EECS 470 Final Project Report: N-Way Superscalar R10K Processor

Project Duration:
February 5, 2025 - April 19, 2025

Group 13:
Nathan Casey
Jayce Clarke
Holden Halucha
Siddharth Karthikeyan
Soham Patel
Wenbo Xu

Date Prepared:
April 22, 2025

Prepared For:
Dr. Ronald Dreslinski Jr.
Associate Professor, EECS – Computer Science and Engineering
University of Michigan

Dr. Krisztián Flautner
Professor, EECS – Computer Science and Engineering
University of Michigan

Table of Contents

Introduction

Project Background

Building upon the VeriSimpleV RISC-V pipeline, we were required to implement out-of-order features in our pipeline to minimize wasted cycles. The ultimate goal for this was to implement a high-performance design through the implementation of basic, simpler advanced, and difficult advanced features that demonstrate the superior performance of out-of-order execution.

System Goals

For the system level, our goal was to create a processor that has quick execution, keeps speculation to a minimum, and tries to manage decent CPI. We built support for N-way superscalar and Early Tag Broadcast very early into our design. We decided to not pursue Early Branch Resolution due to the added complexity of selectively squashing and keeping checkpoints.

With the R10K architecture, we prioritized decreasing our clock period over minimizing our CPI. We anticipated that we could manage a higher CPI if our clock period was lower to compensate. This could provide the same latency if managed correctly. Therefore, we went for both Direct Mapped Caches on the Icache and Dcache for quicker access time. However, we didn't just ignore CPI, and tried to mitigate our CPI by adding a Retirement Buffer, Writeback Buffer, and a Victim Cache. All of these changes also prioritized loads to memory over stores allowing for a lower CPI. Furthermore, we mitigate our CPI using a Prefetcher, IMSHR, and a DMSHR which allows for our system to not be stopped when an instruction or data address misses and requests memory in parallel. Thus, we could achieve a greater performance improvement through a lower clock period than what we could achieve by minimizing our CPI. If we managed it correctly, then we could do so without the added complexity of multi banking, or set associativity.

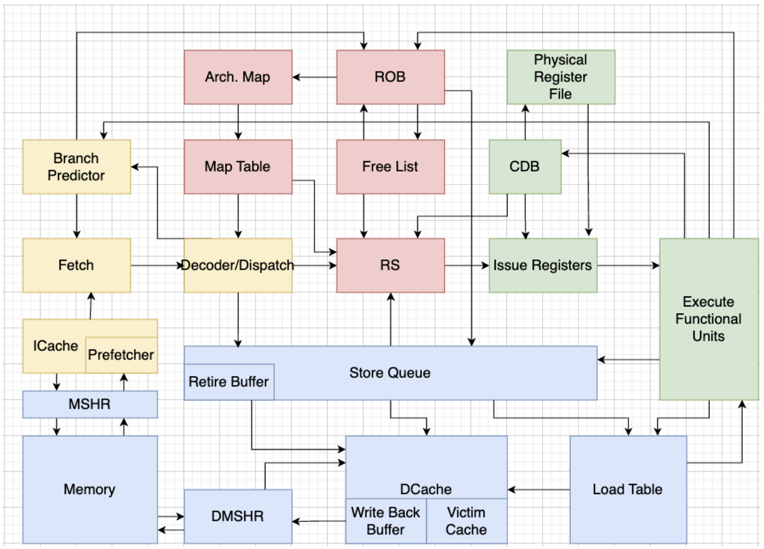


Fig. 1: High Level Processor Overview

Implementation

Out-of-Order Architecture

R10K was used as the foundation for our project. We chose this architecture over P6 and Tomasulo's algorithm for a few reasons, mainly its clear separation of stages and cleaner register renaming. This architecture is out-of-order, and uses a reorder buffer as an intermediary to write the outputs to memory in-order.

Difficult Advanced Features

We built two different advanced features into our design: N-way superscalar, and Early Tag Broadcast. We were successful in building both into our system. The table below summarizes the advanced features.

Feature	**Status**	**Notes**
N-way superscalar	Complete, Integrated, and in Final Submission	It allows us to change the amount of minimum width in the pipeline in issue, execute, and retire. However, due to only being able to fetch two instructions from one block of memory each cycle, more than N = 2 wasn't very helpful.
Early Tag Broadcast	Complete, Integrated, and in Final Submission	Allows dependent instructions to execute one after another without sacrificing clock period.

