mp_ooo

Introduction

This report details the design and implementation of our out-of-order (OOO) RISC-V processor. Our implementation of an OOO processor represents a significant undertaking in advancing our understanding of computer architecture. This project integrates theoretical principles learned throughout the semester with practical design challenges to develop a fully functional, efficient processor capable of exploiting instruction-level parallelism for improved performance. This processor adheres to the RV32I instruction set with the “M” extension for integer multiplication and division. It also incorporates advanced features like branch prediction, cache optimizations, and age-order scheduling, making it a comprehensive exercise for us in computer architecture. In short, the motivation behind this work is to bridge the gap between theoretical knowledge and real-world implementation. Completing this project required mastering various essential concepts like pipelining, memory organization, and speculative execution, providing a holistic view of processor design.

Project Overview

At its core, the processor's architecture is built on an ERR-style microarchitecture, emphasizing modularity and extensibility. Key components include the Free List, RAT (Register Alias Table), ROB (Reorder Buffer), RRF (Retirement Register File), CDB (Common Data Bus), and reservation stations, alongside standard von Neumann architecture modules such as fetch and decode. Advanced features, including a next-line prefetcher, branch prediction, age-order scheduling, and a return address stack, further enhance the processor's performance and efficiency.

This report is organized into several sections. We begin with an overview of key milestones achieved during the project, detailing the development of individual modules and their integration into the processor. Next, we dive deeper into discussion of the advanced features implemented to optimize performance. Finally, we conclude with lessons learned and potential future enhancements to the design. Through this report, we aim to share the insights and challenges encountered in building a high-performance OoO RISC-V processor.

