Team Oreos: mp_ooo Final Report

Date: December 10, 2024
Class: ECE 411: Computer Organization and Design
Submitted By: Cameron Marchese, Eshaan Tibrewala, Raahul Rajah

Introduction

This report presents the design and implementation of an out-of-order (OoO) processor. This project implements a processor that adheres to the RV32IM instruction set while demonstrating performance optimizations through advanced architectural features. The processor uses explicit register renaming (ERR) to manage data dependencies, minimize pipeline stalls, and execute instructions out of order.

The RV32IM instruction set is part of the RISC-V architecture, an open and extensible instruction set standard. It defines a set of 32-bit instructions supporting integer arithmetic (RV32I) and multiplication/division (M), making it suitable for various general-purpose computing tasks. The RISC-V architecture follows a Reduced Instruction Set Computing (RISC) approach, prioritizing simplicity and efficiency over the more complex instruction sets of CISC (Complex Instruction Set Computing) architectures like x86. Development and debugging for this project utilized tools such as Verdi for waveform analysis and VCS for simulation. Testing was conducted using test cases that verified functionality, performance, and edge-case handling across the processor's pipeline and architectural features.

Key advanced features incorporated into the design include split load-store queues (LSQs) for improved memory operation handling, a Gselect branch predictor for enhanced control flow accuracy, and a next-line prefetcher to reduce cache latency. Additionally, advanced analyses were conducted to evaluate benchmarks, explore design choices, and visualize processor performance.

This report is organized into sections detailing the project overview, design descriptions with milestones, advanced features, and additional observations. By integrating these advanced features and applying thorough performance analysis, the project demonstrates a comprehensive approach to out-of-order processor design.

Project Overview

Our project is an OoO processor capable of executing RISC-V RV32IM instructions out of order so that they are executed when ready instead of following a strict execution order. This exploits instruction level parallelism (ILP) because it can immediately execute instructions without dependencies instead of waiting for the earlier instruction to finish. Building a fully functional OoO processor involved much planning and design. Many design decisions were considered and tested, and each aspect of the processor had to be mapped out by hand before being implemented in code. We created a processor that carried out the fundamental functionalities of an OoO processor and equipped our processor with advanced features that would create noticeable improvements in overall performance.

As a group, we all had a strong conceptual understanding of the processor we were designing and were on the same page about how the final product should look. This shared understanding allowed us to function effectively and stay on track with our checkpoint deadlines. We did not have a formal organizational or administrative structure. Instead of delegating major tasks or modules to individual members, we

collectively worked on each aspect of the processor. This method worked well for us because it encouraged constant communication and collaboration, which helped us individually understand all aspects of the processor and catch bugs and errors much faster. All code was shared through Github, and all other work, such as diagrams and designs, were shared through Discord.

Design Description

Overview

This project implements an OoO processor developed through checkpoints. In CP1, we built the front-end instruction fetch stage, integrating a 4KB 4-way set associative cache with a 3-stage cache line adapter and a configurable instruction queue. CP2 expanded functionality by implementing decode, dispatch, and execution stages, adding ALU operations, multiplication, division, and LUI. CP3 addressed branch instructions (auipc, jal, and jalr) and implemented a static "branch not taken" predictor.

Advanced features include a next-line prefetcher to reduce instruction cache miss rates and a split load-store queue to improve memory efficiency. A Gselect branch predictor, using local-global history and a pattern history table, enhanced branch prediction accuracy. Performance improvements were guided by GEM5 simulations, which validated optimizations. A Python-based design space exploration automated testing and configuration. Each of these features is discussed below in further detail.

Checkpoints

CP1

During the first checkpoint of our OoO processor, we focused on developing the front-end instruction fetch stage and integrating it with the DRAM module. The memory module we worked with was a banked burst memory module, which supported multiple outstanding requests. The first step in designing the front-end fetch was to develop a cache line adapter that could translate memory bursts into a cache line for our 4-way set associative 4KB cache where each cache line was 256 bits. We already had a functional 2-stage pipelined cache from mp_cache, so the cache itself did not have to be separately implemented. The final piece of the front-end fetch stage was the instruction queue that would access the cache for the RISC-V instructions and enqueue them to a parameterized queue that would dequeue instructions into the rest of the OoO stages.

For checkpoint 1, the cache line adapter was implemented as a simple three-state pipeline. Since we were only accessing instructions and interfacing with a singular instruction cache, we did not consider writing to memory or designing an arbiter that selected between an instruction and data cache. Our initial cache line adapter defaults to the WAIT stage, where it stays until the cache triggers a read miss when the instruction queue attempts to enqueue an instruction because it is not full. The PC value is assigned to

bmem_addr, and bmem_read is set high, triggering a memory read request. The cache line adapter FSM transitions to the BURSTING stage, where it stays for at least 4 cycles while DRAM responds with instruction data. The FSM stays in BURSTING for 4 cycles minimum because the memory data response size is 64 bits. Since the cache line is 256 bits, the cache line adapter puts each consecutive response into a 256-bit buffer until the buffer is full, then sends the buffer back to the cache, where the cache continues with its read miss protocol as normal. Cache line adapter functionality was verified by manually issuing enqueue requests from the instruction queue, monitoring the cache to see if dfp_read (read miss) was triggered, and then waiting for a dfp_resp to see if the cache line was written to cache with a valid 256-bit value or 8 RISC-V consecutive RISC-V instructions.

The cache itself did not have to be modified for this checkpoint. The pipelined nature of the cache allowed us to reduce access latency and increase throughput. The 4-way set associativity of the cache also provided performance benefits because it improved overall cache utilization and reduced conflict misses. We verified the cache's functionality by monitoring the cache lines after each instruction fetch to see if the data was being put into the cache properly. Then, we triggered accurate cache hits when the instruction queue requested an instruction from an existing memory address in the cache.

The final module of the front-end fetch design was the instruction queue. We decided to make the width and depth parameterizable to find the optimal configuration as our processor evolved. We implemented the queue as a circular queue with a head and tail pointer, where instructions were enqueued at the tail and dequeued from the head. The challenging part of this implementation was figuring out how to determine if the queue was full or empty because simply checking if the head was equal to the tail was not enough. We solved this problem by adding a bit to the pointers. The queue was marked as empty if head==tail and the most significant bits were equal. If the most significant bits were opposites, then the queue was marked as full. The goal was to keep the queue full at all times so that instructions could flow through our processor efficiently. Therefore, the enqueue_flag stayed high as long as is_full_flag stayed low so the CPU could keep fetching instructions from the cache. Queue functionality was verified by manually driving the enqueue and dequeue signals from an external testbench and checking if instructions were flowing through the queue as expected. We also extensively tested the specific edge cases where the queue was full and enqueue was attempted and when the queue was empty and dequeue was attempted.

CP2

The second checkpoint successfully implemented all immediate and register instructions including all ALU operations, multiplication, division, and remainder, and load upper immediate (LUI) instructions. We extensively used our block diagram from CP1 to guide our implementation. We decided to implement each module separately and unit test them individually to verify the functionality before moving on to the next module. We started with decode and dispatch to translate the instruction data into data that our processor could use to execute the instruction. Then, we moved to implementing the free list, retirement register file (RRF), and register alias table (RAT) to collect all the necessary information to create reservation station entries. Then, we implemented the reorder buffer (ROB), the last module that directly interfaced with our dispatch module, and, therefore, it was necessary to verify the full functionality of dispatch. We then moved down the execution steps and implemented the reservation stations, physical

register file, and functional units. We concluded checkpoint 2 by connecting all the modules by implementing the common data bus (CDB).

Decode

Most of the logic for the decode module was the same as the pipelined processor we had created earlier this semester. The most important aspect of the module was creating a struct that organized all the important metadata from the 32-bit instruction that told the processor exactly how to execute the incoming instruction. Verification for this module was very simple since we just had to ensure that the struct retained all the necessary data when it was sent to dispatch so that the dispatch module could properly distribute the instruction to the necessary modules.

