

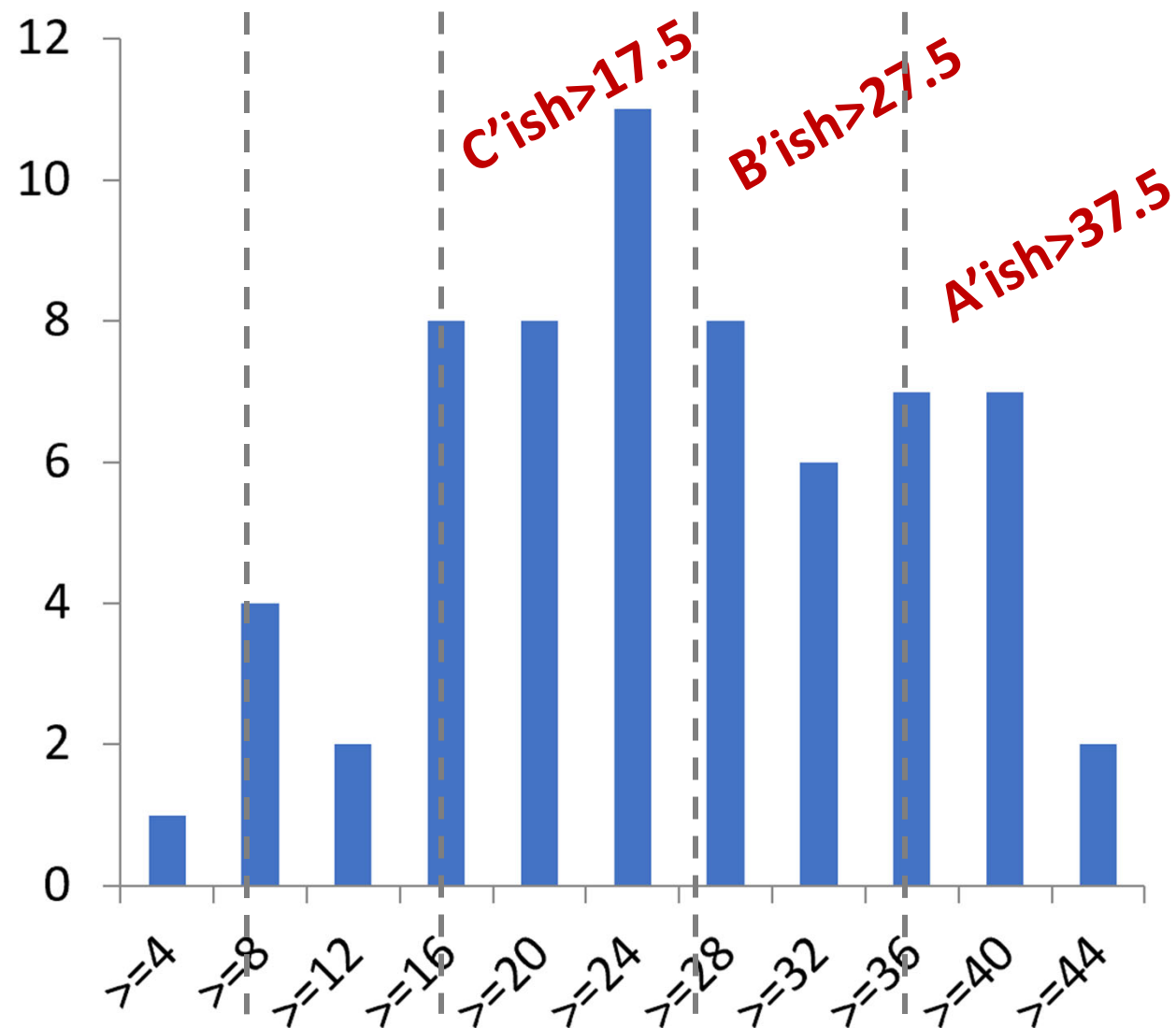
# **18-447 Lecture 20: ILP to Multicores**

James C. Hoe

Department of ECE

Carnegie Mellon University

# Midterm 2 Histogram



# Housekeeping

- Your goal today
  - transition from sequential to parallel
  - enjoy (only slides before S14 on midterm3)
- Notices
  - get going on Lab 4: **status check 4/26, due 5/7**
  - Handout #15: HW5
- Readings (advanced optional)
  - MIPS R10K Superscalar Microprocessor, Yeager
  - Synthesis Lectures: *Processor Microarchitecture: An Implementation Perspective*, 2010

# Parallelism Defined

- $T_1$  (work measured in time):
  - time to do work with 1 PE
- $T_\infty$  (critical path):
  - time to do work with infinite PEs
  - $T_\infty$  bounded by dataflow dependence

- Average parallelism: *let's call  $p$  concurrency*  

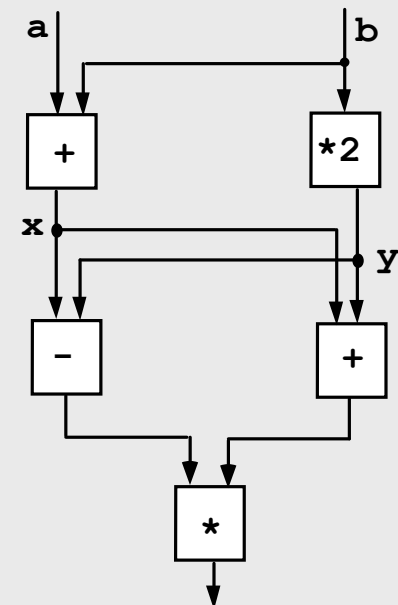
$$P_{avg} = T_1 / T_\infty$$
- For a system with  $p$  PEs

