Lecture 23
Introduction to Parallel Processing Hardware

Suggested reading:

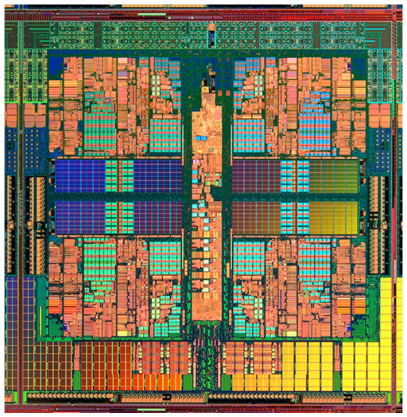
(HP Chapter 7 – over the next few weeks)

Important Dates

• Tuesday	11/15	Lab 5 out
• Thursday	11/17	Project summary due (email)
• Thursday	11/17	Lab 4 due
• Tuesday	11/22	HW 7 due
• Tuesday	11/22	HW 8 out
• Thursday	12/1	Lab 5 due
• Thursday	12/6	HW 8 due
• Thursday	12/8	Last day of class
• Monday	12/13	Final Exam (10:30-12:30)
• Thursday	12/15	Last day to turn in project

- Review session TBD

Multicore processors and programming

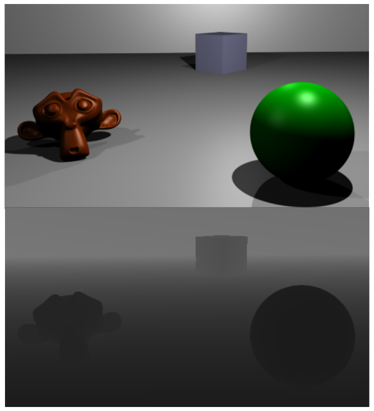


Processor components

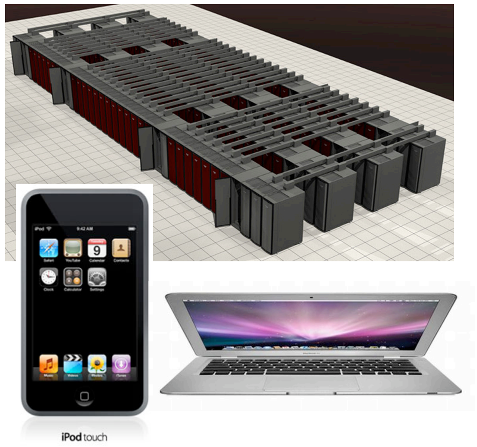
Processor comparison

AMD Athlon 64 vs. intel Pentium Dual-Core inside

Goal: Explain and articulate why modern microprocessors now have more than one core and how software must adapt to accommodate the now prevalent multi-core approach to computing.



Writing more efficient code



The right HW for the right application

```
for i=0; i<5; i++ {
       a = (a*b) + c;
}
```

```
MULT r1,r2,r3    # r1 ← r2*r3
ADD r2,r1,r4     # r2 ← r1+r4
```

110011	000001	000010	000011
001110	000010	000001	000100

HLL code translation

Fundamental lesson(s)

- This lecture will highlight:
  1. Different types of parallel systems
  2. What things count as "non-parallelizable" and (significantly) degrade parallel performance
  3. That parallel systems that once existed at the "room-level" now exist on-chip
     - Thus, "room-level" issues must now be dealt with "on-chip".

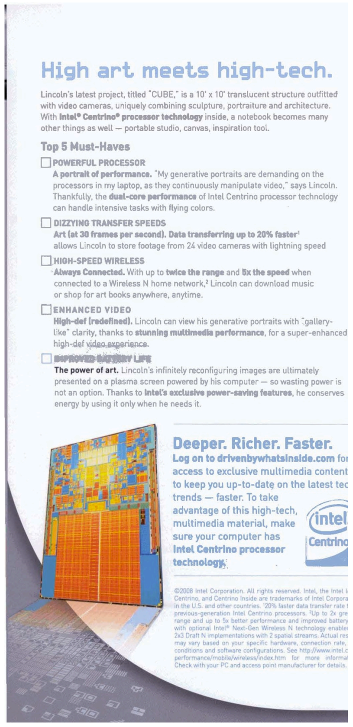
Why it’s important…

- It’s a multi-core world out there…
- Understanding what degrades performance is ESSENTIAL for writing efficient software.

INTRODUCTION

General context: Multiprocessors

- Multiprocessor is any computer with several processors



High art meets high-tech.

Lincoln's latest project, titled "CUBE," is a 10' x 10' translucent structure outfitted with video cameras, uniquely combining sculpture, portraiture and architecture. With **Intel® Centrino® processor technology** inside, a notebook becomes many other things as well — portable studio, canvas, inspiration tool.

Top 5 Must-Haves

☐ **POWERFUL PROCESSOR**
A portrait of performance. "My generative portraits are demanding on the processors in my laptop, as they continuously manipulate video," says Lincoln. Thankfully, the **dual-core performance** of Intel Centrino processor technology can handle intensive tasks with flying colors.

☐ **DIZZYING TRANSFER SPEEDS**
Art (at 30 frames per second). Data transferring up to 20% faster[1] allows Lincoln to store footage from 24 video cameras with lightning speed

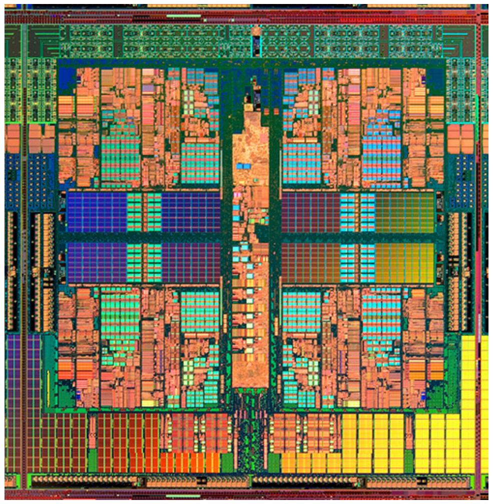
☐ **HIGH-SPEED WIRELESS**
Always Connected. With up to **twice the range** and **5x the speed** when connected to a Wireless N home network,[2] Lincoln can download music or shop for art books anywhere, anytime.

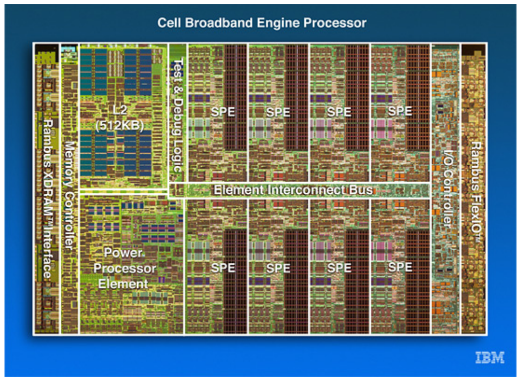
☐ **ENHANCED VIDEO**
High-def (redefined). Lincoln can view his generative portraits with "gallery-like" clarity, thanks to **stunning multimedia performance**, for a super-enhanced high-def video experience.

☐ **IMPROVED BATTERY LIFE**
The power of art. Lincoln's infinitely reconfiguring images are ultimately presented on a plasma screen powered by his computer — so wasting power is not an option. Thanks to **Intel's exclusive power-saving features**, he conserves energy by using it only when he needs it.

Deeper. Richer. Faster.
Log on to drivenbywhatsinside.com fo
access to exclusive multimedia content
to keep you up-to-date on the latest tec
trends — faster. To take advantage of this high-tech, multimedia material, make sure your computer has **Intel Centrino processor technology.**

©2008 Intel Corporation. All rights reserved. Intel, the Intel l
Centrino, and Centrino Inside are trademarks of Intel Corpora
in the U.S. and other countries. [1]20% faster data transfer rate t
previous-generation Intel Centrino processors. [2]Up to 2x gre
range and up to 5x better performance and improved battery
with optional Intel® Next-Gen Wireless N technology enable
2x3 Draft N implementations with 2 spatial streams. Actual res
may vary based on your specific hardware, connection rate,
conditions and software configurations. See http://www.intel.c
performance/mobile/wireless/index.htm for more informat
Check with your PC and access point manufacturer for details.



