MP3 Checkpoint Four Report

ECE 411: Computer Organization and Design

Room 3032

Robert Altman / raltman2

Robert Jin / naiyinj2

Yan Xu / yanxu2

Document Contents:

Checkpoint Four: Implemented Eviction Write Buffer

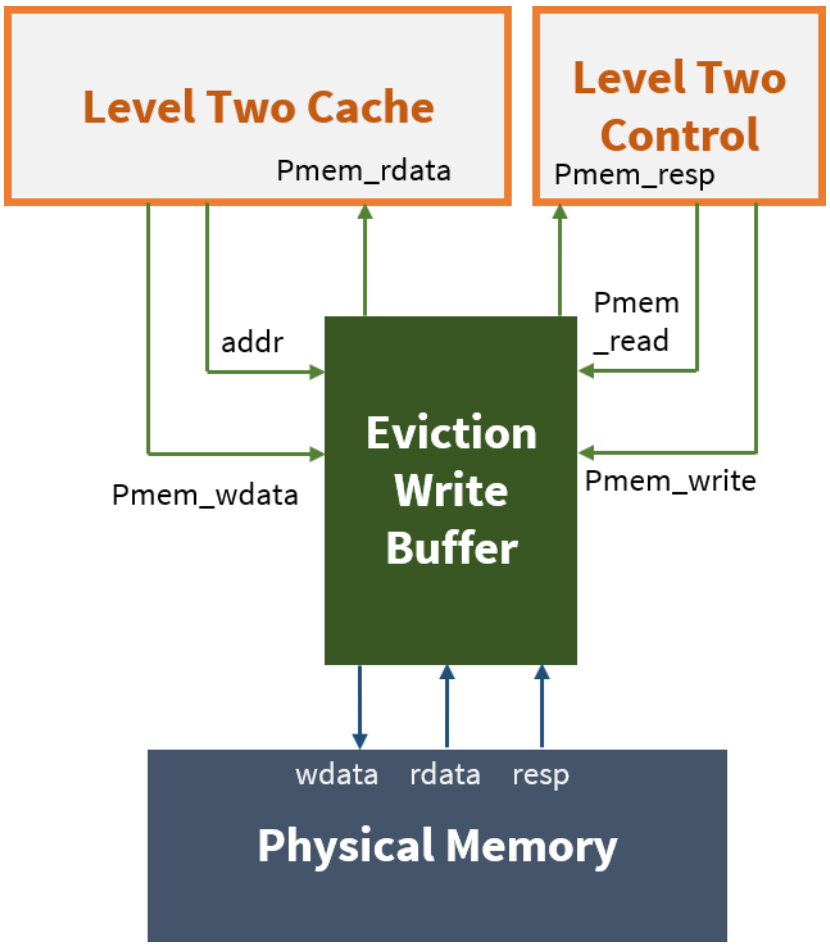
In this checkpoint, beyond the already implemented static branch prediction mechanism demonstrated from Checkpoint Three, we began differentiating our pipelined processor design with advanced design features. One of these categories was to optimize the way that writebacks are handled from the cache to physical memory, as much of the latency is induced by having a separate writeback and read stage in the original control that waits for memory to first commit old data and take new data. This component is the eviction write buffer. Here, we offer a single physical data line buffer for this implementation as proof-of-concept.

Despite internal conversations on expanding the buffer to multiple entries for further iterations, we have decided to keep the EWB's footprint small for now. Already, we have noticed significant improvement in runtime for most load-and-store sequences available. Since further growth to this buffer would create another module in the memory hierarchy modelled after a cache and further induced latency as a result if constantly writing back (presuming that our physical memory does not allow for synchronous reads and writes, which it currently does not), we have decided not to expand on the buffer further. An expansion would also increase the space, number of cycles, and logic complexity associated with the buffer. This is not desired in a design that is already struggling in terms of clock performance, so we will wait to consider any changes only until we have satisfied other combinational logic path constraints unresolved still in our original cache hierarchy.

Thus, the eviction write buffer is a single register file with control and multiplexer that holds an address and data associated with an address that will be written back here. That is, whenever there is a writeback that occurs, the data being evicted is written into this buffer to be written back at some later time, first telling the cache that it has accepted its data by simulating the response signal it expects to get on completion. Since we know in the single-buffer case that a writeback will be followed by a read of memory, we allow this read to happen and effectively cut the access time of this miss in half by avoiding a writeback first. Only when this read is serviced can we continue to writeback, delaying any other new read requests until we can clear the buffer and service the write that we delayed earlier. This still holds significant positive performance impact in the interim, as data may continue, and the pipeline stop stalling while we handle a writeback when the cache and physical memory is otherwise not being used.

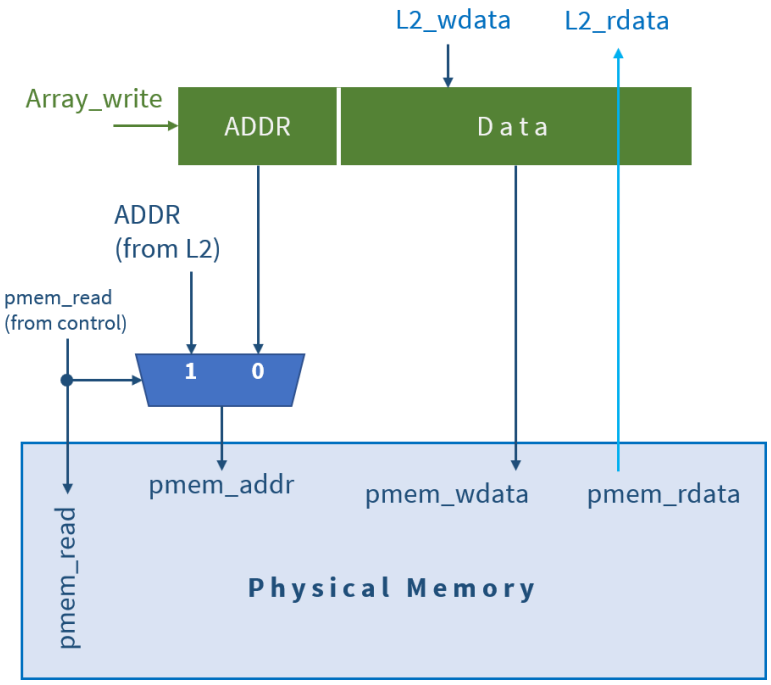
Should the design be expanded to multiple rows, note that a valid array would be required for this mechanism and the writeback stage would not be available in the same way – determination on what the most recent addition was would need to be included. Further, reads could happen whenever the pipeline was not full, but writeback could still occur as soon as data is present nonetheless, forcing reads to wait for a writeback that was previously committing as in this scenario (but worse, stalling when the queue is not empty in this model…thus stressing its need for revision).

The primary integration point, datapath view, and control is expanded on the following pages.



Since the eviction write buffer is meant to reduce latency caused through expensive writes, we found it is most effective to place it between the level two cache and physical memory, where writing costs the most amount of cycles. To do this, however, the buffer intercepts and routes the physical memory's data to the level two cache, take the 256-bit data from the level two cache to write and place it to physical memory, and also intercept any response from the physical memory and silence it if it does not line up with the read-write cycle the level two cache expects. For this latter case, the eviction write buffer will instead take care of the response handling, routing indeed this response signal on read requests from the memory but generating its own response on write and silencing the one physical memory gives on a writeback. This will become apparent from the utilization of the control, which is covered two pages from now.

