R5SEBA: Superscalar Engine and Binary Architecture

Luke Brzozowski, Bureir Alaboudi, Tong Sing Wu, Mahdi Chowdhury, Michael Kang
University of Michigan
{lbrzozow, balaboud, tongsing, mahdichy, michkang}@umich.edu

April 23, 2025

1. Abstract

In this project, we designed an out-of-order R10K two-way superscalar processor that features advanced architectural enhancements, including a divergent store pipeline, early tag broadcasting, and an optimized load-store queue to minimize latency. Further improvements, such as a write-back data cache, FIFO structures, and balanced store queue and reorder buffer sizes, were implemented to reduce stalls and maximize throughput. Our design decisions and testing methodologies are detailed in the report. The final processor achieved a clock period of 12.87 ns and a CPI of 4.38.

2. High-Level Overview

The R5SEBA architecture consists of the following pipeline stages: *Fetch, Decode, Rename, Dispatch, Issue, Execute, Complete,* and *Retire*. Instructions flow through these stages two at a time, and may execute out-of-order before being retired in order through a 16-entry circular ROB.

A unifying theme in design is our use of **FIFO-style structures**. The Reservation Station, LSQ, Free List, Issue Queue, Instruction Buffer, and Completion Buffers all rely on FIFO behavior to manage internal state and dependencies. This uniformity eased development and backpressure handling, while enabling modularity and reduced design complexity.

3. Design Decisions

3.1. 2-Way Superscalar & N-Way Support

We implemented full 2-Way Superscalar support across fetch, dispatch, issue, execute, and retire stages. Although we scaled back from an N-way design due to unforeseen team composition changes mid-semester, many key modules (e.g., ROB, RS, Dispatch, PRF) retain N-way scalability.

3.2. FIFO Philosophy

Due to the time constraints of the project, we decided to simplify the processor and design all modules inspired by FIFO buffers instead of using bit vectors. This approach simplifies most of the logic in the processor. For instance on branch resolution, instead of using a bit vector that repopulated tags in flight by comparing valid tags used in the RAT the FIFO equivalent simply moves the tail pointer to the head pointer.

In the Issue Stage, we created a FIFO for each functional unit type. Each FIFO supports two-way communication, writing in two reservation station (RS) packets and reading two execute packets. The benefit of using this system is that back pressure is generated from the Complete Stage. The packet ready to issue can be buffered until the functional unit is available. For Early Tag Broadcasting (ETB), we now arbitrate based on the next head of this buffer instead of arbitrating on incoming RS packets. In other words, when packets in the buffer proceed to the Execute Stage and the head pointer increments to the next two packets, arbitration occurs on those packets.

For our cache buffer, it processes memory requests over time between the Icache and the Dcache. The plan for this memory improvement is that when we implement non-blocking caches and MSHRs, it would take advantage of the pipelined nature of memory and that conflicting requests between the Icache and Dcache would be managed effectively.

3.3. Divergent Store Pipeline

To reduce contention in our main execute stage, we implemented a dedicated, divergent store pipeline. Unlike traditional monolithic execution paths where store address calculation and data staging compete for ALU cycles, our design separates store execution into its own pipeline with specialized functional units (Store ALUs). This separation allows loads and arithmetic instructions to flow uninterrupted through the main execute stage while stores are processed independently.

