CSED311 Computer Architecture – Lecture 17

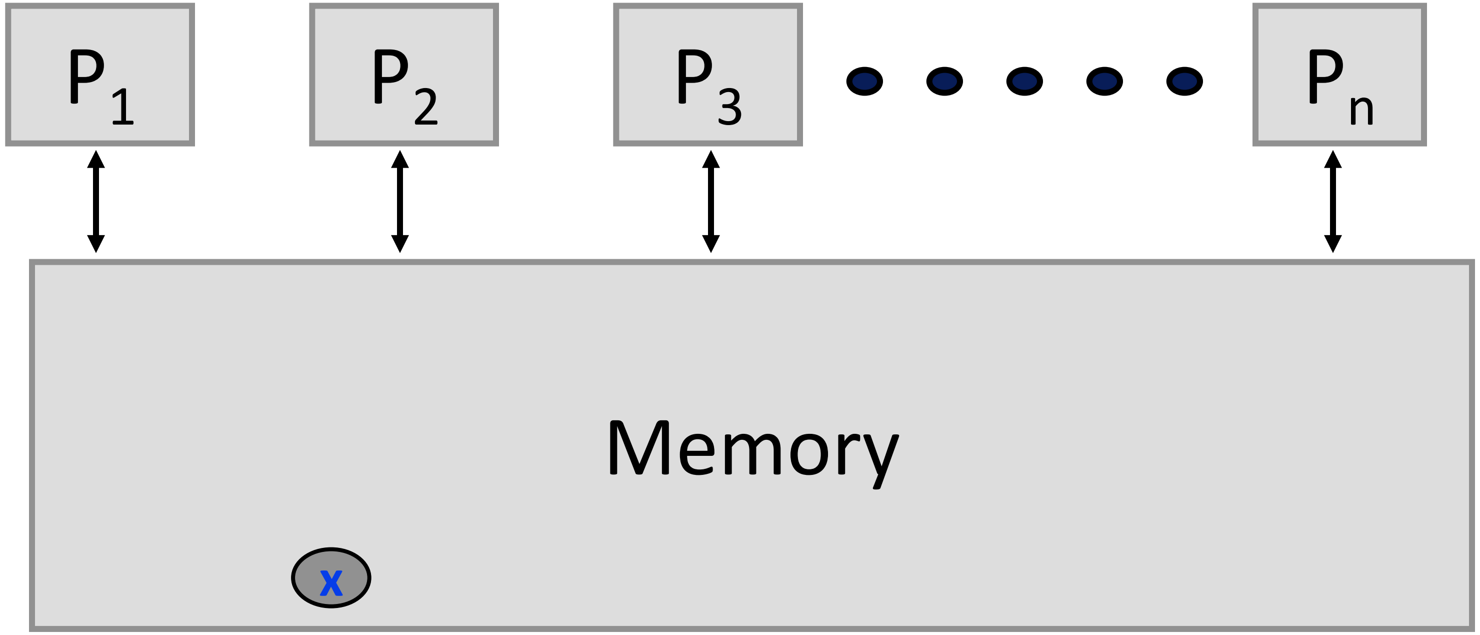
# Cache Coherence

Eunhyeok Park

Department of Computer Science and Engineering
POSTECH

Disclaimer: Slides developed in part by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Mutlu, and Wenisch, Gwangsun Kim @ Carnegie Mellon University, University of Michigan, Purdue University, University of Pennsylvania, University of Wisconsin, and POSTECH

# Shared Memory Multiprocessor Abstraction



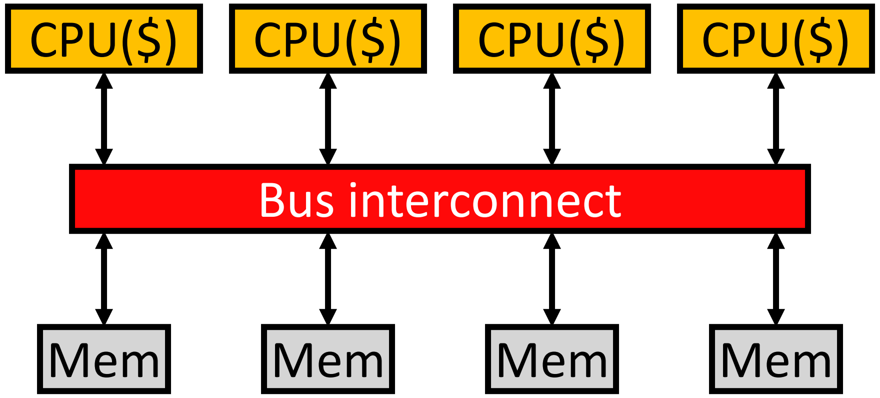
- What is the key advantage of shared-memory multiprocessor (MP) system?

  "Single Physical Address Space"

- Latencies from $P_n$ to Memory

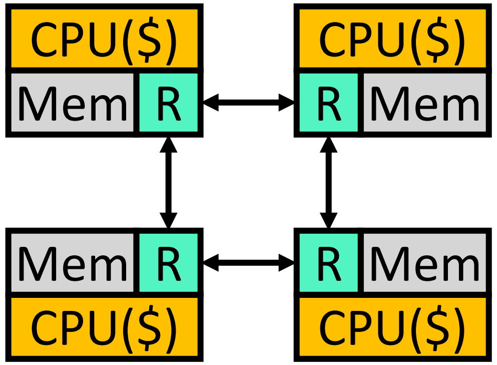
  *UMA (uniform memory access)* vs. *NUMA (non-uniform memory access)*

# Implementation #1: Bus Interconnect



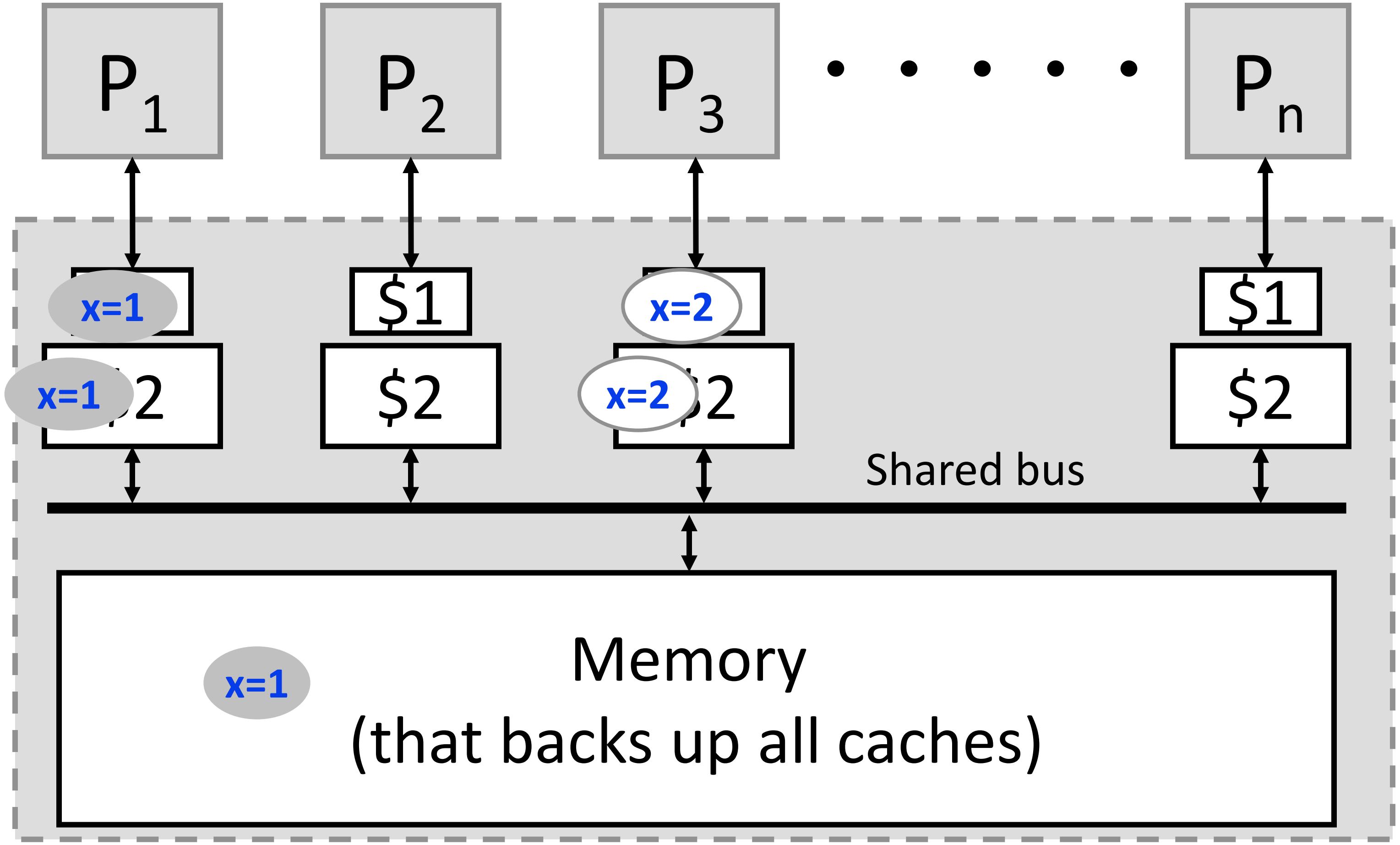
- Bus-based systems
  - Limited scalability: 2–8 processors
  - The bus can be a bandwidth bottleneck
  - Typically, processors split from memories (UMA)
    - Sometimes multiple processors on single chip (CMP)
    - Symmetric multiprocessors (SMPs)
  - Used to be common until ~2006 (e.g., Intel Front-Side Bus)

# Impl. #2: Point-to-Point Interconnect



- General point-to-point network-based systems
  - Typically, processor/memory/router blocks (NUMA)
    - Glueless MP: no need for additional “glue” (or router) chips
  - Better scalability than bus (can scale to even 1000s of processors)
    - Massively parallel processors (MPPs)
  - Used by most dual-socket or quad-socket servers today
    - Intel Ultra-Path Interconnect, AMD Infinity fabric, etc.

# MPs with Private Caches: Cache Coherence?



Assume bus interconnect for now

The goal of "cache coherence" is to make all the processors believe they are connected to the same memory directly (warning: not a formal definition)

# Extreme Solutions to Cache Coherence

#1. No caching of shared variables

#2. Allow multiple copies, but make sure they all have the same value at all times

- Updates to one copy must be visible to all copies wherever they may be (memory and all of the caches)
- Thus, can have **multiple readers and writers** at once

#3. Allow only one copy of a memory location at a time

- If location **x** is cached in one cache then it is not valid in memory or another cache
- Another processor must have a way to find out who has location **x** and take over ownership before reading or writing
- Thus, can only have **one reader/writer** per location

What are the drawbacks of the approaches?

