CSEE4824 Computer Architecture 3-Way Superscalar R10K Out-of-Order Processor Design

Group 6

Yuan Jiang yj2848@columbia.edu
Yuxi Zhang yz4935@columbia.edu
Junfeng Zou jz3850@columbia.edu
Lingxi Zhang lz2991@columbia.edu
Zhelin Su zs2709@columbia.edu
Hongrui Huang hh3084@columbia.edu

Abstract

This paper presents the design and implementation of an R10K out-of-order processor supporting a 3-way superscalar pipeline with advanced microarchitectural features. Key components include a Reorder Buffer (ROB), Freelist, Reservation Station (RS), physical register management (MapTable and Architecture Map), and an enhanced Gshare branch predictor. Further optimizations such as Instruction Prefetch, Store Queue, and an Advanced Data Cache are integrated to improve performance. The processor successfully executes all 33 benchmark instructions provided in the course, achieving full functional correctness in simulation and synthesis. Performance analysis shows an average CPI (Cycles Per Instruction) of 2.826, with a minimum synthesizable clock period of 11.2 ns and an average TPI (Time Per Instruction) of 31.651 ns. These results demonstrate the efficiency of our design in balancing complexity and speed while maintaining correctness in a multi-issue, out-of-order execution environment.

1 Introduction

Out-of-order execution is a fundamental technique for increasing instruction-level parallelism in modern high-performance processors. Inspired by the R10K microarchitecture, we designed and implemented a robust out-of-order processor capable of handling dynamic instruction scheduling, precise exceptions, and speculative execution. Our design features key architectural modules including a Reorder Buffer (ROB) for in-order retirement, a Free List for physical register allocation, Reservation Stations (RS) for dynamic issue, Map Table and Architectural Map Table for register renaming, and a branch predictor to support speculative execution.

To enhance throughput, our processor supports 3-way superscalar instruction issue and incorporates advanced modules such as an Instruction Prefetch Unit to reduce fetch latency, an improved Gshare Branch Predictor for higher prediction accuracy, a Store Queue to manage memory ordering, and an Advanced Data Cache for efficient memory access. The processor successfully runs all 33 benchmark programs provided, achieving a synthesis-friendly implementation. Our simulation results show an average CPI of 2.826, with a minimum synthesizable clock period of 11.2 ns, leading to an average TPI of 31.651 ns. These results demonstrate the effectiveness and practicality of our out-of-order processor design.

To view our source code and further documentation, please visit OoO Processor R10K.

2 Design Overview

Our processor design is composed of a modular pipeline architecture optimized for instruction-level parallelism and speculative execution. The processor features 16-entry Reservation Stations (RS), a 32-entry Reorder Buffer (ROB), and a 64-register Physical Register File (PRF). Instruction dispatch supports 3-way superscalar execution, and functional units include 3 ALUs, 2 Load units, 1 Branch unit, and 2 4-stage Multipliers.

The front-end includes a 256-Byte instruction cache and a prefetcher that fetches up to 12 lines ahead to reduce fetch stalls. A 32-entry Gshare-based branch predictor improves control flow accuracy. The back-end memory system includes a Store Queue (SQ) with 8 entries and a 256-Byte data cache.

Figure 1 illustrates the overall microarchitecture. Instructions are fetched into the pipeline with the help of the I-Cache and prefetcher, and control flow predictions are made by the branch predictor. The dispatch stage interacts with the freelist and register renaming modules (map table and archi map) to allocate physical

registers. Dispatched instructions are queued into the RS, from where they are issued to appropriate functional units. Execution results are broadcast via the Common Data Bus (CDB) and later committed in program order through the retire logic and the ROB. Memory access operations interact with the D-Cache and SQ via a unified memory controller.

Together, these components form a high-performance out-of-order core capable of achieving low CPI and efficient resource utilization under realistic workloads.

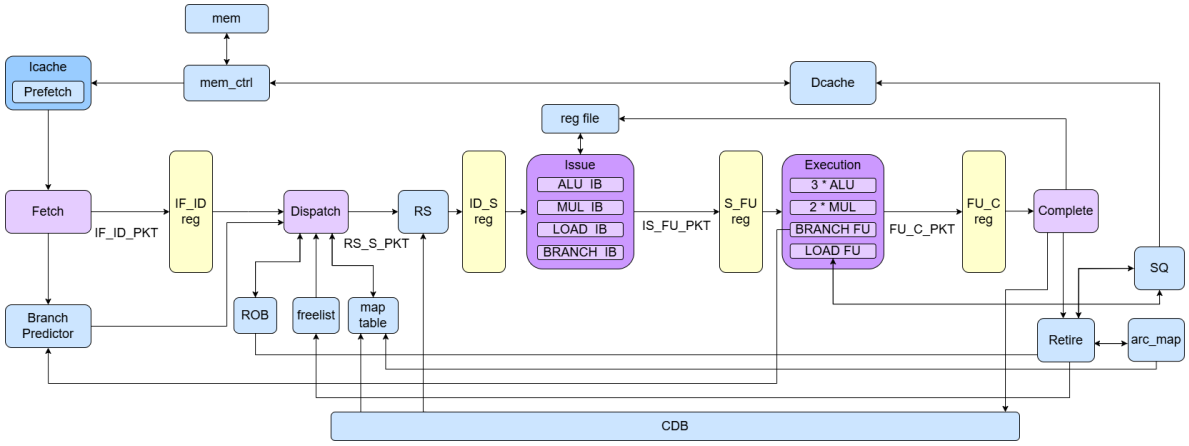

Figure 1: Block Diagram of R10K OoO Processor

3 Basic Function Implementation

3.1 Fetch Stage

The fetch stage is the very first step in our 3-way superscalar pipeline, responsible for pulling raw instruction words out of the I-cache and packaging them for dispatch. Its primary role is to maintain a sliding "window" of up to three program-counter (PC) values, fetch the corresponding 32-bit words, and send them forward into the IF/ID packet format (PC, next-PC, instruction bits, and valid flag). By handling branch redirects and back-pressure from downstream stalls, it ensures that the decoder always sees the freshest, in-order stream of up to three candidate instructions per cycle.

Each cycle, the front end issues three contiguous fetch addresses, retrieves their instruction words, and pauses on cache misses or when downstream stages are full. A taken-branch instantly pivots the next-PC window to the new target without flushing the whole pipeline. Once valid data arrives, each word is tagged with its PC and successor PC, bundled as an IF/ID packet, and forwarded—while the same PCs feed the branch predictor. This logic sustains a steady 3-instruction (12-byte) throughput per cycle, only interrupted by genuine stalls or holds.

3.2 Reorder Buffer

Reorder Buffer (ROB) is a critical component in out-of-order CPU execution that ensures instructions are retired in program order. It supports up to three instruction dispatches and retirements per cycle.

The ROB tracks entries with head and tail pointers, where the tail handles incoming dispatched instructions and the head manages retirements of completed instructions. It calculates available slots based on the head-to-tail distance and considers store instruction stalls from the store queue to control how many instructions can be safely retired in a given cycle.

The module also handles special cases like an empty ROB and adjusts pointers accordingly. When instructions complete, they are marked as completed in the buffer and become candidates for retirement at the head, respecting store-related stall conditions.

The module coordinates with the store queue and dispatch logic via interfaces for instruction entry, completion status, and recovery from mispredicted branches, maintaining consistency and correctness in the processor pipeline.

