

# 18-447 Lecture 20: ILP to Multicores

James C. Hoe

Department of ECE

Carnegie Mellon University

# Housekeeping

- Your goal today
  - transition from sequential to parallel
  - enjoy (only first part, before OOO, on 447 exam)
- Notices
  - Midterm Regrades past due
  - HW4, **due 4/8**
  - Lab 4, **less than 4 weeks to go**
  - **NO LECTURE on MONDAY**
- Readings (advanced optional)
  - MIPS R10K Superscalar Microprocessor, Yeager
  - Synthesis Lectures: *Processor Microarchitecture: An Implementation Perspective, 2010*

# Parallelism Defined

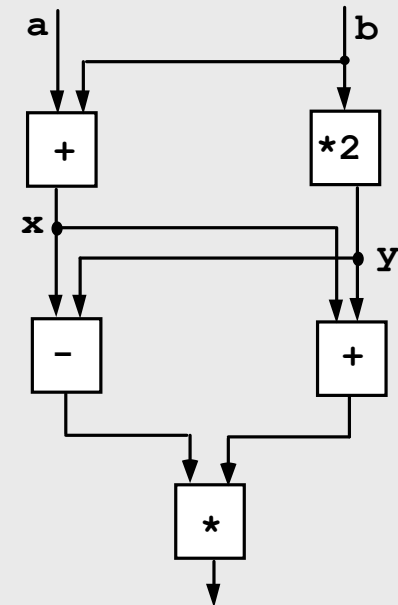
- $T_1$  (work measured in time):
  - time to do work with 1 PE
- $T_\infty$  (critical path):
  - time to do work with infinite PEs
  - $T_\infty$  bounded by dataflow dependence

