Superscalar* Club Meeting #1

*we really mean: superscalar speculative out-of-order

James C. Hoe
Department of ECE
Carnegie Mellon University

Goal

- Achieve an RTL-precise understanding of superscalar speculative out-of-order register dataflow—as with 5-stage pipeline in 18-447
  - know you could make a working design and know what it does (how good is it?)
  - know shortcomings and limits in the simplifications to help get started
  - know what you have not figured out yet

Plan of Attack

- Focus
  - mainly on register dataflow
  - lightly on memory dataflow
  - not at all on i-fetch (a well decoupled subject both conceptually and physically)
- Path
  - further develop concepts in L20 we didn't have time for
  - study Metaflow DRIS to flesh out conceptual-level understanding
  - study how things were really done in R10K
  - play with an RTL-precise executable model (in C++)

Let's Get Started

Parallelism Defined

- T_1 (work measured in time):
  - time to do work with 1 PE
- T_∞ (critical path):
  - time to do work with infinite PEs
  - T_∞ bounded by dataflow dependence
- Average parallelism:

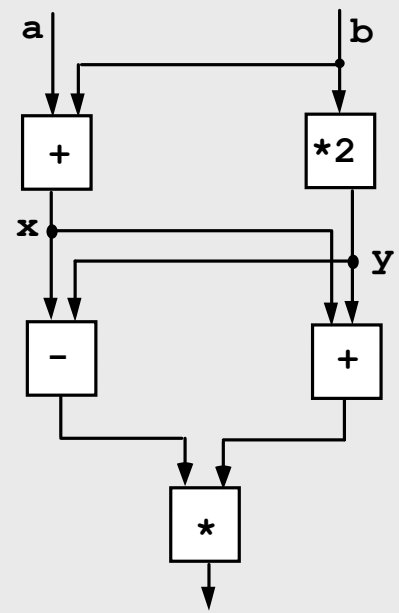
  let's call **p** *concurrency*

  $$P_{avg} = T_1 / T_\infty$$
- For a system with **p** PEs

  $$T_p \geq \max\{ T_1/p, T_\infty \}$$
- When $P_{avg} >> p$

  $T_p \approx T_1/p$, aka "linear speedup"

```
x = a + b;
y = b * 2
z =(x-y) * (x+y)
```



ILP: Instruction-Level Parallelism

- Average **ILP** = T_1 / T_∞
  = no. instruction / no. cyc required

code1: **ILP** = 1

i.e., must execute serially

code2: **ILP** = 3

i.e., can execute at the same time

```
code1:   r1 ← r2 + 1
         r3 ← r1 / 17
         r4 ← r0 - r3
```

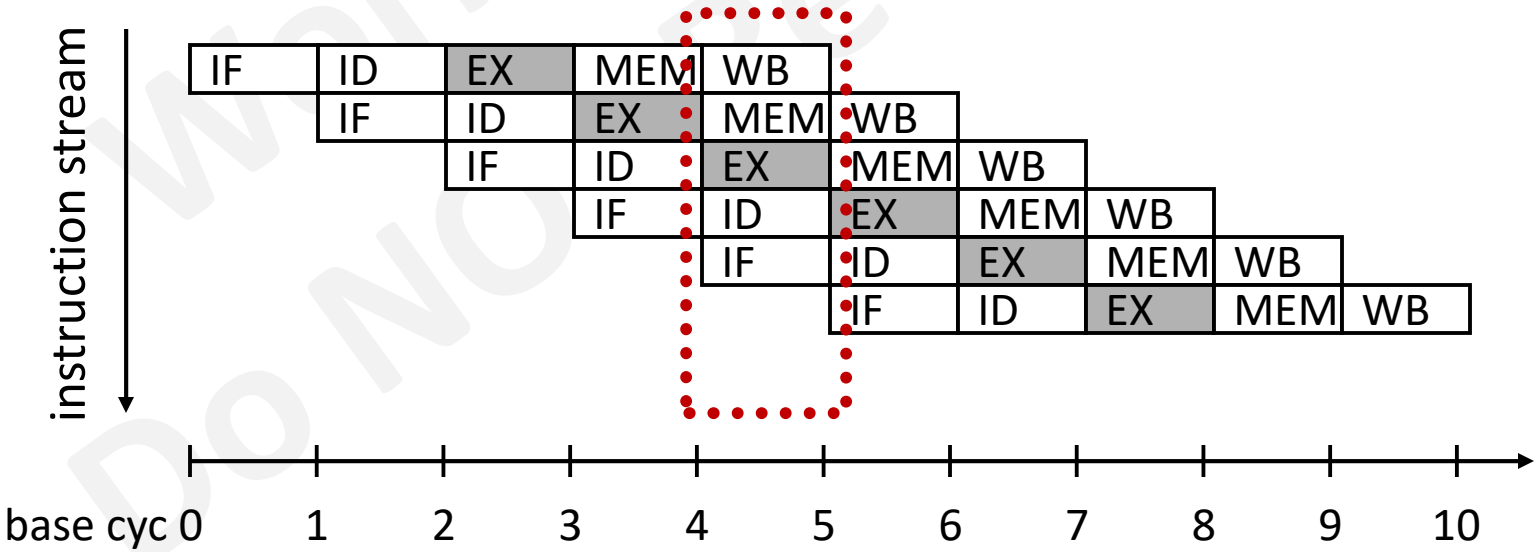
```
code2:   r1 ← r2 + 1
         r3 ← r9 / 17
         r4 ← r0 - r10
```

Superscalar Speculative Out-of-Order Execution

Exploiting ILP for Performance

Scalar in-order pipeline with forwarding

- operation latency (**OL**)= **1** base cycle
- peak **IPC** = **1** *// no concurrency*
- required **ILP** ≥ **1** to avoid stall

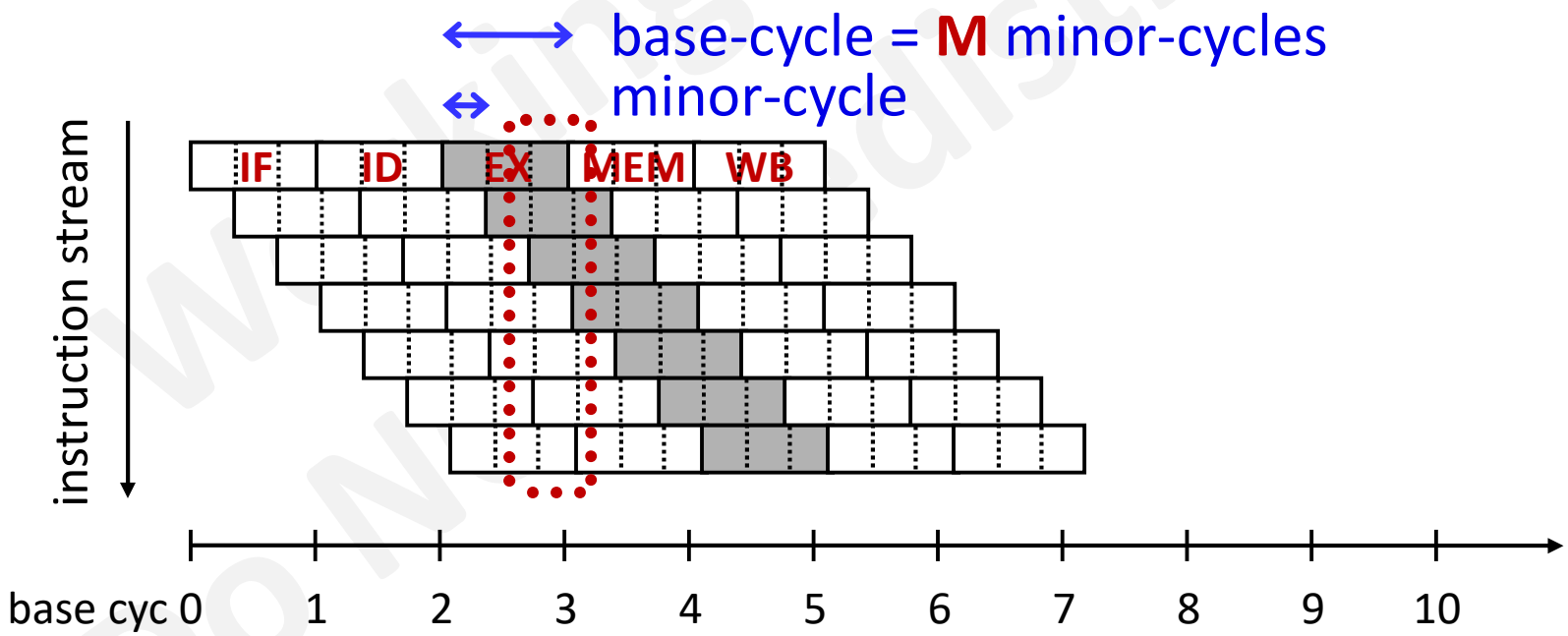


Superpipelined Execution

OL = **M** minor-cycle; same as **1** base cycle

peak **IPC** = **1** per minor-cycle *// has concurrency though*

required **ILP** ≥ **M**



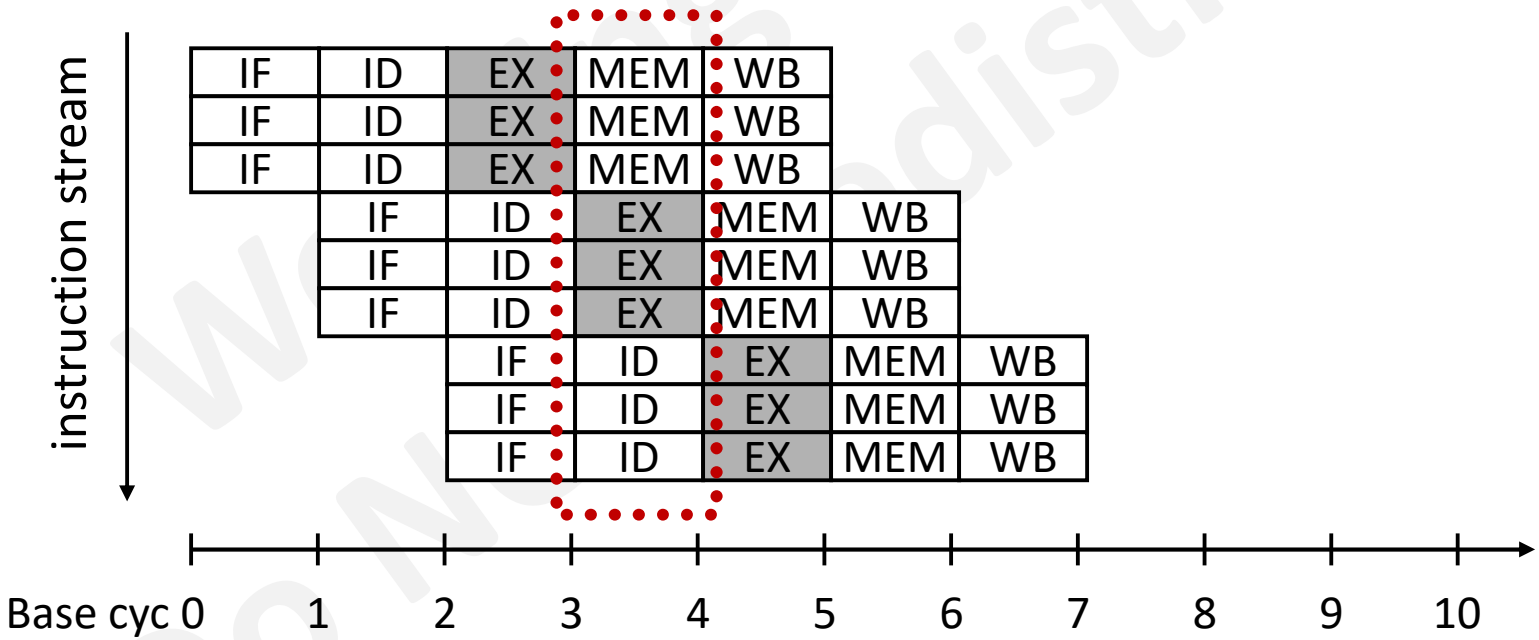
Achieving full performance requires always finding **M** "independent" instructions in a row

