ECE 411: mp_ooo Final Report

By: Pratyush Ashok Anand, Stanley Auyeung, and Siddh Shah

“Branch Mispredictors”

12/15/2025

Mentor: Jason Yan

1: Introduction

Performance improvements have become critically important in modern computer architecture, as engineers and researchers continue pushing the boundaries of processor capability without compromising cost or yield. The Out-of-Order (OoO) processor has emerged as a central solution, improving CPU performance by allowing instructions to execute as soon as their operands are ready rather than strictly adhering to program order. This approach offers significant advantages over a conventional in-order design: lower latency, higher throughput, and more efficient handling of dependencies.

This report details the development of our OoO processor over the last eight weeks—a RISC-V Out-of-Order core implementing the RV32IM instruction set architecture. Its out-of-order nature is crucial, as it enables the exploitation of Instruction Level Parallelism (ILP) by executing instructions as soon as resources and operands are available. As a result, multiple instructions can run in parallel. Our processor relies on dynamic scheduling to achieve this, effectively hiding stalls and increasing parallelism by reducing delays that would otherwise arise from data dependencies.

The report is organized into the following sections: Project Overview, Design Review (Checkpoint 1, Checkpoint 2, Checkpoint 3, Advanced Features Review), Final Thoughts, and Conclusion.

2: Project Overview

The goal of this project was to design and implement an out-of-order processor with strong performance—as measured by a high IPC (Instructions Per Cycle)—while also achieving low power consumption and small area. The latter two metrics are especially important because benchmark performance in this course is evaluated using the equation:

$Performance = P * D^4$

*where P represents the total power (static + dynamic) and D represents the Delay

Early in the project, we decided to use an Explicit Register Renaming (ERR) scheme for our processor. We chose ERR because it provides clear physical register allocation and precise recovery on mispredictions or exceptions, making the design more modular and easier to debug. It also avoids false dependencies (WAR/WAW), enabling the high degree of ILP necessary for competitive OoO performance.

With this decision made, we created a block diagram of our overall processor that fully supports the RV32IM ISA and includes all major OoO components: fetch, decode, rename, dispatch, issue, execute, writeback, and commit.

Throughout the checkpoints, we made a conscious effort to distribute the work evenly. This allowed each team member to focus on a specific subset of modules while enabling effective unit testing of each subsystem before integration. Despite this, we inevitably faced challenges during integration—ranging from mismatches and unexpected stalls to deadlocks—which required collaborative debugging sessions. These experiences also strengthened our Git workflow skills. We frequently resolved merge conflicts using separate branches, Visual Studio Code's merge tools, and by isolating advanced features into dedicated branches before merging them into the main graded version.

In the end, our repository contained two primary branches: one implementing a fully functional processor that passed all required tests, and another containing additional advanced features built for the competition. Overall, we successfully delivered a working OoO core and gained deep insight into how modern processors exploit parallelism, handle dependencies, and maintain correctness in the presence of speculation.

3: Design Description

3.1: Design Overview

As mentioned before, we decided to use the ERR scheme because it makes it easier for us to eliminate false dependencies and enable high instruction-level parallelism for OoO processors. Notably in ERR, each architectural register defined by the ISA is dynamically mapped to a physical register during renaming. Instead of using this physical register directly, the processor substitutes all future reference registers with the corresponding physical register. This mapping is implemented by a structure known as the Renaming Alias Table (RAT), which records the most recent physical register associated with each architectural register.

By allocating a new physical register for every write, this scheme completely eliminates WAW (write-after-write) and WAR (write-after-read) hazards, since instructions no longer interfere through shared architectural names. Only true RAW (read-after-write) dependencies remain, which the scheduler handles naturally. This precise tracking of physical registers also ensures clean rollback on mispredictions and exceptions.

As shown in Figure 1, architectural-register dependencies (left) contain unnecessary constraints because multiple instructions write to the same architectural names (a1, a2, a4), introducing false hazards that serialize execution. With our ERR and OoO schemes (right), each write receives a unique physical register (p1, p2, p3, p4), removing these false dependencies and exposing significantly more parallelism. Instructions like i1 and i3 can now execute independently, accelerating the overall pipeline.