Table 1: Advanced feature summary

Simpler Advanced Features

Feature	**Status**	**Notes**
GShare Branch Predictor	Complete, Integrated, and in Final Submission	We used GShare Branch Predictor with BTB to predict our taken and not taken on branch instructions.
Instruction Prefetching	Complete, Integrated, and in Final Submission	A prefetching module allows us to prefetch the next sequential address following the last PC requested from the fetch stage, which can change to different amounts of prefetch blocks to be prefetched.
Non-Blocking Icache	Complete, Integrated, and in Final Submission	We have a MSHR (Icache MSHR), which allows for multiple Icache requests to occur without stalling due to miss.
Non-Blocking Data Cache	Complete, Integrated, and in Final Submission	We have a DMSHR (Dcache MSHR), which allows for multiple Dcache requests to occur without stalling due to miss.
Load Table/Store Queue	Complete, Integrated, and in Final Submission	After the address of all previous stores are calculated, the load would enter the load table, and be sent to execute when it is completed. Allowing for multiple loads to be in the process simultaneously.
Byte Level Data Forwarding From Stores to Loads	Complete, Integrated, and in Final Submission	Successfully forwards values whenever a load or store with

		matching addresses issues.
GUI Debugger	Complete, Integrated, and in Final Submission	Reads a binary dump file and uses Python to allow cycle by cycle visuals of all components in the processor.
Victim Cache	Complete, Integrated, and **NOT** in Final Submission	Fully Associative with LRU (least recently used) eviction policy.

Table 2: Simple Feature Summary

Module Microarchitecture

Reservation Station

The reservation station is a buffer that temporarily holds instructions and their operands until they're ready to execute. By buffering instructions awaiting operand availability, it allows the processor to continue dispatching and issuing subsequent instructions without stalling. It resolves data hazards by tracking operand readiness, ensuring instructions only execute once their input data dependencies are satisfied, which preserves correct program order in out-of-order execution. Additionally, it schedules instructions based on resource availability, efficiently utilizing execution units by issuing instructions as soon as their operands become available.

ROB

The Reorder Buffer is a FIFO structure that tracks all instructions issued for out-of-order execution, holding their results until they can be safely retired in program order. It buffers instruction status, destination register results, and exception information, allowing the processor to continue issuing and executing instructions speculatively without waiting for sequential retirement. The ROB ensures precise exceptions and correct program semantics by retiring instruction results strictly in their original order, even if execution completes out-of-order. It also enables efficient recovery from branch mispredictions by discarding speculative instruction results from incorrectly predicted paths.

Execute/Functional Units (Basic)

The design includes 4 ALU FUs, 2 MULT FUs, 2 LOAD FUs, 2 STORE FUs, and 2 dedicated address calculation units for load operations. This configuration allows for parallel execution of different instruction types, with the ALU units being the most numerous to handle the high frequency of these kinds of operations. The multiplication units are fewer in number and take multiple cycles to operate, while the load and store units are balanced in number to maintain memory operation throughput. The dedicated address calculation units for loads help differentiate the loads having the address calculated versus LOAD FUs, which are the loads getting sign extended or not returning to get onto the CDB.

Load Table

Our Load Table is responsible for supplying loads with their correct values, using a simple FSM-based design. Loads first check the Store Queue for any forwarded data. If the data isn't fully available, they transition to check the DCache, and finally the DMSHR for memory responses. Once a load has all its required data at any stage, it moves to the valid state and is sent to Execute for completion. This clear flow ensures reliable load handling across all stages and loads get the appropriate forwarded data if it exists.

Store Queue & Retire Buffer

The store queue is a FIFO buffer that temporarily holds store instructions, their addresses, data, and byte-level masks until committed to memory in program order. It enables store-to-load forwarding, allowing loads to directly access matching data from pending stores, significantly reducing load latency. The queue ensures memory consistency by maintaining sequential store retirement and preserving correct memory ordering. Additionally, it supports memory disambiguation by tracking in-flight stores and resolves dependencies for accurate out-of-order execution. At retirement, the store is then put into a Retire Buffer, which then commits the data to the cache.

Data Cache, Writeback Buffer, & Victim

Our Data Cache was Direct Mapped with a small Full Associative Victim with an LRU (Least Recently Used) eviction policy. We also added a Writeback Buffer, so that we could prioritize load instructions requesting memory blocks from memory over storing evicted blocks going to memory. This would allow loads to finish and free up dependent instructions after it. The Dcache has these two components internally initialized and allows for parallel access. We also have appropriate stalling logic for Load Table, Retire Buffer if the Writeback Buffer is full, and proper obliterate signals to obliterate load requests when branch mispredict.

