Contents

3.7.9 Reorder Buffer (ROB)

3.7.9.1 Hardware-based Speculation

Speculation in hardware is a foundational technique in high-performance processors, used to execute instructions **before it is know whether they are needed**. This is especially relevant when instructions are **control-dependent** on unresolved branches.

In particular, hardware-based speculation enables:

- **Speculative instruction execution** before knowing branch outcomes.
- **Rollback** in case a mispredicted path was taken.

This approach increases Instruction-Level Parallelism (ILP), allowing more instructions to be in-flight, even if some turn out to be unnecessary.

Why speculation needs the ROB

Executing instructions speculatively introduces a challenge: "*how do we prevent side effects (e.g., register or memory updates) from mispredicted instructions?*". The **solution**: **defer the architectural state update** until it's safe to commit.

This is exactly what the **Reorder Buffer (ROB)** is for:

- ✓ It holds results of instructions that have **finished execution** but are **not yet committed**.
- ✓ It allows **instruction results to be written in-order**, preserving program semantics.
- ✓ In case of a misprediction, the ROB enables the processor to **flush invalid speculative results** quickly and precisely.

But the ROB is **critical** for:

- ⚠ Decoupling **execution completion** from **state update (commit)**.
- ⚠ Supporting **precise exceptions** (so the CPU can cleanly stop at the last correct instruction).
- ⚠ Managing **speculative and non-speculative instruction tracking**.

3.7.9.2 Why ROB is really needed

Modern high-performance processors use **out-of-order execution (OoO)** to improve instruction-level parallelism. However, they must still ensure **in-order commit** to preserve correct program behavior. This is where the **Reorder Buffer (ROB)** comes into play.

? Why Out-of-Order Execution and In-Order Commit?

Out-of-Order execution because:

- ✗ **Dependencies** (e.g., RAW hazards) and **delays** (e.g., memory access) prevent some instructions from being executed immediately.
- ✓ OoO execution **allows the processor to bypass stalled instructions** and execute independent ones earlier.
- This helps keep functional units busy and **improves throughput**.

For example, if an instruction is waiting for a memory load, another instruction, for example an ALU op, can execute immediately.

Unfortunately, Out-of-Order execution **introduces risk**:

- ⚠ Executing instructions out of order can break the **precise exception model**.
- ⚠ Also, **speculative** instructions (e.g., those after a branch) could be **incorrect**.

To preserve **program correctness**, **instructions must commit (retire) in order**, exactly as they appear in the original program.

✓ ROB's role in bridging OoO and In-Order Commit

The **Reorder Buffer (ROB)** is a hardware structure used in modern out-of-order processors to: **support out-of-order execution while enforcing in-order commitment of instructions**, ensuring correct architectural state and precise exception handling.

The ROB is the key mechanism enabling this balance:

- During **issue**: allocate an entry in the ROB, marking the instruction's place in program order.
- During **execution**: store the result in the ROB as soon as the instruction finishes.
- During **commit**: only commit the instruction (i.e., update architectural registers or memory) if:
  1. It is **at the head** of the ROB
  2. Its **execution has completed**

3. It is **not speculative**

This design ensures:

- ✓ Architectural **state** (register file, memory) is **updated in program order only**.
- ✓ **Speculative results** are kept in the ROB until validated.
- ✓ It is possible to **undo speculated instructions** if needed (e.g., branch misprediction).

The ROB allows a **processor to execute instructions as soon as their operands are ready**, regardless of program order, while **ensuring they commit their results in-order**.

❓ What about Precise Exception support?

An exception is *precise* if:

1. The processor can **stop at a well-defined point**, corresponding to a specific instruction.
2. All **previous instructions are fully committed**, and
3. **No subsequent instruction** has **modified** the architectural **state**.

This allows the OS or exception handler to reliably identify and handle the error. In **out-of-order execution**, instructions complete in a different order than they appear in the program. **Without careful control** instructions *after* the faulting one could have modified registers or memory and this would **leave the system** in an **inconsistent state**.

The **Reorder Buffer (ROB) solves** this by:

- ✓ **Storing all results temporarily**: Instructions write their output to the ROB instead of the register file or memory.
- ✓ **Committing results in-order**: Only when an instruction is at the **head** of the ROB and marked **completed**, its result is written to the architectural state.
- ✓ **Rejecting speculative results**: If an exception or misprediction occurs:
  - (a) The ROB **flushes all speculative entries after the faulting instructions**.
  - (b) Execution restarts from the faulting instruction or the exception handler.

  This rollback is clean because the **architectural state remains untouched** beyond the last committed instruction.

