# EECS 470 Project 4 Report

Group 5: always @(work)

Advait Iyer (adviyer), Arjun Laxman (arlx), Noel Pamenan, (pamenann), Sanidhya Patel (sanmapat), Bradley Schulz (byschulz), Jack Wildes (jwildes)

# Introduction

For our project, we implemented a P6-style N-way superscalar processor. Our overall goal was to minimize stalling whenever possible using techniques such as data forwarding, early tag broadcast, and early branch resolution.

## System Goals

We aimed to be aggressive with our advanced features. We built support for N-way superscalar and early tag broadcast from the moment we began our design, as the reservation station and reorder buffer needed to have these features integrated from the start. Early branch resolution was added after our initial integration as we realized the added complexity of implementing this feature was worth the performance benefits.

With the P6-style architecture, we prioritized decreasing our CPI over minimizing clock period. We anticipated that choosing P6 over R10K would make it easier to debug, facilitating the integration of more complex features. Thus, we could gain a greater performance improvement through a low CPI than what we could achieve by minimizing our clock period.

## System Architecture

Figure 1 documents our high level architecture with the interface wires between each module

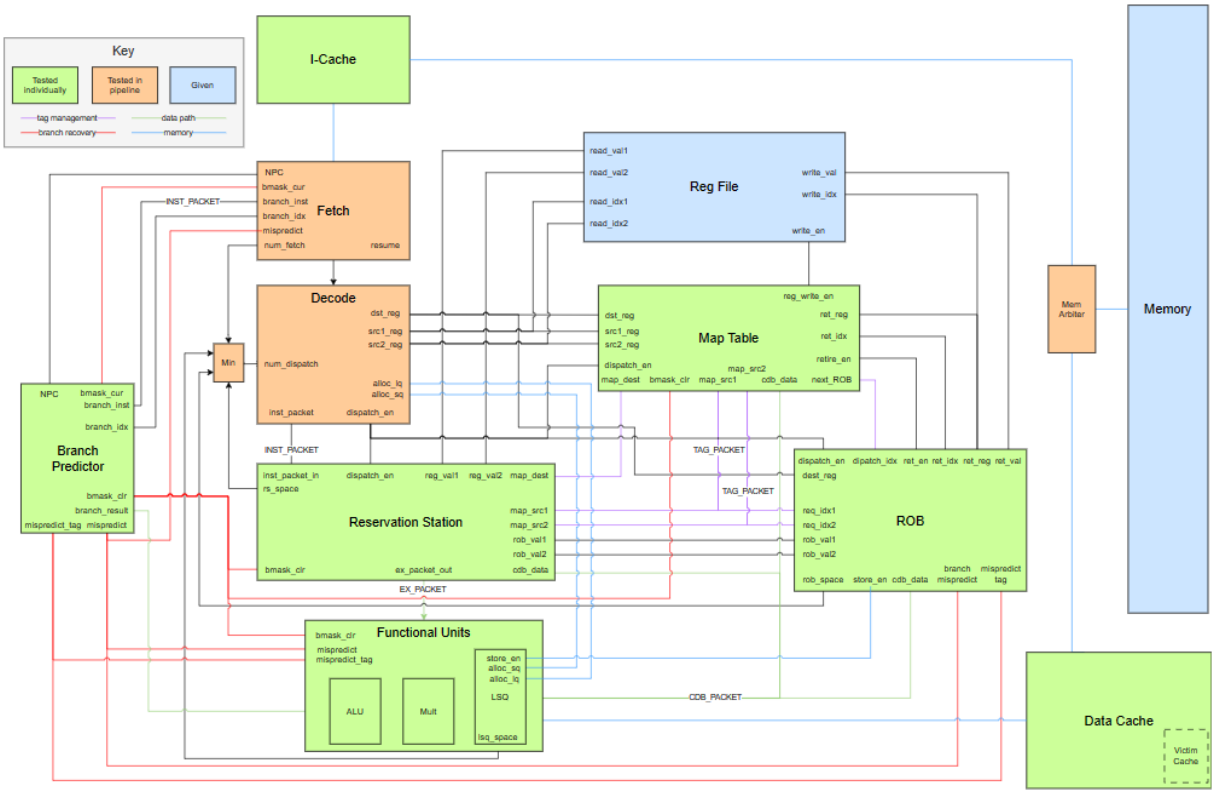


*Figure 1: High level system diagram*

## Difficult Advanced features

We built 3 difficult advanced features into our design: N-way superscalar, early branch resolution, and early tag broadcast. We were successful in building all three into our system. The table below summarizes the advanced features.

**Table 1**: Advanced feature summary

| Feature | Status | Notes |
|---|---|---|
| N-way superscalar | Complete and integrated in final design | Tested for N values 1, 2, 3, and 4 |
| Early tag broadcast | Complete and integrated in final system | Has been working since our first system integration (see Figure 2) Allows for a smaller ROB as instructions get retired faster. |
| Early branch resolution | Complete and integrated in final system | Mostly involved map table checkpointing, rolling back the tail in the ROB and clearing FUs on mispredicts. |

## Simpler features

Table 2 summarizes the simple advanced features we implemented in our design

**Table 2**: Simple feature summary

| Feature | Status | Notes |
|---|---|---|
| Instruction prefetching | Complete | Uses sequential prefetching to continually request instructions following the last PC requested from the fetch stage. |
| Gshare branch predictor | Complete | In our final design, we have 8 bits of history with another 8 for overflow with 2 bits per pattern in our PHT. |
| Issue memory accesses out of order | Complete | Ran into many bugs during integration, but the memory accesses pass all our tests. |
| Store to load forwarding | Complete | Successfully forwards values whenever a load or store with matching addresses issues. |
| Non-blocking L1 | Complete | Operations split across a load buffer, memory |

| | | |
|---|---|---|
| instruction cache | | access buffer, and an MSHR. |
| K-way set associative data cache | Complete | Associativity is parameterizable and tested for K={1, 2, 4, 32}. LRU eviction policy. Ran into many edge cases with memory coming back on the same cycle as a LSQ request. |
| Victim cache | Complete | Fully associative with an NMRU eviction policy. Similar edge cases to the data cache. |

