EECS 470 - Final Project Report

Team 8: Andrew Ye, Davis Sheppard, Dhruv Dighrasker, Kevin Jia, Matthew Jia, Yan Cheng Poon

1. Overview

We chose to base our processor on the R10K architecture. From the beginning, we chose to implement the following "Difficult Advanced Features":

- N-way Superscalar Width
- Early Branch Resolution
- Early Tag Broadcasting

In addition to this, we also implemented these "Simpler Advanced Features":

- Tournament Branch Predictor (GShare vs. Simple)
- Victim D-Cache
- 16-way Associative D-Cache
- Non-Blocking D-Cache
- Instruction Prefetching
- Store Queue for out-of-order memory accesses
- Data Forwarding from stores to loads

We generally followed the R10K architecture and data structures as they were presented in the lecture, with a few significant alterations and additions as described below:

- Freddy List (Free + Ready/Complete) as opposed to Map Table ready bits
- MSHR FIFOs
- Combined Store Queue and Post-Retirement Buffer
- Writeback Buffer
- Post-Data-Stage Load Buffer

Given these additional modules as well as the standard data structures required for the R10K architecture, such as ROB, Reservation Station, Branch Stack, Instruction Buffer, etc., we ended up with the following configurations regarding sizing, space, and associativity:

- 32 ROB entries
- 32 RS entries
- 32 Instruction Buffer entries
- 4 Branch Stack entries
- 8 Combined Store Queue/Buffer Entries
- 8 Load buffer Entries
- 2 ALUs
- 2 Mult Units
- 1 Load Unit
- 1 Store Unit
- 1 Branch Unit
- 5 History bits for Branch Predictor BHR
- 8-way BTB
- 16-way D-Cache
- Direct Mapped, 2-Banked I-Cache
- 16 MSHRS each, in FIFO ordering, for I-Cache and D-Cache
- Prefetch Window of 32 instructions

Final Performance Metrics:

All of the following metrics are based on performance analysis done for the C programs we were provided when compiled with the default optimization level. We took CPI data averaged over each program as well as a weighted average over the total number of instructions across all programs.

Clock period:	7.85ns
Average Weighted CPI:	1.12
Average Unweighted CPI:	1.43
Weighted Iron Law:	8.79 ns/inst
Unweighted Iron Law:	11.18 ns/inst

System Diagram:

