EECS 470 Final Report

Jeffrey Yang, Grant Hincher, Artem Demenchuk,
Michael Stankovich, John Collison, Carl Beckstrom

April 2025

Contents

1 Design Overview

At the highest level, our processor utilizes an architecture modeled after MIPS R10K, which we decided on because it avoids the repeated mass transmission of large 32 bit values that the P6 style architecture necessitates. We implemented several difficult advanced features to increase performance (N-way Superscalar, Early Tag Broadcast, and Early Branch Resolution), which are discussed in full in their relevant sections below. In addition, some other features included in our processor are a GShare branch predictor, a store queue with load forwarding for memory hazard handling, and split reservation stations based on functional unit type. We made high-level design decisions to incorporate these features, with smaller details and more advanced features interspersed between. A general diagram of our entire system architecture is shown below.

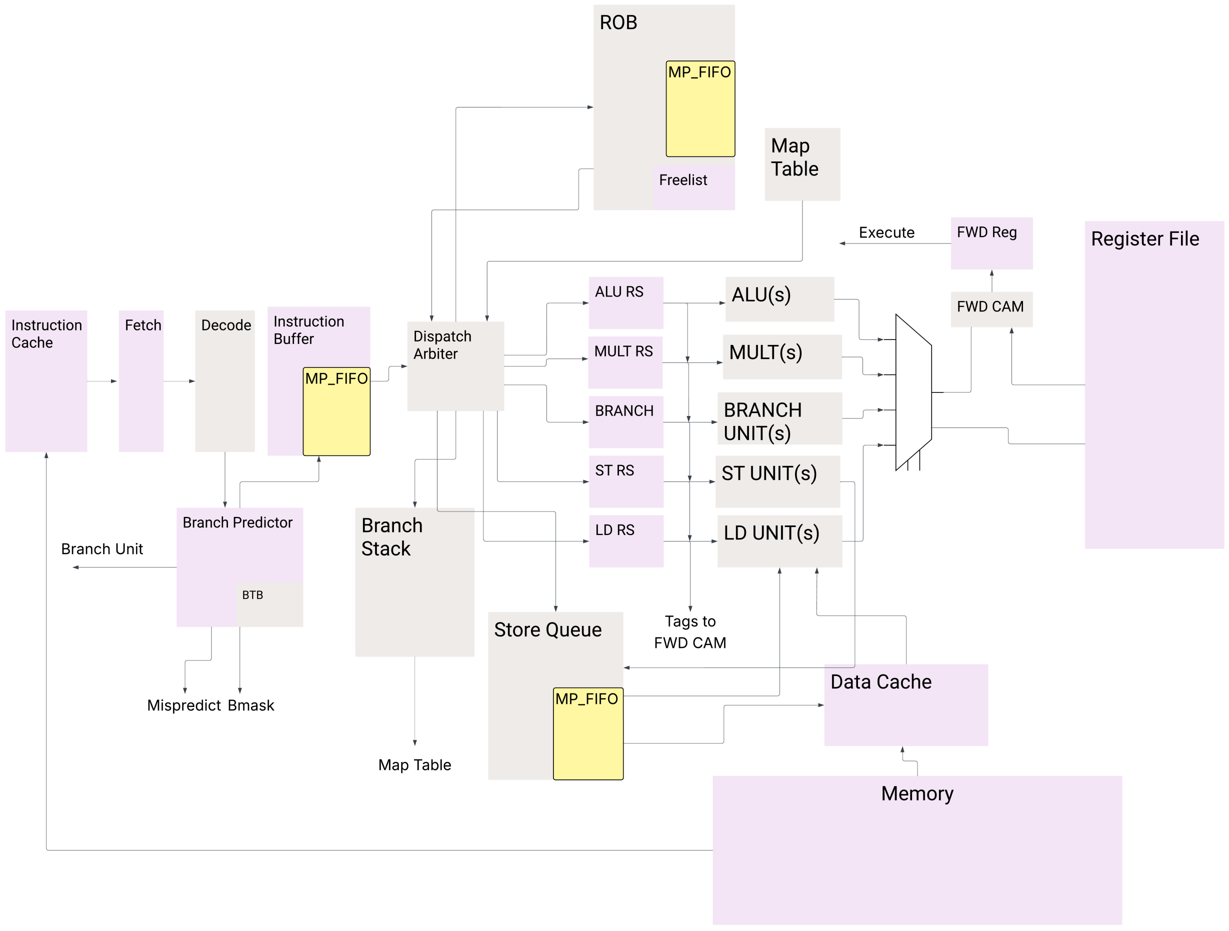

Figure 1: Abstract Full System Diagram

2 Advanced Features

As briefly mentioned before, our processor utilizes an N-way superscalar, with the width definable at compile time. In addition, early tag broadcast and early branch resolution are the other difficult advanced features which we chose to include based on their perceived performance benefit and heavily influence the core architecture of our design. Our exact implementation of each feature will be discussed in the following subsections.

2.1 N-Way Superscalar

Our processor has an adjustable superscalar width, tested up to 5 way. The main effects of this on our design are on the dispatch, issue, and CDB size. All 3 are limited to the set scalar width and give our processor the ability to process multiple instructions per cycle, enhancing the performance of more parallel programs. In addition, due to the blocking and single-banked icache used, increasing way-ness beyond 2 way produces diminishing