EECS 470 Final Project Report

Porvesh Balasubramanian, Coy Catrett, Lohit Kamatham,
Velu Manohar, Aidan Spizz, Nikhil Sridhar

Abstract

This project implements a MIPS R10K-style processor design based on the RISC-V Instruction Set Architecture (ISA), emphasizing scalability, modularity, and extensive, non-fragile unit tests. The processor is designed to achieve high Instruction Level Parallelism (ILP) while maintaining a balance between cycles per instruction (CPI) and synthesized clock period. Leveraging an N-way superscalar pipeline, the design integrates advanced features such as instruction prefetching, non-blocking data and instruction caches, and a victim cache to minimize cache miss penalties. Additionally, a Store Queue and Load Buffer were implemented to streamline memory access, reduce latency, and enhance throughput.

This report provides an in-depth overview of the architectural design, details of the components implemented, and the features integrated into the processor. It also includes a comprehensive analysis of the impact of these components on CPI and clock cycle time, demonstrating the effectiveness of the design in achieving both performance and architectural efficiency.

1 Introduction

In modern computing, the demand for high-performance processors capable of executing multiple instructions simultaneously has driven innovations in processor design. Instruction Level Parallelism plays a pivotal role in meeting these demands by enabling the concurrent execution of instructions within a single program.

This project focuses on the implementation of a MIPS R10K-style processor architecture based on the RISC-V Instruction Set Architecture. The processor incorporates advanced features, including an N-way superscalar pipeline, memory optimization techniques such as non-blocking caches, victim caches, and specialized memory structures like Store Queues and Load Buffers. These components work together to minimize execution stalls, optimize memory access patterns, and enhance overall performance.

The primary objective of this project is to design a processor that achieves high ILP while maintaining a low cycles-per-instruction (CPI) and a short synthesized clock period. This report outlines the architectural decisions made during the design process, detailing the implementation of key components and their integration into the final R10K-style processor. It also provides a comprehensive performance evaluation, highlighting the trade-offs between throughput, latency, and clock cycle time.

1.1 Project Background

The primary goal of this project is to design and implement a processor using SystemVerilog, building on foundational and advanced concepts discussed in the course EECS 470.

The project integrates theoretical knowledge of microarchitectures, including instruction-level parallelism (ILP), pipeline hazards, cache hierarchies into a practical design framework. Teams are tasked with creating a correct, out-of-order processor that adheres to the RISC-V ISA while meeting performance metrics, including low cycles per instruction (CPI) and a minimal synthesized clock period.

As part of the MDE, students gain hands-on experience in implementing sophisticated architectural features such as superscalar pipelines, branch prediction, memory optimization. Additionally, the course emphasizes the importance of hardware verification and testing, requiring students to develop comprehensive testbenches to ensure the functionality and reliability of their designs.

The design process is iterative, involving multiple stages of planning, development, and evaluation. Students must balance performance optimization with hardware complexity, making critical trade-offs between throughput, latency, and clock cycle time. The collaborative nature of the project simulates real-world engineering environments, preparing students for future roles in hardware design and computer architecture.

1.2 Design Overview

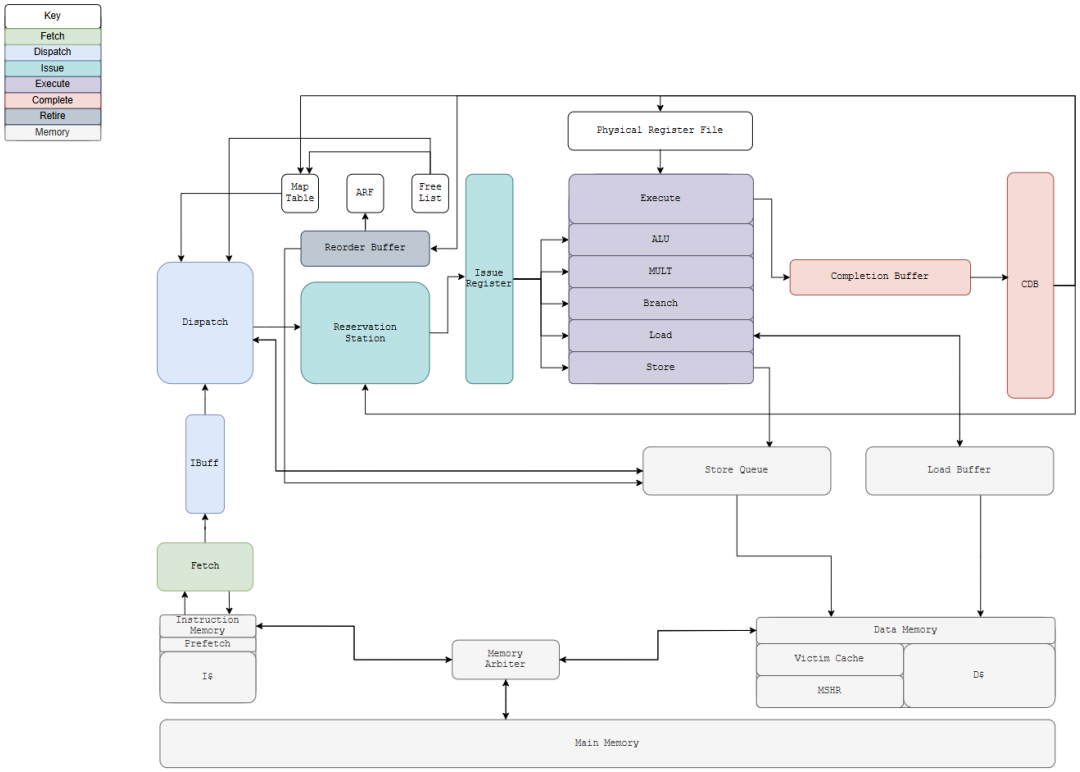

Figure 1: Architecture Diagram

Our design incorporates several advanced components and features to optimize performance, including structures for efficient memory handling, and speculative execution. This section provides a detailed overview of the key components in the processor design.