Other data that would go to the physical memory, such as the address, read, or write signals, also must be intercepted by this eviction write buffer. This is done, as the buffer effectively enforces its own order of handling these requests instead of using the order given by the L2 cache. (This is slightly like the arbiter's determination of which data to handle at a time.) Further, the data for a read or write may not be used right away or sent to physical memory, may be used to drive control, or may be substituted with the address data that is in the write buffer itself.

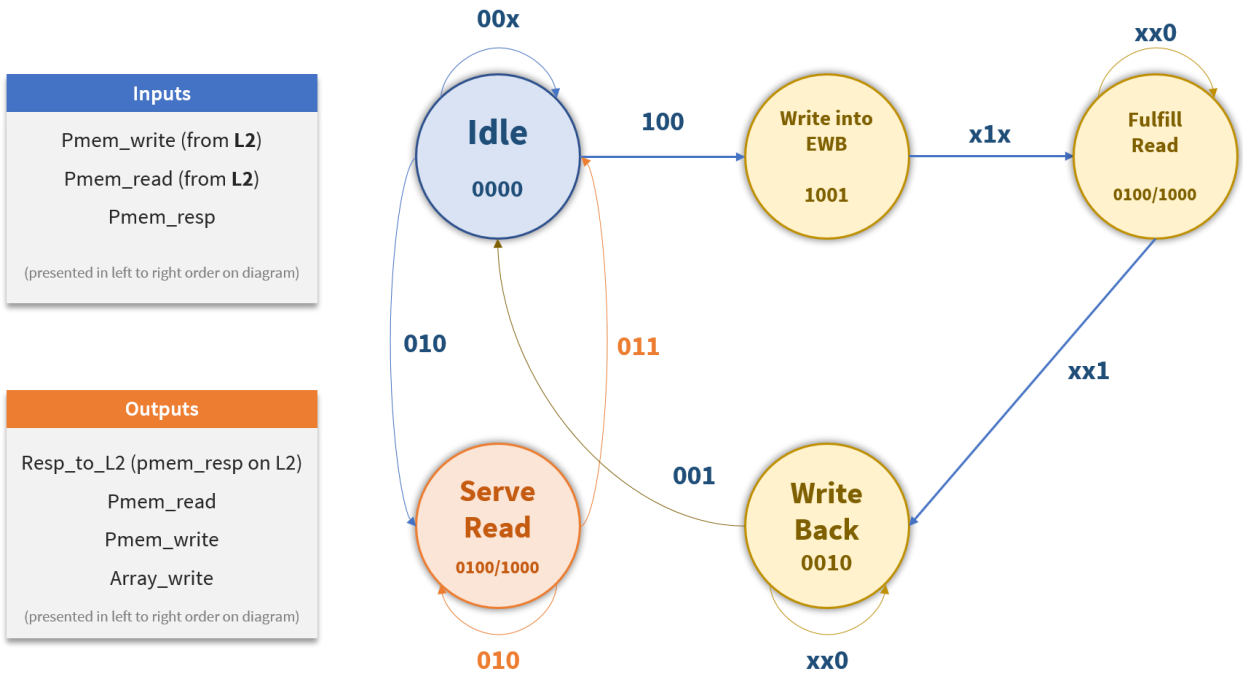


Whenever data is available from the level two cache, we want to place it in the input of our data buffer, should there be room available. However, this is not always the case, especially if a previous writeback is not serviced. Therefore, we rely on a control signal output *array_write* that tells if the address and data contained in the buffer can be replaced. This allows us to save array storage for a valid bit between memory reads and writes, as we know a writeback will be serviced after a writeback-read L2 cache sequence is intercepted.

Any data that is written to physical memory, thus, does not come directly from this L2 cache write data, but must already be available. Data from the physical memory module is quite the opposite, however, where this buffer cannot hold this information and expects to service it back to the cache as quickly as possible. Therefore, the data is sent past the data “buffer” used here without storing in. This is especially helpful in cases where the buffer has already committed, but there is still a read operation anyway, where the buffer can act as transparent as possible throughout the process (except for a single cycle delay for transitioning between *Idle* to *Serve Read*, detailed in the next page).

One other subtlety arises, however, in that the address to use for physical memory depends on whether a writeback is being completed or a read is allowed. If a read is allowed, then we can just take the value from ADDR that was passed into the eviction write buffer (via the diagram visible on the previous page). Otherwise, if we are writing, we need to use the address corresponding to the data we write, acting as a tag of sorts. The signal that goes out to pmem_read that differentiates between these operations is given by our control here.

Note that the pmem_write signal from the level two cache, as well as the pmem_read signal, are missing from this datapath. They will be used by the control instead to drive state transition and are not used in this logic otherwise.



The control logic provides the physical memory signals that determine whether a read or write is happening, placing the order of these reads and writes to memory completely to the control of the eviction write buffer. Our EWB starts in the idle state, which signifies that all data that is in the write buffer (if there is any) has already been written and new data can be stored. If our cache just wants to read, then we intercept this request and allow it until getting a response back from memory. When finishing this request, we came from a state that had an empty buffer, so we return to this state.

Say, however, that a write request is received instead. In this case, we need to fill our buffer, and so we intercept the request and write into the eviction write buffer. Note that, according to the diagram above, we respond with a high signal for our response to the level two cache, which it interprets as a response from physical memory. However, no actual write is done here. Instead, we wait until a read signal follows as normally done from the L2 cache, since its control is written to send the writeback data first and then request a read. Once this signal comes in, we fulfill the read in a very similar behavior as our *Serve Read* state. This separate state exists due to different transitions, however, due to a different status of data that is stored in the write buffer – when we complete this read operation, we cannot simply return to idle and allow new writes to happen, but we must write back the data we promised to handle before this read request. Therefore, as soon as the read request is done, we send a response to L2 that the physical memory responded, allowing it to complete operation on data generating a hit in the cache now (1000 as output signals). In the meantime, we focus on writing back the data to physical memory, setting pmem_write as high for an output signal to specify the address from the buffer to be used for physical memory. Only after this writeback has been truly satisfied can we go back to the *idle* state – note that no other reads are allowed during this process nor any stores, since we are currently interfacing with memory and have a full buffer that needs clearing here.

Checkpoint Four: Emendations to Performance Counters

The implementation of our performance counters, as well as the memory-mapping protocol that takes low addresses and resets or reads from these counters, was largely like that already completed and proposed in the previous report. This stated, there were some notable changes that still differentiate our actual implementation from the original proposal, perhaps most notably the removal of a counter altogether and the modification of how clocks were used in the clearable registers.

Starting with the first notable change, we decided to remove a counter for the number of stalls caused by branch hazards. Recall from the proposed implementation that a counter for number of branch mispredictions has already been added, where the counter would increment whenever a branch was taken (since our static prediction was to assume branches were *not* taken here). Another counter was added for a special branch hazard calculation, however, that would count the number of stalls whenever a branch misprediction happened. Since the number of stalls caused by a branch misprediction would always be twice the number of branch mispredictions that happened, and the latter information is already provided, we thus decided to remove this counter.

