ECE 411

Spring 2025

Team 14 Out of Order Processor

Elliot Dziemiela (edziem2@illinois.edu), Prakatish Krishnan (pk31@illinois.edu), Xuhang Xiao (xuhangx2@illinois.edu), Dhruv Panchmia (dhruvp4@illinois.edu)

Table of Contents

Introduction

This project implements a 32-bit out of order RISC-V processor modeled after the Tomasulo algorithm. We first implemented a base processor that was able to run the RV32IM instruction set (with the exception of FENCE, ECALL, EBREAK, and CSRR instructions), and then included a variety of advanced features to improve the processor's performance.

In order to verify the functionality of our processor, we built unit tests to run on our processor and also provided benchmark programs. The testing was configured to provide functionality matches as well as performance metrics. Using these tests allowed us to identify limitations and errors in our processor and come to a final working implementation.

This report will provide a thorough outline of our process in designing, developing, and verifying the out of order processor, and our thinking behind the steps we took. Building an out of order processor was imperative in giving us a deeper understanding of computer architecture and allowed us to put the information we learned in lecture into practice.

Project Overview

When it came to designing and developing our out of order processor, the project was split up into multiple checkpoints to help us compartmentalize certain aspects of the CPU. The checkpoints also kept us on track by giving us explicit requirements and deadlines to meet. Each checkpoint was started by identifying the required deliverables and how the work could be split in the most efficient manner. The goal for each checkpoint was to finish the minimum viable requirements for the checkpoint and then focus on improvements to those. That way we could ensure that we have the base functionality and not get lost implementing other aspects. We also set deadlines within the checkpoints to have each of our parts done so we could combine and test them as a whole.

There was only so much we could each do separately to progress the processor, as all of the subcomponents had to mesh with each other. A significant amount of our work included meeting in person to ensure that we were all on the same page and that each person's specific part would work with the overarching design we agreed upon. This was also done so each teammate could have a deep understanding of the processor as a whole and not just their individual contribution.

Design Description

(a) Overview

After deciding what type of architecture we would use, we worked on the implementation of the base RISC-V instruction set along with the RISC-V M extension. After getting a design that allowed us to run the base set and the M extension, we were tasked with adding a few different advanced features to help our design execute instructions more efficiently by increasing the IPC of the processor and also by limiting critical paths that could otherwise limit the frequency at which our processor operates.

(b) Milestones

i. Checkpoint 1

Checkpoint 1 marked the beginning of our out-of-order processor design. During this phase, our primary focus was on understanding the principles of out-of-order execution and implementing several foundational components. The key modules developed included the cache line adaptor, the instruction fetch stage, modifications to the provided cache with an added line buffer, and an instruction queue.

The cache line buffer (inside cache_with_linebuffer.sv) serves to decrease the instruction cache (icache) hit latency. It does this by loading the most recently read line into a buffer which is then read directly from the idle state every time there's a read with a match. This avoids the latency of a normal read where addresses must be given to all of our internal arrays for them to only respond on the next cycle. This was tested by running simple test code and observing that consecutive icache reads to the same line were happening in one cycle.

The cache line adapter (cacheline_adaptor.sv) serves to translate the burst reads from burst memory (bmem) into single 256-bit lines to be read by the caches, and to translate the 256-bit written lines from the caches into four 64-bit bursts written to bmem. For its design, we chose to implement a finite state machine with states IDLE, READ, WRITE, and RESPOND. For the IDLE state, we wait for a read or write request from the up-facing port (ufp). Upon a read request, we register the address, set the bmem_read (the signal that tells bmem to read) register high, and move to the READ state. Upon a write request, we register the address, fill a buffer with our write data, and move to the WRITE state. In the READ state we wait for a response from bmem. When that response is received, we check that the address provided by the bmem response matches our registered address for this transaction, and if so, we then burst to our data buffer and increment our burst count from 0 to 1. We repeat this three more times, where each time the incremented burst count gives a new bit select of our buffer to copy data into. Once the burst count reaches 3, we enter the RESPOND state where we simply assert the ufp response with the ufp read data assigned to our data buffer. Then we immediately move back to the IDLE state. The WRITE state follows a similar structure but instead of reading bursts into our buffer, we write bursts out of our buffer into bmem. When the burst count reaches 3, we similarly enter the RESPOND state. To test our adapter, we wrote a testbench (adaptor_tb.sv) that simulates a read transaction and a write transaction according to spec.

