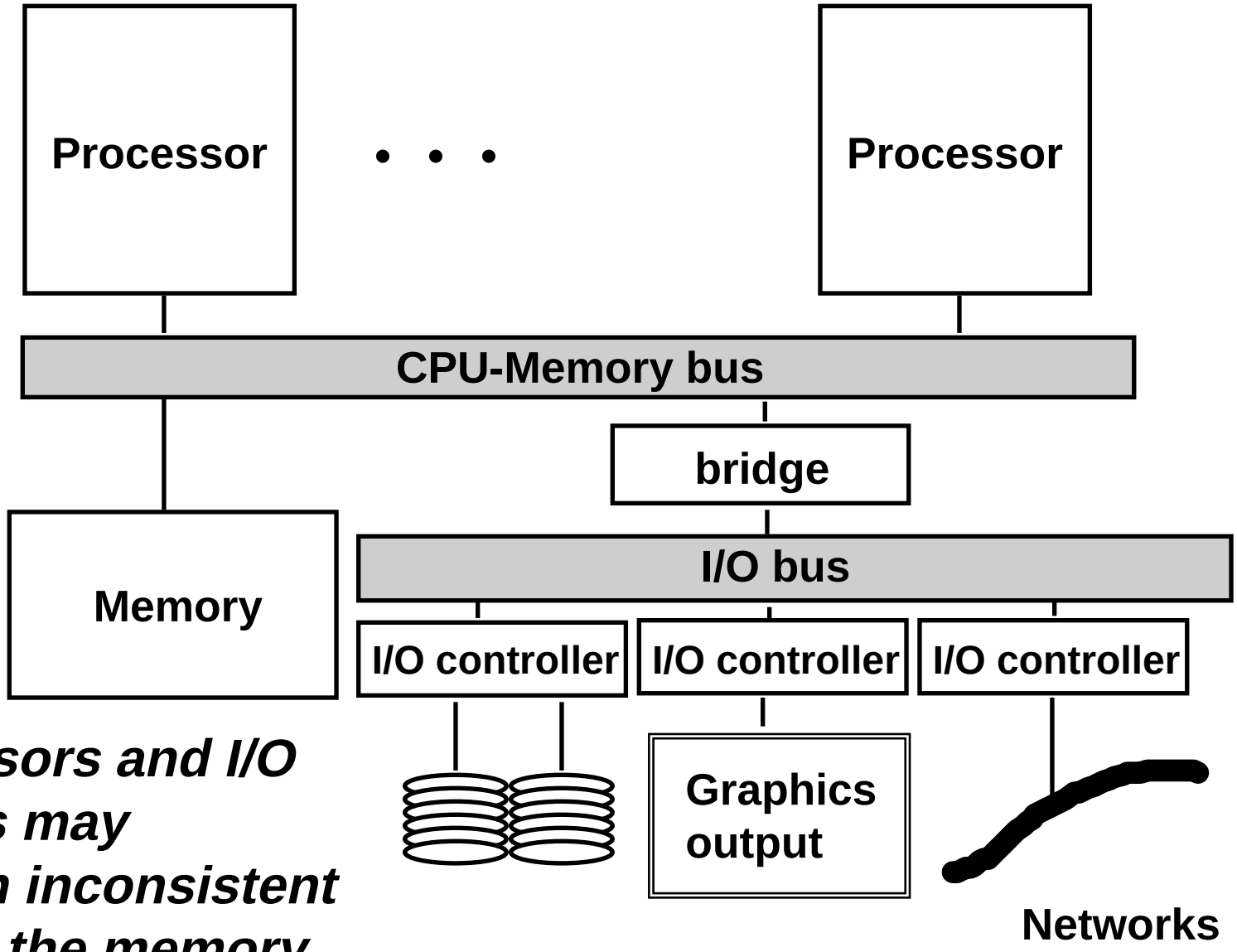


# Cache Coherence Protocols: Implementation Issues on SMP's

**Krste Asanovic**
**Laboratory for Computer Science**
**M.I.T.**

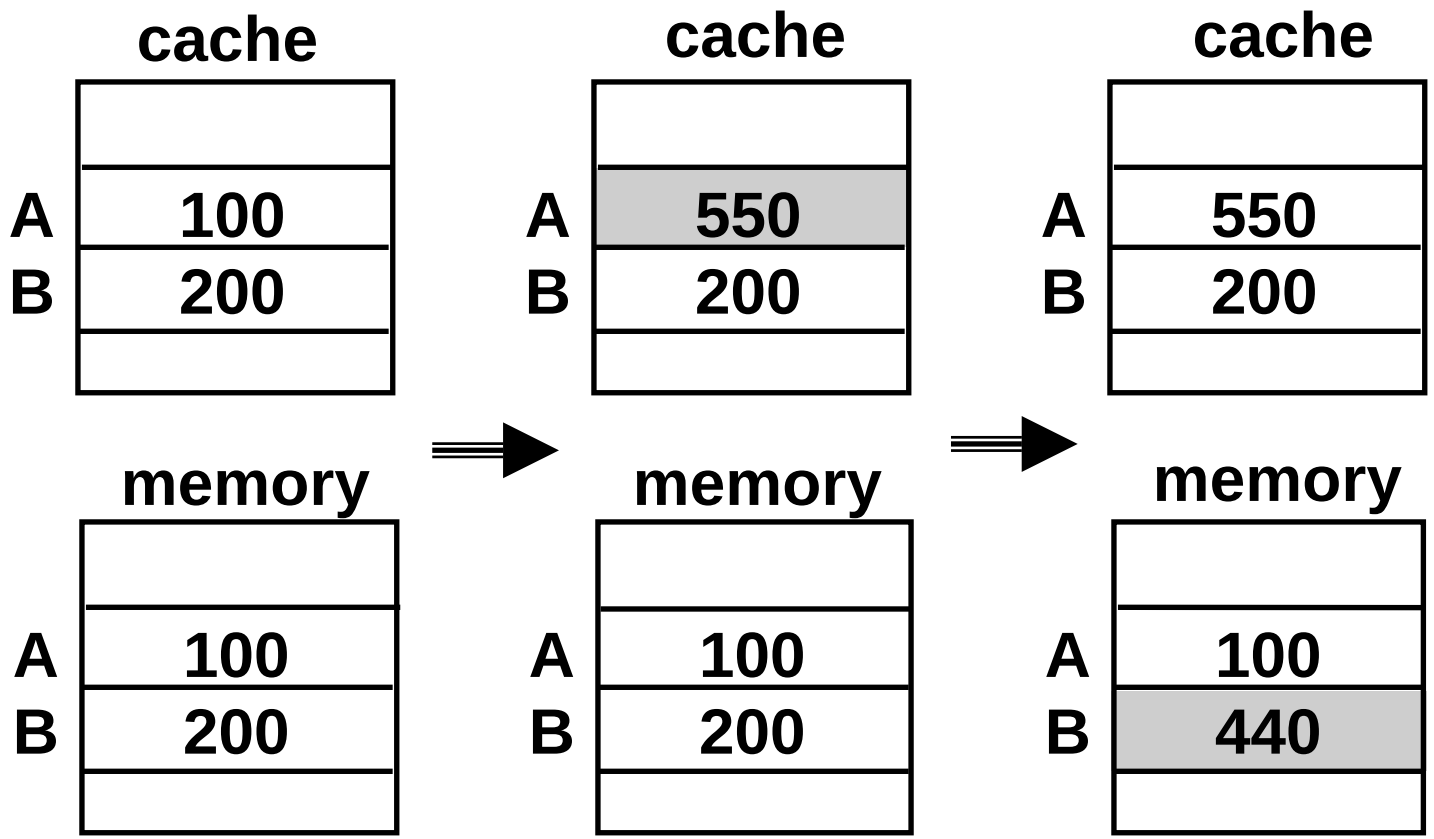
**http://www.csg.lcs.mit.edu/6.823**

# Cache Coherence Issue in I/O




***Processors and I/O devices may have an inconsistent view of the memory because of caches***

# DMA & Cache Coherence Problem




1. Processor cache and memory are coherent
2. CPU writes 550 in A, I/O may read A and get 100 *(stale)*
3. DMA writes 440 in B, CPU may read B and get 200 *(stale)*



# Possible Solutions

***Bypass caches:***

- **make all I/O reads and writes go to the memory**