A cache coherence protocol is the “rule” between caches to enforce a particular policy

# #2. “Multiple Identical Copies” Protocol

- A cache line can be either **Valid** or **Invalid**

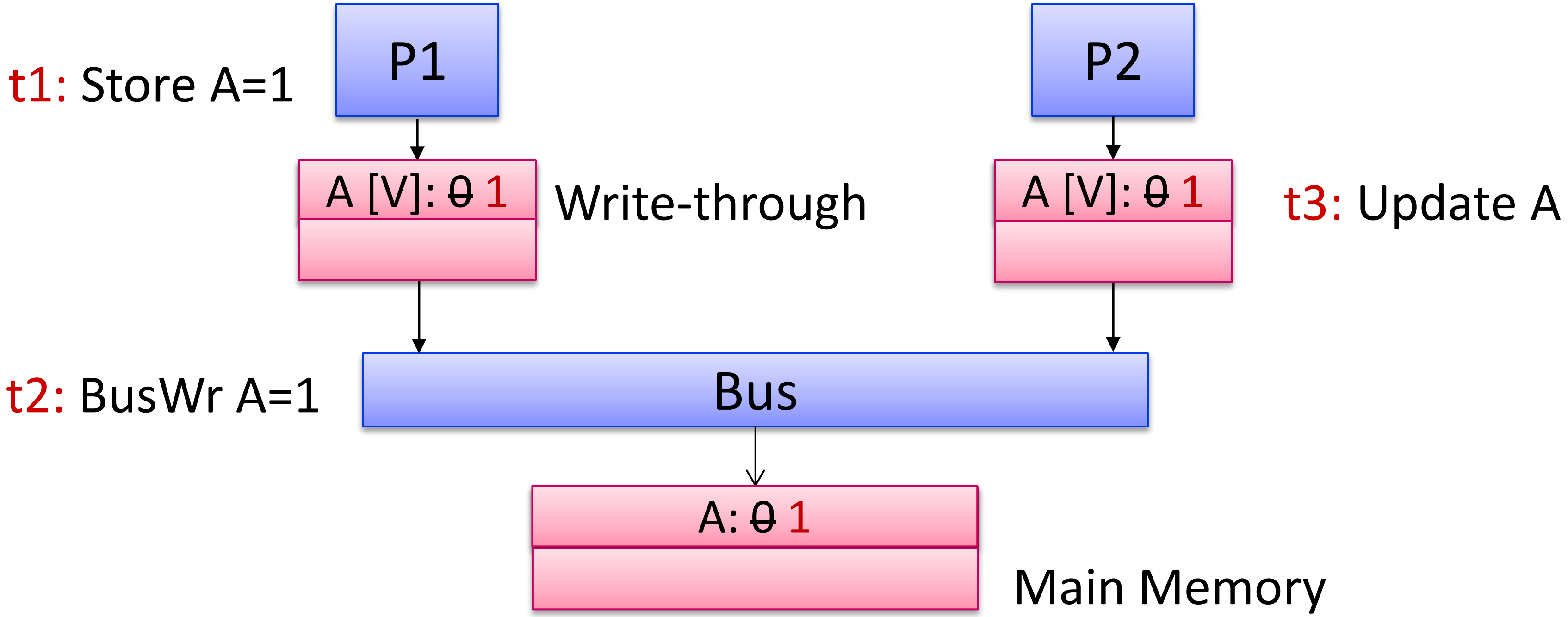
- Based on a write-through (and write-no-allocate) scheme
  - A cache issues a read transaction on a read
  - A cache issues a write transaction to memory whenever its cached line is modified
  - A cache does not need to write back when a line is displaced

- Allows multiple readers, but must write through to bus

- All caches “snoop” the bus for other’s write transactions
  - Check if the write is to a currently cached location
  - Tremendous bus traffic and tag BW (typically 15% of cache accesses are stores)
  - **If a write goes to a cached location**

    **(1) overwrite the old value with the new snooped value (=write-update protocol)**
    **OR (2) delete the old value (=valid-invalid protocol)**

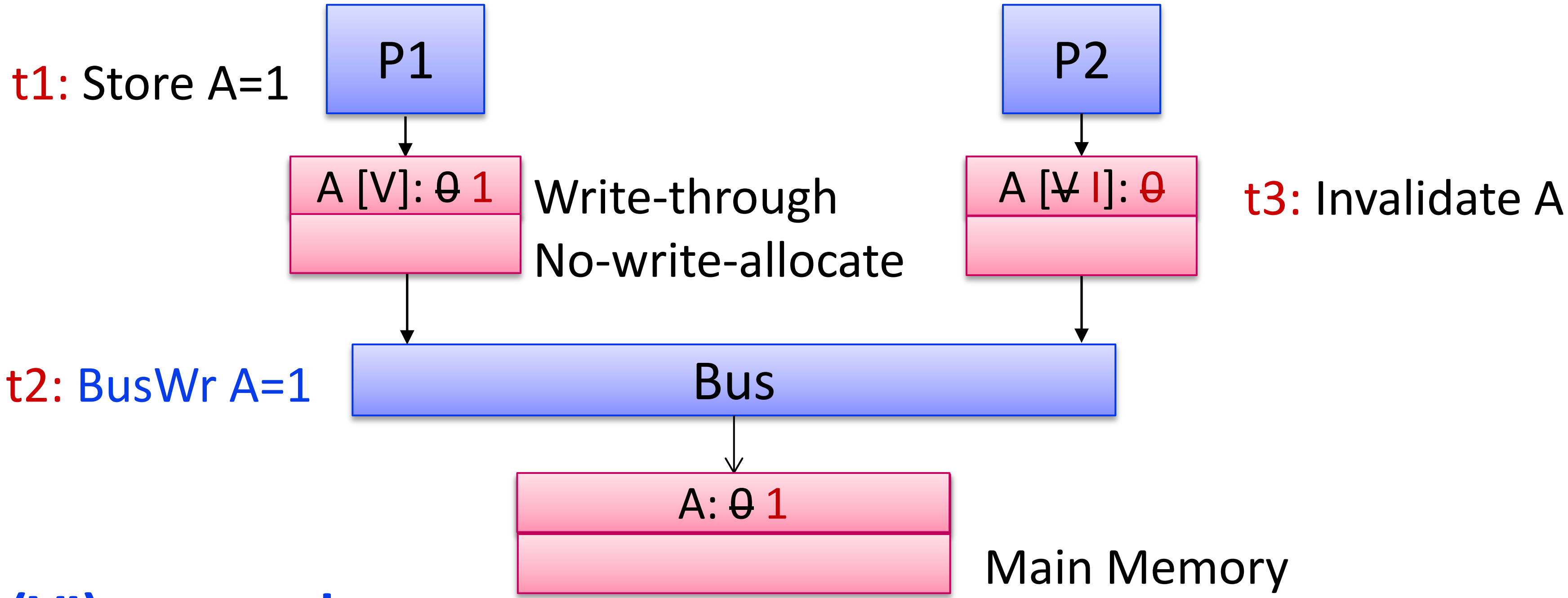
# #2-1: Write-Update Protocol (for Write-Through $)



## Write-Update protocol

- “Update” all values of A in caches and memory
  → support multiple copies (for both reads & writes)

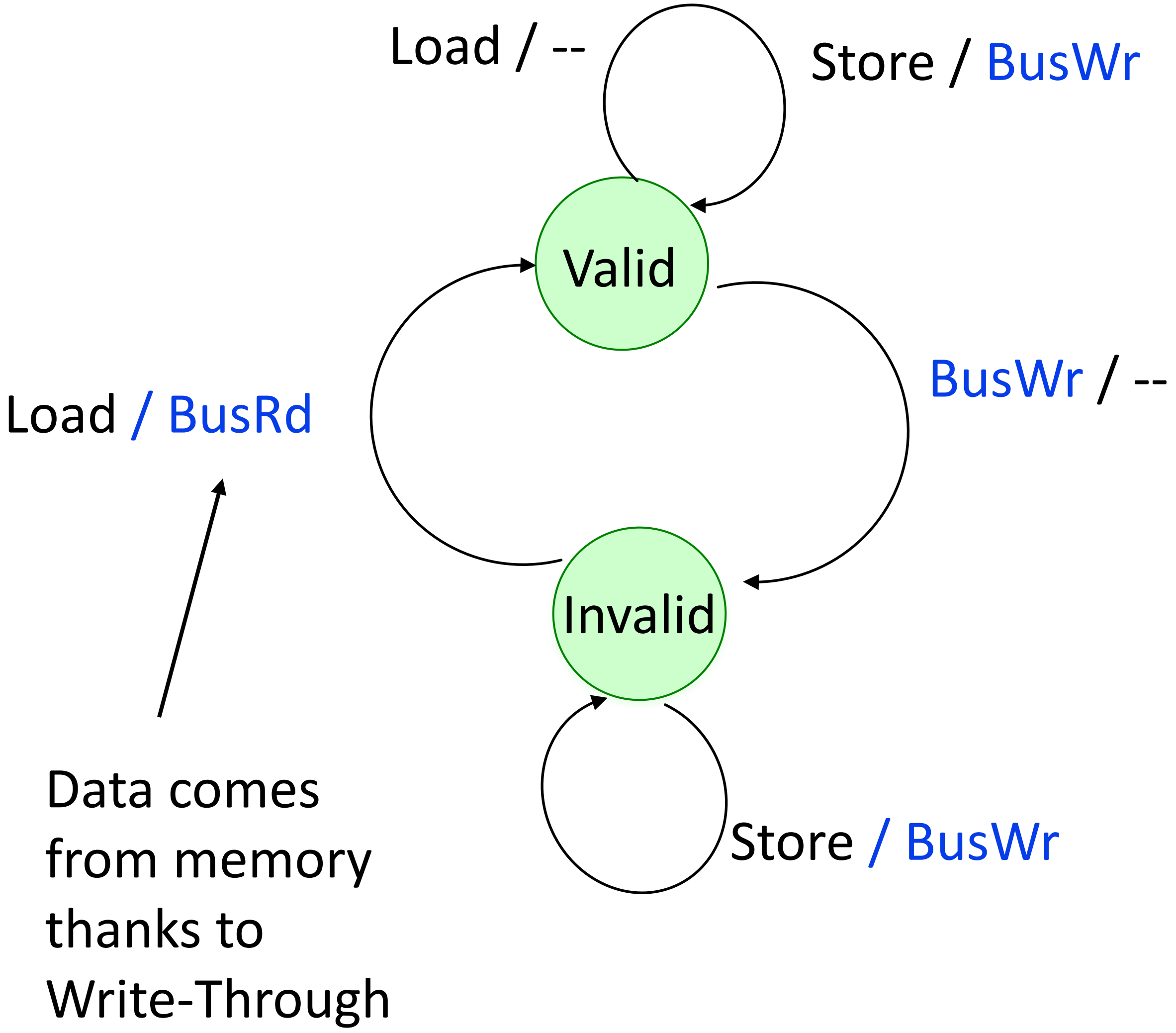
# #2-2: Valid-Invalid Protocol (for Write-Through $)



## Valid-Invalid (VI) protocol

- Whenever a cache write is done, invalidate other copies
  → support multiple copies for reads but not writes
- Both write-update and valid-invalid protocol can be implemented with a simple state machine for each cache line

# Valid-Invalid Snooping Protocol



Data comes from memory thanks to Write-Through

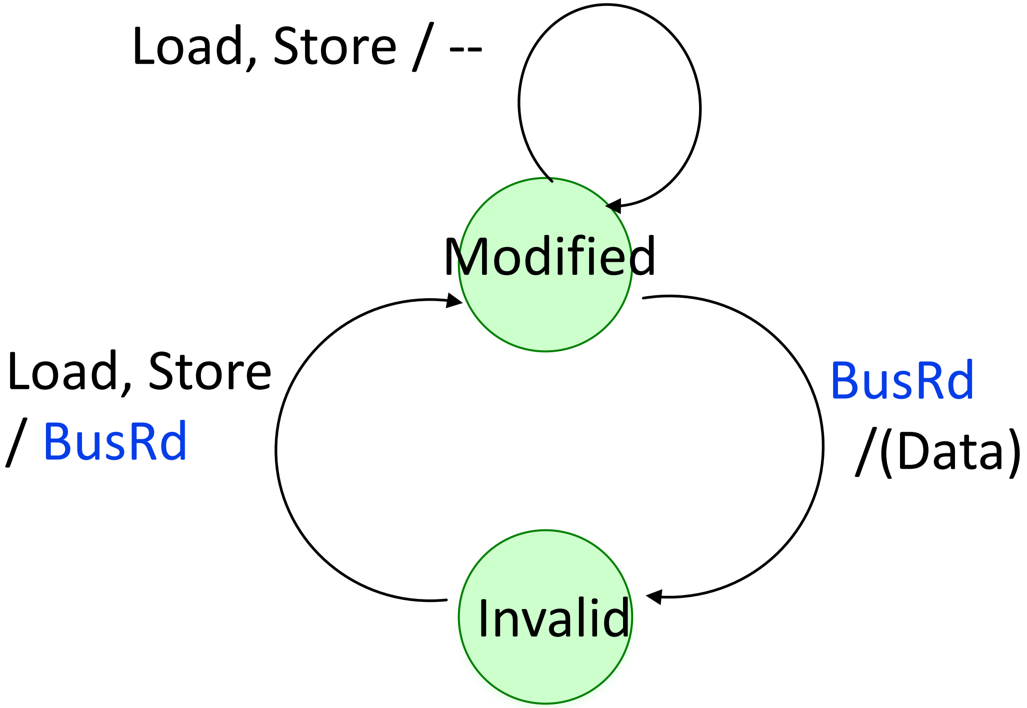
- Write-through, no-write-allocate cache
- Actions: Ld, St, BusRd, BusWr
- 1 bit of storage overhead per cache line (valid bit)
- Support multiple readers

# #3. "One Copy at All Time" Protocol, Using Write-Back Cache

- Write-through schemes consume too much bandwidth
- How about using write-back caches?
  - Cache issues a read transaction on a read or write miss
  - Cache issues a write-back to memory only when a line is displaced
- A cache line can be either **Modified** or **Invalid**
- All caches "snoop" the bus for other's ***read*** transactions
  - If a cache observes a request to a currently cached line, then respond with a value in its cache and mark its own copy **Invalid**
  - Alternatively, a cache can also ask the requestor to retry later and, in the meantime, write back its copy to memory

Why don't caches need to snoop for ***write-back*** transactions?

