EECS 470 Final Project Report

N.E.V.I.L. & Co. (Group 8)

(Not Very Efficient Verilog ISA L)

Mustafa Miyazawala: mmiyazi@umich.edu, **Wynn Kaza**: wynnkaza@umich.edu,
Caroline Xu: carolinx@umich.edu, **Andrea Liu**: aaliu@umich.edu,
Jason Yu: Jsnyu@umich.edu, **Bozhou Chen**: kotori@umich.edu

Abstract

This is the project report for EECS 470 Computer Architecture at the University of Michigan. We designed an N-way superscalar, R10K-inspired out-of-order processor with early branch resolution, instruction prefetching, and a non-blocking, k-way associative data cache in SystemVerilog. The design choices were made with the objective to learn the effects of different structures and their respective performance impacts.

I. INTRODUCTION

As a project in a Multidisciplinary Design Experience course, this project is designed to allow us to apply what we have learned in class to build a working out-of-order processor. Our team of six people had two months to design and implement our processor design in SystemVerilog. Since all of our group members had limited experience with Verilog, HDL was difficult to grasp at the beginning, and has proven to be difficult to master.

In this report, we will describe the architecture and performance of our custom-designed processor, detailing the rationale behind our design choices and the efficacy of the implemented features. We will outline the processor's structure, from its basic modules to its advanced capabilities, and discuss the integration of key components such as I-cache and D-cache with data memory. Additionally, we will present our testing methodologies and evaluate the processor's performance, using visual graphs and charts to illustrate our findings.

In Section II, we will focus on the design choices and summarize everything included within our processor. We will also include a high level diagram of how we implemented our out-of-order processor. In Sections III and IV, we will provide more details about each basic module included in the processor and focus on the advanced features we included. In Section V, we will provide a detailed explanation of how we integrated our I-cache and D-cache modules into the data memory. In Section VI, we will discuss our testing strategies. Finally, in Section VII, we will evaluate the performance of our processor, presenting results through visual graphs and charts. The report will conclude with Section VIII, summarizing our key findings and observations.

A. Broader Impacts

As AI and machine learning become more prevalent in people's everyday lives, processors need to be able to handle increasingly complex computations such as matrix multiplication and NP-hard optimization problems. Matrix operations are some of the most common operations used in machine learning, and one way to make these computations more efficient is to reduce the computation time. In response, we designed a processor that can optimally handle recursive matrix multiplication to reduce computation time and thus increase overall performance of machine learning algorithms.

II. DESIGN OVERVIEW

In this project, we implemented an R10K inspired N-way superscalar out-of-order processor. There were four base requirements we implemented:

1) A base out-of order processor (MIPS R10K)
2) 3 arithmetic logic units, 2 multiply, 1 load, 1 store, 1 branch functional units
3) Instruction and data cache
4) Branch predictor and branch target buffer

Furthermore, we implemented 2 advanced features:

1) N-way superscalar
2) Early branch resolution (EBR)

Finally, we implemented 8 basic features

1) Per-PC branch history, global pattern history branch predictor (PAg)
2) Instruction prefetching
3) Banked instruction cache (I-cache)
4) Associative data cache (D-cache)
5) Non-blocking D-cache
6) Victim cache
7) Store queue (SQ)
8) Data forwarding from SQ to loads

III. BASIC OUT-OF-ORDER FEATURES

This section will look at all of the base features required to have a working MIPS R10K processor. We will go over each module, what it does, and how it connects to the other modules. Advanced features will be discussed in Section IV and V. Figure 1 shows the high level diagram of our processor.

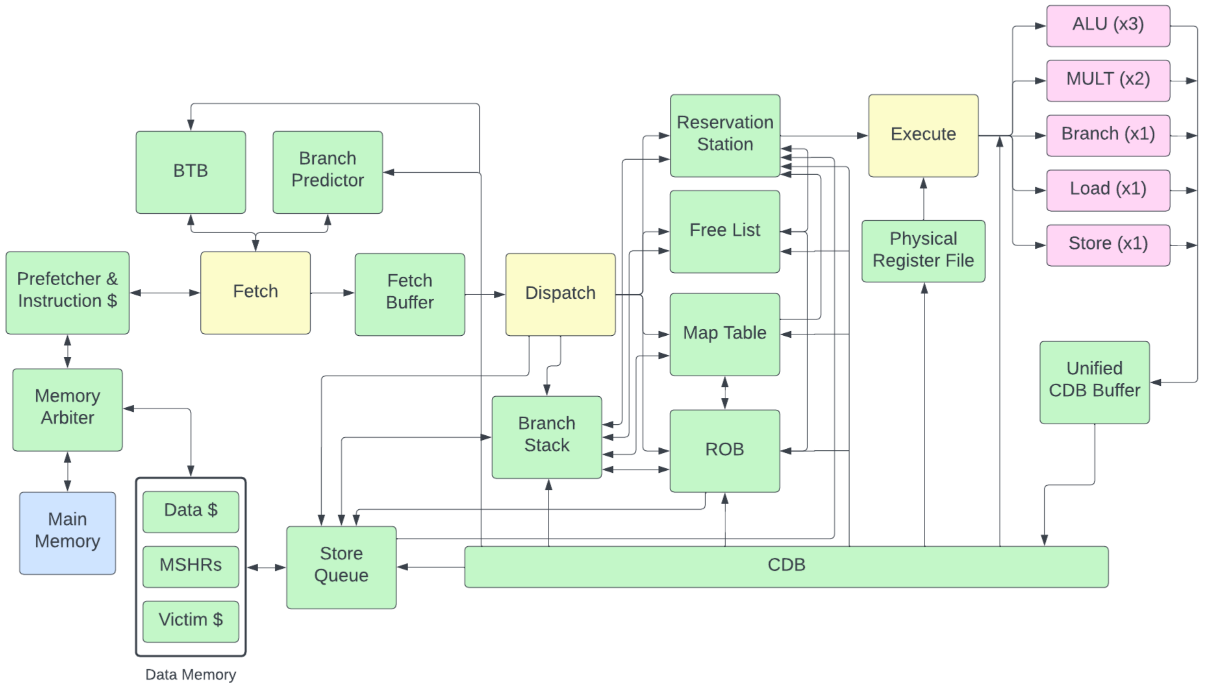

Figure 1 High-level diagram of out-of-order processor

A. Reorder Buffer (ROB)

The ROB is the fundamental building block of the out-of-processor and gives the processor the ability to handle precise state. It keeps track of the order of instructions, even as they are executed out of order by other components in the pipeline. The ROB size directly correlates to the number of instructions that can be active within the processor. Our ROB is sized at 16 entries, which is based on a size large enough to handle a decent case of in-flight instructions, while not increasing the clock period.

