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.