ECE 411 MP3 Report

Team DedRater
Filip Cakulev, Robbie Ernst, Ricky Machado

Introduction

This report will go over the overarching design of Team DedRater's pipelined processor. It will go into depth into design decisions and overall features implemented. The project was completed to get a better understanding of how a pipelined processor can speed up computation time and how it does it.

It also shows how simple features such as a next line hardware prefetcher can do a lot to performance based on the given test code. Exploration and research on features such as these and figuring out on what kind of code such features would be useful on was also pivotal to this project.

Project Overview

The goal of this project was to create a five-stage pipelined with split L1 caches and a unified L2 cache for RISC-V. Furthermore, additional features, such as a tournament predictor, a prefetcher and an eviction write buffer were at least attempted to be added. With the design of all these features the tradeoff between performance and added complexity needed to be considered.

We decided to assign tasks for each individual on the team at each checkpoint. For larger or more complex parts of the projects sometimes two of us would work on one feature. As the due date for each checkpoint would draw near, we would reconvene and help debug each other's additions as a whole and work on combining everyone’s work for full functionality.

This fell apart slightly when Filip unsuccessfully implemented writeback in the L2 cache bringing down the accomplishments of his teammates a little bit and making the eviction write buffer completely useless.

Ricky however got a working tournament predictor in record time.

Given a chance to redo this project, perhaps assigning more people to work on the cache would be a smart idea.

Design Description

Overview

The next few sections are split up into milestones corresponding with three different checkpoints spread out through the project. We will discuss how the processor evolved during these checkpoints, how we went about testing our features, and any design considerations we made during this time.

Milestones

Checkpoint 1

During this checkpoint we implemented LUI, AUIPC, LW, SW, ADDI, XORI, ORI, ANDI, SLLI, SRLI, ADD, XOR, OR, AND, SLL, SRL, BEQ, BNE, BLT, BGE, BLTU, and BGEU. At this point we had no data or control hazard detection.

To be able to get this checkpoint working, we had to build a 5-stage pipelined processor. This was hard just because you needed a pretty good paper design in order to know exactly what needs to get pipelined and where.

We hooked up the magic memory to the processor for testing the instructions, and were able to successfully run the cp1 code before the deadline. At this point we had a functioning 5-stage processor, missing a few RISC-V instructions.

We were able to pass the autograder. For a more detailed design of the 5 stages see Appendix A

Checkpoint 2

During this checkpoint, we completed all of the RV32I instructions (with the exception of FENCE*, ECALL, EBREAK, and CSRR instructions). We were also required to implement a data cache and an instruction cache hooked up to an arbiter that would feed into a cache line adapter and then finally to main memory.

The arbiter was designed with the L2 cache that was eventually going to be implemented in mind. Designed by Robbie and Filip, the arbiter went through probably the most amount of changes compared to anything else in the project. Its final form consisted of a rather simple state machine that would primarily sit in an idle state and then wait for read or write from the instruction or data cache. It would then wait for the L2 response (or for the purpose of this checkpoint, main memory's response) before it would move on to service a possible waiting request from the other cache or if no request was made it would return to an idle state.

Filip worked on the L1 caches and initially struggled to find a way to make it have single cycle hits. For testing, Filip worked on the Mp2 test code with shadow memory enabled before porting it into mp3 as a split cache. After that testing was focused on the arbiter as the main source of problems on how we would want to prioritize requests and send data from the L1 cache to L2 cache. For a more detailed view of our L1 cache see Appendix B.

Checkpoint 3

During this checkpoint, Robbie hooked up the RVFI monitor to the project somewhat unsuccessfully, and helped Ricky debug the code once we got things hooked up together. Also hooked up the shadow memory and parameter memory, with the new shadow mem file.

For the datapath, we implemented fixing data hazards and control hazards. For the data hazards, we solved those with two approaches. The first was forwarding data further in the pipeline down to instructions not as far in the pipeline. For example if my destination register was being written too and that answer is computed in execute, we can forward that data back to the decode stage if that same register is being used as an operand. There was one case of forwarding not being suitable for preserving the datapath operation, so we had to insert a stall for when we load data from memory into a register then use that data in a read in execute. For control hazards, we implemented a static non-taken branch to eliminate control hazards. If br_en was 0, nothing happened, but if br_en was 1 we had to jump to the correct pc address and throw away the work that was being done since it is invalid work.

