ECE 411

Fall 2022

5-Stage RISC-V Pipeline

Calvin Lee, Mandy Liu, and Ayush Vikram
Zhuxuan

Table of Content

Introduction

Overview and Motivation

The goal of the project is to design a 5-stage pipeline processor in the RISC-V ISA. Pipelining is a technique to exploit instruction-level parallelism by allowing the processor to service multiple instructions simultaneously with the caveat that the work done for each instruction is different. Pipelining improves throughput and hardware utilization compared to multi-cycle designs. The processor supports the 32-bit, integer version of the RISC-V ISA, which is a load-store ISA.

The machine features a split L1 cache and a unified L2 cache, both of which are 4-way set associative. The refill path of the L2 cache includes a victim cache.

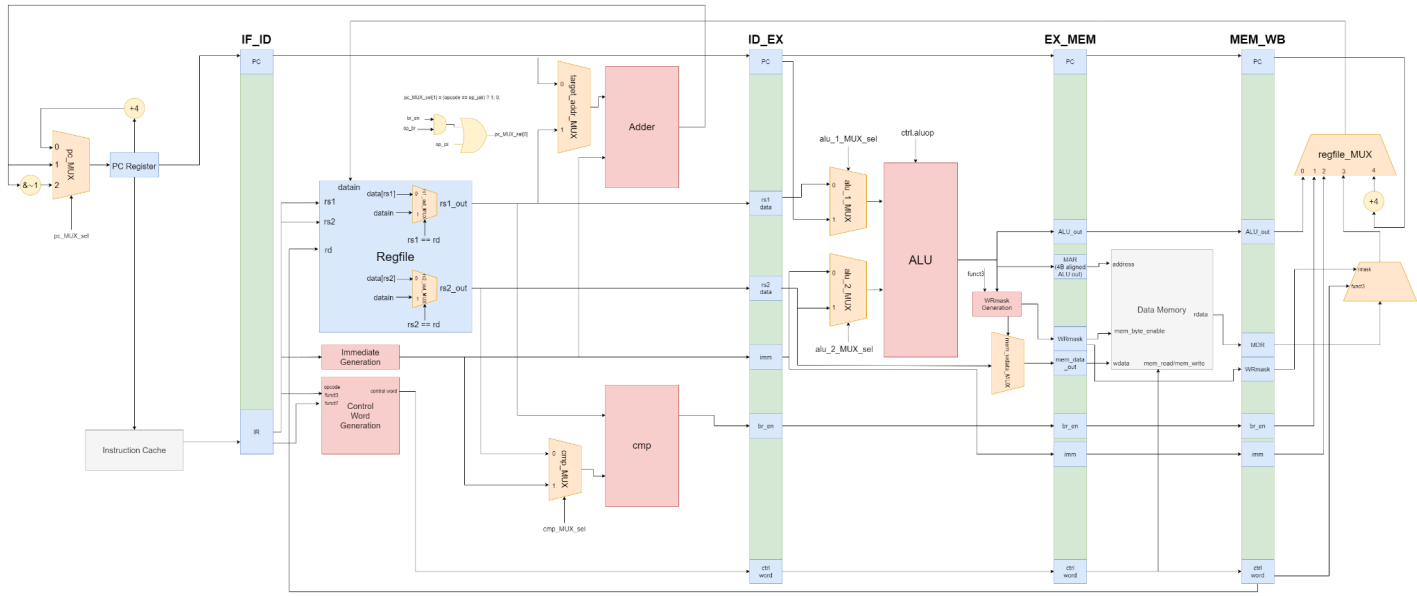
Major design consideration includes the handling of hazards. Stalling and forwarding logic is required to handle data hazards that arise when there exists RAW dependencies among instructions. The in-order nature of the pipeline eliminates WAW and WAR data hazards. Structural hazards arise when there is competition for main memory access and are handled by the arbiter.

The design includes additional features like branch predictors and memory stage leapfrogging for performance reasons.

Milestones

Checkpoint 1

Datapath



Implementation

The CP1 design is a basic 5-stage pipeline with no forwarding and hazard detection. It will only work with 4 NOPs inserted between each instruction. Magic purely-combinational memory is used without any caches.

The most important design work done in this checkpoint is the *establishment of structs*. We defined a packed struct for the control word. The four pipeline registers IF_ID, ID_EX, EX_MEM, and MEM_WB are four different modules. IF_ID pipeline register inputs a packed struct of type if_id_pipeline_reg and outputs a packed struct of the same type. The same applies to the other three pipeline registers. The control word packed struct is a member in the packed structs for pipeline registers. This provides automatic scoping that proved very valuable for the remainder checkpoint. `Id_ex_in.ctrl.opcode` is the opcode of the instruction that is in the decode stage. `Id_ex_out.ctrl.opcode` is the opcode of the instruction that is in the execute stage.

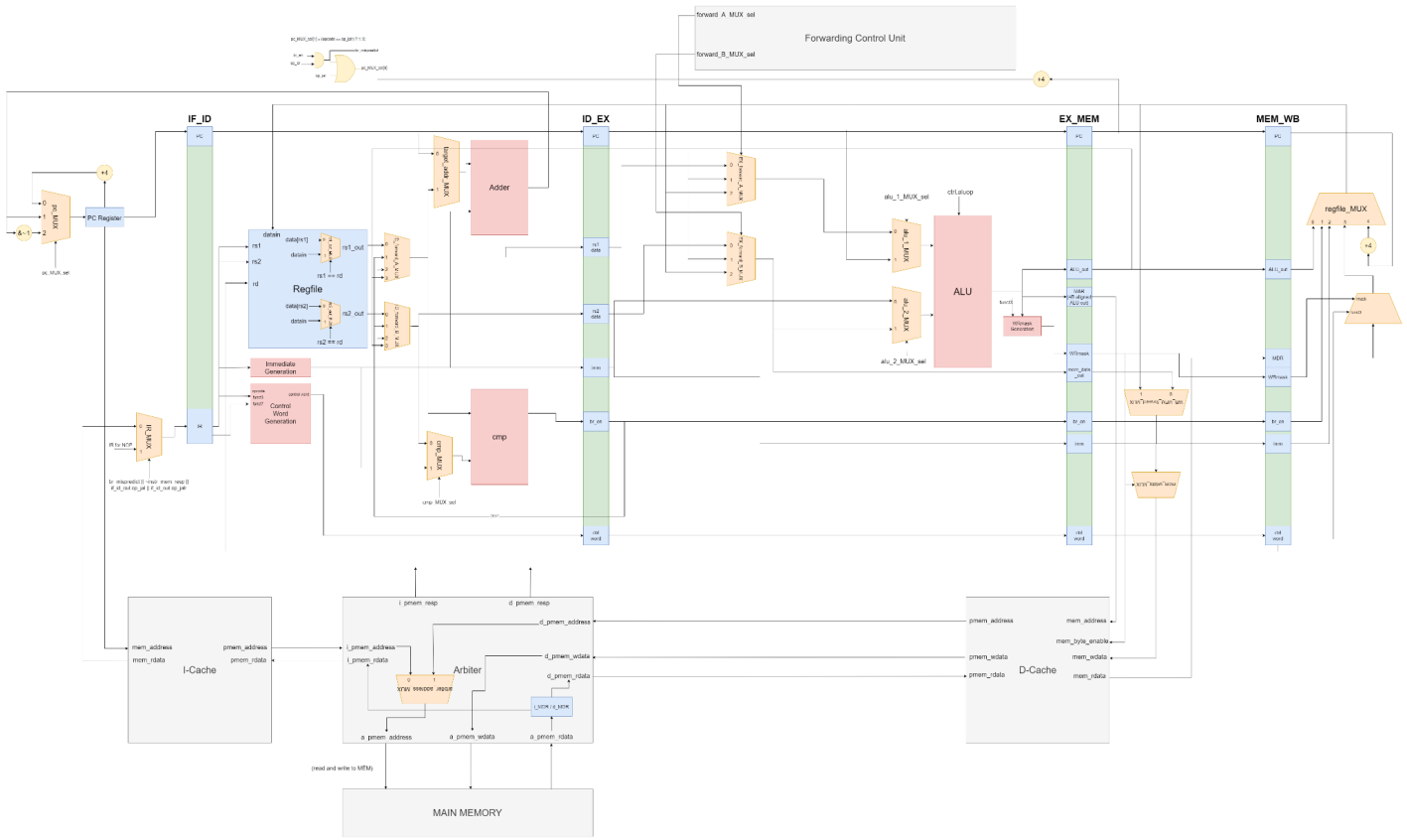
To reduce branch misprediction penalty, we instantiated the comparator in the ID stage. This means BR/SLT will be resolved in the ID stage. The tradeoff for this is that we will essentially double the number of forwarding paths in the next CP. Since BR/JAL/JALR requires calculation of the target address, we instantiated an address adder in the ID stage as well.

