MP4 Final Report

December 7th, 2020

Alex Vetsavong
Peter Kircher
Mohan Li

1 Introduction

This machine problem involves the design of a pipelined microprocessor that can execute the RV32I Instruction Set. Instruction pipelining is a technique for implementing instruction-level parallelism within a single processor. Compared with multiple-cycle processors, pipelined processors greatly increase the overall instruction throughput. In this processor design, the instructions flow through 5 different stages: fetch(IF), decode(ID), execute(EX), memory access(MEM) and writeback(WB). We also added support for hazard detection and data forwarding, as well as integrating a basic cache system. This design project helped us understand computer architecture in general, and was a great opportunity for us to apply the knowledge from classes into practice.

2 Project Overview

The goal of the project was to create a simple 5-stage pipelined RISC-V processor with a limited instruction set. Given the priorities of the group, there was no motivation to pursue an out-of-order or superscalar processor for the sake of time and complexity. As such, the group wanted to pursue design goals that maximized performance gains versus the time invested into designing them, such as larger caches, a lower-level cache (L2), and a multiplier extension. For the span of the project lifetime, communication was done through Discord: this included scheduling times to do partner programming and explicitly setting personal deadlines. Work was, for the most part, equally distributed amongst the members by expected complexity, not including the work needed to debug and fix functionality after integrating each member's work, which was often done together through partner programming.

3 Design Description

3.1 Overview

Our design can be broken up into individual components that are combined to make our RISC-V processor. Aside from the options, each component is needed to create a functional and correct processor. For instance, without an arbiter, the I-Cache and D-Cache will collide when allocating cache lines. Without forwarding logic, correctness on data hazards will be compromised. Here are the components of our design:

- Basic pipeline registers, PC register, regfile
- Instruction decode and control ROM
- ALU and CMP
- L1 I-Cache and D-Cache
- Arbiter and cache line adaptor

- Data forwarding and stalling logic
- L2 Unified Cache
- Divider and Add-shift Multiplier

Since it doesn't fall into any of the checkpoints and stands as a skeleton for our design, Figures 1 and 2 show the basic design for our pipeline, before forwarding paths and options are included. The Caches and Control ROM are black-boxed as they will be explained in further detail.

