MP4 Final Report

Pipeline Trio
Justin Zhou, Ashhal Shamsi, Davin Clark
TA: Shreyas Mohan

Table of Contents

1. Introduction

The purpose of this project is to design, implement, and test a 5-stage pipelined processor that supports the RISC-V I instruction set. This report will discuss the basic features of the processor as well as all the advanced features we added to improve the performance of the processor. This project is a culmination of the knowledge we learned in our computer architecture class. It gives us a good overview of how modern CPUs are designed and optimized, and it shows us the importance of thorough testing throughout the design process.

2. Project Overview

The ultimate goal of this project was to have a fully working RISC-V CPU. This was the baseline of the project, so we also wanted to add additional performance improvements if there was time in order to do well in the final competition. To accomplish this, the entire team worked together when designing each feature of the CPU. This was to ensure that everyone understood how everything works on a fundamental level. When implementing the features, each member worked on whatever they felt the most comfortable with, allowing us to efficiently create an initial design of each feature. After everyone was done with their initial implementations, we came together and tested each feature together. That way, there would be more people to help catch and fix bugs in the design. Through this distribution of work, we were able to fully complete the base CPU and implement many different advanced features.

3. Milestones

3.1 Checkpoint 1: Pipelining

The first step in creating our processor was to pipeline the existing non-pipelined processor. To do this, we created a detailed datapath for each stage of the pipeline. We split up the processor into five stages: instruction fetch, instruction decode, execute, memory, and write back. There is also a stage register between all the stages to pass data on to the next stage. The following diagram is the overall datapath of the processor. We also created diagrams for each stage, featuring all of the intermediate and control signals at that stage.

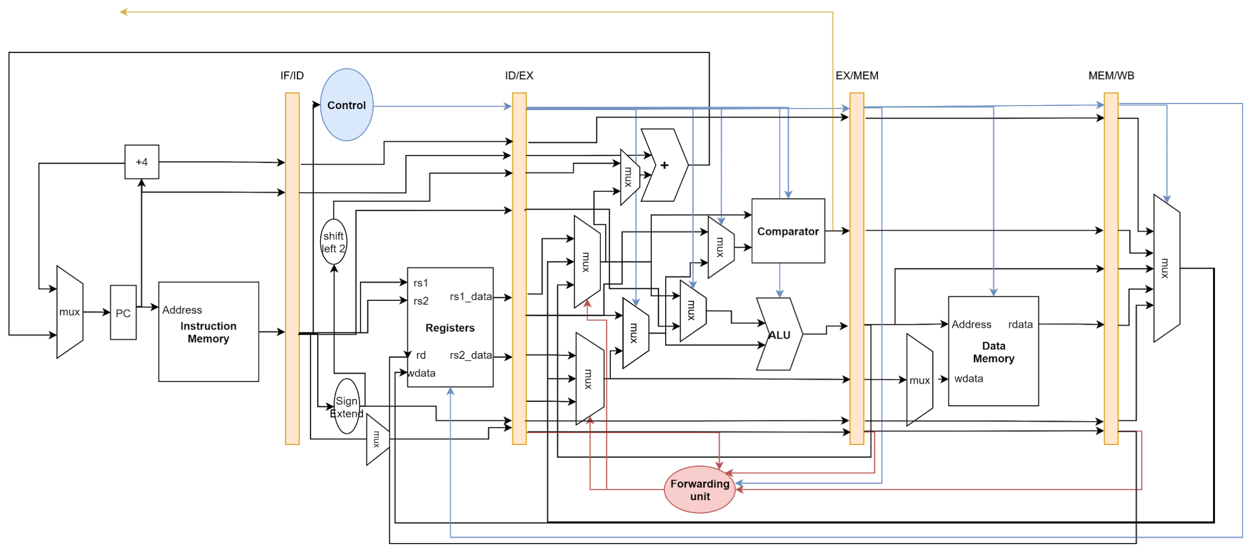

Figure 1: Basic Pipelined CPU Datapath

After creating the datapath, we needed to define the control signals. To make things easier, we created packed structs in which we would store the specific control signals for each stage. Next, we went through all of the instructions and assigned the appropriate control signals for each. This would give us a basic 5-stage pipelined CPU.

