MP_OOO Report

Group: I_Wanna_Drop

Members: Chiming Ni, Siying Yu, Kongning Lai, Hengjia Yu

1. Introduction

This project implements an out-of-order RISC-V processor using Explicit Register Renaming (ERR). In modern computers, improvements in processor performance depend not only on faster clock frequencies, but more importantly, on increasing instruction-level parallelism (ILP). Out-of-order execution is a key technology in achieving this goal, breaking the constraints of sequential program execution by allowing instructions to execute as soon as their operands are ready. Our implementation uses ERR rather than Tomasulo's algorithm to achieve better scalability and reduced complexity in the register management system. The project brings together what we've learned in our computer architecture class, giving us a better understanding of how modern CPUs are designed and optimized.

In this report, we will provide an overview of our processor, followed by a detailed description of the progress we made at each checkpoint. We will then discuss the motivation, implementation, and performance analysis of the advanced features. Finally, we will conclude with a summary of the project.

2. Project Overview

Our project involved the design and implementation of a 2-way superscalar out-of-order microprocessor supporting the RV32IM instruction set. The key architectural features of our processor include explicit register renaming with a 64-entry physical register file, 2-way superscalar execution with dual instruction fetch, dispatch, and commit, distributed reservation stations, split load/store queues, a post-commit store buffer, CDB and ALU bypasses for back-to-back ALU operation execution, age-ordered issue scheduling, and GShare branch prediction with an associative BTB.

In terms of organization, our team divided the work early in the project. Each member was responsible for different modules of the processor, and we carefully designed the interfaces and handshake protocol between them to ensure smooth integration later on. As we moved into the implementation of advanced features, each team member took ownership of specific enhancements.

One of our notable achievements was winning first place in a competition among 28 teams, which showed the strong teamwork, technical skills, and hard work we put into the project.

3. Design Description

(a) Overview

During project development, we progressed through four main phases: (i) In checkpoint 1, we established the basic infrastructure and frontend components. (ii) In checkpoint 2 we focused on implementing core pipeline components including the out-of-order execution units. (iii) Checkpoint 3 integrated units such as the branching handler and memory subsystem. (iv) After CP3 we implemented advanced features such as branch predictor, split load/store queue, bypass network, completing the processor's functionality for high-performance instruction execution.

(b) Milestones

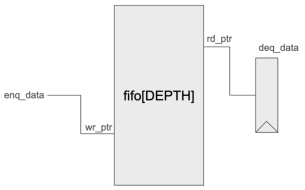
i. Checkpoint 1

In CP1, we implemented basic frontend components and key infrastructure. The frontend design includes a fetch unit and cacheline adapter supporting scalar instruction fetch. We implemented

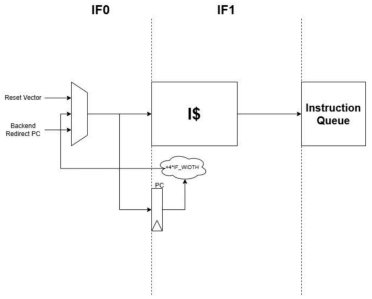
FIFO buffers for instruction flow management. For testing, we not only wrote testbenches for the FIFO queue and cacheline adaptor, but also integrated our fetch into mp_pipeline to ensure it functions correctly. After careful consideration we decided to develop an ERR-style microarchitecture over Tomasulo due to its better scalability and ability to achieve higher clock frequency, essential for our superscalar design. Then we completed block diagrams for the key design of our ERR datapath including frontend, decode/rename/dispatch, issue/execute/write back and ROB, laying the groundwork for subsequent implementation.

Instruction Queue (FIFO) w/ bit-extension technique

```
always_ff @(posedge clk) begin
    if (rst) begin
        wr_ptr <= '0;
        rd_ptr <= '0;
    end else begin
        if (enq_en && ~full) begin
            fifo[wr_ptr_actual] <= enq_data;
            wr_ptr <= (ADDR_IDX+1)'(wr_ptr + 1);
        end
        if (deq_en && ~empty) begin
            deq_data <= fifo[rd_ptr_actual];
            rd_ptr <= (ADDR_IDX+1)'(rd_ptr + 1);
        end
    end
end
assign empty = (wr_ptr == rd_ptr);
assign full = (wr_ptr_actual == rd_ptr_actual) &&
(wr_ptr_flag == ~rd_ptr_flag);
```