$$T_p \geq \max\{T_1/p, T_\infty\}$$

- When  $P_{avg} \gg p$   

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

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



# ILP: Instruction-Level Parallelism

- Average **ILP** =  $T_1 / T_\infty$   
= no. instruction / no. cyc required

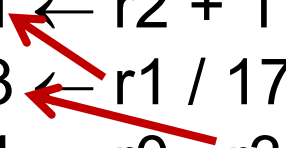
code1: **ILP** = 1

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
```



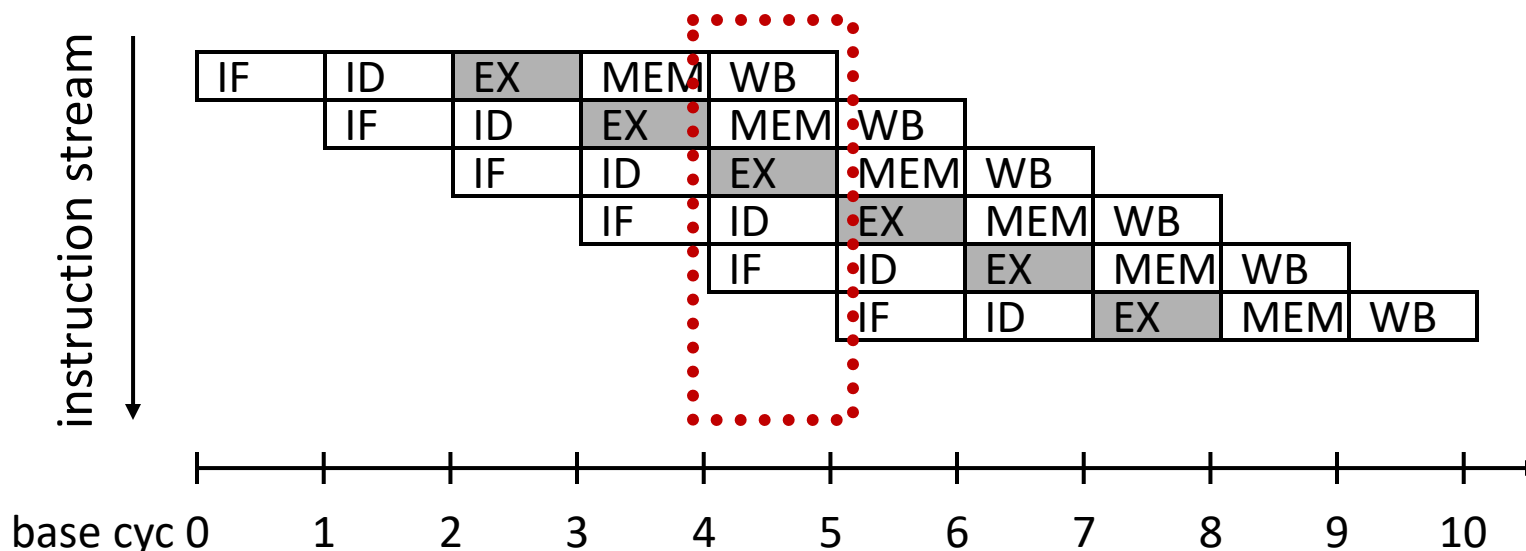
```
code2:  r1 ← r2 + 1
        r3 ← r9 / 17
        r4 ← r0 - r10
```

# 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**  $\geq 1$  to avoid stall

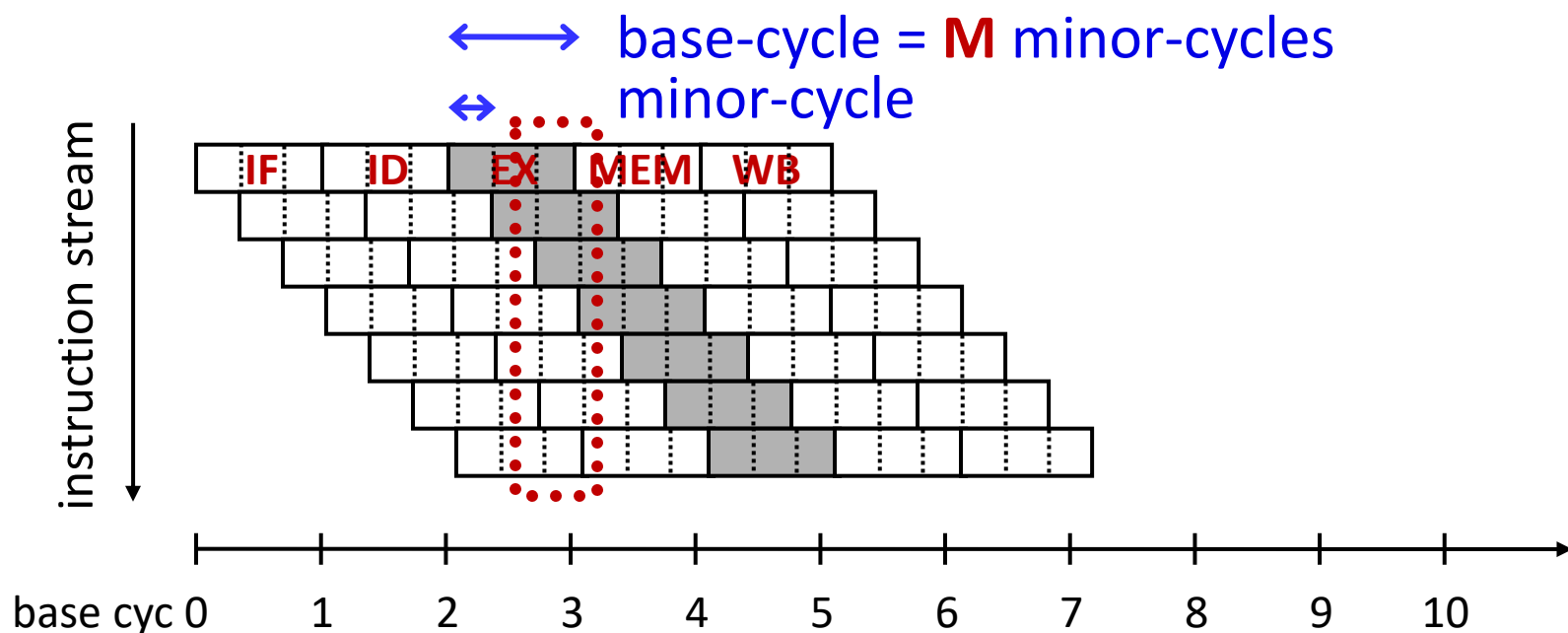


# Superpipelined Execution

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

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

required **ILP**  $\geq$  **M**



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

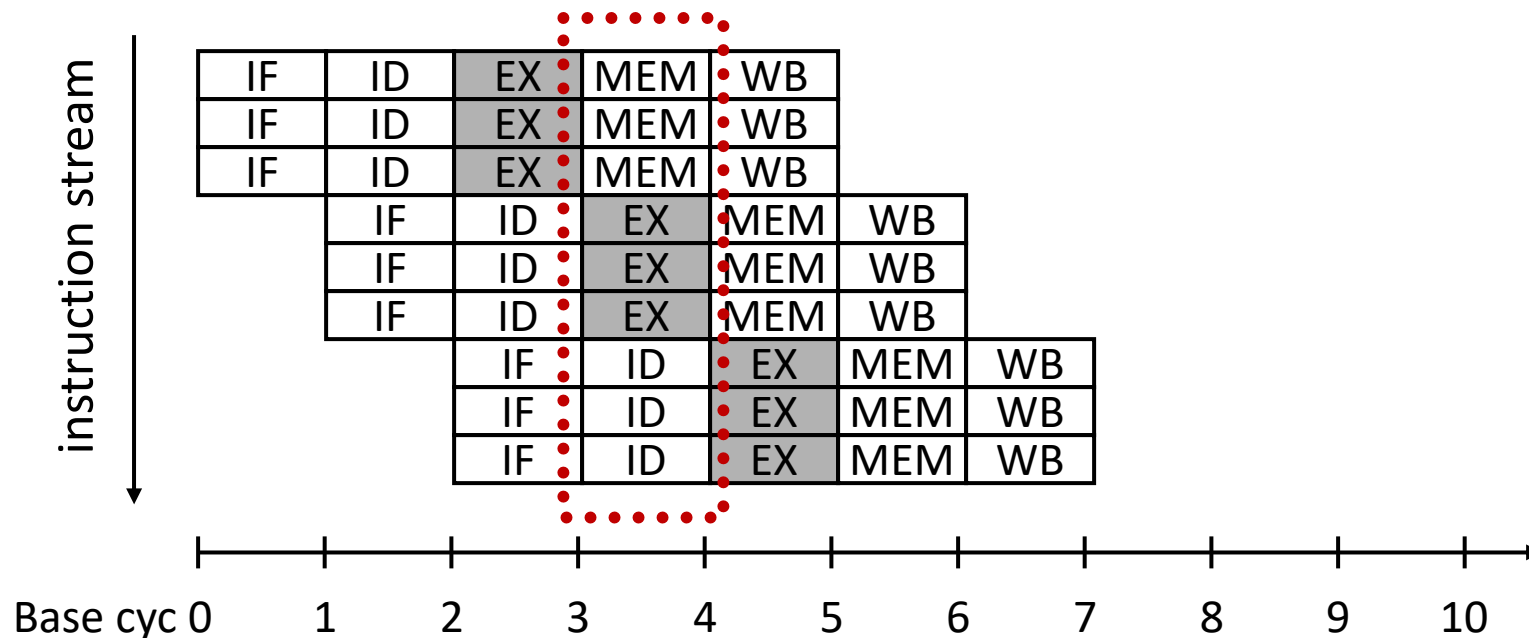


# Superscalar (Inorder) Execution

**OL** = 1 base cycle

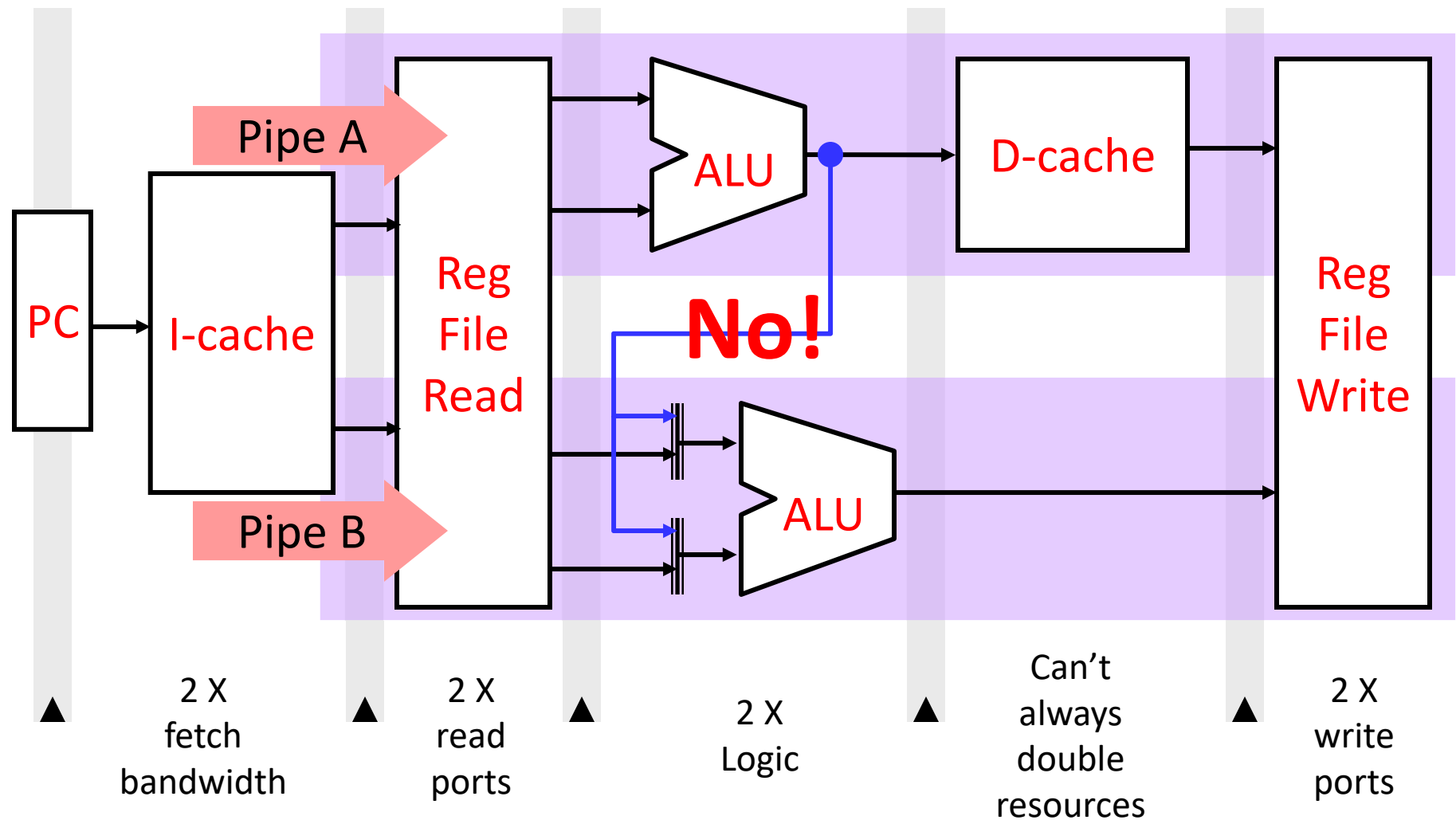
peak **IPC** = **N**

required **ILP**  $\geq$  **N**

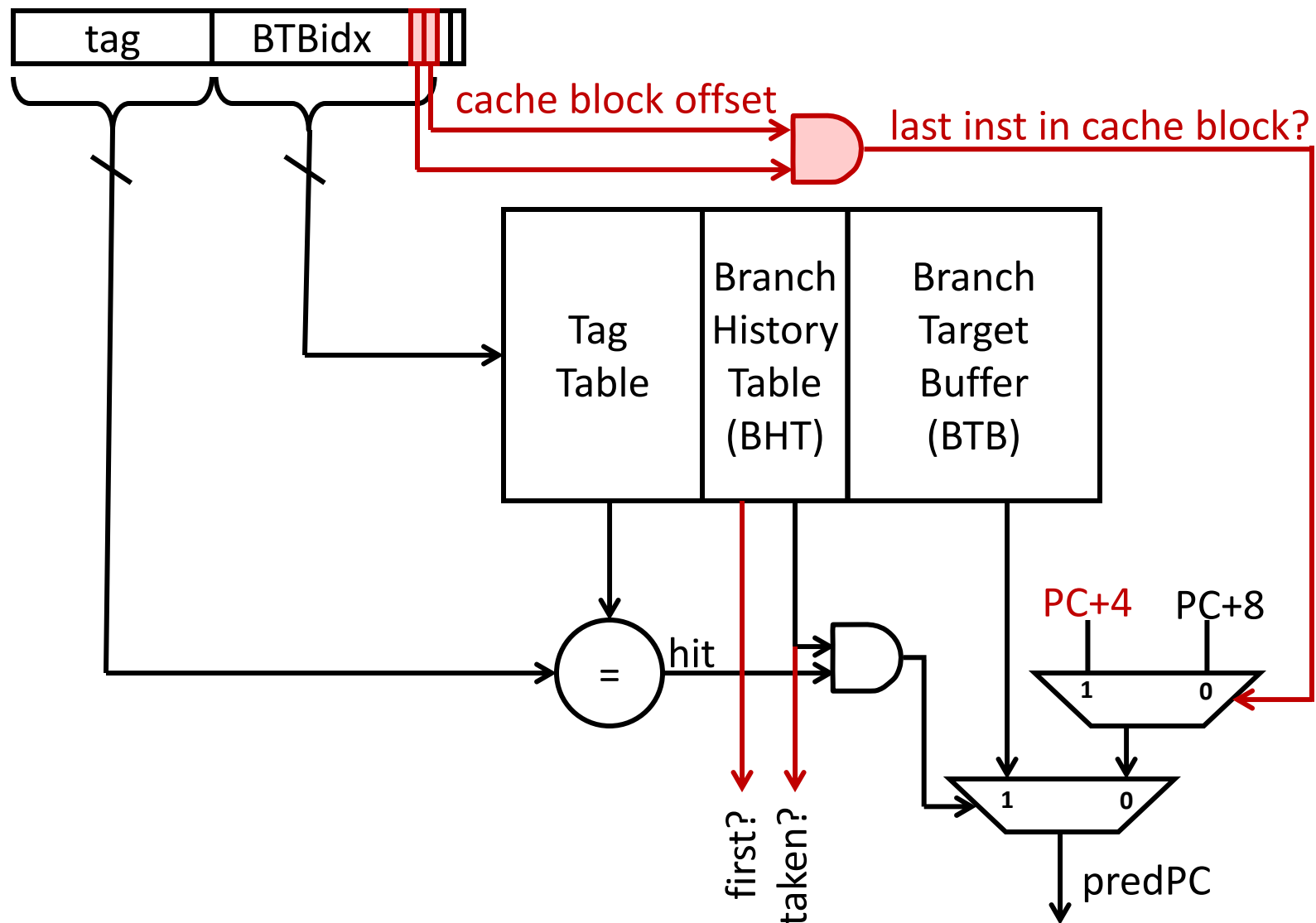


Achieving full performance requires finding **N**  
"independent" instructions on every cycle

# Lab 4: 2-way, In-order Superscalar

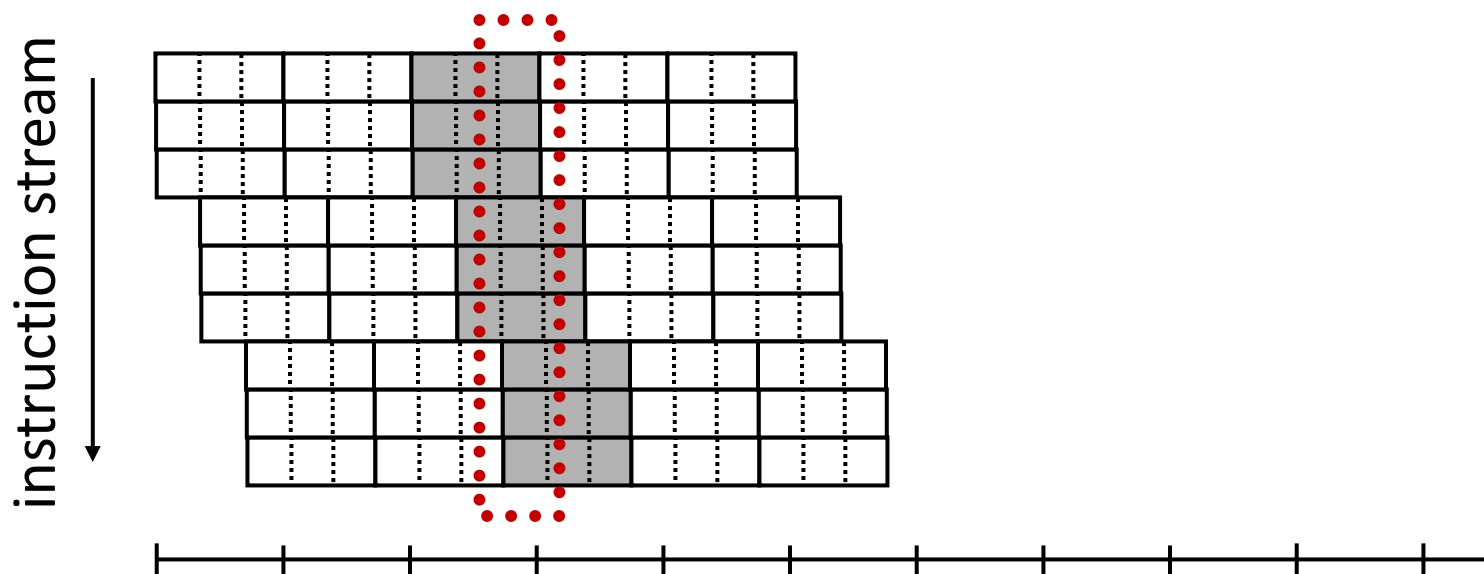


# Lab 4: 2-way Supercalar BP



# Limitations of Inorder Pipeline

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



# Out-of-Order Execution

- **ILP** is scope dependent

**ILP=1** {  
r1  $\leftarrow$  r2 + 1  
r3  $\leftarrow$  r1 / 17  
r4  $\leftarrow$  r0 - r3  
r11  $\leftarrow$  r12 + 1  
