ECE411: Computer Organization and Design
Final Lab Report

Eviction Notice

Members: Joshua Cudia, Logan Cudia, and Lorenzo Bujalil Silva
TA: Stanley Wu
May 1, 2024

I. Introduction

As the demand for computing power continues to rise, the relevance of computer architecture in today's world becomes ever more apparent. In this report, we will introduce our implementation of a two-way superscalar RISC-V out-of-order processor. Furthermore, this report will explain the basic features of the processor as well as the advanced design features along with its performance metrics. As computer architecture students, this project applied the culmination of the knowledge and skills acquired throughout this class. Through the research of this project, we have learned the different design choices, trade offs, and optimizations that industry-standard processors must consider throughout the design process.

II. Project Overview

This project focuses on the implementation of a two-way superscalar out-of-order processor architecture, aiming to improve instruction-level parallelism and overall computational throughput. Due to the complexity of a two-way superscalar out of order architecture, the primary goal of this project was to successfully design a robust two-way superscalar processor by fully exploring the intricacies of this architecture and gaining insight into the design process and performance optimizations. To achieve this goal, tasks were allocated to different segments of the project to ensure minimal interference. Once each component was ready to be integrated, the team worked together in order to completely understand the design contributed by each member. Moreover, in order to prevent complications and delays in project management, the team consistently reviewed and updated the project roadmaps and functional block designs.

III. Design Description

A. Overview

This processor design follows an explicit register renaming convention capable of accommodating RISC-V's RV32I Base Integer Instruction Set along with "M" Standard Extension for multiplication and division. With our design, we implemented three types of functional units: a memory unit, branch unit, and a general computation unit including an ALU, Wallace Tree Multiplier, and a shift subtract based divider. All branch and memory instructions are all held via respective queues and similarly, M-Extension instructions and remaining instructions are held via reservation stations. For dispatching of instruction from the reservation stations, we implemented a scheduler which would prioritize the youngest instruction that is currently valid, reducing stalls in the pipeline due to the ROB waiting on an instruction. Due to most instructions using the general computation unit, we opted to include two pairs of these units along with their reservation stations as well. On Writeback, this design included 2 CDBs (Central Data Buses). Both CDBs are capable of writing back operations from the general computation unit, however only one CDB writes back branch/jump instructions while the other CDB writes back memory operations. The scheduling within these CDBs both prioritized branch/jump instructions (for CDB1) and memory instructions (for CDB2). Upon fetching, decoding and dispatching instructions to their respective reservations stations, both instructions are enqueued into a single ROB (reorder buffer) index together. Thus,

once ready to commit, both entries must be valid and ready to commit. Upon a flush, a retired free list was implemented. The retired freelist acts similar to the retired rat in the sense that it stores the committed state of the freelist. With the front-end of our design being the most prominent bottleneck, we decided to implement a prefetcher which fetches the next six instructions along with the two target instructions by returning the whole cache line to the processor. To further counteract this bottleneck, we also implemented a tournament predictor including a perceptron branch predictor and a GShare/GSelect predictor. A two-bit counter was used to select which predictor to be used. For our L1 Caches, we implemented a 4-way Set-Associative Cache for both data and instruction memory. An arbiter and cache line adaptor were both needed for communication between the caches and DRAM where the arbiter was used to choose between which of the caches can communicate with DRAM and the adaptor correctly manages the transfer of data between the DRAM burst effect.

B. Milestones

Checkpoint 1

To start with our scalar processor, we implemented a simple FIFO data structure that we would later use to store fetched instructions, branches, memory operations, and much more. The queue generally acts as a buffer between modules and maintains an order within the data.

The bulk of the work during this checkpoint was designing the queue. The approach that we followed to implement the queue was to build a circular buffer with a head and tail pointer. With this design, we can simply have a fixed size array and with the enqueue operation writing to the position of the tail pointer and incrementing the tail pointer, and the dequeue operation will read from the head pointer location and increment the head pointer. The main implication of this design is determining when the queue becomes full or empty that will ensure data consistency in the queue. We verified the queue by reviewing spike ensuring the appropriate PC value connected to the correct instruction out of the queue.

After designing, implementing, and verifying the queue we integrated it with the simple memory model to receive instructions in order then dispatch them to be decoded and renamed.