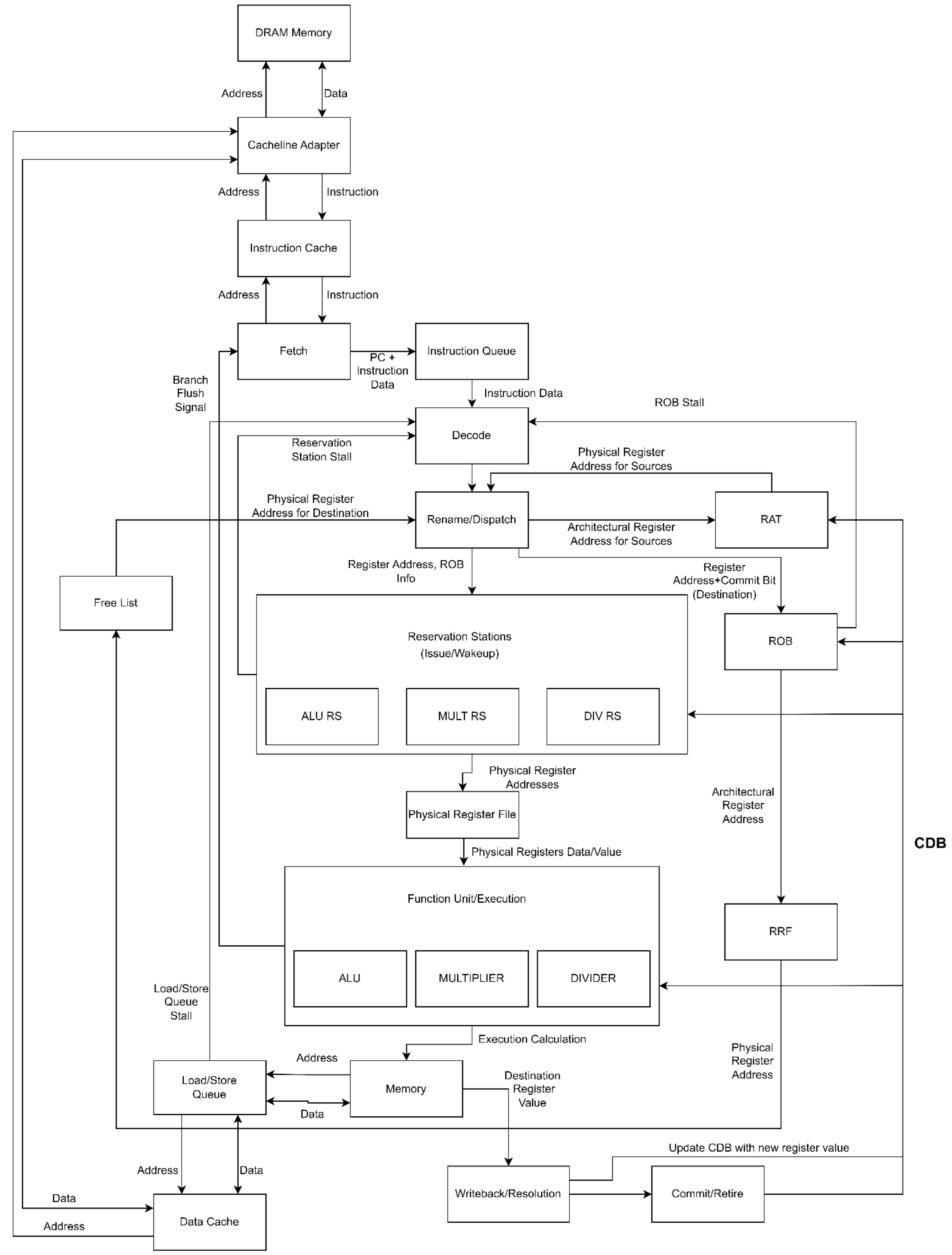
Design Description

Milestones

Checkpoint 1

For the first checkpoint, we made several design decisions, the most significant being the adoption of an ERR-style microarchitecture. The primary feature implemented was the fetch module, which we integrated with a cacheline adapter and an instruction queue. The cacheline adapter deserializes bursts from the DRAM model, while the instruction queue is a parameterizable design we developed. This parameterizable queue will also be utilized in other processor components, such as the ROB and the free list. To test these, we developed targeted testbenches to evaluate specific components of the fetch module. Initially, we thoroughly tested the parameterizable queue and the cacheline adapter individually. Subsequently, we used additional testbenches to test the fetch module with both components integrated.

Design Block Diagram



Checkpoint 2

By the end of checkpoint 2, our processor was able to execute all immediate and register instructions in the RV32I spec. This included a multiplier and divider. We created the rest of the components of the ERR processor (excluding the load store queue). We added: a decode stage that parses the instruction, the free list that stores all readily available physical registers, the RAT that holds the architectural to physical register mappings, reservation station to queue functional unit operations, the physical register file, the re-order buffer to commit instructions in order, the functional units (ALU, multiplier, divider), CDBs to drive the functional unit outputs, and an RRF that had the register mapping of committed instructions. In the initial state, architectural register x1 is mapped to physical register x1, and so on. The remaining registers (x32 to x63) start in the free list. The decode stage uses the RAT to find physical source registers and dequeues from the free list to get a physical destination register. It adds to the ROB and saves the entry. It queues the operation to the reservation station - 2 stations for each functional unit. When both source registers are valid, the operation is sent to the functional unit, and the operation is performed and then driven by the CDB. The bus writes the value to the physical destination register and makes the instruction ready to commit (in the ROB). The RRF sees if the top of the ROB is ready to commit - if yes, then update RRF, push to the free list, and dequeue from ROB.

