ECE 411 FINAL

REPORT

Team: Robert Dennard Max Graphics Society (RDMGS)

Author: Charles Lai (jiayeyl2), George Vintila (gvinti2), Gabe Wozniak (gfw3)

Mentor: Guo, Jason

Professor: Prof. Kim, Nam Sung

Dec. 11, 2024

Table of Contents

1. Introduction & Project Overview

The project focuses on the design and implementation of an out-of-order (OoO) RISC-V central processing unit (CPU) with explicit register renaming (ERR) and several advanced features. In modern processors, OoO execution allows instructions to be processed as soon as resources become available instead of having to follow programming order. Thus, time can be saved as we can execute multiple instructions in parallel. By using ERR, we solve certain false dependencies that can be caused from write after write (WAW) and write after read (WAR). The CPU is further improved with multiple advanced features such as early branch recovery (EBR), prefetchers, and split load store queue (LSQ). Completing this project provides hands-on experience with the underlying principles that drive modern processor design, offering insights into the challenges and trade-offs involved in creating efficient, high-speed computational systems.

2. Design Description

2.1 Overview

This section provides a comprehensive description of the design and development process for our OoO CPU. Initially, we focused on completing and testing the core functionality of the CPU through three distinct checkpoints to ensure the correctness of our OoO CPU. Following this, we dedicated three weeks to incorporating advanced features to improve the CPU's performance. These enhancements are targeted to decrease delay while maintaining minimal area and power consumption.

2.2 Milestones

The completion of the project was divided into three checkpoints each lasting a span of one week. We will further discuss the developments made in each checkpoint below:

2.2.1 Checkpoint 1

The first checkpoint focused on laying the foundational components of the OoO datapath and implementing critical building blocks for instruction execution. A design block diagram of the OoO datapath was drawn as shown in Figure 1 to ensure a clear visualization of the system's architecture and that all team members share the same goal. Additionally, we implemented and tested a parameterizable queue that supports configurable depth and width, with thorough validation against edge cases typical in FIFO behavior. The tests demonstrated the queue's correct handling of data, ensuring that boundary conditions such as underflow, overflow, and simultaneous read and write were properly managed. At the same time, a cacheline adapter was implemented to decode the

bursts from the given DRAM model. This adapter was integrated with the instruction fetch unit, which was initialized to the program counter value of 0x1ECEB000. The integration of these components allowed for a smooth transition to the later phases of the project.