Superscalar (Inorder) Execution

OL = **1** base cycle

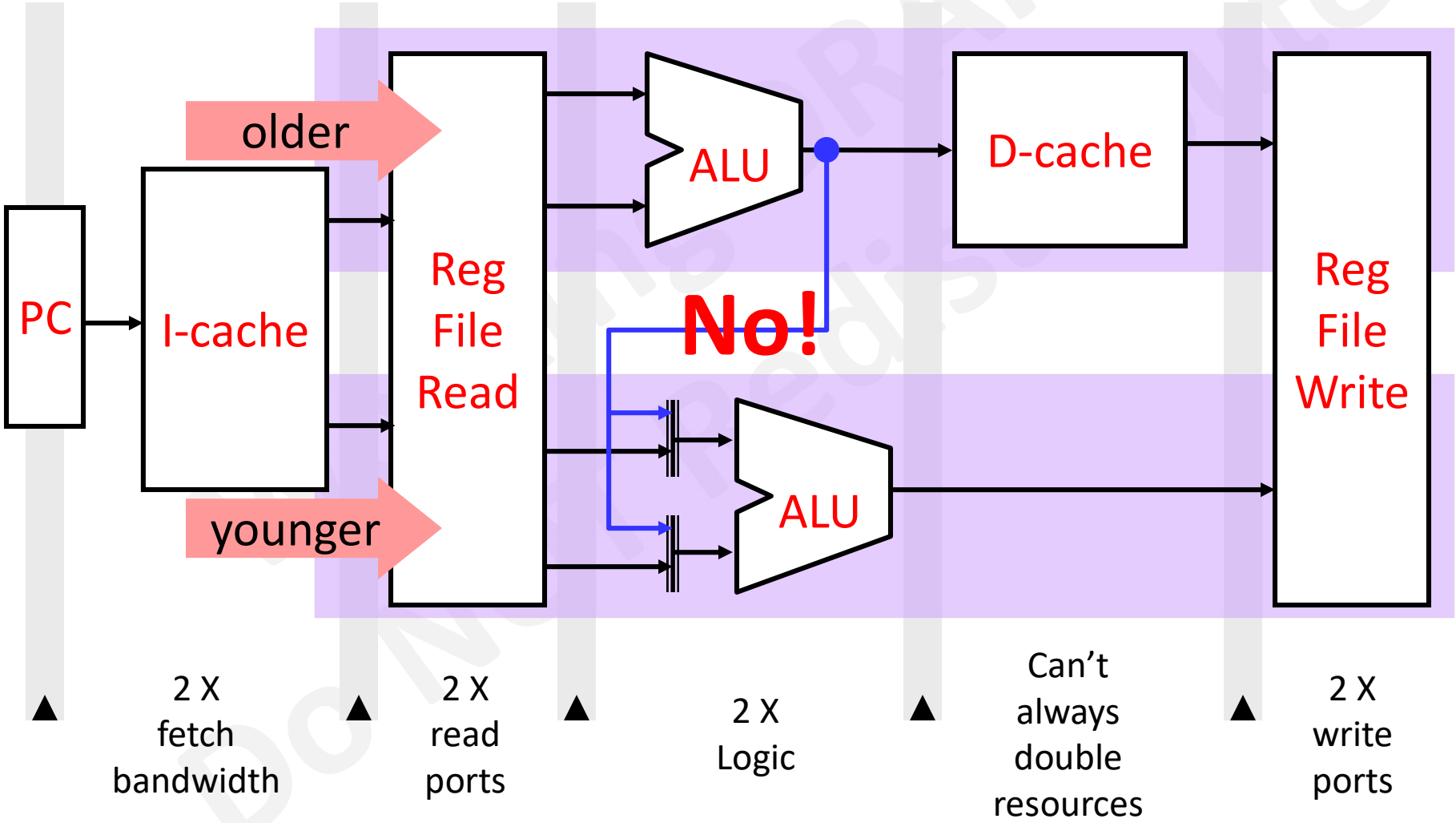
peak **IPC** = **N**

required **ILP** ≥ **N**



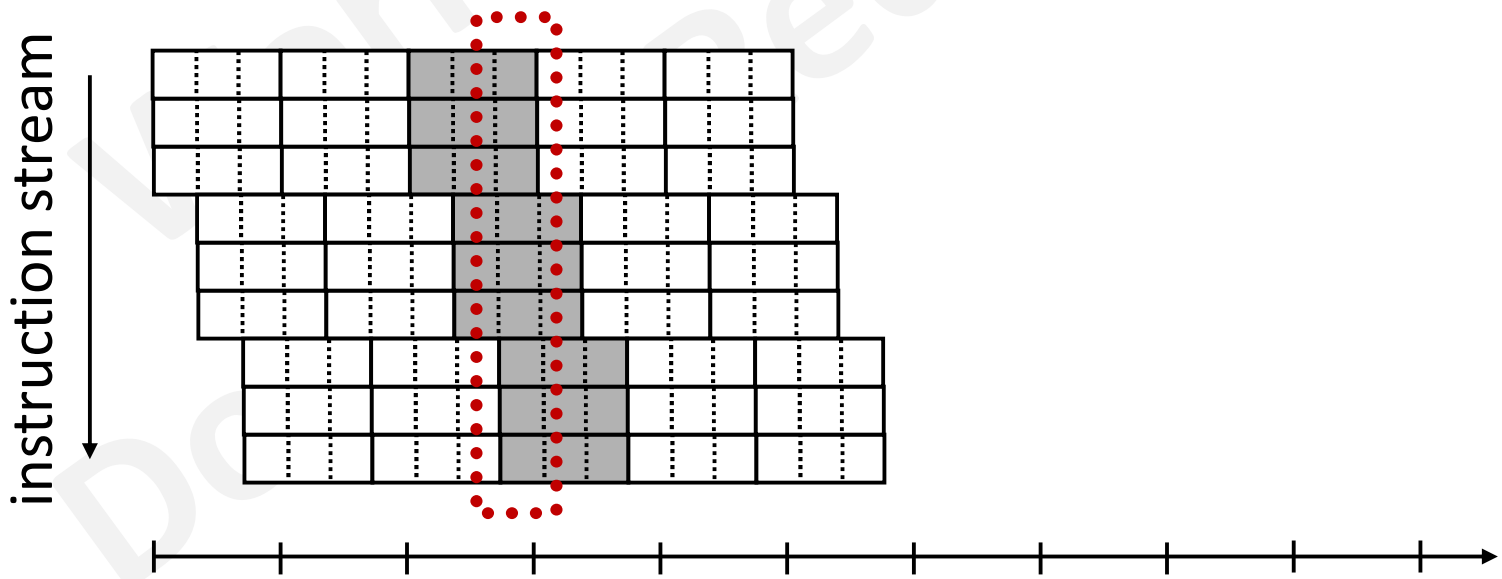
Achieving full performance requires finding **N** "independent" instructions on every cycle

In-order Superscalar



Limitations of Inorder Pipeline

- Achieved **IPC** of inorder pipelines degrades rapidly as **N**x**M** approaches **ILP**
- Despite high concurrency potential, pipeline never full due to frequent dependency stalls!!



Why Out-of-Order Execution?

- **ILP** is scope dependent

ILP=1

$$\left.\begin{array}{l} r1 \leftarrow r2 + 1 \\ r3 \leftarrow r1 / 17 \\ r4 \leftarrow r0 - r3 \end{array}\right\}$$

$$\left.\begin{array}{l} r1 \leftarrow r2 + 1 \\ r3 \leftarrow r1 / 17 \\ r4 \leftarrow r0 - r3 \\ r11 \leftarrow r12 + 1 \\ r13 \leftarrow r19 / 17 \\ r14 \leftarrow r0 - r20 \end{array}\right\}$$

ILP=2

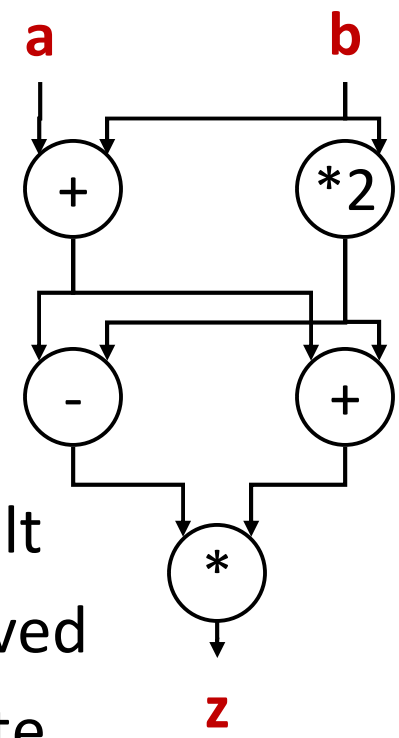
Accessing **ILP**=2 requires not only (1) larger scheduling window <u>but also</u> (2) out-of-order execution

Superscalar Speculative Out-of-Order Execution

von Neuman vs Dataflow

- Consider a von Neumann program
  - What is the significance of the program order?
  - What is the significance of the storage locations?

```
v := a + b;
w := b * 2;
x := v - w;
y := v + w;
z := x * y;
```



- Dataflow program instruction ordering implied by data dependence
  - instruction specifies who receives the result
  - instruction executes when operands received
  - no program counter, no* intermediate state

[dataflow figure and example from Arvind]