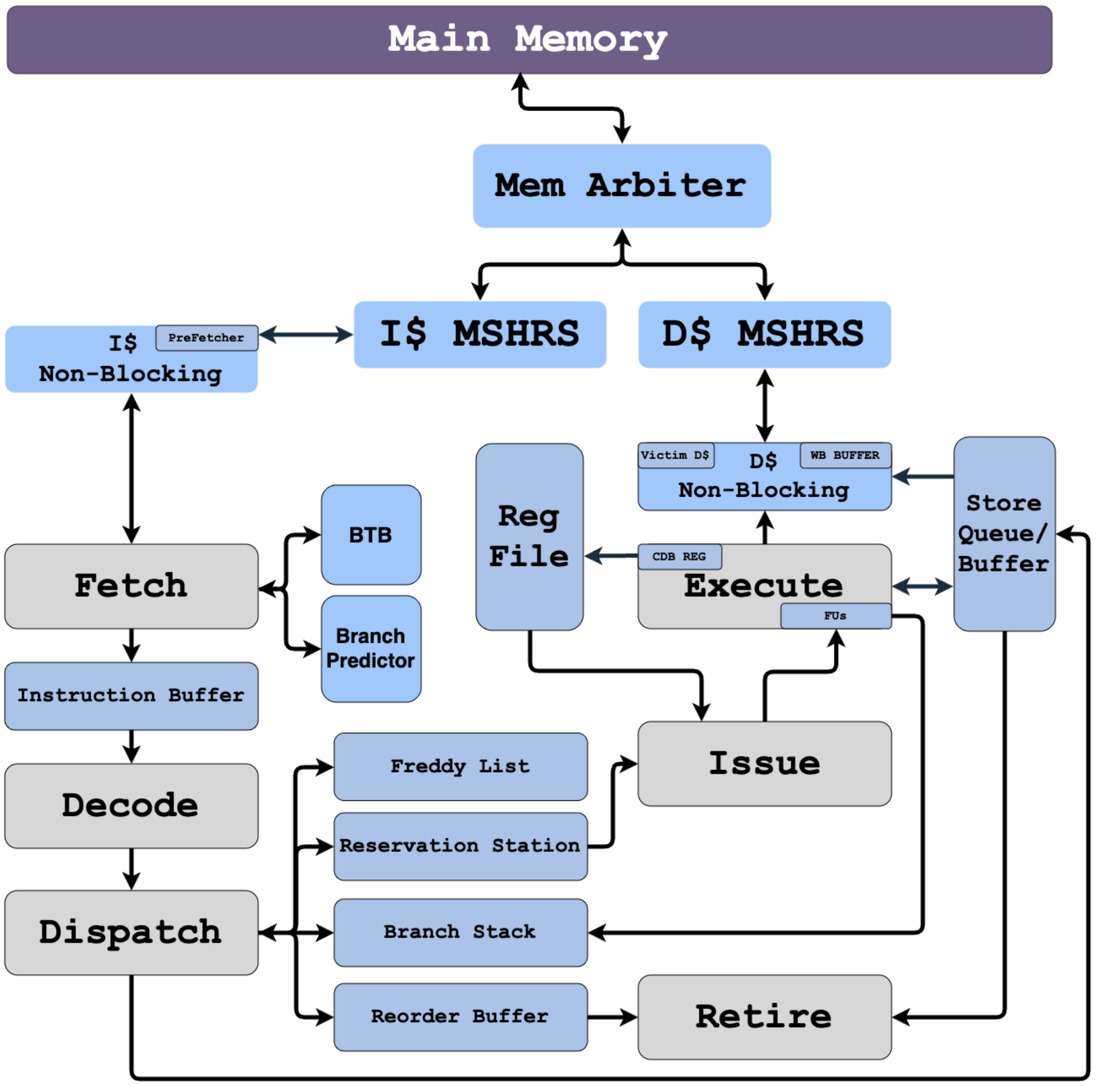

Figure 1. High-Level Architecture Diagram

2. Testing Methodology

We went through several phases of testing throughout the semester, each of which involved different approaches for debugging and analyzing correctness. To understand our testing methodology, it is important to know how we organized our modules in code. To start, we made the strategic decision to make our data structure modules very passive and containerized. The bulk of the combinational logic was done in our "Stage" modules which primarily took in state information from the data structures and calculated the next outputs to send to each container.

Phase 1: Pre-Milestone 2
During the first few weeks of the project, we focused on implementing the main data structures in the R10K architecture. These included the ROB, RS, Branch Stack, and Freddy List. In order to test the correctness of our implementations, we employed SVAs. We wrote a significant amount of rigorous assertions and property statements for each data structure, and ran these assertions on test benches with both edge cases and generic test coverage. We are confident in the correctness of these data structures, both due to the passive and simplistic nature of their code as well as the strength and coverage of the assertions encoded in our SVAs.

Phase 2: Post-Integration
Once we integrated the backend of our processor and passed mult_no_lsq, we switched our debugging approach to using display statements. We would print out state information from each cycle during execution into a log file and search for key signals to pinpoint when things went wrong. This worked fairly well for us, as we were able to achieve full correctness on the public test cases over a week before the deadline. Not only this, but we were also able to quickly achieve full correctness on the C programs when using different compiler optimization settings, including o1, o2, o3, and os.

Phase 3: Full Performance Analysis
With our remaining time, we wanted to fine-tune the parameters and logic in our CPU to maximize performance and minimize our clock period by finding our critical path. The main considerations we focused on with the time we had left were the following:

- What to set N to?
- How many multiplier stages to use?
- When to reset the PC in the fetch stage on a branch mispredict?
- Associativity of the D-Cache and BTB
- Prefetch Distance
- BHR size

3. Advanced Feature Analysis

For any features not analyzed in depth here, we lacked the ability to empirically compare our implementations to any reasonable alternatives since many of the decisions we made in code were not easily toggled on and off, and we did not have sufficient time to address this by the end of the project. However, our high overall performance is indicative of the large amount of time and thought we put into these decisions before coding. The many hours of discussion and debate that we engaged in as a group are what made us confident that we could achieve our desired performance with the implementations we decided on.

