Parallelism, Multicore, and Synchronization

Cache Coherency

Cornell Bowers CIS
Computer Science
[K. Bala, A. Bracy, G. Guidi, M. Martin, S. McKee, A. Roth, A. Sampson, Z. Susag, E. Sirer, and H. Weatherspoon]

Parallelism & Synchronization

Multicore → more cores!

Cache Coherency

- Processors cache *shared* data → they see different (incoherent) values for the *same* memory location

Threads

- Mechanism to take advantage of parallelism

Synchronizing parallel programs

- Atomic Instructions
- HW support for synchronization

How to write parallel programs

- Threads and processes
- Critical sections, race conditions, and mutexes

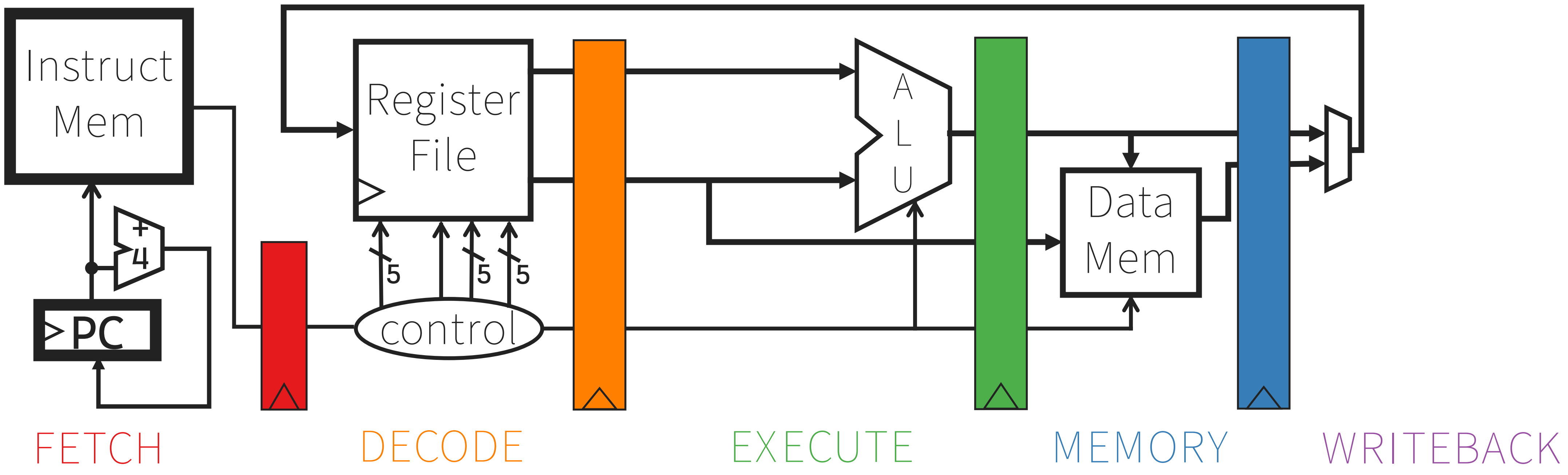
Parallelism & Synchronization

Cache Coherency Problem: What happens when to two or more processors cache shared data?

i.e. the view of memory held by two different processors is through their individual caches.

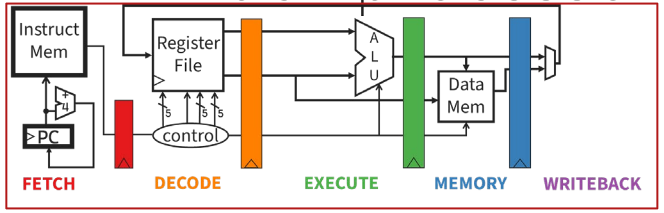
As a result, processors can see different (incoherent) values to the *same* memory location.

