MP3 Checkpoint One Report

ECE 411: Computer Organization and Design

Room 3032

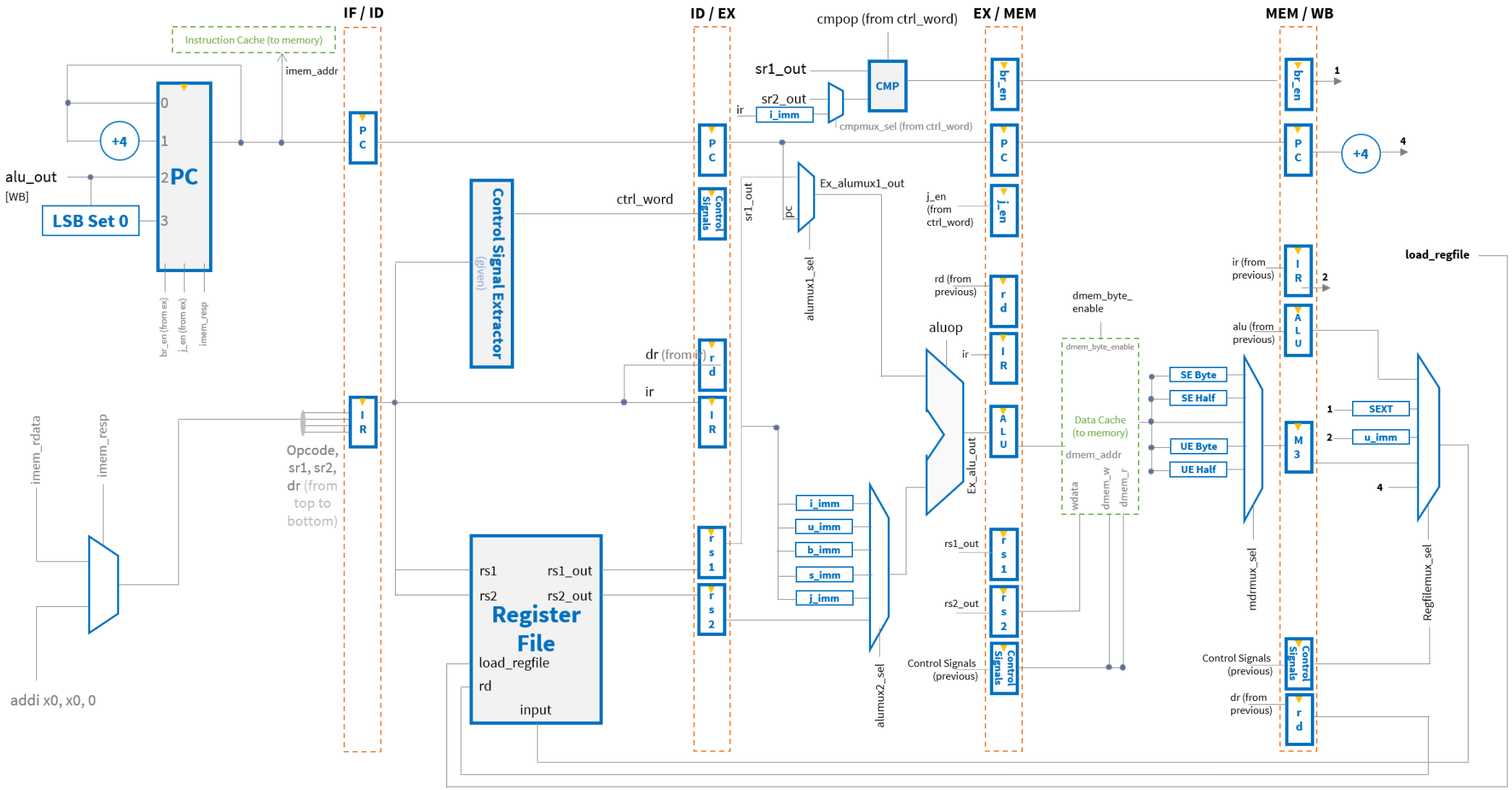
Robert Altman / raltman2

Robert Jin / naiyinj2

Yan Xu / yanxu2

Document Contents:

Revised Datapath Diagram



Enumeration of Revisions

The instruction register that was placed before another instruction register latch between the IF/ID stages has been removed. This was inducing an extra clock cycle delay for passing through the instructions, and thus causing bugs especially at the start of the application. There was also, from the previous register's existence, a delay of two cycles instead of only one induced at the latch.

The comparator logic has been moved until after the execute stage, to simplify what happens in the instruction decode stage. This is because branch enable is computed in the execute stage but relies on interpretation of the values at SR1 and SR2, which may only be ready and stable at the end of ID. Therefore, this eliminates potentially negative circumstances from keeping this logic as well.

The logic for computing *load_regfile* has been removed. There is still an option, as discussed in the previous paper design submission, to modify the register file such that writes to x0 are prohibited always. However, since pin x0 is connected to ground at all times, it does not matter if we do load the register file at x0 during a no-op cycle, since the result of the no-op will not write. As a result, the true intent of the no-op holds, in that no write or operation of any sort is truly executed that alters program or processor execution.

SR1 and SR2 were also renamed to RS1 and RS2, to match with the naming conventions given in the RISC-V specification. For similar reasons, the term "DR" for destination register has also been reversed to read "RD" on the new diagram.

Several signals were misspelled or misnamed according to their MP0 to MP2 equivalents, requiring emendations in this copy. Namely, the selectors for the ALU Multiplexors have now been renamed to *alumux1_sel* and *alumux2_sel* instead of *alu1mux_sel* and *alu2mux_sel* (respectively). Furthermore, some names were given to the outputs of this and the ALU output that is visible in the diagram. Finally, *mem_byte_enable* on the data cache has been renamed to *dmem_byte_enable*, as it is in the "magic memory" that was given for this MP.

A connection between branch enable to the sign extend on the first pin of *regfilemux* was previously missing from the paper diagram. As a result, it has been assigned to the top register named br_en with an arrow pointing to **1**, now aligned correctly with the input marked before here.

On the same register file multiplexer, the PC value that was placed into pin 4 was only the PC of the instruction that was currently executing here. This, however, would not work correctly for jump operations that required that the *next* instruction was stored in a register, leading to ultimately incorrect branching (or jumping) behavior. As a result, the same arrow out to 4 is still used, but a "+4" component has been added before the input to the pin on the register file multiplexer in this iteration.

Lastly, the input for alu_out going into pin 2 of PC during the instruction fetch stage (and into the "LSB Set 0" component of the same stage was not clearly marked as to where this signal originated. Therefore, a [WB] has been added to clarify that this is a value obtained from the output of the writeback signal of alu_out (stored in the latch titled ALU from the MEM/WB latch series).

Checkpoint One Progress Report

The proposed goal, as given in the original RISC-V documentation, is to complete most of the RISC-V instruction set. This was, however, without 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 work completed in MP1 and MP2 of ECE 411 across the class, we have implemented and run tests on *all* instructions.

