ChatGPT

Tomasulo's Algorithm

Tomasulo's algorithm is a hardware scheme for dynamic (out-of-order) scheduling of instructions that maximizes the use of multiple functional units [1]. Its key ideas are **register renaming**, **reservation stations**, and a **common data bus (CDB)** on which computed results are broadcast to any waiting units [2]. In modern implementations a **reorder buffer (ROB)** is added so results are *commit*(ed) in program order, providing precise exceptions and in-order write-back [3] [4]. In essence, instructions issue in program order but may execute and complete out of order, while the CDB and ROB ensure correct data flow and final write-back.

Core Components

- **Reservation Stations (RS):** Reservation stations are small buffers associated with each functional unit (FU). An RS holds a pending instruction until it can execute. Each entry contains fields for the operation type, the values of source operands (Vj, Vk) or tags (Qj, Qk) identifying the RS/ROB producing those operands, an address field (A) for load/store operations, and a "busy" flag [5] [6]. For example, a floating-point ADD unit might have several RS entries; each entry stores one ADD instruction's opcode and operands. When the FU and operands are ready, the RS dispatches the instruction to the FU. The RS thus isolates the FU from hazards by buffering instructions and tracking dependencies [5] [6].

- **Register Status (Register Renaming):** The processor maintains a *register status* table (or *register alias table*) that records, for each architectural register, which reservation station (or ROB entry) will eventually produce its next value [7]. When an instruction is issued, its destination register's status is updated to point to the RS/ROB where its result will appear. If the register has no pending producers, it is marked free. This renaming table avoids write-after-write (WAW) and write-after-read (WAR) hazards: each new write to a register gets a new tag, and instructions read values by tag, not by fixed register name. In practice, name-dependent instructions use the tag of the producing RS (or ROB) as their source, and "only the last instruction to write to a register actually updates it" [8]. Thus, WAR/WAW conflicts vanish because the first writer's tag will be overwritten by a later writer's tag, and earlier readers have already captured the correct value by tag [8].

- **Common Data Bus (CDB):** The CDB is a broadcast bus connecting all FUs, RS entries, and the ROB. When a functional unit finishes an operation, it broadcasts its result (with an identifying tag) on the CDB [9]. All reservation stations monitor the CDB: if an RS has a source tag (Qj or Qk) matching that tag, it loads the value into Vj or Vk and clears the tag. In addition, the result is written into the ROB entry. This immediate forwarding means dependent instructions need not wait for a register write – the value arrives directly from the producing FU. As one lecture note explains, "produced results are immediately forwarded to functional units on the CDB…so [instructions] don't have to wait to write into the register file" [10]. The CDB thus eliminates the need for extra register-file read ports and resolves RAW hazards by direct data forwarding.

- **Reorder Buffer (ROB):** The ROB is a circular buffer that holds the results of instructions until they can be committed in program order [3]. In Tomasulo's original (IBM 360/91) design there was no ROB, but most modern OoO designs use one. In an extended Tomasulo pipeline, when an instruction issues it is assigned an ROB entry (index). On completion, the FU writes the result into that ROB entry (instead of directly into the register file) and broadcasts on the CDB [11]. Each ROB entry holds fields like the instruction type, destination register or memory address, the result value, and a ready bit [12]. Instructions only *commit* (update the architectural state) when they reach the head of the ROB and their result is ready. This in-order commit ensures *precise exceptions* and correct program order. In particular, since the pipeline can complete instructions out of order but only commit them sequentially, the processor can always roll back to the last known committed state if a misprediction or exception occurs [4]. The ROB thus enforces that "results are committed in order... preventing data hazards such as RAW, WAR and WAW" [13] and providing precise exception semantics [4].