MIPS Pipeline: In-order, Consumer-Pull Dependence Resolution

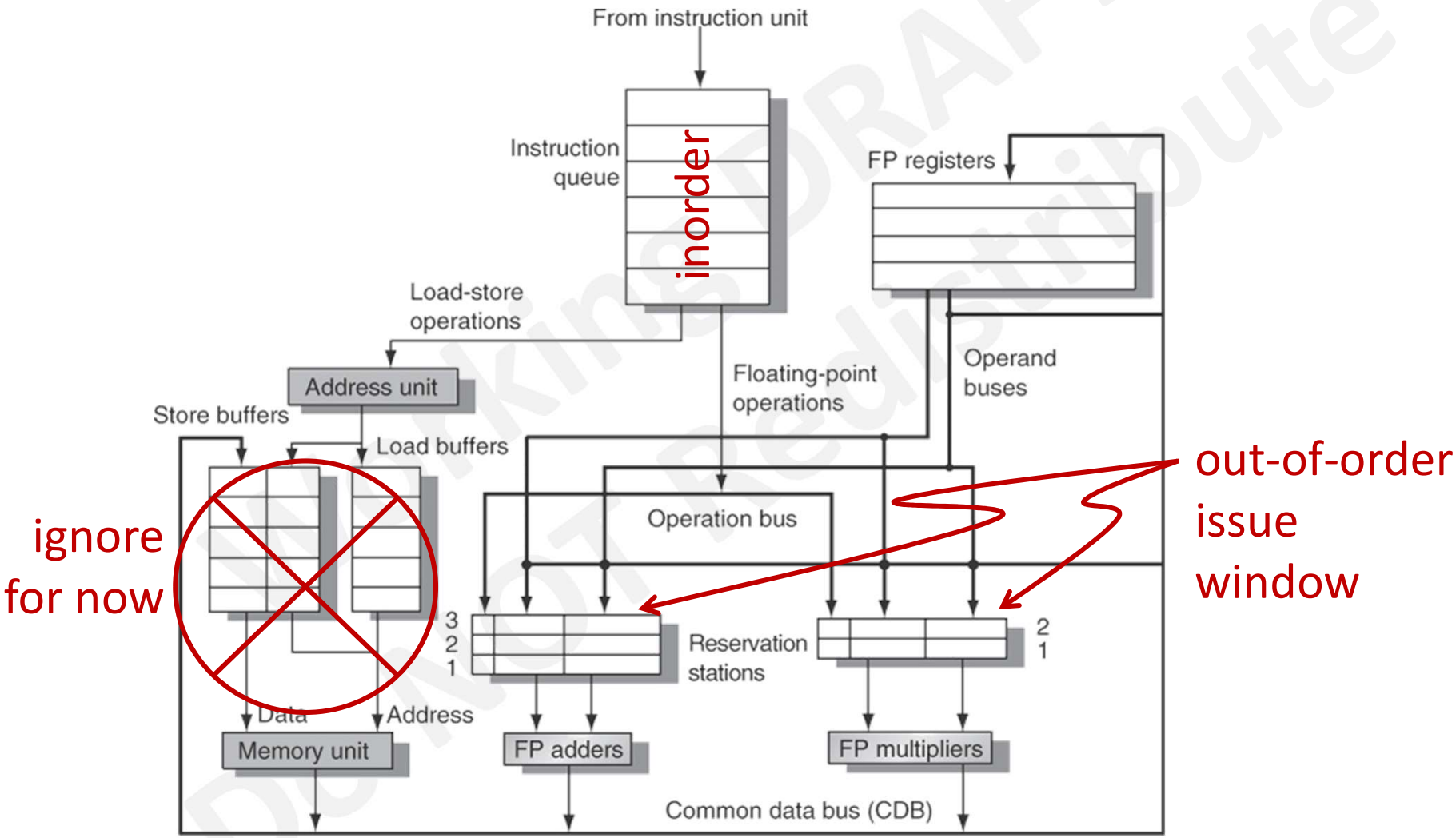
A hazard exits

- Older I_A and younger I_B have RAW hazard iff
  - I_B (R/I, LW, SW, Bxx or JALR) reads a register written by I_A (R/I, LW, or JAL/R)
  - dist(I_A, I_B) $\leq$ dist(ID, WB) = 3
- To detect hazard in time to prevent, before I_B in ID reads a register, I_B needs to check if any I_A in EX, MEM or WB is going to update it
- Before: I_B need to stall for I_A to <u>update RF</u>
- Now: I_B need to stall for I_A to <u>produce result</u>
  - retrieve I_A result from datapath when ready
  - must retrieve from **youngest** if multiple hazards

Instruction Micro-Dataflow

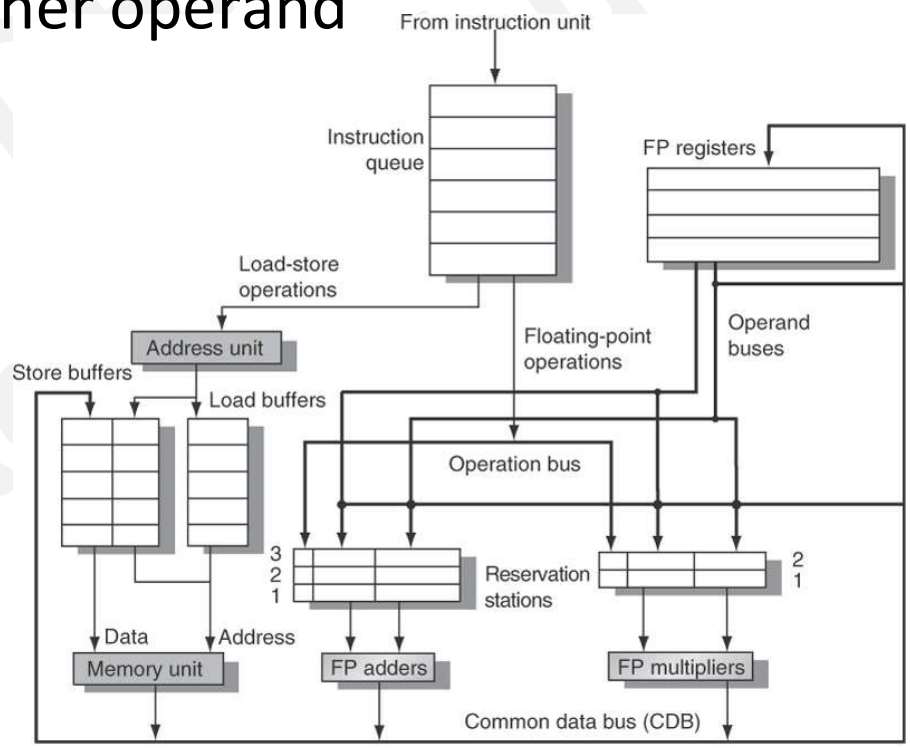
- Maintain a buffer of many pending instructions, a.k.a. reservation stations (**RS**s)
  - wait for functional unit to be free
  - wait for required input operands to be available
- Decouple execution order from who is first in line (program order)
  - select inst's in **RS** whose operands are available
  - give preference to older instructions (heuristical)
- A completing instruction (producer) signals dependent instructions (consumer) of operand availability *(producer-push resolution)*

IBM 360/91 FP Module [1967]



Tomasulo's Algorithm [IBM 360/91, 1967]

- Dispatch an instruction to a **RS** slot after decode
  - decode received from RF either operand value or placeholder **RS-tag**
  - mark RF dest with **RS-tag** of current inst's **RS** slot
- Inst in **RS** can issue when all operand values ready
- Completing instruction, in addition to updating RF dest, broadcast its **RS-tag** and value to all **RS** slots
- **RS** slot holding matching **RS-tag** placeholder pickup value



Tomasulo's Algorithm

Instruction state	Wait until	Action or bookkeeping

Issue
FP operation
	Station r empty	if (RegisterStat[rs].Qi≠0)
		{RS[r].Qj ← RegisterStat[rs].Qi}
		else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};
		if (RegisterStat[rt].Qi≠0)
		{RS[r].Qk ← RegisterStat[rt].Qi
		else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};
		RS[r].Busy ← yes; RegisterStat[rd].Q ← r;

Load or store	Buffer r empty	if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes;
Load only		RegisterStat[rt].Qi ← r;
Store only		if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rs].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};
Execute FP operation	(RS[r].Qj = 0) and (RS[r].Qk = 0)	Compute result: operands are in Vj and Vk
Load-store step 1	RS[r].Qj = 0 & r is head of load-store queue	RS[r].A ← RS[r].Vj + RS[r].A;
Load step 2	Load step 1 complete	Read from Mem[RS[r].A]
Write Result FP operation or load	Execution complete at r & CDB available	∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0}); RS[r].Busy ← no;
Store	Execution complete at r & RS[r].Qk = 0	Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no;

ignore for now

FuBusy?

ignore for now

why for all x?

ignore for now

from [Hennessy&Patterson, CAAQA]

RS entry
- **Busy**: in use
- **Op**: opcode
- **Vj**: op1 value
- **Vk**: op2 value
- **Qj**: op1 RS-tag
- **Qk**: op2 RS-tag

case (RegisterStat)
0: RF val current
RS-tag:
to be produce by corresponding instruction

Tomasulo's Algorithm (caption)

Figure 2.12 Steps in the algorithm and what is required for each step. For the issuing instruction, rd is the destination, rs and rt are the source register numbers, imm is the sign-extended immediate field, and r is the reservation station or buffer that the instruction is assigned to. RS is the reservation station data structure. The value returned by an FP unit or by the load unit is called result. RegisterStat is the register status data structure (not the register file, which is Regs[]). When an instruction is issued, the destination register has its Qi field set to the number of the buffer or reservation station to which the instruction is issued. If the operands are available in the registers, they are stored in the V fields. Otherwise, the Q fields are set to indicate the reservation station that will produce the values needed as source operands. The instruction waits at the reservation station until both its operands are available, indicated by zero in the Q fields. The Q fields are set to zero either when this instruction is issued, or when an instruction on which this instruction depends completes and does its write back. When an instruction has finished execution and the CDB is available, it can do its write back. All the buffers, registers, and reservation stations whose value of Qj or Qk is the same as the completing reservation station update their values from the CDB and mark the Q fields to indicate that values have been received. Thus, the CDB can broadcast its result to many destinations in a single clock cycle, and if the waiting instructions have their operands, they can all begin execution on the next clock cycle. Loads go through two steps in Execute, and stores perform slightly differently during Write Result, where they may have to wait for the value to store. Remember that to preserve exception behavior, instructions should not be allowed to execute if a branch that is earlier in program order has not yet completed. Because any concept of program order is not maintained after the Issue stage, this restriction is usually implemented by preventing any instruction from leaving the Issue step, if there is a pending branch already in the pipeline. In Section 2.6, we will see how speculation support removes this restriction.

from [Hennessy&Patterson, CAAQA]

RAW Example:

i: R2 ← R0 + R4

j: R8 ← R0 + R2

cyc1:

RS	Qj	Vj	Qk	Vk
1				
2				
3				

Add

RS	Qj	Vj	Qk	Vk
4				
5				

Mult

RF	Q	value
0	0	6.0
2	0	3.5
4	0	10.0
8	0	7.8

cyc2:

RS	Qj	Vj	Qk	Vk
1				
2				
3				

Add

RS	Qj	Vj	Qk	Vk
4				
5				

Mult

RF	Q	value
0		
2		
4		
8		

cyc3:

RS	Qj	Vj	Qk	Vk
1				
2				
3				

Add

RS	Qj	Vj	Qk	Vk
4				
5				

Mult

RF	Q	value
0		
2		
4		
8		

RAW Example:

assume 1-cyc add, 2-cyc mult
issue up to 1 add & 1 mult per cycle
complete up to 1 add & 1 mult per cycle

cyc1:

RS	Qj	Vj	Qk	Vk
i 1	0	6.0	0	10.0
2				
3				

Add issue 1

RS	Qj	Vj	Qk	Vk
4				
5				

Mult

RF	Q	value
0	0	6.0
2	1	--
4	0	10.0
8	0	7.8

cyc2:

RS	Qj	Vj	Qk	Vk
i 1	0	6.0	0	10.0
j 2	0	6.0	1	--
3				

Add bcast <1, 16.0>

RS	Qj	Vj	Qk	Vk
4				
5				

Mult

RF	Q	value
0	0	6.0
2	1	--
4	0	10.0
8	2	--

cyc3:

RS	Qj	Vj	Qk	Vk
1				
j 2	0	6.0	0	16.0
3				

Add issue 2

RS	Qj	Vj	Qk	Vk
4				
5				

Mult

RF	Q	value
0	0	6.0
2	0	16.0
4	0	10.0
8	2	--

Invariants

RegisterStat[**rs**].Qi≠0 ⇒

let **q**=RegisterStat[**rs**].Qi in {

RS[**q**].Busy=true ∧

rs is logical dest of inst in RS[**q**] }

?RegisterStat[**rs**].Qi=0 ⇒∀ **q** (**rs** is not dest of inst in RS[**q**])?

RS[**r**].Busy ∧ RS[**r**].Qj≠0 ⇒

let { **q**=RS[**r**].Qj; **rs**=operand of inst in RS[**r**] }

in { RS[**q**].Busy ∧

rs is logical dest of inst in RS[**q**] ∧

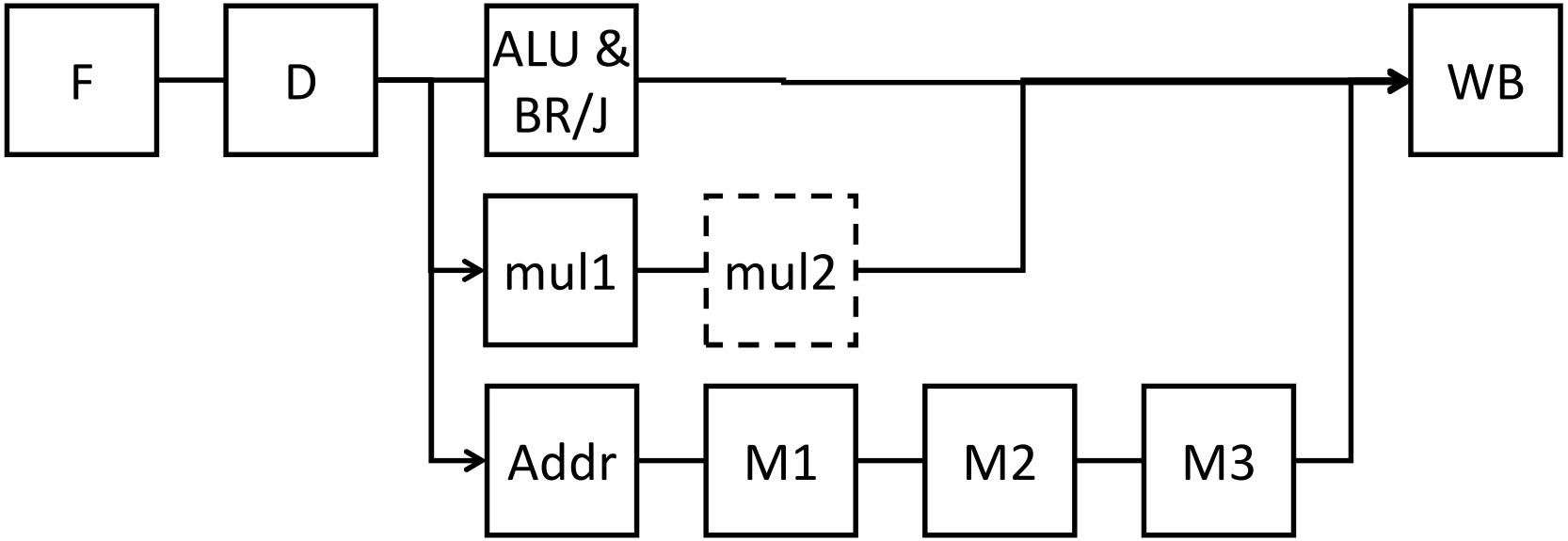
inst in RS[**q**] is older than inst in RS[**r**] ∧

no RS[**s**] hold inst older than RS[**q**] younger than RS[**r**] and has **rs** as dest

}

WAW and WAR in Tomasulo??

- No WAW and WAR in 5-stage pipeline because
  - single write stage
  - write stage at the end *(later than any read stage)*
  - in-order progression in pipeline



WAR Example:

i: R4 ← R0 x R8
j: R0 ← R4 x R2
k: R2 ← R2 + R8

cyc1:

RS	Qj	Vj	Qk	Vk
1				
2				
3				

Add

RS	Qj	Vj	Qk	Vk
4				
5				

Mult

RF	Q	value
0	0	6.0
2	0	3.5
4	0	10.0
8	0	7.8

cyc2:

RS	Qj	Vj	Qk	Vk
1				
2				
3				

Add

RS	Qj	Vj	Qk	Vk
4				
5				

Mult

RF	Q	value
0		
2		
4		
8		

cyc3:

RS	Qj	Vj	Qk	Vk
1				
2				
3				

Add

RS	Qj	Vj	Qk	Vk
4				
5				

Mult

RF	Q	value
0		
2		
4		
8		

WAR Example:

i: R4 ← R0 x R8
j: R0 ← R4 x R2
k: R2 ← R2 + R8

cyc1:

RS	Qj	Vj	Qk	Vk
1				
2				
3				

Add

	RS	Qj	Vj	Qk	Vk
i	4	0	6.0	0	7.8
	5				

Mult issue 4

RF	Q	value
0	0	6.0
2	0	3.5
4	4	--
8	0	7.8

cyc2:

	RS	Qj	Vj	Qk	Vk
k	1	0	3.5	0	7.8
	2				
	3				

Add issue 1

	RS	Qj	Vj	Qk	Vk
i	4	0	6.0	0	7.8
j	5	4	--	0	3.5

Mult

RF	Q	value
0	5	--
2	1	--
4	4	--
8	0	7.8

cyc3: *j's use "**R2**" disassociated from k's update **R2** in RF*

	RS	Qj	Vj	Qk	Vk
k	1	0	3.5	0	7.8
	2				
	3				

Add bcast <1,11.3>

	RS	Qj	Vj	Qk	Vk
i	4	0	6.0	0	7.8
j	5	4	--	0	3.5

Mult bcast <4, 46.8>

RF	Q	value
0	5	--
2	1	--
4	4	--
8	0	7.8

WAW Example:

i: R4 ← R0 x R8
j: R2 ← R0 + R4
k: R4 ← R0 + R8
l: R8 ← R4 x R8

cyc1:

RS	Qj	Vj	Qk	Vk
1				
2				
3				

Add

RS	Qj	Vj	Qk	Vk
4				
5				

Mult

RF	Q	value
0	0	6.0
2	0	3.5
4	0	10.0
8	0	7.8

cyc2:

RS	Qj	Vj	Qk	Vk
1				
2				
3				

Add

RS	Qj	Vj	Qk	Vk
4				
5				

Mult

RF	Q	value
0		
2		
4		
8		

cyc3:

RS	Qj	Vj	Qk	Vk
1				
2				
3				

Add

RS	Qj	Vj	Qk	Vk
4				
5				

Mult

RF	Q	value
0		
2		
4		
8		

WAW Example:

i: R4 ← R0 x R8
j: R2 ← R0 + R4
k: R4 ← R0 + R8
l: R8 ← R4 x R8

cyc1:

	RS	Qj	Vj	Qk	Vk
j	1	0	6.0	4	--
	2				
	3				

Add

	RS	Qj	Vj	Qk	Vk
i	4	0	6.0	0	7.8
	5				

Mult issue 4

RF	Q	value
0	0	6.0
2	1	--
4	4	--
8	0	7.8

cyc2:

	RS	Qj	Vj	Qk	Vk
j	1	0	6.0	4	--
k	2	0	6.0	0	7.8
	3				

Add issue 2

	RS	Qj	Vj	Qk	Vk
i	4	0	6.0	0	7.8
l	5	2	--	0	7.8

Mult

RF	Q	value
0	0	6.0
2	1	--
4	2	--
8	5	--

cyc3: *the architecturally younger re-define of "**R4**" wins*

	RS	Qj	Vj	Qk	Vk
j	1	0	6.0	4	--
k	2	0	6.0	0	7.8
	3				

Add bcast <2,13.8>

	RS	Qj	Vj	Qk	Vk
i	4	0	6.0	0	7.8
l	5	2	--	0	7.8

Mult bcast <4, 46.8> ·····→ ?

RF	Q	value
0	0	6.0
2	1	--
4	2	--
8	5	--

Tomasulo Recap: Out-of-order yes, but . . .

- IPC ≤ 1
  - superscalar execute but actually about hiding multiply latency
  - IPC<<1 when ILP=1: successive RAW-dependent instructions cannot issue back-to-back
- A lot of duplicated operand values stored
- No precise RF state on exception
  - on WAR example: if *J* raises exception after *K* already wrote back to RF; can't undo *K*'s RF update
- No way to rewind if speculative execution

Need still more powerful magic!!

Superscalar Speculative Out-of-Order Execution

(The hard part of speculation is in rewinding!)