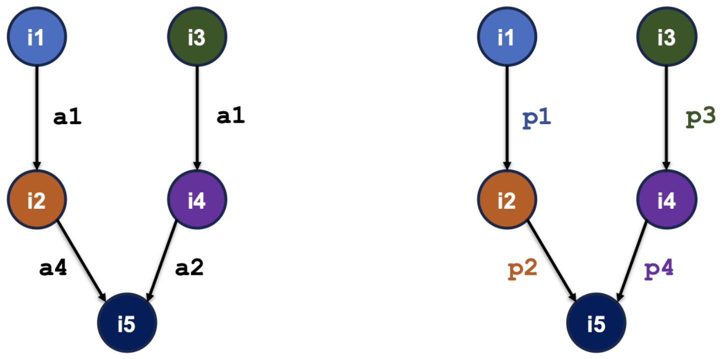


Figure 1: Image detailing architectural and physical register dependencies

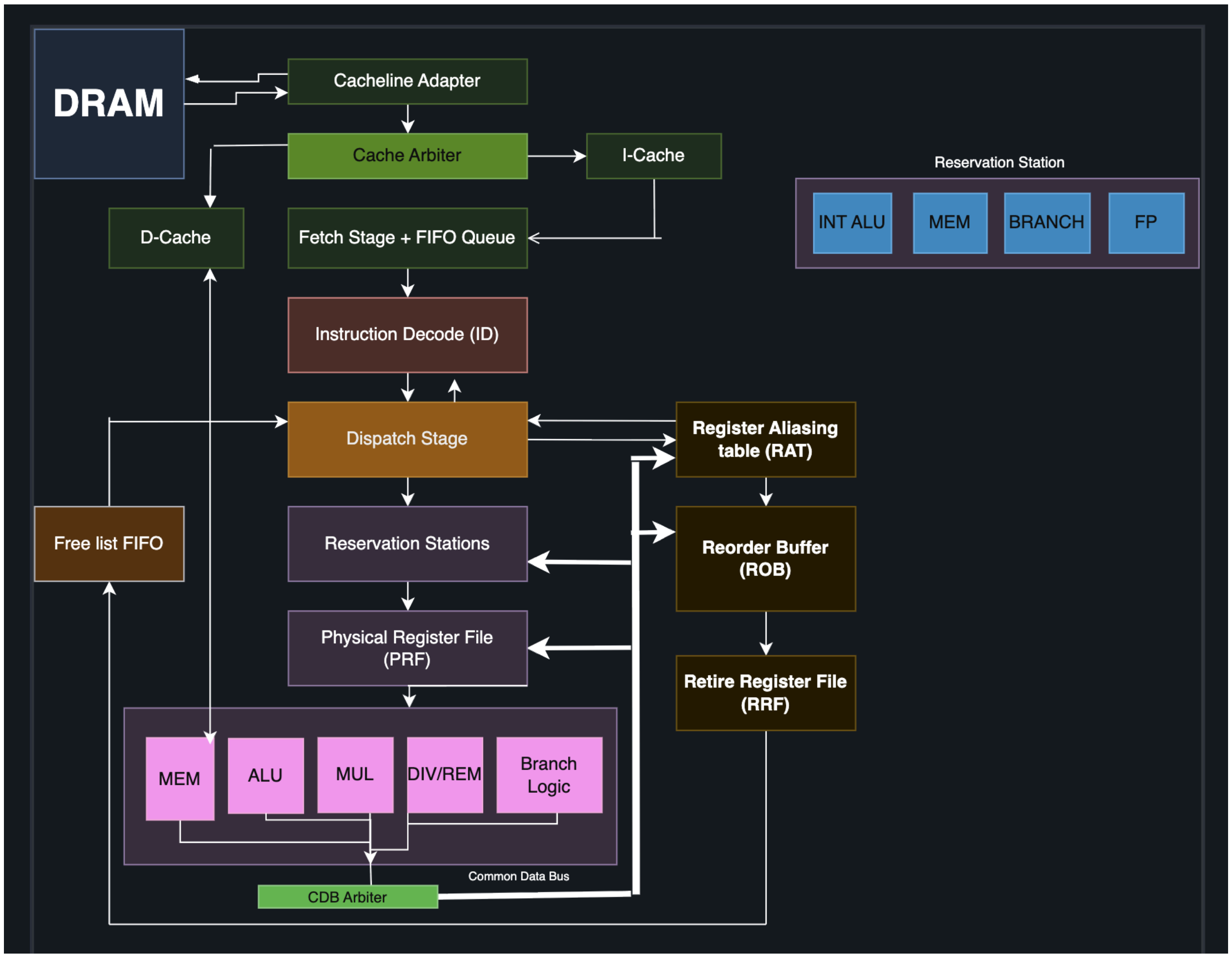


Figure 2: High level block diagram of our OoO Processor architecture

Figure 2 represents the high level architecture of the processor, and this section will explain the basic dataflow and architecture of our processor. The first stage of the processor is the FETCH stage. Instructions that come in from our DRAM memory first go through the cacheline adapter, and are subsequently sent to the I-cache using the cache arbiter. The fetch stage unit subsequently grabs these instructions from the I-cache, eventually pushing them (enqueueing) them into FIFO queue one-by-one.

The **Instruction Decode (ID)** stage reads instructions from the fetch queue, decodes the opcode and operand fields, and identifies the instruction type (ALU, memory, branch, FP).It also generates control signals and preliminary dependency information (source/destination architectural registers) that will be used during renaming and dispatch. The various variables used in this classification are as follows: instruction’s opcode, destination register, source registers, immediate values, funct3, and funct7.

During the rename stage, architectural source and destination registers are translated into physical registers via the Register Alias Table (RAT). Source operands are read from the RAT's current mappings, while a new physical register is allocated for the destination using the Free List FIFO, which tracks all unoccupied physical registers. The Physical Register File (PRF) stores the data values for in-flight registers, while the Retire Register File (RRF) stores the committed architectural-to-physical mappings. Unlike the RAT—which contains speculative updates—the RRF only reflects architectural state. As instructions eventually commit, the RRF updates its entries with the newly committed physical register and returns the old physical register to the free list. It is crucial that the RRF and Freelist FIFO interact with each other constantly, as the irregular dequeuing/enqueuing of physical registers may lead to unnecessary stalls or the wrong RS1/RS2 values being read by the program.

After renaming, instructions are inserted into the Reorder Buffer (ROB), which operates as a circular queue holding all in-flight instructions in program order. Each ROB entry records the instruction's destination register, completion status, and whether an exception occurred. The dispatch stage then forwards instructions to the correct Reservation Station (RS) bank—INT ALU, MEM, BRANCH, or FP—where they wait until their operands become ready. The ROB enables precise exceptions and in-order retirement, while the RS enables out-of-order execution by letting ready instructions issued independently of older stalled ones.

When the reservation stations and functional units are finally ready, the instruction is issued to the corresponding functional unit (ALU, MEM, MUL, DIV/REM, Branch Logic). When a functional unit finishes, the result is broadcast on the Common Data Bus (CDB)—managed by the CDB Arbiter—updating the PRF, waking dependent RS entries, and marking the corresponding ROB entry as complete. Once the instruction reaches the head of the ROB and is marked complete, the ROB commits it: the RRF updates its committed mapping, and the prior physical register is returned to the Free List FIFO, closing the rename-to-commit pipeline loop.

3.2: Baseline Milestones

Checkpoint 1:

In this checkpoint, we implemented our processor's instruction fetch stage, which utilized a CPU-to-cache burst memory model, requiring a cacheline adapter to convert four 64-bit bursts into a 256-bit dataline for our 32-bit ISA. To prevent a one-cycle delay that limits IPC to 0.5, the entire cacheline is loaded into a linebuffer on fetch, enabling consecutive instructions from the same line to be quickly retrieved from the buffer. Once fetched, the instruction is pushed into a FIFO-based instruction queue, where it awaits decoding.

