MP3 Checkpoint Three Report

ECE 411: Computer Organization and Design

Room 3032

Robert Altman / raltman2

Robert Jin / naiyinj2

Yan Xu / yanxu2

Document Contents:

Checkpoint Three: Revised Pipeline with Static Prediction

The implementation of our pipeline in this checkpoint was largely like that already completed and proposed in the previous report. Namely, all the forwarding paths remain unchanged from our previous proposal, despite some new considerations proposed on how to reroute the values from writeback to avoid latching altogether (thus skipping a WB → EX forwarding mechanism). We have decided that adding a latch here also allows for avoidance of longer critical paths (combinational paths), which would help our frequency to remain higher during the fitting process. This frequency has already become a problem because of the previous cache implementations, and with additional state logic connected to the caches on the way that may make for even longer paths, we did not want to induce additional constraints to frequency within the front-end of the pipeline itself.

That said, even in the previous implementation, we noted that there must be some way to detect and prepare for a branch mispredict. Earlier, we relied on checking if there was a branch and having all instructions stall before it in our previous implementation, using the data hazard mechanism and signal that we already used whenever there was an arithmetic operation after a load operation. However, this would not prove to work the same way, as we would have to stall the pipeline for two cycles instead of 1, check multiple latches for the value of branch, and still wait for the branch to evaluate in the execute stage. In going through with this approach as well, we would be adding additional logic to the Instruction Fetch and Instruction Decode stages, which are already congested and frequency-limited from our current designs leading up to Checkpoint Three.

Therefore, we decided to try an alternative approach to branch handling, in predicting a result by default and reversing our decision afterwards should we be incorrect. That is, we implemented a static branch predictor, in our case predicting *not taken* at every branch encountered. Not only was this the least taxing option in terms of combinational logic needed to do the branch check, but it ensured normal flow of the pipeline with clearance if our speculation was incorrect, much like a reorder buffer clears results after a branch mispredict was detected.

To do this, we needed to forward some signals from when branches occurred later in the pipeline but had to evaluate the tradeoffs of certain stages and early detection. The earliest one could know about a branch being taken or not taken beyond just the point of speculation was the *execute* stage, as the comparator module (labeled CMP in the datapath) is the one that computes the *br_en* signal that determines definitively whether to change the PC based on a branch. Since we wanted to induce the least amount of stalling as possible, we took this stage as the earliest in which branch enable could be ready to determine, and perhaps reverse a *not taken* decision. In the typical not taken decision, we would otherwise just determine to continue execution with the PC incremented by 4. If there was a branch however, then we would have mispredicted with this static mechanism. Instead of having to deliberately issue no-ops, however, we can simply clear out all items from execution, because this decision to check if branches were actually taken or not taken in execute means that all instructions loaded down the pipeline are not in the execute stage yet. If the signals from the instruction decode and instruction fetch stages are prevented from reaching the pipeline, our pipeline can act as if these instructions never were loaded in the first place (beyond the obvious use of stalls to accomplish this).

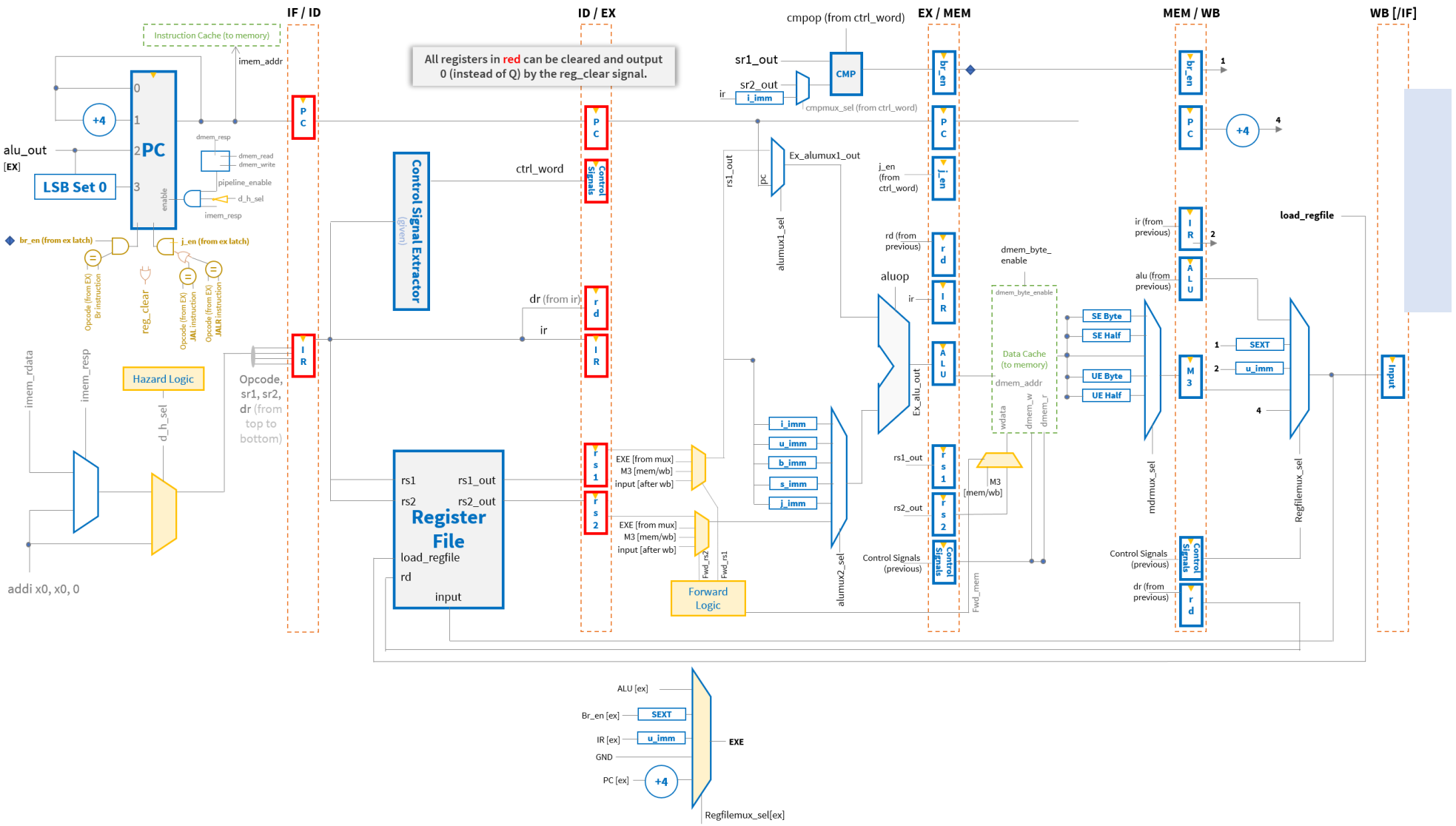
Therefore, we induce a mechanism such that the registers can be cleared given a certain signal, *and* the output of the registers is also clear such that data previously stored in them does not propagate through the rest of the pipeline. We would want to do this to all registers that are present in the IF and ID stages only as soon as we can guarantee the stability of br_en, and thus we wait for it to commit. Generating this signal requires knowledge that there was both a branch that existed and a branch was taken, indicating that our static not taken prediction was incorrect here. We can also take the result of this combinational operation and use it to change the PC value, using the branch enable signal that we used in previous iterations to update and take the value of a new PC. Since that value is present already from the execute stage, we do not have to wait for the writeback stage for it to be present and can instead take our execute signals and align them with execute data to update the PC and only induce two stalls. This logic is further enumerated in a diagram on the next page.