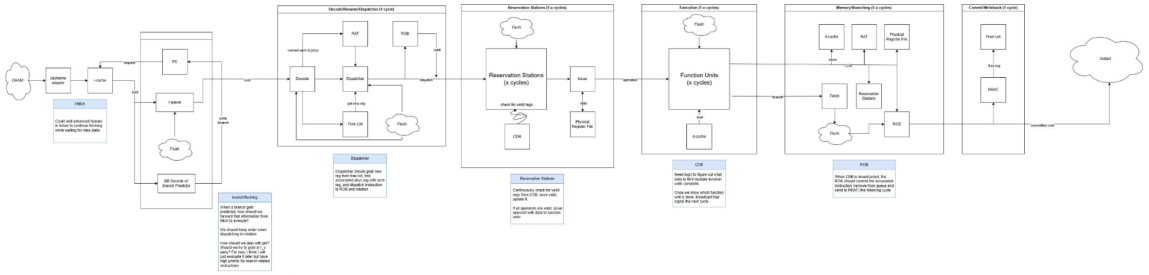

Figure 1. Stage Diagram of the Baseline Design

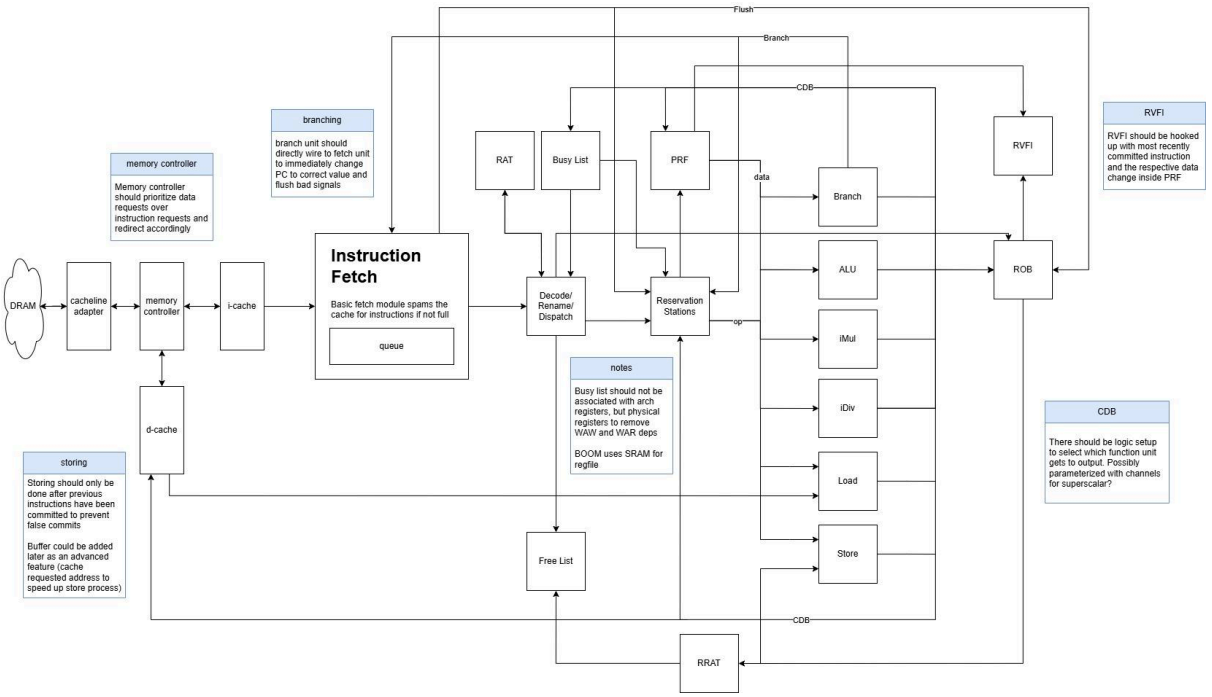

Figure 2. Block Diagram of the Baseline Design

2.2.2 Checkpoint 2

At the second checkpoint, the main goal was to upgrade our CPU such that it could support all immediate and register instructions in RV32I and RV32M. We successfully integrated the pipelined multiplier IP and pipelined divider IP to handle MUL/DIV/REM instructions. To verify the functionality of these instructions, we modified the random testbench from mp_verif to include coverage for all instructions targeted in this phase to account for all edge cases. Furthermore, we demonstrated that the processor was capable of executing instructions out-of-order by using a dedicated test code (ooo_test.s) and checking waveform with Verdi manually. The processor also passed linting through Spyglass and Verilator. These integrations and verifications marked significant progress in bringing the processor closer to full functionality.

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Figure 3. Supported RV32M Instructions

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Figure 4. Supported RV32I Instructions

2.2.3 Checkpoint 3

The third and final checkpoint was focused on adding memory handling and support for control flow instructions such as branches and jumps. We also included both data cache and instruction cache into our architecture. This included adding any required write logic and arbiter functionality to the cacheline adapter and memory controller (our

arbiter), ensuring that memory transactions were properly handled. We decided to keep this simple at first with the memory controller acting as a state machine that only services one cache at a time if a request is made. We iterated upon this simple design once the prefetchers were designed and implemented.

