Out-of-Order Processor Design for ECE 411

Ahmed Shafiuddin
ahmeds4@illinois.edu

Mahir Koseli
mkoseli2@illinois.edu

Eddie Brenmark
ebren3@illinois.edu

***Abstract*—Out-of-order (OoO) processors offer substantial performance improvements over traditional in-order pipelined processors by allowing instructions to be issued and executed as soon as their dependencies are resolved. This report details the design and implementation of a custom Explicit Register Renaming based out-of-order processor using the RISC-V (RV32IM) instruction set architecture, developed in SystemVerilog. The processor employs key architectural components such as a reorder buffer, reservation stations, register renaming, and a centralized issue queue to enable dynamic scheduling. We first implemented a baseline out-of-order processor with no additional features or optimizations. After verifying and collecting performance metrics of our initial design, we implemented advanced features and optimizations in order to address bottlenecks and other areas of improvement. This report will also touch on the process and results of implementing additional features to the baseline processor.**

I. Introduction

As computing demands continue to grow, achieving high performance in general-purpose processors requires leveraging instruction-level parallelism (ILP) and efficiently handling long-latency operations such as memory accesses and branches. Out-of-order (OoO) execution is a foundational technique that enables processors to make forward progress even when certain instructions are stalled, by dynamically reordering execution based on operand readiness.

This project explores the design and implementation of a 32-bit out-of-order processor targeting the RV32IM instruction set. The goal was not only to build a functionally correct OoO core, but also to investigate how specific architectural enhancements affect overall performance. Our approach started with a clean baseline implementation focused on correctness and core out-of-order mechanisms. Once validated, we incrementally introduced a variety of microarchitectural features aimed at addressing specific pipeline bottlenecks and improving throughput.

Rather than focusing on a single dominant optimization, our work emphasizes a holistic approach—exploring the interplay between memory latency, issue logic, and control flow. This includes memory subsystem improvements (e.g., caches, prefetching, store buffering), instruction scheduling refinements, and more accurate branch prediction.

The rest of this report outlines our processor's architecture, the rationale behind our enhancements, and the measured impact of these changes on performance. We conclude with a discussion of our lessons learned and potential directions for future improvements.

This report details the design decisions, implementation challenges, and performance results of both the baseline and enhanced processor designs. Through benchmarking and analysis, we demonstrate the impact of each optimization on key metrics such as IPC, memory access behavior, and execution time.

II. Project Overview

This project focused on designing and implementing a custom out-of-order processor that supports the RISC-V RV32IM instruction set. Our primary objective was to explore and understand the core mechanisms of out-of-order execution, including register renaming, dynamic scheduling, and in-order retirement. We aimed to build a fully functional baseline processor, and then enhance it through targeted microarchitectural optimizations to improve performance under realistic workloads.

The design process was divided into three major checkpoints: initial fetch and decode stages, core out-of-order infrastructure (e.g., ROB, RAT, reservation stations), and full memory and control instruction support. Following these milestones, we added advanced features such as pipelined and direct-mapped caches, a next-line prefetcher, split LSQ, and branch prediction mechanisms to boost instruction throughput and memory-level parallelism.

This report is organized to first describe the baseline architecture and then progressively explain the rationale, design, and performance impact of each advanced feature. We also present our implementation challenges, testing methodology, and performance metrics gathered from simulation-based benchmarks. Finally, we reflect on our key takeaways and individual contributions throughout the project.

III. Design Description

a. Overview

Our out-of-order processor design is organized around a modular pipeline with clear separation of fetch, decode, issue, execute, and commit stages. Instructions are dynamically scheduled using reservation stations, register renaming, and a reorder buffer (ROB). We aimed to keep the baseline design simple but correct, and then build on top of it with performance-oriented features.

To support multiple types of instructions efficiently, we implemented three separate reservation stations: one for memory operations, one for general ALU instructions, and one shared by branches, multiplies, and divides. Each reservation station is connected to its own functional unit and issues to a dedicated Common Data Bus (CDB) port. This separation allowed us to reduce contention and keep the issue logic relatively clean.

