A Superscalar, Out-of-Order RISC-V Processor

Vikram Rao
Electrical and Computer Engineering
University of Illinois Urbana-Champaign
Urbana, IL, USA
vikramr5@illinois.edu

Krishnan Shankar
Electrical and Computer Engineering
University of Illinois Urbana-Champaign
Urbana, IL, USA
ks128@illinois.edu

Aniketh Tarikonda
Electrical and Computer Engineering
University of Illinois Urbana-Champaign
Urbana, IL, USA
aniketh8@illinois.edu

***Abstract*—This report details the design and implementation of a superscalar, out-of-order RISC-V32IM processor with early branch resolution and a load-store queue. The processor was designed in SystemVerilog and synthesized using the Synopsys Design Compiler. Evaluation was performed on the CoreMark benchmark, and 10 other benchmark programs, to evaluate performance on various workloads. The processor achieved 0.646 instructions per cycle on CoreMark, with a clock frequency of 561.8 MHz and an area footprint of 296,030 µm². Detailed performance analysis and feature trade-offs are discussed, along with technical details of many of the processor's key features.**

I. Introduction

Instruction-level parallelism (ILP) is a key avenue for improving processor performance, and out-of-order execution is a powerful technique for exploiting this parallelism in the presence of control and memory hazards. In this project, we designed and implemented a superscalar, out-of-order RISC-V32IM processor. Our design includes features such as explicit register renaming (ERR), a reorder buffer (ROB), separate issue queues, a split load-store queue (LSQ), separate instruction and data caches, early branch resolution (EBR), a return address stack (RAS), and a GShare branch predictor. These features work together to improve performance while still allowing precise recovery on branch mispredictions and memory hazards.

Quantitatively, we aimed to achieve a clock frequency of at least 600 MHz and an area of no more than $300,000$ µm² after synthesis. These goals shaped most of our design decisions, as we had to balance the benefits of advanced features against their implementation complexity and area cost.

Rather than focusing on just one advanced feature, we attempted to build a balanced design that performs reasonably well across a variety of workloads. We used 64 physical registers and a 32-entry reorder buffer to expose enough parallelism without too much complexity. We also separated issue logic by functional unit, added lightweight branch checkpointing, and used conservative memory ordering in the load-store queue. Overall, the design tries to be aggressive where the benefit is clear, but simple enough to remain synthesizable and debuggable.

Development was mostly done using Git and GitHub, with heavy use of branches and a merge-based workflow. This allowed us to work on different advanced features in parallel, while maintaining a stable functional main branch.

In the following sections, we will describe the design and implementation of our processor, including chronological progress and technical details of key features. We will then evaluate the performance of our processor on a variety of benchmarks, analyze the impact of different design choices, and conclude with main takeaways and potential future directions.

II. Milestones

To aid in development and testing, we organized the main functional features of our processor into three milestones, which we refer to as Checkpoints 1, 2, and 3. Each checkpoint represents a significant stage in the design and implementation process, with specific features and testing goals.

