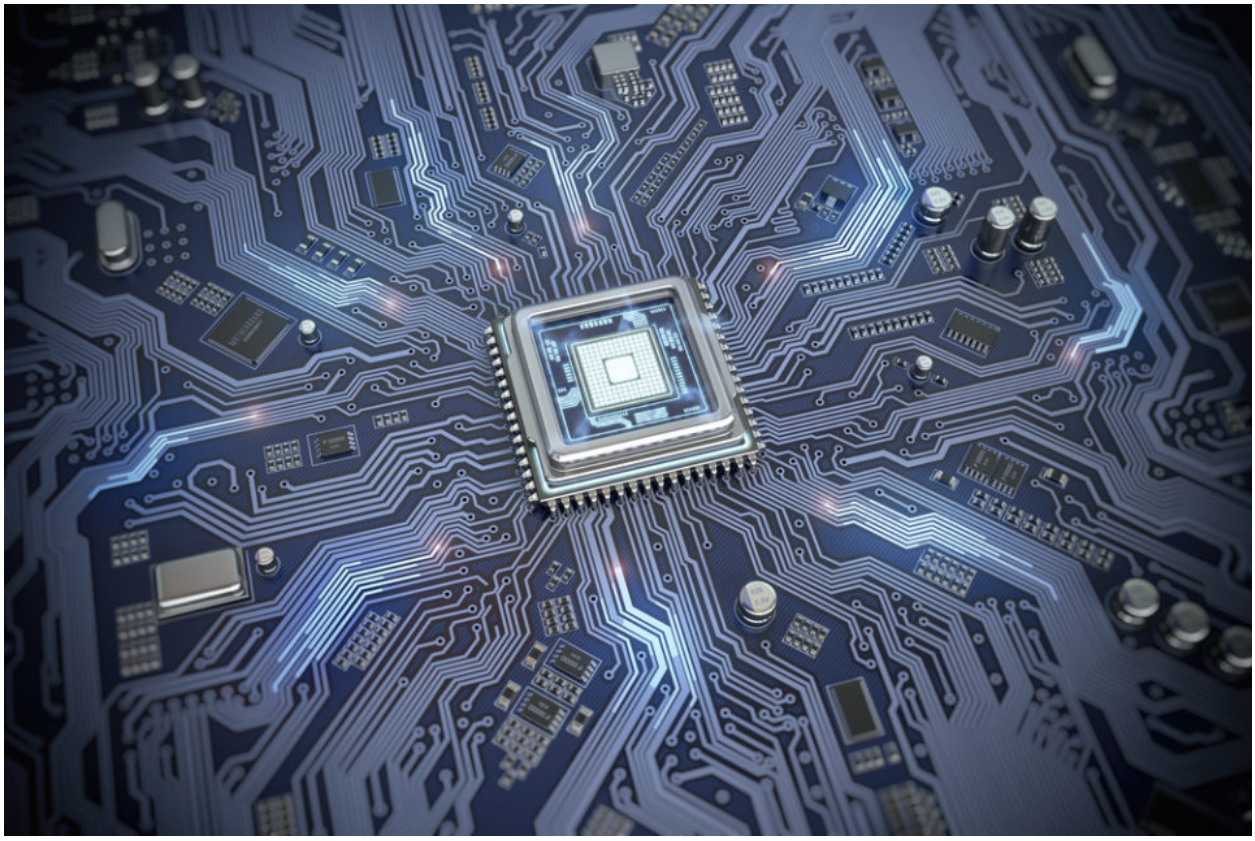
ECE 411 Computer Organization and Design

Five-Stage Pipelined RISC-V Processor



By Sahil Patel, Cindy Feng, and Justin Bifeld

Introduction

In this project we built a five-stage pipelined RISC-V processor with forwarding and hazard detection. With the basic processor in place we added advanced features including a parameterized cache module for an easy L2 cache, an eviction write buffer for the L2 cache, a multiplier extension, a branch target buffer, and both a local and global branch predictor. We will go through each of these features' implementation and the results they each provided on the competition test code. We will also review the final implementation of our design with what we found to be the best parameters using some of these advanced features. This design process was very important in learning computer architecture, as it not only thoroughly taught us how a pipelined CPU works, but also helped us learn how to research and implement complicated hardware.