There is a slight danger in doing this clear operation, as the operation of an instruction containing all zeroes is not a valid *no-op* instruction. However, since one of the registers that is zeroed is the control signal register, no devices will be taken for read or execute, and no data will be stored or altered (effectively creating a no-op, thus).

It is worth noting, however, that this particular static prediction most likely performs worse than static taken or forward taken, backwards not taken approaches (despite being the most cost effective of the options). This is due to the common use of loops that are present that exploit branch behavior, and often using branches to get to completely different subroutines in our code *based on condition* (as an alternative, say, to the LC-3 based JSR happening only after a branch). While this was acknowledged, this was done again with the tradeoff of logic added to complete the prediction, which was minimal compared to logic needed to implement the other techniques. (Taken would require a refresh of the PC and early detection of this command in IF, and a similar problem would arise with FTBNT implementations.)

Once this static prediction mechanism was added, we could exploit the same techniques by detecting a jump and getting the new PC value during the execute stage, without “stalling” the pipeline but instead clearing the registers before it. As a result, this modification was made to update PC based on jumps, as these are essentially unconditional branches with a known prediction of *taken* each time. To complete this distinction, we could simply check the opcode out of the instruction decode state, clear all other registers based on the value of JALR, etc. being present in the opcode register in the set of ID → EX latches, and act. This is very similar to the approach taken for our branch distinction, where we checked if a branch occurred, and used the computation at ALU for the next PC.

To accomplish all these actions, a diagram has been placed on the next page for review. This is mostly a regurgitation of the same diagram that was presented for the first proposal, but some additional “branch logic” is added to this diagram to demonstrate how the PC is computed and where register flushing occurs (i.e. what “module” generates these signals). Hazard detection and forwarding logic, as it follows the same design enumerated from the last proposal, is given as a black box and will be expanded in detail in the *Progress Report* section.


IF / ID
ID / EX
EX / MEM
MEM / WB
WB [/IF]
All registers in red can be cleared and output 0 (instead of Q) by the reg_clear signal.
Instruction Cache (to memory)
imem_addr
PC
+4
alu_out [EX]
LSB Set 0
dmem_resp
dmem_read
dmem_write
pipeline_enable
d_h_sel
imem_resp
br_en (from ex latch)
j_en (from ex latch)
Opcode (from EX) Br instruction
reg_clear
Opcode (from EX) JAL instruction
Opcode (from EX) JALR instruction
Hazard Logic
imem_rdata
imem_resp
d_h_sel
addi x0, x0, 0
Opcode, sr1, sr2, dr (from top to bottom)
Control Signal Extractor
ctrl_word
dr (from ir)
ir
Register File
rs1
rs2
rs1_out
rs2_out
load_regfile
rd
input
EXE [from mux]
M3 [mem/wb]
input [after wb]
Forward Logic
Fwd_rs2
Fwd_rs1
cmpop (from ctrl_word)
sr1_out
sr2_out
i_imm
CMP
cmpmux_sel (from ctrl_word)
Ex_alumux1_out
rs1_out
pc
alumux1_sel
j_en (from ctrl_word)
rd (from previous)
aluop
ir
i_imm
u_imm
b_imm
s_imm
j_imm
alumux2_sel
EX_alu_out
dmem_byte_enable
Data Cache (to memory)
dmem_addr
wdata
dmem_w
dmem_r
rs1_out
rs2_out
M3 [mem/wb]
Control Signals (previous)
Fwd_mem
SE Byte
SE Half
UE Byte
UE Half
ir (from previous)
alu (from previous)
mdrmux_sel
SEXT
u_imm
load_regfile
Regfilemux_sel
Control Signals (previous)
dr (from previous)
ALU [ex]
Br_en [ex]
IR [ex]
GND
PC [ex]
SEXT
u_imm
+4
EXE
Regfilemux_sel[ex]

Checkpoint Three Progress Report

The proposed goal, as given in the original RISC-V documentation and as altered given our previous implementation of an L2 cache in the memory hierarchy, was to support all forwarding, as well as data and control hazard detection. This would allow for safe execution of all instructions in the pipeline without the need to induce software stalls. By completing this task, however, we also wanted to optimize the number of stalls that could be placed between any instructions with true dependencies, which would be a maximum of one between loads and arithmetic operations. By closely following our design proposed in the previous stage of this MP, we have implemented and run tests on *all* instructions and multiple branch scenarios with hazards placed deliberately in the tests, to ensure code from an incorrect branch prediction is not run, and register values could be read-after-read, written after read, read after a write (whether from memory or arithmetic operations), or written to after write (again, whether from memory or arithmetic operations). We have monitored the behavior of our cache closely to ensure any delays induced previously have been removed, thus tracking the amount of cycles taken in data reads and misses in different levels of the caches, and subsequent stalling in the main pipeline to ensure it is less than previously.