Pipeline Flush for Exceptions

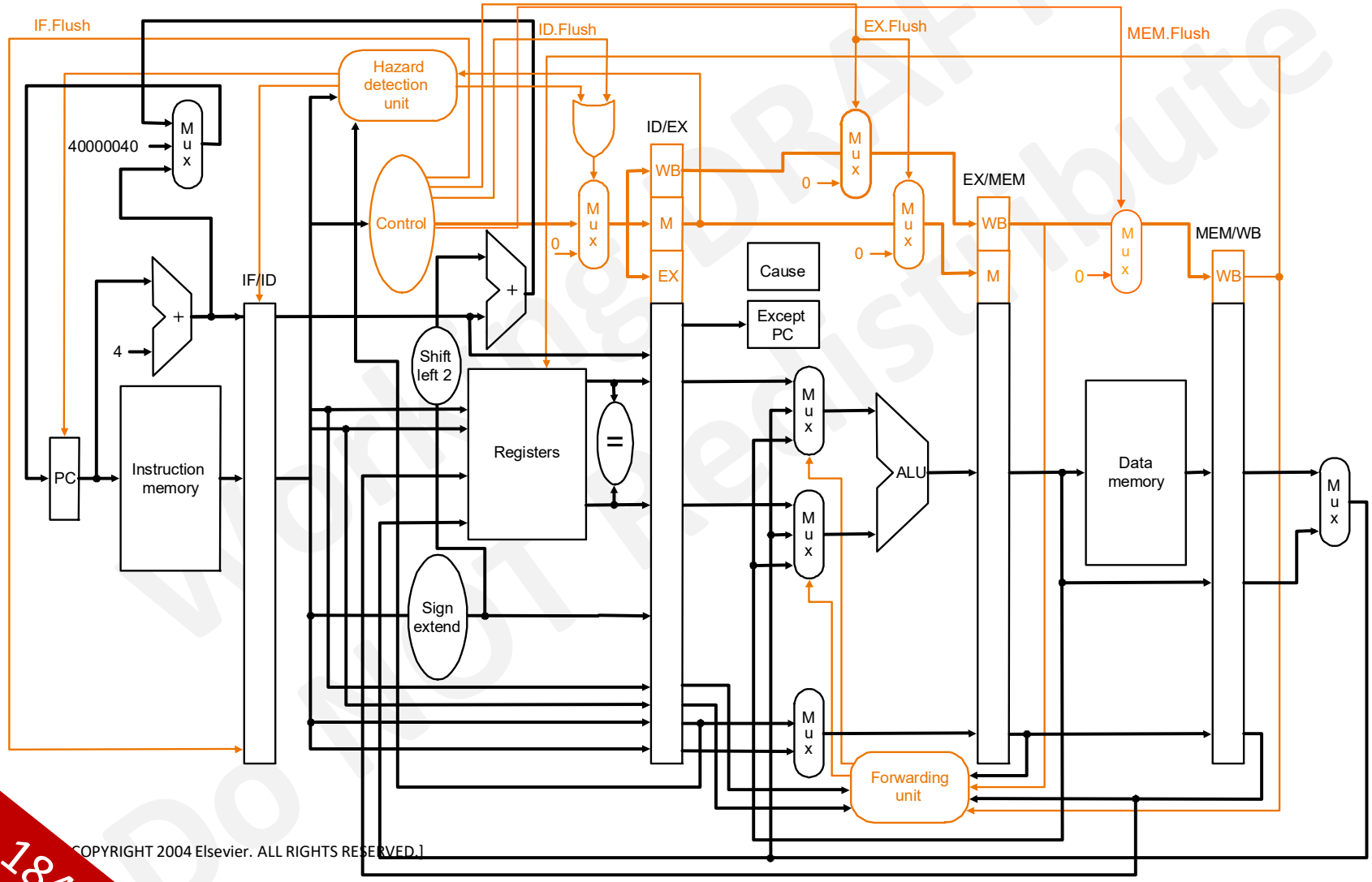


Figure fr COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

from 18447

Where is "the current instruction"?

Program State Views

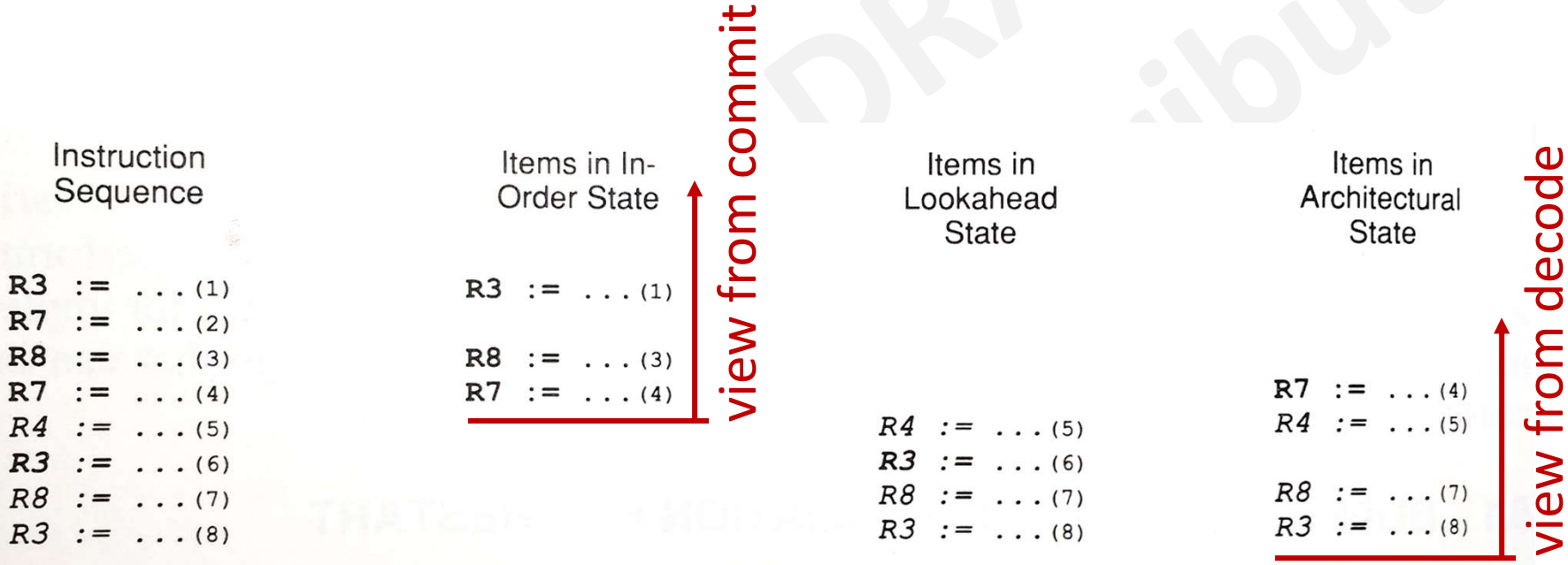


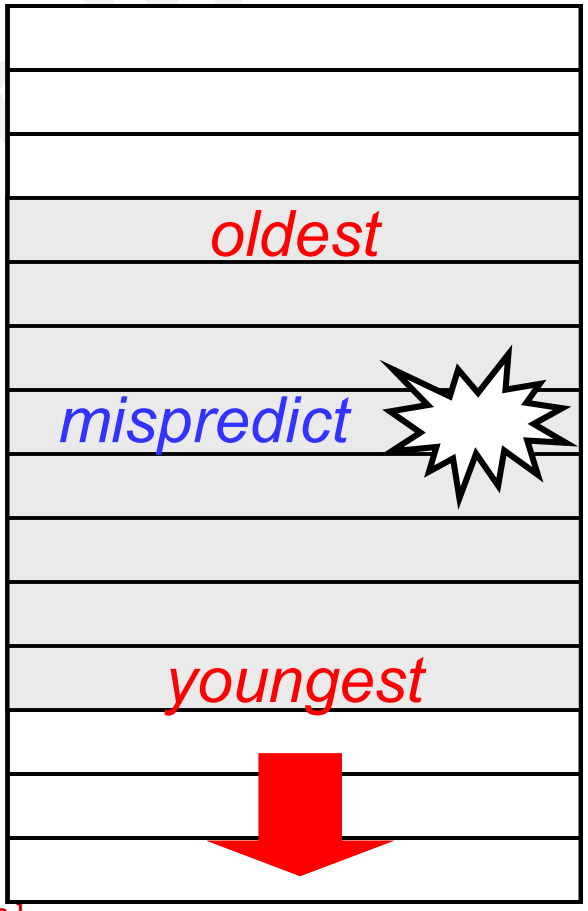
Figure 5-1. Illustration of In-Order, Lookahead, and Architectural States

from [Johnson, 1990]

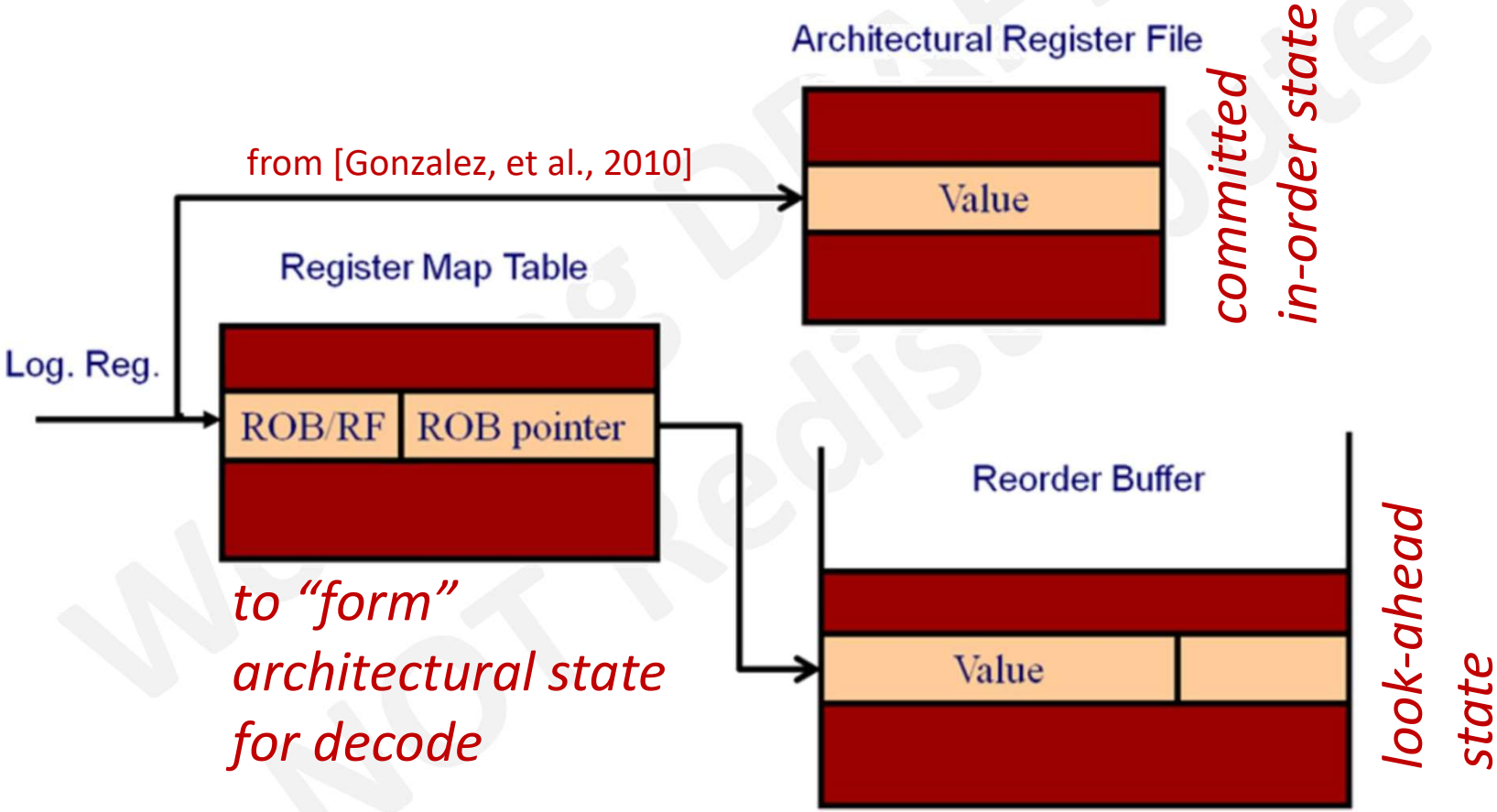
Lookahead instructions and their effects need to be reversible!