# Modified-Invalid Snooping Protocol



- Write-back, write-allocate
- Actions:
  Ld, St, BusRd,
- 1 bit of storage overhead per cache line
- One reader, one writer

# How about "Multiple Copies" Protocol Using "Write-Back" Cache?

- Key idea: differentiate between "mutually exclusive" and "shared" states
  - **Mutual exclusion** – when "owner" has the only replica of a cache block, it may update it freely
  - **Sharing** – multiple readers are ok, but they cannot write without gaining ownership

(1) Must find which cache (if any) is an owner on read misses

(2) Must eventually update memory so that writes are not lost

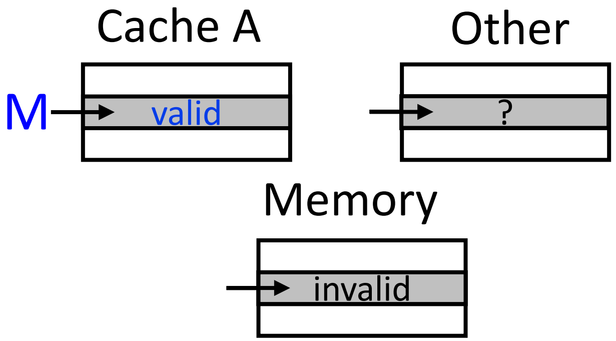
# CC Protocol for Bus-based Systems

- Bus is a broadcast medium, bus “snooping” allows every cache to see what everyone else wants to do
- A cache can even intervene in another cache’s bus transaction (e.g., a cache might ask another cache to “retry” the transaction later or respond in place of the memory)
- Besides the usual status bits, additional information (i.e., cache coherence states) is recorded with each cache line
- Example:
  - **Invalid**: cache line does not have valid data
  - **Shared**: valid line, but other caches may have copies
    (presumably all identical and unchanged from memory)
  - **Modified**: cache line has been written to since it was brought in

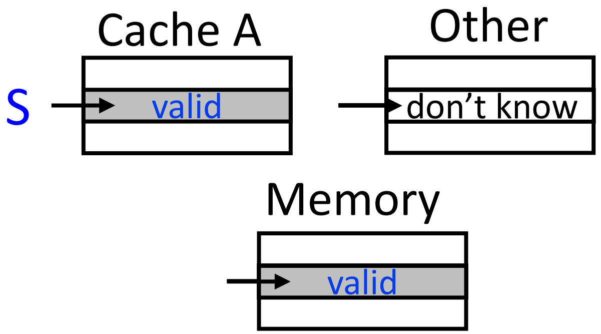
# MSI Cache Coherence States

- Given the state of an address in one cache, what are the possible states of the same address elsewhere (if they exist)?

**Modified in Cache A**

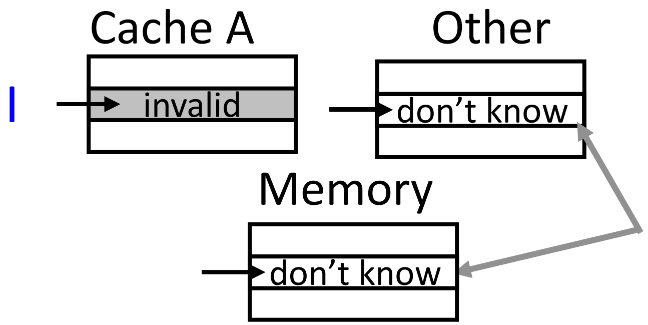


**Shared in Cache A**



shared or invalid

**Invalid in Cache A**



except that it must be valid somewhere

# MSI Protocol

- An efficient policy for single-writer/multi-reader usage
  - Allow multiple read-only copies (all identical) **(Shared)**
  - Allow only a single writable copy **(Modified)**
  - Reduces the number of bus transactions

- Based on a write-back scheme
  - On a read miss, issue a read transaction for a read-only copy
  - On a write miss, issue a "read-with-intent-to-modify" for an exclusive copy
  - On a write hit to a read-only copy, issue an "invalidate" transaction for other caches
  - When displacing a **Modified** line, write the dirty value back to memory

- All caches "snoop" the bus for other caches' read, "read-with-intent-to-modify" and invalidate transactions

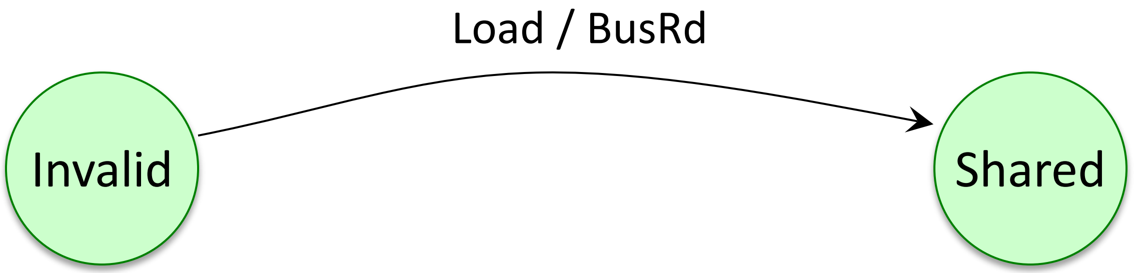
# Modified-Shared-Invalid (MSI) Protocol

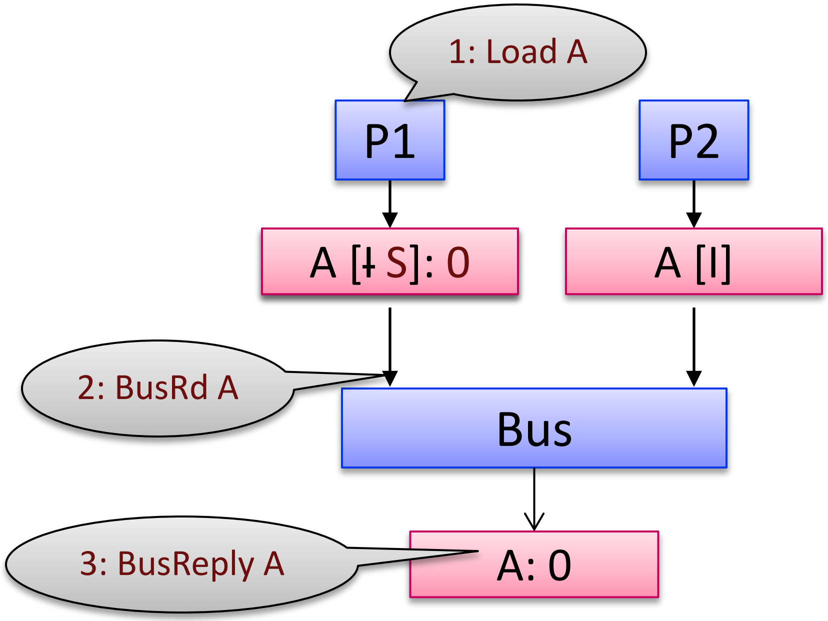
- Three states tracked per-block at each cache
  - **Invalid** – cache does not have a copy
  - **Shared** – cache has a read-only copy; clean (i.e., memory is up-to-date)
    - Some other caches might have this copy as well
  - **Modified** – cache has a writable, “dirty” copy
    - Dirty = memory is out of date
    - No other cache has this copy

- Three actions **from processor**
  - **Load, Store, Evict**

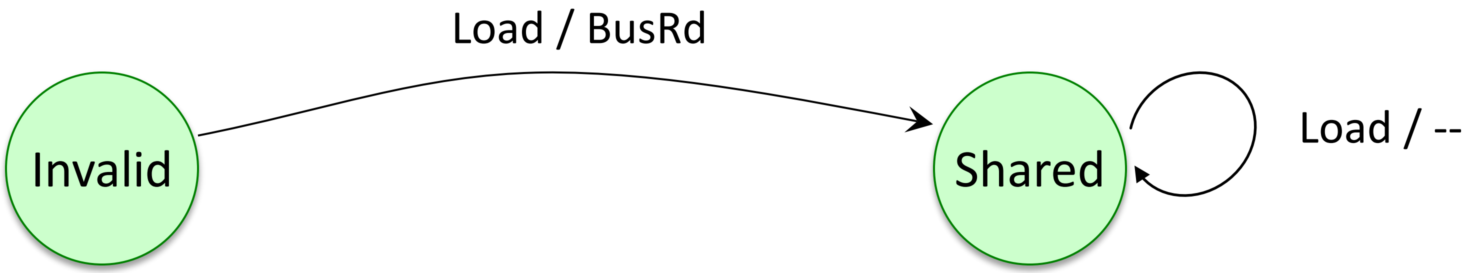
- Five messages **from bus**
  - **BusRd, BusRdX, BusInv, BusWB, BusReply**

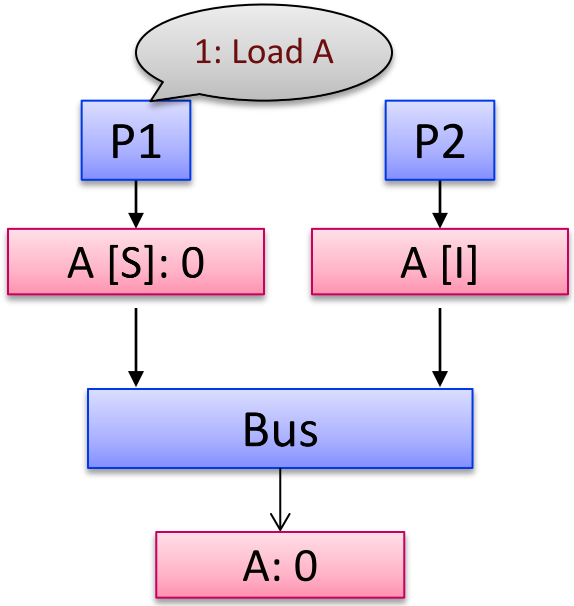
# Modified-Shared-Invalid (MSI) Protocol



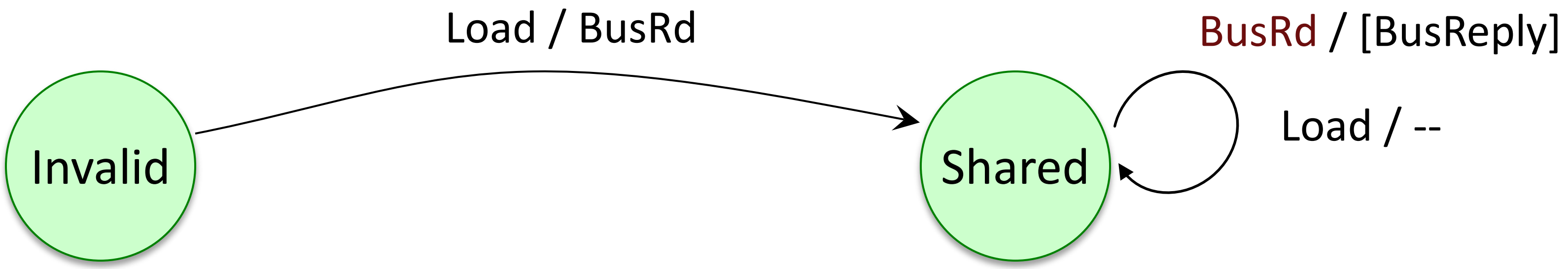


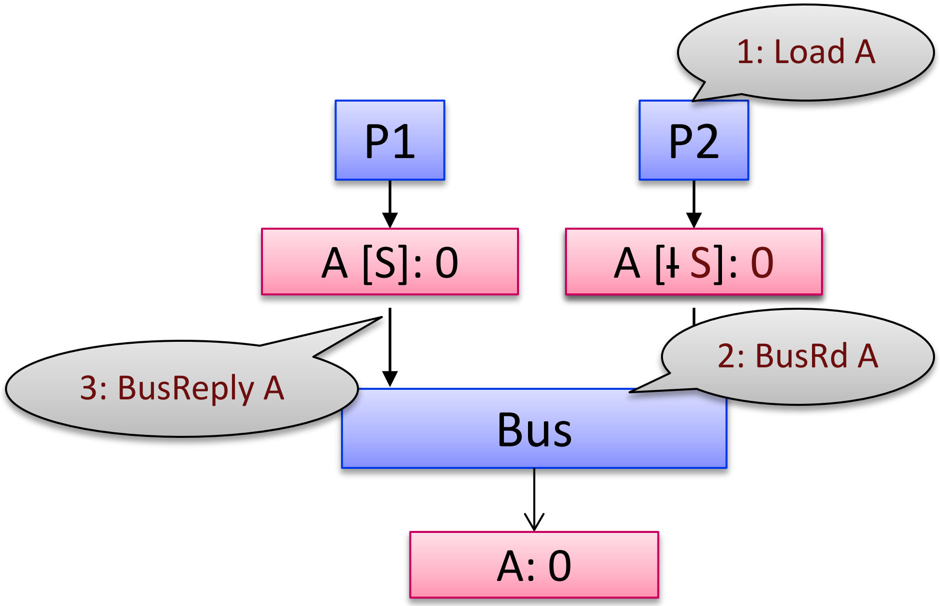
# Modified-Shared-Invalid (MSI) Protocol



