MeOoOw
Final Report

Minseob Shin Ved Vyas Neil Deo

Introduction

Out-of-order execution is the norm in modern CPUs, from servers to edge devices. In this project, we designed and built an out-of-order RISC-V processor implementing the unprivileged RV32IM instruction set, using explicit register renaming, where we map each architectural register directly to a specific physical register via a rename table and a free list, and we maintain a retired register file to restore precise architectural state on flushes. RISC-V is an open-source instruction set architecture with a modular set of extensions that add specific capabilities; we implemented the multiply extension on top of the base 32-bit integer ISA. This report covers three project checkpoints: (1) implementing the fetch stage, (2) adding support for all integer and multiply instructions, and (3) supporting all memory and branch instructions. Finally, we describe the advanced features we implemented and evaluate their impact on processor performance.

Checkpoint One

For checkpoint one where we start off our implementation for the out of order processor, we decided to implement the basis and the building blocks that form the foundation of the processor. For instance, we made a parameterized, synchronous fifo and wrote test cases for it:

- ➢ Populate fifo till full
- ➢ Single cycle write/read
- ➢ Read fifo till empty

We implemented a burst memory deserializer based on the given memory model behavior and wrote tests for it:

- ➢ Multiple consecutive responses from DFP
- ➢ Deassert control signals mid response
- ➢ Back to back responses

We modified our cache implementation from mp_cache to be a read-only cache and connected it to a line buffer. To conclude, we integrated the memory deserializer, instruction cache, fifo, and linebuffer and tested it on fetching consecutive PC addresses, ensuring that there was 1 IPC for instructions in the line buffer.

Checkpoint Two

For checkpoint 2, we implemented a processor that can implement ALU, MUL, and DIV operations out of order. We do this by implementing these structures:

- ➢ Our fetch queue feeds the instruction to a decode unit, that fills an instruction packet struct with relevant information. For example, it sets the specific destination and source register fields, a valid signal, a PC and the next PC.

- ➢ This decode state sends the instruction packet to the Rename/Dispatch unit. This unit pops off an available register from a "Physical Register Free List", which contains all of the available physical registers ready to be used for a given program. It puts the renamed value in the Register Alias Table, which maps physical registers to architectural registers. It then adds the physical register values to the instruction packet, and sends it to the Reorder Buffer (used to ensure that instructions are committed in program order), and Reservation Stations, which hold the instruction before sending it to the functional units until it is "ready" to execute.

- ➢ We implement 3 separate reservation stations, the ALU reservation station, the MUL reservation station and the DIV reservation station. These units hold the instruction until the functional unit is ready and the instruction is valid

- ➢ We implement 3 functional units for the ALU, MUL, and DIV. The ALU implements all I and R type instructions with the exception of AUIPC. The MUL unit is pipelined (4 stages), and the DIV unit is sequential (32 cycles).

- ➢ The output of the functional unit is broadcasted onto 3 separate CDBs, that feed into the rename/dispatch unit, RAT, ROB, and PRF.

- ➢ On ROB commit (when the instruction gets to the top of the circular buffer), the Retired Register File (RRF) is written to with the physical registers that need to be retired, and the free list is updated accordingly

We created an in-depth test suite that tested backpressure and stalling. Specifically, we implemented test programs that populated each reservation station until they were full, and ensured that the frontend of the pipeline stalled. We also implemented test programs that issued instructions until the ROB was full, and ensured that the frontend stalled.

Checkpoint Three

In this checkpoint, we completed baseline implementation of the out of order processor by adding branch and memory instruction support. We used a static not-taken predictor for our branches, and a combined load and store unit. To be specific, we implemented the following structures:

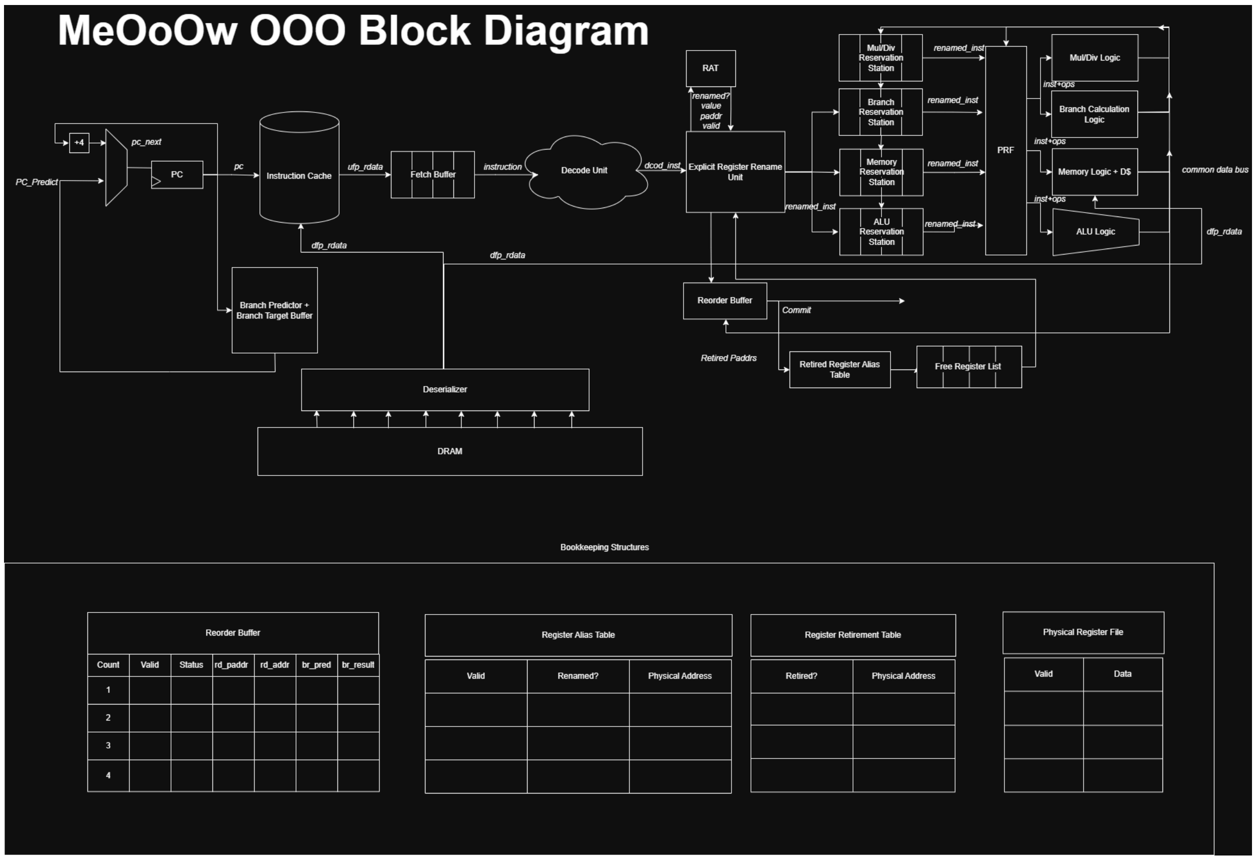
Branch Unit: This unit calculates the branch address, taken/not taken, and the AUIPC instruction. The branch calculation vs prediction executes on instruction commit, which is when the processor gets flushed. To roll back the processor to the architectural state, we use the RRF to roll back the RAT, we store the head pointer of the free list with each branch in the ROB to roll back the free list to, and we clear all reservation stations, pipeline stages, and younger entries in the ROB. While this flush is happening, we fetch the updated PC from memory and start processing the correct instructions.

Memory Unit: This unit calculates the load/store address when they get to the rob head to not disrupt the architectural state of the processor. This module instantiates the data cache and connects the data cache to memory using the memory deserializer for reads, and the memory serializer for writes. The memory module interfaces with the data cache, and stalls until a response comes back from the cache.

Memory Port Arbiter: To ensure cache writebacks and allocation was done between the instruction cache and data cache properly, we implemented an arbiter that was able to provide read priority to instruction cache, then the data cache, and finally write priority to the data cache, since the instruction cache does not write back to main memory.

Memory Access Serializer: To format writebacks to DRAM in the way the model supports them, we implemented a serialier module that takes the write data, ensures that no reads are pending, and sends them to memory. This is used with our existing memory deserializer to permit DRAM read/writes.

After Checkpoint 3, we had implemented this processor:



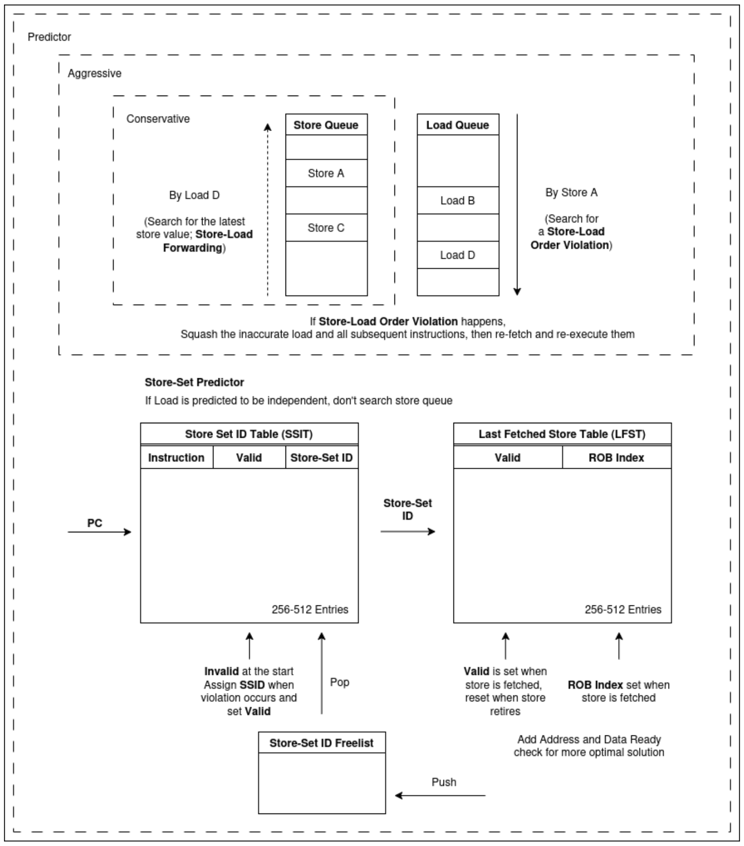
Advanced Features

Split Load-Store queue

Files Modified:

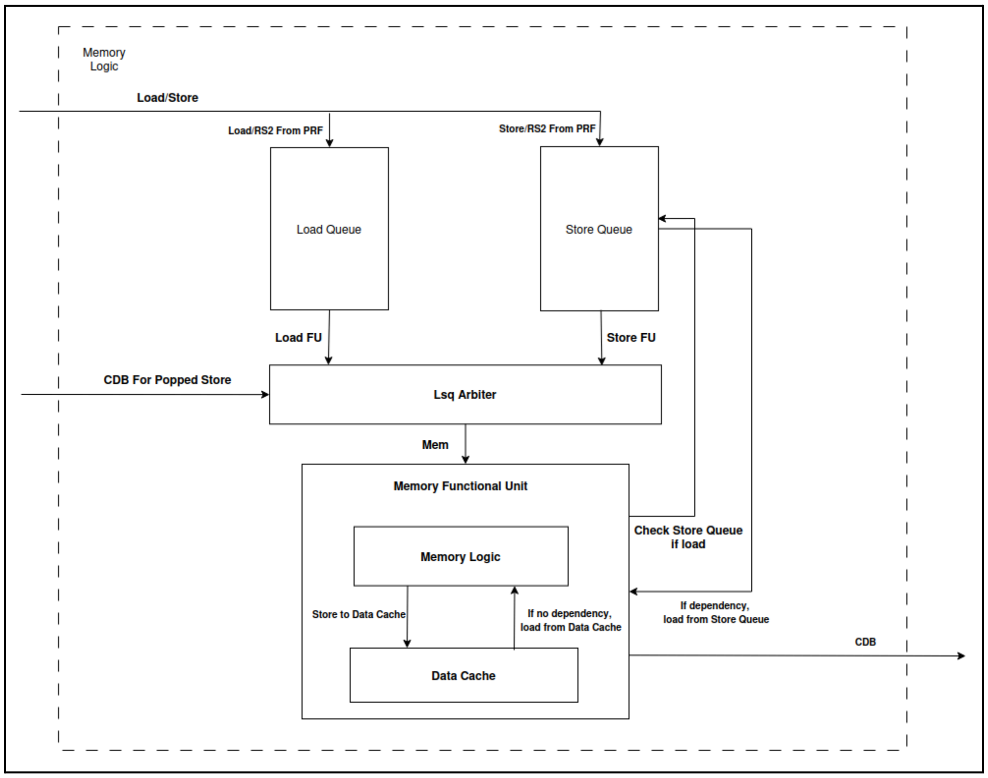
- ➢ mem_load_queue.sv
- ➢ mem_store_queue.sv
- ➢ mem_func_unit.sv
- ➢ lsq_arbiter.sv
- ➢ prf.sv and rob.sv

One of the main bottlenecks in modern out-of-order processors comes from the memory operations. The main problem of the memory instructions in the out of order processor is the unknown address. The memory instructions, as to interface with the memory, inherently requires the memory address that they are reading from/writing to. This in the out of order setting means that the memory instructions that have the address ready (register for the address calculation) will issue out of order. However, this could cause incorrectness if the previous instructions with the unknown address happen to be dependent on the instruction that we issued out of order, i.e. load is issued out of order, but the previous store with unknown address happens to be writing to the same address.


Predictor
Aggressive
Conservative
Store Queue
Store A
Store C
By Load D
(Search for the latest store value; Store-Load Forwarding)
Load Queue
Load B
Load D
By Store A
(Search for a Store-Load Order Violation)
If Store-Load Order Violation happens,
Squash the inaccurate load and all subsequent instructions, then re-fetch and re-execute them
Store-Set Predictor
If Load is predicted to be independent, don't search store queue
Store Set ID Table (SSIT)
Instruction
Valid
Store-Set ID
256-512 Entries
PC
Store-Set ID
Last Fetched Store Table (LFST)
Valid
ROB Index
256-512 Entries
Invalid at the start Assign SSID when violation occurs and set Valid
Pop
Valid is set when store is fetched, reset when store retires
ROB Index set when store is fetched
Store-Set ID Freelist
Add Address and Data Ready check for more optimal solution
Push

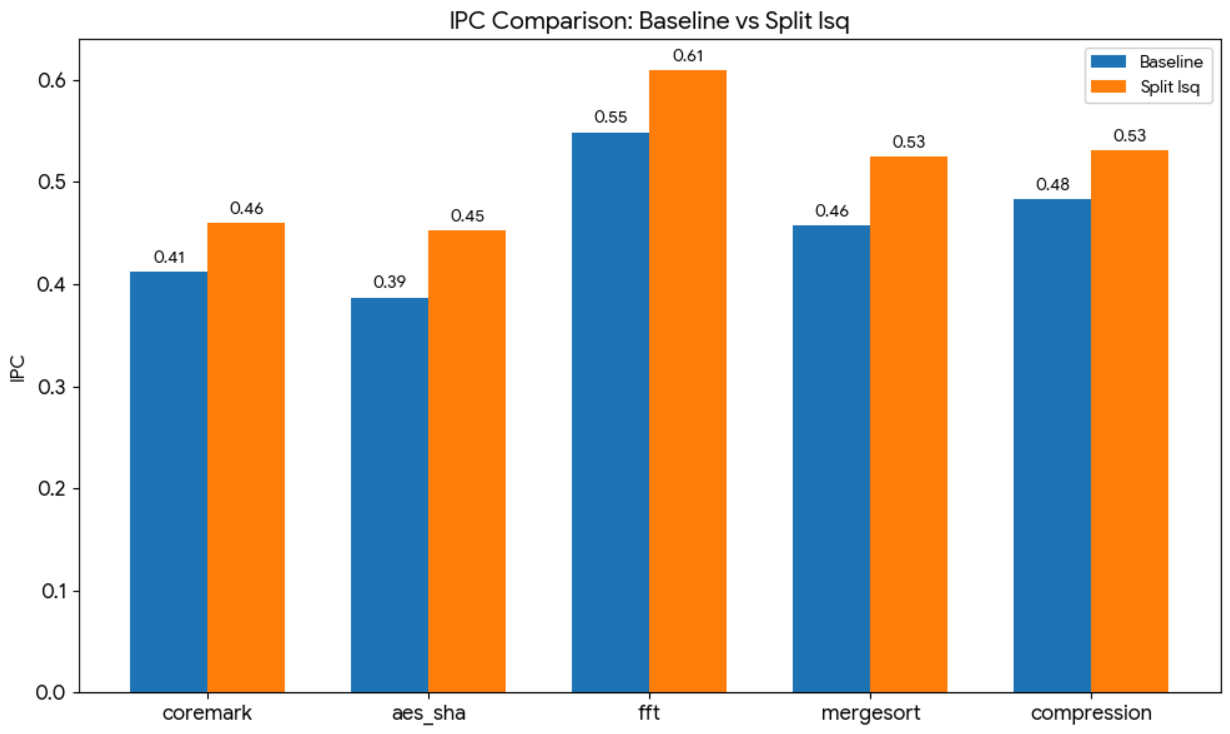
Overview of split load-store queue

There are several ways to handle this through split load-store queue implementation. The basic way is the conservative implementation of split load-store queue, which is what we chose to implement. The conservative implementation of split load-store queue allows us to issue load instructions speculatively if the addresses of all the stores prior to the load instruction is known. This essentially assumes the dependency between the load and stores with unknown addresses and results in speedup in limited applications due to the fact that we have to know the addresses of all the preceding stores. Another way of implementing split load-store queue would be assuming that the loads are not dependent on stores with unknown addresses, which is very aggressive. This implementation is another extreme of split load-store queue and even though it could result in better performance than conservative approach in general cases due to the insignificant amount of store-load dependencies in many applications, it has potential to perform worse as the store-load order violation checking and load replay requires significantly more work. Finally, the most efficient implementation of split load-store queue is the implementation with the load-store dependency predictor. Similar to the idea of branch prediction, predicting the load-store dependency theoretically results in significant speedup as we are able to take the benefit of both sides.



Split load-store queue diagram

We will talk more in depth about our implementation of split load-store queue. We implemented conservative split load-store queue with partial forwarding. Compared with our combined memory instruction queue, we have separated the load and store queue and added an arbiter that handles concurrent issues from the load and store queue. We prioritized the load instruction to leverage speculative load issuing if the previous store addresses are known. Note that we only allow load instructions to issue if they are known to be older than the current ready store instruction. For the partial store-load forwarding, we have added a store-load forwarding write mask and we update the data and write mask if the range check finds the matching previous store on the load execution. For such forwarding purposes, we don't pop the store instruction from the store queue on execution, rather on retirement. This could be further improved in the future by adding a store-load forward buffer that holds the executed but not retired store instructions and data. This means that we will have to write back to the memory when the store instruction retires, and we handle this by giving the retiring store the priority for the memory write to keep the program correct.



Split load-store queue IPC comparison

We have run 5 benchmarks to compare the IPC from the baseline with the combined memory issue queue and the split load-store queue implementation. These include coremark, aes_sha, fft, mergesort, and compression. Here is the description of each benchmark.