Prior to completing all else of the five-stage pipeline design throughout this checkpoint, we first emphasized completing the control word ROM, whose skeleton was provided in the original documentation. Led by Robert Jin in the group but contributed to and discussed by all, we composed a sequence of signals that would be output into a word by this ROM, such that they might be used in future stages of the pipeline and could be easily (and directly) assigned to latches across stages in the pipeline. Each signal carried the naming convention of the signals that were used in the first MP, as well as the signals that were proposed on the original paper design (such as ALUMUX1_SEL, which as noted in the previous section has been renamed from ALU1MUX_SEL in the original submission, a typo). However, we noted that the use of special signals for the data cache would be necessary, and we wanted to ensure read, byte enable, and write signals were isolated specifically for the data memory (data cache) only. Therefore, extra signals called *dmem_read*, *dmem_write*, and *dmem_byte_enable* were created to suffice this.

Once the control word was set here, all signals could be properly used and assigned in all subsequent stages. Next, however, we needed to design the latches that were placed in these stages, as well as the logic and components that would be used in each one of them. Following initially our proposed datapath design closely, we soon found that there were inconsistencies in the names throughout the data path, along with some other changes. The changes themselves have been enumerated in the previous section.

Outside of this, the first step of the two was to complete the latch design. Therefore, specific locations were placed in the code for all the signals used during each stage, as well as their corresponding registers (latches) that would toggle changes to them (should they be available) on the next clock cycle. All of this created the framework for *datapath_pipeline*, the equivalent to cpu_datapath from the previous MPs but now with its own self-sustained method of handling control (without a state machine). Robert Jin led the implementation for this stage, Robert Altman completed timing and initial steps for paper design and revision here, and Yan Xu contributed to both programming aid through debugging and work with Robert Jin, as well as discussions held with Robert Altman on the integrity of original concepts as translated into code here.

The next step, while originally feared to be difficult, was more or less simpler than the previous: completing the rest of the design by using components in all the stages. Our design had already detailed much of this, including the exact instruction to use for *no-op* (stall) simulation (although it did not set the equivalent bits, which had to be referenced via the RISC-V Instruction Set Manual given). Perhaps more importantly, however, we could simply reuse most of the components we had

already created or used for previous MPs designed for the RISC-V architecture. For multiplexer logic, register files, PC handling, and others, we could simply thus reuse the components. The primary difference that was notable here was a connection to memory, which required examining the given “magic memory” with support for both data and instruction caches that always hit. Since the construction of said memory was relatively similar to that used in previous MPs (such as MP1), this process was generally handled with concern. However, the instruction cache read and write signals proved to be an interesting case, as we theoretically wanted to pull a new instruction as soon as we could in the pipeline, knowing a no-op would otherwise be spawned if one was not available. Further, we never had to write a change to an instruction, as our design assumes no self-modifying code was placed or used that would change the instruction about to be loaded. Therefore, we set the write signal to the instruction cache to low always, and the read signal to high always. We will modify this for future iterations of the MP, however, only indicating a read when there are more instructions loadable in the pipeline (useful if, for example, five load words are creating delay and the instructions should not be overwritten).

Once this was finished and modules from previous MPs were reused and configured as appropriate, testing was now emphasized as a next step in the project. To first understand how the pipeline was functioning here, a variable number of no-ops was placed in test code to see where data hazards may exist in the current process. We found that a total of three no-ops were needed to ensure that there were no issues with write-after-write and read-after-write dependencies, as the items from the pipeline would commit to the register file after the writeback stage completed, and the data would only be interpreted for the values of RS1 and RS2 three stages earlier. One solution was to make the register file itself triggered on the negative edge for write, but as this would heavily affect timing, the solution was skipped (it also would only remove one no-op, or stall, from the pipeline). Our first test consisted of simple immediate arithmetic instructions of various types, with some immediately followed by branches where there was no register value dependency to test that multiple instructions could be computed simultaneously. Care was taken to place no-ops after the branches so that no other items would execute here. Interestingly, even by executing our first test, we already noticed the effects of a bug in which we were not taking in the newest value from the IF stage into the IF/ID latch for the Instruction Register, something that fundamentally delayed the execution of the pipeline and gave incorrect results of what instruction was executed (namely, high impedance and incorrect execution throughout).

(continued on the next page)

Only after these cases were tested did we move on to further tests, where we took old test codes that were written for testing all individual instructions (including jumps) in MP0 and MP1 and added no-ops where necessary. This includes adding no-ops after all jumps that were present as well, as we treated these as close as we could to branches even though there was no need for guessing which instruction was taken after this (jumps, even those done by JALR, would always be taken). This proved difficult in some cases, as some of the tests were deliberately written to test that WAW, RAW, and RAR dependencies did not influence the result of our interactions even if executed immediately after each other. Indeed, re-running and modifying these tests again will become incredibly valuable in the future MP checkpoints, especially for the purposes of testing MP3 Checkpoint Three.

It is worth noting that more comprehensive tests that caught specific corner cases, especially those dealing with immediate value arithmetic for example, were the ones that were first tested in depth. That is, all instructions that were meant for the second checkpoint were evaluated for some correctness at this stage as well but were done so to lesser extent than the functions evaluated at this checkpoint. Tests are still being developed for this set as adapted from our previous MP1-based cases (and MP2-based cases for word/byte/half differences) to ensure accuracy.