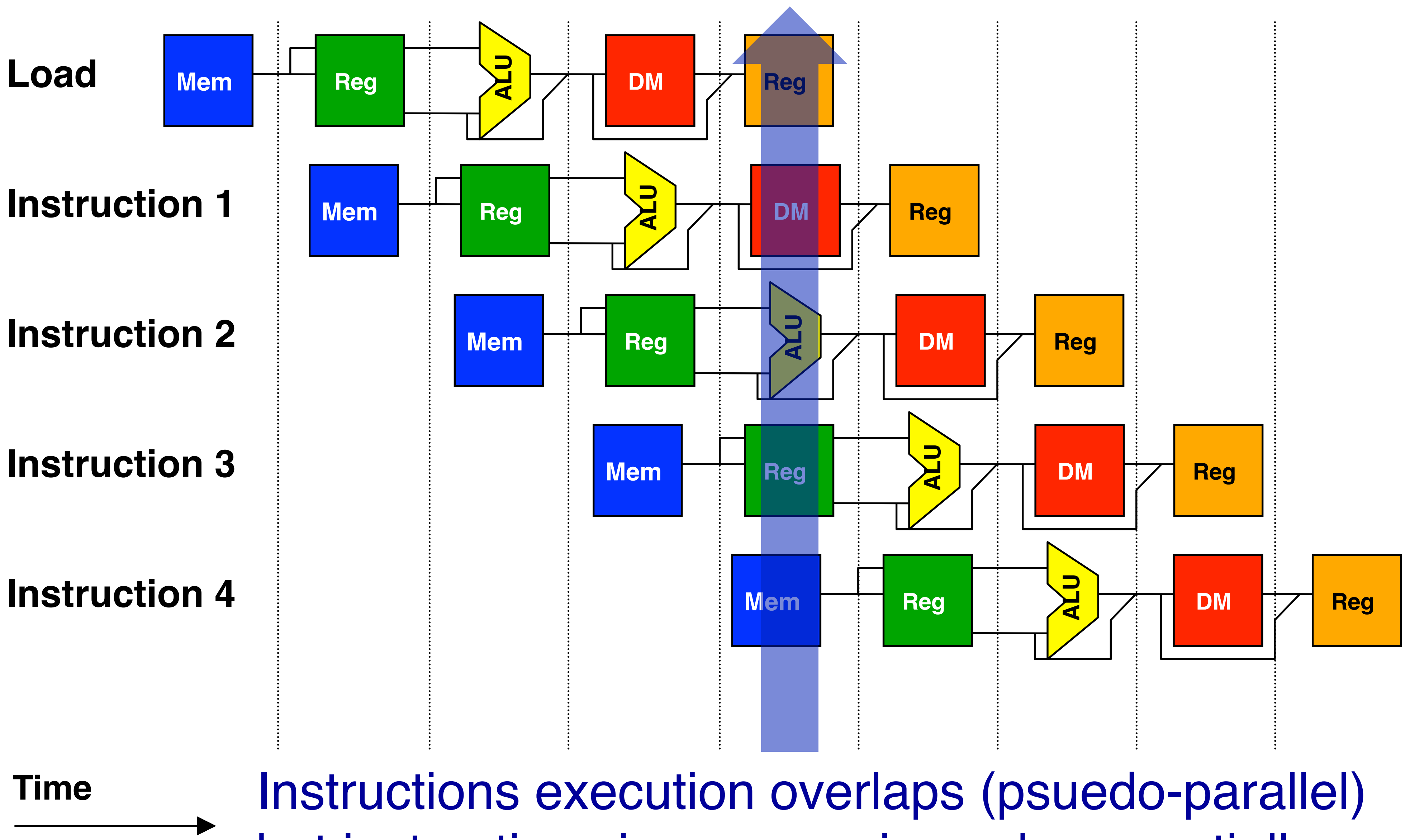


Lemieux cluster, Pittsburgh supercomputing center

Important idea:

- For a long time, "parallel processing" = connecting multiple chips
- It still does, but now done on single chip too; challenges of connecting multiple chips now exist "on-chip"

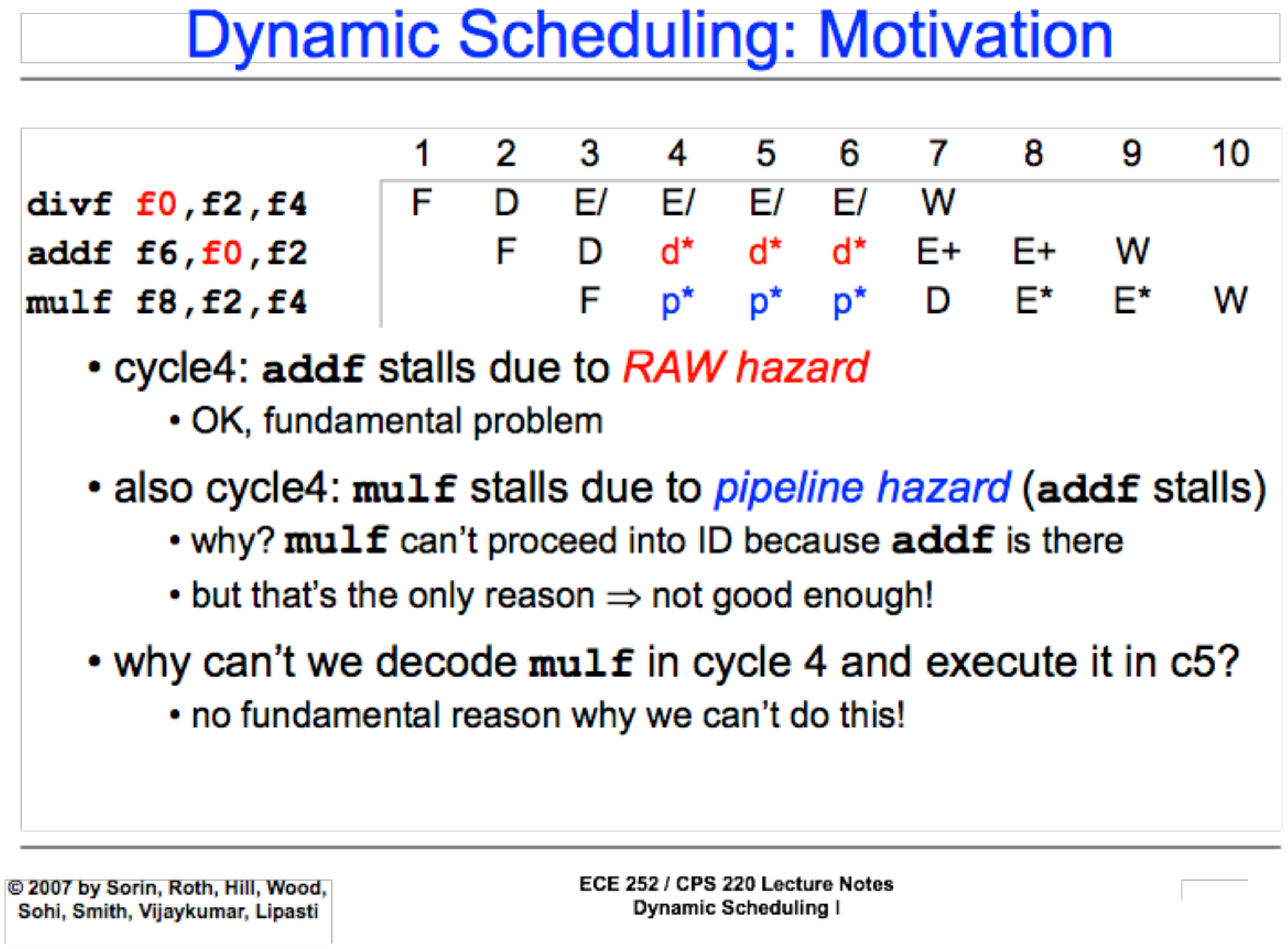
Pipelining and "Parallelism"



Instructions execution overlaps (psuedo-parallel) but instructions in program issued sequentially.

