Computer Organization and Design
MP3 Final Report

Nathan Beauchamp, Rishi Thakkar, Melissa Jin
beaucha3, rrthakk2, mqjin2
Group Name: Silver Pipelining
ECE 411 Spring 2017

I: Introduction.

The most basic type of computer processor is one that simply fetches instructions from memory and executes them sequentially, fully completing all operations for a given instruction before beginning execution of the next instruction. While intuitive in concept, this type of processor is very limited in the amount of performance it can achieve. This is because each instruction requires multiple cycles to execute but doesn't use all of the processor's hardware in every cycle, resulting in underutilization of the processor. In order to solve this problem, we subdivide the processor architecture into stages that can each accommodate one instruction at a time, enabling the CPU to process multiple instructions simultaneously. This approach is referred to as "pipelining," and is pervasive in processors today. In Reduced Instruction Set Computing (RISC) architectures such as LC-3b, the pipeline architecture is classically divided into five stages, with registers in-between to buffer relevant control and data signals. These five stages are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Writeback (WB). The main technical thrust for this project was to implement a pipelined LC3-b processor along with a three-level memory hierarchy consisting of a split L1 cache, L2 cache, and physical memory. Secondarily, we aimed to implement advanced processor design features such as a victim cache, hardware prefetching, and one-level dynamic branch prediction. Like the motivation for implementing pipelining itself, the motivation for implementing advanced features is to further increase the performance of our processor and to gain exposure to historically successful processor design principles. As will be shown later in this report, these advanced features all served to increase the performance of our processor to some degree, depending on the test code used as a benchmark. Overall, completing this design was a very satisfying task that enabled us to learn the fundamentals of performance-centric computer architecture. Although our design is very simple compared to those of real processors used today, it still relies on many of the same fundamental principles; learning these are fundamental to understanding how computers work.

II: Project Overview.

In order to undertake a massive project such as implementing a pipelined LC-3b processor, we had to follow a very methodical design process. Communication between group members was essential, especially with regard to the integration of features that we had worked on separately. As such, our first step was to decide our goals for the project. As mandated by the project requirements, our primary goal was to implement a pipelined LC-3b processor that can correctly execute any program regardless of dependencies. In order to do this, we sought to implement solutions to the structural, data, and control hazards inherent to pipelined execution in a manner that minimizes the number of lost cycles. Furthermore, outside the scope of the specifically required features, we had many options to choose from for advanced design features with the goal of improving the functionality and/or performance of our processor. When we first began the project, we didn't yet know enough about the design and about the tradeoffs involved to commit to working on particular advanced features. However, we began to develop a better conception of this as we continued working on the project and did more research. After much deliberation and consideration of the performance deficiencies of our base processor design as indicated by test code, we decided to implement the advanced features of expanded L2 cache, victim cache, hardware prefetching, and one-level dynamic branch prediction. Our overall goal of the project was to get as much performance as possible out of our processor, and this definitely motivated our choice of advanced features.

Throughout our work on the project, work was split relatively evenly between the members of the group. Group members sometimes diverged to work on separate topics that were both relevant for the checkpoint at hand; other times, we worked together on the same aspect of the design. It was especially helpful to combine our efforts for debugging, as each group member would have different- yet valuable- ideas regarding potential problems and fixes. For example, for Checkpoint 1, Rishi and Melissa implemented the pipelined datapath while Nathan implemented the control ROM. Then, all three group members worked together on feature integration, testing, and debugging. This is the pattern that was employed for almost all checkpoints and for the advanced features work. Since group members were sometimes working on different tasks during the implementation stage, project management could get complicated. Thankfully, the fact that we reconvened for integration mitigated potential ill effects of this; if only one person had done integration, it would have been much more challenging as that person might not know all the intricacies of the others' code. We also heavily utilized git branches when working on features that were not quite working yet. This style of project management enabled us to successfully complete the project, integrating all of our performance-boosting advanced features into our master branch.

III: Milestones.

For Checkpoint 1, our group implemented a basic pipelined processor supporting six instructions in the LC-3b ISA (ADD, AND, NOT, LDR, STR, and BR). To do this, our first step was to modify our initial datapath paper design according to feedback from our mentor TA. Next, we implemented this datapath design in SystemVerilog, creating the appropriate registers and combinational logic for each pipeline stage. Third, we designed the control ROM by tracing the execution of instructions through the datapath and determining what the control signal values needed to be at each stage. In SystemVerilog, our control ROM module was implemented using a simple case statement on the input opcode. Finally, we created a top-level module to interface with our testbench and connect our control ROM and datapath instances. It is important to note that, at this time, our processor was simply connected to a dual-port magic memory. This is because we hadn't yet implemented a memory hierarchy or any kind of structural hazard detection. In order to verify that our design could successfully execute the given six LC-3b instructions, we wrote code to test the correctness of each instruction (or pairs of related instructions) individually. For example, we wrote code to verify that we could load data from memory into the general-purpose registers, and then store it back in a different order. The fact that our processor passed these individual test cases gave us confidence that it would correctly run the official CP1 test code. The datapath for our CP1 design is shown below in Figure 1 (zooming in is required to see everything):

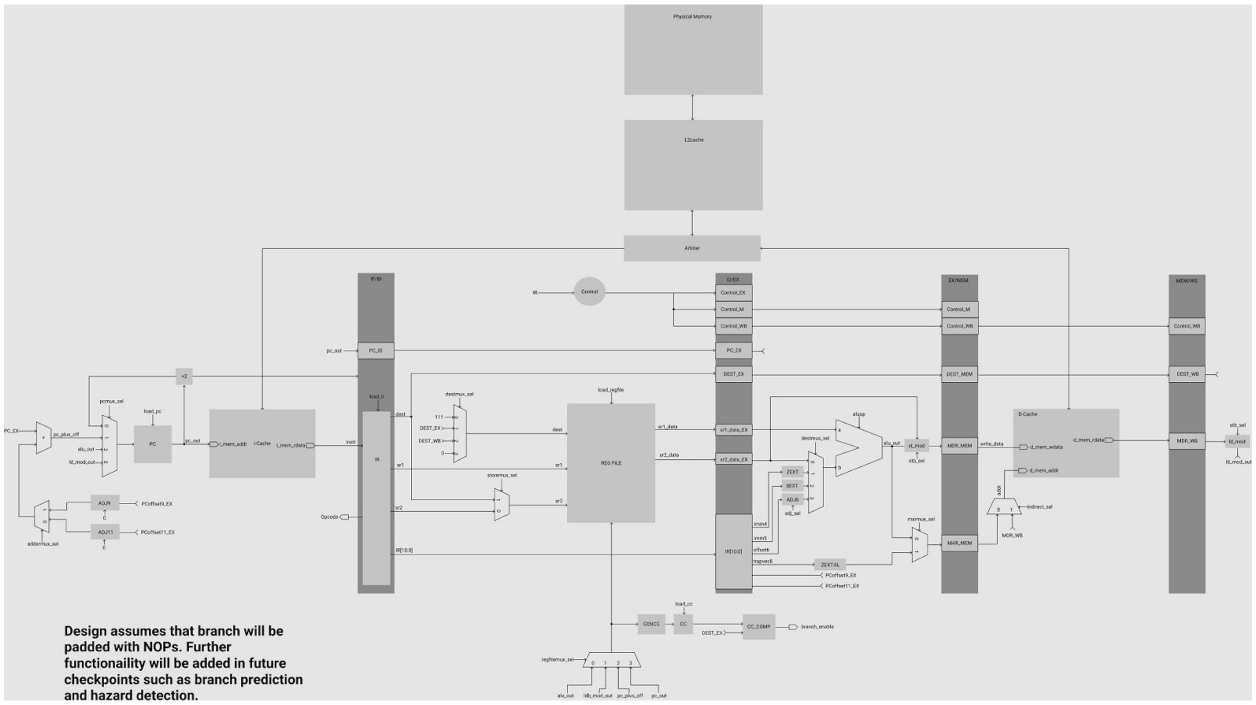
Physical Memory
L2cache
Arbiter
Control
IF/ID
ID/EX
EX/MEM
MEM/WB
PC
I-Cache
REG FILE
D-Cache
GENCC
CC
CC_COMP
branch_enable
Design assumes that branch will be padded with NOPs. Further functionaility will be added in future checkpoints such as branch prediction and hazard detection.