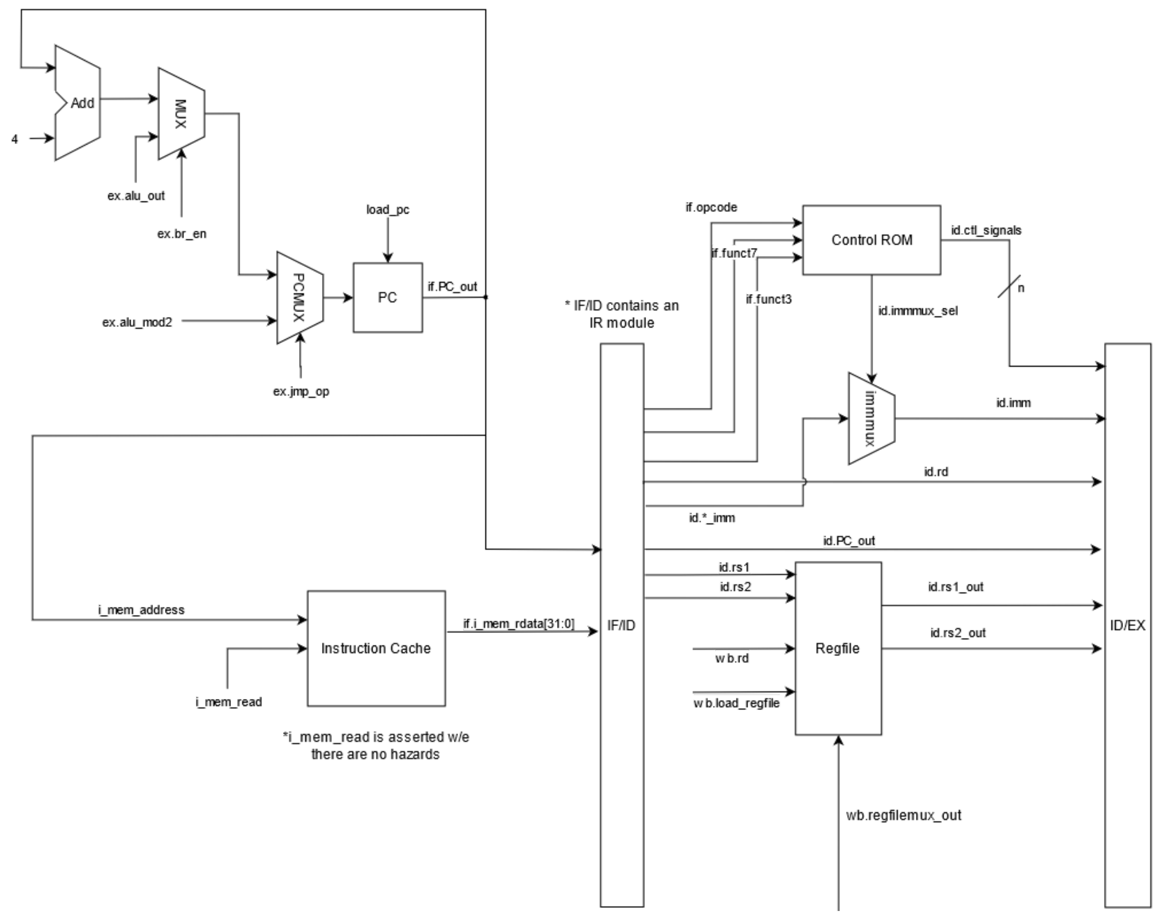

Figure 1: IF and ID stages of basic pipeline

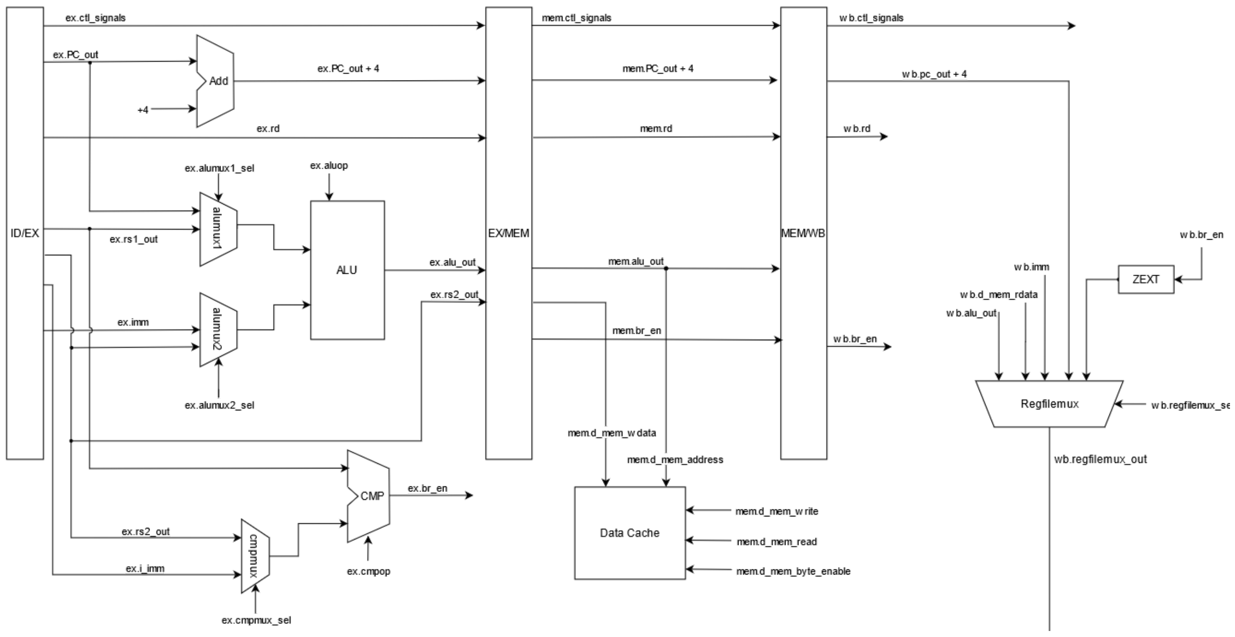

Figure 2: EX, MEM, and WB stages of basic pipeline

3.2 Milestones

3.2.1 Checkpoint 1:

We started this project by drawing the datapath diagram above in 3.1. We had a discussion with our TA Nathaniel on the datapath design, and received constructive feedback from him, such as integrating CMP to the ALU module, and writing the register file on negative clock edges.

