RISC-V Out-of-Order Processor Design and Analysis

mp_ooo

Aditya Pola (pola3)
Sehwa Jung (sehwaj2)
Rushil Vora (rvora)

ECE 411: Computer Organization & Design
Spring 2026

Introduction

This project focused on designing and implementing an out-of-order RV32IM processor. The processor was developed across multiple checkpoints, beginning with the frontend fetch path and instruction cache, then expanding into out-of-order execution, register renaming, memory support, and full benchmark execution. After completing the baseline processor, we added advanced features such as branch prediction, a branch target buffer, cache improvements, and memory-system optimizations to improve IPC and reduce stalls.

Project Overview

This section provides an overview of how we developed, validated, and optimized our processor across the course of the project. The design was built incrementally across three checkpoints. In Checkpoint 1, we established the fetch frontend, instruction cache, and parameterizable queue infrastructure. Checkpoint 2 expanded the design into a full OoO execution pipeline capable of correctly running all RV32I and RV32M instructions. Checkpoint 3 integrated the data cache and memory hierarchy, and we verified correctness across all seven benchmark programs using the RVFI/Spike monitoring interface.

Following the baseline checkpoints, we entered the advanced features phase, where we profiled our processor using hardware performance counters and identified bottlenecks in the frontend, memory system, and backend scheduling. We then implemented and individually evaluated each advanced feature, using IPC and microarchitectural counters such as frontend stall rate, branch accuracy, cache miss rate, and LSU backpressure events to quantify the impact of each change.

Design Description

I. Overview

The design implements an out-of-order RV32IM core utilizing an Explicit Register Renaming (ERR) infrastructure. The frontend fetch stage leverages a pipelined instruction cache capable of serving back-to-back requests at a one-cycle latency, pushing data into a decoupling instruction queue. During the decode and rename phase, instructions are assigned a fresh physical register from the free list for each destination, while the register alias table (RAT) is updated to maintain current architectural mappings. Instructions then dispatch into both reservation stations and the reorder buffer (ROB) for tracking. Execution units, including a fully combinational ALU, a custom three-stage pipelined multiplier, and a sequential divider, issue from reservation stations once operands are ready, broadcasting results over a common data bus (CDB) to facilitate wakeup and completion. A unified load-store unit (LSU) and load-store queue (LSQ) manage memory operations end-to-end. Upon reaching the head of the ROB, completed instructions commit by updating the architectural register file and driving signals to the RVFI monitor for verification.

Selecting ERR over Tomasulo's algorithm allowed for centralized speculative state within the ROB and physical register file, simplifying correctness reasoning while providing a baseline for later optimizations like the split LSQ and increased commit width. Throughout the development process, we tuned critical parameters such as ROB capacity, dispatch width, and CDB lane priority. Our memory system consists of split L1 caches interfaced with a shared DRAM model via a cacheline adapter. The frontend features a stream buffer prefetcher that performs lookahead on instruction misses, while the data cache utilizes parametrizable MSHRs to tolerate multiple outstanding misses without stalling hits. In the final optimized design, the LSU supports out-of-order load execution between in-order stores to mitigate backend bottlenecks.

II. Milestones

A. Checkpoint 1

For Checkpoint 1, we designed the Fetch, Cache, and Parameterizable Queue modules and tested them separately as individual modules before combining them together with targeted tests. The fetch module contained the PC generation logic. Each cycle, fetch selected the next PC and sent it to the instruction cache. After the cache returned an instruction, fetch pushed the instruction into the instruction queue. At this checkpoint, the instruction queue was temporarily designed to always accept instructions from the decode stage (which was not built yet). The FIFO queue was used as the instruction queue between fetch and decode. It was parameterized by width and depth, allowing the same module to be reused with different entry sizes and queue capacities. The queue was implemented as a circular buffer with a head pointer, tail pointer, and count register. The head pointer tracks the next entry to dequeue, while the tail pointer tracks where the next enqueue should be written and the count register determines whether the queue is full or empty. The queue supports enqueue and dequeue in the same cycle. If the queue is full but a dequeue also occurs, the queue still allows the enqueue because space is being freed in the same cycle. The cache used in CP1 was a pipelined read-only instruction cache. This mitigated the need for outputting the entire cacheline for back to back hits, and also allowing for pipelined hits across cacheline boundaries. The fetch stage sent a PC to the cache, and after the cache access completed, the cache returned the instruction data. Since this checkpoint focused only on instruction fetch, the cache did not need store support or data-cache behavior yet. The cache was designed to provide instruction data to the frontend while hiding some of the memory access details from fetch. Finally, we had a cacheline adapter that combines 4 bursts of 64 bit DRAM reads into a single 256 bit cacheline.

