MP3 Checkpoint Two Report

ECE 411: Computer Organization and Design

Room 3032

Robert Altman / raltman2

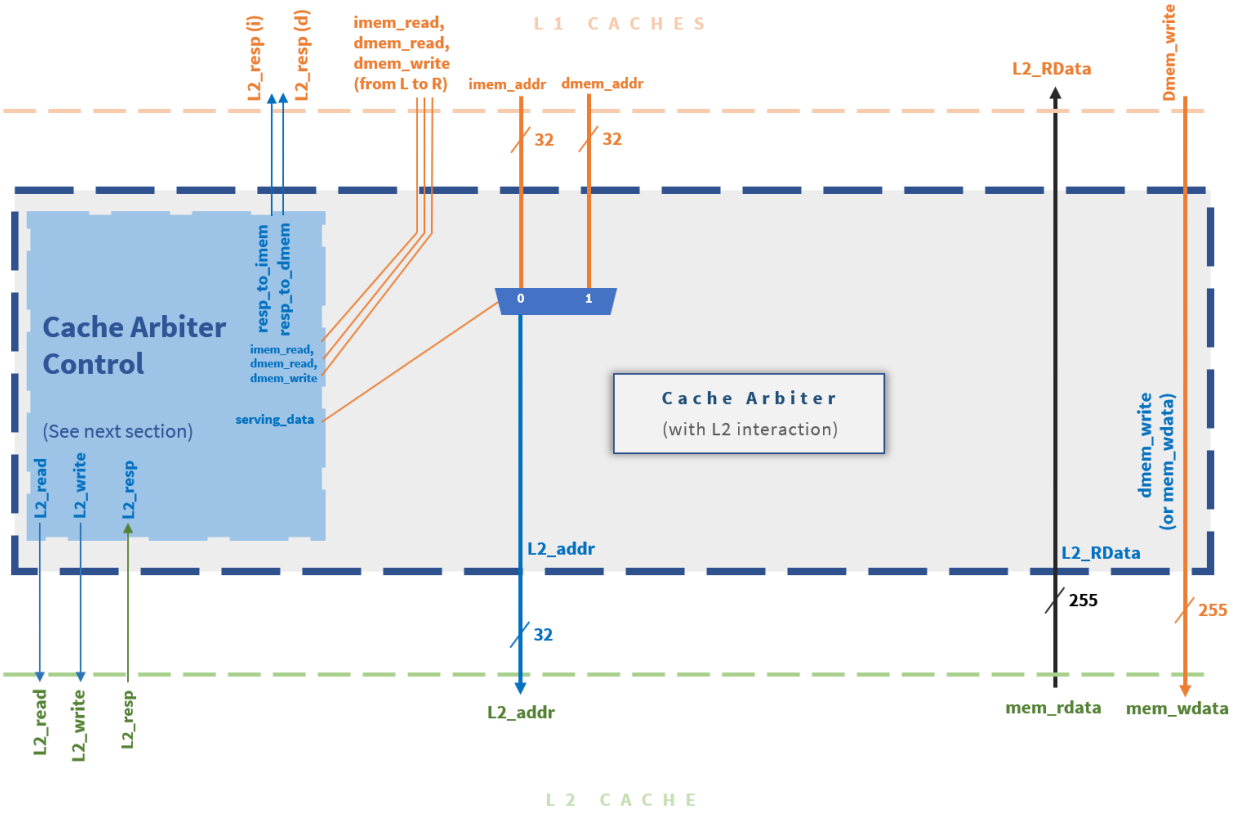
Robert Jin / naiyinj2

Yan Xu / yanxu2

Document Contents:

Checkpoint Two: Revised Arbiters and Caches

The arbiter implementation that was suggested in the previous checkpoint report was largely implemented as presented, although minor revisions have been added for consistency and performance optimization. The latter will be covered in the explanation of arbiter control changes, which is found in the next section of this report.



The only notable change to the arbiter design itself is nearly trivial and was discussed in the last report, albeit with an incorrect diagram. Namely, the mem_byte_enable has been completely removed from the cache arbiter, as the level one caches will be passing 256-bit data blocks to and from the level two cache. Here, stores do not depend on any specific offset to address from a data write – only the level one data cache will need to address this offset and modify a single byte, half word, or word to return to the processor. Furthermore, due to the size of the blocks being passed between caches, the offset bits that would be used alongside mem_byte_enable are all zeroed when giving a memory address to the level two cache (that is, there is no individual byte, half word, or word that is read from the 256-byte blocks, but instead just the 256-byte blocks themselves).

Thus, the updated version of the cache arbiter can be seen above. The only change is a mem_byte_enable signal that has been removed from the circuit here. All control changes will be specified in the next section, as noted earlier.

There were some critical differences, however, between our previously proposed cache designs and our currently implemented cache designs. Thus, we will break down each case by cache. Following these descriptions, an updated view of the datapaths for each cache is provided.

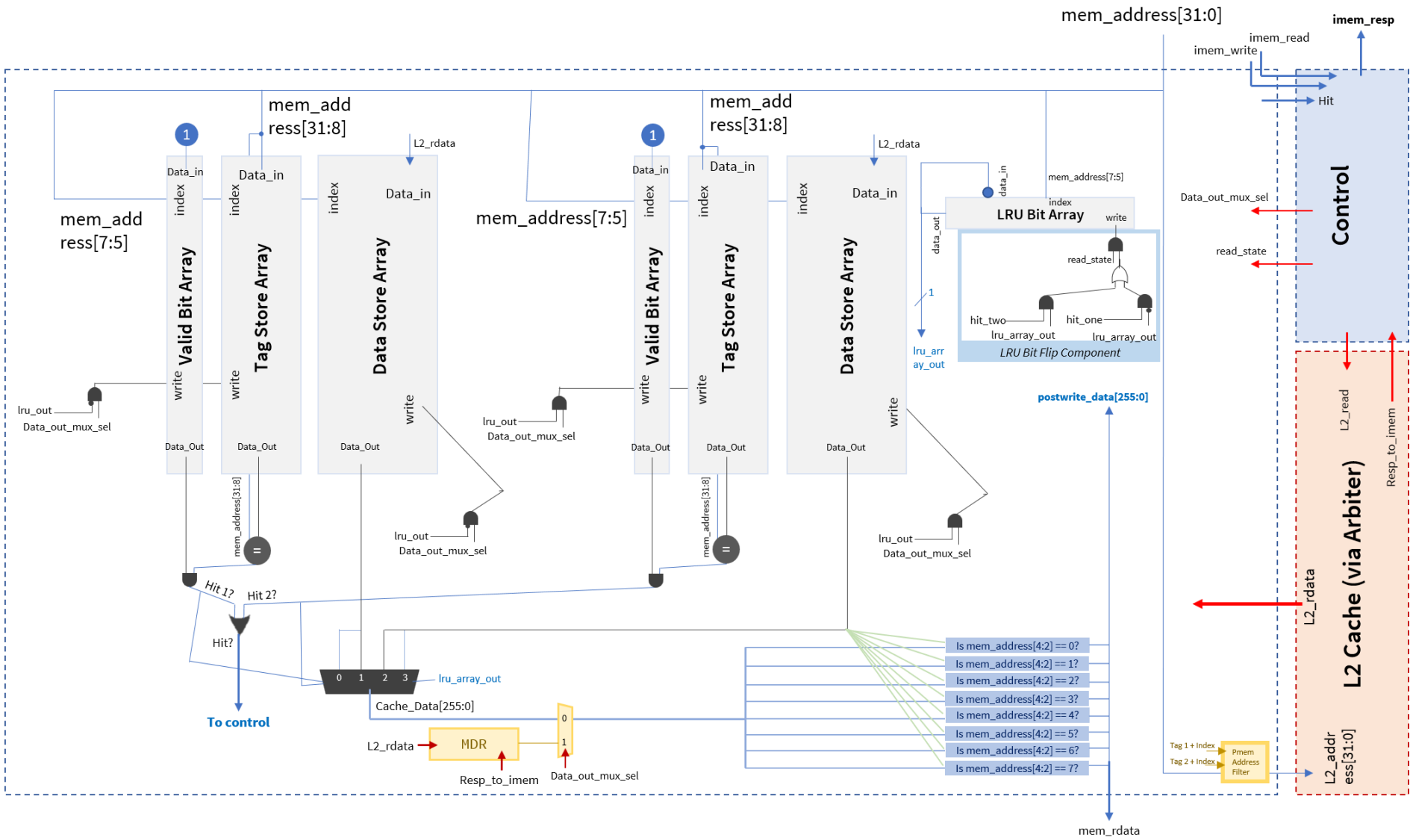
Starting with the level one instruction cache, we already discussed in the previous report that the use of any write signals could be removed from this specific cache. As a result, this would allow removal of the dirty bit arrays that were used in the cache. We did not, however, extend this to include removal of the *dbit* signal that is used by the control, which is otherwise used to determine if a write to memory must first happen prior to reading new data. The adjustment of removing writes would also allow for our physical memory address given to no longer require the tags of what is being read from in memory, since there is no place to write first prior to reading new data at a memory address. Ergo, these changes have been made in the most recent copy of the instruction cache. Further, we could simplify the logic on what is passed into the data array, purely taking the value from L2_rdata at any time as the data to store in an array as no external write data could be obtained for storage in the instruction cache.