In order to test the processor and verify that it is working, we made sure to test each stage as its own DUT before assembling it all together. We did this by passing in values into a stage, and making sure its output values were correct, e.g. passing an instruction into the instruction fetch and making sure it output the correct opcode and register/immediate values. After we verified each stage worked, we put all the stages and stage registers together and began testing each instruction by using both self-written tests and given tests. We followed the execution of the code through the processor and made sure each instruction gave the intended behavior. If it did not, then we would trace the execution of the instruction through the stages to find out what control signal caused an error. By doing this for each instruction, we were able to get a functional pipelined RISC-V processor.

3.2 Checkpoint 2: Arbiter, Hazards, Static Branch Prediction

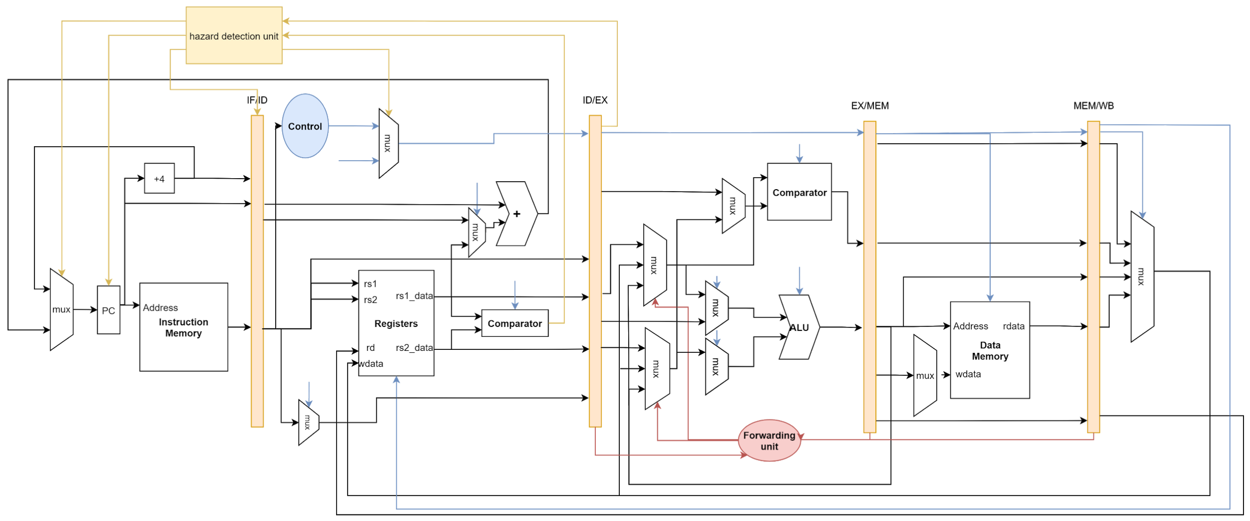

Figure 2: Pipelined CPU with Forwarding, Hazard Detection, and Static Branch Prediction

In Checkpoint 2, we added static branch prediction. We simply predicted that the branch was not taken in this checkpoint. In the event of a branch, we clear the IF-ID Register and send a NOP to the ID-EX Register. This ensures that instructions that occur after a branch that should not be executed are not executed.

To test this, we used the cp2 test code, which issues the following branches: forward taken, forward not taken, backwards taken, backwards not taken.

We also implemented hazard detection and forwarding. There are three forwarding paths: WB to EX, MEM to EX, and WB to ID. We simply check to see if the desired register matches the destination register. These forwarding paths do not account for when the destination register for a memory operation is a source register for the next instruction. For this reason, when this situation occurs, we insert a nop in between them.

To test this, we tested instructions with either the first or second source register being written to one, two, and three instructions ahead. We also used the cp2 testcode, which has a lot of scenarios where forwarding is needed.

Finally, if there is a miss on either Cache, we simply stall the whole pipeline by not loading the pc and stage registers.

In addition to hazards and static branch prediction, we also had to implement the arbiter to interface between the two caches (i-cache and d-cache) and the main memory. The arbiter sits between memory and the caches and handles read and write requests from both caches and decides which cache will read/write to memory at any given time.

