# ECE 411

Spring 2021

# Final Project Report

Michal Kalita, Maximilian Goldstein, Derek Alyne
mkalita2, mgg2, dalyne2

# Table of Contents

# Introduction

The motivation behind this project was to gain hands-on experience with the computer architecture related materials introduced in lecture, demonstrate a deep understanding of hardware-design principles and programming paradigms, and develop skills related to the effective communication and coordination of individual efforts among a larger team to achieve an ambitious and technically involved goal. The deliverables we intended to produce by the final checkpoint was a five-stage pipelined RISC-V processor, supporting a subset of the RISC-V ISA as specified in the official RISC-V documentation, and offers improved performance compared to those processors from the second checkpoint and previous MPs through the integration of specific advanced design features chosen to optimize execution of the provided competition codes, as well as written documentations explaining new aspects of the design and underlying motivations, overall progress of the project, and planning of future goals and individual responsibilities ahead of the next checkpoint. Each of these motivations were addressed, evaluated, and refined over the course of the project, which can be broken down into a series of four checkpoints. The first two checkpoints primarily tested our understanding of the pipelined architecture and ability to operate as a group. Checkpoints three and four evaluated our ability to profile the execution of the competition codes, identify performance bottlenecks and tradeoffs, and develop solutions to minimize penalties and increase efficiency of the pipelined design in response. The first two objectives played into our conceptual understanding for the impacts that different execution flows have on a pipelined design. Because the competition codes were known, the advanced features we chose to implement were not arbitrarily selected, instead we based decisions on what aspects of the design required the most performance boosting. The third objective helped develop skills related to modular design and unit testing. The final checkpoint focused on optimizing the design to achieve specific timing and power constraints.

## Project Overview

By the completion of checkpoint one, we demonstrated a conceptual understanding of pipelining by implementing the different pipeline stages and control logic in separate hardware modules and combining them as part of the CPU datapath. A high-level understanding of each pipeline stage and its role motivated how we organized the flow of data between each stage. Additionally, checkpoint one established individual expectations for the team's ability to coordinate design, development, and debugging efforts asynchronously. At first, the codebase was developed synchronously using the LiveShare real-time code editing feature of VSCode. However, this approach to collaboration proved to be a bottleneck to productivity since each team member had different windows of availability. By the later checkpoints, contributions were made asynchronously via git and each team member maintained several branches with changes relevant to his responsibilities for that checkpoint. After verifying the correctness of one's proposed changes, a merge to the master branch was made.

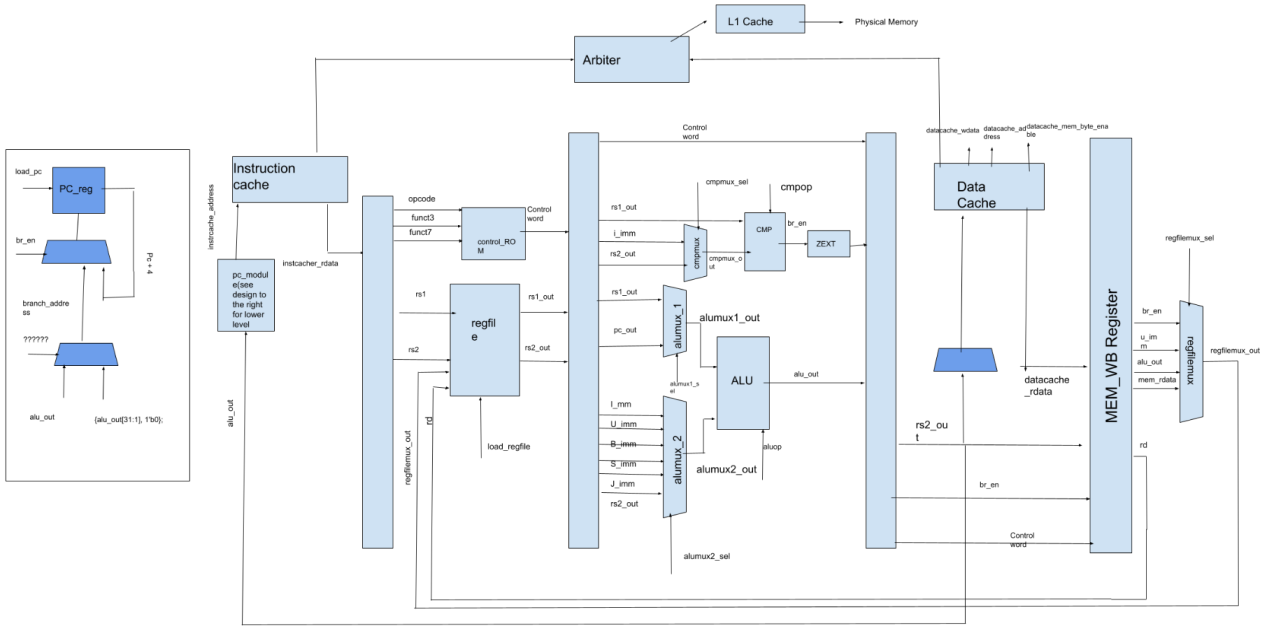
Through completion of checkpoint two, we further improved the design by reworking aspects related to handling dependencies between instructions and data, and introducing a memory subsystem to the top-level of the design. The memory subsystem was composed of an instruction cache, data cache, arbiter unit, and cacheline adapter. In turn, the CPU modules from the earlier checkpoint were encapsulated into a single module instantiated at the top-level and interfacing with the memory subsystem. This checkpoint introduced additional complexity to the design by making us account for non-uniform memory access patterns and latencies, and tested our understanding of various design concepts needed by the pipelined processor. Namely, this checkpoint required the team to rework the design to anticipate and accommodate data hazards, inter-instruction dependencies, and both conditional or unconditional branch instructions. A forwarding unit and static branch prediction unit were added to the datapath. Further, this checkpoint forced us to reconsider how we verify and debug the functionality of another individual's work and the overall design. Through this process, we developed better approaches for debugging and decoupling design components from each other to determine from which module errors were propagating from.

For checkpoints three and four, team efforts were focused on introducing new features to the design to optimize performance and modify aspects of the design to reduce critical paths and meet target constraints for eligibility in the competition. Since the advanced features chosen largely affected different aspects of the design or stages of the pipeline, our initial hope to develop each independently and concurrently with another. Further, with regards to checkpoint four, the team focused on evaluating the overall performance of the processor and considered eliminating components that failed or negatively impacted the pipeline processor. This meant reworking the design to minimize critical paths and reconfiguring the compilation and design fitting settings of the project to achieve a target operating frequency of at least 100MHz. Further discussion about the responsibilities and contributions of each team member at each checkpoint are described in the next section. We have left discussion of the project management complications to a dedicated section preceding the conclusion of the report.

# Milestones

As explained in the introductory sections of the report, the project was broken down into a series of checkpoints. Each checkpoint builds on the work from the previous ones, and the final checkpoint-- the design competition-- was intended to demonstrate our pipeline's peak performance. More detail into the goals of each checkpoint, distribution of work, and objectives made for checkpoint 2 are given in the progress reports below.

