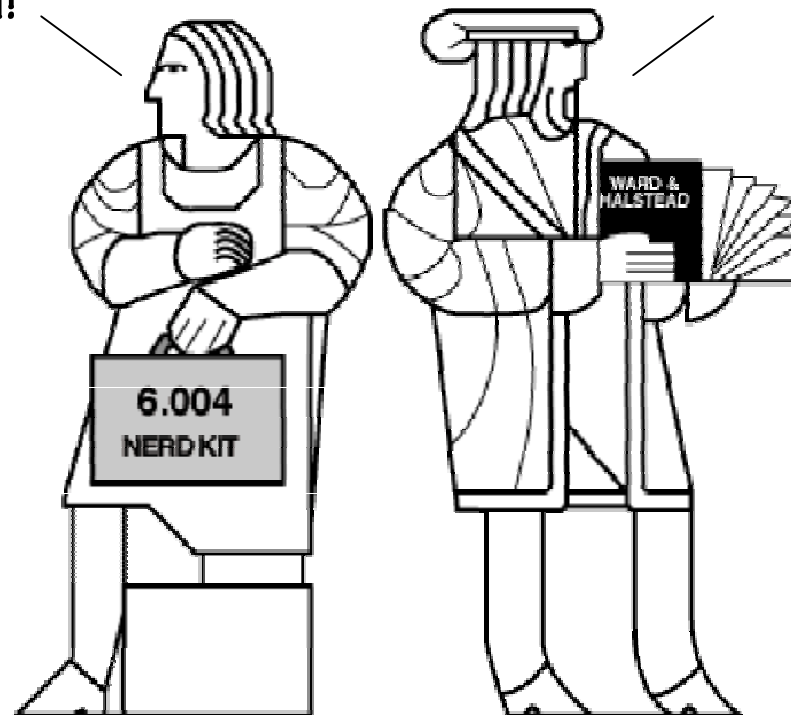


# Parallel Processing

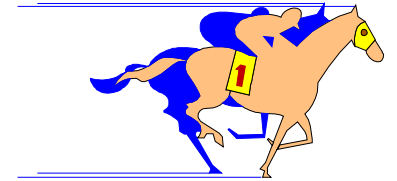
*I've gotta spend at least  
20 hours studying for  
the 6.004 final!*

*I'll get 20 friends to help...  
... we'll be done in an hour.*



**Handouts: Lecture Slides**

# The Home Stretch



All labs **MUST** be checked-off in by Friday (5/18).

**ALL LABS MUST BE COMPLETED TO PASS 6.004!**

THURSDAY 5/17 - The Future of Computers...

THURSDAY 5/17 at 4:00 - Beta Design Contest (34-501)  
Cool Prizes, Fame, and a cure for lab withdrawal pains

FRIDAY 5/18 sections - Final Quiz Review

TUESDAY 5/22 - **FINAL EXAM (1:30 – 4:30, Rockwell)**

TUESDAY EVENING -

*Immense Satisfaction/Rejoicing/Relief/Celebration/Wild Partying.*

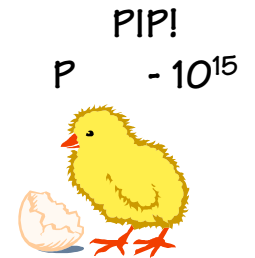
# TIPs Anyone?

I guess that means  
that there are  $10^{12}$   
microphones in a  
Megaphone?



$$\text{MIPS} = \frac{\text{Clock Frequency (in MHz)}}{\text{Clocks per Instruction}}$$

Mega –  $10^6$    Giga –  $10^9$    Tera –  $10^{12}$



Light travels about 1 ft /  $10^{-9}$  secs in free space.

In a Tera-IP uniprocessor no clock-to-clock path can be  
no larger than 300 microns...

We already know of problems that require greater than a TIP  
(Simulations of weather, weapons, brains)

# Driving Down the Denominator

Techniques for increasing parallelism:

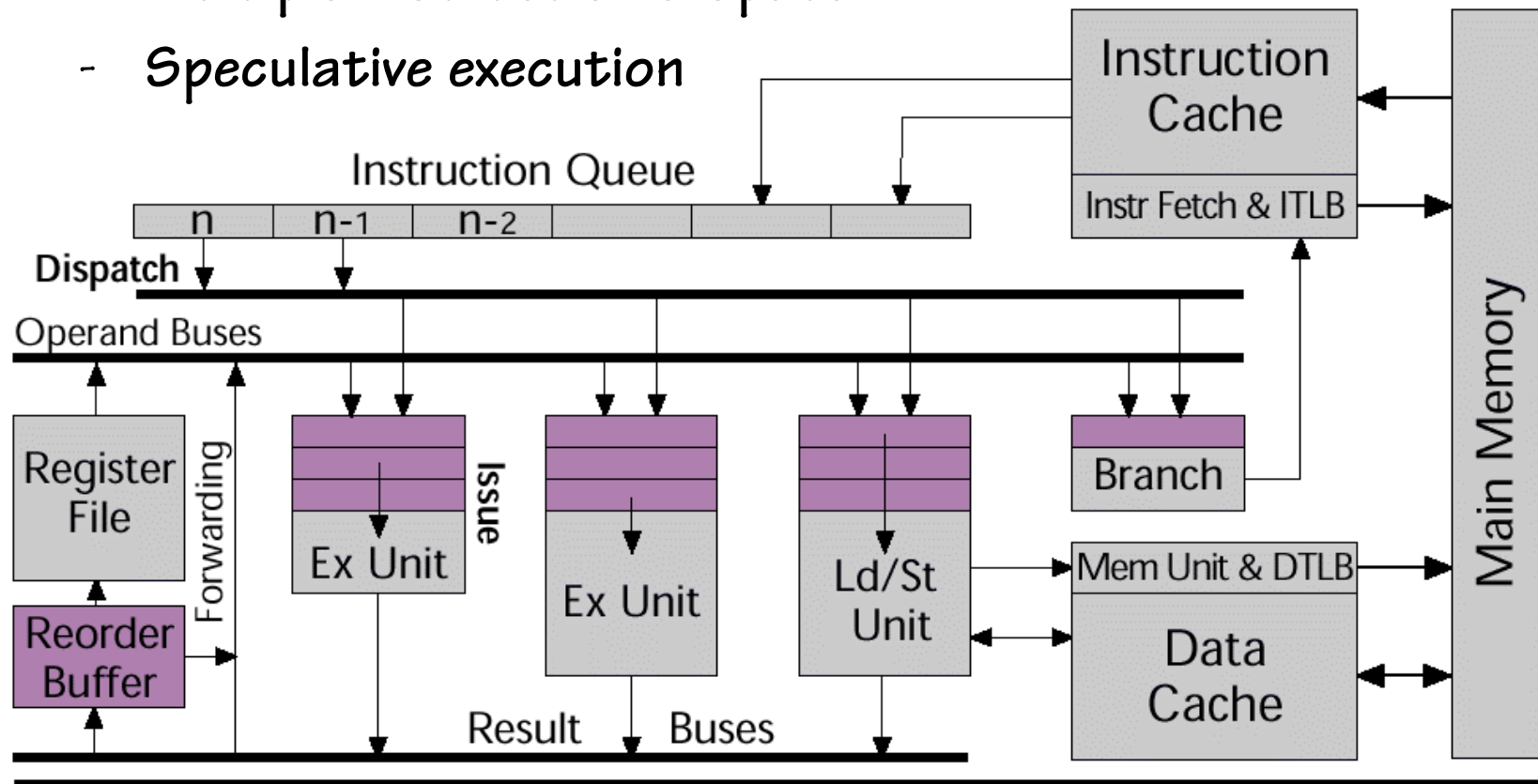
**Pipelining** – reasonable for a small number of stages (5-10), after that bypassing and stalls become unmanageable.

