

CS 152 Computer Architecture and Engineering

CS252 Graduate Computer Architecture

Lecture 10 Multi-Processors& Cache Coherence

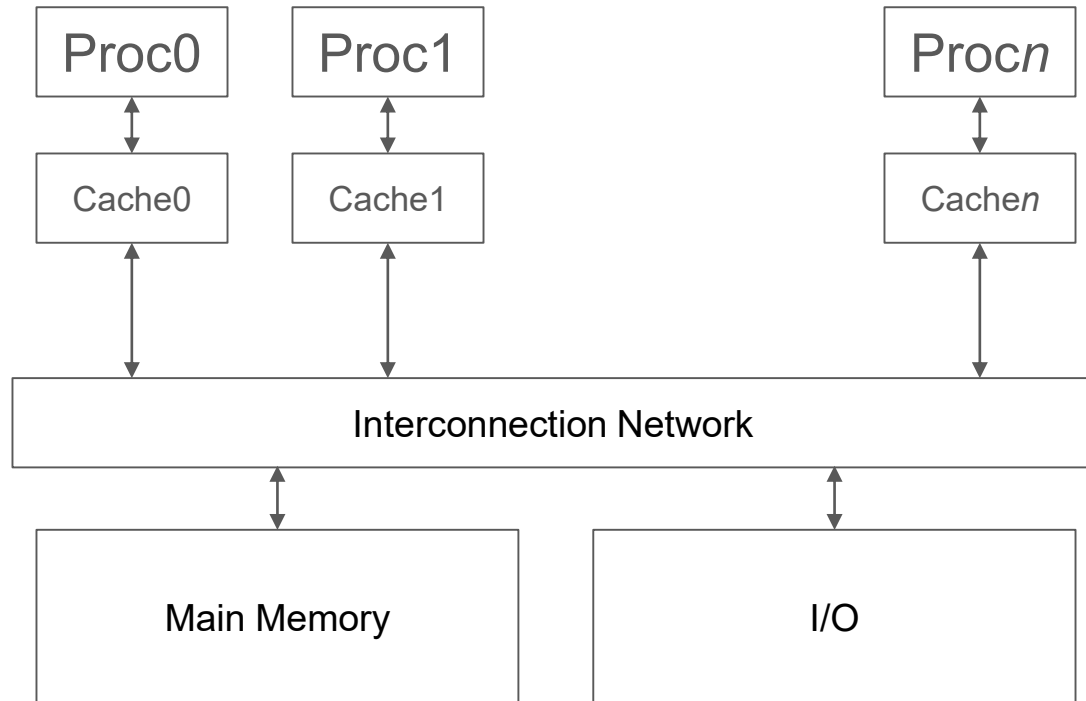
Qiang Cao revised from UCB Krste Asanovic

`http://www.eecs.berkeley.edu/~krste`

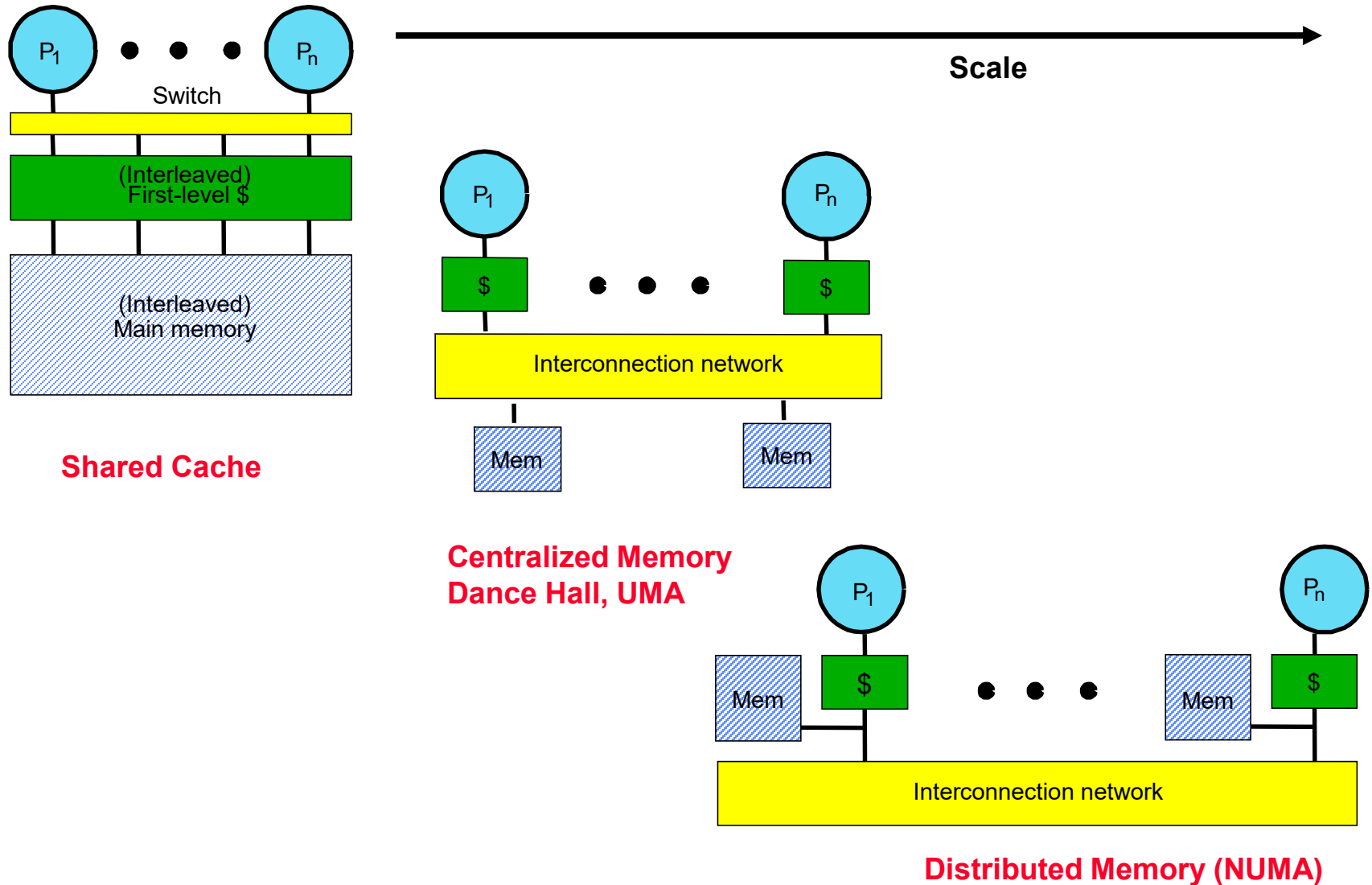
`http://inst.eecs.berkeley.edu/~cs152`

Multiprocessed System Caching

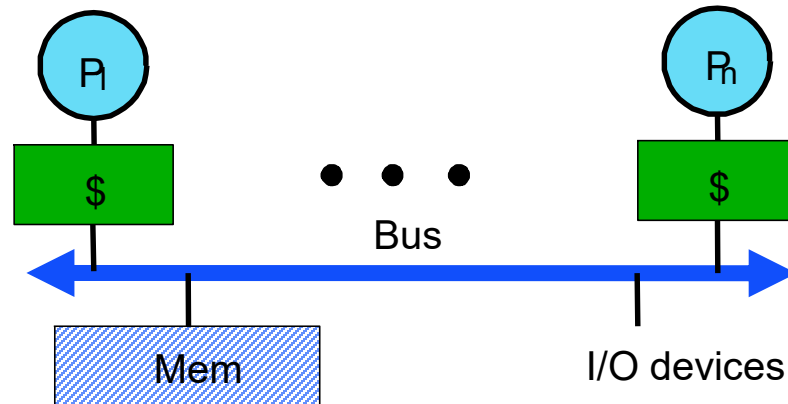
- Memory is a performance bottleneck even with one processor
- Use caches to reduce bandwidth demands on main memory
- Each core has a local private cache holding data it has accessed recently
- Only cache misses have to access the shared common memory



Natural Extensions of Memory System



Bus-Based Symmetric Shared Memory

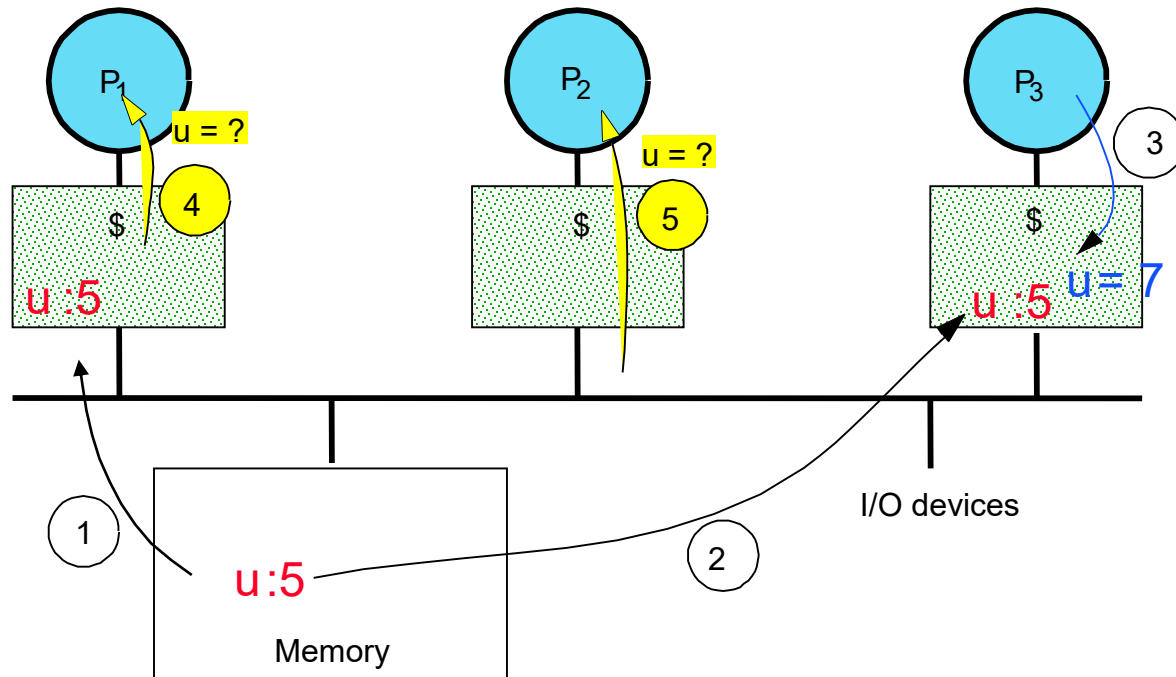


- Still an important architecture - even on chip (until very recently)
 - Building blocks for larger systems; arriving to desktop
- Attractive as throughput servers and for parallel programs
 - Fine-grain resource sharing
 - Uniform access via loads/stores
 - Automatic data movement and coherent replication in caches
 - Cheap and powerful extension
- Normal uniprocessor mechanisms to access data
 - Key is extension of memory hierarchy to support multiple processors

Caches and Cache Coherence

- Caches play key role in all cases
 - Reduce average data access time
 - Reduce bandwidth demands placed on shared interconnect
 - private processor caches create a problem
 - Copies of a variable can be present in multiple caches
 - A write by one processor may not become visible to others
 - » They'll keep accessing stale value in their caches
- ⇒ Cache coherence problem
- What do we do about it?
 - Organize the mem hierarchy to make it go away
 - Detect and take actions to eliminate the problem

Example Cache Coherence Problem



Things to note:

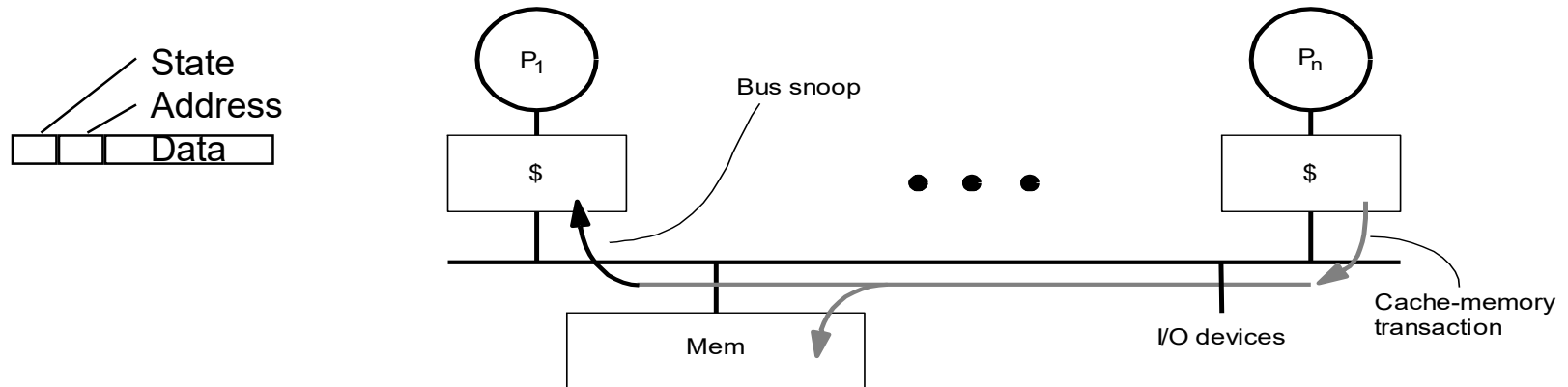
Processors see different values for u after event 3

With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when

Processes accessing main memory may see very stale value

Unacceptable to programs, and frequent!

