loop counter, and a comparison key all touching memory in the same iteration.

The 2-way layout splits the same 256 bytes into 16 sets of 2 lines each. An address now picks a set, and either of the two ways inside that set can hold the line. Two addresses that would have collided in the direct-mapped cache now coexist. Each set carries a single least-recently-used (LRU) bit that tracks which way was touched more recently. On a miss into a set whose ways are both valid, the eviction picks the LRU way. At 2-way, one bit of state is exact LRU rather than an approximation. Sub-word stores still work the way they did before. Per-byte valid and dirty masks let a byte or half-word store update only the bytes it touches, and writebacks still emit the full 8-byte line.

The cost is a second tag comparator per set and the LRU bit, which is small in absolute terms but is paid on every access. The geometry helps when conflict misses dominate and does nothing when the program is just streaming through memory, since the prefetcher and the cache size set the ceiling there.

This feature is structural, which makes its contribution harder to measure than the others. The 2-way layout is wired into the array dimensions of `dcache.sv` and into the address decode, and there is no `+define` flag that turns it back into a direct-mapped cache without a substantial RTL rebuild. We left it out of the leave one out ablation sweep for that reason. Its contribution shows up in the D-cache hit rate rather than in a single ablation row, and Sec. VII reports the hit rate breakdown.

V.D.2. Next-Line Prefetch / Stream Buffer. Programs spend a lot of their time walking through memory in order: instruction fetch through straight-line code, array sweeps in matrix kernels, struct-field reads in image processing. Every cold line in that walk costs the full 100 ns memory latency, and the cache cannot start the fetch until the program asks for the line. A small prefetcher can fetch the next line in the background while the program is still using the current one, so by the time the program asks for line N+1, it is either already in the cache or close to landing [1].

The mechanism is a one-line stream buffer between the cache and main memory. On a miss fill for some line N, the buffer queues a request for line N+1 and issues it once the bus is free. If the program later misses on line N+1 while the buffer is holding it, the line transfers into the cache without a fresh round trip to memory. If the program goes elsewhere first, the buffered line just sits there until the next request overwrites it. The same module is reused on the I-cache side because instruction fetch is the most predictable sequential walk in the machine. The bus has a fixed priority: D-cache demand requests go first, then I-cache demand requests, then either

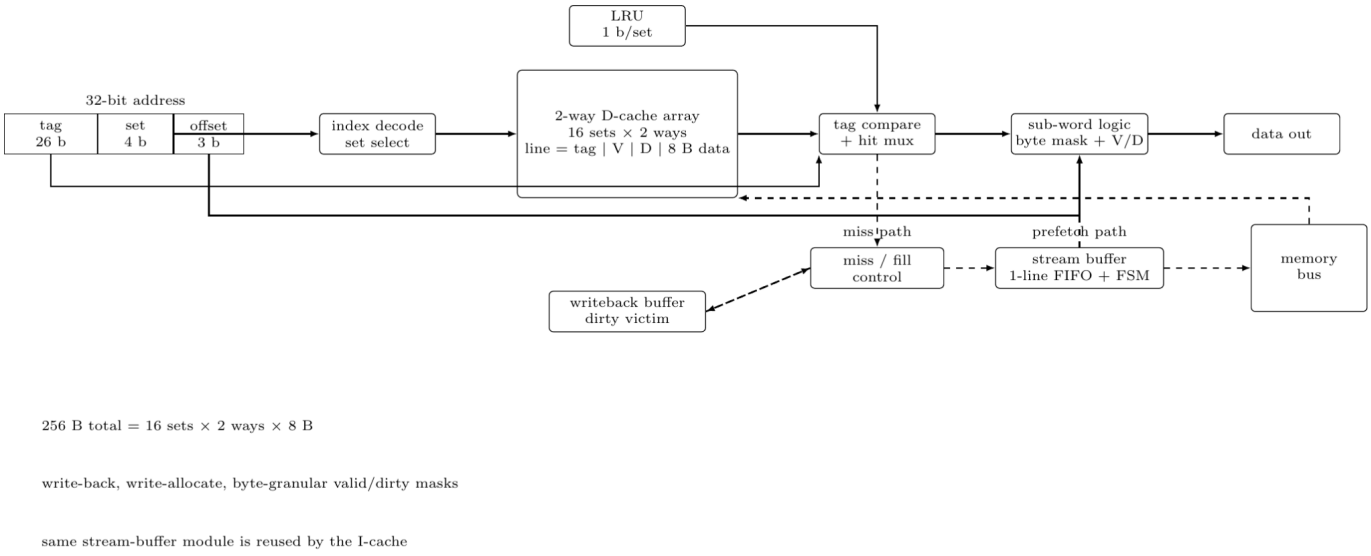

Fig. 3. D-cache organization with tag/set/offset decode, two-way array, LRU state, miss/fill control, stream buffer, and memory-bus path.