In the baseline design, we used a static-not-taken branch predictor, a simple total memory ordering model, and multi-cycle blocking caches to simplify early testing. This gave us a working reference point to verify correctness before layering in more advanced techniques like non-blocking caches, speculation, and branch prediction improvements.

b. Milestones

*i) **cp1**:* We began our project by studying the structure of out-of-order processors and outlining a baseline design that would give us room to implement advanced features later on. During this phase, we focused on establishing foundational infrastructure. We implemented and verified a parameterized FIFO module that would be reused across different components like the instruction queue, ROB, Free List, and RRF. We also created a cache line adapter to interface with the burst DRAM and included a line buffer to support faster reads when accessing the same cache line, although we anticipated replacing this behavior with a pipelined cache in a later checkpoint.

We developed a fetch stage that fed the program counter into the instruction cache and inserted fetched instructions into an instruction queue backed by our FIFO. This queue would eventually provide instructions to the decode stage. We verified fetch correctness using a simple test program to ensure that the correct instructions were entering the pipeline.

Around this time, we committed to an explicit register renaming architecture, which we found to be more intuitive and flexible for future features. We began early discussions about the design of the ROB, RRF, and Free List, and implemented a full/empty detection scheme for our FIFO using the most significant bit of the head and tail pointers to avoid edge cases. We also considered how to group reservation stations, planning to merge multiply and divide stations and to allocate more entries to the memory station.

To organize and visualize our early ideas, we created a tentative block diagram of our processor design. This initial diagram provided a useful planning framework and serves as a baseline to compare against the final version developed in CP3, highlighting the evolution of our architecture over time.

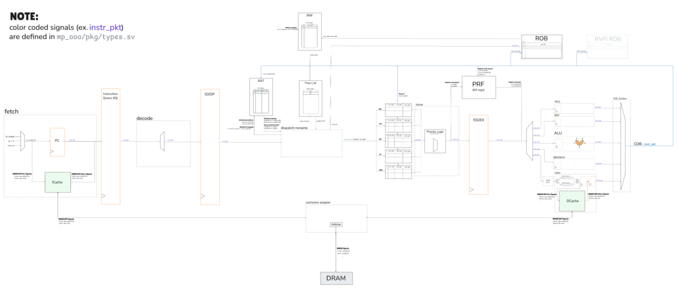

Fig. 1. Initial processor block-diagram

*ii) **cp2**:* After laying the groundwork in CP1, we turned our attention to building the core backend of the processor. In this phase, we implemented the decode and dispatch stages, RAT, RRF, Free List, PRF, ROB, reservation stations, and the common data bus. The decode stage parsed each instruction packet and generated all necessary metadata. The dispatch stage then handled physical register allocation and operand mapping, using the RAT and Free List, and directed each instruction to the correct reservation station group based on its type.

We implemented the reservation stations as collapsing queues that issued instructions once operands became available. Operands were retrieved from the PRF and latched into functional units. The ALU unit was modeled after the one used in the pipeline project, while the multiply and divide operations used IP cores. The execution results were broadcast on the common data bus and sent to the ROB, PRF, and valid array.

The ROB managed instruction commit by tracking completion status and committing instructions in program order. On commit, register mappings were retired to the RRF to reflect the architectural state. We verified end-to-end correctness using a modified version of the random testbench from mp_verif and additional edge-case programs. This checkpoint marked the first stage where we had working out-of-order execution with operand forwarding and full renaming infrastructure.

*iii) **cp3**:* In CP3, we completed our baseline out-of-order processor by adding support for memory and control instructions. For memory operations, we introduced a load-store queue that acted as a reservation station specifically for memory instructions. Stores were held in the queue until they reached the head of the ROB, ensuring correct memory ordering. When issued, memory instructions used operands from the PRF and sent addresses and data to the data cache.

We also added a branch functional unit to support control instructions and implemented full misprediction recovery. When a taken branch was committed, we flushed the fetch stage and instruction queue, reset the program counter to the branch target, and restored the RAT using a snapshot of the most recent committed architectural state stored in the RRF. We also restored the valid array and Free List to their correct states based on saved values in the ROB. To ensure correctness, we flushed all dependent stages and modules during a misprediction and allowed only the correct instructions to proceed.