Additionally, a 2 way 16 set associative cache was added to act as our L2 cache. This cache was made similarly to the L1 cache design but did not include single cycle hits. Testing was weak in this department, as we solely relied on the given testcode and shadow memory to insure it worked. This would come up later to bite us as it turned out

writeback was not working correctly but this would only be discovered after the addition of the eviction write buffer.

Advanced Design Options

Tournament Predictor
Design

The Design of the predictor was pretty complicated. We made a predictor that picked between a local and global predictor based on the address of the instruction. What would happen is that the local and global predictor decide which way to branch based off of past branches. The local just takes the address to index into the pattern history table and the global predictor XORs the address and the last 10 branches to index into the pattern history table. Each index in both tables has a 2-bit counter whether it wants to take the branch or not. Then after the global and local history table have their predicted outcomes, the tournament predictor picks which predictor to use by again indexing into another separate table. The tournament predictor will then update every table and global history register by updating the state of each counter and register.

Testing

The Tournament Predictor was able to be tested in two different ways. Firstly I ran the cp3 code to make sure that each pattern history table was being written too and changing accordingly. The other way I tested if the tournament was working was by making a counter for every branch happening and making a counter for everytime a branch was wrong. By doing this I got about a 97% branch prediction accuracy rate. Each branch predicted correctly saved 120ns assuming 100% cache hits

Performance Analysis

For the cp4 tests code we have a 97% accuracy with 6621 branches total. Each branch predicted correctly saved 120ns assuming 100% cache hits. This saved about 771,000ns for all of cp4

Prefetcher

Design

The design of the basic hardware prefetcher is pretty simple. As said in the documentation “ This approach initiates a prefetch for line `i+1` whenever line `i` is accessed and results in a cache miss. If `i+1` is already cached, no memory access is initiated.” The module is placed in between the L2 cache and the physical memory, and forwarded the L2 cache signals when appropriate. The module would then retain the prefetched data line, and the most recently read address.

Testing

After reviewing the waveform to make sure the prefetcher was working as intended, Robbie set up a counter in the test bench to count the amount of times the L2 cache pulled a prefetched line from the prefetcher instead of having to wait for a memory read. Each time there is a hit in the prefetcher you save a pmem read cycle. However, the time saved varies on how often you jump from instruction to instruction.

Performance Analysis

Running the CP4 code, we have 97 prefetcher hits and the code takes 2731565 ns to run. Without the prefetcher the code takes 2749635 ns to run. So on average we save ~186.29ns per prefetch hit.

Eviction Write Buffer

Design

The design for the eviction write buffer was made by Filip and was designed to be based off of L2 cache’s signals to main memory. It would sit in between the L2 cache and main memory as such. It would hold dirty evicted cache lines and allow for immediate use of the now free line instead of waiting for a response from memory. This would allow for the buffer to write to main memory while the L2 cache could read into the now clean line.

Testing

Upon testing the eviction write buffer on the cp3 test code it was found that it actually slowed down the time to completion of the code. Upon analyzing waveforms and through use of cp4 test code, it was discovered that the writeback on the L2 cache was not working correctly as lines were never becoming dirty and as such it saw no reason to evict any lines making the implemented write buffer completely useless. It was at this point that the eviction write buffer was put to the wayside to work on more pressing matters. However, when it was added to Mp2 code and its test code, it did in fact speed it up, so it is possible that it was working as intended.

Performance Analysis

As mentioned before the performance technically was lowered upon the addition of this feature but a more thorough analysis was never done and it would be pointless to give any real data points in this section.