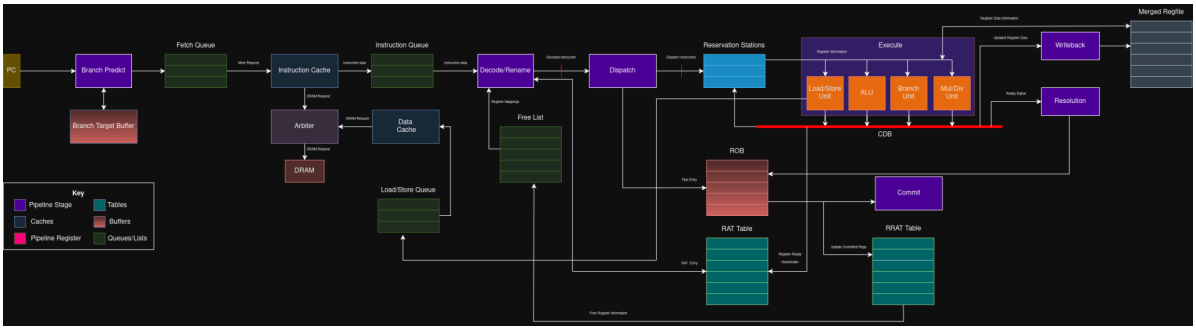

Figure 1: CP1 Block Diagram

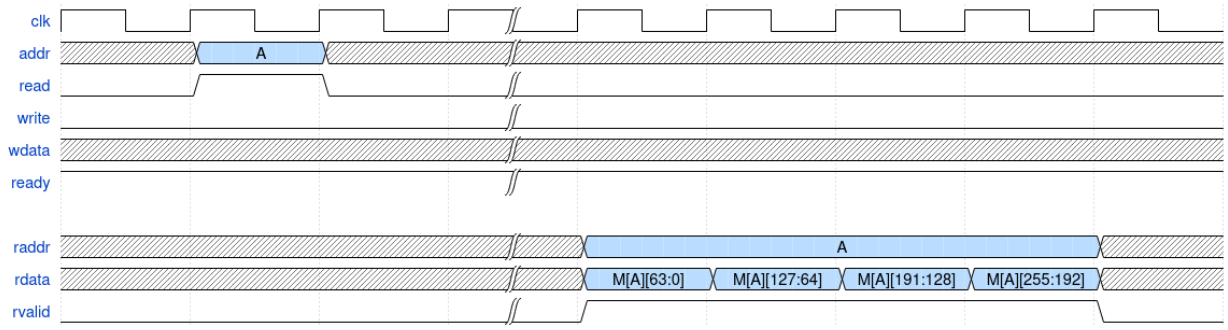

Figure 2: DRAM Burst-Timing for Cacheline Adapter

B. Checkpoint 2

Checkpoint 2 focused on building out the full out-of-order execution pipeline, which we implemented in roughly pipeline order and verified incrementally as each stage came online.

We began with the scheduling logic, implementing the decode and rename stage, physical register file, register alias table (RAT), and reorder buffer (ROB) together as a unit. The decode stage extracts source and destination register specifiers from each incoming instruction. The RAT maps architectural registers to physical registers, and on dispatch a new physical register is allocated from the free list for each instruction's destination. The ROB tracks all in-flight instructions in program order, storing each instruction's physical destination register, completion status, and the metadata needed for commit and RVFI reporting. After integrating these pieces, we ran a small set of instructions through a scratchpad testbench to verify that instructions were being renamed correctly and that the ROB was tracking state as expected before moving further.

With scheduling in place, we implemented the execution units. The ALU handles all standard integer and immediate RV32I instructions and is fully combinational, producing results in a single cycle. For the RV32M extension, we integrated a pipelined Synopsys multiplier IP for all MUL instructions and a sequential Synopsys divider IP for DIV and REM instructions, including correct handling of the divide-by-zero edge case as specified by the RISC-V ISA. Results from all execution units are broadcast over a common data bus (CDB), which is parameterizable in the number of lanes. Lane priority is assigned to the ALU first, then the multiplier, then the divider, reflecting the relative frequency of each instruction type in a typical program. Wakeup logic listens to the CDB and marks

reservation station entries as ready when their source operands are broadcast, and the ROB similarly listens to mark instructions as complete. When an instruction reaches the head of the ROB and is marked complete, it is committed: the result is written back to the architectural register file and the corresponding signals are driven to the RVFI monitor for checking against the Spike reference model. With the monitor hooked up, we wrote a testbench that exercised all RV32IM instructions excluding control flow and memory operations, including a range of register dependencies, and ran the full suite to verify correctness.