A. Checkpoint 1

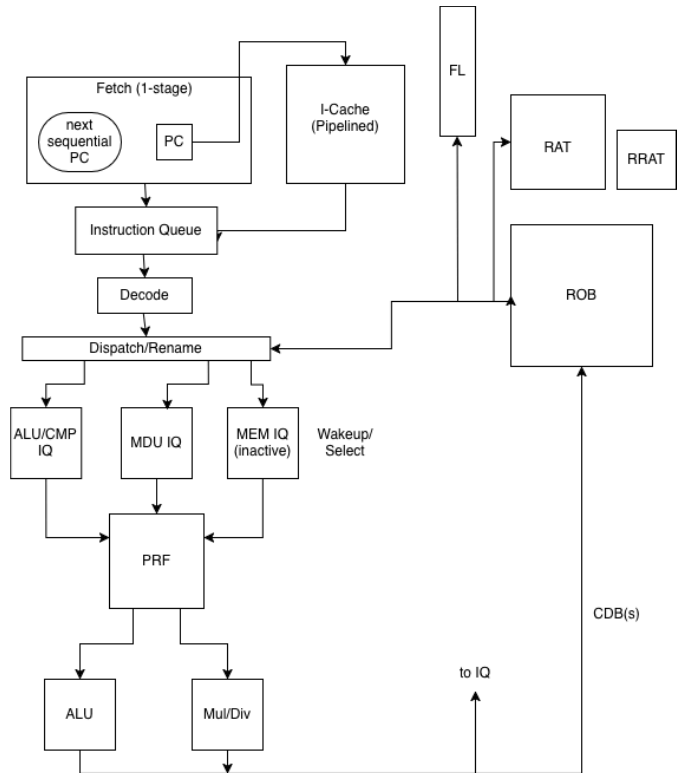

Fig. 1. Checkpoint 1 Design Diagram

By Checkpoint 1, we had the main out-of-order execution backbone assembled. The design already included fetch, rename, dispatch, issue queues, a physical register file, a RAT, a ROB, a free list, and separate ALU, multiply, and divide units. At this stage the frontend was still scalar, and later features like branch prediction and superscalar support were not yet implemented, but the basic rename-issue-commit pipeline was working. A diagram of the design at this point is shown in Fig. 1.

Testing at this point focused on correctness of the core control logic rather than performance. We used directed dependency and OoO tests, random test generation, and standalone testbenches for structures such as the FIFO and cacheline adapter. This let us verify that instructions could execute out of order while still retiring correctly through the ROB.

B. Checkpoint 2

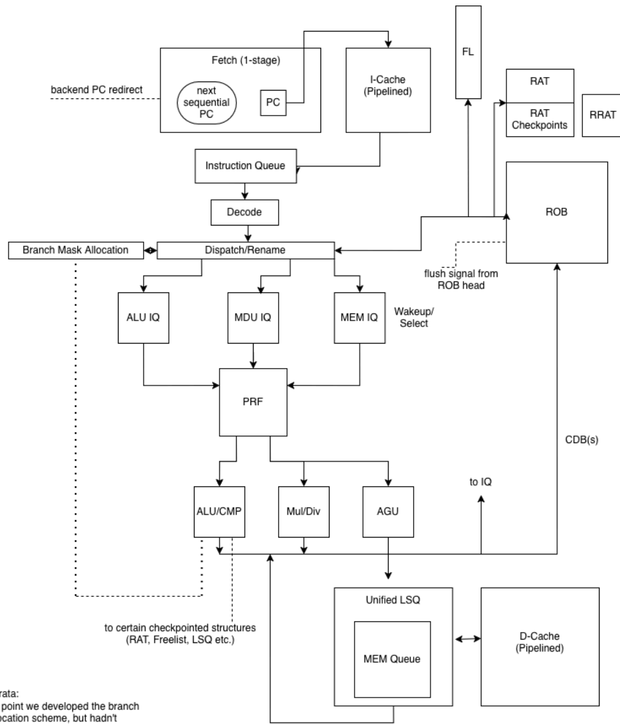

Fig. 2. Checkpoint 2 Design Diagram

Checkpoint 2 expanded the processor into a more complete base design by integrating the memory system and adding the first version of branch recovery support. Compared with Checkpoint 1, this version includes dedicated instruction and data caches, a connected load-store path, and branch-mask metadata for tracking speculative instructions. The frontend was still scalar and used simple static branch handling, but the pipeline now had the basic plumbing needed to recover from wrong-path execution. A diagram of the design at this point is shown in Fig. 2.

Verification also broadened in this checkpoint. In addition to the earlier dependency and random tests, we used memory- and jump-oriented programs. This was important because the main challenge in this checkpoint was no longer isolated arithmetic execution, but correct interaction between caches, loads, stores, and branch recovery.