3.1. N-way Superscalar Width

The first "major" advanced feature that we chose to implement was N-way Superscalar Width. This would help us increase our throughput by increasing the number of instructions we retire per cycle, therefore lowering our CPI. Although we updated several modules to support N-way logic with dependent for-loops, the most significant change was in the execute module, which is responsible for selecting n instructions for completion and latching them onto the CDB register.

We chose our superscalar width by conducting a performance analysis with Mult Stages set to 4 and varying N, and concluded that N = 2 ways was optimal. N = 1 would bottleneck our CPI due to not taking advantage of the program ILP, as clearly shown in the graph below. Oppositely, since the C programs did not have enough ILP to take advantage of the greater superscalar width, N = 3 way did not provide a sufficient CPI boost to justify the increased clock period (9.1ns) stemming from our superscalar-width-dependent for-loop in Dispatch. Thus, N = 2 would result in the best balance between CPI and clock period for the lowest Iron Law.

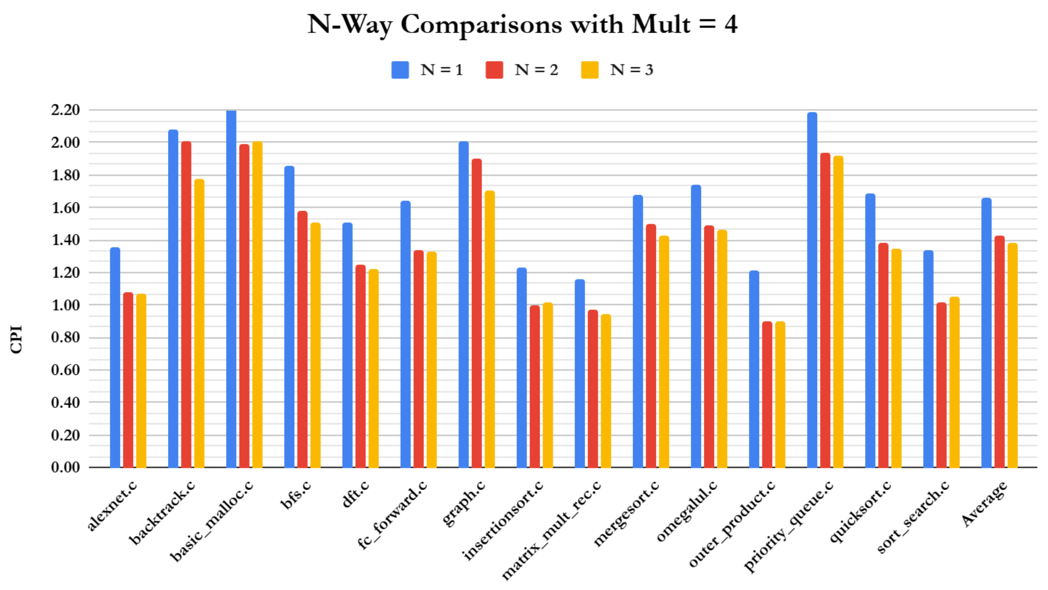

Figure 2. N-Way CPI Comparison Chart

3.1.1. Multiplier Stages

After selecting N = 2 Ways, we wanted to optimize the number of multiplier stages, because the multiplier is likely the critical path. We began by conducting a similar analysis to superscalar width, varying Mult Stages for a 2-way superscalar width. 2, 4, and 8-stage multipliers resulted in very similar CPI, so it was easy to rule out 2 and 4-stage multipliers due to their critical path bottlenecks. The 16-stage had a higher CPI, but could potentially allow us to push our clock period below 7.85ns (which is the clock period set with an 8-stage multiplier on the critical path). We were curious how low we could push our clock period without the multiplier on the critical path, and whether this would result in an improved Iron Law given the increased CPI.

