32-Bit RISC-V Pipelined Processor

MP4 Final Report

Spring 2022

Kolten Ackerman, Zixi Li, Andrew Maslankowski

May 13, 2022

Contents

1. Introduction

Over the course of microprocessor history, architectural design has been a vital part of achieving maximum performance by all metrics (performance per watt, execution time, etc.) Today, processor architecture becomes increasingly crucial as Moore's Law appears to reach its end. Thus, in this project, we explore advanced architectural features used in modern processors to increase CPU performance. By implementing, testing, evaluating, and integrating these advanced features, we gain valuable insight into the common tradeoffs of today's processor design.

The ISA chosen for the project was the RISC-V 32I instruction set, a promising open-source RISC instruction set that may one day compete with industry giants such as ARM. Throughout this report, we will introduce our progress in chronological order, from a basic pipelined design to our final design with advanced technologies.

2. Project Overview

The main goal of this project was to improve the performance of a basic, sequential implementation of a RISC-V 32I processor. To accomplish this, weak points and suboptimal functions in the base design were targeted for improvement. For example, idle functional units during basic instruction propagation inspired the 5-stage pipelined design to increase instruction throughput, and memory access latency influenced our implementation of a large pipelined L1 cache to reduce CPU stalling. With goals and projected features agreed upon, our group worked both jointly and independently depending upon the feature being developed. All-encompassing features, such as the pipeline design, were implemented and tested together in group meetings. More isolated components, such as branch prediction and prefetching, were implemented separately, allowing for more flexibility. This separation of work led to a few difficulties when integrating features down the line, but these were primarily minimal complications.

3. Design Description

3.1. Overview

The design of this project centers around a 5-stage pipelined CPU with fetch, decode, execute, memory, and write-back stages, with the final aim of the project to integrate advanced features into this design to optimize performance while minimizing power consumption. The project was divided into four discrete checkpoints, each one designed to steadily advance the progress of our CPU design towards this goal. At each checkpoint, our current progress was summarized and evaluated, and future designs were also planned. Here we present each of those checkpoints in order, with their accomplishments, design documents, the testing methodology, and performance results seen at each stage.

3.2. Milestones

3.2.1. Checkpoint 1

By the first checkpoint, we had implemented the basic pipeline for the RISC-V 32I CPU without fully functional hazard detection or data forwarding. CPU operation at this point was evaluated with magic memory (no latency) and program-inserted NOP instructions to avoid any kind of hazard. Initial data forwarding was laid out with inter-module connections made, but these connections were not leveraged as no hazards were expected. Also implemented was a memory arbitrator, which supported the separation of instruction and data caches; however, this feature was disabled to avoid introducing memory latency.

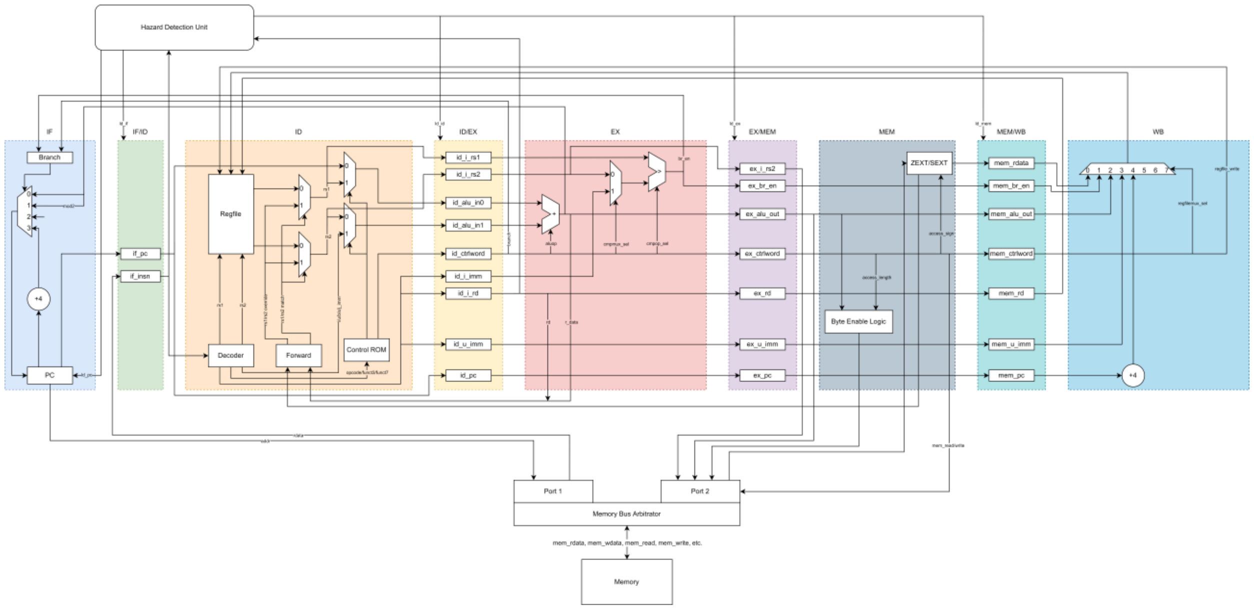


Figure 1: Checkpoint 1 Datapath (See Appendix for Enlarged Version)

Testing for this checkpoint was accomplished by comparing the execution of commands with a verified non-pipelined RISC-V 32I CPU. Among the values checked were memory read/write values, register read/write values, and program counter flow. The implementation was verified to have an identical outcome to the Checkpoint 1 program. The Fmax for this checkpoint was measured at 241.37 MHz, and the power consumption at 335.26 mW. Looking towards checkpoint 2, we planned to finalize the data forwarding and implement hazard detection. We also knew that eventually, we would want to implement an advanced cache, as the currently used given cache was very small and simple.

3.2.2. Checkpoint 2

During checkpoint 2, many functionalities were implemented and verified. Data forwarding was now finished with forwarding possible from EX, MEM, and WB stages to the ID stage. Hazard detection had also been completed to detect, stall, and flush memory latency, data, and branch hazards. Additionally, a simple static-not-taken branch predictor was added to the IF stage as a preparation for a more advanced branch predictor in the future. Finally, with hazard detection in place, we verified the round-robin memory arbitrator.