Testing

Testing is done by running mp4-cp1.s and examining the entire register state and PC.

Checkpoint 2

Datapath



Implementation

The major design work for this checkpoint is to develop the stalling and forwarding unit. The forward paths are WB to MEM, WB to EX, MEM to EX, MEM to ID, EX to ID. Note that forwarding is never necessary. Only dependency detection and stalling are required to guarantee correct execution. *Forwarding is a performance feature.* The two questions to answer when forwarding is (1) which stage to forward from (2) what to forward. With regard to (1), if we can forward from both MEM and WB to EX, then we forward from MEM since that instruction is the most recent writer. With regard to (2), we need to check the instruction type: a dependency on LUI will require forwarding of immediate value while a dependency on SLT will require forwarding of comparator output.

The general approach is to check for dependency and insert NOPs between the producer and consumer until a forwarding path becomes available. Suppose we need to insert a NOP into the EX stage at the next cycle, then we stop loading the architectural PC and IF_ID pipeline register and flush the control word of the ID_EX pipeline register.

For forwarding to the ID stage, we have two MUXes to replace the value of rs1 and rs2 read from the register file. Instructions that use the comparator or address adder for branch condition comparison, SLT comparison, or jump address calculation need to be forwarded at this point. For forwarding to the EX stage, we have two MUXes to replace the value of rs1 and rs2 read from the ID_EX pipeline register. Instructions that use the ALU for memory address calculation, addition, subtraction, and logical operation need to be forwarded at this point. For forwarding to the MEM stage, we have a MUX to replace the value of mem_wdata read from the EX_MEM pipeline register. This is for the particular case of ST after LD where the value written to memory is the value just read from memory (most likely at a different address).

For structural organization purposes, we have two main control units. Forwarding Control Unit (FCU) outputs the MUX selection signals for the 5 forwarding MUXes mentioned above. Stalling Control Unit (SCU) outputs the load and flush signals for the architectural PC register and the 4 pipeline registers.

In addition to data dependencies, we also stall the entire pipeline whenever we have a miss in the I-cache or D-cache (this is the handling strategy for the given cache, not pipelined cache).

In the case of a branch mispredict, we do the following in the same cycle: (1) set PC_MUX to load PC with the output of the address adder, which contains the target address (2) insert a NOP into the IF_ID pipeline register (3) let the backend of the pipeline continue as normal.

Since I-cache and D-cache can both miss in the same cycle and compete for main memory access, an arbiter is required to prevent Structural Hazard. The arbiter policy is simple: only process I-cache requests when there are no D-cache requests. The arbiter only routes the addresses and memory response signals. It does not have an internal buffer.

Testing

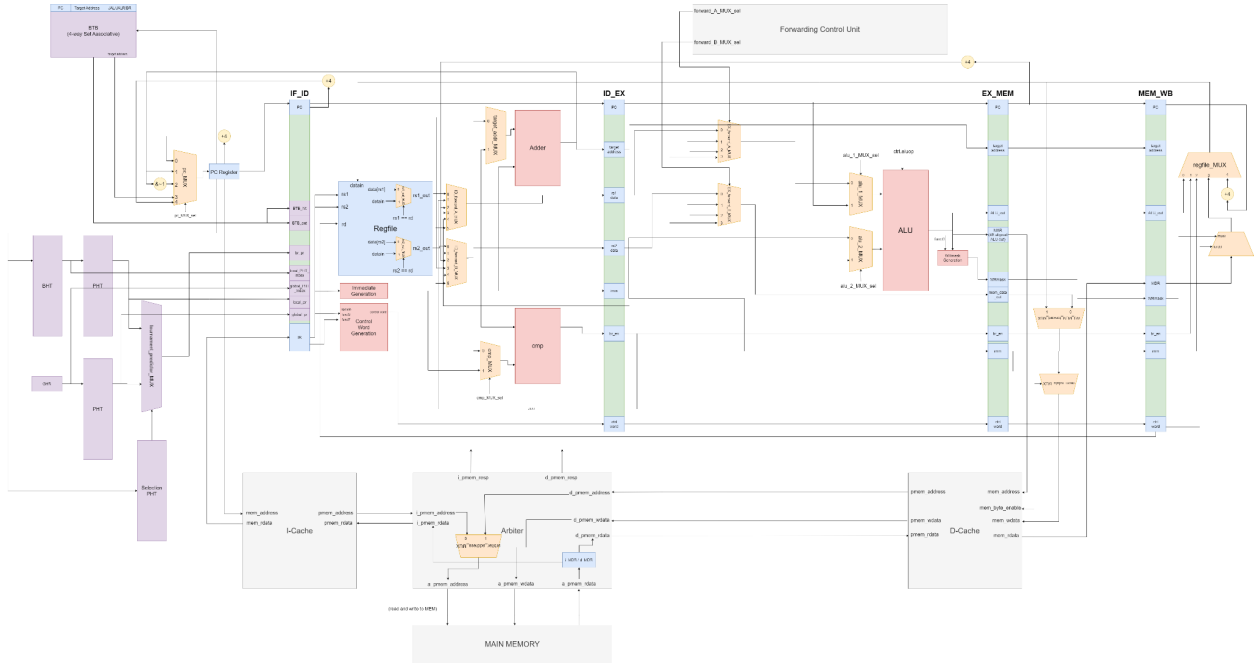
Testing is done by running mp4-cp2.s and the three competition codes and examining the entire register state and PC. The RVFI monitor and shadow memory are connected. We also made sure the design synthesizes.

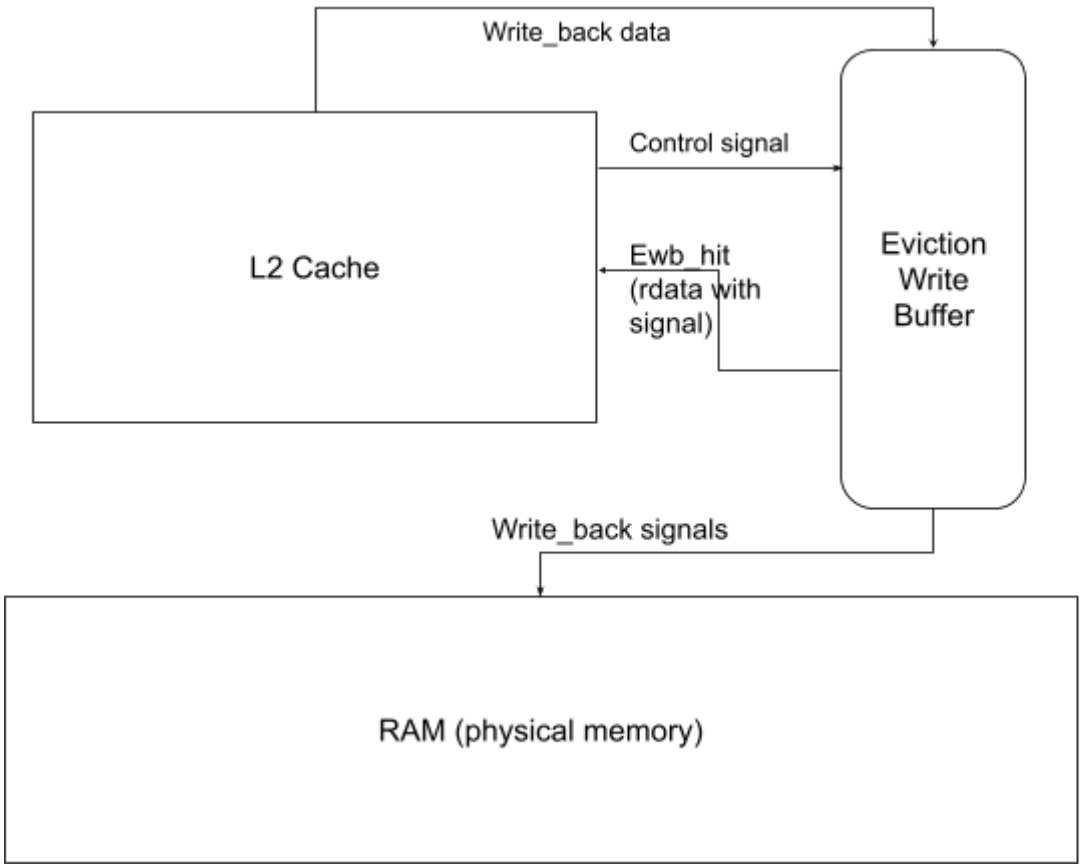
Checkpoint 3

See the Section Advanced Features.

