

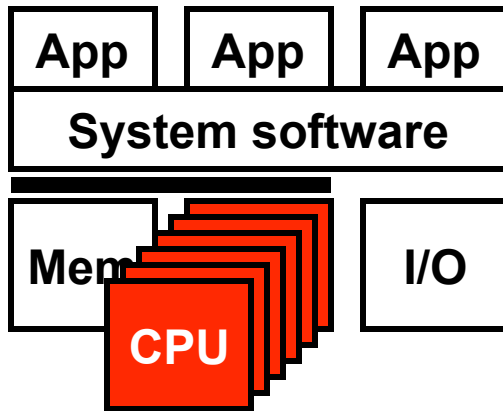
CIS 371

Computer Organization and Design

Unit 11: Multicore

Slides developed by Milo Martin & Amir Roth at the University of Pennsylvania
with sources that included University of Wisconsin slides
by Mark Hill, Guri Sohi, Jim Smith, and David Wood

This Unit: Shared Memory Multiprocessors



- Thread-level parallelism (TLP)
- Shared memory model
 - Multiplexed uniprocessor
 - Hardware multithreading
 - Multiprocessing
- Cache coherence
 - Valid/Invalid, MSI, MESI
- Parallel programming
- Synchronization
 - Lock implementation
 - Locking gotchas
 - Transactional memory
- Memory consistency models

Readings

- P&H
 - Chapter 7.1-7.3, 7.5
 - Chapter 5.8, 5.10

Beyond Implicit Parallelism

- Consider “daxpy”:

```
double a, x[SIZE], y[SIZE], z[SIZE];  
void daxpy():  
    for (i = 0; i < SIZE; i++)  
        z[i] = a*x[i] + y[i];
```

- Lots of instruction-level parallelism (ILP)
 - Great!
 - But how much can we really exploit? 4 wide? 8 wide?
 - Limits to (efficient) super-scalar execution
- But, if SIZE is 10,000, the loop has 10,000-way parallelism!
 - How do we exploit it?

Explicit Parallelism

- Consider “daxpy”:

```
double a, x[SIZE], y[SIZE], z[SIZE];
void daxpy():
    for (i = 0; i < SIZE; i++)
        z[i] = a*x[i] + y[i];
```

- Break it up into N “chunks” on N cores!

- Done by the programmer (or maybe a really smart compiler)

```
void daxpy(int chunk_id):
    chunk_size = SIZE / N
    my_start = chunk_id * chunk_size
    my_end = my_start + chunk_size
    for (i = my_start; i < my_end; i++)
        z[i] = a*x[i] + y[i]
```

SIZE = 400, N=4

Chunk ID	Start	End
0	0	99
1	100	199
2	200	299
3	300	399

- Assumes

- Local variables are “private” and x, y, and z are “shared”
- Assumes SIZE is a multiple of N (that is, $SIZE \% N == 0$)

Explicit Parallelism

- Consider “daxpy”:

```
double a, x[SIZE], y[SIZE], z[SIZE];
void daxpy(int chunk_id) :
    chunk_size = SIZE / N
    my_start = chunk_id * chunk_size
    my_end = my_start + chunk_size
    for (i = my_start; i < my_end; i++)
        z[i] = a*x[i] + y[i]
```

- Main code then looks like:

```
parallel_daxpy() :
    for (tid = 0; tid < CORES; tid++) {
        spawn_task(daxpy, tid);
    }
    wait_for_tasks(CORES);
```

Explicit (Loop-Level) Parallelism

- Another way: “OpenMP” annotations to inform the compiler

```
double a, x[SIZE], y[SIZE], z[SIZE];
void daxpy() {
    #pragma omp parallel for
    for (i = 0; i < SIZE; i++) {
        z[i] = a*x[i] + y[i];
    }
}
```

- Look familiar?
 - Hint: from the homework
- But only works if loop is actually parallel
 - If not parallel, incorrect behavior may result in unpredictable ways

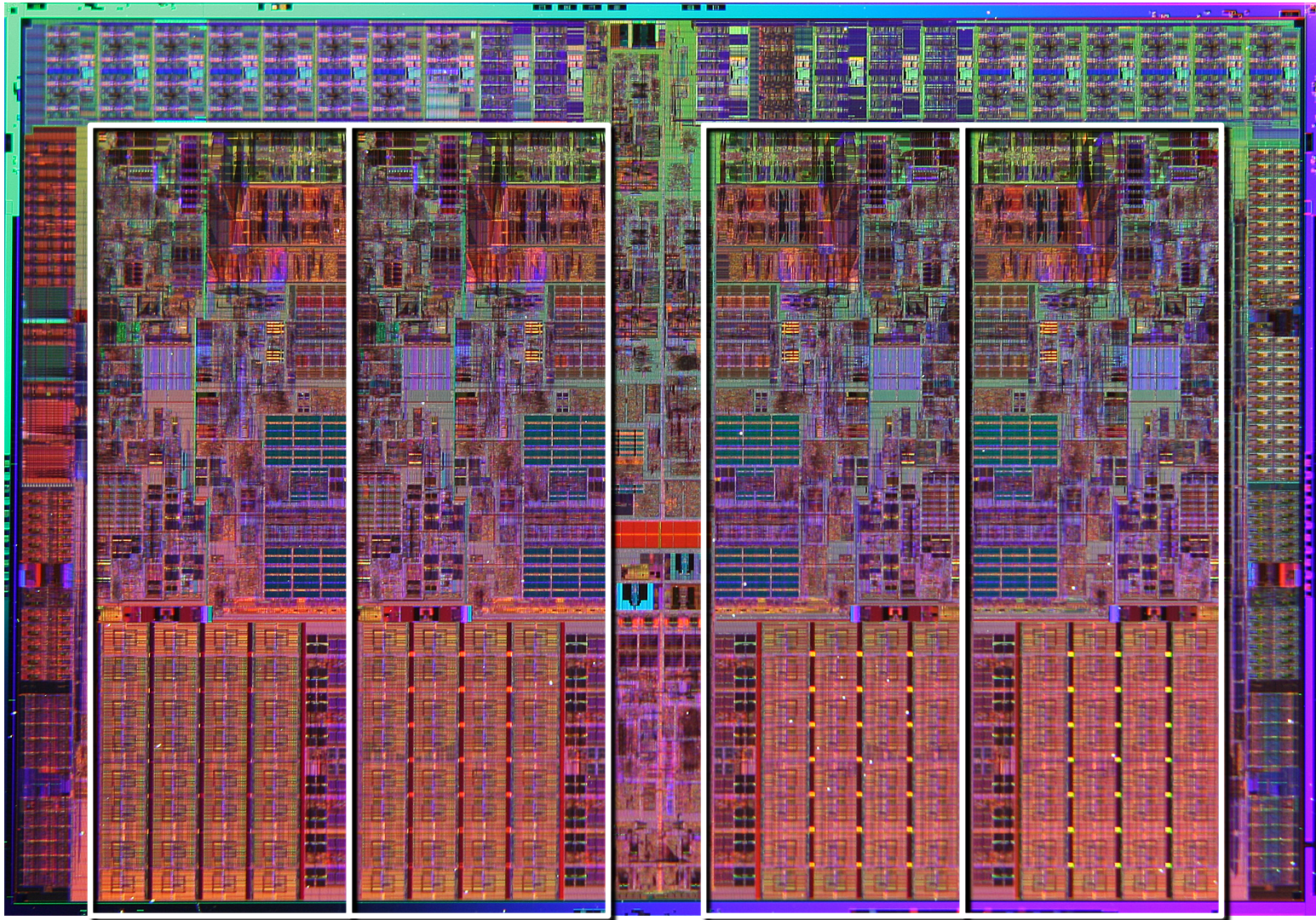
Multicore & Multiprocessor Hardware

Multiplying Performance

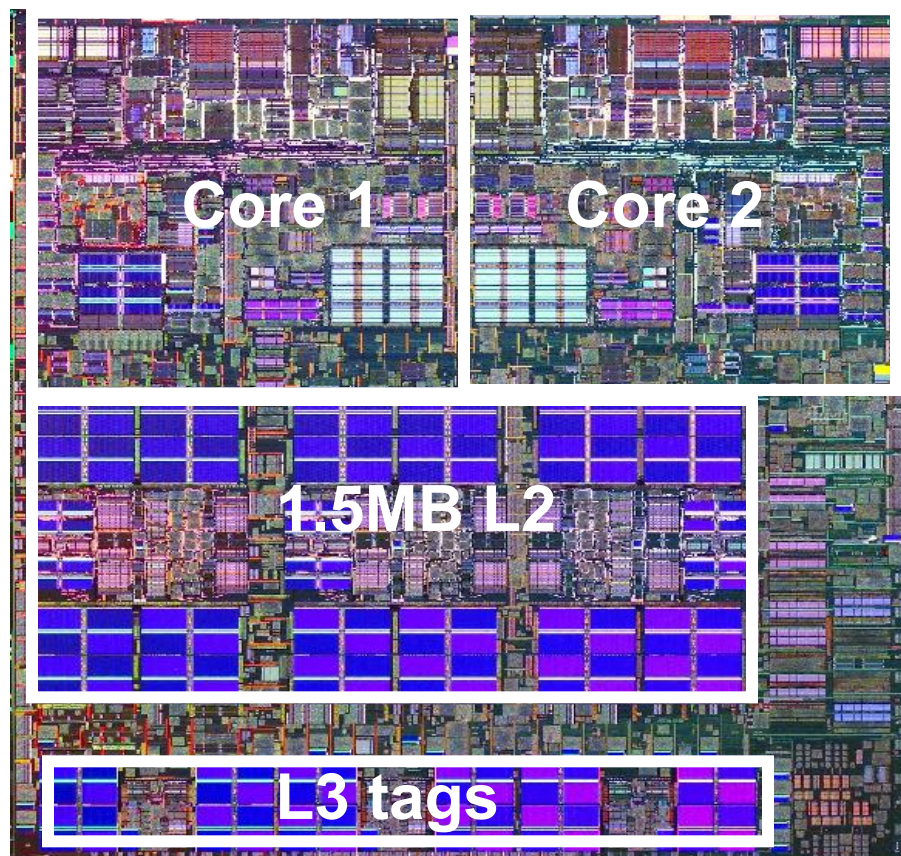
- A single core can only be so fast
 - Limited clock frequency
 - Limited instruction-level parallelism
- What if we need even more computing power?
 - Use multiple cores! But how?
- Old-school (2000s): Ultra Enterprise 25k
 - 72 dual-core UltraSPARC IV+ processors
 - Up to 1TB of memory
 - Niche: large database servers
 - \$\$\$, weights more than 1 ton
- Today: multicore is everywhere
 - Dual-core ARM phones



Intel Quad-Core "Core i7"



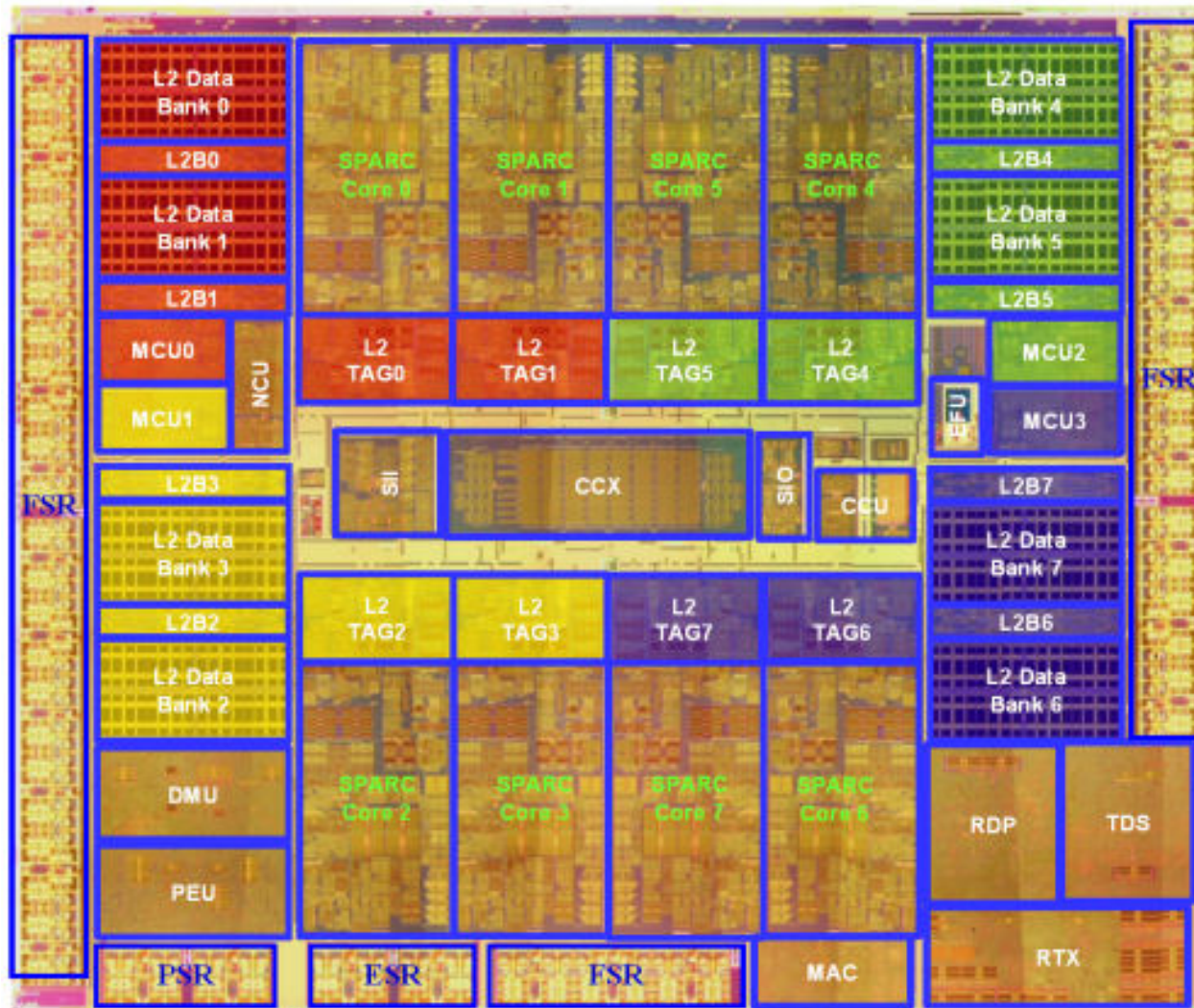
Multicore: Mainstream Multiprocessors



Why multicore? What else would you do with 1 billion transistors?

- **Multicore chips**
- **IBM Power5**
 - Two 2+GHz PowerPC cores
 - Shared 1.5 MB L2, L3 tags
- **AMD Quad Phenom**
 - Four 2+ GHz cores
 - Per-core 512KB L2 cache
 - Shared 2MB L3 cache
- **Intel Core i7 Quad**
 - Four cores, private L2s
 - Shared 8 MB L3
- **Sun Niagara**
 - 8 cores, each 4-way threaded
 - Shared 2MB L2
 - For servers, not desktop

Sun Niagara II



Application Domains for Multiprocessors

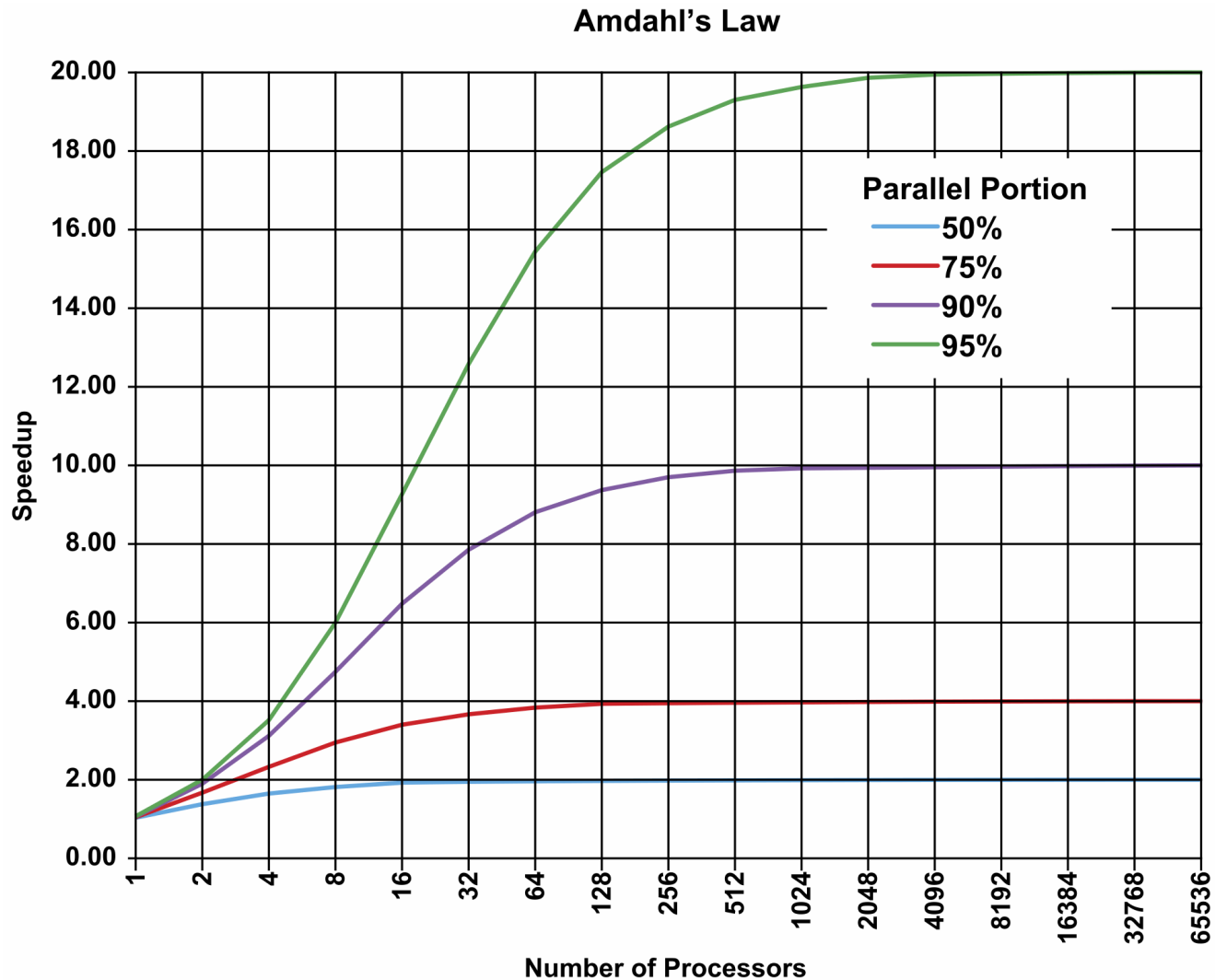
- **Scientific computing/supercomputing**
 - Examples: weather simulation, aerodynamics, protein folding
 - Large grids, integrating changes over time
 - Each processor computes for a part of the grid
- **Server workloads**
 - Example: airline reservation database
 - Many concurrent updates, searches, lookups, queries
 - Processors handle different requests
- **Media workloads**
 - Processors compress/decompress different parts of image/frames
- **Desktop workloads...**
- **Gaming workloads...**

But software must be written to expose parallelism

Amdahl's Law

- Restatement of the law of diminishing returns
 - Total speedup limited by non-accelerated piece
 - Analogy: drive to work & park car, walk to building
- Consider a task with a “parallel” and “serial” portion
 - What is the speedup with N cores?
 - $\text{Speedup}(n, p, s) = (s+p) / (s + (p/n))$
 - p is “parallel percentage”, s is “serial percentage”
 - What about infinite cores?
 - $\text{Speedup}(p, s) = (s+p) / s = 1 / s$
- Example: can optimize 50% of program A
 - Even “magic” optimization that makes this 50% disappear...
 - ...only yields a 2X speedup

Amdahl's Law Graph



“Threading” & The Shared Memory Execution Model

First, Uniprocessor Concurrency

- **Software “thread”**: Independent flows of execution
 - “Per-thread” state
 - Context state: PC, registers
 - Stack (per-thread local variables)
 - “Shared” state: globals, heap, etc.
 - Threads generally share the same memory space
 - “Process” like a thread, but different memory space
 - Java has thread support built in, C/C++ using a thread library
- Generally, system software (the O.S.) manages threads
 - “Thread scheduling”, “context switching”
 - In single-core system, all threads share the one processor
 - Hardware timer interrupt occasionally triggers O.S.
 - Quickly swapping threads gives illusion of concurrent execution
 - Much more in an operating systems course

Multithreaded Programming Model

- Programmer explicitly creates multiple threads
- All loads & stores to a single **shared memory** space
 - Each thread has its own stack frame for local variables
 - All memory shared, accessible by all threads
- A “thread switch” can occur at any time
 - Pre-emptive multithreading by OS
- Common uses:
 - Handling user interaction (GUI programming)
 - Handling I/O latency (send network message, wait for response)
 - **Expressing parallel work via Thread-Level Parallelism (TLP)**
 - This is our focus!

Shared Memory Model: Interleaving

- **Initially: all variables zero** (that is, x is 0, y is 0)

thread 1	thread 2
store 1 → y load x	store 1 → x load y

- What value pairs can be read by the two loads?

Shared Memory Model: Interleaving

- Initially: all variables zero (that is, x is 0, y is 0)

thread 1	thread 2
store 1 → y load x	store 1 → x load y

- What value pairs can be read by the two loads?

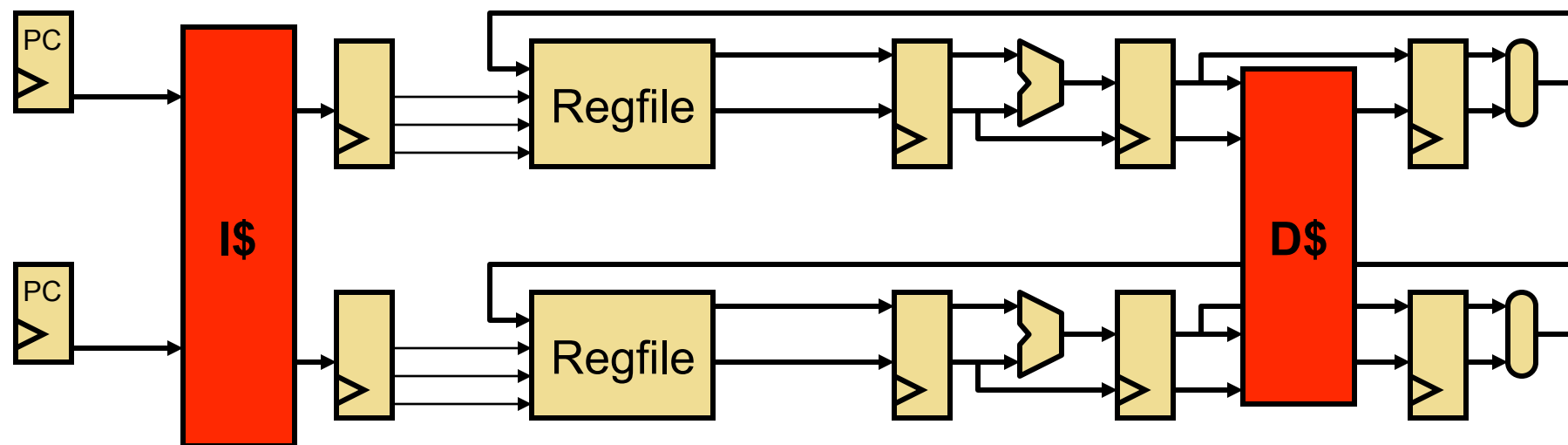
store 1 → y load x store 1 → x load y (x=0, y=1)	store 1 → y store 1 → x load x load y (x=1, y=1)	store 1 → y store 1 → x load y load x (x=1, y=1)
store 1 → x load y store 1 → y load x (x=1, y=0)	store 1 → x store 1 → y load y load x (x=1, y=1)	store 1 → x store 1 → y load x load y (x=1, y=1)

- What about (x=0, y=0)?

