Superscalar and Out-of-Order Processor Design Report

Advanced Computer Architecture

Table of Content

I. Introduction

1.1 Purpose of the Study

This project focuses on the design and implementation of an out-of-order (OoO) RISC-V processor prototype. The study provides hands-on experience with advanced microarchitectural concepts, including instruction-level parallelism, dynamic scheduling, hazard detection, memory consistency, and precise exception handling. By independently designing a functional, multi-component RTL model, the project demonstrates the ability to manage complex digital systems while maintaining correctness and performance.

1.2 Background and Context

Modern processors achieve high performance through instruction-level parallelism, executing multiple instructions simultaneously while maintaining program correctness. Out-of-order execution allows instructions to proceed as soon as their operands are ready, rather than strictly following program order, thereby maximizing resource utilization and throughput.

Implementing an OoO processor requires careful coordination of multiple microarchitectural components, including reservation stations, load/store queues, the reorder buffer (ROB), and hazard detection units. This project demonstrates how contemporary CPUs manage dependencies, speculative execution, and memory ordering to achieve high efficiency while preserving precise program behavior.

1.3 Scope and Design Goals

1. Follow RISC-V Integer Instruction Set Architecture for the instructions.
2. Instructions pre-stored in the instruction queue, with simulation ending when instructions run out.
3.Develop a basic dispatch unit capable of handling two instructions per cycle.
4. Maintain in-order instruction dispatch, stalling the second dispatch if the first cannot proceed.
5. Apply Tomasulo's algorithm to enable out-of-order execution while preserving the appearance of in-order completion.
6. Incorporate hazard detection and resolution mechanisms to manage data and structural dependencies.
7. Implementing a Reorder Buffer (ROB) that commits up to two instructions per cycle, ensuring in-order commit of instructions, despite out-of-order execution.
8. Implement a mechanism for precise exceptions, ensuring all instructions prior to an exception have committed and none after it have modified architectural state.

1.4 Objectives

1. Gain a solid foundation in digital design principles applicable to real hardware systems.
2. Comprehend the basic principles of modern computer architecture.
3. Build an end-to-end understanding of microprocessor operation.

4. Apply the design principles in the implementation of a functional microprocessor.
5. Test and debug the design systematically.
6. Demonstrate the ability to manage the design complexity of microprocessors.
7. Appreciate the trade-offs between performance optimization and architectural correctness in modern processors.

1.5 Report Overview

This report presents the design and implementation of the OoO RISC-V processor in a structured manner:

Chapter 2: Design Overview presents a high-level view of the architecture, instruction flow, and simplifications adopted in the project.

Chapter 3: Microarchitecture Details provides in-depth descriptions of internal components, including the instruction queue, register file and renaming, reservation stations, functional units, load/store queues, common data bus, reorder buffer, and hazard handling.

Chapter 4: Verification covers the testing methodology, including example programs, randomized tests, and comparison with a reference RISC-V simulator to ensure correctness.

Chapter 5: Conclusion summarizes outcomes, reflects on learning objectives, and suggests directions for future enhancements.

II. Design Overview

2.1 High-Level Architecture

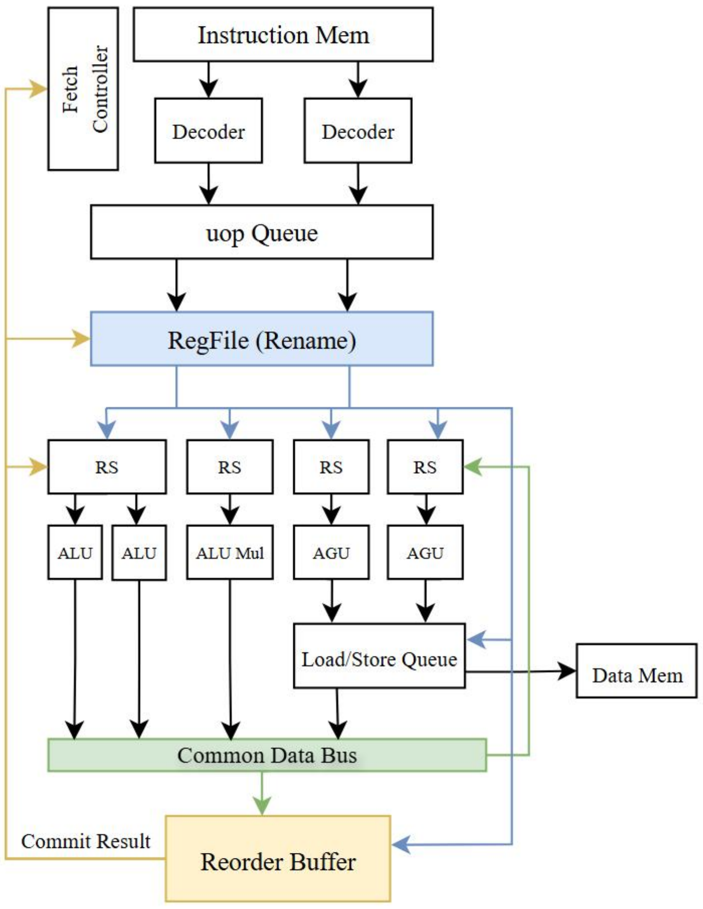


Figure 2.1: High-Level Block Diagram of the Superscalar Out-of-Order RISC-V Core

The processor is based on a RISC-V superscalar out-of-order (OoO) execution core, designed to support dual-issue and dual-commit per cycle. At a high level, the architecture integrates the following major subsystems:

Frontend: A fetch controller drives the instruction memory and feeds into dual decoders, enabling parallel decoding of instructions per cycle. Decoded instructions are queued in a micro-op (uop) buffer for downstream dispatch.

Register Renaming & Issue: A rename-based register file eliminates false dependencies and supports out-of-order scheduling. Instructions are dispatched into dedicated reservation stations (RS) associated with execution units.

Execution Units: The datapath includes two integer ALUs for general-purpose logic, one multiplication ALU, and two address-generation units connected to a load/store queue. These execution units are fed in parallel, exploiting instruction-level parallelism (ILP).

Memory System: The load/store queue provides out-of-order memory access capability while maintaining memory consistency, backed by a data memory module.

Commit & Reorder Buffer (ROB): A reorder buffer manages in-order retirement of instructions, ensuring precise exceptions and architectural state consistency. Results are broadcast on a dual-channel common data bus (CDB), which allows up to two instructions per cycle to complete and commit.

2.2 Instruction Flow

The instruction pipeline proceeds as follows:

Fetch & Decode: The fetch controller retrieves instructions from instruction memory. Two decoders operate in parallel, translating instructions into uops.

Dispatch & Rename: Uops enter the rename register file, where logical registers are mapped to physical registers, eliminating write-after-read (WAR) and write-after-write (WAW) hazards.

Issue: Instructions are placed into reservation stations until their operands are ready. Dependency resolution occurs dynamically through result forwarding on the dual CDB.

Execution: Ready instructions are dispatched to ALU logic units, the multiplier, or address-generation units. Memory instructions flow through the load/store queue, which handles out-of-order memory accesses while enforcing consistency.

Writeback: Execution results are broadcast to dependent instructions via the dual-channel CDB, accelerating wakeup and operand forwarding.

Commit: Instructions are retired in-order through the ROB, with dual commit per cycle supported. This ensures the architectural state remains precise despite out-of-order execution.

This pipeline design enables multiple instructions to be in-flight simultaneously, improving throughput while maintaining correctness.

2.3 Simplifications

This design implements the essential components of a modern OoO superscalar processor, but several simplifications were made to maintain feasibility and project scope:

No Branch Prediction: The architecture does not include a branch predictor. Branch instructions are not added in the verification process also.

Textbook-Guided Components: The design primarily follows the classical Tomasulo's Machine, though some implementation details deviate for practicality and/or improvements.

Focused Execution Units: Only a subset of execution units (two ALU logic, one multiplier, two address units) are implemented. Additional functional units (e.g., floating-point, vector) are excluded to reduce design complexity.

Simplified Memory System: While the load/store queue enforces memory consistency, advanced optimizations like memory disambiguation prediction or speculative loads are omitted.

Specific simplifications and deviations will be explained in detail in their corresponding sections.

These simplifications aim at a balance between technical depth and educational accessibility, producing a design that demonstrates the principles of superscalar and OoO execution while remaining feasible for implementation as an individual project.