Coremark
- ➢ Industry standard benchmarking suite

FFT
- ➢ Performs Fast Fourier Transform on multiple input signals
- ➢ Utilizes arithmetic units heavily
- ➢ Recursive control flow

Mergesort
- ➢ Performs merge sort on multiple input arrays
- ➢ Many data-dependent branches
- ➢ Many loads/stores
- ➢ Recursive control flow

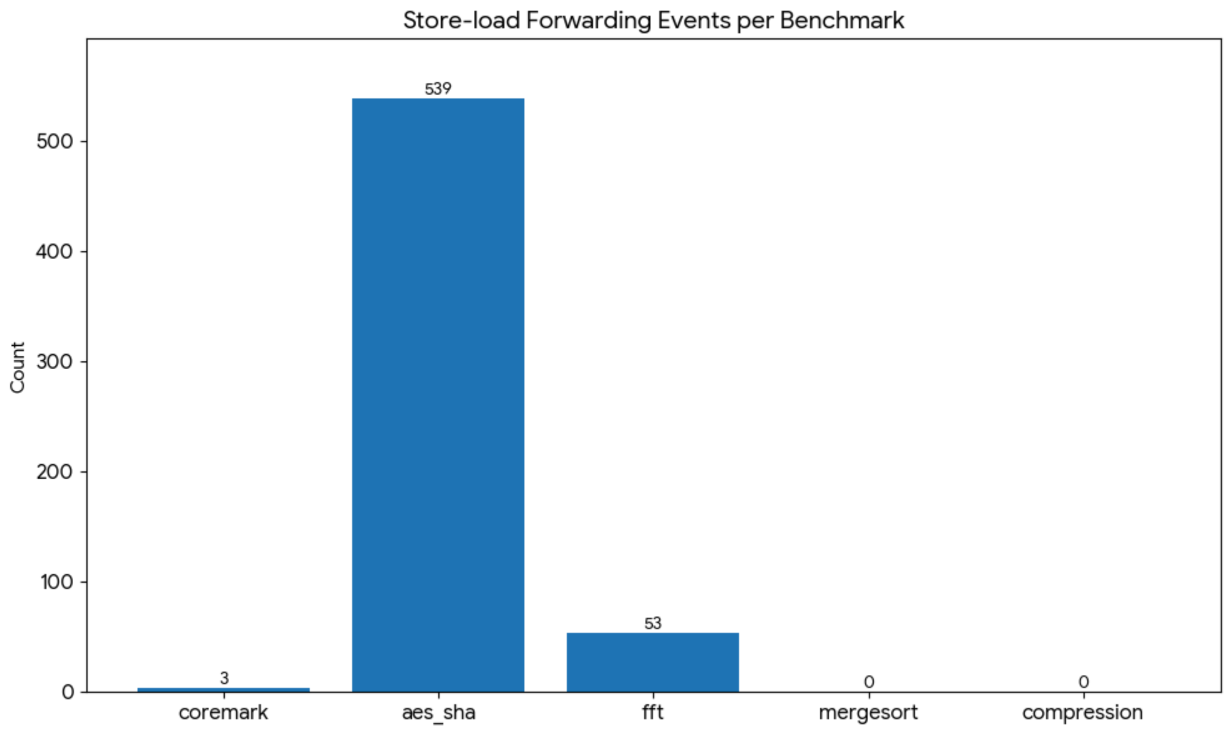
AES-SHA
- ➢ Performs AES encryptions and SHA-256 hash on input stream

- ➢ Has data reuse
- ➢ Many helper functions
- ➢ Utilizes arithmetic units heavily
- ➢ Expresses good ILP
- ➢ Many loads/stores, with high memory-level parallelism

Compression

- ➢ Performs Huffman Compression on a random string from Mary Shelley's Frankenstein
- ➢ Low data reuse
- ➢ Large loops with data-dependent accesses.
- ➢ Link to full text of Frankenstein: https://www.gutenberg.org/ebooks/84

From our observation, aes_sha shows the highest improvement at 17.08% and compression shows the lowest improvement at 10.04%. To further prove the improvement of the split load-store queue, we have added a performance counter that tracks the number of store-load forwardings in each benchmark suite.



Store-load Forwarding Count

From our collection of the number of store-load forwardings, we were able to observe the large number of store-load forwardings in aes_sha. This makes sense because we were indeed able to gain the highest improvement in aes_sha through the implementation of split load-store queue. It is interesting to note that other benchmark suites without a large number of store-load forwardings were still able to gain significant improvement in performance. We believe that this is due to the fact that implementation of split load-store queue enabled speculative load instruction, even though it is limited scope of speculation, it certainly played an important role in improving the performance. Supporting this, the mergesort with the high number of memory instructions had the second largest improvement with 14.68%. This analysis illustrates the importance of optimizing and eliminating memory bottlenecks in the modern out-of-order processor.

Early Branch Recovery

Files Modified:

- ➢ br_sta.sv
- ➢ rat.sv
- ➢ rob.sv
- ➢ Lots of other files had minor changes to support EBR

General Description

During our implementation of the baseline OoO processor, we implemented branches (and jalr instructions) by resolving mispredicts on their commit. Although valid, this approach does not take advantage of the time between when the branch output is calculated and when it is actually committed. This is where early branch recovery (EBR) can help. Reacting to a branch misprediction when it is calculated, as opposed to when it is committed, can help the processor by allowing for more time for the new cacheline to be loaded into the linebuffer and fetch queue. Here, misprediction can mean two things: mispredicted direction (branch instructions) and mispredicted target (jalr instructions). However, this approach makes branch recovery significantly more complex, as we cannot rely on the architectural structures that maintained state through mispredictions. The retired register file, which served as our rollback structure for the register alias table (RAT), is no longer a valid rollback method, since it changes on commit, not calculation. To solve this problem, we must checkpoint several states of the processor corresponding to branches in flight in the processor. Our checkpoints must include the state of the RAT and its valid signals, it should include the free list head, so we can roll back the free list as well, and it should include the predicted target address and direction. Further, there needs to be a mechanism to index into these checkpoint registers. We use branch stack masks to achieve this functionality. A branch stack mask is a N (our processor has a 4-wide EBR unit) wide binary vector. Element i of the vector being set signifies that the instruction depends on the branch in checkpoint i. Consider the following code snippet:

(i1) ADDI x1, x0, 5

```
(i2) BEQ x0, x1, 16
(i3) ADDI x2, x0, 10
(i4) BLTU x1, x2, 24
```

In the code above, assuming all branch checkpoints are clear, i1 would have a branch stack mask of 0000, meaning that it does not depend on any branches. If any branch resolves as either a mispredict or a correct prediction, the validity of i1 would not depend on it. Instruction i2, upon renaming, would trigger a checkpoint creation, since it is a branch/jalr instruction. This checkpoint would store the state of the RAT, the head of the free list, and this branch's target and direction predictions upon renaming. It would also update the current branch stack mask. This mask is used to tag all in-flight instructions with their 'branch dependency' which marks the older, unresolved branches that this instruction depends on resolving correctly to be valid. Instruction i3 is renamed and gets tagged with a branch stack mask of 0001. This signifies that if i2 resolves as a mispredict, i3 and all following instructions with index 0 set to 1 must be cleared. Then, as i4 and future instructions fill the processor, the branch stack mask would update accordingly, and with them would the dependencies.

Branch resolution can be categorized into two outcomes: correct prediction and misprediction. Our branch predictor predicts the direction of branch instructions, and the actual calculation/processing of the instruction in the branch functional unit confirms this prediction to be correct or a misprediction. In the case of a correct prediction for branch i, all in-flight instructions that were dependent on this branch (branch stack mask index i set) will have index i cleared in their branch stack mask, those checkpoint registers will be marked invalid. In the case of a misprediction for branch i, the EBR mechanism will flush all instructions with index i set in their branch stack mask. For example, a branch mispredict resolution would have a branch stack mask of a one-hot vector that corresponds to the branch that just resolved. It will also roll back the free list head and RAT to their checkpointed value. The tail pointer of the reorder buffer (ROB) is rolled back to the ROB index after the branch instruction that was just calculated, so all newly fetched instructions overwrite the entries of the ROB that were flushed. Finally, the PC is updated with the new calculated branch target, and the processor starts fetching values from this address. It is important to note the changes that this makes in the flushing behavior. Before, it was safe to squash *all* in-flight instructions, and clear the entire front-end on a branch mispredict. Now, the EBR unit must only squash the instructions that were dependent on the resolved branch. The front-end is still free to be cleared on a flush, as any fetched instructions not yet dispatched are guaranteed to be dependent on the branches in-flight.

Description of Changes

To implement EBR, the changes we made can be grouped into 3 chunks: checkpointing, recovery, and dispatching. Checkpointing refers to the changes made for creating and maintaining the checkpoints required for rollback. This includes changes to the RAT, Free list, and the creation of an EBR unit (br_sta.sv). Recovery refers to the changes made for resolving a branch, whether or not it was a mispredict. This virtually affects every single structure in the module past the fetch queue. Finally, dispatching refers to the changes made for enforcing early branch recovery in the processor, changing the rename/dispatch unit and the EBR unit.

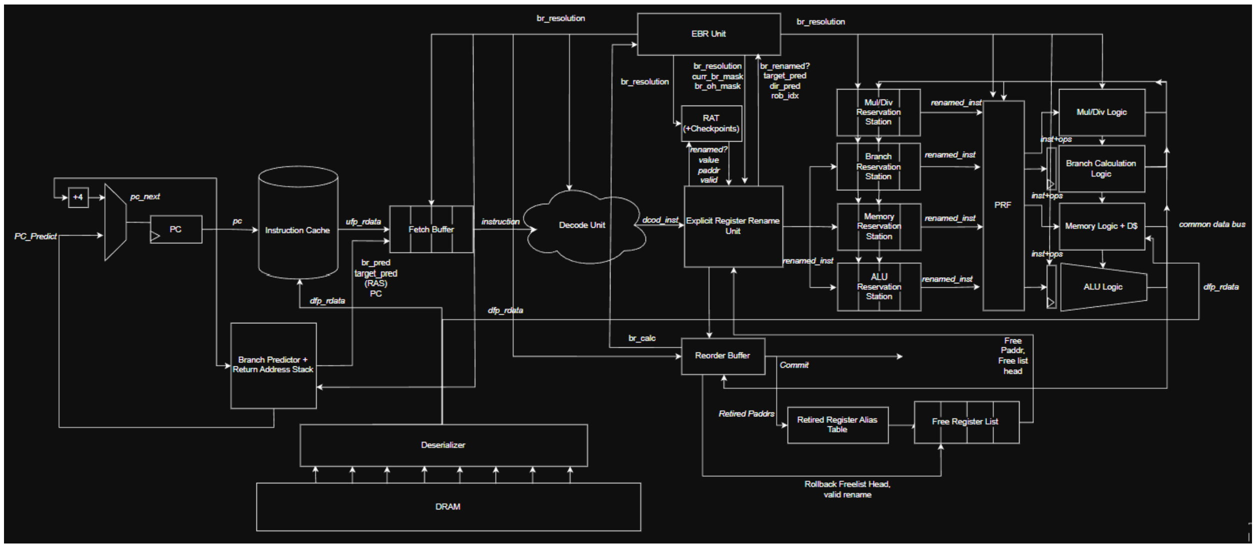


Figure E1: EBR-Processor Interface

In the above block diagram, we note the large number of consumers of the branch resolution signals, which includes the branch stack mask of the resolved branch and whether the branch resolved as a mispredict.

