Speculative execution

From ISA lecture: micro-ops

Translation of complex machine instruction (macro-op) to multiple steps

For example, addq 8(%rdi) %rax might be translated into:

- add 8 to rdi
- load that address from memory
- add that value to rax
- store the result in rax

Each of these can be issued separately into a different functional unit!

x86 processors have had a uop decoder since the Pentium Pro (1997)

Even RISC processors decode into uops – driven by design of FUs

Functional units

SW example

worksheet

Control hazards

```
for (int i = 0; i < 100; i++) {
    A[i] = A[i] + B[i];
}
```

We reduced CPI by about 6% by reordering… but the real culprit keeping CPI > 1 is the branch and jump!

We can use hardware to compute jump addresses earlier (P&H 4.8) … but there will still be at least one cycle wasted

```
addi t0, x0, 0   # t0/i = 0
addi t1, x0, 100 # t1 = 100
loop: bge t0, t1, end # loop while i < 100
slli t2, t0, 2   # t2 = t0/i * 4
add t3, a0, t2   # t3 = A + t2 (A + i * 4)
add t4, a1, t2   # t4 = B + t2 (B + i * 4)
lw t2, 0(t4)     # t2 = B[i]
lw t4, 0(t3)     # t4 = A[i]
addi t0, t0, 1   # t0/i++
add t4, t4, t2   # t4 = A[i] + B[i]
sw t4, 0(t3)     # A[i] = A[i] + B[i]
j loop
end: nop
```

Control dependences

```
if x:
    P1 // depends on x, not y
if y:
    P2 // depends on y, not x
```

When dealing with control dependences:

1) instruction dependent on branch should not be moved before branch
2) instruction not dependent on branch should not be moved after branch

But this is pretty restrictive… instead, we may allow for instructions to be executed (or partially executed) as long as we can preserve the correctness of the program somehow

Branch delay slot

Some older architectures execute one instruction immediately after a branch/jump instruction (regardless if the branch is taken)

Up to compiler and/or CPU to move an independent instruction into that slot

Along w/ hardware, this helps us basically hide the cost of unconditional jumps

What about branches?

```
addi t0, x0, 0
addi t1, x0, 100
loop: bge t0, t1, end
nop
slli t2, t0, 2
add t3, a0, t2
add t4, a1, t2
lw t2, 0(t4)
lw t4, 0(t3)
add t4, t4, t2
addi t0, t0, 1
j loop
sw t4, 0(t3)
end: nop
```

5s assumption about branches

`addi t0, t0, 1`	IF	ID	EX	Mem	WB									
`addi t2, t2, -1`		IF	ID	EX	M	W								
`bne, t2, x0, l2`			IF	ID	EX	M	W							
`slli t0, t0, 1`				IF	ID	EX	🟧	🟧						
`addi t1, t1, -1`					IF	ID	🟧	🟧	🟧					
`bne t1, x0, x11`						IF	🟧	🟧	🟧	🟧				
`addi t0, t0, 1`							IF	ID	EX	M	W			
`addi t2, t2, -1`								IF	ID	EX	M	W		
`bne t2, x0, l2`									IF	ID	EX	M	W	
`slli t0, t0, 1`										IF	ID	🟧	🟧	🟧

Waste 3 cycles on 2 of every 3 iterations (6 cycles/full loop)

PC dest. when branch taken

5s assumption about branches

addi t1, t1, -1	IF	ID	🟧	🟧	🟧									
bne t1, x0, x11		IF	🟧	🟧	🟧	🟧								
addi t0, t0, 1			IF	ID	EX	M	W							
addi t2, t2, -1				IF	ID	EX	M	W						
bne t2, x0, l2					IF	ID	EX	M	W					
slli t0, t0, 1						IF	ID	EX	M	W				
addi t1, t1, -1							IF	ID	EX	M	W			
bne t1, x0, x11								IF	ID	EX	M	W		

not taken

Speculative execution

Assume branch behavior (e.g. not taken), don’t commit instructions until outcome is known, squash in-flight instructions if needed

What needs to change about Tomasulo’s to support speculative execution?

- Remember: effect of execution became permanent (written to reg file) on W stage of instruction

Enabling speculation with the ROB

Tomasulo's allows instructions to execute and **be committed out of order**

- Problem: doesn't work very well for stores
- Problem: doesn't work very well for control hazards

What if instructions could execute out of order but had to **commit in-order**?

- Re-order buffer (ROB) helps us do this
- On issue, create a ROB entry
- Only commit (write result of) instruction when it's next in the ROB
- Requires stores to happen in-order (**why?**)
- Allows us to execute before we know the result of branch

The speculative CPU with ROB

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)

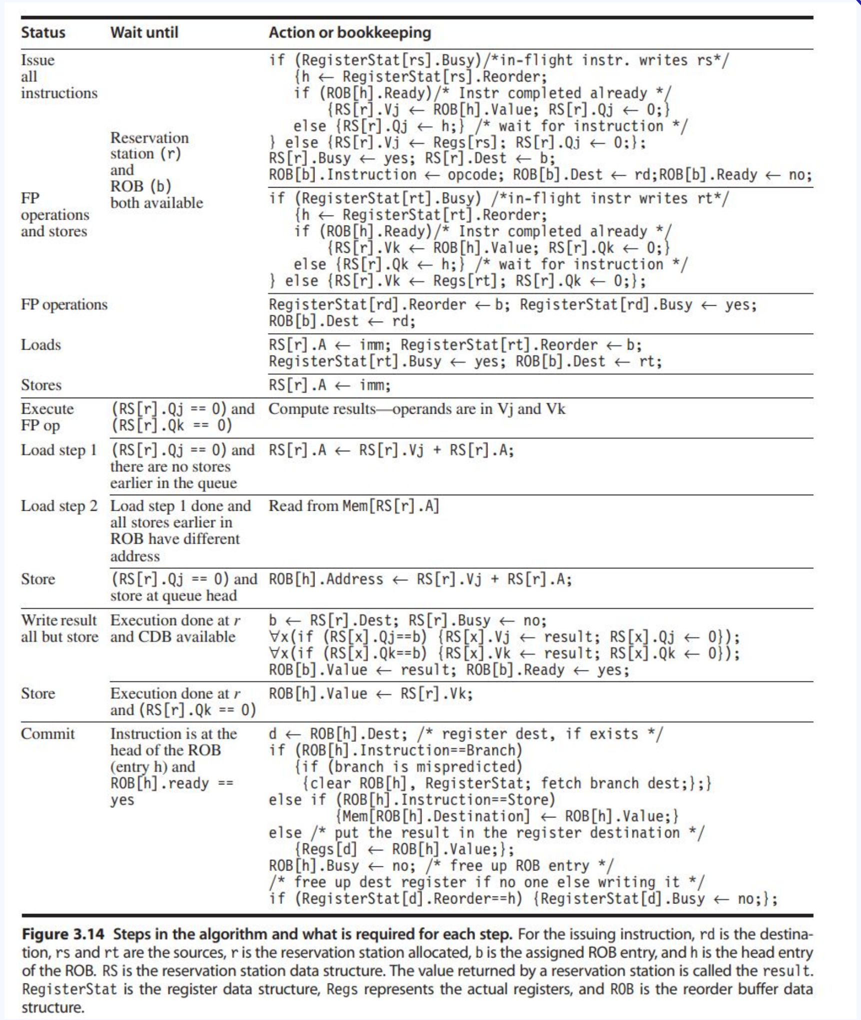
Commit

- 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

? ? ?

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	`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.