Fresh off the boat: mp_ooo report

1 Introduction

Our project is a speculative, out-of-order processor built on the explicit register renaming (ERR) microarchitecture and based on the conceptual foundations of instruction-level parallelism. Our processor is targeted to run on the RISC-V instruction set. RISC (Reduced Instruction Set Computer) is a special category for processors such that the goal is to execute one instruction per clock cycle. It is a streamlined version of the Complex Instruction Set Computer (CISC). Essentially, processors running programs on a RISC foundation generate a larger number of instructions but with reduced clock cycles per instruction, thus creating a tradeoff favoring software over hardware and consuming larger regions of instruction memory in exchange for higher speed. Thus, our processor is built on the principles of larger numbers of instructions per program and a high RAM demand. RISC-V is a rich ISA with a myriad of instructions performing register-level and system-level operations, but in our processor, we simplify things by focusing mainly on the processor/register state rather than the system-level state. The aim of our project concerning our learning in computer architecture fundamentals is to explore the principles behind processor optimization and understand the underlying hardware that software needs to run on. This project also taught us particular design tradeoffs, especially those in area and power. We saw firsthand the tradeoffs between processor speed and power when constructing our processor. This is apparent with the many hardware units and data structures necessary for instruction-level parallelism. The underlying hardware we employed/built consists of register arrays, SRAMs imported from IPs (Synopsys DesignWare sequential divider/multiplier), ALUs, and a common data bus (CDB). The construction of these hardware units helped us both to grasp further the difficulties of processor optimization and to appreciate why this field is so important in accelerated computing and applications.

2 Project Overview

The project was split up into two parts: baseline and optimized. The baseline was split up over three weeks, each week comprising a single checkpoint for a total of three checkpoints. Each of these checkpoints had its own goals: to divide the work accordingly, we analyzed them and discussed who wanted to take on each goal. We then created designated branches on GitHub with our name, the checkpoint number, and the goal/hardware unit tasked to implement. Once we implemented each of our tasks, we came back together to merge our branches into the main branch, squash merge conflicts, and perform testing and debugging on the main branch with our features. We used a message thread with pinned messages that outlined who was assigned to which task. We also pinned messages with instructions detailing version control workflows, including git branch creation/checkout, hard resetting/reverting, force pushing, and merging. Doing this ensured that we had a smooth workflow, but we ran into a few conflicts between code implementations in our processor. We also had different code styles with formatting and

indentation that varied from person to person, which was a hindrance in some way, but it allowed us to identify each person's code. For the optimized portion, which consisted of the final three weeks, we started on advanced features but also experimented with optimizing our processors to improve our ranking in the competition and make improvements to our timing and area.

3 Design Description

(a) Design Overview

Checkpoint 1 had 4 goals: to create a block diagram detailing the high-level infrastructure of our processor, to implement a functional parameterizable queue, to implement the fetch stage, and to construct a cacheline adapter serving as an interface between the instruction cache and the DRAM burst memory model. This checkpoint was simple to divide among each of us. Each of us took one of the implementation tasks. We all collaborated on the block diagram with help and input from office hours. We also utilized office hours for the cacheline adapter and the queue.

Checkpoint 2 involved implementing all the other stages, including decode, rename/dispatch, execute, and commit. We implemented support for ALU instructions (add, sub, sll, srl, sla, sra, etc), multiply and divide instructions, and lui. To accomplish this, we designed a myriad of necessary hardware units and components. These include the reservation stations, ROB, RAT, functional units, physical register file, RRAT, and CDB. We also hooked up our output signals to the RISC-V Formal Interface (RVFI) for ease of debugging.

Checkpoint 3 involved finishing off the baseline with support for memory and branch instructions, with the ultimate goal being to pass the provided benchmarks. We added more reservation stations and functional units to complete this, changed the PC logic, abstracted the PC into a separate PC queue, and created a data cache and cache arbiter to arbitrate between instruction memory and data memory.

After checkpoint 3, we implemented advanced features and optimized our processor. We implemented the post-commit-store buffer and a stream prefetcher and made changes to reduce delay and area. We attempted a few other advanced features but did not feel good about our results and did not include those in our final processor.