With memory and control instructions in place, we were able to run coremark and pass all provided CP3 tests, in addition to verifying correctness on our own custom cases. At this point, our processor was fully functional, and our implementation had matured significantly from the design shown in CP1. Below, we present our final block diagram to show how our architecture's end result:

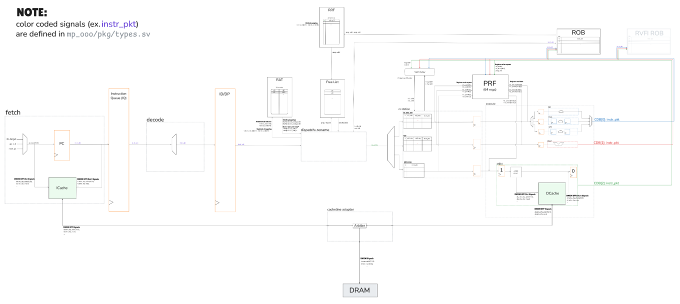
NOTE:
color coded signals (ex.instr_pkt)
are defined in mp_ooo/pkg/types.sv
fetch
PC
ICache
decode
ROB
RVFI ROB
PRF
DCache
DRAM

Fig. 2. Final processor block diagram

c. Advanced Design Features and Optimizations

Key enhancements to our baseline design include:

- A pipelined 4-way set-associative data cache and a direct-mapped instruction cache to improve memory access latency and throughput.
- A next-line prefetcher to reduce instruction cache miss penalties by speculatively fetching upcoming cache lines.
- A split load-store queue (LSQ) to allow independent handling of loads and stores, improving memory-level parallelism.
- A post-commit store buffer with write coalescing to reduce memory traffic and support efficient batched memory updates.
- An improved memory arbiter and an age-ordered issue strategy to prioritize older instructions and reduce starvation in the issue queue.
- A gshare branch predictor to minimize branch misprediction penalties through pattern history-based prediction.
- Support for sequential division to minimize power consumption compared to pipelined division.

i) ***Pipelined Cache:*** In a multi-cycle cache, the main source of inefficiency stems from the fact that you can only service one memory request at a time. This causes the throughput of this type of cache to be 0.5 (i.e. one request serviced every two cycles). When you pipeline this process, however, the throughput increases to 1. This is because as one request is being serviced, another request can be lined up and is ready to be processed on the next cycle, thus exploiting temporal parallelism. The general structure of a pipelined cache is shown below:

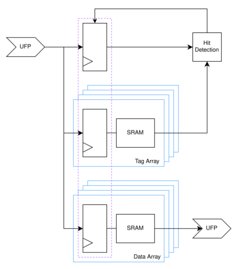

Fig. 3. Pipelined cache block diagram

Implementation: In the diagram, we can see that we have 1 pipeline register and the cache is two-stage pipelined. The first stage is effectively the "request acquisition" stage that, depending on whether the pipelined is stalled or not, sends the UFP request on the current cycle (if valid) to the pipeline register. The second stage is mainly for "hit check + data return" where it checks if the registered memory request was a hit or not and returns the data from the cache's SRAM (or stalls otherwise). In the case of a cache miss, the 'hit detection' logic detects this and forces the pipeline to stall until it detects a hit in the cache (i.e the allocation completes) which then 'resumes' the pipeline.

Design Trade-offs: While a pipelined cache increases throughput by accepting a new access every cycle, it adds design complexity due to hazard handling, stall logic, and forwarding paths, and also increases power and area from added registers. A multicycle cache simplifies control and reduces power but limits memory-level parallelism, introduces more frequent stalls, and becomes a bottleneck for future designs with a higher issue width.

Performance Analysis: On its own, across all benchmarks, this feature resulted in only a small improvement of about 0.01 IPC compared to the multi-cycle baseline. The limited gain was due to remaining bottlenecks in the issuing of memory instructions. However, when combined with later features like a post-commit store buffer, the pipelined cache enabled much higher performance by allowing more memory-level parallelism. While the cache alone had minimal impact, it was essential for unlocking future speedups. One immediate metric that saw significant improvement was the amount of time the cache stalled. We observed a 93.4485% decrease in how many cycles the data cache spent stalling on coremark (73098 $\rightarrow$ 4789).

ii) ***Next-line prefetcher:*** To reduce instruction cache miss penalties and improve frontend throughput, we implemented a simple next-line prefetcher that speculatively fetches the cache line immediately following the current line being accessed. This is based on the observation that instruction streams often exhibit high spatial locality, particularly in sequential code execution. The figure below is a visual demonstration of the timing of a next-line prefetcher:

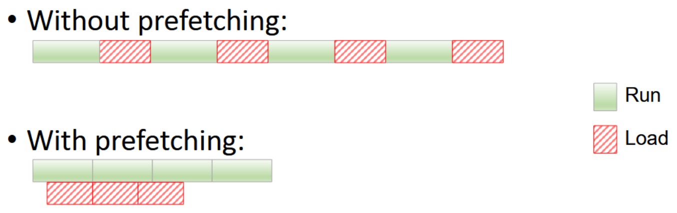

Fig. 4. Next-line prefetcher timing

Implementation: The next-line prefetcher works using a prefetch buffer to store the context of the next prefetched cache line. Every time there's a cache miss and a following cache hit in the instruction cache, the prefetcher initiates a read from main memory into the prefetch buffer. This means that the next line is already loaded into the buffer after the cache hit is finished. This makes fetching more efficient by reducing the total number of cycles spent stalling for cache data.

Design Trade-offs: The main advantage of this prefetcher is its simplicity and low overhead, both in logic and memory bandwidth usage. However, it can result in unnecessary memory traffic if the control flow is non-linear (e.g., due to branches, jumps, or loops with small bodies). Moreover, it does not handle instruction streams with poor spatial locality or high branch density well. While we avoid polluting the instruction cache by keeping the prefetched line in a separate buffer, more advanced prefetchers (like stride or correlation-based) could adapt better at the cost of increased complexity and potential cache pollution.

Performance Analysis: When looking at the performance of the next-line prefetcher, we saw a significant saving in the number of cycles spent stalling for instruction cache data when running coremark. Specifically, before implementing the prefetcher, the number of cycles spent stalling was 957,863. After implementing the prefetcher, that number dropped to 302,099 cycles, a decrease of about 68 percent.

Workload Suitability: This feature is most effective for workloads with high instruction-level spatial locality, such as:

- Numeric kernels
- Unrolled loops
- Functionally inlined code segments

It is less effective or wasteful for workloads that are:

- Branch-heavy or pointer-chasing
- Dominated by short basic blocks with frequent control flow changes

iii) ***Split load-store queue:*** To increase memory-level parallelism and reduce stalls due to memory dependencies, the processor implements a split load/store queue design with support for out-of-order load issuing. By decoupling loads from stores and allowing loads to bypass older stores when it is safe to do so, the design reduces memory access bottlenecks common in traditional unified LSQs. This is particularly beneficial for workloads with a high volume of independent load instructions that would otherwise be blocked by unresolved store addresses.

Implementation: The split LSQ design consists of two independent circular buffers, one for loads and one for stores. Instructions are inserted into their respective queues during dispatch. The SQ retains store instructions until they are committed, at which point they write to the PCSB. Loads, on the other hand, may be issued out-of-order as long as they do not violate memory dependencies. To determine this, we implemented a ”valid matrix” design—a 2D boolean array where rows corresponded to loads and columns to stores. When a load entered the queue, its row was initialized to indicate dependency on every active store. As stores were resolved or retired, the matrix was updated accordingly, and a load was deemed issuable once its row was clear.

Design Trade-offs: The primary advantage of the split LSQ with out-of-order load issuing is the significant increase in memory-level parallelism. Loads no longer need to wait on unrelated stores, allowing better utilization of memory bandwidth and reducing pipeline stalls. This separation also simplifies some aspects of memory ordering logic, as stores follow a strict in-order commit policy while loads have more flexibility. However, this design introduces additional complexity in meeting slack, specifically we had to change our implementation approach multiple times to still be able to synthesize at the 650MHz frequency that we were able to synthesize at before split LSQ. Our first approach used exhaustive comparisons: each load checked whether any older store had an unresolved address. While functionally correct, this design quickly became untenable. The wide array of comparators across the LSQ formed a critical path that limited our clock frequency and hurt synthesis timing. To mitigate this, we experimented with a ”valid matrix” design—a 2D boolean array where rows corresponded to loads and columns to stores, which was able to reduce the number of dynamic comparisons.