Overview of Cache and Arbiter Interaction

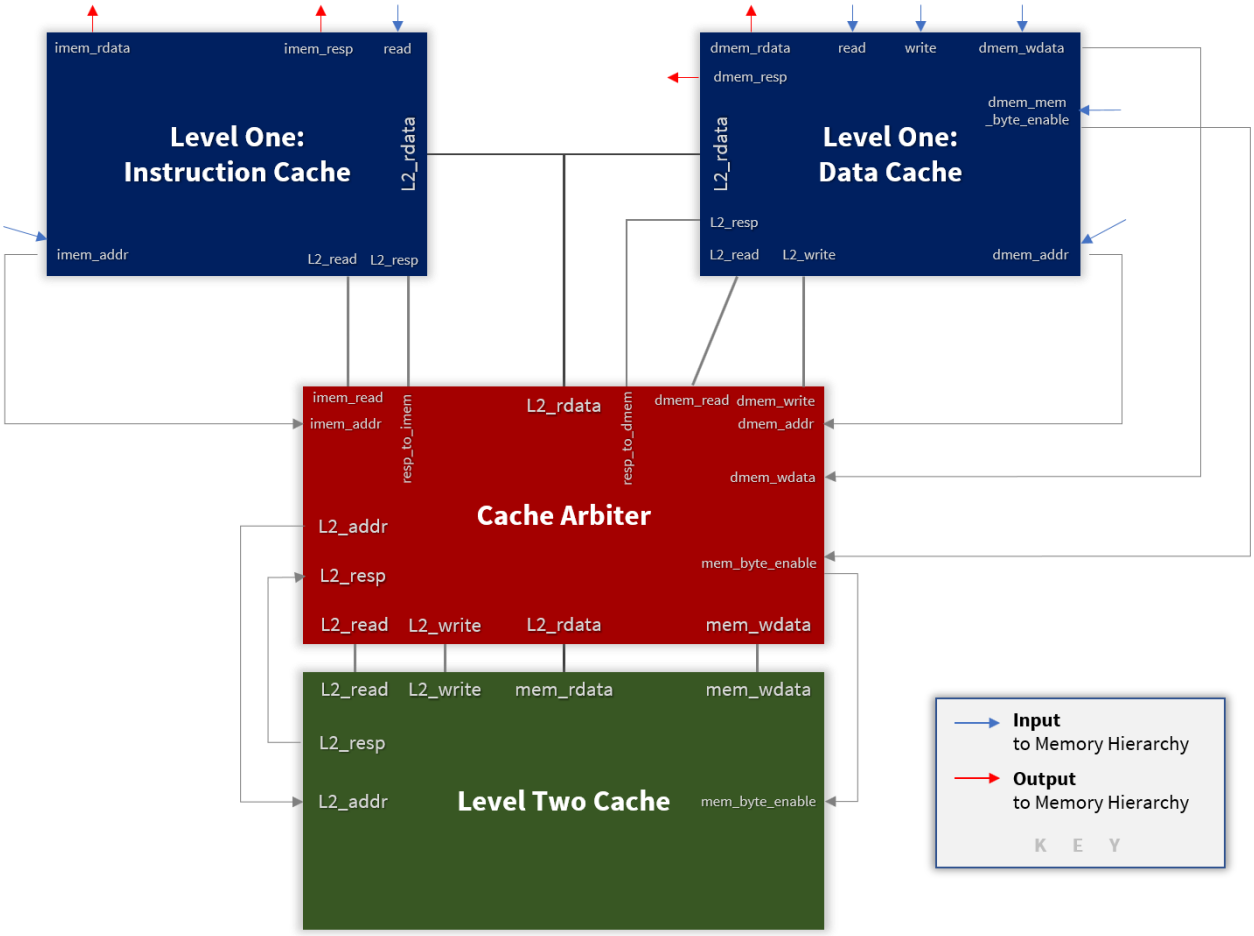


Figure 1: A full view of the caches and arbiter in black-box form, signals labeled.

In the next checkpoint, we will build our fundamental memory hierarchy by separating our instruction and data caches into two separate, smaller level one caches. These will interface and interact with a cache arbiter that handles traffic to a level two cache deeper in the hierarchy, which is larger in storage size. Past this second part of the hierarchy, a data connection is then made to physical memory from the level two cache, which is not pictured in the above view. Thus, the memory hierarchy will start with the level one caches at the first part of the hierarchy (with smallest storage size and lowest latencies).

For all caches, the model used will be heavily based off the two-way, set-associative, Least Recently Used model cache designed and implemented for ECE 411's MP2. The caches themselves will indeed be two-way, set associative, LRU caches, but not all will have the same number of arrays or require storage depending on their use by the pipeline and frontend of the processor. The sizes, as discussed in the last paragraph, will also differ in a tradeoff between latency, energy draw, and storage space.

Prior to discussing the specifics of the arbiter and level two cache design, we must first review the design for the level one caches upstream in this hierarchy. The instruction and data cache will have

eight (8) sets available, with 256-bit cache blocks, allowing thus for 512 bits of storage per set given its two-way construction. These 256-bit cache blocks will match the largest size of memory that can possibly be obtained from physical memory in a single read, and thus limits the size of the cache blocks that is used for the level two cache (discussed later). Since there are two ways, a Least Recently Used array will be needed for each set to determine which way was used last, such that the least recently used way can be evicted at a certain set should no tag match exist.

Note, however, that the instruction queue will only be used to hold instructions, of which will not be re-written while in the queue since self-modifying code requires access to the data cache and population of values there. As a result, all write functionality, including physical memory writeback functionality, can be removed fully from the instruction queue. This was an abstraction that was partially avoided (as mentioned in our progress report) through checkpoint one, where the write signal to the instruction cache was set to low always. Therefore, we can remove the dirty bit arrays from these caches altogether, as there is no need to track stores other than those coming from the level two cache or physical memory itself. This reduces the energy draw even further for this cache design, which is beneficial at this position of the hierarchy. Additional combinational logic that would be used to distinguish between a read from memory and a write from the datapath, as well as a pool of signals themselves for the write data, can also be removed here.

In both use cases for the instruction and data caches, there will be misses that will require interaction with other memory caches or physical memory structures down the hierarchy. However, both can potentially request access to the L2 cache's data, or only one, or neither at all. Therefore, there needs to be the arbiter that handles this mechanism and properly routes all signals from the data and instruction caches, making sense of which cache to service first and what data to load or write from the L2 cache. The arbiter itself does not need to interact with physical memory, leaving this interaction the responsibility of the L1 cache. That is, while the control logic of the L1 caches will send a request to L2 for data, which the arbiter must intercept to prepare for the L2 cache where simultaneous requests may occur, the L2 cache interfaces directly with physical memory as "next" in the hierarchy.

Separate response signals will thus be given to the instruction and data caches, depending on which one is being serviced. Similarly, a response signal will itself be generated by the level two cache, which in turn will be used by the arbiter to generate one of these signals. It does not matter that read data from the cache is given to both here, as only the cache with its incoming memory signal as high will interpret the data here. This will be discussed in the forthcoming sections.

Prior to closing this section, however, it is worth noting that the entire process of handling a miss will no longer take just a single cycle, as with our previous design. Therefore, our entire datapath design must introduce stalling here. Whenever the data cache has a signal high but is not reporting a response from the memory hierarchy, all instructions should be stopped as the instruction waiting for data load or store cannot complete execution otherwise (the data for the instruction, from execute and memory stages, will have been overwritten by a different instruction). Thus, combinational logic will be added to our datapath that does this exact check, disabling any writes to any latches to "stall" the pipeline until a response signal is given.

