MP3 Final Report

ECE 411: Computer Organization and Design

Room 3032

Robert Altman / raltman2

Robert Jin / naiyinj2

Yan Xu / yanxu2

Document Contents:

Introduction

The primary interface of working with computation resources is handled through the processor core of the system, where microinstructions are interpreted and distributed to the remainder of said resources. On a traditional Von Neumann model, much like the ones used by LC-3 architecture, input/output and memory interaction is handled off-chip with a register file and arithmetic processing unit kept handling microinstructions (given instructions could be pulled from memory and set into a register as well). The RISC-V instruction set architecture can be supported on processors like the RV32I, proposed by the ISA's creators, to run on such a model given a direct means of mapping the processing unit to memory and write and read ports. RISC-V architecture offers 31 writeable 32-bit registers and a 32-bit instruction set, with support for byte-addressable memory. Created at the University of California at Berkeley, the ISA has become a tool used in universities and industry[1], even with its own C compiler that allows evaluation of processors supporting the architecture for certain workloads given power, area, and performance design constraints. The ISA has also been entered into commercial products, where new techniques in processor design considering dynamic power tradeoffs can be explored.

At the beginning of ECE 411 at the University of Illinois, such an architecture was built in a single-stage processor constrained at 100 MHz clock frequency with the area of a Stratix III FPGA. While full functionality could be exploited on the processor, these initial efforts were nothing more than proof-of-concepts of another instruction set architecture meeting design constraints, until a single cache memory hierarchy was established later in the course. Though significant frequency tradeoffs occurred in some personal designs, this opportunity allowed for us to observe the detriments of waiting for a single point of weakness to return data, amidst an already lengthy process that executes instructions in-order. While our designs may have been tightly fitted to hold all our resources, it turned out that these designs were never truly using all the resources across each stage of an instruction's execution.

If there is an entire application that should be run, or perhaps competing threads trying to use system resources whose instructions are interweaved and running through microinstructions on the dye, then we will want to increase the throughput of our entire workload to improve performance. This is done instead of latency of a single task, as the number of resources and pattern in which they are execute still does not change, but there is still a recognition that not all these resources are in need or even being used by every instruction at the same time. We can thus start to break down the process into fetching, decoding, and executing, as with a typical ECE 120-esque LC-3 Von Neumann architecture. Acknowledging that memory reads from instruction and data can be considered separately as well, but also that latency induced in connecting to this memory hierarchy is coupled with a typically large critical path, we also further separate the number of stages to give memory storage its own consideration – a separate stage of "execution" so to speak that is only using memory. A fifth stage can thus be used to then write back to a register file that is present, a different component only accessed in the decode and very end of instruction execution, and thus representing another stage. Therefore, five stages emerge as possible given the resources required for proper

[1] As noted by ECE 411 course staff in the Piazza post, @7, *RISC-V Project Signup and Questions*.

operation in a RISC-V architecture. Thus, to allow for this larger amount of throughput at (at least) the same spatial and timing constraints as previous processors, we present a five-stage pipelined processor akin to RV32I that allows us to compare performance increase through this throughput increase as compared to single-cycle and multi-cycle, single-stage designs. Indeed, while this still leads to some instruction latencies that may come about, we can take advantage of each of these resources being utilized in different cycles of a multiple-cycle CPU and take advantage of abstractions already used in control state logic to move between the cycles to define storage of multiple instructions down the pipeline.

The influence of completing this pipelined example is purely in the name of speed, where the ideal speed up gained is typically five times that of a single-stage design with throughput of a single instruction per five cycles (at best). In completing such a design, however, we also get to witness the need for data forwarding and stalls within the pipeline and explore where code may be optimized to run faster if multiple registers (for example) are used. This will also explore tradeoffs between physical and architectural registers by doing so, where our design will treat them with a one-to-one mapping but consider the order in which they are updated, as well as availability of the data during any of the execution stages that may depend on it. By completing this design and addressing these challenges, thus, we will investigate benefits of various pipelining and unrolling techniques of software instruction-level programming by running the instructions with hazards ourselves on our own hardware, and analyzing number of stalls using a series of counters, etc. This will lead to an appreciation for how these hazards can invoke latency where forwarding the data is not enough to ensure correctness and give feedback on processor performance under these types of workloads.

Alongside fulfilling this processor model on the "front-end" through the pipeline, we also want to ensure that our architecture resembles that of a modern processor design in the memory unit. These processors typically keep data close if it is still being used throughout the pipeline, in caches that are connected to physical memory in a memory hierarchy. By implementing this hierarchy, processors can achieve temporal and spatial locality to reference the same or near memory locations many times without the latency of going far off-chip. A typical structure has thus been adopted (and mimicked in processors like Intel Nehalem, albeit with three levels instead of two), where the CPU interacts directly with instruction and data level one caches, then a shared level two cache, and then main memory (Kumar[2]). Handling requests from two parallel caches, however, also must be achieved through an arbiter that sits at the level between L1 and L2. Thus, to optimize both this temporal and spatial locality using the pin assignment and space limitations of an FPGA, we are extending the same architecture and presenting it here for the memory hierarchy structure employed in our design.

Completing this pipelined processor and memory hierarchy structure, even in complete disregard of power and energy tradeoffs that emerge from using up to five stages in a pipeline for fetch/decode/execute, is a critical task for developing remedial systems and processor cores as witnessed since the early 1990s. Through investigating imbalance between the stages, but the overall speedup afforded under multiple workloads with different instruction structure, we can thus gain an

[2] As noted in Lecture Slides for Lecture Four of ECE 411, Fall 2018.

appreciation and learn further the tradeoffs that are considered when designing processors for a given environment or shipped setting.

Further, this project allows us to attach other modern affordances such as branch prediction into multiple places of the pipeline, allowing for extension of common memory and data prediction schemes found in today's processors by offering a platform in which they may be included and useful. Therefore, our project includes a branch target buffer and global (stateful) g-share branch predictor as well, a dynamic branch prediction model based on history that allows for even faster execution of known addresses that are jumps, branches, or unconditional movement within subroutines. These features are critical and bundled into nearly every type of processor in use today and offer significant performance benefits that also modify how many instructions are read, and in what order delays may occur, should an appropriate scheme be determined. Therefore, design of this processor and integration of the branch predictor helps us to see this scheme and differentiate between highly iterative and highly localized code in terms of performance to see tradeoffs and delays invoked by having features meant to increase performance in the pipeline.

Further high-level insights of the project can be found in the next section. Details about each advanced design feature used to optimize performance via throughput are also given throughout the report, with a level of progress first detailed regarding designs of just the pipeline existing itself (while assuming no hazards). From this initial pipeline model, we will then append caches and memory interaction, and simulate the results with a delayed-response "physical" memory on the third level of a memory hierarchy to show how much of instruction handling is still at the mercy of periodic (about every eight instruction) reads far off-chip to DRAM-based memory (or worse, to disk). After this, we will tighten the delays invoked in software-level to deal with hazards, and force these to either be eliminated via forwarding or correctness be insured by stalling the pipeline. After this, a scheme for measuring how many times this stall happens, especially if on branch misprediction as well with only a static scheme, will then be explored with the addition of performance counters. An eviction write buffer will also be added for moving data into a separate level of cache should memory need to read immediately after data has changed and needs to be stored. This allows us to reduce double penalty per instruction, effectively cutting our stall time in half on some data reads if data has been changed through program execution (but only in some scenarios). Finally, the addition of this aforementioned branch predictor will be given in global type in conjunction with a branch target buffer, to allow for only a single state to be needed in initial branch prediction. This allows instructions to start immediately and further increases throughput, unless later determination of the branch or jump proves otherwise here.

Project Overview and Group Evaluative Review

The pipelined implementation of the RISC-V RV32I processor presented is a five-stage processor with a three-level memory hierarchy. In addition, it offers support for branch prediction, instruction forwarding, data hazard management, and early branch detection. While doing so, the second layer of memory is extended to four ways to decrease latency from reading from physical memory in our design and is also attached to a single line eviction write buffer to avoid servicing writes to physical memory unneeded during a cache miss.

An introductory goal, prior to development of the entire design, was to construct a microarchitecture that could run computer-executable programs and load and store data on the same system, packed into a series of instructions as connected via memory in the Von Neumann model. However, we found that multiple instructions had little to no dependency on taking a full set of cycles to complete, where the only program instruction running on a processor corresponded with a single instruction in memory. We found that we could greatly improve performance by breaking down the components of the processor's execution into multiple stages, using the original control design for a single-cycle processor (MP1, MP2) as a motivator. Therefore, our core motivation of designing this pipelined processor with advanced design features was to improve performance of programs, benchmarks, as well as data storage and interaction as highly as possible.

In addition, our design was built for the intention of running a small operating system or dedicated system binary that could be used on the system. This would require knowledge of state and proper eviction of data and instructions should multiple programs force context switches or branches at any current setting. Some of the phenomena that arise from this, such as the ability to write back data on evicting a cache line unrelated to the previous application or subroutine, also motivate our initial goals to improve data and memory management performance. For designers to check their programs and locality issues that may cause performance issues in this processor iteration, we thus added performance counters that were directly mapped to lower memory addresses. This was done in the style of interacting with buffers, as done with the keyboard buffer in architectures like x86.

From the very start, our pipeline was focused on reducing cycles taken for instructions as we determined which resources could be used in certain places. Our instruction fetch and memory stages, due to the known latency that will be incurred due to a memory read from our experience with cache additions in the single-cycle design, were deliberately kept empty unless components could generate signals without connecting to cache critical paths. A core feature of our processor was completing any arithmetic tasks that might be needed in a register, branch, or memory computation, and therefore we completed this in a stage of its own – execute. Instruction decode was also kept clean but outside of the instruction fetch again to meet the goals of reducing latency and keeping stage functionality isolated from each other – its signals for the rest of the stages were computed from latched values coming from the instruction fetch stage. Some of these cycles allowed for instructions to flow through the processor with results happening at much faster rates, given at first that there were no hazards that arose in receiving data or having registers ready for new data to come. Once we found these hazards arose, however, there was a motivation to make some hazards clear even faster – including additional latches used from some writeback stages and PC updating that could arise from

EX stage. (The movement of early branch prediction to our execute stage was also inspired by this need, which is discussed in more detail later on as well.)

Unfortunately, this logic-driven performance improvement approach meant that there were complications often in keeping track of the maximum frequency of the design. As we looked for new tricks that would save ourselves from doing further computation or pipeline stalling later in our design, we often would find ourselves complicating some combinational logic or working between stage signals and complicate critical paths as a result. When we first started our design, our pipeline signals were well organized, much by the help of Robert Jin and communication among the entire team on each signal that would align with our presented initial pipeline draft. Frequencies were as high as 171 MHz in some early iterations, but our main concern was with frequency drops incurred by adding our memory hierarchy (in particular, two levels that include complicated cache logic). Thus, we often found ourselves designing to reduce the amount of combinational delay found in the memory hierarchy (including the arbiter) itself, all while neglecting the changes this caused when tracking stalls or adding advanced features around the caches. As a result, we can only report a maximum frequency of 112.1 MHz at this time, a number below the 120 MHz threshold that we found ideal given the number of cycles cut by our processor. Had we paid more attention to the layout of our pipeline beyond just the initial designs, especially as forwarding was integrated into our circuit, we feel that flaws leading to additional latency could have been easily avoided. That is, our current critical path remains found in the pipeline itself at this time.

This, however, was the only major downside of any organizational or administrative aspects of this project. Work was typically completed through a rigorous design phase to start it, which would be heavily documented typically with the help of Robert Altman in formal reports. Yan Xu would typically coordinate meetings and deliver initial talking points during our sessions, while Robert Jin would produce his latest findings from any SystemVerilog implementation of our design. After initial designs were given and the team came together for a whiteboarding session (where all typically participated in physically drawing out the circuit design), each member would take time to reflect on the recent meeting and prepare notes for a next meeting to follow. The process was typically made faster through Robert Jin's work on laying out signals and intermediate combinational logic of each next step in design, such that initial complications could be resolved by this secondary meeting. Therefore, it was the very focus on group communication that allowed us to complete designs largely to the performance metrics we had in mind, and most always group members had an idea of what each person worked on, and *why* the feature was beneficial. It was this *why* that continued to motivate us throughout the entire design, and allowed us to split up work based on personal interests as well.

To further break down on a person-by-person basis, Robert Jin was our primary code writer and typically presented most complications or optimizations from an application point-of-view. When it came to recapitulation on past commentary, Robert could offer hard data and tendencies of our design to raise concerns and questions that saved us valuable time. By staying close to the design itself as implemented in our design files presented for this project, Robert was also able to see initial performance metrics. Therefore, many of the datasheets as well as unexpected results were typically presented through Robert first, many of which prompted additional meetings that kept all in the group busy determining design changes.

Where Robert Jin would typically work on later phases of our design work, Robert Altman was responsible for getting the group started with formalized reports, meeting notes, and presentations. Almost all the physical resources. Because of his high integration into the initial design process, Robert typically would be a primary test writer for much of the design's components. This included unit tests, for example, that would display certain inefficiencies (such as warm up delay in branch predictor) or highlight proper pipeline and memory data flow (such as with evictions in a four-way L2 cache) during project development. Robert was also responsible for his first interpretations of the report as well, typically giving both formal and informal notes on what each outside or new test code may evaluate.

Finally, Yan Xu was primarily responsible for initial high-level design questions, but also for presenting corner cases in the form of tests and notes in meetings. The former, tests written for the implemented design, were critical to catch inconsistencies such as sequential loads and store stalls where original tests may not have found them or covered them (and instead focused on data integrity, following this example). Yan was also the primary author of the four-way L2 cache extension, which will be discussed later in this report.

Because of this coordination, we can present multiple advanced branch prediction and determination features that satisfy initial performance goals of building this pipeline, where highly iterative test code or return-oriented programming is consistently less than 900,000 cycles of operation. In addition, we also present a three-level memory hierarchy with level one instruction and data caches, level two cache, and physical memory at the third level with eviction write buffer interface intercepting writeback requests. Perhaps most notably, we were able to reduce original runtime from a fully forwarded model of the processor without advanced design features by 35.9%, as computed by a weighted average of percentage improvement from end execution to original.[3] When compared to runtime against a model with cache that is single-cycle, the improvement becomes nearly 7 times higher by cycles per instruction here[4].

An overview of the core features, as well as the pipeline stages themselves, can be found in the design description overview on the next page.

[3] Computed by ((1326/858.5) + (312.1/285.5) + (1009/879.4) + (1234/747.4))/4.

[4] The end value that was found as an average was 3,933,655 cycles, compared to our 547,245 average.

Design Description Overview

Ultimately, through analysis of this control, the components of the design could be pipelined into five separate stages. The first is Instruction Fetch (IF), which connects a program counter to a level one instruction cache in the memory hierarchy, pooling for the next instruction that shall start executing in the pipeline. True to the stage name, the instruction is then retrieved from memory should data be available. If not, a stall is induced in the pipeline for sending new instructions in, but other instructions through other stages in the pipeline may continue. Additional logic is later added to predict whether this instruction, just by its placement in a program or corresponding memory address, is related to a branch or jump that will be taken. Thus, our branch predictor and branch target buffer that holds destination data is also found in this stage and will be discussed in further detail shortly. Finally, stalling and hazard logic are also added in this stage given its ability to send instruction data to the rest of the pipeline, where a no-op is sent through the pipeline whenever a hazard or memory stall of some sort is detected.

The second of the five stages that was added is an Instruction Decode (ID) stage. This was separated from the instruction fetch stage not only due to its isolation in earlier design control, but also to reduce the amount of logic and combinational delay already induced from memory access in the IF stage. This decode stage isolates all the signals that must be given to various components further in the pipeline, whether this include memory signals for read and write, multiplexer signals to read from an ALU, etc. The primary component of this stage, unsurprisingly, is thus a control signal determination component that reads the instruction received from the last stage and determines this logic through a sequence of combinational logic. Otherwise, data is forwarded about the instruction itself from latches that straddle the IF and ID stages to ones that are between the ID and MEM stages. Perhaps most importantly, however, this is the state at which our register file is located, such that we have data available for computation in the execute stage. We can directly read, based on the operation decoded from the control signal determination module previously mentioned, values from the first source register (RS1) and second source register (RS2) if desired by the operation. However, this register file was also responsible for handling writes back to the registers after a value has been determined and signal has been raised from the writeback stage. That is, it would receive a signal hat was present from this register file in the writeback stage to tell the register load, where the result of either a memory read, ALU computation, PC-based value, value of a comparison (for instructions like *slt*), or an immediate value based on the past instruction. The value is sent through a multiplexer in the writeback stage, where all of these values are prepared and kept at the end of latches.

The previous stages, thus, handle all the fetch and decode portions of the fetch-decode-execute cycle typical for architectures following the Von Neumann model. The only item left was execute, but since this consisted of utilizing various resources given different steps, was split into three states of its own. To start, the *Execute (EX)* stage consisted of two forwarding multiplexers that would either take data as computed from the current execute stage for the next instruction, data from the memory stage that either passed through or was found because of a load, or data that was about to be written back and was given to the register file that is one stage too late otherwise for the currently executing instruction. Forwarding logic is then added in this stage based on which register is assigned for reading in the EX stage, and which one has been written to in the last EX, MEM, or WB stages. If any of these registers match the value of either RS1 and RS2, are not the zeroth register (which always stays at value zero, according to RISC-V's Instruction Set Architecture Manual) and are being used (the instruction loads from RS1 and RS2 values), then a forwarding conflict is raised depending on the stage taken.

Execute is also responsible, after knowing this information, for parsing through any immediate values or register values that must be added to a different PC or register value. Therefore, either RS2 or a handful of immediate values based on the type of instruction (e.g. I-Type, J-Type, etc., all of which are again enumerated in the RISC-V Manual) are taken as the first operand for an ALU out, multiplexed based on a select bit that is determined by the control signal extraction module previously mentioned. A similar process is used for deciding between PC and RS1 as the second operand with a second multiplexer. The values are then taken and computed according to a specific arithmetic operation (defaulting to add, but determined by the control signals again) before the data is stored to a latch.

During this ever-important execute stage, however, there is still one more item that must be evaluated – if a branch is to occur or not, and where said branch (or jump) would occur. To find out if a branch is happening, a comparator module is used between sr1_out and sr2_out or a certain immediate value if instead parsing an instruction like *set less than immediate*. If the branch signal is high and a branch indeed is found, then the PC back at the IF stage is notified of this request to branch. Similarly, if a jump is known to happen, the value at the end of ALU's output can be retrieved and fed to the PC for the next instruction.

However, we should only update the value of the PC here if the value was mispredicted from the BTB and branch predictors found in the IF stage. For example, if the branch or jump was predicted not taken, the PC should be updated based on the value from EX stage. Similarly, if a jump was predicted but the destination did not match our output value (due to a change in the base register, for example, with *jump and link register* instructions), the branch should be marked as mispredicted and the phenomena corrected. Finally, if br_en is low on a branch line because of register comparison, then we should also mark a mispredict and instead just give back PC + 4 (the next instruction, indicating that there was an item not taken).

Our fourth stage in this pipeline is the MEM stage, which consists of its own forwarding logic as well should a load and store happen right after each other. Since the store must wait for the load to respond, and the data for the store is provided otherwise typically from the EX/MEM latch containing SR2, we also forward the memory value from the last memory result that was present and inject a stall. The stall is given due to the data needing to be stored on the other side of the MEM/WB latch containing memory data, which itself gets filtered through an MDR Multiplexer that sign-extends or truncates bits to get a half-word or byte-word representation of what was just read.

The fifth and final stage, then, of the pipeline was our writeback stage. This uses the same multiplexer for data that the execute stage filtered for EX data forwarding, with the exception that one potential output is now the value that was extracted out of memory. This output is directly sent to the register file without a latch, as the delay in determining which logic to take from the other stages is minimal. However, the data is still fed to an input latch for the WB forward case in EX discussed earlier.

To continue in our efforts to drastically improve runtime of programs that run on our pipelined processor, we also had to apply optimization to data reads and writes with components that were distinctly off processor – our memory hierarchy. The first layer of cache, or level one, used two distinct caches for data and instructions that different stages could interact with. This led to less resource contention if hits were available, or if one cache led to a hit while another missed and had to search farther down the memory hierarchy for data. As a result, there were less stalls incurred for both memory retrieval and for instruction retrieval, but one still needed clear prioritization over the other. In our design, we decided to award this prioritization to the data cache, which can stall the entire pipeline if not available since no memory instructions can leapfrog past it (without specialized logic to do so).