Fetch, Instruction Cache, & Prefetch

Our fetch stage is responsible for retrieving at most two instructions per cycle and operates based on the PC which ultimately determines the addresses of the instruction line to be fetched. The Instruction Cache is a Direct mapped, non-blocking, 256 byte cache. If the instruction line is present in our I-cache, fetch proceeds normally. Otherwise a miss is flagged which sends out a request to main memory. To reduce the stall penalty associated with an Icache miss, we implemented a prefetcher that activates on an Icache miss. The prefetch routine stops its service naturally upon reaching our prefetch threshold, or may end early if the fetch stage encounters a mispredicted branch, or if the Icache loses priority to main memory

IMSHR & DMSHR

To make our instruction cache non-blocking and capable of multiple in-flight memory requests, we have embedded our Instruction Miss Status Handle Register (IMSHR) directly into our ICache. As for the Data Miss Status Handle Register (DMSHR), it is an external module not embedded into the DCache. This is because it has to interact with our load table as well to forward load values. Transaction tags upon arrival act as indices to our MSHRs, where each entry retains the address that is currently in-flight. This tag is later returned alongside the data from main memory. Upon receiving this tag, we are able to correctly index into our MSHR, and send out the corresponding address with its data.

A key design decision we implemented was a coalescing mechanism, disallowing our MSHRs from creating multiple duplicate entries for the same address. We found that this optimization prevented redundant traffic to main memory, and preserved more open slots in our MSHRs, allowing for only unique misses, improving our performance significantly.

GShare Branch Predictor

The GShare branch predictor is a type of branch prediction that combines global branch history with program counter information to make accurate predictions. In our design, we maintain a global history register that tracks the outcomes of recent branches and use this history to index into a branch history table that stores 2-bit saturating counters. We compare the global history register with bits from the program counter, allowing us to capture both local and global branch patterns. This hybrid approach combining global history with program counter information makes the GShare predictor more effective than other branch prediction algorithms that rely solely on recent branch outcomes.

Map Table

The Map Table holds the speculative mapping between Architectural and Physical Registers, and updates on every instruction dispatch. It renames source and destination registers while tracking ready bits to signal when values are available. On a branch mispredict, the table is squashed and restored from the Arch Table, making recovery simple and reliable.

Arch Map Table

This is a non-speculative state of all of Physical Register to Architectural Register mapping, and is only updated when an instruction retires. When we hit a miss predict on a branch, this allows us to squash the Map Table and revert it to the Arch Map Table. This module allowed for a simple way to revert on a branch miss predict, and also was the only module which did not have any integration bug. This led to it being a reason why we decided not to implement Early Branch Resolution, as we had a well working system for squashing already implemented.

Design and Testing Process

Module Testing

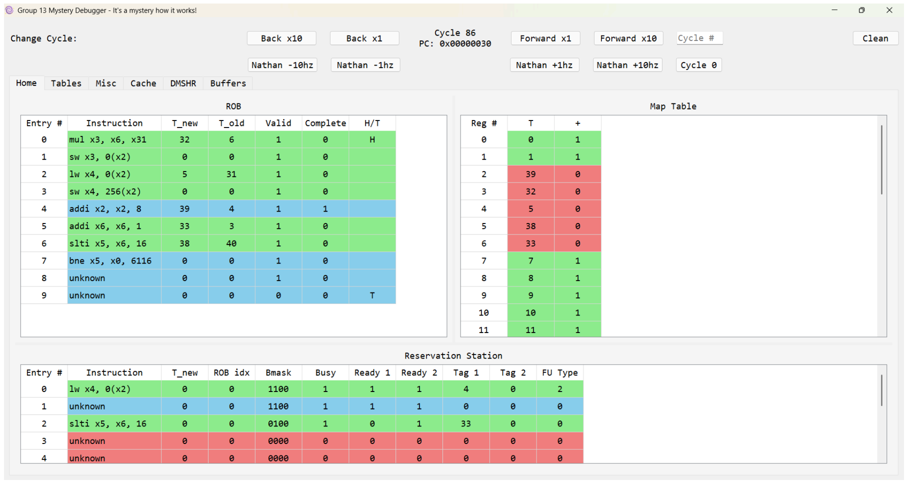
Our testing strategy began with traditional Verilog testbenches, which were used to verify each module in isolation and ensure correct functionality at the component level. During this phase, we also employed *snake testing* — feeding structured, easily traceable data patterns through some of the modules — to validate basic data flow and interconnect integrity in early designs on different N-way sizes. The modules that had snake testing employed virtually had zero bugs throughout our whole development process.