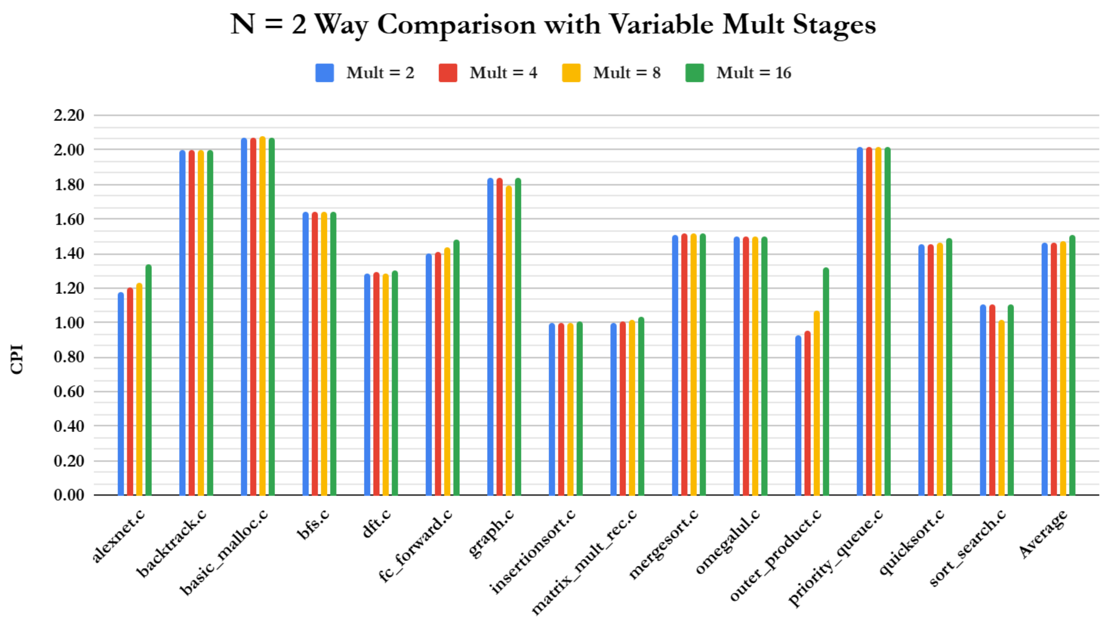

Figure 3. Multi-Stage CPI Comparison Chart

Therefore, after determining that we had to choose between 8 and 16 stage multiplier, we conducted a more involved analysis, testing each C program and optimization with both pipeline widths for both weighted and unweighted CPI (weighted = total cycles/total program lines across all C programs; unweighted = average of program CPIs). We also ran multiple syntheses to push our clock periods for both pipeline widths, bottoming at 7.85ns for 8-stage and 7.4ns for 16-stage (in which case the critical path was not actually the multiplier, but rather, our Fetch stage). After collecting all of this data, we compared the speedup of the 8-stage multiplier over the 16-stage multiplier, and concluded that the 8-stage multiplier was substantially faster for weighted Iron Law, and marginally slower for unweighted Iron Law, hence, we are convinced that the 8-stage multiplier is clearly a better choice. However, from a more critical stance, we should point out that outer_product.c substantially shifts the odds in favor of the 8-stage multiplier, but it is more than reasonable that multiplication is a real-world workload for a processor. As a result, it makes sense to choose a pipeline width that offers better performance for such a workload.

Weighted CPIs across all C programs for 8 vs 16 Stage							
Stages/Opts	O0	O1	O2	O3	OS	Average	Clock Period (ns)
8 Stage	1.120	1.455	1.352	1.456	1.310	1.339	7.85
16 Stage	1.247	1.625	1.538	1.634	1.457	1.500	7.40

