# MP OOO Report

Peter Kim, Jeremy Wu, Vinay Patel

Fall 2024

## Introduction

This MP aimed to create an out-of-order (OOO) pipelined processor. The reason for creating an out-of-order processor was due to the data dependencies that a pipelined processor has. Through this process, we learned how to manage the problems created by an out-of-order processor, as well as the optimizations and concerns relevant to out-of-order execution. This is very relevant due to the prevalence of out-of-order in modern computer architecture. We also used explicit register renaming to facilitate our out-of-order execution, precisely due to its presence in the industry.

## Project Overview

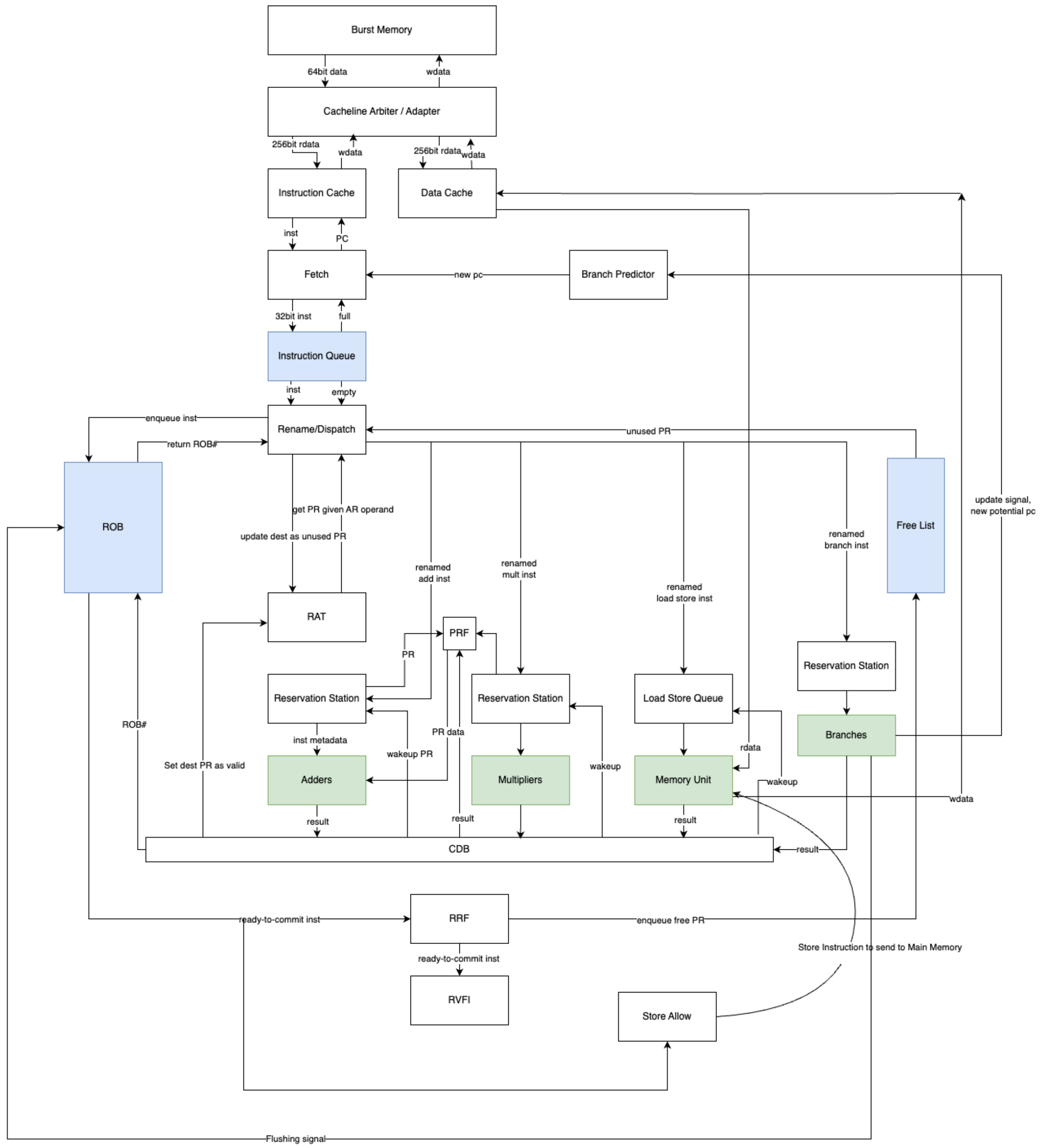
Overall, we implemented an OOO processor that minimized the impact of branch flushes, which meant we were attempting to 1) minimize the number of branch flushes and 2) minimize the impact of a branch flush when it inevitably occurred. Our biggest advanced features were motivated by these goals: EBR and GShare Branch Predictor, which we go into more detail in our Advanced Features section.