One correctness issue required deeper investigation with Verdi. The problem involved the interaction between the instruction cache and the fetch stage's skid buffer. When fetch issues a request to the instruction cache in cycle N and the cache responds in cycle N+1, the fetch stage does not yet know whether the instruction queue is full, since the previous instruction was only enqueued in cycle N. Without handling this, a valid cache response could be dropped under backpressure. To fix this, we introduced a skid register that captures the cache response. If the skid register holds a valid instruction and the queue is still backed up, the skid register drains into the queue first before normal fetch resumes, ensuring no instructions are lost.

A separate issue arose not from correctness but from synthesis. The Synopsys multiplier IP was unwrapping during synthesis in a way that created an extremely long critical path, making it impossible to meet timing. After many hours of trying to get the synthesis tool to cooperate, we replaced it entirely with a custom three-stage pipelined multiplier of our own design. This required some design work to get the pipeline staging right, but the result synthesized cleanly and allowed us to meet a 500 MHz target without any timing violations. It's worth noting that the multiplier remained as our critical path for the remaining development process.

C. Checkpoint 3

Checkpoint 3 focused on extending the processor to fully support the RVI32IM instruction set while integrating the cache hierarchy and memory system into the core. Through the course of this checkpoint, the processor went from being able to execute out of order instructions to executing benchmark programs and data memory interactions. A major aspect of this checkpoint was the integration of the data cache into the processor pipeline. The processor already supported an instruction cache for frontend fetching, so we expanded the memory hierarchy by connecting the load/store path to a data cache. Support for all memory instructions was implemented through a split load/store queue (LSQ) with no load-store forwarding. The LSQ receives memory operations from dispatch and tracks them until they are ready to execute. Each entry stores operand readiness information, physical register mappings, ROB indices, and instruction metadata. Regarding testing, we didn’t use any separate testbenches for our individual modules or test files. Instead, we primarily used the provided benchmark programs and checked the IPC, area, and delay in each of those. In addition, we created a script that

would automatically tell us the number of branches taken, prediction accuracy, and any other helpful information as we tuned different parameters for optimization.

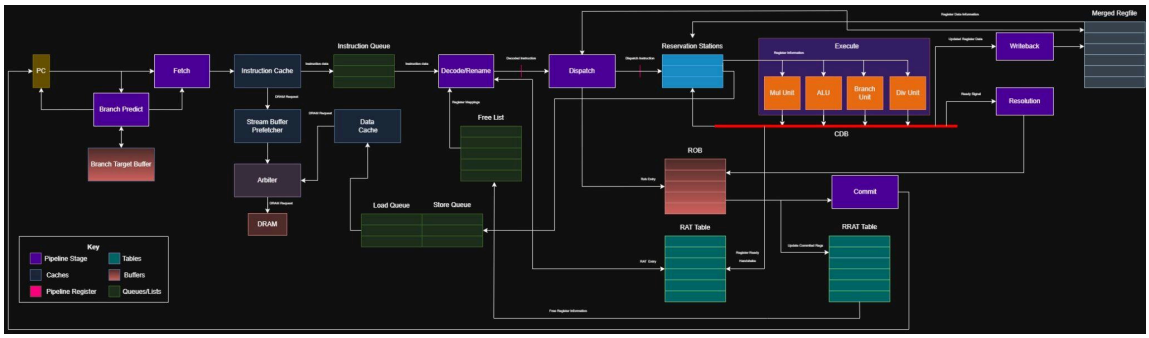

Figure 3: CP3 Block Diagram

III. Advanced Design Choices

A. Bimodal Branch Predictor

The bimodal branch predictor uses a BHT to predict whether a branch will be taken or not taken. Each BHT entry contains a 2-bit counter representing strongly not taken, weakly not taken, weakly taken, or strongly taken. The most significant bit of the counter is used as the prediction and the BHT is indexed using bits from the instruction PC. It is implemented with a SRAM with both read and write ports that allows the frontend to read a prediction while the backend updates the table when a branch resolves. Since the SRAM itself is not resettable, a separate flip-flop valid-bit array is used to track whether each entry has been initialized. Invalid entries default to weakly not taken. When a branch resolves, the corresponding counter is incremented if the branch was taken and decremented if it was not taken.

The bimodal predictor was tested by running full benchmarks such as coremark_im, fft, aes_sha, agr, compression, image, and mergesort. Since branch prediction was added after the rest of the pipeline was mostly functional, we mainly verified it through full-program correctness and performance comparison against the SNT baseline.

The bimodal predictor with BTB improved the average IPC from 0.2746 to 0.5261 compared to SNT, which is about a 91.6% improvement. On coremark_im, IPC improved from 0.2374 to 0.5783. Branch accuracy also improved significantly. For example, coremark_im improved from 46.06% to 85.56%, and compression improved from 52.40% to 92.35%. The frontend stall rate also dropped on coremark_im from 54.58% to 18.52%, showing that the predictor helped the frontend provide useful instructions more consistently.

