ECE 411 Project Report

Team: Anonymous Calc

Yousef Khadadeh yfk2@illinois.edu

Qingzhe Liu qingzhe3@illinois.edu

Jacob Zheng jzhen24@illinois.edu

Date: 2024/12/11

Instructor: Dr. Nam Sung Kim

Table of Content

Introduction

This project was developed as part of the ECE 411 course at the University of Illinois, Urbana-Champaign, taught by Dr. Nam Sung Kim during Fall 2024. Our team, Anonymous Calc, designed and implemented an RV32IM CPU in SystemVerilog. The primary objective of this project was to explore advanced computer architecture while gaining practical experience in register-transfer-level (RTL) design and verification.

Project Overview

RISC-V (Reduced Instruction Set Computer - Five) is a modern, open-source instruction set architecture (ISA). Our CPU implements the RV32IM subset of the RISC-V ISA, featuring 32-bit registers and supporting integer operations as well as multiplication and division instructions.

The CPU employs an Explicit Register Renaming (ERR) architecture to enable Out-of-Order (OOO) execution and supports the RV32IM ISA. Leveraging the ERR structure, advanced features such as superscalar execution, prefetching, a post-commit store buffer and a 4-way set-associative pipelined cache were implemented.

Throughout the project, we used Discord for team communication, Git for collaboration and version control, and Google Docs for documentation and record-keeping.

Design Description

Overview

The architecture of our CPU is shown in Figure 1 below.

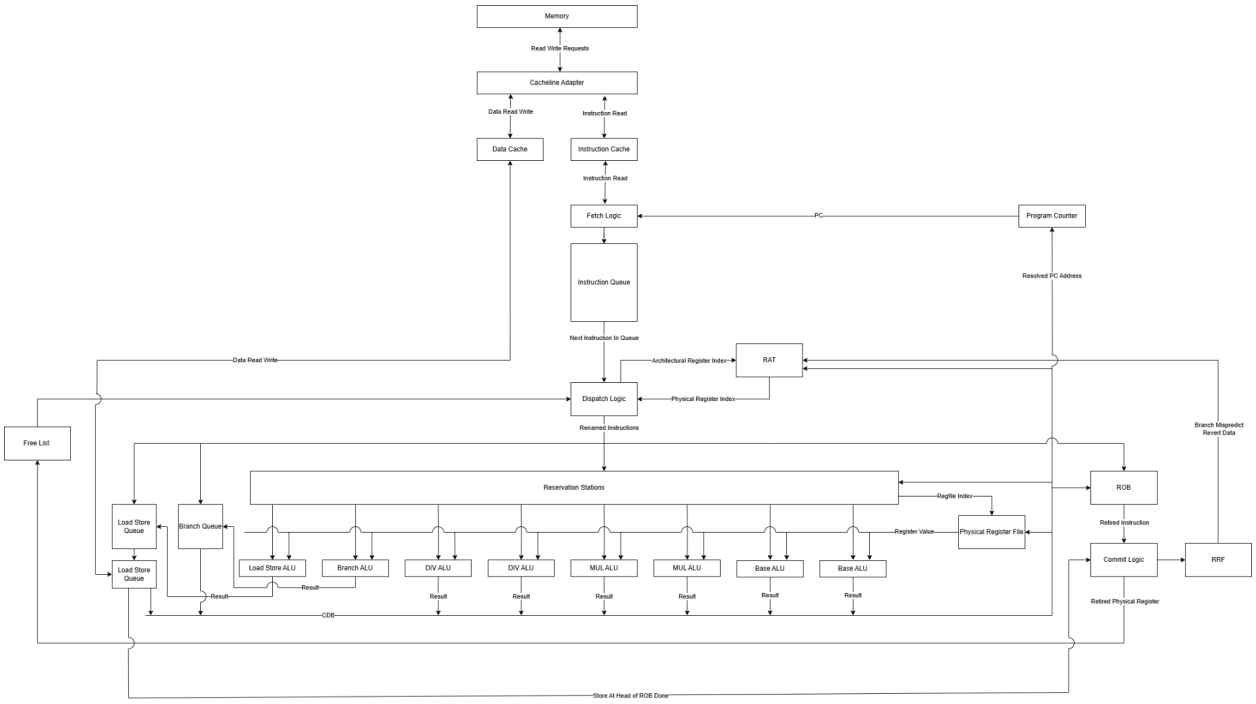

Figure 1: Overview of CPU Architecture

Milestones

Checkpoint 1

At this stage, we have a block diagram to describe the overall architecture of our CPU, shown in Figure 2. We also implemented a queue structure and a cacheline adapter for memory and cache communication. Then, we integrated the modules together and implemented the fetch stage of the OOO execution. We first tested the queue individually with a randomized unit test, then tested the functionality of the fetch stage by demonstrating that instructions could be read.

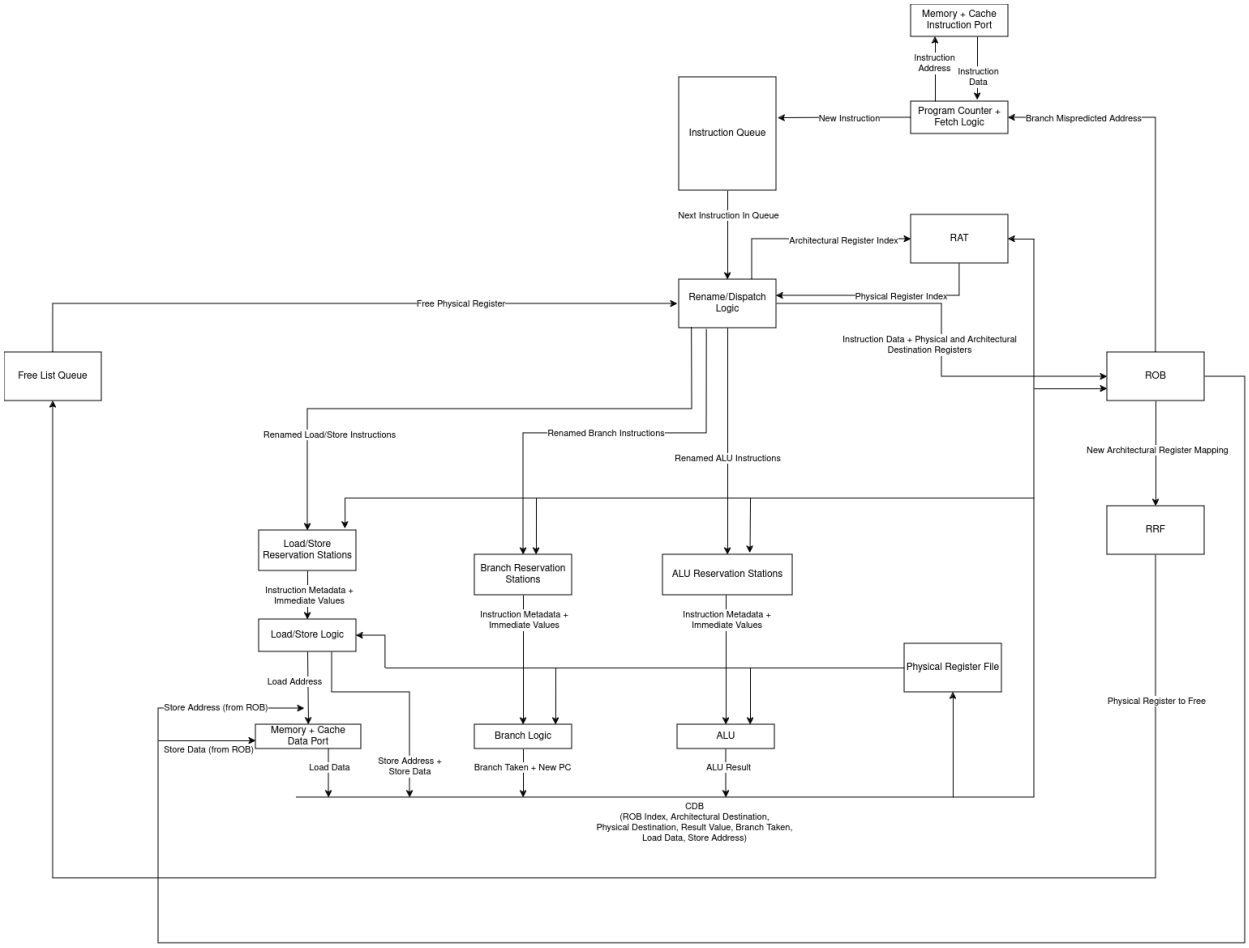

Figure 2: Overall CPU Block Diagram for Checkpoint 1.