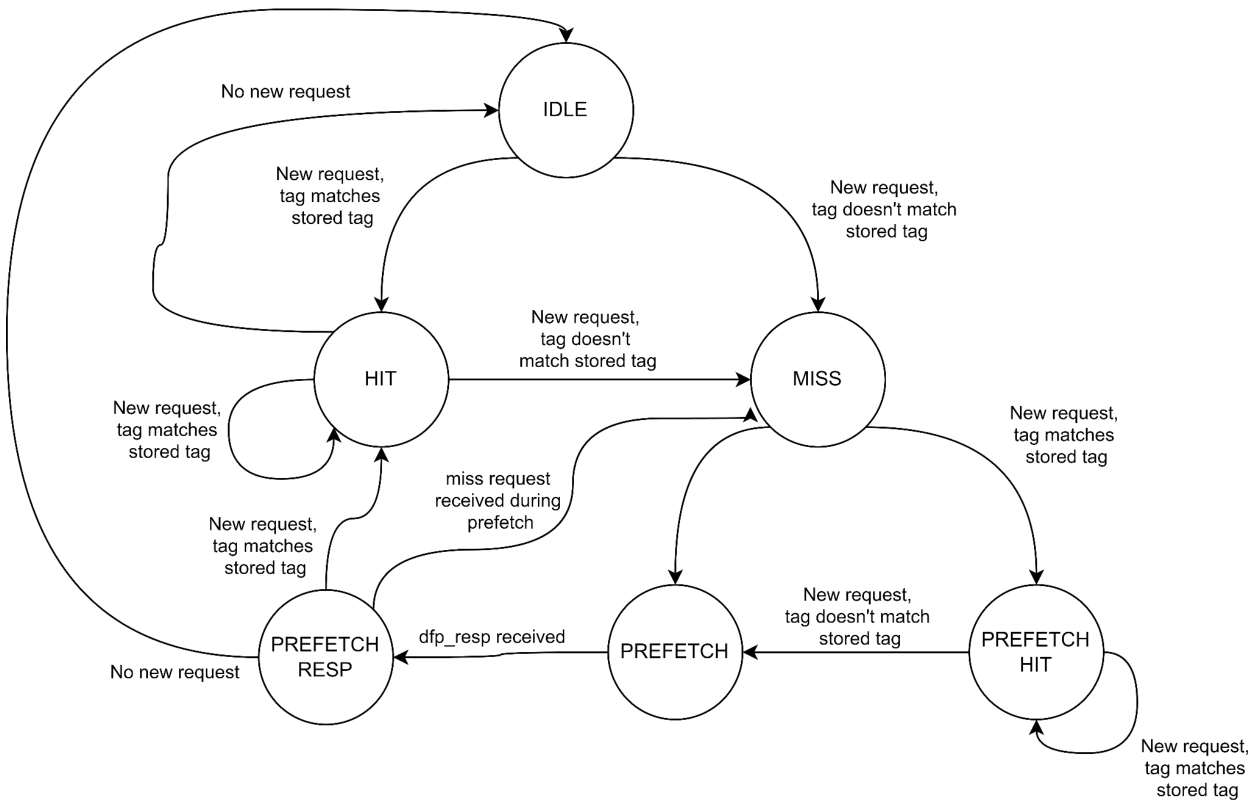
Checkpoint 3

For checkpoint 3, we added all the remaining components to create the processor and execute the entirety of RV32IM spec. We integrated our own cache and added cacheline adapter logic to select between dcache and icache dfp request. We added a load-store queue to support memory operations and also added logic to support branching. The load store queue is parallel to the reservation station. During decode, if the instruction is a load or a store, it is sent to the LSQ instead of the reservation station. If all source registers are valid, the operation at the top of the queue is performed. Our stores are non-speculative, so we only perform the store operation if it is on the top of the ROB. Once the dcache request is processed, similar to functional unit CDBs, there is another CDB for the LSQ. It makes the corresponding ROB entry ready to commit and writes the value in the physical register. For branches, it goes through the reservation station and ALU. We started with static not-taken to simplify the implementation. No additional components were added to accommodate flushing but the existing components were modified. When the branch instruction reaches the functional unit, it is resolved and if the branch needs to be taken, then the processor is flushed. On a flush, the RAT gets a copy of the RRF, the free list is marked as full, the reservation stations and load store queue are emptied, and the instruction queue and the ROB are also emptied. In addition, if icache or dcache requests were made prior to the flush, the response is masked.