Checkpoint 2

For the second checkpoint, we continued on our scalar design with the main focus of building out the out of order datapath that supported immediate and register operations. We decided to follow the explicit register renaming datapath where each instruction will first go through a queue then be decoded, renamed, and queued onto the ROB. Once it is renamed, it will be stored into its appropriate reservation station

to resolve any true dependencies. Then after its source registers are valid, the instruction will be dispatched into its appropriate functional unit, ALU or Multiplier, and compute its value. After this point it would go into a CDB module and wait until our writeback scheduler would allow it to write back into the ROB, RAT, Physical Register File, and reservation stations. Finally, the moment that the instructions would reach the bottom of the ROB, they would be dequeued and then committed.

Some of the most challenging parts of this design was ensuring data consistency between modules. At the beginning of the pipeline, when we needed to have a condition on whether or not to dequeue off the instruction queue and since we did not know what instruction it was, we needed to speculatively dequeue it based on any reservation station being open. Then when we would determine the instruction, we would have to potentially stall if its reservation station was not open, which led us to add a buffer between the instruction queue and decode/rename.

After we resolved the main issue of dispatching instructions, we mainly followed the rest of the explicit renaming datapath with simple memory and these parameters:

- Arithmetic Reservation Stations: 4
- Multiplication Reservation Stations: 4
- ROB Entries: 16
- Instruction Queue Entries: 16
- Freelist Entries: 32
- Physical Registers: 64
- CDB: 1
- ALU: 1
- Shift Add Multiplier: 1

Checkpoint 3

The third checkpoint mainly focused on integrating branches + flushing, memory operations, and integration of caches + burst memory. We spent many hours designing the dispatch logic, so that later when we would integrate more instructions, we could simply add another condition to the direction of that instruction.

In order to support branch instructions in our processor we integrated a branch queue that would ensure that branches are executed in order to ensure proper control flow of the program. These branches would be dequeued off and sent to a branch unit where the branching address would be calculated and it would be determined if we should be branching + flushing. Once the branching address was calculated it would be enqueued into a control queue. This queue would maintain all of the branching addresses and would be popped off the moment that the branch instruction would commit and it would be fed into fetch to start fetching from that address.

In order to support memory operations, we needed to implement a load/store queue. This would have the same idea as the branch queue, ensuring the order of every memory operation. Therefore we would never encounter the time that load depending on a store executes after that store. Moreover, stores would not dequeue until they reach the bottom of the ROB where at that point we know that there is no instruction in front of the store and it can write to memory at that time. In order to actually do these memory operations, we created a memory unit that would calculate the memory address to read/write and it would communicate to memory to retrieve its information.

After branches and memory operations would complete their operation in their respective units, they would arrive at the CDB and wait until our scheduler would choose that instruction. Our scheduler would prioritize branch instructions, then memory operations, then multiplication operations, then arithmetic operations.

Then we needed to integrate both caches for memory operations and instruction fetch. We also needed to use the new memory model that was a singular port. Since we could only perform one memory operation at a time, we needed to integrate a cache arbiter to choose between the cache that can access memory at that time and keep the other one waiting. This resolved any issues where both caches needed to access memory at the same time. However the memory model that we were provided is a burst memory that can respond after any period of time. This means that 64 bit responses would be provided from memory until 256 bits were provided for read operations. Then for write operations, the processor needed to wait 4 cycles to provide all 256 bits of information. We resolved this issue by using a state machine that would move through different read/write states until the operation would be complete. We had 9 states in total: 4 read states, 4 write states, and 1 idle state.