Checkpoint 2

At this stage, the CPU supports all functions except load/store, branch/jump, and AUIPC instructions. Our team divided the implementation into three independent sections:

- Dispatch logic: register aliasing table (RAT)
- Commit logic: reorder buffer (ROB), free queue, retired register file(RRF)
- Computation: reservation station, physical register file, arithmetic logic unit (ALU)

These components were then integrated into the fetch logic developed earlier.

Currently, the CPU includes one base ALU (adder) and one dedicated MUL/DIV ALU to support Out-of-Order (OOO) execution. The MUL / DIV ALU contains 4 Synopsys IPs for signed and

unsigned multiplication and division for support of the M extension. The base ALU handles all other arithmetic instructions and was copied from mp_verif. For simplicity, we initially implemented separate reservation stations for each ALU. Later, these were replaced with a unified reservation station in a subsequent checkpoint since the enqueuing and broadcasting logic would be simpler for superscalar. The CPU architecture at this stage is illustrated in Figure 3.

To validate the CPU's functionality, we manually wrote assembly code excluding the unimplemented instructions. We also slightly modified the randomized testbench from mp_verif to generate multiplication and division instructions to test correctness. This approach ensured correctness of execution and proper OOO behavior.

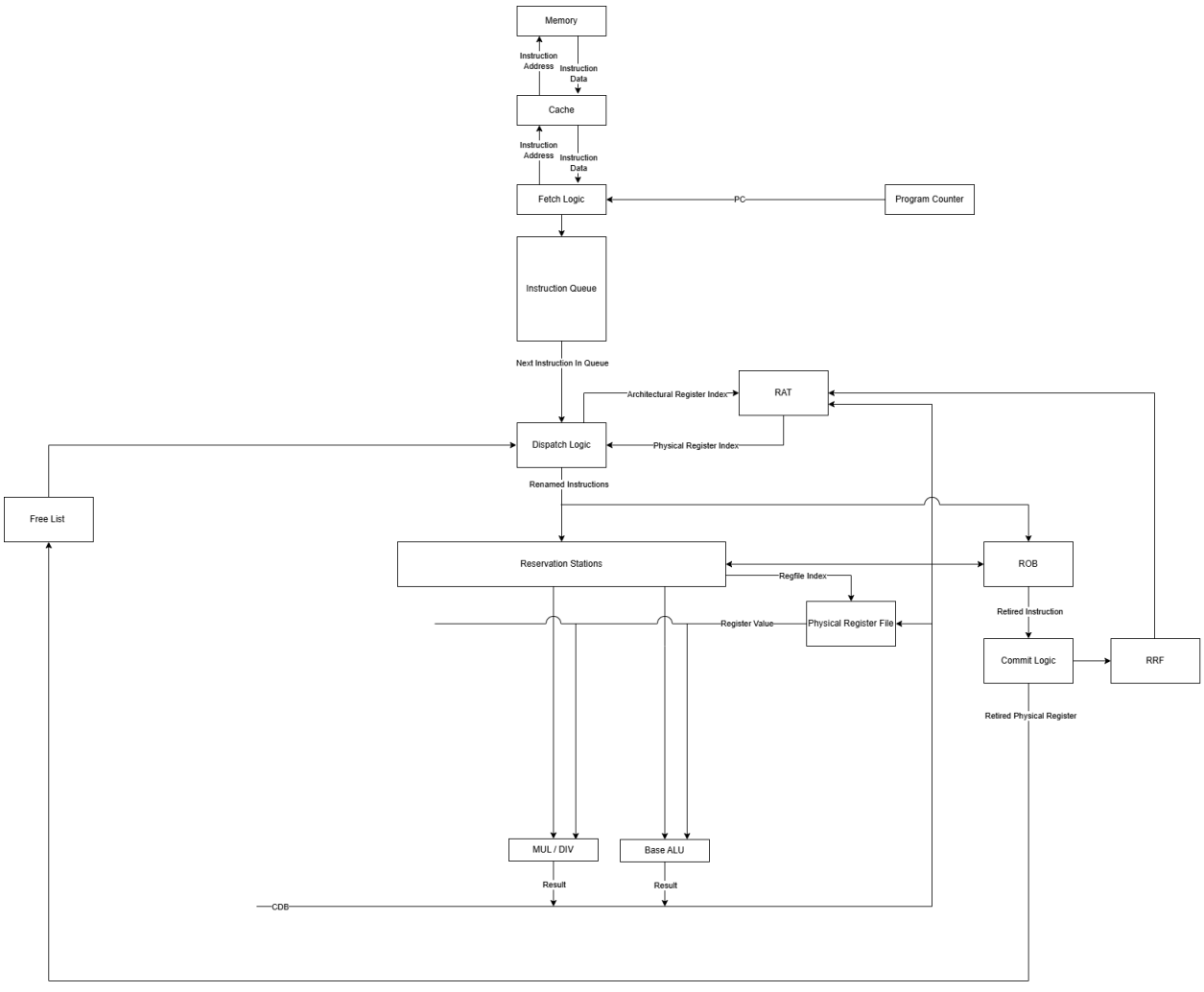

Figure 3: CPU Architecture at Checkpoint 2

Checkpoint 3

At this stage, we have a fully operational CPU implementing the RV32IM ISA. Compared to the previous checkpoint, the base ALU was enhanced to support the AUIPC instruction, and two additional ALUs were added to handle load/store operations and branch/jump instructions. Additionally, a PC module was implemented to perform static not-taken branch prediction and manage branch mispredictions.

Current CPU is the base design for our following optimization for advanced features. The Area used for this CPU is 175595 μm^2, and the max frequency is fixed to 100 MHz. Table 1 below shows the performance of the design at checkpoint 3.

Table 1: Base Design Performance

	IPC	Delay (µs)	Power (mW)
coremark_im	0.4001	7287.18	6.528
aes_sha	0.5251	12433.79	6.478
compression	0.4572	9187.05	6.527
fft	0.5250	9799.01	6.492
mergesort	0.4923	9482.40	6.544

Load and Store

In this checkpoint, we are expected to support all memory instructions. The following diagram is an overview of the parts of the processor that are important to loads and stores:

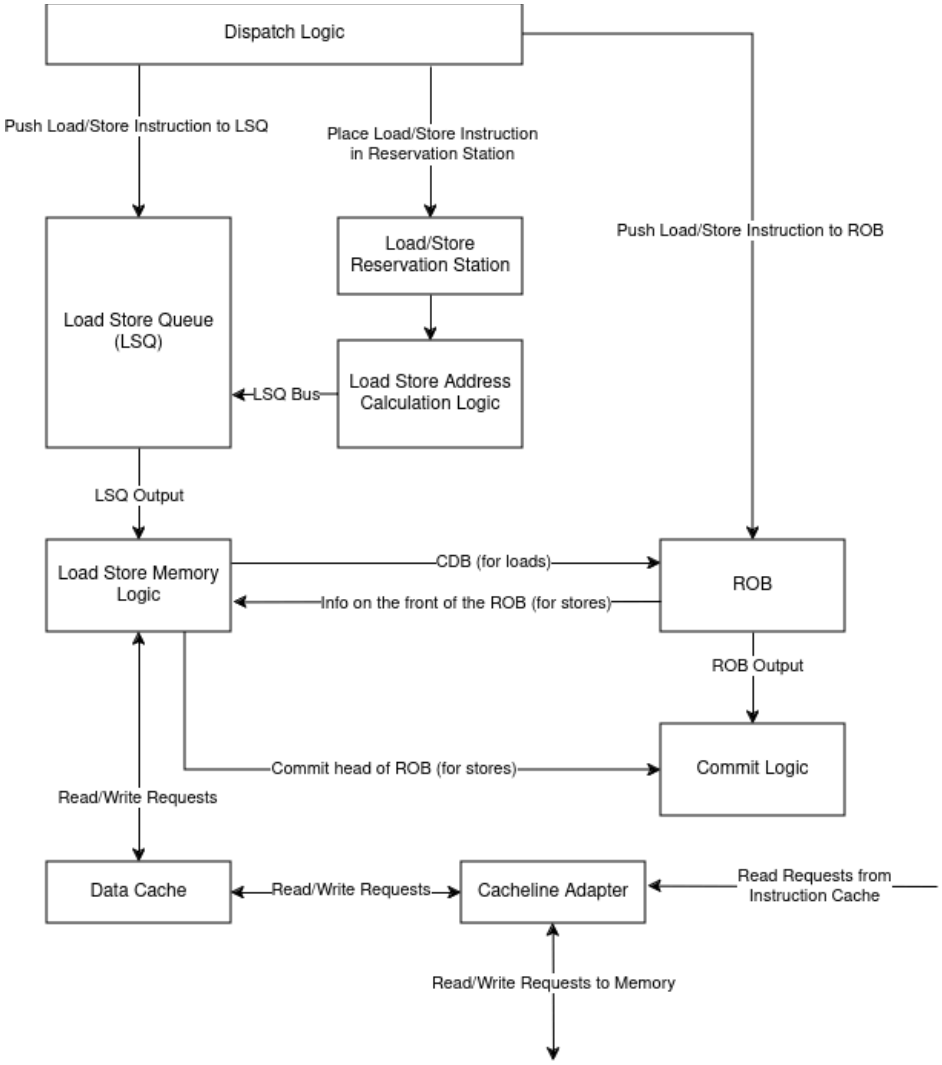

Figure 4: Load Store Logic

When a load or store instruction arrives at the dispatch stage, they are placed into a reservation station and enqueued into the ROB as usual. In addition to that, they are enqueued into the Load Store Queue (LSQ), which is a structure that is similar to the ROB in that it's a queue that can take input from a bus. The bus, much like the common data bus (CDB), can set an element in the queue as ready, and fill in some extra information about the element. The logic responsible for setting an element in the LSQ as ready is the Load Store Address Calculation Logic block, which calculates the address of the load/store operation based on immediate values and registers.

Once an element reaches the front of the LSQ and is set as ready, the Load Store Memory Logic will begin to process it. For loads, it will send a read request to the data cache, and once

the data arrives, it will send the ready signal with the read data over the CDB, similar to any other instruction. Stores, however, must wait for the store to be both at the front of the LSQ and at the front of the ROB before sending the write request to the data cache. This is because any changes done by a store are permanent, so the processor must ensure that the store is meant to happen, and isn't part of a mispredicted branch. When it's at the front of both structures, the write request is sent to the data cache, and when the response arrives, the logic immediately informs the commit logic that the store at the head of the ROB is ready to be committed.

One important thing about this milestone is that the cacheline adapter must handle requests from both the instruction cache and data cache. To do this, an arbiter and state machine are used to govern what cache port has control of the memory input ports. If the data cache and instruction cache both have a request at the same time, the data cache port is prioritized, while the instruction cache request is stored until the data cache is done sending its request. Both the instruction cache and data cache can have a read request in flight at the same time. To govern which port the read data belongs to, we store the address of the incoming read requests from both ports, and compare it to the read address (raddr) port from memory. This port tells the cpu which address the data that came from memory was stored in. This gives us a slight performance boost, as the data cache and instruction cache don't stall each other when one has a read request in flight.

To test this, we had a python script that could automatically generate programs with many instructions. We had this script generate a program with many loads and stores in succession that depend on each other. After checking this program against RVFI and spike, we could be sure that loads and stores were running as intended.

Branch and Jump

The implementation of branches and jumps follows a structure similar to the load / store. For simplicity, all jump instructions are treated as branch instructions. The branching logic structure is illustrated in Figure 5.

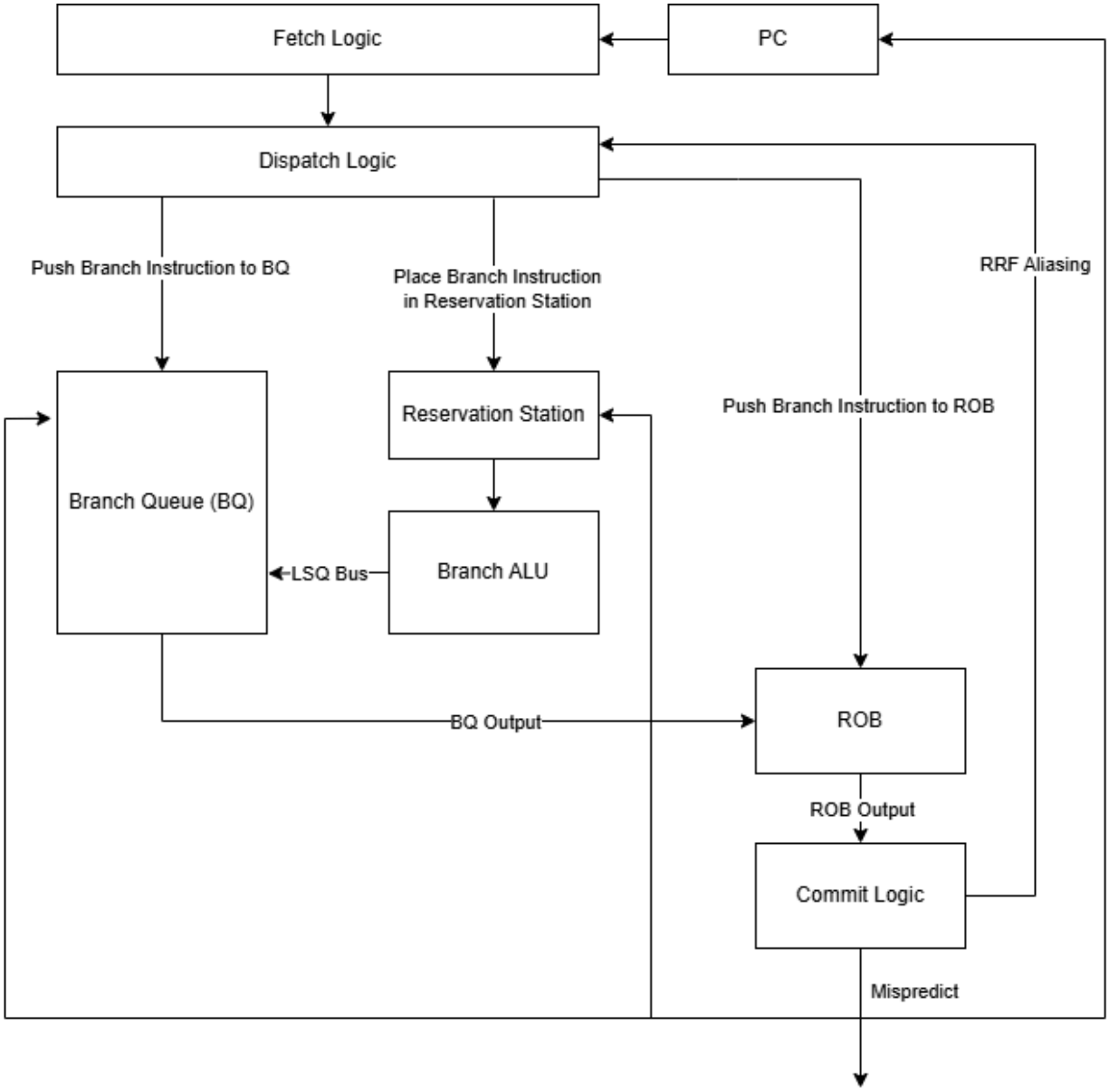

Figure 5: Branch Jump Logic

Upon encountering a branch instruction, the dispatch logic pushes it into the branch queue to await resolution. If a branch is mispredicted, a misprediction signal is propagated throughout the CPU. This signal flushes all instructions in the pipeline, resets the PC to the branch target, and restores the register aliasing in the RAT using the RRF.

Advanced Features

N-way Superscalar

Design

A superscalar CPU is a processor that allows for multiple instructions to be processed at every stage of the pipeline per cycle. This involves modifying every stage and structure in the CPU to allow for more instructions to go through at once. The end result of implementing this feature is a possible IPC of greater than 1 if there are no stalls in execution. For an N-way superscalar processor, the number of instructions that can be handled at each stage is a parameter that can be modified in SystemVerilog. In our case, the value of N can be anywhere from 1 to 8.

The largest obstacle in implementing this feature is allowing for queues to push and pop more than one element per cycle. There were multiple ways to implement this. The first method involved using a queue that only pushes 1 element and pops 1 element per cycle, but making these queues wider to accommodate for more instructions. This would be easy to implement in terms of the queues, as it would only involve making the queues wider. However, there is a chance that some of the registers must be filled with placeholder data in case not enough information is available for the queue to push or pop (for example, filling unused space in an instruction queue element with nop instructions). The method we used involves making a queue that can actually push and pop more than one element. This involves adding more logic to the queue, but it would make the queues more effective at using the space they have available.