We implemented our ROB as a circular first-in-first-out (FIFO) queue, so that the instruction order would wrap around from the last entry to the first. Our ROB includes a head pointer, which keeps track of the oldest active instruction in the processor. The head pointer controls when an instruction is allowed to be retired (written back to memory and committed to the register file), and it moves down upon retirement to free up ROB entries to be reused by another instruction. The ROB also has a tail pointer, which keeps track of the last valid instruction active within the processor. The tail pointer is moved down whenever a new instruction is dispatched, so that it is always pointing to the next available ROB entry. Because our processor is N-way, the tail and head pointer of the ROB can move up to a distance of N on every cycle. Whenever an instruction is put on the common data bus (CDB), it is marked within the ROB as complete. This means that if an instruction is at the head of the ROB and is complete, it can retire on the next cycle.

Because our design implemented EBR, whenever a branch is dispatched, the current ROB tail pointer is saved. Then, if a branch is completed (put on the CDB) and determined to be a misprediction, the branch stack will send the saved ROB tail from the branch's dispatch cycle to the ROB, and the ROB's current tail pointer can simply be updated to saved tail pointer to quickly and efficiently revert to the last correct ROB state. The branch stack will be discussed in more detail in Section IV.

B. Reservation Station (RS)

The RS handles out-of-order issuing of instructions that have been dispatched. It keeps track of instructions that have been dispatched and their dependencies. Our RS is sized at 8 entries, and does not include any ability to handle internal forwarding, which means an instruction cannot be dispatched to the RS and issued on the same cycle. Our RS receives decoded instructions, tags from the free list and map table, and indices from the ROB and SQ. The RS sets completion bits for source registers based on the feedback from the CDB. Furthermore, we optimized the packet being sent out of the RS to only include information that will be required in later stages of the pipeline, stripping the transfer of any unnecessary data.

Due to the implementation of EBR as an advanced feature, we included a branch mask in each entry within the RS. The explanation of how this works is defined within the branch stack section of the report. In terms of memory instructions, the RS will wait to issue loads until all stores are dispatched before the load has been completed. The RS is able to track this using 3 pieces of information from the SQ: the SQ head, the SQ tail, and SQ complete tail. By applying some logic on these three values, the RS is able to handle issuing of memory instructions without worrying about them being handled incorrectly.

A sub-optimal implementation of the RS lies in the priority selector module, which does not have any sense of oldest instruction first. Therefore, there is a possibility that this causes starvation of an instruction (specifically a branch that will be mispredicted later), thus causing a hit on performance.

C. Map Table

The map table facilitates the dynamic mapping of architectural registers to their physical register locations. Our map table size is based on the number of architectural registers which in this case is 32. It stores the physical register mapping of the architectural registers.

When instructions are put into the ROB, the map table outputs the corresponding physical registers names to the architectural registers used. When instructions retire, it receives signals from the CDB to update the completion status of the corresponding physical register. Essentially, the map table helps organize the physical registers and renaming registers to organize register information in the rest of the processor. On a branch misprediction, the map table reverts to a previous state that is stored in the branch stack.

D. Free List

The free list plays an integral role in register renaming by choosing what each register gets renamed to. It is the same size as the number of ROB entries, and its primary function is to output N physical registers that can be utilized by other processor components like the RS, ROB, and map table.

Like the ROB, the free list is also implemented as a circular FIFO queue. The FIFO structure is maintained through head and tail pointers which keep track of the beginning and end of available physical register tags. When N instructions dispatch, the head pointer will increment N entries and output the corresponding physical registers. When N instructions retire in-order, their corresponding physical register tag tail will increment N times, indicating that those registers are free again. On a branch mispredict, the branch stack returns the free list head to the free list which will restore the free list entries that were used after the mispredicted branch.

E. Common Data Bus (CDB)

The CDB broadcasts data from retired instructions to all of the other modules so they can update their status. The CDB can only complete a maximum of N instructions. The CDB also has a CDB Buffer that holds extra packets that can not be broadcasted due to either CDB size or memory limitations. This buffer is the same size as the ROB to remove the necessity to send stall signals to execute stage and RS.

All data from the functional units go into the CDB Buffer before being broadcasted. Any load instruction that is waiting for data from memory/D-Cache will wait in the CDB Buffer. Upon receiving data, a bit within the instruction will go high, allowing for that load instruction to be completed.

Every cycle, up to N entries from the CDB Buffer are chosen to be broadcasted using a priority selector. There is no prioritization of one instruction type over the other, although this could be an optimization to implement in the future.

F. Fetch Stage

Fetch stage is in charge of sending instructions into the pipeline faster than it can complete it. Fetch sends the instruction PC of the first entry in N entries to I-cache. If the instruction buffer is not full, and I-cache sends valid instructions to Fetch, fetch will output the N valid instructions to the instruction buffer and update the PC based on the number of valid instructions. If I-cache does not send valid instructions or the instruction buffer is full, Fetch will stall.