returns on performance as there are not enough instructions being sent to the function units, so we settled on 2 way as the final superscalar width. The performance metrics for superscalar width can be seen in section 9. Beginning our processor journey, we created the ROB as a FIFO structure. Given the N-way design, the FIFO was designed with N read and write pointers to a multi-port memory module, instead of the typical singular pointer. This N-way FIFO later was revised to be a MP-FIFO (MultiPort FIFO), as the store queue read size was limited by data cache banks, but could have N stores be dispatched on one cycle.

2.2 Early Tag Broadcast

Early tag broadcast allows our processor to resolve dependencies and efficiently forward data from instructions finishing execution to instructions which will begin executing on the next cycle. This eliminates the cycle gap for dependent instructions that would normally be caused by the need for data to write to the register file before it can be read. After investigating the ETB diagram presented in lab, our group discovered the approach had a cycle-gap and lost the benefits of ETB. This oversight temporarily stalled our progress as we had to redesign the ETB to minimize clock period impact whilst allowing for there to be no structural hazard between dependent instructions. To simplify the CDB design, the CDB both gives issue grants and broadcast grants. This means our issue width was N instructions, which was a design choice to limit the amount of read ports on the register file. Given this design, our backpressure had to be carefully considered to allow the multi-stage pipelines to issue and broadcast effectively.

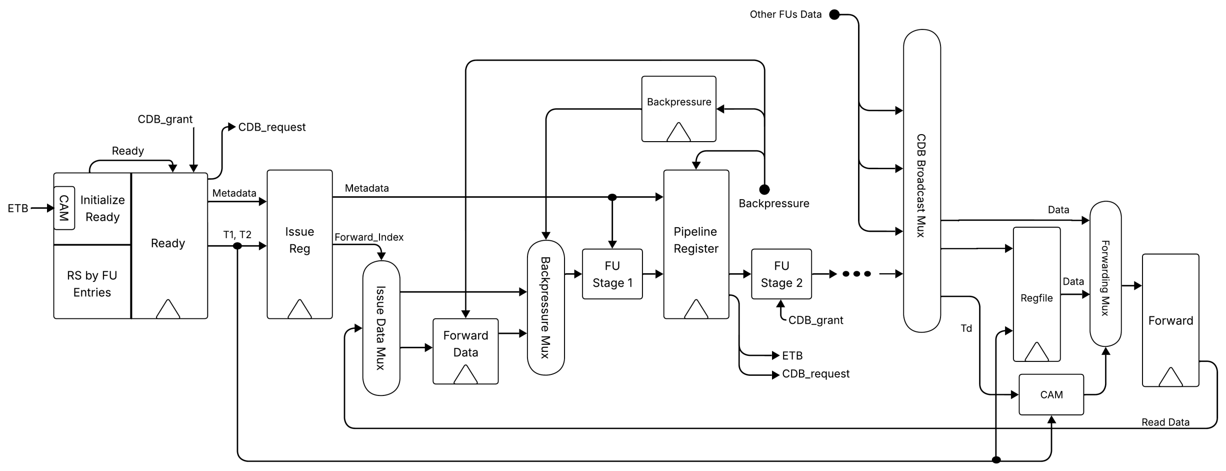

Figure 2: The ETB diagram shows the broad ETB functionality in respect to one functional unit, focusing on the data forwarding design, the decision to mark entries ready a cycle prior to issue in the RS, and the necessary backpressure logic to maintain data correctness within multi-stage pipelines.

