MP4 Final Report

Team PIPT

16 December 2022

Jing Lin (jlin208)
Albert Chen (albertc6)
Amadeus Haryanto (amadeus2)

Table of Contents

1. Introduction

Throughout MP4, we have started from a basic RV32I five-stage pipeline and incrementally expanded upon it. Additional features included forwarding paths, an arbiter between our L1 I-cache and D-cache, hazard detection, strided data prefetch, local, global and tournament branch predictors, and parameterized caches. The goal of this project was to expand our knowledge of computer architecture and to explore the differences between a pipeline design and non-pipeline design. In this report, we will first go over an overview of the project and then discuss the milestones and advanced features implemented. Finally, we will go over an analysis of the runtimes of our design and improvements that could be made.

2. Project Overview

This project aims at implementing a 5-stage pipelined processor with instructions based on the RISC-V ISA. We were required to properly handle data hazard detection, data forwarding, and usage of a cache system. Finally, we were needed to implement advanced features of our choice to improve the performance of our processor.

3. Design Description

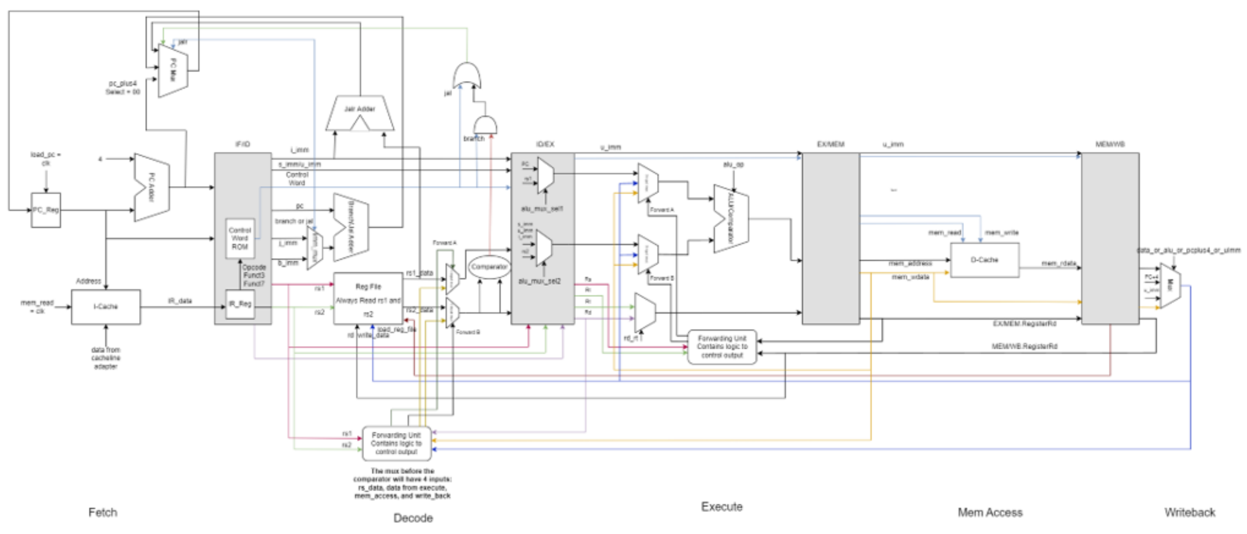
3a. Overview

Starting with checkpoint 1, we implemented a basic RV32I five-stage pipelined design. In checkpoint 2, we expanded on the previous design, adding L1 instruction and data caches, an arbiter, hazard detection, forwarding units, and a static branch predictor. Finally, in checkpoint 3, we implemented our advanced features, which included local, global, and tournament branch predictors, strided data prefetch, and a parameterized four-way set-associative cache.

