Computer Architecture

Spring 2016

Lecture 21: Cache Coherence & Memory Consistency

Shuai Wang

Department of Computer Science and Technology

Nanjing University

[Slides adapted from CSE 502 Stony Brook University and EECS 6.823 MIT]

Directory-Based Coherence Protocols

- Extend memory to track caching information
- For each physical cache line, a *home* directory tracks:
  - Owner: core that has a dirty copy (i.e., M state)
  - Sharers: cores that have clean copies (i.e., S state)
- Cores send coherence events to home directory
  - Home directory only sends events to cores that care

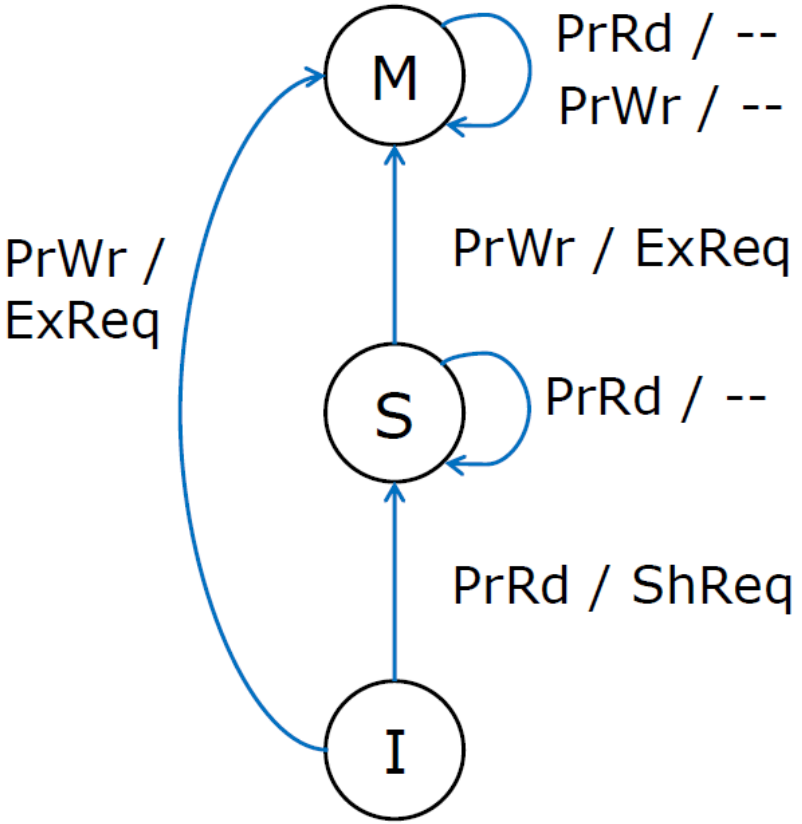
MSI Directory Protocol

- 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)

MSI Directory Protocol: Caches (1/3)

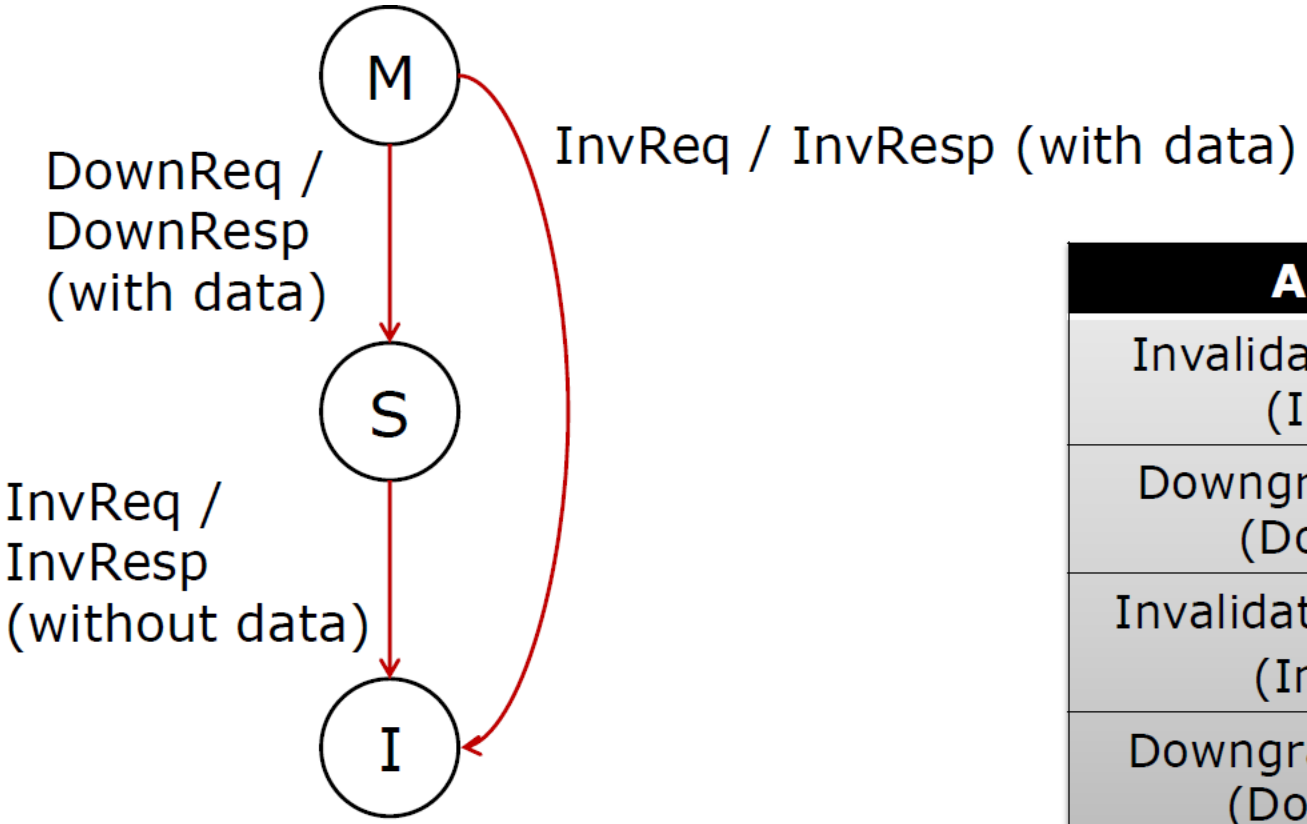
Transitions initiated by processor accesses:



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

MSI Directory Protocol: Caches (2/3)

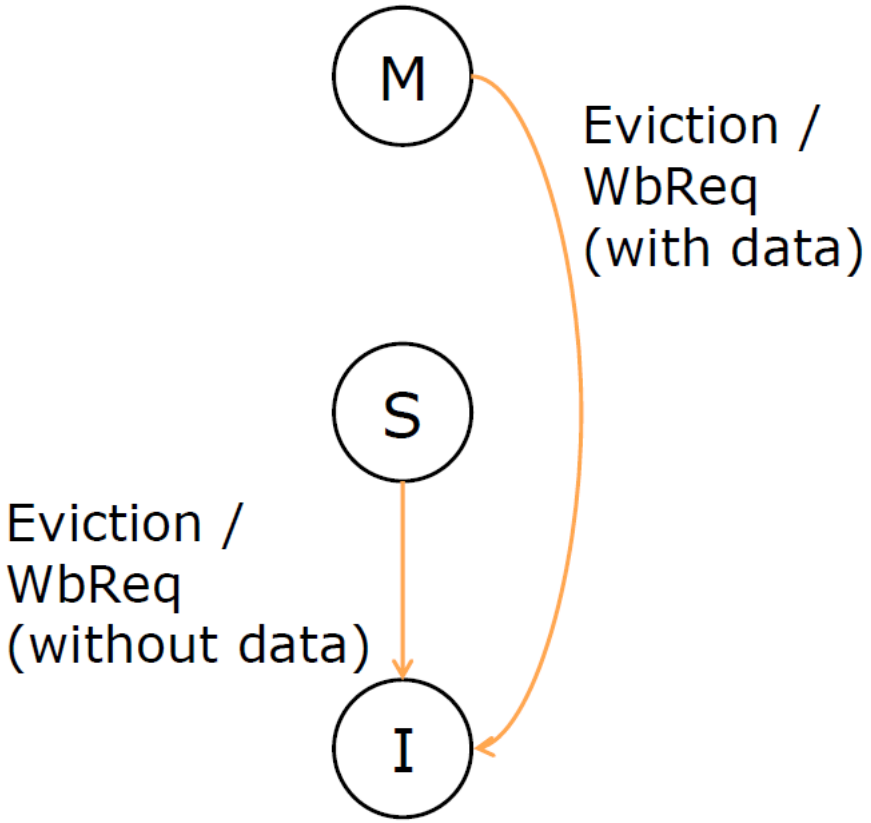
Transitions initiated by directory requests:



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

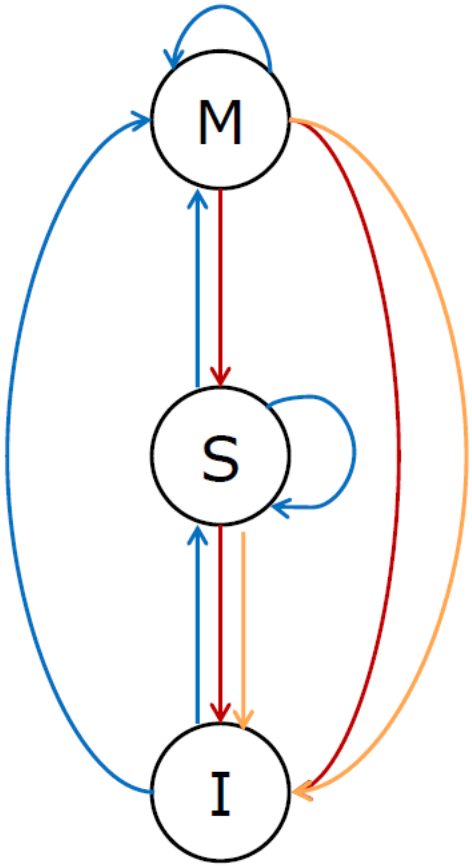
MSI Directory Protocol: Caches (3/3)

Transitions initiated by evictions:



Actions
Writeback Request (WbReq)

MSI Directory Protocol: Caches



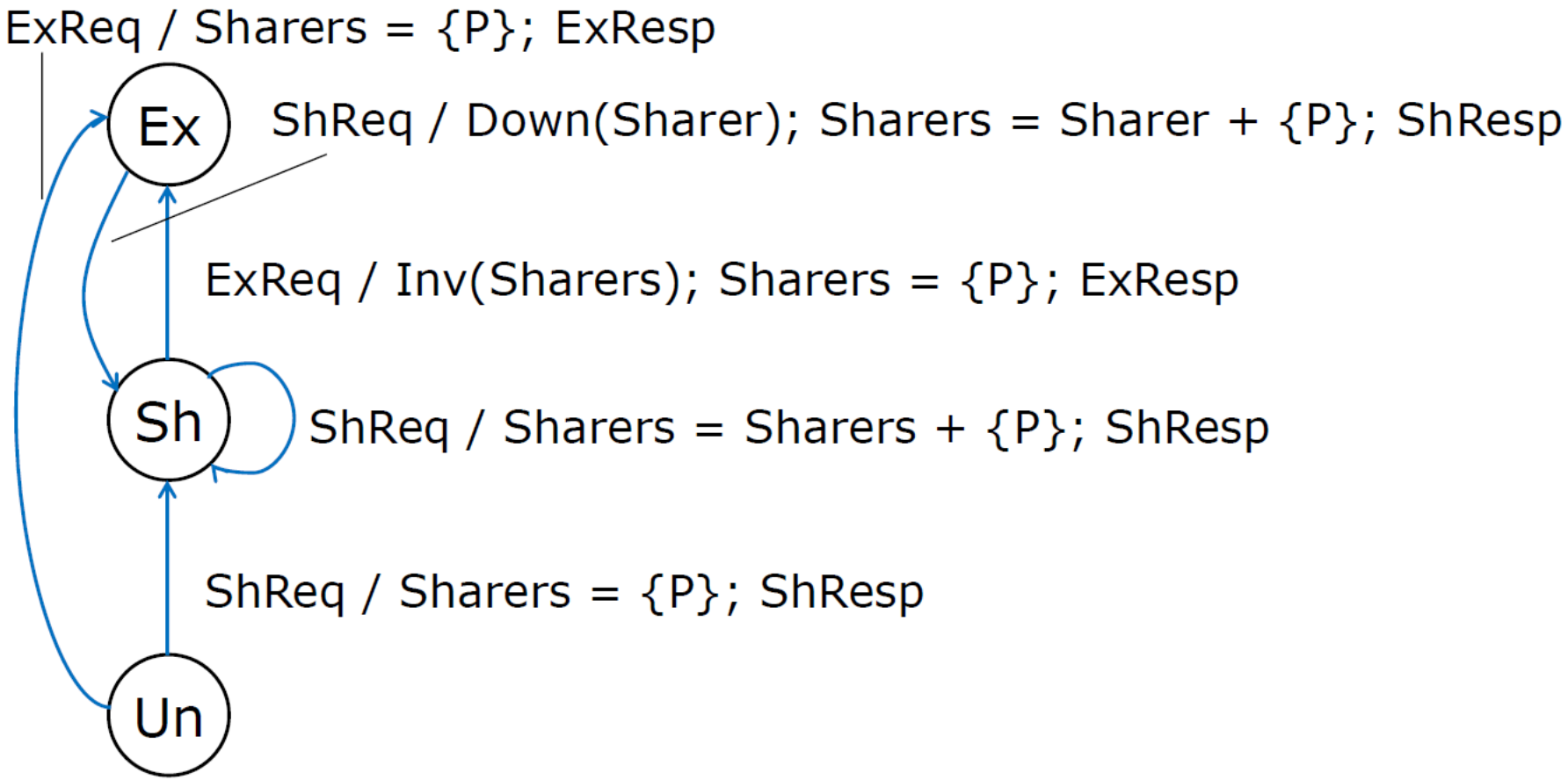
→ Transitions initiated by processor accesses

→ Transitions initiated by directory requests

→ Transitions initiated by evictions

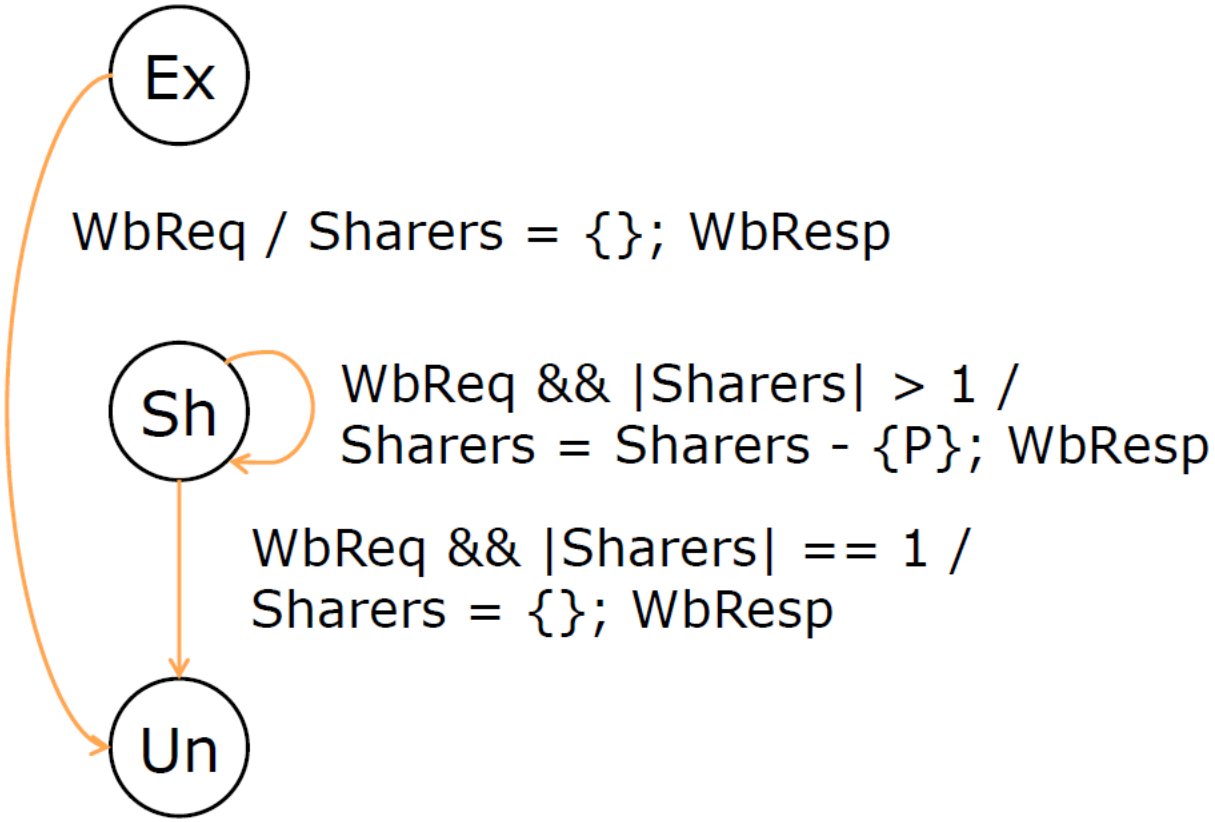
MSI Directory Protocol: Directory (1/2)