Shared Memory Implementations

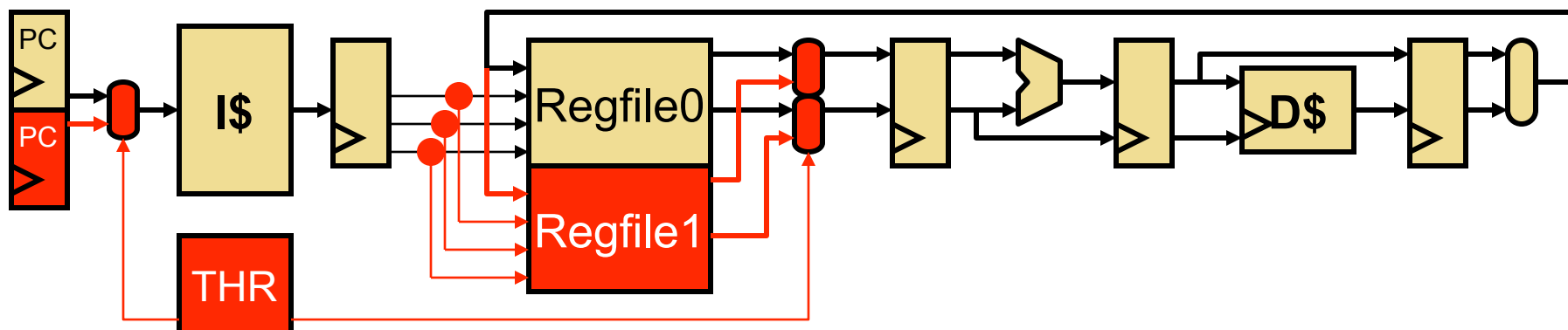
- **Multiplexed uniprocessor**
 - Runtime system and/or OS occasionally pre-empt & swap threads
 - Interleaved, but no parallelism
- **Multiprocessing**
 - Multiply execution resources, higher peak performance
 - Same interleaved shared-memory model
 - Foreshadowing: allow private caches, further disentangle cores
- **Hardware multithreading**
 - Tolerate pipeline latencies, higher efficiency
 - Same interleaved shared-memory model
- **All support the shared memory programming model**

Simplest Multiprocessor



- Replicate entire processor pipeline!
 - Instead of replicating just register file & PC
 - Exception: share the caches (we'll address this bottleneck soon)
- Multiple threads execute
 - Shared memory programming model
 - Operations (loads and stores) are interleaved "at random"
 - Loads returns the value written by most recent store to location

Hardware Multithreading



- **Hardware Multithreading (MT)**
 - Multiple threads dynamically share a single pipeline
 - Replicate only per-thread structures: program counter & registers
 - Hardware interleaves instructions
- + **Multithreading improves utilization and throughput**
 - Single programs utilize <50% of pipeline (branch, cache miss)
- **Multithreading does not improve single-thread performance**
 - Individual threads run as fast or even slower
- **Coarse-grain MT**: switch on cache misses Why?
- **Simultaneous MT**: no explicit switching, fine-grain interleaving

Four Shared Memory Issues

1. Cache coherence

- If cores have private (non-shared) caches
- How to make writes to one cache “show up” in others?

2. Parallel programming

- How does the programmer express the parallelism?

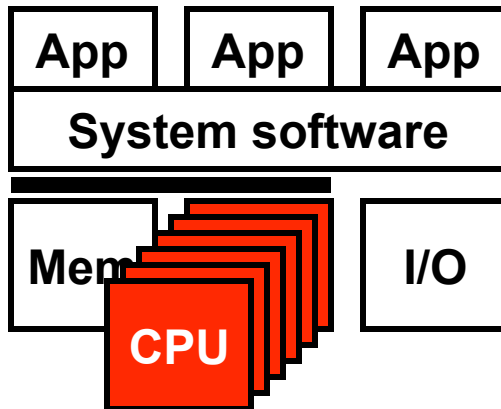
3. Synchronization

- How to regulate access to shared data?
- How to implement “locks”?

4. Memory consistency models

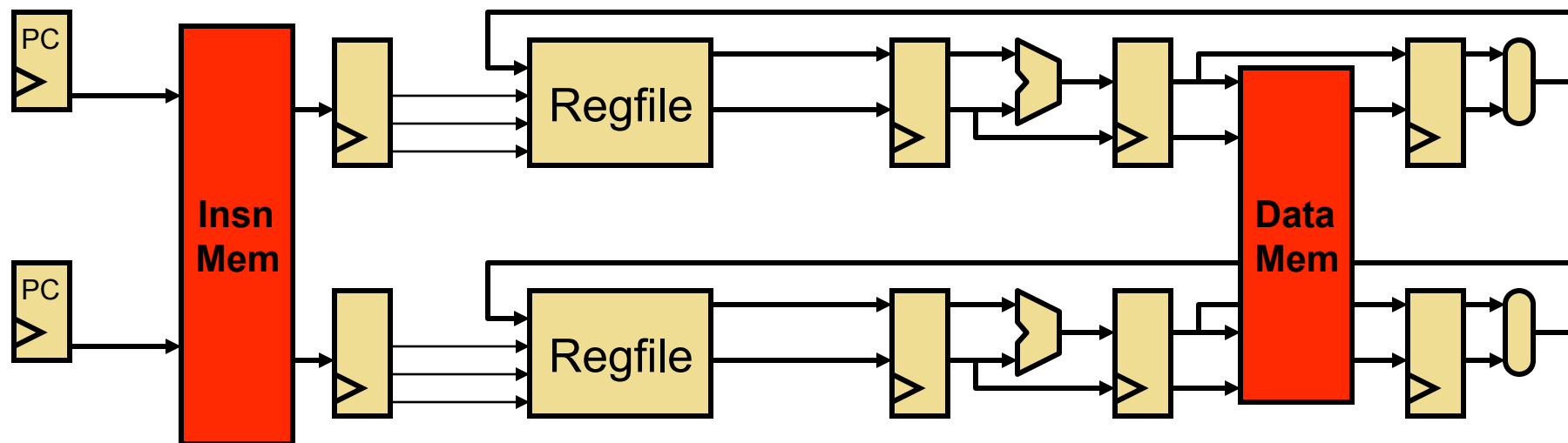
- How to keep programmer sane while letting hardware optimize?
- How to reconcile shared memory with compiler optimizations, store buffers, and out-of-order execution?

Roadmap Checkpoint



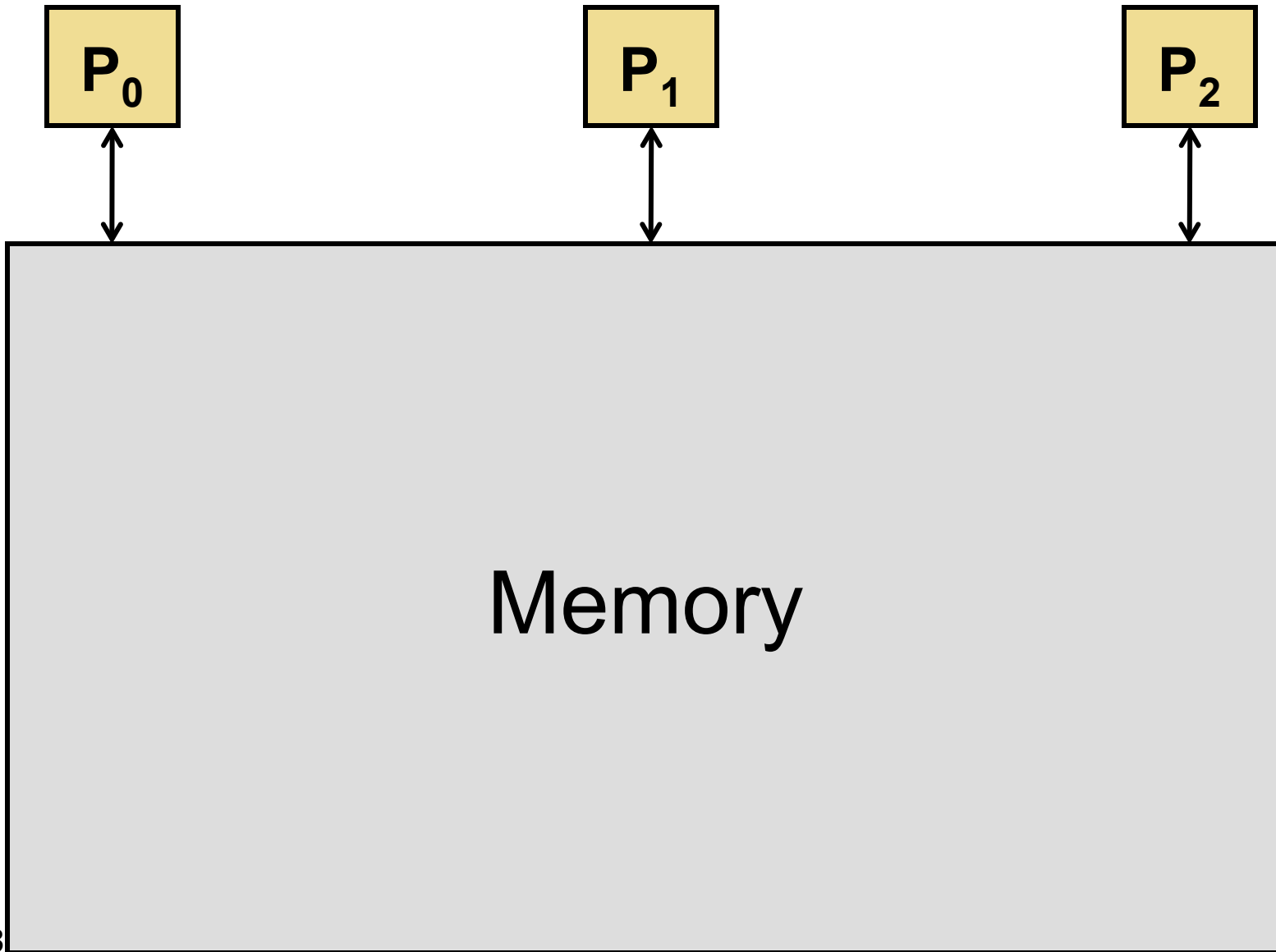
- ~~Thread-level parallelism (TLP)~~
- ~~Shared memory model~~
 - ~~Multiplexed uniprocessor~~
 - ~~Hardware multithreading~~
 - ~~Multiprocessing~~
- **Cache coherence**
 - **Valid/Invalid, MSI, MESI**
- Parallel programming
- Synchronization
 - Lock implementation
 - Locking gotchas
 - Transactional memory
- Memory consistency models

Recall: Simplest Multiprocessor

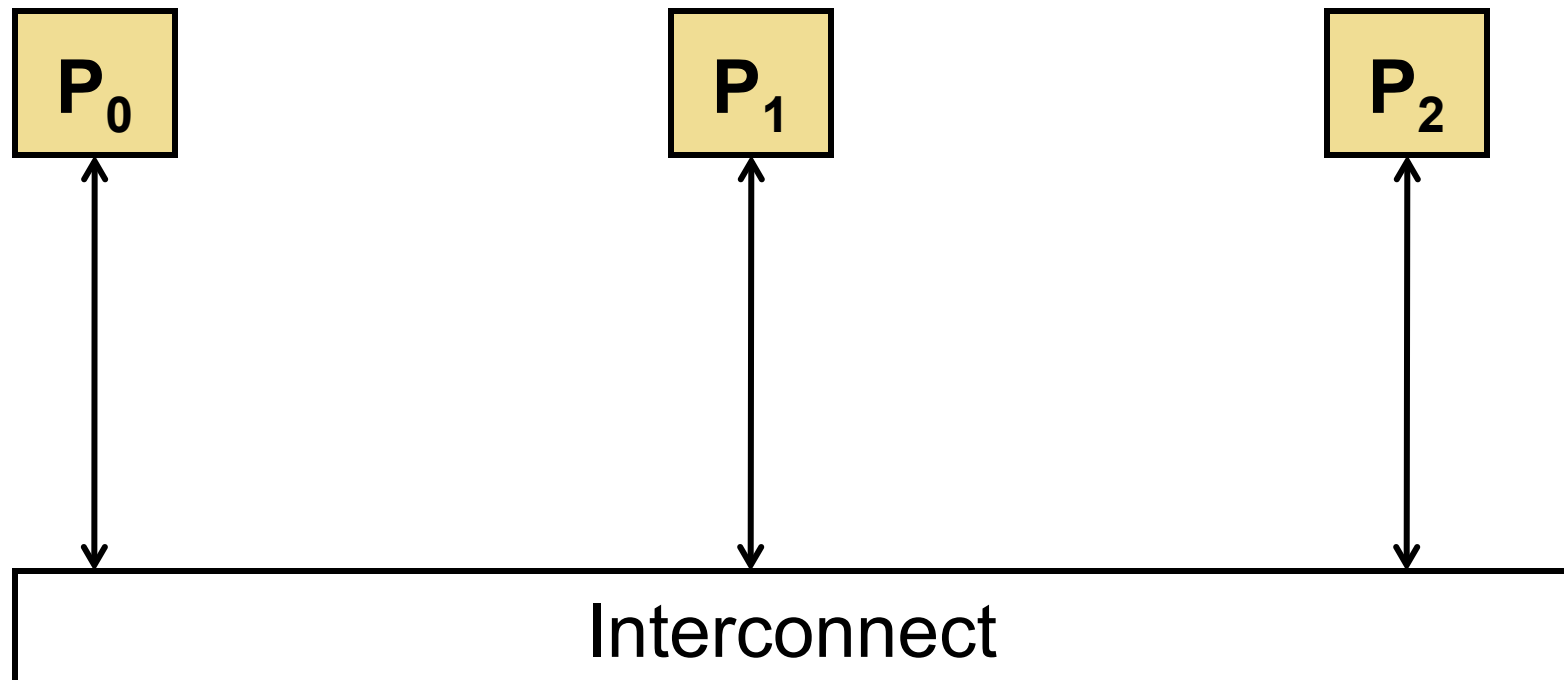


- What if we don't want to share the L1 caches?
 - Bandwidth and latency issue
- Solution: use per-processor ("private") caches
 - Coordinate them with a **Cache Coherence Protocol**
- Must still provide shared-memory invariant:
 - **"Loads read the value written by the most recent store"**

No-Cache (Conceptual) Implementation



No-Cache (Conceptual) Implementation

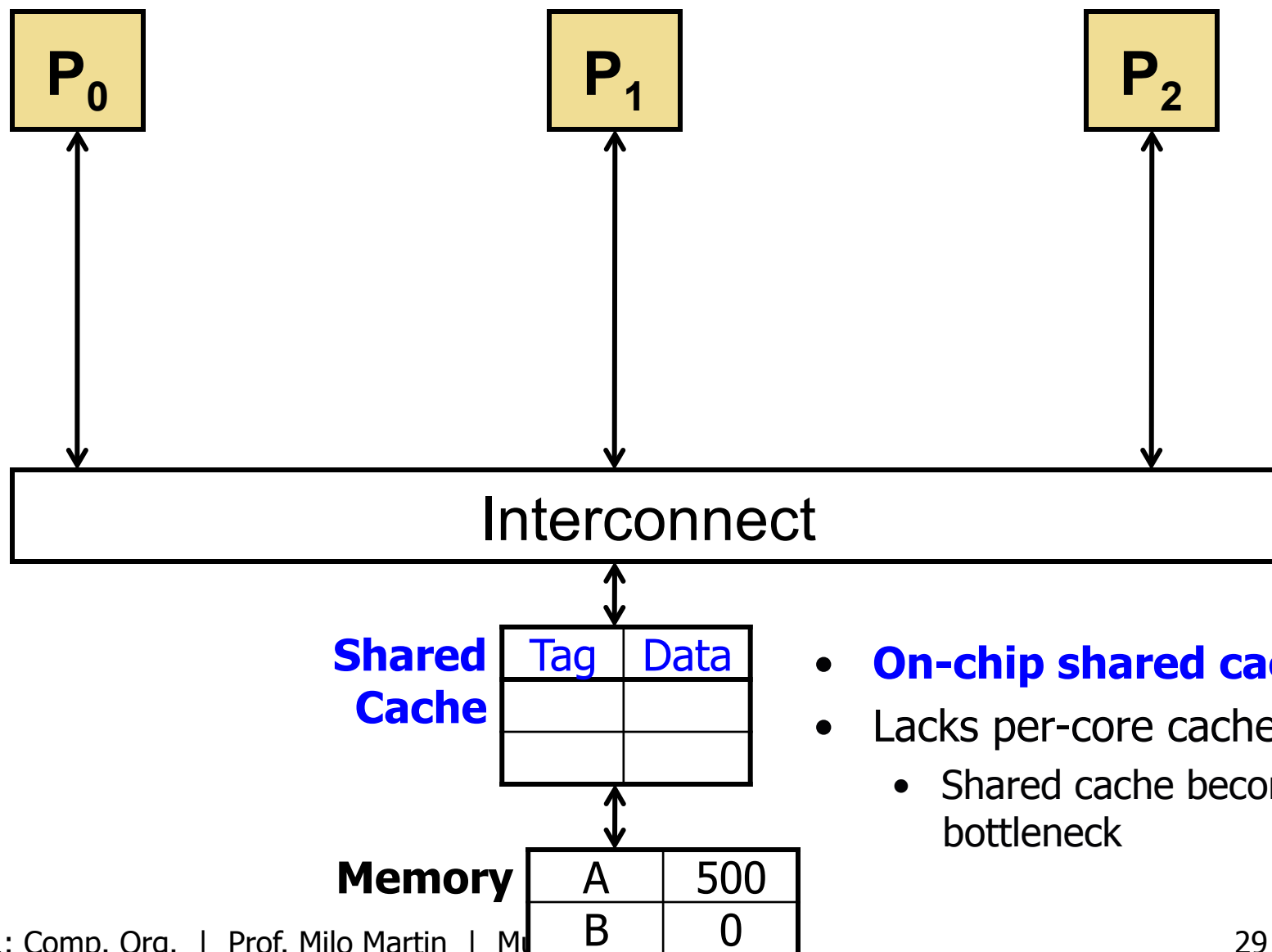


- **No caches**
 - Not a realistic design

Memory

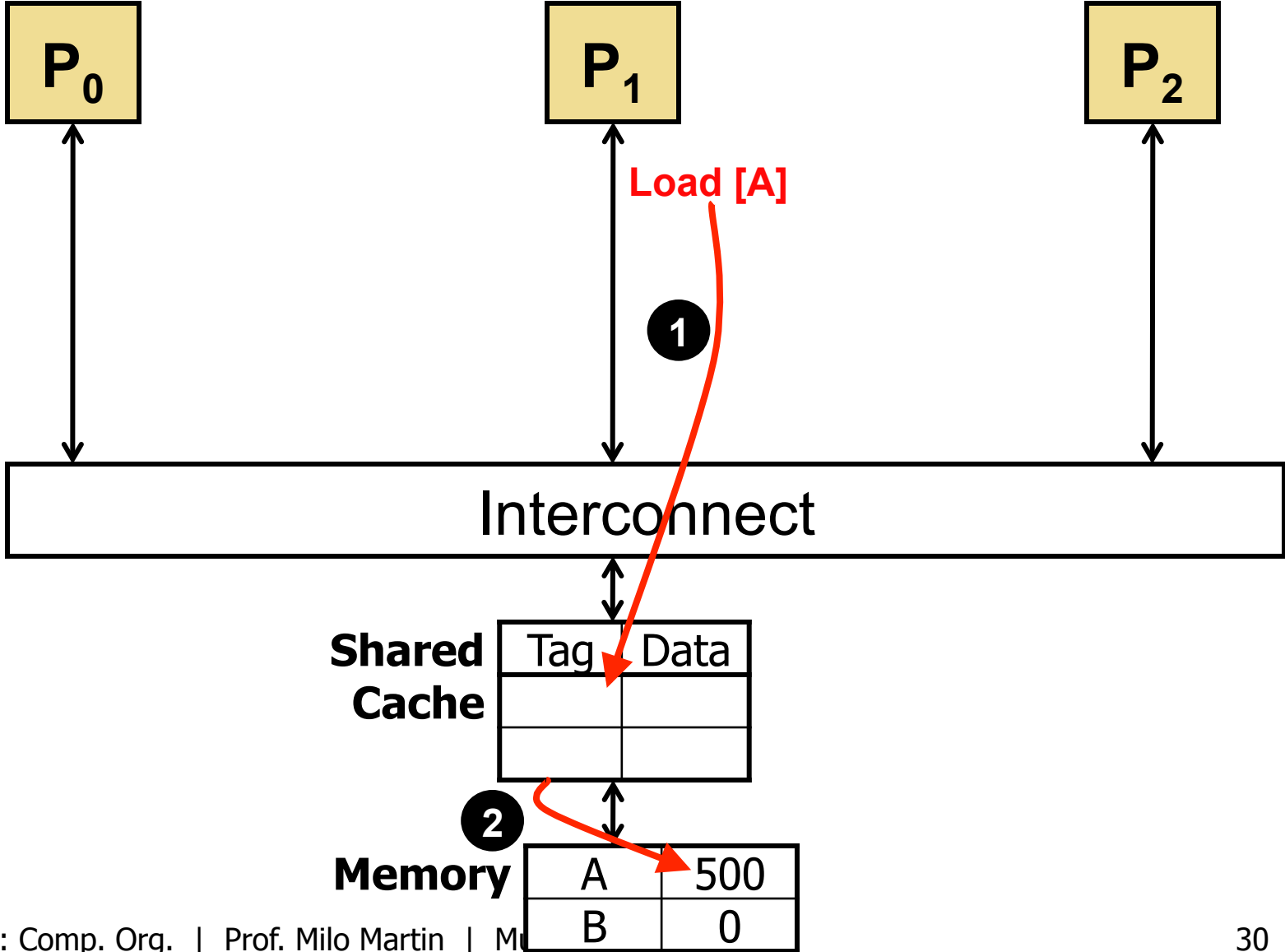
A	500
B	0

Shared Cache Implementation

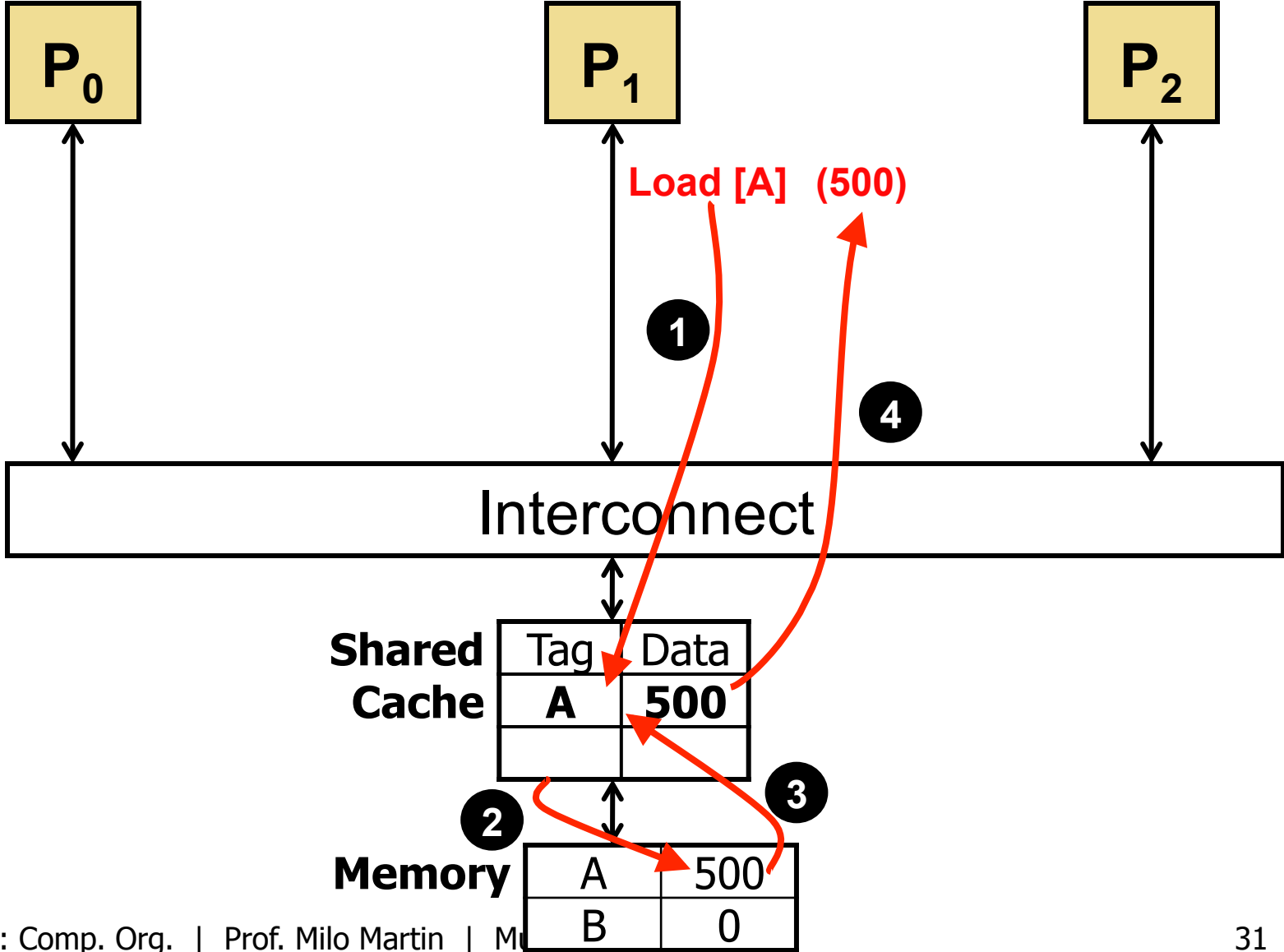


- **On-chip shared cache**
- Lacks per-core caches
 - Shared cache becomes bottleneck