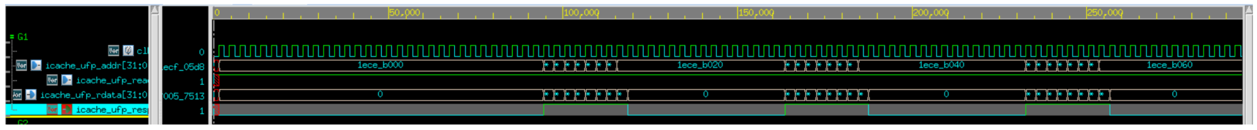
Advanced Design Features

Next-Line Prefetcher

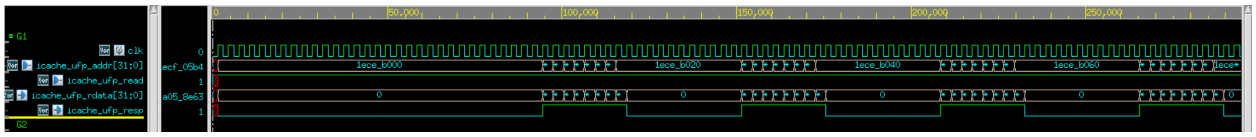
When the fetch module fetches an instruction from memory it hasn't fetched before, the i-cache labels that request as a miss and loads the corresponding data line into the cache, occupying one cache line. If fetch requests consecutive instructions in memory, the next seven fetch requests would be hits. During this time, the i-cache is idle. After the seventh fetch, the next memory request would be a miss, and the i-cache would start retrieving the next cache line—something it could have started immediately after servicing the first miss. To address this, we created an extra module between the CPU and the i-cache called the icache_prefetch. After a miss, the icache_prefetch module saves the complete 256-bit cache line and services hit requests from the CPU. During this time, the module starts fetching the next cache line from memory into i-cache. To achieve this, the module uses a specific finite state machine.


IDLE
No new request
New request, tag matches stored tag
New request, tag doesn't match stored tag
HIT
New request, tag doesn't match stored tag
MISS
New request, tag matches stored tag
miss request received during prefetch
New request, tag matches stored tag
New request, tag matches stored tag
PREFETCH RESP
dfp_resp received
PREFETCH
New request, tag doesn't match stored tag
PREFETCH HIT
No new request
New request, tag matches stored tag

Without Next-Line Prefetch



With Next-Line Prefetcher:

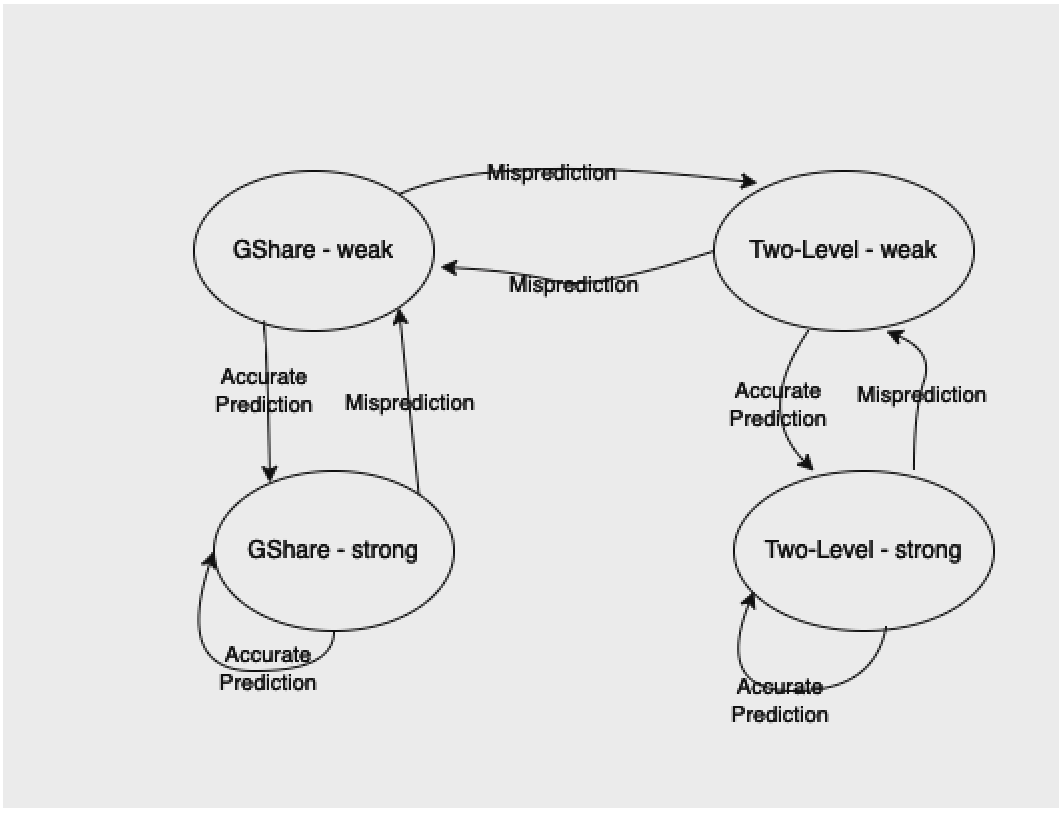


Branch Prediction: GShare, Two-Level, Tournament, BTB

The motivation for adding a branch predictor was to improve the prediction accuracy (compared to static not-taken). When a misprediction occurs, the entire processor flushes and starts fetching instructions from the calculated PC. This requires a lot of resources and waste cycles.
First, GShare is a branch predictor incorporating a global history register with a two-bit saturated counter. The global history register (GHR) stores the branching pattern (taken or not taken) for the last 16 branches. When a branch is resolved (taken or not taken), a bit is left-shifted into the global history register (1 if taken and 0 otherwise). The GHR is used along with the PC to index into a history table; GHR ⊕ PC[3:0] is the hashmap/indexing for the table. Each entry in the table consists of a two-bit saturating counter that determines whether to take the branch or not. Since the history table (containing two-bit counters) is an SRAM, it takes a cycle to read the data. So, to ensure that no additional cycles are required, the SRAM history table and instruction are accessed simultaneously (using PC_next). The prediction, along with the instruction and PC, is enqueued. When dequeued in the decode/dispatch stage, the prediction is used to determine the next PC. If the branch is taken, the new PC address is fetched and the instruction queue is cleared. When the branch instruction reaches execute, it is resolved and if there is misprediction, a flush occurs**.** Compared to static not-taken, which has 46.5% accuracy on coremark, GShare improved the prediction accuracy to 75%. The tradeoffs are greater area due to SRAM, connections, and prediction logic. But since it only has 16 entries, it is negligible compared to the caches. It works best with workloads with nested loops that create a specific branching pattern (for example T, T, T, NT). When the GHR gets a pattern T, T, T, it will predict NT due to the pattern. Aliasing is a problem that can occur in GShare, caused by many PCs sharing the same table index.