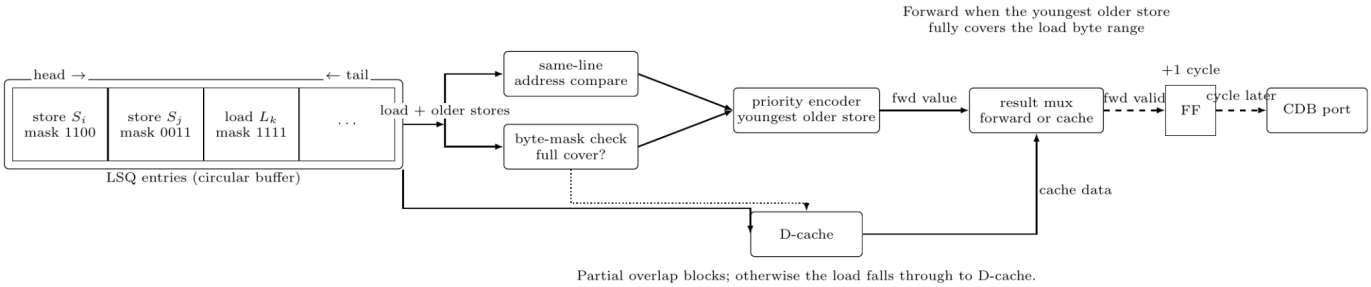

Fig. 4. Store-to-load forwarding lanes in the Load-Store Queue.

stream buffer. Stream-buffer requests are speculative, so they yield to anything the program actually asked for.

The cost is one extra bus request per miss. When the program's access pattern is predictable, that request becomes a future hit and saves a memory round trip. When the pattern is random, the request was wasted, but the bus arbitration ensures it never delays a real demand fetch. The bus arbitration mask in `pipeline.sv` is what keeps the rest from breaking. With two caches plus the stream buffer all watching `mem2proc_response`, the mask is the only thing that keeps each module from latching whichever response happens to land in its memory tag window.

The prefetcher is the dominant feature in the per-feature ablation data. Disabling it raises geomean cycle count by 37.14% across the 33-program suite, and the worst single program, `alexnet`, regresses by 94.22%. For comparison, removing all five ablate-able features together raises geomean cycle count by 37.86%, so the prefetcher alone accounts for nearly the entire gap. This is consistent with the rest of the design. The machine is one issue slot wide on the front end, the I-cache is small, and a 100 ns memory latency on every cold instruction line is not something the rest of the pipeline can hide. The per program breakdown is in Sec. VII.

E. Store-to-Load Forwarding

A load that asks for an address an older store has just written should not have to wait for the cache. In a write-back design, the store sits in the LSQ until commit, then waits for a cache port, then marks the line dirty. A dependent load behind it pays every one of those cycles for what is logically a register-to-register move. In tight loops where each iteration writes a value and immediately reads it back, that wait fires every iteration.

Store-to-Load Forwarding (STLF) handles the easy case directly. When a load reaches the head of the LSQ, the queue compares its address against every older un-committed store on the same 8-byte line. If the most recent matching store fully covers the load's byte range and its data is already known, the load completes in one cycle out of the LSQ and the cache never sees the request. The order walk runs from oldest to youngest so the load always sees the right store, even if a younger store on the same line would also match. Anything that breaks the chain blocks the forward: an older store with an unresolved address, an older store whose data has not arrived yet, or a partial overlap where the load needs bytes the store did not write. Partial overlap could in principle merge cache bytes with store bytes, but we did not wire that path. The