System Testing

Once all individual modules had passed their respective testbenches, we shifted focus toward unit tests targeting the fully integrated system. These tests were designed to evaluate specific behaviors under more complex conditions, such as byte-level forwarding and edge case handling, helping to expose subtle bugs that only emerge at the system level. As the project matured, our debugging workflow evolved to rely more heavily on Verdi and the Group 13 Mystery Debugger, allowing for streamlined visualization and analysis during system-level testing. We mainly tested two systems: one system without memory, and one system with memory components. Overall we believe that our testing methodology was thorough. We started with testing edge cases at the module level, then moved on to testing edge cases on the systems level.

GUI Debugger Application

To better graphically visualize the internal workings of our processor, we decided it was necessary to develop a visual debugger. We chose PyQt5 to build the graphical interface. The backend consists of a Verilog debugger module connected directly to the CPU. This module captures packets of data on every clock cycle and writes them into a binary dump file. This binary dump is later read and decoded to display the debugger.



Cycle 86
PC: 0x00000030

ROB

Entry #	Instruction	T_new	T_old	Valid	Complete	H/T
0	mul x3, x6, x31	32	6	1	0	H
1	sw x3, 0(x2)	0	0	1	0	
2	lw x4, 0(x2)	5	31	1	0	
3	sw x4, 256(x2)	0	0	1	0	
4	addi x2, x2, 8	39	4	1	1	
5	addi x6, x6, 1	33	3	1	0	
6	slti x5, x6, 16	38	40	1	0	
7	bne x5, x0, 6116	0	0	1	0	
8	unknown	0	0	1	0	
9	unknown	0	0	0	0	T

Map Table

Reg #	T	+
0	0	1
1	1	1
2	39	0
3	32	0
4	5	0
5	38	0
6	33	0
7	7	1
8	8	1
9	9	1
10	10	1
11	11	1

Reservation Station

Entry #	Instruction	T_new	ROB idx	Bmask	Busy	Ready 1	Ready 2	Tag 1	Tag 2	FU Type
0	lw x4, 0(x2)	0	0	1100	1	1	1	4	0	2
1	unknown	0	0	1100	1	1	1	0	0	0
2	slti x5, x6, 16	0	0	0100	1	0	1	33	0	0
3	unknown	0	0	0000	0	0	0	0	0	0
4	unknown	0	0	0000	0	0	0	0	0	0

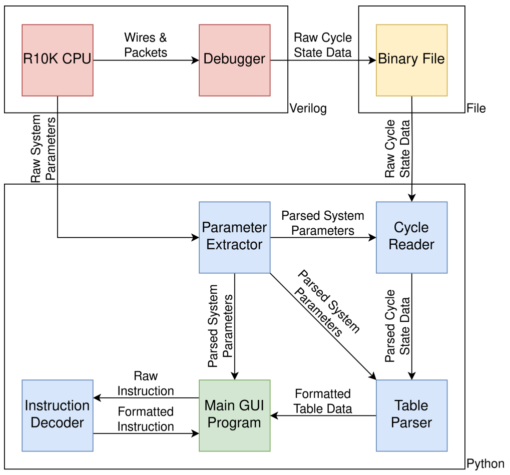


Fig. 2: Group 13 Mystery Debugger "Home" Tab (left) and System Diagram (right)

Due to the long runtime of all the programs, we also created a shell script that iteratively edits a variable multiple times in steps and outputs a graph of CPI and latency. The script calls several Python programs during execution and is helpful for figuring out the optimal parameter values for lowest CPI or latency.

Performance Analysis

Overall Analysis

A major bottleneck of our design was that at any given cycle we could only fetch 2 instructions as there were only two instructions retrieved from memory or Icache. This hindered us from utilizing our N-way capabilities. This bottleneck was unpreventable given we could only fetch one memory block a time with the given memory module.

Another bottleneck of our design was the conflict misses in our direct-mapped caches. To address this bottleneck, we implemented a victim cache in the Dcache. This allowed us to get the lower access time of a direct-mapped cache while also addressing some of the conflict misses present in the direct-mapped caches.

