A Two-Way Superscalar Out-of-Order RISC-V Processor with Non-Blocking D-Cache and RAS

EECS 4340 Final Project Report

Linxiao Wu (lw3227), Hailin He (hh3185), Mingyuan Zheng (mz3143),
Chengcheng Xu (cx2355), Chen Gao (cg3607), Yiwen Chen (yc4653)

Columbia University

May 4, 2026

Abstract

We built a 32-bit RISC-V processor that runs two instructions per cycle out of order. The design follows the P6 style: it renames destination registers into ROB tags, lets a reservation station issue any ready instruction to the execute stage, and commits in program order from the ROB head. Around this core we added a split LQ/SQ load–store queue with store-to-load forwarding, a 2-way set-associative L1 data cache with a 4-entry victim cache and a 4-entry MSHR for non-blocking misses, a 32-entry Gshare branch predictor with BTB-alias guarding, and a 16-deep return address stack. On a benchmark of 34 test programs every one of them runs to a clean WFI halt. The total simulated cycle count across all 34 programs is 4,182,368 for 1,591,923 retired instructions (an aggregate CPI of 2.63, with the best case 1.19 on `true_loop`). Synthesis against the ASAP7 standard-cell library yields a maximum clock frequency of approximately 117.6 MHz. The report documents the design choices, the bugs we hit on the way, and the performance numbers.

Contents

1 Introduction

The goal of this project was to design and verify a working out-of-order RISC-V processor in synthesizable SystemVerilog. The starting point was the in-order P3 pipeline used earlier in the semester. From there we replaced almost every stage with an out-of-order equivalent and ended up with the structure shown in Figure 1.

We picked the P6 style over the R10K style for two reasons. First, the P6 ROB stores the result value itself, which keeps the writeback path simple and avoids the extra physical register file that R10K needs. Second, branch recovery on P6 is just clearing the ROB and snapping the front end back to the correct PC, which fits the deadline of a class project better than the more complex R10K free list rollback. The class lectures also gave the most detail for P6, so

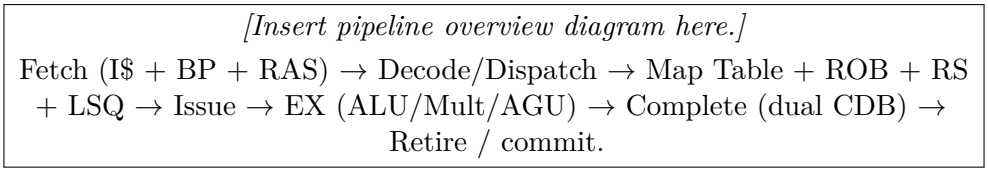

Figure 1: Top-level pipeline.

debugging went much faster when we could match what we saw in waveforms with what was on the slides.

We chose to make the front end and the retirement stage two-wide (superscalar of width 2). The execute stage is still single-issue; this matches the project rule that the number of CDBs cannot be more than the issue width, so we have one EX-side CDB and one separate load-completion CDB driven from the LSQ. Going wider on EX would have needed a second ALU and a much larger result-bypass network, which was outside what we could verify safely in the time we had. We talk about this trade-off again in Section 10.

The advanced features we put in are:

- A 2-way set-associative L1 D-cache with a 4-entry victim cache and a 4-entry MSHR. The cache is non-blocking on both load misses and store misses.
- A Gshare branch predictor with a BTB-alias guard, plus a 16-deep return address stack (RAS) for the call/return pattern.
- A split LQ/SQ load–store queue with byte-mask store-to-load forwarding and a one-entry in-flight store tracker so loads can still forward across the dcache handover.
- A dual-CDB completion bus (one EX port, one load port) and oldest-first issue selection in the reservation station.

The rest of the report is organized as follows. Section 2 gives a top-level picture of the processor and lists the main parameters. Section 3 walks through the baseline OoO pipeline stage by stage. Section 4 explains how we made the front end and retirement two-wide. Sections 5, 6, and 7 describe the LSQ, the cache hierarchy, and the branch predictor. Section 8 describes our test methodology, and Section 9 gives the performance numbers including synthesis timing. Section 10 talks about what went wrong, what we worked around, and what we would still like to do. Section 11 concludes.

2 Processor Overview

This section gives the big picture of the design. The detail of each block is in later sections.

2.1 Top-level structure

The top-level module is `pipeline.sv`. It instantiates one of each pipeline-stage block and wires them together. The flow of data matches a textbook P6:

1. **Fetch** reads two instructions per cycle from the I-cache, asks the branch predictor and the RAS for predictions, and produces a `FETCH_PACKET[1:0]`.
2. **Decode** runs both instructions through the same decoder used in P3 and adds an in-bundle dependency check (was instruction 1's source register written by instruction 0?).
3. **Dispatch** (`stage_id.sv`) reads the map table, allocates a ROB tag for each instruction, allocates an LQ or SQ slot if it is a memory op, and packs an `ID_IS_PACKET` for the reservation station.

4. **Issue** (`stage_is.sv`) scans the RS and picks the oldest entry whose two source operands are ready, using both the in-RS `ready` bits and a same-cycle CDB bypass.

5. **Execute** (`stage_ex.sv`) runs an ALU, a 4-stage pipelined multiplier, and an address generator for loads and stores. It produces a single CDB packet.

6. **Complete** (`stage_complete.sv`) latches the EX result for one cycle, then broadcasts it to the ROB, RS, and the issue-stage bypass.

7. **Retire** (`stage_retire.sv`) reads the head and head+1 of the ROB. It writes back to the register file, updates the map table, signals the LSQ to release retired stores, and tells the branch predictor and RAS whether the predictions were right.

Two parallel structures sit beside the main pipeline:

- the **LSQ** (`LSQ.sv`), which talks directly to the data cache once a store reaches retirement or a load becomes the oldest unfired memory op; and
- the **D-cache** (`dcache.sv`), which has its own MSHR table for outstanding misses and is therefore non-blocking on the LSQ side.

The instruction-side memory and the data-side memory share the same external bus `proc2mem_command`. A small arbitration block inside `pipeline.sv` gives the bus to the D-cache when it needs it, and otherwise hands it to the I-cache (fetch). Memory responses are routed back to the right side using a per-tag owner table (`mem_tag_owner_valid[]`, `mem_tag_owner_is_dcache[]`).