Because of removing this counter, the memory-mapping was moved up with MEM→WB hazard-induced stalls was moved to 0xA, the previously used memory address. Therefore, the following is the current configuration for memory mapping:

0x0	Count of hits in Level One Instruction Cache
0x1	Count of misses in Level One Instruction Cache
0x2	Count of hits in Level One Data Cache
0x3	Count of misses in Level One Data Cache
0x4	Count of hits in Level Two Cache
0x5	Count of misses in Level Two Cache
0x6	Number of branch mispredictions
0x7	Number of branch statements encountered
0x8	Count of data cache miss induced stalls
0x9	Count of instruction cache miss induced stalls
0xA	Count of stalls caused by a detected data hazard (from MEM → WB)

As for our second modification, we simplified the complexity to clock interfacing and took out unnecessary logic to drive the clocks updating a register. In the previous implementation, we noted that our counters not used for stall counting only wanted to increment once, and thus the signal that could stay high over multiple clock cycles (for a cache hit or miss, for example). Therefore, we noted that the signal to enable our register and count up would be driven to the clock signal, automatically incrementing with the clock. This was also problematic, however, whenever one wanted to clear the register as this would also be driven by a separate processor clock.

To prevent driving the register with combinational logic, then, we have redesigned our counter to take a separate enable signal that is driven by this combinational logic. The increment signal will then only be taken when this enable signal is high. This is separated from a reset signal that is generated in

the control logic we have defined in our previous implementation. Here, our control signal logic (that generates signals like alumux1_sel, etc.) also controls these signals directly to switch between incrementing or clearing on the traditional processor clock signal, simply by evaluating what memory value was computed, as well as whether load or store was requested. (Recall that a store would clear the register, and a load would take the value in from the register associated with the counter here.)

To accomplish this change, however, this does mean that all cache counters would need to drive logic only once to the enable signal. Therefore, an additional state was added to the controls of the level one instruction, level one data, and level two caches to output their hit and miss status a single time per occurrence. Thus, while our clear signal is still connected to *L2_hit_clear* from the control output logic discussed in previous pipeline iterations, the enable signals are set to previously proposed combinational logic. Particularly, for a hit, we check that the cache is in the idle state, a response was given, and a read or write is active indicating that a hit is present. With a miss, any transition about to take place outside of *ready* (the cache idle state) to a write back or physical memory stage would increment the number of cache misses.

Note that, for the stall counters, we could simply keep enable high for all cycles that a stall was active. There is no longer combinational logic that is added to compare with the clock signal, determining when the register is active based on said AND combinational logic. Since the clock is again just connected to the processor clock instead, all ANDs have been removed that were previously proposed.

Checkpoint Four Progress Report

The proposed goal, as given in the original RISC-V documentation and as altered given our previous implementation of a static not taken branch prediction mechanism, was to add performance counters and memory-mapped I/O that would tell of certain performance activity in the pipeline given a program, and an eviction write buffer. The latter EWB is a single line in length, and stores both the address and data of an item that was requested for writeback until physical memory. This buffer waits until the read request that followed the writeback has completed, and then decides to start writing back the data after this is done. No other data can read or write until the writeback buffer is again available, as physical memory is being used throughout this process. Construction of the buffer in this way allows for a program to resume execution while a writeback is in process, since the program is not reliant on the data from the writeback until another physical memory read is issued. The former, addition of performance counters, is added within a larger module that receives memory-mapped I/O signals and produces a result multiplexed across all available counters (0x0 and 0xA).

Due to our development and confirmation via testing, we demonstrate multiple functions of the processor all working simultaneously with all hazards properly handled and data forwarded where possible, while tracking how many stalls are caused by these hazards, how many mispredictions have occurred, and how many cache reads and misses have taken place. We also present an eviction write buffer implemented as described in the above paragraph, allowing for stale data evicted from the level two cache to be delayed in its writeback to serve the running program faster.

Prior to beginning new work, however, we first wanted to be certain about the status of our static branch prediction as implemented in checkpoint three. Therefore, led by Robert Altman and Robert Jin, with Yan Xu available for some design consultation, we started diving into further tests dealing with branches by exploring high jump/branch active tests such as those we wrote for MP1. Once we confirmed that jumps were working correctly (causing pipeline delays appropriately each time, as jumps are branches that are taken and our scheme predicts *not* taken always), and all results were still evaluating as done in a single-stage pipeline (or no pipeline), we continued and started new tests to demonstrate this functionality for the purposes of this checkpoint. We started by modifying the test code that was provided for MP3 heavily. Specifically, we wanted to demonstrate the need to check what is present in the IR at each stage, and what enters the execute stage, and thus we spaced out branches according to the original style given to ensure that branch instructions are cleared when *pipeline_enable* was high or *EX_branch_mispredict* (a signal used later for counters) was low. Should the first signal not be high, we should instead expect that no data changes in these registers, and thus there is noticeable delay in the contents of the IF and ID transition latches.

After this test was written, we wanted to also test all types of branches that were used in a program, as well as different times to not take or take branches to complete execution. While Robert Jin led development with the first test, along with the help of Yan Xu, Robert Altman completed the secondary test that checked branch sequences in a manner of bunches. The first batch of branches had one where the branch should be taken right away, and the next two not taken (nor accessed). Therefore, two cycles of stall should be observed. The next should predict not taken right away, but

then go into a case where the branch is taken, given the way that unsigned values work in comparison to signed values (the values 1 and -1 were compared, and while -1 < 1 for the first case, -1 > 1 when the values are read as unsigned). Such a sequence where one branch is predicted correctly not taken, followed by another branch mispredicted and taken, is then duplicated with *greater than* and *greater than unsigned* cases. Therefore, the test would distinguish proper functionality if eight stalls were visible, seven branches were encountered, and in three cases a branch statement was able to follow another branch statement immediately (due to correct prediction to not take the branch). All of this is done before leading to the branch instruction to halt.

From this point, we inverted the sequence presented in the initial roadmap but still used it from the previous proposal as a guide to completing our work. That is, after confirming the functionality of the branch prediction (or so we believed), implementation of the eviction write buffer started. This work, unlike with the tests for the static branch prediction, was nearly entirely led by Robert Jin after design proposals and discussions co-led by Robert Altman. Robert first started by writing up the datapath to utilize previously defined variable-width registers that hold the data and memory address. These were both then connected to a signal defined as *load_reg*, driven by a control and assigned to static logic until the control signal would later be written. A multiplexer was then added to control between the addresses to present, in case data was to be read from the physical memory instead of being written back from the buffer. (In the case of a read, the eviction write buffer was to be as transparent as possible and only deny the request if it was also waiting for physical memory to respond to a write request.) This concluded the datapath for the eviction write buffer, designed according to the specification given in the previous design document and in the first section of this report.