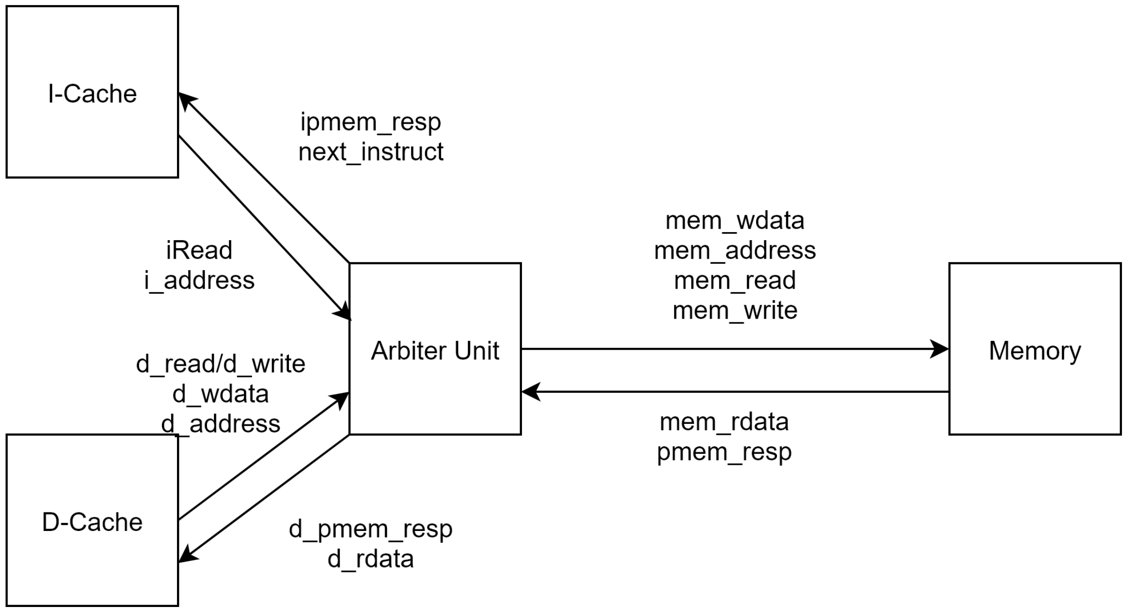

Figure 3: Arbiter Datapath With Signals

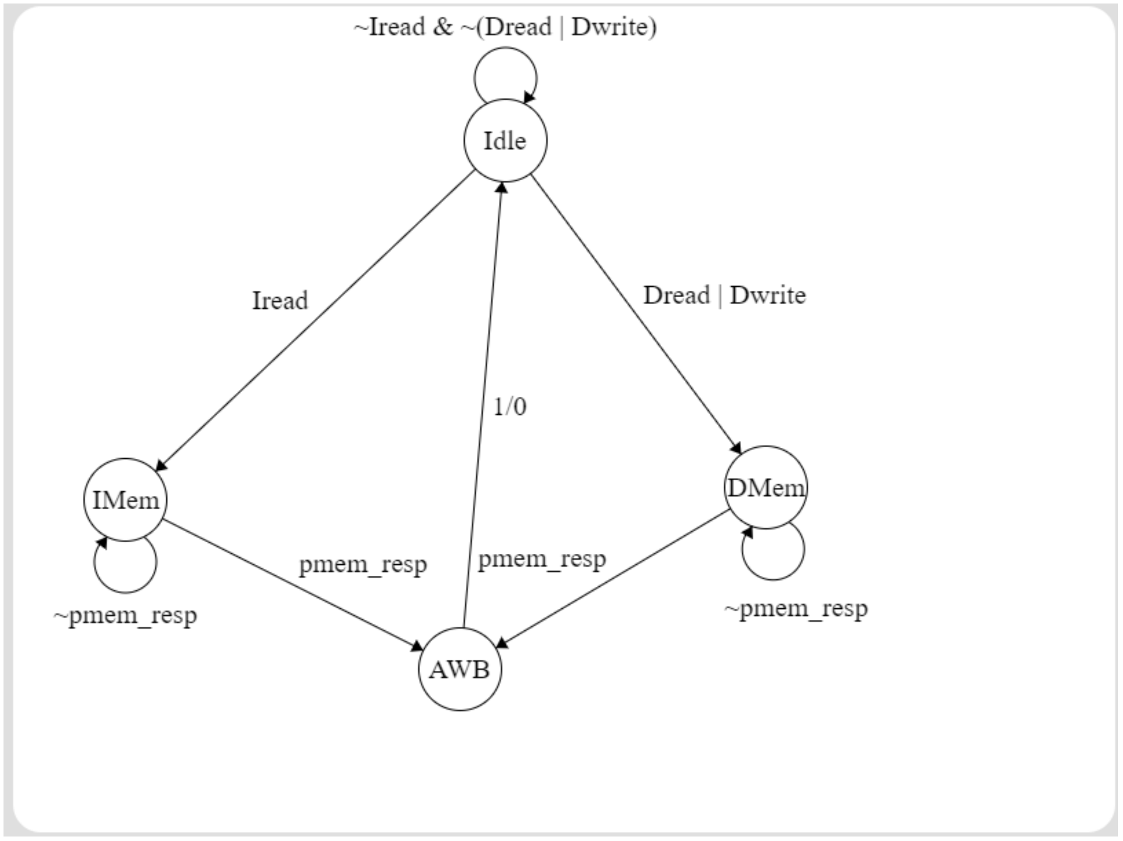

Figure 4: Arbiter State Machine

The arbiter features four stages: idle, IMem, DMem, and AWB. Idle is a waiting state, IMem handles i-cache reads, DMem handles d-cache reads and writes, and AWB is an additional state to wait for the cache responses.

To test the Arbiter, we first used a testbench and made it a DUT. We then simulated i-cache reads and d-cache reads/writes. Once we made sure the Arbiter worked properly, we attached it to the CPU after hazard detection and forwarding was implemented since it would not work otherwise. We then made sure all test code ran the same as with magic memory, which it did.

3.3 Checkpoint 3: Advanced Design Options

For this checkpoint, we chose to implement the following advanced design options: tournament branch prediction, a pipelined L1 cache, an L2 cache, hardware prefetching, and the RISC-V M Extension. For each feature, we created a new branch from our master branch (working base CPU) and implemented and tested each feature separately. Once each feature was tested and working, we then merged all of them together into one CPU. Finally, we performed more tests to ensure that all of the features worked together properly. The specific implementation and testing of each design feature is explained in the next section.