III. Micro-architecture Details

3.1 Instruction Queue

1. Fetch Control

The fetch logic is responsible for generating the next program counters (pc_next[0], pc_next[1]) and determining how many instructions are issued into the decode/uop queue each cycle. The design supports dual-way fetch using two program counters:

pc[0]: the base PC of the fetch group.

pc[1] = pc[0] + 4: the second instruction in the fetch group.

(1) Program Counter Update

On resume from exception, pc_next[0] is set to resume_pc to continue correct execution.

On flush, pc_next[0] is updated to cmt_pc_next, which reflects the next sequential commit PC.

When two instructions are fetched, the PC advances by 8 bytes.

When one instruction is fetched, the PC advances by 4 bytes.

If no fetch occurs, the PC holds its value.

In all cases, pc_next[1] = pc_next[0] + 4, enabling dual-way fetch.

(2) Commit PC Tracking

The commit PC is tracked independently of the fetch PC.

If two instructions commit, the Commit PC advances by 8 bytes.

If one instruction commits, the Commit PC advances by 4 bytes.

On resume, cmt_pc_next = resume_pc.

Otherwise, the commit PC holds its value.

This ensures that recovery after flush always redirects fetch to the last known good commit point.

(3) Fetch Decision

Fetch is disabled if:

i) A pipeline stall occurs.

ii) End of program is reached (pc[0] >= program_pc).

iii) The uop queue is full.

If only one slot is available or pc[1] exceeds the program boundary, fetch issues one instruction.

Otherwise, two instructions are fetched.

2. Decode

Instruction	opcode	RS mask	wr_to_rf	ALUSrcB
lw	0000011	0010	1	1
sw	0100011	0001	0	0
R-type	0110011	1000	1	0

I-type	0010011	1000	1	1
Mul	0110011	0100	1	0
ecall	1110011	1000	0	0

Table 3.1 Decoder Truth Table

3. Micro-op Queue

The uop queue serves as a staging buffer between the decode stage and the rename stage. It is implemented as a dual-banked FIFO, supporting up to two writes (from dual decode) and two reads (into dual rename) per cycle. The control interface uses sequential read/write enable signals, meaning that multi-bit enables must follow in-order encoding (e.g., 10 is valid, but 01 is not). This constraint simplifies pointer management and reduces hardware complexity.

Since the uop queue functions primarily as a buffering structure, its internal details are not central to the project's objectives. The design details are omitted.

3.2 Register File and Renaming

1. Rename Flow

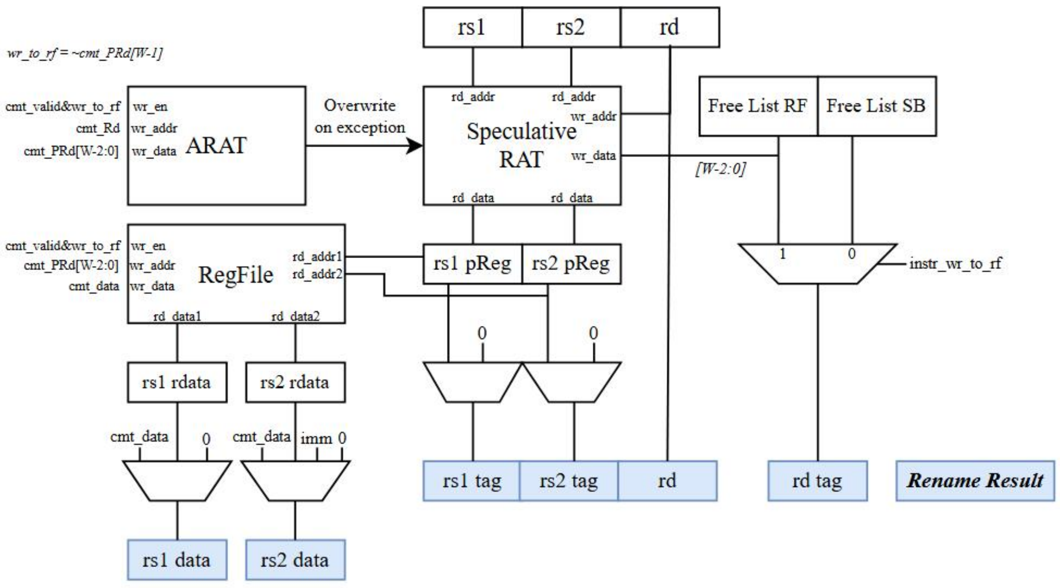


Figure 3.1: Register File and Rename Datapath (Single Rename Channel)

The renaming stage maps architectural registers to physical registers, eliminating false dependencies and enabling multiple instructions to execute out-of-order. Each instruction entering the rename stage is processed through the following flow:

(1) Destination Register Handling:

Instructions are divided into two types:

Register-writing instructions (ALU operations, loads) that update the architectural register file (RF).

Non-register-writing instructions (stores, branches) that do not produce a new architectural value.
To differentiate these two classes, the allocator assigns tags from separate freelists. A type bit is appended to the allocated tag:
MSB = 0: Register-writing instruction.
MSB = 1: Non-register-writing instruction.
The final destination tag (rd_tag) is therefore {type_bit, physical_tag}. This ensures that all in-flight instructions are uniquely identifiable.
(2) Source Operand Resolution:
For each source register (rs1, rs2), the rename logic selects between a data value and a tag:
i) If the source value is already committed in the RF, the operand data (rs1_rdata, rs2_rdata) is taken directly.
ii) If the value is being committed in the same cycle, it is bypassed from the commit stage (cmt_data).
iii) If the architectural register is x0 (hardwired to zero in RISC-V), the operand value is set to 0.
iv) For immediate-type instructions, rs2 is replaced by the immediate field (imm).

If none of these conditions apply, the rename stage outputs the physical tag for the operand, ensuring the instruction waits for its producer.

Commit Snoop and Operand Availability:
In this design, snooping on the commit result is sufficient to guarantee operand availability. If a renamed instruction misses its pending operand on the CDB because the broadcast occurs in the same cycle, it will still have another opportunity to capture it on later cycles from commit channels—since pending operands always originate from earlier instructions that will eventually commit. However, the commit stage represents the final chance to obtain the value. Without same-cycle commit bypass, an instruction could stall indefinitely, waiting on an operand that has already been retired.
A same-cycle commit bypass is therefore necessary to prevent livelock. Same-cycle CDB bypassing could allow instructions to become ready earlier in some corner cases, but this optimization is not essential and adds extra complexity. Later pipeline buffers (e.g., reservation stations) continue to snoop on the CDB. This guarantees forward progress and correctness without aggressive bypassing logic.

2. Register Alias Table
This design uses a shadow RAT organization consisting of two mapping tables with distinct roles: the Architectural RAT (ARAT) and the Speculative RAT (SRAT). The SRAT is the active mapping table used during normal rename/dispatch for speculative, out-of-order execution. The ARAT holds the committed architectural mappings and is only updated as instructions retire. This separation preserves the textbook semantics of a RAT while providing a low-latency and low-complexity recovery path.

(1) SRAT (Speculative RAT).
During rename, incoming instructions consult the SRAT to translate each architectural source register into either a physical tag (if the value is still speculative) or a “data_present” indicator (if the architectural register is already backed by a committed physical register).
The SRAT receives speculative destination allocations from the free list and records the new physical destination (the rd_tag described earlier). These speculative mappings are what downstream stages use to produce producer/consumer dependencies.
The SRAT supports the usual RAT operations: multiport reads for source lookup and dual-port writes on allocation - It needs to handle the dual-dispatch width of the design (two allocations per cycle).

(2) ARAT (Architectural RAT).
The ARAT is the authoritative architectural mapping visible to software state. It is updated only on commit: when an instruction retires and its result becomes architecturally visible, the ARAT entry for the destination register is overwritten with the retiring instruction’s physical tag.
Because the ARAT reflects only committed state, it provides a safe checkpoint for recovery. On an exception or flush the pipeline can restore architectural mappings instantly by restoring SRAT from ARAT (methods described below), without having to walk and clear ROB entries or broadcast flushes to many structures.

