ECE 411 - UIUC

BigBrainCPU MP4: RISC-V Processor

Designed By: Yash Tyagi, Aayush Patel, Vishal Narasimhan

I. Introduction

In this project, we have designed a 5-Stage Pipelined RISC-V Processor. These stages include an Instruction Fetch, Decode, Execute, Memory Access, and Writeback referred to as IF, ID, EX, MEM, and WB, respectively. This processor includes forwarding paths as well as advanced features, both will be discussed in detail later on in this report. This processor serves an important purpose in understanding the fundamentals of a computer organization and design. Many topics covered in this report are used in the world today, such as pipelining, forwarding, branch prediction, cache organization, etc. This project was separated into checkpoints, with the first being a basic architectural setup, the second developing this setup with forwarding paths, the third incorporating the advanced features (explained in detail below), and the last being a design analysis to prioritize performance. Many various considerations went into making this processor, where we have to weight the advantages and disadvantages between speed and efficiency. This project teaches how to approach a holistic design for the processor, which is a crucial skill to have when considering much more complex designs. The next section will cover the goals of our project and the organizational aspects required for completing this processor in a structural, progressive manner.

II. Project Overview

Like mentioned before, the goal of this project was to design a RISC-V 5-stage processor. In addition to basic pipelining, the goals were to include forwarding paths, and multiple advanced features, all while considering performance and speed of the processor. The advanced features we chose to implement were a Tournament Branch Predictor, a Branch Target Buffer, a Return Address Stack, an Eviction Write Buffer, and a Prefetching Unit. In the beginning checkpoint of this project, we had a vague idea of what design features we wanted to include, which played a part in the consideration of the rest of the design. One of the guidelines of this project was that a control ROM must be used for pipelining and not a state machine. However, as seen in our checkpoint one diagram, there were many similar elements to our MP2. Because of this, we were able to utilize much of the previous modules we created.

The choices of our advanced features were influenced based on feasibility as well as the improvement on previous designs. Our first choice was to create a Tournament Branch Predictor, as these instructions would cause the most delay in our performance. Because of this choice, adding a Branch Target Buffer and a Return Address Stack would improve the performance. The Eviction Write Buffer was to ensure that writing to memory would be done when necessary, and the prefetcher to reduce the time spent receiving instructions. Additionally, an L2 Cache was used to decrease the amount of time of memory retrievals, to increase performance speed. More information regarding these features will be discussed in the Advanced Design.

In an organizational and administrative sense, we worked on Microsoft Visual Studio Code Live Share throughout this project in order to create an environment where all of us could code/debug simultaneously. We all worked on various elements throughout the checkpoints, which is described in more detail in the Milestones section. One of the complications we expected was with all being in different places due to COVID, but fortunately a good network of communication and frequent meetings online allowed us to complete all the checkpoint and design features successfully, with decent performance.

III. Design Description

a. Overview

In this section, we will discuss the various steps that were taken at each checkpoint and what the update was for our design at each of these milestones. The designs can be found in Appendix A and the corresponding figure will be noted in the rest of this report. As seen in our design, in between each stage is a series of registers that hold values needed for future stages in the pipeline. For the advanced design options, high level description diagrams are presented, as well as a list of inputs and outputs for each of the connections. In the section for each checkpoint, we will explain the progress on the design, why certain design decisions were made, and what the modules consist of.

b. Milestones

i. Checkpoint 1

This checkpoint consisted of making the basic pipeline that could handle all of the RISC-V instructions that were implemented in MP2. At this point, we did not consider stalls and forwarding paths as were working with our “magic memory” that would respond within one cycle. The test code for this checkpoint also consisted of NOP instructions, so we know that the data would be ready by the time the next instruction that utilized that register or memory location. The dual-port memory sets the mem_resp signal to high immediately, which means that we do not need to account for cache misses or memory stalls.

In appendix A, you can see the design for checkpoint 1. The design, like mentioned in the overview, consist of four interstage pipelines, namely IF/ID (ifid), ID/EX (idex), EX/MEM (exmem), and MEM/WB (memwb). These registers helped us keep track of signals that are used throughout the datapath, such as the PC for the stage, the control signals, and registers that are needed to resolve branches (which will be discussed in more detail later). On the left side of the IFID registers is the PC implementation. The PC mux selects between the output of the ALU which is calculated in the execute stage. The br_en signal lets us select whether we take the calculated address or if we

increment PC by 4 and pass that into the I-Cache to get the correct instruction. This is then passed into the IFID registers where we give the necessary signals to the Control ROM which outputs the control word that is used in the subsequent staged.

We also implemented a comparator and Jump Logic unit that allows us to know when a branch is taken and whether we need to account for this. The comparator unit can be seen in Appendix A. This signal takes in the two inputs from rs1 and either rs2 or i_imm (depending on the instruction), and then will compare them depending on the type of comparison needed. This will then output a br_en_cmp signal which is passed into the JMP logic (design can be seen in appendix A). This signal is then compared to the opcode and if the opcode is a branch instruction and our comparator output is also high, we make sure that the instruction is only one of the jump instructions and will set this output equal to the br_en signal that goes into the pcmux. For unconditional jumps, we check if the instruction is a JAL or JALR, and if so, do the same XOR as shown in the diagram to set pcmux.

The last black box section we have in our CP1 design is the wData logic. This logic is used to determine what bytes we want to shift and store when we have sw, sh, and sb instructions. We shift the data from exmem_alureg_out by 3 to convert this to the bytes, and then set the dcache_mbe signal accordingly so that we only mask the bytes we want for a particular instruction.

In terms of testing for this checkpoint, we went through each instruction by writing small test programs and seeing if the outputs in the registers were the values that we expected them to be. We also ran the CP1 given test program and went through every instruction in ModelSim. We then created designs for the forwarding path which can be seen in Appendix A under the CP2 design section.

ii. Checkpoint 2

After completing the basic pipeline in checkpoint 1, checkpoint 2 required us to implement hazard detection and forwarding, while also having a static-not-taken branch prediction unit. Additionally, we had to implement an arbiter which would act as a control unit for both I-Cache and D-Cache. There were two forwarding units that we had: one in the execute stage, and the other in the mem stage. The forwarding unit logic in appendix A shows the inputs, outputs, defaults, and the conditions that we forward on. Appendix A also includes the mem forwarding unit logic. These conditions are whenever there is a data dependency that we can pass on the value for. Depending on the instruction and if the destination register in the mem stage is the same as the

one for the execute stage, we forward the value. We created new muxes that go to the alumux2 and the comparator.

Since this checkpoint was not connected to the dual-port magic memory anymore, we needed to account for stalling in the pipeline as the memory could not be fetched in one cycle anymore. We had a stall signal for D-Cache and I-Cache. The D-Cache signal was set to active if the dcache_read or dcache_write signal was active and if there was no dcache_resp signal active. The I-Cache stall signal was set if there was no icache_resp and if the icache_read signal was high. We then set the flush signal to whenever there is a branch instruction, and we are not stalling. This combination allows for a static-not-taken branch.

The arbiter is used in order to control whether we prioritize the D-Cache or I-Cache. In appendix A, we can see the Arbiter Control Unit State Machine which is the FSM that controls how we transition between icache_read, dcache_read, and d_cache write. The Arbiter diagram in appendix A shows the high-level inputs and ouputs of the arbiter to the D-Cache and I-Cache. The arbiter design was later on modified when introducing the prefetcher in checkpoint 3, but this will be discussed later on in the report.

iii. Checkpoint 3

Checkpoint 3 was when we added all of our advanced features. Integrating all these complex components into our existing CPU brought performance improvements but also a lot of challenges which we had to work through. We began by proposing high level diagrams to our TA which we got feedback on. Then we implemented our designs based on these diagrams. The advanced features are discussed in depth in their respective sections and the appendix so here I will focus more on how we integrated them.

We began by building out our branch prediction system. This system was composed of the tournament predictor, branch target buffer, and later on the return address stack. Since this was a fairly large and complex part, all three of us worked together on it. The branch target buffer was created and integrated first by itself. We used our diagrams as a starting point and as we progressed we realized some of the details that we had missed and added them in. Connecting the BTB was the first time that we had to significantly modify some critical parts of the code. We faced many issues and eventually did get it to work. The main thing we figured out was that the branch prediction system does not necessarily live in a single stage. It connects to multiple, specifically the fetch and execute stages. Everything that has to do with writing to or modifying the BTB happens in the execute stage. Everything that has to do

with hits and reads happens in the fetch stage. Up until now, we had our BTB connected but did not have a branch prediction system.

The next step was making the branch predictors which we did fairly quickly. We followed some guidelines from lecture slides and made parts of our code reusable which was really efficient. Again, we did not make it correctly the first time and had many additions along the way as we figured out more of what we needed. The modules were similar to the BTB in that they connected between the fetch and execute stages. The predictors were updated and loaded when a branch was executed and they were read or indexed in the fetch stage. We also had to update our flush signal to no longer be static not taken. The toughest part about this was dealing with how to load the PC and resetting it on mispredictions. We had to redo our pc input logic to allow for a whole new set of possible cases. We had to choose between incrementing by 4, loading the btb target address, loading the alu output, or restoring the pc in the execute stage for mispredictions. There were also many edge cases where you had predictions within a misprediction and we had to account for those as well. We ran the test code over and over again fixing one edge case after another. It was a tedious and time consuming task but eventually it all worked. It was possibly the most time consuming part of the checkpoint.

After we were confident that the BTB and tournament predictor worked as wanted, we started to see the deadline coming up soon and decided to split the features up individually. We had one group member begin work on the basic prefetching system, another on the write buffer, and the other on cache improvements. This allowed us to work faster as a group. When we had all completed our parts we integrated them together.

The first part to be integrated was the write buffer. The buffer was simple to attach because it was a part of the memory hierarchy and it had a higher and lower level port. We began debugging by attaching it between the arbiter and cache line adapter. The way that the EWB was set up required the read and write signals to be sent one after the other. Our arbiter however, was not doing exactly that. We had to tweak the way that those signals came through to make sure the EWB was actually being used. After that issue was solved, everything worked as expected.

The next part to be added was the L2 cache. Similar to the EWB, this was also a plug and play in our memory hierarchy. The L2 cache was based off of our L1 cache which we knew worked for sure. The only thing that changed was the number of sets from 8 to 16. The cache was a shared once that sat between the arbiter and the EWB. The EWB was moved to connect between the L2 cache and cache line adapter.

Next we introduced the prefetching system. Thanks to some good planning, we were able to easily connect this as well. We made a copy of our existing L1 cache and arbiter and modified those to be prefetching compatible. We had to add some intermediate wires to connect the I-cache to the prefetcher and new arbiter. Every time the I-cache did a read, the prefetcher would start another read after it was done. An edge case that we had to cover here was if the I-cache started another read while the prefetcher still hadn't completed it's previous fetch. We solved this issue by adding a flag to our prefetcher and only initiating a prefetch if there wasn't already one in progress. In total, the prefetching system added a modified L1 cache, modified arbiter, and prefetch unit.

At this point, we had everything working and still had time to add in one more advanced feature. We chose to do the return address stack. We did this because we had a decent understanding of what it did and how to implement it. Having a day left, we were confident that we could get it done. It also had a lot of upside for our performance as well. Building out the stack itself was relatively quick. The official RISC-V specification was a great reference that told us exactly what we needed to implement. The stack was modified according to the spec in the execute stage and peaked in the fetch stage for predictions. The hard part came when we had to integrate it. Like with the BTB, we had to tear out our PC input logic and redo it. Along with all the cases that we were already previously covering, we now had to deal with all the RAS cases. We had to carry through a lot of signals from the fetch stage to the execute stage to check for prediction correctness. Unlike branches, you have to also compare the target address with the alu output to check if the JALR prediction was correct. We took it step by step and made sure we had all of our cases covered and the logic checked out. As always though, some edge cases did slip through and the given test code helped us resolve those.

We had now successfully implemented all of our advanced features and were very satisfied with how things went. The approach we took to this checkpoint was modular. This was our intention from the beginning so that it could be easy to debug and narrow down issues. All of our advanced features were more or less a module that we could easily disconnect if needed. Debugging was easier because we could isolate single modules from others. This method proved to be useful again and again throughout the course of this MP.

A final detail to add about this checkpoint was the effect of all this to our FMax. All of our features in the memory hierarchy took a toll on the FMax. There were several long critical paths that came from the datapath and worked their way up through the hierarchy. We figured that this was because of all the one cycle hits that we did. Since it's all combinational, the delays all added up

in the end. At the time however, we did not concern ourselves too much about it. We chose to tackle that problem in the next checkpoint.

iv. Checkpoint 4

A big part of checkpoint 4 was adding enhancements to our processor to improve our program runtimes and power consumption. Nevertheless, we did uncover and debug a critical error in our pipeline: incomplete support for the SLTU and the SLT instructions. Upon debugging, we realized that we had not forwarded the inputs of the comparator unit yet and did not accurately pass the comparator result to the register file input. As a result, we had to modify the pipeline to pass the entire 32-bit branch result through the stages, so that it can be written back correctly to the register file. Furthermore, we also had to make the comparator use the inputs from the forwarding unit instead of from the pipeline stage.

There were several timing issues that arose from the branch prediction and load operations. In our previous checkpoints, we were passing the branch enable signal and the branch address directly from the comparator and the ALU units respectively. This then created a long combinational path where the inputs for the branch address computation would have to be forwarded from the EX_MEM stage, meaning the signal would have to go through the forwarding unit and the ALU before finally reaching the branch prediction unit. To solve this compounded logic delay, we decided to move the branch resolution to the EX_MEM stage instead of the ID_EX stage. Furthermore, we also added an adder unit to compute the branch addresses, so that the path of the inputs to the branch prediction unit would not be hindered by the ALU, which contains multiple combinational levels. By adding these two changes, we were able to significantly improve our timing and increase our frequency to above a 100 MHz. Creating the adder unit also helped us in reducing the combinational delay for the load instructions, as by creating this unit, the address computing unit for loads does not need to get input from the ALU.

While performing the timing analysis on our processor with the L2 cache, we saw that connecting to the L2 cache increased the delay of our critical path, which decreased the frequency of the processor to below a 100MHz. To solve this issue, we added a buffer between the arbiter and the L2 cache, which fixed this timing issue. While the number of cycles for our program slightly increased, the improved timing allowed us to achieve better performance.

Nevertheless, to improve the Fmax of our code, we had to make a few more changes to our processor. Specifically, we had to remove a few advanced design units and make modifications to the other advanced units. We found that since the Prefetcher and the Eviction Write Buffer did not improve the

performance significantly but did add more combinational delays to the processor, those units had to be removed. Moreover, we modified the L2 Cache to have 16 sets per way instead of just 8 sets per way. We explored increasing the size of the Branch Target Buffer and the Return Address Stack, but the increase in combinational path delay did not justify the slight improvement in the performance that the modification gave. As a result, we kept the original sizes of the BTB and the RAS, 32 and 8 respectively.