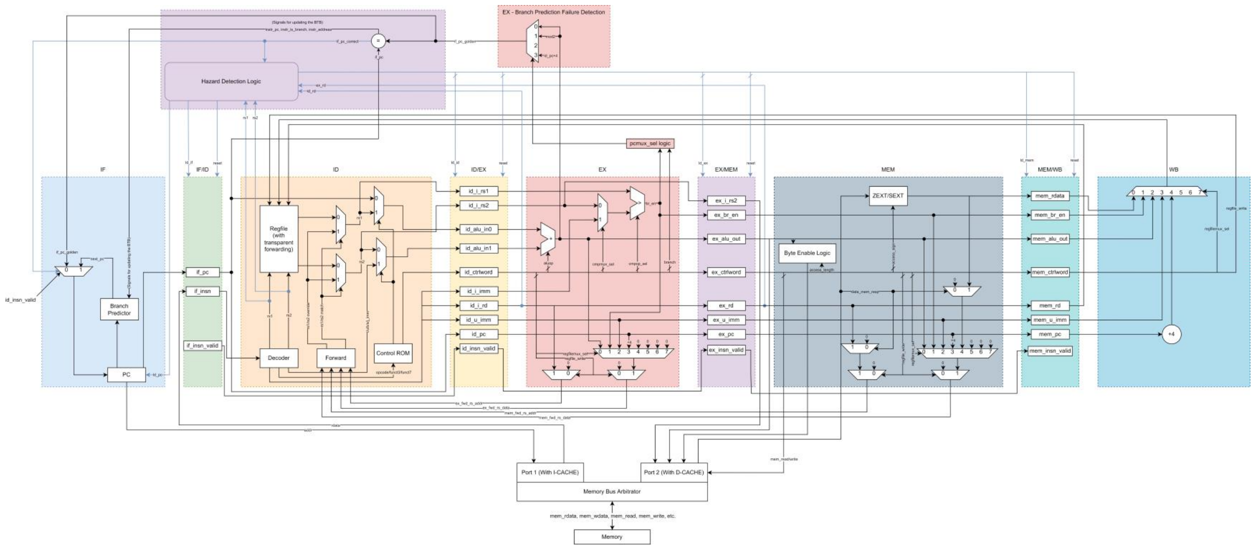


Figure 2: Checkpoint 2 Datapath (See Appendix for Enlarged Version)

Testing for this checkpoint was enhanced by connecting our CPU to the RISC-V formal interface (RVFI). Combined with comparing final register values to the verified non-pipelined RISC-V 32I CPU, we were able to verify functionality on the checkpoint 2 test program, as well as on competition codes 1, 2, and 3. In response to the new features, Fmax had dropped to 113.39 MHz, and total power consumption rose to 421.63 mW. Looking ahead towards checkpoint 3, we planned the advanced features we thought would best improve performance. These advanced features were a pipelined L1 cache, RV32M (multiplier) extension, branch target buffer with return address stack, and one block lookahead prefetcher.

3.2.3. Checkpoint 3

Checkpoint 3 saw the greatest variety of advanced features added to increase performance. By this checkpoint, a pipelined L1 cache, branch target buffer, return address stack, and one block lookahead prefetcher was added (see Advanced Design Options). Not all functionalities were integrated to work together in time, but each had been separately verified and had its performance evaluated.

Testing was again achieved using both the RVFI monitor and comparing final register values against the verified CPU. Performance counters were also added as part of the testing strategy to better evaluate the function and correctness of the advanced features (i.e., cache hit and miss counter for advanced L1 cache). The addition of these advanced features caused Fmax to drop once more to 97.82 MHz and total power consumption to rise to 659.31 mW

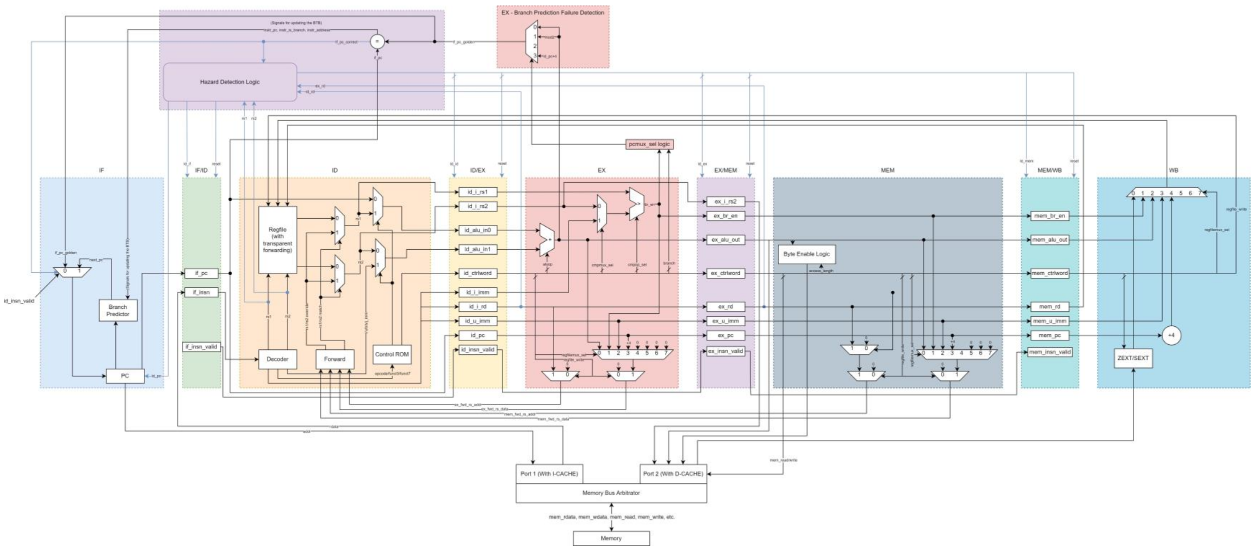


Figure 3: Checkpoint 3 Datapath

3.2.4. Checkpoint 4

As the final checkpoint, Checkpoint 4 consisted more of project optimization and tweaking than the implementation of new features. The most significant improvement made was the expansion of the instruction cache. Up until this point, we were using the given instruction cache, which, while functional, had very suboptimal performance due to its small size. In addition to the expanded instruction cache, separate features from Checkpoint 3 were integrated into one design and had their parameters adjusted to find a desirable tradeoff between power and performance, increasing Fmax to 99.65 MHz and power consumption to 839.51 mW. Ultimately, many minor adjustments lead to our final design with the greatest performance.