Transitions initiated by data requests:

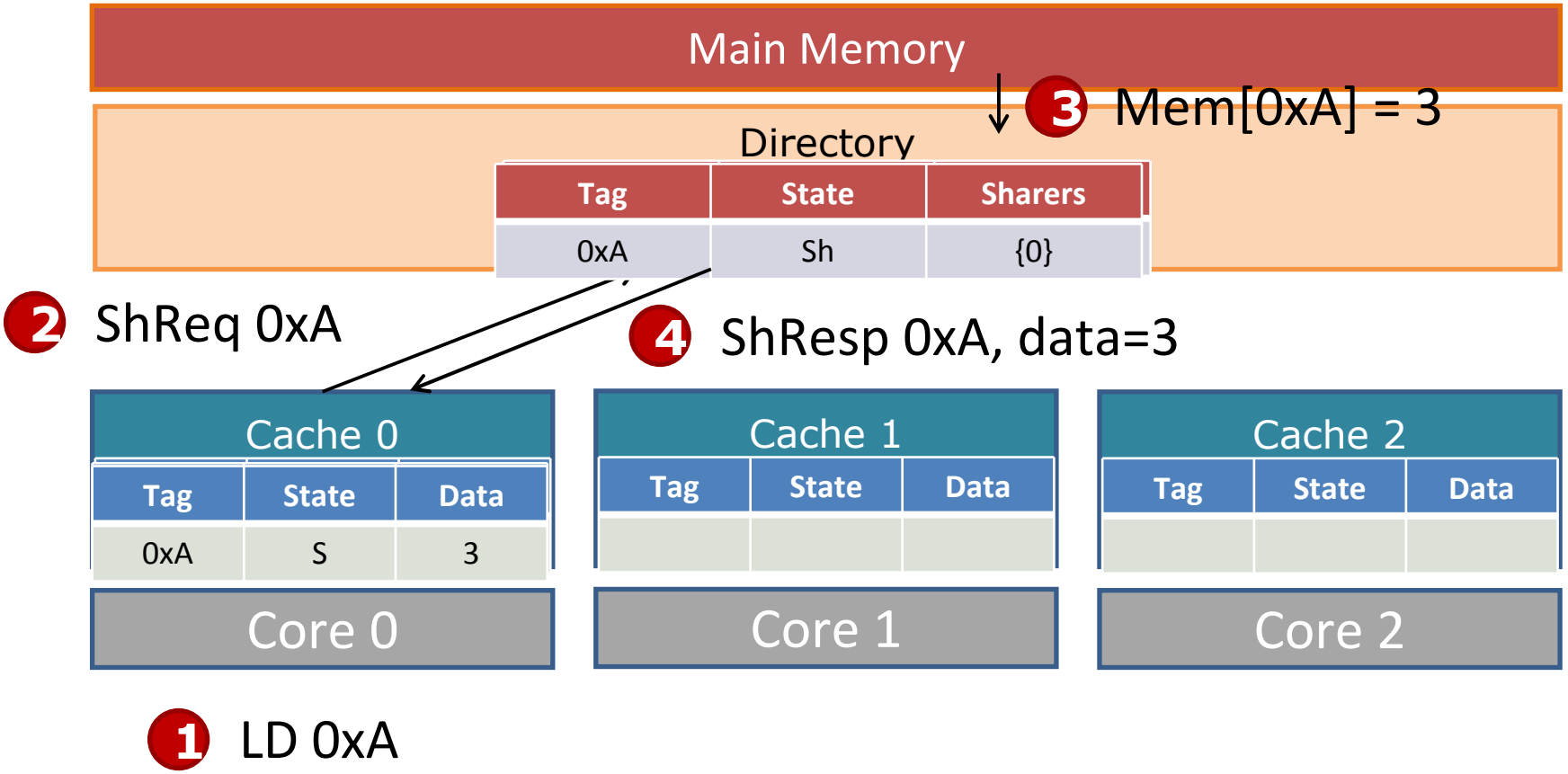


MSI Directory Protocol: Directory (2/2)

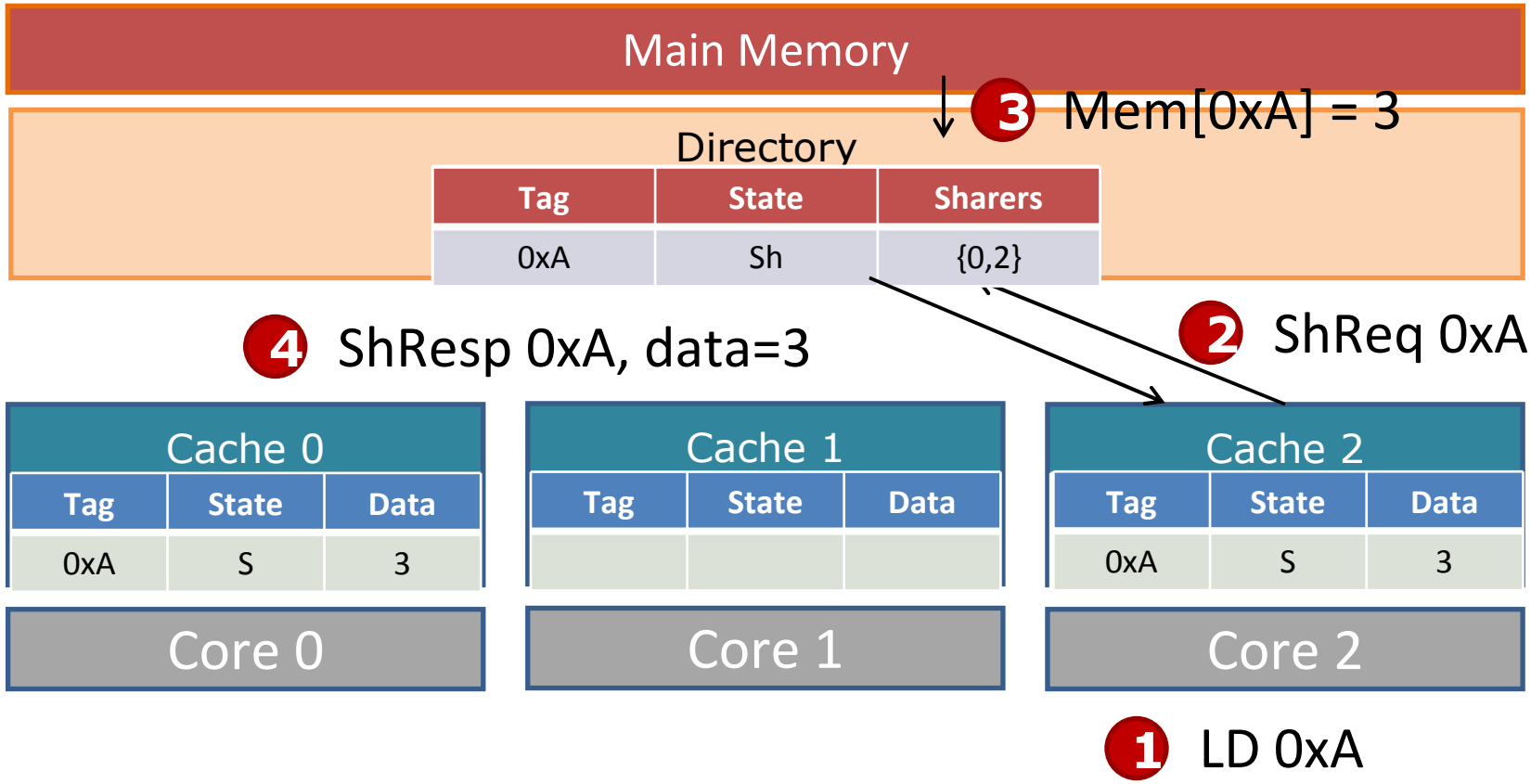
Transitions initiated by writeback requests:



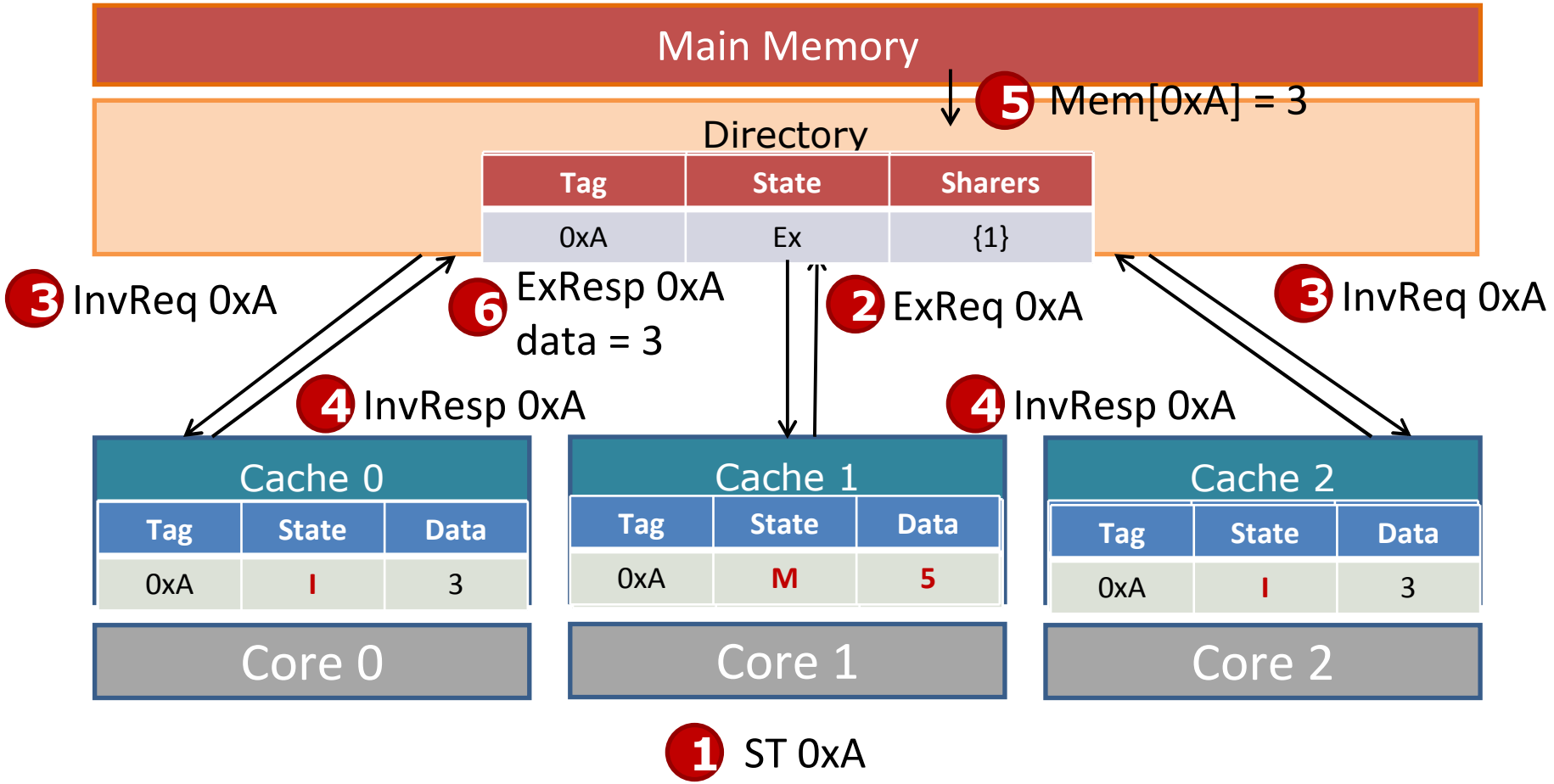
MSI Directory Protocol Example



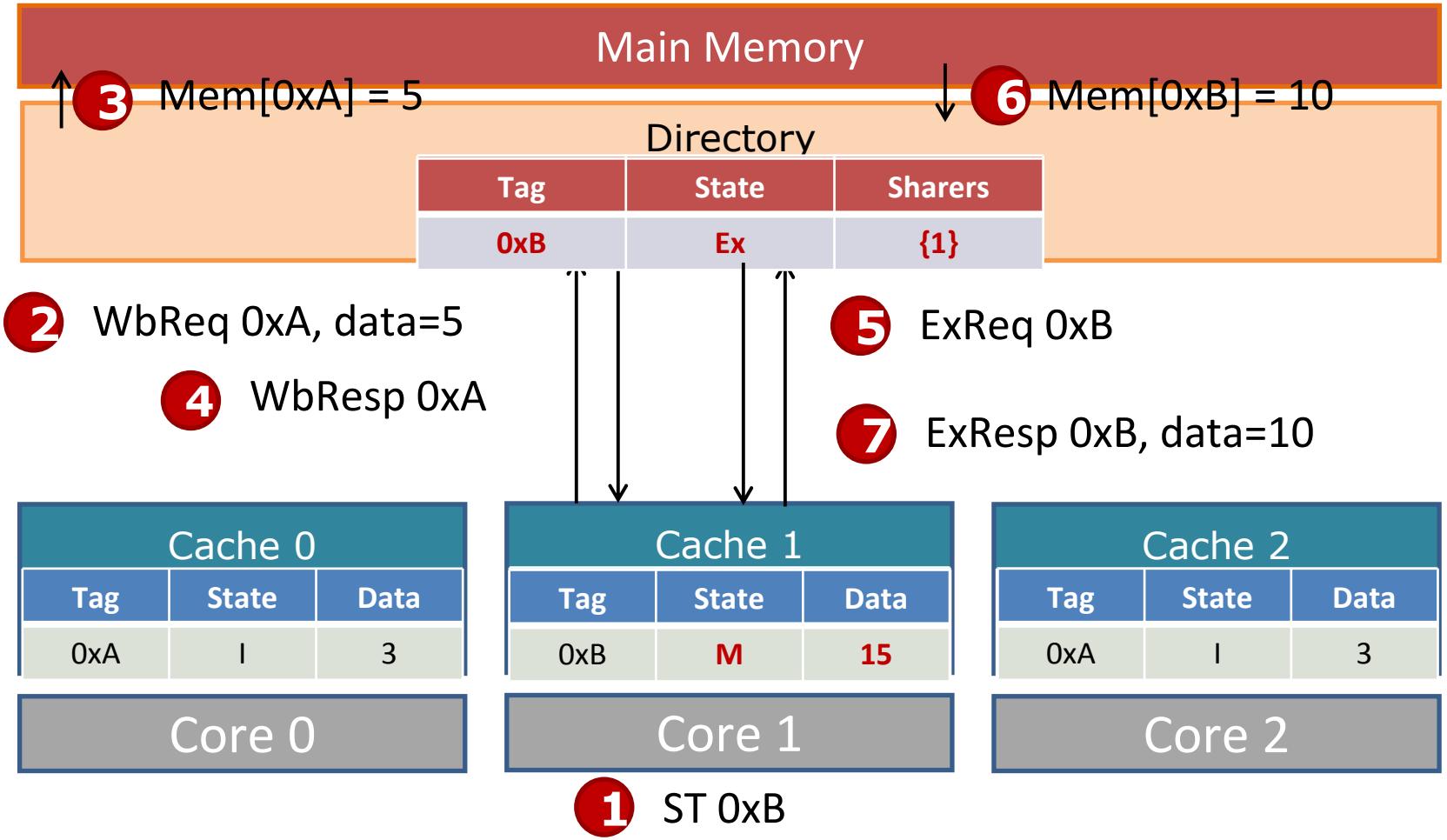
MSI Directory Protocol Example



MSI Directory Protocol Example



MSI Directory Protocol Example

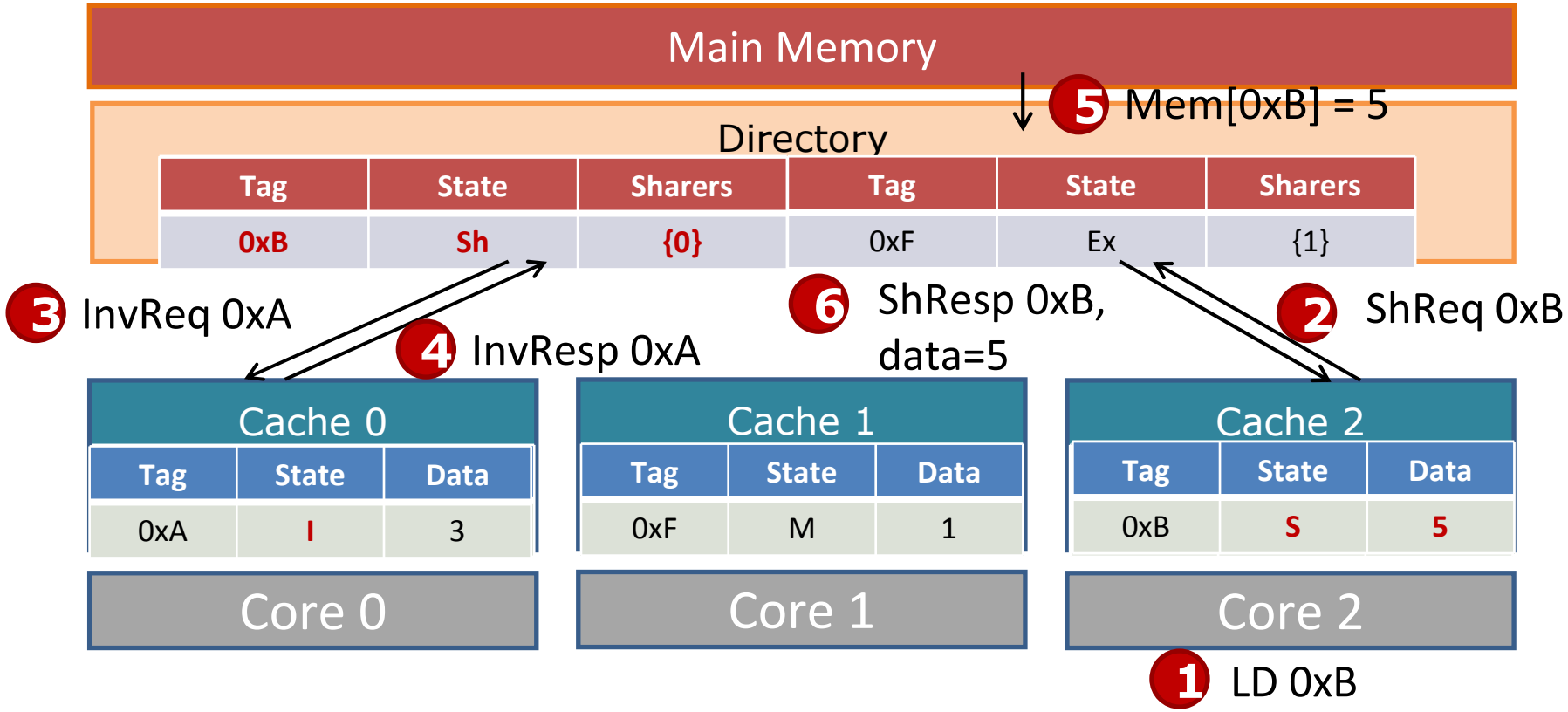


Why are 0xA's wb and 0xB's req serialized? Structural dependence