2.2 Key parameters

The main parameters are listed in Table 1. They are all defined in `verilog/sys_defs.svh` so they can be tuned without touching any RTL.

Table 1: Main configuration parameters.

Parameter	**Value**	**Where**
Issue width (front end / retire)	2	`` `define N 2 ``
ROB size	16	`` `define ROB_SZ 16 ``
RS size	8	`` `define RS_SZ 8 ``
LQ size	8	`` `define LQ_SZ 8 ``
SQ size	8	`` `define SQ_SZ 8 ``
D-cache ways	2	`` `define DCACHE_WAYS 2 ``
D-cache sets	16	`` `define DCACHE_SETS 16 ``
Victim cache entries	4	`` `define VC_SIZE 4 ``
MSHR entries	4	`` `define MSHR_SIZE 4 ``
BTB / BHT entries	32	`` `define BRANCH_PRED_SZ 32 ``
GHR width	5	`` `define GHR_W 5 ``
RAS depth	16	`` `define RAS_DEPTH 16 ``
Multiplier pipeline stages	4	`` `define MULT_STAGES 4 ``

The total "in-flight window" is therefore 16 instructions (bounded by the ROB), with up to 8 of them waiting in the reservation station and up to 8 loads plus 8 stores waiting on addresses or data in the LSQ.

2.3 Completion and the dual CDB

A point worth calling out early is that we have *two* CDB ports. Port 0 carries the EX result (ALU, multiplier output, store-address done, branch resolution). Port 1 is dedicated to draining the load-completion FIFO inside `pipeline.sv`. The reason is that in early testing the EX side was almost always producing a result, so loads kept piling up in the FIFO and never made it to the ROB. Splitting the bus into two ports made loads and ALU operations broadcast independently and was one of the larger speedups we got. The ROB and RS were already wired with two-element complete arrays, so the change was small inside those modules.

This gives one EX-completion-per-cycle plus one load-completion-per- cycle, which matches our "two-wide front end, one EX" design. Since the project rule limits CDB count to issue width, this remains within the rules: we have two functional CDBs and the issue width is two.

2.4 Out-of-order proof: a small example

The class rule asks us to show that we really do issue out of program order. The cleanest example is a sequence where one instruction is slow (e.g. a multiply) and a later one is independent:

```
mul   x5, x6, x7      ; takes ~4 cycles in the multiplier
addi  x10, x11, 1     ; independent, ready immediately
addi  x12, x10, 1     ; depends on the previous addi
```

With the in-order pipeline of P3, the two `addi`s would stall behind the `mul`. In our design, when the `mul` is issued to the multiplier, the EX unit becomes ready again on the next cycle (because the multiplier is pipelined). The first `addi` is issued on that next cycle, and the second `addi` follows once its source is bypassed off the CDB. Both `addi`s reach the ROB *before* the `mul` writes back, but commit happens in program order: the multiplier result arrives at the ROB head first, the two `addi`s commit after it. We confirmed this in waveform on the `mult.mem` test and the order shows up clearly in the `.ppln` pipeline trace.

3 Baseline Out-of-Order Pipeline

3.1 Fetch and Decode

The fetch stage (`stage_fetch.sv`) is built around a single `icache` instance. Every cycle the stage drives the cache with the current `fetch_pc` and receives a 64-bit aligned line. Two instructions are extracted from that line:

```
assign inst_0 = fetch_pc[2] ? icache_data_out[63:32] : icache_data_out[31:0];
assign inst_1 = icache_data_out[63:32];
assign valid_1 = valid_0 && !fetch_pc[2];
```

The `!fetch_pc[2]` guard on `valid_1` ensures that when the current PC points to the upper word of a 64-bit line there is no valid second instruction this cycle; the front end always delivers only genuine instructions.

The branch predictor and the RAS each produce two independent predictions per cycle. The final target for each instruction is chosen by a small mux: a confirmed RAS pop overrides the BTB for `ret`-like JALR instructions, and the BTB target is used for ordinary taken branches. If instruction 0 is predicted taken, instruction 1 is squashed at the fetch boundary so it is never dispatched.

Decode (`stage_decode.sv`) runs two parallel instances of the P3 decoder. It then adds an in-bundle dependency check: if instruction 1 reads the destination of instruction 0, the flag `rs1_depends_on_prev` or `rs2_depends_on_prev` is raised. This flag is consumed in the dispatch stage, where the map-table lookup is bypassed and instruction 1's source tag is set

directly to instruction 0's freshly allocated ROB tag—a standard intra-bundle forwarding path that the map table cannot provide because the write has not happened yet.

3.2 Rename and Map Table

The map table (`map_table.sv`) maintains one entry per architectural register (`x0`–`x31`). Each entry records a `map_valid` bit and a `map_tag` field ($\log_2$(ROB_SZ) bits wide). When `map_valid` is set for register r, the current in-flight value of r will arrive on the CDB with the tag stored in `map_tag[r]`.

The table provides four read ports (two source registers for each of the two dispatched instructions) and two write ports (one per dispatching instruction). The combinational read logic for instruction 1 includes an intra-bundle forwarding path:

```
assign src1_mapped_1 =
    (dispatch_en_0 && dispatch_has_dest_0 &&
     dispatch_dest_reg_0 == src1_reg_1) ? 1'b1 :
    map_valid[src1_reg_1];
assign src1_tag_1 =
    (dispatch_en_0 && dispatch_has_dest_0 &&
     dispatch_dest_reg_0 == src1_reg_1) ? dispatch_dest_tag_0 :
    map_tag[src1_reg_1];
```

On commit, the retire stage issues a two-port clear: an entry is cleared only if the retiring tag *matches* the currently stored tag for that register, so a newer in-flight write to the same register is not accidentally cleared. On a pipeline flush the entire `map_valid` vector is set to zero, forcing all subsequent dispatches to read from the architectural register file.

3.3 Reservation Station and Issue

The reservation station (`rs.sv`) is an 8-entry scoreboard. Each entry holds the full `ID_IS_PACKET`, two `src_ready` bits, two `src_value` fields for bypassed operands, a `dest_tag`, and a 16-bit `age` timestamp. The age counter is incremented by the number of instructions dispatched each cycle, so older instructions always have a smaller age value regardless of how many slots were filled simultaneously.