To improve our timing further, we decided to explore Quartus's fitter settings, namely the fitter mode and the fitter seed. After trying out various seeds, we saw that our fitter performed best on the default initial seed of 1. However, by modifying the fitter mode from Balanced to Aggressive Performance, we were able to increase our frequency by around 5 MHz. Although the modification in fitter settings increased the power of the processor, the performance had improved well enough to offset that, as a result of which our raw scores for our design competition improved.

c. Advanced Design Options

i. Tournament Branch Predictor

1. Design

For the tournament branch predictor, we made a local branch predictor and a global branch predictor as the two predictors we would select between. Our branches were resolved in execute, and because branch instructions could lead to stalling and fetching the wrong instructions, we decided to implement a tournament branch predictor to improve our performance. The local branch history table was designed as a pattern history table, which can be seen in appendix A. Our tables were parameterized so that we could experiment with the ideal sizes. Each entry holds a particular branch of the program and is controlled by a 2-bit counter. The counter represents whether we predict the branch to be weakly/strongly taken or not taken.

The global branch predictor is made up of two levels: The first being a Branch History Register (BHR), and the second being the Pattern History Table explained above (can be seen in appendix A). The branch history register keeps track of whether a branch was taken or not taken, which is noted by a bit 1 or 0, respectively. As more branches are encountered in the program, we left shift in order to bring in the new value.

These two predictor outputs are then fed into another 2-bit counter that denotes whether the local branch or global branch predictor output

should be used as the predicted branch output. The FSM updates based on whether the branch was taken by the cpu_br_en signal. The tournament_br_sel takes the first bit from the FSM and selects the local or global predictor output if it's a 0 or 1, respectively. The high-level design can be seen in appendix A, and the inputs, outputs, and descriptions of the signals can be found in the table in appendix B.

2. Testing

In order to test our design, we designed our own test program with multiple branches to see whether the tables were getting the correct values. The BTB was included in the test as branch predictors only make sense to implement with a BTB, as we need to know what address we need to go to for the branch. We also printed out the state of the FSM for the local, global, and tournament predictors to see if the states were being updated accordingly. After verifying this, we ran the checkpoint codes to see if this reduced stalls and increased our speed, which we'll discuss in the next section.

3. Performance Analysis

After implementing the tournament predictor, we noticed a significant increase in the performance of our processor. While running the checkpoint 3 code, we saw that the total number of cycles spent flushing decreased from 9369 to 3357. The total number of branches were 12774, and our tournament predictor predicted 12066 of these correct, forming a 94.5% accuracy. After modifying our MP3 cache to have a one-cycle hit and implementing the BTB/Tournament Branch Predictor, the number of I-Cache misses also decreased dramatically. The predictor works well with workloads that includes frequent branches, as we now know where and when the instruction is going to. The full analysis table of our performance counters and runtime calculations can be seen in the Additional Observations section.

ii. Branch Target Buffer

1. Design

The goal of the Branch Target Buffer (BTB) is to give us the target address of a branch if it is taken so that we can have the address ready beforehand (design in appendix A). The BTB and the tournament predictor work hand-in-hand in order to form an accurate branch predictor. Once we implemented the branch predictor, we had to change the logic of how we choose our address. This can be seen in

the CP3 design in Appendix A. The PC Input Logic, as explained earlier, lets us select the address that was given from our BTB/RAS. The BTB is loaded with the address in the execute stage, whenever a BR or JAL instruction is resolved. It is then read in the fetch stage when we see another one of these instructions. The design of the BTB is pretty simple, but we had to make sure we took care of the logic of when we use the BTB address and when we had to fetch the address.

2. Testing

The testing process for the BTB was in tandem with the tournament branch predictor. After a BR or JAL instruction was executed, we update the BTB with the address the BR/JAL instruction goes to so that if we encounter the BR/JAL again, we know what the predicted PC output should be. We checked if the BTB was loaded whenever we had a branch instruction, and the predictor was right. We also made our BTB parameterizable so that testing the different sizes was possible. Once we saw that the values of the BTB were being updated and the BTB predicted addresses were being loaded in the pipeline and held the correct value, we tested with the various checkpoint test codes to make sure they also worked with our BTB.

3. Performance Analysis

Like mentioned above, the BTB and Tournament Branch Predictor had the same performance, as they must be implemented together. As seen in the performance analysis section of the tournament branch predictor, we noted a significant improvement after implementing these modules. The BTB sizes did not make much of a difference for us, as the test programs that we had to run for the checkpoints and competitions were relatively small. We chose to have a BTB of size 32. Because the BTB can only take care of jumping to particular addresses, the BTB does not help too much with JALR instructions. This is why we decided to implement a Return Address Stack (RAS), which is described in the section below.

iii. Return Address Stack - Aayush

1. Design

The return address stack serves the same purpose as branch prediction except for JALR instructions. The high-level diagram can be found in appendix A. The core of the RAS is the stack and following the official RISC-V specification, we created push and pop signals for it. When we detect a JALR instruction in the execute stage that’s a

function call, we push to the stack. In the fetch stage, if we detect that the instruction is a JALR return instruction, we peak the top of the stack and load that PC to queue up the predicted instructions in the pipeline. Once that JALR gets to the execute stage, it gets popped off the stack. If we find that there is a mismatch between the predicted target and the actual target, we flush the pipeline and recover. If the stack is full, we stop pushing to it.

The RAS works in combination with our tournament branch predictor and branch target buffer to accurately predict every branch or jump taken in our CPU.

2. Testing

Like we did with every other feature, we started by writing a small test program that called a function and returned from it. We monitored the contents of the RAS and confirmed that they were what they were supposed to be. Once we got our small program to work, we started to run some of the given test code on it. We monitored the stack again and made sure that the predictions were lining up and flushing when needed. Something that we monitored more closely was the behavior when the stack became full. We had considered a stack full policy but wanted to see what happened without it. The RAS stopped pushing new addresses to the stack when it was full and when the function calls started returning, the first couple were mis-predicted and flushed but then they started to hit. This was expected and acceptable behavior to us if we did not want to make the stack bigger.

3. Performance Analysis

We got some metrics from our CPU with and without the RAS. We found that the RAS was a clear success and, as long as the stack didn’t completely fill up, it was almost 99% accurate. It also decreased the number of flushes in the pipeline by around 15%. When the stack did full up, the accuracy dropped because it was not able to hold enough nested function calls. Since we parameterized our stack, we easily changed its size and re-did some performance analysis. We found that the max number of nested function calls from all the test code was only barely above our original size. Because of this, we decided that doubling the size was not worth it and reverted back to a smaller size.

What we did here was optimize the size of the RAS for the given workload. Had the programs been larger and had more nested function calls, like in a large recursive program, a larger stack size would have been better. But because of test code that we were going to be evaluated on, we were able to choose a smaller, more efficient size.

iv. L2 Cache

1. Design

Observing that a big bottleneck in our processor's performance was our memory access time during a cache miss, we wanted to find a way to reduce the average time taken to fetch data during a cache miss. Understanding that adding a bigger cache between the two units would reduce the miss latency, we decided to add a Level-2 cache (L2 Cache). To make the bus adapter logic simpler, we decided to make the cache-line size in our L2 cache the same size as our instruction and data caches. However, we parameterized the number of sets in this cache, and increased the number of sets to 16.

2. Testing

We connected the RISC-V monitor, the shadow memory and the param memory modules to our testbench. From there, we wrote some simple test programs which contained addresses mapping to different cache indices to ensure that adding the L2 cache did not affect the functionality of our program. To see if adding the L2 cache had any effect on our program, we measured the cycles spent on I-Cache and D-Cache stalls. Since a reduced miss latency means that the processor would stall less when fetching data from either the I-Cache or D-Cache, we determined that the cycles spent on pipeline stalling would be a good proxy to the time spent on cache misses, as a result of which we measured that signal and observed any associated waveforms. In addition, we also measured the cycles spent on cache hits, since by adding the L2 cache, we would be able to reduce the amount of cache evictions, resulting in important cache lines staying in both the I-Cache and D-Cache and thereby improving performance. As a final check, we tested our processor on all the competition and checkpoint codes and debugged any errors that came during the simulations.

3. Performance Analysis

To quantify the improvement that the L2 cache provided, we compared a version of the processor which had branch prediction, the return address stack and the prefetcher with another version that was identical to the base processor except for the added L2 cache. The metrics we used to observe the change in performance were I-Cache stall time and D-Cache stall time. We saw that by adding the L2 cache, we were able to decrease the I-Cache stall time by 17.06% and decrease the D-Cache stall time by 27.98%.

We also wanted to see if by adding the L2 cache, we could reduce the number of evictions in the I-Cache or D-Cache, thereby increasing the time spent on hits. To quantify this, we compared a version of the processor with just the branch prediction and return address stack to another version that had the same features but also had the L2 cache. We observed that by adding the L2 cache, the number of cycles spent on I-Cache hits increased by 23.63% and the number of cycles spent on D-Cache hits increased by 2.64%.

Through our analysis, we saw that our L2 cache provided tangible improvement to our processor, because of which we added it to our design competition submission.

v. Eviction Write Buffer

1. Design

The high-level diagram and description of our write buffer can be found in the appendix. The purpose of this buffer is to prioritize reads overwrites in our memory hierarchy. We observed that when d-cache was evicting a dirty line, the pipeline would stall for a long time, almost twice as long as any other stall. This was because the cache had to propagate a write up through the memory hierarchy, wait for a response, then send another read and wait for another response. Adding this buffer swaps the order of this. The buffer first intercepts the write request and sends a response back to the d-cache. When the read comes in on the next cycle, it lets the read pass through before completing the write. It essentially cut our stall time in half when evicting dirty lines.

The buffer itself sits in between two levels of memory and can hold a single cache line. To check for hits, it also stores the full 32-bit address and keeps a valid bit. On startup, the valid bit is 0 to show that the buffer is empty. The buffer also has single cycle hit times to minimize the effect it would have on normal read and write requests from the d-cache.

2. Testing

To test the write buffer, we placed it in our datapath with nothing other than the D-Cache to make sure that there were no outside issues being introduced into the system. Once we had it connected properly, we wrote a short and simple piece of test program that would evict a dirty cache line from the D-Cache. We monitored the contents of the registers inside the buffer to make sure it captured the eviction and passed through the read request before completing the write request.

We also confirmed that the stall time was decreasing as we were expecting. After running our own test program, we ran all the given test programs we had and sorted out any minor issues that we had with those.

3. Performance Analysis

To assess the performance of the buffer, we ran different test code on it and marked down the results of our counters as well as runtime and Fmax. We then compared these to metrics that we had without the EWB. We found that results to greatly vary with the test code that was being run. It turned out that the majority of programs we were running had a minimal number of dirty evictions. For ones that had a decent number of dirty evictions, the benefits of the buffer were definitely seen. The amount of time spent in a d-cache stall decreased by 28%.

Adding the buffer also affected our CPUs Fmax. Since the L1 caches were single cycle hit, and the EWB was made single cycle hit as well, we got a longer critical path. This dropped our Fmax by a couple megahertz. We tried to move it around to different places in our hierarchy and figure out where it performed best. The FMax issue still persisted however. Ultimately, it became a trade where we got a faster runtime with a slower FMax.

For the final checkpoint, we found that the test programs did not have too many dirty evictions. Where the EWB is really beneficial is in large programs where there are a lot of evictions and writing to memory. Because of this, for our specific test code, we decided that the small performance gain was not worth the FMax drop and added power usage.

vi. Basic Hardware Prefetching - Vishal

1. Design

We implemented a One-Block-Lookahead Prefetcher, which we attached to the instruction cache. Every time a miss occurred in the instruction cache for block index i, a request is sent concurrently to the prefetcher with the address that just got missed. The prefetcher then sends a request to the arbiter to fetch the cache block/line for index i+1. If there are no icache or dcache requests, the arbiter then handles the prefetcher’s request and then sends the data back for block i+1. The prefetcher then lets the cache know that the data is ready, after which the cache loads the i+1 cache-line along with its associated tag

bits. In order to make the prefetcher, we had to modify the input and output signals of the cache and add a new state for the arbiter to process prefetch requests. Although we could have added ports to the data arrays in our cache so that the prefetcher can directly modify the arrays, we felt that doing so would significantly add combinational delays as we would have to handle simultaneous icache read requests as well as prefetch write requests. Consequently, we decided to simply add a new state to the cache's control machine so that prefetch writes and icache reads would not happen at the same time. A compromise we made was that the instruction cache would prioritize the prefetch write request over the icache read requests. While this resulted in unnecessary icache stalls, it nevertheless reduced situations where the icache would send a read request for the same cache block that the prefetcher recently fetched. The diagrams in the appendix give more details about the prefetcher's design and the modifications done to the icache and arbiter.

2. Testing

To test the prefetcher, we attached the RISC-V monitor, the shadow memory and the param memory. We then wrote simple programs to monitor the prefetch signals to see that the prefetcher did in fact fetch block i+1 when the cache encountered a miss, and the cache loaded the prefetched data when the unit indicated to the cache that the prefetch request was completed. For this, we monitored the pf_cline_address and the cacheline_address to check that the address was actually i+1 apart, and that when the icache had a read for the same address, it did not get a miss. To further check if the prefetcher provided any performance improvements, we monitored the cycles spent on icache hits, the cycles spent on icache stalls and the number of icache misses. By fetching instruction blocks before they are accessed, we would increase the number of hits, and therefore the cycles spent on hits. By consequence, we would also reduce the number of misses and hence reduce the time spent on waiting for icache read requests to finish. As a result, these metrics were monitored during our testing process to check if the prefetching unit did improve the performance of the processor. We then ran all the checkpoint and competition codes to ensure that our prefetcher did improve our processor's performance when running these programs.

3. Performance Analysis

To analyze the performance improvement that the Prefetcher unit provided, we decided to run Checkpoint 3 code on two versions of our processor, one that contained the prefetcher and one that didn't. To

accurately see the performance improvement of the prefetcher, we removed the L2 cache and the Eviction Write Buffer from our processor and simply retained our branch prediction and return address stack components in both versions. To quantify the improvement in performance, we monitored the number of misses in the instruction cache, the time spent on icache stalls and the number of cycles spent on icache hits. By adding the prefetching unit and modifying the cache and arbiter accordingly, the number of cache hits reduced by 14.29% from 91 to 78, the cycles spent on icache stalls decreased by 20.95% from 5183 cycles to 4097 cycles, and the cycles spent on icache hits increased by 0.81% from 113633 cycles to 114550 cycles. Considering that our prefetcher only fetched the block for the next consecutive set (index) in the cache, it is not surprising that our icache hits and our icache misses did not improve significantly, especially since the CP3 code contained multiple subroutines. Nevertheless, it was surprising to find that the prefetcher provided vast improvements to the stall time. This can be attributed to the fact that the arbiter was not designed optimally, because of which the time spent on stalls was much greater than expected. It seems that by slightly increasing the number of hits, we are able to significantly decrease the stall time as we reduce the instruction cache's interaction with the arbiter.