## Checkpoint 1

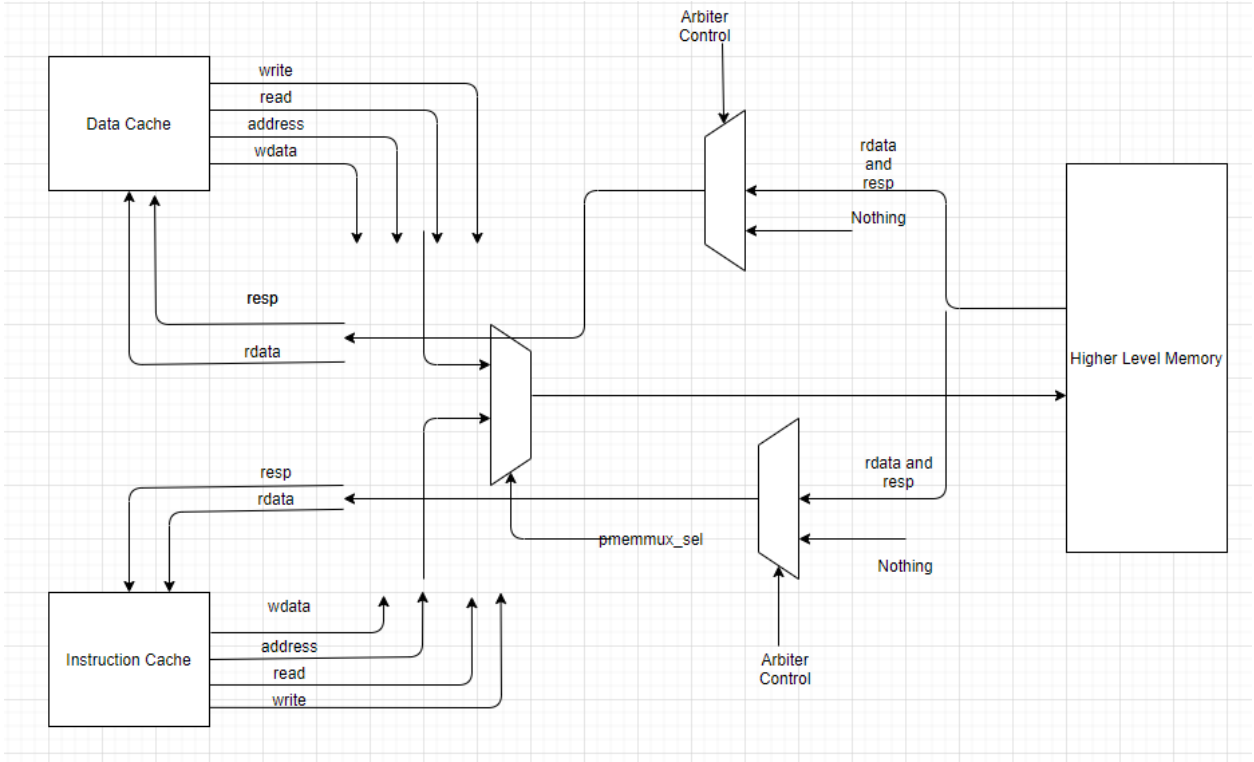



_____For the first checkpoint, the deliverable we produced was a basic pipelined processor which implemented support for a subset of instructions defined under the RISC-V ISA specification. Also, we familiarized ourselves with the group dynamic and established expectations for individual responsibility and availability. Since the codebase was sparse at the beginning of this checkpoint, there was less flexibility to work asynchronously because it was important that each individual working on different core components of the design shared a common vision and high-level understanding for how the components were going to interoperate. Each team member contributed to the design of the Control ROM for the processor and its interface with the pipeline stage registers. In particular, the responsibility of the design of the datapath's four inter-stage pipeline register modules, IF_ID ,ID_EX, EX_MEM, and MEM_WB, was divided evenly among the group. The control ROM  module, responsible for setting signals of the control word structure based on the opcode, funct3, and funct7 fields of the decoded instruction, was developed largely by Max. The datapath module, responsible for connecting pipeline stages together and carrying out instructions by producing certain outputs given a set of inputs and associated control signals, was developed jointly by Michal and Derek.

The efforts to integrate the two components of the design checkpoint and test overall functionality was a group endeavor. Testing was done incrementally and on the design as a whole by writing small, basic RISC-V test programs, verifying execution, and reiterating the process with more sophisticated programs until the design was eventually capable of correctly executing the provided code for the checkpoint one. Notably, Michal proved to be significantly adept at debugging various aspects of the design while Derek and Max focused on wiring up the design to the top-level design and verifying the modules, mp4.sv and top.sv, respectively.

For checkpoint two, we had planned to introduce a memory subsystem to the design, a forwarding unit to handle data hazards, anda static branch predictor. To distribute the design effort for the next checkpoint, Michal will share the responsibilities involved with designing, implementing, and integrating the forwarding unit and branch predictor with the datapath. Max will be responsible for implementing the memory subsystem by instantiating L1 caches and the cacheline adapter unit at the top-level of the design, mp4.sv, and designing a memory arbiter unit to service requests serially from the data cache or instruction cache.

**Checkpoint 2**



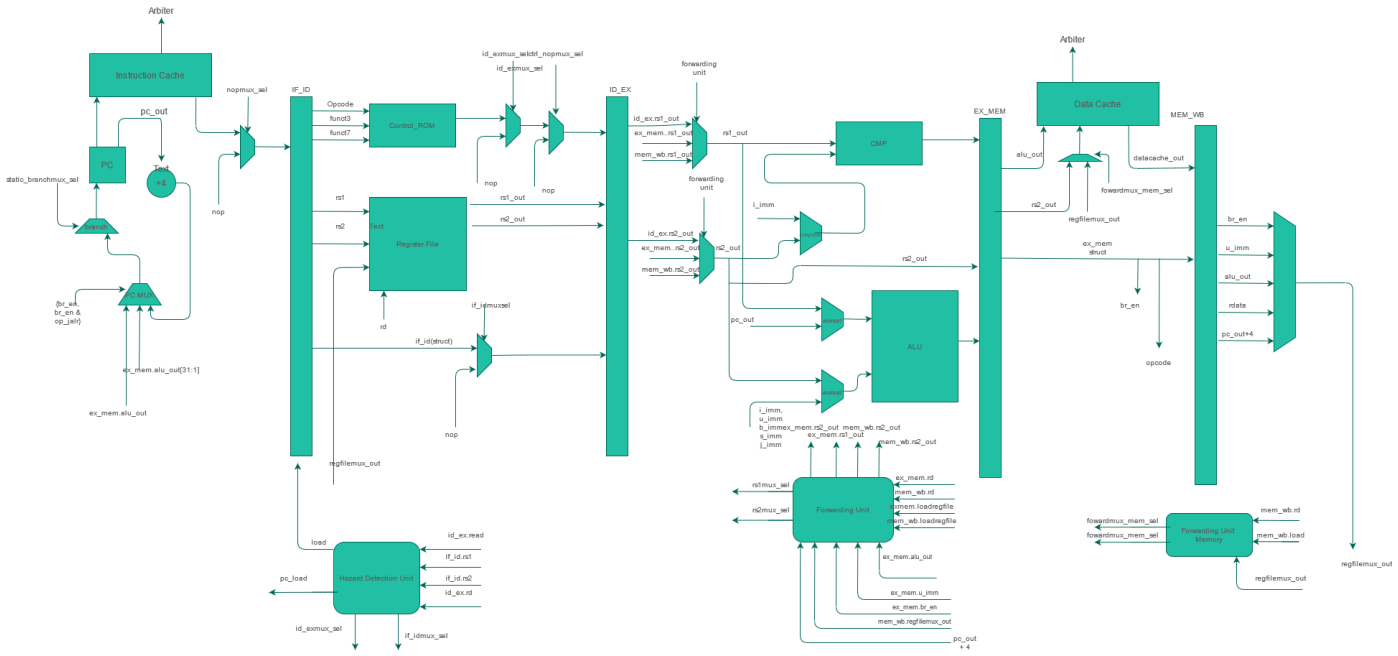

**Description**: This is the Arbiter Datapath. The pmemmux_sel signal chooses which cache's data to forward. Arbiter control are select signals from the arbiter control that decide whether to output memory data or nothing.