Possible solutions? Buffer outside of cache to hold write data

Directory-Induced Invalidations

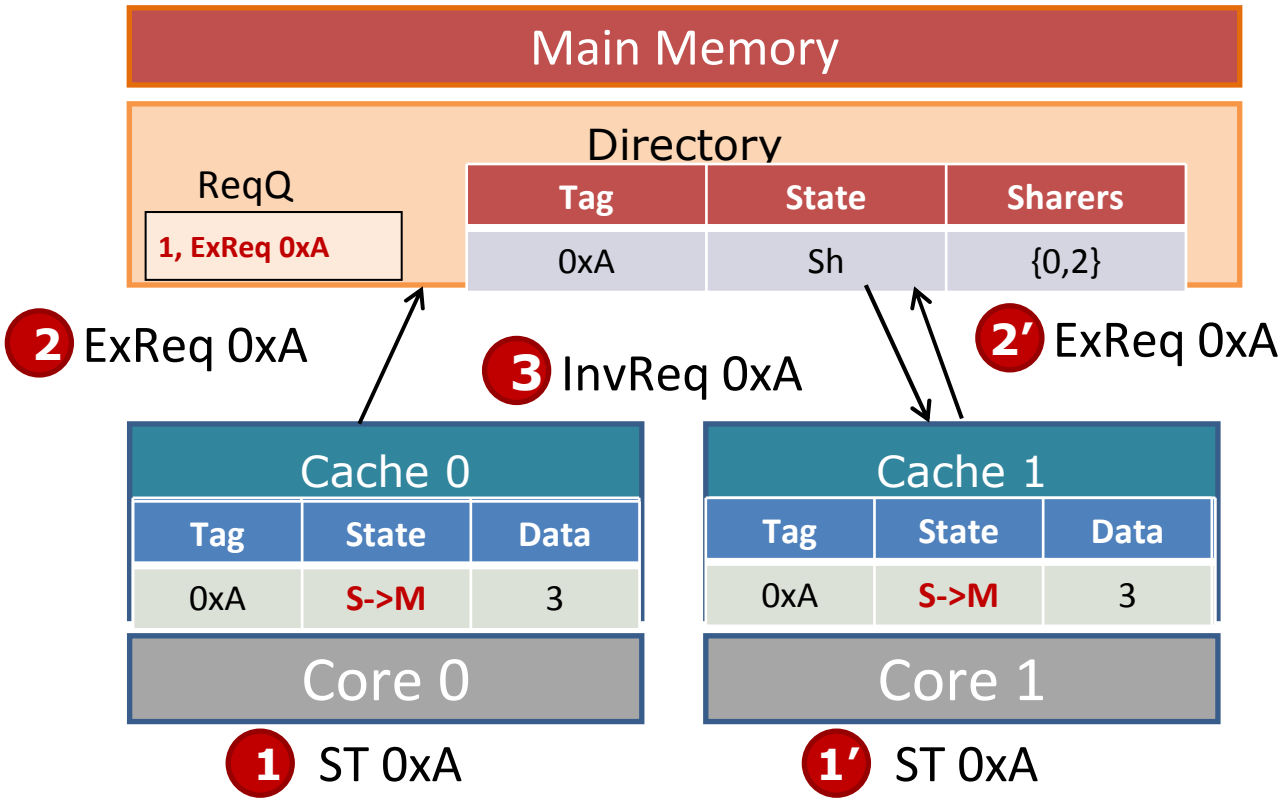
- To retain inclusion, must invalidate all sharers of an entry before reusing it for another address
- Example: 2-way set-associative sparse directory



How many entries should the directory have?