(3) Restore
Speculation is confined to the SRAT. Any speculative rename changes are solely in SRAT and in freed/allocated entries tracked by the free list bookkeeping. If an exception or mis-speculation occurs, the system can discard the SRAT (or overwrite it from ARAT).
This design guarantees precise state because the ARAT is only modified at commit time; until commit, the committed architectural mapping remains unchanged. Thus, the ARAT+SRAT separation implements a simple checkpointing model without complicated multi-version RAT encodings.

(4) Implementation approaches and tradeoffs.
A straightforward implementation is to maintain two full-size RAT arrays (ARAT and SRAT) and copy ARAT to SRAT on recovery. A more advanced alternative is incremental checkpointing (record per-entry previous mapping) or pointer-swapping when the architecture supports banking; these approaches trade implementation complexity, area, and restore latency.

(5) Benefits relative to textbook RAT designs.
The shadow RAT (ARAT+SRAT) provides an efficient and simple recovery mechanism: in the common case recovery is a fast copy rather than a multi-cycle ROB walk and broadcast.
This ARAT/SRAT organization therefore keeps the rename logic functionally equivalent to textbook designs during normal operation, while improving flush/recovery performance and implementation simplicity.

3. Free list (Allocator)

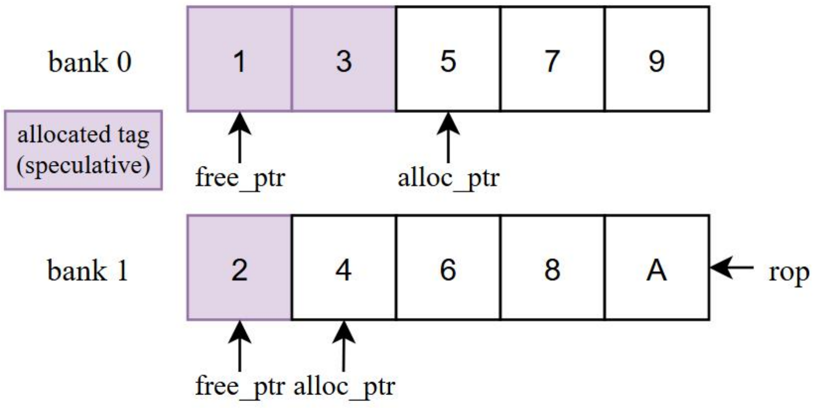


Figure 3.2: Dual-Banked Freelist with Pointer-Based Allocation and Retirement

Each bank corresponds to one dispatch way, matching the dual-issue pipeline of the processor. Conceptually the freelist in this design behaves as a FIFO of tag identifiers, but implemented without storing per-entry data. Physical tags are arranged in a fixed numerical order so that an encoder can translate a pointer value directly into the corresponding tag. From this perspective, whether one interprets the structure as a freelist or an allocate list is purely a matter of viewpoint:
One may regard the list as initially empty, with tags entering as instructions are dispatched.
Alternatively, it may be considered initially full, with tags consumed by allocation and returned on commit.
Both views are functionally identical in hardware, since the pointers alone define the state of availability.

(1) Allocation:
On dispatch, each bank consumes a free physical register tag by advancing its alloc_ptr. Allocated entries are marked speculative and remain “in-flight” until the corresponding instruction retires. Because allocation proceeds strictly in program order, the tag sequence assigned to instructions is also in-order, simplifying dependency tracking.

(2) Freeing (Commit):
On commit, the free_ptr advances to release the physical register previously associated with the retiring instruction. This mirrors a FIFO read, where tags added earlier at dispatch are removed later at retirement. Both dispatch and commit proceed in program order, so explicit tag matching between pointers is unnecessary—the FIFO guarantees correctness.

(3) Dual Dispatch and Commit:

Since both dispatch and commit can handle up to one instruction per cycle per bank, the system requires a reorder pointer (ROP) for each side.
The reorder pointer toggles priority between banks when only a single allocation or commit is valid, which is necessary to preserve correct sequencing.

(4) Flush Handling:
On an exception flush, the system instantly clears speculative allocations by moving the free_ptr to the current alloc_ptr, effectively emptying the FIFO. This single pointer update discards all speculative tags in one step, eliminating the need for per-entry invalidation.

(5) Extension for Branch Handling:
Although this project does not implement branch prediction, the dual-banked freelist naturally supports it. By recording an anchor at the time of prediction, the allocator can roll back to that anchor on misprediction, restoring the freelist to the state at the prediction point.

(6) Integration with Shadow RAT:
The freelist works in tandem with the shadow RAT (ARAT/SRAT) organization. The SRAT consumes speculative tags from the freelist during rename, while the ARAT updates only at commit.
On recovery, restoring SRAT from ARAT and resetting freelist pointers together ensures a consistent architectural state in one cycle.

This dual-banked FIFO freelist design thus enables efficient dual-issue allocation, precise in-order commit, and fast one-cycle recovery, aligning with the goals of a high-throughput, out-of-order RISC-V pipeline.

3.3 Reservation Station

1. RS Entry

valid	alu_op	rdTag	Qj	Qk	Vj	Vk	imm

valid: Set if this entry is in use

aluop: ALU control signal

Qj, Qk: The physical register tags that the operands are waiting, 0 if data is ready

Vj, Vk: The actual operand values

rdTag: The tag allocated to this instruction

2. Accept Logic

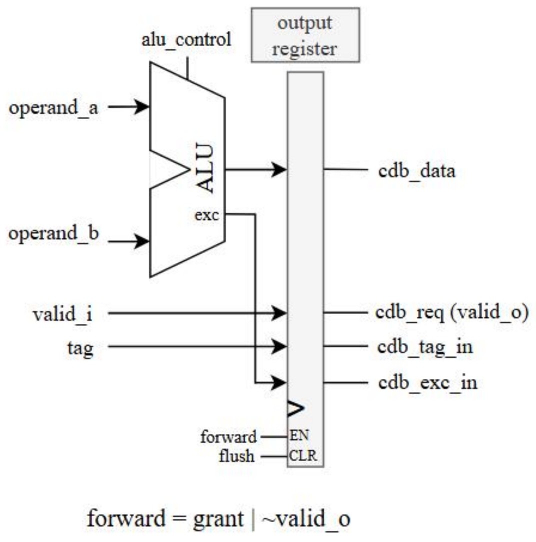
The reservation stations (RS) act as distributed instruction buffers, holding operations until all operands are ready and a functional unit is available. This project follows the classical RS model with a few extensions for dual-issue support.

At decode/rename, each instruction is assigned a rs_mask that specifies which RS is responsible for it. Only one RS can accept a given instruction. Each RS can accept up to two instructions per cycle by selecting two free entries.
An RS asserts its accept signal when the following conditions are satisfied:
1) The instruction is valid and its rs_mask matches the RS.
2) The RS has free entries available.
3) The pipeline is not stalled.
4) For load/store RS only: the LSQ also has free space available

3. Issue Logic
Each RS entry continuously snoops the CDB (both channels) as well as the commit result bus.
When a snooped tag matches an operand tag, the operand value is captured, and the tag is cleared (set to zero).
An entry is considered ready to issue when both operand tags equal zero, meaning the operands are available either from the register file, commit bypass, or CDB.
Each RS can select up to two ready entries per cycle for dispatch to the corresponding functional units.

3.4 Functional Unit

1. ALU



forward = grant | ~valid_o

Figure 3.3: Datapath for ALU

The ALU's result passes through an output register, which serves as a staging point before broadcasting onto the CDB.

The forward condition is central to allowing the functional unit (FU) to operate continuously without stalling or losing results. It provides a mechanism to control when the ALU can safely load a new result into its output register:

Granted Bus Access:

If the FU wins arbitration (grant), the current result will be driven onto the CDB in the next cycle. This frees the output register to accept a new result from the ALU, allowing uninterrupted execution.

Empty Output Buffer:

If the output register is empty (~valid_o), a new result can be directly loaded without overwriting any pending value.