C. Checkpoint 3

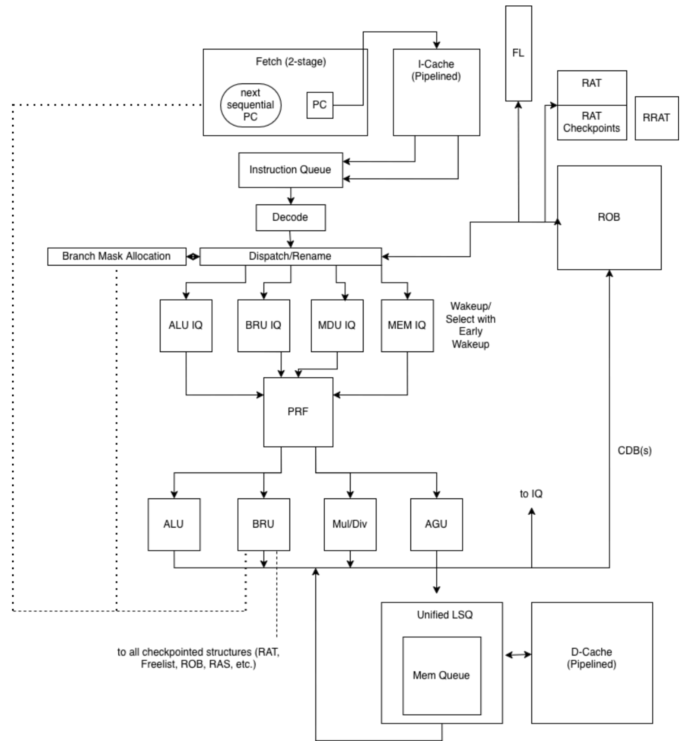

Fig. 3. Checkpoint 3 Design Diagram

Checkpoint 3 represents the completed base out-of-order processor before we added advanced features. By this point, the design included a working scalar frontend, separate issue queues for ALU, multiply, divide, and memory operations, integrated load-store execution, connected instruction and data caches, and branch recovery support throughout the backend. In addition, early branch resolution was fully implemented and tested, but not yet integrated with some parts of the design (notably the fetch stage). A diagram of the design at this point is shown in Fig. 3.

The emphasis of this checkpoint was integration and stability. Most of the architectural blocks were already present, but they had to be connected together and made to work as a single system. This involved a lot of debugging and testing to ensure that instructions could flow through the pipeline correctly, that the caches worked with the load-store queue, and that branch recovery was functioning properly. This checkpoint served as the baseline for our final design, and it was important to have a stable and correct out-of-order core before we added more complex features like superscalar and early branch resolution.

D. Final Design

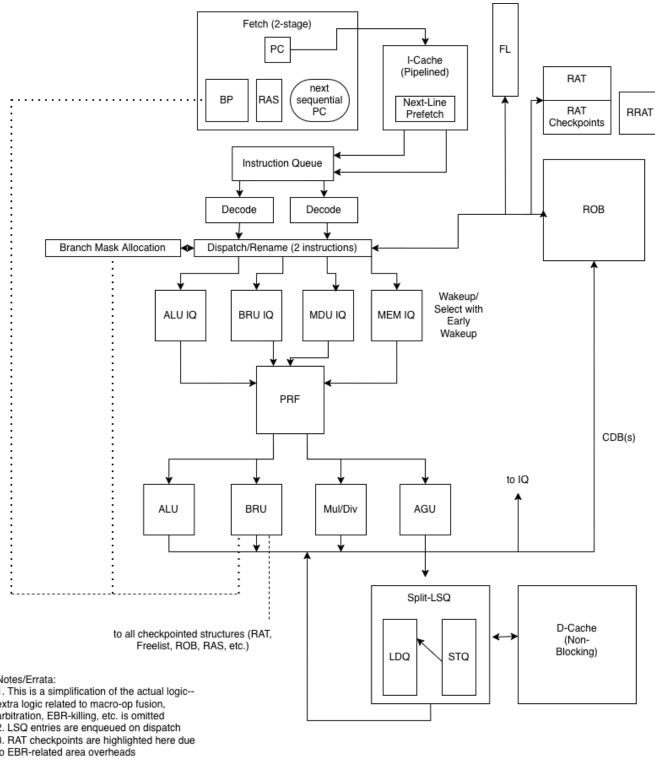

Fig. 4. Final Design Diagram

The final design builds on the Checkpoint 3 base by adding several advanced features. Most notably, we extended the scalar frontend into a 2-way superscalar fetch/dispatch/commit design, added a GShare branch predictor and return address stack, integrated early branch resolution with checkpoint-based recovery, and replaced the earlier memory path with a split load-store queue that supports store-to-load forwarding. We also refined the cache hierarchy with instruction cache prefetch and a non-blocking data cache so that it better matched the needs of the out-of-order backend. A diagram of the final design is shown in Fig. 4, and the following section describes these features in more detail.