After completing these tests, we first considered writing the control according to the document. However, we wanted to verify its logic by writing tests that would cause evictions, verifying the process in which evictions happened in the processor just with L2 cache connected to physical memory. These same tests would be used to test the effectiveness of the eviction write buffer, as we could tell simply by latency of the program (if a reduced number of cycles was caused by a single physical memory read being needed until later) to see if a writeback was delayed. Once this was done, Robert finalized logic by connecting all control signals as specified earlier in the design document. When running the tests, all results appeared to look promising with each case involving a store and evicted cache line generally finishing faster. This indicated a performance increase, which was expected.

These tests, admittedly, were shells of the original tests we and course staff had written for other portions of this MP, as well as previous MPs altogether. Therefore, Robert Jin led the writing of a completely new test, which upon running unveiled issues…about our branch prediction, which we already believed to be fixed. Indeed, Robert started writing the test such that one instruction caused an instruction cache miss, but also was immediately after the PC needed modification because of a branch statement preceding it. Dangerously, our previous pipeline design would have PC enable low because of the instruction cache miss. While this would safely give *no-ops* through the pipeline in the meantime, our access to the PC upon branch was temporarily blocked, and we could not commit the computation done from the execute stage. That is, the signal to PC is getting overruled by the instruction cache miss, where PC instead stays the same and is effectively disabled. This is a relatively

rare occurrence, usually only encountered in longer programs and rare given that they must happen such that the instruction from the “not taken” misprediction also causes a miss here. A fix was issued by inverting the order of the PC reset check, where an instruction cache miss could still stall the pipeline, but PC could be overwritten *if and only if* a branch or unconditional branch (jump) deemed it necessary for the next instruction. If this could not happen, then the instruction after the branch would be invalid (the one if not taking the branch, not the one where the new PC is specified), leading to incorrect behavior.

Once this was patched, the remainder of the test was written by Robert Jin, with Robert Altman providing analysis and Yan Xu also providing some assistance in interpreting results. The eviction write buffer tests are based on a set of tests written for MP2 cache line evictions first devised by Robert Altman. Here the entirety of eight cache lines were defined in specific segments of code, and loads would strategically cause the cache to writeback after previous stores, should the caches be working correctly. This can be adapted for the new eviction write buffer and level two cache first by expanding on the number of lines available - this is 16 in the L2 cache.

From here, Robert Jin expanded the test for its new purpose - checking separated write back cases and ensuring other arithmetic and cache store operations could continue while the eviction buffer handled a writeback. The first line would be loaded into the cache first, bypassing the cache write buffer for the most part, but noticeably taking an additional cycle to complete. By showing the control states of the eviction write buffer, we can demonstrate that indeed an additional cycle is taken to transition from idle to our Read from Memory stage, without adding any other additional latency between our interface to physical memory.

We then transform the value loaded to a different register to signify 0x78, 0x56, 0x34, and 0x12, put together as 0x12345678. We do this and store the data, however, to a separate location in memory that causes a read again from physical memory - this is line 3, whose data is stored to the second way of the Level Two cache. However, this can only stay in the cache for so long as well, and once stores are complete, we will induce loads that cause our written line to write back. Since it is the most recently used line, we first take data from line 1 and load it again into a register, making the line we stored data to before (namely, 0x12345678) now the least recently used. From here, we load another separate line of data from physical memory (called "line two" in the test code) to evict this line that had data written, forcing a writeback.

This is where analysis starts. Our pipeline should be able to continue execution and even allow stores to the new data that we loaded into line two, all while physical memory is still engaged and writing back data. Therefore, there are multiple items to check just from this step, the first being timing and execution on the pipeline in a minimal amount of cycles. When the writeback happens, we should see that the level two cache only waits for a single cycle in the writeback stage, going back to reading from memory and having its highest latency from this step. Since the physical memory is typically fixed at about 50 cycles of delay for evaluation purposes in ModelSim, we should thus see only one set of 50 cycles of delay, and not 100 cycles as in implementations without the eviction write buffer present. This can be checked simply by looking at the number of cycles, the signal for pmem_resp given by the write buffer, and the control state of the L2 cache.

Second, we want to ensure that physical memory is truly engaged, but also that a conflict in the eviction write buffer will not disturb this process until writeback is complete. Therefore, we check the signal of physical memory and the control state of the eviction write buffer, ensuring that it stays in writeback stage immediately after handling the read-after-writeback from the current eviction it is handling. When another eviction comes in, there should be noticeable delay and this state should not change, even if the level two cache's control must wait now in the writeback stage for multiple cycles. This will signify that the EWB notices itself as full and is not allowing any new data until the previous data has been invalidated (stored to memory, in this case). Once this happens, only then should we see the control logic for our eviction write buffer to return and allow a response signal to be generated to the level two cache, taking in the new stored data. This data should remain unmodified, and stores 0x1234 in part (but not 0x12345678, so to differentiate from data that was just written).

To accomplish this and induce another check after we have written data to the newly loaded line as well, evicting this afterwards, Robert Jin wrote this new test suite to deliberately cause the actions and analysis described above. Other tests previously written to check load and store patterns, particularly that of the checkpoint two code for MP2, were amended as well with stores added to check that performance was doubled (latency cut in half) on writes to memory.

The final addition to our pipeline design was then to add the performance counters and memory-mapped I/O, which we did largely according to the plan of our original proposal. Robert Jin and Robert Altman started the work for the entire process, where Robert Altman enumerated all the conditionals and logic that are needed for the performance counters first before implementation, according to the design in the previous proposal. This was not, however, completed in SystemVerilog. From here, Robert Jin created the modifications to control logic first, with Robert Altman consulting on progress and approach that ensure the correct control signals are sent for memory-mapped addresses. Once the signals were properly tracked, tests were added to ensure these controls were intercepted properly.

The individual counter logic will then be made, as a standard module named *counter* that Robert Jin (instead of Robert Altman, as originally planned) constructed in large part. When doing this, however, all in the group had concerns about the current clock approach that was agreed on for the previous report. While Robert Altman previously believed that it was feasible to have the signal being counted in a register acting as a clock, this caused problems for stall counters that were dependent on the clock itself. Therefore, Robert Jin proposed removing the clock gating altogether, if a signal was added to the idle state of the cache whenever a miss and physical memory read/write would be detected. The group ultimately agreed on this, as even with the number of branches, the signal would only be raised over one clock cycle and could return control of the module back to the main processor clock (instead of perhaps driving additional logic that could further impact frequency conditions).