2. Multiplier

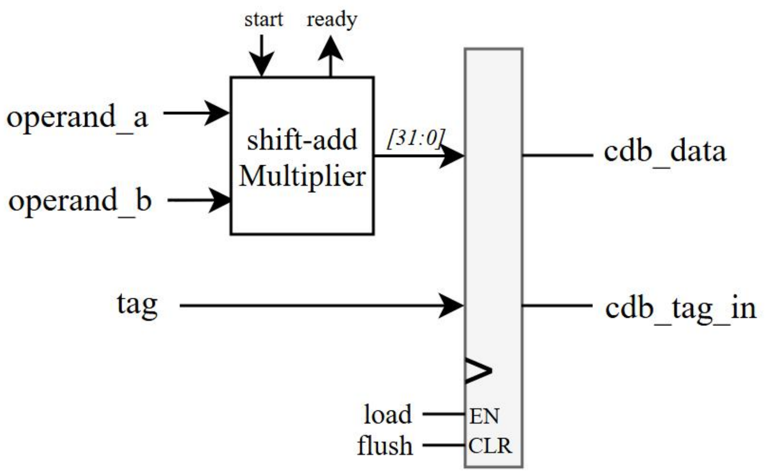
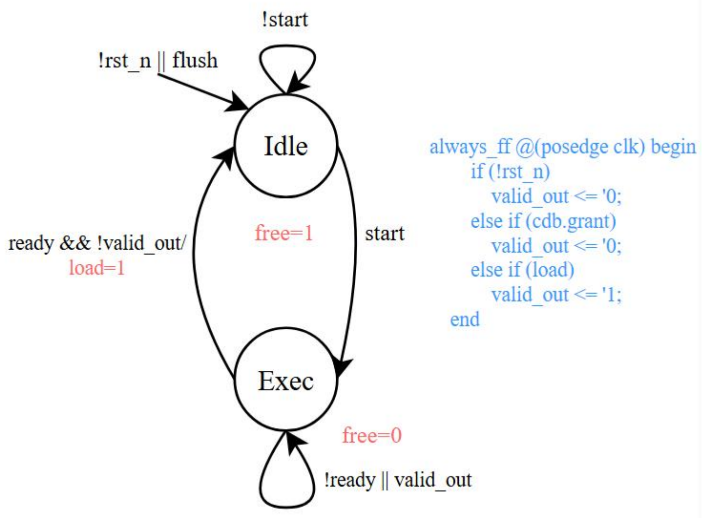


Figure 3.4.1: Datapath for Multiply FU



```
always_ff @(posedge clk) begin
    if (!rst_n)
        valid_out <= '0;
    else if (cdb.grant)
        valid_out <= '0;
    else if (load)
        valid_out <= '1;
end
```

Figure 3.4.2: FSM for Multiply FU

The design uses a simple shift-add multiplier. Since it is irrelevant to the goal of this project, the design detail is omitted here.

3. Address Unit

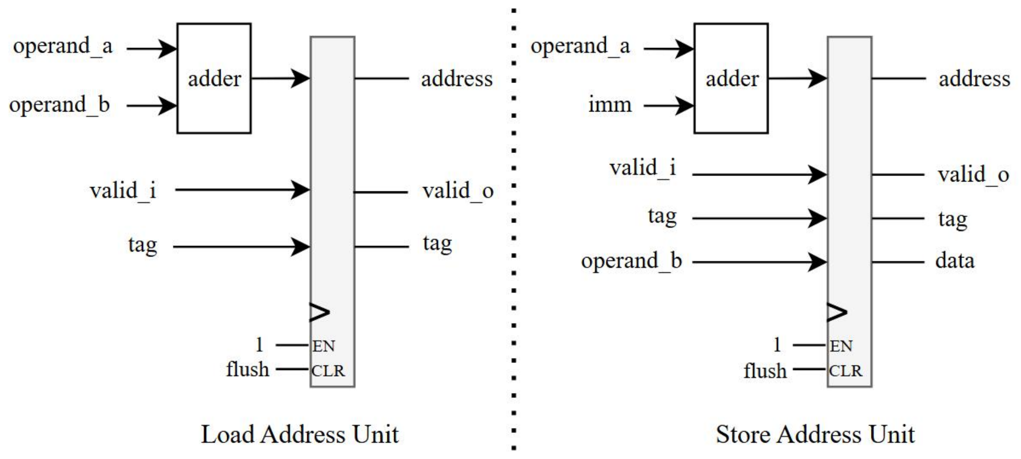


Figure 3.5: Datapath for Address Generation Units(AGU)

The Address Generation Units (AGU), consisting of the Load Address Unit and Store Address Unit, can operate without stalling because the output registers are guaranteed to forward results immediately. This is allowed by two structural properties of the design:
(1) Instruction Duplication in Reservation Station and LSQ:
Any instruction dispatched to a load/store reservation station is simultaneously inserted into the Load/Store Queue (LSQ). This ensures that the LSQ already tracks the instruction, so once the AGU produces an address, the LSQ is ready to accept it.
(2) Unconditional LSQ Acceptance:
The LSQ is designed to always accept new address resolutions every cycle. There is no scenario where the LSQ would backpressure the AGU. This eliminates the possibility of the AGU's output register being blocked by downstream resources.
As a result, the AGU output register never needs to hold results longer than a cycle, and forwarding is always enabled. This design simplifies control logic.

3.5 - Load/Store Queue (LSQ)

1. Store Queue

(1) Store Queue Entry:

valid	ready	cmt	tag	addr	data

valid: Indicates that the entry contains a valid store instruction.
ready: Set when both the address and the data of the instruction are resolved.
cmt: Set once the instruction has been committed by the Reorder Buffer (ROB), indicating it is ready for execution.
tag: A unique identifier assigned to the instruction
addr: The memory address targeted by the store instruction.
data: The value to be written to memory.

(2) Execution Logic
This project adopts a lazy execution approach for store instructions. Stores are executed only after they are committed. This approach avoids the complexity of speculative memory writes, which would require notifying memory and handling corrective actions in the event of a rollback. Executing stores on commit simplifies memory management while ensuring correctness.
Store Queue Entries are removed only when the memory unit informs so. This can relieve the bypass logic complexity in the cache controller by avoiding a new write be immediately read.

2. Load Queue
(1) Load Queue Entry

valid	ready	rd_tag	addr	sq_idx	no_older_st

valid: Indicates whether the entry currently holds a valid load instruction.
ready: Set when the memory address for the instruction has been resolved.
rd_tag: The unique tag allocated to this instruction.
addr: Stores the memory address associated with the instruction.
sq_idx: Captures the head pointer of the store queue at the time the instruction is enqueued. This checkpoint allows the processor to identify which store instructions are older than the current load instruction.
no_older_st: Set when there are no older store instructions preceding this load, indicating it can safely access memory without dependencies on prior stores.

(2) Execution Logic

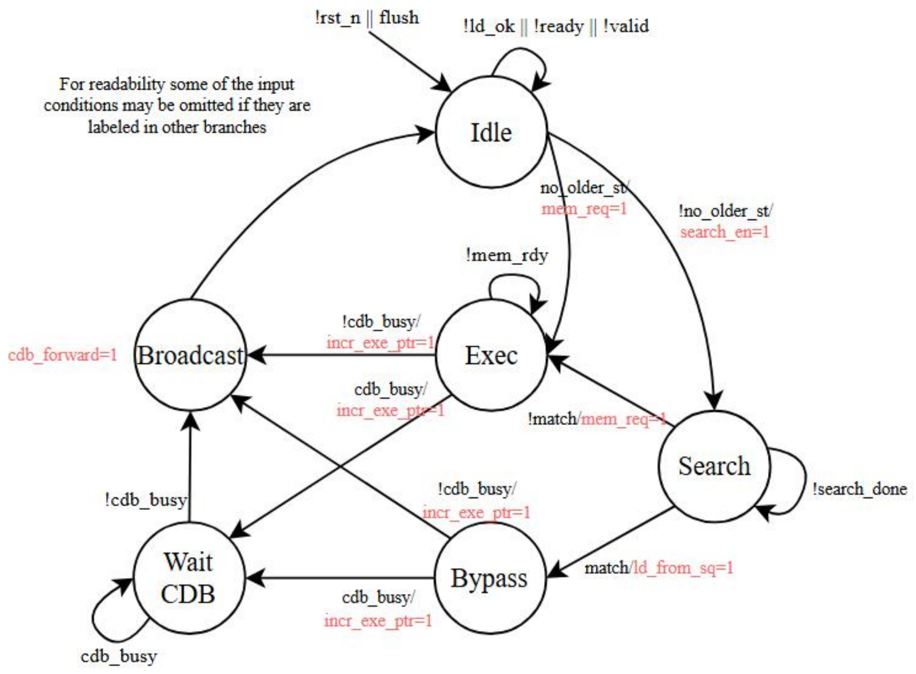


Figure 3.6: Execution FSM for Load Queue

The no_older_st flag is used to determine whether any older store instructions exist in the Store Queue (SQ). Its value governs whether a load can safely access memory directly or must check for pending stores:
If there are older stores targeting the same memory address, the load must forward data from the store instead of accessing memory. no_older_st is set when either:

i) The store queue is empty at the time the load is enqueued, or

ii) The tail pointer of the store queue has passed the sq_idx checkpoint of the load.