2.3 Early Branch Resolution

Early branch resolution gives our processor the ability to process branch results before the branch reaches the head of the ROB. This reduces the impact of mispredictions by allowing the processor to correct its mistake and load in the correct address while the rest of the processor is still occupied with running instructions in the reservation stations. Taking into account the test suite of programs and the GShare style branch predictor we implemented, being able to quickly resolve mispredicts was a desirable feature. Implementing this feature involved checkpointing the state of the map table, the tail of the ROB, and the tail of the store queue in the branch stack. The Branch Stack checkpointed the current state of the Map Table at the beginning of dispatch. On the following clock cycle, the branch stack updates the Map Table that it had previously checkpointed up to the branch instruction, effectively becoming the previous cycle's map table state. The Branch Stack continues to update mappings *ready* by being connected to the ETB. Given that the store queue uses the MP-FIFO module, which was originally created with the ROB in mind, there was improper handling of rollback to an empty state. After debugging, we added a saved full state of the store queue associated with the pointer to correctly rollback the store queue FIFO state.

2.4 Branch Prediction

We utilize a GShare branch predictor with a BTB for branch prediction. This decision was made due to GShare's relatively good performance across a variety of test cases whilst not being too complex to implement. Furthermore, we decided to only fetch up to 1 branch at a time to reduce the critical path as processing more branches at once would have resulted in increasingly more sequential logic in the branch predictor. In addition, we also decided on a bmask size of 4 bits since we chose to incorporate early branch resolution. We noticed a 4 bit bmask size restricted performance in branch heavy test cases when we were running 1 way, which was due to the priority selector logic employed by our CDB arbiter, and saw significant gains from switching to an 8 bit bmask size. However, after switching to 2 way operation, this issue was effectively eliminated and we saw next to no performance gain from increasing the bmask size. The prediction accuracy across all tests hovers around 60%, but a weighted average based on program length is closer to 80% (since many of the shorter tests do not warm up the predictor enough to get good accuracy).

