Name: Steven Lee, Scott Liu, Benoit Ortalo-Magne
Net ID: solee2, scottl4, beo2
Team: Hydration is Key

ECE 411 MP4 Final Report

Introduction

Over the course of the last few months, our team has created a pipelined microprocessor. This paper will outline an overview of the project, followed by more specific descriptions of each component and checkpoint that was completed along the way. By developing our final project, we learned the inner workings of a microprocessor including how the instructions are processed and how to use optimizations to improve overall performance.

Project Overview

A pipelined microprocessor uses several stages to execute parts of instructions in parallel and therefore speed up execution time. The end goal of this project was to successfully handle all instructions in the RV32I instruction set. We began our implementation by constructing a pipelined microprocessor that handles all required instructions. Once working, our team added hazard detection, data forwarding, and a basic cache system. This step completed, we focused on further improving speed and efficiency by adding design options of our choice. Until the last portion, we chose to work through the requirements together as a group, meeting in person and online. Since the optimizations of our choice lent itself well to splitting up work we then decided to work separately and merge changes once complete.

Design Description

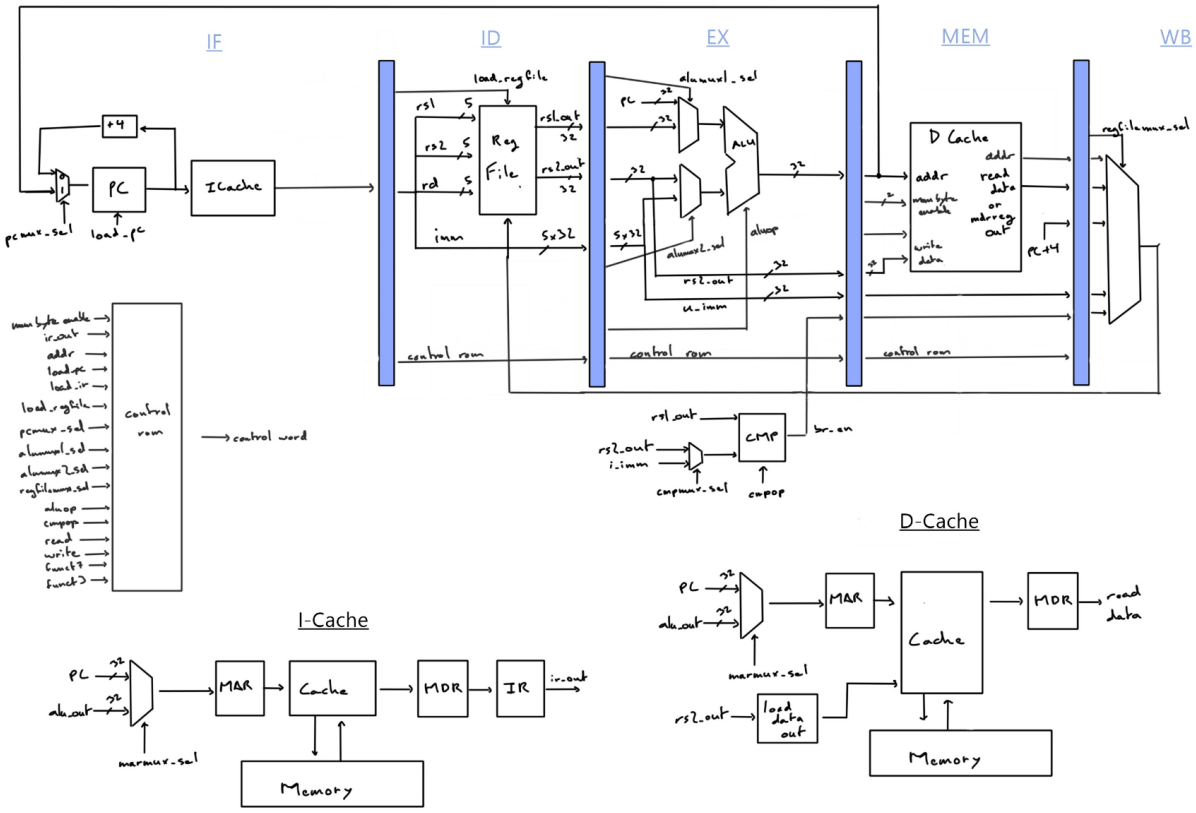
Overview

As of the final submission, our project included a fully functional pipelined processor with hazard detection, data forwarding, and cache. As our choice of further improvements, we chose to alter the basic cache system to have a pipelined L1 4-way cache and an L2 cache.