Benchmark	SNT	Bimodal+BTB	Gshare	Final (Bimodal-128)
coremark_im	0.2374	0.5783	0.5196	0.7007

fft	0.237	0.5729	0.5812	0.6721
aes_sha	0.2126	0.4165	0.4125	0.5176
agr	0.2575	0.5119	0.4968	0.6069
compression	0.3572	0.6528	0.6215	0.8254
image	0.2957	0.4684	0.4693	0.4974
mergesort	0.3291	0.4816	0.4249	0.5716
Geomean	0.2746	0.5261	0.5007	0.6336

Table 1: Branch Predictor IPC Statistics

Benchmark	SNT	Bimodal+BTB	Gshare	Final (Bimodal-128)
coremark_im	46.06	85.56	74.36	91.52
fft	37.59	75.32	82.26	75.66
aes_sha	54.41	46.54	44.66	55.56
agr	58.85	75.33	70.37	83.46
compression	52.4	92.35	85.28	92.35
image	63.46	62.92	63.95	89.33
mergesort	70.44	77.73	68.31	76.84

Table 2: Branch Predictor Accuracy Statistics

Benchmark	Config	Frontend Stall	Redirect	Predicted Taken
coremark_im	SNT	54.58%	2.72%	0.00%
coremark_im	Bimodal+BTB	18.52%	1.70%	8.80%
coremark_im	Final	17.26%	1.19%	10.83%
fft	SNT	67.57%	0.55%	0.00%
fft	Bimodal+BTB	30.41%	0.40%	2.64%
fft	Final	18.95%	0.47%	2.29%
aes_sha	SNT	71.87%	0.57%	0.00%
aes_sha	Bimodal+BTB	44.42%	0.75%	1.79%
aes_sha	Final	38.13%	0.70%	2.80%
agr	SNT	62.41%	1.64%	0.00%
agr	Bimodal+BTB	26.31%	1.40%	5.05%
agr	Final	21.66%	1.09%	6.62%

Table 3: Branch Predictor Profiling Metrics

B. Gshare Predictor

The gshare predictor is similar to the bimodal predictor because it also uses a BHT with 2-bit saturating counters. The main difference is the indexing scheme. Instead of indexing only with PC bits, gshare XORs PC information with a global history register. The global history register records the recent taken/not-taken outcomes of previous branches. When a branch resolves, the GHR shifts in 1 for taken and 0 for not taken. This allows gshare to use global branch behavior when making predictions, which can help when one branch's behavior depends on previous branches. The counter update logic is the same as bimodal. Taken branches increment the counter, and not-taken branches decrement it.

Gshare was tested using the same benchmark-based method as bimodal. We ran the long benchmarks, checked that the programs completed correctly, and compared IPC and branch accuracy against both SNT and bimodal.

Gshare improved over SNT but performed consistently worse than bimodal overall. SNT had a geometric mean IPC of 0.2746, gshare reached 0.5007, and bimodal + BTB reached 0.5261. Gshare did help on some benchmarks. For example, on fft, IPC improved from 0.5729 with bimodal + BTB to 0.5812 with gshare, and branch accuracy improved from 75.32% to 82.26%. However, gshare was worse on benchmarks such as coremark_im, compression, and mergesort. This suggests that global history was not consistently helpful for our benchmark set or the indexing for our XOR operation caused too much aliasing. Because the bimodal was more stable overall, the final design used the bimodal predictor instead of gshare.

C. Branch Target Buffer

The BTB stores target PCs for taken branches. The branch predictor decides whether a branch is taken, but if the branch is predicted taken, the frontend also needs the target address. The BTB provides this address so the frontend can continue fetching from the predicted target path. The BTB is structured like a small direct-mapped cache. It is indexed using PC bits, and each entry stores a tag and a target PC. The tag is compared with the current PC tag to confirm that the BTB entry matches the current branch. A valid-bit array is also used to track initialized entries. The BTB uses a SRAM, similar to the BHT with simultaneous reads and writes so the frontend reads from the BTB during prediction, while the backend updates the BTB when a taken branch resolves. The BTB only updates on taken branches because not-taken branches do not need a target address.

The BTB was tested together with the bimodal and gshare predictors by running the same full benchmarks.

The BTB helped reduce the penalty of taken branches by allowing the frontend to fetch from the predicted target earlier; however, it is difficult to measure the individual effect of the BTB because our branch predictor design was made that the bimodal and gshare needs the BTB to function.

D. Non-Blocking Data Cache

The non-blocking data cache extends the pipelined cache design with parametrizable MSHR support, allowing multiple outstanding cache misses to be serviced concurrently

without stalling hits to already-resident lines. On a miss, rather than asserting backpressure immediately, the cache allocates an MSHR for the missing line and issues the request to DRAM, freeing the cache to continue serving hits from other lines. The ready signal logic is extended relative to the instruction cache: backpressure is asserted not only when all MSHRs are occupied but also when a write must occur to the SRAM, since the SRAM is single-ported and a fulfilled MSHR returning data must be given priority over new incoming requests to avoid port contention.

