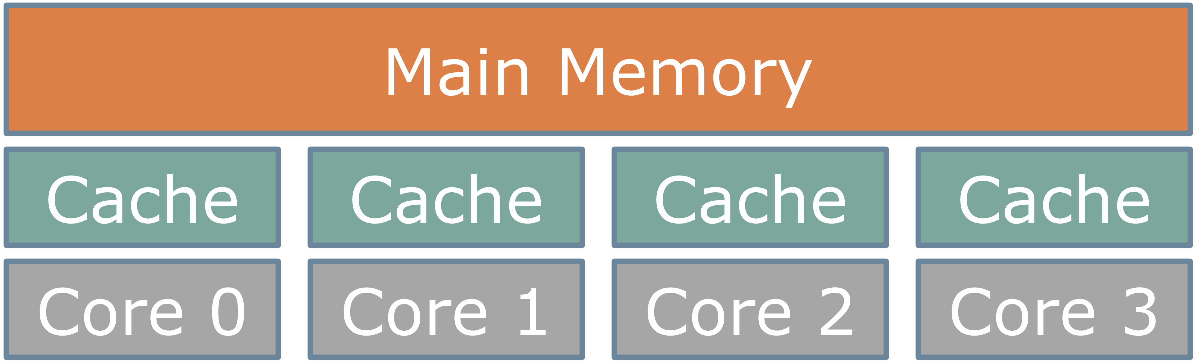
Cache Coherence

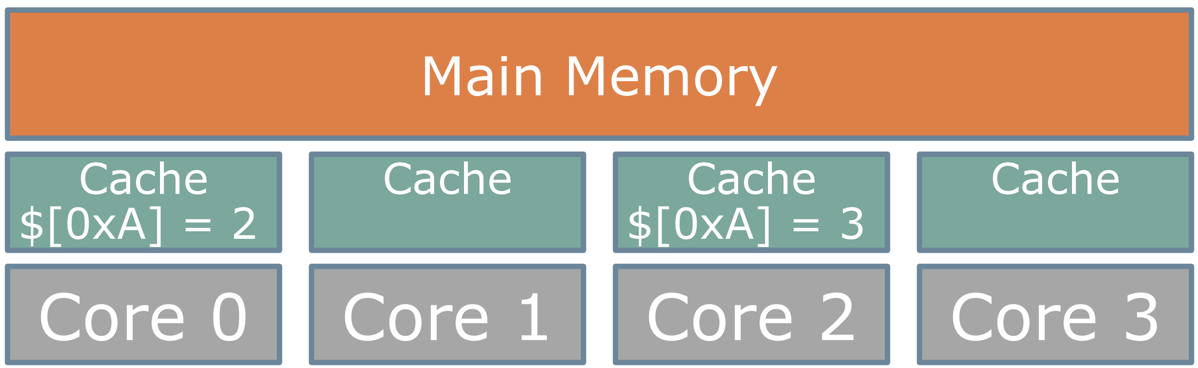
Multicores



- Modern microprocessors usually have 2 to 8 cores where each core has a private cache for performance
- Cores can be used cooperatively to speed up an application
- Cores communicate with each other via memory

Cache Coherence Avoids Stale Data

- Need to provide the illusion of a single shared memory even though multicores have multiple private caches
- Problem:



1. LD 0xA → 2
2. ST 3 → 0xA
3. LD 0xA → 2 (stale!)

- Solution: A cache coherence protocol controls cache contents to avoid stale lines
  - e.g., invalidate core 0’s copy of A before letting core 2 write to it

Maintaining Coherence

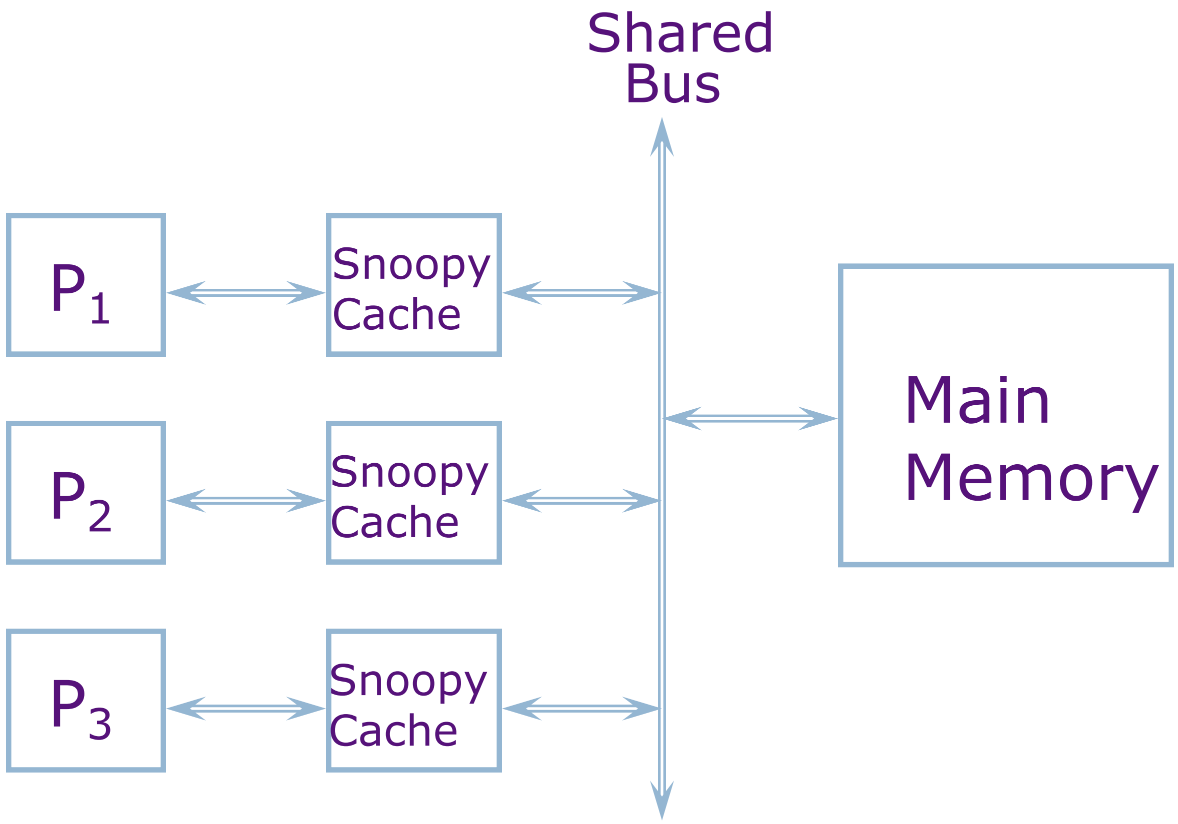
- In a *coherent memory* all loads and stores can be placed in a global order
  - multiple copies of an address in various caches can cause this property to be violated

- This property can be ensured if:
  - Only one cache at a time has the write permission for an address
  - No cache can have a stale copy of the data after a write to the address has been performed

Implementing Cache Coherence

- Coherence protocols must enforce two rules:
  - **Write propagation**: Writes eventually become visible to all processors
  - **Write serialization**: Writes to the same location are serialized (all processors see them in the same order)
- How to ensure write propagation?
  - **Write-invalidate protocols**: Invalidate all other cached copies before performing the write
  - **Write-update protocols**: Update all other cached copies after performing the write
- How to ensure write serialization?
  - **Snooping-based protocols**: All caches observe each other's actions through a shared bus
  - **Directory-based protocols**: A coherence directory tracks contents of private caches and serializes requests