Milestones

1. For milestone 1, we created a basic pipeline that handles all RV32I instructions. We implemented this by making five stages the instruction must go through before being finished. FETCH, the first, gets the instruction and updates PC. Next, DECODE deciphers the instruction so the microprocessor knows what to

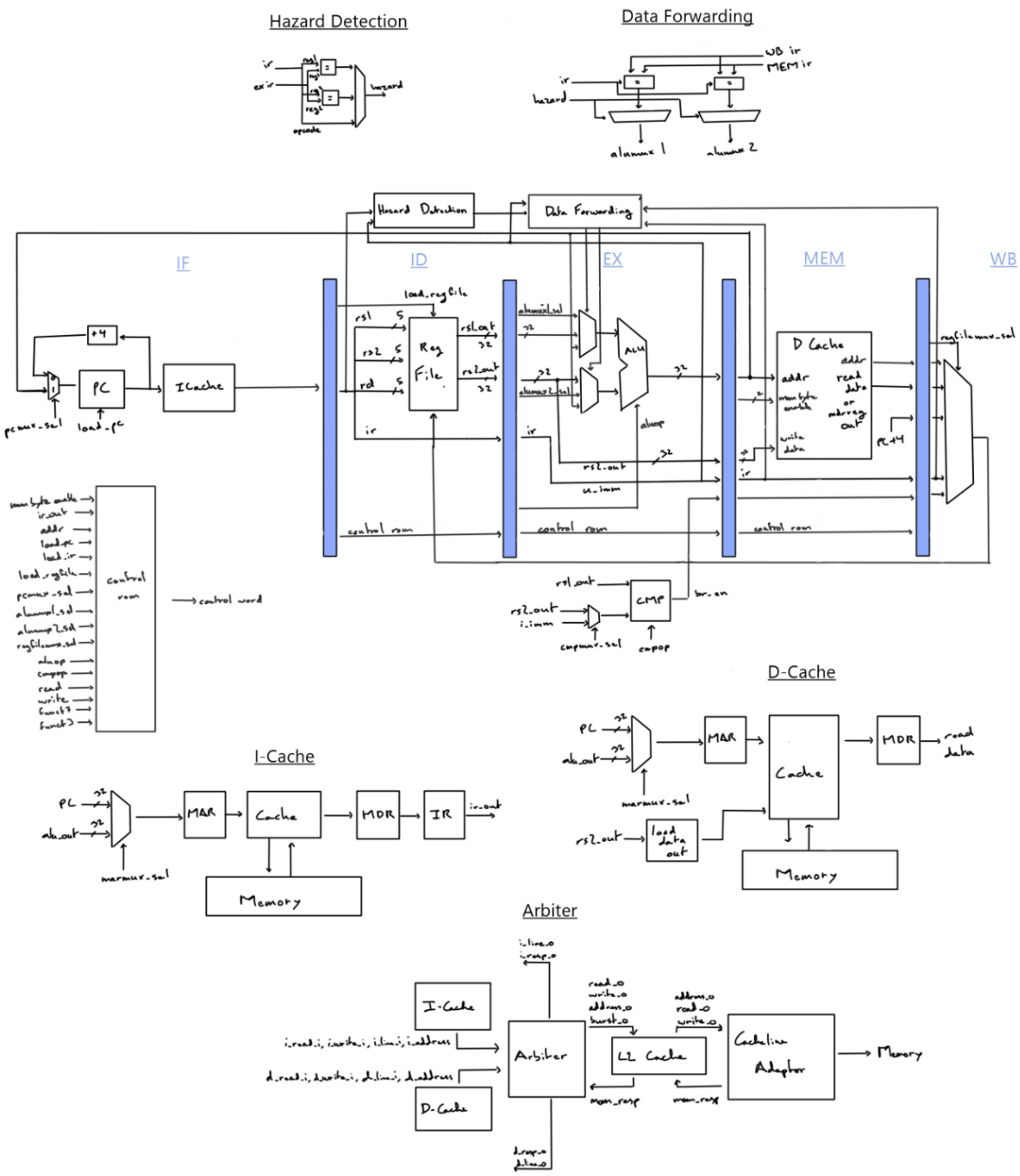
do. EXECUTE then carries out the operation. This can include add, and, or, store, load, and other. MEMORY stage accesses the caches and memory as needed for any instructions that deal with memory. Finally, WRITE BACK stage means the instruction is finished and the data is stored and ready to be used in other instructions. Testing was carried out by writing test code around each instruction and verifying that it's output was correct using waveforms. This checkpoint did not include data forwarding or hazard detection. We also used a "magic memory" to avoid needing to implement cache hits and memory stalls.