Checkpoint 4

Datapath





Implementation

Since we already have some extra advanced features implemented from checkpoint 3, such as eviction write buffer and memory stage leapfrogging. Before this checkpoint, since our memory stage leapfrogging only passes the *mp4-cp3.s* test, we need to debug for all three competition codes.

Testing

We tested our performance by running the four test codes, mp4-cp3.s, comp1.s, comp2.s and comp3.s and for each test code. According to the website, *for each test code, your processor will be assigned a score calculated as* $PD^2 * (100/Fmax)^2$, *or* $energy * (delay * 100/Fmax)^2$. Therefore, we will test our processor by running synthesis on each four test codes, and calculate the Fmax, area, and energy to calculate the assigned score. Then parameterize the cache if necessary to improve on the performance.

Advanced Features

Branch Predictor

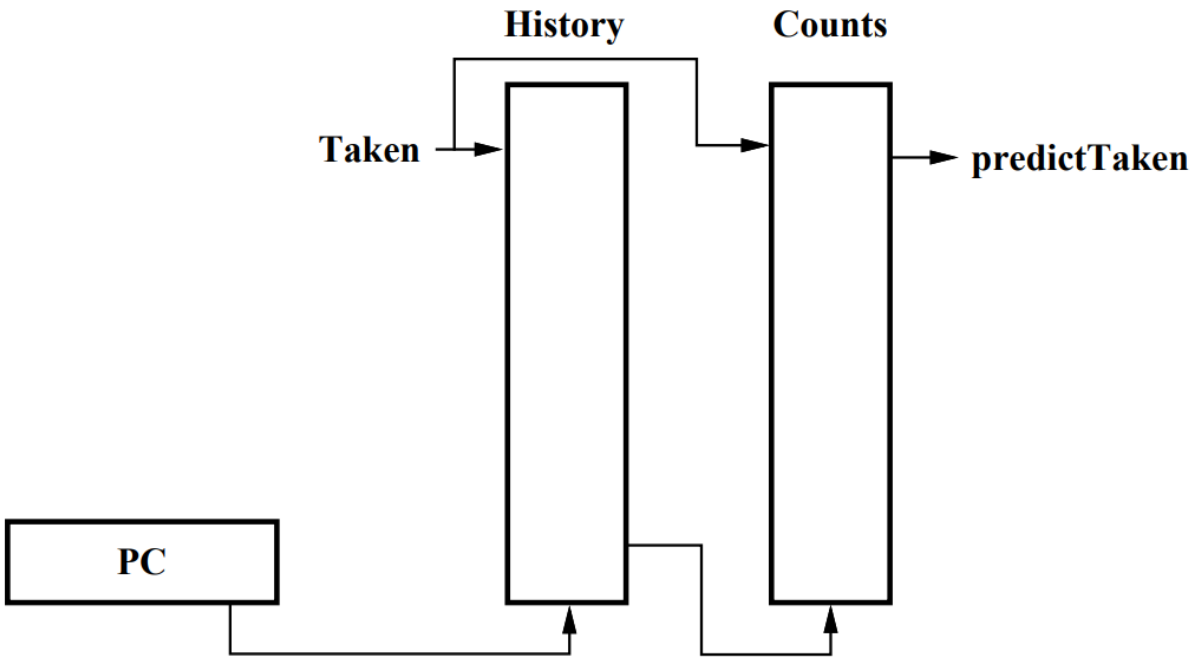
Overview

In our design, branch, JAL, JALR instructions are resolved in the ID stage. A penalty of 1 cycle is incurred when (1) branch prediction is incorrect (2) target address is unknown (3) in the case of JALR, target address is incorrect. For performance reasons, we proposed a branch prediction scheme that exploits global and local correlation between branches. A branch target buffer (BTB) is used to (1) identify if an instruction is a branch, JAL, JALR (2) identity target address.

The correct execution of the pipeline is independent from branch prediction accuracy. All predictor subcomponents are read in IF and written in ID.

Branch predictor design *heavily inspired* by this document. The figures in this section are taken from this document as well.[1]

Local Predictor



The lower bits of the PC are used to index into the Branch History Table (BHT), which is an array of shift registers. Note the lowest 2 bits of PC are not used since instructions are 4B aligned. Each BHT entry, N-bits, stores the direction taken by the N most recent dynamic instances of the branch at the given PC. The history is used to index Prediction History Table

[1] https://www.cs.utah.edu/~rajeev/cs7810/papers/mcfarling93.pdf

(PHT), which is an array of 3-bit, both-sides-saturating counters. The MSB of the counter is the prediction.

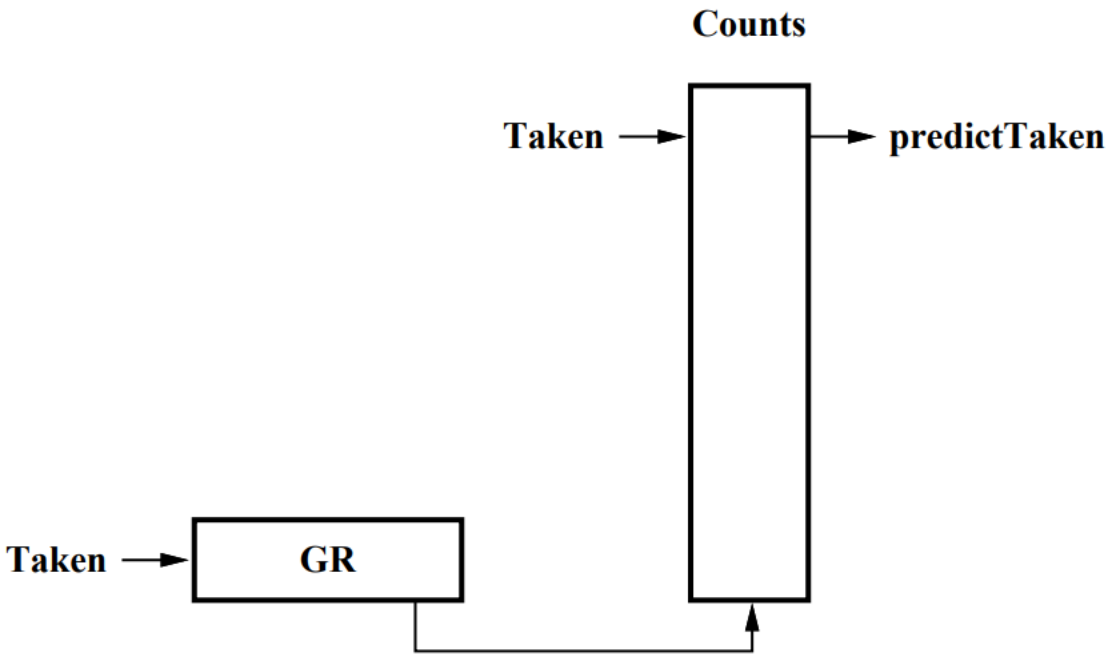
To update the predictor in ID, we store into the IF_ID pipeline register the PHT index corresponding to the PHT entry that was read in IF. Upon resolution of the branch, the counter is incremented or decremented if the branch is taken or not taken, respectively.

Consider the following code.

```
for (i=1; i<=3; i++){}
```

The dynamic pattern is TTN TTN TTN TTN. Assuming no aliasing in the current working set, a trained local predictor with a history depth of 2 will have 100% prediction accuracy. This is a local correlation.

Global Predictor



Global History Register (GHR) stores the direction taken by the N most recent dynamic instances of branches at *any* PC values. The GHR content is used to index a PHT, identical to the PHT in the local predictor.