Dispatch

The next module we implemented was dispatch. The goal of this module was to dequeue an available register from the free list, access the RAT to check for physical and architectural register mappings, create a ROB entry, and create a reservation station entry. Another important functionality of the dispatch module was driving the stall_dispatch signal, which stopped instructions from getting dequeued from the instruction queue when the signal was high.

We determined three independent conditions for this stall signal that should ideally cause our processor to stop executing new instructions. The first was the ROB full flag because the ROB keeps track of all the instructions that are currently in the processor and when they commit, so if that queue is full, our processor should not read new instructions. Another constraint we identified was an empty. If the free list is empty, no physical registers are available for new instructions coming into the processor, so we want to stall in that case. The third constraint we identified was when the reservation station was full. For the reservation stations, we decided to implement separate reservation stations for each functional unit, so if any of the reservation stations were full, the reservation station full flag would go high in dispatch, stalling all new instructions. Since out decode and dispatch happen combinationally, once the instruction is dequeued, it has to be enqueued into a reservation station. Therefore, if the stations are full, the instruction has nowhere to go when it is dequeued and should be stalled. We verified the stall logic by tracing the flag to its source every time it went high to ensure that the corresponding queue was actually full at that time.

We decided to make all the reservation stations' entries the same even though we have different reservation stations for each type of functional unit. We made this decision because the consistency would make debugging and data management easier as we implemented additional features.

Verifying the dispatch module fully was difficult because we had to wait until the ROB, free list, and RAT were fully implemented since most of the data for the ROB entries and reservation station entries came from these modules. However, once we successfully implemented and verified the remaining modules, we confirmed functionality by monitoring the incoming instructions, checking if the structs were populated correctly, and then writing to their respective queues on the next cycle.

Free List

After our initial dispatch implementation, we moved on to the free list. Our initial design of the free list was a circular queue where the length was the *number of physical registers - number of architectural registers*. Each index had a physical register, and when the register was dequeued, we wrote a zero to that index. Since we had already implemented a circular queue template for our instruction queue, we used a slightly modified version for our initial free list. The verification processor was similar, where we tested the most common and edge cases by driving enqueue and dequeue signals from the connected modules.

RRF

We also implemented the RRF along with the free list and the RAT. Since the RRF would enqueue available free registers back to the free list, we decided it was easier to implement and test the modules together. The RRF was implemented as a basic array of length 32 (total number of architectural registers), where each index had the value of the physical register it was mapped to. When a mapping was ready to be updated, the RRF enable signal would go high, and the module would sequentially update the architectural register mapping. On the same cycle, it received the enable signal, the free list enqueue flag would go high, and sequentially, the now freed physical register would be written back to the free list. We verified our implementation of the RRF by checking if enable goes high when an instruction commits from the head of the ROB and then if the RRF array and free list are updated accordingly on the next cycle.

RAT

We implemented our RAT using logic similar to the RRF. However, our biggest challenge with the RAT was understanding how to initialize the array. After considering different implementations, we concluded that the most efficient design was a small struct with a valid bit and a corresponding physical register. So when the processor is initialized, all the valid bits are set to high, and the physical registers 0 - 31 are mapped to the same architectural registers, represented by the array indexes. The remaining physical registers are added to the free list. The RAT is modified as the mappings change as instructions flow through the processor. The array is updated sequentially, and all reads are combinational so that the dispatch module can get the mappings for whatever source registers are needed by the instruction. The module also outputs whether or not the mapping is currently valid, which will be used once the instruction arrives in the reservation station to check whether or not the instruction is ready to be executed in the corresponding functional unit.

ROB

We once again implemented a circular queue for our ROB implementation and modified our queue template to serve the specific ROB functionalities. Like our other queues, our ROB was designed to be parameterized so that we could determine the ideal ROB size for the best processor performance. The ROB struct consisted of the ready_commit signal, the architectural destination register, and the physical destination register. Per our initial block diagram, ROB entries would be populated with data through the dispatch module and then enqueued into the ROB queue on the next cycle, where the instruction would be

executed. Our initial verification of the ROB ensured that the queue functioned as expected and the correct ROB entries were populated into the queue based on the current instruction in dispatch. Once we completed the CDB, we had to conduct further verification on the ROB to ensure that instructions were committing to their corresponding ROB index in the queue and that the ROB was then committing these instructions correctly. We then traced the physical and architectural register mappings through the RRF and free list to confirm that the processor behaved as expected post-commit.

Reservation Stations

For the reservation station, we decided to have separator reservation stations for each functional unit type. Since the goal of this checkpoint was to implement all register and immediate instructions, we designed three reservation station/functional unit pairs. We had one set for ALU operations, one for multiplication operations, and one for division and remainder operations.

We designed a general reservation station template and added a port where the CPU could assign a funct_unit_key signal, which would tell the reservation station which instruction type it holds. We made this design decision because we could generalize all of our reservation stations, making debugging and verification much easier. This design also made it significantly easier to add additional functional units and reservation station pairs in checkpoint 3 when we implemented control and memory instructions.

Our reservation stations were structured as a parameterizable array where each entry was a struct consisting of information that would be used by the physical register file, functional unit, and CDB and ROB once it exited the reservations station. To update our reservation station entries with any dependencies on the CDB, we would iterate through all the reservation stations and write the new values to any dependent entries. To dequeue from the reservation station, the corresponding functional unit would raise the dequeue signal, and the reservation station would poll its entries to see if any of the structs had the valid bit set to 1. If the entry is marked as valid, it is ready to be executed, and the processor sends it to the functional unit.

Verification of the reservation station module was challenging due to all the dependencies we had to handle coming in from the CDB. However, we unit-tested each aspect independently before putting it all together. We first started by checking that the correct instruction was coming into the correct reservation station with accurate data. Then, we tested if the CDB could properly poll the valid entries in the reservation station and find and update the dependencies.

Physical Register File

The structure of our physical register file was very similar to the register file we used from mp_pipeline. However, because we had multiple reservation stations and functional units, we added separate ports for each pair so that they could independently interact with the register file. The register file itself has one array whose length is equal to the number of physical registers. All writes to the register file are sequential, and reads are combinational so that physical registers can send the data to the functional units on the same cycle they leave the reservation station. The challenging part of this module was to independently handle read dependencies for all the separate instruction types. We handled this issue by

individually writing read logic for each instruction type. We also directly wired the immediate value through the register file to maintain consistency and keep the reservation station data together. Since we already had a register file template from before, verification of the physical register file did not take long.

Functional Units

For CP2, we designed 3 separate functional unit modules for each corresponding reservation station unit. For multiplication and division/remainder, we utilized the pipelined Synopsys IPs. These IPs enabled us to use verified modules that could be parameterized for efficient performance with our processor. Since the IPs were already verified, we did not have to verify the calculations. Still, we did have to ensure that the other information in the functional units was synchronized. We handled this case by creating a counter variable, which essentially latches important information in the functional unit outside of the IP while it calculates the result of the instruction. Once the result is ready, our functional unit regroups the result with other important instruction information. It sends it as a struct to the CDB adapter, where it is broadcasted to the rest of the processor.

However, there was no IP for the ALU unit, so we had to design our own. The ALU needed to handle all shift operations, addition and subtraction, and logical operations for both signed and unsigned values. We utilized the ALU template from mp_pipeline and modified that ALU to work effectively with our OoO processor. We determined that ALU operations made up most of almost any program run on our processor, so the functional unit should be ready quickly so that the corresponding reservation station doesn't stall often. As a result, ALU operations only take 1 cycle to produce the result and has priority over multiplication and division/remainder instructions in the CDB adapter. To verify the ALU module, we tested all the different operations separately to ensure that the module correctly calculated the results.

Common Data Bus

The CDB was the most challenging part of CP2 because we had to broadcast the outputs of the functional units to multiple modules in our processor and design an adapter that efficiently chose which functional unit outputs would be on the bus at any given time.