3.3 Freelist

The freelist is responsible for managing the pool of available physical registers and plays a central role in register renaming and recovery. In our design, the freelist holds up to 32 physical registers, supporting three allocations and three retirements per cycle through a circular buffer structure. Head and tail pointers advance on dispatch and retire events respectively, with wrap-around support to fully utilize the register pool.

Each cycle, the freelist checks the availability of up to three physical registers by calculating the distance between the head and tail. Based on this, it sets a valid allocation mask to indicate whether 1, 2, or 3 new physical registers can be granted. When instructions retire, their old physical registers are returned to the stack in order, ensuring that physical register resources are recycled efficiently and safely.

Special care is taken for branch misprediction recovery: when a recovery signal is asserted, the freelist resets its state by rolling both the head and tail pointers to the same location, marking the freelist as full. This enables an immediate and consistent restart of the renaming pipeline. The freelist guarantees that all physical registers are unique and only reused after explicit retirement, ensuring correctness in the renaming and retirement path.

3.4 Reservation Station

The Reservation Station (RS) buffers dispatched instructions that are waiting for operands to become ready and for functional units to be available. Our design instantiates 16 RS entries shared across all functional units and supports three-way dispatch and issue per cycle. Each entry holds operand tags, ready flags, and the functional unit type. A set of priority selectors identifies up to three available slots per cycle to accept new instructions. When insufficient space is available, the RS asserts a stall signal back to the dispatch stage, preventing overflows and maintaining structural correctness.

To track operand readiness, the RS monitors the Common Data Bus (CDB). When a broadcasted tag matches an entry's source register, the corresponding ready bit is updated. This enables dynamic wake-up of instructions without requiring centralized control. The RS also receives memory dependency signals from the Store Queue to ensure loads are issued only when memory order constraints are satisfied.

Instruction issue is based on both operand readiness and functional unit availability. The RS evaluates each entry and selects up to three instructions per cycle using reverse-priority masking. Selected entries are cleared after issue, and corresponding instruction packets are forwarded to the issue queue.

3.5 Map Table & Architecture/Physical Registers

The map table implements register renaming by translating architectural register (AR) indices to physical register (PR) tags. Our design supports 64 physical registers, with 32 dedicated architectural registers and the remainder dynamically allocated via a freelist. At any cycle, the map table can rename up to three instructions simultaneously, each generating a new mapping from an AR to a freshly allocated PR. These mappings are stored in a 32-entry array representing the current architectural state.

To support correct instruction execution and precise exception handling, the map table maintains both tag mappings and readiness bits. When a physical register is allocated for a destination AR, the corresponding ready bit is cleared. Completion of that instruction is detected through a 3-wide Common Data Bus (CDB) broadcast, which sets the ready bit to 1 once a matching tag is seen. This enables efficient wakeup logic in the reservation station and avoids redundant stalls due to pending operands.

Branch recovery is handled through a checkpoint mechanism: a separate map table copy is maintained, and when a misprediction occurs, the map table state is immediately rolled back to the checkpointed version. Combined with the freelist and ROB rollback, this ensures that register renaming remains consistent and reversible under speculative execution.

3.6 Dispatch Stage

The dispatch stage is responsible for processing up to three decoded instructions per cycle and preparing them for out-of-order execution by allocating resources including physical registers, ROB entries, and optionally Store Queue entries. It filters instructions based on structural stall signals (from ROB, RS, SQ and freelist) and branch predictions to ensure only valid instructions proceed.

Each instruction is decoded to extract control signals like function unit selection and operand types, and if the instruction writes to a destination register, a new physical register is allocated from the freelist. The module updates mappings between architectural and physical registers, prepares instruction packets for the Reservation Stations with operand readiness and dependencies, sets up corresponding ROB entries with metadata like program counters and prediction info, and signals Store Queue allocation for store instructions. Overall, it coordinates instruction dispatch, dependency tracking, and resource allocation to enable efficient execution in a 3-way superscalar out-of-order processor.

3.7 Issue Stage

The issue stage is responsible for routing ready instructions to their corresponding functional units through FIFO buffers. Our processor supports four types of functional units: ALU, Load/Store, Multiplier, and Branch. Each connected to a dedicated 3-slot FIFO. The issue stage accepts up to three instructions per cycle from the reservation station and classifies them based on their target functional unit. Each instruction is packaged and inserted into the appropriate FIFO if space is available.

Each functional unit FIFO is implemented with a configurable depth (set to 32 entries in our design) and supports up to three enqueues and three dequeues per cycle. Instructions are reordered based on availability and priority before insertion, ensuring that the most critical operations are issued first. The FIFOs generate back-pressure signals (`almost_full`) to the upstream logic when nearing capacity, ensuring that no new instructions are dispatched into a full unit queue.

3.8 Execution Stage

The execution stage is where all dispatched micro-operations actually do their work, translating issued instructions into arithmetic, logical, memory, and control actions. It hosts a pool of specialized functional units—multiple ALUs for integer work, pipelined multipliers for long-latency multiplies, load units that interface with the data cache and store queue, and a dedicated branch unit for comparisons and target calculation. By centralizing these units, the stage turns renamed operands into real results and tags them for the rest of the pipeline.

Because our design instantiates eight functional units in total (three ALUs, two multiplier pipelines, two load ports, and one branch unit), it can broadcast up to three completion packets per cycle over the CDB. Load units can generate two simultaneous cache requests and two completion broadcasts; ALUs can retire up to three operations at once, and the branch unit always handles one. This parallelism ensures high out-of-order throughput, bounded only by functional unit availability and downstream resource readiness.

3.9 Complete Stage

The complete stage is responsible for selecting up to three instructions that have finished execution in the functional units and preparing them for retirement. Each cycle, it scans the ready instructions in priority order and picks up to three of them to be completed.

For these selected instructions, it gathers their computed results and physical destination registers, broadcasts them on the common data bus for register file write-back, and generates the necessary information for the reorder buffer to commit them in order. If a selected instruction is a taken branch, its updated program counter is also passed along for correct control flow handling. Instructions that are finished but not selected in the current cycle are marked to be stalled, preventing their state from being overwritten until they are processed. This stage ensures smooth handoff from execution to retirement while maintaining correct instruction ordering and data flow.

3.10 Retire Stage

The retire stage sits at the tail of the pipeline, turning completed reorder-buffer entries into permanent architectural updates. Each cycle it examines up to three in-order ROB head entries: if an entry is marked done, it issues a retire enable, writes its final mapping into the architectural map table, and frees the old physical register back to the free list (and signals the store queue when it's a store).

If any of those entries indicates a precise-state event—such as a taken-branch misprediction or a fault—the retire logic immediately asserts a recovery enable, supplies the correct PC for refetch, and rolls the map table and free-list pointer back to the checkpointed state. This flush-and-recover mechanism stops further retirements in that cycle, ensuring correctness without draining unrelated in-flight instructions.

By committing instructions in strict order—up to three per cycle when no exception intervenes—the retire stage both advances the architectural state and recycles precious resources, balancing high throughput with precise exception handling.

3.11 Caches & Memory Controller

The processor features a split memory hierarchy consisting of an instruction cache (Icache) and a data cache (Dcache), both interfaced through a shared memory controller.

Icache is a direct-mapped, non-blocking instruction cache with a total size of 256 bytes. It supports 8-byte cache lines and contains 32 lines in total. When a cache miss occurs, Icache sends a memory request to the controller.

Dcache is a two-way set-associative cache with 16 sets and 8-byte cache lines, also totaling 256 bytes of capacity. It is integrated with Miss Status Handling Registers (MSHRs) to support non-blocking memory accesses, enabling simultaneous in-flight memory requests and handling of cache misses without stalling the pipeline.