Figure 1: Our initial pipelined datapath design

As stated earlier, although this datapath depicts the planned three-level memory hierarchy, we had not yet implemented it as of CP1 .

Next, for Checkpoint 2, our group expanded on our basic pipelined processor design to support all of the instructions in the LC-3b ISA (with the exception of RTI). We also implemented the remaining elements of the memory hierarchy: a split L1 cache connected to an L2 cache, with accesses mediated by an arbiter. The L2 cache interfaces directly with physical memory. This required a solid understanding of pipelined processor design as we had to implement pipeline stalls to account for cache misses. Our first step in this process was to design the L1 cache, arbiter, and L2 cache. We decided to use the 2-way, 8-set cache we implemented in MP2 as our L1 cache- then, since the L2 cache should be larger than the L1, we decided to use an 8-way, 8-set writeback cache to implement that. This required a fair amount of extra design effort as we had to extend the pseudo-LRU replacement policy to 8 ways. This required us to store seven LRU bits per set, organized in a tree pattern with each subtree choosing between two subsets of the ways. Although we didn't know it at the time, the larger L2 cache would end up becoming one of our advanced design features. We used a state-machine-based arbiter design to mediate access between the split L1 cache and L2 cache. Next, we modified our datapath paper design to account for the remaining LC3-b instructions, adding MUXes and control signals when necessary. We utilized a modular testing strategy, ensuring that our pipeline design worked for all instructions and for the CP2 test code with the dual-port magic memory before attempting to add the split L1 cache. To test the new instructions added in this checkpoint (e.g. STI/LDI), we wrote individual test programs as before. Essentially, we added memory hierarchy components one at a time, debugging the design on each step. This assisted us in debugging as it enabled us to know for sure as to which component had the bug. Throughout the process, we used code with unpadded STIs/LDIs in order to stress-test our structural hazard detection mechanisms. Shown below are images of our cache arbiter (Figure 2), and our L2 cache datapath (Figure 3) (zooming in may be required to see everything):

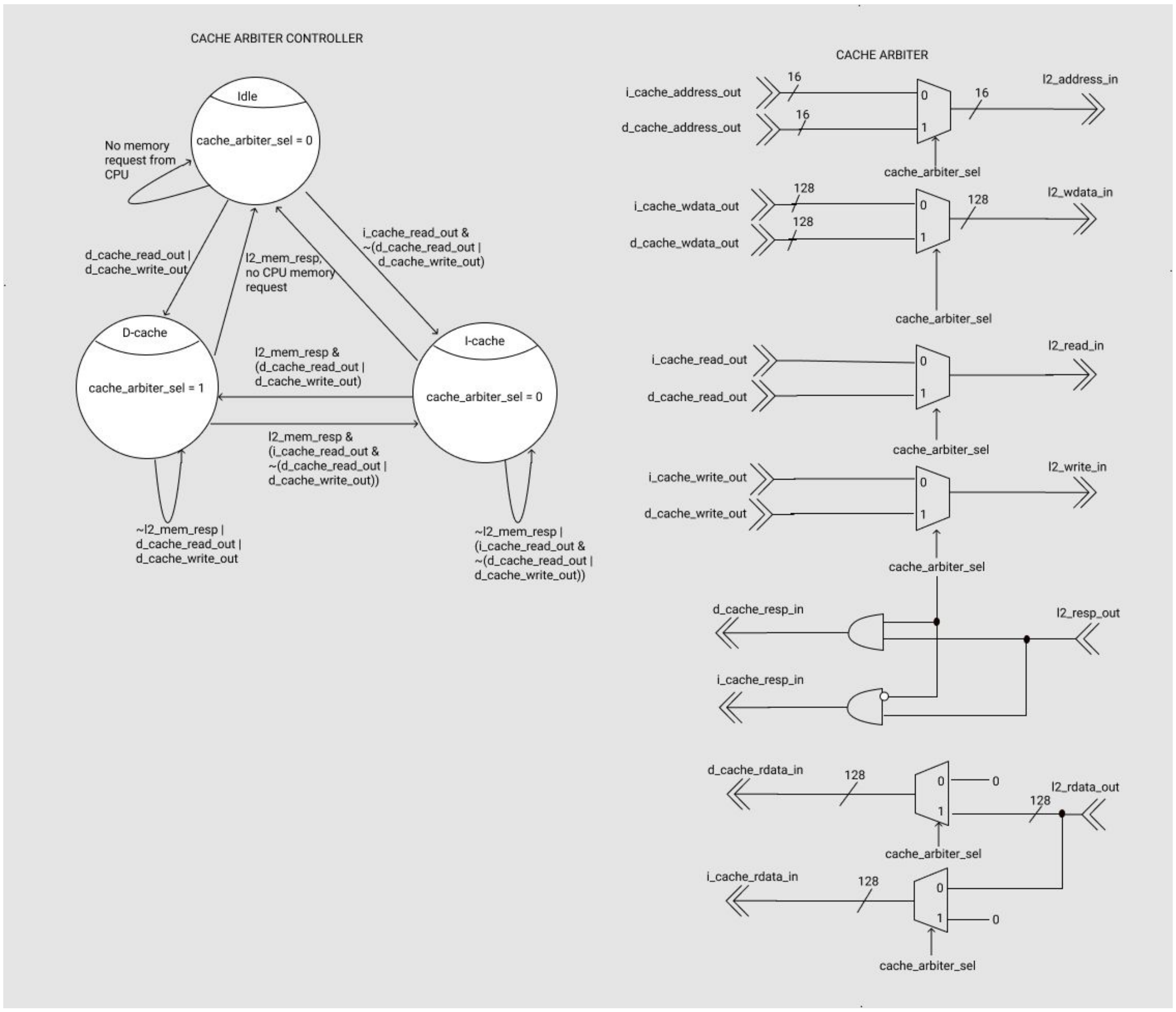

Figure 2: Cache Arbiter, including datapath and controller.