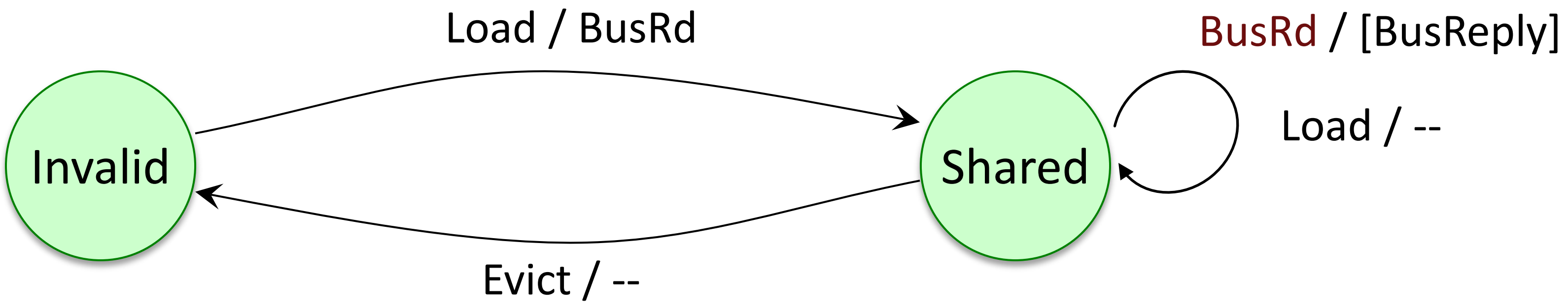


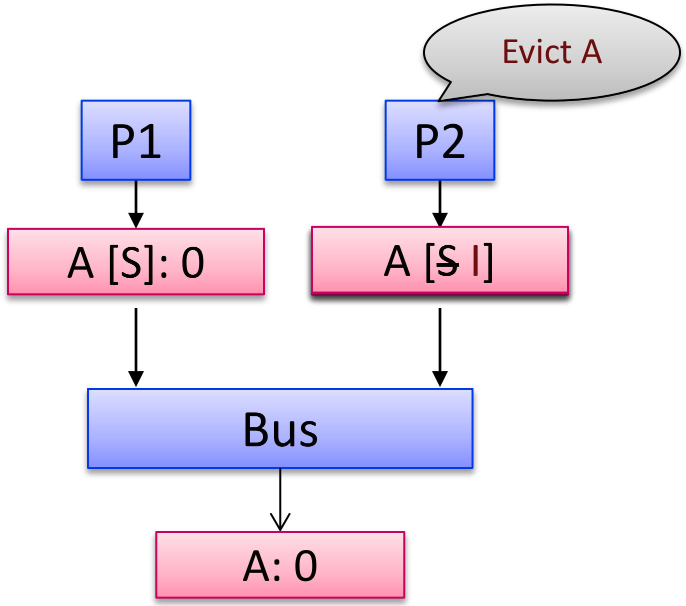
# Modified-Shared-Invalid (MSI) Protocol