Fetch stage also includes the Branch Predictor (BP) and Branch Target Buffer (BTB) module within it. Our fetch stage did not implement multi-branch predictor and a cascading BTB so we only allow one branch instruction per N valid instruction packets to Dispatch. To do this, there is a branch decoder module that is integrated into the N valid instruction from I-cache that determines if an instruction is a branch. During a Branch instruction, the PC of that branch instruction is sent to the BTB to determine if there is a valid entry, and also sent to BP to determine if it is taken. This information is sent to the instruction buffer alongside the instruction from the I-cache. The BTB and BP are updated through data sent from the CDB into Fetch. More details about Branch Predictor will be discussed in Section IV, and details about I-cache will be discussed in Section V.

G. Dispatch Stage

Dispatch Stage takes the packet sent from Fetch and puts it into the Fetch Buffer, which is a FIFO queue for all of the instructions it receives from Fetch. The Fetch Buffer is the same size as the ROB. If the buffer is full, Fetch Stage stalls until there are empty entries in the Fetch Buffer.

Dispatch also takes information from ROB, RS, and SQ to determine how many instructions it can dispatch in the next cycle. The number of instructions that it can dispatch is equal to the minimum number of empty entries in the RS, ROB, or SQ. These instructions are then sent to the decoder to decode the information needed to execute the instruction. Once decoded, they are sent out as packets to the RS where they stay until they execute.

Because of our EBR using a branch stack, we only allow one branch to be dispatched per cycle, and it has to be the first instruction dispatched. If the branch was not the first instruction, or if there was an occurrence of a second branch, the branch instruction would be delayed until the next cycle to be dispatched. Furthermore, if our branch stack was full, we had to stall branch dispatches until an ongoing branch was resolved.

H. Execute Stage

The Execute Stage is where the instructions do what they are supposed to do. Execute takes in packets sent from the RS and sends the information to the corresponding functional units.

Execute has 8 functional units (FUs): 3 ALU, 2 MULT, 1 LOAD, 1 STORE, and 1 BRANCH. In each cycle, the execute stage accepts up to 8 entries from the RS station. As the CDB Buffer is sized to be the same size as the ROB, there is no stall logic in the execute stage. Our MULT FU is a pipelined 4-stage multiplier, meaning that each multiply instruction would take 4 cycles to complete, but a new multiply instruction can be sent through each cycle. Completed instructions are sent from the FUs to the CDB Buffer.

Since Execute Stage is when the values of the registers are accessed, the register file exists within Execute. Thus, CDB broadcasts information to the register file when an instruction completes. The execute stage and SQ also talk to each other, sending signals to update the SQ complete head pointer and searching the SQ for store-to-load forwarding.

If there is a branch mispredict, all of the FUs have logic to handle the misprediction. The FUs squash any instruction dependent on that branch, including through all cycles of the pipelined multiplier.

I. Branch Prediction

Branch prediction is handled by the branch predictor (BP) and the branch target buffer (BTB). This section will focus on the implementation of BTB and the branch predictor will be discussed in Section IV.

The BTB stores the branch instruction PC that will be used for indexing, and the target PCs for branches that have been previously taken. If a branch instruction is predicted to be taken by the branch predictor and the BTB has a valid entry for the branch's PC, it will output the target PC and fetch stage will update the current PC to the target PC value and begin fetching from there.

The BTB is updated when a branch completes. If the completed branch is taken, the BTB will either update a previous entry with a new target PC, or allocate a new entry that will have the target PC and the branch instruction PC.

The BTB is implemented as an array of 8 entries, which each consist of a 32-bit target PC address, 32-bit branch PC address, and a valid bit. The BTB is directly-mapped by the branch PC.

IV. ADVANCED FEATURES

Table 1 below summarizes all of the advanced features we attempted to integrate. Each of these features will be discussed more in detail below in both Section IV (pipeline) and Section V (memory).

Advanced Features	Integrated in the final submission	Implemented and working but discarded	Partially Implemented and not integrated
N-way superscalar	X		
Early branch resolution	X		
PAg branch predictor	X		
Store queue	X		
Data forwarding inside SQ	X		
Instruction prefetching	X		
Banked I-cache	X		
Associative D-cache	X		
Non-blocking D-cache	X		
Victim cache	X		

Table 1 Table of advanced features

We implemented two advanced features: N-way superscalar and EBR. We chose to make our processor N-way to improve throughput and increase our IPC. We implemented EBR so we could quickly restore a previous state of the processor instead of spending multiple cycles working backwards to revert the pipeline.

A. N-way Superscalar

For N-way Superscalar to work, all of our modules had to be coded N-way. Some modules could be more than N, but the number of instructions dispatched per cycle and number of instructions broadcasted through the CDB were restricted to N-number of instructions to ensure that at most N instructions were dispatched and retired.

To implement N-way without significantly slowing down the clock period, we made sure that our for-loops could unroll so each iteration of the for loop can run in parallel. This meant the code inside the for-loops had to be independent of each iteration.

B. Early Branch Resolution (EBR)

For EBR, we implemented a branch stack to keep track of the state of the processor right before a branch instruction is dispatched. That way, if a branch misprediction occurs, we can immediately restore the state of the processor and fetch the correct instruction.

Each time we dispatch one branch instruction in a set of N dispatch packets, a corresponding branch mask index bit is assigned to the branch and the branch mask is updated for the branch instruction. The corresponding index to the branch mask is also stored in the branch stack as a one-hot encoding. Each instruction follows that branch instruction will contain the branch mask of the dependent branch instruction. If an instruction has multiple branch dependencies, all of the indices of active and incomplete branch instructions will appear in the branch mask. The branch mask is stored with instructions in the RS, execute units, and the CDB. If the number of branches in the RS is equal to the maximum number of entries in the branch stack, the branch stack is full, and the processor stalls until a branch is completed and a spot is opened up in the branch stack.

Our branch stack has 4 entries. Each branch stack entry contains the previous branch mask, current branch mask, and the one-hot encoding of the current branch stack index, which we call the checkpoint mask. The previous branch mask saves the branch mask before the current branch stored in the branch stack entry was dispatched. The current branch mask is the branch mask of the current branch in the branch stack entry. The checkpoint mask indicates the current branch's corresponding bit in the branch mask as a one-hot encoding.