***Write-through caches:***

- **follow write-through policy and initiate the I/O transfer after all the writes are complete**

  ***does not solve the I/O-to-CPU problem***

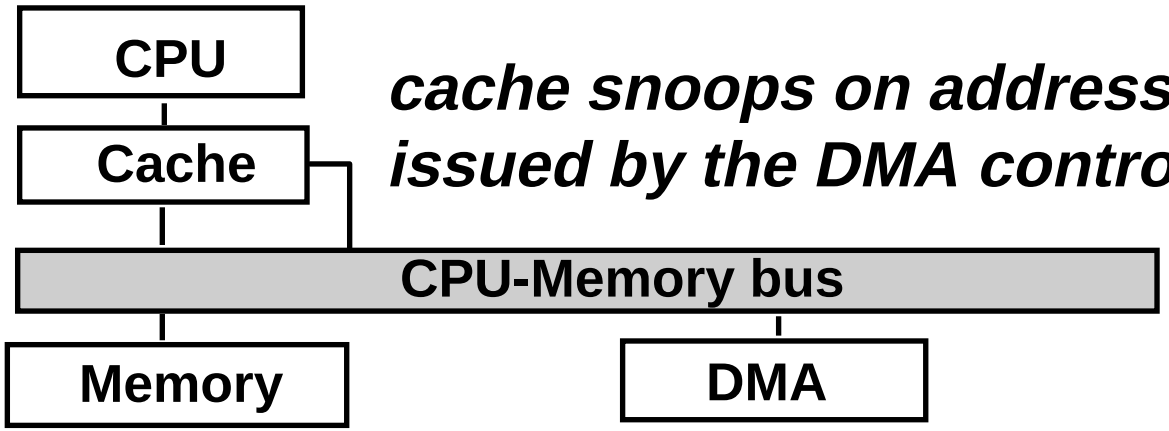
***Cache flush or forced writeback:***

- **flush cache blocks selectively - tricky & expensive**
- **flush the whole cache - increased cache-miss rate**

**Many processors provide some of these controls over caches through the ISA (usually in the privileged mode)**

# A Transparent Solution for I/O Cache Coherence






***cache snoops on addresses issued by the DMA controller***

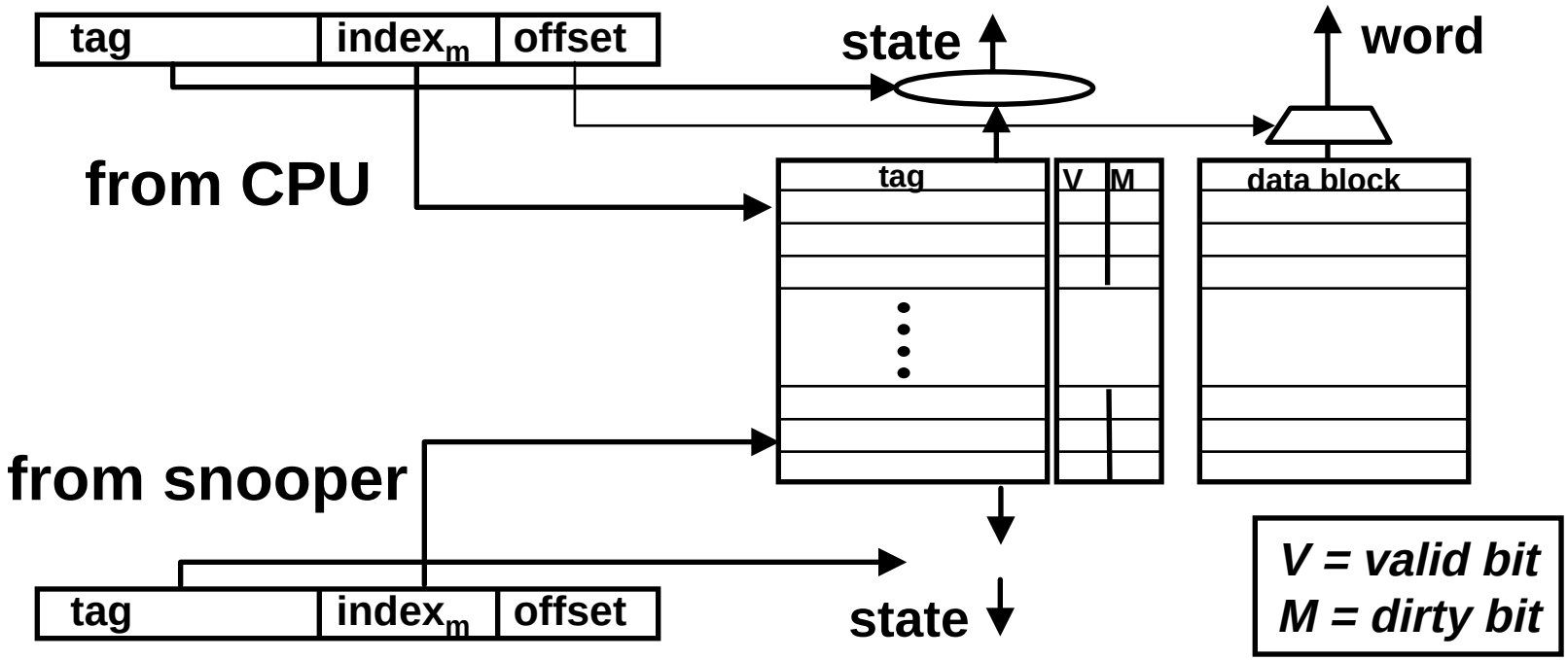
**Suppose DMA issues an address which is in the cache**