**Superscalar** – replicate data paths and design control logic to discover parallelism in traditional programs.

**Explicit parallelism** – must learn how to write programs that run on multiple CPUs.

# Superscalar Parallelism

- Popular now, but the end is near
- Multiple instruction dispatch
- Speculative execution



# Explicit Parallelism

Control	Communication	processing elements
unified	shared Memory	homogeneous
distributed	Message passing	heterogeneous

## Decoding the Parallel Processor Alphabet Soup:

**SIMD** - Single-Instruction-Multiple-Data

Unified control, Homogeneous processing elements

**VLIW** - Very-Long-Instruction-Word

Unified control, Heterogeneous processing elements

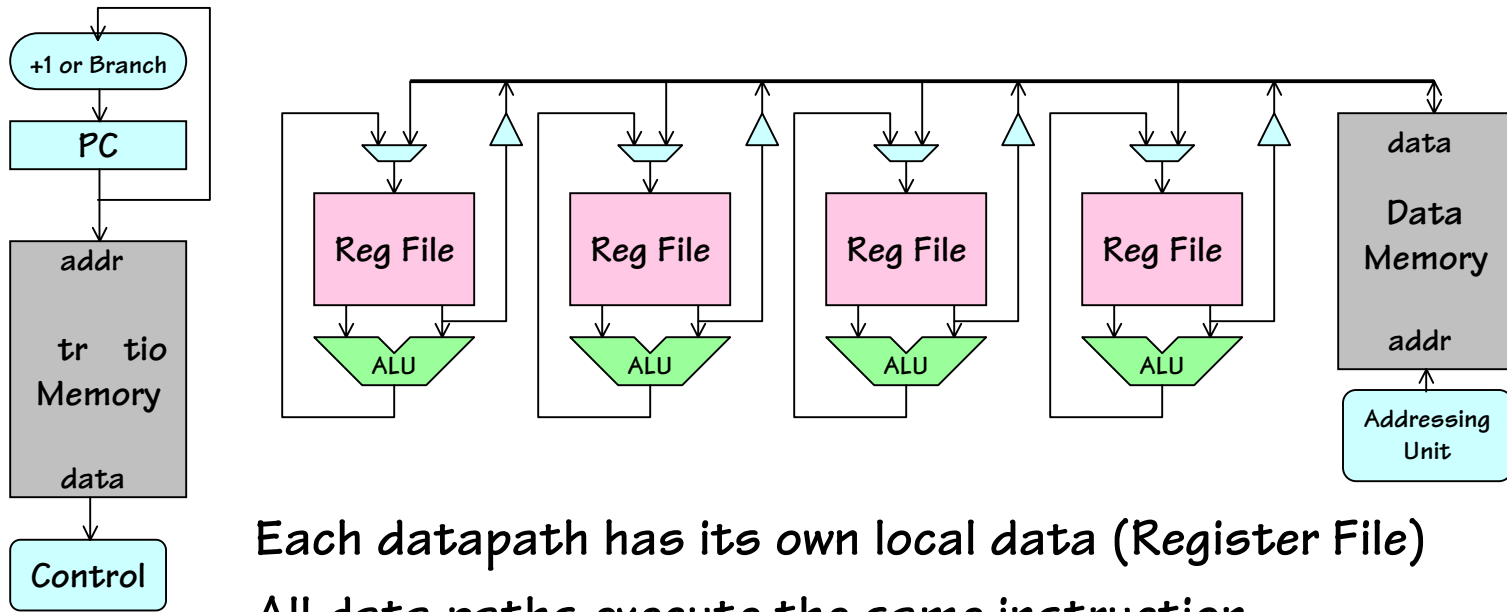
**MIMD** - Multiple-Instruction-Multiple-Data

Distributed control

**SMP** – Symmetric Multi-Processor

Distributed control, Shared memory, Homogeneous PEs

# SIMD Processing



Each datapath has its own local data (Register File)

All data paths execute the same instruction

Conditional branching is difficult...

(What if only one CPU has  $R1 = 0$ ?)