Important idea: We've already done some parallel processing!

What comes after pipelining?

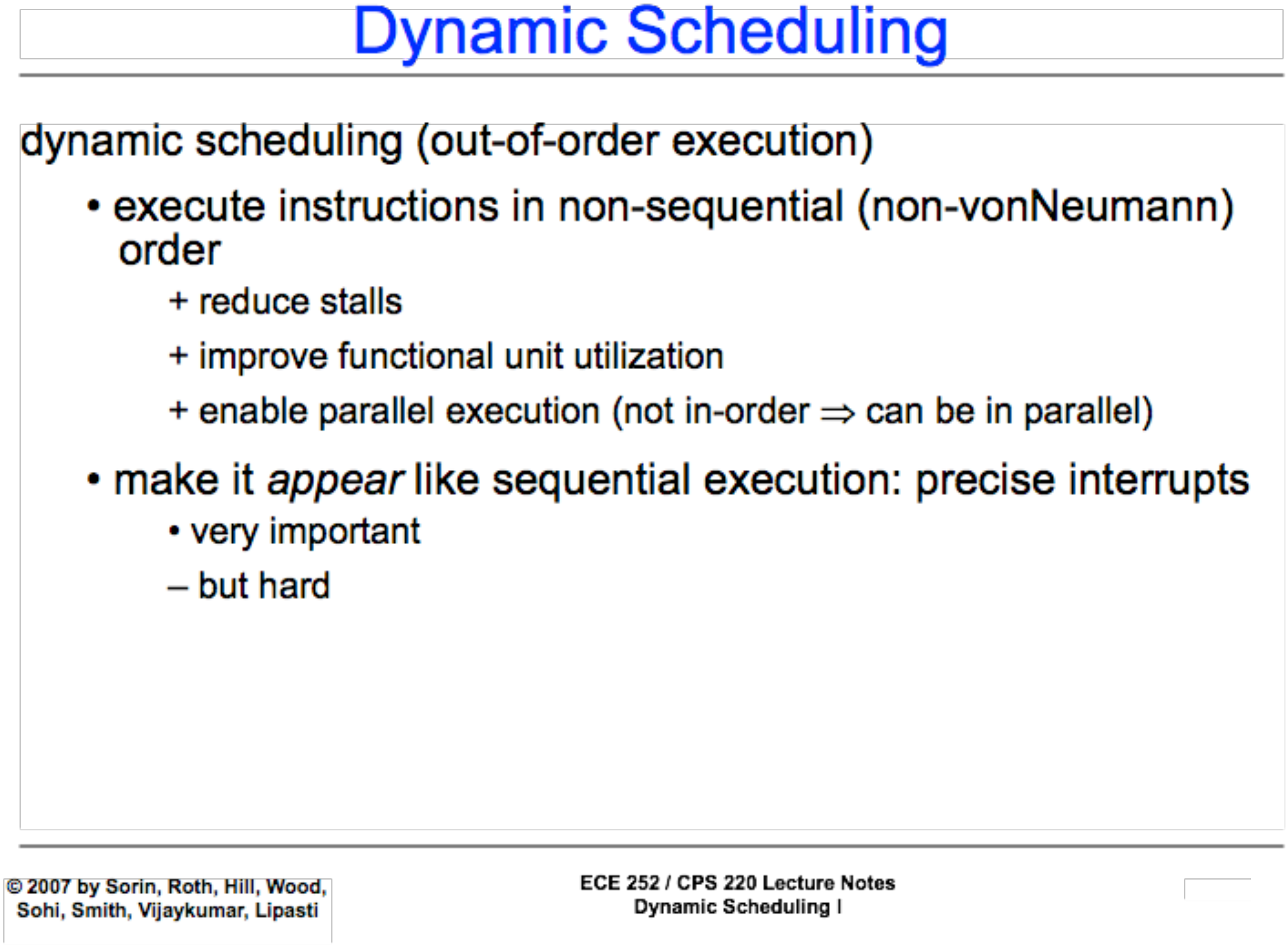


Dynamic Scheduling: Motivation

	1	2	3	4	5	6	7	8	9	10
`divf f0,f2,f4`	F	D	E/	E/	E/	E/	W			
`addf f6,f0,f2`		F	D	d*	d*	d*	E+	E+	W	
`mulf f8,f2,f4`			F	p*	p*	p*	D	E*	E*	W

- cycle4: **`addf`** stalls due to *RAW hazard*
  - OK, fundamental problem
- also cycle4: **`mulf`** stalls due to *pipeline hazard* (**`addf`** stalls)
  - why? **`mulf`** can't proceed into ID because **`addf`** is there
  - but that's the only reason ⇒ not good enough!
- why can't we decode **`mulf`** in cycle 4 and execute it in c5?
  - no fundamental reason why we can't do this!

What comes after pipelining?

