MP_OOO: A Beautiful Tale

Introduction:

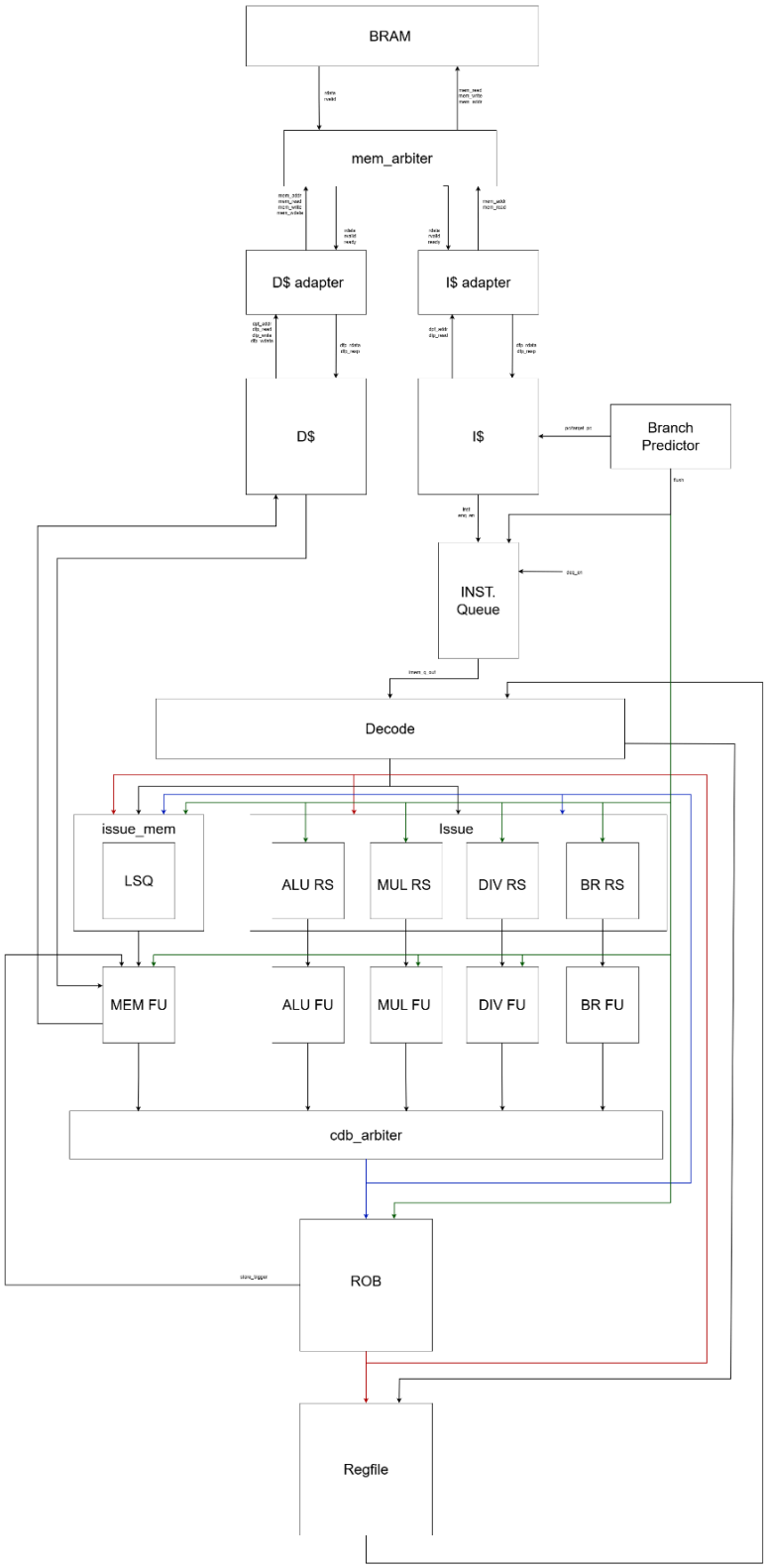
Our previous processor was a 5-stage pipelined in-order processor. This version improved throughput by overlapping instruction execution with the multiple stages, but it still required a strict program order execution. This means that any sort of memory miss or long operation would force any younger instruction in the pipeline to wait even if they were independent. This was not suitable for real workloads.

We needed to improve performance when we are stalling for long memory operations or just long latency operations. Enter out-of-order processing. Out-of-order execution allows the processor to continue executing ready instructions while waiting for slower operations like cache misses, multiplication, and division to complete. This is achieved by dynamically scheduling instructions based on operand availability and dependence. Furthermore, there are many improvements and features that can be added to further increase an out-of-order processor's efficiency to hide latency and exploit instruction level parallelism.

