EECS 470 Final Project Report

Griffin Schenker
University of Michigan
gsschenk@umich.edu

Shinjo Satoh
University of Michigan
ssatoh@umich.edu

Yash Sinha
University of Michigan
ysinha@umich.edu

Abraham Vega
University of Michigan
abevega@umich.edu

Elaina Mann
University of Michigan
elainamn@umich.edu

Abstract

Our project is an implementation of an N-way superscalar R10K-style processor. Our optimal configuration (3-way) results in a clock period of 8.44ns and an average CPI of 1.003. This report details our architecture, advanced features implementation, and their performance impact across various configurations. It will also cover the methodology for testing and the way we tackled various bugs.

I. Introduction

Our design process began with a conservative approach, prioritizing completeness and correctness before shifting our focus to optimization. This report mirrors that progression, detailing how we first ensured that our processor functioned reliably and then refined it into a model that we are proud to present. We'll start off by covering our base features along with the architecture of our processor. We will then go into our memory features, advanced features, and our optimizations.

II. Architecture Overview

We have designed an out-of-order N-way superscalar R10K-style processor, with our optimal configuration being 3-way. Our architecture incorporates advanced features, such as GShare branch predictor with BTB, early branch resolution (EBR) with fast branch recovery (FBR) and a sophisticated memory system. The memory hierarchy features a non-blocking banked instruction cache with prefetching and a non-blocking banked data cache with a victim cache. Our store queue implementation supports byte-level store to load forwarding.

III. Out-of-Order Processor

This section outlines the design of our R10K-style processor, developed to sustain a clock period of 8.44 ns in synthesis while maintaining a CPI of 1.003.

A. Instruction Cache

We have implemented a 32-line non-blocking direct-mapped instruction cache (Icache) that continues to serve memory requests while handling misses. The cache features a banked structure that allows simultaneous access to multiple banks, which supports a wider instruction fetch of up to twice the number of instruction banks. Our design also incorporates prefetching to hide memory latency.

B. Instruction Fetch and Buffer

Our Instruction Buffer is a 16-entry array that accumulates fetched instructions from the instruction cache and sits before the decoder. Our fetch system supports wide fetch up to 4 instructions, enabled by the 2 Icache banks. Fetched instructions pass through the branch predictor before being placed in the Instruction Buffer. Our processor supports fetching 1 branch per cycle, and up to the second branch if the first branch was predicted not taken. For dispatch, the instruction buffer takes into account structural hazards posed by the RS, ROB, SQ, as these are all structures that become utilized during dispatch. Additionally, our processor has a restriction that branches and stores may only dispatch if they are the first instruction to do so in that cycle. We kept this invariant for simplicity and due to interactions with FBR checkpoints and the store queue, respectively.

C. Branch Target Buffer

Our Branch Target Buffer (BTB) is a 128-entry direct mapped cache that uses the PC of an incoming branch to speculate the target address. We allow for the BTB to forward the target address from a completing branch to an incoming branch if the PC's are a match. We determined 128 as the optimal size by making it as large as possible until it became the critical path. However, we did see that anything over 128 did not boost performance noticeably.