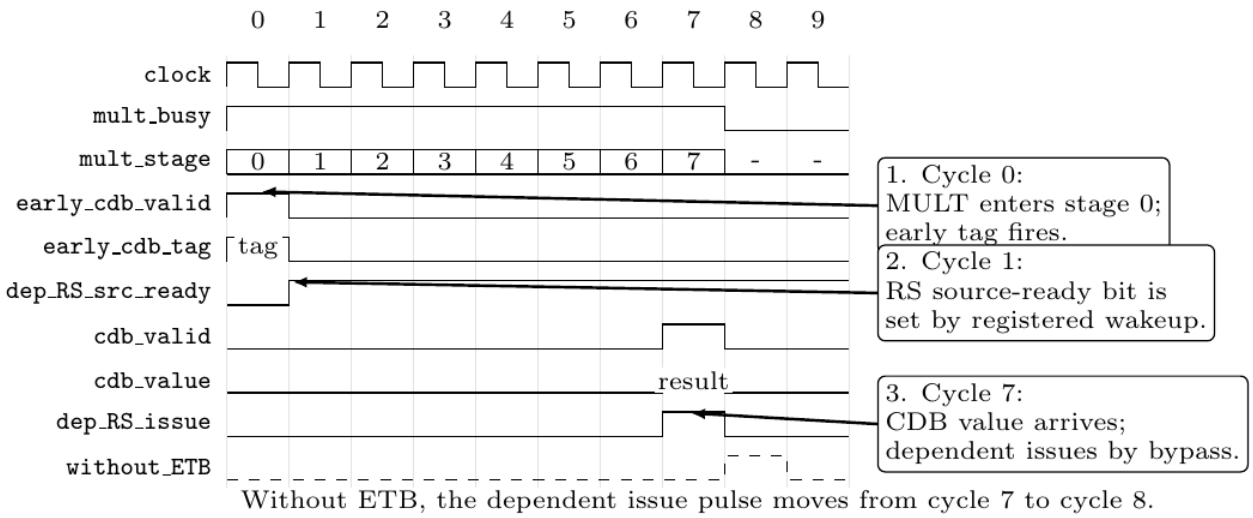

Fig. 5. Early-tag-broadcast timing for a pipelined multiply and a dependent reservation-station entry.

regression does not contain programs where it would fire often enough to matter, and the correctness corners are easier to reason about when partial overlap simply waits. Figure 4 shows the comparator and the mux that selects between the forwarded value and the cache return.

A late timing-cleanup pass deferred forwarding by one cycle (mergesort 200,072 -> 200,073 cycles) for synth-slack reasons. The earlier `-504.66 -> -244.54 ps` measurement was retracted as stale build-artefact data, and the standalone slack contribution has not been re-baselined. The architectural change in `verilog/lsq.sv` is real and remains in the design (forwarded loads broadcast one cycle later via the existing STLF latch). Only the slack numbers attributed to it are retracted. The cycle cost lands only on the small set of loads that actually forward.

The marginal contribution is small. Disabling STLF at the all features on operating point raises geomean cycle count by about 0.20% across the 33-program suite. The largest single program regression is `insertionsort` at +1.64%, which is the canonical pattern: an inner loop that writes an array element and reads it back on the next iteration. Other sort-style and linked-list-walking programs see similar but smaller effects. Programs without RAW-through-memory chains barely move. The full per program breakdown is in Sec. VII.

VI. Verification and Testing Methodology

The verification harness has three layers, and each catches a different class of bug. At the bottom, every nontrivial module has its own SystemVerilog testbench. A test passes only if the run prints `@@@ Passed`, and any failure prints `@@@ Incorrect`. Above that, the full pipeline runs 33 RV32IM programs end to end, checked for a clean halt and a correct register-write trace. On top of both layers, the same tests run a second time against the gate-level netlist that Synopsys Design Compiler (DC) produces. Unit tests catch local regressions on the module the test owns. Full pipeline regressions catch integration bugs that only show up when modules talk to each other. The synthesized rerun catches netlist-versus-RTL bugs that surface when DC flattens a port shape, drops a wire it thinks is unused, or otherwise produces something that does not match what the source claimed.

The RTL unit suite covers `mult`, `rob`, `rs`, `lsq`, `dcache`, `icache`, and `branch_predictor`. The ROB tests cover dispatch, CDB completion, in-order commit despite out-of-order completion, the same-cycle RAT bypass, the stale-clear protection that keeps a committing slot from erasing a younger instruction's RAT mapping, the x0 guard, flush on mispredict, full-detection at the boundaries, and head-tail wraparound. The RS suite covers ready-on-dispatch issue, dependency wakeup, same-cycle CDB bypass into the issued entry's value, backpressure, and full behavior. It also includes a guard scenario for the early-tag-broadcast invariant from Sec. V.B: the early-tag wire must wake an entry's registered ready bit but must not feed the issue selector combinationally. If a future change puts the early tag back on the combinational path, that test fails before the change reaches a full-pipeline run, and the feedback loop that froze the simulator on tight-loop programs in the base design does not return. The LSQ, dcache, icache, and multiplier suites cover their own corner cases the same way.