Checkpoint 2:

In this checkpoint, we extended our processor to complete issuance, execution, and commitment for all ALU-based and multiply/divide instructions. This included the dispatch/rename stage, RAT, RRAT, ROB, CDB and its arbiter, all functional units including reservation stations, and the freelist FIFO. The multiplier and divider/remainder modules were implemented using the Synopsys IPs *DW_mult_pipe* and *DW_div_pipe*, respectively, which are pipelined with parameterizable stages, implemented as wrappers with the IPs instantiated inside. All edge cases like division by zero, negative operands, etc., were handled.

Checkpoint 3:

In checkpoint 3, we extended our core to support the full RV32I instruction set by implementing all memory operations (loads and stores) and all control-flow instructions.

We added a dedicated memory reservation to store loads and stores until they were execution ready, in addition to a load-store queue (LSQ), where memory operations are queued separate from other instructions to increase efficiency, and interfacing with a separate data cache.

Upon detecting a branch misprediction (assuming a static not-taken mechanism), the processor initiates a global pipeline redirect, flushing all modules upstream of the ROB, including the fetch stage, i-queue, all reservation stations, and all instructions in the ROB younger than the branch instructions, while restoring the RAT to the architectural state stored in the RRAT.

3.3: Advanced Features

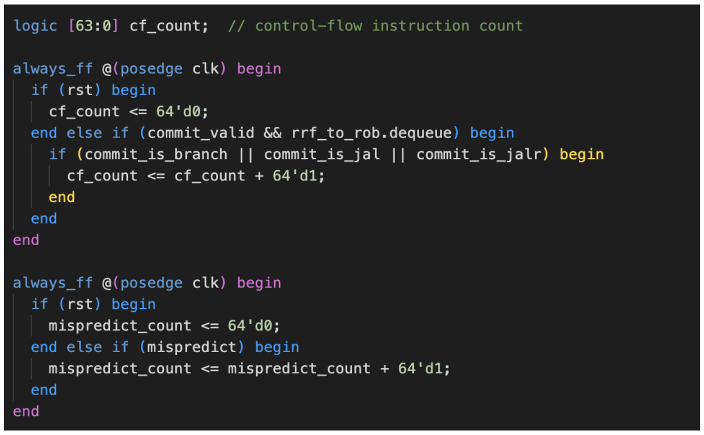
3.3.1: Gshare Predictor w BTB

Our very first advanced feature that we looked to implement following checkpoint 3 was an effective branch predictor, as it was clear the branch not-taken was highly ineffective especially given our high mispredict cost. We implemented a GShare branch predictor combined with a small BTB to reduce control-flow stalls in our OoO processor. GShare uses an 8-bit Global History Register (GHR) XORed with the PC to index a 256-entry table of 2-bit saturating counters, while the BTB (32 entries) stores predicted targets to redirect and fetch immediately on predicted-taken branches. In the main design, the BHT and BTB use simple flip-flop arrays for simplicity; in our advanced-features branch, these were replaced with a compact dual-ported SRAM to lower area and reduce flip-flop overhead.

Prediction and BTB lookup occur in the fetch stage, and the predictor is trained strictly at commit via the ROB (bpred_update_valid), ensuring that speculative branches never corrupt the GHR or BHT. This commit-based update strategy provides correct global history recovery after flushes, and this was the only to do so accurately, given that we were not pursuing early branch recovery.

We measured these benchmark statistics directly in hardware by counting all committed control-flow instructions (branches, JAL, and JALR) and incrementing a separate counter on each detected misprediction (**see Figures 3 and 4**). These counters were latched at the end of program execution, allowing us to compute predictor accuracy and correlate it with IPC. Using mispredicts relative to all control-flow instructions (cf_count), our measured control-flow prediction accuracies were 92.7% for Coremark (4527 mispredicts out of 60,470 control-flow instructions), 78.9% for mergesort (20,702 / 98,078), 77.7% for FFT (7433 / 33,271), 92.3% for compression (5017 / 65,570), and 51.8% for aes_sha (12,286 / 25,505). Despite the predictor's improvements, our overall IPC gains remained modest because the front end was still constrained by the CP3 I-cache and line-buffer implementation. Even so, the GShare+BTB integration produced consistent performance uplift across all benchmarks: coremark_im improved from 0.1127 to roughly 0.139, mergesort from 0.1211 to about 0.143, compression from 0.1544 to around 0.176, aes_sha from 0.1489 to roughly 0.170, and fft from 0.1490 to approximately 0.171, representing a typical increase of 0.02–0.03 IPC. The net improvement was initially modest, given that we had recurring issues with our cache and linebuffer implementations following the end of CP3. The results matched the high percentage of correctly predicted branches towards the end of our implementations following the addition of more advanced features in addition to fixing our IF stage.

In the very end of our project (after implementing complementary advanced features), we also observed in **Figure 5** that the number of clock cycles spent on mispredict recovery versus correct-prediction execution differs dramatically: a mispredicted branch incurs roughly a 6-cycle penalty, while a correctly predicted branch completes with only 1 cycle of overhead, highlighting the practical significance of reducing mispredicts even when IPC gains appear modest.



```
logic [63:0] cf_count;  // control-flow instruction count

always_ff @(posedge clk) begin
  if (rst) begin
    cf_count <= 64'd0;
  end else if (commit_valid && rrf_to_rob.dequeue) begin
    if (commit_is_branch || commit_is_jal || commit_is_jalr) begin
      cf_count <= cf_count + 64'd1;
    end
  end
end

always_ff @(posedge clk) begin
  if (rst) begin
    mispredict_count <= 64'd0;
  end else if (mispredict) begin
    mispredict_count <= mispredict_count + 64'd1;
  end
end
```

Figure 3: Image showing code for counters (we only measured branches/control flow instructions) at commit

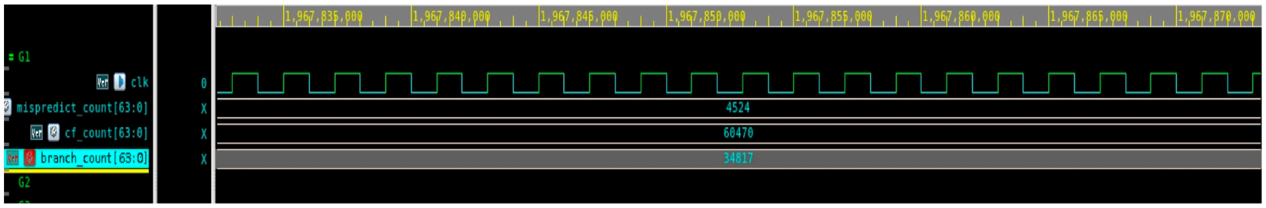


Figure 4: Image showing the various values of these counters at the end of the coremark program.