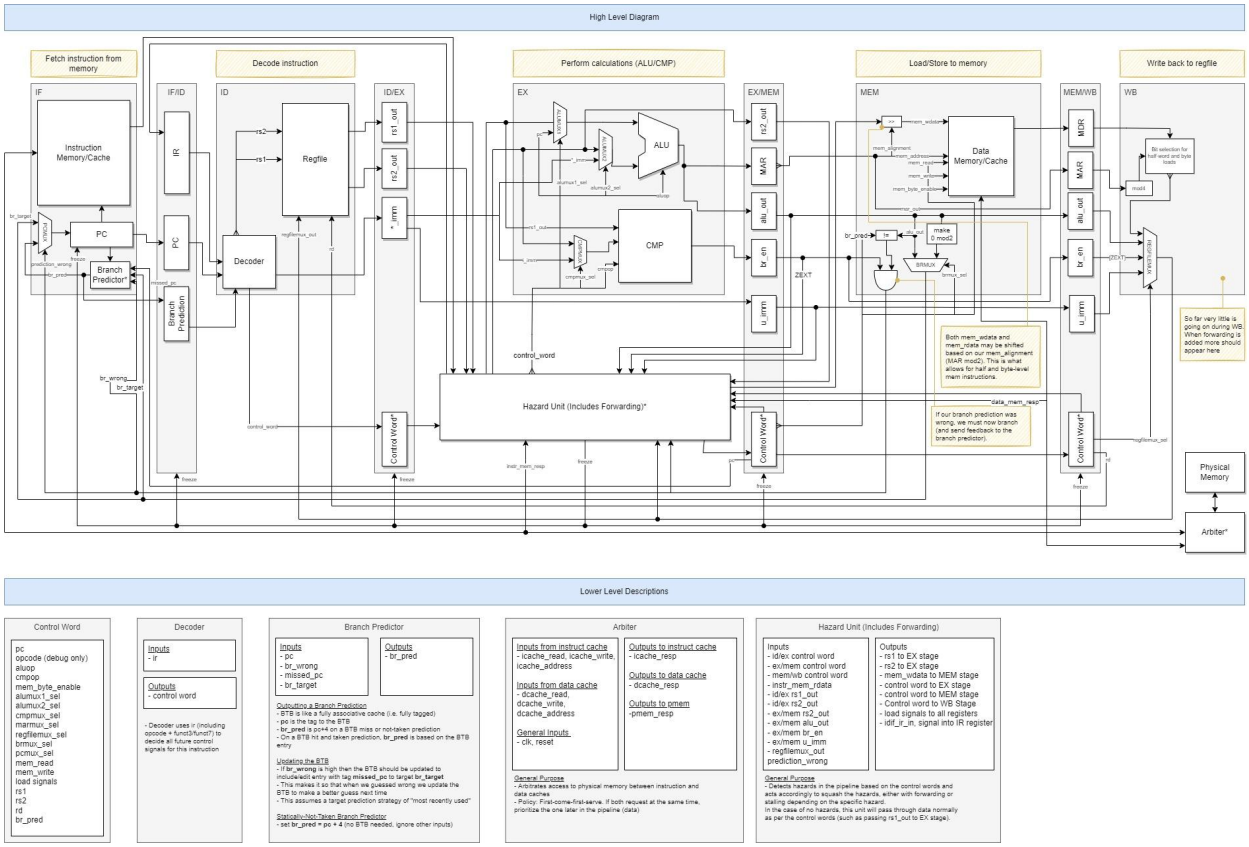
Project Overview

Our main goal was achieving the desired number of advanced feature points while also maximizing our results in the design competition. We wanted to implement several features that would yield noticeable results in our final product. Another goal we had was also maintaining a clean and well documented codebase and design. We knew from the beginning that this would greatly help in designing, receiving feedback, implementing features, experimenting, and debugging.

For checkpoint one and two we worked closely together on getting our base pipelined CPU working. We did not split up any work, allowing all team members to have a strong understanding of the fundamental design we will build off of in the following checkpoints. For our advanced features we did assign certain features for individual team members to focus on and implement. Throughout the entire project we maintained a shared Draw.io file containing all of our designs from the base pipeline to the advanced features. This allowed us to reference any part of our design when necessary.

Checkpoint 1: A Pipelined Processor

To start our project we designed our datapath for our 5-stage in-order RISC-V processor. We referenced many of the lectures as well as our given textbook, *Computer Organization and Design*, written by David Patterson and John Hennessey. We ended up designing the datapath as shown below:



We implemented the basic RISC-V in-order 5-stage pipeline design that handles all of the basic RV32I instructions (with the exception of FENCE, ECALL, EBREAK, and CSRR instructions). We implemented the datapath we had made in our design diagram, making sure to update the diagram as we were implementing with updated signal names and units. At the end we were able to run our code with no data hazards and stalls and make use of the magic memory module we were provided that allowed us to deal with single cycle memory responses. This magic memory module was used to replace our data and instruction cache which were implemented later.

We also started designing some features for the next checkpoint such as our forwarding and our hazard detection unit as well as our design for our arbiter that will allow us to interface with our instruction and data cache with main memory.