4. Advanced Design Features

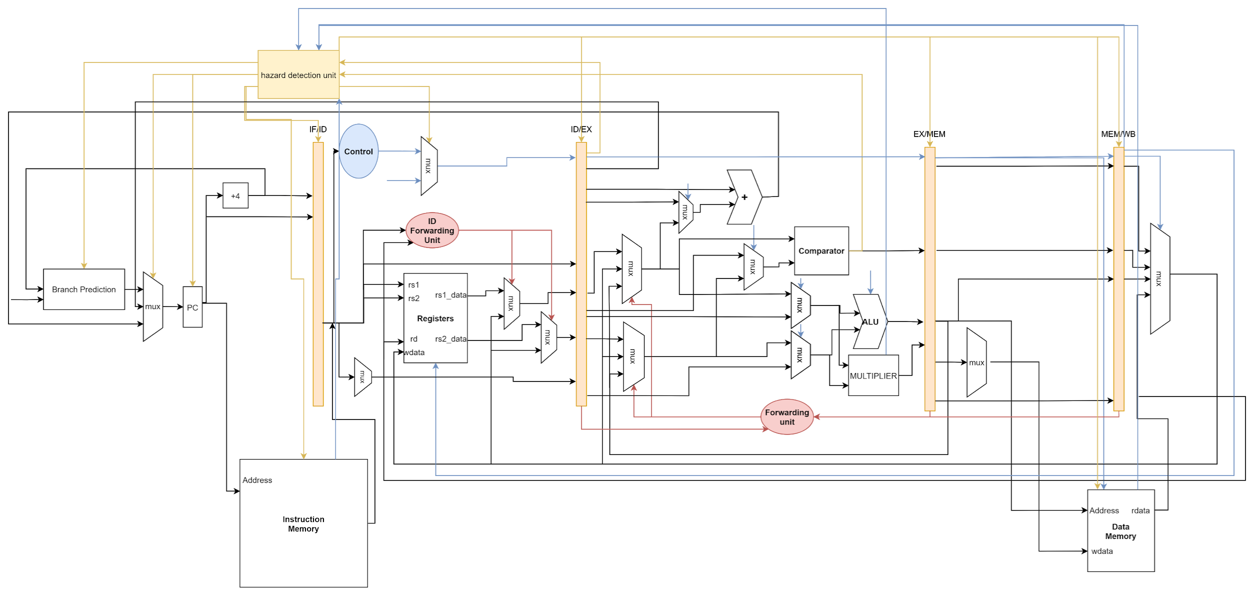

Figure 5: Complete CPU Datapath Overview

4.1 Tournament Branch Prediction

4.1.1 Design

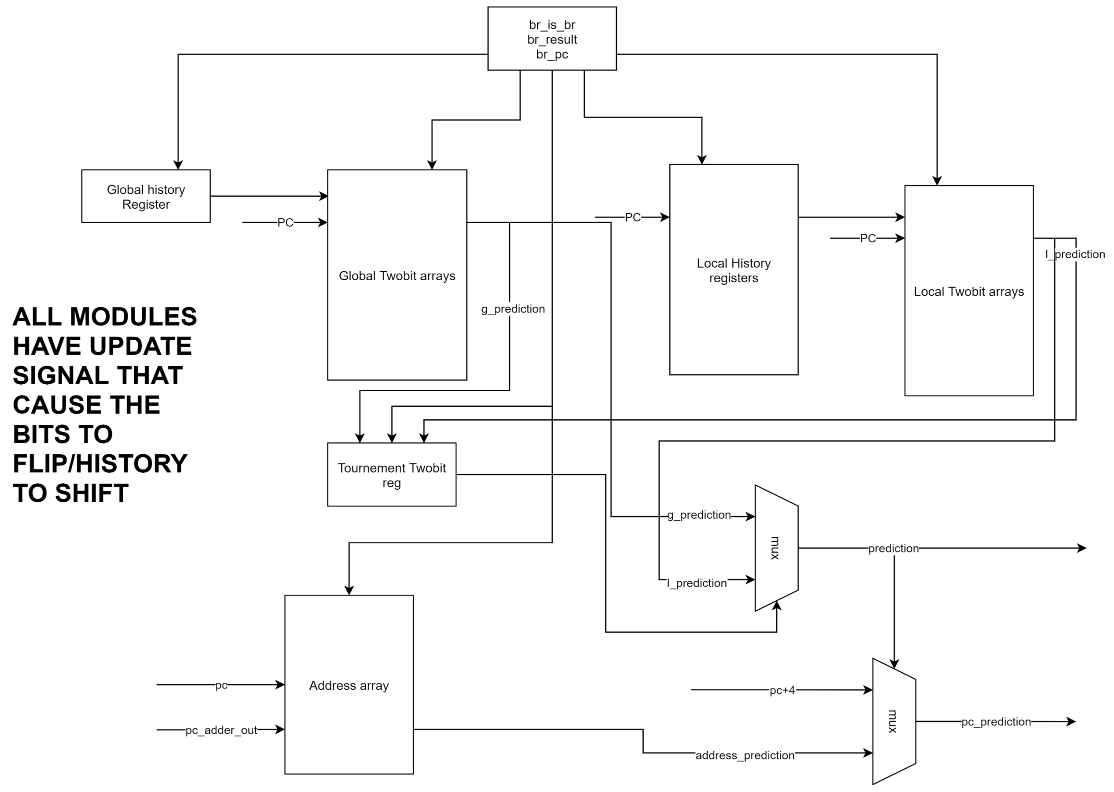

Figure 6: Tournament Branch Prediction Datapath

We implemented a tournament style branch predictor, which uses a local branch history table and global branch history table. Either the global history or local history was appended to the pc to lookup the corresponding two-bit counter. These counters give a prediction on what branch to take. Another two bit counter is used to keep track of which branch predictor has performed more accurately recently, and chooses between the two predictions. The pc is also used to look up a branch address. At the same time, the result of the branch from two instructions ahead has been calculated, and this result is used to update all of the arrays.

This creates two additional hazards: branch taken incorrectly and branch address incorrect. The hazard detection module detects these two hazards, as well as branch not taken incorrectly, and uses this to determine pc. If a hazard occurs, either pcadderout or oldpc+4 is loaded into pc and IF-ID register and ID-EX register are fed nops. Otherwise, the prediction is used.

4.1.2 Testing

To test out our branch predictor, we created all of the possible hazard scenarios to ensure that they worked. We also made sure that when predicting taken, forwarding would still work. We also used mp2 testcode and mp3 testcode to make sure that everything was working.

4.1.3 Performance Analysis

Time (ns)	Comp1	Comp2	Comp3
No Branch Prediction	584,995	1,203,385	1,225,205
Tournament Branch Prediction	595,775	1,203,145	1,225,545

Table 1: Execution time for Tournament Branch Prediction

Hit %	Comp1	Comp2	Comp3
No Branch Prediction	0.8865941714	0.8899757299	0.9310779
Tournament Branch Prediction	0.8790365227	0.8906652914	0.9307923771

Table 2: Hit % for Tournament Branch Prediction

We did not have time to tune our branch prediction. As a result, there was not much improvement. We think that implementing a branch target buffer, instead of a partial pc lookup would have ensured that we do not incorrectly predict on a nonexistent branch.