With the detailed datapath diagram, we were able to effectively implement the datapath and control module. We made their I/O interfaces simple, by defining a new struct ctrl_word, which grouped all the signals passed from the control ROM, including opcode, funct3 and funct7. For this checkpoint, we used a dual-port magic memory that always sets mem_resp high immediately, so that we did not need to handle cache misses or memory stalls.

The testing process of our basic pipeline design went well: we mainly used the cp1 test code for debugging. Through analyzing the cp1 test code simulation, we found a couple of minor issues, such as alignment in load/store operations that we forgot to add to the design. After we verified the correctness, we also figured out the halting condition for the rvfi halt signal. Apart from the problems mentioned above, we did not find any major issues related to the pipeline structure. Our design passed the autograder for checkpoint 1.

By the end of this checkpoint, we also finished our paper designs for data forwarding and hazard detection, as well as a design for the arbiter to interface your instruction and data cache with main memory.

3.2.2 Checkpoint 2:

By this point, the processor datapath had been set-up and organized to allow for a pipeline structure without data forwarding and branch prediction. In order to implement data forwarding, new MUXs had to be instantiated, and a mechanism for detecting branch hazards and flushing the pipeline registers was needed. The new MUXs can be seen in the below figure.

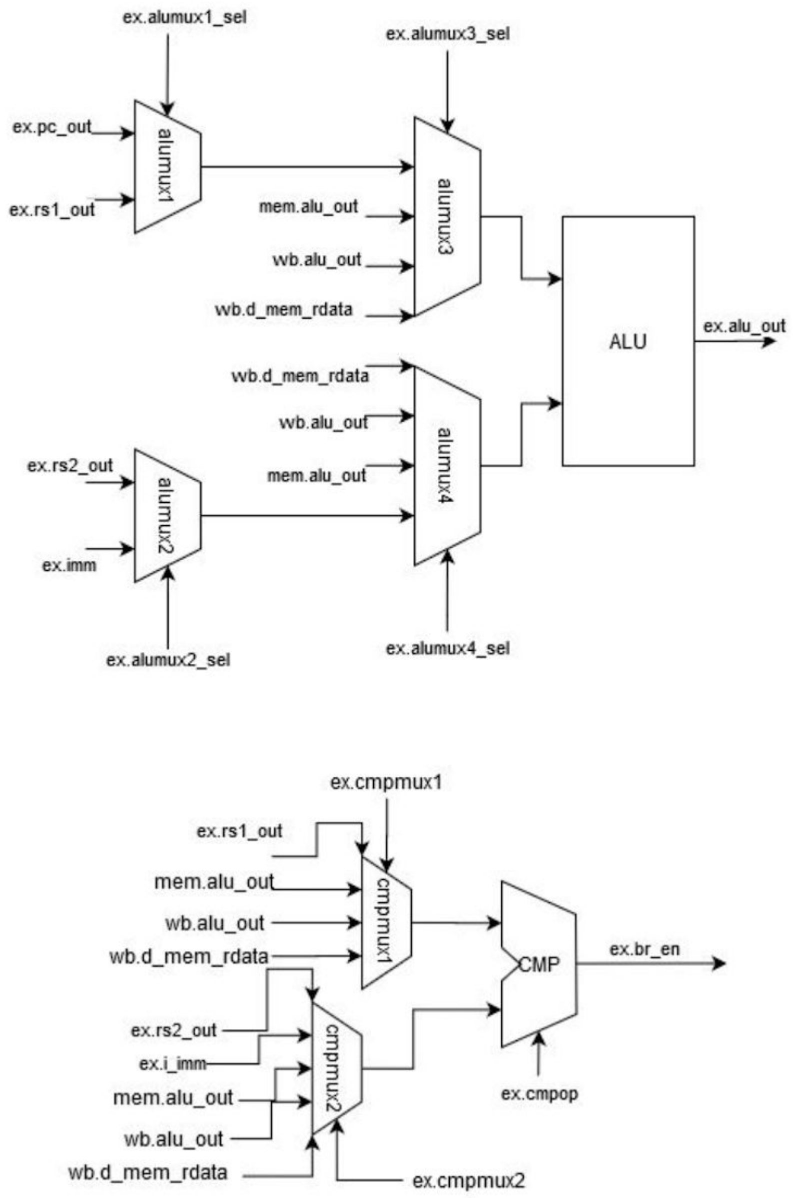

Figure 3: new MUXes with control signals to handle data forwarding

As seen in the figure, various types of data hazards are handled by the MUXs (i.e. MEM -> EX forwarding to handle RAW hazards). As for whether or not there will be a data hazard, we use combinational logic and take advantage of the pipelined structure to look at the destination and source registers of the instruction being issued. The combinational logic is similar to the below figure:

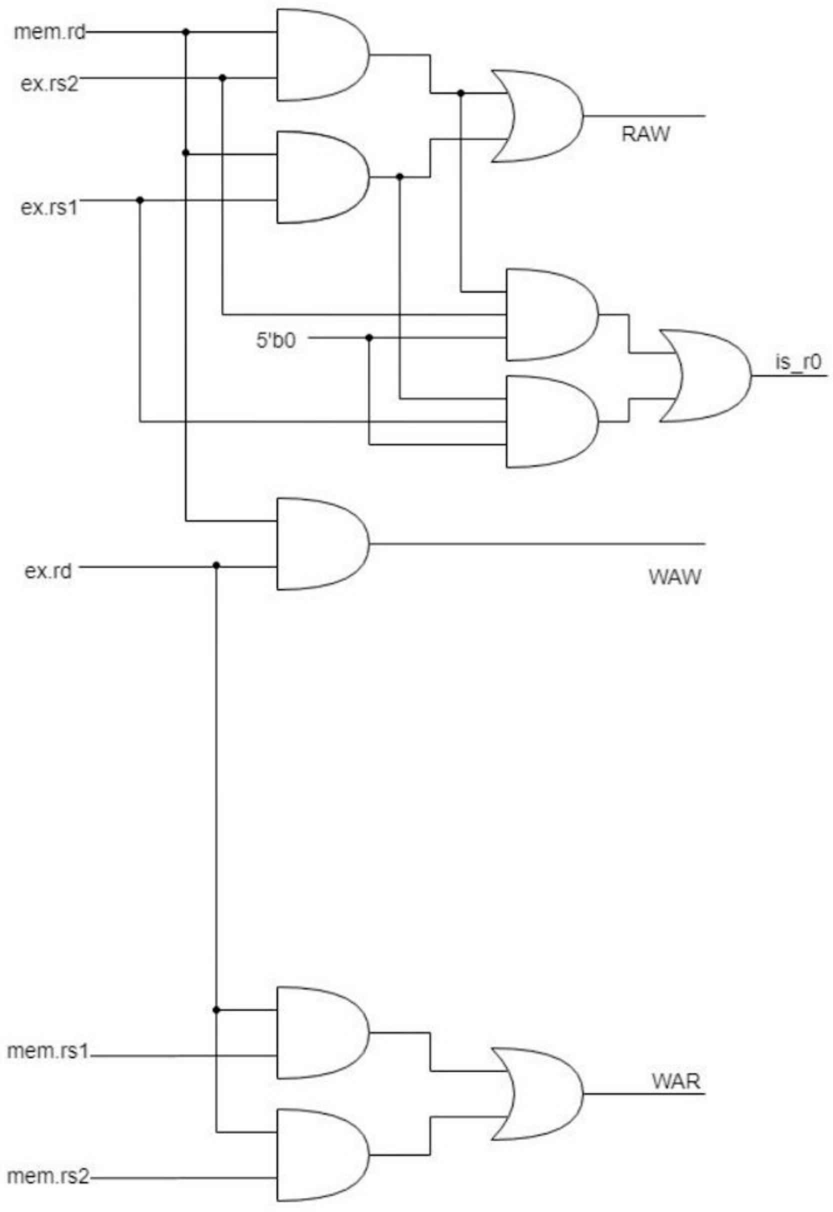

Figure 4: example of hazard detection mechanism

Because of the pipeline structure, and the way we handle double-pumping on the register file, we only need focus on the RAW hazards, so the upper section of the diagram is of more interest to us, and is the only check actually implemented in the final design.