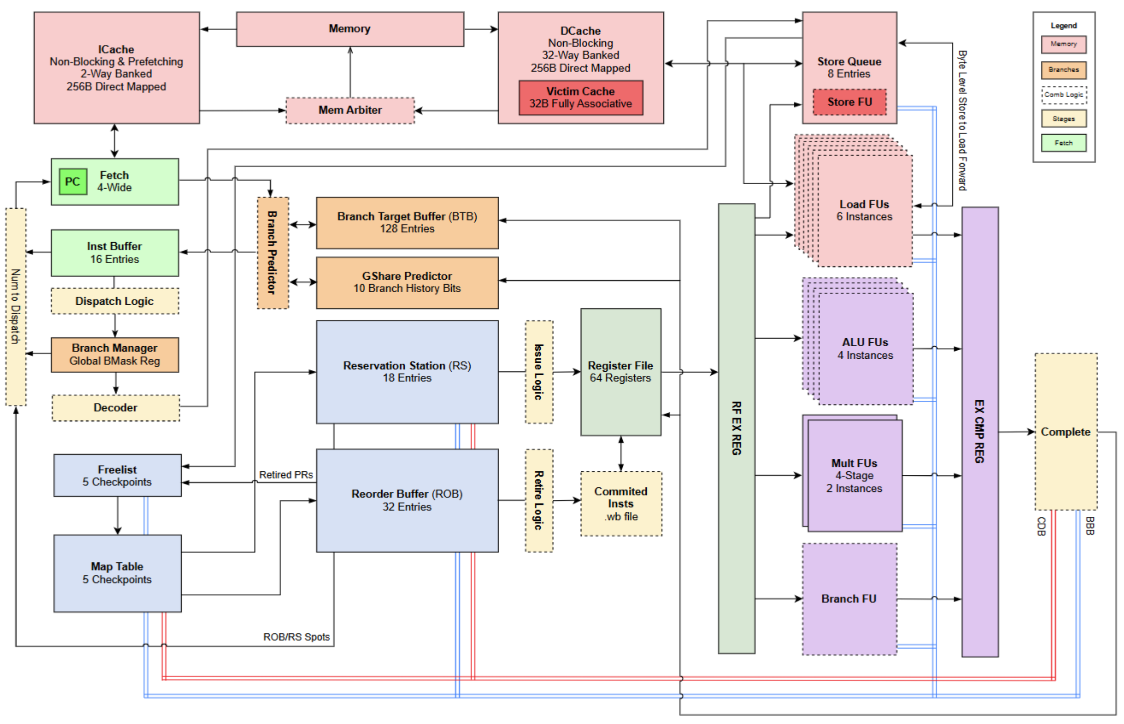

Figure 1: Architectural diagram of final design

D. Branch Predictor

The branch prediction model that we implemented is a GShare predictor that uses a 10-bit branch history register (BHR). The BHR is xor'd with the lower ten bits of a branch's PC (excluding the bottom two PC bits). The result of this is used to index into a pattern history table (PHT), which holds 2-bit predictors for every possible index. Ten bits were chosen for the BHR by testing the branch prediction accuracy. We tested the initialization of the 2-bit predictors as well, and ultimately chose to initialize them to weakly not taken.

E. Store Queue

Our store queue (SQ) was implemented as an array of 8 entries. Stores in the queue dispatch and retire in order, but they may execute and complete out-of-order. The store queue also serves loads with forwarded data.

F. Reservation Station

The reservation station (RS) is the core of the out-of-order capabilities of the processor. It can store 18 instructions. The implementation is divided into many tasks, namely complete, issue, and dispatch. In order to manage reservation station entries we: update the done bits of entries based on the Common Data Bus (CDB), determine which instructions are ready to issue, use priority selectors over the entire reservation station to issue instructions to available functional units, and dispatch new instructions into free available slots in the reservation station. Our processor supports bypassing complete to issue, meaning woken up instructions may issue in the same cycle. The processor supports issuing as many instructions as there are functional units. Dispatched instructions set flags in the reservation station to indicate which functional unit type they use, which is helpful for issue logic.

G. Reorder Buffer

Our reorder buffer (ROB) was implemented to hold up to 32 entries and as an array. Our implementation includes: retiring up to N instructions that are at the head of the rob and ready, marking renamed destination registers as completed based on the CDB, unraveling the tail pointer based on branch miss-predictions, and dispatching up to N instructions. In addition to its general instruction flow, it also communicates with the store queue to let it know when a store is at the head of the rob so stores in the store queue can request memory access and retire.

H. Register Renaming: Freelist & Map Table

Our Freelist and Map Table handle register renaming by dealing with data dependencies while preserving program correctness. When instructions dispatch, the freelist allocates physical registers and the maptable updates register mappings accordingly. During retirement, the freelist reclaims old physical registers. However, we are only allowing those physical registers to be reused for dispatch on the next cycle in order to let retire and dispatch execute in parallel, decreasing our clock period.