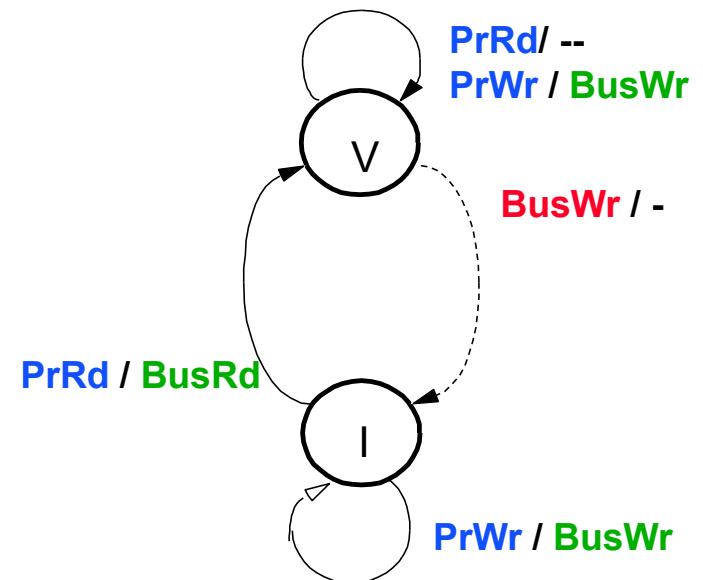
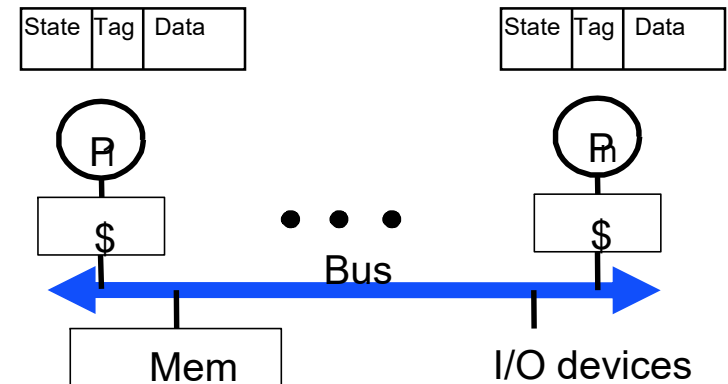
Snoopy Cache-Coherence Protocols



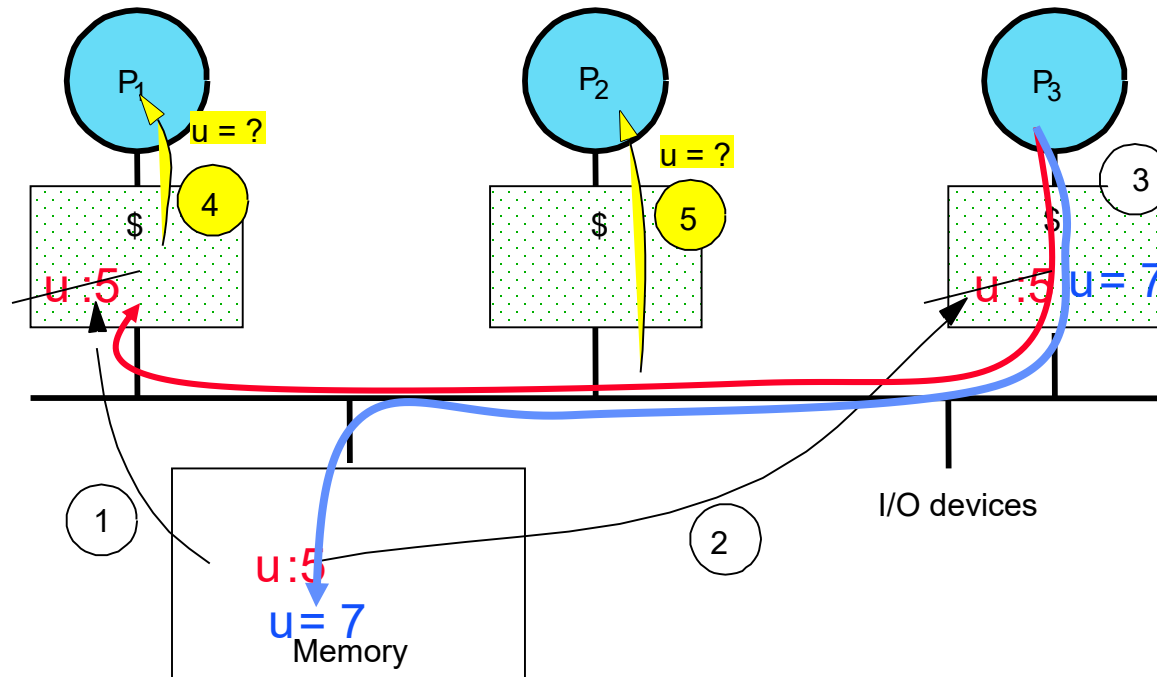
- Works because bus is a broadcast medium & Caches know what they have
- Cache Controller “snoops” all transactions on the shared bus
 - relevant transaction if for a block it contains
 - take action to ensure coherence
 - » invalidate, update, or supply value
 - depends on state of the block and the protocol

Write-through Invalidate Protocol

- **Basic Bus-Based Protocol**
 - Each processor has cache, state
 - All transactions over bus snoop
- **Writes invalidate all other caches**
 - can have multiple simultaneous readers of block, but write invalidates them
- **Two states per block in each cache**
 - as in uniprocessor
 - state of a block is a *p*-vector of states
 - Hardware state bits associated with blocks that are in the cache
 - other blocks can be seen as being in invalid (not-present) state in that cache



Example: Write-thru Invalidate



Write-through vs. Write-back

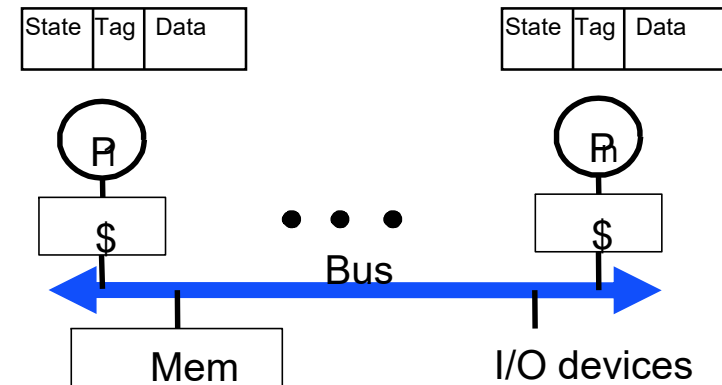
- Write-through protocol is simple
 - every write is observable
- Every write goes on the bus
 - ⇒ Only one write can take place at a time in any processor
- Uses a lot of bandwidth!

Example: 200 MHz dual issue, CPI = 1,
15% stores of 8 bytes

30 M stores per second per processor

240 MB/s per processor

1GB/s bus can support only about
4 processors without saturating



Invalidate vs. Update

- Basic question of program behavior:
 - Is a block written by one processor later read by others before it is overwritten?
 - Invalidate.
 - yes: readers will take a miss
 - no: multiple writes without additional traffic
 - » also clears out copies that will never be used again
 - Update.
 - yes: avoids misses on later references
 - no: multiple useless updates
 - » even to nodes that have dropped value from cache!
- ⇒ Need to look at program reference patterns and hardware complexity
- ⇒ Can we tune this automatically???
- but first - correctness

Coherence?

- Caches are supposed to be transparent
- What would happen if there were no caches
- Every memory operation would go “to the memory location”
 - may have multiple memory banks
 - all operations on a particular location would be serialized
 - » all would see THE order
- Interleaving among accesses from different processors
 - within individual processor => program order
 - across processors => only constrained by explicit synchronization
- Processor only observes state of memory system by issuing memory operations!

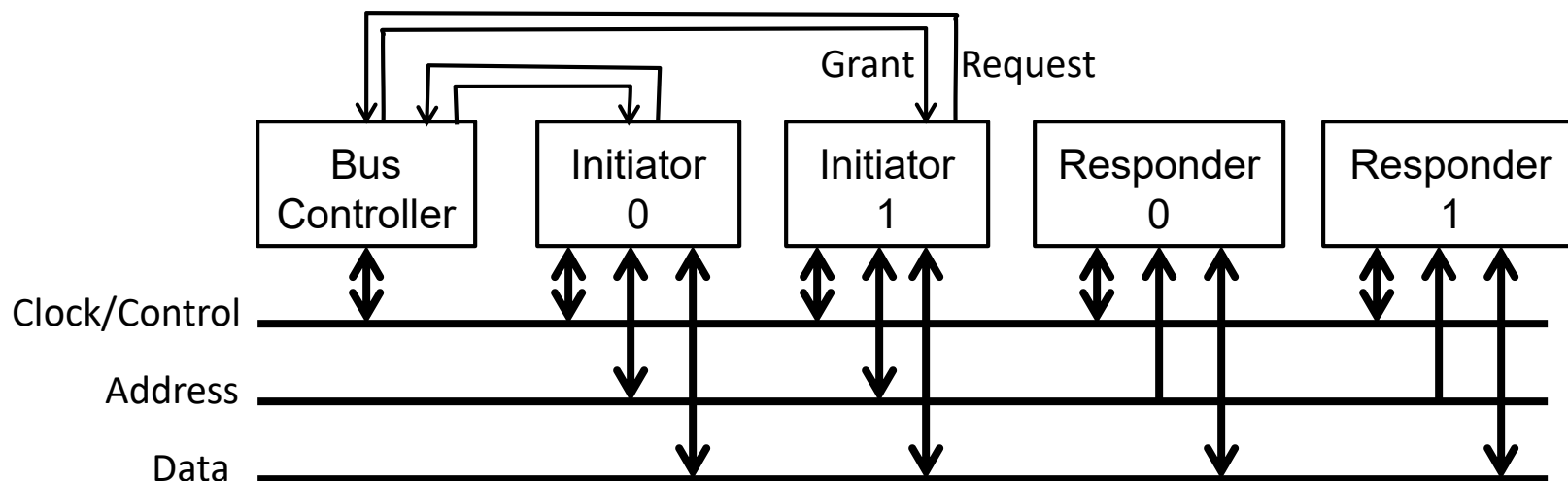
Definitions

- **Memory operation**
 - load, store, read-modify-write
 - **Issues**
 - leaves processor's internal environment and is presented to the memory subsystem (caches, buffers, busses, dram, etc)
 - **Performed with respect to a processor**
 - write: subsequent reads return the value
 - read: subsequent writes cannot affect the value
 - **Coherent Memory System**
 - there exists a serial order of mem operations on each location s.t.
 - » operations issued by a process appear in order issued
 - » value returned by each read is that written by previous write in the serial order
- => write propagation + write serialization

Is 2-state Protocol Coherent?

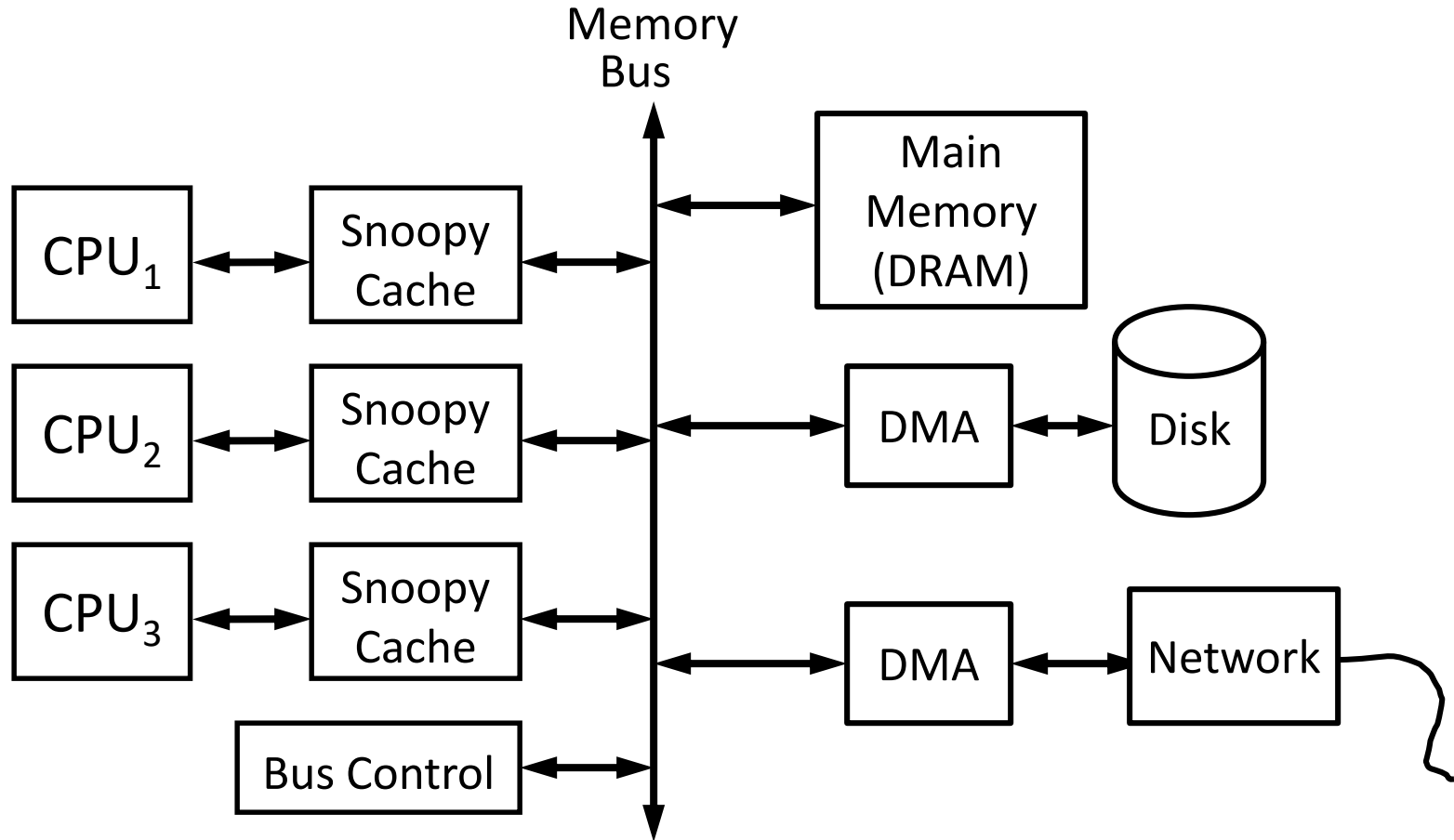
- Assume bus transactions and memory operations are atomic, one-level cache
 - all phases of one bus transaction complete before next one starts
 - processor waits for memory op to complete before issuing next
 - with one-level cache, assume invalidations applied during bus xaction
- All writes go to bus + atomicity
 - **Writes serialized** by order in which they appear on bus (bus order)
⇒ invalidations applied to caches in bus order
- How to insert reads in this order?
 - Important since processors see writes through reads, so determines whether write serialization is satisfied
 - But read hits may happen independently and do not appear on bus or enter directly in bus order

Bus Management



- A “bus” is a collection of shared wires
 - Newer “busses” use point-point links
- At any instant, only one “initiator” can initiate a transaction by driving wires
 - Initiators arbitrate for access with requests to bus “controller”
 - Some busses only allow one initiator (in which case, it’s also the controller)
- Multiple “responders” can observe and conditionally respond to the transaction on the wires
 - responders decode address on bus to see if they should respond (memory is most common responder)
 - some initiators can also act as responders

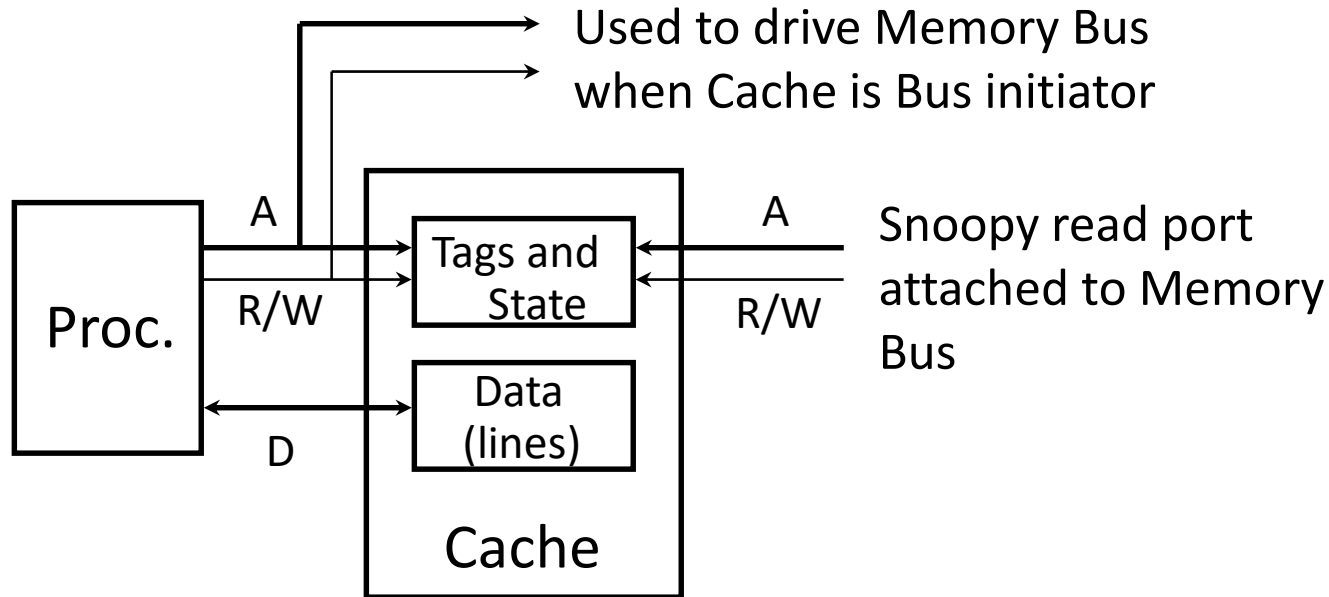
Shared-Memory Multiprocessor



Use snoop mechanism to keep all processors' view of memory coherent

Snoopy Cache, *Goodman 1983*

- Idea: Have cache watch (or snoop upon) other memory transactions, and then “do the right thing”
- Snoopy cache tags are dual-ported