3.3. Advanced Design Options

3.3.1. Pipelined L1 Cache

3.3.1.1. Design

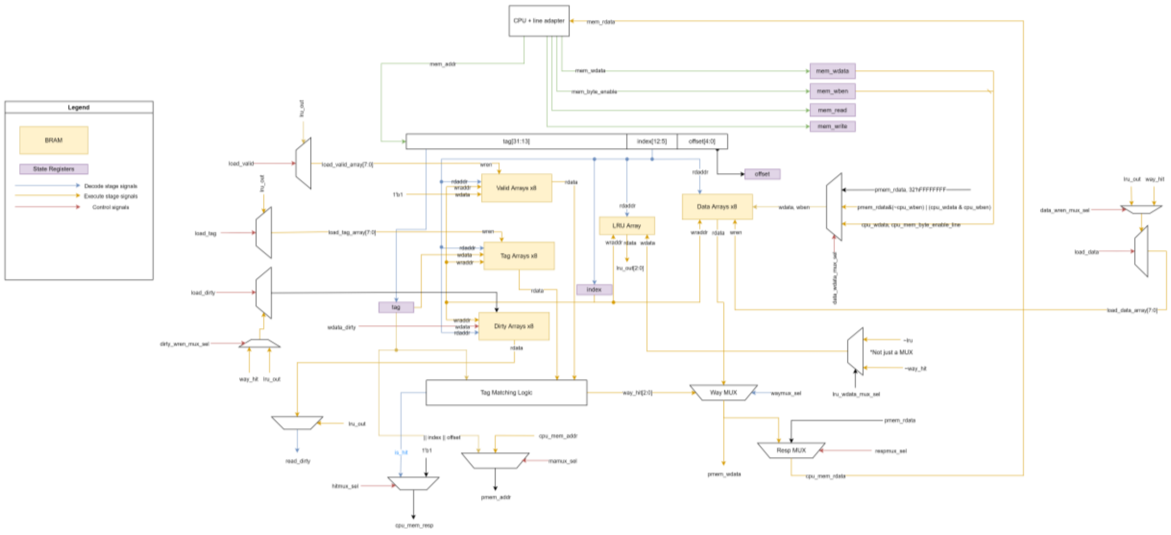


Figure 4: Pipelined L1 Cache Diagram

Our design consists of a pipelined data cache with parameterized number of data word width, address width, offset bits, index bits (hence the number of entries per way), and the number of ways. The CPU core is also modified so that all data memory requests are issued at the MEM stage, and the response will be sent directly to the WB stage on the next clock cycle. This pipelining technique helps reduce the critical path length and increases the maximum frequency.

3.3.1.2. Testing

Due to the special timing and access sequence of a pipelined cache, testing is performed after modifying the CPU core, reconnecting the memory signals of the RVFI monitor, and modifying signals for shadow memory. After these changes were made, we ran all competition codes and all checkpoint codes, monitoring the simulation outputs, as well as comparing final register file values with golden results.

We also wrote a piece of code that deliberately evicted the cache and used a Python script to generate assembly codes that access the memory randomly; Shadow memory is used to monitor the coherency of results.

All the above tests were done for different numbers of ways and different widths of index bits by modifying the parameters. However, we did not change the length of offset bits, word length, or address length since the 32-bit CPU and memory fixed those values.

3.3.1.3. Performance Analysis

To find the best balance between cache size, frequency, and power, we tried several groups of different parameters and simulated all competition codes, recording the cache hit/miss rates, frequency, and consumed power. The results are recorded in the following table:

Ways	Entries/ Way	Comp1.s Hits	Comp1.s Misses	Comp2.s Hits	Comp2.s Misses	Comp3.s Hits	Comp3.s Misses	Fmax (MHz)	Power (mW)
4	128	2806	7	4375	24	10851	283	96.04	76.26
8	128	2806	7	4375	24	10851	283	90.29	132.1
4	256	2806	7	4375	24	10851	283	97.43	73.04
8	256	2806	7	4375	24	10851	283	88.49	124.91
4	512	2806	7	4375	24	10851	283	99.65	103.66

Table 1*: Cache Test Results*

As seen in Table 1, the cache hit and miss rates are not affected through all trials. By checking the simulation result, we found that even the smallest cache size is sufficient for the work set of the competition codes, and there were no cache evictions. We did not try smaller cache sizes because the chip already has a sufficient amount of block RAM, and there are other parts of the design limiting Fmax.

Originally we thought there would be a tradeoff between cache size, power consumption, and Fmax. However, according to the data we measured, it seems that we do not need to trade off Fmax and power for larger cache size. This cache would bring the most benefit to memory-intensive workloads. This cache would not handle cases when the program being run has bad data locality, leading to more cache misses, but the same can be said for almost all cache designs.

3.3.2. Branch Target Buffer and Return Address Stack

3.3.2.1. Design

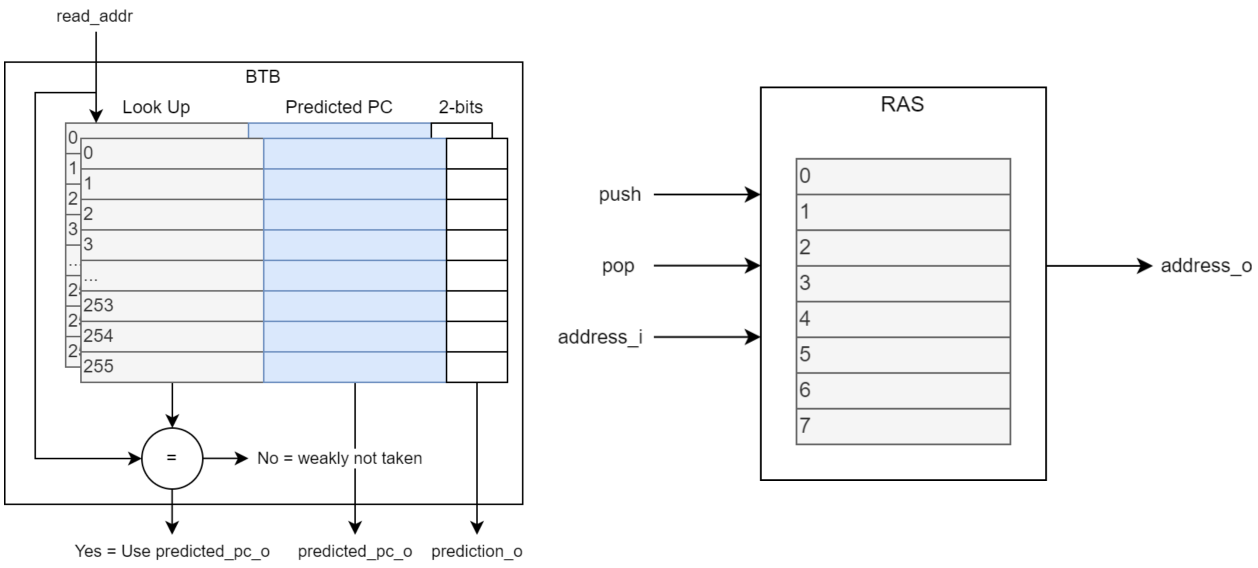


Figure 5: Branch Target Buffer and Return Address Stack Diagrams

To improve branch prediction accuracy, we designed a branch target buffer and a return address stack to get more accurate predictions than the default static-branch prediction of the baseline processor. The branch target buffer we designed (BTB) is a parameterized buffer consisting of a two-bit history table. The table stores PC values and predicted PC values, along with prediction bits, in order to make better branch guesses. The BTB predicts weakly not taken upon cold misses and otherwise uses the prediction bits to determine the predicted PC value and anticipate branches. The original BTB was direct-mapped and much larger. However, due to power consumption issues and underutilization of many of the direct-mapped indices in the BTB, we switched to a smaller 2-way set associative BTB that consumed much less power with a slight performance decrease.