The overall branch stack keeps track of and outputs a current branch mask, next branch mask and a next checkpoint index. If there is no branch dispatched on the current cycle, all instructions are assigned the current branch mask value. If there is a branch, the branch instruction and following instructions are assigned the next branch mask value. The dispatched branch is assigned the next checkpoint index and is allocated an entry in the branch stack. After the branch dispatches, the branch stack will update the current branch mask to the next branch mask, update the next branch mask to include the next available branch stack bit index, and update checkpoint mask to the one-hot encoded next available branch stack bit index.

All instructions are moved through the pipeline with their branch masks. In addition to the branch mask, branch instructions are moved with their corresponding checkpoint index. This allows the CDB to broadcast the checkpoint mask with the completion of the branch so all instructions that are dependent on that branch can either clear the bit that corresponds to the checkpoint mask if the branch is predicted correctly. If the branch is mispredicted, all the instructions that are dependent on that branch instruction are invalidated. This implementation of EBR required us to include logic for updating branch masks and squashing mispredicted instructions in multiple modules, including the RS, and FUs.

The branch stack also holds information from the ROB tail pointer, SQ tail pointer, map table, and free list head pointer. On a branch misprediction, this information is used as follows:

- The ROB tail is restored to the position stored by the branch stack so that the correct next instruction dispatched is immediately after the branch instruction.
- The SQ tail is restored to the position stored by the branch stack so that any stores issued after the branch instruction are cleared within the queue.
- The map table is replaced by the map table in the branch stack so any register writes after the branch are reverted.
- The free list head is restored by the head stored by the branch stack so all of the instructions that have been dispatched, and thus have taken a physical register tag from the free list, have their tags returned to

the free list.

C. Branch Predictors

Instead of a simple bimodal branch predictor, we decided to implement a per-PC branch history register (BHR) and a global pattern history table (PHT), otherwise known as a PAg branch predictor. This means we use the PC to index into the BHR entry, and then use the BHR entry to index into the PHT. Our PAg implementation included a BHR with 8 entries each holding a 4-bit branch history and a PHT with 16 entries each holding a 2-bit saturating branch predictor.

The PAg branch predictor is updated in the fetch stage using the CDB inputs of completed branch PC and whether the branch was taken.

We also implemented G-SHARE where we hash the BHR with the branch PC to index into our global PHT.

D. Store Queue

The SQ allows for stores and loads to be completed out-of-order but allow for in-order memory writebacks. This is achieved by the integration of S, and store-to-load forwarding. The SQ has three pointers that handle the logic with all the other modules that it interacts with. The head pointer points to the next store that will be retired. The complete head pointer points to the current head of complete store instructions. Finally, the tail pointer points to the next location for a dispatched store. The SQ interacts with most of the other modules in our pipeline, including D-Cache, ROB, RS, Dispatch Stage, CDB Buffer, and Execute Stage.

Firstly, whenever a store instruction is dispatched, the SQ tail pointer is incremented. Furthermore, whenever the SQ is full, no other instructions (even non-stores) will be allowed to be dispatched. As the SQ is sized to 16 entries, it will almost never be full, therefore eliminating the chance of this being a bottleneck for CPI in our design.

Only one store can be retired each cycle, and the ROB will send a signal whenever a store retires, allowing for the head pointer of the SQ to move down one. When the SQ head moves down, it will send the data at that entry to the D-Cache, which will handle the rest of the logic for writing it to memory.

During the execution stage, a load instruction will send a request to the SQ to see if there is a store in the SQ that matches the address of the load. If such a store exists, then the data is forwarded from the SQ to the execute stage.

Whenever a store instruction completes, its complete bit in the SQ is set high. The SQ complete tail is set to the currently complete inorder store. While stores can be completed out-of-order, the SQ complete tail can only be completed in-order. This is because the loads in the RS cannot be issued until all stores before the load either exist with data in the SQ, D-Cache, or memory.

V. MEMORY

For this project, the main memory was simulated with a 100 ns access latency with a single read/write port. To mitigate the effect of this access latency on the performance of our processor, we implemented a non-blocking data cache and victim cache for data memory and an instruction cache with prefetching for instruction memory.

A. Memory Arbiter

The memory arbiter plays a critical role in controlling which component has access to memory within a cycle. The modules that require access to memory are instruction cache (with prefetcher built into it) and data cache. Data cache has the highest priority so the arbiter will allow data cache to have access to memory in that cycle, and send a stall signal to I-cache. Only when D-cache does not require access to memory does I-cache get access.

B. Banked Instruction Cache with Prefetcher

The I-cache was implemented as a dual-banked direct-mapped cache with 7 next-line prefetching. Odd addresses go into one bank and even addresses go into the other. Because each instruction is 32 bits and each memory address contains 64 bits, every memory access fetches two instructions, and each cache line stores two instructions.

When the processor is fetching the next instruction, it sends the PC address to I-cache. If I-cache contains valid instructions, it will output the number of valid instructions and the instructions back to fetch. If I-cache does not contain the correct data, it will send the PC address to memory. Once memory returns the data, I-cache will update the entry of the corresponding PC address and output it to fetch in the same cycle. I-cache can only send the PC address to memory if D-Cache is not accessing memory in the same cycle, and the stall signal is sent from the memory arbiter.

Prefetcher relies on the PC sent to I-cache where it calculates the PC for the next 7 lines, and sends each PC address to memory when I-cache has the data fetch it is requesting, and D-Cache does not require memory access. The benefit of next-line prefetching is that instructions are typically sequential, so it reduces the cycles needed to wait for memory to send the next instructions to I-cache by keeping memory active.

C. Data Memory

Figure 2 shows the high-level diagram of the data memory. Each section will be discussed in more detail below.