r13  $\leftarrow$  r19 / 17  
r14  $\leftarrow$  r0 - r20  
} **ILP=2**

Accessing **ILP=2** requires (1) larger scheduling window but also (2) out-of-order execution  
(even Lab 4 can go slightly OOO)

# Superscalar Speculative Out-of-Order Execution

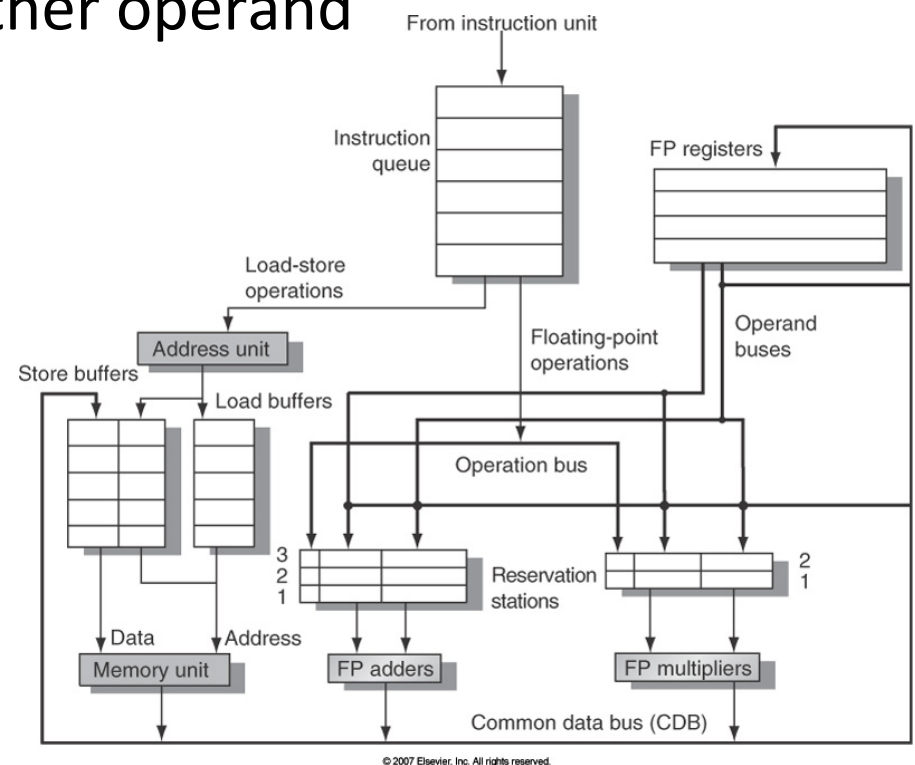
# Dataflow Execution Ordering

- Maintain a buffer of many pending instructions, a.k.a. reservation stations (**RSs**)
  - wait for functional unit to be free
  - wait for register RAW hazards to resolve (i.e., required input operands to be produced)
- 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 (heuristic)
- A completing instruction frees pending, RAW-dependent instructions to execute

*What about WAW and WAR?*

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

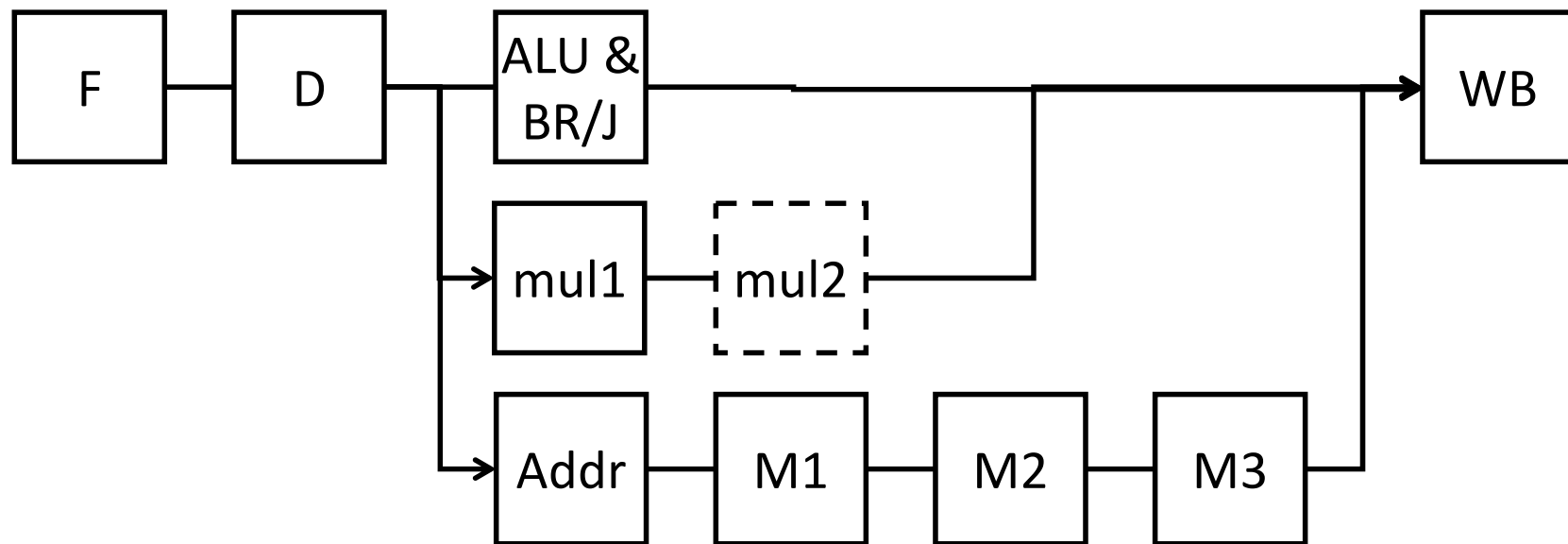
- Dispatch an instruction to a **RS** slot after decode
  - decode received from RF either operand value or placeholder **RS-tag**
  - mark RF dest with **RS-tag** of current inst's **RS** slot
- Inst in **RS** can issue when all operand values ready
- Completing instruction, in addition to updating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





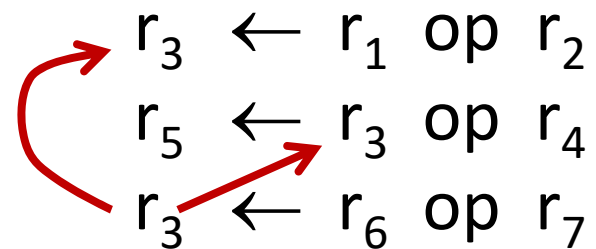
# WAW and WAR

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



# Removing False Dependencies

- With out-of-order execution comes WAW and WAR hazards
- Anti and output dependencies are false dependencies on register names rather than data



- With infinite number of registers, anti and output dependencies avoidable by using a new register for each new value