The return address stack we implemented is a simple but effective tool to get better performance out of our branch predictor. It is a small circular buffer that attempts to mirror a software stack by storing the return address of function calls upon detection of call instructions. The stack then pops off an address when a return instruction is detected and sends out the address as the predicted PC value, always overriding the predicted BTB value. We choose to do this because BTBs tend to perform poorly with function calls and returns, so the RAS is privileged. If branch misprediction does not cause erroneous pushes/pops, the RAS has a perfect prediction rate.

3.3.2.2. Testing

The initial testing of the BTB and RAS consisted of verifying accuracy with the RVFI monitor and shadow memory. One of the initial problems we encountered was inadequate hazard detection to deal with mispredictions within the cache. This was a hard bug to solve, but we ended up adding a flush signal to the cache that would clear out any erroneous cache lines. After this problem was solved and the RVFI monitor was not throwing any errors, performance counters were added to see if the BTB and RAS were behaving as expected. Initially, the RAS stack was not working due to incorrect timing (the popped address was being sent to the CPU a cycle too late), but the performance counters and waveforms revealed this error. A major BTB issue uncovered in testing was the erroneous update of prediction bits, causing very poor branch prediction accuracy. After this was resolved, the BTB and RAS were behaving far better. With a working BTB and RAS stack and accurate performance counters, the branch predictor was tested well with the competition codes as there are lots of branches and function calls in the competition codes.

3.3.2.3. Performance Analysis

The branch prediction results were fairly good, and certainly a major improvement over the static prediction. The speedup was noticeable but not significant for all competition codes. There was a roughly 1.18x speedup for competition code one, a major improvement. Competition codes two and three had a roughly 1.04x speedup, which is decent but not as good as competition code one. These speedup numbers are based on runtimes we got for our CP3 submission of the branch predictor, which initially had a minor bug where BTB prediction addresses would be updated to (PC + 4) upon writing to the BTB that included a not taken branch. This was obviously a mistake since only taken branch addresses should be stored in the BTB. Interestingly, this seemed to benefit competition codes one and three, perhaps because they had more ‘not taken’ branches. This bug was fixed for the competition, and the differing results can be seen in Table 2 and Table 4. Note that branch prediction and BTB performance decreases for the competition CPU, but this is due to power optimization through the decrease of the BTB size. The power tradeoff for minimal performance loss was well worth it.

	Comp1			**Comp2**			**Comp3**		
	Branch perf	**RAS perf**	**BTB perf**	**Branch perf**	**RAS perf**	**BTB perf**	**Branch perf**	**RAS perf**	**BTB perf**
Branch Predictor	0.9033	0.99261	0.9962	0.73921	0.60837	0.9897	0.87679	1	0.98265

Table 2: CP3 BTB and RAS Performance (Minor Bug, Larger BTB)

	Comp1 runtime (ns)	**Comp2 runtime (ns)**	**Comp3 runtime (ns)**	**Fmax (MHz)**	**Power (mW)**
Static Branch Predictor	2,116,295	5,929,875	3,840,675	97.82	0.05
Adv. Branch Predictor	1,780,185	5,755,125	3,684,335	93.52	261.58

Table 3: CP3 BTB and RAS Runtime Results

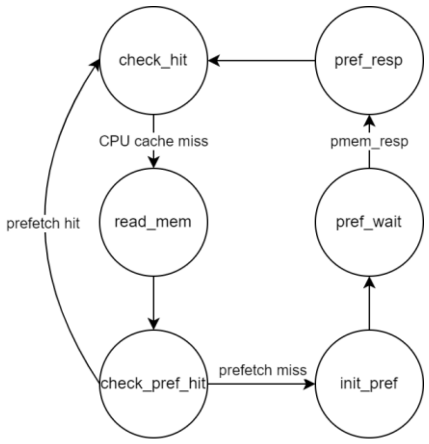
	Comp1			Comp2			Comp3		
	branch perf	RAS perf	BTB perf	branch perf	RAS perf	BTB perf	branch perf	RAS perf	BTB perf
Branch Predictor	0.850754	0.99322	0.937577	0.788201	0.748132	0.964269	0.817913	1	0.87828

Table 4: Competition BTB and RAS Results (Solved Bug, Smaller 2-Way Associative BTB)

The branch predictor we currently have does well with programs that execute jumps, for loops, and function calls, as well as standard branches that have a tendency to favor taken or not taken. The BTB has a very high hit rate, so programs including repetitive branching will perform well with our branch predictor. Programs that have correlated branches or branches that can frequently go one or the other would perform the worst with our predictor. We have no way to track correlated branches, and more 'random' branches will not take advantage of the two-bit predictor, which will lead to a decent amount of mispredictions. Misprediction that leads to a lot of function calls will also cause poor predictor performance because the return address stack will be corrupted, decreasing RAS performance.

3.3.3. One Block Lookahead Prefetcher

3.3.3.1. Design



State	Description
check_pref_hit	Access cache arrays using prefetch address, check if already in cache
init_pref	Initialize read of main memory using the prefetched address
pref_wait	Waiting for main memory, handle CPU cache hit requests
pref_resp	Write contents into cache arrays using prefetch address, continue normally

Figure 6: Instruction Cache Prefetcher FSM

The one-block lookahead prefetcher initiates a prefetch of cache line (i + 1) whenever there is a cache miss on cache line i. The prefetcher exploits (sequential) spatial memory locality as it initiates the main memory access before the CPU demands it, thus hiding some of the latency. A workload that benefits the most from this locality exploitation is the instruction cache of a program that has minimal branching. These programs have the program counter increment sequentially and therefore prefetch useful data often. A program with a lot of branches, however, might lead to unnecessary prefetches and cache evictions, thus actually leading to decreased performance versus no prefetching at all.

To accomplish this functionality, the instruction cache has a fetch signal to alert the prefetcher that a cache miss took place. When the prefetcher is signaled that a fetch has occurred, it registers the address fetched and raises its read signal to the cache. The instruction cache then responds to the prefetcher as soon as there are no pending CPU instruction cache requests. The instruction cache FSM was also extended to support a slightly different fetch functionality when prefetching – notably, the ability to respond to CPU cache hits in the pref_wait state. Once a prefetch has occurred, everything returns to a normal state waiting for the next cache miss.

3.3.3.2. Testing

Testing of the prefetcher was primarily done via waveform inspection. If the FSM entered, executed, and exited a prefetch correctly, then there weren’t many edge cases to consider. Performance counters, including instruction cache hits and misses, prefetch hits and misses, and average memory latency (defined as # of clock cycles with CPU read asserted / instructions read), were added to evaluate performance.

3.3.3.3. Performance Analysis

	Comp1		Comp2		Comp3		Power & Frequency	
	CPU hit %	avg_latency (clk cycles)	CPU hit %	avg_latency (clk cycles)	CPU hit %	avg_latency (clk cycles)	Fmax (MHz)	Power (mW)
Baseline	0.932513	3.75147	0.924034	4.37257	0.896616	4.95641	97.41	31.36
Prefetcher	0.963876	3.43213	0.949572	4.65611	0.946368	4.70899	101.17	32.03

Table 5: OBL Prefetcher Performance Results

The performance results were not as impressive as we had hoped. Although the average latency did decrease in most cases, the decrease in runtime was not significant. This inconsistent small runtime decrease did not make up for cases where (due to unnecessary prefetches and evictions) runtime increased.