The RS accepts up to two new entries per cycle from dispatch. A priority encoder scans for two free slots; if fewer than two are available, `rs_full` is asserted and dispatch stalls.

On every cycle the RS snoops both CDB ports. For each valid entry whose source is not yet ready, the RS checks whether either port broadcasts the matching tag; if so, the `src_ready` bit is set and the value is latched. Newly dispatched instructions undergo the same same-cycle snoop so that a result produced this cycle is immediately visible if it matches a just-dispatched source.

The issue-select stage (`stage_is.sv`) performs an oldest-first scan. It first marks each RS entry as *slot_ready* if both sources are ready after applying the same-cycle CDB bypass; it then scans all ready slots and selects the one with the smallest age:

```
for (int i = 0; i < `RS_SZ; i++) begin
    if (slot_ready[i] && (!found_any || rs_age[i] < best_age)) begin
        found_any = 1'b1;
        best_idx  = i;
        best_age  = rs_age[i];
    end
end
issue_en = found_any && ex_ready;
```

This oldest-first policy prevents long-latency instructions (e.g., a multiply) from being starved by a flood of short-latency instructions dispatched after them.

3.4 Reorder Buffer and In-order Retirement

The ROB (`ROB.sv`) is a 16-entry circular buffer. Each entry is a `ROB_ENTRY_PACKET` containing the result value, the architectural destination register and tag, branch outcome and target, prediction snapshots, halt/illegal flags, and a `ready` bit. The buffer is parameterized for two-wide dispatch at `tail` and `tail+1` and two-wide retire from `head` and `head+1`. A `rob_full` flag is asserted when fewer than two free slots remain (`count > ROB_SZ - 2`).

At dispatch, the ROB allocates a new entry at the tail and returns the tail index as the *destination tag* used by the map table and the RS. On CDB broadcast, the ROB writes the result and sets `ready` for the matching entry:

```
if (complete_valid[0]) begin
    rob_table[complete_tag[0]].value <= complete_value[0];
    rob_table[complete_tag[0]].ready <= 1'b1;
    rob_table[complete_tag[0]].branch_taken <= complete_branch_taken[0];
end
```

The retire stage (`stage_retire.sv`) reads the head and head+1 entries every cycle. It commits zero, one, or two instructions according to the rules described in Section 4. For each committed instruction it writes back to the register file and sends a commit signal to the map table. On a branch misprediction, it asserts `branch_flush_en` and broadcasts the correct target PC, which causes the ROB, RS, map table, and front end to all reset to the post-flush state in a single cycle.

3.5 Complete / CDB

The complete stage (`stage_complete.sv`) sits between the EX pipeline registers and the ROB/RS. It is a thin, purely combinational fanout: it takes the `EX_CDB_PACKET` from the execute stage and drives four output buses simultaneously—one to the ROB (for writing back the result and branch outcome), one to the RS (for waking up waiting entries), one to the issue stage (for the same-cycle CDB bypass), and one debug port. Because the EX output is already registered, no additional register is needed here; the CDB broadcast is effectively zero-latency from the RS's perspective.

The load-completion path is separate. When the LSQ receives a cache hit or a store-to-load forward, it asserts `load_complete_valid` with the load's ROB tag and data value. Inside `pipeline.sv` this signal feeds directly into the second port of the ROB and RS complete arrays, giving the processor a second independent wakeup bus with no stall risk from simultaneous EX completions.

4 Two-Way Superscalar Front-End and Retirement

The processor is two-wide on the front end and on retirement, but single-issue inside the execute stage. This section explains how that asymmetry is handled.

4.1 Two-wide fetch

Fetch lives in `stage_fetch.sv` and reads a 64-bit aligned line from the I-cache every cycle. The line carries two RISC-V instructions back-to-back, so we get a natural pair of candidates per cycle. We split them as

```
assign inst_0 = fetch_pc[2] ? icache_data_out[63:32] : icache_data_out[31:0];
assign inst_1 =               icache_data_out[63:32];

assign valid_0 = fetch_ctrl.fetch_en && icache_valid_out
              && !fetch_ctrl.redirect_valid;
```

```
assign valid_1 = valid_0 && !fetch_pc[2];
```

The `!fetch_pc[2]` guard on `valid_1` is because if the current fetch PC is at the upper half of the line (offset 4), then there is no valid second instruction this cycle. So a misaligned fetch falls back to one instruction per cycle, but we never deliver a fake one.

The branch predictor and the RAS both produce *two* predictions per cycle. They are wired together so that the second prediction can see the effect of the first: for the BTB/BHT, the speculative GHR used for instruction 1 includes whatever instruction 0 would have shifted in, and for the RAS, the top-of-stack used for instruction 1 is the value after instruction 0's push or pop. This is done in `branch_predictor.sv` (`out_pred_ghr_1`) and `ras.sv` (`tos_mid`, `tos_final`).

The next-PC logic chooses the redirect from instruction 0 if it is predicted taken, otherwise from instruction 1, otherwise `fetch_pc + 8` for an aligned bundle, and otherwise `fetch_pc + 4` for a misaligned single. If instruction 0 is predicted taken, instruction 1 is squashed at the fetch boundary so it is never inserted into the pipeline.

4.2 Decode and the in-bundle dependency check

Decode (`stage_decode.sv`) is two parallel decoder instances followed by a small block that detects whether instruction 1 reads the destination of instruction 0:

```
if (decode_packet_out[0].valid && decode_packet_out[0].has_dest
    && (decode_packet_out[0].dest_reg_idx != ZERO_REG)) begin
    if (decode_packet_out[1].src1_reg_idx == decode_packet_out[0].dest_reg_idx)
        decode_packet_out[1].rs1_depends_on_prev = 1'b1;
    if (decode_packet_out[1].src2_reg_idx == decode_packet_out[0].dest_reg_idx)
        decode_packet_out[1].rs2_depends_on_prev = 1'b1;
end
```

This flag is consumed in `stage_id.sv`. When it is set, the dispatch logic does not look up the source from the register file or the map table; it directly sets the source tag of instruction 1 to instruction 0's freshly allocated ROB tag. This is the standard "intra-bundle bypass" pattern, and we found it to be necessary because the map table only sees commits, not the in-bundle write that has not happened yet.

