Albert Jiang
Taowei Huang
Varun Pal
May 11, 2026

ECE 411 mp_ooo report

Introduction

In this MP, we created an Out-of-Order processor supporting the RV32IM ISA except fencing and csr instructions. An Out-of-Order processor allows instructions to be executed in a different order than they were written. However, this does not mean instructions commit out of order. From the programmer's perspective, all instructions still complete in program order. In a non Out-of-Order processor design, instructions that take many cycles such as load, store, multiply, and divide will cause the processor to stall while waiting for them to complete. By processing instructions out of program order, the CPU can continue executing instructions that are ready instead of sitting idle waiting on slow or register dependent instructions. We divide our Out-of-Order processor into 7 stages: PC, Fetch, Decode, Rename, Dispatch, Execute, and Commit. The primary goal of this project was to gain hands-on experience with the architectural techniques used in modern high-performance processors by implementing a functional Out-of-Order CPU from the ground up.

Project Overview

In this report we will discuss our micro architecture, design decisions, and performance. We chose to implement Explicit Register Renaming (ERR) to execute instructions out of order because it seemed like a more modular approach compared to Tomasulo's Algorithm which offloaded much of the complexity to the ROB to keep track of operand values and their readiness. This allowed us to be more flexible with what advanced features we could implement in the last stage of our design although it added more complexity with a physical register file (PRF), free register list, and a rename stage. Our work flow generally was to implement the data structures first like rat, rob, free list, reservation station, and then hook them up to each stage of the pipeline. During checkpoints 1 to 3 we worked closely together using vscode liveshare to integrate the modules and types together to get a baseline ooo cpu. For advanced features, we worked individually off the baseline and combined them together.

Design Description

Overview

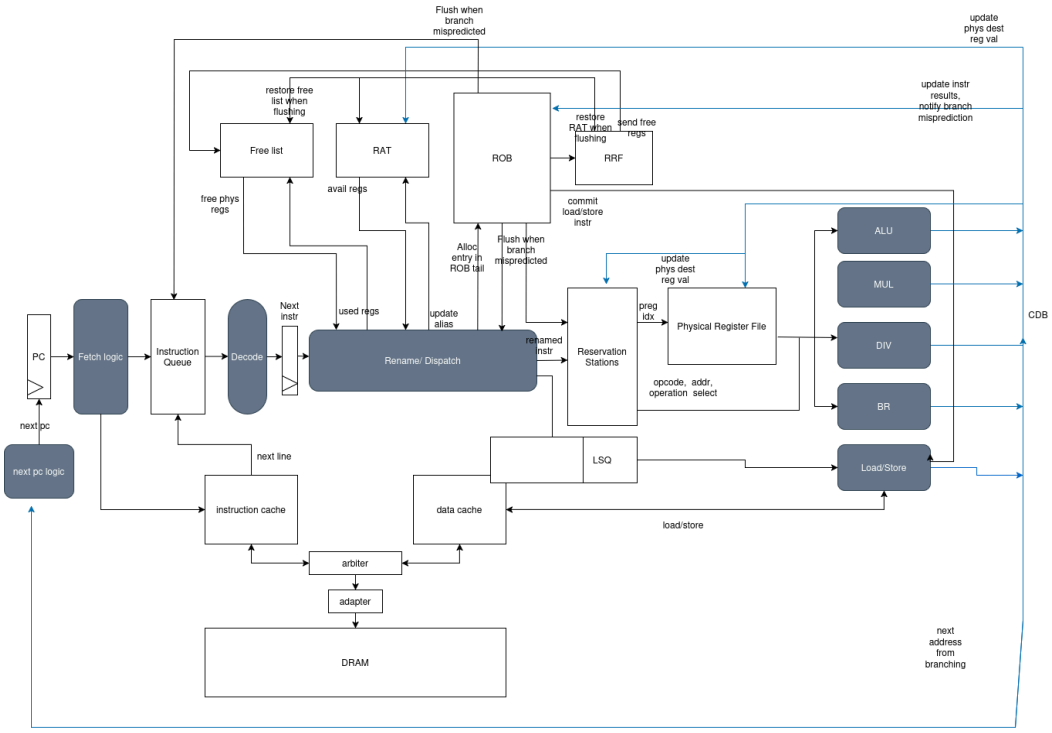
The design of the Out-Of-Order is splitted into 4 parts. In the first checkpoint, we designed the cpu and implemented the Fetch stage. In the second checkpoint, we implemented all instructions except branch and memory instructions. The implemented instructions can execute out-of-order. In the third checkpoint, we implement branch and memory instructions. In the fourth and final checkpoint, we implemented split LSQ, 2-way super-scalar, and branch predictor into our cpu. We also analyzed the performance of our design against a baseline cpu on 6 benchmark programs.

