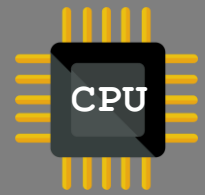


Topic 7

Introduction to Multicore I

Xinfei Guo
xinfei.guo@sjtu.edu.cn

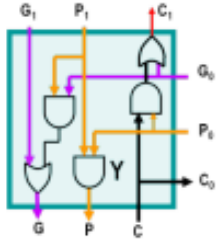
July 6th, 2022

T7 learning goals

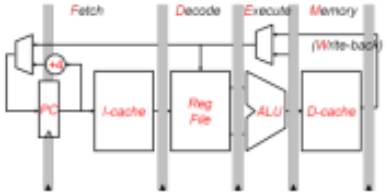
- Multicore/Multiprocessing
  - Section I
    - Motivation
    - SMT vs. Multicore
    - Communication
    - Cache coherence protocol
  - Section II
    - Memory consistency
    - Parallel programming
    - Dark silicon
    - Roofline model

Spectrum of Parallelism

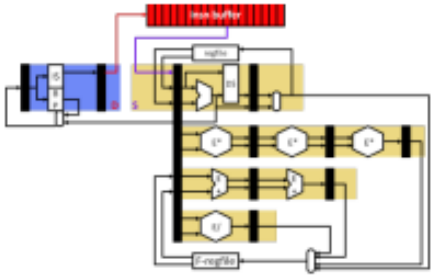
Bit-level



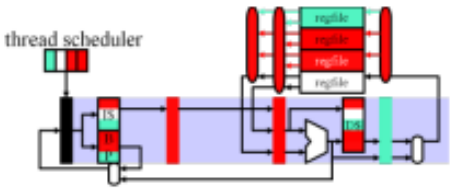
Pipelining



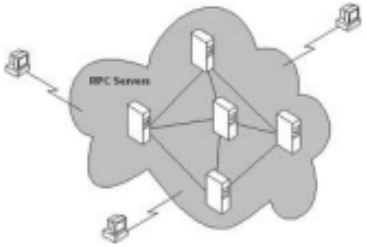
ILP



Multithreading
Multiprocessing



Distributed



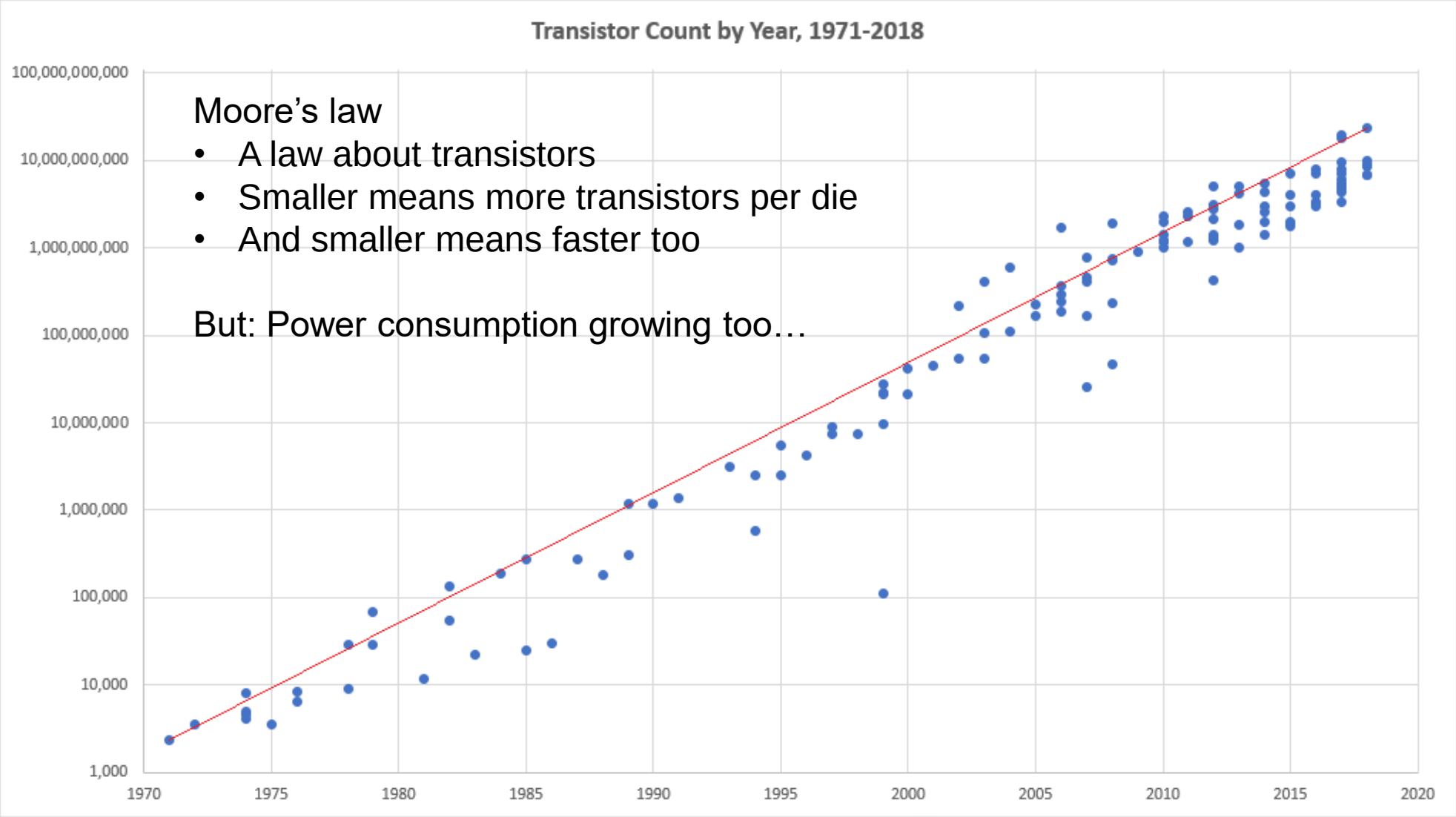
Slide Credit: Prof. Ron Dreslinski @ UMich, EECS 470

Why Muliprocessing?

- 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
  - $$$, weighs more than 1 ton
- Today: multicore is everywhere
  - Can't buy a single-core smartphone…
    - …or even smart watch!

Slide Credit: Prof. Joe Devietti @ UPenn

Recall: Moore's Law



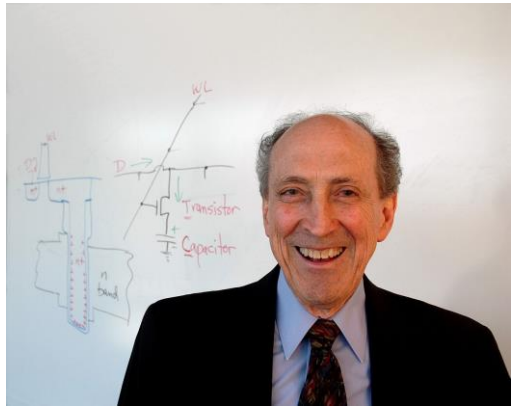
Moore's law

- A law about transistors
- Smaller means more transistors per die
- And smaller means faster too

But: Power consumption growing too…

Dennard Scaling

- In 1974, Robert Dennard observed that power density remained **constant** for a given area of silicon (square nanometers) when the dimension of the transistor shrank, thanks to technology improvements (Dennard Scaling).
- Capacitance is related to area
  - So, as the size of the transistors shrunk, and the voltage was reduced, circuits could operate at higher frequencies at the same power



- Power = alpha * CFV^2
  - Alpha – percent time switched
  - C = capacitance
  - F = frequency
  - V = voltage

source: Wiki

End of Dennard Scaling

- Dennard scaling ignored the “leakage current” and “threshold voltage”, which establish a baseline of power per transistor
- As transistors get smaller, power density increases because these don’t scale with size
- These created a “Power Wall” that has limited practical processor frequency to around 4 GHz since 2006

End of Dennard Scaling

source: https://silvanogai.github.io/posts/dennard/

Power Wall

Power = alpha * capacitance * voltage^2 * frequency

In practice: Power ~ voltage^3

Reducing voltage helps (a lot)

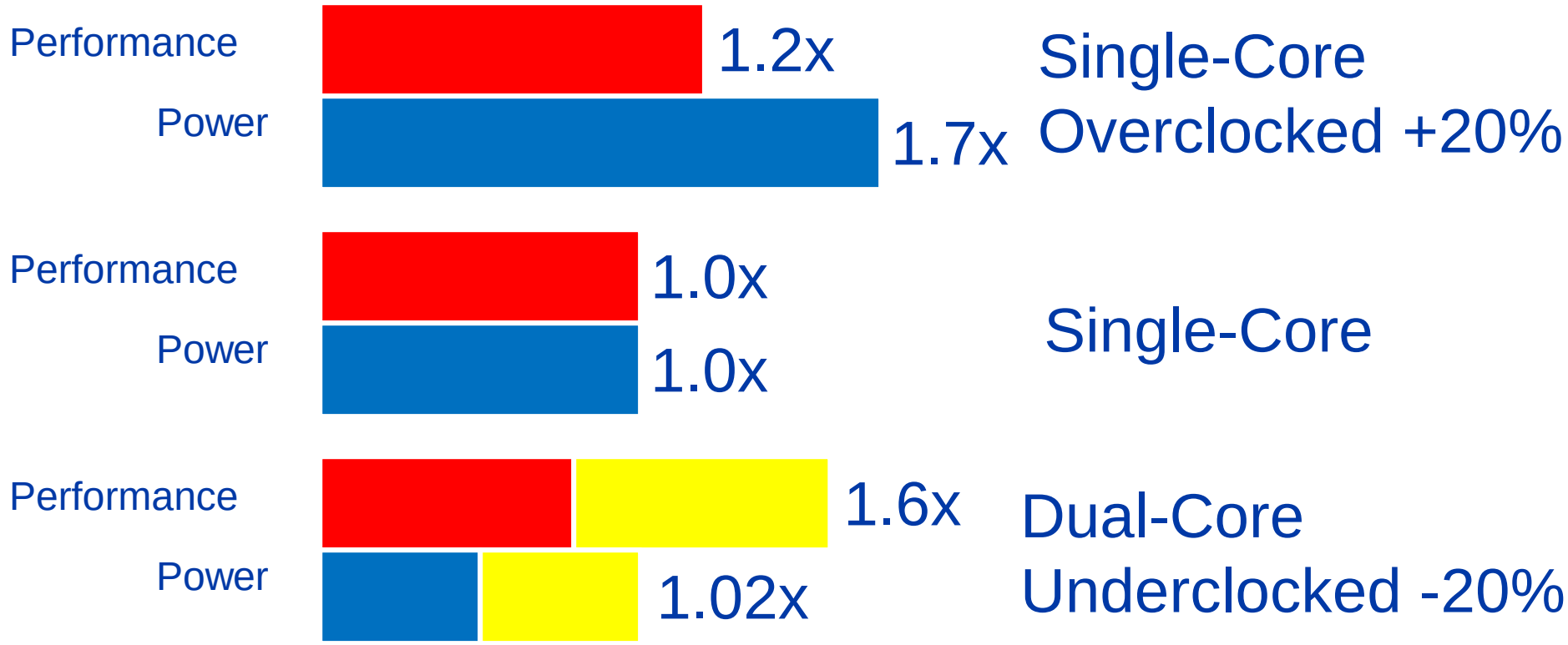
... so does reducing clock speed

Better cooling helps

The power wall

- We can't reduce voltage further (leakage)
- We can't remove more heat

Why Multicore?



Slide Credit: Prof. Hakim Weatherspoon @ Cornell

Power Efficiency

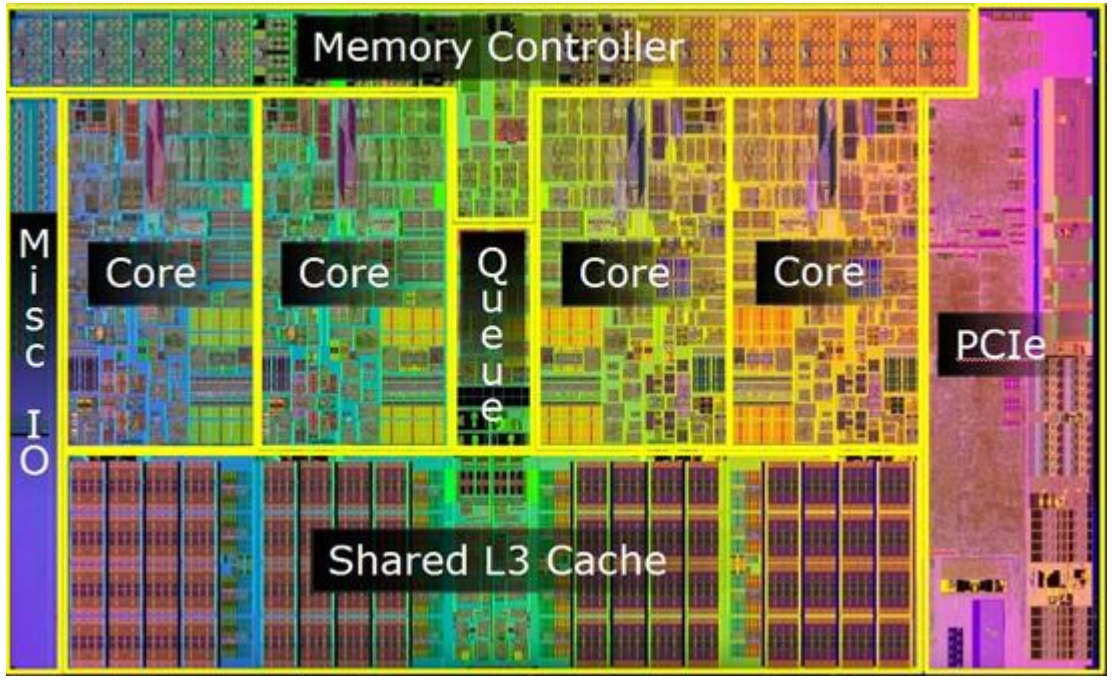
Q: Does multiple issue / ILP cost much?

A: Yes. → Dynamic issue and speculation requires power

CPU	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
Core i5 Nehal	2010	3300MHz	14	4	Yes	1	87W
Core i5 Ivy Br	2012	3400MHz	14	4	Yes	8	77W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

Why Multicore?

- Difficult to make single-core clock frequencies even higher
- Deeply pipelined circuits:
  - heat problems
  - speed of light problems
  - difficult design and verification
  - large design teams necessary
  - server farms need expensive air-conditioning

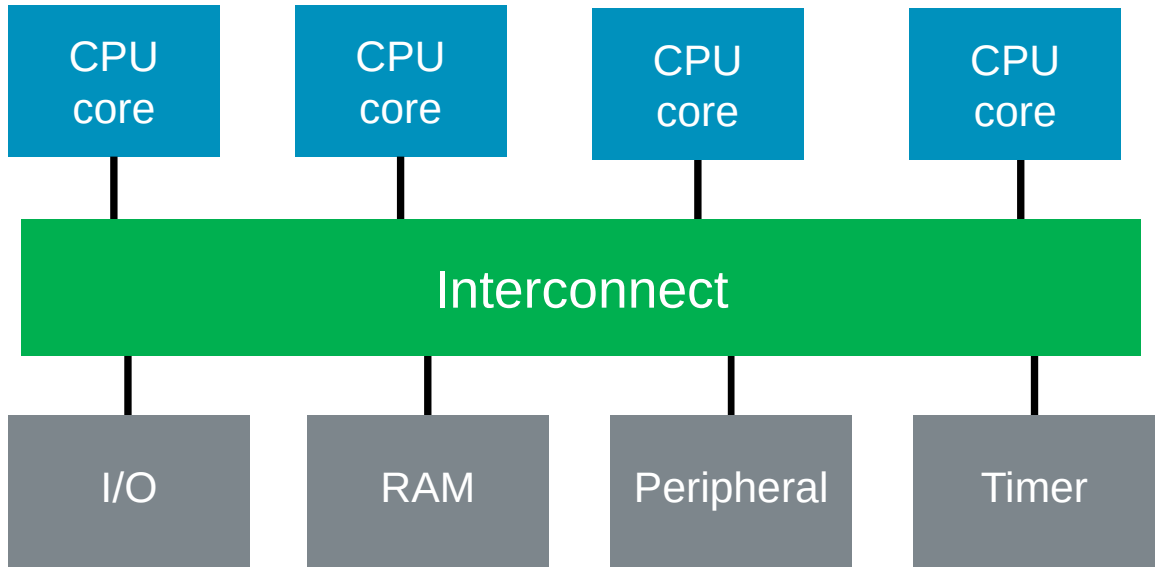


source: Intel

- Many new applications are multithreaded
- General trend in computer architecture (shift towards more parallelism)

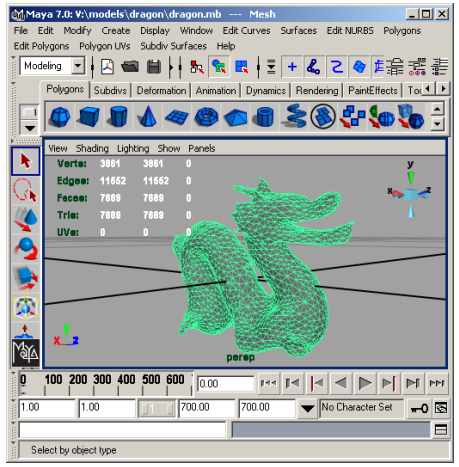
Multicore/Multiprocessing Overview

- A **multi-core** microprocessor is one that combines two or more independent processors into a single package, often a single integrated circuit (IC).
- A **dual-core** device contains two independent microprocessors.
- In general, multi-core microprocessors allow a computing device to exhibit some form of thread-level parallelism (**TLP**) without including multiple microprocessors in separate physical packages.
- Multi-core processors are MIMD: Different cores execute different threads (Multiple Instructions), operating on different parts of memory (Multiple Data).
- Multi-core is **usually** a shared memory multiprocessor:
  All cores share the same memory

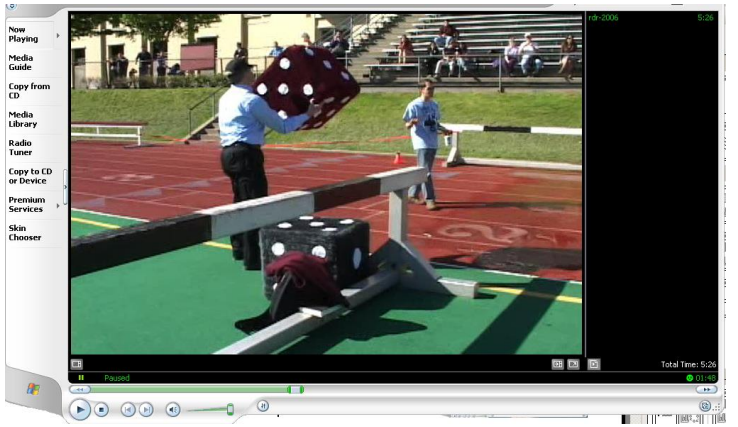


What applications benefit from multi-core?

- Database servers
- Web servers (Web commerce)
- Compilers
- Multimedia applications
- Scientific applications, CAD/CAM
- In general, applications with *Thread-level parallelism* (as opposed to instruction-level parallelism)



Each can run on its own core



More examples

- Editing a photo while recording a TV show through a digital video recorder
- Downloading software while running an anti-virus program
- “Anything that can be threaded today will map efficiently to multi-core”
- BUT: some applications difficult to parallelize

Multicore Design Considerations

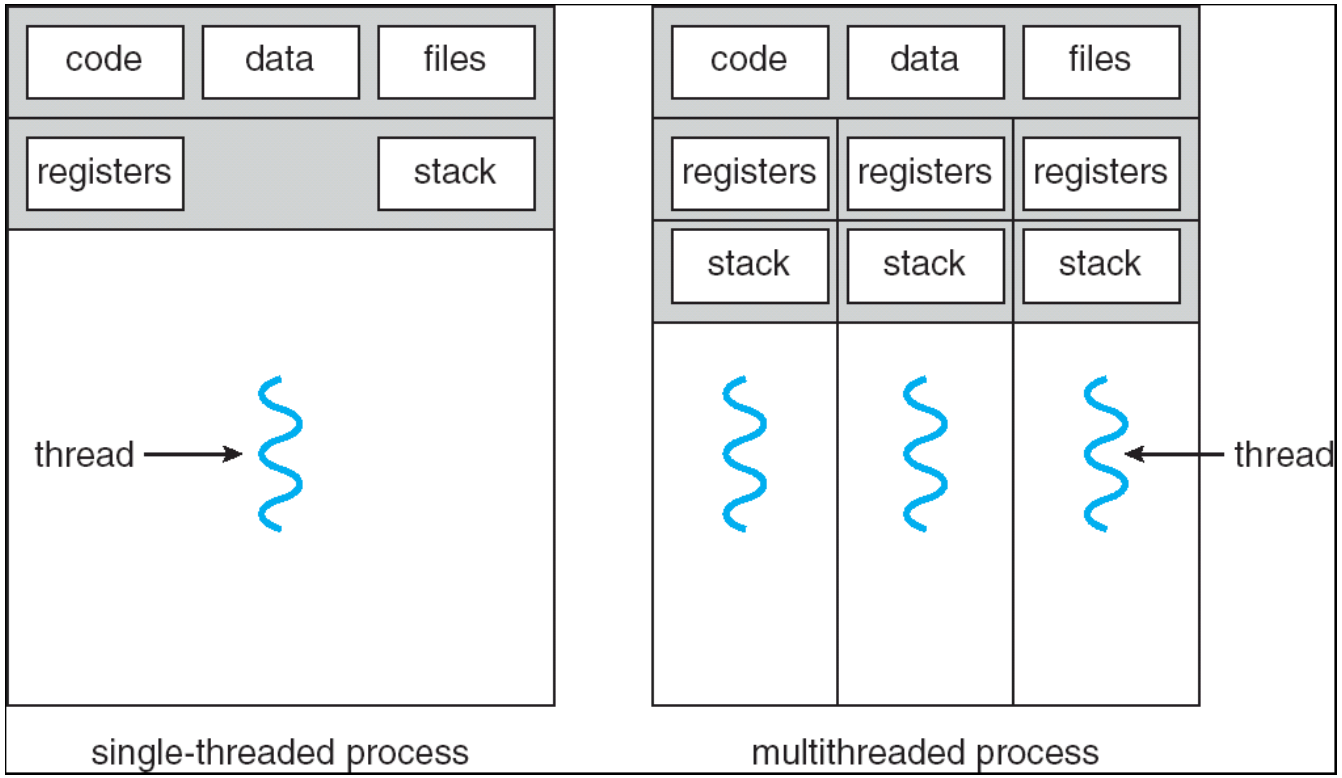
- Cores in a multicore processor are connected together and can collaborate.
- Why not staying with multi-threading style?
- There are a number of challenges to consider when creating a system like this.
  - How do cores communicate with each other?
  - How is data synchronized?
  - How do we ensure that cores don't get stale data when it's been modified by other cores?
  - How do cores see the ordering of events coming from different cores?
  - How to program multicore?
- We'll explore each of these by considering the concepts of
  - Shared memory and message passing
  - Cache coherence
  - Memory consistency
  - Parallel programming