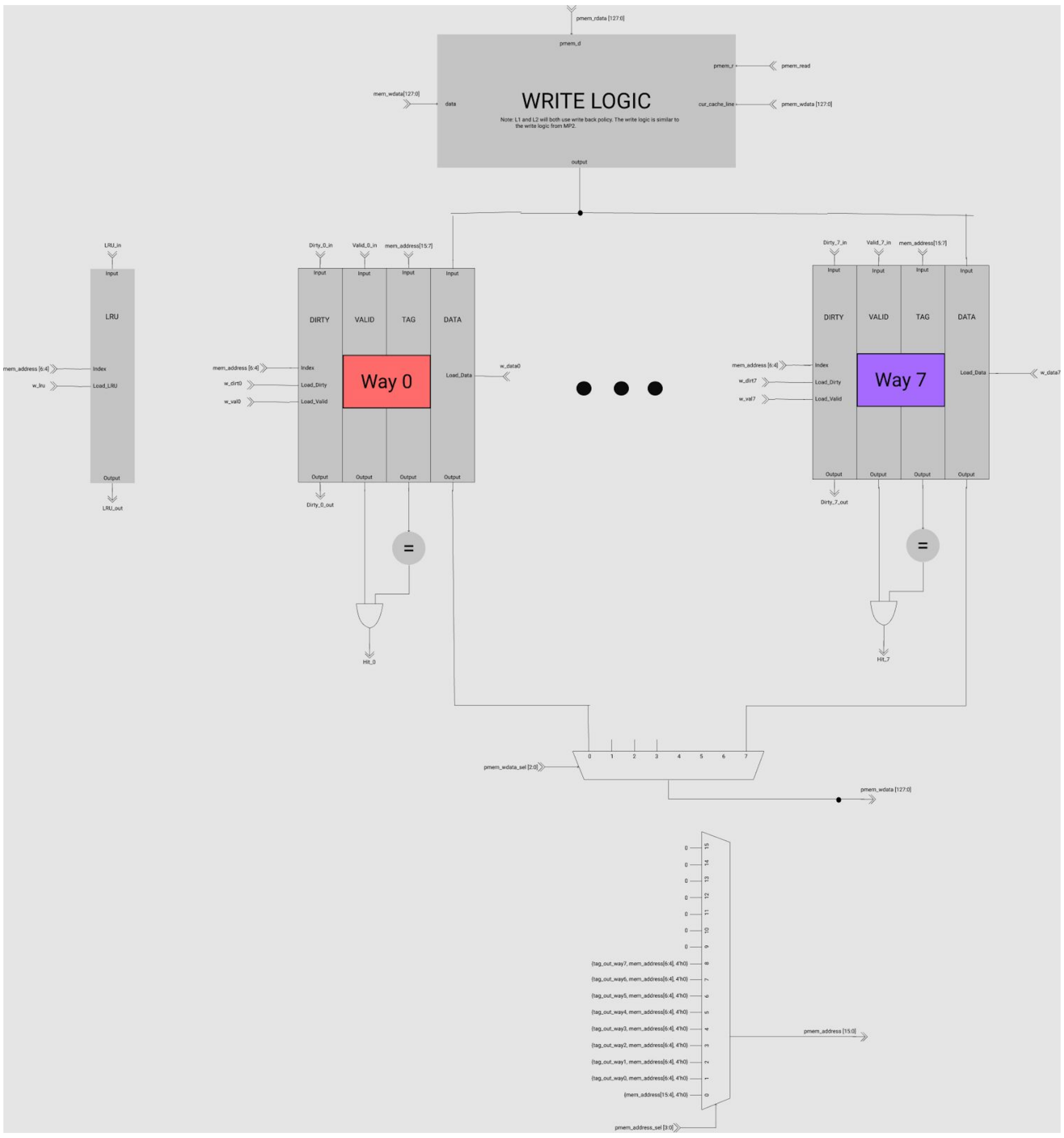

Figure 3: L2 Cache datapath

The L2 cache has eight ways; in the figure above, I used ellipses to indicate them in order to save space. The L2 cache datapath is very similar to the L1 cache datapath, but with more ways and different write logic. Because the L2 cache takes an entire cache line as input from the arbiter, the write logic no longer has to deal with with “splicing” input data with the current cache state. The pseudo-LRU update policy is handled by the L2 cache controller, which will update the LRU bits for a particular set whenever that set is accessed.

Third, for Checkpoint 3, our group expanded upon the work done in Checkpoint 2 to create a pipelined processor design that can handle all LC3-b programs, including those with data dependencies and control hazards. Since we had already designed and integrated an 8-way, 8-set L2 cache in Checkpoint 2, our work for this checkpoint amounted to handling data and control hazards only. To solve the issue of data hazards, we implemented data forwarding, creating the EX->EX and MEM->EX forwarding paths. This is a simple yet effective strategy that enables us to avoid stalling the pipeline in the case of data dependencies, since the result of a previous instruction can be forwarded to an earlier pipeline stage before it is written back to the register file. To solve the issue of control hazards, we decided to stall the pipeline upon detection of an instruction that modifies the value of the PC (BR, JMP, JSR, JSRR, TRAP) until its address calculation has been determined. Once this has occurred, we resume normal pipeline operation. This functionality is enforced by our hazard detection unit, the same unit that deals with structural hazards. In order to test our design, we wrote sample programs that included unpadded branches and arithmetic and memory instructions with many dependencies. We then ran these programs on our processor in ModelSim, and verified that the final register values produced were the same as those given by the LC3-b simulator. We also ran “sanity checks” on our cache memory lists to verify that they correspond to the values being read/written. This gave us confidence that our design would successfully run the CP3 test code. In order to implement forwarding, we essentially added combinational paths between the relevant pipe stages, with

MUXes selecting the inputs appropriately. A forwarding unit detects data dependencies and determines the MUXes' select signals as a result. Our pipelined datapath with forwarding accounted for is shown in Figure 4 below (once again, zooming in is required):

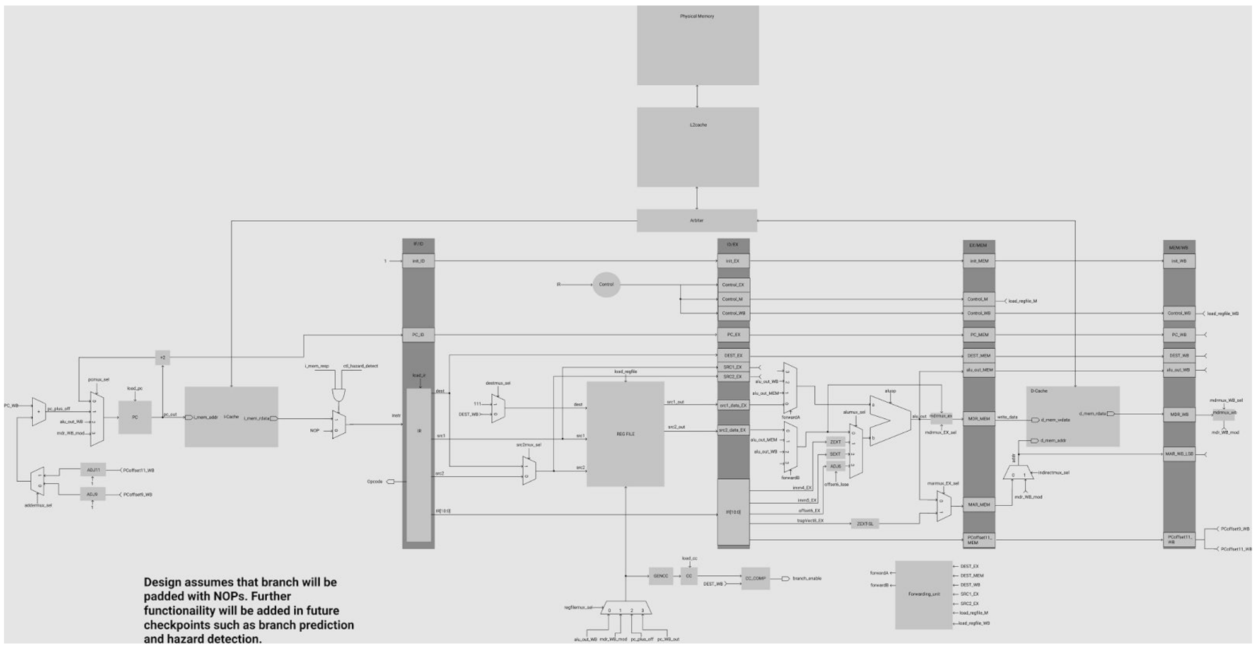
Physical Memory
L2cache
Arbiter
IF/ID
ID/EX
EX/MEM
MEM/WB
PC
I-Cache
Control
REG FILE
D-Cache
Forwarding_unit
Design assumes that branch will be padded with NOPs. Further functionaility will be added in future checkpoints such as branch prediction and hazard detection.