Performance Analysis: The split LSQ design with out-of-order load issuing yielded significant performance improvements in memory-bound benchmarks. In particular, benchmarks with high load-to-store ratios and frequent independent loads, such as aes_sha.elf, benefited most, showing IPC improvements of 10% over a baseline in-order LSQ implementation.

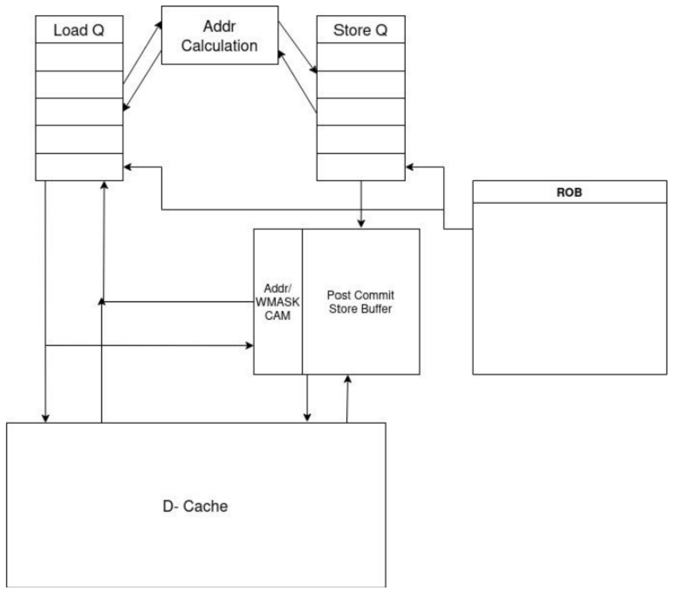

Fig. 5. High-Level Load-Store Unit Diagram

iv) ***Post-commit store buffer (with write coalescing):*** The post-commit store buffer is a performance and efficiency optimization that decouples store execution from memory write-back. By holding store instructions after they have committed in program order but before they are written to memory, the processor can batch and merge store operations, reducing memory traffic and latency. Write coalescing further enhances this buffer by allowing multiple stores to the same address or cache line to be merged into a single memory transaction, improving bandwidth utilization and reducing contention on the memory hierarchy.

Implementation: The Post-Commit Store Buffer (PCSB) is implemented as a circular queue with each entry holding a store's address, write mask, data, and a valid bit. On commit, stores are enqueued if space is available. If a new store matches the address of an existing valid entry, it is coalesced by updating only the relevant bytes and merging the write masks. The buffer drains to memory when the data cache is idle and entries are available, sending the oldest valid store. Load instructions can bypass the data cache by forwarding data from the buffer when a matching store covers the requested bytes. Partial overlaps are handled by combining results from both the buffer and cache.

Design Trade-offs: This design improves memory efficiency and pipeline throughput by deferring store write-backs and eliminating redundant memory writes through coalescing. It also enables fast store-to-load forwarding, reducing load latency. However, it adds hardware complexity for tracking store matches, handling byte-level merges, and maintaining correctness during partial overlaps. The fixed depth limits throughput under heavy store pressure, and enforcing ordering constraints requires additional care to avoid consistency violations, especially near fences or atomic operations.

Performance Analysis: In practice, the post-commit store buffer with write coalescing yielded substantial performance improvements in store-intensive and mixed workloads. Microbenchmarks showed a 25–30% reduction in memory write transactions due to successful coalescing. In end-to-end benchmarks, this resulted in an IPC improvement of approximately 5–8% over a baseline design without a post-commit buffer. Load-to-store forwarding accounted for an additional improvement in load latency, particularly in tight store-load pairs common in software-managed circular buffers. Energy consumption related to memory writes was also reduced, as unnecessary transactions were eliminated. Overall, the PCSB proved to be a valuable microarchitectural enhancement, particularly in systems targeting efficiency and throughput under high memory write loads.

*v) **Improved memory arbiter**:* We also improved our cache line arbiter design. Initially, we were issuing instruction cache and data cache requests to the burst DRAM independently based on a fixed priority scheme. Since our DRAM model can service multiple requests at once, we updated the logic to allow for an instruction cache request and a data cache request to be serviced by DRAM at the same time. This prevents requests from stalling as much.