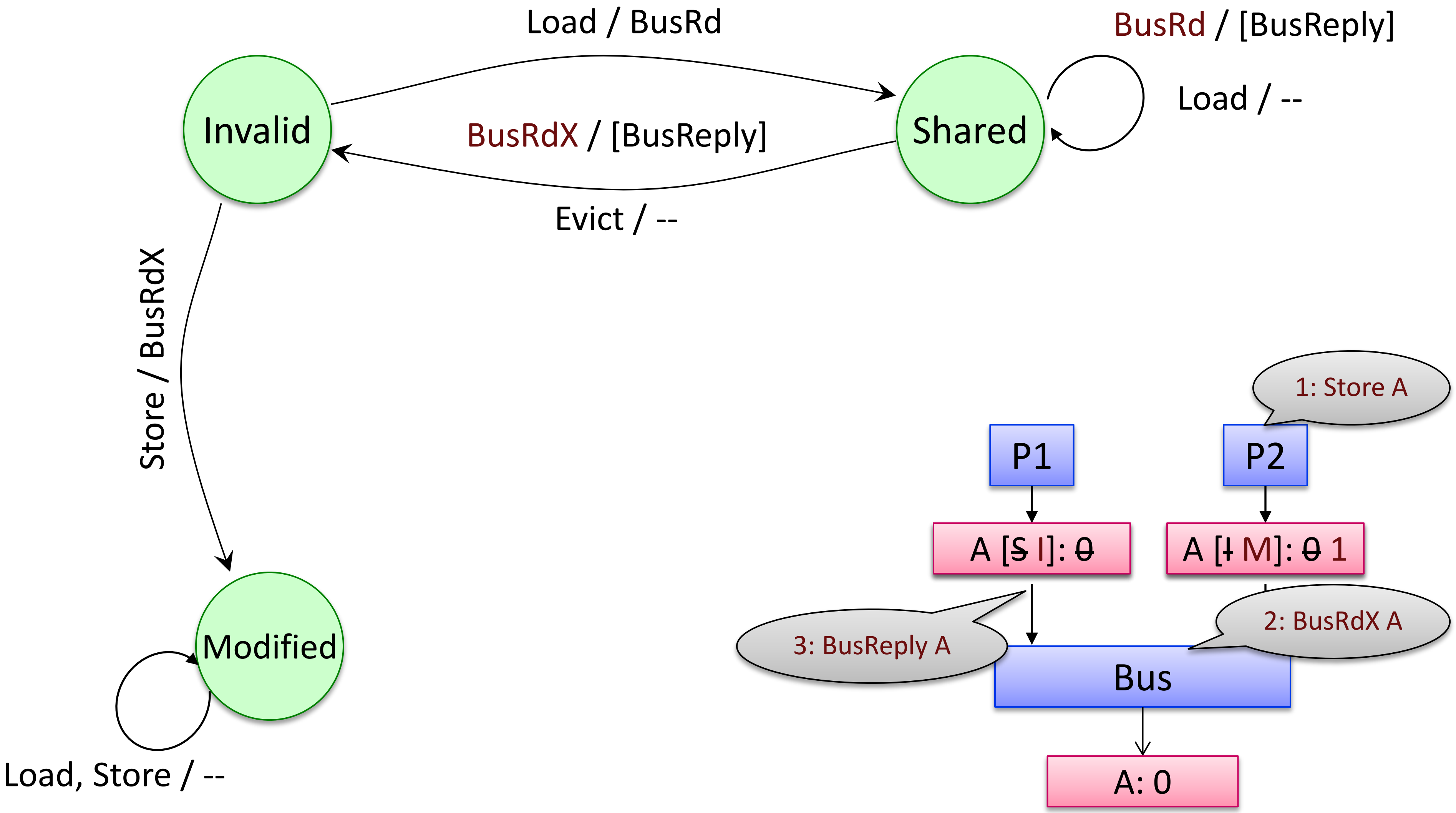


# Modified-Shared-Invalid (MSI) Protocol

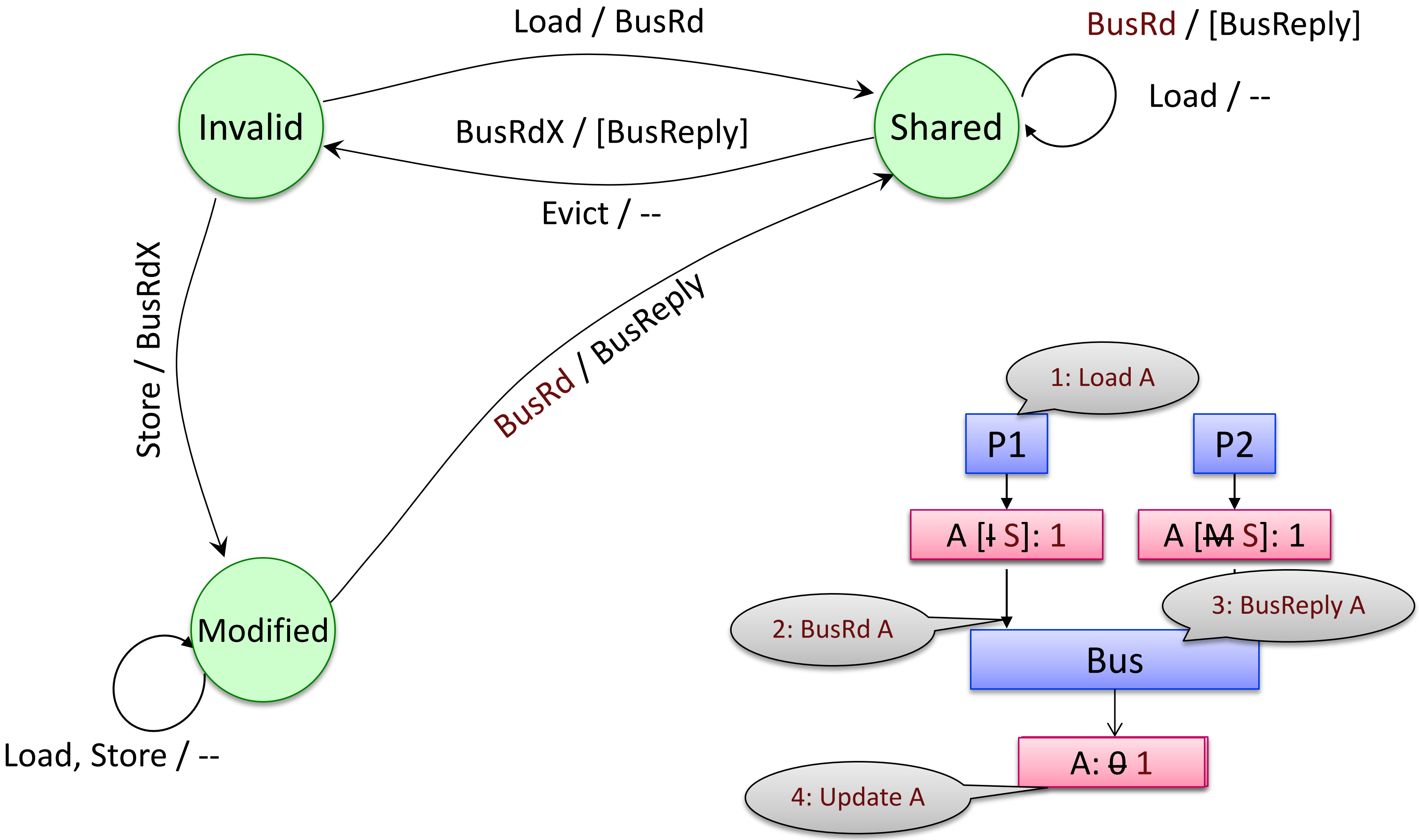




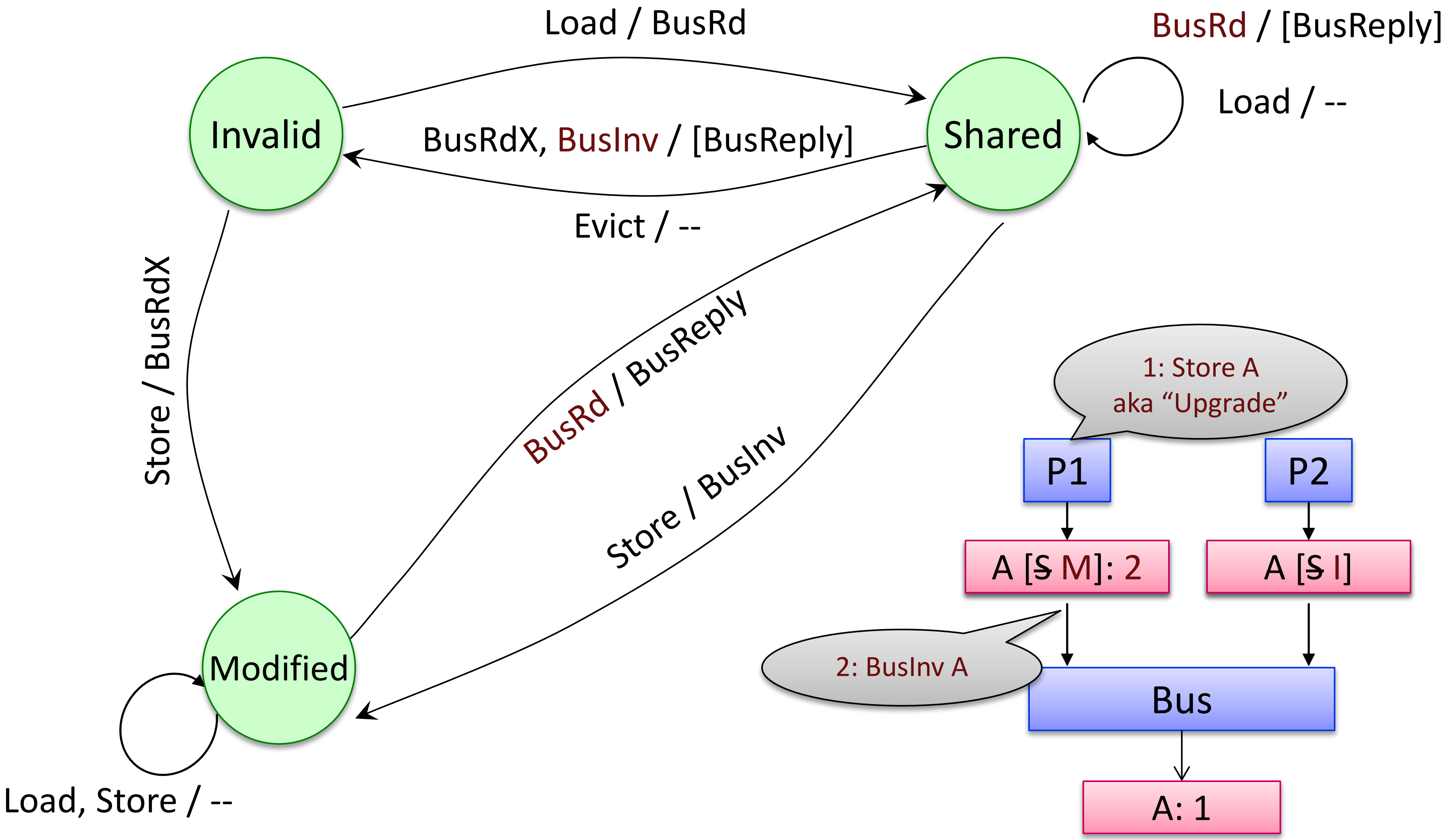
# Modified-Shared-Invalid (MSI) Protocol



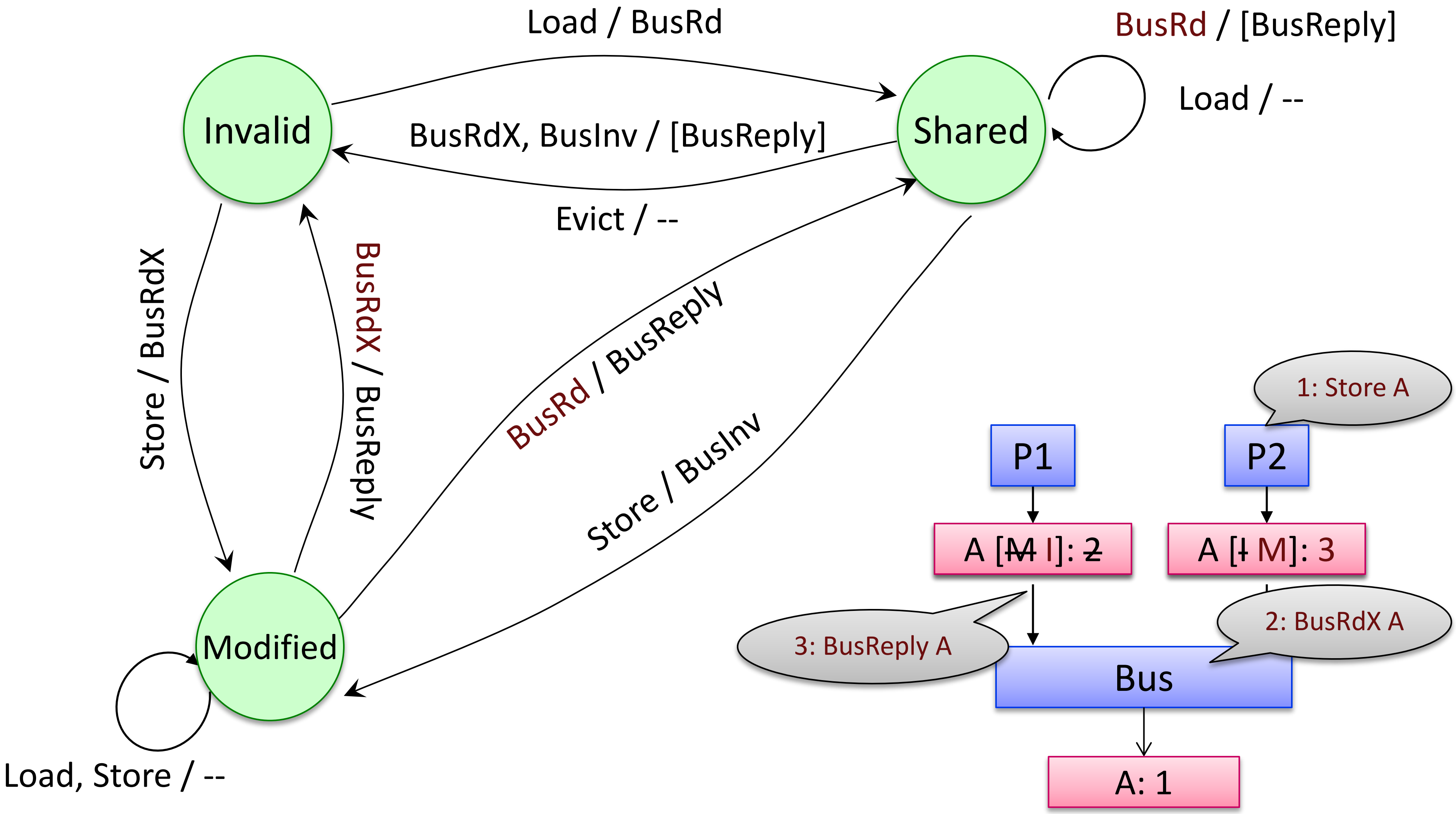
# Modified-Shared-Invalid (MSI) Protocol



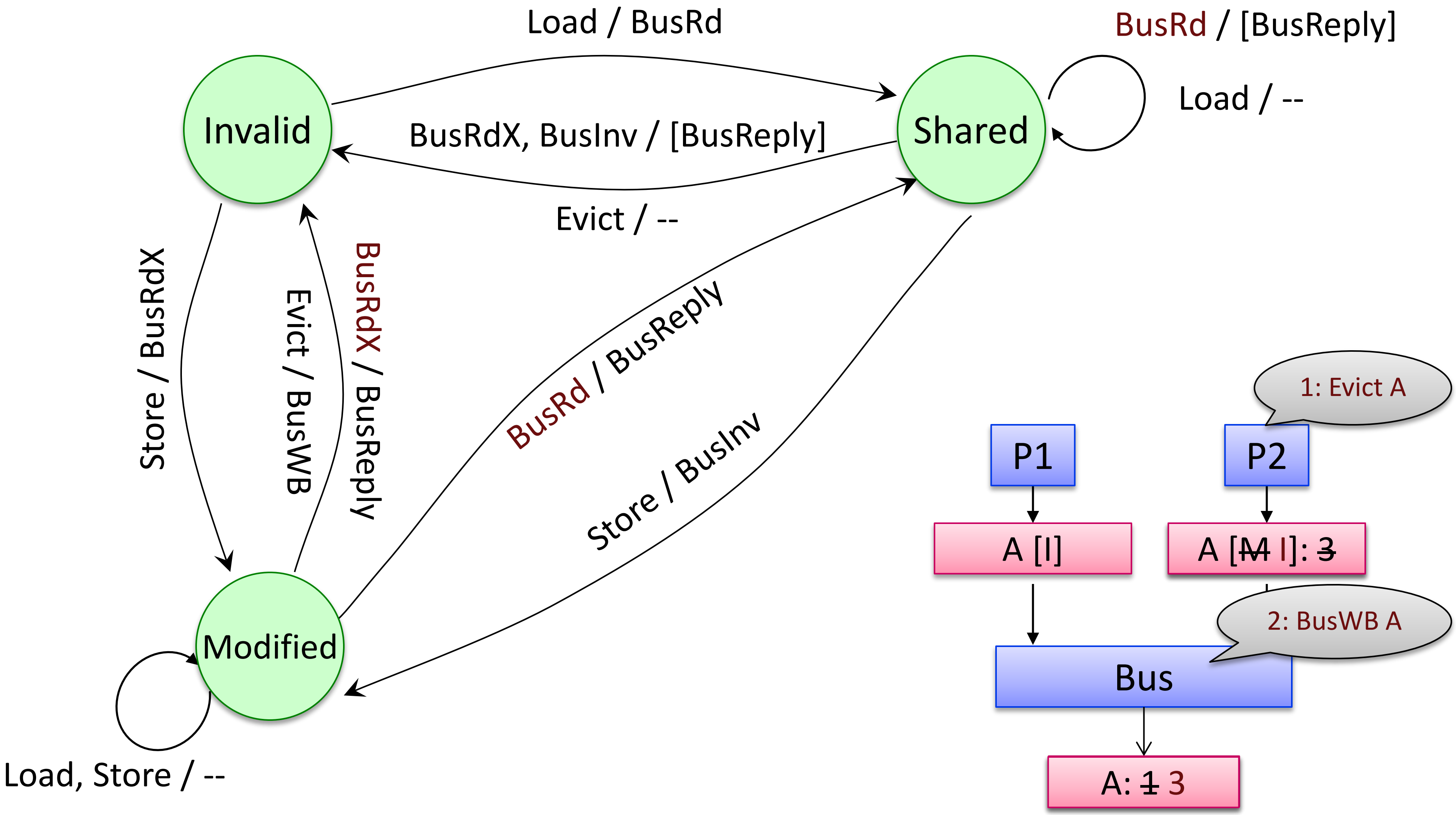
# Modified-Shared-Invalid (MSI) Protocol



