Directory-Based Cache Coherence

Mengjia Yan
Computer Science and Artificial Intelligence Lab
M.I.T.

Maintaining Cache Coherence

It is sufficient to have hardware 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

⇒ A correct approach could be:

write request:

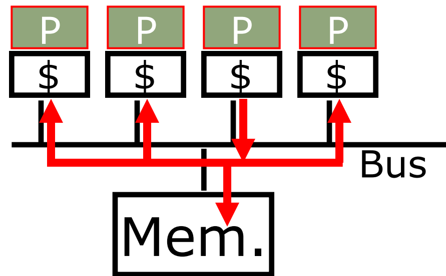
The address is *invalidated* in all other caches *before* the write is performed

read request:

If a dirty copy is found in some cache, a write-back is performed before the memory is read

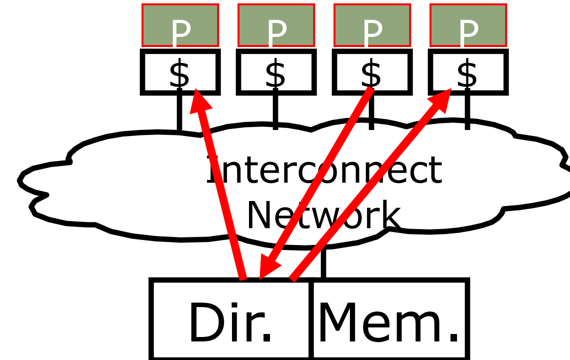
Directory-Based Coherence
[Censier and Feautrier, 1978]

Snoopy Protocols



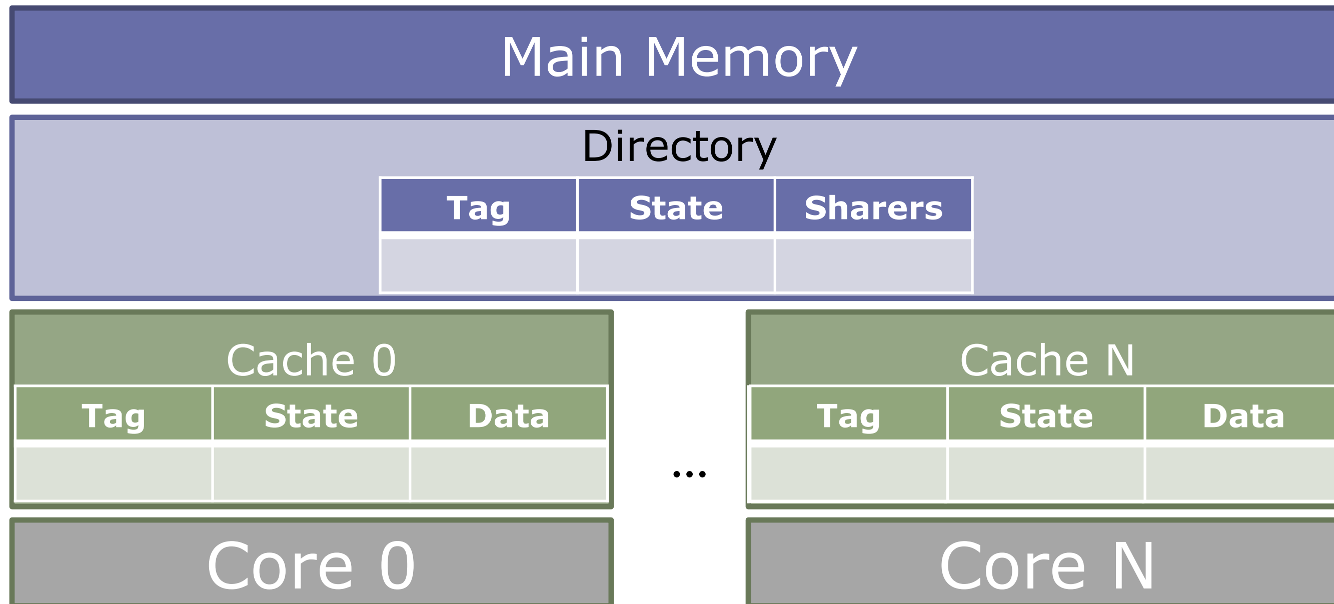
- Snoopy schemes broadcast requests over memory bus
- Difficult to scale to large numbers of processors
- Requires additional bandwidth to cache tags for snoop requests

Directory Protocols



- Directory schemes send messages to only those caches that might have the line
- Can scale to large numbers of processors
- Requires extra directory storage to track possible sharers

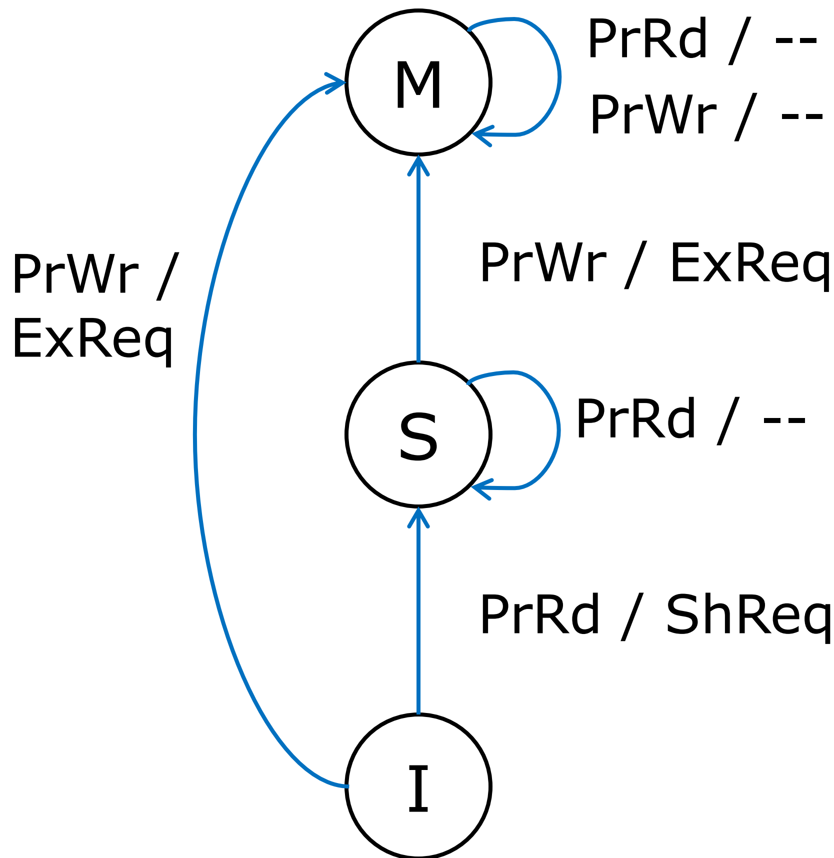
An MSI Directory Protocol



- Cache states: Modified (M) / Shared (S) / Invalid (I)
- Directory states:
  - Uncached (Un): No sharers
  - Shared (Sh): One or more sharers with read permission (S)
  - Exclusive (Ex): A single sharer with read & write permissions (M)
- Transient states not drawn for clarity; for now, assume no racing requests

MSI Protocol: Caches (1/3)