3bi. Checkpoint 1

Our checkpoint 1 design consisted of a basic RV32I five-stage pipelined design: fetch, decode, execute, memory access, and writeback. In the fetch stage, we fetched the instruction data from memory corresponding to the pc stored in a pc register. In the decode stage, we decoded the instruction data and created a custom module that creates a control word based on the data. The control word would then be propagated down the pipeline and would act as the control signal for each instruction. In the execution stage, we utilized an ALU and a comparator to perform any necessary calculations. In the memory access stage, any load or store instructions would interact with memory. Finally, the writeback stage would write the result of the instruction back to the register file in decode.

During this checkpoint, no data forwarding or branch predictor were implemented and as a result, NOPs had to be inserted whenever there were instruction dependencies or branch instructions in the test code. At this point, we were using the magic memory, which had 1-cycle reads and writes so we did not need to implement a stalling unit. In order to test the functionality of the pipeline, we utilized the provided checkpoint 1 test code and compared the final result in the register with a working MP2 CPU. By the end of this checkpoint, we had a slack of 6.34.



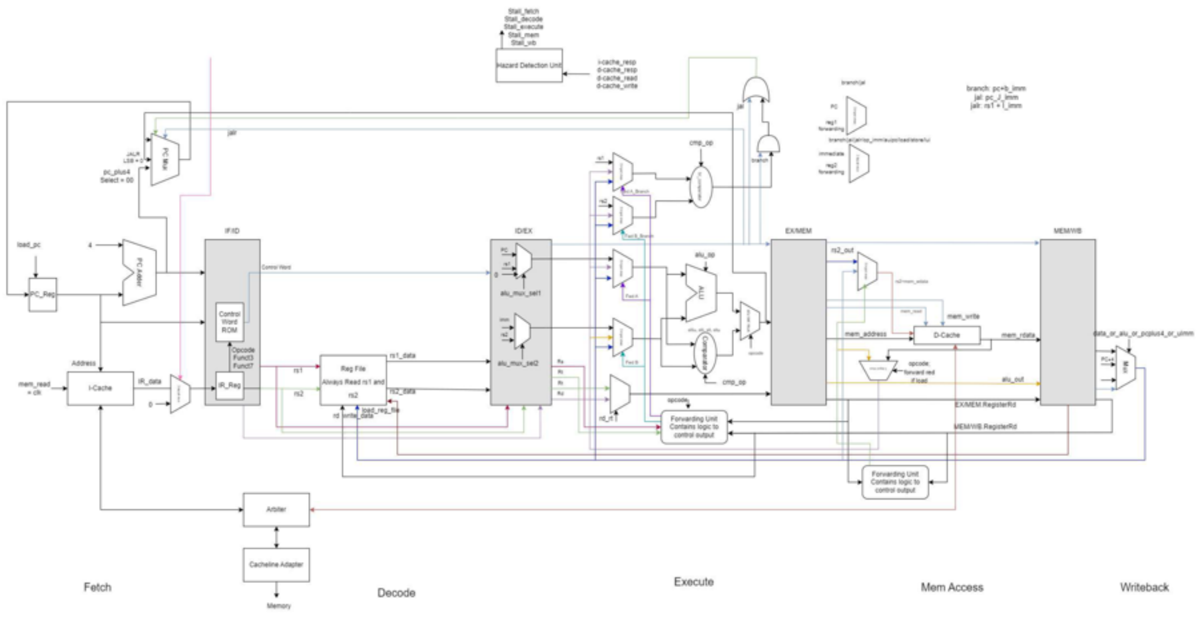
3bii. Checkpoint 2

During checkpoint 2, the design was changed so that it interfaced with memory that had a delay rather than magic memory, which had 1-cycle reads and writes. As a result, we had to add an instruction cache that interacted with the fetch stage and a data cache that interacted with the memory access stage to reduce stalls of repeated instructions and data accesses. We used the provided direct-mapped cache for the I-cache and D-cache. In addition, since there was only 1 input and output port to memory, we had to implement an arbiter, which is used to handle requests from the I-cache and D-cache at the same time. In the event of simultaneous requests from both I-cache and D-cache, the arbiter would prioritize the D-cache's request first and then work on the I-cache's request.