# Register Renaming: Example

Original

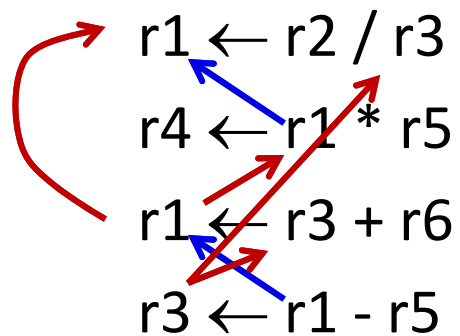


Diagram illustrating the original code with data flow arrows:

- $r1 \leftarrow r2 / r3$
- $r4 \leftarrow r1 * r5$
- $r1 \leftarrow r3 + r6$
- $r3 \leftarrow r1 - r5$

Arrows indicate dependencies: a red arrow from  $r2$  to the first  $r1$ , a blue arrow from the first  $r1$  to  $r4$ , a red arrow from  $r3$  to the second  $r1$ , a blue arrow from the second  $r1$  to  $r3$ , and a red arrow from  $r5$  to both  $r4$  and  $r3$ . A red curved arrow also points from the first  $r1$  to the second  $r1$ , indicating a write-after-read dependency.

Renamed

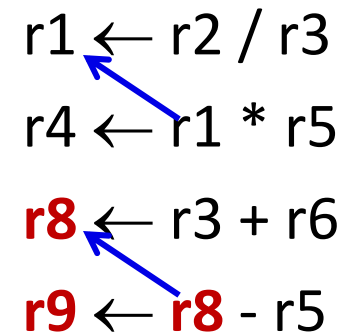
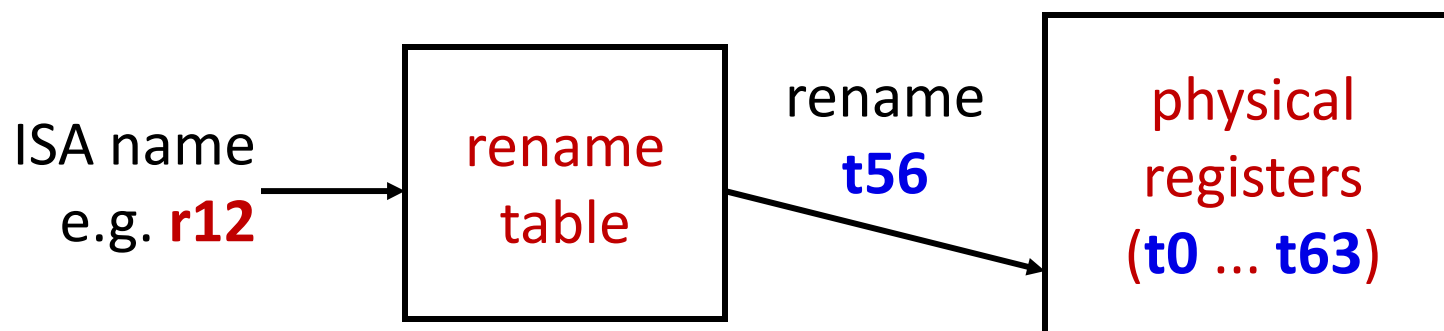


Diagram illustrating the renamed code with data flow arrows:

- $r1 \leftarrow r2 / r3$
- $r4 \leftarrow r1 * r5$
- $r8 \leftarrow r3 + r6$
- $r9 \leftarrow r8 - r5$

Arrows indicate dependencies: a blue arrow from  $r1$  to  $r4$ , a red arrow from  $r3$  to  $r8$ , and a blue arrow from  $r8$  to  $r9$ . The registers  $r8$  and  $r9$  are highlighted in red in the original image.

# On-the-fly HW Register Renaming



- Maintain mapping from ISA reg. names to physical registers
- When decoding an instruction that updates ' $r_x$ ':
  - allocate unused physical register  $t_y$  to hold inst result
  - set new mapping from ' $r_x$ ' to  $t_y$
  - younger instructions using ' $r_x$ ' as input finds  $t_y$
- De-allocate a physical register for reuse when it is never needed again?

^^^^^when is this exactly?

$r1 \leftarrow r2 / r3$

$r4 \leftarrow r1 * r5$

