An Out-of-Order RISC-V 32IM Processor with Explicit Register Renaming

Yugal Kithany, Arpan Swaroop, Om Padmani
University of Illinois Urbana-Champaign

***Abstract*—This paper presents the design and implementation of an out-of-order (OoO) microprocessor targeting the RISC-V RV32IM ISA. The processor adopts an Explicit Register Renaming (ERR) microarchitecture to maximize instruction-level parallelism while avoiding centralized Common Data Bus (CDB) contention common in Tomasulo-style designs; throughout this paper, the term CDB is used to refer generically to result broadcast mechanisms, although the implementation employs per–functional-unit result buses rather than a single shared bus. Support for integer, multiply/divide, control, and memory instructions is combined with several advanced features, including instruction and data prefetching, a G-share branch predictor with a Branch Target Buffer (BTB), and a split load-store queue. The design was developed incrementally and validated using directed tests, random instruction streams, and benchmark execution. Experimental results show measurable IPC improvements on memory- and control-intensive workloads, while also exposing important tradeoffs in area, critical path, and predictor accuracy.**

I. Introduction

Out-of-order execution is fundamental to modern high-performance processors, enabling latency tolerance and increased instruction-level parallelism (ILP). This project implements an OoO RISC-V RV32IM processor intended as a pedagogical yet realistic exploration of contemporary microarchitectural techniques. The design centers on Explicit Register Renaming (ERR), which decouples architectural and physical registers while avoiding scalability issues associated with a centralized CDB. The primary objective is correctness and performance across the full RV32IM ISA, followed by the evaluation of selected advanced microarchitectural optimizations.

II. Processor Overview

The processor implements a fully out-of-order pipeline with register renaming, dynamic scheduling, and in-order retirement. It supports the complete RV32IM ISA, including integer arithmetic, multiplication and division, memory operations, and control-flow instructions. Development was structured across three checkpoints to ensure functional correctness before introducing advanced features.

Checkpoint 1 focused on instruction delivery, including the fetch stage, instruction cache, cache-line adapter, and instruction queue. Checkpoint 2 introduced the core OoO mechanisms, including rename, reservation stations, execution units, and the Reorder Buffer (ROB). Checkpoint 3 completed the design by adding memory and control support, including the load-store queue and branch handling logic. This staged approach reduced integration risk and allowed systematic validation at each step.

A high-level block diagram of the ERR-based datapath is shown in Appendix Figure 3.

III. Microarchitectural Design

A. RTL-Level Implementation Details

At the top level, the CPU integrates fetch, rename/dispatch, execution, and retirement using explicit valid/ready signaling. Rather than enforcing fixed pipeline stages, modules communicate through queues and reservation stations, allowing fetch and execution to proceed independently when downstream pressure exists. This decoupling is visible in the fetch logic, which maintains its own enqueue/dequeue control and can stall independently of rename or execution.

The fetch unit implements an instruction queue with head-/tail pointers and full/empty detection. It supports speculative fetch using both a BTB and a G-share predictor. BTB hit qualification is gated by predictor validity, and mispredictions trigger explicit flush control that resets the instruction queue and suppresses enqueues during recovery. The fetch logic also tracks memory wait state and includes simple performance counters and latency histograms to quantify instruction fetch response time under memory stalls.

Branch prediction state is updated using non-speculative branch outcomes provided by the ROB, ensuring predictor correctness under rollback. G-share prediction outputs both direction and index metadata, which are later consumed during commit for table updates. This design choice simplifies recovery logic by avoiding speculative predictor writes.

Reservation stations are partitioned by functional unit type (ALU, multiply, divide, control, and load/store), each implemented as a fixed-size array of entries. Each entry stores decoded instruction metadata, destination physical register ID, architectural destination, source physical register tags, readiness bits, immediate flags, and ROB index. Dispatch logic selects the appropriate station based on opcode and funct fields, ensuring structural hazards are localized per functional unit.

Operand readiness is resolved via explicit RAT update broadcasts. The reservation station listens to up to five concurrent physical register completions, comparing broadcast tags against stored source tags and marking operands ready when matches occur. Issue logic selects ready entries only when the corresponding functional unit signals availability, preserving correctness while enabling limited out-of-order execution. Commit logic retires instructions strictly in order via the ROB,

and physical registers are reclaimed only after architectural state is guaranteed precise, enabling reliable recovery from branch mispredictions.