Checkpointing the state of the processor during the renaming of the branch instruction includes saving the state of the RAT, the free list head, and some information about the branch. However, one bug we ran into while testing was when the RAT checkpoints would be reloaded on a branch mispredict, but instructions that were in-flight during the checkpoint and broadcast on the CDB after did not reflect in the restored values, leading to indefinite stalling. To resolve this, we implemented speculative checkpoint repair, a technique that updates checkpoints along with the original RAT. This way, we are able to maintain the checkpoints as they consume completed instruction data from instructions that are not dependent on the checkpoint. Reloading from the checkpoint is done through the branch resolution signals: once a branch resolves as a mispredict, we restore values from that checkpoint given the branch stack mask of the resolved branch/jalr.

Repairing the backend of the processor includes clearing the reservation stations of any instructions squashed by a branch misprediction, clearing any in-flight instructions in the functional units, disregarding any squashed instructions on the CDBs, rolling back the ROB, and clearing the front-end of the processor. These actions are all performed through checking the branch stack mask of the resolved branch against the tagged branch stack mask of the instruction. This tagged branch stack mask assigned at rename is simply cleared at the index of a branch that resolves as a correct prediction. In the case of a correct prediction, the front-end is allowed to continue, and all in-flight instructions are left as they were. Rolling back the ROB is done by setting the tail pointer of the ROB to be one after the branch instruction. The reason we do not roll back the ROB head or the free list tail is because those are structures that are updated on commit. In other words, they only change with the architectural state of the processor, not the speculative state, so we do not need to checkpoint or restore them.

Dispatching the instructions involves tagging the renamed instructions with a branch stack mask and updating the current branch stack mask on branch renames. In the processor, the current branch stack mask is a register maintained by the EBR unit. This register tracks all in-flight branches at any given point in time. The value of this register is tagged onto all renamed instructions. This way, if any of the future resolutions of these branches are mispredictions, then these instructions can be squashed. Otherwise, the individual bits in this signal that correspond to resolved correct predictions can be cleared. The current branch stack mask is consumed by the rename/dispatch unit, produced by the EBR unit. When a branch instruction is renamed, it is tagged with another branch stack mask, specifically the one-hot branch stack mask that corresponds to itself. This mask is used when resolving the branch.

Results

This feature was implemented to mitigate the high cost of branch misprediction in our processor. We can track the number of branches that resolve before they are at the head of the rob and compare that against the number of total branches in the program:

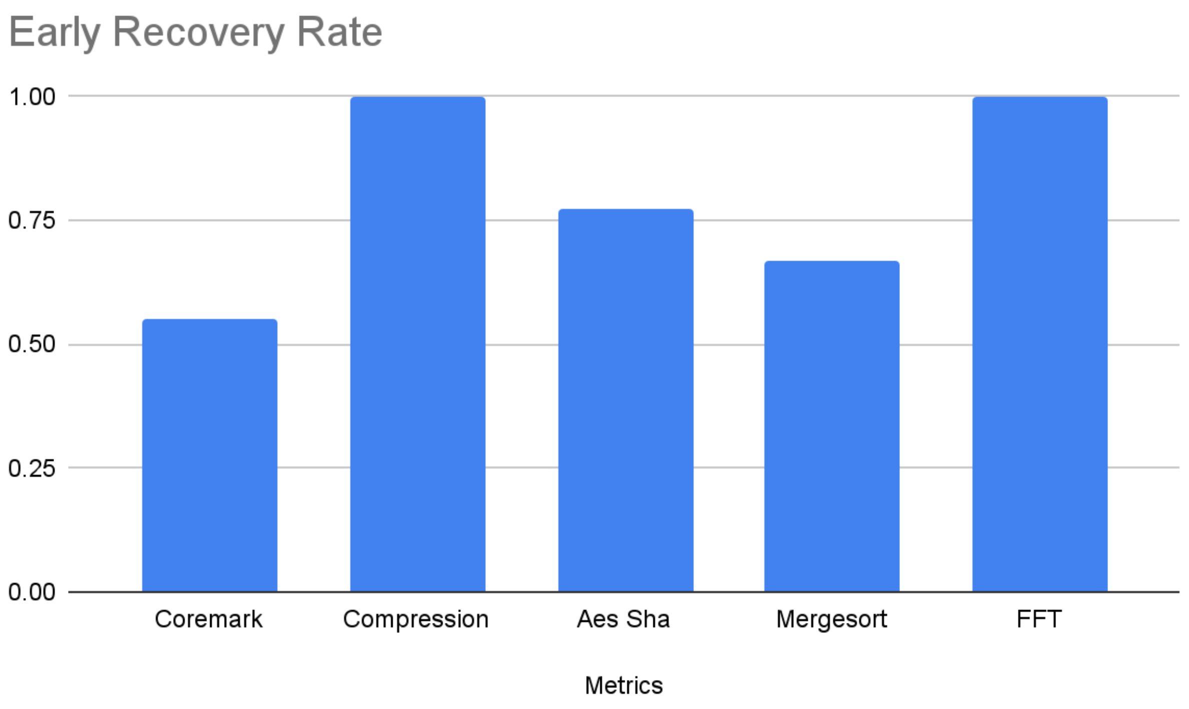


Figure E2: Early Recovery Rate by Benchmark

We see in the above chart that the early recovery rate for the programs in our CP3 benchmark suite are above 50%, with some like compression and FFT getting close to a 100% early recovery rate. This means that the programs are able to take advantage of EBR, since a majority of all the branches in the code resolve before the instruction is at the head of the ROB. The early recovery rate also gives a good understanding of how the performance of EBR depends on the programs run on the processor. If the program includes branches with operands that resolve quickly, EBR can be very effective. However, if those operands take longer to resolve, then EBR and flushing at ROB head get similar performance.

We can also visualize the efficacy of the EBR mechanism by examining how many instructions were squashed on average in the programs:

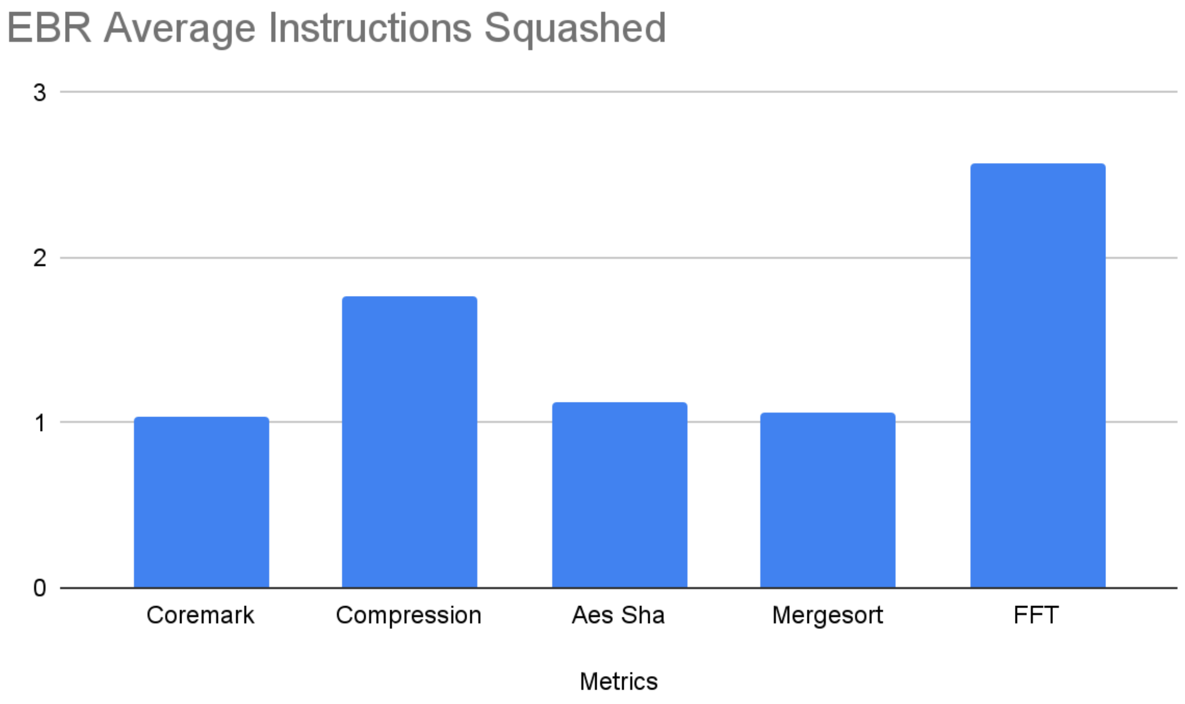


Figure E3: Number of Instructions Squashed by Benchmark

As seen above, all of the programs averaged over one squashed instruction per branch. This is great performance, especially when complemented with the above graph comparing the early recovery rate. Given that, in benchmarks like coremark and mergesort, the early recovery rates were relatively low, saving one instruction per branch resolution on average is a great result. This highlights the importance of the EBR mechanism. Since we are able to react on branch calculation, we are able to squash a lower number of instructions, which means less wasteful work enters the processor.