3.3.4. Expanded Instruction Cache

3.3.4.1. Design

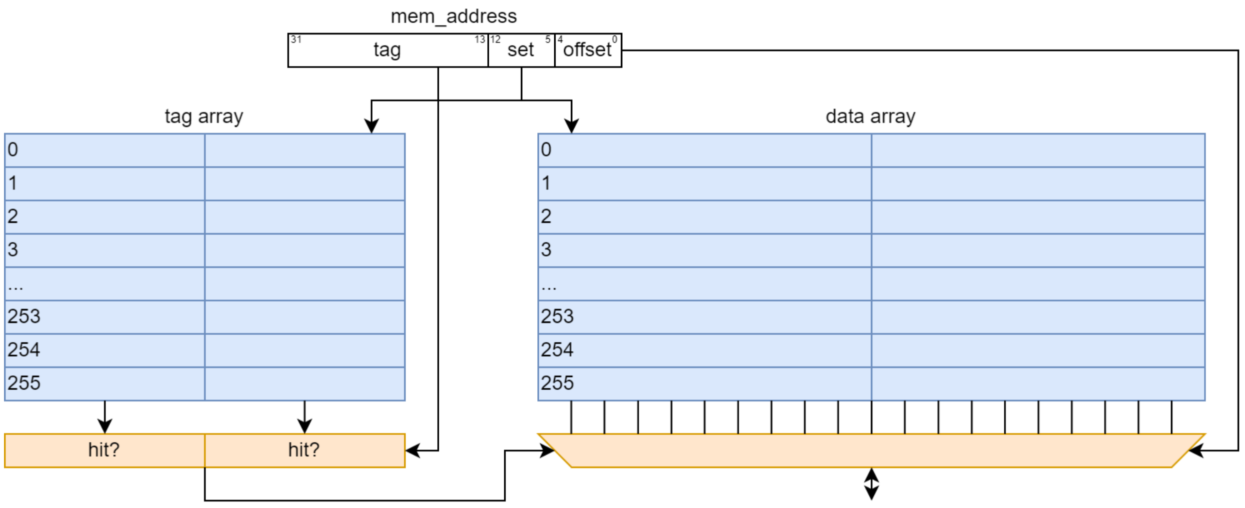


Figure 7: Expanded Instruction Cache Diagram

The main goal behind the expanded instruction cache was to increase capacity while preserving the one-clock response. The given cache that had been used through checkpoint 3 had only 8 entries (1 way), so improving upon this baseline was not a difficult task to undertake. Different pairings of entries and ways were tried, but we ultimately wound up on 256-entries and 2-ways as the best utilization of power, performance, logic utilization, and one-clock response time.

3.3.4.2. Testing

Verification of the expanded instruction cache was achieved using the RVFI monitor, performance counters, and waveform inspection. The parameterization of the cache arrays made entry expansion a relatively simple task. Adding a second way to the cache was a more complicated upgrade and therefore made up most of the verification effort.

3.3.4.3. Performance Analysis

	Comp1 runtime (ns)	Comp2 runtime (ns)	Comp3 runtime (ns)
w/o Expanded Instruction Cache	1,780,185	5,755,125	3,684,335
Expanded Instruction Cache	496,995	1,057,615	701,695

Table 6: Expanded I-Cache Runtime Results

	Comp1		Comp2		Comp3		Power & Frequency	
	CPU hit %	avg_latency	CPU hit %	avg_latency	CPU hit %	avg_latency	Fmax (MHz)	Power (mW)
w/o Expanded I-cache	0.93123	3.79009	0.890977	5.3834	0.889689	5.16345	93.52	30.94
Expanded I-cache	0.999384	1.02102	0.999577	1.01673	0.99955	1.01835	83.79	253.97

Table 7: Expanded I-Cache Performance Results

The performance improvement gained by expanding the instruction cache was our largest improvement yet. Runtimes became 4-5x faster at the cost of around 200 mW. CPU hit percentage and average latency were also both improved to near their theoretical maximums, signifying there was not much improvement left to be made (regarding the instruction cache). If nothing else, this expansion demonstrated the importance of a well-designed memory hierarchy on performance and the close-knit relationship between CPU and memory.

4. Conclusion

Overall, we accomplished our main goal of achieving substantial performance increases over a basic sequential processor using a pipelined design and some advanced features. Our pipelined RISC-V CPU was fully functional, and advanced features successfully improved the performance beyond that of the pipelined CPU on its own. The baseline pipelined CPU metrics and our advanced CPU metrics are listed in the following table for each of the competition codes:

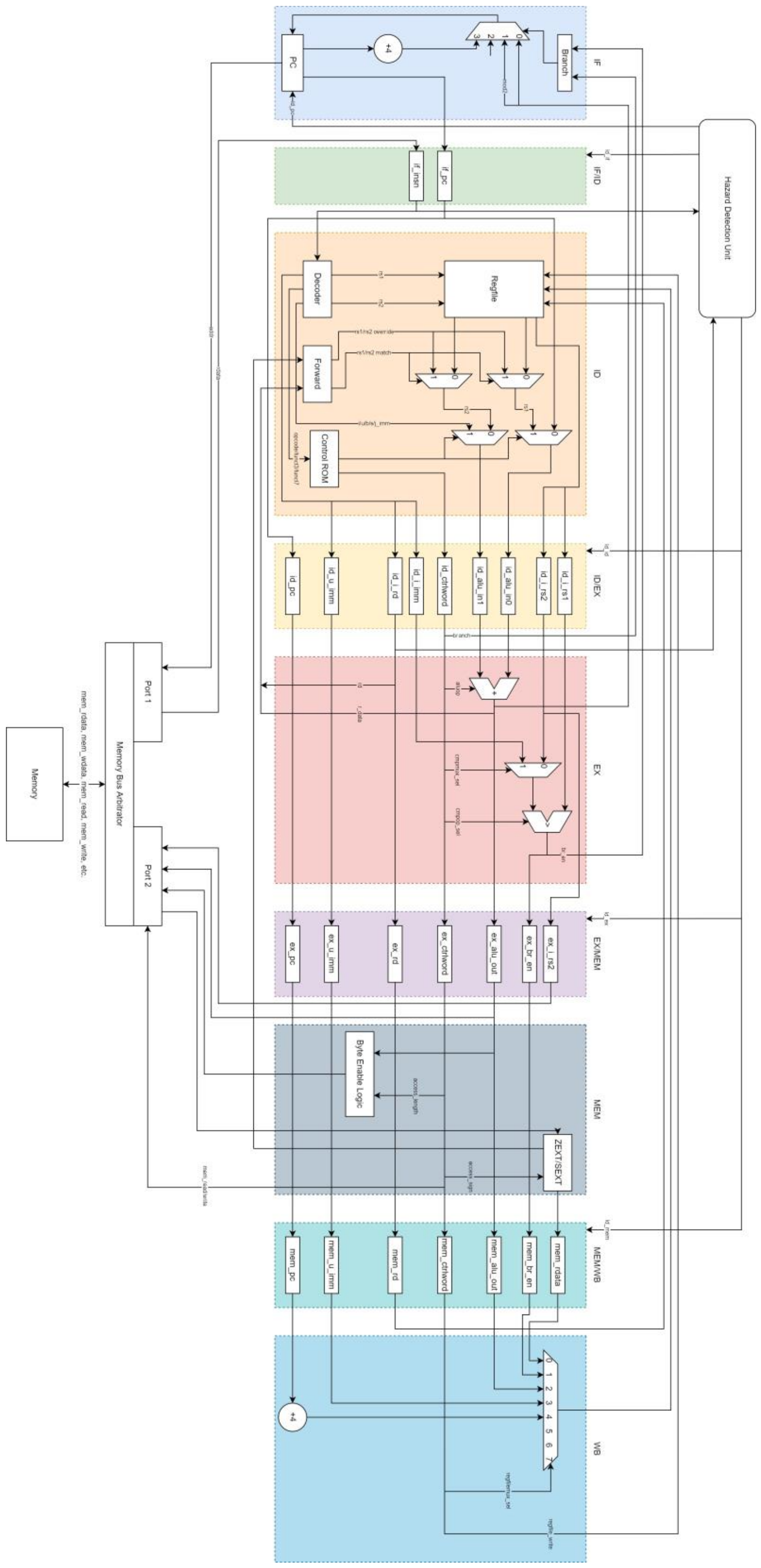
	Comp1 Time (ns)	Comp1 Power (mW)	Comp1 Score	Comp2 Time (ns)	Comp2 Power (mW)	Comp2 Score	Comp3 Time (ns)	Comp3 Power (mW)	Comp3 Score
Baseline	720,755	448.44	1.68E-10	4,521,285	430.48	3.98E-08	3,639,265	425.27	2.05E-08
Best Design	505,035	654.1	8.43E-11	1,094,505	679.09	8.90E-10	707,805	569.38	2.02E-10

Table 8: Overall CPU Results

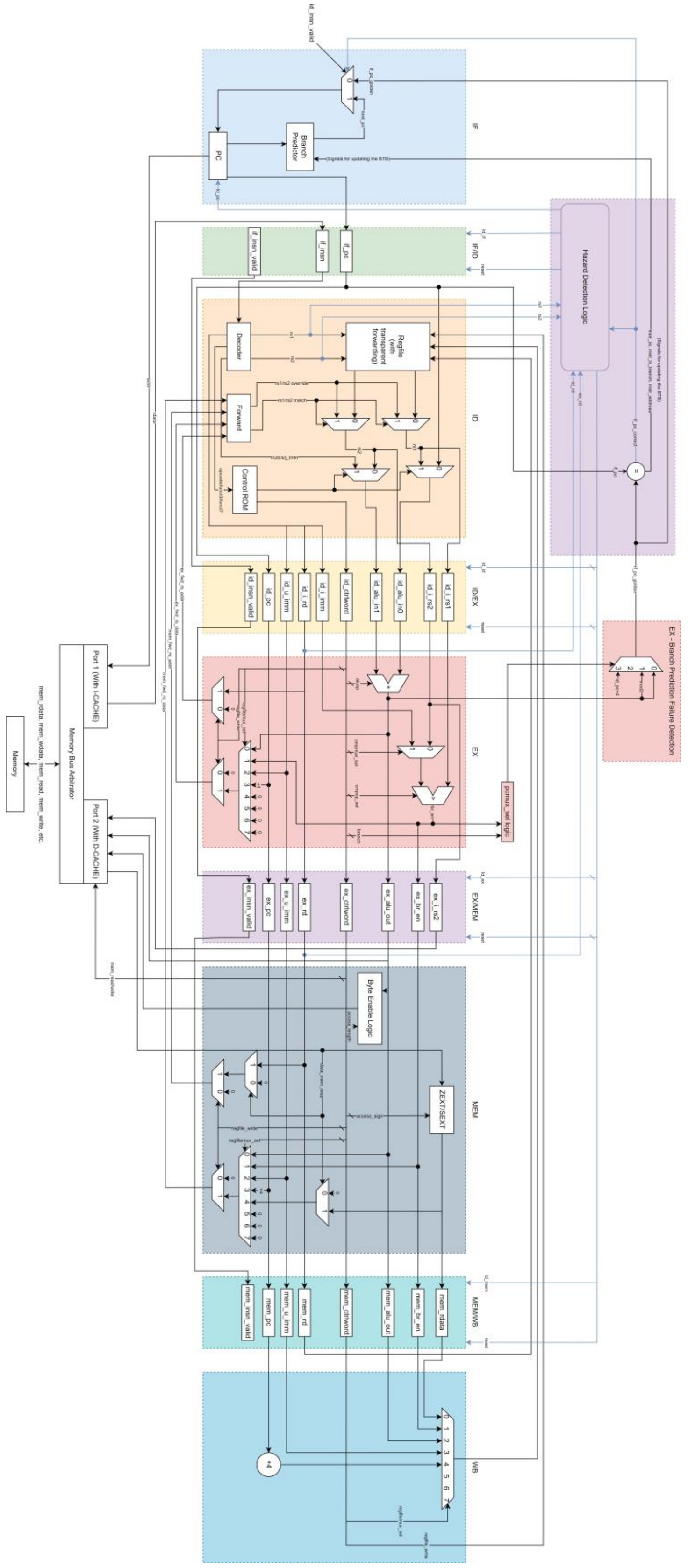
In this project, the score was proportional to power multiplied by the running time to the third power, so our decision to minimize runtime at the cost of some increase in power was worth it. Lowering runtime across all competition codes compared to the baseline demonstrates our success. Beyond the numbers, this project taught us how to manage project complexity, work as a team, and meet deadlines together. We also learned how to deal with the adversity that comes with such a large project. We were able to solve many of the problems we faced, but there were setbacks, such as being unable to integrate branch prediction and prefetching together. Deadlines would constrain us well. Finishing everything in a timely manner was a challenge. However, despite wishing we could have done some things differently and had more time to improve our design, we can look back and be satisfied with what we accomplished. We implemented complex advanced features that can be found in our best modern processors, which gave us first-hand knowledge of the challenges computer engineers currently face in improving processor performance. Overcoming memory latency was the most difficult challenge we discovered, and when we were able to eliminate this latency, we saw the biggest performance increases. If there was a single takeaway from this project, it would be to never underestimate the value of large and efficient caches in processor design– it does not matter how fast the CPU alone is if it’s bottlenecked by poor memory latency.