B. Design Rationale

Several architectural choices were driven by scalability, correctness, and implementation complexity. Explicit Register Renaming (ERR) was selected over Tomasulo-style renaming to avoid centralized Common Data Bus (CDB) bottlenecks and excessive multi-ported structures. By allocating dedicated result buses per functional unit, the design reduces wakeup latency at the cost of increased area, a tradeoff deemed acceptable for improving IPC and simplifying scheduling logic.

Early integration of the instruction cache and cache-line adapter was a deliberate design choice to minimize late-stage integration risk. This allowed memory-related corner cases to be identified early and reduced coupling between fetch and downstream pipeline stages. Modular cache-line adapter design also enabled later experimentation with instruction and data prefetchers.

C. Fetch and Instruction Delivery

The fetch stage interfaces with a DRAM model through an instruction cache and a modular cache-line adapter that supports burst accesses. A parameterizable circular instruction queue buffers fetched instructions and decouples fetch from downstream pipeline stages. The queue design includes head and tail pointers with wraparound tracking to correctly handle full and empty conditions.

D. Rename, Dispatch, and Scheduling

Instructions are renamed and dispatched using an ERR scheme. Destination registers are allocated from a free list and mapped via a Register Alias Table (RAT). Instructions are placed into reservation stations associated with specific functional units. Separate reservation stations per functional unit were chosen to simplify issue logic and reduce arbitration complexity. This design improves clarity and correctness but increases the number of structures that must be maintained.

E. Execution and Writeback

The execution stage includes ALU/CMP, multiply, divide, control, and load/store functional units. Synopsys-provided sequential multiply/divide IPs were integrated to reduce critical-path delay and improve timing closure. An early implementation challenge involved handling divide-by-zero behavior, which required explicit exception handling to maintain architectural correctness.

Each functional unit broadcasts results on a dedicated result bus, reducing structural hazards during writeback. While area-intensive due to the required read/write ports, this approach eliminates result serialization and improves wakeup responsiveness.

F. Reorder Buffer and Retirement

The ROB enforces in-order retirement and maintains precise state. A Retirement RAT (RRAT) tracks the last committed architectural state, enabling recovery from mis-speculation. Physical registers freed at commit are returned to the free list. Maintaining both RAT and RRAT was critical for correctly handling control-flow instructions and future branch recovery support.

G. Reorder Buffer Organization

The processor employs a centralized reorder buffer (ROB) with 16 entries to enforce in-order retirement and precise architectural state. Each ROB entry tracks destination architectural and physical registers, completion status, and control-flow metadata for branch instructions. Execution units signal instruction completion out of order via ROB indices, while retirement proceeds strictly in program order from the head of the ROB. For branch instructions, the ROB records prediction metadata at dispatch and performs non-speculative updates to the branch predictor structures at commit.

The core is single-issue, allowing at most one instruction to be renamed and dispatched per cycle, subject to reservation station and ROB availability. Reservation stations are statically partitioned by functional unit, with completion signaled via dedicated result buses.

IV. Memory and Control Support

A. Load-Store Queue

A split load-store queue allows loads to execute out of order with respect to stores, subject to memory ordering constraints. Stores are prioritized when ready and at the head of the ROB, while independent younger loads may bypass stalled stores. This design improves memory-level parallelism at the expense of increased queue logic complexity.

B. Control Flow Handling

Control-flow instructions, including conditional branches and jumps, are executed using a dedicated control functional unit. In the baseline configuration used during early development checkpoints, branches were statically predicted as not taken, and any misprediction triggered a pipeline flush and recovery using the reorder buffer to restore precise architectural state. This baseline design served as a correctness reference and enabled early validation of branch resolution and recovery logic.

In the final integrated design, static prediction is superseded by the dynamic branch prediction mechanisms described in Section V.D. Speculative fetch and control-flow redirection are enabled only when dynamic prediction is active, and all performance results reported in this paper reflect the use of the G-share predictor and Branch Target Buffer (BTB) unless otherwise stated.

V. Advanced Features

A. Implementation Notes

From an RTL perspective, advanced features were integrated incrementally to minimize destabilization of the core pipeline. Prefetchers and branch predictors interface with fetch through narrow, well-defined control signals, allowing them to be enabled or disabled without modifying core execution logic. Predictor state is updated at commit rather than at execute time, simplifying correctness under speculation.

B. Next-Line Instruction Prefetcher

The next-line prefetcher was introduced to reduce instruction fetch stalls caused by compulsory and capacity misses. Positioned between the fetch stage and instruction cache, it speculatively requests a future cache line (PC + offset) when the instruction queue is not full. A key design challenge was ensuring that prefetch requests did not block demand requests; this was addressed by temporarily buffering memory responses until both requests completed. Measured accuracy of approximately 82% led to reduced average instruction fetch latency on sequential workloads, though increased delay was observed due to imperfect prediction.