The memory controller handles arbitration between Icache and Dcache. It prioritizes Dcache requests to avoid pipeline stalls due to store or load operations. Icache requests are only issued when Dcache is idle. The controller forwards commands and addresses to memory and routes the responses back to the appropriate cache based on tag matching.

4 Advanced Feature Implementation

4.1 3-Way Superscalar

Our processor features a 3-way superscalar out-of-order (OoO) pipeline, enabling up to three instructions to be fetched, dispatched, issued, executed, completed, and retired in parallel each cycle. This significantly enhances instruction throughput and overall performance compared to scalar or in-order pipelines, particularly under workloads with high instruction-level parallelism (ILP). To support this, all key backend components, including the reservation station, functional units, and reorder buffer, are designed to process three instructions per cycle in parallel. Implementing 3-way superscalar execution introduces considerable architectural complexity, particularly in instruction classification, dependency tracking, stall back-pressure control, and maintaining precise state recovery.

4.2 Instruction Prefetch

Prefetching is a technique that issues memory requests for data—or in this case, instruction cache lines—ahead of the processor's actual demand, with the aim of hiding memory latency behind useful work. In its simplest form, a prefetch unit watches the stream of fetch addresses and requests the "next" line (or a fixed number of lines ahead) as soon as the current line is found in the cache. Because instruction fetches are highly predictable—especially in straight-line code and tight loops—and because the instruction cache already knows which lines are present or missing, it makes sense to place the prefetch logic right alongside the ICache controller. Integrating prefetch into the ICache module minimizes added latency (no extra crossings between different blocks), reuses the existing tag–index comparators and write–back ports, and ensures that requests naturally follow the same arbitration and

error-recovery paths as regular fetch misses.

When we began integrating a prefetch unit into the ICache, our first thought was simply to exploit the most obvious form of spatial locality: whenever the fetch stage reads address A, we might as well request line $A + \texttt{cache\_line\_size}$ before the processor actually needs it. This basic step already requires careful bookkeeping: we must tag each prefetch request so that when the memory controller's response arrives, we know which cache index to fill and which tag to clear. Without this tag–index pair, responses could write to the wrong way or be dropped entirely.

Once the simple next-line idea was in place, we confronted the issue of bus contention. A raw prefetch would happily collide with a genuine fetch-miss, potentially delaying the very instruction the CPU waited for. To avoid that, we introduced a bus-priority signal `pf_bus_priority` that is asserted only when no fetch-miss request is outstanding. In the ICache's assignment to `icache_mem_req_cmd`, we mask out prefetches whenever `require_load` (the fetch-miss strobe) is asserted, thereby guaranteeing that real fetch misses always win the arbiter. That arbitration logic lives in the ICache top module, but in our pipeline it ultimately wires through the memory controller's own priority scheme, so we verified that `pf_bus_priority` dovetailed correctly with `mem_controller` arbitration.

The prefetch logic continuously monitors the pipeline's branch signal (`icache_branch` in `icache.sv` and `branchEN` in `pipeline.sv`). Whenever a branch takes effect, we immediately flush the prefetch queue, deassert any pending requests, and set `icache_pref_done` high so that no old entries linger. This ensures that after a mispredicted branch, the processor never sees stale or irrelevant lines that were speculatively prefetched along the discarded path.

With arbitration and branch awareness handled, we turned to making prefetch robust in the context of pipeline stalls and hold conditions. Since the ICache can be asked to stall (`icache_pipeline_hold`) during flushes or memory backpressure, the prefetch unit must freeze its state—no new addresses, no partial queue updates—until the pipeline resumes. We accomplished this by gating all queue head/tail updates on the inverse of that hold signal, and by AND-masking incoming memory-response codes with $\sim\texttt{mc\_ic\_hold\_flag}$ so that pending responses during a hold do not inadvertently clear our tags.

Finally, we considered integration with the wider RISC-V pipeline. The prefetch module's outputs (`icache_pref_wEN`, `icache_pref_idx`, `icache_pref_id`) had to connect to the same cache arrays that the fetch stage and data cache drive, so we connected the existing write–back ports of the ICache. The ICache instantiation and subsequent `mem_controller` arbiter merge fetch and prefetch commands into a single stream, preserving the original hand-off to the rest of the memory hierarchy. We confirmed that our prefetch never changes the semantics of the standard ICache–fetch–branch interaction but simply augments it in the background. By building up from a naive next-line prefetch, adding priority arbitration, incorporating branch flushing, and respecting pipeline holds, we arrived at a design that fits cleanly into your existing modules and significantly reduces instruction-cache miss penalties in predictable code sequences.

4.3 Branch Predictor

Branch prediction is an essential feature for improving performance in deep-pipeline and superscalar processor architectures. To minimize pipeline stalls caused by conditional branch instructions, we have implemented a sophisticated Tournament Branch Predictor that leverages a hybrid approach by combining the strengths of local and Gshare prediction strategies. The fundamental motivation is to capture both static, per-branch behavior and dynamic, global correlation among branches, resulting in significantly improved prediction accuracy.

Initially, simple branch predictors such as the bimodal predictor (1-bit or 2-bit) were used. A 1-bit bimodal predictor relies purely on the previous outcome of a branch instruction to predict its future behavior. However, this approach quickly flips its prediction upon a single misprediction, leading to frequent incorrect predictions, particularly in looping structures. To address this instability, the 2-bit saturating bimodal predictor was introduced, employing four distinct states—strongly taken, weakly taken, weakly not-taken, and strongly not-taken. This improved predictor only changes prediction states after two consecutive mispredictions, greatly enhancing stability and accuracy for loops and repetitive branch patterns.

To further exploit program behavior, the Gshare predictor incorporates global branch history by XORing the lower bits of the branch instruction's program counter (PC) with a Global History Register (GHR) containing the results of the most recent branches. This mechanism allows the predictor to identify and utilize correlations between different branch instructions, thereby significantly reducing mispredictions compared to purely local bimodal methods.

Our Tournament Predictor implementation further enhances prediction accuracy by combining these two approaches into a unified scheme. It maintains separate predictor tables (Local and Gshare) and employs a chooser table to dynamically select between them. Both Local and Gshare predictor tables contain multiple entries, each consisting of a validity bit, a tag (to uniquely identify each branch and mitigate aliasing), a 2-bit saturating counter indicating branch direction, and a stored branch target address. The chooser table is composed of entries with 2-bit saturating counters indicating historical accuracy, dynamically adjusting preference towards either Local or Gshare prediction strategies.

During the instruction fetch stage, our predictor simultaneously accesses both the Local and Gshare tables using the indexed addresses derived from the PC bits and the GHR. The chooser table then determines which predictor's outcome (taken or not taken) and associated target address should be used for the actual prediction. The use of separate tags and validity bits ensures that predictions are only considered valid when the stored entries match the current PC.

When a branch outcome resolves during execution, the predictor updates both Local and Gshare predictor tables using the actual branch result and target address. The 2-bit counters within each table are adjusted accordingly, reinforcing correct predictions. Concurrently, the chooser counter updates its value based on predictor accuracy: incrementing towards Gshare if it outperforms Local, or decrementing towards Local if Local proves more accurate. Additionally, the Global History Register shifts in the latest resolved branch outcome, maintaining an updated global context.

Our Verilog implementation utilizes configurable parameters for predictor dimensions and indexing offsets clearly indicated by symbolic constants, such as `BP_TBL_SIZE`, `BP_INDEX_OFFSET`, and `SYS_BRANCH_PREDICTION`. Predictor tables initialize upon reset to weakly favor global predictions, thus optimizing performance during initial execution phases.