At this stage, a primitive load store queue was also added to the processor, and proper logics were added such that memory instructions were executed in order. Additionally, our implementation ensured that load was executed before ROB commit and only stores were required to wait for ROB commit for execution to further enhance performance. In addition to memory instructions, control instructions such as branches (BR), jumps (JAL/JALR), and the AUIPC instruction were implemented. As branch prediction was not required at this stage, only a static branch prediction (always taken) was employed for simplicity. This checkpoint marked the completion of our CPU as all instructions shown in Figure 2 and Figure 3 were supported as requested by our processor. It also laid the groundwork for more advanced features like branch prediction in future phases.

2.3 Advanced Design Options

2.3.1 EBR

In an OoO processor without EBR, the most general scheme for handling control flow mispredictions involves waiting until the mispredicted control instruction commits before flushing the entire CPU and redirecting fetch to the correct target. This is a severe waste, since we detect mispredictions far earlier than we commit the mispredicted instructions and the time spent waiting directly adds to the mispredict penalty. In fact, performance counter data shows that the average penalty for the late flushing scheme in our CPU is twelve cycles for every misprediction. EBR or early branch recovery is an architectural

feature that allows mispredictions to be corrected as soon as they are detected by tracking which in-flight instructions should or shouldn't be invalidated by a branch misprediction.

2.3.1.1 Design Overview

At the core of our EBR scheme are branch masks which are assigned to every instruction at dispatch. When a branch goes through dispatch, it is able to reserve a bit in the branch mask, and all trailing instructions will have a 1 on this bit of their branch mask until the branch is resolved. On a misprediction, this branch mask is bitwise ANDed with the bit reserved by the branch to determine if the instruction should be flushed. On the other hand, if a branch was accurately predicted, the bit is set to 0 on all branch masks essentially recycling this bit so it can be reserved by a new branch. Additionally, in the case of misprediction, certain CPU state information needs to be reset including the RAT, and the freelist which we refer to as EBR checkpoints. Since this information is expensive to store, and we will rarely have cases where the number of in-flight branches hits the maximum allowed by the CPU, we parameterized the number of EBR checkpoints. In the case where all checkpoints are reserved, additional branches will default to late flushing behavior. In the final version of the design, the average misprediction penalty (late flushes and early flushes) was reduced to about 4.5 cycles.

2.3.1.2 Performance Analysis and Trade-offs

The branch mask scheme is by far the most sensible option when considering ways to implement EBR. While we originally considered using ROB IDs to track branches instead of bitmasks, this idea quickly became unsavory when we considered the overhead of many comparators needed to implement this. Alternatively, using a bitmask to manage the logic for invalidating instructions and updating EBR information relies on only bitwise ANDs - a cheaper solution by far. As for parameterizing the number of EBR checkpoints, we've already mentioned how this decision can reduce the amount of

expensive EBR state information we need to store at the cost of infrequently late-flushing. However, additionally, the number of checkpoints directly corresponds to the width of the branch mask on every instruction, and when this logic is duplicated across the CPU, saving even just a single bit can make a big difference. However, EBR was not all smooth sailing. The biggest challenge we encountered with EBR resulted from the difficulty of implementation. Since EBR is an architectural change for the whole CPU, even people developing features unrelated to branches like the LSQ had to maintain compliance with precise EBR branch mask policies. In fact, a small mistake implementing EBR in a new LSQ feature led to one of our most difficult bugs to find and fix. Finally, an oversight that we only realized after the competition, is that we did not set up JALR instructions to work with EBR though it would've been minimal overhead to make this modification. The result of this is that all JALR instruction mispredictions cause late flushes.

2.3.2 Upgraded Memory Controller with Prefetching

2.3.2.1 Design Overview

We decided it would be best to alleviate our cache miss penalties through incorporating prefetchers for each cache. There are a multitude of different ways to incorporate prefetchers in processors but we decided that it would be best to incorporate it into our memory controller, which acts as the arbiter for our memory system. However, to take full advantage of the DRAM model, we had to restructure our memory controller from CP3 to allow for higher request throughput along with prefetching capabilities.