The instruction queue was a fairly simple circular queue design, with the ability to deal with enqueues, dequeues, and simultaneous enqueues and dequeues. We parametrized the queue and kept the design simple, outsourcing the fetching logic to the fetch module. The fetch module accepted a PC value and worked with the instruction cache to fetch the instruction at that address. It then fed the instruction to the instruction queue, where it waited to be dispatched. The complexity for this instruction fetch design at checkpoint 1 came from having possible fetch stalls when the instruction queue was full, in which case the fetch stage held the current PC until the queue was free to accept enqueues. Testing for this system consisted of unit tests for the queue, testing for all of its operations, and feeding placeholder PC values into the fetch stage to observe functionality between fetch, the instruction cache, and the instruction queue.

A major design decision in this checkpoint was to base our processor on Tomasulo's algorithm, as illustrated in the block diagram in Figure 1. We chose Tomasulo over the Explicit Register Renaming approach due to its simpler dependency tracking and data forwarding mechanism, which allow for a more modular and manageable implementation during development.

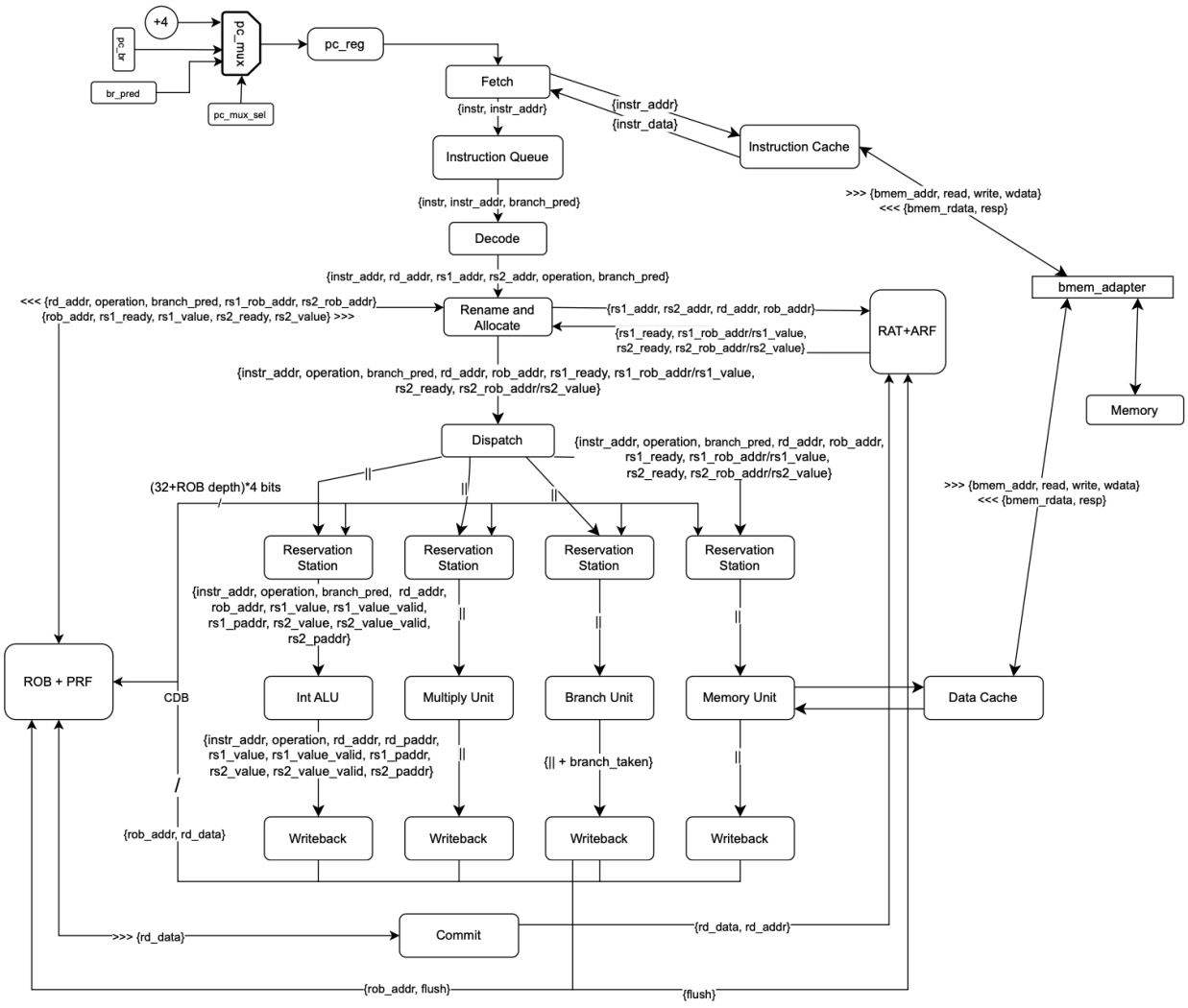

Figure 1: Block Diagram

ii. Checkpoint 2

In Checkpoint 2, we implemented and integrated the key components necessary to enable out-of-order execution in our processor pipeline. We are now able to correctly execute all immediate and register-type instructions defined in the RV32I specification.