This mechanism ensures correct memory ordering while minimizing unnecessary store searches.

3. Store Queue Search

(1) Search Logic

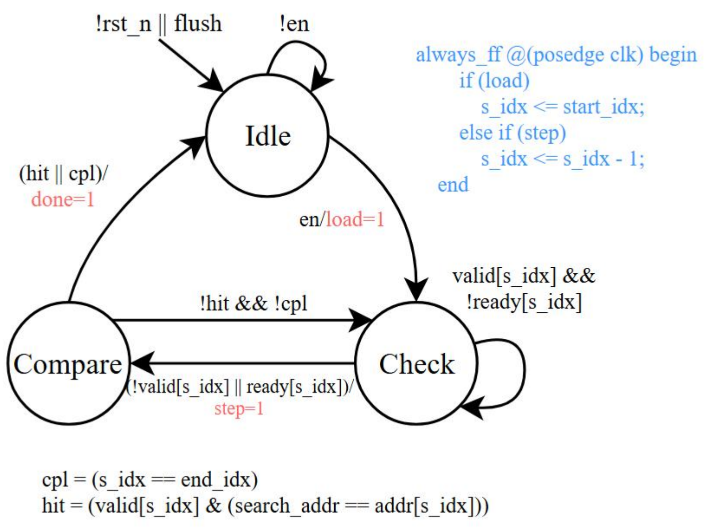


Figure 3.7: FSM for Store Queue Search

In this project, a simplified load execution design is employed:
A load instruction stalls if it encounters an unresolved older store in the SQ, preventing unsafe execution and eliminating the need for a reverse search.
In more advanced, unblocking designs, the load queue can continue executing younger loads speculatively, even when an older store is unresolved. Once the store resolves, all younger loads are checked: loads targeting the same address must replay their memory access to ensure correctness.
This simplified approach prioritizes correctness and design simplicity, while unblocking designs provide higher performance at the cost of additional complexity.

3.6 Common Data Bus (CDB)

The Common Data Bus (CDB) is the critical broadcast fabric that allows completed instruction results to be shared with all dependent instructions in flight. In this design, the CDB is implemented with dual channels, enabling two independent results to be transmitted concurrently per cycle.

1. Datapath

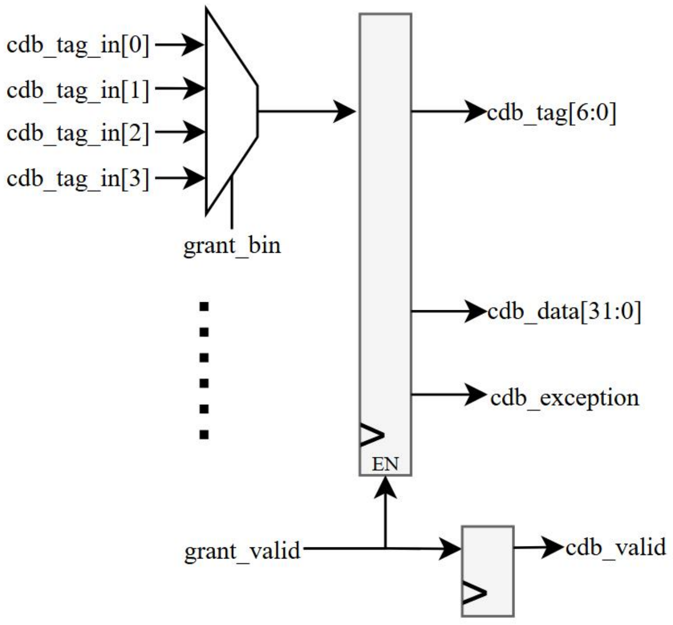


Figure 3.8: Common Data Bus (CDB) Datapath for a Single Channel

As shown in Figure 3.8, each CDB channel datapath consists of a multiplexer-driven selection logic that receives candidate result tags (cdb_tag_in), corresponding data values, and exception signals from execution units. The arbiter determines which source is granted access, producing a binary grant signal (grant_bin) that selects the winning input. Once granted, the datapath outputs a tag (cdb_tag[6:0]), the computed value (cdb_data[31:0]), and any associated exception status. A valid bit (cdb_valid) is generated from the grant enable signal (grant_valid) to indicate the presence of a broadcast.

2. Dual-Channel Arbitration

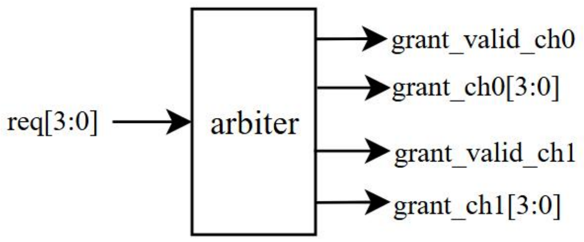


Figure 3.9: Dual-Channel Grant Arbiter for the CDB

Coordination is required when multiple execution units request broadcast in the same cycle. As shown in Figure 3.9, a dual-channel arbiter accepts request signals (req[3:0]) from execution units and generates separate grant signals for each channel (grant_ch0 and grant_ch1). Each channel also asserts its own validity signal (grant_valid_ch0 and grant_valid_ch1) to indicate whether a broadcast has been successfully allocated.

By supporting two concurrent broadcasts per cycle, the design sustains the dual-issue and dual-commit pipeline targets, while keeping implementation overhead manageable.

3. Arbitration Logic

The arbiter ensures that up to two requesters can be granted access to the CDB in a single cycle using a rotating priority scheme. Its operation can be broken down into three key steps:

(1)Rotation: Requests are rotated so that the current pointer (ptr) maps to the lowest index. This enables round-robin fairness by changing which requester is considered "first" over time.

```
// Rotate requests right by ptr: rot_req = ROR(req, ptr)
rot_req = ({req, req} >> ptr);
```

(2)Winner Selection: The first grant is chosen by isolating the lowest set bit (g0_rot). The second grant is chosen from the remaining requests (g1_rot), again by picking the lowest set bit.

```
g0_rot = rot_req & (~rot_req + 1);   // First winner
rot_req_2 = rot_req & ~g0_rot;      // Remove first winner
g1_rot = rot_req_2 & (~rot_req_2 + 1); // Second winner
```

(3)Un-Rotation and Pointer Update: The selected grants are rotated back to their original positions. The pointer is then advanced by the number of grants issued (0, 1, or 2), ensuring fairness across cycles.

```
// Restore grant positions
grant0 = ({g0_rot, g0_rot} << ptr)[2*N-1:N];
grant1 = ({g1_rot, g1_rot} << ptr)[2*N-1:N];
// Advance pointer by number of winners
grant_cnt = (|g0_rot) + (|g1_rot);
ptr_next = (ptr + grant_cnt) % N;
```

3.7 Reorder Buffer (ROB)

1. ROB Entry

valid	completed	rd	rd tag	data	exception

valid: Indicates whether the current ROB entry contains a valid instruction. If set, the entry is active and tracks an instruction that has been issued but not yet retired.

completed: Specifies whether the instruction in this ROB entry has finished execution. Once the instruction completes execution, this bit is set, indicating that the result is ready to be committed

rd: The architectural destination register of the instruction.

rd_tag: The unique tag allocated to this instruction.
data: Holds the result produced by the instruction once it has executed.
exception: Flags any exceptions or errors encountered during instruction execution. If set, this entry will trigger exception upon retirement.