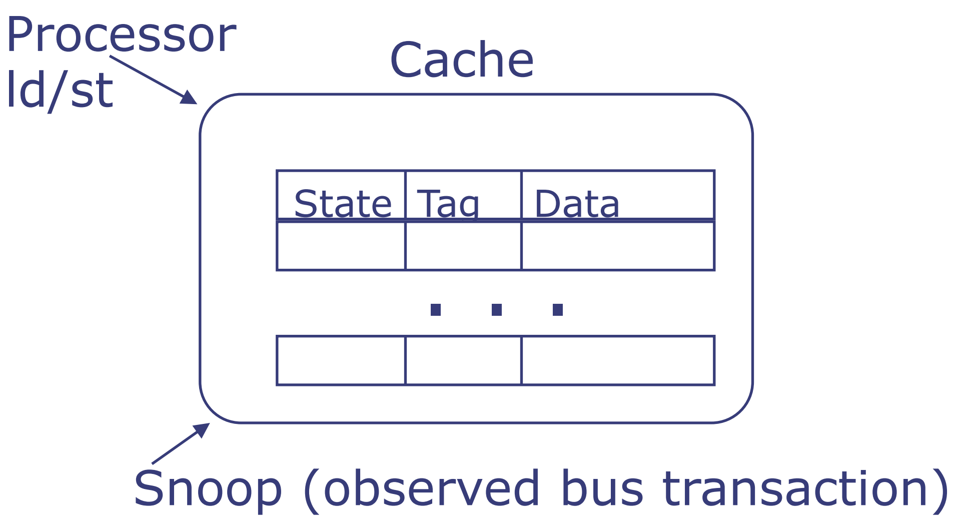
Snooping-Based Coherence [Goodman 1983]



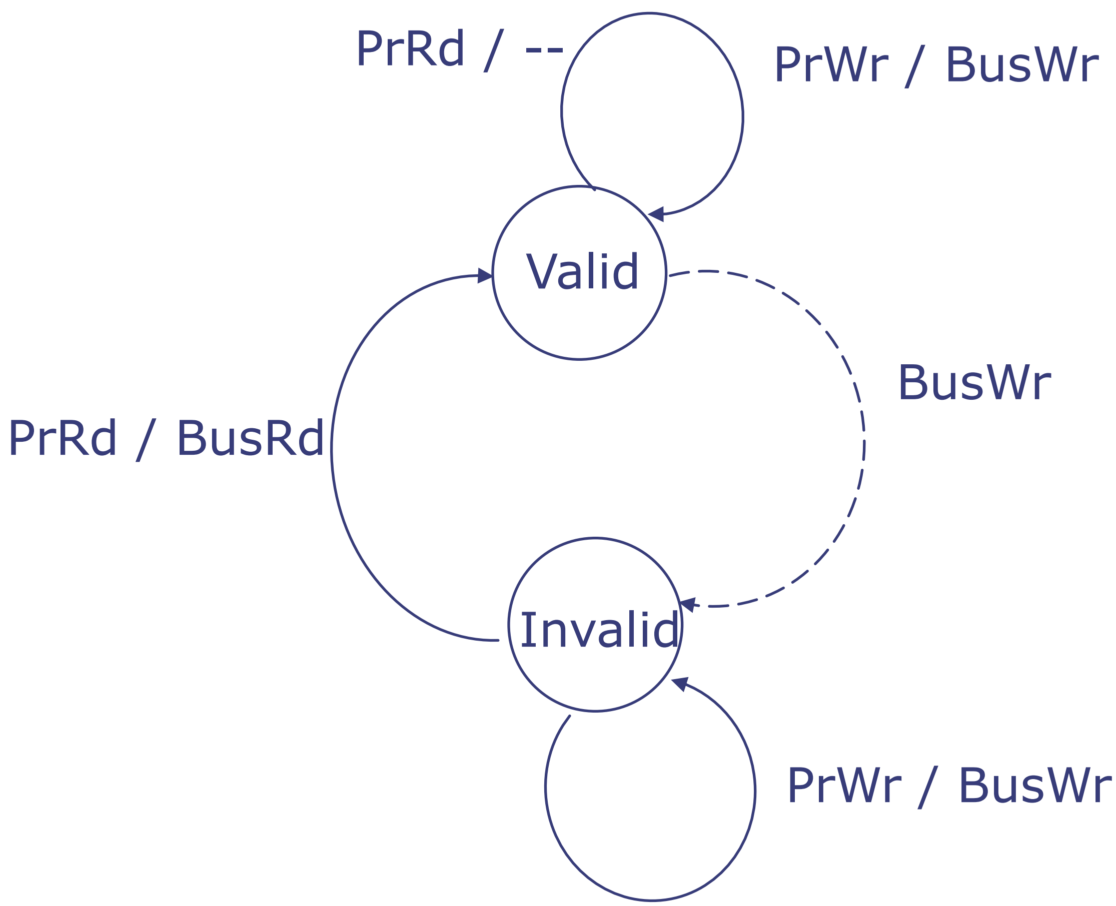
Caches watch (snoop on) bus to keep all processors' view of memory coherent

Snooping-Based Coherence

- Bus provides serialization point
  - Broadcast, totally ordered
  - Each cache controller "snoops" all bus transactions
  - Controller updates state of cache in response to processor and snoop events and generates bus transactions
- Snoopy protocol (FSM)
  - State-transition diagram
  - Actions



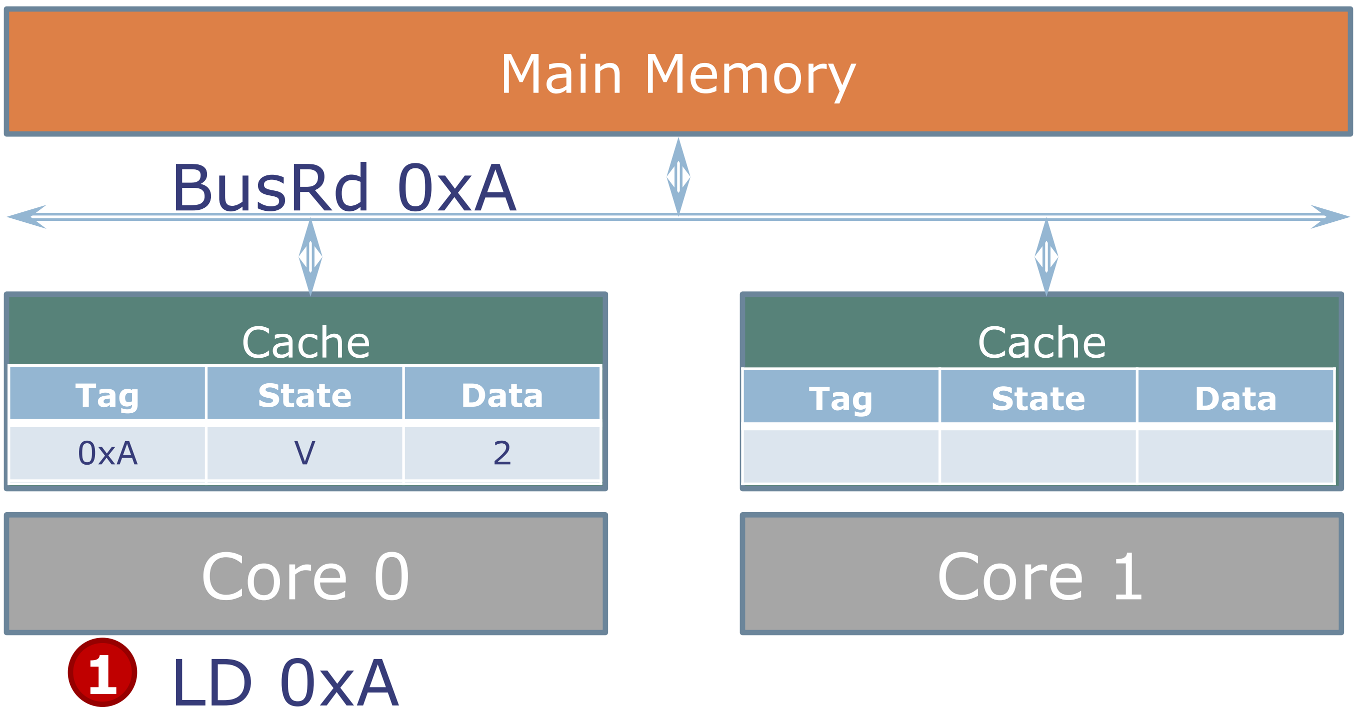
A Simple Protocol: Valid/Invalid (VI)



