ECE 411 MP4 Final Report

Pipelined RV32I Processor

Anthony Nguyen & Hemang Nehra

TA: Victoria Colthurst

Introduction:

The goal of this project was to design and implement a pipelined RISC-V processor using Systemverilog and simulate the design using Modelsim and Quartus. The processor is designed to execute the RV32I instruction set (with the exception of FENCE*, ECALL, EBREAK, and CSRR instructions). Due to the nature of a pipelined processor, the design is required to handle data hazards that occur due to data dependencies in the code. Over the course of the project, we applied pipelining techniques learned in class to better understand how a pipeline works. After implementing the basic pipelined processor with data forwarding and hazard detection, our next goal was to improve the efficiency by applying optimizations such as prefetching and changing the memory hierarchy to improve memory latency.

Project Overview:

The project was broken up into 3 checkpoints. The first being the basic pipeline without data forwarding and hazards. The second being able to support data forwarding, hazards, and integrating both caches. The final checkpoint was where we were able to freely include any features that can improve the performance of our pipeline. The work was fairly straight forward when it came to implementing the pipeline. The design and functionality of the processor was worked on by both members. We believed that if we had the same idea and understanding going into the implementation of the design, it would be much easier to collaborate when adding each module together. For debugging, we checked each other's implementations. We found that when someone worked on a functionality, they tend to overlook some things that another eye can easily find. Generally, when we were stuck, we would try to trace waveforms from Modelsim and find areas in the simulation that are incorrect or unexpected. We also utilized the RVFI monitor provided to us. Being able to see when and what signals are incorrect during the execution of the code was very beneficial since it allowed us to pinpoint where errors could be.

Design Description:

The pipelined processor was primarily implemented from our earlier course work, MP2. This MP was to implement a non-pipelined RISC-V processor that also executes the RV32I instruction set. Before writing code, we took this design and modified the datapath into 5 stages (fetch, decode, execute, memory, and writeback) for instruction execution. The goal of a pipelined processor is to increase the throughput of instructions; however, the non-pipelined processor design is only able to execute a single instruction at a time. To compensate for this, we needed to add monolithic registers in between each of the pipelined stages to hold values for the current instruction as it makes progress. To allow every instruction to use different hardware in various stages of the pipeline, we created a Control ROM that generates simple control signals (mux selects, ALU operational codes, load signals, etc.) for other hardware in the processor. These controls signals follow the current instruction being executed down the pipeline until the instruction finally finishes.

We designed each component of the pipeline in a modular fashion. This allows us to easier modify certain parts of the pipeline if need be. Each of the 5 stages of the pipeline were implemented in their own modules that included smaller hardware components that were required for that stage. For example, the execute stage required using the ALU and Comparator to perform calculations and those components are included in the overall execute stage module.

Alongside the processor, we also had a memory hierarchy that contains two L1 caches and the physical memory for data and instructions. The caches were provided to us from the course. The split caches (Icache and Dcache) are connected to the processor in the fetch and memory stage respectively. The caches themselves are connected to an arbiter that controls where physical memory is sending and receiving its data. The physical memory we used sends 4 bursts of 64Bytes. However, since our processor is 32-bits, we had to use an adapter that connects the processor to both the caches. The caches will take the 256-bits sent from memory and send 32-bits of the cache line at a time.

Milestones:

1. Checkpoint 1:

Overview:

For this checkpoint, we had to implement the basic pipelined processor. This included all 5 stages (fetch, decode, execute, memory, and writeback), the monolithic registers, or barriers, between the stages (IF_ID, ID_EX, EX_MEM, MEM_WB), and connect the processor to a "magic" memory that generates both instructions and data to the processor. This magic memory will later be replaced by our split caches and physical memory.

To start, we first created a paper design based on our previous non-pipeline processor in MP2. We took each component and put them into stages of the pipeline where they are required. For example, the PC register will go into the fetch stage and ALU and comparator (CMP) will be required in the execute stage. After getting the basic design of the pipeline on paper, we traced through to follow how an instruction will be carried out through the processor.