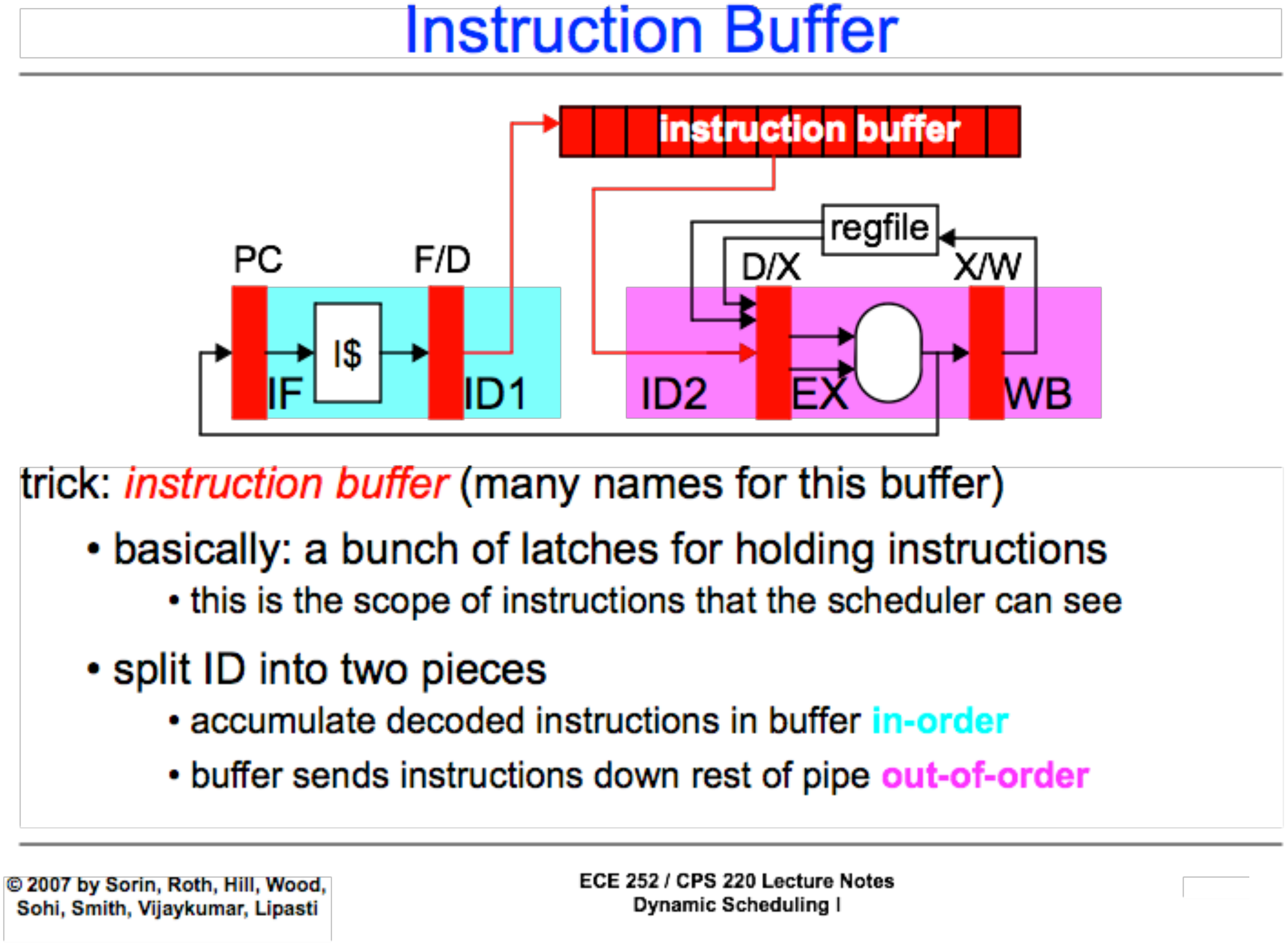


Dynamic Scheduling

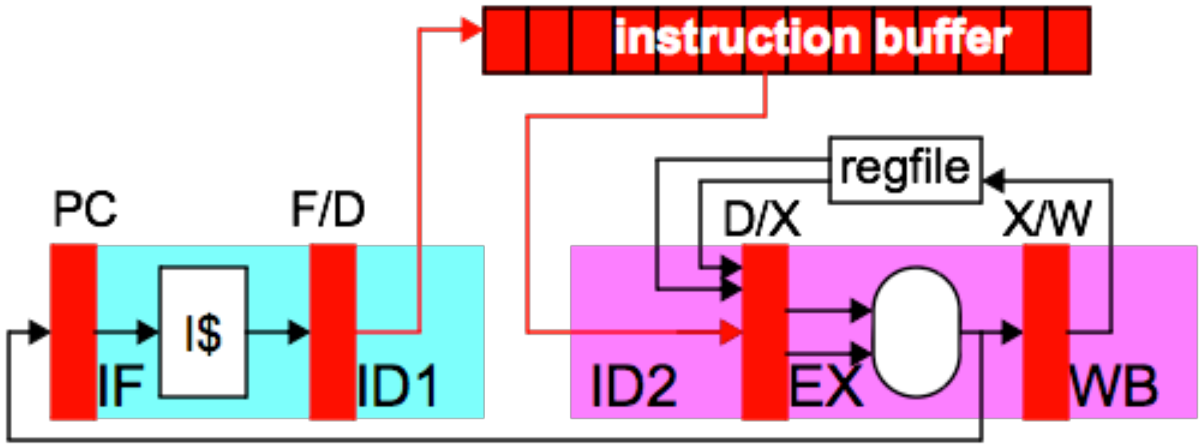
dynamic scheduling (out-of-order execution)

- execute instructions in non-sequential (non-vonNeumann) order
  - + reduce stalls
  - + improve functional unit utilization
  - + enable parallel execution (not in-order ⇒ can be in parallel)
- make it *appear* like sequential execution: precise interrupts
  - • very important
  - – but hard

Superscalar machines



Instruction Buffer

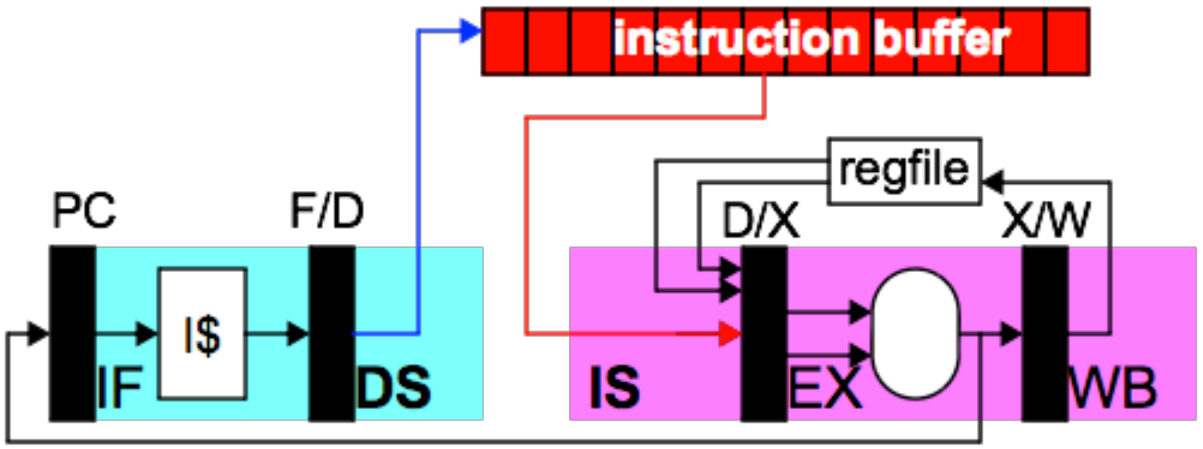


trick: *instruction buffer* (many names for this buffer)

- basically: a bunch of latches for holding instructions
  - this is the scope of instructions that the scheduler can see
- split ID into two pieces
  - accumulate decoded instructions in buffer **in-order**
  - buffer sends instructions down rest of pipe **out-of-order**

How Superscalar Machines Work

Dispatch and Issue



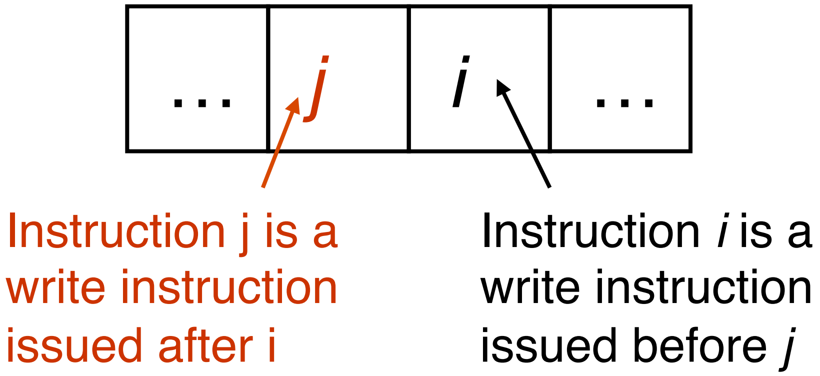
- dispatch (DS): first part of ID
  - allocate resources in instruction buffer
  - – new kind of structural hazard (instruction buffer could be full)
  - *dispatch is in-order, and <u>stall</u> propagates to younger instructions*
- issue (IS): second part of ID
  - send instructions from instruction buffer to execution units
  - *out-of-order, <u>wait</u> does NOT propagate to younger instructions*

Superscalar Introduces New Hazards

- **With WAW hazard, instruction *j* tries to write an operand before instruction *i* writes it.**

- **The writes are performed in wrong order leaving the value written by earlier instruction**

- **Graphically/Example:**

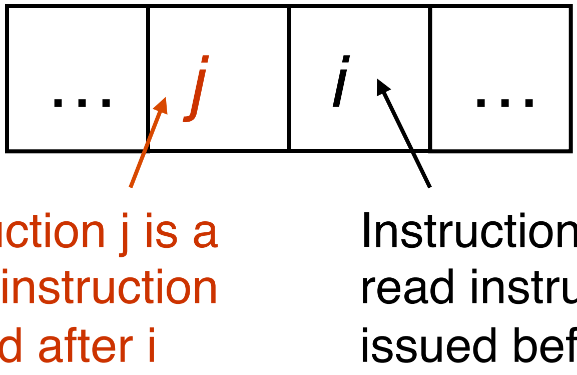

…
j
i
…
Instruction j is a write instruction issued after i
Instruction *i* is a write instruction issued before *j*

i: DIV F1, F2, F3
j: SUB F1, F4, F6

Problem: If *j* after *i*, and *j* finishes first, and then *i* writes to F1, the wrong value is in F1.