We achieved an overall clock period of 7.8 ns and an average cpi of 1.97 (without halt, btest1, and btest2). We chose to use Gshare for the branch predictor, include a victim cache, use a Rob Size of 12, a Prefetch Size of 12 memory blocks, and use 2 Way Superscalar. The justification for each feature or parameter size are provided below.

GShare Branch Predictor

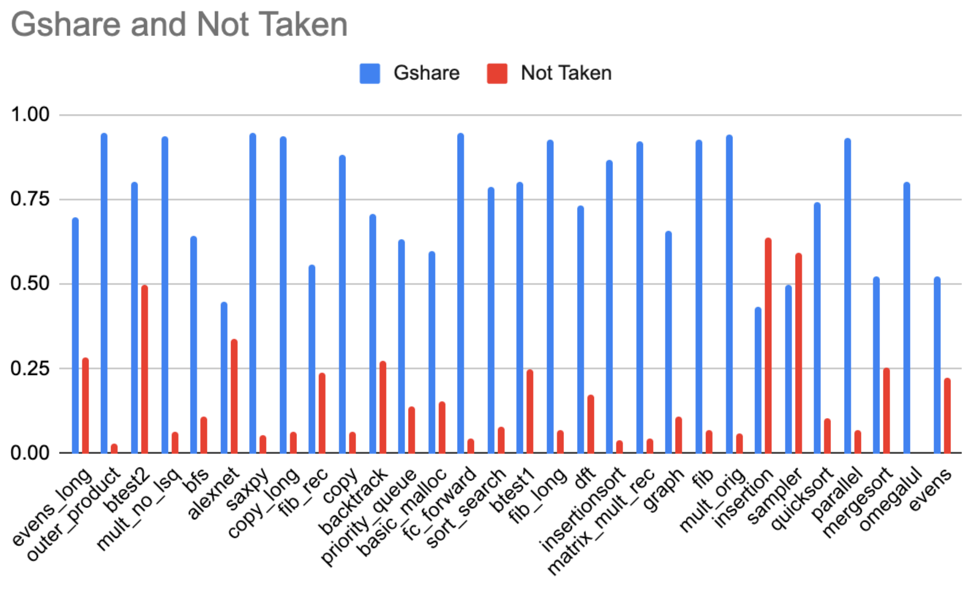
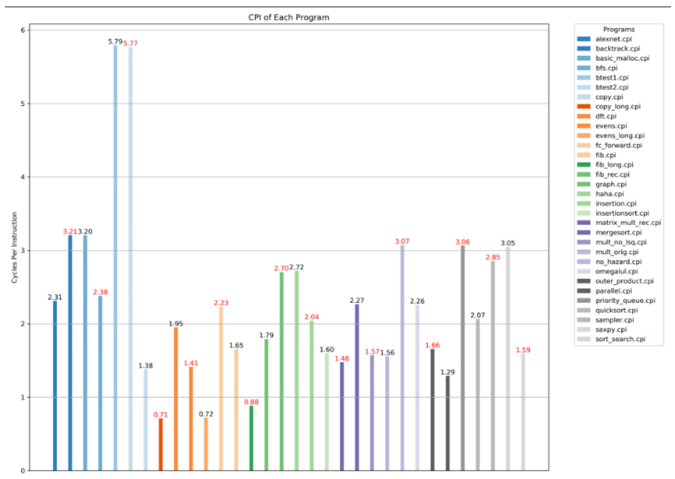


Fig. 3: GShare Branch Predictor Accuracy vs Not Taken for Sample Program Suite

Gshare consistently outperforms the "Not Taken" branch prediction across nearly all benchmarks (Fig. 3), with notable advantages in computationally intensive and recursive tests. The average prediction accuracy for Gshare across all tests is approximately 75.67%, highlighting its effectiveness in adapting to varying branch behaviors compared to a static branch predictor. The variability in performance across different benchmarks indicates the influence of workload-specific branch behaviors, with Gshare effectively capturing these dynamic patterns to minimize mispredictions. However, Gshare does fall short in certain benchmarks, such as "insertion" and "alexnet" indicating that there remains room for improvement in handling specific branch prediction patterns.