MULTICORE VS SMT

Recall thread

Process: multiple threads, code, data and OS state

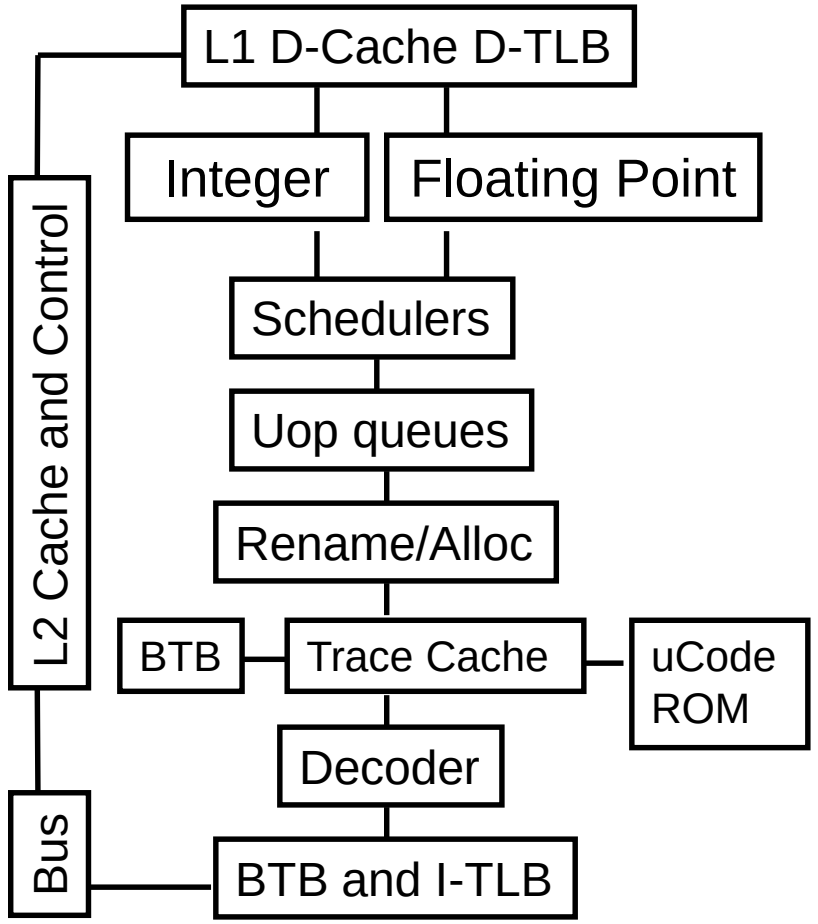
Threads: share code, data, files, **not** regs or stack



A technique complementary to multi-core: Simultaneous multithreading

- Problem addressed:
  The processor pipeline can get stalled:
  - Waiting for the result of a long floating point (or integer) operation
  - Waiting for data to arrive from memory

  Other execution units wait unused



Source: Intel

Simultaneous multithreading (SMT)

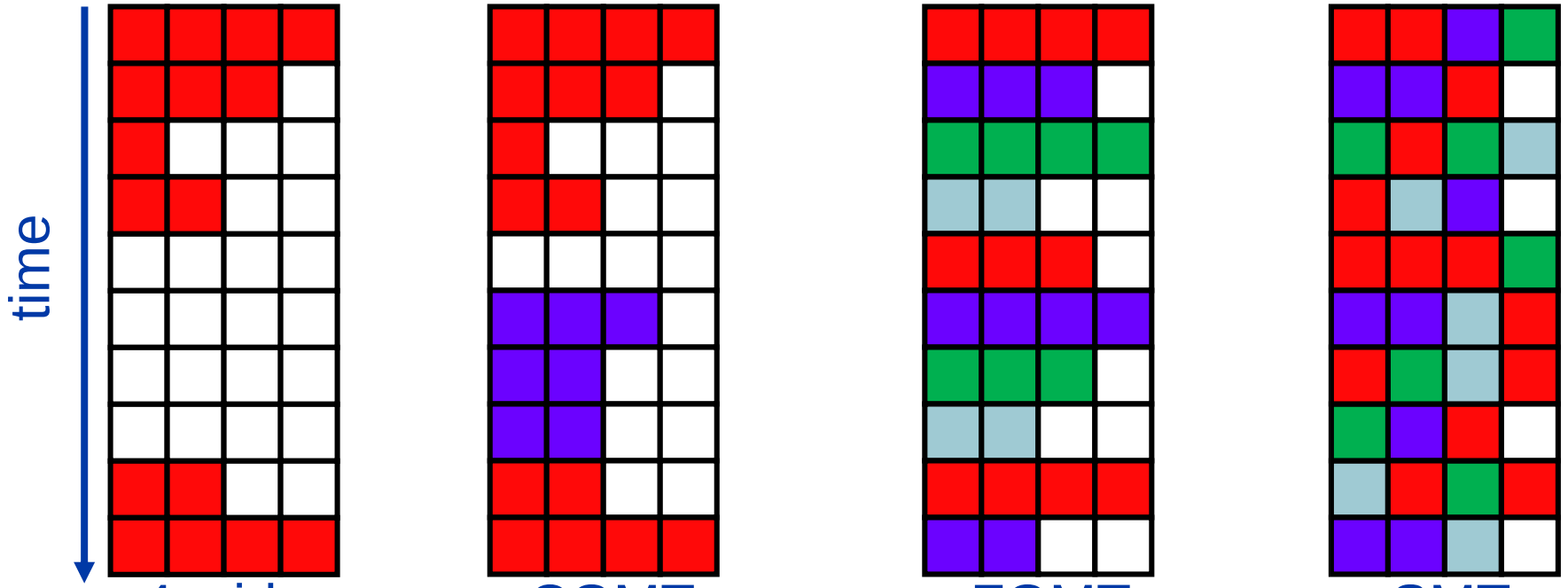
- Permits multiple independent threads to execute SIMULTANEOUSLY on the SAME core
- Weaving together multiple “threads” on the same core

- Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units

Standard Multithreading Picture

Time evolution of issue slots

- Color = thread, white = no instruction (bubble)



4-wide Superscalar

CGMT

Switch to thread B on thread A L2 miss

FGMT

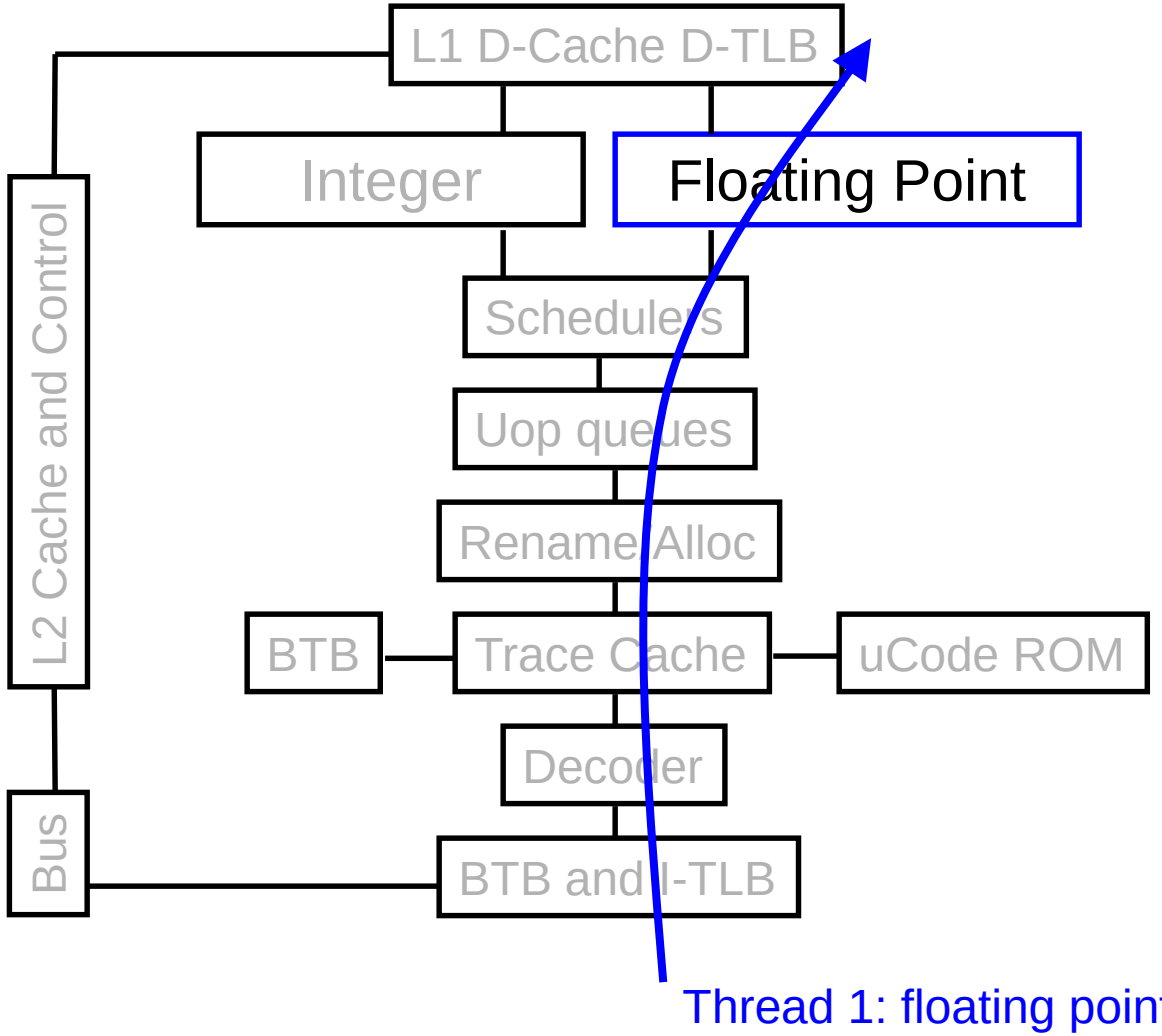
Switch threads every cycle

SMT

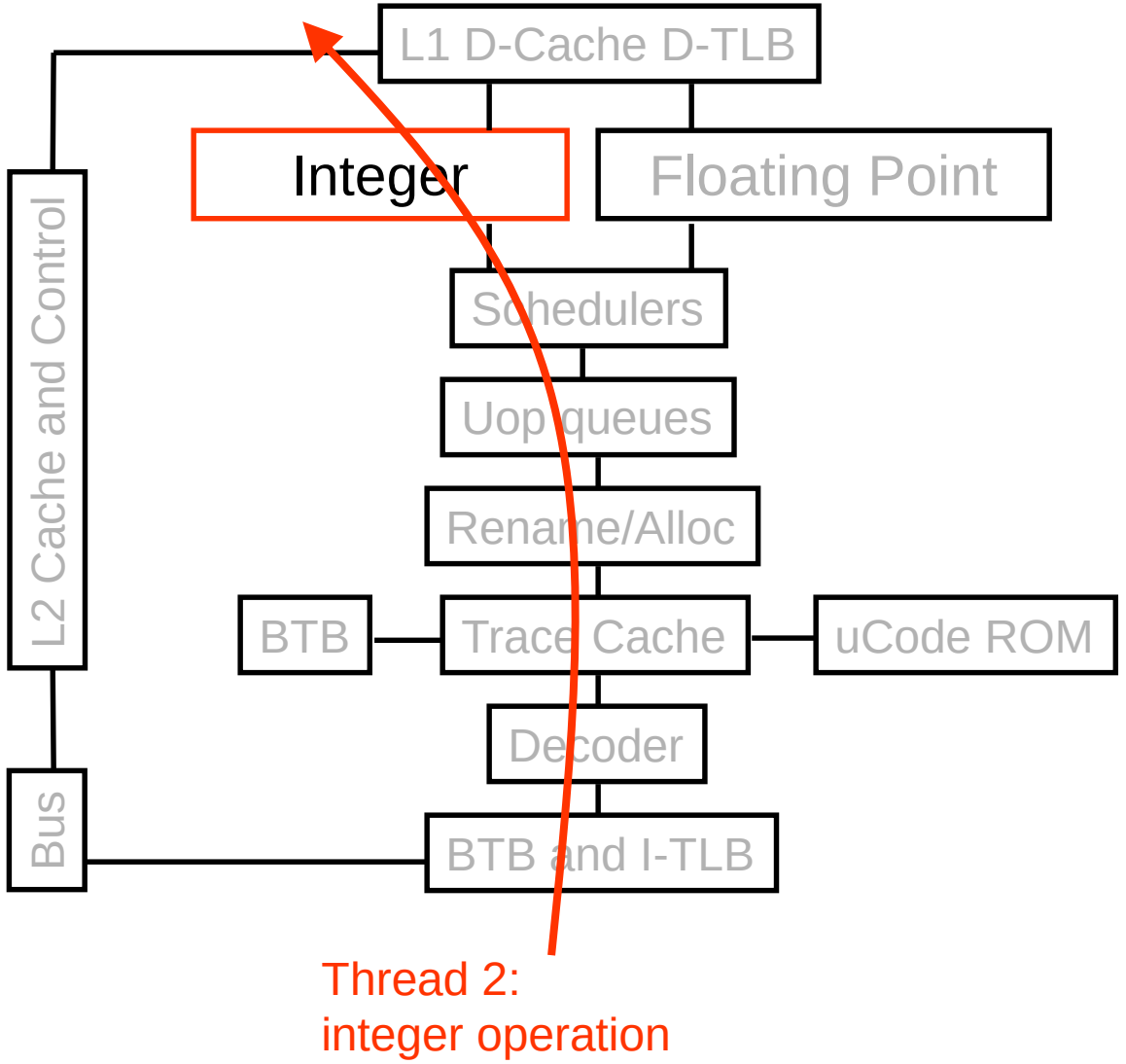
Insns from multiple threads coexist

CG: coarse grain
FG: fine grain

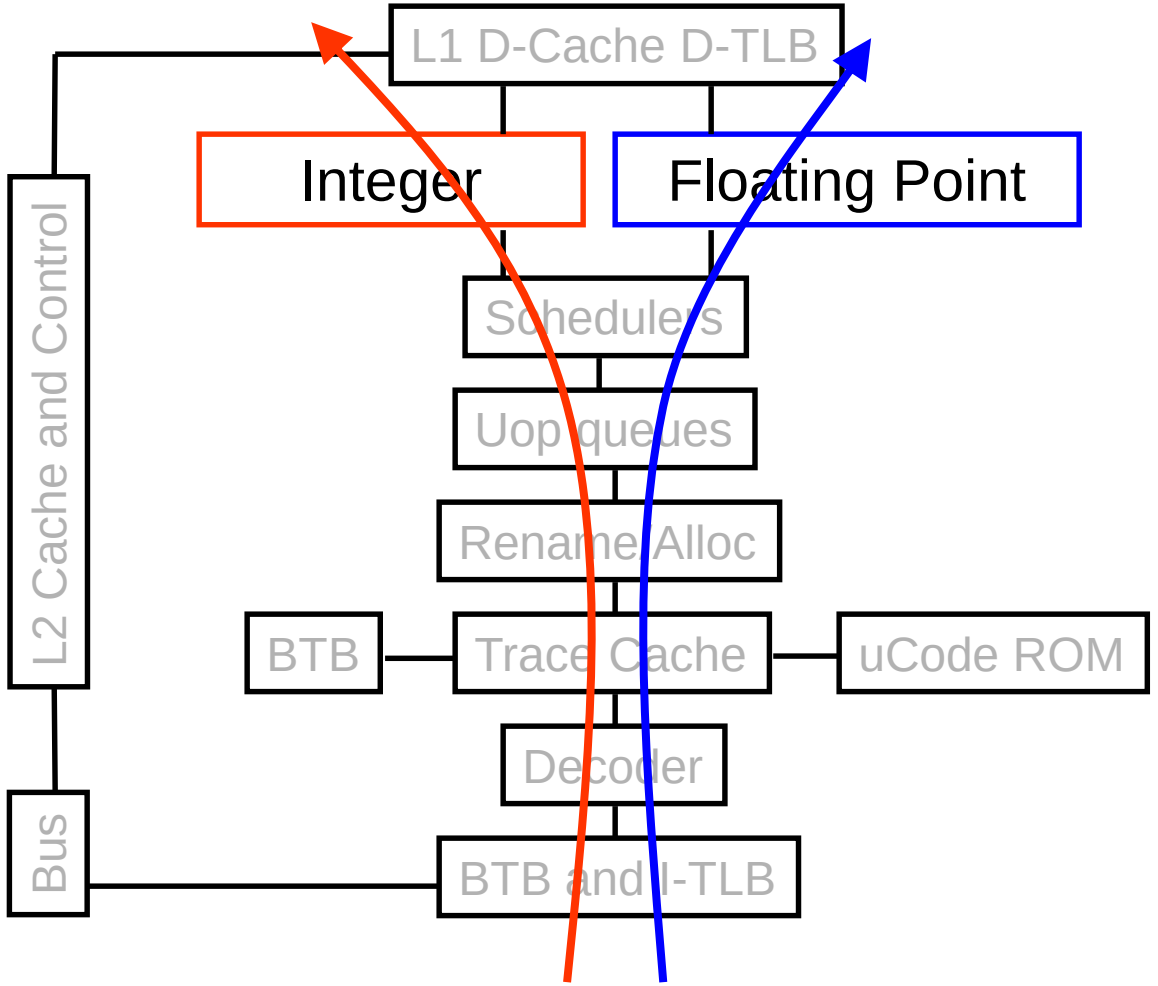
Without SMT, only a single thread can run at any given time



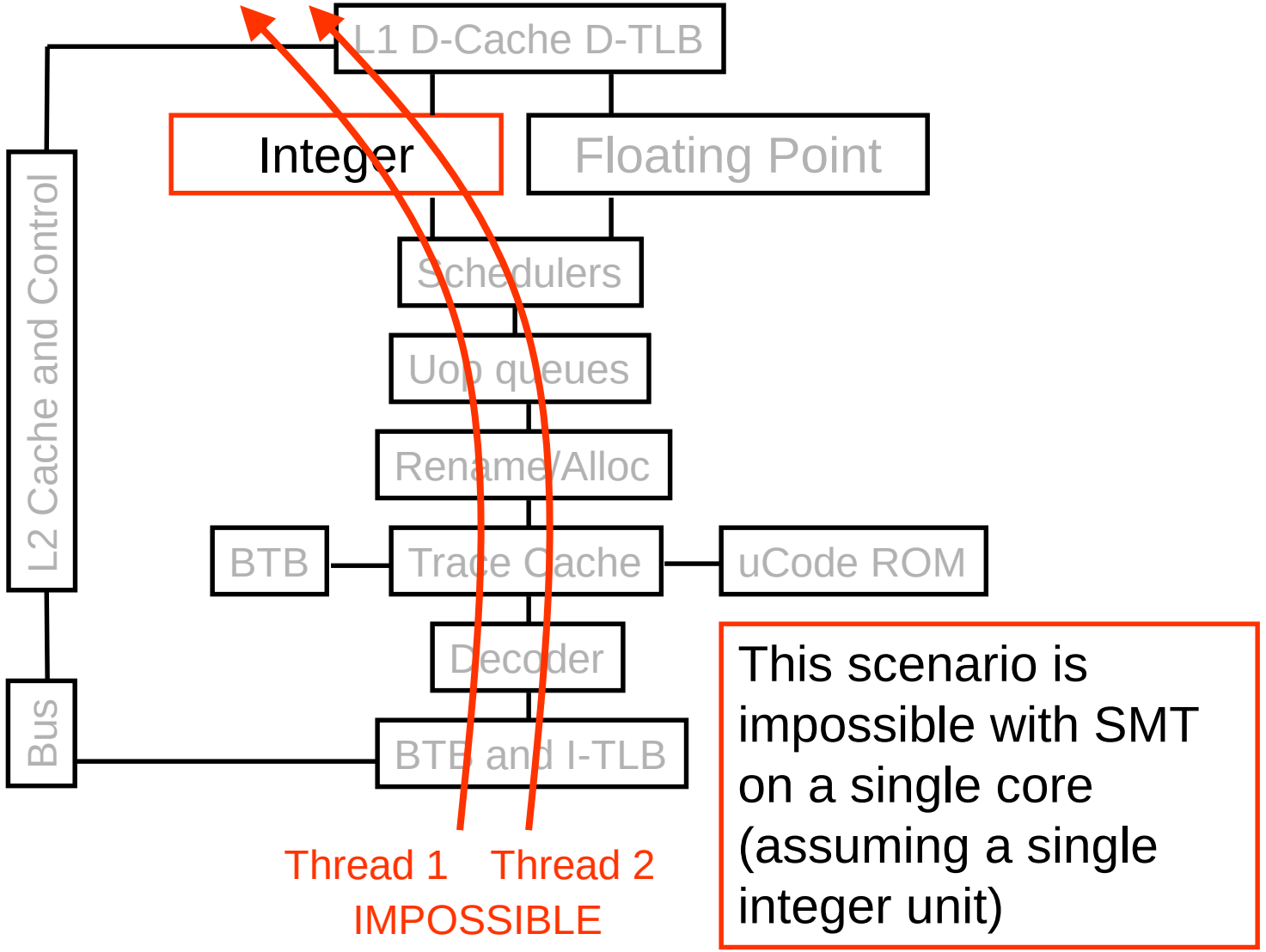
Without SMT, only a single thread can run at any given time



SMT processor: both threads can run concurrently



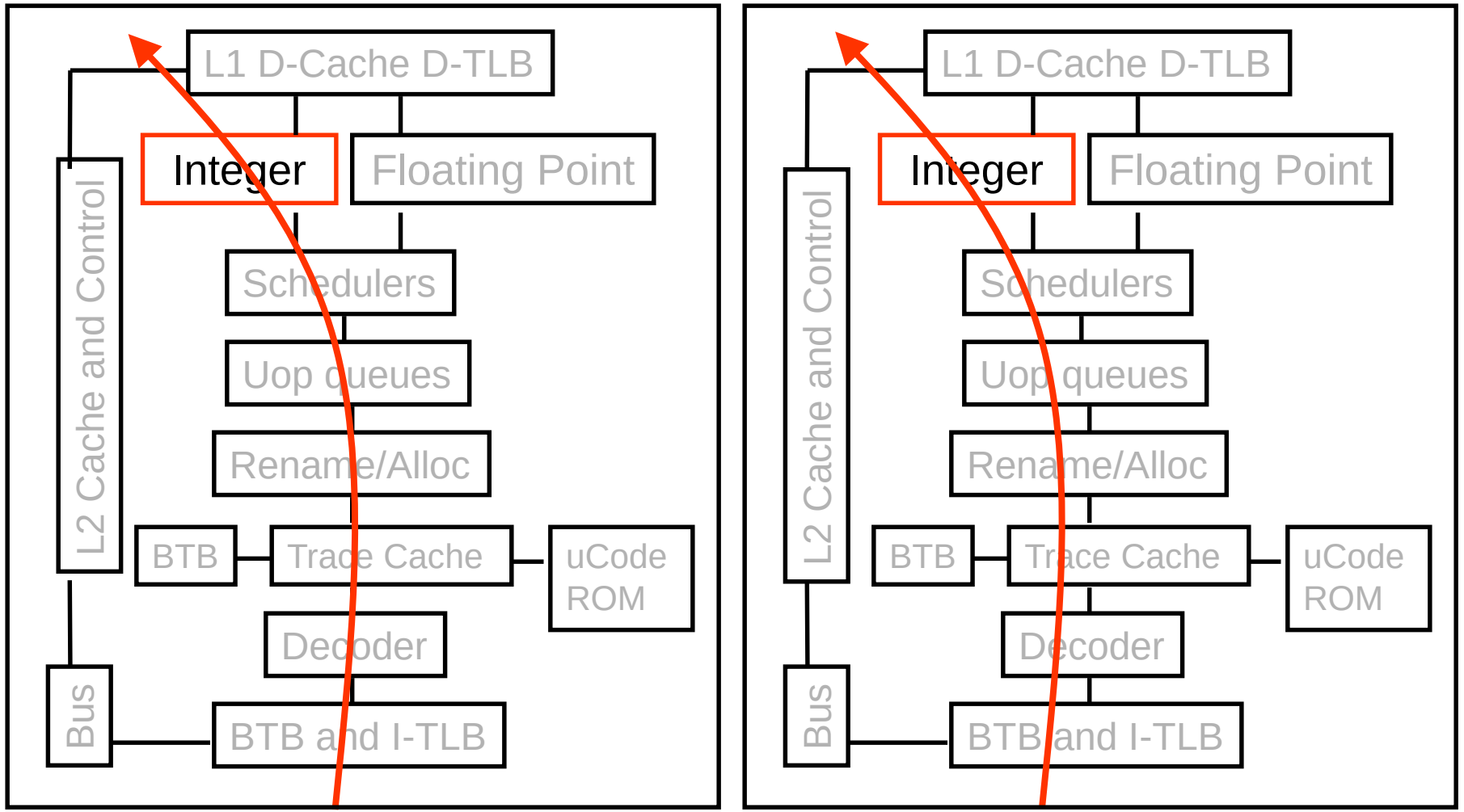
But: Can't simultaneously use the same functional unit