8 Stage % Speedup over 16 Stage, Unweighted						
Programs/Opts.	O0	O1	O2	O3	OS	Average
alexnet.c	8.06%	10.06%	18.54%	0.00%	10.33%	9.40%
backtrack.c	6.03%	0.00%	0.00%	0.15%	-0.28%	1.18%
basic_malloc.c	2.97%	0.00%	0.00%	0.00%	0.00%	0.59%
bfs.c	2.21%	0.00%	0.00%	0.00%	1.08%	0.66%
dft.c	3.39%	3.83%	4.92%	4.66%	3.97%	4.15%
fc_forward.c	5.01%	5.46%	6.89%	6.89%	14.22%	7.69%
graph.c	2.77%	-0.14%	0.00%	0.00%	0.44%	0.61%
insertionsort.c	3.02%	5.13%	5.37%	5.37%	0.00%	3.78%
matrix_mult_rec.c	3.05%	2.06%	1.54%	1.32%	2.19%	2.03%
mergesort.c	1.91%	0.00%	0.00%	0.00%	0.00%	0.38%
omegalul.c	2.60%	0.00%	0.00%	0.00%	0.00%	0.52%
outer_product.c	20.14%	17.35%	17.98%	17.95%	19.63%	18.61%
priority_queue.c	2.57%	0.00%	0.00%	0.00%	0.00%	0.51%
quicksort.c	1.25%	4.13%	3.22%	4.64%	3.07%	3.26%
sort_search.c	2.21%	0.00%	0.00%	0.00%	-0.10%	0.42%
CPI and Iron Law Speedup Summary						
Weighted CPI	10.18%	10.46%	12.09%	10.89%	10.09%	10.74%
Unweighted CPI	4.35%	3.24%	3.94%	2.75%	3.74%	3.60%
Weighted Iron Law	5.99%	6.28%	7.99%	6.74%	5.89%	6.58%
UW Iron Law	-0.12%	-1.28%	-0.54%	-1.79%	-0.75%	-0.90%
8 Stage Overall Average Iron Law Speedup over 16 Stage						
Weighted	6.61%					
Unweighted	-0.92%					

Table 1. Analysis on Speedup between 8-stage and 16-stage multiplier

3.2. Early Branch Resolution

Our EBR feature takes advantage of the idea that once a branch (unconditional or conditional) exits its associated functional unit, the processor knows whether or not it was mispredicted and can immediately rollback accordingly. It achieves this by giving each dispatched branch a unique one-hot bit-vector called a branch mask mask and turning this bit high in an identically sized branch mask register. This branch mask would then be attached to every subsequently dispatched instruction. Additionally, to keep track of the state of the processor at the time of dispatch, EBR copies the current free list, map table, and store queue mask as well as pointers to the tail of the ROB and store queue in the case of a mispredict. It also collects the PC to recover via the BTB or the current PC itself and stores all this information in a data structure named the branch stack. During branch resolution, EBR turns the bit associated with the branch off in the branch mask register and branch masks of dependent instructions. In the case of a mispredict, EBR squashes all aforementioned instructions, sets fetch along the correct path, and reverts the state of the processor through the branch stack.

Given how interconnected the feature is with other aspects of the processor, we began molding our design with its implementation in mind very early on in the semester. Thus, no data of the processor's performance without EBR was collected, but we assume it greatly reduced CPI.

3.3. Early Tag Broadcasting

Another advanced feature we implemented was Early Tag Broadcast (ETB). Without ETB, the tags that the CDB broadcasts to the Reservation Station arrive and are latched on the same cycle that the data is latched to the CDB register. This causes a cycle delay between data being calculated and instructions waiting on that data from being allowed to issue. With ETB, tags are broadcast the same cycle that data is finished being calculated, allowing dependent instructions to issue early and grab forwarded data right before they start execution. This helped reduce CPI dramatically (around a 1.33x speedup).

3.4. Tournament Branch Predictor (GShare, Simple, Meta)

The Tournament Branch Predictor can predict both history-dependent and independent branches, which is useful across a range of programs and warm-up times. We speculatively update BHR with the status of the prediction in Fetch by setting Not Taken until the first Taken prediction, but only update the PHTs after the branch's result is known in Complete. While testing the size of the BHR, we found that 5 bits was sufficiently large to be useful for history-dependent branches, and that increasing beyond 5 bits did not net any increase in performance. Additionally, we initialized GShare PHTs to Weakly Taken and Simple PHTs to Weakly Not Taken, with the Meta predictor initialized to Weakly Simple. Analyzing each predictor in Figure 4, we found that simple PHT does better in basic_malloc, dft, mergesort, and quicksort, while GShare does better in alexnet and backtrack.

The Tournament Branch Predictor takes the best out of both in most cases, or at least the average of both, further solidifying our BP choice. The weighted average accuracy across all C programs for this configuration was 87%, a significant improvement over the weighted average accuracy of Always Not Taken at 31%.