Once this was completed, Robert Jin continued and attached the new control signals created in the modified control signal generator module to a module consisting of eleven performance counters (one for each metric that was reviewed in the previous section). Note that this is one less performance counter than before, as during this process, Robert discovered the duplicate utility of the number of

branches mispredicted and the number of stalls induced by misprediction (which was just two times the former value). After this, a series of instructions was developed and run where the number of branches is known, specifically based on the MP3 test code, to test the individual branch and branch mispredict counters. The specific values, with explanation, are as follows:

55	**0x0**	Count of hits in Level One Instruction Cache There are 57 lines of instructions in total loaded from eight distinct memory locations, with a hit each time data is loaded again after an initial miss in the instruction cache. Given that instructions can also be loaded before another instruction has completed in this pipeline, and the instruction cache must see it in the IF stage, this would count if found in the instruction cache as a read and hit instruction. Note, however, that in our code this is one of the first instructions out of the 11 to evaluate. Therefore, 55 instructions actually pass the instruction cache by the time this data is loaded properly in the MEM stage.
8	**0x1**	Count of misses in Level One Instruction Cache The initial set of instructions will always cause a miss in the level one instruction cache. However, there are eight distinct memory locations that contain instructions (57 lines of instructions in total) that are loaded by the time all load word instructions getting from the counters are retrieved. This will induce 8 misses full of 8 instructions each, or up to 64 instructions. Therefore, this value should hold.
7	**0x2**	Count of hits in Level One Data Cache NOPE, TEST, FULL, and GOOD are all memory locations that share a cache line with A. Since there will be a hit the first time the line is loaded *after* the miss has been processed, and TEST is loaded twice from memory, there are six hits expected. Further, a second line is loaded into the cache, which after the initial miss, will cause a hit in the level one data cache. This totals to seven hits in all.
2	**0x3**	Count of misses in Level One Data Cache A and B, and data in the 256-bit chunks of data that they share space with, are the only cache lines that are ever loaded during execution of the program. This is only two times that unique data must be read into the cache, thus, and two misses should result.
10	**0x4**	Count of hits in Level Two Cache Each time there is a miss and a read back from physical memory occurs into the level two cache, a hit occurs on the next line. Since there are 10 misses, thus, we should expect at least 10 hits that

		should be present in the same caches. Since the instruction and data cache would miss with unique memory addresses to pull from, thus not pulling from data in the L2 cache. Therefore, *only* 10 hits should be experienced in the data cache. (Namely, A and B, which caused two misses in the Level One Data Cache, were in separate cache lines from instructions.)
10	**0x5**	Count of misses in Level Two Cache One can simply check the sum of misses in the level one instruction cache and level one data cache to check if this value is correct. Any data that is present in the level one caches should still be available in the lower caches, as it has not been overwritten here. Therefore, 10 cache misses at level two are expected.
5	**0x6**	Number of branch mispredictions There are five branches that are taken within the test code that are given, by the time in which values are loaded into the registers. These can be found at lines 16, 20, 24, 32, and 133. Since a misprediction with the static not taken scheme is simply the number of branches that are taken, this number is correct if appearing.
14	**0x7**	Number of branch statements encountered There are five branch statements resulting in changing the PC (being taken), and nine that are not. These total 14 total branch statements that are encountered by the time of the loads.
68	**0x8**	Count of data cache miss induced stalls There are two misses that happen in the data caches, each of which take 250ns by the limits of physical memory to process. Given that a clock pulse happens every 20ns in the simulation, and this process happens twice, one should expect that 26 cycles are wasted simply on misses from physical memory. This, however, also includes the number of cycles taken to call on the level two cache for data, and how long this took to respond, in addition to the number of stalls needed to simply read from the cache even when the data is available.
312	**0x9**	Count of instruction cache miss induced stalls There are eight misses, and about 13 cycles per miss in the level two cache given the delay parameter that is used, hence about 104 cycles that are purely caused by delays from physical memory to get items in the instruction cache. However, it still requires one cycle to read from the instruction cache each time, which adds additional delay in our design, even if from the level one cache.

		When a miss occurs as well, just as in the previous case, there are even more cycles that are present. After noting this value, this exposed a critical issue with the way that our instruction cache is delaying the pipeline each time. We plan to modify this such that the read state (merged with idle) can produce a signal in time such that no-ops are not frequently visible throughout the pipeline.
3	**0xA**	Count of stalls caused by a detected data hazard (from MEM → EX) The initial test code started with one stall that would be necessary after loading A into register X1. This was expanded in this modified test suite by loading again and completing a branch in the next test suite, which relied again on data that was loaded in the previous instruction. Finally, near the cache miss control test, one needs to try to forward after a cache miss with the data here, which effectively also tests MEM → EX forwarding with a stall. This is three stalls in total, which should be the result returned at this stage.

Once these values are confirmed, we still needed to confirm that the registers would properly clear upon receiving a store command. Therefore, a series of stores were given to all the registers, with loads coming immediately after to check if the values have changed. Indeed, while the values should change, not all of them should be placed immediately to zero, especially those indicating instruction or data cache hits by the time a *lw* instruction returns. Thankfully, tests confirmed that these values were non-zero, but much closer to zero than they were prior to the previous evaluation, verifying correct functionality of both the memory-mapped I/O addition and operation of the registers.

Checkpoint Four: Statement of Individual Contribution

A breakdown of contributions to the state of the project thus far, by member.

Robert Jin

Robert Jin took the lead on nearly all code writing for this MP and coordinated most initial meetings that discussed implementation questions whenever issues or questions arose. His feedback and his work cycle were effective and critical to our combined progress, as he could point out inconsistencies or potential connections that we neglected in our first conversations, as well as redundant or problematic logic. By far, the completion of the MP would not have been complete and implemented had it not been for Robert's vigilance.

More than with the previous checkpoint, however, Robert returned to most of the code writing role rather than leading design changes or meetings that discussed design questions. (Instead, such meetings were held and led together between Robert Altman and Robert Jin.) Still, his commentary was critical in pointing out major necessary changes throughout this process, especially for our modifications to existing cache control to account for new performance counter metrics (for hits and misses within the caches). Robert (Jin) was also the first one to point out issues with branch prediction even from our previous implementation, as detailed in the previous *Progress Report* section.

This analysis came with the addition of many tests throughout this process, where Robert Jin worked with Robert Altman in his work throughout this semester to create new tests demonstrating functionality. Working often with Robert Altman to ensure that all added functionality was adequately demonstrated in our written test code, Robert Jin offered the first evaluation on agreed test code. As he was often the first one evaluating waveforms to check for the design's functionality, Robert was also often the first to provide reactions and commentary on where the design may need revision. These were often included in informal chats, but also in coordinated meetings to discuss changes to our agreed design that were necessary.

By developing the first set of tests and modification of old tests written in prior MPs (for checking evictions that would happen after stores, causing writeback to physical memory), Robert was the first to monitor the behavior of the eviction write buffer that was added here. This also made him the first to notice the branch hazard prediction bug that was mentioned in the previous part, where he quickly led a fix after discussing with the rest of the team. The rest of the counter tests were written in coordination with Robert Altman, specifically for sequences that would clear and load from the same counters to ensure the control of each counter behaved and cleared appropriately.

Robert Altman

Robert Altman again coordinated most large-scale discussion meetings upon request from Robert Jin and worked most on the design given on paper for the eviction write buffer, individual performance counters, and memory-mapping mechanism. This latter part was particularly confusing at first for the group, so the address and signal generation was discussed in small meetings with the group to ensure address calculation could be completed in a timely manner to prevent any data cache signals going high erroneously (loads and stores are used to read or clear from counters). Usually the first one waiting to draw on the whiteboard, Robert did most of his work during the planning and revision stages that led to completing this checkpoint.

Left over from the previous checkpoint, Robert Altman also completed much of the branch prediction decisions, rallying the team to work with static not taken and providing an overview of findings and motivations for using the *Static Not Taken* approach ahead of time. These were done in comparison to a standard stalling mechanism used when branches were found and was particularly raised and designed by Robert (with coordination with Robert Jin) as an alternative to adding more logic checking for branches in the ID stage (as was proposed in the previous document here). When Robert Jin found issues and repaired some of the initial issues held with an instruction cache as a result, Robert Altman stayed in communication for this and other cache control changes made (such as signals added for the purposes of storing misses and hits in performance counters).