Figure 4: Pipelined datapath that accounts for data, control, and structural hazards.

Finally, for Checkpoint 4, our group implemented three core features: an eviction write buffer, static branch prediction, and performance counters. First, the eviction write buffer is an optimization meant to improve the performance of our write-back caches. The eviction write buffer holds data being written back to the next memory hierarchy level (e.g. physical memory for the L2 cache). This enables the cache to respond to read requests while the write is taking place. This is very useful as the cache can potentially service multiple read hits while writing back a dirty line. Next, for branch prediction, we decided to implement a static not-taken approach for Checkpoint 4. Essentially, when the processor encounters a branch instruction, the following instructions are loaded into the pipeline as normal (i.e. we're predicting not taken). Whether or not the branch is actually taken is determined in the writeback stage- if the branch is indeed not taken, then pipeline operation continues. Else, all instructions currently in the pipeline are flushed, and the next instruction is loaded from the calculated PC address. Unfortunately, the pipeline still has to stall for one cycle to account for the case of an STR instruction appearing immediately after a BR. Nonetheless, a static not-taken approach to branch prediction is still more efficient than having no branch prediction and stalling the pipeline whenever a branch is encountered. Finally, we implemented performance counters to track the following metrics: number of cache hits and misses (for all three caches), number of correct and incorrect branch predictions, and number of pipeline stall cycles. We built a memory-mapped I/O structure so that the performance counters are accessible from software. A programmer can read the registers by reading from memory addresses FFE0-FFF0; they can clear the registers by writing to the same addresses. This is very useful from both a programmer's perspective and a hardware designer's perspective. From the perspective of a hardware designer, these metrics can be used to determine how to best improve the performance of the caches and the branch predictor in a future design iteration. From the perspective of a programmer, these metrics can be used to help tune software to a specific hardware platform. To test the static branch predictor, we wrote test code with both taken and not-taken branches and verified that the processor worked correctly in both cases. For taken branches, the branch predictor's prediction was incorrect; as a result, we flushed the pipeline once the branch reached the writeback stage. For not-taken branches, the branch predictor's prediction was correct, so the processor continued executing instructions as normal. To test the performance counters, we simply added the appropriate memory-mapped I/O calls (i.e. load/store instructions) at the end of our previous test programs. This enabled us to see that the counters were being updated with the appropriate values throughout the program's execution and clear when the programmer specifies it. We also viewed the counter registers themselves in ModelSim as we traced through the program's execution, verifying that they increment under the correct conditions. Finally, to test the eviction write buffer, we wrote a testbench to verify its correct operation for all input cases. This enabled us to deduce that it would correctly when inserted into the memory hierarchy. The eviction write buffer controller is shown below in Figure 5, and a datapath updated for CP4 is shown in Figure 6:

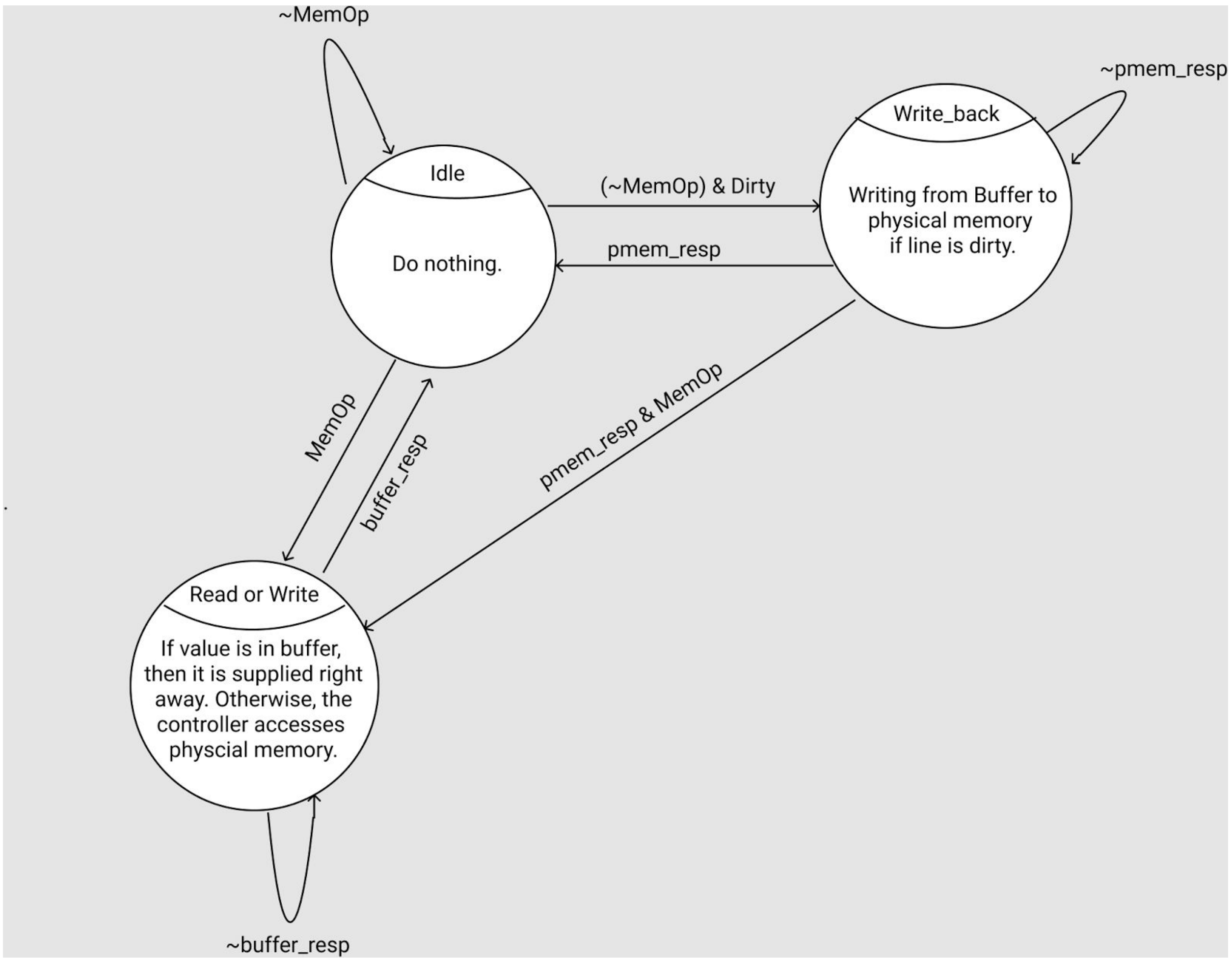
~MemOp
Idle
Do nothing.
(~MemOp) & Dirty
pmem_resp
Write_back
Writing from Buffer to physical memory if line is dirty.
~pmem_resp
MemOp
buffer_resp
pmem_resp & MemOp
Read or Write
If value is in buffer, then it is supplied right away. Otherwise, the controller accesses physcial memory.
~buffer_resp