Design choices

We made a few design simplifications to decrease the complexity of our design. The noteworthy ones are described in Table 3 below

**Table 3**: Noteworthy design decisions

| Decision | Relevant Modules | Benefits | Potential downsides |
|---|---|---|---|
| One branch fetched per cycle | Fetch and branch predictor | Only one access to the pattern history table per cycle | Reduced benefits of superscalar for branch-heavy programs |
| Data cache always has priority over instruction cache | Load-store queue, data cache, instruction cache, and fetch | Simple arbitration logic and decreased delay coming from loads that missed in the cache | May stall instruction fetch for memory-intensive programs |
| Loads always have priority over stores in load-store queue | Load-store queue, data cache | Will get loads back from memory sooner which decreases the time it takes for loads to complete | Store queue may fill up with stores that are prevented from going to memory if there are many loads issued consecutively |

# Module Microarchitecture

## Reservation station

The reservation station (RS) is composed of two separate components: a reservation station and an issue register. In the reservation station, dispatched instructions are stored until both their source tags are resolved. When both source tags are resolved, a priority selector chooses whether that instruction will get issued, as we are limited to issuing, at most, N instructions per cycle. Once an instruction has been issued, those instructions are sent to the functional units. Each instruction in the RS includes a b-mask in order for mispredicted instructions to get properly squashed. Despite having an issue register, instructions do not leave the reservation station until they also leave the issue register. We never decided to change this because RS stalls did not occur frequently enough.

Due to early tag broadcast, instructions are ready to issue as soon as their source tags are broadcasted on the CDB. The indices of the matching tags are stored in the issue register and used to route the correct data to the functional units on the next cycle when the data appears on the CDB. Additionally, we need to keep track of the previous CDB packet because instructions that are dispatched may have source tags that should be marked ready if they were broadcast on the CDB in the previous clock cycle.

We do not treat loads and stores any differently than normal instructions besides associating them with a respective load and store queue position supplied by the load-store-queue. This is because the load-store-queue handles most of the logic for loads and stores

We can only dispatch as many instructions as there are spaces available in the RS, so if there is no space in the RS, we stall dispatch. Additionally, the issue register is stalled if there are not enough free load tags in the load-store-queue (which are used to associate loads with memory transactions—our early branch resolution implementation prevents us from simply using the ROB tag due to some edge cases).

## Map table

The map table module comprises two components: a map table and a checkpoint table. The map table stores a tag packet for each register which includes the tag of the instruction in the ROB that writes to that register, a plus bit to denote completion, and a valid bit to identify whether the tag is empty or not. The map table receives either the register or the tag of instructions that are being allocated, completed or retired every cycle and updates the map table entries accordingly. Along with this, the map table also supplies the tags of the source registers as required by the ROB and the RS corresponding to the registers given to the map table from the decode stage.

Early branch resolution requires us to have a checkpoint table. Whenever a branch is dispatched, a checkpoint entry is allocated to that branch and a copy of the current map table is stored along with the tag of the branch instruction. For every subsequent allocation, completion and retirement request, we go through all entries in the checkpoint table and make any change to the checkpointed map table if and only if the update corresponds to an instruction older than the branch. This allows us to maintain multiple states of the map table such that all instructions until that branch are retired.

In the case of a mispredict, the map table is restored to the checkpointed table corresponding to the mispredicted branch and the execution of instructions is resumed from that point.

## Reorder buffer

The Reorder Buffer (ROB) was designed to simultaneously allocate dispatched instructions, complete instructions already present therein, and retire completed instructions, alongside rolling the tail back on a mispredict—all in the same cycle. It achieves this functionality while maintaining precise state.

It has a parameterizable size, with each entry containing an instruction packet (used for debugging and handling some contingencies), its destination register, a valid bit, a done bit (to indicate completion), and its completed value (if applicable).

On dispatch, the ROB allocates up to N instructions to entries past its current tail, wrapping around if necessary. Then, it sends the completed values for entries present therein to the reservation station, as specified by the map table for the instructions dispatched in the current cycle. It also allows for correctly dispatching instructions on the same cycle as a mispredicted tail roll-back. Nevertheless, we decided not to use this feature due to other complications caused by dispatching in this scenario.

On completion, the ROB marks set the ‘done’ bits of corresponding entries to 1. As we implemented early tag broadcast, it sets a variable called ‘prev_cdb_match’ to 1 for the same entries, indicating that a value is expected on the next clock cycle. Up to N relevant instructions are marked as ‘enabled’ for retirement at this stage. These entries can also be dispatched to in the subsequent cycle. ROB ensures that no entries past the mispredicted branch or a halt are marked for retirement, so as to maintain precise state.

At retirement, the ROB marks up to N entries as invalid, depending on how many are enabled for retirement. Then, if any retired instructions are ‘stores’, it sends a signal to the Load Store Queue indicating that the particular store can be evicted from the store queue. This signal is not sent after a ‘halt’ has been retired, to address scenarios in which store instructions are dispatched after the program has halted, which can corrupt our processor’s data memory.

## Functional Units

Our functional units contain N ALUs, N 4-stage multipliers, and a load store queue. We have N ALUs and multipliers to avoid structural hazards when issuing N instructions. Each ALU can resolve branches, so multiple branches can be resolved on the same cycle.

On issue, the functional unit module uses an N x 3N switching network to route the incoming issue packets to the correct functional unit.