For implementing the design, we reused several files and modules from our non-pipelined processor MP. At this point of the project, we were taking pieces of the non-pipelined processor and reorganizing it such that it becomes pipelined. In a processor, there is an IR, instruction register, that holds instructions that are fetched from memory. We chose not to include this in our pipeline since the Icache will provide the instructions and that instruction is decoded into control signals in our ROM. For our barriers, we had to decide what information the instructions required at each stage. To do so, we traced some basic instructions like ADD and AND to get some insight on what data would be required. Hazard detection and data forwarding was not required at this point of the project which simplified how we handled instruction executions.

Testing:

To test our design, we were given assembly code to compile and write to memory. We then ran the code through Modelsim and monitored specific signals to ensure that they are what was expected, and more importantly, what matched the dissembler. Since this was the first time we are experiencing working with a pipeline processor, it took us a while to understand what signals needed to be sent through the pipeline, but more importantly, when they needed to be sent. One aspect that troubled us was what stages of the pipeline some signals must come from for the next instruction to properly execute. For example, the BR_EN (branch_enable) signal that is wired from the CMP to the PC mux select. We struggled to figure out where this signal should come from, execute or 1 cycle after execute (memory stage). We eventually figured it out with help from our TA and going through Modelsim waveforms.

To verify that our design is working properly, we wrote basic assembly code, and used provided testcodes, and tried to include as many instructions as possible to fully test our design.

Since forwarding and hazard detection was not implemented yet, NOP (no-ops) were inserted to give instructions time to make it through the pipeline and write to the register file before another instruction starts. At this time, we were using "magic" memory that would generate instructions and data as a replacement to our split caches and physical memory.

2. Checkpoint 2:

Overview:

After implementing the basis of the pipeline, we now had to incorporate the memory hierarchy into the design, as well as handling hazards, data forwarding, a static non-taken branch predictor, and an arbiter design to interface between physical memory and our split caches.

We first began by designing the arbiter as a state machine that passes signals to one of the split caches from and to physical memory. Since physical memory has only a single port, both caches may not interact with memory at the same time. Our design for the arbiter prioritized giving the Icache control of the arbiter over the Dcache. We chose this because not all instructions will require data from physical memory, but instructions are always being fetched from memory. Using the provided cache, we created the split caches and connected those caches to the appropriate stages of the pipeline.

After testing that the arbiter is functional, we started working on data forwarding and hazard detection as well as including stalling logic. Before we began, we drew out a basic design on what information needed to be passed to determine where data should be forwarded. Our processor needed to include MEM→EX, that is, forward data from the memory stage of the current instruction to the execute stage of the next instruction, WB→EX, and WB→MEM. Using the information we learned from the course, we had to check if the destination register from the current instruction is being used as a source register of the next instruction. These conditions are checked each of the forwarding paths listed previously. Depending on the stage, different data will be passed to the next stage, for example, MEM→EX will need to forward the ALU result of the memory stage to one of the inputs of the ALU in the execute stage. This will allow the updated results of the destination register, RD, to be used as a source register, RS1 or RS2, of the next instruction. We integrated 2 forwarding units, one for execute and one for memory, into our processor. These units would generate mux selects for new muxes that took values from different stages of the pipeline and passed them through to the preexisting muxes. For example, the execute forwarding unit would generate mux selects for the ALU mux1 and ALU mux2 selects. If there was a hazard, data forwarded from another stage to the inputs of the ALU.

Along with this, we included a stalling unit that would stall the pipeline if there was a data dependency when the current instruction is a Load instruction and the next instruction used that result. This unit contained logic that would stall the entire pipeline so that the Load instruction can write to the register file before allowing the pipeline to make progress. To stall

the pipeline, all registers' load signals are set to "0" so their outputs would not be updated for the next instruction.

For the static non-taken branch predictor, we always start fetching the next instruction after the branch instruction. When the BR_EN signal goes "high", we "squash", by raising the reset signals on registers, that fetched instruction and allow the program to branch to the calculated address.