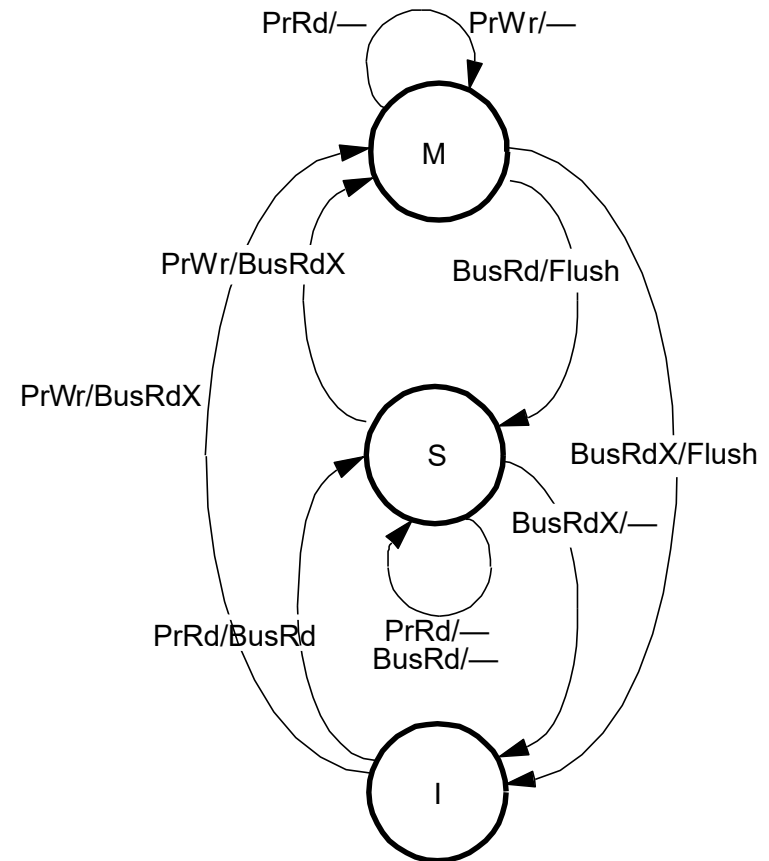


Snoopy Cache-Coherence Protocols

- Write miss:
 - the address is invalidated in all other caches before the write is performed
- Read miss:
 - if a dirty copy is found in some cache, a write-back is performed before the memory is read

MSI Invalidate Protocol

- **Three States:**
 - "M": "Modified"
 - "S": "Shared"
 - "I": "Invalid"
- **Read obtains block in "shared"**
 - even if only cache copy
- **Obtain exclusive ownership before writing**
 - BusRdx causes others to invalidate (demote)
 - If M in another cache, will flush
 - BusRdx even if hit in S
 - » promote to M (upgrade)
- **What about replacement?**
 - S → I, M → I as before



Cache State-Transition Diagram

The MSI protocol

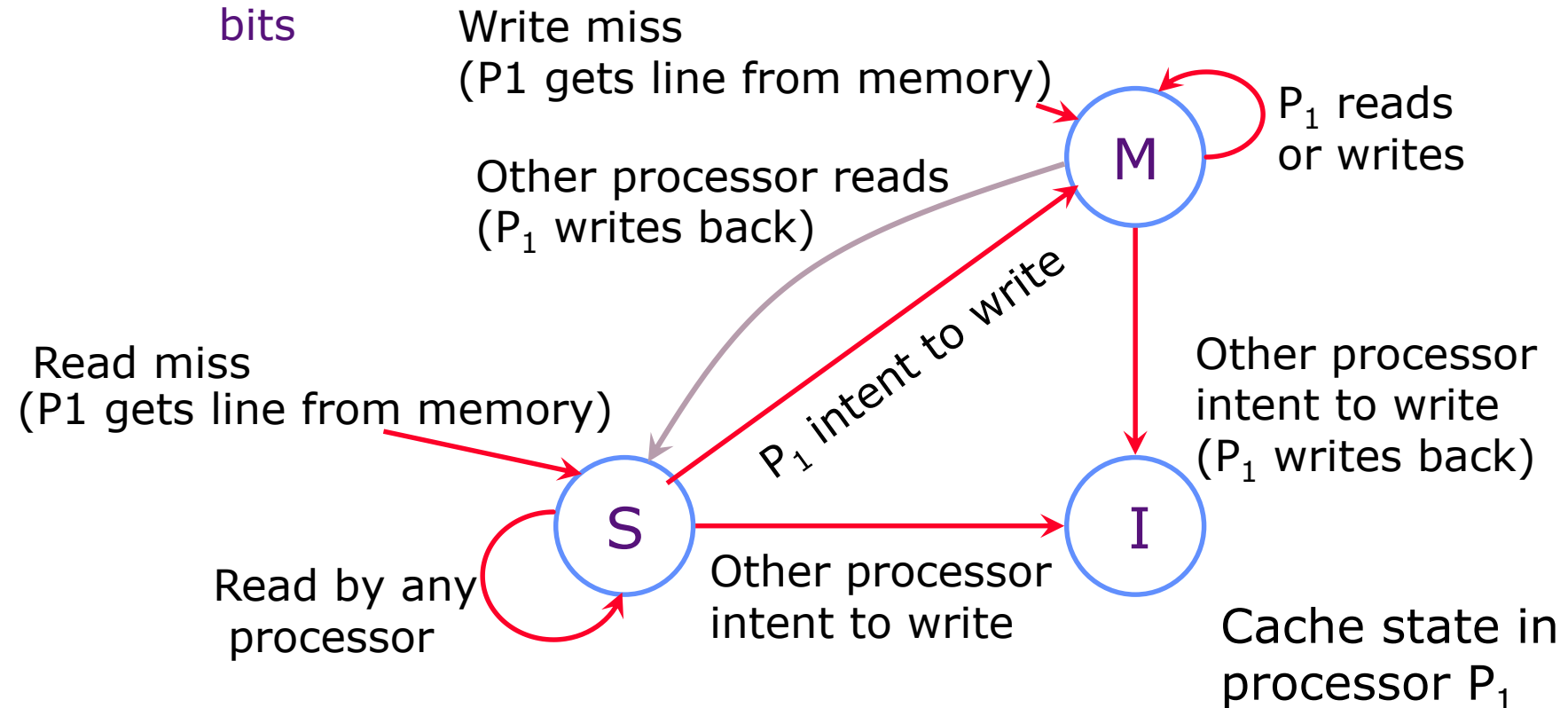
Each cache line has state bits



M: Modified

S: Shared

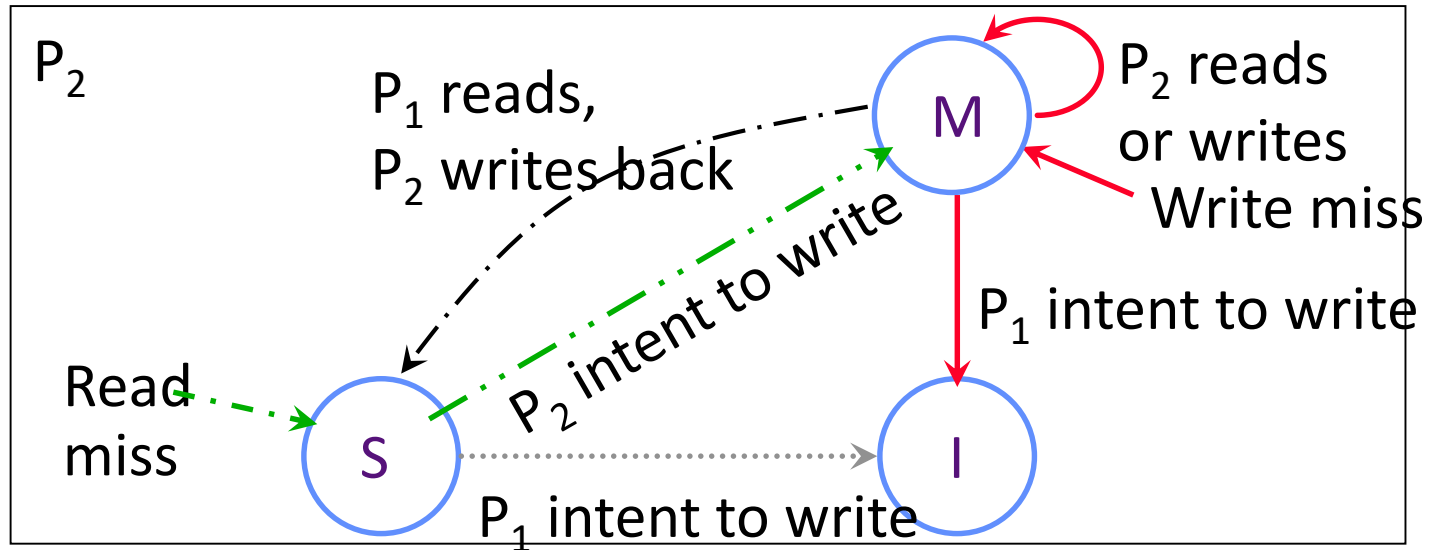
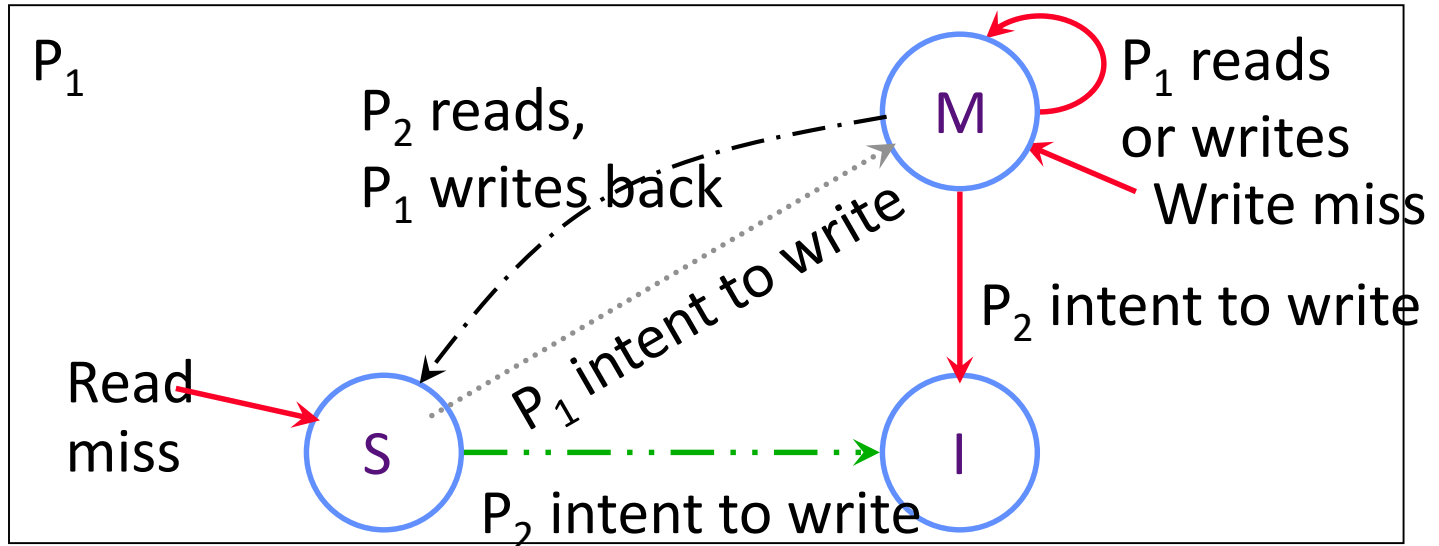
I: Invalid



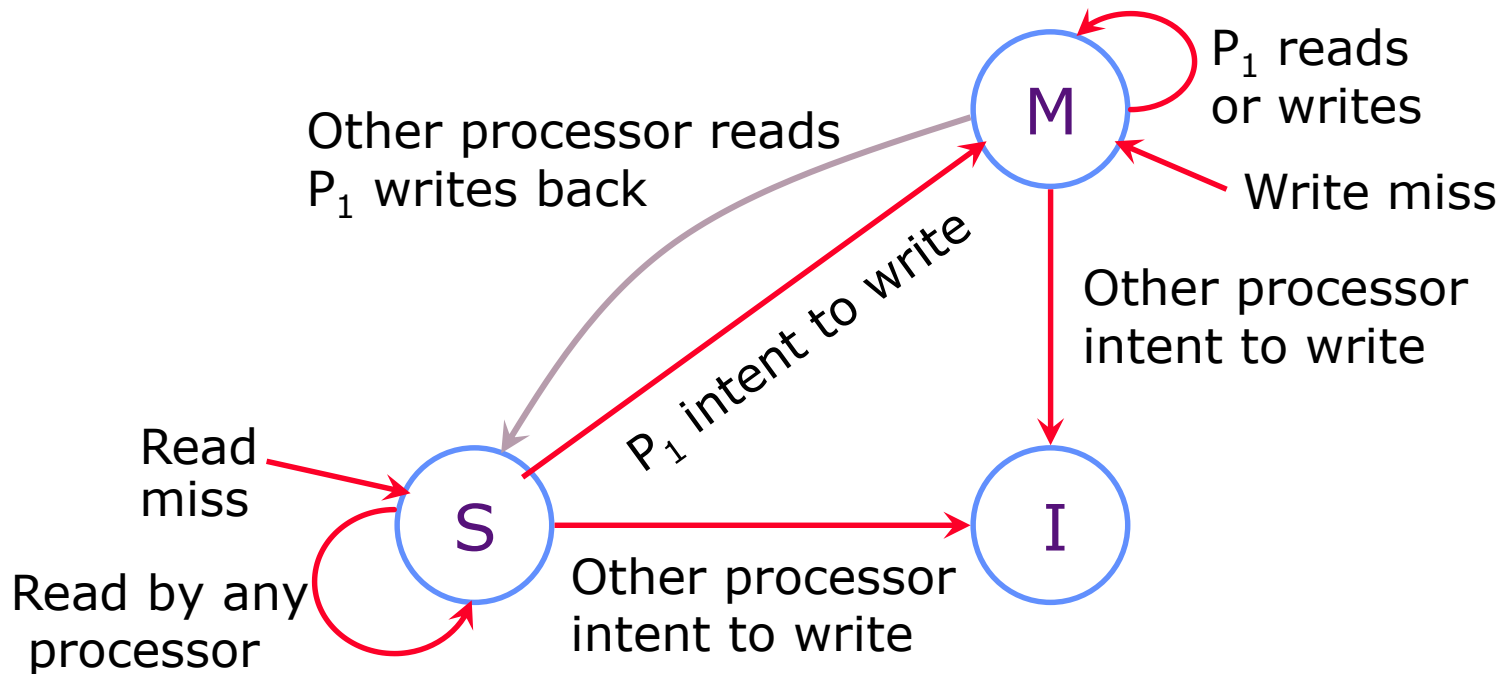
Two-Processor Example

(Reading and writing the same cache line)