_____For the second checkpoint, we introduce the concepts of data forwarding, hazard detection, and static branch prediction. The forwarding module's purpose was to anticipate and mitigate data hazards occuring between various pipeline stages, such as read after writes, write

after reads, write after writes, and read after reads. Dependencies between instructions in the pipeline were checked between the Execute, Memory, and Writeback stages. The dependencies between stages will be denoted as (preceding stage)-(succeeding stage), e.g. EX-MEM for a Execute and Memory stage dependency. There was also an additional hazard detection unit that was responsible for checking if there will be a read from a register after a load, with the exception of stores, between the ID-EX stages. The motivation for the hazard detection between these stages was to stall the pipeline so that the instruction reading from the register file would not be receiving forwarding data from the memory stage. Instead, it receives data from the writeback stage since it will be available at the beginning of the clock cycle unlike in the memory stage where the load instruction has to wait for a cache hit or miss to be processed in order for the data to be ready. The EX-MEM and EX-WB dependencies were mainly for reads from registers after an instruction loads or writes data to them. The MEM-WB dependency was solely meant for a load followed by a store. The reason we do not want to take care of this dependency in the ID-EX stages is because we can save a clock cycle of pipeline stalling by forwarding the data from a load instruction in the WB stage to a store instruction in the MEM stage because the MEM stage is when the store instruction uses the register file data. The efforts to design, implement, and integrate the forwarding unit were undertaken primarily by Michal. Further, a static not-taken branch prediction unit was introduced to the pipelined datapath. The branch predictor consistently kept predicting that a branch would not be taken, so that the pipeline would not have to wait until the output of the ALU is ready, instead the next instruction may be fetched on the clock cycle right after the branch instruction. If prediction was wrong, the pipeline would detect this in the MEM stage and flush all previous instructions loaded into the pipeline. While this type of branch prediction improves performance if a branch is consistently not taken, otherwise, the pipeline wastes 3 clock cycles to flush the branch misprediction and reload the correct branch address. While this type of branch prediction does offer somewhat of a performance boost compared to not having any type of branch prediction at all, it is not ideal. Improving the limited performance boost from the static branch predictor will motivate the design for our advanced features in the next checkpoint. The design and integration of the branch predictor logic was done by Michal. Next, efforts to design and introduce the memory subsystem to the top-level of the design was assigned to Max for this checkpoint. This responsibility included instantiating L1 caches in the mp4.sv module and connecting these caches with the input and output ports of burst memory specified in the top.sv and source_tb.sv modules. Further, a cacheline adapter module was added to mp4.sv to transfer data between physical burst memory and the caches. For the specific goals of this checkpoint, a memory arbiter unit was introduced to the mp4.sv module to manage and serve requests made by the processor to both instruction and data caches. The arbiter unit served as the interface between the cache subsystem, processor, and physical burst memory. The responsibility to design, test, and integrate the arbiter unit with the rest of the design was placed on Max. Max developed a test bench to check basic functionality of the arbiter and correctness of state transitions when concurrent requests for both data and instructions were raised by the processor.

For the next checkpoint, we intend to verify the correctness of the design when executing the competition codes as well as introduce advanced features intended to minimize observed performance bottlenecks and optimize the execution of the given competition codes. With respect to the division of responsibilities for the next checkpoint, Michal will design and implement pipelined L1 caches and L2 cache, Max will implement a Tournament Branch Predictor and Return Address Stack, and Derek will implement pLRU and Eviction Write Buffer modules.

## Checkpoint 3




**Description**: Final Datapath of our pipelined processor including forwarding unit, branch prediction, and hazard detection.

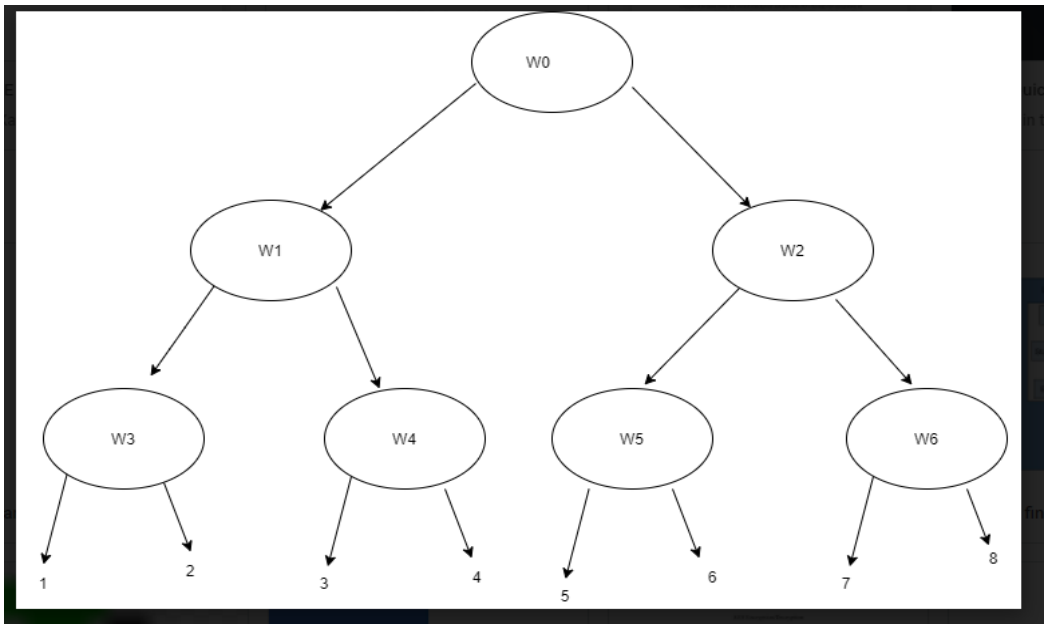
For the third checkpoint, we ironed out some remaining bugs in our CPU implementation, and attempted to develop and introduce our planned advanced design features. Much of the debugging efforts were taken on by Michal and Max, and they worked to ensure the design-- without advanced features-- could properly execute the mp4-cp3 code and previous checkpoint codes without regressions. Many of the bugs were uncovered by peer code-review in which someone other than the original programmer looks over the code and suggests changes or clarifications to the code. This forced all of us to better consider design decisions and understand parts of the design one originally did not author. Michal discovered two significant bugs, one of which was in the arbiter unit and the other in the forwarding unit. Max discovered more minor bugs in the datapath and forwarding units which could introduce “don’t cares” into the pipeline and produced unintended behaviors. For the advanced features implemented for this checkpoint, Michal adapted the MP3 cache to serve as a foundation for a pipelined cache implementation.