The design premise is that each cache has its own cache buffer controller (CBC) which saves tags to request to DRAM based on previous miss requests. These buffer controllers hold buffers which act as queues containing our prefetched data. Whenever a new cache request is being serviced, the cache checks the heads of its respective CBC to see if the prefetch is already available, if not, then a new request is made to DRAM and

the buffer controllers update their state based on the missed request. The memory controller needs a new structure to handle multiple pending OoO requests, so we introduced the data request table (DRT), which keeps track of the tag sent out from each CBC, their respective cache, and buffer index. Whenever a response is received from DRAM, the DRT checks if the tag is matched in the table and sends it to the respective CBC slot it was requested from. The prefetched cacheline data was originally supposed to be inside the buffers themselves until we realized that storing cacheline data in flip flops cost too much area and power, so we switched to a main SRAM block that stores all of our prefetched data from both buffers.

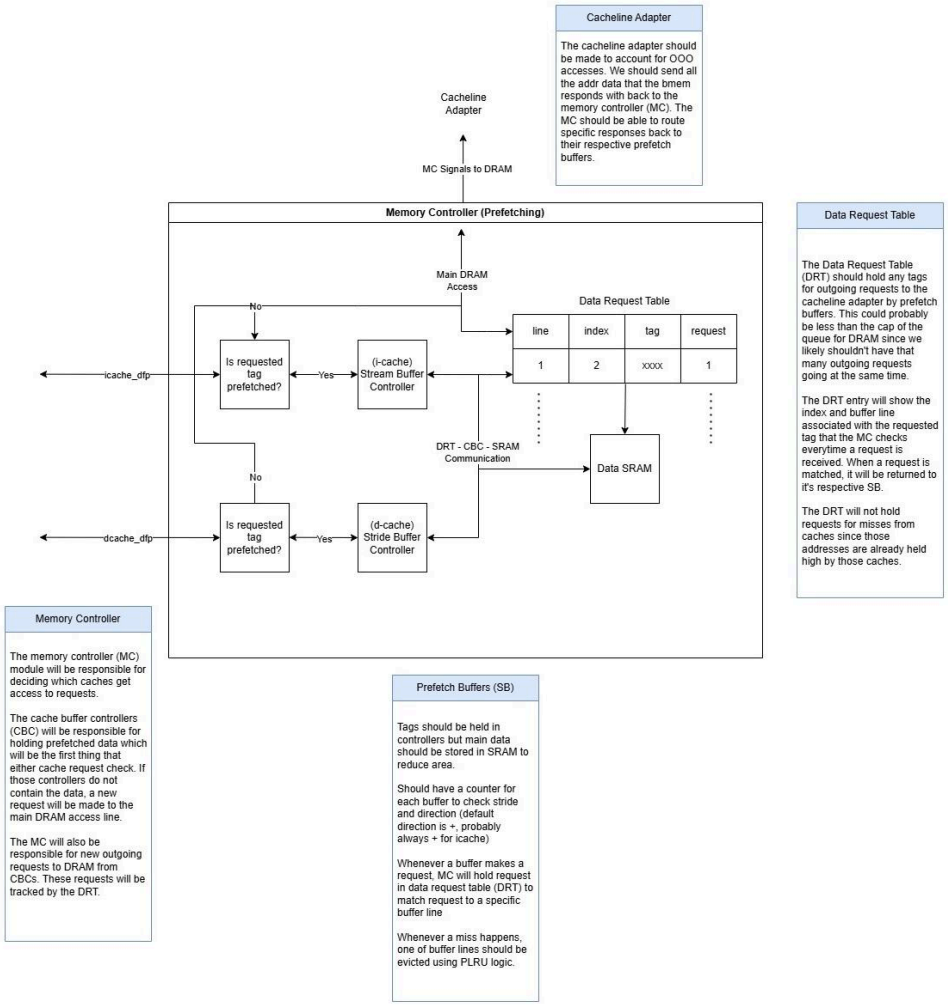

Figure 5. Block Diagram for Upgraded Memory Controller

There are multiple caveats to this design that we need to take into account, mainly on how to effectively manage servicing multiple different structures at once while maintaining optimal area and power. There are four different structures that can make outgoing requests (I-CBC, D-CBC, I-Cache, and D-Cache) and three that can take incoming requests (DRT, I-Cache, and D-Cache). Only one outgoing request and one receiving request can happen per cycle based on the current memory model. To prevent multiple conflicting requests, the hierarchical priority for each structure is: I-Cache > D-Cache > I-CBC > D-CBC. The design idea is that the I-Cache should be prioritized first before the D-Cache due to higher instruction throughput and the caches should be serviced before any new prefetches happen. However, the receiving requests can all take the same receiving request if available. If the cache takes data that was previously requested by a prefetcher but not yet placed inside the prefetched data SRAM, the associated prefetch slot is dequeued in the CBC and no data will be inserted, since the prefetch was already taken. Any structures that are waiting to get serviced will stall until it's their turn.

Finally, writes for the D-Cache operate in a similar manner compared to CP3, but now the D-CBC needs to check if the D-Cache wrote to any previously prefetched cachelines. If this occurs, the D-CBC must evict that specific line to ensure that faulty prefetched data doesn't get passed to the D-Cache.

2.3.2.2 Next-Line and Stride Prefetching

Both cache buffer controllers (CBCs) have fairly similar logic in terms of how they communicate with their respective caches and enqueue and dequeue data. They both have different buffer lines, allowing for multiple possible prefetch options to check for their respective caches. These lines are evicted on a PLRU policy, similar to how we designed