III. Technical Details

A. Out-of-Order Execution

The core uses a rename-issue-execute-commit organization adapted for two-wide dispatch. Architectural registers are mapped onto 64 physical registers through a RAT, and new destinations are allocated from a checkpointed free list. A separate RRAT tracks committed architectural state so that old physical mappings can be freed in order at commit time.

Up to two renamed instructions can be inserted into the backend each cycle, subject to available space in the ROB, free list, issue queues, and LSQ. The ROB contains 32 entries and provides in-order retirement and a single place to merge completion information from the functional units. It can commit up to two instructions per cycle, although stores still wait for explicit acknowledgment from the LSU so that memory remains in order at retirement.

Instruction scheduling is distributed across separate issue queues for ALU, multiply/divide, branch, and memory operations. Source readiness is tracked through a ready list and updated by common-data-bus broadcasts from the four functional units. The ALU and branch unit also provide fast wakeup signals for newly produced destinations. This is a relatively simple approach compared to a large centralized scheduler, but it was enough to support useful out-of-order execution in our design.

B. Branch Prediction

The fetch stages use a GShare branch predictor with a 9-bit global history register (GHR) and a 9-bit pattern history table (PHT) index. The PHT is stored in static RAM (SRAM), and contains a 2-bit saturating counter for each entry. The SRAM is gated with a flip-flop array to store validity bits, so that the predictor can distinguish between initialized and uninitialized entries.

Once a prediction is made, the prediction data is attached to the instruction bundle as it flows through the pipeline. This eventually reaches the branch unit, where the branch is resolved and the prediction is checked. The prediction data along with the branch result are broadcast on the common data bus (CDB), allowing the branch predictor to update itself based on the actual branch result.

Branch direction prediction is intentionally limited to one control instruction per fetch group. In the second fetch stage, conditional branches and jalr instructions are treated as serializing controls, so they issue alone within the bundle. Having only a single branch prediction per cycle simplifies the predictor indexing and SRAM design significantly, while still allowing the processor to achieve good performance on a wide range of benchmarks. However, as a trade-off, it does reduce fetch bandwidth on branch-heavy code, since the second instruction in the bundle will be treated as a NOP if the first instruction is a control.

The predictor is complemented by a few targeted control-flow mechanisms. For example, the RAS has two entries and is updated speculatively in fetch, with checkpoints used so that it can be restored correctly after a misprediction. Targets for jal instructions are computed directly in the frontend, ret instructions are predicted through a small return address stack, and jalr targets benefit from a simple AUIPC-jalr fusion path in some cases.

C. Early Branch Resolution (EBR)

Early branch resolution is handled by a dedicated branch unit, branch allocation logic within the dispatch stage, and a checkpointing scheme for speculative structures. Branches and jalr instructions are dispatched into a separate branch issue queue and execute in a dedicated branch unit rather than sharing the normal ALU path. Once their operands are ready, they are eligible for execution. After the branch direction (or jump target) has been resolved, the results are broadcasted to the entire processor.

To support fast recovery, each unresolved branch is assigned one bit in an N-bit parametrizable branch-mask space (through profiling, we determined that a 4-bit branch mask was sufficient for our design). That mask is attached to renamed instructions, issue-queue entries, LSQ entries, and memory requests. Dispatch also checkpoints the RAT, free-list read pointer, ROB tail pointer, LSQ allocation state, predictor history, and RAS top for each live unresolved branch. On a correct prediction, the corresponding bit is cleared. On a misprediction, younger instructions carrying that bit are invalidated and the checkpointed state is restored.