To implement early tag broadcast, we use an output buffer that sends the tags of completed instructions out combinationally but the data out sequentially. The output buffer has a tag head and a data head, where the data head is always the tag head on the previous cycle.

So as to thoroughly implement early branch resolution with minimal stalling, instructions older than the mispredicted branch are also squashed at every stage in the functional units, including but not limited to the output buffer and the multi-stage multiplier.

## Load-store queue

We experimented with an alternative load-store queue architecture where loads and stores can issue out of order but must leave the LSQ in order. The goal was to minimize the memory access delay through issuing memory accesses of loads before all prior stores are resolved, but not have to deal with cleaning up mis-speculated loads outside the load queue.

The load queue and store queue entries are allocated when a load or store is dispatched, and the positions in the respective queue are recorded in the reservation station. On issue, the address of the load or store is calculated, any necessary forwarding logic is performed, and load addresses are requested from memory. Stores leave when they receive an enable signal from the ROB to send the store to the data cache on retirement, and loads leave when all prior stores have resolved and it has received all its data, either through forwarding or from the data cache.

The LSQ can forward values from loads to stores whenever a store issue and a load issues. When a store issues, the address and location of the store in the store queue are compared against every entry in the load queue. When a load issues, the load queue sends the store queue head and tail at the time it was dispatched, and the store queue uses that to find the newest store that can forward to this load.

To manage loads and stores of different sizes, all data is kept word-aligned. This makes it easy to forward each relevant byte to a load. When sending data out on the CDB, the load-queue shifts the data so that the requested byte, half, or word starts at the least significant bit.

The LSQ assigns tags to each load so the data returned from the data cache can be sent to the correct load in the load queue. We maintain a free list of load tags so no two outstanding requests to the data cache can have the same load tag at any given point in time.

## Data cache

We made a non-blocking, $2^k$-way set associative, write-back, allocate-on-write cache with a least-recently-used (LRU) eviction policy. It allows for the number of sets to be toggled so as to enable Direct Mapped and Fully Associative implementations. On eviction from the data cache, a cache block is fed into a 32 byte victim cache with a not-most-recently-used (NMRU) eviction policy. The non-blocking feature of our data cache is achieved through a miss status holding register (MSHR). If a memory request from the LSQ misses in either the data cache or victim cache, whether it be a load or a store due to our allocate-on-write policy, this request will be stored in the MSHR until it receives a response from memory in which case the data coming back from memory will be written to the data cache. This allows us to keep receiving and resolving requests from the LSQ even if there is a cache miss. In order to reduce stalls even further, our data and victim caches are also designed to be able to accept multiple requests/evictions in the same cycle. For example, if there is a store to the cache on the same cycle as data from a previous load request is returning from memory, the data cache will allocate two entries on that cycle.

We have a couple of extra components in our cache module as well. There is a memory buffer that sends a single request to memory every cycle if there are any pending requests from the data cache. There is also a load buffer that keeps track of all pending load requests and broadcasts load requests back to the LSQ when they complete, either from a memory response or the data/victim caches.

In terms of stalling, the data cache is only forced to stall if the MSHR is full, if the memory buffer has less than three open entries, and if the LSQ sends it a store request to a set that is full of “unevictable” entries, has dirty data but the cache line is not filled with valid data. In these scenarios, the request from the LSQ is rejected and will be received again in the next cycle.

## Fetch + Instruction Cache

Our fetch logic revolves around an instruction buffer that fills up with future instructions. The fetch stage aims to fill up this instruction buffer and send N instructions to dispatch each cycle.

Our instruction cache is banked to allow us to move N instructions every cycle. The number of banks is the lowest power of 2 above N/2. We chose to make the number of banks a power of 2 so that a set of bits in the PC address can be designated only for indexing into the correct bank.

Branch prediction occurs at the fetch stage so the fetch stage knows which PC to fetch next. We only predict one branch per cycle, so we stop fetching if we receive a branch instruction from the I-cache, and do not fetch any instructions after that branch regardless of whether it is taken or not taken. To implement this, the fetch stage does a preliminary partial decode of instructions by comparing their opcodes against that of a branch, JAL, or JALR instruction.

Since the data cache has priority over the instruction cache, the instruction cache has to request access to the memory system through an arbiter in the top level pipeline. It continually requests the same address until that arbiter accepts the request.

Regarding prefetching, the instruction cache sequentially prefetches instructions following the last requested PC from fetch. It maintains its own memory with the last requested address, and increments that address by 4 every cycle in which it receives a grant from the top level arbiter.

## Branch predictor

The branch predictor module is responsible for providing the next PC to the fetch unit and maintaining the current bmask. The module accepts at most one branch instruction to be used for prediction. This must be the last instruction fetched if it is a branch.

The branch predictor uses Gshare to predict whether a branch is to be taken or not. The main components that comprise the predictor are a history buffer, a pattern history table (PHT) and a branch target buffer (BTB). Our history buffer consists of HIST_BUF_SZ bits of history (taken/not taken) with another HIST_BUF_SZ bits for overflow history consisting of the history of resolved branches only. This overflow is required in the case of mispredicts, when the history buffer may be shifted back using bits from the overflow buffer, so that the current history does not need to be retrained (no repeated warmup). Varying the HIST_BUF_SZ changes the warmup time and in some cases, the mispredict percentage. We use the bits of our history buffer (the currently active half) XORed with corresponding least significant bits of the branch PC to index into the PHT, giving us ($2^{HIST\_BUF\_SZ}$) entries in the PHT. We also have 2 bits of state per pattern in the PHT in order to introduce hysteresis so that we may potentially save a misprediction while breaking out of double nested loops. We use the 6 least significant bits of the branch PC to index into the BTB and obtain the potential branch target. If either the bmask is full, or there exists a branch at the end of the history buffer that is not yet resolved, we stall fetch.