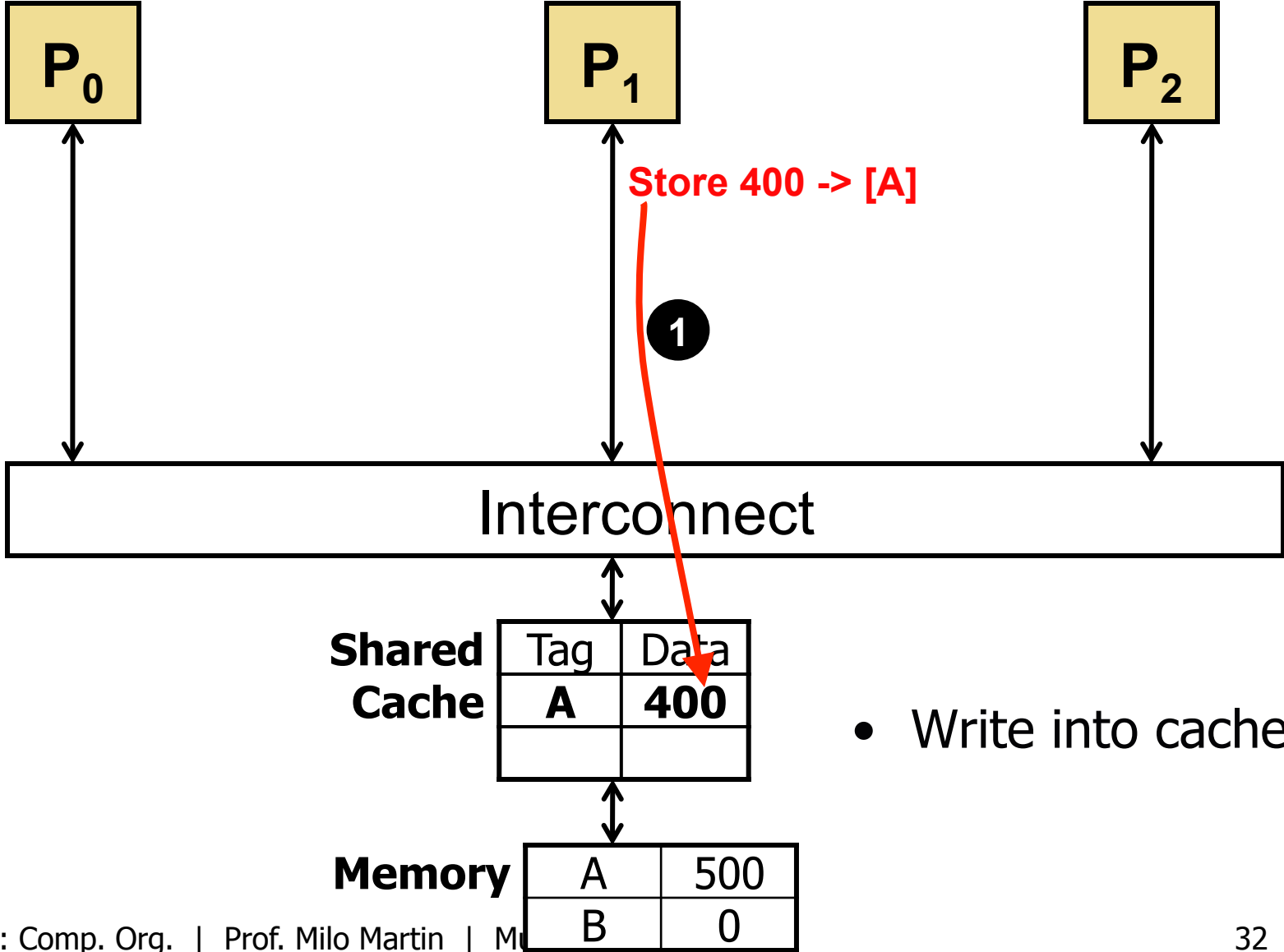
Shared Cache Implementation



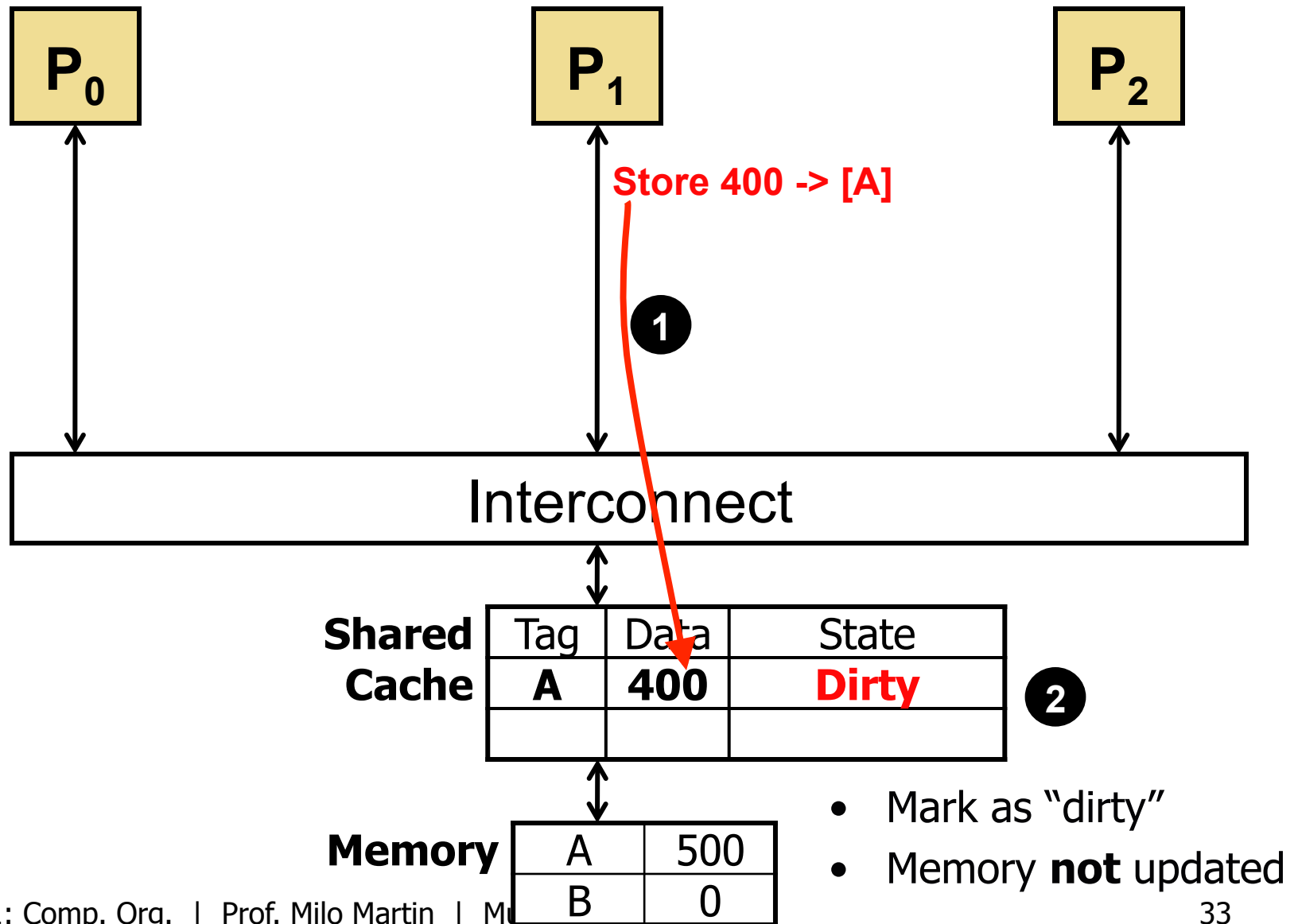
Shared Cache Implementation



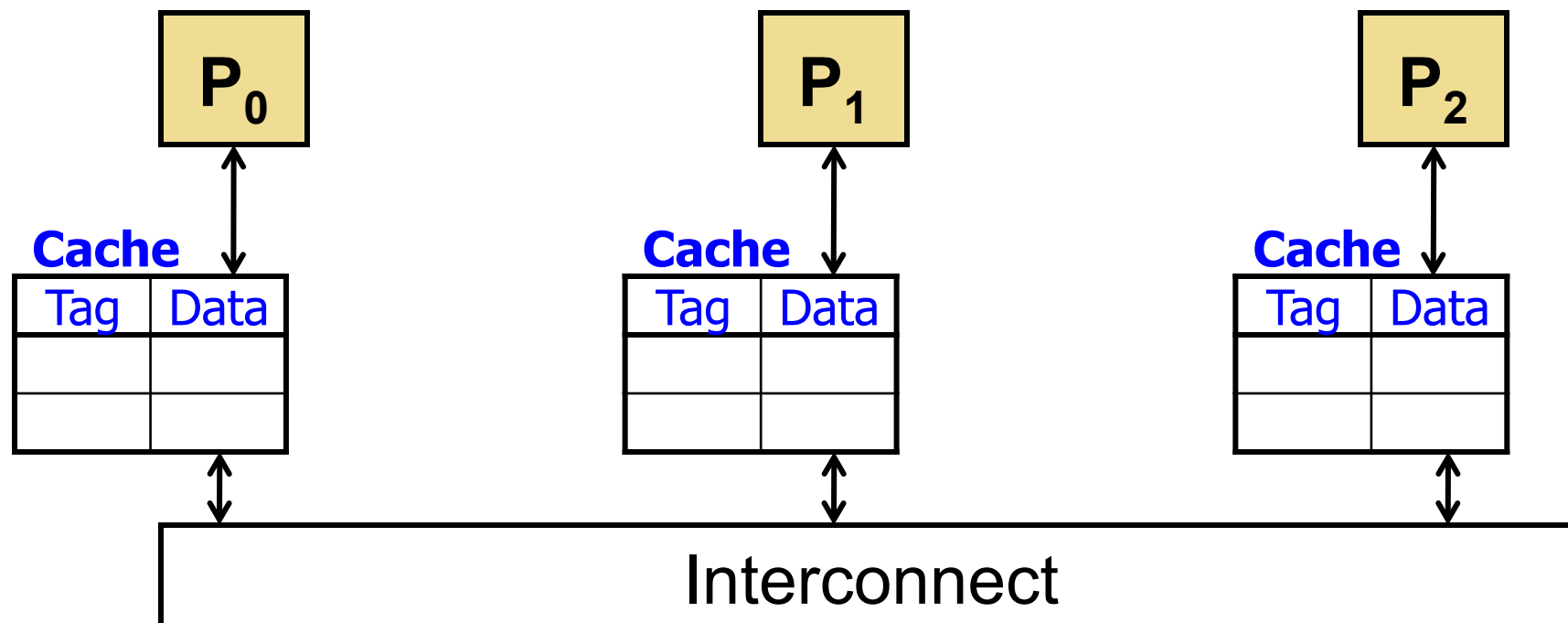
Shared Cache Implementation



Shared Cache Implementation



Adding Private Caches



- **Add per-core caches**

(write-back caches)