Testing:

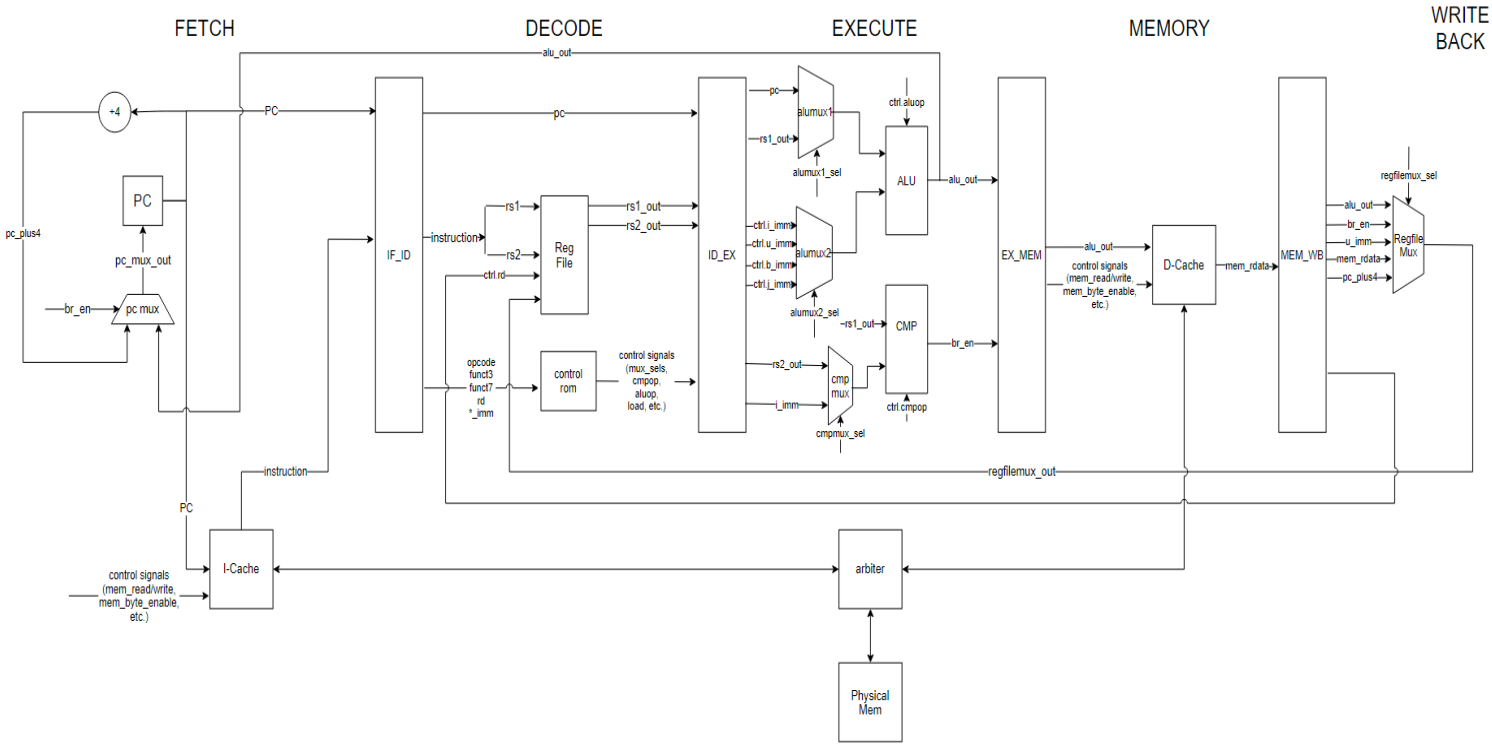
Testing our design at this stage was very time consuming and much more difficult than the previous checkpoint. Since we were adding multiple new additions to the pipeline, we needed to ensure that everything worked together properly. We made sure that, for the most part, each new addition worked on their own before integrating them into the pipeline.

To test our arbiter, we connected the caches, arbiter, and physical memory to create the memory hierarchy as a separate unit from the processor. We then wrote tests that would send values to the caches and monitor if the data returned from memory was correct. In our testing, we wanted to see if the arbiter prioritized the Icache like we wanted, and made sure that the state machine transitioned correctly. The memory hierarchy was then connected to the pipeline and we ran the provided testcode from the previous checkpoint to ensure that the pipeline still functioned correctly.