Frontend



ii. Checkpoint 2

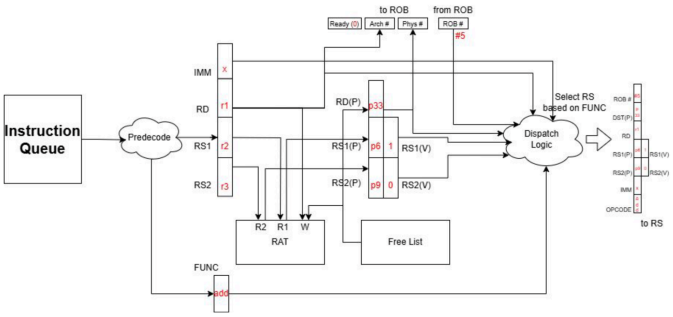
In CP2 we focused on implementing the processor's backend components (except Store and Branch). We completed the decode, rename, and dispatch logic, and implemented distributed Reservation Stations (RS). Additionally, we created a separate Mult/Div RS and utilized DW_mult and DW_div modules to integrate multiplication and division operations for RV32M extension. We implemented the Reorder Buffer (ROB), Register Alias Table (RAT), Renamed Register File (RRF), and Free List. To facilitate collaborative development, we defined the interfaces between different modules before starting implementation and adopted a valid-ready handshake mechanism across all modules. Additionally, although we have not yet implemented advanced features, we have already considered superscalar adaptability during development, laying a solid foundation for future progress.

Decode, Rename, Dispatch

Decode: We extract instruction fields (opcode, funct3, funct7) and generate control signals (fu_type, fu_opcode, etc.) to determine the functional unit and operation. Immediate values are decoded, and architectural registers are identified with a default fallback to `r0` if unused. This stage maps instructions to reservation station (RS) types and functional units for downstream processing.

Rename: We use a free list (FL) to allocate physical registers and a register alias table (RAT) to handle source register mappings, ensuring correct dependency tracking. Renamed uops are enriched with physical register mappings (rs1_phy, rs2_phy, rd_phy) and forwarded to the reorder buffer (ROB) for tracking instruction order and recovery.

Dispatch: We route *uops* to appropriate reservation stations (INT, INTM, BR, MEM) based on their rs_type. A valid-ready handshake mechanism ensures synchronization, and we handle backpressure to prevent pipeline stalls.



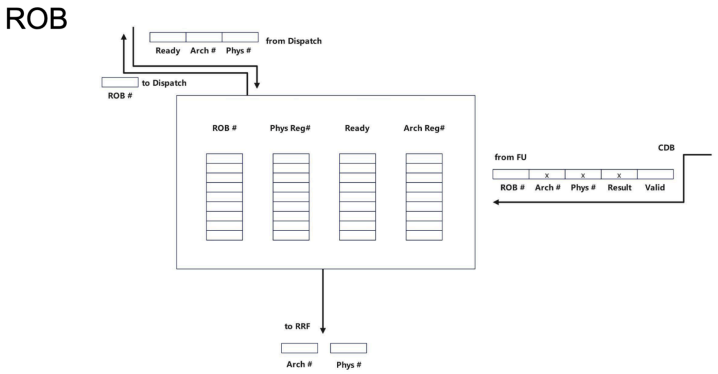
ROB, RAT, RRF, Free List:

ROB: The ROB ensures program order retirement and supports speculative execution recovery. In our design, the ROB tracks each instruction's architectural destination (*rd_arch*) and physical destination (*rd_phy*). It integrates with the RRF for committing final register states and triggers resource reclamation (via the FL) upon instruction commit.

RRF: The RRF maintains the committed architectural state of the processor. In our implementation, it receives updates from the ROB during commit and interfaces with the RAT to provide architectural register fallback values during misprediction recovery. This ensures the architectural state is always correct and consistent, serving as the processor's golden source of committed data.

RAT: The RAT maintains mappings between architectural and physical registers to track dependencies. In our implementation, it provides source register mappings (*rs1_phy*, *rs2_phy*) and updates destination register mappings (*rd_phy*) during renaming. The RAT ensures correctness by distinguishing between valid and invalid physical registers and is tightly coupled with the FL to dynamically allocate destination physical registers. Our design supports simultaneous multi-instruction renaming, ensuring scalability for superscalar pipelines.