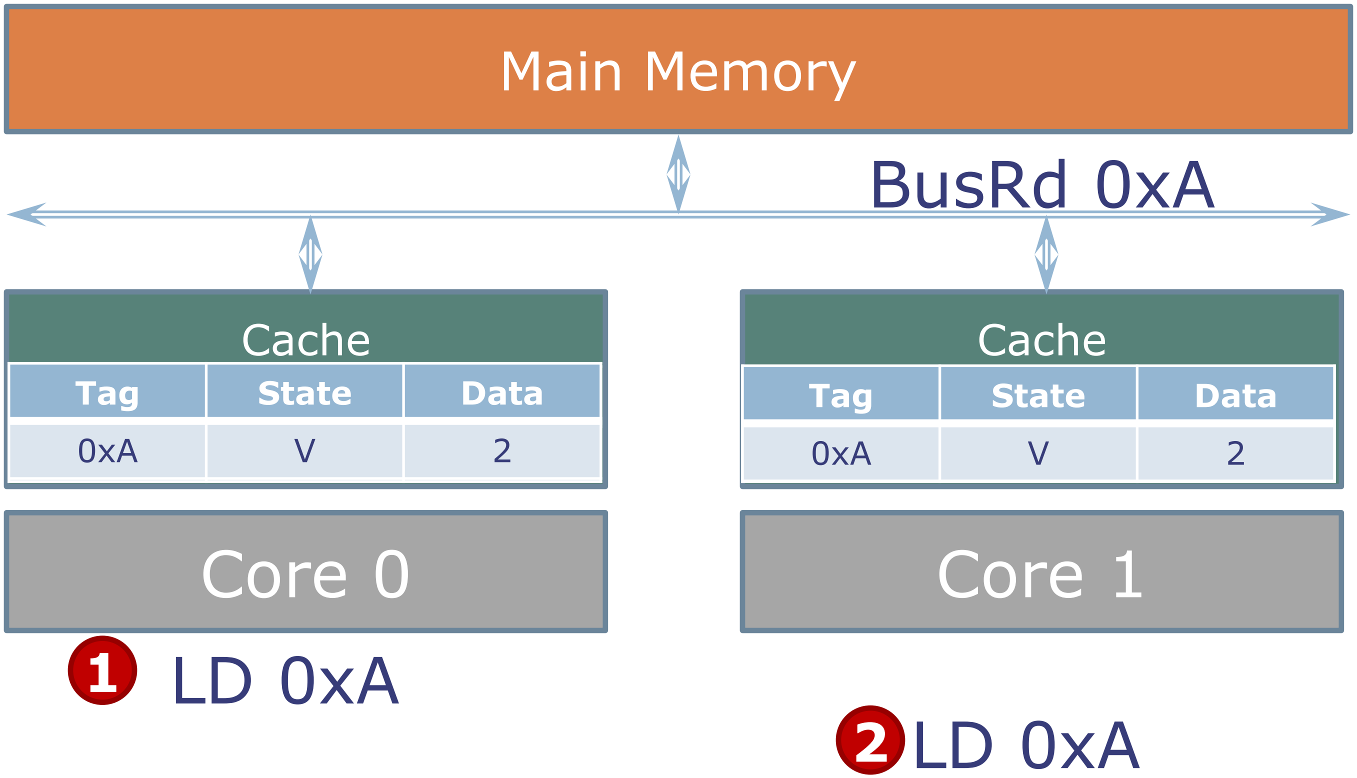
- Assume write-through caches

Actions
Processor Read (PrRd)
Processor Write (PrWr)
Bus Read (BusRd)
Bus Write (BusWr)