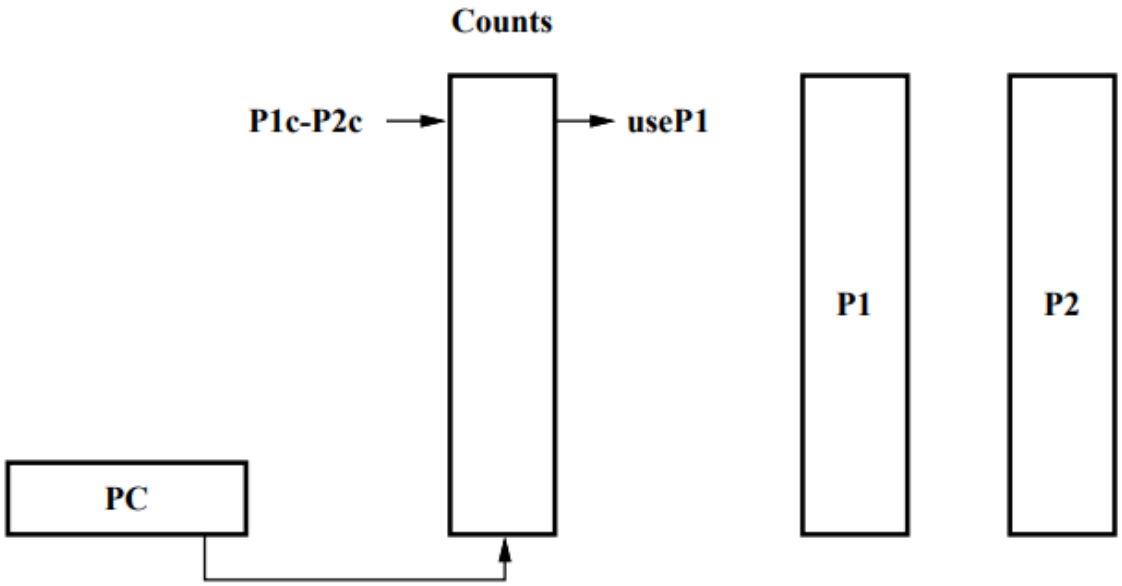
Similarly, to update the predictor in ID, we store into the IF_ID pipeline register the PHT index corresponding to the PHT entry that was read in IF.

Consider the following code.

```
if(predicate_A) {}
if(predicate_B) {}
if(predicate_A && predicate_B) {}
```

The direction of the third branch is logically determined by the first two. This is global correlation.

Tournament Predictor



Depending on target applications, there might be stronger local or global correlation. A promising strategy is to train both local and global predictors and use an additional Tournament PHT as the selection function. The Tournament PHT is indexed by lower bits of the PC and its structural organization is identical to PHTs in previous contexts. The MSB of the Tournament PHT is used to make binary decisions between local and global predictors.

To update the Tournament PHT, we store the prediction of both predictors into the IF_ID pipeline register. In ID, if local and global predictors have consensus, no update is done. If the predictions differ (one of the two must be correct), we increment or decrement the PHT based on whether the local or global predictor produced the correct prediction, respectively.

Branch Target Buffer

Even with a branch predictor with 100% accuracy, the 1-cycle penalty is not avoided unless (1) we know the instruction in IF is BR/JAL/JALR (2) we know the target address. Both of these are provided by BTB. The BTB is organized as a 4-way, 8-set, p-LRU cache with parts of the PC as

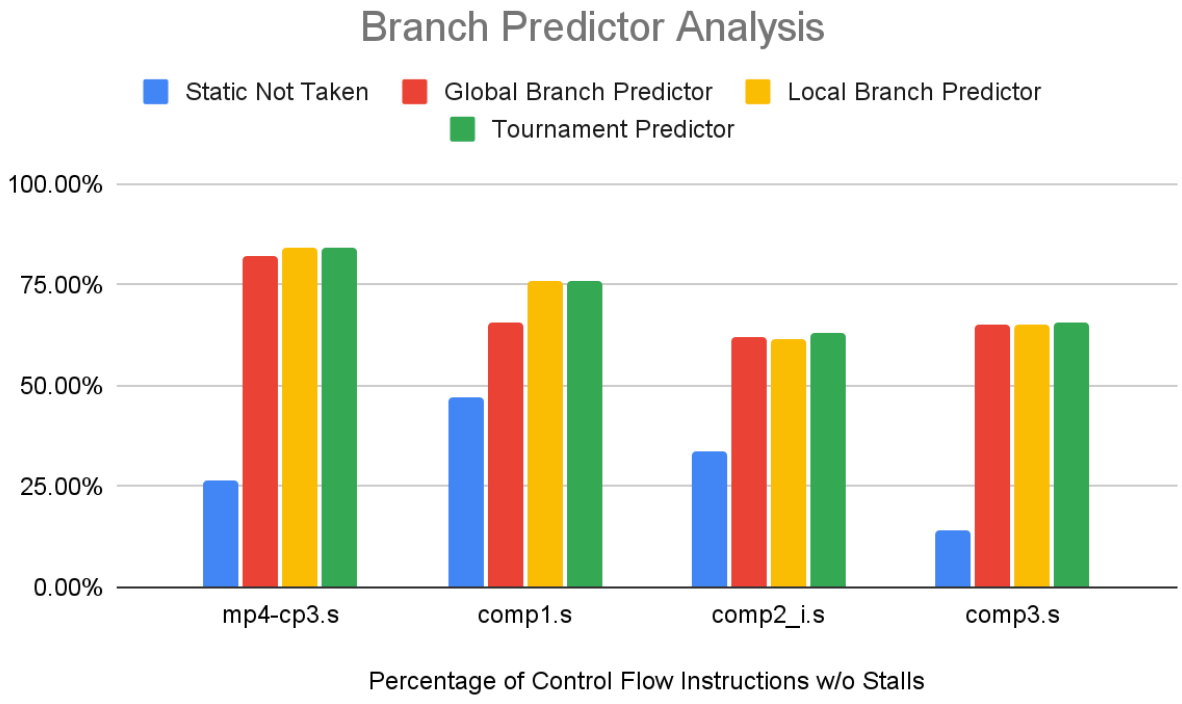
tag. Each cache line stores the 32-bit target addresses and a 2-bit value differentiating BR/JAL/JALR.

In the IF stage, we check if the current PC is a hit in the BTB. If BTB is a hit and the instruction is BR, then we will use the branch predictor to determine if we should fetch PC+4 or target address at the next cycle. If BTB is a hit and the instruction is JAL/JALR, then we fetch the target address at the next cycle. *If BTB is a miss, we always fetch PC+4.* Note that for BTB miss, the instruction can still be a BR/JAL/JALR since it can be a first encounter of said instruction or the instruction has been evicted from BTB due to capacity/conflict miss.

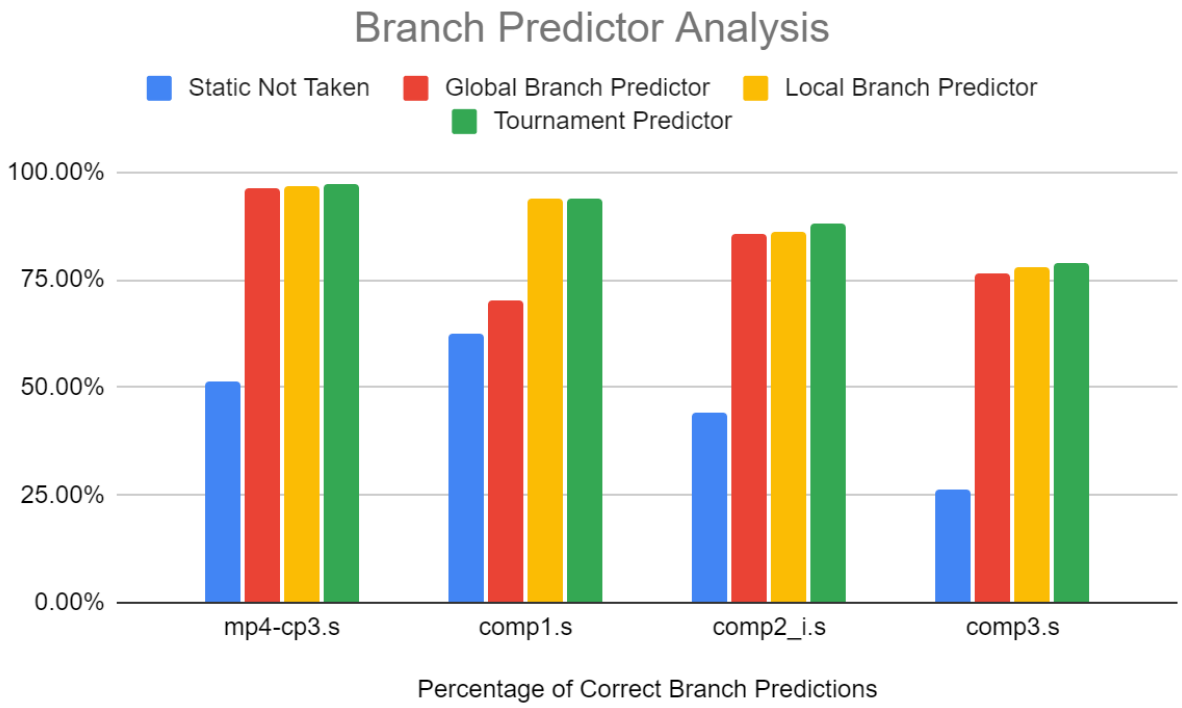
In the ID stage, there are several scenarios where we need to insert a NOP into IF_ID pipeline register and pay 1 cycle penalty (1) it is a BTB hit with a branch but the branch prediction is incorrect (2) it is a BTB hit with a JALR but the target address is incorrect (JAL's target address never change, but JALR's target address do change) (3) it is a BTB miss and the instruction turns out to be a JAL/JALR/branch that is taken.