(b) Milestones

(i) Checkpoint 1

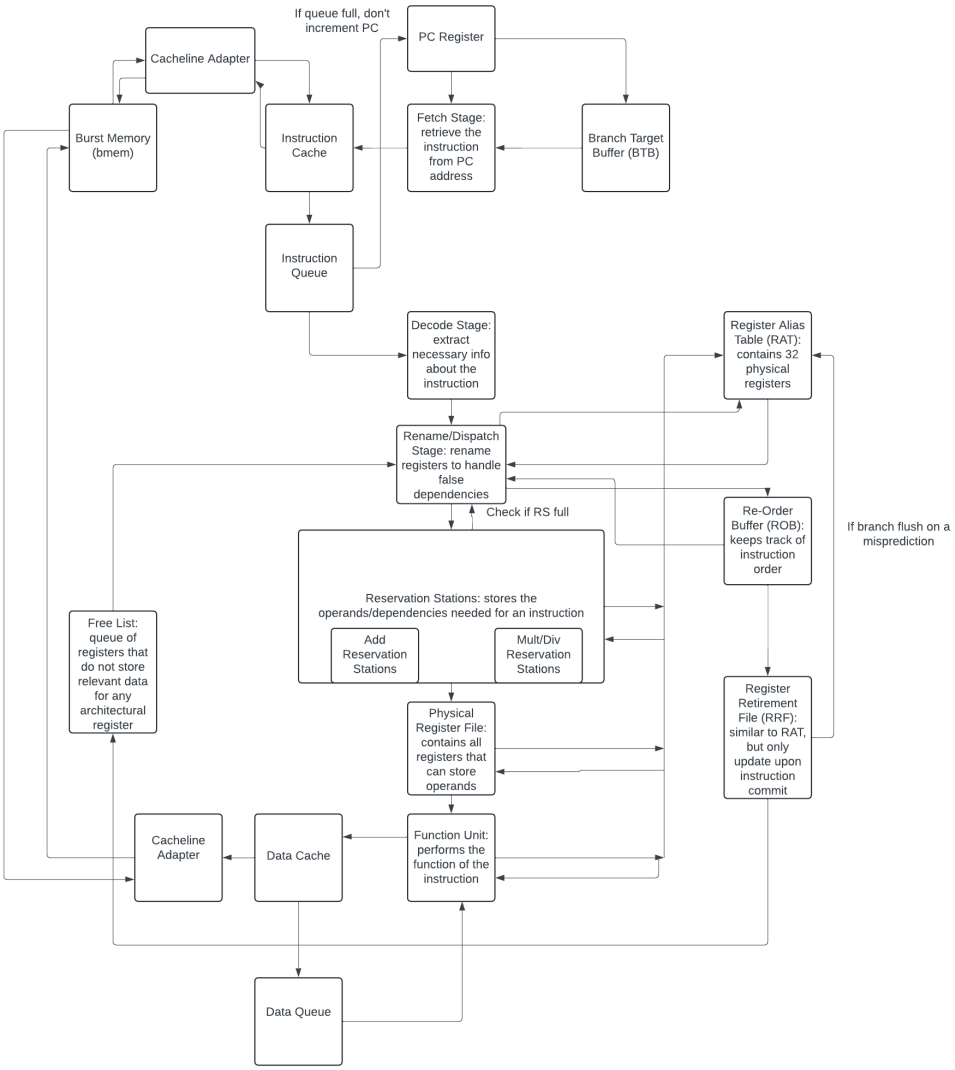
For this checkpoint, we were tasked with implementing the queue, the cacheline adapter, and the fetch stage, as well as a high-level block diagram.

The queue is parameterizable. Using the imported parameter from pkg/types.sv, we can experiment with the size of an arbitrary queue by changing that number without needing to make huge code changes. The queue contains QUEUE_DEPTH entries; each entry is of size DATA_WIDTH:0. Note that an entry is NOT of size DATA_WIDTH - 1:0. A queue is indexed by a head and a tail for dequeuing and enqueuing. The “head” and “tail” registers are each of size $clog2(QUEUE_DEPTH):0. Again, the head and tail pointers are NOT of size $clog2(QUEUE_DEPTH) - 1:0. The extra bit in the queue is for validity: checking if an entry inside the queue is valid. The extra bit in the head and tail registers is an overflow bit. Empty and full logic in the queue is determined through this overflow bit. The queue takes in data and an enqueue_valid signal and outputs data if there is a dequeue_valid signal. We tested the queue by creating a special testbench for the queue and looking to ensure that output data and the dequeue_valid signal are high on the same clock cycle. We tested various data inputs and tested for underflowing and overflowing.