Valid/Invalid Example

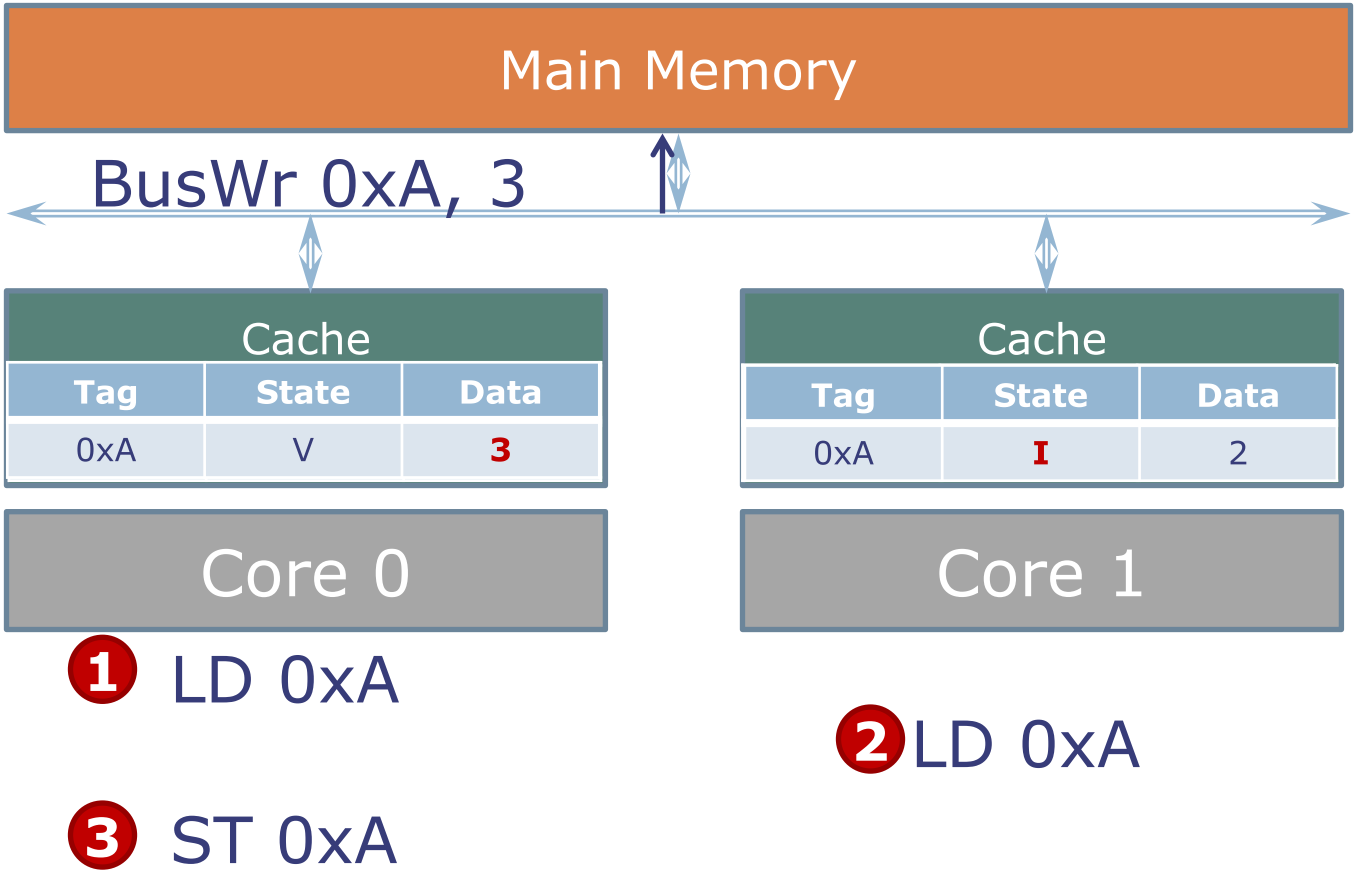


Valid/Invalid Example

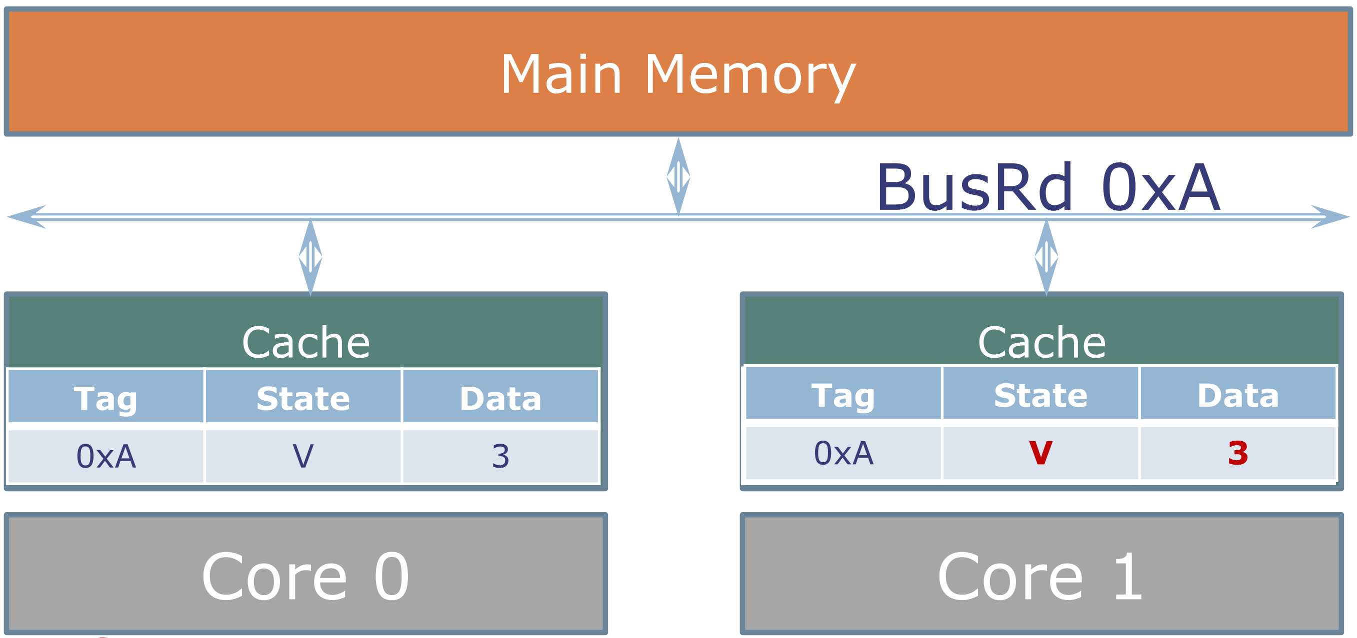


Additional loads satisfied locally, without BusRd

Valid/Invalid Example



Valid/Invalid Example



1. LD 0xA (Core 0)
2. LD 0xA (Core 1)
3. ST 0xA (Core 0)
4. LD 0xA (Core 1)

VI Problems?

Every write updates main memory
Every write requires broadcast & snoop

Modified/Shared/Invalid (MSI) Protocol

- Each line in each cache maintains MSI state:

  I - cache doesn't contain the address

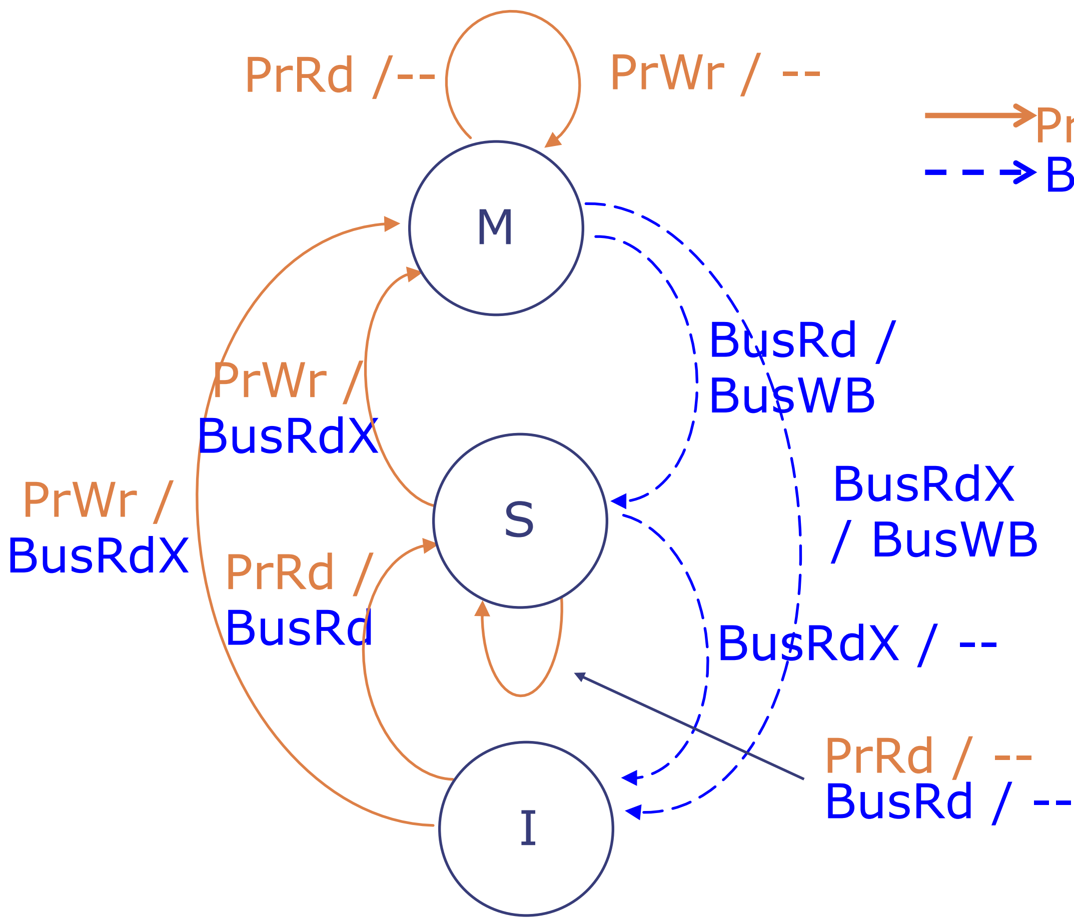
  S - cache has the address but so may other caches; hence it can only be read