This mechanism paired well with our superscalar implementation; superscalar processors tend to have higher issue queue occupancy, thus increasing branch misprediction penalties significantly. EBR allows for more 'surgical' recovery from branch mispredictions, reducing back-end flush penalties significantly. Furthermore, older branches have the opportunity to invalidate all newer branches which fall on a wrong path, without having to wait until the instruction reaches the head of the ROB.

Despite these benefits, we found that our EBR implementation had three glaring trade-offs:

1) The branch allocation logic introduced a new structural hazard–the processor can only support a limited number of speculative branches at a given time. Thus, certain assembly patterns (e.g. repeated nested branching with operands coming from load/store operations) would stress the branch allocator.
2) EBR has a non-negligible area and power overhead. Structures like the RAT require full copies when snapshotted–with 32 architectural-physical mappings, 64 physical registers, and 4 supported speculative branches, the checkpoints can take up significant area. Furthermore, the associative logic required to broadcast branch masks and invalidate instructions across the backend in parallel causes high switching activity, leading to increased dynamic power consumption.
3) On benchmarks with highly-predictable branches (e.g. FFT, Compression), we observed a near 99% successful branch prediction rate. In these situations, the pipeline rarely encounters the misprediction penalties that EBR is designed to mitigate. Consequently, the performance gains become negligible.

D. Load-Store Queue (LSQ)

Memory ordering is managed by a split load-store unit with separate 8-entry load and store queues. Dispatch allocates queue indices before execution so that memory instructions can be tracked independently of when they issue. After issue, the LSU performs address generation, computes byte masks for subword accesses, and records the metadata needed for forwarding or cache access. Stores enter the store queue speculatively, but they are not allowed to update memory until the ROB commits them.

Loads are checked against older stores before they are allowed to access the data cache. If an older store already has a known address and overlapping bytes, the LSU forwards the matching data directly from the store queue. Partial forwarding is supported through byte masks.

The LSU also tracks outstanding memory requests explicitly. Loads and stores are assigned request identifiers, tracked through a small request table, and carried through branch recovery using the same branch-mask mechanism used elsewhere in the core. Wrong-path requests are filtered on a flush, and stores only acknowledge commit after the cache-side transaction has completed. This makes the memory system one of the more complex parts of the design, but it is necessary for combining out-of-order execution with correct in-order retirement.

E. Cache Design

The instruction cache is a 2-way set-associative cache with 256-bit lines. It can return up to two consecutive 32-bit instructions per access and reports whether the second word is in the same line, which helps the frontend build a two-instruction bundle. Replacement is tracked with a 1-bit per-set policy, and the cache also includes a small next-line prefetch mechanism.

The prefetcher is intentionally simple. After a demand miss is serviced, the I-cache records the next sequential cache line as a pending prefetch target. When the memory port is idle, it fetches that line into a single-entry prefetch buffer. If a later demand miss hits this buffer, the cache installs the prefetched line directly instead of going back to memory. This keeps the mechanism small and low risk, while still helping on straight-line instruction streams where the frontend is likely to request the next line soon afterward.

The data cache is more aggressive because it must work with the out-of-order LSU. It is a 4-way set-associative cache with 256-bit lines, per-line valid and dirty tracking, and pseudo-LRU replacement. It is non-blocking, using MSHRs to track misses and merge requests to the same line. If multiple CPU requests target a line that is already missing, the cache attaches them to the same outstanding miss instead of sending duplicate memory traffic. Dirty evictions are handled through a small writeback buffer, and when the fill returns, the cache writes the line into the array, merges in any buffered store bytes for that line, and then drains responses back to the waiting requests using their stored request IDs. This is more complex than the instruction cache, but it helps prevent a single miss from stalling the whole processor.

Below the caches, a non-blocking cacheline adapter arbitrates traffic onto the DRAM interface. It gives the instruction cache one outstanding line request at a time, allows the data side to queue tagged requests, and serializes each 256 -bit writeback into four 64-bit DRAM beats. The D-cache side of the adapter also keeps a small table of outstanding line requests so that returned DRAM data can be routed back using the correct ID. Together, these structures form a memory hierarchy that is much better suited to an out-of-order backend than a single blocking cache.