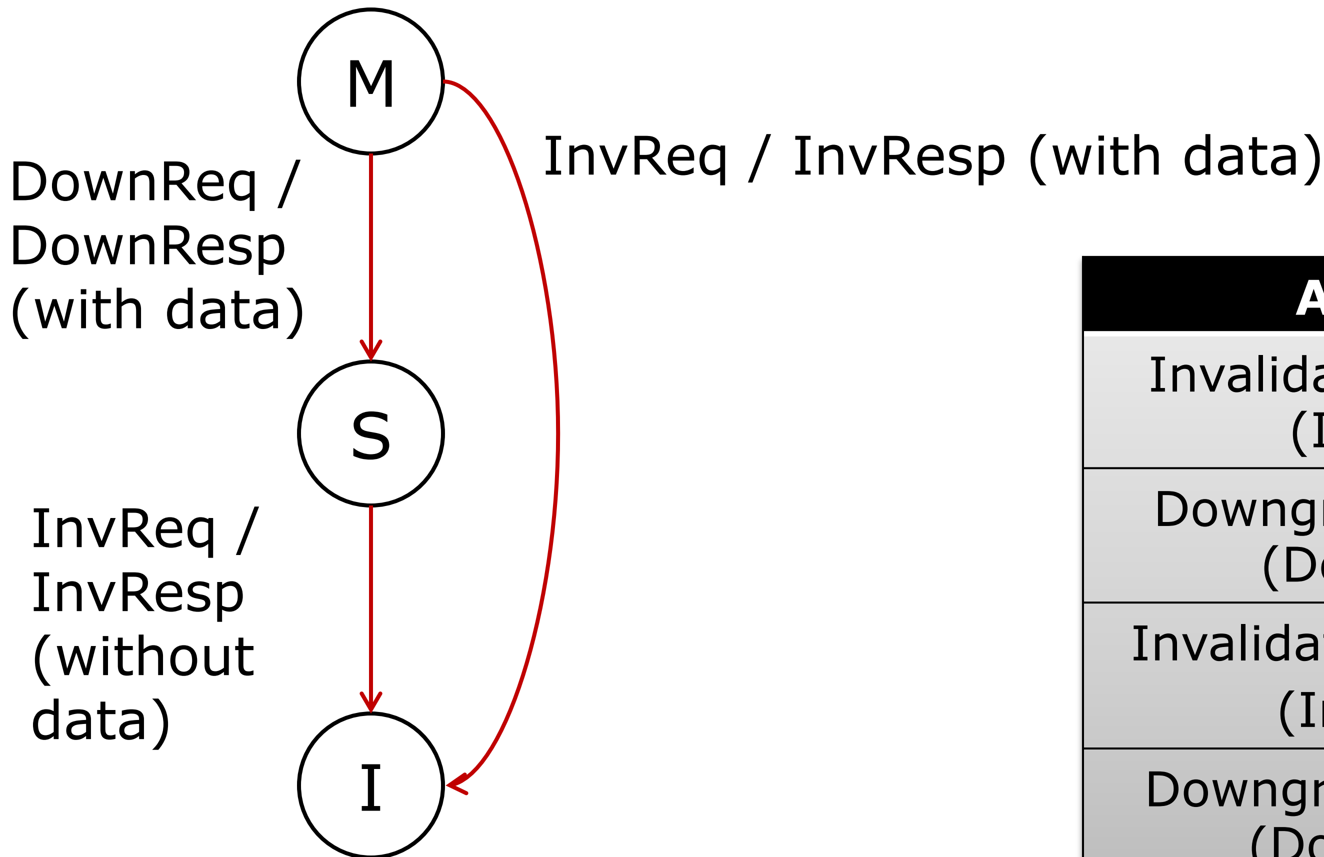
Transitions initiated by processor accesses:



Actions
Processor Read (PrRd)
Processor Write (PrWr)
Shared Request (ShReq)
Exclusive Request (ExReq)

MSI Protocol: Caches (2/3)

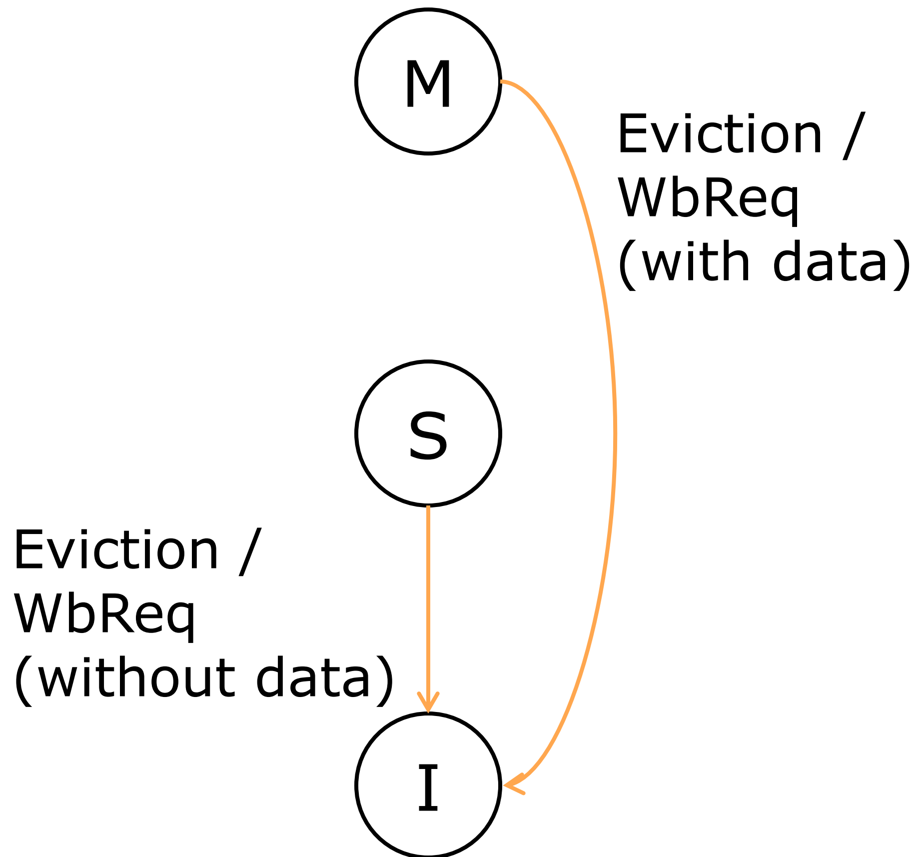
Transitions initiated by directory requests:



Actions
Invalidation Request (InvReq)
Downgrade Request (DownReq)
Invalidation Response (InvResp)
Downgrade Response (DownResp)

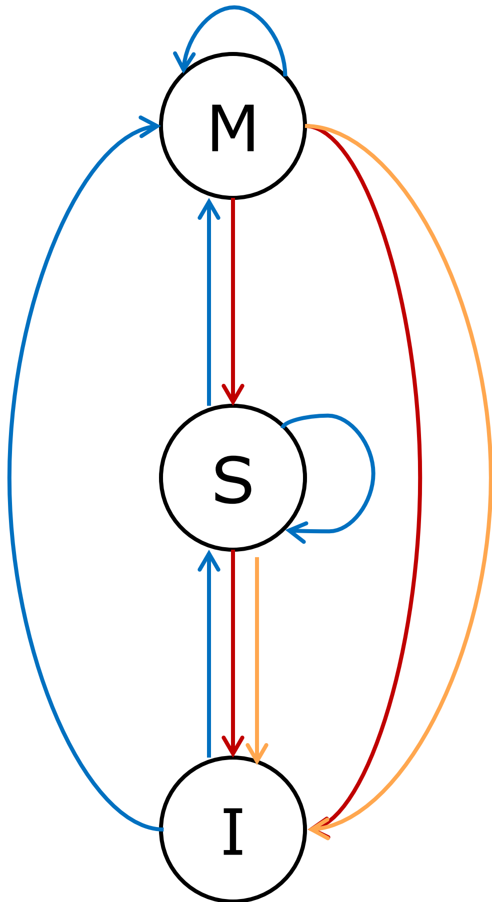
MSI Protocol: Caches (3/3)

Transitions initiated by evictions:



Actions
Writeback Request (WbReq)