I. Register File

Our register file (RF) contains 64 entries that store the data values. It supplies operand values to functional units based on physical register numbers of instructions issues from the reservation station. Retired instructions with destination registers also wire through the register file to produce write-back output. The register file is written to by completing instructions with destination registers. The register file has internal forwarding, allowing issued instructions to read the register file with the correct values before being stored in the rf_ex pipeline register.

J. Execute

Instructions from the rf_ex pipeline register enter execute and navigate to the specific functional unit they need. The instructions are categorized into five types, each requiring different functional units: alu, mult, load, store, and branch. All functional units are single-cycle except the mult, which is fully pipelined and 4-stages, allowing different instructions to occupy different stages of the same unit. Our final configuration for functional units were 4 alu, 2 mult, 6 load, 1 store, 1 branch. Instructions that completed execution funneled into the ex_cmp pipeline register to begin the complete stage on the next cycle.

K. Load Functional Units and Data Cache

Our data cache (Dcache) is 32-lines, non-blocking, direct-mapped, 32-banked and given priority to memory accesses over the Icache. Loads were given priority over stores for memory accesses because there may be dependent instructions on loads. Getting performance out of non-blocking caches calls for allowing several loads to query the cache/memory system despite cache misses.

Instead of having a load buffer in the functional unit and using complex logic to decide which load gets to query the cache, we decided to have a large number of load functional units and allow all loads to query the cache in parallel. This was beneficial as all pending loads were considered for a cache hit. Direct-mapped cache hit logic is lightweight, so it was suitable for evaluating cache hits on several requests. Performance was further improved by making each line in the Dcache a separate bank. This feature required little logic extended from a single-banked cache and provided the benefit of all cache hits being serviced.

L. Complete and Retire

Instructions in the ex_cmp pipeline register enter the complete stage in order to write to the register file and notify dependent instructions that they may issue. The CDB is limited to the superscalar width of the processor. Since there are more functional units than entries in the CDB, backpressure is essential for the path from the reservation station to the ex_cmp register. Our processor implements backpressure as a ready-valid handshake. Instructions in ex_cmp that weren't granted a spot in the CDB drive a busy signal. This signal propagates back to the functional units, which will assert their own busy signals if both, the incoming busy signal and the instruction itself is valid. This system propagates back to the reservation station, eventually reducing the number of functional units that are available if necessary. Our design enforces that stores don't have destination registers and branches are prioritized for the CDB, so backpressure was only implemented for alus, mults, and loads.

All instructions that successfully enter complete will update the map table to reflect that the registers are ready, notify the ROB of which instructions are ready to retire, and wake up dependent instructions in the RS. Completed branches and stores have further interactions that will be discussed in section IV. For the retire stage, the ROB iterates from the head sequentially to clear instructions if they have completed. This logic has high latency due to its sequential nature, so we made the decision not to let freed ROB entries during retire be reused for dispatch in the same cycle, which reduced our critical path.

IV. Advanced Features and Analysis

This section outlines the advanced features that we have implemented for our processor and the various performance testing and analysis for each. We chose to combine advanced features that would give us the best return on investment. We also were open to changing our plan on the fly if a feature was not performing as we expected in our system.

A. N-way Superscalar

Our processor is N-way superscalar, capable of dispatching, issuing, executing and retiring up to N instructions per cycle. While some pipeline stages could technically handle more than N instructions simultaneously, the CDB width is set to N, which serves as our processor's limiting factor. This design balances performance and hardware complexity while enabling significant instruction-level parallelism (ILP).

We analyzed the effects of the N-way on our processor's performance in Figure 4. We discovered that moving from N=1 to N=2 results in a dramatic improvement, reducing the overall CPI in all programs from 1.321 to 1.051. The N=3 configuration achieves our optimal performance with a CPI of 1.003, while N=4 shows diminishing returns with a slight regression of the CPI of 1.005. This clearly justifies our decision to choose N=3 as our optimal configuration.