2. ROB Diagram

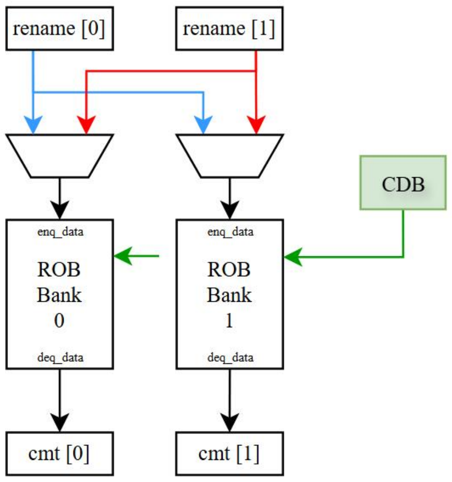


Figure 3.10: Reorder Buffer (ROB) Datapath

The Reorder Buffer (ROB) enforces in-order commit while allowing out-of-order execution. To support dual-dispatch and dual-commit, the ROB is implemented as a dual-banked structure, as shown in Figure 4.1.
Bank 0 and Bank 1 hold independent subsets of in-flight instructions. Each bank has its own enqueue and dequeue ports, allowing instructions from the two rename lines (rename[0], rename[1]) to be allocated in parallel. Commit operations are similarly supported in parallel, enabling up to two instructions to retire per cycle.
Bank 0 must hold all even-numbered instructions (i0, i2, i4...) and Bank 1 must hold all odd-numbered instructions (i1, i3, i5) This is accomplished by using a Input Selection Logic discussed in subsection 3.

3. Input Selection Logic

The enqueue logic maintains program order across two banks by using a rotating reorder pointer(rop) to control which bank has priority each cycle. If rop is 0, Bank 0 has priority, bank 0 takes the earlier instruction and bank 1 takes the later. If rop == 1, the roles are reversed. Reorder pointer will toggle if there is only 1 valid dispatch.   The logic can be illustrated in Table

rename[0] valid	rename[1] valid	reorder pointer	bank 0 enq	bank 1 enq	bank 0 sel	bank 1 sel
0	0	x	0	0	x	x

1	0	0	1	0	0	x
1	0	1	0	1	x	1
1	1	0	1	1	0	1
1	1	1	1	1	1	0

Table 3.2: Truth Table for ROB Bank Input Selection Logic

Since the dispatch unit forces stalling the second dispatch if the first stalls, case that rename[1] is valid but rename[0] is not valid is impossible.

4. Commit Logic

Similarly, The commit logic maintains program order by using a rotating pointer (retire_ptr) as well.
If retire_ptr == 0, Bank 0 has priority:
Bank 0 can commit if its instruction is completed and has no exception.
Bank 1 can only commit in the same cycle if its instruction is also ready and Bank 0 is committing.
If retire_ptr == 1, the roles are reversed:
Bank 1 gets priority, and Bank 0 can only commit in the same cycle if Bank 1 also commits.

When exactly one commit occurred, retire pointer is rotated to give the other bank priority next cycle. Otherwise (both banks commit or neither commit), the pointer does not change.

5. Exception Logic

An exception triggers a flush pulse that kills all in-flight instruction, clear ROB and restore register alias table (RAT) to fully restore architectural state. As shown in Section 3.2, the design leverages a checkpointed register file and freelist that can restore the precise architectural state in a single cycle. As a result, the ROB no longer needs to broadcast and individually invalidate flushed instructions after an exception or mis-speculation. This represents a significant improvement over the classical Tomasulo machine, where the ROB was responsible for squashing younger instructions by actively walking and clearing entries. By offloading recovery to the register file and freelist, the flush mechanism is both simpler and faster while still maintaining precise exception semantics.

In a dual-retire structure, hazard arises when both exception and commit appear in the same cycle. There are 2 possible scenarios:
(1) Commit - Exception. (the earlier instruction is a commit)
Consider: *i0* - commit, *i1* - exception, *i2*, *i3* (retire_ptr = 0)
i0 can commit, *i1* cannot commit because it is an exception. commit valid signal will be ‘10’. Next cycle, retire_ptr becomes 1 so *i1* retire first, triggering the flush pulse. *i2* cannot retire because *i1* does not commit. Next cycle all instructions are flushed.

(2) Commit - Exception. (the earlier instruction is an exception)
This case is covered in out previous example. The later instruction cannot commit because the previous instruction does not commit. On the next cycle it will be flushed out.

The banked ROB design simplifies implementation compared to a unified multi-ported ROB, which would require higher complexity and area cost. By splitting instructions across two banks, the ROB can support dual-enqueue and dual-dequeue operations with reduced hardware overhead. This structure effectively balances performance scalability with implementation feasibility, making it well-suited for the superscalar dual-issue pipeline.

3.8 Hazard Control

The hazard control logic manages global pipeline stalls and flushes to ensure correctness in the presence of resource exhaustion or exceptions. To simplify implementation, this design applies a dual-lane stall policy: if one dispatch lane stalls, the other lane also stalls. This avoids the need for additional multiplexing and selection logic. The performance penalty is minor, since at worst a single instruction is unnecessarily delayed.

1. Stall Conditions

The pipeline asserts a stall signal if any of the following conditions are met:
1) ROB Full: The reorder buffer cannot accept new entries.
2) RS/LSQ Full: An instruction targeted to a reservation station cannot be accepted because the RS is full, or for load/store instructions, the LSQ is full.
3) No free physical tags (freelist cause): allocator cannot issue a destination tag. (An optimized design would contain this locally but this project intentionally lets it propagate downstream for simplicity.)
4) Exception Flush: A pending flush signal forces the pipeline to halt until the resume signal clears it.

2. Scope of Effects

Stalls are applied after uop queue dequeue, preventing new instructions from entering the reservation stations until resources are available.
Only the exception cause propagates back to fetch (i.e., normal resource stalls do not throttle fetch because uop queue will backpressure fetch controller).

9.3 Exception Handling

When an exception is detected, a flush pulse is triggered.
The flush invalidates all in-flight instructions and operations across the uop queue, rename logic, reservation stations, functional units, and LSQ.
The pipeline state is then restored from the shadow RAT and freelist mechanism, enabling a precise architectural rollback.

IV. Verification

4.1 Testbench

1. Memory Unit Model

The memory unit in the testbench models load and store behavior at the boundary between the processor and data memory. It responds to processor requests using simple tasks to emulate variable memory latency and completion signaling.

(1) Load Task

When a load request is issued, the testbench marks the memory unit as occupied and block additional load requests.

A random delay between RD_DELAY_A and RD_DELAY_B cycles is introduced to emulate realistic memory response latency.

After the delay, the memory asserts ready for one cycle and provides the requested data word from the data memory array.

The signal is then deasserted and ld_busy is cleared, allowing the processor to issue the next memory request.

Limitation: The processor does not support multiple outstanding reads (no MSHR). As a result, loads must complete before a new load request can be accepted.

(2) Store Task

Store requests are acknowledged with a fixed latency.

After the fixed delay, the memory asserts store_cpl for one cycle to indicate the store has completed. This handshake ensures the processor sees a deterministic completion event for each store, even though actual memory array writes are immediate in the testbench.

The store completion delay serves a specific verification purpose. In this design, the store queue retains entries until an explicit completion signal is received. By introducing a fixed delay before asserting store_cpl, the testbench ensures that store instructions remain valid in the queue for several cycles. This prevents a newly issued store from being immediately removed and then speculatively read back by a following load.

In a full cache-coherent system, the cache controller would typically forward ongoing writes directly to dependent loads, requiring coherence protocol configuration to guarantee correctness. Since coherence handling is outside the scope of this project, the delayed store completion model provides a simpler mechanism: it reduces the chances of immediate load–store forwarding pressure while still exercising the pipeline’s ability to handle pending stores.

This approach models the realistic behavior of memory systems where stores are not instantly retired architecturally, and aligns the testbench with the simplified project assumptions.

