

1 Tomasulo's Algorithm

Recall from last week that a naive out-of-order pipeline is bottlenecked by WAR and WAW hazards. These are *false* dependencies: they come from reusing a small set of architectural registers, and would not exist if we had an unlimited number of registers. **Register renaming** removes them by mapping each new write of an architectural register to a different physical register.

Original code:

A: fmul f1, f0, f2
B: fadd f0, f3, f1
C: fmul f3, f2, f3
D: fadd f3, f3, f1

Renamed code:

A: fmul P4, P0, P2
B: fadd P5, P3, P4
C: fmul P6, P2, P3
D: fadd P7, P6, P4

Rename table		
	Initial	Updated
f0	P0	P5
f1	P1	P4
f2	P2	
f3	P3	P6 P7

Renaming in practice. The IBM 360/91 had only four floating-point registers (f0–f3), too few to keep its FP pipeline busy. Tomasulo's algorithm (1967) renames each destination to a reservation station entry:

- **Register file:** each entry has a present bit *p* and holds either a value or the tag of the entry that will produce it. Indices map to ISA registers (f0, f1, ...).
- **Reservation stations:** each source operand is a <*p*, tag/data> pair. The tag of an RS entry is the new name for that instruction's result.
- **Common data bus:** a finishing functional unit broadcasts <tag, result>. Every RF and RS entry waiting on that tag copies the result and sets *p*. We call results V0, V1, and so on.

The algorithm:

- **On dispatch** (in program order):
 - Allocate a reservation station (RS) entry. Its ID is the tag.
 - For each source register, if its register file (RF) entry has *p* set, copy the value into the RS and set *p* for that operand. Otherwise, copy the tag from the RF and clear *p*.
 - Write the destination register's RF entry as <*p* = 0, tag>.
- **Before execution:**
 - For missing operands, monitor the result bus for a matching tag. Replace the tag with the value and set *p*.
 - When all operands are present, issue to the functional unit.
- **On completion:**
 - Broadcast <tag, result> on the result bus for the RF and other RS entries to consume.
 - Deallocate the RS entry.

- 1.1 Trace the code below through a Tomasulo machine with three adder reservation stations (T0–T2) feeding a pipelined 2-stage adder, and two multiplier reservation stations (T3–T4) feeding a pipelined 3-stage multiplier. The comments name each instruction's result. Assume:
- At most one instruction dispatches per cycle, in program order, into the lowest-numbered free RS of the right type.
 - An instruction whose operands are all present can issue in the same cycle it dispatches.
 - A result is broadcast in the cycle after its last execute stage. In that cycle, the RF and every waiting RS entry (including one being dispatched) pick it up, the producer's RS entry is freed, and any instruction that is now ready can issue.
 - The bus can broadcast more than one result per cycle.

Fill in the state at the end of each cycle. The initial state is given; the initial value of f1 is never read, so it is left blank.

A: fmul f1, f0, f2 # V3
 B: fadd f0, f3, f1 # V4
 C: fmul f3, f2, f3 # V5
 D: fadd f3, f3, f1 # V6

Cycle 0 Dispatched: –

		p	tag/data	p	tag/data
Adder RS	T0				
	T1				
	T2				
Mult RS	T3				
	T4				

	Stage	Dest tag
Adder	1	
	2	
Mult	1	
	2	
	3	

Reg	p	tag/data
f0	1	V0
f1		
f2	1	V1
f3	1	V2

A is fetched.

Cycle 1 Dispatched: –

		p	tag/data	p	tag/data
Adder RS	T0				
	T1				
	T2				
Mult RS	T3				
	T4				

	Stage	Dest tag
Adder	1	
	2	
Mult	1	
	2	
	3	

Reg	p	tag/data
f0	1	V0
f1		
f2	1	V1
f3	1	V2

A is decoded.

Cycle 2 Dispatched: A: `fmul f1, f0, f2`

		p	tag/data	p	tag/data
Adder RS	T0				
	T1				
	T2				
Mult RS	T3	1	V0	1	V1
	T4				

	Stage	Dest tag
Adder	1	
	2	
Mult	1	T3
	2	
	3	

Reg	p	tag/data
f0	1	V0
f1	0	T3
f2	1	V1
f3	1	V2

A gets T3. Both operands are present, so it copies V0 and V1 and issues to the multiplier right away. f1 now waits on T3.

