



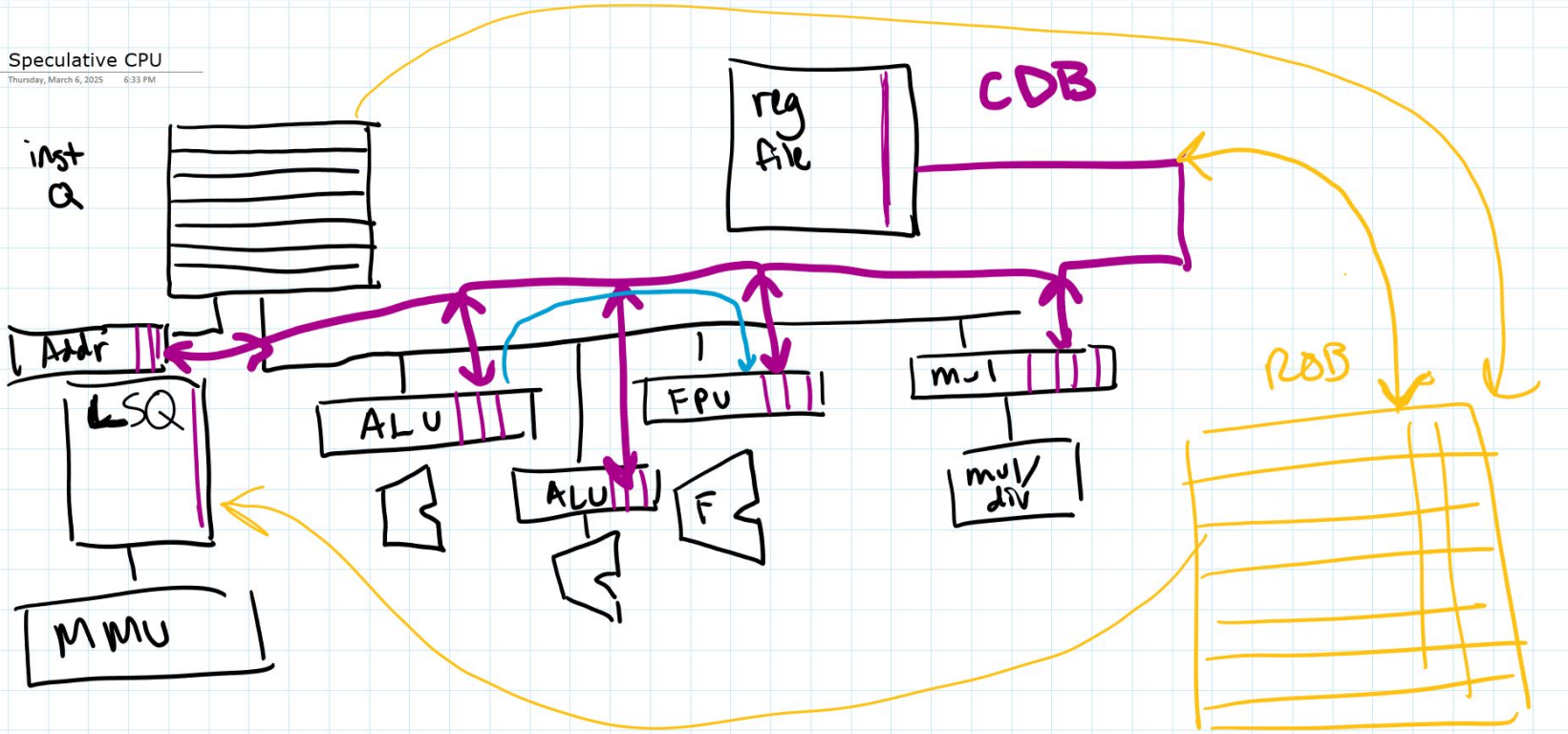
The speculative algorithm



The speculative CPU with ROB

Speculative CPU

Thursday, March 6, 2025 6:33 PM



Four stages of execution

Issue

If reservation station and ROB available, issue to both; update control entries

Execute

If operands available, execute instr; otherwise wait (for stores: this stage only computes effective address)

Write

When result is available, send on CDB (update ROB, reservation stations)

Commit

If normal commit or store: update register/memory and remove instr from ROB

If incorrectly predicted branch: flush ROB and reservation stations, fetch correct instr