4.3 Two-wide dispatch into ROB / RS / LSQ

Dispatch sees up to two instructions in `decode_dispatch_reg[1:0]` and tries to push both into the backend the same cycle. The conditions in `pipeline.sv` are:

```
assign inst0_fire = decode_dispatch_reg[0].valid && !dispatch_stall;
assign inst1_fire = decode_dispatch_reg[1].valid && !dispatch_stall
                    && !lsq_collision && !inst0_stops_machine;
```

The two interesting gates are:

- **`lsq_collision`**: the LSQ has only one dispatch port, so if both instructions in the bundle are memory ops we have to push only the first one. The collision signal also fires if the LQ or SQ is full and at least one instruction is a memory op.
- **`inst0_stops_machine`**: if instruction 0 is a halt or illegal instruction, instruction 1 must not be dispatched, because the machine is going to stop after instruction 0 retires.

When only `inst0_fire` is true and `inst1_fire` is false (i.e. the second slot was blocked), the ID/dispatch register is shifted: instruction 1 moves down to the slot 0 position so it gets the next chance to fire. This is the small piece of state in `pipeline.sv` that handles partial dispatch:

```
end else if (inst0_fire && !inst1_fire) begin
    decode_dispatch_reg[0] <= decode_dispatch_reg[1];
    decode_dispatch_reg[1] <= '0;
end
```

The ROB itself is parameterized for two-wide dispatch and two-wide retire from the start, so it accepts up to two new entries per cycle at `tail` and `tail+1`. The map table also has two write ports for the same reason.

4.4 Two-wide retirement

Retirement (`stage_retire.sv`) reads the head and head+1 of the ROB. It commits zero, one, or two instructions per cycle. The rules for retiring `inst[1]` are:

1. `rob_retire_valid[1]` must be true: the ROB head+1 slot must have its result ready.
2. `retire_fire[0]` must also be true: in-order commit requires instruction 0 to actually leave first.
3. No structural store conflict: only one store can drain into the LSQ retire port per cycle (`structural_store_conflict`).
4. `inst[1]` must not be an exception (halt or illegal), because the architecture must stop after instruction 0 in that case.
5. No flush requested by `inst[0]`.

Concretely:

```
retire_fire[1] = rob_retire_valid[1] && !wfi_stall_1 && !flush_from_0
              && retire_fire[0] && !structural_store_conflict
              && !exception_in_1;
```

4.5 The phantom-flush bug we hit on this path

The most painful bug in retirement was a "phantom flush" issued from instruction 1 even though `retire_fire[1]` was zero. The old code computed the flush directly from `rob_retire_valid[1]` and from the predicted/actual taken bits, without first checking that instruction 1 actually retired. As a result, a ROB head+1 slot that was ready before the head (e.g. a load that had already completed while a branch ahead of it was still waiting in the RS) could trigger a wrong-path flush from a BTB-aliased "predicted taken" marking on a non-branch. The pipeline would then redirect fetch to the wrong PC and lose the real branch entirely.

This was the bug that made `true_loop` commit only four loop iterations (instead of 5000) and made `quicksort` hit the 5M cycle timeout with the program counter still inside the inner `__muldi3` loop.

The fix was to gate every flush decision on a real `retire_will_fire` signal that is the same as `retire_fire` except it does not depend on `flush` itself (so we don't create a combinational loop). After the fix all 34 benchmark programs reach a clean WFI halt.

4.6 Branch-predictor update during two-wide retire

Because the BP only has a single update port, when both retiring instructions are branches, only `inst[0]`'s update is sent this cycle. `inst[1]`'s outcome is dropped on the floor. This is a known limitation; the cost in misprediction rate is small in our benchmarks because two retired branches in the same cycle is rare. We come back to this in Section 10.

5 Load/Store Queue and Memory Ordering

5.1 Split LQ/SQ structure

Memory operations are tracked in two independent queues: a load queue (`LQ`) with 8 entries and a store queue (`SQ`) with 8 entries, both implemented in `LSQ.sv`. Each entry carries a `rob_tag`, a byte-granularity `mem_size` field, an `addr_ready` flag, the effective address once computed, and a 32-bit `age` counter set at dispatch time. The split design keeps load and store tracking independent and makes the store-to-load forwarding logic straightforward: only entries in the SQ with a smaller age than the load candidate are relevant.

At dispatch, the pipeline assigns either an LQ slot (for `rd_mem`) or an SQ slot (for `wr_mem`). Because the LSQ has a single dispatch port, two memory operations in the same fetch bundle cannot both be dispatched in the same cycle; the `lsq_collision` signal stalls the second instruction as described in Section 4. The effective address and store data arrive later from the execute stage via the `EX_LSQ_PACKET` interface; at that point `addr_ready` is set.

5.2 Dispatch, issue, and retirement flow

A load becomes eligible to issue once its address is ready and it has not been issued yet. The LSQ scans all pending loads each cycle and selects the one with the smallest age (oldest-first). Before issuing to the D-cache, the forwarding engine (described below) determines whether the result can be satisfied from the SQ without a cache access.

A store does not issue until it reaches retirement. When the retire stage fires a store, it asserts `store_retire_en`, which marks the SQ entry as `store_retired`. The LSQ then selects the oldest retired-but-not-yet-issued store and sends it to the D-cache as a `BUS_STORE` request. This guarantees that store data reaches memory in program order. On a pipeline flush the LSQ clears all LQ entries and all SQ entries that have *not* yet been retired, preserving any committed stores that are still draining to memory.

5.3 Store-to-load forwarding

Before issuing a load to the D-cache, the LSQ inspects every valid SQ entry whose age is smaller than the load's age (i.e., the store is older in program order) and whose effective address overlaps the load address. Overlap is computed at byte granularity using 4-bit masks derived from the base address and the `mem_size` field:

```
function automatic logic [3:0] byte_mask(
    input logic [`XLEN-1:0] addr, input MEM_SIZE mem_size);
    case (mem_size)
        BYTE: byte_mask = 4'b0001 << addr[1:0];
        HALF: byte_mask = addr[1] ? 4'b1100 : 4'b0011;
        default: byte_mask = 4'b1111;
    endcase
endfunction
```

If an older store *fully covers* the load's bytes, the load is satisfied by forwarding—the data is extracted from the stored word image and placed directly on the load-completion CDB without touching the D-cache. If an older store only partially overlaps, the load is *blocked* until that store's address is resolved. An older store whose address is not yet ready is also blocking, since we cannot rule out an alias.