P_1 reads
 P_1 writes
 P_2 reads
 P_2 writes
 P_1 reads
 P_1 writes
 P_2 writes
 P_1 writes



Observation



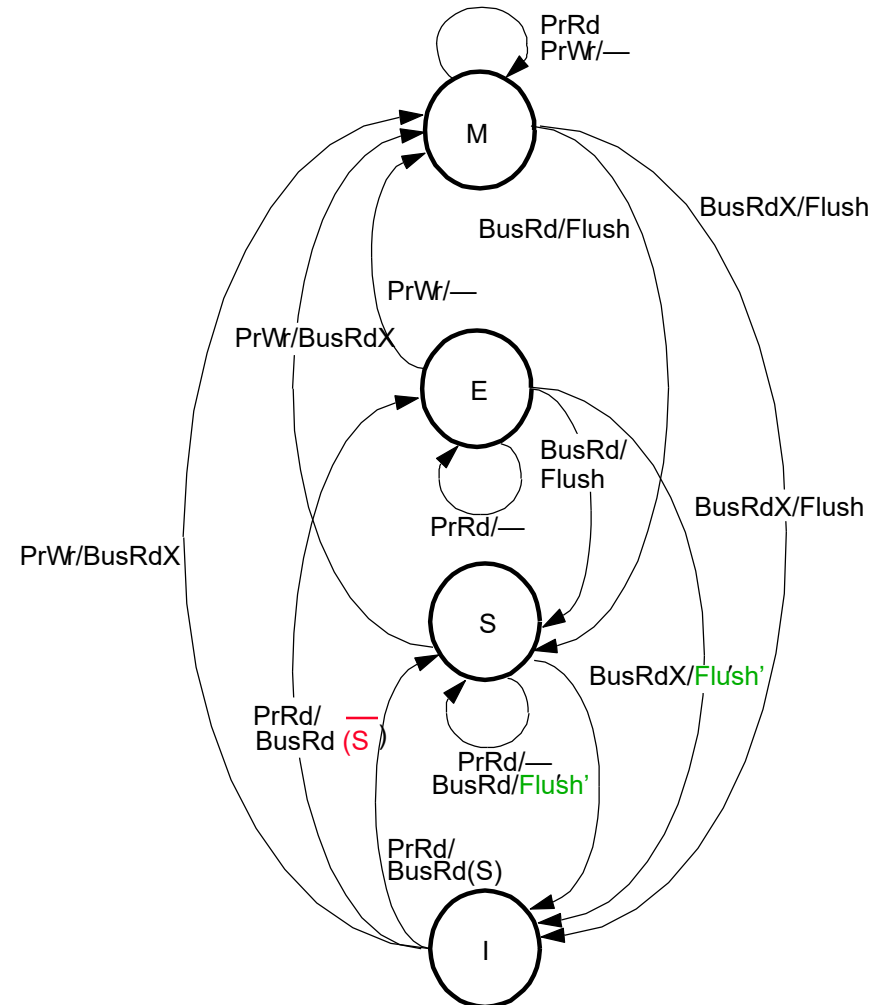
- If a line is in the **M** state then no other cache can have a copy of the line!
- Memory stays coherent, multiple differing copies cannot exist

MESI (4-state) Invalidation Protocol

- Four States:
 - "M": "Modified"
 - "E": "Exclusive"
 - "S": "Shared"
 - "I": "Invalid"
- Add *exclusive* state
 - distinguish exclusive (writable) and owned (written)
 - Main memory is up to date, so cache not necessarily owner
 - can be written locally
- States
 - invalid
 - exclusive or *exclusive-clean* (only this cache has copy, but not modified)
 - shared (two or more caches may have copies)
 - modified (dirty)
- I -> E on PrRd if no cache has copy
 - => How can you tell?

MESI State Transition Diagram

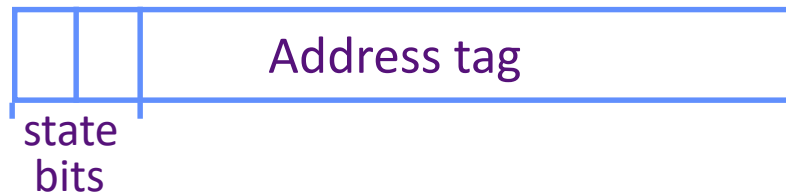
- BusRd(**S**) means shared line asserted on BusRd transaction
- Flush': if cache-to-cache xfers
 - only one cache flushes data
- Replacement:
 - S→I can happen without telling other caches
 - E→I, M→I
- MOESI protocol: Owned state: exclusive but memory not valid



MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag

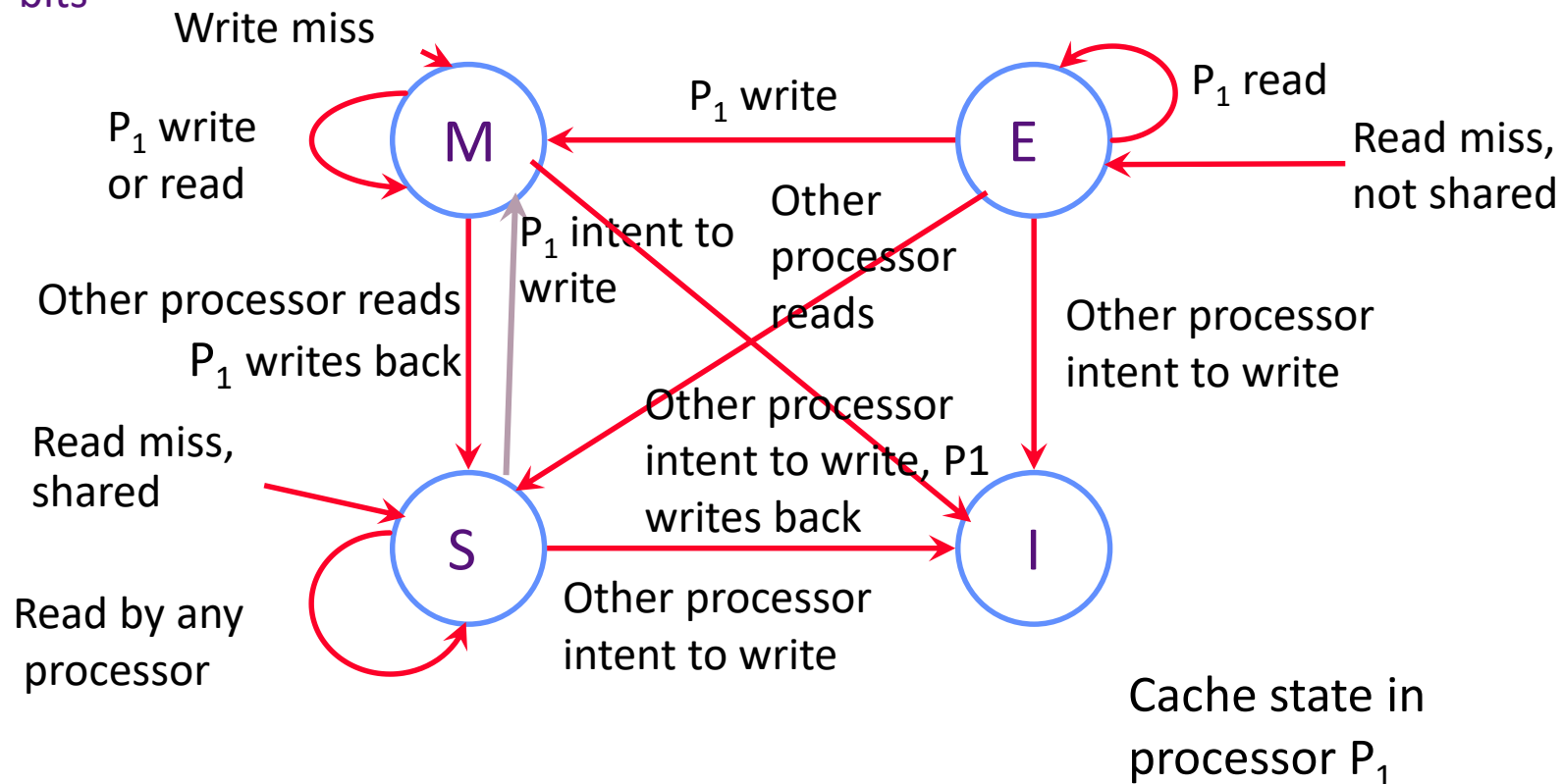


M: Modified Exclusive

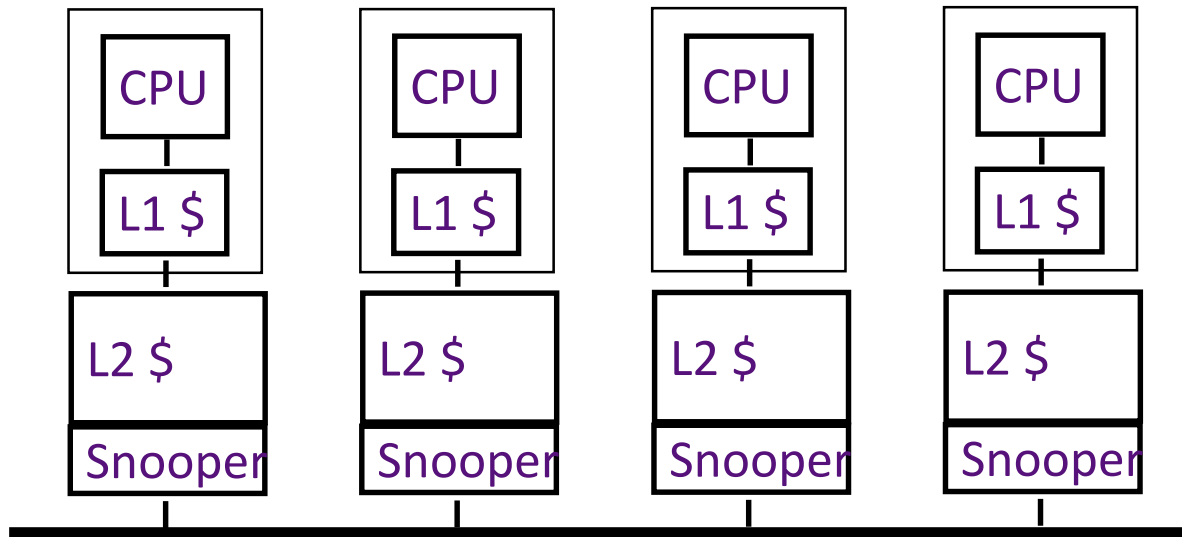
E: Exclusive but unmodified

S: Shared

I: Invalid

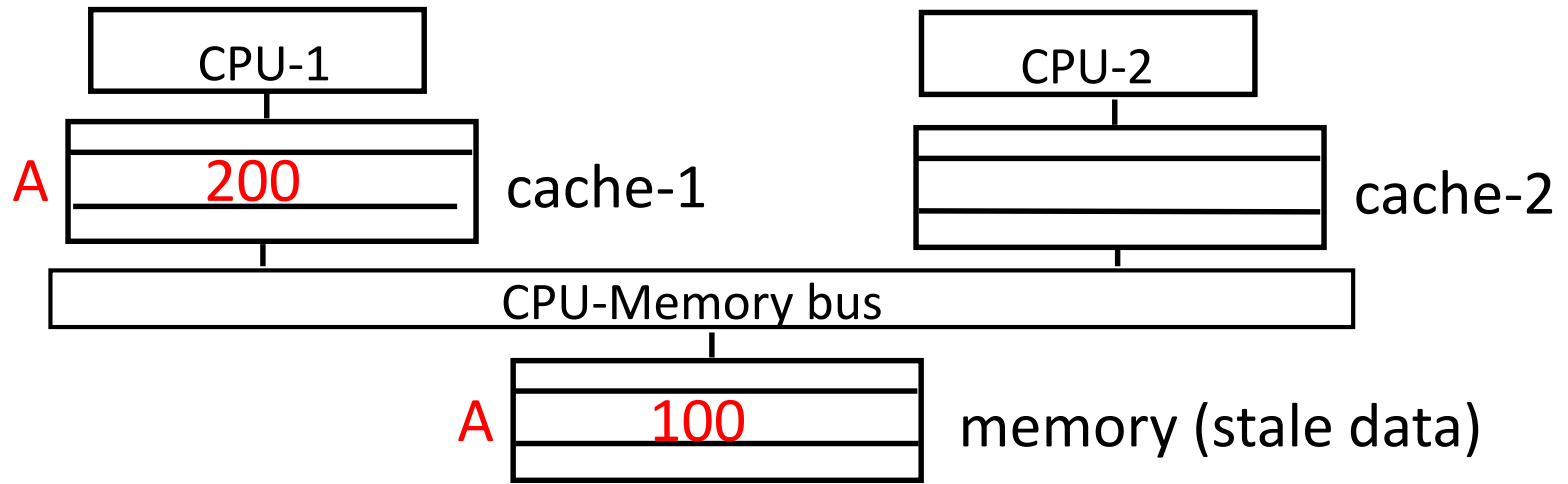


Optimized Snoop with Level-2 Caches



- Processors often have two-level caches
 - small L1, large L2 (usually both on chip now)
- Inclusion property: entries in L1 must be in L2
 - Miss in L2 \Rightarrow Not present in L1
 - Only if invalidation hits in L2 \Rightarrow probe and invalidate in L1
- Snooping on L2 does not affect CPU-L1 bandwidth

Intervention



When a read-miss for **A** occurs in cache-2,
Cache-2 initiates a read request for **A** on the bus

- Cache-1 needs to supply & change its state to shared
- The memory may respond to the request also!

Does memory know it has stale data?

Cache-1 needs to *intervene* through memory controller to supply correct data to cache-2

False Sharing

state	line addr	data0	data1	...	dataN
-------	-----------	-------	-------	-----	-------

A cache line contains more than one word

Cache-coherence is done at the line-level and not word-level

Suppose M_1 writes word_i and M_2 writes word_k and $i \neq k$ but both words have the same line address.

What can happen?