1.2.1 R10K Design Choice

The R10K algorithm is a widely regarded out-of-order execution model known for its simplicity and efficiency, making it an excellent choice for our processor design. Unlike the P6 algorithm, which requires the entire data value to be passed between various components such as the architectural register file, reorder buffer, reservation station, and functional units, the R10K algorithm only needs to propagate tags. This fundamental difference reduces the complexity of data movement within the processor, leading to a more streamlined and efficient design. The R10K's tag-based communication minimizes the need for large data buses and complex forwarding paths, thereby improving scalability and reducing the overall hardware overhead. Ultimately, our decision to implement the R10K algorithm stems from its balanced trade-off between performance and implementation complexity. The R10K's ability to handle out-of-order execution with minimal overhead, combined with its efficient tag-based communication, aligns with our goal of designing a high-performance processor that is both scalable and resource efficient. For these reasons, we believed that the R10K was ideal foundation for our architecture, ensuring robust performance while maintaining manageable design complexity.

1.2.2 Reservation Station

Our reservation station has 24 entries and dynamically schedules instructions for execution into an Issue Register. If there are no functional unit hazards and if there are no dependencies, then the reservation station will issue instructions into the issue register. The selection of instructions is categorized by the functional unit which the instruction uses (alu, mult, load, store, branch) and is arbitrated by a priority selector. In order to simplify the management of load and store instructions, our reservation station issues at most one load and it also issues at most one branch instructions.

1.2.3 Reorder Buffer

The Reorder Buffer is a circular queue used to maintain the program order of instructions. It ensures correct commit order by tracking instructions from dispatch to retirement. The ROB has a size of 32 entries, providing sufficient capacity to hold instructions in flight while balancing hardware complexity. This structure is critical for handling branch misdirection, as it allows the processor to roll back changes in case of branch mispredictions or exceptions, ensuring precise state recovery.

1.2.4 Store Queue

The Store Queue is a crucial component of the processor's memory subsystem, designed to handle store instructions efficiently while preserving memory consistency and correct program behavior. The challenge of maintaining the correct order of memory writes, particularly when store instructions are delayed or dependent on prior instructions, is resolved in the store queue by maintaining the program order of each of the stores

The Store Queue operates as a circular buffer with a size of 4 entries, which is optimized to balance hardware complexity and performance. Each entry in the Store Queue contains metadata about the store instruction, including the target memory address, the data to be stored, and any associated flags or status indicators. By keeping track of these details, the Store Queue allows the processor to handle multiple in-flight store instructions simultaneously, thereby improving throughput and reducing pipeline stalls. We did

not decide to implement the forwarding of data to dependent loads due to the complexity of the implementation. Even though this would have improved performance, our group was unable to implement it due to communication issues and tight deadlines.

The Store Queue interfaces with other memory structures, such as the Data Cache and Miss Status Holding Registers (MSHRs). Store instructions in the queue are typically committed to memory only after passing all dependency checks and once the corresponding instruction has been retired by the Reorder Buffer (ROB). This synchronization ensures that speculative stores are not prematurely written to memory, preserving the integrity of program execution.

Despite its benefits, the Store Queue introduces some challenges, particularly in workloads with high memory traffic or frequent store instructions. The fixed size of the queue may lead to contention or stalls when the queue becomes full, requiring careful management of store instructions to prevent bottlenecks.

In summary, the Store Queue is an essential component for achieving high memory throughput and maintaining correctness in a superscalar, out-of-order processor. By buffering store instructions, handling memory dependencies dynamically, and integrating with other memory subsystems, the Store Queue contributes significantly to the processor's overall performance and efficiency.

1.2.5 Load Buffer

The Load Buffer complements the Store Queue by managing load instructions and ensuring efficient access to memory. In a high-performance processor, load instructions are among the most frequent and latency-sensitive operations. The Load Buffer is designed to handle these instructions dynamically, reducing stalls and ensuring that the pipeline can continue executing instructions even when memory access delays occur.

The Load Buffer tracks pending load instructions, including their target memory addresses, the data to be fetched, and the status of the memory operation. Our pipeline only allows one load to be issued per cycle, however, multiple load instructions can be in-flight due to memory latencies and instruction dependencies. By maintaining this information, the Load Buffer enables the processor to manage multiple in-flight load instructions simultaneously, improving throughput and reducing idle time in the pipeline. While other modules in our processor are parametrizable, the load buffer is not due to our design decision to simplify stalling logic as much as possible. It must have 16 entries at all times due to 100 ns latency to the cache. Since our clock period is above 6 ns, then we will never have more than 15 in-flight load instructions. A size of 16 was decided upon for a micro-optimization as we are taking the modulus of the load buffer size, which is expensive in hardware if the modulus is not a power of 2.

In conclusion, the Load Buffer is a vital component of the processor's memory subsystem, enabling efficient handling of load instructions and ensuring data correctness and coherence. Together with the Store Queue, it forms the backbone of the processor's dynamic memory handling capabilities, supporting high ILP and contributing to the processor's overall performance and efficiency.

1.2.6 Free List

The Free List is responsible for tracking available physical registers in the processor. During instruction dispatch, it provides a pool of unused physical registers for allocation, ensuring efficient utilization of register resources. The size of the free list is the size of

reorder buffer plus the architectural register file plus the superscalar width in order to accommodate for all possible in-flight and incoming instructions and register renaming of those incoming instructions. In our final configuration, our reorder buffer size is 32, the number of architectural register files is 32, and the superscalar width is 2 resulting in a size of 66.

1.2.7 Architectural Register File

The Architectural Register File maintains the committed state of the processor, mapping architectural registers to the physical registers holding their most recent values. Unlike the speculative mappings in the Map Table, the Architectural Register File provides a stable reference for the processor's architectural state, ensuring correctness during exceptions, interrupts, and resets. This table enables the processor to recover its precise state efficiently, resuming execution seamlessly after disruptions. By serving as the definitive source of committed state, the Architectural Register File ensures robustness and reliability in the processor's operation.

1.2.8 Map Table

The Map Table is a fundamental part of the processor's register renaming process, dynamically mapping architectural registers to physical registers during instruction dispatch. By eliminating false dependencies such as Write-After-Write (WAW) and Write-After-Read (WAR) hazards, the Map Table allows instructions to execute out of program order while preserving correctness. In case of misprediction, the processor restores the map table setting it equal to the Architectural Register File. The Map Table also helps recycle physical registers by updating mappings upon instruction retirement, optimizing resource utilization.