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. This ensures that even data that has not written to the register file is properly used by subsequent instructions needing the data, and that the data is correct here. First, we have confirmed and present a five-stage pipeline for the processor that handles all RISC-V instructions (excluding those like FENCE), with support for stalling instructions where data or instructions are not present from memory, but primarily limiting stalling to this and a read-after-load dependency. (Since the memory stage is one deeper than the execute stage, we require one stall or at least one instruction not using the destination register of a memory operation to be placed in the pipeline.) We also present a memory hierarchy including two level one caches (one for holding instructions, and another for data), as well as a cache arbiter for managing miss and read/writeback requests from these two caches to the rest of the memory hierarchy, *and* a Level Two cache of larger size that interfaces with the arbiter to process any data to be written from or back to the level one caches higher in the hierarchy. (These were already presented in the previous implementation as well, but emphasis is added that the L2 cache was confirmed working again with this implementation, and tests were completed to track its behavior here.

Starting with the initial roadmap from the previous proposal as a guide to completing our work, we began with adding the writeback latch that was previously not found in our designs, to ensure that a register can be available for instructions writing to a stage that may have been bypassed by an instruction needing its source register data. No forwarding was completed during this first preliminary step but do want to check our timing constraints and overall design of the datapath to ensure such an operation does not take away from our performance and does not invoke any new hazards that are needed to treat. Robert Jin lead the development of this, as he was the team member originally pitching this idea, with Robert Altman minimally available to discuss some of the advantages of the process while writing. As stressed through the previous report's plan as well, and fulfilled here, tests

were written by the entire team to check that no operation has particularly changed in the pipeline by adding this, except for checking correct values that are stored simultaneously to the register file and this new latch that has been present here.

From here, the additional multiplexer that mimics the execute operation decision in the writeback stage was added to the execute stage. The operation itself was a simple copy-paste of logic already completed for the writeback stage already but started our discussions on how to implement the control logic selecting forward data for the EX → EX forwarding. This multiplexer computes **EXE** for the multiplexers that can set RS1 and RS2 for forwarding will be inserted next. Robert Jin worked on the control signals driving all multiplexers through the design during this time, including this multiplexer, and stepped up as well to check tests and values that tested early execute with SLI and instructions not using the ALU.

Since this had already started the discussion of how control logic needed to be implemented, the team focused on the rest of the logic with much of the discussion lead by Robert Jin and Yan Xu. Once there was agreement, any pseudocode determined was translated by Robert starting with signals *forward_rs1* and *forward_rs2*. Out-of-step with the original plan, however, the team continued immediately after with all types of forwarding that were proposed in this checkpoint report. Starting with the execute to execute forwarding type, we created multiplexers for both SR1 and SR2 and left disconnected the signals for memory and writeback signals temporarily. Since the multiplexer that calculates EXE has already been completed, we connected this signal to input one of our multiplexers and wrote tests that confirm functionality before proceeding. Again, Robert Jin lead most of this process with some assistance from Yan Xu, and Robert Altman provided some testing of his own on a modified version of the architecture (that is, one that was a few stages ahead of this current state, thus indicating a bit of a delay in test writing versus progress made on the totality of the project). Tests were done with sequential arithmetic operations to ensure correctness, specifically batching special cases and multiple read-after-write and write-after-write sequences to stress test on similar instructions (i.e. and, or, xor as a suite). Robert Jin wrote and ran a large sequence of tests with Robert to test this implementation as well.

After this, MEM → EX forwarding was handled and led by Robert Jin. Since Robert Altman was the writer of the hazard detection and control that was done in the Instruction Fetch stage, he made himself available for consultation on the device's construction, and especially started to argue for its inclusion once this forwarding step was written. Robert Altman devised the branch prediction scheme that was discussed in the previous section to go along with the data hazard detection previously discussed, effectively presenting two courses of action. The first was that branch detection would be handled at the IF stage alongside of the data hazard detection and insert *no*-ops here with random stalls. The second was an option that could save time in execution even if often wrong, which was a static not taken predictor that continued to load the data and cleared out data after values were present for branch taken in execute. Here, Robert Altman and Robert Jin wrote the initial logic and approved the latter approach for implementation, such that this forwarding may be evaluated correctly later. Tests were specifically run that tested load and arithmetic type sequences (read-after-write and write-after-write) that are completed here.

Prior to finishing the final EX forwarding step, Yan Xu and Robert Altman were to start on writing the memory to memory forwarding design. However, Robert Jin presented an argument that the stall induced in handling MEM → EX dependencies could be extended to check for source registers in a store operation being used in the store type commands as well. That is, instead of just adding a stall whenever a dependency existed between a load and arithmetic operation, the same was done for memory to memory. Thus, the modification was made without any additional logic being required, even though the traditional module was included and present in the pipeline itself. (It was, however, that the module was not ultimately connected to anything.)