Victim Cache


With Victim Cache

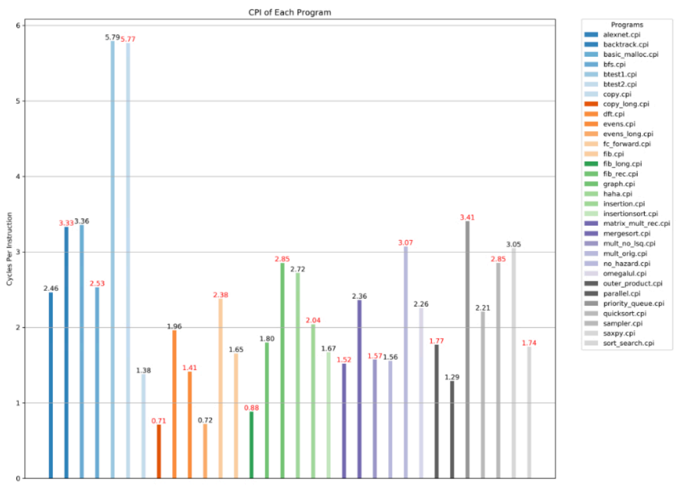

Without Victim Cache

Fig. 4: Comparing With and Without Victim Cache CPI

In the benchmarks (Figure 4), all programs experienced either a reduction in CPI or no change in CPI due to a Victim Cache. The impact of Victim Cache was more profound on larger programs. Programs with a reduction in CPI were outer_product, bfs, alexnet, haha, fib_rec, backtrack, priority_queue, basic_malloc, fc_forward, sort_search, DFT, insertionsort, matrix_mult_rec, graph, quicksort, and mergesort. Furthermore, our processor got a boost in total latency, where without the victim cache, our total latency was 14546687.9 ns, and with the victim cache, our total latency was 13696948.1 ns, a 5.84% reduction in total latency. This justifies our implementation and addition of the Victim Cache.

Parameter Analysis

ROB Size

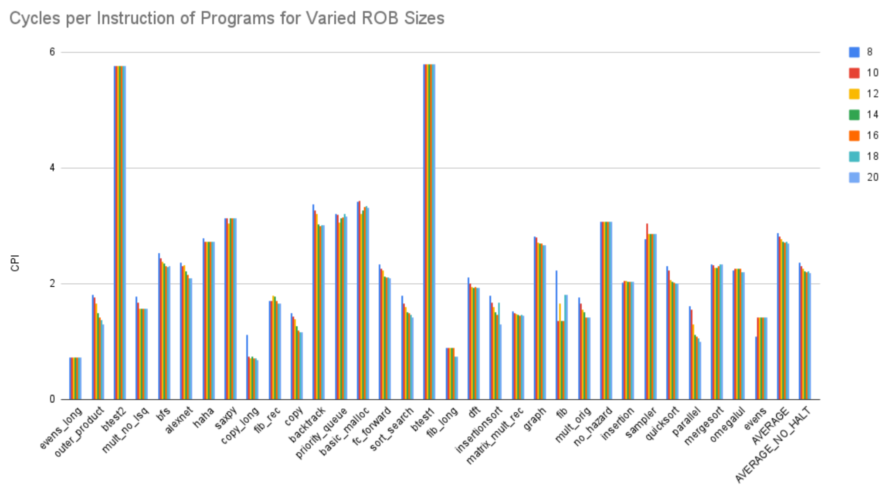


Fig. 5: Optimization Output for ROB_SZ from 8 - 20 CPI

We iteratively tested the ROB with different sizes using our optimization script to determine the best size for us. From the graph above, as the ROB size increases, the CPI slightly decreases. The main changes are increases on the lower end. We used this data to set the ROB size to 16 originally, but later changed it to 12 to reduce strain on the final clock period. This data informed our decision, allowing us to confidently see that the change from 16 to 12 would have a minimal performance hit, as the performance improvements get marginally smaller as the ROB increases in size.