F. *Superscalar*

The processor is superscalar in fetch, dispatch, and commit, with a maximum width of two instructions per cycle. The second fetch stage constructs a two-slot bundle from the instruction cache response, handles misaligned PCs, and tracks per-slot validity. Decode and dispatch preserve this two-lane structure as long as downstream structures have capacity. The ROB, free list, RAT writes, and LSQ allocation logic all support up to two instructions per cycle.

In the backend, superscalar behavior comes from parallelism across specialized structures rather than a single large scheduler. Dispatch can place different instructions into the ALU, multiply/divide, branch, and memory issue queues in the same cycle, and these functional units can complete independently onto the common data bus. This gives the processor useful overlap without the timing and area cost of a much wider monolithic backend.

The superscalar design is intentionally asymmetric. Straight-line integer code benefits the most, since both fetch bandwidth and backend parallelism are high when there are few control or memory hazards. Branches and jalr instructions are still serialized in the frontend, and memory operations are still limited by conservative dependency checks in the LSU. These choices reduce peak width in some cases, but they were reasonable trade-offs for our design goals.

IV. Results

A. *Final Performance*

Table I details the final performance of our processor on all 11 benchmark programs. Qualitative performance is evaluated based on the achieved IPC relative to the average IPC among other out-of-order RISC-V32IM processors. The first six benchmarks were used for evaluation during development (validation set), while the last five were hidden during development (test set). This allows us to evaluate the generalization of our design to unseen workloads, preventing overfitting to certain types of programs.

TABLE I
Achieved IPC on Benchmarks

Benchmark	IPC	Qualitative Performance
CoreMark	0.646	Average
AES-SHA	0.410	Below Average
Compression	0.889	Average
FFT	0.768	Average
Image	0.508	Average
Mergesort	0.710	Above Average
CNN	0.476	Below Average
Graph	0.630	Above Average
Physics	0.445	Abysmal
Raytracing	0.925	Excellent
Sudoku	0.609	Excellent

Additionally, we evaluate the performance of our processor in terms of clock frequency and area. The processor achieved a clock frequency of 561.8 MHz, which is below our target of 600 MHz, but is still competitive with other out-of-order RISC-V32IM processors. The area footprint of our processor is 296,030 μm², which meets our target of 300,000 μm². This is shown in Table II.

TABLE II
Synthesis Results

Metric	Target	Achieved
Clock Frequency	≥ 600 MHz	561.8 MHz
Area	$\leq 300,000$ μm²	$296,030$ μm²

Synopsys Design Compiler reports the design's worst negative slack (WNS) as 0.000130 ns, indicating that the design meets timing requirements at the achieved clock frequency. The critical path is through the load-store queue and data cache.

The area breakdown of the processor is shown in Table III.

TABLE III
Area Breakdown

Component	Area (μm²)	Percent
Fetch Logic	5,490	1.9%
Cache/Memory	129,790	43.9%
↳ Instruction Cache	38,249	12.9%
↳ Data Cache	84,197	28.4%
↳ Cacheline Adapter	7,344	2.5%
Out-of-Order Structures	53,367	18.0%
↳ Physical Register File	37,653	12.7%
↳ Register Alias Table	9,053	3.1%
↳ Reorder Buffer	6,661	2.2%
Issue Queues	34,672	11.8%
↳ ALU	10,959	3.7%
↳ Branch	8,454	2.9%
↳ Multiply/Divide	4,981	1.7%
↳ Memory	10,278	3.5%
Execution	39,735	13.4%
↳ Load-Store Unit	22,485	7.6%
↳ Multiply/Divide Unit	6,254	2.1%
↳ Other Execution Logic	10,996	3.7%
Other	32,976	11.1%

B. *Comparisons*

To further evaluate our processor, we selectively alter and/or disable certain features of our design and report the resulting performance. This allows us to analyze the contribution of each feature to overall performance, as well as to identify potential bottlenecks in our design.

a) *Size of Physical Register File:*

We evaluated the impact of the size of the physical register file on performance by testing three configurations: 48, 64 (final design), and 96 physical registers. This also affects the size of the free list, which is always equal to the number of physical registers minus 32. The results are shown in Table IV.