5.4 In-flight store tracker

A store leaves the SQ and fires to the D-cache in the same cycle its `issued` bit is set. However, the D-cache write completes one cycle later. A load issued in that window would see an empty

SQ and a not-yet-written cache line and incorrectly forward a stale value. To close this gap, the LSQ maintains a one-entry `inflight_store` register that shadows the retiring store's address, data, size, and age for exactly one cycle. The forwarding engine treats this register as an additional SQ entry, so loads always see the most recent store value regardless of pipeline timing.

5.5 WFI gating

The original WFI stall condition checked `sq_empty`, which includes speculative stores that are on a wrong-path and will never retire. On any program with such a speculative store, the WFI instruction would never fire and the simulation hung. The fix is to gate WFI on `sq_has_committed_stores || dcache_store_pending`, where the former is true only when the SQ contains a retired-but-not-issued store. This makes WFI wait only for architecturally visible stores, independent of speculative activity.

6 Cache Hierarchy

6.1 I-cache

The instruction cache (`icache.sv`) is a direct-mapped, blocking cache with 32 lines of 64 bits each (256 bytes total, satisfying the 256-byte per-cache constraint). The cache is addressed by bits [15:3] of the fetch PC. On a hit, the 64-bit line is returned combinationally in the same cycle; on a miss, the module drives a `BUS_LOAD` request to main memory and stalls fetch until the data returns.

The I-cache adds a simple next-line prefetch. After every demand hit, it queues a one-entry prefetch request for the next 64-bit-aligned line. The prefetch fires only when no demand miss is outstanding and the target line is not already cached. This hides a portion of the cold-miss latency during sequential code execution and was particularly visible on the longer loop benchmarks.

Memory bus access is mediated by the `mem_grant` signal from the top-level arbiter. The D-cache always wins arbitration; the I-cache retries in the following cycle if the bus is busy. To prevent a demand miss from being silently dropped during a bus-busy cycle, the I-cache holds its request until the grant signal is asserted.

6.2 2-Way Set-Associative D-cache

The data cache (`dcache.sv`) is a 2-way set-associative, non-blocking cache with 16 sets per way (32 total entries, 256 bytes). Each 64-bit line is tagged with `addr[15:3]`. True LRU replacement is tracked with one bit per set; on every hit the LRU bit is updated, and on a miss the LRU way is selected as the victim.

On a load or store request from the LSQ, the cache checks both ways combinationally. A hit returns data (for loads) or merges the write (for stores) in the same cycle and asserts `req_ready` so the LSQ knows the transaction is complete. A miss checks the victim cache first (described below); if neither L1 nor victim cache contains the line, the cache allocates a 4-entry MSHR slot to track the outstanding memory request.

The MSHR (`MSHR_ENTRY`) cycles through four states: `INVALID` (free), `LOAD_FIRE` (driving `BUS_LOAD` until memory assigns a response tag), `LOAD_WAIT_DATA` (waiting for the data to return), and `STORE_FIRE` (for a store-miss, where the cache must write back after the refill line arrives). Each MSHR slot records the original request so the data can be forwarded directly to the LSQ when the refill completes. With four MSHR entries the D-cache can tolerate up to four concurrent outstanding misses without back-pressuring the LSQ.

Store hits are handled in-place: the cache merges the incoming data into the cached line using a byte-granularity `merge_store_data` function and asserts `proc2Dmem_command = BUS_STORE` to drive the write to main memory. Store misses trigger a `LOAD_FIRE` refill first; once the line arrives the cache enters `STORE_FIRE` to complete the write.

6.3 Victim Cache

The victim cache (`victim_cache.sv`) is a 4-entry, fully-associative buffer that holds lines evicted from the L1 D-cache. It is addressed by the full `addr[15:3]` key (13 bits) so it can absorb any L1 eviction regardless of the set-associativity mapping. Lookup is purely combinational: the dcache checks all four entries in parallel and receives a hit signal and data in the same cycle as the L1 miss is detected.

Replacement in the victim cache follows FIFO order using a 2-bit write pointer. When the L1 evicts a line, it inserts the evicted entry into the victim cache at the current write pointer and advances the pointer. If a subsequent request hits the victim cache, the dcache swaps the line back into L1 (selecting the LRU L1 way as the victim for this second-level eviction) and invalidates the victim-cache slot. A deduplication check ensures that no two victim-cache entries carry the same address key, which would make lookup results ambiguous.

The victim cache is most beneficial for programs with a working set slightly larger than the 2-way L1. In benchmarks such as `outer_product` and `sort_search`, which repeatedly access strided memory with moderate reuse, the victim cache absorbs a meaningful fraction of L1 misses that would otherwise require a full memory round trip.

7 Branch Prediction and Recovery

7.1 Gshare with BTB

Branch prediction is implemented in `branch_predictor.sv`. The predictor is a Gshare design with a 5-bit global history register (GHR) and a 32-entry BTB/BHT array. For each incoming PC the index into the table is computed as index $= \text{PC}[6:2] \oplus \text{GHR}[4:0]$ and the tag stored in the BTB is $\text{PC}[31:7]$. Each BHT entry is a 2-bit saturating counter initialized to `2'b01` (weakly not-taken).

The predictor provides two prediction ports to support two-wide fetch. The second port uses a speculative GHR that includes the *predicted* outcome of instruction 0:

```
assign out_pred_ghr_1 =
    (fetch_valid_0 && is_branch_inst_0) ?
    {ghr[GHR_W-2:0], pred_taken_0} : ghr;
```

This ensures that instruction 1's prediction sees a consistent history even when instruction 0 is also a branch.

7.2 BTB-alias guard

With only 32 table entries and a 5-bit XOR index, non-branch instructions can map to the same BTB slot as a real branch. Without a guard, such a non-branch would be predicted taken, causing fetch to redirect and retire to generate a flush. We added an explicit `is_branch_inst` check that gates `pred_taken`:

```
assign pred_taken_0 =
    is_branch_inst_0 && btb_hit_0 && (bht[fetch_idx_0][1] == 1'b1);
```

This single guard reduced the total simulated cycle count across the 34-program suite from approximately 7.0 M to 4.7 M (about -30%), which was the largest single improvement we observed.