Robert also oversaw development of more test cases and adaptation of previous tests. More so than in the previous checkpoint, Robert Altman worked with Robert Jin on mechanisms to test functionality and the rationale behind specific values used within the tests. These were specially created for easier evaluation of these features, but also to ensure that all specific cases between the counters (for example) were evaluated. For the purposes of the eviction write buffer test, the test separating memory space to force eviction of specific cache lines was used and written between Robert Altman and Robert Jin, finalized by the latter. This also helped for the construction of the performance counter tests. Outside of this, Robert finished some evaluation of the counters under the same eviction write buffer code as well, specifically checking that the number of misses was specifically tied to the number of loads forcing writeback or physical memory interaction.

Finally, Robert constructed all control and flow diagrams, as well as all published physical resources and writing throughout the conversation. (Robert Jin would typically provide annotated copies for review, as with previous checkpoints.) This includes all graphics that were done for the report, which were mostly derivatives of similar graphics he generated for completing the checkpoint. Because of this and familiarity with the group's work as well, Robert composed nearly all this Update Design Document for Checkpoint Four. All written components, as well as much of the summary for each member's contribution and future goals in the upcoming checkpoints, were devised with permission from the rest of the group by Robert here.

Yan Xu

Yan has been one of the first to offer primary review and meeting coordination, often leading with guiding questions in our meetings that challenged our ideas of planned pipeline design. However, throughout this process, Yan has been a bit less active through *this* checkpoint's initial design, composition, report contribution, and test writing. Yan would still question the flow of various sets of instructions prior to implementation in SystemVerilog, however. Unfortunately, Yan did not participate as much in whiteboard planning and in overseeing code to implement our pipeline, as well as evaluating the simulation results that coalesced. While we admit as a group that the number of physical meetings was decreased throughout this process, we are hoping to increase activity in this planning as we near especially difficult design options yet to complete.

Due to his typical work alongside Robert Jin, however, Yan continues to be a supplementary contributor to any programming in SystemVerilog that was completed to satisfy this checkpoint. In communication often with Robert, he would pose questions during its development and check over what was currently written before execution to ensure that the general flow matched that agreed upon on paper before here.

Some of this work has already started, with Yan completing some initial work for the advanced design options throughout this checkpoint. Therefore, while he has done less work to complete some of the verification or design that was necessary for this goal, he has been instrumental in moving our design forward for the final stages of development. These first steps, as well as the rest of our roadmap as we develop the final features, will be enumerated in the roadmap on the next page.

Checkpoint Five: Roadmap and Planned Member Contribution

A roadmap for interaction in the next checkpoint's goals, derived from current progress.

Our overall goal by the next checkpoint is to extend our current pipeline design with the addition of multiple advanced design features, all to increase the performance of said pipeline. To do so, we propose the addition of a four-way set-associative cache extension for the Level Two cache, global two-way branch predictor with branch history tracking, and branch target buffer. We also assert the completion of early branch prediction the execute stage as accomplished from our pipeline design, as of Checkpoint Three. While some branch prediction correction will change due to the addition of these features in the MP, the pipeline_enable signal and br_en confirmation that is accomplished in the EX stage will not change, thus still accomplishing this advanced design feature. Further, a *jump* bit will be appended to each entry of the branch target buffer for a static *taken* handling of all unconditional branches seen as jumps (assuming no self-modifying code is possible).

Although we note here that early branch resolution in the execute stage is complete, where evaluation also occurs for jumps at this stage, we will nonetheless modify tests used for proving static not taken branch prediction to demonstrate proper behavior of this advanced design option. This will include stacking multiple branches next to each other, as well as demonstrating how no arithmetic operation or any operation following the mispredicted branch is allowed in the execute stage (having its values cleared in registers that come before it). Though all members of the group are largely responsible for this design, Robert Altman and Robert Jin will place the most amount of time into developing these tests and confirming functionality. We do expect that, by having this functionality and only inducing two stalls at worst for a branch hazard, the performance of the pipeline improves with this feature existing by two cycles per branch hazard.

The first design option listed in the introductory paragraph of this section not completed is the four-way set-associative Level Two cache. Largely borrowing from the cache design that is already in place for the current Level Two cache, we will add two sets of dirty, valid, tag, and data arrays to expand the level two cache to four ways per cache here. This, however, comes at a much higher combinational computation cost, and is very likely to drive down frequency even if some evaluation is happening in parallel. Namely, the OR gate used to check between hit signals in each way's tag array is expanded to four pins wide, to start. A choice between four addresses will also need to be made when specifying a physical memory address, while filtering the correct way's used data will also increase in complexity.

All of this, however, is relatively negligible compared to the changes necessary for the Least Recently Used policy we adopt in this new cache design. In our previous L2 cache, we could simply switch one bit placed in a single LRU array whenever we detected a hit that matched the bit in this array. (A hit in a specific way would make it the most recently used, so if the bits matched, then the least recently used way has become the most recently used array and the other way must be made the least recently used.) With four ways, however, the space and time complexity for evaluation becomes higher if using a True LRU policy. Therefore, our group has agreed to use a Pseudo-LRU policy, where three bits of data is instead kept for eviction in a tree-like evaluation structure. Combinational logic is

used to determine what each type of array state computes to, typically in a multiplexer like fashion where the three bits are used as select signals, and tags and data arrays are placed into inputs with a specific output determined by these bits.

Yan Xu has already begun work to complete the LRU policy, using a separate evaluation model altogether coupled with a three-bit data array for the same number of indices used in the previous L2 iteration. This has been shared with work Yan completed to expand other combinational logic to four ways and add arrays, of which is not included in the published version of our pipeline for this checkpoint but has been completed thus far. Robert Altman will lend tests for cache evaluation based on those written for previous parts of this MP3 and MP2 to complete further evaluation, specifically extending an eviction test that forces four ways to be occupied and one of the ways to be evicted after storage in the cache. Robert Jin will also contribute tests during this process, offering help on evaluating any performance hit taken for frequency due to longer combinational paths. No cycles are expected to be added, and the procedure should cause less misses altogether in larger benchmarks. Therefore, the cache is expected to improve performance, even as our frequency may become further constrained by additional combinational logic used here.

The next feature that will be added is led by Robert Altman, with the global two-level branch history table. Having familiarity with branch prediction due to his design lead on the static not taken branch prediction and hazard detection, he will work closely with Robert Jin (and Yan Xu, especially for testing) in completing the implementation of this branch history table. Planned for placement in the Instruction Fetch stage to determine PC modification as early as possible (without using stalls), this scheme will utilize the lower half of the PC bits (16 bits, for 256 possible entries in the table) coupled with an XOR of a history register to determine what type of branch to take. There is concern about this approach, in that it requires determining a branch or jump outside of the Instruction Decode stage. Doing so in the latter stage, however, would require one cycle of stall still if there was a branch taken and predicted correctly.