The data cache was verified through a custom testbench targeting hit, miss, and concurrent miss scenarios before being validated end-to-end via the full benchmark suite and RVFI monitor. One significant correctness issue arose specifically with pipeline flushes interacting with outstanding MSHRs. If a speculative load issued a request to the data cache for line N and that request missed, an MSHR was allocated and the DRAM request was in flight. If a branch misprediction then flushed the pipeline before the response arrived, a new load could come in targeting the same line N. Because the LSQ had previously been responsible for preventing duplicate requests to the same line, after a flush this invariant no longer held, and the returning DRAM data for the original speculative request was incorrectly matched to the new load, producing wrong results. We resolved this by having the LSQ mark those in-flight requests as pending even across a flush, allowing the outstanding MSHRs to drain and their responses to be discarded before any new requests to the same line were issued.

The non-blocking data cache with two MSHRs improved IPC consistently across nearly all benchmarks relative to the single-MSHR baseline. On aes_sha, IPC improved by 9.9%, on fft by 9.1%, and on coremark_im by 5.9%. These gains are well explained by the reduction in LSU backpressure events. On coremark_im, LSU backpressure dropped from 65,156 events to 10,261, and on aes_sha from 597,757 to 156,141, which confirms that the second MSHR meaningfully reduces time spent stalling on concurrent misses. The image benchmark, which has a very low data cache miss rate of 0.05%, saw only a 0.9% IPC improvement, as expected since there are few misses to overlap in the first place.

Benchmark	MSHR=1 IPC	MSHR=2 IPC	Δ IPC	MSHR=1 D$ Ready	MSHR=2 D$ Ready	MSHR=1 LSU BP	MSHR=2 LSU BP
coremark_im	0.662	0.7007	+5.9%	98.92%	99.59%	65,156	10,261
fft	0.6162	0.6721	+9.1%	92.78%	97.02%	404,349	115,988
aes_sha	0.4711	0.5176	+9.9%	88.58%	96.94%	597,757	156,141
agr	0.5685	0.6069	+6.8%	92.93%	97.82%	1,323,576	383,021
compression	0.7975	0.8254	+3.5%	94.77%	99.04%	71,747	43,253
image	0.4931	0.4974	+0.9%	99.94%	99.89%	16,962	4,577

merges ort	0.5413	0.5716	+5.6%	89.56%	97.04%	167,605	52,801

Table 4: Non-Blocking DCache Performance Metrics

E. Stream Buffer Prefetcher

The stream buffer was designed as a two-line prefetch buffer sitting between the instruction cache and the memory hierarchy. When the instruction cache misses on line N, rather than requesting only line N from DRAM, the request is first sent to the stream buffer. The stream buffer checks if line N is already resident in one of its two storage slots. If it is, it returns the line immediately with only a one-cycle latency, turning what would have been a DRAM miss into a near-hit. If line N is not present, the stream buffer issues requests for lines N, N+1, and N+2 through an arbiter to DRAM via the cacheline adapter. The arbiter contains its own MSHRs, allowing all three requests to be issued back-to-back without waiting for the first response, taking advantage of the DRAM model's support for multiple outstanding requests. When DRAM responds, the arbiter matches each response to the corresponding outstanding request and routes it appropriately: for instruction cache misses, responses flow back through the stream buffer, which forwards line N to the instruction cache and retains lines N+1 and N+2 for future accesses. For data cache misses, responses are routed directly back to the data cache since there is no prefetcher on the data side.

The stream buffer was primarily tested by running the full benchmark suite and verifying correctness through the RVFI monitor, since its behavior is transparent to program correctness and its effect is only visible in performance.

The stream buffer had the largest impact on instruction-fetch-bound workloads. On aes_sha, IPC improved by 20.5% and the frontend stall rate dropped from 49.56% to 38.13%, reflecting the stream buffer's effectiveness at hiding instruction cache miss latency by prefetching ahead. On coremark_im, IPC improved by 10.9% and frontend stalls dropped from 25.50% to 17.26%. Workloads with high instruction cache locality and few cold misses, such as fft, compression, and mergesort, saw negligible improvement since the instruction cache was already serving most requests from its resident lines. The primary design tradeoff of the stream buffer is the added DRAM bandwidth consumed by prefetching lines N+1 and N+2 that may never be used, though this had no measurable negative effect on our benchmarks.