  M - only this cache has the address; hence it can be read and written

  - any other cache that had this address got invalidated

MSI Protocol FSM

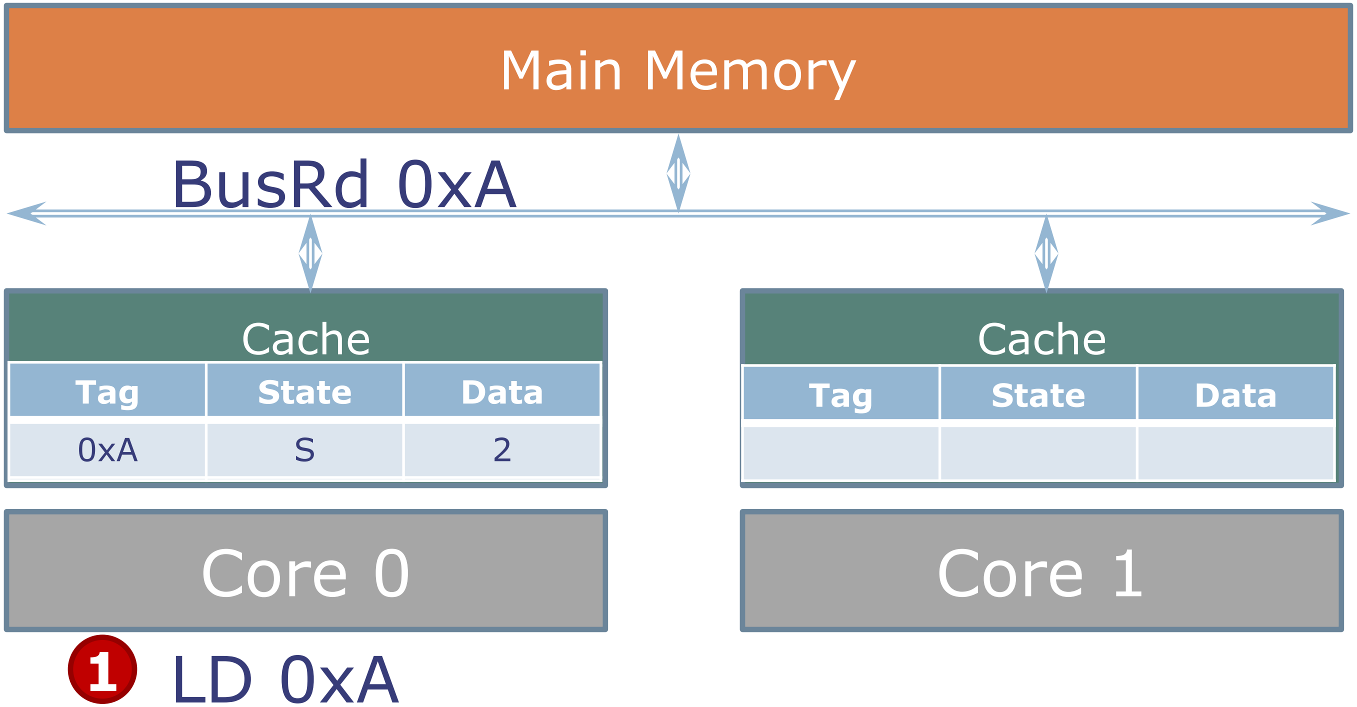
- VI Drawbacks: Every write updates main memory, and every write requires broadcast & snoop
- MSI: Allows writeback caches + satisfies writes locally



Processor-initiated transitions
Bus-initiated transitions

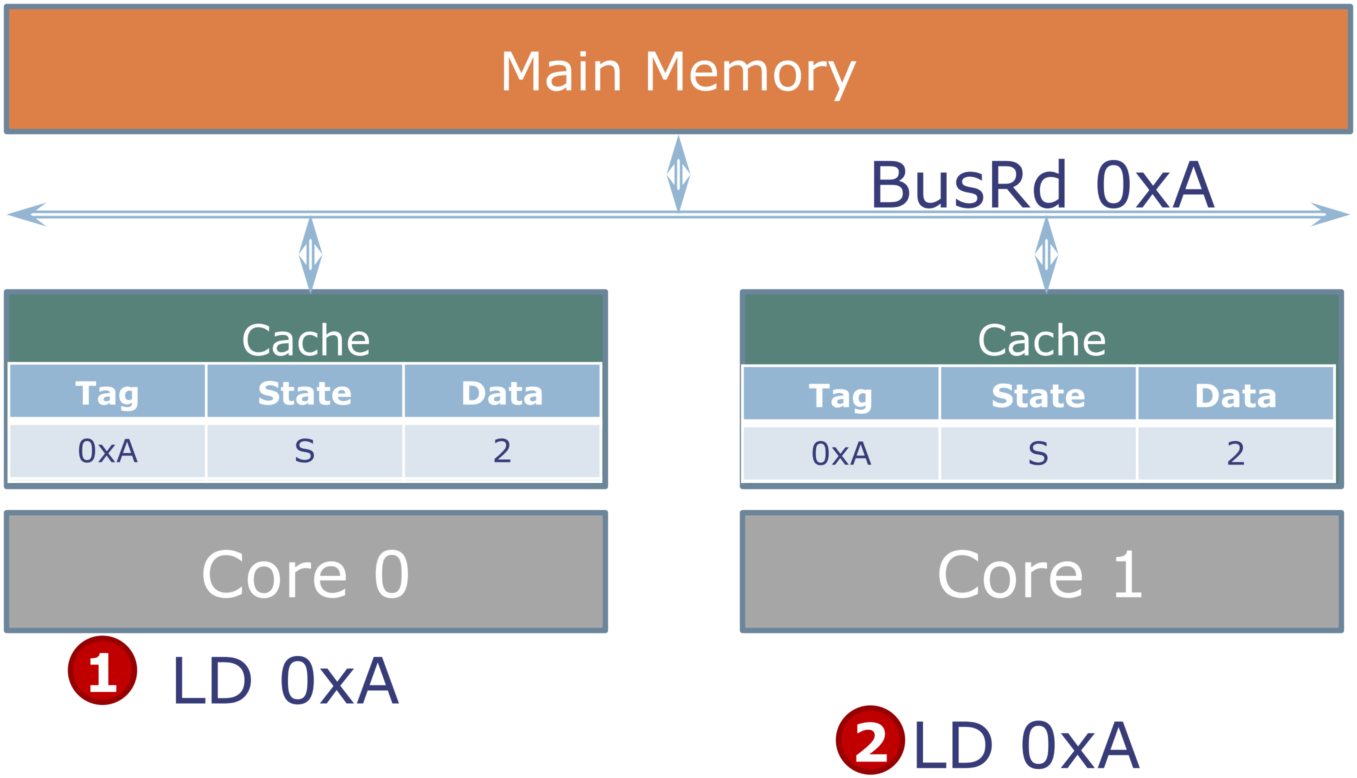
Actions
Processor Read (PrRd)
Processor Write (PrWr)
Bus Read (BusRd)
Bus Read Exclusive (BusRdX)
Bus Writeback (BusWB)