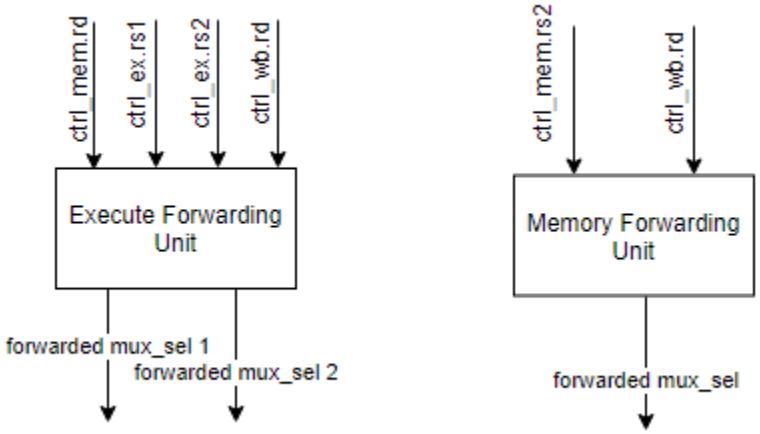
After being sure that the caches worked properly, we had to integrate and test our forwarding and stalling units. This was the biggest hurdle for us at this checkpoint because even though the units themselves are simple, integrating them took some time. The forwarding units were working properly, but we were overlooking some basic changes. For example, all dependencies of RS1 or RS2 needed to be changed to have a forwarded option. At the time, we did not see this until we started debugging and some instructions were not being forwarded correctly. We took time to double check our code and ensure that all instances of source registers had a forwarded option. After doing so, and using Modelsim to monitor their values, we were able to complete our forwarding.

To test our stalling logic, we wrote very simple code that created data dependencies with a Load instruction and any instruction that used that register as its source. One issue we ran into was that the pipeline was completely stalled. We realised that we did not want to stall the entire pipeline because we needed that Load instruction to hit the writeback stage and commit to the register file. Another issue we found was that when that Load instruction did reach the register file, the pipeline will continue, but the register file has not updated until 1 cycle later. We adjusted the register file to account for this and that fixed our issues with Load instructions and stalling.

For our branch predictor, we wrote basic code with Branch instructions and monitored that when a Branch was reached, the next instruction after the branch, if not taken, was fetched. We then had to make sure that the fetched instruction was also being squashed.



Pipeline Datapath without Data Forwarding Units



Black Box Forwarding Units

3. Checkpoint 3:

Overview:

In the final checkpoint of this project, our task was to implement advanced features to the pipeline that will help the performance while keeping total functionality. Some of the advanced features included modifying the L1 Caches for performance increases, a more advanced branch predictor and different hardware prefetching schemes. Based on the performance of our processor, the biggest factor that is causing slowdown in our pipeline was the memory latency. We also wanted to explore how a more advanced branch predictor affected the performance of our processor.

We decided that we wanted to hide some of the latency when it came to fetching instructions by implementing a basic prefetcher for the Icache. We also wanted to pipeline the provided cache to increase its performance. Since the cache given to us by the course was not the best, we wanted a way to improve it. Unfortunately, neither of us were able to have a fully functional 2-Way Set-Associative Cache from a previous MP, we could not use that. It would have increased the performance of our split cache thus reducing the memory latency and further increasing the speedup. Our last advanced feature was to implement a tournament branch predictor. The idea was that the code provided to us for testing and competition contained several branch instructions. By being able to make progress in the pipeline without having to wait until the Branch instruction has passed the execute stage, the predictor attempts to avoid this overhead thus increasing the performance.

Testing:

Due to the nature of this checkpoint, we had to unit test each advanced feature and ensure that the functionality of the processor does not change when integrating each feature. For the prefetcher, testing was very straightforward. We run any code on the processor and simply check if the Icache holds 2 new cachelines after a miss. For the pipelined L1 Cache, we had to walk through simulations to ensure that the core functionality of the cache has not changed. Similar to testing our pipelined processor, we ensured that signals are passed along the pipeline following the same instruction that was fetched from memory. For the branch predictor, we had to ensure to maintain a high accuracy while ensuring that the predictions made were correct. To do so, we monitored the state diagram for the predictor and ensured that the transitions were correct.