5. Appendix

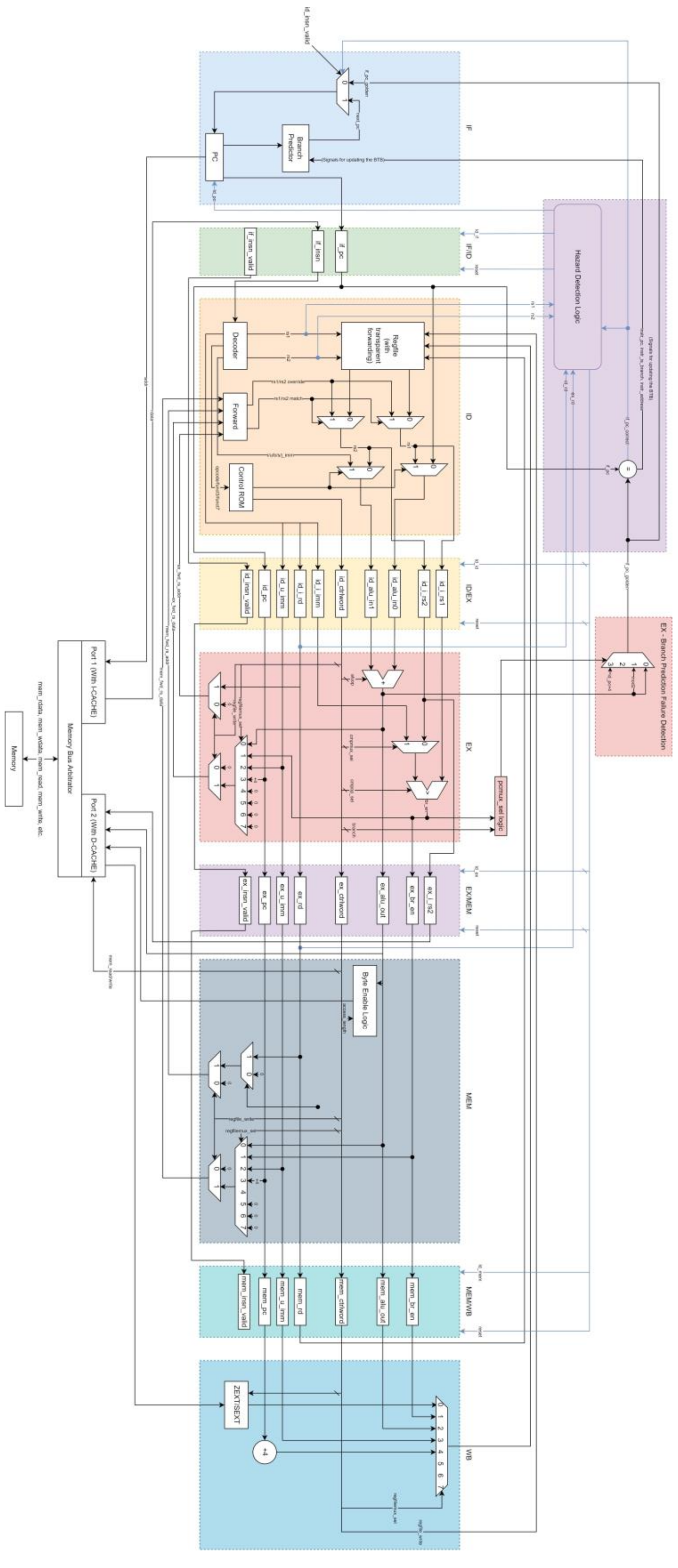
5.1. Checkpoint 1 Datapath Diagram



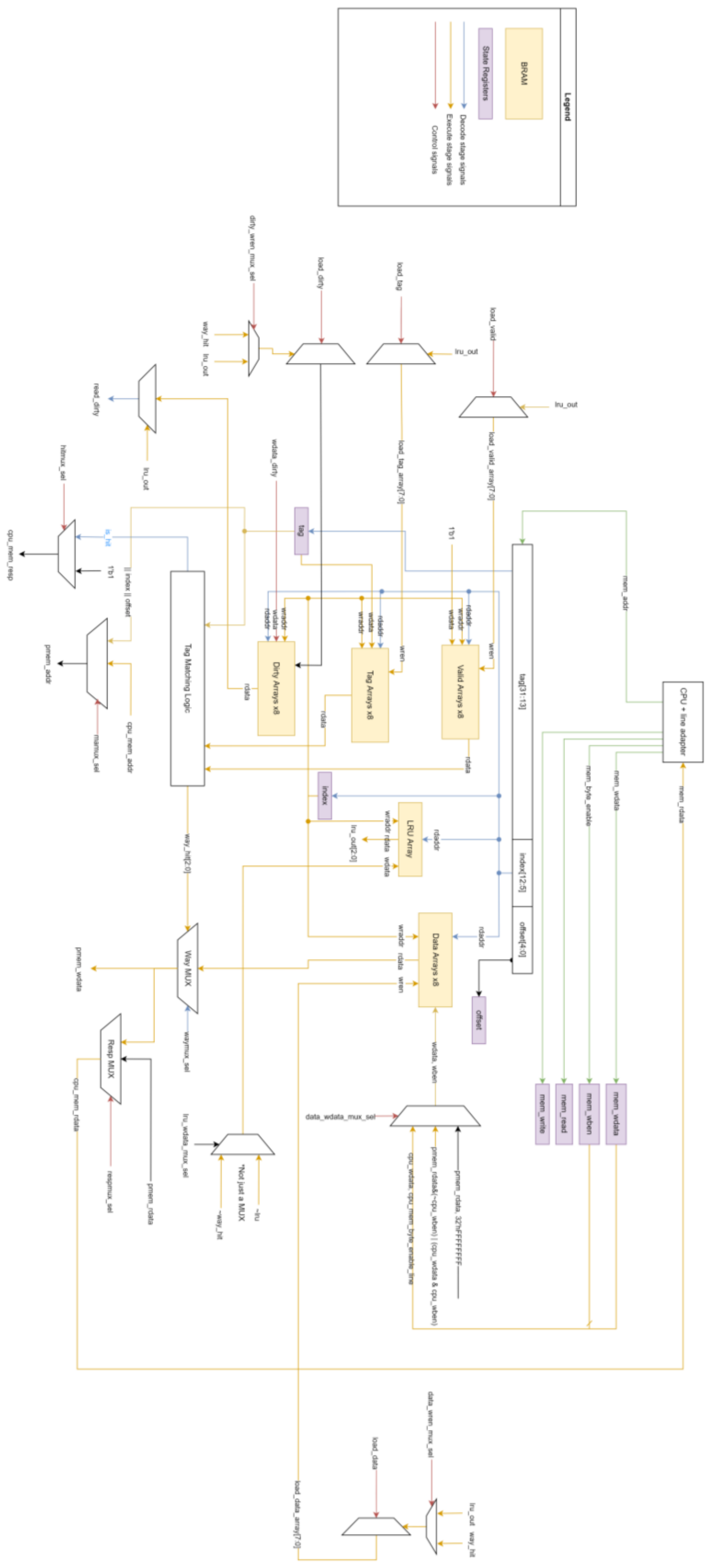
5.2. Checkpoint 2 Datapath Diagram



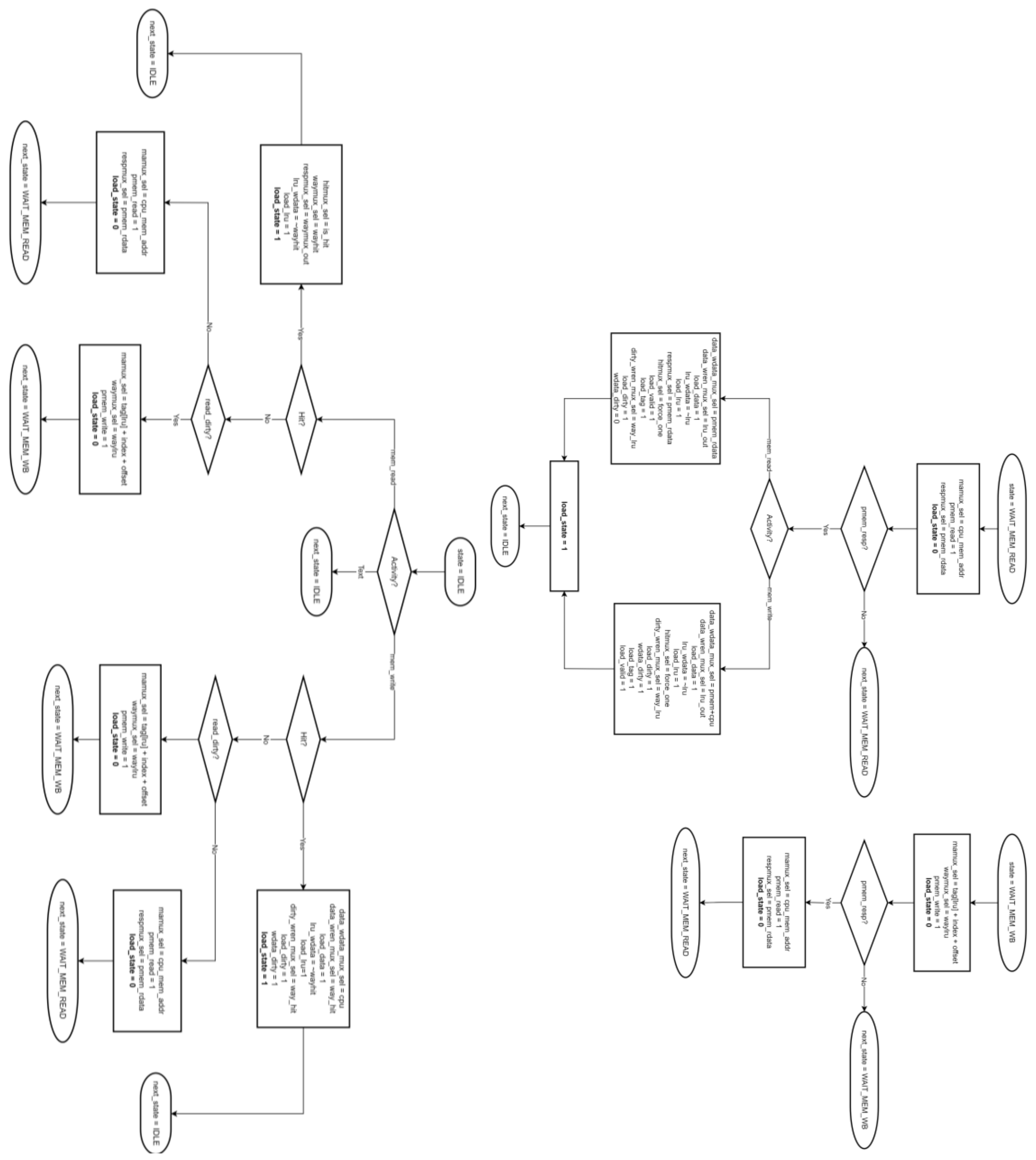
5.3. Checkpoint 3 Datapath Diagram



5.4. L1 Cache Datapath



5.5. L1 Cache Control Logic



state = IDLE
Activity?
Text
next_state = IDLE
mem_read
Hit?
Yes
hitmux_sel = is_hit
waymux_sel = wayhit
respmux_sel = waymux_out
lru_wdata = ~wayhit
load_lru = 1
load_state = 1
next_state = IDLE
No
read_dirty?
No
marmux_sel = cpu_mem_addr
pmem_read = 1
respmux_sel = pmem_rdata
load_state = 0
next_state = WAIT_MEM_READ
Yes
marmux_sel = tag[lru] + index + offset
waymux_sel = waylru
pmem_write = 1
load_state = 0
next_state = WAIT_MEM_WB
mem_write
Hit?
Yes
data_wdata_mux_sel = cpu
data_wren_mux_sel = way_hit
load_data = 1
lru_wdata = ~wayhit
load_lru=1
dirty_wren_mux_sel = way_hit
load_dirty = 1
wdata_dirty = 1
load_state = 1
next_state = IDLE
No
read_dirty?
No
marmux_sel = cpu_mem_addr
pmem_read = 1
respmux_sel = pmem_rdata
load_state = 0