Figure 6 is a diagram of how a queue of size N functions:

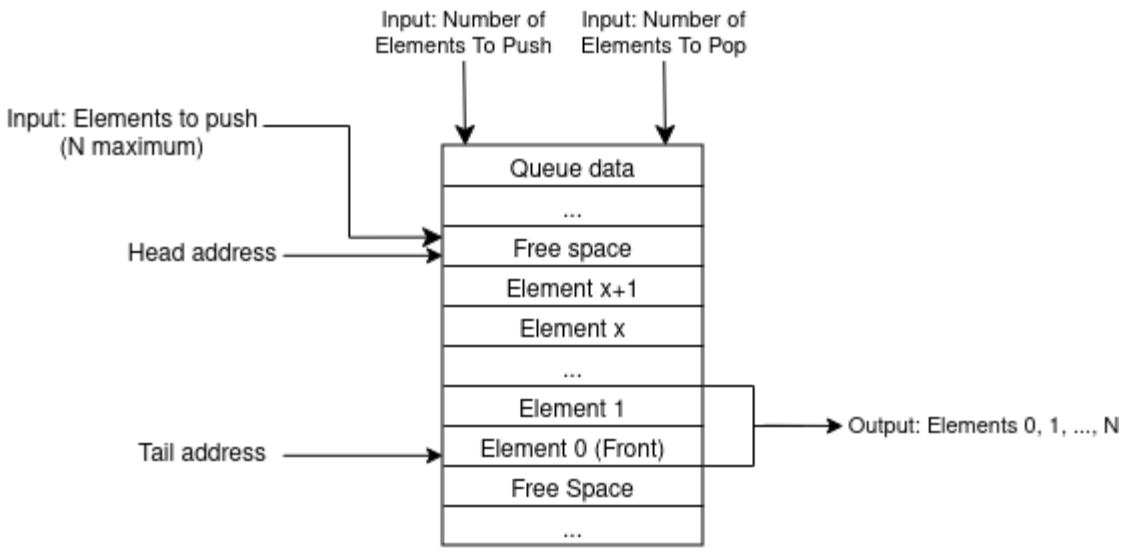

Figure 6: N-Size Queue

The queue exposes the N elements at the front of the queue, starting from the tail, at its output. It can also push an N number of elements at the location of the head address. To indicate the number of elements to push or pop, two input ports are used. Note that the output is constantly displaying the data at the head; the pop signal is only there to remove the data.

There are other concerns to be addressed with superscalar. First of all, multiple execution units have to be able to write to the CDB on the same cycle. To do this, we created a separate CDB for each execution unit, which creates a large increase in area, but is required to allow for multiple instructions in the writeback stage. Second, the dispatch logic and commit logic must also be able to handle more than one instruction. This was mostly handled by a for loop looping the same logic N times, but with some additional tracking combinational logic to keep track of dependencies between the instructions processed in the same cycle. Finally, the fetch logic must be able to handle N instructions. To do this, we made the instruction cache output port output the entire cache line instead of just 32 bits. The fetch logic can then extract the separate instructions from the cache line in the same cycle. This also limits the number of N to 8, as every cache line can only hold 8 instructions due to the size of a line being 256 bits.

To make use of the superscalar processor’s capabilities to the fullest, we also added the capability to add more than one general ALU, multiplier, or divider to the execution stage by changing some parameters. This allows the processor to execute more than one of the same types of instruction in the same cycle.

Testing

To test the correctness of the superscalar processor, we ran all the usual benchmarks and ensured that they passed. However, to ensure that the superscalar processor was able to process more than one instruction at every stage, we wrote an assembly test with minimum stalling to measure whether the processor could reach an IPC of greater than 1 in favorable conditions. This test involved a series of LUI instructions which are not dependent on each other running in a loop. The instructions were put in a loop to ensure that the instructions could be loaded from the instruction cache. Therefore, the only stalling present in this test is from the branch misprediction after every loop. To reduce stalling even further, these tests were done with 16 base ALUs present at the execution stage.

Figure 7 is a graph of the IPC from this test for N = 1 (which is not superscalar) to 8:

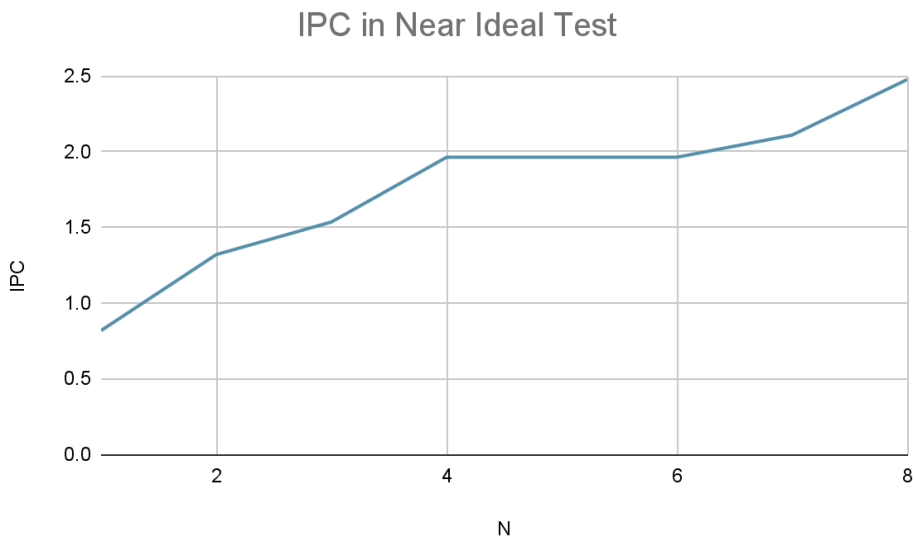

Figure 7: Near-Ideal Test IPC for N-Way Superscalar

As can be seen from the graph, as N increases, the IPC for the test either increases or stays the same. At N = 2, the IPC increases past 1, indicating that more than one instruction is being processed per cycle, and showing that superscalar is functional. At N = 7 and 8, the IPC goes past 2, showing that this N-way superscalar processor can process more than a 2 way superscalar processor per cycle if it is set with the correct parameters.

Performance Analysis

The testing was done with an ideal test case, and with many base ALUs. In practice however, that test was unrealistic, as increasing the number of ALUs increases area and power consumption, and programs run in a real world environment stall much more than the ideal test case. In addition, some tests are more parallelizable than others (can express higher ILP). Therefore, to analyze the performance of this processor, we will fix the frequency and number of execution units. Then, we will measure the effect of the parameter N (superscalar ways) on the IPC in various benchmarks, the area of the CPU, and the power consumption of the CPU in Coremark.

Figure 8 through 10 are graphs of the data collected, with the frequency fixed at 100MHz, and 2 ALUs, 2 multipliers, and 2 dividers. The branch unit and load store unit are also considered two separate execution units, making the total number of execution units 8. The data was collected for N = 1, 2, 4, 6, and 8.