4.4 Store Queue

This Store Queue module is responsible for managing in-flight store instructions, ensuring correct memory ordering, and supporting store-to-load forwarding.

The core logic maintains a circular buffer using head and tail pointers to track the valid entries and their allocation positions. When new store instructions are dispatched, they are allocated into available slots in the queue, and when stores are retired, the corresponding entries are cleared and prepared for writeback to memory.

The module carefully tracks how many entries are free and generates a block signal to prevent overflows. It also calculates the next positions of the head and tail pointers based on the number of stores dispatched and retired each cycle. A key function is store data commitment—if a store instruction is executed, the result is written into the appropriate SQ entry, replacing any previous content.

Additionally, the module handles logic for selecting the oldest store entries for potential writeback, provides allocated indices to dispatch units, and prepares data for load units to determine if forwarding from older stores is needed. This ensures proper in-order commit and memory consistency, especially in out-of-order execution environments.

4.5 2-Way Set-Associative Dcache & MSHRs

To support high-throughput memory access in our out-of-order processor, we implement a 2-way set-associative data cache (dcache) backed by a Miss Status Handling Register (MSHRs) system. The dcache consists of 16 sets with 2 ways each, totaling 256 bytes of capacity. Each cache line is 8 bytes, and each set maintains a 1-bit Least Recently Used (LRU) indicator for replacement decisions. The design supports three concurrent store ports

(from the Store Queue) and two load read ports, enabling high parallelism in memory operations. Compared to a direct-mapped cache of the same size, this 2-way configuration significantly reduces conflict misses, particularly under workloads exhibiting spatial or temporal locality stress.

The cache operates with a write-back and write-allocate policy, maintaining per-line valid and dirty bits. To handle memory misses without stalling the pipeline, the top-level dcache module integrates an MSHRs controller capable of tracking up to 8 in-flight memory transactions. Each MSHRs entry stores full 64-bit addresses, operation type (load/store), dirty flags, byte-enable masks, and in some cases, partial store data to be merged with the incoming refill.

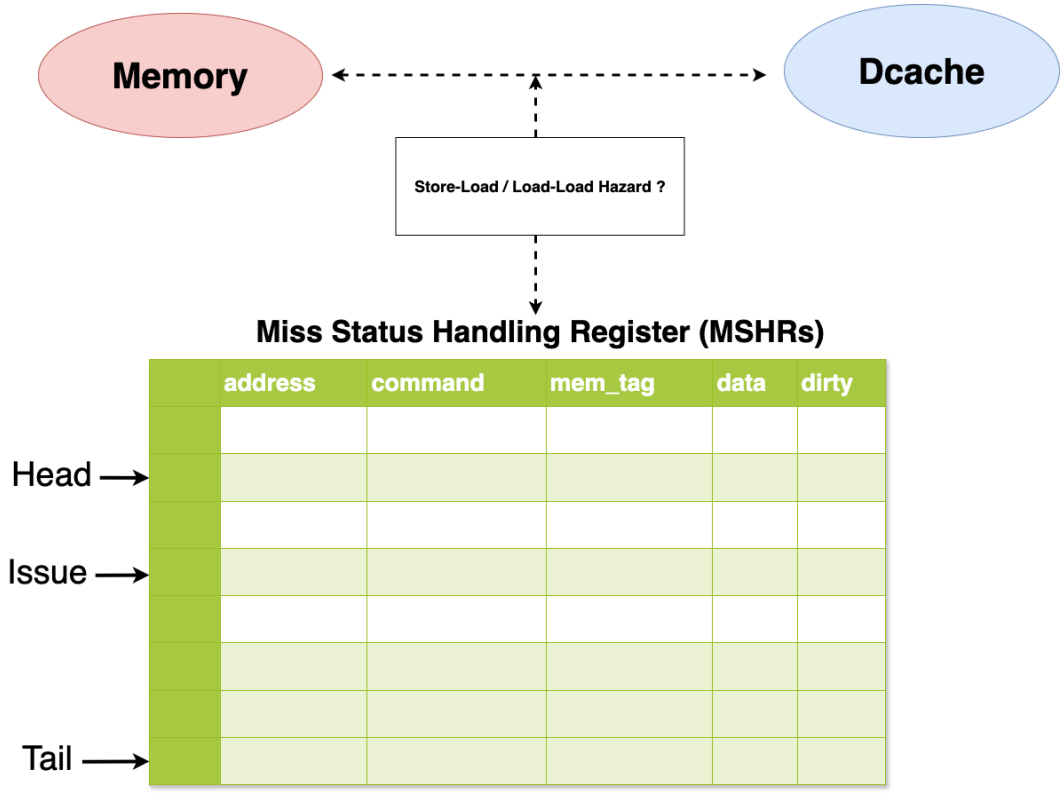

Figure 2: MSHRs Structure and Hazards Logic

When a load miss occurs, a new MSHRs is allocated with the aligned memory address and marked as a `BUS_LOAD` request. If a pending MSHRs entry already exists for the same cache block, the new load merges with the existing one, avoiding duplicate memory requests. Upon receiving the memory response, the corresponding entry is matched using the memory tag, and the returned data is written back into the cache with optional byte-masked merging if it originated from a store-miss.

If cache store miss occurs and the store data must eventually overwrite the incoming memory content, a `BUS_LOAD` entry is created with the dirty bit set. When the refill returns, the partial store data is byte-selectively merged with the memory response based on the byte-enable mask before being committed to the cache.

The MSHRs logic also proactively detects and resolves memory hazards. Load-Load merging avoids redundant requests by checking for in-flight MSHRs targeting the same cache block. Load-Store hazards are detected by comparing new load addresses against all pending store-miss entries. If any match, the load is forced to miss and wait for the store to complete. Finally, backpressure is applied via head-tail distance tracking; when the number of free MSHRs entries drops below a threshold, the Store Queue is selectively stalled to prevent overflow.

During a dirty eviction, the dcache already contains the latest committed data for that cache block. Instead of writing it back immediately through a blocking path, we create a `BUS_STORE` MSHRs entry, which queues the

eviction as a non-blocking memory transaction.

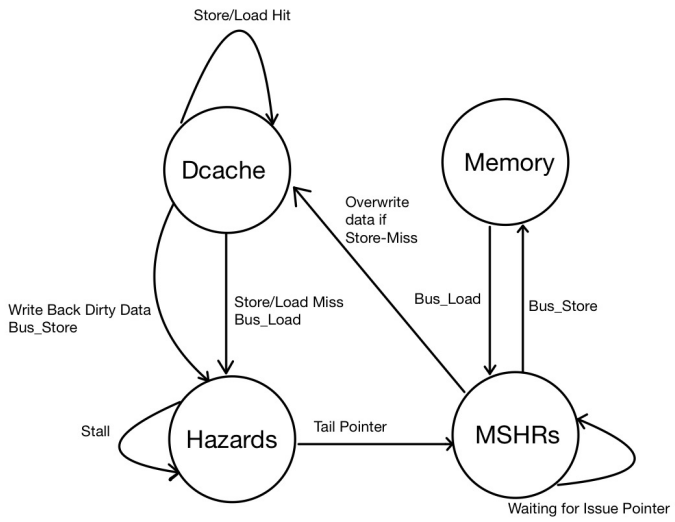

Figure 3: Logic FSM for MSHRs and Date Cache

At the end of program execution, to guarantee memory coherence and prevent data loss, all remaining dirty blocks are flushed from both the D-cache and any pending MSHRs entries. These entries are walked through systematically, and any outstanding `BUS_STORE` or dirty `BUS_LOAD` entries are serialized into memory write operations. This ensures that all cached updates are fully committed before system halt.