The modules developed and refined include the decode and dispatch stages, the register file with an integrated Register Alias Table, the reorder buffer, reservation stations, functional units such as ALU and MUL/DIV module, the common data bus and its arbiter, and the commit stage.

The reservation stations are what instructions are dispatched to before they are issued. These are necessary for out-of-order processing, and the instructions wait in these stations until their operands are ready (usually via broadcasting from writebacks on the central data bus). We implemented these with tables of entries containing all the necessary data for the instruction. Whenever an instruction is dispatched to the station, we loop through all entries to find the first free index, and if one exists, we insert the instruction. We also snoop the cdb, and whenever there is a broadcast, we loop through all table entries and assign the cdb value wherever there is a match with the cdb register and a source register. Every cycle we also loop through the table and issue the first entry we find that has its operands ready. Unknowing that it was an advanced feature, we implemented age ordering to prevent starvation. However, for our cp3 baseline, we reverted this change. For testing, we created a testbench (reservation_station_tb.sv) that dispatches several instructions, broadcasts cdb values, and allows instructions to issue and have their slots filled with new dispatched instructions.

The CDB arbiter (arbiter.sv) serves to arbitrate between writeback values from execution units to send on the cdb. To prevent starvation, we used a round robin approach where after every writeback we prioritize the next execution unit in our checks. We first check if the execution unit at the round robin index is requesting a writeback, and if so, we broadcast it and send confirmation to the unit. Otherwise, we loop through all units and broadcast the first value that is ready. For testing, we wrote a simple testbench (arbiter_tb.sv) that sends execution unit writeback requests in an order that allows the arbiter to broadcast based on the round robin index to verify it works. The cdb is a combinational unit that accepts values from the cdb arbiter and puts them on the common bus for other units like the reservation stations and dispatch stage to use.

The commit stage is what writes destination register (rd) values from the rob to the register file. The commit stage is always reading the entry at the rob head. If that entry is both ready and its completion status is high, then we write the rd value to the register file. If the opcode is a branch, then we also check for mispredictions here by comparing the predicted and actual branch outcome fields of the rob entry. If a misprediction occurs, we update the pc with the correct value that should’ve been predicted. We tested this stage by running test code and verifying that values written to the register file were correctly read by subsequent instructions, and that branch mispredictions were recovered from in the correct way that satisfied the spike monitor.

The reorder buffer (ROB) acts very similar to a circular queue. We have multiple modules contributing to the data stored in the ROB. Firstly, the dispatch stage, which handles renaming and updates the rob with information about the instruction being dispatched such as the instruction opcode, operand values, etc. The ROB then returns the rob tag/address associated with that instruction back to the dispatch stage. The ROB is also accessed after an instruction finishes executing and passes through the cdb, where any incomplete values like the result of the instruction get updated before being committed when they arrive at the ROB head. To test this unit, we wrote a testbench that simulated updating the ROB with placeholder units and ensuring that the correct entry (head) was being committed. Other parts of the ROB test were very similar to the instruction queue test in checkpoint 1.

Another major aspect of checkpoint 2 was to have working functional units. For this checkpoint we were only required to have all register and immediate instructions, including the M extension, so we only had to implement an ALU and a MUL/DIV unit. The ALU was a very simple implementation, almost completely taken from MP Pipeline. Extra FSM logic was added on the frontend to notify reservation stations when the ALU unit was ready and to notify the writeback/CDB stage when the ALU unit had a valid instruction to broadcast. The MUL/DIV unit was a little bit more involved as we had to use the multiplier and divider IPs from Synopsys. The FSM for this unit was very similar to the ALU FSM. The Synopsys IPs required the frontend to handle some of the parameters that went into the IPs themselves such as instruction hold. There was also extra logic to wait for the result to be outputted by the IPs as they did not reply in one cycle like the ALU did. Testing these units involved passing in placeholder values and ensuring that the right operation was being performed with the correct result being passed out of the unit. A significant amount of testing also came from writing test code using which functionality was tested through Spike.

The decode stage is fully combinational, and the dispatch stage is also combinational in nature, allowing immediate decoding and register renaming upon instruction cache dequeue. The dispatch stage works with the instruction queue implemented in checkpoint 1 to accept an instruction, rename and package the values encoded in the instruction in a format accepted by the reservation stations, and dispatch the instruction. Dispatch occurs only when both the ROB and the appropriate reservation station have available entries. The register file is based on the mp_pipeline design but includes an integrated RAT for managing register renaming. To prevent instruction stalls at the decode stage due to busy functional units, we implemented reservation stations that use an age-based priority mechanism to ensure that the oldest ready instruction is issued first.