Perhaps most importantly, however, there was a missing register that would hold the data in an intermediate register between the level two cache and the level one instruction cache. In the earlier MPs of the course, our RISC-V processor design held a memory data register that intercepted the data taken from memory reads and placed it in a location such that it could not be read right away. Our top-level design of the pipeline still will ensure a clock cycle is added for the memory to be stable, but the same interface is not mimicked between the caches here.

The register was added for communication between the level two and level one caches. Reading from the data array is slow, however, and writing to the data array is also slow here. Therefore, only one of these operations should happen within a single clock cycle. Should there be a hit in the level two cache, then, but a miss in the level one, there may have been an issue where data to write and read was both constrained to a single clock cycle. Therefore, we want to separate these into separate cycles such that we get a stable value from the L2 cache, focus on reads for the first cycle, and complete a write in the second cycle. It is worth noting that this step was added mostly for changing the frequency of the design here.

With this addition, then, we also must adjust the logic that corresponds to the data that is read. Since we can return this data immediately upon its availability and setup to the register, we simply take the output of the memory data register and place it into a multiplexer with data from the arrays themselves. Whenever there is a writeback signal that is indicated by the control logic of the cache, we take the new data that was provided by the L2 cache here.

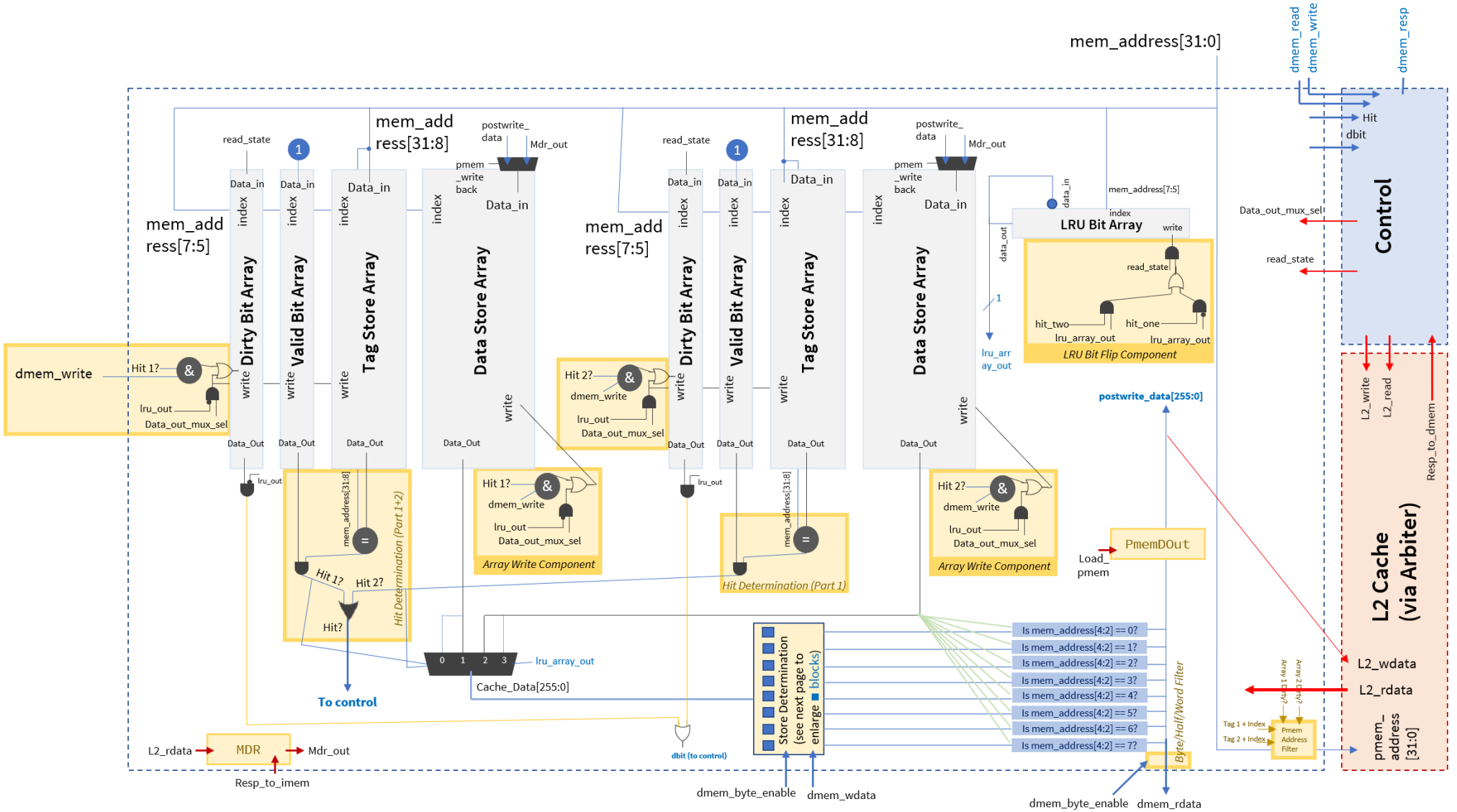
The updated hierarchy, thus, for the level one instruction cache is provided on the next page.


mem_address[31:0]
imem_resp
imem_read
imem_write
Hit
Control
Data_out_mux_sel
read_state
mem_address[7:5]
mem_address[31:8]
L2_rdata
Data_in
index
write
Valid Bit Array
Tag Store Array
Data Store Array
Data_Out
lru_out
Data_out_mux_sel
LRU Bit Array
data_in
data_out
lru_array_out
hit_two
hit_one
read_state
LRU Bit Flip Component
postwrite_data[255:0]
Hit 1?
Hit 2?
Hit?
To control
lru_array_out
Cache_Data[255:0]
MDR
Resp_to_imem
Data_out_mux_sel
Is mem_address[4:2] == 0?
Is mem_address[4:2] == 1?
Is mem_address[4:2] == 2?
Is mem_address[4:2] == 3?
Is mem_address[4:2] == 4?
Is mem_address[4:2] == 5?
Is mem_address[4:2] == 6?
Is mem_address[4:2] == 7?
mem_rdata
L2 Cache (via Arbiter)
L2_read
Resp_to_imem
L2_rdata
L2_address[31:0]
Tag 1 + Index
Tag 2 + Index
Pmem Address Filter