5 Testing

5.1 Test Method

Our verification strategy uses four complementary methods. Module-level testing exercises each RTL block in isolation: a dedicated SystemVerilog testbench applies predetermined input vectors and checks outputs with simple pass/fail markers to confirm basic functional correctness. We also compile the top-level design with a TEST MODE macro that exposes internal signals—such as cache tags, queue pointers, and prefetch statistics—so simulations can read these ports directly and observe internal state without intrusive probes. Integration-level simulation then runs the complete RTL pipeline on realistic RISC-V program images (‘.c $\rightarrow$ .elf $\rightarrow$ .mem‘), capturing write-back logs and pipeline-trace outputs to verify that modules interact correctly under true instruction streams. Finally, post-synthesis verification repeats both module and integration tests on gate-level netlists produced by logic synthesis, and gathers timing-slack data to ensure that synthesis optimizations have not introduced functional discrepancies or timing violations.

- **Module-Level Directed Testing:** Each module's SystemVerilog testbench drives a small set of corner-case vectors under clock/reset, emits "Passed" or "Incorrect" flags, and is automated via `make <module>.simv`, `make <module>.out`, and `make <module>.pass`.
- **Embedded Test-Mode Diagnostics:** A `TEST_MODE` compile-time guard in the top-level RTL exposes extra debug ports (queues, tags, counters), allowing any simulation to sample internal state without waveform probing.

- **Integration-Level Simulation:** Real programs are compiled to memory images (`.c → .elf → .mem`) and loaded into the full RTL pipeline; write-back (`.wb`) and pipeline-trace (`.ppln`) outputs are diff-checked against references to catch inter-module or control-flow errors.
- **Post-Synthesis Verification:** Modules and the full pipeline are synthesized to gate-level netlists, re-simulated with the same testbenches (`make synth/<module>.vg`, `make syn_simv`, `make simulate_all_syn`), and timing-slack reports are collected to detect any synthesis-induced faults.

5.2 Testbench

During development, we employed four types of testbenches: module testers, embedded test-mode harnesses, full-pipeline simulation, and post-synthesis validation.

We began by writing **module testers** in `test/<module>_test.sv` for each RTL block. Each harness declares the DUT ports, drives a free-running clock and reset, applies a handful of hand-crafted input sequences (e.g. branch-predictor cold-miss or reservation-station full), and uses `$display` or `assert` statements to report simple "Passed" / "Incorrect" outcomes. These early checks ensured that every component's basic functionality was correct before integration.

Next, we introduced an **embedded test-mode harness** by compiling the top-level RTL with `+define+TEST_MODE`. In this mode, additional debug ports—such as cache tags, queue pointers, and prefetch counters—appear on the `pipeline` interface. Our testbench connects to these ports and logs internal signals each cycle, providing non-intrusive visibility into the processor's state during both RTL and gate-level runs.

Once module-level tests passed, we moved to a **full-pipeline simulation harness** (`test/pipeline_test.sv`). This harness instantiates the complete `pipeline`, preloads RISC-V memory images via `$readmemh`, and uses a small DPI-C monitor to record write-back (`.wb`) and pipeline-trace (`.ppln`) files. By diff-checking these outputs against reference logs, we verified inter-stage hand-offs, control-flow correctness, and data-path integrity under realistic workloads.

Finally, for **post-synthesis validation**, we reused the same simulation harness under `+define+SYNTH`, linking in the gate-level netlists generated by `synth/pipeline.vg`. Gate-level simulations confirmed cycle-for-cycle equivalence with the RTL design and allowed us to gather timing-slack metrics via our `make slack` script.

5.3 Self-defined Script Shell

To streamline testing and verification, we implemented custom shell scripts to automate simulation output comparison with ground truth.

The script `sim_test.sh` is used for testing a single program. It cleans previous simulation outputs (`make nuke`), compiles and runs a specified program (default is `dft`), and extracts memory output lines (prefixed with @@@) for comparison. If the output matches the ground truth located in `sim_ground_truth/`, the script reports a pass; otherwise, it shows the differences and marks the test as failed. The script `sim_test_all.sh` generalizes this procedure to batch test all 33 programs under the `programs/` directory (excluding `crt.s`). It maintains counters for passed and failed tests, logs detailed results to `sim_all_result.txt`, and prints a summary of all test outcomes.

The scripts `syn_test.sh` and `syn_test_all.sh` follow the same logic, but are applied to post-synthesis simulation outputs and `syn_ground_truth/` ground truth.

To further analyze performance across test programs, we developed a custom script `get_cpi.sh` that parses simulation outputs to extract key metrics such as the number of executed cycles, instructions retired, CPI (Cycles Per Instruction), and TPI (Time Per Instruction). The script accepts the clock period as a command-line argument and calculates TPI as the product of CPI and the given clock period. For each simulation result file in the `output/` directory, the script locates summary lines reporting performance statistics, extracts numerical values using regular expressions, and logs the results into a CSV file named `cpi_report.csv`.

If a program executed zero instructions (e.g., due to a simulation error), the script sets both CPI and TPI to `0.000000` to avoid division by zero. At the end of the report, an additional `Average` row is appended, summarizing the total number of cycles and instructions as well as the average CPI and TPI across all programs. This automated workflow simplifies the process of performance evaluation, allowing rapid comparisons and data visualization using tools like Excel or plotting libraries.

6 Performance Evaluation

6.1 Performance Metrics

The iron law is the best guideline for assessing a program's performance:

$$\begin{aligned} Performance &= \frac{\#Instructions}{Program} \times \frac{\#Cycles}{Instruction} \times \frac{Time}{Cycle} \\ &= CodeSize \times CPI \times ClockPeriod \end{aligned}$$

Since codesize is a parameter other than the processor, for processor performance we only consider the TPI, i.e. Time per Instruction:

$$\begin{aligned} Time/Instruction &= \frac{\#Cycles}{Instruction} \times \frac{Time}{Cycle} \\ &= CPI \times ClockPeriod \end{aligned}$$

6.2 Overall Performance

Our 3-way Out-of-Order (OoO) processor successfully operates under a **minimum clock period** of **11.2 ns** in both simulation (pre-synthesis) and post-synthesis verification. All provided testbenches execute without errors, and the processor passes **all 33** benchmark programs under both verifications.

In terms of performance, our OoO processor achieves an **average CPI (Clock Per Instruction) of 2.826372** (Figure 4), significantly outperforming the baseline in-order implementation, which reports a CPI of 5.000109. Based on the synthesizable clock constraint of 11.2 ns, our processor yields an **average TPI (Time Per Instruction) of 31.655366 ns**,