Finally, below is a chart comparing the IPC on the benchmarks before and after implementing EBR.

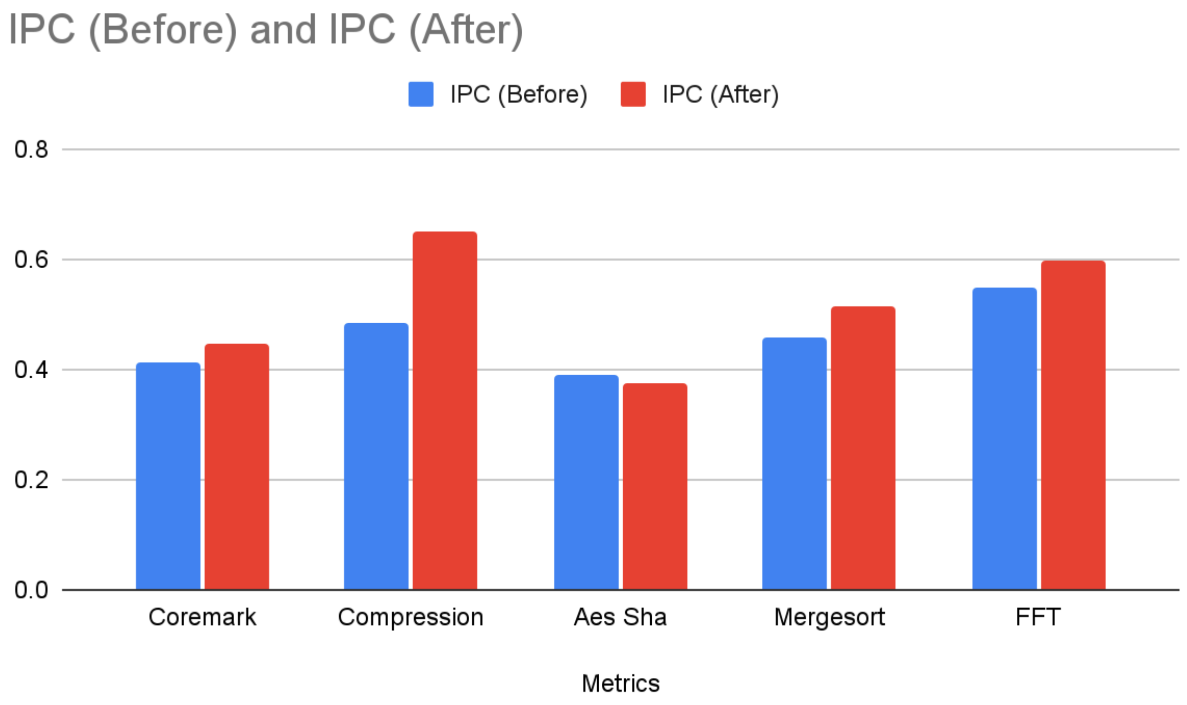


Figure E4: EBR IPC Improvements (100 MHz)

We can appreciate the performance benefits of EBR, especially in benchmarks like compression, where we see more than a 30% increase in performance, due to the loopy nature of the program. All in all, EBR enables the processor to react to branch mispredictions faster, expediting the recovery of the processor's speculative state.

Branch Prediction

General Description

For our Branch Prediction Unit, we decided to make a Tournament-Style Branch Predictor Unit that selects between a Global History predictor and a Local History predictor. We implemented everything in flip-flops, so we didn't have to split our instruction fetch across two stages. The BPU is implemented in full in bpu.sv in the main branch. It predicts based on the PC register, and tags the input of the fetch queue with a prediction, which then feeds to the decode register (Figure B1). It updates based on when the branch it forms a prediction on commits, which is when it is at the head of the ROB (Figure B2). The next state logic is shown in Figure B3. Note that some of the names for the signals aren't reflected in the actual logic implementation, in order to make it clear that the signals come from the ROB. Specifically, all of the update signals change from prefix “ffp_” (front-end facing port) to prefix “ROB_”.