Superscalar Introduces New Hazards

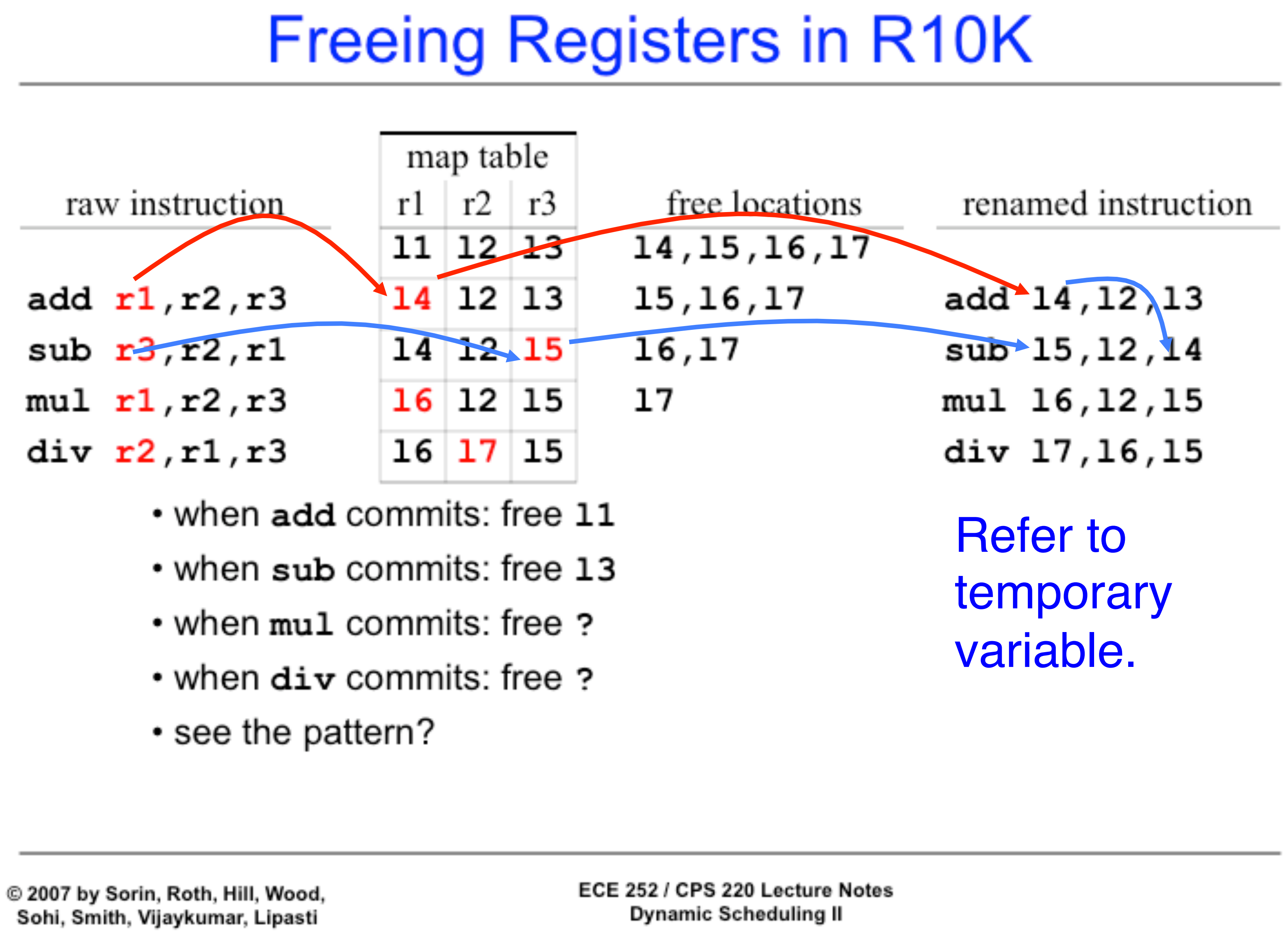
- **With WAR hazard, instruction *j* tries to write an operand before instruction *i* reads it.**
- **Instruction *i* would incorrectly receive newer value of its operand;**
  - **Instead of getting old value, it could receive some newer, undesired value:**
- **Graphically/Example:**


…
j
i
…
Instruction j is a write instruction issued after i
Instruction i is a read instruction issued before j

i: DIV F7, F1, F3
j: SUB F1, F4, F6

Problem: what if instruction *j* completes before instruction *i* reads F1?

Solution: Register Renaming



Freeing Registers in R10K

raw instruction	map table r1	map table r2	map table r3	free locations	renamed instruction
	11	12	13	14,15,16,17	
add r1,r2,r3	14	12	13	15,16,17	add 14,12,13
sub r3,r2,r1	14	12	15	16,17	sub 15,12,14
mul r1,r2,r3	16	12	15	17	mul 16,12,15
div r2,r1,r3	16	17	15		div 17,16,15

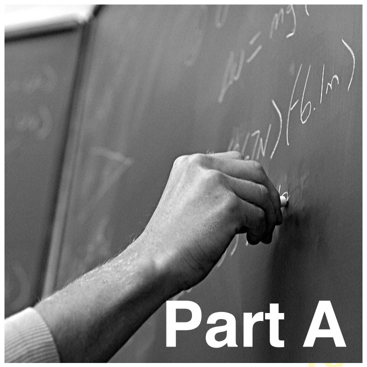
- when **add** commits: free **11**
- when **sub** commits: free **13**
- when **mul** commits: free **?**
- when **div** commits: free **?**
- see the pattern?

Refer to temporary variable.

© 2007 by Sorin, Roth, Hill, Wood, Sohi, Smith, Vijaykumar, Lipasti

ECE 252 / CPS 220 Lecture Notes
Dynamic Scheduling II

Board Example: Superscalar Trace



CLASSIFYING PARALLEL MACHINES

Multiprocessing (Parallel) Machines

- Flynn's Taxonomy of Parallel Machines
  - How many Instruction streams?
  - How many Data streams?
- SISD: Single I Stream, Single D Stream
  - A uni-processor
  - Could have psuedo-parallel execution here
    - (These are the techniques we just talked about/reviewed…)
  - Example: Intel Pentium 4
- SIMD: Single I, Multiple D Streams
  - Each "processor" works on its own data
  - But all execute the same instructions in lockstep
  - Example: SSE instructions in Intel x86
    - (e.g. for vector addition in graphics processing)