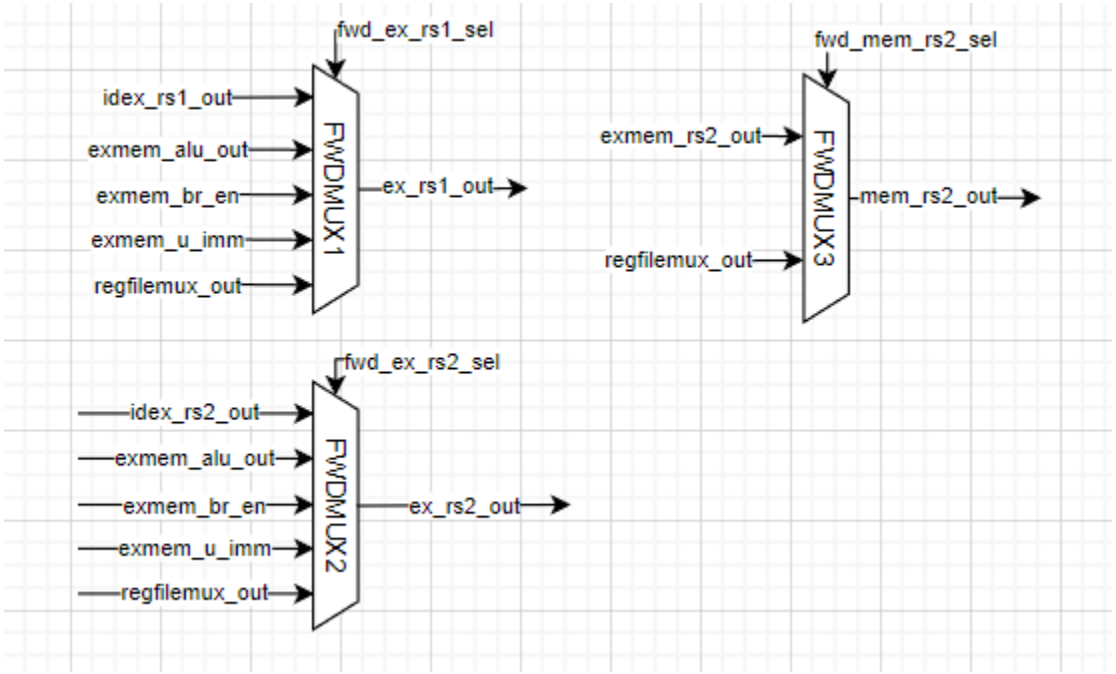
As far as testing, our main testing strategy consisted mainly of using test code we had written for MP2, our non-pipelined CPU. We had to make some minor changes to that test code, such as including nop instructions. However, once those changes were made we had some strong test code for testing individual instructions. This was very helpful in singling out any problems with a specific instruction, which did happen. We found that specifically our JALR instruction was not performing correctly due to a problem with pcmux. Testing instructions one at a time allowed us to find this error. Lastly, we also used the factorial code from MP2 and the provided Checkpoint 1 test code to validate our design.

Checkpoint 2: Forwarding and Hazard Detection

In Checkpoint 2, we had finished our initial design of our 5-stage pipelined CPU and extended our design by adding an instruction and data cache, cache arbiter, a hazard detection unit as well as a forwarding unit. We then connected all of these elements to our main datapath. We used the datapath and state machines we designed in Checkpoint 1, updating changes in the diagram as we ran into issues implementing our design. We were able to successfully integrate our cache and properly deal with multi-cycle memory responses.

The Hazard Unit

We bundled both forwarding and stalling into a single “hazard unit”. Our hazard unit covers three forwarding cases : forwarding MEM to EX, forwarding WB to EX, and forwarding WB to MEM. (WB to ID forwarding is handled with a write-through register file.)



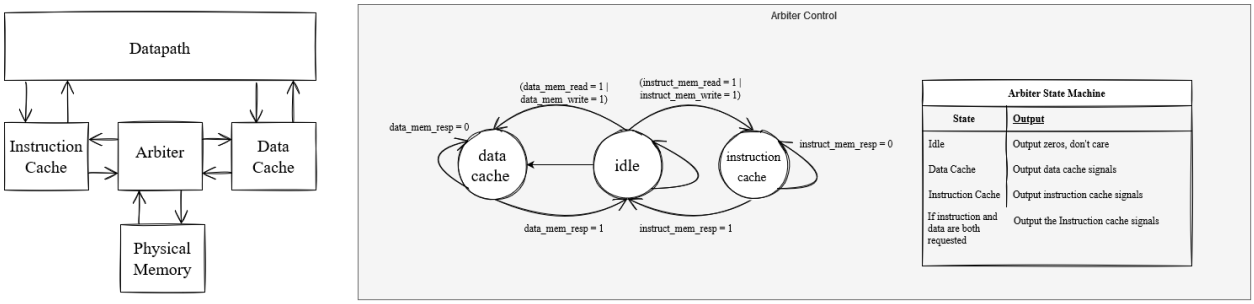
Forwarding Mux IO

The second part of our hazard unit is a mechanism for stalling. Essentially, this portion of the hazard unit detects situations in which a pipeline stage isn’t ready to move on yet, such

as delayed memory access. If a stage needs to wait, all previous stages' registers do not load their input value and the next stage is sent a nop control word via our stall muxes.

The Arbiter

We were given a basic cache module, so integrating the L1 caches was simply a matter of connecting the modules. The larger task was to create the arbiter to moderate access to physical memory. The arbiter is used when both L1 caches need to access physical memory at the same time, and is controlled by a simple state machine.



Arbiter placement (left) and state diagram (right)