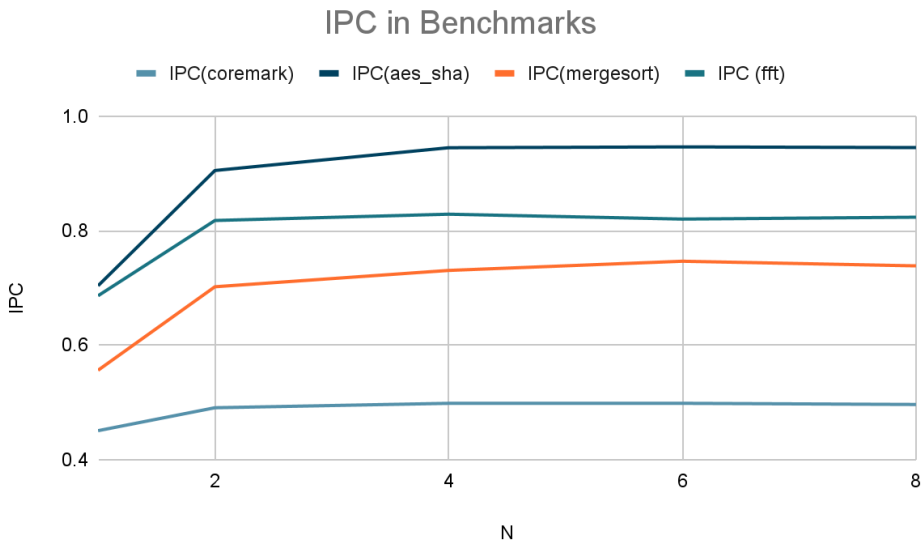

Figure 8: Benchmark IPC for N-Way Superscalar

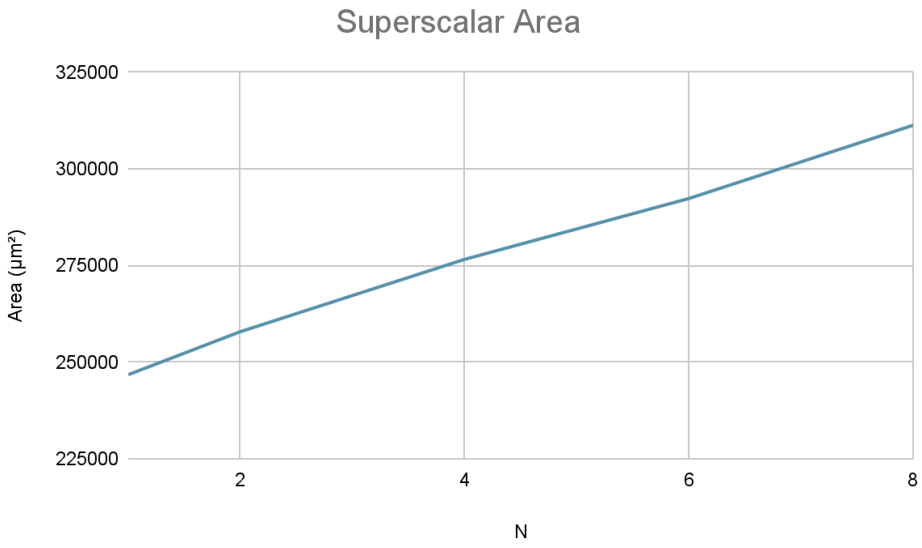

Figure 9: Area Used for N-Way Superscalar

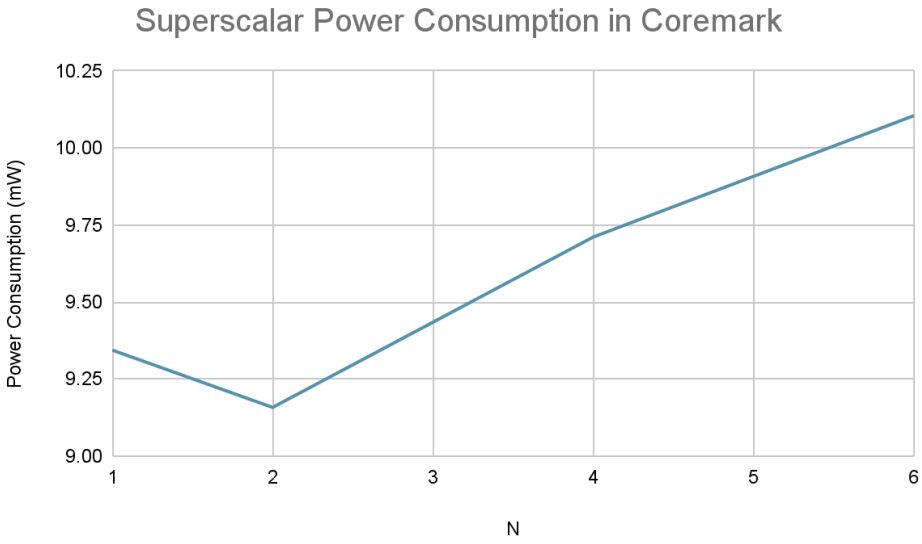

Figure 10: Power Consumption for N-Way Superscalar

For IPC, there are diminishing returns, as the difference between N = 1 and 2 is large, but the difference decreases as N increases. There is a moderate increase between N = 2 and 4 that could make it viable to use as well.

It should also be noted that the IPC for coremark does not increase as much as the other tests when the superscalar ways are increased. This is most likely due to the instructions in coremark

depending on the results of previous instructions more than the other benchmarks. These types of less parallelizable programs do not benefit as much in IPC from a superscalar processor, as the processor can only finish more than one instruction at a time when the instructions do not depend on each other. Due to the increase in area and power caused by superscalar, they can perform worse overall than a processor without superscalar in such non parallelizable tests.

The increase in area is completely linear with respect to the increase in superscalar ways. There is additional logic involved in making sure the dependencies between instructions in the same cycle are resolved, so the increase in area is expected.

The power consumption is much less predictable. It is linear from N = 2 to 4, but there is a decrease from N = 1 to 2. In addition, we did not include the data from N = 8 in this chart, as it was an outlier with a power consumption of 127mW. This unpredictable spike in power is something we noticed more at higher frequencies.

Using all of this data, we have determined that we should use 2 superscalar ways. There is little difference between the IPC of N = 2 and N = 4 to warrant the linear increase in both power and area. In addition, by using N = 2, we can increase the frequency of the CPU even further, as the critical path created from the superscalar logic decreases in length. This makes up for the smaller IPC.

Stream Buffer Prefetching

Design

Our superscalar processor can fetch and execute multiple instructions per cycle. However, that is only in the ideal case where the instructions are already in the instruction cache. In reality, many of the instructions will most likely be in memory, and fetching the instructions from memory is a large bottleneck to our superscalar processor. Thus, we decided to design a prefetcher that can read instructions from memory before the cache asks for them.

In stream buffer prefetching, when the cache asks to read from instruction memory, the cacheline adapter will also read a number of addresses ahead in memory at the same time. The adapter will respond to the cache with its current request, and store the data from the other reads it made in a buffer. When the cache then asks to read the next address in memory, the cacheline adapter can then respond with the data it stored in the buffer from the previous read. This can heavily decrease the number of cycles spent on instruction cache misses, which will increase the IPC of the processor.

Figure 11 is a diagram of our stream buffer prefetching mechanism:

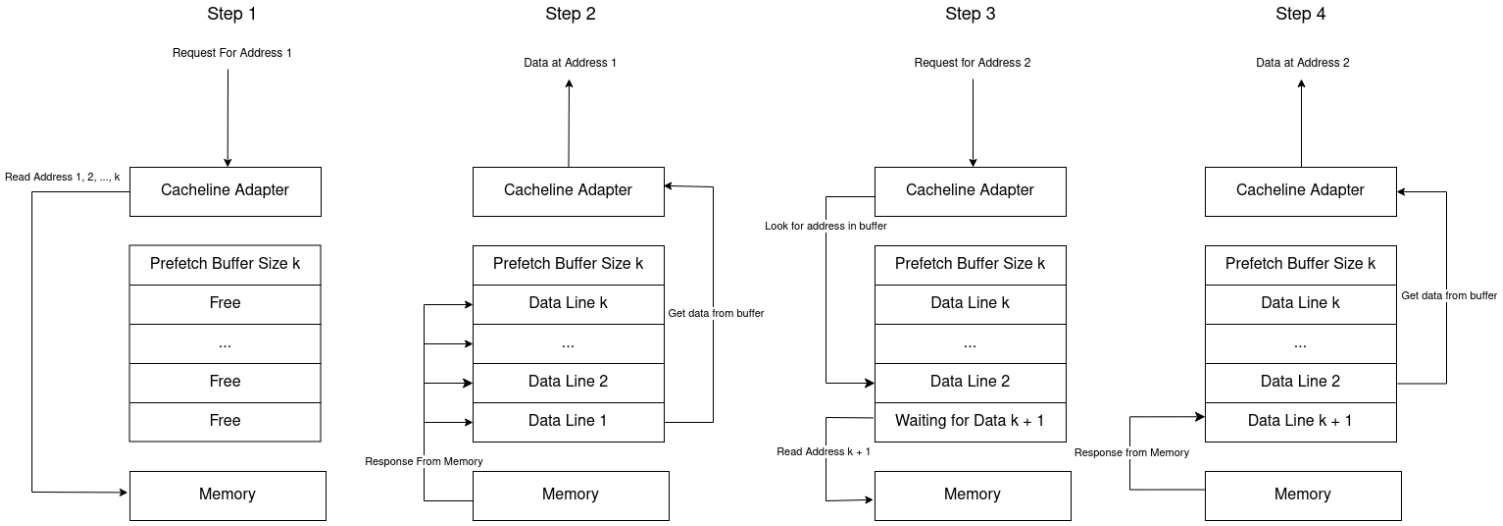

Figure 11: Stream Buffer Prefetching Structure

In step 1, the cacheline adapter gets a request from the instruction cache for data line 1. Since the data it has been requested to fetch is not in the buffer, it will send multiple requests to memory for enough data to fill the entire buffer of size k lines. The responses from memory will then be stored in the buffer. When the response for data line 1 arrives, the cacheline adapter

can respond to the request in step 2. Then, when the cache requests for data line 2 in step 3, the cacheline adapter will find the data request in the stream buffer. It will then respond with that data in step 4.