When we started integrating the prefetcher with the L2 cache and the Eviction Write Buffer, we found that the improvements that the prefetcher provided overshadowed the improvements provided by the L2 cache and the debugger. Furthermore, adding the prefetcher increased the combinational delay and hindered our timing, as a result of which we did not include the prefetching unit and the modifications to the cache and the arbiter in the final design.

IV. Additional Observations

We found out that contrary to our expectations, the Quartus program in the EWS machines compiled our design to achieve a much better performance. While compiling on our local machines, we only reached 103 MHz, we were able to attain an FMax of 105.2 MHz on the EWS machine, with negligible increase in the power consumption of our processor.

	Comp1 (ns)	Comp2 (ns)	Comp3 (ns)	FMax (MHz)	Comp1 Adjusted Time	Comp2 Adjusted Time	Comp3 Adjusted Time	Average Comp Runtime (ns)	Adjusted Average Runtime (ns)	CP3 (ns)	CP3 Adjusted Time	Comp1 Power (mW)	Comp1 Score	Comp2 Power (mW)	Comp2 Score	Comp3 Power (mW)	Comp3 Score	Final Score
BTB (32), RAS (8), BR Pred	517,685	3,601,345	3,309,025	99	522,914	3,637,722	3,342,449	2,476,018	2,501,029	1,810,925	1,829,217							
BTB (64), RAS (16), BR Pred	545,235	3,594,405	3,304,525	100.61	541,929	3,572,612	3,284,490	2,481,388	2,466,344	1,810,985	1,800,005	766.31	1.22E-10	596.97	2.72E-08	513.20	1.82E-08	
BTB (32), RAS,(8), EWB Between D-Cache & Arbiter	545,745	3,599,675	3,309,125	95	574,468	3,789,132	3,483,289	2,484,848	2,615,630	1,477,365								
BTB (32), RAS (8), L2 Cache	544,035	1,553,685	1,037,615	89.09	610,658	1,743,950	1,164,682	1,045,112	1,173,096	1,472,225	1,652,514	833.97	1.90E-10	775.63	4.11E-09	680.35	1.07E-09	
BTB (32), RAS (8), L2 Cache w/ Buffer Between Arbiter/L2 Cache	544,395	1,602,805	1,105,575	100.6	541,148	1,593,246	1,098,981	1,084,258	1,077,792	1,495,465	1,486,546	797.61	1.26E-10	740.17	2.99E-09	634.94	8.43E-10	6.83E-10
BTB (32), RAS (8), L2 Cache & EWB Between L2 Cache & Cacheline Adaptor	546,405	2,876,755	3,060,225	97.28	561,683	2,957,191	3,145,791	2,161,128	2,221,555	2,066,185	2,123,957		0.00E+00					0.00E+00
BTB (64), RAS(16), L2 Cache	543,575	1,545,515	1,033,115	87.97	617,910	1,756,866	1,174,395	1,040,735	1,183,057	1,472,285	1,673,622		0.00E+00					0.00E+00
BTB (32), RAS(8), L2 Cache, EWB Between L2 Cache & Cacheline Adaptor, Prefetcher	541,545	1,613,245	1,086,985	80.99	668,657	1,991,906	1,342,122	1,080,592	1,334,229	1,210,165	1,494,215		0.00E+00					0.00E+00
BTB (32), RAS (8), L2 Cache w/ Buffer Between Arbiter/L2 Cache (AGGR PERF)	544,395	1,602,805	1,105,575	103.49	526,400	1,551,509	1,078,207	1,084,258	1,052,039	1,495,465	1,445,033	913.07	1.33E-10	851.16	3.18E-09	713.14	8.94E-10	7.23E-10
BTB (32), RAS (8), L2 Cache w/ Buffer Between Arbiter/L2 Cache (default)	544,395	1,602,805	1,105,575	100.6	541,447	1,594,908	1,104,886	1,084,258	1,080,413	1,495,465	1,486,546	849.45	1.35E-10	789.82	3.20E-09	667.98	9.01E-10	7.30E-10
BTB (32), RAS (8), L2 Cache w/ Buffer Between Arbiter/L2 Cache (AGGR PERF, EWS)	544,395	1,602,805	1,105,575	105.2	518,205	1,528,104	1,066,483	1,084,258	1,037,597	1,495,465	1,421,545	843.85	1.17E-10	783.17	2.79E-09	670.64	8.13E-10	6.44E-10

This figure shows the runtime and power calculations based off of estimated adjustments with Fmax. These were the different permutations we ran in order to find the best combination.

	Total branches	BTB correct	Total JALR	RAS correct	Tournament correct	icache Stalls	dcache Stalls	Total flushes	dcache hits	dcache misses	icache hits	icache misses
CP2 given cache	12774	X	X	X		343158	140377	9369	28900	61754	193564	6739
Branch Prediction w/ BTB	12774	9417	X	X	12066	3985	123425	3387	9775	1228	173806	91
L2 Cache with Prediction (BTB & RAS)	12774	9387	X	X	12069	5137	164487	3387	10033	1228	214868	214868
Prefetching with Branch Prediction (No L2 Cache)	12774	9389	X	X	12069	4923	123837	3385	9755	1228	174199	78
Everything except Prefetcher	12774	9389	X	X	12069	5183	63256	3385	9993	1228	113633	91
Everything except EWB	12774	9389	X	X	12069	4083	89182	3385	9543	1228	139546	78
Everything	12774	9389	X	X	12069	4097	64187	3385	9551	1228	114550	78
Everything (Comp1)	12371	8036	2978	1478	9959	838	373	2857	2922	7	50535	24

This figure shows the performance counter when running the test code from CP3. Note that the counters here count the number of cycles spent in the particular field.

V. Conclusion

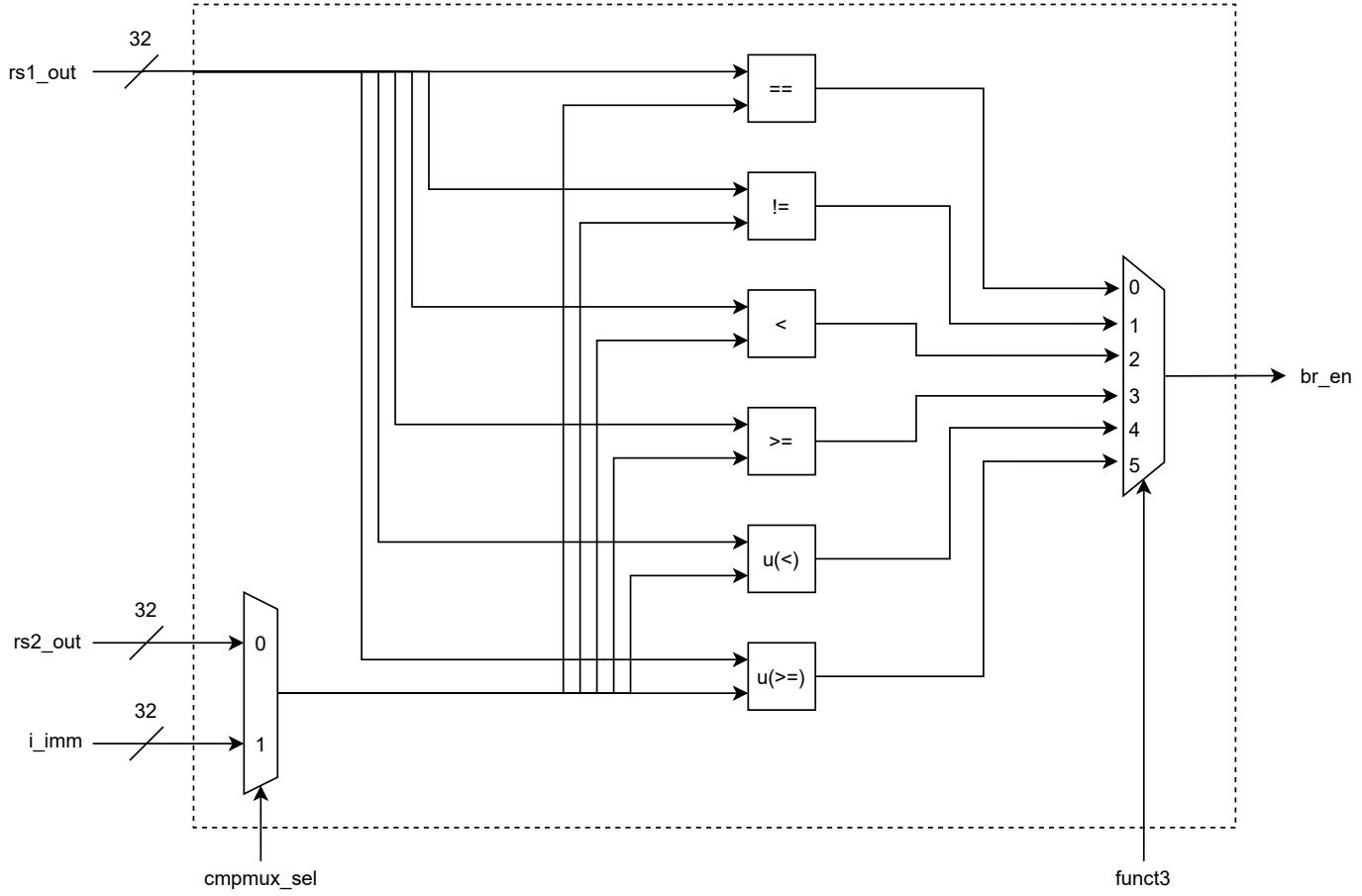
Through this project, we learned not only about Computer Architecture concepts such as pipelined processors and memory hierarchy, but also about hardware verification, engineering design and debugging. Furthermore, we also learned valuable lessons in communication and time management. Above all, we are glad to have designed our own working processor in an assembly language actually used by the industry. Looking back at about the last two months, we are pleased to say that we did indeed BigBrain the MP.

Appendix A

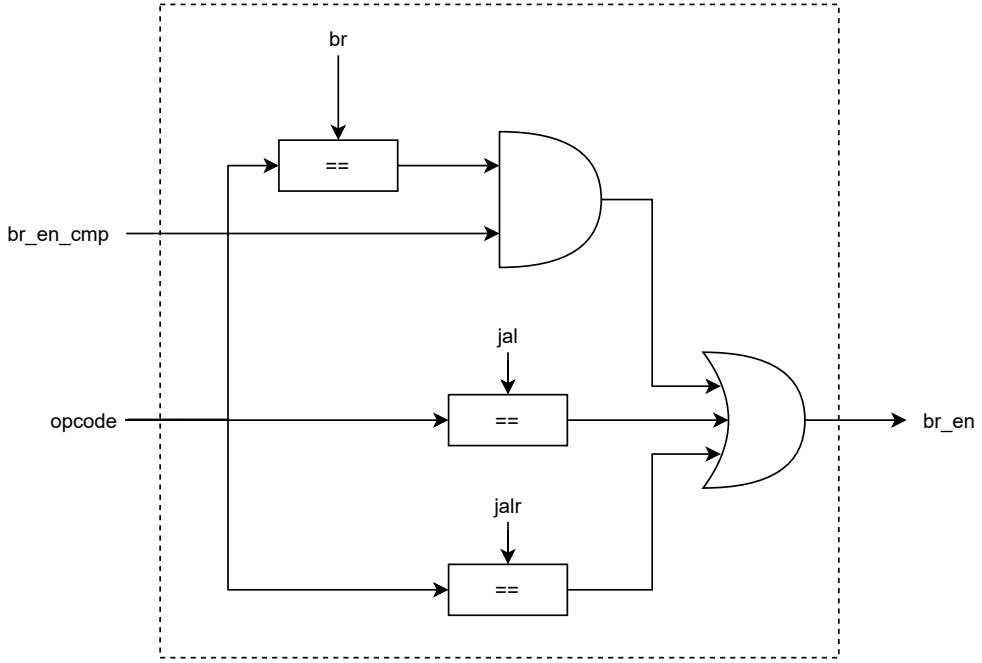
Diagrams

CMP

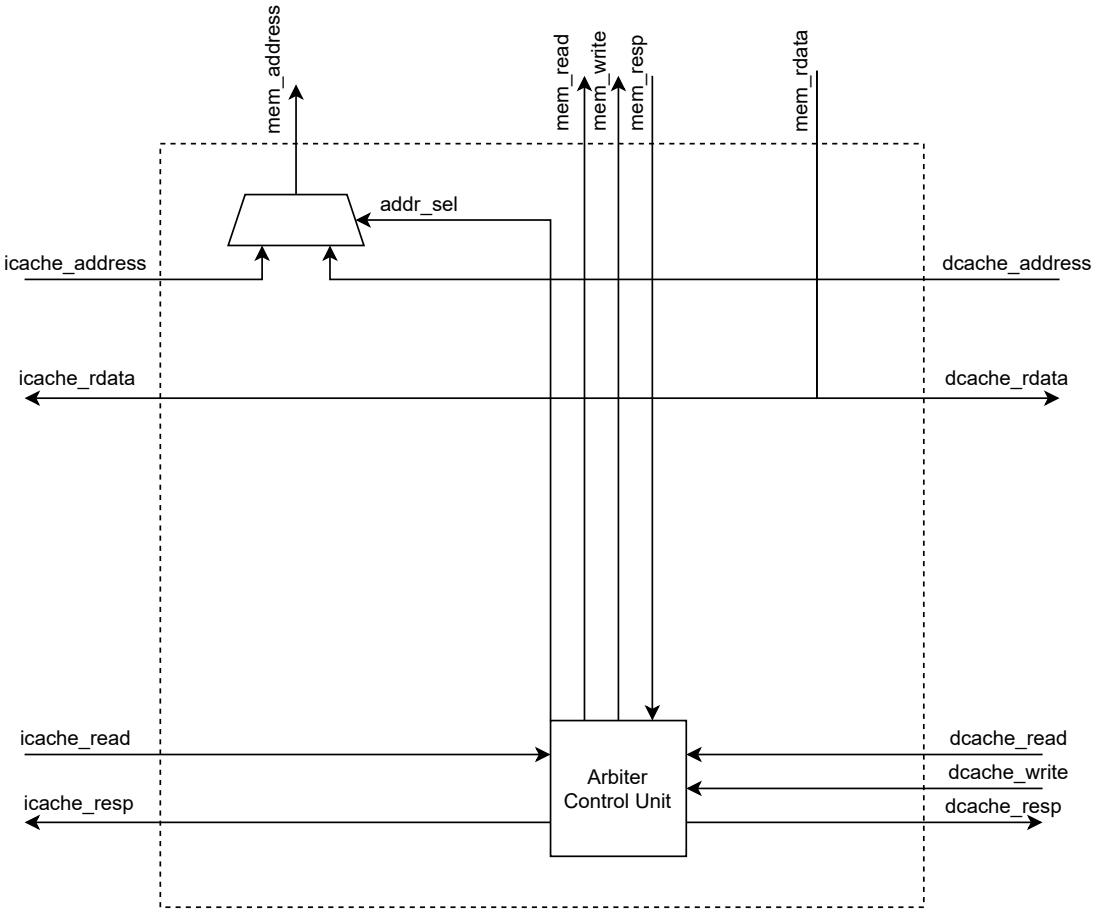
Key:
u(<) : Unsigned less than
u(>=) : Unsigned greater than equal to