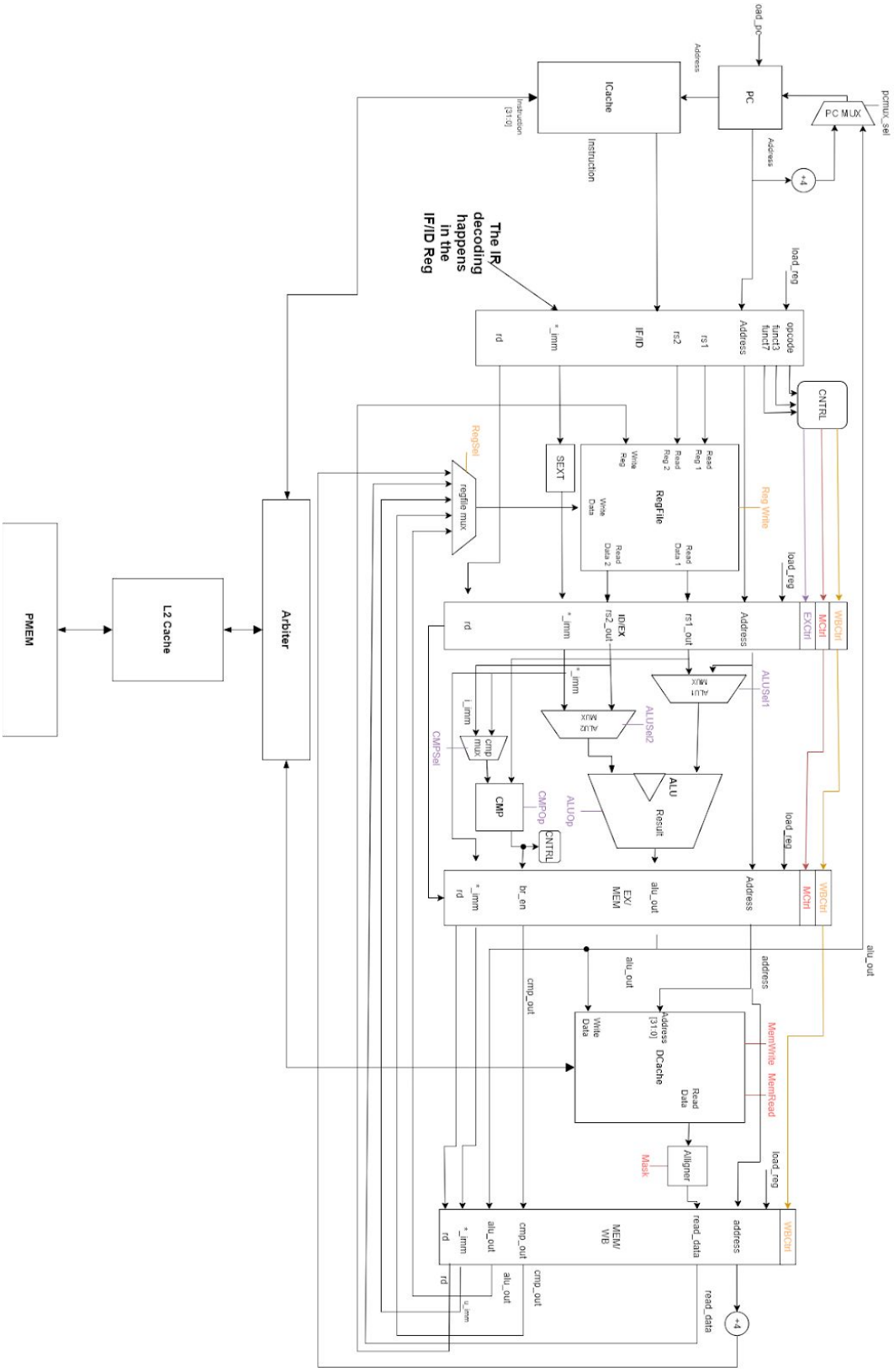
Conclusion

While we successfully completed designing and implementing a 5 stage processor with split L1 caches and data and control hazards with a working tournament predictor and prefetcher, it did not have a working L2 cache and as such we can not say that we were completely successful. The process for accomplishing said tasks was unlike many before and required a different approach to testing and designing, requiring elaborate paper designs and meticulous research before actual coding.