C. Stride Data Prefetcher

A stride-based data prefetcher was explored to accelerate workloads with regular memory access patterns, such as mergesort. While functionally correct, the prefetcher suffered from low accuracy (34%), causing unnecessary memory traffic and increased AMAT. Attempts to tune the base stride and update policy did not sufficiently improve behavior, and the feature was excluded from the final integrated design. This highlighted the importance of prefetch accuracy over aggressiveness.

D. G-share Branch Predictor with BTB

The processor employs a dynamic branch prediction mechanism that combines a G-share direction predictor with a direct-mapped Branch Target Buffer (BTB) to enable speculative instruction fetch. The G-share predictor uses a global history register (GHR) and a pattern history table (PHT) composed of 2-bit saturating counters, indexed by the XOR of selected program counter bits and the global history. The BTB provides speculative target addresses for taken branches and is indexed independently using the branch program counter.

During instruction fetch, the BTB and G-share predictor are accessed in parallel. A BTB hit supplies a predicted target address, while the G-share predictor provides a taken/not-taken decision. Speculative redirection of the fetch stream occurs only when both the BTB reports a valid entry and the G-share predictor predicts the branch as taken; otherwise, fetch proceeds along the sequential path. This gating prevents redirection on direction-only predictions without a valid target.

To ensure correctness under speculation, predictor structures are updated non-speculatively at commit. For each branch instruction, prediction metadata—including the G-share prediction outcome, pattern history table index, and BTB hit and target information—is recorded in the reorder buffer at dispatch. When the branch reaches the head of the reorder buffer and its actual outcome is known, the predictor tables are updated using the recorded metadata. The predictor structures themselves (BTB, BTB valid bits, and PHT) are implemented as dedicated SRAM arrays and are not checkpointed or modified speculatively.

Experimental results indicate that while the G-share predictor achieves high direction prediction accuracy in isolation, overall speculative effectiveness is frequently limited by BTB coverage. Workloads with regular control flow benefit from high BTB hit rates, whereas applications with indirect branches or infrequent branch reuse experience reduced speculative gains due to BTB misses. These observations highlight the importance of balanced direction prediction accuracy and target prediction coverage, and motivate future extensions such as increased BTB capacity, associativity, or the addition of a return address stack (RAS).

E. Split Load-Store Queue

The split load-store queue improves memory-level parallelism by allowing loads to bypass older, unresolved stores when safe. Loads are issued out of order based on readiness and relative age, while stores execute only when at the ROB head. This design improved average load latency and IPC for memory-intensive workloads but increased critical-path complexity due to additional comparison logic.

VI. Evaluation Methodology

Correctness was verified using directed unit tests, random instruction streams, and benchmark execution with RVFI and Spike reference logs. Performance was evaluated using Coremark, FFT, mergesort, and AES/SHA workloads. IPC, execution latency, cache wait times, and predictor accuracy were collected via performance counters and post-processing scripts.

VII. Results

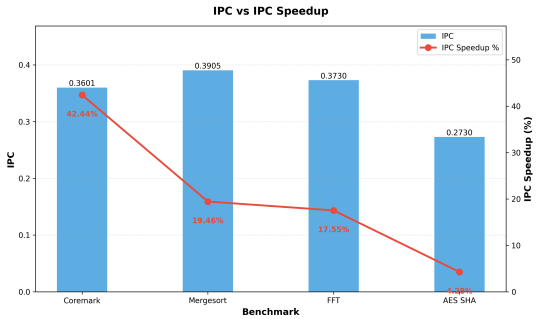

Fig. 1: IPC and IPC speedup with BTB + G-share branch prediction.

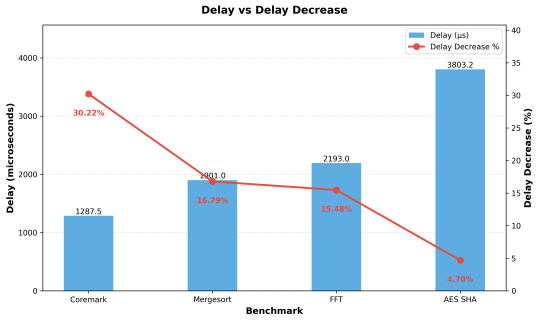

Fig. 2: Execution delay and delay reduction with BTB + G-share branch prediction.