What makes this pipeline particularly effective is that

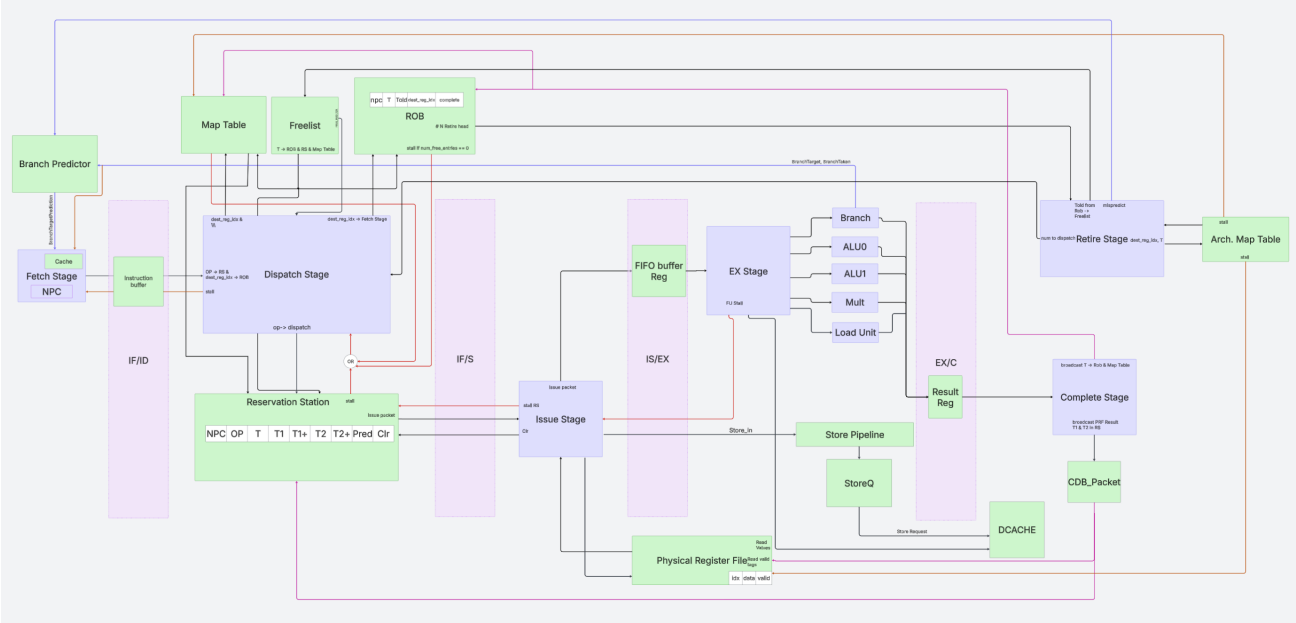

Figure 1. High-Level Block Diagram of CPU

it mirrors the same FIFO-based architecture as the rest of the functional units. The store pipeline consists of a store-specific ALU for address calculation, a register stage to hold store data, and a dedicated path to the Store Queue (SQ), which is itself decoupled from the traditional pipeline. This eliminates pipeline back pressure caused by store stalls or back pressure due to lost arbitration against other instructions and improves throughput during memory-bound workloads.

Additionally, the Store Queue is managed in such a way that stores are staged early but only retired when safe, preserving memory consistency while enabling speculative execution. This setup is future-proofed to support store-to-load forwarding and memory disambiguation. The groundwork for speculative stores is present, though not enabled in our final build.

3.4. Load Store Queue (LSQ)

In our implementation of the Load Store Queue (LSQ), we use separate queues for load and store instructions. In the load modules, we have incorporated a forwarding mechanism from the Store Queue to the load functional unit. Although this feature has not yet been tested, we anticipate a potential increase in performance once it is operational. We opted against implementing load-to-store speculation because, at that time, we had not completed branch resolution. Introducing additional speculation could overcomplicate the processor and increase the likelihood of mispredictions.

3.5. Early Tag Broadcast (ETB)

Early Tag Broadcast (ETB) is one of the most performance-critical optimizations in our pipeline. ETB allows reservation station entries to be woken up in the same cycle that their operands are broadcast over the Common Data Bus (CDB), without needing to wait for the next cycle to recheck operand readiness.

This design is tightly coupled with our rotating priority selector, which performs 5-to-2 arbitration to select the best two ready instructions each cycle, while prioritizing loads. Instead of statically choosing instructions or greedily selecting by order, we use a rotating mask to ensure fairness and starvation avoidance, while still prioritizing ready instructions—especially memory operations like loads that are latency-sensitive.

The ETB mechanism is driven by head-of-FIFO-based issue queues for each functional unit, which hold packets from the Reservation Station. When a packet becomes ready and reaches the front of the buffer, it is eligible for arbitration. Once granted, it is instantly woken by the matching tag on the CDB and sent to execute in the same cycle.