For flushing, we originally based it on the branch opcodes; whenever a branch opcode entered the pipeline, it would set a control signal called is_br high, and we would use that as an indicator for whether or not we needed to flush the IF/ID pipeline registers.

This was rather naive as it did not account for stalls in the pipeline. This meant that even though we should be stalling while waiting for memory reads and writes, the pipeline could be flushed, and this caused conflicts with what we would expect to happen during writebacks for arithmetic operations.

As such, we had to fix our detection mechanism to prevent flushing from occurring during a stall for a memory hazard.

Another error that we encountered was the mechanism for stalling allowing for a runaway program count. This means that even though we should stall the program counter, there were cases where the PC would keep incrementing and cause instructions to be entirely ignored even though they should be issued in the pipeline.

Note, that we have not addressed the condition where unconditional branches or jumps or taken, due to not accounting for those instructions since the test code in CP2 did not cover those cases. We fixed this by adding a control signal for JAL and JALR opcodes similar to the one we use for branches, so that we can indicate a misspeculation and flush the registers appropriately.

3.2.3 Checkpoint 3:

At this third checkpoint, our design was complete with full correctness on the RISC-V limited instruction set, along with some advanced design options. This section of the project gave us the most trouble, as we encountered issues with correctness, issues with performance gains from options, and huge hits to Fmax.

The options added for this checkpoint are explained under the Advanced Design Options section of this paper. They include an L2 cache, a 4-way set-associative cache, and the M-extension to the RISC-V instruction set. Along with these options which made it into our final design, there was a next-line hardware prefetcher which did not add performance to the design and was excluded. To address the troubles faced during this checkpoint, we can look to our hardware prefetcher, data forwarding algorithm, the use of BRAM, and L1 cache complexity.

The next-line hardware prefetcher was designed to work alongside the L2 cache as a quick allocator in the event of an L1 cache miss. The next two blocks of memory were added to the prefetcher on idle memory cycles so that the L1 would not have to bother the L2 cache, which may also miss (in the early cycles of the program). The L2 would then serve as a container for write-back cache lines from L2, and ideally would play less of a part in new data being served to L2. This implementation was found to not improve performance, as it ran into bus conflicts with the L2 cache through the arbiter, slowing down accesses to main memory from L2.

For our data forwarding, we ran into a large bug which our testing did not find. In the RISC-V instruction set, store instructions use the rs2 operand to hold the data to be stored. This register is specified in the instruction word, but no operations happen to rs2 before it reaches the MEM stage. Our data forwarding was not complete by this point, and did not consider the store instruction when forwarding. Therefore, when variables were altered in the two lines before stores, the data was not forwarded and the correct data did not reach the memory location.

Our use of BRAM to scale back on power in our L1 caches caused a huge hit to our Fmax. Below, you can see abridged performance metrics from our original checkpoint 3 cache configuration, which was 4-way L1 caches, and a 2-way L2 cache. Baseline refers to 2-way L1 caches with no L2 cache. The metrics are gathered from the checkpoint 3 test code.

Parameter	Baseline	2-way L2	4-way L1,L2
Total cycles	188214	88647	85545
Fmax (MHz)	52.1	47.2	44.8
L1 I-Cache hit rate	99.9%	99.9%	99.9%
L1 D-Cache hit rate	78.8%	79.3%	85.7%
L2 hit rate	N/A	85.2%	79.6%

Table 1: Checkpoint 3 metrics (bad) for original cache config

As one can see from the downward trend in Fmax, the timing problem was with the caches. One of the first issues we were led to by our TA was that we used BRAM for data arrays in both the L1 and L2. Since we needed to resolve the address and way selection, we at first decided to double pump our data arrays, giving them negative edge reads and writes to accommodate the front-end logic. This cut our time in half, but switching from BRAM to registers did not solve all of our problems.

Our L1 cache complexity caused many issues for us in increasing our Fmax. Due to how we forwarded from the MEM stage, the timing analyzer saw the single-cycle read from the data cache as some abomination that started at the mem.alu_out, went all the way down to the L2 cache, all the way back up to the MEM stage, and ended in the PC register. This path technically couldn't happen, but the combinational assignments could chain that far. We found the main issues here to be the write-through policy of our cache data arrays (allocate cache line while reading) and the logic needed to determine a cache hit in, at one point, a 4-way L1 D-cache. We fixed this by removing the write through policy and trimming the L1 caches down to direct-mapped. These improvements alone contributed to about a 30MHz improvement to our Fmax.