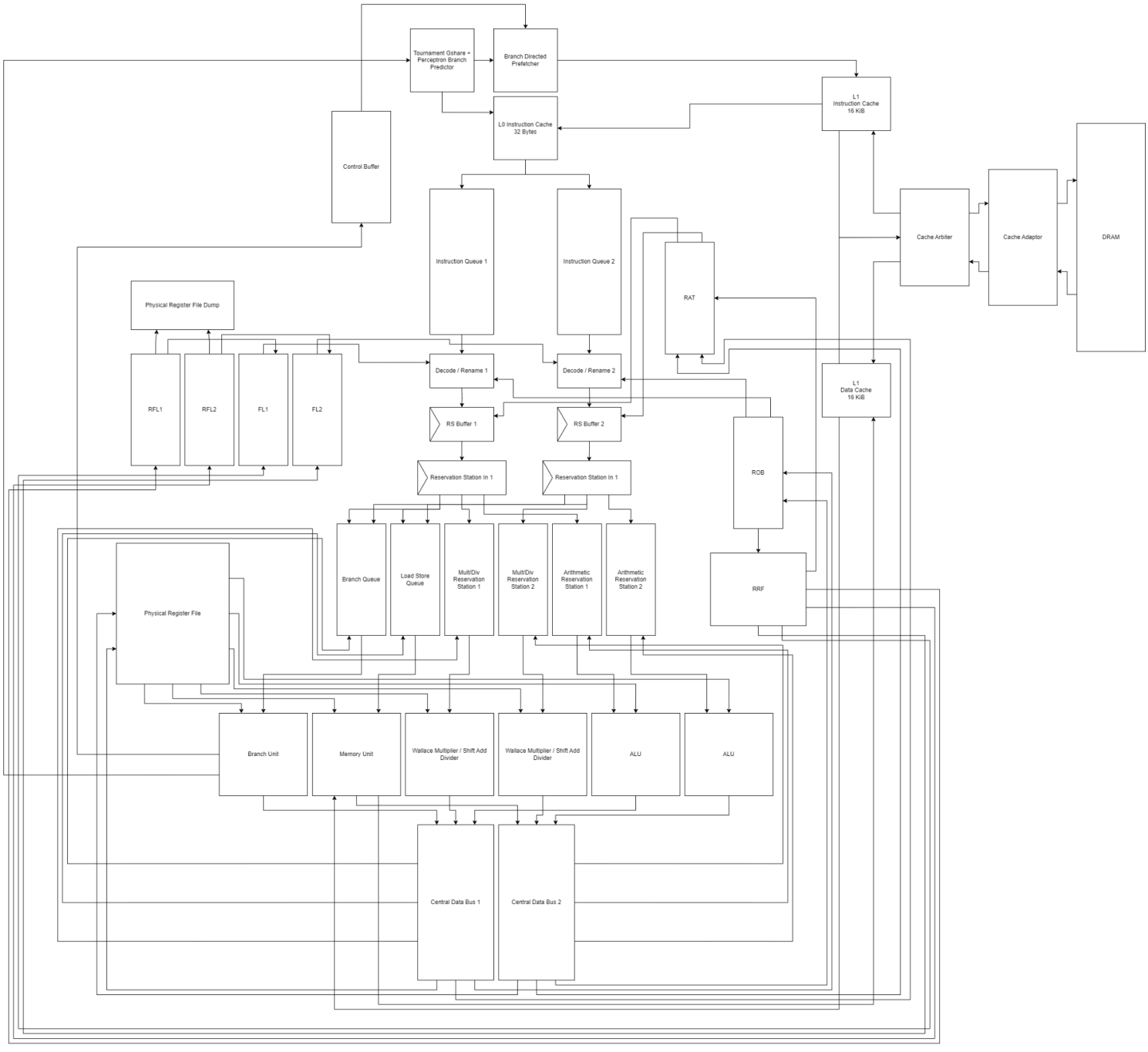
The final part of this checkpoint was supporting flushing logic. In order to flush instructions, we would clear everything the moment that a branch is dequeued off the ROB with the flush signal on. Flushing everything at this point would ensure that no other instruction processes information and changes the committed state of the processor.

Here are the parameters of the processor at this checkpoint:

- Arithmetic Reservation Stations: 4
- Branch Queue Entries: 16
- Load/Store Queue Entries: 16
- Control Buffer Entries: 16
- Multiplication Reservation Stations: 4
- ROB Entries: 16
- Instruction Queue Entries: 16
- Freelist Entries: 32

- Physical Registers: 64
- CDB: 1
- ALU: 1
- Shift Add Multiplier: 1

C. Final Design Diagram



D. Advanced Design Options

1. Superscalar Two-Way

a) Design: The design of the superscalar was essentially duplicated with respect to the front end of the data path. Instead of one instruction queue, decode stage, and queue/reservation station flip-flops, all of the above were duplicated in order to process and hold two instructions at the same time. We believed this was the simplest and most efficient way to process these instructions, maximizing the ILP and throughput. However, in order to properly retrieve these instructions, the fetch stage was completely redesigned to properly fetch two instructions by incrementing PC by 8 (instead of 4) and changing necessary widths of port connections between the L1 instruction cache and the processor, specifically the instruction address, instruction data, and the instruction memory read mask. In the backend, due to two instructions being in flight, scheduling and edge-case detection was needed in almost every aspect of the pipeline. For example, in the RAT (register alias table), if two instructions enter with conflicting registers (matching architectural destination registers), proper scheduling based on age is needed to proceed with the flow of the program. Furthermore, in the ROB, if a store and branch were both enqueued into the same index, where the store is after the branch in the program, stalling needed to occur in LSQ in order to prevent the store from writing into memory before knowing if the branch was taken or not. Also, in the case of two stores being enqueued into the ROB, the instructions age was the main distinguisher on writeback instead of ROB index due to writeback only validating the first entry in the index due to matching physical destination registers. Upon commit, extra logic was implemented due to the case of a taken branch committing with a younger instruction.