***write request:*** **the address is flushed or updated in the cache before the write is performed in the memory**

***read request:*** **a write-back is performed before the memory is read (not needed if the CPU is following a write-through policy)**

***Software has to ensure that there are no read-write races between the CPU and DMA***

# Snooping Cache




- **Snooper needs to query the cache on every external memory transactions, but rarely needs to access the data**
- ***Dual ported Cache*** **(<Tag, V, M> only) to allow independent accesses from the CPU and the snooper**
- ***Interlocking*** **is required if either port needs to change the status bits, i.e., *updates must be serialized***



# Maintaining Sequential Consistency

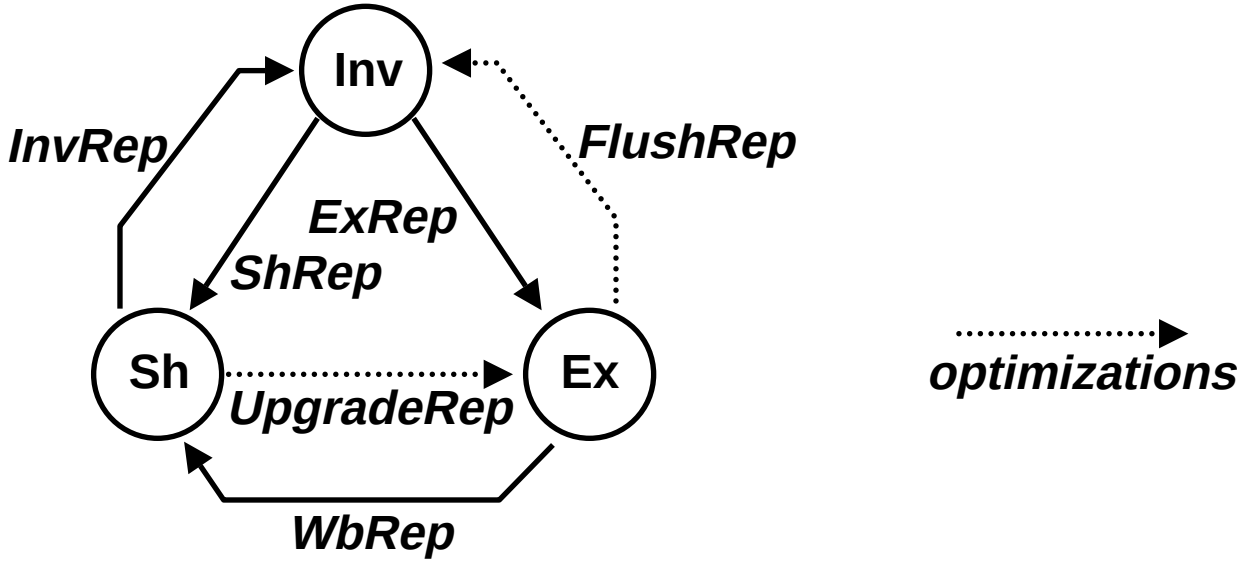
**Hardware support is required such that**

- **only one processor at a time has write permission for a location**
- **no processor can load a stale copy of the location after a write**

$\Rightarrow$ ***cache coherence protocols***



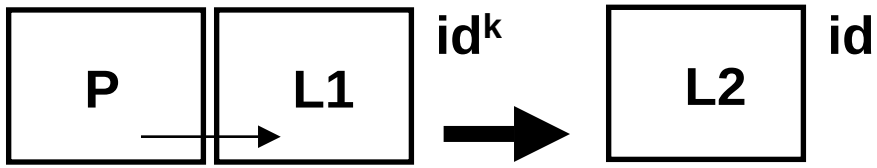
# Cache State Transitions




**What causes a state transition ?**



# Issuing Requests




**Load (a):**

**If a is not in L1, send a Sh request (ShReq) to L2**

**Store (a,v):**

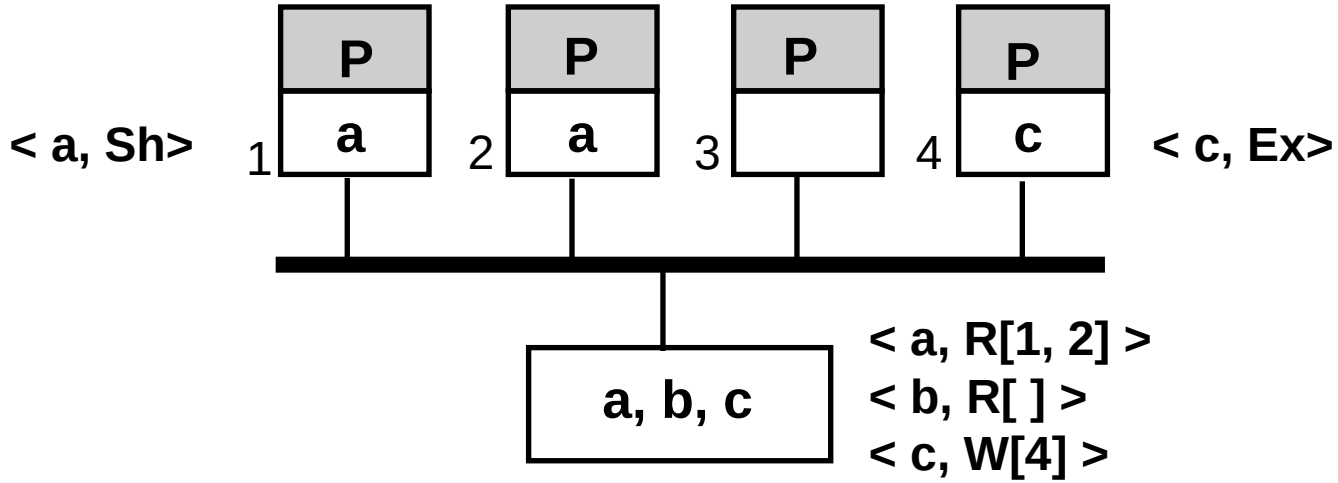
**If a is not in Ex state in L1, send a Ex request (ExReq) to L2**

**In both cases the cache state is set to a transient state (CachePending)** ***and the instruction remains suspended.***