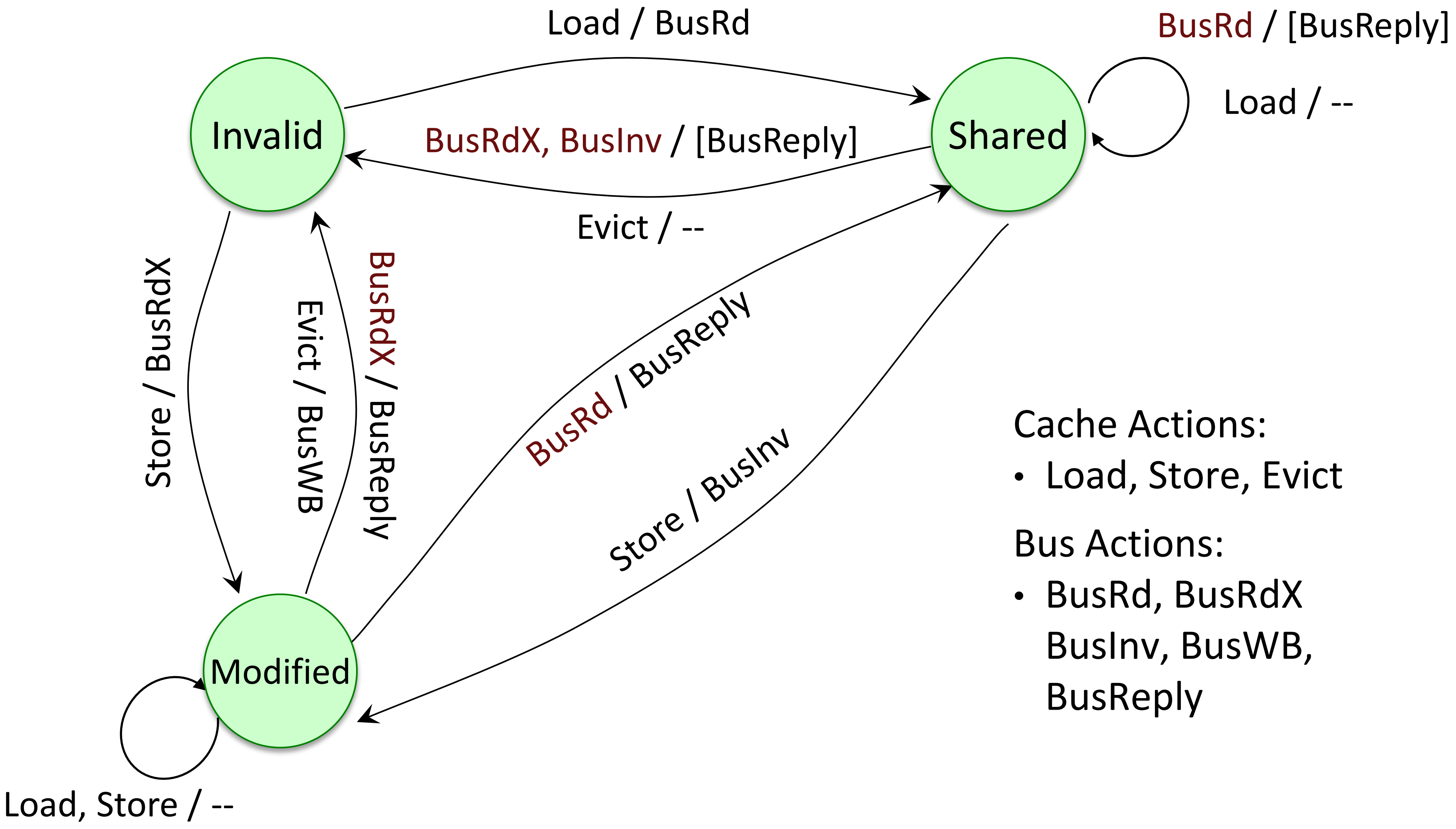
# Modified-Shared-Invalid (MSI) Protocol



# Modified-Shared-Invalid (MSI) Protocol



# MSI Protocol Summary



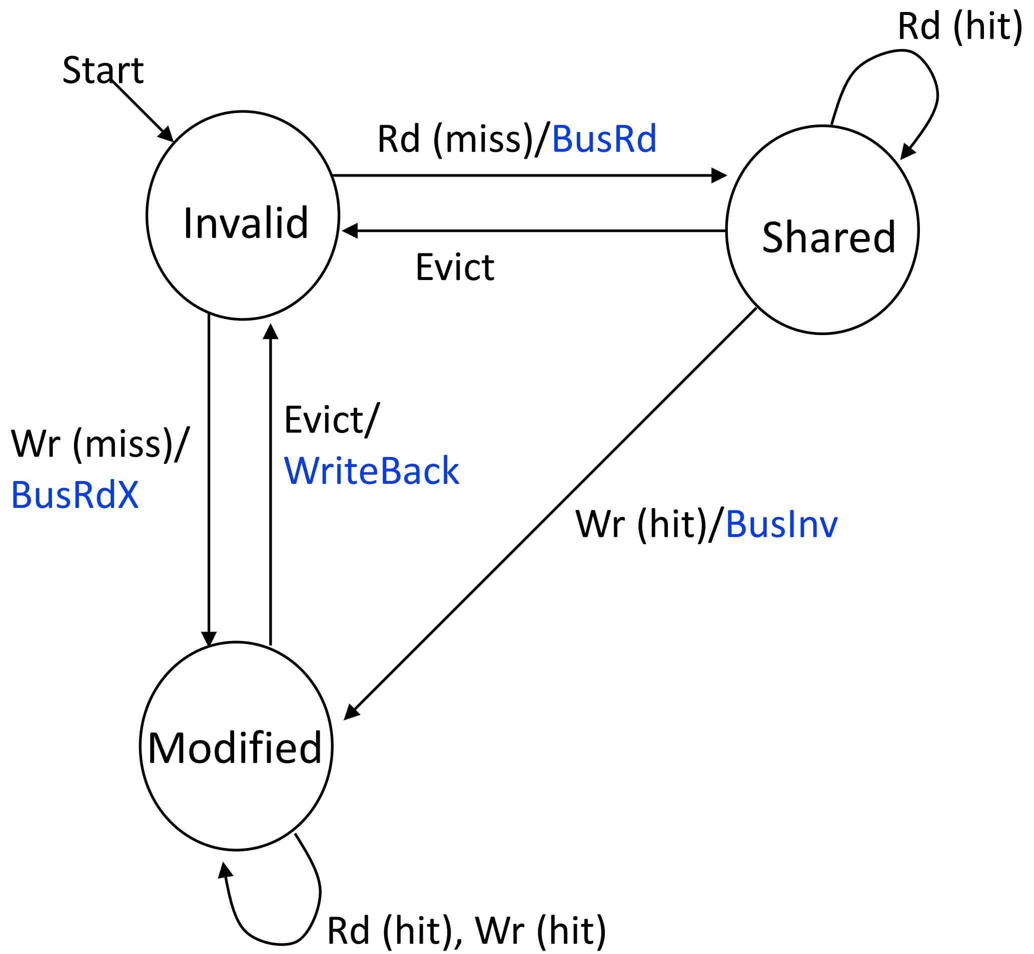
Cache Actions:

- Load, Store, Evict

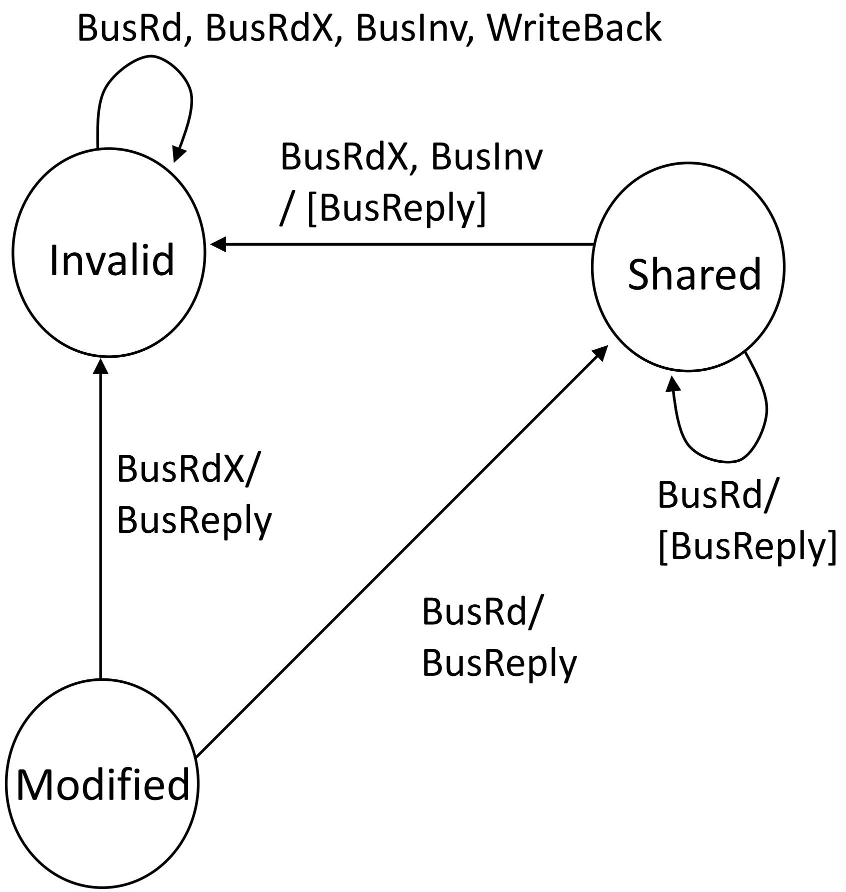
Bus Actions:

- BusRd, BusRdX BusInv, BusWB, BusReply

# MSI State Transition Diagram



a. State transitions using signals from the CPU



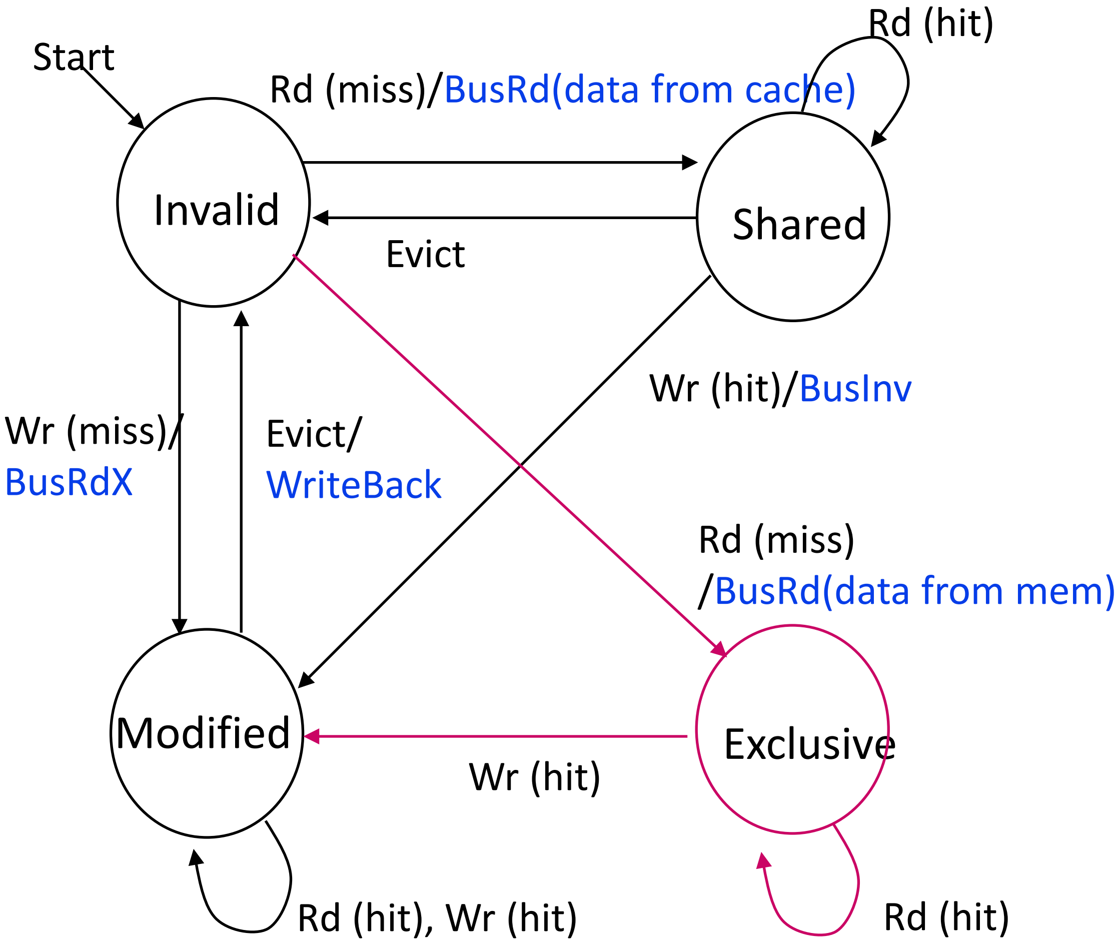
b. State transitions using signals from the BUS