On a mispredict, we roll back the history buffer to the mispredicted branch, correct the target address in the BTB (if it was taken) and broadcast the mispredicted bmask bits to the rest of the pipeline. A noteworthy issue we noticed with our initial implementation was that in the case when two fetched branches (mispredicted target address) with the same PC resolved out of order, the second branch corrected the BTB entry for that branch PC when detected as a mispredict, causing the first branch to not register a mispredict as it appeared as if the address was correct

when comparing against the BTB. This required us to also keep track of the original predicted address taken so that we would not compare against an updated BTB entry.

## Decode

Our decode module acts as a combinational interface between our fetch stage, specifically our instruction buffer and branch predictor, and many of the other components of our pipeline including the map table, reservation station, and reorder buffer. It decodes what instructions are being fed into it, i.e. what source registers each instruction is dependent on, what their destination registers are, if they are loads or stores, etc…, and sends that information in an instruction packet to the modules that need it. For instance, the map table needs to know which registers an instruction depends on, whereas the ROB needs their corresponding destination registers, while the RS needs to keep track of instructions' load and store queue positions. Additionally, we only dispatch one branch— a constraint in our prediction approach. Upon detection, the relevant instruction is sent to the branch predictor so a prediction for the next PC can be made which in turn changes what instructions we will fetch from memory.

Our decode will only dispatch the number of instructions the rest of our pipeline can handle, so as to avoid structural hazards. This is determined by the minimum of a variety of values: the number of valid instructions in the instruction buffer, the amount of space in the load and store queue respectively, the amount of free entries in the RS, and the amount of space in the ROB. Decode dispatches the minimum of any of these values and N.

# Design and Testing Process

Figure 2 shows our high level design process. We had two phases of individual design with integration steps following each phase of individual work. We chose to test fetch, decode, and the memory arbiter mostly during pipeline integration due to timing constraints and the relative simplicity of those components. This followed from our general approach of ironing out modules through unit testing to make integration more seamless.

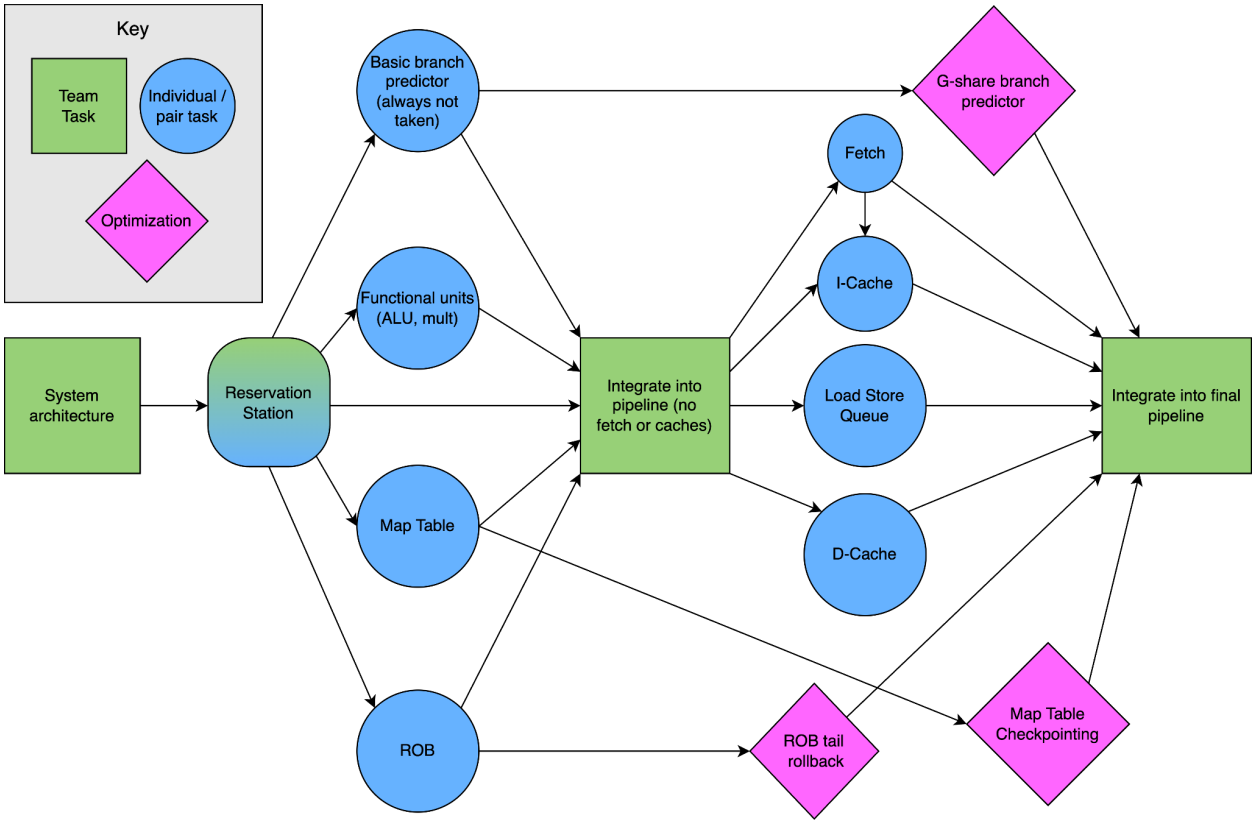


*Figure 2: Overview of design timeline*

## Unit Tests

All main modules were tested individually through targeted test benches. Below is an overview of all the tests we ran for each module.