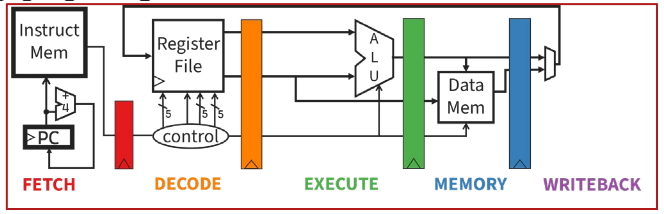
Parallelism & Synchronization

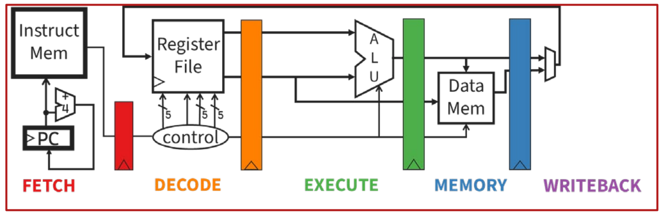


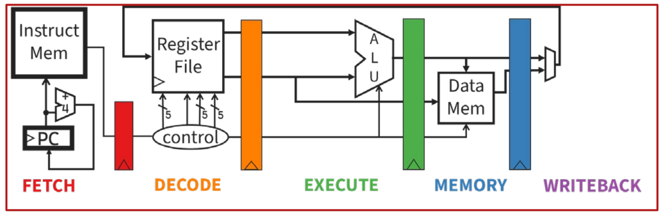
Parallelism & Synchronization

Each processor core has its own L1 cache



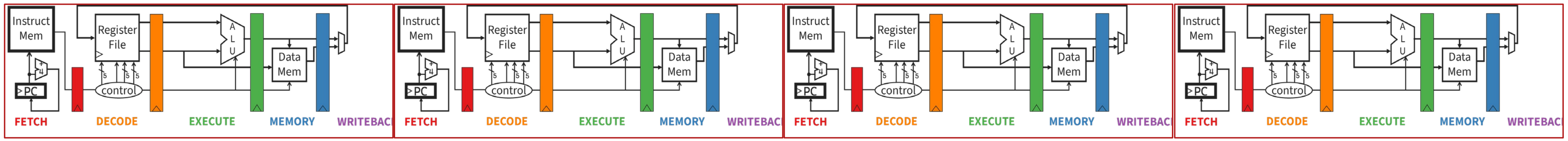






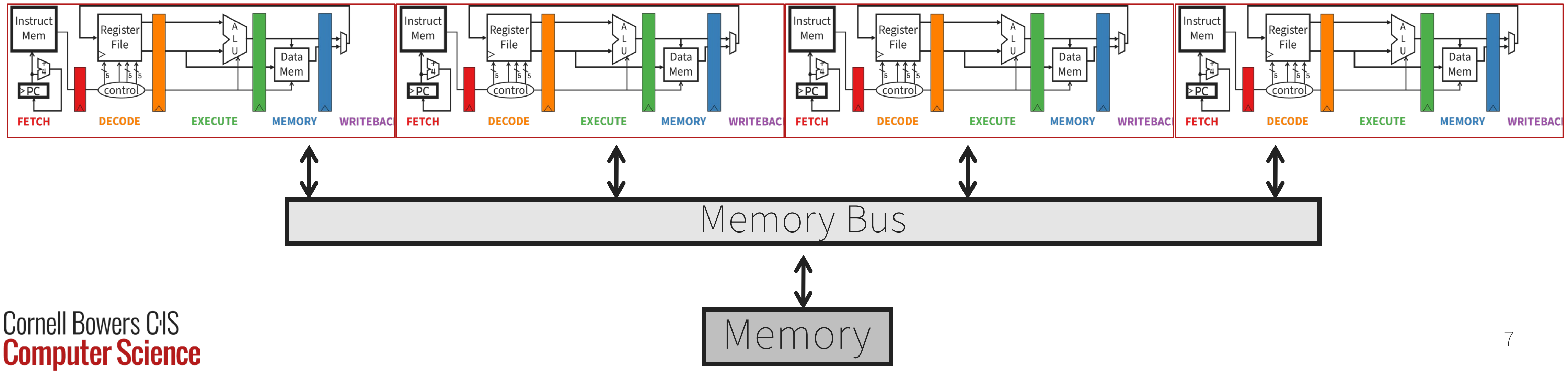
Parallelism & Synchronization

Each processor core has its own L1 cache



Parallelism & Synchronization

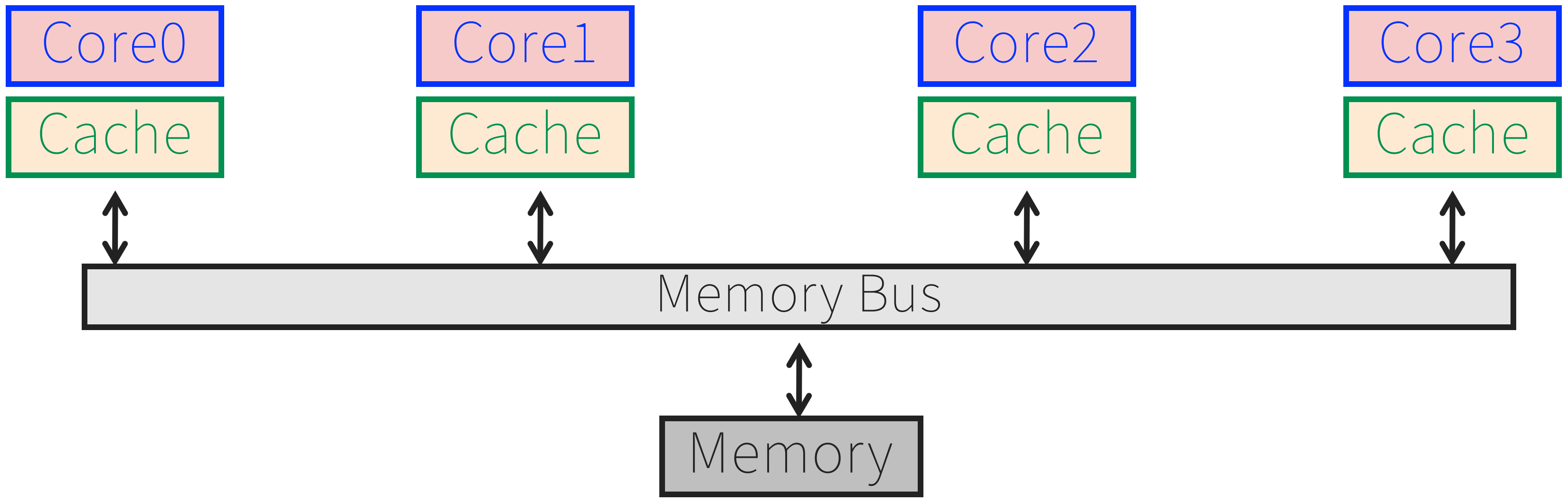
Each processor core has its own L1 cache



Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

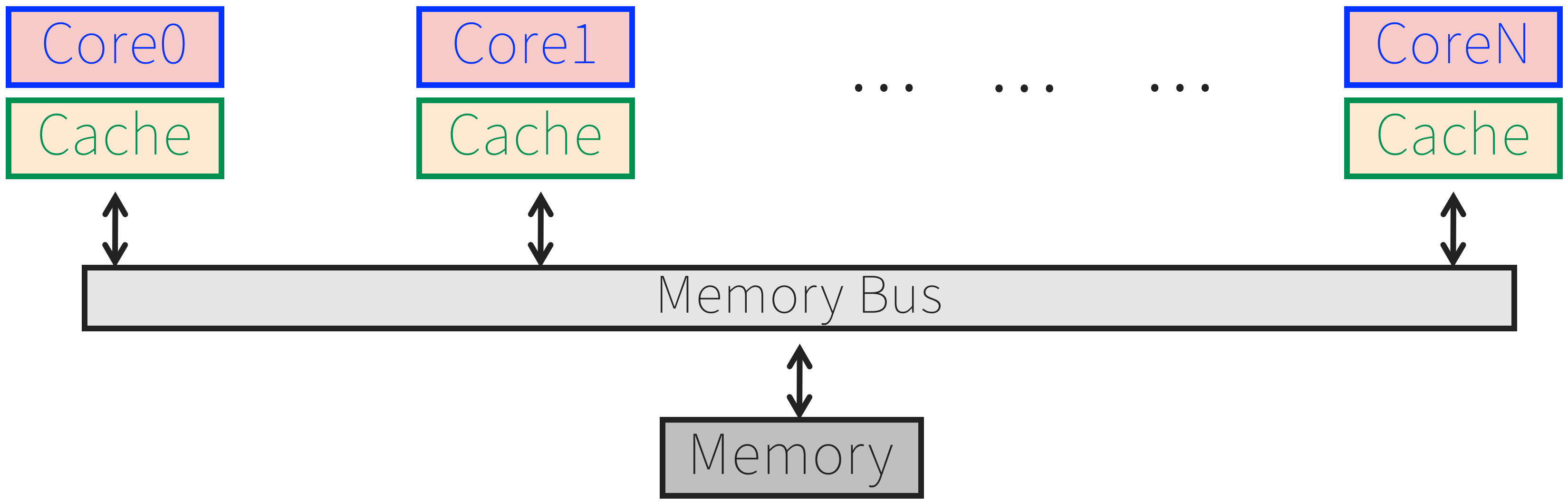
- Typical (today): 1 – 4 processor dies, 2 – 8 cores each
- Hardware provides *single physical address* space for all processors



Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

- Typical (today): 1 – 4 processor dies, 2 – 8 cores each
- Hardware provides *single physical address* space for all processors



Cache Coherency Problem

Thread A (Core0)

```
for (i = 0; i < 5; i++) {
        x = x + 1;

}
```

Thread B (Core1)

```
for (j = 0; j < 5; j++){
        x = x + 1;

}
```

x is a global variable in Data Segment, initialized to 0.
What will the value of x be after both loops finish?