Advanced Features:

1. Basic Hardware Prefetching

Overview:

Our hardware prefetcher was implemented using a one block lookahead (OBL) prefetching scheme. The prefetcher tries to take advantage of spatial locality of instructions since they are almost always executed in order. The general idea of the prefetcher is to fetch the next cacheline after a cache miss. After line ***i*** results in a cache miss, we fetch line ***i+1*** and store it into the cache. Since we're only prefetching for instructions, each cacheline, 256-bits, will contain 8, 32-bit instructions.

At first, we wanted to implement this feature inside the Icache itself. However, we ultimately decided that incorporating the prefetching into the arbiter was more straightforward. The reason being is that the arbiter handles all cache misses so after a miss, it will continue to fetch the next cacheline and return it to the Icache. To do this, we modified the state machine for the arbiter to account for prefetching after an Icache miss. In the prefetching stage, it would either transition back to idle and wait for a cache to send requests, or, it will hand off control to the Dcache if the Dcache requires data from memory.

Performance Analysis:

Metric	Quantity
Icache Misses	6,739
Dcache Misses	9,061
Stalls	19,373
Branch Instructions	6,618
Mispredictions	12,903
Execution Time	5.5 ms

Baseline performance on CP3 Testcode

Metric	Quantity
Icache Misses	6,739
Dcache Misses	9,061
Stalls	19,373
Branch Instructions	6,618
Mispredictions	3,751
Execution Time	7.5 ms

Prefetching Processor on CP3 Testcode

From the above results, we see that the number of Icache misses remained the same; however, the execution time increased by 2ms with almost 30% less mispredictions. In theory, a prefetcher should reduce the number of cache misses since the next cacheline is already fetched and stored in the cache. With our design and implementation, it seems that all the prefetcher is doing is increasing the miss penalty from the Icache but not affecting cache misses. We believe this is due to our implementation of prefetching within the arbiter. Since we prefetch after every Icache miss, we prefetch the next line of instructions; however, we seem to be stalling the

pipeline while doing so. An ideal prefetcher should allow the pipeline to continue while prefetching to allow the memory latency to be hidden.

In hindsight, it makes sense why the pipeline performance is slower than the baseline because we are not allowing already fetched instructions to continue through the pipeline. The pipeline will stall while waiting for either caches to finish servicing requests from the CPU. So, on a cache miss, the entire pipeline waits for that data to return before allowing instructions to make progress. If we were to change this logic and allowed the pipeline to continue while prefetching the next line, we believe that the execution time should be reduced. We are also unsure as to why the number of Icache misses are the same. Our logic to detect Icache misses was that every time Icache sends a read request to physical memory, that is considered a miss. We observed that when line ***i*** misses and is fetched, line ***i+1*** is prefetched and stored in the cache. Using Modelsim, we viewed the data array of the cache and the next 8 instructions are already stored in the cache. If implemented correctly, we expect to see that the number of Icache misses should be reduced by possibly half. Since for every cache miss, we are fetching an extra line of instructions so in theory, the number of misses should be reduced by a factor of 0.5.

We believe that the performance of prefetching would be dependent on the program that is executing. For example, an OBL prefetching scheme that we used would perform very well if the program does not contain many branch instructions. However, an OBL prefetching scheme would not be beneficial in the Dcache since the memory access for data is not as predictable as instructions. Because of this, we chose not to implement prefetching for the Dcache; however, if we were to, we would choose a different scheme like stride prefetching since it works better for random or unpredictable memory addresses.

2. Pipelined L1 Caches

Overview:

By looking at the baseline performance of our processor, we see that the memory latency is the biggest factor when it comes to the performance. We wanted to improve this latency by changing the provided L1 Caches to be more efficient. We decided that pipeline the caches would be most beneficial to our processor design because the penalty for cache misses was very high. Our motivation behind this decision was to improve the Icache to increase the throughput of the cache. This will increase the cache performance thus reducing the memory latency and increasing our processor performance.