Example 11: Precise Exception support

Imagine a sequence:

```
Instr 1 → OK
Instr 2 → OK
Instr 3 → Exception (e.g., divide by zero)
Instr 4 → Executed early (OoO), wrote to reg R5
```

✗ Without the ROB:

- Instruction 4 might modify R5 *before* the exception at *Instr 3* is recognized.
- This makes the exception imprecise, corrupting the state.

✓ With the ROB:

- Instruction 4's result is held in the ROB.
- Since `Instr 3` caused an exception, no later instruction commits.
- `R5` is unaffected, precise state is preserved.

Functional Roles of the ROB

1. **Result Buffering**. Holds results of instructions that:
   - Have **completed execution**.
   - But have **not yet committed** to the architectural state (e.g., register file or memory).
2. **Speculative Result Propagation**. ROB acts as a **buffer to pass results among instructions that have started speculatively after a branch**. This allows speculative instructions (e.g., those after a predicted branch) to forward their results internally via the ROB without prematurely updating architectural registers.
3. **Precise Interrupt Support**. Originally introduced in 1988, the ROB was created to:
   - Preserve the **precise interrupt model**.
   - Guarantee that **only committed instructions affect** the architectural **state**.
   - Enable **rollback on branch misprediction or exceptions** by flushing speculative ROB entries.

The ROB is not merely a commit buffer, it is also a **speculation-aware result forwarding structure**, enabling safe communication among instructions that may never actually commit.

3.7.9.3 ROB as a Data Communication Mechanism

In Tomasulo's original algorithm, **Reservation Stations (RSs)** handled register renaming and operand forwarding. However, with the introduction of the ROB, it **replaces the renaming and forwarding function of RSs**. Before we explain how, we need to understand some *basic* concepts of ROB.

? What are ROB numbers?

A **ROB number** is simply an **index** (or tag, in Tomasulo's algorithm) **identifying** a specific **entry in the Reorder Buffer**. Each instruction that is issued receives a unique ROB number that corresponds to its place in the ROB, its **slot identifier** (slot ID).

The ROB number is then used in two key ways:

1. **As a destination tag (Register Renaming)**. When an **instruction** is issued and it **will write to a register**:

   - The destination register is **not renamed to another physical register** (like in a pure register renaming scheme).
   - Instead, the **architectural register is mapped to the instruction's ROB number**.

   It is the *identical* logic of tag in Tomasulo algorithm.

2. **For Operand Dependency Resolution**. Later **instructions that depend on the result** of the ROB number do not need to stall:

   - They **record the ROB number** as the operand source.
   - Once the ROB number becomes "ready" (i.e., the value is written to the ROB), dependent instructions can **read the value from the ROB number** and proceed to execution.

   In tomasulo algorithm this feature is provided by the CDB, where each instruction listens to that because it waits for the result of the tag.

   As happens in tomasulo algorithm, this mechanism:

   ✓ Avoids **WAR and WAW hazards**.
   ✓ Enables **data forwarding** even before instructions commit.

Tomasulo	ROB-based Tomasulo
Tags = RS entry IDs	Tags = ROB entry numbers
Values broadcast via CDB	Same, but identified by ROB number
RS buffers result locally	ROB stores result globally
Commit on write-back	Commit delayed and via ROB head

Table 21: Quick comparison between Tomasulo with and without ROB.

Updated Role of Reservation Stations

The ROB not only buffers instruction results but also **propagates those results to dependent instructions** as soon as they are ready, even if not yet committed. So, with the ROB managing renaming and data forwarding:

- ✗ Reservation Stations are **no longer responsible for naming/tagging**.
- ✓ Their role is now focused on:
  - **Buffering decoded instructions** before they're issued to the Functional Units (FUs);
  - **Holding operand values** (or ROB tags) temporarily;
  - Helping reduce **structural hazards**.

Reservation Stations now act like **staging areas**, not tracking results or resolving dependencies directly.

Function	Tomasulo	ROB-based Tomasulo
Register Renaming	RS	ROB
Data forwarding	RS	ROB
Instruction buffering	RS	RS (same)
Operand availability tracking	RS	RS (via ROB tags)

Table 22: Summary of architectural changes.

In modern speculative Tomasulo architectures, the ROB becomes the central structure for both result tracking and inter-instruction communication. RSs are demoted to lightweight instruction and operand buffers.