SMT not a “true” parallel processor

- Enables better threading (e.g. up to 30%)
- OS and applications perceive each simultaneous thread as a separate “virtual processor”
- The chip has only a single copy of each resource
- Recall GPU (SIMT)
- Compare to multi-core:
  each core has its own copy of resources

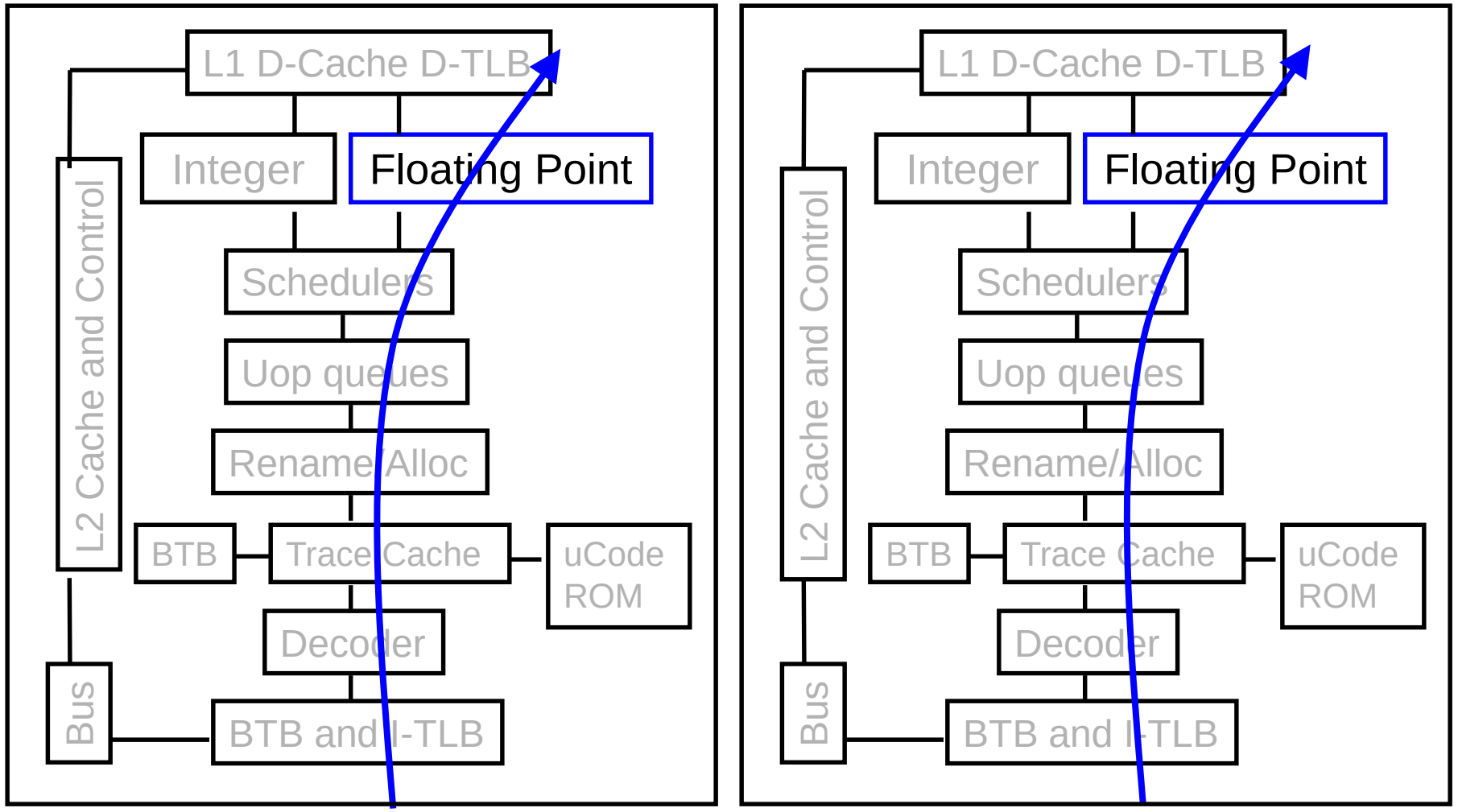
Multi-core: threads can run on separate cores



Thread 1

Thread 2

Multi-core: threads can run on separate cores



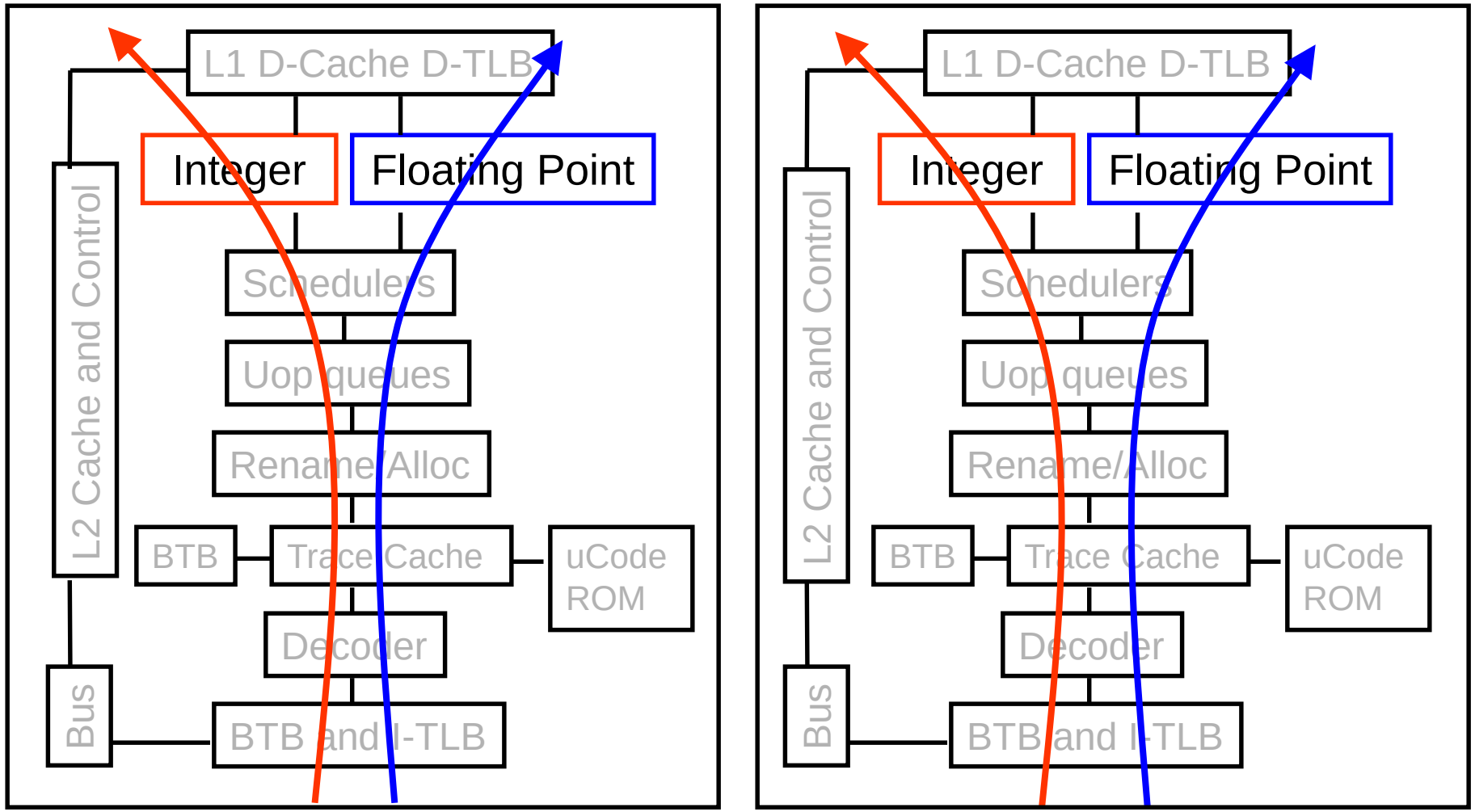
Thread 3

Thread 4

Combining Multi-core and SMT

- Cores can be SMT-enabled (or not)
- The different combinations:
  - Single-core, non-SMT: standard uniprocessor
  - Single-core, with SMT
  - Multi-core, non-SMT
  - Multi-core, with SMT: our fish machines
- The number of SMT threads:
  2, 4, or sometimes 8 simultaneous threads
- Intel calls them “hyper-threads”

SMT Dual-core: all four threads can run concurrently



Comparison: multi-core vs SMT

- Advantages/disadvantages?

Comparison: multi-core vs SMT

- Multi-core:
  - Since there are several cores,
    each is smaller and not as powerful
    (but also easier to design and manufacture)
  - However, great with thread-level parallelism
- SMT
  - Can have one large and fast superscalar core
  - Great performance on a single thread
  - Mostly still only exploits instruction-level parallelism

Hyperthreading (HT)

Multi-Core vs. Multi-Issue vs. HT

Programs:	*N*	*1*	*N*
Num. Pipelines:	*N*	*1*	1
Pipeline Width:	*1*	*N*	*N*

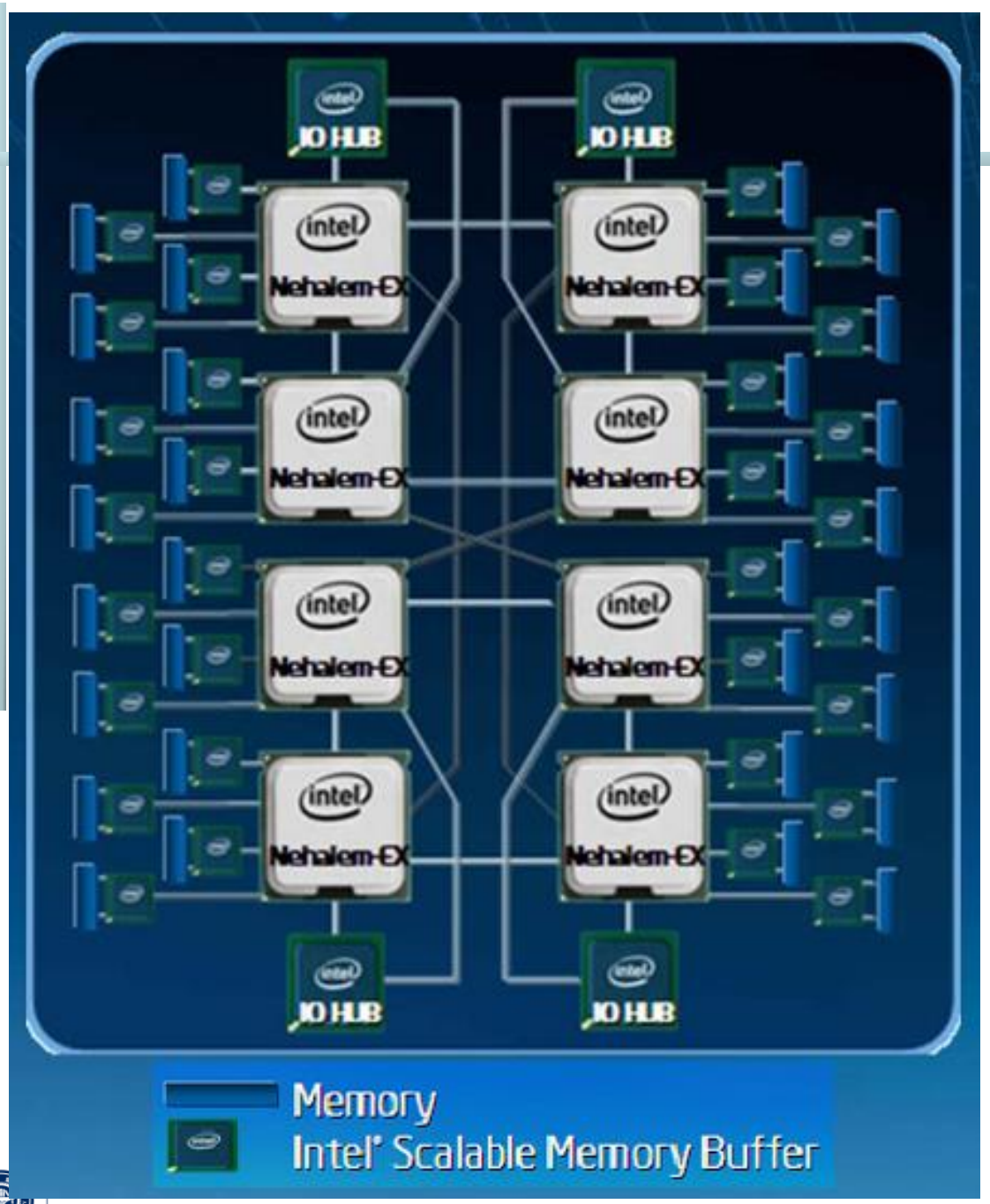
Hyperthreads

- HT = MultiIssue + extra PCs and registers – dependency logic
- HT = MultiCore – redundant functional units + hazard avoidance

Hyperthreads (Intel)

- Illusion of multiple cores on a single core
- Easy to keep HT pipelines full + share functional units

Example: All of the above



8 die (aka 8 sockets)
4 core per socket
2 HT per core

Note: a socket is a processor, where each processor may have multiple processing cores, so this is an example of a multiprocessor multicore hyperthreaded system

COMMUNICATION

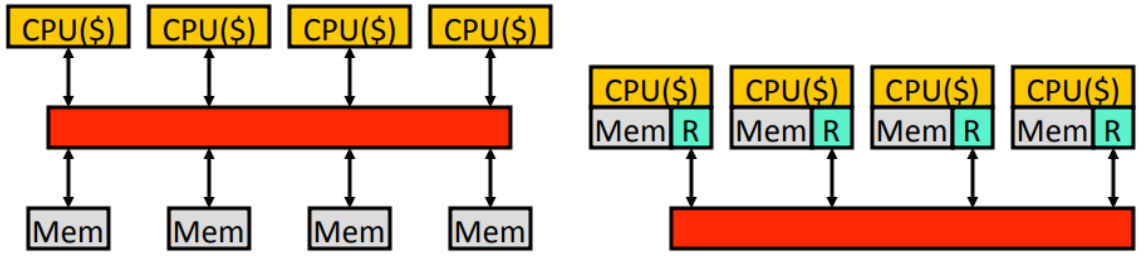
Communication

Message Passing

- In this paradigm, applications running on each core wrap up the data they want to communicate into messages sent to other cores.
- Explicit communication via *send* and *receive* operations
- Synchronization is implicit via blocks of messages.
- Distributed memory: In this model, each processor has its own (small) local memory, and its content is not replicated anywhere else

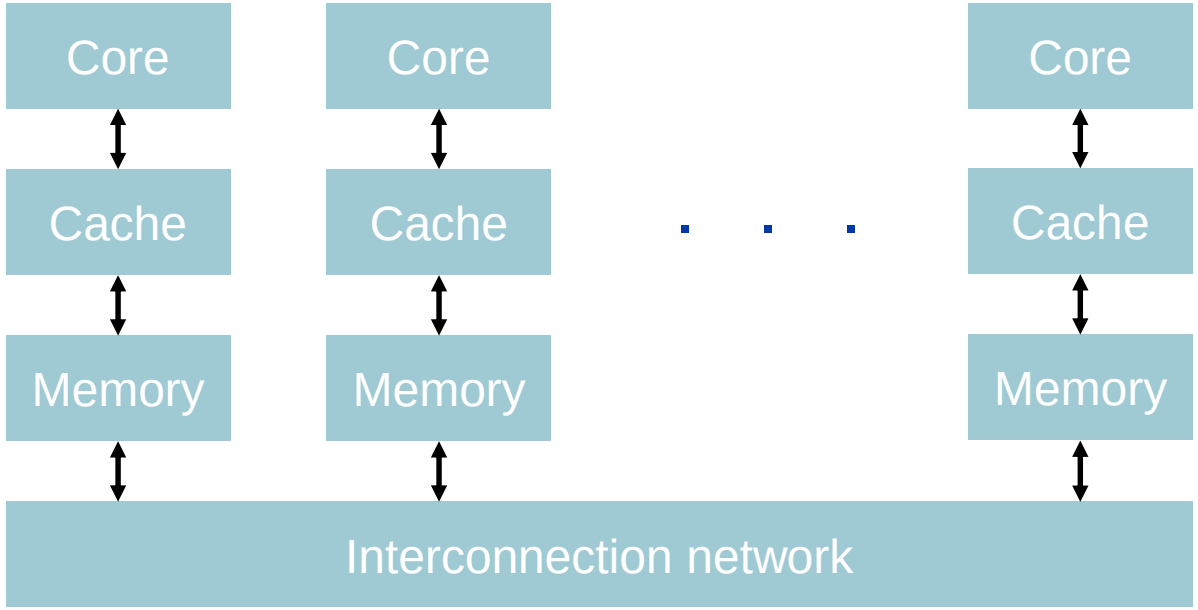
Shared Memory

- In this paradigm, there is a shared memory address space accessible to all cores, where they can read and write data.
- Implicit communication via memory accesses
- Synchronization is performed using atomic memory operations.



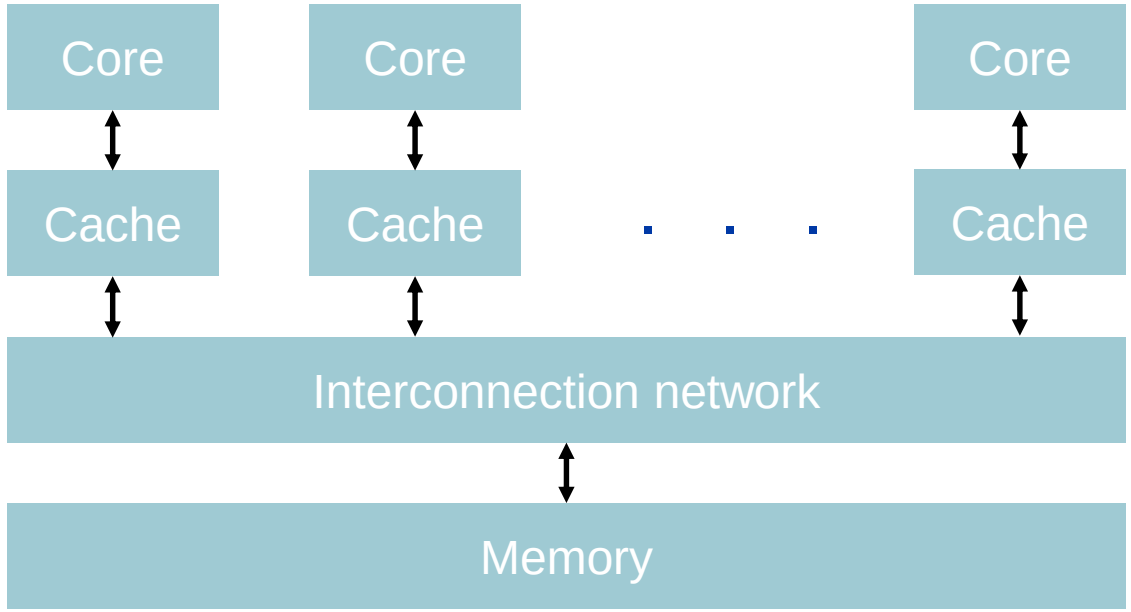
Message Passing

- Cores do not rely on a shared memory space.
- Each processing element (PE) has its core, data, I/O.
  - Explicit I/O to communicate with other cores
  - Synchronization via sending and receiving messages
- Advantages
  - Less hardware, easier to design
  - Focuses attention on costly non-local operations



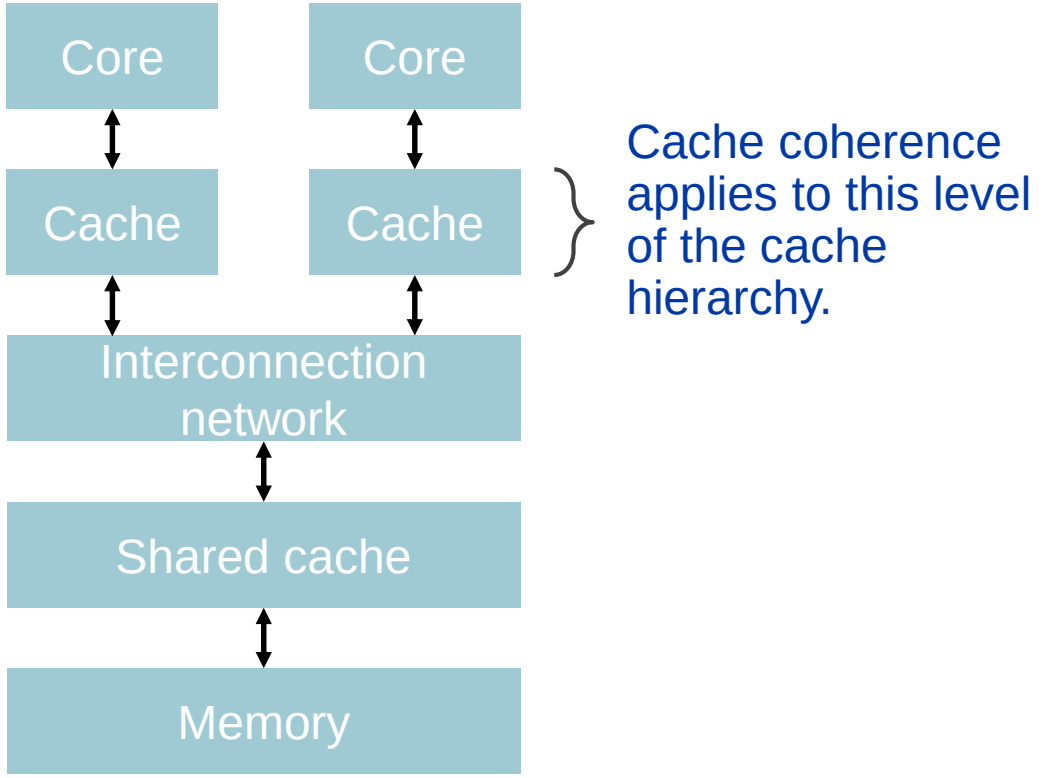
Shared Memory

- Cores share a memory that they see as a single address space.
- Cores may have caches holding data.
  - Communication involves reading and writing to locations in memory.
  - Synchronization via atomic operations to modify memory
    - Specific instructions provided in the ISA
    - The hardware guarantees correct operation



Shared Memory with Caches

- Caching shared-memory data has to be handled carefully.
  - The cache stores a copy of some of the data in memory.
- In particular, writes to the data must be dealt with correctly.
  - A core may write to data in its own cache.
  - This makes copies in other caches become stale.
  - The version in memory won't get updated immediately either, if it's a write-back cache.
  - In these situations, there is a danger of one core subsequently reading a stale (old) value.