- 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)
```



[Shiloach&Vishkin]

# 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:	$r1 \leftarrow r0 + 1$
	$r3 \leftarrow r1 + r12$
	$r5 \leftarrow r14 - r3$

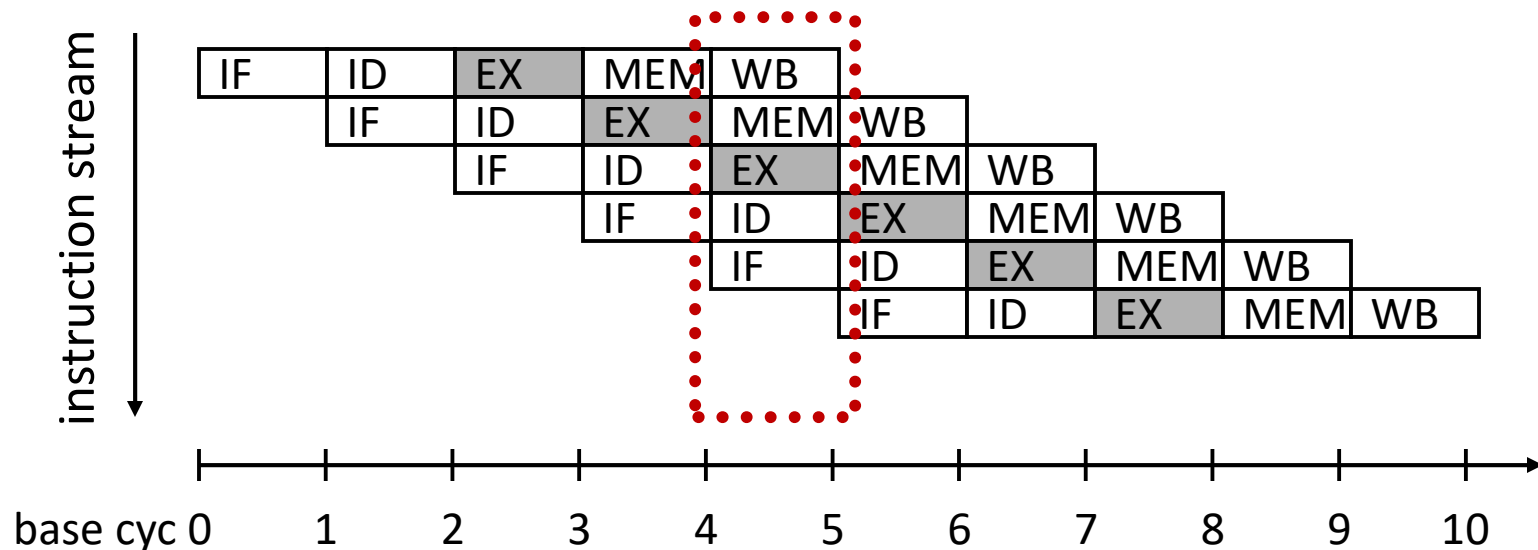
code2:	$r1 \leftarrow r0 + 1$
	$r3 \leftarrow \underline{r11} + r12$
	$r5 \leftarrow r14 - \underline{r13}$

# Add Superscalar to Pipelining to get more Performance from ILP

# Exploiting **ILP** for Performance

Scalar in-order pipeline with forwarding

- operation latency (**OL**)= **1** base cycle
- peak **IPC** = **1** // *no concurrency*
- require **ILP**  $\geq$  **1** to avoid stall

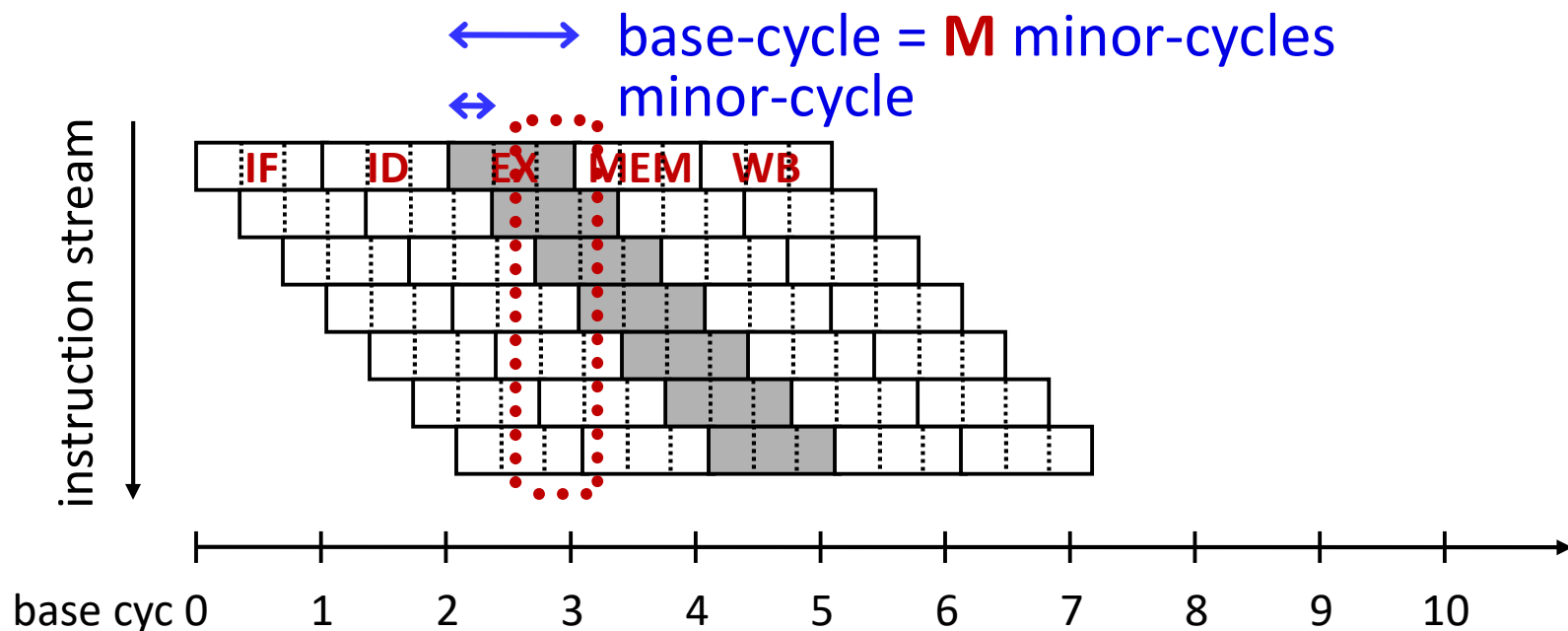


# Superpipelined Execution

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

