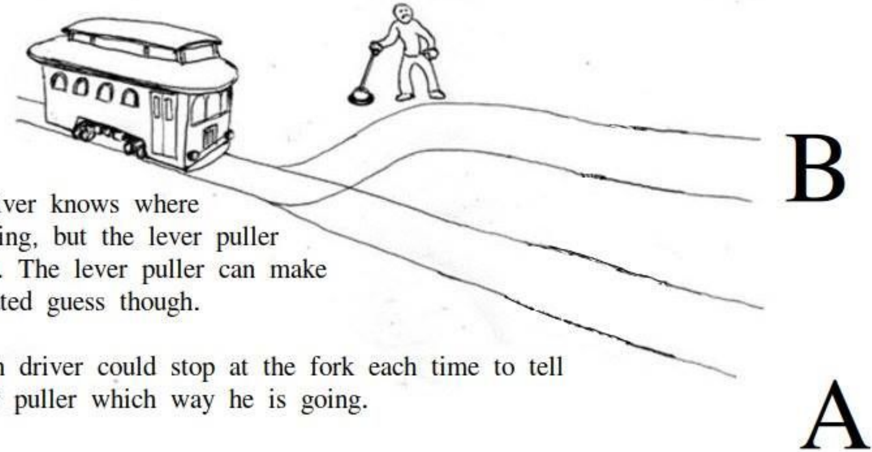
Out of order execution; advanced branch prediction



5s pipeline when assume not taken

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 pipeline when assume not taken

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

1-bit BPB entry

Works great for outer loop (one misprediction and then 400 correct predictions)

Works less great for inner loop:

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

Branch prediction

Wednesday, March 6, 2024 9:42 AM

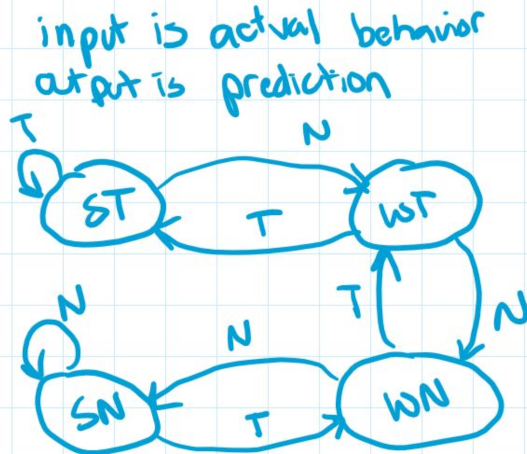
```
addi t0, x0, 0      // t0 = 0
addi t1, x0, 400    // t1 = 400
l1: addi t2, x0, 3  // t2 = 3
l2: addi t0, t0, 1  // t0++
addi t2, t2, -1     // t2--
bne t2, x0, l2      // loop while t2 > 0
slli t0, t0, 1      // t0 <<= 1
addi t1, t1, -1     // t1--
bne t1, x0, l1      // loop while t1 > 0
```

time encountering branch	t2	BPB entry	branch taken?	co… predic…
0	-	0	-	-
1	2	0	y	X
2	1	1	y	✓
3	0	1	n	X
4	2	0	y	X
5	1	1	y	✓
6	0	1	n	X
	⋮			

2-bit BPB entry

2 bits can keep track of 4 states: strong taken, weak taken, weak not taken, strong not taken

Keeps some of history (means branch prediction needs to be wrong twice instead of once before changing) – works better for the inner loop!



time encountering branch	t2	BPB entry	branch taken?	correct prediction
0	–	WN	–	
		WN	y	X
		WT	y	✓
		ST	n	X
		WT	y	✓
		ST	y	✓
		ST	n	X

⋮

Assumes instruction can be fetched in the next cycle (fast target calculation)

Waste 3 cycles on every 3rd iteration (3 cycles/full loop)

Branch target buffers

Cache for computing branch target address (new PC)

Potentially faster to fetch next instruction

- Common in modern systems; unlike branch delay slots

Multiple ways to set this up (see Agner document)

- Multiple levels
- Different behavior for different types of branches/jumps

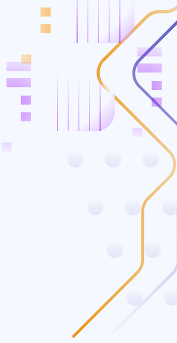
Correlated branches

```
if (x == 2) // branch A
    x = 0;
if (y == 2) // branch B
    y = 0;
if (x != y) // branch C
    …
```

A taken and B taken implies C not taken!
→ branch outcomes are often **not independent** of each other

How would we design a branch predictor that can handle correlated branches?