This advanced feature would increase the overall throughput of the design by allowing the memory subsystem to serve some cache requests immediately while others were waiting for data from physical memory. The testing strategy used to verify the pipeline caches was to develop small programs in assembler that would be easy to trace and compare the final register values against the same program executed on an MP2 or MP3 design. Derek attempted to implement a 4-way associative cache(pLRU) and EWB for this checkpoint. The pLRU would improve performance by decreasing the amount of bookkeeping and hardware complexity involved in tracking usage of cache entries. Given a sequence of items and events acting over those items, the pLRU chooses to evict the item that is most likely to no longer have relevance. The EWB would improve performance by functioning as a temporary storage for dirty evicts from the L1 cache. Combined with the pipelined cache design feature, the EWB enables the CPU to serve cache misses right after a dirty evict and without waiting for a write-back to complete. The pLRU and EWB were tested using basic assembly programs and comparing the final register values with a working MP3. Max implemented a Return Address Stack and Tournament Branch Predictor for this checkpoint. The RAS would improve performance by removing the need to lookup the next PC value following a function call after a function returns. This feature was implemented by creating a multidimensional register and adjusting an index value pointing to the top of the stack when an op_jalr executes following an op_jal. Coupled with the tournament branch predictor, which improves performance by using information about previous branches using both a global branch predictor and local branch predictor to make predictions about whether the next branch is taken, these features improved the overall performance by minimizing the amount of times the pipeline needed to be flushed when a misprediction occurs. These features were verified using simple programs and comparisons with an MP3 design.

For the next checkpoint, we intend to further optimize the design for the competition and iron out remaining bugs in the branch predictor and pipelined cache features introduced in this checkpoint. Also, we want to review the performance analytics for the design made available through Quartus to reduce critical paths and area complexity of the design wherever possible. Michal will lead the debugging efforts and remaining advanced feature design integrations, and Max and Derek will focus on simplifying various aspects of the design according to the analysis tools and optimizing performance for the competition codes.

# Advanced Features

## PLRU




*Motivation:*

A pLRU is meant to evict data that is least likely to be relevant to current execution context. The structure of the pLRU is a binary tree, which reduces hardware complexity though efficient lookups and cache access. For our implementation, we constructed a three level pLRU, which was an 8 way associative cache.

*Testing Methodology:*

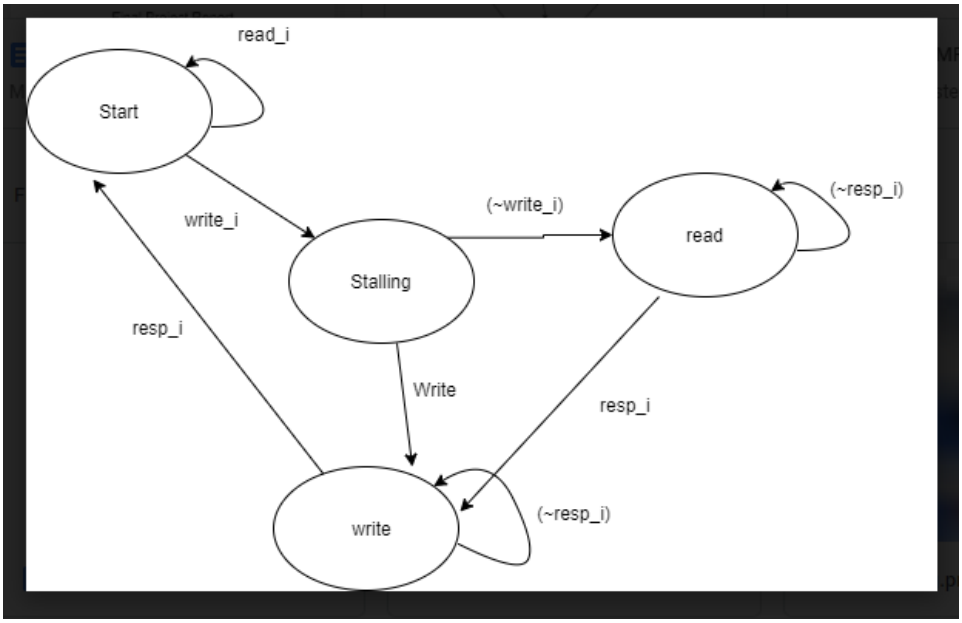
The methods used to test the pLRU was to instantiate it in the top level of the design and monitor its internal state during the execution of the processor without connecting the output signals to other design components. This allowed us to monitor the internal state and catch hardware bugs and flaws in the implementation, but otherwise proved to be insufficient.

*What went wrong:*

The challenges faced with this feature was its integration with our L2 cache. We discovered that the pLRU output was not selecting the closest approximation to the least recently used cacheline, and our L2 cache would sometimes incorrectly evict cache lines. In our design, we go left if our bit select is 0 and right, if our select is 1. A possible reason for our cache incorrectly evicting data, is that we did not correctly implement one of our pLRU bit select muxes, which would explain why we were selecting incorrect cache-lines for eviction. The pLRU would have reduced the critical path from just using an LRU as it makes use of an efficient binary search structure, which reduced hardware complexity compared to an LRU. It also reduces power consumption, offering a reduction to operating frequency, as its design uses less hardware.

*Hypothesized Metrics*: Implementing an 8-way pLRU would decrease the area used by hardware because implementing the code is simpler than normal LRU, and less combinational logic and arrays means less transistors which means less area is taken up by the design.The power consumption would be reduced due to the implementation of the pLRU because there paths to check when evicting and less bits to store to decide which way to evict. Since an L2 cache isn't bound by the one cycle hit limitation like the L1 caches, implementing the pLRU shouldn't affect the critical path, especially since it requires less combinational logic and arrays anyways. The feature should also reduce performance, since it is expected to increase the number of misses, since the pLRU is merely an approximation of the true LRU policy. The tradeoffs associated with this advanced design are that it lowers power consumption, reduces area, and reduces latency and overhead; however, it reduces performance and speed. The code for the pLRU is much simpler to implement than true LRU for an 8-way cache because it requires less bits to keep track of the least recently used block, specifically N-1 bits where N is the value of associativity.