1. Data Cache

We implemented a write-back, write-on-allocate, k-way set associative data cache. The write-back policy limits the number of expensive main memory operations by only writing to memory when evicting a dirty block (see section on victim cache). By the project specifications, the overall size of our data cache was restricted to 256 bytes. We chunked the cache into 8 byte blocks for a total of 32 blocks. For simplicity, our data cache has one read port and one write port. Logic within data memory arbitrates access to these read/write ports between the MSHRs, victim cache, and pipeline. After sweeping through parameters and seeing the effect of associativity on clock period and CPI, we chose to set the data cache as 4-way set associative.

For the optimal eviction policy given our small cache size, we implemented true LRU (least recently used) replacement. It was a fun exercise to consider all the edge cases that occur with a true LRU and more than one cache access ports that impact LRU rankings. For example, if a read is occurring to the LRU block of a set, and a component is also attempting to write to that same set, then the write should not evict the LRU of the set, but the block that is second to the LRU. This ensures that the LRU rankings are correct for the next cycle.

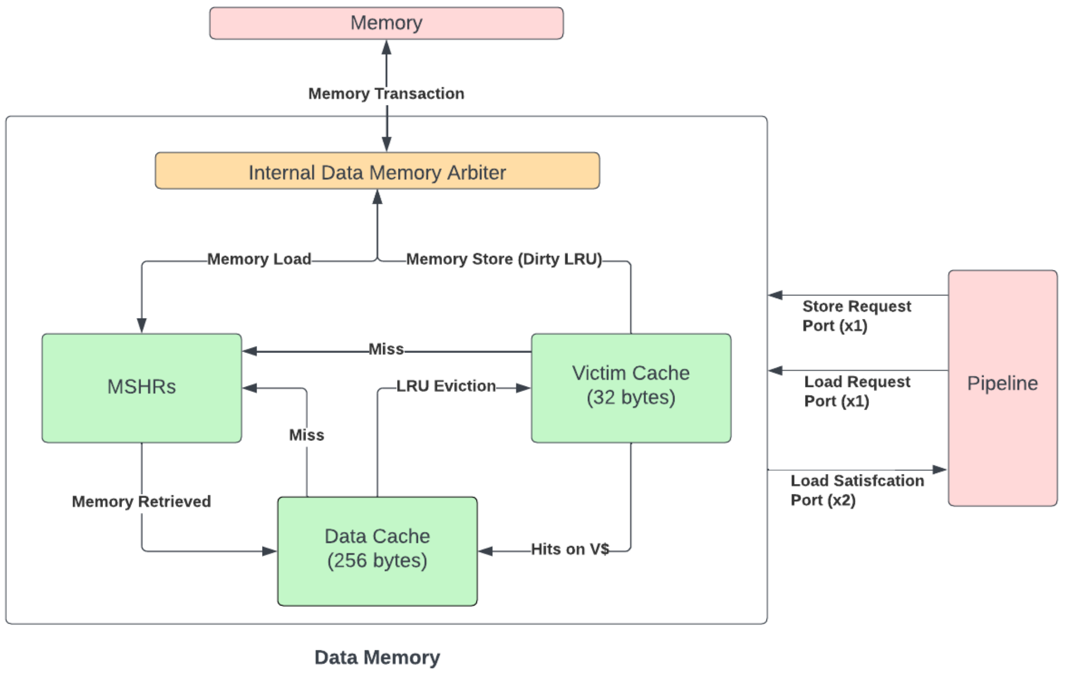

Figure 2 Data memory diagram

2. Miss Status Handling Registers (MSHRs)

The MSHRs allow data memory to accept additional load and store requests even with outstanding misses. The MSHRs is implemented as a FIFO with four pointers: head, complete, issue, and tail. In the event of a miss in the caches (can have up to two in each cycle from the load and store request ports), the address of the miss is searched in the MSHR. If the missed address is found in the MSHR, meaning the missed block has already been requested from memory, the memory requests are merged and no additional MSHR entry is created. If the missed address is not found in the MSHR, a new entry is allocated at the tail.

The issue pointer tracks MSHR entries that have made a request to memory. This pointer is necessary since main memory only has one read/write port and two misses to different addresses can occur in a single cycle. The issue pointer is where an entry receives its memory tag.

The complete pointer tracks the entry that is next-in-line to receive data from memory. Having this pointer avoids the need for a CAM searching for the entry with a memory tag matching the current tag from memory.

The head pointer tracks the entry that is to be populated in the data cache. This pointer is necessary since the data cache has only a single write port. So, if the victim cache or the pipeline is writing to the data cache, the MSHR needs to be able to stall and maintain the MSHR entry until the data cache write port is free. When data is sent from the MSHR to the data cache, the data loaded from memory is combined with the data from write miss(es) to the block saved by the MSHR entry. A bit in the MSHR entry tracks if there is an outstanding load miss for the entry's block. If this bit is set, before the MSHR entry is invalidated, it asserts the combined memory and store data (if any exists) on one of the load satisfaction ports to the pipeline.

3. Victim Cache

Blocks that are evicted from the data cache are not immediately written to memory (if they are dirty) or removed, but are placed in a small 4 block (32 byte) fully-associative cache (the victim cache). This victim cache effectively reduces the total number conflict misses incurred by our set-associative data cache, thus reducing our overall CPI. If a hit occurs in the victim cache, the block that hit is swapped with the appropriate LRU block in it the respective data cache set.

Blocks that are evicted from the victim cache write to main memory if they are dirty, otherwise they are simply replaced. A true LRU replacement policy is used to determine which block will be evicted.

4. Internal Data Memory Arbiter

The data memory module has a single memory output port that interacts with the global memory arbiter and main memory. The internal arbiter always prioritizes memory store requests caused by dirty block evictions from the victim cache over memory load requests from the MSHR. As discussed in the MSHR section, the MSHR has a mechanism to stall in the case that its memory request is preempted by the victim cache. Prioritizing victim cache evictions over MSHR loads is essential to allow the data cache to clear for incoming MSHR data.

VI. TESTING

Testing was integral to making our out-of-order processor work. We wrote testbenches for each of our individual modules, and also to test our entire pipeline. This is what we used to ensure our processor was working properly. We also did a lot of testing to optimize our parameters, with the goal of minimizing our CPI without increasing our clock period by too much.