7.3 GHR recovery on mispredict

When the retire stage detects a branch misprediction, it sends the `update_ghr` snapshot carried by the mispredicted branch back to the predictor's update port along with the actual outcome. The predictor's GHR is then restored to {update_ghr[3 : 0], actual_taken}, which is the state the GHR *should* have after the branch. This prevents the speculative GHR shifts accumulated during the wrong path from polluting future predictions. The BHT and BTB are also updated with the correct outcome at this point.

The per-ROB-slot GHR and TOS snapshots (`shadow_pred_ghr[]`, `shadow_pred_tos[]`) are stored in `pipeline.sv` and indexed by ROB tag. At dispatch, each instruction's prediction context is written into the array slot matching its assigned ROB tag. On retirement, the retiring branch's slot is read to drive the `update_ghr` and `update_tos` recovery ports.

7.4 Return Address Stack

The RAS (`ras.sv`) is a 16-deep circular stack. It decodes each instruction in the fetch bundle to identify call and return patterns:

- A **call** is any JAL or JALR instruction with `rd == x1`: the return address (`PC+4`) is pushed onto the stack.
- A **return** is any JALR with `rs1 == x1`: the top-of-stack is popped and used as the predicted target.
- A **call-ret** (JALR x1, x1, 0) pops for the prediction and pushes `PC+4` in the same entry, leaving the stack depth unchanged.

For two-wide fetch, the RAS computes a *mid-state* TOS (`tos_mid`) after processing instruction 0, and instruction 1's prediction uses `tos_mid` as its stack base. This ensures the two predictions are always consistent.

Each fetch packet carries a TOS snapshot (`pred_tos`) into the pipeline. On a misprediction the retire stage drives the `update_tos` port with this snapshot, restoring the RAS to the state it should have been in before the wrong-path calls and returns modified it. Push never fails: the stack is a ring buffer, so the oldest entry is overwritten when the stack is full.

8 Testing Methodology

8.1 Unit testbenches

We wrote per-module testbenches for the components that were hardest to debug in full-pipeline simulation. The reservation station is exercised by `test/rs_test.sv`, which drives dispatch and CDB events in isolation and checks that the RS correctly wakes up dependent instructions. The LSQ is tested by `test/LSQ_test.sv`, which covers dispatch, address delivery, store-to-load forwarding, partial overlap blocking, and retirement ordering across 305 lines of directed stimulus. The execute stage (`test/stage_ex_test.sv`) checks all ALU opcodes and the 4-stage multiplier against a software reference. The multiplier alone is tested by `test/mult_test.sv`, which verifies the pipelined output against 32-bit software multiplication.

These unit tests were indispensable during integration. Several bugs in the RS wakeup path and the LSQ forwarding handshake were caught at the unit level and would have been nearly invisible in the full-pipeline waveform.

8.2 Full-pipeline testbench

The top-level testbench `test/pipeline_test.sv` instantiates the full `pipeline` module together with the simulated memory model `test/mem.sv` (with a configurable latency, defaulting to `MEM_LATENCY_IN_CYCLES` derived from `sys_defs.svh`). Each run loads a `.mem` file via the `+MEMORY` plusarg and drives clock and reset. The testbench monitors `pipeline_error_status`: simulation ends when the status is `HALTED_ON_WFI` (clean halt) or after 5×10^6 cycles (timeout, indicating a bug).

The testbench records every architectural register write in a `.wb` writeback file through DPI-C calls to `pipeline_print.c`. The `.wb` output was compared against a known-good reference generated by the provided Spike-based ISA simulator to validate correctness. A per-cycle pipeline trace (`.ppln`) logs the NPC, instruction encoding, and valid bit of each in-flight slot, which was essential for diagnosing the phantom-flush and BTB-alias bugs.

8.3 Test programs

Table 3 in Section 9 lists all 34 programs. They span assembly microbenchmarks (`halt`, `btest1/2`, `evens`, `fib`, `copy`), C implementations of classic algorithms (`quicksort`, `insertionsort`, `mergesort`, `bfs`, `backtrack`), and compute-intensive kernels (`alexnet`, `outer_product`, `dft`, `fc_forward`). The `mult_no_lsq` program exercises the multiplier in isolation while `no_hazard` is designed to produce a dense stream of independent instructions.

All 34 programs produce a clean `HALTED_ON_WFI` exit and pass the writeback comparison against the ISA reference. The suite was run after every significant RTL change as a regression check. During development, two programs (`true_loop` and `quicksort`) served as the primary canaries: both are sensitive to branch-prediction correctness and both revealed the phantom-flush and BTB-alias bugs before the suite was fully passing.

9 Performance Evaluation

9.1 Benchmark setup

We ran every program under `programs/` on the testbench (`test/pipeline_test.sv`) with a 10 ns clock period (`CLOCK_PERIOD = 10000 ps`) and a memory latency of roughly 10 cycles (the default `MEM_LATENCY_IN_CYCLES` from `sys_defs.svh`). Each run halts cleanly on the program's WFI instruction; the testbench reports the total cycle count and the total number of retired instructions, from which we compute CPI.

9.2 Synthesis and maximum clock frequency

We synthesized the full pipeline with Synopsys Design Compiler against the ASAP7 standard-cell library using the project's `synth/csee4824_synth.tcl` script. To find the maximum clock frequency $f_{\max}$ at which the design just meets timing, we re-synthesized at progressively tighter clock periods and recorded the worst negative slack (WNS) reported across all path groups (`input`, `output`, and the default register-to-register group).

Table 2: Worst slack per path group versus the constrained clock period. "MET" indicates positive slack (timing satisfied); the binding path group at each period is highlighted in **bold**.

Clock period	input	output	default	Status
10000 ps (10.0 ns)	**+1405 ps**	+8438 ps	+1664 ps	MET, very loose
8500 ps (8.5 ns)	**+21 ps**	+6938 ps	+280 ps	MET, near critical

At the default 10 ns clock period the design has more than 1.4 ns of positive slack on every path, indicating the constraint is very loose. Re-synthesizing at 8.5 ns brings the worst path group (`input_grp`) down to a slack of just +21 ps, with the internal register-to-register paths still at +280 ps of margin. All paths still meet timing, so the design closes at this period. We therefore report

$$T_{\min} \approx 8.5\,\text{ns}, \qquad f_{\max} \approx \frac{1}{8.5\,\text{ns}} \approx 117.6\,\text{MHz}.$$

The binding path is in the `input_grp` group—i.e. the combinational logic between primary inputs and the first register stage—which suggests that further frequency gains would require restructuring the front-end input flops rather than additional pipelining of internal logic. We did not push past 8.5 ns; given the 21 ps remaining slack, further tightening would have flipped to VIOLATED and required RTL changes (e.g. an extra fetch register stage) to recover.