b) Testing: In order to test a feature as prominent as this, we first tested the front end of the processor. We first verified that there were in fact two instructions being retrieved, placed into their respective instruction queues, decoded in the correct decode stage, and then placed into the correct queue/reservation station flip-flop. Once we confirmed this behavior, we tested all instructions (not including loads, stores, or branches/jumps). Once we were able to properly write back and commit these instructions, we moved onto branches and memory operations, ensuring proper behavior during flushes.

c) Performance Analysis:

2. M-Extension (Wallace + Shift Subtract Divider)

a) Design: The design of the Wallace Tree Multiplier focuses on visualizing the process of multiplication into the summation of partial products. Knowing this information,We designed the process of multiplication into an initial stage of a 32x64 bit array being compressed into an array of 32 6 bit vectors. The rationale for a 64x32 array comes from the fact that we are multiplying two 32-bit which always produces a 64 bit value which means that we have 64 columns. Additionally, we have 32 rows because we will have at most 32 partial products that we need to add. Finally, after adding each and storing into a sum, we will need 6 bits maximum to hold that sum. With the compressed array of 32 6-bit vectors,Werepeatedly added each overflow bit of each column sum into its next column. For example, if column 3 had a sum of binary 11 or decimal 3, then the most significant bit would be the carry-in bit into the next column, which is column 4. With this logic,We Added the carry-in bits until each entry was less than 1 or only 1 bit in size. Some tradeoffs with this design is the amount of area needed to recreate each stage and store intermediate results. Additionally, since my design was made behaviorally rather than structurally,We were able to debug much easier and organize stages much more clearly.

The design of the Shift Subtract Divider (SSD) was very similar to the design of the Shift Add Multiplier. However, there were several things to consider. The first is that our quotient would be 32 bits and that we only have signed-signed or unsigned-unsigned division. Next, it was important to consider that many times we do not need to go through the entire 32 cycle division. Edge cases such as dividing or being divided by 0 or 1 make it easy to avoid the entire division process. With that being said, the design of the SSD consisted of a 3-state FSM: Idle, Shift/Subtract, Done. A slight change we decided to make was to combine the shift and subtract state into one state. This is since we almost always enter a shift state, then from here, we check whether we need to subtract which is when our accumulator is greater than or equal to our divisor. The shifting/subtracting states end when our internal counter reaches 32, since we only need to shift 32 times which is the size of our dividend.

b) Testing: To test,We began with basic multiplication of 4x4 bit numbers to see if my stages are operating correctly. After verifying that,We Began a series of targeted testing to verify all cases of multiplication are working. With my new targeted testsuite,Welater tested every type of multiply then continued to test on the testsuite given to us. After main functionality,We began fine tuning edge cases with zeros, negative numbers, etc. Another aspect of testing was getting this process to 3 cycles and minimizing the

delay needed to perform.Wewas able to minimize the delay by adding registers in between specific stages.

To test the SSD,We began testing it directly with the physics_div and rsa_div to get an overall sense of functionality. These two tests help me understand how to plan for negative overflow, divided by 0 and 0 divided by negatives.

c) Performance Analysis: There was an immediate improvement in IPC by ~0.7 for programs that used a multiply, this is because of the drastic cycle decrease from a 32 cycle multiply to a 3 cycle multiply. However, we did see a slight decrease in IPC from programs that do not use a multiply. Programs suffered around a 0.001 decrease in IPC. We suspect that this is due to the large area devoted to this multiplier. This multiplier added around 25k units of area.

The overall performance of the SSD allowed us to perform division operations in 32 cycles max, similar to the shift add multiplier. Comparing the physics and rsa test cases that had and did not have a divider, we saw that the test cases without a divider needed had higher IPC, however, it was also longer commit time and simulation time compared to that of the test cases with a divider. A divider will be useful in workloads that need remainders or division of numbers. However, this implementation is very costly in terms of CPI since almost every DIV/REM instruction requires 32 cycles.