## Eviction Write Buffer

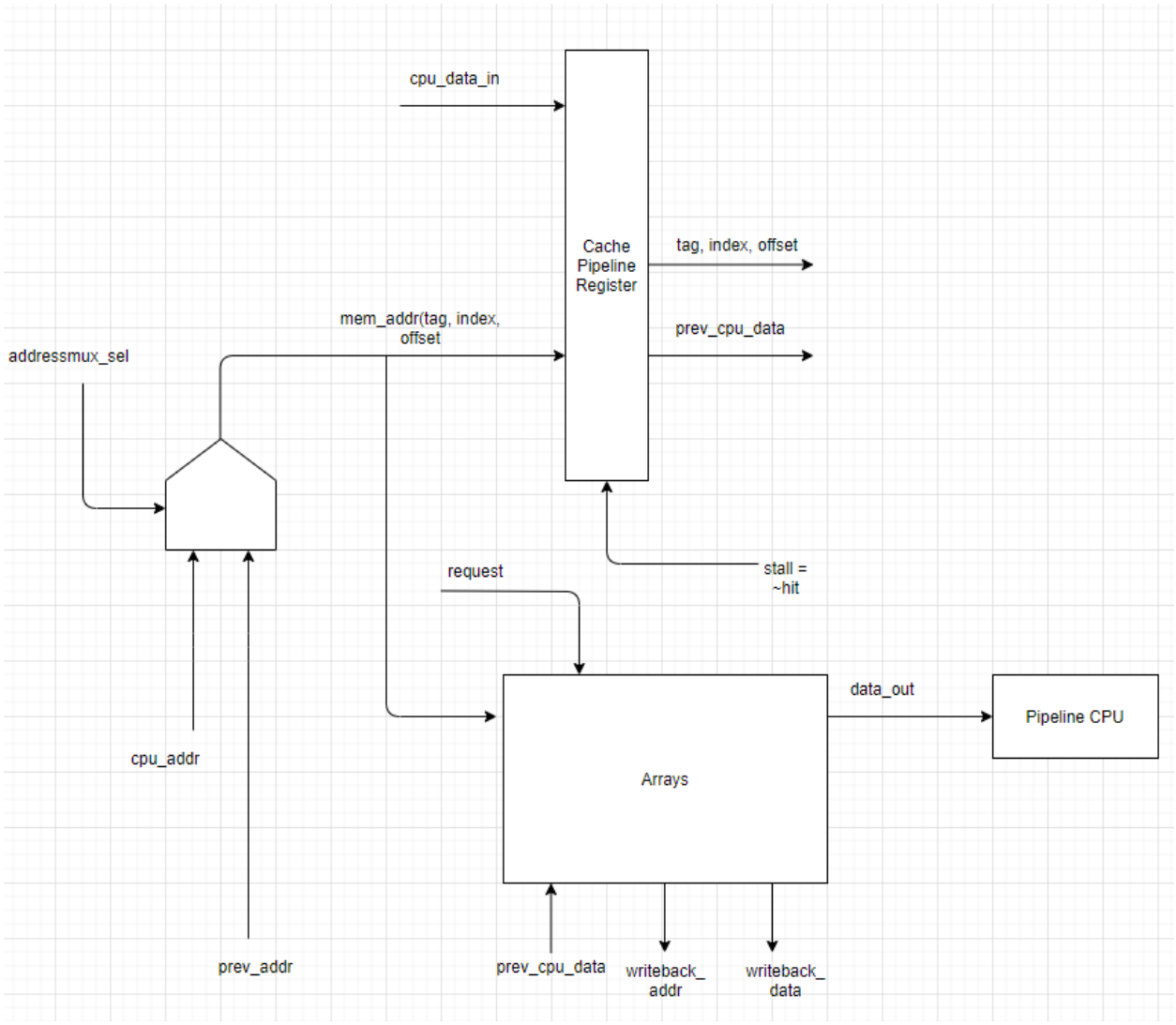



*Motivation*: The eviction write buffer is a register that is meant to hold dirty evicted blocks from L2 so that missed addresses may be sent to the L1 caches faster. For determining when to evict dirty into the temp register or write to physical memory, we used a finite state machine that had four states, Start, read, write, and stalling.

*What Went Wrong*: Issues that we encountered with the eviction write buffer was its integration with the L2 cache. If we implemented the eviction write buffer correctly, our cpu would have been able to service cache reads while performing a write to memory. This would have improved the runtime performance of our cpu by allowing our cpu to store the evicted cache line in a register and reuse the now unallocated cacheline before writing the evicted cache to memory.

The cpu can now service cache-line hits that immediately occur after the miss, which takes less time than the writing back to memory. For testing our design, we wrote some assembly code to fill up our cache. Then we modify data in our cache creating a dirty, once the evict happens we try to see if we are able to service other cache hits immediately after without performing the write back to memory. However, we noticed that memory was not up to date. A possible source for this bug, might have been the buffer was not correctly latching our write data from our L2 cache into the eviction buffer register. Another possible reason for this bug would be our write_i acknowledgement from L2 cache being incorrect, which means L2 cache could have a bug.
*Hypothesized Metrics*: Integrating a pLRU would increase the area of our design since a memory storage buffer needs to be implemented between the L2 cache and the memory in order to temporarily store data. Since the buffer can't be too big since we want very fast access times to buffer, the effect on area will not be significant. The advanced design increases the power consumption because the buffer needs to be powered and transistors inside the buffer need to be toggled periodically in order to write or read to or from it. This design boosts performance because it speeds up access to memory on a writeback by storing the data in a faster local memory and retrieving the address first. This feature shortens the combinational path that carries the dirty block data from the L2 cache to memory because it adds a register between these two memory systems which frees up time that the data needs to be moved from the L2 cache to the main memory. This shortening of the critical path between the memory and L2 cache could also improve our fmax, which would also improve performance. The tradeoffs associated with integrating the Eviction Write Buffer are that it increases performance and shortens a critical path between memory and the L2 cache, but it increases the amount of area used and the power consumption of our design and is more complex to design and implement.

**Pipelined L1 Caches:**




**Description**: Proposed datapath of the pipeline cache advanced feature. The addressmux_sel and pipeline register load would be set in the cache controller. Additional logic is needed for forwarding "data_out" to the next pipeline CPU stage.

*Motivation*:

Pipelining L1 caches allows the caches to maintain their original 2 cycle hits while also servicing 2 requests per clock cycle. The first pipelined cache stage is the request stage, when the cache will receive a request from the CPU. The second stage is when the pipeline cache determines if there is a hit or not and whether to service a read or a write request. If there is a miss in the second stage, the entire processor is stalled to service the miss. Only consecutive hits allow for a performance boost for our CPU. The implementation of pipelined L1 Caches also allows us to use BRAMs since BRAMs require that a cache is a 2 cycle hit cache.

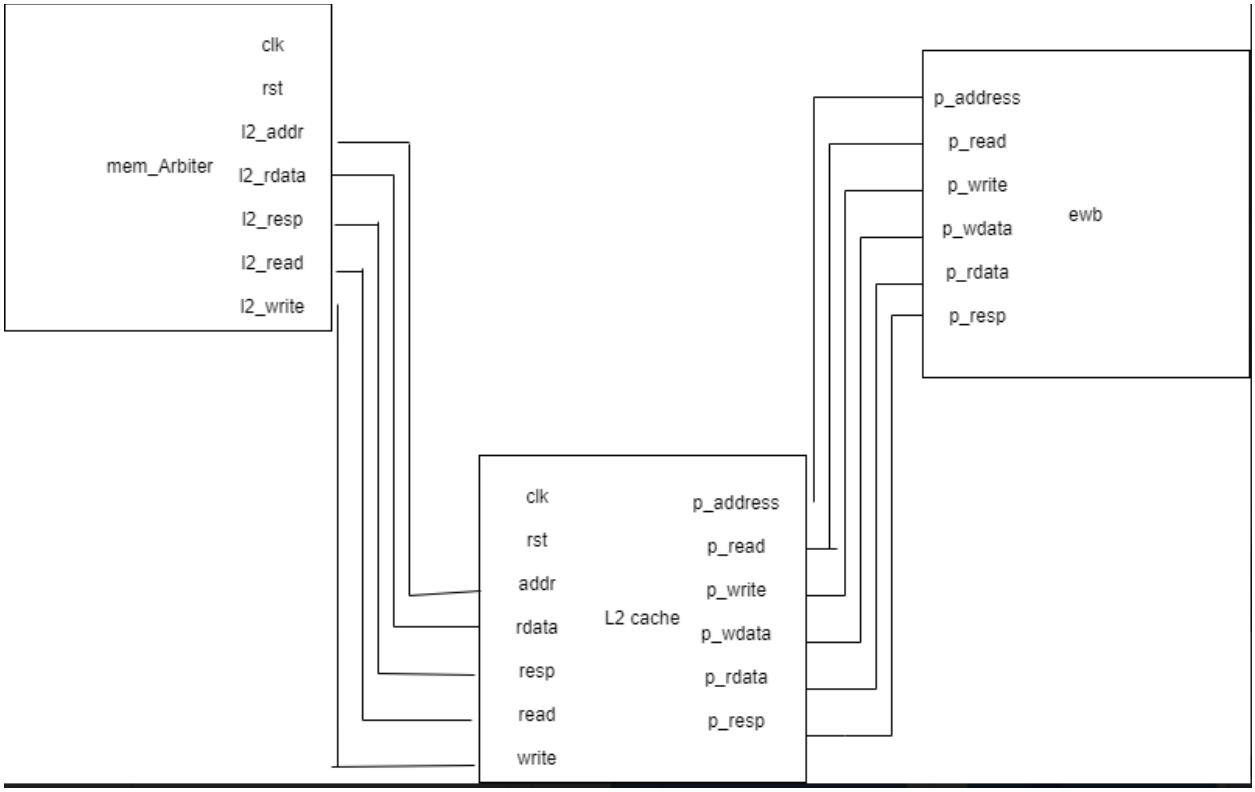
*What Went Wrong*:

Upon testing our pipelined cache design, modelsim kept outputting that we were accessing invalid memory. Upon further investigation, because of the complexity of pipelined caches, bugs in control logic, forwarding logic, and write logic were found. When debugging on the wave simulator, the pipeline cache unit didn't properly service the next request, after the first one. We suspect that our control logic next state logic was incorrect since we didn't self-transition the hit state on a hit. Also, for our control output logic, we forgot to set the newly installed signals such as addressmux_sel-- the signal responsible for selecting prev_address or CPU address, depending on the read or write signals and cache pipeline load signal. We suspect our pipeline cache forwarding logic to the next stage was incorrect since there has to be additional complex logic that decides whether the next stage should use data from the pipeline register or data directly from one of the caches.

*Hypothesized Metrics*:

Implementing the pipeline caches introduces more hardware, specifically, the cache pipeline register and additional muxes, both of which increase the amount of area needed for the caches. However, the increase in area is not a lot relative to how much area L1 cache lines take up. It also gives the opportunity to increase the size of our L1 caches without affecting the $F_{max}$ too much. Increasing the size of the L1 caches would increase the area since there would be more cachelines. The additional feature increases the power necessary to run the pipelined processor because we are adding more sources of dynamic, short-circuit, and leakage power consumption, i.e. transistors. Since it also allows for a high $F_{max}$ while increasing the size of the pipelined L1 caches, the higher $F_{max}$ would increase our dynamic power and caches with more cachelines means more transistors which increase dynamic, short-circuit, and leakage power consumption. The pros of implementing a pipeline cache are that it increases performance by allowing us to increase L1 caches without affecting $F_{max}$, which speeds up the processor since it reduces the number of requests to a higher level memory system. It also allows the processor to use BRAMs which are much faster than the data arrays given in mp3, allowing a higher $F_{max}$, which offers a speedup to the processor because clock cycles are shorter. The cons of implementing pipelined caches are the increases in power consumption, and area metrics as well introducing additional complexity to the pipeline. The additional complexity is from the interaction of the forwarding logic and the pipeline, the cache controller and setting the correct pipeline signals, as well as rerouting the cache datapath so the arrays are connected to the cache pipeline register.

**L2 Cache:**




Description High level diagram of the L2 cache.

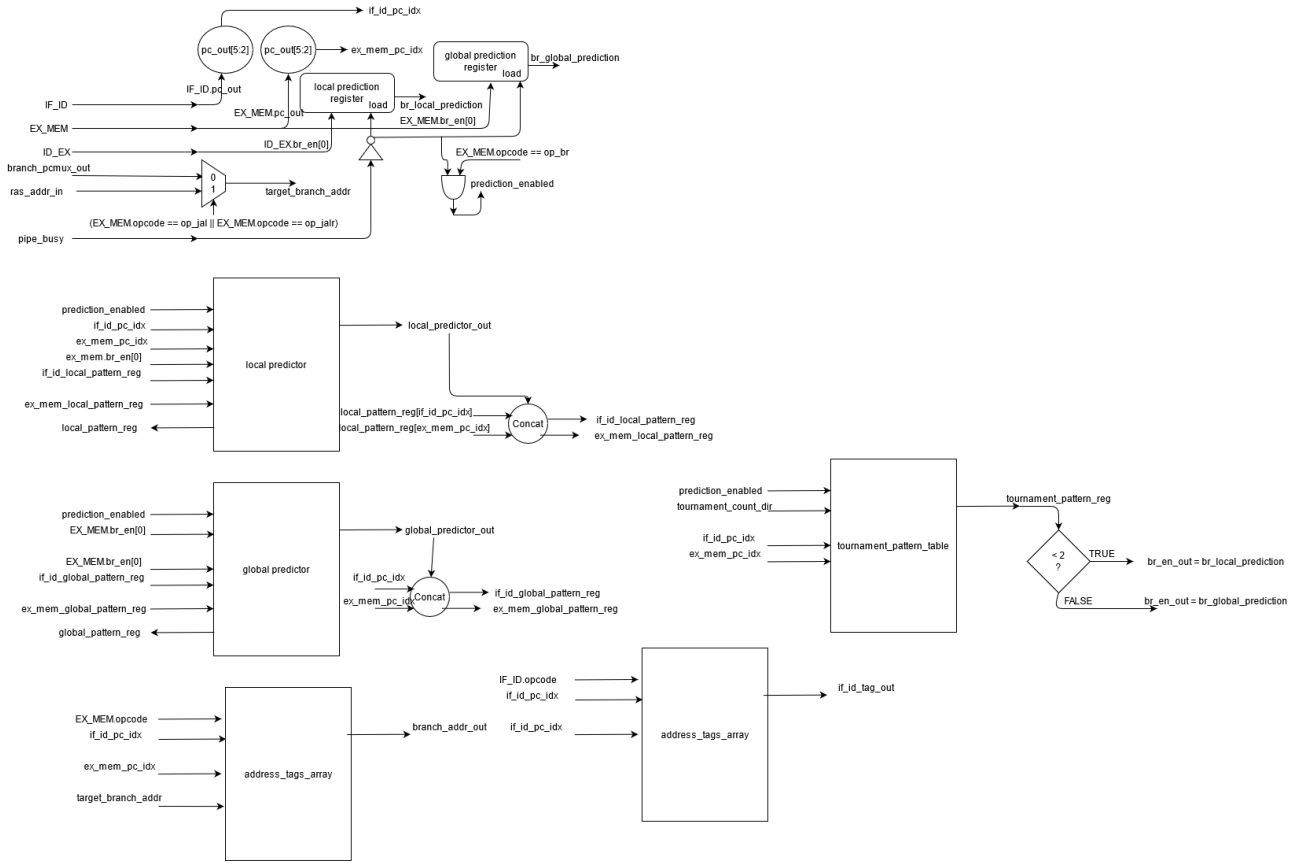
*Motivation*: Implementation of a L2 cache would reduce memory stall time while also allowing a high $F_{max}$ because blowing up the L2 cache will not form too large of a critical path.