For the final type of forwarding (WB → EX), Robert Jin again led development and ensured all latches are updated in time to give to the execute instruction. The spacing between all instructions, both in this function and in all types of forwarding, was aided by Robert Altman mostly in concept, with contributions as well from Yan Xu. Tests were written or modified written such that different spacing levels are used to test each type of forwarding (with the final batch of tests using no spacing (no no-ops) at all), with three instructions separating read-after-write (or write-after-write) dependencies, two instructions for MEM→EX cases, and one instruction for EX → EX and MEM → MEM cases here. This will also offer the first chance to test much of the entire process with all forwarding types implemented, as well as testing the introduction of *no-ops* and different register values at precise times of execution. Modifications to old test codes that utilized no-op spaces in previous checkpoints also occurred, such that the original premise of the program written was confirmed here.

After all tasks were completed, a final frequency analysis was completed and discussed among the entire team. The frequency max was still exceeding 100 MHz, at about 105 MHz, with no additional critical path length induced through any of the process. Indeed, removing the caches still created a significant difference between the frequency available to run this processor. Therefore, Robert Altman continues to investigate long combinational paths and potential use of registers that would not impact the control sequence used currently by the level one and level two caches. This will continue to be applied concurrently with combinational logic timing analysis in the primary pipeline datapath now, after more multiplexers and combinational logic have been added to enable the process of forwarding here.

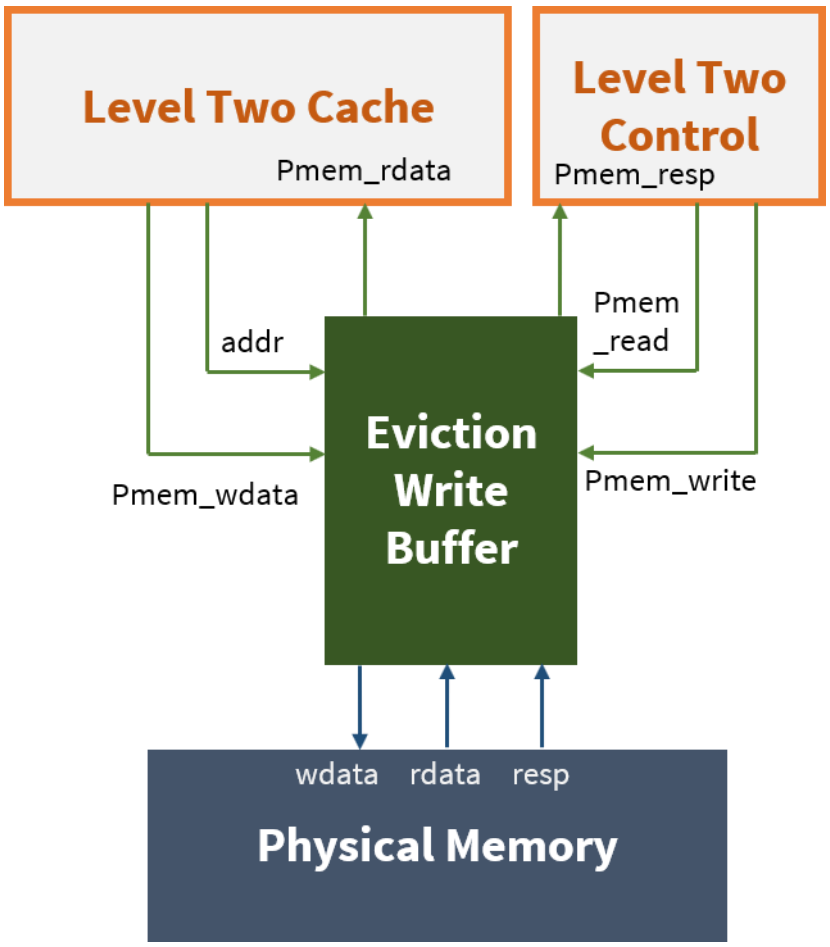
Checkpoint Four: Proposed Eviction Write Buffer

In the next checkpoint, we will start differentiating with advanced design features given a set of three categories of components. One of these categories, in static branch prediction, has already been completed for the sake of finishing this checkpoint and offering a stable mechanism of detecting (and handling) branches. A second component seeks 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, of which we offer a single physical data line buffer for this next implementation as proof-of-concept but have plans to expand on the buffer for better performance in later iterations of our processor.

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.

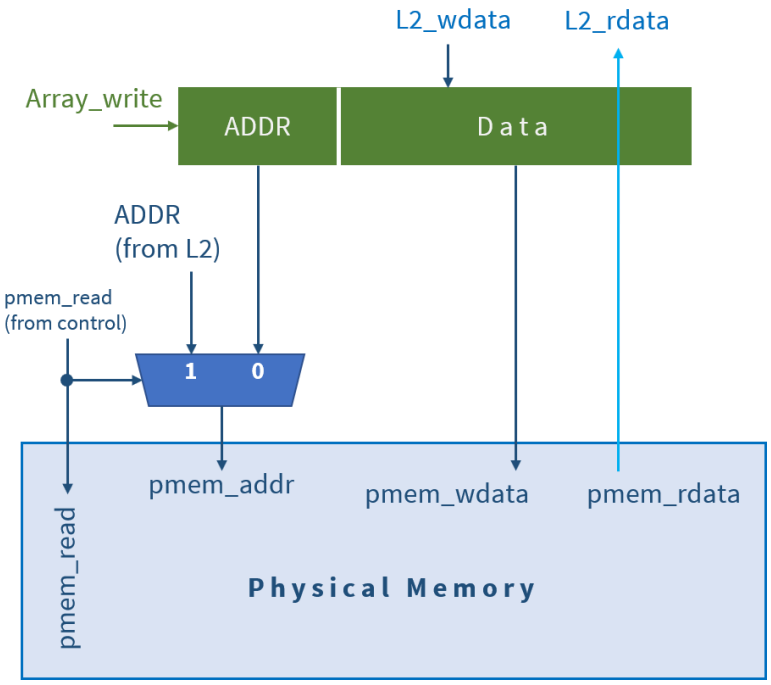
Should the design be expanded to multiple rows, note that a valid array would be required for this mechanism and the writeback stage would not be available in the same way – determination on what the most recent addition was would need to be included. Further, reads could happen whenever the pipeline 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 will be expanded on the following pages.