Therefore, if a memory request reached the second layer of our memory hierarchy, it would first pass through an arbiter to determine which cache miss to service first. Again, by servicing the data cache, we could resume the pipeline to at least clear out instructions from the ID stage to beyond, even if an instruction cache miss results that prevents update of the program counter in later stages. Otherwise, we should treat this on a first-come-first-save basis, where we service the first cache that has a miss and do not disturb the process to ensure data integrity.

All of this should interface with a larger level two cache, which is meant again to provide faster data return on a hit present in the cache instead of interfacing directly with memory. The cache was set ultimately as a four-way, pseudo-least-recently-used, set-associative cache with 8 sets, kept small in the number of sets still to meet area constraints but made wider to allow for multiple ways that would have fallen under the same index. Utilizing this cache as our level two again allows for direct lookup, with response times for found data as fast as a single clock cycle (about 9 ns, at our current frequency rating). The rest of the connection is then handled with an eviction write buffer to hold writeback data and enable reads to complete to bring in new data (prior to writing to physical memory, which no process needs to wait for unless accessing the same memory line). After this, the physical memory is attached to the system.

In all designs throughout the processor and memory-based system model afforded in this project, performance and reducing the number of cycles has always been a primary priority. Therefore, extracting as much parallelism out of components as possible in multiple stages and across the level one cache (as well as inside the caches themselves for tag matches), and reducing the amount of latency incurred by going off-chip (to physical memory), we are able to present a heavily modified processor design that substantially decreases the number of cycles per execution of various benchmarks, operating system code, and general programs.

Checkpoint One: Initial Pipeline Design Without Forwarding

Our initial pipeline design aimed to disperse the components of our former, single-stage processor into the context of five stages, each with latches that contained information that may be useful later in execution.

We start with the IF stage, where a connection is first placed to the instruction cache based on the contents of PC as the imem_address. The current memory model that we start with is a two-port magic memory interface, which allows for single-cycle responses from memory based on a request to either instruction or data cache. Hence, the model works based on a lookup into a highly enlarged register file, which is synthesized as RAM logic but assumed to have no delay in initial timing. Should the memory not respond right away, however, we do have an attached imem_resp signal to a multiplexer, which either allows a *no-op* in the form of *addi x0, x0, 0* to pass through, or the data from memory if indeed a response signal was found. The opcode is then fed and separated at an IR latch for IF/ID stage, and the PC is also given to the rest of the design in a latch (which will simply be passed on in the rest of the design from this point).

As defined earlier, the control signal extractor and register file are the only components to add now for ID stage. The control signal extractor will use the items in IR to determine what opcode is being used, as well as any additional function parameters packed into the opcode such as *funct7* or *funct3* (which can drive the logic of the ALU, for example, from addition to XOR). The register file itself is also connected to the instruction register contents, where the data is separated into rs1 and rs2 to get the most recent data from the register. For now, it is this data from rs1 and rs2 that is then passed into a latch interfacing between ID and EX stage – no forwarding is assumed in this initial design. The IR is again also carried to the EX stage, as it will be needed for evaluating whether a branch was taken (early branch resolution will happen in this stage, which will be discussed as an advanced feature later in this report). Finally, the destination register, as well as the control signals that are generated by the extractor itself, are passed in to direct which resources need to be used for proper handling of this instruction.

In this stage of the processor's design, there is no forwarding that is enabled, and thus our logic can be simplified for what is brought into evaluation for the ALU. That is, when we look at the value of a register, whether it is RS1 or RS2, we just use the values that were in the latch between ID and EX stages for these values. Later, we will have to deduce a mechanism that brings this data from different stages, which should involve another multiplexor that chooses between latch register data and MEM/WB/EX register data. The same must be done for the comparator logic that determines if a certain condition evaluates to true, such as a condition used from branching. This signal is matched and ANDed with the value of the opcode compared to that of branch, saved in *br_en* if there was indeed a branch that could be performed here. The jump enable signal is just a control signal and is taken from the control word. All other signals that previously came from the ID/EX latches are also passed into EX/MEM latches for the next stage of the pipeline as well.

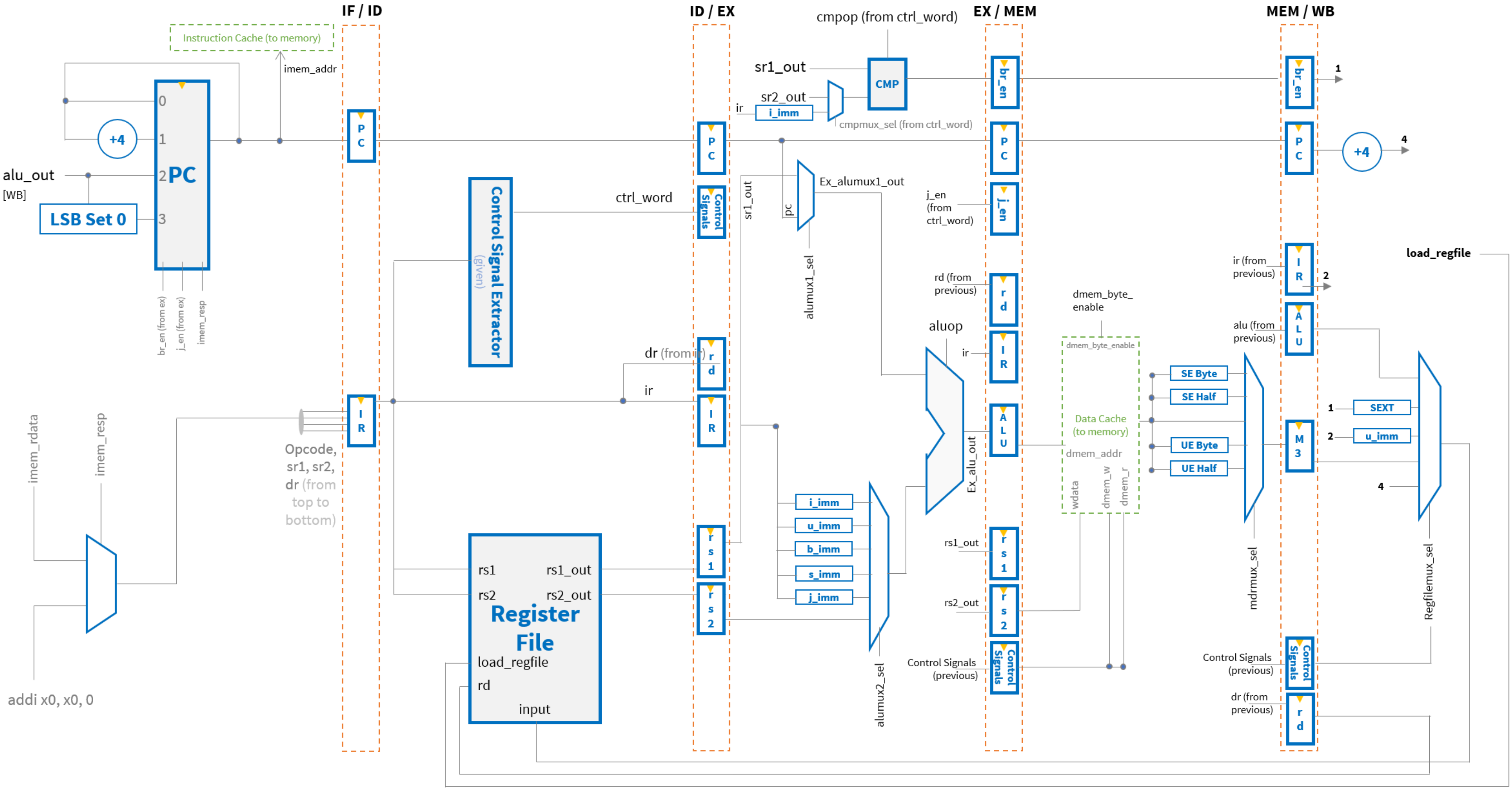
Indeed, this memory stage is perhaps the most lightweight of them all at this current state, as the data cache can be access via a simple lookup. As a result, there is no stall logic that we need now since our data should be available within a single cycle, managed through the MEM stage. From here, the data that is output goes through a filter to truncate data to half-word or byte content, dependent on the control signals that have been passed on from the ID stage, to EX, and now to MEM via a control signals latch. Once this is done, the result goes into its own register named *M3*, which signifies that its latch will be used by WB at the third (zero-indexed) input of the register file multiplexer in the writeback stage. The value from the ALU in the execute stage is also passed along, as is *rd*, control signals, PC, and branch enable, which will be used either for data determination or for loading the register file.

Indeed, the destination register is directly connected from the writeback stage to the register file that we have present in the ID stage, signifying where any new input may come from after it has been evaluated and needs to be written back here. The data itself must first be evaluated after a series of operations that may come from memory loads or EX stage resources. We can send the value of an unsigned immediate, in the form of *u_imm*, directly to the register on instructions like AUIPC – this is connected to pin 2. A sign-extended version of the branch enable is also taken and given as possible input, where this value can be used for writing the conditional result of evaluating two registers (as in a *slt* command here). One may take the direct ALU output instead, such as in an add or AND instruction, and thus the data passed along from the execute stage is option 0. Finally, option 4 is the value of the PC kept passed from the IF stage, where the next value is 4, stored on JALR and JAL commands where the next point of execution to return to is kept.

There are a few miscellaneous notes to pass about this datapath before continuing. Note that addi x0, x0, 0 is one of the options fed to a multiplexer, along with the data that carries the instruction to evaluate. If we don't have a response from memory right away, we still must provide some values throughout the pipeline. Thus, we feed a no-op instruction so that cycles where no execution is happening do not do anything destructive or unpredictable – latches at each stage always have values, and our cycles where we are waiting to still read a new instruction will induce delay while completing a harmless instruction. Also, branch enable and jump enable are taken from the latches available after the execute stage and used for computing the next PC. This allows us to more quickly allow for evaluation of where our next instruction is, saving us some cycles (as it is likely an instruction cache miss may happen as well).

All signals like regfilemux_sel, mdrmux_sel, and others are part of the control word ctrl_word that is passed to each stage's control signals. Also, the comparator is placed in the Instruction Decode stage, as we can evaluate br_en immediately after knowing SR1 and SR2 – there is no combinational delay. Therefore, the br_en can be set earlier, which we exploit as noted in the early branch determination in EX stage section. A pipeline of the work completed in this checkpoint may be seen on the next page.

Revised Datapath Diagram



Checkpoint One Progress Report Summary

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 like 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.

Checkpoint Two: Overview of Cache and Arbiter Interaction

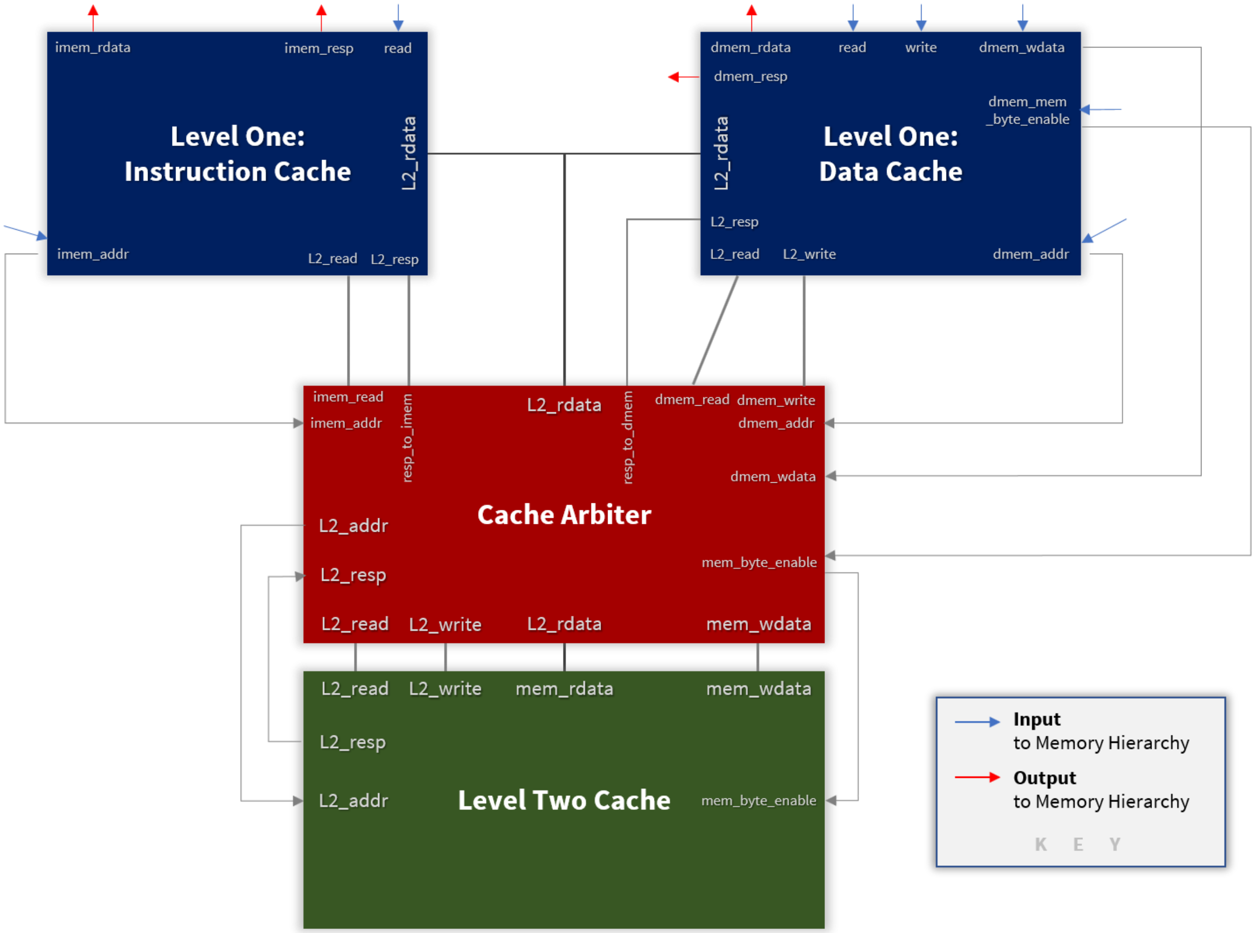


Figure 1: A full view of the caches and arbiter in black-box form, signals labeled.

In the next checkpoint, we built 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.

Checkpoint Two: Level Two Cache and Arbiter Design

The single Level Two cache found in checkpoint two's 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, we initially wanted an L2 cache to 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.

There were some critical differences, however, between this model of 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.

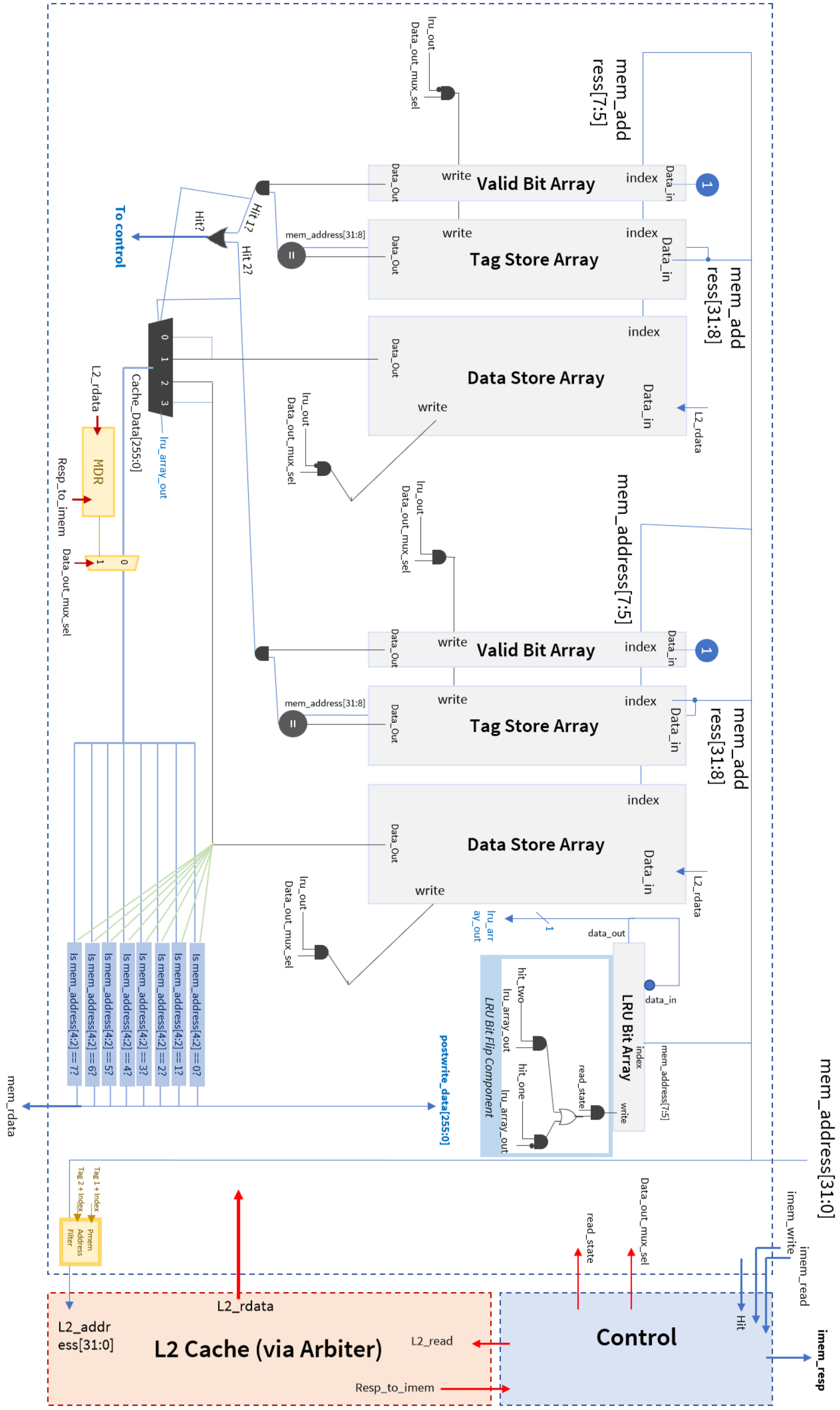
Starting with the level one instruction cache, we already discussed in the previous report that the use of any write signals could be removed from this specific cache. As a result, this would allow removal of the dirty bit arrays that were used in the cache. We did not, however, extend this to include 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.

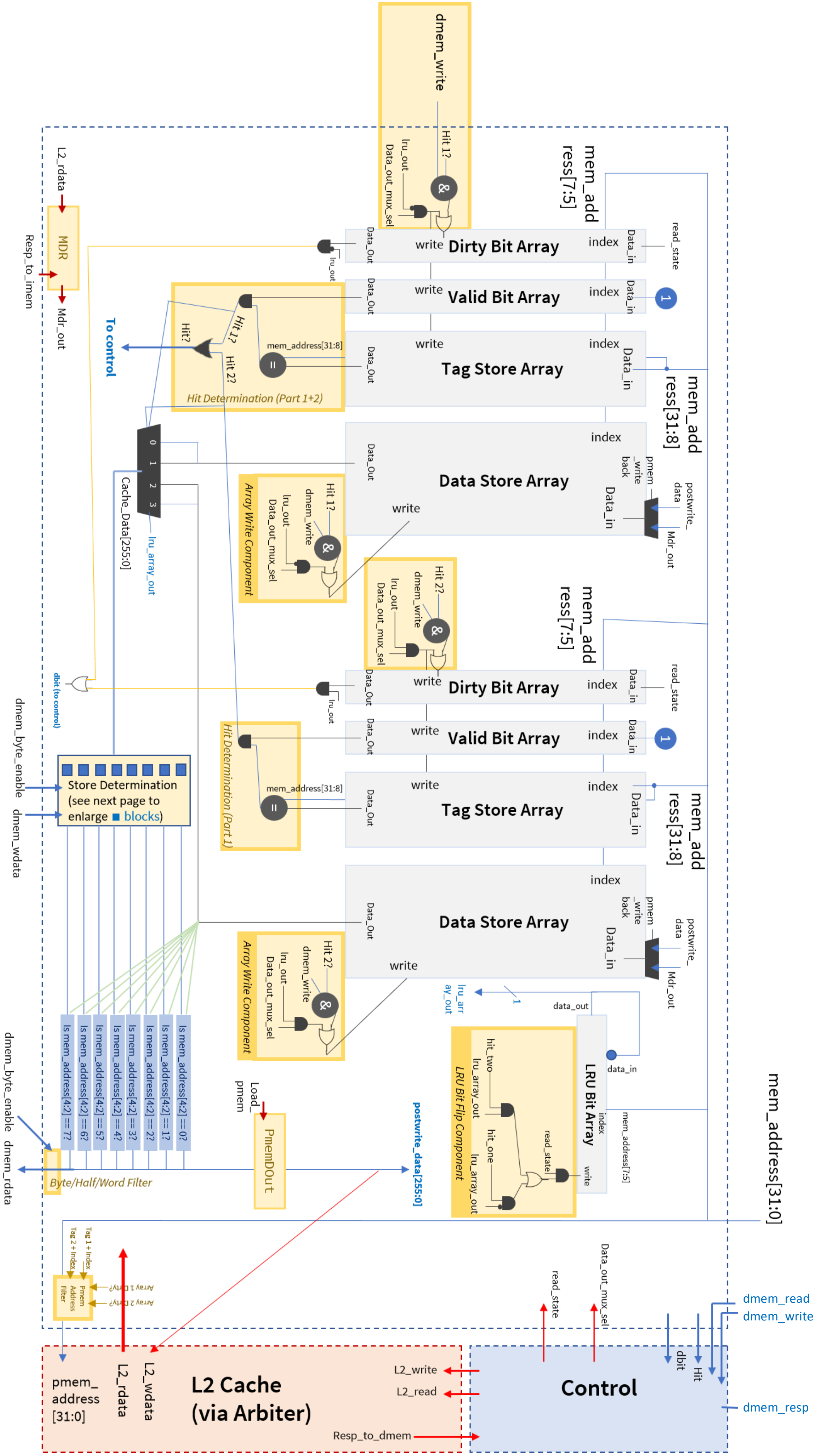
The updated hierarchy, thus, for the level one instruction cache is provided on the next page.