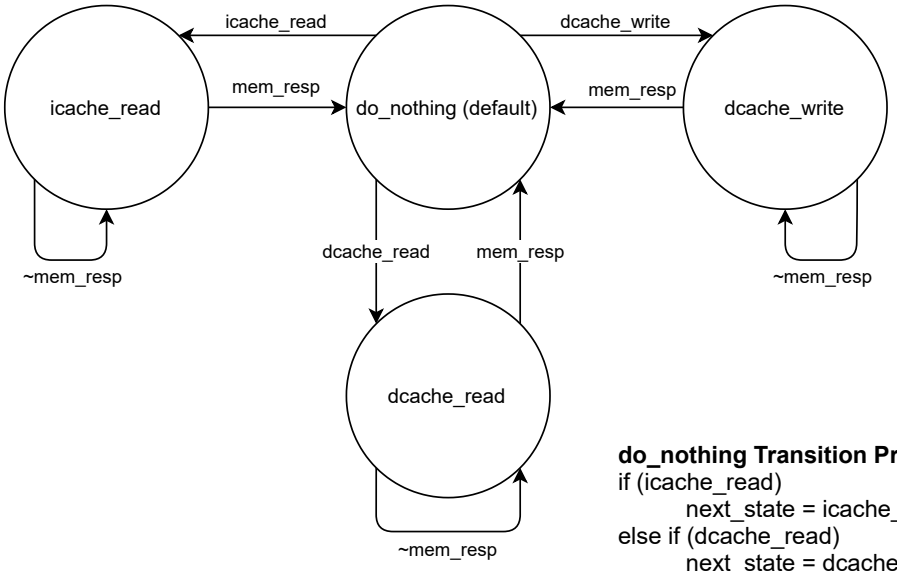
JMP Logic



Arbiter


mem_address
mem_read
mem_write
mem_resp
mem_rdata
addr_sel
icache_address
dcache_address
icache_rdata
dcache_rdata
icache_read
dcache_read
dcache_write
icache_resp
dcache_resp
Arbiter
Control Unit

Arbiter Control Unit State Machine



do_nothing Transition Priority:

```
if (icache_read)
        next_state = icache_read
else if (dcache_read)
        next_state = dcache_read
else if (dcache_write)
        next_state = dcache_write
else
        next_state = do_nothing
```

Output at each state:

do_nothing
- mem_read = 0
- mem_write = 0
- addr_sel = addr_mux::icache_addr
- icache_resp = 0
- dcache_resp = 0

icache_read
- mem_read = 1
- mem_write = 0
- addr_sel = addr_mux::icache_addr
- icache_resp = mem_resp
- dcache_resp = 0

dcache_read
- mem_read = 1
- mem_write = 0
- addr_sel = addr_mux::dcache_addr
- dcache_resp = mem_resp
- icache_resp = 0

dcache_write
- mem_read = 0
- mem_write = 1
- addr_sel = addr_mux::dcache_addr
- dcache_resp = mem_resp
- icache_resp = 0

Forwarding Unit Logic

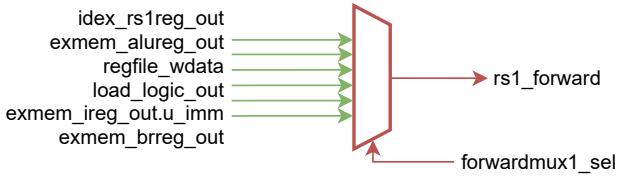
Inputs:
idex_ireg_out.rs1
idex_ireg_out.rs2
exmem_ireg_out.rd
exmem_ireg_out.opcode
memwb_ireg_out.rd
exmem_ctrlreg_out.regfile_ld
memwb_ctrlreg_out.regfile_ld

Outputs:
forwardmux1_sel
forwardmux2_sel

Defaults:
forwardmux1_sel = forw_mux::idex_rs1reg_out
forwardmux2_sel = forw_mux::idex_rs1reg_out

Forwarding Mux

* For both RS1 and RS2




```
// Mirrored for RS2

if (exmem_ctrlreg_out.regfile_ld
&& exmem_ireg_out.rd != 0
&& exmem_ireg_out.rd == idex_ireg_out.rs1)
        if (exmem_ireg_out.opcode == op_load)
                forwardmux1_sel = forwardmux1::load_logic_out;
        else if (exmem_ireg_out.opcode == op_lui)
                forwardmux1_sel = forwardmux1::mem_uimm;
        else if ((exmem_ireg_out.opcode == op_imm
        || exmem_ireg_out.opcode == op_reg)
        && (exmem_ireg_out.funct3 == 3'b010
        || exmem_ireg_out.funct3 == 3'b011))   //SLT
                forwardmux1_sel = forwardmux1::cmp_br;
        else
                forwardmux1_sel = forwardmux1::exmem_alu;
else if (memwb_ctrlreg_out.regfile_ld
&& memwb_ireg_out.rd != 0
&& !(exmem_ctrlreg_out.regfile_ld
&& exmem_ireg_out.rd != 0
&& exmem_ireg_out.rd == idex_ireg_out.rs1)
&& memwb_ireg_out.rd == idex_ireg_out.rs1)
        forwardmux1_sel = forwardmux1::regfile_wdata;
else
        forwardmux1_sel = forwardmux1::idex_rs1;
```

Mem Forwarding Unit Logic

Inputs:
idex_ireg_out.opcode
idex_ireg_out.rs2
memwb_ireg_out.rd
memwb_ctrlreg_out.regfile_ld

Outputs:
memforw_mux_sel

Defaults:
memforw_mux_sel = memforw_mux::exmem_rs2reg_out

```
if (exmem_ireg_out.opcode == op_store
&& memwb_ctrlreg_out.regfile_ld == 1'b1
&& memwb_ireg_out.rd != 0
&& memwb_ireg_out.rd == exmem_ireg_out.rs2)
        memforwmux_sel = memforw_mux::regfilemux_out
else
        memforwmux_sel = memforw_mux::exmem_rs2reg_ou
```

PC Input Logic

Input signals originating in fetch stage:
use_ras: if a ret instruction is detected
btb_hit: if the btb has a target address for a branch
pred_br: predicted branch by the tournament predictor
pcreg_out: pc register output
ras_addr_out: target pc for ret instruction

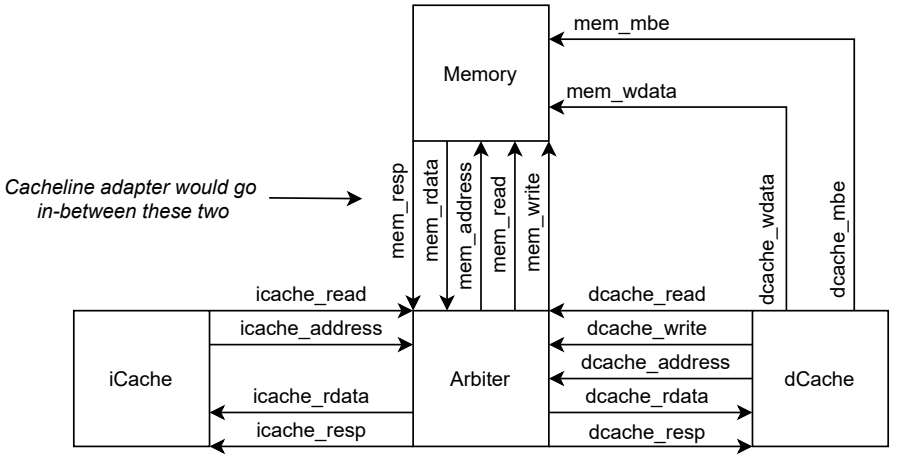
Input signals originating in execute stage:
alu_out: calculated offset/address
br_en: if branch was taken
idex_*

Outputs:
pc_input: input of pc register

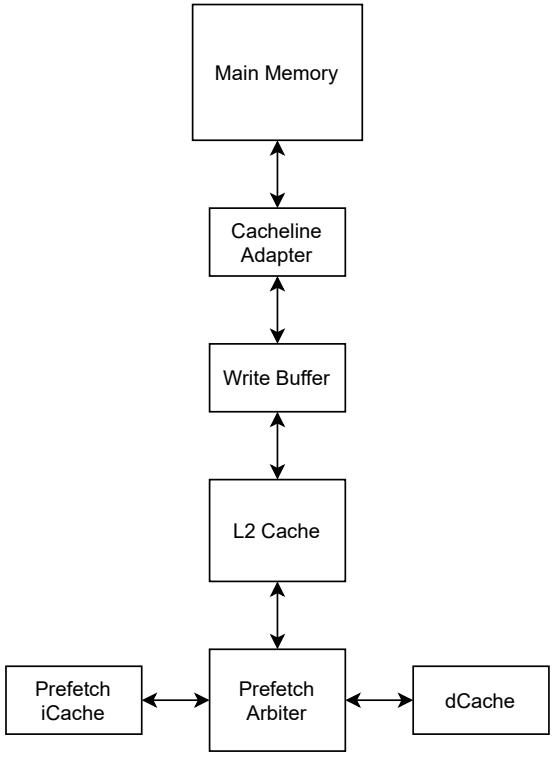
* For CP4, to optimize performance, branch resolving was moved from execute to mem and the PC input logic was modified accordingly.

```
btb_correct = idex_btb_hit & idex_pred_br
ras_correct = idex_use_ras & idex_ras_addr == alu_out

case ({btb_correct, ras_correct})
        2'b00: pcmux_out = pcreg_out + 4;
        2'b01: pcmux_out = idex_pcreg_out + 4;
        2'b10: pcmux_out = {alu_out[31:2], 2'b0};
        2'b11: pcmux_out = pcreg_out + 4;
endcase

if (use_ras) begin
        if (idex_ireg_out.opcode != op_jalr
        && (br_en == btb_correct))
                pc_input = ras_addr_out;
        else if (idex_ireg_out.opcode == op_jalr
        && ras_correct
                pc_input = ras_addr_out;
        else
                pc_input = pcmux_out;
else if (pred_br & btb_hit
&& ((br_en == btb_correct)
|| (idex_use_ras_out && ras_correct)))
        pc_input = btb_pc_out;
else
        pc_input = pcmux_out;
```