The same testbenches run a second time against the gate-level netlist. The bridge that makes this work is a small wrapper per module, `synth/<m>_svsim.sv`, that sits between the testbench and the netlist. DC flattens an unpacked-array port like `logic [4:0] dispatch_dest_reg [2]` into one packed bus, while the testbench still declares the unpacked shape so it can keep working against the RTL. The wrapper accepts the unpacked view on one side and uses the SystemVerilog stream operator `{>>{ }}` to repack into the bus form on the other. A `+define+SYNTH` switch picks between the bare module and the wrapper. The branch predictor test needed more than a wrapper. The original suite asserted against raw-PC BHT indices, which broke once gshare made the index `bht_pc_bits(PC)^ GHR`. The

rewrite carries a TB-side gshare reference model that mirrors the GHR, the BHT, and the BTB and computes the prediction the RTL should produce on every step. Test 7 exercises the saturate-then-flip semantic of the original test and preserves it under XOR indexing by picking colliding PCs at every step. Each update PC is chosen so `bht_pc_bits(pc_k)^ ghr_pre_k` lands on the same target index regardless of how the GHR has shifted. With both pieces in place, all seven modules pass on the netlist.

The full-pipeline regression runs every program in `programs/` end to end. A program passes when the run halts on `@@@ System halted on WFI instruction` and the writeback trace is correct. `make simulate_all` runs the suite on the RTL build, and all 33 programs pass. `make simulate_all_syn` runs the same suite on the synthesized netlist. The most recent full-suite synth run was on the pre-multiplier-operand-register baseline, where every `.syn.wb` was byte-identical to its `.wb`. The subsequent multiplier-operand-register addition is a flop insertion with no value change, so same-commit RTL <-> netlist byte-equivalence should hold post-merge, but we need to re-run `make simulate_all_syn` before final sign-off. The suite spans workload shapes: toy programs like `no_hazard` and `btest1`, kernel-style programs like `mergesort`, `bfs`, and `backtrack` that exercise the memory hierarchy and branch predictor on real control flow, and `alexnet`, a forward pass through a small convolutional neural network (CNN) that exercises the long-running ALU-bound steady state.

Two byte-level equivalence checks back up the regression. First, every `.syn.wb` writeback trace is byte-identical to its `.wb` counterpart on all 33 programs on the pre-multiplier-operand-register baseline. The synthesized gate-level netlist commits the same architectural register-write stream as the RTL, cycle for cycle (modulo the standard one-cycle Synopsys reset offset). The property should survive the subsequent flop insertion in front of the multiplier, but `make simulate_all_syn` has not been re-run after that change. This is what tells us the residual -797.58 ps slack on `synth/pipeline.vg` is a static-timing concern and not a correctness one. The netlist behaves the way the RTL does on every program, so a later timing fix can change gates without worrying about architectural divergence the regression already ruled out. Second, every `.wb` on the post-merge build is byte-identical to the trace produced by rebuilding the same commit under `+define+SERIALIZE_BRANCHES`, which forces the front end to serialize on every conditional branch and turns the OoO machine into something close to a one-at-a-time reference. If out-of-order issue plus speculation drifted from what a serialized front end would have done on the same program, this check would fail. It does not. Out-of-order is doing the same architectural work as the simpler model, just with the cycles arranged differently.

Five compile-time `+define` knobs disable individual advanced features for the leave one out ablation: `DISABLE_EARLY_TAG`, `DISABLE_GSHARE`, `DISABLE_RAS`, `DISABLE_STLF`, and `DISABLE_PREFETCH`. Each knob compiles cleanly and produces a working binary that halts every program. We generated the per-feature attribution numbers in Sec. VII by running `make simulate_all` once per knob and once with all five set together as the baseline. Two of the seven advanced features are not in this list. The 2-way superscalar pipeline and the 2-way set-associative D-cache are structural changes that touch port widths, array dimensions, and address decode in ways no single `ifdef` can roll back without a substantial RTL rebuild. Their contribution comes from cycles per instruction (CPI, average clock cycles per retiring instruction) bounds on ILP-rich code and from D-cache hit-rate analysis, rather than from a feature-off rebuild, and Sec. VII says so.