Distributed vs shared caches

- Advantages of distributed:
  - They are closer to core, so faster access
  - Reduces contention
- Advantages of shared:
  - Threads on different cores can share the same cache data
  - More cache space available if a single (or a few) high-performance thread runs on the system
  - Matches the programmer's view of the system
  - Hardware handles communication implicitly

CACHE COHERENCE

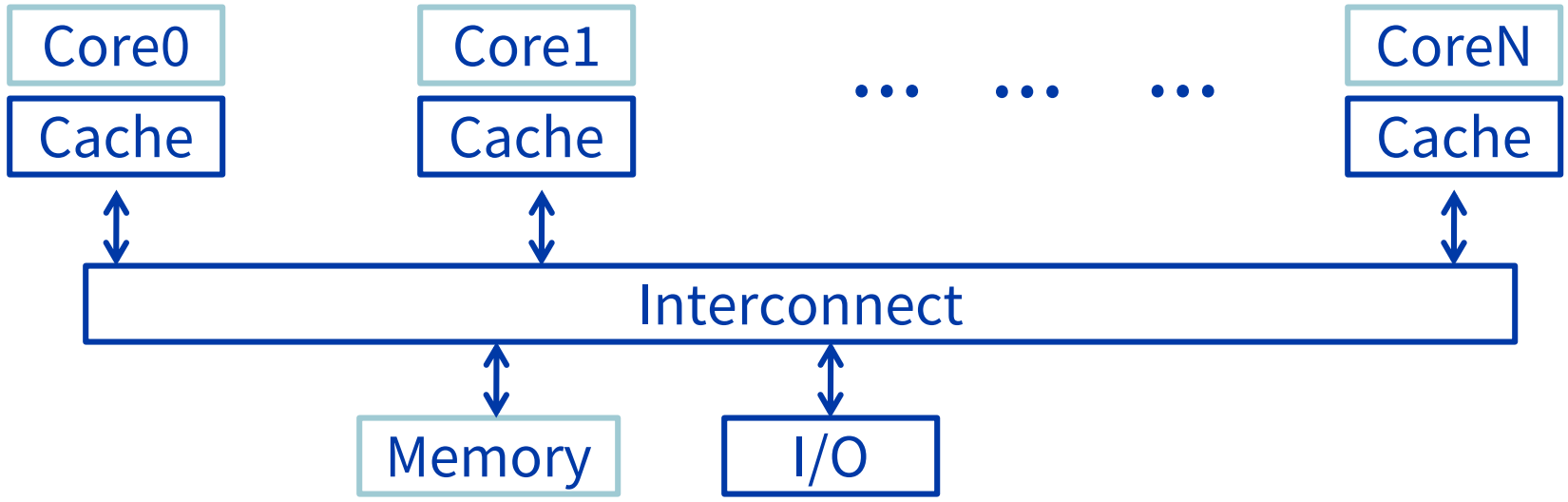
Recall: 3C Model -> 4C Model

- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - **Would miss even in infinite cache**
  - **Capacity**: miss caused because cache is too small
    - **Would miss even in fully associative cache**
    - Identify? Consecutive accesses to block separated by access to at least N other distinct blocks (N is number of frames in cache)
  - **Conflict**: miss caused because cache associativity is too low
    - Identify? **All other misses**
  - **COHERENCE: miss due to external invalidations**
    - **ONLY IN SHARED MEMORY MULTIPROCESSORS**

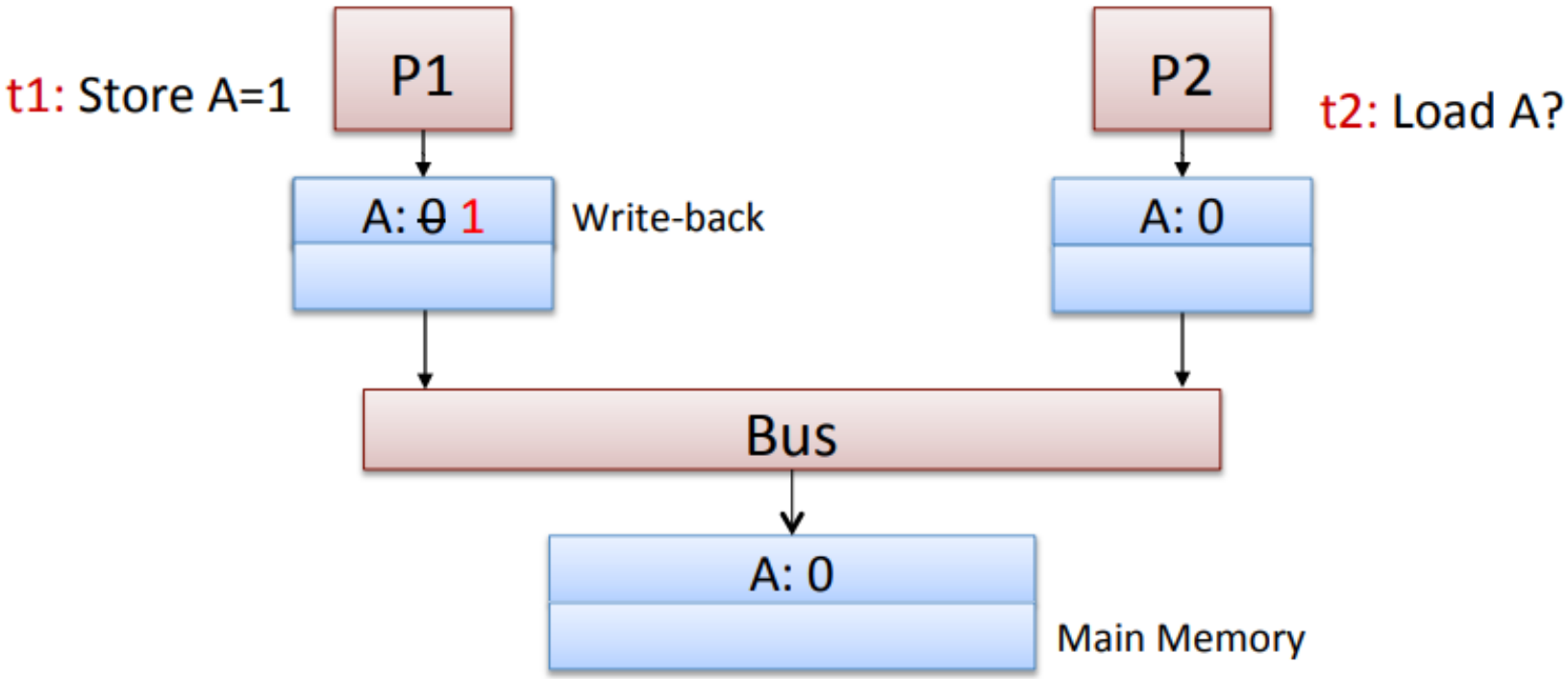
Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

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



Cache Coherency Problem



Cache Coherency Problem

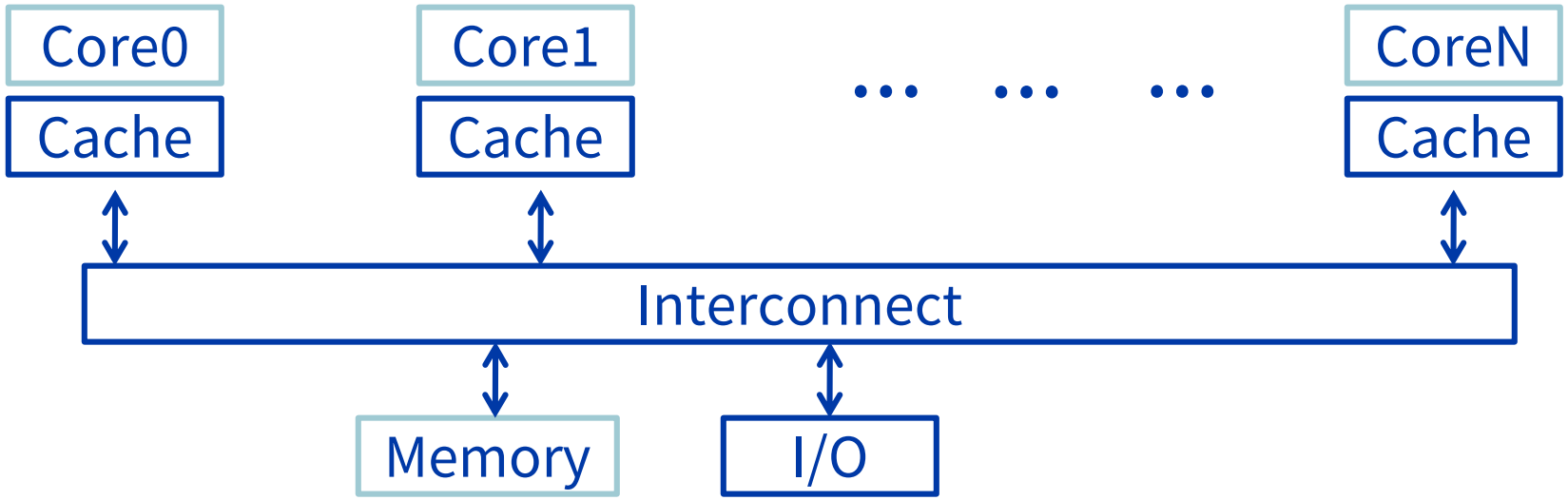
Thread A (on Core0)

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

Thread B (on Core1)

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

What will the value of x be after both loops finish?



Cache Coherency Problem

Thread A (on Core0)

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

Thread B (on Core1)

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

What will the value of x be after both loops finish?

(x starts as 0)

A. 6

B. 8

C. 10

D. Could be any of the above

E. Couldn't be any of the above

Cache Coherency Problem

Thread A (on Core0)

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

Thread B (on Core1)

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

What will the value of x be after both loops finish?

(x starts as 0)

A. 6

B. 8

C. 10

D. Could be any of the above

E. Couldn't be any of the above

Cache Coherency Problem, WB $

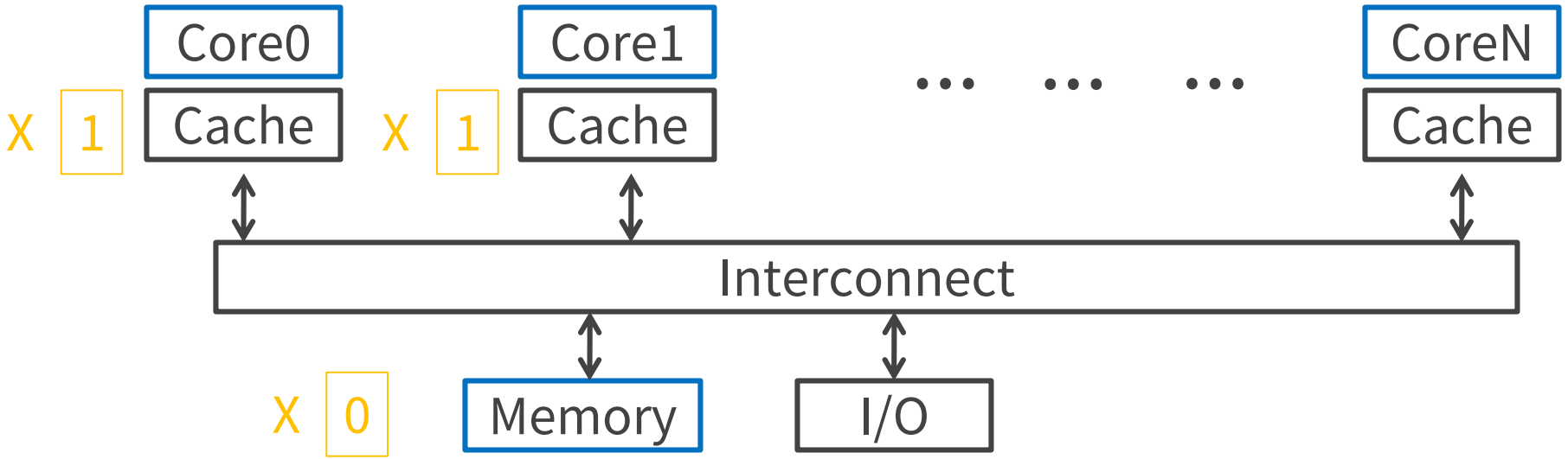
Thread A (on Core0)

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

Thread B (on Core1)

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

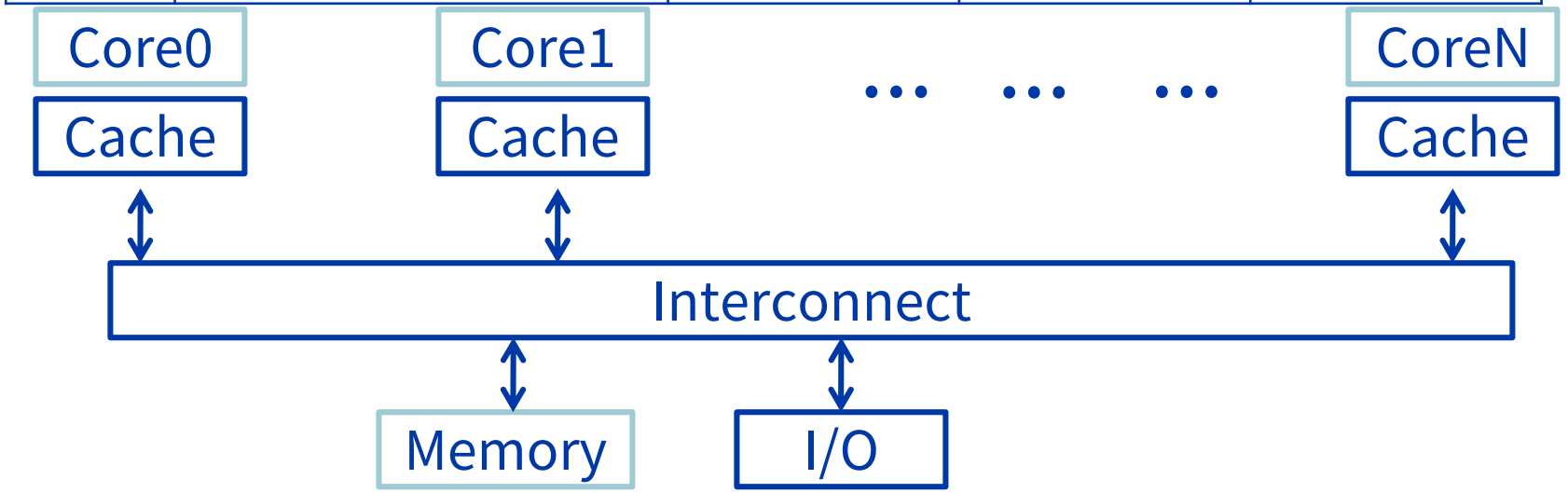
Problem!



Not just a problem for Write-Back Caches

Executing on a write-thru cache

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



Two issues here

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

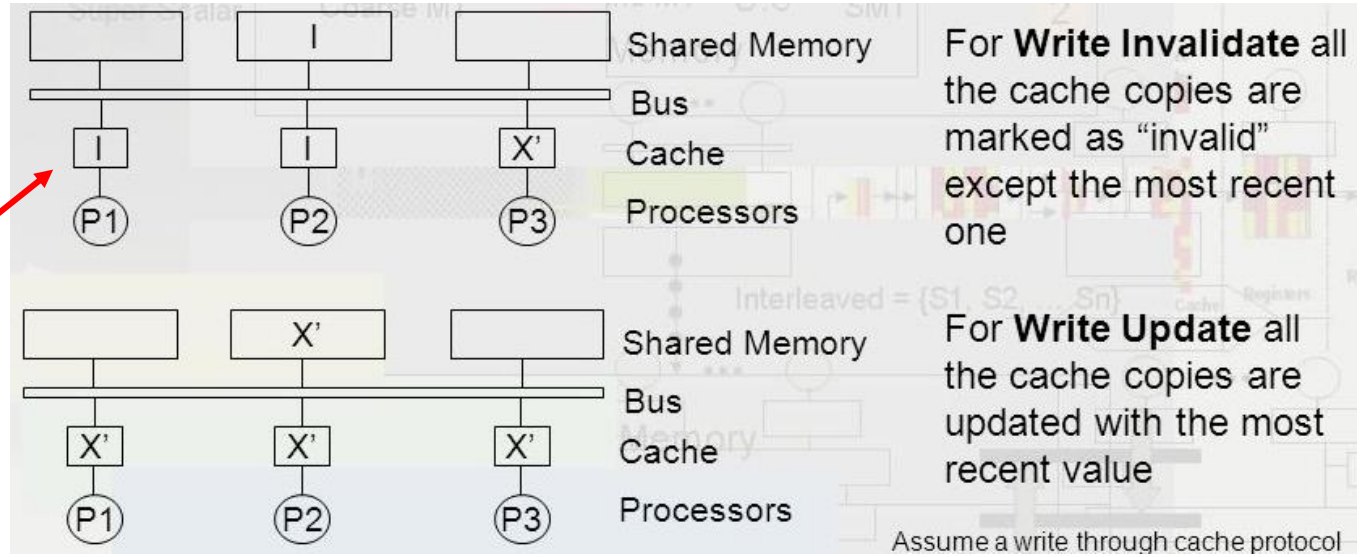
Approaches to Cache Coherence

- Software-based solutions
  - Mechanisms
    - Mark cache blocks/memory pages as cacheable/non-cacheable
    - Add “Flush” and “Invalidate” instructions
  - Could be done by compiler or runtime system
  - Difficult to get perfect
- Hardware solutions are far more common
  - Simple schemes rely on broadcast over a bus (Snooping Protocol)

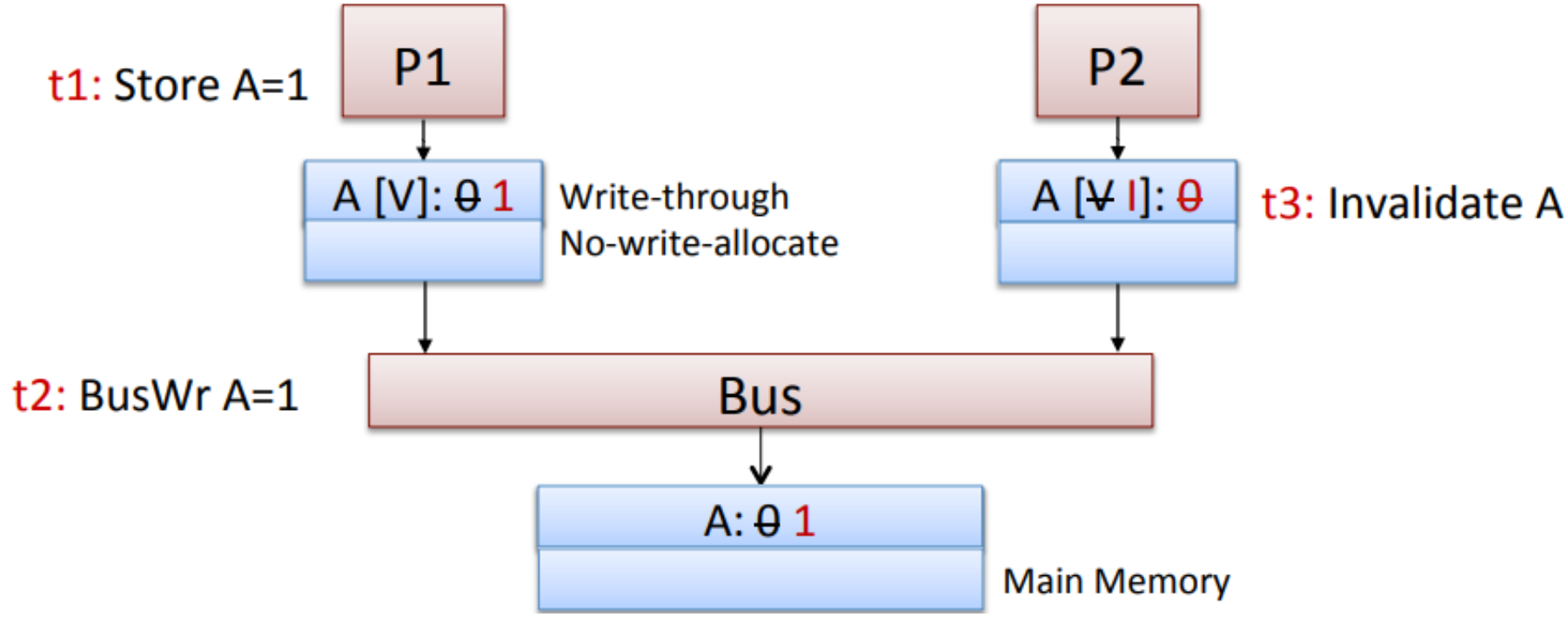
Snooping-based Protocol

- broadcasting coherence information to all processors over the shared interconnect
- Every time a cache miss occurred, the triggering cache communicated with all other caches!
- Two types
  - Write update
  - Write invalidate

Experience has shown that invalidate protocols use significantly less bandwidth.



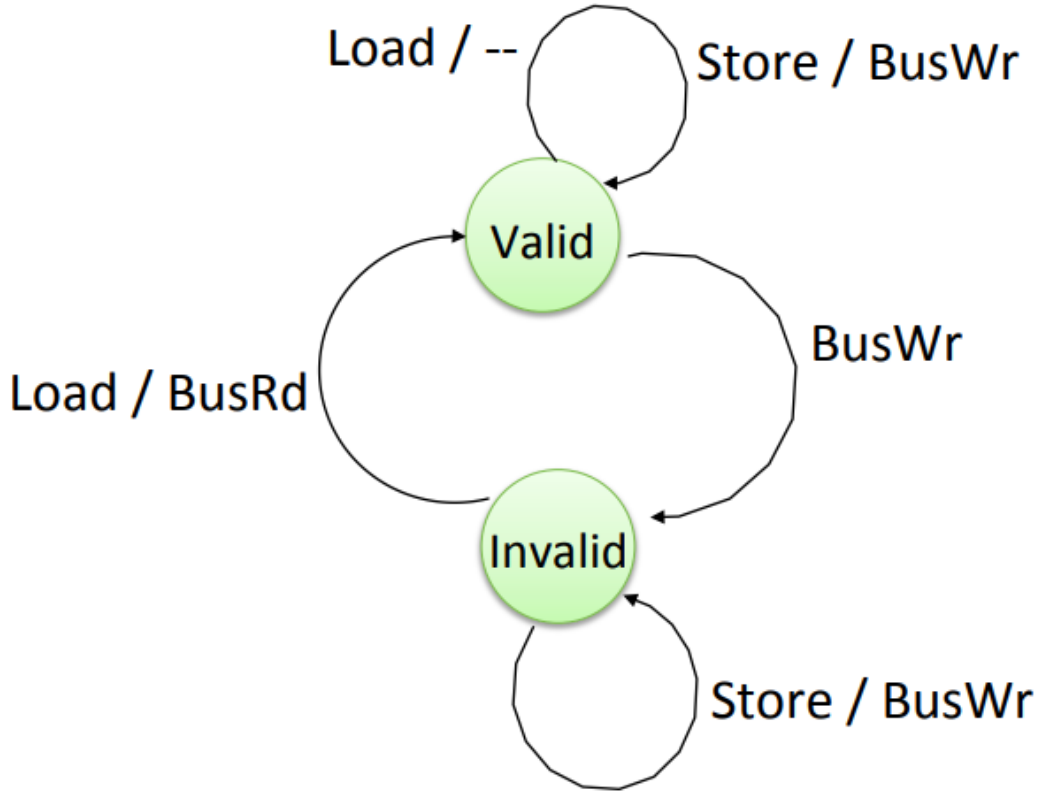
Simple Write-Through Scheme: Valid-Invalid Coherence



Valid-Invalid Coherence

- Allows multiple readers, but must write through to bus
  - Write-through, no-write-allocate cache
- All caches must monitor (aka “snoop”) all bus traffic
  - simple state machine for each cache frame