Free List: The Free List tracks available physical registers and ensures efficient allocation and reclamation. In our implementation, physical registers are allocated dynamically to instructions that have a valid destination register (*rd_arch*), with unused registers being released when instructions retire.



Issue, Execute, Write back (Reservation Station):

Reservation Stations (RS): The reservation stations act as buffers for instructions waiting to be issued to functional units. In our implementation (int_rs_normal.sv), reservation stations:

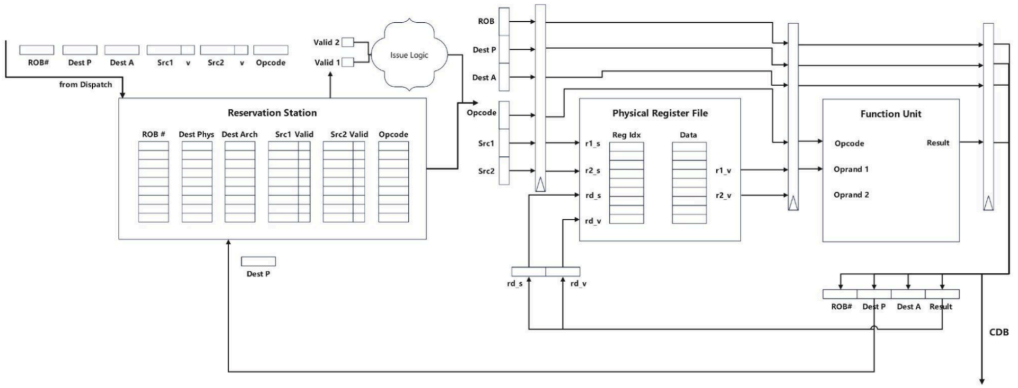
- Store instructions and their operands until all dependencies are resolved.
- Track operand readiness through a valid-ready mechanism, with inputs coming from the register file (prf.sv) or forwarded results from the execution units.

- Issue ready instructions to the functional units based on availability, ensuring out-of-order execution.

For scalability, we implemented distinct RS modules for each functional unit type.

Functional Units (Execute): Functional units, such as the ALU (fu_alu.sv), execute instructions dispatched from the reservation stations. Our functional units are pipelined to maintain throughput and enable parallel execution of multiple instructions. The function units also support multi-cycle operations, with backpressure signals to synchronize with the reservation stations.

Write-Back: Results from the functional units are written back to the PRF and forwarded to the reservation stations to resolve dependencies for waiting instructions.



Mult/Div Integration:

We integrated multiplication and division operations with specialized modules.

Dedicated Reservation Station (RS): Multiplication and division instructions are routed to a specialized reservation station (*intm_rs*), to handle the latency and operand dependency challenges specific to mult/div operations. It supports multi-cycle instruction tracking, ensuring that instructions remain in the reservation station until execution completes or results are ready for forwarding. Also, Dependencies for mult/div operations are managed independently to avoid blocking other integer instructions.

Testing

For correctness testing, we not only unit-tested each module, but also ported in some assembly programs without branch and memory operations.

iii. Checkpoint 3

In CP3 we completed the memory subsystem and branch prediction implementation. We developed a complete memory subsystem including Load/Store Queues (LSQ) and cache hierarchy. The branch handling unit implementation supports speculative execution, while the BMEM arbiter ensures proper coordination of memory accesses. This phase's work integrated all components together, building a fully functional out-of-order processor capable of efficiently handling control flow and memory operations.

Memory Subsystem

Our memory system efficiently handles out-of-order memory operations using a reservation station (RS), an Address Generation Unit (AGU), and a Load/Store Queue (LSQ). Instructions are dispatched to the RS, where their operands are dynamically updated by snooping on the Common Data Bus (CDB). Once operands are ready, the RS issues instructions to the AGU, which calculates the effective memory address using agu_reg.rs1_value and agu_reg.imm. For store

instructions, the AGU generates write data (*to_lsq.wdata*) and a write mask (*to_lsq.mask*) based on the instruction type, such as MEM_SB, MEM_SH, or MEM_SW.