Bench mark	No Prefetch IPC	With Prefetch IPC	Δ IPC	No Pf I$ DRAM Reads	Pf I$ DRAM Reads	No Pf Frontend Stall	Pf Frontend Stall
corema rk_im	0.6315	0.7007	+10.9%	3,088	3,585	25.50%	17.26%
fft	0.6721	0.6721	0.00%	65	77	18.96%	18.95%

aes_sha	0.4297	0.5176	+20.5%	31,056	52,511	49.56%	38.13%
agr	0.572	0.6069	+6.1%	37,791	65,302	26.53%	21.66%
compression	0.8246	0.8254	+0.1%	26	34	5.82%	5.73%
image	0.4917	0.4974	+1.2%	3,519	9,052	20.45%	18.85%
mergesort	0.571	0.5716	+0.1%	37	43	13.67%	13.56%

Table 5: Prefetcher Performance Metrics

F. Pipelined Cache

The pipelined instruction cache was designed to allow back-to-back reads every cycle with a one-cycle response latency on hits. When a request comes in, the address is fed into the SRAM arrays and saved into a pipeline register simultaneously. On the following cycle, the tag comparison is performed to determine if the access is a hit. If it is, the data is returned and a ready signal is asserted to the fetch stage, indicating that the cache can accept another request that same cycle. This creates a fully pipelined flow where hits are serviced at one per cycle. On a miss, the ready signal is deasserted, preventing fetch from issuing further requests while the cache services the miss by fetching the line from the next level of the hierarchy. Since the instruction cache is read-only, dirty writebacks are never required, simplifying the miss handling path considerably. The ready signal serves as the primary handshake between the cache and the fetch stage, cleanly encoding whether the cache is busy handling a miss or available to accept new work.

The pipelined cache was verified through a custom testbench that exercised hit and miss scenarios, back-to-back requests, and boundary conditions around the ready signal. It was then further validated by running the full benchmark suite and confirming correctness through the RVFI monitor.

The pipelined cache primarily benefits fetch-bound workloads by eliminating the one-cycle bubble that a non-pipelined cache would incur between consecutive hits. The main design tradeoff is the added complexity of the pipeline register and the need to correctly handle the case where a request is in-flight in the pipeline stage when a miss occurs, requiring care to not drop or corrupt the in-progress access. The pipelined cache was implemented from the start, and therefore, we did obtain any metrics to compare it to besides inspecting the performance counters of the workloads independently.

G. Split LSQ

The Split LSQ helped to manage loads and stores with separate readiness, issue, and commit logic. Before this, the unified LSQ simplified ordering but limited how efficiently loads and stores could be tracked independently. The main trade-off was the added complexity in exchange for better MLP. A split design required more bookkeeping to preserve ordering, since we did not have any load-store forwarding.

Overall, the split LSQ improved the IPC on most benchmarks. The largest improvement came from image, which had a 12.3% speedup, followed by an 8.6% speedup for coremark_im, and an 8.2% speedup for compression. These results suggest that the split LSQ helped most on workloads with frequent memory operations where allowing loads and stores to progress independently reduced pipeline stalls. Additionally, mergesort improved by 6.7%, likely due to its memory-heavy behavior. While some benchmarks noted speedups, the split LSQ did not help every benchmark. FFT had a 1.9% slowdown, suggesting that there may have been fewer useful opportunities for load/store overlap or that there wasn't enough parallelism throughout the pipeline for this benchmark. The performance counters show that the split LSQ also affected LSU backpressure. For most benchmarks, post-split LSU backpressure increased, such as aes_sha rising from 388,195 to 584,726, agr rising from 913,778 to 1,402,413, and fft rising from 251,743 to 427,551. This indicates that although IPC often improved, the split LSQ created more attempts to issue memory operations toward the LSU/cache path, increasing pressure on the memory system. In other words, the split LSQ exposed more MLP, but the downstream LSU and cache interface became a more visible bottleneck. The exception was image, where LSU backpressure decreased from 22,831 to 21,025 while IPC improved by 12.3%. This was the strongest result for the split LSQ because it improved top-level throughput while also reducing memory-side blocking. For benchmarks like coremark_im, compression, and mergesort, the IPC gain outweighed the increase in LSU backpressure. For fft, the increased LSU backpressure aligned with the IPC slowdown, showing that the split LSQ made the memory subsystem busier without improving useful instruction throughput.

Overall, the split LSQ was a net positive advanced feature. It improved IPC on most of the measured benchmarks and produced especially strong gains on image, coremark_im, compression, and mergesort. The main cost was increased design complexity and greater LSU/cache pressure, but the performance results show that separating load and store tracking generally allowed the processor to make better forward progress on memory-intensive workloads.