Figure 5: Eviction write buffer controller

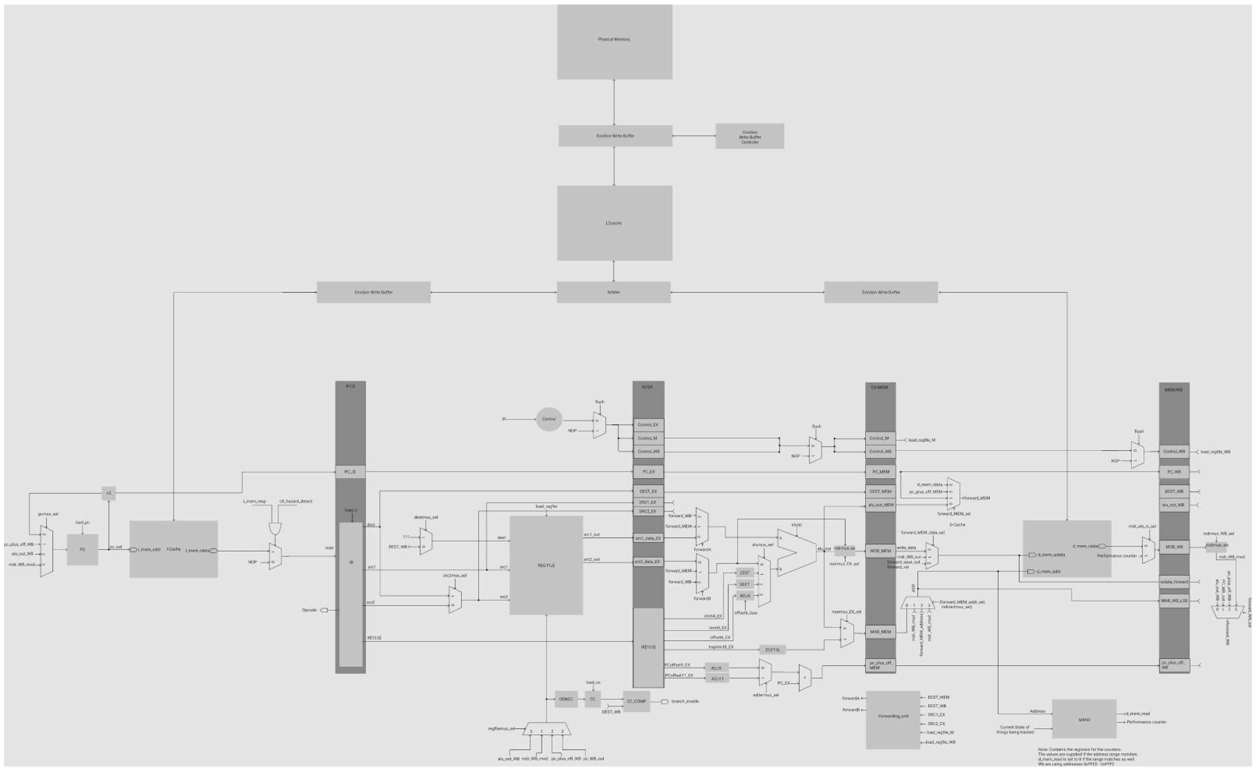

Figure 6: Pipelined datapath that accounts for performance counters and eviction write buffer.

Overall, our work throughout the checkpoints left us with a solid base implementation on top of which to implement advanced features. In the next section, we will describe the advanced design features we chose to implement as well as timing benchmarks showing how they improved the performance of our processor.

IV: Advanced Design Features.

4-way set-associative or higher L2 cache with LRU Policy (Pseudo or True) [5]

The first advanced design feature we decided to implement was an 8-way, 8-set, set-associative L2 cache with pseudo-LRU replacement policy. As described in the milestones section, this cache is very similar to the L1 cache, with differences in write logic and replacement policy. Like for the L1 cache, an entire cache line is always read from the L2 at a time; however, unlike the L1, an entire cache line is also written at a time when the L1 D-cache evicts a dirty line and writes back. Therefore, the write logic only has to select between the data from the L1 cache, the data from physical memory, and the currently stored data; no "splicing" is required. Furthermore, since we now have eight ways instead of two, we had to update our replacement policy to account for true tree-based pseudo LRU. The true LRU replacement policy, while being the most accurate, is known to be rather inefficient since it requires $k * log(k)$ bits to be stored per set, where k is the number of ways (in this case, 8). In contrast, pseudo-LRU only requires $k - 1$ bits to be stored per set. The pseudo-LRU algorithm is outlined as follows. We start with a binary tree of $k - 1$ nodes, each of which contains a single bit. Each bit is interpreted as a flag, with 0 meaning "the least recently used element is in the left subtree" and 1 meaning "the least recently used element is in the right subtree." Then, to select the way to replace, we start at the root and continue downwards, selecting the LRU subtree on each step. Then, the leaf node is used to select the exact way to replace. Updating the pseudo LRU is the most difficult part- we implemented this using a SystemVerilog case statement featuring different splicing logic for each case.

Because this L2 was implemented as our base version's L2 cache all the way back in Checkpoint 2, we were unable to benchmark how this larger L2 would compare to an L2 cache with only two ways. However, we did test our processor with a four-way, 16-set version of the L2 cache as well as the 8-way, 8-set version. Interestingly, there were no performance differences between the two; however, the maximum frequency was less for the four-way, 16-set version. Therefore, we decided to use the eight-way, eight-set version in our implementation as it allowed us to clock our design at a higher frequency.

Victim Cache [8]

Our victim cache is a fully associative cache which captures and stores any evicted value from the L1 or L2 caches. If these evictions were false evictions, then it supplies the values back to its respective cache. Otherwise, on dirty evictions, the victim cache writes back the value on the next available opportunity and on clean evictions, it simply holds the values. The victim cache is implemented with 8 cache lines and the same pseudo-LRU policy used with the L2 cache. We added a victim cache to the L2-cache, L1 I-cache, and L1 D-cache. The main purpose of this type of cache is to reduce the performance hit caused by a false eviction and decrease the amount of times a higher level of memory is accessed.

The victim cache improves performance by deferring memory writebacks to memory reads; however, if a read request is signaled in the middle of a memory write, the read must wait to be serviced. If a previously evicted value causes a read miss in a L[X] cache, where the X refers to the level of the cache, then the victim cache supplies its captured value instead of accessing the X+1 level of memory. Once the value is supplied, it is invalidated and the spot is made available for the next cache eviction. Therefore, the victim cache acts like an extension to the cache it is associated with and effectively increases the perceived associativity of that cache.