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

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



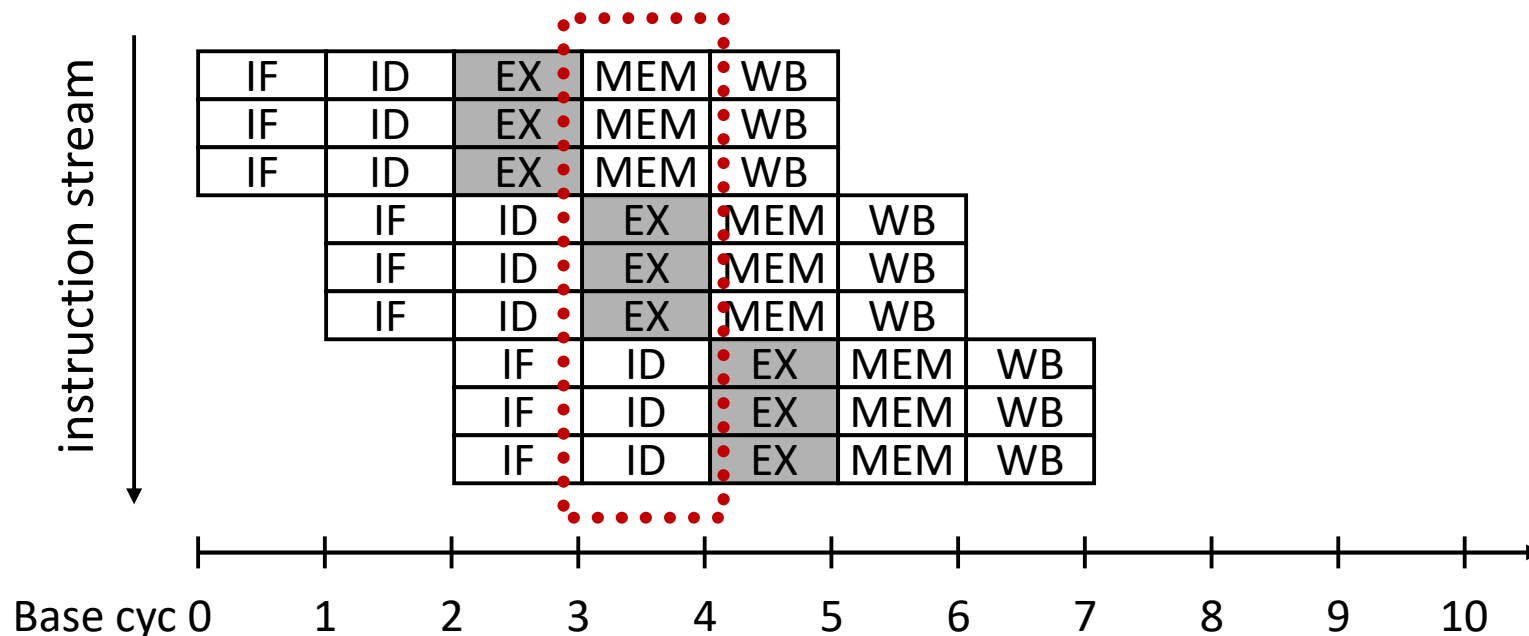
Achieving full performance requires always finding **M** "independent" instructions in a row

# Superscalar (Inorder) Execution

**OL** = 1 base cycle

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

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

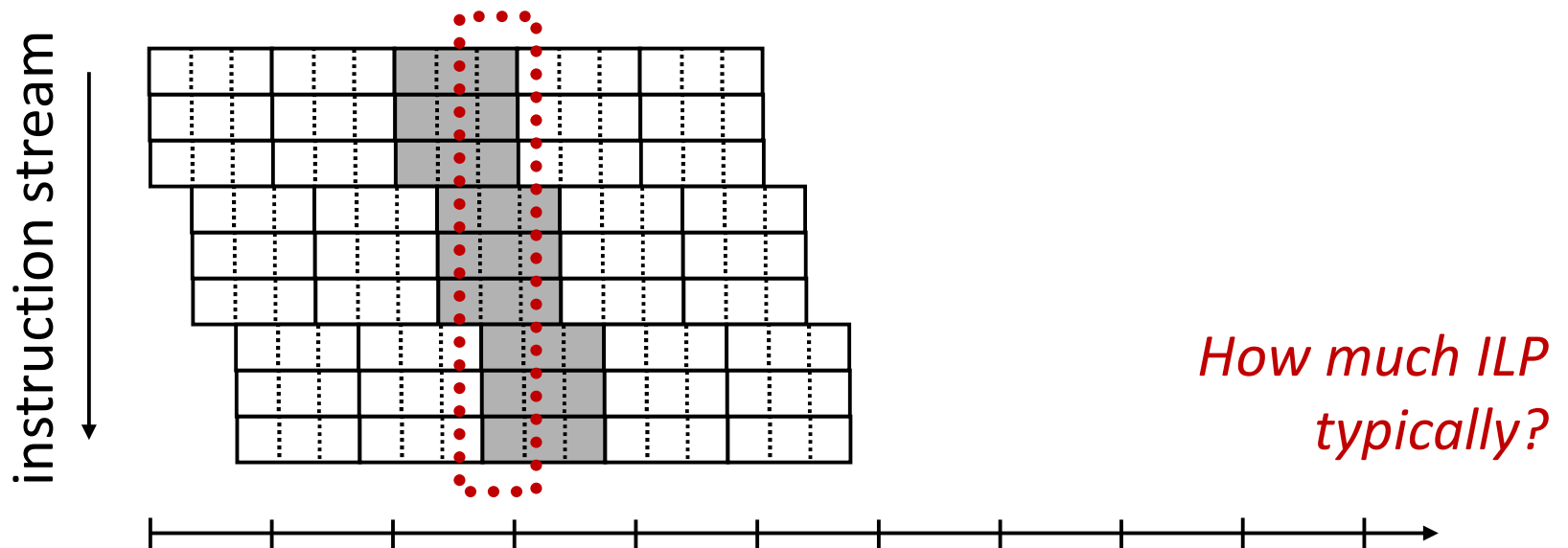


Achieving full performance requires finding **N**  
“independent” instructions on every cycle

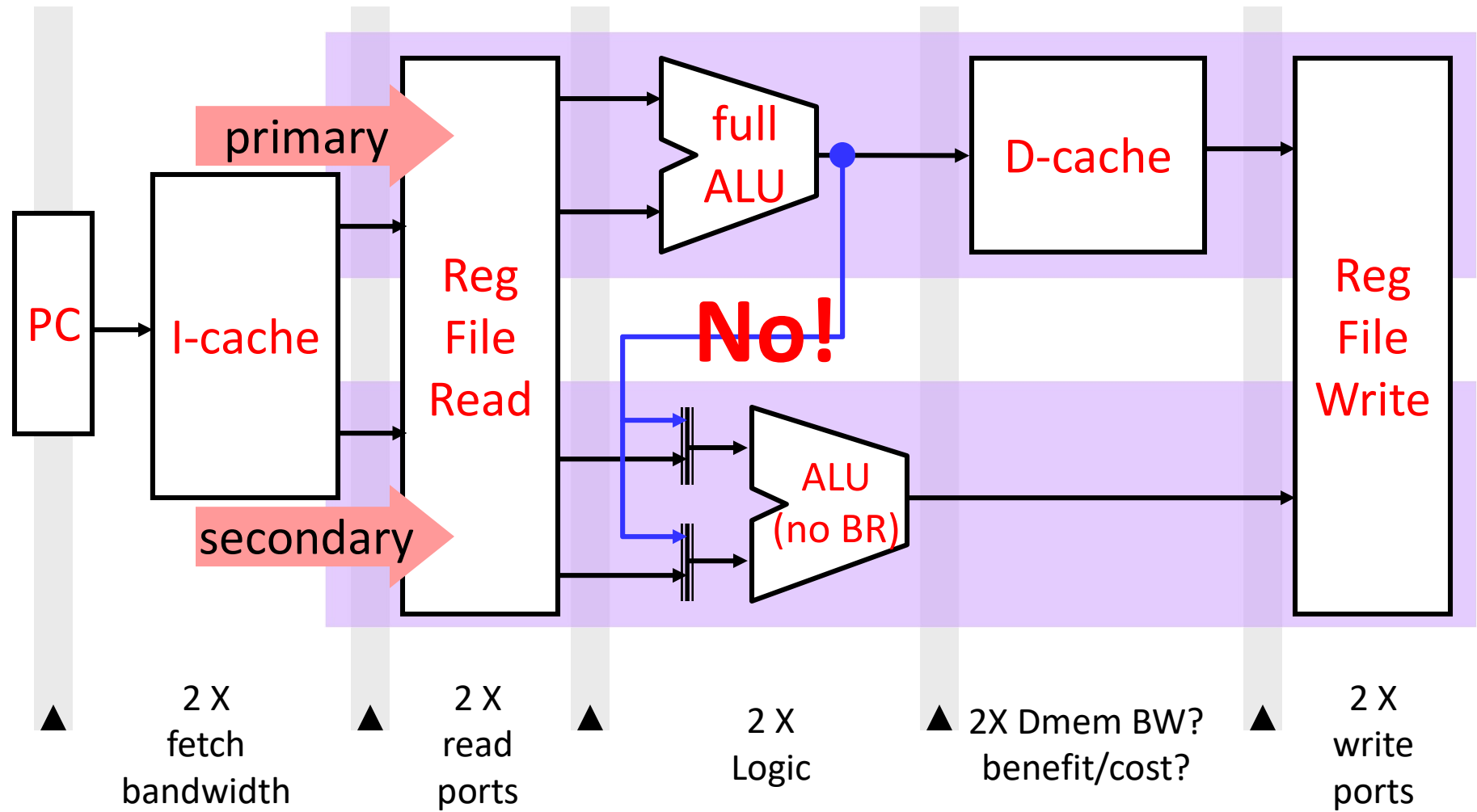


# Limitations of Inorder Pipeline

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



# E.g., 2-way, In-order Superscalar



# Sample Assembly [P&H]

for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ..... }