Performance of Symmetric Multiprocessors (SMPs)

Cache performance is combination of:

- Uniprocessor cache miss traffic
- Traffic caused by communication
 - Results in invalidations and subsequent cache misses
- Coherence misses
 - Sometimes called a Communication miss
 - 4th C of cache misses along with Compulsory, Capacity, & Conflict

Coherency Misses

- True sharing misses arise from the communication of data through the cache coherence mechanism
 - Invalidates due to 1st write to shared line
 - Reads by another CPU of modified line in different cache
 - Miss would still occur if line size were 1 word
- False sharing misses when a line is invalidated because some word in the line, other than the one being read, is written into
 - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
 - Line is shared, but no word in line is actually shared
⇒ miss would not occur if line size were 1 word

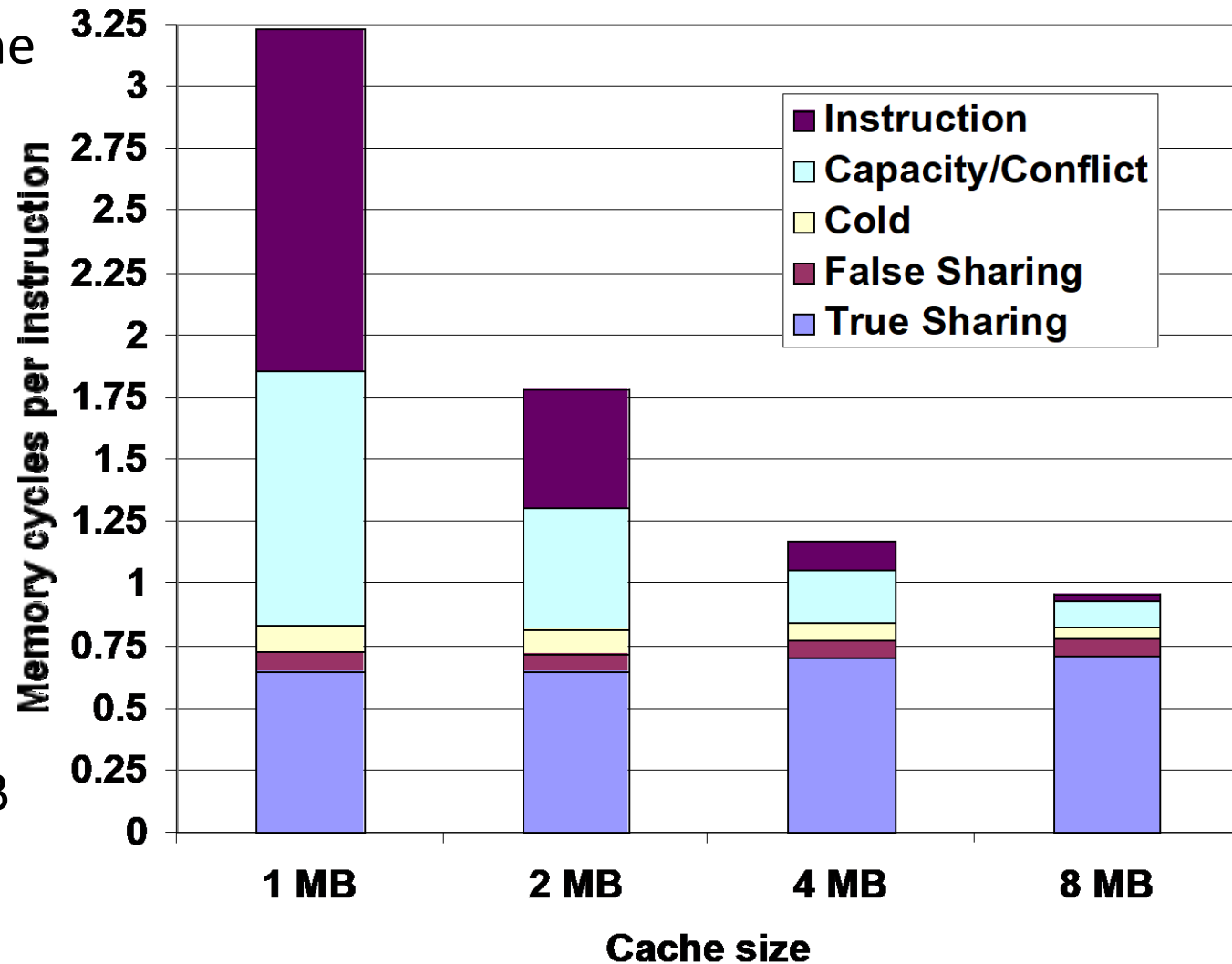
Example: True v. False Sharing v. Hit?

- MSI protocol
- Assume x1 and x2 in same cache line.
P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	True miss; x2 not writeable
5	Read x2		True miss; x2 invalid in P1

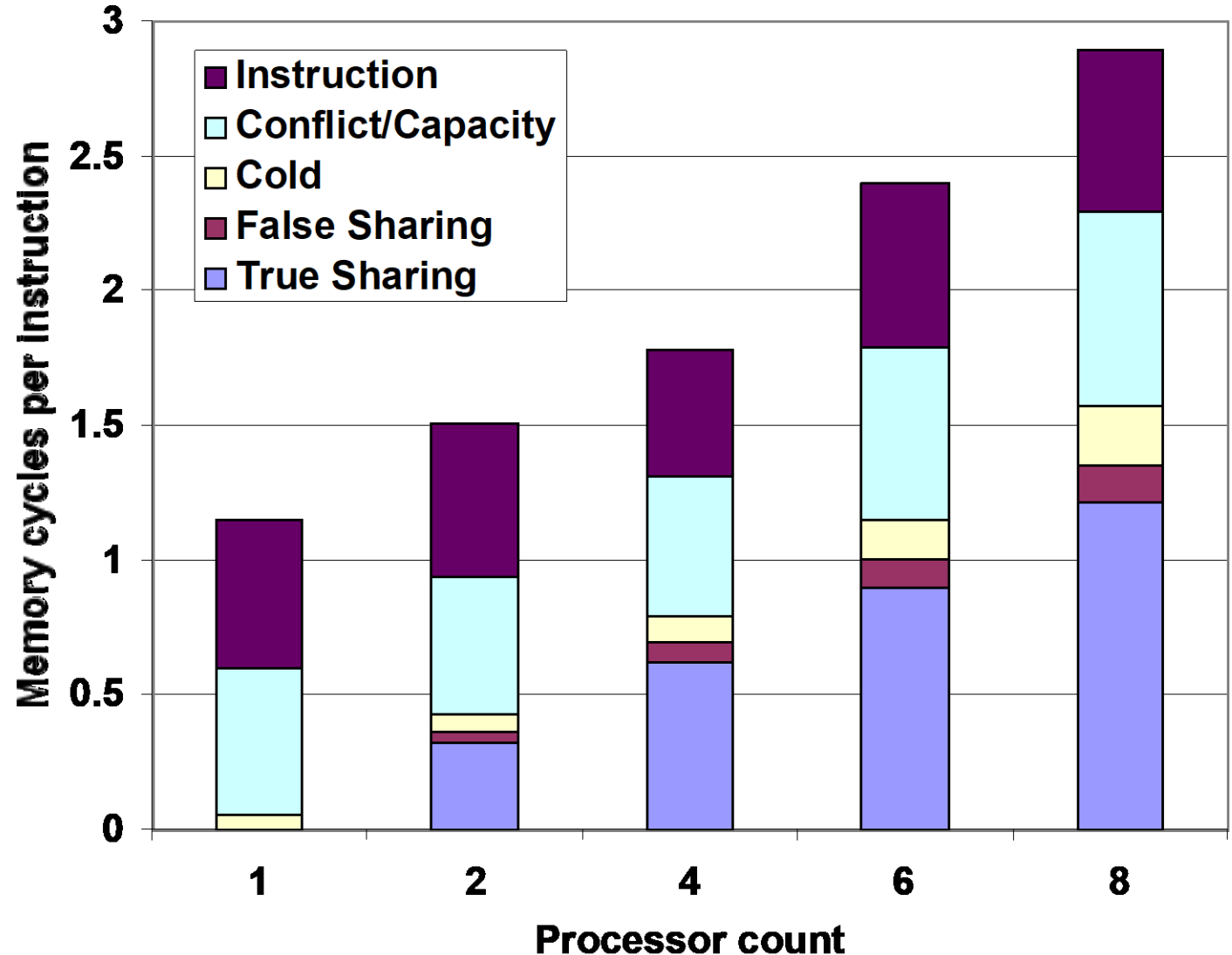
MP Performance 4-Processor Commercial Workload: OLTP, Decision Support (Database), Search Engine

- Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)
- True sharing and false sharing unchanged going from 1 MiB to 8 MiB (L3 cache)



MP Performance 2MiB Cache Commercial Workload: OLTP, Decision Support (Database), Search Engine

- True sharing, false sharing increase going from 1 to 8 CPUs



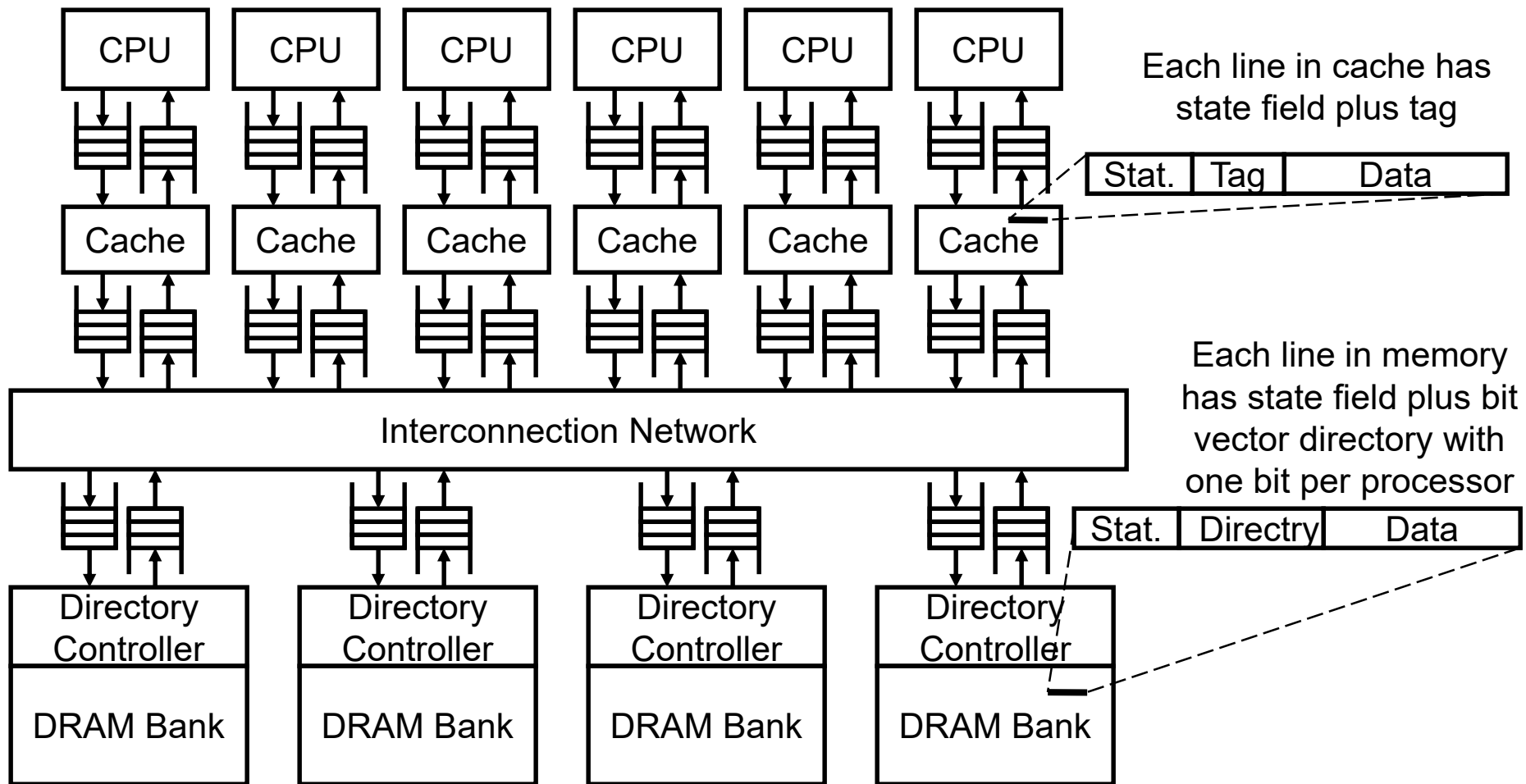
Scaling Snoopy/Broadcast Coherence

- When any processor gets a miss, must probe every other cache
- Scaling up to more processors limited by:
 - Communication bandwidth over bus
 - Snoop bandwidth into tags
- Can improve bandwidth by using multiple interleaved buses with interleaved tag banks
 - E.g, two bits of address pick which of four buses and four tag banks to use
 - (e.g., bits 7:6 of address pick bus/tag bank, bits 5:0 pick byte in 64-byte line)
- Buses don't scale to large number of connections, so can use point-to-point network for larger number of nodes, but then limited by tag bandwidth when broadcasting snoop requests.
- **Insight:** Most snoops fail to find a match!

Scalable Approach: Directories

- Every memory line has associated directory information
 - keeps track of copies of cached lines and their states
 - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
 - in scalable networks, communication with directory and copies is through network transactions
- Many alternatives for organizing directory information

Directory Cache Protocol



- Assumptions: Reliable network, FIFO message delivery between any given source-destination pair

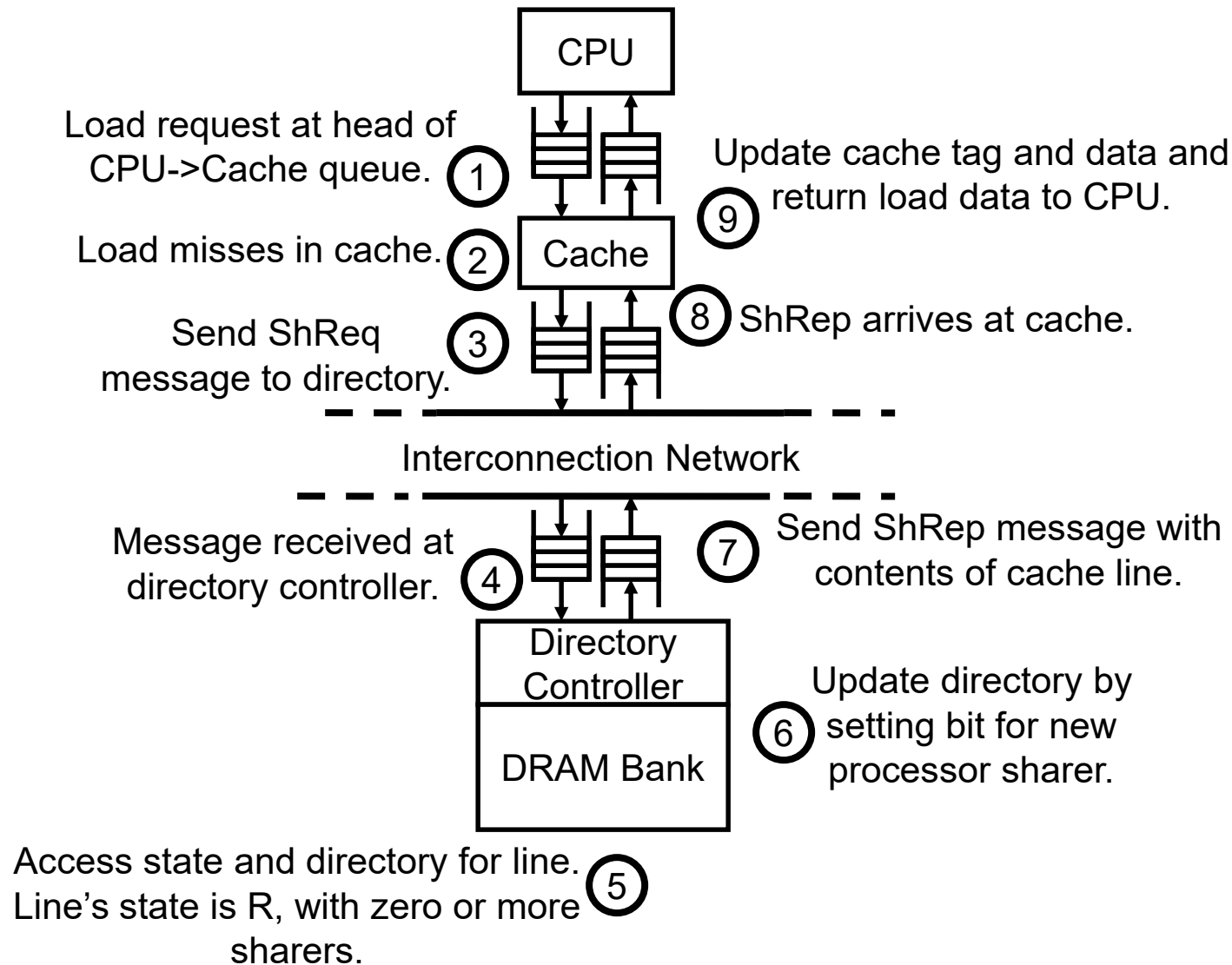
Cache States

- For each cache line, there are 4 possible states:
 - **C-invalid** (= Nothing): The accessed data is not resident in the cache.
 - **C-shared** (= Sh): The accessed data is resident in the cache, and possibly also cached at other sites. The data in memory is valid.
 - **C-modified** (= Ex): The accessed data is exclusively resident in this cache, and has been modified. Memory does not have the most up-to-date data.
 - **C-transient** (= Pending): The accessed data is in a transient state (for example, the site has just issued a protocol request, but has not received the corresponding protocol reply).