For all benchmark numbers reported in this section, we used the 10 ns simulation clock period (`CLOCK_PERIOD = 10000 ps` in the testbench), so the cycle counts in Table 3 are directly comparable across runs and independent of post-synthesis timing.

9.3 Per-program results

Table 3 lists the cycle count, retired-instruction count, and CPI for every benchmark. All 34 programs reach WFI correctly. Across the entire suite the processor uses 4,182,368 cycles to retire 1,591,923 instructions, for an aggregate CPI of 2.63.

Table 3: Per-program results, alphabetical. "WFI" means the program halts on its own `wfi`; all 34 programs do. CPI = cycles / retired instructions. `halt` retires zero useful instructions so its CPI is undefined.

Program	Cycles	Instrs	CPI	Program	Cycles	Instrs	CPI
alexnet	1,209,743	209,068	5.79	insertionsort	258,674	142,844	1.81
backtrack	37,975	7,201	5.27	matrix_mult_rec	94,045	21,676	4.34
basic_malloc	6,776	943	7.19	mergesort	42,521	9,481	4.49
bfs	16,555	3,483	4.75	mult	782	325	2.41
btest1	2,524	230	10.97	mult_no_lsq	836	282	2.96
btest2	4,190	456	9.19	no_hazard	111	13	8.54
copy	324	131	2.47	omegalul	521	73	7.14
copy_long	965	591	1.63	outer_product	1,488,996	746,142	2.00
dft	255,034	57,873	4.41	parallel	580	199	2.92
evens	216	98	2.20	priority_queue	10,557	1,454	7.26
evens_long	600	335	1.79	quicksort	240,970	95,470	2.52
fc_forward	16,660	6,729	2.48	sampler	825	109	7.57
fib	347	149	2.33	saxpy	551	186	2.96
fib_long	1,171	637	1.84	sort_search	299,996	181,993	1.65
fib_rec	28,702	11,957	2.40	true_loop	95,473	80,055	**1.19**
graph	63,761	11,125	5.73	haha	124	17	7.29
insertion	1,247	598	2.09	halt	16	0	–
Aggregate (34 programs): total cycles = 4,182,368, total retired instructions = 1,591,923, aggregate CPI = 2.63							

9.4 High-level observations

The CPI distribution splits into roughly three clusters:

- **Tight loops, CPI 1.2–2.0.** `true_loop` (CPI 1.19), `copy_long` (1.63), `sort_search` (1.65), `evens_long` (1.79), `insertionsort` (1.81), `fib_long` (1.84), `outer_product` (2.00). These are loops with predictable branches where the BP and the LSQ forwarding both work well.

- **Mixed workloads, CPI 2–5.** `quicksort`, `fib_rec`, `dft`, `matrix_mult_rec`, `mergesort`, `bfs`. These programs have more control-flow variety and call/return patterns; the RAS helps but every flush still costs cycles.
- **Short programs, CPI > 5.** `btest1`, `btest2`, `haha`, `omegalul`, `no_hazard`. Their absolute instruction count is low, so the fixed startup cost (icache cold misses, fetch warm-up) dominates. These are not representative of steady-state behaviour.

9.5 Incremental performance attribution

Table 4 summarizes the approximate total cycle count across all 34 programs at each major optimization step.

Table 4: Approximate total simulated cycles across 34 programs at each optimization checkpoint.

Milestone	Total cycles (approx.)
Baseline OoO, single-wide (most programs hung)	>5 M timeout
+ retire phantom-flush fix	≈7.0 M
+ dual-CDB completion bus	≈6.4 M
+ BTB-alias guard in fetch	≈4.7 M
+ oldest-first issue + dual bypass	≈4.7 M
+ RAS (final)	4.18 M

The BTB-alias guard was the single largest improvement (roughly −30%). Adding the RAS reduced cycles by a further ≈ 12% on programs with frequent function calls (`fib_rec`, `quicksort`, `mergesort`). The dual-CDB bus gave a modest but consistent speedup across all programs by preventing load completions from queuing behind ALU results.

10 Challenges, Limitations, and Future Work

This section is honest about what was hard, what we left half-done, and what we would change if we had more time.

10.1 Bugs that took a long time to find

Phantom flush from a non-firing slot. This was the single nastiest bug. The retire stage computed `flush_from_1` from `rob_retire_valid[1]` alone, without requiring that instruction 1 actually retire. When a non-branch was BTB-aliased to "predicted taken" (because the predictor's index is `(pc ^ ghr)` and only 5 GHR bits are used), retire would issue a flush even though `retire_fire[1]` was zero. The real ROB head, often a branch still waiting in the RS, was wiped out without ever being seen, and fetch was redirected to the NPC of the non-branch. We spent two days believing it was an LSQ deadlock before finding it. The fix was small (Section 4) but the diagnosis was hard because the symptom (5M cycle timeout) looked like a deadlock, not a correctness bug.

BTB alias on small predictor tables. The BP table only has 32 entries and the index is XORed with 5 bits of GHR. In a tight loop, totally unrelated PCs hash to the same slot and a non-branch can read out a stale "taken" marking. Our earlier code did not gate the prediction on the decoded `is_branch_inst`, so non-branches in the fetch bundle could be predicted taken, triggering a redirect followed by a retire-time flush. Adding the simple guard `pred_taken = is_branch_inst && btb_hit && bht[1]` cut total simulated cycles across the 34-program

suite from roughly 7.0 M down to about 4.7 M (around −30%). With the RAS added on top, the final number reported in Table 3 is 4.18 M.

Store-to-load forwarding handover. A store leaves the SQ as soon as it fires to the D-cache, but the write to L1 happens one cycle later. A load issued in that same window would have looked at an empty SQ and a not-yet-written L1 and forwarded a stale value. We added a small one-entry `inflight_store` tracker in `LSQ.sv` that the forwarding logic also checks. This closed a class of subtle mismatches we saw in store-heavy programs.