***A complete protocol is included in the slides of Lecture 20***



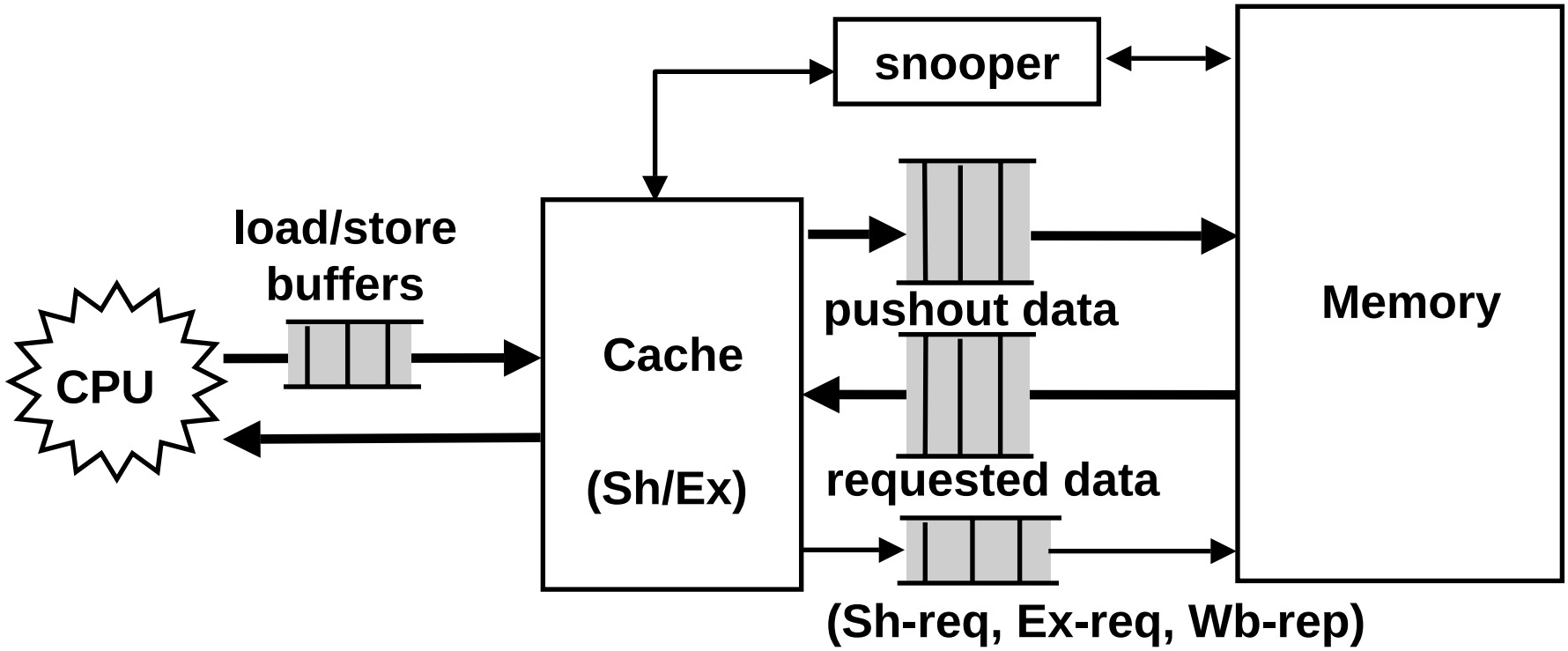
# Simplification of Cache States Due to Two Levels




- **Uniprocessor cache is extended with Sh/Ex bit**
- **In a bus based system, it may be more efficient to broadcast the request directly to *all* caches and then collect their responses**

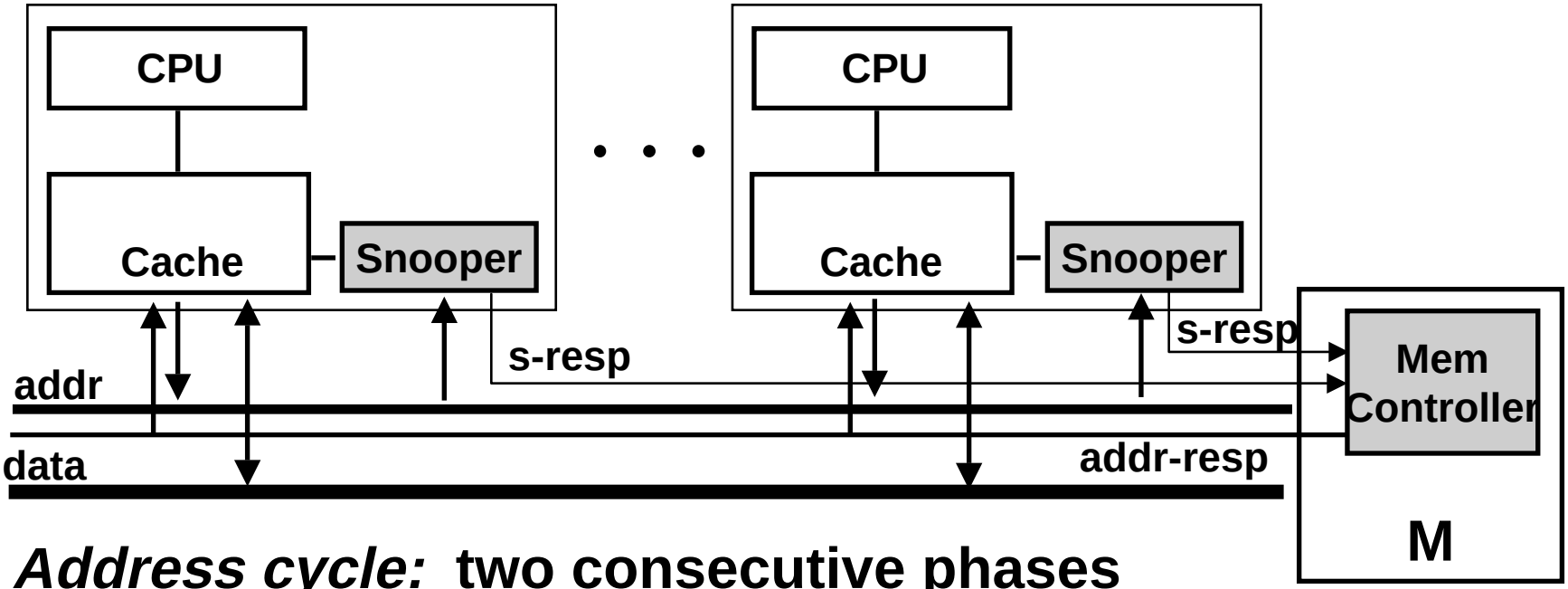
  **⇒ *eliminates the need for home directory***