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

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.

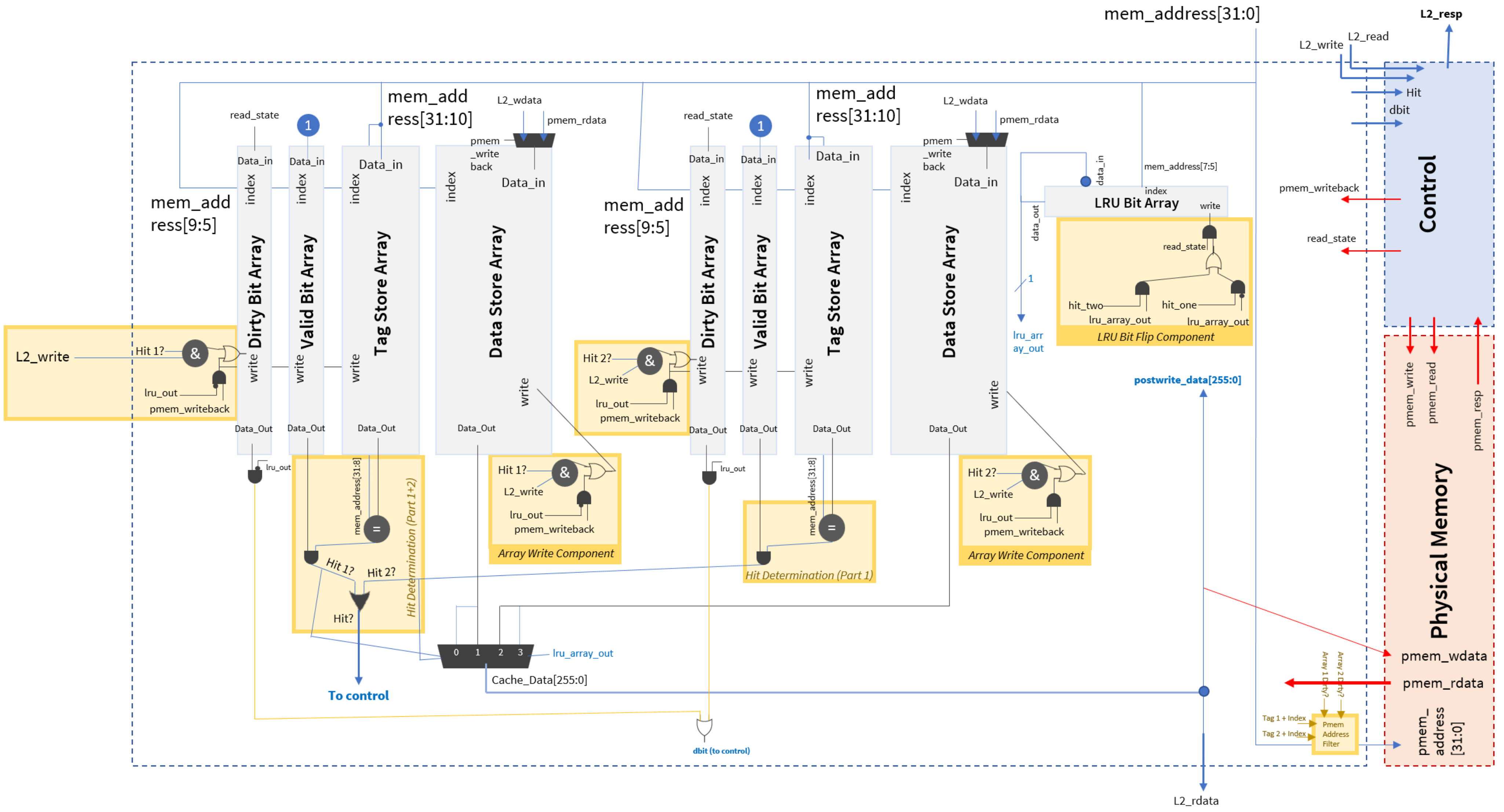
This leaves emendations that were given to the Level Two cache, outside those that were already discussed in the previous report. Previous discussions included a removal of mem_byte_enable, as well as a removal of all offset logic and a direct reference to write data being input back into the data array. 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.

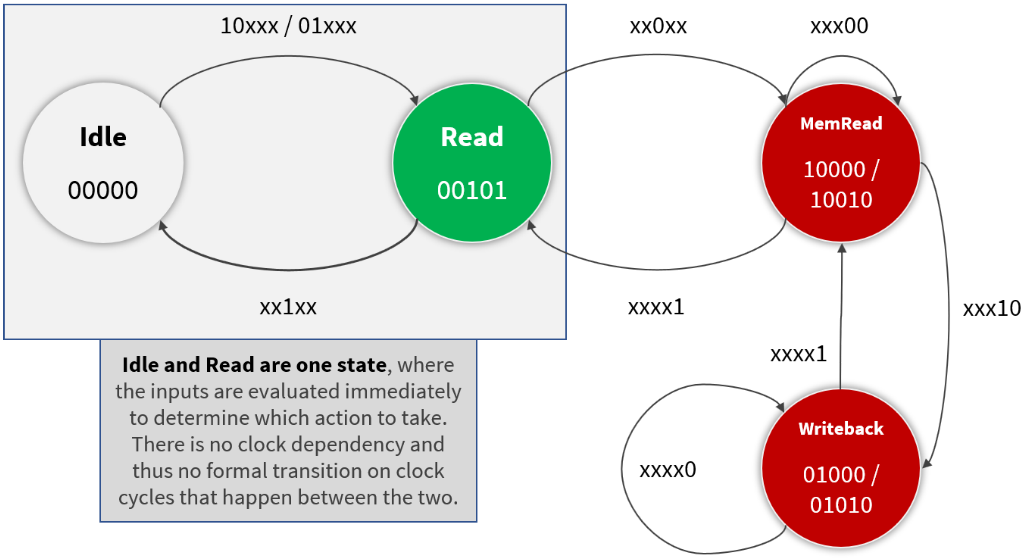
The two designs that follow these descripts correspond to the level one data cache and the level two cache, respectively.



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

1.


mem_address[31:0]
L2_resp
L2_write
L2_read
Hit
dbit
Control
pmem_writeback
read_state
pmem_write
pmem_read
pmem_resp
Physical Memory
pmem_wdata
pmem_rdata
pmem_address[31:0]
L2_write
Hit 1?
&
lru_out
pmem_writeback
mem_address[9:5]
read_state
1
mem_address[31:10]
L2_wdata
pmem_rdata
pmem_write back
Data_in
index
write
Data_Out
Dirty Bit Array
Valid Bit Array
Tag Store Array
Data Store Array
lru_out
mem_address[31:8]
=
Hit 1?
Hit 2?
Hit?
Hit Determination (Part 1+2)
To control
Hit 1?
L2_write
lru_out
pmem_writeback
Array Write Component
Hit 2?
L2_write
lru_out
pmem_writeback
0
1
2
3
lru_array_out
Cache_Data[255:0]
dbit (to control)
Hit Determination (Part 1)
Array Write Component
data_in
data_out
1
lru_array_out
mem_address[7:5]
LRU Bit Array
read_state
hit_two
hit_one
lru_array_out
LRU Bit Flip Component
postwrite_data[255:0]
L2_rdata
Array 1 Dirty?
Array 2 Dirty?
Tag 1 + Index
Tag 2 + Index
Pmem Address Filter



Idle and Read are one state, where the inputs are evaluated immediately to determine which action to take. There is no clock dependency and thus no formal transition on clock cycles that happen between the two.

Inputs	Outputs
L2_read	pmem_read
L2_write	pmem_write
hit	L2_resp
dbit	pmem_writeback
pmem_resp	read_state
(presented in left to right order on diagram)	(presented in left to right order on diagram)

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.

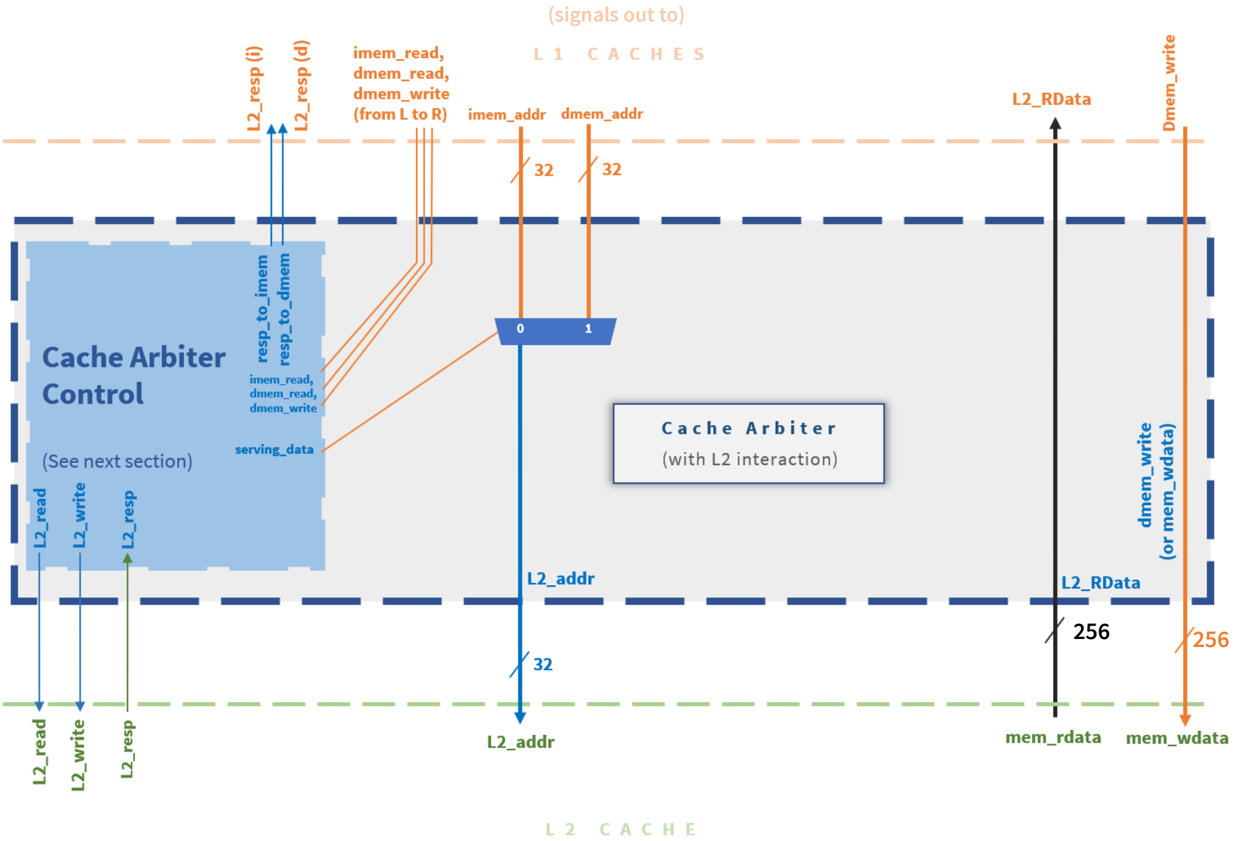
The first of the two physical memory states that are involved in this process is the **memread** state, which by its name, handles all interactions with physical memory to read data alone. It also, however, determines whether data must first be updated in the physical memory if it had been updated in 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.

Arbiter Design



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.

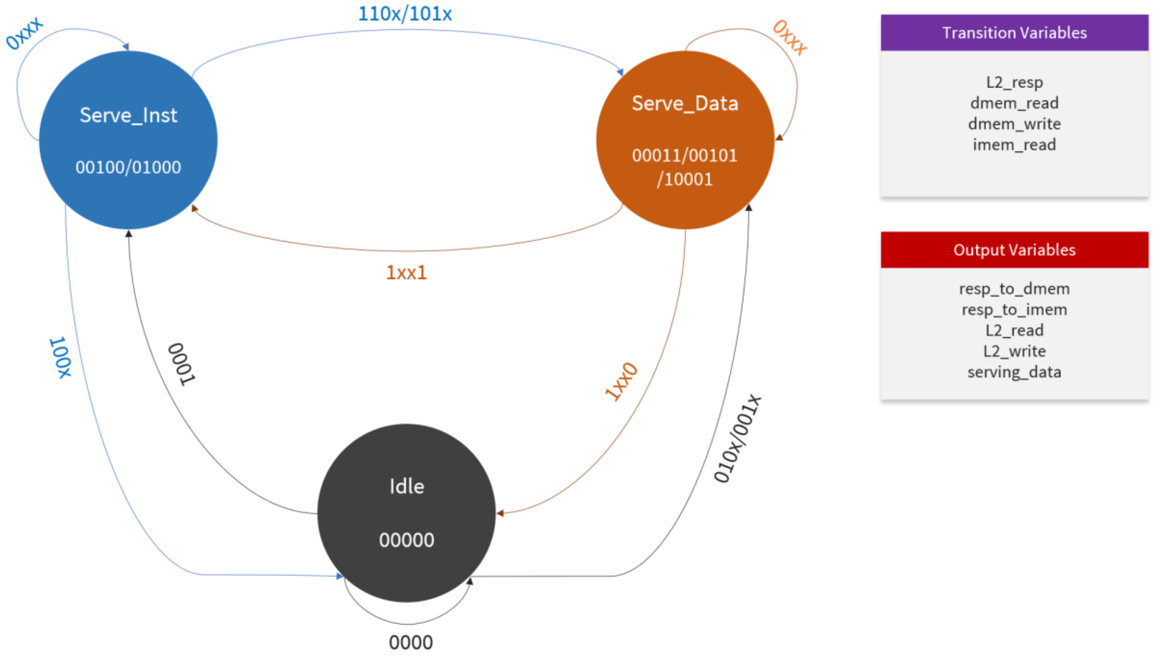
Moving from right to left, we first encounter two 256-bit signal pools that seem to pass through the arbiter, mapped from the cache thus to specific L1 caches. The read data goes to a signal called *L2_RData* from the L2 caches' known *Mem_Rdata*, essentially taking the output of the read data from the 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 seeing no 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.

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 work 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 and read 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 because 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 level of tests was 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 original checkpoint two 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.

Checkpoint Three: Hazard Detection and Forwarding Design Overview

For our third checkpoint, we 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 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.

Checkpoint Three: 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.

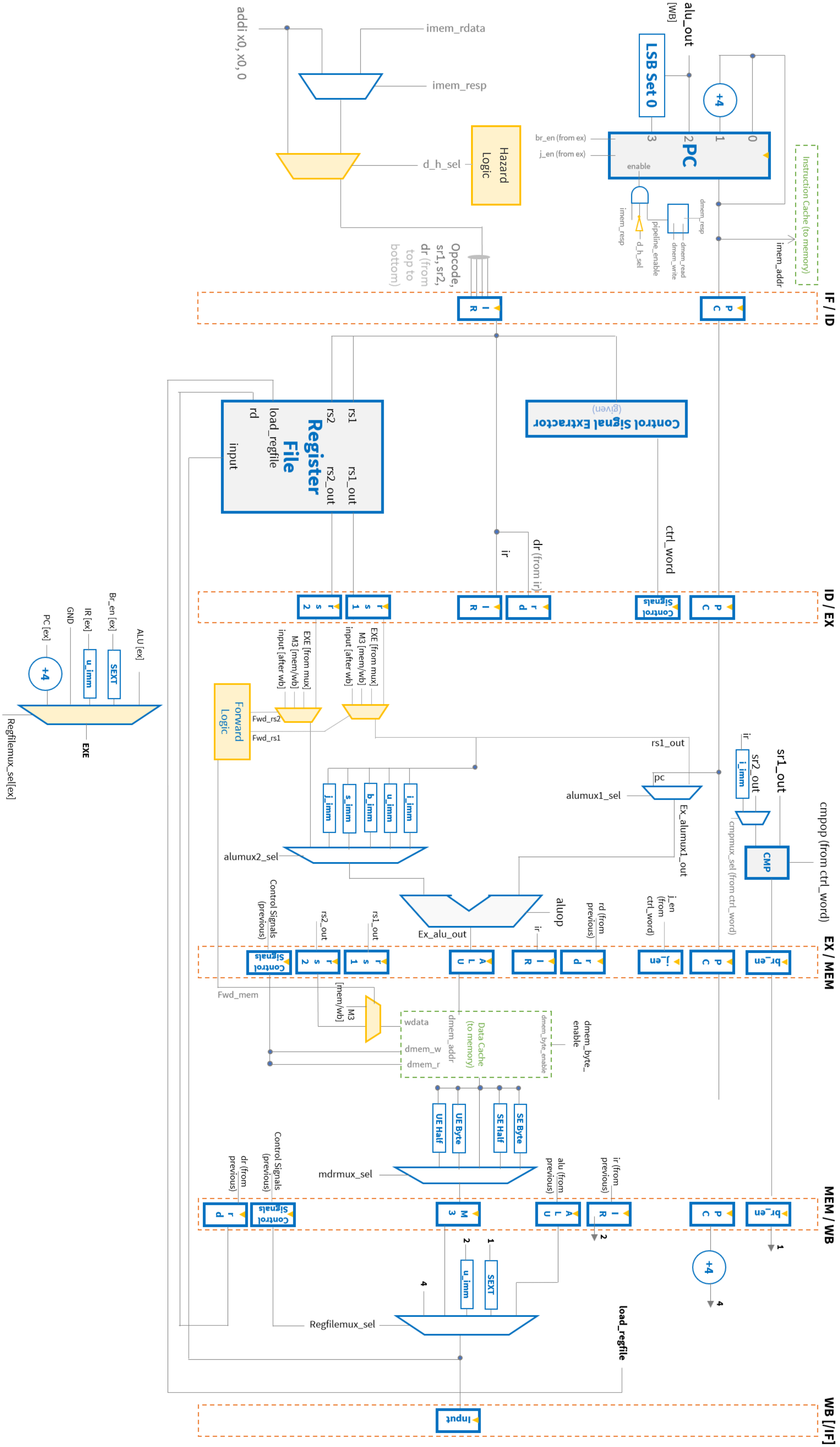
Now that this case is handled, there are now two more forwarding cases still left to consider. To finish review of the signals that are present in the forwarding multiplexers covered thus far in the execute stage, we note that there is an "input" signal that comes from wb. This is the same signal that 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, up to now, 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
enable
0
1
2
3
+4
LSB Set 0
alu_out
[WB]
br_en (from ex)
j_en (from ex)
dmem_resp
dmem_read
dmem_write
pipeline_enable
d_h_sel
imem_resp
Hazard
Logic
d_h_sel
imem_rdata
imem_resp
addi x0, x0, 0
Opcode,
sr1, sr2,
dr (from
top to
bottom)
P
C
I
R
Control Signal Extractor
(given)
Register
File
rs1
rs2
load_regfile
rd
input
rs1_out
rs2_out
ctrl_word
dr (from ir)
ir
Control
Signals
d
r
r
s
1
r
s
2
EXE [from mux]
M3 [mem/wb]
input [after wb]
Forward
Logic
Fwd_rs1
Fwd_rs2
rs1_out
pc
alumux1_sel
Ex_alumux1_out
i_imm
u_imm
b_imm
s_imm
j_imm
alumux2_sel
aluop
Ex_alu_out
sr1_out
sr2_out
i_imm
ir
CMP
cmpop (from ctrl_word)
cmpmux_sel (from ctrl_word)
j_en
(from
ctrl_word)
rd (from
previous)
ir
Control Signals
(previous)
rs2_out
rs1_out
br_en
j_en
A
L
U
Fwd_mem
M3
[mem/wb]
wdata
dmem_w
dmem_r
dmem_addr
Data Cache
(to memory)
dmem_byte_enable
dmem_byte_
enable
UE Half
UE Byte
SE Half
SE Byte
mdrmux_sel
ir (from
previous)
alu (from
previous)
Control Signals
(previous)
dr (from
previous)
M
3
1
2
4
u_imm
SEXT
Regfilemux_sel
load_regfile
Input
ALU [ex]
Br_en [ex]
IR [ex]
GND
PC [ex]
SEXT
u_imm
+4
EXE
Regfilemux_sel[ex]