Another feature that we implemented was forwarding and a hazard detection unit that stalls the pipeline whenever necessary. The forwarding paths we had were from WB to EX and MEM to EX. These forwarding paths were used to avoid any unnecessary stalls when there are data dependencies between instructions in the pipeline. Our hazard detection unit detected any cache misses and stalled the corresponding stages until the cache hits. For I-cache misses, it stalled the fetch stage. For D-cache misses, it stalled the fetch, decode, execute, and mem_access stages.

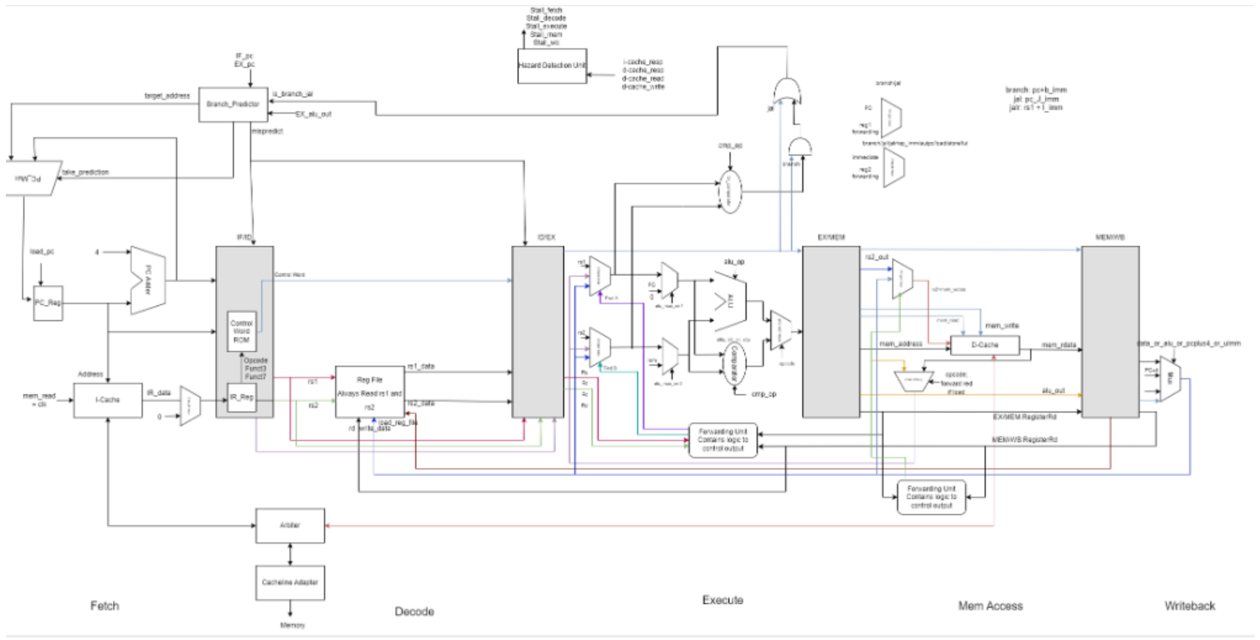
Finally, we implemented a static not-taken branch predictor. Whenever a branch instruction is encountered in the decode stage, it will predict that the branch is not taken and fetch the next instruction. Once the branch reaches the execution stage, if it detects that we mispredicted, meaning the branch was actually taken, it would flush the previous 2 stages by inserting NOPs and fetch the instruction at the taken branch.

In order to test the functionality of the pipeline, we utilized the provided checkpoint 2 test code and compared the final result in the register with a working MP3 CPU and cache. By the end of this checkpoint, we had a slack of 4.62. We also had a combinational area of 27869 and non-combinational area of 24012.