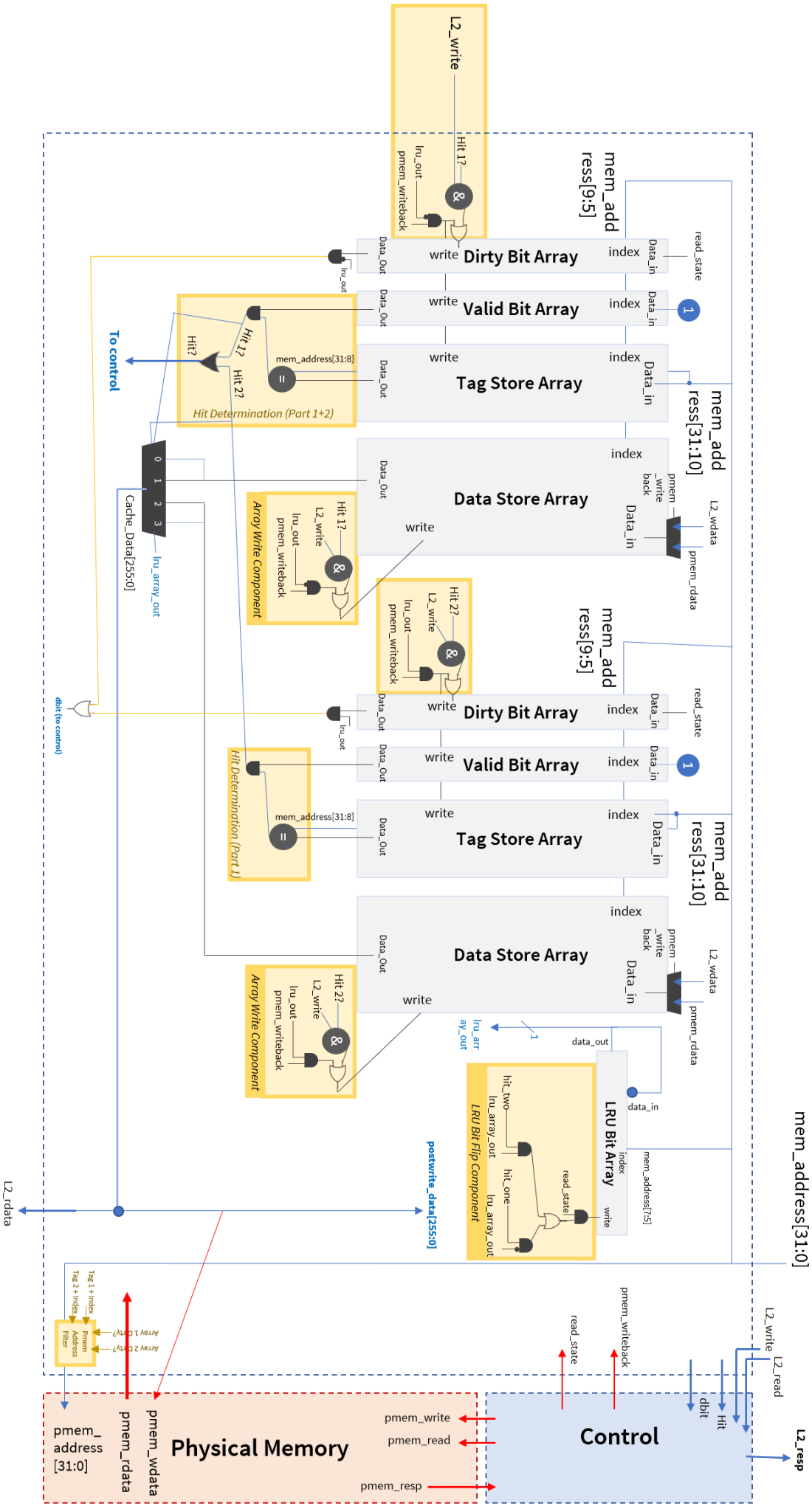
A similar edit for adding this register was placed with the level one data cache as well, for the same reasons as specified with the level one instruction cache. However, a second register was needed with a load signal for data that would interface with our level two cache as write data, to ensure that this data is also received with a single cycle of delay only after the data has been read from the level one array (if it needs to be evicted). This ensures that the level two cache does not try to read prematurely here before potentially doing a write, since the logic connecting the two caches themselves with all signals is combinational (despite the control logic being clocked here). Additionally, for this cache, we could again directly interface with the level two cache's read data through a memory data register, where the output of this is what is passed into the multiplexer for the data arrays as seen on the previous iterations of the cache. For reference, these additions have been presented with the rest of the datapath on the next page.

This leaves emendations that were given to the Level Two cache, outside those that were already discussed in the previous report. Previous discussions included a removal of mem_byte_enable, as well as a removal of all offset logic and a direct reference to write data being input back into the data array. However, an additional change was added to this cache, in that we can skip over doing an extra read cycle entirely if there is a write case that is being used. Since the data to write is already 256 bits, reading back memory from physical memory after a writeback or when a writeback is not necessary would become redundant. That is, the data from memory would be read, only to be completely overwritten by the write data that is presented by the cache here. Therefore, we can remove logic that determines which data is presented to the multiplexer attached to data input of the data arrays, and simply have the multiplexer take in mem_wdata or pmem_rdata, depending on the operation. We do this with modification of the control logic, such that whenever L2_write is presented to the control, the control sequence will become idle → writeback → read (where write happens to the data array, and doubles as the idle state. In other cases, where a read is not used but writeback still occurs, idle → writeback → readback will be used, indicating an extra state that is taken here.

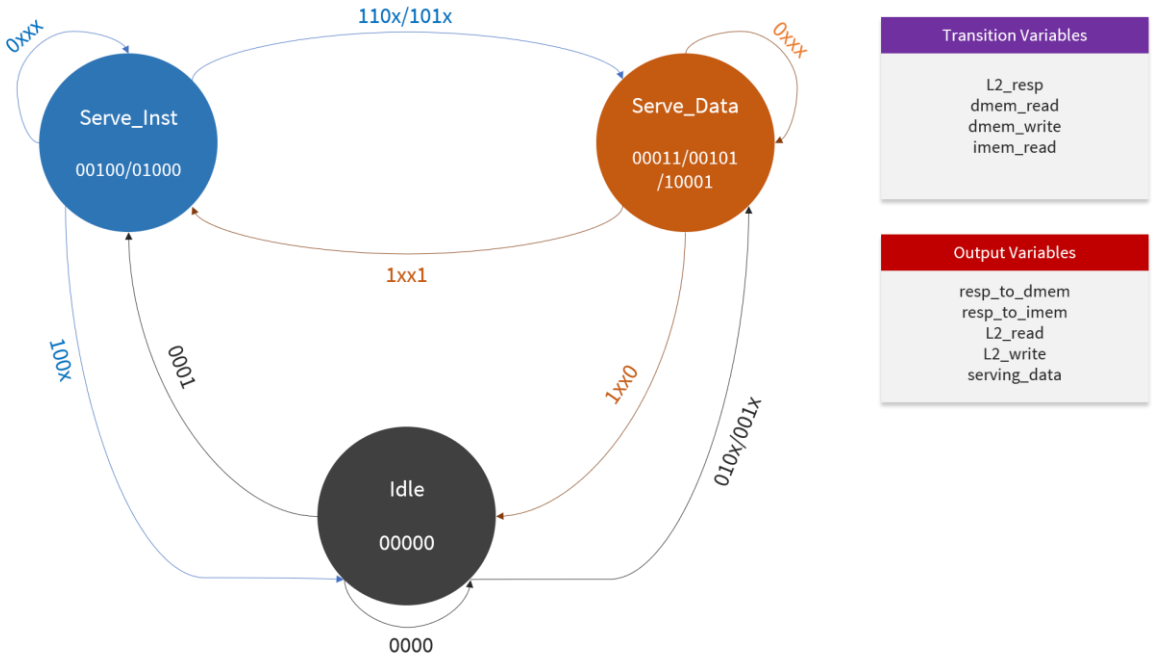
The two designs that follow these descripts correspond to the level one data cache and the level two cache, respectively.


mem_address[31:0]
dmem_read
dmem_write
dmem_resp
Hit
dbit
Control
Data_out_mux_sel
read_state
L2_write
L2_read
Resp_to_dmem
L2 Cache (via Arbiter)
L2_wdata
L2_rdata
pmem_address[31:0]
mem_address[7:5]
mem_address[31:8]
read_state
postwrite_data
Mdr_out
pmem_write back
Data_in
index
write
Dirty Bit Array
Valid Bit Array
Tag Store Array
Data Store Array
LRU Bit Array
LRU Bit Flip Component
lru_array_out
hit_two
hit_one
dmem_write
Hit 1?
Hit 2?
lru_out
Data_out_mux_sel
Array Write Component
Hit Determination (Part 1)
Hit Determination (Part 1+2)
mem_address[31:8]
Hit?
To control
Cache_Data[255:0]
Store Determination (see next page to enlarge ■ blocks)
Is mem_address[4:2] == 0?
Is mem_address[4:2] == 1?
Is mem_address[4:2] == 2?
Is mem_address[4:2] == 3?
Is mem_address[4:2] == 4?
Is mem_address[4:2] == 5?
Is mem_address[4:2] == 6?
Is mem_address[4:2] == 7?
Byte/Half/Word Filter
postwrite_data[255:0]
PmemDOut
Load_pmem
MDR
L2_rdata
Mdr_out
Resp_to_imem
dbit (to control)
dmem_byte_enable
dmem_wdata
dmem_rdata
Pmem Address Filter
Tag 1 + Index
Tag 2 + Index

1.