Cornell Bowers C·IS
Computer Science

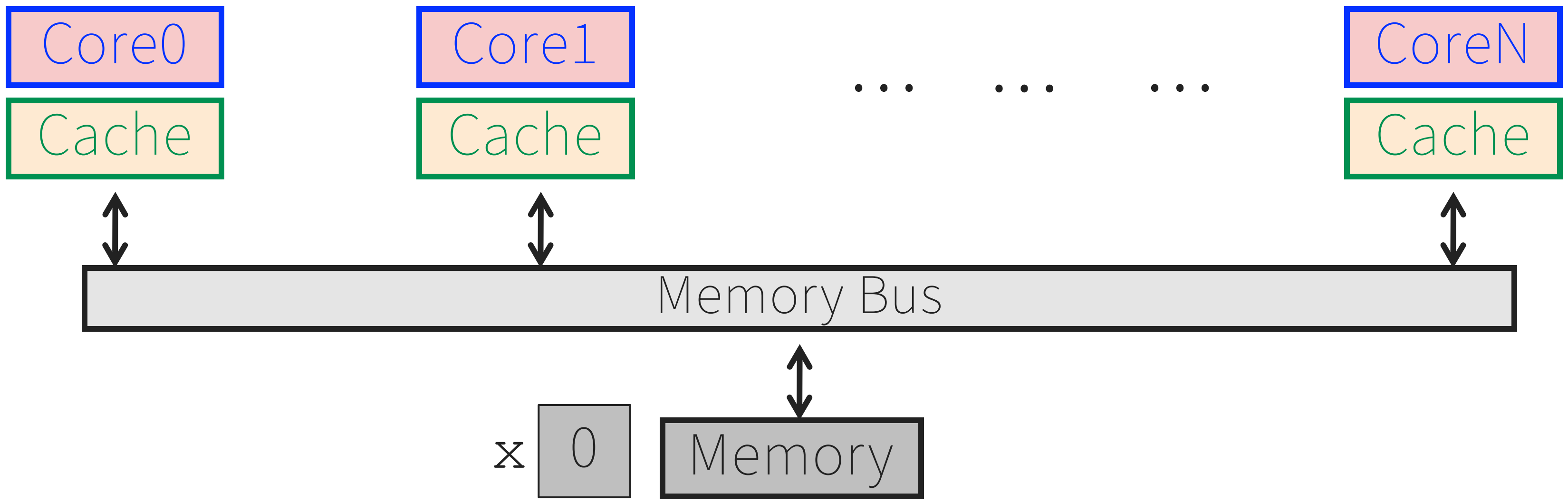
Cache Coherency Problem, WB $

Thread A (Core0)

```
for (i = 0; i < 5; i++) {
    LW t0, addr(x)
    ADDI t0, t0, 1
    SW t0, addr(x)
```

Thread B (Core1)

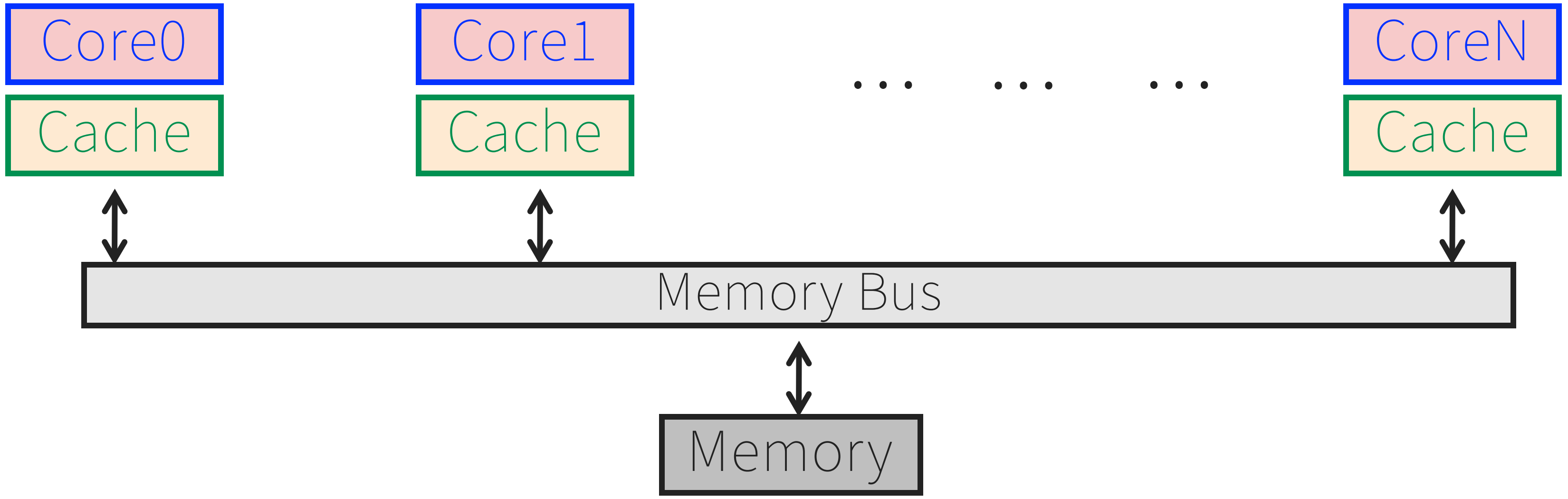
```
for (j = 0; j < 5; j++) {
    LW t0, addr(x)
    ADDI t0, t0, 1
    SW t0, addr(x)
```



Not just a problem for Write-Back Caches

Write-Thru $:

Time step	Event	Core 0's cache	Core 1's cache	Memory
0				0
1	Core 0 reads X			
2	Core 1 reads X			
3	Core 0 writes 1 to X			



Two issues

Coherence

- What values can be returned by a read
- Need a globally uniform (consistent) view of a single memory location

Solution: Cache Coherence Protocols

Consistency

- When a written value will be returned by a read
- Need a globally uniform (consistent) view of *all memory locations relative to each other*

Solution: Memory Consistency Models

Coherence Defined

Informal: Reads return most recently written value

Formal: For concurrent processes P_1 and P_2 and memory location X

- P writes X before P reads X (with no intervening writes)
  $\Rightarrow$ read returns written value
  - (preserve program order)
- P_1 writes X before P_2 reads X
  $\Rightarrow$ read returns written value
  - (coherent memory view, can't read old value forever)
- P_1 writes X and P_2 writes X
  $\Rightarrow$ all processors see writes in the same order
  - all see the same final value for X
  - Aka write serialization
  - (else P_A can see P_2's write before P_1's and P_B can see the opposite; their final understanding of state is wrong)

Cache Coherence Protocols

Operations performed by caches in multiprocessors to ensure coherence

- Migration of data to local caches
  - Reduces bandwidth for shared memory
- Replication of read-shared data
  - Reduces contention for access