This report details the three stages of creating a baseline out-of-order processor and then advanced features that were added on to improve performance.

Milestone 1:

For milestone one, we were tasked with creating a block diagram of what our architecture was going to look like and then implementing the fetch stage. The design we chose to implement was a Tomasulo design due to the somewhat simpler architecture.


BRAM
mem_arbiter
D$ adapter
I$ adapter
D$
I$
Branch Predictor
INST. Queue
Decode
issue_mem
LSQ
Issue
ALU RS
MUL RS
DIV RS
BR RS
MEM FU
ALU FU
MUL FU
DIV FU
BR FU
cdb_arbiter
ROB
Regfile

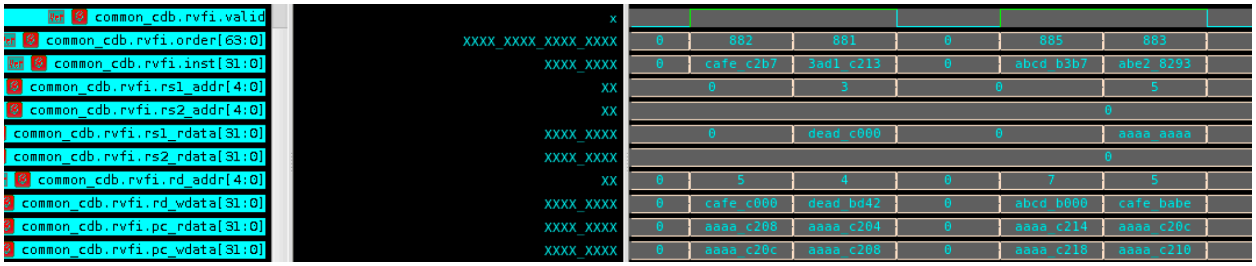
Milestone 2:

For milestone two, our objective was to implement all ALU instructions as well as multiply and divide operations. To support this, we implemented the ROB, decode stage, issue stage, reservation stations, register file, common data bus, and RVFI for verification. The reorder buffer (ROB) is implemented as a circular queue that ensures instructions are committed in program order, even though they may execute out of order.

Register renaming uses the ROB index as the destination tag, preventing WAR and WAW hazards by giving each instruction a unique write location. RAW hazards are handled by holding instructions in their respective reservation stations, where source operands are tracked as either valid values or ROB tags and are updated directly when matching results are broadcast on the common data bus.

The issue stage contains reservation stations for each functional unit and dispatches operations once all required operands are available. Execution results are broadcast on the common data bus and written back to the ROB, marking the corresponding entry as complete. When the head of the ROB is valid and complete, the instruction is committed by updating the architectural register file and driving the RVFI signals, ensuring correct in-order retirement.

Verification was performed using the provided ooo_test.s assembly program as well as a handful of our own. By inspecting the order of result broadcasts on the common data bus and comparing them to the original program order assigned at decode, we observed that instructions completed out of order while still committing in order, demonstrating correct out-of-order execution.



Milestone 3:

For milestone three, our objective was to implement the remaining essential functionality of the RISC-V ISA. To handle this, we implemented branching and memory operations. For memory operations we added the d-cache, loads and stores within our decode and issue stages, and a new memory functional unit. The d-cache (within the scope of this milestone) was implemented almost identically to how the i-cache was while just replacing the instruction cache parts with data from the cache. Meanwhile for branching functionality, there were similar edits to the decode and issue stages as well as a new branching functional unit. On top of that, branching functionality calls for flushing logic which we handled via added signals into every file. The flush is coded such that whenever we need to flush, any instruction that is younger than the mispredicted branch is evicted from any stage in the processor. For testing our milestone 3 additions, we used the benchmarks and were confident in our implementation if every single one passed without spike or RVFI errors. We finished off milestone 3 by recording our processors statistics for proper recordings of improvements during advanced features.

Advanced Features:

Branch Prediction:

The penalty for mispredicting a branch is one of the largest sources of performance degradation in modern processors. To reduce this penalty, we evaluated several branch

predictors, including static not taken, static taken, GShare, a two-level predictor with a local history table, and a tournament predictor selecting between GShare and the two-level predictor.

Our branch predictor is implemented as a four-stage pipeline located between the fetch stage and the instruction memory queue. Each time the line buffer reads an instruction from the instruction cache, it is passed into the branch prediction unit, and a prediction is produced four cycles later. If a branch is predicted to be taken, the front end is flushed and the program counter (PC) is updated accordingly.