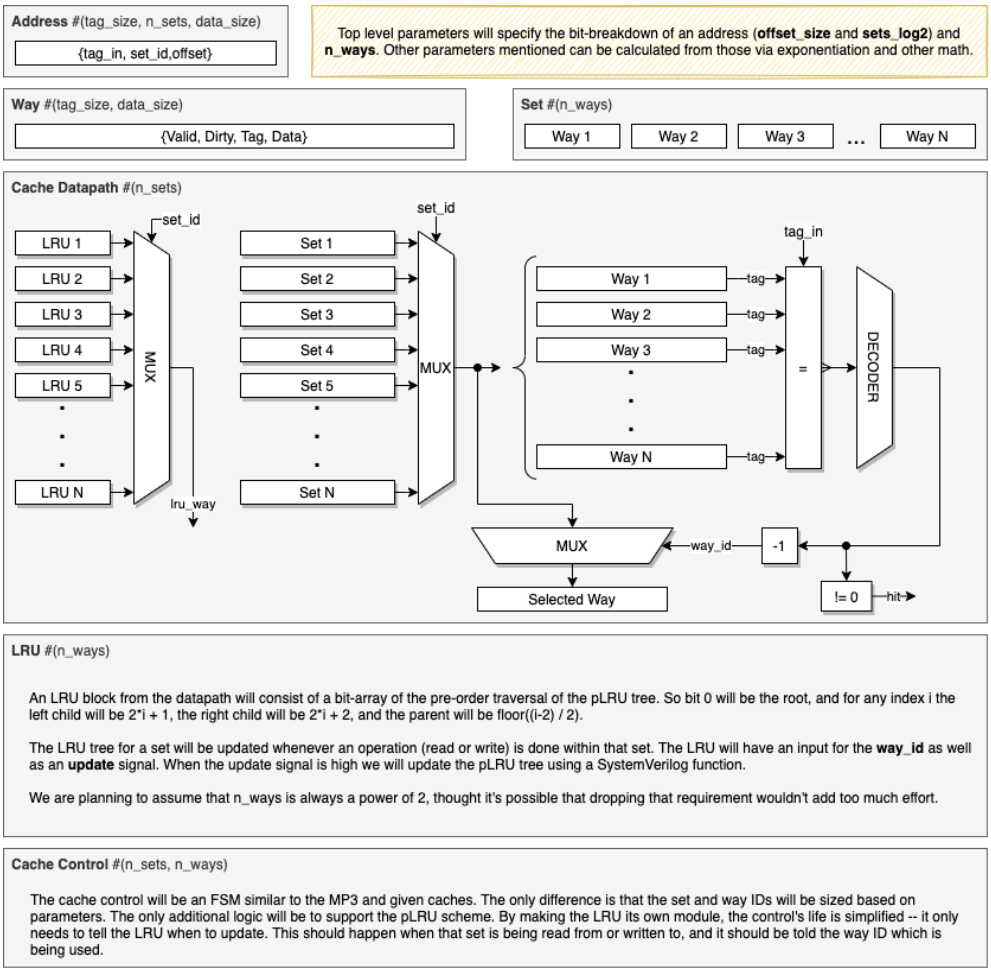
Our main testing strategy was using the provided test code for Checkpoint 2 and using test code we had written for MP2. We made sure to test before and after implementing each design feature to ensure that each part of our design worked. As well as making test codes with hazards and without hazards and testing with our checkpoint 1 code to ensure everything worked properly.

Checkpoint 3: Advanced Design Features

With the basic pipelined design complete, we were ready to add advanced features. These features each focused on improving a different part of the design including the cache system, hardware multiplication/division, and branch prediction.

Parameterized Cache

The cache module provided to us in checkpoint two was non-associative for simplicity. Early on, however, we knew that we would want to experiment with the associativity of our caches for optimal performance. Instead of creating separate modules for two-, four-, and eight-way associative caches, we instead created a parameterized module where both the number of sets and ways could be specified.

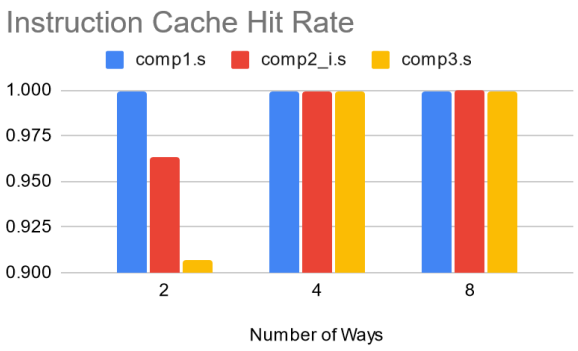


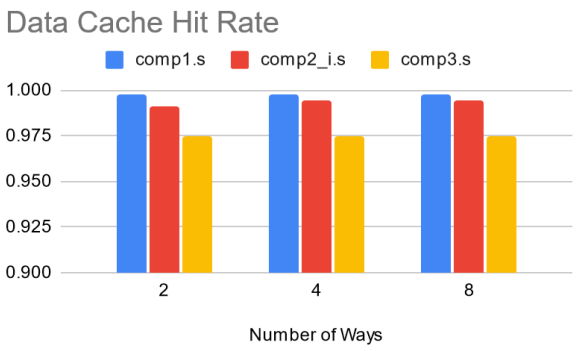
An early design diagram of the parameterized cache

This parameterized cache had a few limitations. We did not allow for non-associative caches with the module (i.e. the number of ways must be at least two). Associativity and the number of sets were also forced to be powers of two to keep indexing simple. The cache line size was also not parameterized (despite what the above diagram planned for) because we did not parameterize the cacheline adaptor. And finally, this module implemented a single-cycle hit cache no matter its parameters. This results in high combinational delay for highly associative caches.

Results

Using this parameterized cache module for our L1 caches allowed us to experiment with the associativity of each. Interestingly, the optimal associativity for the instruction and data cache were not the same. The charts below show how the associativity of each cache impacted its cache hit rate on each of the provided benchmarks.





The first thing to note is that all configurations achieved a hit rate of greater than 90%. The data cache hit rate was not significantly improved by increased associativity, so it is not worth the area or reduction in speed. For the instruction cache, however, moving from two to four ways dramatically improved the hit rate of the cache. Eight ways was well past the saturation point for these benchmarks, though.