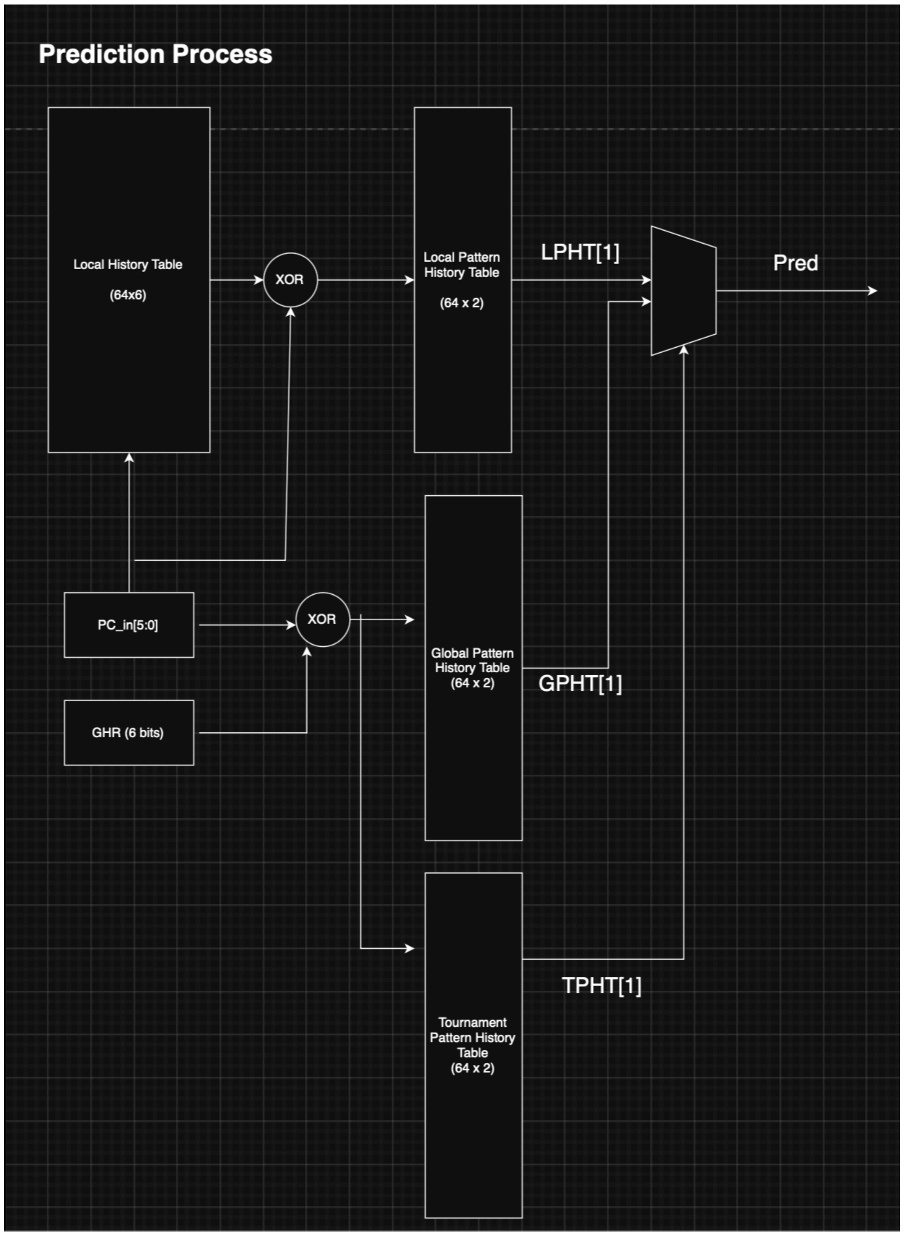


Figure B1: Prediction Process

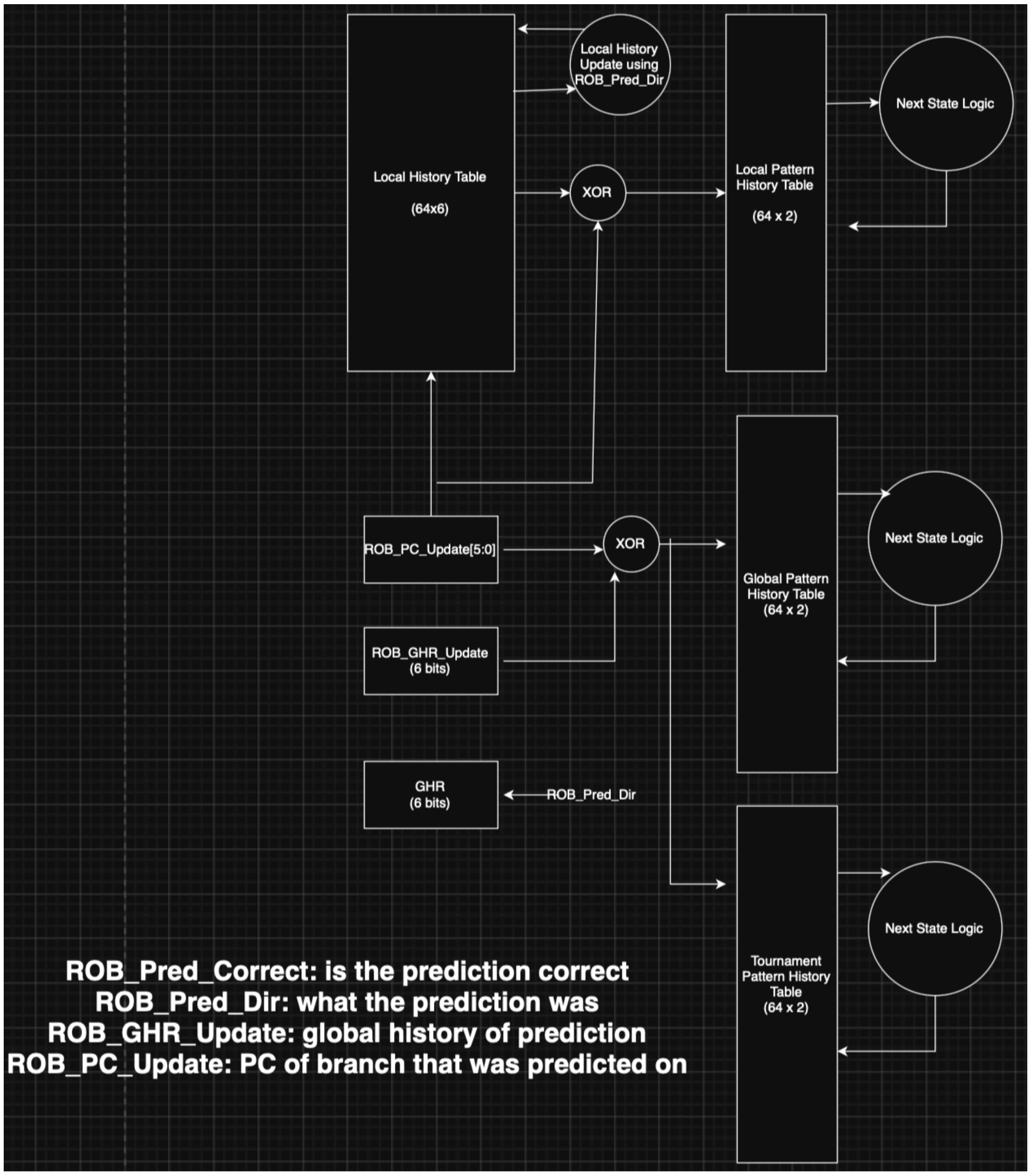

Local History Table
(64x6)
Local History Update using ROB_Pred_Dir
XOR
Local Pattern History Table
(64 x 2)
Next State Logic
ROB_PC_Update[5:0]
XOR
Global Pattern History Table
(64 x 2)
Next State Logic
ROB_GHR_Update
(6 bits)
GHR
(6 bits)
ROB_Pred_Dir
Tournament Pattern History Table
(64 x 2)
Next State Logic
ROB_Pred_Correct: is the prediction correct
ROB_Pred_Dir: what the prediction was
ROB_GHR_Update: global history of prediction
ROB_PC_Update: PC of branch that was predicted on

Figure B2: Updating the Branch Predictor

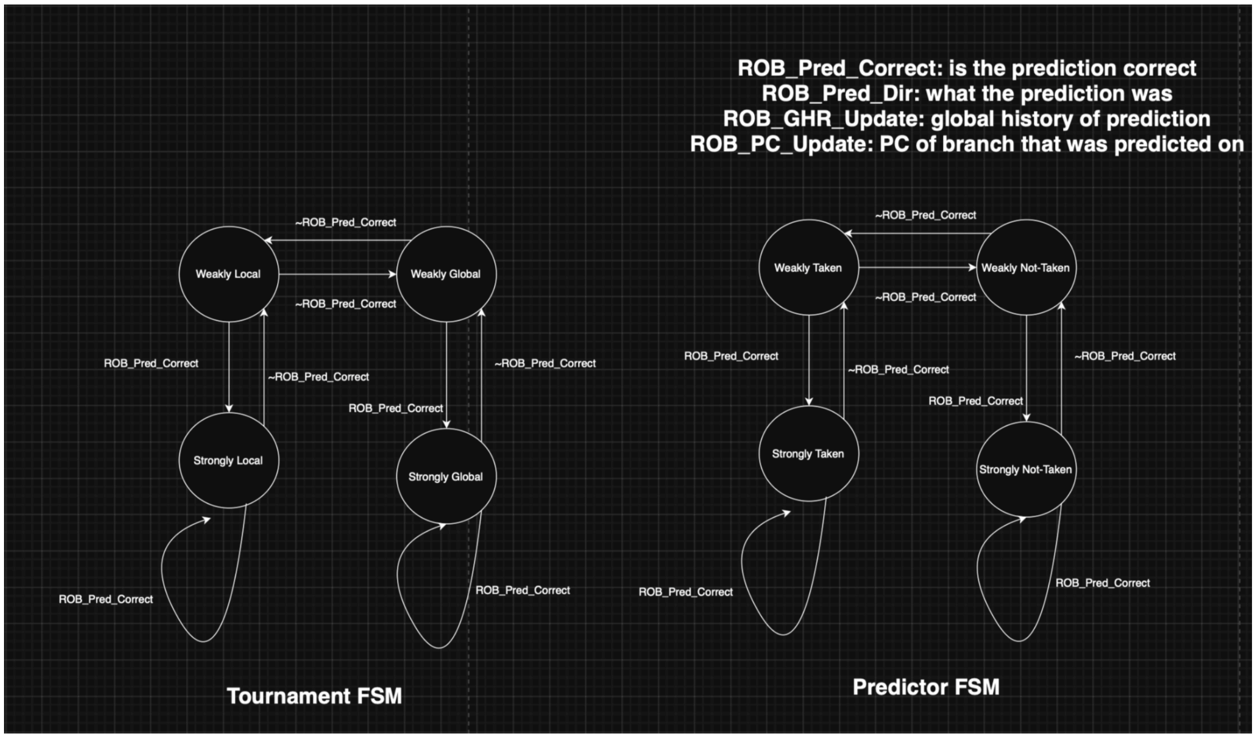


Figure B3: Update FSM

Description of Structures

The Local History predictor is a 2-level predictor, that takes certain bits of the PC, indexes into a Local History Table that encodes all of the local history information of the branch. If the branch was taken, a 1 is shifted into the local history. If it was not taken, a 0 is shifted into the local history. After the local history is retrieved, it is used to index into a Pattern History Table, which is a table that contains 2-bit saturating counters. In order to increase accuracy, and correlate parts of the PC to the local history when indexing into the Pattern History Table, we XOR together the PC and the Local History as an index to the PHT. When the branch commits, the specific PC of that branch is used to update its local history in the Local History Table, and update the state in the Pattern History Table.

The Global History predictor is a GShare predictor. It takes certain bits of the PC, XORs it with a global history register (GHR), and indexes into a Global Pattern History Table of 2-bit saturating counters. The GHR contains information about all branches that have committed. When a branch commits, its outcome is shifted into the GHR. The GHR is also checkpointed and correlated with each prediction, so that when we want to update the state of the 2-bit saturating counters, we can index into the specific entry that was used for the prediction itself.

The Tournament Predictor uses the same index as the GShare predictor (certain bits of the PC XORed with the GHR), and indexes into a tournament pattern history table. This pattern history table has a 2-bit saturating counter that chooses between the local and global predictor for the certain global branch state of the processor.

Instead of zeroing out all structures on reset, we use valid arrays for each of the structures, which save on power, while also ensuring that we do not propagate X's through the design.

Impact of Table Size on Accuracy

Using performance counters to track the accuracy of the predictor, we can see a correlation between table size and the accuracy of the predictor.

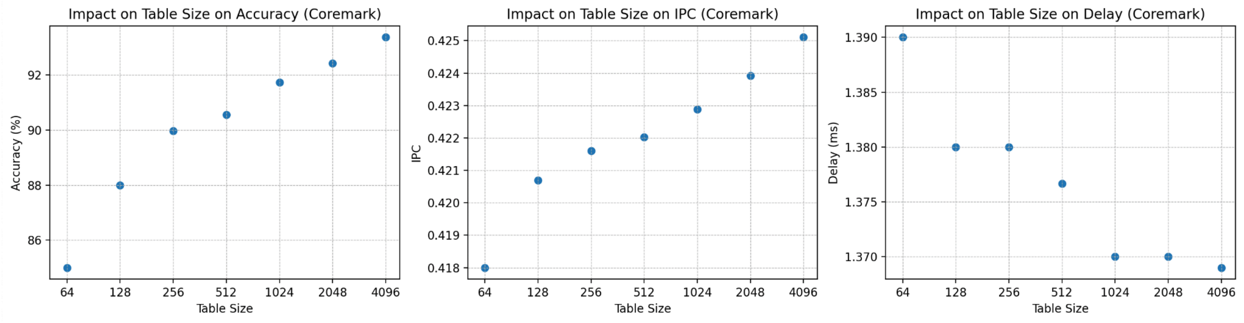


Figure B5: Impact of Table Size on Accuracy, IPC, and Delay for Coremark

Based on Figure B5, in order to optimize for power, while not losing too much accuracy, we decided to pick table sizes of 64 entries for all structures. While increasing the table size causes the accuracy, IPC, and Delay to increase, we found that it comes at a tradeoff of significant power consumption and a significant area increase. Because we chose to use Flip-Flops (as they can be easily integrated into a 1-stage fetch), we decided on a smaller structure size. If we had decided to implement the Branch Predictor in SRAMs, we probably would have been able to push the table sizes to above 256 entries, however it would come at an IPC hit, as we would have to add another stage to our instruction fetch.