Conditional operations are common in SIMD machines

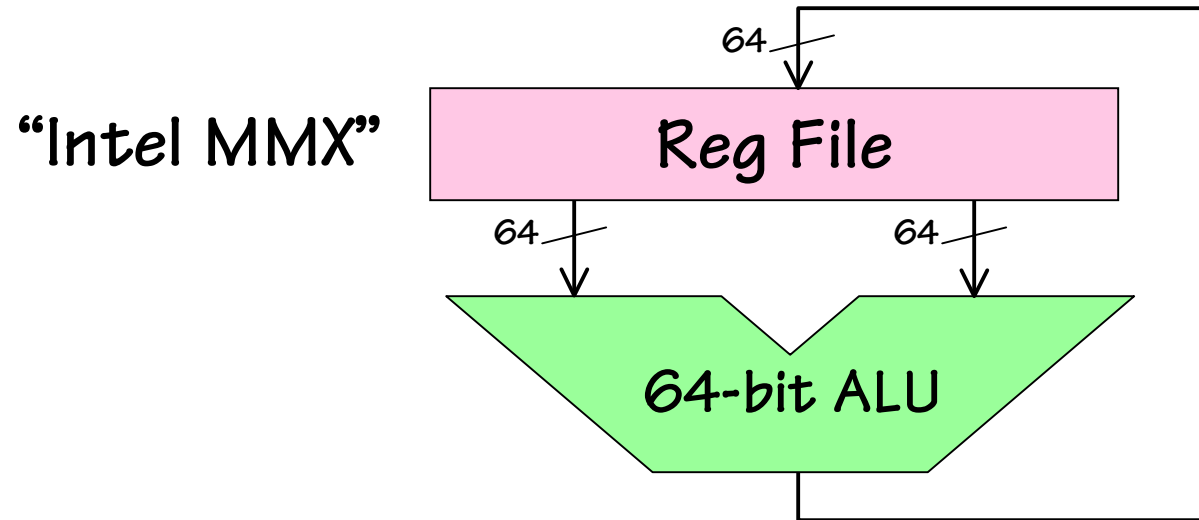
if (flag1)  $Rc = Ra <op> Rb$

Global ANDing or ORing of flag registers are used for high-level control

This sort of construct is also becoming popular on modern uniprocessors



# SIMD Coprocessing Units

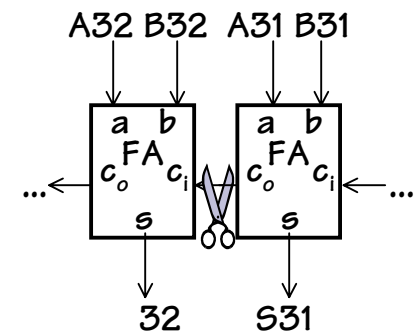
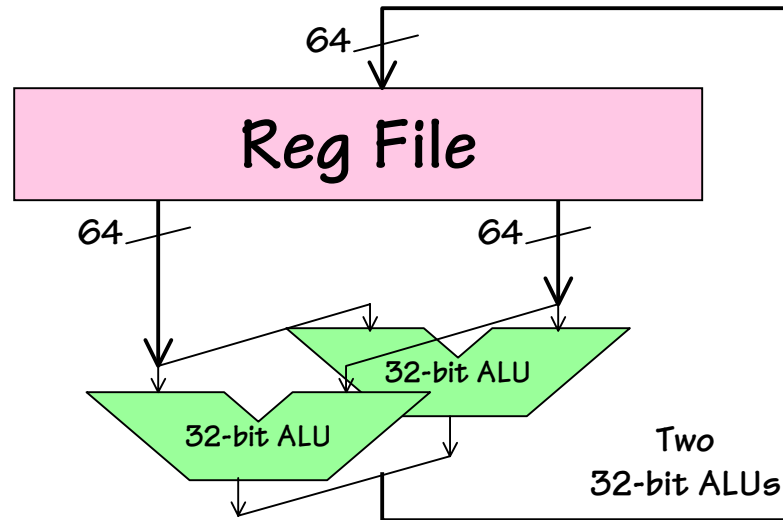


SIMD data path added to a traditional CPU core  
Register-only operands  
Core CPU handles memory traffic  
Partitionable Datapaths for variable-sized  
“PACKED OPERANDS”



# SIMD Coprocessing Units

“Intel MMX”



SIMD data path added to a traditional CPU core

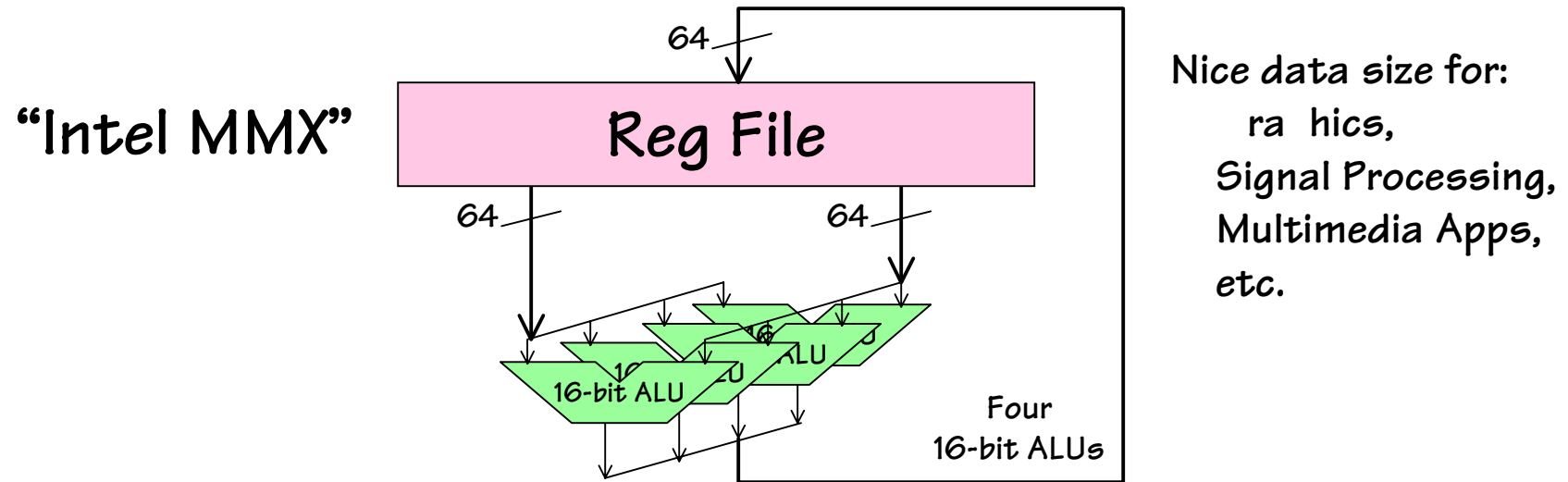
Register-only operands

Core CPU handles memory traffic

Partitionable Datapaths for variable-sized

“PACKED OPERANDS”

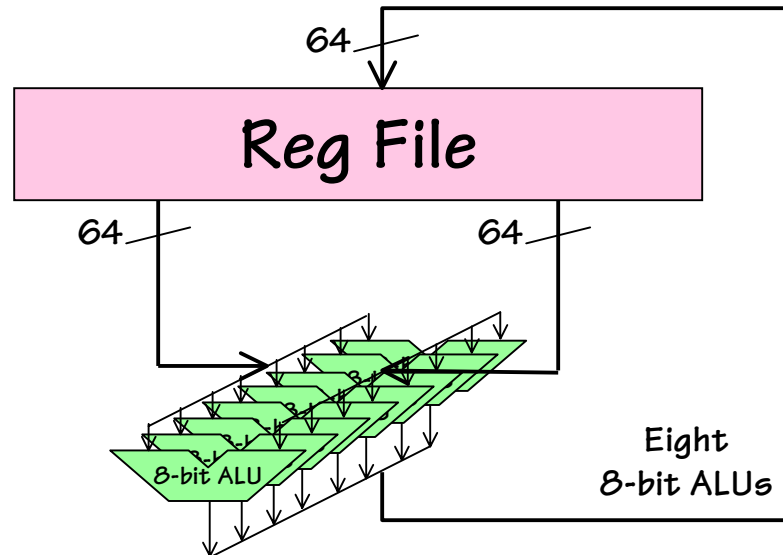
# SIMD Coprocessing Units



SIMD data path added to a traditional CPU core  
Register-only operands  
Core CPU manages memory traffic  
Partitionable Datapaths for variable-sized  
“PACKED OPERANDS”