If we have a BTB miss in IF and the instruction turns out to be BR/JAL/JALR in ID, a new line is allocated in the BTB. JALR's target address is also updated in the BTB if the currently stored target address is incorrect.

Figures & Analysis



This figure presents the percentage of control flow instructions that do not incur a penalty. This requires a hit in the BTB, a correct branch prediction in the case of BR, and a correct target address in the case of JALR.



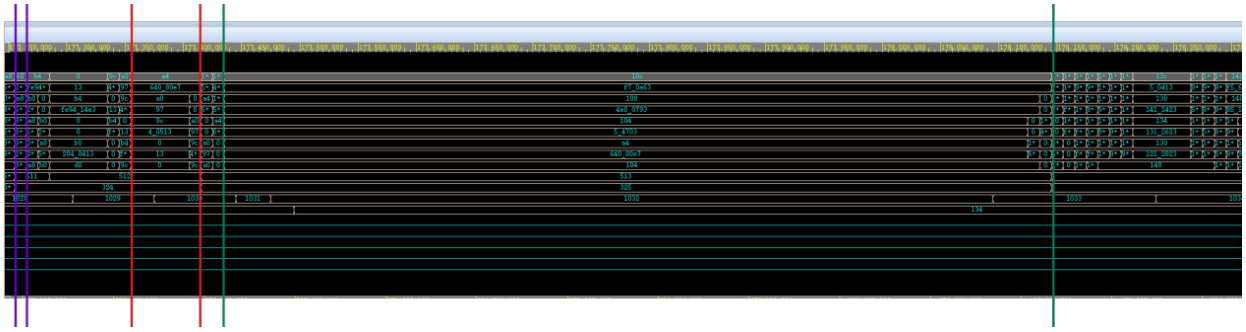
This figure presents the percentage of branches that are predicted correctly by the branch predictor.

We can make the following observations.

1. Branch predictors do work. They offer vastly improved accuracy compared to static not taken predictors.
2. Global branch predictors do not work well in competition 1, most likely because one of its kernels specifically tests uncorrelated branches. It has more globally uncorrelated branches than the other test codes. It is globally uncorrelated because each one of its if-statements check different bit positions.
3. We see that the BTB is the limiting factor when it comes to avoiding the 1-cycle penalty. Competition 1 has a 94% prediction accuracy but only 76% of control flow instructions do not incur penalty. This means a lot of times the branch predictor is correct, but since BTB fails to provide the target address, the prediction result is not used.
4. In general, the tournament predictor outperforms both local and global predictors.
5. We observe significant variation in branch predictor accuracy across workloads due to the biasness of branches.

Note on Performance Impact

Although we see large improvement in branch predictor accuracy, it should be noted that branch predictors do not make a large impact at the system level *in the context of our design*.



In this picture, the purple interval is 1 clock cycle, which is what we save with branch prediction. The red interval is L1 miss. The green interval is L2 miss. Having the entire code and data segment in the cache will make significantly more improvement to performance than branch predictor.

Cache

Pipelined L1 Instruction Cache

The reason why we want to implement pipelined L1 Instruction cache is because it could maintain a short critical path like the cache from our MP3, and we would achieve full utilization of our pipelined I-Cache since instruction read is performed every cycle. Since directly changing a cache from two-cycles hit to one-cycle hit will increase the number of things done in one cycle, such as tag-checking, the critical path for our processor using the provided cache is slow. We think using a pipelined L1 Instruction cache could enhance our performance, by allowing us to break-apart the cache operation into 2 stages.

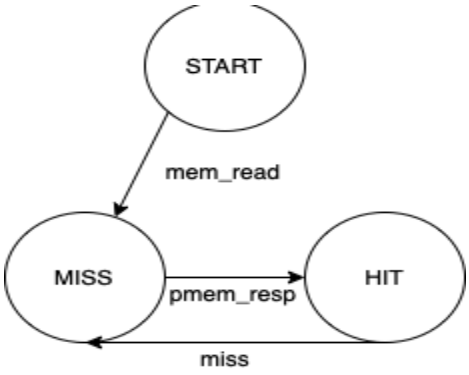
We used the data_array and cache array modules from MP3, which were based on SRAM, and so had a sequential read and write operation. Thus, in the first stage of the our pipelined I cache, we simply let in the new address/request and let it begin to read all of the arrays, so that by the next cycle, in the second stage, we would have an updated read from our arrays (remember that they were sequential read), and so in the second stage of the pipeline, we could return the requested instruction data (setting mem_resp high), and *at the same time* as returning the requested data for the request in the second stage, we allow a new request to enter and begin reading from our sequential read arrays.

To summarize:
1st stage of Pipelined I-cache:

- Letting current request read from sequential arrays so that by second stage, we will have updated read values from array for this request

2nd stage of Pipelined I-cache:

- Current request, which began reading in first stage from sequential read-arrays, has finished reading by second stage, and so we can deliver the requested data for this request and set mem_resp = 1 if there is a hit, and if a miss, return to the miss stage and obtain the data from the lower levels of the memory hierarchy and then return requested data to CPU.
- At the same time as the first bullet_point, allow a new memory request/address to begin reading from the cache arrays. This request is in its 1st stage.



State Machine of Pipelined I-Cache

We kept a pipeline register that we passed between stages of our pipelined cache. Since all of our reads were sequential, our pipeline register only needed to keep track of the cpu address, so that when we were handling a new memory request in the second stage of the pipeline, we still had access to the address of the previous request, so that we could, for example, provide the correct address to the bus adaptor as we sent the data back to the CPU.

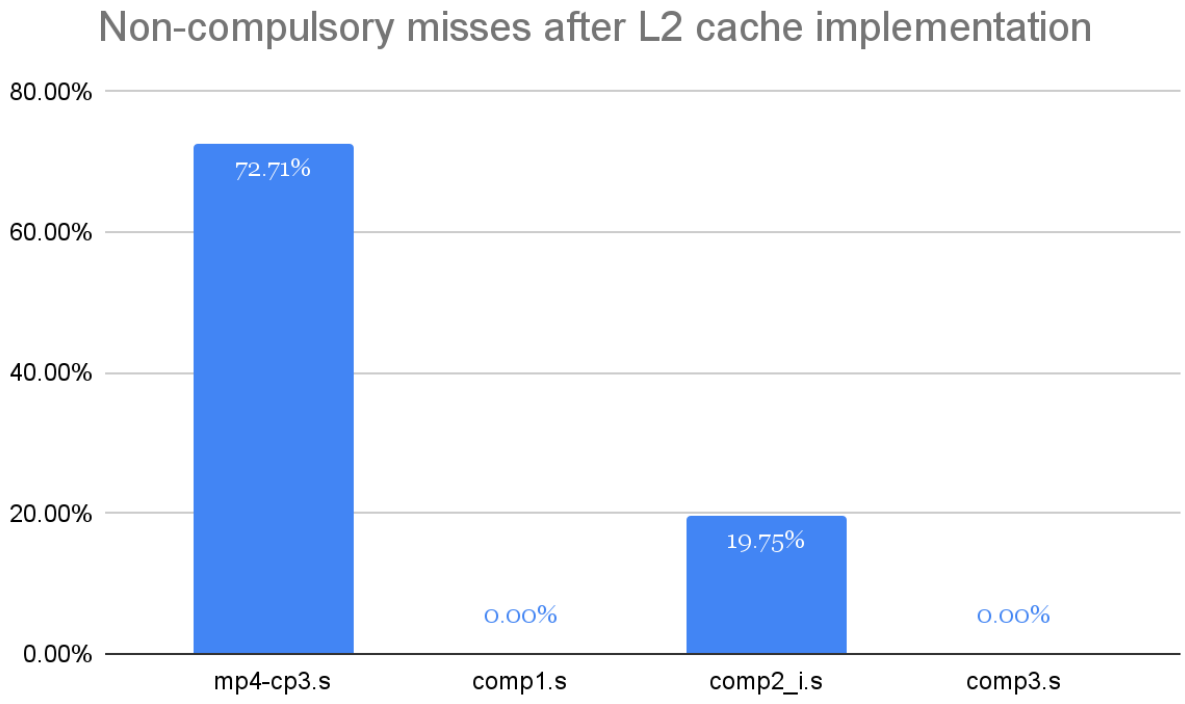
We also had to stall our pipelined I-cache, from addressing a new memory request, when our processor had a stall in its pipeline, which would stop our IR from being loaded, so that any data read by our cache was not lost. There were very few downsides for implementing a pipelined I-cache. We were able to improve our critical path timing by splitting up the cache operation into 2 stages. We decided not to implement a pipelined D-cache, since the pipelined D-cache would not be fully utilized, unless there were consecutive data accesses, which is unlikely. On the other hand, the pipelined I-cache would be fully utilized since every cycle would require an I-cache read. Indeed, using the pipelined I-cache resulted in a ~30% speedup on the CP3 code, and resulted in modest speedups on all the competition codes. We believe this improvement was due to the fact that our pipelined I-cache was a set associative cache, and was bigger than the provided direct-mapped I-cache that we were using before this pipelined I-cache, and so our instruction data retrieval time would be faster. Thus, using our pipelined I-cache instead of the