Something else to note is that, as soon as any data is retrieved from the buffer, the buffer will send a read request to memory to read memory at k + the address of the retrieved data. This can be seen in step 3, and the response comes in step 4 to fill in the space of the completed read request. This allows the stream buffer to constantly have data available for contiguous blocks of instructions in memory.

Testing

To test this processor's correctness, we wrote a program that contains a large amount of load and store instructions. The buffer will have to interface with memory, and the memory ports will be in use by both the instruction cache and data cache. This test ensures that the instruction cache's stream buffer did not interfere with the usual operations of the data cache.

Second, to test whether this feature works as intended, we created an ideal test that will see a heavy increase in IPC if the prefetch is working correctly. This test program contains 900+ LUI instructions in series that do not have data dependencies with each other. The biggest reason for the processor to stall in this program is if it cannot fetch the instructions from memory fast enough, which is what we aim to mitigate with this prefetcher.

Figure 13 is a graph of the IPC in this test with stream buffer sizes 0 to 8. Buffer size 0 doesn't exist, but we used it to indicate performance without the prefetcher. We also graphed the average number of cycles taken to resolve an instruction cache read miss. Note that these tests were done at 625MHz, with a 2 way superscalar processor and 2 general ALUs:

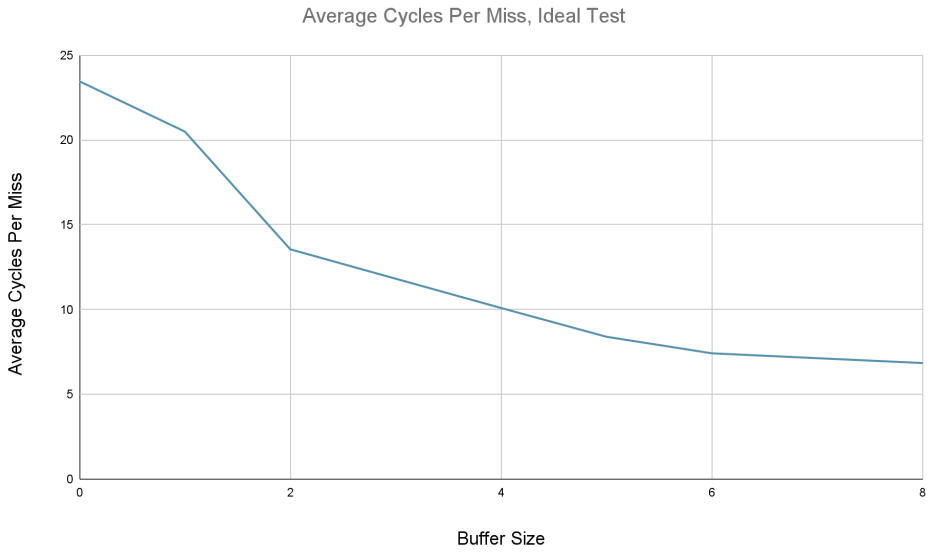

Figure 12: Ideal Test Average Cycles per Miss for N-Size Stream Buffer

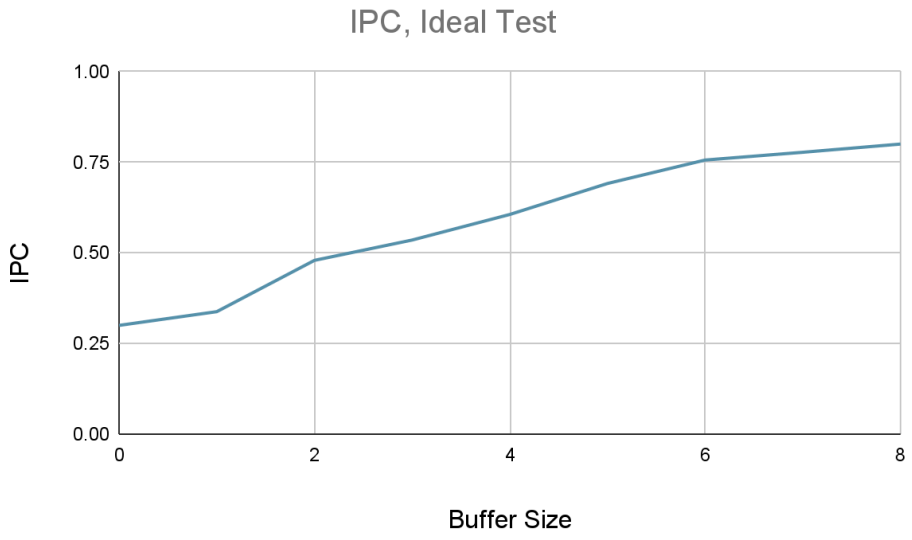

Figure 13: Ideal Test IPC for N-Size Stream Buffer

As can be seen, the average cycles per miss decreases, and IPC increases as the buffer size of the prefetcher increases. This shows that the prefetch is effective at fetching contiguous instructions.

Performance Analysis

The testing done in the ideal test case is not representative of most real world applications. This prefetcher does well in programs where the instructions executed come in succession. Therefore, in programs with more jumps/branching, especially with jumps that go to distant addresses, stream buffer prefetching will not perform well. In fact, in some tests with heavy branching the prefetch might make fetching instructions slower, as it could be wasting memory bandwidth fetching instructions that will never be requested.

We tested the IPC and average cycles per instruction cache miss, as we did in the test case, but for the two benchmarks coremark and aes_sha. We varied the buffer size from 1 to 8, and are using a 2 way superscalar processor at a frequency of 625MHz.