# SIMD Coprocessing Units

“Intel MMX”



MMX instructions:

PADDB - add bytes

PADDW - add 16-bit words

PADDQ - add 32-bit words  
(unsigned & w/saturation)

PSUB{B,W,D} - subtract

PMULTLW - multiply low

PMULTHW - multiply high

PMADDW - multiply & add

PACK -

U PACK -

PAND -

POR -

SIMD data path added to a traditional CPU core

Register-only operands

Core CPU manages memory traffic

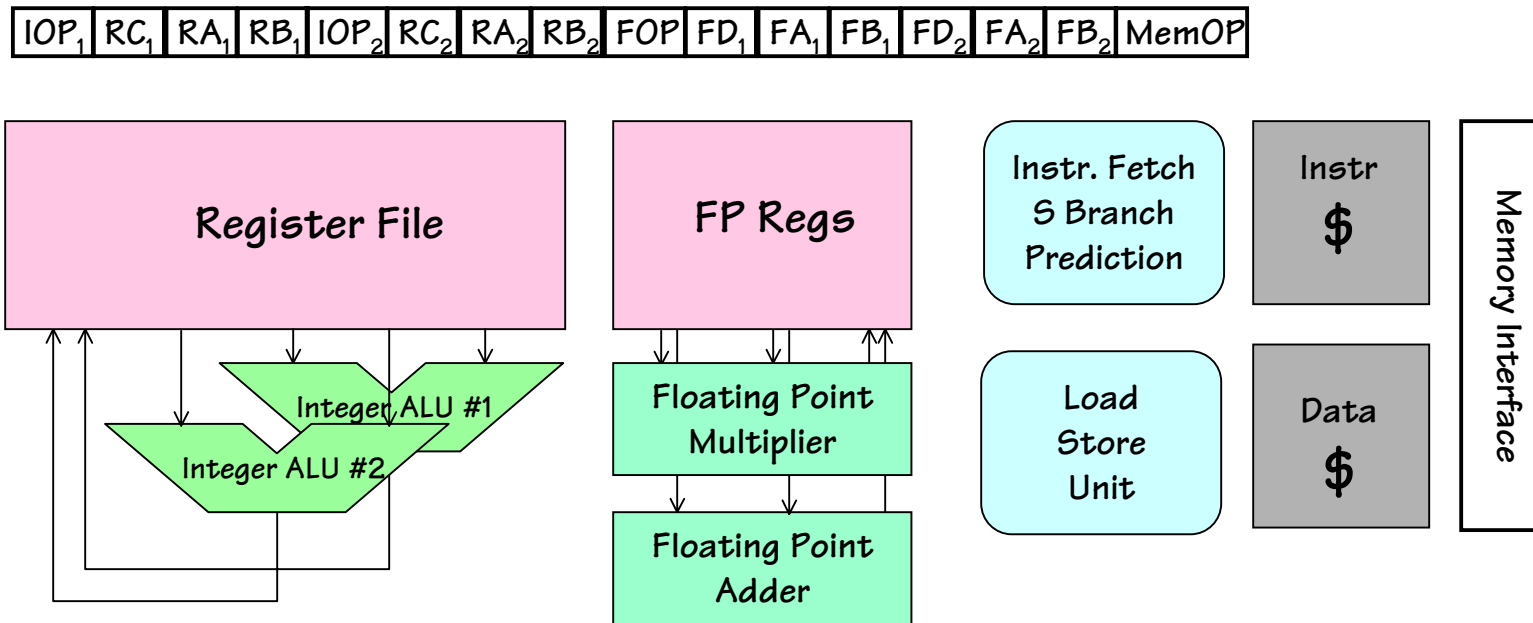
Partitionable Datapaths for variable-sized

“PACKED OPERANDS”

# VLIW Variant of SIMD Parallelism

A single-WIDE instruction controls multiple heterogeneous datapaths.

Exposes parallelism to compiler (SpW vs. HpW)



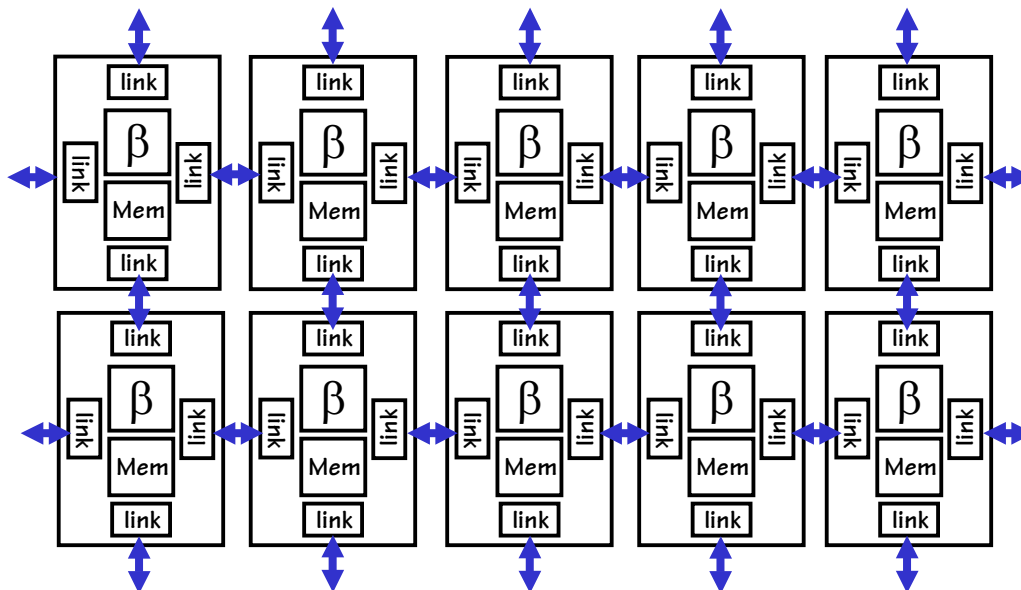
# MIMD Processing - Message Passing

Distributed Control, Homogeneous PEs

Can Leverage existing CPU designs / development tools

H/W focuses on communication (2-D Mesh, N-Cube)

S/W focuses on partitioning of data & algorithms



# MIMD Processing - Shared memory

All processors share a common main memory

Leverages existing CPU designs

Easy to migrate “Processes” to “Processors”