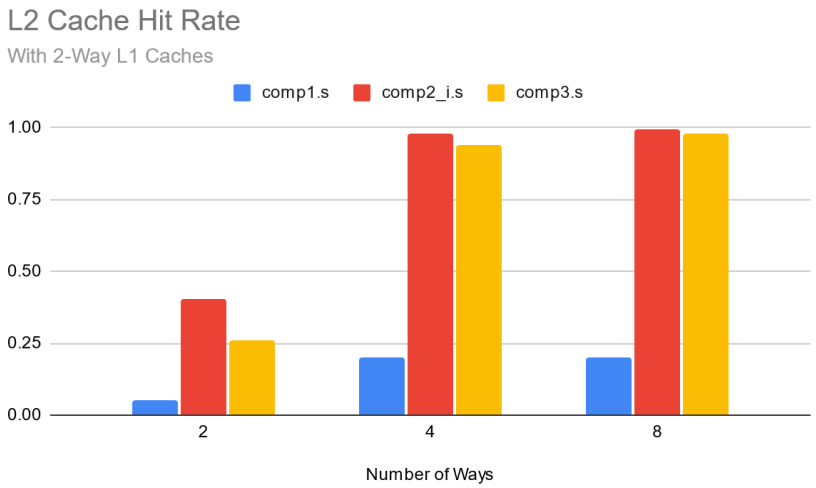
L2 Cache

The parameterized cache module made it relatively simple to add an L2 cache to our design. The main challenge with the L2 cache was reducing its combinational delay. Because our parameterized module is a single-cycle hit cache, a memory request which

misses in the L1 cache and passes clear through the arbiter to the L2 cache forms a critical path drastically longer than any other in the design. To avoid this we modified our arbiter to always wait a cycle before issuing a request to the L2 cache, and also wait a cycle before forwarding on a result. With these changes we were able to maintain a clock frequency above 100MHz with our L2 cache.

Results

We also experimented with the associativity of the L2 cache. In general, however, the L2 cache was very underutilized in the given benchmarks. The following charts assume two-way L1 caches because any more associativity caused the L2 cache to have a 0% hit rate.



There is a marked increase in the hit rate when moving from two to four ways in the second and third benchmarks. However, the first benchmark fails to ever reach a meaningful hit rate. It is also worth noting that each cache takes up a significant amount of the space allotted by our target FPGA. For the L1 caches which had both high utilization and high hit rates, this is clearly a cost worth paying. However, even in the benchmarks with higher hit rates for the L2 cache, actual utilization was relatively low.

Ultimately, we expect an L2 cache to perform well under a benchmark which loads from memory often from the same large set of addresses over time. This would cause the addresses to be evicted from the L1 cache but remain in the L2 cache. Our benchmarks,

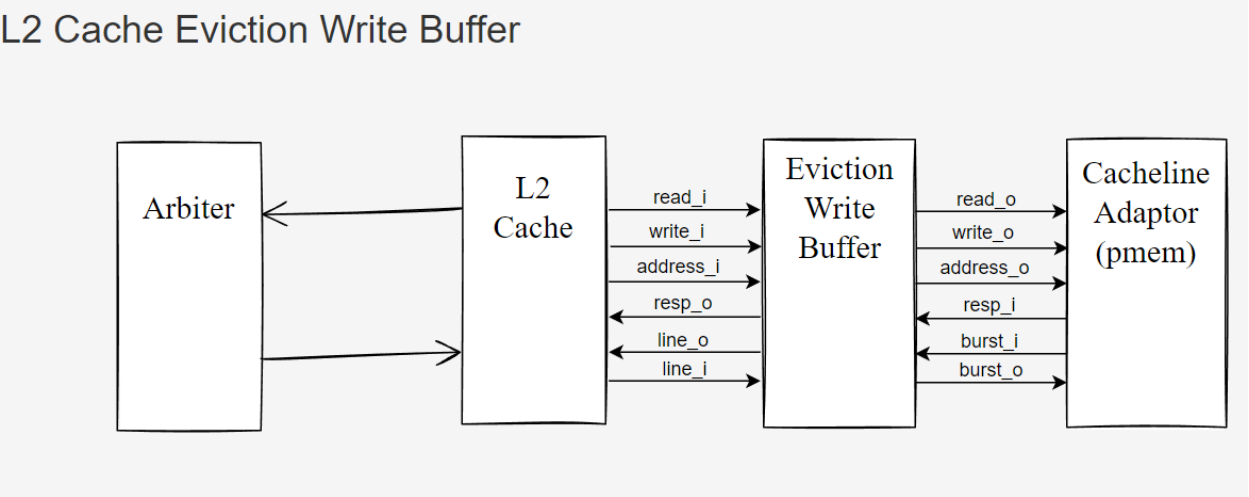
however, did not do this to a large enough extent compared to the size of our L1 caches for this performance advantage to be noticeable.