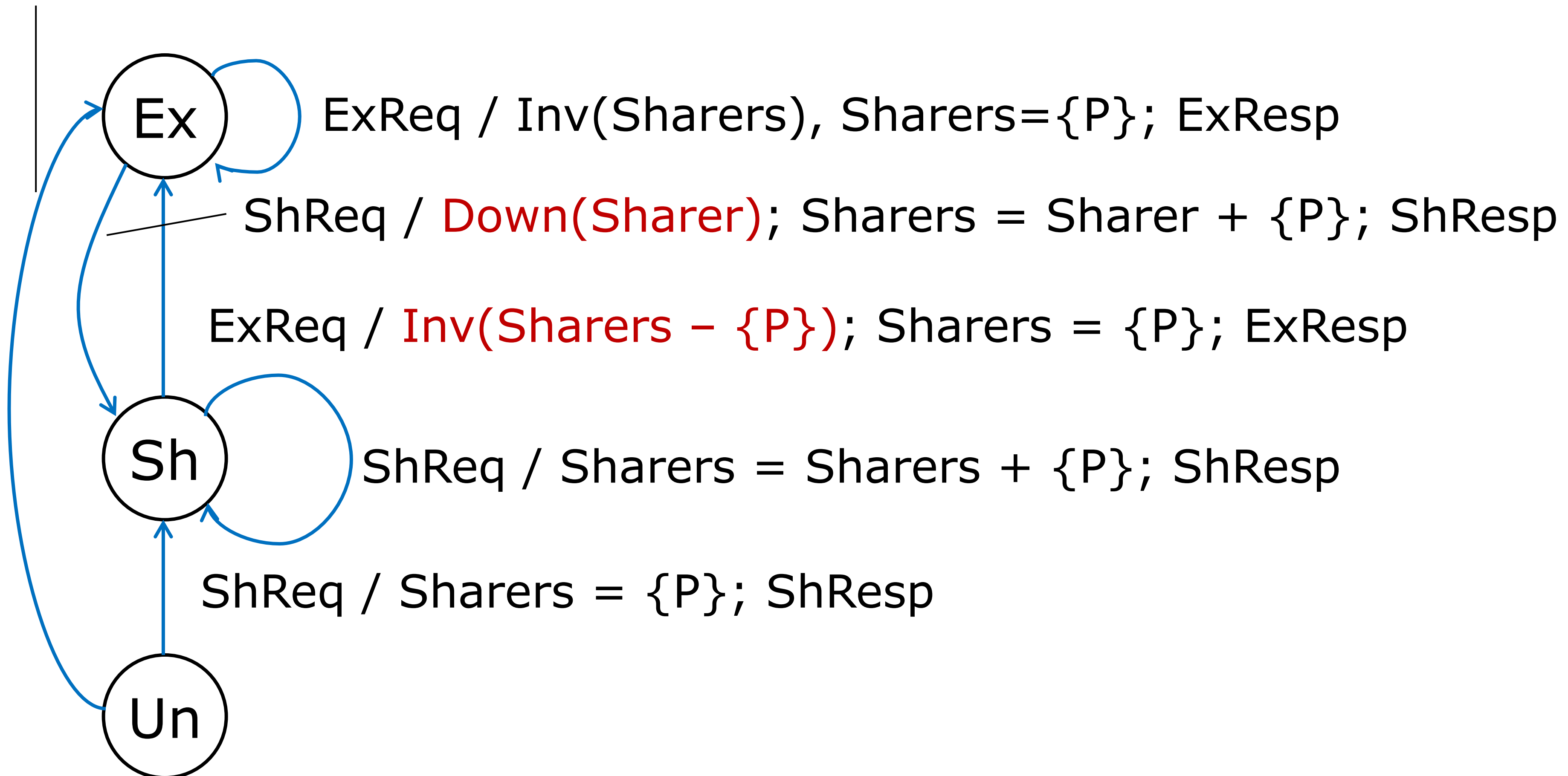
MSI Protocol: Caches



- Transitions initiated by processor accesses
- Transitions initiated by directory requests
- Transitions initiated by evictions

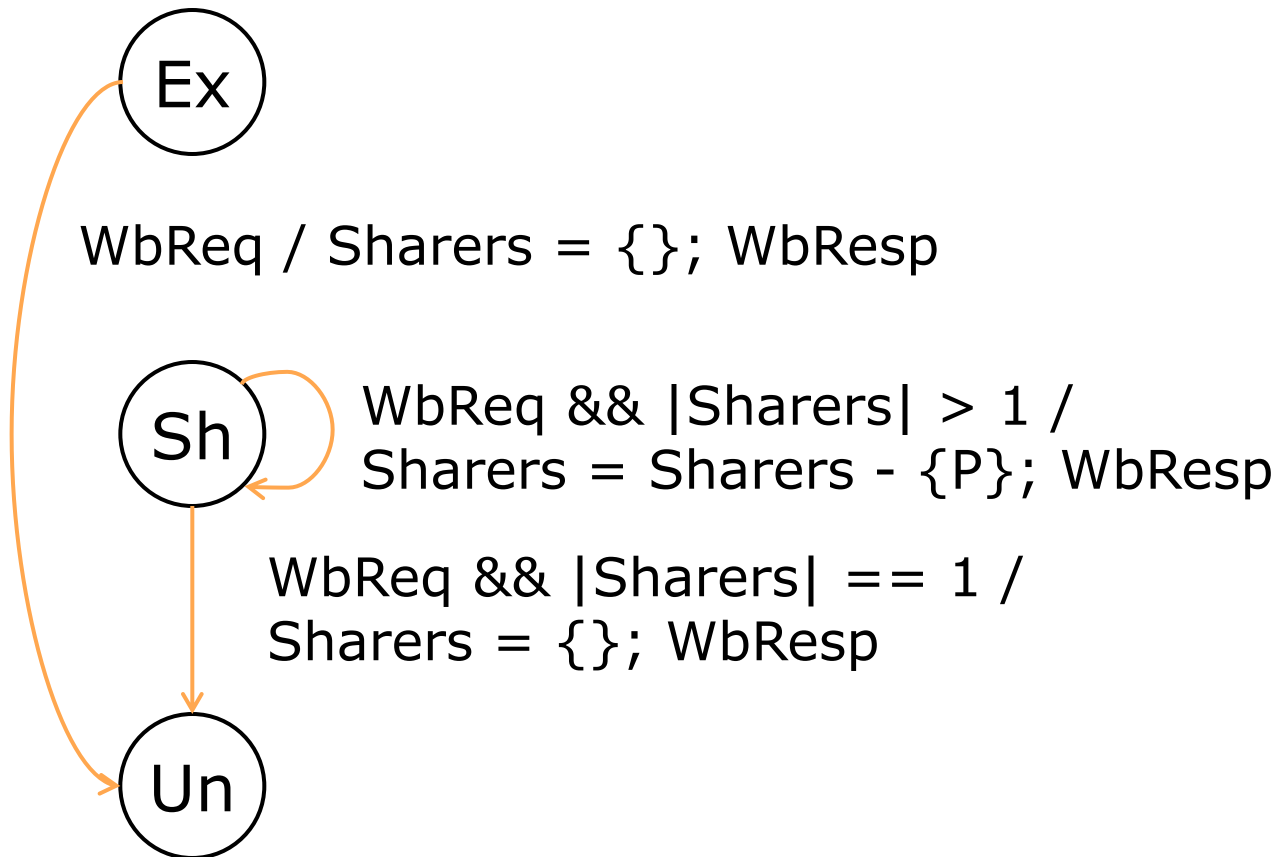
MSI Protocol: Directory (1/2)

Transitions initiated by data requests:

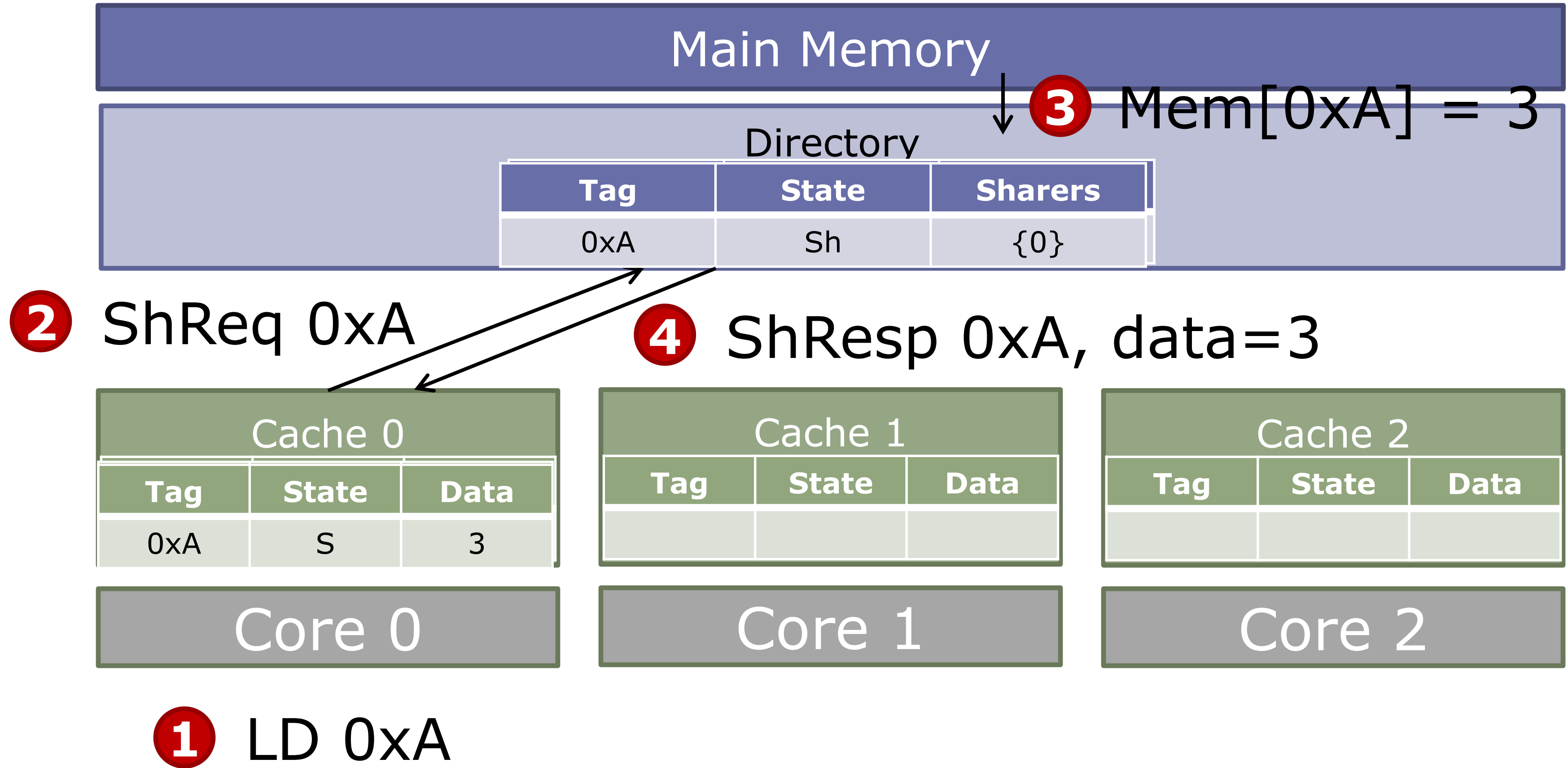


MSI Protocol: Directory (2/2)