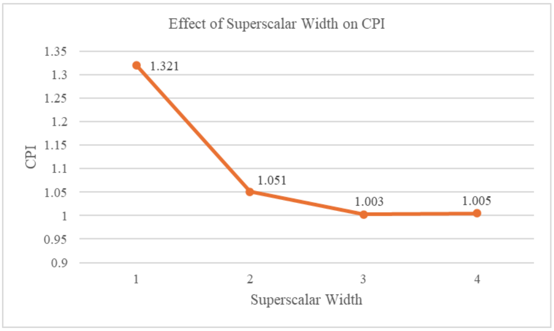

Figure 2: Overall CPI across N-way with values 1,2,3 and 4.

We also analyzed our CPI across six diverse benchmarks such as alexnet, dft, insertionsort, outer_product, quicksort and sort_search in Figure 5. Compute-intensive workloads like dft and quicksort showed small improvements with a higher N, maintaining high CPIs even at N=4. However, programs with higher ILP such as insertionsort and sort_search scaled effectively with higher N, achieving CPIs below 0.8 at N=3. The minimal performance improvement between N=3 and N=4 across all benchmarks indicate that additional hardware complexity and a higher clock period for N=4 would not justify the negligible performance gain for our workloads.

B. Early Branch Resolution

Our processor has early branch resolution which minimizes the performance penalty of branch mispredictions. We maintain checkpoints in both the Maptable and the Freelist for each branch, allowing fast branch recovery when mispredictions occur. For implementation simplicity, branches may only dispatch if they are the first instruction in the cycle.

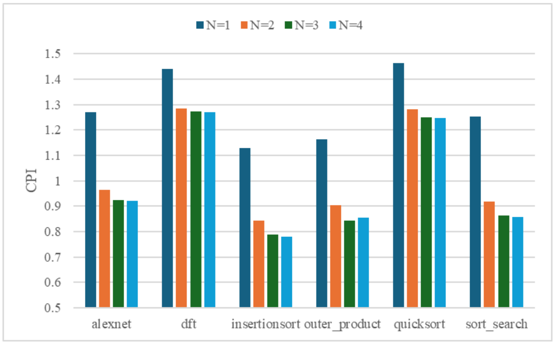

Figure 3: N-way effects across six large programs

To implement this, instructions in the RS and in execute store a bmask. The bmask reflects which branches a particular instruction depend on. When a misprediction is detected, a branch ID (bid) broadcast immediately triggers instruction squashing throughout the pipeline. The ROB and Store Queue simply adjust their tail pointers to squash instructions, whereas the RS compares the broadcasted bid against all instructions and squashes only those affected by the mispredicted branch. If a branch completes but was predicted correctly, then instructions dependent on that branch update their bmask to reflect that they are no longer dependent on it anymore.

For simplicity, we utilize a single branch functional unit that receives priority on the CDB. We discovered that we save about 4.89 cycles per branch misprediction on average by having mispredictions handled during the complete stage as opposed to retirement.

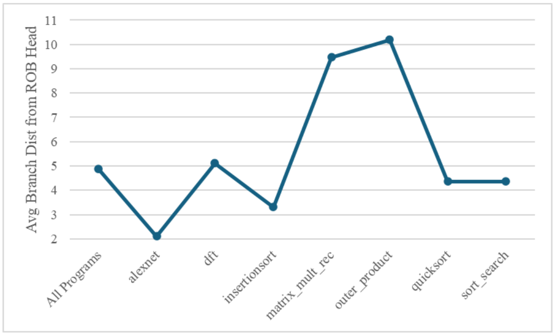

Figure 4: EBR average distance from ROB head

C. GShare Branch Predictor

As mentioned above, our branch predictor is a GShare predictor with 10 BHR bits initialized to weakly not taken (WNT). During testing, we compared the performance of our GShare predictor against the original basic 2-bit bimodal predictor as well as differing BHR bit widths. As seen in the table, the predictor that resulted in the highest prediction accuracy, is not the predictor setup that ultimately gave us the best performance. This could potentially have to do with the fact that we optimized our structure sizes around the 10-bit BHR before we tested any other sizes. There could also be edge cases where the 12-bit mispredicts a higher number of high cost branches while the 10-bit mispredicts more low risk branches and therefore the performance offsets the change in accuracy.