Milestones

Checkpoint 1

In this checkpoint, we implemented the pipeline pc stage to the instruction queue. We finished a parametrizable queue, cache line adapter, a linebuffer for instruction cache, and fetch stage logic. We also design a diagram for the rest of the processor shown below.

Figure 1: Diagram of Initial Micro Architecture



We use the same cache as the one made in mp_cache for instruction cache. However, this cache uses a state machine design where even with always cache hit the maximum throughput is 0.5 instruction per cycle. Therefore, we use the linebuffer to store one cacheline at a time to achieve maximum 1 instruction per cycle throughput when it's cache hit. Each cacheline is 256 bits (8 instructions), the DRAM responds to a request in 4 bursts of 64 bits. The cache line adapter collects the 4 bursts from DRAM and presents the 256 bits data to the cache. By the end of this checkpoint we could fetch instructions into the instruction queue. To verify this worked, we made the I-Queue size of 8, fetched until the queue was full which triggered a front end stall stopping all I-Cache requests to DRAM, and looked at Verdi waveform signals to observe the behavior.

Checkpoint 2

For Checkpoint 2, we completed the execution back-end so we could execute all i and r type instructions excluding branches, jumps, memory operations and auipc. The rename stage reads the Register Alias Table (RAT) to map architectural source registers to physical ones and pops the free list to allocate a new physical destination register for each instruction. The renamed instruction is then passed to dispatch, which simultaneously writes an entry into the Reorder Buffer (ROB) and the appropriate reservation station. The ROB operates as a circular queue and dispatch enqueues at the tail and commit dequeues from the head. Each ROB entry stores the architectural and physical destination registers, the full instruction, and the RVFI monitor state needed at commit time.

We decided that each functional unit (ALU, MUL, DIV) has its own dedicated reservation station which is modular and supports more parallelism with a superscalar. Entries sit in the reservation station until both source operands are marked ready, either at write time or by snooping the Common Data Bus (CDB) on subsequent cycles. Once an entry is ready, it is issued to its functional unit. The functional unit latches the reservation station entry into an internal register for one cycle before execution. This was done both for timing and to preserve bookkeeping data such as the ROB index and RVFI signals that are needed at writeback. For the MUL and DIV units, we used a 10 cycle Synopsys DW_mult_seq and 20 cycle DW_div_seq sequential IPs for simplicity in terms of area and combinational logic.

Each functional unit drives its own dedicated CDB line. The CDB result is broadcast simultaneously to the RAT (to mark the destination register ready), all reservation stations (to wake up waiting entries), the ROB (to mark the completing entry as ready), and the physical register file (to write the result value). We chose to have multiple CDBs because it greatly simplified our rtl but it was at the expense of combinational area due to fanout since each cbd carries 434 bits.

The commit stage retires instructions in-order from the ROB head. When the head entry's ready bit is set, the commit stage dequeues it, outputs the RVFI signals for Spike comparison, and uses the Reverse RAT (RRAT) to identify and return the previously-mapped physical register back to the free list, making it available for future renaming.