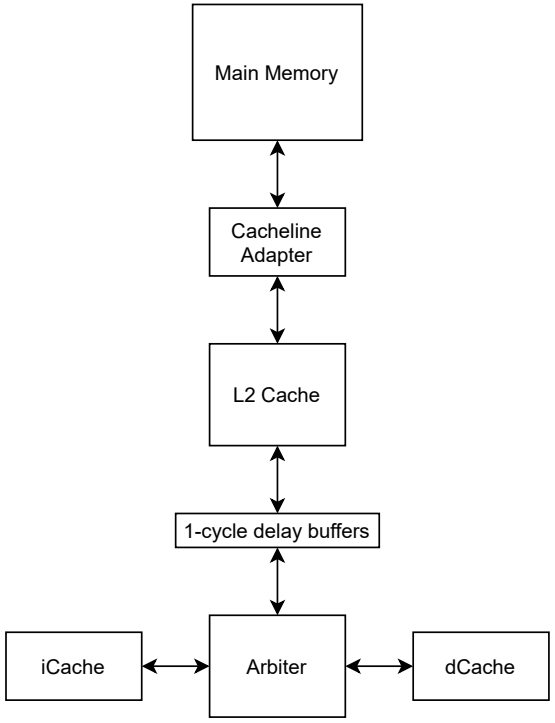
CP2 Memory Hierarchy



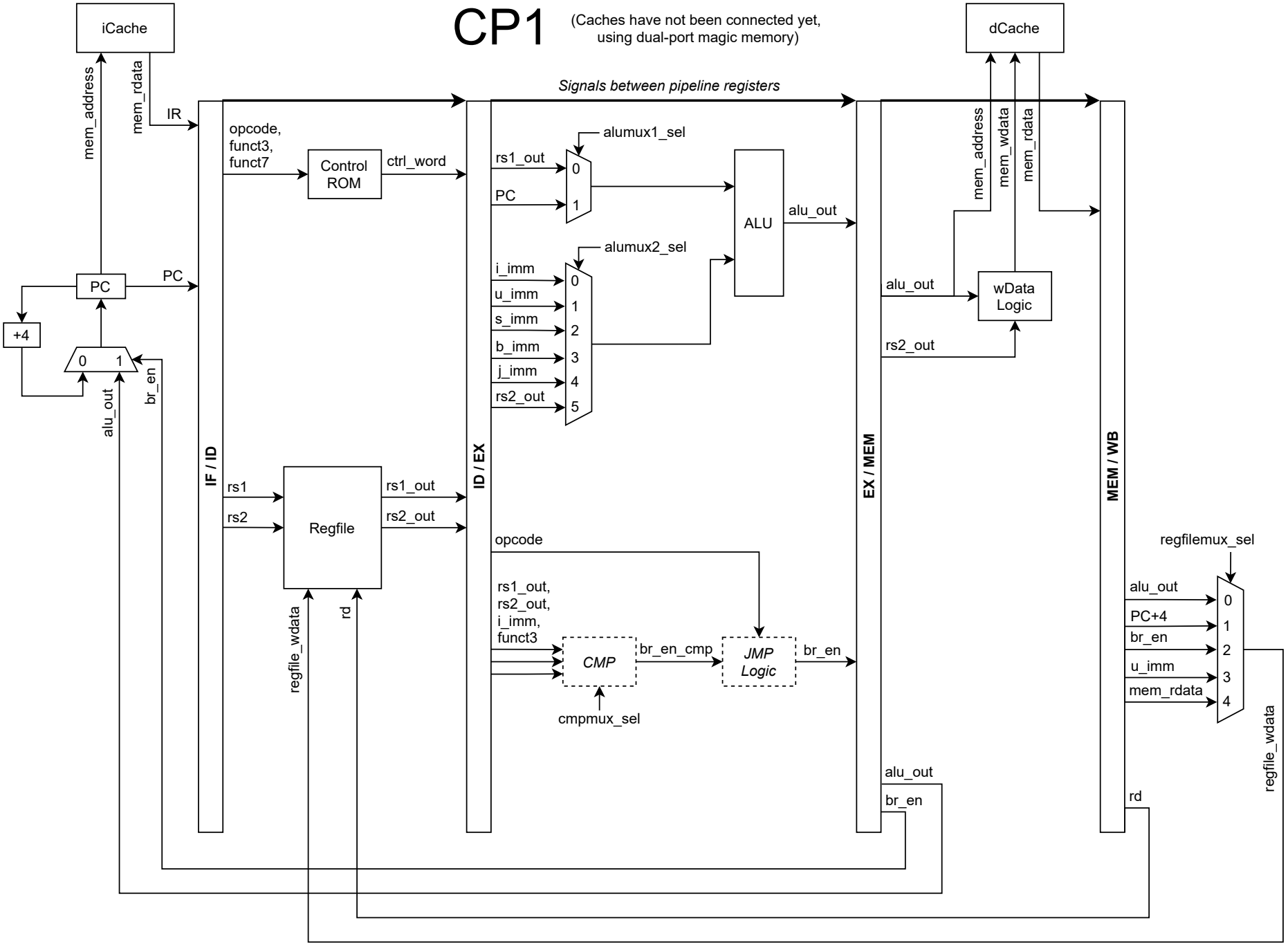
CP3 Memory Hierarchy

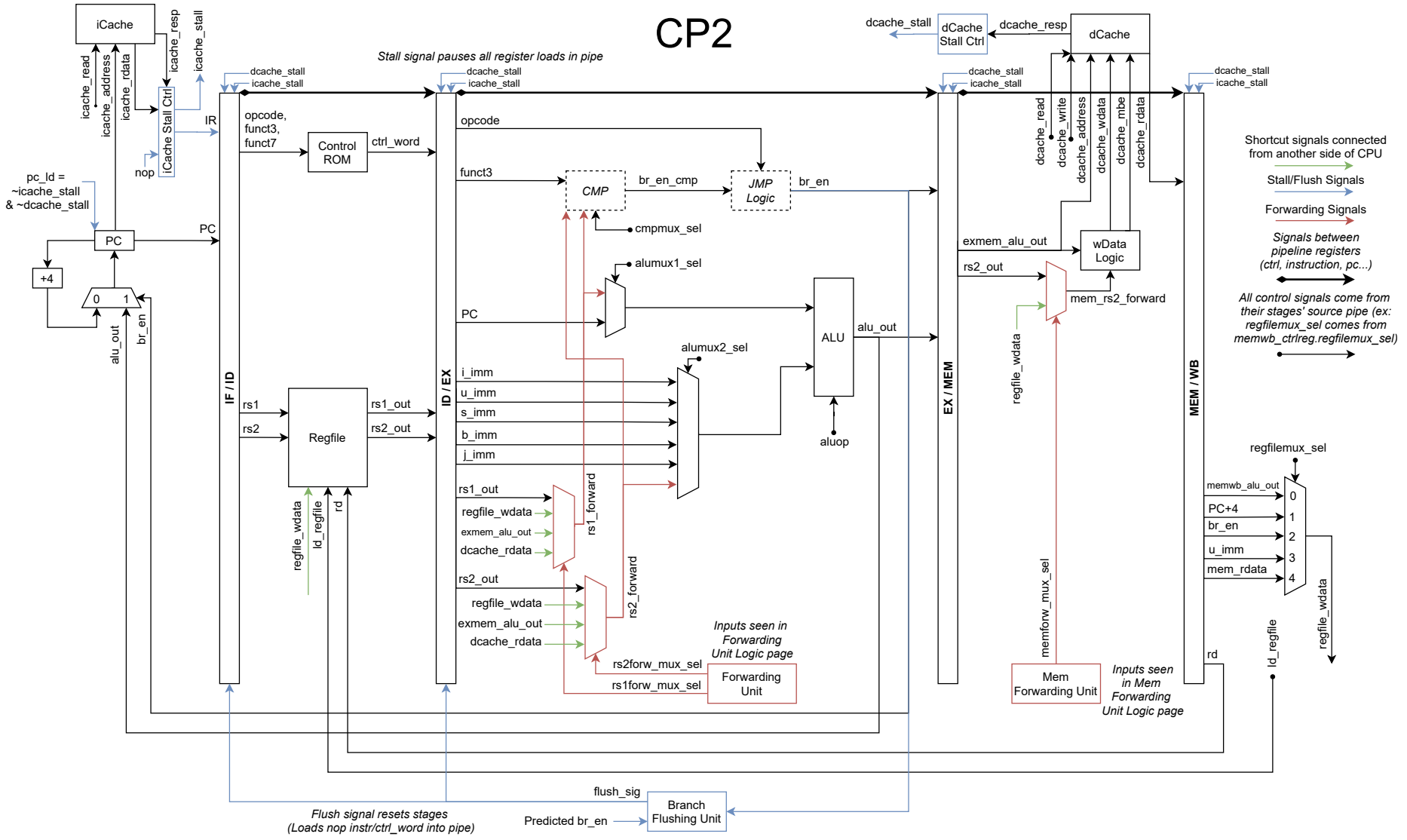


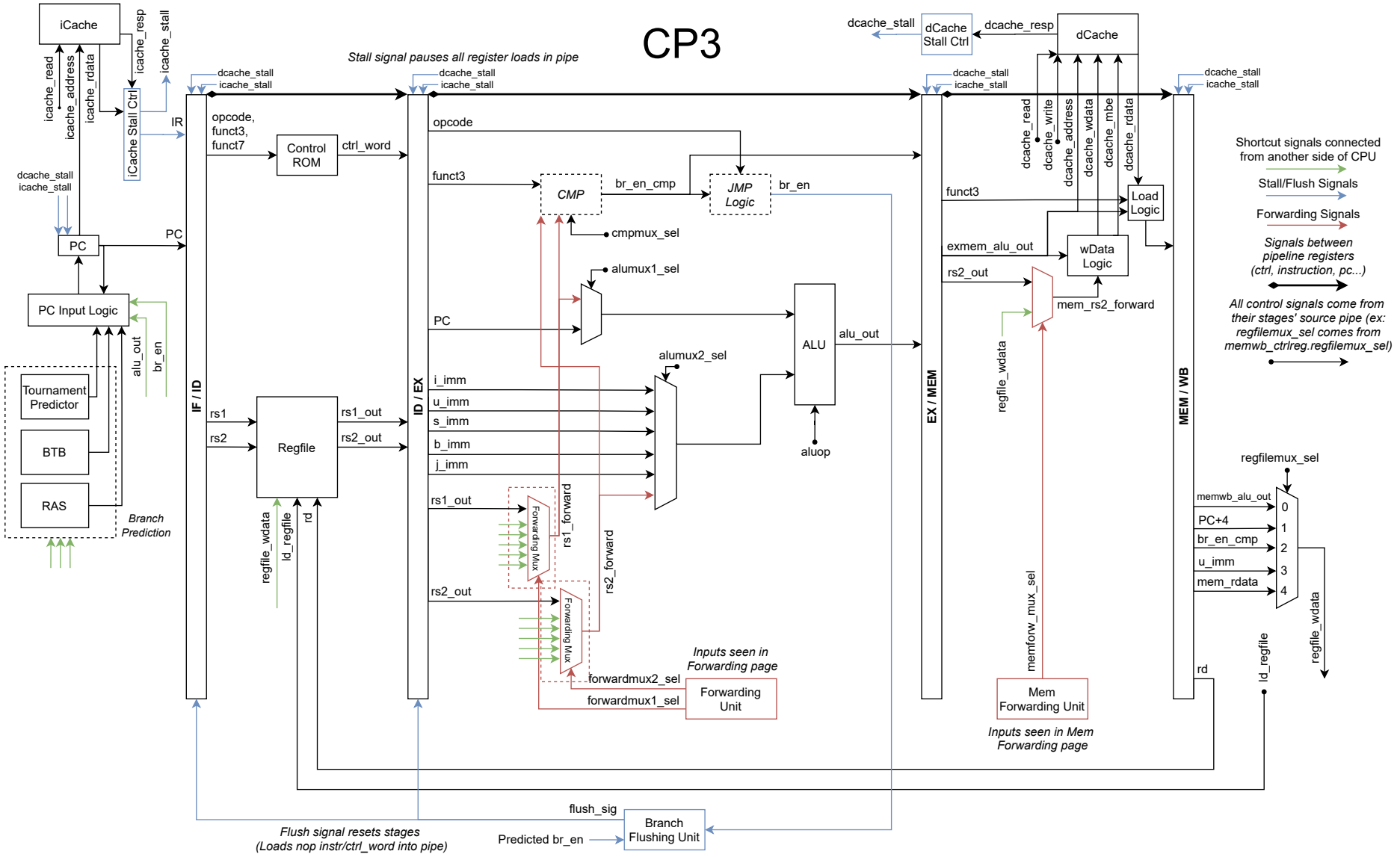
CP4 Memory Hierarchy

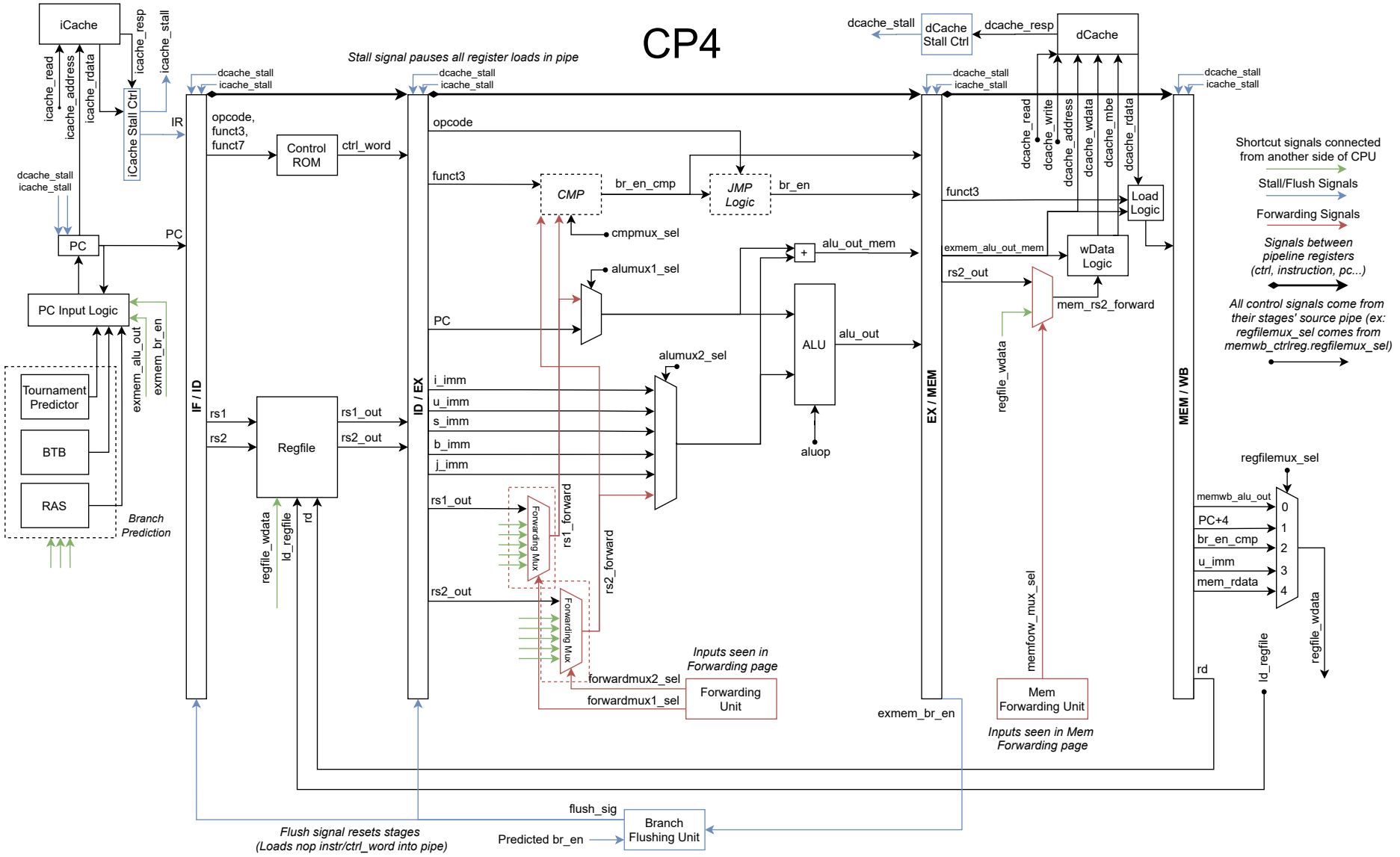


The 1-cycle delay buffers helped break up a long critical path and increased FMax


CP1
(Caches have not been connected yet, using dual-port magic memory)
iCache
dCache
mem_address
mem_rdata
IR
Signals between pipeline registers
opcode, funct3, funct7
Control ROM
ctrl_word
alumux1_sel
rs1_out
PC
0
1
ALU
alu_out
alumux2_sel
i_imm
u_imm
s_imm
b_imm
j_imm
rs2_out
0
1
2
3
4
5
mem_address
mem_wdata
mem_rdata
alu_out
wData Logic
rs2_out
PC
+4
0
1
br_en
alu_out
IF / ID
rs1
rs2
Regfile
rs1_out
rs2_out
ID / EX
EX / MEM
MEM / WB
opcode
rs1_out, rs2_out, i_imm, funct3
CMP
br_en_cmp
JMP Logic
br_en
cmpmux_sel
regfile_wdata
rd
regfilemux_sel
alu_out
PC+4
br_en
u_imm
mem_rdata
0
1
2
3
4
regfile_wdata
alu_out
br_en
rd


CP2
iCache
icache_read
icache_address
icache_rdata
icache_resp
icache_stall
iCache Stall Ctrl
nop
IR
pc_ld =
~icache_stall
& ~dcache_stall
PC
+4
0
1
alu_out
br_en
IF / ID
opcode,
funct3,
funct7
Control
ROM
ctrl_word
rs1
rs2
Regfile
rs1_out
rs2_out
regfile_wdata
ld_regfile
rd
Stall signal pauses all register loads in pipe
dcache_stall
icache_stall
ID / EX
opcode
funct3
CMP
br_en_cmp
JMP
Logic
br_en
cmpmux_sel
alumux1_sel
PC
alumux2_sel
i_imm
u_imm
s_imm
b_imm
j_imm
ALU
alu_out
aluop
rs1_out
regfile_wdata
exmem_alu_out
dcache_rdata
rs1_forward
rs2_out
rs2_forward
Inputs seen in
Forwarding
Unit Logic page
rs2forw_mux_sel
rs1forw_mux_sel
Forwarding
Unit
flush_sig
Branch
Flushing Unit
Predicted br_en
Flush signal resets stages
(Loads nop instr/ctrl_word into pipe)
dcache_stall
dCache
Stall Ctrl
dcache_resp
dCache
dcache_read
dcache_write
dcache_address
dcache_wdata
dcache_mbe
dcache_rdata
EX / MEM
exmem_alu_out
wData
Logic
rs2_out
mem_rs2_forward
regfile_wdata
memforw_mux_sel
Mem
Forwarding Unit
Inputs seen
in Mem
Forwarding
Unit Logic page
MEM / WB
rd
ld_regfile
regfilemux_sel
memwb_alu_out
PC+4
br_en
u_imm
mem_rdata
0
1
2
3
4
regfile_wdata
Shortcut signals connected
from another side of CPU
Stall/Flush Signals
Forwarding Signals
Signals between
pipeline registers
(ctrl, instruction, pc...)
All control signals come from
their stages' source pipe (ex:
regfilemux_sel comes from
memwb_ctrlreg.regfilemux_sel)