Benchmark	Pre-Split IPC	Post-Split IPC	Δ IPC	Pre-Split LSU BP	Post-Split LSU BP
coremark_im	0.6414	0.6969	+8.6%	62,515	76,588
fft	0.6438	0.6318	-1.90%	251,743	427,551
aes_sha	0.4836	0.4924	+1.8%	388,195	584,726
agr	0.568	0.5903	+3.9%	913,778	1,402,413
compression	0.7042	0.7621	+8.2%	88,495	106,349
image	0.4793	0.5381	+12.3%	22,831	21,025
mergesort	0.5321	0.5679	+6.7%	99,999	186,174

Table 6: Split LSQ Performance Metrics

H. Age-Ordered Issue Scheduling

The next advanced feature we implemented was age-ordered issue scheduling in the reservation stations. Before this change, the reservation station could issue the first ready instruction it found, which was simpler but allowed younger instructions to repeatedly issue before older ready instructions. This tended to delay instructions that were closer to commit. With this scheduling, the issue logic would always prioritize the oldest ready instruction in the reservation station. To implement this feature, each RS entry tracks the instruction's age using its ROB index. When multiple instructions are ready in the same cycle, the scheduler compares their ROB indices and selects the oldest valid ready entry. This chosen instruction is then issued to the execute stage, and the corresponding reservation station entry is cleared. The design still respects normal readiness conditions, meaning an instruction can only issue if both operands are ready and the downstream execute unit is available. The main trade-off of age-ordered scheduling is that it improves fairness and commit progress at the cost of more complicated selection logic. Instead of a simple priority encoder, the scheduler must compare multiple ready entries to determine which instruction is oldest. This can increase combinational delay in the issue path, especially as reservation station size grows. However, the benefit is that older instructions are less likely to be starved behind younger independent instructions, which can help keep the ROB draining smoothly. However, age-ordered scheduling does not always improve performance. On workloads where there are rarely multiple ready instructions at the same time, the scheduler behaves similarly to the simpler issue policy. It may also slightly hurt timing or performance if the oldest ready instruction is not the one that would best utilize a specific functional unit. For example, a purely oldest-first policy may issue an older long-latency instruction or memory instruction while a younger simple ALU instruction could be completed quickly.

While the IPC increased for all of the benchmarks, the max clock frequency decreased from 500 MHz to 300 MHz. The largest speedup was for aes_sha at 13%, while all other improvements were between 1-5%. In the end, this wasn't worth it due to the larger critical path from the decrease in clock frequency. For this reason, we ultimately chose not to submit this feature to our main branch for passing performance benchmarks.

Benchmark	Baseline IPC	Age IPC	IPC Ratio	Baseline Throughput (MIPS)	Age Throughput (MIPS)	Throughput Ratio	Baseline Time (us)	Age Time (us)	Wall-Clock Speedup
coremark_im	0.7007	0.72	1.03	350.4	216	0.62	868	1408	0.62
fft	0.6721	0.6893	1.03	336.1	206.8	0.62	3472	5642	0.62
aes_sha	0.5176	0.5842	1.13	258.8	175.3	0.68	2999	4428	0.68
agr	0.6069	0.6372	1.05	303.5	191.2	0.63	12037	19106	0.63

compression	0.8254	0.8345	1.01	412.7	250.4	0.61	1055	1739	0.61
image	0.4974	0.5115	1.03	248.7	153.5	0.62	1918	3109	0.62
mergesort	0.5716	0.5911	1.03	285.8	177.3	0.62	1726	2781	0.62
Geomean	0.6272	0.6502	1.04	313.6	195	0.62			0.62

Table 7: Age-Ordered Issue Scheduling Performance Metrics

Benchmark	Baseline Rename Blocked (%)	Age Rename Blocked (%)	Baseline RS-Full Blocks	Age RS-Full Blocks	Baseline LSU Backpressure	Age LSU Backpressure	Baseline Frontend Stall (%)	Age Frontend Stall (%)	Baseline Avg ROB Occ	Age Avg ROB Occ
coremark_im	10.83	11.46	39101	42493	10261	9936	17.26	13.95	8.2	8.6
fft	23.41	21.88	360674	294244	115988	98484	18.95	17.08	14.7	14.1
aes_sha	9.16	8.04	60459	46653	156141	134908	38.13	29	9.7	10.3
compression	5.18	4.03	25420	19297	43253	42872	5.73	4.82	8.9	8.7
image	19.44	21.72	147343	142239	4577	4881	18.85	17.59	11.6	11.5
mergesort	13.98	9.29	69050	38449	52801	42598	13.56	9.9	10.2	9.4

Table 8: Age-Ordered Issue Scheduling Profiling Metrics

I. Paramatrizable Sets & Ways for Caches

Another advanced design feature implemented in our processor was making the instruction and data caches parametrizable in both the number of sets and the number of ways. Instead of hardcoding a single cache configuration, the cache modules were designed using parameters so that associativity and cache size could be modified without changing the overall cache logic. This made the cache subsystem significantly more flexible for experimentation and testing. The cache design used parameters such as NUM_sets and NUM_WAYS to determine cache organization at compile time. Address decomposition logic automatically derived the required tag, index, and offset widths from these parameters using $clog2. The replacement logic, valid bits, dirty bits, and PLRU metadata were all generated based on the chosen cache configuration. The main trade-off of a parametrizable cache design is increased implementation complexity in exchange for flexibility and scalability. Hardcoded caches are simpler to debug because all dimensions are fixed, while parametrizable designs require more generalized indexing, replacement, and metadata handling logic.