2. Main Simulation Process

The main simulation loop coordinates execution across multiple test runs and manages reset, program loading, execution, and result collection. The flow is as follows:

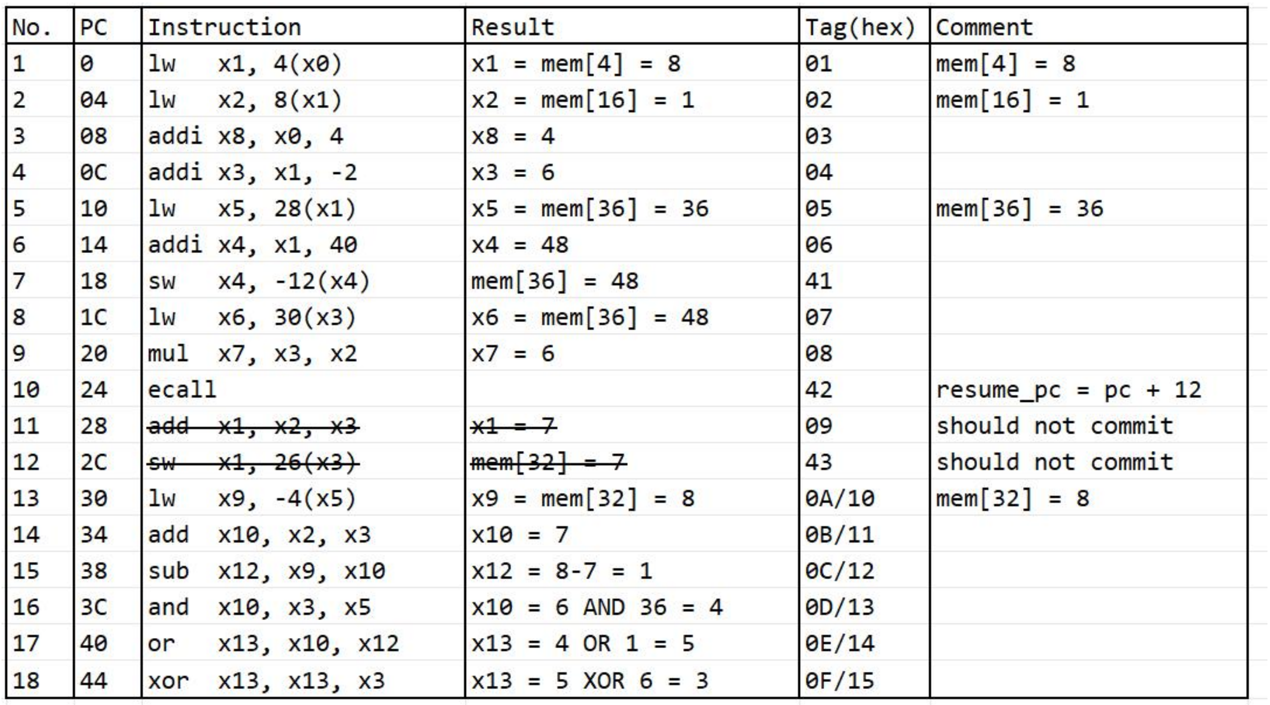
(1) Reset: The processor is reset at the start of each run.
(2) Program Load: The testbench loads instructions into instruction memory for the current run.
(3) Execution and Completion: Simulation advances until the processor signals proc_idle, indicating that all instructions for the program have been retired.
(4) Result Dump: The dump_result() task writes simulation outputs to a file.

Results include snapshots of data memory and the architectural register file, which is reconstructed from the physical register file and the ARAT (Architectural RAT).

3. Snapshot Mechanism
Snapshots are maintained by a separate monitoring process that records state at the moment instructions commit. This aligns with architectural correctness: only retired instructions should be reflected in the architectural state.

Data memory: Captured after store commits, ensuring no speculative writes are recorded.
Architectural register file: Reconstructed at commit using ARAT mappings and physical register contents, preserving the correct architectural view.
dump_result simply outputs these committed-state snapshots, guaranteeing that the recorded results represent a precise architectural state, independent of speculation or in-flight operations.

4.2 Example Test Program



No.	PC	Instruction	Result	Tag(hex)	Comment
1	0	lw x1, 4(x0)	x1 = mem[4] = 8	01	mem[4] = 8
2	04	lw x2, 8(x1)	x2 = mem[16] = 1	02	mem[16] = 1
3	08	addi x8, x0, 4	x8 = 4	03	
4	0C	addi x3, x1, -2	x3 = 6	04	
5	10	lw x5, 28(x1)	x5 = mem[36] = 36	05	mem[36] = 36
6	14	addi x4, x1, 40	x4 = 48	06	
7	18	sw x4, -12(x4)	mem[36] = 48	41	
8	1C	lw x6, 30(x3)	x6 = mem[36] = 48	07	
9	20	mul x7, x3, x2	x7 = 6	08	
10	24	ecall		42	resume_pc = pc + 12
11	28	~~add x1, x2, x3~~	~~x1 = 7~~	09	should not commit
12	2C	~~sw x1, 26(x3)~~	~~mem[32] = 7~~	43	should not commit
13	30	lw x9, -4(x5)	x9 = mem[32] = 8	0A/10	mem[32] = 8
14	34	add x10, x2, x3	x10 = 7	0B/11	
15	38	sub x12, x9, x10	x12 = 8-7 = 1	0C/12	
16	3C	and x10, x3, x5	x10 = 6 AND 36 = 4	0D/13	
17	40	or x13, x10, x12	x13 = 4 OR 1 = 5	0E/14	
18	44	xor x13, x13, x3	x13 = 5 XOR 6 = 3	0F/15	

Table 4.1: Example Test Program

1. Test program purpose

The test program is structured to validate the processor with emphasis on: memory subsystem correctness, precise exception handling, and register renaming hazard resolution. It is divided into three parts:

P1. Load/Store Validation (Before i9)

The early sequence stresses the load and store path.

Dependent loads and stores confirm that the LSQ preserves correct ordering and forwards/store-completes properly.

Commit must wait for stores to become architecturally visible, ensuring memory state is precise.

P2. Commit/Exception Hazard (i9 + i10)

The mul instruction introduces a long latency, keeping the ROB pending until the result is ready.

When mul completes, it becomes eligible to commit in the same cycle as the following ecall.

This creates a commit/exception hazard:

The multiply must commit first to ensure its result is architecturally visible.

Immediately after, the exception triggered by ecall must flush the pipeline.

P3. Post-Resume Register Dependency Tests (After i12)

After the exception, execution resumes from i13.

The subsequent code sequence tests the register renaming system under different hazard conditions:

RAW (Read After Write): Consumers must wait for producers to finish.

WAR (Write After Read): New writers must not clobber values still being read.

WAW (Write After Write): Later writes must overwrite earlier ones in correct order.

2. Instruction Flow Overview

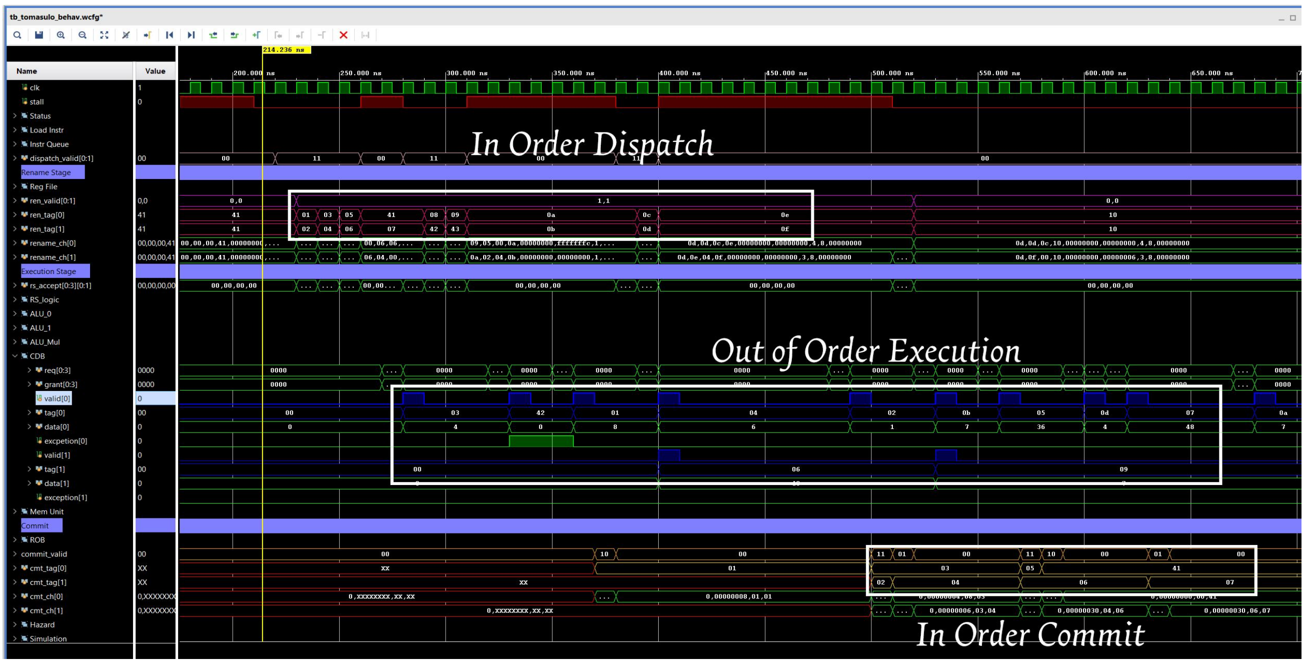


Figure 4.1.1 Simulation Waveform Showing Instruction Flow (Part 1)

The simulation waveform (Figure 4.1.1) shows that the processor preserves program order in both dispatch and commit stages despite out-of-order execution. Each instruction enters the pipeline, is renamed, and ultimately retires in the exact sequence defined in Table 4.1.