Verification: To verify our processor was executing out of order, we used Verdi to see later alu instructions completing in ROB before subsequent mul or div instructions finished. Additionally we made testbenches to unit test our functional units, and queue data structure.

Checkpoint 3

In this checkpoint, we completed the memory and branch functional units needed to support the RV32IM ISA. The branch functional unit has its own reservation station and computes both control-flow decisions and branch targets. For conditional branches, it compares the source operands using the instruction’s funct3 field, while for jumps it computes the target PC, and for auipc it computes the writeback value without causing a control-flow redirect. The branch FU broadcasts the ROB index, destination physical register information, computed writeback value for link or auipc instructions, br_taken, and br_pc on the CDB. The ROB stores the br_taken and br_pc values from the CDB result, and later, when a taken branch or jump reaches the head of the ROB, the commit stage asserts flush and drives next_pc to the recorded target. This clears speculative frontend and backend state so execution restarts from the correct PC.

For the memory unit, we designed a split LSQ that can support out-of-order load execution in the presence of older stores. If a load has ready source operands and all older stores that could affect it have been resolved, then that load can execute before earlier loads that are still waiting on operands. To support this, loads are tagged with store-age information so the unit can determine which stores are older than each load, and we used a one-hot style encoding for area and comparison efficiency. The full split-LSQ design is described in the advanced features section. In the CP3 version, however, memory instructions still passed through a FIFO reservation station after dispatch, so the LSQ was more conservative than the final split design. This ordering helped us reliably pass the baseline while avoiding unresolved edge cases in the more aggressive out-of-order memory path.

Additionally we had to add arbitration between I-Cache and D-Cache memory requests to the dram adapter. We made a simple FSM based arbiter that acts as a switch between the 2 cache ports. If both I-Cache and D-Cache make a request at the same time we service the I-Cache request, then service the D-Cache request. If a request comes while DRAM is still responding we latch the request and service it next.

By the end of CP3 we were able to synthesize at 526MHz with an area of 200K. The following are performance metrics of the baseline. To verify our design we mainly relied on the 6 provided programs and targeted programs to stress test our cpu when common data structures like rob, reservation stations, and instruction queue filled up.

Program	IPC	Delay (ms)	Power (mW)	PD4
coremark	0.309516	1.507194	36.223	186.922
aes_sha	0.298736	4.155827	35.746	10662.457
compression	0.408127	1.685797	36.365	293.701
fft	0.381183	4.873307	36.085	20352.689
mergesort	0.393158	1.960925	36.181	534.964
image	0.310749	2.435317	36.176	1272.455
Geometric mean	0.3473863335	2.50271862	36.12882853	1417.431197

Table 1: Performance metrics of the baseline CPU across benchmark programs.

Structure	**Size**

Physical register file	64
Architectural registers	32
Free list	32
ROB	16
CDB channels	5
Reg-file read ports	10
Instruction queue depth	8
ALU/MUL/DIV/BR reservation station	4 each
LSQ reservation station (FIFO)	4
Store queue (STQ)	2
Load buffer (LDB)	4
MUL latency (DW NUM_CYC)	10
DIV latency (DW NUM_CYC)	10
I-cache / D-cache	16 sets x 4 ways x 256-bit line (2.0 kB each)

Table 2: Hardware structure sizes and latencies for the Out-of-Order processor.

Advanced Design Options

Option 1: Split LSQ

Design

The advanced design feature we implemented in the memory subsystem is a split load/store queue consisting LSQ reservation station (lsq_reservation_station.sv) feeding a load/store functional unit (ld_st.sv) that drives the data cache and broadcasts results on the CDB. The function unit is splitted into Load Buffer (LDB) and Store Queue (STQ), allowing out of order issue of load instructions between store instructions.