Elaborating on this approach, a shift register will be created that holds the result of the last four branches that are used. Four bits are used, as the number of branches that are expected over the course of execution for a program simply by conditional statements (and less by, say, iterative structures such as loops) is about 4. This is echoed from some of the test cases as well, where four branches were taken on average across about fourteen branch statements presented altogether. These bits are then XORed with the last four bits of the last 16 bits of the PC that are used, using these last 16 bits as the scope of recent branches influencing future results mostly occurs within a fixed range of program history. Also, it is worth noting that the branch and jump statements used in RISC-V do not allow for the PC to change to an arbitrary 32-bit value anywhere in the range 0x00000000 to 0xffffffff through a single instruction, and thus the scope was limited considering this instruction width limitation. Finally, an XOR is used in combination with the PC bits for indexing, instead of OR and AND logic that could favor the bits used in the PC too highly and keep resulting bits either 1 (in the OR case) or 0 (in the AND case) no matter what the branch history is.

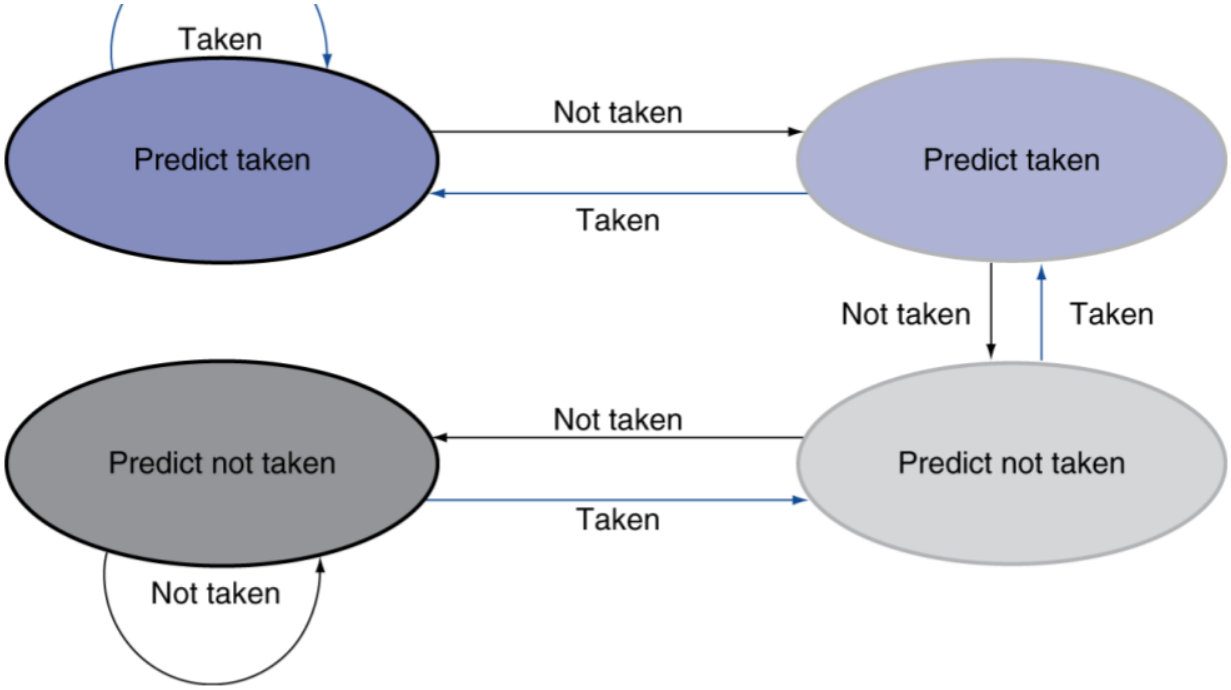


Figure 1: A class-provided copy of the two-bit predictor state machine that is contained at every BHT entry of this pipeline. (ECE 411 Lecture Slides, Lecture 09, Slide 19)

At each index that is specified in the global history predictor, there will be a state machine that is available which cycles between strongly not taken, weakly not taken, weakly taken, and strongly taken. The construction and transition logic of this state machine is precisely what was covered in materials for ECE 411 Lecture and is reproduced above. Each state machine will start at weakly not taken as its branch prediction, as this was the static prediction previously used in past iterations and induces the least amount of penalty to stall the pipeline if the destination is not previously known and prediction is correct. (Preloading this destination will be discussed with the branch target buffer, which will be covered shortly.)

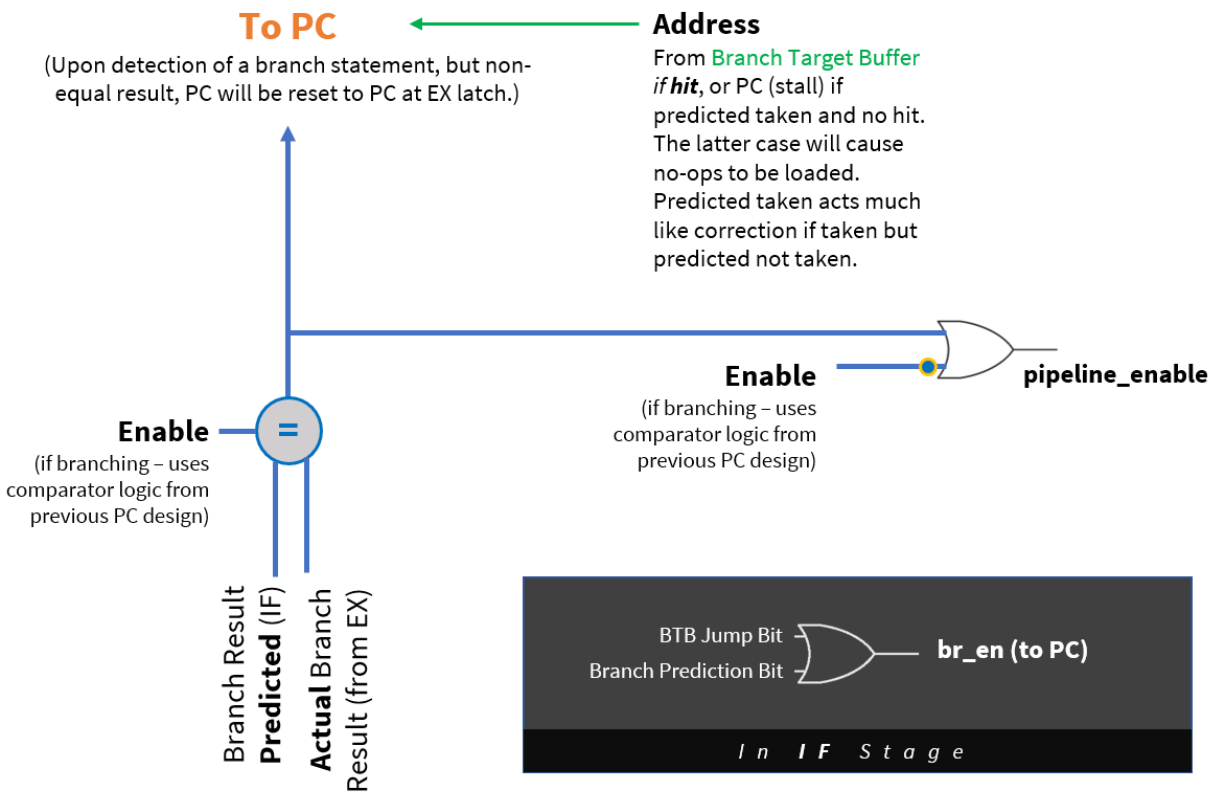
Note that this allows for some prediction of branches that are taken, which was fundamentally excluded in the previous task. Led by Robert Altman on design, however, with his and support of the rest of the team in implementation, taken prediction will be introduced into the pipeline by considering the value of Taken (1) or Not Taken (0) output by the branch history table. This value will be logically OR-ed with the current signal generating *no-op* instructions on instruction cache misses, thus sending no-ops into the pipeline whenever taken is predicted. This is done, as the value of which to send PC on a correct branch will not be known by the execute stage. Should the prediction be incorrect anyway, the most harm done is a clearing of no-ops and associated signals in the latches, and the PC stays the same value with the pipeline re-enabled. Notice that, however, this causes a performance decrease if we only were to introduce this history table – before, static not taken prediction would at least allow for branches not taken to execute faster, given that the instructions would never be cleared or denied entry into the pipeline. While this scenario of eliminating these stalls could not be removed in all cases, if we had already executed the branch and seen some sort of behavior, it would be nice to calculate the address right away based on this behavior.