3. Perceptron + Gshare Tournament Branch Predictor

a) Design:

Perceptron - The perceptron branch predictor is made from the simplest version of a neural network. It is a very simple linear classifier capable of dividing a hyperplane of space to accurately separate all different entries in this space. The entries in the space that we are referring to are the branches themselves. A linear classifier is a great solution to use when predicting branches since only a binary output is required, take or don't take.

In order to understand the perceptron, we followed a paper by Daniel A. Jimenez and Calvin Lin, *Dynamic Branch Prediction with Perceptrons.* This paper offered a clear explanation of the concept of perceptrons and how they can be used to predict on branches.
In essence, the perceptron branch predictor is made up of a series of many perceptrons each storing an array of weights. When a prediction needs to be made, fetch will supply an address and using a hash function, we can index the perceptron table to choose the perceptron that will provide the prediction. Then we will calculate the dot product between the branch history register and the weights. This value will be the output

and if the output is less than 0 it will be predicted don't take the branch and on the other hand it will predict take when the output value is greater than 0.

At the start of the program, the weights for the perceptron will all be zero. The moment that a branch is fetched, a prediction will be made on and if we should take, we will start fetching from the calculated address. Later in the pipeline, the branch will be resolved and it will be determined if we mispredicted the branch. Once the branch is resolved, it will be sent back into the branch predictor and it will compare the prediction to the outcome. If the branch was mispredicted or the training threshold has not been met, training will be done on the perceptron by updating the weights. These weights will be updated based on the prediction and the global history shift register. If the prediction matched the history then it would increment the weight of that weight, and decrement it if predicted wrong. Effectively, we can train these weights to trend to a certain weight. A positive correlation means that that weight will supply more likelihood for the prediction to be taken, and a negative correlation will generally supply a don't take prediction. Then some weights will have weak correlation which will generally not supply much information to the outcome of the prediction.

We decided to build our perceptron with the hardware budget of 32 KB. The paper mentions that the best parameters to have for the perceptron for 32KB of data available is to maintain 512 perceptrons, 59 bits for history length, 127 for a threshold weight, and 8 bits to store weights.

Gshare - The Gshare branch predictor is a much simpler version of the perceptron, but combines well in a tournament method. Effectively, the Gshare will follow the same concept of storing a table of predictors and using the branch address as an index into this table. As branches are fetched, their addresses are XORed with a global history register to act as a hash function to determine the index into the table. Once the index is created, we can use the stored prediction at this point in the table as the prediction for that branch. The predictors at each point of this table are made up of 2 bit saturating counters. Similarly, once the branch is resolved, the branch address, prediction, and outcome are all fed back into the branch predictor to perform training. This address will index the table and update the predictor at this point in the table to its new state.

The main concept behind these two predictors is using a good hash to be able to distribute the branches across the entries in the table so each has a good prediction already stored.

Tournament - Both of these predictors are good on their own, but at some times they will make a misprediction and if they make many bad predictions it would be detrimental to the performance of the CPU. In order to avoid this happening, we can set up these branch predictors in a tournament style where the prediction that is chosen is the one that a 2 bit saturating counter is pointing to. Essentially, as long as a branch predictor makes good predictions it will be used as the main predictor, but the moment it stops making good predictions the saturating counter will point towards the other predictor to see if that one can make a good prediction. Effectively we are considering the performance of the predictors along with improving their independent performance.

b) Testing: We first started with building both predictors and connecting it to the CPU without fetching a different address when predicting taken. This allowed us to receive information about the prediction logic that wouldn't affect the performance of the rest of the CPU. We were able to detect many of the bugs in both predictors using this strategy. After a series of many tests, we were able to achieve an accuracy we were comfortable with. There were many bugs that we encountered to ensure the appropriate signed calculation of the perceptron output, and ensuring that the data in the weights was appropriately maintained. Finally, we later integrated the fetching logic to both of these predictors and determined a noticeable difference in the IPC of the program as we did not have as many flushes. We also added some coverage testing to ensure that we were hitting all states of the tournament predictor.

c) Performance Analysis: The main performance benefit that we encountered from integrating both of the predictors was the increase in IPC since we were predicting at 90% accuracy for the perceptron and 85% accuracy for the Gshare predictor on the coremark_im benchmark. We also saw a speed up of 1.18x on the IPC of the coremark_im benchmark, going from 0.33 to 0.39.