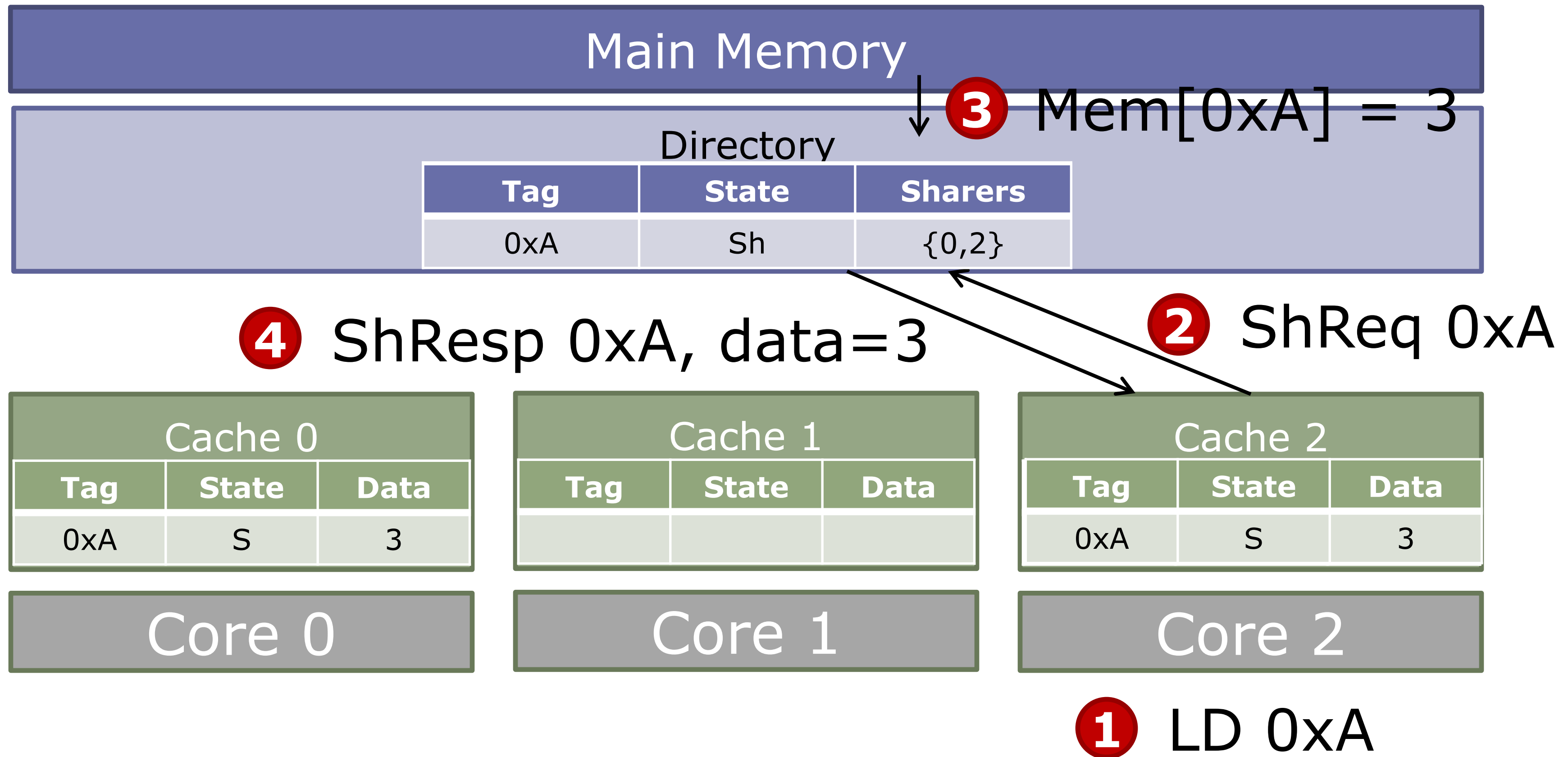
Transitions initiated by writeback requests:



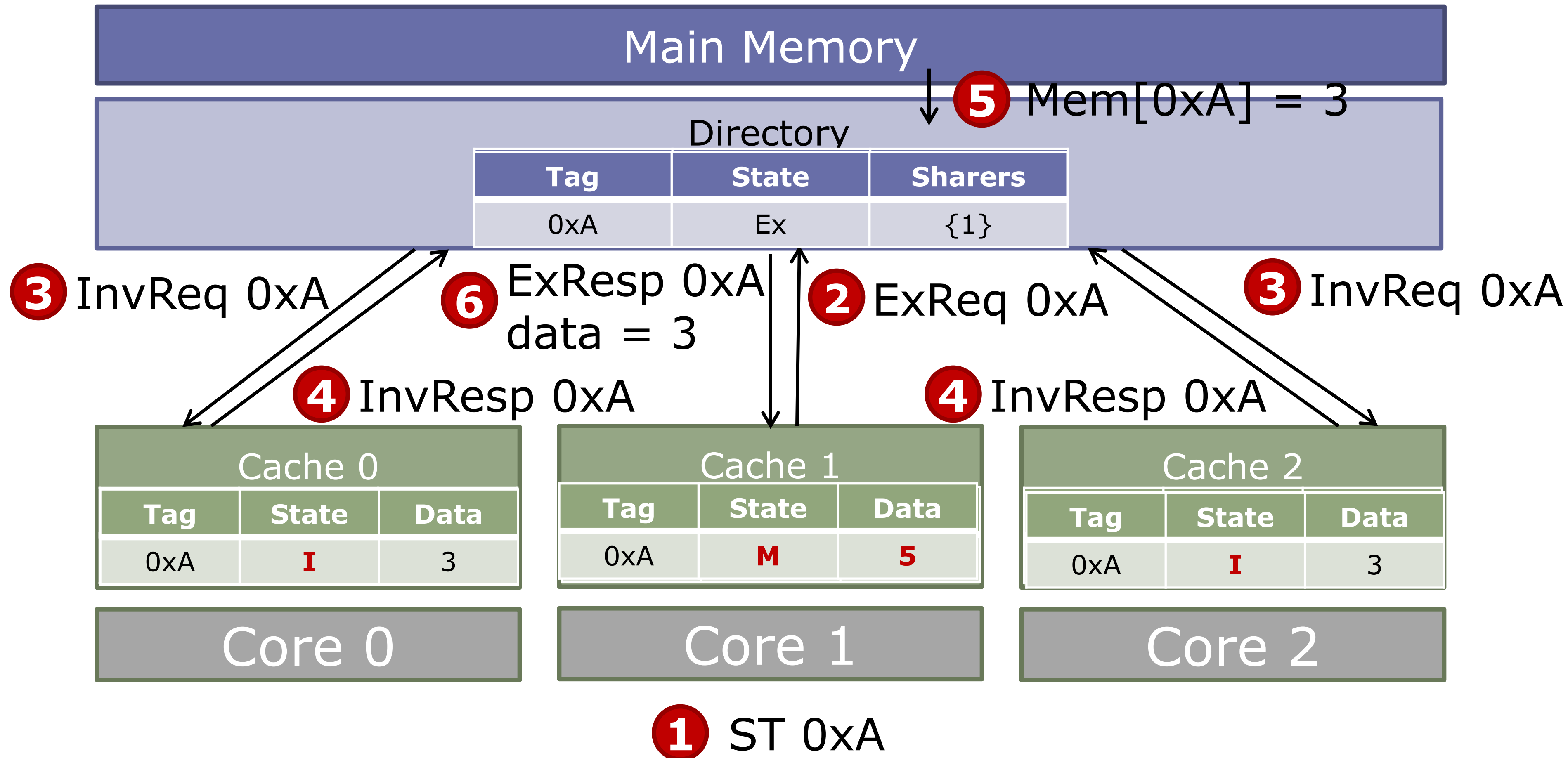
MSI Directory Protocol Example



MSI Directory Protocol Example

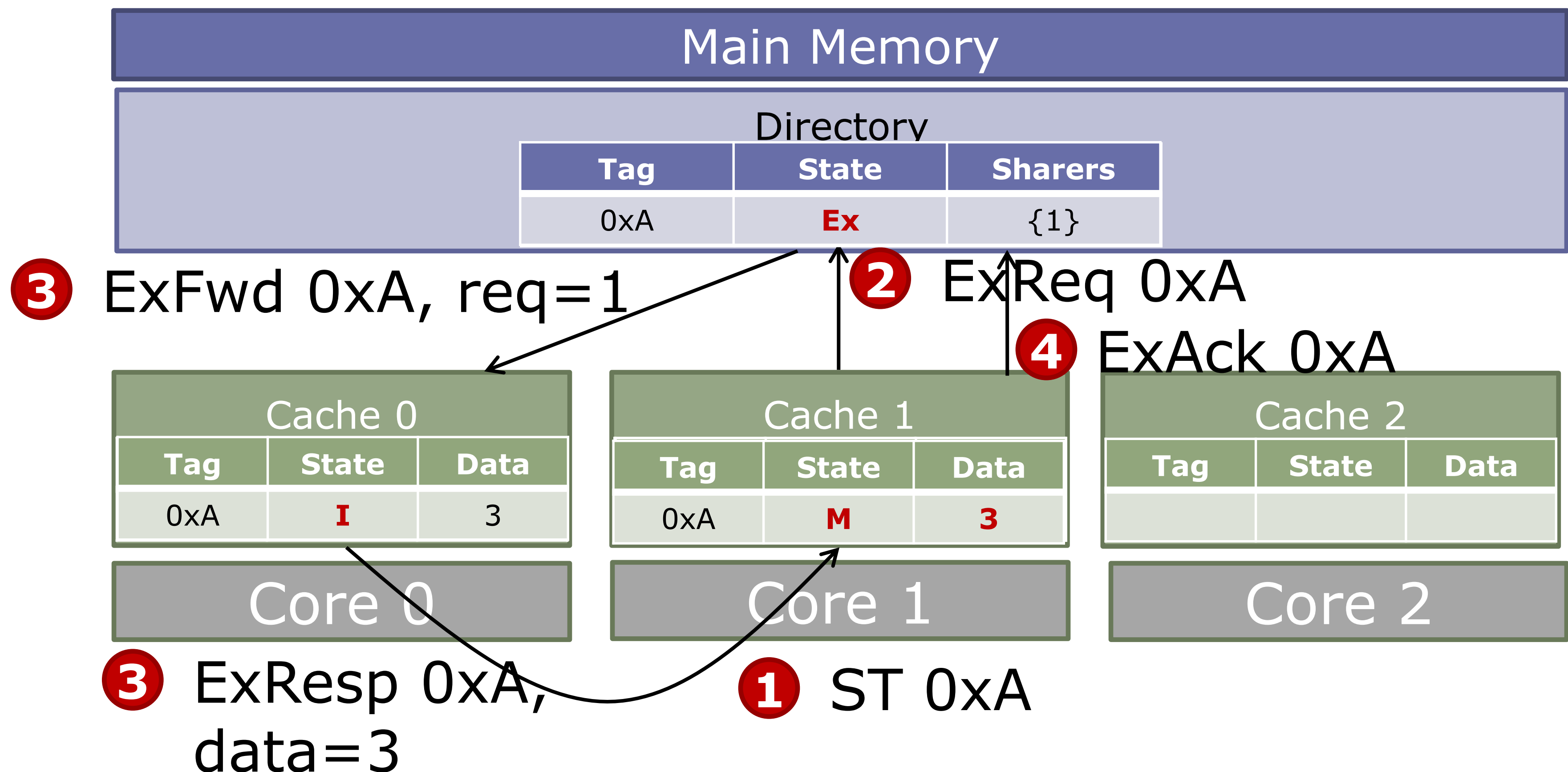


MSI Directory Protocol Example

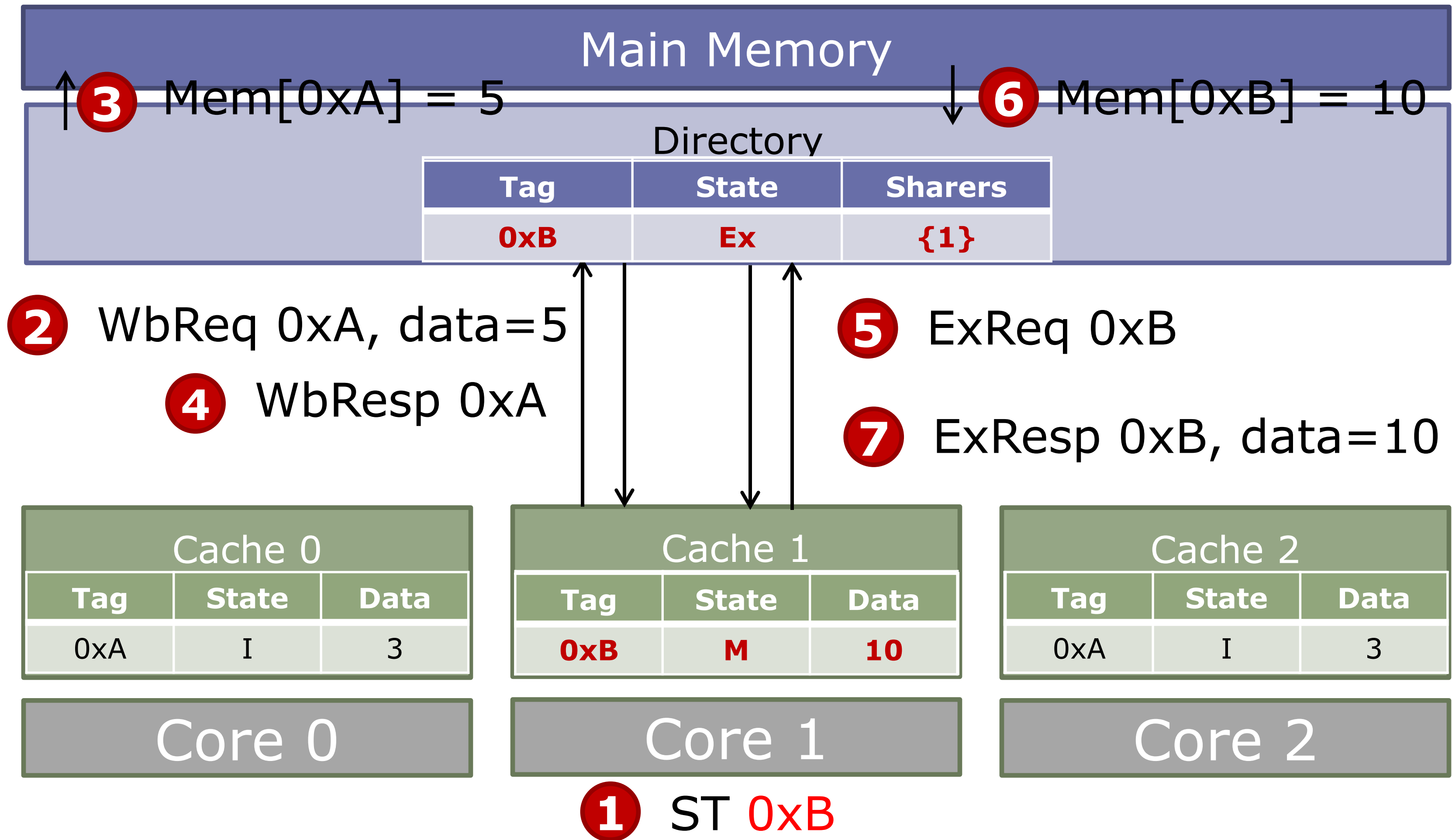


Extra Hops and 3-Hop Protocols
Reducing Protocol Latency

- Problem: Data in another cache needs to pass through the directory, adding latency
- Optimization: Forward data to requester directly



MSI Directory Protocol Example



Why are 0xA's wb and 0xB's req serialized?

Possible solutions?

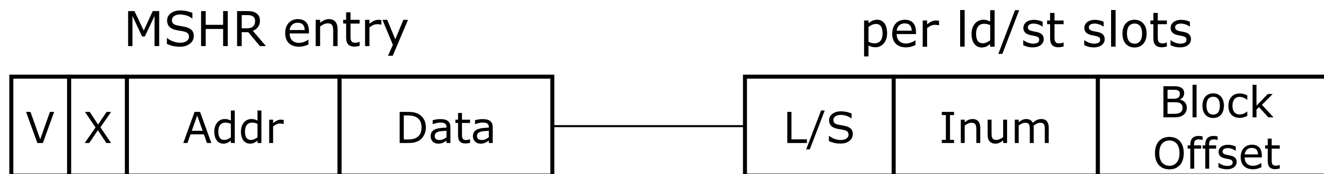
Miss Status Holding Register

MSHR – Holds load misses and writes outside of cache

- On eviction/writeback
  - No free MSHR entry: stall
  - Allocate new MSHR entry
  - When channel available send WBReq and data
  - Deallocate entry on WBResp