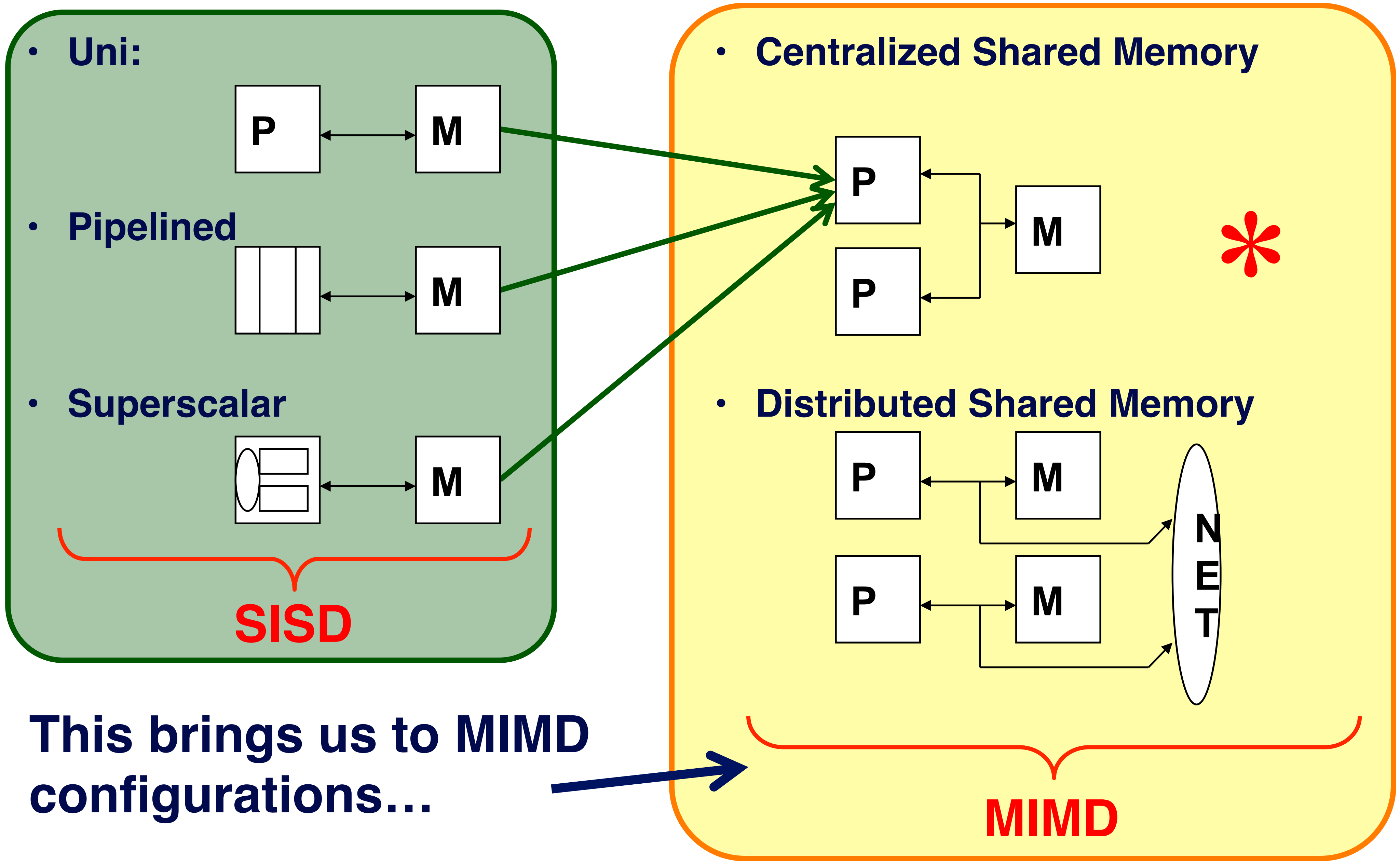
Flynn’s Taxonomy

- MISD: Multiple I, Single D Stream
  - Not used much, no example today
- MIMD: Multiple I, Multiple D Streams
  - Each processor executes its own instructions and operates on its own data
  - Pre multi-core, typical off-the-shelf multiprocessor (made using a bunch of “normal” processors)
    - Each node could also be superscalar
    - Historical lessons, challenges now apply to multi-core too!
  - Example: Intel Xeon e5345

MIMD MACHINE TYPES

What’s Next?

Processors can be any of these configurations



This brings us to MIMD configurations…

Multiprocessors

- Why did/do we need multiprocessors?
  - Uni-processor speed improved fast
  - But there are problems that needed even more speed
    - Before: Wait for a few years for Moore's law to catch up?
      - Or use multiple processors and do it sooner?
    - Now: Moore's Law still catching up.

- Multiprocessor software problem
  - Most code is sequential (for uni-processors)
    - MUCH easier to write and debug
  - Correct parallel code very, very difficult to write
    - Efficient and correct is much more difficult
    - Debugging even more difficult

Let's look more at example MIMD configurations…

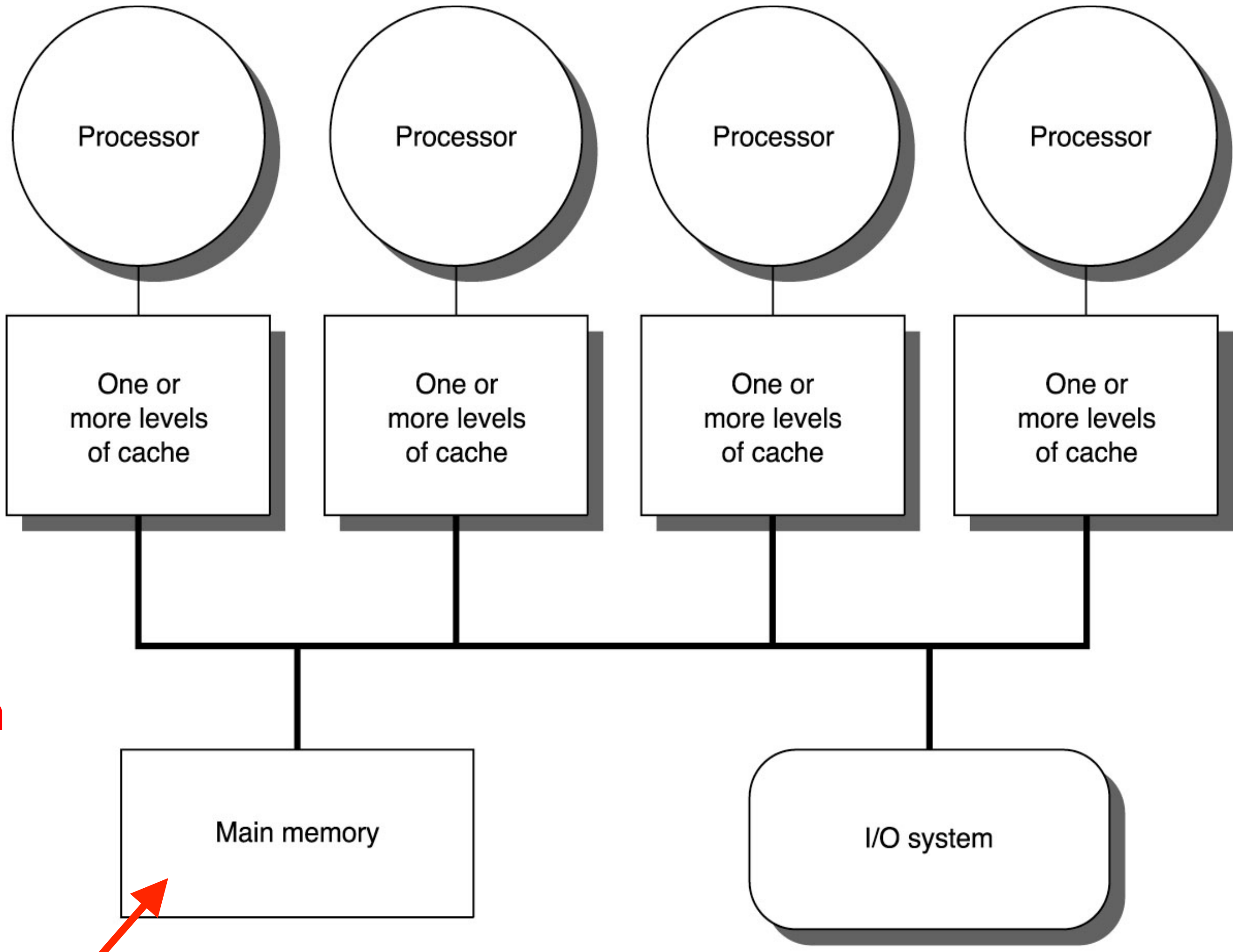
Multiprocessor Memory Types

- **Shared Memory:**
  - In this model, there is one (large) common shared memory for all processors
- **Distributed memory:**
  - In this model, each processor has its own (small) local memory, and its content is not replicated anywhere else

But processors can (and do) cache data from memory (wherever that may be).