Prefetching

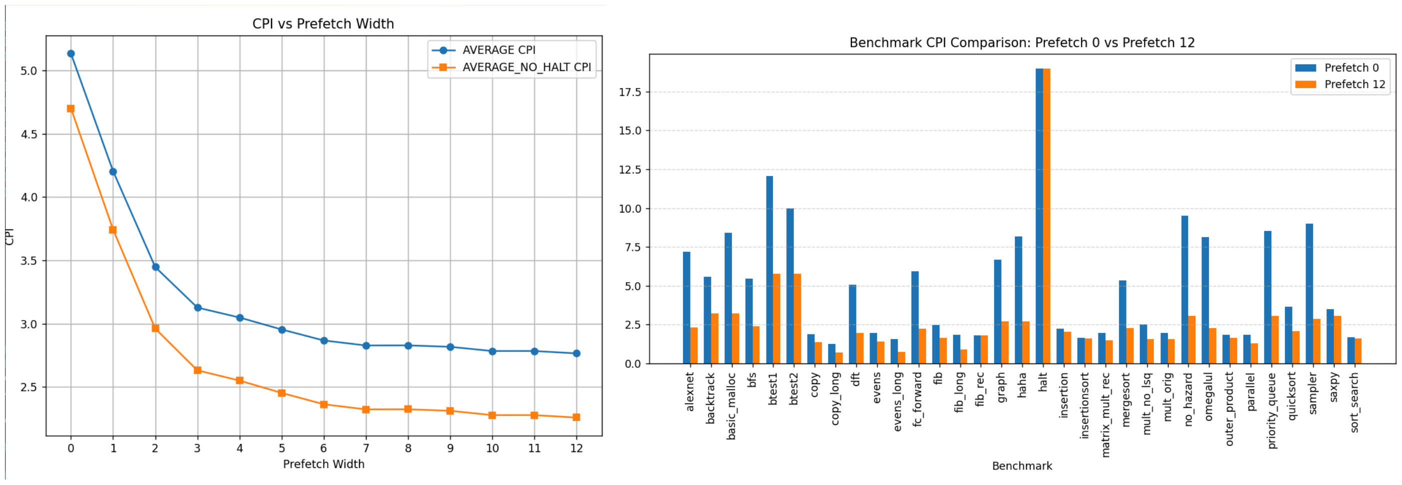


Fig. 6: Comparing Average CPI across different widths (left) and CPI improvements from 0 to 12 (right)

From our parameter sweeps (Figure 6), we determined that a prefetch width of 12 provided the lowest average CPI across all programs. On average, enabling prefetching reduced CPI from 5.14 to 2.76, a roughly 46% improvement. Enabling prefetching with a width of 12 resulted in substantial CPI improvements in nearly every program. Programs with higher baseline CPI values, such as btest1, btest2, alexnet, and sampler, benefit the most, with some showing more than a 50% reduction in CPI. Overall, prefetching significantly enhances performance by increasing our throughput and by keeping the ICache well populated throughout runtime.

N-way Superscalar

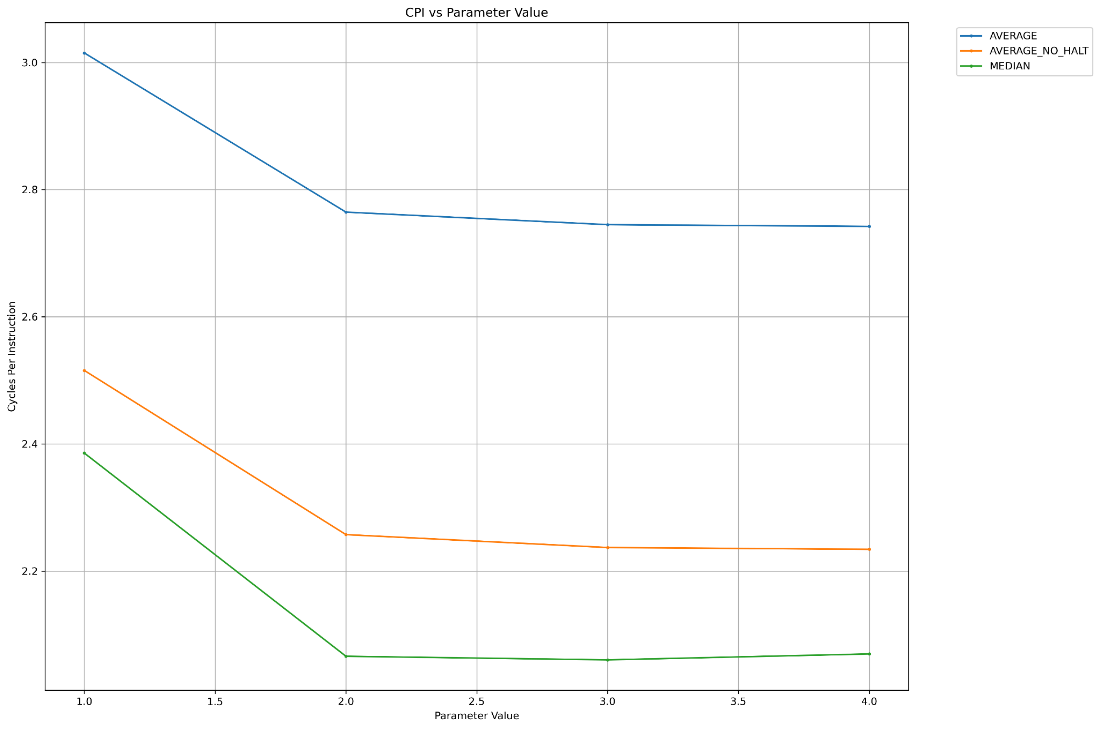


Fig. 7: N vs. Average CPI

Fig. 7 is a graph of N 1-4 with CPI on the Y axis. CPI is average across all programs in the public test bench (Figure 7). The CPI drop from N=1 to N=2 is substantial; however, further increasing N does not provide any significant decrease in CPI. Because of this, we kept N=2 in our final processor so as to not strain out clock frequency.