The LSQ enforces in-order memory operations by maintaining a FIFO structure (fifo[LSQ_DEPTH]) to track memory requests. Load instructions issue memory reads as soon as their addresses are computed, with data responses forwarded to the CDB via cdb_out.rd_value. Store instructions wait until they reach the head of the LSQ and the ROB to ensure correctness before committing to memory. The enqueue and dequeue logic ensures synchronization: instructions are added to the LSQ when dispatched (enqueue logic) and removed after their memory operations complete (dequeue logic).

Branches

Our design employs specialized branch reservation stations that monitor both CDB and ALU bypass networks, enabling efficient tracking of control dependencies and quick branch resolution. These stations work in conjunction with a dedicated control buffer that tracks branch instructions from dispatch to commitment, maintaining prediction information and actual outcomes for precise control flow recovery. At the heart of our branch execution is a specialized functional unit that handles all control instructions, including conditional branches (BEQ, BNE, BLT, BGE, BLTU, BGEU), unconditional jumps (JAL, JALR), and PC-relative addressing (AUIPC). For each branch instruction, this unit computes both the actual outcome and target address, comparing them against predicted behavior to detect mispredictions. Upon detecting a misprediction, the branch result is communicated to both the control buffer and ROB, enabling precise recovery through a backend flush and fetch redirection when the branch commits.

BMEM arbiter

Our BMEM arbiter connects the instruction cache (i-cache), data cache (d-cache), and main memory adapter, managing memory requests and resolving conflicts efficiently. It prioritizes d-cache requests because i-cache only performs reads.

In the **PASS_THRU** state, the arbiter forwards requests from the caches to the adapter, with d-cache getting priority. For d-cache writes, it stores the address and data in buffers and switches to the **WAIT_WRITE** state until the write is complete. Once memory operations finish, the arbiter forwards the responses (*rdata, rvalid, raddr*) to both caches, ensuring smooth and fast communication. This design keeps the system simple and efficient.

(c) Advanced Design Options

i. 2-Way Superscalar

A. Design

We implemented 2-way superscalar for our processor. This means 2-way fetch, 2-way decode, 2-way dispatch and also 2-way commit.

Some simplifications were made to make the circuit more viable. The dispatch stage only allows at most one branch and one memory instruction to be dispatched at a time. The motivation comes from the fact that there are rarely two branch instructions present in a fetch packet and the memory subsystem cannot achieve a throughput higher than 1. Therefore it is not worth it to implement 2-way dispatch of branch and memory instruction since the area/timing cost is too high to support simultaneous dispatch of multiple branch instructions.

The ROB is also banked to make dispatch and commit simpler. The 32-entry 2-wide banked ROB enjoys the similar timing as a 32-entry normal ROB but can store up to 64 instructions in-flight. The tradeoff is the possibility of suffering from fragmentation. But since it’s not very likely to have two branch instructions or memory instructions in a fetch packet, the cost is minimal and the

benefit of having a much larger ROB (twice the size) outweighs the cost.

B. Testing

Since superscalar improves the instruction-level parallelism, the only meaningful metric is IPC (Instruction Per Cycle). IPC is a good indicator of how much ILP is exploited since the ILP is the upper bound of IPC. If implemented correctly, we should be expecting an increase in IPC and the IPC is no longer limited by 1 now.

C. Performance Analysis

IPC	Coremark	FFT	AES_SHA	Mergesort	Compression
w/ SS	0.758571	0.945852	0.560007	0.842045	1.165134
wo/ SS	0.630426	0.724554	0.469626	0.701197	0.987223

Superscalar provides a steady increase to IPC across all benchmarks. Notably, the compression test case shows IPC more than 1, which is a big surprise to us. This is mainly due to high memory-level parallelism and very predictable branch behavior.

ii. Branch Predictor and Branch Target Buffer (BTB)

A. Design

We implemented a GShare branch predictor with associative BTB to improve the accuracy of branch predictions and reduce the high number of pipeline flushes caused by our initial static predictor.

The GShare predictor combines the program counter (PC) and the global branch history register (GHR) by XORing their lower bits to generate an index for the prediction table, reducing aliasing in the 2-bit saturating counters. For target address prediction, we implemented two separate branch target buffers (BTBs): one for br and another for jal. To improve the accuracy of unconditional jump predictions, if a matching PC is found in the jal BTB, the instruction is immediately predicted as taken, overriding the GShare result.