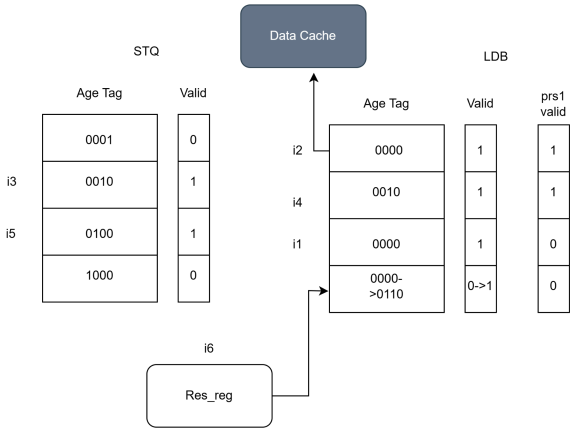
The LSQ reservation station is a parameterizable ring buffer that can enqueue instruction while dequeuing a single head entry when the LSQ is ready. It performs same-cycle CDB forwarding

so newly-ready operands are reflected immediately, but it intentionally does not wait for operands at the head. Entries can flow forward as soon as they are valid and LSQ is ready. This allows bigger storage for the load/store instruction and prevents non-ready instruction stalls in the queue.

The LSQ then handles the memory ordering and operand capture. Because the physical register file read has a one-cycle latency, it latches the dequeued RS entry and only enqueues into the LDB or STQ when the corresponding rs1_v/rs2_v values arrive. When enqueuing, loads are tagged with an age mask seeded from the current STQ occupancy so they wait behind all older stores, and stores carry a one-hot tag so that when a store retires, the unit can clear that store's bit from every in-flight load's age tag. This feature allows loads with the same age tag to fire out of order.

For example, in the figure below, instructions are numbered in chronological order. The incoming i6's age tag is the XOR sum of the age tags from valid stores i3 and i5. i1 is older than i2 and with the same age tag, but because i1 doesn't have its source register ready, i2 will be issued to data cache first. Later when i3 is fulfilled, i4's age tag will be cleared to 0000, and i5's will be 0100, maintaining the same relative ordering.

Figure 2: Diagram Age Tag in Split LSQ



Cache issue is single-outstanding with loads prioritized. LSQ selects the first load whose base operand is ready and whose age tag is zero, while stores issue strictly in order and are only issued when they are at the head of ROB. On a cache response, LSQ produces a CDB broadcast carrying either a sign/zero-extended load value (LB/LBU/LH/LHU/LW) or a store completion notification, along with the monitor's memory address/masks/data, so the rest of the out-of-order core can receive value properly and commit instructions in order.

Testing

We added the Split LSQ to the CP3 version of CPU and ran all six provided testbenches at 625Mhz.

Performance analysis

Program	IPC	Delay (ms)	Power (mW)	PD4
coremark	0.314175	1.484845	37.213	180.892
aes_sha	0.308279	4.027179	36.850	9692.621
compression	0.424132	1.622181	37.354	258.663
fft	0.382154	4.860926	36.971	20641.309
mergesort	0.403361	1.911326	37.265	497.325
image	0.312104	2.424744	37.115	1282.960
Geometric mean	0.3542485184	2.454238921	37.12759742	1346.989646

Table 3: Performance analysis of the Split LSQ advanced design option.

As the chart above shows, Split LSQ provides consistent improvement on IPC, while allowing us to synthesize at the same frequency. The biggest increase is at compression (from 0.408 to 0.424). Similarly, because of the additional logic, powers are universally increased by around 1mW. Overall, PD4s all decreased by various amounts. Therefore we decided to use this feature for our final version.

Option 2: 2-way super scalar

Design

The advanced design feature we implemented is a 2-wide superscalar front-end and middle pipeline. From fetch through dispatch, all stages operate on pairs of instructions simultaneously. The PC advances by 8 bytes per cycle, fetching two instructions at a time, and decode and rename process both instructions each cycle. When a situation arises where we cannot supply two valid instructions. For example, at a branch boundary or when only one instruction is available from the linebuffer, we insert a bubble in the second slot. The bubble is still enqueued into the ROB as a pre-ready entry with its monitor valid bit cleared, so commit can retire it silently without producing any RVFI output; it is never written into a reservation station.