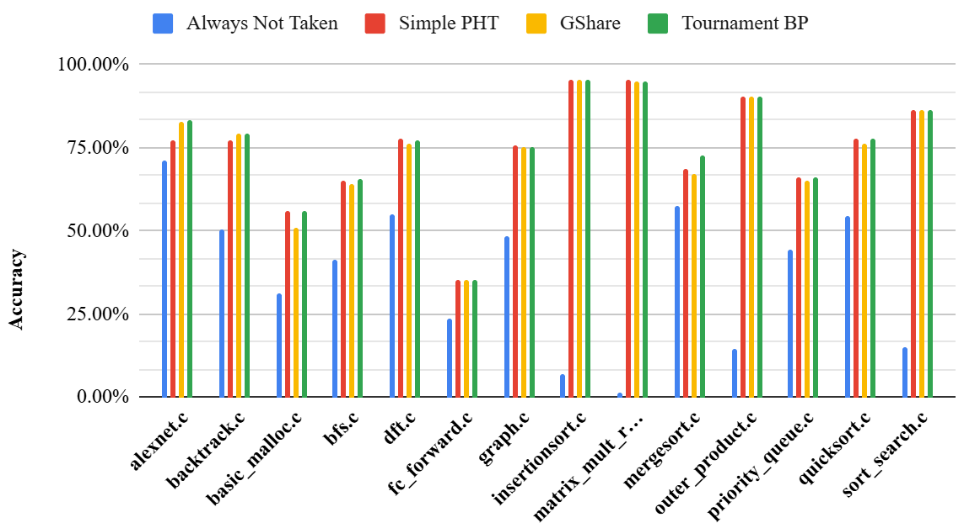

Figure 4. Branch Prediction Accuracy of Different Branch Predictors

3.5. Memory Arbitration

We employed a priority scheme for memory arbitration. The priority for our main memory arbitration is as follows, from most prioritized to least prioritized: Load/Store Request, Fetch Instruction Miss, Dirty D-Cache Line, Instruction Prefetcher.

3.6. Non-Blocking 16-Way Set-Associative D-Cache

The data cache (D-Cache) is implemented as a non-blocking, write-back, 16-way set-associative cache. It has 16 miss status holding registers (MSHR) with a fully associative 4-line victim cache and a 4-line write-back buffer. We used pure LRU for each set in the D-Cache as well as our V-Cache. Our D-Cache allows for only one access to its memDP module at a time. There are two possible sources that these accesses could come from: a load/store hit in the D-Cache/V-Cache, or an MSHR that has gotten data back from main memory. We prioritize the load/store hits over the MSHR, realizing that MSHRs rarely need to immediately push their data into the cache. Our D-Cache was never part of our critical path, so we chose the highest associativity possible.

3.7. Instruction Prefetching

Each instruction executed by our processor must have once been read from main memory. Our instruction prefetcher attempts to amortize this latency by preparing multiple memory requests for future instructions, defined as the prefetch window, and making use of the mem-line when no other higher priority requests exist. Since there is no fear of a line in the I-Cache being dirty and thus, no eviction policy, an instruction fetch memory request being fulfilled can immediately be entered into the cache, following a different priority to D-Cache MSHRs. As a result, we gave the I-Cache its own MSHRs to monitor. While testing, we ran the processor with either a prefetch window of 16 or 32 instructions. After analysis, it was determined that a window of 32 instructions gave us the best performance boost, reducing the CPI by more than half.

3.8. Store Queue

Our store queue works to ensure ordering of reads and writes to memory while also trying to reduce the number of cache and memory requests between load and store instructions. Ordering of writes is achieved by storing dispatched instructions in a store queue; the ability to write to the cache is only permitted to the store at the head of the queue. When entering the store queue, a one-hot bit vector similar to the branch mask is appointed to the store instruction, and the associated bit is turned on in the store queue mask register. All future loads are then labeled with the current store queue mask. After a store has calculated its target address, its associated bit is turned off everywhere it was being referenced.

To achieve ordering of reads, a load instruction must first wait on all older store instructions to calculate their target address before being capable of being issued. This is done by simply waiting until its store queue mask equals zero. After being issued, loads will calculate their addresses and check the store queue for any data to pull. All data transferring was done on a byte-level granularity, which was kept track of with a byte mask. After parsing through the store queue, an aging algorithm was applied to every store with the same target address tag for each individual byte. This aging algorithm would choose the youngest of all the aforementioned stores, ensuring that the most recent write in relation to the load is read. If any bytes still remain to be filled, the load instruction then retrieves them from the D-Cache or main memory.