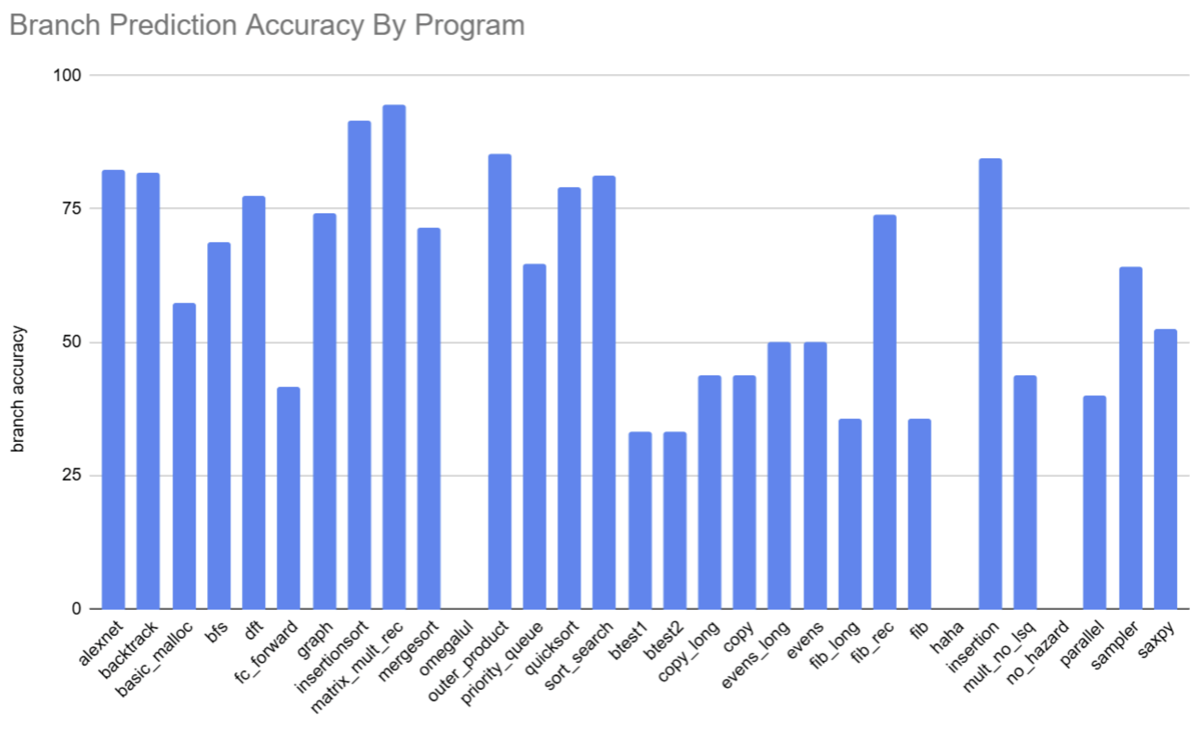

Figure 3: Branch prediction accuracy for different test programs

3 Instruction Memory & Fetching

Due to time constraints, the submitted processor uses the default icache design included with the project. This is a blocking, single-bank, direct-mapped cache with no prefetching functionality. However, at the time of submission, a substantially more advanced icache was nearly complete, with the fully-implemented design passing 13/17 assembly test programs in `programs/*.s` (Table 1). Because of its increased complexity and scope, this report describes the intended advanced icache design in depth, while still mentioning the basic submitted design above for completeness.

3.1 Prefetching

Prefetching in the advanced icache is accomplished by a single 4-block stream buffer, whose start address was reset after each instruction cache miss. More complex alternatives to the single 4-entry stream buffer were considered, such as a pair of 2-entry stream buffers that prefetched at both the taken and not-taken next PC in the case of a branch instruction. However, this proposal was rejected due to project timeline concerns as well as potential performance issues. The de-facto target superscalar width for our design was 4, meaning that in an ideal case with no stalling, the processor would consume 2 memory blocks of instructions per cycle (recall that each memory block contains two instructions). Thus, a 2-block stream buffer would run out of prefetched instructions after just one such cycle, whereas a 4-entry stream buffer could sustain this high-performing regime longer. Furthermore, the processor supports up to 4 nested unresolved branch instructions, making it non-obvious which branch instruction should get the prefetching support.

Program	Basic I-cache CPI	Advanced I-cache CPI
`btest1`	8.229	—
`btest2`	6.982	—
`copy_long`	0.992	0.686
`copy`	1.538	1.288
`evens_long`	1.318	0.804
`evens`	1.687	1.333
`fib_long`	1.235	0.690
`fib_rec`	1.503	—
`fib`	1.533	1.093
`haha`	5.278	1.889
`halt`	15.000	15.000
`insertion`	1.147	1.070
`mult_no_lsq`	1.845	1.300
`no_hazard`	5.500	2.000
`parallel`	1.1650	0.995
`sampler`	5.845	—
`saxpy`	1.642	1.337

Table 1: Comparison between basic and advanced icache design performance on assembly program test suite. All measurements obtained on a processor with zero data memory latency.

3.2 I-Cache Design

In order to support stream buffer prefetchingoogg, a non-blocking icache is necessary. Non-blocking operation is achieved using a miss status holding register (MSHR), which keeps track of the block address, memory tag, and request source (prefetch/cache miss) of every ongoing memory transaction. The included `mem.sv` has a constant delay for each memory transaction, so a simple memory transaction FIFO would have sufficed in this context, but we designed a full-featured MSHR that is also compatible with real-world, variable-latency memory.

The icache itself is set-associative and multibanked. The motivation for multibanking is that only one set within the cache can be read or written per cycle (a limitation stemming from `memDP`), so multibanking allows us to split the icache into several "mini-caches" (banks) which can be read in parallel. As mentioned previously, the de-facto target superscalar width of the processor was 4, so 2 cache banks are used to ensure that 4 consecutive instructions (in 2 consecutive mem blocks) can be read out from icache in parallel. A limited degree of set associativity (4-way) was chosen to decrease the likelihood of cache eviction during branches, where fetch PCs can change greatly and thus conflict with existing instructions in the cache. By contrast, during sequential operation, cache associativity does not improve eviction rate – nearby addresses map to different sets (in the direct-mapped case) or different ways within the same set (in the set-associative case).