4.2 Pipelined L1 Cache

4.2.1 Design

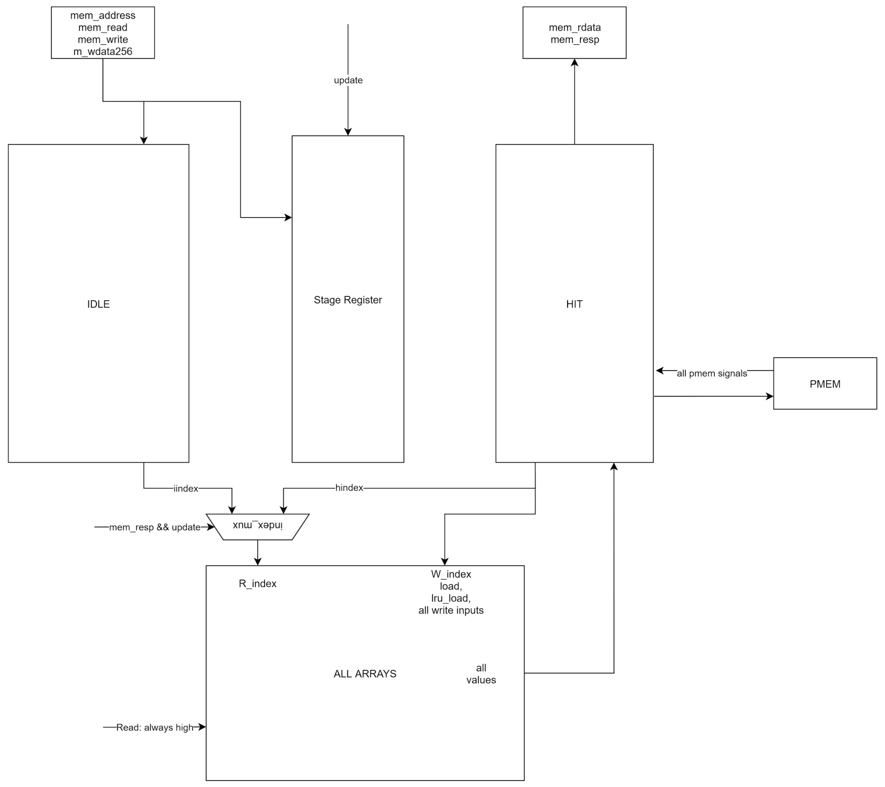

Figure 7: Pipelined L1 Cache Datapath

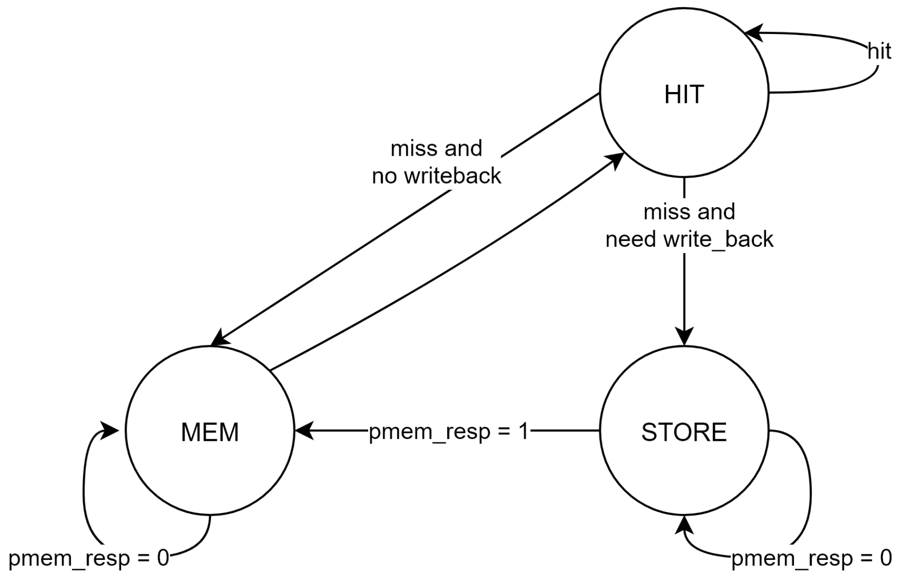

Figure 8: Pipelined L1 Cache State Machine

We based our design off of the mp3 cache. We simply made the idle state the first stage in the pipeline, and the rest of the states the second stage in the pipeline. This means instead of requiring all of the arrays to output a value within one clock cycle, they are given two clock cycles. While there are no misses, the idle stage sends the address of the next access to all of the arrays while the hit stage reads the previous values and sends them back to the cpu. If there is a miss, the hit stage's address is used instead for the arrays, making them hold their previous value. During a miss, the hit stage goes through a very similar state diagram as mp3 to writeback values and read new ones. Only the hit stage interacts with the actual memory.