L2_write
Hit 1?
lru_out
pmem_writeback
mem_address[31:8]
write
Dirty Bit Array
Valid Bit Array
Tag Store Array
Data Store Array
index
Data_in
Data_Out
read_state
To control
Hit?
Hit 2?
Hit Determination (Part 1+2)
Array Write Component
Hit Determination (Part 1)
Cache_Data[255:0]
lru_array_out
dbit (to control)
L2_wdata
pmem_rdata
pmem_write_back
mem_address[31:10]
mem_address[9:5]
mem_address[7:5]
mem_address[31:0]
data_out
data_in
LRU Bit Array
LRU Bit Flip Component
hit_two
hit_one
postwrite_data[255:0]
L2_rdata
Tag 1 + Index
Tag 2 + Index
Pmem Address Filter
Array 1 Data?
Array 2 Data?
pmem_address [31:0]
pmem_rdata
pmem_wdata
Physical Memory
pmem_write
pmem_read
pmem_resp
Control
L2_read
dbit
Hit
L2_resp

Checkpoint Two: Revised Cache Arbiter Control



The control unit for the cache arbiter is one that determines which cache, out of the fetch and instruction level one caches, to serve. It may either be serving none of them, only the instruction cache, or only the data cache, but never both at the same time as they connect to a single point of reference (either physical memory for the purposes of checkpoint two, or the Level Two cache for the purposes of the entire planned pipeline). In this control layout, we assume that our cache arbiter is properly connected to the Level Two cache as described in the previous section.

While not much has changed between the initial proposed implementation of the cache arbiter control and the version that is currently in use, there has been a critical modification to the transition sequence that changes priority of one level one cache being served over another. That is, while before the instruction cache would be served before the data cache when both signals were high (and the current state was Idle), the current implementation now serves data before instruction. This is seen with the transitions of 0001 to serve_inst from Idle, which deliberately checks if the data memory cache signals are low before evaluating whether imem_read is high (and instruction needs to read). Note, however, that the imem_read bit is ignored in the transition to serve_data, where if dmem_read or dmem_write is enabled and high, the transition is taken. This places hierarchy, as it essentially does a check for these signals first before looking at instruction cache's request for memory service and serves data first if a request from the cache indeed was indicated.

It may, however, be unclear why such a change has occurred. In the previous iteration of this document, we stressed that the instruction cache is favored to accommodate for performance, especially in the case where there is no dependency immediately after in the next instruction that would cause a miss in the instruction L1 cache. That is, we wanted this instruction (and ideally, subsequent instructions that follow it) to still be able to execute and not have to wait for data service from some instruction before it.

This, however, leads to a problem of still loading new instructions into the pipeline while an instruction that is stalling the pipeline is not served. Not only could this lead to hazards if the functions now are found to continue through the pipeline again, but the instruction whose data is waiting is now delayed for even further time prior to completing. (That is, even if we continued to stall the pipeline, our performance is in truth worse since we would load an instruction that cannot execute yet until data is returned).

Therefore, we instead now serve data caches first if a request comes in simultaneously with the instruction cache (a scenario that can only be interpreted from the Idle state). We place this trust in data first, again, to ensure that the instructions delaying all others in the pipeline can be serviced to *stop* stalling this pipeline and continue execution as fast as possible here. Even if there was an instruction cache miss after we would have returned data, other instructions would at least be able to continue that were waiting on this load/store instruction causing a data cache miss, thus allowing for better temporary throughput during those times.

Checkpoint Two Progress Report

The proposed goal, as given in the original RISC-V documentation, was to finalize support for all instructions in the RISC-V instruction set (minus instructions such as FENCE), as well as add a cache arbiter with interaction between two level one caches (data and memory) and physical memory. The latter ensured the start of a memory hierarchy being formed, in which both caches may request data to be read and written back from a larger (albeit slower) level of memory. The pipeline this time included support for instructions like jumps (including JSLR), arithmetic shifts (such as SRA), and smaller data type loads (LB/LBU/LH/LHU) along with their store equivalents (SB/SH). By closely following our design proposed in the previous stage of this MP, however, along with a close reference to both works completed in MP1 and MP2 of ECE 411, we have implemented and run tests on *all* instructions, and expanded our coverage through further test case application. Most importantly in this cycle, however, we have monitored the behavior of our cache, the amount of cycles taken in data reads and misses in different levels of the caches, and subsequent stalling in the main pipeline.

Due to our development and confirmation via testing, we demonstrate multiple functions of the processor. 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. 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.

Multiple goals were accomplished in parallel here, but our first efforts had to turn to the pipeline itself before addressing a new memory scheme (namely, one that involved slower, physical memory instead of the “magic” memory abstracted in the previous checkpoint). We have already addressed how to deal with stalls from the instruction memory, in that no-ops were passed. However, we never had a situation in which this was explicitly necessary, and therefore had to confirm that our multiplexer would indeed take in the *no-op* instruction whenever our function was not ready. The team worked together on this first step, with Robert Jin leading most of the implementation work, where physical memory was directly connected to the instruction memory and the signals were tested for a response. (This required a temporary re-write in which the only items requesting read/write to memory were components from the *Instruction Fetch* stage.) Thankfully, there was no issue found when testing this individually, and an appropriate number of *no-op*s were passed that did not modify any data. To test this, Robert Jin (and Robert Altman, in proposing the design test) could simply read the very beginning of the execution cycle, as each program loaded into memory for execution would have an instruction cache miss immediately.

Once this was completed, the concept of stalling across the pipeline had to be confirmed as well for the data cache. This was a design issue noted as needing completion in the previous document, since it was assumed in the original implementation that memory would respond fast enough to prevent any extra operations from happening. (Our tests also used a sufficient number of *no-ops*, beyond just

a threshold of three, to prevent data hazards as well. This essentially allowed for an extra cycle of memory read to happen with no instruction executing in the background.) While all in the group did not question that stalling was necessary, and even agreed that it should disable the PC from incrementing, the mechanism for how it should be determined remained unclear. After a few iterations and polling each member in the group for their perspective, the team agreed upon using combinational logic that determined the stall as pipeline_enable from all data memory cache signals. This decision was made based on the concept of the data cache responding, where if a request indeed was given to read or write to the data cache (dmem_write or dmem_read as 1), but there was no response yet (dmem_resp as 0), then data was not returned to the instruction and stalling needed to occur. This led to simple two-level logic, with an OR on dmem_write and dmem_read, fed into an AND with an inverted dmem_resp signal. Robert Altman checked that this would not be a bottleneck given the two-level nature of the design, while Robert Jin wrote the logic itself. Yan Xu was responsible for writing tests like the one proposed earlier that tested the same concept, except this time restricting the number of *no*-ops and using physical memory with instruction through a secondary port.

In doing so, we realized that it was not just the PC that needed to be stalled. In fact, every register needed to be stalled in the pipeline after this step, and thus the load signal needed to be modified for each one. Thankfully, however, we already had this pipeline_enable signal in place to indicate when execution should happen, which was low whenever stalling occurred. By just feeding this signal into each stage's latches themselves, the latches would simply retain the data they had during a stall situation and not propagate any results through the pipeline. Robert Jin was mostly responsible for this change here.

Now that we had deemed the circuit ready to deal with the latency issues of physical memory, it was time to add the memory hierarchy that it along with the two levels of cache here. (It was never planned to only use physical memory and level one caches, as we felt designing the level two cache to save on performance was a better option to complete as soon as possible). To do this, we decided to split duties between Robert Altman and Robert Jin with Yan Xu to work on the arbiter setup and cache setup, respectively.