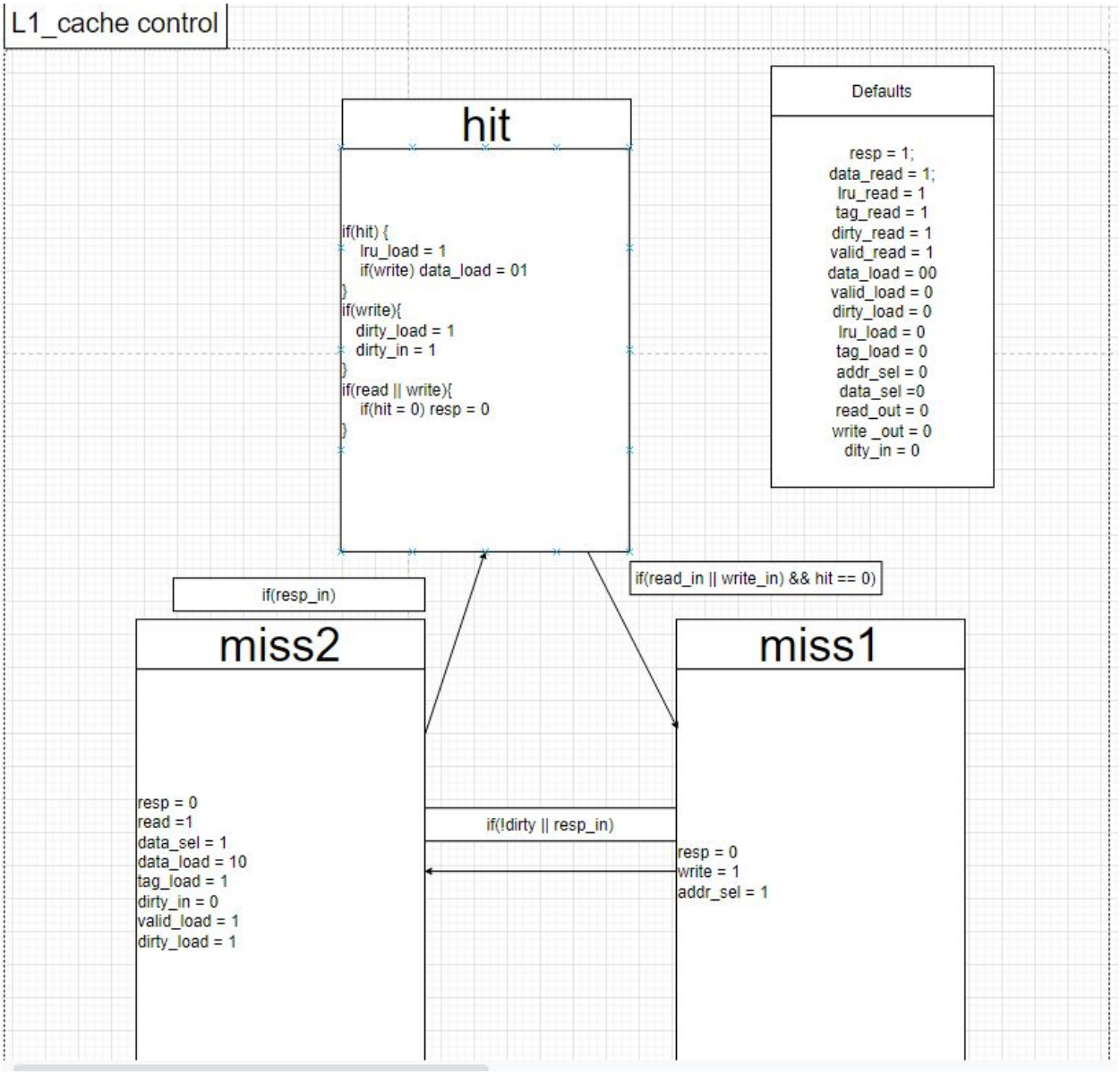
Given more time we would have hopefully fixed the L2 cache and tested our eviction write buffer in depth. We were also considering pipelining our L1 caches and even attempted to do so but slightly misunderstood the two stages and ultimately were not able to get it working. We also intended to create a return address stack for our processor as it seemed like a good way to speed up JAL with minimal work and design from our end.

At the beginning of this semester, our group did not really know what we were getting into when taking 411. And while COVID-19 did slightly hamper our project by killing off some motivation to work on things, we believe it is safe to say that a lot was learned from this class and we are better off because of it. A lot was learned on aspects of design and how it should be done. We learned of the importance of written designs before actual implementation. We learned how to more efficiently program in system verilog and put a lot of hours in the language under our belts. Finally, we learned how to work better as a team and hopefully learned from our mistakes made throughout this class.