We designed the adapter to choose functional unit outputs based on a priority scheme where ALU operations had the highest priority, followed by multiplication, division, and remainder instructions. We chose this priority for the scope of CP2 because ALU instructions made up the largest portion of the program, so they were most likely to stall if not handled immediately. To verify our adapter's functionality, we tested all different instruction types to see how the adapter was cycling through the functional unit and if all the outputs were eventually handled as expected.

After verifying the adapter, we connected the CDB to the ROB, physical register file, reservation stations, and RAT. We created a CDB struct that included the register information, data, and ROB index to be committed. The corresponding modules are all updated accordingly if dependencies are detected, and the ROB struct at the index is set to commit. Since the CDB was implemented in CP2, we verified it by putting all the modules together and testing random instructions. Since we had already tested each module

independently, most bugs were related to the CDB. This verification method helped us debug the processor as a whole and fix the issues from our initial implementation of the CDB.

CP3

In CP3, we integrated instruction and data caches into our processor, implemented all memory operations for both loads and stores and all control instructions (BR, JAL, JALR). To integrate the caches, we made another module instantiation of our 4-way set associative cache for data since we already had the instruction cache implemented from CP1. We also had to design a cache arbiter so that our CPU could efficiently switch between interfacing with the instruction and data caches. For memory and control instructions, we added two more reservation station and functional unit pairs, one each for memory and control. This allowed us to efficiently integrate the remaining instructions into our existing processor design from CP2. The overall verification process was tedious because we encountered many edge cases with memory operations that we had not considered when we originally mapped out our logic.

Cache Arbiter

We implemented our cache arbiter as a two-state FSM that switched between the INSTRUCTION and DATA states. The default state was INSTRUCTION because we always want to be fetching instructions from the instruction cache when the queue is not full, but if the processor is executing a memory operation, the state switches to DATA. Our cache line adapter switches signals so that it is interfacing with the data cache instead. Since our FSM only had two states, the implementation did not take long. For verification, we constantly switched the states between INSTRUCTION and DATA to see if the corresponding read and write signals were switching as expected. This also helped us accurately drive the corresponding masks to both caches so that data was not lost or sent to the wrong cache.

Memory Instructions

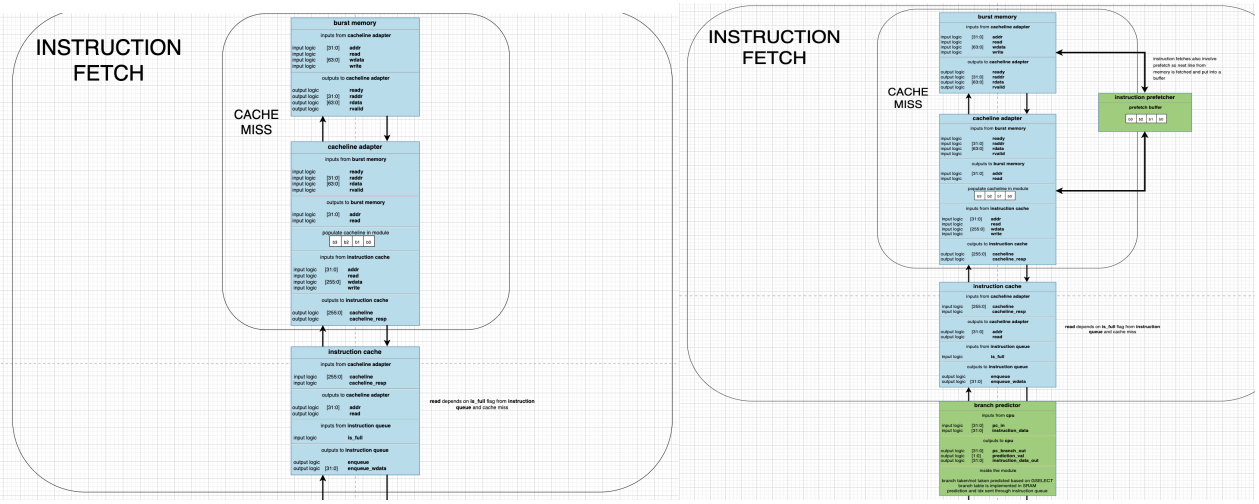
For memory operations, we had to consider both loads and stores. We created one functional unit for all memory operations, so loads and stores go to the same module after leaving the reservation stations. Our memory instructions functional unit design included a separate queue in the unit, where instructions would leave the reservation station and get queued on the circular queue. The target memory addresses and other necessary information would be calculated separately in the functional unit outside the queue. Once this address is determined, the load or store instruction will be updated in the queue and marked ready so that the functional unit knows the instruction is ready to execute the memory operation. Once the instruction is ready to execute, the functional unit reads the instruction at the head of the queue and sets the corresponding mask, memory address, and write data depending on the instruction type. Once the cache responds, the instruction is dequeued from the load store queue and travels down the functional unit until it's ready to commit. One edge case for store instructions we had to factor in was on flushes, recovering data from memory is very difficult, so stores should not be executed until the processor knows that the instruction is valid. We solved this problem by holding the store instruction even though it was ready until the corresponding ROB index reached the head of its queue so that the processor knew it was ready to commit. So once the cache responds and the store has been successfully executed, it is directly

sent to ROB and not through CDB to be committed. Loads function as normal and are sent to ROB and the rest of the processor through the CDB. Since load and store instructions make up the smallest percentage of most programs, memory instructions are given the lowest priority in the CDB adapter. To verify functionality, we created a testbench that was exclusively load and store instructions to see if they went through the processor correctly and committed in order.

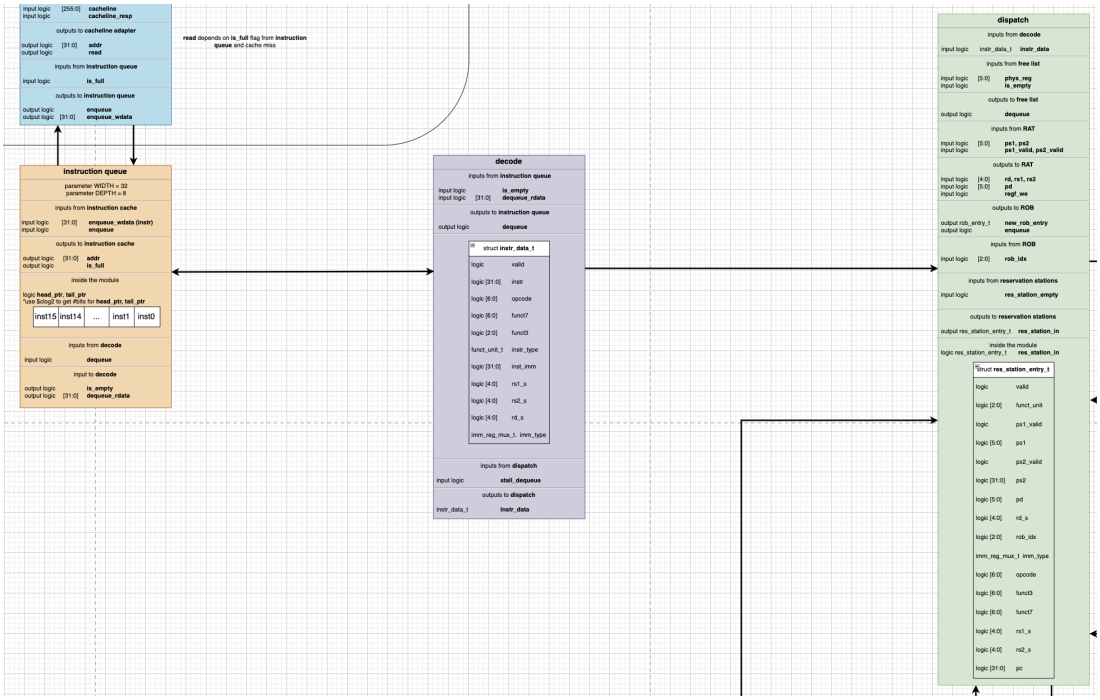
Control Instructions

The functional unit for control instructions was similar to the functional unit for memory operations. We implemented a control instruction queue where instructions were enqueued after leaving the reservation station. The target memory address is calculated separately in the functional unit, and then the instruction is updated in the control queue. When the instruction reaches the head of the queue, the functional unit checks if the PC address is updated and then dequeues the instruction to execute it. For CP3, we used a static, not taken branch predictor scheme. So until branch instructions were determined to be taken, our processor functioned as normal. Once branch instructions are dequeued from the control queue, they are sent directly to the ROB, where the processor flushes all current instructions and dequeues from the new PC if the branch is taken or does nothing and commits. JAL and JALR instructions, however, do have destination registers, so they are sent to the ROB through the CDB so that it can update all the dependencies. We verified the functionality of control instructions by creating a test case that tested different kinds of branches and then separately tested jump instructions. The most challenging part of verification was fixing the bugs with ROB flushing since that signal changed the entire state of our processor, so it was connected to all the other modules in our processor.