4.2.2 Testing

At first, we created our own test code to test that reads and writes to an empty cache works. This was to test the misses that require a read. Then we tested this with cp3 test code, since the code was specifically designed to create writebacks.

4.2.3 Performance Analysis

The pipelined cache did not help our competition time, but it did ensure that there were no timing issues with the L1 cache. We could have tested bigger caches since we no longer need to worry about timing, but we did not have the time.

4.3 L2 Cache

4.3.1 Design

The L2 Cache we created is a 16 line 2-way set associative cache. It is basically the same as the L1 cache except with double the lines. We chose to expand the cache this way because we can double the cache size while barely increasing memory access times and while not increasing the complexity of the LRU cache.

For implementation, the L2 Cache goes between the Arbiter and the main memory. To make the cache work, we had to slightly modify the inputs and outputs of the cache to support 32-byte inputs and outputs from the Arbiter. Otherwise, there are very few modifications from the L1 cache. The following is a datapath with signals.

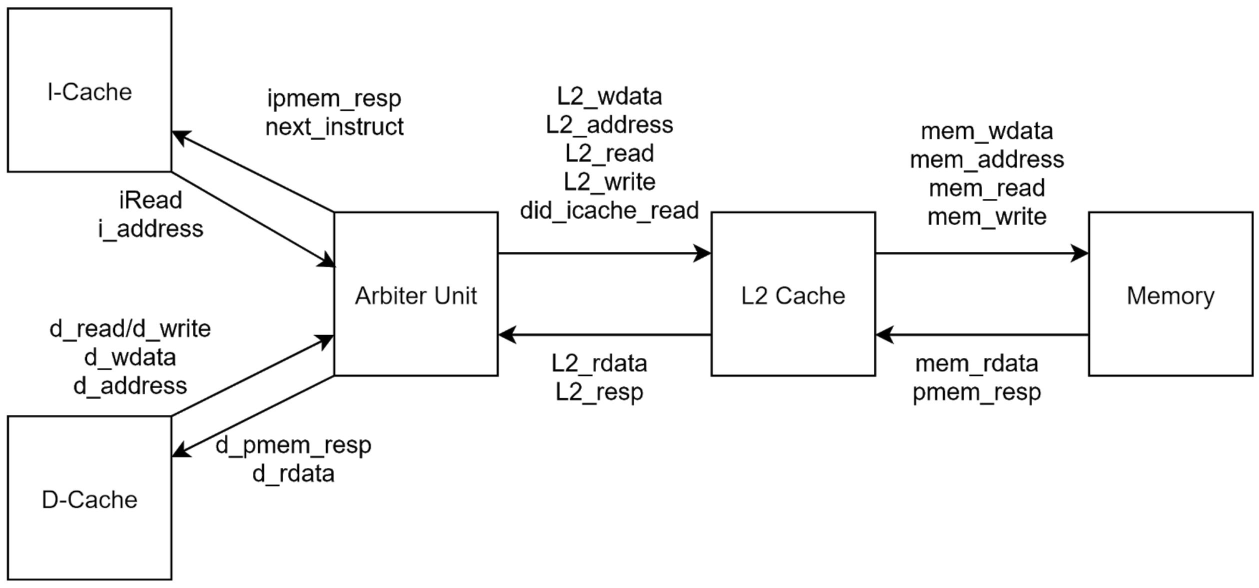

Figure 9: L2 Cache Datapath With Signals

4.3.2 Testing

To test the L2 cache, we tested it as a DUT and made sure the signals passed in and out were correct. Next, we connected it to the CPU and ran cp3 code which tested memory reads and writes. The behavior should be identical to the CPU without the L2 cache. All the code ran perfectly, so the L2 cache was working as intended.