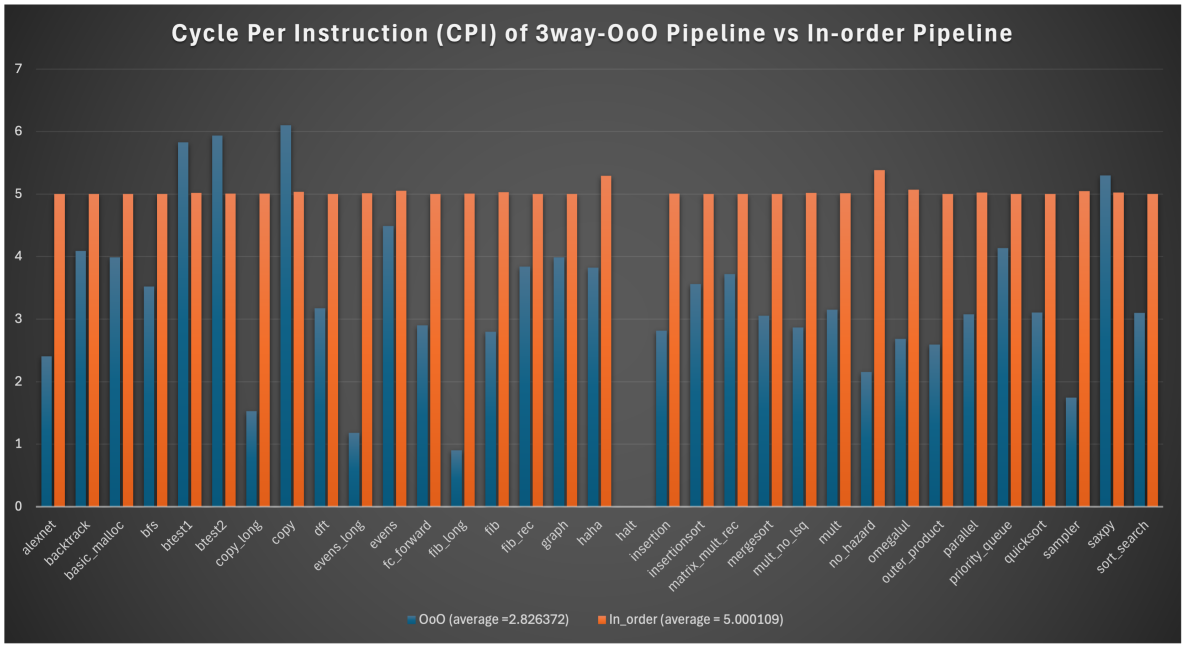

Figure 4: Cycle Per Instruction (CPl) of 3 way-OoO Pipeline vs In-order Pipeline

6.3 Prefetch Influence on Performance

In our R10K RISC-V core, the global `SYS_PREFETCH_DISTANCE` macro in `sys_defs.svh` sets how many consecutive I-Cache lines the prefetch unit issues ahead of demand. As shown by our sweep from `SYS_PREFETCH_DISTANCE=2` through `SYS_PREFETCH_DISTANCE=12`.

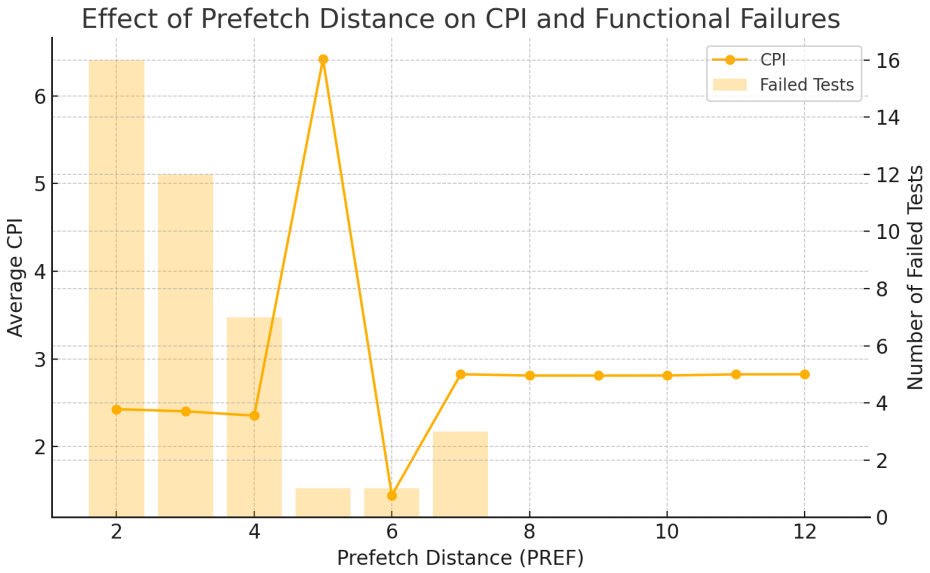

Figure 5: Impact of Prefetch Distance on CPI and Functional Test Pass Rate

First, when we increase `SYS_PREFETCH_DISTANCE` from 2 up to about 6, overall CPI falls sharply (from 2.4283 down to 1.4466). This steep decline arises because a modestly larger prefetch distance exploits multiple hardware mechanisms in concert. Sequential fetches pull in upcoming cache lines early, so the fetch stage rarely stalls waiting for memory. At the same time, issuing several outstanding line-fill requests leverages memory-level parallelism: while one cache refill is pending, the memory controller services others in flight, reducing average latency per miss. Additionally, warming the TLB with adjacent-page entries mitigates page-walk stalls for code crossing 4 KB boundaries. Finally, early prefetches allow the branch-prediction unit's speculative PC stream to prime the cache and branch target buffer, further reducing misprediction recovery cost. All these overlapping effects drive down CPI dramatically in the low-to-mid `SYS_PREFETCH_DISTANCE` range.

Second, beyond `SYS_PREFETCH_DISTANCE`8 the CPI curve flattens around 2.81 (2.8116 at 8, 2.8113 at 9, 2.8119 at 10, etc.). Here, additional prefetch aggressiveness yields diminishing or even negative returns due to resource saturation. The Miss Status Handling Registers (MSHRs), limited to N concurrent misses, become fully occupied by speculative fills, forcing genuine misses to queue until an entry frees up and increasing effective miss-service time. Our four-way associative I-Cache then suffers pollution: prefetched lines displace still-needed blocks before they are executed, so some prefetches never pay off. Excess in-flight refills also contend for reorder-buffer and load–store-queue slots in our out-of-order backend, increasing backpressure and lengthening retire-to-issue latency when branches mispredict and flush stale fetch requests. Moreover, large bursts crossing page boundaries trigger multiple TLB refill operations whose walk latency can no longer overlap with other fetches, creating isolated stalls. These interacting bottlenecks establish a "sweet spot" for `SYS_PREFETCH_DISTANCE` beyond which CPI cannot improve further.

Third, when `SYS_PREFETCH_DISTANCE` drops below about 6, functional failures appear: at `SYS_PREFETCH_DISTANCE=5` the `sort_search` test fails; at 4: seven tests fail; at 3: twelve; at 2: sixteen. Aggressive prefetch had been masking corner-case bugs in our I-Cache and prefetch logic. With insufficient look-ahead, many misses traverse the direct demand path and expose latent flaws such as off-by-one pointer-index errors or wrap-around in the prefetch queue, stale request pointers not cleared atomically on branch or pipeline flush, and MSHR tag collisions when request IDs wrap without proper modulo handling. Under conflict misses, our replacement policy can choose an incorrect way, evicting live lines that subsequent fills expect to find. Finally, on-demand fetches across

virtual-page boundaries trigger TLB-refill handshake routines that in some cases mis-sequence the response signals, causing the fetch FSM to stall indefinitely. Each of these corner-case defects can corrupt fetch ordering or hang the finite-state machine, leading to the observed sim-test failures.

6.4 Branch Predictor Influence on Performance

To evaluate the effectiveness of different branch predictors on the performance of our out-of-order processor, we conducted comprehensive experiments across 33 benchmark programs. Specifically, we compared four distinct predictors: a basic 1-bit bimodal predictor, a Gshare predictor, our implemented Tournament predictor, and a Gselect predictor.