The cacheline adapter takes in 4 “bursts” from the cache/DRAM and sends them one at a time to the DRAM/cache. For this checkpoint, however, we didn’t need to worry about stores or loads to memory. We read from memory when fetching an instruction, so we only need to worry about memory read requests. Therefore, in our cacheline adapter, when reading burst memory, we used an adder, a register, and a comparator to track how many bursts we obtained. Once we reach 4 bursts, we send the 256-bit cache line back to cache.

For the fetch stage, we decided it would be easier to implement this stage directly in the CPU. We implemented an instruction queue and fed instructions from imem to this queue. Our logic is simple, with the only nuance being how we change the program counter (PC). Since we have to account for the possibility of the instruction queue being filled up, we change the PC only if an instruction can be enqueued and we get a ufp_resp from the instruction cache.

The last part of this checkpoint involved creating a block diagram that could serve as a base for the later checkpoints and provide us with a blueprint for our other implementations. In hindsight, we definitely could've added more detail to it, including labels and what specific data needed to be transmitted between hardware modules. Putting more thought into this block diagram would've helped guide us through the later stages of our processor construction. Nevertheless, this is the result we produced during this checkpoint:



(ii) Checkpoint 2

Our next course of action was to handle all ALU and mult/div instructions, as well as lui, and ensure that our processor could run out of order. To do this, we had to implement and instantiate most of the hardware and connect them, meaning that we had to write the most RTL code for this checkpoint. Since we also handled a subset of the RISC-V instruction set, we wrote a random testbench

with proper coverage and bins to ignore instructions that we knew we didn't have support for yet. In addition, we hooked up RVFI to signals generated from the commit stage to ease the debugging process.

Since our processor utilizes the ERR microarchitecture, we wanted to map each architectural register (32 total) to a physical register. The purpose of the renaming is such that false dependencies can be handled out of order. The register alias table (RAT) holds each of these mappings. Since the RAT can hold speculative mappings, we also needed a register retirement file (RRF), or what we called the retirement register alias table (RRAT). The RRAT is used to communicate with the free list since whenever we complete an instruction, we can enqueue the physical register we used to the free list to indicate that it is available for future use. The RRAT will also be useful during flushes, which will be discussed in the Checkpoint 3 section.

We also had to implement the rename/dispatch stage, where we check if an instruction is ready to be dequeued from the instruction queue. If so, we extract the necessary information from each instruction and communicate with the RAT to determine if the registers needed for the instruction are available or if there is a dependency. After that, we send that to the appropriate reservation station based on what kind of instruction it is.

The reservation station stage is where we receive the information about an impending instruction from the rename/dispatch stage and register availability from an FU's CDB. This information includes register availability and whether or not a register is busy. We implemented 3 reservation stations and added 2 more in CP3: add, multiply, divide (and later memory and branch). The module is split into three logical portions: insertion, updation, and issue. The insertion condition is when rename/dispatch sends a dispatch_valid signal and a select signal to the reservation station stage. This tells the module which reservation station to insert an entry in. The update condition is when an FU signals to the reservation that it is done with a source register. A reservation station entry contains valid signals for its two source register IDs, and they are set high when that corresponding register becomes available for use. The issue condition is when a reservation station entry can now issue an instruction to the corresponding functional unit after both of the source registers needed in that instruction are available. When this happens, the entry is marked as not busy, and all the relevant information is combinationally read from that entry and propagated to the CDB. A typical reservation station flow pattern goes as follows:

Step 1: Insertion: The module receives an instruction from rename/dispatch, and it updates an entry in a reservation station and marks the entry as busy.
Step 2: Updation: That instruction waits for several clock cycles to receive confirmation from an FU that its desired source register is ready for use again.
Step 3: Issue: Once both source registers for that entry are marked as valid, the entry is marked as not busy on the subsequent clock cycle, and the information is broadcasted to the CDB with a cdb_valid signal set high.