3biii. Checkpoint 3

During checkpoint 3, we began work on our advanced features, which included local, global, and tournament branch predictors, strided data prefetch, and a parameterized four-way set-associative cache. Our main method of testing this checkpoint consisted of running the provided checkpoint 3 test code and finding errors using a combination of the RVFI monitor and running the test code on our MP3 CPU and cache.



3ci. Local, Global, and Tournament Branch Predictor

Originally, our branch predictor was static not-taken, which meant upon decoding a branch, it always guesses that it is not taken and fetches the next instruction. In order to improve the accuracy, we implemented local, global, and tournament branch predictors. Both our local and global predictor uses 2-bit saturating bits, where the four states are strongly taken, weakly taken, weakly not taken, and strongly not taken. Our tournament predictor also uses 2-bit saturating bits representing the states: strong local, weakly local, weakly global, and strongly global.

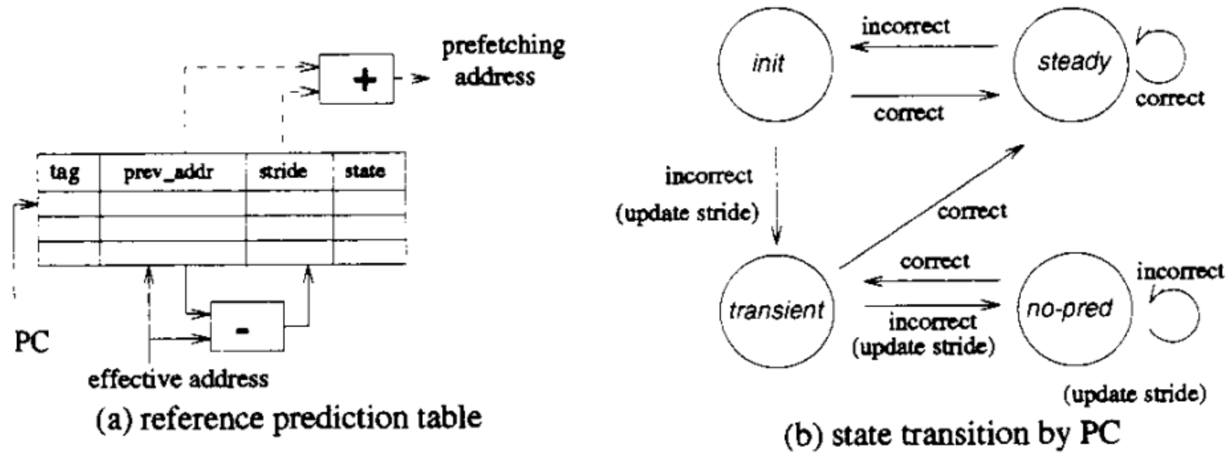
Using only 3 bits for the set-index, which allows for only 3 entries in our history tables, the predictors had very low accuracy. This was because upon replacement, it wiped out the history of the branch and it is not able to reuse the prediction the next time it encounters the branch instruction. By increasing the set bits to 8, it allows for 256 entries, which allows it to store the history for more branches. This greatly increased the accuracy of the predictor, which led to a faster runtime due to fewer flushes on mispredicts.

Set-index bits	Comp1 Runtime	Comp2 Runtime	Comp3 Runtime
3-bits	639485000 ps	1378975000 ps	673155000 ps
8-bits	501945000 ps	1103335000 ps	609315000 ps
Speedup	1.27	1.25	1.10

Set-index bits	Comp1 Accuracy	Comp2 Accuracy	Comp3 Accuracy
3-bits	3302/12370 = 27%	11118/34620 = 32%	452/4391 = 10%
8-bits	10179/12370 = 83%	24900/34620 = 72%	3644/4391 = 83%