Share data and program

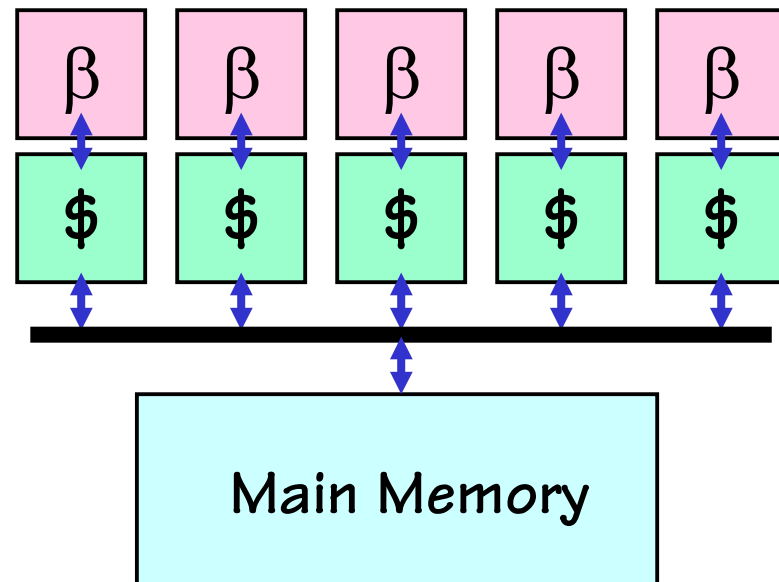
Communicate through  
shared memory

Upgradeable

Problems:

Scalability

Synchronization



# Programming the Beast



6.004 (circa 2000):

```
int factorial(int n) {
    if (n > 0)
        return n*fact(n-1);
    else
        return 1;
}
```

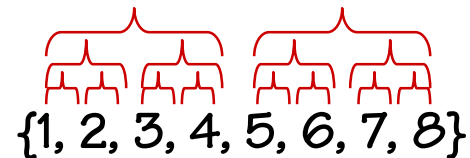
Calls factorial() only  $n$  times

Runs in  $O(N)$  time

6.004 (circa 2020):

```
int factorial(int n) {
    return facthelp(1, n);
}

parallel int facthelp(int from, int to) {
    int mid;
    if (from >= to) return from;
    mid = (from + to)/2;
    return (facthelp(from, mid)*facthelp(mid+1, to));
}
```



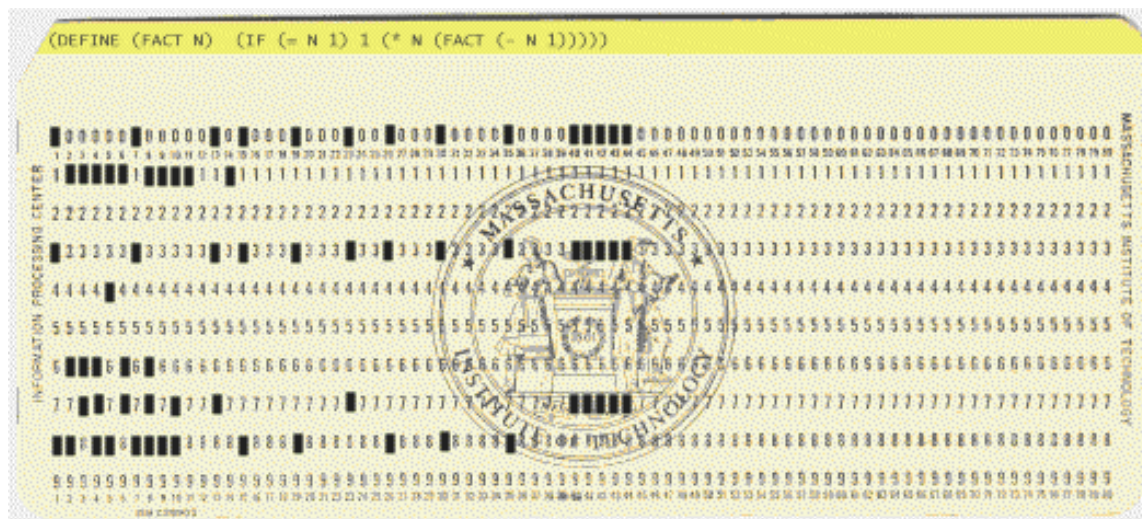
Calls facthelp()  $2n - 1$  times  
(nodes in a binary tree with  $n$  leafs).

Runs in  $O(\log_2(N))$  time  
(on  $N$  processors)

# "Dusty Deck" Problem

How do we make our old *sequential* programs run on parallel machines? After all, what's *easier*, designing new H/W or rewriting all our S/W?

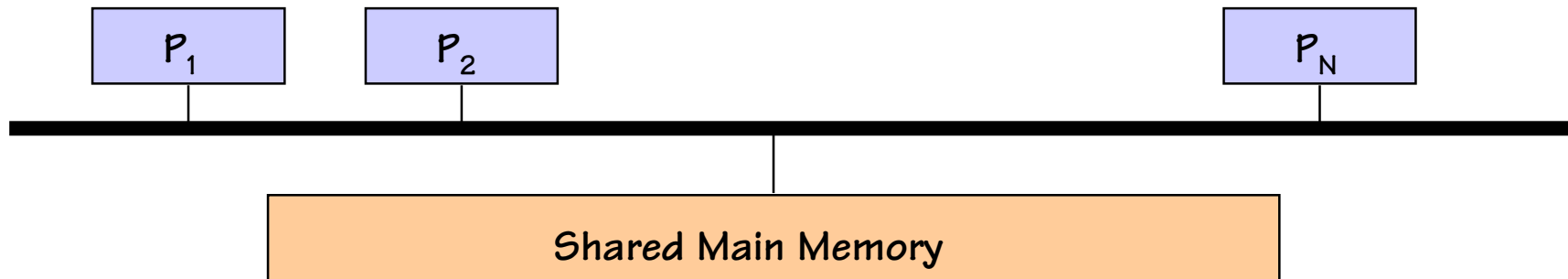
If we treat PROCESSES as a programming constructs (see last lecture)... and assign each process to a separate processor... can't we easily take advantage of parallelism.





# Multiprocessor Fantasies

If one processor is *good*, N processors are *GREAT*:



IDEA:

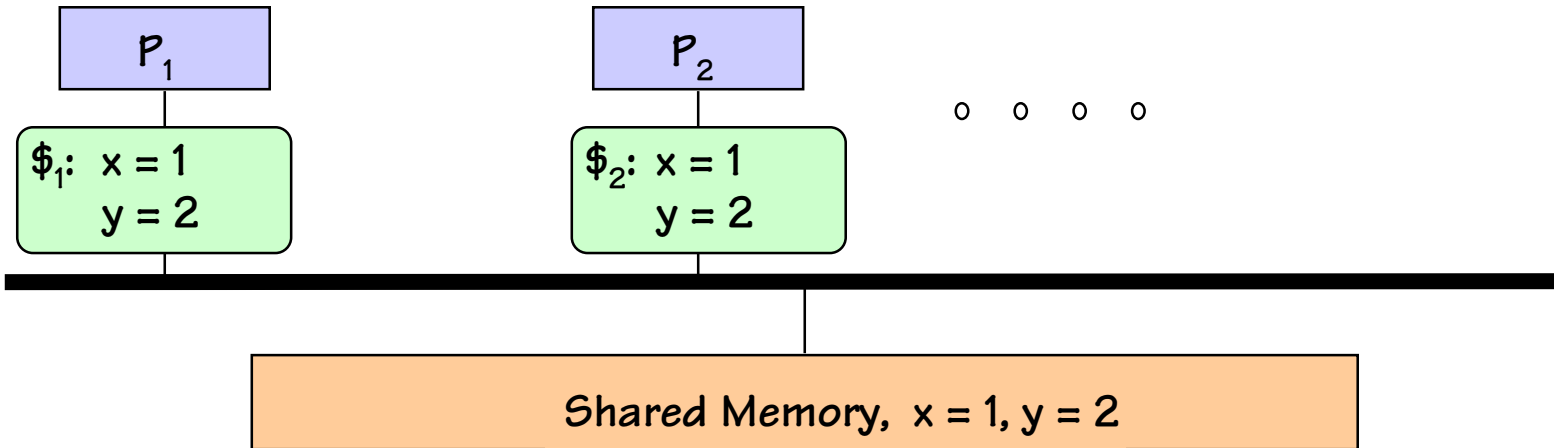
- Run N processes, each on its OWN processor!
- Processors compete for bus mastership, memory access
- Bus **SERIALIZES** memory operations (via arbitration for mastership)

PROBLEM:

The Bus quickly becomes the BOTTLENECK

# Multiprocessor with Caches

But, we've seen this problem before. The solution, add CACHES.



Consider the following trivial processes running on  $P_1$  and  $P_2$ :

Process A

Process B

# What are the Possible Outcomes?

Process A

$\$1$ :  $x = 1$   
 $y = 2$

Process B

$\$2$ :  $x = 1$   
 $y = 2$

Plausible execution sequences:

<u>SEQUENCE</u>	<u>A prints</u>	<u>B prints</u>
$x=3$ ; $\text{print}(y)$ ; $y=4$ ; $\text{print}(x)$ ;	2	1
$x=3$ ; $y=4$ ; $\text{print}(y)$ ; $\text{print}(x)$ ;	2	1
$x=3$ ; $y=4$ ; $\text{print}(x)$ ; $\text{print}(y)$ ;	2	1
$y=4$ ; $x=3$ ; $\text{print}(x)$ ; $\text{print}(y)$ ;	2	1
$y=4$ ; $x=3$ ; $\text{print}(y)$ ; $\text{print}(x)$ ;	2	1
$y=4$ ; $\text{print}(x)$ ; $x=3$ ; $\text{print}(y)$ ;	2	1

# Uniprocessor Outcome

But, what are the possible outcomes if we ran Process A and Process B on a **single time-shared processor**?

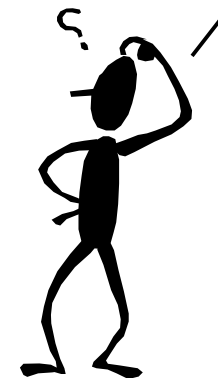
Process A

Process B

Plausible Uniprocessor execution sequences:

<u>SEQUENCE</u>	<u>A prints</u>	<u>B prints</u>
x=3; print(y); y=4; print(x);	2	3
x=3; y=4; print(y); print(x);	4	3
x=3; y=4; print(x); print(y);	4	3
y=4; x=3; print(x); print(y);	4	3
y=4; x=3; print(y); print(x);	4	3
y=4; print(x); x=3; print(y);	4	1

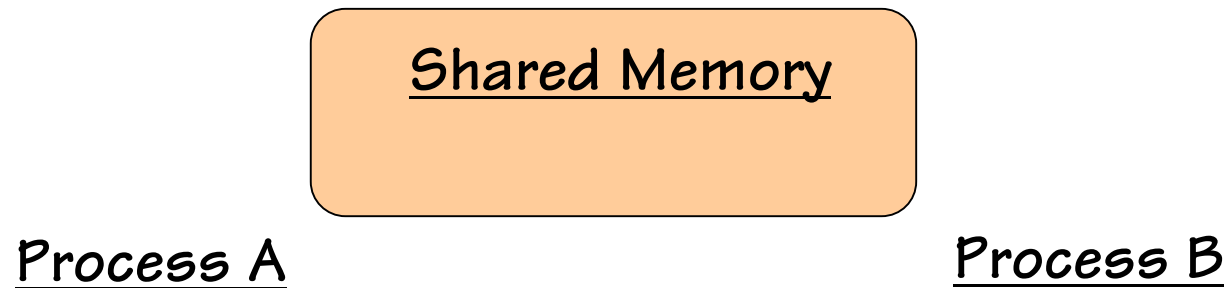
Notice that the outcome 2, 1 does not appear in this list



# Sequential Consistency

Semantic constraint:

Result of executing N parallel programs should correspond to *some* interleaved execution on a single processor.



Possible printed values: 2, 3; 4, 3; 4, 1.  
(each corresponds to at least one interleaved execution)

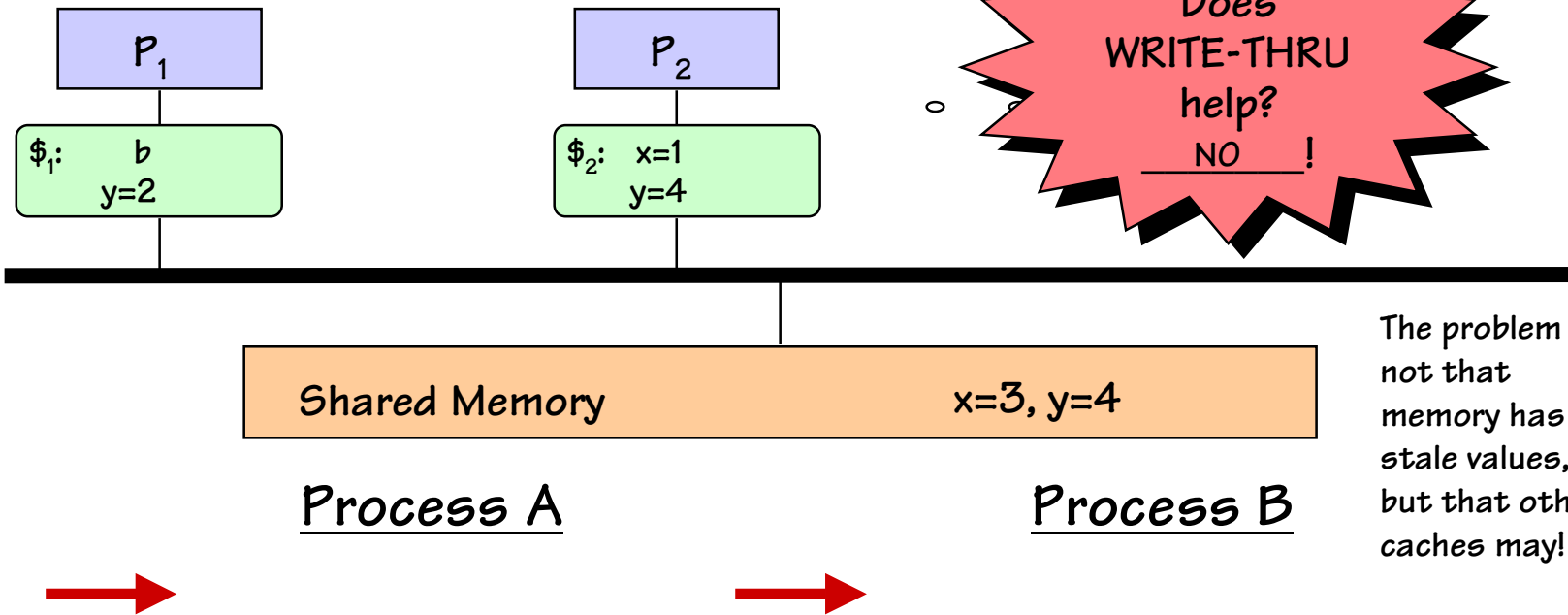
IMPOSSIBLE printed values: 2, 1  
(corresponds to NO valid interleaved execution).