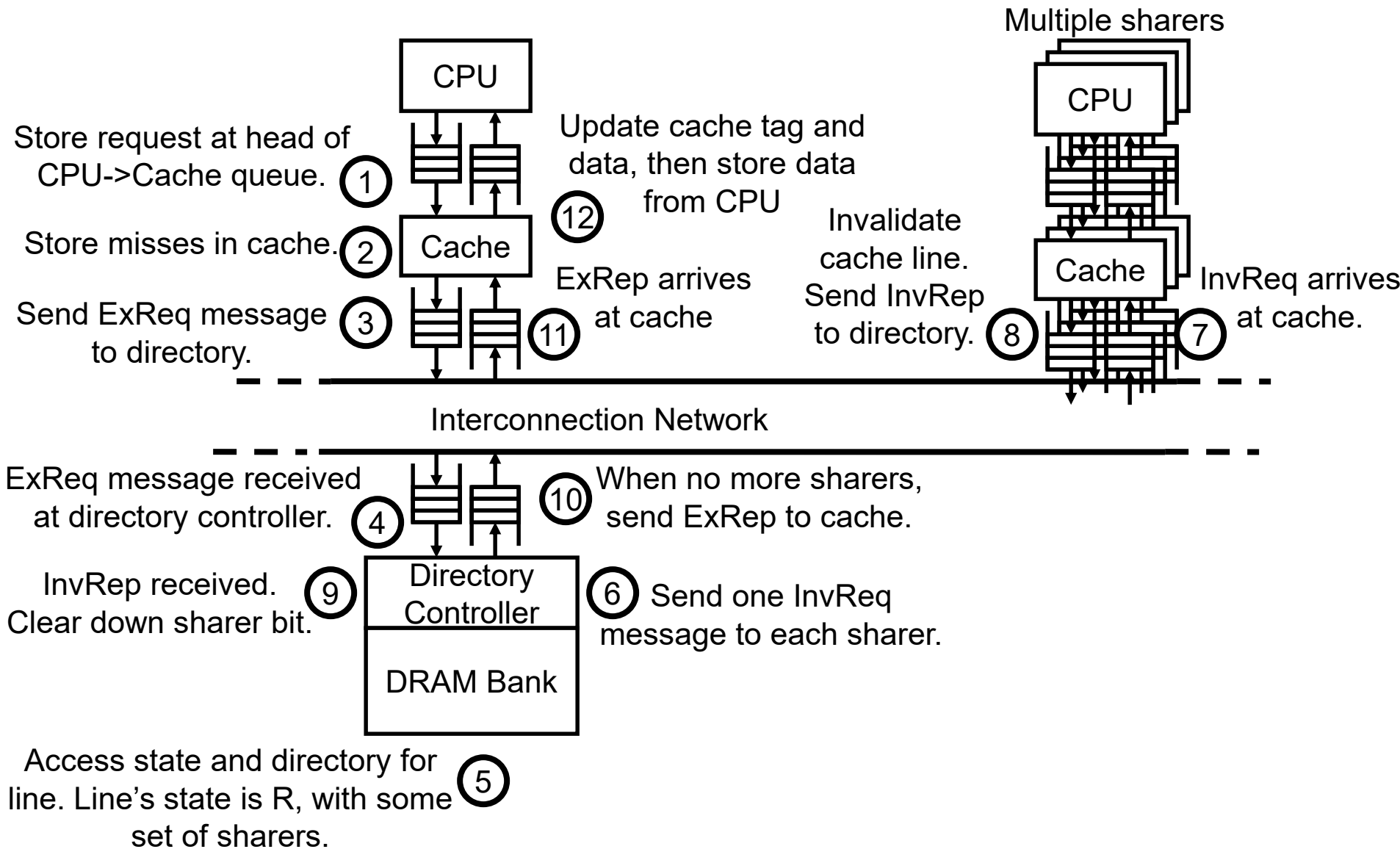
Home directory states

- For each memory line, there are 4 possible states:
 - **R(dir)**: The memory line is shared by the sites specified in dir (dir is a set of sites). The data in memory is valid in this state. If dir is empty (i.e., $\text{dir} = \epsilon$), the memory line is not cached by any site.
 - **W(id)**: The memory line is exclusively cached at site id, and has been modified at that site. Memory does not have the most up-to-date data.
 - **TR(dir)**: The memory line is in a transient state waiting for the acknowledgements to the invalidation requests that the home site has issued.
 - **TW(id)**: The memory line is in a transient state waiting for a line exclusively cached at site id (i.e., in C-modified state) to make the memory line at the home site up-to-date.

Read miss, to uncached or shared line



Write miss, to read shared line



Concurrency Management

- Protocol would be easy to design if only one transaction in flight across entire system
- But, want greater throughput and don't want to have to coordinate across entire system
- Great complexity in managing multiple outstanding concurrent transactions to cache lines
 - Can have multiple requests in flight to same cache line!

CS 152 Computer Architecture and Engineering

CS252 Graduate Computer Architecture

Lecture 10-2 Memory Consistency Models

Krste Asanovic

Electrical Engineering and Computer Sciences
University of California at Berkeley

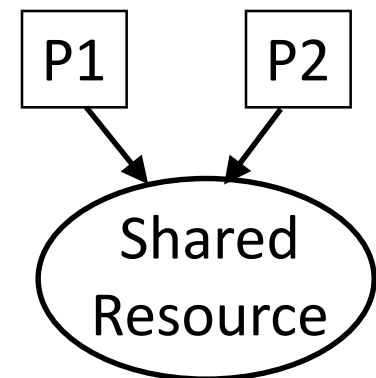
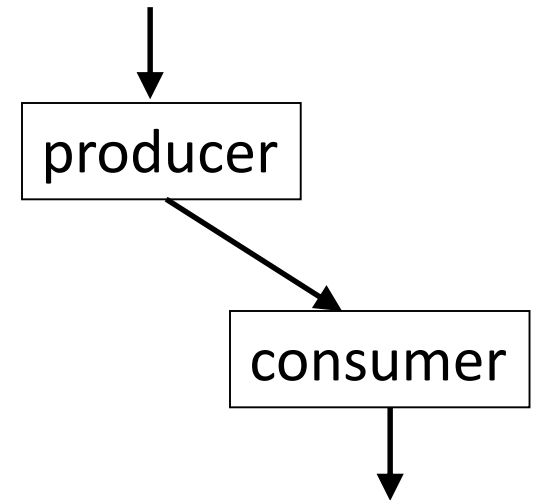
`http://www.eecs.berkeley.edu/~krste`
`http://inst.eecs.berkeley.edu/~cs152`

Synchronization

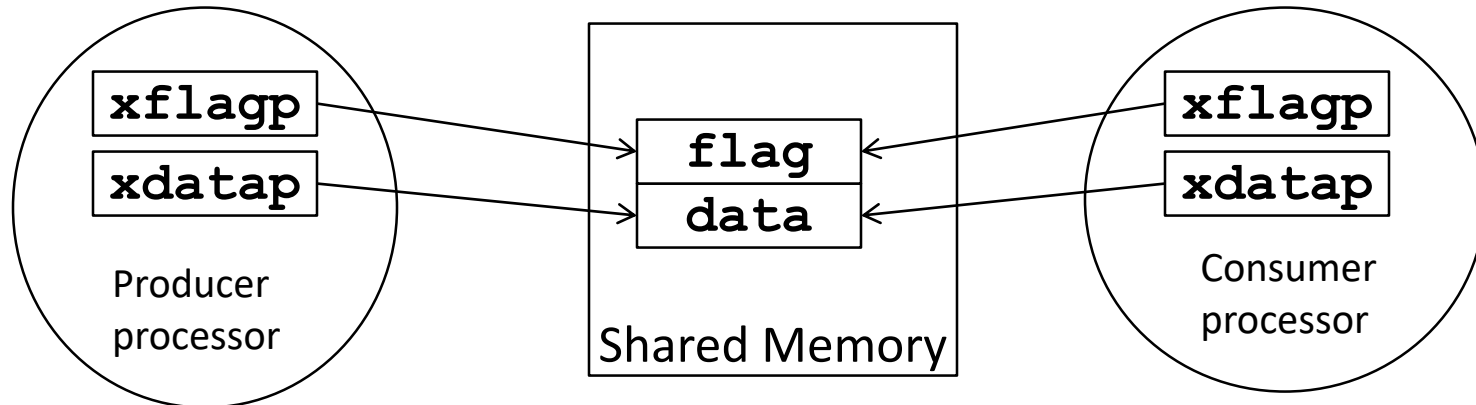
The need for synchronization arises whenever there are concurrent processes in a system (*even in a uniprocessor system*).

Two classes of synchronization:

- *Producer-Consumer*: A consumer process must wait until the producer process has produced data
- *Mutual Exclusion*: Ensure that only one process uses a resource at a given time



Simple Producer-Consumer Example



Initially **flag=0**

```
sw xdata, (xdatap)
li xflag, 1
sw xflag, (xflagp)
```

```
spin: lw xflag, (xflagp)
      beqz xflag, spin
      lw xdata, (xdatap)
```

Is this correct?

Memory Consistency Model

- Sequential ISA only specifies that each processor sees its own memory operations in program order
- Memory consistency model describes what values can be returned by load instructions across multiple hardware threads
- *Coherence* describes the legal values a *single* memory address should return
- *Consistency* describes properties across *all* memory addresses

Simple Producer-Consumer Example



Initially **flag=0**

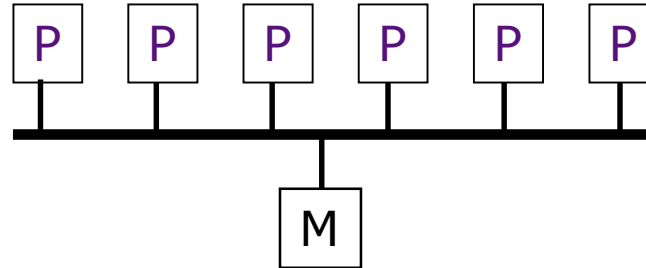
```
sw xdata, (xdatap)
li xflag, 1
sw xflag, (xflagp)
```

```
spin: lw xflag, (xflagp)
      beqz xflag, spin
      lw xdata, (xdatap)
```

Can consumer read **flag=1** before **data**
written by producer visible to consumer?

Sequential Consistency (SC)

A Memory Model



“ A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program”


Leslie Lamport


Sequential Consistency = arbitrary *order-preserving interleaving* of memory references of sequential programs

Simple Producer-Consumer Example




Initially flag = 0

 `sw xdata, (xdatap)`
`li xflag, 1`
`sw xflag, (xflagp)`

`spin: lw xflag, (xflagp)`
 `beqz xflag, spin`
`lw xdata, (xdatap)`

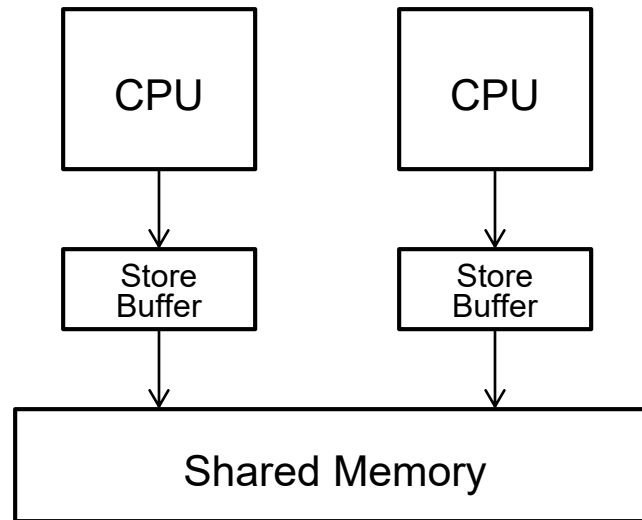
 Dependencies from sequential ISA

 Dependencies added by sequentially consistent memory model

Most real machines are not SC

- Only a few commercial ISAs require SC
 - Neither IBM 370 nor x86 nor ARM nor RISC-V are SC
- Originally, architects developed uniprocessors with optimized memory systems (e.g., store buffer)
- When uniprocessors were lashed together to make multiprocessors, resulting machines were not SC
- Requiring SC would make simpler machines slower, or requires adding complex hardware to retain performance
- Architects/language designers/applications developers work hard to explain weak memory behavior
- Resulted in “weak” memory models with fewer guarantees

Store Buffer Optimization



- Common optimization allows stores to be buffered while waiting for access to shared memory
- Load optimizations:
 - Later loads can go ahead of buffered stores if to different address
 - Later loads can bypass value from earlier buffered store if to same address

TSO example

- Allows local buffering of stores by processor

Initially $M[X] = M[Y] = 0$

P1:

li x1, 1

sw x1, X

lw x2, Y

P2:

li x1, 1

sw x1, Y

lw x2, X

Possible Outcomes

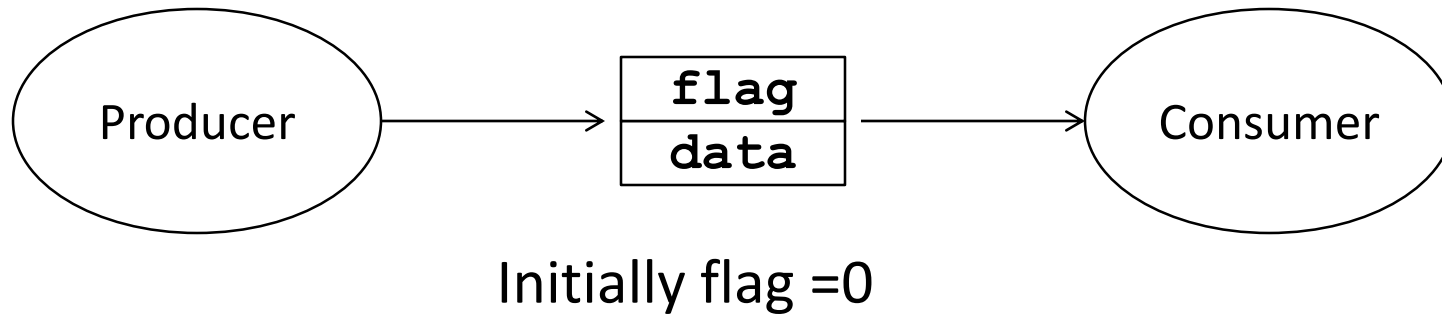
P1.x2	P2.x2	SC	TSO
0	0	N	Y
0	1	Y	Y
1	0	Y	Y
1	1	Y	Y