Checkpoint Three: Revised Pipeline with Static Prediction

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

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

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

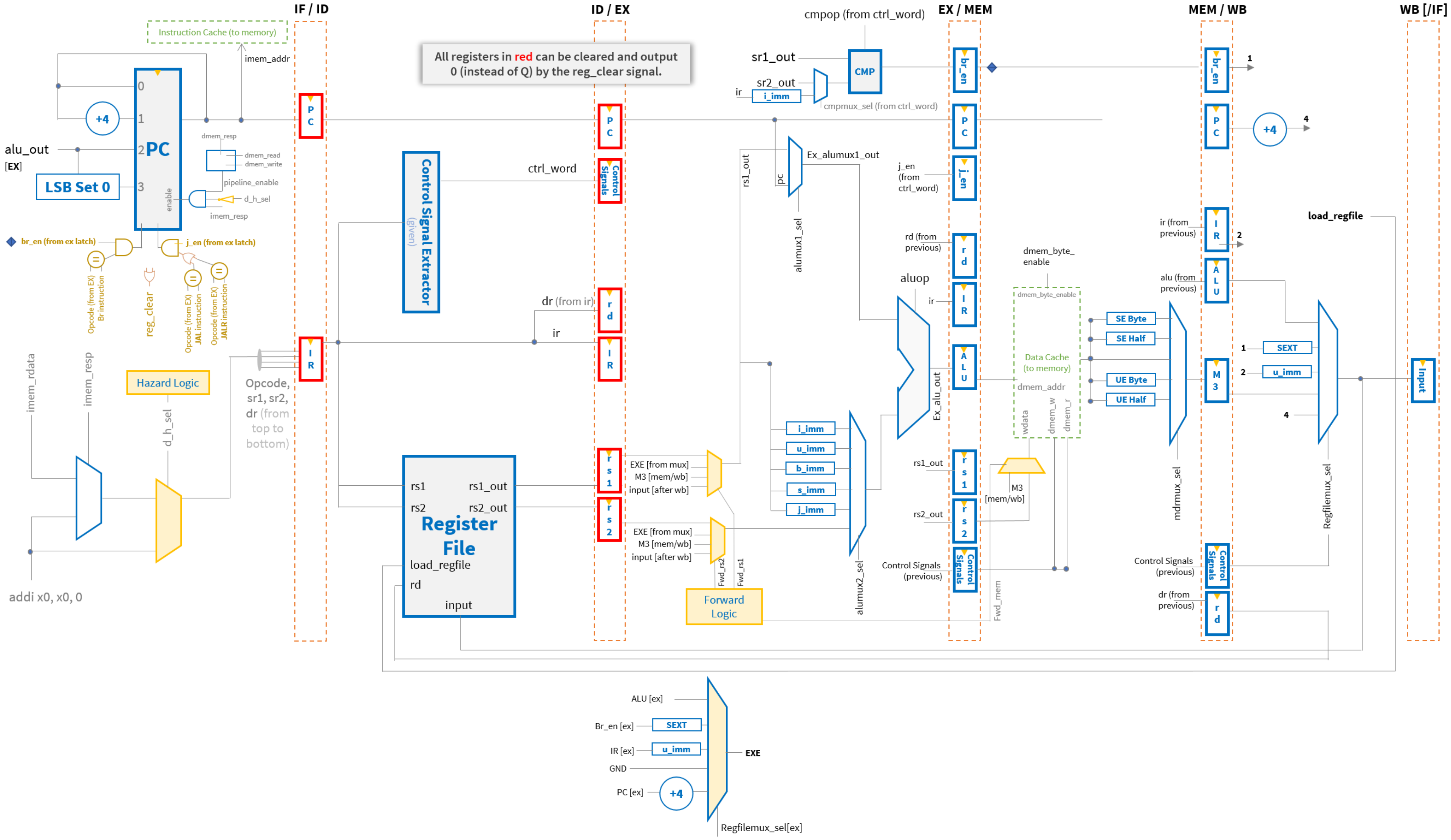
Therefore, we induce a mechanism such that the registers can be cleared given a certain signal, *and* the output of the registers is also clear such that data previously stored in them does not propagate through the rest of the pipeline. We would want to do this to all registers that are present in the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

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

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

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

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



All registers in red can be cleared and output 0 (instead of Q) by the reg_clear signal.

Checkpoint Three Progress Report

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

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

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

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

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

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

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

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

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

Checkpoint Four: Implemented Eviction Write Buffer

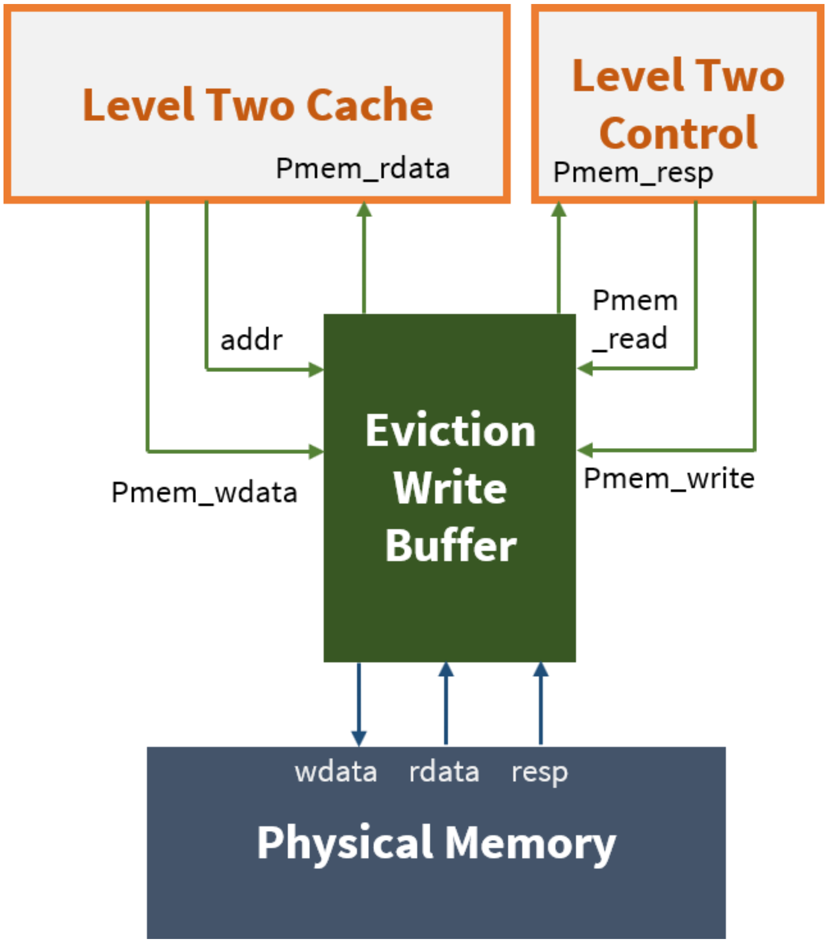
In this checkpoint, beyond the already implemented static branch prediction mechanism demonstrated from Checkpoint Three, we began differentiating our pipelined processor design with advanced design features. One of these categories was to optimize the way that writebacks are handled from the cache to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

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

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

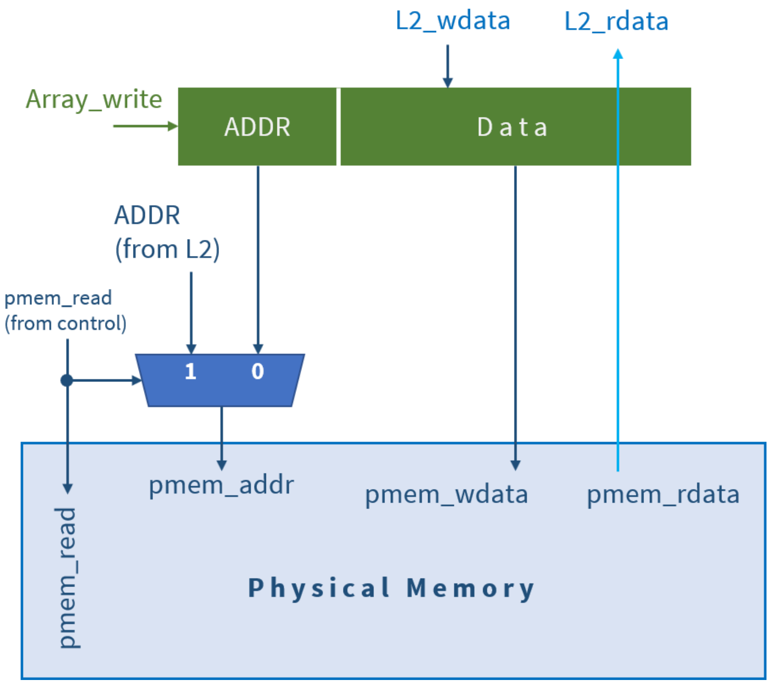
Should the design be expanded to multiple rows, note that a valid array would be required for this mechanism and the writeback stage would not be available in the same way – determination on what the most recent addition was would need to be included. Further, reads could happen whenever the pipeline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

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



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

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

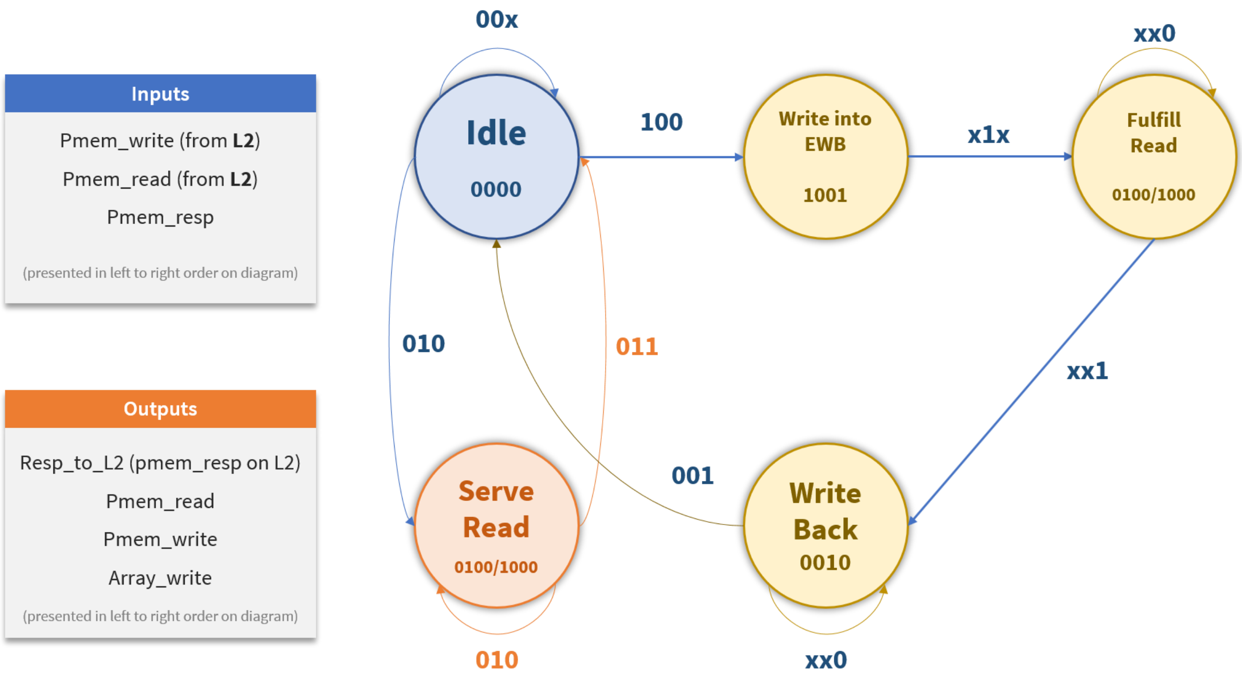


Whenever data is available from the level two cache, we want to place it in the input of our data buffer, should there be room available. However, this is not always the case, especially if a previous writeback is not serviced. Therefore, we rely on a control signal output *array_write* that tells if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

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

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

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



Inputs

Pmem_write (from **L2**)

Pmem_read (from **L2**)

Pmem_resp

(presented in left to right order on diagram)

Outputs

Resp_to_L2 (pmem_resp on L2)

Pmem_read

Pmem_write

Array_write

(presented in left to right order on diagram)

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

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

Checkpoint Four: Proposed Addition of Performance Counters

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

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

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

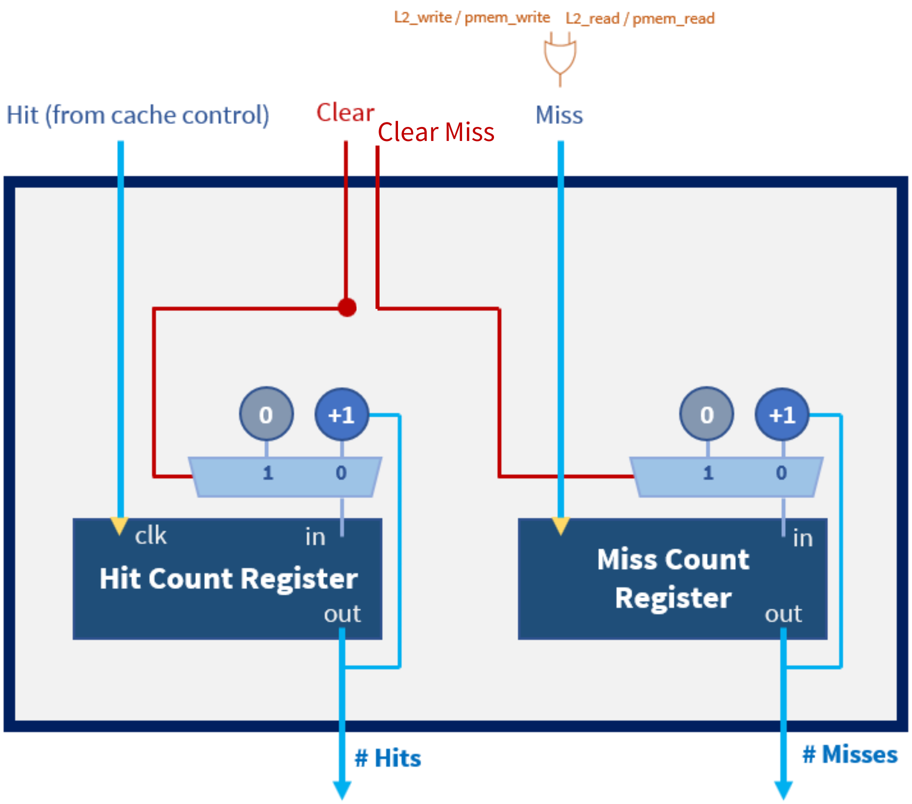
Especially in relation to the Level One cache data, then, we would like information about how well our static branch predictor is performing. Therefore, we will also use two counters for the number of branches that were predicted correctly, and how many branch statements were found altogether. **The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

Finally, there are four different ways that the pipeline can stall here, and therefore we want to categorize these into four different series of stages in which stalls can happen caused by a certain phenomenon. The first is a data cache stall counter, checking how many cycles pipeline_enable is 0. This tracks the stalling that happens across all the stages, since pipeline_enable is given to all registers for an enable signal. (In essence, the WB stage is not really tracked from this since the operation will write to a register file even when waiting for the data cache, but since the signal is given even to the new WB latch, this is added.) Therefore, whenever a new cycle comes up where pipeline_en is 0, we count it and accumulate. For this option, we will have to *AND* with the clock to get register behavior that accumulates on each cycle. This implementation can be abstracted based on the way that signals to accumulate are directly connected to the clock port of internal registers within these counters.

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

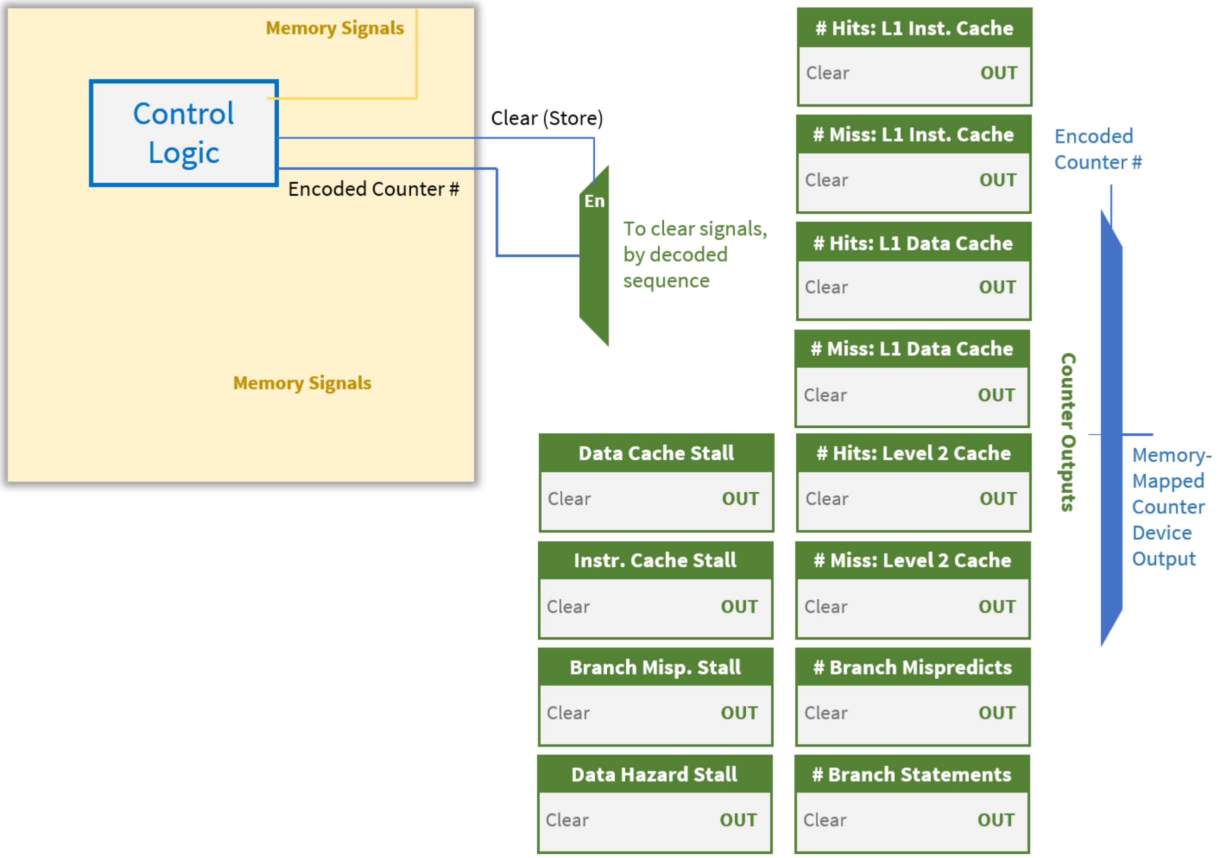
The third of the four counters that are used for stalls is a branch mispredict stall counter, which uses the same signal that our branch mispredict counter itself uses but accumulates by **2** each time a signal goes back high. Whenever this is raised, we know that IF and ID stages will have to be stalled,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

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



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

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



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

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

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

Ultimately, there were still some notable changes implemented that differ from this initial design, when it came to the performance counters themselves. These include, perhaps most notably, the removal of a counter altogether and the modification of how clocks were used in the clearable registers.