When adapting the predictor for a superscalar design, we ensure predictions are generated for all PCs in the fetch bundle. This includes verifying the validity of each instruction in the bundle, as invalid instructions could interfere with prediction accuracy.

B. Testing

Since the branch predictor is designed to reduce the misprediction rate and improve IPC, particularly for branch-heavy benchmarks, the testing primarily focuses on evaluating its impact on these metrics. We use a generate statement to enable/disable the branch predictor, and use performance counters to calculate the branch mispredict rate.

The performance counter is integrated on the branch `bp_profile`. Modify BP_ON in pkg/types.sv to enable/disable branch prediction.

C. Performance Analysis

IPC	Coremark	FFT	AES_SHA	Mergesort	Compression
w/ BP	0.758571	0.945852	0.560007	0.842045	1.165134
wo/ BP	0.508645	0.776496	0.563092	0.702822	0.740097

mispredict_rate	Coremark	FFT	AES_SHA	Mergesort	Compression
w/ BP	20.78%	27.75%	79.05%	30.15%	17.59%

wo/ BP	61.04%	99.03%	76.28%	42.08%	47.46%

Overall, branch prediction has a clear positive impact on our processor's performance. However, we believe there is still room for improvement in prediction accuracy. Currently, our branch predictor is implemented using flip-flops; in the future, we may explore more sophisticated branch prediction algorithms and leverage SRAM to further enhance performance.

iii. Split Load/Store Queue and Post-commit Store Buffer

A. Design

The store queue is implemented as a FIFO, just like the previous unified Load/Store queue. The load queue is more like a reservation station, it issues load entries out-of-order and removes the issued entry directly when DCache is ready. The post-commit store buffer is also a FIFO, and is almost the same as the store queue, the only difference is that the store buffer will never be flushed.

The conflict resolution of the load queue is implemented as follows. Each load queue entry keeps track of the number of store instructions before it. (This is easily done by a saturating counter that decrements whenever the store queue dequeues). During each cycle, the load queue will scan the store queue and the store buffer for potential conflicts. A potential conflict means: `raddr` and `waddr` are the same and `wmask` overlaps with `rmask`.

Store forwarding is not implemented as it adds too complicated logic and has minimal performance gain in the benchmarks in this MP.

B. Testing

The performance of the memory subsystem is hard to measure. Of course IPC can capture its performance, but IPC is also affected by many other components and may not be a good metric for the memory subsystem if the memory is not the bottleneck (e.g. when branch prediction is very bad).

A better metric is to track how long it takes for the memory subsystem to process a load/store request. This can be done by attaching a starting time for each memory instruction and tracking the number of cycles it takes to dequeue that instruction. We call them “# of cyc to store” and “# of cyc to load”. They reflect how fast the memory subsystem can process each load/store request. If our implementation is correct, these two metrics should both go down significantly when the advanced feature is integrated.

This performance counter is integrated on the branch `profile`. Checkout commit `9531c6b` and `31c4f32` to get the patch.

C. Performance analysis

	Coremark	**FFT**	**AES_SHA**	**MergeSort**	**Compression**
# cyc to store	9.99 | 6.25	20.57 | 10.30	20.61 | 13.84	13.32 | 8.74	11.57 | 6.89
# cyc to load	6.66 | 3.60	12.02 | 5.80	18.18 | 7.955	10.12 | 6.71	6.29 | 3.78
IPC	0.39 | 0.46	0.48 | 0.59	0.43 | 0.48	0.52 | 0.58	0.48 | 0.59

A split load/store queue design can significantly speed up the memory subsystem, we can see the response time almost halved in some test cases (FFT, mergesort). A small increase in IPC is also observed, further confirming that this feature is beneficial to the overall performance.

iv. Age-Ordered Issue Scheduling

A. Design

We implemented age-ordered issue scheduling to maintain program order when possible while allowing out-of-order execution. This approach reduces the occurrence of pipeline bubbles. Since older instructions have a higher possibility to be the dependency for following instructions, issues are prioritized by the age of the instructions.