TABLE IV
Impact of Number of Physical Registers on Performance

Statistic	48 Regs	64 Regs	96 Regs
Total Area (μm²)	286, 191	296, 030	324, 505
CoreMark IPC	0.622	0.646	0.646
AES-SHA IPC	0.398	0.410	0.410
Compression IPC	0.864	0.889	0.889
FFT IPC	0.687	0.768	0.768
Image IPC	0.471	0.508	0.508
Mergesort IPC	0.674	0.710	0.710

More fine-grained testing would be helpful in further analyzing this design trade-off. However, from this data, we can see that going above 64 physical registers does not provide a significant performance boost, while going down to 48 physical registers results in a noticeable performance degradation. This suggests that the optimal number of physical registers for our design is at or slightly below 64.

b) *Size of Reorder Buffer:*

We also evaluated the impact of the size of the reorder buffer on performance by testing three configurations: 16, 32 (final design), and 64 entries. The results are shown in Table V.

TABLE V
Impact of Reorder Buffer Size on Performance

Statistic	16 Entries	32 Entries	64 Entries
Total Area (μm²)	293, 522	296, 030	306, 656
CoreMark IPC	0.606	0.646	0.648
AES-SHA IPC	0.394	0.410	0.410
Compression IPC	0.829	0.889	0.880
FFT IPC	0.678	0.768	0.768
Image IPC	0.427	0.508	0.532
Mergesort IPC	0.656	0.710	0.717

These results show a more complex relationship. While going down to 16 entries results in a noticeable performance degradation across all benchmarks, going up to 64 entries only provides significant performance boosts on the Image and Mergesort benchmarks, while providing little to no performance boost on the others. This suggests that the optimal size of the reorder buffer for our design is around 32 entries, as this provides a good balance between performance and area.

c) *Branch Predictor:*

We evaluated the impact of the branch predictor on performance by comparing the final design with a configuration that has no branch predictor. In this setup, branches are predicted using a static not-taken predictor. The results are shown in Table VI.

TABLE VI
Impact of Branch Predictor on Performance

Statistic	Default	Without BP	Percent Change
Total Area (μm²)	296, 030	291, 311	−1.59%
CoreMark IPC	0.646	0.427	−33.90%
AES-SHA IPC	0.410	0.397	−3.17%
Compression IPC	0.889	0.788	−11.36%
FFT IPC	0.768	0.749	−2.47%
Image IPC	0.508	0.343	−32.48%
Mergesort IPC	0.710	0.673	−5.21%

This highlights the importance of the branch predictor in our design. Some benchmarks (like CoreMark and Image) benefit heavily from the branch predictor, while others (like AES-SHA and FFT) see only slight performance boosts.

d) *Load-Store Queue Forwarding:*

To test the effectiveness of load-store queue forwarding in our design, we compared the final design with a configuration that has no load-store queue forwarding. This means that loads must wait for all prior stores to commit before they can begin execution. The results are shown in Table VII.

TABLE VII
Impact of Load-Store Queue Forwarding on Performance

Statistic	Default	Without LSQ Forwarding	Percent Change
Total Area (μm²)	296, 030	288, 292	−2.61%
CoreMark IPC	0.646	0.602	−6.81%
AES-SHA IPC	0.410	0.364	−11.22%
Compression IPC	0.889	0.630	−29.13%
FFT IPC	0.768	0.672	−12.50%
Image IPC	0.508	0.505	−0.59%
Mergesort IPC	0.710	0.616	−13.24%

Similar to the branch predictor, load-store queue forwarding has a varying impact on different benchmarks. However, unlike the branch predictor, the actual benchmarks that benefit the most from load-store queue forwarding are different. For example, Image sees significant performance degradation without a branch predictor, but sees almost no performance change without load-store queue forwarding. On the other hand, Compression and Mergesort see moderate performance degradation without load-store queue forwarding, while seeing significantly less performance degradation when the branch predictor is disabled instead. This highlights the importance of using a variety of benchmarks to

evaluate processor performance, as different features may have varying impacts on different workloads.

e) *Return Address Stack (RAS):*

The importance of the return address stack (RAS) was tested by comparing the final design with a configuration that has no RAS. In this setup, return instructions are predicted using a static not-taken predictor. The results are shown in Table VIII.