3.9. Advanced Features Parameter Tweaking

Finally, to determine the best advanced feature parameters, such as prefetch distance, D-cache Associativity, and BHR bits, we ran through various combinations to test for the most optimized parameter combinations. Table 2 illustrates the various combinations our team has experimented with. Figure 5 shows the CPI analysis of different combinations. From this analysis, we were able to conclude that Combination 5 yields the best CPI performance; hence, our processor is finalized with the parameters defined in Combination 5.

Table 2. Parameters of different combinations

Combination	**1**	**2**	**3**	**4**	**5**
Prefetch Distance	32	32	32	24	32
D$ Associativity	8	8	16	8	16
ROB Size	32	32	64	64	32
RS Size	32	32	32	32	32
BTB Ways	4	8	4	4	8
Branch History Bits	5	5	8	10	5
Store Queue Size	4	8	8	8	8
Load Buffer Size	4	8	4	8	8

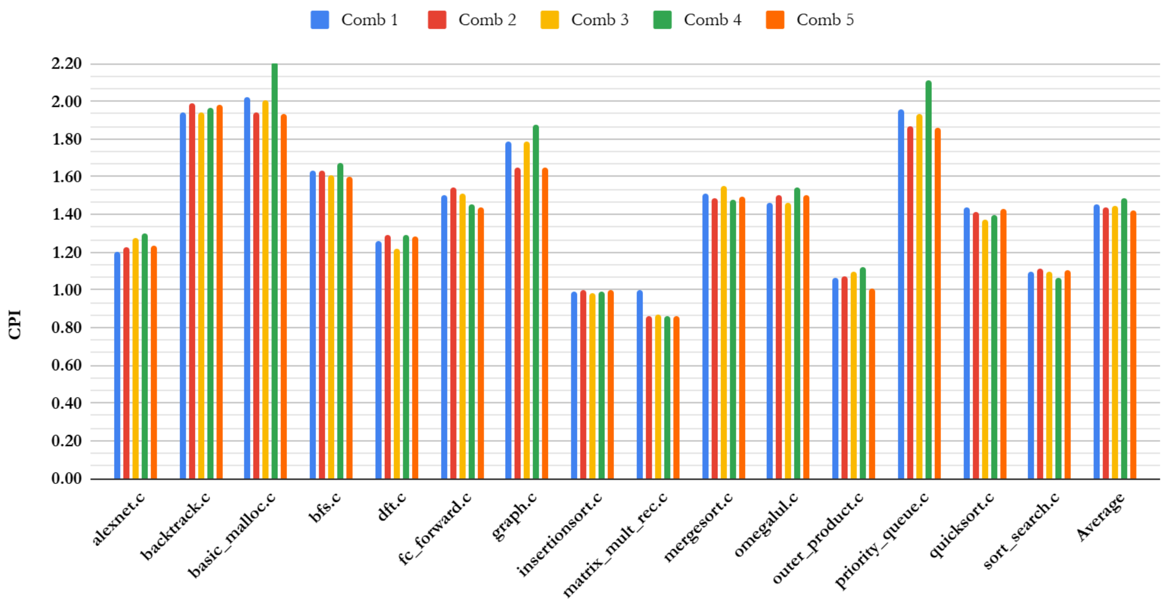

Figure 5. CPI Performance Analysis with different parameter combinations

4. Project Management

Once we had implemented the ROB for milestone 1, we divided into 3 pairs to work on the implementation and testing of the other primary data structures (RS, Branch Stack, and Freddy List). However, beyond that point, almost the entirety of the project was done as a whole group in person. Everyone contributed to the high-level discussions regarding how to design each module, and everyone contributed to translating our designs into algorithms and code. This approach, although somewhat unconventional, worked very well since we always had many perspectives to consider for each aspect of the processor, and it made debugging much easier. In addition, this made the overall experience of the course much more enjoyable since everyone was able to learn how each component worked, and we meshed well as a team.

Our goals were originally centered around the advanced features we wanted to implement, and we all had the understanding that we wanted to aim for a design that was as efficient as possible in terms of performance. Each member made it clear in the beginning that they were willing to put in the necessary work to achieve a top-performing processor, and because of our shared ambition, we collaborated well together and met each of the deadlines we set out to meet.