provided direct-mapped I-cache, we would expect to see speedups on any workflows, provided that workflow is long enough, with jumps and branches, so that our bigger I cache would speed up retrieval times.

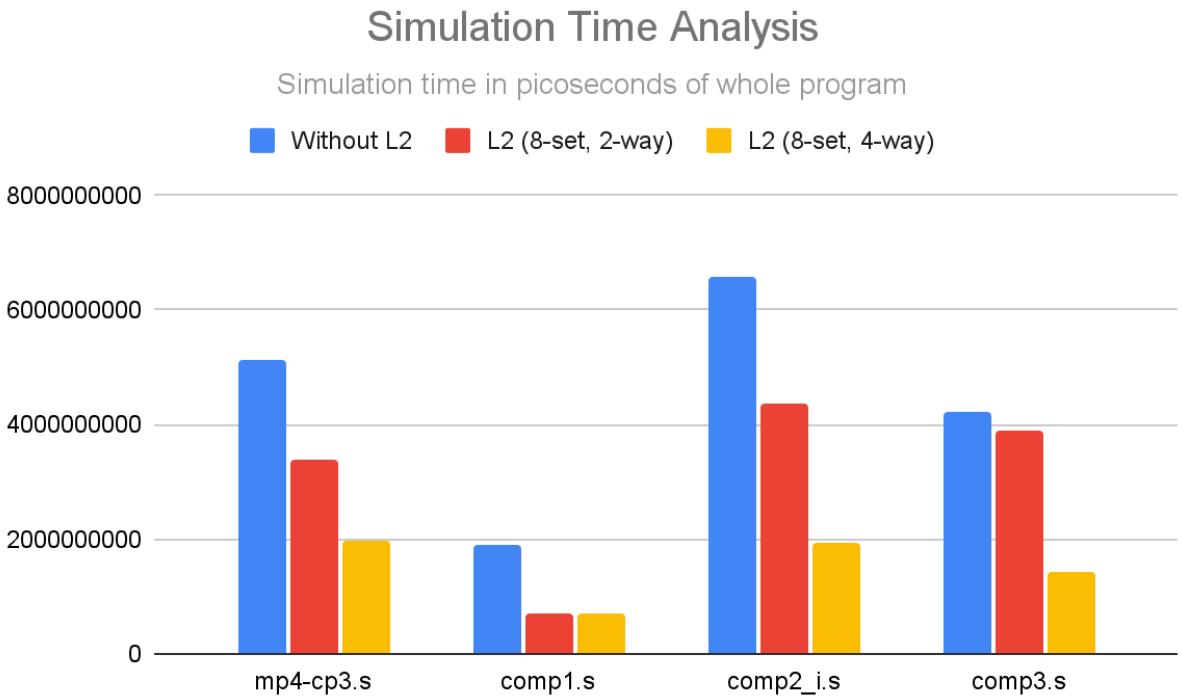
L2 Cache

One of the reasons we want to implement L2 cache is because: *Having additional caches can greatly speed up your design by keeping your Fmax high while also mitigating the effects of memory stalling.*(411 release instruction) Since each miss in L1 cache requires us to communicate with the physical memory, and the response from RAM usually takes a few cycles, therefore, we want to eliminate the number of physical requests to improve performance. Adding L2 cache can significantly reduce the memory request time on average because bigger cache could store more physical memory. Therefore non-compulsory misses which are data misses that the system have encountered before can be eliminated by reducing the percentage of evicted data from cache. Increasing the memory hierarchy could decrease the non-compulsory miss. Intuitively, adding one more layer of cache could decrease the memory access time when you can store more data in the processor. But the tradeoff is the increase of area and the limitation of such memory hierarchy is that it could help a lot on performance when the program is large enough that there are lots of non-compulsory misses.

For implementation, we add a L2 cache between our arbiter for L1 cache and pipeline adaptor for physical memory. Since the arbiter already takes care of the control flow of memory requests from both I-cache and D-cache, we don't need to worry about whether it is instruction read or data read. On the other hand, we would need the write_back stage since there could be a write request. The logic of L2 cache is almost similar to our MP3 cache which could respond to a hit request in two cycles partly because it is not directly wired to pipelined CPU which requires one cycle hit to facilitate. Another thing to be noted is that L2 cache is united, which means when fetching instruction requests, it might also suffice the request from data request and doesn't need to fetch from physical memory again.



Before adding any performance counter, we first count the number of non-compulsory misses for each test code. We found that there are relatively large percent of compulsory misses compared to total misses. This means there are more than half of total misses we can improve to eliminate. The exception is the comp1 test. Here you can see there are no non-compulsory misses and all are misses that first occurred.



We test the improvement on the whole program after adding L2 cache by looking at the simulation time. As we can see from the graph below, adding the L2 cache could reduce the total simulation time, while increasing the size of the L2 cache could also do the same effect. Another interesting point is that comp1 code doesn't have any non-compulsory misses and the graph below proves this because the simulation time for 2-way L2 cache and 4-way L2 cache is exactly the same.