Regarding work sharing, we all contributed equally to the overall project. Surprisingly, we were most effective when meeting online as opposed to in person, most likely because we were most effective at developing and debugging on our personal workstations.

## Design Description

### Overview

Our design block diagram, precluding the Branch Stack, is as follows:



## Milestones

### Checkpoint 1

As required by this milestone, we implemented a parameterizable ring-buffer-based implementation queue with two pointers, head and tail. When they were equal, the queue was empty, and when they were wrapped around, it signified that the queue was full. Verifying this queue was important as we used it as the basis for many hardware units, e.g., ROB, LSQ, Free List, etc. We wrote an exhaustive testbench to verify the queue.

Besides creating a cacheline adapter, we also integrated Vinay's cache from the previous MP. Finally, we wrote a simple fetch module that integrated with our adapter and instruction queue and could fetch instructions using a program counter.

### Checkpoint 2

Here, we implemented a partially RISC-V compliant OOO processor that could execute all immediate and register instructions. As requested, we also integrated the Synopsys IPs to support multiplication and division RISC-V M instructions. To test, we modified and ran the random testbench that covered all possible instructions. This was critical as we were able to find bugs like divide-by-zero and fix them promptly.

To test the OOO-ness of our processor, we used the given OOO test file and confirmed that the ready-to-commit signal went high out of order (with respect to the original order of instructions scheduled).

On top of achieving the requirements, we also made our processor compliant with Verilator lint.

### Checkpoint 3

For our final checkpoint, we integrate a cacheline arbiter to arbitrate the access to memory between instruction cache and data cache. Additionally, we added support memory instructions by building a Load Store Queue as outlined in the lab slides (address generation could happen OOO but all loads and stores were initiated when they were at the top of the LSQ i.e. total ordering). Finally, we added support for all control instructions and used a static not-taken branch predictor.