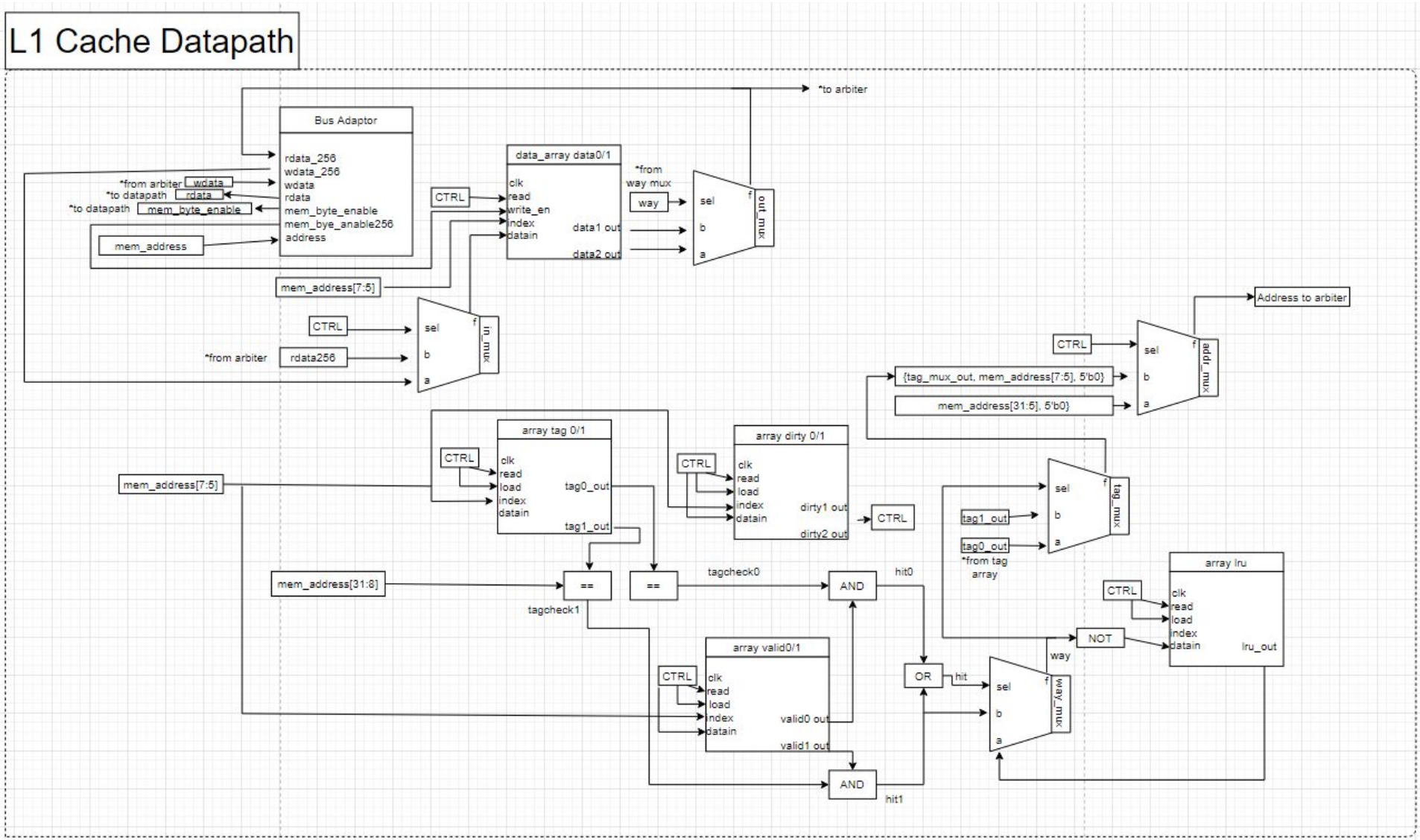
Appendix A : Pipelined Datapath



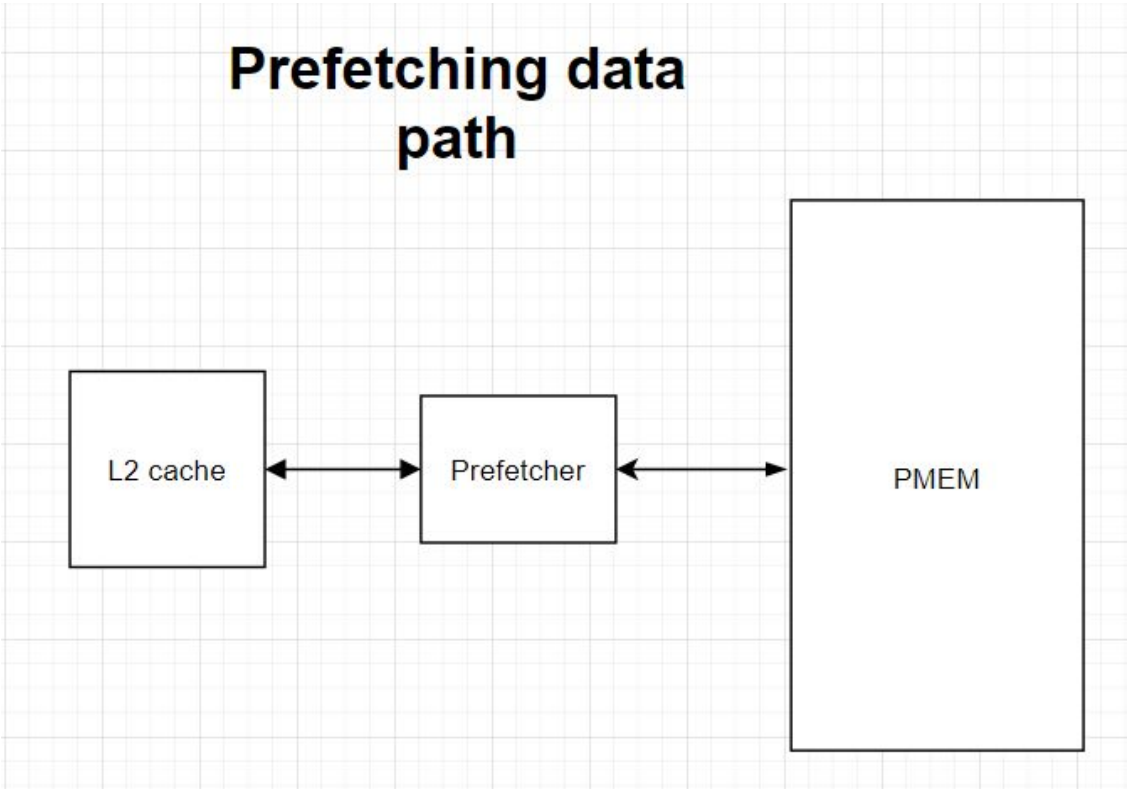
Appendix B : L1 Datapath and Control

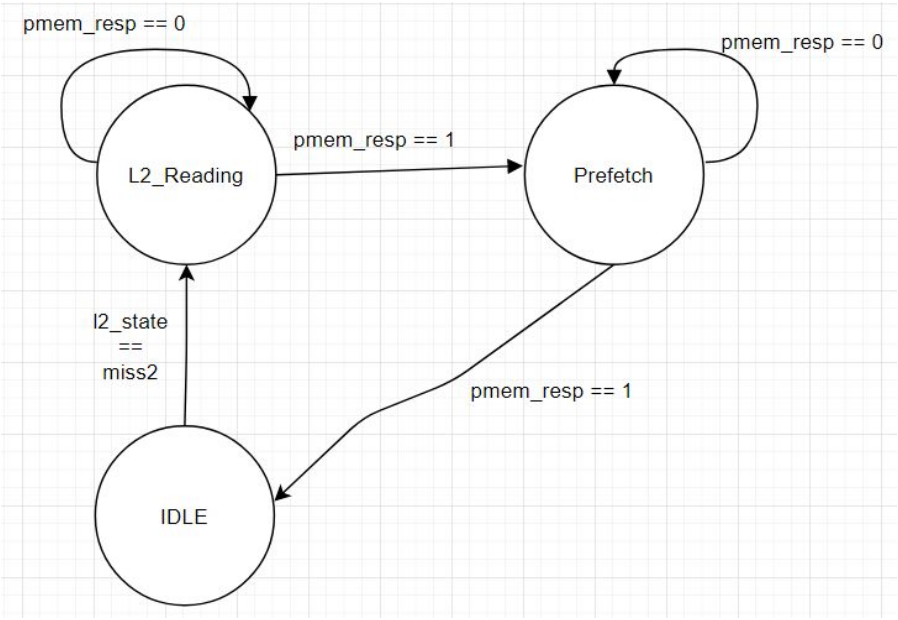


L1 Cache Datapath

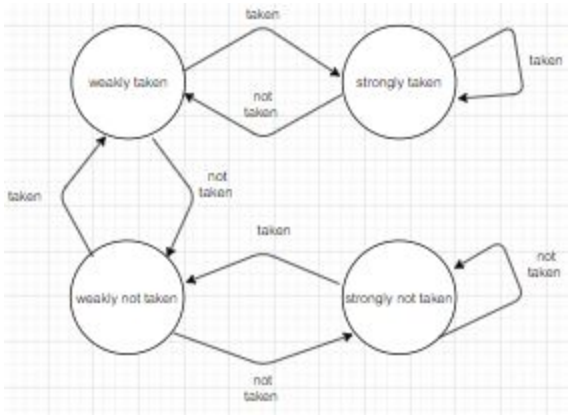


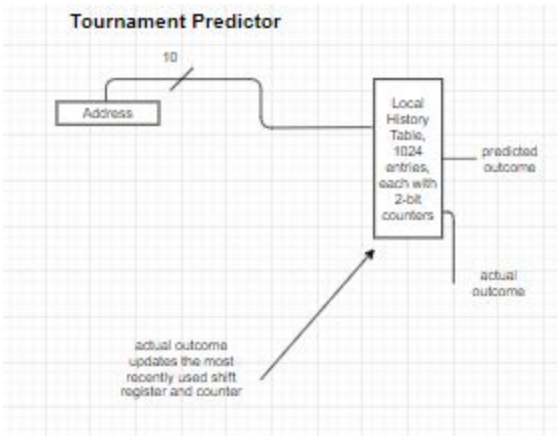
Appendix C : Prefetcher Datapath and Control

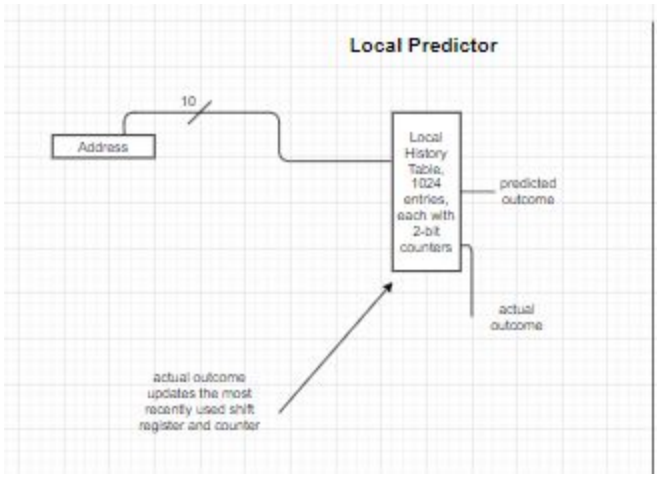