MSI Example



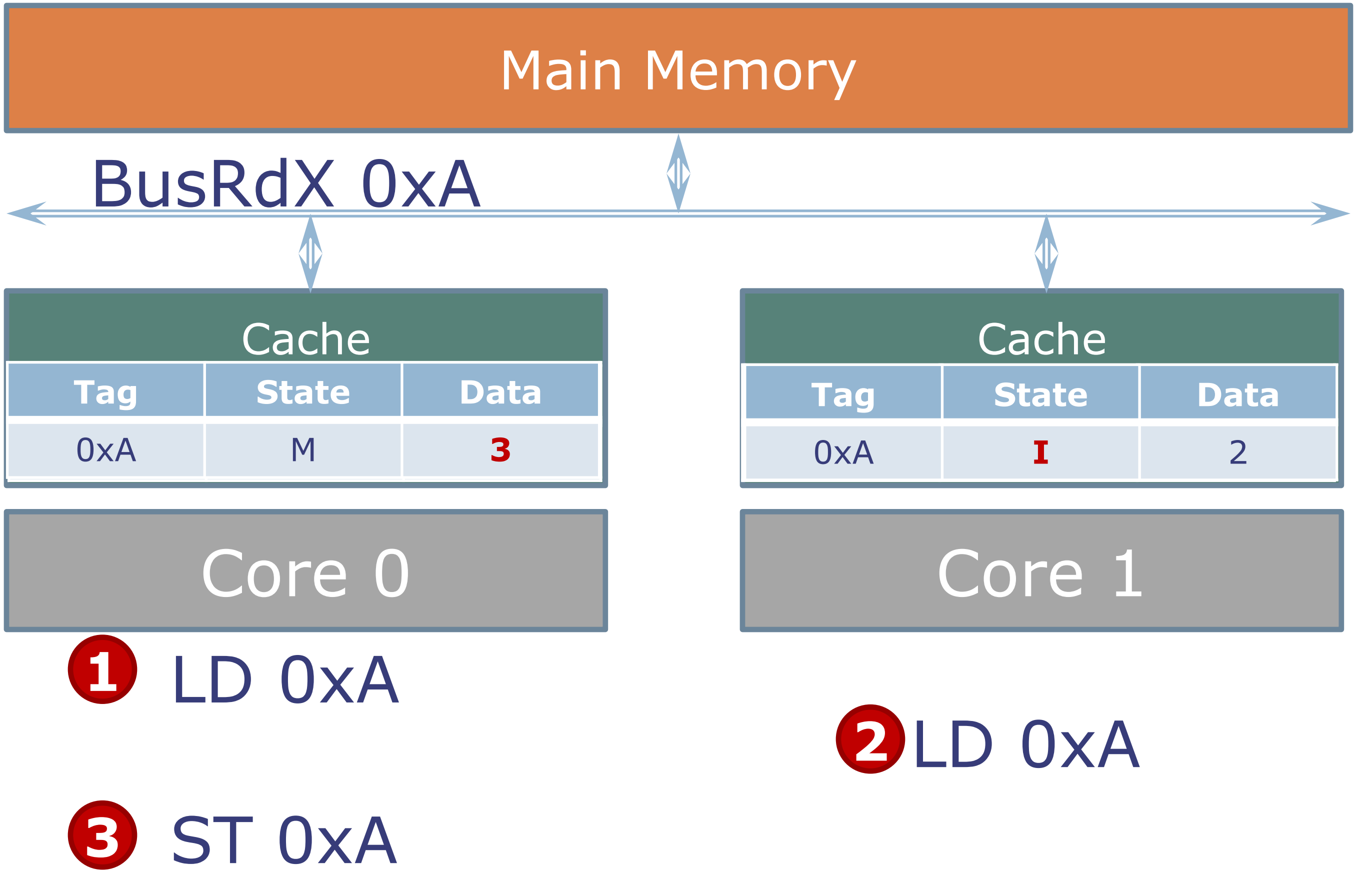
Main Memory
BusRd 0xA
Cache
Tag
State
Data
0xA
S
2
Cache
Tag
State
Data
Core 0
Core 1
1
LD 0xA

MSI Example



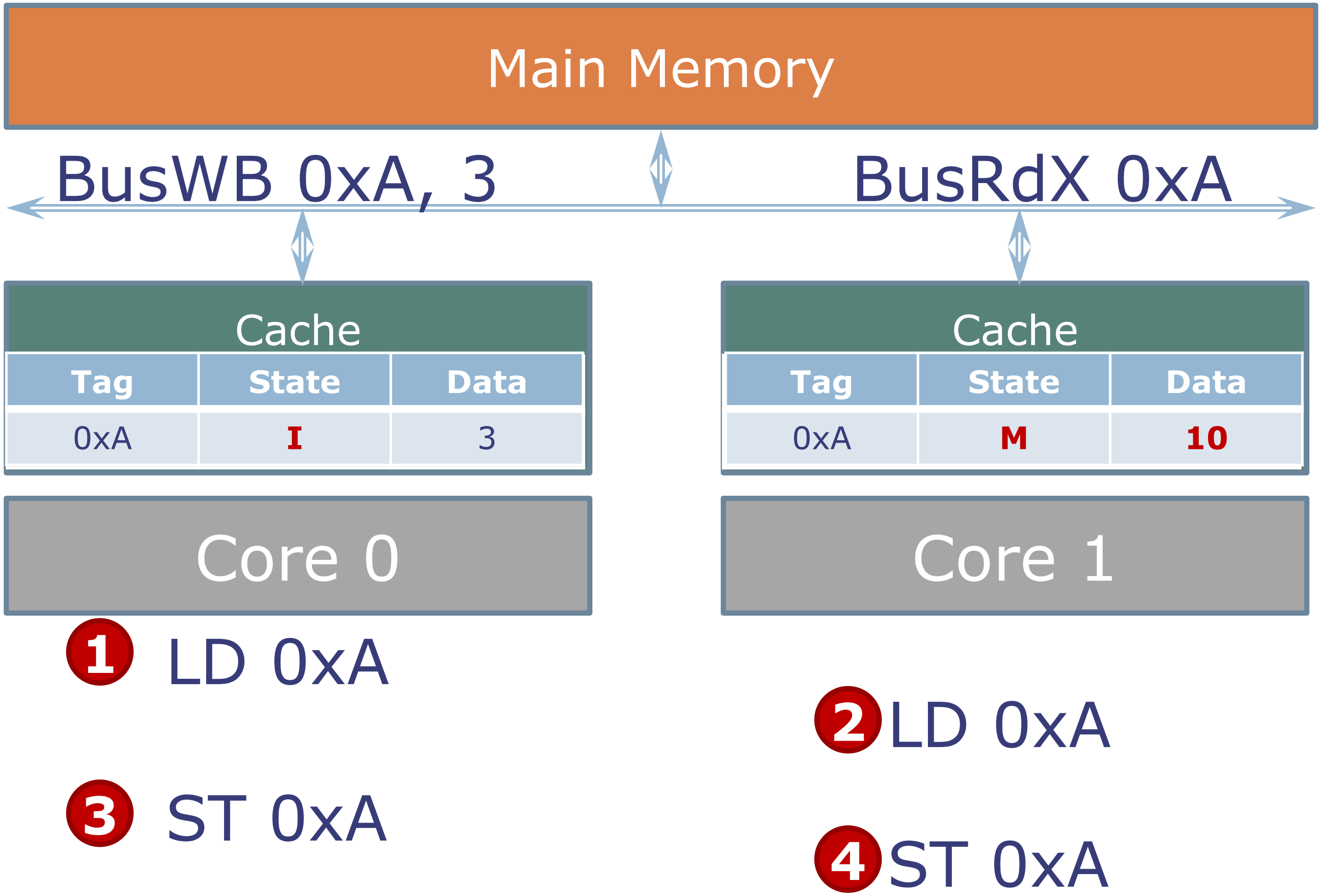
Additional loads satisfied locally, without BusRd (like in VI)