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

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

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

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

To prevent driving the register with combinational logic, then, we have redesigned our counter to take a separate enable signal that is driven by this combinational logic. The increment signal will then only be taken when this enable signal is high. This is separated from a reset signal that is generated in the control logic we have defined in our previous implementation. Here, our control signal logic (that generates signals like alumux1_sel, etc.) also controls these signals directly to switch between incrementing or clearing on the traditional processor clock signal, simply by evaluating what memory value was computed, as well as whether load or store was requested. (Recall that a store would clear the register, and a load would take the value in from the register associated with the counter here.)

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

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

Checkpoint Four Progress Report

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

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

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

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

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

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

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

These tests, admittedly, were shells of the original tests we and course staff had written for other portions of this MP, as well as previous MPs altogether. Therefore, Robert Jin led the writing of a completely new test, which upon running unveiled issues…about our branch prediction, which we already believed to be fixed. Indeed, Robert started writing the test such that one instruction caused an instruction cache miss, but also was immediately after the PC needed modification because of a branch statement preceding it. Dangerously, our previous pipeline design would have PC enable low because of the instruction cache miss. While this would safely give *no-ops* through the pipeline in the meantime, our access to the PC upon branch was temporarily blocked, and we could not commit the computation done from the execute stage. That is, the signal to PC is getting overruled by the instruction cache miss, where PC instead stays the same and is effectively disabled. This is a relatively rare occurrence, usually only encountered in longer programs and rare given that they must happen such that the instruction from the “not taken” misprediction also causes a miss here. A fix was issued by inverting the order of the PC reset check, where an instruction cache miss could still stall the pipeline, but PC could be overwritten *if and only if* a branch or unconditional branch (jump) deemed it necessary for the next instruction. If this could not happen, then the instruction after the branch would be invalid (the one if not taking the branch, not the one where the new PC is specified), leading to incorrect behavior.

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

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

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

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

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

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

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

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

Once this was completed, Robert Jin continued and attached the new control signals created in the modified control signal generator module to a module consisting of eleven performance counters (one for each metric that was reviewed in the previous section). Note that this is one less performance counter than before, as during this process, Robert discovered the duplicate utility of the number of branches mispredicted and the number of stalls induced by misprediction (which was just two times the former value). After this, a series of instructions was developed and run where the number of branches is known, specifically based on the MP3 test code, to test the individual branch and branch mispredict counters. The specific values, with explanation, are as follows:

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

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

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

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

Checkpoint Five: Overview of Planned Member Contribution

A roadmap for interaction in the fifth checkpoint's goals

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

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

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

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

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

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

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

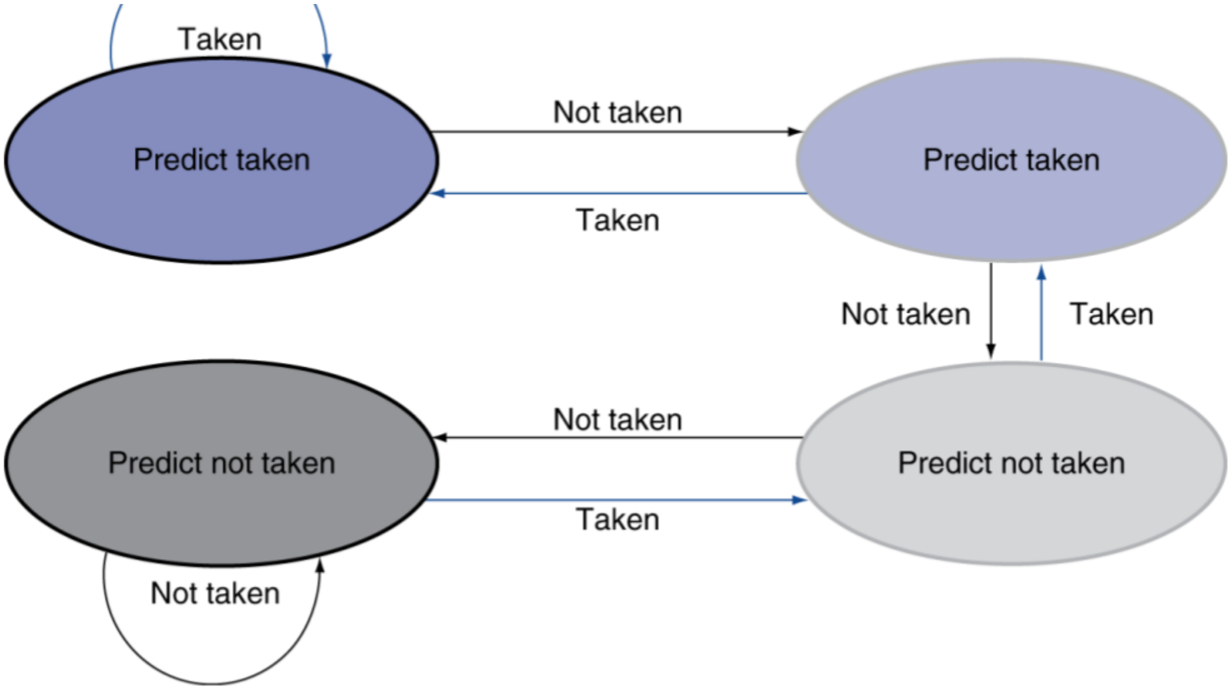


Figure 2: A class-provided copy of the two-bit predictor state machine that is contained at every BHT entry of this pipeline. (ECE 411 Lecture Slides, Lecture 09, Slide 19)

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

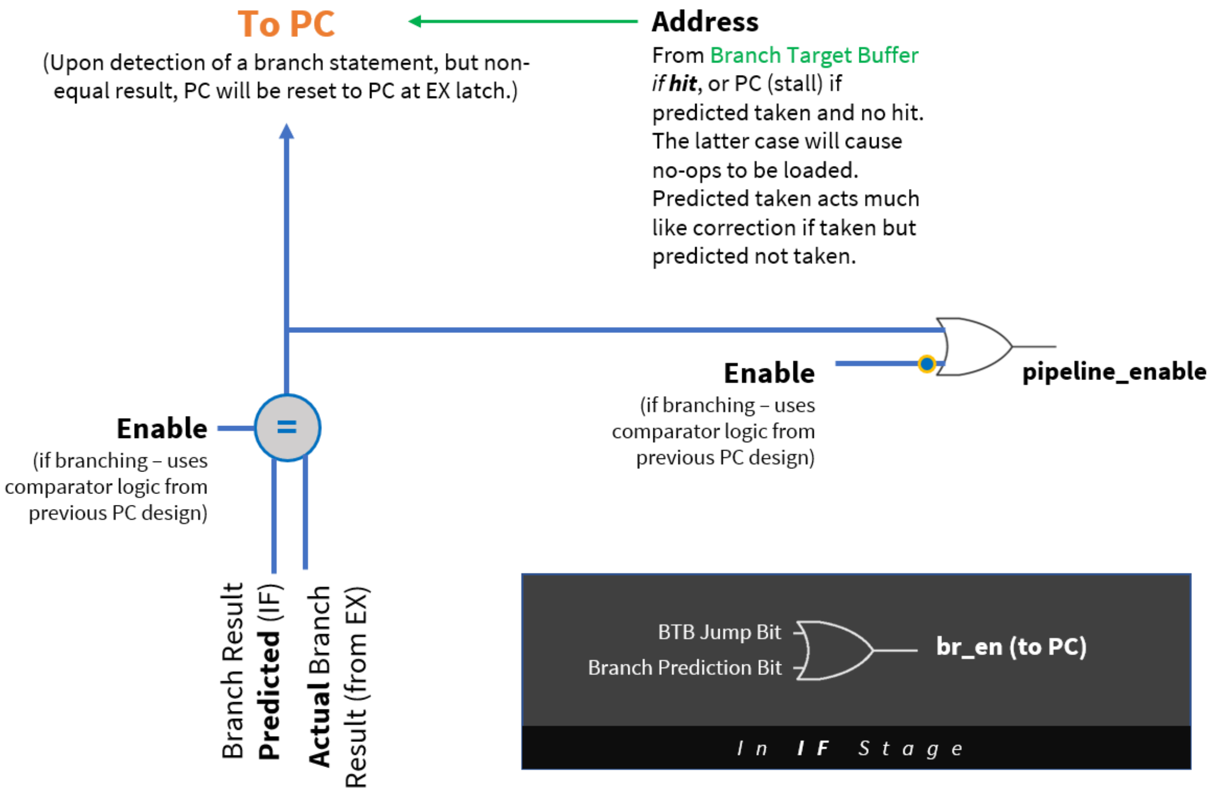
Note that this allows for some prediction of branches that are taken, which was fundamentally excluded in the previous task. Led by Robert Altman on design, however, with his and support of the rest of the team in implementation, taken prediction will be introduced into the pipeline by considering the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

Therefore, we are introducing a branch target buffer to complement the effectiveness of a global two-level branch history table. This branch target buffer acts similarly to a cache, with the entirety of the current PC loading an instruction used as the tag bits, and the data associated with each tag another value of the program counter held after the branch was evaluated. This data will be stored after a branch is evaluated whether the branch was taken, so that if the exact PC is encountered again in the IF stage, the branch target buffer (on a hit) can produce a valid PC. To do this, the hit signal will be output and given to the PC register in use, which will route the result from the BTB only when this signal is high. Recall that in our current design of the PC, depending on the value given by branch execute, jump from execute stage, etc., a different value may be loaded. Hence, this is an extension of our current design.

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

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

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


To PC
(Upon detection of a branch statement, but non-equal result, PC will be reset to PC at EX latch.)
Address
From Branch Target Buffer *if* ***hit***, or PC (stall) if predicted taken and no hit. The latter case will cause no-ops to be loaded. Predicted taken acts much like correction if taken but predicted not taken.
pipeline_enable
Enable
(if branching – uses comparator logic from previous PC design)
=
Enable
(if branching – uses comparator logic from previous PC design)
Branch Result **Predicted** (IF)
Actual Branch Result (from EX)
BTB Jump Bit
Branch Prediction Bit
br_en (to PC)
In IF Stage

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

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

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

Advanced: Early Branch Resolution in EX Stage

Design

From the early stages of constructing this pipeline, we have been concerned with the inability to know if a proper instruction could be evaluated should the PC not be updated in time with the result of a branch. In LC-3 and similar architecture designs, branches are dependent on condition codes, which must be set by previous instructions to determine feasibility of jump. Therefore, data would need to be forwarded to the execute stage before this early determination could happen.

However, in RISC-V architecture, these condition codes do not exist. Further, the branch enable signal is a combination of result from a comparator module in EX (for branch not equals, branch equals, branch less than, branch greater than, and their unsigned equivalents) and an evaluation of what instruction has been passed to the stage. In the scope of a pipeline, this instruction data can purely be passed with the addition of an IR latch from the IF/ID stages and in the ID/EX transition latch as well. From here, the only other operation that needs to be completed is a calculation of where the destination is to go should a branch be taken. Since this is done via the ALU out in the same execute stage, it thus becomes more easily possible to update the PC with a definitive branch resolution in the EX stage.

To do so, wherever the value of ALU_out was being used by the PC to update an instruction after branch or jump, the signals can instead be taken from the execute stage assuming stability in a single clock cycle. Since the PC does not update until the next rising edge of the clock, this ALU out value may be prepared, and the corresponding branch enable signal from the comparator likewise in the EX stage. From here, the PC can evaluate whenever this signal goes high, and map it to the ALU_out input that is forwarded from the execute stage. A diagram of the PC receiving these signals is shown below.

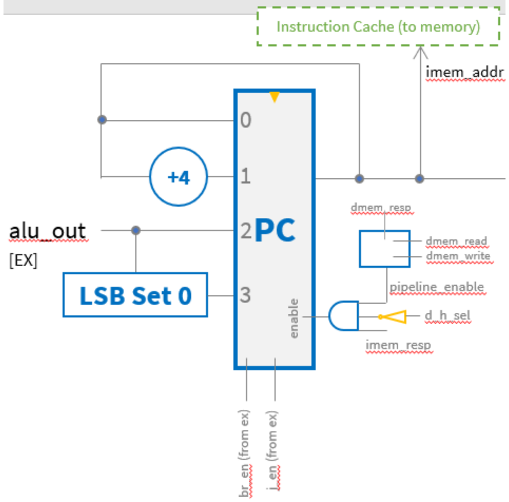


Figure 3: An instance of the PC as obtaining the result of branching from execute, as well as the branch enable and jump enable signals that indicate taking the branch here.

Testing

The early resolution is mostly indicated through any reduction of two cycles in a branch test, but with the addition of branch prediction features can be challenging to evaluate simply by performance. Therefore, a simple test was typically used as the starter to write more elaborate test codes checking on the integrity of branch enable and ALU out signals at the execute stage. (For example, the mp3-final test code that is provided for checkpoint five evaluation was used at times to ensure that the value was properly prepared by the EX stage, and that all mispredict, clearance, and stall signals were generated.) The base test code is as follows, adapted from MP0 and MP1 base tests:

```
riscv_mp3testbranches.s:
.align 4
.section .text
.globl _start
    # (Kept from the test documentation.)
    # Refer to the RISC-V ISA Spec for the functionality of
    # the instructions in this test program.
_start:
    lw  x1, one
    lw  x2, negone
testone:
    blt x2, x1, testtwo
    blt x1, x2, deadend
    beq x0, x0, deadend
testtwo:
    bltu x2, x1, deadend
    bltu x1, x2, testthree
    beq x0, x0, deadend
testthree:
    bge x2, x1, deadend
    bge x1, x2, testfour
    beq x0, x0, deadend
testfour:
    bgeu x1, x2, deadend
    bgeu x2, x1, halt
    beq x0, x0, deadend

halt:                 # Infinite loop to keep the processor
    beq x0, x0, halt  # from trying to execute the data below.


# In case the processor design is not working successfully and the
branch is not comparing the values correctly,
# typically the case of the XOR used in the comparator design being
completed incorrectly, then this is when
# deadend will show here.
deadend:
    lw x8, bad     # X8 <= 0xdeadbeef
# Simulate a halt by continually going to the same argument here.
deadloop:
    beq x8, x8, deadloop
```

One could evaluate that the branches were being properly executed in the EX stage by attaching the *IF_PC*, *ID_PC, and EX_PC* signals to a testbench, as well as the IR signal that contained the operation about to take place, and ALU out and CMP_out of the execute stage. One should observe that, when CMP_out gives 1, an operation of op.br is discovered, and the ALU out is reporting data, this ALU out is routed to one of the inputs of the PC register found in the IF stage of the pipeline. On the next cycle, *IF_PC* will show the value that was previously calculated in ALU_Out.

Note that the BTB test code provided later in this report was also written for purposes of testing early branch prediction.

Performance

When run with test codes that mostly consisted of all branches, performance was seen to approach **166.7%** faster execution if the branches were evaluated back-to-back. This was more evident in early stages of testing, where multiple no-ops from initial test code could be "skipped over" once a branch was known, and the pipeline did not require fill of additional instructions that prompted no control signals to be raised. This was all done, in the long run, with no performance decrease observable. However, in some of the initial stages of development (including checkpoint one), this did decrease the frequency of the entire design due to an extended path being created starting at the latches from ID to EX (that provide signals to the ALU and comparator) to the PC input taken in on a rising edge clock signal.

It is worth noting, however, that by introducing this early resolution early in the development of the pipeline, a structure dealing with stalls of memory was also deduced here. Especially in the construction of checkpoint three, a branch mispredict signal had to be used with static not taken updates of the PC, such that the value of the PC could not continue to increment and data in the IF and ID stages were flushed on detection of a branch. In having this functionality implemented from an early stage in development, however, we allowed to save multiple cycles of clearing registers just by moving the evaluation into execute stage and allowing other instructions to finish ahead of the branch here.

Advanced: Four-Way Set-Associative Level Two Cache with Pseudo-LRU Policy

Design Overview

Our current two-way level two cache was useful to hold concurrent data and instruction values in its memory, in case of instructions that are accessed over long periods of time being cleared out of the respective level one cache. However, the number of instances that data would not be available in level one of cache, but would be in our second level of cache, was strikingly low. This data was supported by the number of hits recorded in the level two cache versus number of misses, which were identical under almost every workload, and was identical in our own tests in giving **10** misses and **10** hits (one per miss). Many times, we waited for data to write back just because there were misses in L2, only for this data to be referenced soon again and cause too many misses. To decrease number of writebacks and to retain recent data, we built this cache for larger operations and shared libraries with commonly called subroutines.

One consideration for increasing performance, so to make the L2 cache more than just a transparent stop for memory to be stored on recent evictions or updates from memory and higher-level caches, was to double the number of sets that were supported in memory. However, this did not improve any tracking of temporal usage, or when instructions or data were last truly used by the program. While it may allow for this data to fill up more locations in memory, it relied also on a specific offset of one PC bit changing to determine a collision with data from our original eight sets – that is, data that shared 20 upper bits of tag for example with data in the L2 cache would collide and want entry at the same cache line. Therefore, this was only helpful for a limited number of 256-bit items that could be pulled from memory. In fact, the very scheme relied on the concept of sequential misses, that items writing into the cache would occupy different sets instead of different ways since their locations in memory only differed by 32 to 128 bits. (Mem_Set, used as the index, would look from the memory address bits of seven to five, and $2^5 = 32$ while $2^7 = 128$.)

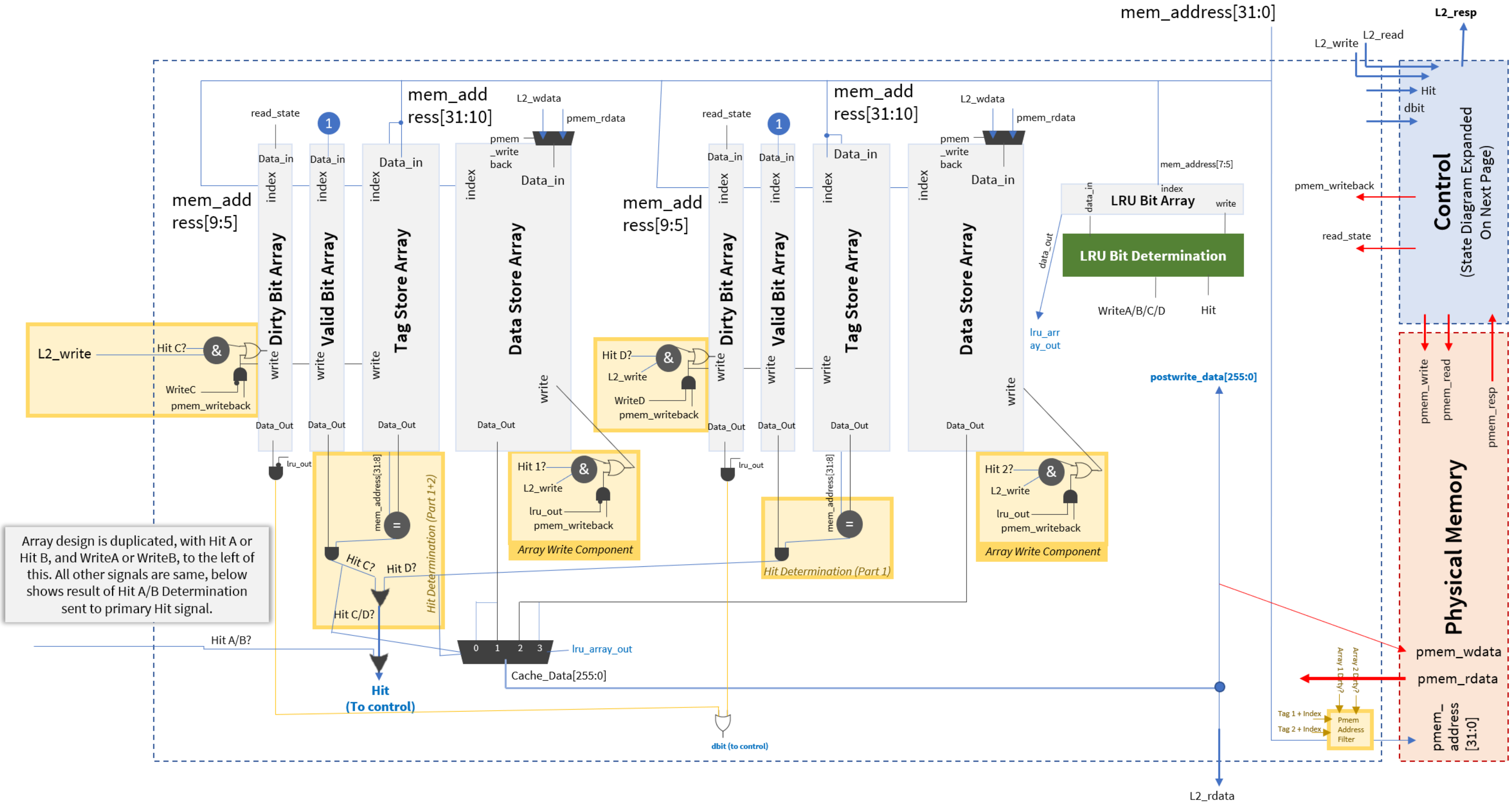
Therefore, an alternative to scaling up was to scale out the design, allowing for multiple instructions that would have otherwise collided and forced eviction in the cache to stay in a different way in the same line. By allowing for this, we hypothesized that we could reduce the number of writebacks by forcing data to occupy a space instead of potentially forcing a collision, should one be available. This is one of the benefits of a fully-associative cache, that if a cache block is not empty but others are free within the same line, memory can be written to the blocks that are free instead of causing some sort of eviction (which could cause a writeback, inducing even further performance issues).