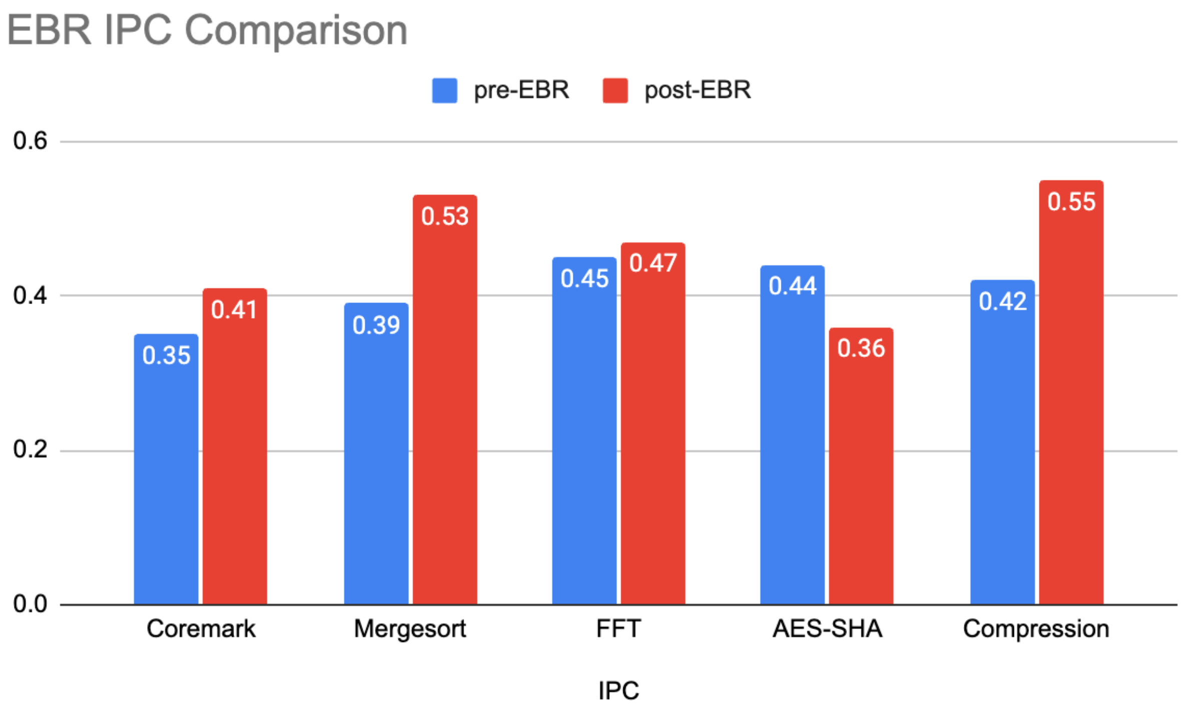
## Advanced Features

### Early Branch Recovery (EBR)

The first advanced feature we wanted to implement was early branch recovery (EBR) because we noticed that the ROB took about 33.74 cycles to commit the next instruction whenever a flush occurred. When analyzing performance, we realized how significant the penalty of flushing was. As a result, implementing EBR would significantly increase our performance by removing the 33.74-cycle stall.

We began designing EBR by creating a branch stack to store all the necessary information to recover the subsystems, such as the free list, RAT, and ROB. When an instruction reached the decode stage and was determined to be a control instruction, we captured snapshots of the various subsystems and saved them in the first available position in the branch stack. Specifically, we stored a copy of the full RAT list, the head of the free list, and the head of the ROB in the branch stack. Additionally, we created a flush mask and a taken mask to track instruction dependencies. Both mask sizes were based on the size of the branch stack. For example, if a branch instruction occupied the first entry of a branch stack with a capacity of four entries, the instruction following the control instruction would have a flush mask of 0001 in its data structure to indicate dependency on the branch stack result of the first entry. Similarly, an instruction dependent on two branch prediction results (e.g., the first and third entries of the branch stack) could have a mask of 0101. This masking occurred because it was added to the first free entry in the branch stack whenever a control instruction was encountered. Once every instruction had a mask, the mask was used to determine which instructions would be flushed by sending out two possible signals. If the branch result was incorrectly predicted, a flush signal and a flush mask were sent to clear instructions dependent on this control instruction. For instance, if a flush occurred for a branch instruction in the first entry of the branch stack, a flush mask of 0001 would be sent and bitwise AND-ed with every instruction's flush mask. If the result was non-zero, the instruction was determined to depend on the mispredicted branch and was flushed. Additionally, the ROB, free list, and RAT would be restored using the snapshot saved during the flush. Conversely, if the branch was correctly predicted, a taken signal and a taken mask were sent instead. The taken mask indicated the position in the mask. For example, if the first entry of the branch stack was correctly predicted, it sent a 0 to each instruction, zeroing out the 0th position of their masks and indicating that those instructions were no longer dependent on an uncertain control instruction result.

In the figure below, you can see how, after adding EBR, our IPC increased for almost all of the test programs, demonstrating that there was less execution stalling present in the processor. Additionally, the flush stalling decreased from an average of 33.74 cycles to only 0.4 cycles, which is a significant performance boost. However, the tradeoff of EBR is that it is only beneficial if the program has plenty of flushes that occur. This would allow the processor to recover all of its subsystems to the necessary state quickly. However, if the program doesn't have many flushes, EBR wouldn't demonstrate any benefits and instead would consume power as it needs to take care of the taken mask.