Format	WT Accuracy	WNT Accuracy
6-Bit	84.64%	84.14%
7-Bit	85.22%	84.95%
8-Bit	85.58%	84.94%
9-Bit	86.57%	85.60%
10-Bit	86.79%	85.38%
11-Bit	87.23%	84.77%
12-Bit	87.23%	84.33%
Bimodal	79.88%	79.88%

Table 1: Prediction accuracy of different prediction formats

We found that our first implementation of GShare did not handle branch misprediction recovery, which was a huge hit on our performance. Our final implementation first saves the BHR when a branch enters the predictor, and this is passed through the pipeline with the packet. Then it speculatively updates the BHR based on the prediction issued from the xor and the pattern history table. Along with that we also updated the PHT entry speculatively when the branch came into the unit. On a misprediction, both the BHR and specific PHT entry are updated accordingly. One future add on could be to find a way to track all PHT entries that are changed on branches that are squashed. This is currently an edge case that we do not account for and do not have the capability to roll back those specific PHT predictors.

D. Non-Blocking Banked Icache with Prefetching

Our instruction cache has 2 banks, where sequential memory blocks are placed in opposing banks, enabling wide fetch of up to four instructions per cycle. This banked structure allows simultaneous access to adjacent blocks, maximizing fetch bandwidth.

We have also utilized Miss Status Handling Registers (MSHRs) to track outstanding memory requests, allowing the processor to continue executing while misses are served in the background. Memory requests from the Dcache are given priority over the Icache requests.

Our aggressive prefetching mechanism requests up to 28 instructions ahead of the current PC when memory bandwidth is available. This hides the memory latency by requesting instructions before they're needed. Importantly, when a branch is taken, any outstanding prefetched memory requests are thrown away, preventing Icache corruption. Prefetching aided in providing an Icache hit rate of 80.33%.

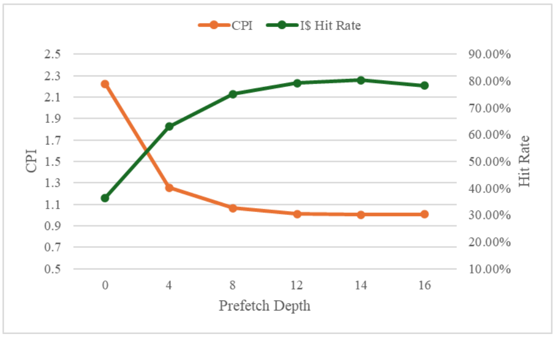

Figure 5: Prefetch effect on ICache hit-rate and CPI.

E. Non-Blocking Banked Data Cache

Our non-blocking direct-mapped Dcache has 32 banks (one per MEM_BLOCK), enabling high throughput memory access. This aggressive cache design can potentially serve cache hits on all requests simultaneously. This cache requires having separate write enable, write data, and read data signals for each bank which may not be affordable in real systems; due to our small cache size however, this was an optimization that did not affect our critical path.

Our Dcache also utilizes a MSHR table of up to 16 outstanding memory requests, allowing execution to continue despite cache misses. When a memory request is a hit in the cache, data is returned immediately. On a miss, the request is recorded in the MSHR and memory access proceeds in the background. This non-blocking behavior reduces load latency and memory-related stalls, enabling the processor to exploit instruction-level parallelism.

Each incoming memory request is checked against both the cache and MSHR table to determine if it is a hit, already pending miss or new miss. New misses generate memory requests when MSHR entries are available.