- TSO is the strongest memory model in common use

Strong versus Weak Memory Consistency Models

- Stronger models provide more guarantees on ordering of loads and stores across different hardware threads
 - Easier ISA-level programming model
 - Can require more hardware to ensure orderings (e.g., MIPS R10K was SC, with hardware to speculate on load/stores and squash when ordering violations detected across cores)
- Weaker models provide fewer guarantees
 - Much more complex ISA-level programming model
 - Extremely difficult to understand, even for experts
 - Simpler to achieve high performance, as weaker models allow many hardware reorderings to be exposed to software
 - Additional instructions (fences) are provided to allow software to specify which orderings are required

Fences in Producer-Consumer Example



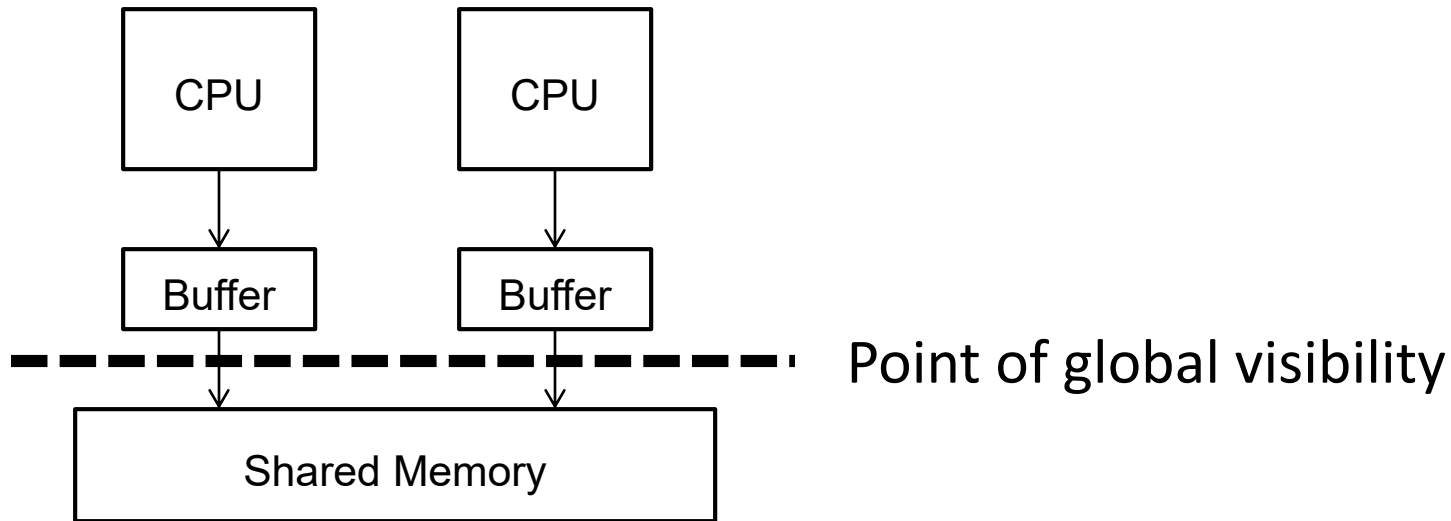
```
sw xdata, (xdatap)
li xflag, 1
fence w,w //Write-write fence
sw xflag, (xflagp)
```

```
spin: lw xflag, (xflagp)
      beqz xflag, spin
fence r,r // Read-read fence
      lw xdata, (xdatap)
```

Range of Memory Consistency Models

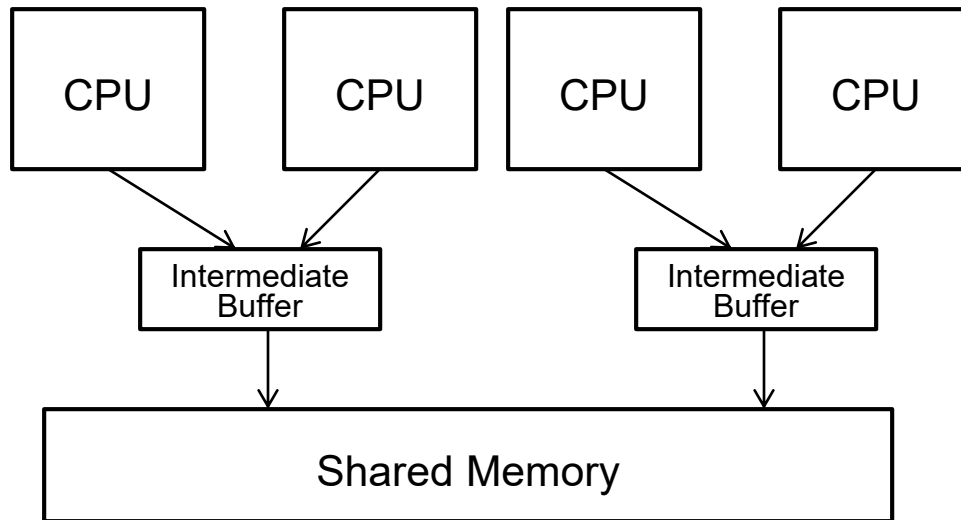
- SC “Sequential Consistency”
 - MIPS R10K
- TSO “Total Store Order”
 - *processor can see its own writes before others do (store buffer)*
 - IBM-370 TSO, x86 TSO, SPARC TSO (default), RISC-V RVTSO (optional)
- Weak, multi-copy-atomic memory models
 - *all processors see writes by another processor in same order*
 - Revised ARM v8 memory model
 - RISC-V RVWMO, baseline weak memory model for RISC-V
- Weak, non-multi-copy-atomic memory models
 - *processors can see another’s writes in different orders*
 - ARM v7, original ARM v8
 - IBM POWER
 - Digital Alpha (extremely weak MCM)
 - Recent consensus is that these appear to be too weak for general-purpose processors

Multi-Copy Atomic models



- Each hardware thread must view its own memory operations in program order, but can buffer these locally and reorder accesses around the buffer
- But once a local store is made visible to one other hardware thread in system, all other hardware threads must also be able to observe it (this is what is meant by “atomic”)

Hierarchical Shared Buffering



- Common in large systems to have shared intermediate buffers on path between CPUs and global memory
- Potential optimization is to allow some CPUs see some writes by a CPU before other CPUs
- Shared memory stores are not seen to happen atomically by other threads (non multi-copy atomic)

Non-Multi-Copy Atomic

Initially $M[X] = M[Y] = 0$

P1:	P2:	P3:
li x1, 1	lw x1, X	lw x1, Y
sw x1, X	sw x1, Y	fence r,r
		lw x2, X

Can $P3.x1 = 1$, and $P3.x2 = 0$?

- In general, Non-MCA is very difficult to reason about
- Software in one thread cannot assume all data it sees is visible to other threads, so how to share data structures?
- Adding local fences to require ordering of each thread's accesses is insufficient – need a more global memory barrier to ensure all writes are made visible

Relaxed Memory Models

- Not all dependencies assumed by SC are supported, and software has to explicitly insert additional dependencies where needed
- Which dependencies are dropped depends on the particular memory model
 - IBM370, TSO, PSO, WO, PC, Alpha, RMO, ...
 - Some ISAs allow several memory models, some machines have switchable memory models
- How to introduce needed dependencies varies by system
 - Explicit FENCE instructions (sometimes called sync or memory barrier instructions)
 - Implicit effects of atomic memory instructions

How on earth are programmers supposed to work with this???

But compilers reorder too!

```
//Producer code  
*datap = x/y;  
*flagp = 1;
```

```
//Consumer code  
while (!*flagp)  
    ;  
d = *datap;
```

- Compiler can reorder/remove memory operations:
 - Instruction scheduling, move loads before stores if to different address
 - Register allocation, cache load value in register, don't check memory
- Prohibiting these optimizations would result in very poor performance

Language-Level Memory Models

- Programming languages have memory models too
- Hide details of each ISA's memory model underneath language standard
 - c.f. C function declarations versus ISA-specific subroutine linkage convention
- Language memory models: C/C++, Java
- Describe legal behaviors of threaded code in each language and what optimizations are legal for compiler to make
- E.g., C11/C++11: `atomic_load(memory_order_seq_cst)` maps to RISC-V `fence rw,rw; lw; fence r,rw`

CS 152 Computer Architecture and Engineering

CS252 Graduate Computer Architecture

Lecture 10-3 Synchronization

Krste Asanovic

Electrical Engineering and Computer Sciences
University of California at Berkeley

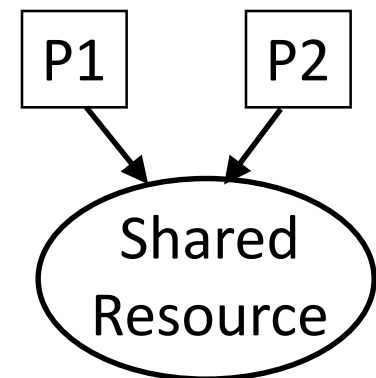
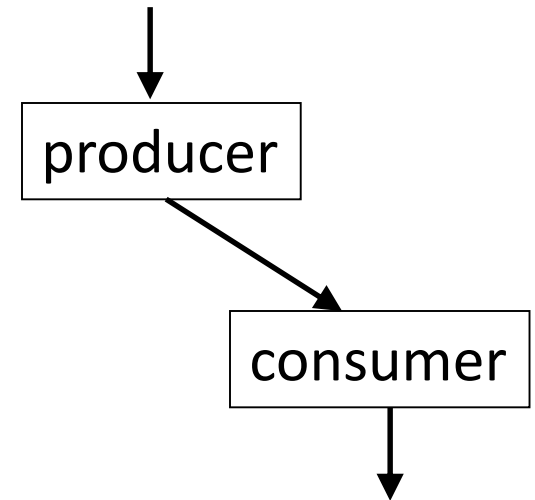
`http://www.eecs.berkeley.edu/~krste`
`http://inst.eecs.berkeley.edu/~cs152`

Synchronization

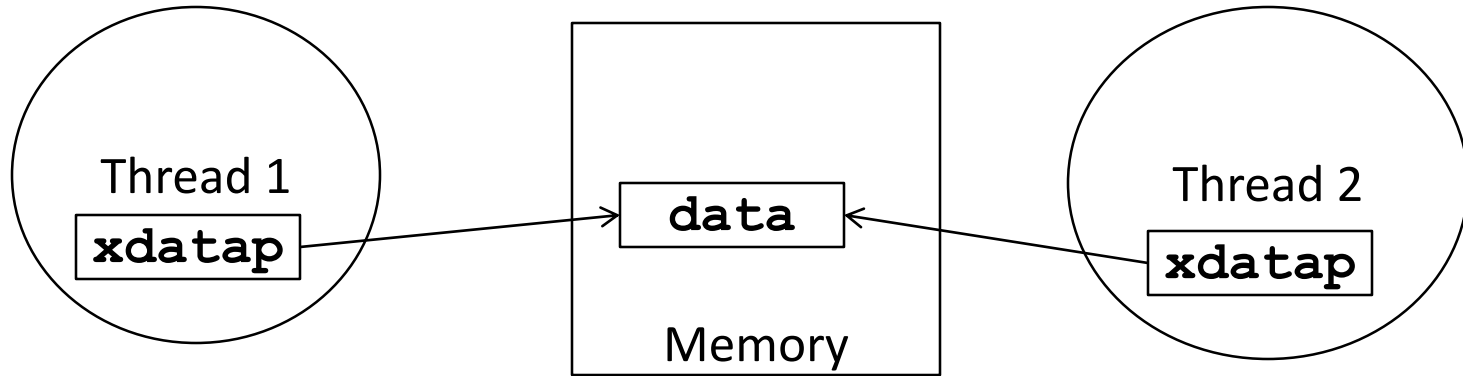
The need for synchronization arises whenever there are concurrent processes in a system (*even in a uniprocessor system*).

Two classes of synchronization:

- *Producer-Consumer*: A consumer process must wait until the producer process has produced data
- *Mutual Exclusion*: Ensure that only one process uses a resource at a given time



Simple Mutual-Exclusion Example



```
// Both threads execute:  
ld xdata, (xdatap)  
add xdata, 1  
sd xdata, (xdatap)
```

Is this correct?

Mutual Exclusion Using Load/Store (assume SC)

A protocol based on two shared variables $c1$ and $c2$.
Initially, both $c1$ and $c2$ are 0 (*not busy*)

Process 1

```
...  
c1=1;  
L: if c2=1 then go to L  
   < critical section >  
c1=0;
```

Process 2

```
...  
c2=1;  
L: if c1=1 then go to L  
   < critical section >  
c2=0;
```

What is wrong? *Deadlock!*

Mutual Exclusion: *second attempt*

To avoid *deadlock*, let a process give up the reservation (i.e. Process 1 sets c_1 to 0) while waiting.

Process 1

```
...  
L:   $c_1=1$ ;  
    if  $c_2=1$  then  
        {  $c_1=0$ ; go to L }  
    < critical section >  
     $c_1=0$ 
```

Process 2

```
...  
L:   $c_2=1$ ;  
    if  $c_1=1$  then  
        {  $c_2=0$ ; go to L }  
    < critical section >  
     $c_2=0$ 
```

- Deadlock is not possible but with a low probability a *livelock* may occur.
- An unlucky process may never get to enter the critical section \Rightarrow *starvation*

A Protocol for Mutual Exclusion

T. Dekker, 1966

A protocol based on 3 shared variables $c1$, $c2$ and $turn$.
Initially, both $c1$ and $c2$ are 0 (*not busy*)

Process 1

```
...  
c1=1;  
turn = 1;  
L: if c2=1 & turn=1  
    then go to L  
    < critical section >  
c1=0;
```

Process 2

```
...  
c2=1;  
turn = 2;  
L: if c1=1 & turn=2  
    then go to L  
    < critical section >  
c2=0;
```

- $turn = i$ ensures that only process i can wait
- variables $c1$ and $c2$ ensure *mutual exclusion*

*Solution for n processes was given by Dijkstra
and is quite tricky!*

Analysis of Dekker's Algorithm

Scenario 1

```
...          Process 1
c1=1;
turn = 1;
L: if c2=1 & turn=1
    then go to L
    < critical section >
c1=0;
```

```
...          Process 2
c2=1;
turn = 2;
L: if c1=1 & turn=2
    then go to L
    < critical section >
c2=0;
```

Scenario 2

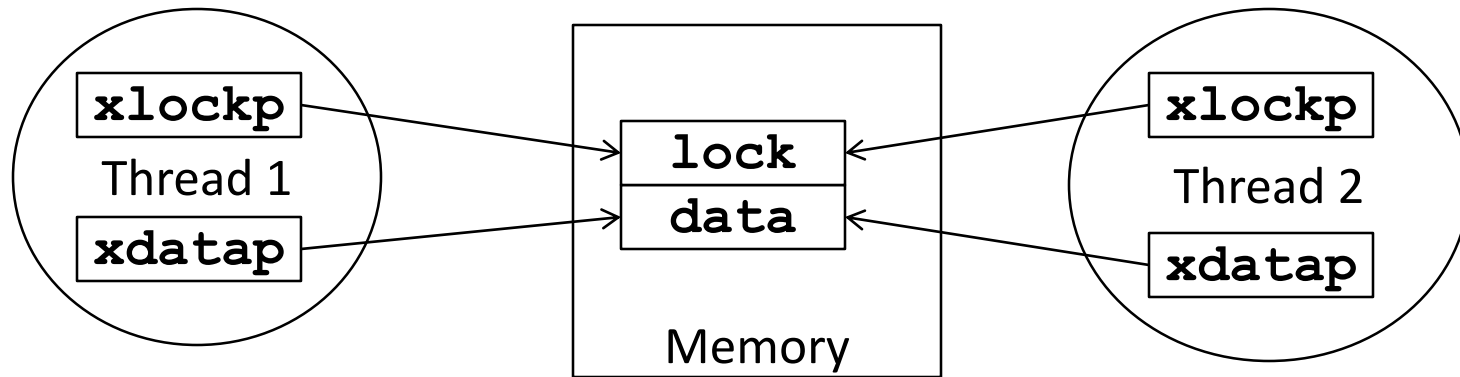
```
...          Process 1
c1=1;
turn = 1;
L: if c2=1 & turn=1
    then go to L
    < critical section >
c1=0;
```

```
...          Process 2
c2=1;
turn = 2;
L: if c1=1 & turn=2
    then go to L
    < critical section >
c2=0;
```