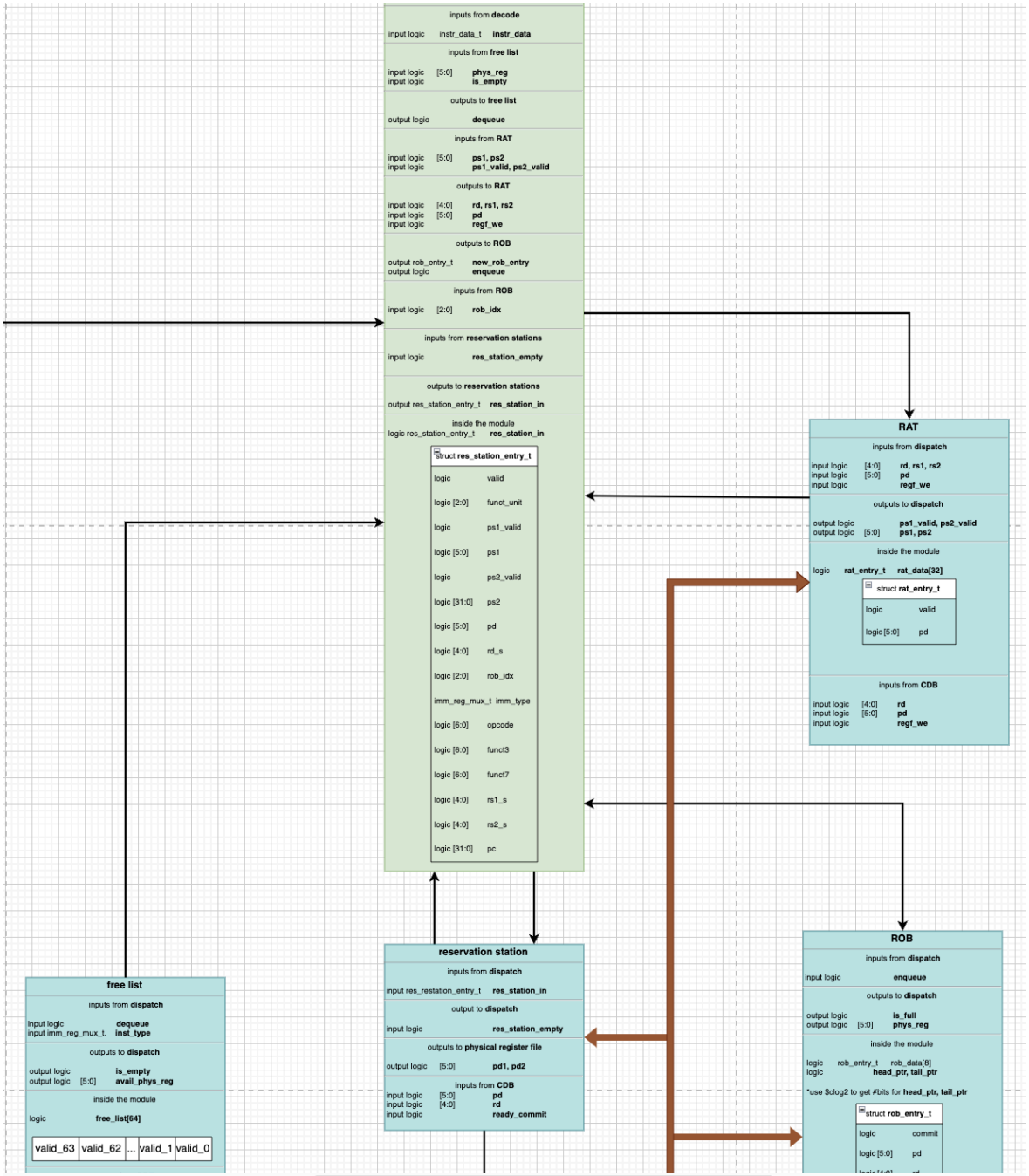
Block Diagrams



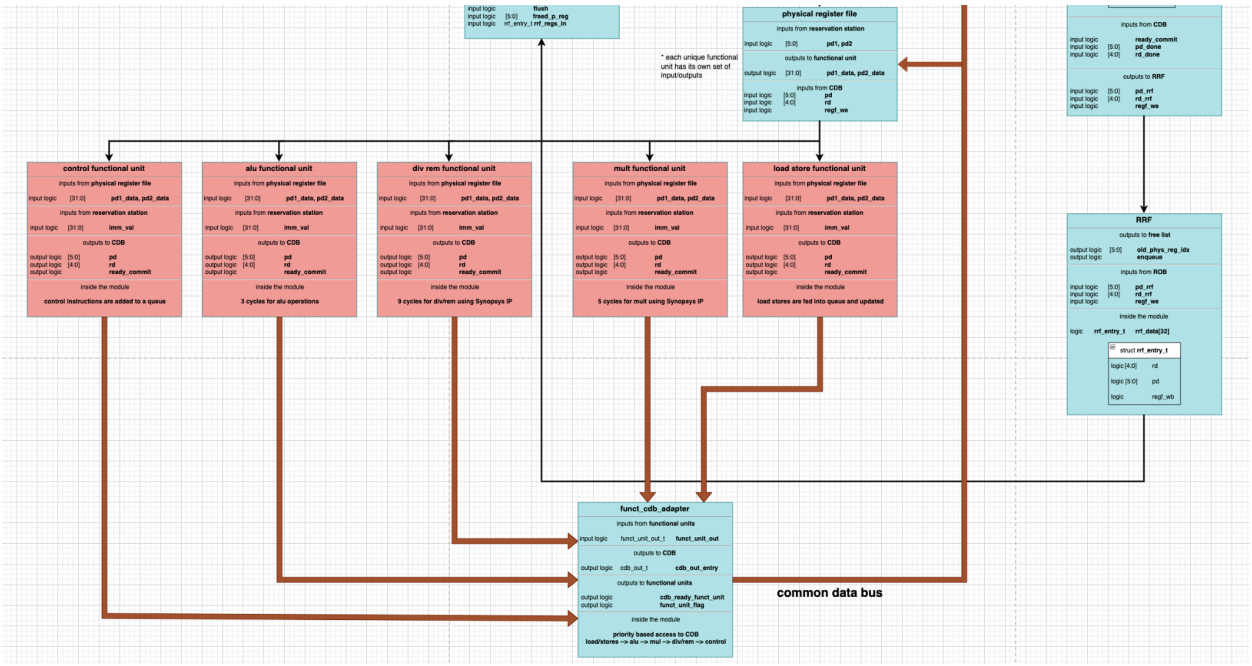
Post-CP3 Front-End Diagram (Left) and Post-Advanced Features Front-End Diagram (Right) (Figure 1)



Instruction Queue Through Dispatch Combinational Path (Figure 2)



Decode, Free List, Reservation Station, RAT, and ROB (Figure 3)



Physical Register File, Functional Units, CDB Adapter, RRF (Figure 4)

Advanced Features - Design Options

Gselect Branch Predictor

Design

When we first considered what advanced features we wanted to implement, we first considered implementing a branch predictor. Flushing on a branch misprediction is costly and takes many cycles, so ideally, we want to avoid as many mispredictions as possible. We settled on implementing a Gselect Branch Predictor because it is simple and does not use too many resources. Still, it is also very effective (about 92% for an optimal implementation without BTB, according to the data we found).