- Reduces latency
- Increases throughput
- Decreases energy

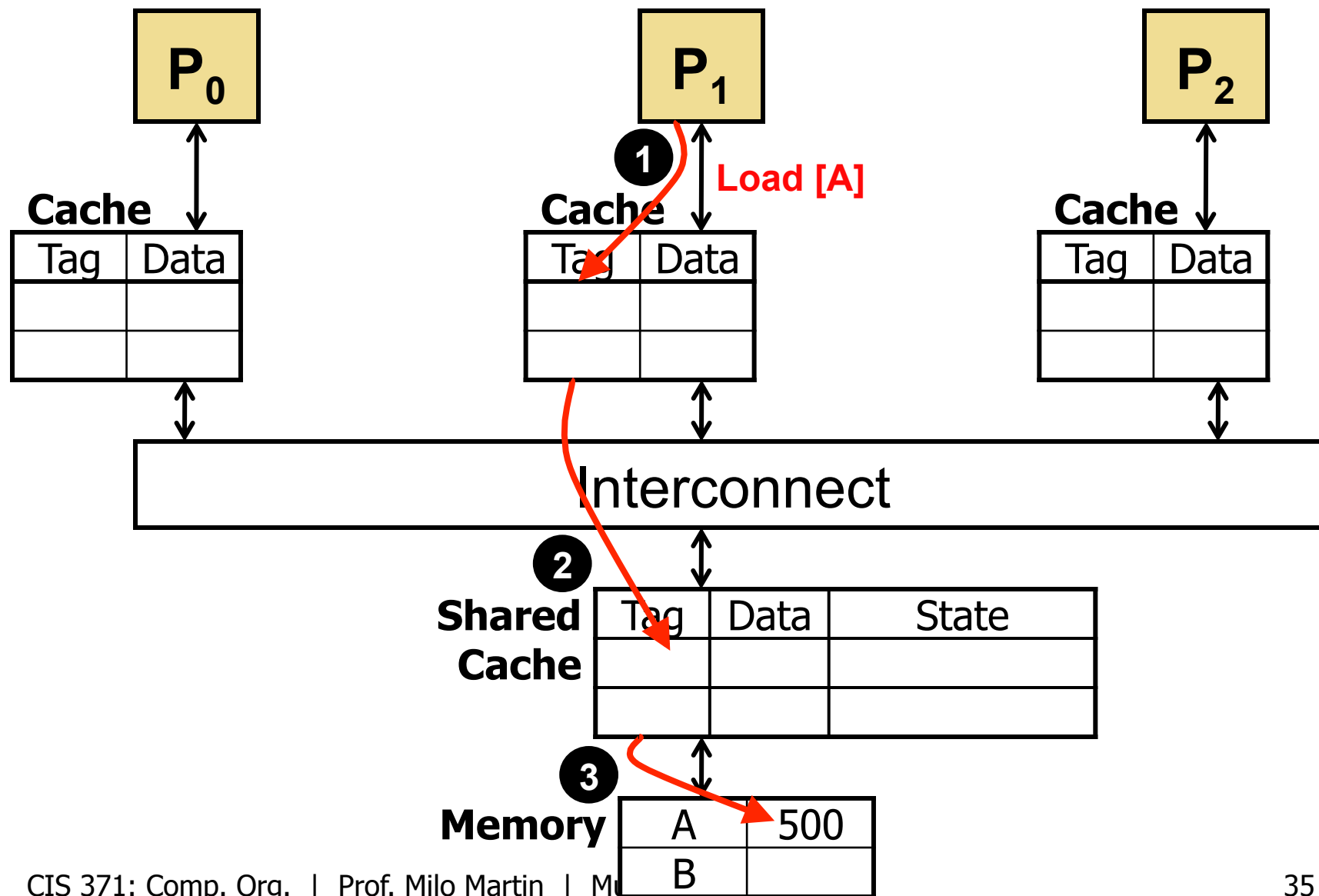
**Shared
Cache**

Tag	Data	State

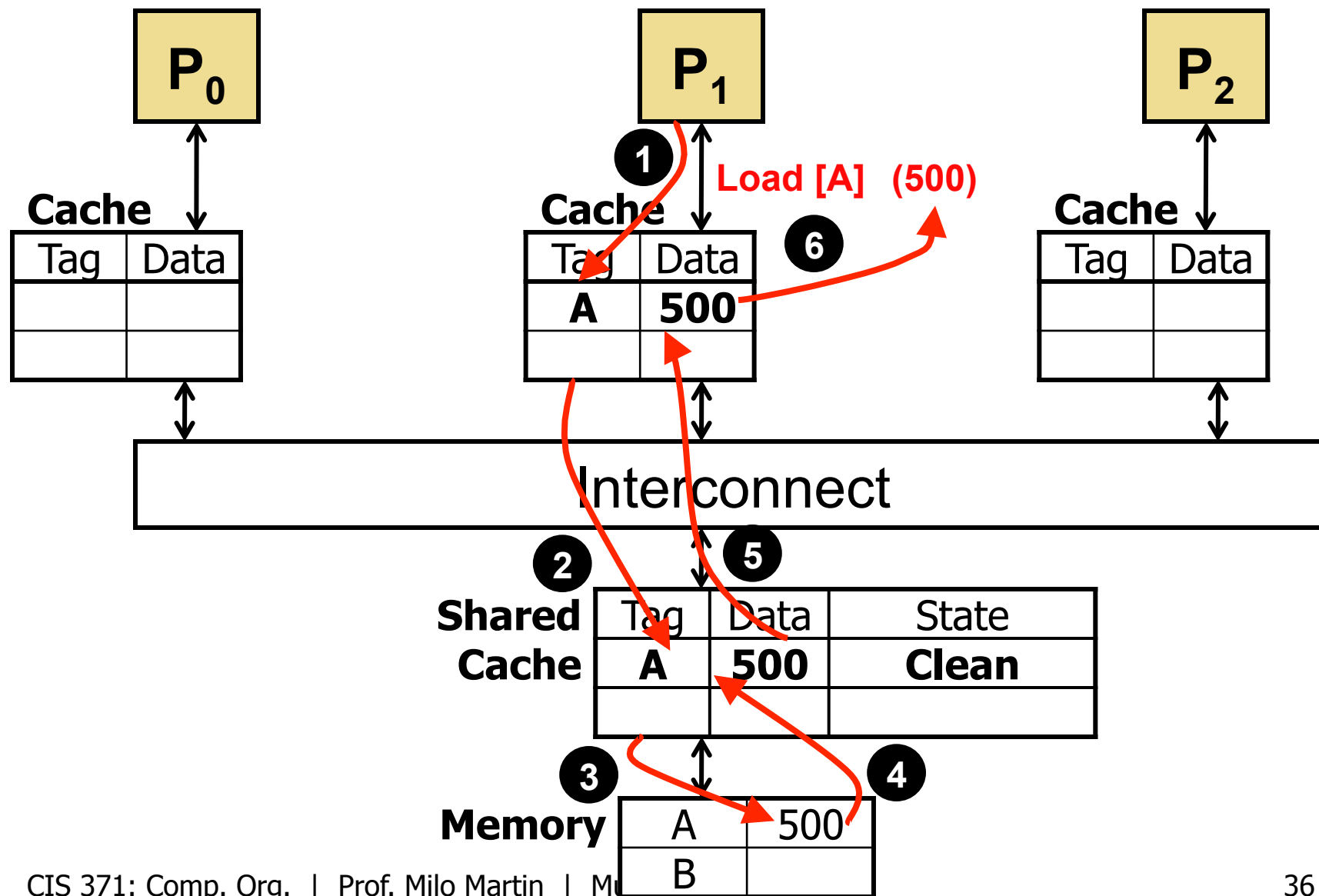
Memory

A	
B	

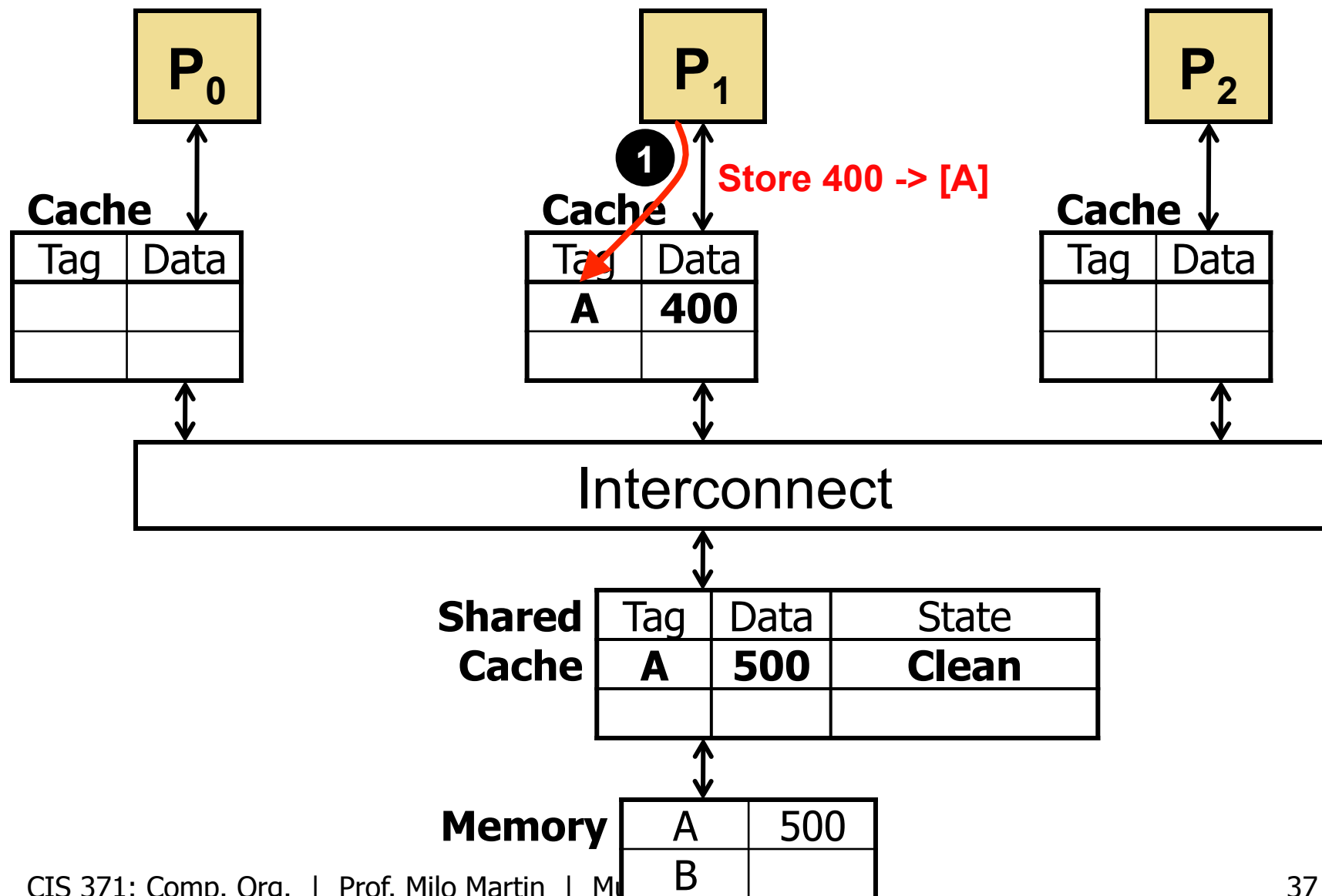
Adding Private Caches



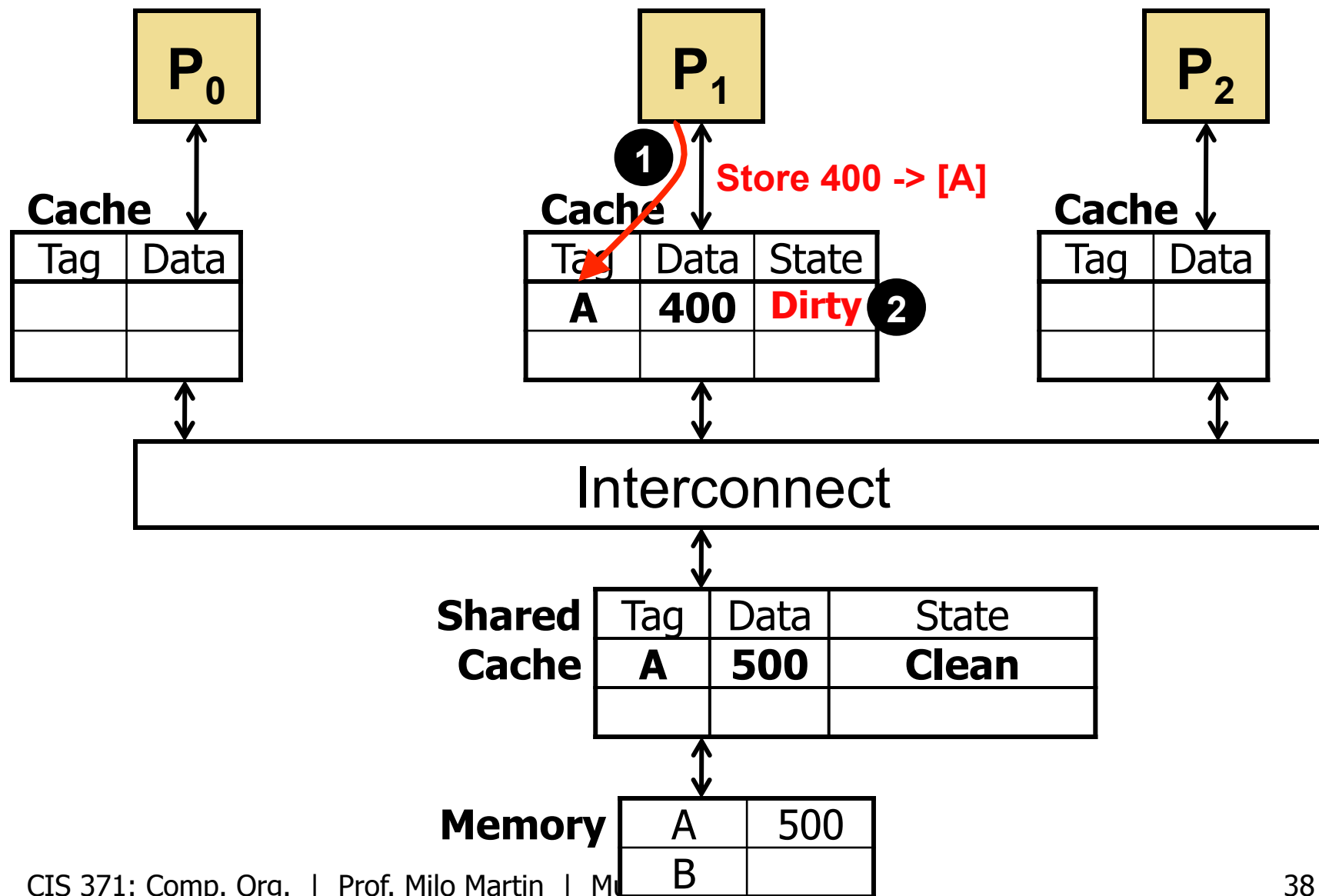
Adding Private Caches



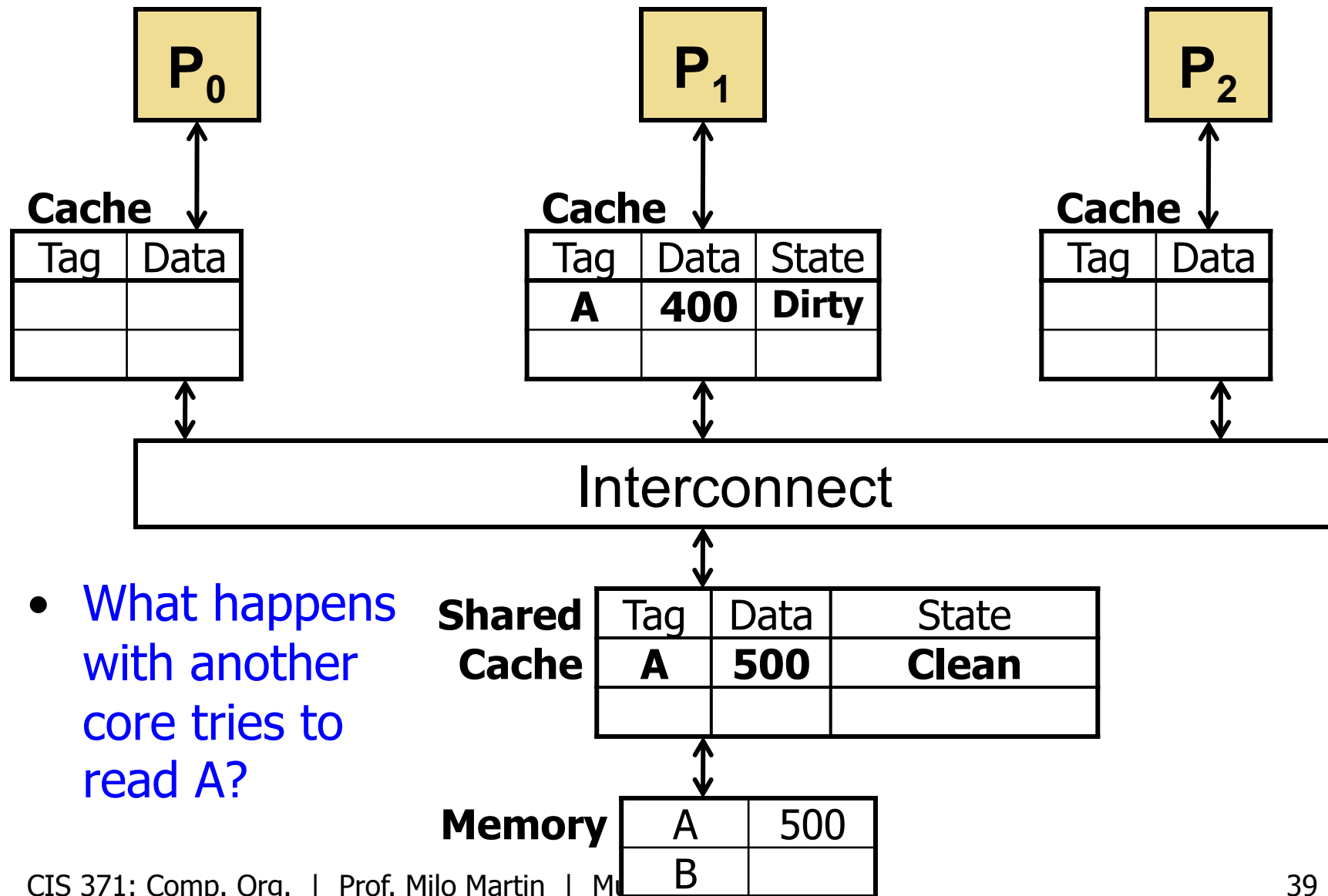
Adding Private Caches



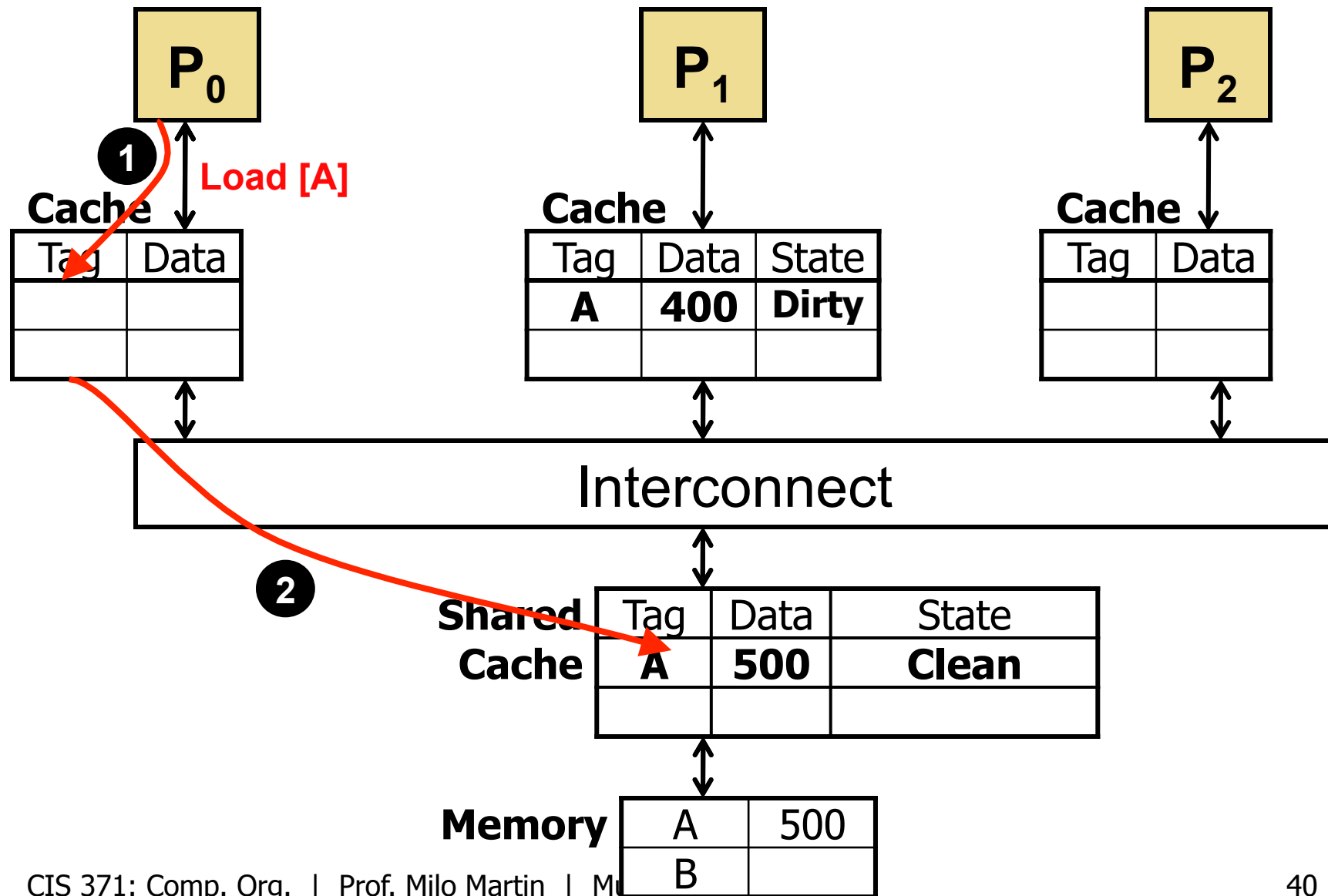
Adding Private Caches



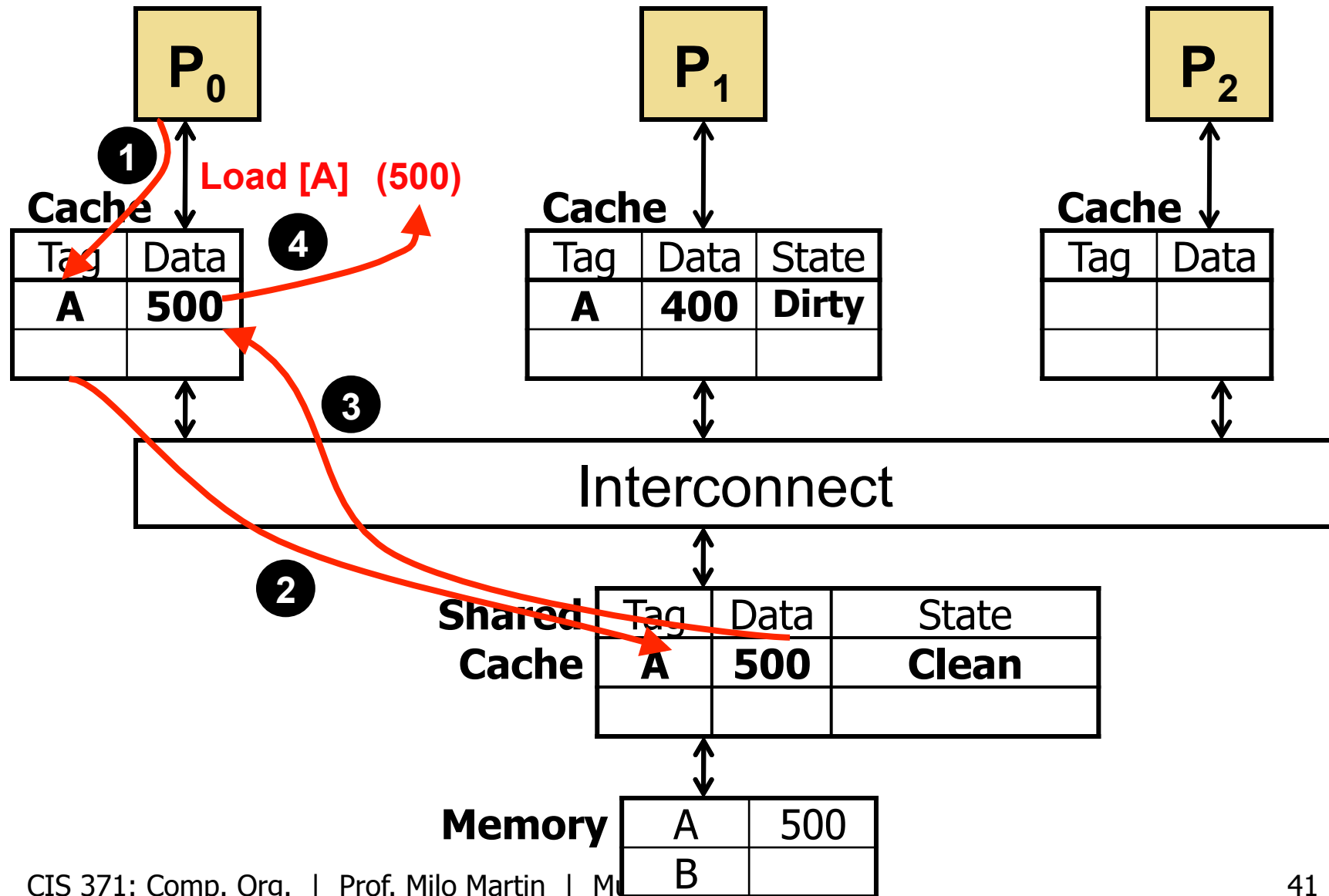
Private Cache Problem: Incoherence



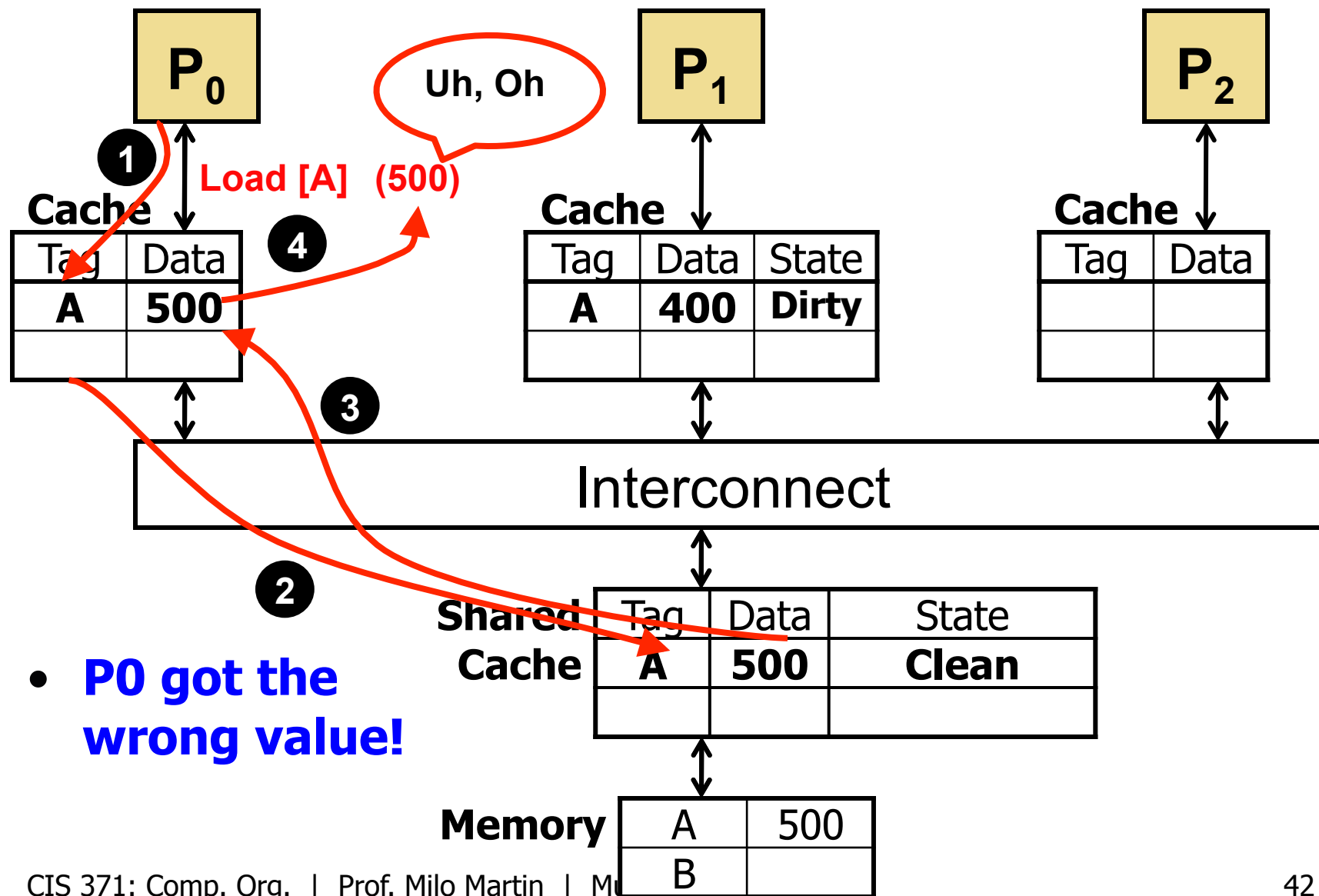
Private Cache Problem: Incoherence



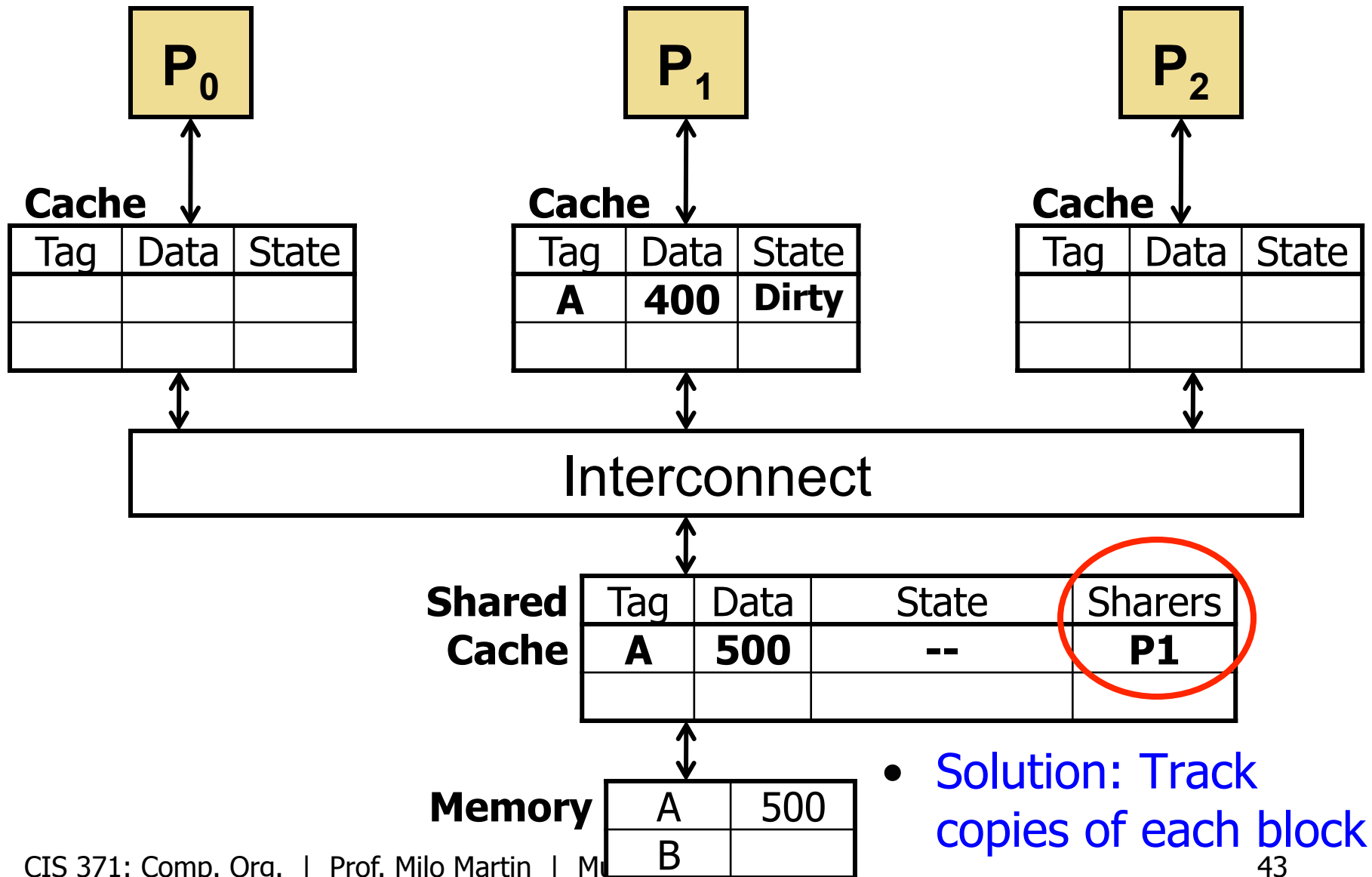
Private Cache Problem: Incoherence



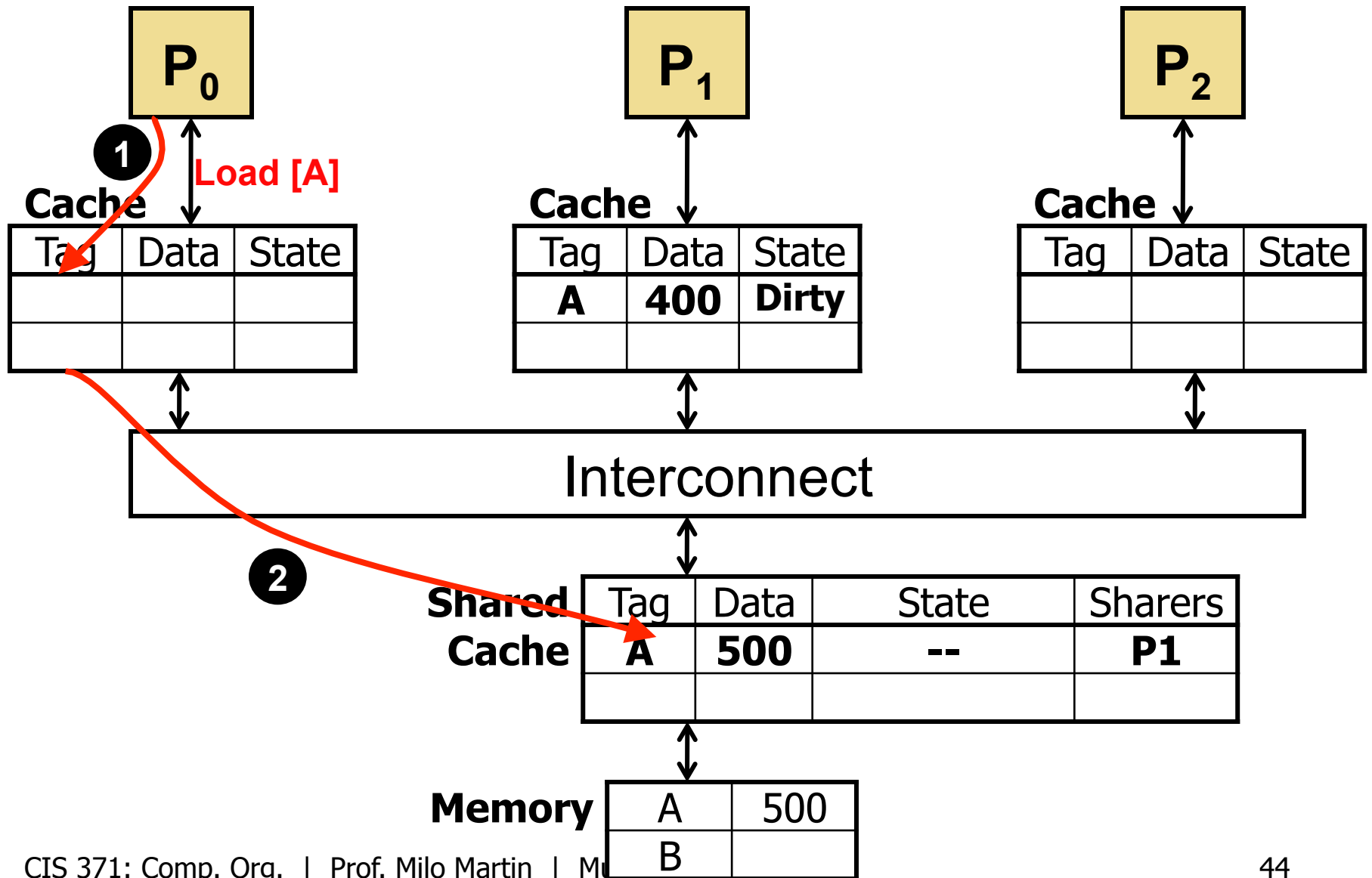
Private Cache Problem: Incoherence



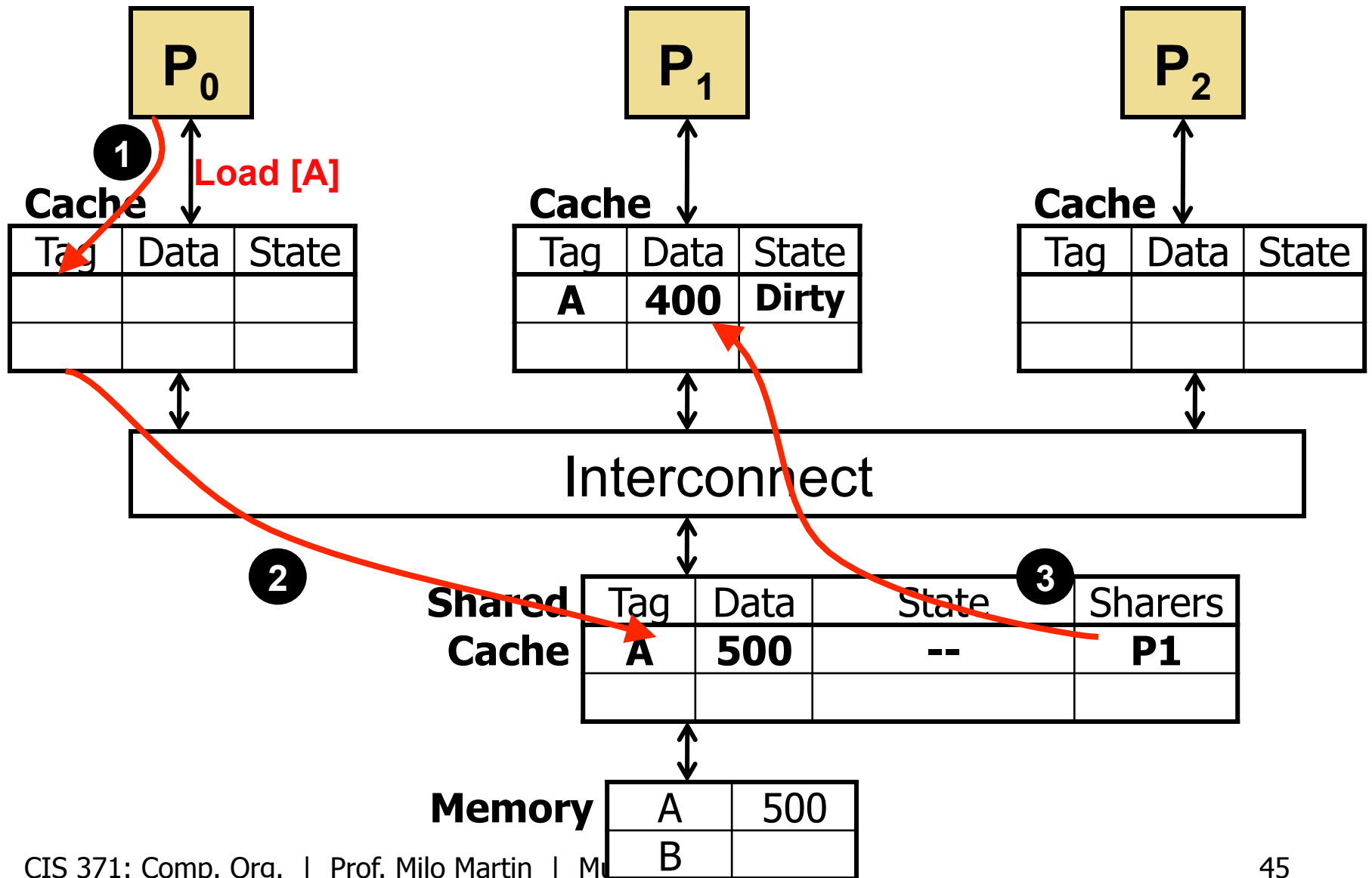
Rewind: Fix Problem by Tracking Sharers



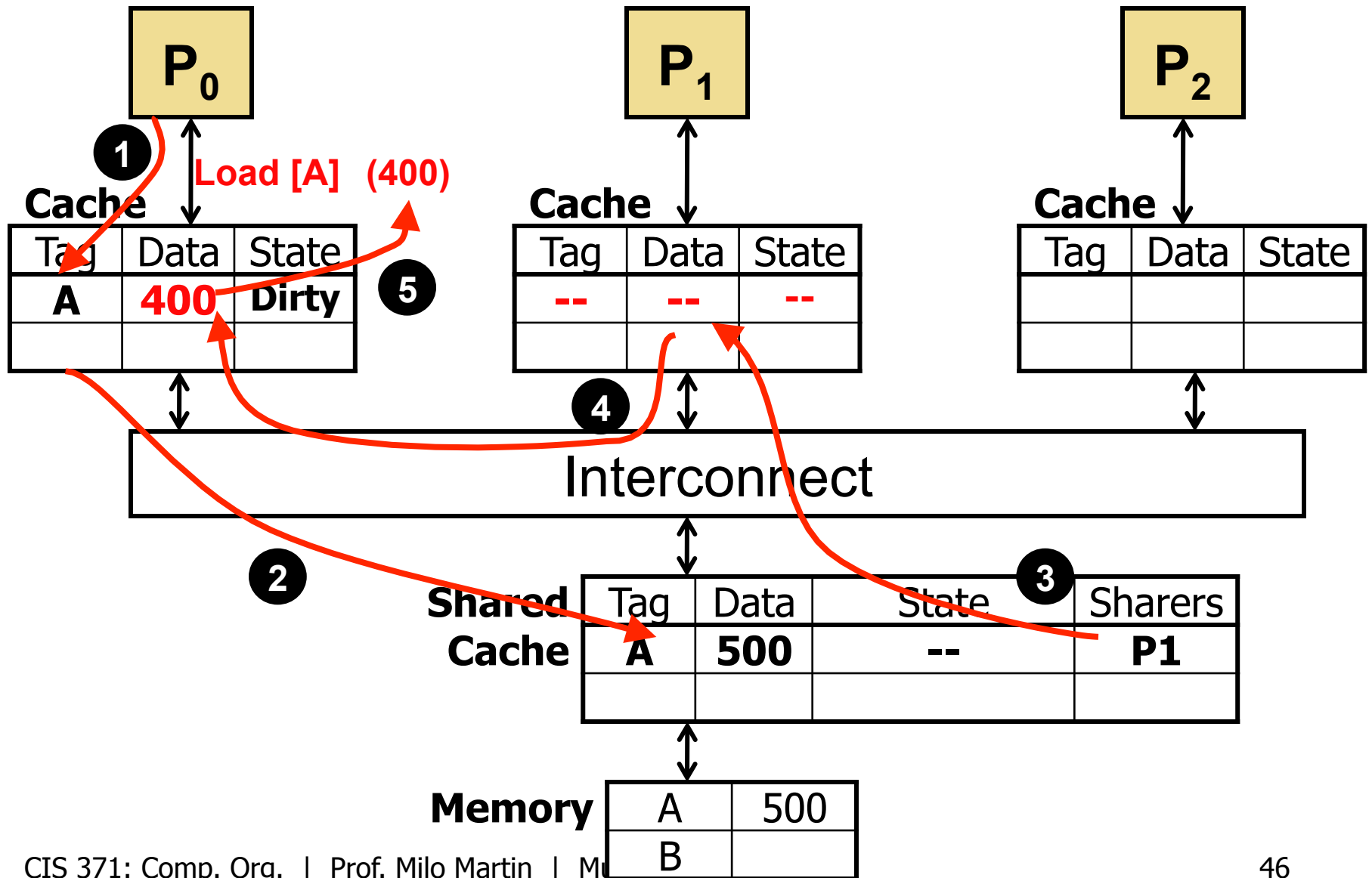
Use Tracking Information to "Invalidate"



Use Tracking Information to "Invalidate"



Use Tracking Information to "Invalidate"



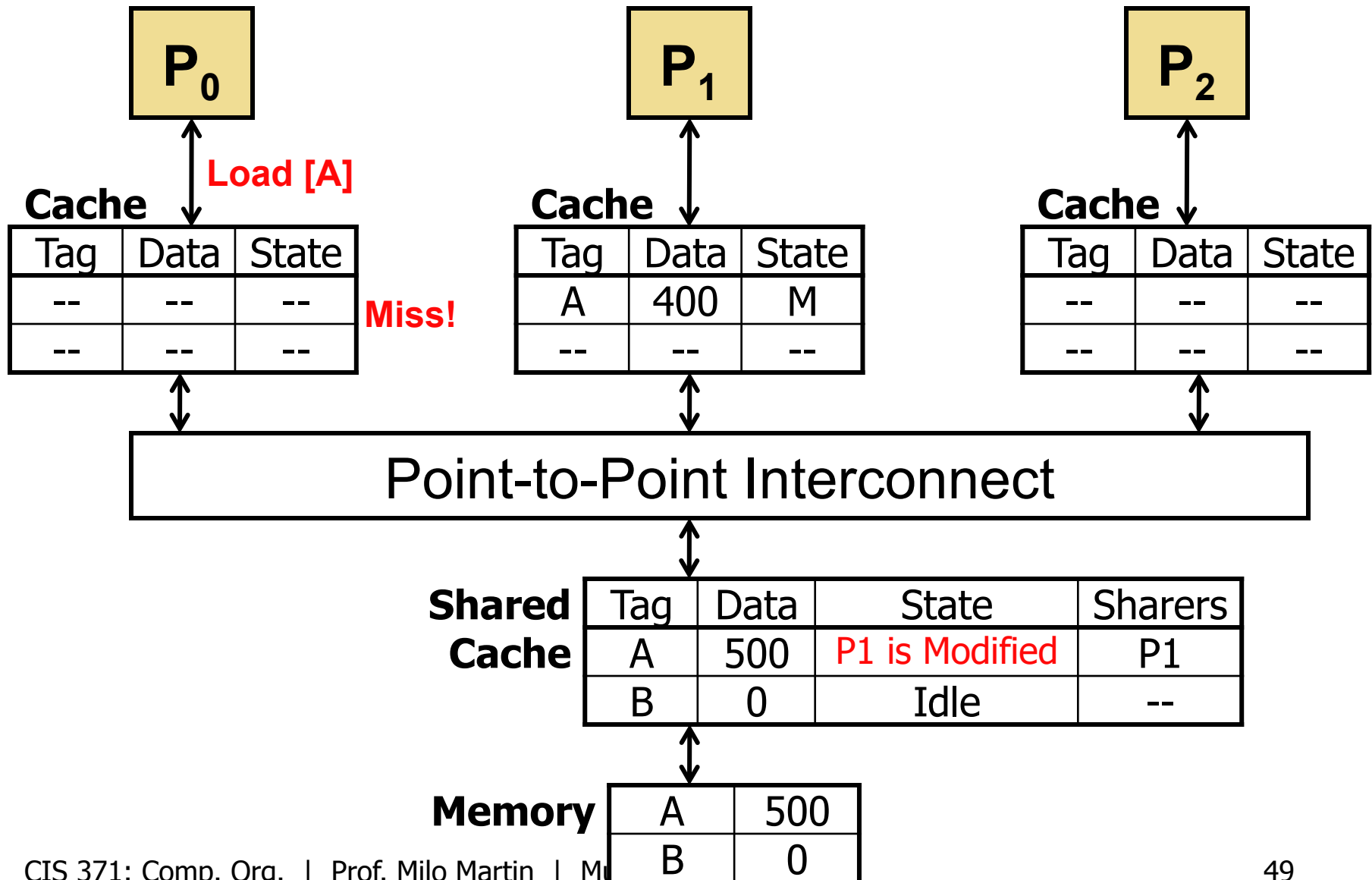
“Valid/Invalid” Cache Coherence

- To enforce the shared memory invariant...
 - “Loads read the value written by the most recent store”