Miss Status Holding Register

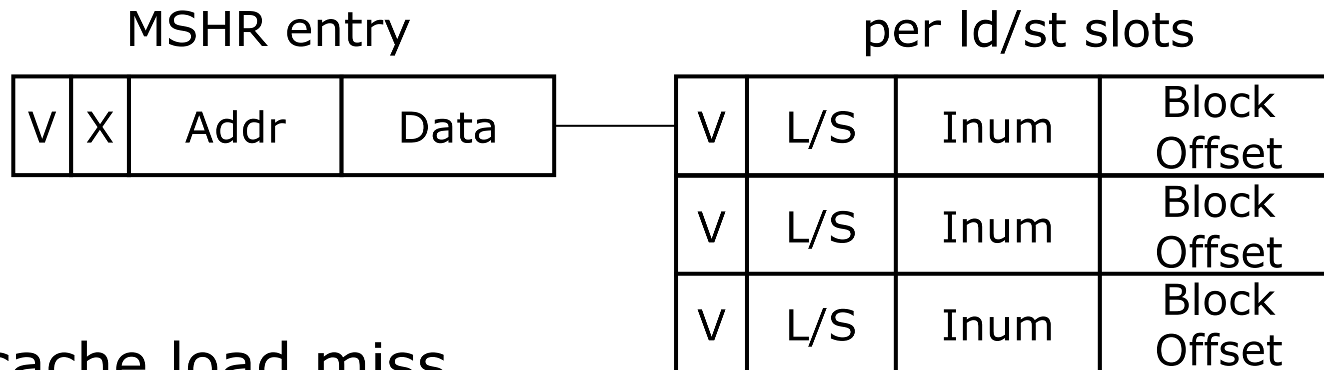
MSHR – Holds load misses and writes outside of cache



- On cache load miss
  - No free MSHR entry: stall
  - Allocate new MSHR entry
  - Send ShReq (or ExReq)
  - On *Resp forward data to CPU and cache
  - Deallocate MSHR

Miss Status Holding Register

MSHR – Holds load misses and writes outside of cache



- On cache load miss
  - Look for matching address in MSHRs
    - If not found
      - If no free MSHR entry: stall
      - Allocate new MSHR entry and fill in
    - If found, just fill in per ld/st slot
  - Send ShReq (or ExReq)
  - On *Resp forward data to CPU and cache
  - Deallocate MSHR

Per ld/st slots allow servicing multiple requests with one entry

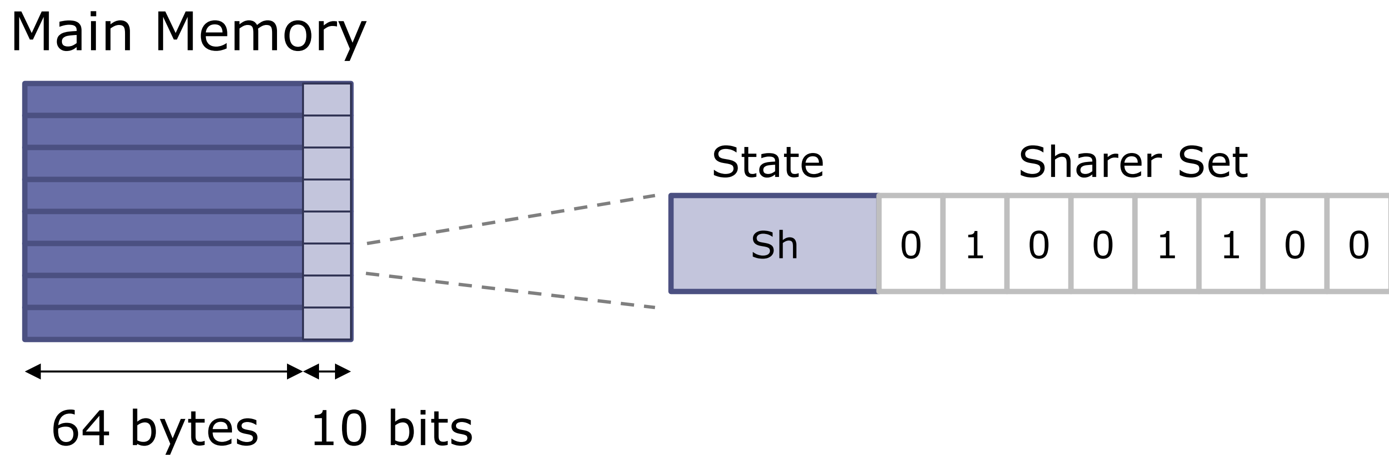
Directory Organization

- Requirement: Directory needs to keep track of all the cores that are sharing a cache block

- Challenge: For each block, the space needed to hold the list of sharers grows with number of possible sharers…

Flat, Memory-based Directories

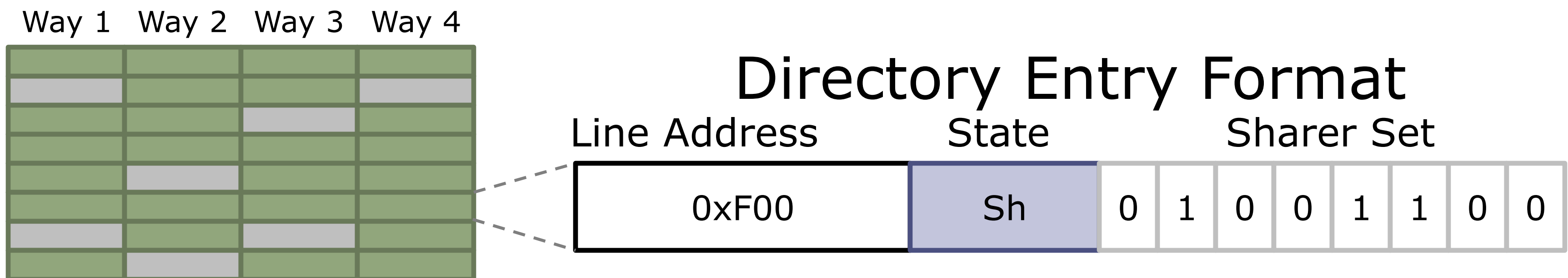
- Dedicate a few bits of main memory to store the state and sharers of every line
- Encode sharers using a bit-vector



✓ Simple
× Slow
× Very inefficient with many processors (~P bits/line)

Sparse Full-Map Directories

- Not every line in the system needs to be tracked – only those in private caches!
- Idea: Organize directory as a cache



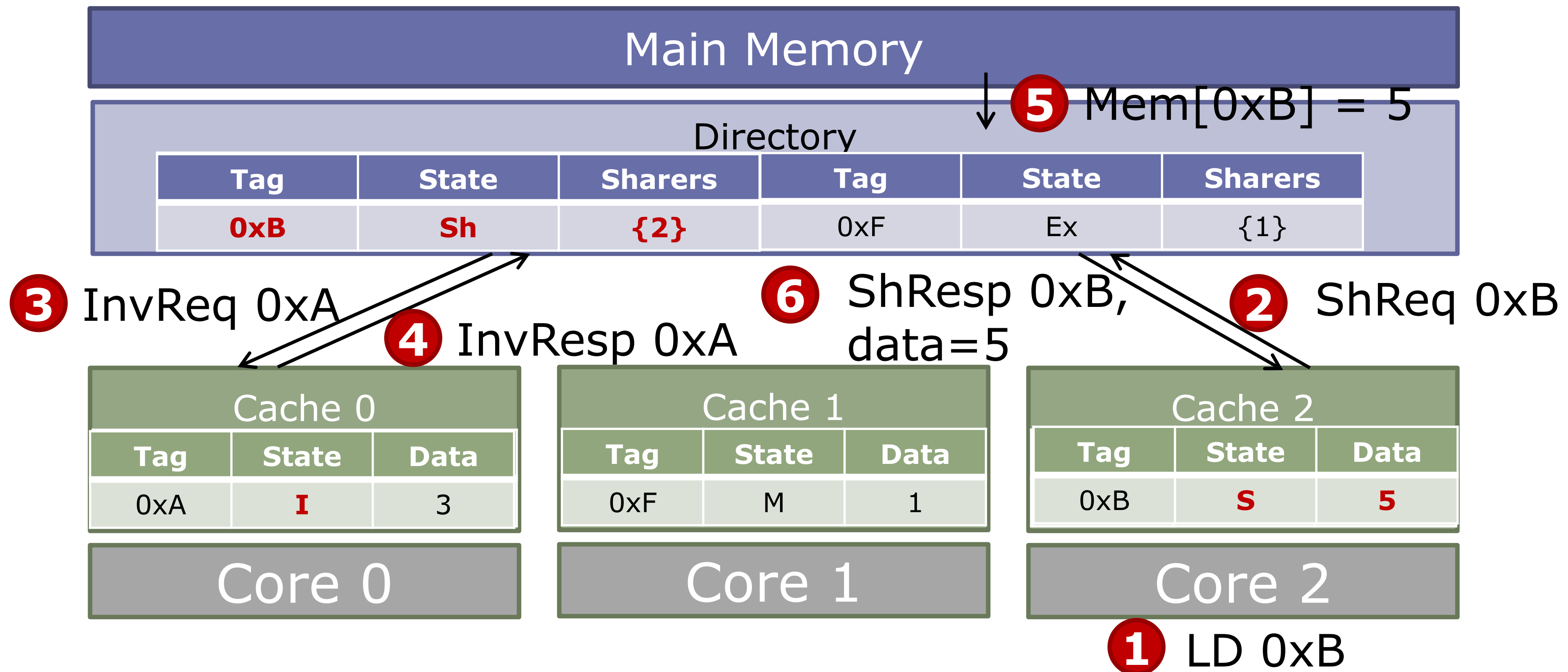
✓ Low latency, energy-efficient

✗ Bit-vectors grow with # cores → Area scales poorly

✗ Limited associativity → Directory-induced invalidations

Directory-Induced Invalidations

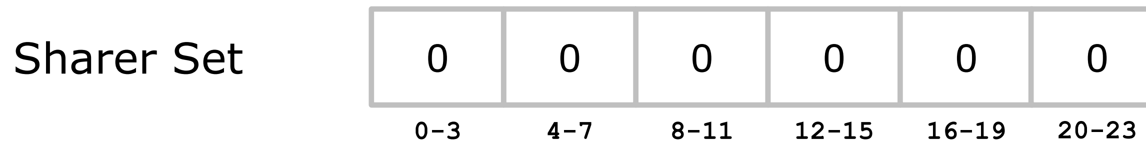
- To retain **inclusion**, must invalidate all sharers of an entry before reusing it for another address
- Example: 2-way set-associative sparse directory