Weren't  
caches  
supposed to  
be invisible  
to  
programs?

# Cache Incoherence

PROBLEM: "stale" values in cache ...



The problem is not that memory has stale values, but that other caches may!

Q: How does B know that A has changed the value of x?

# Cache Coherence Solutions

Problem: A writes data into shared memory; B still sees “stale” cached value.

Solutions:

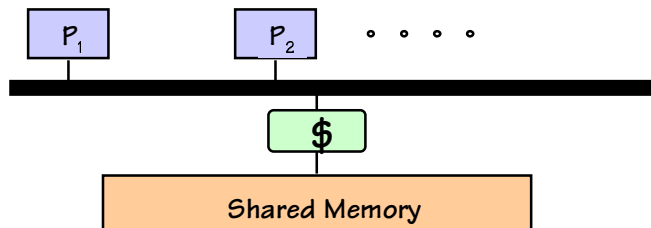
1. Don't cache shared Read/Write pages.

COST: Longer access time to shared memory.

2. Attach cache to shared memory, not to processors...  
... share the cache as well as the memory!

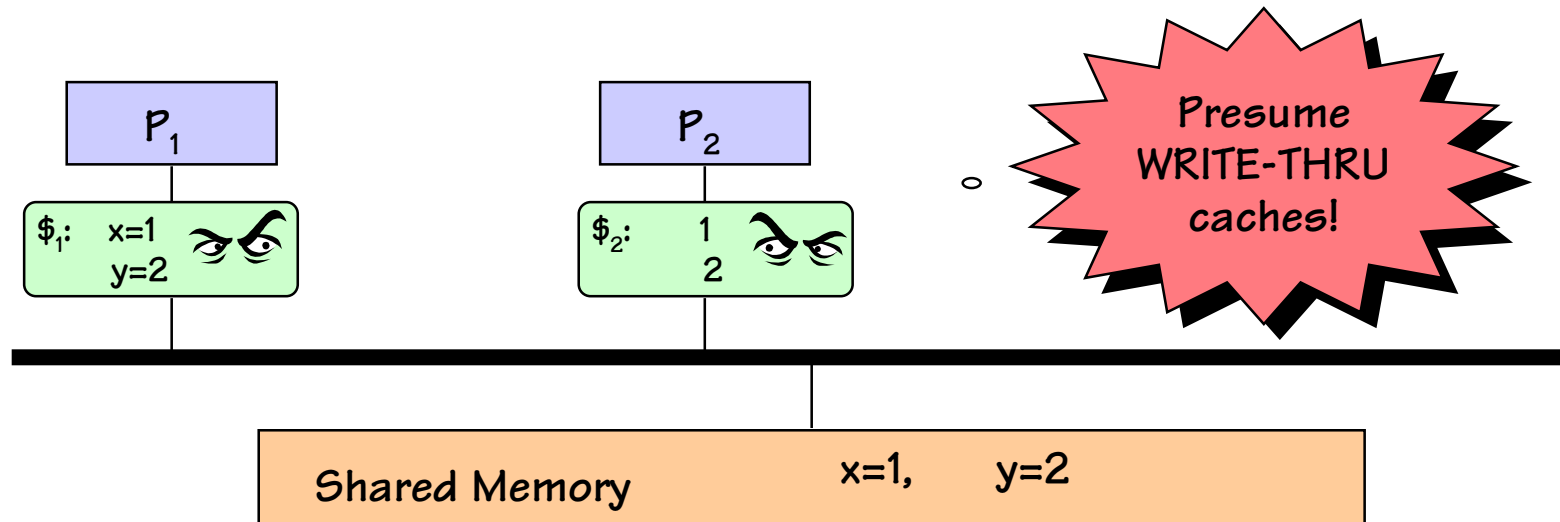
COSTS: 1. Bus Contention

2. Locality



3. Make caches talk to each other, maintain a consistent story.

# "Snoopy" Caches



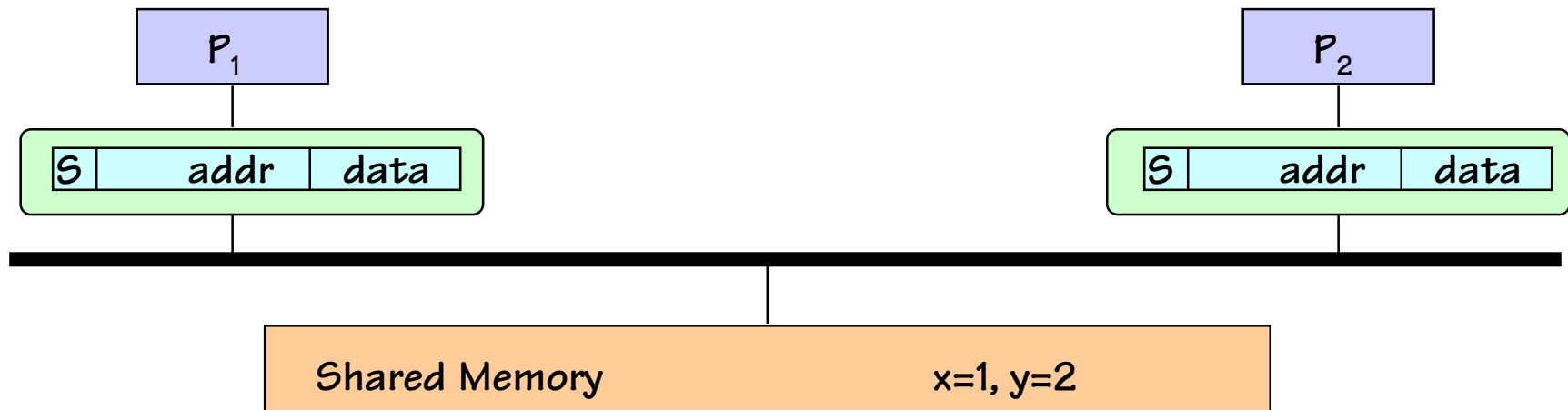
## IDEA:

- $P_1$  writes 3 into  $x$ ; write-thru cache causes bus transaction.
- $P_2$ , snooping, sees transaction on bus. INVALIDATES or UPDATES its cached  $x$  value.