```

for2tst:
  addi  $s1, $s0, -1
  slti  $t0, $s1, 0
  bne   $t0, $zero, exit2
  sll   $t1, $s1, 2
  add   $t2, $a0, $t1
  lw    $t3, 0($t2)
  lw    $t4, 4($t2)
  slt   $t0, $t4, $t3
  beq   $t0, $zero, exit2

```

9 inst  
7 cyc

*if in-order*

.....

```

  addi  $s1, $s1, -1
  j     for2tst

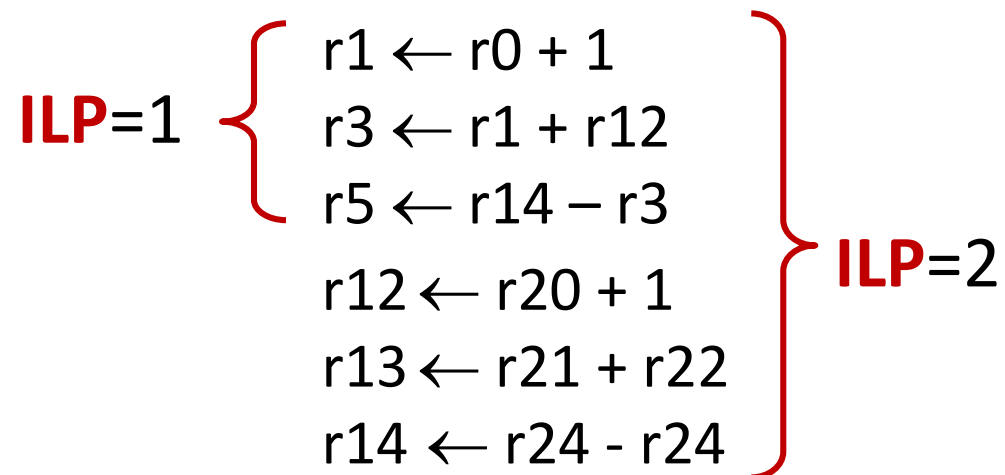
```

exit2:

*Assume all inst 1 cyc*

# Out-of-Order Execution for ILP

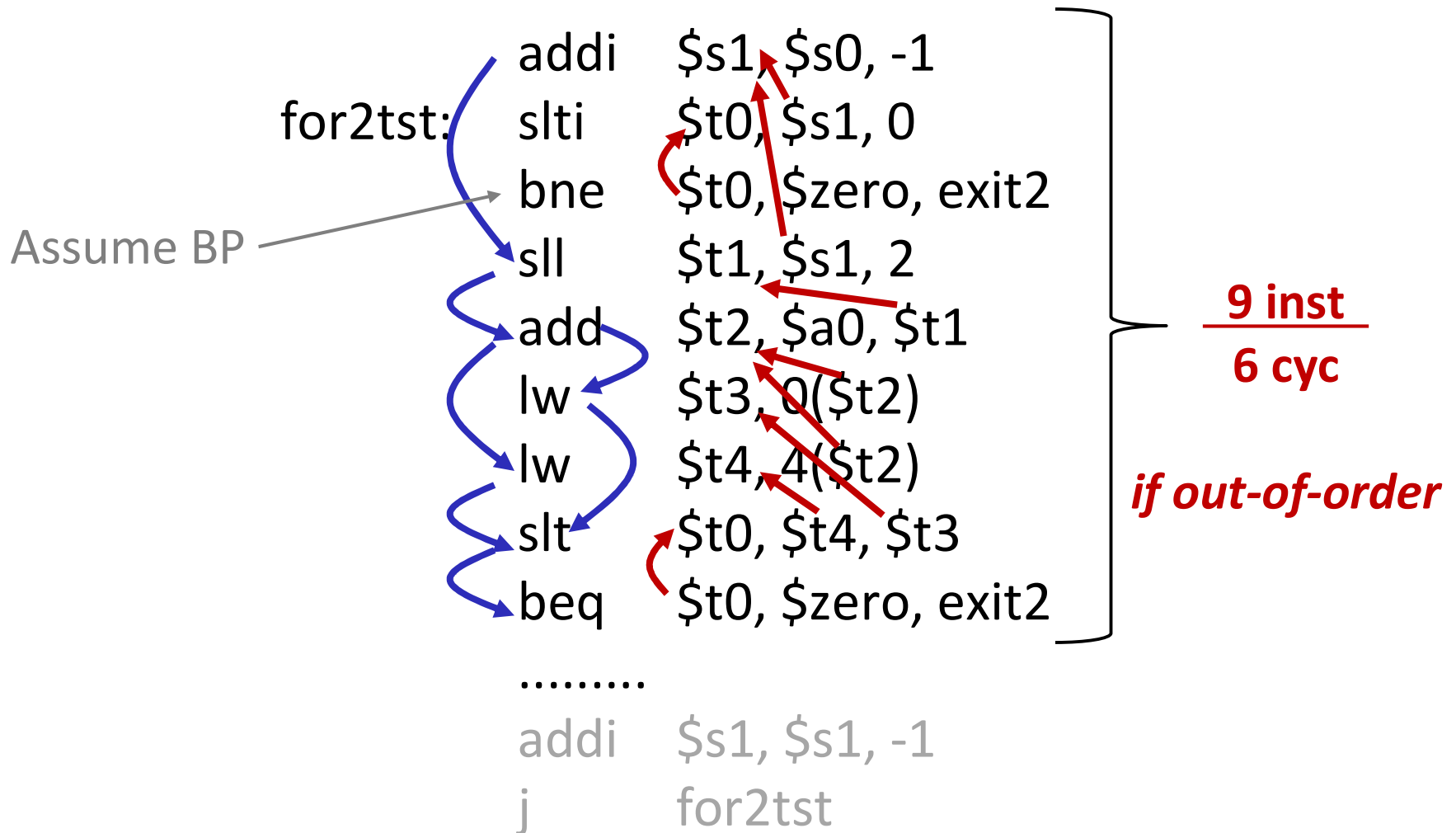
- **ILP** is scope dependent



Accessing **ILP=2** requires not only (1) larger scheduling window but also (2) out-of-order execution and (3) handling WAW and WAR

# Sample Assembly [P&H]

for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ..... }



exit2:

*Assume all inst 1 cyc*

# Pass this point not on exams

*For more, go read “Synthesis Lectures: Processor  
Microarchitecture: An Implementation Perspective,”  
2010*

# Add Out-of-Order to Superscalar to get more ILP

*How much more?*

*And What for?*

# 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?

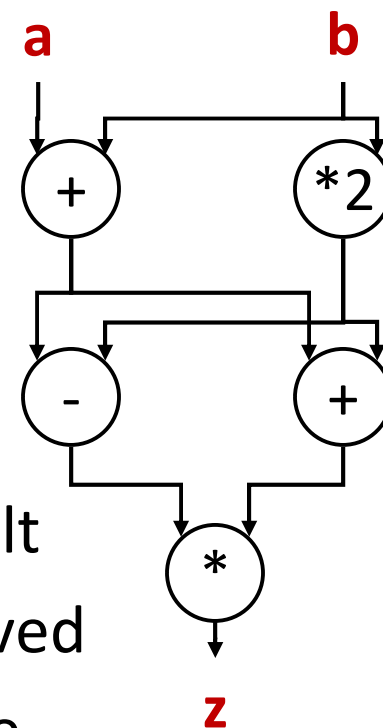
```

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



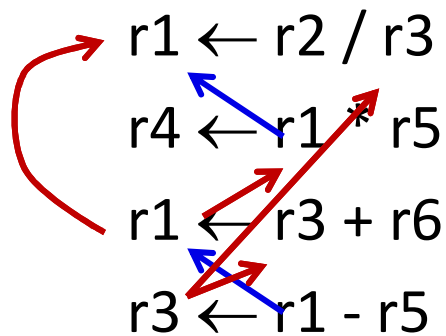
[dataflow figure and example from Arvind]

Recall

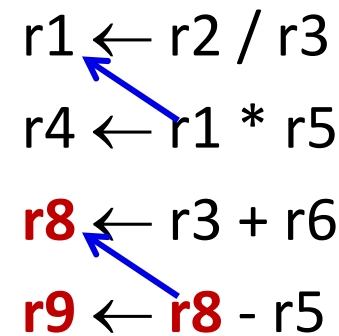


# Converting von Neumann to Dataflow: Register Renaming

Original



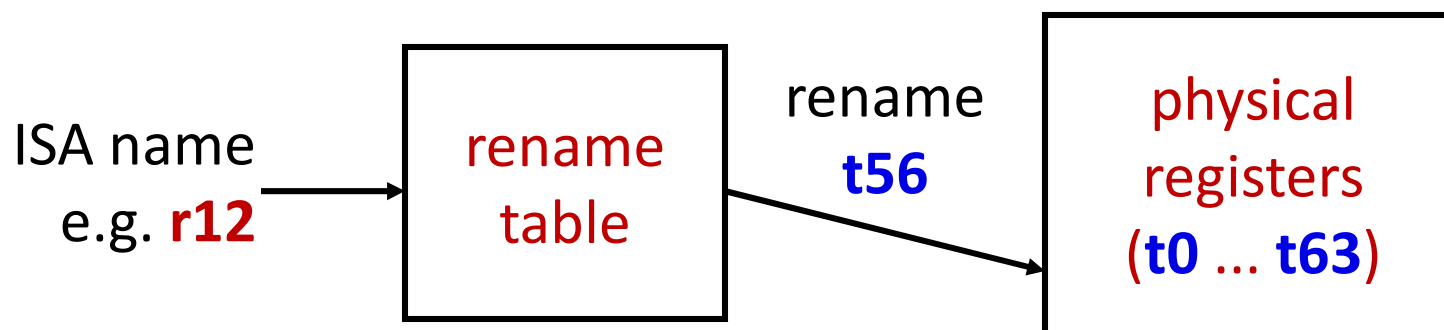
Renamed



- von Neumann model relies on finite **state** to convey producer-consumer dependence  
 $\Rightarrow$  must reuse state in correct **order**
- If infinite registers, use a new register for each new value  $\Rightarrow$  no WAW, no WAR, just dataflow left

*Can  $\mu$ arch fake infinite registers?*

# On-the-Fly HW Register Renaming



- Maintain mapping from ISA reg. names to physical registers
- When decoding an instruction that updates '**r<sub>x</sub>**':
  - allocate unused physical register **t<sub>y</sub>** to hold inst result
  - set new mapping from '**r<sub>x</sub>**' to **t<sub>y</sub>**
  - younger instructions using '**r<sub>x</sub>**' as input finds **t<sub>y</sub>**
- De-allocate a physical register for reuse when it is never needed again?

$$r1 \leftarrow r2 / r3$$

$$r4 \leftarrow r1 * r5$$

$$r1 \leftarrow r3 + r6$$

^^^^^when is this exactly?