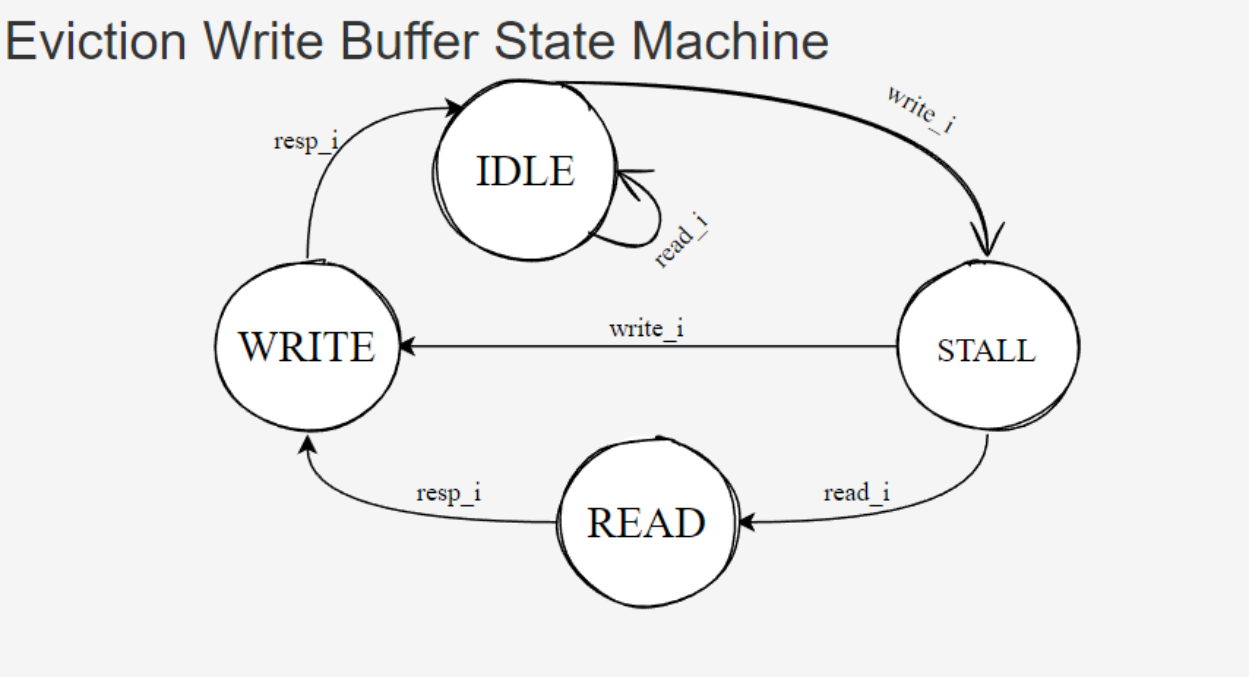
Eviction Write Buffer

The eviction write buffer is a buffer placed between the L2 cache and the cacheline adaptor that holds dirty evicted blocks between the cache levels and physical memory. It does so allowing missed addresses to be processed first. We implemented it by assuming that every time a memory write is performed, we could assume that there would be a memory read that would happen right after. Allowing us to write back after the next read completes.

The addition of the eviction write buffer was effective at hiding write back latency however the total performance benefit is underwhelming because of the low amount of write backs in the competition code. It was also due to our large L2 cache, so there are a very low amount of writebacks caused by the L2 cache, thus the EWB is used more infrequently.

Attached below is our datapath and state machine for our buffer:





RISC-V M Extension

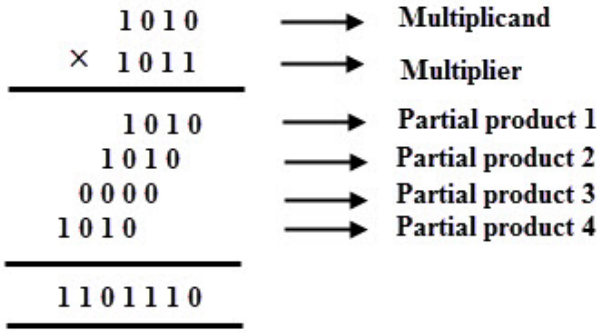
The second provided benchmark used multiplication and division heavily, so early on we were interested in supporting M extension instructions to improve our performance on this code. The M extension supports three main operations: multiplication, division, and remainder (with both signed and unsigned versions). To achieve this, we created a new unit within our EX pipeline stage: the Multiplication/Division Unit (MDU). Within the MDU there are two modules, one for multiplication and one for division/remainder (which can be calculated simultaneously).

Wallace Tree Multiplier

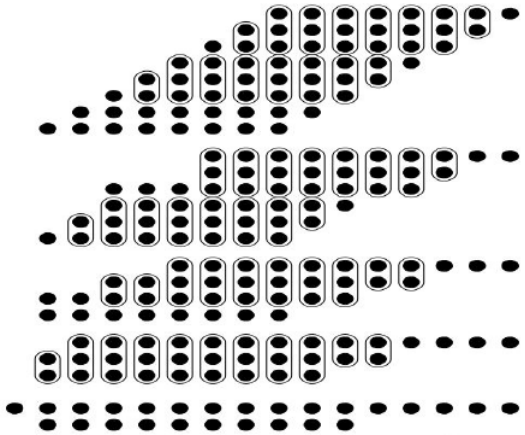
We decided to implement the Wallace Tree algorithm, an advanced multiplier. In traditional multiplication by hand we first create partial products and then add up each column in one big step, resulting in the final product.

The Wallace Tree algorithm essentially splits the large summing step into multiple stages. In each stage we reduce groups of three partial products down to two by using full and half adders (leftover partial products are carried over to the next stage). This allows us to

reduce 32 partial products into just two in eight cycles. Then, in the final step, we simply add the remaining two rows.



Binary multiplication by hand on 4-bit integers



Wallace tree multiplication on 8-bit integers

Subtract-And-Shift Divider

The divider is much simpler. In this algorithm we are searching for the largest number q such that $x = qy + r$ where $r < y$. Here x is the dividend, y is the divisor, q is the quotient, and r is the remainder of a division operation.

We can solve these equations by solving for one bit of q at a time. For each integer $i \in \{0 \ldots N - 1\}$ we check if $x \geq (y << i)$. If it is, we can set q_i to 1 and subtract the shifted quantity from x. After doing this 32 times, the remainder of the division will be whatever is left in x.