4 Dispatch

From fetch, instructions are decoded and sent to the instruction buffer, which utilizes the MP-FIFO module. The instructions can be read from the buffer each cycle by the asynchronous dispatch arbiter module, which handles all the potential structural hazards that are present in our design. It first takes into account all of the space in every module, checking that we can fit what instructions we have. This is done by type since our reservation stations are split by functional unit type. This logic is also weighed against the "orderedness" of the incoming instruction packet, which will be N-wide after synthesis. We cannot dispatch subsequent instruction after space runs out for one type even if they may fit, as this would violate the in order expectation for fetch & retirement. Once this has been done, it outputs instruction packets to the relevant pipeline, tags to the ROB after passing through the map table, store queue information for load forwarding, and new rollback data to the branch stack/checkpoint if there are any branches being dispatched in the N way packet.

5 Issue & Execution

The processor is split into pipelines for instructions utilizing the ALU, multiplier, load unit, store unit, and branch unit. Each has its own reservation station which can issue to a parameterizable number of unique functional units. This number is generally chosen in relation to scalar width, since if we go much beyond this we are creating hardware that we are not utilizing. Ready signals corresponding to the readiness of physical registers are marked to send to issue, with an additional constraint that all stores younger than the SQ tail associated with a load instruction must be processed (executed) before that load can issue. Readiness and sending to issue operate on two separate cycles to cut down on the critical path. They then make their way through the execution pipelines, which are all 1 stage save for the multiplier (4) and the load unit (4). Upon completion, they are marked in the ROB as per usual.

6 N-Way CDB Arbitration & Broadcast

Due to our decision to implement early tag broadcast, the CDB arbiter takes in tags from the reservation stations for single cycle functional units, such as the ALU, and requests from reservation stations and second to last stage of multi-cycle functional units. This allows us to keep our issue width to N as well since we decided to only have 2N issue read ports on our register file. This design decision was made on the assumption that having too many read ports would create a long critical path. After receiving the tags and requests, the CDB arbiter uses a priority selector to choose which requests to grant, allowing instructions to then leave the reservation stations and execute or finish execution in pipelined functional units. The grant created by the CDB arbiter is then passed to CDB broadcast one cycle later, where it is used to determine which functional units can pass their data and destination registers through to the register file.

7 Data Memory

7.1 Store Queue With Forwarding

Upon dispatch, a load instruction captures the current SQ tail as its load tail, identifying the most recent store before that load. To issue the load, every store from the SQ head up to the load tail must have computed its address; likewise, forwarding can only occur from entries in that same range. Each cycle, the SQ broadcasts its relevant data to the load pipeline so loads can forward from pending stores.