As a result, we were committed to extending our eight set, set-associative cache to four ways, thereby doubling the associativity and up to doubling performance improvement should all entries in the cache require writeback. The number of arrays could not simply be doubled, however – the write logic had now changed based on the state and responses from memory, and the least recently used of the options had to be computed in a different way. However, computing LRU in the way we had previously done it would increase complexity in an unfavorable way – a queue of sorts would have to be built, and the latency for updating this policy (and generate write signals to each array) would grow higher with algorithm complexity.

To solve this issue, we conceded that most evictions are typically more reminiscent of the least recently accessed duo of ways that have been used, from which the least recently used of these can be established. While not offering a perfect mechanism for how arrays should be isolated and removed, this **pseudo-LRU** mechanism was much faster to compute, bearing a tree structure that only required three bits of storage at any single time representing least recent way used.

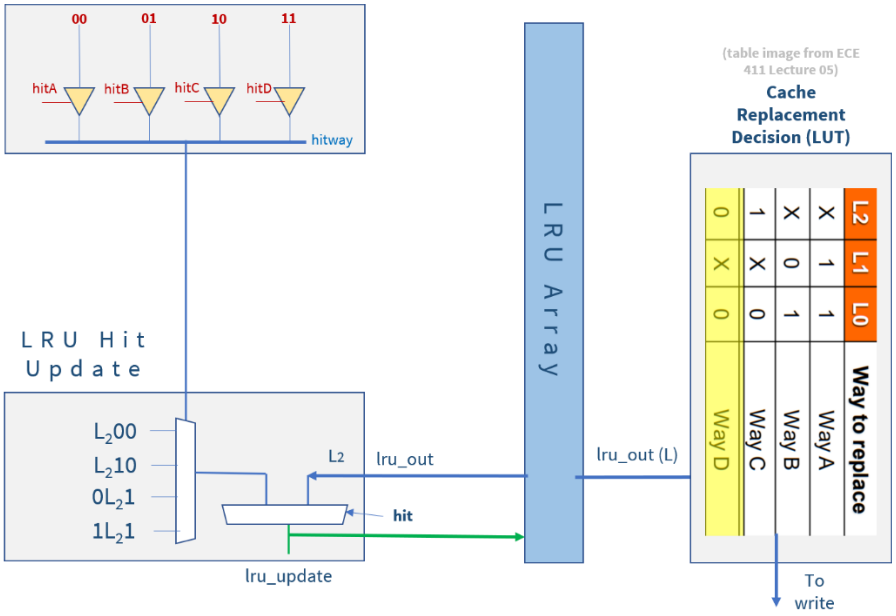
The structure itself may have been efficient, yet the overhead had still not been considered. Though tree traversal was known to be O(*nlogn*), since our size complexity of storage through a tree was the same here (O(nlogn)), we would at least need this amount of upper-bounded time to traverse just *what* the least recently used array was (as well as the least recently used duo of ways). This computation, however, could be simplified by precomputing cache replacement policy and state update tables across the eight different states that were possible for these bits. Further, only two bits would ever be updated at a single time with an LRU instruction, as one duo is modified and an internal way in that duo of ways is modified, but the other status bit in the non-modified duo is not. Therefore, we could compute and store four-row look-up tables that acted as buffers themselves – given a specific input of what our current least recently used way and duo was (represented in the three bits discussed earlier), we could update the bits by running through the state update LUT.

With the cache replacement state table as well, we could predetermine which way should be removed and give it a two-bit identifier. From here, the value can be decoded into a specific way signal that should be overwritten, where write enable could be properly set to take in this previously non-existent data. Note that this is made feasible due to a luxury of the L2 cache – we never have to worry about direct stores from the program execution with this data, but instead receive full 256-bit data "words" each time that never need to be directly written or modified in memory (unless they are replaced in full). When data is replaced in full, we consider this through an update at a specific memory value and can simply use the hit signal generated from parallel tag comparisons across a set's ways to know which way to write to. Therefore, our write signal can be written as a simple combinational AND of two signals: write[A,B,C,D for which array was decided for eviction] AND (L2_write AND tag[A,B,C,D]). The logic was thus replaced in this simplified L2 cache, leading to the implementation seen on the next page.



Array design is duplicated, with Hit A or Hit B, and WriteA or WriteB, to the left of this. All other signals are same, below shows result of Hit A/B Determination sent to primary Hit signal.

A zoomed-in feature look at the LRU Bit Determination is broken down between multiple LUT and buffer components. The LRU thus updates according to the values in lookup tables, as seen in the graphic below. Note that this comes in whenever a hit signal is not produced correctly, and the cache is currently active (an L2_read or L2_write is being serviced).



(table image from ECE 411 Lecture 05)

Cache Replacement Decision (LUT)

L2	L1	L0	Way to replace
X	1	1	Way A
X	0	1	Way B
1	X	0	Way C
0	X	0	Way D

A series of tri-state buffers monitors the hit signals that are taken from the datapath of the cache. Since only one of these signals can be high at a time and kept high, tri-state buffers are used that give Hi-Z on the hitway when not active, thus not updating the LRU array until an actual hit and use case of a way has been recorded. Once one of these is true, the hitway bus is loaded with the corresponding bit sequence for the way (e.g. D = 11). This is then used by the LRU Hit Update function, which takes the value of LRU out and, on hit, follows a look-up table encoded as a multiplexer to update the LRU bit state. This signal is given as *lru_update*, which is fed to the write signal of the LRU Array.

The LRU out, however, must also work in conjunction with the write enable signals that were seen in the previous page. That is, whenever we determine which way can be replaced by data, we use the cache replacement decision table (with a proper enable signal should indeed the data need to be replaced, which in this case is just the use of a hit). To demonstrate the equivalent lookup of this information, but this time in tabular form, the table shows that the lru out bits (denoted L) are matched and a way to replace is determined. This value is determined according to the same values associated with ways that hitway uses, such that way A = 00 and D = 11.

Testing

The data that is used and evicted happens according to a policy of which duos had the least recently used data, but these do not necessarily update on every hit that data may have if the data is present. Therefore, our best mechanism to follow is to see, according to the values of the LRU present in the pipeline's memory hierarchy at a current state, if the correct transitions are taken to mark which duo of ways (A and B, or C and D) was truly used last, and which way out of the duo of ways was used last from these.

Prior to completing any of this, however, we first wanted to ensure that any data was filling these ways at all, to ensure that no remnants of our design for the level two caches was left over and only occupying a specific set of ways. To do this, thus, we ran our baseline test code (defined in the following subsection, *Performance Analysis*, as mp3-final.s) through full execution to see that all ways were indeed being occupied. It turns out that one signal was not being written the first time that these tests were run, and thus only two rows from our look-up tables were ever found, thus emphasizing only two of the four ways that we defined. After repairing this, however, this specific test case had completed its job and could not demonstrate any particular update strategy correctness.

To enforce this, then, another test was created with five 256-bit items of data, separated by 15 256-bit "words" in memory to all compete for the same cache set in the L2 cache when loaded. One way would come in first, then a second would, with a third soon to follow. Once this pattern had been established, a write would happen to the second way and a writeback (load of the first way again) to the level two cache forced to recognize the change, such that way B (which has the second "way" of data loaded in) is now the most recently used and way C from the second duo is the last. Throughout this process of loading and storing, however, we can observe that the least recently used section goes from the first duo of ways (A/B) to the second once A forces writeback of B, and B is again accessed with its data.

At this point, D is then loaded with another "way" of data in the cache, with a fifth set of data to follow. This fifth set will force an eviction on the first way (A) since it has been used least recently, and the bit for the root node of the LRU "tree" has cache replacement policy LUT point to the left side of the tree (A/B duo, which then points to A as least recently used). From this point, various accesses are made given the data that is present to test the cache policy, such that various elements are written both to and from the data cache and data is updated appropriately in level two.

Similar tests to these were then also run, but with the byte and half word storage to ensure that any writes in the data cache would not modify the size of data received in the L2 cache. Indeed, all items still reported to work, with the state updating based on the LRU table previously defined as expected here.

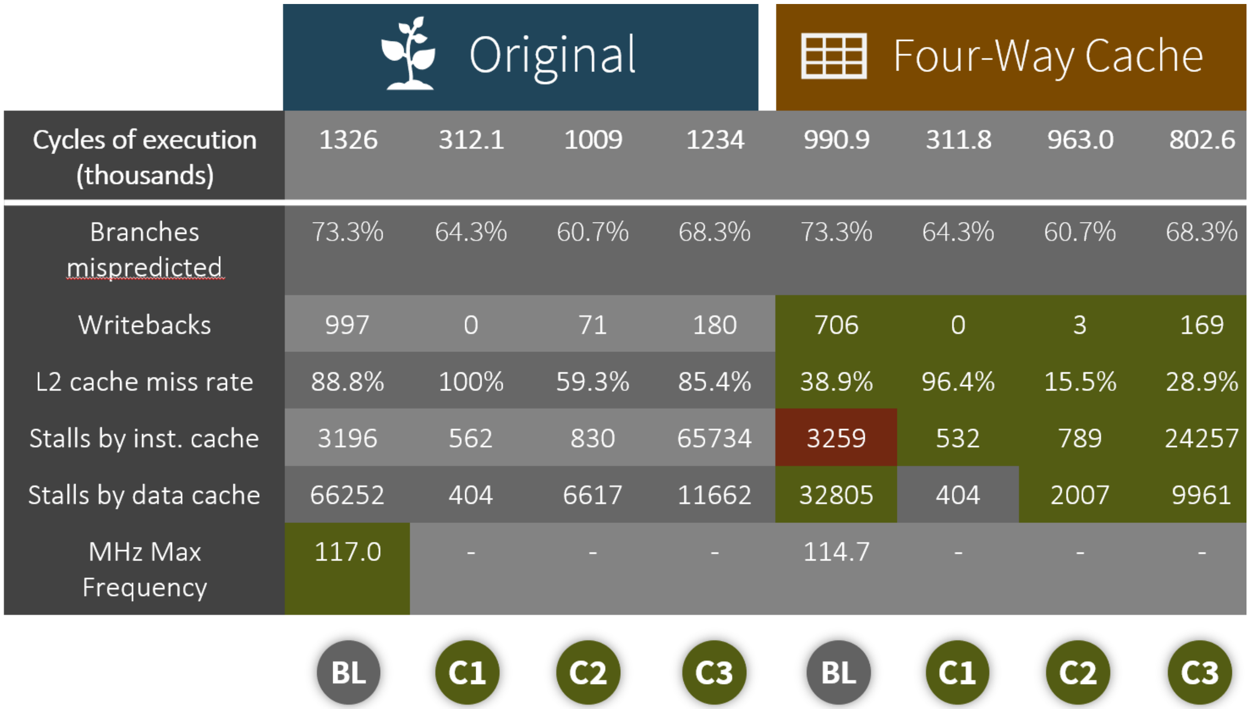
How to Read Performance Analysis

Prior to starting analysis for this cache, it is worth pointing out some conventions that will be used to analyze tabular data on the next features. Any element that is marked in green shows a distinctly different, and better, performance in a certain category than compared to the original design without the advanced feature. The comparison feature will not be handled on cycles of execution but will instead be discussed in the text following the data presented. All red data will show where the new feature performed worse – this will only be the case, however, with the four-way cache on the baseline test code, and the performance for the category is notably improved (beyond the original) when all new advanced design features are integrated into our final design.

Four codes are run per each feature that is presented, which include a baseline test code (mp3-final.s) that has been provided by the course staff, as well as three competition codes. The first competition code is a result of linked list sorting, and thus does not use as many iterative cycles but instead causes more instruction misses – a strong test to run to see if incorrect instructions are being taken when branch prediction mechanisms are integrated, for example. The second competition code uses a high amount of iteration, again meant to test the ability for the system branch predictors to capture state, but also to cache and properly manage data and incoming instructions across multiple new objects that are allocated onto a stack. Finally, the third competition code runs a calculation of a Minimum Spanning Tree for a given bipartite graph, but uses return statements on a function that is consistently called from the same address. This return statement involves the function *jalr x0, x1, 0*. The functions that were completed by the competition codes are provided in description by our mentor TA, Kenneth Umenthum.

All metrics were otherwise found using performance counters, following the design that was proposed in used through checkpoint four of this project. The first results can be seen on the next page here.

Performance Analysis



	Original				Four-Way Cache			
Cycles of execution (thousands)	1326	312.1	1009	1234	990.9	311.8	963.0	802.6
Branches mispredicted	73.3%	64.3%	60.7%	68.3%	73.3%	64.3%	60.7%	68.3%
Writebacks	997	0	71	180	706	0	3	169
L2 cache miss rate	88.8%	100%	59.3%	85.4%	38.9%	96.4%	15.5%	28.9%
Stalls by inst. cache	3196	562	830	65734	3259	532	789	24257
Stalls by data cache	66252	404	6617	11662	32805	404	2007	9961
MHz Max Frequency	117.0	-	-	-	114.7	-	-	-
	BL	C1	C2	C3	BL	C1	C2	C3

When designing to include this four-way cache, we were motivated by the idea that too many writebacks were currently happening throughout each program, and this data was constantly being swapped in and out of memory simply because there were not enough ways to hold it. Because of this, we had to constantly keep reading and writing back to memory, which caused a miss rate every time this data was written back when an iterative structure or search-based algorithm needed to access it again. Therefore, our four-way cache was written to primarily reduce the number of writebacks, and to drop the miss rate in the level two cache – by far, except for the baseline for stalls in the instruction cache, this goal was achieved.

In the baseline, there is a significantly lower miss rate in the level two cache, where the number of stalls caused by the data cache is nearly cut in half compared to the original value. Because of taking out these stalls induced by the data cache, however, there are instead more ways and more cache blocks being occupied with the data cache that is needed by the function. As a result, any instruction that was found outside of a currently executing subroutine while data was being pulled was being missed more often, even though a tremendous improvement had occurred in the miss rate overall. Therefore, the number of stalls induced by the instruction cache increased, though seemingly by only the equivalent of how many cycles were needed to complete a physical memory read for the instructions, and populate them into L2, prior to returning the data to the instruction cache for use.

Indeed, where instructions are more often retrieved instead, and there is high movement between different data structures but especially different routines altogether, a high cache miss rate was expected. This was especially true in competition code one, where all items in the L2 cache caused a miss here, due to high locality that was found in the code being run (a linked list sorter with pointer chasing). Because of this type of sorting, paired with a constant “find and forget” attitude from the data retrieval, there were no less stalls caused by the data cache than there were in the past. However, common subroutines that were called did hit once more often in the L2 cache than previously held, such that the instruction cache could return the correct instruction without calling on physical memory. Therefore, the rate of cache misses, while still not positive, has decreased here.

The true impact of this cache, however, can truly be observed in the competition codes two and three. Here, there is a high attempt at running the same iterative code, so the number of instruction stalls decreases significantly already due to the number of instructions that can now be stored in the extended cache. The number of data stalls, for similar reasons, does the same – even though there is another general case of finding large numbers of nodes in a bipartite graph for the third competition code, the nodes are still nearby enough in execution such that traversing through the node set causes fewer stalls in data cache. Altogether, the miss and hit rate improved tremendously throughout these two codes, at least by factors of 50% per each L2 cache miss rate simply by holding double the data that was being computed on. The second competition code in particular, due to its high iteration level on less data altogether, performed the best out of all codes in this category with a 15% miss rate.

The data that is used and evicted happens according to a policy of which duos had the least recently used data. However, these do not necessarily update on every hit that data may have if the data is present. Therefore, noticing from our tests that there are often not ways to truly signify when a line in the cache is being used if the line was instead hit in one of the level one caches, we extended the signals for hits in the data and instruction caches to the Level Two cache as well. Should a hit happen in one of these states, and the data still exist in the level two cache as well, then a hit is registered in the L2 cache as well. This was discussed as a *hit-through* scenario, where even though no request to data in L2 was necessarily made right away, we could still say that this specific line in the L2 cache has been updated or touched by some cache in level one, and thus has been recently used. Therefore, it prioritizes true activity that the data has in the program, rather than the number of times data has been seen being written back or read since there was a miss in the level one caches when responding to the program. Therefore, this hit-through policy is a better metric of keeping actively used data and has led to improved performance results that contribute to those seen above. Notably, in competition codes two and three where a high number of iterations are used on the same data, we can retain this data instead of forcing writeback right away (especially in the case of competition code two, where the number of writebacks decreased by 23.6 times).

Therefore, especially coupled with the performance boost offered with this policy, we conclude that the Pseudo-LRU four-way set associative cache extension by far exceeds the performance of the original, two-way, true LRU equivalent L2 cache. Specifically, there was a **23.1% performance increase** on final benchmark **at 98% original frequency**, thus retaining a high speed without high drop from a combinational logic critical path. The performance value is computed as the sum of the new result over the old result for cycles of execution, divided by the number of test codes run (four, in this case here).

Advanced: Branch Target Buffer with Jump Support

Design Overview

As of checkpoint four of this pipeline design, we had assumed that all branches encountered were not to be taken to reduce overhead logic that was found in the instruction decode state. We did not want to evaluate any items from the instruction, as this would force a critical path that connects to the instruction cache and waits for the data to be returned in the instruction.

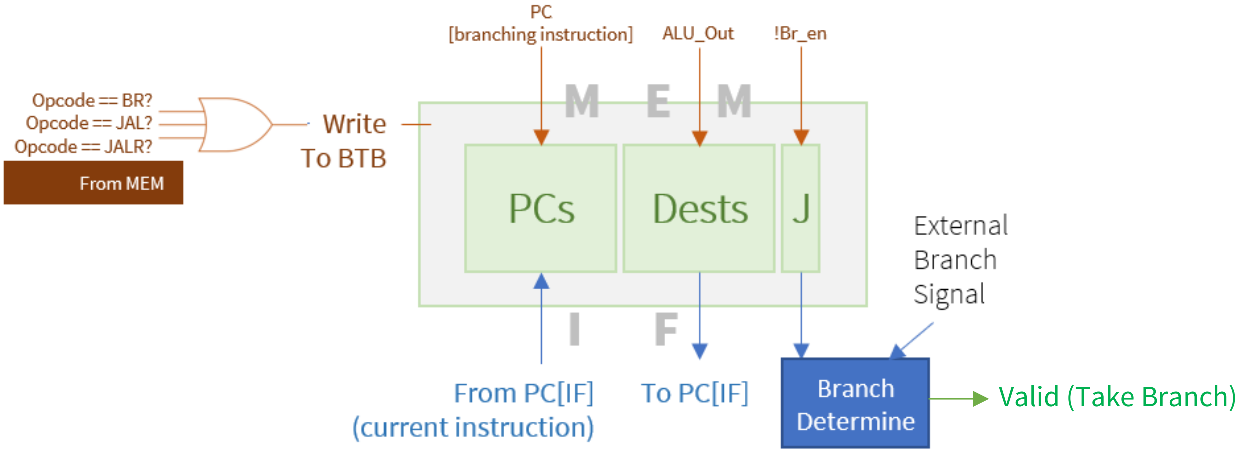
Some results were acceptable at first, where cases where the number of branches taken was low causing relatively fast performance throughout the program. Too, in our current structure, any prediction to take some sort of branch would not yield any benefit, as the data for the branch destination would not be available until the execute stage at earliest. The same could be said for any jumps as well, as these also relied on register values or immediate value addition, which could only be computed via the execute stage's ALU out (and rely on data from the instruction itself).

However, if we have seen a branch or a jump in a specific place before, and know it to be taken, we should be able to note if the same instruction comes through again where it is, what the result was, and where the destination of the instruction specified to go. By doing so, we need no interpretation of what the instruction reads to be, but only its location in memory. When this location is approached again by the PC, there is thus a history kept of what this instruction translates to, and where it goes because of executing it should a branch or jump truly be taken.