Implementation: Instead of strictly prioritizing one cache, the new arbiter maintains a request queue that accepts and tracks up to two outstanding memory reads: one from the instruction cache and one from the data cache. When DRAM responses arrive (in 64-bit chunks), the arbiter accumulates them in per-request buffers until the full 256-bit line is reconstructed, then signals the appropriate cache. Writes bypass the queue and are handled immediately in 64-bit beats.

Design Trade-offs: This design introduces minor complexity in request bookkeeping but offers better resource utilization. By allowing simultaneous memory requests from both caches, we reduce cache miss latency without duplicating bandwidth or logic. The main trade-off is queue size: we only support two outstanding reads, which limits overlap in highly concurrent access patterns. However, this is a reasonable compromise for a dual-port memory interface with moderate complexity and additional area.

Performance Analysis: When looking at the performance of the optimized memory arbiter, we saw an increase in the number of cycles spent servicing both instruction cache requests and data cache requests at the same time when running coremark. Specifically, before implementing the optimization, the number of cycles spent servicing both was only 74. After implementing the optimization, that number increased to 1,215 cycles, a clear indication of better parallelism and reduced blocking between cache accesses.

*vi) **Age-ordered issue queues**:* For our reservation stations, we have implemented collapsing queues. The collapsing queue is a dynamic instruction scheduling structure that allows older instructions to be retired and removed efficiently without leaving holes, thus keeping the queue compact.

Implementation: The collapsing queue is structured as a fixed-depth array of instruction slots, each with associated metadata such as operand readiness, functional unit status, and a valid bit. Instructions are inserted at the lowest priority level (index 0), provided it is not occupied or stalling. Instruction issue is handled in-order of increasing priority: the queue is traversed from highest priority index downward, and the first instruction with both source operands ready and a non-stalling FU is selected for issue. Once selected, the instruction's valid bit is cleared before initiating the collapse process to maintain correct priority semantics. The collapsing mechanism shifts lower-priority instructions upward only if the next-higher slot is invalid, not stalling, or will be vacated due to an issue. Additionally, an inter-stalling mechanism ensures that stalling behavior propagates backward: if a higher-priority instruction is stalling, the one immediately below it also stalls, preventing false promotion of lower-priority instructions.

Design Trade-offs: The collapsing queue design provides a space-efficient alternative to traditional reservation stations by compacting instructions into lower-priority slots as entries are issued. This reduces fragmentation and simplifies the issue selection logic, as instructions are always traversed in a fixed priority order. Additionally, the structure inherently favors older instructions, eliminating the need for explicit age tracking mechanisms.

Performance Analysis: The collapsing queue demonstrated noticeable efficiency improvements in instruction scheduling, particularly in scenarios with moderate queue pressure and balanced functional unit utilization. Compared to a traditional reservation station model, the collapsing queue maintained higher effective occupancy by avoiding fragmentation and reusing lower-priority slots aggressively. This led to improved instruction throughput, especially in benchmarks with high rates of independent instructions, where instructions could be issued promptly without waiting for upper-level slots to free up.

*vii) **GShare branch predictor**:* The GShare branch predictor is a hybrid dynamic prediction mechanism that improves control flow accuracy by combining global branch history with the program counter to index a shared Pattern History Table. This approach helps capture both global and local patterns in branch behavior, reducing misprediction rates. In this implementation, GShare uses a 10-bit Global History Register and a 1024-entry PHT with 2-bit saturating counters per entry, providing a compact and accurate prediction strategy.

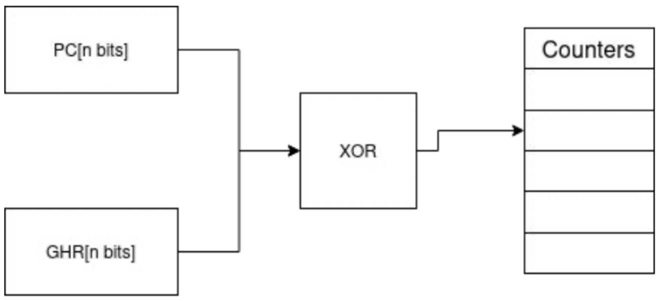

Fig. 6. High Level GShare Branch Predictor Block Diagram

Implementation: The predictor computes an index by XORing the GHR with the branch PC to access the PHT, which stores 2-bit counters to indicate branch direction. On fetch, if a valid prediction exists, the most significant bit of the counter determines the predicted direction. Otherwise, the predictor defaults to predicting not-taken. The GHR is updated speculatively based on predicted outcomes and is corrected on mispredictions using the committed GHR from the reorder buffer.