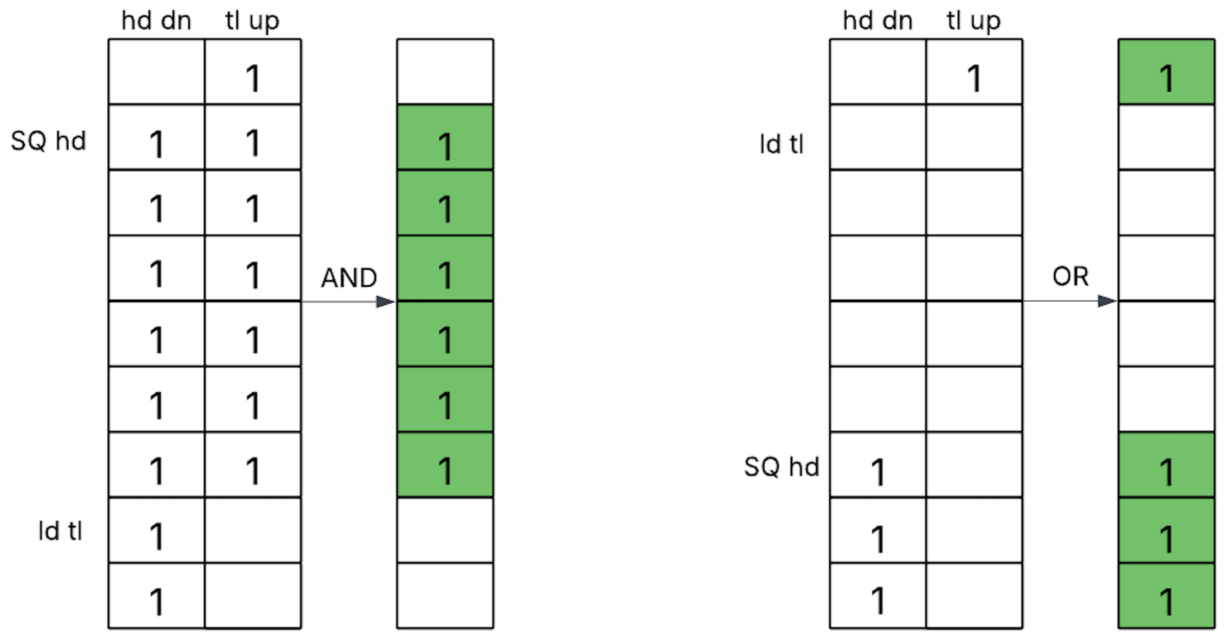

Figure 4: Two SQ forwarding scenarios: (left) SQ hd dwn AND tail up, and (right) SQ hd down OR tail up.

The figure above illustrates the two forwarding scenarios, one where the SQ head is "above" the load tail and one where it is "below" the load tail. We encode each case with two bit-vector arrays, head up and tail down: if the head is above the tail, we bitwise-AND them; otherwise, we bitwise-OR them. The resulting mask precisely flags which SQ entries are eligible to forward data. Next, we perform a CAM lookup on the masked (eligible) entries to isolate only those whose addresses match the load. Finally, to achieve byte-level granularity, four per-byte PSEL units each use this mask to pick the most recent store targeting that byte for forwarding.

7.2 Data Cache & Arbitration

Due to time constraints, we kept the data cache design straightforward: a two-way set-associative, dual-banked structure that boosts memory bandwidth and cuts down on conflict misses. We chose a blocking implementation because integrating non-blocking logic proved too complex and time-consuming. Within each bank, we added byte-level write-enable signals and used eight memDP units per bank, which let us bypass the usual read-modify-write sequence on store retirement. In practice, once data is loaded into the cache, store instructions can update it directly without first reading it back out.

8 Testing Strategies

In the earlier stages of the project, we performed thorough unit testing in order to validate all of our modules before integration. This was done because bugs are easier to trace in a relatively simple and isolated module compared to the full CPU with many possible sources of error. The out-of-order logic components and non-memory pipelines were all unit tested to some extent by the authors of each module. Each module had its own testbench that would test various edge cases and a .vcd file to view the corresponding waveforms. Our goal was to unit test each module as much as possible to avoid complex bugs later in the project during integration, and we were able to accomplish this goal with main modules like the Reservation Station, ROB, Map Table, Free List, and Branch Predictor. However, as the project pressed on and time ran thin, it proved more efficient (seemingly at least) to debug the processor as a whole with each new addition, since it served as a de facto testbench for each new iteration. For memory features in particular, testing was done on programs running through our integrated CPU. Writing testbenches for these modules would have taken away time from debugging the whole system given the time remaining, and we were proficient at debugging the processor utilizing waveforms and a primitive text debugger. We created a shell script to run all of the .c and .s programs on our processor. The script generated ground truth files using the project 3 processor and compared them to the write-back files from our project 4 processor. The script would display all of the tests, whether they passed or failed, and their respective CPI in the terminal. This allowed us to easily run the entire provided test suite without having to individually run each program and diff the .wb files. We ran our CPU on 1-5 ways and with 2 decode side, in addition to running our out of order core on 1-6 ways with 1-6 decode side and direct memory access (zero simulated latency). All of these configurations produced correct output for the given test cases. When looking at the tests, our byte, halfword, and word forwarding from the store queue to the load pipeline was tested through the .c files and we tested them with varying amount of store and load functional units to allow different amounts of stores and loads at once. In addition, we stressed our store queue retirement and load backpressure logic by randomly simulating cache grants to the two modules and running all the test programs. The multiplier backpressure was tested as part of the unit testing done on the multiplier.