In the first cycle, reads are issued to the GShare saturation table, tournament table, local history table, and their corresponding valid arrays. In the second cycle, these values are returned from SRAM. In the third cycle, the branch target address is computed and reads are issued to the pattern history table using the index provided by the local history table. In the fourth cycle, all predictor outputs are available and the final prediction is made.

If both the GShare and two-level predictors agree, that prediction is used. Otherwise, the tournament predictor selects between them. Predictor state is written back on commit through the ROB to ensure precise updates and to avoid training on speculative instructions that are later flushed due to misprediction. For example, if back-to-back branches are fetched and the first is mispredicted, the second branch is flushed and should not update predictor state.

Several inefficiencies were identified in this design. All instructions pass through the branch predictor pipeline regardless of whether they are branches, which introduces unnecessary latency. This could be optimized by bypassing non-branch instructions. Additionally, pattern history table accesses could be issued earlier to reduce overall prediction latency. Predicting using PC plus four instead of PC could further reduce delay, though this was not implemented.

Using the tournament predictor, we achieved approximately 92 percent prediction accuracy across five test cases, which resulted in improved IPC by reducing the frequency of pipeline flushes. The recorded accuracy and IPC metrics are shown below.

Branch Prediction Accuracy

Test	Static Not Taken	Static Taken	GSHARE Only	Two Level Only	Tournament
Coremark	46.50	53.50	90.18	61.15	91.50
AES	40.71	59.29	94.70	60.35	94.85
Compress	41.27	58.72	98.51	41.70	99.54
FFT	24.47	75.52	93.63	41.51	93.18
Merge	63.79	36.21	76.87	65.82	79.04

Branch Prediction IPC

Program	Static Not Taken	Static Taken	GShare Only	2-Level Only	Tournament
coremark	0.286	0.283	0.388	0.313	0.393
aes_sha	0.265	0.277	0.286	0.276	0.286
compression	0.352	0.390	0.547	0.354	0.553
fft	0.410	0.417	0.431	0.402	0.430
mergesort	0.355	0.293	0.392	0.368	0.401

Next-line Prefetcher:

In the base processor design, there is simply a call to cache for an instruction which is handled on a need basis. The goal of the next line prefetcher is to fetch the next line in the cache before it is needed such that if the next line is exactly what the processor needs, the fetch time is reduced. The treadoff for this advanced feature is that while it can see extremely satisfactory IPC increases, its success is heavily dependent and curated towards certain programs. For example, an aggressive prefetcher will not work well on a program with many branches but very well on a program with none. In the end, we decided to design an aggressive prefetcher. In our testbenches, it brought an IPC from 0.231 to 0.355; that is roughly a 54% increase in efficiency.

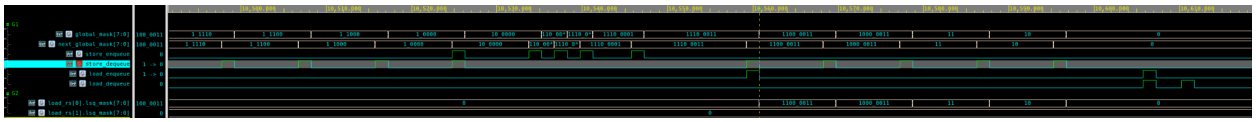
```
################################################
$finish called from file "/home/sslaw2/ece411/mp_ooo/sim/../hvl/common/top_tb.svh", line 53.
Monitor: Total IPC: 0.355542
Monitor: Total Time:          6295855000
$finish at simulation time          6295855000
          V C S   S i m u l a t i o n   R e p o r t
Time: 6295855000 ps
CPU Time:    180.470 seconds;       Data structure size:   1.3Mb
Mon Dec 15 11:23:22 2025
```

Split LSQ:

In our baseline processor design, loads and stores were placed into a single combined load-store queue and processed strictly in program order. Stores are only allowed to issue to memory once they reached the head of the ROB, while loads were serviced when they reached the head of the combined LSQ. This design was simple and correct but lacked certain parallelism that would increase performance.

To address this issue we implemented a split load-store queue by separating the memory operations into an independent store queue and a load reservation station. Stores are maintained in a FIFO structure and are still required to execute strictly in program order while loads are tracked independently with their own wakeup and issue logic. However, because we did not implement address-based disambiguation, loads are blocked by the store mask mechanism: a

load may only issue once all the older stores recorded in its lsq_mask have issued.



As a result, the split LSQ removed the structural serialization in the combined design and allowed loads to be managed independently of store commit. But, because loads still wait for older stores to issue the performance didn't change because the overall effect was still the same. While this design establishes the structure we need for better memory scheduling, other external issues prevented address comparison which sacrificed a significant speedup in memory workloads.