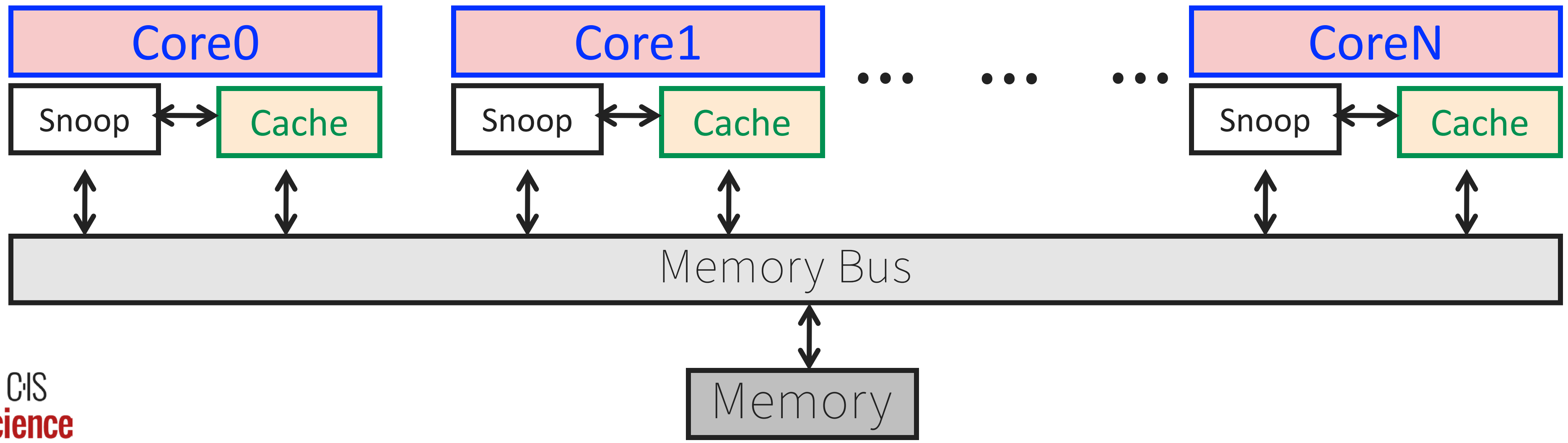
Snooping protocols

- Each cache monitors bus reads/writes

Snooping

Snooping for Hardware Cache Coherence

- All caches monitor bus and all other caches
- Bus read: respond if you have dirty data
- Bus write: update/invalidate your copy of data



Invalidating Snooping Protocols

Cache gets exclusive access to a block when it is to be written

- Broadcasts an invalidate message on the bus
- Subsequent read in another cache misses
  - Owning cache supplies updated value

Time Step	CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
0					0
1	CPU A reads X	Cache miss for X			
2	CPU B reads X	Cache miss for X			
3	CPU A writes 1 to X	Invalidate for X			
4	CPU B read X	Cache miss for X			

Writing

Write-back policies for bandwidth

Write-invalidate coherence policy

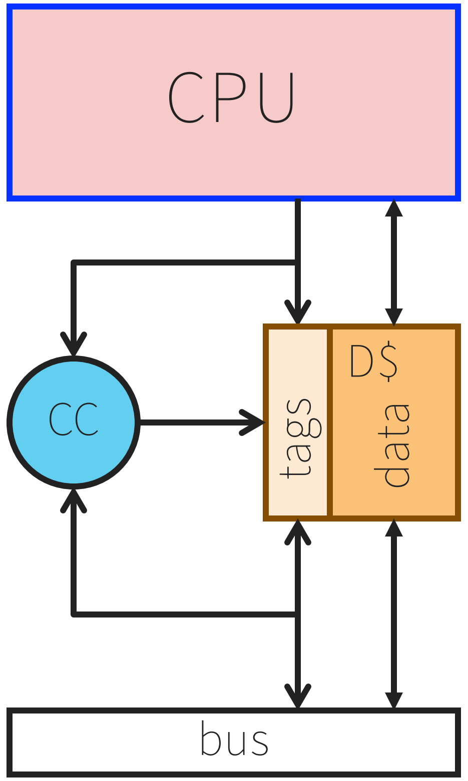
- First invalidate all other copies of data
- Then write it in cache line
- Anybody else can read it

Permits one writer, multiple readers

In reality: many coherence protocols

- Snooping doesn't scale
- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory

Hardware Cache Coherence



Coherence:

- all copies have same data at all times

Coherence controller:

- Examines bus traffic (addresses and data)
- Executes coherence protocol
  - What to do with local copy when you see different things happening on bus