In summary, our design follows Tomasulo's algorithm and we optimized our processor to the lowest possible CPI we can achieve at this stage. To support future tuning and scalability, we parameterized major modules such as the ROB, reservation stations, and functional units.

iii. Checkpoint 3

In Checkpoint 3, we completed the final components necessary to fully support execution of the RV32IM specification. The modules developed and refined during this phase included the memory unit, load-store queue, control instruction unit, instruction/data cache arbiter, comparison module, and additional logic for flushes and PC control. Much of the new logic introduced in this checkpoint was adapted from the mp_pipeline, with modifications to accommodate out-of-order execution.

The cache arbiter (cache_arbiter.sv) serves to arbitrate requests coming from the caches going to bmem. We implemented this via a state machine with states IDLE and WAIT_FOR_RESP. In IDLE, we wait for requests from either cache, and upon receiving them, we immediately route the signals from the cache to bmem and move to state WAIT_FOR_RESP. It is here that we hold the signals from the corresponding cache connected to bmem, and wait for a response from bmem. Upon receiving that response, we give the response to the cache and move back to the idle state. At first we used a round robin approach to prioritize which cache got access upon contention, but during advanced features we reverted it to prioritizing the icache since we observed that the icache latency had more of an effect on ipc. For testing we wrote a testbench (cache_arbiter_tb.sv) which tests reads, writes, and contending operations.

The load-store queue (memory_reservation_station.sv) has all the responsibilities of a reservation station except it functions as a queue. Since, at this point, our reservation stations were age ordered and already functioned as queues, we simply copied the code. However, this naive implementation of looping through every entry and keeping track of an age counter created a very long critical path, so for the split load store queue we implemented collapsing queues instead. We already tested our reservation stations, so since we simply copied the code from those we didn't write any explicit testbench for this. We simply ran the benchmarks and verified that it worked.

The memory unit was largely taken from MP pipeline. New FSM logic was added to support the ability to wait for responses from the data cache after a read/write request was passed in. This FSM dictated how the memory unit indicated to the load store queue about when it was ready to accept a new instruction to execute. Testing this unit included writing a testbench (mem_unit_tb.sv) and passing in example values to then check for correct outputs to data memory and out of the memory unit.

To improve control flow handling, we separated comparison logic from the ALU into a dedicated comparator module. To address branch mispredictions, we introduced a flush signal coming from the ROB that propagates through almost every module to clear registers and to restart the pipeline. Additionally, we implemented a PC mux that selects between PC + 4 and the branch target based on the outcome of control instructions. We used a static taken prediction for all branches in this checkpoint as we only had to meet basic functionality requirements. We

experimented with both static taken and not-taken approaches and found that static taken led to marginal improvements. This did mean that we encountered a lot of flushes, but we decided to handle that in advanced features.

The control instruction unit simultaneously calculates the target PC via adders and the branch outcome via a comparator. Like the other functional units, these results are then broadcasted on the CDB, and if the CDB is occupied, this module holds the outputs until they are confirmed as written back. The logic for this unit was also largely taken from MP pipeline.

In summary, this checkpoint finalized our processor's functionality by incorporating memory access support and control instruction handling, along with thorough testing of these features. We now had a low performance, but working out-of-order processor!

Advanced Design Features

i. Split Load Store Queue

A. Design

The load queue and store queue were implemented via collapsing queues. We chose this design since we knew we'd have to issue the oldest ready entry, and we wanted to avoid a long critical path when issuing. To avoid data hazards where loads are dependent on earlier stores, we only issue loads that are older than the oldest uncommitted store. To do this we have an arbiter that arbitrates between load-queue issues and store-queue issues. The arbiter first checks if the entry at the head of the store queue is ready and if the ROB head points to a store instruction. Since we only issue stores if said conditions are true, and since the entry at the head of the store queue is always the oldest uncommitted store, we know that this entry must correspond to the instruction at the ROB head. If those conditions are true, then the arbiter sends the store to the memory unit and the store queue collapses by moving every entry one space toward the head. If said conditions aren't true, then the arbiter tries to issue a load. The load queue will loop through its entries starting at the head until it reaches a load with its operands ready. This will be the oldest ready load in the queue. During dispatch, we've attached an age to every memory instruction. With that addition, the arbiter compares the age field of the ready entry given by the load queue to the age field of the oldest uncommitted store (the store at the head of the store queue), and if the load is older than the store, the load is issued and the load queue collapses. By issuing the oldest load with its operands ready, we allow loads to execute out of order in between stores.

B. Testing

We didn't run into any problems testing the split queue. We created some testcode called split_lsq_test.s that executes some loads out of order in between stores. A lot of the testing was done using the provided benchmarks while observing the performance improvements and specific observation parameters mentioned in the next section.