A compressing issue queue is used to maintain the instruction order. All newly dispatched instructions would be pushed to the bottom of the queue. Each time any entry is issued from the queue, all entries below would move forward to ensure there's no empty space in the middle of the queue. A pointer is maintained to keep track of the bottom of the queue. Each queue entry uses a multiplexer to pick its input, either from dispatch or the entry below.

The issue arbiter prioritizes upper entries in the issue queue. The circuit would first scan all entry requests, generate a *prev_assigned* signal, and then issue entries based on the index of the entry.

The scheduling strategy mainly makes a difference in the issue queue update circuit. It requires more area for the multiplexers and consumes more power to update entries when an earlier one is issued. Also since the push pointer is also determined by issue logic, it makes the dispatch path longer. The path from ds_stage to int_rs dispatch was once the critical path, and we later optimized it from the dispatch stage side.

B. Testing

Age-ordered scheduling adjusts issue priorities. This design is based on the assumption that older instructions are usually more critical, but it doesn't guarantee performance improvements in all cases. While we expect most benchmarks to see higher IPC, occasional small drops are acceptable.

C. Performance Analysis

	Coremark	FFT	AES_SHA	Mergesort	Compression
IPC (Dual Isse)	0.76 | 0.76	0.95 | 0.95	0.56 | 0.56	0.84 | 0.84	1.17 | 1.17
IPC (Single Issue)	0.71 | 0.73	0.81 | 0.81	0.53 | 0.54	0.77 | 0.82	0.88 | 1.15

Age-ordered scheduling works well in single-issue designs. With only one ALU, computing resources are limited, so there are always available instructions waiting in the queue, making the scheduling strategy impactful.

However, in dual-issue designs, its impact is limited. With two ALUs, most instructions are processed right away, leaving fewer entries in the queue and reducing the chances of multiple instructions needing scheduling.

v. Bypass Network

A. Design

We implemented an extensive bypass network, including both CDB bypass and ALU bypass for back-to-back execution, reducing instruction wakeup latency.

Bypass control is centralized in the issue queue and PRF, eliminating the need to handle it in each reservation station. Bypass detection happens in the issue queue, where entries monitor the CDB and ALU's rd_sel to update bypass enable and selection signals. The PRF uses large multiplexers to select values from registers, the CDB, or ALU results based on these signals.

The trade-off here is mainly area and timing. Our final design has 5 CDB and 2 ALU bypass networks, which are huge and complex. We improved timing by applying one-hot encoding, combining selection signals, and replacing standard multiplexers with one-hot versions.

B. Testing

The bypass network enables back-to-back execution, which is crucial for benchmarks with heavy data dependencies. It increases ALU utilization by eliminating the two-cycle wait for dependent instructions. Since the bypass network reduces wake-up latency without drawbacks, we expect an IPC increase across all benchmarks.

C. Performance Analysis

	Coremark	FFT	AES_SHA	Mergesort	Compression
ALU Utilization Rate	40.22% | 50.48%	40.27% | 62.44%	25.26% | 32.27%	46.65% | 64.48%	51.37% | 91.77%
IPC	0.58 | 0.76	0.74 | 0.95	0.55 | 0.56	0.72 | 0.84	0.70 | 1.17

The ALU utilization rate results from the combined efforts of age-ordered scheduling, the bypass network, with dual-issue. And the IPC improvements are only driven by the bypass network.

Both ALU utilization and IPC improved across all benchmarks, meeting our expectations. Compression showed the biggest gain, with about 1.7x IPC and 1.8x ALU utilization. This is likely because compression heavily uses arithmetic units and has many data-dependent instructions. The bypass network unlocked the ALU's potential.

4. Conclusion

Our out-of-order RISC-V processor successfully implements several key performance optimizations while maintaining elegance and compactness. We achieved our design objectives - high ILP through superscalar execution and minimal pipeline stalls through dynamic scheduling, while the implementation of explicit register renaming and distributed reservation stations provides efficient dependency handling.

Notable aspects of our design include the extensive bypass network reaching almost all functional units and the efficient organization of the ROB into two 32-entry banks. These features, combined with age-ordered issue scheduling and split load/store queues, demonstrate an effective balance between performance optimization and implementation complexity.

This project achieves high performance through modern architectural techniques while maintaining clean and efficient resource utilization through careful design choices and trade-offs.