Surprisingly, our empirical results showed that simpler predictors, particularly the 1-bit and 2-bit bimodal predictors, yielded competitive or even superior CPI performance in comparison to the more complex Tournament predictor. As illustrated in Figure 6, the average CPI when using the basic bimodal predictors hovered around 2.2–2.7, while our Tournament predictor averaged approximately 2.8, and the Gshare predictor averaged a notably higher CPI around 3.7.

We initially expected that the Tournament predictor, combining local and global prediction strategies, would outperform simpler predictors significantly. However, our results indicated otherwise. Upon careful analysis, several reasons were identified for this unexpected outcome:

First, the complexity of the Tournament predictor introduces additional latency in updating and accessing multiple prediction tables (local, global, and chooser), possibly offsetting the benefits of increased accuracy. While the prediction accuracy might be theoretically higher, the overhead in pipeline timing could lead to increased pipeline stalls or delays, thereby diminishing practical performance gains.

Second, our chosen parameters—such as predictor size (32-entry tables) and indexing strategy—might have resulted in higher aliasing or table conflicts, which undermined the theoretical advantage of having both local and global predictors. A smaller or suboptimal size for the GHR could further amplify aliasing issues, causing frequent mispredictions in correlated branches.

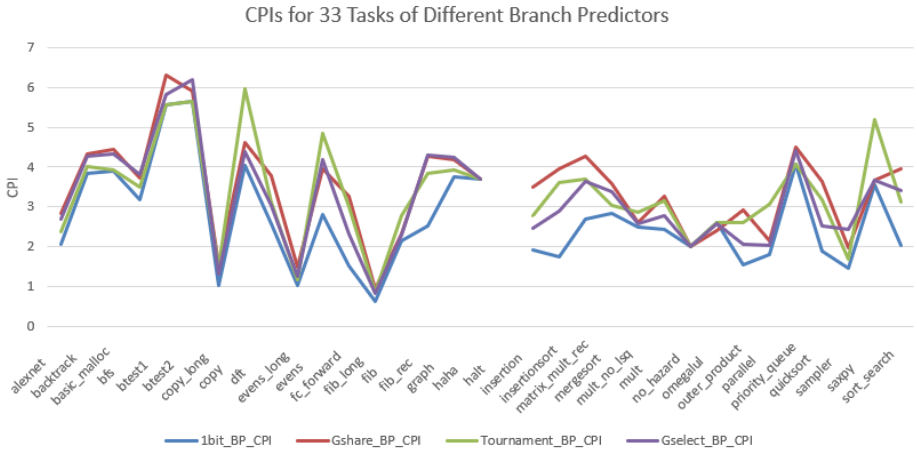

Figure 6: CPIs for 33 Tasks of Different Branch Predictors

Third, the programs in our benchmark suite might inherently favor simpler predictors due to their predominantly regular and repetitive branch behaviors. For such workloads, simple bimodal predictors can achieve high accuracy without the complexity overhead introduced by Gshare or Tournament predictors. In contrast, sophisticated predictors typically excel in workloads featuring complex branch correlations, which may not be heavily represented in our benchmarks.

Despite these findings, our Tournament predictor did demonstrate significant improvements compared to

standalone Gshare implementations, reducing the average CPI from around 3.7 to approximately 2.8. This confirms the theoretical benefit of combining predictors, as global-only approaches may perform poorly on certain predictable patterns that local predictors handle well.

In conclusion, while simpler bimodal predictors showed surprisingly strong performance on our benchmark set, the Tournament predictor still provides a valuable improvement over pure Gshare strategies. Future work could explore further tuning of predictor parameters and predictor table sizes, or incorporate dynamic adaptive strategies, to better realize the potential of Tournament branch prediction in practical processor designs.

6.5 Data Cache Influence on Performance

To evaluate the impact of cache associativity on performance, we compare our standard 2-way set-associative Dcache (16 sets × 2 ways, 8 bytes per line) against a direct-mapped Dcache of the same total size (32 lines, 8 bytes per line). Both configurations provide 256 bytes of total data capacity and use a write-back, write-allocate policy.

Our testing results in Figure 7 show that the 2-way cache implemented in our design achieves an average CPI of 2.826372, whereas the direct-mapped cache achieves 2.849599, indicating a minor performance improvement of approximately **0.02 CPI**.

In theory, a 2-way associative cache should outperform a direct-mapped cache due to its reduced conflict miss rate. However, in our case, the performance difference remains small. This could be attributed to several factors:

- **Low miss pressure in benchmarks**: Many test programs may exhibit memory access patterns with relatively few conflict misses, thereby limiting the advantage of associativity.
- **Small working set size**: The total cache size is only 256 bytes, which may be sufficient for the active memory footprint of most test cases, reducing the overall miss rate in both configurations.

Thus, while associativity helps reduce conflict misses impacts, its impact on CPI is less pronounced under the given benchmarks and cache size constraints.

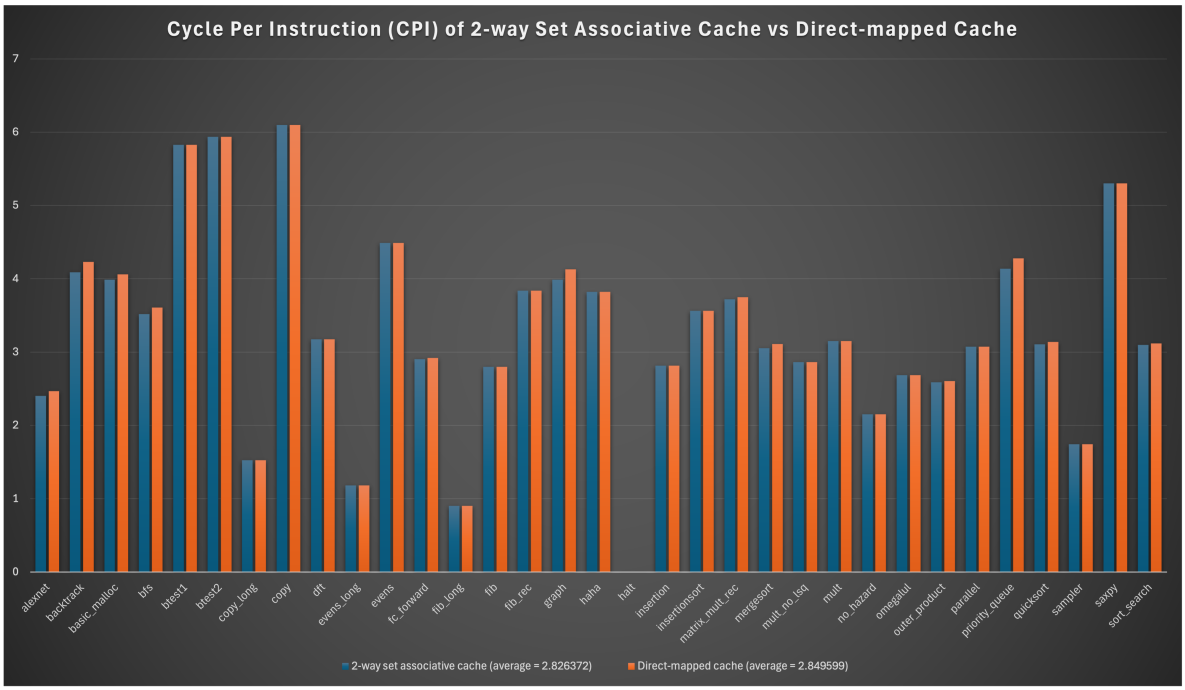

Figure 7: Cycle Per Instruction (CPI) of 2-way Set Associative cache vs Direct-mapped cache

7 Future Work

Although our current processor successfully implements many advanced architectural features, including the 3-way superscaler, instruction prefetcher, store queue, and an advanced data cache, there remain several potential areas for future improvement. Based on our experimental observations and performance analyses, we propose the following directions to enhance both processor performance and robustness in future iterations:

7.1 Branch Predictor Optimization and Early Branch Resolution

Our evaluations indicated that, contrary to expectations, simple bimodal branch predictors occasionally outperformed our more sophisticated tournament predictor. One plausible cause is the latency and complexity introduced by maintaining multiple prediction tables and the associated chooser logic. Future improvements include resizing predictor tables to mitigate aliasing conflicts, tuning indexing strategies, and reducing the complexity of update logic. Additionally, we currently lack an early branch resolution mechanism, which could significantly reduce branch misprediction penalties by resolving branch outcomes at an earlier pipeline stage. Implementing early branch resolution techniques could substantially enhance overall processor performance.