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

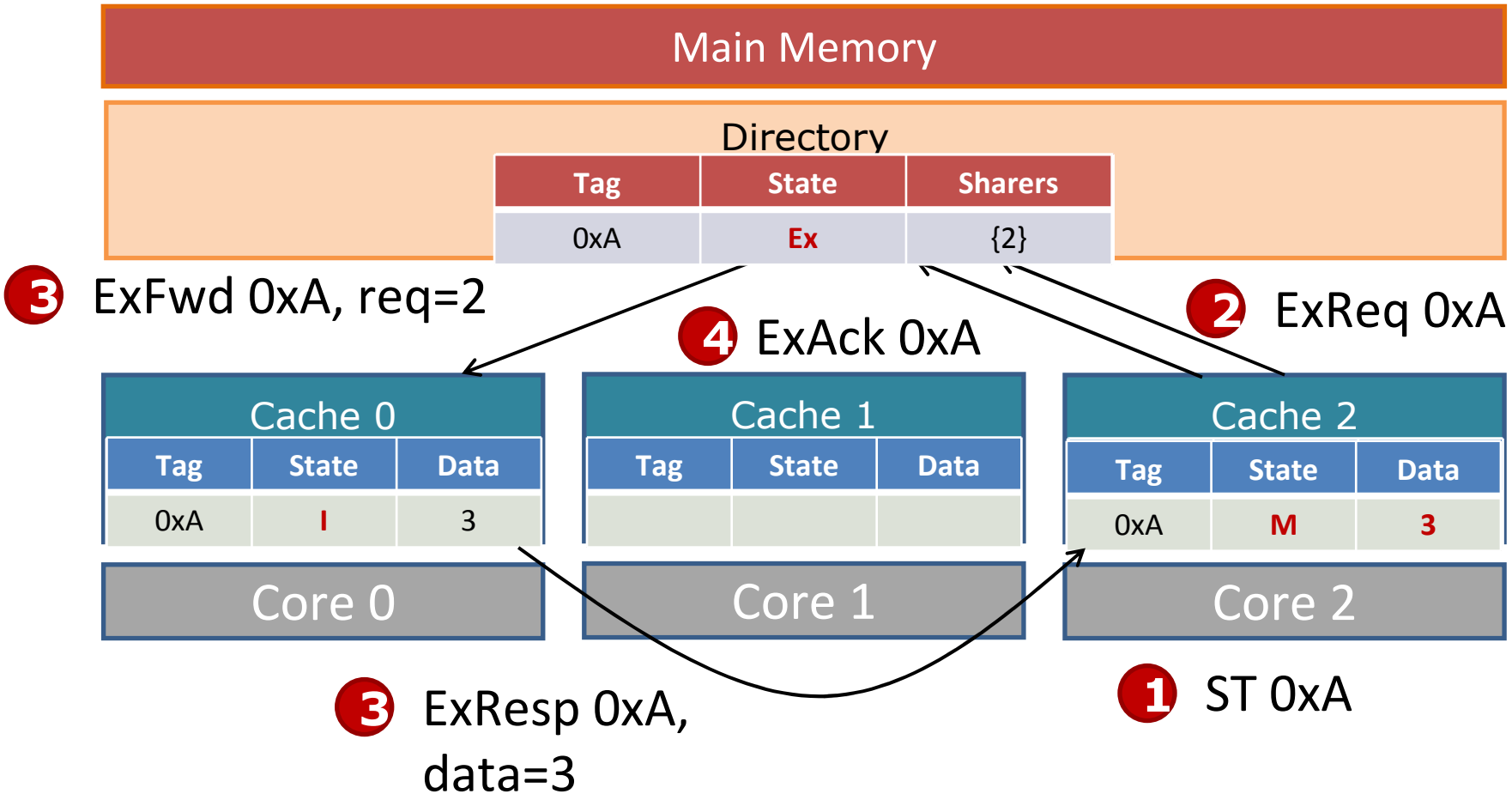
Directory starts serving cache 0's ExReq, queues cache 1's

Cache 1 expected ExResp, but got InvReq!

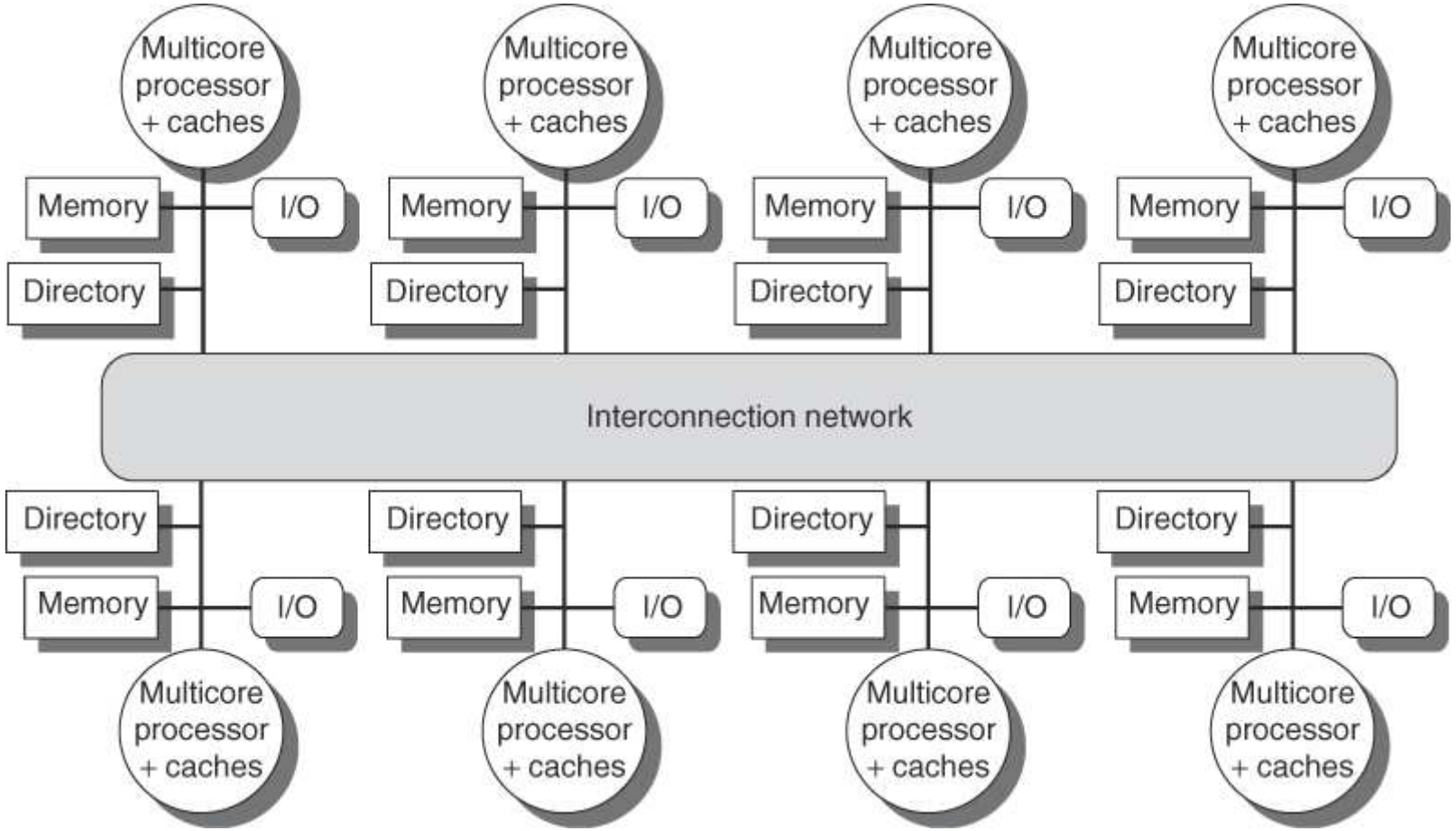
Cache 1 should transition from S->M to I->M and send InvResp