VII. Performance Evaluation and Analysis

Cycle counts come from the simulation harness, which prints both the cycle total and the dynamic instruction count at halt. Cycles per instruction (CPI) is the ratio of those two. Two reference points anchor every speedup claim in this section. The first is the full-feature build, where all seven advanced features are active. We call this **all-on** and treat its measured cycle counts as the operating point. The second is the build where the five compile-time disable knobs from Sec. VI are set together (`DISABLE_EARLY_TAG`, `DISABLE_GSHARE`, `DISABLE_RAS`, `DISABLE_STLF`, `DISABLE_PREFETCH`). We call this the **OoO base** because the out-of-order machinery is intact but every ablate-able advanced feature is off. The two structural features (2-way superscalar, 2-way set-associative D-cache) are present in both builds because no `ifdef` rolls them back without a substantial RTL rebuild.

Per feature attribution uses leave one out at the all-on configuration. We disable one feature at a time, holding the other four ablate-able ones on, and report the cycle-count regression. That measures the **marginal** value of each feature at the operating point the design ships in, which is what a speedup claim should mean: this is what the feature buys you in the configuration we run, not what it buys against an unrelated baseline that already has six other things turned off. Leave one out does not add cleanly to the all-features-off gap, so we cross-check the gap explicitly in Table III.

Across the 33-program suite, the seven advanced features cut cycle count by 27.46% on geomean over OoO base. Geomean CPI at all-on is 14.87 (arithmetic mean 20.77), pulled up by the small toy programs that pay full memory latency on a handful of fetches. The more representative kernel-style workloads sit between 3 and 30 CPI. Branch prediction accuracy at all-on is 76.13% arithmetic mean across the 30 programs that execute at least one conditional branch, with a geomean of 74.03%. The lift over the bimodal baseline is 8.87 percentage points arithmetic mean (bimodal averaged 67.25% on the same set). One program, `insertion`, shows a small +2.13% regression at all-on. The data attributes this to gshare BHT aliasing on a particular pair of branches, and disabling gshare alone speeds that program up by 0.73%. We treat that as a known artifact of XOR indexing rather than a defect, since the same predictor helps far more programs than it hurts.

The marginal cost of disabling each feature one at a time tells a sharper story than the headline.

TABLE II

PER-PROGRAM PERFORMANCE ACROSS THE FULL 33-PROGRAM SUITE. Δ% IS THE SIGNED CHANGE IN CYCLE COUNT FROM OOO BASE TO ALL-ON, COMPUTED AS (CYCLES_ALL_ON - CYCLES_OOO_BASE) / CYCLES_OOO_BASE X 100. NEGATIVE MEANS THE ADVANCED FEATURES REDUCE CYCLES (LOWER IS BETTER). CPI AND BRANCH ACCURACY ARE AT THE ALL-ON OPERATING POINT. GEOMEAN AND ARITHMETIC MEAN ARE OVER THE SUITE.

Program	cycles OoO base	cycles all-on	Δ%	CPI all-on	branch acc. all-on
alexnet	9,186,436	4,730,247	-48.51%	22.63	84.74%
backtrack	250,581	146,853	-41.40%	20.39	83.74%
basic_malloc	49,284	27,798	-43.60%	29.45	56.39%
bfs	111,376	66,438	-40.35%	19.07	64.31%
btest1	17,087	10,357	-39.39%	44.84	60.00%
btest2	27,207	14,013	-48.50%	30.66	33.33%
copy	3,686	3,472	-5.81%	26.30	87.50%
copy_long	5,773	5,264	-8.82%	8.89	88.23%
dft	1,685,731	1,008,057	-40.20%	17.42	82.48%
evens	1,166	1,170	+0.34%	11.82	76.74%
evens_long	2,934	2,563	-12.65%	7.63	76.74%
fc_forward	51,721	33,419	-35.39%	4.97	82.88%
fib	2,371	2,048	-13.62%	13.65	86.66%
fib_long	6,240	4,940	-20.83%	7.74	85.71%
fib_rec	32,018	29,132	-9.01%	2.44	65.56%
graph	450,656	259,337	-42.45%	23.31	59.83%
haha	935	528	-43.53%	29.33	n/a
halt	106	106	0.00%	106.00	n/a
insertion	3,093	3,159	+2.13%	5.27	87.27%
insertionsort	750,097	554,803	-26.04%	3.88	88.46%
matrix_mult_rec	712,425	662,478	-7.01%	30.56	94.37%
mergesort	294,331	200,073	-32.02%	21.10	75.38%
mult	7,565	7,430	-1.78%	22.79	83.33%
mult_no_lsq	2,920	2,251	-22.91%	7.95	88.23%
no_hazard	725	422	-41.79%	30.14	n/a
omegalul	3,944	2,220	-43.71%	30.00	33.33%
outer_product	3,983,006	3,166,519	-20.50%	4.24	85.18%
parallel	2,328	2,135	-8.29%	10.68	87.50%
priority_queue	77,416	43,389	-43.95%	29.82	56.47%
quicksort	871,758	568,772	-34.76%	5.96	84.28%
sampler	6,220	3,378	-45.69%	30.71	74.35%
saxpy	4,515	4,230	-6.31%	22.62	85.00%
sort_search	718,427	600,637	-16.40%	3.30	85.81%
geomean (33)			**-27.46%**	**14.87**	**74.03%**
arith. mean (33)			**-25.54%**	**20.77**	**76.13%**