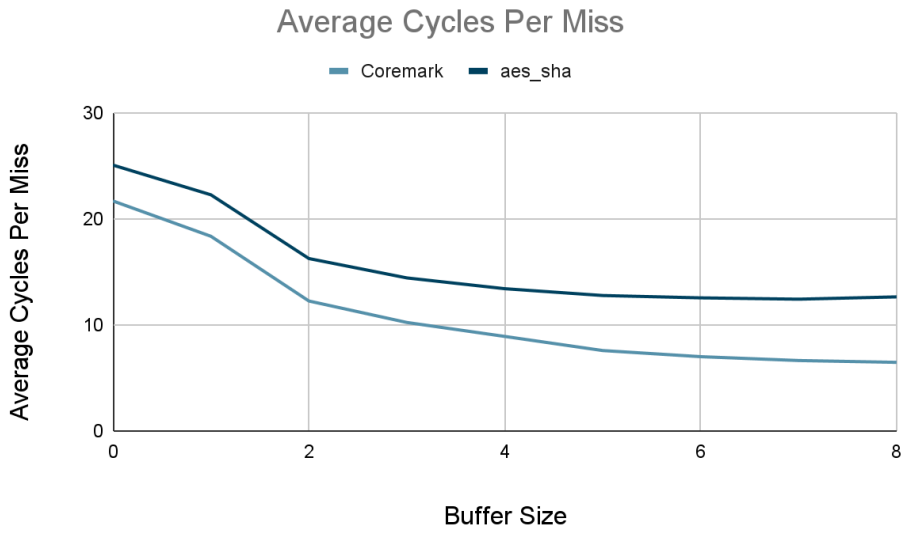

Figure 14: Benchmark Average Cycles per Miss for N-Size Stream Buffer

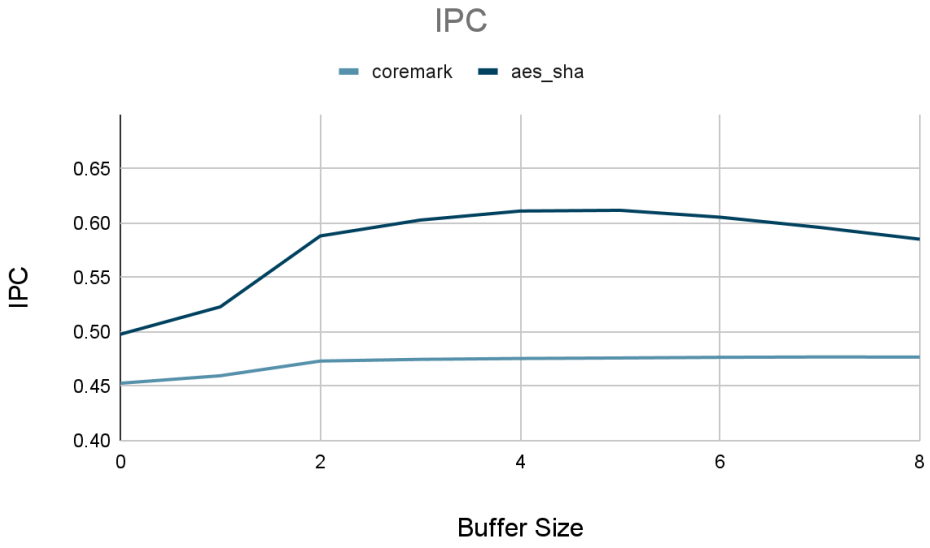

Figure 15: Benchmark IPC for N-Size Stream Buffer

Both tests saw a large reduction in average cycles per instruction cache read miss. However, these differences did not translate as well into IPC as the ideal test case due to branching. Coremark saw a very slight increase in IPC from buffer size 1 to 2, but it was mostly linear, showing there was little benefit from the prefetcher. Aes_sha saw much more improvement, although it also shows a slight decrease in IPC towards the larger buffer sizes. As discussed before, prefetchers could waste memory bandwidth fetching instructions that are not needed, and this effect grows larger as the buffer size increases.

After looking at both the IPC graphs and the additional power and area consumption of using larger buffers, we decided to use a prefetch buffer of size 3. This number of lines is enough to improve the IPC of some programs like aes_sha, while still not heavily impacting the power consumption and area of the processor.

Post-Commit Store Buffer

Design

While our processor can execute multiple arithmetic instructions per cycle and out of order, memory instructions still must be executed and committed one at a time and in order. The data cache is only capable of processing one request at once, which for cache misses, leads to slowdowns in commit for memory instructions. However, there is no reason to wait for cache response to commit store instructions. Thus, for our final advanced feature, we added a post-commit store buffer to buffer stores and allow for committing store instructions without a cache response. This relieves back pressure on the ROB for store instructions and allows for data forwarding for loads.

The post-commit store buffer is a queue structure after the LSQ that contains post-commit store instructions. Whenever a store is committed and dequeued from the LSQ, it is enqueued, and whenever the store is requested from the data cache, it is dequeued. The queue ensures that stores are requested in order for data correctness. During typical operation with only store instructions, the store buffer continuously sends write requests to the data cache to empty the store buffer.

If instead the LSQ dequeues a load instruction, then there is a possibility of forwarding. Forwarding refers to the case where the load address matches the address of store(s) in the store buffer and the load byte mask (rmask) and store byte masks (wmask) match, indicating that the data to be read has not been written to the data cache/DRAM.

- In the case of forwarding, there is no need to send a read request from the data cache and the data can be reconstructed using the data to be written (wdata) in the entries of the store buffer. This offers speedup for many memory operations that occur in the same region of memory.
- In the case of no forwarding, the LSQ waits for the current write request to complete (if there is one), then sends the read request and commits/dequeues the LSQ when the data cache responds. In this scenario, no speedup is achieved from the store buffer.

Testing

To test the correctness of the store buffer, we ran all the given benchmarks and ensured they passed. To test if this feature offered a speedup, we utilized the python script to generate a

large number of spatially localized and unlocalized memory instructions (increasing the probability of data forwarding). Then we compare the IPC of these randomly generated programs without and with the store buffer. In this idealized test, the IPC difference should be largely noticeable. This will be evident in the performance analysis.

Performance Analysis

We perform a performance analysis of the store buffer and determine how much it improves our processor. Here, the primary parameter to change is the size of the store buffer. We define a store buffer size of 0 as the processor without a store buffer.

First, we compare the average cycles per store and load instruction. This is measured as the time the instruction is dequeued from the LSQ to the time it is finally committed. For stores, the benefits are very clear: in all tests, the average cycles per store decreases dramatically (Figure 16).

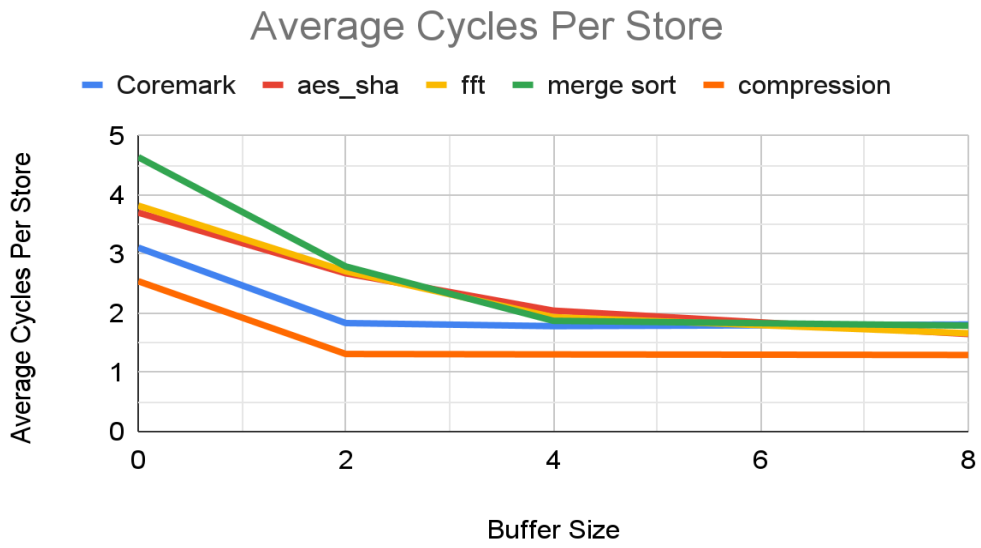

Figure 16: Average Cycles per Store Instruction

Similarly for loads, in most tests, the average cycles per load decreases, but only slightly (Figure 17). This is expected as the store buffer primarily improves store instructions, and only improves load instructions in the case of forwarding. We measure the percentage of forwarded loads (Figure 18), which agrees with the results from Figure 17.