mp_cache, but there are only two lines instead of four. While both CBCs have similar logic, we decided to design a different prefetching scheme for both I-Cache and D-Cache as they have different memory access patterns by most test programs.

The I-CBC uses next-line prefetching since in most cases where branches aren't occuring, the I-Cache will want to take the very next cacheline on a miss. Whenever the I-Cache checks the I-CBC and detects a miss, the associated PLRU line for the I-CBC is immediately evicted and a new tag based on the predicted next line in the new buffer is enqueued. The CBC will manage new requests to insert to the DRT whenever a new request is available to be serviced. Whenever the DRT acknowledges that a request is received, a new slot will be enqueued into the buffer with the old tag incremented by 1. Whenever the DRT receives a matching data response from DRAM, the CBC will receive this information to know that the prefetched data is available and stored inside the data SRAM.

The D-CBC uses a stride prefetching mechanism, since data memory requests are not often as simple as getting the next available line for the case of the D-Cache. To detect these strides, a new structure is used called the repeat pattern table (RPT), which is comprised of logic surrounding a large SRAM data block. Each RPT entry in the SRAM block contains bits corresponding to tag, base address, stride distance, direction, and count. Whenever a cache miss occurs in the D-CBC, instead of eviction, this tag is routed to the RPT and compares among 4 ways based on the set addressed into the tag (similar to how it's done in mp_cache). If a miss occurs, the PLRU way is evicted and a new tag is inserted based on the missed tag. If a hit occurs, the associated way is updated with new information.

The general idea of how the RPT gets updated is determining to see if there's a valid stride direction and distance among previous patterns. On the first request, the data is

inserted. On the second request, the distance and the direction from the first request is recorded. Finally, the third request confirms if this stride in the set direction has continued to form. If the third request confirms an existing stride is present, the associated buffer lines inside of the D-CBC are evicted and placed with the new stride pattern. These buffer lines operate in the same way as the I-CBC buffer lines but update the newest requested tag based off of the recorded stride.

2.3.2.3 Performance Analysis and Trade-offs

In the vast majority of test cases, the prefetching improvements for the I-Cache helped significantly. For the original memory controller design, an I-Cache DFP read would take on average 15 cycles for a response (based off of *Coremark*). The new prefetcher enables the I-Cache DFP read to respond within 5-10 cycles depending on the program, with a fairly high prefetch hit rate of >75%. As predicted, the prefetcher would suffer from branch heavy programs due to the simple miss mechanism implemented. Additionally, the prefetcher allowed the fetch queue to have data sooner and more frequently, which resulted in less stalls due to a lack of instructions to fetch, leading to a bump in IPC and moving the bottleneck to other areas of the CPU design related to the dispatcher and execution unit stages.