C. Performance Analysis

All performance changes are relative to the branch cp3_baseline_final. We noticed IPC losses for AES and Compression, and for a long time we couldn't figure out why. Then we realized that the number of cycles the icache spent allocating went up significantly, likely due to memory contention from increased inflight memory instructions and due to the unique instruction ordering of these benchmarks. We will discuss this more in section 4.

	Coremark	AES	Compression	FFT	Mergesort
IPC change	+0.002216	-0.033569	-0.002833	+0.013413	+0.063541
Change in # of cycles LSQ was full compared to cycles either the load queue or store queue was full	-1082	-179438	+4310	-104474	-17777
Change in # of cycles ROB was stalled due to mem instruction	-39771	-126045	+5186	-74378	-33608

Table 1: Performance Impact of Split LSQ

ii. Age-Ordered Reservation Stations

A. Design

For these we simply implemented collapsing queues. We loop through each entry starting at the queue head, and if an entry has both its operands ready and the corresponding functional unit is free, we issue that entry and break from the loop.

B. Testing

We had already implemented collapsing queues with our split load-store queue, so we just copied the code from that. There was no need to test further, our IPC gains showed that it was working.

C. Performance Analysis

All performance changes are relative to the branch cp3_baseline_final. Age-ordered reservation stations allowed older instructions to get issued before newer ones, changing the timing that operands for future instructions were ready which overall increased IPC across the board.

	Coremark	AES	Compression	FFT	Mergesort
IPC change	+0.009812	+0.007142	+0.072698	+0.012042	+0.021482

Table 2: Performance Impact of Age-Ordered Reservation Stations

iii. Design Space Exploration Script

A. Design

When designing the Design Space Exploration script, our initial choice was Bayesian Optimization. We selected this method because running simulations is extremely time-consuming, and Bayesian Optimization is well-suited for problems where evaluations are expensive. We chose Python for two main reasons: it offers built-in support for Bayesian Optimization through various libraries, and it is generally more user-friendly. Given that our processor already performs well in surpassing the baseline IPC but suffers from high delay, we prioritized optimizing delay while placing less emphasis on IPC. To reflect this focus, we formulated the objective function as:

Objective = -(IPC/Baseline IPC) + 2 * (Delay/Baseline Delay)

This normalization ensures both metrics are on a comparable scale. By assigning a higher weight to delay, the function emphasizes reducing delay as a priority. The optimizer is then tasked with minimizing the overall objective score. Our Python script successfully modified the selected parameters in cpu.sv and recorded the corresponding IPC and delay values for each parameter set. Using these evaluation functions, we implemented Bayesian Optimization and gained preliminary insight into the parameter ranges that yield better performance across different testbenches.

Although Bayesian Optimization is effective for expensive evaluations, it struggles with high-dimensional search spaces. Therefore, we chose to implement the Genetic Algorithm as the second optimization method. Since the evaluation functions were already in place from the Bayesian Optimization setup, adapting the script for GA required only a few additional components such as functions for crossover, mutation, population generation, and tournament selection.

B. Testing

We began by testing our script through individual component verification. Specifically, we confirmed that the parameter modification function correctly updated the parameters in cpu.sv, that the run_simulation function successfully executed the simulation and accurately retrieved IPC and delay upon completion, and that both the parameters and results were properly logged. These tests were straightforward, as the effects of parameter modifications could be directly observed, and simulation logs provided clear feedback.

Subsequently, we evaluated the integration of these components by employing the built-in Bayesian Optimization library. During this phase, simulation messages were suppressed in the console to facilitate clear observation of the Bayesian Optimization output. To expedite testing, the n_calls parameter was set to its minimum value of 10.

Following successful validation with Bayesian Optimization, we proceeded to develop the Genetic Algorithm , applying the same systematic approach: testing each component independently before conducting integrated tests of the complete GA implementation.

C. Performance Analysis

During the analysis, we set the parameter ranges between 4 and 32, except for ROB depth, which is limited to 64. We chose a wider range for ROB depth because we believe a larger ROB often offers a larger instruction window, and most benchmarks can benefit from this. For the Genetic Algorithm parameters, we chose a population size of 10, generation size of 5, mutation rate of 0.5, and tournament size of 3. We selected relatively small

population and generation sizes because the evaluation function is very expensive to run, even with simulations executed in parallel. The mutation rate is set high primarily to encourage exploration of a wider range of parameter combinations and to better cover the search space, especially given the relatively small generation and population sizes. Similarly, we perform crossover every time a new individual is created to further promote diversity. The optimization result is also modeled using Python's built-in Random Forest Regression algorithm to measure the importance of parameters for each testbench. Table 3 and Figure 2 below present the optimization results and parameter importance plots. We provide several observations along with corresponding interpretations.