- Enforce the invariant...
 - **“At most one valid copy of the block”**
 - Simplest form is a **two-state “valid/invalid” protocol**
 - If a core wants a copy, must find and “invalidate” it
- On a cache miss, how is the valid copy found?
 - Option #1 **“Snooping”**: broadcast to all, whoever has it responds
 - Option #2: **“Directory”**: tracker sharers at known location
- **Problem**: multiple copies can’t exist, even if read-only
 - Consider mostly-read data structures, instructions, etc.

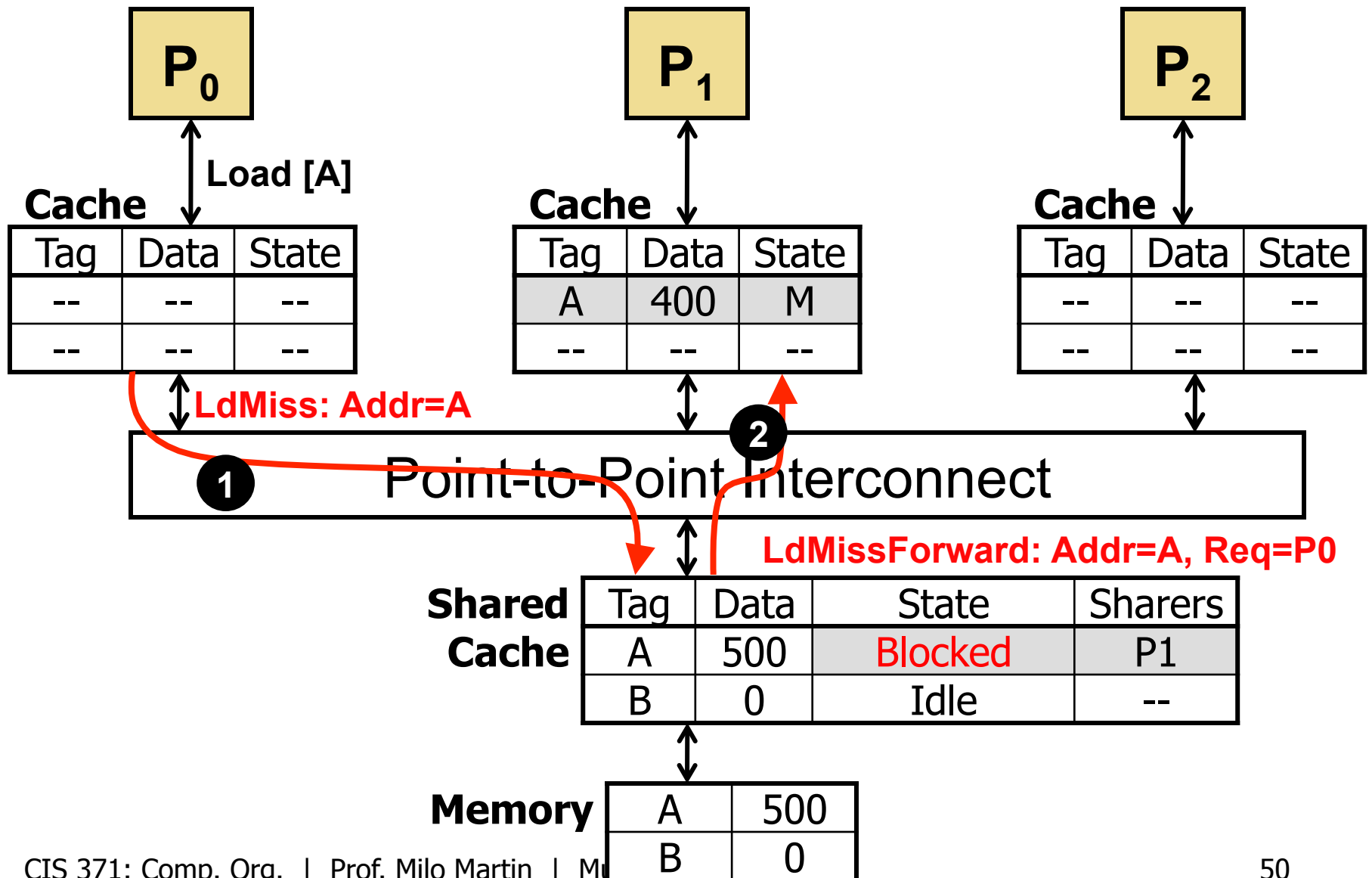
MSI Cache Coherence Protocol

- Solution: enforce the invariant...
 - **Multiple read-only copies** —OR—
 - **Single read/write copy**
- Track these MSI permissions (states) in per-core caches
 - **Modified (M): read/write permission**
 - **Shared (S): read-only permission**
 - **Invalid (I): no permission**
- Also track a **“Sharer” bit vector** in shared cache
 - One bit per core; tracks all shared copies of a block
 - Then, *invalidate all readers* when a write occurs
- Allows for many readers...
 - ...while still enforcing shared memory invariant (“Loads read the value written by the most recent store”)