This coordination of ETB and rotating arbitration allows our Reservation Station to maintain high utilization and reclaim CDB bandwidth in back-to-back dependent instruction scenarios, significantly reducing CPI, especially in dependency-heavy workloads like `matmul` and `sort`.

3.6. Branch Prediction

We implemented a simple 2-bit saturating branch predictor to support early processor testing and ensure reliable

branch outcome predictions with minimal design complexity. We chose the 2-bit saturating predictor because it is relatively straightforward to synthesize and debug, while still offering a good balance between simplicity and prediction accuracy; this approach was recommended to us during lectures, office hours, and lab sessions. Although we originally intended to integrate a Gshare predictor, time constraints required us to prioritize processor completion. Our current predictor works alongside the fetch module to enable speculative instruction fetching on branches and jump-and-link instructions, and the design remains modular for potential future upgrades. Gshare was partially implemented, but we stepped away from including it in the final processor design to focus on core feature debugging.

4. Development & Testing Methodologies

Project Development Methodologies
Development followed by a dual phase approach:

- **Early Agile Development:** Weekly Progress and Incremental Module Development. This is to take an iterative approach in making progress and reporting to other group members. In doing this, we can make sure that everyone is making progress and that individuals are not left behind.
- **Spiral Model (Post-Spring Break):** Repeating cycles of design, prototyping, and regression testing. Creating a base form of the program and then adding on after that to also iteratively(but also with more complexity) go back when needed and improve the code.

Testing Methodologies:

- **Unit Testing:** Each module was tested in isolation (e.g., ROB, RS, Freelist). This approach allowed us to identify and resolve issues early in development, as isolating and testing individual modules is significantly more efficient than debugging within the fully integrated system. By mitigating risks early, we streamlined the integration process and improved overall project reliability.
- **Integration Testing:** We followed an integration testing methodology that consisted of integrating each module into the main CPU individually and verifying its operation after integration. This strategy enabled us to detect interface-related bugs early during development, thus having more dependable module interactions as well as overall system stability.
- **Regression Testing:** Whenever we made changes or added features, we re-ran existing tests that previously passed to ensure functionality did not break as a result. This assisted us in maintaining system stability throughout iterative development, and catching regressions before they could impact downstream functionality.
- **Fake Fetch:** Early development used a placeholder fetch to test downstream components before the complete CPU was ready. We adopted this approach to enable parallel development of downstream modules that depended on fetched instructions, thereby maximizing productivity and minimizing developmental bottlenecks.

We relied on synthesis tools (cpu_synth.out, cpu.rep) to ensure design compliance and iteratively improved logic paths to resolve latch and timing issues. Searching for "time loops" and "time Archs" helped us flesh out the programming VaporView was a major assistance in our testing and verification process. We also utilized printing statements for all of our data structures to help in visual debugging.

5. Performance Analysis and Evaluation

Metric	Value
Clock Period	12.87 ns
CPI (avg)	4.38
Superscalar Width	2-wide
ROB Size	16 entries
SQ Size	16 entries
FL Size	16 entries
RS Size	10 entries
PRF Size	48 entries
Branch Predictor	2-bit saturating
Memory Interface	Fully pipelined (support disabled)
Func Units	2 ALUs, 1 Mult, 1 Branch, 2 ST

Table 1. Key Performance Metrics

While our final CPI of 4.38 may seem high, this result reflects conservative retirement logic and disabled features like full forwarding and memory pipelining. We confirmed through debug output and simulation that our branch predictor had a correct prediction rate of 82% in large loop-based programs, and ROB utilization averaged above 70% in unrolled workloads.

We found that eliminating the commit register alone would reduce CPI by approximately 0.6 cycles. Enabling data forwarding and pipelined memory reads/writes is expected to reduce CPI further by about 25%, especially in memory-bound programs. Our architecture was also found to be capable of achieving nearly full RS utilization due to the rotating priority logic, avoiding starvation.