# Processor-Memory Interface




**Address and data cycles are distinct**

# Bus: A Broadcast Medium




***Address cycle:*** **two consecutive phases**

- ***request phase:*** **a processor is selected to issue a request which is assigned a bus tag**
- ***response phase:*** **summary of responses from all the snoopers is returned to the requesting processor**

***Data cycle (if necessary):*** **the data and its bus tag appear on the data bus**

**The bus tag is retired when the transaction terminates**



# Snooper's Input & Output

***L1 & Snooper State***

**< id, cache, out, obt>**

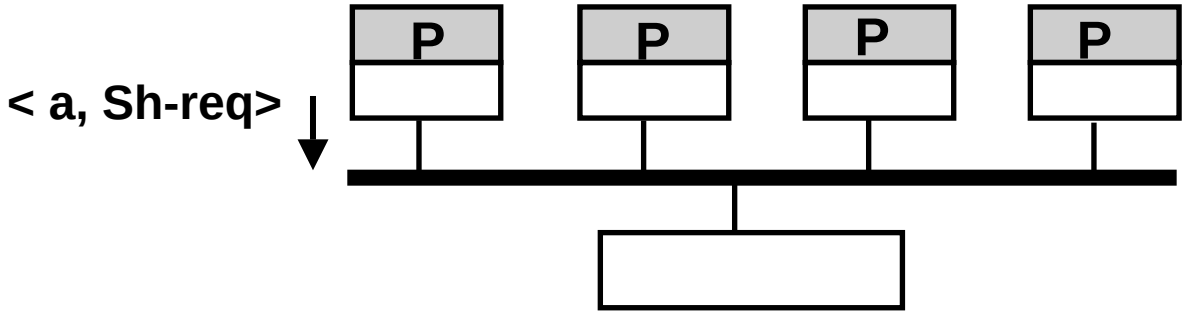
**Outstanding bus transactions**
**a set of <btag, a>**

***Needed to capture the data during a data cycle***

**<Sh-req, a>**
**<Ex-req, a>**
**<Wb, a, v>**
**<Inv, a>**

- **When L1 gets control of the bus, one message from out is assigned the tag and put on the bus**
- **<btag, Wb, a, v> type transactions only affects M**
- **<btag, Sh-req, a> and <btag, Ex-req, a> types of transactions are input to all other Snoopers**
  **Each Snooper responds *ok* or *retry***
- **MC summarizes s-resp's into *unanimous-ok* or *retry***

# Snooper's Response: *Sh-req*




**< id, cache, out, obt>**

***if*** **a ∉ cache *and* <Wb, a, - > ∉ out**

**→ < id, cache, out, obt>, ok**

**< id, Cell(a,v,Sh)|cache, out, obt>** ***if*** **<Wb, a, - > ∉ out**

**→ < id, Cell(a,v,Sh)|cache, out, obt>, ok**

**< id, Cell(a,v,Ex)|cache, out, obt>**

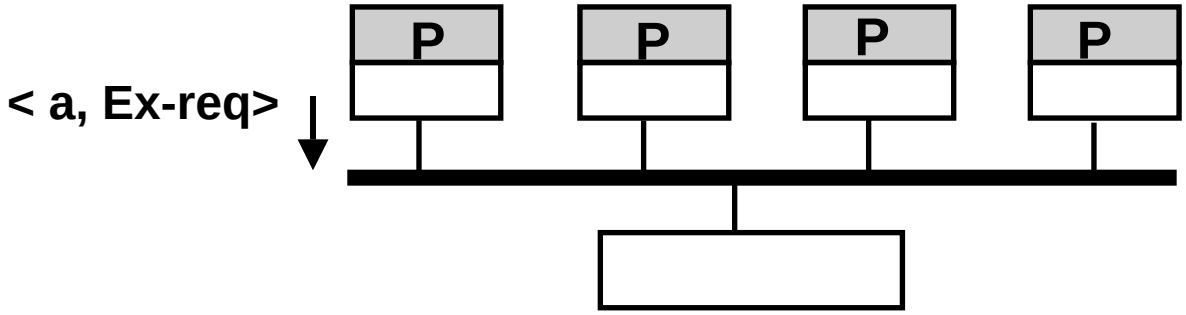
**→ < id, Cell(a,v,Sh)|cache, out | (Wb, a, v), obt > , retry**