Level Two Cache and Arbiter Design

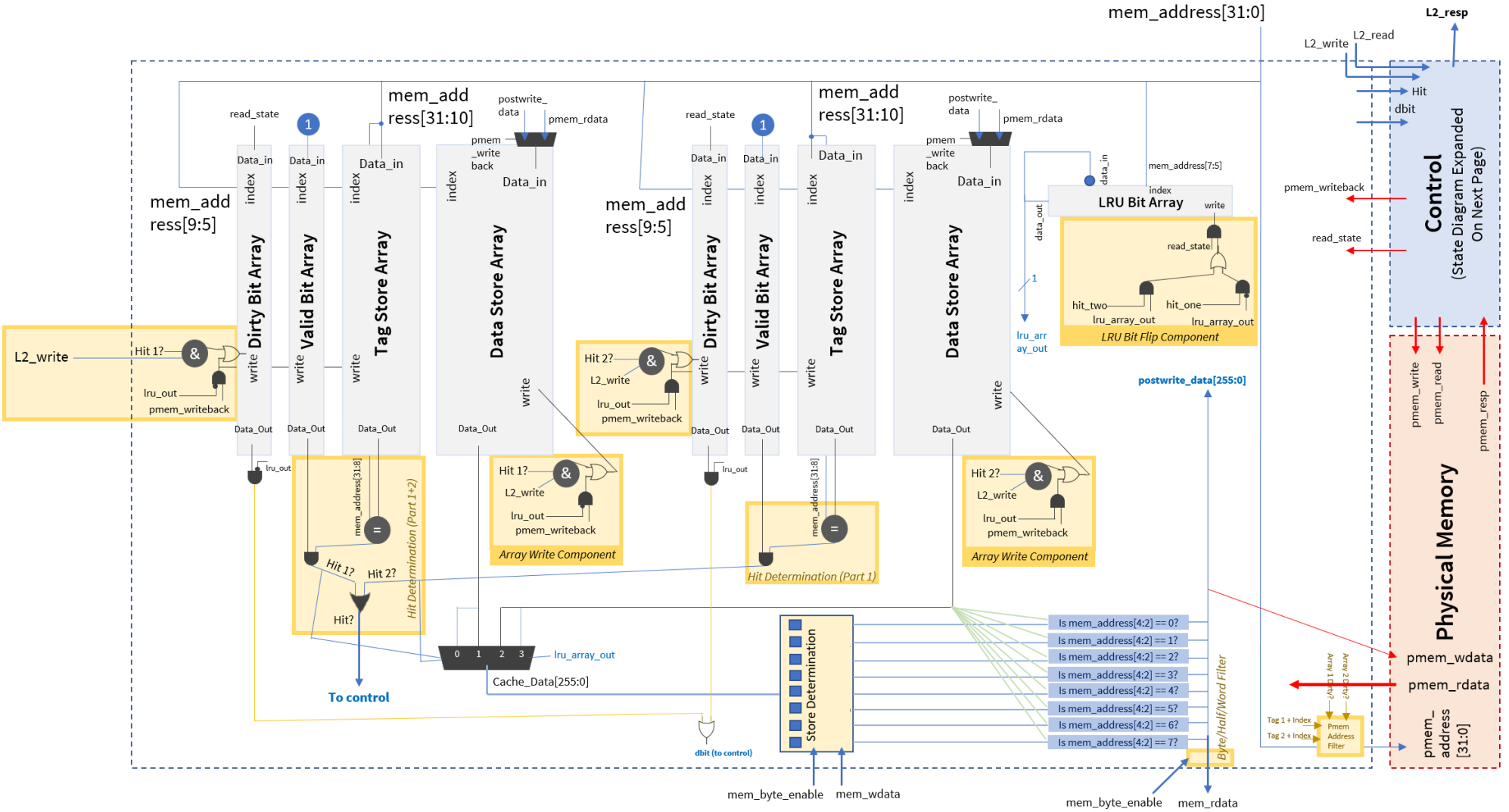
The single Level Two cache found in our proposed memory hierarchy also mimics an “MP2-esque” implementation, as a two-way, set-associative, Least Recently Used cache. However, making a cache that is identical to the ones that were placed in Level One would offer no benefit to the pipeline, essentially copying over the data (at best, or half the data at worst) found in the level one caches. Therefore, the size consideration of this cache must be weighed given its place in the memory hierarchy, which also comes with its associated power draw tradeoffs.

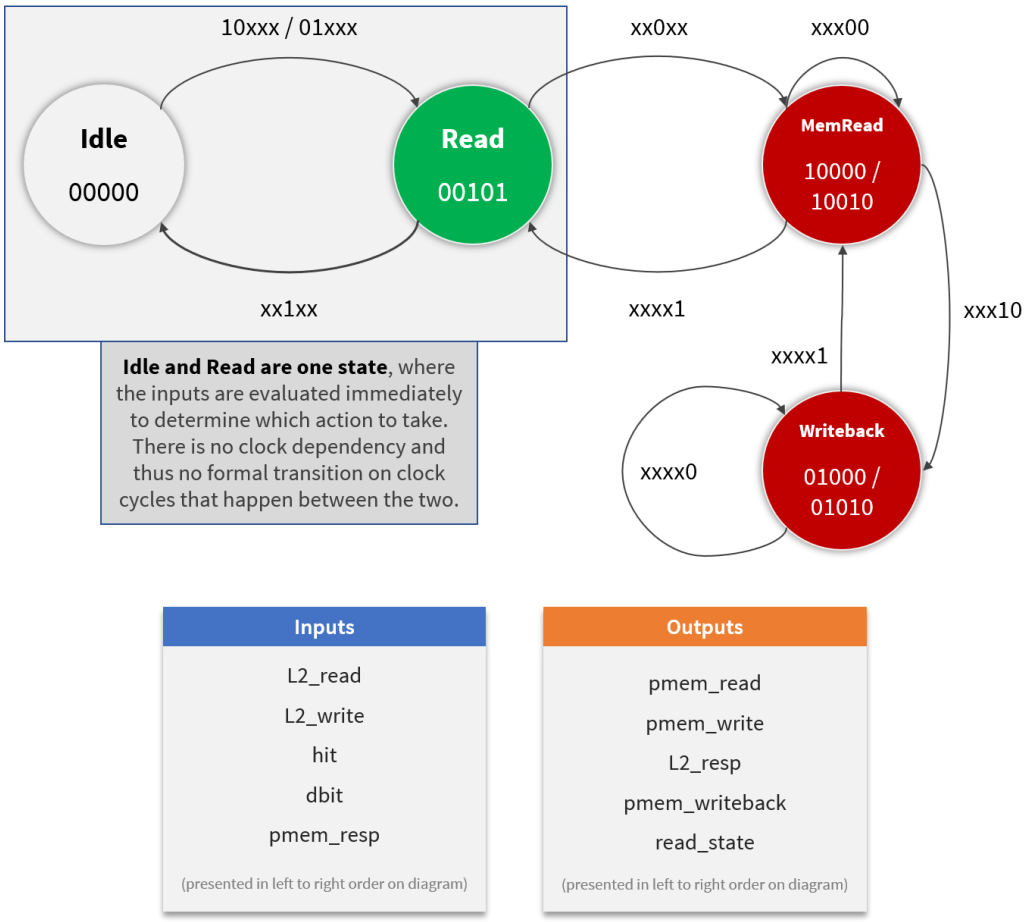
For this proposed processor, this L2 cache will have **32 sets** compared to the 8 of L1 caches. At 256 bits per cache block, capable of holding eight words per way (and thus sixteen words per way), this cache is thus four times the size of a single level one cache, and two times the size of both level one caches combined. Doubling this size instead of just limiting at 16 sets is practical considering the typical use of L2 caches versus L1 caches, where data present in the L1 cache must also stay in the L2 cache (particularly, immediately after a load). In fact, this very capability and availability to access cache data by different processes in software is what leads to cache side-channel attacks, which was exploited along with prediction protocol in recent Spectre and Meltdown attacks. Nonetheless, this large size is stressed due to the typically large size and locality of programs that may involve branches and loops, where items evicted out of an L1 instruction cache (for example) or since rewritten and removed from an L1 data cache may be easily pulled from the L2 cache without needing to interface with physical memory.

Of course, however, such cases in interfacing with physical memory are still unavoidable. Since there are no larger caches that are available in this hierarchy, and this is the last “pass-through” of sorts for data to be placed when it is loaded for the pipeline to use, there must be an interface to the last item in the memory hierarchy (which is physical memory). Therefore, any misses in this cache will prompt the controller of the L2 cache to signal to physical memory a need for reading and/or writing back data. This data to writeback will have been provided by the L1 data cache (more on this later). The high latency that this induces, however, causes even further delay and stalling for our pipeline in the frontend of our processor. Not only has the processor taken a cycle to get to a request for the L2 cache, but now many more cycles must be used before data can even be returned from physical memory, so that L2 can report to L1 via the arbiter.