Code	Perceptron Branch Prediction Accuracy	Gshare Branch Prediction Accuracy	Tournament Branch Prediction Accuracy	IPC (no BP)	IPC (tournament BP)
coremark_im	90.01%	85.3%	88.4%	0.33	0.39

4. **Post-Commit Store Buffer**
    a) Design: The design of the Post-Commit Store Buffer was to store the data of a store instruction upon commit into a queue within the memory unit that interfaces with the load/store queue. Stores were still computed once ready to commit in the ROB, however loads were first interfaced with this store buffer, checking for a matching data memory address. If there was a match, the load instruction would retrieve the necessary data from the buffer and then proceed to the CDB. If there are no matches, loads will enter the memory unit and proceed to the CDB.
    b) Testing: To verify the functionality of the post-commit store buffer, we proceeded with the given testing coremarks and competition suites. We ensured that proper forwarding of the load to the CDB (instead of the memory unit) upon a match was occurring and that post commit store data properly enqueued and dequeued from the store buffer. Furthermore due to the previous memory unit design, we made sure to verify that the forwarded load instruction waited on a memory response, rather defaulted the memory address and used the forwarded data instead.
    c) Performance Analysis:

Code	IPC(no store buffer)	IPC(with store buffer)
mergesort.elf	0.2956	0.3033
fft.elf	0.3187	0.3213

5. **Parameterized Cache**
    a) Design: To implement this, we just added parameters for the number of sets we can have. I added these parameters by changing the tag size, set bits, and SRAM size (using the config file). This was a pretty straightforward design and implementation.
    b) Testing: To test, we pretty much checked to see if we can maintain functionality for the test suite.
    c) Performance Analysis: There was no performance impact from adding parameters.
6. **Next-Line Prefetcher**
    a) Design: This feature was implemented through changing the design of the instruction fetching of the processor along with the UFP (upward facing ports) interfacing with the L1 instruction cache. By incrementing the PC by 32 (instead of 8), we were able to effectively return the entire cache line upon request. The remaining 6 instructions were then stored in a queue acting as an L0 cache. This queue enqueued two instructions into a single index and then dequeued both instructions at once into their respective instruction queues (IQ), where the first instruction fed into

IQ1 and the second instruction fed into IQ2. Upon branches and flushes, the L0 cache/queue is flushed and the branch address is then correctly aligned to the nearest cache line before sending to the cache. When receiving the data from the branch PC, we invalidated the unnecessary instructions and proceeded with the original prefetching process.

b) Testing: To test this feature, we proceeded with the same procedure as testing our whole CPU. We used both coremarks as our benchmarks and based the performance improvement off of IPC. However, we made sure to verify two main points. Firstly, upon a branch, we confirmed that the processor's PC was aligned to the nearest cache line and upon receiving this data, that the correct instructions were invalidated. Secondly, we confirmed that the invalidated instructions were never enqueued to the instruction queues from the L0 cache/queue.

c) Performance Analysis: As the table shows, there was a speed up of about 1.2 when introducing the prefetcher. Although a small improvement, its performance benefit still contributed to the final design

Code	IPC (Prefetcher)	IPC (No Prefetcher)	Speedup
coremark	.38	.33	1.15
coremark_im	.43	.36	1.19

IV. Conclusion

In conclusion, our design consists of a Two-Way Superscalar Processor with an in-order front end paired with an out-of-order backend using Explicit Register Renaming. Our greatest feature was the addition of 2-Way Superscalar paired with many other features such as a tournament branch predictor consisting of GShare + Perceptron, an M-Extension consisting of a Wallace Tree Multiplier and Subtract-Shift Divider, post commit store buffer, parameterized cache sets, and a next-line prefetcher that we used to optimize of fetching logic. Our design also consisted of some unique design in terms of our multi banked freelist, which consists of an addition of a retired free list to keep track of the state of the free list prior to commit, similar to the retired RAT, and a CDB scheduler prioritizing age of instructions with an explicit priority for branches and stores . We saw our greatest increase in IPC with the addition of our branch predictor and the implementation of 2-Way Superscalar. Finally, we would like to thank Jason Guo for the continued help and mentorship especially with the design of our front-end, store buffer, and M-Extension. This project has been a great learning experience for both RTL design and design verification. In the future, we would like to look at more verification techniques we could implement using Cocotb and also design our CPU to be capable of hosting a 32-bit Linux Operating System.