$r1 \leftarrow r3 + r6$

# Superscalar **Speculative** Out-of-Order Execution

# Control Speculation

- Modern CPUs can have over 100 instructions in out-of-order execution scope
  - if 14% of avg. instruction mix is control flow, what is average distance between control flow?
  - instruction fetch must make multiple levels of branch predictions (condition and target) to fetch far ahead of execution and commit

---

- **Question:**

- **how much more ILP is uncovered with look ahead**
  - **how much useful work is done during look ahead**

**Ans: not much and not much**

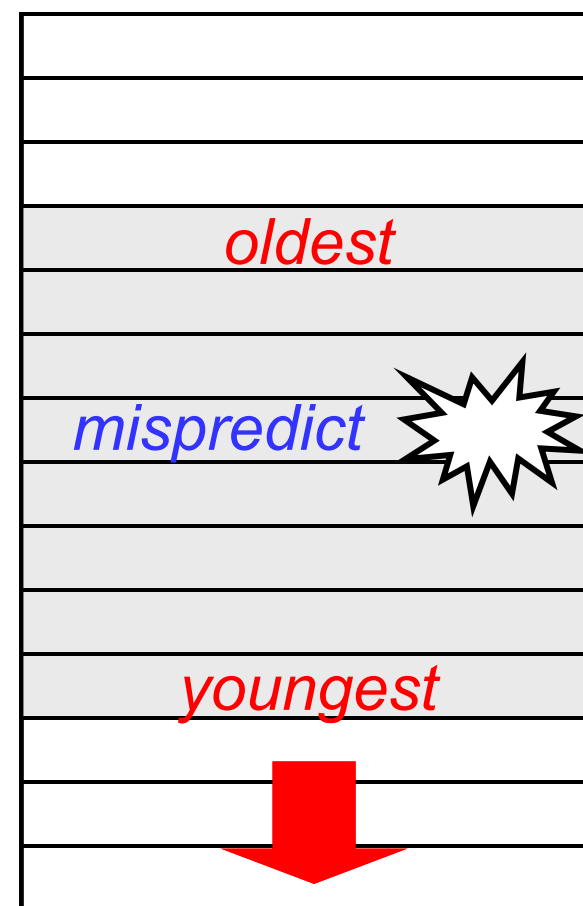
# Speculative Out-of-order Execution

- A mispredicted branch after resolution must be rewound and restarted and fast!
- Much trickier than 5-stage pipeline . . .
  - can rewind to an intermediate speculative state
  - a rewind 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

# 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





# In-order vs Speculative State

- In-order state:
  - cumulative architectural effects of all instructions committed in-order so far
  - can never be undone!!
- Speculative state, as viewed by a given inst in **ROB**
  - in-order state + effects of older inst's in **ROB**
  - effects of some older inst's may be pending
- Speculative state effects must be reversible
  - remember both in-order and speculative values for an RF register (may have multiple speculative values)
  - store inst updates memory only at commit time
- Discard younger speculative state to rewind execution to oldest remaining inst in **ROB**