To implement this, we changed our PC incrementing logic first so that the branch prediction could be made first. We first send our PC to the instruction cache to retrieve our instruction data. Previously, we used to enqueue the data we had into the instruction queue immediately after receiving the response. We still input the data into the branch predictor module. This module first checks if the instruction is a branch instruction, and if it is not, it increments PC normally and enqueues the instruction onto the queue. However, if it is a branch instruction, a prediction on whether the branch should be taken or not taken is made, and the PC is incremented accordingly before enqueueing onto the instruction queue.

To construct our Gselect branch predictor, we used a local-global history register and a pattern history table initiated in SRAM that is indexable by 8 bits. The global history table holds the previous four outcomes of whether the branch was taken, where 1 indicates that it was taken and 0 indicates that it was not. This is a shift register which consists of 4 bits where the new outcome is shifted in from the right. Both the global history register and bits from the PC are used to index the pattern history table. The bottom 2 bits of the PC are always 0, so we cannot use those bits to calculate the index because they are not unique. Instead, we used Bits 5 through 2 ([5:2]) to get 4 bits from the PC. The 4-bit global history register is then concatenated with the 4 bits from the PC to get the full 8-bit index into the pattern history table. The pattern history table holds the prediction for the branch instruction where it can be one of many 2-bit values: 00, 01, 10, 11. 00 represents strongly not taken, 01 is weakly not taken, 10 is weakly taken, and 11 is strongly taken. If the branch was taken, this counter goes up; if it was not, the counter goes down. Our pattern history table is a dual port module so that it can be written to and read from simultaneously, and it is initialized so that all entries are weakly not taken.

On a branch instruction, the instruction, the prediction value, and the index used to access the pattern history table are saved onto the instruction queue. Then, after dequeuing the instruction, it is entered into the reservation station as usual in dispatch, and the pattern history table index as well as the prediction value are saved into ROB simultaneously. Inside the functional unit for the control instruction, if there is a branch instruction, its pattern history table index and prediction value are read in from the rob. The correct PC write data values, branch enable, and the new counter value for the pattern history table are calculated. The index, the updated counter value, and the branch outcome are then entered into the branch predictor module, which is then written into the global history register and the pattern history table. Finally, when the instruction is done and committed to the ROB, we check the previous prediction value and what branch enables it and determine if we need to flush the pipeline.

Testing

Our first step was to change the PC incrementation logic because that was all done right away when we got a response from the instruction cache. Our design has two PC counters: the PC that will be sent to request an instruction from i-cache and the other that will propagate through our pipeline when the instruction is dequeued off the instruction queue. After we changed the logic to change the PC being sent to the instruction cache based on prediction, our other PC counter was not in line with the instruction being dequeued off the instruction queue, causing issues when testing our branch predictor. This first bug was solved by making it so the PC associated with the instruction would be enqueued onto the instruction queue simultaneously with the instruction and then dequeued. This way, there were no timing issues.

After fixing this issue, we realized our branch predictor was not working because we forgot to account for the fact that SRAM takes one cycle to read, and it was not combinational, so none of our predictions were right. Coincidently, we also noticed that our prediction values were all don't cares when being accessed for the first time, so that was also throwing issues for our processor when trying to use don't care values. We first made it so the processor would not send a request to the instruction cache on the next cycle if a branch instruction was found, essentially stalling for one cycle. However, this would hurt the processor by stalling every time there is a branch prediction, but we believe that this is far better than having many mispredictions. After stalling for one cycle, we could have gotten our prediction value on time and used

that correctly. Next, to avoid accidentally using don’t care values we created a valid array the same size as our pattern history table and would be indexed the same. We would use this valid array to let us know whether the pattern history table has been written to that index or is being accessed for the first time. If the valid array has the value 0 at that index, then the prediction value of don't care from the SRAM would not be used but instead, the value 01, which represents weakly not taken. We also made the valid array read to take one cycle to sync up with the timing of the read from the SRAM.

After this, the branch predictor seemed to work, but we noticed that for the same indexes in the pattern history table, ROB flush was always high for those instructions. This meant the pattern history table was never updated to loops with similar branch outcomes, and we always got the wrong prediction. This was due to some wrong connection we made when connecting the module, but after fixing those, the actual values were updated, and correct predictions were made.

Performance Analysis

The best way to see if the branch predictor works is to see the number of flushes the processor made. We can see many more ROB flushes before advanced features were added. After adding the branch predictor, the number of ROB flushes went significantly down. We can also see that the branch predictor was taking a different prediction than our static not-taken branch prediction because the number of instructions for the different types of instruction, such as load, stores, or branches, etc., is drastically different, indicating that different prediction paths are being taken. This can be seen in the over 70% decrease in ROB flushes between baseline and advanced features in the graph below titled “Rob Flush” in the supporting data section of the report, which shows the branch predictor working.

Next-Line Prefetcher

Design

The next-line prefetcher fetches sequential cachelines to reduce memory latency. The prefetcher issues two back-to-back memory requests: one for the requested cacheline (A) and another for the next line (A+1). This is done by consecutively sending memory addresses bmem_addr and bmem_addr + 32 to memory (bmem). The prefetcher checks that the responses match their corresponding requests upon receiving data. Specifically, the address of the first response must match the address for line A, and the second response must match the address for line A+1.

When a subsequent memory request is made, the prefetcher first checks the buffer for a match. If the requested line matches the prefetch buffer, the data is sent directly from the buffer to the cache. If there is no match, a new memory request is issued to memory. The buffer is always updated with the latest prefetch data when a new sequential cacheline is fetched, ensuring the most recent prefetched line is available for future requests. This eliminates additional reordering buffers using the cacheline adapter's existing capacity to manage out-of-order responses. It also avoids redundant checks for data already in the cache because the cache is always accessed first before accessing the buffer or issuing a new memory request.

Testing

Setting up testing for this design was straightforward, relying on the provided testbench files for debugging. We began by analyzing the waveform from the point of the first fetch, which also triggered the initial prefetch. From there, we focused on handling a few specific cases: the initial prefetch, scenarios where a prefetched line was stored and needed to be used, and branching cases. While these cases required some time to address correctly, resolving them eliminated most other debugging challenges.

The main bug was that it initially seemed like prefetch was working, but the processor never actually utilized the prefetched data. Instead, the processor would fetch two lines with every fetch and discard the buffer's prefetch contents rather than using it when it should. Overall, all other tests functioned properly by getting one provided test to work fully and confirm the correctness of the waveform.

Performance Analysis

The next-line prefetcher primarily reduced memory stalls, improving access times by ensuring sequential data was readily available in its buffer. This allowed the processor to access prefetched data without waiting for memory responses. The prefetcher is designed to minimize its impact on cache behavior, as it bypasses the cache for prefetched data and directly serves the processor when a match occurs. Any changes in cache accesses or cache misses could reflect indirect effects from other advanced features and optimizations, but the main benefit is reduced memory delays.

There is also an increase in cache misses, but this is because cache misses are calculated as dfp_read which can also be triggered by prefetches. Therefore, this data can be misleading as this is somewhat of a cache “miss” however, it is not necessarily negatively impacting performance in the case of prefetch. It should be noted that the way this metric is calculated is accurate for the baseline statistics but needs to be investigated more carefully for the advanced features statistics depending on what is being considered a cache miss. This can be seen in the change in cache misses below in the graph titled “Cache Misses”. While misses increase, if you include prefetch accesses in this statistic as we have, the increase of about 100% is beneficial and shows prefetch working.