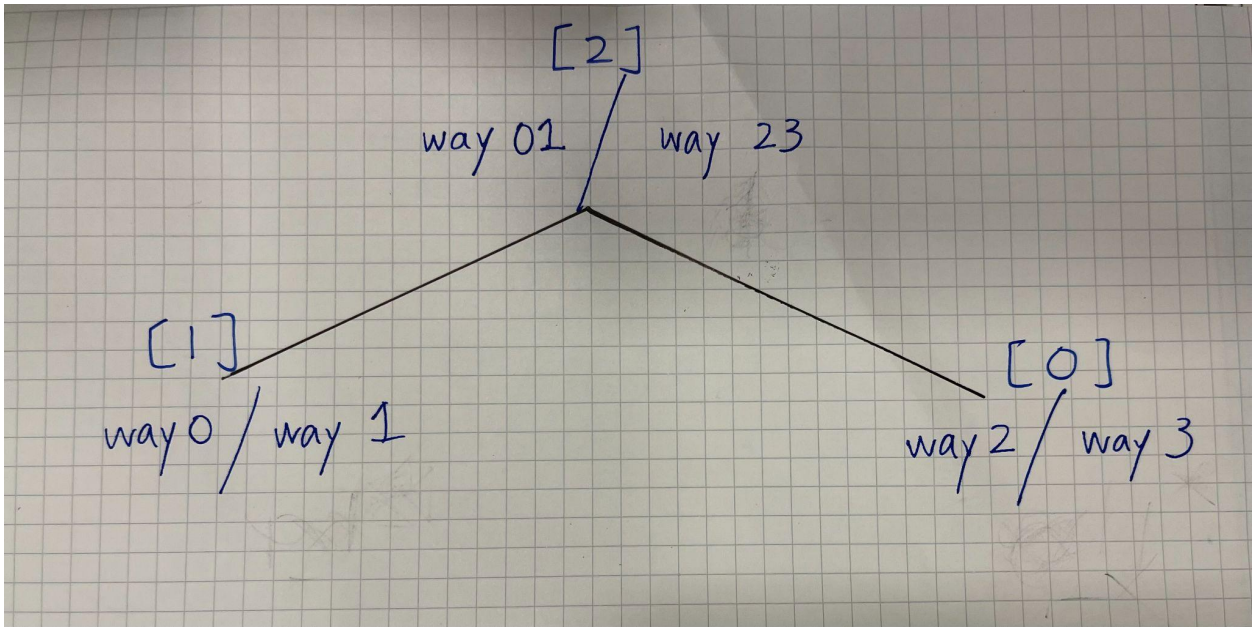
4-way Set Associative Cache

Initially, we are provided with a simple direct-mapped cache with 8 sets and each line contains 32 words. As we all know, direct-mapped cache has the worst utilization of memory since if two different addresses are aligned to the same set, the old one has to be evicted from the cache even though there might be other unoccupied space in the cache. Additionally, the number of sets is not large enough that the previous situation might happen very often. Therefore, we want to change our cache to a 4-way set associative cache and with 8 sets in default. This also includes adding the replacement policy, p-LRU to determine the pseudo-least recently used way for a new address if all four ways are occupied.

The reason why this implementation would improve our performance is the situation when one address needs to be stored in the cache while another address with the same set bits is present. With direct-mapped cache, we have no choice but to evict the old one. But with our new cache, we can keep all two addresses in the cache as they will occupy two out of four ways in that set. However, the limitation of such cache is the number of MUXes and additional arrays used for

replacement policy. Moreover, the time it takes to request the memory will increase as the program proceeds, and more ways in the cache are occupied.

The pseudo-LRU replacement policy can be structured as follows.



We can see from the tree that p-LRU needs 3 bits to store information. The most significant bit stores whether the most recently used way is in the former two or the latter two ways. Then, if the middle bit is 0, it means that way 0 is most recently used. Similar logic applies to the least significant bit of LRU. Before implementing the cache, we have several if statements for the p-LRU logic. Part of the logic when all ways are valid/occupied and we need to utilize one of the ways are shown below.

```
if(LRU_array_dataout[2] == 1'b0)
    begin
        if(LRU_array_dataout[0] == 1'b0)
        begin
            // Alloc way 3
        end
        else
        begin
            // Alloc way 2
        end
    end
    else
    begin
        if(LRU_array_dataout[1] == 1'b0)
        begin
```

```
            // Alloc way 1
        end
        else
        begin
            // Alloc way 0
        end
    end
```

One of the tradeoffs for 4-way set associative is the increase of storage of the processor. The good part is increasing the size of cache leads to increasing the number of data stored in the processor, which could fasten the memory request time. The bad part is that more registers, more MUXes and gates are used to determine the logic for replacement policy, and more logic to check four tags at the same time, which could potentially increase the latency for a cycle.

Parameterized Cache

Our group parameterized the cache by only parameterizing the number of sets in the cache. We didn't parameterize the number of ways in cache since it is relatively hard to implement, which involves changing the pseudo-LRU logic.

We parameterize the number of sets by first defining the parameter for the number of sets. Then, we pass this parameter to the array modules we instantiate in datapath, so the module can also use this parameter. Therefore, when we want to change the number of ways for the performance, we can change the defined number in the datapath. Additionally, we also partly parameterize the number of ways except for parameterizing the p-LRU logic. We achieved this by making array instantiation for data, dirty, valid and tag array. Since the number of ways determine the number of module instantiation for each of these four, we could change the number of ways by increasing the size of the array in such instantiation.

The tradeoff of parameterizing the cache could be the area of cache increases, causing the whole area report to become larger.

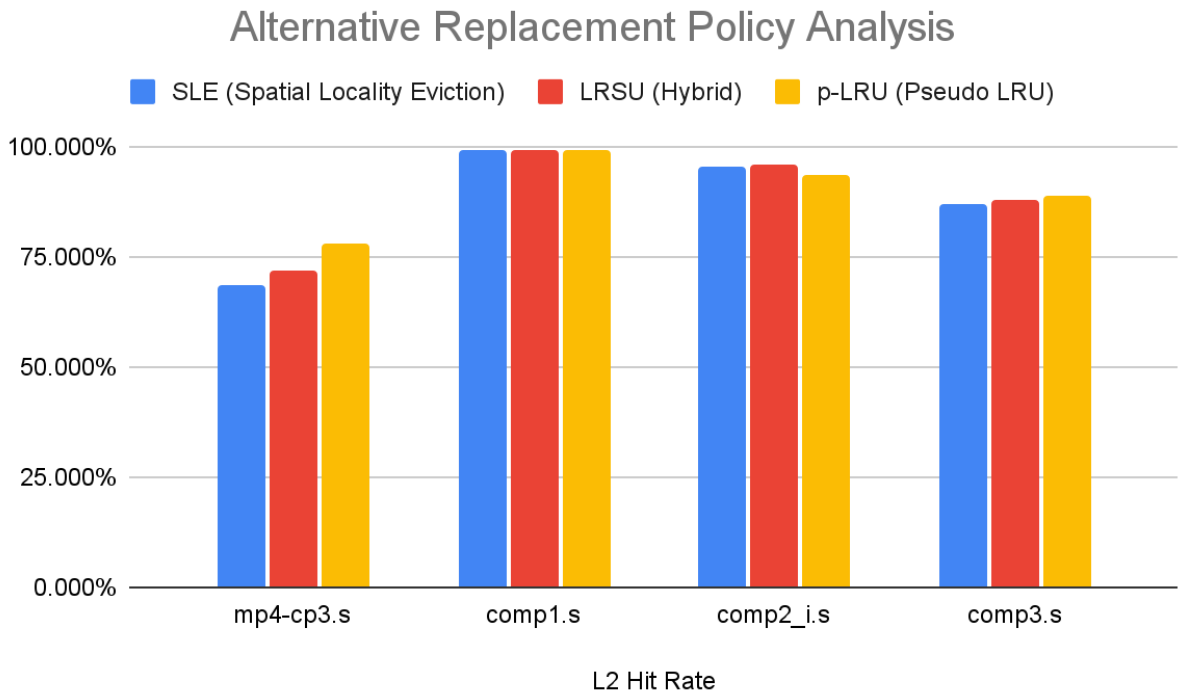
Alternative Replacement Policy

At a fundamental level, caches work because of spatial and temporal locality. p-LRU exploits temporal locality— afterall, it is least "recently" used. So it only makes sense for us to explore a replacement policy that uses spatial locality. The policy is simple: *evict the cache line that is farthest from the current memory address given by the CPU*. For I-cache, the memory address given by the CPU is always PC. For D-cache, it is the memory address that the CPU wants to read or write to. We call this Spatial Locality Eviction (SLE).

One major advantage of SLE is that it eliminates the need for an LRU array, which can be large depending on the associativity and the number of sets. Instead the distance of each way from the current PC is calculated combinationally. The downside is potentially high power consumption since the adder circuit is constantly toggling in our implementation. It is difficult to judge whether the critical path will be shorter since it depends on whether the delay to read the LRU array is higher than the delay to compute the distance.

We also explored a hybrid replacement policy that is a combination of p-LRU and SLE, which we call LRSU. For a 4-way set associative cache, the p-LRU is a decision tree with 2 levels (assuming root is level 0). LRSU uses least recently used to decide if way 01 or way 23 should be evicted. Then, it uses the distance from the CPU memory address to decide between way 0 and way 1 or way 2 and way 3.

The figure below compares the hit rate of L2 cache with SLE, p-LRU, and LRSU.



We can make the following observations.

1. For competition 1, all three replacement policies perform equally well since there are no non-compulsory misses. A bad replacement policy lowers the hit rate of a cache because it replaces a cache line that will be needed soon after. But this does not happen with competition 1 since we never bring back a cache line. No non-compulsory miss means each distinct memory address is loaded into the cache at most once.
2. For CP3 code and competition 3, SLE performs the worst, p-LRU performs the best, and LRSU performs somewhere in the middle.
3. For CP2, we see interestingly that LRSU performs the best.

One interesting design point that we did not experiment with is using different replacement policies for I-cache vs D-cache. For example, using SLE for I-cache and p-LRU for D-cache.