Milestone 1 Datapath

2. Milestone 2 involved adding data hazards and forwarding along with a cache and memory system. In a pipelined processor, data hazards occur when a value is needed before it is ready. For example, since instructions execute in stages, if an instruction stores its result in x1 and the next instruction wants to carry out an

addition with x1 and x2, the result of the second instruction would be incorrect. One option would be to simply stall the pipeline until the first instruction has finished processing. Instead, we can implement data forwarding. By forwarding the calculations from memory and write back stages, we can effectively use the results of instructions before they have completed. The cache system also helps speed up execution by placing recently used memory in an instruction and a data cache. This way, the processor doesn’t need to wait for memory accesses each time it needs to carry out an instruction and can instead use the significantly faster caches.



Milestone 2 Datapath

3. Checkpoint 3 required us to implement design optimizations of our choice to continue to optimize our processor. We chose to attempt a pipelined L1 4-way cache, an L2 cache, prefetching, and branch prediction. These are further described below.

Advanced Design Options

1. Cache Optimization
    a. *Design*:
        i. Scott's role in the checkpoint 3 design was figuring out how the pipeline the L1 caches for instructions and data reads and writes. He started out avoiding using states in the cache, hoping that combinational logic in the datapath would take care of all the execution. This resulted in an increased difficulty and complexity in debugging, and after 3 different implementations of no state machine, he decided to give in. While still using 2 stages and one state machine, Scott split up the L1 cache datapath by reading in the parameters given by the CPU, and taking care of all the information processing through the L1 cache control.
        ii. Scott also increased the number of cache ways in the L1 cache from one-- which was provided from the git release, to four. To solve the LRU logic, he made each entry in the LRU array an 8 bit word, with 2 bits representing the cache way number. On an LRU load, the most recently used number would be inputted, and the LRU array would replace the MSU entry-- or the most significant bits-- with the input, shifting the rest of bits in the entry accordingly. The least significant bits would then represent the LRU cache way, which was outputted rom the LRU array.
        iii. The final thing Scott designed was the L2 cache. This was relatively easy compared to the other two tasks, since theoretically it was just another cache that took information and requests from the L1 cache and made communications with the memory for fetching on misses or dirty data. He took the provided cache from the code given by the course and made some adjustments so that the write mask was always high on every bit, and the data was always 256 bits, since cache between cache communication is different than CPU between cache.
    b. *Testing*:
        i. Scott started testing his design by using the CP2 testing code, since it was more simple and he was more familiar with how the

states and register file data should behave at each point of the test. Once the design successfully passed the CP2 tests, he ran it on the CP3 test code, and was not able to receive the expected end result that the test code gave. Thus, he took chunks of the CP3 code, and debugged using the ModelSim waveforms to see how the caches would perform given different situations, including multiple hits and misses for lw's, sw's, sw after a lw, and lw after a sw. He found that a missed sw following a lw created a combinational loop in his design, preventing him from further debugging the cache, which created the biggest problem for him during this checkpoint. After realizing this, he wrote his own test code for each test case for the pipeline cache, which included continuous load hits, continuous store hits, loads after stores, stores after loads, using all four cache ways, writing back to L2 cache, and writing back from L2 cache to memory. After individually coding these test cases, he found that load_regfile was set to the incorrect value in the control_rom in the cpu_datapath. Turns out, this small bug was able to slide past the CP2 test cases, but when it came to more scenarios included in CP3, it became an issue, loading don't cares into data arrays and infinite loops.