**Module**: Reservation Station

- Allocate entries
- Multiple instructions ready to issue are dispatched
- Every permutation of source tags receiving values from ROB, register file, CDB, and the previous CDB
- Dispatch an instruction with sources that do not use/need register values
- Dispatch instructions with neither sources ready

- Dispatched instruction has one source ready and one not ready but its tag will show up the cycle after dispatch
- Verify order RS is filled
- Fill entire RS
- Multiple instructions become ready to issue on the same cycle
- Dispatch a RS entry that gets cleared on the same cycle
- Dispatch an instruction with the same source tags
- Clear RS without any instructions being dispatched
- Clear multiple entries in RS with a mispredicted branch
- Clear multiple entries in issue register with cleared b-mask

**Module**: Reorder Buffer
- Dispatch a single entry.
- Dispatch multiple entries.
- Keep dispatching entries until full.
- Complete entries at the head.
- Complete entries not at the head.
- Retire a single entry.
- Retire N entries.
- Retire and dispatch entries in the same cycle.
- Complete and retire different entries in the same cycle.
- Roll the tail back on a mispredict for early branch resolution.
- Roll the tail back at the same time as retiring other instructions.
- Roll the tail back for a mispredicted branch currently at the head.
- Roll the tail back for other head and tail configurations.

**Module**: Decode
- Decode N invalid instructions
- Ensured sources were determined correctly
- Ensured only one branch was dispatched

**Module**: Map table
- Allocate single instruction
- Allocate multiple instructions with overwrite
- Complete and allocate to different register
- Complete and allocate to same register
- Retire and allocate to different register
- Retire and allocate to same register
- Retire and complete different registers (cannot retire and complete same)
- Complete multiple instructions (including overwritten non-existent tags)
- Retire one existing tag and one non-existing tag

- Random tests for allocating, completing and retiring separately.

**Module**: Functional Units

- Issue one packet to the ALU
- Issue N packets to the ALU
- Issue one packet to the multiplier
- Issue one packet to the multiplier for MULT_STAGES consecutive cycles
- Issue N packets to the multiplier
- Issue a packet to the ALU and multiplier on the same cycle
- Issue N packets to the multiplier for MULT_STAGES consecutive cycles and then N packets to the ALU for MULT_STAGES consecutive cycles (to require maximum buffering of outputs to the CDB)
- N-1 alu operations and N-1 multiplies completing on same cycle (to check switching network to CDB)
- Issue and resolve a load (to check integration of lsq)
- Issue 2 stores and a load

**Module**: Branch Predictor

- Send multiple branches
- Resolve (predicted correctly) one branch
- Send a branch and resolve a branch in the same cycle
- Mispredict one branch
- Send a branch and mispredict a branch in the same cycle
- Multiple resolves
- Multiple mispredicts
- Allocate, mispredict and resolve with resolve newer than mispredict
- Allocate, mispredict and resolve with resolve older than mispredict
- Simulate an i<3 for loop in an infinite while loop and check the correct pattern in the history buffer and correct prediction after warm-up.

**Module**: Load Store Queue

- Allocate entries
- Resolve a load not following any stalls
- Issue load that was dispatched after a stall
- Issue store forwards to load
- Issue load that forwards from store in store queue
- Issue another store that forwards to the previously forwarded to load
- Issue 2 stores simultaneously with only one forwarding to a load
- Memory returns a value to a load with a forwarded value
- Store issues that overwrites a value from memory
- Retire stores and see output on memory interface
- Issue two simultaneous loads and ensure memory accesses go out one by one
- Forward between a store and load issued on same cycle
- Clear entries due to a branch mispredict
- Forward between loads/stores of different sizes

- Check CDB output for different sized loads
- 2 stores simultaneously forward to a load word
- D-cache returns load value on the same cycle it is issued

**Module**: D-Cache

- Store writes to a cache line
- Load requests from memory if it misses in the cache
- Any store or load misses in the cache go in the MSHR until their requests are resolved
- Multiple stores to the same set in succession
- Multiple loads to the same set in succession
- Differing load and store request sizes (i.e. byte, half, and word)
- Multiple stores to the same address
- Store misses in caches and is being allocated to the same set as a returning cache line
- Two evictions from the victim cache on the same cycle due to returning data and store that missed in cache
- Fill top and bottom sets of D$
- Load in load buffer gets all of its data from the cache, from memory, and a mix between
- Vary “way” parameter (i.e. test fully associative and direct mapped)
- No unevictable cache lines in a set (stall case)
- Load buffer and MSHR completely fill up

**Module**: I-Cache

- Receive an address from fetch and request instructions from memory
- Have transactions blocked due to d-cache having priority

## System Tests

We had 2 main system tests that we did: our initial test where we integrated our main modules without memory operations, and then the final integration of all the modules.

Our first integration was meant to test our pipeline without any memory interface. We had a simulated fetch stage in the testbench that fed instructions straight to the decode stage, and we used programs that contained no loads or stores. We also had a very simple branch management system composed of a branch predictor that always predicted not taken. This allowed us to test our logic to handle branch mispredictions. Once this pipeline worked to our satisfaction, we entered the second phase of individual module design.

The second system integration was where we added memory operations including the instruction cache, data cache, and load store queue. We also had implemented early branch resolution and our more advanced branch predictor by this point. This phase of integration was much more complicated to get working due to the added complexity of the pipeline and tedious nature of debugging memory operations.

By altering parameters such as the size of the ROB, the RS, the load and store queues, and the number of multiplication stages (alongside N), we exposed the modules in our processor to different configurations of instructions. This revealed multiple shortcomings in the individual implementations and interfaces—another advantage of keeping sizes parameterisable. Given our aggressive aversion to stalling, we greatly benefited from such parameter sweeps, uncovering different types of structural hazards that we had previously discounted.