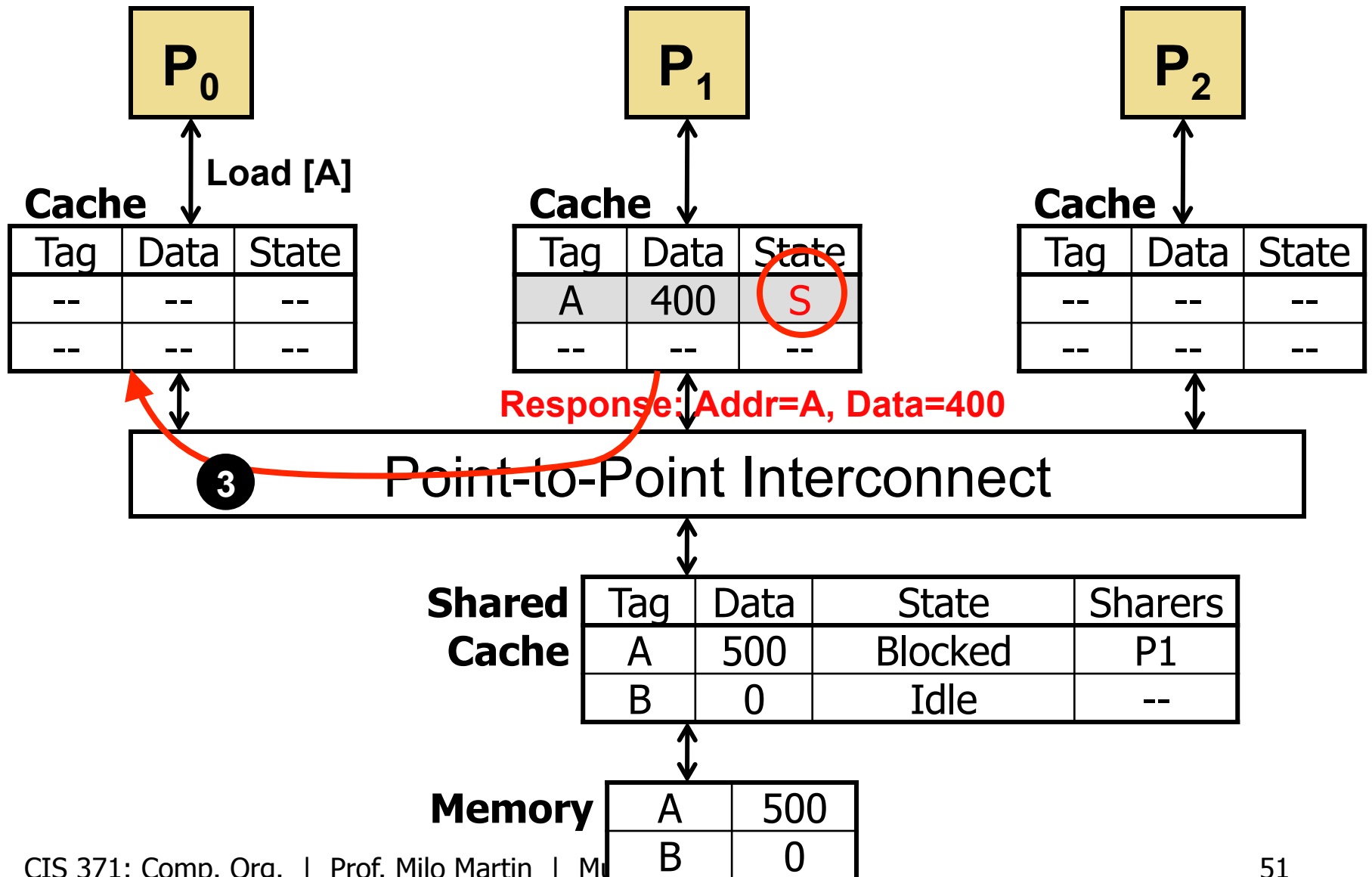
MSI Coherence Example: Step #1



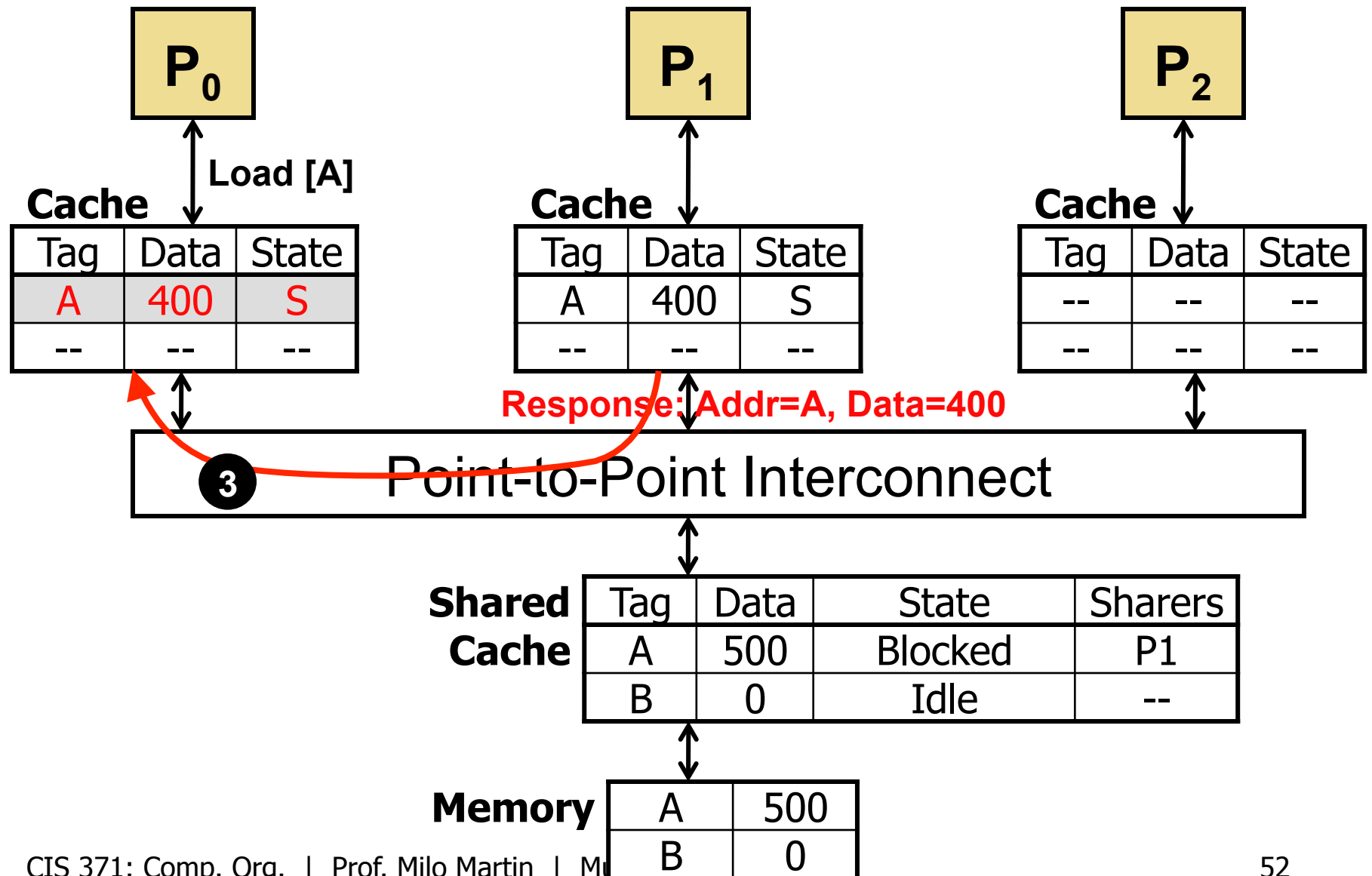
MSI Coherence Example: Step #2



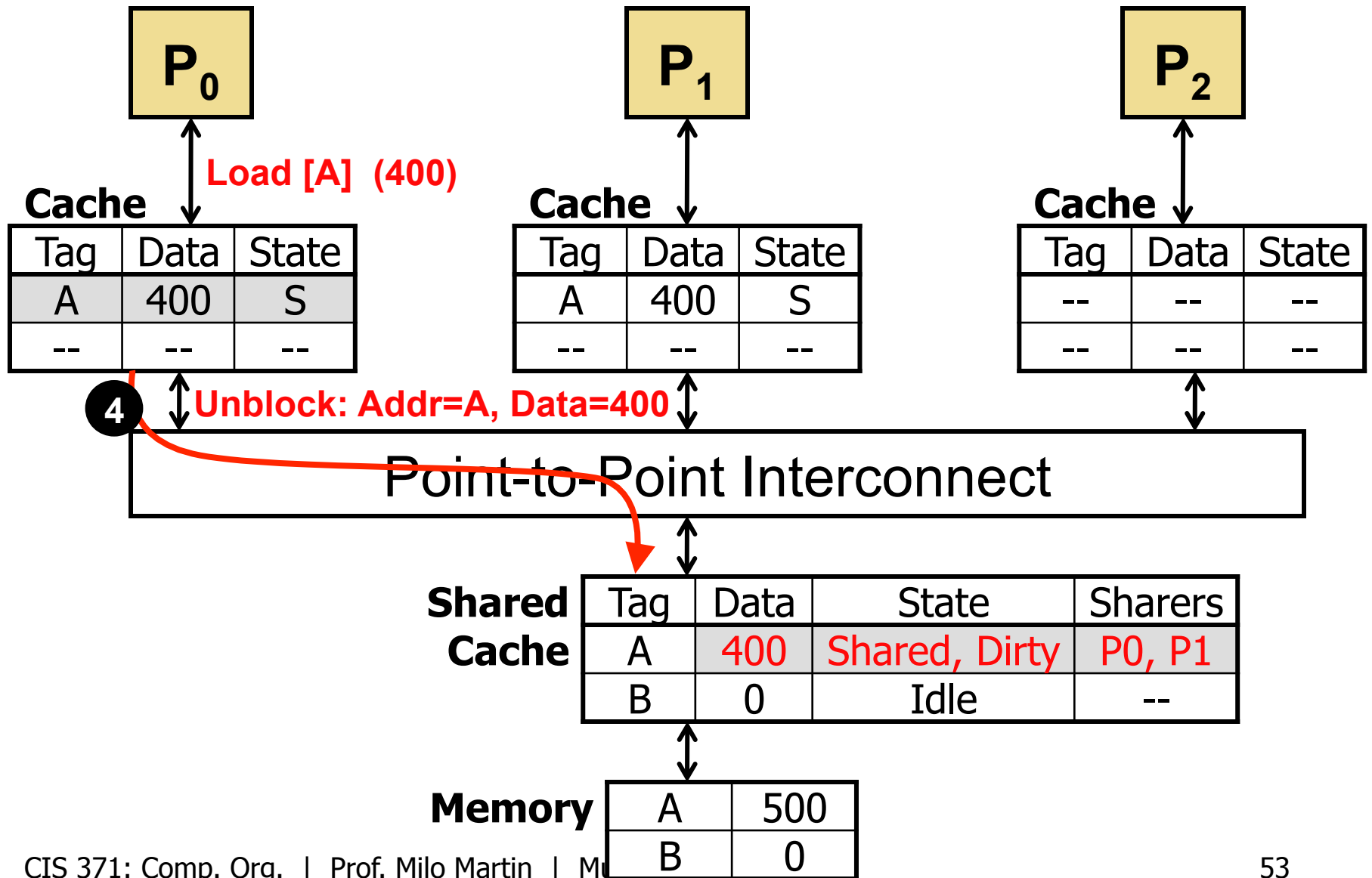
MSI Coherence Example: Step #3



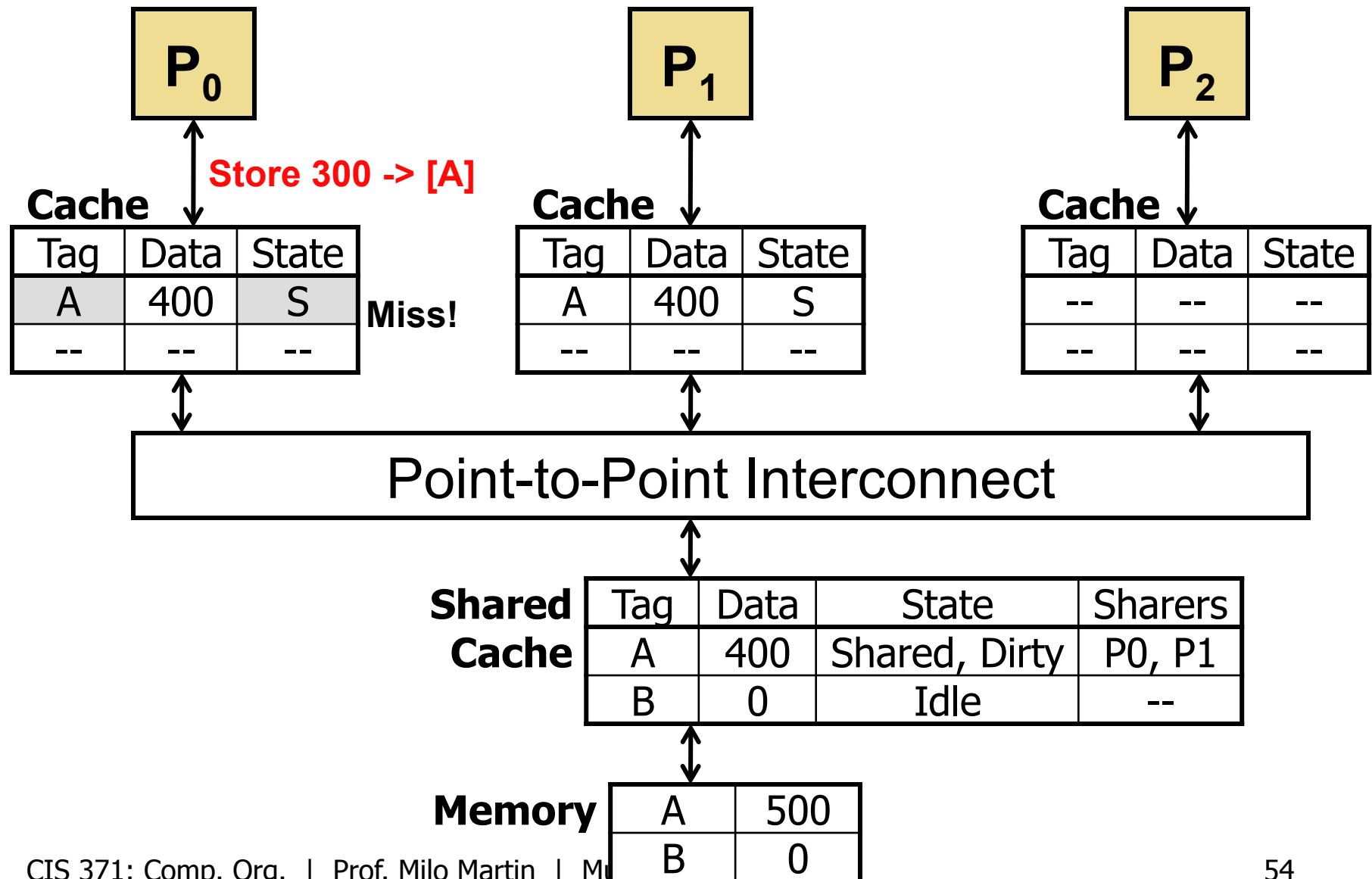
MSI Coherence Example: Step #4



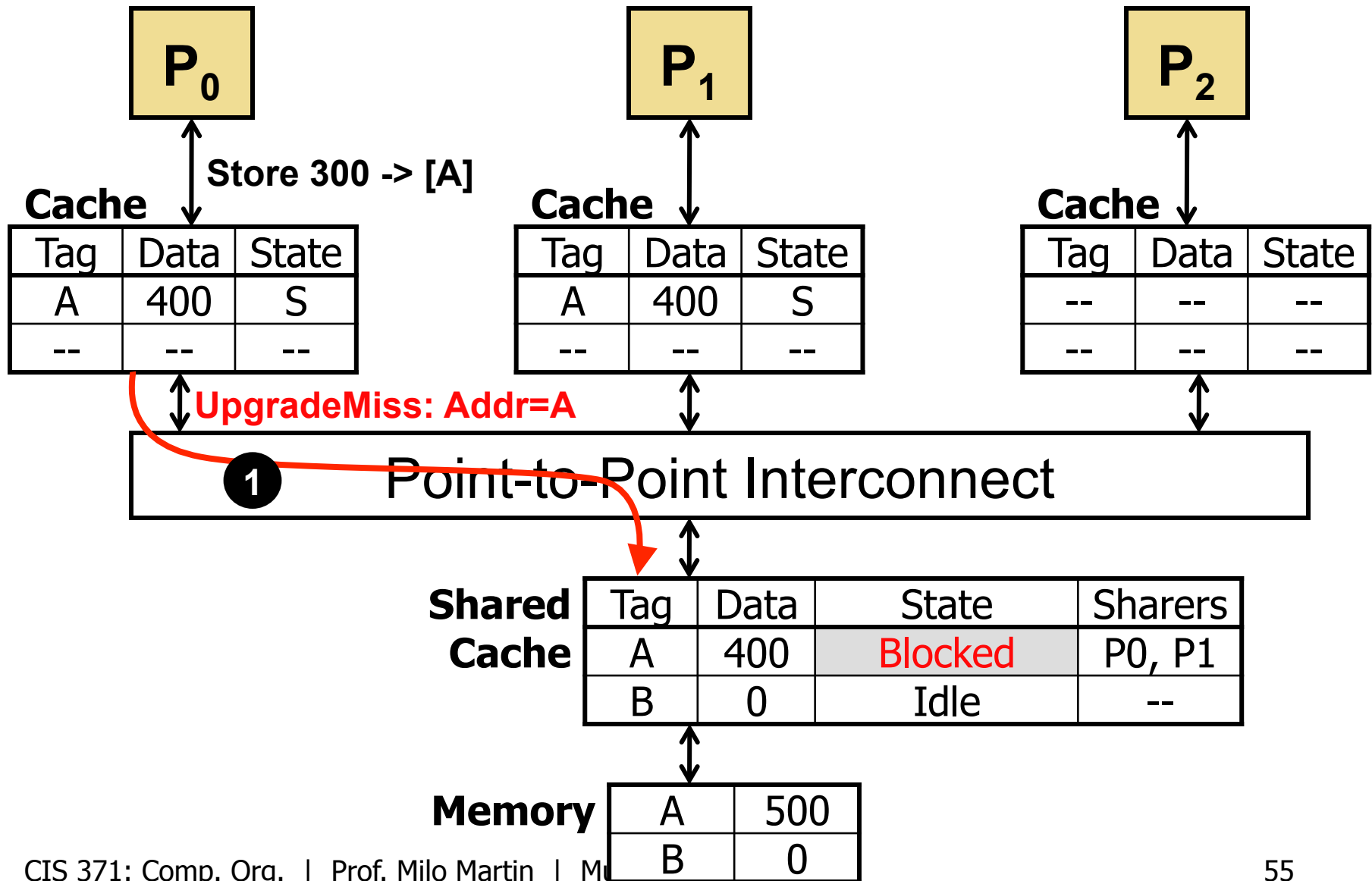
MSI Coherence Example: Step #5



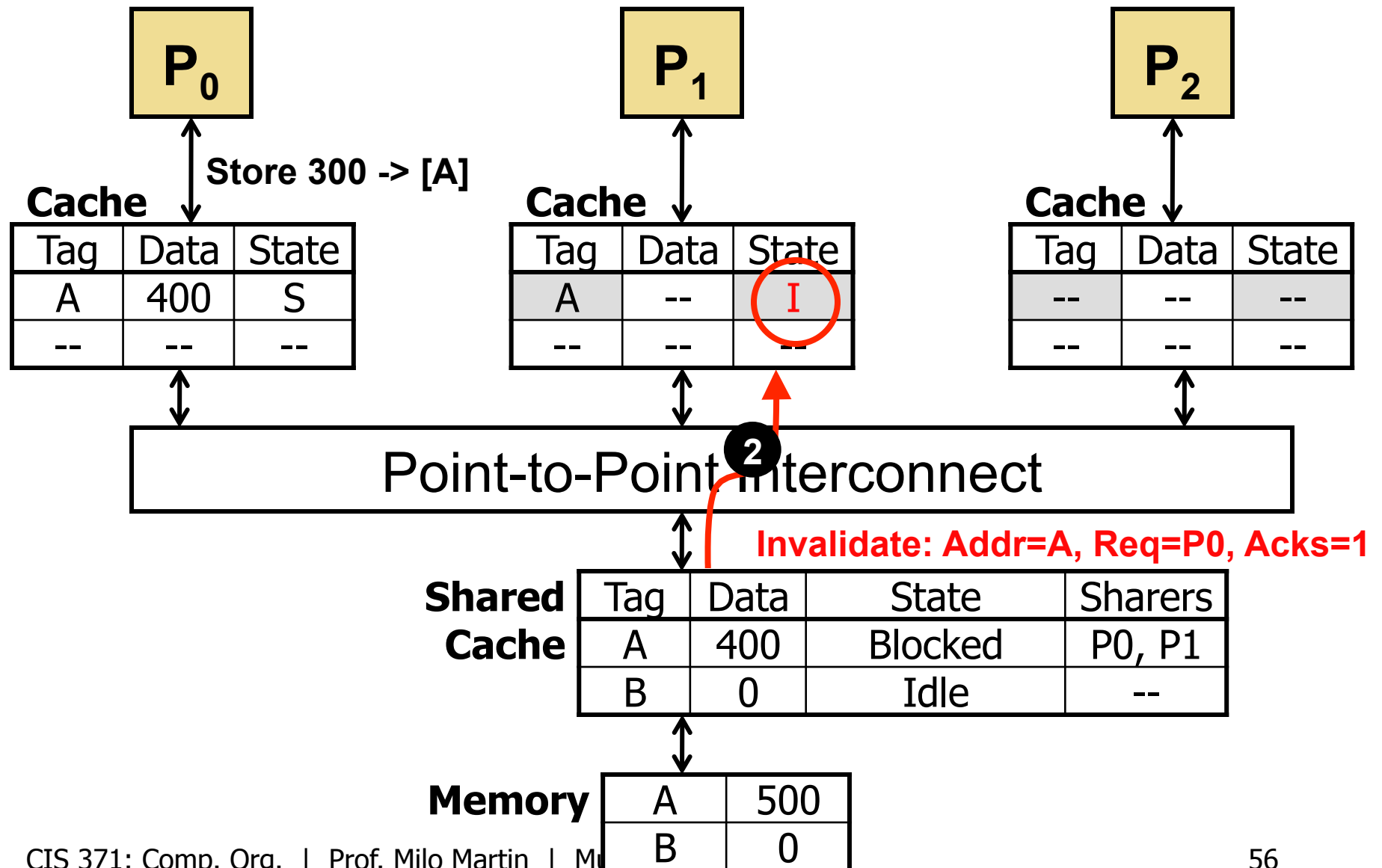
MSI Coherence Example: Step #6



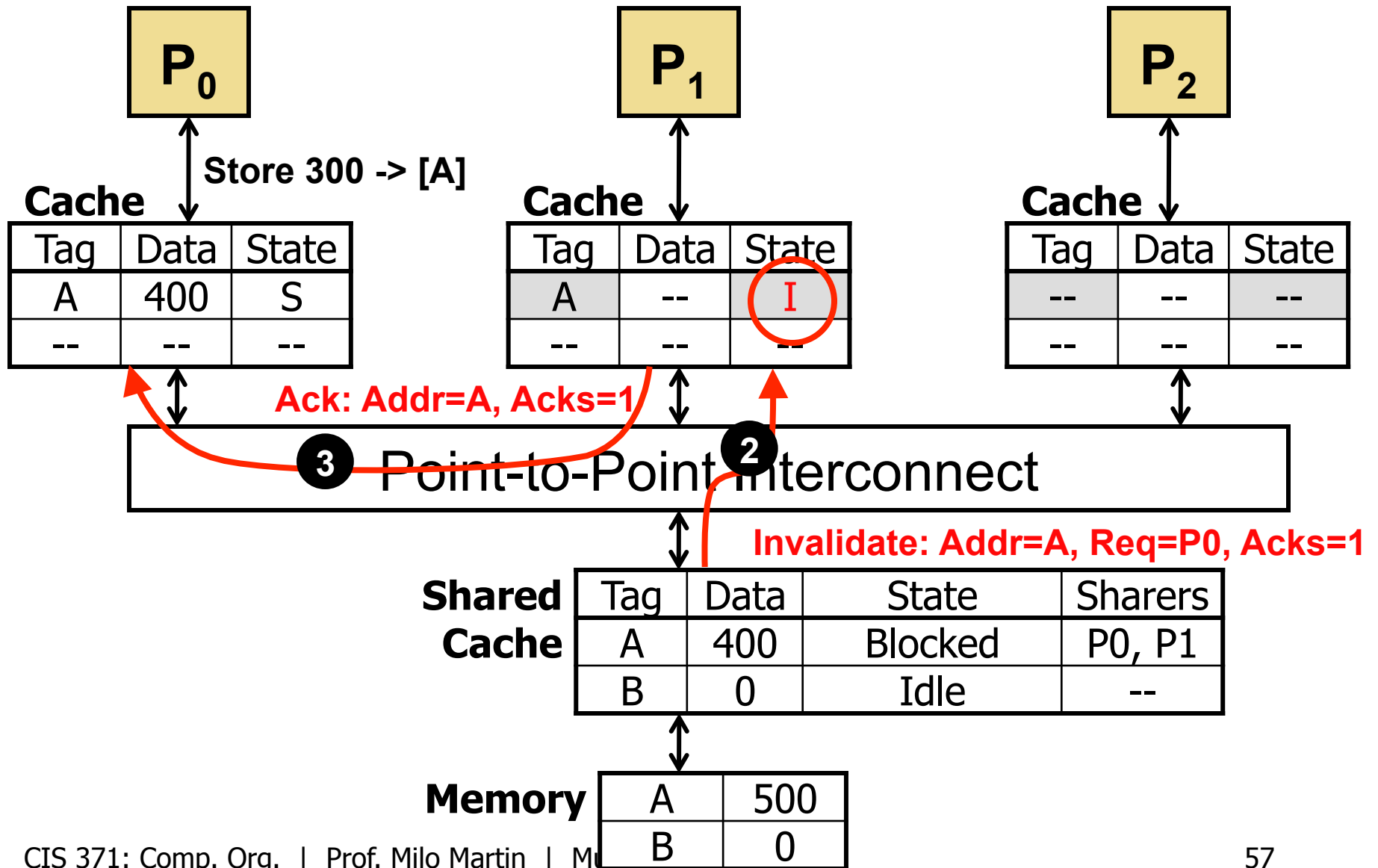
MSI Coherence Example: Step #7



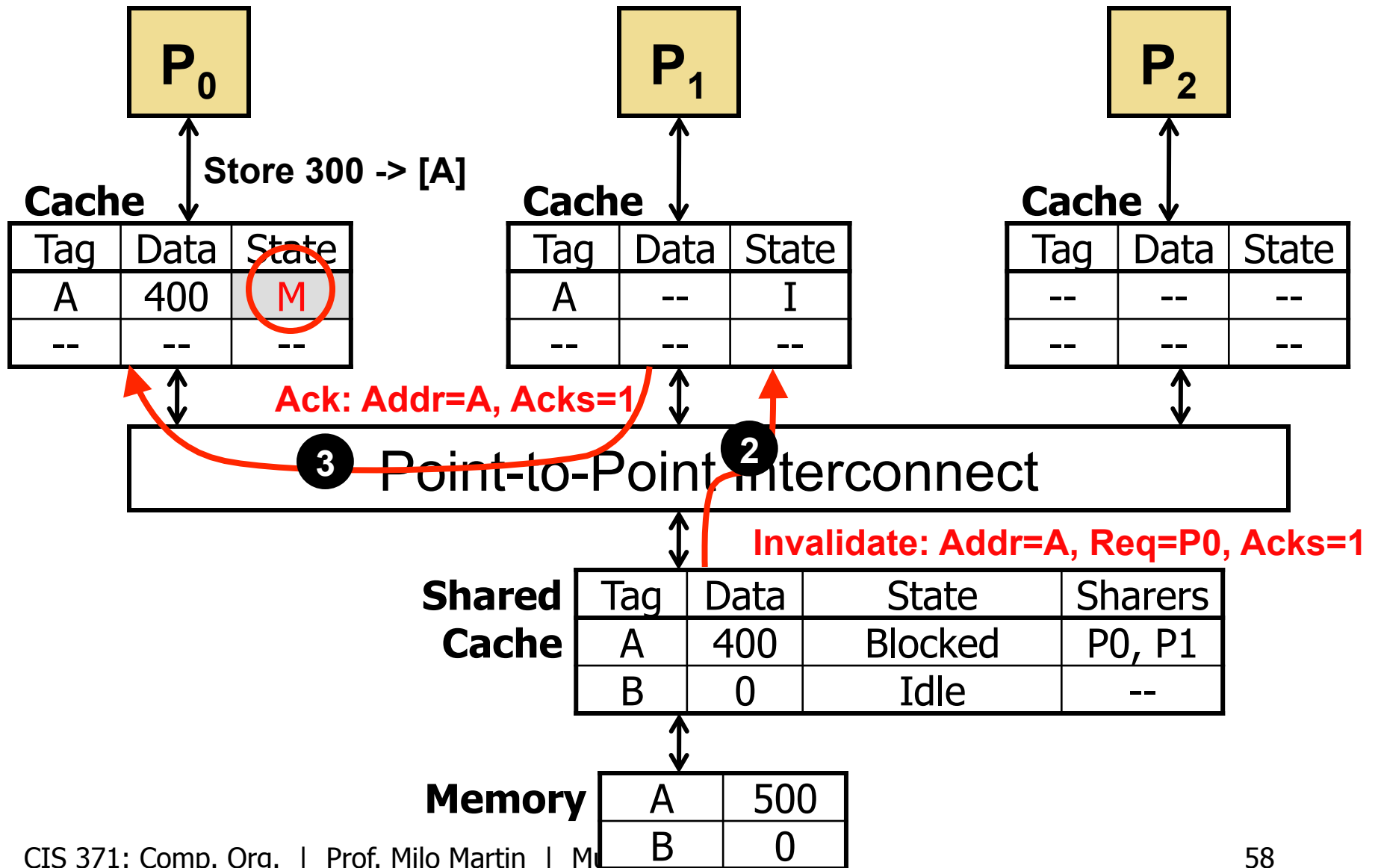
MSI Coherence Example: Step #8



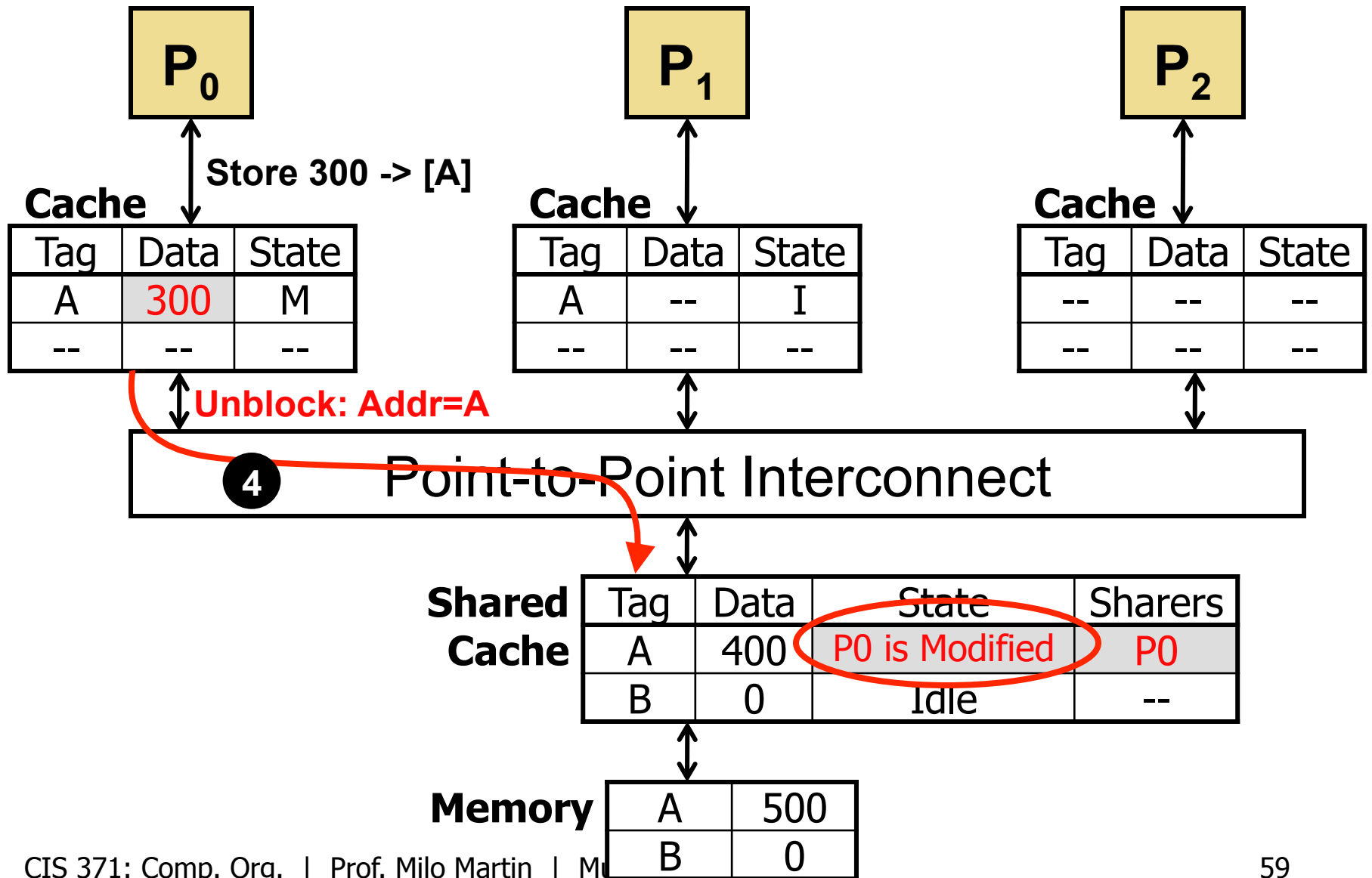
MSI Coherence Example: Step #9



MSI Coherence Example: Step #10



MSI Coherence Example: Step #11



MESI Cache Coherence

- Ok, we have read-only and read/write with MSI
- But consider load & then store of a block by same core
 - Under coherence as described, **this would be two misses: “Load miss” plus an “upgrade miss”...**
 - ... even if the block isn't shared!
 - Consider programs with 99% (or 100%) private data
 - Potentially doubling number of misses (bad)
- Solution:
 - Most modern protocols also include **E (exclusive)** state
 - Interpretation: “I have the only cached copy, and it's a **clean** copy”
 - Has read/write permissions
 - Just like “Modified” but “clean” instead of “dirty”.