# Not Entirely New Concept for You

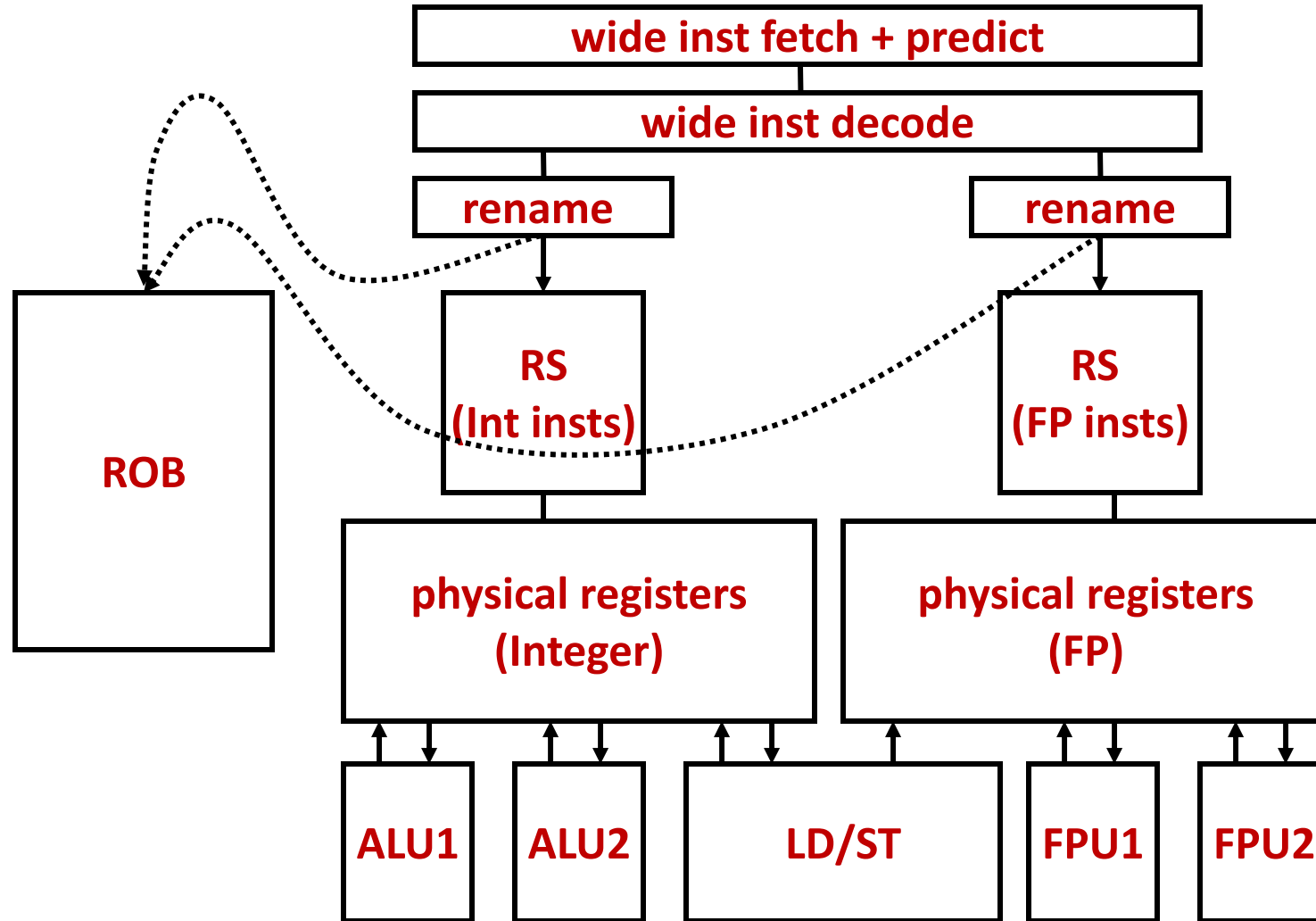
privileged mode

	$t_0$	$t_1$	$t_2$	$t_3$	$t_4$	$t_5$	$t_6$	$t_7$	$t_8$	$t_9$	$t_{10}$
IF	$I_0$	$I_1$	$I_2$	$I_3$	$I_4$	bub	bub	bub	$I_h$	$I_{h+1}$	$I_{h+2}$
ID		$I_0$	$I_1$	$I_2$	$I_3$	bub	bub	bub	bub	$I_h$	$I_{h+1}$
EX			$I_0$	$I_1$	$I_2$	$I_3$	bub	bub	bub	bub	$I_h$
MEM				$I_0$	$I_1$	$I_2$	bub	bub	bub	bub	bub
WB					$I_0$	$I_1$	$I_2$	bub	bub	bub	bub

- Kill faulting and younger inst; drain older inst
- Don't start handler until faulting inst. is oldest
- Better yet, don't start handler until pipeline is empty

Better to be safe than to be fast

# Superscalar Speculative OOO All Together



# Truth about Superscalar Speculative OOO

- If memory speed kept up with core speed, we would still be building in-order pipelines
- But, by 2005 we were seeing e.g., Intel P4 at 4+GHz
  - 16KB L1 D-cache
    - $t_1 = 4$  cyc int (9 cycle fp)
  - 1024KB L2 D-cache
    - $t_2 = 18$  cyc int (18 cyc fp)
  - Main memory
    - $t_3 = \sim 50$ ns or 180 cyc
- Speculative OOO has really been about
  - finding independent work to do after cache hit&miss
  - getting to future cache misses as early as possible
  - overlapping multiple cache misses for BW

# At the 2005 Peak of Superscalar OOO

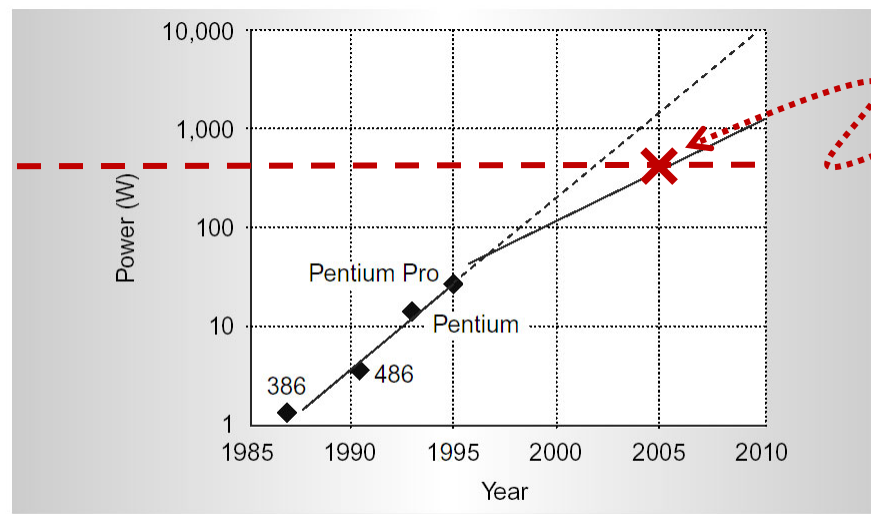
	Alpha 21364	AMD Opteron	Intel Xeon	IBM Power5	MIPS R14000	Intel Itanium2
clock (GHz)	1.30	2.4	<b>3.6</b>	1.9	0.6	1.6
issue rate	4	3 (x86)	3 (rop)	<b>8</b>	4	8
pipeline int/fp	7/9	9/11	<b>22/24</b>	12/17	6	8
inst in flight	80	72(rop)	126 rop	<b>200</b>	48	inorder
rename reg	48+41	36+36	128	48/40	32/32	<b>328</b>