The D-Cache prefetcher started to showcase some of the flaws present in this memory model. For the original memory controller design, the D-Cache DFP read had a similar cycle time of 15. The new model showed a wide variety of different stats based on the test case. For test cases that benefit from strided patterns such as *compression* and *FFT*, the D-Cache prefetcher proved beneficial and maintained a high hit rate of >50% while reducing the average D-Cache DFP read time to <10 cycles. However, the prefetcher struggled with more complex memory patterns, such as those shown in *aes_sha*, which came up with a ~5% hit rate, and slowed D-Cache DFP read time to >20 cycles at times.

Lastly, due to the complex nature of the memory controller design created, it had to be iteratively designed over the course of the advanced feature stage of this MP. These iterative designs attempted to improve upon initial issues with timing and area that the structure initially had which caused failing timing pretty badly at first and being very area (>50k) and power (~25% of total CPU) intensive. In the end, we managed to get it to take up a respective area (~20k) with minimal timing costs, and netting overall improvement in memory throughput, which provided a decent bump in IPC.

2.3.3 Issue Queues

2.3.3.1 Design Overview

We wanted to iterate on the base design incorporated for our reservation station with a new design that provides a higher issue throughput. Since the CP3 base design only had one giant reservation station containing every stored instruction, it was only able to issue once per cycle. We decided it would be best to redesign the reservation stations to allow for different issue queues that can issue to their respective execution units. Judging by the benchmark analysis, it seemed ideal to have queues dedicated to only ALU and LS instructions as those are among the most common instruction types, with branches, multiplies, and divides in a separate queue. It also seemed optimal to design the new issue queues in a way that represents an actual queue structure with a FIFO format, so that old instructions don't stay stuck inside the reservation stations for too long. Other small optimizations were made that tangentially improved performance for issuing, specifically allowing for two CDB broadcasts at the same time.

2.3.3.2 Issue Shift Queue (Age Ordered Issue)

A new module called the issue shift queue was created to hold the new queue logic for these updated reservation stations. Since reservation stations can't act as traditional

head-tail queues as they can have holes in them, we thought it would be best to design this queue as a collapsing queue that shifts entries up each cycle. The queue checks for any entries that are getting dequeued or invalidated (including EBR) and appropriately marks each entry with a state that corresponds to if that entry will be free on the next cycle. If the very first entry is free, then it's able to accept new data from the dispatcher.

2.3.3.3 Performance Analysis and Trade-offs

Designing this new issue stage did net us a decent IPC bump in most test cases (~.01-.06). It also generally helped with the fine-tuning of our processor around the end as we were able to create performance counters that tracked how busy each queue was, which let us better understand where our existing bottlenecks are. For example, our ALU queue often filled up even though it was a fairly high size (~6-8), which makes sense considering most instructions are ALU and they might be waiting on data from a longer operation such as a load which can take many cycles. Even though many of these optimizations weren't tied to a specific advanced feature, it was important to take the time to improve upon them as we realized how important it was to expand upon our existing design based on where it was lacking. These small improvements allowed for good IPC gains that otherwise might've limited future advanced features.

Regarding area and power metrics, we thought that the nature of a constantly shifting queue would eat up a good amount of power and area, but that didn't turn out to be the case. The rs_issue module which encapsulates every issue queue including issue logic only took up ~10% of our total power which seemed reasonable concerning how critical the issue stage is to the overall structure of an OoO processor.