TABLE III

PER FEATURE ATTRIBUTION. GEOMEAN Δ% IS THE CYCLE-COUNT REGRESSION FROM DISABLING ONE FEATURE AT THE ALL-ON OPERATING POINT. WORST CASE IS THE LARGEST PER-PROGRAM REGRESSION. SOURCE = ABLATION IF MEASURED BY LEAVE ONE OUT, STRUCTURAL IF INFERRED FROM CPI BOUNDS OR HIT-RATE ANALYSIS.

Feature	Geomean Δ% when disabled	Worst case (program)	Source
Next-line prefetch	+37.14%	alexnet (+94.22%)	ablation
STLF	+0.20%	insertionsort (+1.64%)	ablation
gshare	+0.19%	fib_rec (+9.65%)	ablation
ETB	+0.10%	outer_product (+1.07%)	ablation
RAS	+0.10%	basic_malloc (+0.52%)	ablation
2-way superscalar	structural	(see prose)	analytical
2-way set-assoc D-cache	structural	(see prose)	analytical
all 5 disabled	**+37.86%**	**alexnet (+94.21%)**	ablation

Prefetch dominates the marginal contribution. Disabling next line prefetch alone raises geomean cycle count by 37.14%, while disabling all five ablate-able features together raises it by 37.86%. The four non-prefetch features contribute roughly 0.72 pp combined. That is consistent with the design's memory profile. The machine has a one-instruction-wide front-end ceiling on most cycles, the I-cache is 256 bytes, and the memory latency is 100 ns per cold line. Without a prefetcher, the front end stalls on every fresh cache line and the rest of the pipeline has nothing to hide. The stream buffer turns most of those stalls into hit-latency accesses on sequential walks, and instruction fetch is the most predictable sequential walk in the machine. The ablation strongly suggests the I-cache side is the dominant beneficiary, since the worst-regressing programs under `no_prefetch` (`alexnet` +94.22%, `btest2` +94.16%, `sampler` +84.13%) are the ones whose cycle count is most sensitive to instruction-fetch latency rather than to data-side stride behavior.

ETB, gshare, RAS, and STLF each contribute between 0.10% and 0.20% on geomean. We want to be plain about that: these are small numbers. The features are not broken. They do what their reports say they do, and the worst-case columns show real per-program effects (gshare cuts `fib_rec` by 9.65% over its `no_gshare` cycle count, STLF cuts `insertionsort` by 1.64%

over its no_stlf cycle count, ETB cuts outer_product by 1.07% over its no_etb cycle count). The geomean stays small because most of the suite is not bottlenecked on the thing each feature optimizes, and because the prefetcher already absorbs most of the cycles those features could have saved on their own. The single-CDB cap also gates ETB: a non-MULT consumer woken by the early tag still has to wait its turn on the bus when the MULT broadcast lands the same cycle, and the second CDB from Sec. V.A is what would unlock the full ETB win on a wider machine.

The two structural features sit outside the ablation. The 2-way superscalar pipeline is wired into port widths, RS issue logic, ROB allocate-and-commit, and the CDB count. Rolling it back to one-wide is a re-design, not a +define. Its contribution shows up indirectly in the all-on CPI numbers. Several ILP-rich programs reach CPI well below 1.0 if you discount fetch latency (fib_rec 2.44, sort_search 3.30, insertionsort 3.88, outer_product 4.24), and a strictly one-wide machine cannot sustain CPI below 1.0 by definition. The widening therefore must be doing real work on those programs. The 2-way set-associative D-cache is similar: the geometry is wired into dcache.sv's arrays, not selectable, and its contribution shows up as a hit-rate change rather than as an ablation row. The harness does not print a hit-rate counter, so the cleanest proxy we have is the per-program cycle reduction on programs whose inner loops touch two stride-aligned arrays (sort kernels and BFS-style traversals), which match the conflict-miss pattern the second way is built to absorb.