Therefore, we want to avoid misses as much as possible by increasing the size of this cache within limitations of our design. That is, with a higher amount of placement of single bits and combinational logic from this cache, there are clear fitting issues and locality concerns that may invalidate our timing constraints if not careful. Leaving this at 32 sets of 2-way, 256-bit cache block size is a way to moderate and prepare for this potential constraint while not sacrificing the benefit of having a larger size and more data available here. In the future, when analyzing timing constraints, we will have to particularly watch the behavior of this cache and its compliance to our target frequencies here.

The full datapath of this L2 cache, again bearing resemblance to the previous cache presented for MP2, has been included on the next page. Note that, like with the diagram at the beginning of this report, rotation may be necessary to view the cache datapath correctly. Also note that the design of this cache is nearly identical for the Level One caches discussed in the previous section, except for the Level One Instruction Cache that bears no write functionality. Otherwise, the only difference between each cache is the distribution of memory address bits and size of the tag arrays, to handle a larger number of indices in the level two cache. That is, where 3 bits suffice for distinguishing between sets in the level one caches, 5 bits will be needed to distinguish between 32 sets in the L2 cache here. This datapath can be seen on the next page.


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



The control unit, while consisting of only three states, handles all the cache and physical memory interactions taken in the two-way set-associative cache model. Note that Idle and Read are treated as the same state above – for the purposes of examining this cycle, we will start by investigating these as "separate" states.

For most of our operation, however, we must have a state that we indicate where the cache has completed processing the memory task it was assigned. For this reason, the model starts with an **idle** state that continues to hold. It must not, however, report anything that has happened with physical memory, give any writeback information to the cache, or tell the CPU that a new computation is ready. That is, in this state, the cache is explicitly disabled as it is not being used at all – therefore, all output signals should be low, and no new operation should be happening in the cache (besides preserving the data already present on each clock cycle).

Whenever a memory read or write is requested, however, we must intercept it and wake up the cache from previous operation, telling it that it must read or write data in what we call the **read** state. Our cache should be designed to get data and write or return it immediately if there is a hit in the cache, but first we must know when such a hit happens. Therefore, we take this as an input to the control, and note that when a hit does happen, read will complete. It is also only in this case that we get the first sequence of output bits written in the above diagram, of *00101*, which indicates that memory operation has completed after this cycle and the data can be safely read out of mem_read or

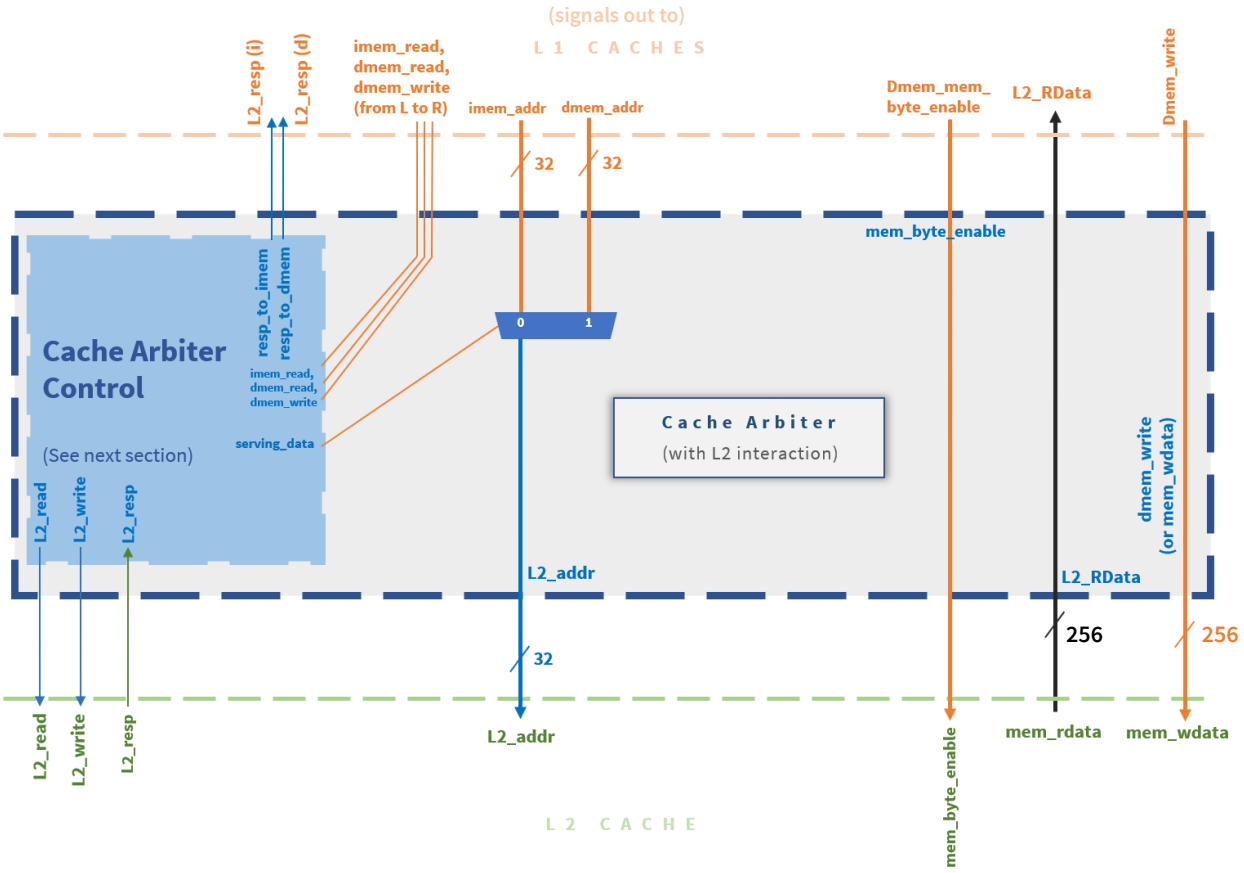
operation may continue after a store. Note that there is no self-loop to this state, as this should complete without having to rely on physical memory or multiple cycles to read here. Not every transition, however, leads back to *idle*…if the cache determines that there is no hit, then we must go back to physical memory to retrieve data. This might happen on both reads and writes, so the only item that should determine our transition to these physical memory interaction states (labelled maroon in the diagram) is the hit bit being 0 – indeed, we see that is the case.

The first of the two physical memory states that are involved in this process is the **memread** state, which by its name, handles all interactions with physical memory to read data alone. It also, however, determines whether data must first be updated in the physical memory if it had been updated in the cache before (through stores). This data is gathered through the dirty bit of the element that is getting evicted, or *dbit* as it is passed into our control. Should this be high, and no physical memory be interacted with yet (xxx10 as a transition set), we want to make sure that we write back the data first before proceeding, and thus transition here. Otherwise, we are doing a read, in which case we want to return to the read state only when there is a physical memory response that occurs. Only in this case of granting a response will we also output the *pmem_writeback* signal, a signal to our cache that indicates new data has either been stored or read from memory and the dirty bit data should be flushed, with data in the evicted array also replaced with what has been read back from memory. In all other cases, then, this signal is low, and we wait for physical memory to respond (one of our inputs). As soon as it does, we stop looping to this state and go back to read.