The physical register file is another module that we had to implement. This module contains the actual data contained within each physical register. It receives the physical register index from the reservation station and then sends the corresponding data of each operand of the instruction into the functional unit. Then, from CDB, it receives the physical register value that needs to be written to the instruction's output register and updates it accordingly.

Finally, we implemented functional units for ALU, multiply, and divide instructions. We had an execute module containing instantiations from each of the functional units. Once this module receives the start signal from the reservation station, it triggers the corresponding functional unit. The functional unit for ALU instructions is done with combinational logic and broadcasts the CDB on the same cycle that it starts. The logic we used was derived from previous MPs. For the multiply and divide/remainder units, we imported IPs from Synopsys DesignWare. For both, we used sequential IPs (mult_seq and div_seq). We set our multiply to take 3 cycles and our divide/remainder to take 12 cycles. Similarly, these two units also broadcast the CDB upon completion, while also sending a busy signal to ensure that another reservation station entry does not send a start signal while the current one is running.

We also hooked up our RVFI at this stage. We expanded the rob_entry_t struct to include RVFI signals. As such, the ROB receives its RVFI signals from different stages. The register src and dest IDs, PC info (except in the case of a branch, where a branch signal and input PC is passed in), instruction, and regfile write enable bits are inherited from rename/dispatch. The other signals are received from a functional unit and passed via the CDB. The CDB passes in an index to the ROB so that the correct entry is updated in the ROB. When a ROB entry is dequeued, the rob_entry_t struct is output to a signal in the CPU, and RVFI is hooked up to these signals; order is also incremented. In CP2, we hardcoded memory-relevant signals to 0. In CP3, memory-relevant signals were propagated from the memory CDB to the ROB, where updation occurs.

We also created a random testbench. We enabled coverage and constraints for the instructions we supported and configured the generation of random instructions under these constraints using mp_verif as a base. We later expanded on this random testbench in checkpoint 3.

To handle register x0, we simply always had architectural register x0 map to physical register x0, and we do not update the physical register file if x0 is attempted to be modified.

(iii) Checkpoint 3

With the implementation of the core stages and hardware units necessary for the baseline, we finished our baseline with the last pieces of the puzzle: memory and branch instructions. In hindsight, this checkpoint was the most difficult due to issues with debugging rather than implementation. We had three simple but time-consuming tasks: implement a load-store (or memory) queue, instantiate a data cache, and write logic to arbitrate between requests to the data cache (loads/stores) and instruction cache (fetching).

The cache arbiter is a module that redirects memory requests and cache data between the cacheline adapter and the CPU's caches. We used simple state machine logic consisting of 3 states: data, instruction, and idle. When planning the theoretical implementation, the state machine would idle until the cache delivers valid data and a ufp_resp, and the data and instruction states are visited in a round-robin fashion. However, despite some notes taken on paper, we wrote the cache arbiter incorrectly but did not have any way to test it until the load/store queue was finished. This slowed down our process and caused unnecessary debugging. We had to adjust for stalling, cache misses, and misaligned clock cycles. The main problem was the possibility of requests for instruction and data at the same time. Our state logic chose one over the other but completely discarded the other request: thus, we had to instantiate registers and register flags to save the original request and send a new request later. For example, if the cache arbiter is in the instruction state but gets requests for both instruction and data memory, it sends a request to the icache but saves the original data address and data request signal in registers, then uses those registers and the state logic in the next clock cycle to honor the original request for the dcache. We did not account for the discard of an icache or dcache request originally, which messed with our design. We eventually rectified our state transition logic and implemented arbitration to ensure both data and instruction memory requests were satisfied.