# MESI Protocol

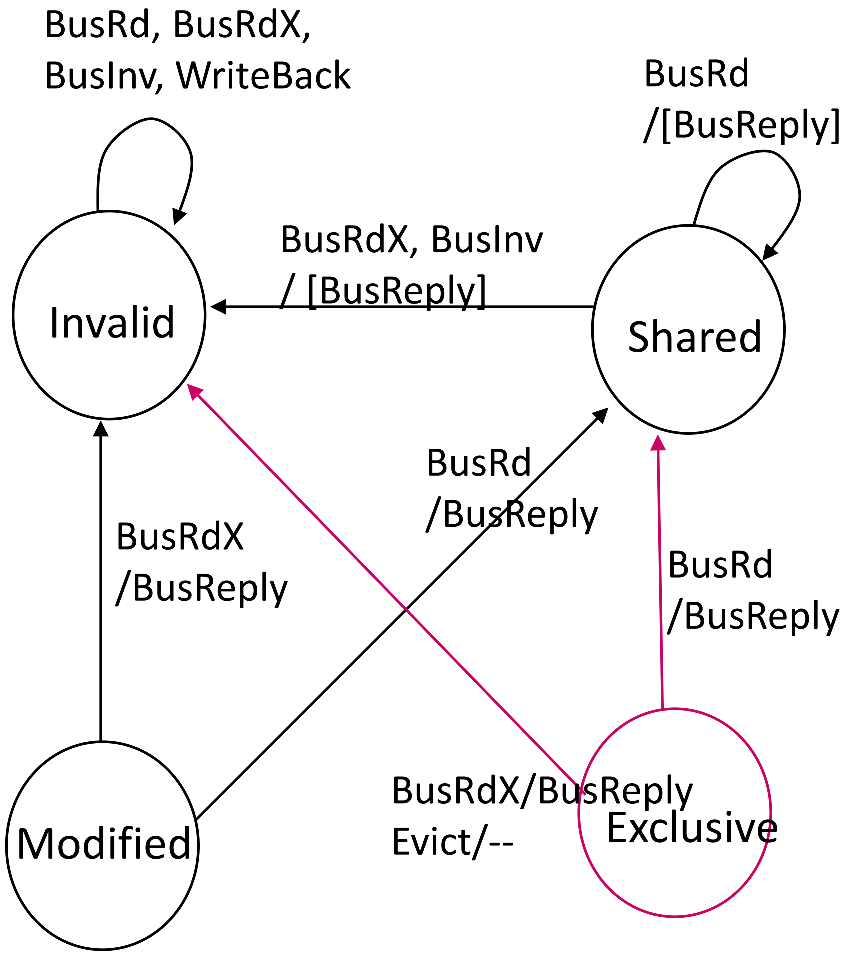
- MSI suffers from frequent read-upgrade sequences
  - Leads to two bus transactions, even for private blocks (=one sharer)
    - BusRd: gets the block in S
    - BusInv (or BusRdX, or BusUpgr): convert S to M
      - Must broadcast useless invalidation messages
  - Uniprocessors don't have this problem

- Solution: add an "Exclusive" state
  - Exclusive – only one copy; writable; clean
    - Can detect exclusivity when memory provides reply to a read
    - No need for a BusWB from Exclusive
  - For stores, silently transition to "Modified" to indicate data is dirty

# Modified-Exclusive-Shared-Invalid (MESI) Protocol



a. State transitions using signals from the CPU



b. State transitions using signals from the BUS

# Implementation of Snoopy Bus

- **Every bus snoop results in a cache lookup**
  - Needs a dual ported cache or at least an extra tag lookup port
  - In an inclusive L1 and L2 arrangement, snoop (i.e., messages from the bus) only goes to L2 and does not contend with processor for L1 bandwidth

    (True benefits of “inclusive” cache!)
- **Bus is not very scalable**
  - Physical limits (such as number of drops and physical extent of the bus) force bigger busses to clock slower
  - Bus bandwidth is divided when you add more processors
- **Implementing snoopy (or any coherence) protocols can get complicated**
  - MSI and MESI state transitions are not really atomic (need many *transient* states)
  - CPU and bus transactions are not atomic
  - CC issues can become intertwined with memory consistency

# Scalable Cache Coherence

- **Scalable cache coherence**: two part solution

- Part I: **Bus bandwidth**

  → Replace non-scalable bandwidth substrate (bus) with scalable bandwidth one (point-to-point network, e.g., mesh)

- Part II: **Processor snooping bandwidth**
  - Observation: most snoops result in no action

  → Replace non-scalable broadcast protocol (spam everyone) with scalable **Directory protocol** (only spam processors that care)

# Directory Coherence Protocols

- Idea: A logically-central directory keeps track of where the copies of each cache block reside. Caches consult this directory to ensure coherence.

- **Directories** for non-broadcast coherence protocol
  - Extend memory to track caching information
  - For each physical cache line, the directory records:
    - **Owner**: which processor has a dirty (M state) or exclusive (E state) copy
    - **Sharers**: which processors have clean (S state) copies – 1 bit for each processor
  - Processor sends coherence event to the directory. For example, (for the corresponding directory entry)
    - On a read: set the cache's sharer bit and arrange the supply of data
    - On a write: invalidate all caches that have the block and reset their bits
- Advantage: scalable (less coherence traffic)
- Disadvantages: additional storage overhead, increased latency ($ → dir. → mem.)

# Directory-based Cache Coherence Example

P1 Cache

| Line | State |
|---|---|
| A | M |
| | |

P2 Cache

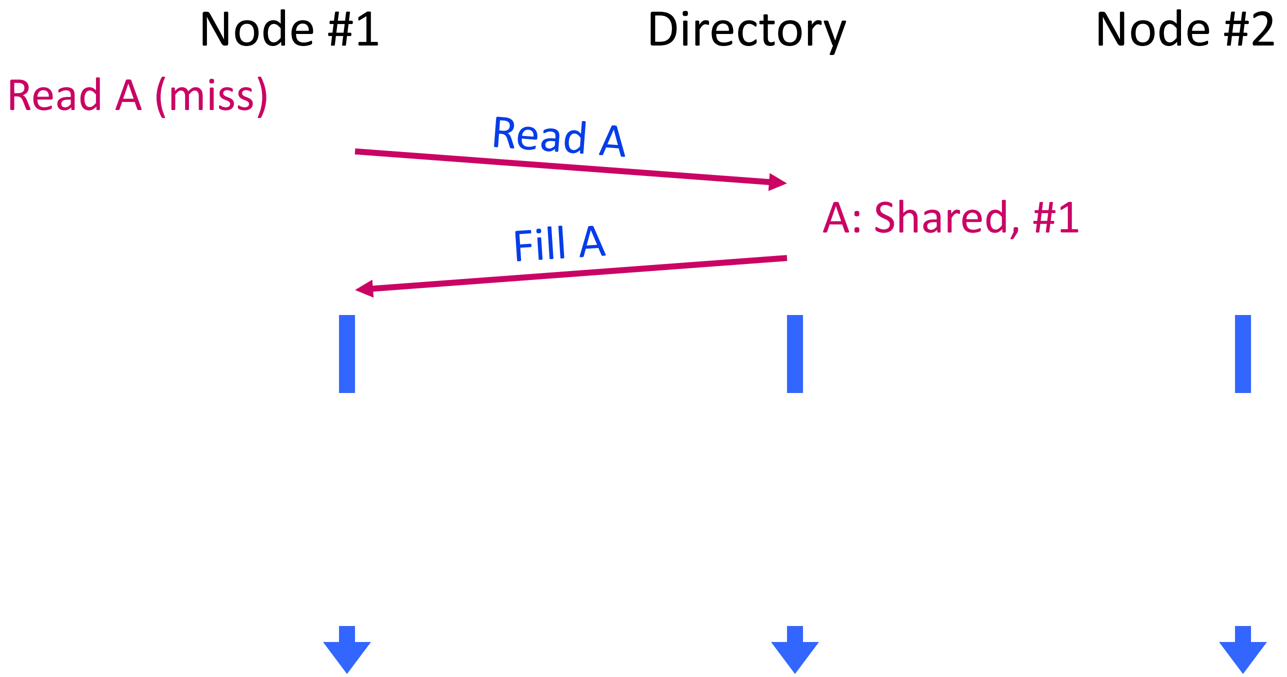
| Line | State |
|---|---|
| B | S |
| | |

P3 Cache

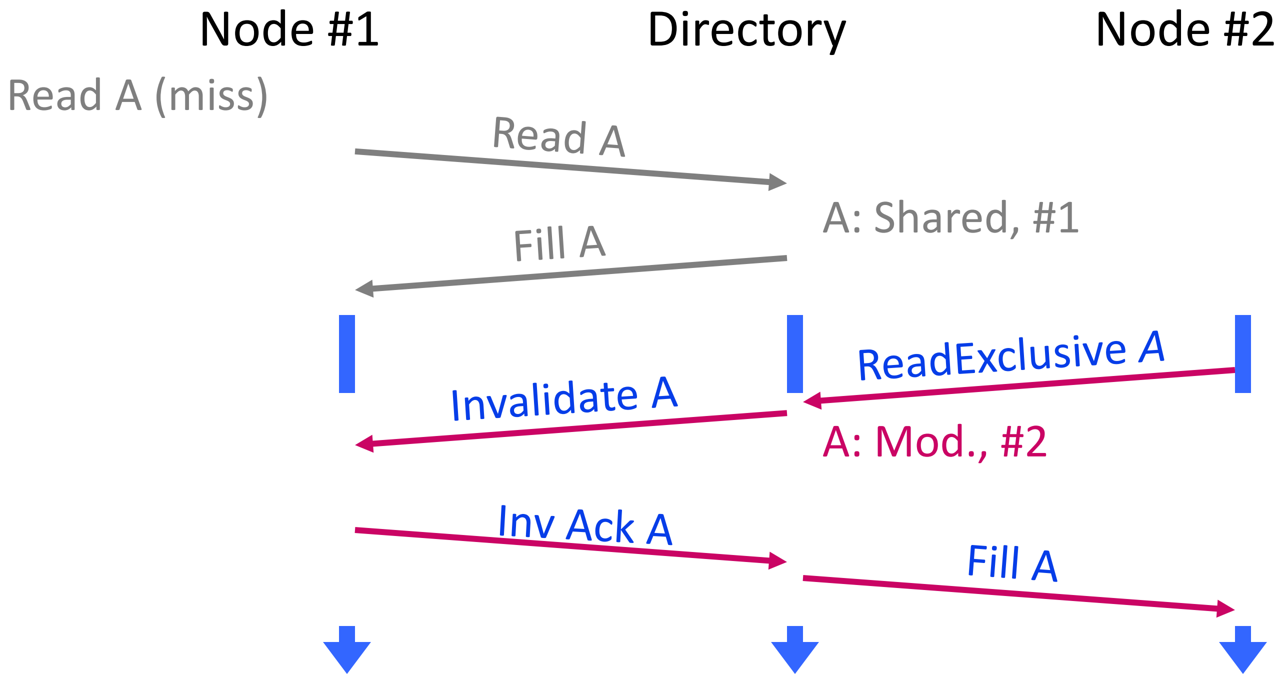
| Line | State |
|---|---|
| B | S |
| | |

| Location | Nodes | State |
|---|---|---|
| A | 100 | EM |
| B | 011 | S |
| C | 000 | U |
| D | 000 | U |
| ... | | |

# Basic Operation: Read



# Basic Operation: Write



# Centralized Directory

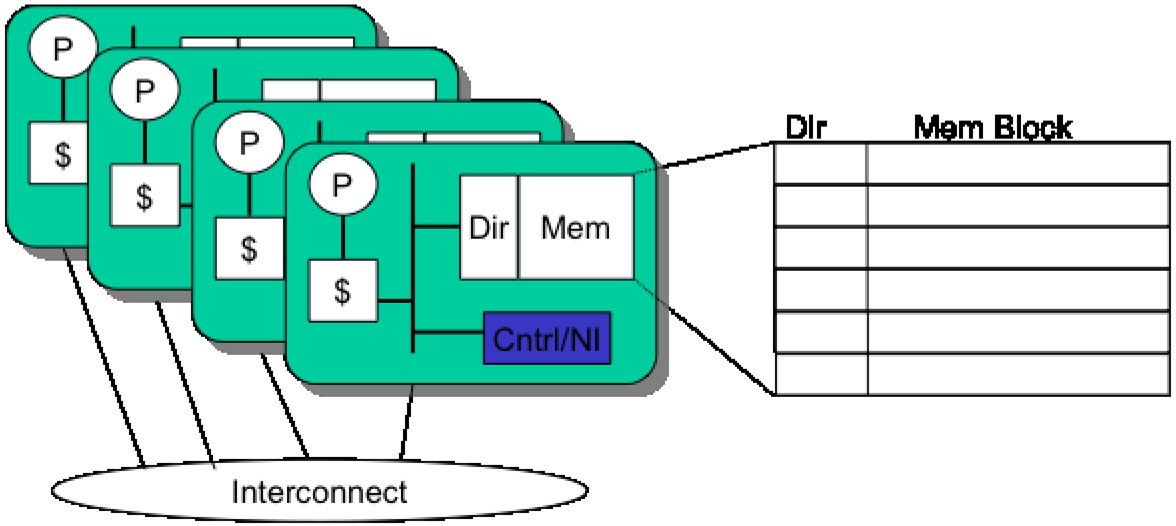
- Single directory contains a copy of cache tags from all nodes

- Advantages:
  - Send Invalidate/Update only to nodes that have copies
  - Central serialization point: easy to get memory consistency (just like a bus…)

- Problems:
  - Less scalable than a distributed directory (imagine traffic from 1000's of nodes…)
  - Directory size/organization changes with the number of nodes

# Distributed Directory

- Distribute directory among memory modules
  - Memory block = Coherence block (usually = cache line)
  - “**Home node**” → Node with directory entry
    - Usually also dedicated main memory storage for cache line
  - **Scalable** – Directory grows with memory capacity
  - But the directory can **no longer serialize** accesses across all addresses
    - Memory consistency becomes responsibility of CPU interface



# Directory for 1000 nodes?

- Scalability Problem
  - What if I have 2, 4, 8, 16, 32, …., 1000s CPUs?