Correlating predictors

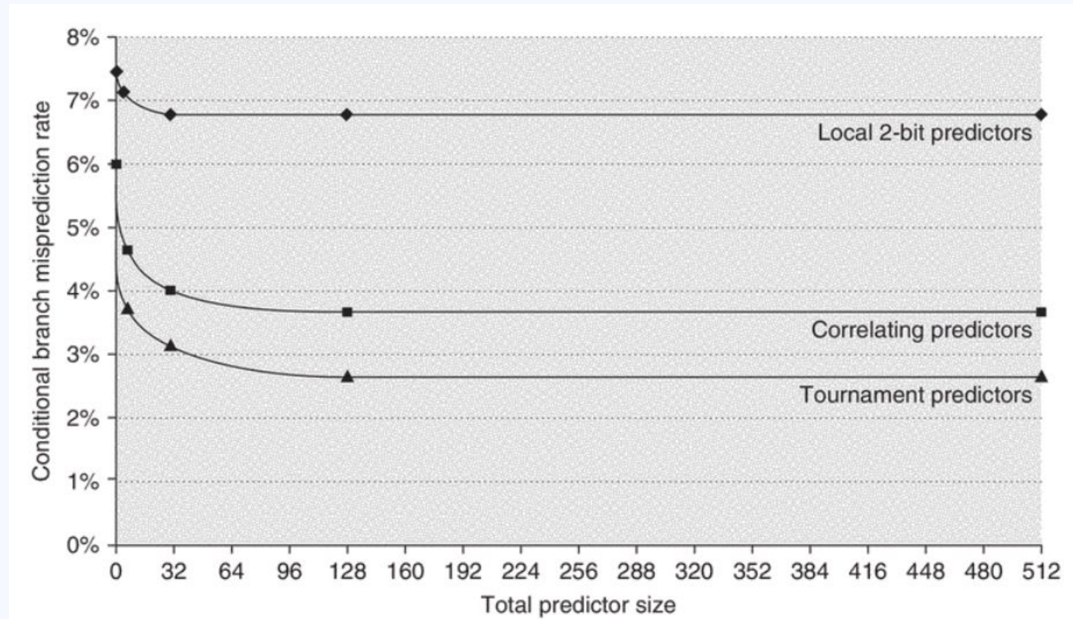


Tournament branch predictors

Run multiple branch predictors at once

Keep track of which one is doing better (using eg 2-bit predictor!) and use that one

H&P fig. 3.4



Modern branch prediction

Agner document

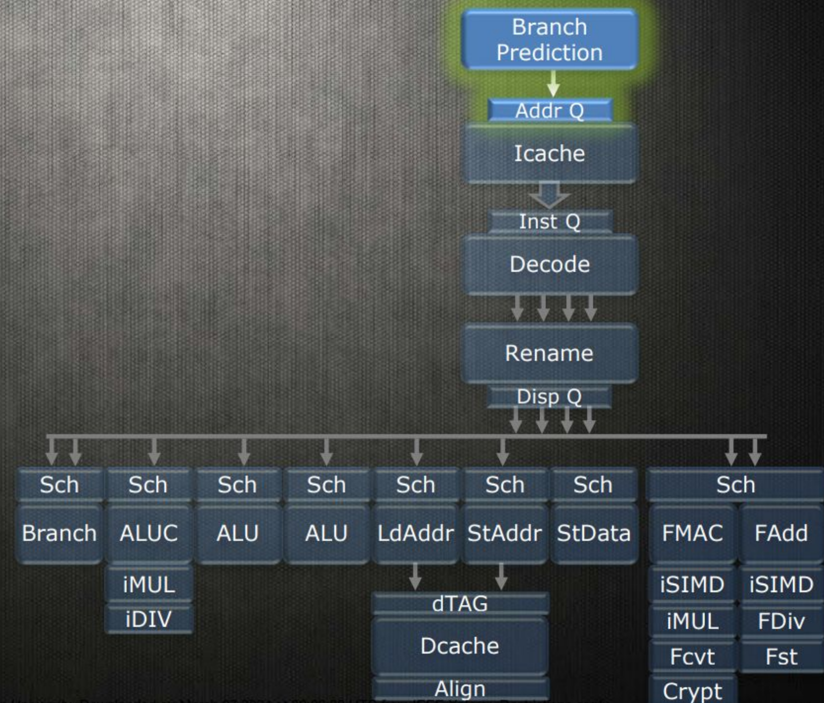
H&P talking about i7:

- small first-level predictor to handle cost of predicting at every cycle
- larger second-level predictor “as a backup”
- combines two-bit, global, and loop exit predictors in tournament

The ~future~

B. Burgess, "Samsung exynos M1 processor," 2016 IEEE Hot Chips 28 Symposium (HCS), Cupertino, CA, USA, 2016, pp. 1-18, doi: 10.1109/HOTCHIPS.2016.7936205.

link



Dynamic scheduling (OOO execution)

Allows executions to be rearranged at runtime (also called Out of Order, or OOO)

Advantages:

- Pipeline-agnostic (code written/compiled for one uarch can work efficiently on OOO uarch)
- Allows handling of dependences that can't be resolved by compiler
- Allows code to execute during delay (e.g. cache miss, div, floating point)

Much more complex! But worth it in modern systems

Example

```
div t0, t1, t2
add t3, t0, t4
sw t3, 0(s0)
sub t4, t5, t6
mul t3, t5, t4
```

In-order pipeline will stall to allow div instruction to finish

Can we do better? What are the dependences?