4.3.3 Performance Analysis

To test performance, we ran the competition code with and without the L2 cache and recorded the number of total cache misses. The following table shows the results:

Code	L2 Cache	No L2 Cache
Comp1	31	33
Comp2_i	161	4864
Comp2_m	47	72
Comp3	614	6664

Table 3: Cache Misses for L2 Cache

As the table shows, the number of cache misses went down for all competition codes with L2 cache. The biggest improvements are in Comp2_i and Comp3, with 91% and 97% fewer caches misses, respectively. This shows the importance that a larger L2 cache has with larger and more complex programs with repeating segments of code, or with programs that perform many reads and writes to memory.

4.4 Prefetching

4.4.1 Design

The hardware prefetching was implemented into the L2 cache. We modified the L2 Cache control and Arbiter to have an extra signal, did_icache_read. This signal tells the L2 Cache when an instruction is being read and prompts the L2 cache to automatically perform an additional memory read for the next 32 bytes of instructions. This simply prefetches the next set of instructions so that there will be fewer cache misses. We did not have time to implement a more complex prefetcher, but as the performance analysis demonstrates, there is still a tangible reduction in cache misses.

4.4.2 Testing

To test the prefetcher, we tested it as a DUT. We just made sure that when we set did_icache_read to high that the L2 Cache would perform two consecutive reads from memory. Once we verified that worked, we connected it to our CPU and ran test code and competition code and made sure that everything worked properly. We did have some bugs at first because it would attempt to read every time i-cache read, but we fixed that by adjusting the logic to only prefetch on a cache miss.

4.4.3 Performance Analysis

To test performance, we ran the competition code with and without the prefetcher enabled and recorded the number of total cache misses. The following table shows the results:

Code	L2 Cache with Prefetching	L2 Cache	No L2 Cache
Comp1	10	31	33
Comp2_i	86	161	4864
Comp2_m	30	47	72
Comp3	383	614	6664

Table 4: Cache Misses for Prefetcher

As the table shows, the number of cache misses is reduced by up to 66% with prefetching compared to only an L2 cache. This is a very large performance improvement and demonstrates how many cache misses are due to just fetching consecutive instructions, for both simple and complex programs.

4.5 M Extension

4.5.1 Design

The m_extension was implemented as a supplement to the ALU to perform multiply and divide operations. The multiplier design follows the design given to us in MP1. This device uses bitwise adds and shifts to complete the multiplication rather than repeated adds that would normally be done using repeated adds of various bits through an ALU that would require a multitude of greater cycles. A similar format was followed for with the divide that would essentially subtract and shift bitwise rather than subtracting multiple bits repeatedly. These two designs were originally separated, but were connected with a top file that was treated as the overall M extension. Part of our design required us to add opcodes to the original design that would allow a MUL instruction to be computed using the m_extension.

4.5.2 Testing

For this device we created an entire testing suite that tested the entire device as a DUT for every single possible multiplication and division via the m_extension. Any errors would be manually reported as an error and we could adjust to make sure all divisions and multiplications were being performed correctly. We had an issue where our testing suite would return as X whenever there would be a divide by zero, but we decided to set the value of that to be 0 in the divider and check the dividend prior to the actual calculation to stop it from actually happening.

4.5.3 Performance Analysis

Comp2_i	Comp2_m
1546865ns	1203145ns

Table 5: Timing Difference Between Comp2_i and Comp2_m

Our overall improvement between the competition programs was roughly 22%. This was measured by taking the difference between the two times that we got in between the competition programs.

5. Conclusion

The purpose of this project was to explore various different advancements that can be made to a single core cpu to make it faster. We first parallel pipelined the cpu to accept an instruction every cycle rather than waiting for the instruction to complete. This required hazard detection and data forwarding that allowed the cpu to function safely and properly even with multiple instructions passed in with one every cycle. After we implemented this basic cpu the next phase was to improve our performance on the cpu by incorporating various additional features that would potentially help reduce the runtime. We added an L2 cache in between our L1 cache and memory that was double the size of the L1 cache. Because of the L2 cache we were also able to add prefetching which allowed the time consuming part of fetch to be reduced. Our next time reduction was to add branch prediction to alleviate the issues we would have if we were to fetch incorrect instructions when a branch would occur, we were able to actually incorporate a tournament branch predictor (which turned out to be slower than we expected so we ended up removing it). The next option we considered was the m_extension to try and drastically speed up our competition 2 program. Finally we decided to pipeline our L1 cache to eliminate any possible bottlenecks in memory accesses. Our most notable achievement came from our competition 2 code where we were able to reduce the time from approximately four million nanoseconds to one and a half million. Overall, while this project was quite challenging, it gave us the capacity to think outside the box in our efforts to try and create a faster solution that used as little power as possible.