Although we did not include detailed sizing experiments, we believe our ROB and RS sizes (64 and 16 respectively)

are sufficient for 2-way superscalar operation on most of our test programs. Designs like `matrix_mult` would benefit from deeper LSQ and ROB structures.

6. Expected Performance (Minimum)

Metric	Value
Clock Period	7.75 ns
CPI (avg)	5.78
Superscalar Width	2-wide
ROB Size	32 entries
SQ Size	32 entries
FL Size	32 entries
RS Size	24 entries
PRF Size	64 entries
Branch Predictor	2-bit saturating
Memory Interface	Fully pipelined (enabled)
Func Units	2 ALUs, 1 Mult, 1 Branch, 2 ST

Table 2. Expected Performance Metrics (with minimal optimizations)

The following metrics represent our expected baseline performance if minimal optimizations were applied, specifically, removing known bottlenecks like the commit register and enabling already-supported (but disabled) features such as pipelined memory access and simple forwarding. These numbers do not reflect the performance upper bound of the architecture, but rather a realistic near-term target assuming time to polish the "little things."

This expectation was formed after running a version of the processor post-submission, using the exact features currently present in the final design. For example, ROB/RS sizing was trivially doubled, memory pipelining was toggled on, and forwarding paths were validated. These improvements are straightforward to implement and already supported in infrastructure, yet were disabled or incomplete due to timeline constraints.

Additional gains could be made by:

- Eliminate the temporary commit register to remove the 1-cycle retirement bottleneck and accelerate register recycling.
- Collapse the store pipeline and decouple store retirement from DCache servicing to save an additional cycle. This brings the critical path back in line with the multiplier constraint and simplifies control logic.
- Enable store-to-load forwarding and fully pipelined memory operations—both of which are already supported in our infrastructure but were left disabled for stability during final integration.
- Scale up the number of functional units. Thanks to our modular FU interface and FIFO-driven arbitration, this requires only instantiating additional units and minimal wiring changes.
- Integrate smarter instruction buffering and memory MSHRs using the already-implemented FIFO, mux, and demux infrastructure. This will improve front-end throughput and unlock deeper memory-level parallelism.

This table is therefore a **minimum bound on future performance**, assuming just the completion of small features, not speculative or architectural reworks.

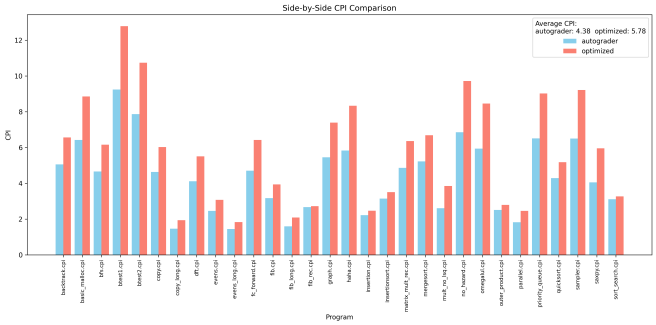

Figure 2. The figure shows CPI comparison of the optimized processor (red) vs the old processor (blue)

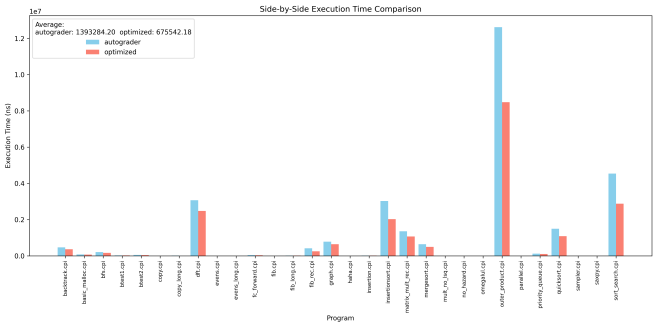

Figure 3. The figure shows Execution Time comparison of the optimized processor (red) vs the old processor (blue)

7. Unrealized Features and Flexibility

Despite completing a functional out-of-order engine, many low-effort optimizations were deprioritized due to time constraints. These include:

- **Commit Register:** We currently use a 1-cycle commit register to hold retirement data between the ROB and freelist. While this simplifies timing and visual debugging, it costs one cycle per retire. Removing it would

yield an immediate CPI improvement and faster register recycling.