MESI Operation

- Goals:
 - Avoid “upgrade” misses for non-shared blocks
 - While not increasing eviction (aka writeback or replacement) traffic
- Two cases on a load miss to a block...
 - **Case #1:** ... with no current sharers
(that is, no sharers in the set of sharers)
 - Grant requester “Exclusive” copy with read/write permission
 - **Case #2:** ... with other sharers
 - As before, grant just a “Shared” copy with read-only permission
- A store to a block in “Exclusive” changes it to “Modified”
 - Instantaneously & silently (no latency or traffic)
- On block eviction (aka writeback or replacement)...
 - If “Modified”, block is dirty, must be written back to next level
 - If “Exclusive”, writing back the data is not necessary
(but notification may or may not be, depending on the system)

Cache Coherence and Cache Misses

- With the “Exclusive” state...
 - **Coherence has no overhead on misses to non-shared blocks**
 - Just request/response like a normal cache miss
- But, coherence introduces two new kinds of cache misses
 - **Upgrade miss:** stores to read-only blocks
 - 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 types of misses
 - So, as cache grows large, these sorts of misses dominate
- **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

Cache Coherence Protocols

- Two general types
 - Update-based cache coherence
 - Write through update to all caches
 - Too much traffic; used in the past, not common today
 - **Invalidation-based cache coherence** (examples shown)
- Of invalidation-based cache coherence, two types:
 - Snooping/broadcast-based cache coherence
 - No explicit state, but too much traffic; not common today
 - **Directory-based cache coherence** (examples shown)
 - Track sharers of blocks
- For directory-based cache coherence, two options:
 - Enforce “inclusion”; if in per-core cache, must be in last-level cache
 - **Encoding sharers in cache tags** (examples shown & Core i7)
 - No inclusion? “directory cache” parallel to last-level cache (AMD)

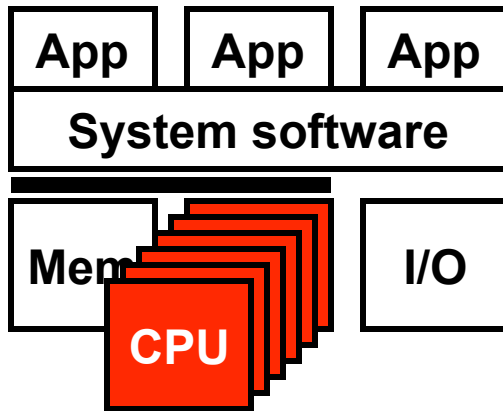
Scaling Cache Coherence

- **Scalable interconnect**
 - Build switched interconnect to communicate among cores
- **Scalable directory lookup bandwidth**
 - Address interleave (or “bank”) the last-level cache
 - Low-order bits of block address select which cache bank to access
 - Coherence controller per bank
- **Scalable traffic**
 - Amortized analysis shows traffic overhead independent of core #
 - Each invalidation can be tied back to some earlier request
- **Scalable storage**
 - Bit vector requires n-bits for n cores, scales up to maybe 32 cores
 - Inexact & “coarse” encodings trade more traffic for less storage
- Hierarchical design can help all of the above, too
- See: “Why On-Chip Cache Coherence is Here to Stay”, CACM, 2012

Coherence Recap & Alternatives

- Keeps caches “coherent”
 - Load returns the most recent stored value by any processor
 - And thus keeps caches transparent to software
- Alternatives to cache coherence
 - #1: no caching of shared data (slow)
 - #2: requiring software to explicitly “flush” data (hard to use)
 - Using some new instructions
 - #3: message passing (programming without shared memory)
 - Used in clusters of machines for high-performance computing
- However, directory-based coherence protocol scales well
 - Perhaps to 1000s of cores

Roadmap Checkpoint



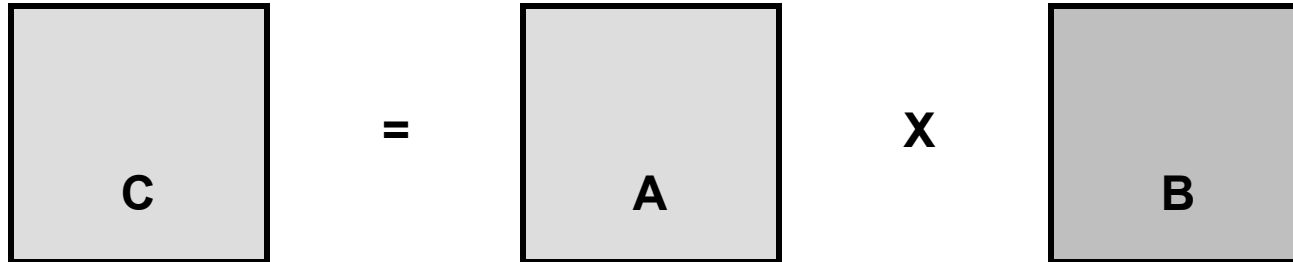
- ~~Thread-level parallelism (TLP)~~
- ~~Shared memory model~~
 - ~~Multiplexed uniprocessor~~
 - ~~Hardware multithreading~~
 - ~~Multiprocessing~~
- ~~Cache coherence~~
 - ~~Valid/Invalid, MSI, MESI~~
- **Parallel programming**
- **Synchronization**
 - **Lock implementation**
 - **Locking gotchas**
 - **Transactional memory**
- Memory consistency models

Parallel Programming

Parallel Programming

- One use of multiprocessors: **multiprogramming**
 - Running multiple programs with no interaction between them
 - Works great for a few cores, but what next?
- Or, programmers must **explicitly** express parallelism
 - “Coarse” parallelism beyond what the hardware can extract **implicitly**
 - Even the compiler can’t extract it in most cases
- How? Several options:
 1. Call libraries that perform well-known computations in parallel
 - Example: a matrix multiply routine, etc.
 2. Add code annotations (“this loop is parallel”), OpenMP
 3. Parallel “for” loops, task-based parallelism, ...
 4. Explicitly spawn “tasks”, runtime/OS schedules them on the cores
- Parallel programming: key challenge in multicore revolution

Example #1: Parallelizing Matrix Multiply



```
for (I = 0; I < SIZE; I++)  
  for (J = 0; J < SIZE; J++)  
    for (K = 0; K < SIZE; K++)  
      C[I][J] += A[I][K] * B[K][J];
```

- How to parallelize matrix multiply?
 - Replace outer "for" loop with "**parallel_for**" or OpenMP annotation
 - Supported by many parallel programming environments
- Implementation: give each of N processors loop iterations

```
int start = (SIZE/N) * my_id(); // my_id() from library  
for (I = start; I < start + SIZE/N; I++)  
  for (J = 0; J < SIZE; J++)  
    for (K = 0; K < SIZE; K++)  
      C[I][J] += A[I][K] * B[K][J];
```

- Each processor runs copy of loop above
 - No explicit synchronization required (implicit at end of loop)

Example #2: Bank Accounts

- Consider

```
struct acct_t { int balance; ... };
struct acct_t accounts[MAX_ACCT];      // current balances

struct trans_t { int id; int amount; };
struct trans_t transactions[MAX_TRANS]; // debit amounts

for (i = 0; i < MAX_TRANS; i++) {
    debit(transactions[i].id, transactions[i].amount);
}

void debit(int id, int amount) {
    if (accounts[id].balance >= amount) {
        accounts[id].balance -= amount;
    }
}
```

- Can we do these “debit” operations in parallel?
 - Does the order matter?

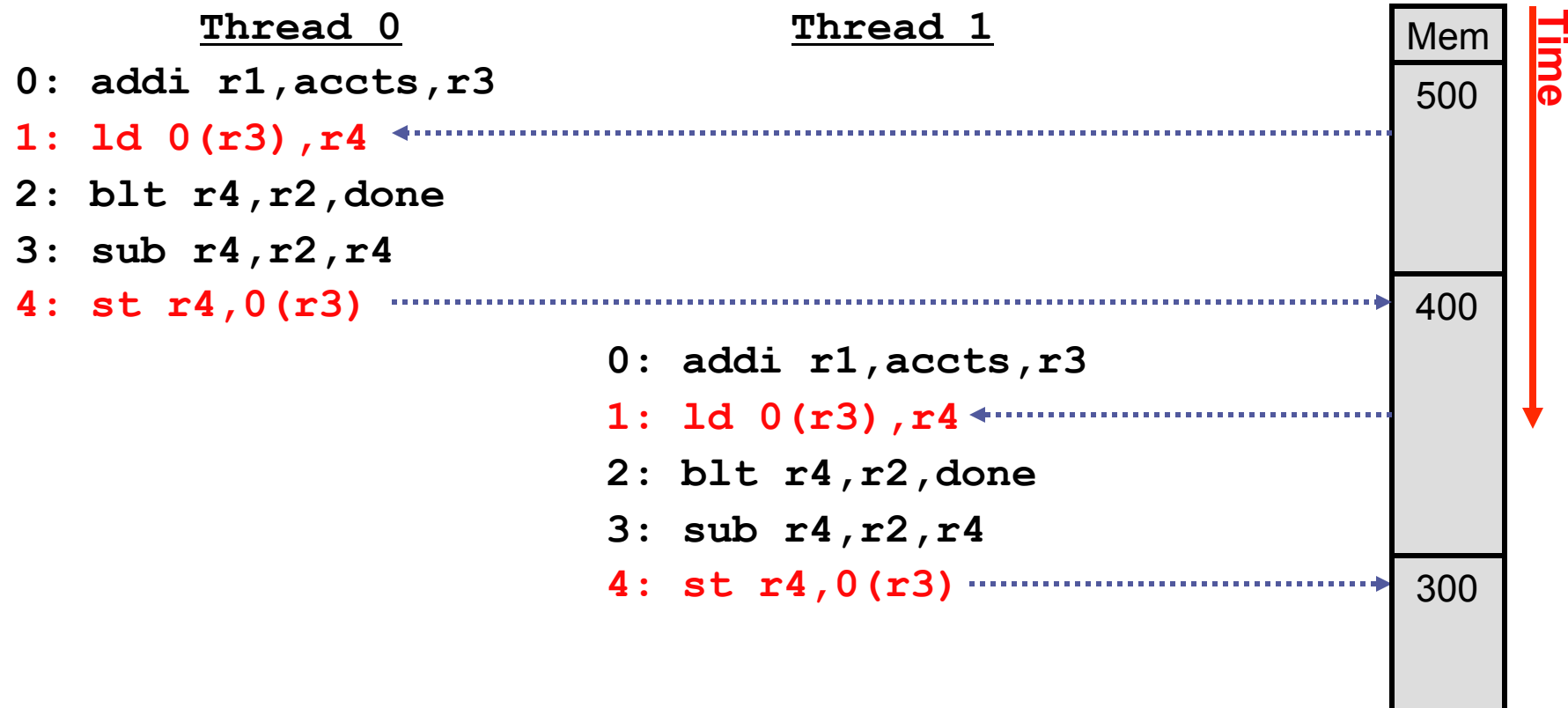
Example #2: Bank Accounts

```
struct acct_t { int bal; ... };
shared struct acct_t accts[MAX_ACCT];
void debit(int id, int amt) {
    if (accts[id].bal >= amt)
    {
        accts[id].bal -= amt;
    }
}
```

0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2,r4
4: st r4,0(r3)

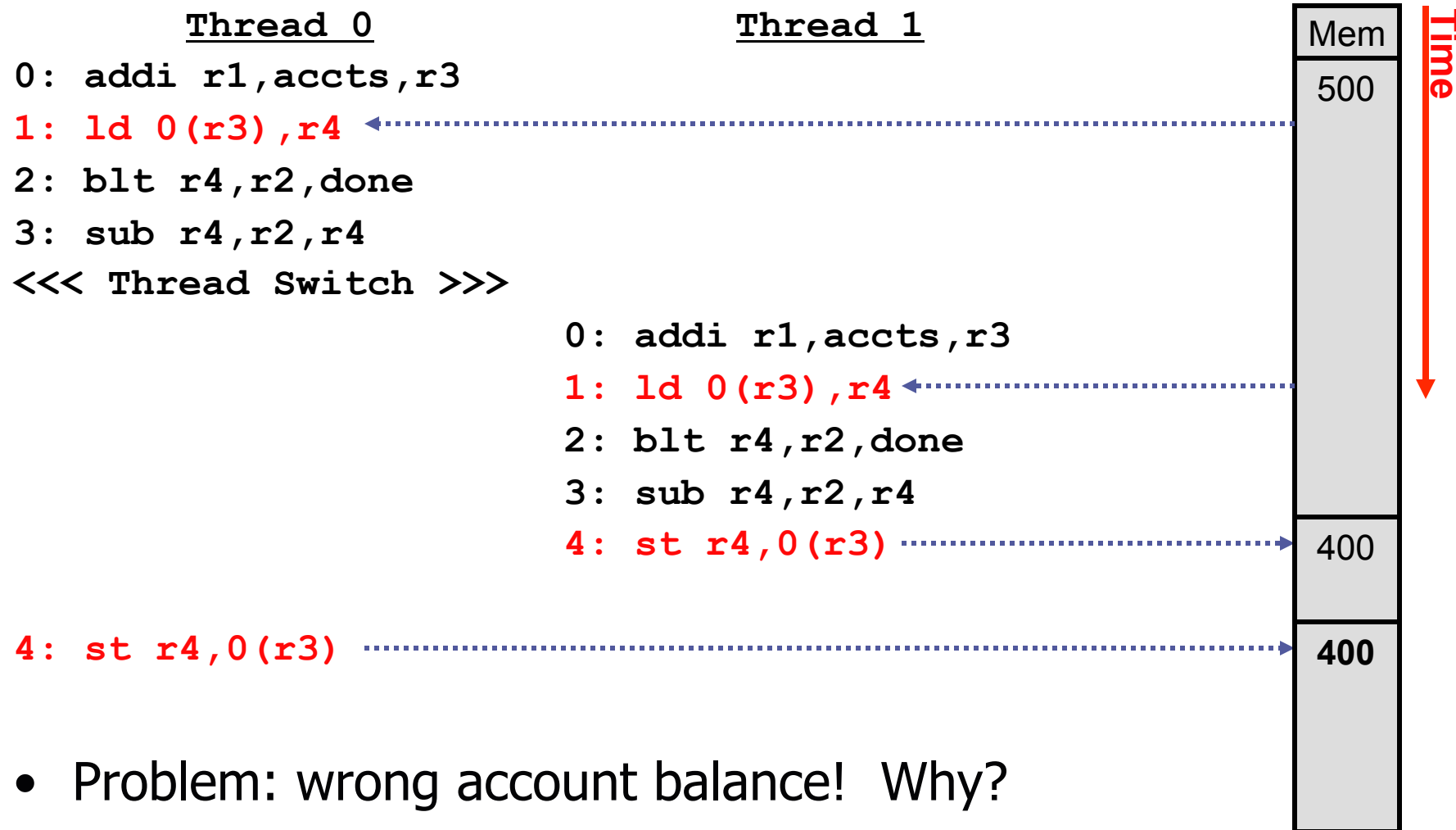
- Example of **Thread-level parallelism (TLP)**
 - Collection of asynchronous tasks: not started and stopped together
 - Data shared “loosely” (sometimes yes, mostly no), dynamically
- Example: database/web server (each query is a thread)
 - **accts** is global and thus **shared**, can't register allocate
 - **id** and **amt** are private variables, register allocated to **r1**, **r2**
- Running example

An Example Execution



- Two \$100 withdrawals from account #241 at two ATMs
 - Each transaction executed on different processor
 - Track `accts[241].bal` (address is in `r3`)

A Problem Execution



- Problem: wrong account balance! Why?
 - Solution: synchronize access to account balance

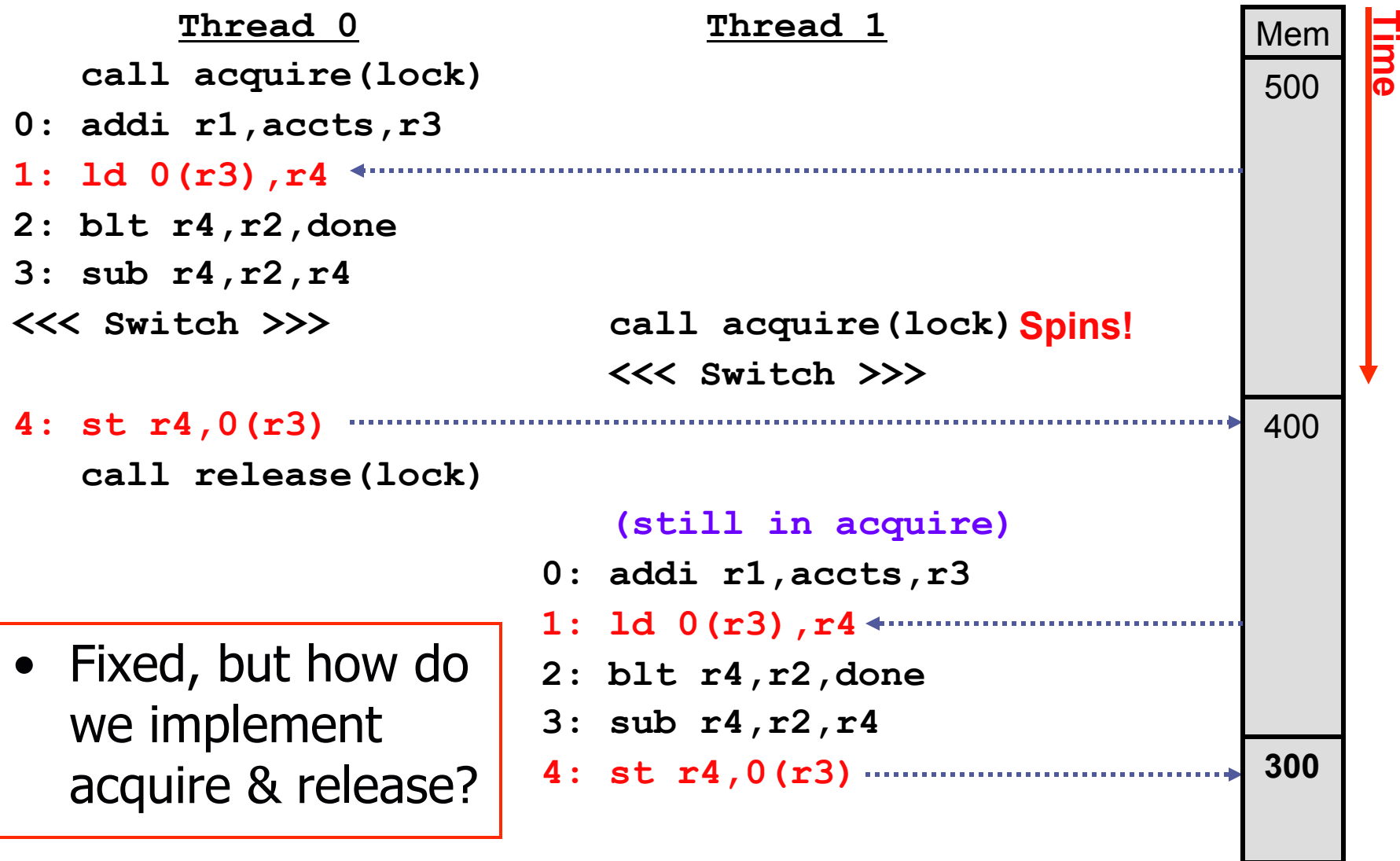
Synchronization

Synchronization:

- **Synchronization**: a key issue for shared memory
- Regulate access to shared data (mutual exclusion)
- Low-level primitive: **lock** (higher-level: "semaphore" or "mutex")
 - Operations: **acquire(lock)** and **release(lock)**
 - Region between **acquire** and **release** is a **critical section**
 - Must interleave **acquire** and **release**
 - Interfering **acquire** will block
- Another option: **Barrier synchronization**
 - Blocks until all threads reach barrier, used at end of "parallel_for"

```
struct acct_t { int bal; ... };  
shared struct acct_t accts[MAX_ACCT];  
shared int lock;  
void debit(int id, int amt):  
    acquire(lock);                                critical section  
    if (accts[id].bal >= amt) {  
        accts[id].bal -= amt;  
    }  
    release(lock);
```

A Synchronized Execution



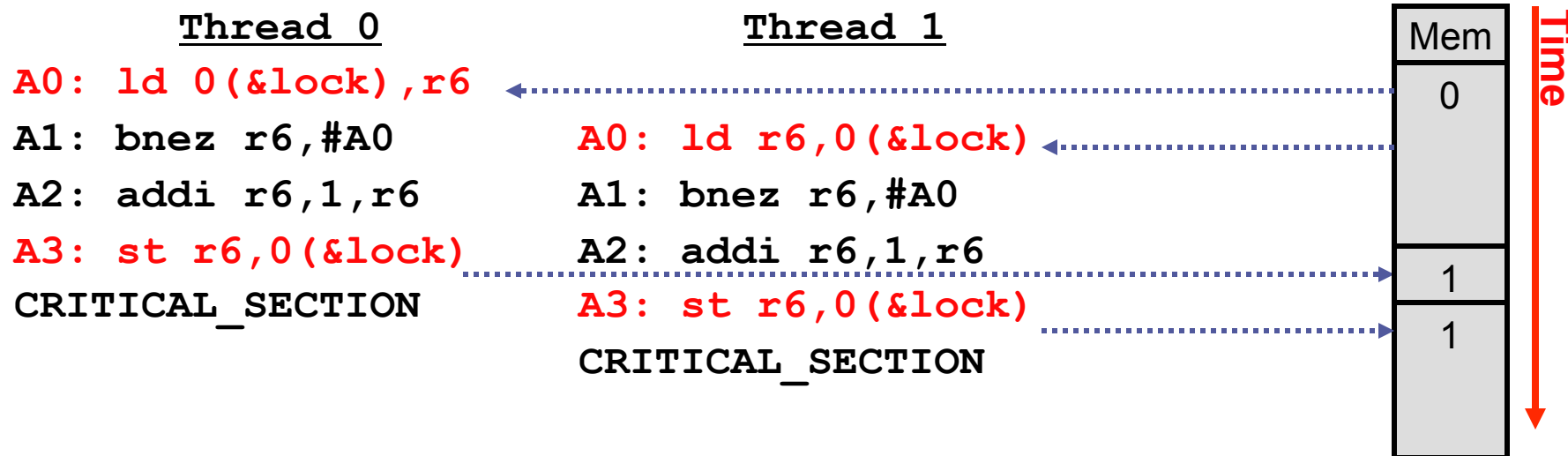
Strawman Lock (Incorrect)

- **Spin lock**: software lock implementation
 - `acquire(lock): while (lock != 0) {} lock = 1;`
 - "Spin" while lock is 1, wait for it to turn 0

```
A0: ld 0(&lock), r6
A1: bnez r6, A0
A2: addi r6, 1, r6
A3: st r6, 0(&lock)
```
 - `release(lock): lock = 0;`

```
R0: st r0, 0(&lock) // r0 holds 0
```

Incorrect Lock Implementation



- Spin lock makes intuitive sense, but doesn't actually work
 - Loads/stores of two **acquire** sequences can be interleaved
 - Lock **acquire** sequence also not atomic
 - **Same problem as before!**
- Note, **release** is trivially atomic

Correct Spin Lock: Use Atomic Swap

- ISA provides an atomic lock acquisition instruction

- Example: **atomic swap**

```
swap r1,0(&lock)
```

- Atomically executes:

```
mov r1->r2
ld r1,0(&lock)
st r2,0(&lock)
```

- New acquire sequence

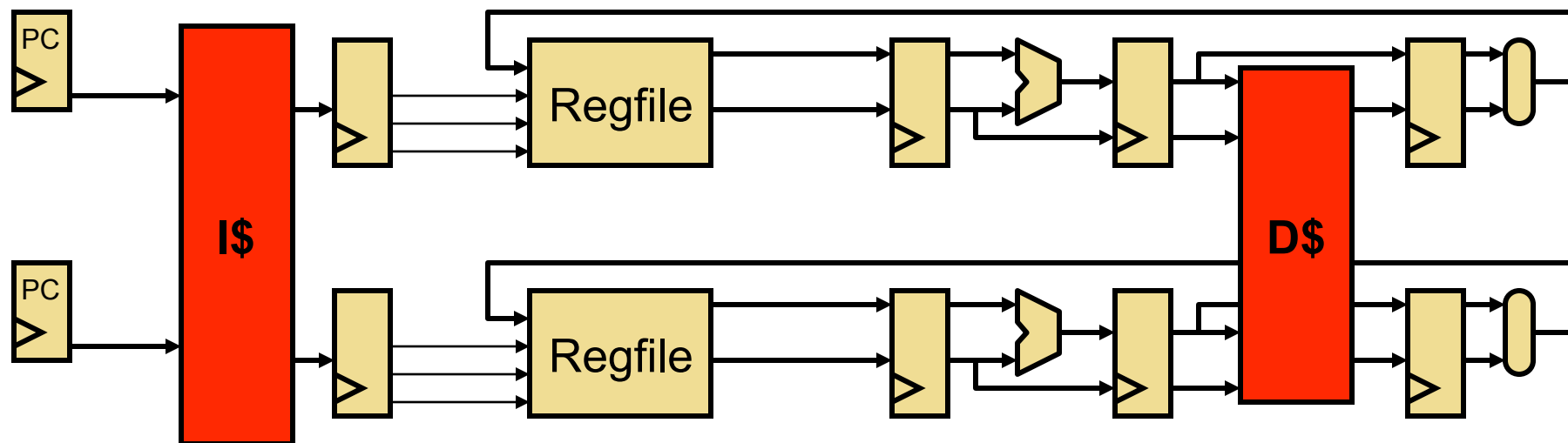
```
(value of r1 is 1)
```

```
A0: swap r1,0(&lock)
```

```
A1: bnez r1,A0
```

- If lock was initially busy (1), doesn't change it, **keep looping**
 - If lock was initially free (0), acquires it (sets it to 1), break loop
- Insures lock held by **at most one thread**
 - Other variants: **exchange**, **compare-and-swap**, **test-and-set (t&s)**, or **fetch-and-add**

Atomic Update/Swap Implementation



- How is atomic swap implemented?
 - Need to ensure no intervening memory operations
 - Requires blocking access by other threads temporarily (yuck)
- How to pipeline it?
 - Both a load and a store (yuck)
 - Not very RISC-like

RISC Test-And-Set

- **swap**: a load and store in one insn is not very "RISC"
 - Broken up into micro-ops, but then how is it made atomic?
- "Load-link" / "store-conditional" pairs
 - Atomic load/store pair
 - `label:`
 - `load-link r1,0(&lock)`
 - `// potentially other insns`
 - `store-conditional r2,0(&lock)`
 - `branch-not-zero label // check for failure`
 - On `load-link`, processor remembers address...
 - ...And looks for writes by other processors
 - If write is detected, next `store-conditional` will fail
 - Sets failure condition
- Used by ARM, PowerPC, MIPS, Itanium

Lock Correctness

Thread 0

A0: swap r1,0(&lock)

A1: bnez r1,#A0

CRITICAL_SECTION

Thread 1

A0: swap r1,0(&lock)

A1: bnez r1,#A0

A0: swap r1,0(&lock)

A1: bnez r1,#A0

+ Lock actually works...