Split LSQ

Design

The Split LSQ consists of two FIFO queues, one containing only loads and the other containing stores. If a store at the head of the store queue has its address ready and it is a valid instruction, the store address is sent to the data cache as a request, and the data is written into memory just like before implementing the split load store queue. This only happens when the store instruction is also at the head of the ROB queue. If a load instruction at the head of the load queue has its address ready and it is a valid instruction, then a number of things happen that are different from before. First, the load address for the instruction and its corresponding ROB index value are sent to the store queue. The store queue is iterated from the tail

pointer backward to the head pointer. This ensures the queue is iterated with the most recent store instruction being found first. The distance from the head pointer of the ROB to the corresponding load and store instruction being checked on the ROB are also calculated. This helps determine if the store came before the load instruction or not. The code checks for stores that arrived before the load instruction and whether the address matches the load address to determine if the value needs to be forwarded to the load instruction.

Many scenarios can happen now. One scenario could be that a store was found with the same address before the load instruction in the queue. In this case, the value is directly sent to the load instruction, and corresponding flags go high. Another scenario could be that the store that was before the load instruction was found but does not contain its store address yet in the queue. In this case, the load instruction is stalled until the store address is calculated and determines whether there is dependency. If the store's address is not the same as the load address, then the rest of the queue is also checked to determine if any other stores have the dependency. This is calculated by counting the number of stores that could have a dependency when iterating through the queue, and if the number is greater than 1, then stall the load instruction. Finally, no stores could be found; in that case, the load address and mask will be sent to the data cache to perform the load. There is an array containing the ready status of both the instructions at the head of both queues and a for loop iterates to pick which instruction to handle first. Only loads and stores of the same type are considered dependencies in the code. For example, LW and SW are considered dependencies, but LB and SW are not. These implementation features makes it so that loads not associated with the corresponding stores in the store queue can be committed out of order, and loads with a dependency have the value from the store forwarded without the load having to send a request to the data cache.

Testing

When testing initially, the load store queue seemed to be all working. However, when examining the waveforms more closely, we could see almost no difference between the Split LSQ and the singular queue. The loads were always sending addresses to the cache. It was even worse as the loads requested values from memory even before stores were written in that memory location. This was solved by changing the logic to check the stores in the store queue, as the previous checks always failed, even if there was a possible store that could cause a dependency. New flags were added to properly check whether a valid store instruction can forward a value. The PC value was also used to check the store instruction before the load instruction, but if a branch has occurred, this might not always be the case. Instead, the distances of the instruction in the ROB from the head counter were used to determine which instruction came first more accurately.

Another issue that we found was that the distances found in the ROB for both instructions were not accurate sometimes, and this was due to a simple parenthesis missing. This was fixed, and the distances were calculated properly afterward.

Finally, there was an issue about how if a store that could have a dependency was found not to have the same address as the load, then the rest of the store queue was not being checked. The fix for this bug is discussed more thoroughly in the Major Bugs section, as this directly led us to fail the RSA test.

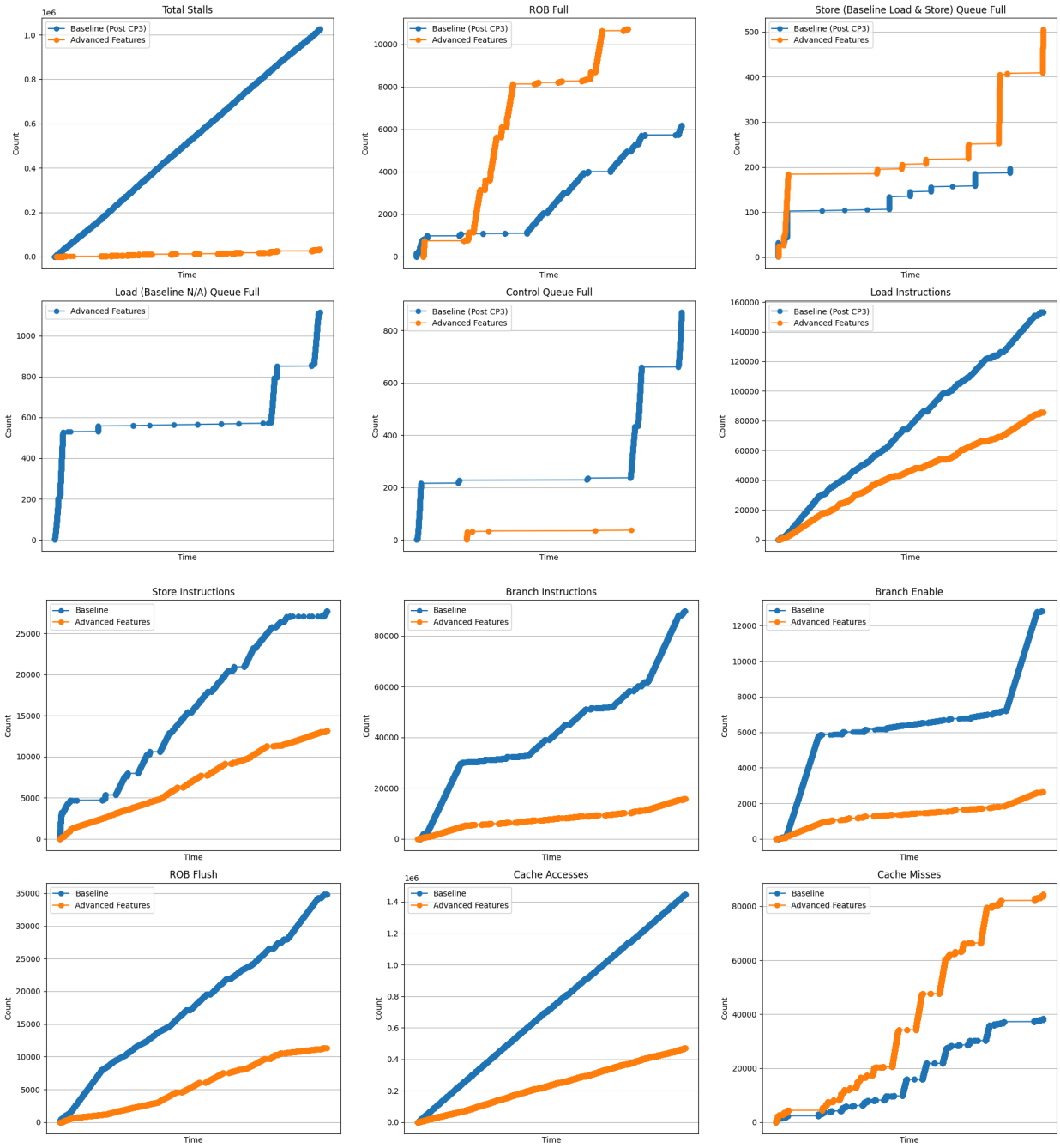
Performance Analysis

The Split LSQ improved memory operation handling by separating loads and stores into distinct queues. The main benefit is reduced stalling, particularly for loads, as dependencies are resolved more effectively through store-to-load forwarding. This ensures that loads retrieve values directly from the store queue, reducing delays. The reduction in stalling is reflected in the separate load and store queue graphs, which show fewer stalls compared to a combined queue design.

The split design also allowed independent loads to execute out of order with respect to stores, further reducing pipeline stalls caused by memory dependencies. While separating queues increased the complexity of dependency checks, it minimized unnecessary memory requests for loads dependent on recent stores. Additionally, the store_found_count graph identifies stores that can forward data to dependent loads, demonstrating the functionality and efficiency of the forwarding mechanism. Reduced stalls and improved pipeline utilization demonstrated Split LSQ's ability to improve efficiency, especially in tests with high memory access. This is shown by the incrementing of store_found_count below, which would have been 0 before implementing the split LSQ feature but now handles store forwarding.