c. *Performance Analysis:*
    i. In Checkpoint 2, we used a cache provided by the release code, which included 1 cycle hits. We also combined the arbiter with the cache line adapter, which allowed for the memory response delay to be shortened. In checkpoint 3, since it was a pipelined cache, it would take 2 cycles to hit, and even longer on misses since there were multiple states in the cache control to make sure the pipeline would function correctly. The L2 cache also created an increased delay, since on complete misses for L1 and L2, the processor would need to write data into L2 first before writing into L1. The arbiter and cache line adapter were split into separate modules, which would cause an increased delay in providing the memory response from memory to the cache line adapter, then to the L2 cache, then to the arbiter, then to the L1 cache, and finally back to the CPU. We believe this is what caused the pipelined cache to take longer to finish executing on the CP2 checkpoint code, since both test codes do not rely much on the L2 cache for better performance due to the lack of data writes and reads.
    ii. Also something to note was the original design without any advanced features did not completely work on the CP3 code. We

tried changing it up, and the best we got was it executing at just under 2,000,000 ns. However there is a significant difference in data misses and instruction misses. We believe that despite it not working, increased L1 cache ways and L2 still demonstrates to be useful since it holds more previously written addresses, allowing less fetches to memory to be made, which is also demonstrated in Scott's cache test cases when the test code writes to the same index but different tags 10 times. In a normal one way cache, every new store would cause the processor to write back to and fetch from memory 9 times, but given a 4 way L1 cache and L2 cache, the processor would only write back to the memory once, given Scott's test code.

iii. With the caches provided by the release code:
    1. CP2 test code
        a. I-cache miss count: 8
        b. D-cache miss count: 2
        c. Runtime: 4825 ns
    2. Scott's test code:
        a. I-cache miss count: 8
        b. D-cache miss count: 14
        c. Runtime: 12565 ns
    3. CP3 test code:
        a. I-cache miss: 2215
        b. D-cache miss: 670
        c. Runtime: 1,872,462 ns

iv. Pipeline cache + L2 cache
    1. CP2 test code:
        a. I-cache miss count: 8
        b. D-cache miss count: 2
        c. Runtime: 5355 ns
    2. Scott's test code:
        a. I-cache miss count: 8
        b. D-cache miss count: 10
        c. Runtime: 11555 ns
    3. CP3 test code:
        a. I-cache miss count: 91
        b. D-cache miss count: 61
        c. Runtime: 153165

2. Prefetching

a. *Design*: Benoit worked on basic prefetching for this checkpoint. After researching the best way to implement it he decided that directly modifying the cache would be best. By adding extra states to the state machine, the cache control could be modified to automatically fetch for the address + 4 after each memory read. This works well because on a hit the cache will bypass the prefetch, but on a miss it can use the exact same protocol it uses to fetch from memory normally, thus minimizing overhead and code complexity. Merging with the new cache systems proved difficult.
b. *Testing*: Prefetching was simple to test as the given test code could be used. Simply look for a cache miss, check that the prefetch activates, and once finished make sure the cache holds the data that was fetched. Test code was eventually adapted to fetch more precise addresses that should result in hits and misses to simplify the process.
c. *Performance Analysis*: The advantage of prefetching is that expensive and time consuming memory accesses can be done while code that doesn't require memory is executed. This is advantageous for our processor because it often performs sequential memory accesses so the speedup should be significant. Our processor completes around 4 instructions, which takes 4 cycles, per i-cache miss. With prefetching, those 4 cycles could be taken off the total time it takes for a cache to complete a miss and get the correct data from memory.
d. *What Went Wrong*: Unfortunately this optimization wasn't finished. Although it was mostly working, functionality for fetching from memory while other instructions executed was missing. I believe this had something to do with how our cache system treated mem_resp and other memory signals.

3. Branch Prediction
    a. *Design:*
        i. Steven worked on implementing the Global 2-Level Branch History Table for the advanced design options. He was able to add a new module for branch predictions which took signals clk, rst, predict_pc (from output of pc register in IF), write_pc (pc output from state register EX_MEM), read, write, taken, and outputted a signal called prediction. The clk, rst, and pc signals are self explanatory, but as for read, write, taken, and prediction, here is a brief description: read is inputted to the module from stage IF and indicates that it is trying to read a prediction from the module; write is a one bit logic signal that is outputted from state register EX_MEM and it indicates that it wants to write to the our module data structures (incrementing/decrementing the pattern history table based on

signal taken); taken is the output from the cmp register which shows if the branch instruction was taken or not. Also, to clarify, there are two pc's -- write_pc and predict_pc -- because when the module is predicting it should use the predict_pc coming IF, and when the module is writing, it should use the write_pc which could be altered in the ID stage depending on the prediction (could be target address).