Influenced by this, we present an additional feature to the pipeline in a branch target buffer that offers support for recording jumps as well. By itself, a branch is always predicted taken and the PC is loaded with an output destination value should a match be found in the buffer for the current PC of the instruction to execute. A jump array is added for when branch prediction is defined by a separate module, where knowledge of the fact that a jump does indeed happen supersedes any decision to take or not take a branch or jump. That is, all jumps are always taken, if a bit that aligns with the entry of a PC and destination in the branch target buffer is also set to one.

Often, one skips data lines or routines, or makes unconditional branches at the end of loops, with **j**. By adding jump support, we can predict taken on instructions known to be jumps just by their PC. If there is a branch present instead, we still have data ready to immediately update PC, thus. To implement this, we use a cache-like structure with sixteen direct mapped sets, which are indexed using the fifth to second (zero-indexed) bits of the program counter aligned with the incoming instruction. Each of these sets has an entry for a tag (which contains the remainder of the PC bits, with exception for 1 and 0 since each instruction is 4 bytes in length), corresponding destination, and whether the branch recorded is a jump (and should automatically be taken). On a tag hit, which is done by evaluating the tag in the set of index PC[5:2], PC destination data is released to the PC and next cycle will bring in an instruction from this destination. This is to happen, however, only if the entry is a jump, or a separate branch predictor also is assuming that a branch is taken at a current PC. When no branch predictor is available (as in the next tests), the branch is taken unconditionally until a mispredict proves otherwise in the execute stage.

To allow for this functionality, the lookup itself is not particularly complicated and can be done using a direct mapped cache structure. However, the connection to the pipeline itself must be inserted at strategic places, such that the PC knows now when to update should a branch be predicted earlier (from the IF stage, and not from EX's deduction of a branch), and how to react to a branch mispredict when the branch was predicted taken (and not just not taken, as before).



First, however, the construction and read/write mechanism from the BTB itself should first be evaluated. An external branch signal will be used in conjunction with the presence of a jump, where if the branch is truly taken due to a PC match in the BTB, a Valid signal will be raised for the rest of the pipeline. The same will occur whenever a jump signal is returned from the array that is being used on lookup. As for the PC used in lookup, this directly interfaces with the PC array, and destination gets written to a port on the PC that is used as new PC should the valid signal be raised by the BTB (and no misprediction has occurred that must first be addressed down the pipeline).

As for after our computation from the EX stage, to further reduce complexity of the single stage, we wait and allow for the data to be written into latches first. From here, we then evaluate whether the opcode was a branch, JAL, or JALR, and if a branch was used, evaluation of whether it was taken. If any of these turn out to be true, then we know to write the current instruction as one causing a branch of some sort, such that if we see it again, our BTB will recognize it. Therefore, the PC of the branching instruction (from the latch entitled MEM_pc) will be input with this write signal, as well as the value of the ALU output that holds the destination, and the inverse of the branch enable signal. The latter is done such that, if one is writing and no branch was taken, br_en would be low and the instruction would have to be a jump of type JAL or JALR.

The only remaining portion, then, becomes the modifications to the PC itself. First, it must always prioritize times when the entire pipeline is disabled, and no movement should occur. Then, a branch misprediction should be checked, such that a signal may be sent out to nearby registers that clearing needs to happen and the PC needs to update to a known good value from EX stage. (Note that this relies on early branch determination, thus, in said EX stage.)

How, then, is branch misprediction determined? As before, if there is a branch that should be taken but the signal to take said branch was low, then we prompt a mispredict. However, now that we can predict that a branch was taken, there are some additional circumstances that we must check. First, we must have a signal that passes through latches from the IF stage, through ID, and onto the EX stage signaling if a taken prediction was speculated by the BTB. If this indeed happens, we then check if a branch instruction was passed through. Should there be one, and the br_en signal was not raised, then the value of PC + 4 is taken in by the PC instead after EX_branch_mispredict is raised for this specific case. However, if a jump was instead predicted in the form of JALR, and the register value has since changed from the last store to the BTB (causing the destination predicted to do likewise), a mispredict must again occur as this data was not available to the PC when it first reset the value based on BTB destination contents. PC + 4, however, is *not* taken in this case, and the ALU out instead is as if a branch was taken but predicted not taken.

Testing

With the various types of additions that must be made to the pipeline, described in the above section, to support the ability for branches and correction on misprediction (without a static predictor), there should be tests to just check on this misprediction logic first. Therefore, before even testing the BTB itself, the signals should be generated on random instructions with preknown destinations, given by hardcoded signals that are temporary placed in the design. Should mispredictions come about whenever these incorrect signals do not match the ALU output and/or a branch is not taken, then the PC should revert to either ALU value (if the values simply did not match but a branch or jump happened anyway), or to PC + 4. Not taken cases should also be tested to ensure no previous incorrect revisions on taken when predicting not taken have also not been broken. Finally, additional tests can be run that force stalls in the pipeline. Namely, a sequence of data loads should be considered prior to a branch occurring, such that several stalls is incurred retrieving data from physical memory and preventing PC from incrementing to an incorrect instruction without knowing data later in the pipeline.

Only once this misprediction logic and the PC can be verified as working together and clearing data on proper mispredicts can the BTB then be tested. The following test code was one commonly employed to go through the various sequences of mispredictions, while still allowing for a certain number of iterations that would utilize the BTB.

```
# This test is meant to isolate behavior for the BTB, showing where
there should be a hit,
    # where misprediction cases should happen, and how jumps should be
treated (including JALR,
    # where the destination can change based on a register value).

    # As a result of heavily using branches, one can use the PC values
to track early branch
    # prediction in the EX stage as well. Thus, the test is written for
this purpose as well.

    # For fast evaluation metrics with only BTB connected, you can use
this:
    #   - 6 Branch Mispredicts caused by branches
    #   - 3 Branch Mispredicts caused by Jumps (1 on first discover of
JAL, another for first JALR, another for JALR incorrect dest.)
    #   - 9 Branch Mispredicts in Total
    #   - 6 occupied lines in the Branch Target Buffer
    #   - 2 occupied jumps for JAL and JALR occurrences
    #
    # __|__ like lines (as opposed to __||__ lines) do not count as
branch mispredicts, since the low value would be read by all items
reading from this signal.
    # This is due to the fact that all of these are registers and wait
for a stable signal still.

_start:
    lw  x4, four
    add x1, x0, x0
    # Test one consists of a loop that iterates for four (4) iterations.
The first branch should miss.
    # The next three branches should predict taken from the BTB alone,
with the last one incorrectly taken
    # since it would conclude iteration. However, the result can be
modified from our branch predictor's input as well.
    # Therefore, we simply should see that a "btb_valid" was listed here
with correct storage data:
    # - storage happens in set 13 (zero-indexed), as PC at br is x74,
and x74[5:2] = 13_10.
    # - dest_storage has 0x6c (0x74 - 0x08, since each instruction is
0x04)
    # - pc_storage has 0x1 (tag)
    # - jump_storage has 0x0 (no jumps added yet)
testone:
    add x1, x4, x4
    addi x4, x4, -1
    bne x0, x4, testone

    # Test two moves on and utilizes BTB knowledge with jumps, the
traditional JAL kind.
    # These, since these are valid, should always be taken no matter
what the branch predictor is suggesting to do.

    # Again, we're doing four iterations of this. On the fourth
iteration, we go past the jump instruction.
    # For testing the branch predictor, this causes an interesting case
where we can keep seeing a NT/T pattern
    # across the iterations. Since it lasts for three iterations, it
nearly fills up all 8 bits of the register
    # used to XOR with the PC values.

    # Note that the branch instruction that comes before will also cause
a write to happen inside of the BTB.
    lw  x4, four
    add x1, x0, x0
testtwo:
    la x30, testtwo_x7check
    add x1, x4, x4
    addi x4, x4, -1
    beq x0, x4, testtwo_x7check
    jal x7, testtwo
testtwo_x7check:
    bne x7, x30, deadend
    # Before moving on, even if you make it past this line, that branch
should NOT
    # show up or change the BTB contents. It should never be taken, so
never should be written.

    # Then there is JALR. We have already seen what happens with
traditional jumps, but what happens when the destination
    # itself changes because of a register? We predicted the wrong
destination, we would have to overwrite even though we are taken.
    # Test that here and track the stalls of the system.
    addi x2, x0, 3
    la x31, testthree
    la x29, testthree_x7trap
    la x28, testthree_x7check
    # The first jump will miss, the second will jump with correct PC,
the third will try to jump but with incorrect PC.

    # Again, note that a branch pops up here, so it will modify some of
the original taken prediction from BTB here.
testthree:
    add x1, x2, x2
    addi x2, x2, -1
    bne x0, x2, testthree_x7jalr
    la x31, testthree_x7check   # Change the register to force a
different jump
testthree_x7jalr:
    jalr x7, x31, 0
testthree_x7trap:
    lw x7, bad
testthree_x7check:
    bne x7, x29, deadend
    lw x7, good


halt:                 # Infinite loop to keep the processor
    beq x0, x0, halt  # from trying to execute the data below.
```

Not only should performance increase when running this sequence, but not taken should remain the default action with only the BTB enabled here should no hit be visible within our direct-mapped buffer. However, if any branch or any jump is recognized, after the first not taken round has been successfully been proven mispredicted, the next couple of iterations that are allowed should be done with PC updated in the IF stage (with no mispredict trailing).

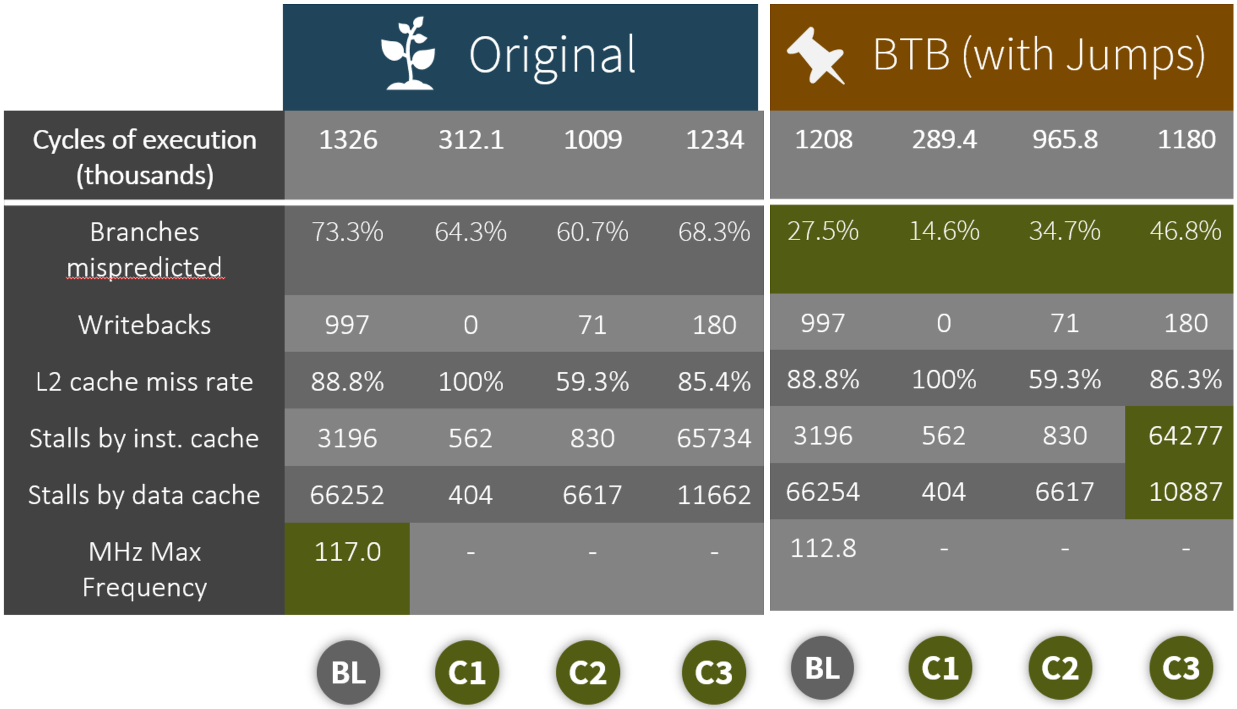
Stepping through, we first check if a series of branches will complete based on comparison a counter going from 4 to 0. The first occurrence should mispredict, as the default behavior is not taken. The second, however, should be ready given that the branch is updated in the MEM state and correction happens in the EX stage. Third iteration should also function correctly, but then a fourth iteration should mispredict as taken, when the loop should be exited. This further tests our ability to handle multiple types of taken/not taken predictions and raise correction with PC + 4 this time to the PC register after correcting in EX stage.

The second test not only looks at the jump that is about to execute for the same number of iterations, but also ensures that the register it writes with PC + 4 is indeed loaded with the correct value. Therefore, we expect the first occurrence to mispredict both for the jump and an intermediate iteration check handled by a branch not equals. After four iterations, however, the jump can no longer be accessed. It is here that we evaluate the update protocol of jump, to ensure that our early prediction of the jump did not prevent us from writing the correct PC value.

Finally, we test the same iteration structure with JALR. For the first two iterations, the register that JALR uses for computation holds a value to continue the iterative process. However, when iteration ceases, the value in the register changes and the JALR is forced to move to an instruction (depending on the value written) past this iterative loop. Therefore, we should see not only a misprediction for branches and the initial jump recognition, but indeed for when jump is recognized a last time, but the destination no longer matches the computation of register + offset from EX stage. If this is not handled via a mispredict, a trap is loaded to show in the registers. This concludes testing of all cases that could be used with branches, jump and link, and jump and link register.

It is worth noting that this entire prediction mechanism is predicated on the idea of not having self-modifying code. Should this happen, revisions would be necessary to check the integrity of the address as a branch, jump, or jump and link register altogether.

Performance Analysis



	Original				BTB (with Jumps)			
Cycles of execution (thousands)	1326	312.1	1009	1234	1208	289.4	965.8	1180
Branches mispredicted	73.3%	64.3%	60.7%	68.3%	27.5%	14.6%	34.7%	46.8%
Writebacks	997	0	71	180	997	0	71	180
L2 cache miss rate	88.8%	100%	59.3%	85.4%	88.8%	100%	59.3%	86.3%
Stalls by inst. cache	3196	562	830	65734	3196	562	830	64277
Stalls by data cache	66252	404	6617	11662	66254	404	6617	10887
MHz Max Frequency	117.0	-	-	-	112.8	-	-	-
	BL	C1	C2	C3	BL	C1	C2	C3

Evaluating performance with branch prediction models is hypothetically easier, in that there should simply be less branches mispredicted and the result should cause a decrease in the number of cycles as a result. Indeed, this type of evaluation was seen, with the cycles of execution on average decreasing from a range of 30 to 120 thousand cycles, all with a maximum frequency drop of only 4.2 MHz from the original design as well.

The statistic for branches mispredicted, however, was by far much better across all test codes. Starting with the lowest improvements first, competition code three was most effective at working with larger sets of data, thus typically leading to a lower cache miss rate if we were to extend our cache. Indeed, in the case invoked here with our BTB that does not have the four-way cache, we in fact see that the cache miss rate goes up as some of this data is not available, but more readily and consistently accessed than in previous cases. That is, there were less cycles that caused instruction hits in the level two cache, but in turn also less cycles that were able to hit for data in the level two cache as well. Similarly, competition code two that induced a high level of iterations across different type of objects dropped from 60.7% to 34.7% branch misprediction. This can be expected, as the number of iterations was smaller in the subroutines invoked in competition code two. Thus, with more instructions having different branches (and *especially* jumps in the case of CC2), we see that the number of branches mispredicted does not drop significantly. The third condition code can also be reasoned to result in this, as it is the only one that involves the *ret* instruction. Since the point at which a subroutine can be called may often change, a great deal more mispredicts can happen with this type of jump more often occurring than a standard jump-and-link (which was more common in the other test codes provided).

However, the first competition code and the baseline drastically improved in prediction rates, both approaching 50% decreases in the number of branches mispredicted. In the case of competition code one, where there is a higher amount of pointer chasing that is used as well as many branches taken to iterate through a linked list, this single function and single conditional check that is called from the same location always may be predicted and logged in our BTB for constant reuse. As a result, our performance improves from BTB usage. Similarly, again, with the introduction of our BTB in the baseline code that uses more common iterative steps in its checksum calculation, we see the rate also decrease by a factor of 45.8% here.

One final interesting note, however, is that a noticeable decrease in stalls from the instruction and data cache were achieved after involving the BTB in execution. This can be explained by the ability for correct taken predictions to not force loading new instructions that have not been visited, but instead reuse the instructions and jump in based on what another cache (the BTB, in its effective construction) has logged. Interestingly, given the locality of data when iterating through points as well, the number of data stalls also declines due to branch prediction being taken and instructions not returning immediately (to a state where another branch unseen before, *jalr* through *ret*, is encountered).

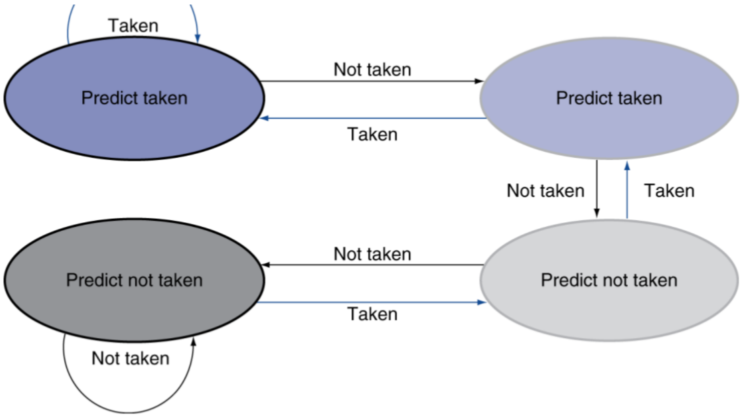
Therefore, while the performance boost was not as noticeable as with the four-way cache modification, we conclude that the addition of the BTB in nearly all cases allows one to exceed the performance of the original, two-way, true LRU equivalent L2 cache by pure cycle count. Specifically, there was a **6.67% performance increase** on final benchmark **at 96% original frequency**, thus retaining a high speed but with a slightly higher drop from a combinational logic critical path. The performance value is again computed as the sum of the new result over the old result for cycles of execution, divided by the number of test codes run (four, in this case here).

Advanced: GShare Global Branch Predictor with Two-Bit Local Predictor Entries

Design Overview

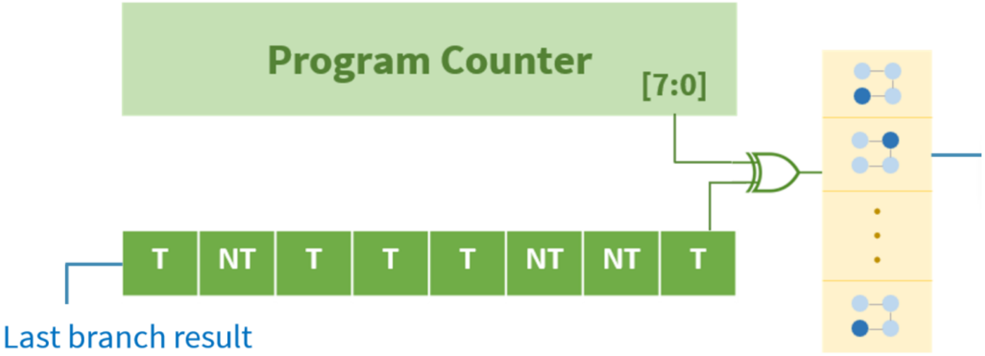
The aforementioned BTB helps for jumps, but some branch cases depend on the current state of the whole machine, instead of just a single PC. Indeed, especially with loops with multiple branches, we saw that we incurred stalls for branches we should not have predicted in all cases, especially in the data sets that were given for codes two and three in the last section. Therefore, we wanted to track tendency and local state in programs and the system as a whole, not relying just on the branch results of a specific target (unless they yielded a jump) but instead trusting how typically a branch should be taken given the history of branches coming before it.