TABLE VIII
Impact of Return Address Stack on Performance

Statistic	Default	Without RAS	Percent Change
CoreMark IPC	0.646	0.634	−1.86%
AES-SHA IPC	0.410	0.375	−8.54%
Compression IPC	0.889	0.889	0.00%
FFT IPC	0.768	0.763	−0.65%
Image IPC	0.508	0.450	−11.42%
Mergesort IPC	0.710	0.701	−1.27%

These results show that the return address stack has a relatively small impact on performance, compared to larger features like the branch predictor and load-store queue forwarding. However, it still provides noticeable performance boosts on certain benchmarks, particularly AES-SHA and Image. Notably, Image benefits significantly from the branch predictor as well, while AES-SHA does not, suggesting that the RAS does fulfill a unique niche in our design that is not fully covered by the branch predictor.

f) *Superscalar:*

To test the impact of the superscalar design of our processor, we compared the final design with two configurations: one with no superscalar dispatch (i.e., only one instruction can be dispatched per cycle) and one with no superscalar commit (i.e., only one instruction can commit per cycle). The results are shown in Table IX and Table X, respectively.

TABLE IX
Impact of Superscalar Dispatch on Performance

Statistic	Default	Without Superscalar Dispatch	Percent Change
CoreMark IPC	0.646	0.620	−4.02%
AES-SHA IPC	0.410	0.408	−0.49%
Compression IPC	0.889	0.864	−2.81%
FFT IPC	0.768	0.739	−3.78%
Image IPC	0.508	0.501	−1.38%
Mergesort IPC	0.710	0.661	−6.90%

TABLE X
Impact of Superscalar Commit on Performance

Statistic	Default	Without Superscalar Commit	Percent Change
CoreMark IPC	0.646	0.627	−2.94%
AES-SHA IPC	0.410	0.409	−0.24%
Compression IPC	0.889	0.714	−19.69%
FFT IPC	0.768	0.670	−12.76%
Image IPC	0.508	0.504	−0.79%
Mergesort IPC	0.710	0.661	−6.90%

These results again show a varying impact of different parts of the superscalar design on different benchmarks. Compression and FFT see significant benefits from superscalar commit, while CoreMark finds superscalar fetch/dispatch more beneficial. This suggests that both scalar fetch/dispatch and scalar commit can be bottlenecks depending on the workload, and that a balanced superscalar design is important for achieving good performance across a variety of benchmarks.

Additionally, we noticed that the performance boost from our superscalar design is not as high as we expected, especially considering that our 2-way superscalar design should ideally provide up to a 2x performance boost. This suggests that there may be bottlenecks in our design that are preventing us from fully utilizing the benefits. For example, our design may be bottlenecked by execution resources or memory bandwidth, which would significantly reduce the speedup from a superscalar design. Further analysis and testing would be needed to identify and address these bottlenecks for an improved superscalar design.

V. Conclusion

Overall, we successfully accomplished our goal of building a performant out-of-order RISC-V32IM processor. The final design reached a frequency of 561.8 MHz and an area of $296,030\ \mu m^2$, which meets our area target and comes reasonably close to our original frequency goal. The processor achieves competitive performance on a variety of benchmarks, with an IPC of 0.646 on CoreMark and varying performance across other workloads.

Several parts of the design were especially important to its final performance. Lightweight branch checkpointing, early branch resolution, and the GShare branch predictor all contributed to improving control flow performance, while the split load-store queue and non-blocking data cache helped improve memory-level parallelism. Overall, the design strikes a good balance that allows it to adapt to a variety of workloads effectively. We believe that given a larger area budget, our design easily scales up to provide even better performance on a wider range of benchmarks, making it an extensible foundation for future improvements and optimizations.