ISA Support for Mutual-Exclusion Locks

- Regular loads and stores in SC model (plus fences in weaker model) sufficient to implement mutual exclusion, but code is inefficient and complex
- Therefore, atomic read-modify-write (RMW) instructions added to ISAs to support mutual exclusion
- Many forms of atomic RMW instruction possible, some simple examples:
 - Test and set ($\text{reg_x} = \text{M}[a]; \text{M}[a]=1$)
 - Swap ($\text{reg_x}=\text{M}[a]; \text{M}[a] = \text{reg_y}$)

Lock for Mutual-Exclusion Example



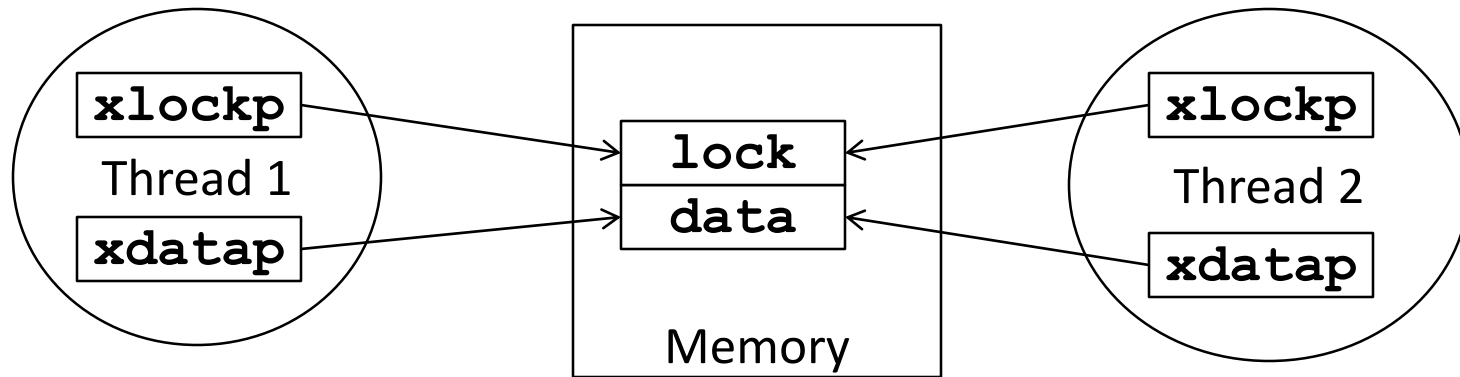
// Both threads execute:

```
li xone, 1
```

spin: amoswap xlock, xone, (xlockp)	Acquire Lock
bnez xlock, spin	
ld xdata, (xdatap)	
add xdata, 1	Critical Section
sd xdata, (xdatap)	
sd x0, (xlockp)	Release Lock

Assumes SC memory model

Lock for Mutual-Exclusion with Relaxed MM



// Both threads execute:

```
li xone, 1
```

```
spin: amoswap xlock, xone, (xlockp)
```

```
    bnez xlock, spin
```

Acquire Lock

```
    fence r,rw
```

```
    ld xdata, (xdatap)
```

```
    add xdata, 1
```

Critical Section

```
    sd xdata, (xdatap)
```

```
    fence rw,w
```

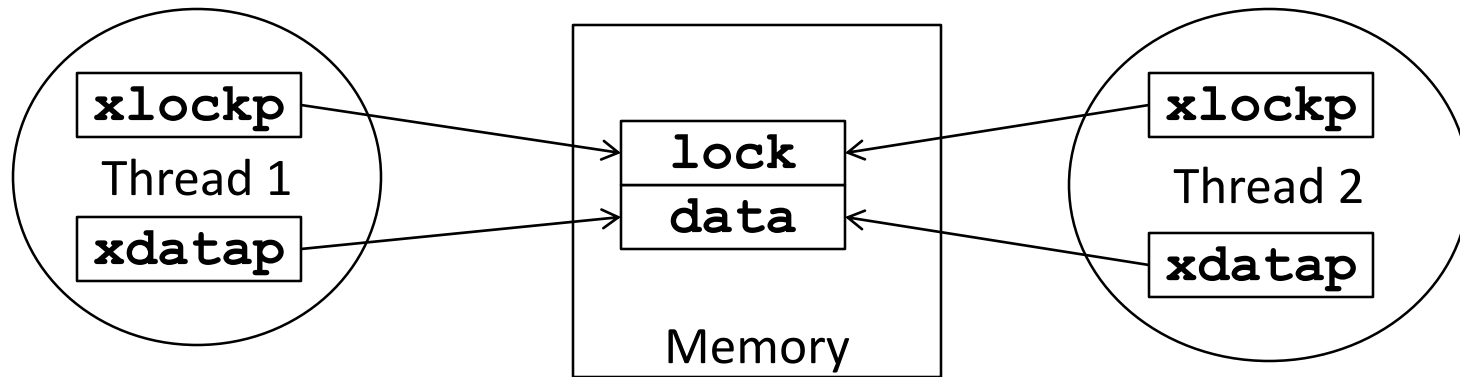
Release Lock

```
    sd x0, (xlockp)
```

RISC-V Atomic Memory Operations

- Atomic Memory Operations (AMOs) have two ordering bits:
 - Acquire (aq)
 - Release (rl)
- If both clear, no additional ordering implied
- If aq set, then AMO “happens before” any following loads or stores
- If rl set, then AMO “happens after” any earlier loads or stores
- If both aq and rl set, then AMO happens in program order

Lock for Mutual-Exclusion using RISC-V AMO



// Both threads execute:

```
li xone, 1
```

```
spin: amoswap.w.aq xlock, xone, (xlockp)
```

```
bnez xlock, spin
```

Acquire Lock

```
ld xdata, (xdatap)
```

```
add xdata, 1
```

```
sd xdata, (xdatap)
```

Critical Section

```
amoswap.w.rl x0, x0, (xlockp)
```


Release Lock

RISC-V FENCE versus AMO.aq/rl

```
sd x1, (a1) # Unrelated store
ld x2, (a2) # Unrelated load
li t0, 1
```

again:

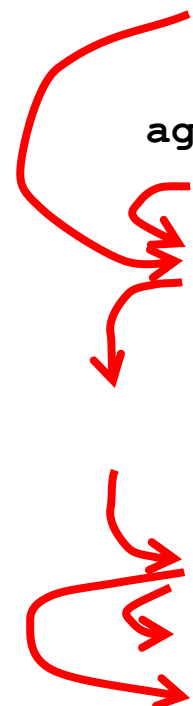
```
amoswap.w.aq t0, t0, (a0)
bnez t0, again
# ...
# critical section
# ...
amoswap.w.rl x0, x0, (a0)
sd x3, (a3) # Unrelated store
ld x4, (a4) # Unrelated load
```



```
sd x1, (a1) # Unrelated store
ld x2, (a2) # Unrelated load
li t0, 1
```

again:

```
amoswap.w t0, t0, (a0)
fence r, rw
bnez t0, again
# ...
# critical section
# ...
fence rw, w
amoswap.w x0, x0, (a0)
sd x3, (a3) # Unrelated store
ld x4, (a4) # Unrelated load
```



AMOs only order the AMO w.r.t. other loads/stores/AMOs

FENCES order every load/store/AMO before/after FENCE

Executing Critical Sections without Locks

- If a software thread is descheduled after taking lock, other threads cannot make progress inside critical section
- “Non-blocking” synchronization allows critical sections to execute atomically without taking a lock

Nonblocking Synchronization

```
Compare&Swap(m), Rt, Rs:  
  if (Rt==M[m])  
    then M[m]=Rs;  
        Rs=Rt;  
        status ← success;  
  else status ← fail;
```

status is an
implicit
argument

```
try: Load Rhead, (head)  
spin: Load Rtail, (tail)  
      if Rhead==Rtail goto spin  
      Load R, (Rhead)  
      Rnewhead = Rhead+1  
      Compare&Swap(head), Rhead, Rnewhead  
      if (status==fail) goto try  
      process(R)
```

Compare-and-Swap Issues

- Compare and Swap is a complex instruction
 - Three source operands: address, comparand, new value
 - One return value: success/fail or old value
- ABA problem
 - Load(A), Y=process(A), success=CAS(A,Y)
 - What if different task switched A to B then back to A before process() finished?
- Solving ABA:
 - Add a counter, and make CAS access two words:
- Double Compare and Swap (DCAS)
 - Five source operands: one address, two comparands, two values
 - Load(<A1,A2>), Z=process(A1), success=CAS(<A1,A2>,<Y,A2+1>)

Load-reserve & Store-conditional

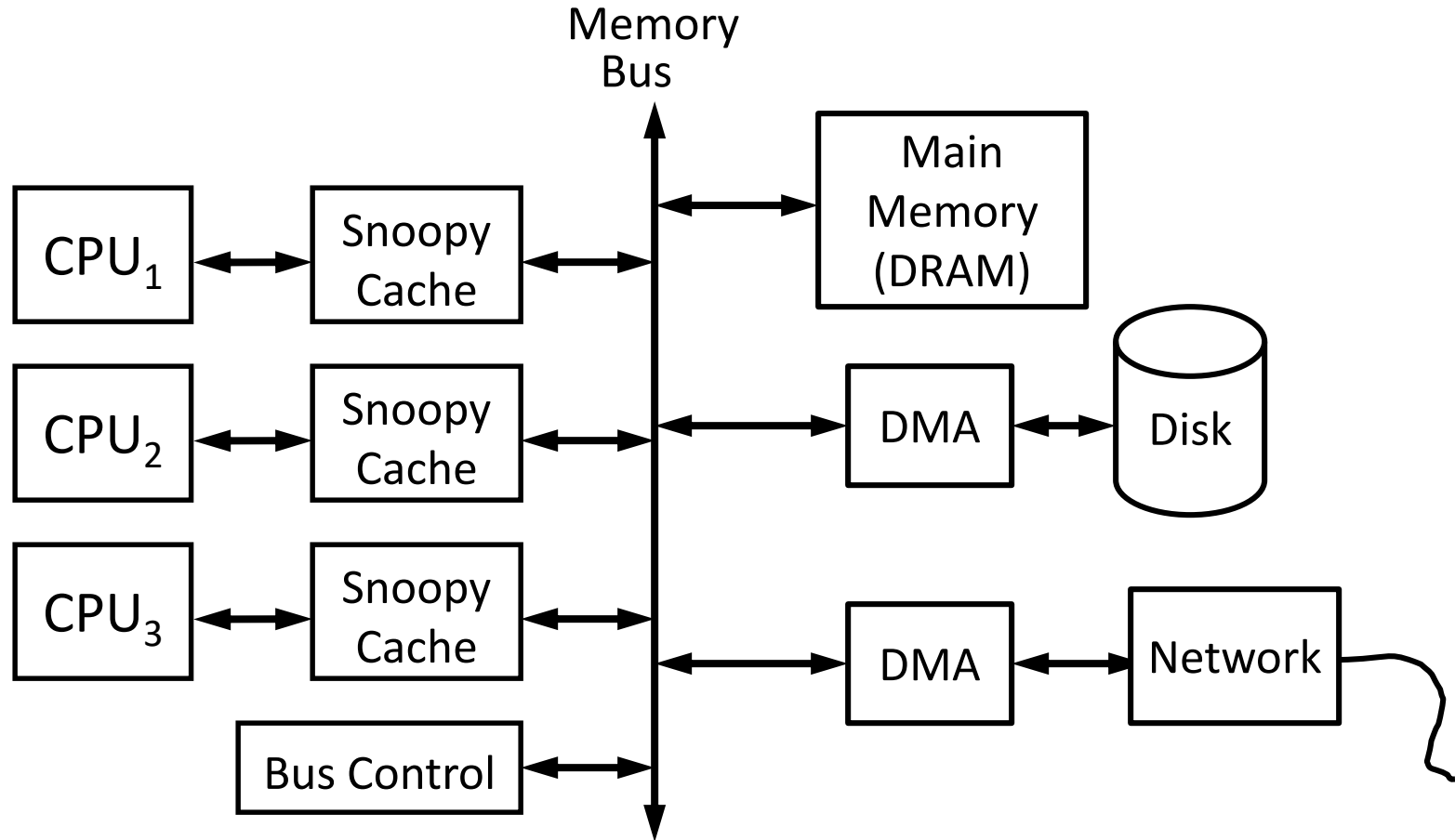
Special register(s) to hold reservation flag and address,
and the outcome of store-conditional

```
Load-reserve R, (m):  
    <flag, adr>  $\leftarrow$  <1, m>;  
    R  $\leftarrow$  M[m];
```

```
Store-conditional (m), R:  
    if <flag, adr> == <1, m>  
    then cancel other procs'  
        reservation on m;  
        M[m]  $\leftarrow$  R;  
        status  $\leftarrow$  succeed;  
    else status  $\leftarrow$  fail;
```

```
try: Load-reserve Rhead, (head)  
spin: Load Rtail, (tail)  
      if Rhead == Rtail goto spin  
      Load R, (Rhead)  
      Rhead = Rhead + 1  
      Store-conditional (head), Rhead  
      if (status == fail) goto try  
      process(R)
```

Load-Reserved/Store-Conditional using MESI Caches



Load-Reserved ensures line in cache in Exclusive/Modified state
Store-Conditional succeeds if line still in Exclusive/Modified state
(In practice, this implementation only works for smaller systems)

LR/SC Issues

- LR/SC does not suffer from ABA problem, as any access to addresses will clear reservation regardless of value
 - CAS only checks stored values not intervening accesses
- LR/SC non-blocking synchronization can livelock between two competing processors
 - CAS guaranteed to make forward progress, as CAS only fails if some other thread succeeds
- RISC-V LR/SC makes guarantee of forward progress provided code inside LR/SC pair obeys certain rules
 - Can implement CAS inside RISC-V LR/SC

RISC-V Atomic Instructions

- Non-blocking “Fetch-and-op” with guaranteed forward progress for simple operations, returns original memory value in register
- AMOSWAP $M[a] = d$
- AMOADD $M[a] += d$
- AMOAND $M[a] \&= d$
- AMOOR $M[a] |= d$
- AMOXOR $M[a] \wedge= d$
- AMOMAX $M[a] = \max(M[a], d)$ *# also, unsigned AMOMAXU*
- AMOMIN $M[a] = \min(M[a], d)$ *# also, unsigned AMOMINU*

Transactional Memory

- Proposal from Knight ['80s], and Herlihy and Moss ['93]

XBEGIN

MEM-OP1

MEM-OP2

MEM-OP3

XEND

- Operations between XBEGIN instruction and XEND instruction either all succeed or are all squashed
- Access by another thread to same addresses, cause transaction to be squashed
- More flexible than CAS or LR/SC
- Commercially deployed on IBM POWER8 and Intel TSX extension, ARM announced TME

Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
 - Arvind (MIT)
 - Joel Emer (Intel/MIT)
 - James Hoe (CMU)
 - John Kubiatowicz (UCB)
 - David Patterson (UCB)