Since the eviction write buffer is meant to reduce latency caused through expensive writes, we find 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 must intercept and route 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.

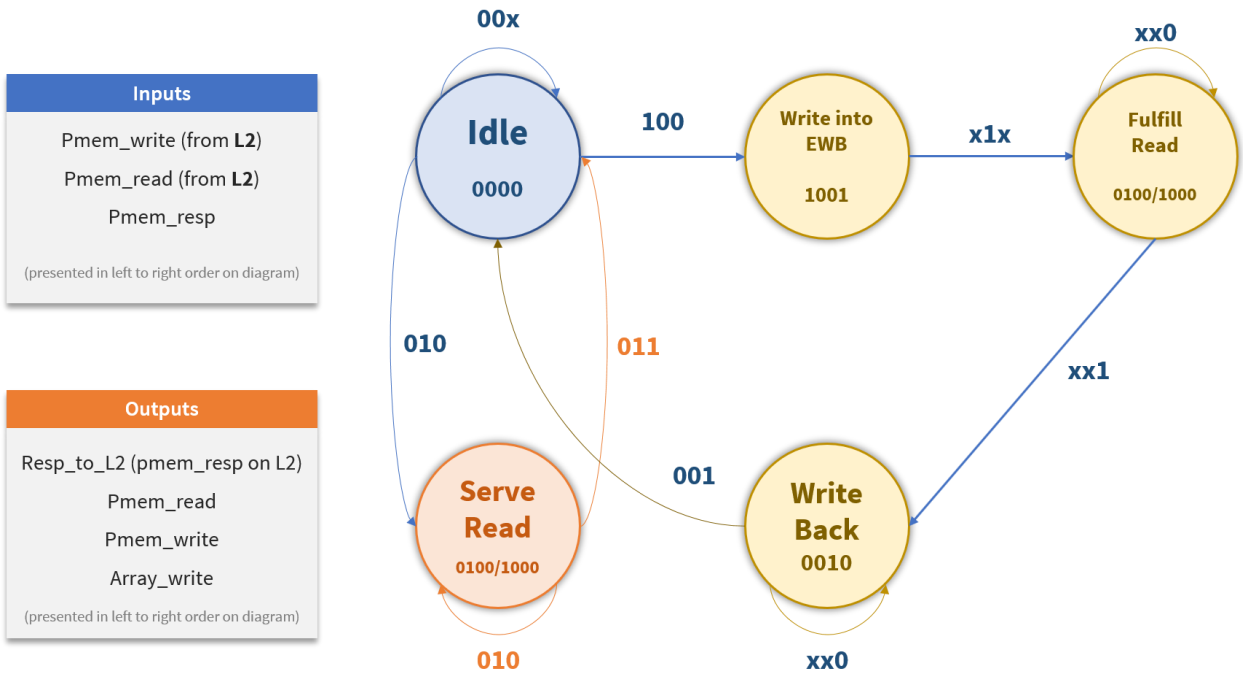


Whenever data is available from the level two cache, we want to place it in the input of our data buffer, should there be room available. However, this is not always the case, especially if a previous writeback is not serviced. Therefore, we rely on a control signal output *array_write* that tells if 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: Proposed Addition of Performance Counters

A performance counter, for the scope of how it is used in this processor, will keep an accumulating account of information in a memory-mapped accessible register. Whenever a store is issued to a certain memory region, the accumulator corresponding to that memory region will be cleared. Alternatively, however, it one reads from this memory address, they can access the value of whatever data is being stored at this performance counter.

Thus, there are two components to adding multiple performance counters into our design. The first is the actual component itself, which is mostly modeled as a register, and will be discussed on the page after next. The second is the memory mapped IO process that is used to either store or read from these counters, which is enumerated on the page following the first design.

There are multiple use cases to use these counters for, especially considering that this is constraining to performance. To track the tendencies of some programs to use certain resources over others, or cause more hazards, memory misses, or unwanted behavior that takes away from performance, we need to set up these counters in appropriate places so that they may be paged by a program or test suite (and collect metrics, for example). The first is that we wish to track the number of hits and misses per each cache, each of which tells us different information. For the instruction cache, this tells how many times we either execute longer sequences of code or emphasize higher code locality, whose data can be used to better optimize the pipeline based on history. With the data cache, the amount of information that is stored and/or the size used can be altered from historical information as well, or inform a program of memory accesses that are far away from each other and cause high latency due to irresponsible memory references. Finally, with the L2 cache, we can track how much latency is truly induced because of connecting and missing to physical memory by tracking the total number of times active (with hits and misses) and weighing performance by comparison within a program here. (Say, for example, that two consecutive reads are made for number of hits and number of misses into two different registers. The registers can then be compared, say, with an *sli* instruction that checks if the number of misses is less than the number of hits.) **For each counter, whenever there is a hit, we accumulate the value in each hit counter by 1. Similarly, whenever a miss is recorded, we accumulate the value in each miss counter by 1.** The logic for this is specifically designated in the datapath sample on the next page.

Especially in relation to the Level One cache data, then, we would like information about how well our static branch predictor is performing. Therefore, we will also use two counters for the number of branches that were predicted correctly, and how many branch statements were found altogether. **The logic associated with accumulating with branch misses is a logical AND between br_en and a match between the branch instruction and current instruction processed. The other accumulator, for number of branch statements encountered, simply runs a comparator against the instruction taken in and checks for the BR opcode.**

Finally, there are four different ways that the pipeline can stall here, and therefore we want to categorize these into four different series of stages in which stalls can happen caused by a certain phenomenon. The first is a data cache stall counter, checking how many cycles pipeline_enable is 0. This tracks the stalling that happens across all the stages, since pipeline_enable is given to all

registers for an enable signal. (In essence, the WB stage is not really tracked from this since the operation will write to a register file even when waiting for the data cache, but since the signal is given even to the new WB latch, this is added.) Therefore, whenever a new cycle comes up where pipeline_en is 0, we count it and accumulate. For this option, we will have to *AND* with the clock to get register behavior that accumulates on each cycle. This implementation can be abstracted based on the way that signals to accumulate are directly connected to the clock port of internal registers within these counters.