The ERR-based OoO core demonstrates consistent IPC improvements over an in-order baseline when sufficient

instruction-level parallelism is present. Memory-intensive workloads benefit from reduced load latency due to the split load-store queue, while loop-heavy programs see gains from speculative execution enabled by branch prediction.

TABLE I: Baseline Core Performance

Benchmark	IPC	Delay (μs)
Coremark	0.2528	1845.12
Mergesort	0.3269	2284.56
FFT	0.3173	2594.53
AES/SHA	0.2618	3990.69

TABLE II: Performance with BTB and G-share Branch Prediction

Benchmark	IPC	Delay (μs)	IPC Speedup	Delay Reduction
Coremark	0.3601	1287.49	42.44%	30.22%
Mergesort	0.3905	1900.96	19.46%	16.79%
FFT	0.3730	2192.96	17.55%	15.48%
AES/SHA	0.2730	3803.17	4.28%	4.70%

TABLE III: Branch Prediction Performance

Benchmark	G-share Acc. (%)	Speculative Acc. (%)	BTB Hit Rate (%)
Coremark	87.82	77.55	49.93
Mergesort	77.38	68.14	44.49
FFT	99.04	64.76	78.26
AES/SHA	95.72	18.74	27.09

TABLE IV: Impact of Split Load-Store Queue

Metric	Coremark_im	Mergesort
IPC Improvement (%)	1.21	3.26
Avg. Load Time Reduction (%)	11.72	20.91

Instruction prefetching reduces fetch-related stalls on sequential code, though imperfect accuracy introduces additional delay. Branch prediction results show high G-share accuracy in isolation, but overall speculative effectiveness is constrained by BTB miss rate. These results highlight the importance of balanced predictor design.

Overall performance improvements are workload-dependent, emphasizing that microarchitectural optimizations must be carefully matched to application behavior.

VIII. Lessons Learned and Limitations

Several important insights emerged during the design and evaluation process. First, architectural scalability must be considered early; while multiple CDBs improve wakeup latency, they significantly increase area and routing complexity. Second, predictor accuracy is more critical than predictor aggressiveness, as demonstrated by the stride prefetcher degrading performance despite additional hardware.

The branch prediction subsystem revealed that BTB capacity and organization can dominate speculative effectiveness, even when the underlying direction predictor performs well. Additionally, features such as a return address stack (RAS) are often more impactful than general-purpose BTBs for common control-flow patterns in real software.

From an implementation perspective, early performance instrumentation proved invaluable for identifying bottlenecks and guiding design decisions. However, limited synthesis support for the baseline core constrained direct area comparisons. Future iterations would benefit from deeper integration of memory-side optimizations, reduced structural duplication, and more aggressive sharing of predictor and scheduling resources.

IX. Conclusion

This work demonstrates a complete OoO RISC-V RV32IM processor using Explicit Register Renaming and several modern microarchitectural enhancements. The design balances performance, complexity, and correctness, and serves as a practical study of OoO execution tradeoffs. Future work includes improving BTB coverage, integrating a return address stack, refining prefetch accuracy, and reducing area through more aggressive resource sharing.

References

[1] RISC-V Foundation, 'The RISC-V Instruction Set Manual," 2019.
[2] R. Tomasulo, 'An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM JRD, 1967.
[3] EEMBC, "CoreMark Benchmark," 2023.

Appendix A
RTL and Evaluation Details

This appendix provides supplementary RTL-level information and additional evaluation artifacts that support the architectural discussion in the main body of the paper. The material here is included for completeness and reference.

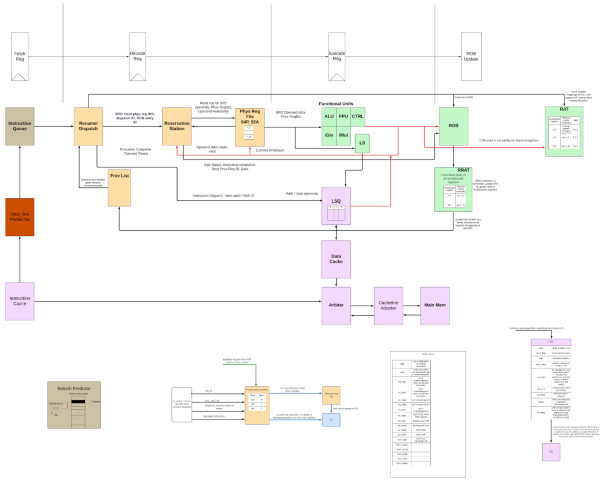

Fig. 3: RTL block-level overview of the out-of-order core