**< id, cache, <Wb, a, - >|out, obt>**

**→ < id, cache, <Wb, a, - >|out, obt> , retry**



# Snooper's Response: *Ex-req*




**< id, cache, out, obt>**

**$\quad$ *if* a $\notin$ cache *and* <Wb, a, - > $\notin$ out**

**$\rightarrow$ < id, cache, out, obt>, ok**

**< id, Cell(a,v,Sh)|cache, out, obt> $\quad$ *if* <Wb, a, - > $\notin$ out**

**$\rightarrow$ < id, cache, out, obt>, ok**

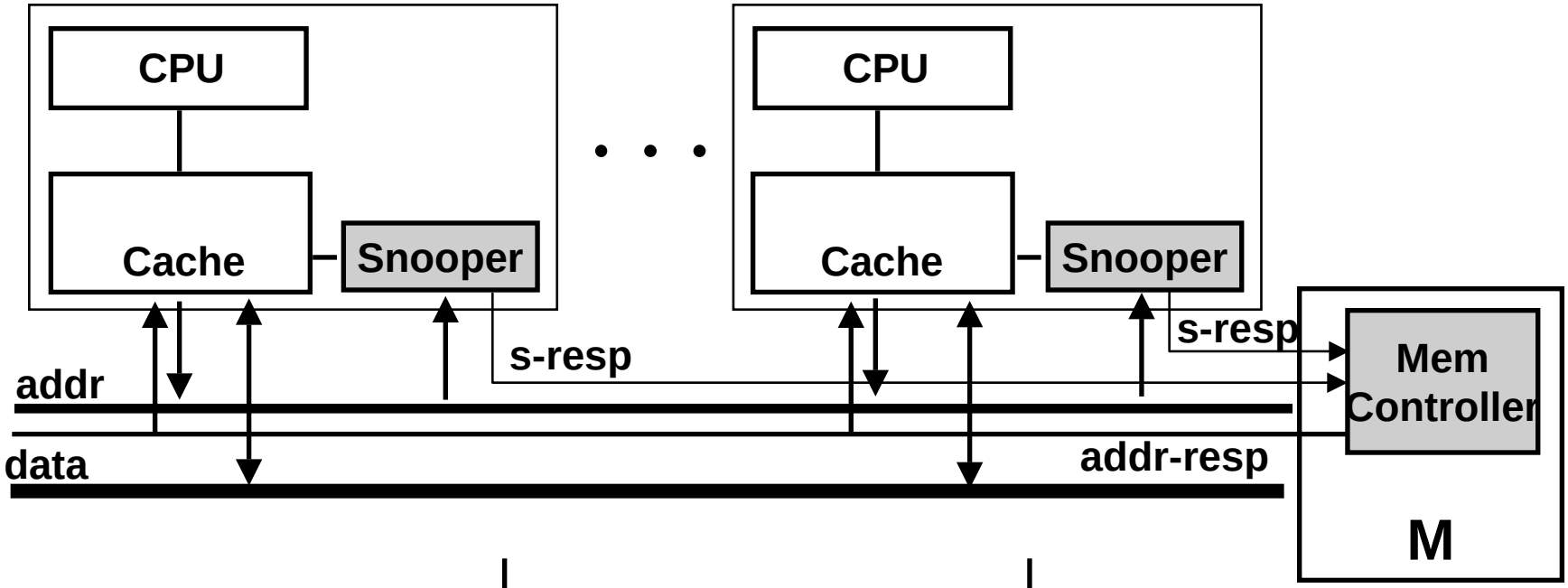
**< id, Cell(a,v,Ex)|cache, out, obt>**

**$\rightarrow$ < id, Cell(a,v,Sh)|cache, out|(Wb, a, v), obt >, retry**

**< id, cache, <Wb, a, - >|out, obt>**

**$\rightarrow$ < id, cache, <Wb, a, - >|out, obt>, retry**

# Memory Controller Response




| Addr-Request | Addr-Response | Data |
|---|---|---|
| <tag,Sh-req,a> | *retry* | |
| | *u-ok* | <tag,Sh-rep,a,data> |
| <tag,Ex-req,a> | *retry* | |
| | *u-ok* | <tag,Ex-rep,a,data> |
| <tag,Wb,a> | *u-ok* | <tag,Wb,a,data> |



# Effect of Address Response on the Bus Master

***Address Bus transaction*** **<tag, type, a>**

**Unanimous-ok**

**< id, cache, <type, a >;out, obt>**

**→ < id, cache, out, obt |<tag, type, a > >**

**Retry**

**< id, cache, <type, a >;out, obt>**

**→ < id, cache, out;<type, a > , obt >**

***Data Bus transaction:***

**<tag, Sh-rep, a, v>**

**< id, cache, out, obt |<tag, Sh-req, a > >**

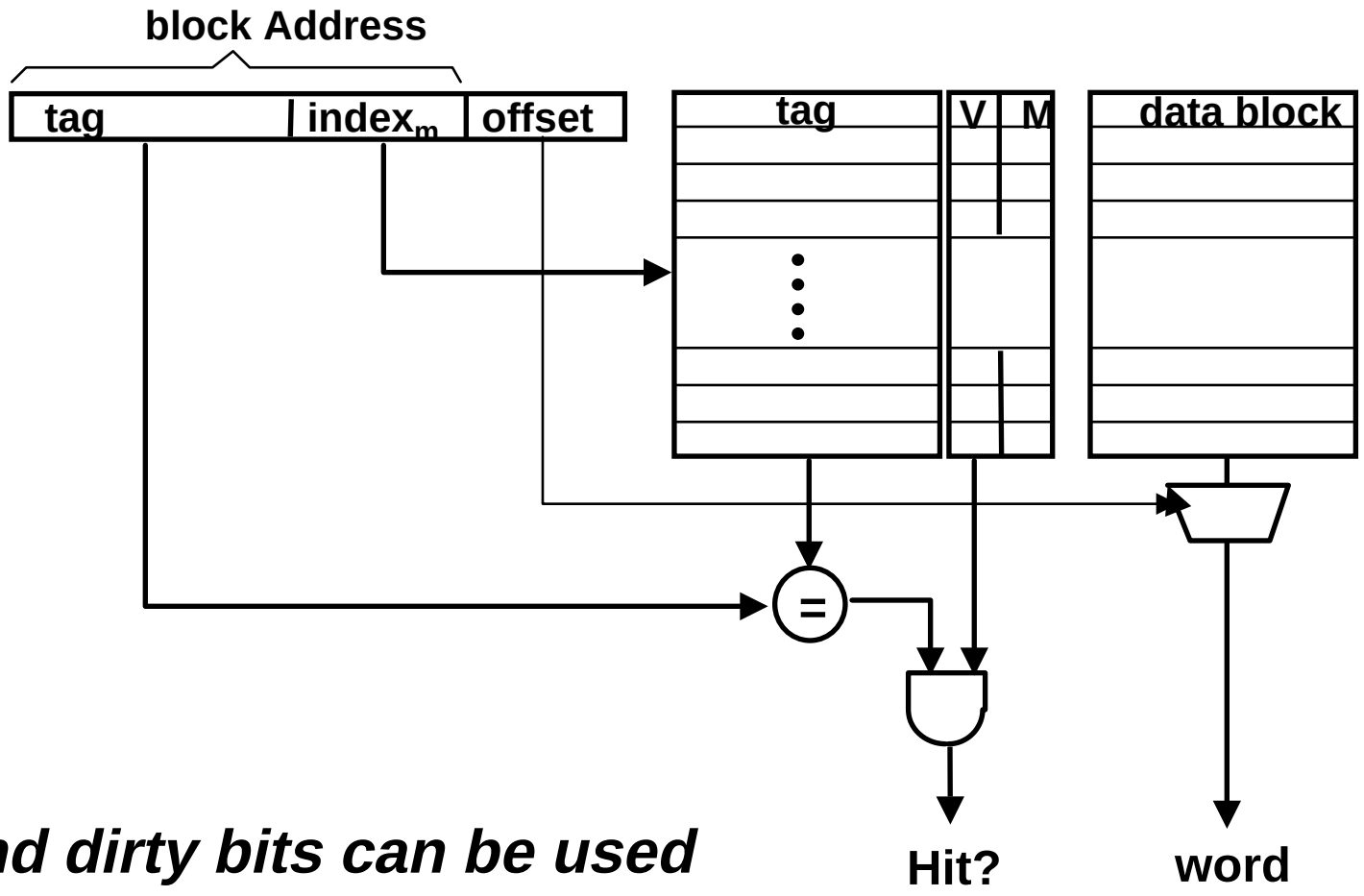
**→ < id, Cell(a,v,Sh)|cache, out, obt >**