3cii. Strided Data Prefetch

To decrease the number of cache misses on data instructions, we decided to implement a data prefetcher which would preemptively fetch data into our D-cache. The prefetcher implementation revolves around the reference prediction table (RPT), in which each entry contains 4 elements—tag, address, stride, and state. The tag is an 8-bit index into the RPT table itself, with the 8 bits being derived from bits [9:2] of the instruction address (input as PC into prefetch module). Thus, our RPT table is 256 entries long. Traditionally, this index is determined by the entire PC, but due to area limitations, we decided to restrict our index to these specific bits in the PC. The address is the current data address. The stride is the difference between the previous data address (obtained from the address field from the previous RPT struct) and the current address. Finally, the state is a 2-bit indexed field consisting of states {init, transient, steady, and no-pred}. State transitions and state actions can be summarized in the tables below. Whenever the prefetcher detects that 1) the current state is steady, 2) memory is free, and 3) we a load instruction, we are able to raise a prefetch signal, signaling to top level that we should take the calculated data address as input to mem_access over the standard mem_address into D-cache. Adding the prefetch module added ~90142.3 to the area, with ~15942.7 being non-combinational logic and ~73178.2 being combinational. Prefetch took ~16.4% of the total design area, mostly due to the size of the RPT tables (256 entries with 100 bits of data each).



(a) reference prediction table
(b) state transition by PC

init	Entry was just initialized with the correct tag. All other fields are populated as 0 by default.
transient	First stride has been calculated, but second stride has not been calculated to determine whether we are in steady state
steady	Current stride = previous stride. We can use this stride to confidently predict the next data address by adding the current data address with the stride value.
no-pred	Current stride ≠ previous stride. In this case, we cannot use our calculated stride, and thus we cannot perform a prefetch prediction.

One major limitation of data prefetching is the lack of sequential instruction prefetching into I-cache. Adding a sequential prefetch module in addition to strided data prefetching would greatly increase the hit rate of both I-cache and D-caches. A large portion of test code has contiguous instruction addresses, and since our current prefetcher does not include instruction prefetching, we have limited our total hit rate considerably.

3ciii. Parameterized Four-Way Set-Associative Cache

For our cache, we changed from the given direct-mapped cache to the two-way set-associative cache we created in MP3. We then increased the number of ways from two to four by changing the pseudo-LRU bit from 1 bit to 3 bits. We also allowed for parameterizing the number of sets in each way. By increasing the number of ways and sets, this reduced the amount of cache evictions, leading to fewer cache misses in the future and as a result, reduced the amount of stalls in the pipeline, which led to faster execution of the code. However, by increasing the number of ways from two to four and the number of sets from 8 to 256, the area spent on the I-cache and D-cache increased significantly, growing to around 169738 total area for both.

Set-index bits	Comp1	Comp2	Comp3
4-bits	501945000 ps	1103335000 ps	609315000 ps
8-bits	501945000 ps	1101785000 ps	608495000 ps
Speedup	1	1.0014	1.0013

Overall, we have found that the speedup provided by doubling the set index bits was not worth the significant increase in area for these competition codes, likely because the testcodes involved a lot of repeated accesses from the for loops. Perhaps if we were to run code that accessed a lot of unique cachelines, we would see greater benefit from increasing the size of our arrays.

4. Conclusion

The primary goal of this final project was to implement a 5-stage pipelined processor adhering to the RISC-V ISA with proper data hazard detection, data forwarding, as well as a basic cache system. We also had the task of designing, and implementing a few advanced features of our choice. With the addition of our advanced features, we were able to improve the efficiency of our design at the cost of increased area. Our processor has an fmax of 235.294 MHz. For future improvements on our processor, we could implement a 2-cycle hit pipelined cache which would allow reading and fetching data in succession without waiting. Another improvement that could have been made was to implement a sequential instruction prefetcher, which would increase the hit rate for I-cache.

	CP1	CP2	CP3
Final Design	7545000 ps	5015000 ps	509115000 ps
Baseline	8085000 ps	5265000 ps	655535000 ps
Speedup	1.0716	1.0498	1.2876