CP3
Stall signal pauses all register loads in pipe
iCache
icache_read
icache_address
icache_rdata
icache_resp
icache_stall
iCache Stall Ctrl
dcache_stall
icache_stall
PC
PC Input Logic
alu_out
br_en
Tournament Predictor
BTB
RAS
Branch Prediction
IR
opcode, funct3, funct7
Control ROM
ctrl_word
IF / ID
rs1
rs2
Regfile
rs1_out
rs2_out
regfile_wdata
ld_regfile
rd
ID / EX
opcode
funct3
CMP
br_en_cmp
JMP Logic
br_en
cmpmux_sel
alumux1_sel
PC
alumux2_sel
i_imm
u_imm
s_imm
b_imm
j_imm
ALU
alu_out
aluop
rs1_out
Forwarding Mux
rs1_forward
rs2_forward
rs2_out
Inputs seen in Forwarding page
forwardmux2_sel
forwardmux1_sel
Forwarding Unit
flush_sig
Predicted br_en
Branch Flushing Unit
Flush signal resets stages (Loads nop instr/ctrl_word into pipe)
dcache_stall
dCache Stall Ctrl
dcache_resp
dCache
dcache_read
dcache_write
dcache_address
dcache_wdata
dcache_mbe
dcache_rdata
funct3
Load Logic
exmem_alu_out
wData Logic
rs2_out
mem_rs2_forward
regfile_wdata
memforw_mux_sel
Mem Forwarding Unit
Inputs seen in Mem Forwarding page
EX / MEM
MEM / WB
regfilemux_sel
memwb_alu_out
PC+4
br_en_cmp
u_imm
mem_rdata
0
1
2
3
4
rd
ld_regfile
regfile_wdata
Shortcut signals connected from another side of CPU
Stall/Flush Signals
Forwarding Signals
Signals between pipeline registers (ctrl, instruction, pc...)
All control signals come from their stages' source pipe (ex: regfilemux_sel comes from memwb_ctrlreg.regfilemux_sel)

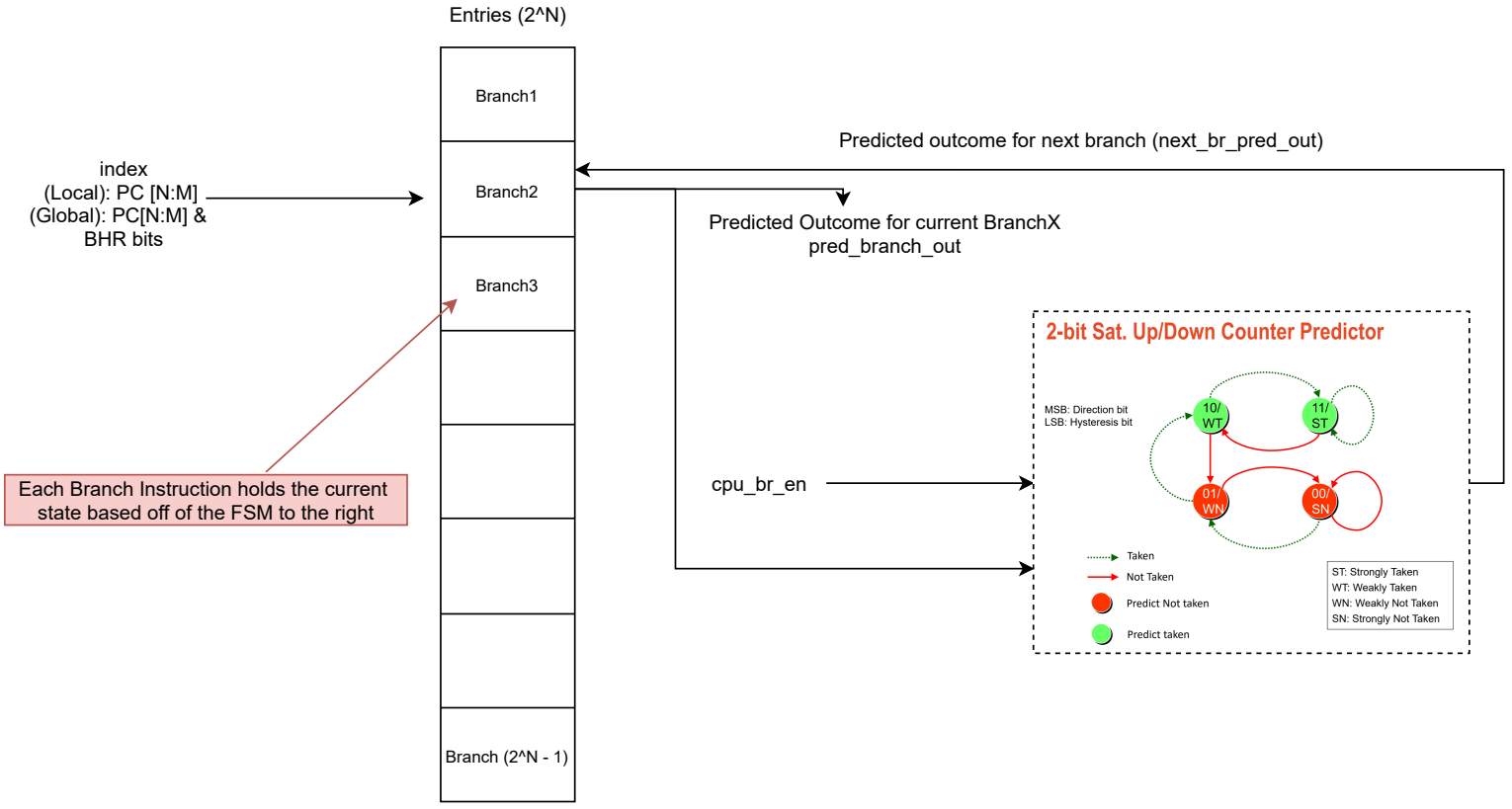

CP4
Stall signal pauses all register loads in pipe
iCache
icache_read
icache_address
icache_rdata
icache_resp
icache_stall
iCache Stall Ctrl
dcache_stall
icache_stall
IR
opcode,
funct3,
funct7
Control ROM
ctrl_word
PC
PC Input Logic
Tournament Predictor
BTB
RAS
Branch Prediction
exmem_alu_out
exmem_br_en
IF / ID
rs1
rs2
Regfile
rs1_out
rs2_out
regfile_wdata
ld_regfile
rd
ID / EX
opcode
funct3
CMP
br_en_cmp
JMP Logic
br_en
cmpmux_sel
alumux1_sel
alumux2_sel
i_imm
u_imm
s_imm
b_imm
j_imm
ALU
alu_out
aluop
alu_out_mem
Forwarding Mux
rs1_forward
rs2_forward
Inputs seen in Forwarding page
forwardmux2_sel
forwardmux1_sel
Forwarding Unit
dCache Stall Ctrl
dcache_resp
dCache
dcache_read
dcache_write
dcache_address
dcache_wdata
dcache_mbe
dcache_rdata
EX / MEM
exmem_alu_out_mem
funct3
Load Logic
wData Logic
rs2_out
mem_rs2_forward
regfile_wdata
memforw_mux_sel
Mem Forwarding Unit
Inputs seen in Mem Forwarding page
exmem_br_en
MEM / WB
rd
ld_regfile
regfilemux_sel
memwb_alu_out
PC+4
br_en_cmp
u_imm
mem_rdata
0
1
2
3
4
regfile_wdata
Shortcut signals connected from another side of CPU
Stall/Flush Signals
Forwarding Signals
Signals between pipeline registers (ctrl, instruction, pc...)
All control signals come from their stages' source pipe (ex: regfilemux_sel comes from memwb_ctrlreg.regfilemux_sel)
flush_sig
Predicted br_en
Branch Flushing Unit
Flush signal resets stages
(Loads nop instr/ctrl_word into pipe)

Advanced Features

High-Level Diagrams

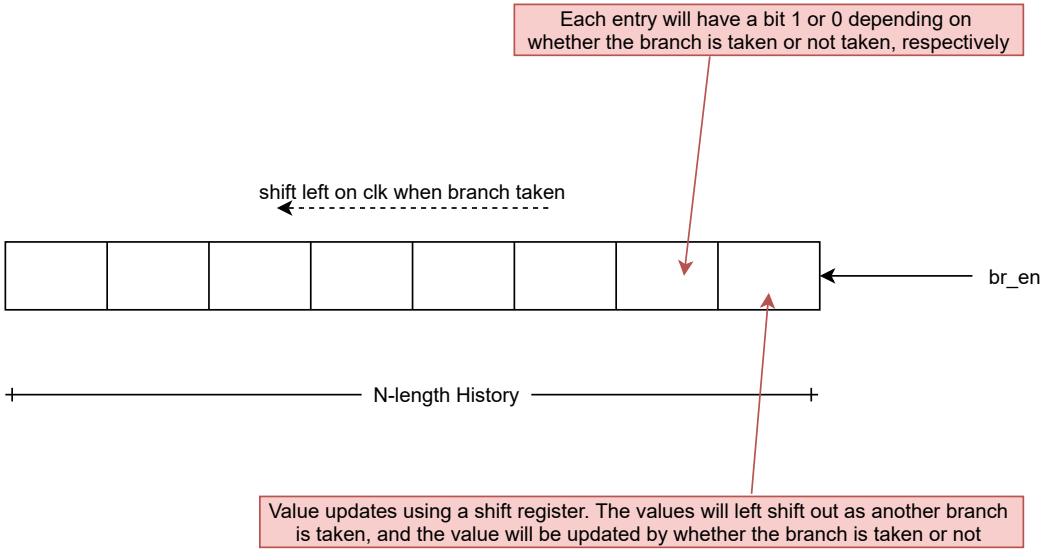
Appendix A (cont)

Pattern History Table (Local History Table & 2nd Level of Global Predictor)



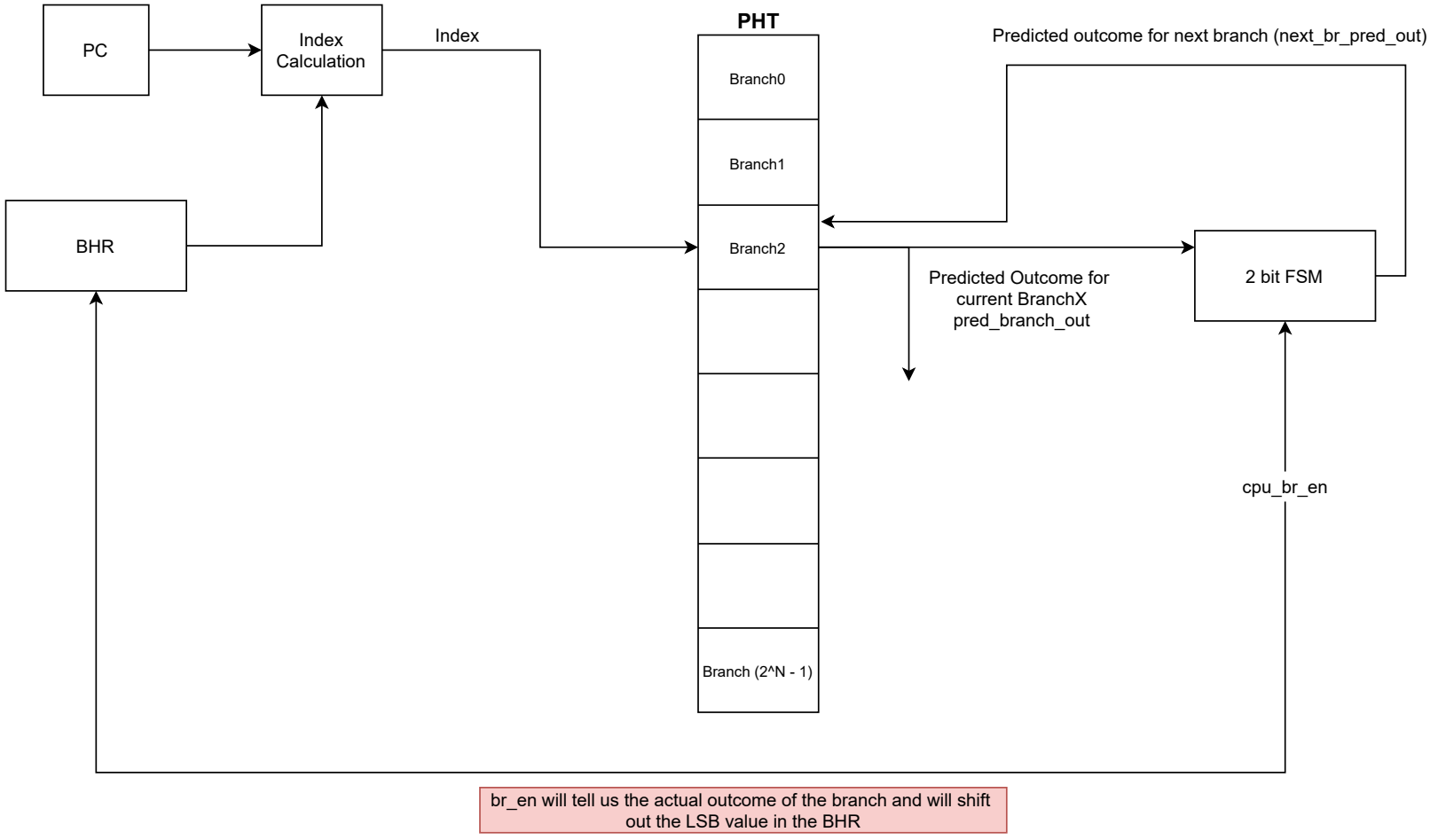
The FSM shown to the right decides what state the branch entry will be in. This is based off of whether a specific branch is taken or not taken. The state inside the table will be updated depending on the FSM.