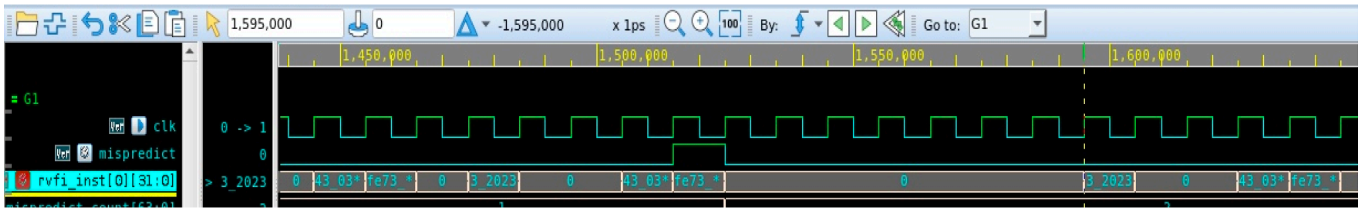
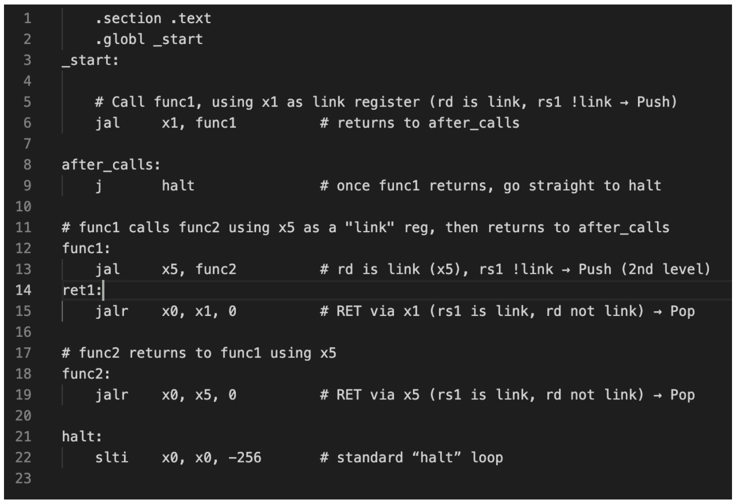


Figure 5: Difference in clock cycles flushed for the same branch instruction that was mispredicted (6 cycles) vs predicted correctly (1 cycle)

3.3.2: Return Address Stack

To improve prediction accuracy for subroutine returns, we implemented an 8-entry Return Address Stack (RAS) in the fetch stage. The RAS tracks nested call/return behavior by pushing the return address on JAL instructions that write to a link register (x1 or x5 in our design) and popping on JALR-based returns. This allows the processor to predict the correct return target immediately, avoiding the expensive recovery penalty normally incurred when treating JALR instructions as indirect branches.

Because a JALR mispredict typically costs ~6 cycles compared to ~1 cycle for a correct prediction, even small reductions in return mispredictions yield meaningful performance improvements. In our design, enabling the RAS (via the ENABLE_RAS parameter) resulted in a substantial IPC increase—from 0.14287 to ~0.20 on recursive and nested-call workloads such as the test program shown below (see Figures 6 and 7). This code exercises multiple levels of call/return nesting, and without a RAS the predictor misclassifies nearly all JALRs as indirect jumps. With the RAS, however, both returns in func1 and func2 are predicted perfectly, essentially eliminating JALR mispredicts at very little power and area cost (only about 3k).



```
    .section .text
    .globl _start
_start:

    # Call func1, using x1 as link register (rd is link, rs1 !link → Push)
    jal     x1, func1          # returns to after_calls

after_calls:
    j       halt               # once func1 returns, go straight to halt

# func1 calls func2 using x5 as a "link" reg, then returns to after_calls
func1:
    jal     x5, func2          # rd is link (x5), rs1 !link → Push (2nd level)
ret1:
    jalr    x0, x1, 0          # RET via x1 (rs1 is link, rd not link) → Pop

# func2 returns to func1 using x5
func2:
    jalr    x0, x5, 0          # RET via x5 (rs1 is link, rd not link) → Pop

halt:
    slti    x0, x0, -256       # standard “halt” loop

```

Figure 6: Code used to test full extent of the RAS functionality, exercises all the calls

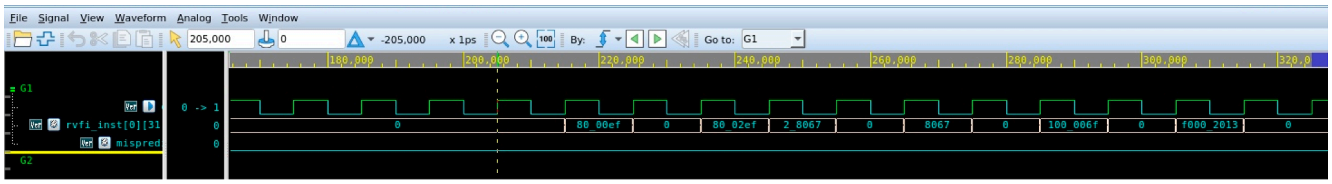


Figure 7: Waveform for the current program with RAS functionality enabled

In terms of our benchmarks, we had a small but non-trivial increase to branch accuracy with RAS enabled, eg: coremark branch prediction accuracy went up from 92.4% to 92.7%.