transistor ( $10^6$ )	135	106	125	276	7.2	<b>592</b>
power (W)	<b>155</b>	86	103	120	16	130
SPECint 2000	904	1,566	1,521	1,398	483	<b>1,590</b>
SPECfp 2000	1279	1,591	1,504	2,576	499	<b>2,712</b>

# At peak minus 5 years

	Alpha 21264	AMD Athlon	Intel P4	MIPS R12000	IBM Power3	HP PA8600	SUN Ultra3
clock (MHz)	833	1200	<b>1500</b>	400	450	552	900
issue rate	4	3 (x86)	3 (rop)	4	4	4	4
pipeline int/fp	7/9	9/11	<b>22/24</b>	6	7/8	7/9	14//15
inst in flight	80	72(rop)	<b>126</b> rop	48	32	56	inorder
rename reg	48+41	36+36	<b>128</b>	32+32	16+24	56	inorder
transistor ( $10^6$ )	15.4	37	42	7.2	23	<b>130</b>	29
power (W)	75	<b>76</b>	55	25	36	60	65
SPECint 2000	518		<b>524</b>	320	286	417	438
SPECfp 2000	<b>590</b>	304	549	319	356	400	427

# Performance (In)efficiency

- To hit “expected” performance target
  - push frequency harder by deepening pipelines
  - used the 2x transistors to build more complicated microarchitectures so fast/deep pipelines don’t stall (i.e., caches, BP, superscalar, out-of-order)
- The consequence of performance inefficiency is



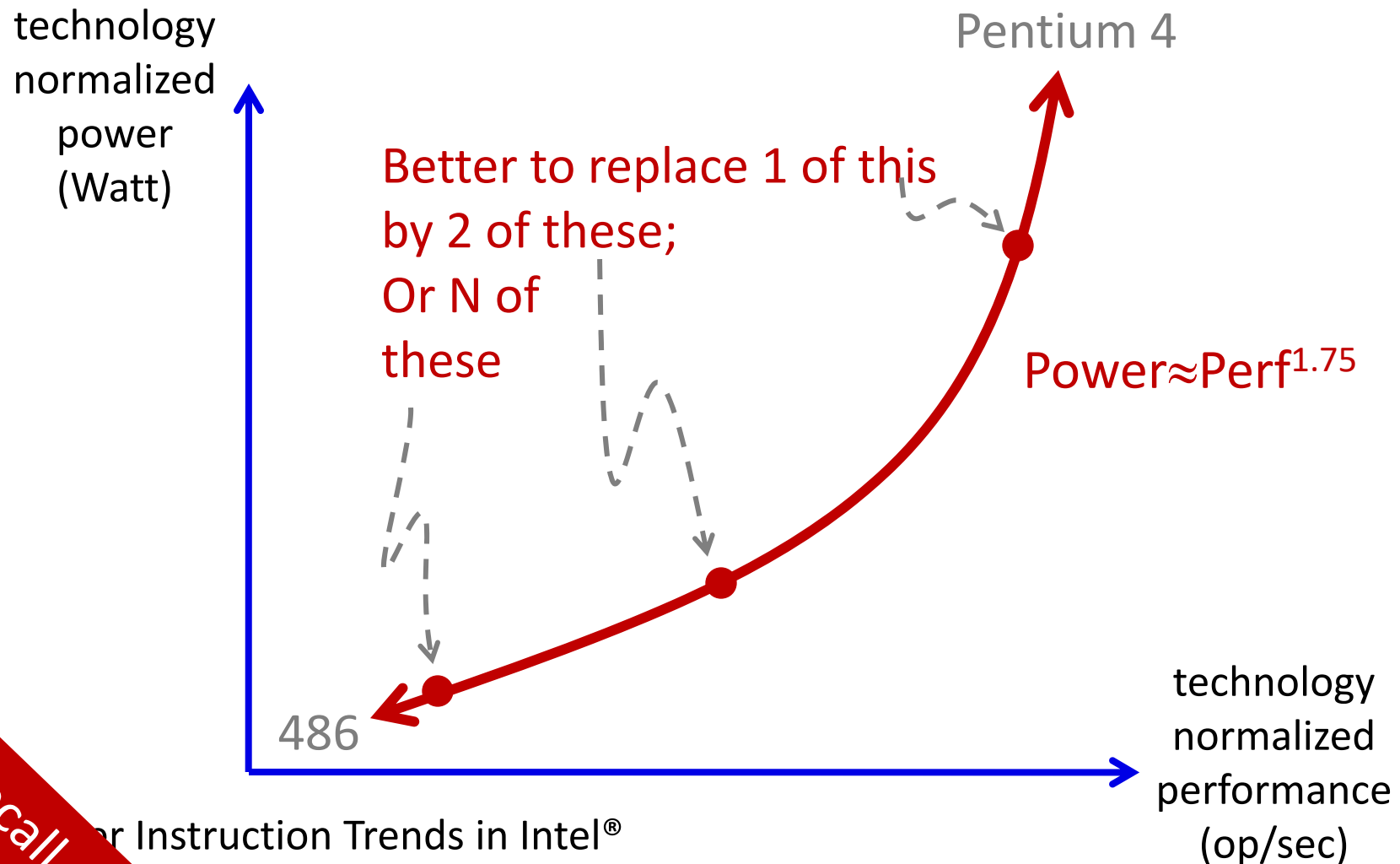
2005, Intel  
P4 Tehas 150W

limit of  
economical  
cooling [ITRS]

Recall

Figure 8. Power dissipation projections. [Borkar, IEEE Micro, July 1999]

# Efficiency of Parallel Processing



Recall

[End of Instruction Trends in Intel®  
Microprocessors, Grochowski et al., 2006]



# At peak plus 1 year

	AMD 285	Intel 5160	Intel 965	Intel Itanium2	IBM P5+	MIPS R16000	SUN Ultra4
cores/threads	<b>2x1</b>	<b>2x2</b>	<b>2x2</b>	<b>2x2</b>	<b>2x2</b>	1x1	<b>2x1</b>