Timing Analysis and Critical Path

Our initial unoptimized critical path measured 8.5 ns. It was through the stages of the mult_stage module. To address this, we first reviewed the additional buses we added to the multiplier beyond its original design. We observed that both the physical destination register and the ROB tag were being passed through the multiplier, unnecessarily increasing its load. To reduce this overhead, we restructured the interaction between the ROB and the Execute stage. Instead of passing both the physical register and ROB tag through the multiplier, we modified the design so that when the done signal from the multiplier asserted “ready,” the execute stage would send the associated physical destination register to the ROB. The ROB would then return the corresponding ROB tag. This adjustment offloaded responsibility from the multiplier module and successfully reduced the critical path to 7.8 ns. The current critical path is PC register to RS, which is essentially the fetch/dispatch path.

N	2	Clock Period	7.8 ns
Fetch_Width	2	FU_ALU	4
NUM_PREFETCH (MEM Blocks)	12	FU_MULT	2
ROB_SZ	12	FU_Load	2
RS_SZ	10	FU_Store	2
Load_Table_Entries	16	MULT_STAGES	4
DMSHR_Internal Entries	5	BTB_SIZE	256

Table 3: Final Parameters Used in Our Processor

Conclusion

Design Summary

Overall, we implemented an N-way superscalar processor with early tag broadcast, characterized by an aggressive aversion to unnecessary complexity in the memory system (speculation, stalling, associativity, multibanking, etc.).

Societal impacts

Our design aimed to minimize CPI and clock period, but real-world implementations would need to take into consideration power consumption and area constraints. It might be optimal to run the processor at a slower clock period and higher CPI if it allows for a reduction in energy consumption or silicon area. A lower energy consumption would produce more efficiency and less climate impact, and less silicon area would make the processor less expensive.

Lessons Learned & Project Management

We learned quickly that we work best together when we work in groups of two, but only if all groups of two are present in the same place. Often during the beginning, we would not be able to directly communicate with other groups, resulting in a misunderstanding of how our modules interfaced with each other. Seeing this, we decided to have a dedicated meeting spot where we would meet every day. We were able to work at a more efficient pace and lessen miscommunication. Finally, we learned that asking for clarification is always better than assuming. We lost 3-4 valuable days at the end of the project because we did not understand that the memory module data comes at the negative edge of a clock, and we should offset it to the next positive edge. In hindsight, coming to office hours would have eased this confusion.

Work Delegations

Nathan	RS, ROB, Freelist, Dcache, Branch Predictor, ETB, PRF, CDB, Dispatch, Issue Registers Debugging
Jayce	RS, ROB, Dispatch, Store Queue w/Forwarding, Retirement Buffer, Gshare/BTB, Debugging
Holden	GUI Debugger App, Iterative Optimization Script, ROB, Misc. Help, Debugging
Siddharth	IF Stage, Prefetch, IMSHR, Icache, CDB, Freelist, Phys Reg File, Debugging
Soham	Execute, Load Table, DMSHR, Dcache, Writeback Buffer, Victim Cache, Arch Map Table, Debugging
Wenbo	Execute, Load Table, Map Table, IMSHR, DMSHR, Critical Path, Debugging

Table 4: Group Member Task Assignment

Next Steps & Acknowledgments

From the start, we wanted to optimize speed. That is why we went with a direct-mapped cache. With the victim cache, we hope to further implement it into a "selective victim cache," which picks what to take in. The selective victim would use a hit bit and stick bit for each memory block selectively exchanged between the victim cache and data cache. It would essentially be predicting whether a block is useful or not to keep based on prior hits and sequential access. We hope to potentially see this system outperform a traditional two-way or four-way set-associative cache in terms of total latency. Finally, we would like to thank the course staff of EECS 470 for their support throughout the project. This has been an amazing learning experience for us.