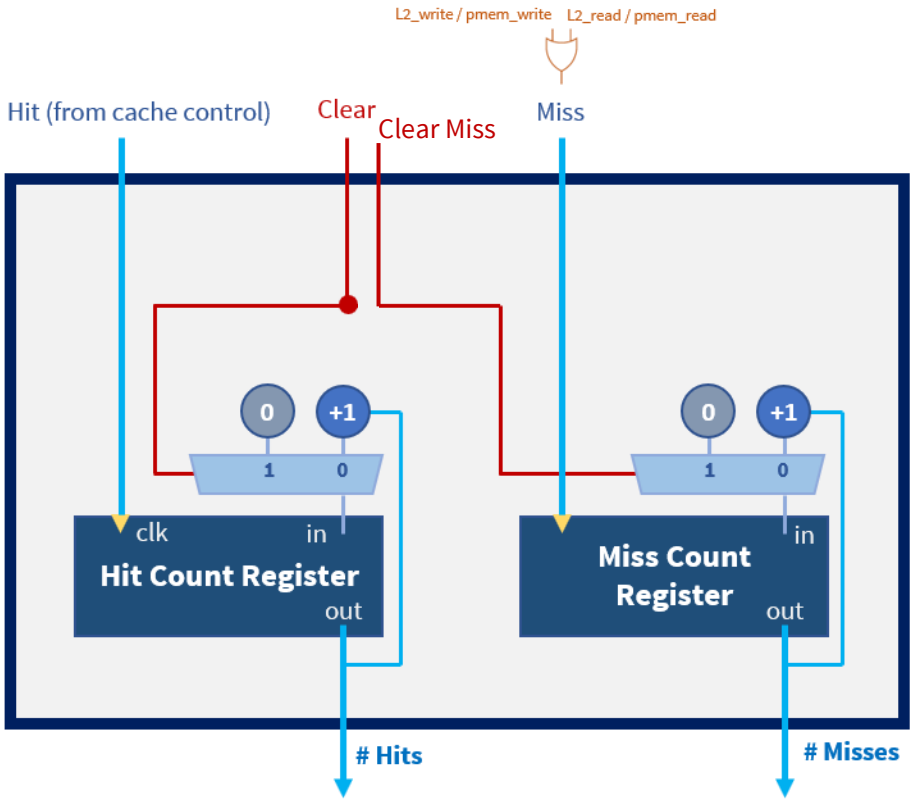
Similarly, we need to look at where the instruction cache stalls, which is how many cycles imem_resp is set to 0, but pipeline_enable is set to 1. Due to the latter condition, the only stage that is being stalled here is the Instruction Fetch stage. Therefore, we take the signal imem_resp and invert it, AND with the pipeline_enable signal, and logically AND with the clock as our value that goes into the performance counter input.

The third of the four counters that are used for stalls is a branch mispredict stall counter, which uses the same signal that our branch mispredict counter itself uses but accumulates by **2** each time a signal goes back high. Whenever this is raised, we know that IF and ID stages will have to be stalled, and therefore we want to track this as a set of stages that are being stalled. Therefore, there is a distinction between this counter and the branch mispredict counter already created here.

Finally, there is one more counter given for data hazard stalls, which are applied mostly across the ID, IF, and EX stages (and thus another unique set of stages). When a module throws NOPs, which will be known by the value in the IR latch of the ID stage, then we wish to accumulate. Since each stall is again a unique occurrence, we want to AND this with a clock signal so that each stall is uniquely taken and written into the accumulator.

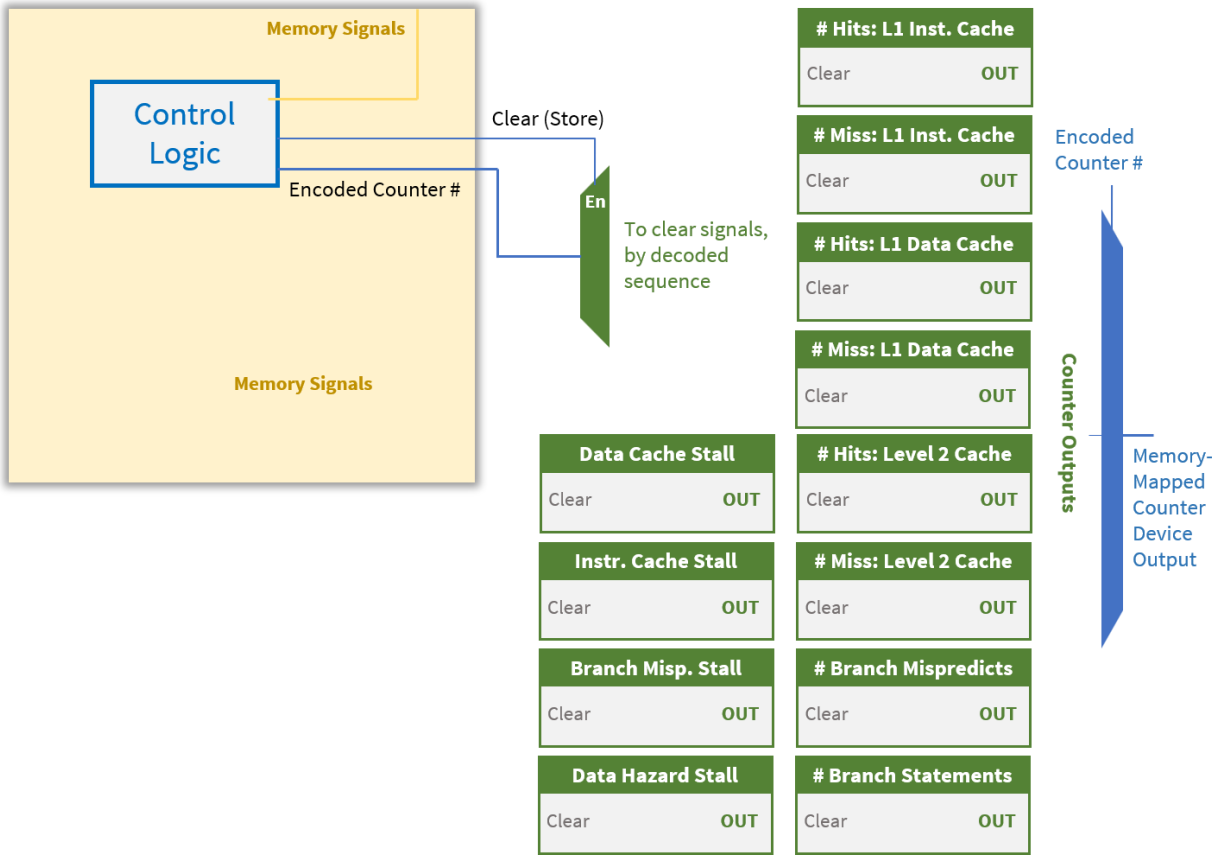
The following addresses are used to reference each counter:

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 branch mispredicts
0xB	Count of stalls caused by a detected data hazard (from MEM → WB)



As noted before, each performance counter is essentially a register supporting either accumulation or clear, depending on when it is accessed, or an event occurs. The above case shows a place where the number of hits and the number of misses is combined into one of these modules, but they can be thought of as distinct counters in this case. Note that both have a clear signal for each, which when high, goes to a multiplexer that takes a zero value for input to the register. We then update the register on the next time a miss is hit, so that the option can be updated, and the number of misses can be reset here. In all other cases, we accumulate the value that we already have, connecting to the +1 module that is visible here. Note, however, that we do not want to take a +1 per each cycle, and thus we must update this for each type that a miss or hit is present, and only once per distinct case. In this scenario, we trust that hit will stay high and go low whenever operation ceases, and assume the same for miss, so we have it clock based on hit or miss going from low to high signal (essentially, a clock pulse). Only on this pulse will the performance counter properly update by 1.

For this case, we note that a miss is the result of seeing an L2_write or L2_read issued by an L1 cache, indicating that there is data that is needed that the L1 (instruction or data) cache does not have. If, however, this was to attach to the L2 cache, we would instead look for pmem_write or pmem_read to determine that a miss occurred. In theory, one should be able to use only *read* signals for this miss logic, and therefore this is considered a design alternative that we will work through upon implementation.



The diagram above demonstrates the logic as placed for the memory-mapped I/O interaction with these performance counters. Our control logic will be modified as part of the additions to our pipeline, where alongside issuing memory signals, it can interpret a load or read based on the memory address of certain items and determine whether to send active signals to the data cache for reading. If the address is found to be below 0xC, the memory signals will be all set to 0. In this memory-mapped range, then, the clear signal will be set high if a SW/SB/SH instruction is interpreted with this memory address, no matter what source register is used. For both loads and stores, the counter number will be encoded in a four-bit sequence (from 0x0 to 0xB) to determine (via a decoder) which register to clear or read from.

The decoder is used with an enable signal, where enable is mapped to the presence of a store opcode being encountered. Should it be found, the line that corresponds to the encoded counter number goes high, and all the lines from the decoders are connected to the corresponding counter's *Clear* signals. Therefore, only one counter can ever be cleared at a time.

In terms of retrieving data, the multiplexer at the end is used for getting the output of the register that is used and will be placed as the result of execution in the MEM stage whenever there is not data being returned from the data cache. Depending on the encoded counter used for select bits, we take the requested counter's output (which are all mapped to an input port on the multi-bit multiplexer) and treat this as our memory-mapped counter device output to be placed into a register. In the case of overflow on any counter, the values shall return to 0, and values should be treated as unsigned.

Checkpoint Three: 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 most code writing and evaluation for this MP and led the write up on initial implementations of all items that we discussed and agreed on as a group. 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.

Compared to past checkpoints, Robert also coordinated more meetings to discuss changes to our agreed design that were necessary and lead these discussions to present problems and receive feedback to revise some inconsistencies in thought. His commentary was especially critical for the addition of another writeback stage, as well as a late change to remove the memory forwarding logic that was placed (and adding a larger critical path) in the MEM stage of the pipeline.

Outside of being the primary "code writer" for this step of the checkpoint, Robert also developed the first set of tests and modification of old tests for prior MPs that tracked where stalls were still being placed in the pipeline, and where data and branch hazards may occur. Working specifically to induce cases where there were false dependencies and *read-after-write* or *write-after-write* dependencies, he wrote tests that would test the effectiveness of interpreting which register was present at a certain location, whether it was being used in some instruction across the pipeline (with help from Robert Altman), and if it represented a hazard.

In this cycle, Robert also pushed the group to contribute to forwarding designs early in the process, so that issues with hazard detection could be spotted and resolved quickly.

Robert Altman

Robert Altman coordinated most large-scale discussion meetings, and worked most on the design of the forwarding, including the multiplexer logic and data hazard logic that transcended all stages. 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. He particularly emphasized which signals should be used or forwarded, and where latency may arise from adding combinational logic for hazard detection, cooperating with Robert Jin (and Yan Xu to an extent as well) especially on the data hazard detection and comparator logic used in the *Instruction Fetch* stage.

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).

Robert also oversaw development of more test cases and adaptation of previous tests, as well as the mechanism in which one should test to check both handling of branches and subsequent instructions with

dependencies, as well as evaluating no stalls where false dependencies existed instead (as this processor executes in order).