The **writeback** state works similarly, except that it sends a pmem_write signal instead of a pmem_read signal to the physical memory here (as told by the difference of the first two bits between the outputs of the two states). Whenever no response is held from memory, we cannot report that any items have changed, and want to stay in this state until our data is marked as reliable. Once it is, we report this through our *pmem_writeback* signal to again flush the dirty bit corresponding to the evicted data array, prior to going back and overwriting this data with the data requested from memory (that will now replace old data in the cache). This read process from physical memory can then continue when physical memory responds, allowing the rest of the cache control cycle to be completed in time.

It is possible, by condensing how writeback is handled through an extra bit or tracking whether pmem_write is high, to condense the writeback and readmem states into a single state. For simplicity that differentiates these states between clock cycles, however, the states were kept separated so that a memory response in writeback was known to always be for a write, and the same in readmem was only for physical memory reads. This also helped for distinguishing and debugging in waveforms which state was active during cache misses, particularly if data was dirty and needed to be written back here.

Prior to discussing the arbiter's implementation and mapping technique, the Idle and Read state merging should be explained. The very transition from Idle happens as soon as there is a data read or write that happens, but this only moves to a state that then looks for hit available. Such information is available, however, as soon as read and write are as well, and therefore evaluation can happen without changing state. If a hit indeed is found, we can return a response and not worry about returning to *Idle* to complete the process, saving a cycle in the entire scheme thus.



The arbiter is the final piece of this memory hierarchy, again mapping all the signals that were previously discussed here. To know what signals should be mapped between the level two cache and the level one caches, however, there needs to be a control that can keep track of state – that is, the cache in L1 being serviced, if any. Too, since instruction reads from the cache are more frequent than data reads, we want a mechanism to prioritize this, and thus signals are set up in this arbiter in such a way that instructions are assumed accessing the cache. A breakdown of how this control works, and how it prioritizes specific signals given certain inputs and requests is provided in the next section and will also be briefly recapitulated.

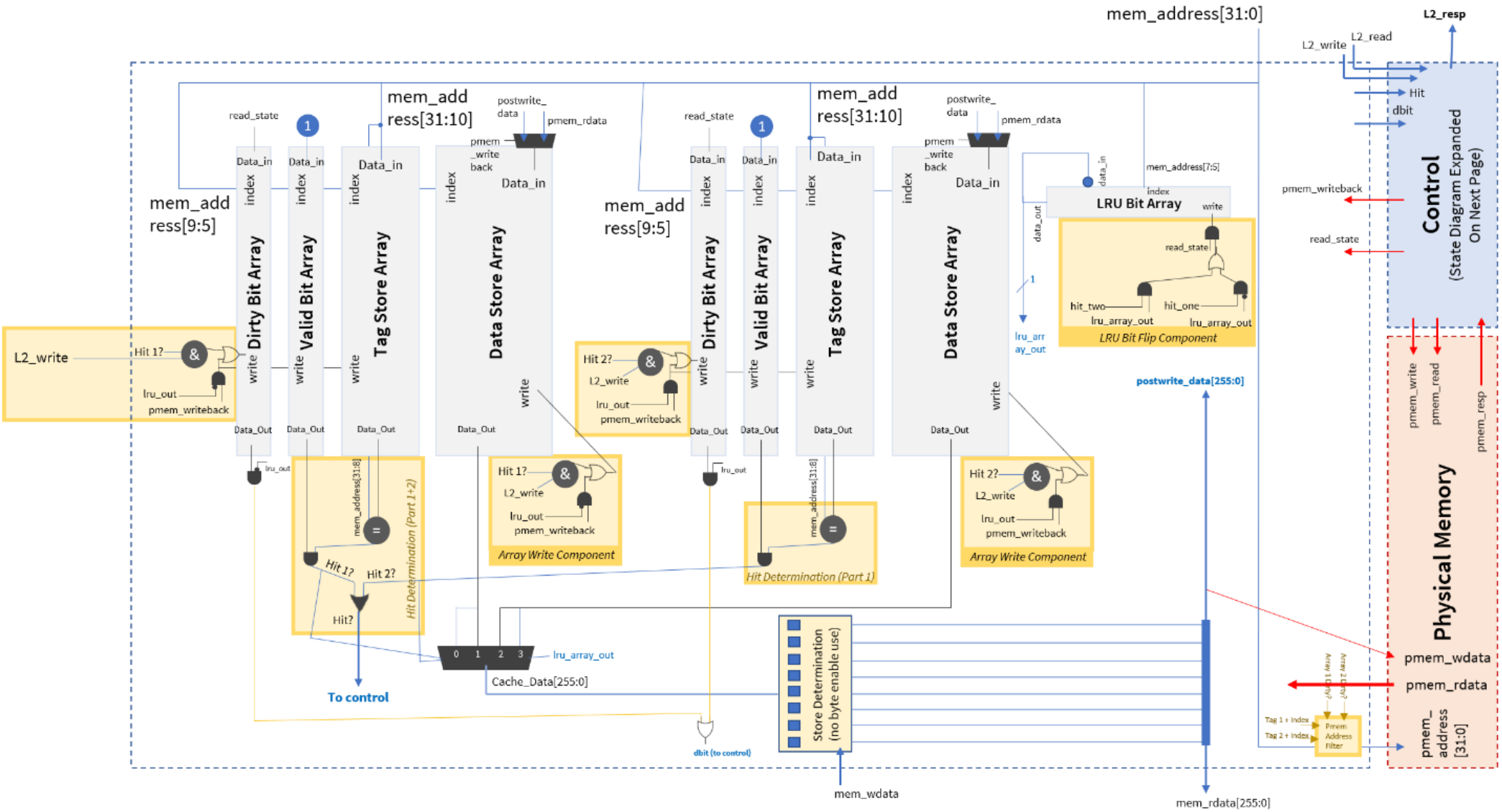
Moving from right to left, we first encounter two 256-bit signal pools that seem to pass through the arbiter, mapped from the cache thus to specific L1 caches. The read data goes to a signal called *L2_RData* from the L2 caches' known *Mem_Rdata*, essentially taking the output of the read data from the L2 cache and delivering it to a write-to-cache-from-hierarchy buffer on the L1 caches. Since both caches may read this data should their response signal go high, L2_RData is delivered to both instruction and data caches. This behavior can be seen with a bridge that is established in the previous section's hierarchy diagram. The mem_wdata signal, however, that is mapped to the L2 cache specifically comes from the data cache however in the form of dmem_write. This is done, as there is no write data that is given to an instruction cache ever, so there is nothing for said cache to pass through the hierarchy or request to be written back to physical memory. On the other hand, in the data cache, an eviction of a value where a dirty bit is still present would require traversal back to

the Level Two cache to update, and at worst (should another miss be found here), in the physical memory structure as well.