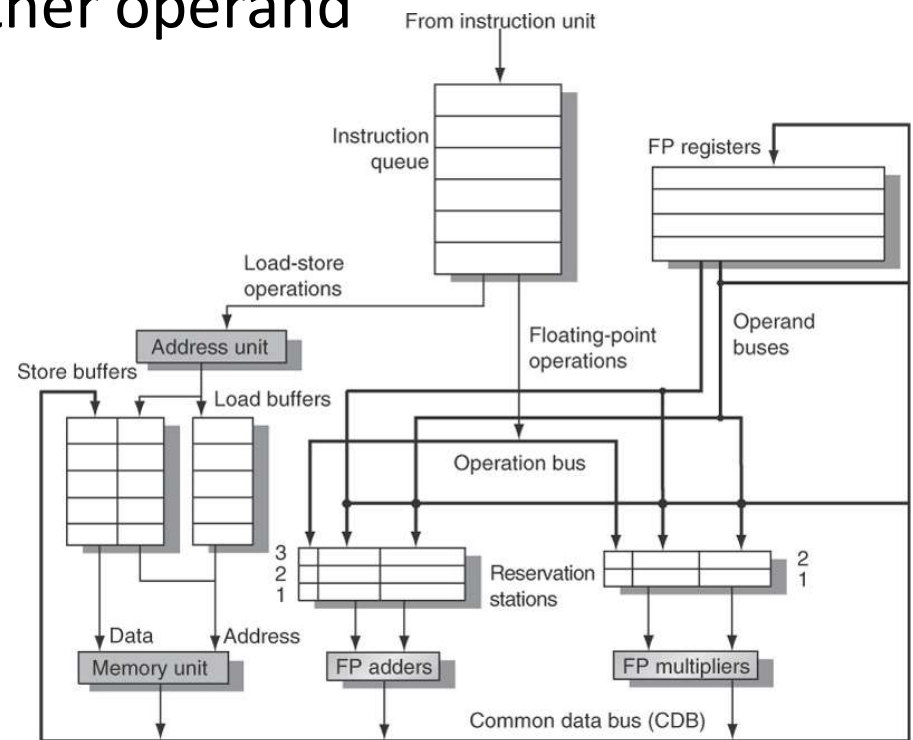
# Register Micro-Dataflow

- Maintain a buffer of many pending renamed instructions, a.k.a. reservation stations (**RSs**)
  - 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 (heuristic)
- A completing instruction (producer) signals dependent instructions (consumer) of operand availability

*Memory also out-of-order but much harder*

# 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



# Control Dependence also Limits ILP

- Average basic block size is only 6~7 instructions
- In-order pipelines used branch prediction to cover delay from fetch to branch resolution

*Expect 1 or less BP in flight in 5-stage pipeline*

- Out-of-order pipelines must speculate past many levels of BP to present a large out-of-order window of instructions!

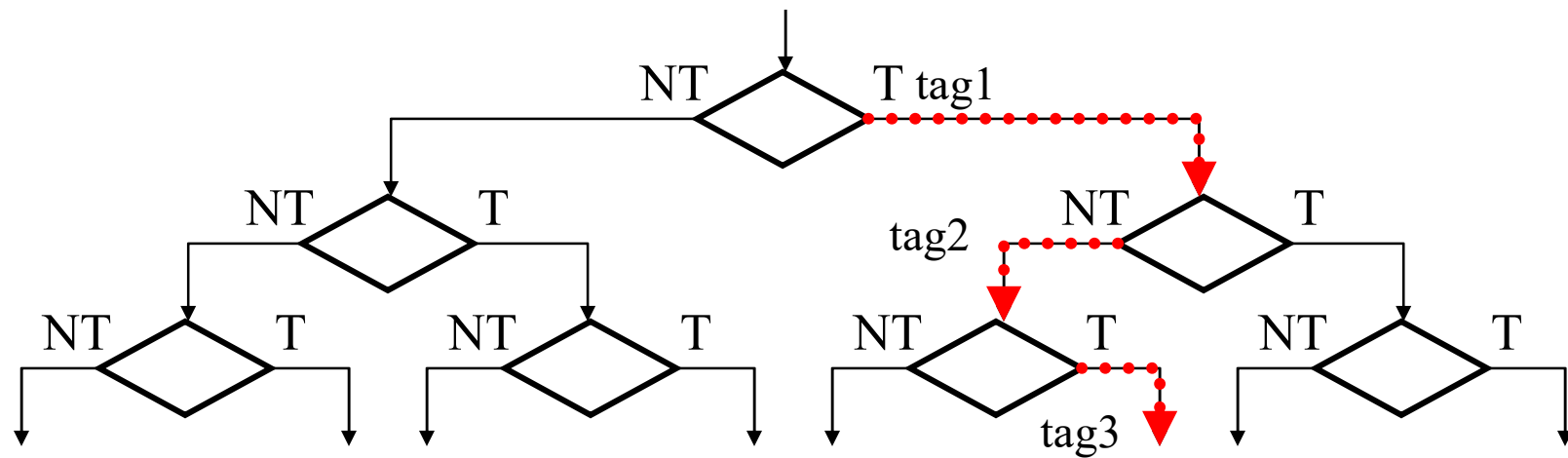
*Over 100 insts out-of-order in modern superscalar OOO CPUs*