3.3 Advanced Design Options

3.3.1 L2 Cache:

For the L2 Cache, we take advantage of the arbiter and include it in the definition of the L2 cache module, allowing the L2 cache to handle the assignment of cache lines to the I-cache and D-cache. The L2 Cache itself is a set-associative cache with a combinational read/write on a hit. It was originally a two-way cache that held 32 sets. The arrays of the L2 cache are instantiated in BRAM, similar to our original L1 cache from CP1 through CP2, and the cache is unified meaning that we do blocking reads and writes to minimize the complexity of the hardware.

Metric	Baseline	Baseline + L2
Total cycles	188214	88647
Fmax (MHz)	52.1	47.2
Main memory accesses	2324	343
Memory stall cycles	125091	25524
L1 I-Cache hit rate	99.9%	99.9%
L1 D-Cache hit rate	78.8%	79.3%

Table 2: CP3 testcode baseline performance vs performance w/ a 2-way L2

Looking at the table, we see very good gains from implementing the 2-way L2 cache in terms of the total number of cycles needed to execute CP3 test code. Although our Fmax dropped between the baseline and L2 implementation in the original configuration, we do see a massive reduction in the number of main memory accesses (~85% reduction), which accounts for the ~10k cycles that we gain from not stalling. Our L1 hit rates also remain roughly the same (.6% change in the D-cache hit rate is insignificant enough to be in some margin of error).

Metrics	Baseline	w/ 4-way L2
Time (ns)	3,821,912	1,911,601
Power (mW)	488.5	600.13
BRAM size (Kb)	0.736	33.504
I-misses	4685	4685
I-serves	141904	135050
D-misses	146	146
D-serves	12673	5216
L2 Misses	N/A	65
L2 Serves	N/A	4831
Fmax (MHz)	83.28	81.23

Table 3: comp2_i.s baseline performance vs. 4-way L2 cache performance

After making the necessary changes to our datapath to increase our clock speeds, the metrics taken from the competition code further justify the presence of having the L2 cache. We see similar gains in the total time needed to execute the program as in CP3 testcode, but our number of misses in the D-cache specifically is reduced drastically thanks to the L2 being able to serve that data. Although we do see an increase in power consumption, the ~22.8% increased power consumption is still less than the doubling in performance, making the L2 cache a worthwhile feature to keep.

3.3.2 4-way Set-associative Cache:

Starting with modified code from MP3, we already had a cache control design that could support 2-way set-associative cache lines. This means that a LRU (least recently used) array was included, which held a bit for each set. The bit would determine, in the event of a cache miss, which way would be chosen to house the new chunk of memory in the cache. On each access to a set, the LRU would be updated to the other way to keep the LRU bit up-to-date.

However, the single bit system breaks down when there are more than 2 ways, and a queue which orders the ways by how recently they have been accessed requires a lot of logic. Because of this limitation, the main design alteration to incorporate a 4-way cache was to enable a pseudo-LRU replacement policy. This includes 3 bits for each set which act as a sort of binary search tree for the ways. Say you have BST leaves numbered 1-4. They can be grouped into subtrees of 1+2 and 3+4. The first bit of the LRU segment determines which subtree was used least recently, then the next two bits function as normal for the subtrees. The logic determines which way is least recently used by navigating down this representation of a tree.

In order to test this advanced design option, we decided to use the comp3.s, due to its large instruction size and the cache evictions it requires. Since the pseudo-LRU logic was the main change aside from instantiating more register arrays and due to time constraints, we did not rigorously test this option with its own testbench.

The performance of 2-way versus 4-way was tested specifically using the CP3 test code. An abridged version of the results are in Table 4 below:

Metric	2-way L1 + L2	4-way L1 + L2
Total cycles	88647	85545
Main memory accesses	343	332
Memory stall cycles	25524	22422
I-Cache hit rate	99.9%	99.9%
D-Cache hit rate	79.3%	85.7%

Table 4: Performance metrics for 2-way and 4-way cache on CP3 test code

From the table, we see modest performance gains, like a general x1.04 speedup in cycles and 3% fewer accesses to memory on the given test code, which does not make full use of the 4-way set-associativity. It mainly improves the cycles lost swapping data out from the D-Cache, but it does not warrant the Fmax hit from the added logic in single-cycle hit L1 cache service.