Speedup of Integrated Design

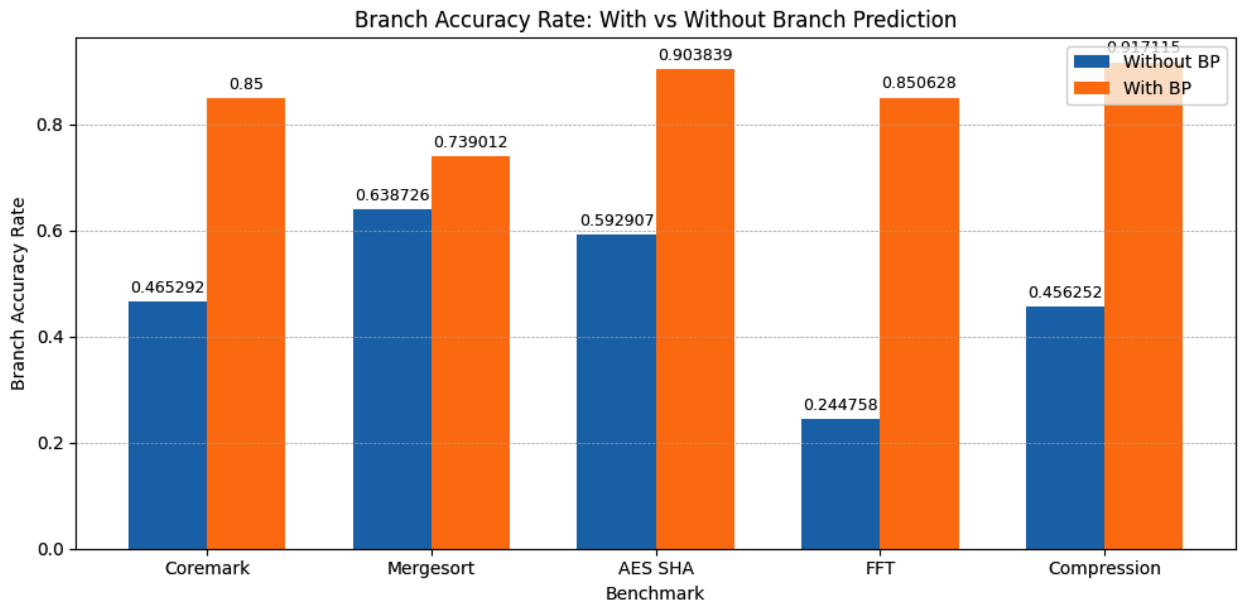


Figure B6: Branch Accuracy for a 64-entry table size across benchmarks (500 MHz)

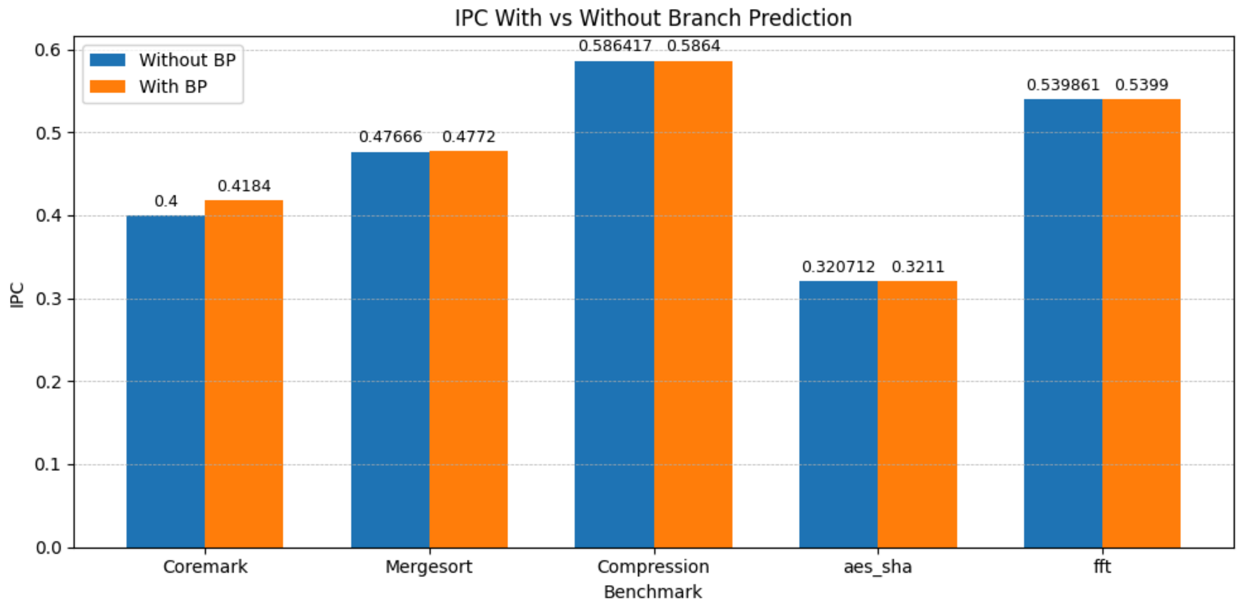


Figure B7: IPC with and without Branch Prediction for a 64-entry table size (and static-not taken) (500 MHz)

Figure B6 shows us how the accuracy changes after the integration of the branch predictor. As expected, for a static predictor, the accuracy degrades significantly. Figure B7 shows us how much IPC improves when integrating a branch predictor, over just predicting static-not-taken. Coremark sees a significant IPC improvement, while aes_sha, mergesort, FFT, and compression see minimal improvement. We believe that our choice of 64-entry table size contributed to this minimal improvement, however it is also possible that those benchmarks contain control flow that doesn't benefit a lot from a combination global/local predictor. It is also possible that the integration of EBR saturates the IPC even if the prediction mechanism isn't super accurate, because quick recovery can mitigate the stalling of mispredicts.

The workloads that the branch prediction unit helps are programs with non-trivial control flow. "Loopy" programs or programs with a lot of function calls/returns see limited speedup from a branch prediction unit. However, programs that have a lot of branches, and especially programs with a good distribution of branch types (those that are correlated heavily with other branches, and those that don't have any correlation with any other branches) see a massive speedup through the integration of the tournament predictor.

Parameterized Cache

A parameterized cache is a cache design that should easily be able to support a different cache size, in terms of the number of sets and ways, just by changing the SETS and WAYS parameter. This design also required a re-engineering of the PLRU in order to support a variable number of ways.

For our instruction cache, we chose a Direct-Mapped cache with 64 sets, and for our data cache, we chose a 16-set 4 way cache. This was specifically chosen in order to limit power consumption and area. Having a direct mapped instruction cache makes sense, as Instruction Caches are simple read-only caches, and instruction fetch benefits from a simple, fast access time. A 4-way data cache was chosen as a balance between power/performance, while taking into account diminishing returns.

The most important part of a parameterized sets and ways cache is the ability to parameterize the pLRU. We made our LRU array store NUM_WAYS-1 bits. In order to ensure that our design is synthesizeable, we use the heap indexing algorithm:

- ➢ To access left child (update a hit on the left subtree), we use the formula 2*Node + 1
- ➢ To access right child (update a hit on the right subtree), we use the formula 2*Node + 2

The bits of the way access itself tell us a lot about which side of the pLRU was most recently accessed. From MSB to LSB, if the bit is set high, we know what side of the pLRU tree was accessed.
For example, a 4-way cache would have a way-index of W1W0. If W1 was set high, we know that the way that was just accessed is either way 2 or way 3, which corresponds to the right side of the tree. Therefore, we traverse the index from MSB to LSB, and if the bit is set high, we set the pLRU value of that specific level to be 1. We can further go to the next subtree by using the heap indexing algorithm. Here is an example of the pLRU indexing algorithm for a hit in way 3:

Way Index: 10
Current pLRU: 000
Current pLRU index node: 0

Iteration 1:
First we look at the MSB of the way index. Since it is set to 1, we set the corresponding pLRU bit to be 1. We update the next index node to be the next subtree by using the heap indexing algorithm, so the index node becomes 2.
pLRU: 001
pLRU index node: 2

Iteration 2:
We look at the LSB of the way index. Since it is set to zero, we set the corresponding pLRU bit to be 0. We update the next node (pLRU bit 2) to be the next subtree by using the heap indexing algorithm, so the index node becomes 2
pLRU: 001
pLRU index node: 4

The final pLRU is 001. This corresponds to a hit in way 3, since that would be 0-1.

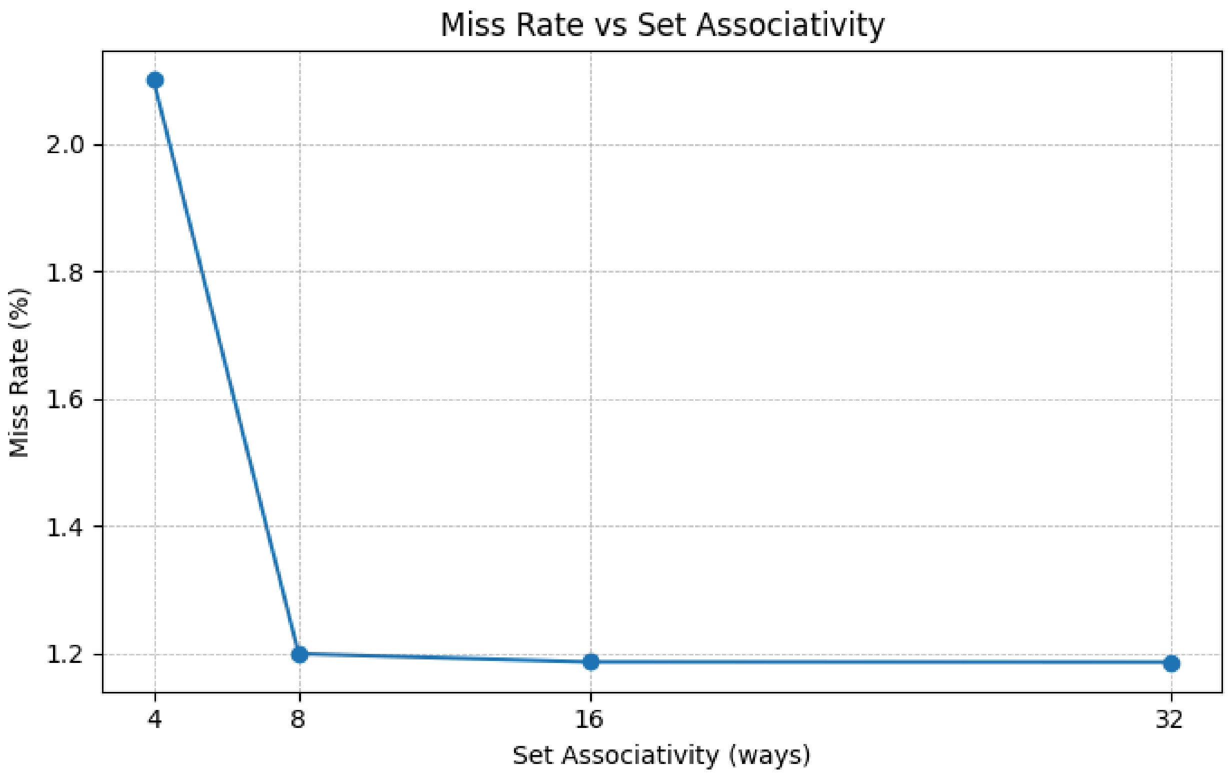


Figure P1: Miss Rate vs Associativity Modulation using the Parameterized Cache

The parametrized cache helps for code that has a lot of Conflict-misses, as increasing the number of ways decreases the total number of conflict misses (Figure P1). Specifically, programs that have large hash-tables and dictionaries benefit a lot from caching, and usually have high conflict misses for smaller caches. Increasing the number of ways can improve their performance. However, for energy and power-limited processors, increasing the sets and ways can be useless, as there is usually a fundamental limit to how many extra sets/ways the cache can handle. It also doesn't help for programs that only operate on one stream of data in one pass, for example workloads that iterate through a large chunk of data that is copied into memory.