3.3.3: Streambuffer Prefetcher

Another clear bottleneck was front-end latency for instruction cache misses. Our core spent many cycles stalled in fetch waiting for instruction lines to return from memory, especially in loop-heavy code. To reduce these stalls, we implemented a small instruction streambuffer in front of the i-cache.

Architecturally, the fetch stage communicates to the streambuffer with a simple valid/ready handshake. Internally, the buffer forwards demand misses to the i-cache, but also speculatively streams ahead by prefetching the multiple data blocks ahead. When the fetch stage issues a request that hits in the buffer, the buffer returns the word in a single cycle without needing the i-cache. On a miss, it forwards the request to the cache and simultaneously schedules a prefetch for the next datalines.

The buffer is demand-priority, meaning it never issues a prefetch if there is an outstanding demand fetch in-flight, and it only processes one memory request at a time. This ensures that the speculative prefetches do not delay real instruction misses. Furthermore, since the streambuffer is only a front-end interface (only interacts with the i-cache's UFP ports), data cache traffic is never blocked by prefetches. On a flush or redirect, the streambuffer is invalidated, ensuring that no prefetched lines from the wrong path are delivered to the i-fetch stage.

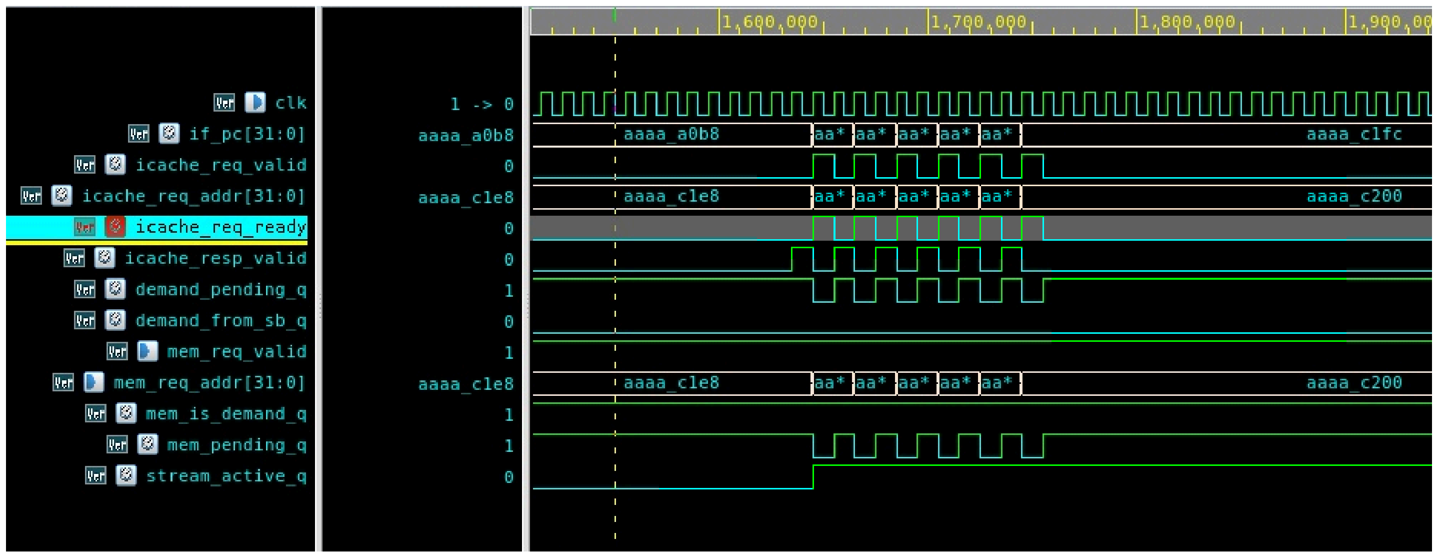


Figure 8: Streambuffer Cold Miss

Figure 8 shows the behavior of the instruction path on a cold miss with the streambuffer enabled. When the instruction request is issued (icache_req_valid goes high) for address 0xAAAAC1E8, the streambuffer demand is low (as the buffer is initially empty). The stream buffer then forwards this demand to memory (mem_req_valid asserts with mem_is_demand_q = 1), stalling the fetch until the response is returned (mem_resp_valid high).

Then, the instruction is returned to the fetch stage (icache_resp_valid), and simultaneously, a new instruction stream is initialized (stream_active_q goes high, stream_next_q advances to next address), showing that the cold miss both services the current fetch instruction and initializes future instruction fetching.

After the miss, fetches to 0xAAAAC200 and 0xAAAAC220 don't require separate memory accesses, as they'll now hit in the streambuffer, letting icache_resp_valid assert regardless of mem_resp_valid. As seen in Figure 8, the memory requests are overlapped with instruction execution. This corresponded to the following IPC changes:

Benchmark	Original IPC	IPC w/ SB Prefetcher	IPC Change
aes_sha	0.164045	0.173856	+5.98%
compression	0.212088	0.212090	~0%
fft	0.204812	0.210996	+3.00%
mergesort	0.154013	0.168790	+9.59%
coremark_im	0.170353	0.176721	+3.74%

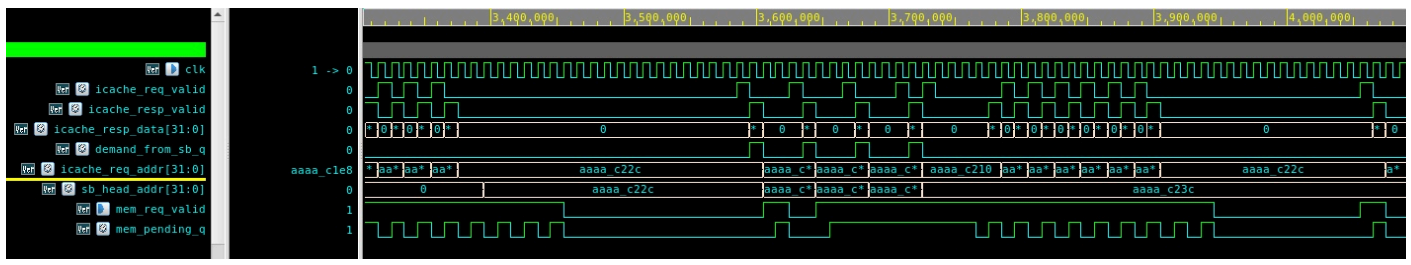


Figure 9: Streambuffer hit

Figure 9 shows the streambuffer hit, where the fetch request address matches the head entry of the buffer (icache_req_addr = sb_head_addr, demand_from_sb_q high), and the instruction is returned immediately (icache_resp_valid high), showing that the streambuffer eliminates a cache miss by sending the prefetched instruction directly to the fetch stage, reducing latency.