A. Testbenches

In addition to writing all of the models for the processor, we also needed to write testbenches for each of the modules. The goal of these testbenches were to make sure each module had no bugs prior to integration, and to make sure each module is synthesizable. There were also debug prints that we made for each module that helped us visualize what was going on inside and to pinpoint exactly where bugs occur.

The testbenches usually consisted of some simple simulation inputs to make sure the module was working properly, then some edge cases to catch any discrete bugs. All tests were originally run on simulation because it is quicker. Once all test cases pass on simulation, the module is tested on synthesis, which simulates hardware behavior.

B. C and Assembly Files

Once our pipeline could handle instructions, most of our testing was done by running C and assembly files on our pipeline and comparing the writeback and memory with the in-order pipeline, which we know handles all instructions correctly, albeit slower. Additionally, we wrote test cases to stress test portions of our pipeline, such as hammering the MULT FUs or memory accesses. Debug prints showing the status of each module at every cycle helped us pinpoint where our bugs are.

C. Scripting

Instead of testing each file individually every single time, scripts were written to speed up the testing process. A script was written to compare the writeback and memory of a given program between our processor and the in-order pipeline. We created ground truth folders of each C and assembly file run under different optimizations. Changes could be made to the script to choose which ground truth folder we wanted to compare against, and to run the program on either simulation or synthesis. An additional script was written to run and compare all of the test files using the compare script.

To find the minimum clock period on each of our modules and on our entire pipeline, we wrote a script that did a binary search until it found the lowest whole number clock period that did not violate slack. It would run synthesis on an arbitrary clock period and then look in the synthesis file to see if anything violated slack, then update the clock period accordingly to make synth again. This was especially useful during the beginning of our testing phase because we had no notion of how fast our clock was. However, once we got a better idea of our processor, we were able to manually test clock periods and be more precise.

D. Synthesis

The purpose of running synthesis on our pipeline was to simulate our SystemVerilog code as if it were physical hardware. This helped us catch some edge case bugs that otherwise wouldn't have been discovered if we only tested on simulation.

Our synthesis took a surprisingly short amount of time, with our longest synthesis time being around 1.5 hours, and our final synthesis time with the optimal clock period being 30 minutes. We chose to flatten our design instead of using hierarchical synthesis. Although it is supposed to take longer, we saw an improvement in clock period with the flattened design.

When synthesizing our design, we found that the critical path lied in the unified CDB Buffer in our design. While unable to find how to speed up the CDB Buffer, we recognized many issues within the design. For example, the CDB Buffer must parse two load addresses from memory, and compare it against every address in the CDB Buffer. However, due to current verilog implementation, this is not done in parallel, thus being a huge bottleneck on the clock speed. Another possible issue is the priority selector logic used to decide upon which values to place onto the CDB. Similar to the load address issue, our implementation does not work in parallel, allowing for the possibility for this to be a bottleneck. To reduce this bottleneck, we reduced the ROB size, as we found the clock speed improvement vastly outperformed the decrease in CPI from reducing the ROB size.

E. Parameter Optimization

Once our pipeline was fully implemented, we tested it with varying parameters to find the optimal sizes. This included testing different N-ways, ROB size, RS size, branch stack size, prefetch size, D-Cache associativity, and type of branch predictor. Details about our results will be discussed in Section VII.

VII. EVALUATION

Figure 3 shows how changing prefetch size affects CPI with a ROB size of 32 and a RS size of 16. We can see that around a prefetch size of 7, the CPI performance increase flattens out. Thus we chose a prefetch size of 7 as it seems unnecessary to have it larger.

We tested 2 branch predictors (G-SHARE and PAg) and compared them to always not taken. As seen in Figure 4, G-SHARE and PAg had very similar performance, but PAg was minimally better. Thus we concluded by using PAg.