Regarding memory instructions, we had to consider what necessary steps need to be taken in the event of memory instructions. Two factors are necessary for a memory instruction to be issued to the ROB: the calculated address and the type of memory instruction (a load or a store?). Thus, the memory functional unit is an adder that calculates the address. This address needs to be sent to the cache, which can take multiple cycles, so a **load/store queue** is implemented. The load/store queue gets its information from the rename/dispatch stage and the memory functional unit. The basic flow for a memory instruction goes as follows:
Step 1: Enqueue: The memory queue sends its tail pointer to the rename/dispatch stage. rename/dispatch and subsequent stages will propagate this pointer through the reservation station and functional unit stages; meanwhile, the rename/dispatch stage decodes the received instruction, determines that it is a memory instruction, and sends that information to the load/store queue. The queue then updates its entry with that memory instruction but does not know the address, so it will not send a request to the data cache yet.
Step 2: Wait for the address: The memory instruction is sent to its reservation station. When the instruction's source registers are ready, it is sent to the functional unit. In addition, the reservation station sends the tail pointer that it inherited from the LSQ. The functional unit calculates the address and sends that address and the original tail pointer back to the LSQ. Read and write masks are also sent to the LSQ.
Step 3: Updation + Cache request: The LSQ updates the corresponding entry using the inherited tail pointer. The memory instruction's address is known, so it sends a request to the data cache on the subsequent clock cycle.
Step 4: Cache Data received: The LSQ gets a ufp_resp and data from the cache. It now knows it has valid data, so it sends the relevant data to the ROB. It updates memory-specific data for RVFI (r/w masks, r/wdata) and propagates those through the CDB. Even though there are 5 reservation stations and 5 FUs, the memory CDB is connected to the LSQ, not the memory FU.

We tested the functionality of the load/store queue with a simple testbench to ensure that entries were enqueued, updated, and issued properly for the correct enqueue and valid signals. Since we did the branch and memory instructions separately, we then tested a version of the processor with CP2 code merged with the load store queue, enabled memory instructions in our random testbench by enabling coverage of load and store instructions, and continued to debug from there.

Regarding branch instructions, we had to implement hardware similar to ALU and mult/div instructions, including a new reservation station, functional unit, and

CDB. The functional unit was done in combinational logic, meaning it only takes one cycle. In hindsight, we could have used some more registers in the branch functional unit, as it made implementing a more advanced branch predictor much more difficult due to the large critical path. Additionally, the flush signal and the flush address signal were calculated in the functional unit. These signals indicate that there was a branch mispredict and later will be used upon a commit to restore the CPU to the old state before the branch prediction. We added these two signals to our CDB. Our branch prediction method was static non-taken, meaning that we always predict that the branch is not taken, and flush whenever a branch is taken. The benefit of static non-taken is that the branch prediction is easy to implement, while the drawback is that it does not have a very high prediction accuracy.

Once we commit a branch instruction from the ROB, we check whether the flush signal is high. On a flush, there were some extra steps that we had to take that wouldn't happen on any other instruction. Since the processor continues to perform speculative instructions, we need to flush those to prevent them from being processed and committed. We had to flush the ROB, instruction queue, reservation stations, and functional units, while also marking the free list as full. In addition, we had to update the program counter to the flush address signal we calculated in the branch functional unit. Finally, since the RAT holds speculative register mappings while the RRAT holds the previously committed register mappings, we flush the RAT by restoring its contents with the contents of the RRAT.

We also had to edit some RVFI signals concerning a flush. On a mispredict, we would have to change the pc_wdata of the RVFI of the branch instruction to the flush address signal. We also had to change some logic with the RVFI's order, since we originally did not account for the fact that the order needs to be flushed as well. This was pretty simple, as all we had to do was move the order logic into the ROB and only increment upon a commit.

As previously mentioned, we tested the branch and memory instructions separately. Concerning testing the branch instructions, we first tested the branch instructions with basic assembly files which included a mix of taken and non-taken branches. We started with simpler test cases which included only a few branch hits and misses, and then moved on to larger test cases with multiple loops.

Later, we merged the branch and memory instructions. Firstly, we first had to fix many merge conflicts. After that, we finally tested our baseline against the

provided benchmarks. Once we ensured that all of the provided benchmarks passed with no RVFI mismatches or Spike conflicts, we knew we had full functionality.

(c) Advanced Features

(i) Stream Prefetcher

To implement a stream prefetcher for our instruction cache, we began by creating a next-line prefetcher. This required modifying our existing cache code to allow simultaneous memory access and cache reads. Our implementation is as follows:

When the instruction cache misses and retrieves a new set of instructions from memory, prefetching is triggered. The targeted address, calculated as current PC + 32 with the last 5 bits zeroed out, is sent to the instruction cache with a flag identifying it as a prefetch. If this address is already in the cache, nothing happens. Otherwise, the cache initiates a memory fetch without stalling (ufp_resp = 1) and stores the data upon completion. If a miss occurs during prefetching, the cache stalls until the prefetch memory fetch finishes.