Starting with the arbiter, Robert Altman created the initial hierarchy structure that was used throughout the project as a location where the arbiter and its main routing mechanisms could be stored. This was a module that represented the memory hierarchy altogether, which while referencing the arbiter control through use of a separate SystemVerilog module, would simply route pins immediately between cache modules as this was most of the arbiter's design. To do this, however, caches needed to be in place to sustain the arbiter construction. Therefore, temporary and unmodified versions of the MP2 caches were placed for both data and instruction level one caches, with the rest of the signals simply given to physical memory from the arbiter to at least check that routing and response signals were being handled in the correct order. Once the mappings had been completed, the arbiter control was next, in which Robert designed much of the structure based on checks for memory signal response. To enforce the hierarchy of serving the data cache before the instruction cache, as revised according to the previous section, conditional statements were written to check the signals of dmem_read and dmem_write first before even evaluating the imem_read signal input to the control. This would confirm the need for data to write first while reducing the

number of pins that would be needed for each comparison. Only upon verifying that there was no write or read coming from data could the imem_read be verified in the idle state.

As for the logic determining output signals themselves, all signals could be nearly directly mapped based on the values of read and write that were given to the control. However, these still needed to be separated depending on the state – that is, a write request to the level two cache (or physical memory, whichever was being tested) could only be written when data write was high, where read was issued only when instruction cache had read high or data cache labelled read as requested. In the serve_data instruction, since we knew only one of these signals could be high at a time, we routed the dmem_read and dmem_write signals directly to L2_read and L2_write outputs of the arbiter control, avoiding excessive combinational logic that may have been used otherwise. Similarly, the signal for imem_read was routed directly to L2_read in the serve_inst state, with L2_write wired to 0 since this was not a supported operation (nothing could ever be written back from the instruction cache here).

However, it was this very routing mechanism that originally caused a problem. The routing was still at first conditional on whether a response has been given from the level two cache. Since this signal was generated with combinational logic in our original cache design, it could go high in the middle of operation and change whether a read signal (for example) was set to 0 (the default state) or mapped to the read signal being input into the control. This would induce a loop condition, which Robert Jin discovered after his implementation of caches (a process discussed in the next paragraph). Therefore, the signal could not hide behind the combinational logic, and only the response to the instruction cache or data cache itself could be determined based on the L2_resp signal that was being input here. Otherwise, the cache would not receive the signal in time and would have its read signal marked as if no operation was happening at all. Once this issue was resolved such that signals were routed without any combinational logic interference, correct behavior was again seen through a sequence of MP2 modified tests and original test cases written that checked known misses between chunks of data (as well as instruction cache via branches utilized).

To confirm this behavior fully, this required Robert Jin's implementation and Yan Xu's testing of the level one and level two caches themselves. Robert started this process by simply the MP2 cache that had already been written for the level one caches, removing the dirty bit array as specified in the previous MP as well as any write signals for the instruction cache itself. There was an issue in doing this, however, discovered while writing the caches that some form of delay was needed to ensure correct address and data configuration between the rest of the memory hierarchy was truly represented here. That is, since data that was written back from the cache could not just be immediately written but required some sort of delay before the cache could write back, an interface was needed such that a register file would hold the value prior to proceeding operation in the cache. (Thankfully, this was also discovered and aided by the direction of our TA, Kenny.) This also induced changes that needed to be made through the control, which Robert carefully walked through during the series of meetings that took place over the course of the week to ensure his understanding aligned with the team's before implementing any functionality. To test whether functionality of the cache still worked at all, even with the new cycle, Robert Jin and Yan both ran test cases against those they had constructed for MP2 caches (except for the instruction cache, where only a series of reads were used instead) to ensure a difference in cycles was found and writeback or reading happened properly.

After the caches were completed, the temporary caches that were in place from Robert Altman's implementation of the arbiter were removed and the new caches were placed into the memory hierarchy module. This was then tested with the same series of tests that were previously issued, and no new behavior seemed to be present as a result of completing this operation (since physical memory was already interfaced with in the original designing of the level one caches, thus inducing similar latency). Any difference in delay was caused by the data cache being serviced before the instruction cache in the pipeline.

Once this was done, the remaining step was to add the level two cache and conclude the work present in this memory hierarchy. This, in fact, is the step that yielded (upon running the same tests on the same level one caches) the cache arbiter control routing issue discussed on the previous page. However, a fundamental issue in responses was also found in the MP2 cache being adapted for Level Two as well. Starting with the latter, it was noticed in the arbiter that there was simply never any routing of data to the level two cache to begin with, as the idle state of L2 would return mem_resp as high if it was idle. Therefore, we had to create a stage such that mem_resp was only high once a hit was found on the read state, keeping this signal high for only one cycle but low on all others (so that a read or write signal may properly be passed through from the cache arbiter control to the level two cache itself). This emendation did not need to happen to the level one caches, as the signal for memory response was only used by the processor to verify that data was received a cycle after read or write was set to high (which, in this time, would have already changed the signal given back by the mem_resp output). Once both issues were resolved, and correctness was confirmed with another short test sequence of instructions sequentially loading and writing stores and bytes, the cache arbiter and memory hierarchy altogether was deemed working and implemented.

On a final note, we had expressed that only a small number of tests were run to confirm functionality of the "full" instruction set, that is the one including instructions such as jumps or variably stored words. Interestingly, writing and running more MP2-based tests for this (that heavily utilized instructions like JALR or SB, for example) demonstrated one more revision that was noted earlier – mem_byte_enable did not need to be passed between the level one and level two caches, as the block size was the same. This note was made but the diagram never corrected in the previous document, but the signal was simply removed in the design after discovering extra latency trying to interpret the data at the second level of caches (when there was no need, since we would write 256 bits or read them already). This also led to an optimization of skipping over reading back data from the cache if there was a write command issued to L2, as we could just replace the entire block (after writing back to physical memory, if necessary, what was present in the to-be-replaced block of L2) in the L2 cache. While this required another state altogether in the control logic, it did not induce any extra delay since either a sequence of two or three states would at most be accessed in the caches with this logic here.

Hazard Detection and Forwarding Design Overview

For our next checkpoint, we are going to implement hazard detection and forwarding across three different types of stage transitions: EX → EX, MEM → EX, MEM → MEM, and WB → WB. Starting by evaluating the conditionals that will be needed in our project, after the register file in the ID stage gets the loading signal (WB_ctrl.load_regfile), it needs one more cycle to actually update the value in destination register. In this circumstance, we are conceiving a new data forward as well called WB → EX. Just as with other normal hazard correction via forwarding, we will directly forward the value in the previous destination register to the input port of ALU on the multiplexers placed in the execute stage, filtering the source register values that are being used for execution of an instruction. This particular instance will be used if the previous destination register and the current source register are the same. With regards to the priority of the four hazards we have here, it is determined by the pipeline order (IF, ID, EXE, MEM, WB). We need two 4-to-1 multiplexers to make forwarding happen in this execution stage, which are the multiplexers referenced earlier in this paragraph. That is, then, one multiplexer corresponds to filtering the value of rs1, and the other one corresponds to rs2. In each multiplexer, Pin 0 should keep the original value as interpreted from the register file, in case no forwarding is truly necessary. All other pins, however, should be the result of some sort of forwarding in the order defined above. That is, Pin 1 should get the previous result of the execute stage, with checks on the *br_en*, *alu*, *pc*, and *IR* values. Pin 2 of each multiplexer should get the previous result of a memory stage, indicating a result from either two instructions ago, or an instruction ago that has been processed and delayed the pipeline while memory calculation occurred. The final pin, pin 3, should get the previous result of the writeback stage. For the result of the EX stage and WB stage, we choose the result coming out of the regfilemux, because each input of regfilemux can be needed for forwarding.