Our cache design changed greatly between checkpoint 3 and our final design for competition. Due to the drawbacks of set-associativity and with input from our TA, we decided to change the L2 cache to have 4-way set-associativity while the L1 split caches were reverted all the way to direct-mapped for better performance in Fmax especially. Although the design was switched between levels of the cache hierarchy, the logic for the pseudo-LRU replacement policy did not change, and was easily ported to the L2 cache.

With our final implementation of a 4-way L2 cache, we gain performance in code that accesses a lot of sparse data and reuses it. This would include programs that have long operation times that create large stacks and heaps. The 4-way set-associativity would decrease the time spent accessing main memory, as the L2 cache could store parts of the stack in some of the ways, variables in another, and parts of the heap in others. Since spatial locality can be spread across more than two places in memory, 4-way improves stack and heap operation.

3.3.3 M-Extension:

For the RISC-V M-Extension, we implemented a 32 * 32-bit add-shift multiplier, and a 32-bit divider module. The multiplier and divider were added to the EX stage, and take rs1_out and rs2_out as inputs. We already handled data forwarding in checkpoint 2, so we can simply connect the multiplier and divider to ALU inputs. We also modified the control ROM and regfile MUX to handle the new instructions.

One challenge we encountered was handling stalling for the multiplier and divisor. Since they take multiple cycles to finish the computation, we stall the pipeline until the multiplier/divider finishes. We were able to do this by adding new stalling signals and modifying the stalling logic we had in previous checkpoints.

Both the multiplier and divider were fully tested with testbenches before being integrated into the processor. Instead of testing 32-bit operations thoroughly, we converted the multiplier and divider to 4-bit or 8-bit modules by setting the width parameter. This greatly reduced the simulation time of the tests.

After unit-testing the multiplier and divider, we integrated them to the pipeline and tested with the competition code, through which we found a couple of timing issues in our design. Specifically, the start and done signal of the multiplier were off by 1 cycle, which led to incorrect results.

Metric(comp2_m.s)	Baseline	w/ Multiplier
Time (ns)	3821912	1602546
Power (mW)	488.5	463.36
I Misses	4685	22
I Serves	141904	132450
D Misses	146	66
D Serves	12673	3292
Fmax (MHz)	83.28	83.28

Table 5: baseline performance vs performance w/ multiplier & divider

Eventually, our processor passed the competition test code, and we were able to see some performance improvements. The table above shows the performance metrics of our processor with multiplier and divider only (without cache modifications in 3.3.1 and 3.3.2). Processing multiplication and division through hardware improved the execution time by 58%. This is a reasonable improvement, as we implemented a basic add-shift multiplier.

4 Results and Conclusion

For reference, our final configuration of design options was:

- Direct-mapped split L1 caches with registered arrays
- 4-way unified L2 cache with BRAM arrays
- M-extension with divider and add-shift multiplier

Below are our final performance metrics for our competition design:

Metrics	comp1.s	comp2_m.s	comp3.s
Time (ns)	773259	1619941	1163707
Power (mW)	539.81	534.04	538.07
I Misses	1044	22	5905
I Serves	57750	130664	69701
D Misses	7	66	502
D Serves	2932	3278	18354
L2 Misses	32	45	316
L2 Serves	1051	88	6407

Table 6: Final competition metrics

With more time, we would focus on improving our Fmax, which ended up at 81.23 MHz. This would entail rewriting our forwarding logic to appease Quartus and to shorten the chain of logic to resolve a forward. We also had plans to implement a Wallace fast multiplier and to tweak the hardware prefetcher for it to be included in our design. Another task that would have benefited our competition result would be to iteratively tune our cache parameters to get the optimal score on the combined competition code suite.

In the end, our project met our goals of being a pipelined processor that prioritized large chunks in performance gain, rather than wildly complex additions which yielded x1.02 speedups. Although our design had minimal (not by choice) options to show for checkpoint 3, we take pride in our solid performance. It is satisfying for all teams to see performance gains due to additions that groupmates designed from scratch.