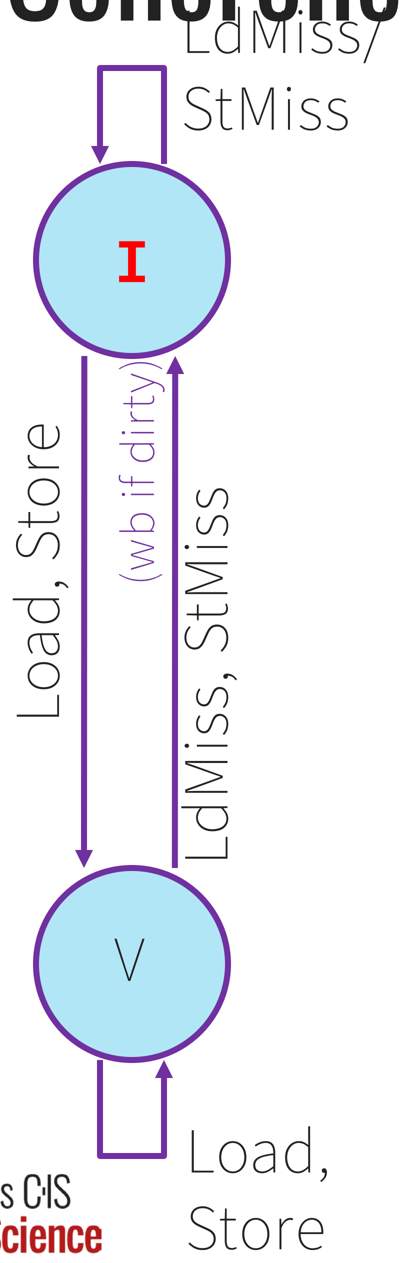
Three processor-initiated events

- Ld: load
- St: store
- WB: write-back

Two remote-initiated events

- LdMiss: read miss from another processor
- StMiss: write miss from another processor

VI Coherence Protocol



VI (valid-invalid) protocol:

- Two states (per block in cache)

  V (valid): have block

  I (invalid): don't have block

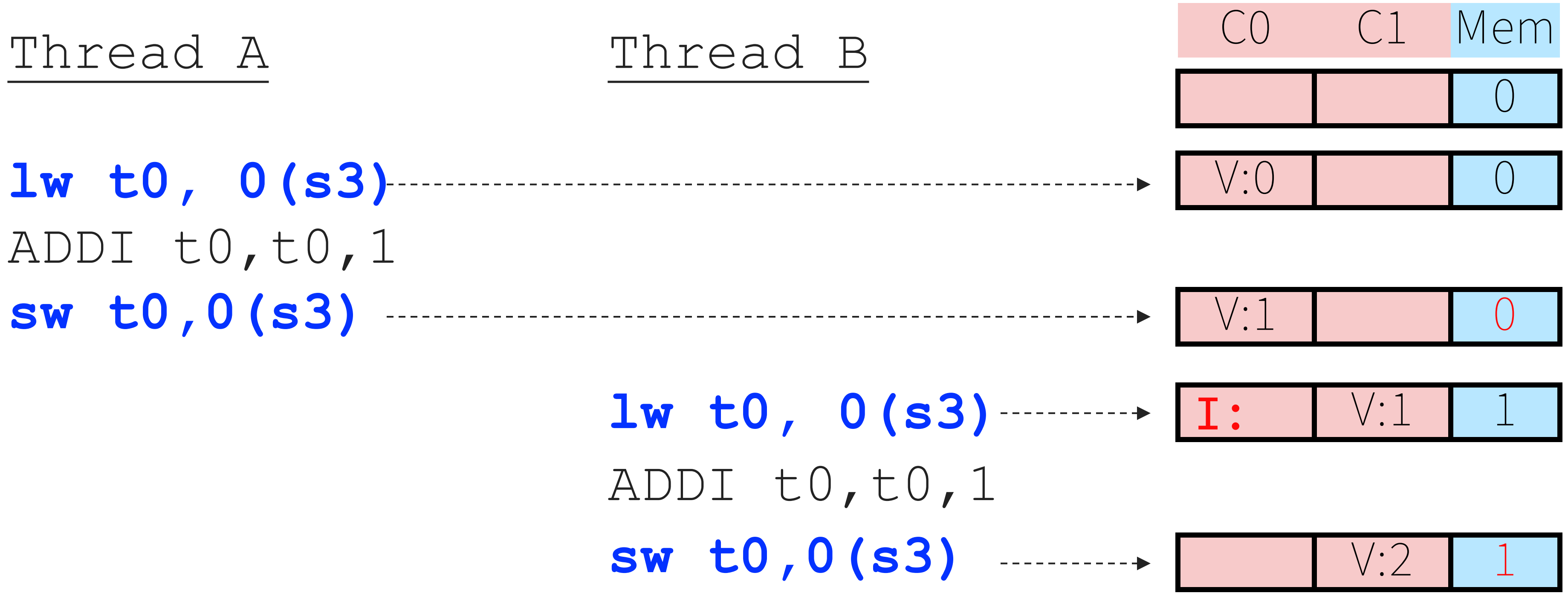
  + Can implement with valid bit

Protocol diagram (left)

- If *you* load/store a block: transition to V
- If anyone *else* wants to read/write block:
  - Give it up: transition to **I** state

  → Write-back if your own copy is dirty

VI Protocol (WB $)