Example 4: speculative branch

Speculative example

Monday, March 10, 2025 9:39 AM

Mul: 3 cycles
ALU: 1 cycle

Entry	Ready?	State	Dest	Val
mul t2, t1, t0	1 ✓ ₍₅₎	I(1) E(2-4) W(5) C(6)	t2	✓
mul t3, t2, t2	2 ✓ ₍₂₎	I(2) E(6-8) W(9) C(10)	t3	✓
bne t1, t2, END	3 ✓ ₍₇₎	I(3) E(6) W(7) C(11)		taken
addi t3, t1, 8	4 ✓ ₍₆₎	I(4) E(5) W(6)	t3	✓
add t4, t2, t3	5 ✓ ₍₂₎	I(7) E(8) W(9)	t4	✓

Station	Busy	Op	Source 1 (V _r /Q _r)	Source 2 (V _r /Q _r)	Dest
Mul1					
Mul2					
ALU1					
ALU2	✓	✓	✓	✓	✓

*There would also be an A column for load/store units; we're leaving them off for space

t0	t1	t2	t3	t4	t5	t6	s0
			4	5			



What other hazard can result from the store operation?

Status	Wait until	Action or bookkeeping
Issue all instructions	Reservation station (r) and ROB (b) both available	<pre> if (RegisterStat[rs].Busy) /*in-flight instr. writes rs*/ {h ← RegisterStat[rs].Reorder; if (ROB[h].Ready) /* Instr completed already */ {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;} else {RS[r].Qj ← h;} /* wait for instruction */ } else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;}; RS[r].Busy ← yes; RS[r].Dest ← b; ROB[b].Instruction ← opcode; ROB[b].Dest ← rd;ROB[b].Ready ← no; </pre>
FP operations and stores		<pre> if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/ {h ← RegisterStat[rt].Reorder; if (ROB[h].Ready) /* Instr completed already */ {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;} else {RS[r].Qk ← h;} /* wait for instruction */ } else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;}; </pre>
FP operations		RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd;
Loads		RS[r].A ← imm; RegisterStat[rt].Reorder ← b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;
Stores		RS[r].A ← imm;
Execute FP op	(RS[r].Qj == 0) and (RS[r].Qk == 0)	Compute results—operands are in Vj and Vk
Load step 1	(RS[r].Qj == 0) and there are no stores earlier in the queue	RS[r].A ← RS[r].Vj + RS[r].A;
Load step 2	Load step 1 done and all stores earlier in ROB have different address	Read from Mem[RS[r].A]
Store	(RS[r].Qj == 0) and store at queue head	ROB[h].Address ← RS[r].Vj + RS[r].A;
Write result all but store	Execution done at r and CDB available	<pre> b ← RS[r].Dest; RS[r].Busy ← no; ∀x(if (RS[x].Qj==b) {RS[x].Vj ← result; RS[x].Qj ← 0}); ∀x(if (RS[x].Qk==b) {RS[x].Vk ← result; RS[x].Qk ← 0}); ROB[b].Value ← result; ROB[b].Ready ← yes; </pre>
Store	Execution done at r and (RS[r].Qk == 0)	ROB[h].Value ← RS[r].Vk;
Commit	Instruction is at the head of the ROB (entry h) and ROB[h].ready == yes	<pre> d ← ROB[h].Dest; /* register dest, if exists */ if (ROB[h].Instruction==Branch) {if (branch is mispredicted) {clear ROB[h], RegisterStat; fetch branch dest;};} else if (ROB[h].Instruction==Store) {Mem[ROB[h].Destination] ← ROB[h].Value;} else /* put the result in the register destination */ {Regs[d] ← ROB[h].Value;}; ROB[h].Busy ← no; /* free up ROB entry */ /* free up dest register if no one else writing it */ if (RegisterStat[d].Reorder==h) {RegisterStat[d].Busy ← no;}; </pre>

Figure 3.14 Steps in the algorithm and what is required for each step. For the issuing instruction, rd is the destination, rs and rt are the sources, r is the reservation station allocated, b is the assigned ROB entry, and h is the head entry of the ROB. RS is the reservation station data structure. The value returned by a reservation station is called the result. RegisterStat is the register data structure, Regs represents the actual registers, and ROB is the reorder buffer data structure.



Can we do better than “predict branch not taken?”

Backwards and forwards branches

```
while(guard) {  
    ...  
}
```

```
loop: bne s0 s1 end  
...  
j loop  
end: ...
```

Backwards branches are *very frequently* taken (~90% by some counts)

Forwards branches are a coin flip without any other information

```
j start  
loop: ...  
...  
start: beq s0 s1 loop
```