3.3.4: Post-Commit Store Buffer + Write Coalescing

In our CP3 baseline implementation, store instructions created unnecessary backpressure at the commit point because the core had to preserve in-order memory semantics while also interacting with the data cache. To reduce this bottleneck, we added a post-commit store buffer (PCSB) that separates ROB retirement from the data cache write path, so instead of forcing the commit stage to wait for a store to fully write to the cache, committed stores enqueue into the PCSB, which drains stores in-order to the data cache when the memory system is ready. In our design, stores still execute through the memory pipeline and enter the LSQ for ordering, but the actual cache write is handled after commit by the PCSB. The PCSB drains stores to the cache in the background when the downstream interface is ready, and it supports correctness by forwarding full-word hits to younger loads that are fully covered by buffered committed stores. We also added support for write coalescing, which if a newly committed store targets an address already buffered, we merge its byte mask/data into the existing entry, reducing PCSB pressure and lowers write traffic to the cache and memory system. Workloads that a PCSB with write coalescing can help with is when stores are frequent and/or repeatedly touch the same locations, but it helps less when stores are streamed to mostly unique addresses or in workloads where loads need a very recent store that hasn't committed yet.

A tradeoff from implementing a PCSB with write coalescing was an increase in measured area. From the latest commit prior to implementing the PCSB, the area of our pipeline was 251480.38704, but after adding the PCSB, and then write coalescing logic, the area increased to 259361.699745, and 260063.141758 respectively. This showed a 3.13% increase in area when adding the PCSB, and then an additional 0.27% increase when adding in logic for write coalescing. Although there were tradeoffs present, our core also experienced some performance improvements. From the baseline to PCSB only version, we saw a 2.24% average speedup across all 5 required benchmarks when considering delay, and a 2.32% average IPC increase across all 5 required benchmarks. From the PCSB only version to the PCSB with write coalescing support version, IPC and delay remained the same for all tests except for aes_sha, which experienced a 0.054% increase for IPC, and a 0.054% speedup. Since adding write coalescing added negligible performance across these benchmarks, it could suggest that there are limited coalescable store patterns for these benchmarks.

	Baseline	PCSB	PCSB + Write Coalescing
IPC	aes_sha: 0.150358 compression: 0.159990 fft: 0.154231 mergesort: 0.138062 coremark_im: 0.139248	aes_sha: 0.156000 compression: 0.159990 fft: 0.159695 mergesort: 0.143041 coremark_im: 0.140153	aes_sha: 0.156084 compression: 0.159990 fft: 0.159695 mergesort: 0.143041 coremark_im: 0.140153
Delay (Monitor Segment Time)	aes_sha: 51.60589ms compression: 26.7747ms fft: 75.2775ms mergesort: 34.90082ms coremark_im: 20.93837ms	aes_sha: 49.73925ms compression: 26.87746ms fft: 72.70211ms mergesort: 33.68591ms coremark_im: 20.8032ms	aes_sha: 49.71261ms compression: 26.87746ms fft: 72.70211ms mergesort: 33.68591ms coremark_im: 20.8032ms
Area	251480.385704	259361.699745	260063.141758

Figure 10: Performance Analysis

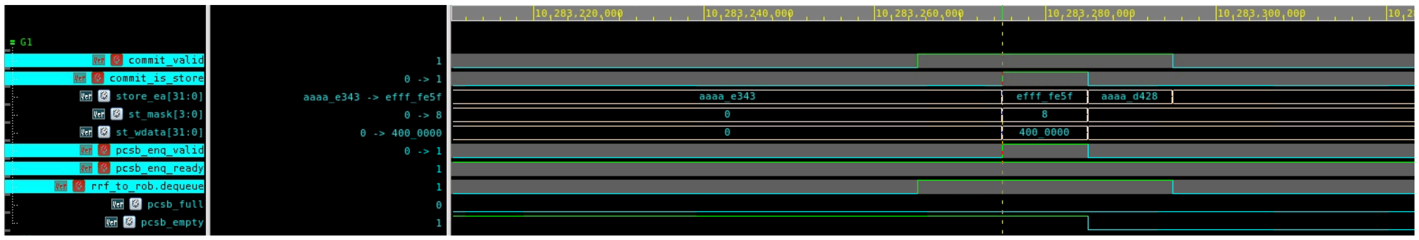


Figure 11: Store retirement enqueues into the PCSB (commit decoupling)