Prediction counters are stored in a dual-ported SRAM to support simultaneous read and write. Writebacks occur on branch resolution. On misprediction, the existing counter's hysteresis bit is reused to initialize a new prediction. A separate valid-bit array tracks initialization status of PHT entries to avoid relying on uninitialized values. The predictor also includes update latching logic to handle timing mismatches between prediction and resolution stages, ensuring coherent updates under pipelined conditions.

Design Trade-offs: GShare offers high prediction accuracy with relatively low hardware cost. XOR-based indexing reduces aliasing in the PHT compared to simpler schemes like bimodal prediction, especially for correlated branches. Its speculative GHR update improves frontend responsiveness, while correction logic ensures architectural correctness after mispredictions. However, the XORing mechanism can still lead to destructive aliasing between unrelated branches with similar PC and history patterns. Moreover, managing the dual-port SRAM and valid-bit arrays introduces moderate design complexity, particularly in timing-critical stages of the fetch pipeline.

Performance Analysis: The GShare predictor achieved prediction accuracies of up to 99.9% accuracy (on compression.elf). Compared to a baseline not-taken, it reduced the branch misprediction rate by almost 90%, leading to a 15-20% improvement in IPC, particularly in branch-heavy workl oads. Speculative updates and prompt misprediction recovery contributed to smoother frontend operation and reduced branch resolution stalls. The design maintained high accuracy even under frequent control flow changes, validating its effectiveness in real-world workloads with mixed branching patterns.

*viii) **Sequential division**:* Another necessary optimization was within our division functional unit. Originally, we implemented division using a pipelined IP core, which at first seemed optimal due to its higher throughput. However, we found that the pipelined IP incurred a significant power cost, even when division was infrequent in typical workloads like coremark. This motivated us to replace it with a lower-power sequential divider.

Implementation: Most of the implementation consisted of reading documentation, understanding the meaning of signals, and following waveforms to assure correct functionality. We also updated our execute stage control logic to stall the pipeline while the divider was busy, ensuring correctness without introducing structural hazards.

Design Trade-offs: The primary trade-off was performance versus power. Sequential division has higher latency compared to the constant low-latency output of the pipelined IP. However, this is acceptable for our workloads, where division instructions are rare and typically not on the critical path. The massive power savings outweighed the occasional stall cycles.

Performance Analysis: When analyzing coremark, the switch to sequential division resulted in a dramatic reduction in power consumption. The pipelined divider consumed 408.769 mW, while the sequential divider reduced this to just 26.972 mW, a nearly 15× improvement. Despite the increased latency of individual divisions, the overall coremark IPC remained virtually unchanged, confirming the optimization had minimal performance impact for typical use cases. This switch to sequential division also had minimal IPC impact on benchmarks that used more divisions such as aes sha.

*ix) **Nonblocking DCache (Single Miss)**:* In a pipelined cache, any cache miss typically causes the entire pipeline to stall until the miss resolves. This prevents subsequent accesses from making progress, even if they are hits, and limits the throughput benefits of pipelining. To address this, we implemented a nonblocking DCache that can continue operation through one outstanding miss. This allows the pipeline to make forward progress on later accesses, as long as they hit in the cache.

Implementation: This behavior was enabled by adding a single MSHR (Miss Status Holding Register) to the design. When the cache detects a miss, it stashes the relevant metadata into the MSHR and allows the pipeline to continue operating. Any subsequent accesses that hit in the cache are processed normally. However, if another miss is encountered while the first is still unresolved, the pipeline stalls as usual. In effect, this provides limited nonblocking behavior where only a single outstanding miss is tolerated at a time.

Design Tradeoffs: The addition of one MSHR offers a balance between improved memory-level parallelism and low design complexity. It enables the cache to handle isolated misses more gracefully without introducing the complexity of a fully nonblocking design. However, it is still susceptible to stalls under higher miss rates since only one miss can be in flight. Extending this design to support multiple MSHRs would improve parallelism further but would also require more complex control logic and arbitration.