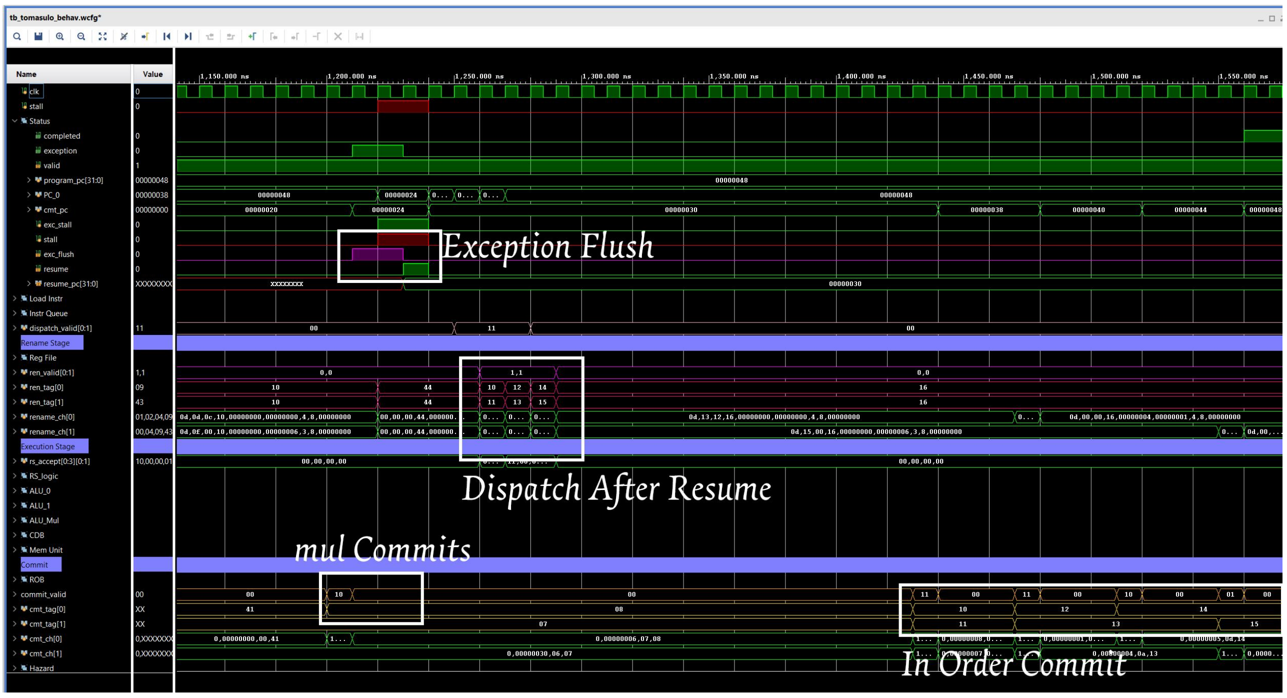
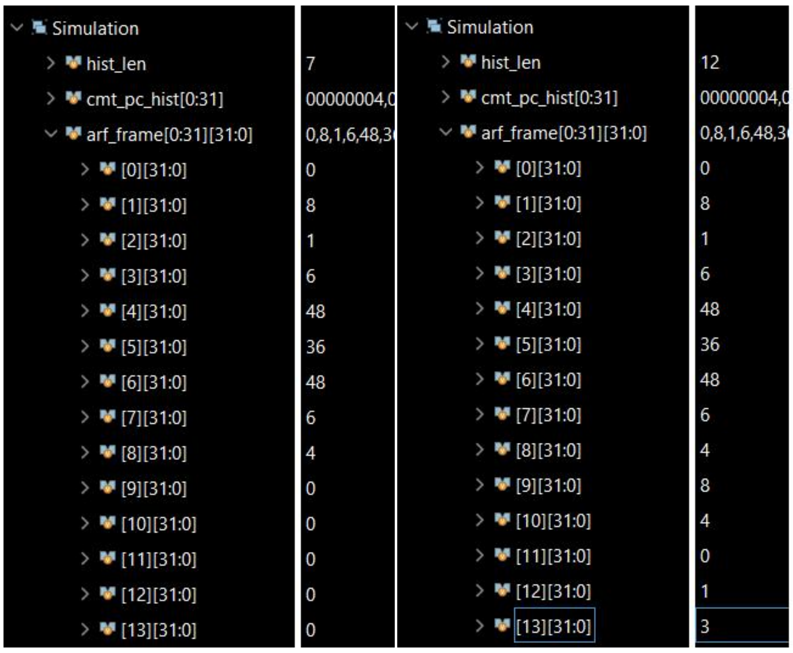


Figure 4.1.2 Simulation Waveform Showing Instruction Flow (Part 2)

The simulation waveform (Figure 4.1.2) demonstrates precise exception handling: all instructions prior to the exception retire successfully, while none of the younger instructions commit or alter the architectural state. Following the flush, the processor resumes execution at the specified resume_pc, correctly fetching, dispatching, and committing subsequent instructions in program order. Furthermore, Figure 4.2 shows that the architectural register file (ARF), reconstructed from the RAT and physical register file (PRF), exactly matches the expected values, confirming both correctness of state recovery and consistency of the renaming mechanism.



```
Simulation
  hist_len                   7
  cmt_pc_hist[0:31]          00000004,0
  arf_frame[0:31][31:0]      0,8,1,6,48,3
    [0][31:0]                0
    [1][31:0]                8
    [2][31:0]                1
    [3][31:0]                6
    [4][31:0]                48
    [5][31:0]                36
    [6][31:0]                48
    [7][31:0]                6
    [8][31:0]                4
    [9][31:0]                0
    [10][31:0]               0
    [11][31:0]               0
    [12][31:0]               0
    [13][31:0]               0
```

```
Simulation
  hist_len                   12
  cmt_pc_hist[0:31]          00000004,0
  arf_frame[0:31][31:0]      0,8,1,6,48,3
    [0][31:0]                0
    [1][31:0]                8
    [2][31:0]                1
    [3][31:0]                6
    [4][31:0]                48
    [5][31:0]                36
    [6][31:0]                48
    [7][31:0]                6
    [8][31:0]                4
    [9][31:0]                8
    [10][31:0]               4
    [11][31:0]               0
    [12][31:0]               1
    [13][31:0]               3
```

Figure 4.2 Architectural Register Values After Exception (L) / On Completion (R)

V. Conclusion

This project successfully implements a dual-issue, out-of-order RISC-V processor featuring reservation stations, a reorder buffer, a shadow RAT design (ARAT/SRAT), dual-channel commit, and a dual-channel common data bus. Through systematic testbenching and simulation, the design demonstrates correct handling of memory operations, precise exception recovery, hazard resolution via register renaming, and high-throughput instruction dispatch/retirement.

Several simplifications—such as the absence of branch prediction, unified stall control, and a minimal memory subsystem—kept the focus on the core aspects of superscalar out-of-order execution. Despite these simplifications, the project shows critical behaviors of modern processors: in-order commit, precise exceptions, flush/recovery in a single cycle, and correct handling of RAW/WAR/WAW hazards.

The results confirm that the architectural mechanisms—ROB, RS, LSQ, shadow RAT, and freelist—interact correctly to maintain program order at commit while exploiting instruction-level parallelism. The test programs and waveforms demonstrate that the design not only executes correctly but also adheres to expected architectural semantics.

Overall, the project achieves its primary goal: providing a clear and functional educational prototype of a superscalar out-of-order machine. While simplified compared to industrial implementations, it highlights the key microarchitectural structures and their interactions. This project builds a solid foundation for future extensions such as deeper pipe-lining, branch prediction, float-point instruction set, more advanced scheduling, and cache coherence.