Another type of predictor is a two-level predictor. The first level is a local history table (LHT) and the second level is a pattern history table (PHT). In contrast to Gshare, this predictor has two levels of indexing. There are 16 entries in each table; each entry in the LHT stores the branching pattern (similar to GHR) and the pattern is used to access the PHT entry that stores the corresponding two-bit counter. LHT and PHT are implemented using SRAMs, so it takes two cycles to access the PHT entry since it is two levels; one cycle to read the LHT entry, and another cycle to read the PHT entry. Similar to GShare, the LHT is indexed using PC_next, but when the instruction is ready, there is a stall for an extra cycle to access the prediction from PHT. Once the prediction is ready (one cycle after the data cache response), the prediction is enqueued along with the instruction and PC. In addition, along with the prediction, even the pattern is saved. The decode logic is the same as GShare, but when the branch is resolved (at execute), based on the previous pattern and branch taken/not taken, the new pattern and state are written to the LHT and PHT respectively. The prediction accuracy, compared to static not-taken, increased to 56%. This number could be improved by modifying the implementation, but there would be an area tradeoff. The main tradeoff is the area of the SRAMs along with the flip-flops used to propagate states and patterns. Similar to GShare, this would work well on workloads with nested loops or repetitive branch patterns since repetitive patterns will access the same two-bit saturated counter.

To improve the accuracy of the prediction and take advantage of both predictors, we implemented a tournament between GShare and a two-level branch predictor. The tournament outputs 0 to choose GShare prediction and 1 to choose two-level prediction.



Based on the tournament output, either the state from two-level or gshare is enqueued to the instruction queue along with the tournament output. The decode and execute logic stays the same. On a misprediction, the tournament state also gets updated based on its previous state. This state machine works independently from the two predictors. The IPC improved and the prediction accuracy increased to 67%. Adding tournaments allowed a more dynamic selection of predictors, making it accurate for all types of problems.

The BTB is an SRAM that stores the predicted addresses and it has 16 entries. On decode, when predicted to take the branch, the PC bits are used to index into the BTB and that becomes

PC_next. The PC_next is sent to fetch to start fetching new instructions. When the branch is resolved, if the predicted PC is incorrect, the BTB is updated with the correct address, and the processor is flushed. Since BTB is also an SRAM, PC next is used to index into the BTB. The performance worsened significantly after adding this since another prediction needs to be made (leading to more flushes). It adds to the area, as well as logic for the states.

For all SRAM modules that have states, I added a valid array that determined if the contents of the entry were valid or not (X's). If valid was false for a specific index a default value was sent, otherwise, the value from the SRAM is sent.

Split-LSQ

A Split Load/Store Queue (LSQ) is a design enhancement that separates the load and store queues, improving parallelism and reducing contention. In our implementation, these two queues are connected to a Split-LSQ Arbiter, which sends the final requests to the D-Cache. This baseline Split-LSQ enables loads to execute out of order with respect to stores.

To further enhance performance, we introduced two key improvements. The first was allowing loads to issue out of order with respect to stores targeting different addresses, implemented using age ordering. This optimization improved the efficiency of memory operations by leveraging the inherent parallelism of unrelated memory accesses. The second and most impactful improvement was non-committed store forwarding. This feature allowed load requests to use values from stores that had not yet been committed, significantly reducing D-Cache accesses. We measured the impact of this optimization using a performance counter, which showed 272 instances of store-forwarded values for load requests in the Mergesort benchmark. This reduction in D-Cache accesses saved many cycles and resulted in a substantial increase in Instructions Per Cycle (IPC).

To implement these features, relevant information from the store queue, such as store validity, PS1 and PS2 validity, age, and store address, was sent to the load queue. These values were compared with the corresponding load values to decide whether a store or load should execute. The key logic, age_order_satisfied, ensured that loads executed only when safe.

Our performance analysis showed IPC improvements across all benchmarks, with the Mergesort benchmark—characterized by frequent loads and stores—exhibiting the most significant gains. However, these enhancements came with trade-offs. The critical path of the processor was extended, running from decode to reservation stations, to execute, to the physical register file, and ultimately to the load queue and D-Cache. The design also introduced significant area overhead, as the split LSQ required doubling the queues and adding 12 ports of length 32 and 12 ports of length 6, further increasing area usage.