A key challenge in 2-wide rename is intra-group forwarding. Since both instructions are renamed in the same cycle, the RAT has not yet been updated with slot 0's destination when slot 1 reads its sources. If slot 1's source register matches slot 0's destination, the rename stage detects this and directly forwards slot 0's newly allocated physical register to slot 1's source mapping.

All structural resources were extended to support 2-wide operation. The free list, ROB, and reservation stations all support enqueueing and dequeueing two entries per cycle. A particular challenge with the free list was handling bubbles. Finally, the commit stage was extended to retire two ROB entries per cycle, allowing the full throughput benefit of the 2-wide pipeline to be realized at commit time.

Testing

To verify our design we use the 6 provided benchmark programs for performance testing. We also made testbenches for the queue and reservation station data structures to ensure we could support 2 instructions enqueuing and dequeuing.

Performance analysis

Program	IPC	Delay (ms)	Power (mW)	PD4
coremark	0.330537	1.411341	36.181	143.552
aes_sha	0.315108	3.939899	36.181	8718.081
compression	0.454745	1.512979	36.181	189.589
fft	0.413894	4.48815	36.181	14680.810
mergesort	0.442708	1.741450	36.181	332.755
image	0.326233	2.319731	36.181	1047.686
Geometric mean	0.376246	2.186523	36.181	11330.138

Table 4: Performance metrics for the 2-way superscalar implementation.

While the superscalar did provide modest improvements, we expected greater results closer to a 1.5 - 1.75 times multiplier on ipc. We realized through profiling that repeated taken branches and flushes throughout the pipeline greatly hindered throughput.

Program	Baseline IPC	Superscalar IPC	IPC Gain	Flushes
coremark	0.309516	0.330537	+6.79%	34,822
aes_sha	0.298736	0.315108	+5.48%	20,857
compression	0.408127	0.454745	+11.42%	40,570
fft	0.381183	0.413894	+8.58%	27,132
mergesort	0.393158	0.442708	+12.60%	61,804
image	0.310749	0.326233	+4.98%	47,127
geometric mean	0.347386	0.376246	+8.31%	-

Table 5: Comparison of IPC gains and flush counts between baseline and superscalar designs.

Although the superscalar design improves geometric mean IPC from 0.347 to 0.376, the overall gain is modest because frequent branch flushes limit how much useful two-wide work can be preserved. Every taken branch or jump that reaches commit assert flush, discarding younger speculative instructions and forcing the frontend/backend to restart from the redirected PC. This reduces the benefit of fetching, decoding, dispatching, and executing multiple instructions per cycle, since some of that extra in-flight work is later squashed. With the exception of mergesort, programs with high flush count like image see less gains than lower flush count programs like aes_sha and fft.