*What went wrong*: Derek first attempted to create an L2 cache with 8-ways and a have pLRU policy for eviction. The direct implementation of this had a lot of bugs and it was hard to identify the solution. Michal decided to first integrate an MP3 cache as an L2 in order to simplify the problem and steer towards some foundation. When attempting to implement the MP3 L2 cache, we ran into memory access errors on modelsim when running the competition and checkpoint 3 code. We suspected the only change we had to make to wiring was to get rid of the Bus adaptor and rewire so that the data from the arbiter went straight to the L2 cache. It is possible that some connections were messed up without us realizing. Upon investigation in the waveform, we were able to find that the memory address sent from the arbiter was not consistent with the one sent to physical memory. We hypothesize that it was an inconsistent connection with the arbiter and cache line adaptor. If more time was allotted, we would have been able to trace the reason for the incorrect address and at least have a simple L2 cache working.

*Hypothesized Metrics*: Integrating the L2 cache into our memory memory design will increase the area, with the cachelines being the major contributor to increasing the area. Since originally, we planned to make the L2 cache be a 2-way set associative cache with a total of 8 sets and each way holding an 8-word cache line, this would make the size of the L2 cache be 512 bytes. This design of the cache was meant to simplify testing before implementing a pseudo-LRU replacement policy with 8-way associativity. A cache's power consumption is a

major contributor to the power consumption of our CPU. The L2 cache takes a lot of power through dynamic and static power consumption. Dynamic power is consumed when the cache is accessed and static power is the amount of leakage power from the cache. Because the cache lines are always on in the cache, a lot of power is wasted, and this accumulates to a significant amount of power consumption because there are so many cache lines. Static power also plays into this because for each cache line, there are numerous transistors with various voltages, all of which leak power. The L2 cache also has the potential to shorten any critical paths from the L1 caches to memory because it is not required to respond in 1 clock cycle like the L1 caches. This also allows use to increase the size of the L2 cache without affecting the fmax of our design too much. Implementing an L2 cache also boosts performance because it mitigates memory stalling by acting as a buffer for the L2 caches. The tradeoffs associated with the L2 cache is that they allow for a great performance boost, without affecting any critical paths, however, they take up more area, consume more power, and introduce more complexity due to the design.

## Tournament Branch Predictor




*Motivation:*

The Tournament Branch Predictor (TBP) advanced feature was designed to infer branch predictions early in the pipelined execution before the true outcome of the branch was known. The motivation behind adding this feature was to improve performance on correct predictions and minimize penalties resulting from flushing the pipeline of branch operations. The

implementation of the TBP uses metabits about previous execution flows to maintain two competing branch predictors, one representing the local execution context and the other representing the global execution context. The prediction made early in the pipeline is based on index bits of the PC coming from the IF_ID pipeline register and state of the tournament branch predictor unit. Using signals from the EX_MEM pipeline register, the TBP module updates the state of the local predictor or global predictor depending on which predictor correctly anticipated the branch. The next state of the tournament predictor depends on whether the predicted branch matches the actual branch known later in the pipeline. The states of the tournament, local, and global branch predictors were implemented using independent 2-bit counters. The direction of the count was a result of the correctness of the prediction. If correct, the count was incremented, else decremented. A local pattern history table and global pattern history table were maintained by both predictors as well. The index into these tables was determined by the index bits of the PC[5:2] contents from the IF_ID and EX_MEM registers concatenated with some suffix bits depending on the output of each predictor. The use of suffix bits distinguished predictions from each other by pointing to different entries in the branch history tables. Further, a Branch Target Buffer (BTB) was maintained by the TPB to function like a cache and maintains branch target addresses using the PC index bits from the IF_ID register as tags.

*Testing Methodology:*

Testing of the TBP functionality was performed primarily by tracing waveforms during the execution of small and simple programs with the inputs and outputs of the TBP connected to datapath signals. Overall, this testing methodology proved to be insufficient.

*Hypothesized Metrics*::

The hardware that affects the area metric of the TBT most significantly are the global, local, and meta branch predictors. What will take up the most space are the global and local predictors since they need to contain the target branch address and the PC address. The TBT adds to power consumption because we need to keep a buffer and a tag for all the target branch addresses of our predictor. The theoretical performance boost would be a minimized penalty on branch misprediction since they would occur less often should the tournament predictor accurately predict the branch at an earlier pipeline stage. Further, the use of the BTB would reduce the number of cycles needed to resolve a branch target address if the PC value coming in from the IF_ID register contains a tag that is currently in the BTB. Essentially, this would eliminate the cycles needed for determining the target address in the Execute pipeline stage. While the TBP would increase the area complexity and operating frequency of the design by adding more flip-flops and submodules to the datapath, the performance boost observed may have been worthwhile on programs with a considerable amount of branches

*What went wrong:*

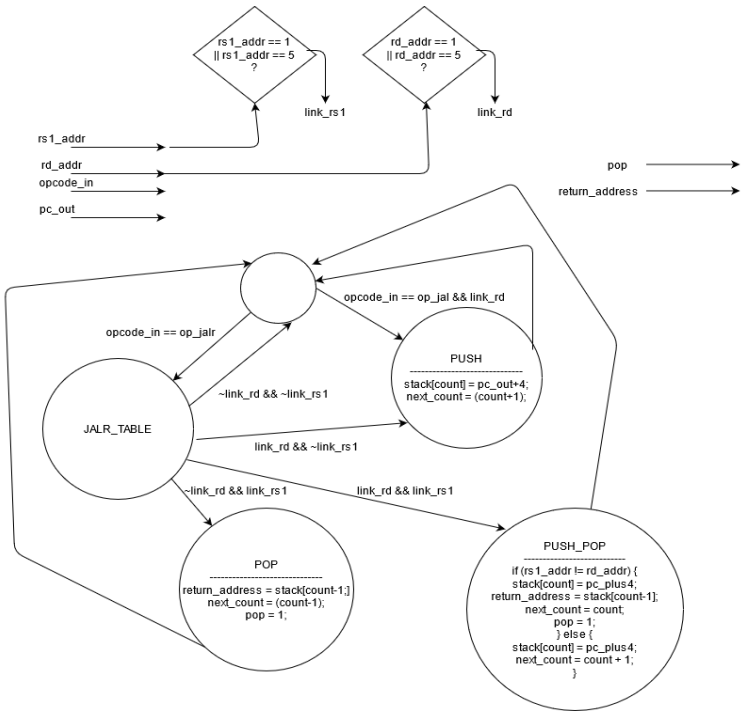
A major flaw in the TBP design was the lack of implementation logic to handle branch mispredictions. The diagram for the TBP is incomplete in this sense and would not correctly prompt the datapath to request the next instruction on mispredicts. Further, there were no mechanisms within the TBP module to modify the internal state of the different branch

predictors, both local and global, after mispredictions. This flaw manifested as a bug where the counters maintained by the TBP submodules would not change after a misprediction and the branch predictor output became a “don’t care” value. This output caused the PCMUX_OUT muxes on the datapath to select the default case and incorrectly modify the PC value. Further, the PC value associated with the misprediction should have been cached by the BTB to eliminate the need to recalculate branch target address. This additional logic would improve performance by enabling the design to use the entry corresponding to the PC value on BTB hits instead of requesting data from the L1 instruction cache since both lookups are done in parallel and the BTB hit would be faster. This would make the calculated branch target address available earlier in the pipeline.