First, we must look at what a local branch predictor appears as. This takes in a PC of some sort and given an index through a table that is kept for each of the PCs available, a state machine outputs whether items are predicted to be taken or not taken given the history of that PC. When the branch is evaluated, the predictor is then updated with the result of that branch, to possibly update the decision to take or not take for the next run. Typically, these state machines will be two-bit local predictors, where one can decide strongly not to take (meaning it would take two transitions taking a branch to change the decision), weakly not to take (one transition to move to taking a branch), weakly *to* take, and strongly to take. Borrowing from the lecture slides provided again and rehashed from an earlier section of this report, the state diagram appears as follows:



According to Dan Luu in his talk for *localhost*, one of the tradeoffs typically taken when looking at a global scheme is that bits are dedicated to either history or address, but not both. History is typically used with representation of a shift register, where a serial input shifts out the stalest data and brings in the newest to give a new index entry that can be used within a table of local branch predictors. However, this history does not tell the entire story of what a branch may correspond to, and indeed a branch that has been visited multiple times when the system is in a specific state (for example, a system variable continues to meet a condition) should hash to an index it influences among local branch predictors as well.

Therefore, we opt for what is defined as a *gshare* branch predictor, taking both the history of the entire system as originally intended by our observations but also considering the specific location of a branch instruction (or *any* instruction) when encountered. This is connected to a table of local branch predictors as defined earlier, where the index that one uses to update and read prediction is dependent on a hash between the global history register and the PC value currently used. This hash is computed using an XOR structure between a certain number of bits in the program counter, particularly the lower bits starting from bit 2 (since no instructions are located with bits 1 and 0 ever anything but set to ground), and higher bits ending at bit 9. In our implementation, these are then individually XORed with the eight bits (and thus, last eight branch results) of the program or system that is currently running, yielding this index. Initially, the index starts at strongly not taken, as no previous state has been currently defined but the ability to simply update the PC sequentially is far simpler than updating (and correcting later) if a branch is indeed predicted but not taken. Furthermore, information would need to be known about a branch first before we could take the branch in an earlier stage than EX, which means that the branch must be encountered and taken first to change the state of the pipeline.



Because of creating an index of eight bits, there is thus a table of 256 four-state local predictors, which again default to strongly not taken (due to least stall cost) on each entry. The first bit that is output for the state of all of these corresponds to the taken bit, whereas the zeroth bit corresponds to the strong (1 if Taken, 0 if Not Taken) versus weak attribute of the state.

The global branch predictor can be mostly summarized thus to this logic, but still needs to have the shift register updated when a branch is indeed encountered. Therefore, a variable is kept and output with the index of the branch predictor and sent through the pipeline as an instruction executes, should a writeback be necessary and the table entry need updating. Indeed, whenever a branch statement is found, the first step taken by the pipeline is to send an update signal to the global history register indicating the result of the branch (taken or not taken, directly passed in as br_en from the pipeline). Second, the index that was calculated with the instruction is passed back into the branch predictor to update state, using the same br_en signal as a transition. Should a 0 be passed, the state at the index will either go from strongly taken to weakly taken, weakly taken to weakly not taken, or weakly not taken to strongly not taken. The opposite direction of update ensues when br_en is 1 here.

Once this is completed, the branch predictor is plugged in to the external branch signal slot that is compared against BTB availability, discussed in the previous section. Note that this implementation is limited only for branches, as jumps are already assumed and handled by the BTB. Therefore, no branches are used to update the state of the global history shift register, nor are the states found in the table after indexing with an XOR of PC bits here updated.

Testing

While we are in theory only introducing one signal to our prediction scheme by testing this branch predictor with the help of a BTB, there should still be tests to just check on this misprediction logic first completed here. Once misprediction can be verified according to the parameters of the previous testing section, we may continue here.

It should be noted that all tests and performance from this point forth will also include the BTB, as there is no way to calculate taken early without additional adder logic or ID logic that could finish *and determine* the branch calculation itself. (If this was to happen, ultimately, then the utility of the branch predictor would be low anyway. However, adding such a scheme would significantly cause performance issues in terms of maximum frequency resulting, especially if connected to instruction cache output.)

Only once this misprediction logic and the PC can be verified as working together and clearing data on proper mispredicts can the BTB then be tested. Our first verification comes from known highly iterative code, which is especially found in the case of competition code two that we have been provided (as well as the baseline). After a few iterations, the same function will run through a large set of nodes or data elements, constantly taking a branch or denying a certain terminal condition also defined by a branch to create a well-set system state. We should see the index associated with this particular state (when XORed between PC and global history) start to be updated from strongly not taken to strongly taken over the course of a few cycles, even as the history jumps from one index to a second one (as T/NT/T/NT… patterns will cycle to NT/T/NT/T… patterns before returning to T/NT/T/NT…). Further, the update should be associated with a write three states after the original index was computed, as the IF stage will carry the original branch prediction and MEM stage will write back the result of a branch here.

After this has been verified, we can run the previous section’s test code to ensure that all mispredictions are handled correctly still, but that branch prediction only occurs whenever the global state has been flooded correctly to deem it ready. The only issue with the previous test code, however, is that this will never happen due to the low amount of iterations that are being incurred. Therefore, a separate test with a larger number of iterations is thus used:

```
riscv_btb_earlybranch.s:
.align 4
.section .text
.globl _start
    # We will iterate 40 times, over a series of branches and jumps,
to flood the global branch predictor with a certain:
    # NT/T/NT/T/NT/T/NT/T OR T/NT/T/NT/T/NT/T/NT pattern. On one of
these patterns, our branch predictor will be trained by the
    # iterations to be strongly not taken - since the branch is
missed each time. On the other pattern, our branch predictor will be
    # trained for the corresponding local branch predictor TO take
the branch. The pattern is broken after 30 iterations complete.

_start:
    addi x1, x0, 40
    add x2, x0, x0
loop:
    addi x1, x1, -1
    addi x2, x2, 1
    beq x0, x1, halt
testone:
    add x3, x2, x0 # Add computation that sets x3 to x2, force taken
    beq x3, x2, loop

halt:                 # Infinite loop to keep the processor
    beq x0, x0, halt  # from trying to execute any data below.
```

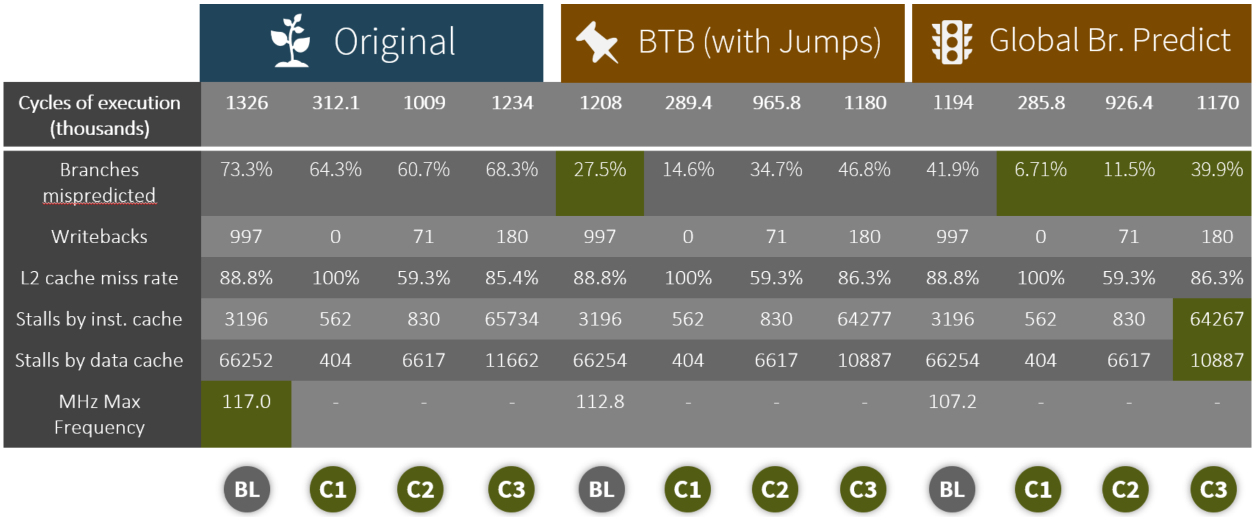
As noted by the test code commentary from above, this will force a branch predictor to eventually take a pattern of strongly taken on one state, and strongly not taken on another. Therefore, we should observe that the T/NT… pattern that eventually builds has a state of strongly not taken, as the branch that jumps to halt at end of iteration has now been not taken up to 39 times. Meanwhile, the NT/T… pattern should build to strongly taken, and with a BTB value ready from a successful jump before, the branch causing a loop should be executed quickly after here with branch taken signal raised in the pipeline. On the final execution, when the halt line is taken, a branch mispredict should be raised as there was an assumption not to take the branch, and yet it has finally been taken as the number of iterations went down to zero. One should also ensure that the state previously stuck to strongly not taken updates to weakly not taken as a result.

Even on the halt line, further inspection can be done to ensure that the first few branches are mispredicted and cause excessive stalls. This is caused by the global history register needing to reassign its state from a NT/T… pattern to all NT results again, before it can hash into the index that matches the PC bits values (since all NT is a sequence of all zeroes, and any value XOR with zero is itself) and start updating the state from strongly not taken to strongly taken.

It is worth noting that, again when considering a connection to the BTB, that this entire prediction mechanism is predicated on the idea of not having self-modifying code. Should this happen, revisions would be necessary to check the integrity of the address as a branch, jump, or jump and link register altogether. Also, branch prediction would not offer any speedup from this code had the BTB not existed, as calculation would have happened in the EX stage and branch resolved here on every iteration instead.

Performance Analysis

Prior to evaluating performance against the BTB and against the original code, it is worth noting a phenomenon resulting from running the above code with and without the branch predictor. Should no branch prediction be enabled here, we experience a type of *warm-up delay* where the branch predictor must both prepare the global history register with a constant pattern *and* the state hashed because of XORing PC and global branch history register bits together. Even when flooding the branch history with a specific T/NT sequence, it still takes **14** iterations before our branch predictor correctly predicted to take an iterative branch. If one was to instead take out the branch predictor and just use the branch target buffer, then the delay for these iterations would be saved and only the first branch would be calculated as a miss here.



	Original				BTB (with Jumps)				Global Br. Predict			
	BL	C1	C2	C3	BL	C1	C2	C3	BL	C1	C2	C3
Cycles of execution (thousands)	1326	312.1	1009	1234	1208	289.4	965.8	1180	1194	285.8	926.4	1170
Branches mispredicted	73.3%	64.3%	60.7%	68.3%	27.5%	14.6%	34.7%	46.8%	41.9%	6.71%	11.5%	39.9%
Writebacks	997	0	71	180	997	0	71	180	997	0	71	180
L2 cache miss rate	88.8%	100%	59.3%	85.4%	88.8%	100%	59.3%	86.3%	88.8%	100%	59.3%	86.3%
Stalls by inst. cache	3196	562	830	65734	3196	562	830	64277	3196	562	830	64267
Stalls by data cache	66252	404	6617	11662	66254	404	6617	10887	66254	404	6617	10887
MHz Max Frequency	117.0	-	-	-	112.8	-	-	-	107.2	-	-	-

Evaluating performance with branch prediction models can be done by the same comparison strategy used in the previous section, but given the use of a BTB to allow the branch predictor to work, it is still useful to run tests where the BTB is isolated and see if there is a true benefit to involving the branch predictor. Indeed, when comparing global branch predictor alone to original, there again are far less branches mispredicted and the result caused a decrease in the number of cycles as a result. Indeed, however, though a frequency drop of 9.8 MHz was incurred, significant improvement to the second competition code, coupled with mild improvement on the first and third competition codes, were also experienced.

Prior to exploring further with branches mispredicted, it is worth noting that a slight decrease in stalls from the instruction and data cache were again achieved, even more so than when compared to the BTB. This seems to indicate that considering state of the program instead of just jumping and forcing load of new instructions as soon as a verified branch is seen saves on performance at the memory level as well. After all, the correct not taken predictions can be made even more so than with a BTB when the state of the program or system is still fluctuating, and all internal branch prediction signals have not converged on a taken solution.

The statistic for branches mispredicted, however, was by far the most interesting and most improved across all test codes, even when considered against the BTB. Indeed, single digit branch mispredictions were found in the high locality test code of competition code one, where not taken could be responsibly used when jumping in and out of subroutines, but an aggressive taken prediction result could be returned when the state of the program was built back through a few iterations on data.

The second competition code also significantly improved compared to the original, this time by a factor of 49.2% compared to the 26% improvement that was provided by the BTB in its comparison. Again, this is likely attributed to building state of the program and making responsible decisions regarding branches not always taken for certain internal conditions in highly iterative code. Indeed, this second competition code is highly iterative, but a series of taken / not taken histories can be recorded by the register properly. Further, the state that is hashed to from the branch predictor can more clearly indicate when a condition has been taken and some code was skipped, versus the typical branch behavior with the condition evaluated alternatively.

Interestingly, the same result comes about even when calculating a minimum spanning tree, where a slight improvement is found compared to the BTB. However, the baseline did make a case for warm-up delay becoming detrimental to execution, with the number of branches mispredicted nearly 14.4% higher than when the same code was run on a system with only a BTB to drive predictions. It is hypothesized that, due to the small length of iterations that are used in this test code versus others, no one state could be properly set to taken or not taken given a constantly fluctuating global branch history.

Therefore, while the performance boost was not as noticeable as with the four-way cache modification, it performed better on average than the BTB, and thus we conclude that the addition of the branch prediction in all cases allows one to exceed the performance of the original, two-way, true LRU equivalent L2 cache by pure cycle count. Specifically, there was an **8.67% performance increase** on final benchmark **at 92% original frequency**, thus retaining a high speed but with even slightly higher drop from a combinational logic critical path, after being connected to the BTB here. The performance value is again computed as the sum of the new result over the old result for cycles of execution, divided by the number of test codes run (four, in this case here).

Conclusion

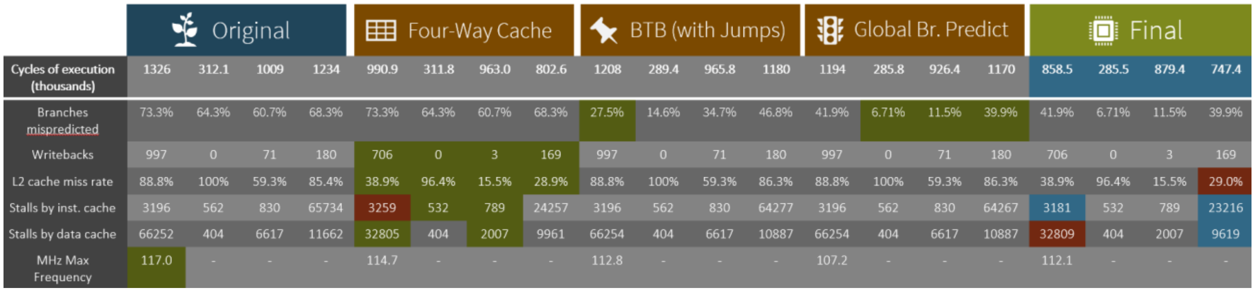
Through utilization of an academia-accepted and industry-used RISC-V instruction set architecture, we have adapted the RV32I microprocessor from single-stage to multi-stage pipeline and demonstrated performance increase effects. In doing so, we offer a theoretical speedup determined by the latency of each register (simulated at 0 ns) and the number of stages, with five stages of *Instruction Fetch, Instruction Decode,* and three Execute (*Execute, Memory Read/Write, and Write-Back to Register File*) stages appended at the end. These execute stages are split to diversify resources that are used by any one instruction, such that there is more full utilization of all components connected to and on the dye with our processor. The amount of throughput, as a result, also is found to generally increase. Thus, our primary motivations in building this pipelined processor with additional branching and extended-way cache features was to explore performance improvement through increased throughput when compared to single-cycle or single-stage models.

With latency and stalls incurred from unavailable memory in local cache structures, however, our design also forced us to consider multiple levels of said cache, where the first level was split between instructions and data to minimize evictions based on what type of data or code is seen at a memory address. Many times, we waited for data to write back just because there were misses in L2, only for this data to be referenced soon again and cause too many misses. To decrease number of writebacks and to retain recent data, we extended our level two cache to four ways (instead of more indices) for larger operations and shared libraries with commonly called subroutines. Through this action, a 23.1% performance increase on final benchmark was observed at 98% original frequency.

Additionally, in early branch tests and on benchmarks that jumped based on large initial counters, some of our execution was waiting for branching to write back from WB even though no value change happened from the EX stage. Since conditionals are managed in EX as well, we could save latches and runtime by simply rerouting signals and adding an opcode check. This afforded up to 166.7% performance increases on codes that were simply constructed of near all branches here.

Often, one skips data lines or routines, or makes unconditional branches at the end of loops, with j. By adding jump support, we can predict taken on instructions known to be jumps just by their PC. If there is a branch present instead, we still have data ready to immediately update PC. Therefore, a branch target buffer with jump support was added, giving a 6.67% performance increase on final benchmark at 96% original frequency.

Finally, while the BTB helps for jumps, some branch cases depend on the current state of the whole machine, instead of just a single PC. Indeed, especially with loops with multiple branches, we saw that we incurred stalls for branches we should not have predicted in all cases. Therefore, we wanted to track tendency, and could do so with a gshare global branch predictor with local look-up. Addition of this led to 8.67% performance increase on final benchmark at 92% original frequency (with BTB).



	Original				Four-Way Cache				BTB (with Jumps)				Global Br. Predict				Final			
Cycles of execution (thousands)	1326	312.1	1009	1234	990.9	311.8	963.0	802.6	1208	289.4	965.8	1180	1194	285.8	926.4	1170	858.5	285.5	879.4	747.4
Branches mispredicted	73.3%	64.3%	60.7%	68.3%	73.3%	64.3%	60.7%	68.3%	27.5%	14.6%	34.7%	46.8%	41.9%	6.71%	11.5%	39.9%	41.9%	6.71%	11.5%	39.9%
Writebacks	997	0	71	180	706	0	3	169	997	0	71	180	997	0	71	180	706	0	3	169
L2 cache miss rate	88.8%	100%	59.3%	85.4%	38.9%	96.4%	15.5%	28.9%	88.8%	100%	59.3%	86.3%	88.8%	100%	59.3%	86.3%	38.9%	96.4%	15.5%	29.0%
Stalls by inst. cache	3196	562	830	65734	3259	532	789	24257	3196	562	830	64277	3196	562	830	64267	3181	532	789	23216
Stalls by data cache	66252	404	6617	11662	32805	404	2007	9961	66254	404	6617	10887	66254	404	6617	10887	32809	404	2007	9619
MHz Max Frequency	117.0	-	-	-	114.7	-	-	-	112.8	-	-	-	107.2	-	-	-	112.1	-	-	-

When combining all these features together, our combined processor greatly performs even late iterations of a pipelined processor, demonstrating that these advanced features have significant effect in reducing runtime and should be generally considered for modern computing design. Where four-way caches faltered in the original baseline code, correct branch prediction gets the correct instructions in this low-locality program. Further, the code for competition code one (column two of each set of tests) work sequentially, missing to get instructions, but an extra way in this case helps when data is eventually received in a swarm of loads and stores. Through competition two, we see drastic improvement in the branch prediction scheme with a 50% drop in total mispredicts aiding this highly iterative (for high duration) code. Finally, in the final code where a special case of *ret* (*jalr x0, x1, 0*) was caught and corrected branch predictors, the routine called many times on separate data objects corresponding to minimum spanning tree caused a significant drop in cache miss rate and stalls in the pipeline. In the above, blue indicates where our final was better with all three features versus only one. Clearly, the result on each test code demonstrated that having all features working together in the pipelined design allowed for the best performance. However, some individual cases have also reported better than usual, such as in instruction and data cache stalls due to four-way cache mostly improving physical memory read/write latency, while branch predictors cause stalls of their own that would have otherwise been taken by expensive lower cache and physical memory reads here. Note, however, that this comes at a hit for frequency, with these data points being measured at 112.1 MHz maximum frequency compared to some of the higher frequencies allowed in less complicated pipelines here.

While we are ecstatic about the implications that this proves for creating pipelines with modern branch, memory hierarchy, and ALU features, the work completed should be next evaluated in the space of energy consumption. According to Hartstein and Puzak's *Optimum Power/Performance Pipeline Depth*, it is discovered that there are incredibly high energy costs as the amount of stages per pipeline increases. While pipeline performance was already limited by number of stages in an earlier IBM report, this paper suggests that some processor designs would offer better power and energy benefits if no pipelining was used at all. Indeed, mobile devices today still utilize pipelining for some of the benefits found here, but limit to two or similar stages due to the high cost incurred in (for example) battery-powered devices here. Therefore, we look forward to continuing our work with this pipeline to explore some of the benefits in reducing number of stages or combining control logic to decrease power usage, if a reasonable frequency can still be achieved. As a result, our current processor can be evaluated against as a baseline in a continued search for optimum power and performance pipeline depth when dealing with the emerging RISC-V architecture.