Age-ordered Issue Scheduling

Files Modified:

- ➢ age_ordered_res_sta.sv

General Description

Age Ordered Issue Scheduling is an optimization where instructions that live in the reservation station for longer get priority over younger instructions. When instructions get dispatched to their reservation station, they are 'woken up' by their pending operands being broadcast on the CDB. In many cases, multiple instructions can be ready within one cycle. In this case, a naive implementation of the reservation station would issue the first ready instruction from the top of the reservation station. The inherent ordering of the reservation station usually does not have any relation to the age of the instruction. Hence, sometimes, younger instructions can be dispatched to the functional units over older ones. Especially in functional units like the 32-cycle division, this can delay the commit of an instruction in the ROB by quite a bit.

To implement the age ordered issue queue, we use a structure called the age matrix, where age_matrix[i][j] = 1 means that the element at index i is older than the element at index j. Whenever there are ready signals in the reservation station, we check the ready elements against elements in the age matrix and ensure that we pick the oldest element that is ready. This can mitigate the earlier described issue where older elements are skipped due to their index in the reservation station.

Results

In the below chart, we see the scheduling contention breakdown by reservation station by benchmark:

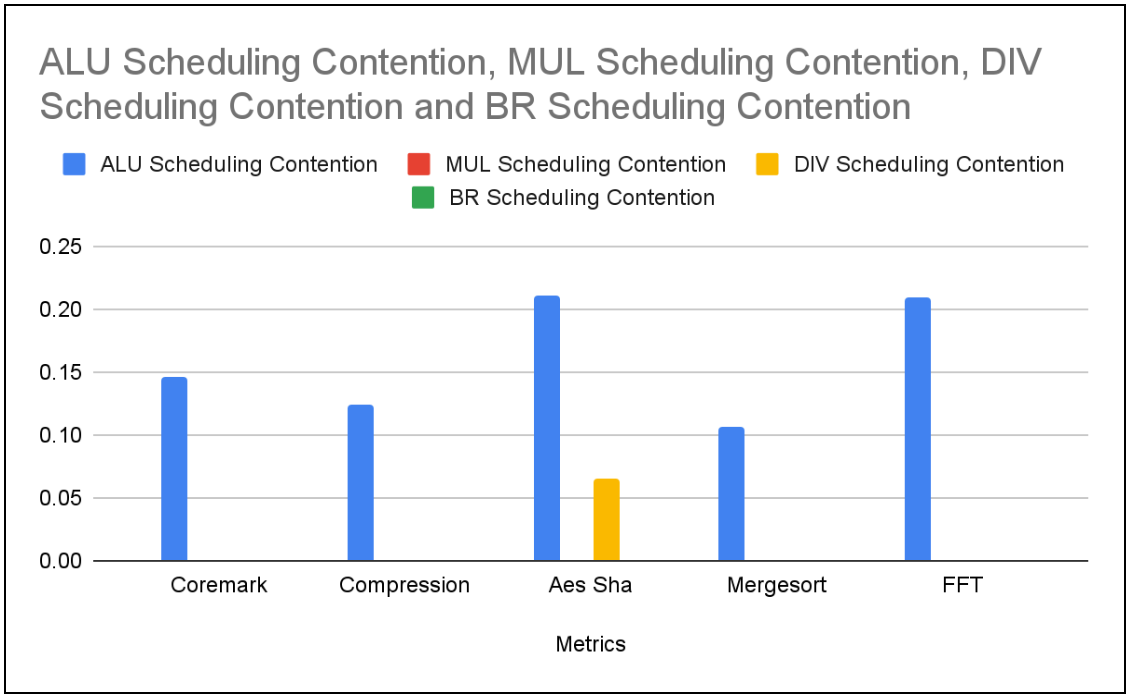


Figure A1: Scheduling Contention by Benchmark by Reservation Station

The chart illustrates the scheduling contention in the ALU reservation station. Scheduling contention is defined by having multiple ready instructions in a cycle. There is also some scheduling contention in the DIV reservation station. To resolve these even further, we could consider superscaling the processor to create more functional units that can reduce the contention in the reservation stations. For this feature, we did not see a significant speedup or improvement in IPC. This could be due to older instructions getting dispatched first even without age ordered issue scheduling. This feature is useful only when there are instructions with long wait-time dependencies mixed with instructions with shorter wait-time dependencies.

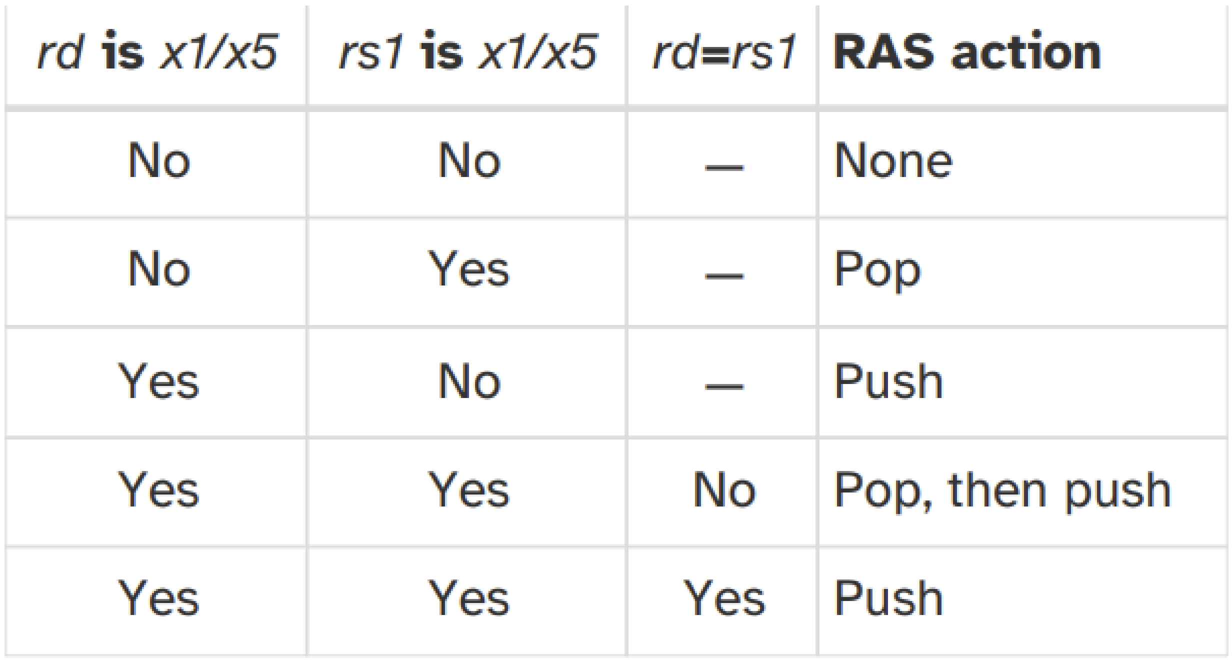
Return-Address Stack

Files Modified:

➢ ras.sv

General Description

A Return Address Stack (RAS) is a small hardware stack that predicts return addresses for function calls to improve branch prediction accuracy. This is specifically useful for unconditional jumps for recursive calls. For example, when the processor fetches a function call instruction, it pushes the return address onto the RAS and when it fetches a return instruction, it pops the top of the RAS to predict the return target. This helps us to model the function call and return behavior and could benefit the processor performance just with a simple stack. The RAS is implemented as a stack that wraps around, making it able to accept a burst of return addresses, but at the cost of potentially losing some return addresses due to this wrap behavior. To implement this, we consider the pop and push behavior as detailed in the RISC-V Manual:



rd **is** *x1/x5*	*rs1* **is** *x1/x5*	*rd=rs1*	**RAS action**
No	No	–	None
No	Yes	–	Pop
Yes	No	–	Push
Yes	Yes	No	Pop, then push
Yes	Yes	Yes	Push

Figure R1: RAS Push/Pop Behavior

This behavior encapsulates the expected behavior for actions like function calls and other subroutines that can benefit from the return address stack. To make this return address stack compatible with EBR, we included checkpoint registers for the top of the stack. On a branch resolution that is a mispredict, we resolve the branch by setting the top of stack to be the checkpointed value. If a branch resolves as a correct prediction, we simply invalidate the checkpoint to make room for the next branch.

Results

We found that the accuracy of this RAS was very good on the RET subroutine. Specifically, we were able to get 100% accuracy on RET return address predictions using the RAS on all benchmarks except for Mergesort, which had a 67% accuracy on RET return address predictions. We did not see a significant difference in IPC with this implementation, which could be due to the programs in the CP3 benchmark suite not having many function calls.

Conclusion

In conclusion, we implemented a 32-bit Out-of-Order RISC-V CPU with explicit register-renaming, early branch recovery, a tournament-style branch predictor, a parametrized cache, an age-ordered issue scheduler, a return-address stack, and a split-load store queue. By employing and embracing parameterization, this core enables design-space exploration, which we plan on doing. This project was a learning experience for all of us, and we learned a lot about computer architecture, hardware implementation hygiene, and working as a team.