The speculative algorithm

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 (f**or stores: this stage only computes effective address**)

Write

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

Commit (**for whatever instruction is at head of ROB**)

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

	Entry	Ready?	State	Dest	Val
`mul t2, t1, t0`					
`mul t3, t2, t2`					
`bne t1, t2, END`					
`addi t3, t1, 8`					
`add t4, t2, t3`					

Station	Busy	Op	Source 1 (V_j/Q_j)		Source 2 (V_k/Q_k)		Dest
Mul1							
Mul2							
ALU1							
ALU2							

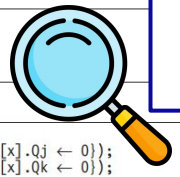
*There would also be an A column for load/store units; we're leaving them off for space

t0	**t1**	**t2**	**t3**	**t4**	**t5**	**t6**	**s0**

When should we execute loads?

Status	Wait until	Action or bookkeeping
Issue all instructions	Reservation station (r) and ROB (b) both available	`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;`
FP operations and stores		`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;};`
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	`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;`
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	`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;};`

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.



Load step 2 | Load step 1 done and all stores earlier in ROB have different address

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

Backwards and forwards branches

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

Backwards branches are *very frequently* taken (~90% by some counts), mostly because this is how loops are compiled

Forwards branches are a coin flip without any other information

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

branch **and** jump every iteration (yuck!)

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

preferable compilation

Software should be optimized such that the sequential code path is the most common path, with less-frequently taken code paths placed out of line. Software should also assume that backward branches will be predicted taken and forward branches as not taken, at least the first time they are encountered. Dynamic predictors should quickly learn any predictable branch behavior.

What’s the problem with assuming backwards branch taken? (Hint: think about why 5s executed as if branch wouldn’t be taken by default)

Pre-decode

For **PC-relative instructions** (RISCV: `bne`, `jal`): can compute branch/jump **target address** to fetch target instruction in next cycle

Also does things like detect instruction sizes (basically *bare-minimum decode* to inform fetch unit)

Why doesn't this work for register-relative instructions (`jalr`)?

BTB

Branch Target Buffer

A fast cache for branch/jump target addresses (indexed by current PC)

Pre-decode reads from the BTB and gives address to fetch unit

If BTB is wrong: flush the ROB and refetch

When do we update the BTB?

Unlike some other architectures, the RISC-V jump (JAL with *rd*=x0) instruction should always be used for unconditional branches instead of a conditional branch instruction with an always-true condition. RISC-V jumps are also PC-relative and support a much wider offset range than branches, and will not pressure conditional branch prediction tables.