Instruction Flow: Issue, Execute, Write Result

Issue Stage: The next program instruction is fetched in order and issued if resources are available. Issuing does the following: (1) Allocate a free Reservation Station (and an ROB entry). (2) For each source operand, consult the Register Status table. If the register has no pending tag (meaning its value is already ready), the operand value is read from the register file into Vj or Vk. If the register is waiting on a previous instruction, the RS records the tag (from Register Status) in Qj or Qk, leaving the value field empty [14]. (3) Record the instruction's opcode and any immediate/address (A). (4) Mark the RS busy and update the Register Status of the destination register to point to this RS (or ROB index). This renaming means any younger instruction needing that register will use this new tag, eliminating WAW (since older writes will be overwritten) and WAR hazards (since any earlier reads have already been satisfied) [15] [16]. If no RS (or ROB slot) is available, issuance stalls until one frees. In pseudocode, we essentially "rename the register to the reservation station that will produce it" for operands not yet ready [14] and update the dest-reg alias.

Execute Stage: In parallel with fetching new instructions, each RS monitors whether its operands are ready (i.e. Qj=Qk=0). When an instruction's sources are all available (having been provided at issue or via earlier CDB broadcasts), that instruction is dispatched to its functional unit and begins execution [17]. While executing (e.g. a multi-cycle add or multiply), the RS remains busy. There is no RAW hazard at this point, since the operands were locked in by renaming/forwarding. For load/store instructions, there may be separate steps: the effective address A is computed when base register is ready; a load waits for any required memory access; a store buffers its value. In any case, once the FU completes the operation, it produces a result value.

Write-Result Stage (and Commit): When an FU finishes an instruction, it writes the result into the ROB entry and broadcasts the value and its tag on the Common Data Bus [11] [9]. Any RS whose Qj or Qk matches the tag grabs the value into Vj/Vk and clears its tag. If not using an ROB, the result would simply be written to the register file here; in Tomasulo with ROB, the architectural register is updated later at commit. After broadcasting, the reservation station is freed for new instructions. Finally, as soon as the instruction's result is in the ROB, it may *commit*. The ROB's head entry commits when it is ready: its result is written into the destination register (or memory) and the entry is retired. Only one head instruction commits per cycle, preserving program order. Because all updates to state (registers/memory) happen in ROB commit stage, the processor maintains precise state. If a branch misprediction or exception occurs, any instructions later

in the ROB are simply discarded [4]. In short, the ROB "allows the pipeline to continue OoO while ensuring results are committed in order" [13], providing precise exceptions and correct in-order retirement.

Throughout this flow, **register renaming** plays a crucial role: by giving each pending write a unique tag, the algorithm removes false dependencies. WAR and WAW hazards cannot occur because earlier writes are renamed away. As one source notes, after renaming "only the last instruction to write to a register updates it" [16], and name dependencies are handled by using tags (RS identifiers) instead of fixed register names [16]. **The Common Data Bus** ensures completed values are immediately visible: it "preserves precedence while encouraging concurrency" [2] by broadcasting results to all waiting units. Finally, **the Reorder Buffer** guarantees in-order commit. Only when an instruction reaches the ROB head and its result is ready does it update the register file. This in-order commit stage creates precise exceptions and allows speculative or out-of-order work to be rolled back safely [4] [18].

Key Takeaways: Reservation stations hold pending instructions with operand values or tags; the register status table does the renaming. On issue, operands are read or tagged, and the destination register's status is set. In execute, each RS waits until its sources arrive (via CDB) then fires the FU. In write-result, completed results go on the CDB (waking dependents) and into the ROB. Finally, the ROB commits results in program order, ensuring correct architectural state and precise exception handling [9] [4]. These mechanisms together allow multiple instructions to execute in parallel without data conflicts.

Sources: Classic computer-architecture references and lecture notes on Tomasulo's algorithm [2] [19] [10] [8] [13] [4].

---

[1] [2] [6] [9] Tomasulo's algorithm - Wikipedia
https://en.wikipedia.org/wiki/Tomasulo%27s_algorithm

[3] [4] [11] [12] [13] [18] Re-order buffer - Wikipedia
https://en.wikipedia.org/wiki/Re-order_buffer

[5] [7] [14] [15] [17] [19] Microsoft PowerPoint - ch3
https://people.cs.pitt.edu/~melhem/courses/2410p/ch3-4.pdf

[8] [10] [16] Microsoft PowerPoint - tomasuloStudent.pptx [Read-Only]
https://courses.cs.washington.edu/courses/cse471/10sp/lectures/tomasuloStudent.pdf