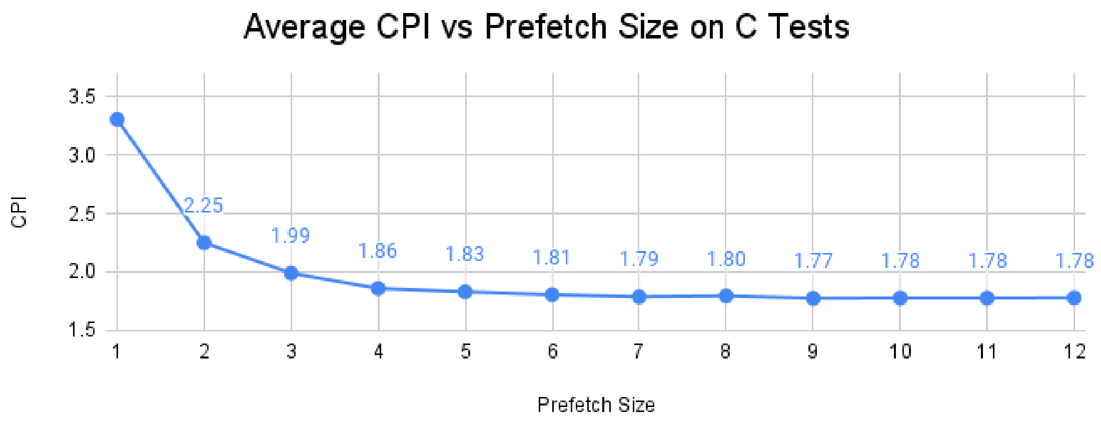

Figure 3 Average CPI vs Prefetch Size on C Tests

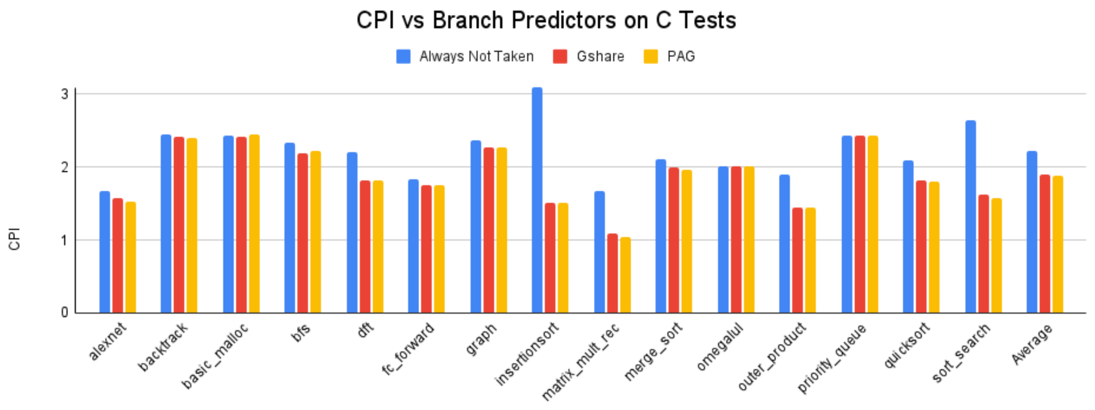

Figure 4 CPI vs Branch Predictors on C Tests

We tested the associativity of D-cache and we concluded with 4-way because it provided similar performance to 2-way, as seen in Figure 5. We did not choose 8 way even though it had a lower CPI because it increased our clock period.

Looking at Figure 6, a ROB size of 32 gives us the best CPI. However, we concluded with a ROB size of 16 because the decrease in clock period trumped the increase in CPI.

Looking at Figure 7, an RS size of 16 and 32 had marginal differences in CPI, and the size of 8 had a slightly worse CPI for all of the programs. However, we ultimately chose and RS size of 8 because it gave us a much faster clock period with only a slightly slower CPI.

We chose a branch stack size of 4 because the performance benefits of increasing branch stack did not justify increasing the size, as seen in Figure 8.

Although 3-way superscalar had a better CPI, shown in Figure 9, we chose a 2-way superscalar because the decrease in clock period trumped the increase in CPI.