1.2.9 Functional Units

The processor is equipped with multiple functional units to enable concurrent execution of instructions. It includes two Arithmetic Logic Units (ALUs) for general-purpose computations, one Arithmetic Logic Unit for branches, two units dedicated to multiplication operations, and one Load-Store Unit for memory access instructions. This configuration ensures sufficient resources to handle the high throughput demands of superscalar execution.

1.2.10 Physical Register File

The Physical Register File stores the values of physical registers, with a total size of 64 entries. This structure is designed to accommodate the high number of in-flight instructions supported by the processor, ensuring that data dependencies can be resolved efficiently without resource contention.

1.2.11 Instruction Cache (I Cache) and Prefetching

The processor features a single-ported, direct-mapped instruction cache designed to optimize instruction fetch bandwidth while maintaining simplicity in hardware implementation. A direct-mapped cache is chosen for its straightforward indexing mechanism, which reduces access latency compared to more complex associative cache designs. The

single-port architecture ensures that the cache can be accessed by one instruction at a time, striking a balance between hardware simplicity and performance. In the proposed processor design, the Instruction cache has 32 lines, each 64 bits wide.

A key feature of the instruction cache is its constant prefetching mechanism. Prefetching anticipates future instruction fetch requests by loading instructions into the cache ahead of time, based on the program's expected execution path. This proactive approach reduces cache miss penalties by ensuring that required instructions are readily available when needed. In addition, constant prefetching minimizes the latency incurred by instruction fetch operations, which is critical for maintaining high throughput in superscalar designs.

The benefits of this cache design extend beyond reduced miss rates. By decreasing the number of cache misses, prefetching also reduces the frequency of pipeline stalls caused by instruction fetch delays. This, in turn, enhances the efficiency of the instruction pipeline and contributes to the overall reduction in cycles per instruction (CPI).

While constant prefetching is effective in most scenarios, it has limitations, particularly in handling irregular instruction access patterns. For example, highly unpredictable branches or frequent jumps in the instruction stream may cause prefetching to load instructions that are not immediately needed, leading to a waste of cache bandwidth.

Overall, the instruction cache's design, augmented by constant prefetching, plays a pivotal role in achieving the processor's performance goals. It ensures efficient instruction delivery to the pipeline while maintaining a balance between hardware complexity and execution speed.

1.2.12 Data Cache (D Cache) and Miss Status Holding Registers (MSHRs)

The processor is equipped with a write-back, non-blocking Data Cache designed to efficiently handle memory read and write operations. This cache design prioritizes throughput by allowing multiple outstanding memory requests to be processed concurrently, a feature made possible by the inclusion of Miss Status Holding Registers (MSHRs). In the proposed processor design, the data cache has 32 lines, each 64 bits wide. The write-back policy ensures that modified data is written back to main memory only when it is evicted from the cache, reducing the frequency of costly memory write operations. This approach not only saves bandwidth but also enhances the processor's overall energy efficiency.

The non-blocking nature of the Data Cache allows it to service multiple memory requests simultaneously, significantly reducing stalls caused by cache misses. MSHRs play a critical role in this functionality by tracking pending memory accesses and coordinating their completion. Each MSHR maintains information about a specific cache miss, including the memory address, request type, and any dependent instructions or requests. By organizing and resolving these outstanding requests in parallel, MSHRs prevent the cache from becoming a bottleneck during memory-intensive workloads.

In summary, the Data Cache's write-back, non-blocking design, coupled with its use of MSHRs and optimization techniques, ensures that the processor can handle memory-intensive operations with high efficiency. It minimizes memory latency, reduces pipeline stalls, and supports the overall goal of achieving high Instruction Level Parallelism (ILP), making it a cornerstone of the processor's architecture.

1.2.13 Victim Cache

A Victim Cache is a small, fully associative cache placed between the primary data cache and the memory. It is designed to hold cache lines that are evicted from the data cache due to conflicts, thereby reducing the penalty associated with conflict misses. In the proposed processor design, the victim cache has 4 lines, each 64 bits wide, allowing it to store recently evicted data for potential reuse. This helps mitigate the impact of thrashing in the primary cache by temporarily retaining data that might otherwise require costly refetches, improving overall cache performance and reducing memory latency.

1.2.14 Branch Prediction: Gshare

Branch prediction is critical for maintaining high throughput in pipelined processors. Our branch prediction mechanism integrates three main components: the Branch History Register (BHR), the Pattern History Table (PHT), and the Branch Target Buffer (BTB). Together, these components implement a 4-bit Gshare predictor with a direct-mapped BTB, offering a balance of accuracy, speed, and efficiency.

Branch History Register (BHR) The BHR is a register that records the outcomes of the most recent branches as a sequence of bits:

- Each bit represents whether a branch was taken (1) or not-taken (0).
- In our design, the BHR is 4 bits wide, enabling it to capture the outcomes of the last four branches.

The BHR is critical for identifying global patterns in branch behavior. For instance, certain programs exhibit correlated branching behavior where the outcome of one branch depends on previous branches. By maintaining a history of outcomes, the BHR enables the predictor to detect such correlations and make more accurate predictions.

Pattern History Table (PHT) The PHT is a table of 2-bit saturating counters used to predict the likelihood of a branch being taken or not-taken. The PHT works as follows:

- The index into the PHT is derived by XORing the BHR with the lower bits of the branch's program counter (PC). This indexing strategy, unique to Gshare, helps distribute branches across the table, reducing aliasing.
- Each entry in the PHT is a 2-bit counter that adjusts based on the actual branch outcomes:
  - If the branch is taken, the counter increments (up to a maximum value of 3).
  - If the branch is not-taken, the counter decrements (down to a minimum value of 0).
- Predictions are made based on the most significant bit (MSB) of the counter:
  - 1: Predict taken.
  - 0: Predict not-taken.

The PHT complements the BHR by storing and refining the predictions based on historical data.

Branch Target Buffer (BTB) The BTB is a cache-like structure that stores the target addresses of branches predicted as taken. Its primary role is to ensure that when a branch is predicted as taken, the processor can immediately fetch instructions from the target address, reducing stalls caused by control hazards. Our BTB is direct-mapped, meaning each branch maps to a single location in the table.