There is one thing still needed to accomplish this, however, which is a stall between sequences of loads and non-memory, register store operations (like *add*). When the previous instruction, for instance, is *lw* and second instruction is *add*, a delay must occur as the value loaded known to the pipeline comes in the cycle following EX (and is not computed during the EX stage, thus). For handling forwarding in this case, we can check if an LW is present at the ID stage when a second instruction comes in and is interpreted at the IF stage. Comparing the opcode for the load, we also check if the destination register was not zero and matches any one of the rs1 or rs2 values of the incoming instruction, stalling with a no-op if so. The result will be stored with the signal *d_h_sel*, which functions as a selector for choosing between the next instruction or a *no-op*. At the same time, if the *no-op* is chosen, which means pc will not update in this time, we need to use combinational AND logic combing the imem_resp, pipeline_enable and d_h_sel. The result of the and combination will determine whether we should enable the PC register or not.

The above is meant to overview the operation of this forwarding and hazard detection. Further detail about its implementation and interpretation in the pipeline follows in the next section.

Proposed Datapath Emendations for Forwarding

Prior to beginning, the full datapath and all emendations discussed here can be found at the end of the section. Highlighted blocks indicate where new logic is being added to handle forwarding properly. Each highlighted block will be presented in sequence through this writing.

There are multiple emendations across multiple stages that allow for executing with correct data, even if the data has not been committed to a register file for now. Since we are avoiding the use of a register file that uses the negative edge to update and the positive edge to read, as this may constrain the maximum frequency at which our processor can run (due to combinational logic and placement delay), there are multiple accommodations that are made taking available data committed by a former stage. We will skip over the hazard detection that occurs in the instruction fetch for now and elaborate on this when discussing MEM stage to EX stage forwarding to determine what is being done.

The first case of forwarding that we discussed in the previous section, EX → EX forwarding, is used when a computation is ready in an operation through the previous instruction, and the ALU contains the results of this operation. In such a case, we would want to get the value that was just produced in the last cycle from the ALU, which is contained in the ALU latch from the EX/MEM latch sequence provided on our datapath. Therefore, if we route this data back to our ALU for computation again, this time as a source register, we accomplish forwarding.

However, note that this is not the only case in which a value can be outputted. For example, the value in *br_en* could be written to a register instead, which would have been calculated by this stage but is not captured in the ALU latch. (Such a circumstance happens with the *sli* instruction.) To address this, we can use a multiplexer that gets the correct computation from the execute stage as used in the writeback stage, with regfilemux_sel as the selection bits. For the memory line, which would be the only data line unavailable after an execute stage, we simply fill this line with don't cares, as the control logic will not take from this value if MEM → EX forwarding was to be used instead. As a result, an additional multiplexer can be seen linking to the first (zero indexed) signal on two multiplexers. These two multiplexers are used for filtering the RS1 and RS2 value outputs, compared to register file and two other signals yet to discuss.

A second type of potential hazard that is present here is when data is computed and found from memory, but it is to be used in the execute stage and has not been properly written back yet. This is where the case of forwarding data from MEM to EX is needed, or in other words, where the processor uses MEM → EX forwarding. As indicated by the name of this result, we are looking for data that will be placed and has been readied from the previous load that would have been available. This goes to a latch, from our datapath, that is labeled M3 here – named such as it is the third input to the ALU in the writeback stage, and M for its relation to a value from memory. We want this value to be available for consideration as RS1 or RS2 if the destination register of the load matches that of either RS1 or RS2, and therefore it is also fed as an input on line 2 of the aforementioned multiplexers determining RS1_out and RS2_out.

There is a caveat to doing this, however, in that the process at least needs an extra cycle to load, and the stage's latch is not simply reflecting the value one operation ago, but *two* operations ago. That is, in the previous EX → EX case, the previous operation executing before the current needing a value

forwarded (for correctness) was just that, immediately before our current instruction. However, the value in the latch we receive from the memory stage would be available for execute to use at minimum two instructions later. Therefore, there must be some sort of stall for a single cycle to induce (outside of the stalls that the data cache places on the circuit). This requires extra logic that must be placed then in the instruction fetch stage, before the IR can even be loaded, so that a no-op can be passed through the pipeline for a cycle instead.

Therefore, we introduce another multiplexer with a signal named d_h_sel, short for *data hazard selector*. Whenever this selector is high, we will pass through a no-op through the circuit instead of the next instruction that was found from memory. Notice that this is not a stall dependent on waiting for a value that is available, but explicitly blocks the next instruction altogether (overwriting the value instead of just providing one while waiting). This signal, however, needs to be computed for a single cycle alone, and thus is dependent on the current value that is found in the instruction register. Effectively, we check if there is some sort of instruction that does not store to x0 (as a destination register) as the next instruction first. (This satisfies an edge case that we note, in that we should not forward if the value is simply going to a register that is always NULL anyway.) If not, we then check if a load instruction was placed in the previous instruction (thus, present in the IR latch from IF to ID), and if the register to be loaded matches one of the source registers from the current instruction fetched from memory. If indeed, there is a match between one or more of the source registers, then d_h_sel is set to high and the no-op is taken. In every other case, however, we simply pass on the instruction.

This is not the only location, however, in which d_h_sel needs to be used. If we are skipping interpretation of an instruction that was valid, we do not want to skip the instruction entirely when the one-cycle stall has been placed into the pipeline. Therefore, our PC's enable signal that would stall when pipeline_enable was low must also consider this d_h_sel. However, in this case, it is when our data hazard selector signal is high that we are to disable, so the signal is inverted before placing into the AND gate controlling the enable here.

Now that this case is handled, there are now two more forwarding cases still left to consider. To finish review of the signals that are present in the forwarding multiplexers covered thus far in the execute stage, we note that there is an "input" signal that comes from wb. This is the same signal that would be available from the writeback stage as an input into the register file, instead given a stage early to the execute stage to handle WB → EX forwarding. Previously, we did not need to induce a delay for this sequence since this value is not clocked behind a latch (instead determined by combinational logic) and can simply be routed to execute in time. However, say the current instruction that is available has entered the execute stage, and the result of WB has been written to the register file. This write would have still happened a clock cycle too late, and thus we need to get the value from a latch holding the final output after WB. Therefore, a latch has been added to do this direct mapping. Note that this does not require any additional stalling however. If the value was one that was computed in the previous instruction yet being written to the register file, it would have already been forwarded through the EX → EX forwarded signal or MEM → EX forwarded signal given earlier here.

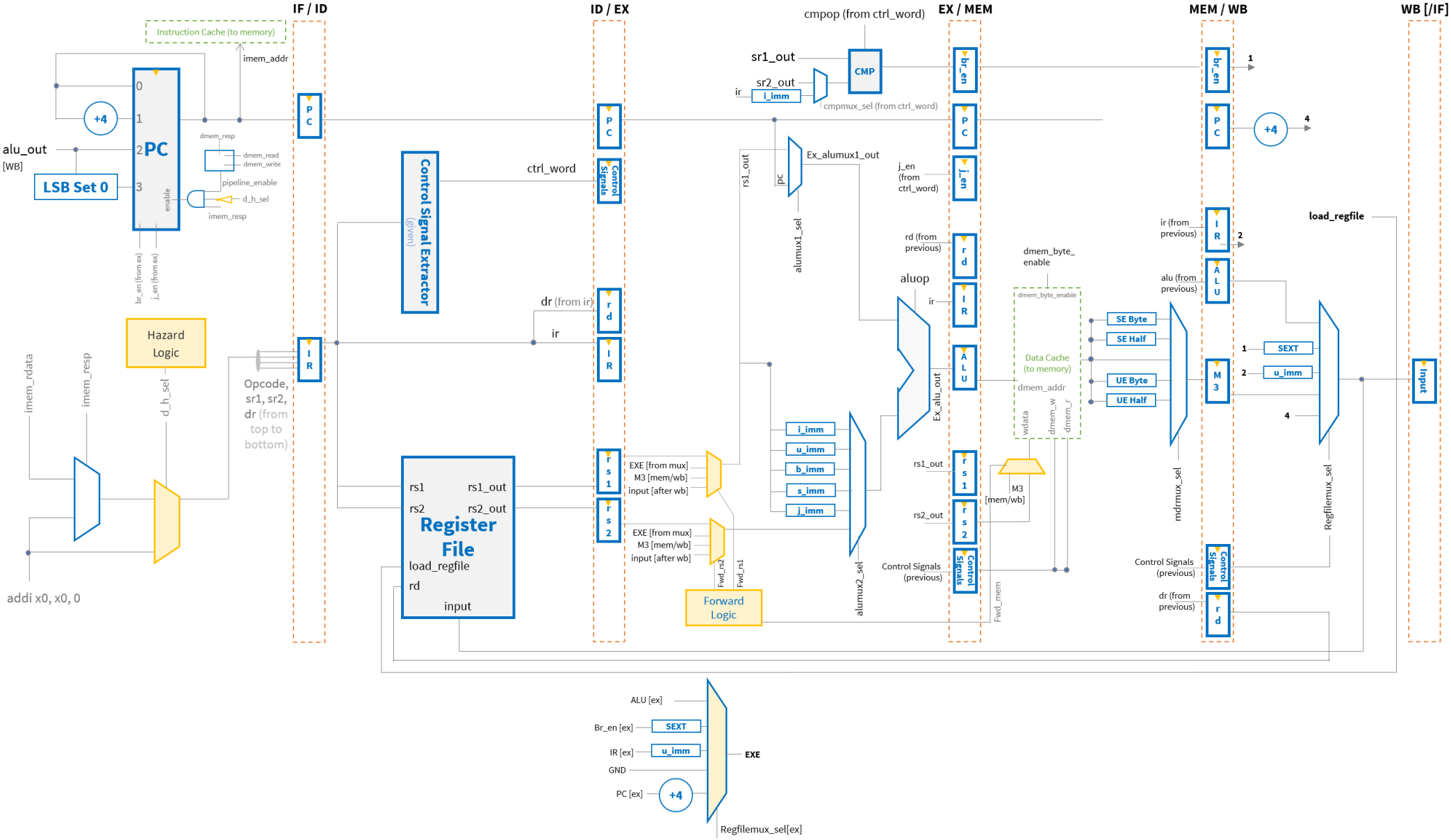
The rest of the multiplexer takes the regular register file output for RS1 and RS2, for whenever one of these forwarding cases is not considered. All of this is controlled by a signal *fwd_rs1* or *fwd_rs2*, respective to the signals RS1 and RS2. Echoing the logic enumerated in the previous section, we will