F. Cache Associativity and Victim Cache

Our Dcache's performance can be largely attributed to evaluating whether a request is a cache hit for all pending loads or stores. Keeping the cache direct-mapped aided in keeping this logic lightweight and satisfying a low clock period. This approach led to a dcache hit rate of 62.46%. After implementing parameterized associativity, our cache hit rate improved with increased associativity, the largest performance jump coming from moving from 8-way to 16-way. However, even a 2-way associative Dcache could not meet slack with our clock period. The reason for this is because the cache hit evaluation logic must be computed for each incoming request, which grows significantly as the number of ways increase. However, the benefits of associativity on CPI and cache hit rate could not be ignored.

In order to address this concern, we decided to include a victim cache (Vcache) for data memory. The Vcache is 4-lines and fully associative. It stores evicted entries from the Dcache and also provide same cycle hits. When a line is evicted from the Dcache, it will be stored in the Vcache and potentially evict something else from the Vcache. Our eviction policy for the Vcache is the bit-based psuedo-LRU algorithm. We chose this for its simplicity and lightweight logic. Similarly to the Dcache, the Vcache will evaluate cache hits for each incoming memory request. If a request is a Vcache hit, it will swap into the Dcache and mark the swapped out line from the Dcache as the most recently accessed line in the Vcache. Having this small cache improved our CPI and served a 2.77% hit rate on top of the original Dcache hit rate. This feature allowed our data memory system to capture some of the benefits of associativity while also meeting our clock period. If we had more time, something we would experiment with further is analyzing the performance of varying number of load functional units and different associativity levels. By reducing the number of load functional units, there are fewer requests that could be serviced simultaneously but the cache hit evaluation could become more affordable for higher associativity. This tradeoff would have been interesting to experiment with.

We also evaluated the effect of associativity in our Icache. Our configuration had 2 banks with each bank as fully associative. However, we saw negligible improvement in performance, which did not justify the increased hardware to support the associativity. We speculate that this is due to two reasons: the Icache access pattern and the eviction policy. Except for workloads with significant branches, the Icache access pattern is largely sequential, which means that cache lines would typically map to different direct mapped lines anyway. This suggested there is limited improvement to associativity to begin with. Our eviction policy for this cache was also the bit-based psuedo-LRU algorithm. We hypothesized that imitating the LRU algorithm may not perform well for code with loops. For example, the structure of a loop has a round-robin instruction access pattern, whereas LRU would most likely evict the instruction that is most likely to be accessed next. If we had more time, we would do more research and experiment with different eviction policies and associativity levels to decide what would work.

G. Byte-Level Store to Load Forwarding

Stores issue memory accesses on retire. This is to preserve precise state and ensure that memory isn't corrupted by mispredicted branches. Our design's store retirement flow is shown in Figure 3. When the ROB's head pointer comes to a store instruction, it asserts a signal to the store queue to convey that. The store queue will check its head to ensure that it has completed and is eligible for retirement. If so, a store request will be sent to the Dcache. The Dcache will respond with a store retired signal to the SQ and the ROB. This signal is asserted when either the store was a cache hit and its update will be visible in the cache on the next cycle or the store was a cache miss, but it was able to get an MSHR entry and send the load request to memory. This store retired signal informs the ROB and SQ that the store was processed and they may move their head pointers forward. For simplicity, our design has an invariant that stores may only retire if they are the first in their cycle to do so.

Along with loads requiring all source operands to be ready, loads must wait until prior stores have resolved their addresses before issuing. To implement this, dispatched loads kept track of a bit vector representing older stores and their complete status. Completed stores will CAM the RS and turn off its corresponding bit in younger loads. Loads may only issue once their bit vector is all 0. However, contrary to a popular approach, we did not have loads resolve their addresses before issuing. A goal we had was to have address generation and cache querying in the same cycle. To simulate the benefit of this, we examined our CPI with and without a 2-stage load and our CPI improved from 1.086 to 1.003. However, since our design may issue loads out of order, there is a potential for a deadlock. This deadlock arises from a possibility of a load clogging a functional unit while waiting for older stores to retire due to an address match, which in turn must wait for an older load to retire. However, this older load cannot retire due to the functional unit being clogged.