Why Gshare? We chose the Gshare predictor for its ability to balance accuracy, complexity, and efficiency:

- Global History Utilization: By using the BHR, Gshare incorporates global branching patterns, making it particularly effective for workloads with correlated branches.
- Reduced Aliasing: The XOR operation between the BHR and the PC address helps distribute branches more evenly across the PHT, mitigating aliasing compared to simpler predictors like bimodal.
- Compact and Fast: Gshare achieves high accuracy without the complexity or resource requirements of more advanced predictors.

Performance Benefits

- Accurate Outcome Prediction: Gshare's use of global history improves the prediction of correlated branches.
- Quick Target Resolution: The BTB allows immediate fetching from the correct target address, minimizing control hazards.
- Flexibility: The 4-bit BHR and associated PHT can be scaled as needed, making this design versatile across different applications.

The combination of the 4-bit BHR, PHT, and direct-mapped BTB in a Gshare predictor provides a robust solution to the challenges posed by control hazards. Gshare's ability to leverage global history, coupled with the low-latency target resolution of the BTB, enables high prediction accuracy. This design ensures our processor can sustain high instruction throughput, even in branch-heavy workloads, while maintaining scalability and efficiency.

2 Visual Debugger

The visual debugger was developed as a full-stack web application, combining a Python and Flask backend with a TypeScript and React frontend. This design was chosen due to the ease of working with Python libraries for parsing VCD files generated during synthesis. The React frontend was selected for its precise control over frame rendering, while TypeScript was used to enhance maintainability and ensure type safety.

The backend is highly modular, well-documented, and designed for easy modification. It retains the hierarchy of the simulated program, enabling users to visualize waveforms and obtain an automated, precise view of the simulation state and its corresponding modules. This functionality provides an intuitive way to debug and understand the processor's behavior.

Several tools and approaches were identified for future enhancements. A notable option is a Node.js parser, which could enable packaging the web application in Electron for improved portability. Additionally, advanced parsers and linters, such as Slang, could provide deeper insights and richer information for the frontend. ANTLR4, a parser generator that processes language grammar, offers potential for creating customized parsers, further expanding the debugger's capabilities.

3 Advanced Features

Table 1: Advanced Features of the Processor Design

Feature	**Description**
Superscalar Execution	N-way superscalar execution to enable multiple instructions to be issued and executed simultaneously.
GUI Debugger	A functional graphical user interface (GUI) for streamlined debugging and analysis.
Unit Tests	Well-made unit tests with excellent coverage of individual modules, ensuring robust verification.
Instruction Prefetching	Proactive fetching of instructions to reduce fetch latency and mitigate pipeline stalls.
Non-blocking L1 Data Cache	A write-back, non-blocking L1 data cache to handle multiple memory requests simultaneously.
Victim Cache	An additional cache layer to reduce penalties caused by conflict misses in the primary cache.
Sophisticated Branch Predictors	A 4-bit Gshare predictor with a direct-mapped BTB and PHT for efficient branch handling.

4 Analysis

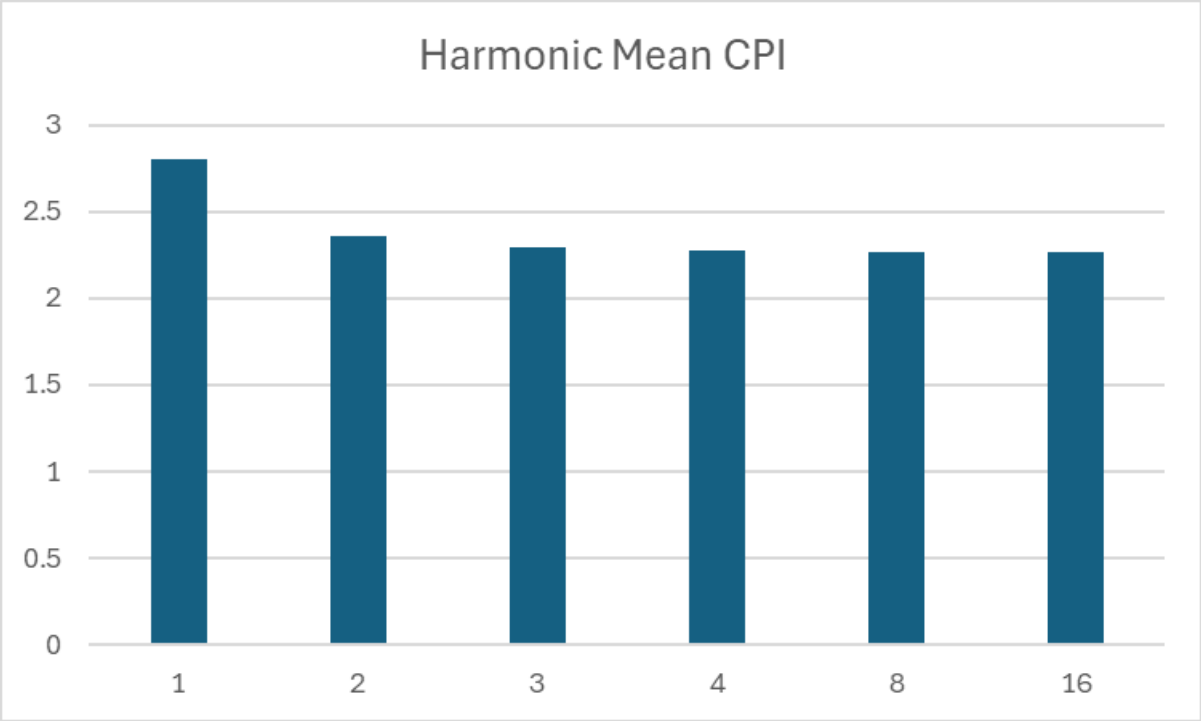

Figure 2: Harmonic Mean CPI

The graph illustrates the Harmonic Mean CPI (Cycles Per Instruction) for varying levels of parallelism or configurations. As observed, the CPI consistently decreases as the configuration progresses from 1 to higher levels such as 2, 3, 4, 8, and 16. This indicates improved processor efficiency as parallelism increases, allowing instructions to be executed in fewer cycles. However, the rate of improvement diminishes at higher configurations, suggesting diminishing returns due to limitations such as instruction dependencies and overhead.