Branch History Register

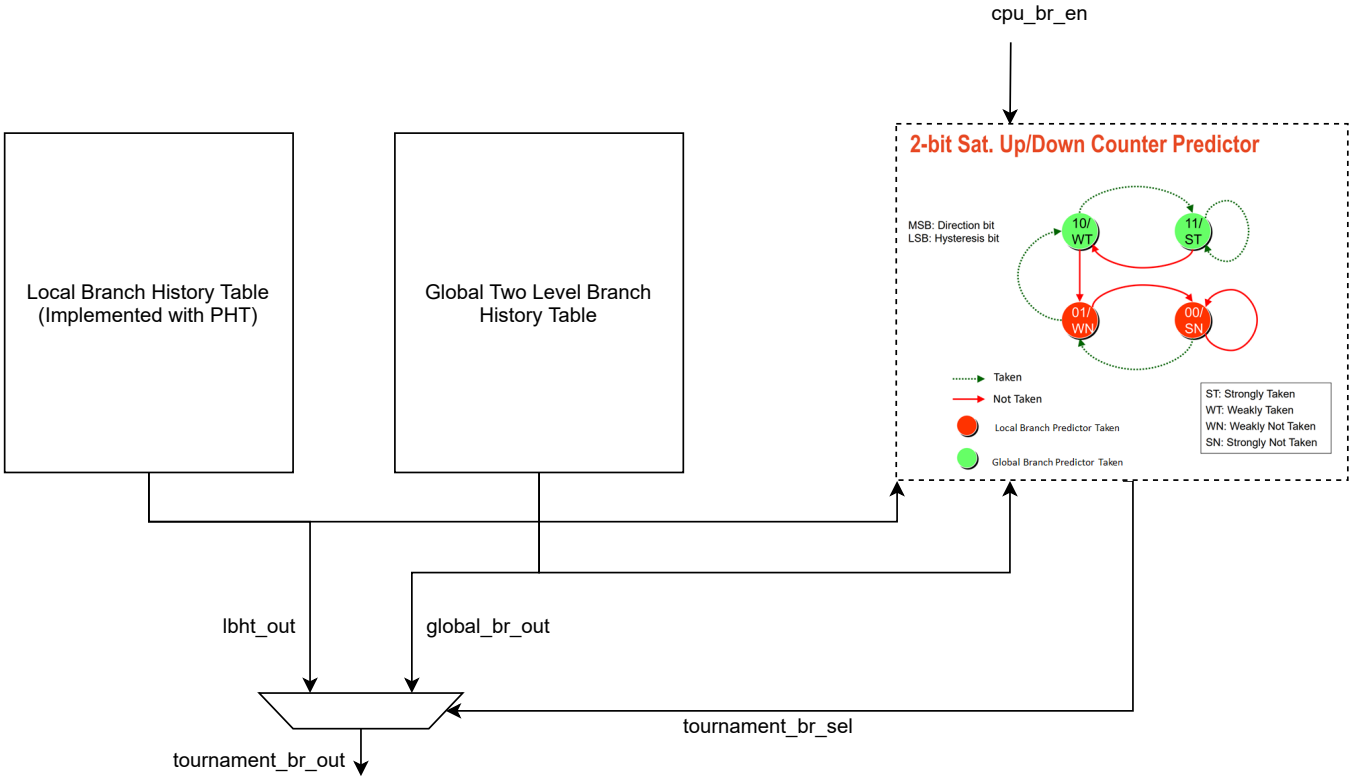


This will keep track of the last N branches, and the N-length history is used as part of the index for the PHT for our global predictor.

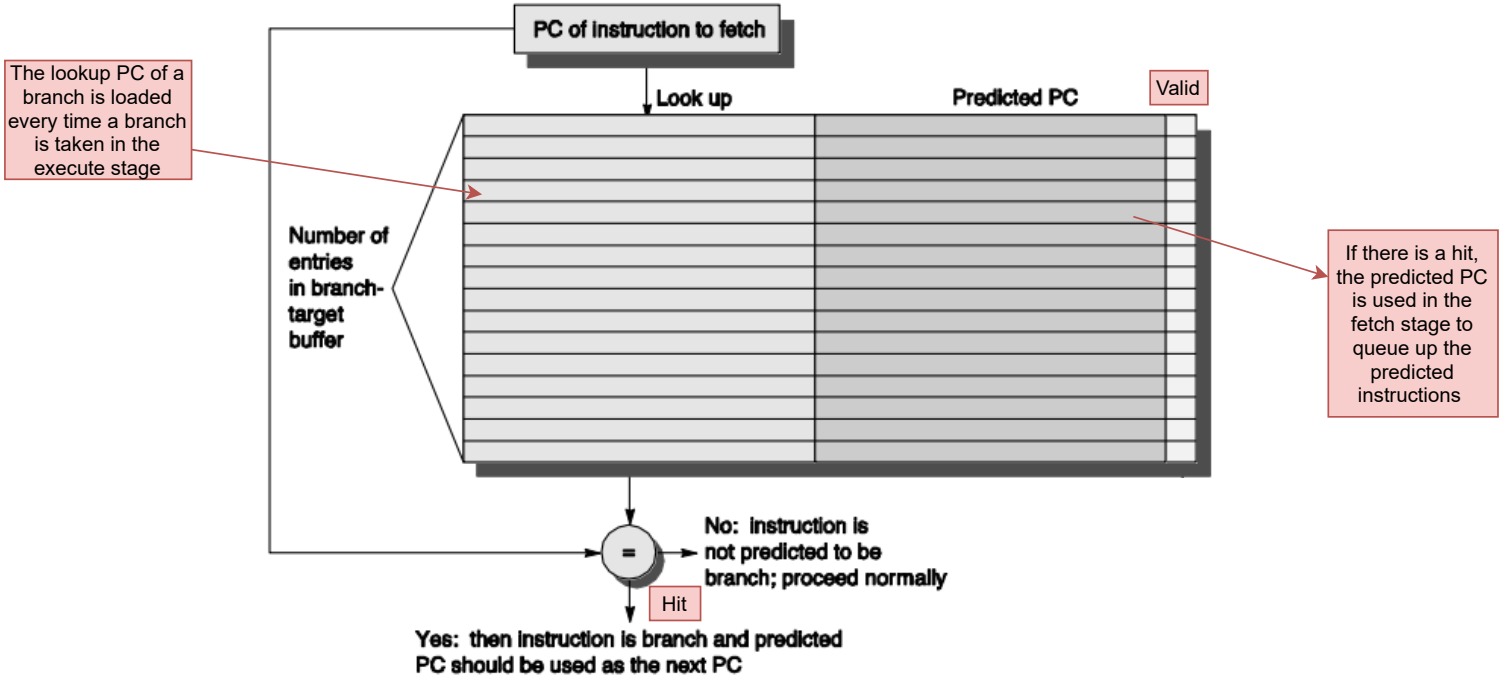
Global Predictor



Tournament Predictor



Branch Target Buffer



The function of the BTB is to supply target address for predicted branches. The BTB is essential because a branch prediction system is only as good as the BTB. If the BTB keeps missing and never knows what pc address a branch jumps to, the branch prediction went to waste.

The BTB can be indexed and read by some select bits of the PC. A tag can be used to check if it is a full hit.

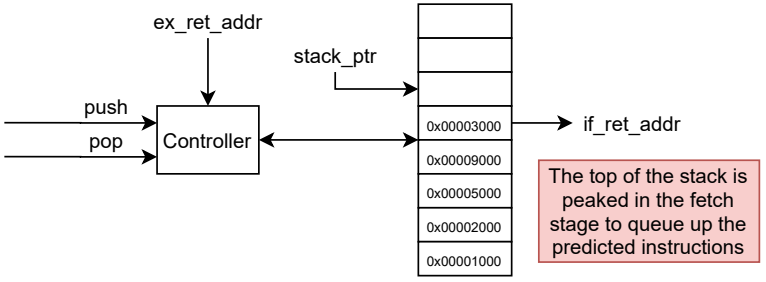
The BTB is used both by the execute and fetch stages. It's loaded in the execute stage when a branch (BR or JAL) is resolved. It is read in the fetch stage when a branch (BR or JAL) instruction just got inserted into the pipeline.

Return Address Stack

The RAS lives in the execute stage, pushing and popping the return address of JALR instructions according to these rules

rd	*rs1*	*rs1=rd*	RAS action
!*link*	!*link*	-	none
!*link*	*link*	-	pop
link	!*link*	-	push
link	*link*	0	push and pop
link	*link*	1	push

link is true when the register is either x1 or x5



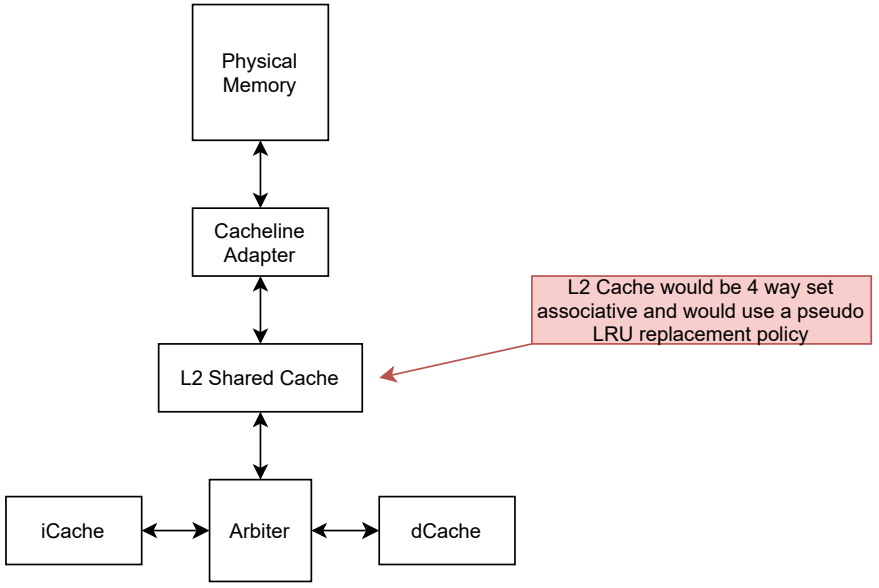
The RAS pushes/pops on JALR instructions being executed. It is read in the fetch stage when a JALR (RET) instruction just got inserted into the pipeline.

It's function is very similar to the BTB: to supply predicted target addresses for JALR (instead of BR and JAL) instructions.

Advanced Cache

L2 Cache, 4-way set associative cache, parameterized cache

**** 4-way L2 cache proposed but not implemented***

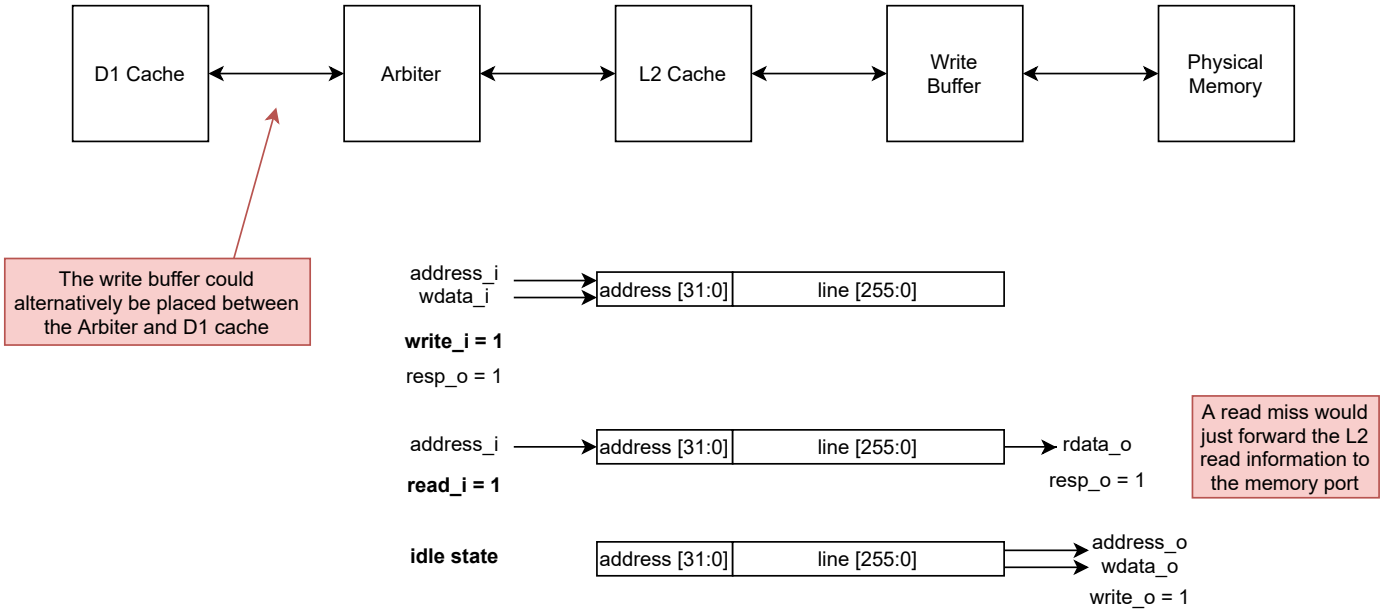


The 4-way cache would use a pseudo LRU to approximate replacement instead of a complete one because it is more efficient. The hardware needed to have a true replacement policy would need to keep an order of previous line accesses which has 4!=24 different scenarios. This would be expensive and not worth. A binary tree instead would only need 3 bits to approximate.

Since the L2 cache does not have to respond to hits in one cycle like the other caches do, it can respond to hits in 2 cycles like in mp3. We will evaluate this hit time further down the line, but for now it will be 1 cycle hit.

We also can parameterize the number of sets in our cache. This gives us an option in the future to tune our memory system by adding more sets. This makes the cache footprint bigger but it can allow more hits. The tradeoffs for this will be evaluated in more detail further down the road when we try to maximize performance.