Instruction Reorder Buffer (ROB)

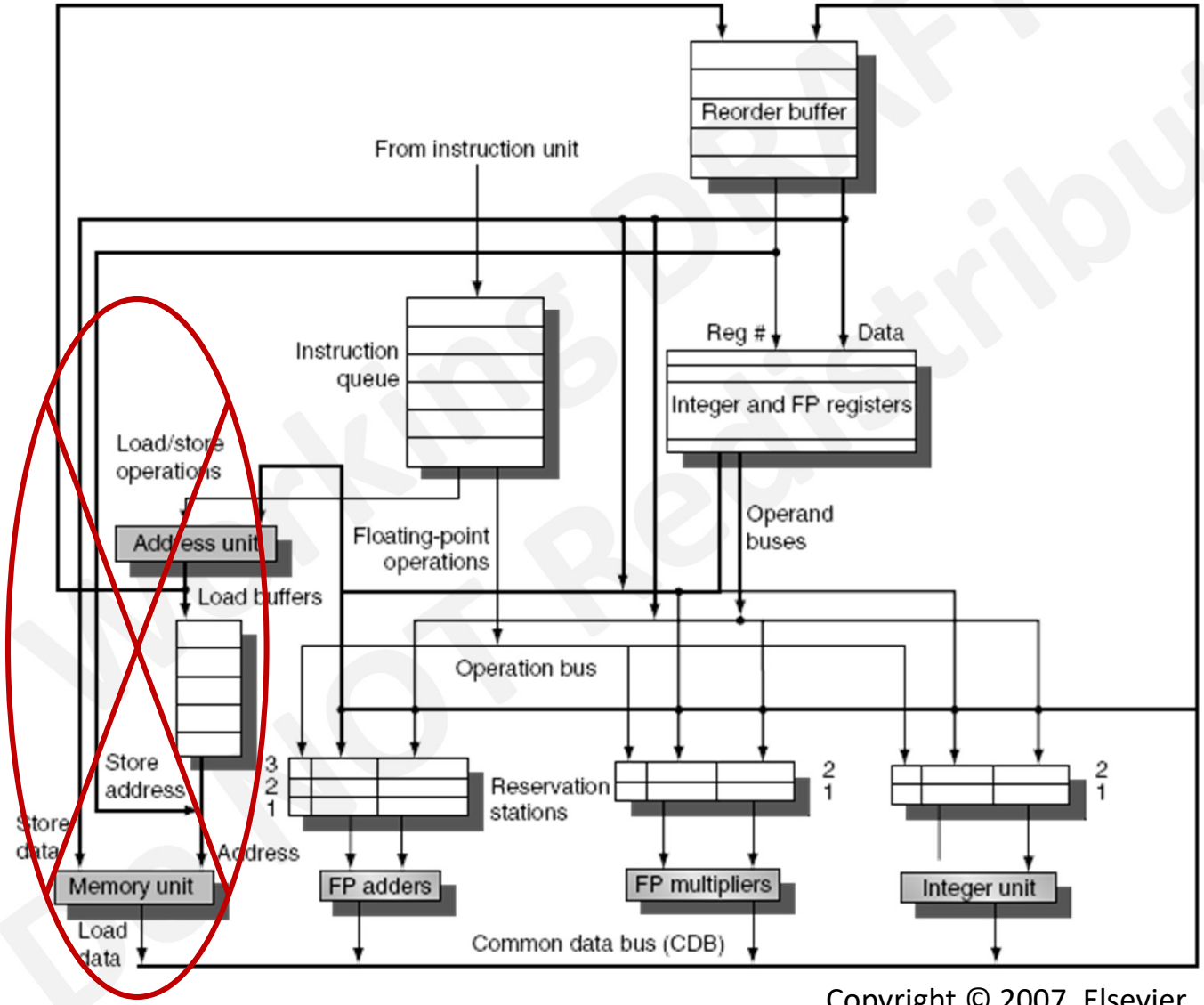
- Program-order bookkeeping (circular buffer)
  - instructions enter and leave in program order
  - tracks 10s to 100s of in-flight instructions in different stages of execution
- Dynamic juggling of state and dependency
  - oldest finished instruction "commit" architectural state updates on exit
  - all ROB entries considered "speculative" due to potential for exceptions and mispredictions



ROB Rename Registers



Tomasulo + Speculative Execution



ignore for now

Tomasulo's Algorithm + ROB

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;`

Ignore for now

from [Hennessy&Patterson, CAAQA]

ROB Entry:

is **Busy** / **Instruction** / logical **Dest** reg / dest **Value** / value is **Ready**

RegisterStat: reg is **Busy** / renamed to **Reorder** buffer entry #

Tomasulo's Algorithm + ROB

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;};

FuBusy?

Ignore for now

Ignore for now

can only fix misprediction when oldest

from [Hennessy&Patterson, CAAQA]

Invariants

- You give it try . . .

RAW Example:

i: R2 ← R0 + R4

j: R8 ← R0 + R2

cyc1:

RS	dst	Qj	Vj	Qk	Vk
1					
2					
3					

Add

RS	dst	Qj	Vj	Qk	Vk
4					
5					

Mult

RF	bsy	rob	value
0	0		6.0
2	0		3.5
4	0		10.0
8	0		7.8

ROB	dst	rdy	value
i			
ii			
iii			
iv			

cyc2:

RS	dst	Qj	Vj	Qk	Vk
1					
2					
3					

Add

RS	dst	Qj	Vj	Qk	Vk
4					
5					

Mult

RF	bsy	rob	value
0			
2			
4			
8			

ROB	dst	rdy	value
i			
ii			
iii			
iv			

cyc3:

RS	dst	Qj	Vj	Qk	Vk
1					
2					
3					

Add

RS	dst	Qj	Vj	Qk	Vk
4					
5					

Mult

RF	bsy	rob	value
0			
2			
4			
8			

ROB	dst	rdy	value
i			
ii			
iii			
iv			

assume 1-cyc add, 2-cyc mult
issue up to 1 add & 1 mult per cycle
complete up to 1 add & 1 mult per cycle

i: R2 ← R0 + R4
j: R8 ← R0 + R2

cyc1:

RS	dst	Qj	Vj	Qk	Vk
1	i	0	6.0	0	10.0
2					
3					

Add issue 1

RS	dst	Qj	Vj	Qk	Vk
4					
5					

Mult

RF	bsy	rob	value
0	0		6.0
2	1	i	3.5
4	0		10.0
8	0		7.8

ROB	dst	rdy	value
i	R2	0	-
ii			
iii			
iv			

cyc2:

RS	dst	Qj	Vj	Qk	Vk
1	i	0	6.0	0	10.0
2	ii	0	6.0	i	-
3					

Add cmpl <i, 16.0>

RS	dst	Qj	Vj	Qk	Vk
4					
5					

Mult

RF	bsy	rob	value
0	0		6.0
2	1	i	3.5
4	0		10.0
8	1	ii	7.8

ROB	dst	rdy	value
i	R2	0	-
ii	r8	0	-
iii			
iv			

cyc3:

RS	dst	Qj	Vj	Qk	Vk
1					
2	ii	0	6.0	0	16.0
3					

Add issue 2

RS	dst	Qj	Vj	Qk	Vk
4					
5					

Mult

RF	bsy	rob	value
0	0		6.0
2	1	i	3.5
4	0		10.0
8	1	ii	7.8

ROB	dst	rdy	value
i	R2	1	16.0
ii	R8	0	-
iii			
iv			

to retire

WAR Example:

i: R4 ← R0 x R8
j: R0 ← R4 x R2
k: R2 ← R2 + R8

cyc1:

RS	dst	Qj	Vj	Qk	Vk
1					
2					
3					

Add

RS	dst	Qj	Vj	Qk	Vk
4					
5					

Mult

RF	bsy	rob	value
0	0		6.0
2	0		3.5
4	0		10.0
8	0		7.8

ROB	dst	rdy	value
i			
ii			
iii			
iv			

cyc2:

RS	dst	Qj	Vj	Qk	Vk
1					
2					
3					

Add

RS	dst	Qj	Vj	Qk	Vk
4					
5					

Mult

RF	bsy	rob	value
0			
2			
4			
8			

ROB	dst	rdy	value
i			
ii			
iii			
iv			

cyc3:

RS	dst	Qj	Vj	Qk	Vk
1					
2					
3					

Add

RS	dst	Qj	Vj	Qk	Vk
4					
5					

Mult

RF	bsy	rob	value
0			
2			
4			
8			

ROB	dst	rdy	value
i			
ii			
iii			
iv			

assume 1-cyc add, 2-cyc mult
issue up to 1 add & 1 mult per cycle
complete up to 1 add & 1 mult per cycle

i: R4 ← R0 x R8
j: R0 ← R4 x R2
k: R2 ← R2 + R8

cyc1:

RS	dst	Qj	Vj	Qk	Vk
1					
2					
3					

Add

RS	dst	Qj	Vj	Qk	Vk
4	i	0	6.0	0	7.8
5					

Mult issue 4

RF	bsy	rob	value
0	0		6.0
2	0		3.5
4	1	i	10.0
8	0		7.8

ROB	dst	rdy	value
i	R4	0	-
ii			
iii			
iv			

cyc2:

RS	dst	Qj	Vj	Qk	Vk
1	iii	0	3.5	0	7.8
2					
3					

Add issue 1

RS	dst	Qj	Vj	Qk	Vk
4	i	0	6.0	0	7.8
5	ii	i	-	0	3.5

Mult

RF	bsy	rob	value
0	1	ii	6.0
2	1	iii	3.5
4	1	i	10.0
8	0		7.8

ROB	dst	rdy	value
i	R4	0	-
ii	R0	0	-
iii	R2	0	-
iv			

cyc3:

RS	dst	Qj	Vj	Qk	Vk
1	iii	0	3.5	0	7.8
2					
3					

Add cmpl <iii, 11.3>

RS	dst	Qj	Vj	Qk	Vk
4	i	0	6.0	0	7.8
5	ii	i	-	0	3.5

Mult cmpl <i, 46.8>

RF	bsy	rob	value
0	1	ii	6.0
2	1	iii	3.5
4	1	I	10.0
8	0		7.8

ROB	dst	rdy	value
i	R4	0	-
ii	R0	0	-
iii	R2	0	-
iv			

assume 1-cyc add, 2-cyc mult
issue up to 1 add & 1 mult per cycle
complete up to 1 add & 1 mult per cycle

i: R4 ← R0 x R8
j: R0 ← R4 x R2
k: R2 ← R2 + R8

cyc3:

RS	dst	Qj	Vj	Qk	Vk
1	iii	0	3.5	0	7.8
2					
3					

Add cmpl <iii, 11.3>

RS	dst	Qj	Vj	Qk	Vk
4	i	0	6.0	0	7.8
5	ii	i	-	0	3.5

Mult cmpl <i, 46.8>

RF	bsy	rob	value
0	1	ii	6.0
2	1	iii	3.5
4	1	i	10.0
8	0		7.8

ROB	dst	rdy	value
i	R4	0	-
ii	R0	0	-
iii	R2	0	-
iv			

cyc4:

RS	dst	Qj	Vj	Qk	Vk
1					
2					
3					

Add

RS	dst	Qj	Vj	Qk	Vk
4					
5	ii	0	46.8	0	3.5

Mult issue 5

RF	bsy	rob	value
0	1	ii	6.0
2	1	iii	3.5
4	1	i	10.0
8	0		7.8

ROB	dst	rdy	value
i	R4	1	46.8
ii	R0	0	-
iii	R2	1	11.3
iv			

cyc5:

RS	dst	Qj	Vj	Qk	Vk
1					
2					
3					

Add

RS	dst	Qj	Vj	Qk	Vk
4					
5	ii	0	46.8	0	3.5

Mult

RF	bsy	rob	value
0	1	ii	6.0
2	1	iii	3.5
4	0		46.8
8	0		7.8

ROB	dst	rdy	value
i			
ii	R0	0	-
iii	R2	1	11.3
iv			

WAW Example:

i: R4 ← R0 x R8
j: R2 ← R0 + R4
k: R4 ← R0 + R8
l: R8 ← R4 x R8

cyc1:

RS	dst	Qj	Vj	Qk	Vk
1					
2					
3					

Add

RS	dst	Qj	Vj	Qk	Vk
4					
5					

Mult

RF	bsy	rob	value
0	0		6.0
2	0		3.5
4	0		10.0
8	0		7.8

ROB	dst	rdy	value
i			
ii			
iii			
iv			

cyc2:

RS	dst	Qj	Vj	Qk	Vk
1					
2					
3					

Add

RS	dst	Qj	Vj	Qk	Vk
4					
5					

Mult

RF	bsy	rob	value
0			
2			
4			
8			

ROB	dst	rdy	value
i			
ii			
iii			
iv			

cyc3:

RS	dst	Qj	Vj	Qk	Vk
1					
2					
3					

Add

RS	dst	Qj	Vj	Qk	Vk
4					
5					

Mult

RF	bsy	rob	value
0			
2			
4			
8			

ROB	dst	rdy	value
i			
ii			
iii			
iv			

assume 1-cyc add, 2-cyc mult
issue up to 1 add & 1 mult per cycle
complete up to 1 add & 1 mult per cycle

i: R4 ← R0 x R8
j: R2 ← R0 + R4
k: R4 ← R0 + R8
l: R8 ← R4 x R8

cyc1:

RS	dst	Qj	Vj	Qk	Vk
1	ii	0	6.0	i	-
2					
3					

Add

RS	dst	Qj	Vj	Qk	Vk
4	i	0	6.0	0	7.8
5					

Mult issue 4

RF	bsy	rob	value
0	0		6.0
2	1	ii	3.5
4	1	i	10.0
8	0		7.8

ROB	dst	rdy	value
i	R4	0	-
ii	R2	0	-
iii			
iv			

cyc2:

RS	dst	Qj	Vj	Qk	Vk
1	ii	0	6.0	i	-
2	iii	0	6.0	0	7.8
3					

Add issue 2

RS	dst	Qj	Vj	Qk	Vk
4	i	0	6.0	0	7.8
5	iv	iii	-	0	7.8

Mult

RF	bsy	rob	value
0	0		6.0
2	1	ii	3.5
4	1	iii	10.0
8	1	iv	7.8

ROB	dst	rdy	value
i	R4	0	-
ii	R2	0	-
iii	R4	0	-
iv	R8	0	-

cyc3:

RS	dst	Qj	Vj	Qk	Vk
1	ii	0	6.0	i	-
2	iii	0	6.0	0	7.8
3					

Add cmpl <iii, 13.8>

RS	dst	Qj	Vj	Qk	Vk
4	i	0	6.0	0	7.8
5	iv	iii	-	0	7.8

Mult cmpl <i, 46.8>

RF	bsy	rob	value
0	0		6.0
2	1	ii	3.5
4	1	iii	10.0
8	1	iv	7.8

ROB	dst	rdy	value
i	R4	0	-
ii	R2	0	-
iii	R4	0	-
iv	R8	0	-

assume 1-cyc add, 2-cyc mult
issue up to 1 add & 1 mult per cycle
complete up to 1 add & 1 mult per cycle

i: R4 ← R0 x R8
j: R2 ← R0 + R4
k: R4 ← R0 + R8
l: R8 ← R4 x R8

cyc3:

RS	dst	Qj	Vj	Qk	Vk
1	ii	0	6.0	i	-
2	iii	0	6.0	0	7.8
3					

Add cmpl <iii, 13.8>

RS	dst	Qj	Vj	Qk	Vk
4	i	0	6.0	0	7.8
5	iv	iii	-	0	7.8

Mult cmpl <i, 46.8>

RF	bsy	rob	value
0	0		6.0
2	1	ii	3.5
4	1	iii	10.0
8	1	iv	7.8

ROB	dst	rdy	value
i	R4	0	-
ii	R2	0	-
iii	R4	0	-
iv	R8	0	-

cyc4:

RS	dst	Qj	Vj	Qk	Vk
1	ii	0	6.0	0	46.8
2					
3					

RS	dst	Qj	Vj	Qk	Vk
4					
5	iv	0	13.8	0	7.8

Mult issue *5*

RF	bsy	rob	value
0	0		6.0
2	1	ii	3.5
4	1	iii	10.0
8	1	iv	7.8

ROB	dst	rdy	value
i	R4	1	46.8
ii	R2	0	-
iii	R4	1	13.8
iv	R8	0	-

cyc5: Add issue *1*

RS	dst	Qj	Vj	Qk	Vk
1	ii	0	6.0	0	46.8
2					
3					

Add cmpl <ii, 52.8>

RS	dst	Qj	Vj	Qk	Vk
4					
5	iv	0	13.8	0	7.8

Mult

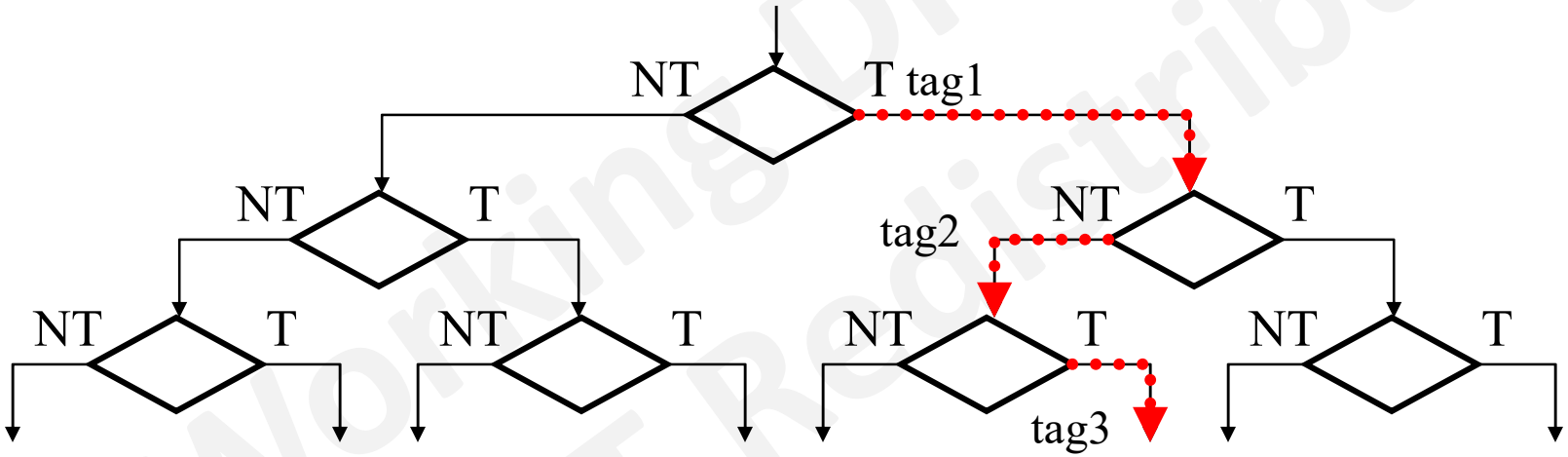
RF	bsy	rob	value
0	0		6.0
2	1	ii	3.5
4	1	iii	46.8
8	1	iv	7.8

ROB	dst	rdy	value
i			
ii	R2	0	-
iii	R4	1	13.8
iv	R8	0	-

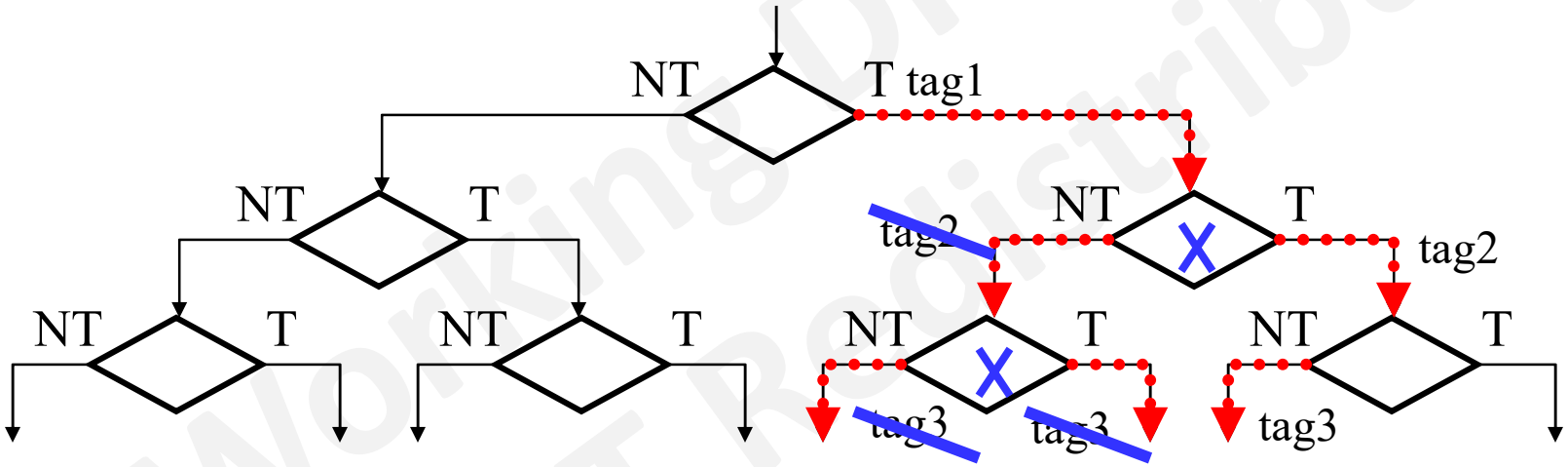
Practical Speculative Out-of-order

- A mispredicted branch after resolution must be rewound and restarted **ASAP**!
- Much trickier than 5-stage pipeline . . .
  - can rewind to an intermediate speculative state
  - a rewound branch could still be speculative and itself be discarded by another rewind!
  - rewind must reestablish both architectural state (register value) and microarchitecture state (e.g., rename table)
  - rewind/restart must be fast (not infrequent)
- Also need to rewind on exceptions . . . .but easier