MIMD Multiprocessors

Centralized Shared Memory

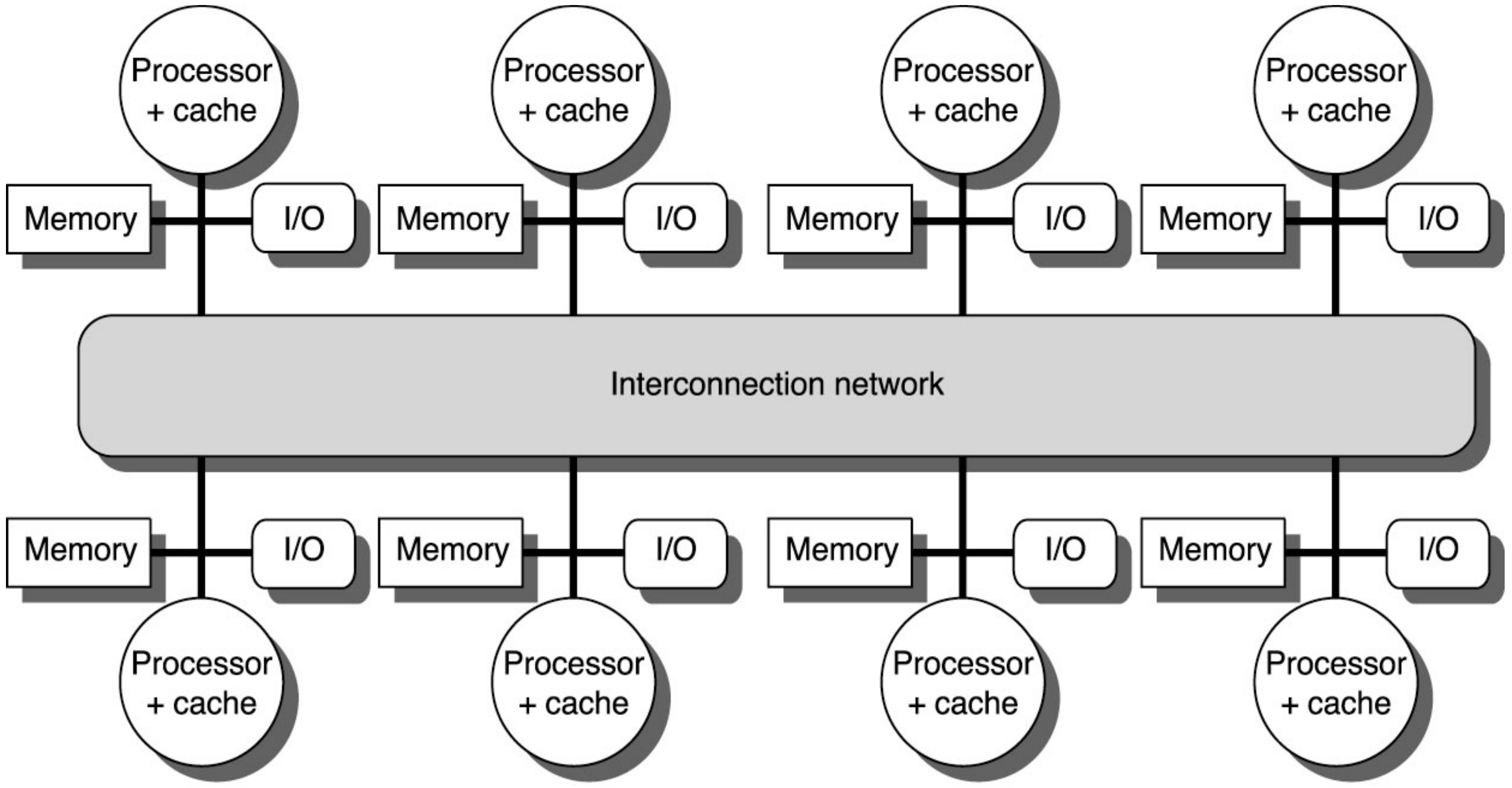

Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O system

Actually, quite representative of quad core chip, with "main memory" being a shared, L2 cache.

Note: just 1 memory

MIMD Multiprocessors

Distributed Memory



Multiple, distributed memories here.

More on Centralized-Memory Machines

- “Uniform Memory Access” (UMA)
  - All memory locations have similar latencies
  - Data sharing through memory reads/writes
  - P1 can write data to a physical address A,
    P2 can then read physical address A to get that data

- Problem: Memory Contention
  - All processor share the one memory
  - Memory bandwidth becomes bottleneck
  - Used only for smaller machines
    - Most often 2,4, or 8 processors
    - Up to 16-32 processors

More on Distributed-Memory Machines

- Two kinds
  - Distributed Shared-Memory (DSM)
    - All processors can address all memory locations
    - Data sharing possible
    - Also called NUMA (non-uniform memory access)
    - Latencies of different memory locations can differ
      - (local access faster than remote access)
  - Message-Passing
    - A processor can directly address only local memory
    - To communicate with other processors, must explicitly send/receive messages
- Most accesses local, so less memory contention (can scale to well over 1000s of processors)

Message-Passing Machines

- A cluster of computers
  - Each with its own processor and memory
  - An interconnect to pass messages between them
  - Producer-Consumer Scenario:
    - P1 produces data D, uses a SEND to send it to P2
    - The network routes the message to P2
    - P2 then calls a RECEIVE to get the message
  - Two types of send primitives
    - Synchronous: P1 stops until P2 confirms receipt of message
    - Asynchronous: P1 sends its message and continues
  - Standard libraries for message passing:
    Most common is MPI – Message Passing Interface

Message Passing Pros and Cons

- Pros
  - Simpler and cheaper hardware
  - Explicit communication makes programmers aware of costly (communication) operations

- Cons
  - Explicit communication is painful to program
  - Requires manual optimization
    - If you want a variable to be local and accessible via LD/ST, you must declare it as such
    - If other processes need to read or write this variable, you must explicitly code the needed sends and receives to do this

Message Passing: A Program

- **Calculating the sum of array elements**

```
#define ASIZE 1024
#define NUMPROC 4
double myArray[ASIZE/NUMPROC];
double mySum=0;
for(int i=0;i<ASIZE/NUMPROC;i++)
  mySum+=myArray[i];
if(myPID=0){
  for(int p=1;p<NUMPROC;p++){
    int pSum;
    recv(p,pSum);
    mySum+=pSum;
  }
  printf("Sum: %lf\n",mySum);
}else
  send(0,mySum);
```

Must manually split the array

"Main" processor adds up partial sums and prints the result

Other processors send their partial results to main

Shared Memory Pros and Cons

- Pros
  - Communication happens automatically
  - More natural way of programming
    - Easier to write correct programs and gradually optimize them
  - No need to manually distribute data
    (but can help if you do)

- Cons
  - Needs more hardware support
  - Easy to write correct, but inefficient programs
    (remote accesses look the same as local ones)

More on this in the next few days.
(i.e. where inefficiencies come from…)

Shared Memory: A Program

- **Calculating the sum of array elements**

```
#define ASIZE 1024
#define NUMPROC 4
shared double array[ASIZE];
shared double allSum=0;
shared mutex sumLock;
double mySum=0;
for(int i=myPID*ASIZE/NUMPROC;i<(myPID+1)*ASIZE/NUMPROC;i++)
  mySum+=array[i];
lock(sumLock);
allSum+=mySum;
unlock(sumLock);
if(myPID=0)
  printf("Sum: %lf\n",allSum);
```

Array is shared

1+(1024/4)=257

Each processor sums up "its" part of the array

i<(2*256); i<512

Each processor adds its partial sums to the final result

Note need for synchronization support

Main processor prints the result

THE CHALLENGE(S) OF PARALLEL COMPUTING

Real Issues and Problems

- Motivation is 2 fold
  - 50s ,60s, 70s, … today – have rooms filled with machines working to solve a common problem
  - Could consider multi-core chips as "CSM"
    - i.e. 4-core chip, each core has L1 cache, share L2 cache