Cycle 3 Dispatched: B: `fadd f0, f3, f1`

		p	tag/data	p	tag/data
Adder RS	T0	1	V2	0	T3
	T1				
	T2				
Mult RS	T3	1	V0	1	V1
	T4				

	Stage	Dest tag
Adder	1	
	2	
Mult	1	
	2	T3
	3	

Reg	p	tag/data
f0	0	T0
f1	0	T3
f2	1	V1
f3	1	V2

B gets T0. f3 is present, but f1 is not, so B copies the tag T3 and waits. f0 now waits on T0.

Cycle 4 Dispatched: C: `fmul f3, f2, f3`

		p	tag/data	p	tag/data
Adder RS	T0	1	V2	0	T3
	T1				
	T2				
Mult RS	T3	1	V0	1	V1
	T4	1	V1	1	V2

	Stage	Dest tag
Adder	1	
	2	
Mult	1	T4
	2	
	3	T3

Reg	p	tag/data
f0	0	T0
f1	0	T3
f2	1	V1
f3	0	T4

C gets T4, copies V1 and V2, and issues right away, so the pipelined multiplier now holds both A and C. f3 now waits on T4. C can rename f3 even though B has not executed yet, because B already copied V2 into its RS entry (no WAR hazard).

Cycle 5 Dispatched: D: `fadd f3, f3, f1`

		p	tag/data	p	tag/data
Adder RS	T0	1	V2	1	V3
	T1	0	T4	1	V3
	T2				
Mult RS	T3				
	T4	1	V1	1	V2

	Stage	Dest tag
Adder	1	T0
	2	
Mult	1	
	2	T4
	3	

Reg	p	tag/data
f0	0	T0
f1	1	V3
f2	1	V1
f3	0	T1

A broadcasts <T3, V3>: f1 and B's second operand pick up V3, and T3 is freed. B now has both operands and issues to the adder. D gets T1, picking up V3 for f1 and the tag T4 for f3. f3 now waits on T1.

Cycle 6 Dispatched: **None**

		p	tag/data	p	tag/data
Adder RS	T0	1	V2	1	V3
	T1	0	T4	1	V3
	T2				
Mult RS	T3				
	T4	1	V1	1	V2

	Stage	Dest tag
Adder	1	
	2	T0
Mult	1	
	2	
	3	T4

Reg	p	tag/data
f0	0	T0
f1	1	V3
f2	1	V1
f3	0	T1

Nothing dispatches or broadcasts. B moves to adder stage 2, and C moves to multiplier stage 3.

Cycle 7 Dispatched: **None**

		p	tag/data	p	tag/data
Adder RS	T0				
	T1	1	V5	1	V3
	T2				
Mult RS	T3				
	T4				

	Stage	Dest tag
Adder	1	T1
	2	
Mult	1	
	2	
	3	

Reg	p	tag/data
f0	1	V4
f1	1	V3
f2	1	V1
f3	0	T1

B broadcasts <T0, V4>, so f0 becomes V4. C broadcasts <T4, V5>, so D picks up V5 and issues. f3 ignores V5 because it now waits on T1: V5 never reaches the register file, which resolves the WAW hazard between C and D.

Cycle 8 Dispatched: **None**

		p	tag/data	p	tag/data
Adder RS	T0				
	T1	1	V5	1	V3
	T2				
Mult RS	T3				
	T4				

	Stage	Dest tag
Adder	1	
	2	T1
Mult	1	
	2	
	3	

Reg	p	tag/data
f0	1	V4
f1	1	V3
f2	1	V1
f3	0	T1

D moves to adder stage 2.

Cycle 9 Dispatched: **None**

		p	tag/data	p	tag/data
Adder RS	T0				
	T1				
	T2				
Mult RS	T3				
	T4				

	Stage	Dest tag
Adder	1	
	2	
Mult	1	
	2	
	3	

Reg	p	tag/data
f0	1	V4
f1	1	V3
f2	1	V1
f3	1	V6

D broadcasts <T1, V6>, so f3 becomes V6 and T1 is freed. All reservation stations are now empty.

1.2 Why can't the reservation station entry for an instruction be deallocated immediately on issue?

The RS entry's ID is the tag that names the instruction's result. If it were freed on issue, a later instruction could be given the same entry, and therefore the same tag, before the first instruction writes back:

```
A: fmul f4, f0, f1    # dispatched and issued immediately; RS entry freed  
B: fmul f5, f2, f3    # allocated the same RS entry before A has written back
```

Now f4 and f5 are both waiting on the same tag in the register file, so A's broadcast incorrectly clobbers f5 on writeback.