To address this deadlock, we implemented byte-level store to load forwarding. As shown in Figure 3, loads

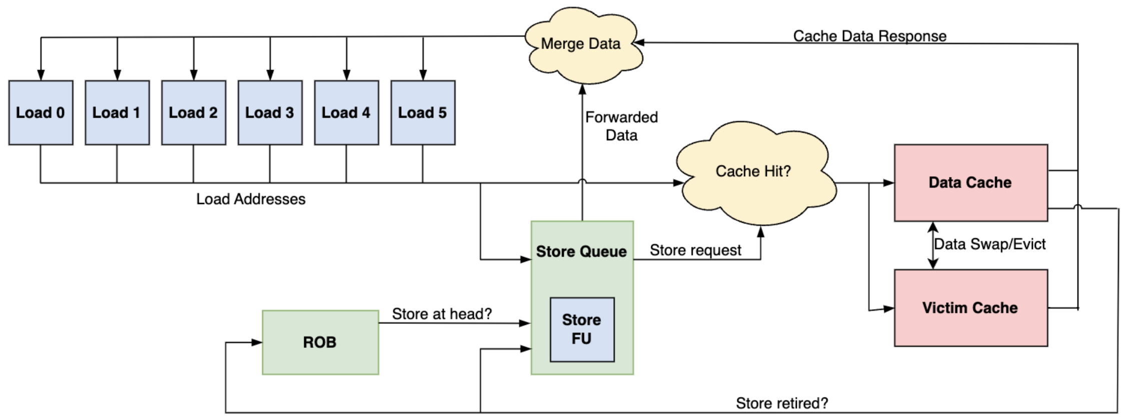

Figure 6: Memory request diagram w/ forwarding

will send their addresses to the Dcache and SQ in parallel. The store queue has 4 priority selectors per load functional unit (one for each byte of the requested data). This is necessary because different bytes of data may come from different stores due to the fine grained memory updates supported by our processor. The data coming back from the caches will merge with the forwarded data before being processed by the load functional units. This ensures that any valid data received by a load instruction is the most up-to-date. If all the required data is available, the load may proceed to the ex_cmp pipeline register. This avoids the aforementioned deadlock as the load in the functional unit could grab the data from the SQ and avoid clogging the functional unit.

V. Testing Methodology and Results

This section outlines the testing strategies explored for our processor, highlighting the evolution of our methods throughout the design and implementation phases.

A. Single Module Testing

In the early stages of development, when only a limited number of modules were implemented, our testing strategy centered around crafting targeted test cases for individual components. For milestone 1, this approach was applied to the reservation station, where we used a combination of directed and randomized test cases to thoroughly evaluate its behavior. As development progresses, we extended this methodology to other core modules, including the branch manager, branch predictor, execution stage, free list, instruction buffer, map table, multiplier, register file, and reorder buffer.

While this module-specific testing proved highly effective at uncovering early-stage bugs, we soon encountered diminishing returns: the growing number of modules made the process increasing time-consuming and tedious. To streamline our workflow, we pivoted towards a more integrated approach-adding modules incrementally to a comprehensive CPU testbench, which allowed us to validate interactions between components more efficiently.

B. CPU Testing

Testing interactions between modules within our CPU proved to be the most effective strategy for identifying bugs during development. This approach enabled us to monitor the state of registers and data-holding modules throughout execution. By exporting this information to a JSON file, we were able to clearly visualize the values and relevant internal states of each module on a cycle-by-cycle basis, significantly improving our ability to debug and validate system behavior.

VI. Project Management

Our team established clear goals early in the project, with weekly milestones that ensure steady progress throughout the semester. We divided tasks based on each member's strengths while working together across shared modules. For complex components, we used pair programming to ensure that team members understood the modules and how it interacts with the system as a whole. Over spring break we were able to assign individual work to each person in the form of new modules. This was vital to our success as it allowed for work to be done without needing to meet up over break. Afterwards we reconvened and filled everyone in on the work done and continued to do group work.

VII. ACKNOWLEDGMENT

We would like to thank Professor Ron Dreslinksi, Professor Kris Flautner and the rest of the EECS 470 staff for their continuous help throughout the semester.