While a lower CPI is desirable, it is important to consider the trade-off with the clock period, as increasing parallelism often requires more complex hardware, which can extend the clock period. Configurations such as $N = 2$ and $N = 3$ strike an optimal balance between achieving a significant reduction in CPI and maintaining a manageable clock period. These configurations are often preferred as they deliver better performance without overly increasing design complexity or power consumption. Beyond $N = 4$, the hardware overhead and increased clock period may outweigh the modest gains in CPI, making these higher configurations less practical for many applications.

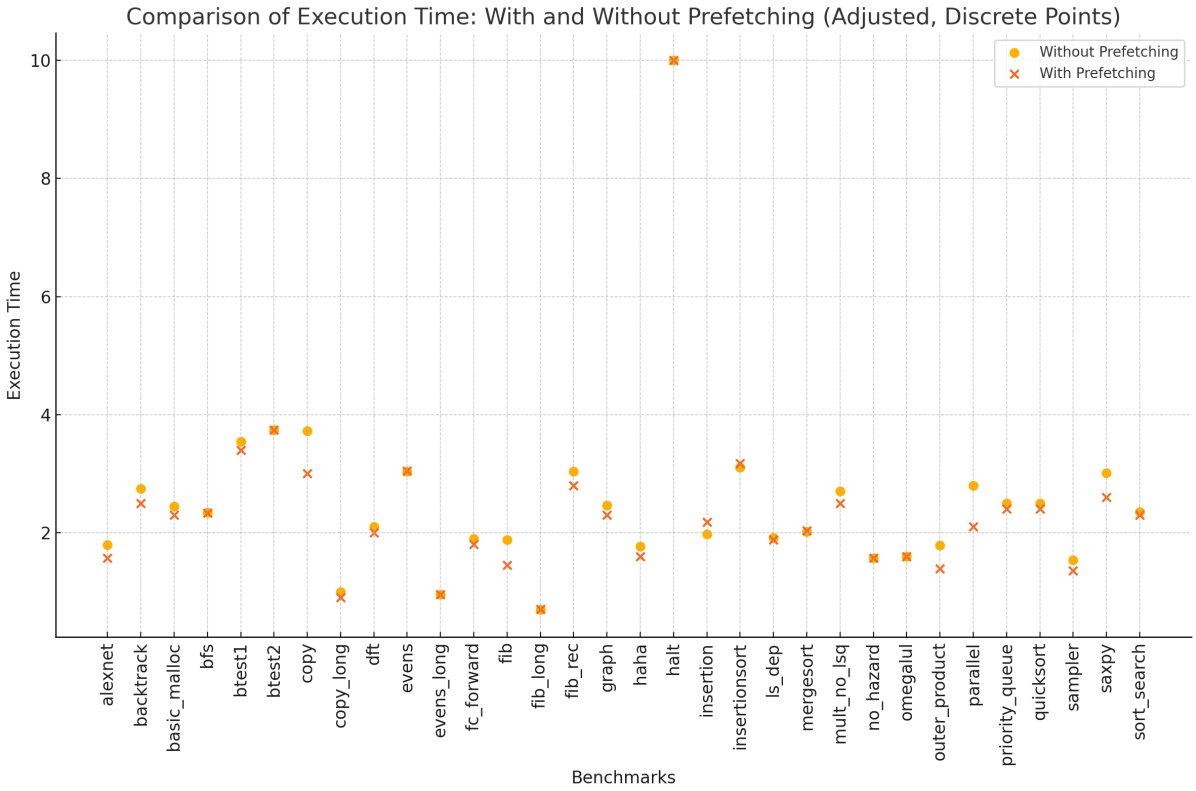

Figure 3:

The graph above illustrates the comparison of execution times across various benchmarks with and without prefetching. Prefetching consistently demonstrates comparable or reduced execution times for most benchmarks, highlighting its effectiveness in improving performance by mitigating memory latency and reducing pipeline stalls. By proactively fetching instructions before they are needed, prefetching minimizes the time the processor spends waiting for memory accesses, which is critical for performance in modern processors. The results reinforce that prefetching is particularly beneficial, as it reduces the overall execution time significantly compared to scenarios without prefetching.

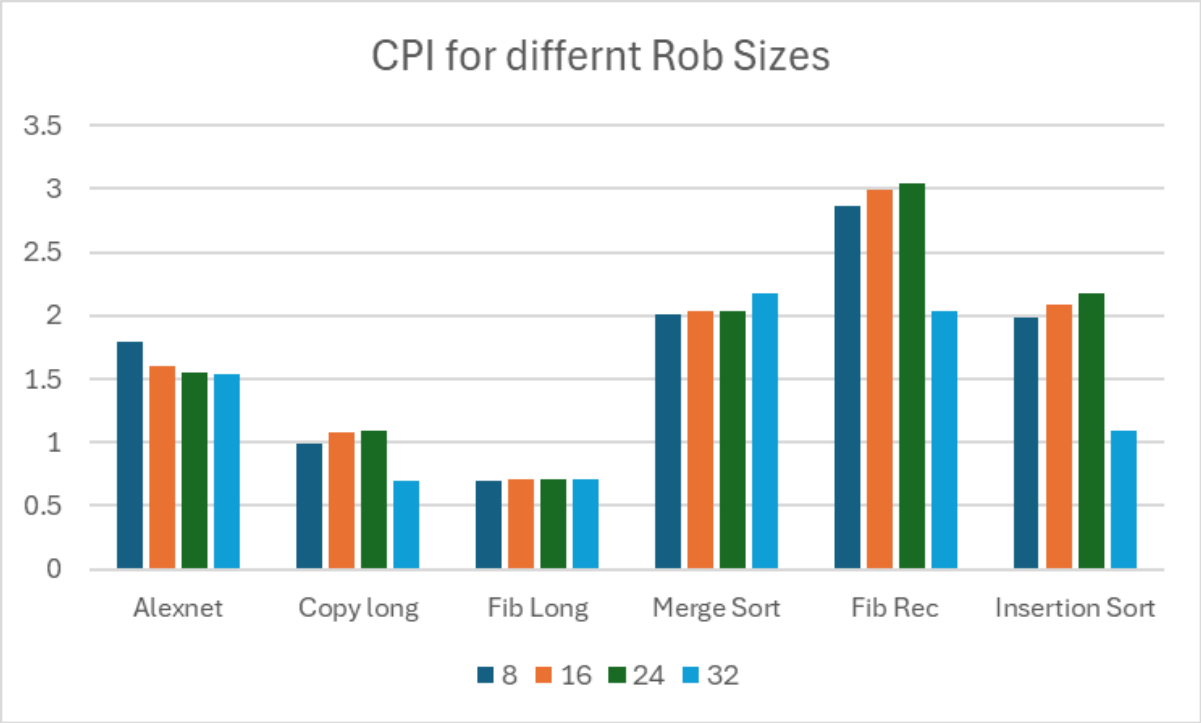

Figure 4:

The graph illustrates the CPI (Cycles Per Instruction) across various benchmarks for different Reservation Station (RS) sizes (8, 16, 24, and 32). The results show that changing the RS size has minimal impact on CPI for most benchmarks, with only negligible differences as the RS size increases. Given this minimal effect, we chose the smallest RS size of 8 to reduce the clock period and critical path, optimizing overall hardware performance and efficiency without compromising execution speed

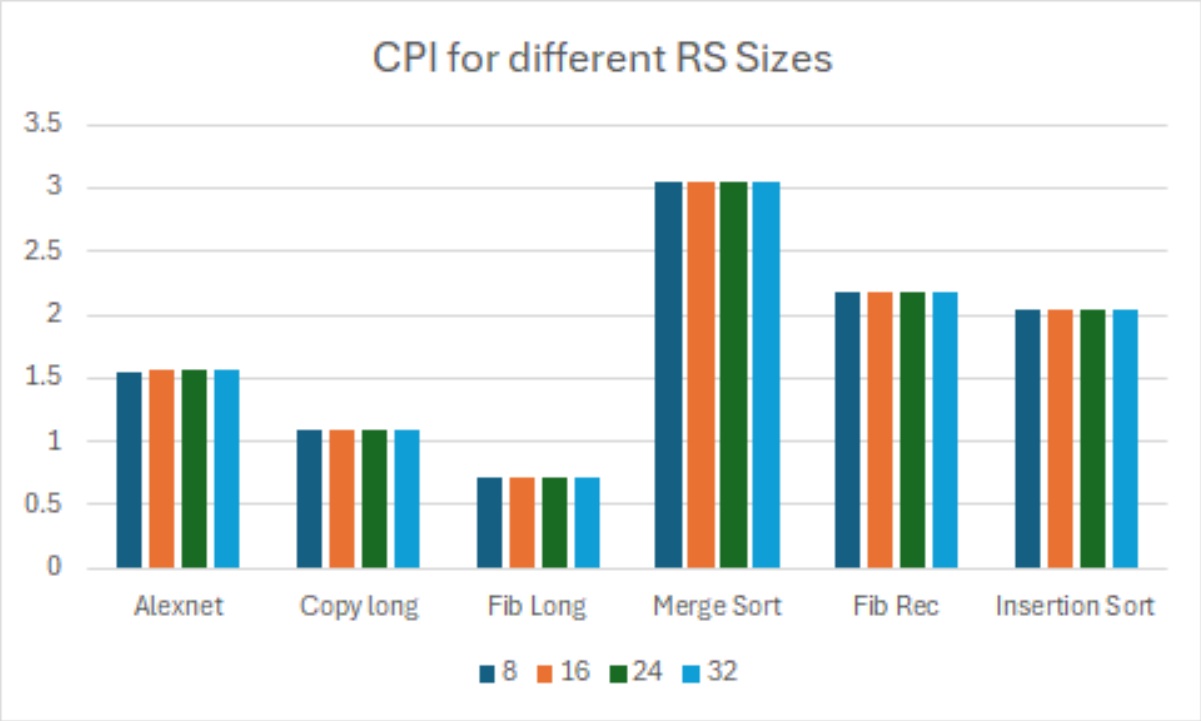

Figure 5:

The graph shows the CPI (Cycles Per Instruction) for various benchmarks across different Reorder Buffer (ROB) sizes (8, 16, 24, and 32). It is evident that increasing the ROB size to 32 significantly improves CPI for larger and more complex programs like "Insertion Sort," "Fib Rec," and "Copy Long." These benchmarks benefit from the larger ROB as it allows better handling of instruction dependencies and out-of-order execution, reducing stalls and improving overall efficiency. Although smaller ROB sizes suffice for simpler programs, the notable performance improvement for larger workloads justifies choosing an ROB size of 32 to optimize CPI and ensure robust performance across diverse applications.