Branch prediction accuracy moves in the directions the design predicts. The largest accuracy lifts over the bimodal baseline land on programs whose branch behavior the bimodal table cannot specialize on: omegalul from 0% to 33.33%, btest1 from 33.33% to 60.00%, priority_queue from 35.46% to 56.47%, bfs from 47.85% to 64.31%, basic_malloc from 33.33% to 56.39%. These all have data-dependent or call-site-dependent branches that gshare's XOR fold and the RAS together capture. Recursion-heavy programs are where the RAS earns its keep: backtrack (4.33 pp lift), matrix_mult_rec (a small 0.24 pp lift on top of an already 94% baseline), and the fib_rec workload that benefits most from gshare also benefits some from RAS via cleaner return targets. Programs whose branches are mostly counted loop tests already sit near 85-88% accuracy under bimodal (copy, evens, fib_long, mult_no_lsq, parallel, saxpy), and gshare and RAS together add fractions of a percentage point or none at all. That is the expected shape: pattern correlation matters most when the pattern is non-trivial.

The data cache side is harder to read directly because the harness does not expose hit-rate counters. We fall back to per-program cycle behavior as a proxy. Programs whose inner loops walk two arrays that fold onto the same direct-mapped index (sort and search kernels, plus bfs and graph) post the largest reductions from OoO base to all-on (bfs -40.35%, graph -42.45%, quicksort -34.76%), and the prefetch ablation does not account for the full size of those cuts. The residual is consistent with conflict-miss relief from the 2-way set-associative geometry. Programs that stream through memory with a single stride (copy, saxpy, evens) get most of their cycle reduction from prefetch and very little from associativity, the expected shape.

All seven module-level testbenches meet timing at 1000 ps. mult and lsq are the tightest at +0.23 ps and +0.05 ps, and the other five clear by hundreds of picoseconds. The only timing miss in the design lives in the full-pipeline netlist.

The full pipeline netlist (synth/pipeline.vg) does not meet timing at 1000 ps. Two endpoints violate. Worst slack is -797.58 ps on the path lsq_0/head_reg[2] to rob_0/entries_reg[2][take_branch], with a companion endpoint at -797.55 ps on lsq_0/head_reg[2] to lsq_0/entries_reg[3][addr][31]. Every other endpoint in the design meets at the same clock. The cone runs LSQ broadcast -> RS operand mux -> ALU 32-bit adder -> {ROB take_branch, LSQ addr}: a single 32-bit ripple-carry adder with high fanout sits in the middle of a long combinational chain. Closing it fully would mean either registering load_complete_value / load_complete_tag between the LSQ broadcast arbiter and the CDB (one extra cycle on every completing load) or splitting the ALU's 32-bit adder into two pipeline stages (one extra cycle on every ALU op). We judged both costs to outweigh the static-timing relief at this point in the project, and Sec. VIII discusses the trade.

Functional correctness and static-timing closure are different things, and the distinction matters here. Every .syn.wb writeback trace produced by the gate-level netlist is byte-identical to the corresponding .wb from the RTL across all 33 programs on the pre-multiplier-operand-register baseline. The synthesized machine commits the same architectural register-write stream as the RTL on every program, cycle for cycle modulo the standard one-cycle reset offset. The subsequent multiplier-operand-register addition is a flop insertion with no value change, so the byte-equivalence property should hold post-merge, but make simulate_all_syn has not been re-run after that change. A fresh re-verification is needed before final sign-off. The -797.58 ps slack is what the static-timing engine reports against an aggressive 1000 ps target. It is not a glitch, and gate-level simulation does not expose any path where the violation manifests as wrong behavior. The design is functionally correct, and two endpoints miss the 1000 ps target.