9 Analysis & Performance

Our group was behind for the overwhelming duration of the project, so we did not have time to synthesize the processor. As a result, our submitted clock period was an estimate based on the added delay caused by the added icache and fetch modules, which were synthesized individually. Our submission synthesized correctly and yielded reasonable CPI, but we wanted to do further testing to validate our theoretical assumptions. After submission, we recorded more data for a more thorough analysis. Our original submission was clocked at 16 ns, but further synthesis attempts showed a 10.8 ns clock period was possible on a 2 way superscalar by decreasing the amount of each type of functional unit to the scalar width.

Since we had an N-way superscalar as part of our design, we also looked into the performance of different scalar widths as shown in the bar graph below.

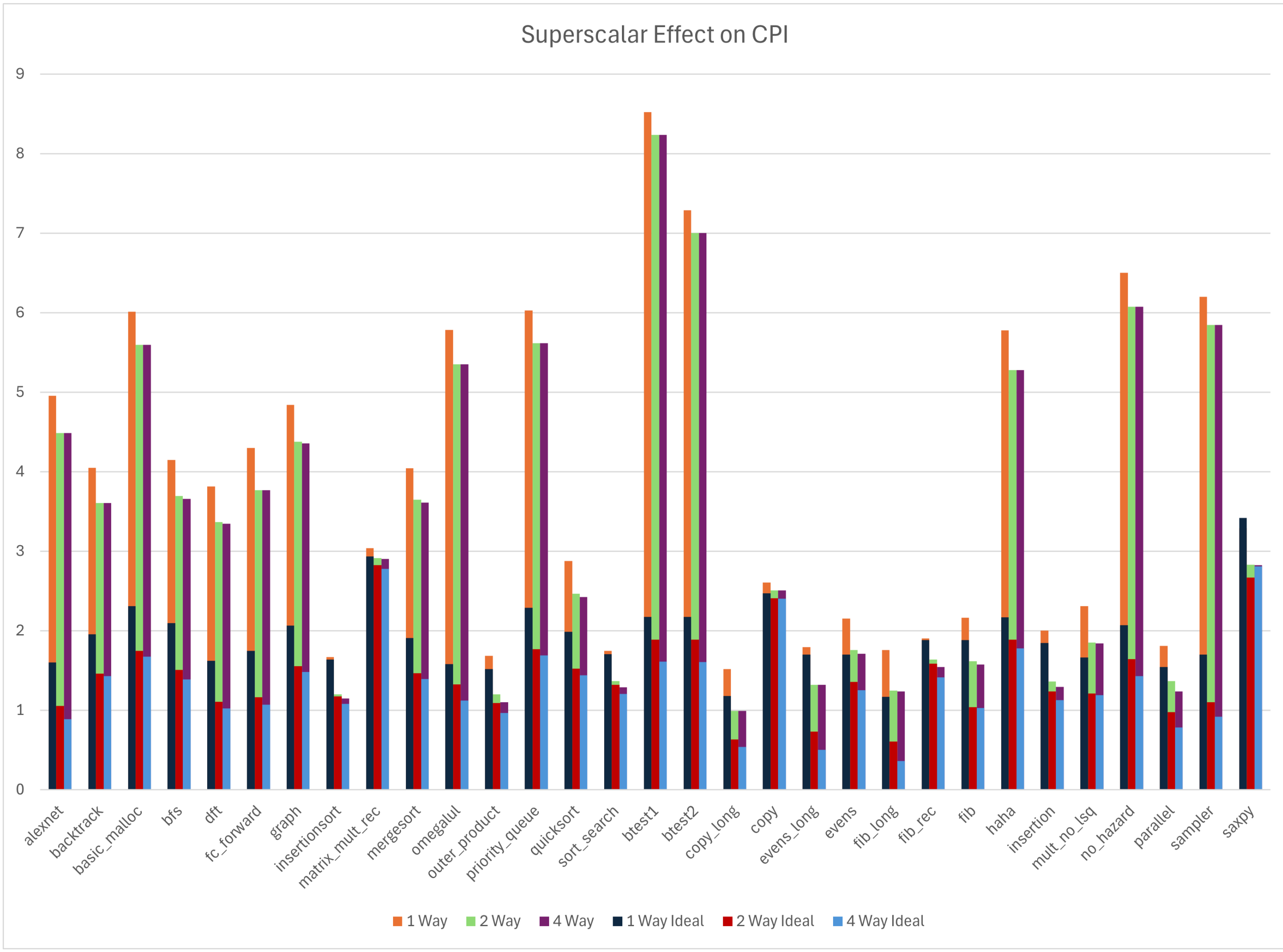

Figure 5: CPI of different superscalar ways

As shown in the diagram, whether running with our icache variant or the direct memory access variant, the performance increase going from 2 way to 4 way superscalar is of a very small amount for almost all test cases across the board. This is likely due to the low level of instruction parallelism in the test suite. From this testing, 2 way appears the most reasonable as it has comparable performance to 4 way while giving a better potential clock period.

10 Project Management

Throughout the project, we effectively divided tasks among group members, set weekly progress goals, and overcame implementation challenges during integration. At the beginning of the project, we split the base out-of-order logic between group members where 1-2 people would be responsible for each module. This approach worked well initially because we were able to have the base features written and tested relatively quickly. However, we ran into issues because we did not dictate exactly how the modules should function. This led to issues during integration, specifically with our reservation station and CDB. Our reservation station introduced an unnecessary delay after tags were broadcast and we had an incorrect original architectural idea for how to implement early tag broadcast. These issues required us to allocate nearly a week to rewriting both modules to ensure that they would work properly with the rest of our processor, which ultimately stalled our progress. However, encountering these issues relatively early on in the project turned out to be a valuable learning experience. It taught us the value of taking our time with high-level architecture design, which resulted in us dedicating more time designing other features - such as the caches and memory arbiter - before coding them. This ensured a much more cohesive and efficient development process moving forward.

11 Key Takeaways

11.1 Communication

Though we often partitioned work, sometimes the different groups would suffer from ineffective communication, leading to implementation flaws. Sometimes interfaces, or timing, would be altered without proper notice, and fundamentally affect another module that was well underway. Also, at times we would be uncertain of the state of our project based off of our individual work, leading to more confusion. Better communication would have remedied some of these issues and probably aided our timeline.