take the regular value whenever one of the last three instructions has not used the same register as its destination register. Three cycles are used due to the delay that is induced for the EX, MEM, and WB stages, as well as the instructions that are present and executing from those stages. If, however, RS1 matches the destination register passed through each stage-to-stage latch array, then there is a conflict to consider. Namely, if we find that the destination register being used after writeback is in RS1, we take the third value given to the multiplexer as the RS1 for computation instead. Similarly, if the RD (destination register) provided in the MEM/EX latch matches, we take option 2 of the multiplexer. Finally, if RD and RS1 match based on the RD from the EX/MEM latch array, then the first (zero-indexed) option is taken as RS1. The same operations are found to compute the select signals for forward_rs2, with the register in comparison RS2 instead of RS1.

There is one final forwarding option that has not been covered, however, where data that has been computed needs to be available for a memory instruction that is happening next. This is different than the memory to execute forwarding predicament, in that we are requesting a read that is dependent on a value stored in a register, but the value is not needed for computation through an ALU. Thus, the MEM → MEM forwarding scenario is unique in that the determination to forward the previous memory stored happens in the MEM stage itself. This requires yet another multiplexer with a different control signal. This control signal is represented as calculated again through the "black box" model used and passing through stages without a latch, checking whether the value of the destination register in the previous memory stage's MEM/WB latch matches the source register used for taking and placing data back into memory. Note, then, that this only happens when a write is about to happen after a read has completed in the previous instruction. That is, when the opcode for the current instruction is to write, but the previous instruction's opcode was to read into a destination register, and the source and destination registers across the instructions match, there is a case where forwarding is required. Therefore, we have the forward_mem signal proposed on the datapath signal high when taking this forward value, taking SR2 otherwise. Only SR2 is considered here, as memory instructions that will use a source register to load will only use the SR2 slot that is interpreted here.

Notice that, in this report, we have not mentioned branching hazards that may be present yet. This is due to no emendations that have been placed since our original report and paper design, and a branch resolution that happens based on the signal as soon as the execute stage. However, this still leaves two operations that could potentially operate (or at least go through the pipeline) if we are not careful, and thus further stalling will be required in this pipeline. Thankfully, with the additions used for data hazard detection through a multiplexer, we can set d_h_sel to high again (and take a no-op) whenever we see an instruction decoded as a branch in the ID stage. From there, we can use the value of br_en from the execute stage to already conclude if the branch was taken and stop stalling the pipeline. Since this can also just be controlled with the signal d_h_sel, we have not included any additional signals outside of a control unit that computes said signal. The signal is then computed based on both rules discussed in this paragraph, and with the memory case from earlier here. This will require values from the instruction register as they enter and exit the latch at IF/ID, as well as the signal of what opcode is from the ID stage and EX stage, and if a branch prediction was taken.

Now that these cases have been elaborated with all logic of the design added enumerated as well, one may refer to each change using the datapath present on the next page.


IF / ID
ID / EX
EX / MEM
MEM / WB
WB [/IF]
Instruction Cache (to memory)
imem_addr
PC
+4
alu_out
[WB]
LSB Set 0
dmem_resp
dmem_read
dmem_write
pipeline_enable
d_h_sel
imem_resp
br_en (from ex)
j_en (from ex)
Hazard Logic
imem_rdata
imem_resp
d_h_sel
addi x0, x0, 0
Opcode, sr1, sr2, dr (from top to bottom)
Control Signal Extractor (given)
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
CMP
i_imm
cmpmux_sel (from ctrl_word)
rs1_out
pc
Ex_alumux1_out
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
rs1_out
rs2_out
Control Signals (previous)
Fwd_mem
M3 [mem/wb]
dmem_byte_enable
Data Cache (to memory)
dmem_addr
wdata
dmem_w
dmem_r
SE Byte
SE Half
UE Byte
UE Half
ir (from previous)
alu (from previous)
mdrmux_sel
SEXT
u_imm
Regfilemux_sel
load_regfile
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 Two: Statement of Individual Contribution

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

Robert Jin

Robert was the primary contributor to all the SystemVerilog code completed in this code, responsible for much of the connections that were placed between all components proposed in this section. His primary objective was to modify each of the caches and stress test both the frequency of the caches and the sequence of instructions that were taken, in coordination with Yan's tests. In reviewing and consulting with the team on all revisions that were necessary, he considered and wrote all three of the level one and two caches implemented in this checkpoint. These were, in the end, heavily modified versions of the MP2 caches that were previously implemented, expanded for new data constraints as well as hierarchy considerations. For example, Robert was the first to consider addition of the MDR within the level two cache instead of in between caches, after consulting Kenny (our mentor TA) on the matter.

Outside of this work, Robert was also primarily responsible for connecting the caches to the arbiter and finalizing the memory hierarchy. This was done in coordination with Robert Altman's work in constructing the initial skeleton of the pipeline, as well as the arbiter control and mapping logic. Beyond this, however, Robert found issues and specified solutions that were instrumental in ensuring correct operation and loop prevention between the level one and two caches. Without this help, the project could very well still be using physical memory now.

Finally, after consideration between the entire team on how to adjust the pipeline via a register load modification, Robert was also responsible for adding the final pipeline stall logic to the main pipeline datapath. After doing so, the first tests for checking on data read and no-op passing through the pipeline were written by Robert, and all further tests were completed with collaboration from Yan Xu.

Robert Altman

Much like in the last checkpoint, Robert was mostly instrumental in constructing the initial logic of cache interaction and devised the original control logic for the arbiter as well as the sizes to use for the caches (with added contributions given from the group that were agreed upon). Further, Robert emphasized points where the design for the arbiter would require multiplexing on the address after being alert from our mentor TA, Kenny. Concerns about delay were also discussed due to Robert's typical role of checking on timing constraints throughout the project, and he collaborated with Robert Jin to check possible ways of reducing large combinational paths and at least ensuring a base frequency of 100 MHz is met. (At the moment, with the current design, the frequency is 107 MHz, with the caches removed going as high as 193 MHz.)