As seen in Figure 7 below, the three additional victim caches did not significantly improve performance; however, it improved a little across all four test codes. The reason for the small improvement can be associated with the fact that our L1 and L2 caches were already fairly big. We implemented an 2-way, 8-set L1 cache and an 8-way, 8-set L2 cache. Due to this, the amount of evictions seen, especially for small programs, are relatively low. On top of this, since the L2 is still a single cycle hit, placing the victim cache between the L1 and L2 did not really reduce the number of cycles required to access a previously evicted value. For larger programs that accessed a lot of memory, the victim cache between the L2 and physical memory resulted in the biggest performance increase because of the large number of cycles required to access physical memory.

Description of Feature	Competition_1 (ns)	Competition_2 (ns)	Competition_3 (ns)	Final (ns)
3 victim caches	920,455	315,785	6,358,425	686,365
no victim caches	927,635	319,155	6,843,435	690,805

Figure 7: Performance Analysis of Victim Cache

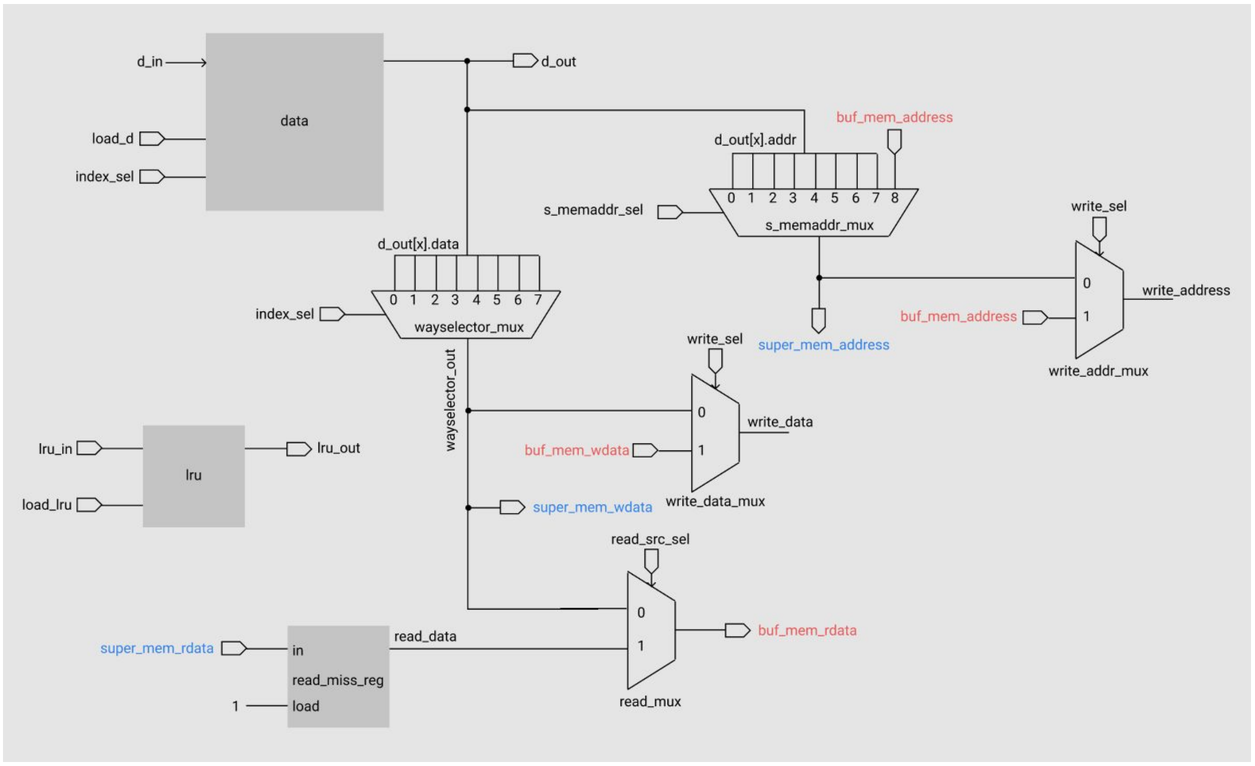

Figure 8: Victim Cache Datapath

Basic Hardware Prefetcher [8]

We created a basic hardware prefetcher to fetch the next line of instructions when there is an I-cache miss. The prefetcher communicates between the I-cache and L2 arbiter, an arbiter that decides whether to send I-cache or D-cache signals to the L2 cache. The prefetcher speeds up performance by performing future memory reads for instructions when the physical memory is not busy. Therefore, the pipeline only has to stall for about half of the memory instruction reads and the prefetcher will supply the other half. The prefetcher does not significantly improve performance when there are a lot of branches taken. This is because the prefetcher would fetch the values that would later be discarded to perform the correct instruction access.

When initially designing the hardware prefetcher, we connected it between the I-cache and a physical memory (pmem) arbiter in addition to the L2 arbiter. The pmem arbiter was inserted between the L2 cache and physical memory. When choosing which signals to send to physical memory, it prioritizes signals from the L2 cache over the signals from the prefetcher (Figure 10). Therefore, when the I-cache sends a read request, its signals are sent through the L2 arbiter, but when the prefetcher sends a read request, the signals bypass the L2 cache and sent to the pmem arbiter. We thought this design would be faster because it enabled us to grab data directly from physical memory instead of waiting an additional cycle for the L2 cache to respond. However, we realized that this design allows for the prefetcher to read from physical memory first before a D-cache read which is not ideal. In addition, the prefetcher would not check the L2 Cache for the line before reading from physical memory which might make the design run slower if there are many branches in the code. In the end, we decided to remove the pmem arbiter and have the prefetcher only communicate between the I-cache and L2 arbiter (Figure 11). This design ran slightly faster across all the test codes as shown in Figure 9.

Another modification we implemented was a 2-bit saturating counter that was used as predictor for the prefetcher. The 2-bit counter represented 4 states: strongly taken, weakly taken, weakly not taken, and strongly not taken. This predictor works in exactly the same way a 2-bit counter for branch prediction works (Figure 13). These states were used to predict whether or not the prefetcher should fetch the next line of data on an I-cache miss. However, when testing the runtimes of various test codes, we discovered that the predictor does not improve run time and in some cases, it even decreases performance. The reason for this could be that the type of predictor we implemented does not accurately predict if a line of data should be prefetched or not. In addition, the stalling due to an incorrect prefetch might not contribute to a significant decrease in performance, so even if the predictor works well, it would not significantly improve performance. Since the predictor did not improve the performance of our design and it adds extra logic which could decrease our maximum frequency (F_{max}), we decided to remove it from the final implementation. The table below shows the runtimes of the four test codes with our various prefetching designs.