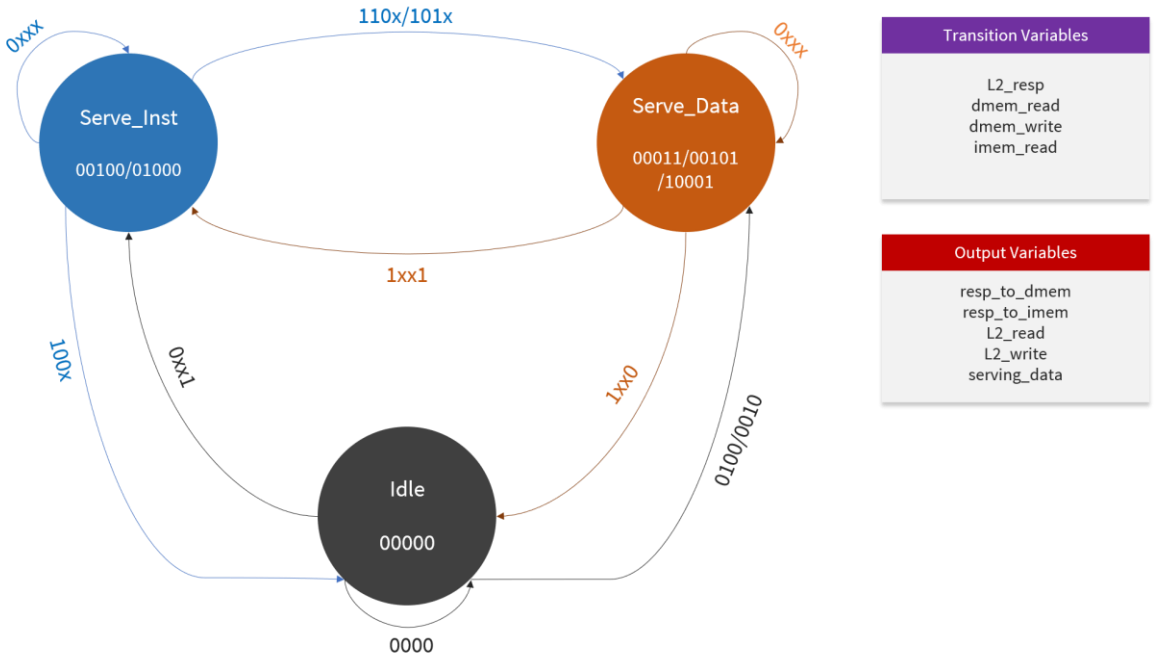
A similar motivation is used for mem_byte_enable when passed to the L2 cache. Since there is no write mechanism that is given to the L2 cache from the L1 instruction cache, but only one that could be possible in the writeback stage of the L1 data cache, the mem_byte_enable can be directly passed to the L2 cache. It is worth noting, however, that this signal will largely go unused, since edits to different-length data controlled by mem_byte_enable will be handled only in the L1 data cache itself. That is, when data is written to the L2 cache, the entirety of the data is taken and written into the address that is defined here. This will require ignoring the offset bits that are used in the L2 cache, thus, which our group speculates will lead to changes in the ultimate L2 cache design. The same can be said for the length of the read data coming from the cache itself, in that the length of the data will now only be 255 bytes long – dependence on the offset bits will have no factor here. As a result, the previously proposed L2 cache may be slightly simplified, where said simplified version is given on the next page. (Indeed, the previous datapath was given more as a guide to the L1 data cache's design. In the new design, you can notice the change by removing logic on the bottom right.)

Defining which address to read or write from, however, yields a problem that again is dependent on the placement and behavior of state – how do we know which L1 cache is being serviced? Since this information is defined by the control itself, we extract it through a bit called *serving_data*, which is high whenever a read or write to data is happening, and low whenever a read to instruction cache (or nothing at all) occurs. Therefore, when serving_data is low, the address that our L2 cache uses (should it need to read, as defined by our control) must align with that provided by the instruction cache, or *imem_addr*. (This assumes that the address passed into the instruction memory is also passed directly to this cache, but for purposes of data alignment, the offset bits are instead discarded here.) However, if we are serving data (the bit is high), then we should instead take the address from the data cache, or *dmem_addr*, and map to the L2 cache. A multiplexer on the serving_data bit handles this conditional operation.

Signals for responses, as stated before, are handled by the control. Similarly, as knowing when to read or when to write into L2 depends on whether one is requesting read from data or instruction cache, state must also be used to control cache signals L2_read and L2_write. Since the responses to these L1 caches may only come when L2 terminates, the L2_resp signal mapped from the L2 cache must be fed thus into the control as well. This is detailed more in the section following the revised cache design found on the next page.


mem_address[31:0]
L2_resp
L2_write
L2_read
Hit
dbit
Control
(State Diagram Expanded On Next Page)
pmem_writeback
read_state
mem_address[9:5]
mem_address[31:10]
postwrite_data
pmem_rdata
pmem_write_back
Data_in
index
write
Data_Out
Dirty Bit Array
Valid Bit Array
Tag Store Array
Data Store Array
LRU Bit Array
data_out
mem_address[7:5]
lru_array_out
LRU Bit Flip Component
lru_array_out
L2_write
Hit 1?
Hit 2?
lru_out
pmem_writeback
Array Write Component
Hit Determination (Part 1)
Hit Determination (Part 1+2)
mem_address[31:8]
Hit?
To control
0 1 2 3
lru_array_out
Cache_Data[255:0]
dbit (to control)
Store Determination (no byte enable use)
mem_wdata
mem_rdata[255:0]
postwrite_data[255:0]
Physical Memory
pmem_read
pmem_write
pmem_resp
pmem_wdata
pmem_rdata
pmem_address [31:0]
Tag 1 + Index
Tag 2 + Index
Pmem Address Filter

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.

Prior to proceeding, it is worth noting that our Cache Arbiter Control outputs five signals and takes in four transition variables – these outputs resp_to_dmem and resp_to_imem, as well as L2_read and L2_write, and then serving_data to differentiate when to pass the address from the data cache or instruction cache (in the control arbiter, as seen in the previous section). These first two will be connected directly to the level one caches of Data and Instruction as response signals, just as pmem_resp was connected to the cache control for the cache designed in MP2. This will satisfy the hierarchy in that L1 caches wait for the L2 cache to return some sort of data and rely on a specific signal to interpret whether this task is completed. The second pair of signals, L2_read and L2_write, will be mapped directly to the states and only allow the L2 to be activated when we are giving service to one of the L1 cache's requests. This allows us to target when a read happens, and to know from which cache a read is being requested, simply by looking at the

The idle state represents the first of the service scenarios presented, in that neither the instruction nor data cache is requesting any data, and the level two cache has either finished returning data already

or is not currently actively reading or writing. We keep staying in this state, as indicated by the *00000* transition, until there is some sort of request from either the instruction cache or data cache to read or write. (This request is a logical OR of signals from dmem_write and dmem_read to give dmem_req, and just the read signal for imem_req since the contents of the instruction cache do not change.) Notice that, by our transitions, we favor the instruction cache if it is requesting on a cycle even if the data cache is simultaneously requesting. We do this 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 – we want 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. For any of these cases, when in the idle state, it should be clear that no response has been made or given to the data and instruction level one caches. The individual service states will handle these responses instead.