# Performance Analysis

## Performance Analysis

We achieved a final clock period of 10.7ns in our synthesized pipeline with N=2, ROB size of 12, RS size of 9, 8 bits of branch prediction history, and a 4-stage multiplier. We chose these parameters by balancing CPI and clock period. We made our structures large enough to avoid extra stalling conditions but small enough to keep our clock period lower.

The instruction cache can only get 2 instructions from memory per cycle. This was a bottleneck when starting a program or after branches. Despite all our other stall avoidance techniques, this one was unavoidable given the memory interface we were constrained to.

In the end, our CPI averaged 1.40 across all test programs besides the halt program and the btest programs. Therefore, our average instruction latency is 14.98ns.

We also found that our branch predictor has an accuracy of 67.3%.

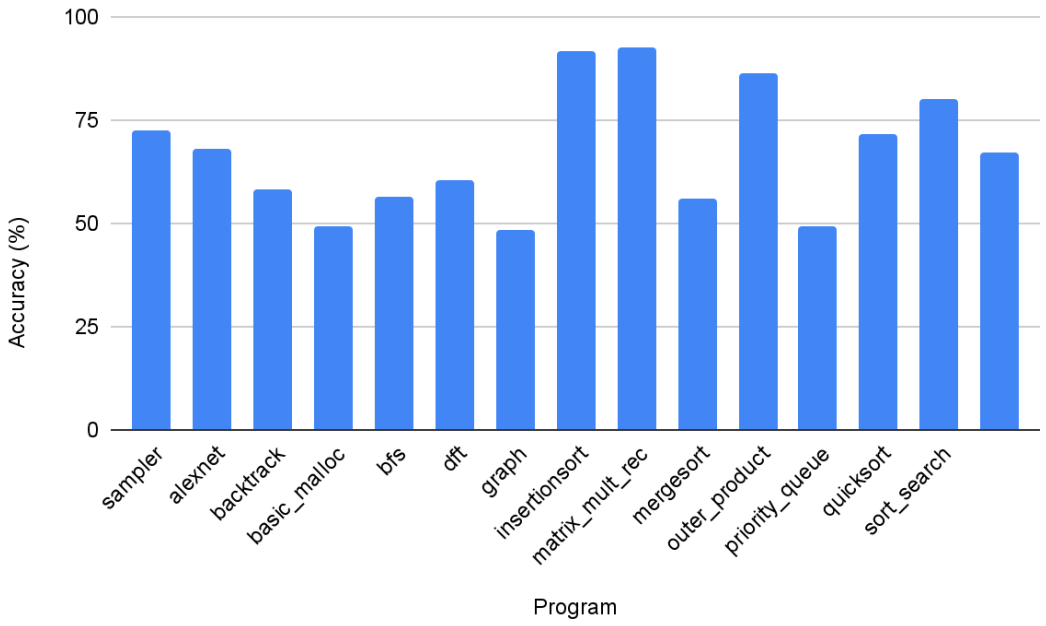


*Figure 3: Graph comparing branch predictor accuracy for example programs*

## Parameter Analysis

We conducted thorough analysis regarding the effects of parameter variations on our overall performance. The specific parameter cases we tested dealt with altering the superscalar value N, ROB size (ROB_SZ), RS size (RS_SZ), size of the branch history buffer (HIST_BUF_SZ), number of icache banks (ICACHE_BANKS), and number of multiplication stages (MULT_STAGES). We did so by tracking the number of stalls caused by different structural hazards for various combinations.

We tested our pipeline with a superscalar value of N = 1, 2, 3, and 4. We found that there is somewhat of a logarithmic relationship between N and performance gain, meaning as N increases, there is exponentially less gain in performance (CPI). There are relatively large performance gains going from N = 1 to N = 2, but the performance gains drop off with much larger values of N after that. Figure 3 below depicts this variation in performance between the different values of N for six example programs in graphical form. In the graph, the other parameters are held constant while N is changed; ROB_SZ is set to 16, RS_SZ is set to 12, and the number of icache banks is set to the lowest power of 2 above N/2.