- 1.3 Why are exceptions *imprecise* in this implementation? Can the current setup support hardware speculation (for example, branch prediction)?

Imprecise exceptions: the register file is irrevocably modified on dispatch, and results are written back out of order. If an instruction causes an exception, there is no mechanism to recover the original value of its destination register, and younger instructions may already have updated the register file.

Speculation: no. There is no notion of committing an instruction, so if we guess a branch incorrectly, there is no way to undo the wrong-path instructions that have already updated the register file.

2 Reorder Buffer

To regain precise exceptions and support speculation, the **reorder buffer (ROB)** separates *commit* from *completion*:

- **Completion:** the result is available (out of order).
- **Commit:** architectural state is updated (in order).

	v	i	op	p	rs1/tag	p	rs2/tag	p	result	rd	xcpt?
oldest →											
free →											

Each ROB entry holds a valid bit (v), an issued bit (i), the opcode, present bits and tags or values for both sources and the result, the destination register (rd), and an exception flag. Instructions enter at the free pointer and commit from the oldest pointer.

Terminology. We use the following names for the phases of instruction execution:

Phase	What happens
Fetch	Instruction bits are retrieved from the instruction cache into the fetch buffer.
Decode / rename	Instructions are dispatched into the appropriate issue buffer (ROB / reservation station).
Execute	Instructions and their operands are issued to functional units. When execution completes , all results and exception flags are available.
Commit	The instruction irrevocably updates architectural state, or takes a precise trap. Also called <i>graduation</i> or <i>retirement</i> .

- **Dispatch vs. issue:** dispatch means the instruction is decoded and enters the ROB / reservation station; issue means it is sent to an execution unit. Some papers and diagrams use “issue” for what we call dispatch.
- **Completion vs. commit:** committing in order despite completing out of order is what gives us precise exceptions. “Retired” is also used to mean “committed”.

There are two common ways to organize the ROB and register state:

	Data-in-ROB	Unified physical register file
Uncommitted values	Held in the ROB, which holds both tags and data	Held in the physical register file, alongside committed values
Architectural state	Separate architectural register file, written at commit	A committed register mapping into the physical register file
ROB contents	Tags and data	Only tags

3 Renaming with a Unified Physical Register File

With a unified physical register file (PRF), renaming works as follows:

- **On dispatch**, provide a fresh location for the new result:
 - (a) Allocate a new physical register for the destination from the free list.
 - (b) Update the decode-stage mapping.
- **On commit**, make the mapping “permanent” in the architectural state, and return the old physical register to the pool for reuse. This is why we want in-order commit!
 - (a) Update the architectural mapping.
 - (b) Deallocate the previous physical register for the destination, and add it back to the free list.
- **On exception**, repair the decode-stage rename table by un-renaming in reverse order: walk through the ROB entries from newest to oldest (the [MIPS R10K](#) approach).

3.1 The x registers below are renamed from a pool of physical registers. The free list is a FIFO: a physical register is taken from the head when allocated and added back to the tail when reclaimed. For each instruction, give the physical register assigned as its destination, and the physical register added back to the free list when it commits. Then give the map table after all ten instructions are renamed, and the free list after all ten commit.

Initial map table:

Arch. reg	x1	x2	x3	x4	x5
Phys. reg	p12	p11	p5	p6	p3

Initial free list: head →

p2	p4	p10	p7	p1	p0	p9	p8	p13	p14
----	----	-----	----	----	----	----	----	-----	-----

Instruction	Arch. destination	Phys. destination	To be freed
ld x2, 0(x4)	x2	p2	p11
sw x2, 0(x3)	–	–	–
addi x4, x4, 0x4	x4	p4	p6
addi x3, x3, 0x4	x3	p10	p5
bne x4, x1, loop	–	–	–
ld x2, 0(x4)	x2	p7	p2
sw x2, 0(x3)	–	–	–
addi x4, x4, 0x4	x4	p1	p4
addi x3, x3, 0x4	x3	p0	p10
bne x4, x1, loop	–	–	–

Map table after renaming:

Arch. reg	x1	x2	x3	x4	x5
Phys. reg	p12	p7	p0	p1	p3

Free list after all ten commit: head →

p9	p8	p13	p14	p11	p6	p5	p2	p4	p10
----	----	-----	-----	-----	----	----	----	----	-----

Stores and branches have no destination register, so they neither allocate nor free a physical register.