This feature is particularly beneficial for memory-intensive applications such as matrix multiplication, graph traversal, and database queries, where parallelized load/store handling boosts performance. However, compute-bound workloads with minimal memory accesses derive little benefit, incurring unnecessary area and power overhead. Similarly, in load-heavy workloads like streaming data analytics, the load queue may become a bottleneck while the store queue remains underutilized. Overall, the Split LSQ and its enhancements provide substantial performance benefits for memory-intensive tasks but require careful consideration of their design trade-offs.

Return Address Stack

In function calls, return addresses are stored in a register, with the target address for the jump remaining constant throughout the function's execution. This jump is unconditional. In our baseline processor, a RET instruction is treated as a regular instruction, causing a flush when it reaches the top of the ROB, and fetch is updated to begin fetching instructions from the target address. We optimized this by introducing a stack in the decode module to handle return addresses. At a function call, the target address is pushed onto the stack, and at a function return, it is popped and sent to fetch to preemptively start fetching the correct instructions. As a result, the instruction eventually commits without requiring a flush, as the correct instructions are already in the pipeline. This optimization improved Coremark IPC by 4%.

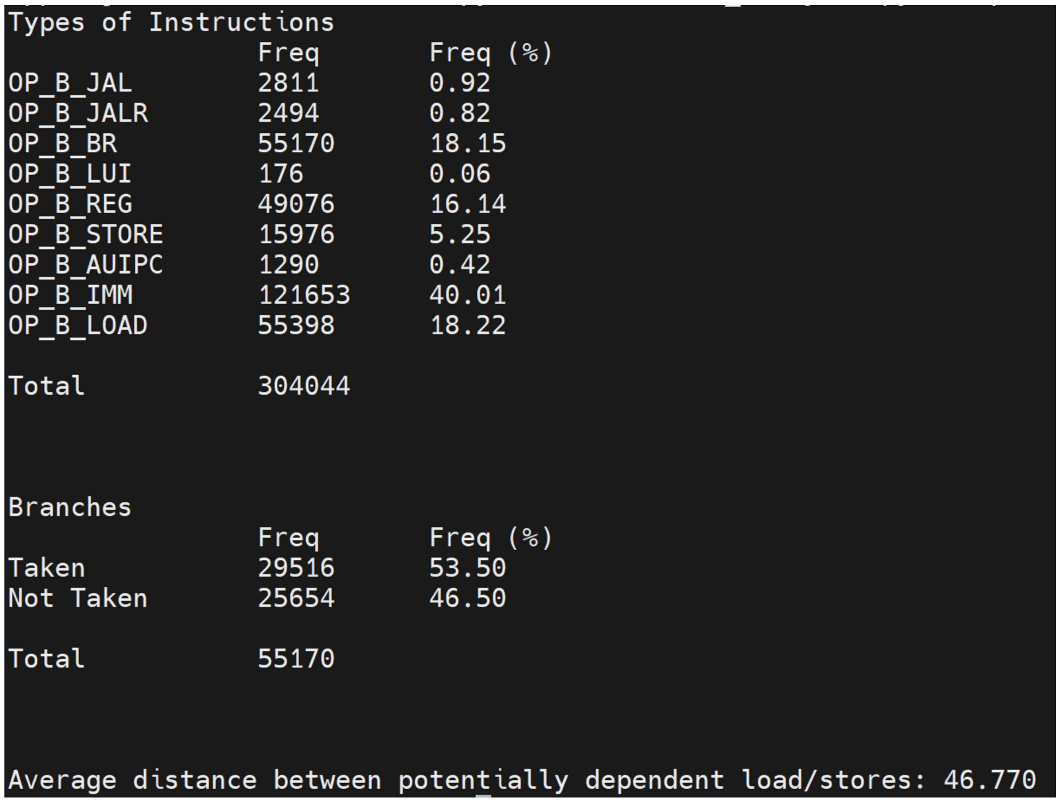
Age-Order Scheduling

When multiple reservation station entries are ready to be executed, the oldest one is sent first. To implement this, we added an age entry to each station. On every enqueue to the reservation station, the age was incremented and reset to 0 on a flush. When there is a conflict, a comparator is used to compare the ages, and the entry with the smallest age is sent. In the mergesort_im testcase, there were 65k conflicts where the older one was chosen. This improves performance as it commits the instructions that were waiting for longer first, allowing new instructions to be processed. The tradeoff was increased critical path due to the addition of comparators.