For FFT, we think that the relatively large ROB size and higher importance corresponds to the potentially long dependency chain due to the recursive nature of the FFT testbench. While not reaching the maximum possible Reservation Station size, a moderate RS size and high importance indicates that functional unit scheduling is important. The moderate importance of store queue depth may imply that the store queue helps hide memory latencies but doesn't dominate performance here.

In comparison, the optimization results and the parameter importance plot for the mergesort testbench show an even higher importance assigned to ROB depth, yet the optimal value for this parameter is only 6. To better understand this outcome, we plotted the partial dependence of ROB depth alongside two other parameters, as shown in Figure 3. The steep increase in performance effect up to a ROB depth of 10, followed by a less steep increase up slightly above 20, and a flat line thereafter, contrasts sharply with the flatter trends of the other parameters on the right, justifying why the model attributed the highest importance to ROB depth.

We believe this diminishing return is due to mergesort being more memory-bound, whereas FFT is more compute-bound. With numerous load/store operations and data-dependent branches, mergesort puts significant pressure on the load/store unit. If memory operations become a bottleneck, the ROB may be unable to retire instructions efficiently, limiting the benefit of increasing its size further and resulting in the observed flat line in performance gains.

For the compression workload, the regression model identifies the IQ size as the most important parameter, with the optimal size found to be 16. This result aligns well with the nature of the workload. Due to low data reuse and large loops with data-dependent accesses, a moderately sized IQ may be helpful in balancing buffering enough instructions to keep the pipeline fed while avoiding excessive resource overhead.

	ROB_DEPTH	IQ_SIZE	RS_DEPTH	LQ_DEPTH	SQ_DEPTH	HISTORY_WIDTH	PHT_IDX_WIDTH	IPC	Delay
Coremark	15	32	7	12	7	5	13	0.532	5.480ms
AES	14	12	16	28	9	30	16	0.497	15.597ms
Compression	6	16	13	28	31	7	13	0.701	5.990ms
FFT	42	8	18	19	28	14	15	0.682	17.034ms
Mergesort	6	26	21	28	24	12	14	0.537	8.968ms