- 3.2 Suppose an older instruction A, still in flight, was assigned p11 as its destination, and the first `ld x2, 0(x4)` (call it B) is assigned p2 and will free p11. B finishes executing in 1 cycle, but A takes 100 cycles. When can p11 be added back to the free list? The map table now says x2 maps to p2, so how does A know where to write its result?

p11 can only be freed when B commits, and B cannot commit until A has committed, even though B finished first. Because B is stuck waiting to commit, p11 stays reserved.

A does not look at the map table after dispatch. When A was dispatched, it was told “your result lives in p11”, and that destination tag is tied to A in the issue window / reservation station. A writes p11 whenever it finishes, and in-order commit guarantees that p11 is not reused until A (and every older instruction that reads p11) has retired.

- 3.3 Suppose the second load instruction causes an exception. Show the state of the map table and free list just before jumping to the exception handler. Assume the exception is taken when the load reaches the head of the ROB, so all older instructions have committed.

Map table:

Arch. reg	x1	x2	x3	x4	x5
Phys. reg	p12	p2	p10	p4	p3

Free list: head →

p7	p1	p0	p9	p8	p13	p14	p11	p6	p5
----	----	----	----	----	-----	-----	-----	----	----

Walk the ROB from newest to oldest and undo each rename: the second `addi x3` returns p0 (x3 back to p10), the second `addi x4` returns p1 (x4 back to p4), and the faulting `ld` returns p7 (x2 back to p2). The free list head moves back to p7. The first five instructions have committed, freeing p11, p6, and p5 to the tail of the free list.

4 Branch Prediction

Modern processors may have more than 10 pipeline stages between the next PC calculation and branch resolution. Each mispredicted branch costs roughly (pipeline depth from fetch to branch resolution) $\times$ (pipeline width), plus the instructions sitting in buffers such as the fetch queue.

Overall, branches are taken about 60–70% of the time, but backward branches (loops) are taken about 90% of the time, while forward branches are closer to 50%. Dynamic predictors also exploit *correlation* between branch outcomes:

Temporal correlation:

```
for (int i = 0; i < 4; i++)
    sum += i;
```

Spatial correlation:

```
if (x[i] < 7) then
    y += 1;
if (x[i] < 5) then
    c -= 4;
```

Bimodal counters. A 2-bit saturating counter is incremented when the branch is taken and decremented when it is not taken. The upper bit gives the predicted direction.

00	01	10	11
Strongly not taken	Weakly not taken	Weakly taken	Strongly taken

Consider the following code:

```
for (int i = 0; i < 4; i++)      // BRANCH_A (taken = stay in the loop)
    for (int j = 0; j < 4; j++) // BRANCH_B (taken = stay in the loop)
        if (j % 2 != 0)        // BRANCH_C (taken = run the body of the if)
            sum += i;
```

Assume a branch history table (BHT) indexed by PC, and that the PCs of the three branches do not alias into the same entry. In the tables, 1 means taken and 0 means not taken, and the counter is the value before the branch is predicted.

- 4.1 With **1-bit** counters initialized to 0 (not taken), fill in the tables. BRANCH_A is done for you. What is the prediction accuracy for each branch over the whole loop nest?

Branch A:

	$i = 0$	$i = 1$	$i = 2$	$i = 3$	$i = 4$
Counter	0	1	1	1	1
Prediction	0	1	1	1	1
Actual	1	1	1	1	0
Accuracy	3/5				

Branch B:

	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 0$	$j = 1$
Counter	0	1	1	1	1	0	1
Prediction	0	1	1	1	1	0	1
Actual	1	1	1	1	0	1	1
Accuracy	12/20 (3/5 per outer iteration)						

Branch C:

	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 0$	$j = 1$	...
Counter	0	0	1	0	1	0	...
Prediction	0	0	1	0	1	0	...
Actual	0	1	0	1	0	1	...
Accuracy	1/16						

- BRANCH_A mispredicts the first iteration and the loop exit: 3/5.
- BRANCH_B mispredicts $j = 0$ (the previous exit reset the counter to 0) and $j = 4$ (the exit) in every outer iteration: 3/5 each, so 12/20.
- BRANCH_C alternates, and a 1-bit counter always predicts the previous outcome. Only the very first prediction is right (the counter starts at not taken, and $j = 0$ skips the body). Every prediction after that is wrong, so 1/16, or 0/4 per outer iteration in steady state.