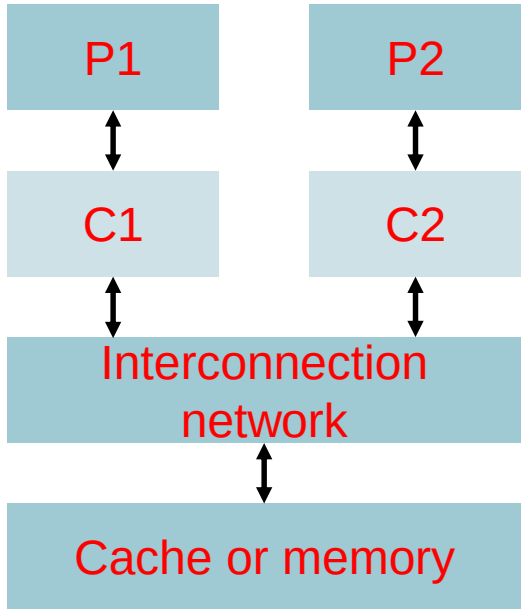
Valid-Invalid (VI) Snooping Protocol



Actions:
Ld, St, BusRd, BusWr

Write-through, no-write-allocate cache

1 bit of storage overhead per cache frame

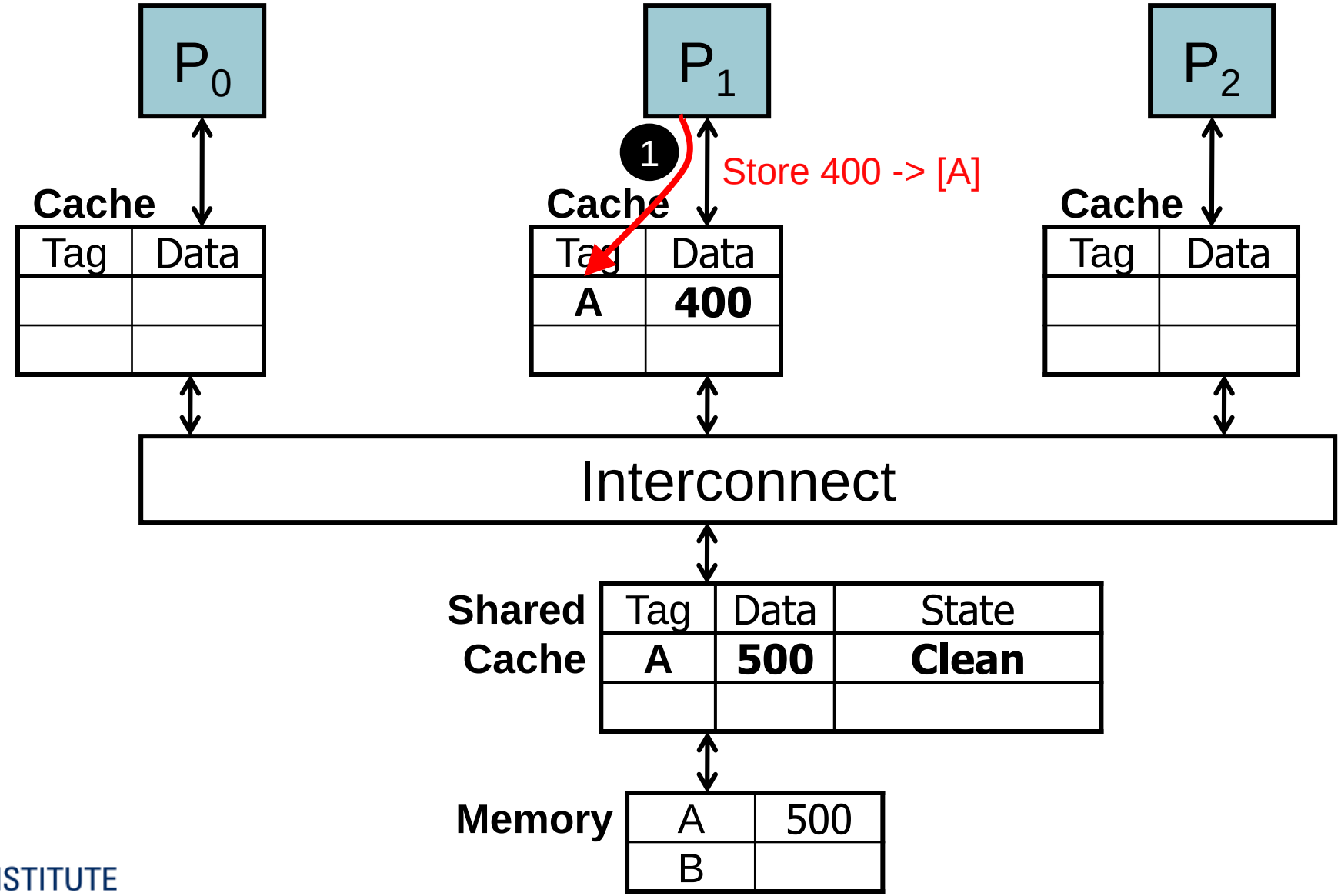


- If *you* load/store a block: transition to **V**
- If anyone *else* wants to read/write block:
  - Give it up: transition to **I** state
  - Write-back if your own copy is dirty

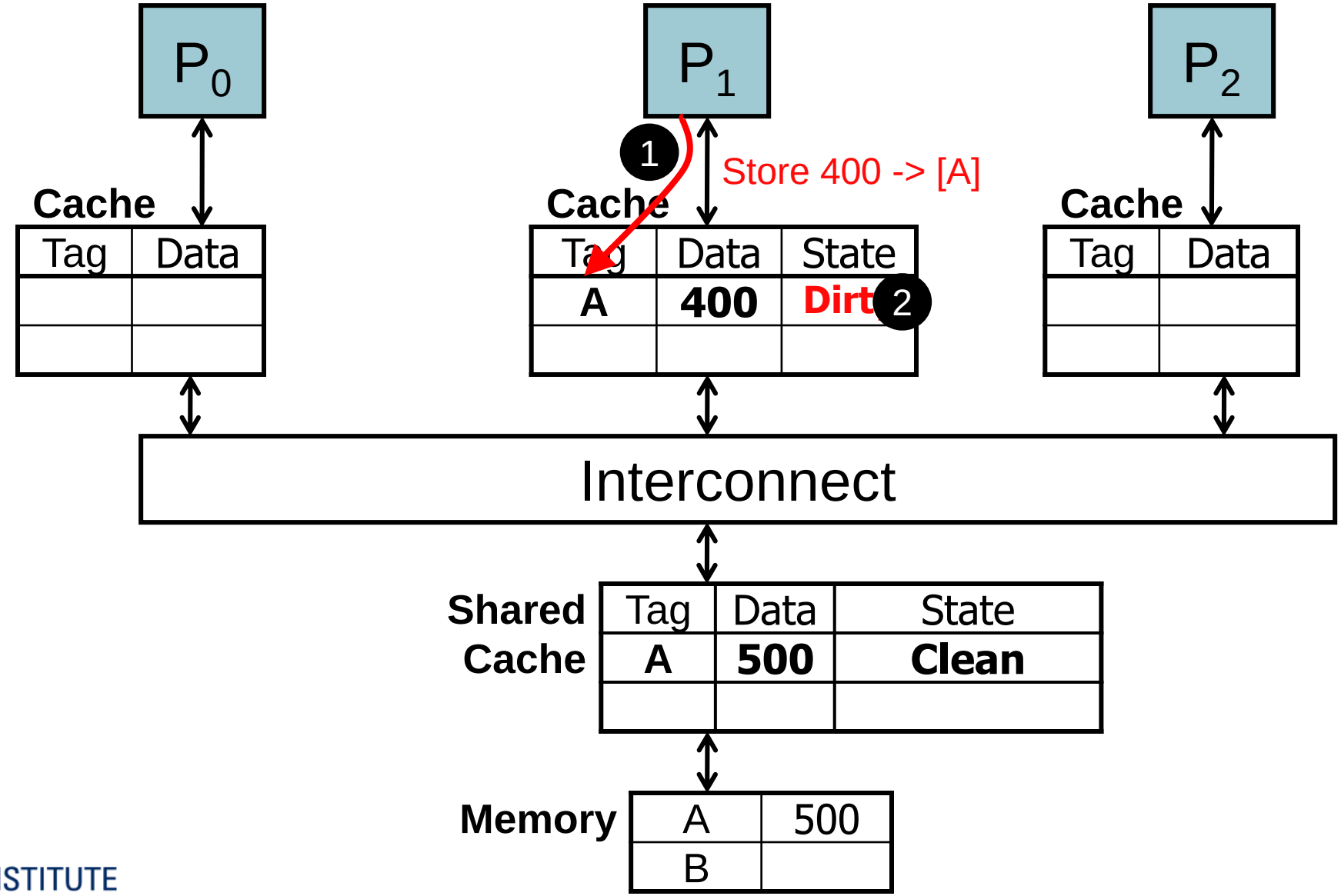
VI Protocol

- So far: Option #1 “Snooping”: broadcast to all, whoever has it responds
- Option #2: “Directory”: track sharers with separate structure

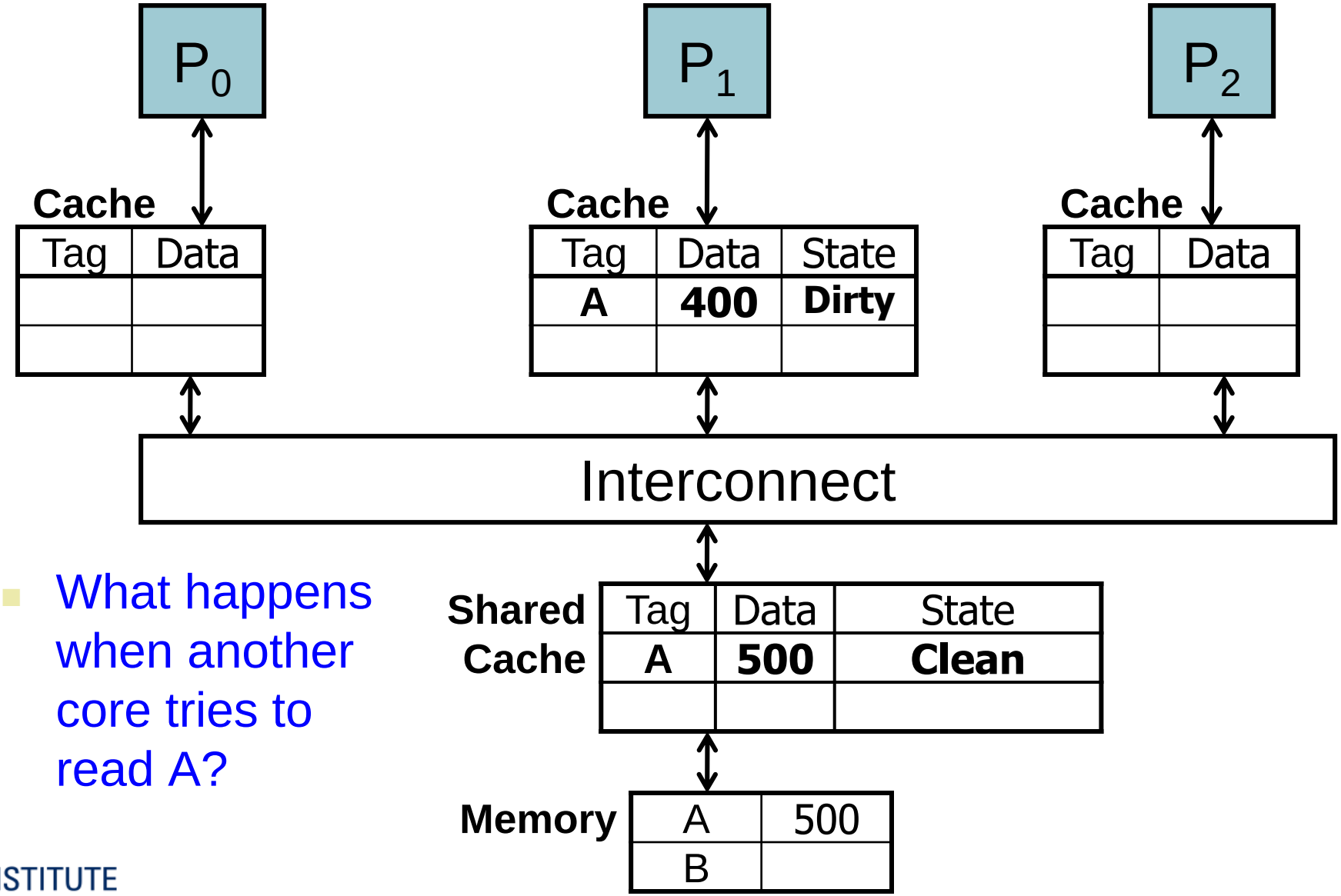
VI Protocol Example



VI Protocol Example

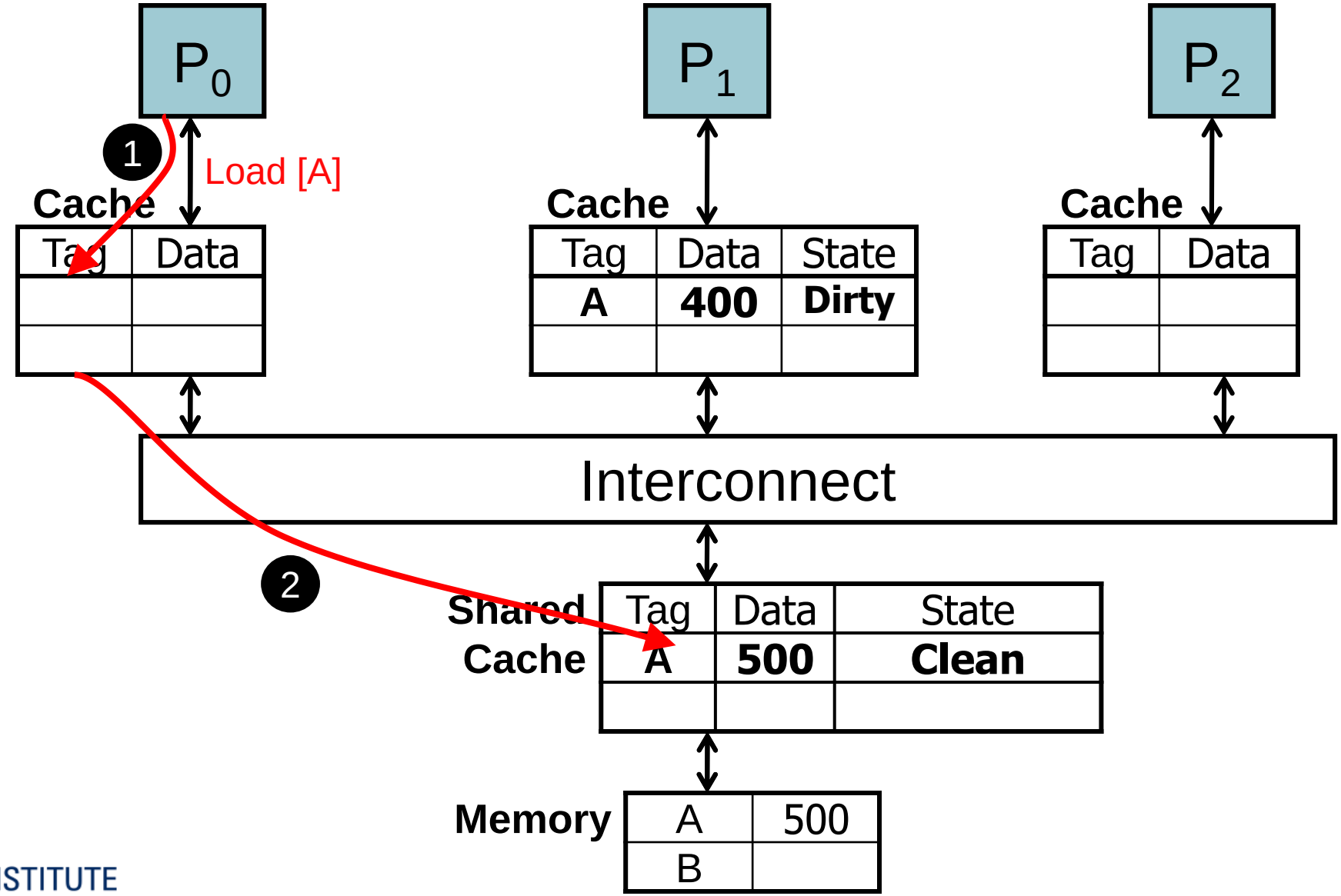


VI Protocol Example

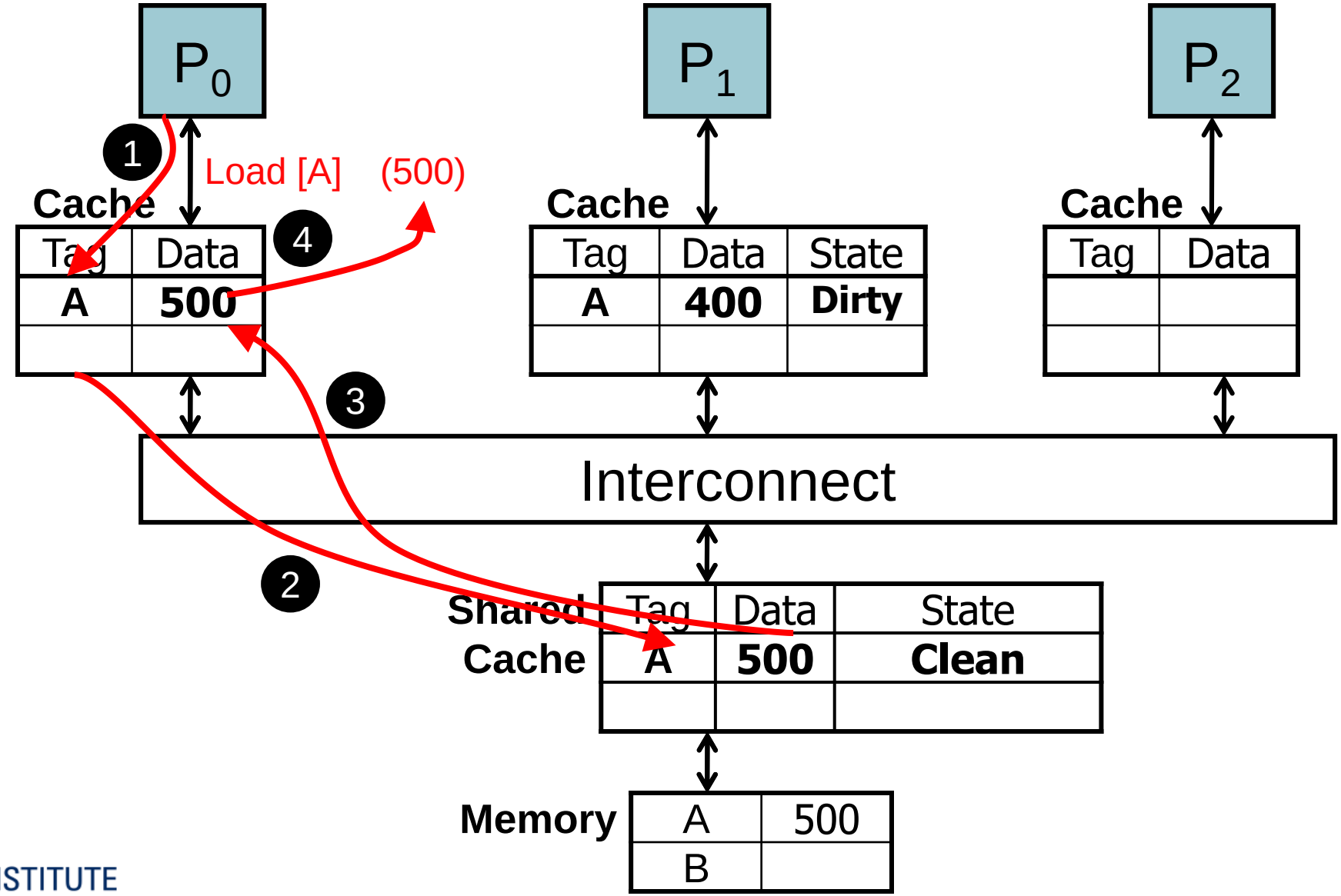


- What happens when another core tries to read A?