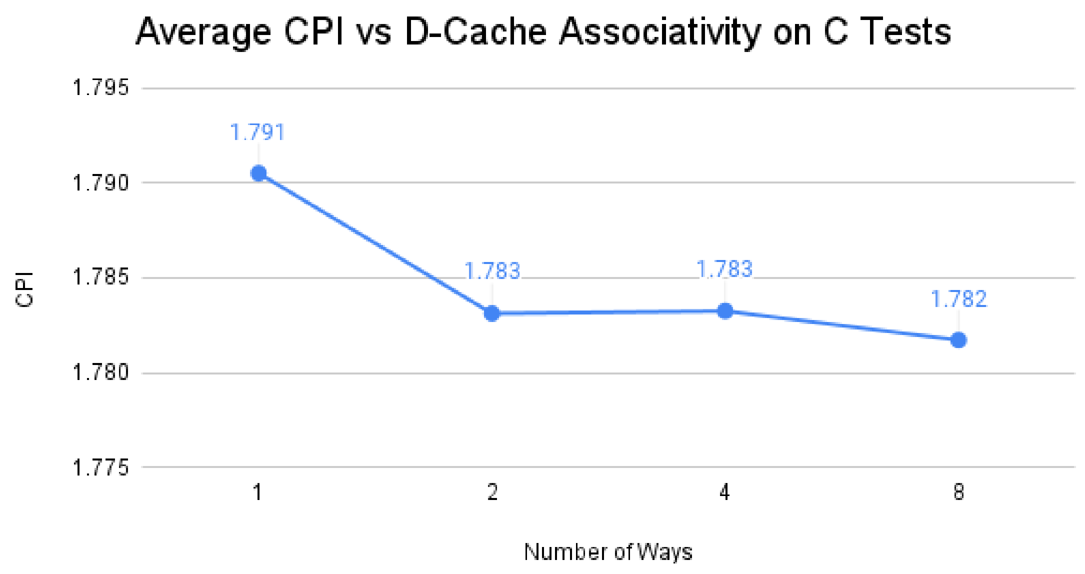

Figure 5 Average CPI vs D-Cache Associativity on C Tests

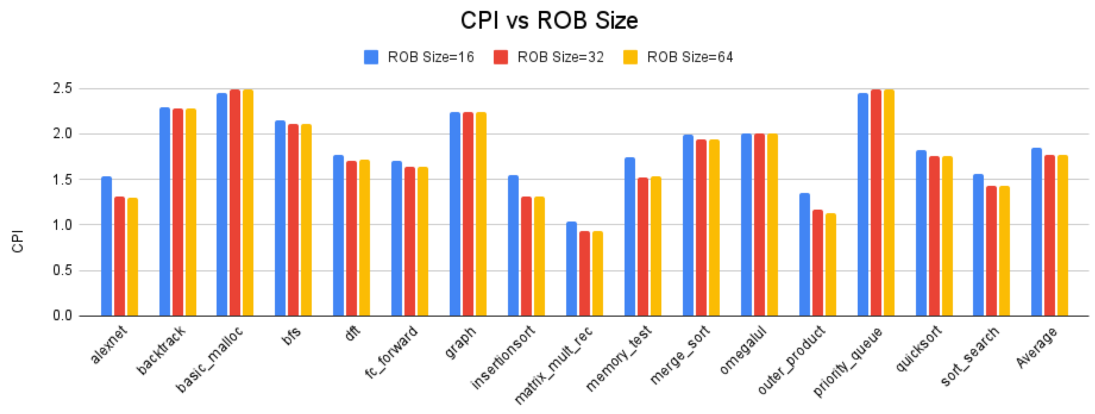

Figure 6 CPI vs Reorder Buffer Sizes

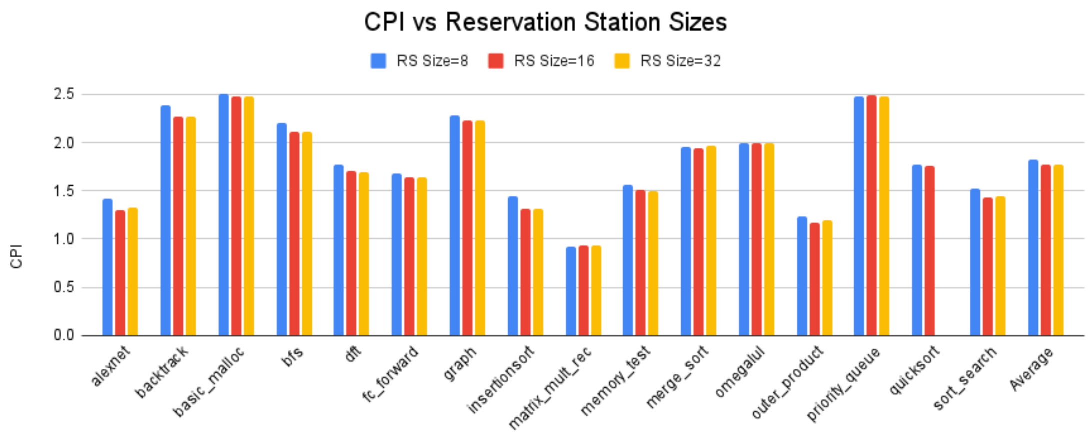

Figure 7 CPI vs Reservation Station Sizes

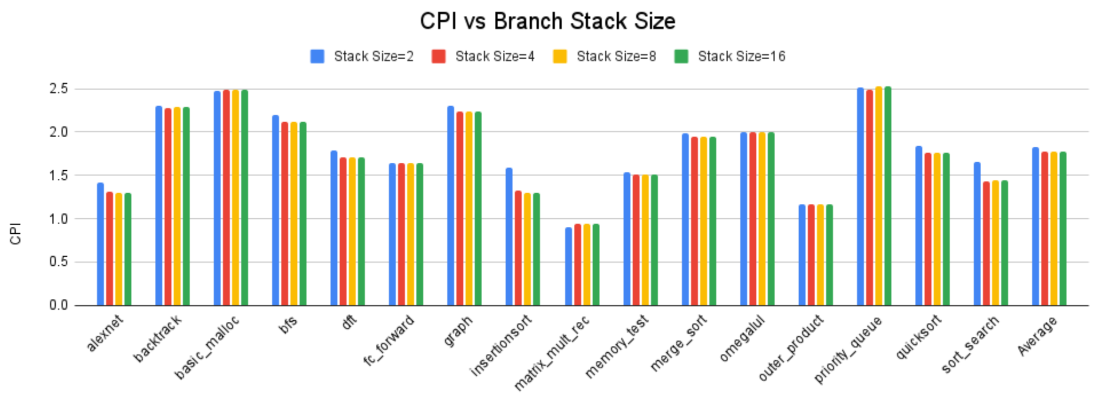

Figure 8 CPI vs Branch Stack Sizes

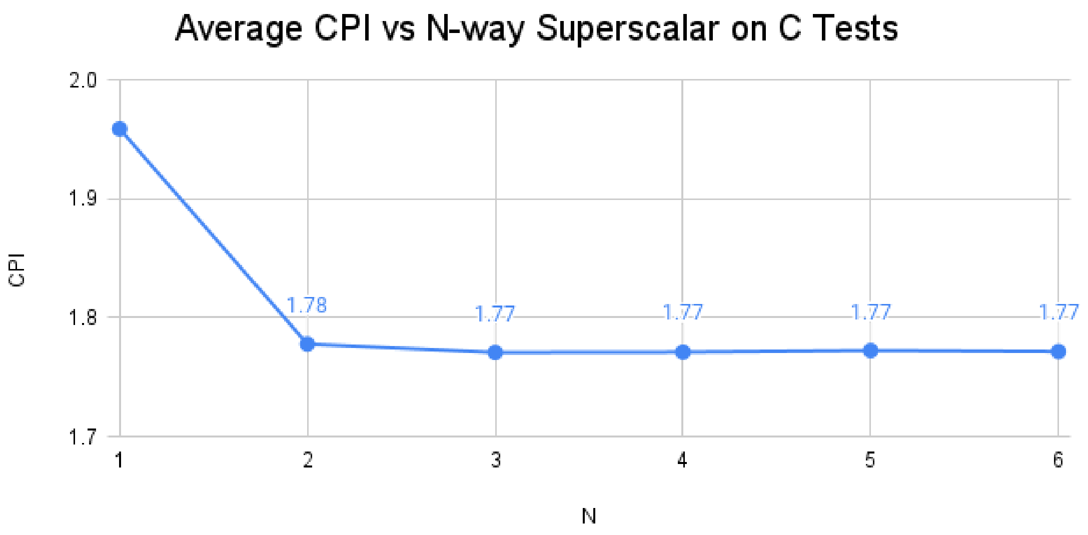

Figure 9 Average CPI vs N-way Superscalar on C Tests

VIII. CONCLUSION

After the completion of this 2 month project, we are proud to have a working out-of-order processor.

Looking back, some of the changes we would have made to improve our project are:

- Immediately separating the tasks of implementing modules instead of working together on one. This would have sped up the process of designing the overall processor as it would mitigate the confusion of different implementation ideas from different people.
- Facilitating better communication on the interfaces between modules
- Enforcing everyone to have a high level understanding of each module

Looking forward, we plan to revisit this project and build on top of the existing features. We are planning to add a return address stack (RAS) and different branch predictors to see if it can improve our CPI further. We were also interested in exploring early tag broadcast because saw many other groups use it with a significant improvement in CPI. Another improvement we would like to focus on is the overall optimization of our existing processor. There were definitely more parameters we could have changed that could improve our CPI but we did not have the time to go through with them.

Our final processor statistics: **Clock Period** - 12.06 ns, **Average CPI for C tests**: 1.83909

We would like to express our gratitude and appreciation for the F23 EECS 470 staff's support for this project.
Thank you.