**Add Speculation to Superscalar OOO  
to get still more ILP (??)**

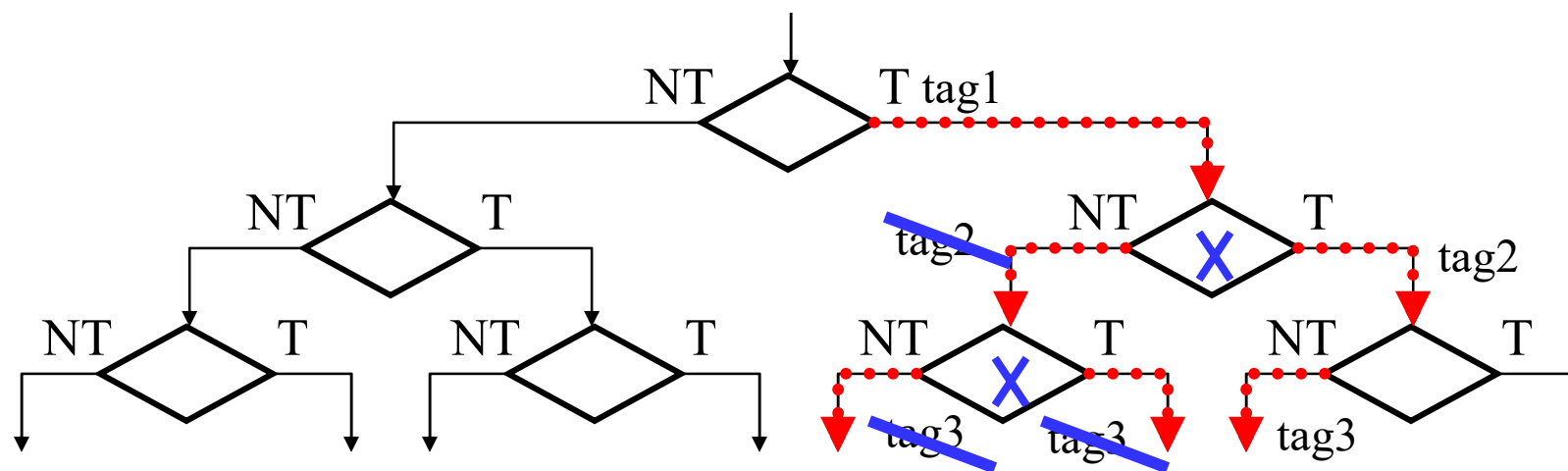
**how much more ILP can be uncovered?  
how much useful work will be done?**

**Ans: not much and not much**

# Nested Control Flow Speculation



# Mis-speculation Recovery can be Speculative



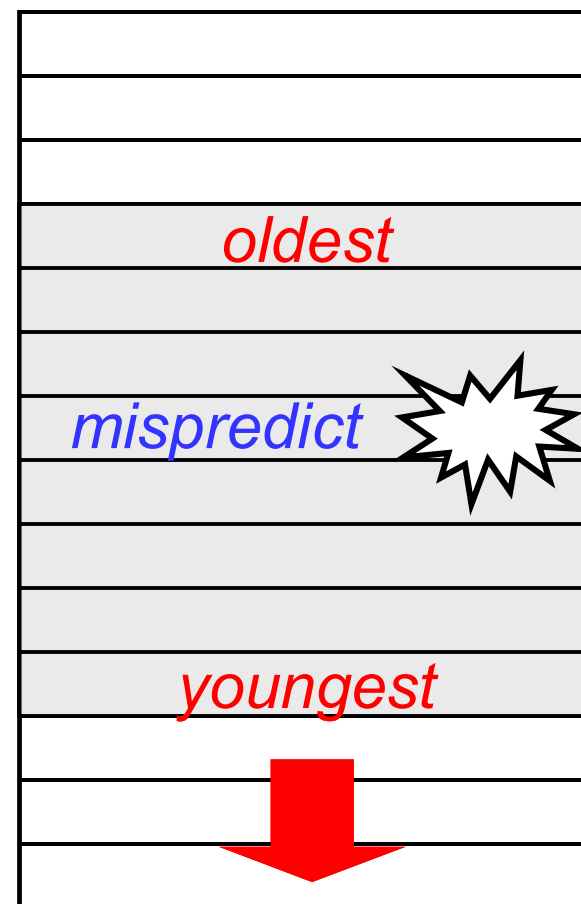


# Speculative Out-of-order Execution

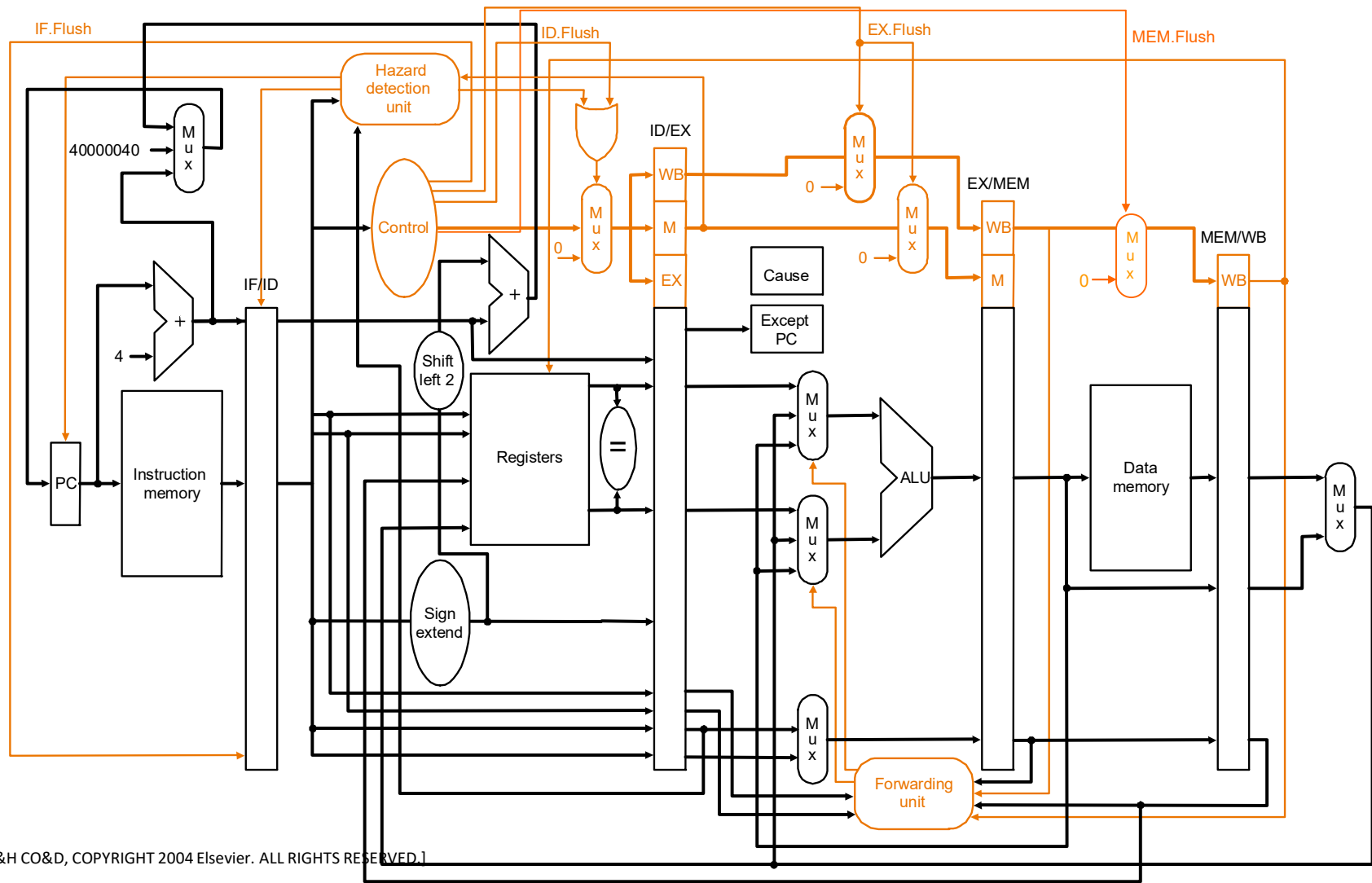
- A mispredicted branch after resolution must be rewound and restarted **ASAP!**
- Much trickier than 5-stage pipeline . . .
  - can rewind to an intermediate speculative state
  - a rewind branch could still be speculative and itself be discarded by another rewind!
  - rewind must reestablish both architectural 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