Table 3: DSE Script Optimization Result

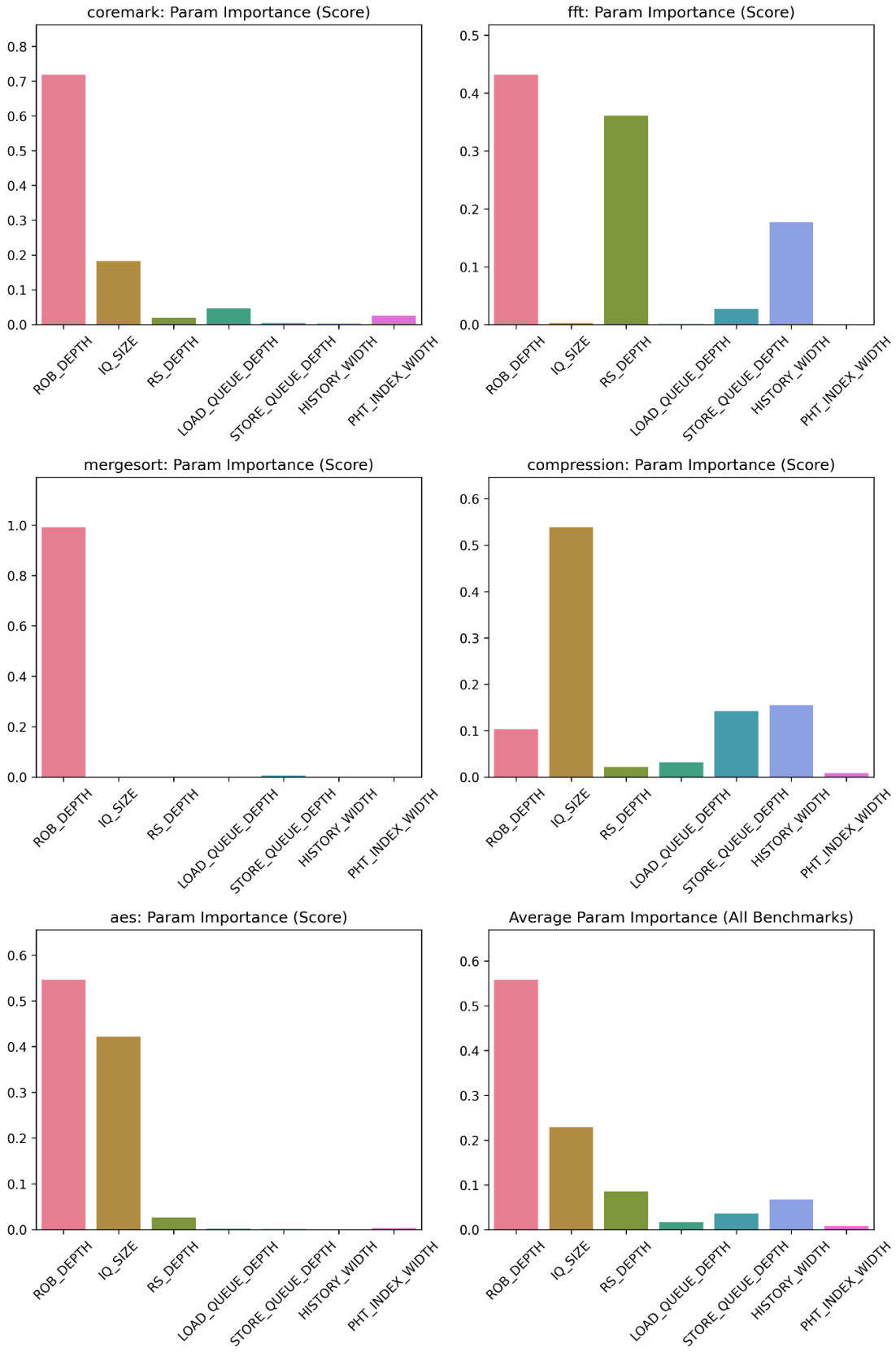

Figure 2: Optimization Result Visualization with Random Forest Regression Model

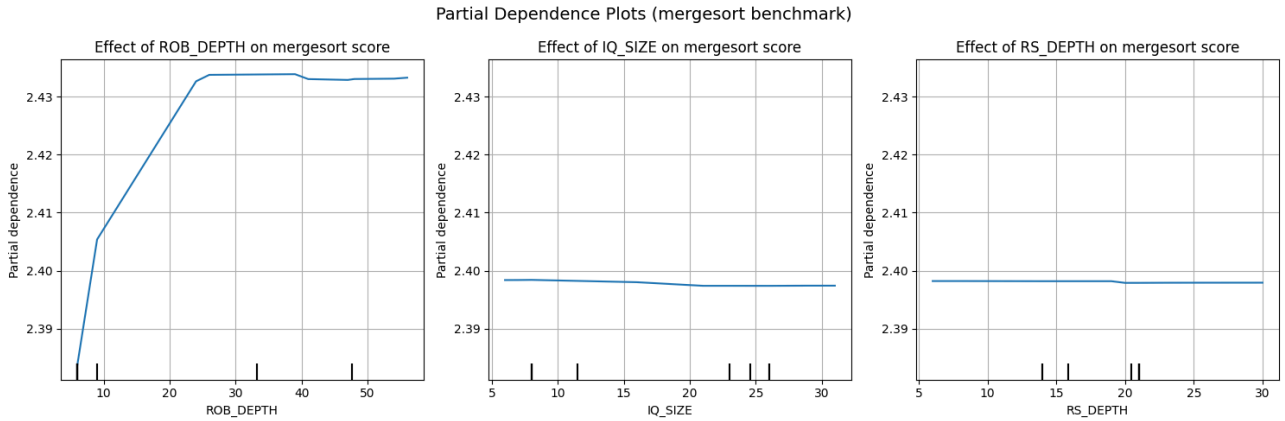

Figure 3: Partial Importance Plot For Mergesort

iv. G-Share Branch Predictor

A. Design

The G-Share branch predictor has two main modules, the Global History Register (GHR) and the Pattern History Table (PHT). The GHR was implemented as a basic configurable N-bit (N=6) shift register and captured the outcome of the last N retired branch entries. It was made parametrizable and had a combinational "history" being output for indexing. The PHT was implemented using a single-port SRAM macro from OpenRAM, indexed by doing an XOR operation between the "history" from GHR and the low N bits (same as GHR width) from the PC. The PHT entries were all initialized to a weakly taken prediction (because we noticed a better prediction percentage with static taken in CP3). Since there was no way to initialize the SRAM or find if the value at a given index was valid, we included a valid array in the PHT. The branch predictor module (branch_predictor.sv) handles the calculations of the PHT index, committing to the GHR on a branch retirement, and has logic to arbitrate between simultaneous branch commits and prediction requests. At fetch time, the branch prediction, the PHT index of the corresponding branch instruction, and type of prediction (strong/weak, taken/not-taken) is stored in the ROB for use when the branch commits.

Extra logic was also needed to account for the read response delay from the SRAM within the PHT. Since the SRAM only responds to a read request the next cycle, we delay the fetch stage by one cycle every time a branch instruction is encountered. While this adds an extra cycle every time a branch instruction is fetched, it is significantly better than the multiple cycles wasted on a flush caused by a branch misprediction in CP3.