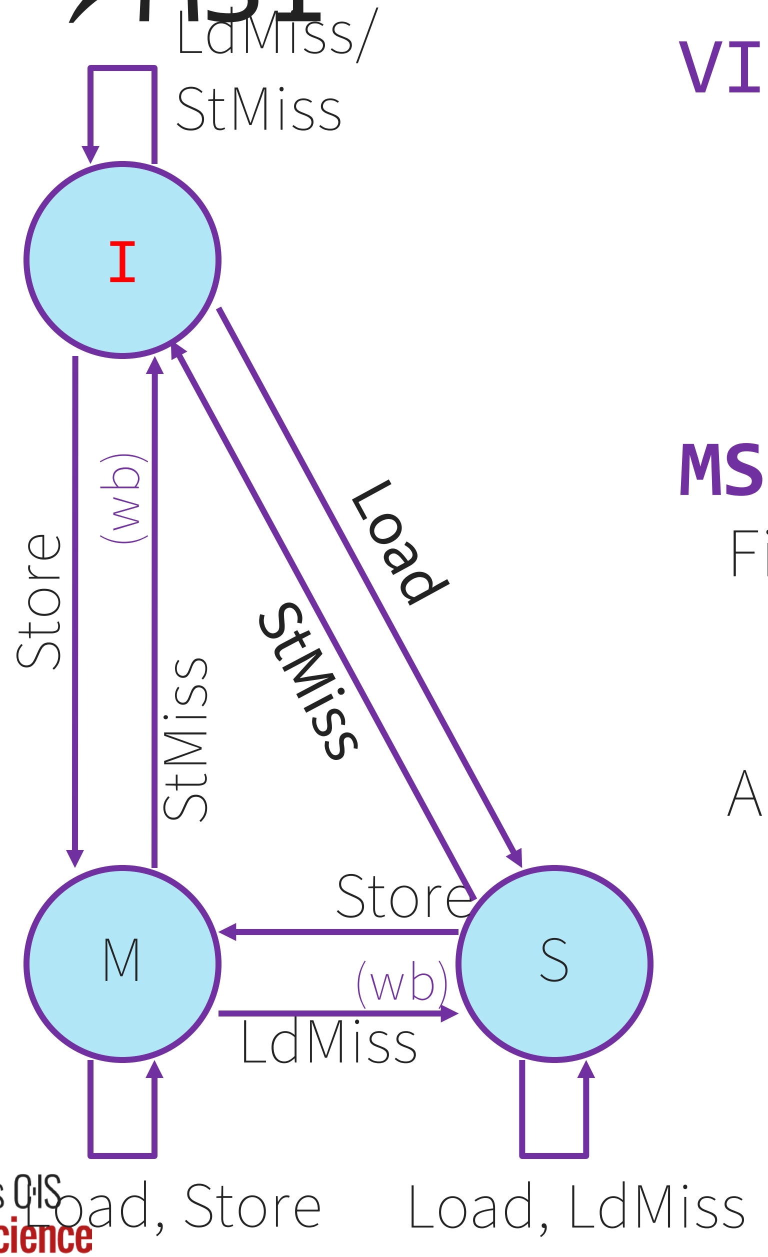
```
Thread A                 Thread B

lw t0, 0(s3)
ADDI t0,t0,1
sw t0,0(s3)
                         lw t0, 0(s3)
                         ADDI t0,t0,1
                         sw t0,0(s3)
```

C0	C1	Mem
		0
V:0		0
V:1		0
I:	V:1	1
	V:2	1

lw by Thread B generates an “other load miss” event (LdMiss)

- Thread A responds by sending its dirty copy, transitioning to **I**

VI → MSI



VI protocol is inefficient

- Only 1 cached copy allowed anywhere
- Can't have multiple read-only copies
  - Not a problem in example
  - Big problem in reality

MSI (modified-shared-invalid)

Fixes problem: splits "V" state into two:

- M (modified): local dirty copy
- S (shared): local clean copy

Allows either

- Multiple read-only copies (S-state)

  --OR--
- Single read/write copy (M-state)

MSI Protocol (WB $)

```
Thread A                 Thread B

lw t0, 0(s3)
ADDIU t0,t0,1
sw t0,0(s3)
                         lw t0, 0(s3)
                         ADDIU t0,t0,1
                         sw t0,0(s3)
```

C0	C1	Mem
		0
S:0		0
M:1		0
S:1	S:1	1
I:	M:2	1

lw by Thread B generates a "other load miss" event (LdMiss)

- Thread A responds by sending its dirty copy, transitioning to S

sw by Thread B generates a "other store miss" event (StMiss)

- Thread A responds by transitioning to **I**

Cache Coherence and Cache Misses

Coherence introduces two new kinds of cache misses

Upgrade miss

- Happens on stores to read-only blocks (i.e., S $\rightarrow$ M)
- Delay to acquire write permission to read-only block