2.3.4 All LSQ Advanced Features and Post Commit Store Buffer

2.3.4.1 Design Overview

This advanced design introduces several advanced features regarding the load store queue to optimize the performance of load-store handling in out-of-order execution processors. First, a separate load queue and store queue is used instead of an unified load-store queue. Altering the structure of the load queue from a queue into a reservation station like structure also allows loads to execute out of order with respect to stores. This modification addresses scenarios where consecutive loads are present, and one load's source register has a long dependency. Second, the design incorporates load bypassing, where loads without store dependencies can be sent directly to data cache, and load forwarding, where loads with full or partial dependencies receive forwarded data immediately. This enhancement is particularly effective in workloads with a high density of loads interspersed with stores as now the execution of loads do not have to wait for the commitment of former stores. Third, the store queue is reconstructed to function as both a store queue and a post-commit store buffer. Stores can commit immediately when at the top of the ROB, with an "issue" bit marking committed instructions. These stores cannot be flushed out and are sent to the cache with lower priority than loads, enabling background execution of stores when no other memory instructions are active. This drastically reduces the delay we have on the system as now we further decrease the stress of data cache.

2.3.4.2 Performance Analysis and Trade-offs

The proposed design demonstrates measurable performance improvements, with an average decrease of 10 percent across all test cases regarding the time it takes to finish program execution. Moreover, in workloads with more load and store instructions such as

aes_sha, delay can be further decreased for up to 20 percent. However, these performance benefits come with certain trade-offs. The increased architectural complexity led to higher area and power consumption. Additionally, the intricate combinational logic required for the design could be the critical path, decreasing our maximum reachable clock frequency. These factors may cause the overall score of our processor to decrease in cases where limited amounts of memory instructions are present in the test cases.

2.3.5 Branch Predictor

A branch predictor is a critical component for attaining low delays on a CPU. Deeper pipelines motivated by the need to shorten critical path rely on accurate branch prediction to ensure high throughput of instructions as any flush invokes a penalty in cycles related to the depth of the pipeline.

2.3.5.1 Design Overview

The branch predictor on our final processor is based on the “Next-Line Predictor” from the Berkeley Out of Order Machine (BOOM) Docs. The design is a fully-associative branch target buffer (BTB) mixed with a counter table for determining branch direction with 16 entries. The counter table is indexed into using a portion of the PC address. Based on if the branch is resolved to be taken or not taken, it updates the counter in the table which informs a state of strongly taken, weakly taken, weakly not taken, or strongly not taken for future branch prediction.

2.3.5.2 Performance Analysis and Trade-offs

Fully-associative was a bad design. The preference was solely based on the fact that BOOM did it this way. We did not have time at the end to properly size the number of entries for optimal score, but even 64 entries on a fully-associative BTB causes the fetch module to use 20 mW of power. Compare this with 16 entries only being around 2.3 mW.

This is especially bad considering 64 entries has only a 2% better hit rate compared to 16 entries. If I had more time I would've changed to a set-associative design for bigger size with far less power. Another oversight is that JAL instructions do not calculate their target before being sent by the dispatcher so they always mispredict unless they have their target provided by the BTB. Still the branch mispredict rate on coremark dropped to about 20% from about 50% without the branch predictor.

2.3.6 Design space exploration

Design space exploration is a broad term referring to automation of testing different design parameters to achieve optimal performance. In our case, we implemented a version of it using python which ran the provided makefiles for sim, synth, and power and could return the results back to the program for further processing.

2.3.6.1 Design Overview

For design space exploration, we developed a robust python API for scripting purposes. The final version of the API has streamlined python functions for running sim, synth, and power which upon the makefile completing, would return all relevant data from the generated logs as return values back to the program. Additionally, adjusting parameters and timing options could be done in a single function call. The point was to eventually enable the development of the ultimate DSE script but this was never fully realized before the deadline. However, the API by itself is still a powerful tool and was used to make a handful of simpler scripts including a script which programmatically can adjust clock and run synths to find the lowest clock interval which passes timing, and also a scoring script which could queue up multiple runs for sim, synth and power to calculate scores for various parameters and clock intervals.