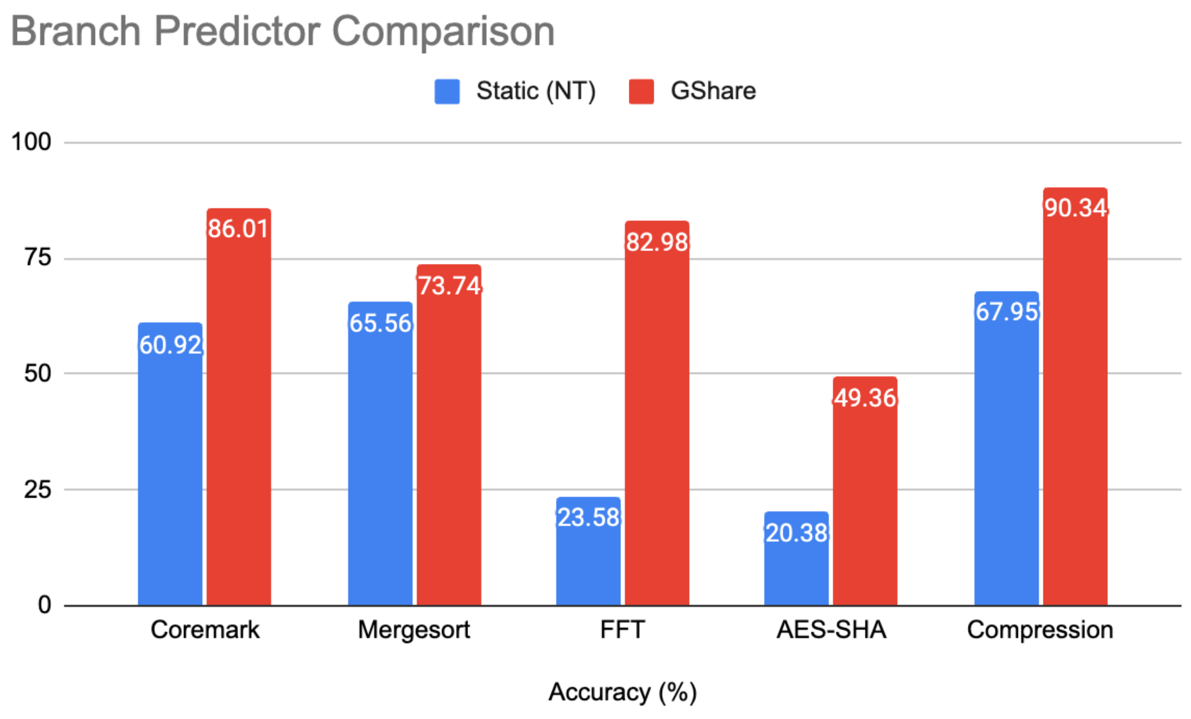


## Next-Line Prefetcher

We implemented a Next Line prefetcher. Our prefetcher was based on a state machine. We have four states for our preference. We start out in the empty phase, which denotes that we have nothing relevant in the prefetcher. On the first request, we go to the fetching phase. In the fetching phase, we directly take the memory access we got and route it to the icache. On the response, which will contain 256 bits or 8 instructions, we create what is essentially an L1 cache by storing the result in a buffer along with the associated address, 8-byte aligned. After this, we go to the prefetching stage. In the prefetching stage, we send the next line to the cache but do not read it. After this, we entered the stable stage in which we serviced item requests directly from our buffer. Any miss would result in a transition to the fetching stage. Through this, we could increase performance by decreasing power while keeping the area relatively constant. We also had possible timing benefits due to the creation of a buffer, which reduced the length of the path to the icache. For AES, we observed that our IPC went up by 0.1 with the usage of the prefetcher, while for coremark, it caused a decrease of 0.06, which we thought was a fine trade off.

## Branch Predictor

For branch prediction, we implemented gShare using sram like in mp_cache and needed to use very similar principles. One thing that we did was feed the input directly into the SRAM and wait for a cycle. This cycle wait was always there due to the reliance on the icache and also gave us time to get the relevant data from the SRAM. We then took this data, along with the icache rdata, and made a prediction on the next pc address. The calculation for branches and JAL was done in the fetch stage due to the address being known, while for JALR, we used a btb entry that we retrieved at the same time as the prediction. Using our branch predictor, we increased the number of instructions per cycle due to use having a higher accuracy on branches. Our accuracy is still possibly a bit low and is an area to improve. We noticed that somehow, the number of flushes that our processor did only went down by 300 after adding gShare, so it is possible that we are not measuring something correctly or predicting inaccurately.



## Parameterizable Cache

We implemented a parameterized cache to give us more flexibility in adjusting our area and power. We did this using a parameterized PLRU module. This module used recursive instantiation of itself in order to determine what the next way to evict would be along with the next thing to write to the PLRU would be. This allowed us to parameterize the ways easily. Due to our reliance on types in our cache, we ended up using an interface to communicate between the two stages of our cache and the top level that put them together. This interface contained critical information such as the request data, such as the address. It also allowed us to have minimal changes to the existing code that relied on the types contained within the interface (we ported types used in cache to the interface). Another thing we did was make the number of bytes that could be output by the cache when making a request parameterizable. This made it very easy to implement our next line prefetcher and take it out if we compared performance with and without it.