3.7.9.4 Architecture

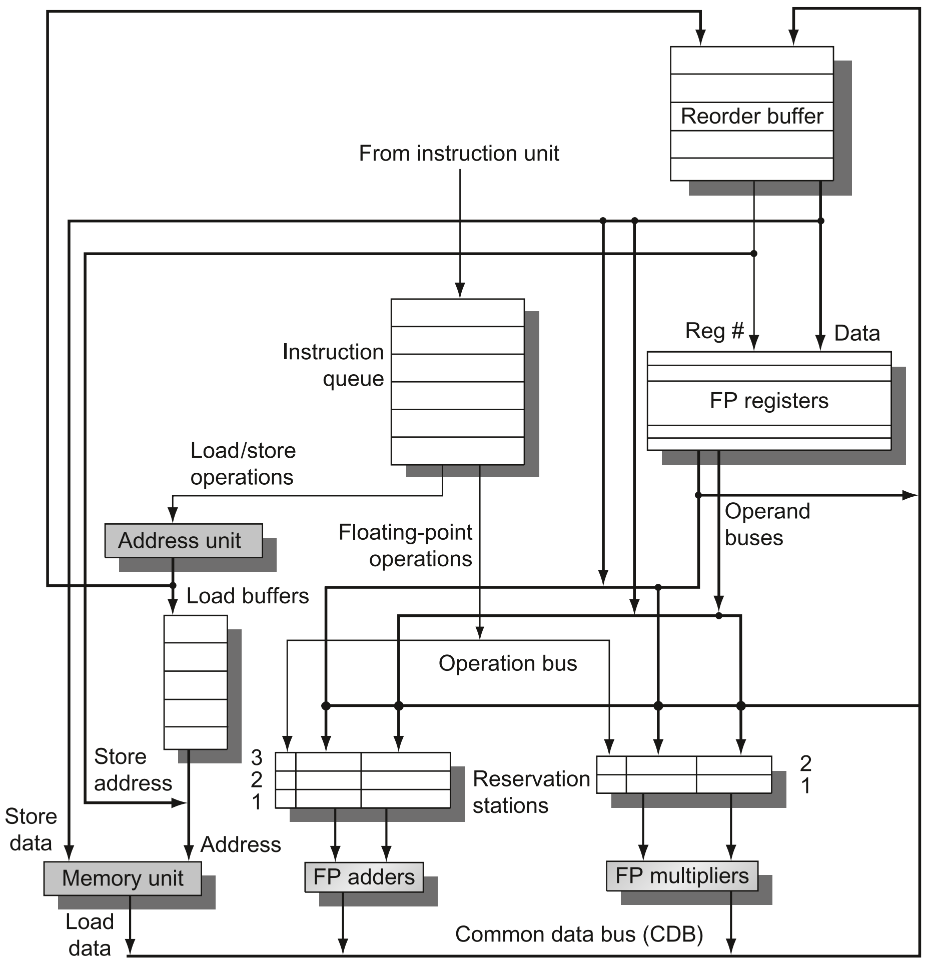

Figure 27: The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation (ROB). [1]

This architecture shows a **speculative version of Tomasulo's algorithm**, integrating a Reorder Buffer (ROB) and eliminating the classic **store buffer**. It's designed for **floating-point**, but the same design applies to general OoO pipelines.

- **Instruction Queue**. Holds fetched instructions awaiting issue. Supplies instructions to **reservation stations** and the **ROB** in parallel.
- **Reservation Stations**. Acts as **temporary buffers** between issue and execution. Each FP unit (adder, multiplier) has dedicated RSs. Stores instructions with operand tags or values. Unlike the classic Tomasulo, **renaming is handled via the ROB**.
- **Floating-Point Units**. Includes **FP Adders** and **FP Multipliers**. Execution units receive instructions from RSs when operands are ready.

- **Common Data Bus (CDB).** Used to broadcast result values and their corresponding **ROB tag**. All waiting instructions and ROB entries **listen** to the CDB.
- **Reorder Buffer (ROB).** Central to this architecture, replaces:
  1. Register renaming logic
  2. Store buffers

  Each ROB entry holds critical metadata about one in-flight instruction:

  - **Busy field.** Indicates whether the ROB entry is currently active (busy) or free (available).
  - **Instruction type field.** Specifies the instruction category:
    * Branch (no destination result),
    * Store (destination is memory address),
    * Load/ALU (destination is a register).
  - **Destination field**
    * For load and ALU instructions: **target register** number.
    * For store instructions: **memory address** where the value must be written.
  - **Value field.** Holds the **result** of the instruction after execution, kept until commit.
  - **Ready Bit.** Set when execution has completed and the result is valid.
  - **Speculative Flag.** Shows whether the instruction is executed speculatively (e.g., after a predicted branch) or not.
- **Load Buffers & Address Unit.**

  Load Buffers is a **queue or small table** that holds load instructions **waiting to access memory**. It **temporarily buffers** loads *after issue* and *before* they actually perform a memory access. Its purpose is shown in the example on page 187.