Description of Feature	Competition_1 (ns)	Competition_2 (ns)	Competition_3 (ns)	Final (ns)
prefetcher connected to pmem_arbiter	1,056,410	334,000	11,202,120	803,545
prefetcher connected to L2_arbiter	1,056,160	325,550	11,157,770	801,100
predictor for prefetcher connected to L2_arbiter	1,056,159	327,110	11,157,770	801,110

Figure 9: Performance Analysis of Hardware Prefetcher Designs

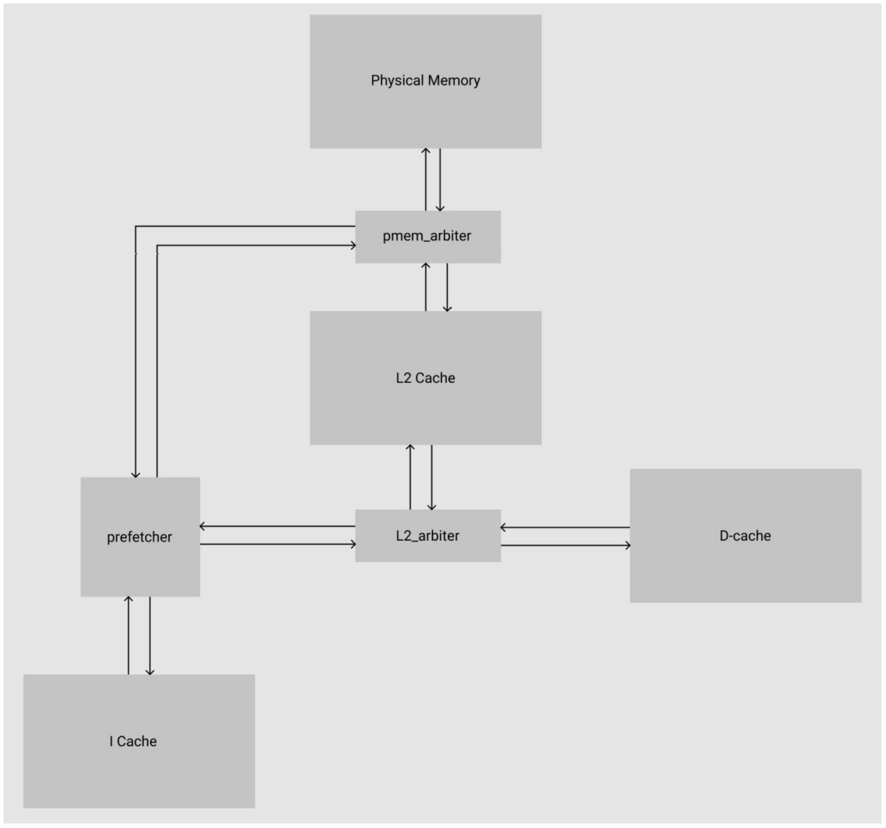

Figure 10: Block diagram with initial prefetcher incorporated.

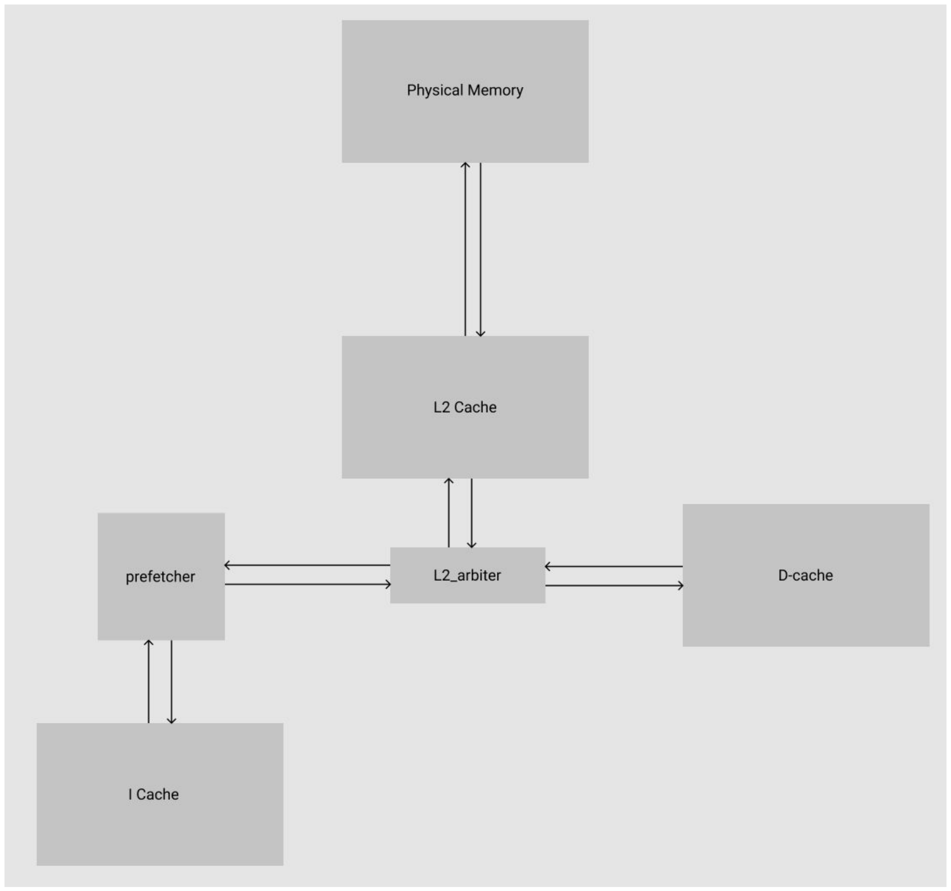

Figure 11: Block diagram with final prefetcher incorporated.