Performance Analysis: This feature led to mixed results in performance. While it reduced stalling in scenarios where a miss was followed by one or more hits, we actually observed a small decrease in IPC across benchmarks. We did not have enough time to investigate the root cause in detail, but a likely explanation is that the design may be introducing unwelcomed bubbles into the pipeline under certain conditions. These bubbles could be interfering with the normal flow of memory requests, offsetting the gains from reduced stalling. In addition, the integration of the NB cache into our design proved difficult and hence we were only able to get it functional across the coremark and aes sha benchmarks.

IV. Additional Discussion / Observations

During benchmarking, we performed a detailed analysis of control flow behavior across testbenches by tracking the frequency and impact of JALR (jump-and-link register) instructions. Unlike regular branches, JALR targets are computed dynamically and cannot be predicted using static or pattern-based predictors like GShare. We classified these as definite mispredictions, since the target address often remains unresolved until the instruction commits. In certain workloads—most notably aes_sha.elf—we observed that up to 2% of total instructions were JALRs. Despite deploying a fairly accurate GShare predictor (93% branch prediction accuracy on aes_sha.elf), such workloads showed minimal IPC improvement, indicating that JALR-induced pipeline flushes remained a dominant bottleneck. This led us to explore early branch recovery techniques as a potential solution, aiming to restore fetch more aggressively upon misprediction. However, our initial attempt at implementing this feature was unsuccessful due to a conceptual misunderstanding in how recovery interacts with speculative state and pipeline control. While not realized in this project phase, the challenge highlighted the need for careful coordination between frontend redirect logic and backend commit state in future iterations.

V. Contributions

a. Checkpoint 1

- Ahmed worked on implementing and verifying the cache line adapter and Fetch stage.
- Eddie worked on implementing and verifying the FIFO and Fetch stage.
- Mahir worked on implementing and verifying the Line Buffer and Fetch stage.

b. Checkpoint 2

- Ahmed worked on implementing and verifying the decode stage, execute stage (alu, mul, div functional units), cdb, and the random testbench.
- Eddie worked on implementing and verifying the dispatch stage, Free List, ROB, PRF, reservation stations, and the random testbench.
- Mahir worked on implementing and verifying the dispatch stage, RAT, RRF, ROB, and reservation stations.

c. Checkpoint 3

- Ahmed worked on implementing and verifying the memory units and LSQ.
- Eddie worked on implementing and verifying the control instructions.
- Mahir started working on the gshare branch predictor.

d. Advanced Features and Optimizations

- Ahmed converted the instruction cache to direct mapped from 4-way set-associative and implemented a pipelined data cache.
- Eddie implemented a next-line prefetcher, an improved cacheline adapter, and converted the division functional unit from pipelined to sequential.
- Mahir implemented the split LSQ, post-commit store buffer (with write coalescing), gshare branch predictor, and helped out with verification and debugging of the pipelined cache.

VI. Conclusion

Designing and implementing an out-of-order processor capable of supporting the full RV32IM instruction set offered a deep exploration into modern microarchitectural techniques. Starting from a baseline design focused on correctness, we successfully built a functionally accurate out-of-order core featuring register renaming, dynamic scheduling, and in-order retirement. Building upon this foundation, we systematically identified performance bottlenecks and introduced a series of targeted optimizations, including caching strategies, memory prefetching, a split load-store queue, branch prediction, and specialized functional unit handling.

Through rigorous testing and performance evaluation, we observed clear improvements in throughput, memory-level parallelism, and overall instruction-level parallelism. Each enhancement contributed incrementally to the robustness and efficiency of our processor, demonstrating the cumulative benefit of thoughtful architectural trade-offs.

This project not only reinforced our understanding of core concepts such as register renaming, reservation stations, and issue logic, but also challenged us to balance performance, complexity, and correctness. Our experience reflects the iterative nature of hardware design—where meaningful progress often comes from refining details and carefully integrating new ideas. Looking forward, future work could include support for out-of-order store execution, a superscalar front-end, early branch recovery, and much more.

Ultimately, this project gave us practical insight into the complexity of modern processors and a strong appreciation for the design decisions that drive high-performance CPU architecture.