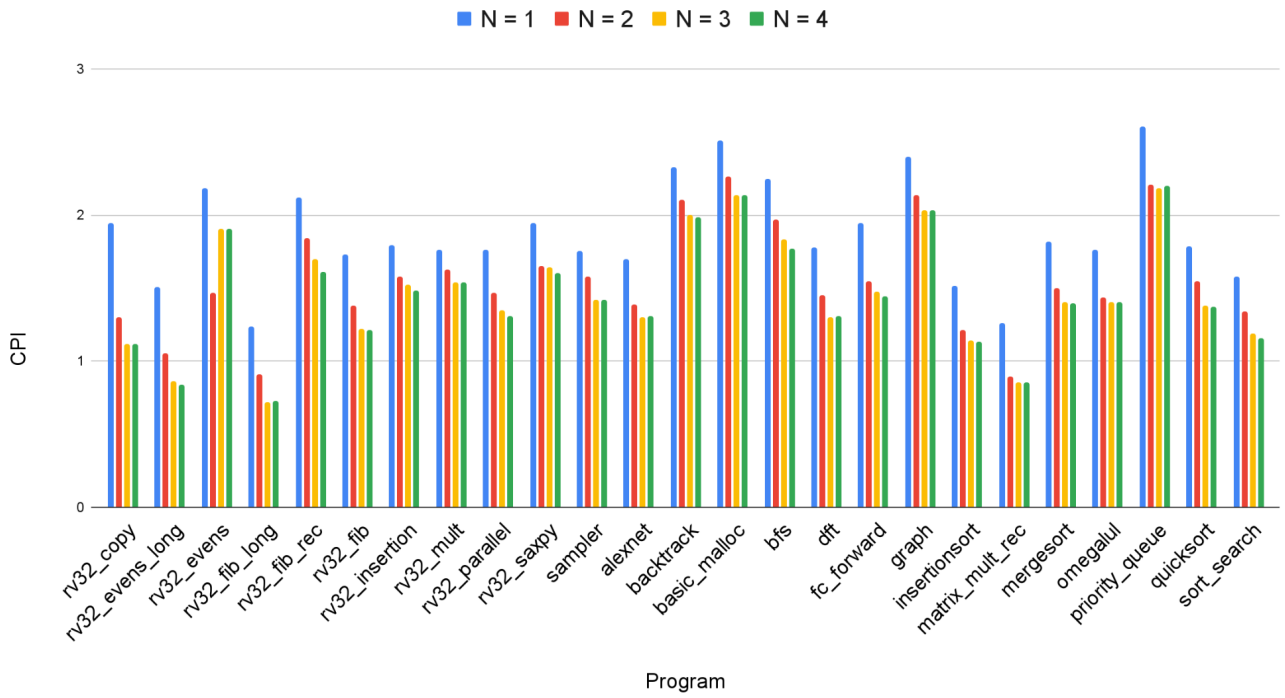


*Figure 4: Graph comparing CPI for varying values of N for example programs*

We additionally tested our pipeline with many different ROB_SZ-RS_SZ combinations, ranging from ROB_SZ = 12 and RS_SZ = 9, to ROB_SZ = 32 and RS_SZ = 24. In our findings, we observed a general trend where larger ROB and RS sizes led to lower CPIs. Specifically, as

instructions retired faster due to early tag broadcast, we could set our RS size to be close to ¾ of our ROB size, as opposed to the more commonly adopted ½. However, we also realized that larger ROB and RS sizes correspond to a higher minimum clock period to meet slack—our critical path is during dispatch, when instructions have to pass through the map table, then the ROB, and finally, to the RS in the same clock cycle. Hence, we decreased our ROB and RS size to 12 and 9 respectively, which with N = 2 gives us a clock period of 10.7ns and an average CPI of around 1.4—our lowest time per instruction (14.98 ns).

Table 4 shows the final parameters we decided upon for our processor.

| Final Parameters Used In Our Processor | | | |
|---|---|---|---|
| N | 2 | Fetch Width | 4 |
| Rob Size | 12 | Multiplication Stages | 4 |
| RS Size | 9 | Load Queue Size | 10 |
| Branch History Size | 8 | Store Queue Size | 10 |
| BTB Size | 64 | D$ Size | 256 |
| BMask Size | 4 | D$ Cache Lines | 32 |
| No. of ALUs | N | D$ Block Size | 8 |
| No. LSQ FUs | 2*N | D$ Sets | 4 |
| No. of Multipliers | N | D$ MSHR Size | 15 |
| I$ Size | 256 | D$ Mem Buffer Size | 15 |
| I$ Cache Lines | 16 | D$ Load Buffer Size | 10 |
| I$ Banks | 2 | V$ Cache Lines | 4 |
| I$ Stride Length | 3 | V$ Block Size | 8 |

*Table 4: Final Parameters for our Processor*

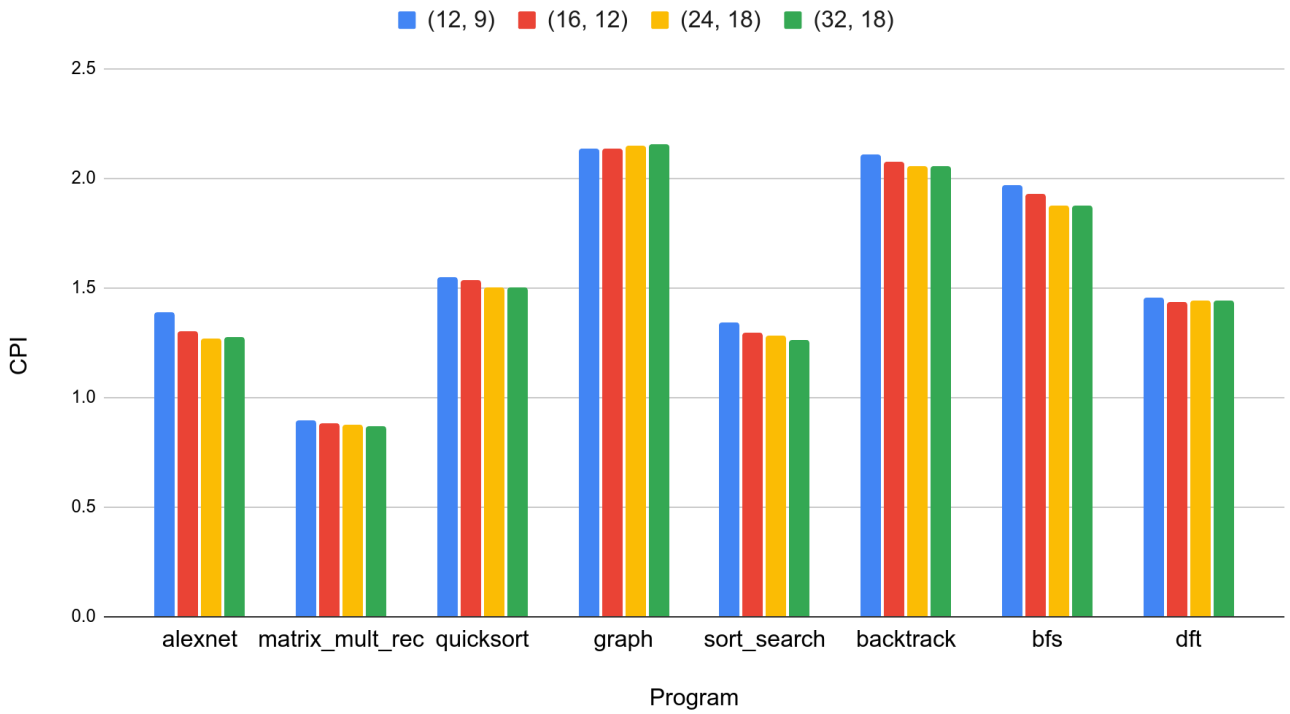


*Figure 5: Graph comparing CPI for Different Values of (ROB_SZ, RS_SZ)*

We also tested our processor performance with two different values for the number of banks in the icache implementation. We tested with icache banks = N and icache banks = the lowest power of 2 above N/2. The larger number of banks present with the N banks implementation was determined to be unnecessary; using the lowest power of 2 above N/2 banks provides the most efficient performance for our processor in the context of icache operations.