Unlike the last time, however, Robert also was the primary author of the cache arbiter datapath and control, and thus wrote the first instances of the memory hierarchy that could be used along with it here. This included some tests that were evaluated solely with the arbiter when connected to magic memory, as well as walkthroughs for this code to ensure that no problems were had if there was a connection between unmodified MP2-based caches interfacing with the arbiter and physical memory (connected downstream by the arbiter). Most tests that were written ultimately for the arbiter, however, were written by Robert Jin and Yan Xu. Robert Jin was also instrumental, as noted in the previous section, in finding a logical loop causing issues in Robert Altman's original code presented.

Finally, as with the last checkpoint presented as well, Robert is responsible for most of all designs, graphics, text, and explanations provided in this report. The final version was agreed upon by the team altogether, but the construction itself in rough draft form along with the digitization of all group written materials was handled by Robert here.

Yan Xu

Yan was the primary coordinator of our meetings, and often started each meeting with a set of reviewed issues that he found since a prior meeting that needed resolution in the datapath. Therefore, he was often responsible for organizing thoughts that drove implementation features for the two caches and presented challenges in our current designs before we had to debug them in our actual implementations. This was a critical role in the long run, as it forced the entire team (including Yan) to think critically about why we were applying certain assumptions about our connectivity in the arbiter, or our general approach to stalling. It also helped collect our thoughts for more formal documents, such as this report.

Outside of this and his work with Robert Jin in collaborating on some SystemVerilog writing, Yan was the primary team member to write tests that confirm all functionality. Alongside Robert Jin's modified tests from MP1 to test all functionality (given hazards in mind), Yan wrote original tests to track behavior of functions that might cause hazards (in preparation for the next checkpoint). Yan also wrote tests that evaluated a precise number of no-ops between instructions to ensure our assumptions on when data was available in the pipeline were still being met despite delay from the cache (and memory) reads here. This saved a great amount of time in the long run when evaluating and fed back into his original objective of raising issues or potential concerns prior to diving deeper into concept during meetings.

On a final note, Yan was the primary author of the *Hazard Detection and Forwarding Design Plans* section of this report. Written alongside Robert Altman in some part, with commentary provided by Robert Jin depending on his observations from preliminary implementations, this was constructed to introduce and elaborate on some of the forwarding types needed (and why they were particularly considered for inclusion in this checkpoint here).

Checkpoint Three: 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 data hazard detection and data forwarding, ensuring correctness without the need of manual *no-op* instructions to be used in program code. This should still support all instructions available in the base RISC-V architecture, except for special instructions as noted in the MP (such as FENCE). In the process of completing this, we will also explore the ability to stall our pipeline from loading any new instructions based on branches and jumps, until the branch or jump has been evaluated and PC properly updated to reflect the correct instructions that follow here.

Work will begin 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. We will not implement any of the actual forwarding 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. Robert Jin will lead the development of this, as he was the team member originally pitching this idea, with Robert Altman available to assist in any development and debugging. Tests will be 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. (Such testing may be confirmed by adding signals for the module in ModelSim.)

From here, the additional multiplexer that mimics the execute operation decision in the writeback stage will be added to the execute stage. That is, the multiplexer that computes **EXE** for the multiplexers that can set RS1 and RS2 for forwarding will be inserted next. This is also the stage where the control signals will need to be modified, and thus multiple persons will likely work on this task. Specifically, we plan for Robert Jin to work on the control signals that will drive all multiplexers through the design during this time, including this multiplexer, and all construction of the values and routing from the EX/MEM latches will occur by Robert Altman and Yan Xu.

Once this is done, the control logic as enumerated earlier in this report will be completed together by the team. While Robert Jin will be writing the SystemVerilog, the group will collaborate and determine what is suitable for the two-level conditional development that drives signals such as *forward_rs1* and *forward_rs2*. Isolating this step allows Yan and Robert Altman to write more tests that confirm the correct outputs of these signals before they modify RS1 and RS2 through a multiplexer.

One more step still left before continuing is addition of the hazard detection and control that will stall the pipeline and detect where no-ops must be passed on. It can be argued that most hazard detection itself was already done in the control signal development led by Robert Jin, since this checks the instructions and data registers that are being used across multiple steps, checking for conflicts through generating these control signals. However, this work led by Robert Altman will be placed into the instruction fetch stage before even setting the instruction register and being able to decode any signals. This allows for a no-op to be passed if it is necessary to ensure data is ready in a certain latch

down the pipeline for placement in the execute stage (particularly, this protects for the MEM → EX forwarding type). Robert will also work with Robert Jin to ensure that the x0 destination register and source register conflict cases are not indeed marked as causing a hazard, as this register writes to ground and stays zero (therefore remaining static and never changing nor causing a hazard here). Through completion of this process as well, Robert Altman will investigate branch prediction that can be linked to the data hazard signal as alluded to two sections ago in the *Proposed Datapath Emendations for Forwarding* section. The same concern will be given to jumps as well, as the PC has already been readied in our pipeline to stay on the current PC (+4) path or update if a jump or branch has been taken, but the instruction load sequence has not been prepared (with the new introduction of stalling from checkpoint two).

After the hazard checking and signals have all been prepared, the team may continue with all types of forwarding that have been proposed in this checkpoint report. Starting with the execute to execute forwarding type, we will create our multiplexers for both SR1 and SR2 and disconnect the signals for memory and writeback signals. Since the multiplexer that calculates EXE has already been completed in a previous step, we can connect this signal to input one of our multiplexers and start to write tests that confirm functionality here. Robert Jin will be responsible for writing the system Verilog, doing so with assistance from Yan Xu. Robert Altman will write the tests 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 will also write tests with Robert to test this implementation as well. (Such tests have already started based on work started, but not confirmed, before the time given to complete this checkpoint.)

After this, MEM → EX forwarding will also be 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 will also collaborate in this work. From here, Yan and Robert Jin will write the initial tests for confirming the design works, specifically testing load and arithmetic type sequences (read-after-write and write-after-write) that are completed here. Robert Altman will also test that no unexpected behavior occurs as a result of storing this data, in that data is not incorrectly forwarded to a following instruction if no change is needed. This can be checked by observing the control signals used.

Prior to finishing the final EX forwarding step, Yan Xu and Robert Altman will collaborate on writing the memory to memory forwarding design. This will include tracking the values provided into the memory latch as well as when they become available, such that they may be forwarded and considered with the rs2 signal otherwise given. Since the control signal that will be used for the multiplexer in this MEM stage was written by Robert Jin, he will also contribute to testing this section. After this, Robert Altman will test load word and store word pairings to ensure functionality is correct here, and Yan will test LA/LW type of pseudo-operations as well as different types that are loaded and written (half words, unsigned bytes, etc.).

For the final type of forwarding (WB → EX), Robert Jin will again lead development and ensure 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, will be aided by Robert Altman, with contributions as well from Yan Xu. Specifically, tests will be written such that different spacing levels are used to test each type of forwarding, 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 will also occur, such that the original premise of the program written may be confirmed here.

While these tasks happen, timing optimizations will continue after a concerning frequency drop from ~ 190 MHz to ~ 107 MHz after addition of the caches here. Particularly, Robert will 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 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.

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 6, 2018 from 3:20 pm to 5:00 pm, outlining some of the initial concerns we may have about placing logic early in the Instruction Fetch stage to check what type of instructions are being processed. We would also like to start discussions on designing for branch prediction and the advanced features listed in checkpoint four here, as well as the load mechanism with respect to future implementation of advanced features (such as memory stage leapfrogging, which is in discussion now). As a reflection session and collaborative effort, we will take work collected after the meeting and review in group from 6:30pm to 8:45pm on Wednesday, November 7. Given the pace that we have already placed with this work, and some success with data forwarding confirmed in some of our tests, we would like to have all data hazard implementation completed by this stage. This should also be where the first evaluations of performance using ModelSim should take place for a fully implemented checkpoint three, prior to checking timing constraints and other design concerns here. Thursday will again be used with split time as it was last week, with Yan and Robert Altman meeting from 3:20 to 4:00, and Yan and Robert Jin meeting for an hour or two after. All sessions after this will consist of planning for advanced design features and forwarding approaches, report clarifications, as well as generating test cases and scenarios, most of which are scheduled for Saturday and Sunday evenings and the week ahead (so to advance ahead in our design for more complicated additions). 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.