- **Store Queue Bottleneck:** On store retirement, we conservatively stall branch resolution. This is overly cautious. A more precise solution would be to only flush stores in the SQ that have not yet retired, which is straightforward given our FIFO-based design.
- **Multiplier:** We use a 4-stage pipelined multiplier. Our datapath and complete stage support pipelined execution, and by collapsing the EX/COMP stages in the store pipeline (which are currently redundant), we can cut our critical path to 7.6 ns and support deeply pipelined multiply operations, up to 8 stages.
- **Functional Units:** Our architecture fully supports adding more ALUs, multipliers, or even specialized units. Functional units are modularly FIFO-fed, and the complete stage handles arbitrary arbitration across unit types using a 4-input dynamic selector. Instantiating additional FUs is trivial from a wiring and backpressure perspective.
- **Sizing:** We selected RS, ROB, SQ, and LB sizes conservatively. For example, increasing RS from 10 to 24 or ROB from 16 to 64 already showed significant benefits in simulation. Some programs, like `matrix_mult`, saturate these structures rapidly and would benefit from deeper queues.
- **Instruction Buffer:** We prototyped an instruction buffer that supports same-cycle enqueue and dequeue, allowing perfect alignment between fetch and dispatch stages. This would reduce front-end stalls and enable smoother branch recovery. However, we did not fully integrate it due to shifting priorities near the final submission.

Collectively, these unrealized features represent low-effort, high-impact improvements that would immediately reduce CPI, improve memory-level parallelism, and unlock performance closer to our architectural ceiling.

8. Project Management

At the beginning of the project, the group was extremely disorganized, and it was difficult to set a clear goal for everyone to follow. We worked more online through voice calls than in person. Ironically, communication was barely there, and nothing was really happening to improve it (not to mention losing personnel). Unfortunately, but expectedly, this leaked into our Milestone 1 performance, and we ended up a couple of weeks behind other groups. Regrettably, this caused stress and mental strain on the group, and coordination dropped even more.

After some much-needed counseling from the staff, we integrated an improved management strategy where we would give the staff daily updates on Piazza. This way, individuals would be more responsible with their time and progress. We also had a day during Milestone 2 where we all came together in person and figured out what in-person scheduling looked like for everyone. We wrote down the conflicts that were occurring at the time, the status of where everyone was in terms of project progress, and future steps. This was crucial for our team because we had something concrete to look at and tackle.

After a couple of weeks, our progress slowly started to get back on track and improve our performance. Parts of the program were assigned to different members based on timing and the importance of overall completeness. Modules that group members worked on were developed using the Spiral development methodology, where they implemented and individually tested the modules. Then, through integration, the code was iterated on and improved for robustness and accuracy.

At this time, everyone was trying their best to get work done, which juxtaposes how we were a couple of weeks earlier. Unfortunately, the pace of work between members was not compatible—be that due to external reasons or not—and some teammates had to do more to meet the deadline. This was unfortunate, but it was the reality. On the other hand, everyone was showing up to meetings, going to office hours, and making sure they managed their time and role as best as possible.

9. Conclusion

Despite a late start, team changes, and early coordination challenges, R5SEBA is a successful implementation of a 2-way superscalar, out-of-order processor. Key architectural features—such as a FIFO design philosophy, divergent store pipeline, load-store queue, and early tag broadcasting—enabled us to meet project goals and lay a strong foundation for further improvement.

This project gave us valuable experience with hardware tradeoffs, timing, speculation, and scalability. With more time, we would address known bottlenecks, such as the commit register and memory pipelining, to further increase performance.

We also learned the importance of clear communication and organization. By improving our team processes mid-project, we greatly enhanced collaboration and productivity. These lessons will serve us well in future engineering efforts.

Special Thanks

We thank the EECS 470 staff for their support and resources throughout this project.