Store_found_count Data Visualization for a Dependency Test (Figure 5)

Supporting Data



Processor Data Visualization (Figure 6.1 & 6.2)

Advanced Features - Performance Analysis

Benchmark Analysis

This section analyzes custom processor performance using GEM5 simulations to assess how branch prediction, prefetching, and frequency scaling impact behavior. Key metrics such as IPC, CPI, cache miss rates, and memory latency were evaluated to identify bottlenecks and guide optimizations. The analysis provides insights into improving processor efficiency and scalability by focusing on instruction fetching, memory accesses, and branching.

We configured GEM5 locally in a virtual environment using Ubuntu Linux to avoid issues with the EWS environment. A simple out-of-order processor was set up, and advanced features were added one at a time for testing. Benchmarks were modified to ensure they operated within the correct memory space and terminated properly, then converted into ELF files. Simulations were run on the benchmarks, and statistics were compiled for multiple configurations. The collected data was analyzed to evaluate the impact of each configuration on performance metrics.

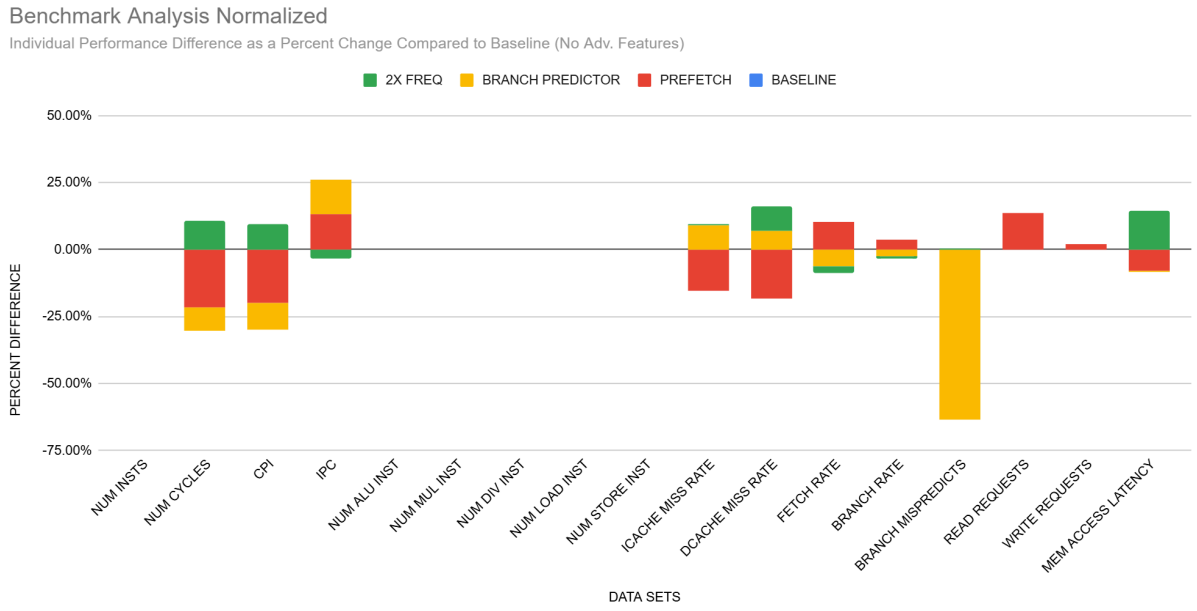
The processor workload is read-heavy, with 1,628,343 integer ALU instructions dominating and load instructions (717,576) far outnumbering store instructions (230,797). Instruction cache inefficiencies are evident with a 23.08% miss rate, while the data cache is utilized effectively with a 0.75% miss rate. The memory system handles 42,990 read requests versus 1,026 writes, aligning with the workload's characteristics, but the average memory access latency is high at 28,802 cycles. Moderate fetching reliance is reflected in a fetch rate of 2.9033 and a branch rate of 0.10851, with 9,336 mispredictions stalling the pipeline.

Addressing the 23.08% instruction cache miss rate, prefetching improved fetch rates by 10.53%, decreased memory stalls by 8.01%, and lowered cache miss rates by 16% on average. For branching, improved prediction accuracy by 63.60% reduced pipeline flushes, leading to an 8.58% reduction in cycle count, though cache miss rates increased by 8%, which prefetching mitigated. Doubling the frequency improved CPI by 9.6%, reduced execution cycles by 10.58%, and improved fetch efficiency (2.43%). However, memory latency increased by 14.64%, highlighting the need for additional memory management improvements.

The split LSQ works well in workloads with many memory accesses and dependencies, as it forwards store data to dependent loads and reduces stalls. However, it provides little benefit in workloads with few memory dependencies. The next-line prefetcher is effective for workloads with sequential memory access patterns, reducing cache misses and latency. Still, it can hurt workload performance with irregular access patterns by causing unnecessary cache loads. The Gselect branch predictor improves branch-heavy workloads by reducing pipeline stalls, but it has a limited impact on workloads with few branches or predictable branch outcomes. Increasing frequency helps workloads with high compute demands but can worsen memory latency in workloads with frequent memory accesses.

This report found that the processor's performance is slowed by high instruction cache misses and long memory access times, even though it runs efficiently overall. Improving branch prediction and adding

better prefetching can help keep the pipeline running smoothly and reduce delays. Doubling the frequency showed promise, but better cache and memory management are needed to work well. We will optimize the instruction cache, add prefetching, and make branch prediction more accurate to improve the processor. These changes will help the processor run faster and handle tasks more efficiently. More detailed data can be found in the graph and accompanying raw data below.



Benchmark Normalized Average Data Visualization (Figure 7)

AVG	SIM INSTS	NUM CYCLES	CPI	IPC	ALU INST	MUL INST	DIV INST	LOAD INST	STORE INST	ICACHE MISS
BASELINE	2603507.667	1521187.333	0.6135976667	1.990038667	1628343.667	25573.66667	1216.666667	717576.3333	230797.3333	0.230867
PREFETCH	2603507.667	1190414.333	0.4912703333	2.253455333	1628343.667	25573.66667	1216.666667	717576.3333	230797.3333	0.195858
BP	2603507.667	1390744.333	0.5519053333	2.249906667	1628343.667	25573.66667	1216.666667	717576.3333	230797.3333	0.2518623333
FREQ	2603507.667	1682168	0.6725166667	1.922917	1628343.667	25573.66667	1216.666667	717576.3333	230797.3333	0.2315616667

AVG	DCACHE MISS	FETCH RATE	BRANCH RATE	BRANCH MISPREDICTS	READ REQUESTS	WRITE REQUESTS	MEM ACCESS LATENCY	FREQ
BASELINE	0.007500666667	2.903307667	0.1085193333	9336.333333	42990.33333	1026	28801.73333	100
PREFETCH	0.006135	3.208985	0.1127673333	9336.333333	48839.66667	1046	26494.73	100
BP	0.008042666667	2.719273	0.1056603333	3398	42995.33333	1027	28668.08667	100
FREQ	0.008174	2.832664333	0.1077553333	9379.666667	42988.33333	1025.666667	33018.01667	200

Benchmark Average Data Raw (Figure 8.1 & 8.2)

Design Space Exploration

We developed a Python script to automate the execution of the five given tests, extracting data including IPC, delay, and power. The script logged these values alongside the current configuration parameters and benchmark file names into a test output file. It also ran synthesis for the current parameters, recording slack and area metrics. We tested 12 configurations selected based on predicted performance benefits, each requiring about an hour, totaling approximately 12 hours of runtime.

We implemented a "resume" feature in the script to manage potential issues like EWS resets during overnight runs. This feature parsed the output file and allowed the script to continue from where it left off, cutting out lost progress. Additionally, we added a small helper function to simulate screen activity with a mouse press and prevent the system from sleeping, as tmux is incompatible with EWS.

Once all tests were complete, a second Python script calculated performance scores for each test using the PD^3A^(1/2) equation provided in the autograder. The script appended these scores to the test data and calculated an overall score by averaging the individual test scores for each configuration. The configuration with the best (lowest) score is then selected as the optimal setup. All scripts are highly configurable, supporting adjustments for different parameters and test cases, ensuring flexibility and reusability for future analyses.