Next, we tested different numbers of mult stages for the mult module. We analyzed performance for values MULT_STAGES = 2 and MULT_STAGES = 4. After obtaining correct functionality for these two values, we determined that MULT_STAGES = 4 provided more of a benefit for the latency of our multiplication operations and therefore better enhanced our overall processor performance.

The number of history bits in the branch predictor was also varied. A lower value improved performance in shorter programs due to a shorter warm up time, but we opted for a larger value due to its significantly better performance for larger programs.

## Timing Analysis and Critical Paths

Initially, our main critical path was for loads where they passed through our functional units to calculate the address and then indexed into the data cache to perform the lookup in the same cycle. This was due to unnecessary combinational logic, which we rectified by dealing with

loads and store requests in our data cache one cycle after receiving them from the load store queue.

After this optimization, our critical path switched to the dispatch stage. A large combinational path starting from fetch traces through decode, proceeding through the map table (to determine dependent ROB tags), then the ROB (to obtain the corresponding values, if applicable), before ending at the RS (where the ROB values are issued to FUs in the next cycle). This varies significantly with ROB and RS sizes.

We also have a critical path from our PC, through the instruction cache, and back to the memory buffer in the fetch stage. Toggling the size of the branch history buffer helped bring this down, but the drop in performance for larger programs exceeded the marginal reduction in minimum clock period. Moreover, accessing the instruction cache is similar to the data cache because it is banked and directly mapped, but it is still a large data structure to index into, making the operation hard to optimize.

When we perform 2 stage multiplication operations instead of 4, our multiplier falls on our critical path. This marginally improves our CPI, but as multiplication operations aren't a significant portion of most programs, the increase in minimum clock period obviates any potential benefits.

# Application And Conclusion

## Design Summary

In summary, we implemented an N-way superscalar processor with early tag broadcast and early branch resolution, characterized by an aggressive aversion to stalling.

## Societal impacts

Our design aims to minimize stalling to maximize pipeline usage at the expense of the clock period. Since the clock consumes a large amount of power in a typical processor, a slower clock and lower CPI can reduce the overall energy required to run a particular program. Therefore, if produced at scale, we can help address climate concerns through reduced energy consumption.

Using early branch resolution, we can also minimize the number of wasted cycles by correcting the execution path as soon as possible. Again, that reduces the amount of energy wasted on unwanted instructions. Our consequently smaller modules also reduce the amount of silicon used.

## Lessons Learned

Towards the end of the project, we faced a large number of bugs arising from the complexity of the design. The final integration would likely have been significantly easier if we had done the following:
1) Formalized our high level interfaces prior to writing individual modules
2) Reduced the complexity of our design by accounting for acceptable stalling conditions. Instead, we tried to handle every single edge case using extensive (and computationally expensive) combinational logic.

Regarding our high level interfaces, we had to resolve multiple bugs regarding branch prediction because there was ambiguity about whether it happened at fetch or dispatch. Modifications to our design in the midst of debugging weren't propagated across modules consistently. This resulted in some issues at the interface. For example, we were predicting our branches at fetch but checkpointing our map table at dispatch. Some other examples of module interface mismatch included the load tags between the LSQ and data cache, and whether we were clearing mispredicted instructions using a bmask clear signal or the tag of the mispredicted instruction. Assertions in code regarding expected input behavior (which we used in some modules) could have mitigated a significant portion of such issues if used across the board.

In terms of complexity, we found that we made our memory architecture more complicated than it needed to be. Both our LSQ and data cache were designed to handle many edge cases without stalling. The complexity of these modules, paired with the difficulty of debugging memory operations, caused many delays in our final integration. If we had instead chosen to prioritize

simplicity over minimizing latency, our final debugging process would have been much smoother. For example, store to load forwarding would have been much simpler if we stalled loads in the reservation station, not a separate load queue, while waiting for prior stores to resolve. The CPI benefit of minimal stalling might have been superseded by that of a lower minimum clock period if we had employed a more flexible approach to stalling from the onset.

## Next Steps

There is significant room in our design to optimize our clock period. Since the complexity of the design required extensive debugging, we did not have much time to optimize our critical paths. Unfortunately, much like our synthesizer at times, we could not meet (the deadline) slack with regards to making improvements to some of the areas we had noted.

For instance, our aforementioned critical path is during dispatch. Breaking this into two cycles (i.e. sending the ROB values in the same cycle as instructions are issued to the FUs) will enable us to reap the benefits of larger ROB and RS sizes with lower clock periods.

Additionally, there is a large amount of not only convoluted, but redundant combinational logic currently present in our processor. For example, in the data cache, there is a load buffer, a MSHR, and a memory buffer. These components all share much common data so it would have been better to combine all three components into one. Calculations to determine cache lines for allocation and eviction can also be optimized.

Finally, we delivered consistently higher CPI for programs with many function calls (e.g. graph.c and backtrack.c). Adding a RAS to our branch prediction infrastructure would significantly improve our performance for such programs.

## Acknowledgments

We would like to thank the course staff of EECS 470 for their support throughout the project. This has been a great learning experience for all of us, and we are grateful for the opportunity.