# You have seen something like this



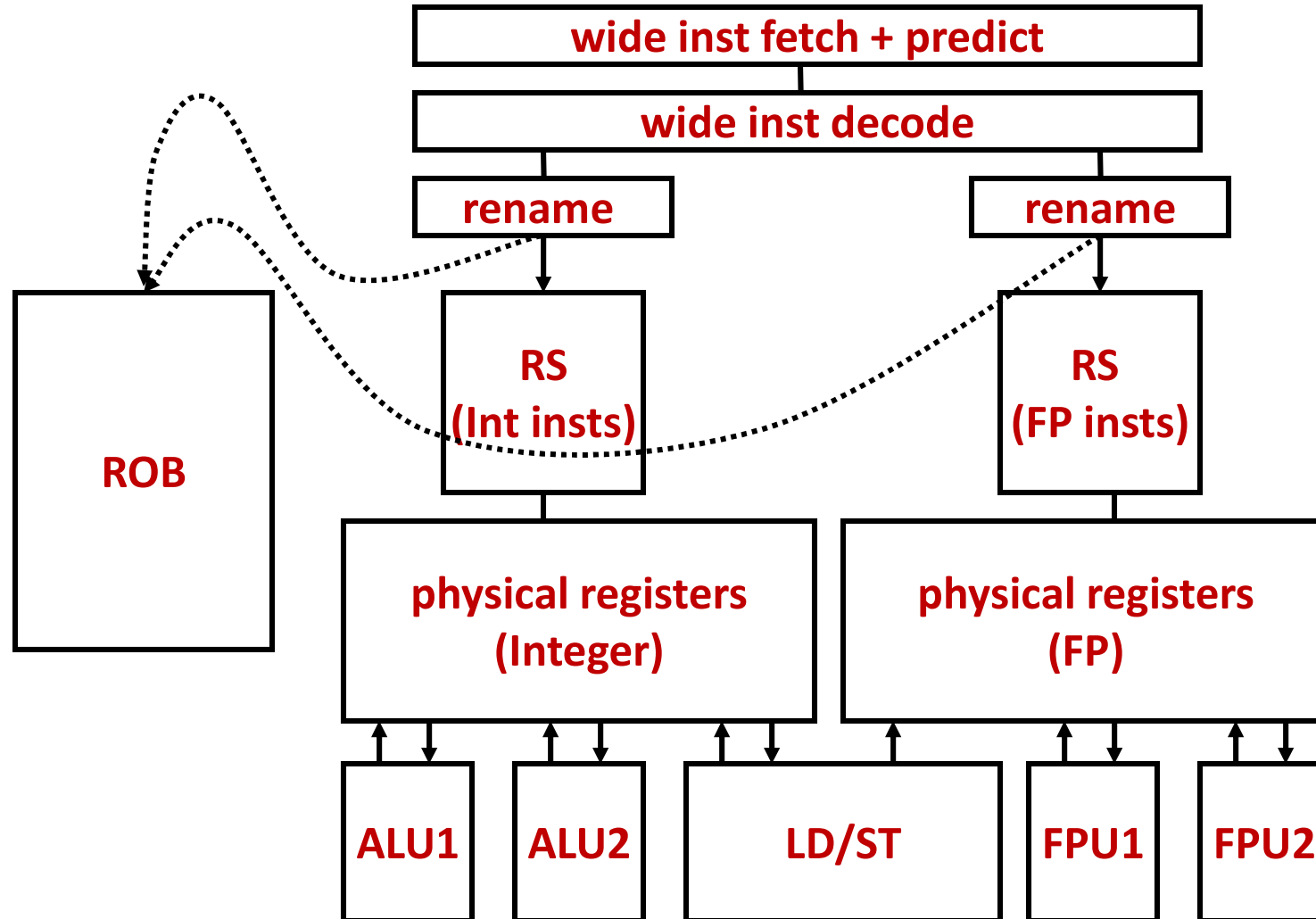
[P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Where is “the current instruction”?

# 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**

# 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 (aka MLP)

# 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>

# 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

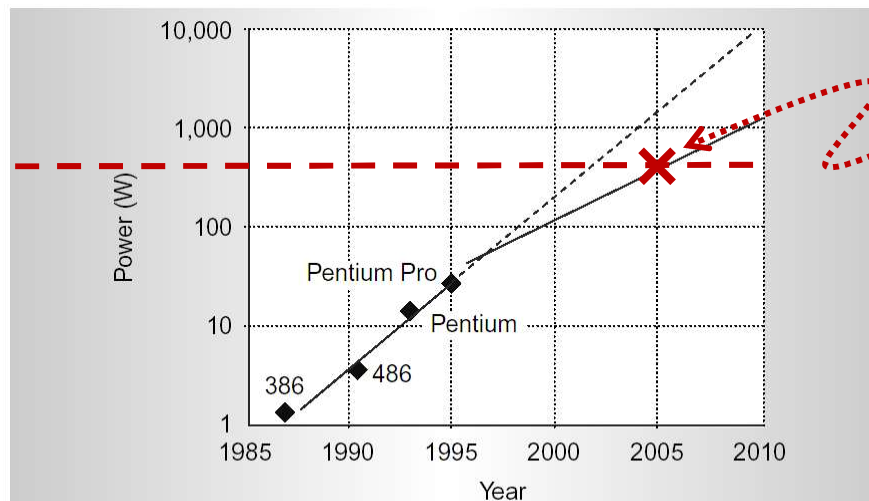


Figure 8. Power dissipation projections. [Borkar, IEEE Micro, July 1999]