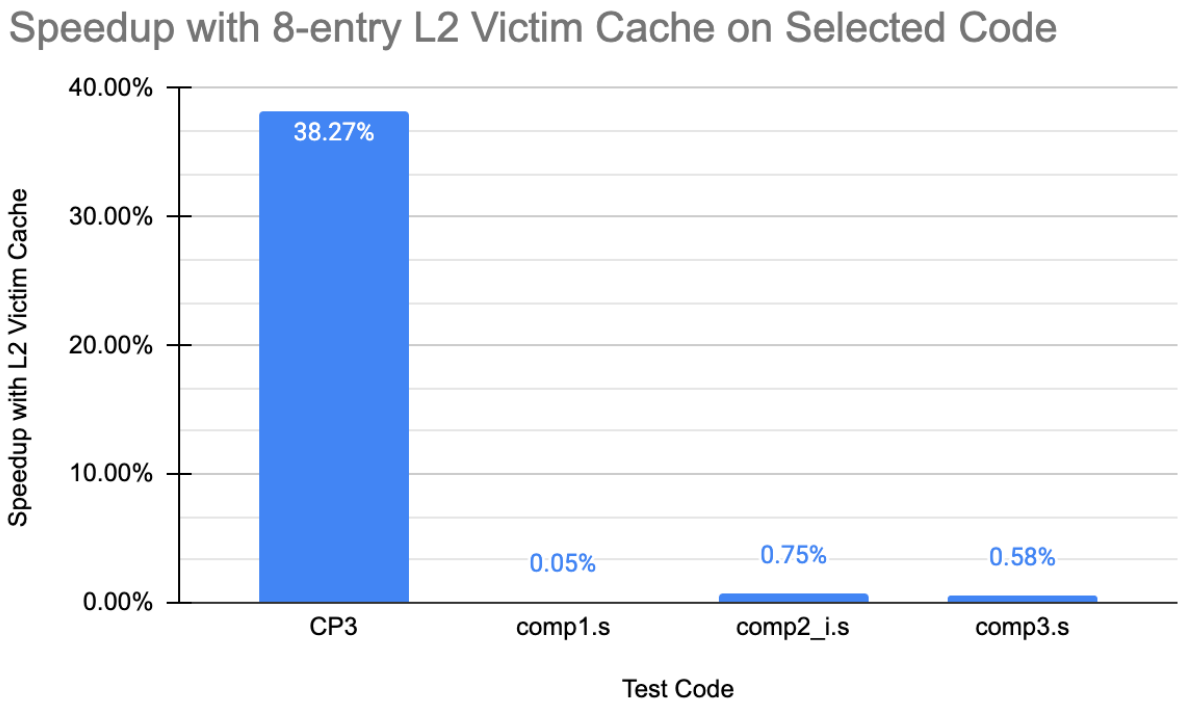
Victim Cache

We turned our 4-way set-associative L2 Cache, which had 8 sets, into a victim cache, which means that we added a fully associative buffer, and that on a writeback eviction, the data was simply loaded onto the fully-associative buffer, and when the L2 cache was free and not handling other requests, which we noted by keeping an internal counter, and when said internal counter was 0, then only we removed an entry from the fully associative buffer into memory, in FIFO

order. The buffer was fully associative, meaning that if there was a hit in the fully associative buffer, then we could modify the entry that we 'hit' with a write, if we received a write request, or we could simply read the entry from anywhere in the fully-associative buffer and return it, if we had received a read request.
Thus, to summarize, the advantages of the Victim Cache were:

- CPU/L1 Cache no longer had to wait for L2 writeback to physical memory to complete before data was retrieved and sent back, writeback to physical memory happened when L2 was free
- The entries of the fully-associative buffer served as additional cache entries, since we could check for hits in the buffer, and if need be, write into any entry of the buffer directly, on a write-request, or read from any entry in the buffer and return the read data, on a read request.

The number of entries for the fully-associative were parameterizable. Here is a figure which shows the speedup on the CP3 and competition codes when our standard L2 4-way set-associative cache was replaced with our L2 4-way set-associative Victim Cache with 8-entries in its fully associative buffer:



From the chart, we notice that the 8-entry L2 Victim Cache provided a massive 38% speedup on the CP3 code, and gave more modest speedups for the competition codes. The speedup can be adjusted by changing the number of entries in the fully-associative buffer.

For larger testcodes, with more data accesses and modifications, such as the three competition codes, the 8 entries may not be sufficient, as the buffer will quickly fill up, and cache time will be diverted to emptying out the buffer, not to answering new requests. Thus for the competition codes, in the future, we should try increasing the number of entries in the buffer and observing the speedup.

Of course, the negative tradeoff of increasing the number of entries is that we now need more comparators, for buffer tag checking, and so have a larger combinational, and potentially, critical path in the processor, and will have more power dissipation due to more logical elements needed.

Thus, the victim cache makes sense for workflows which result in a high number of dirty evictions from the L2, or in other words, evictions after modifications, as we can speedup the dirty writeback into our buffer. A high number of dirty evictions can result from having a smaller cache, or from having many writes with low spatial locality. However, we must note that if we have testcode with many dirty evictions, we will need to increase the number of entries in our buffer, to gain the best performance, or else the L2 will be continually occupied with ensuring that the buffer is not full. Of course, the downside of increasing the buffer size is the increased number of logical elements needed for tag-checking and other operations, which can impact timing and combinational delay.

Memory Stage Leapfrogging

We implemented a simplified memory leapfrogging model, in which a non-memory instruction could directly jump from the Execute stage of the pipeline to the Writeback stage, if the Memory stage was occupied by a data-cache miss on a load or store instruction, and if the instruction in the Execute stage had no data conflicts with the held-up memory instruction (they did not write back to the same register, nor did the later instruction use the result of the memory instruction as one of its operands). Thus, in our simplified model, we avoided issues of data hazards and writing to the register file out of order by choosing to only leapfrog instructions which had no conflicts with the stalled memory instruction.

We have a chart here showing the speedup on CP3 code with our simple memory stage leapfrogging implemented.

CP3 Runtime without Leapfrogging	CP3 Runtime with Leapfrogging	Leapfrogging Speedup	Number of Leapfrogged Instructions
725084250	719721750	0.74%	1449

Our simple model didn't have issues that a more complicated leapfrogging model, which attempted to leapfrog instructions that did have conflicts or dependencies with the stalled memory instruction, would. In that case, the order of committing to the register file would

become very important, and the complexity would become a downside of implementing the feature.

Our simplified model added only a few extra logical elements to the datapath, such as a mux for allowing the execute stage instruction to go directly to the writeback stage in certain circumstances.

Thus, we saved on power consumption compared to a more complex implementation, but we were limited in the amount of Instruction-Level Parallelism (ILP) we could exploit, since we did not leapfrog instructions with data dependencies or conflicts with the instruction that was in the memory stage, and nor did we let any instructions after leapfrog if any instruction before had a conflict with the memory instruction.

More complex implementations could exploit greater ILP for better speedup, such as addressing the aforementioned cases, but at the same time the complex implementation would need more logic gates and would have a higher power dissipation.

Our simplified version of memory leapfrogging is a good feature for workflows which have a lot of instruction-level parallelism, with non-memory instructions having as few data conflicts/dependencies with memory instructions (loads and stores) as possible, since in our implementation we only leapfrog with instructions which do not have any dependencies with the memory instruction currently in the MEM stage.

Conclusions

Summary of Design Objectives

We have implemented a five-stage pipelined processor in RISC-V with a forwarding unit and a stalling unit to prevent data hazards and control hazards. With our design, the processor can avoid RAW and WAW data hazards by checking some specific ordering of instructions and taking actions using forwarding or stalling units. Moreover, a tournament branch predictor to choose between global and local branch predictor using 4-way set associative cache has gained us a branch prediction accuracy as high as 94%. This high accuracy helps the simulation time for the processor by reducing the number of cycles stalled for branch instruction. In addition, a pipelined 4-way set-associative I-cache and the 4-way set associative L2 cache along with its eviction write buffer are successfully implemented and passed all competition codes. Afterwards, these memory hierarchies reduce the simulation time and increase the number of cache hits. We also have tried some alternative replacement policies like spatial locality eviction policy (SLE) and LRSU, a combination of p-LRU and SLE. However, we discovered that p-LRU does sufficient work from the above explanation and decided not to change it. We also alter the

structure of the pipeline processor from cp2 to accomplish memory stage leapfrogging, and it reduces the simulation time for testcodes by a half.