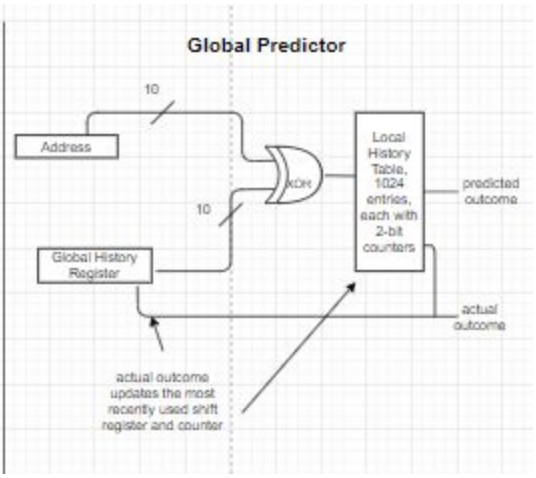


Appendix D: Tournament Predictor


taken
weakly taken
strongly taken
taken
not taken
taken
not taken
taken
weakly not taken
strongly not taken
not taken
not taken


Tournament Predictor
10
Address
Local History Table, 1024 entries, each with 2-bit counters
predicted outcome
actual outcome
actual outcome updates the most recently used shift register and counter


Local Predictor
10
Address
Local History Table, 1024 entries, each with 2-bit counters
predicted outcome
actual outcome
actual outcome updates the most recently used shift register and counter


Global Predictor
10
Address
XOR
10
Global History Register
Local History Table, 1024 entries, each with 2-bit counters
predicted outcome
actual outcome
actual outcome updates the most recently used shift register and counter