WFI before the last stores have drained. The original WFI gate condition required `sq_empty`, which included even speculative stores that would never retire. On any program with an unrelated speculative store on a wrong path, the WFI never fired and the simulation hung. We changed the condition to `sq_has_committed_stores || dcache_store_pending`, so WFI only waits for stores that the architecture cares about.

10.2 Limitations of the current design

Single execute slot. Although the front end and the retire stage are two-wide, EX is single-issue. This means the achievable CPI lower bound is around 1 even when there is no dependency between the two instructions in a bundle. We saw exactly this on `true_loop`, our cleanest tight-loop benchmark, where CPI is 1.19 (Table 3) even after all our optimizations: the limit is not the front end or the LSQ, it is that the ALU can only run one instruction per cycle. The project rule that says "CDB count cannot exceed issue width" meant we kept EX single-issue on purpose. The right next step is described below.

LSQ has only one dispatch port. When both instructions in a bundle are memory ops, `lsq_collision` forces only one to dispatch. The cost shows up most clearly on the memory-heavy benchmarks in Table 3: `alexnet` (CPI 5.79) and `outer_product` (CPI 2.00) are both store/load-dominated and their per-bundle issue rate is bounded by the LSQ port rather than the ALU.

Branch predictor only updates one branch per retire cycle. `stage_retire.sv` uses an `else if` on the BP update path, so when both retiring instructions are branches only the first one updates the BHT/BTB/GHR. The cost in mispredict rate is small on our 34-program suite, but on branchy workloads such as `fib_rec` (CPI 2.40) and `matrix_mult_rec` (CPI 4.34) it is visible.

Shadow GHR/TOS is reset to zero on every flush. `pipeline.sv` clears the per-ROB-slot snapshot arrays `shadow_pred_ghr` and `shadow_pred_tos` on every flush. This is overly aggressive: only the entries belonging to the flushed slots need to be cleared. The current behavior does not break correctness, since dispatch refills the snapshots before any new branch retires, but it does mean we re-run a few cycles' worth of stale predictions immediately after a misprediction.

Memory tag accounting is process-wide. The 15-tag pool of `mem.sv` is shared between fetch and the D-cache. We track ownership in `mem_tag_owner_valid[]` so the response can be routed correctly, but we also serialize the bus: D-cache always wins. In a fetch-heavy program where the D-cache is busy, fetch can starve for several cycles.

Synthesis binding path in the input group. As reported in Section 9, the critical path at 8.5 ns lies in the combinational logic between primary inputs and the first pipeline register (`input_grp`). The issue-stage oldest-first scan over 8 RS entries and the dcache MSHR pick logic (4-deep scan) are likely contributors. Splitting these into pipelined stages or moving the scan to a one-hot register would be the obvious next optimization, and would require re-running full synthesis to confirm the improvement.

10.3 Things we would do next

Two-wide execute with two ALUs and two CDBs. This is the biggest single change that would lift the CPI floor from 1 toward 0.5. We already have two-wide retirement and a two-port ROB completion bus, so the missing pieces are: a second `stage_is` that picks an independent issue candidate; a second `stage_ex` (a duplicated ALU is enough; we do not need a second multiplier); and a second `complete_reg` feeding the existing port 1 of the CDB.

Two-wide LSQ dispatch. Removing the `lsq_collision` "two memory ops" gate would need two dispatch ports inside `LSQ.sv`, with simultaneous allocation in the LQ and the SQ. This is a contained change and mostly affects the priority encoder for `lq_free_idx` / `sq_free_idx`.

Real BP update for both retired branches. A second update port on the BP RAM is straightforward (two write ports, same-cycle, with a tiebreaker rule for the same index). The RAS already supports the dual-issue case in fetch; the same idea on the update side would close the loop.

Better recovery for in-flight loads on flush. Right now a flush kills speculative MSHRs that are still waiting on memory. Keeping the line in the cache once it returns, even if the original LSQ entry is gone, would absorb this cost.

A better store policy. Store hits write the L1 line and drive `BUS_STORE` the same cycle, which is fast but takes the bus from fetch. A small store buffer between the cache and the bus would let the bus arbitration prefer fetch when there is no urgent miss to fire.

Push past 8.5 ns. The synthesis binding path is in the input combinational group. Adding one register stage at the fetch input boundary would likely move the critical path to the register-to-register group, which currently has 280 ps of margin at 8.5 ns—enough headroom for a further clock-period reduction to roughly 8.2 ns ($f_{\max} \approx 122$ MHz).

11 Conclusion

We designed, implemented, and verified a two-way superscalar, out-of-order RISC-V processor in synthesizable SystemVerilog. Starting from the in-order P3 pipeline, we replaced every stage with a P6-style out-of-order equivalent: a register-renaming map table, an 8-entry reservation station with oldest-first issue, a 16-entry circular ROB, a split load/store queue with byte-granularity forwarding, and a non-blocking 2-way set-associative D-cache backed by a victim cache and a 4-entry MSHR. On the prediction side we built a dual-issue Gshare predictor with a BTB-alias guard and a 16-deep return address stack.

All 34 benchmark programs pass the full-pipeline testbench with a clean WFI halt and a correct architectural register state. The aggregate CPI across the suite is 2.63, with the best-case performance of 1.19 on `true_loop`, approaching the theoretical lower bound of 1.0 imposed by the single-issue execute stage. Synthesis against the ASAP7 standard-cell library closes at 8.5 ns ($f_{\max} \approx 117.6$ MHz), with the binding path in the input combinational group. The two

largest single improvements in simulation were the BTB-alias guard (-30% total cycles) and the retirement phantom-flush fix, which was the enabling change that allowed any program to run to completion.

The project reinforced several lessons that are easier to state than to internalize before encountering them in a real design: (1) a small combinational guard can have a disproportionate effect on performance when it prevents speculative misfires deep in the pipeline; (2) unit-level testbenches for the RS and LSQ paid back their development cost many times over by catching forwarding bugs that were nearly invisible in full-pipeline traces; and (3) the gap between "the design looks right in simulation" and "the design is actually correct for all corner cases" is bridged almost entirely by systematic testing across a diverse program suite.

Future work would focus on adding a second ALU and CDB to lift the single-issue bottleneck, widening the LSQ dispatch port, and pushing the clock period below 8.5 ns by pipelining the front-end input logic.