11.2 Deliberate Planning

In the very early stages of the project, we began writing several modules without a clear idea of the interconnections, timing, and its place in the architecture as a whole. This resulted in several rewrites and wasted time as we essentially threw away work put in. Towards the end we started to remedy this by thoroughly depicting the function of all of our new modules and their role in the processor, but the time lost from the beginning proved to be detrimental. This level of planning is necessary in any kind of project with this sort of scope and hierarchical nature.

11.3 Reducing Complexity & Increasing Portability

In this project, we had a recurring habit of continuously trying to optimize and make modules better when the goal should have been to finish them first before proceeding with optimizations. This mindset increased the complexity of the modules and made it take much longer to start integrating. The careful design and implementation paid off in that we did not have to do too much to get a good clock period, but it created a much more tiring and stressful environment with the delays and cluttered logic. The combination of these likely slowed our entire processor down even if it was built on sound, fast premises. Also, we had almost all purpose built modules save for a general FIFO and CAM, as opposed to making things as modular and reusable as possible. Taking the time to find clever, simpler solutions that could be reused was something we struggled to do under the time crunch, and would have paid of for future work as we added more and more functionality.

11.4 Time Management

Time management was absolutely crucial, and we did okay but often took too long trying to write things, leaving us with an unrealistic debugging window. We were able to resolve everything besides the icache, but with improvised and band-aided solutions as mentioned above that hurt our overall understanding of the processor and potentially performance. Leaving enough time at the end of each phase to understand the changes, their functionality, fixes, and optimizations was essential, and it was something we had a hard time doing.

11.5 Iterative Implementation

We put a lot of pressure on ourselves from the beginning to write everything as fast as possible, sacrificing our intuition and design simplicity if need be. This gave us a fuzzy understanding from the get-go, and propagated onward into the project, creating a lot of opaque features and functions for those who were not directly involved in the implementation. These optimizations are important, and several were very beneficial to our project. However, this mental bandwidth could've been saved for the end after implementing all intended features and making sure everyone understands how things are operating. The net performance gain in the end likely would have been better had we a better picture of everything and a correct processor before we began speeding things up.