Finally, Robert constructed all control and flow diagrams, as well as all published physical resources throughout the conversation. (Robert Jin would typically provide annotated copies for review.) 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 Three. 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 was our primary reviewer and meeting coordinator and would often lead with guiding questions in our meetings that challenged our ideas of planned pipeline design. As done before, Yan would carefully limit the number of signals that we used and questioning the flow of various sets of instructions prior to implementation in SystemVerilog. In whiteboard planning and in overseeing code to implement our pipeline, as well as evaluating the simulation results that coalesced, Yan typically was one of the first to investigate results and offer any interpretation of what might be misbehaving in our pipeline design. As a result, Yan was mostly responsible for much of the debugging process outside of both Roberts' work in debugging as well and would offer feedback that started conversation when we followed up on already agreed upon design concepts.

Further, because of working alongside Robert Jin, Yan was 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.

Checkpoint Four: 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 performance counters and an eviction write buffer. The latter EWB will be a single line in length, and store 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. The former, performance counters, will be implemented within a larger module that receives memory-mapped I/O signals and produces a result multiplexed across all available counters (0x0 and 0xB).

Robert Jin and Robert Altman will start the work for the entire process, where Robert Altman will enumerate all the conditionals and logic that are needed for the performance counters first before implementation. These will be done according to the logic that was previously specified but will be presented as if implemented in SystemVerilog for Robert Jin to use in his implementation. From here, Robert Jin will create the modifications to control logic first, and Robert Altman will work with him to create tests that ensure the correct control signals are sent for memory-mapped addresses. Once the signals are properly tracked, we can go further into detail on building the performance counters themselves.

The individual counter logic will then be made, as a standard module named *counter* that Robert Altman will construct in large part. This will likely mimic much of the register design that was given in previous MPs, and thus accuracy checks before testing will be done by comparing against the code that is provided for these registers. Once this is complete, Robert Jin will continue and attach the new control signals created in the modified control signal generator module to a module consisting of twelve performance counters (one for each metric that was reviewed in the previous section). After this, a series of instructions will be run where the number of branches is known, to test the individual branch and branch mispredict counters. Further evaluation will then happen based on the number of expected misses and hits in a cache, with a specific series of tests written for this. As the individual test modules are prepared for each of the counters, Yan will mostly lead these individual "unit" tests of sorts, but Robert Jin and Robert Altman will largely contribute as well.

After this is completed, implementation of the eviction write buffer will start. Robert Jin will lead much of this development due to his work with caches, but Robert Altman will assist in producing and evaluating much of the implementation given in a quest to continue analysis on timing. Should longer critical paths be reported during this process, for example, it is Robert Altman's task here to try and reduce logic or prevent excessive latency. (It is also worth noting that Robert will be working on increasing the frequency altogether to a threshold of 120 MHz, if possible, while evaluating possible surrounding issues in memory organization component additions.) Robert Jin will produce the control logic as enumerated in the previous pages, while Robert Altman and Yan Xu will split the work on datapath logic and control logic as well. All of this will be implemented in full, with Yan leading

from the rest of this time on test writing that focuses on the state of the control logic for this eviction write buffer. Should no abnormalities arise and the control demonstrate transitions that are logically expected (no loops were found, etc.), then the entire team will collaborate on a sequence of tests that force multiple items to write back to physical memory, a single cache line to write back to memory, a case where only reads happen to physical memory (to test transparency and latency of the new module), and related tests.

Furthermore, although some preliminary testing was done already to check correctness of all instructions, we will go through further checks after cache implementation to ensure that our pipeline properly stalls and jumps within two cycles on branch predict *in the execute stage*. While four tests have been run on this thus far, some modified from previous MP1 and MP2-written tests, we have not also tested jumps as comprehensively and want to ensure that our static not taken behavior applies to these unconditional jumps properly as well. Most of this will require adaptation of our current tests, but there will likely be corner cases not yet discussed in depth that we must cover to properly evaluate this design.

As typically done, Yan and Robert (Altman) will meet and lead development of the full test suite used on our pipeline for data correctness. This is done first to ensure that the execution time improves under our new design's conditions versus those that were presented for Checkpoint 3. Since the write eviction buffer should allow for this to happen, we can run comprehensive tests that are meant for the caches themselves and compare the runtimes based on what ModelSim reports. Robert will specifically focus on this layer of tests, as he typically is responsible for timing analysis and performance metrics reported on our processor. Specifically, as well, he will test separated loads and stores that force cache misses and re-writes in a specific order, using test suites written for this MP and to check the LRU and miss protocol of the level two cache taken from MP2. In the meantime, Yan will construct tests with loads and stores that check similar functionality, but instead stress nearby loads and stores that will stress the data cache and ensure hits are found more often than misses. This will also require evaluating waveforms as well, and thus Yan will be mostly responsible again for evaluating our results and reporting concerns or successes back to the group at large. Alongside these responsibilities, Yan will also coordinate and schedule meetings formally within the group.

(continued on next page for logistics)

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 Thursday, November 15, 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 if other applications we are considering would be appropriate for measurement in this project. 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. However, we also need to consider scaling for larger eviction write buffers, as we find that a high level of performance will be achieved if we treat this buffer more like a cache with multiple sets. We do not plan to have multiple sets as of this time, again, but do wish to pursue it for the future here. 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 4:00 pm and hold video calls that complement our normal text-based chat starting Monday, November 19. By the time we complete another call on Wednesday, we would like to have at least the eviction write buffer functioning and properly implemented. A follow-up call will not be done until Friday, but this call expects that the entire design is complete with performance counters added, and the start of tests written to track the functionality of these tests. This should also be where the first evaluations of performance using ModelSim should take place for a conceptually implemented checkpoint three. All sessions after this will consist of planning for advanced design features and forwarding approaches, as well as generating test cases and scenarios, 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.