*Additional challenges:*

The main challenge faced in verifying the functionality of the TBP was the dependency on datapath signals and pipeline registers to drive the submodules of the design. This dependency made designing a testbench after-the-fact very cumbersome since it required providing artificial stimuli to simulate the raising and lowering of various signals during the execution of a program. Had we developed a more complete test-harness or software model of the design at the start of the project, producing such stimuli and sufficiently testing the TBP as an isolated design would have been more practical and straightforward. Further, the lack of modularity in the pipeline registers and datapath made integration of the TBP challenging since it was unnecessarily complicated and confusing to rewire multiple muxes on the datapath and pipeline module connections to accommodate for the new prediction unit. Had the design been more modular, substituting the old prediction unit for the new unit should have been an easier task and required less reworking of the datapath at large to ensure these changes were reflected throughout the datapath.

## Return Address Stack




*Motivation:*

The Return Address Stack (RAS) advanced feature was motivated by the fact that unconditional branches resulting from a function call had predictable return addresses which could be retained and restored later on the function return to eliminate wasted cycles needed to prepare the pipeline and decode the next PC address. Further, this advanced feature, when coupled with the TBP, would improve performance by minimizing the number of branch mispredictions which occur following a function call. The design of the RAS included an array of 32-bit registers which would be indexed by a counter variable representing a pointer to the top of the stack. Using the semantics described by the RISC-V calling convention, when a function call is observed, the value of alu_out from the EX_MEM pipeline register is pushed onto the stack and the value of the count is incremented by one. Conversely, on a function return, the contents of the register pointed to by the top-of-stack counter become the output address of the module and the count is decremented by one, effectively popping the address from the stack. Consistency checks were introduced to obey the calling convention and disambiguate push operations from pop operations, as well as avoid hazards such as popping an address off an empty stack. In specific, the jal opcode was associated with a stack push, and depending on the values of rs1_addr and rd_addr from the EX stage, jalr opcodes were associated with either push operations or pop operations.

*Testing Methodology:*

The testing of the RAS was done by connecting the inputs and outputs of the module to the appropriate datapath signals and tracing the waveforms representing the top-of-stack counter, stack contents, and input and output address signals of the module.

*Hypothesized Metrics*::

The hardware that affects the area metric of the RAS is the stack needed to keep track of return addresses for the unconditional jumps. The RAS affects the power metric because each time there is a function call, the RAS needs to check that there is a function call and either pop or push a return address onto the stack. Because the RAS is purely combinational logic and muxes and is limited to reacting in one clock cycle due to the pipeline, it will add to our critical path, limiting the Fmax of our processor. The theoretical performance improvement of the RAS was to complement the TBP and retain the addresses to resume execution from after a function call returns. This feature should have functioned complementary to the TBP by simplifying the logic of the TBP when jal or jalr instructions are at the EX or IF stage of the pipeline and convert some mispredictions into hits when the return address is known and stored in the return address stack. The RAS would have increased the area complexity of the design and operating frequency since the overhead associated with a large stack would introduce a significant amount of flip-flops and logic gates to the design. For programs with many function calls, the return address stack would have theoretically improved performance, especially for programs with nested function calls. However, for programs with few or no function calls, the return address stack may have negatively impacted the performance by introducing additional gates to the design and critical path.

*What went wrong:*

The main challenge we faced to implement the RAS with the rest of our design was the lack of ability to efficiently verify the design, unanticipated design complexity needed to obey the RISC-V calling convention, and avoid hazards related to feeding wrong addresses to the pipeline. Initially, we thought the RAS would be easy to integrate given its simple interface relative to other design components. However, this assumption was wrong after considering our implementation of the branch predictor and pipeline stages since integrating the RAS meant rewiring numerous datapath connections and adding additional signals and muxes to account for cases where the RAS output address was incorrect. The additional select logic further increased the critical path of the datapath and added more opportunities for bugs and unnecessary complexity. Had we developed a sufficient testbench for the RAS to simulate function calls and function returns, many of the bugs related to maintaining a proper index into the stack and disambiguating whether a jalr operation should invoke a push or pop to the stack could have been worked out before attempting to integrate with the rest of the design. By testing the RAS as an isolated unit, the overall integration process could have gone smoother since RAS specific bugs would have been largely eliminated and remaining bugs would be the result of our datapath and pipeline registers.

## Reflection on Project

After the second checkpoint, design decisions that were not thoroughly peer-reviewed and verified extensively, as well as lapses in communication, became significant bottlenecks to progress moving forward and completing the objectives for checkpoints three and four. These bottlenecks manifested as bugs in the datapath, forwarding unit, and memory subsystem components of the design, specifically the arbiter unit, requiring a substantial amount of time and effort to identify and correct. Since eligibility for the competition and credit for advanced features hinged on our design's ability to correctly execute the checkpoint three code, the majority of the team effort focused on debugging. This left the team a tight window before the deadline to finish implementing the advanced features and integrating them into overall design. Reflecting on these bottlenecks, the biggest lesson we learned was the importance of designing for modularity from the start. Had we done this, many of the entries in the buglog we maintained could have been resolved via isolated unit testing before attempting to integrate them into the design. The lack of modularity in our pipeline stages and datapath made debugging cumbersome and complicated integration of some advanced features, such as the return address stack and tournament branch predictor, since integration required rewiring many connections between submodules and muxes on the datapath.

## Conclusion

Overall, the final project offered hands-on experience in experimenting with the concepts introduced in lecture, evaluating the tradeoffs between different design decisions and implementations, and coordinating individual efforts as a team to accomplish a shared goal. The objectives of this project were ambitious, but flexible enough to allow each team to develop unique pipelined processors that were optimized for certain computations relative to others. In this sense, the project was as demanding as it was rewarding since each team member's contributions and design outlook impacted the success of the final deliverable and the team's ability to achieve its initial goal. Our experience over the course of this project has weaved together the computer architecture topics introduced in this course and previous ones in an extremely thorough and comprehensive manner. This experience sharpened each person's critical thinking skills when it comes to hardware design decisions, performance, and verification. Through this experience, it became clear designing and implementing high-performance hardware is not an extremely challenging endeavor, and designs quickly become so complex to the point it is impractical to expect a single person to understand and manage its entire scope. This takeaway instilled in us the importance of approaching and evaluating hardware design in terms of modules since modules offer engineers a simplified overview for the interfaces between each hardware component and the interoperation between them. Further, modular designs encourage one to keep components very simple so they are easily combined and understood as a coherent whole. Over the course of the project, each team member improves his ability to effectively communicate the technical details of his work and designs so they are understood

straightforwardly by others on the team. With the physical limitations to Moore's law inching closer with each generation of computer hardware, this project offered us a feel for the kinds of optimizations and innovations computer architecture engineers develop to work around performance bottlenecks without sacrificing the reliability or longevity of devices. Despite the setbacks and sheer amount of work which went into the final deliverable, we will look at this experience in a positive light and apply the insights gained toward our future hardware design and team collaboration efforts.