2.3.6.2 Performance Analysis and Trade-offs

The amount of work that went into developing the design space exploration API did not pay off in increases to CPU scores. It's possible we could have utilized it more if the competition was a bit longer, or if we gave ourselves a more conservative deadline for cutting off new advanced feature development so that we could focus on optimizing the existing design. But what ended up happening was we were debugging advanced features up until the final hours before the deadline and the API we made was never fully utilized.

2.3.7 Benchmark analysis

Benchmark analysis broadly refers to analysis of the benchmarks that our CPU was being tested on. This analysis should be agnostic of our CPU design. While there was definitely more advanced analysis that could've been done, we felt that the level of our analysis was appropriate for what we needed to answer design-related questions that we had.

2.3.7.1 Design Overview

For benchmark analysis, we created a python script which analyzed the spike logs of all the given testbenches to determine instruction makeup. The final report we generated is below:

Program	Total Inst	ALU	Branch	Load	Store	Multiply /Divide
coremark_im.elf	304044	162703 (53.51%)	60475 (19.89%)	55398 (18.22%)	15976 (5.25%)	9492 (3.12%)
aes_sha.elf	652971	373398 (57.18%)	8135 (1.25%)	181197 (27.75%)	90241 (13.82%)	0 (0.00%)
fft.elf	520196	343005 (65.94%)	19963 (3.84%)	69782 (13.41%)	77206 (14.84%)	10240 (1.97%)
mergesort.elf	478157	262664 (54.93%)	89106 (18.64%)	71118 (14.87%)	55269 (11.56%)	0 (0.00%)
compression_im.elf	425078	280841 (66.07%)	55476 (13.05%)	46252 (10.88%)	42509 (10.00%)	0 (0.00%)

2.3.7.2 Performance Analysis and Trade-offs

Seeing just how few DIV and MULT instructions there were in the benchmark analysis directly motivated our decision to part ways with the div_pipeline IP and switch to div_sequential. We had had problems up to that point with div_pipeline becoming the critical path despite having as high as twenty stages and this report made us realize that using div_pipeline was complete overkill for the public testbenches.

3. Additional Observations

One prominent issue that came up quite late within our project design stage was the consideration of area and power consumption. Much of our early work within this project had been focused on getting an implementation to work regardless of timing concerns since there was no restriction for timing unlike other MPs. While we did have an easier time designing the processor based on these lax rules, we started to see the rough edges of it when our competition runs started to come out. We didn't take into account how VCS synthesizes to meet timing requirements and found out that it does so in a way that might

be *too* intelligent. We did manage to get timing to meet at high frequencies but at the cost of our power skyrocketing to >500mW in some of our earliest competition runs.

We managed to cut down on a lot of our excess area and power issues by the time the final competition run was due, resulting in a decent final score for the competition, but there was certainly still more work to be done in this respect. The importance of design has come up several times throughout ECE 411 and it was definitely shown here, especially as this was such a large collaborative effort. However, learning from these errors was greatly insightful for us and it taught us a significant amount about effective hardware design and project management as a whole.

4. Conclusion

This project successfully demonstrates the design and implementation of an out-of-order RISC-V CPU with explicit register renaming and several advanced features aimed at improving performance and efficiency. By enabling out-of-order execution, the processor optimizes instruction throughput, leveraging parallelism to maximize resource utilization. The integration of explicit register renaming effectively addresses register dependencies, ensuring smooth instruction flow and reducing delays. Additionally, the inclusion of features such as early branch recovery, prefetching, and a split load-store queue further enhances the processor's ability to handle complex workloads. Through the completion of this project, valuable insights were gained into the intricacies of modern processor architecture, showcasing the trade-offs between complexity, performance, and energy efficiency in high-speed computational systems. However, there were many challenges along the way, such as encountering difficulties with some of our initial design including but not limited to excess area and power consumption. Despite these obstacles, this hands-on experience has reinforced a deeper understanding of the fundamental

principles behind cutting-edge processor design and the challenges inherent in pushing the boundaries of computational performance.