This algorithm takes the same amount of time to compute both quotients and remainders at 32 cycles per operation.

MDU Module

To bring it all together we created an MDU module. The wallace tree module outputs a 64-bit product and the divider module outputs a 32-bit quotient and 32-bit remainder. The MDU uses the input operation code to decide which submodule to activate and how to handle the result. For example, RISC-V supports a multiplication instruction which returns the lower 32-bits of the product and another that returns the higher 32-bits.

The MDU serves one other crucial function. Each submodule assumes that its inputs are unsigned numbers. To enable signed operations the MDU takes a cycle to first negate operands as necessary to keep them positive. If the sign of the operands should result in a negative result, the MDU remembers this and negates the output of the corresponding submodule.

Results

The MDU itself doesn't have an equivalent to a cache's hit rate or a branch predictor's accuracy. Instead, we measured the success of the MDU based on the speedup of M-extension-enabled code compared to manual multiplication in assembly. In total the run time of benchmark two decreased by 31% from 1.85 million cycles to 0.57 million cycles.

Importantly, the MDU did originally create a critical path, but we were able to eliminate it. To save extra cycles on M extension instructions, we initially negated operands combinationally during an operation. This created a long path when forwarding was required in the EX stage. To remove this path we created an extra cycle delay in which the MDU negates operands and stores them in a register to be used in the submodules. This removes forwarding from the combinational delay of the MDU submodules themselves. With this change our CPU was able to maintain its 100MHz clock frequency while supporting the RISC-V M extension.

The final result, then, is that the MDU speeds up workloads with heavy use of multiplication and division without slowing down other workloads. The only disadvantage of including it, then, is the space it takes up. In our case, the disadvantage was already quite negligible compared to the size of other advanced components, so the performance gain was well worth it.

Branch Predictors

From our work in checkpoint 1 and 2 we had a static not taken branch predictor that always predicted PC+4 for all branch or jump operations. We saw an opportunity to greatly increase performance by implementing a branch predictor that would achieve a much higher hit accuracy. We implemented a local and global branch predictor, along with a

branch target buffer. Both predictors used 2-bit counters. These predictors were implemented in the IF stage and had parameterized indexing to allow easy experimentation with various table sizes to optimize for the best results. Both predictors were used on BR and JAL instructions while the BTB was used for JALR instructions. When choosing where to branch to we directly decoded the instruction to find the target address for BR and JAL. Since this can not be done for JALR we instead stored its target address in the BTB so we have a successful prediction every subsequent time we see that instruction. Directly decoding the instruction to find the target address did create a long combinational delay that could be resolved by purely using the BTB. However, due to time constraints and trying to hit the target accuracies we decided to stick with directly getting the target address.

Local Branch Predictor

The first predictor implemented was the local branch predictor. This predictor performed better than the global predictor, hitting accuracies above 80% and providing a 10-20% speedup among all three competition codes.

Local branch predictor Runtime	Comp1	Comp2	Comp3
Static not Taken	2,246,155 ns	6,890,275 ns	4,295,155 ns
10 bit History, 8 bit PC Index	1,754,095 ns	5,945,775 ns	3,901,605 ns

Local Branch Predictor Accuracy	Comp1	Comp2	Comp3
2 bit History, 2 bit PC index	77.48%	60.38%	75.15%
4 bit History, 3 bit PC indx	88.94%	75.28%	77.69%
6 bit History, 5 Bit PC index	92.97%	85.34%	80.09%
8 bit History, 6 Bit PC Index	94.15%	87.13%	81.19%
10 bit History, 8 bit PC Index	95.64%	87.25%	81.88%

The main trend to see in the results is that having a larger local history and PC index yielded better results. However, after a certain point the size's impact became saturated. This is most likely due to there now being more elements in the history and counter tables then branching operations in the code.

Global Branch Predictor

The global branch predictor did not perform as well as we had hoped. It was unable to break the 80% accuracy mark in the first and second competition codes, and only barely hit 80% in the third. While it still provided an overall speedup amongst all the tests it was not as significant as the local predictor.

Global branch predictor Runtime	Comp1	Comp2	Comp3
Static not Taken	2,246,155 ns	6,890,275 ns	4,295,155 ns
10 bit History, 8 bit PC Index	2,090,575 ns	6,394,395 ns	3,900,045 ns

Global Branch Predictor Accuracy	Comp1	Comp2	Comp3
2 bit History, 2 bit PC index	62.84%	58.56%	72.20%
4 bit History, 3 bit PC indx	65.48%	62.00%	73.25%
6 bit History, 5 Bit PC index	66.37%	65.11%	78.49%
8 bit History, 6 Bit PC Index	63.11%	72.27%	81.11%
10 bit History, 8 bit PC Index	63.11%	72.27%	81.11%
10 bit History, 8 bit PC Index (Without GBHR)	86.48%	80.50%	81.77%

One very interesting thing to note is that if we remove the global branch history register (GBHR), the accuracy jumps up dramatically and does hit 80% across all three tests. We are not entirely sure why including the GBHR has such a negative impact and this is unfortunately still a mystery to us.

Branch Target Buffer

The branch target buffer was impactful in increasing prediction accuracy for JALR instructions. When including JALR instructions in our counters for calculating accuracy, we can see a huge improvement in performance. There is anywhere from a 15% to almost 40% increase in accuracy. This adds to the effectiveness of our overarching predictor in achieving faster and more efficient code.