MSI Example



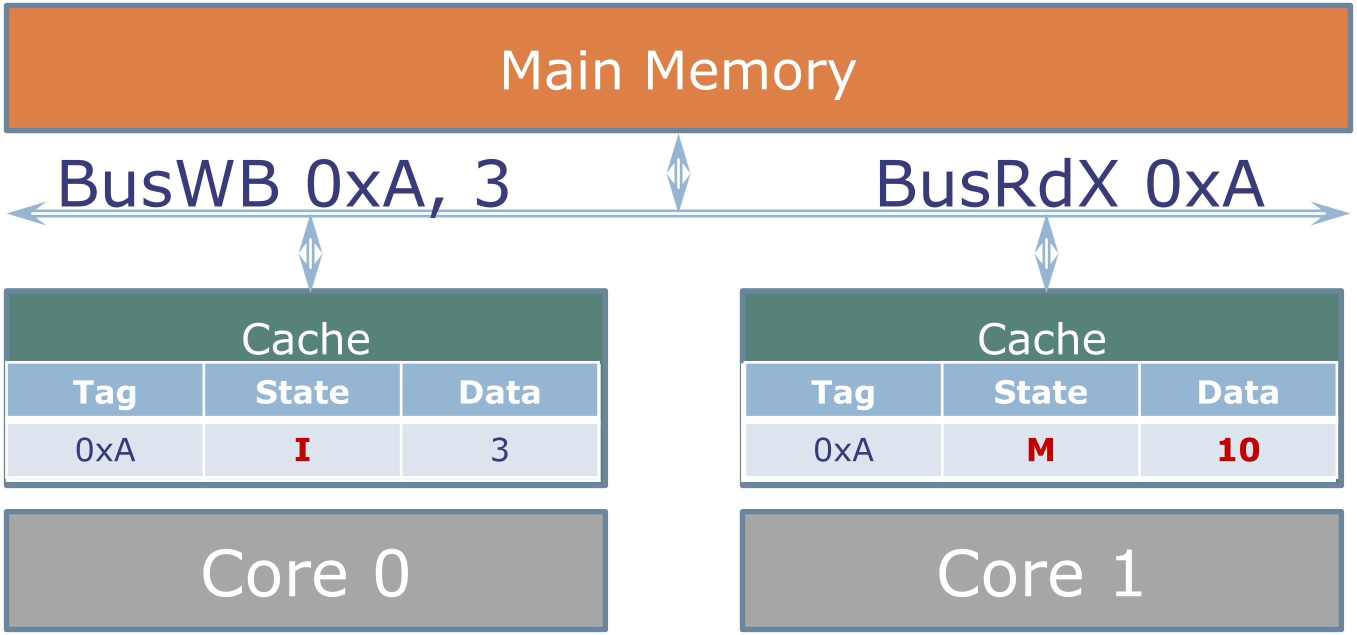
Additional loads *and stores* from core 0 satisfied locally, without bus transactions (unlike in VI)

MSI Example



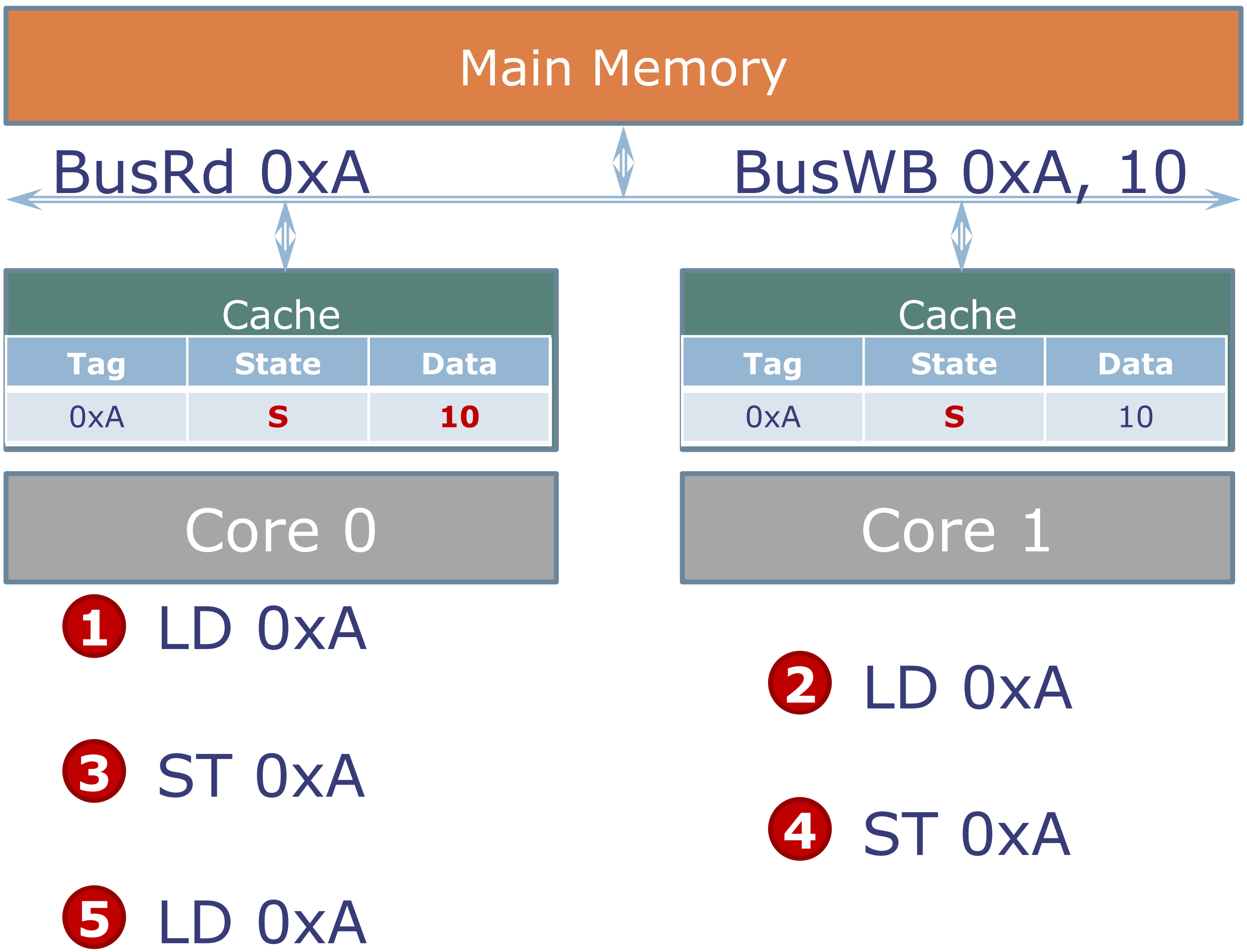
Main Memory
BusWB 0xA, 3
BusRdX 0xA
Cache
Tag State Data
0xA I 3
Cache
Tag State Data
0xA M 10
Core 0
Core 1
1 LD 0xA
3 ST 0xA
2 LD 0xA
4 ST 0xA

Cache Interventions



- MSI lets caches serve writes without updating memory, so main memory can have stale data
  - Core 0's cache needs to supply data
  - But main memory may also respond!
- Cache must override response from main memory

MSI Example



Main Memory
BusRd 0xA
BusWB 0xA, 10
Cache
Tag
State
Data
0xA
S
10
Cache
Tag
State
Data
0xA
S
10
Core 0
Core 1
1 LD 0xA
2 LD 0xA
3 ST 0xA
4 ST 0xA
5 LD 0xA

MSI Optimizations: Exclusive State

- Observation: Doing read-modify-write sequences on private data is common
  - What's the problem with MSI?