**<tag, Ex-rep, a, v>**

**< id, cache, out, obt |<tag, Ex-req, a > >**

**→ < id, Cell(a,v,Ex)|cache, out, obt >**

# CC State Encoding




***Valid and dirty bits can be used to encode Sh and Ex states***

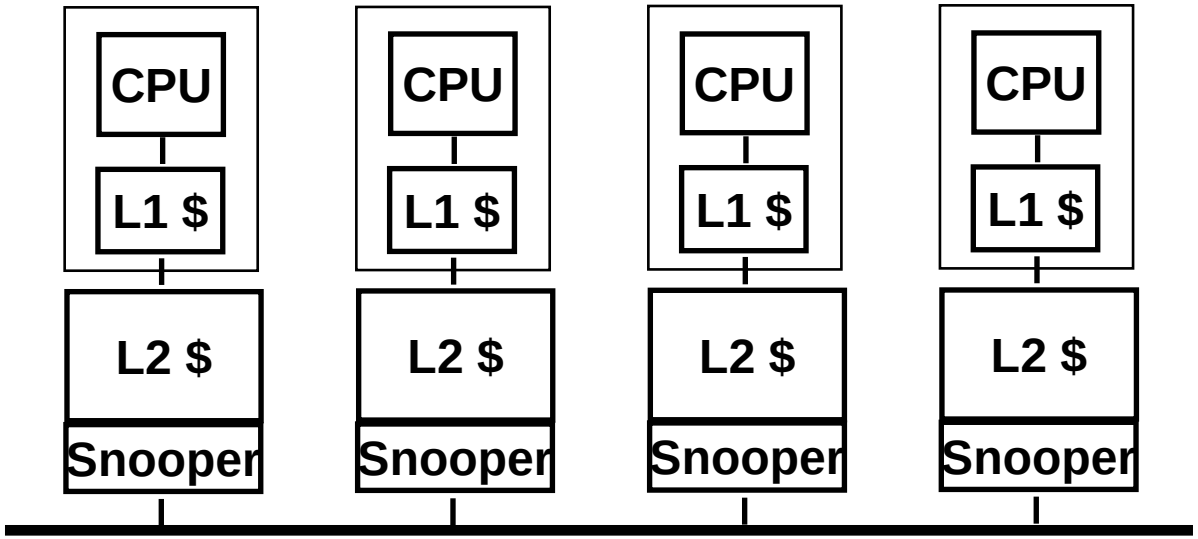
**V=0, M=x ⇒ Invalid**

**V=1, M=0 ⇒ Shared *(not dirty)***

**V=1, M=1 ⇒ Exclusive *(dirty)***



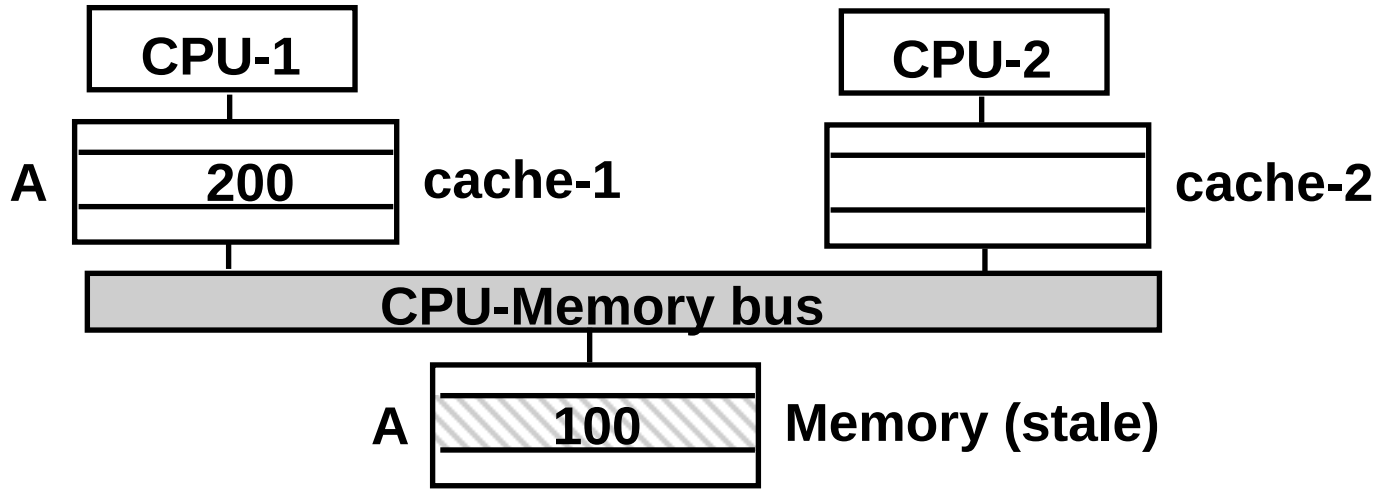
# 2-Level Caches




- Processors often have two-level caches
  - L1 is small and on chip
  - L2 is large and off chip
- *inclusion property:* entries in L1 must be present in L2
  invalidation in L2 $\Rightarrow$ invalidation in L1
- Snooping on L2 does not affect the CPU-L1 bandwidth
- Interlocks are required when both CPU-L1 and L2-Bus interactions involve the same address



# Intervention




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

- **cache-1 needs to supply the data and change its state to shared**
- **the memory may respond to the request also!**
  ***does memory know it has stale data?***