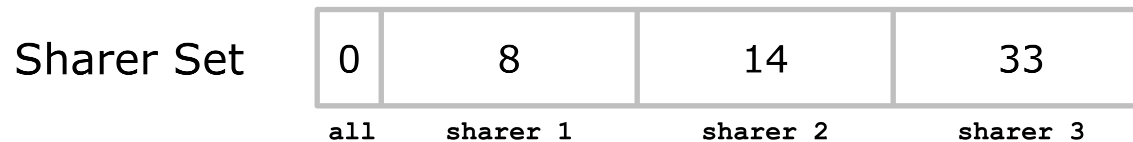
How many entries should the directory have?

Inexact Representations of Sharer Sets

- Coarse-grain bit-vectors (e.g., 1 bit per 4 cores)



- Limited pointers: Maintain a few sharer pointers, on overflow mark 'all' and broadcast (or invalidate another sharer)



- Allow false positives (e.g., Bloom filters)

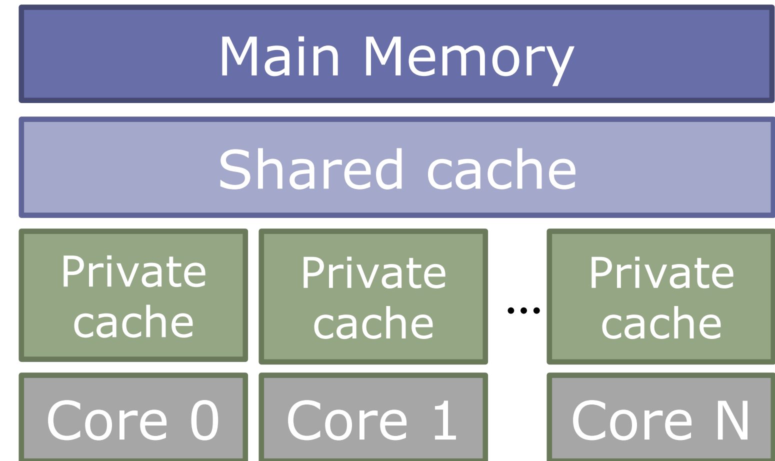
✓ Reduced area & energy

✗ Overheads still not scalable (these techniques simply play with constant factors)

✗ Inexact sharers → Broadcasts, invalidations or spurious invalidations and downgrades

In-Cache Directories

- Common multicore memory hierarchy:
  - 1+ levels of private caches
  - A shared last-level cache
  - Need to enforce coherence among private caches
- Idea: Embed the directory information in shared cache tags
  - Shared cache must be inclusive

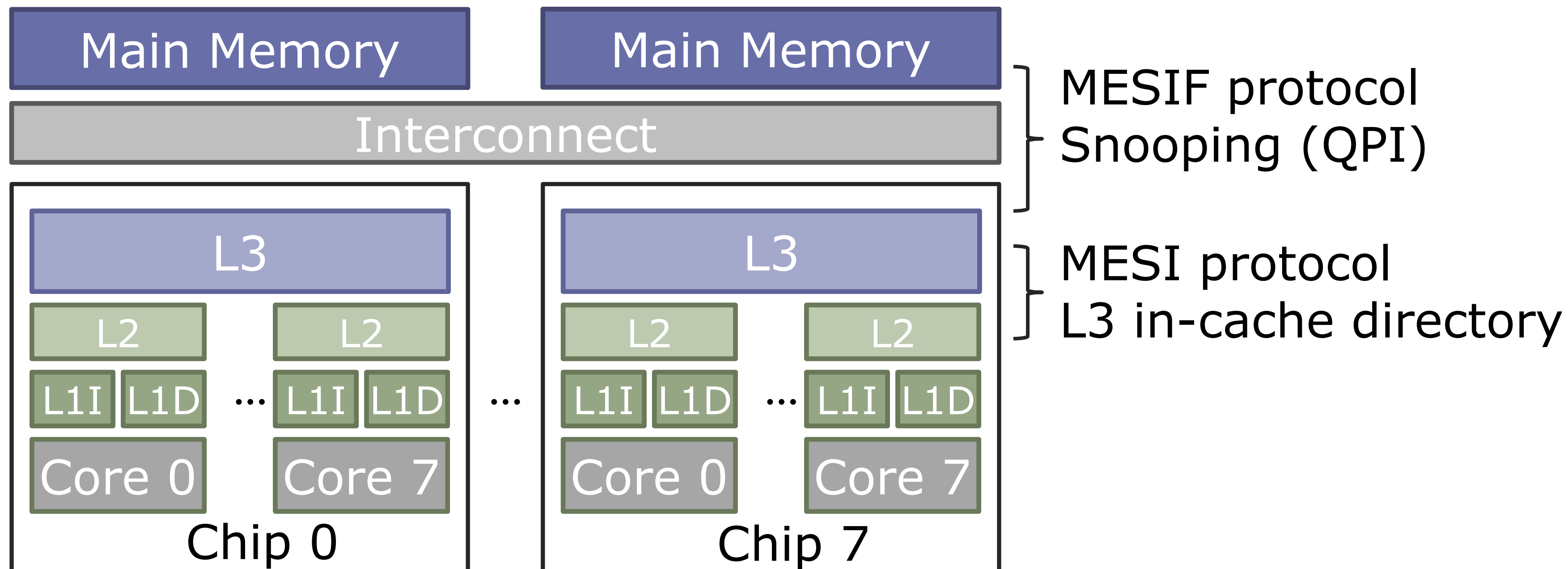


✓ Avoids tag overheads & separate lookups

✗ Can be inefficient if shared cache size >> sum(private cache sizes)