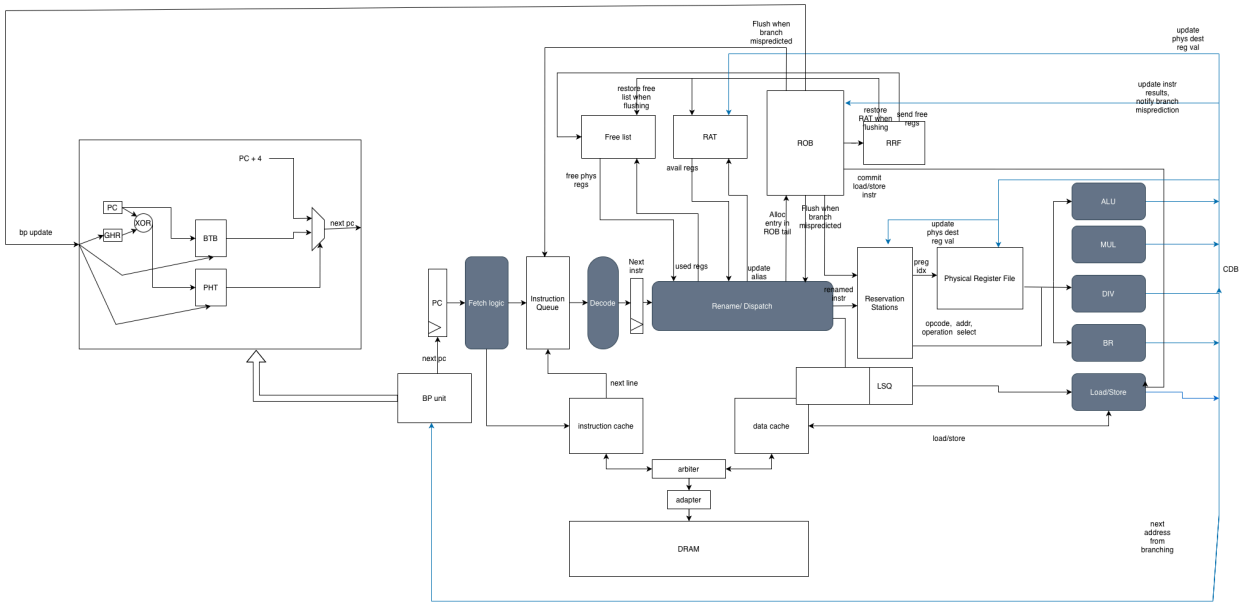
Option 3: Branch predictor with branch target buffer

Design

We design our branch predictor as a single unit in the PC stage. The branch predictor consists of a pattern history table (PHT) of size 2^6 with an entry size of 2 bits, a 6 bit global history register, and a branch target buffer (BTB) of size 2^6 with an entry size of 32 bits for branch address. All three types of branch instructions, JAL, JALR and BR, inquire from the branch

predictor unit for both taken/not-taken and address. The 32 bits of pc address are divided into 3 parts. Bit 0 and 1 are offset and not used since pc addresses are always 32 bits aligned. Bit 2 to 7 are XORed with the global history register and the result is then used as index bits into BTB and PHT. Bit 8 to 32 are used as tag bits for BTB. The diagram showing the branch prediction unit is shown in the figure below.

Figure 3: Diagram of the branch prediction unit architecture.



Testing

We use targeted RISC-V assembly codes with many JALR, JAL, and BR instructions to test for correctness. We then use the 6 provided benchmark programs for performance testing.

Performance analysis

We compare the performance of our branch predictor with the default static non-taken for branch instructions. The comparison is shown in the tables below. This correct predict rate is averaged across 6 benchmarks

Type	BP correct rate	Static non-taken correct rate
BR	81.3%	47.5%
JAL	32.84%	0%
JALR	40.11%	0%

Table 6: Branch predictor vs. static non-taken correct prediction rates.

The branch predictor significantly outperforms the static non-taken prediction across all benchmark programs. Our branch predictor resulted in an average ipc increase of 0.1 across the 6 benchmark programs.

A design choice that made our branch predictor not achieving as high of a branch prediction rate is that all the predicted pc addresses come from BTB. JAL and BR instructions use immediate values for address calculation. For those two types of instructions, PC calculation can be directly done in the PC stage with an additional adder. This will result in a higher predict rate since our current approach will always predict non-taken first during cold start. We implemented this design as well and noticed a higher IPC improvement. However, the critical path became the pc calculation and we had to decrease clock speed to meet timing. This resulted in slightly worse PD^4 despite higher IPC. Therefore, we decided to implement the current design which allowed a clock speed of 645 MHZ.

Option 4: Next-line Prefetcher

Design

We implemented a transparent next-line prefetcher that sits between the CPU's fetch stage and the instruction cache(I-Cache). It issues a speculative next line fetching request to I-Cache whenever the fetch stage is idle.