MUST WE use a write-thru strategy?



# Coherency w/ write back



## IDEA:

- Various caches can have
  - Multiple SHARED read-only copies; OR
  - One UNSHARED exclusive-access read-write copy.
- Keep STATE of each cache line in extra bits of tag
- Add bus protocols -- “messages” -- to allow caches to maintain globally consistent state

# Write Acknowledgement

UNIPROCESSORS can post writes or “write behind” --

-- continue with subsequent instructions after having initiated a WRITE transaction to memory (eg, on a cache miss).

HENCE WRITES appear FAST.

Can we take the same approach with multiprocessors?

Consider our example (again)

Process A

Shared Memory

Process B

SEQUENTIAL CONSISTENCY allows (2,3), (4,3), (4,1) printed; but not (2,1).

# Sequential Inconsistency

Process A

Shared Memory

Process B

Plausible sequence of events:

- A writes 3 into x, sends INVALIDATE message.
- B writes 4 into y, sends INVALIDATE message.
- A reads 2 from y, prints it...
- B reads 1 from y, prints it...
- A, B each receive respective INVALIDATE messages.

FIX: Wait for INVALIDATE messages to be acknowledged before proceeding with a subsequent read.

# Who needs Sequential Consistency, anyway?

## ALTERNATIVE MEMORY SEMANTICS:

### *“WEAK” consistency*

EASIER GOAL: Memory operations from each processor appear to be performed in order issued by that processor;

Memory operations from different processors may overlap in arbitrary ways (not necessarily consistent with any interleaving).

## DEC ALPHA APPROACH:

- Weak consistency, by default;
- MEMORY BARRIER instruction: stalls processor until all previous memory operations have completed.

# Semaphores in Parallel Processors

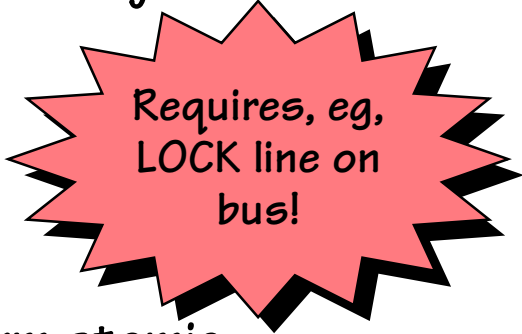
Can semaphores be implemented using ONLY atomic reads and writes?

ANSWER: Only if you're **Dijkstra**

Contemporary solution:

HARDWARE support for atomic *sequences* of memory transactions.

- Explicit LOCK, UNLOCK controls on access to memory:



Requires, eg,  
LOCK line on  
bus!

- Single “Test and Set” instructions which perform atomic READ/MODIFY/WRITE using bus-locking mechanism.
- (ALPHA Variant): “unlock” instruction returns 0 if sequence was interrupted, else returns 1. Program can repeat sequence until its not interrupted.

# Parallel Processing Summary

## Prospects for future CPU architectures:

Pipelining - Well understood, but mined-out  
Superscalar - Nearing its practical limits  
SIMD - Limited use for special applications  
VLIW - Returns controls to S/W. The future?



## Prospects for future Computer System architectures:

SMP - Limited scalability. Harder than it appears.  
MIMD/message-passing - It's been the future for  
over 20 years now. How to program?

NEXT TIME: The REAL future --

New BOUNDARIES, New PROBLEMs

# Coherent Cache States

Two-bit STATE in cache line encodes one of M, E, S, I states (“MESI” cache):

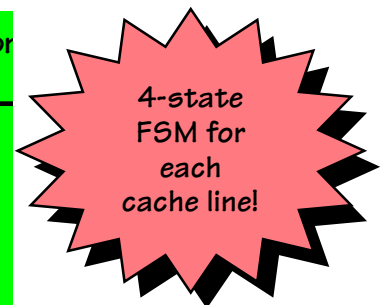
**INVALID:** cache line unused.

**SHARED ACCESS:** read-only, valid, not dirty. Shared with other read-only copies elsewhere. Must invalidate other copies before writing.

**EXCLUSIVE:** exclusive copy, not dirty. On write becomes modified.

**MODIFIED:** exclusive access; read-write, valid, dirty. Must be written back to memory eventually; meanwhile, can be written or read by local processor.

Current state	Read Hit	Read Miss, n Hit	Read Miss. Snoop Miss	rite Hit	Write Miss	Snoop or Read	Snoop for Write
Modified	Modified	Invalid (Wr-Back)	Invalid (Wr-Back)	Modified	Invalid (Wr-Back)	Shared (Push)	Invalid (Push)
Exclusive	Exclusive	Invalid	Invalid	Modified	Invalid	Shared	Invalid
Shared	Shared	Invalid	Invalid	Modified (Invalidate)	Invalid	Shared	Invalid
Invalid	X	Shared (Fill)	Exclusive (Fill)	X	Modified (Fill-Inv)	X	X

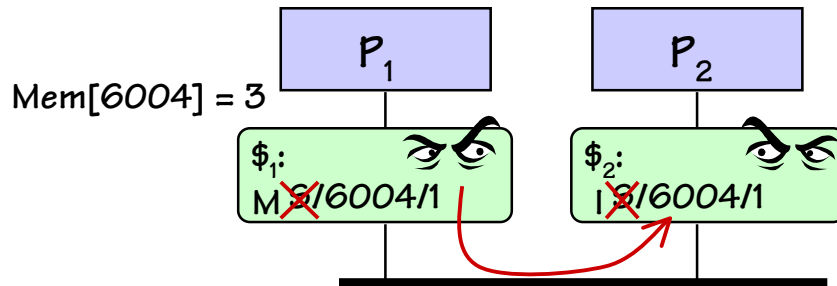


(FREE!!: Can redefine VALID and DIRTY bits)

# MESI Examples

Local WRITE request hits cache line in **Shared** state:

- Send INVALIDATE message forcing other caches to I states
- Change to **Modified** state, proceed with write.



External Snoop READ hits cache line in **Modified** state:

- Write back cache line
- Change to **Shared** state