ii. The branch prediction module essentially was supposed to use the predict_pc from the IF stage, hash it using the branch history register (BHR), and use this as an index into the pattern history table (PHT). This is according to the Gshare model for a more precise prediction. The PHTs hold a 2 bit counter which increments if the branch was predicted correctly or decrements if the branch was mispredicted. This method is very useful because the PHTs are indexed using the hashed value from the predict_pc and BHR which correspond to a unique history per unique hash value rather than having all branch instructions share one pattern history. This value in the PHT is outputted as a prediction (00 -- strongly not taken, 01 -- not taken, 10 -- taken, 11 -- strongly taken). Once the pipelined instruction reaches EX where the CMP register outputs the comparison result from the branch instruction, it is known whether the branch mispredicted or not. The branch prediction module is able to use this CMP output to update the PHT and BHR which make it more accurate. The BHR is left shifted with the value of the one bit logic variable "taken" appended on to the LSB.

iii. By implementing this method, the pipeline would have an accurate prediction as to whether or not a branch instruction is taken or not by stage ID which overwrites the PC directly according to the prediction. This saves cycles and thus increases performance. For example, in a loop, the branch prediction module would quickly learn the pattern of the branch instructions and begin to predict them extremely accurately which progressively saves more and more cycles.

b. Testing:

i. To test the branch prediction module, simple assembly test code that involved branching so that in-depth analysis could be performed on the input and output signals to the module as well as the data in the modules, namely the hashed index (from write_pc or prediction_pc and BHR) and the PHT entries via the modelsim software/waveforms. A big problem was that it seemed to "plow

through” the code, ignoring the branch instruction altogether. Hardcoding the initial predictions allowed more analysis on this phenomena and eventually the branch predictor was working only if it predicted incorrectly while if it predicted correctly and set the pc in the ID stage to the target address, strange behavior occurred and it was concluded that it could be setting a signal from the wrong stage or state register, or missing some offsets somewhere in the pipeline because it would not branch correctly. The textbook “Computer Organization and Design” by Patterson and Hennessy was referenced in order to help combat the issue (had to fix flushing; page 318 of the textbook was referenced), especially around Chapter 4.5-4.8 and the Dynamic Branch prediction text (page 321).

c. Performance Analysis:
    i. The 2 bit PHT scheme takes 2 mispredicts in order for its prediction to be changed. This is advantageous in many cases because a more volatile 1 bit scheme may mispredict more often by being more ambiguous and also cannot predict longer patterns as the 2 bit scheme is capable of. This is important because the higher the accuracy of the branch predictor’s predictions, the more cycles are saved per branch instructions. Although mispredicts take more cycles, learning the pattern of the branches and being able to predict them more accurately is a trade-off that is well worth the cost.
d. What Went Wrong:
    i. This optimization did not end up completely working, unfortunately. The code was being tested on a previous git commit that was allegedly working, but turns out was not. This led to git stashing the changes and branching off a commit that was working (there was some code that our team thought was working, but were mistaken, throwing off some of our organization a bit). This extra time could have been spent debugging. Additionally, the errors at hand were tedious and some insight from the TA should’ve been requested rather than persisting further. The idea and design should’ve been correct but there were likely a few errors nonetheless that caused this optimization to fail in the end.
    ii. Better debugging could have been performed when the new cache features were being performed. Scott implemented 4-5 different designs, and still could not find the issue to the same bug that occurred over all the designs. Turns out, the bug was caused by an

error in the cache line adapter, not in the cache. This led to a whole week of time used on implementing designs that could have been used on debugging further bugs in the checkpoint, which ultimately led the team to not be able to pass every single test case in the MP3 testcode, despite working for over two weeks just on the cache features themselves.

Conclusion

A significant focus of ECE 411 is pipelined processors, and over the last few months our team has managed to create one from scratch. Each checkpoint we completed brought more features and real world benefits to our code. We were able ot get a deeper and practical understanding on how increasing the number of cache ways and adding an L2 cache would benefit the runtime and performance of the processor. When it was finally time to hand in the final version, we had obtained a deeper understanding of how a pipelined microprocessor and certain optimization techniques function. This was a very interesting assignment, since we were able to see in first person how the pipeline states communicate with each other, and different debugging techniques to find imperfections in the design. With the amount of debugging, designing, implementing, and communicating put throughout the semester, this programming assignment spawns an experience that is definitely not easily forgotten after the course is over.