7.2 Implementation of a Load Queue

In our current design, we only implemented a Store Queue (SQ) without a corresponding Load Queue (LQ). This design choice potentially introduces additional latency, especially in cases involving load instructions awaiting memory dependencies. Future work could include adding a dedicated Load Queue to enable more efficient detection of memory dependencies, improve store-to-load forwarding, and consequently reduce unnecessary load-related stalls. This enhancement could result in significant CPI reductions by facilitating more aggressive memory-level parallelism.

7.3 Exploring Wider Superscalar Architectures

Our design currently implements a 3-way superscalar pipeline. Although this configuration successfully increased instruction-level parallelism, exploring wider superscalar architectures (such as 4-way or 6-way) could further exploit potential parallelism in modern workloads. However, increasing the superscalar width introduces challenges, including resource scaling (larger ROB, RS, and physical register files), more complex scheduling logic, and tighter timing constraints. Future designs should investigate optimal trade-offs between complexity and achievable performance to determine the most effective superscalar width.

7.4 Robustness and Efficiency of Instruction Prefetching

Our instruction prefetcher occasionally encountered simulation errors when the number of prefetched instructions fell below a certain threshold (in our case, fewer than six), likely due to boundary conditions or invalid memory accesses. Furthermore, increasing the prefetch depth beyond a certain point resulted in diminishing returns, with CPI improvements plateauing despite increased complexity. Future improvements involve implementing more robust boundary checking mechanisms to handle corner cases gracefully. Additionally, developing an adaptive prefetching strategy that dynamically adjusts prefetch depth based on real-time application behaviors could optimize performance by reducing unnecessary memory accesses and resource overhead.

In summary, future development of our processor could focus on enhancing the branch predictor with early branch resolution, incorporating a Load Queue, scaling to a wider superscalar pipeline, and improving the robustness and efficiency of the instruction prefetch mechanism. These improvements have the potential to significantly elevate processor performance, reduce CPI, and make our design closer to real-world, high-performance commercial processors.

8 Conclusion

In this project, we successfully designed and implemented a high-performance 3-way superscalar out-of-order processor inspired by the R10K architecture. The processor integrates core architectural components including a Reorder Buffer, Reservation Stations, a Physical Register File with register renaming support, and a sophisticated

branch prediction unit. Advanced features such as instruction prefetching, a tournament-style branch predictor, a 2-way set-associative data cache with MSHRs, and a store queue have been added to further improve throughput and memory performance.

Our design passed all 33 benchmark programs in both functional and post-synthesis simulations. Performance analysis revealed an average CPI of 2.826 and a minimum synthesizable clock period of 11.2 ns, demonstrating the processor's ability to deliver high instruction throughput while maintaining precise execution semantics. In particular, our investigation into branch prediction strategies revealed interesting insights: although tournament prediction theoretically offers higher accuracy, simpler bimodal predictors performed better under our specific benchmark suite, highlighting the trade-offs between complexity and practical performance.

Overall, our implementation showcases a balanced design that effectively leverages out-of-order execution, pipeline parallelism, and microarchitectural enhancements to optimize both correctness and efficiency. This project provides a strong foundation for further architectural exploration and optimization in future work.

9 Reference

[1] The MIPS R10000 Superscalar Microprocessor, https://ieeexplore.ieee.org/abstract/document/491460

10 Appendix

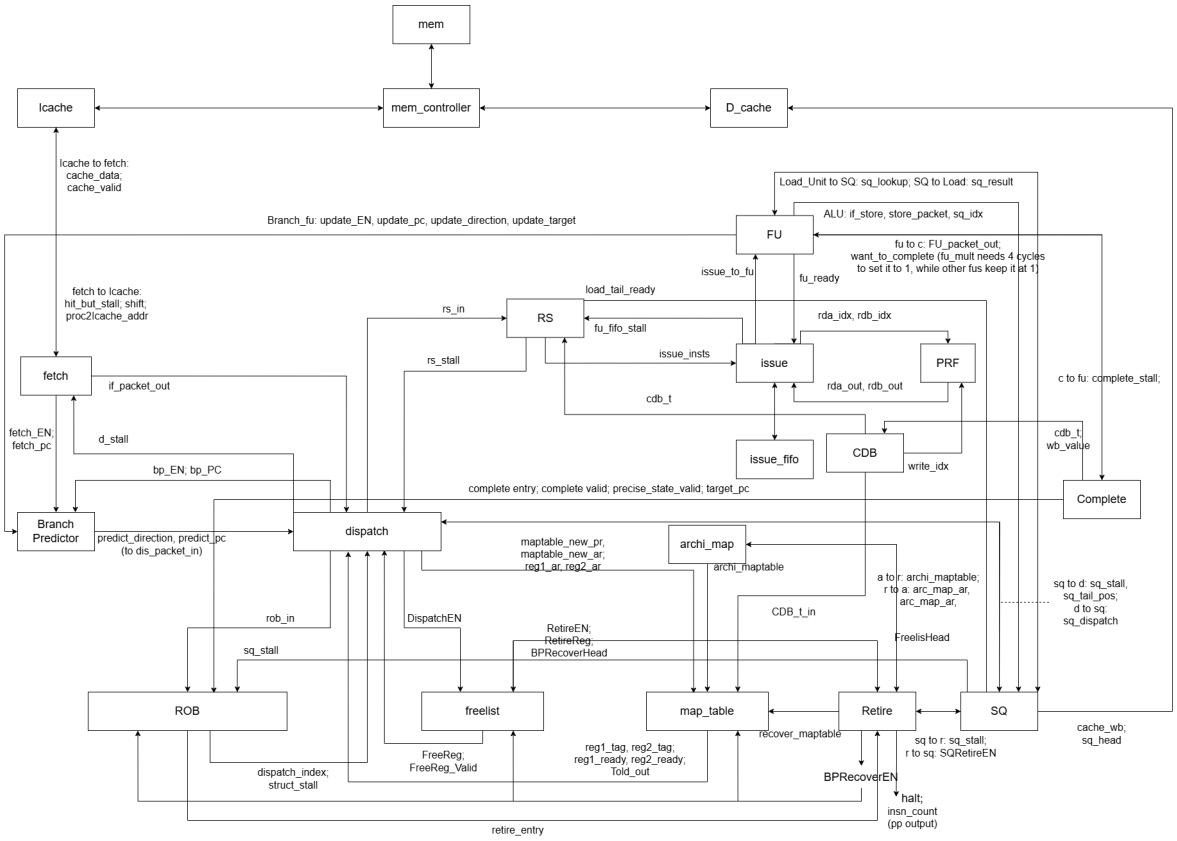

Figure 8: Block Diagram with Explicit Dataflow