Building on next-line prefetching, stream prefetching was straightforward to implement. If a fetched address is less than the current address, we increment a counter. Once the counter exceeds a set threshold, the prefetched direction changes to before the current PC, thus allowing us to prefetch in the opposite direction. Conversely, if the fetched address is greater than the current address, the counter resets, and forward prefetching resumes. Since instruction flow typically moves forward, we set the counter threshold high. As such, backward prefetching rarely occurs.

With stream prefetching, our processor performance showed measurable improvement. In the Coremark test case, IPC improved by 1.4%, rising from 0.4196 to 0.4254. Performance counters revealed a prefetch accuracy of ~94% with 1177 out of 1253 prefetches containing useful instructions. Despite the high accuracy, our IPC gain was modest. After analyzing our prefetcher's effect, we found that the instruction queue empties before prefetching completes, causing stalls until the prefetched instructions are ready. As such, the IPC increase was not as significant as expected. The primary tradeoff was an area increase from ~233000 μm^2 to ~238000 μm^2, which we deemed acceptable given the IPC improvement.

Our prefetcher performs best in sequential instruction flows, where the PC increases without jumps. However, frequent branches degrade performance. Since

ongoing prefetch memory accesses cannot be canceled, branches force stalls until prefetching completes, delaying access to the branch address.

(ii) Post-Commit Store Buffer

To implement a post-commit store buffer, we first modified the connection between the memory queue and the data cache. Instead of directly signaling the data cache when instructions are at the head of the memory queue, store instructions are sent to the store buffer. Once in the buffer, the store responds, allowing it to be committed in the ROB. The store only executes when it reaches the head of the buffer where it begins its memory access. For load instructions, the address and mask are compared against the entries in the store buffer. If a match is found, the load retrieves the store data directly, bypassing memory access. Otherwise, the load stalls until the store buffer is cleared and all queued stores are completed.

We determined the optimal size of the post-commit store buffer by analyzing the number of stores in test cases. The goal was to ensure the buffer was large enough to avoid frequent stalls but compact enough to minimize space usage. Based on this analysis, we selected a size of four entries.

Introducing the post-commit store buffer resulted in a decrease in IPC, from 0.4196 to 0.3893 in the Coremark test case. Although the buffer was correctly implemented with L0 caching and quick store commits, it conflicted with our processor design. Specifically, our cache arbiter prioritizes data requests over instruction requests. Previously, gaps between memory instructions allowed the arbiter to grant memory access to the instruction cache. However, with the store buffer, these gaps disappeared as the buffer monopolized memory access. This caused stalls in instruction fetching until the store buffer was cleared.

Despite the IPC reduction, performance counters indicated efficient utilization of cycles made available by instantaneous store commits. Without the buffer, 33,658 cycles would have been wasted on store stalls—about 4.4% of the 768,979 total cycles—representing a meaningful optimization.

The post-commit store buffer performs best with isolated store instructions followed by non-memory instructions. This setup allows stores to commit instantly while non-memory instructions execute, and the buffer gradually writes to the cache without delaying instruction fetching. Conversely, performance degrades with numerous memory instructions. A full store buffer stalls the processor, while unmatched load instructions also stall until the buffer clears.

Here is a summary of the performance gains/costs that our advanced features yielded for us. All of these are for Coremark.

Features	Baseline	Stream Prefetcher	Stream Prefetcher + Post-Commit Store Buffer
IPC	0.4196	0.4254	0.3943
Power	35.857 mW	38.742 mW	38.518 mW
Delay	1389.85 μs	1405 μs	1495.31 μs
Area	232791 μm^2	240654 μm^2	245300 μm^2

4 Conclusion

In summary, the goal of our project was to code a RISC-V out-of-order processor based on an ERR microarchitecture, while also incorporating some advanced features to improve performance. Although there were some things we would have done differently if given another opportunity, we were still proud of the fact that we were able to achieve a functional processor that passed all of the benchmarks. Overall, this project was a transformative journey that turned all of us into better engineers. It not only sharpened our computer architecture and coding skills, but also in planning, teamwork, and surviving countless late-night debugging sessions.