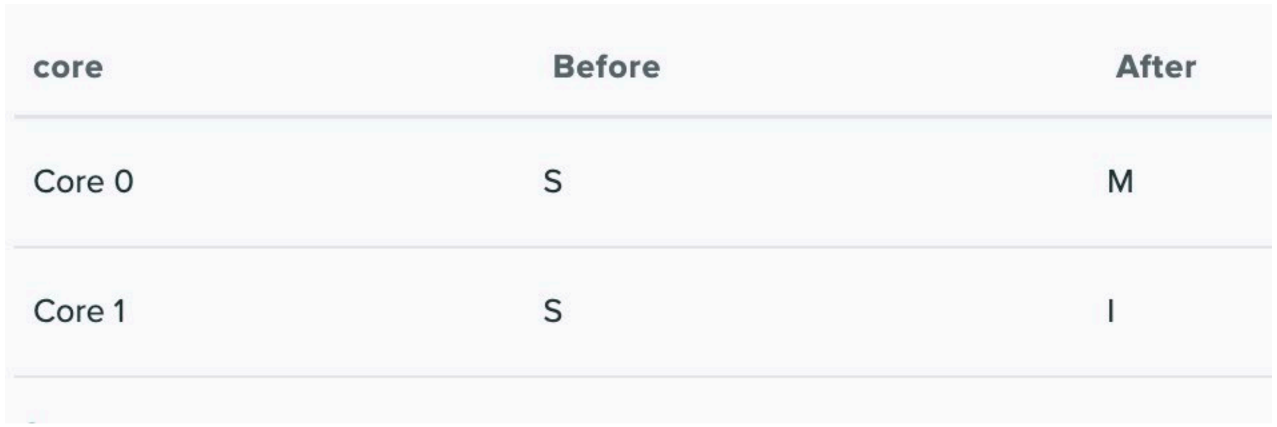
Coherence miss

- Miss to a block evicted by another processor’s requests

Making the cache larger…

- Doesn’t reduce these type of misses
- As cache grows large, these sorts of misses dominate

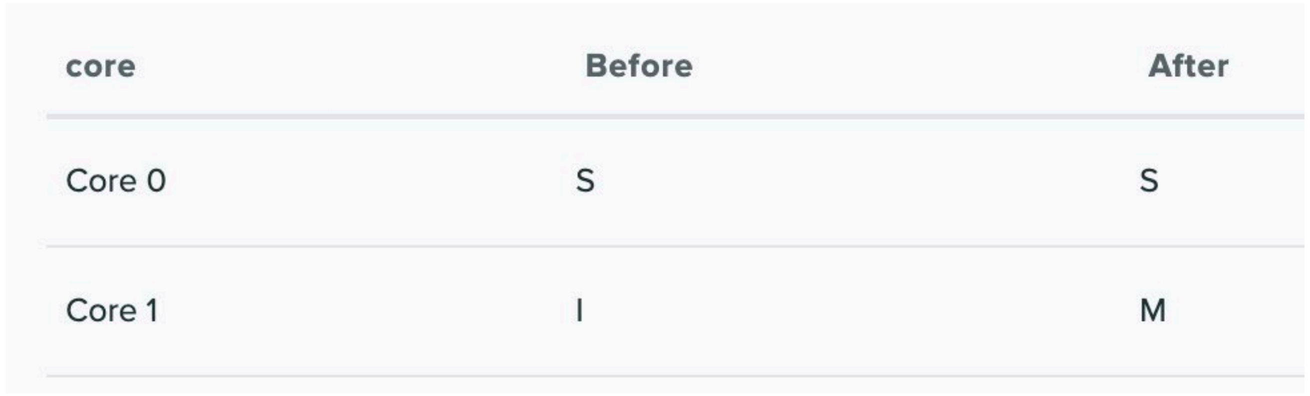
Imagine a two-core processor using the MSI coherence protocol. For each question below, assume that a single line exists in both processors' caches, but possibly in different coherence states. Each problem shows the two states for this line. Then, it shows the two states for the same line after a *single* memory access on *one* core. Your job is to determine what must have happened for that access.



core	Before	After
Core 0	S	M
Core 1	S	I

A. Core 0: upgrade miss
B. Core 0: other (non-upgrade) miss
C. Core 1: upgrade miss
D. Core 1: other (non-upgrade) miss
E. Impossible

Imagine a two-core processor using the MSI coherence protocol. For each question below, assume that a single line exists in both processors' caches, but possibly in different coherence states. Each problem shows the two states for this line. Then, it shows the two states for the same line after a *single* memory access on *one* core. Your job is to determine what must have happened for that access.



core	Before	After
Core 0	S	S
Core 1	I	M

A. Core 0: upgrade miss
B. Core 0: other (non-upgrade) miss
C. Core 1: upgrade miss
D. Core 1: other (non-upgrade) miss
E. Impossible

False sharing

- Two or more processors sharing parts of the same block
- But *not* the same bytes within that block (no actual sharing)
- Creates pathological "ping-pong" behavior
- Careful data placement may help, but is difficult

False Sharing (a.k.a. Cache Thrashing) – Demo

Blue / black vars are local (per thread)
Orange vars are shared b/w threads

A

```
//t0    0, 4, 8, 12...
//t1    1, 5, 9, 13...
for (int i = thread_id; i < SIZE; i += NUM_THREADS) {
   data[i] *= factor;
}
```

B

```
//t0    0,1,2,...499
//t1    500,501,502....,999
int ipt = SIZE / NUM_THREADS;
for (int i = 0; i < ipt; i++) {
  data[ipt*thread_id + i] *= factor;
}
```

Cornell Bowers CIS
Computer Science

More Cache Coherence

Many coherence protocols

- Snooping: VI, MSI, MESI, MOESI, …
    - But Snooping doesn't scale
- Directory-based protocols
    - Caches & memory record blocks' sharing status in directory
    - Nothing is free → directory protocols are slower!

Cache Coherency:

- requires that reads return most recently written value
- Is a hard problem!

Takeaway: Summary of cache coherence

Informally, Cache Coherency requires that reads return most recently written value

Cache coherence hard problem

Snooping protocols are one approach