- Thread 1 keeps spinning
- Sometimes called a “test-and-set lock”
 - Named after the common “test-and-set” atomic instruction

“Test-and-Set” Lock Performance

Thread 0

A0: **swap** r1,0(&lock)

A1: bnez r1,#A0

A0: **swap** r1,0(&lock)

A1: bnez r1,#A0

Thread 1

A0: **swap** r1,0(&lock)

A1: bnez r1,#A0

A0: **swap** r1,0(&lock)

A1: bnez r1,#A0

– ...but performs poorly

- Consider 3 processors rather than 2
- Processor 2 (not shown) has the lock and is in the critical section
- But what are processors 0 and 1 doing in the meantime?
 - Loops of **swap**, each of which includes a **st**
 - Repeated stores by multiple processors costly
 - Generating a ton of useless interconnect traffic

Test-and-Test-and-Set Locks

- Solution: **test-and-test-and-set locks**
 - New acquire sequence
 - A0: ld r1,0(&lock)**
 - A1: bnez r1,A0**
 - A2: addi r1,1,r1**
 - A3: swap r1,0(&lock)**
 - A4: bnez r1,A0**
 - Within each loop iteration, before doing a **swap**
 - Spin doing a simple test (**ld**) to see if lock value has changed
 - Only do a **swap (st)** if lock is actually free
 - Processors can spin on a busy lock locally (in their own cache)
 - + Less unnecessary interconnect traffic
 - Note: test-and-test-and-set is not a new instruction!
 - Just different software

Queue Locks

- Test-and-test-and-set locks can still perform poorly
 - If lock is contended for by many processors
 - Lock release by one processor, creates “free-for-all” by others
 - Interconnect gets swamped with **swap** requests
- **Software queue lock**
 - Each waiting processor spins on a different location (a queue)
 - When lock is released by one processor...
 - Only the next processors sees its location go “unlocked”
 - Others continue spinning locally, unaware lock was released
 - Effectively, passes lock from one processor to the next, in order
 - + Greatly reduced network traffic (no mad rush for the lock)
 - + Fairness (lock acquired in FIFO order)
 - Higher overhead in case of no contention (more instructions)
 - Poor performance if one thread is descheduled by O.S.

Programming With Locks Is Tricky

- Multicore processors are the way of the foreseeable future
 - thread-level parallelism anointed as parallelism model of choice
 - Just one problem...
- Writing lock-based multi-threaded programs is tricky!
- More precisely:
 - Writing programs that are correct is “easy” (not really)
 - Writing programs that are highly parallel is “easy” (not really)
 - **Writing programs that are both correct and parallel is difficult**
 - And that’s the whole point, unfortunately
 - Selecting the “right” kind of lock for performance
 - Spin lock, queue lock, ticket lock, read/writer lock, etc.
 - **Locking granularity issues**

Coarse-Grain Locks: Correct but Slow

- **Coarse-grain locks:** e.g., one lock for entire database
 - + Easy to make correct: no chance for unintended interference
 - Limits parallelism: no two critical sections can proceed in parallel

```
struct acct_t { int bal; ... };
shared struct acct_t accts[MAX_ACCT];
shared Lock_t lock;
void debit(int id, int amt) {
    acquire(lock);
    if (accts[id].bal >= amt) {
        accts[id].bal -= amt;
    }
    release(lock);
}
```

Fine-Grain Locks: Parallel But Difficult

- **Fine-grain locks:** e.g., multiple locks, one per record
 - + Fast: critical sections (to different records) can proceed in parallel
 - Difficult to make correct: easy to make mistakes
 - This particular example is easy
 - Requires only one lock per critical section

```
struct acct_t { int bal, Lock_t lock; ... };  
shared struct acct_t accts[MAX_ACCT];
```

```
void debit(int id, int amt) {  
    acquire(accts[id].lock);  
    if (accts[id].bal >= amt) {  
        accts[id].bal -= amt;  
    }  
    release(accts[id].lock);  
}
```

- What about critical sections that require two locks?

Multiple Locks

- **Multiple locks:** e.g., acct-to-acct transfer
 - Must acquire both `id_from`, `id_to` locks
 - Running example with accts 241 and 37
 - Simultaneous transfers `241 → 37` and `37 → 241`
 - Contrived... but even contrived examples must work correctly too

```
struct acct_t { int bal, Lock_t lock; ...};
shared struct acct_t accts[MAX_ACCT];
void transfer(int id_from, int id_to, int amt) {
    acquire(accts[id_from].lock);
    acquire(accts[id_to].lock);
    if (accts[id_from].bal >= amt) {
        accts[id_from].bal -= amt;
        accts[id_to].bal += amt;
    }
    release(accts[id_to].lock);
    release(accts[id_from].lock);
}
```

Multiple Locks And Deadlock

Thread 0

```
id_from = 241;  
id_to = 37;
```

```
acquire(accts[241].lock);  
// wait to acquire lock 37  
// waiting...  
// still waiting...
```

Thread 1

```
id_from = 37;  
id_to = 241;
```

```
acquire(accts[37].lock);  
// wait to acquire lock 241  
// waiting...  
// ...
```



FAIL Blog

Multiple Locks And Deadlock

Thread 0

```
id_from = 241;  
id_to = 37;
```

```
acquire(accts[241].lock);  
// wait to acquire lock 37  
// waiting...  
// still waiting...
```

Thread 1

```
id_from = 37;  
id_to = 241;
```

```
acquire(accts[37].lock);  
// wait to acquire lock 241  
// waiting...  
// ...
```

- **Deadlock:** circular wait for shared resources
 - Thread 0 has lock 241 waits for lock 37
 - Thread 1 has lock 37 waits for lock 241
 - Obviously this is a problem
 - The solution is ...

Correct Multiple Lock Program

- **Always acquire multiple locks in same order**
 - Just another thing to keep in mind when programming

```
struct acct_t { int bal, Lock_t lock; ... };
shared struct acct_t accts[MAX_ACCT];
void transfer(int id_from, int id_to, int amt) {
    int id_first = min(id_from, id_to);
    int id_second = max(id_from, id_to);

    acquire(accts[id_first].lock);
    acquire(accts[id_second].lock);
    if (accts[id_from].bal >= amt) {
        accts[id_from].bal -= amt;
        accts[id_to].bal += amt;
    }
    release(accts[id_second].lock);
    release(accts[id_first].lock);
}
```

Correct Multiple Lock Execution

Thread 0

```
id_from = 241;  
id_to = 37;  
id_first = min(241,37)=37;  
id_second = max(37,241)=241;
```

```
acquire(accts[37].lock);  
acquire(accts[241].lock);  
// do stuff  
release(accts[241].lock);  
release(accts[37].lock);
```

Thread 1

```
id_from = 37;  
id_to = 241;  
id_first = min(37,241)=37;  
id_second = max(37,241)=241;
```

```
// wait to acquire lock 37  
// waiting...  
// ...  
// ...  
// ...  
acquire(accts[37].lock);
```

- Great, are we done? No

More Lock Madness

- What if...
 - Some actions (e.g., deposits, transfers) require 1 or 2 locks...
 - ...and others (e.g., prepare statements) require all of them?
 - Can these proceed in parallel?
- What if...
 - There are locks for global variables (e.g., operation id counter)?
 - When should operations grab this lock?
- What if... what if... what if...
- **So lock-based programming is difficult...**
- **...wait, it gets worse**

And To Make It Worse...

- **Acquiring locks is expensive...**
 - By definition requires a slow atomic instructions
 - Specifically, acquiring write permissions to the lock
 - Ordering constraints (see soon) make it even slower
- **...and 99% of the time un-necessary**
 - Most concurrent actions don't actually share data
 - You paying to acquire the lock(s) for no reason
- Fixing these problem is an area of active research
 - One proposed solution "Transactional Memory"
 - Programmer uses construct: "atomic { ... code ... }"
 - Hardware, compiler & runtime executes the code "atomically"
 - Uses speculation, rolls back on conflicting accesses

Research: Transactional Memory (TM)

- **Transactional Memory (TM) goals:**
 - + Programming simplicity of coarse-grain locks
 - + Higher concurrency (parallelism) of fine-grain locks
 - Critical sections only serialized if data is actually shared
 - + Lower overhead than lock acquisition
- Hot academic & industrial research topic (or was a few years ago)
- No fewer than nine research projects:
 - Brown, Stanford, MIT, Wisconsin, Texas, Rochester, Sun/Oracle, Intel
 - Penn, too
- Most recently:
 - Intel announced TM support in “Haswell” core! (shipping in 2013)

Transactional Memory: The Big Idea

- Big idea I: **no locks, just shared data**
- Big idea II: **optimistic (speculative) concurrency**
 - Execute critical section speculatively, abort on conflicts
 - “Better to beg for forgiveness than to ask for permission”

```
struct acct_t { int bal; ... };
shared struct acct_t accts[MAX_ACCT];
void transfer(int id_from, int id_to, int amt) {
    begin_transaction();
    if (accts[id_from].bal >= amt) {
        accts[id_from].bal -= amt;
        accts[id_to].bal += amt;
    }
    end_transaction();
}
```

Transactional Memory: Read/Write Sets

- **Read set:** set of shared addresses critical section reads
 - Example: `accts[37].bal`, `accts[241].bal`
- **Write set:** set of shared addresses critical section writes
 - Example: `accts[37].bal`, `accts[241].bal`

```
struct acct_t { int bal; ... };
shared struct acct_t accts[MAX_ACCT];
void transfer(int id_from, int id_to, int amt) {
    begin_transaction();
    if (accts[id_from].bal >= amt) {
        accts[id_from].bal -= amt;
        accts[id_to].bal += amt;
    }
    end_transaction();
}
```

Transactional Memory: Begin

- **begin_transaction**
 - Take a local register checkpoint
 - Begin locally tracking read set (remember addresses you read)
 - Used to detect if anyone other thread (core) tries to write it
 - Locally buffer all of your writes (not yet visible to other processors)
- + **Local actions only: no lock acquire**

```
struct acct_t { int bal; ... };
shared struct acct_t accts[MAX_ACCT];
void transfer(int id_from, int id_to, int amt) {
    begin_transaction();
    if (accts[id_from].bal >= amt) {
        accts[id_from].bal -= amt;
        accts[id_to].bal += amt;
    }
    end_transaction();
}
```

Transactional Memory: End

- **end_transaction**
 - Were any conflict detected? A conflict is:
 - Accesses from two **different** threads...
to the **same** address...
and at least one of them is a **write**
 - Yes? Commit transactions: commit writes
 - No? Abort transaction: restore checkpoint, try again

```
struct acct_t { int bal; ... };
shared struct acct_t accts[MAX_ACCT];
void transfer(int id_from, int id_to, int amt) {
    begin_transaction();
    if (accts[id_from].bal >= amt) {
        accts[id_from].bal -= amt;
        accts[id_to].bal += amt;
    }
    end_transaction();
}
```

Transactional Memory Implementation

- How is read-set/write-set tracking implemented?
 - Track locations accessed **using the cache**
 - Leverage cache coherence protocol for conflict detection
- Read-set: additional “transactional read” bit per block
 - Set it on loads between `begin_transaction` and `end_transaction`
 - If other thread writes a block with bit set → trigger abort
 - “Flash clear” all read bits on transaction abort or commit
- Write-set: additional “transactional write” bit per block
 - Set on writes between `begin_transaction` and `end_transaction`
 - Before first write, if dirty, initiate writeback (“clean” the block)
 - If other thread reads or writes a block with bit set → trigger abort
 - “Flash clear” all write bits to commit transaction
 - To abort transaction: invalidate all blocks with bit set

Transactional Execution

Thread 0

```
id_from = 241;
id_to = 37;

begin_transaction();
if(accts[241].bal > 100) {
    ...
    // write accts[241].bal
    // abort
```

Thread 1

```
id_from = 37;
id_to = 241;

begin_transaction();
if(accts[37].bal > 100) {
    accts[37].bal -= amt;
    accts[241].bal += amt;
}
end_transaction();
// no writes to accts[241].bal
// no writes to accts[37].bal
// commit
```

Transactional Execution II (More Likely)

Thread 0

```
id_from = 241;
id_to = 37;

begin_transaction();
if(accts[241].bal > 100) {
    accts[241].bal -= amt;
    acts[37].bal += amt;
}
end_transaction();
// no write to accts[240].bal
// no write to accts[37].bal
// commit
```

Thread 1

```
id_from = 450;
id_to = 118;

begin_transaction();
if(accts[450].bal > 100) {
    accts[450].bal -= amt;
    acts[118].bal += amt;
}
end_transaction();
// no write to accts[450].bal
// no write to accts[118].bal
// commit
```

- Critical sections execute in parallel

So, Let's Just Do Transactions?

- What if...
 - Read-set or write-set bigger than cache?
 - Transaction gets swapped out in the middle?
 - Transaction wants to do I/O or SYSCALL (not-abortable)?
- How do we transactify existing lock based programs?
 - Replace `acquire` with `begin_trans` does not always work
- Several different kinds of transaction semantics
 - Are transactions atomic relative to code outside of transactions?
- Do we want transactions in hardware or in software?
 - What we just saw is **hardware transactional memory (HTM)**
- That's what these research groups are looking at
 - Best-effort hardware TM: Azul systems, Sun's Rock processor

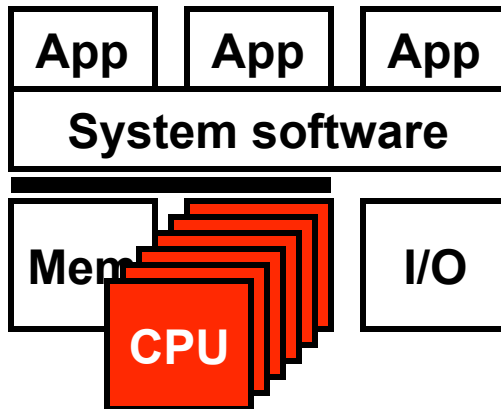
Speculative Lock Elision (SLE)

Processor 0

```
acquire(accts[37].lock); // don't actually set lock to 1
// begin tracking read/write sets
// CRITICAL_SECTION
// check read set
// no conflicts? Commit, don't actually set lock to 0
// conflicts? Abort, retry by acquiring lock
release(accts[37].lock);
```

- Alternatively, keep the locks, but...
- ... speculatively transactify lock-based programs in hardware
 - **Speculative Lock Elision (SLE)** [Rajwar+, MICRO'01]
 - Captures most of the advantages of transactional memory...
 - + No need to rewrite programs
 - + Can always fall back on lock-based execution (overflow, I/O, etc.)
- Intel's "Haswell" supports both SLE & best-effort TM

Roadmap Checkpoint



- ~~Thread-level parallelism (TLP)~~
- ~~Shared memory model~~
 - ~~Multiplexed uniprocessor~~
 - ~~Hardware multithreading~~
 - ~~Multiprocessing~~
- ~~Cache coherence~~
 - ~~Valid/Invalid, MSI, MESI~~
- ~~Parallel programming~~
- ~~Synchronization~~
 - ~~Lock implementation~~
 - ~~Locking gotchas~~
 - ~~Transactional memory~~
- **Memory consistency models**

Shared Memory Example #1

- **Initially: all variables zero** (that is, x is 0, y is 0)

thread 1	thread 2
store 1 → y load x	store 1 → x load y

- What value pairs can be read by the two loads?

Shared Memory Example #1: "Answer"

- Initially: all variables zero (that is, x is 0, y is 0)

thread 1	thread 2
store 1 → y load x	store 1 → x load y

- What value pairs can be read by the two loads?

store 1 → y load x store 1 → x load y (x=0, y=1)	store 1 → y store 1 → x load x load y (x=1, y=1)	store 1 → y store 1 → x load y load x (x=1, y=1)
store 1 → x load y store 1 → y load x (x=1, y=0)	store 1 → x store 1 → y load y load x (x=1, y=1)	store 1 → x store 1 → y load x load y (x=1, y=1)

- What about (x=0, y=0)? Nope... or can it?

Shared Memory Example #2

- **Initially: all variables zero** ("flag" is 0, "a" is 0)

thread 1	thread 2
<code>store 1 → a</code> <code>store 1 → flag</code>	<code>loop: if (flag == 0) goto loop</code> <code>load a</code>

- What value can be read by "load a"?

Shared Memory Example #2: "Answer"

- **Initially: all variables zero** ("flag" is 0, "a" is 0)

thread 1	thread 2
<code>store 1 → a</code> <code>store 1 → flag</code>	<code>loop: if (flag == 0) goto loop</code> <code>load a</code>

- What value can be read by "load a"?
 - "load a" can see the value "1"
- Can "load a" read the value zero?
 - Are you sure?

What is Going On?

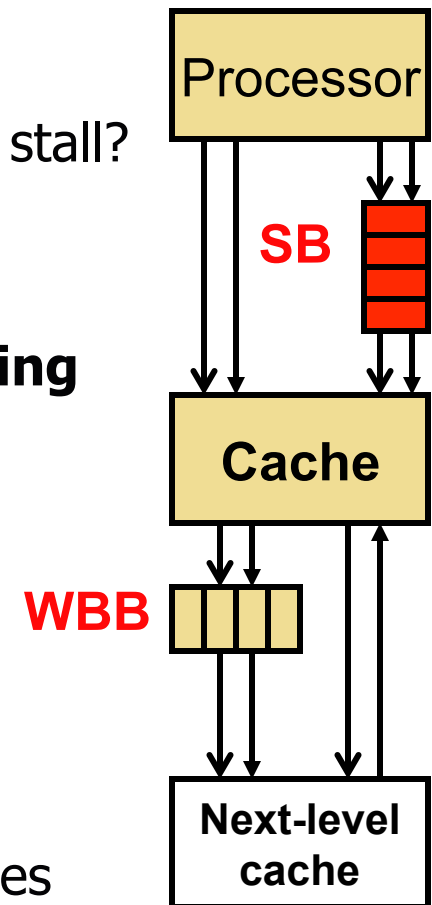
- Reordering of memory operations to different addresses!
 - **Unlike** the *sequential consistency* model we've been assuming
- **In the compiler**
 - Compiler is generally allowed to re-order memory operations to different addresses
 - Many other compiler optimizations also cause problems
- **In the hardware**
 1. To tolerate write latency
 - Cores don't wait for writes to complete (via store buffers)
 - And why should they? No reason to wait on non-threaded code
 2. To simplify out-of-order execution

Memory Consistency

- **Memory coherence**
 - Creates globally uniform (consistent) view...
 - Of **a single memory location** (in other words: cache blocks)
 - Not enough
 - Cache blocks A and B can be individually consistent...
 - But inconsistent with respect to each other
- **Memory consistency**
 - Creates globally uniform (consistent) view...
 - Of **all memory locations relative to each other**
- Who cares? Programmers
 - Globally inconsistent memory creates mystifying behavior

Recall: Write Misses and Store Buffers

- Read miss?
 - Load can't go on without the data, it must stall
- Write miss?
 - Technically, no instruction is waiting for data, why stall?
- **Store buffer**: a small buffer
 - Stores put address/value to store buffer, **keep going**
 - Store buffer writes stores to D\$ in the background
 - Loads must search store buffer (in addition to D\$)
 - + Eliminates stalls on write misses (mostly)
 - Creates some problems (later)
- Store buffer vs. writeback-buffer
 - Store buffer: "in front" of D\$, for hiding store misses
 - Writeback buffer: "behind" D\$, for hiding writebacks



Why? To Hide Store Miss Latency

- Why? Why Allow Such Odd Behavior?
 - Reason #1: hiding store miss latency
- Recall (back from caching unit)
 - Hiding store miss latency
 - How? Store buffer
- Said it would complicate multiprocessors
 - Yes. It does.
 - By allowing reordering of store and load (to different addresses)

- Example:

thread 1	thread 2
store 1 \rightarrow y load x	store 1 \rightarrow x load y

 - Both stores miss cache, are put in store buffer
 - Loads hit, receive value before store completes, sees "old" value

Shared Memory Example #1: Answer

- Initially: all variables zero (that is, x is 0, y is 0)

thread 1	thread 2
store 1 → y load x	store 1 → x load y

- What value pairs can be read by the two loads?

store 1 → y load x store 1 → x load y (x=0, y=1)	store 1 → y store 1 → x load x load y (x=1, y=1)	store 1 → y store 1 → x load y load x (x=1, y=1)
store 1 → x load y store 1 → y load x (x=1, y=0)	store 1 → x store 1 → y load y load x (x=1, y=1)	store 1 → x store 1 → y load x load y (x=1, y=1)

- What about (x=0,y=0)? Yes! (for x86, SPARC, ARM, PowerPC)

Why? Simplify Out-of-Order Execution

- Why? Why Allow Such Odd Behavior?
 - Reason #2: simplifying out-of-order execution
- One key benefit of out-of-order execution:
 - Out-of-order execution of loads to (same or different) addresses

thread 1

thread 2

<pre>store 1 → a store 1 → flag</pre>	<pre>loop: if (flag == 0) goto loop load a</pre>
---------------------------------------	--

- **Uh, oh.**
- Two options for hardware designers:
 - Option #1: **allow** this sort of “odd” reordering (“not my problem”)
 - Option #2: hardware **detects & recovers** from such reorderings
 - Scan load queue (LQ) when cache block is invalidated
- And store buffers on some systems reorder stores by same thread to different addresses (as in thread 1 above)

Simplifying Out-of-Order Execution

- Two options:
 - Option #1: **allow** this sort of “odd” reordering
 - Option #2: add more hardware, **prevent** these reorderings
- How to prevent?
 - Scan the Load Queue (LQ) on stores from **other** threads
 - Flush and rollback on conflict
- How to detect these stores from other threads?
 - Leverage cache coherence!
 - As long as the block remains in the cache...
 - Another core can't write to it
 - Thus, anytime a block leaves the cache (invalidation or eviction)...
 - Scan the load queue. If any loads to the address have executed but not committed, squash the pipeline and restart

Shared Memory Example #2: Answer

- **Initially: all variables zero** (flag is 0, a is 0)

thread 1	thread 2
store 1 → a store 1 → flag	loop: if (flag == 0) goto loop load a

- What value can be read by “load a”?
 - “load a” can see the value “1”
- Can “load a” read the value zero? (same as last slide)
 - **Yes! (for ARM, PowerPC, Itanium, and Alpha)**
 - **No! (for Intel/AMD x86, Sun SPARC, IBM 370)**
 - Assuming the compiler didn't reorder anything...

Restoring Order (Hardware)

- Sometimes we need ordering (mostly we don't)
 - Prime example: ordering between "lock" and data
- How? insert **Fences (memory barriers)**
 - Special instructions, part of ISA
- Example
 - Ensure that loads/stores don't cross synchronization operations

```
lock acquire
fence
"critical section"
fence
lock release
```
- How do fences work?
 - They stall execution until write buffers are empty
 - Makes lock acquisition and release slow(er)
- **Use synchronization library, don't write your own**

Restoring Order (Software)

- These slides have focused mostly on **hardware** reordering
 - But the compiler also reorders instructions (reason #3)
- How do we tell the **compiler** to not reorder things?
 - Depends on the language...
- In Java:
 - The built-in “synchronized” constructs informs the compiler to limit its optimization scope (prevent reorderings across synchronization)
 - Or, programmer uses “volatile” keyword to explicitly mark variables
 - Java compiler inserts the hardware-level ordering instructions
- In C/C++:
 - More murky, as pre-2011 language doesn’t define synchronization
 - Lots of hacks: “inline assembly”, volatile, atomic keyword (new!)
 - Programmer may need to explicitly insert hardware-level fences
- **Use synchronization library, don’t write your own**

Recap: Four Shared Memory Issues

1. Cache coherence

- If cores have private (non-shared) caches
- How to make writes to one cache “show up” in others?

2. Parallel programming

- How does the programmer express the parallelism?

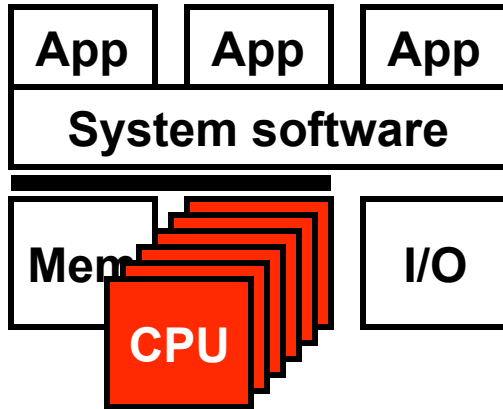
3. Synchronization

- How to regulate access to shared data?
- How to implement “locks”?

4. Memory consistency models

- How to keep programmer sane while letting hardware optimize?
- How to reconcile shared memory with compiler optimizations, store buffers, and out-of-order execution?

Summary



- Thread-level parallelism (TLP)
- Shared memory model
 - Multiplexed uniprocessor
 - Hardware multithreading
 - Multiprocessing
- Cache coherence
 - Valid/Invalid, MSI, MESI
- Parallel programming
- Synchronization
 - Lock implementation
 - Locking gotchas
 - Transactional memory
- Memory consistency models