Local BTB (including JALR in counters)	Comp1	Comp2	Comp3
10 bit History, 8 bit PC Index, With BTB	94.21%	85.81%	88.43%
10 bit History, 8 bit PC Index, Without BTB	72.63%	73.76%	51.23%

Checkpoint 4: Finding the Optimal Configuration

Using the data gathered on each advanced feature above, our final task was to find an optimal configuration of our CPU when measured using the given benchmarks. The formula used to calculate the "score" of a benchmark was $score = power * (time)^3$ where power was measured in Watts and time in seconds, and a lower score was better.

From our initial results, we quickly concluded that the L2 cache, and therefore the eviction write buffer, were not worth including in the final configuration. This was for two reasons. First, the L2 cache used enough space on our FPGA that it limited our options for L1 cache and branch predictor table sizes. Second, the L2 cache increased our power usage significantly with all of its registers.

Since we noted that higher-associativity L1 caches reduced the usage of the L2 cache to negligible levels, we decided that spending our space was better spent on L1 cache size which also would not increase our power consumption as significantly. Additionally, we noticed that only the instruction cache's size needed to be increased to significantly reduce the usage of the L2 cache.

It was also clear that of our two branch predictors, the local predictor was more accurate.

With that in mind, the three parameters we experimented with were the associativity of the instruction cache and the table sizes of our local branch predictor. The table below shows the results of three configurations beside a provided baseline.

Baseline			
Description	Comp1 Power (mW)	Comp1 Time (cycles)	Comp1 Score
Provided baseline scores	448.44	7.21E+05	1.68E-10
	Comp2 Power	Comp2 Time	Comp2 Score
	430.48	4.52E+06	3.98E-08
	Comp3 Power	Comp3 Time	Comp3 Score
FMax (MHz)	425.27	3.64E+06	2.05E-08
100		Geo Mean	**5.15E-09**

Configuration 1			
Description	Comp1 Power (mW)	Comp1 Time (cycles)	Comp1 Score
L1 cache only. 2 way instr, 2 way data. Local branch predictor with 5 History 6 PC	640.7	5.12E+05	7.72E-11
	Comp2 Power	Comp2 Time	Comp2 Score
	560.73	5.69E+05	9.26E-11
	Comp3 Power	Comp3 Time	Comp3 Score
FMax (MHz)	597.3	3.39E+06	2.09E-08
103.67		Geo Mean	**5.31E-10**

Configuration 2: Larger Instruction Cache			
Description	Comp1 Power	Comp1 Time (cycles)	Comp1 Score
L1 cache only. 4 way instr, 2 way data. Local branch predictor with 5 History 6 PC	693.91	5.10E+05	1.12E-10
	Comp2 Power	Comp2 Time	Comp2 Score
	614.34	5.69E+05	1.38E-10
	Comp3 Power	Comp3 Time	Comp3 Score
FMax (MHz)	671.65	6.59E+05	2.34E-10
93.64		Geo Mean	**1.53E-10**

Configuration 3: Larger Branch Predictor			
Description	Comp1 Power (mW)	Comp1 Time (cycles)	Comp1 Score
L1 cache only. 4 way instr, 2 way data. Local branch predictor with 7 history and 7 pc	726.89	4.99E+05	1.19E-10
	Comp2 Power	Comp2 Time	Comp2 Score
	639.68	5.57E+05	1.46E-10
	Comp3 Power	Comp3 Time	Comp3 Score
FMax (MHz)	704.98	6.51E+05	2.57E-10
91.17		Geo Mean	**1.65E-10**

From these tables we learned that increasing the instruction cache size was well worth its power cost and frequency reduction, but a medium-sized branch predictor was better than a larger one. For the final competition we chose configuration two with a four-way instruction cache, two-way data cache, and five bits of local history indexed by six bits of the PC.

At the time of writing we have not received information about how our processor performed compared to the rest of the class.

Conclusion

Through this project we learned about the detailed considerations for implementing a pipelined processor, including more basic concepts including possible hazards and their resolutions, as well as the more complex integrations with other advanced features. We started by building the basic framework for our pipelined design and moved to adding hazard detection and resolution through forwarding. We then improved the design with caches, branch predictors, and a multiplier. Finally, we experimented with our design parameters to find the configuration that performed best on a set of given benchmarks. This process taught us not only basic SystemVerilog design principles, but also more important concepts. For example, we learned the value of parameterization when in Checkpoint 4 it made it a lot easier to try many different configurations. We also discovered how we often needed to modify our original design in order to fit design constraints like our clock frequency and power consumption.

The most illuminating portion of this project was the final checkpoint. Since testing a single configuration took about an hour and a half, we couldn't simply test everything and choose the best version. Instead, we had to use our knowledge of our designs and their performance characteristics in isolation to make inferences about how certain design tweaks would affect the final outcome. The most evident example of this was in our abandoning of the L2 cache. Our data made it clear that a slightly larger L1 cache would be a much more space-efficient way to achieve the same or better performance as an L2 cache. This allowed us to avoid testing any configurations with L2 caches and saved us a lot of time which we then spent on fine-tuning the branch predictor parameters.

Overall, this project represents an effort to solidify our theoretical knowledge of computer architecture through a practical application of the techniques taught in class. With this concrete experience building and validating a SystemVerilog design from paper to simulation, we now feel comfortable with the design process and hope to apply that knowledge in industry in the future.