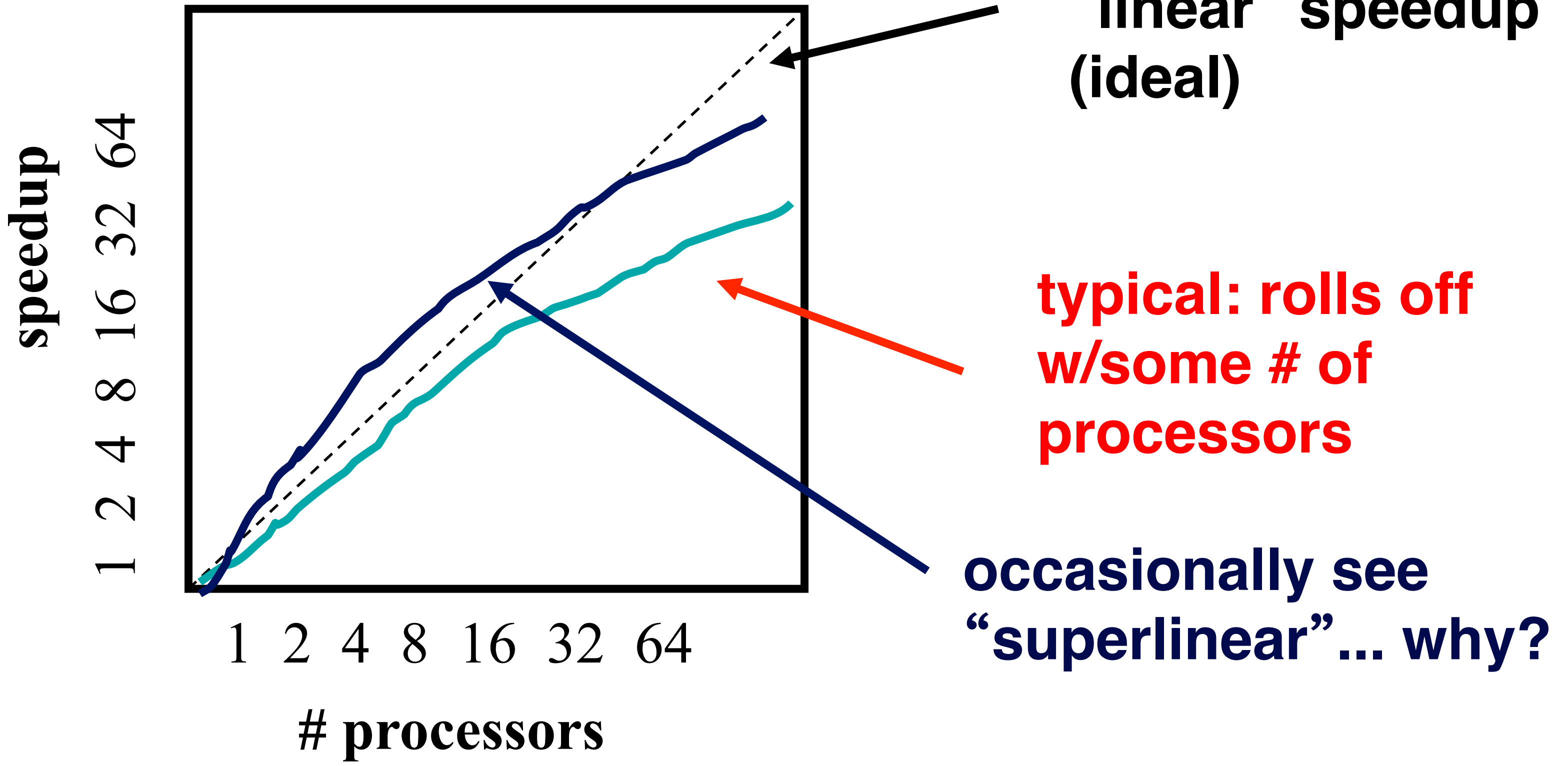
- Problems are often the same, just on different scales

- As technology scales, problems once seen at room level can be found "on chip"

Speedup

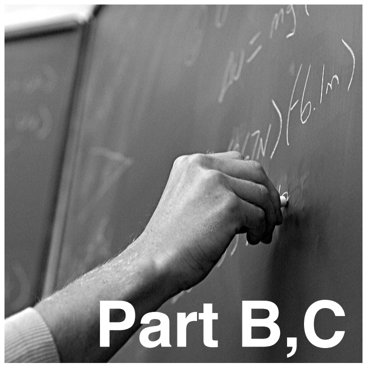
metric for performance on latency-sensitive applications

- Time(1) / Time(P) for P processors
  - note: must use the best ***sequential*** algorithm for Time(1); the parallel algorithm may be different.



Parallel Programming

- Parallel software is the problem
- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it's easier!
- Difficulties
  - Partitioning
  - Coordination
  - Communications overhead



Challenges: Cache Coherency

- Shared memory easy with no caches
  - P1 writes, P2 can read
  - Only one copy of data exists (in memory)

- Caches store their own copies of the data
  - Those copies can easily get inconsistent
  - Classical example: adding to a sum
    - P1 loads allSum, adds its mySum, stores new allSum
    - P1's cache now has dirty data, but memory not updated
    - P2 loads allSum from memory, adds its mySum, stores allSum
    - P2's cache also has dirty data
    - Eventually P1 and P2's cached data will go to memory
    - Regardless of write-back order, final value ends up wrong

If moderate # of nodes, write-through not practical.

Challenges: Contention

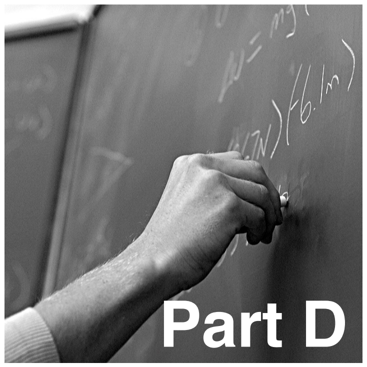
- Contention for access to shared resources - esp. memory banks or remote elements - may dominate overall system scalability
  - The problem:
    - Neither network technology nor chip memory bandwidth has grown at the same rate as the processor execution rate or data access demands

Challenges: Latency

- …is already a major source of performance degradation
  - Architecture charged with hiding local latency
    - (that's why we talked about registers, caches, IC, etc.)
  - Hiding global latency is task of programmer
    - (I.e. manual resource allocation)
- Today:
  - multiple clock cycles to cross chip
  - access to DRAM in 100s of CCs
  - round trip remote access in 1000s of CCs
- In spite of progress in NW technology, *increases* in clock rate may cause delays to reach 1,000,000s of CCs in worst cases

Challenges: Reliability

- **Reliability:**
  - **Improving yield and achieving high "up time"**
    - **(think about how performance might suffer if one of the 1 million nodes fails every x number of seconds or minutes…)**
  - **Solve with checkpointing, other techniques**



Challenges: Languages

- Programming languages, environments, & methodologies:
  - Need simple semantics and syntax that can also expose computational properties to be exploited by large-scale architectures

Impediments to Parallel Performance

- ★ Reliability:
  - Want to achieve high "up time" – especially in non-CMPs
- ★ Contention for access to shared resources
  - i.e. multiple accesses to limited # of memory banks may dominate system scalability
- Programming languages, environments, & methods:
  - Need simple semantics that can expose computational properties to be exploited by large-scale architectures
- Algorithms

Not all problems are parallelizable

$$\text{Speedup} = \frac{1}{\left[1 - \text{Fraction}_{\text{parallelizable}}\right] + \dfrac{\text{Fraction}_{\text{parallelizable}}}{N}}$$

What if you write good code for 4-core chip and then get an 8-core chip?

- ★ Cache coherency
  - P1 writes, P2 can read
    - Protocols can enable $ coherency but add overhead
- ★ Overhead where no actual processing is done.

Challenges: Latency ★

- …is already a major source of performance degradation
  - Architecture charged with hiding local latency
    - (that’s why we talked about registers & caches)
  - Hiding global latency is also task of programmer
    - (I.e. manual resource allocation)

★ Overhead where no actual processing is done.

Impediments to Parallel Performance

- All ★'ed items also affect speedup that could be obtained…

$$\text{Speedup} = \frac{1}{\left[1 - \text{Fraction}_{\text{parallelizable}}\right] + \dfrac{\text{Fraction}_{\text{parallelizable}}}{N}}$$