## Design Space Exploration Scripts

To aid us in finding the most performant parameters for our design (e.g., # of ROB entries, Branch Stack Size, etc.), we wrote a design space exploration script in Python. Given a configurable design space, the script finds the power, area, and coremark latency and calculates the competition score for each configuration. Then, it simply sorts in reverse order and outputs it. An example of a configuration and result is shown below:

```
# Parameters to explore
parameters = {
    'RS_ENTRIES_ALU': [2, 4],
    'RS_ENTRIES_MD': [2, 4],
    # Add more parameters as needed
}
```

*An Example Design Space Setup*

```
$ python3 explore.py
1/4: Running with {'RS_ENTRIES_ALU': 2, 'RS_ENTRIES_MD': 2}
2/4: Running with {'RS_ENTRIES_ALU': 2, 'RS_ENTRIES_MD': 4}
3/4: Running with {'RS_ENTRIES_ALU': 4, 'RS_ENTRIES_MD': 2}
4/4: Running with {'RS_ENTRIES_ALU': 4, 'RS_ENTRIES_MD': 4}

------ exploration results (from best to worst) ------
Parameters: {'RS_ENTRIES_ALU': 2, 'RS_ENTRIES_MD': 2}
Area: 0, Power: 0, IPC: 0.176471, Latency: 0.073, Competition Score: 0.073

Parameters: {'RS_ENTRIES_ALU': 2, 'RS_ENTRIES_MD': 4}
Area: 0, Power: 0, IPC: 0.176471, Latency: 0.073, Competition Score: 0.073

Parameters: {'RS_ENTRIES_ALU': 4, 'RS_ENTRIES_MD': 2}
Area: 0, Power: 0, IPC: 0.176471, Latency: 0.073, Competition Score: 0.073

Parameters: {'RS_ENTRIES_ALU': 4, 'RS_ENTRIES_MD': 4}
Area: 0, Power: 0, IPC: 0.176471, Latency: 0.073, Competition Score: 0.073
```

*An Example Design Space Exploration Output*

## Additional Observations

In hindsight, after seeing the competition result, we were surprised to learn that a team could beat us even though they hadn't implemented EBR. It was clear why, though: their area was incredibly low. They had squeezed every last IPC out of their area. This was a learning experience for us: it's not just about implementing advanced features. It’s about doing them in an area-efficient way. With this in mind, we would optimize as much as possible (remove excess area) in CP2 and CP3 and make our baseline processor as lightweight as possible before diving deep into advanced features.

For our design space exploration script, we were limited by how much we could explore due to our dependency on running a benchmark with VCS. Comparatively, a run with Verilator takes a couple of seconds on Coremark to run, while it takes VCS >10 minutes. As a result, even though we had at least ten different design parameters we wanted to search over exhaustively, we could not do a full grid search on all the possible configurations. In the future, we wish to improve the speed of our search by 1) parallelizing the workload (launching separate processes for each configuration) and 2) relying on Verilator instead of VCS to do the simulations.

Another aspect of our process we could have looked into was memory operation optimizations. We didn’t implement memory operations optimizations, meaning we still used partial ordering. Therefore, we still needed to wait until the store was at the head of the rob before we would send the data to the cache and the burst memory, if necessary. Subsequently, load instructions behind the stores would have to wait as well. This would be a big performance issue, especially if there are many data cache misses.

Our last bottleneck was that our instruction queue was constantly empty. We couldn’t get to the bottom of this issue as we ran out of time. We have two speculations: either flushing is happening too much, meaning that our branch predictor isn’t as accurate as we would like, or the prefetch is doing something incorrectly. The inability to fetch instructions quickly was quite an issue because we were stuck at our current IPC of around 0.5 for coremark for a long time. Even after what we thought were fixes, the instruction queue continued to be empty many times.

## Conclusion

Our primary objective was to design and implement a fully functional explicit register renaming processor. Building on this foundation, we introduced advanced features such as early branch recovery, a GShare predictor with BTB, parameterized caches, and next-line prefetching. Then, we focused on optimizing the processor by improving IPC without significantly increasing area or power consumption. This project highlighted both the challenges and rewards of CPU design. We experienced the excitement of transforming theoretical concepts into functional implementations and the frustration of seeing some features fail to deliver the expected performance gains. We learned that effective processor optimization requires careful trade-offs, from selecting the right combination of features to balancing complexity with benefits to ensuring efficient implementation. This experience highlighted the intricate art of CPU design, where every decision, from architecture to coding, contributes to creating an efficient and high-performing processor.