Processor Visualization

Processor visualization was integrated throughout the advanced features discussed in the report. The modular design allows for easy addition to any commit, enabling data collection from the processor. Key metrics were selected to record and highlight the impact of advanced features. For more information and a detailed analysis of the collected data, refer to the performance analysis sections of each advanced feature above and the accompanying graphs comparing the benchmark processor to the advanced features processor.

Additional Observations

Major Bugs

One of the major bugs we had during CP3 was that the cache we implemented from the mp cache was broken. We encountered a case where the read mask being sent to the cache was turning into don’t cares. When investigating why this was the case, we realized that although the cache works for the most part but for some reason, when integrated with our processor, the dfp_write signal never went high. After realizing this, we had to switch our cache to another one of our team members mp_cache, and the dfp_write signal went high at the appropriate time when there was a dirty miss. However, with this addition, we realized there was a new problem where we would get an error indicating that the mask had turned to don't care when bursting the data from the cache. This was then solved by fixing our cache arbiter so that when the data cache is selected, the read mask to the instruction cache is always zero. Since we are not interfacing with the instruction cache, no read or write requests should be issued to that cache while we are interfacing with the data cache.

Another bug we found in CP3 was that the free list was losing physical registers, as in certain cases, physical registers were not being added to the free list after being dequeued off. This would cause the processor to lose all the physical registers and eventually stall forever. After examining the waveforms,

we realized that this was because, on ROB flushes, we were not taking into account the registers in the processor pipeline that got flushed without being added back into the free list. This was then solved by creating a backup-free list every time there was a branch instruction. If the flush signal went high, the free list was updated with the values from the backup free list, which was backed up when the branch instruction first arrived, eliminating the effects of the instructions afterward. This, for the time being, worked. Still, technically, it also lost physical registers and created a huge performance bottleneck in the future, which will be discussed in the performance bottleneck section.

Finally, when we integrated our advanced features, we passed all the tests except RSA. Initially, we were not passing RSA at all, as the program would just stop midway through and stall. After doing more random testing, we found that in our functional unit for control instructions, we would lose the data associated with an instruction before it was committed to rob, which would stall the processor indefinitely. This was fixed by changing some flags that would dequeue instructions of the control queue we created so that the instructions do not get dequeued prematurely. Then we started passing RSA all the way through but encountered spike issues. We realized this was due to our Split LSQ load forwarding not working properly. A store would happen to a memory address, and then a load from the same memory address would happen soon after, but it would not forward the value from the store for the load. We realized that this was because if a possible store was found in the store queue that could potentially be stored at the same address as what the load instruction is accessing, then it would wait until the address for the store is ready. Then, after it is ready, it compares the address to the load address. If it is the same, then the value is forwarded. However, if it isn't the same, it should continue checking the store queue, but instead, it determined that there were no matching stores found in the queue at all and sent the load address to the data cache to get the value where the cache does not have the updated value yet. We solved this problem by checking how many stores can be forwarded, and if the number is greater than 1 and the current store does not have the address, it continues checking the store queue and does not immediately send a request for the load instruction to the data cache.

Performance Bottlenecks

Once we finished CP3, the first major bottleneck we saw was that our delay was high. We chose to see if something was stalling our processor a lot. We discovered that the free list was too often empty before being updated with a value. When we investigated why this was the case using the waveform, we found that after a branch instruction, the RRF updates the free list. Then, the branch instruction gets committed, and the free list is changed to whatever the free list was before the branch instruction. Now, the physical register that was taken out of the RRF is not on the free list at all. We have lost that physical register forever. Similarly, like this, we lose many physical registers on flushes. Eventually, there is only a small amount of registers you can use for the processor, and it takes a while for them to get back onto the free list, so the processor stalls for a long time while the free list is empty. The first list is also saved to a backup whenever there is a branch instruction. However, if there is a branch instruction and another branch instruction arrives before the first branch instruction commits, the backup will be saved again in a new state. If the first branch instruction triggers a flush, the free list will be updated with the state of the free list at the second branch instruction arrival and not the first. We realized the best way to solve this would be to make the free list the opposite of RRF on a flush. Whatever is not in the RRF should be on

the free list. We solved this by creating an inefficient double-for-loop that would check what values are in the RRF and compare that to all the physical registers possible. If that register is not in the RRF, it will be added to the free list. This worked, and the free list was rarely empty after that, which can be seen by looking at our empty counts from before and after the advanced features. Our IPC, on average, also jumped from 0.20 to 0.37.

We then needed to improve our frequency, which was capped at around 166 Mhz. We needed to improve our slack. After running a synthesis, we found that the critical path was through the double for loop we had created, and we realized this was severely inefficient. Our free list was initially a queue with the actual register values being enqueued and dequeued off the list. However, we decided to change this to avoid the double for loop. Instead, we made the free list the length of the number of physical registers, and we had a single bit to determine if it was free or not and where an index into the free list would be the physical register. 1 would mean it is free, and a 0 would mean it is not. Now, all we had to do on a flush was to see everything in the free list to free and then run a single for loop through the RRF and mark every register in the RRF as invalid in the free list. We removed the head and tail pointers that were once there, as there is no longer a queue. The slack improved significantly, and we increased our frequency, reducing our delay.

To improve our slack even further, we noticed that our critical paths were through the division and multiplication units. To solve this, we registered these units even further by adding the number of cycles it takes for multiplication and division IPs to complete. This improved our slack even further, increasing our frequency to 433 Mhz.

Scripts

We used Python scripts to generate random instructions for debugging and testing during CP3 and beyond. Testing began with simple operations like addition and subtraction to check basic functionality. We then expanded to include multiplication and division, ensuring correct handling of edge cases such as overflows and signed arithmetic. The Python script allowed us to create custom instruction lists that could be modified easily, enabling tests for any specific instruction set. It also allowed us to adjust the number of instructions generated, making the process simpler and more useful than configuring the randtb file. This method also allowed us to run spike on long instruction sequences, another benefit over configuring randtb.

We initialized all registers with a wide range of values to test operations across various inputs and accounted for the full range of immediate values in each instruction. The tests covered all instructions, excluding branch, jal, and jalr. This setup provided a reliable way to test functionality for various instructions.

After CP3, we modified the script to address load and store operations, which encountered memory access issues with the RSA test. We configured specific registers to load initial values and allowed consecutive load and store operations to run alongside ALU, multiplication, and division instructions. This exposed problems with forwarding and dependencies, which we corrected to fix the RSA bugs. These updates

made sure that the processor handled memory operations correctly. The specific debugging steps and resolution are discussed in more detail in the bugs section of the repot.

Conclusion

The final implementation of our out-of-order processor demonstrates the integration of architectural features that significantly improve performance. The split load-store queue resolves memory dependencies by forwarding data from stores to dependent loads and supporting the out-of-order execution of unrelated loads. The Gselect branch predictor reduces pipeline stalls by improving branch prediction accuracy using a global-local history mechanism and a pattern history table indexed by concatenated PC and history bits. The next-line prefetcher reduces instruction cache miss rates by fetching sequential cache lines, minimizing memory latency.

Performance optimizations were validated through GEM5 simulations and synthesis analyses. These showed significant reductions in instruction cache misses, improved IPC, and minimized processor stalls. The critical path optimizations, including modifications to the free list and additional pipelining in arithmetic units, allowed for increased clock frequencies and reduced delay cycles. The free list redesign eliminated inefficiencies, enabling proper physical register management and improved throughput.

Challenges, such as resolving free list stalling, ensuring correctness in branch resolution, and optimizing the load-store queue, were addressed through debugging and architectural changes. The synthesis results confirmed that the processor achieved a clock frequency of 433 MHz. The final processor efficiently handles instruction-level parallelism, resolves memory dependencies, improves branch prediction accuracy, reduces cache latency and demonstrates performance gains across several key metrics.