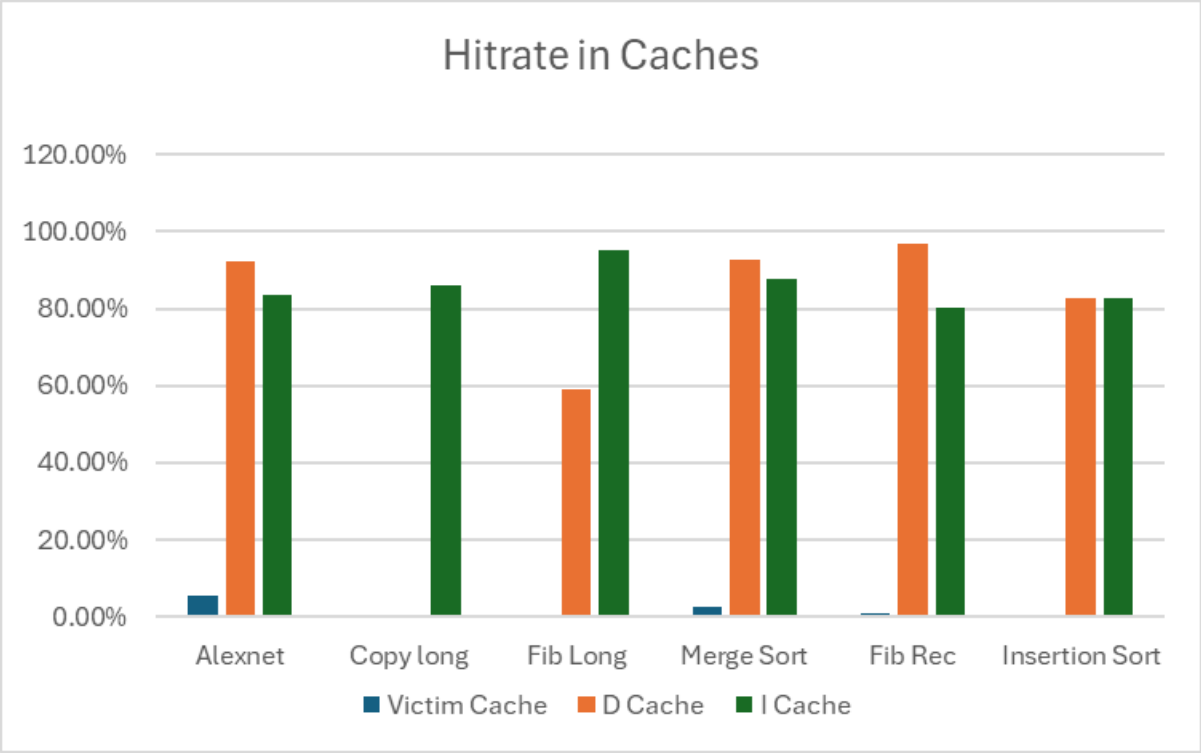

Figure 6:

The graph shows the hit rates for the Victim Cache, D Cache, and I Cache across various benchmarks. While the D Cache and I Cache achieve consistently high hit rates, particularly for benchmarks like "Merge Sort," "Fib Rec," and "Insertion Sort," the Victim Cache has a notably lower hit rate. Despite this, the Victim Cache plays a critical role in reducing conflict misses by capturing blocks evicted from the primary caches, ensuring smoother performance and preventing costly memory accesses. We chose to implement the Victim Cache to complement the D Cache as it enhances overall cache efficiency and resilience, particularly for benchmarks with irregular memory access patterns. This layered caching strategy strikes a balance between maximizing hit rates and minimizing performance degradation in edge cases.

4.1 GShare Analysis

Unfortunately, we cannot include analysis from the branch predictor. We left the GShare branch predictor in our final implementation of the project but commented out the line where we were making the actual prediction and indexing into the BTB. This is because, when making predictions, we encountered difficulties handling `jalr` instructions. After thorough testing, we were unable to determine the root cause of the bug.

In our current pipeline, we squash the pipeline in the following cases:

- If the branch was taken and the calculated branch target does not equal `NPC`.
- If the branch was not taken and `NPC` does not equal `PC + 4`.

The issue appears to be related to the writeback to a register during the execution of `jalr`. In certain files, such as `fib_rec.out`, when the prediction is made, the registers following the `jalr` instruction—although the correct `PC` is produced—start to exhibit incorrect writeback outputs. This behavior leads us to believe the problem lies in how `jalr` and its writeback are handled, particularly in cases where the prediction correctly determines the branch direction as taken.

5 Testing Methodology

Thorough testing is critical to ensure the correctness, reliability, and non-fragility of the processor design. Our testing methodology followed a structured, multi-stage approach, beginning with the individual testing of all modules before integrating them into the overall system. This modular testing strategy allowed us to isolate and address errors early in the development process, minimizing debugging efforts during integration and ensuring that the system was resilient to potential edge cases and unanticipated workloads.

5.1 Unit Testing

The first step in our testing methodology was rigorous unit testing for each individual module. These tests ensured that every component behaved as expected in isolation before moving on to system-level integration. For most modules, excluding memory, we simulated their functionality in C++ and compared the behavior of our SystemVerilog implementation to the ground truth provided by these simulations. This approach enabled us to validate the correctness and robustness of critical components such as the Instruction Buffer (Ibuff), Reservation Station (RS), Reorder Buffer (ROB), Store Queue, and Load Buffer. By maintaining a reference implementation in C++, we systematically tested the state transitions and outputs of each module against expected results, ensuring alignment with the intended behavior and minimizing potential issues during integration.

Each unit test was designed to account for edge cases and stress conditions to verify non-fragility. For example, the Reservation Station's ability to dynamically schedule and dispatch instructions was cross-verified with scheduling decisions in the C++ simulation under high contention scenarios, ensuring that it handled dependencies and resource conflicts accurately. Similarly, we verified the Reorder Buffer's capacity to maintain instruction order and enable precise state recovery by simulating mispredictions and exception scenarios.