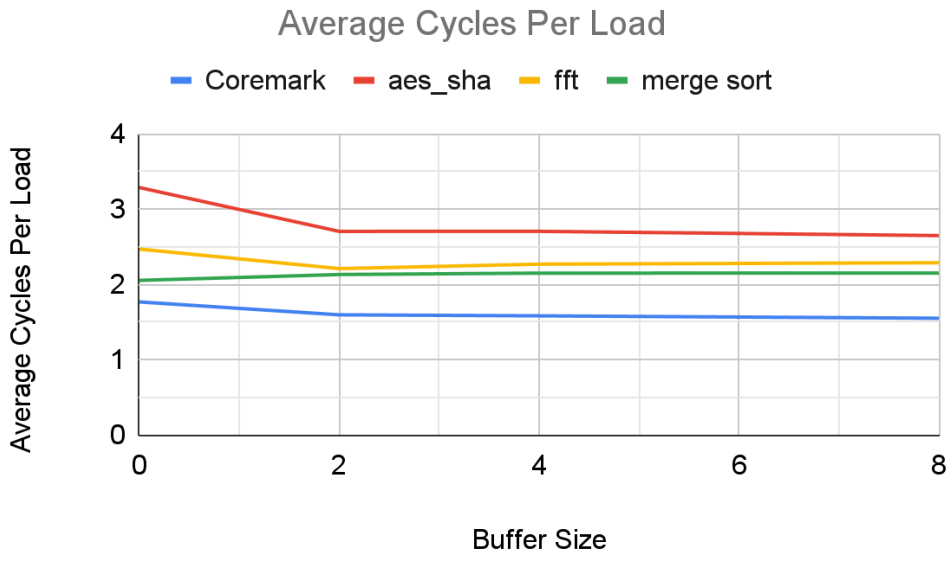

Figure 17: Average Cycles per Load Instruction

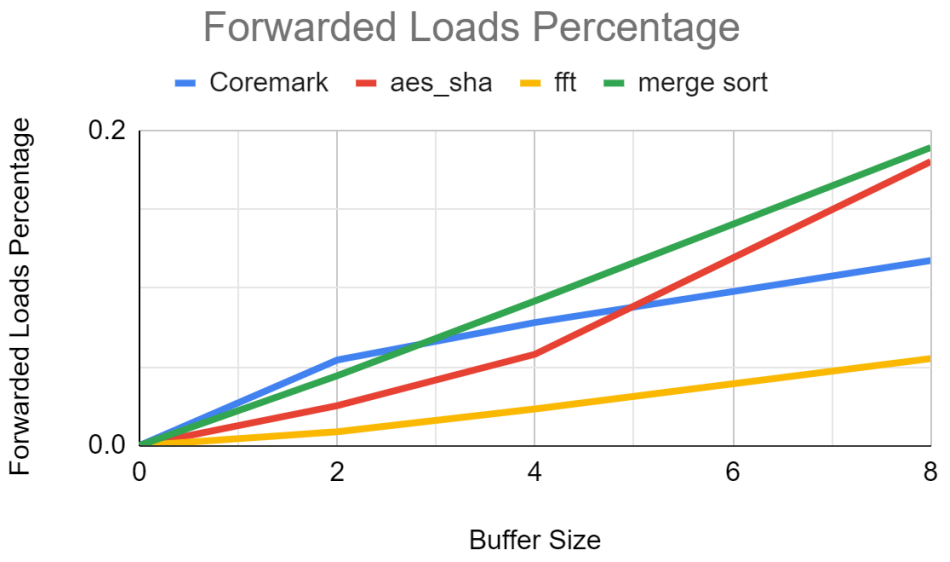

Figure 18: Percentage of Forwarded Loads

Lastly, we compare the increase in IPC according to buffer size (Figure 19). In all tests, the IPC improved, but with diminishing returns past buffer size of 8 (not shown). Past a size of 8, the store buffer would almost never be full and increasing the size would only introduce more area for very little IPC. Thus, in our final design, we settled for a buffer size of 8.

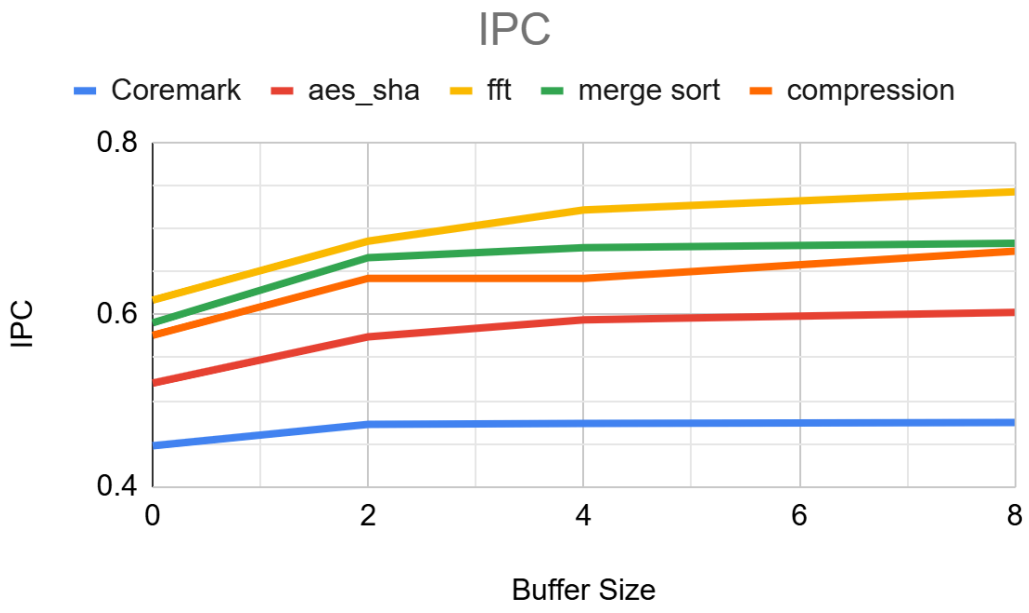

Figure 19: IPC vs. Post-Commit Store Buffer Size

Additional observations

Splitting MUL / DIV ALU

From checkpoint 2, the MUL/DIV ALU module contained both signed and unsigned multiplier and divider Synopsys IP modules. For each mul/div operation, all modules would run in parallel and complete their operation in the same number of cycles (i.e. the multiplier and dividers were configured to contain the same number of stages). This configuration led to significant underutilization of resources when a high number of multiplication and division instructions were present.

To optimize this, we first split the MUL and DIV operations into two independent ALUs: one for multiplication and division. With a unified reservation station, this adjustment required minimal effort. Realizing that our critical path almost always resulted from the dividers, we changed the number of stages of both multipliers and dividers such that the critical path of both were roughly of equal length. In the end, we settled for a 3 stage multiplier and 17 stage divider.

Next, we attempted to remove the unsigned multiplier/divider module from our CPU using only a 33-bit signed multiplier/divider module. However, this approach introduced stricter timing requirements. The addition of an extra bit and the need for more signed calculations also increased the CPU's power consumption. Due to these drawbacks, this optimization was ultimately discarded.

Branch Prediction

After integrating all the advanced features into our CPU, we observed that neither increasing parallelism in the superscalar architecture nor enhancing prefetching significantly improved performance. Upon inspecting waveforms during benchmark execution, we identified excessive branch flushes as a critical issue. The static not-taken branch prediction mechanism had become a major performance bottleneck.

To address this, we attempted to implement a Gshare branch predictor paired with a fully associative branch target buffer (BTB), as illustrated in Figure 20. While the branch predictor performed well in isolation, integrating it with the CPU required more extensive modifications than anticipated. This complexity prevented us from submitting a functional version of the branch predictor before the competition deadline.

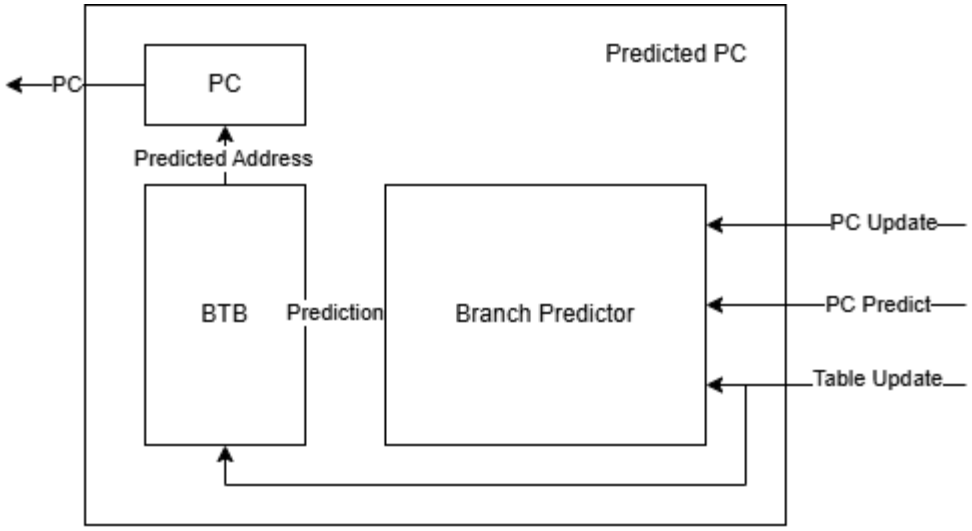

Figure 20: Branch Predictor Architecture

Conclusion

In conclusion, we have achieved our goal of designing an out-of-order CPU that supports RV32IM ISA. The CPU uses an explicit register renaming 2-way superscalar architecture, with a stream buffer prefetcher, post-commit store buffer, along with a pipelined cache. For our final design, we were able to achieve the following:

Table 2: Final Design Performance

	IPC	Delay (μs)	Power (mW)
coremark_im	0.4747	982.65	34.547
aes_sha	0.6027	1733.16	34.534
compression	0.6739	1107.83	34.652
fft	0.7430	1093.49	34.716
mergesort	0.6830	1332.48	34.313

The area used for the final design is 263461 μm^2, and the max frequency is fixed to 625.00 MHz. The CPU achieved a significant increase in IPC and improved program runtime in all tests. While the power consumption and area was greater, the improvement in program delay caused our overall performance to be higher than the base design.

Future Work

For the next steps in our project, we have identified several directions for improvement. First, integrating the branch predictor with the CPU is a priority, as it would likely unlock more of the potential offered by the superscalar architecture. Second, we could expand the CPU's capabilities by adding additional RISC-V extensions, such as floating-point operations. Lastly, an exciting prospect would be to bring our CPU design into the real world by migrating the project onto an FPGA for hardware realization.