On a regular fetch request, the prefetcher will simply forward the request to I-Cache and record the requested PC. When the previous request is completed, the prefetcher will forward the response and immediately send out the speculative next-line request at the next time cycle. When the prefetching request is on the fly, the prefetcher will check whether the new incoming fetch request matches the prefetched line. If they match, the prefetcher will be able to respond immediately after the I-Cache response, allowing the fetch stage to receive data early. If they don't match, the prefetch will wait until I-Cache responds and sends out the actual fetch request. This means the penalty of an extra request to I-Cache on the wrong prefetched line.

Testing

We used 6 provided benchmark programs and counted the I-Cache miss with or without the prefetcher.

Performance Analysis

Program	Without	With	Difference

coremark	214841	231609	7.80%
aes_sha	1102485	1345729	22.06%
compression	115026	194924	69.46%
fft	114634	205947	79.66%
mergesort	205201	249778	21.72%
image	221091	477048	115.77%
Total	1973278	2705035	37.08%

Table 7: Instruction cache miss rates with and without the next-line prefetcher.

The prefecture shows a catastrophic increase of I-Cache miss across all benchmark programs, lowest being 7.8% and highest being 115.77%. This shows that the prefetching significantly changes the access pattern of I-Cache, leading to additional requests to DRAM. This is most likely due to misprediction of the speculative prefetching before the branch and jump instructions. The prefetcher blindly brings the next line into I-Cache, evicting the existing cache line. However, the fetched line will not be used for the next fetch. This penalty will be enlarged in a large loop reaching a wide range and consist of multiple jumps. The benefits of correctly fetching early fail to counter the penalty of the misprediction.

To check if prefetching could result in net benefits, we wrote a short program with mostly ALU and a few Branch instructions.

	Without	With
IPC	0.275219	0.306846
I-Cache Miss	2274	2356

Table 8: IPC and I-cache miss comparison for the prefetcher on a targeted assembly program.

As the table above shows, with prefetcher, even with more I-Cache miss, CPU managed to achieve a higher IPC benefiting from the fetching line early.
Because of the significant increase in I-Cache miss and decrease in IPC, we decided not to include this feature for our main branch for performance evaluation.

Option 5: Non-blocking Cache

Design

We implemented non-blocking data cache(D-Cache) for a single miss. It allows D-Cache to respond to another request while a request is on the fly. If the new coming request is a hit, D-Cache is able to respond to LSQ directly. If it is a miss, D-Cache will latch the request until

the previous request has been fulfilled. This allows D-Cache to respond to load/store instructions out of order, preventing a single miss to stall the D-Cache.

D-Cache uses two state machines to keep track of the requests: Miss-status holding register(MSHR) and Hit register. The Hit has three states: idle, busy, and wait. The MSHR uses three states: idle, allocate, and write back. This is meant to handle hit and miss separately. When Hit state is idle, D-Cache can accept a new load/store request. An acceptance fires a parallel read of all ways for data, tag, valid, dirty, and PLRU and performs a comparison and decode. If it's a hit, for load, D-Cache will return the selected word from the line; for store, D-Cache will merge the line and update the data and the dirty bit. PLRU is updated on hits for the access history. If MSHR is accessing the same set, Hit register will hold the request until MSHR resolves first to avoid race conditions.

On a miss, if the MSHR is free, the missed request is captured and changes the MSHR correspondingly. If evicting an existing line is necessary according to the dirty bit and PLRU, D-Cache will change into the write back state and send out a write request to DRAM. Then D-Cache will change to allocate state to send out a load request to DRAM for the current request. If MSHR is busy when a miss happens, the Hit state changes to wait state waiting for the previous request to be fulfilled.