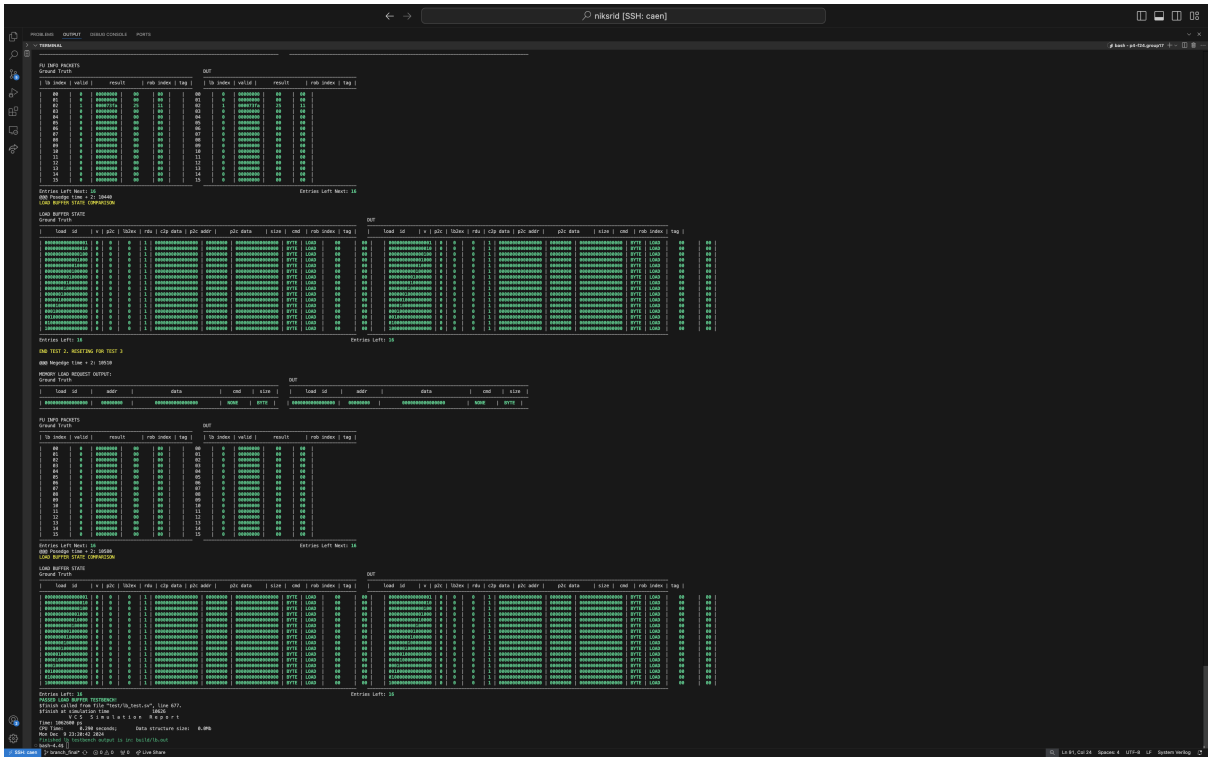

Figure 7: Above is the output for the load buffer unit test. On the left is the output of the processor and on the right is the ground truth.

The memory subsystem, which involved more complex interactions and timing dependencies, was subjected to targeted unit tests. We individually tested components such as the Data Cache, Store Queue, Load Buffer, and Miss Status Holding Registers (MSHRs) for various scenarios, including cache hits and misses, store-to-load forwarding, and dependency resolution. These tests ensured that the memory components handled realistic and edge-case workloads effectively while maintaining data consistency and throughput.

5.2 System Testing and Integration

Once individual modules were verified, we proceeded to system tests, progressively integrating modules to validate their collective behavior. Our system testing approach was incremental, allowing us to isolate issues at each stage of integration and address them promptly.

The integration process began with the memory subsystem. We combined the Data Cache, MSHR, and Victim Cache into a unified data memory module and tested their interactions to ensure efficient handling of memory requests. This stage verified that the memory components worked together seamlessly, resolving dependencies and minimizing latency.

Next, we integrated pre-execution modules, including the Instruction Buffer and Reservation Station, to simulate instruction dispatch and scheduling. These modules were tested for their ability to handle dependency tracking and resource management under varying workloads. Execution modules were then added to the pipeline, and we verified whether the execution logic produced correct outputs. To simplify testing at this stage, we used fake fetch data to provide input to the pipeline, enabling us to focus on

the core execution flow without introducing fetch-related dependencies.

Finally, we integrated the fetch stage and instruction cache into the pipeline, completing the end-to-end system. The fetch stage and instruction cache, which had already been tested independently, were validated in the context of the full pipeline. By introducing components incrementally, we ensured that issues were isolated and addressed at each stage, reducing complexity during debugging. The branch prediction was also occurring in the fetch stage. However, we had trouble testing the branch prediction and recovery until the entire pipeline was finished. This was because we would have to make a prediction and then stimulate receiving the outcome back and updating the states. Since our pipeline came together closer to the deadline than anticipated, it was difficult to debug the issue regarding jalrs that we were running into as mentioned in Section 3.1.

5.3 Full Pipeline Testing and Results

After the full pipeline was assembled, we conducted comprehensive tests to validate its performance and correctness. These tests included a variety of workloads, from simple instruction streams to complex, memory-intensive programs, ensuring that the pipeline handled all scenarios effectively. Testing also included branch-heavy workloads to verify the accuracy of the branch prediction logic and its interaction with the instruction fetch stage.

Our iterative approach to testing and integration proved highly effective. By testing each module individually and integrating them part by part, we built confidence in the design's reliability at every stage. This strategy paid off during our submission to the autograder, where our design passed all tests successfully on the first attempt, demonstrating the robustness and correctness of our implementation.

In summary, our testing methodology emphasized rigorous unit tests for individual modules and systematic system tests for integrated components. By combining C++ simulations, incremental integration, and comprehensive pipeline validation, we ensured the correctness, reliability, and non-fragility of the processor design, achieving a high-performance implementation capable of meeting demanding workloads.

6 Project Management

Effective project management was a cornerstone of our success, ensuring that tasks were distributed evenly among team members and that our goals were met within the designated timelines. At the outset of the project, we established clear objectives and created a detailed work plan to assign roles and responsibilities to each team member. Weekly milestones were set to track progress, identify potential bottlenecks, and ensure alignment among all group members.

We divided the project into logical sub-tasks based on functionality, such as memory subsystems, fetch and execute stages, branch prediction, and debugging. Each member was assigned specific components to work on, as outlined in Table 2. This structured division of labor allowed team members to specialize in their areas while contributing to the broader project objectives. Regular check-ins were held to synchronize efforts across sub-groups, providing an opportunity to share progress, discuss challenges, and adjust plans as necessary.

Throughout the project, we adhered to a flexible yet disciplined project plan. For example, when unexpected issues arose in integrating the store queue, we re-evaluated our

milestones and redistributed tasks to balance the workload effectively. This adaptability ensured that we remained on track to meet our deadlines without sacrificing the quality of our work.

Collaboration was a key strength of our team. We utilized version control systems such as Git to manage our codebase, enabling smooth integration and consistent tracking of changes. Communication tools and platforms were used extensively to coordinate efforts, resolve conflicts, and provide feedback. By fostering an open and supportive environment, we encouraged each team member to contribute their expertise while learning from others.

Overall, our project management strategy ensured a well-organized, collaborative, and goal-driven workflow. The consistent setting and monitoring of milestones, along with a strong emphasis on communication and flexibility, allowed us to deliver a fully functional processor design within the required time frame.

Table 2: Work Division

Member Name	**Component Worked Upon**	**Percentage of Work**
Porvesh Bala	Data Memory, ICache, Fetch, Pre Execute	16.6%
Lohit Kamatham	Data Memory, ROB, Execute	16.6%
Velu Manohar	Data Memory, Execute, Fetch	16.6%
Nikhil Sridhar	Git, Integration, Pre-Execute, Store Queue	16.6%
Aidan Spizz	Branch Predictor, IBuff, Fetch	16.6%
Coy Catrett	Debugger, Store Queue, Load Buffer	16.6%