Parameterized Caches:

The decision to do this was just to have some choices when it came to how big we wanted our instruction and data caches to be, as different sizes provided different levels of performance. The baseline configuration was a 4-way set associative cache with 16 sets for both the I$ and D$. By changing the number of ways and sets the cache was made up of, we could change metrics like cache size, average cycles per cache resp, area, power, and delay of our CPU. Below are three different cache configurations that show how changing the number of ways and sets affected our performance. First we have our baseline specs to use an initial value to compare to.

CP3 Baseline

Clock Period: 10000 ps
Frequency: 100 MHz
Area: 229547 um^2

I$
Ways: 4
Sets: 16
Size: 2048 Bytes

D$
Ways: 4
Sets: 16
Size: 2048 Bytes

	FFT	Compression	Mergesort	AES_SHA	Coremark
IPC	0.395	0.389	0.389	0.332	0.323
Power	12.5	12.529	12.703	12.474	12.62
Delay	29.41	11.04	12.36	23.38	9.04
PD^4	9357168.84	186195.52	296875.95	3730341.15	84177.00
Avg I$ Cycles\Fetch	12.27	7.91	8.22	15.96	8.21
Avg D$ Cycles\Fecth	11.07	12.75	9.84	7.63	10.72
BR Accuracy %	24.48	41.28	63.79	40.71	46.50
JARL Accuracy %	0	0	0	0	0

First we tested larger cache sizes to see how performance changed.

Test 1 (Large Caches)

Area: 405749 um^2

I$
Ways: 8
Sets: 32
Size: 8192 Bytes

D$
Ways: 8
Sets: 32
Size: 8192 Bytes

	FFT	Compression	Mergesort	AES_SHA	Coremark
IPC	0.397	0.39	0.396	0.381	0.327
Power	15.67	15.84	15.97	15.79	15.83
Delay	29.19	11.02	12.17	20.35	8.91
PD^4	11376413.6	233604.688	350321.85	2707944.6	99768.106
Avg I$ Cycles\Fetch	12.24	7.91	8.19	12.90	8.07
Avg D$ Cycles\Fecth	10.99	12.72	9.71	6.64	10.58
BR Accuracy %	24.47	41.27	63.79	40.71	46.5
JARL Accuracy %	0	0	0	0	0

Looking at the results we can see that with a larger number of sets and ways the area increased quite a lot (almost double the baseline) which makes sense as we just doubled the number of sets and ways. Another thing to note is that the power also increases due to the increased sizes of the SRAMs and PLRU. The stats that benefited on average from an increased cache size were program delay and the average number cycles per cache request. This tracks as a larger cache can hold more data and therefore less time is spent needing to handle cache misses

as the chase for a hit to occur is more probable. We then tested small cache sizes.

Test 2 (Small Caches)

Area: 185113 um^2

I$
Ways: 2
Sets: 16
Size: 1024 Bytes

D$
Ways: 2
Sets: 16
Size: 1024 Bytes

	FFT	Compression	Mergesort	AES_SHA	Coremark
IPC	0.391	0.389	0.383	0.277	0.318
Power	8.66	8.90	9.05	8.75	8.90
Delay	29.68	11.04	12.58	28.00	9.15
PD^4	6720065.07	132210.6073	226658.047	5378240	62384.167
Avg I$ Cycles\Fetch	12.31	7.91	8.25	20.07	8.31
Avg D$ Cycles\Fecth	11.16	12.75	10.00	9.14	10.86
BR Accuracy %	24.47	41.27	63.79	40.71	46.5
JARL Accuracy %	0	0	0	0	0

The results are as we expected. The area and power that increased drastically with the larger cache sizes has now decreased compared to the baseline model as we now have smaller SRAMs. We also see that the delay and average cycles per cache request have increased due to the increased cache miss rate which requires more memory requests and cache allocation.

Conclusion:

In conclusion, we designed and implemented a fully functional out-of-order RISC-V processor and overcame the performance limitations of an in-order pipeline. Using our Tomasulo-based architecture, we successfully implemented the standard register renaming,

reservation stations, precise in-order retirement, and everything else that demonstrates correct out-of-order execution with in-order commit. Several advanced features were explored to improve performance, including a pipelined tournament branch predictor that significantly reduced control hazards and improved IPC, as well as an aggressive next-line prefetcher that showed substantial gains for suitable workloads. A split load-store queue was also implemented to remove structural bottlenecks and establish a foundation for more advanced memory scheduling, though limited by the absence of address disambiguation. Finally, parameterized instruction and data caches enabled evaluation of tradeoffs between cache size, performance, power, and area. Overall, this work demonstrates a complete and extensible out-of-order processor design and highlights how architectural enhancements directly translate into measurable performance improvements.