**An intervention by cache-1 through memory controller is needed to supply the correct data to cache-2**



# False Sharing

| state | blk addr | data0 | data1 | ... | datan |
|---|---|---|---|---|---|

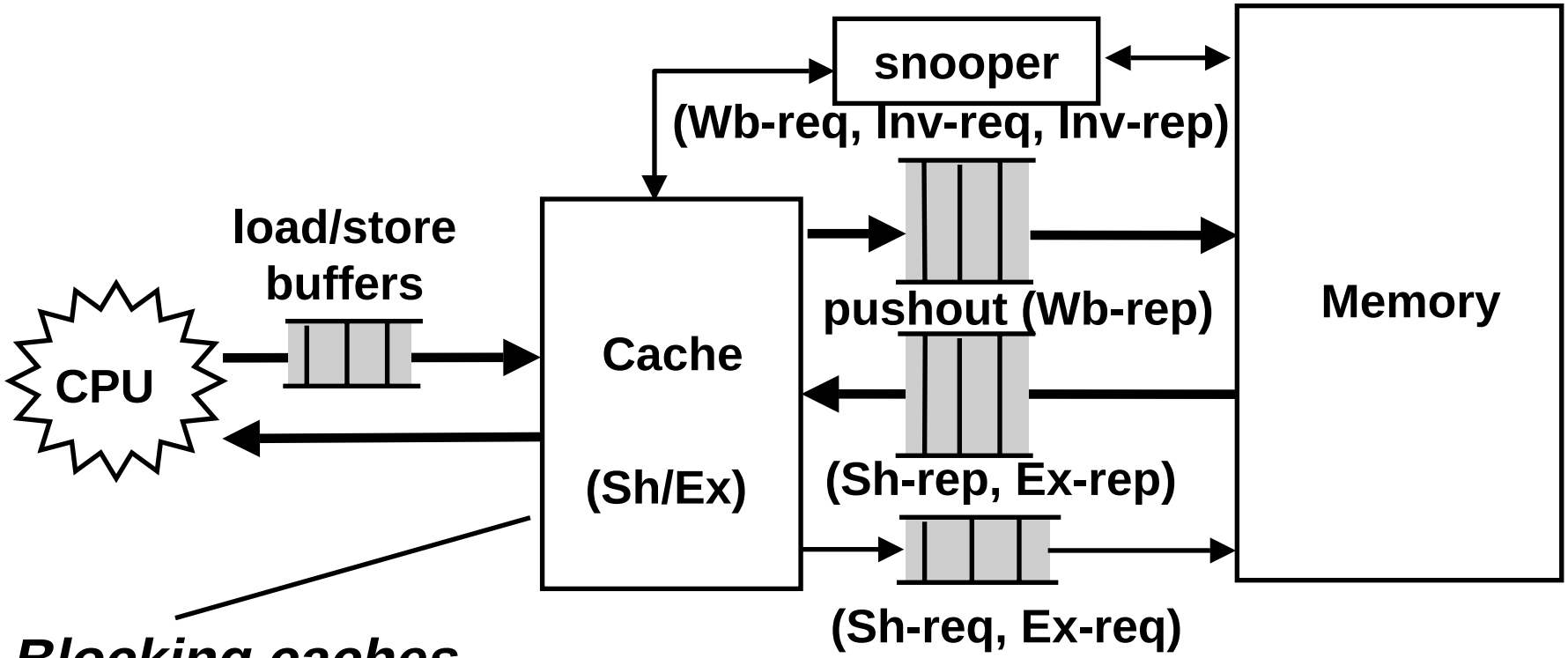
A cache block contains more than one word

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

Suppose P1 writes $word_i$ and P2 writes $word_k$ and both words have the same block address.

The block may be invalidated many times unnecessarily because of the addresses share a common block

# Out-of-Order Loads/Stores & CC




***Blocking caches***

One request at a time + CC $\Rightarrow$ SC

***Non-blocking caches***

Multiple requests (different addresses) concurrently

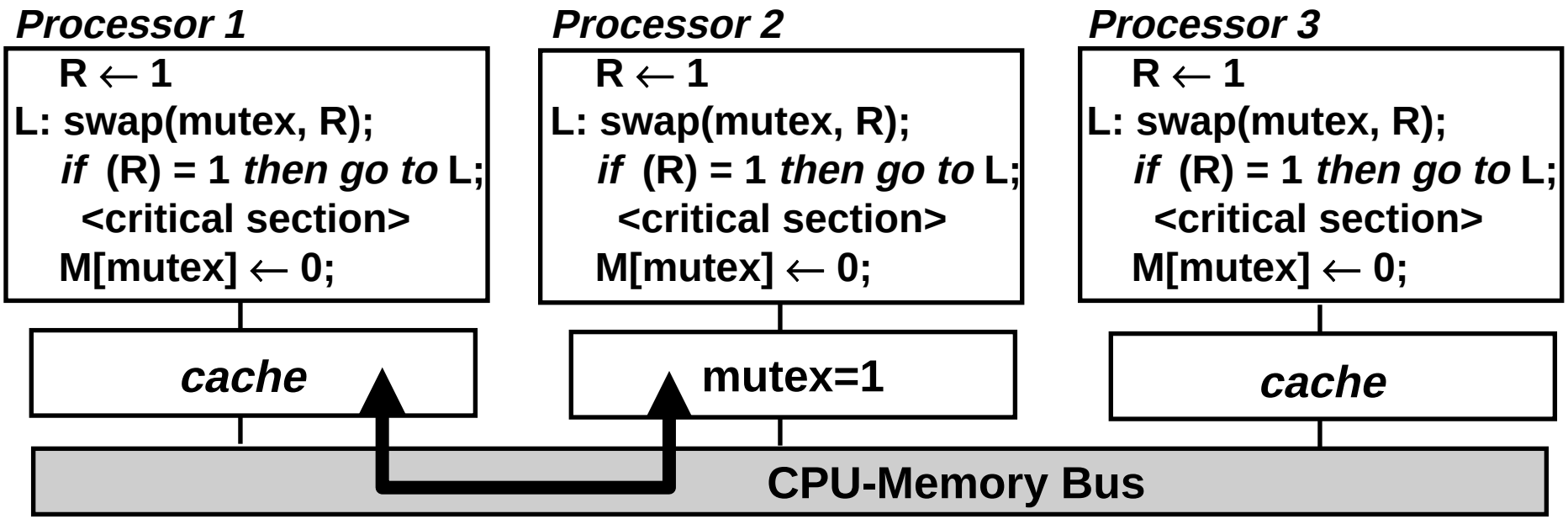
+ CC $\Rightarrow$ Relaxed memory models

CC ensures that all processors observe the same order of loads and stores to an address



# Synchronization and Caches:

## *Performance Issues*




**Cache-coherence protocols will cause mutex to *ping-pong* between P1's and P2's caches.**

**Ping-ponging can be reduced by first reading the mutex location *(non-atomically)* and executing a swap only if it is found to be zero.**

# Performance Issues Related to Bus occupancy



**In general, a *read-modify-write* instruction requires two memory (bus) operations without intervening memory operations by other processors**

**In a multiprocessor setting, bus needs to be locked for the entire duration of the atomic read and write operation**

**⇒ expensive for simple buses**
**⇒ *very expensive* for split-phase buses**

**modern processors use**
***load-reserve***
***store-conditional***

# Load-reserve & Store-conditional



**Special register(s) to hold < reserve bit, address>, and a status bit for the outcome of store-conditional**

**load-reserve(m,R):**
**< reserve, address> ← < 1, m >;**
**R ← M[m];**

**store-conditional(m,R):**
***if* < reserve, address> = < 1, m >**
***then* cancel other processors' reservation on m;**
**M[m] ← (R); status ← succeed;**
***else* status ← fail;**

- ***If the snooper sees a store transaction to the address in the reserve register, the reserve bit is set to 0***
- ***Several processors may reserve m simultaneously***
- ***These instructions are like ordinary loads and stores with respect to the bus traffic***



# Performance:
## *load-reserve & store-conditional*

**The total number of memory (bus) transactions is not necessarily reduced, but splitting an atomic instruction into load-reserve & store-conditional:**

- ***increases bus utilization*** **(and reduces processor stall time), especially in split-phased buses**
- ***reduces cache ping-pong effect*** **because processors trying to acquire a semaphore do not have to perform a store each time**