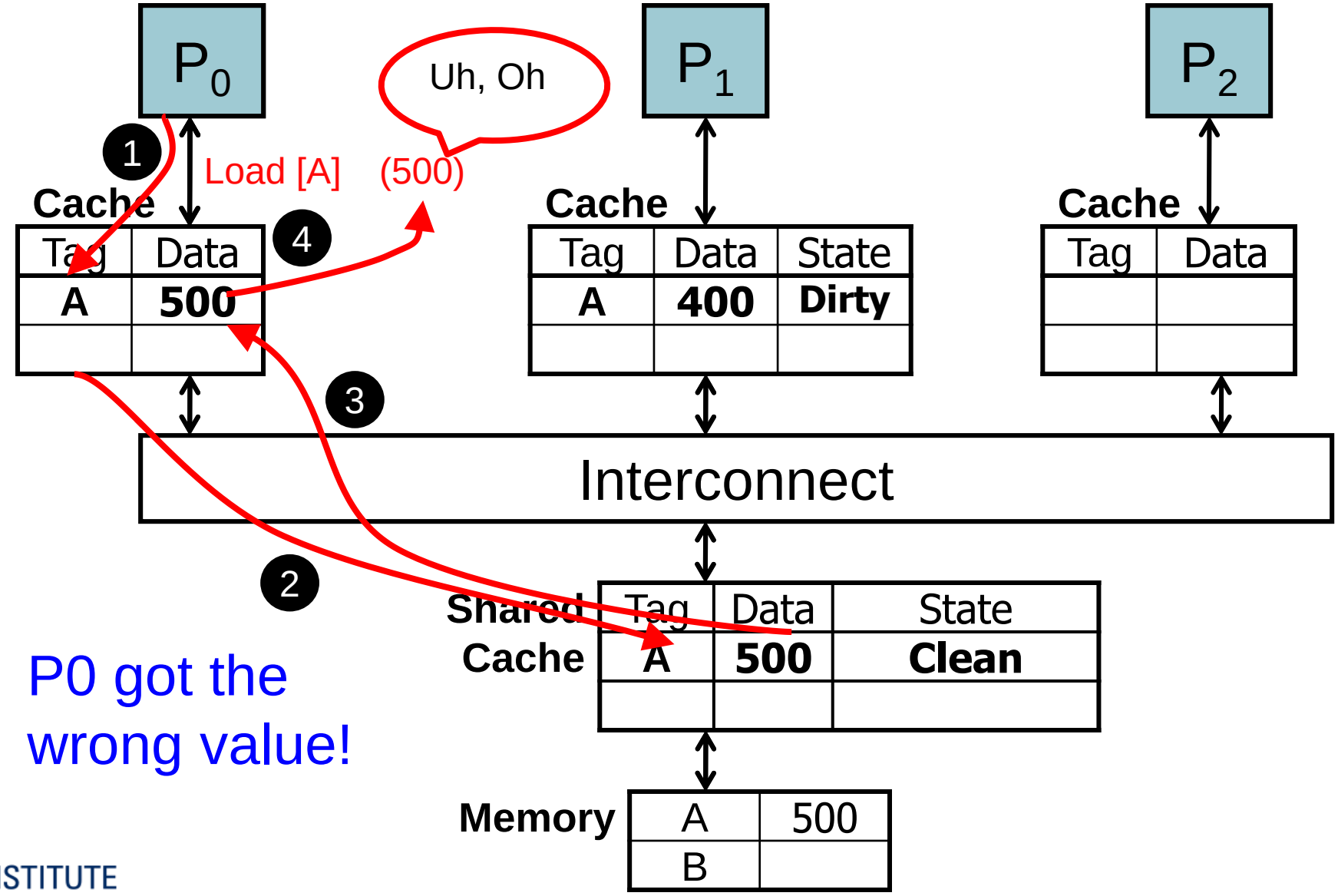
VI Protocol Example



VI Protocol Example

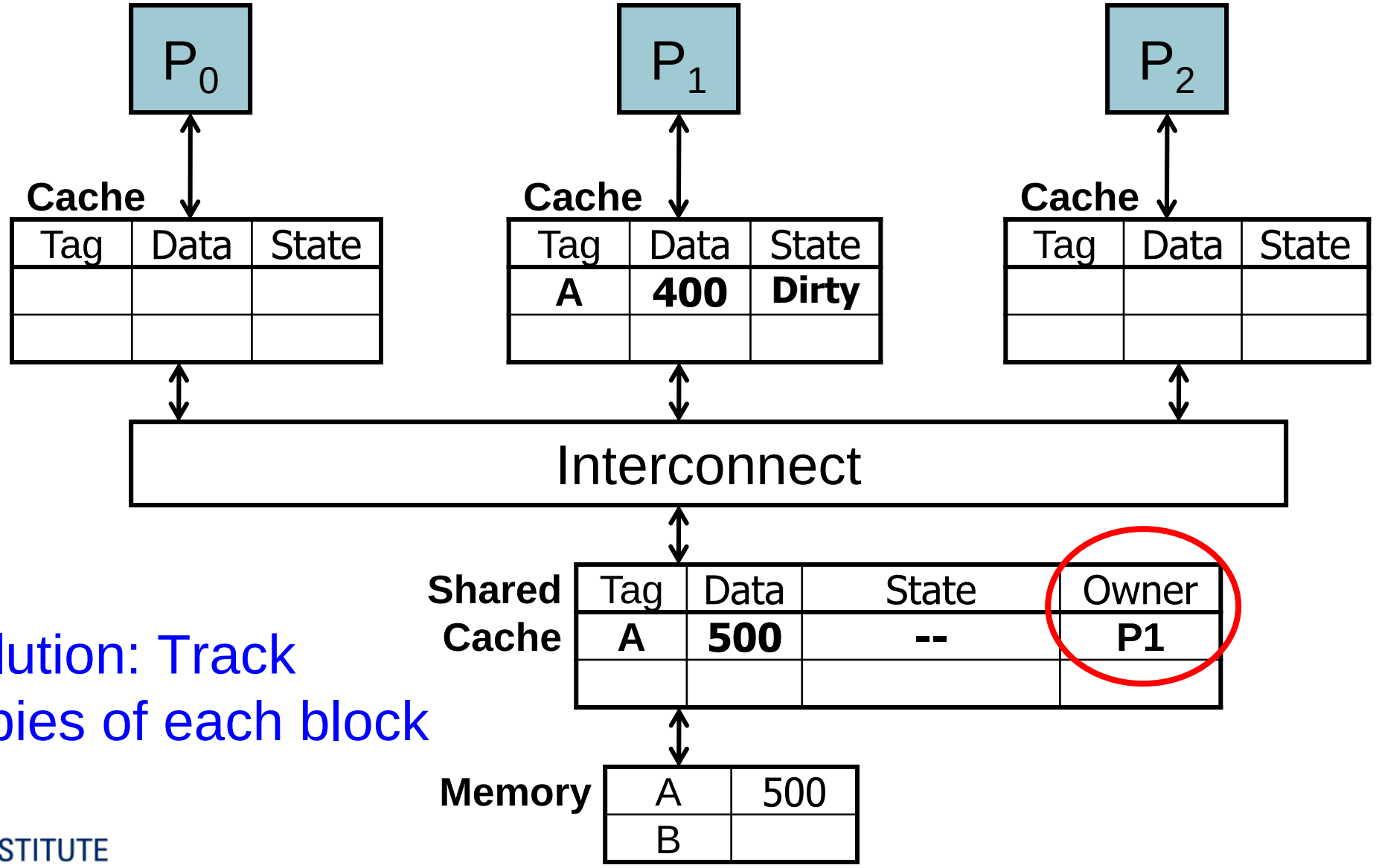


VI Protocol Example



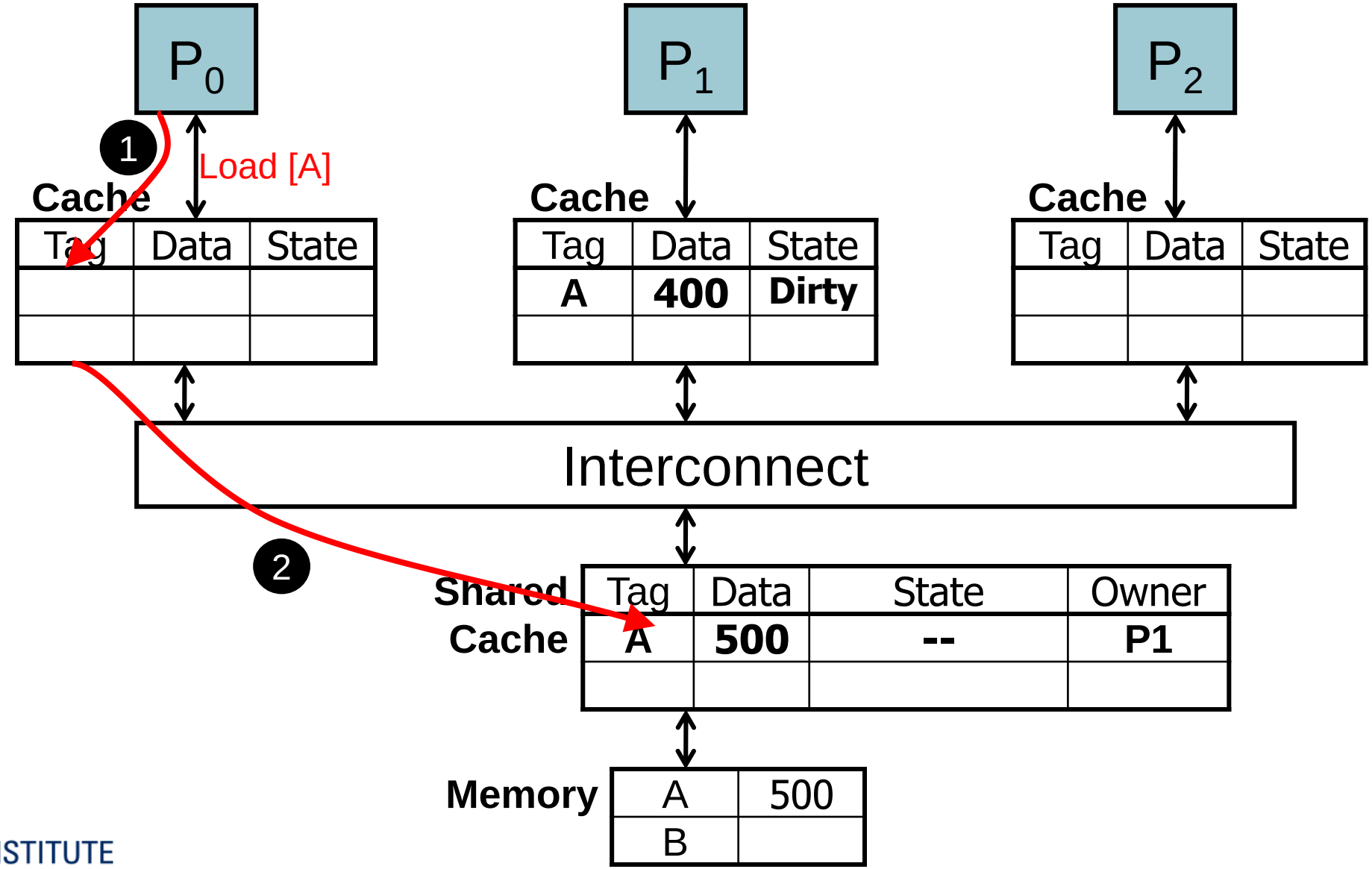
- P0 got the wrong value!

Rewind: Fix Problem by Tracking Sharers

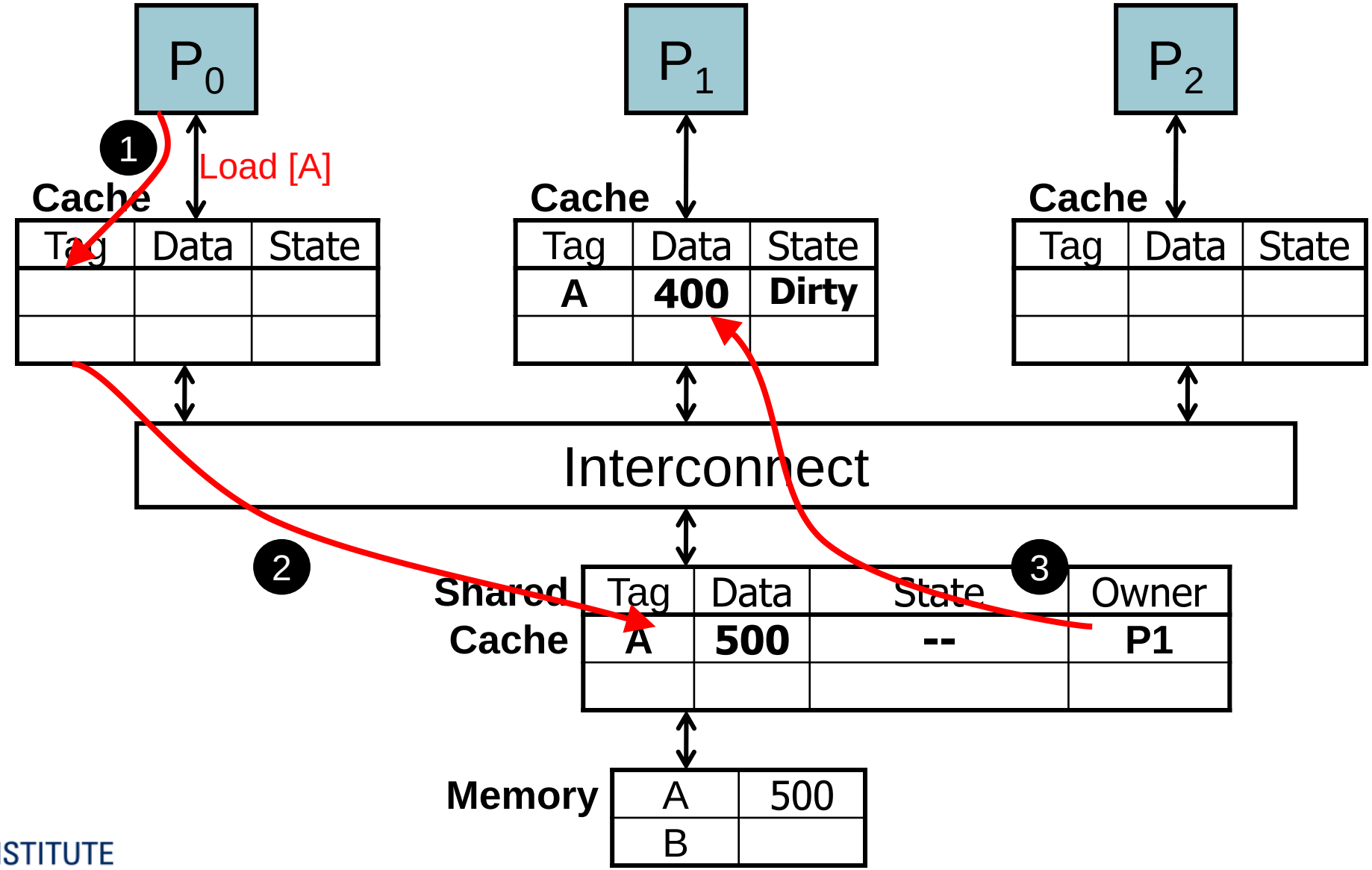


- Solution: Track copies of each block

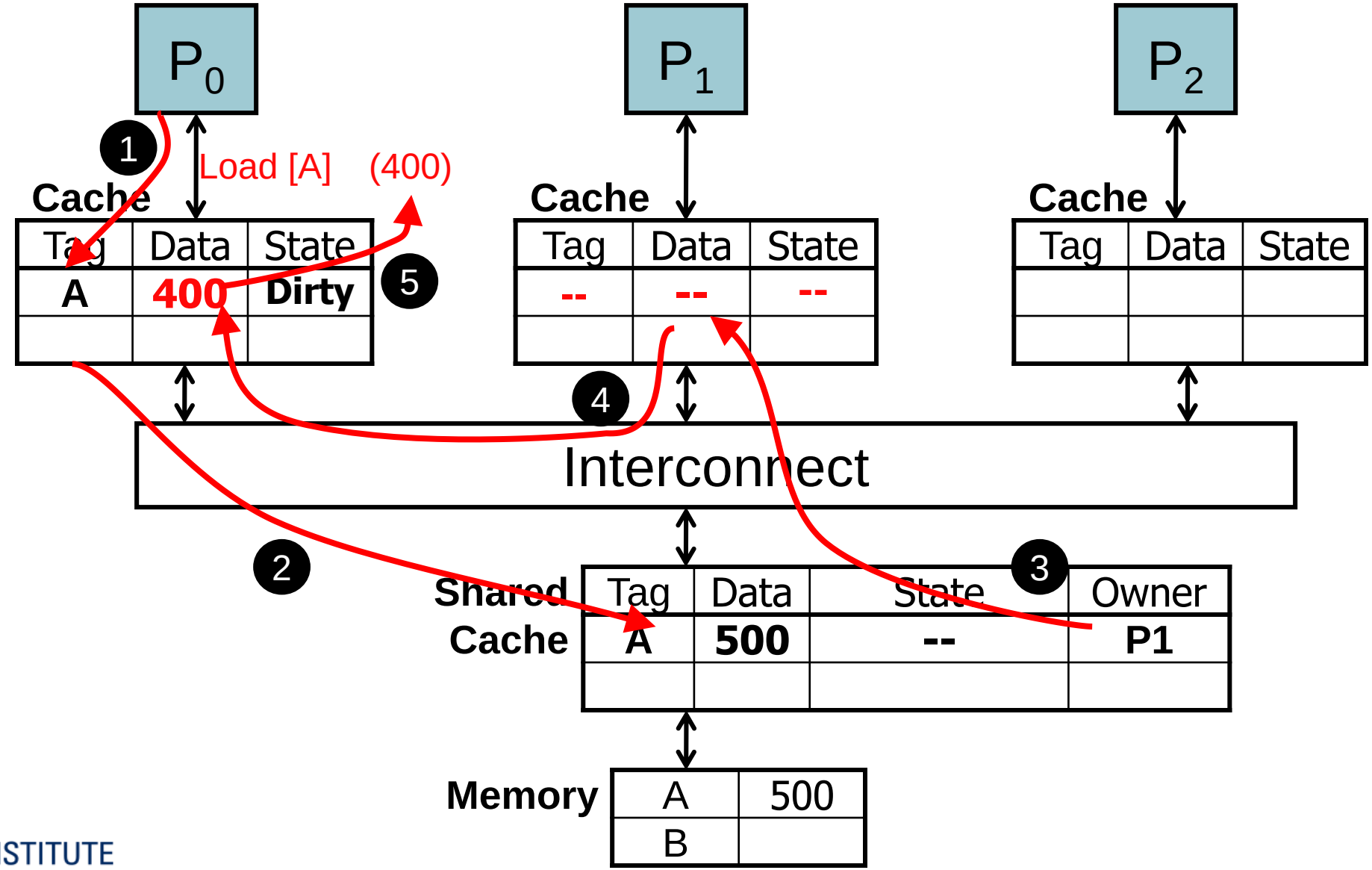
Use Tracking Information to “Invalidate”



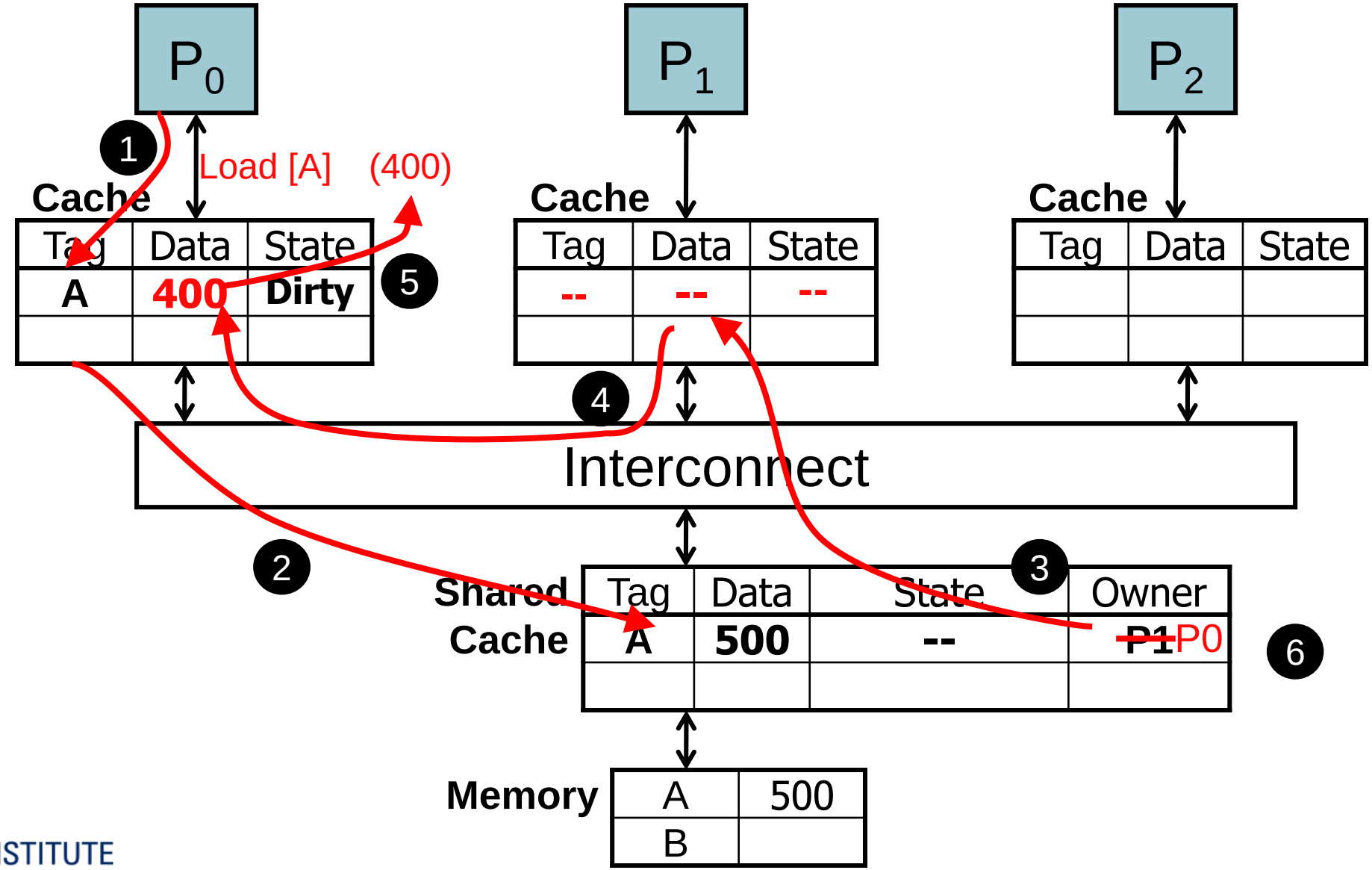
Use Tracking Information to "Invalidate"



Use Tracking Information to "Invalidate"



Use Tracking Information to "Invalidate"



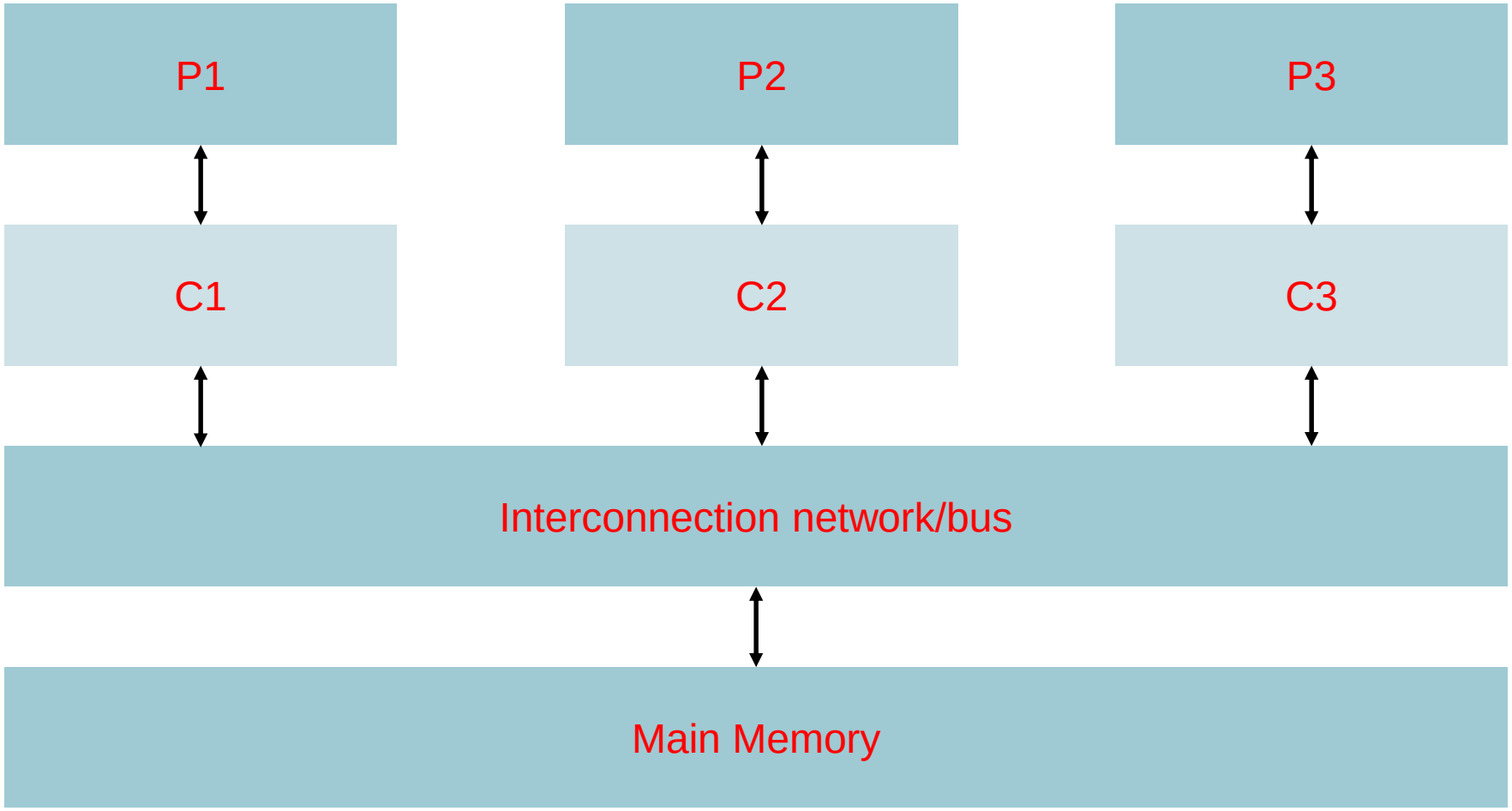
"Valid/Invalid" Cache Coherence Summary

- 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**": track sharers with separate structure
- **Problem**: multiple copies can't exist, even if read-only
  - Consider mostly-read data structures, instructions, etc.