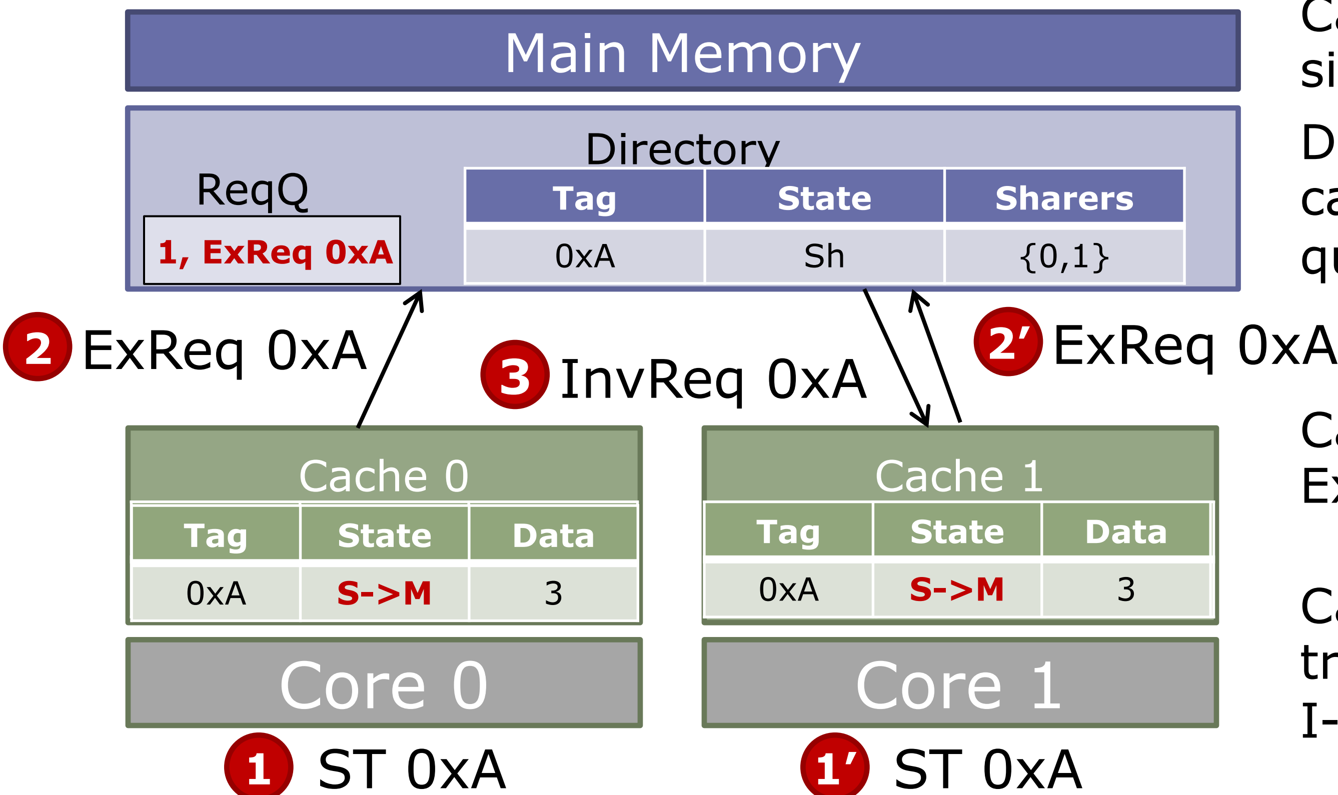
Coherence in Multi-Level Hierarchies

- Can use the same or different protocols to keep coherence across multiple levels
- Key invariant: Ensure sufficient permissions in all intermediate levels
- Example: 8-socket Xeon E7 (8 cores/socket)



Protocol Races

- Directory serializes multiple requests for the same address
  - Same-address requests are queued or NACKed and retried
- But races still exist due to conflicting requests
- Example: Upgrade race



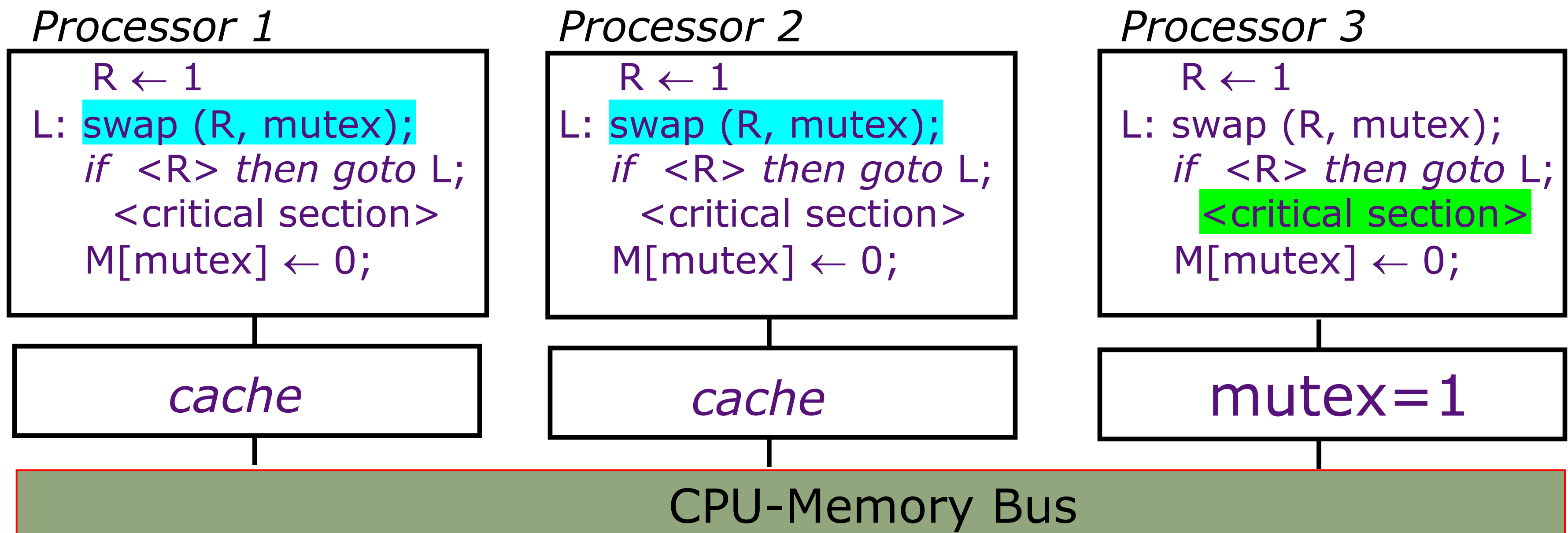
Caches 0 and 1 issue simultaneous ExReqs

Directory starts serving cache 0's ExReq, queues cache 1's

Cache 1 expected ExResp, but got InvReq!

Cache 1 should transition from S->M to I->M and send InvResp

Coherence and Synchronization

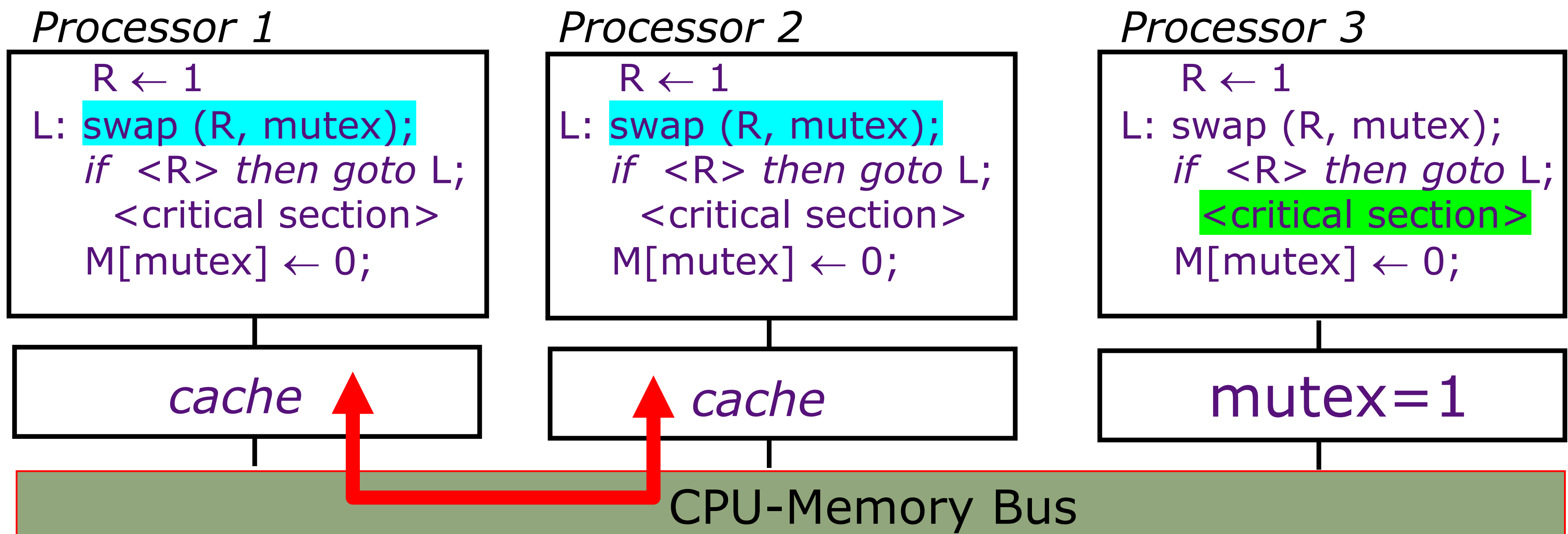



```
swap (R, mutex):
    R = test&set(mutex)

test&set(mutex):
    old_val = M[mutex];
    M[mutex] = 1;
    return old_val;
```

Our cache coherence protocol will introduce a performance issue here. What is the problem?

Coherence and Synchronization



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 (test&test&set).

Implementing Atomic Instructions

```
test&set(mutex):
    old_val = M[mutex];
    M[mutex] = 1;
    return old_val;
```

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

- Implementation options:
  - *With snoopy coherence, lock the bus → expensive*
  - *With directory-based coherence, lock the line in the cache (prevent invalidations or evictions until atomic op finishes) → complex*

- modern processors use

  load-reserve

  store-conditional

Load-reserve & Store-conditional

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

```
Load-reserve R, (a):
    <flag, adr> ← <1, a>;
    R ← M[a];
```

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

If the cache receives an invalidation to the address in the reserve register, the reserve bit is set to 0

- Several processors may reserve 'a' simultaneously
- These instructions are like ordinary loads and stores with respect to the bus traffic

Load-Reserve/Store-Conditional

Swap implemented with Ld-Reserve/St-Conditional

```
#       Swap(R1, mutex):

L:      Ld-Reserve R2, (mutex)
        St-Conditional (mutex), R1
        if (status == fail) goto L
        R1 <- R2
```

Performance:
Load-reserve & Store-conditional

The total number of coherence transactions is not necessarily reduced, but splitting an atomic instruction into load-reserve & store-conditional:

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

Thank you!

Next Lecture:
Consistency and
Relaxed Memory Models