Write Buffer

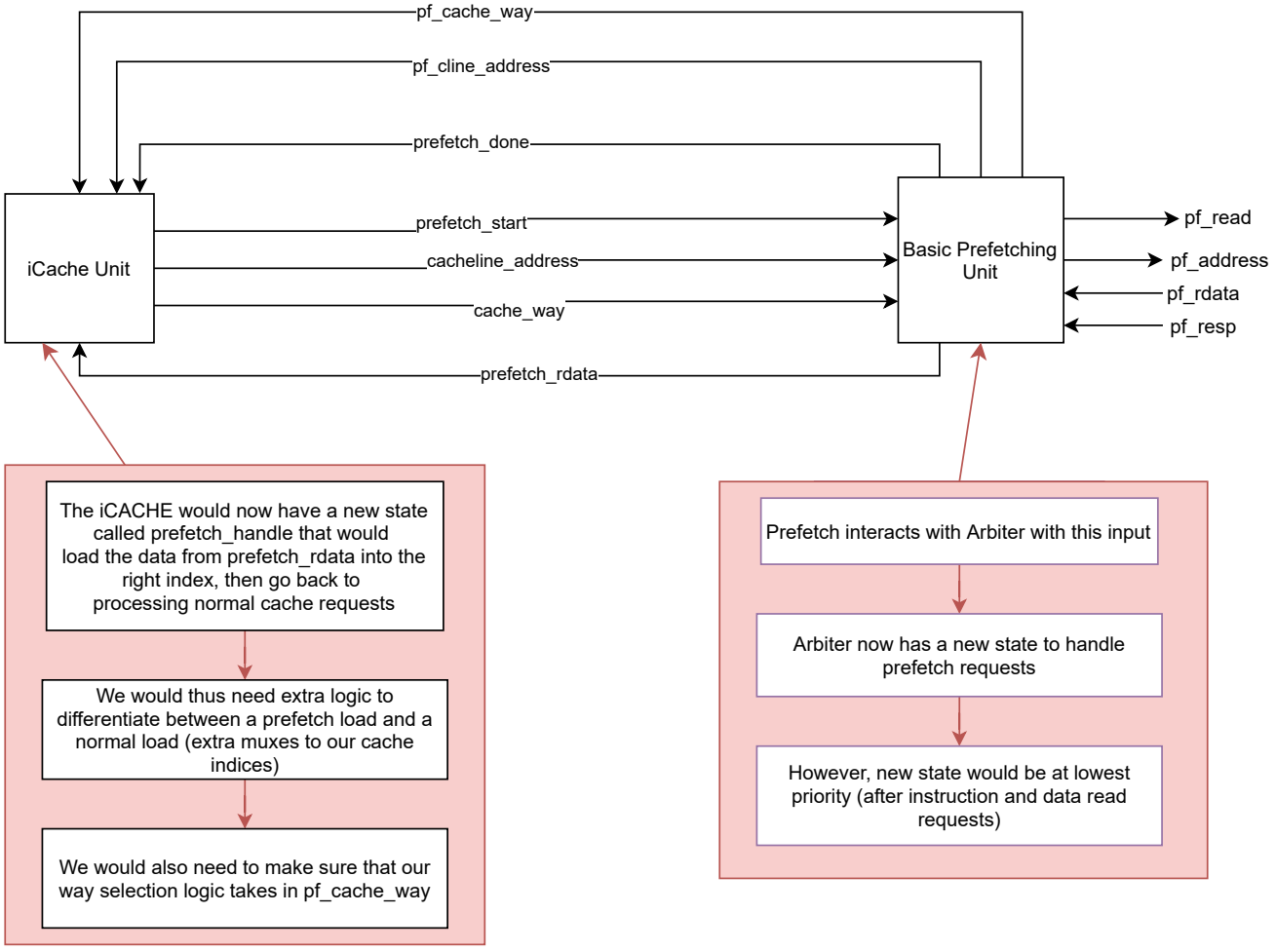


The write buffer has to sit in between levels of memory to intercept incoming writes. This lets lower level caches not waste as much time waiting for writebacks and can instead move on to getting read responses.

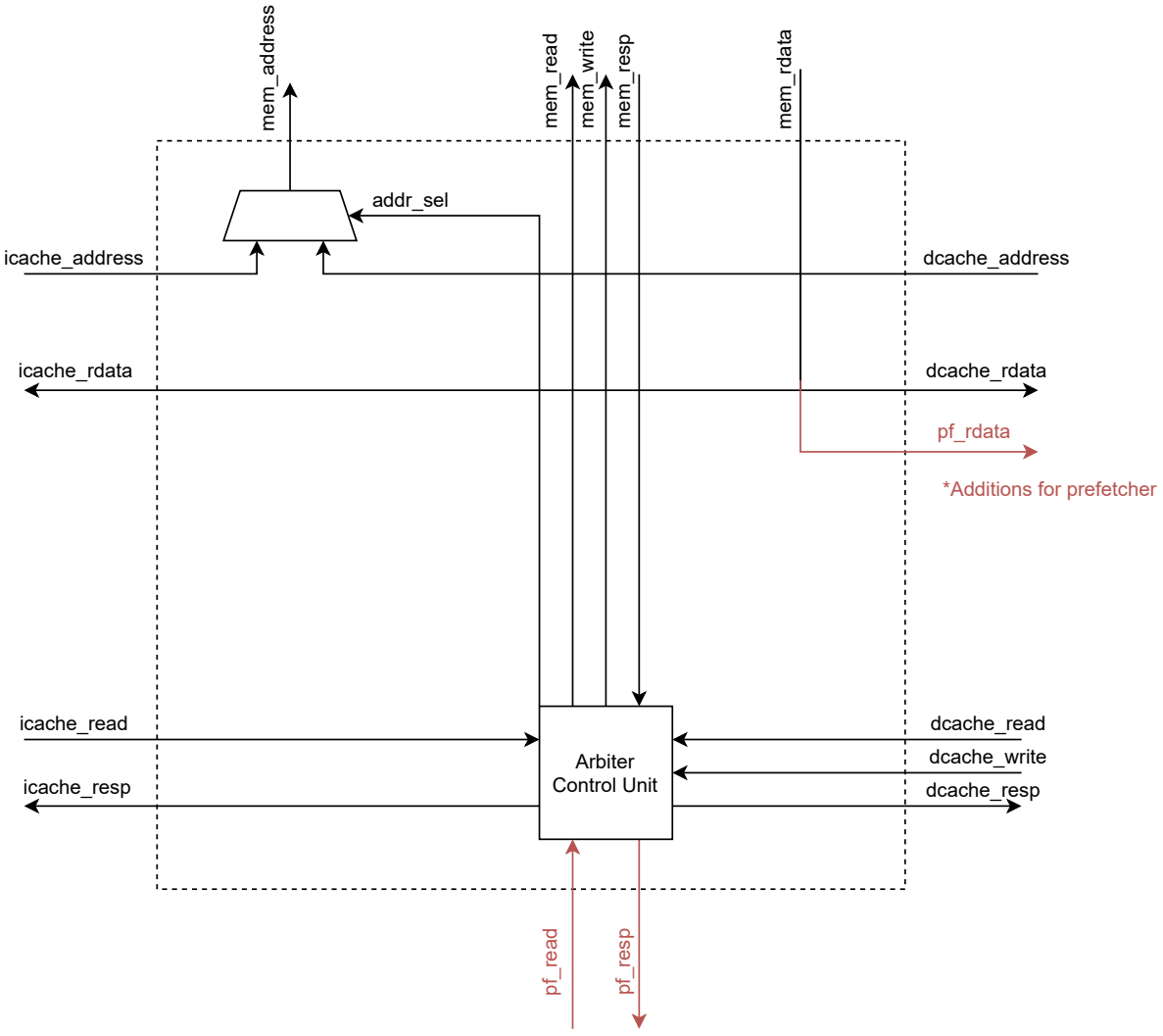
The buffer itself is similar to a cache in that it would need to store a line of information and do logic to check if there was a hit on it.

For incoming writes, if there is space, it stores the data and the address in its internal buffer. If there is no space, it is forced to write it back. For incoming reads if the address hits, the buffer is outputted because it has the most up to date copy. If the read misses, the read request is forwarded to the next level. When the buffer is in a idle state, it can choose a buffer to write back.

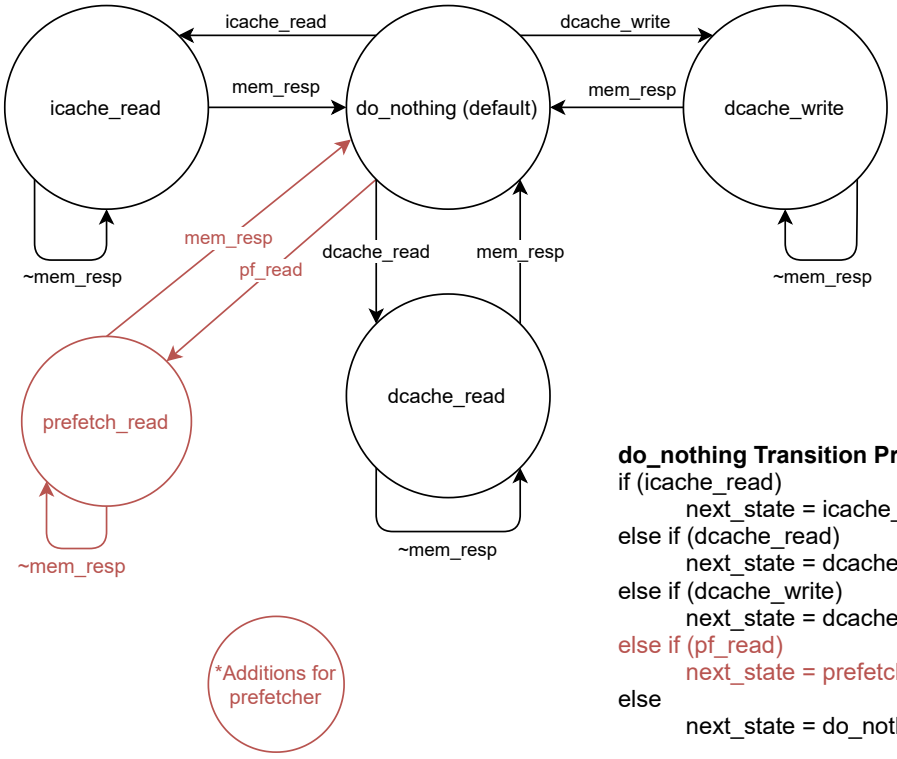
Prefetching Unit



Prefetch Arbiter



Arbiter Control Unit State Machine



do_nothing Transition Priority:

if (icache_read)
    next_state = icache_read
else if (dcache_read)
    next_state = dcache_read
else if (dcache_write)
    next_state = dcache_write
else if (pf_read)
    next_state = prefetch_read
else
    next_state = do_nothing

Output at each state:

do_nothing
- mem_read = 0
- mem_write = 0
- addr_sel = addr_mux::icache_addr
- icache_resp = 0
- dcache_resp = 0
- pf_resp = 0

icache_read
- mem_read = 1
- mem_write = 0
- addr_sel = addr_mux::icache_addr
- icache_resp = mem_resp
- dcache_resp = 0
- pf_resp = 0

dcache_read
- mem_read = 1
- mem_write = 0
- addr_sel = addr_mux::dcache_addr
- dcache_resp = mem_resp
- icache_resp = 0
- pf_resp = 0

dcache_write
- mem_read = 0
- mem_write = 1
- addr_sel = addr_mux::dcache_addr
- dcache_resp = mem_resp
- icache_resp = 0
- pf_resp = 0

prefetch_read
- mem_read = 1
- mem_write = 0
- addr_sel = addr_mux::pf_addr
- pf_resp = mem_resp
- dcache_resp = 0
- icache_resp = 0

Appendix B: I/O Tables for Advanced Features

Pattern History Table (Local Branch Predictor)		
Inputs	pht_rindex	Read index into table for fetch stage
	pht_windex	Write index into table for execute stage
	pht_ld	When to load the table
	cpu_br_en	The pipelines branch signal in execute
	read_opcode	Opcode of instruction in fetch stage
Outputs	predicted_br	Prediction for instruction in fetch stage

Global Branch Predictor		
Inputs	read_pc	PC of instruction in fetch stage
	write_pc	PC of instruction in execute stage
	glob_pred_ld	When to load the predictor
	cpu_br_en	The pipelines branch signal in execute
	read_opcode	Opcode of instruction in fetch stage
Outputs	predicted_br	Prediction for instruction in fetch stage

Tournament Branch Predictor		
Inputs	stall	Pipelines stall signal
	read_pc	PC of instruction in fetch stage
	write_pc	PC of instruction in execute stage
	pred_ld	When to load the predictor
	cpu_br_en	The pipelines branch signal in execute
	read_opcode	Opcode of instruction in fetch stage
Outputs	pred_br	Prediction for instruction in fetch stage

Branch Target Buffer		
Inputs	btb_load	When to load btb
	br_en	The pipelines branch signal in execute
	pc_address_if	PC address of instruction being fetched
	pc_address_ex	PC address of branch that was taken in execute
	br_address	Target address of branch calculated in execute
Outputs	hit	If branch instruction in fetch stage has a entry hit
	predicted_pc	Target address of branch instruction in fetch stage

Return Address Stack		
Inputs	stall	Pipelines stall signal
	ex_instr	Instruction in execute stage
	ec_pcp4	Target address input for loading into stack
Outputs	target_addr_out	Predicted target address of the jump instruction
	empty	If the stack is full

L2 Cache		
Inputs	mem_read	Lower level port
	mem_write	
	mem_wdata256	
	mem_address	
Outputs	mem_rdata256	
	mem_resp	
Inputs (cont)	pmem_rdata	Higher level port
	pmem_resp	
Outputs (cont)	pmem_read	
	pmem_write	
	pmem_wdata	
	pmem_address	

Write Buffer		
Inputs	ewb_read_i	Lower level port
	ewb_write_i	
	ewb_wdata_i	
	ewb_address_i	
Outputs	ewb_rdata_o	
	ewb_resp_o	
Inputs (cont)	ewb_rdata_i	Higher level port
	ewb_resp_i	
Outputs (cont)	ewb_read_o	
	ewb_write_o	
	ewb_wdata_o	
	ewb_address_o	

Prefetcher		
Inputs	prefetch_start	Input signal to tell the prefetcher to start a new request
	cacheline_address	The cache address that was recently missed by the cache
	cache_way	The cache-way to insert the prefetch data into. Cache determines this
	pf_rdata	Fetched data from memory(arbiter)
	pf_resp	Response signal from memory(arbiter)
Outputs	prefetch_rdata	The most recently fetched data from memory. Signal to Cache
	prefetch_ready	Signal indicating to the cache to load the prefetched data
	pf_cline_address	The address of the prefetched data (from which tag bits are determined)
	pf_cache_way	The cache-way to insert the prefetch data into
	pf_read	Read signal to memory (through the arbiter)
	pf_address	Address for the memory read request (sent to arbiter)

Prefetcher L1 Cache		
Inputs	mem_read	Lower level port
	mem_write	
	mem_wdata_cpu	
	mem_byte_enable_cpu	
	mem_address	
Outputs	mem_rdata_cpu	
	mem_resp	
Inputs (cont)	pmem_rdata	Higher level port
	pmem_resp	
Outputs (cont)	pmem_read	
	pmem_write	
	pmem_wdata	
	pmem_address	
Inputs (cont)	prefetch_rdata	The most recently fetched data from memory by the prefetcher
	prefetch_ready	Signal indicating to the cache to load the prefetched data
	pf_cline_address	The address of the prefetched data (from which tag bits are determined)
	pf_cache_way	The way to insert the prefetch data into
Outputs (cont)	prefetch_start	Signal indicate that the prefetching unit needs to initiate a new request
	cacheline_address	Address of the request that was most recently missed
	cache_way	The way in which to insert the prefetched data. Usually the LRU way