Optimization: 3-hop Protocols

- Reduce latency by having a neighbor cache forward data to requester



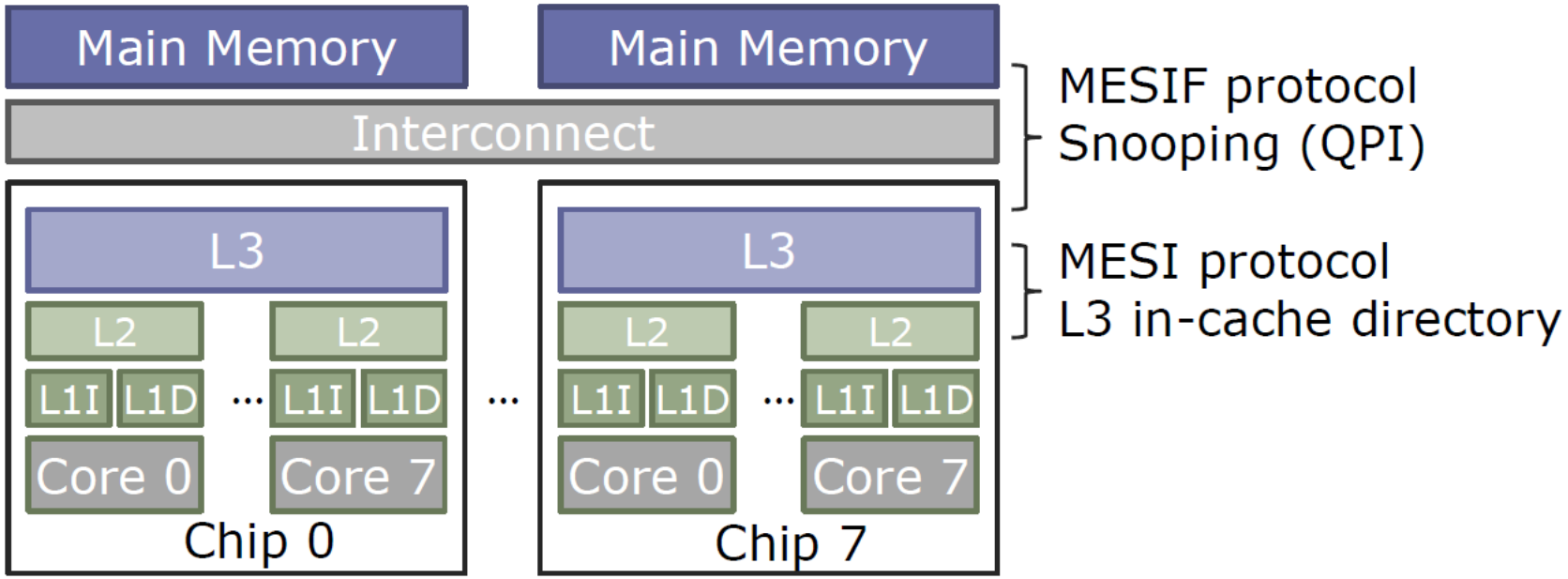
Distributed Directory Implementation



A directory is added to each node to implement cache coherence

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)



Coherence vs Consistency

- Cache coherence makes private caches invisible to software
  - Concerns reads/writes to a single memory location

- Memory consistency models precisely specify how memory behaves with respect to read and write operations from multiple processors
  - Concerns reads/writes to multiple memory locations

Why Consistency Matters

Initial memory contents

a: 0

flag: 0

Processor 1	*Processor 2*
Store (a), 10;	L: Load r1, (flag);
Store (flag), 1;	if r1 == 0 goto L;
	Load r2, (a);

- What value does r2 hold after both processors finish running this code?
  - It depends on the order in which processor 2 observes processor 1's stores!
  - 10 if Store (flag) > Store (a); or 0 otherwise

Sequential Consistency (SC)

" 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"

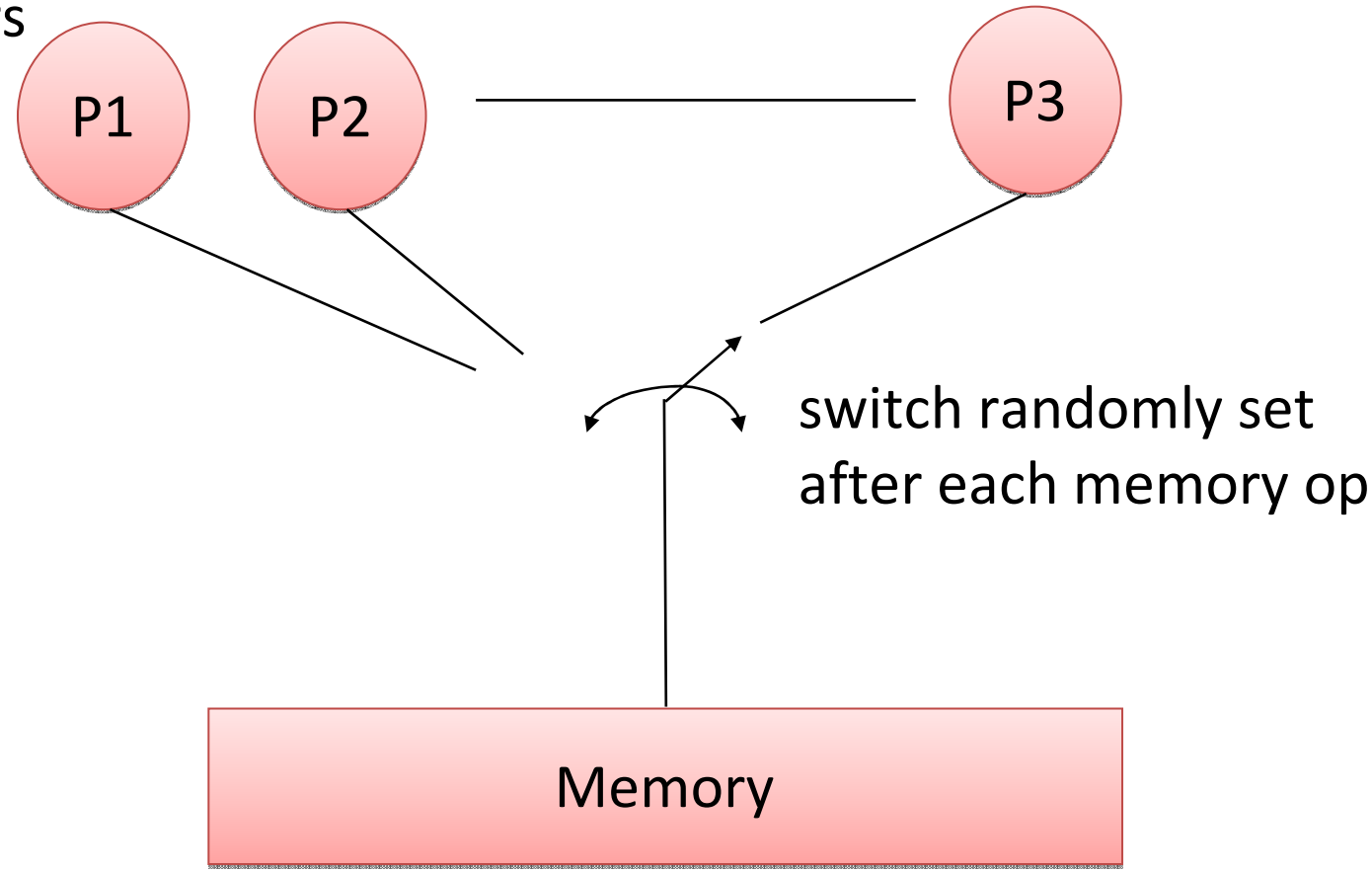
- Leslie Lamport

Sequential Consistency =

arbitrary *order-preserving interleaving* of memory references of sequential programs

Sequential Consistency

processors issue memory ops in program order



Sequential Consistency

- All loads and stores in order.
- Delay completion of memory access until all invalidations caused by that access complete.
- Delay next memory access until previous one completes
  - E.g., Delay read until previous write completes.
  - Cannot place writes in a write buffer and continue with read!
- Simple for programmer.
- Not much room for HW/SW performance optimizations.