    (1000s-CPU systems have been implemented)

Snoop-based CC for 100~1000 CPUs (or cores)?

- Directory-based CC for many nodes

  (1) Bit Vector-based Directory CC: N-bit directory info (=bit vector) added to cache state bits for N-CPU system.

  (e.g., [10010110] → CPU 1, 4, 6, 7 are sharing the block)

  (2) For too many nodes, Pointer-based Directory CC can be used.

  (e.g., [1001 0110] → CPU 9, CPU 6 are sharing the block)

Which one is better?

# Designing a Directory Protocol: Nomenclature

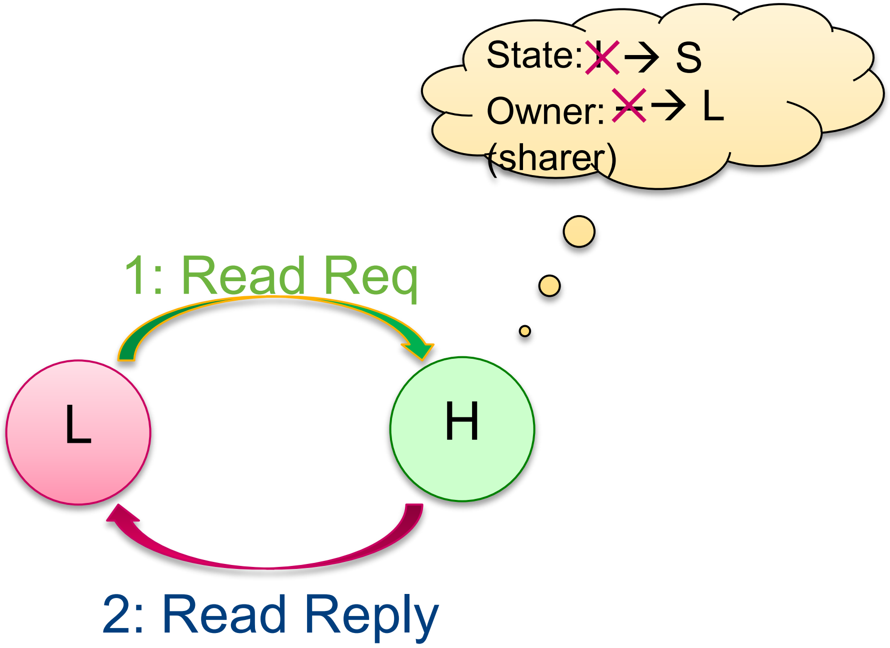
- Local Node (L)
  - Node initiating the transaction we care about

- Home Node (H)
  - The node where the memory location and the directory entry of an address reside

- Remote Node (R)
  - Any other node that participates in the transaction

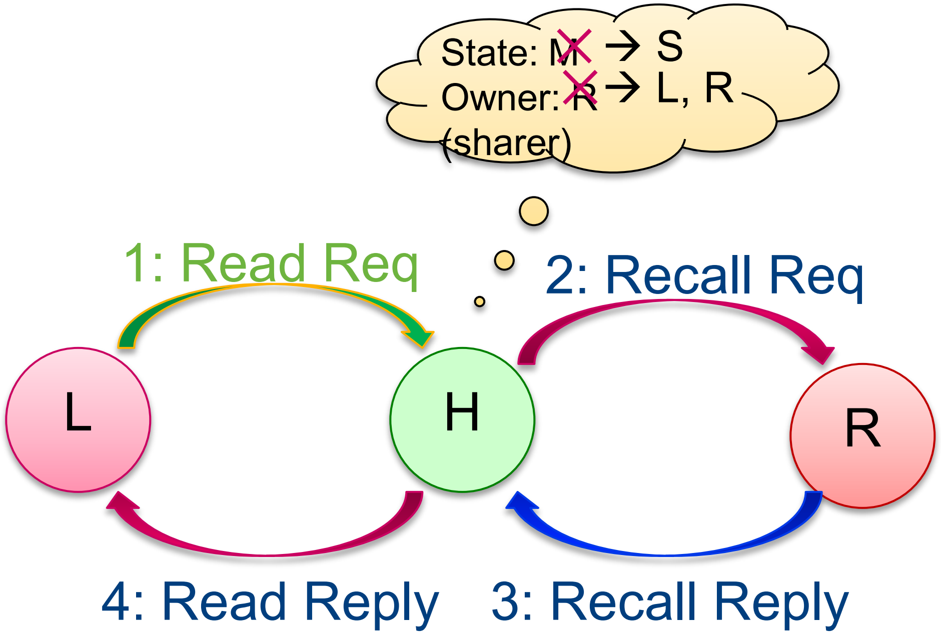
# Read Transaction (data in home memory)

- L has a cache miss on a load instruction



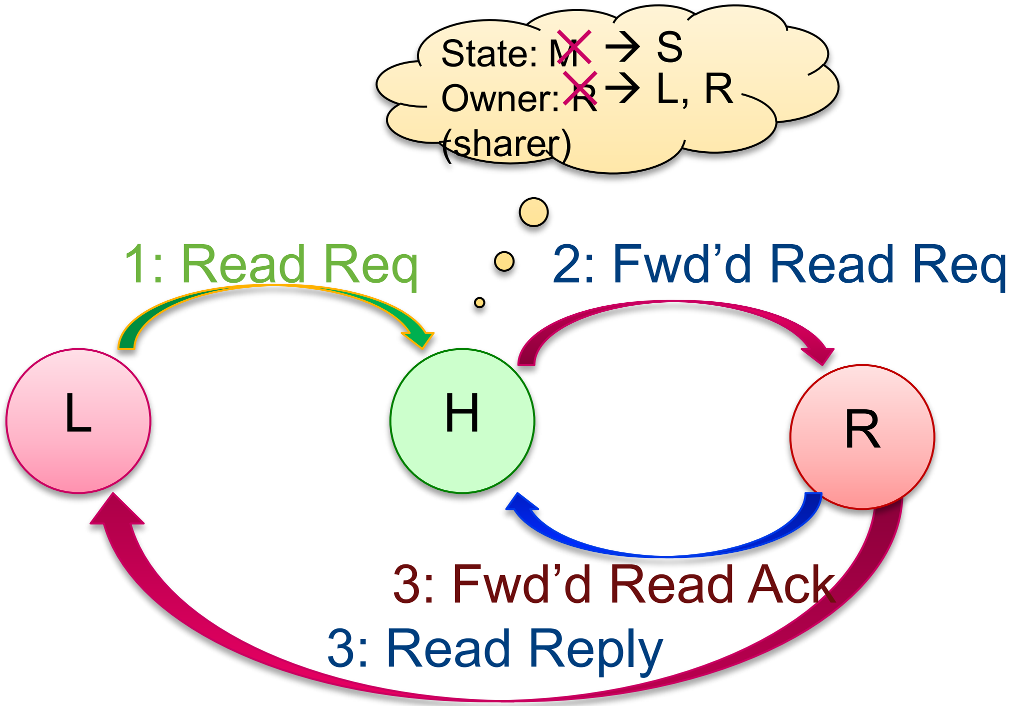
# 4-hop Read Transaction (data in a remote cache)

- L has a cache miss on a load instruction
  - Block was previously in modified state at R



# 3-hop Read Transaction (data in a remote cache)

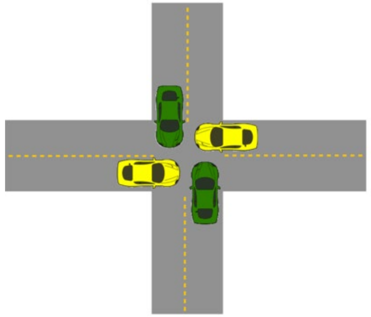
- L has a cache miss on a load instruction
  - Block was previously in modified state at R



# Performance Issues (General Idea)

- Deadlock
  - No one is making a progress

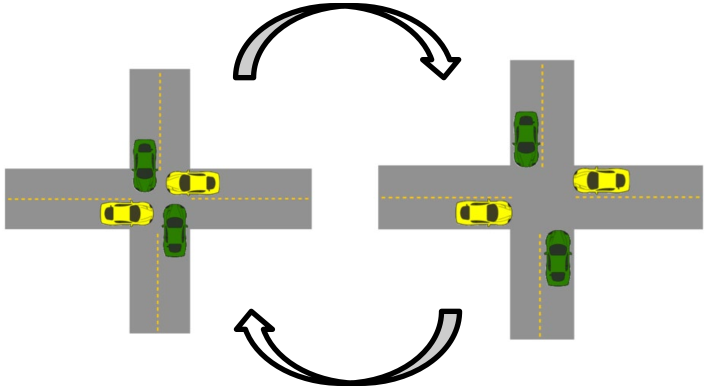
    e.g., Everyone stopped to wait for a forever-busy resource



- Livelock
  - Some are making progress, but to reach the old same status

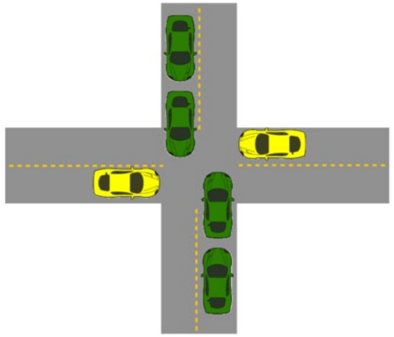
    e.g., Everyone tries to get a resource at every odd second,

    while the resource becomes available at every even second



- Starvation
  - Only some are making progress (= unfair progress)

    e.g., only one guy keeps winning in getting the resource



Images from http://15418.courses.cs.cmu.edu/spring2014/lecture/snoopimpl1/

# Performance Issues in CC

- Deadlock example

  Different addresses A & B happen to map to ~~a same~~ different cache lines.

  $P_A$ with a modified copy of A placed a read-miss req for B on the bus.
  $P_B$ with a modified copy of B placed a read-miss req for A on the bus.

  What if $P_A$ and $P_B$ want to serve their older requests first?

- Livelock example

  If all CPUs try to write to A simultaneously, they must invalidate other copies of A in CPUs.
  What if As in all caches continue to get invalidated before any write is completed?

- Starvation example

  What if a specific CPU continues to become successful in writing to A, while other CPUs are failing to write to A?

Extremely difficult to design a high-performance, scalable, and functionally correct cache-coherence protocol.

# Shared Memory Summary (1/2)

- Shared-memory multiprocessors
  - + Simple software: easy data sharing, easy programming
  - – Complex hardware: must provide illusion of global address space

- Two basic implementations
  - – **Symmetric (UMA) multi-processors (SMPs)**
    - • Underlying communication network: bus (ordered)
    - + Low-latency, simple protocols that rely on global order
    - – Low-bandwidth, poor scalability
  - – **Scalable (NUMA) multi-processors (MPPs)**
    - • Underlying communication network: point-to-point (unordered)
    - + Scalable bandwidth
    - – Higher-latency, complex protocols

# Shared Memory Summary (2/2)

- Two aspects to global memory space illusion
  - **Coherence**: consistent view of individual cache lines
    - Absolute coherence not needed, relative coherence OK
    - VI and MSI protocols, cache-to-cache transfer optimization
    - Implementation?
      - SMP: snooping, MPP: directories
  - **Consistency** (next lecture): consistent global view of all memory locations
    - Programmers intuitively expect sequential consistency (SC)
      - Global interleaving of individual processor access streams
      - Not naturally provided by coherence, needs extra stuff
    - Relaxed ordering: consistency only for synchronization points

# Question?

**Announcements**

- Textbook reading: P&H Ch 5.10
- Paper reading: “The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor”, Lenoski, Laudon, and Gharachorloo, ISCA’90