Therefore, we are introducing a branch target buffer to complement the effectiveness of a global two-level branch history table. This branch target buffer acts similarly to a cache, with the entirety of the current PC loading an instruction used as the tag bits, and the data associated with each tag another value of the program counter held after the branch was evaluated. This data will be stored after a branch is evaluated whether or not the branch was taken, so that if the exact PC is encountered again in the IF stage, the branch target buffer (on a hit) can produce a valid PC. To do this, the hit signal will be output and given to the PC register in use, which will route the result from the BTB only when this signal is high. Recall that in our current design of the PC, depending on the value given by branch execute, jump from execute stage, etc., a different value may be loaded. Hence, this is an extension of our current design.

In the case of a jump, however, we would like to always take the jump if we knew about this instruction in the past, no matter if our branch prediction from the branch history table leads to a different result. Therefore, an additional bit is given on each set of the branch target buffer (effectively acting as a cache), where the bit is stored as 1 if a jump is associated with the instruction and 0 if a branch was instead. These stores are handled with the storage of the PC calculated with the branch, which is handled after the execute stage of the pipeline with the ALU output containing data, PC latch containing the PC of the instruction, and j_en signal indicating if a jump was associated with the instruction or not. It should be briefly noted that the policy for filling this buffer, which will be 16 sets in size, will simply be a counter modulo policy where a four-bit counter is incremented modulo 16 per store. Data that is collected after a branch or jump instruction has been executed will be stored at the value present in the counter and send a signal to increment it (an enable signal, in this case).

On read, if a 1 is found from the branch target buffer line's associated jump bit given that there was a hit found (this latter signal should always be checked first to confirm the validity of the value returned), then the value given for the address should immediately be placed as it is in the BTB to the PC register. However, the signal should also override any not taken prediction that may have come about, being ORed with the taken (1) / not taken (0) signal output by the branch history table to determine whether the address should indeed be updated in the PC register. When dealing with a standard branch instruction, only when the branch is predicted taken should the value of the branch target buffer have its address pre-loaded into the PC, if indeed a hit is available.

Further detail is given about some of the interaction between the branch target buffer, branch result prediction and correction, and pipeline enabling in the diagram on the next page. Note that pipeline will always stay enabled unless there is a signal indicating a branch has been detected and passed through execute stage, and the branch was mispredicted according to our comparator logic on the left side of the diagram here.



Notice that, had we completed a static taken implementation for purposes of this or the previous checkpoint (CP4 or CP3), our performance would fare no better than stalling always when a branch is detected since we evaluate in the Execute Stage. The static taken approach would be beneficial with the LC-3 architecture, where the destination is known before the condition codes may be set or ready from a previous instruction, but these condition codes do not come into play in RISC-V. Therefore, the maximum performance on a static prediction would be achieved by completing static not taken prediction.

With the addition of the branch target buffer, however, a branch that predicts a hit can lead to redirection of the PC and the next instructions after the branch to enter the pipeline right away. This increases performance greatly as it induces no stalls. This is especially true for jumps being present, where a static *taken* approach can always be used for this and the instructions can be available right away, even if the value stored in a register as the result of the jump needs to be forwarded to the next instruction via our procedures completed in checkpoint three. Given this fundamentally different behavior from our previous implementation, it may be wise to re-implement the stall counter caused by branch prediction.

Once the implementation for this is complete, Robert Altman will again lead (most likely with Yan Xu) the team to test and demonstrate functionality of these branch prediction advanced options. Tests plan to review both the global two-level branch history table and branch target buffer together, with the latter being tested last only after a series of jumps and branches have been completed. Jumps will

be tested with the introduction of a loop just prior to halting, run a set number of times with a branch following the loop to ensure that there is no delay in the pipeline obtaining the next instruction after the jump. The same will be done with a branch that is predicted to jump, where a forward, backwards, and forwards branch test sequence is planned that requires execution of a branch twice using the same taken property. (A *beq x0, x0, JUMP_DEST* instruction is likely to be used here.) As for testing the two-level branch history table, which shall be preloaded as all 0 (static not taken, as with our initial implementation), a sequence of simple arithmetic operations will be completed with all branches at first assumed not taken, given the way the initial setup of the value XORed with the PC bits will be arranged. After the 2-bit predictors have been primed to weakly taken for the series of branches, the sequence will be executed again with new PC values and force operation that takes all branches correctly. A noticeable difference should be found here in how fast registers are stored, as well as where delay happens in the pipeline, hence demonstrating the functionality and presence of this history table here.

As for the formal times that we plan to meet, outside of discussions that are completed as of this document's submission, we would like to meet with our mentor TA (Kenny) on Tuesday, November 27, 2018 from 1:00 pm to 1:40 pm. In particular, we want to overview some of our choices for where performance counters may be used and determine the feasibility and approach of our branch prediction options, as well as the number of ways to use in our Level Two cache. Additionally, we wish to review our current logic for the control structure on the write eviction buffer, as we feel that too many states are currently involved, and we could optimize for performance even further. As a reflection session and collaborative effort, we will take work collected after the meeting and review in group on the same day from 3:20pm to 5:00 pm and hold daily meetings that complement our normal text-based chat starting Wednesday, November 28. This will be climaxed by the meeting that same day, at 6:30pm, where our normal overview meeting occurs. By the time this meeting occurs, we would like to have the extended cache completed with the branch target buffer (due to its similarity to a cache) also implemented in the design. Meetings will continue Thursday and Friday, with the latter expecting that the entire design is complete, and the start of tests written to track the functionality of these features. This should also be where the first evaluations of performance using ModelSim should take place for a conceptually implemented final checkpoint. All sessions after this will consist of reviewing and reporting procedures as well as improving timing and frequency constraints of the designs, most of which are scheduled for Saturday and Sunday evenings. We are eager to start on these later considerations as soon as possible and are also willing to modify this schedule as needed to accommodate time for our mentor TA.