B. Testing

Testing for the branch predictor first included unit tests for the individual components making up the predictor. The GHR was tested by shifting arbitrary values into the register and ensuring that the history being passed out matched. The PHT tests included tests of the FSM that determined the branch prediction (weak/strong, taken/not-taken) and testing the SRAM itself. The PHT tests created example branch instructions and simulated the lifecycle of that instruction from fetch to SRAM access to commit. Testing the branch predictor as a whole included testing for edge cases like a branch commit and a branch fetch at the same time. A large portion of the testing also came from writing custom assembly test code to run on the overall CPU to target branch instructions and observe functionality through Spike.

C. Performance Analysis

Observing performance for the branch predictor was very straightforward. We included two variables, num_correct_br_preds and num_branches, that track the number of correct branch predictions and number of committed branch instructions respectively over the lifetime of a benchmark being run. A branch prediction was considered correct if, at the time of commit, a branch operation was detected without a flush being issued. These values were then used to calculate the percentage of branch instructions that were predicted correctly. The below tables show these values observed (along with IPC) for five different benchmarks before (Table 4) and after (Table 5) the branch predictor was used.

Benchmark	**Number of committed branches**	**Number of correct branch predictions**	**Correct prediction percentage**	**IPC**
Coremark	55170	29516	53.5%	0.431
AES_SHA	11417	6769	59.2%	0.432
Mergesort	79877	28925	36.2%	0.431
FFT	25085	18945	75.5%	0.537
Compression	50567	35563	70.3%	0.589

Table 4: Performance analysis with static taken

Benchmark	**Number of committed branches**	**Number of correct branch predictions**	**Correct prediction percentage**	**IPC**
Coremark	55170	44708	81.0%	0.470
AES_SHA	11417	9570	83.8%	0.436
Mergesort	79877	60481	75.7%	0.464
FFT	25085	22519	89.7%	0.541
Compression	50567	50552	99.9%	0.638

Table 5: Performance analysis with G-Share

We made a few observations after performance testing the branch predictor. First, the actual benchmark we ran made a big difference in the branch accuracy and the IPC. As can be seen from Table 4, while Coremark and Mergesort have relatively low branch prediction accuracies, the accuracies for FFT and Compression were already in the 70's without the predictor. This we believe could be because of a high number of branch instructions evaluating to be taken after execution in those benchmarks (as we had a static taken approach initially). Second, the branch predictor unlocked the potential of other advanced features as those features were being bottlenecked by the

sheer amount of flushes being encountered without an accurate predictor. Once we combined the predictor with other advanced features, the other features were able to provide that much extra improvement in performance.

Additional Discussion / Observations

When comparing the split load-store queue and the age-ordered reservation stations to checkpoint 3, we had to undo some things in our baseline processor. As a result, we created a new branch called cp3_baseline_final. Our final split load-store queue is in branch split_load_store_queue_final, and our final age-ordered reservation station is in branch age_ordered_rs_final. These are the branches we got our metrics from to compare to cp3_baseline_final. All metrics have been pre-calculated with results in a file called performance.txt in each branch.

As mentioned previously, we saw IPC losses from the split load-store queue for AES and compression. This appeared to be due to the number of cycles icache was spending allocating, which went up significantly for those benchmarks. Even after prioritizing the icache in the cache arbiter, we couldn't get this number down significantly. However, when we combined the split queue with our branch predictor, this number went down tremendously due to the reduced number of speculative icache block allocations, and the benefits of the split queue started outweighing the costs for those benchmarks.

Contributions

By the end of checkpoint 3, Elliot (edziem2) had completed the caches, the cacheline adapter, the reservation stations, the cdb arbiter, the commit stage, the load-store queue, and the cache arbiter. For advanced features, he completed the split load-store queue and the age-ordered reservation stations. Prakatish (pk31) completed the instruction fetch, cache with linebuffer, the decode stage, the dispatch stage, the ROB, register file and RAT, the CDB, RVFI integration, the control unit, the memory unit, and the PC mux. For advanced features Prakatish (pk31) worked on the branch predictor. Xuhang(xuhangx2) contributed to the circular queue, rob, flush logic, as well as some of the debugging for the mul/div functional unit. For advanced features, he completed the design space exploration script and performed data visualization on the optimization result. Dhruv (dhruvp4) completed the ALU and MUL/DIV functional units and the comparator unit.

Conclusion

Our design objectives were to design a functionally complete Tomasulo's style out-of-order processor supporting the RISC-V instruction set with the M extension and to optimize its performance as best we could. By the end, we hit our objectives and achieved a high number of instructions per cycle (IPC) for all benchmarks, beating their baseline performance by as much as 0.2 IPC in the case of compression. We did miss out on the PD^4 benchmark requirements, which is unfortunate, but we tried our absolute best. Overall, we learned a great deal from this project and are proud of the work we did. It was not perfect, but none of us could make a functioning out of order processor before this class, and now we can!