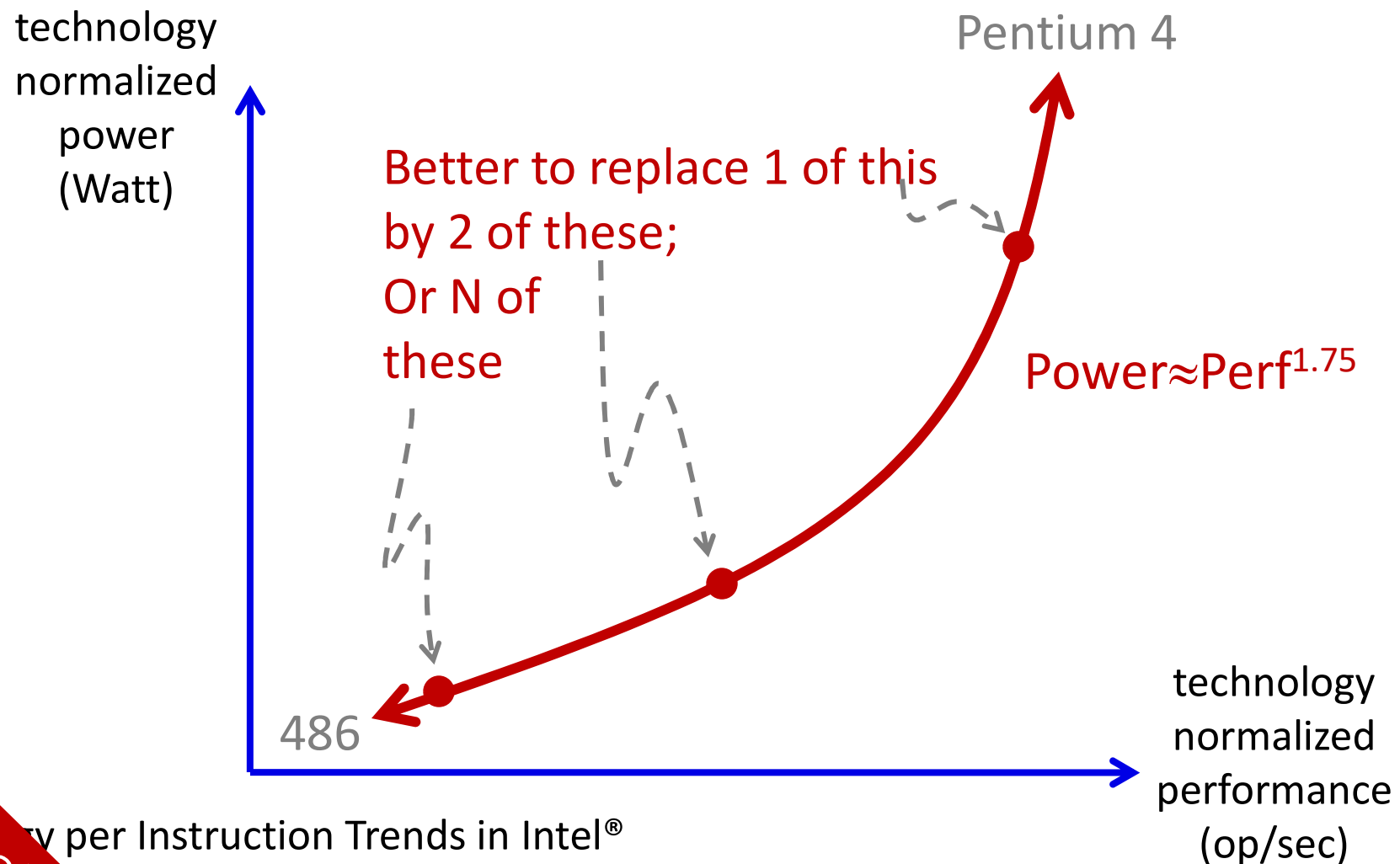
limit of  
economical  
cooling [ITRS]

2005, Intel  
P4 Tehas 150W

Recall



# Efficiency of Parallel Processing



Recall  
 Performance per 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 <sup>6</sup> )	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)

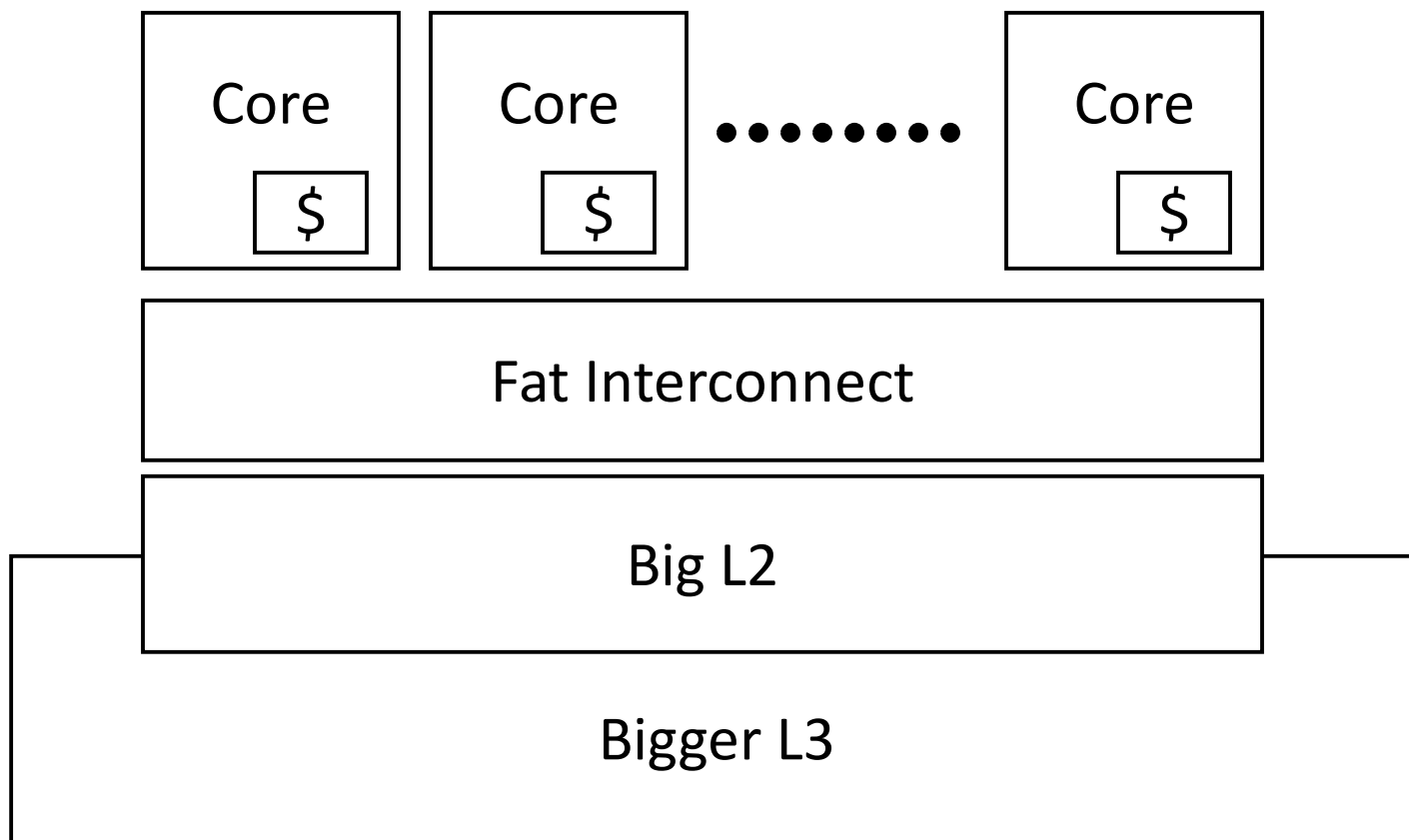
18-447-S24-L20-S34, James C. Hoe, CMU/ECE/CALCM, ©2024

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