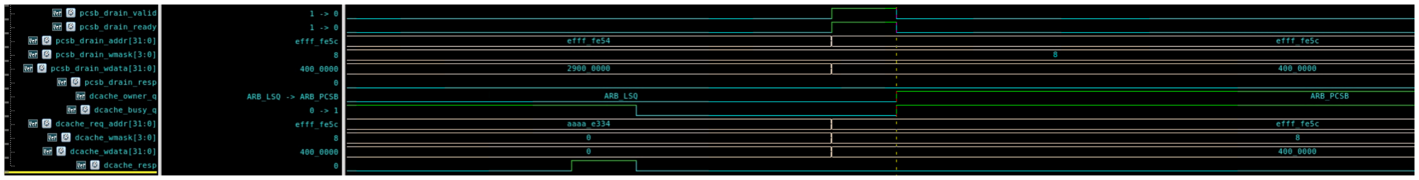


Figure 12: PCSB drain arbitration drives the D-cache write port

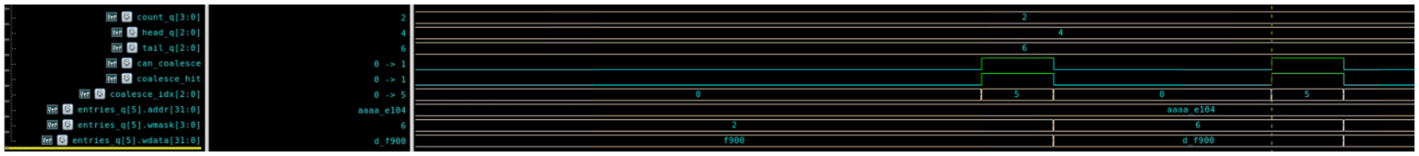


Figure 12: Write coalescing merges a new store into an existing PCSB entry

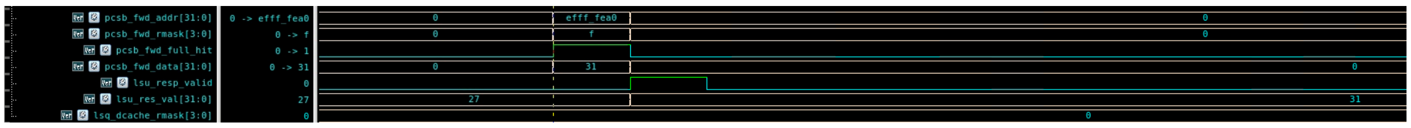


Figure 13: PCSB full-hit forwarding satisfies a load without issuing a D-cache read

3.3.5: Parameterized Sets & Ways Cache (including PLRU)

To address the severe latency and conflict-miss issues observed in our provided direct-mapped, write-through cache, we replaced it with a set-associative cache with parameterizable sets and ways and a pseudo-LRU replacement policy. In the direct-mapped cache, multiple frequently accessed addresses mapped to the same cache index, which we saw caused repeated evictions and redundancy even with no capacity misses. By implementing a simple set-associative cache (using an IDLE → LD → CHECK → WR FSM), we allowed multiple lines to map to the same set and reduced conflict misses. The PLRU lets the cache hold recently accessed data with relatively low overhead. Through this implementation, we saw an approximate double to our IPC (~0.17-0.22 → ~0.42-0.48), with minimal loss when reducing the ways parameter.

Benchmark	Original IPC (with Streambuffer Prefetching and SRAM-based Gshare)	IPC w/ 4x8 SA Cache	IPC Change
aes_sha	X	X	X
compression	0.210565	0.483082	+129.42%
fft	0.215669	0.446226	+106.90%
mergesort	0.168252	0.417852	+148.34%
coremark_im	0.170031	0.423871	+149.29%

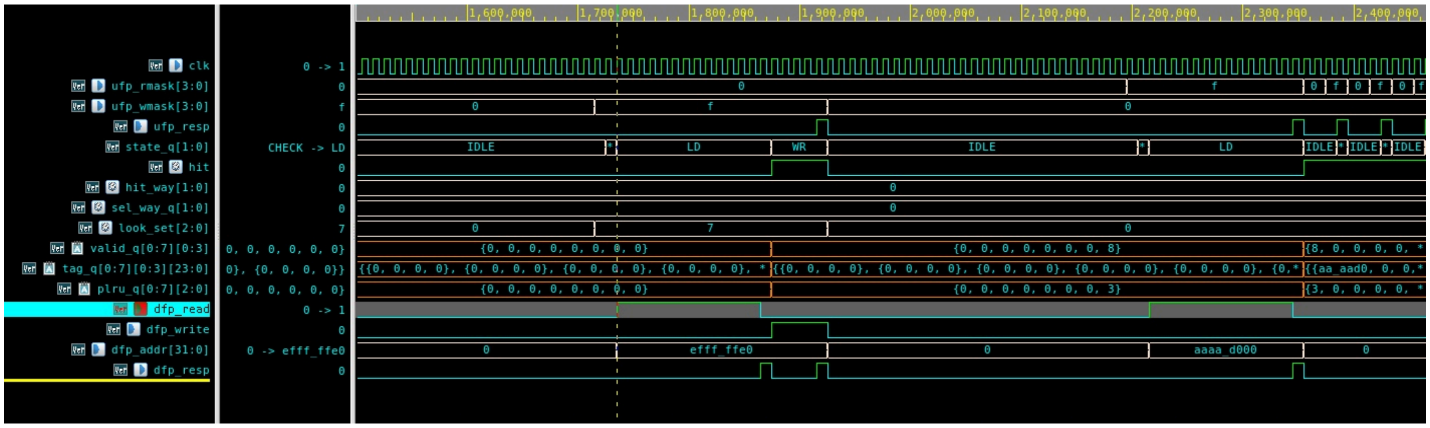


Figure 14: Access A in set, access B in set with different tag, re-access A in set

This waveform shows the key differences between our parameterizable SA cache versus the original DM cache. We start with a cold miss (state transition from IDLE to CHECK), where it detects a miss (hit = 0), entering state LD, fetching a cacheline from memory (dfp_read high). Upon the returned response (dfp_resp high), the line is loaded into a selected way (sel_way_q), the corresponding valid bit in valid_q is asserted, and our PLRU state (plru_q) is updated.

The waveform also shows that when we have a subsequent access that maps to the same set but with a different tag, it does not evict the original line. Our pick_victim function selects a different way, letting both cache lines reside simultaneously. The waveform shows this as well through the multiple valid ways within the same set and changing PLRU bits. In contrast, the DM cache would evict the line immediately, causing repeated misses on alternating accesses.