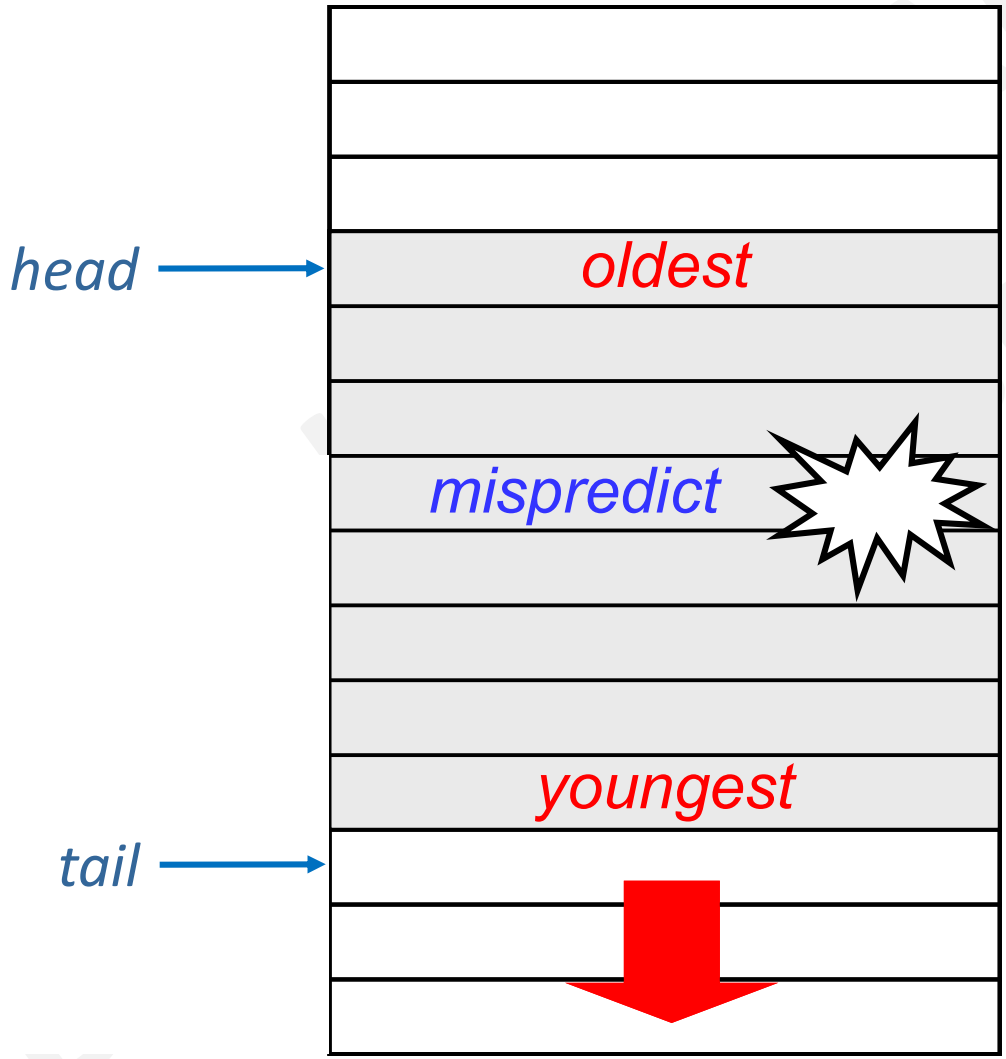
Nested Control Flow Speculation



Mis-speculation Recovery can be Speculative



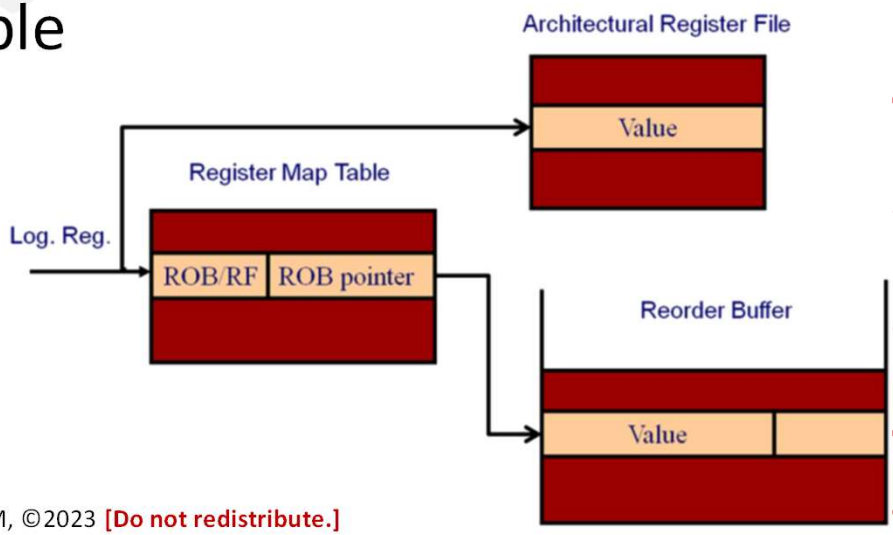
ASAP Midpoint ROB Rewind



Speculative Rewinding with ROB

- In-order RF state never needs undo'ing √
- Lookahead RF state tied to ROB √
- Restoring architectural state view (i.e. map table)
  - at decode, record in ROB, the logical dest and the overwritten previous mapping
  - on rewind, walk-back ROB one entry at a time to restore register map table

What happens if previous mapping is to an already retired ROB entry? How to know that?



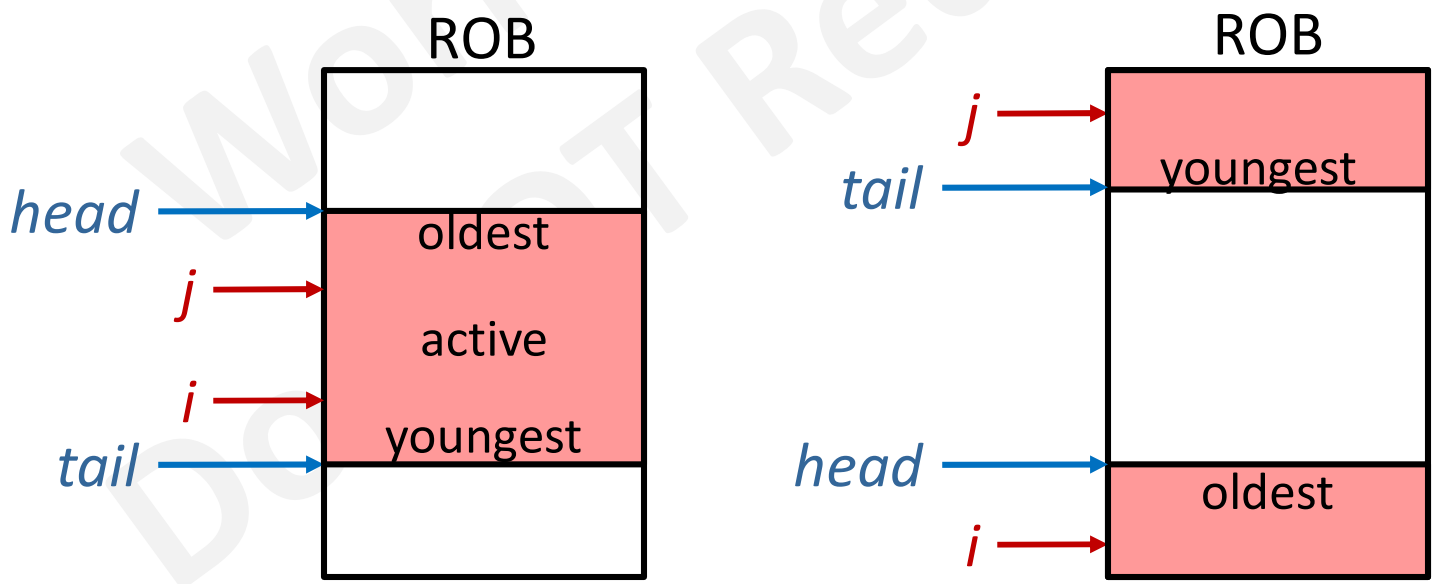
from [Gonzalez, et al., 2010]

Killing In-Flight Instruction

- Tag in-flight instructions by their program-ordered ROB index
- On a mispredicted branch, selectively kill younger instructions based on index comparison

This way needs lots and lots of substractors

- How to determine if index ***i*** "younger *than*" ***j***?



What do we know so far

- Out-of-order: yes, except IPC<<1 when ILP=1
- Superscalar: not exactly
  - superscalar execute out of multiple RS
  - everything else discussed as single atomic action
  - how to do more than 1/cyc quickly and cheaply?
- Speculative: yes, except
  - rewind take as long as degree out-of-order
  - still need to a way to decide, given two ROB entry indices which is younger; costly subtractors

Building what we said today would only be bigger and slower than your 5-stage pipeline

Reading for Next Week

- Popescu, et al., The Metaflow Architecture, 1991.

References

- Popescu, et al., The Metaflow Architecture, 1991.
- Yeager, MIPS R10K Superscalar Microprocessor, 1996.
- Gonzalez, et al., Processor Microarchitecture: An Implementation Perspective, Synthesis Lectures, 2010.
- Hennessy&Patterson, Computer Architecture: A Quantitative Approach, any edition.
- Johnson, Superscalar Microprocessor Design, 1990.
- Shen&Lipasti, Modern Processor Design: Fundamentals of Superscalar Processors, 2013.