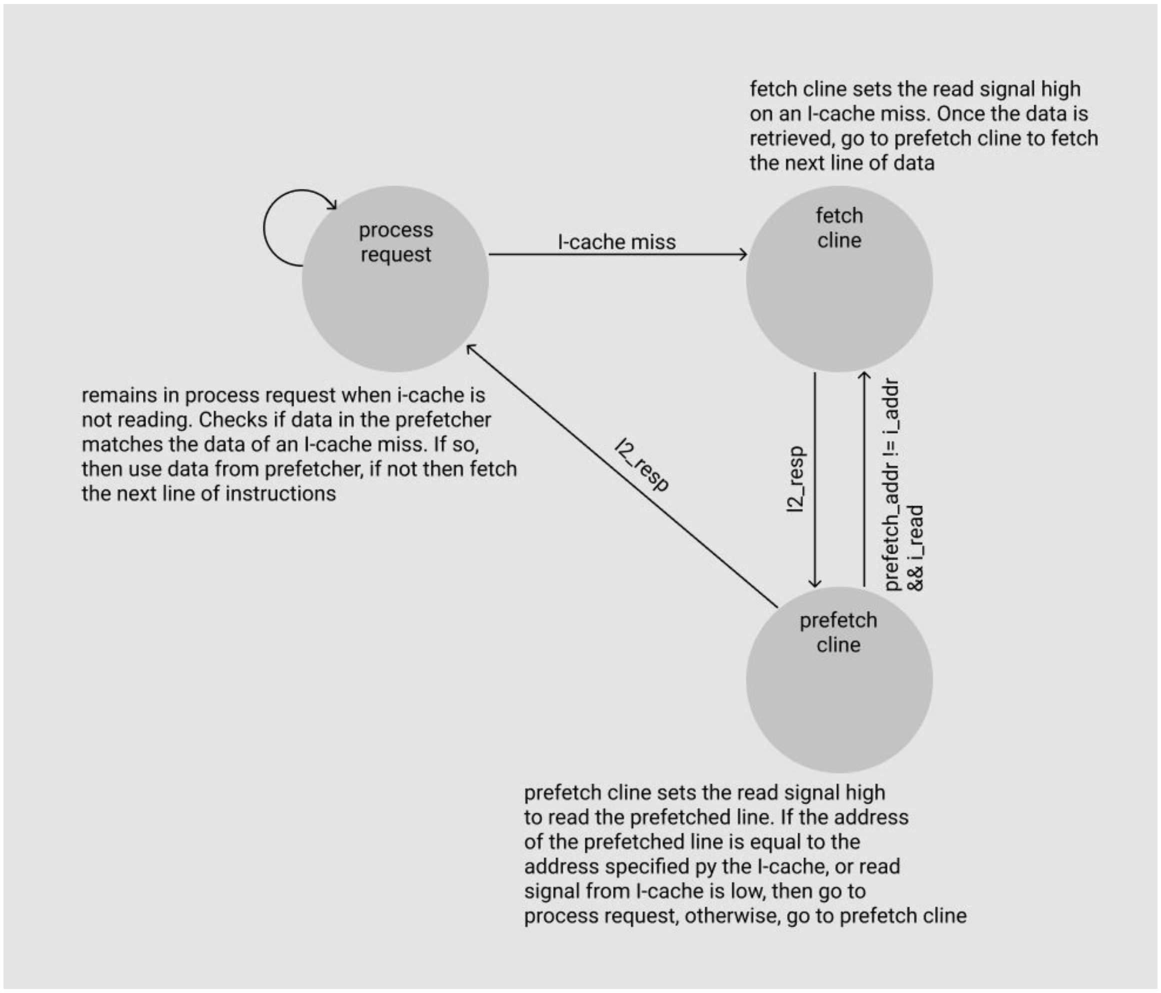
fetch cline sets the read signal high on an I-cache miss. Once the data is retrieved, go to prefetch cline to fetch the next line of data
process request
I-cache miss
fetch cline
remains in process request when i-cache is not reading. Checks if data in the prefetcher matches the data of an I-cache miss. If so, then use data from prefetcher, if not then fetch the next line of instructions
l2_resp
l2_resp
prefetch_addr != i_addr && i_read
prefetch cline
prefetch cline sets the read signal high to read the prefetched line. If the address of the prefetched line is equal to the address specified py the I-cache, or read signal from I-cache is low, then go to process request, otherwise, go to prefetch cline

Figure 12: Control Unit of Prefetcher.

Local Branch History Table [4]

Although static branch prediction is better than no branch prediction, it lacks the ability to react dynamically to prediction results (i.e., whether or not a prediction is correct or incorrect). This can lead to unnecessary pipeline flushing when the branch predictor mispredicts the same branch instruction for a second or third time in a row because it is not accounting for the fact that the branch was taken the last time it was encountered. Such flushing severely impacts programs with a lot of branches, especially those compiled from code with inlaid for-loops. To address this issue, we implemented one-level dynamic branch prediction in the form of a local branch history table. This form of branch prediction relies on a two-bit counter known as a "saturating counter," which is designed to store the previous results of a particular branch instruction. This counter has four states: strongly taken, weakly taken, weakly not taken, and strongly not taken. Its state machine is shown in Figure 13 below:

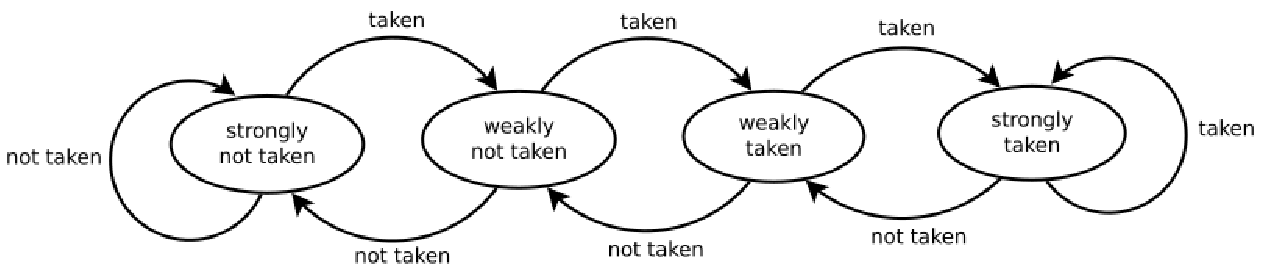

Figure 13: State machine for a two-bit saturating counter.

Effectively, a branch will be strongly taken if it has been taken the last two times it has been executed (and vice-versa for not taken branches). This builds more resiliency into the predictor, as it accounts for the case in which a branch is taken every time except once. If program execution ever returns to that branch, it will be correctly predicted as taken since the one time it was not taken only caused the counter to transition to the weakly taken state, rather than a not taken state. As a result, the predictor will mispredict a branch only once per loop on average, rather than twice. In order to associate each branch instruction with a counter, we implemented a branch history table that holds 32 saturating counters. This table is indexed with bits five through one of the memory address, thus allowing us to associate a branch instruction with a particular counter. Unfortunately, this method of prediction is ineffective if there are a very large number of branches in the program- because there are only 32 counters, the prediction for a particular branch will get muddled with those for other branches. Power and area constraints make this a difficult problem to solve; for example, it would be infeasible to have 2^{16} saturating counters. Fortunately, as indicated by our benchmark results in Figure 14 below, this method of prediction succeeds in improving performance for programs with inlaid for-loops.

Description of feature	Competition 1 (ns)	Competition 2 (ns)	Competition 3 (ns)	Final (ns)
branch prediction, prefetcher connected to pmem_arbiter	923,455	324,065	6,444,275	691,825
branch prediction, prefetcher connected to L2_arbiter, prefetcher predictor	923,205	317,605	6,399,925	689,515
branch prediction, prefetcher connected to L2_arbiter	923,205	315,935	6,399,925	689,515

Figure 14: Benchmark results for the local branch predictor.

These results indicate that competition_1.asm, competition_3.asm, and mp3_final.asm all got significant boosts in execution speed due to the introduction of dynamic branch prediction. In fact, the execution time of competition_3.asm was reduced by 42.9%! This is because competition_3.asm consists of many layers of for-loops. In contrast, competition_2.asm did not really receive a boost in performance due to dynamic branch prediction. This is most likely because the program does not have many inlaid loops (I noticed a lot of subroutine calls when I looked at the code).

<u>V: Conclusion.</u>

In conclusion, we successfully designed and implemented a fully pipelined LC3-b processor with advanced design features such as a victim cache, hardware prefetching, and dynamic branch prediction. We learned a lot from this project, including processor design techniques, debugging strategies, and group work strategies. We also learned that not all theoretical optimizations actually lead to improved performance on real code- and for those that do, there will always be some subsample of code for which the optimization doesn't perform. Although we consider our project a success, there are a few things that we would have liked to try, given more time to do so. First, we would have liked to implement a parameterized cache- this would have enabled us to easily test different cache sizes and benchmark how they affected performance. Next, we would have liked to use performance counters to determine the optimal number of lines in the victim cache. Unfortunately, this was not possible because different group members worked on the victim cache and the performance counters, but at the same time. Therefore, results from the performance counters were unable to be incorporated. Finally, we would have liked to try out different branch predictors and benchmark their performances. Unfortunately, we simply didn't have enough time to do this as we only successfully implemented one-level branch prediction one day before the deadline. Overall, however, we accomplished what we set out to do.