Memory Model Issues

Architectural optimizations that are correct for uniprocessors often violate sequential consistency and result in a new memory model for multiprocessors

Relaxed Consistency Models

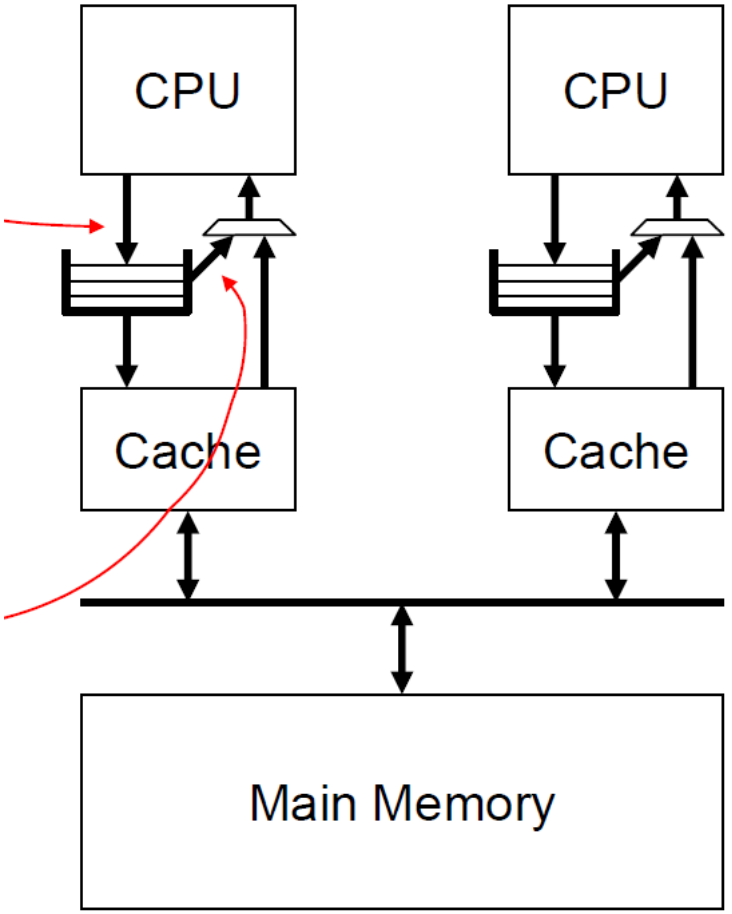
- Sequential Consistency (SC):
  - R → W, R → R, W → R, W → W

X → Y
X must complete before Y

- *Total Store Ordering* (TSO) relaxes W → R
  - R → W, R → R, W → W
- *Partial Store Ordering* (PSO) relaxes W → W
  - R → W, R → R
- *Weak Ordering* or *Release Consistency* (RC)
  - All ordering explicitly declared
    - Use *fences* to define boundaries
    - Use *acquire and release* to force flushing of values

Committed Store Buffers

- CPU can continue execution while earlier committed stores are still propagating through memory system
  - Processor can commit other instructions (including loads and stores) while first store is committing to memory
  - Committed store buffer can be combined with speculative store buffer in an out-of-order CPU
- Local loads can bypass values from buffered stores to same address



Example 1: Store Buffers

Initially, all memory locations contain zeros

Processor 1	*Processor 2*
Store (flag1),1;	Store (flag2),1;
Load r1, (flag2);	Load r2, (flag1);

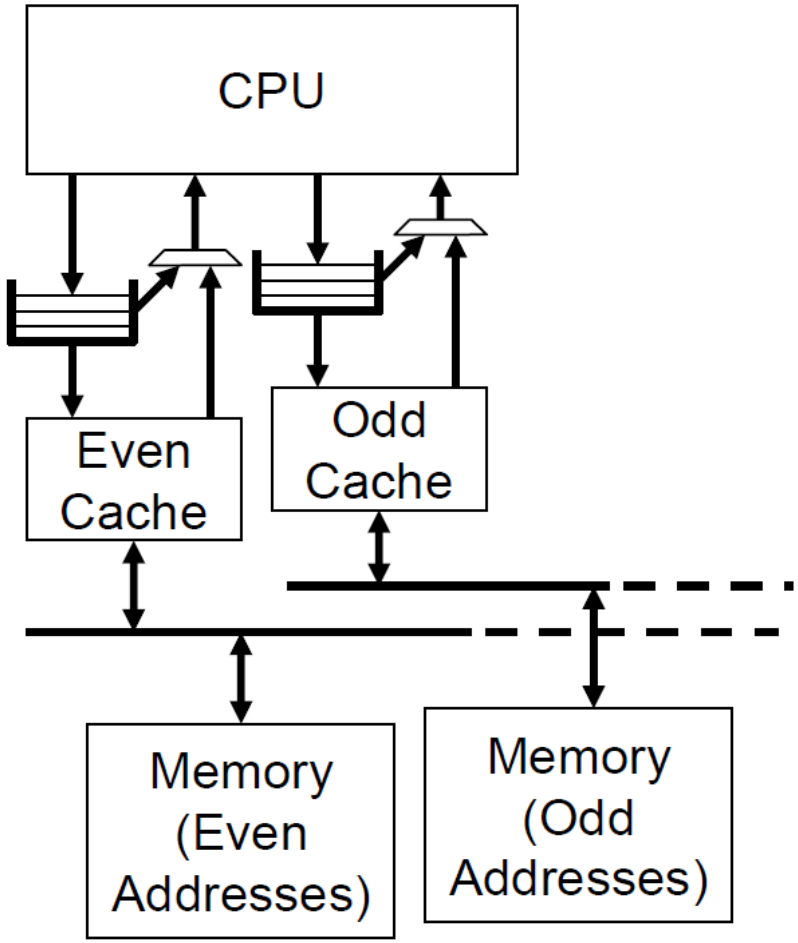
Question: Is it possible that r1 = 0 and r2 = 0?

- Sequential consistency: No
- Suppose Loads can go ahead of Stores waiting in the store buffer: Yes !

  Total Store Order (TSO): Sun SPARC, IBM 370

Interleaved Memory System

- Achieve greater throughput by spreading memory addresses across two or more parallel memory subsystems
  - In snooping system, can have two or more snoops in progress at same time (e.g., Sun UE10K system has four interleaved snooping busses)
  - Greater bandwidth from main memory system as two memory modules can be accessed in parallel



Example 2: Non-FIFO Store buffers

Initially, all memory locations contain zeros

Processor 1	*Processor 2*
Store (a), 1;	Load r1, (flag);
Store (flag), 1;	Load r2, (a);

Question: Is it possible that r1 = 1 but r2 = 0?

- Sequential consistency: No
- With Non-FIFO store buffers: Yes

  Partial Store Ordering (PSO): Sun SPARC

Example 3: Non-Blocking Caches

Initially, all memory locations contain zeros

Processor 1	*Processor 2*
Store (a), 1;	Load r1, (flag);
Store (flag), 1;	Load r2, (a);

Question: Is it possible that r1 = 1 but r2 = 0 if stores are ordered?

- Sequential consistency: No
- With Non-Blocking Caches: Yes because Loads can be reordered

  Weak Ordering (WO); Release Consistency (RC)

Understanding Relaxed Consistency

Three important concepts

- **Atomicity**
  - do writes appear at the same time to all other processors?
- **Program order**
  - do my references have to be ordered with respect to each other?
- **Visibility (causality)**
  - Does anyone care? This is the most subtle...

Weaker (Relaxed) Memory Models

- Hard to understand and remember
- Unstable
- Abandon weaker memory models in favor of implementing SC.

Implementing SC

- The memory operations of each individual processor appear to all processors in the order the requests are made to the memory.
  - *Provided by cache coherence*, which ensures that all processors observe the same order of loads and stores to an address

- Any execution is the same as if the operations of all the processors were executed in some sequential order
  - Provided by enforcing a dependence between each memory operation and the following one

SC Data Dependence

- *Stall*
  - *Use in-order execution and blocking caches*
    - *Cache coherence plus allowing a processor to have only one request in flight at a time will provide SC*
- *Change architecture => Relaxed memory models*
  - *Use OOO and non-blocking caches*
    - *Cache coherence and allowing multiple concurrent requests (to different addresses) gives high performance*
    - *Add fence operations to force ordering when needed*
- *Speculate…*

Properly Synchronized Programs

- Very few programmers do programming that relies on SC; instead, they use higher-level synchronization primitives
  - locks, semaphores, monitors, atomic transactions

- A “properly synchronized program” is one where each shared writable variable is protected (say, by a lock) so that there is no race in updating the variable
  - There is still race to get the lock
  - There is no way to check if a program is properly synchronized

- For properly synchronized programs, instruction reordering does not matter as long as updated values are committed before leaving a locked region

Takeaways

- SC is too low level a programming model. High level programming should be based on critical sections & locks, atomic transactions, monitors, ...

- High-level parallel programming should be oblivious of memory model issues
  - Programmer should not be affected by changes in the memory model

- ISA definition for Load, Store, Memory Fence, synchronization instructions should
  - Be precise
  - Permit maximum flexibility in hardware implementation
  - Permit efficient implementation of high-level parallel constructs