  Address Unit is a specialized hardware block **dedicated to calculating effective addresses** for loads and stores. Given a base register value and an offset, it computes an effective address.

  ❓ How Load Buffers and Address Unit Work Together?

  1. **Issue Stage**: the instruction is issued. Allocates an entry in the ROB and Load Buffer.
  2. **Address Calculation**: Address Unit computes the effective address.
  3. **Memory Access Decision**: if there is no preceding store to the same address (or speculation allows), the load can access memory early. If not, the load must wait until memory disambiguation clears it.
  4. **Load Execution**: read data from memory. Write the result into the ROB entry. When ready and safe, commit to the architectural register file.

- **Register File (FP Registers)**: not updated directly after execution. Instead, it is **updated only when instructions commit via the ROB**, preserving the **precise state model**.

Key Innovations in this design

- **ROB replaces store buffers**: the store buffer doesn't exist anymore, now memory writes are delayed until commit.
- **Register renaming is moved from RSs to ROB**: simplifies dependency tracking.
- **Precise exceptions and speculative execution are both supported** via ROB tracking and flushing.
- **CDB remains critical** for result forwarding and readiness detection.

Ready Bit

The **Ready Bit** is a control bit that indicates whether the instruction associated with this ROB entry has **completed execution** and its **result is available**. It is **set to true** when:

1. The instruction finishes execution.
2. The result is written into the ROB.

Its purpose is to **inform dependent instructions that they can now read this value** (via the CDB or directly from the ROB). Also, it enables commit: an instruction can **only commit if its ready bit is set**.

In other words, the ready bit **tracks the availability of the computed result**.

Speculative Flag

The **Speculative Flag** is a control bit that indicates whether the **instruction was issued after an unresolved branch prediction** (or other speculative control decision). It is:

- **Set to true** only when the instruction is **issued speculatively**, <u>before</u> the control flow is confirmed.
- **Cleared** when the speculation is resolved (e.g., branch prediction validated).

Its purpose is to ensure that **speculative instructions do not prematurely update** the architectural state. Allows all speculative instructions to be **flushed** upon misprediction or exception.

In other words, the speculative flag tracks whether the instruction is **tentative** or **safe**.

ROB Structure and Operation: Circular Buffer

The Reorder Buffer (ROB) is implemented as a **circular buffer** (circular FIFO, First-In-First-Out, queue), managed with **two pointers**:

- **Head pointer**. Points to the **oldest instruction** that is **next to commit** (i.e., retire).

  The head **advances** when instructions commit.

- **Tail pointer**. Points to the **next free entry**, where a new issued instruction will be inserted.

  The tails **advances** when new instructions are issued.

Example 12: ROB Circular Buffer

`ROB#`	`BUSY`	`INSTR. TYPE`	`READY`	`DEST`	`VALUE`	`SPEC`
`ROB0`	Yes	In	No	`F0`	–	No
`ROB1`	No	–	–	–	–	–
`ROB2`	No	–	–	–	–	–
`ROB3`	No	–	–	–	–	–

- **Head** points at `ROB0` (first to commit once ready).
- **Tail** points at `ROB1` (next free entry for a new instruction).

Why Two Pointers?

The head and tail pointers allow the ROB to **efficiently manage** dynamic instruction issue and commit while **preserving program order**, and they **minimize hardware complexity** by avoiding costly entry movement.

1. **Sequential Insertion at the Tail (Issue Stage)**. When a new instruction issues, it needs a free ROB entry. The tail pointer indicates where to insert the new instruction. After insertion, the tail advances to the next free slot. This preserves the program order at issue time.

   New instructions always go at the tail.

2. **Sequential Retirement from the Head (Commit Stage)**. When an instruction completes and satisfies all commit conditions (ready, not speculative, prior instructions retired), it retires. The head pointer shows which instruction should commit next. After commit, the head advances to the next oldest instruction.

   Commit always happens starting from the head, ensuring in-order commit.

3. **Circular Buffer Efficiency**. Memory and hardware are limited: we don't want an infinitely growing ROB. Using a circular buffer:

   - When the tail reaches the end of the buffer, it wraps around to position 0.
   - Same for the head.

   This reuses space efficiently without needing to shift entries manually.

   Circular structure avoids expensive data movement and saves silicon area.

4. **Detecting Full and Empty Conditions**. With just head and tail:

   - If head = tail and entry busy → ROB is full (cannot issue more instructions).
   - If head = tail and entry not busy → ROB is empty (no instructions to retire).

   Simple hardware checks based on two pointers.