To implement the pipeline cache we broke up the cache into 4 stages, *address calculation, disambiguation, cache access, and result drive* while maintaining a 2-cycle hit. The address calculation stage will take the address sent from the CPU and translate the 32-bit address for the cache to use. The disambiguation stage will take the decoded values and pass those values to the appropriate storage arrays, data or tag. The cache access stage will retrieve or store the data and the result drive will output the data and determine if the data was a hit or miss.

Performance Analysis:

Unfortunately, we were unable to implement the functional pipeline L1 Caches. For the most part, data was being cached properly however, when integrating the design to replace the current L1 Caches, the pipeline broke down. We believed that there were timing issues due to the pipelining of the CPU and Caches that we were not able to solve in time. Given more time, we believe that we could have finished the design and have it fully functional with our prefetcher and processor.

We believed that the pipeline cache would have improved the performance of our processor by decreasing the time wasted by waiting for physical memory to respond. By processing new data while waiting for the current data to be returned, we can in theory, speed up the caches therefore increasing the overall performance of our CPU. We speculate that the performance increase would have been much more for the Icache rather than the Dcache since cache misses are more likely to be reduced if the memory accesses are sequential. Since the provided cache were 1-cycle hits, it required a little more work to convert these caches to a 2-cycle hit for the pipelining.

3. Tournament Branch Predictor

Overview:

After running performance counters on the CP3 testcode, we realise that there are a large amount of branch instructions. Since our branch predictor is a static non-taken predictor, it will always fetch and decode the next instruction, ***PC+4***, regardless if the branch is taken or not. What this means is that we waste cycles and resources to fetch a useless instruction if the branch is taken. To eliminate this waste, we wanted to implement a more accurate predictor therefore reducing the amount of wasted cycles by fetching useless instructions.

To do this, we wanted to implement a tournament branch predictor that utilizes both local and global predictors. Using a 2-bit counter, the tournament predictor will choose between the better predictor (local or global). To do this, we had to implement a local branch predictor that kept a history of conditional branches at that ***PC***. The next step was to implement a global branch predictor that records the history of the last *N* branches, in our case we arbitrarily chose *N* to be 32. Both predictor outputs will be inputs to a mux. The 2-bit counter will then choose which predicting scheme would be better and the output will be the mux select.

Local	Global	Choice
0	0	Global
0	1	Global
1	0	Local
1	1	Local

2-bit Counter Values

From the table above, this is how our 2-bit counter system would work. We wanted to implement it as an up/down counter, where if the local branch predictor was correct, we would increment the counter and opposite for the global predictor. For ties, both correct/incorrect, we wanted to use a more relevant predictor. For example, if the Branch instruction at *PC* is correct on both tables, we want to prioritize the local predictor since it would generally be more accurate at this address. The global predictor we believe to be better for predicting the general flow of the program.

Performance Analysis:

Due to time restrictions, we were unfortunately unable to finish and debug the predictor in time. However, we speculate that the tournament branch predictor would have made our processor run much faster for the CP3 testcode. Since there were ~12,000 mispredictions, that is essentially 12,000 wasted instruction fetches. We wanted to cut down on this value to allow the

processor to work on meaningful work that will be used. Since we were using a tournament branch predictor, we gain the benefit of both a local and global predictor. We believe that branch predictors are program dependent and that our static non-taken branch predictor can run very well on code that does not branch often. For example, code that has conditional checks to end execution (i.e. If (n == 1) return), our non-taken branch predictor would perform just as well as the tournament branch predictor. For general use, we believe the tournament predictor will outperform the non-taken. As for the size of the history table, we believe that a larger table will have better accuracy. By having more data and a larger window of branch history, the predictor is able to make more accurate decisions thus improving performance.

Conclusion:

Ultimately, we learned a lot from working on the project. We learned how to utilize different tools to help create and debug our processor design. We got experience in working on a team and learned to adapt to others' coding styles and how important it is to write clean and clear code. We also learned how important it is to verify your design and ensure that it is functional before moving forward. Being able to apply what we learned about pipelining to a physical design was very rewarding. Designing the processor allowed us to quantify how different concepts, such as prefetching, may affect the performance of a processor. Overall, we believe that the project was successful in strengthening our skills and knowledge in computer architecture.