    2 bus transactions for every read-modify-write of private data.

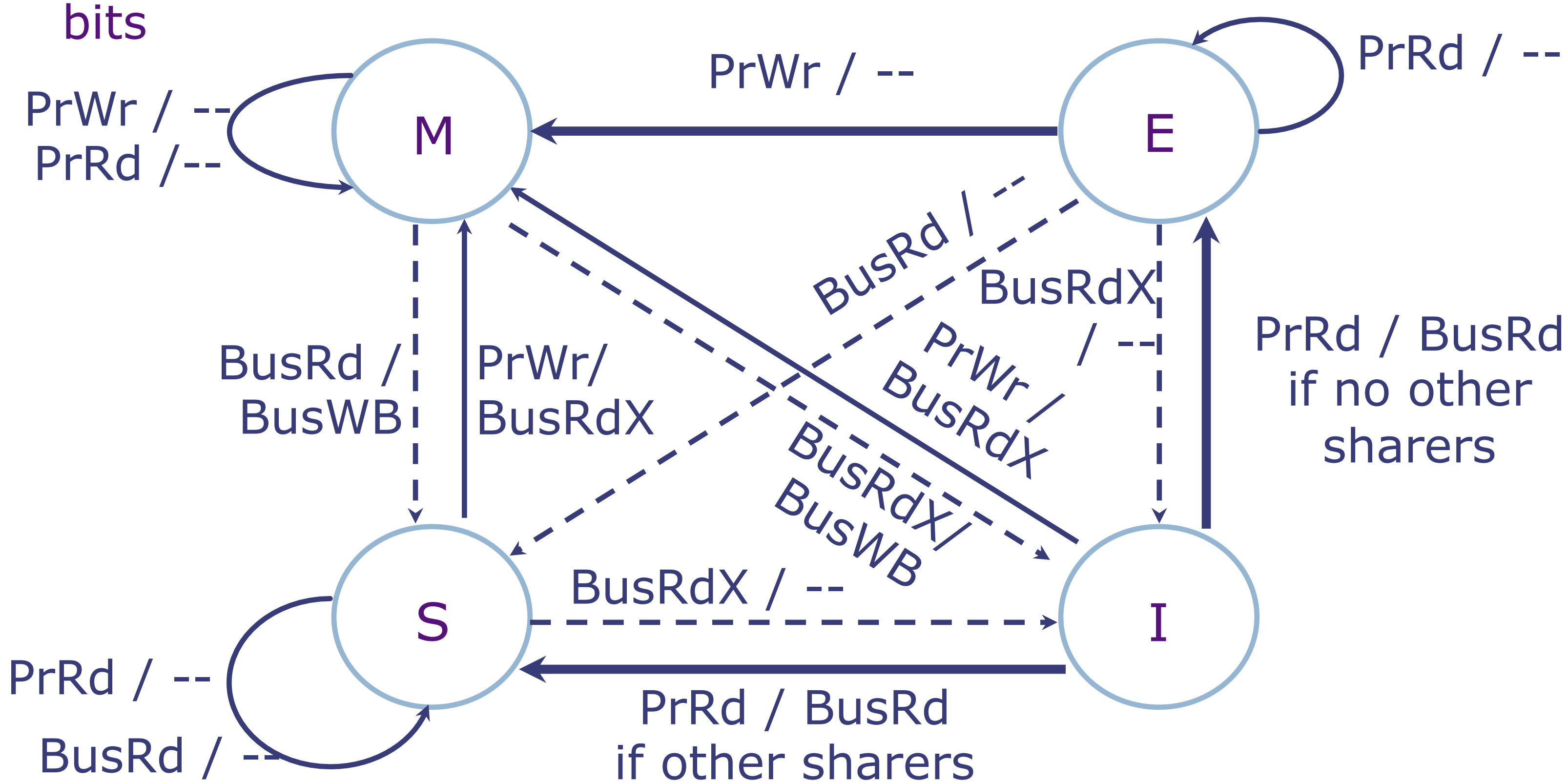
- Solution: E state (exclusive, clean)
  - If no other sharers, a read acquires line in E instead of S
  - Writes silently cause E→M (exclusive, dirty)

MESI: An Enhanced MSI protocol

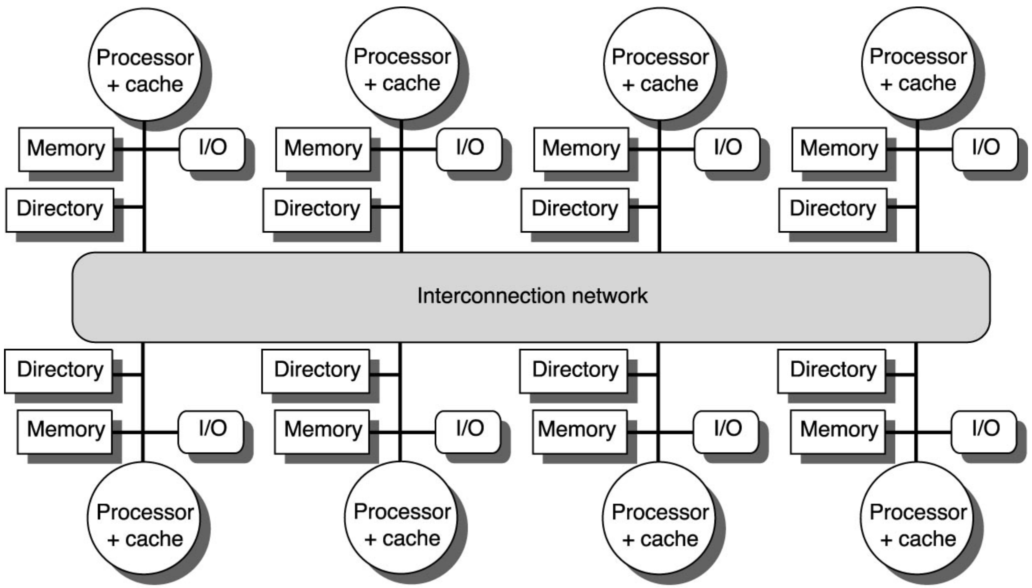
increased performance for private read-write data

Each cache line has a tag

M: Modified Exclusive
E: Exclusive, unmodified
S: Shared
I: Invalid



Directory-Based Coherence



- Route all coherence transactions through a directory
  - Tracks contents of private caches → No broadcasts
  - Serves as ordering point for conflicting requests → Unordered networks

Cache Coherence and False Sharing

Performance Issue #1

- A cache line contains more than one word, and cache coherence is done at line granularity

state	line addr	word0	word1	...	wordN

- Suppose P_1 writes $word_i$ and P_2 writes $word_k$ and both words have the same line address
- What can happen?

The line may be invalidated (ping-pong) many times unnecessarily because addresses are in the same line.

Thank you!

Next lecture: Branch Prediction