Looking now at the *serve_inst* state, there are three scenarios that can happen while in this state: either the level two cache (or underlying physical memory, going deeper into the hierarchy) has not responded to our request to read new data, the L2 cache has responded and the data cache is not waiting for service, or the L2 cache has responded and the data cache *is* waiting for service here. In the first case, we want to ensure that we do not start servicing some other request if it comes in, but instead stay with the current request that the L1 instruction cache issued, so we loop back to the same state and output to both data and instruction caches that no data is ready yet. Again, since only reads can happen at this stage, L2_read is set and kept high here. When L2 does respond, however, we want to bring this L2_read signal down and raise *resp_to_imem*, yielding the output signal sequence 01000. In such a scenario, again, we could either serve the data cache's request next or go back to idle. If either of read or write is on for the data cache, even if there is still a request to read from the instruction cache, we will transition over to serve_data. Otherwise, we will go back to the idle state. In both cases, we ignore the instruction cache's read request since this has not been updated until resp_to_imem is received.

Serve_Data works in a nearly identical way as the Serve_Inst state does, except this time it must distinguish between either a read or write request to pass on to the cache. Since only one of these can be high at a time, we simply pass the values of dmem_read and dmem_write to L2_read and L2_write for the output bits, yielding states 0001 and 0010. Notice that in each of these outputs, there is no response going back to the instruction or data caches. Again, as with the previous state, only a response is given to data cache only when L2 responds, yielding output signals of 10001 (with the signals to read and write from the L2 cache both marked as low after the operation). At this point, if the instruction cache is requesting again, it may be serviced, otherwise the Idle state is reached.

Checkpoint One: 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.

Robert first started by adapting the modules from previous MPs and creating a separate datapath altogether, with the help of the rest of the group to determine the exact design to relay. Robert also coordinated meetings to discuss changes to our initial design that were necessary and lead these discussions to present problems and receive feedback to revise some inconsistencies in thought.

Outside of being the primary "code writer" for this step of the checkpoint, Robert also developed the first set of tests that tracked how many no-ops would be necessary, or where data and branch hazards may occur. Working around these to specifically test the instructions requested at this stage of the pipeline design, he wrote tests that would test the effectiveness of interpreting no-ops and read-after-write scenarios into the same register. He also tested branches after write to *different* registers in his first set of tests, to ensure that multiple instructions could indeed be interpreted at a time when there were no dependencies between the instructions themselves.

In this cycle, Robert also pushed the group to contribute a design that handled all RISC-V instructions already, not just the ones that are available and required to work for this checkpoint. Therefore, he facilitated conversations to handle instructions like JAL and JALR and proposed potential solutions for all dependency conflicts with later forwarding implementation goals. These forwarding ideas are still early in their concept stages, and thus have not been included in this report.

Robert Altman

Robert Altman coordinated most large-scale discussion meetings, and worked most on the design of the pipeline, its components in each stage, and additional module modifications that would be necessary. 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 data should be stored and how no-ops should be treated, cooperating with Robert Jin (and Yan Xu to an extent as well) for register modification.

Due to his experience with timing issues during Machine Problem Two, Robert was also vigilant in checking where similar timing issues might be present in the current design, and where bottlenecks that could constrain certain operation efficiency remained in the pipeline. This was helpful in the design of multiple levels of combinational logic but was also considered based on signal proximity from stages such as writeback that go to a register file in a stage before it. Robert also drove conversation on how to monitor timing restrictions for the upcoming cache arbiter design, and how that might play a factor in the clock frequency needed in the first checkpoint's work (or "front-end" consisting of the pipeline) as well.

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 multiple instructions without dependencies and multiple instructions separated by no-ops where dependencies did exist.

Finally, Robert constructed all control and flow diagrams and parallels for use throughout the conversation. 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 One. 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 crucial in going deep into design questions for our first pipeline, carefully limiting 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 writing code to implement our pipeline, as well as evaluating the simulation results that coalesced, Yan was the one to first 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 Robert's 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 the secondary 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.

Yan also coordinated and began a meeting with our mentor TA, Kenny. Here, we checked on the current state of our design and proposed some ideas for building the Level One and Level Two caches, as well as the arbiter between Level One caches proposed.

Checkpoint Two: 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 L1 instruction and data caches, a cache arbiter, and an L2 cache. This should 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 enable the ability to stall our pipeline on both instruction and data reads, instead of just with the former here, so to accommodate for a physical memory simulation instead of the "magic memory" single-cycle implementation.

Robert Altman and Robert Jin will start most of the process through completion of the arbiter and modified cache implementations. While the former Robert will also guide in modifications to the instruction cache, to reduce logic and potential cycles that are needed in the final design, the latter Robert (Jin) will be primarily responsible for establishing all cache designs as they connect to a single memory hierarchy module. This module will be interacted with as currently done with signals generated to go outside from the datapath. As usual, Robert Altman and Yan will be the first reviewers of this to look for redundant logic in all the controls and caches that are used. Robert Altman will also use this time in parallel to understand how signals are set up on all the caches, to develop a single arbiter module according to the specifications presented earlier in this document.

As noted above as well, this implementation will require stalling at the data memory level due to the presence of a physical cache. This was already handled at the instruction cache level, where a multiplexer was used based on memory response from the instruction cache and a no-op was sent whenever an instruction was not immediately available. A similar stalling mechanism will not, however, work as nicely for this implementation. Thus, Robert Jin will lead development to stop and preserve all signals for all instructions instead of inducing more no-ops while an instruction finishes. In this way, the order can be preserved, albeit at a performance decrease until an advanced design consideration is made for efficiency optimization.

During our cache design, Robert Jin also proposed to the group a possibility to make the current three-cycle cache one cycle faster. Currently, the cache as designed from the previous MP2 (which is the current planned design as proposed in this document) uses three cycles and returns to a read state after updating from physical memory. However, we propose that this is inefficient, as we can guarantee a *hit* already after writeback, and do not have to proceed with a read operation as the data is already stored in the cache on the next cycle. Therefore, the data that will be output from the cache will be the updated data, where any further logic used to complete the load or store is purely combinational. By going back to an essential "Idle" state instead, we can skip this extra "check for a hit" state in reading by knowing we already will receive a hit in the cache (and know what data we are modifying), placing this modified store or read immediately into the cache on the cycle that goes into Idle. To test this theoretical solution, Robert Jin will be leading development for this, with Robert Altman primarily supporting to ensure timing requirements are met and for challenging concept as questions arise during implementation. Yan will also participate in this feedback, primarily giving

reflections on runtime in ModelSim and breaking down stability of values at cycles after memory response signals are given here.

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 on loads and stores where necessary, and that there are no data hazards that are encountered. 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 a result, Yan and Robert (Altman) will meet and lead development of the full test suite used on our pipeline. Robert will focus specifically on checking various types of instructions, especially wanting to target sequential loads and stores, according to the limitations of our current design (that is, acknowledging data hazards and placing no-ops still where needed). 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.

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, October 30, 2018 from 3:20 pm to 5:00 pm, outlining some of the initial concerns we may have about data memory stalling approaches and even approach for designing our memory load mechanism with respect to future implementation of advanced features (such as memory stage leapfrogging, which is in discussion at the moment). 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, October 31. At this point, we would like to have the arbiter and caches implemented with a basic stalling mechanism proposed using a single stall signal (discussed in previous sections). This should also be where the first evaluations of performance using ModelSim should take place for an at least partially-implemented checkpoint two. Thursday will be used with split time, 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, 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.