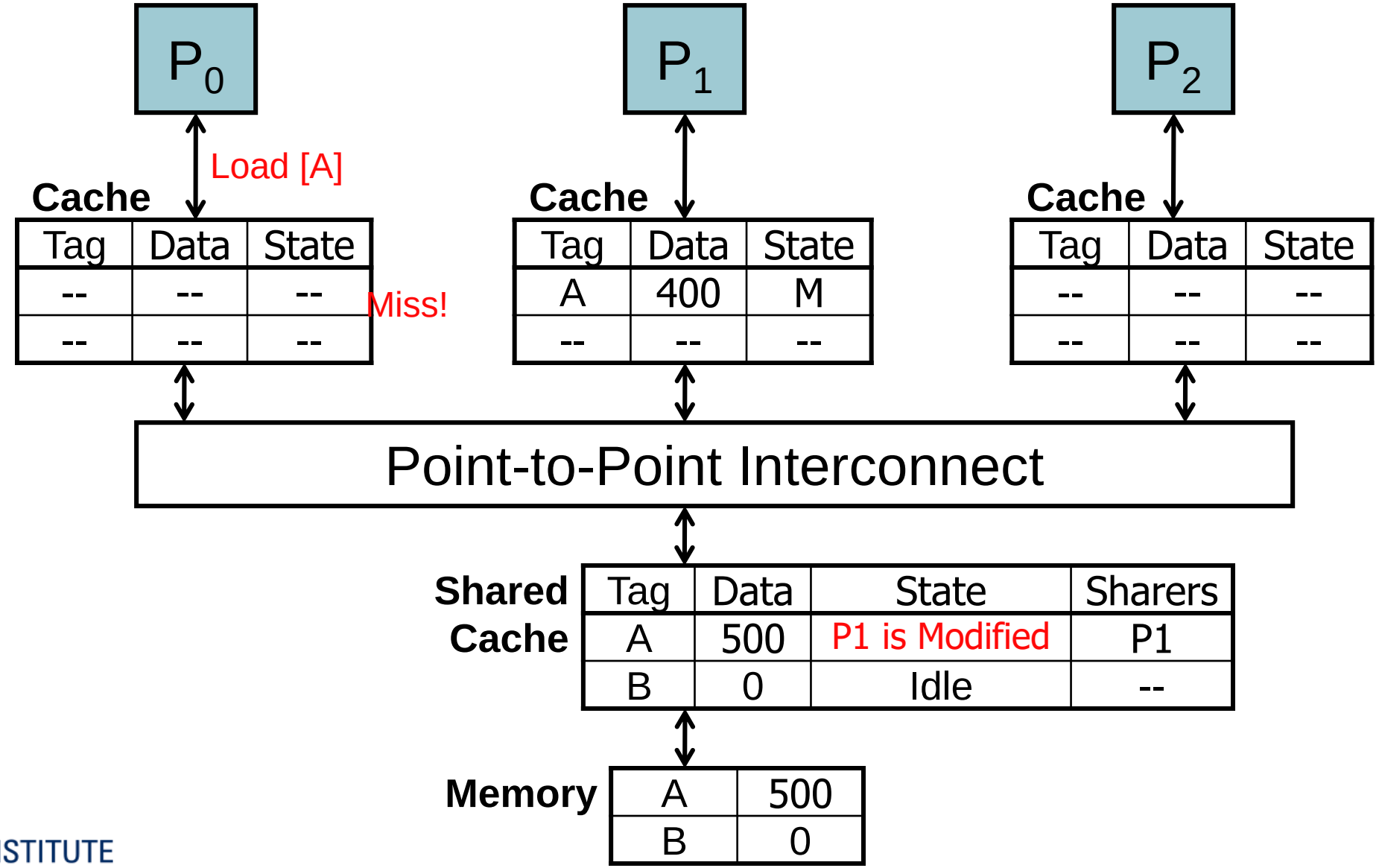
Modified-Shared-Invalid (MSI) Protocol

- Allows for many readers…
- Three states tracked per-block at each cache
  - **M**odified – cache has the only copy; writable; dirty (read/write permission)
    - Dirty == memory is out of date
  - **S**hared – cache has a read-only copy; clean (read-only permission)
    - Clean == memory is up to date
  - **I**nvalid – cache does not have a copy (blocked, no permission)
- Three processor actions
  - Load, Store, Evict
- Five bus messages
  - BusRd, BusRdX, BusInv, BusWB, BusReply
  - Could combine some of these

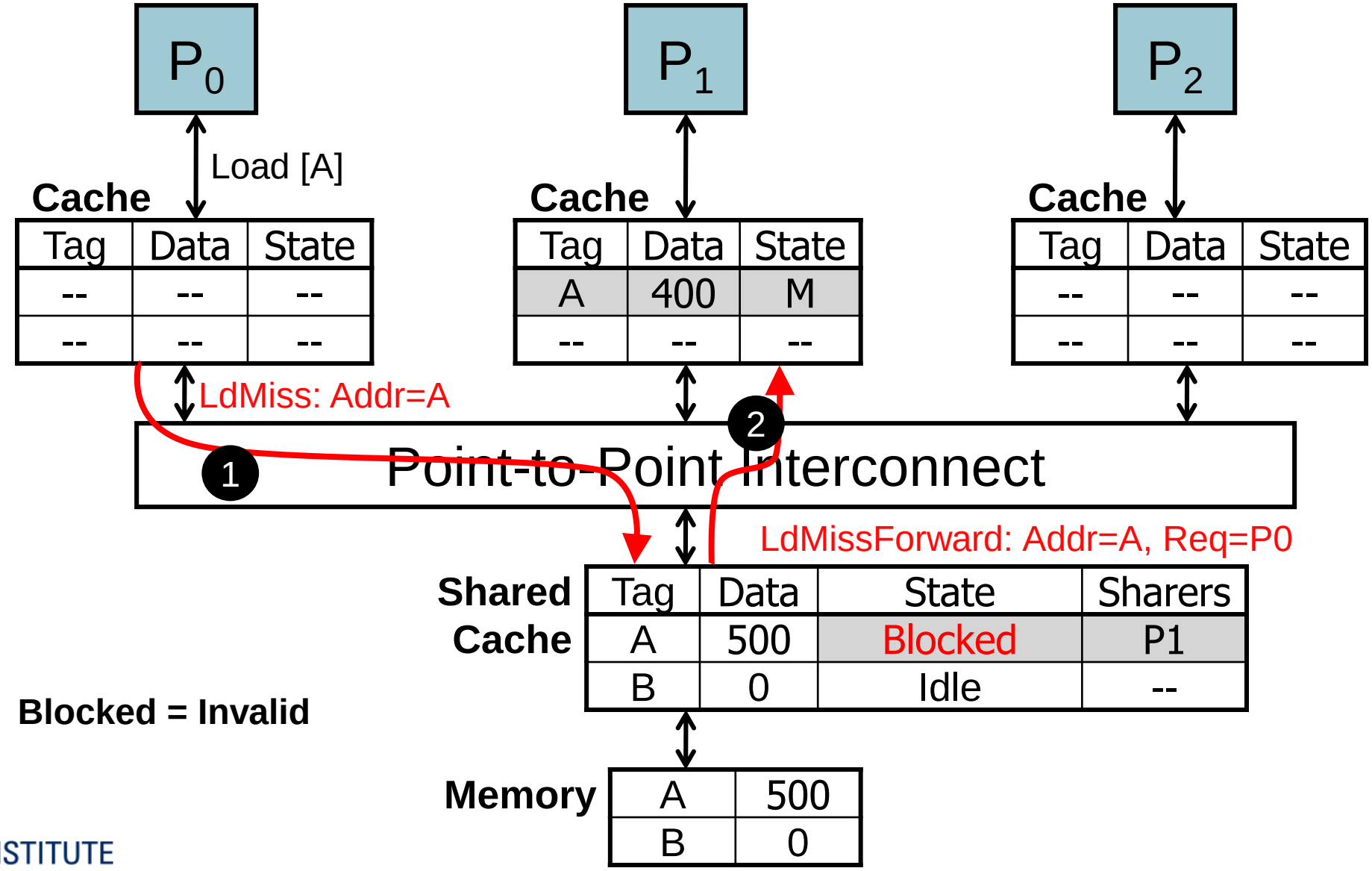
MSI (Snooping Based)