VIII. Discussion: Limitations and Future Work

If we target the 1000ps as the original Makefile do, the remaining 797.58 ps full pipeline timing miss is still to be optimized. The MULT stage-0 multiply tree that was the worst path through milestone 4 is closed by the recent multiplier-operand-register addition (re-synth: prior post-merge baseline ≈ -1600 ps -> current -797.58 ps, ~800 ps recovered). The new bottleneck is LSQ broadcast -> RS operand mux -> ALU 32-bit adder -> {ROB take_branch, LSQ addr}. A single 32-bit ripple-carry adder with high fanout sits in the middle of a long combinational chain. Closing it would mean either registering load_complete_value / load_complete_tag between the LSQ broadcast arbiter and the CDB (one extra cycle on every completing load) or splitting the ALU's 32-bit adder into two pipeline stages (one extra cycle on every ALU

op). Both pay perf on the common case to fix the static-timing residual, so this pass stops short. The netlist is functionally bit-equivalent to the RTL on the pre-multiplier-operand-register baseline. The merge is a flop insertion in front of the multiplier with no value change, so the byte-equivalence property should hold after the merge, but `make simulate_all_syn` has not been re-run. With more time this is the one limitation we would close.

The LSQ is single-ported on a 2-way machine. Two adjacent loads still serialize at the cache, so the front-end widening is not always matched by back-end memory bandwidth. The natural next step on a wider machine is a dual-ported LSQ paired with a dual-ported or banked D-cache.

A single multiplier has the same shape on the arithmetic side. Multiply-heavy code caps out at one issue per cycle through the multiplier. Early tag broadcast recovers some of this by waking dependent consumers a cycle before the result lands on the bus, but a second multiplier would do better. We did not add one because doubling area for a single functional class was hard to justify against an ablation that suggested the marginal cycles available were small.

The ablation finding itself is worth naming as a design lesson rather than a disappointment. Once the prefetcher is in place, the simpler features stacked on top of an already-tuned base contribute less on geomean than they would in isolation, because the prefetcher has already taken the cycles they would have saved. The bottleneck on this suite at this clock is memory latency. One earlier data point makes the same case. The milestone-2 build with no LSQ froze deterministically at cycle 2192 on `mult_no_lsq`, and landing the LSQ closed the gap. That is why memory ordering became its own structural piece rather than an afterthought.

If we were widening past 2-way, we would revisit the embedded-RAT-in-ROB rename approach. A unified physical register pool in the R10K style would make more sense at four-wide, where the simplicity payoff of treating ROB entries as physical registers starts to fall off.

IX. Conclusion

We built a synthesizable P6-style out-of-order RV32IM processor on top of the in-order Project 3 starter, and layered seven advanced features above the base: 2-way superscalar, early tag broadcast, gshare, a return address stack, store-to-load forwarding, next-line prefetch through a stream buffer, and a 2-way set-associative D-cache. All 33 test programs pass on the RTL, and every `.syn.wb` writeback trace was byte-identical to its `.wb` counterpart on the pre-multiplier-operand-register baseline. The subsequent flop insertion should preserve byte-equivalence, but `make simulate_all_syn` has not been re-run after that change. The seven features together cut cycle count by 27.46% on geomean against the OoO base. One limitation remains. Two endpoints in `synth/pipeline.vg` miss the 1000 ps clock by up to 797.58 ps, even though the netlist is functionally bit-equivalent to the RTL on every program.

A. References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Cambridge, MA, USA: Morgan Kaufmann, 2017.

[2] S. McFarling, "Combining Branch Predictors," Digital Equipment Corp. Western Research Laboratory, Palo Alto, CA, USA, Tech. Note TN-36, Jun. 1993.

[3] EECS 4340 course staff, "VeriSimpleV in-order RISC-V pipeline starter," Project 3 release, Columbia University, Spring 2026.

[4] *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*, Document Version 20191213, RISC-V Foundation, Dec. 2019.

[5] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proc. 24th Annu. Int. Symp. Microarchitecture (MICRO-24)*, Albuquerque, NM, USA, Nov. 1991, pp. 51–61.

References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Cambridge, MA, USA: Morgan Kaufmann, 2017.

[2] S. McFarling, "Combining Branch Predictors," Digital Equipment Corp. Western Research Laboratory, Palo Alto, CA, USA, Tech. Note TN-36, Jun. 1993.

[3] EECS 4340 course staff, "VeriSimpleV in-order RISC-V pipeline starter," Project 3 release, Columbia University, Spring 2026.

[4] *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*, Document Version 20191213, RISC-V Foundation, Dec. 2019.

[5] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proc. 24th Annu. Int. Symp. Microarchitecture (MICRO-24)*, Albuquerque, NM, USA, Nov. 1991, pp. 51–61.