clock (GHz)	2.6	3.03	<b>3.73</b>	1.6	2.3	0.7	1.8
issue rate	3 (x86)	4 (rop)	3 (rop)	6	<b>8</b>	4	4
pipeline depth	11	14	<b>31</b>	8	17	6	14
inst in flight	72(rop)	96(rop)	126(rop)	inorder	<b>200</b>	48	inorder
on-chip\$ (MB)	2x1	4	2x2	<b>2x13</b>	1.9	0.064	2
transistor ( $10^6$ )	233	291	376	<b>1700</b>	276	7.2	295
power (W)	95	80	<b>130</b>	104	100	17	90
SPECint 2000 per core	<b>1942</b>	(1556 <sup>+</sup> )	1870	1474	1820	560	1300
SPECfp 2000 per core	2260	(1694 <sup>+</sup> )	2232	3017	<b>3369</b>	580	1800

\*3086/+2884 according to [www.spec.org](http://www.spec.org)

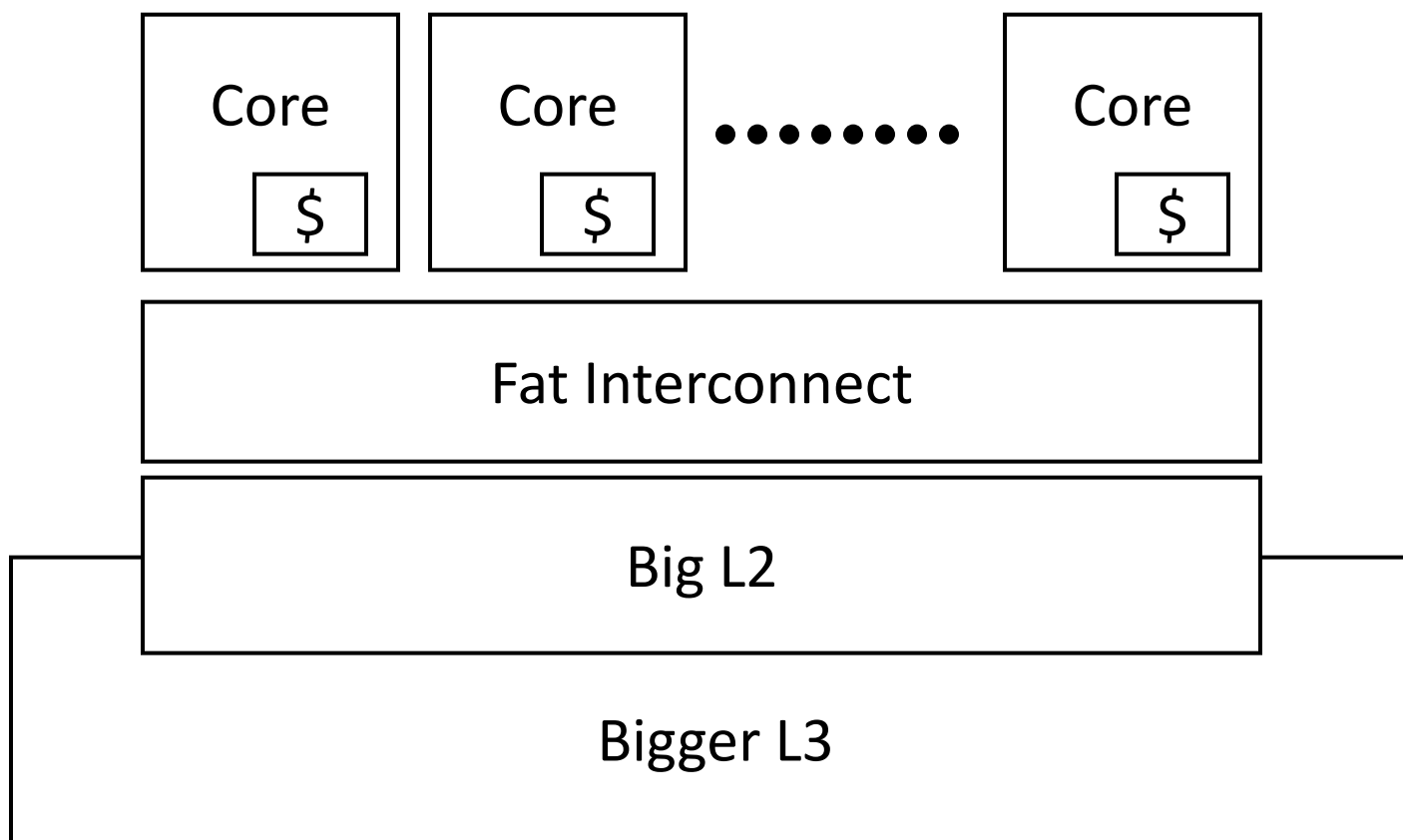
18-447-S21-L20-S33, James C. Hoe, CMU/ECE/CALCM, ©2021

Microprocessor Report, Aug 2006

# At peak plus 3 years

	AMD Opteron 8360SE	Intel Xeon X7460	Intel Itanium 9050	IBM P5	IBM P6	Fijitsu SPARC 7	SUN T2
cores/threads	4x1	<b>6x1</b>	2x2	2x2	2x2	4x2	<b>8x8</b>
clock (GHz)	2.5	2.67	1.60	2.2	<b>5</b>	2.52	1.8
issue rate	3 (x86)	4 (rop)	6	5	<b>7</b>	4	2
pipeline depth	12/ <b>17</b>	14	8	15	13	15	8/12
out-of-order	72(rop)	96(rop)	inorder	<b>200</b>	limited	64	inorder
on-chip\$ (MB)	2+2	<b>9+16</b>	1+12	1.92	8	6	4
transistor ( $10^6$ )	463	<b>1900</b>	1720	276	790	600	503
power max(W)	105	130	104	100	>100	<b>135</b>	95
SPECint 2006 per-core/total	14.4/170	<b>22/274</b>	14.5/1534	10.5/197	15.8/1837	10.5/ <b>2088</b>	--/142
SPECfp 2006 per-core/total	18.5/156	22/142	17.3/1671	12.9/229	20.1/1822	<b>25.0/1861</b>	--/111

# On to Mainstream Parallelism in Multicores and Manycores



Remember, we got here because we need to compute  
faster while using less energy per operation