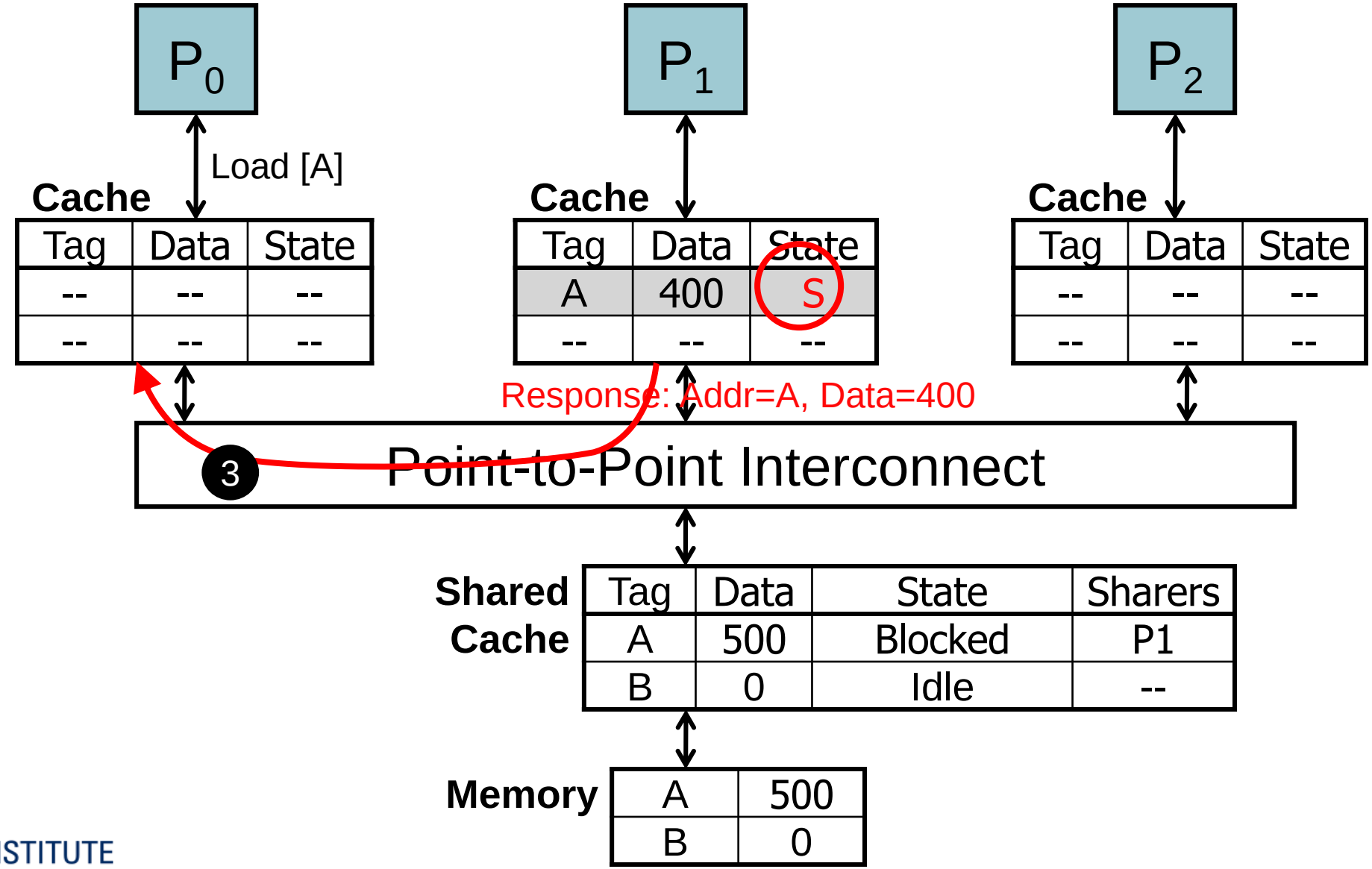
MSI Coherence Example: Step #1



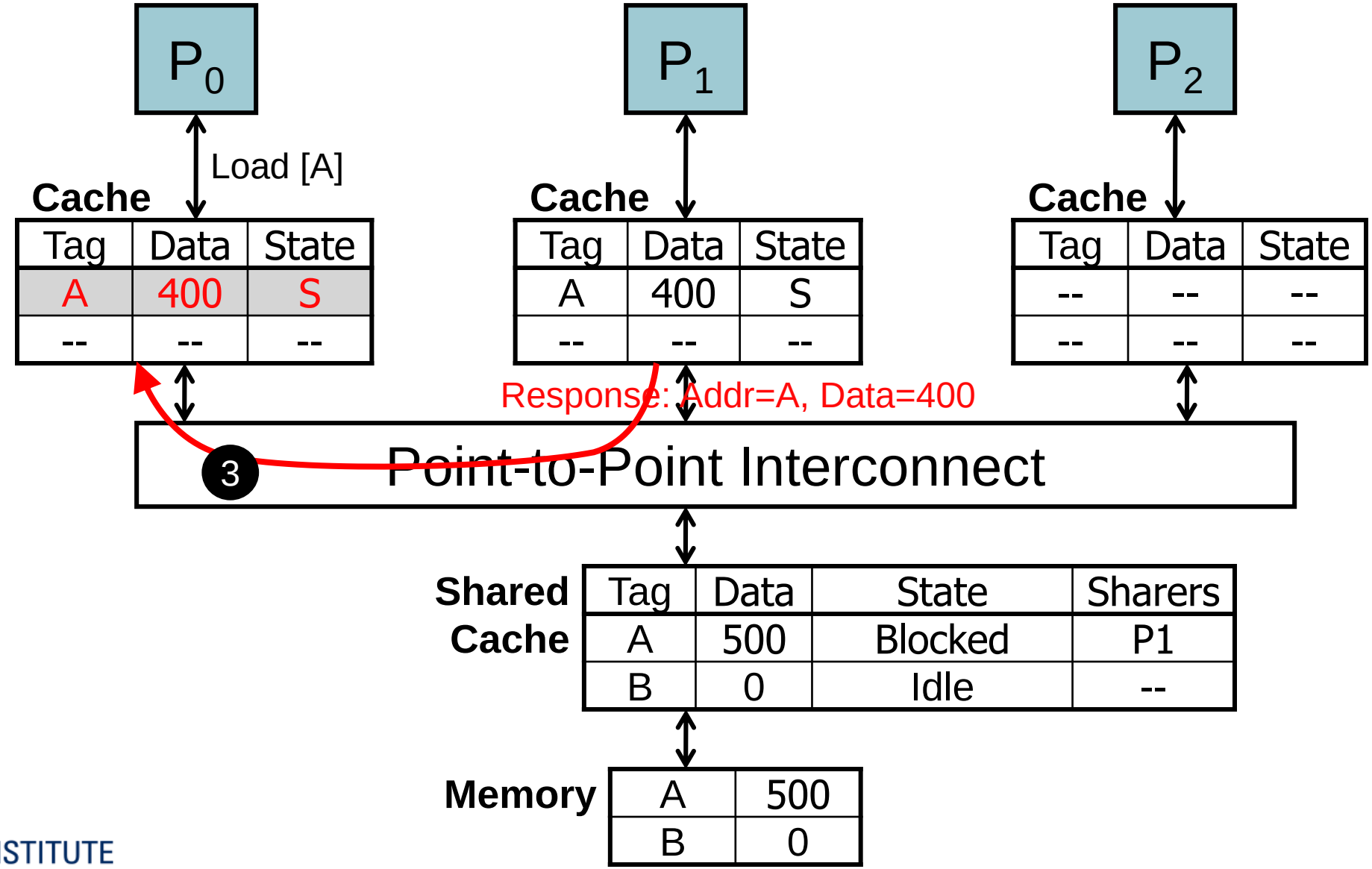
MSI Coherence Example: Step #2



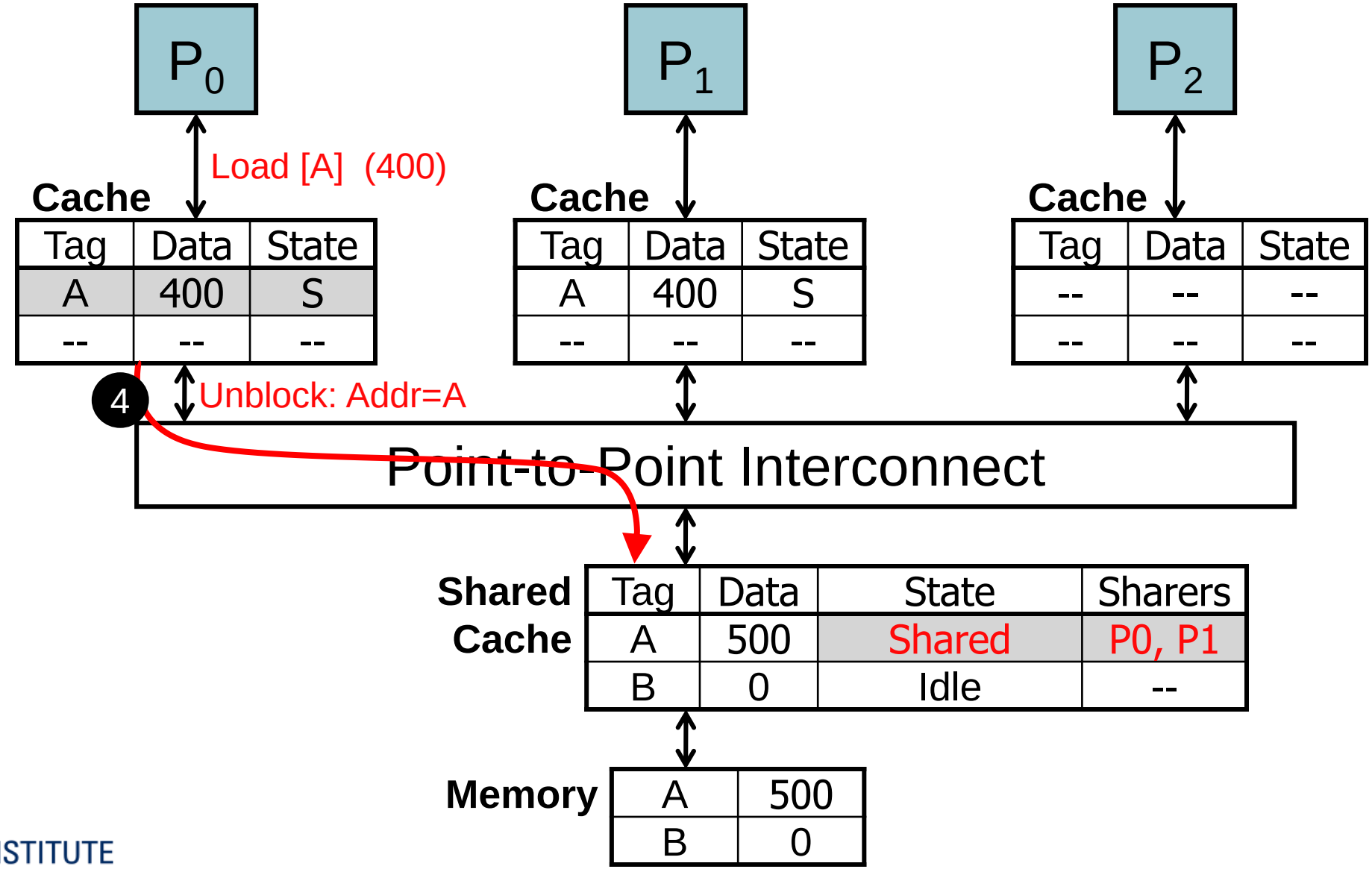
MSI Coherence Example: Step #3



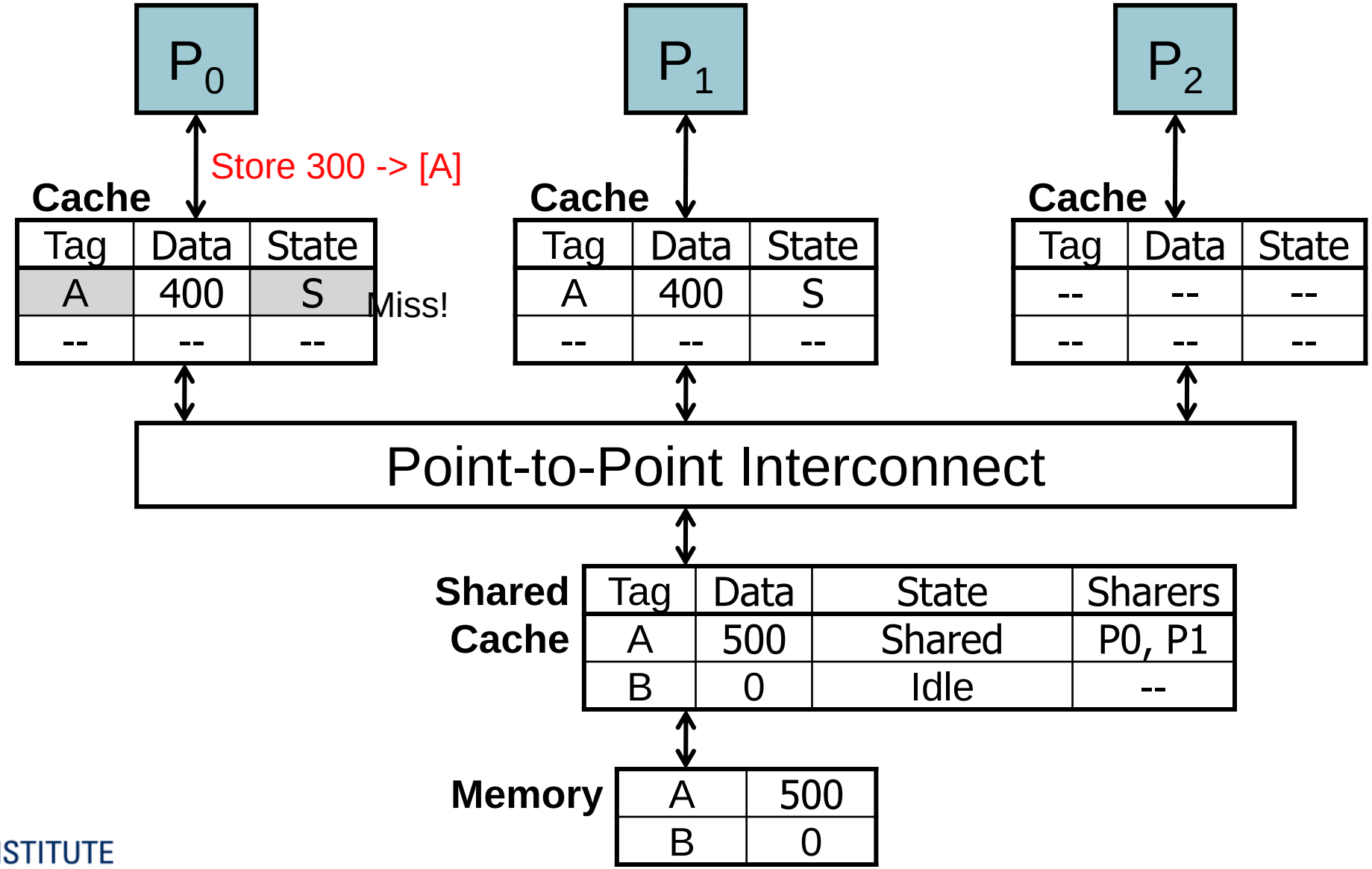
MSI Coherence Example: Step #4



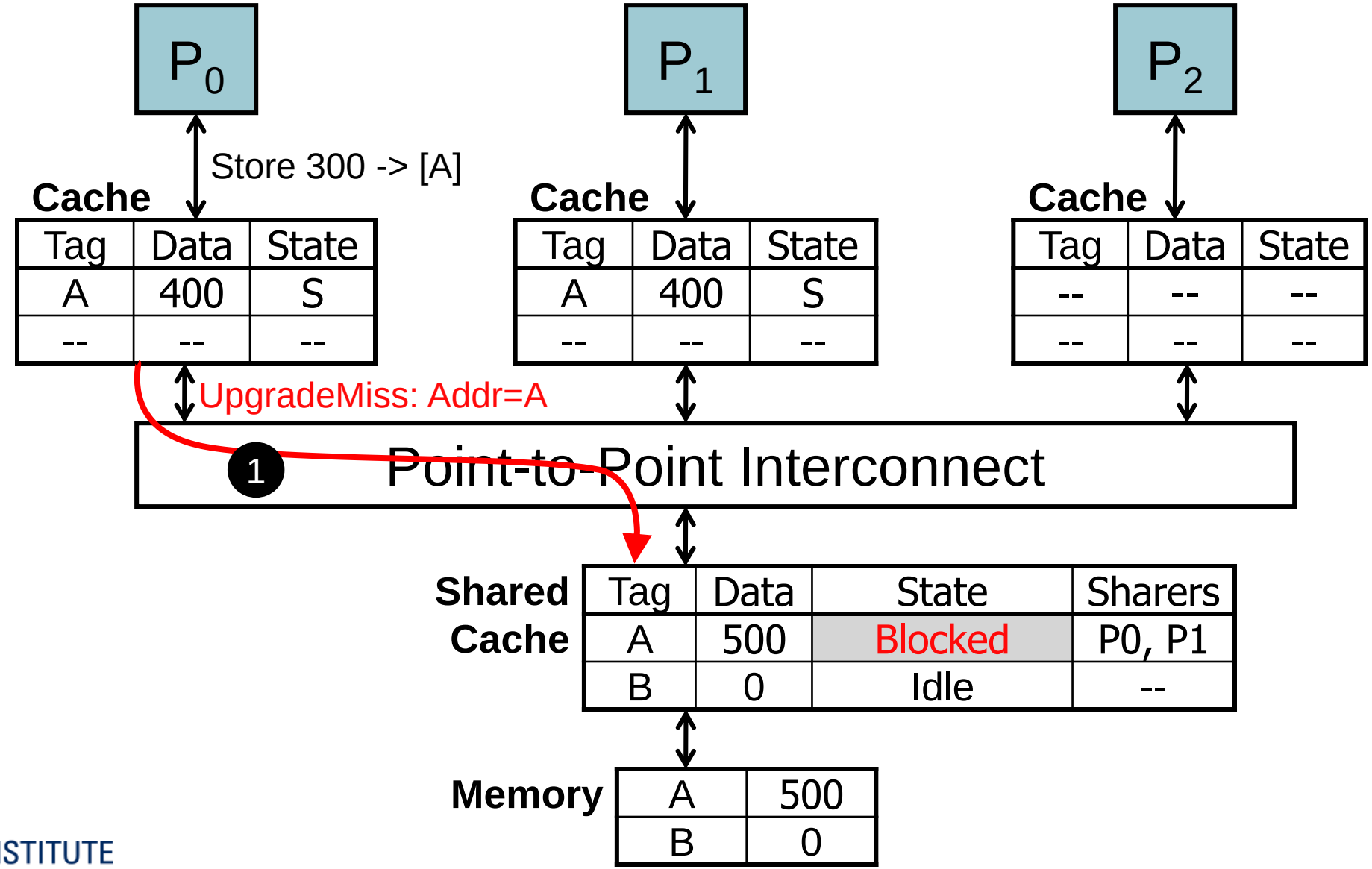
MSI Coherence Example: Step #5



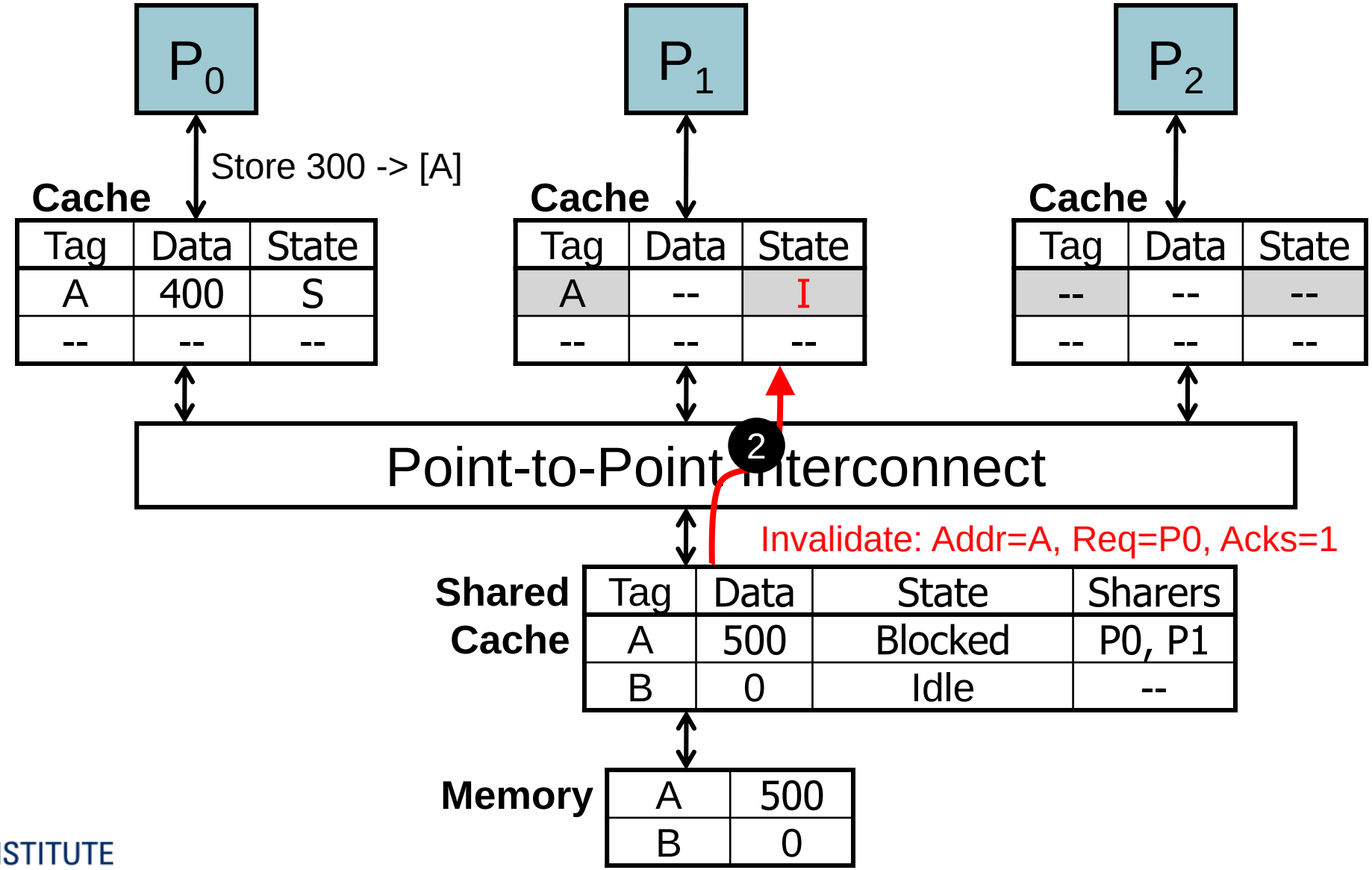
MSI Coherence Example: Step #6



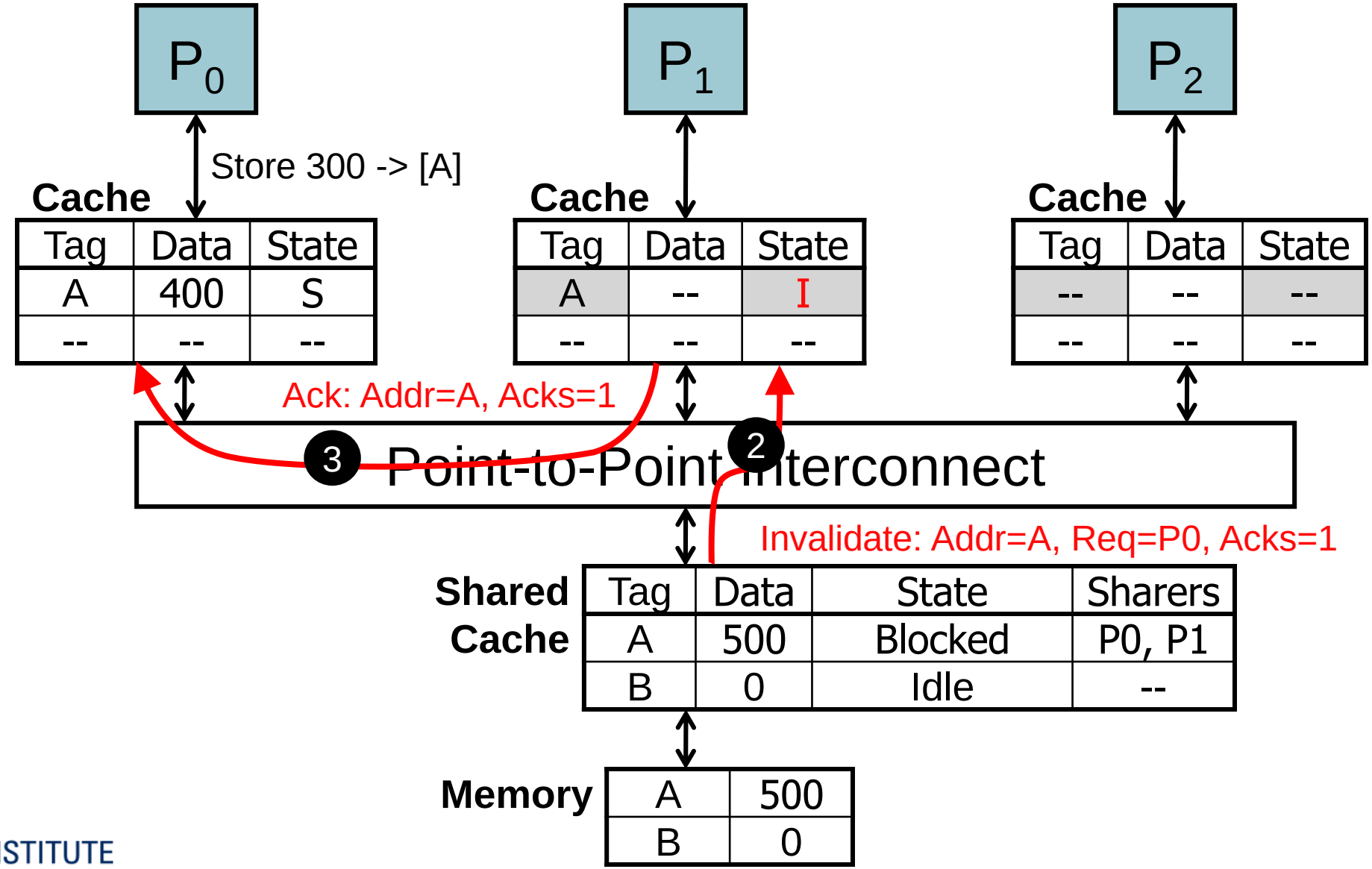
MSI Coherence Example: Step #7



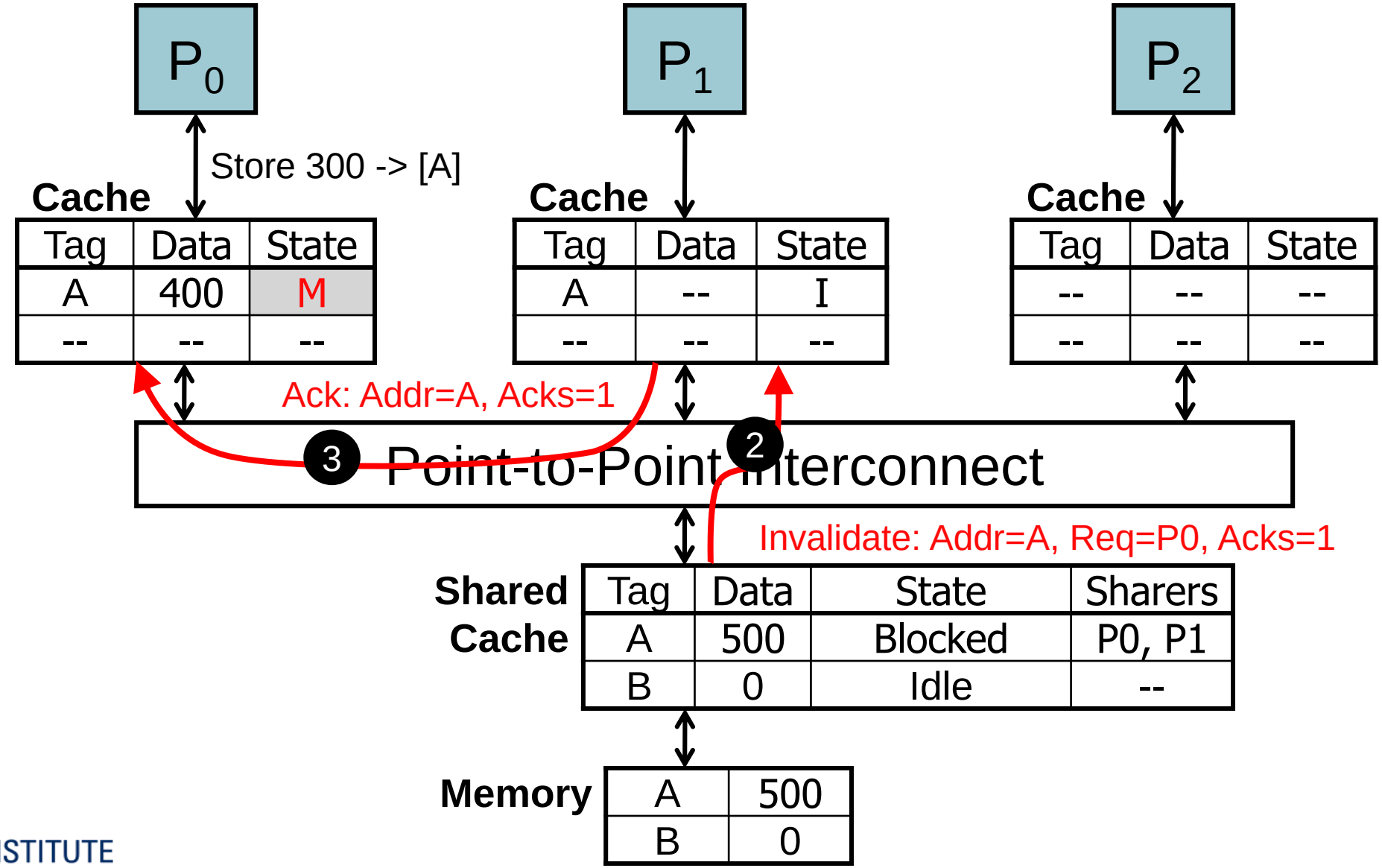
MSI Coherence Example: Step #8



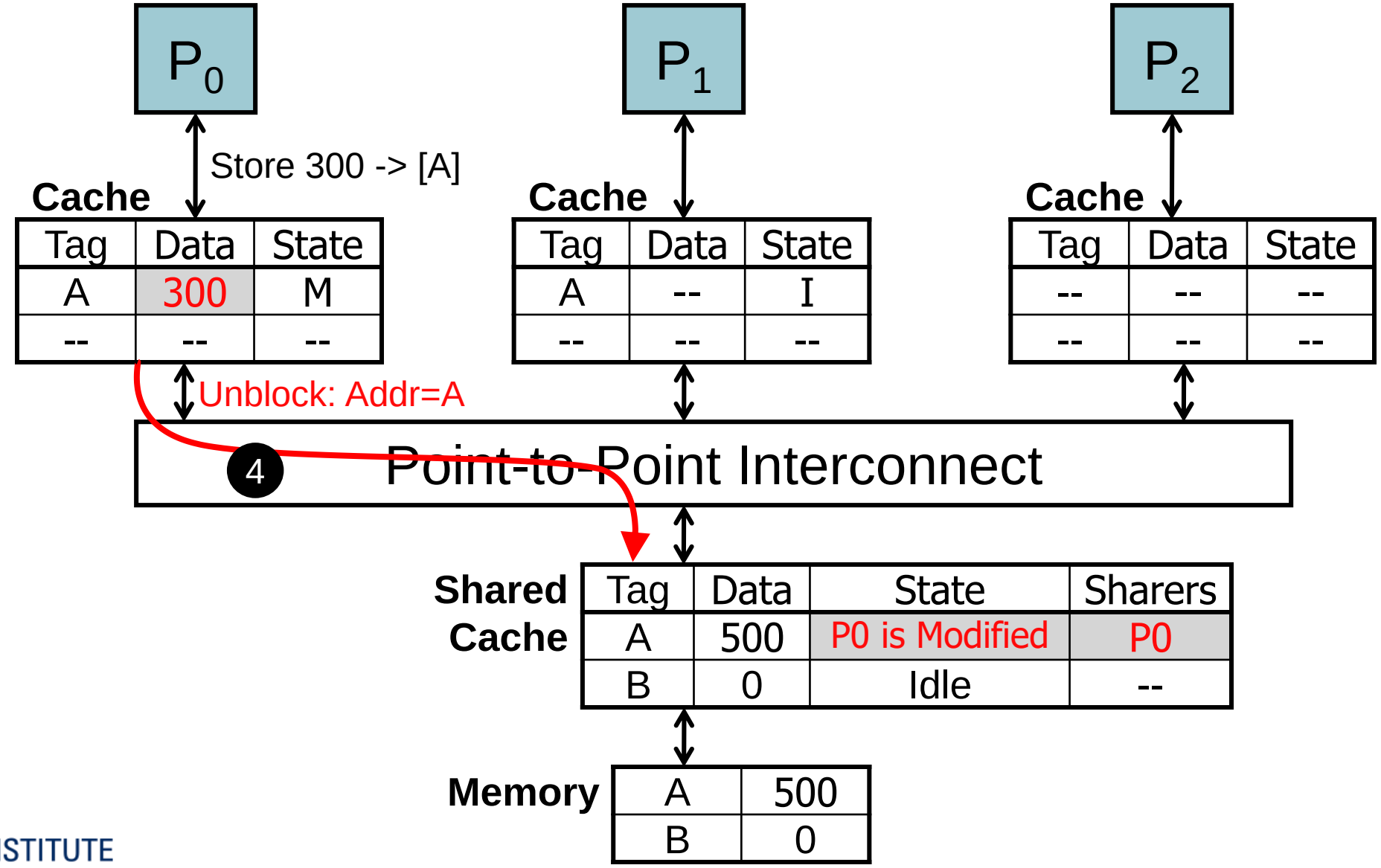
MSI Coherence Example: Step #9



MSI Coherence Example: Step #10

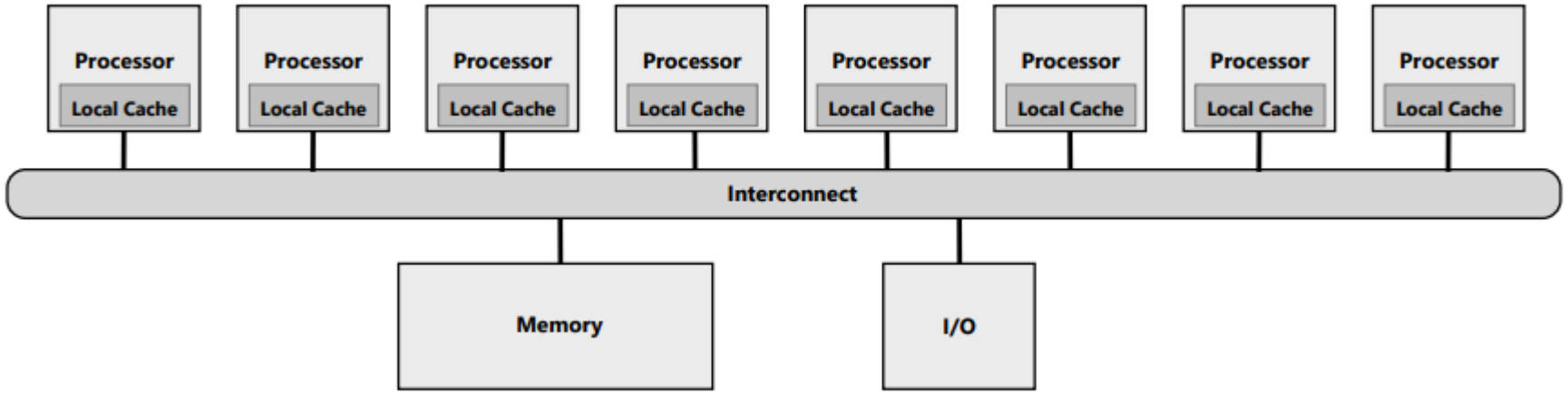


MSI Coherence Example: Step #11

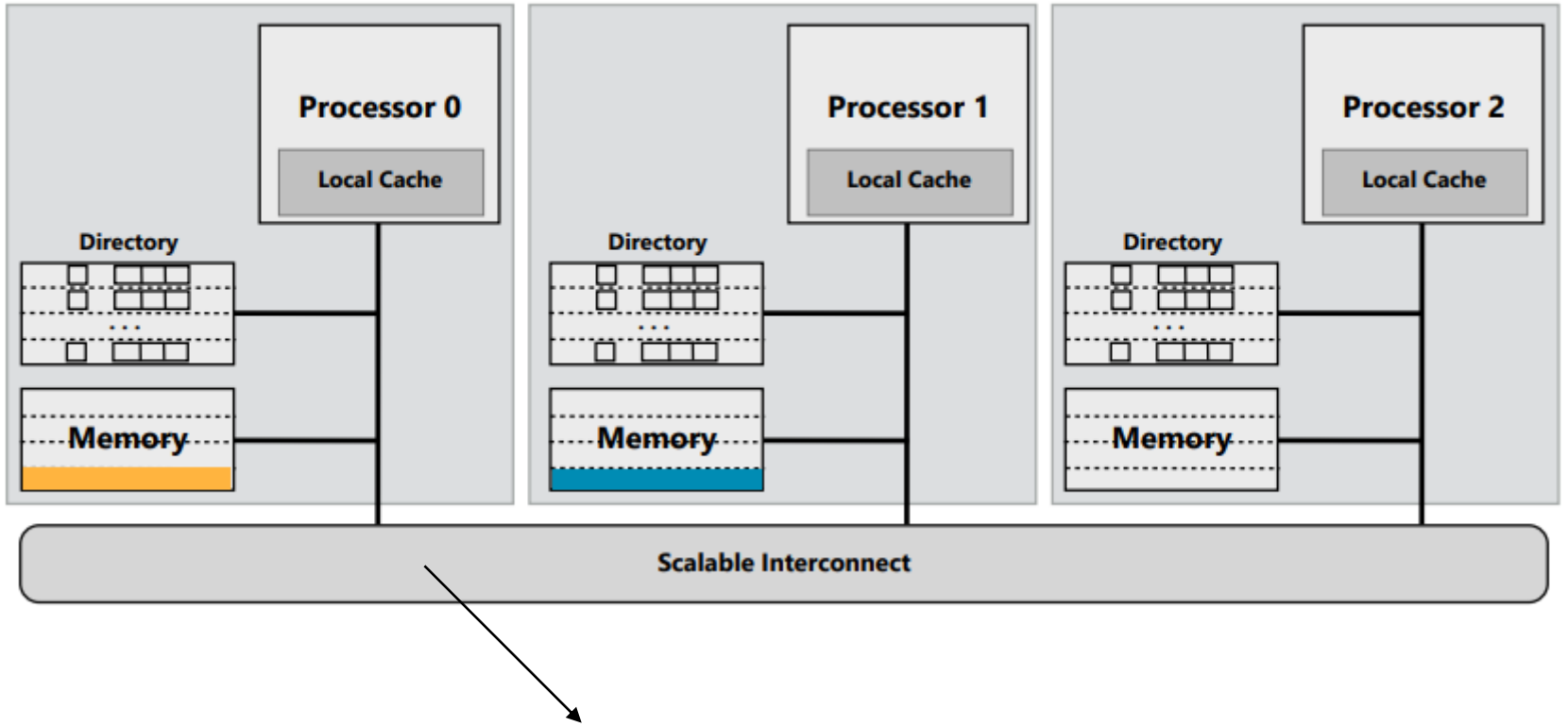


Snooping based protocol

- V-I, MSI, MESI (Read Textbook Page 388)
- Every time a cache miss occurred, the triggering cache communicated with all other caches
- Communication limited to one message at a time
- Scalability is a huge challenge



Directory Based Cache Coherence



Not a bus! A lot of point-to-point connection, can be a ring

Read Ch. 5.4!

Where are we Heading?

- T7: More multicore

Acknowledgement

Slides in this topic are inspired in part by material developed and copyright by:

- ARM Courseware
- Nvidia Courseware
- Prof. Onur Mutlu @ ETH
- Prof. Joe Devietti @ Upenn, CIS 571
- Prof. Hakim Weatherspoon @ Cornell, CS 3410
- Prof. Krste Asanovic @ UCB, CS252
- Xinfei Guo @ JI, VE370 2021 SU

Action Items

- Lab #6 due by 11:59pm July 15th, 2022 (Beijing Time)
- Final project
- Reading Materials
  - Ch. 5.1, 5.2, 5.6