Benchmark Analysis

When adding advanced features, it is crucial to measure performance improvements accurately and ensure the feature works across all benchmarks. If performance degrades for a specific benchmark, it must be verified that the degradation is not due to an implementation error and that the overall performance gain justifies the trade-off. To facilitate benchmark analysis, we developed a Python script to analyze commit logs generated by Spike and generate a report with key performance statistics.

Sample report for Coremark:



```
Types of Instructions
                Freq       Freq (%)
OP_B_JAL        2811       0.92
OP_B_JALR       2494       0.82
OP_B_BR         55170      18.15
OP_B_LUI        176        0.06
OP_B_REG        49076      16.14
OP_B_STORE      15976      5.25
OP_B_AUIPC      1290       0.42
OP_B_IMM        121653     40.01
OP_B_LOAD       55398      18.22

Total           304044



Branches
                Freq       Freq (%)
Taken           29516      53.50
Not Taken       25654      46.50

Total           55170



Average distance between potentially dependent load/stores: 46.770
```

Additional Observations

These are some incidents/bugs/bottlenecks/observations from our project:

- **Performance Comparison Challenges**:
  Many of our speedup times and comparisons for advanced feature improvements relative to the baseline processor are incomplete because we initially utilized the provided direct-mapped cache. During development, we concurrently implemented a pipelined cache alongside other advanced features. This overlap means the observed IPC (instructions per cycle) improvements are influenced by both the advanced features and the significant performance gains from the pipelined cache. As a result, it is challenging to isolate and attribute IPC improvements solely to the advanced features.

- **RVFI Errors in Verdi**:
  While working with the provided cache for CP3, we encountered persistent RVFI errors in Verdi for our processor. After extensive debugging, we discovered that the errors were caused by a bug in the provided cache itself. Resolving this consumed significant debugging time early in the project.

- **Branch Target Buffer (BTB)**:
  We implemented a Branch Target Buffer (BTB) to enhance branch prediction. However, testing revealed that the BTB significantly degraded overall performance. As a result, we moved the BTB implementation to a separate branch for further analysis.

- **Automation with Bash Scripting**:
  To streamline testing and analysis, we developed a Bash script to automate the execution of all test cases and generate a comprehensive summary. The summary included pass/fail statuses, IPC changes, and delays, allowing for efficient debugging and performance tracking.

- **RSA Failures Due to Uninitialized Values**:
  RSA repeatedly failed across multiple commits due to the propagation of don't cares, or 'x's from the cache into memory. These undefined values eventually corrupted other parts of the system when accessing the same memory locations. We resolved this issue by setting bmem_0_on_x to true in the options.json file, ensuring proper handling of don't cares.

Conclusion

In conclusion, this project successfully designed and implemented an out-of-order RISC-V processor, adhering to the RV32I and RV32M specifications and integrating various advanced features such as branch prediction, next-line prefetching, and a split LSQ. Our objective was to explore the practical challenges of exploiting instruction-level parallelism while implementing a high-performance processor. We were able to successfully implement an ERR-style microarchitecture with robust modularity with enhancements to execution efficiency through non-committed store forwarding, and performance optimization using a next-line prefetcher. The incorporation of advanced branch prediction techniques, such as GShare and two-level predictors, alongside a tournament mechanism, demonstrated significant improvements in accuracy and IPC across benchmarks. Novel aspects, such as age-order scheduling and the implementation of the split LSQ, showcased our commitment to maximizing memory operation parallelism. Through this endeavor, we bridged the gap between theoretical understanding and real-world design, gaining invaluable insights into the complexities of modern processor architecture.