next_state = WAIT_MEM_READ
marmux_sel = tag[lru] + index + offset
waymux_sel = waylru
pmem_write = 1
load_state = 0
next_state = WAIT_MEM_WB
state = WAIT_MEM_READ
marmux_sel = cpu_mem_addr
pmem_read = 1
respmux_sel = pmem_rdata
load_state = 0
pmem_resp?
No
next_state = WAIT_MEM_READ
Yes
Activity?
mem_read
data_wdata_mux_sel = pmem_rdata
data_wren_mux_sel = lru_out
load_data = 1
lru_wdata = ~lru
load_lru = 1
respmux_sel = pmem_rdata
hitmux_sel = force_one
load_valid = 1
load_tag = 1
dirty_wren_mux_sel = way_lru
load_dirty = 1
wdata_dirty = 0
mem_write
data_wdata_mux_sel = pmem+cpu
data_wren_mux_sel = lru_out
load_data = 1
lru_wdata = ~lru
load_lru = 1
hitmux_sel = force_one
dirty_wren_mux_sel = way_lru
load_dirty = 1
wdata_dirty = 1
load_tag = 1
load_valid = 1
load_state = 1
next_state = IDLE
state = WAIT_MEM_WB
marmux_sel = tag[lru] + index + offset
waymux_sel = waylru
pmem_write = 1
load_state = 0
pmem_resp?
No
next_state = WAIT_MEM_WB
Yes
marmux_sel = cpu_mem_addr
pmem_read = 1
respmux_sel = pmem_rdata
load_state = 0
next_state = WAIT_MEM_READ