4.2 Repeat with **2-bit** counters initialized to 00 (strongly not taken).

Branch A:

	$i = 0$	$i = 1$	$i = 2$	$i = 3$	$i = 4$
Counter	00	01	10	11	11
Prediction	0	0	1	1	1
Actual	1	1	1	1	0
Accuracy	2/5				

Branch B:

	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 0$	$j = 1$
Counter	00	01	10	11	11	10	11
Prediction	0	0	1	1	1	1	1
Actual	1	1	1	1	0	1	1
Accuracy	14/20						

Branch C:

	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 0$	$j = 1$	...
Counter	00	00	01	00	01	00	...
Prediction	0	0	0	0	0	0	...
Actual	0	1	0	1	0	1	...
Accuracy	8/16						

- BRANCH_A mispredicts twice while the counter warms up, plus the exit: 2/5.

- `BRANCH_B` gets 2/5 in the first outer iteration, like `BRANCH_A`. After that, the exit only drops the counter from 11 to 10, so each of the 3 remaining outer iterations mispredicts only the exit (4/5): $(2 + 4 \times 3)/20 = 14/20$.
- `BRANCH_C` bounces between 00 and 01 and always predicts not taken, so it is right on the half where the body is skipped: 8/16.

Hysteresis helps `BRANCH_B`, because a single not-taken outcome no longer flips the prediction. No bimodal counter can learn `BRANCH_C`'s alternating pattern.

Branch history tables. What we just analyzed is a **branch history table (BHT)**: some bits of the PC are used as a hash into a table of counters that store the direction (taken / not taken) of each branch. This captures temporal correlation. Specifically, it is a *single-level* BHT: one table, and no spatial correlation. A 4K-entry BHT with 2 bits per entry predicts about 80–90% of branches correctly.

Two-level predictors also exploit spatial correlation, using the observation that the dynamic behavior of past branches can predict future branches. A **branch history** is a bit vector of taken / not-taken (1 / 0) outcomes, representing both temporal and spatial history. The history is combined with the PC to select a counter. It can be *local* (each branch has its own shift register) or *global* (one shift register shared by all branches).

- 4.3 Consider a two-level predictor, where a branch history selects a prediction from multiple PC-indexed counter tables, running the loop below. Assume no branch aliasing, and that $x[i]$ is random.

```
for (int i = 0; i < N; i++) {    // BRANCH_L (taken = stay in the loop)
    if (x[i] >= 0) pos++;        // BRANCH_1 (taken = run the body)
    if (i % 2 == 0) even++;      // BRANCH_2 (taken = run the body)
    if (x[i] < 0) neg++;         // BRANCH_3 (taken = run the body)
}
```

What is the **minimum** number of history bits needed to predict each branch perfectly? If using local histories, how many local history bits? If using global histories, how many global history bits?

	Local history bits	Global history bits
BRANCH_1	Impossible: depends on random data	Impossible
BRANCH_2	1	4: alternates T N T N, and its last outcome is 4 branches back
BRANCH_3	Impossible	2: $x[i] \geq 0$ (BRANCH_1) tells you the answer

Each iteration runs BRANCH_L, BRANCH_1, BRANCH_2, BRANCH_3 in that order.

- BRANCH_1 depends only on the random $x[i]$, so no history can predict it.
- BRANCH_2 alternates, so 1 bit of its own history is enough. In the global history just before it, the most recent outcomes are BRANCH_1, BRANCH_L, BRANCH_3 (previous iteration), and BRANCH_2 (previous iteration), so we need 4 bits to reach its own last outcome.
- BRANCH_3's own history is random, but it is taken exactly when BRANCH_1 (2 branches back) was not taken, so 2 global bits are enough.

BHTs vs. BTBs. A BHT provides a direction, but to redirect fetch we need an actual address. That comes from a **branch target buffer (BTB)**: a 2^k -entry table indexed by the PC (direct-mapped, or associative), where each entry holds the branch's PC, a valid bit, and its predicted target PC.

- The next PC is determined before the branch is even fetched and decoded.
- If the PC does not match an entry, PC+4 is fetched.
- Only taken branches and jumps are held in the BTB.

The BTB is consulted in the first fetch stage. The BHT is consulted later, once the instruction is being decoded, and corrects the fetch stream when the BTB misses a predicted-taken branch. Both are only updated after the branch resolves in the execute stage.