One major tradeoff that we had to analyze was area and IPC gain with respect to cache parameters. We saw that with 4-way, 64-set instruction and data caches, we passed 4 out of 5 benchmarks, while being within 10% of the IPC baseline for the fifth. However, this ballooned our area to the point where area reductions in other parts of our processor did not bring us below the maximum (300k) area threshold. In an attempt to reduce the area by reducing the number of sets from 64 to 8, we saw that our IPC marginally fell, unfortunately just under most of the benchmark baselines (passed or gained partial credit for 4/5, failed one [AES_SHA], likely due to the direct-mapped cache matching that benchmark's memory access patterns better than in the set-associative case). Although more ways allowed our processor to perform better, the increase in area was too excessive, and thus we made the decision to take on the relatively lower penalties of our benchmark results rather than the penalties of the increased area (including potential further reductions in IPC from other parts of the processor we would have made area reduction tweaks to). Regardless, this new cache performed far better than our original cache model, allowing our IPC to nearly meet the baseline as compared with the DM cache.

3.3.5: Age-Ordered Issue Scheduling

We implemented age-ordered issue scheduling back in CP1 as part of making our scheduler behave more predictably when lots of instructions are waiting to run. When an instruction enters a reservation station, we store an age field from age_ctr_q in the RS entry. When it's time to issue, the RS uses pick_ready_oldest to choose the ready entry with the smallest age (meaning it's the oldest). For the memory RS, we also set FORCE_IN_ORDER=1, which blocks issuing a younger memory op if there is any older busy entry still in the RS (has_older_busy). On top of that, the CPU-level issue select gives extra priority to memory ops at the ROB head (mem_head_prio = can_mem && (mem_rob_idx_o == rob_head_idx)), so "next-to-commit" memory ops don't get stuck behind other units. Since this feature was added in CP1 and became part of our normal baseline scheduler, we didn't keep a separate "before age-ordering" version to run side-by-side during the advanced features benchmarking, so we don't have a record for performance benefit/tradeoff just for this change.

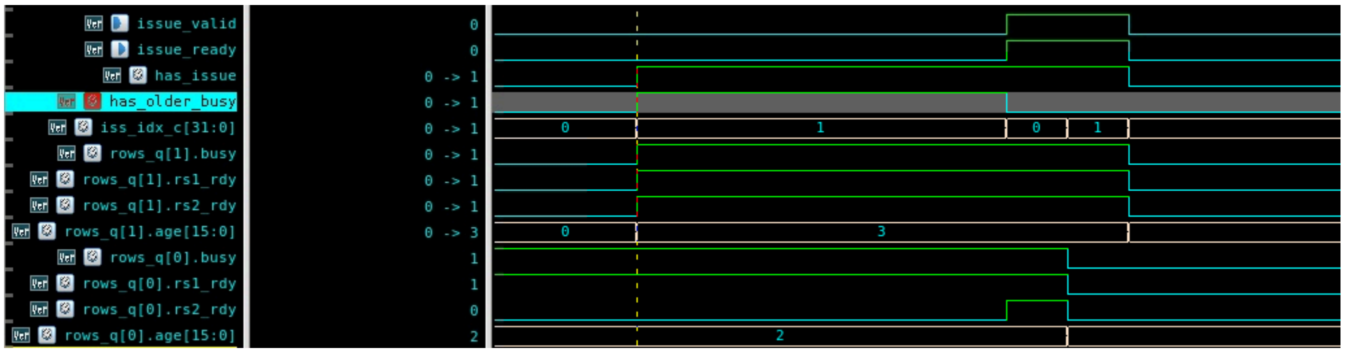


Figure 15: RS age-order gating blocks younger issue until older busy entry clears

In the figure above, the memory RS shows that even when a younger entry is selected and ready (has_issue = 1, iss_idx_c = 1), issue_valid stays low while has_older_busy = 1 because an older busy entry is still present in the RS. Once has_older_busy drops, issue_valid can go high and the entry issues.

4: Future Improvements and Final Thoughts

Although we learnt a lot through this 8-week project, our processor was far from perfect–oftentimes we found ourselves thinking deeply about potential avenues for improvement. Some of these were as follows:

1) Implementing a separate branch instruction unit and reservation station rather than a joint one. While we had planned to do this at the very start of the project, we lost direction and implemented a combined ALU/branch unit. This hurt performance because branch instructions had to compete with ALU ops for issue bandwidth, delaying branch resolution and possibly causing longer mispredict recovery times in our initial implementation.
2) Implementing a more robust cache arbiter, as we believe the I-cache arbiter suffered from starvation early in the advanced-features checkpoint. When the arbiter fails to prioritize instruction fetches correctly, the frontend becomes supply-starved, reducing the rate at which instructions enter the OoO core and sharply lowering IPC.
3) Our reliance on the provided cache was also a major performance bottleneck, as its latency and miss behavior were significantly worse than the optimized *mp_cache* (which none of our group members managed to get fully working unfortunately). Although we eventually implemented a parametrized set/way cache that drastically improved performance, identifying this bottleneck earlier would have given us more time to tune sets and ways without exceeding area or power limits.
4) If we had more time, we would have explored a split load and store queue, since using a single unified LSQ forces all memory operations through the same structure. A split design allows loads to bypass stores when safe, increasing memory-level parallelism and reducing stalls on long-latency operations.

5: Conclusion

In this project, we designed and implemented a fully functional out-of-order RISC-V processor supporting the RV32IM ISA. The process required us to navigate a wide range of architectural challenges—from managing rename and scheduling logic to ensuring precise flush behavior, resolving subtle data hazards, and debugging pipeline deadlocks. These experiences underscored the importance of disciplined modular design, incremental verification, and a deep understanding of microarchitectural interactions.

Implementing advanced features such as a parametrizable multi-way cache, a GShare branch predictor with BTB and RAS support, age-ordered scheduling, and a post-commit store buffer with coalescing pushed us to think critically about the balance between performance, power, and complexity. Each enhancement revealed new bottlenecks and forced us to carefully evaluate trade-offs that real CPU designers face. The measurable improvements in IPC, reduction in mispredictions, and smoother memory behavior demonstrated how even modest microarchitectural refinements can meaningfully shift overall system performance.

Ultimately, this project offered us hands-on experience in constructing a realistic out-of-order processor and strengthened our ability to diagnose, reason about, and optimize complex hardware systems. It deepened our appreciation for how modern processors exploit parallelism and speculation while managing correctness, and it gave us the confidence and intuition needed for larger-scale hardware design work in the future (be it in industries or future coursework)!