Verification involved testing multiple cache configurations to ensure that hit detection, cache fills, writebacks, replacement behavior, and memory accesses continued to function correctly regardless of cache size or associativity. In addition, we ran each of the provided benchmarks. In the end, we used 16 sets and 4 ways in our cache.

J. Benchmark Analysis

To better evaluate architectural changes and advanced design options, we developed an automated benchmark analysis and reporting framework using a custom Bash script. The script automated compilation, synthesis, benchmark execution, power analysis, and report generation for all checkpoint benchmark programs. This allowed consistent and repeatable evaluation of processor performance across multiple design iterations. The framework executes all benchmarks in parallel after synthesis to reduce total runtime and improve iteration speed during development. For each benchmark, the script collects IPC, execution time, power, and PD^4 metrics. Power analysis is performed by generating SAIF switching activity files from FSDB waveforms and running Synopsys DC power estimation tools. The final reports automatically compare results against baseline reference values and compute percentage improvements or regressions.

In addition to top-level metrics like IPC, the benchmark infrastructure was useful for analyzing performance counters such as LSU backpressure, branch prediction behavior, and issue-stage utilization. These counters helped identify pipeline bottlenecks and determine whether architectural changes improved useful instruction throughput or simply shifted pressure elsewhere in the processor. The automation framework significantly improved the efficiency of debugging and architectural exploration throughout the project.

To test this, we ran each of the benchmarks and analyzed all of the results. We then used these stats to further tune our other features or modules, whether it was to update our branch prediction mode, reduce any stalls, or to adjust any of our parameters.

Benchmark	Instructions	Cycles	IPC	Br. Accuracy	D$ Miss Rate	Frontend Stall	Redirect	LSU BP Events
coremark_im	304,044	433,912	0.7007	91.52%	0.27%	17.26%	1.19%	10,261
fft	1,166,732	1,736,010	0.6721	75.66%	1.91%	18.95%	0.47%	115,988
aes_sha	776,008	1,499,363	0.5176	55.56%	2.13%	38.13%	0.70%	156,141
agr	3,652,225	6,018,271	0.6069	83.46%	1.86%	21.66%	1.09%	383,021
compression	435,228	527,281	0.8254	92.35%	1.29%	5.73%	0.95%	43,253
image	477,026	958,959	0.4974	89.33%	0.05%	18.85%	1.47%	4,577
mergesort	493,187	862,746	0.5716	76.84%	2.89%	13.56%	2.62%	52,801
Geomean	—	—	0.6289	—	—	—	—	—

Table 9: Benchmarking Script Basic Performance Metrics

Configuration	Key Features Added	coremark_im IPC	vs Previous
SNT baseline	Static not-taken, no BTB	0.2374	—
Bimodal + BTB	Bimodal BHT=16 + BTB=16	0.5783	+143.7%
Split LSQ + wider ROB	OoO load bypass, ROB=32, COMMIT=4	0.6969	+20.5%
Dual MSHR	Non-blocking cache (MSHR=2)	0.7007	+0.5%
Stream buffer	I-cache 2-line lookahead prefetch	0.7007	—

Table 10: System Configuration Comparisons

Conclusion

Overall, this project successfully developed an out-of-order RV32IM processor from an initial frontend design into a fully functional and optimized core. Across the three checkpoints, we built the fetch path, instruction queue, cache interface, rename and dispatch logic, execution units, ROB, CDB, LSU/LSQ, and memory hierarchy. Each checkpoint expanded the processor's functionality, eventually allowing the design to run the full benchmark suite correctly through the RVFI/Spike monitor.

After completing the baseline processor, we focused on improving performance through advanced features. Branch prediction and the BTB had the largest early impact by reducing frontend stalls and improving IPC compared to the static-not-taken baseline. The split LSQ, non-blocking data cache, stream buffer prefetcher, and parametrizable cache design further improved throughput by reducing backend and memory-system bottlenecks. Some features, such as gshare and age-ordered issue scheduling, were useful

to explore but did not consistently improve final benchmark performance, so the final design kept the features that were most stable across workloads.

The final processor achieved a geometric mean IPC of about 0.6289 across the benchmark suite, with especially strong performance on compression, coremark_im, and fft. More importantly, the project showed how frontend prediction, memory parallelism, cache behavior, and scheduling policy all interact in an out-of-order processor. The final design was not only a working RV32IM core, but also a processor whose bottlenecks were measured, analyzed, and improved through targeted architectural changes.