To send load/store results back to LSQ, a tag is included in each request marking its position in the store queue and load buffer so that the response can match with the instruction. This modification also allows LSQ to handle multiple requests on the fly. When both MSHR and Hit have response ready, D-Cache will prioritize MSHR response to LSQ and latch Hit's for the next time cycle. On flush, D-Cache will clear out both state machines. If there is a request sent out to DRAM not responded yet, D-Cache will mark it and ignore the next upcoming response from DRAM.

Test

We test the performance of our non-blocking cache by running the provided benchmark program. We set up counters for the average response time for each load/store instruction. Additionally, we count the number of requests received by D-Cache while MSHR is handling a request already. Each request at this stage will be net benefits for IPC with more parallelism.

Performance and analysis

Program	Blocking	Non-blocking
coremark	48.01	42.94
aes_sha	42.04	36.35

compression	45.4	36.98
fft	44.44	36.27
mergesort	44.86	37.7
image	41.52	37.95
average	44.378	38.032

Table 9: Average service cycle reduction with non-blocking cache.

As the table shown above, non-blocking cache reduces 4 to 8 cycles for load/store instruction for each program, 6 cycles on average. This shows that non-blocking significantly reduces the service time for load/store instruction.

Program	Request while miss	Hit under miss	percentage
coremark	48	41	85.42%
aes_sha	2,896	2,521	87.05%
compression	156	156	100.00%
fft	109	72	66.06%
mergesort	272	254	93.38%
image	0	0	N/A

Table 10: Analysis of "Hit under miss" requests for the non-blocking cache.

The number of requests received while there is a miss and the number of those are hits are shown in the table above. The majority of requests received while serving another request is hit. The outlier is image which doesn't have this situation at all. However, the logic change in sending out and receiving requests still manages to reduce average response time by 4 cycles.

Unfortunately this change in logic also increases the critical path from CDB broadcast to LSQ and to D-Cache. This reduces the maximum frequency from 645MHz to 500MHz. With this constraint, even with better IPC, the PD^4 is slightly worse than the version without non-blocking cache. Therefore, we didn't include this feature in our final version.

	Best Run with 645MHz		Non-blocking cache with 500MHz	
	IPC	PD4	IPC	PD4
coremark	0.508	45.929	0.552	45.912
aes_sha	0.371	7895.194	0.394	10679.99

compression	0.71	56.825	0.834	52.237
fft	0.573	7069.822	0.663	6860.82
mergesort	0.502	358.848	0.56	405.834
image	0.38	1013.529	0.418	1130.261
Geometric mean	0.495	612.855	0.552	657.249

Table 11: Final performance comparison between the best run and non-blocking cache versions.

Additional observations

While our final implementation exceeds the performance of the baseline cpu, the IPC and PD^4 improvements are not as high as initially expected. Due to the limited time, we spend a significant portion of the time implementing advanced features. However, we did not spend as much time on parameter tuning. We saw good improvement on IPC and PD^4 by just increasing the size of reservation stations and load store buffers. Changing other parameters such as ROB size, Instruction queue size, Cache size and associativity, will also likely lead to further performance improvements. While we can achieve a relatively high frequency of 645 MHz, we are missing out the chance to add non-blocking cache to our CPU due to the critical path issue. We can try breaking down the logic path and tune the frequency to include non-blocking cache for better IPC and PD^4.

Conclusion

In this MP, we designed and implemented an Out-of-Order RV32IM processor to gain hands-on experience with the architectural techniques used in modern high-performance CPUs. Starting from a simple pipeline, we built a complete OoO backend with explicit register renaming, reservation stations, speculative execution, in-order commit, and advanced features such as a split LSQ, superscalar execution, and branch prediction. Through this project, we learned the tradeoffs between IPC, timing, area, and overall processor complexity while analyzing how different optimizations affected real performance.

Overall, this MP gave us a much deeper understanding of modern CPU architecture and the challenges behind building high-performance processors. This was one of the most challenging MPs we have worked on, but it was definitely also the cOoOlest.