Details of OOO execution

In-order issue

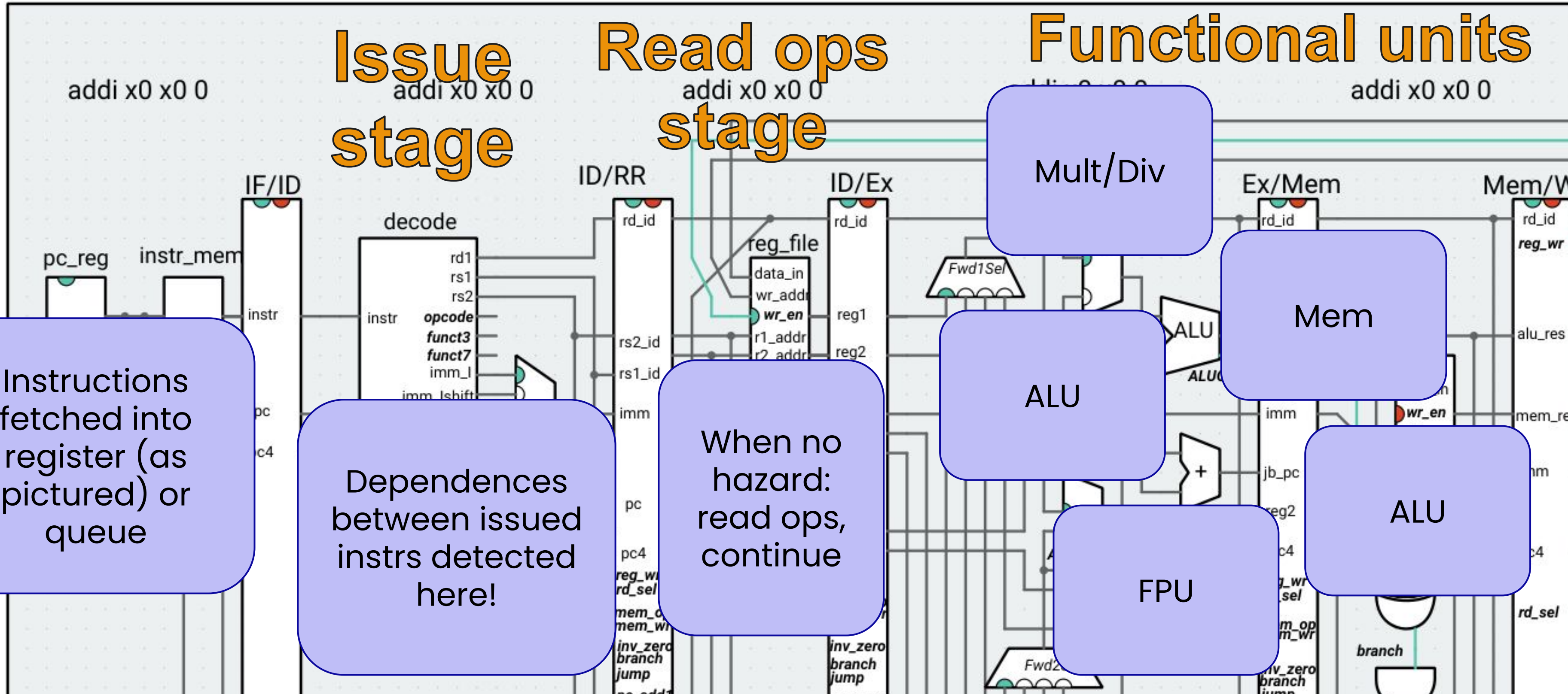
Potentially out-of-order completion

Between the time when an instruction is issued and when it completes, it is *in execution (in flight)*

- Multiple instructions can be in flight at the same time, either due to multiple functional units (ALUs, FPUs, etc) or due to pipelining

How? Split up ID stage

(*we'll redraw this picture next week)



Early OOO: scoreboarding

Fascinating history of CDC 6600

Keep track of dependences between in-flight instructions and fetched instructions using dependency matrices

Issue a fetched instruction only when no dependences arise

But this is really restrictive: we can do better

By Jitze Couperus - Flickr: Supercomputer - The Beginnings, CC BY 2.0, link

Tomasulo's algorithm intuition

Developed by Robert Tomasulo for IBM 360/91

Minimize RAW hazards by tracking data dependences and reordering

Minimize WAR and WAW hazards by *register renaming*

Limited to code within basic blocks (we'll come back to branching soon)

```
div t0, t1, t2
add t3, t0, t4
sw t3, 0(s0)
sub t4, t5, t6
mul t3, t5, t4
```

What if the hardware had two temporary registers, X and Y?

```
div t0, t1, t2
add X, t0, t4
sw X, 0(s0)
sub Y, t5, t6
mul t3, t5, Y
```

```
div t0, t1, t2
sub Y, t5, t6
mul t3, t5, Y
add X, t0, t4
sw X, 0(s0)
```

now the div might have enough time to write to t0 before add needs it!