# SHARED MEMORY SYSTEMS

Mahdi Nazm Bojnordi

Assistant Professor

School of Computing

University of Utah

# Overview

- Shared memory systems
    - Inconsistent vs. consistent data

- Cache coherence with write back policy
    - MSI protocol
    - MESI protocol

- Memory consistency
    - Sequential consistency

# Simple Snooping Protocol

- Relies on write-through, write no-allocate cache
- Multiple readers are allowed
  - Writes invalidate replicas
- Employs a simple state machine for each cache unit
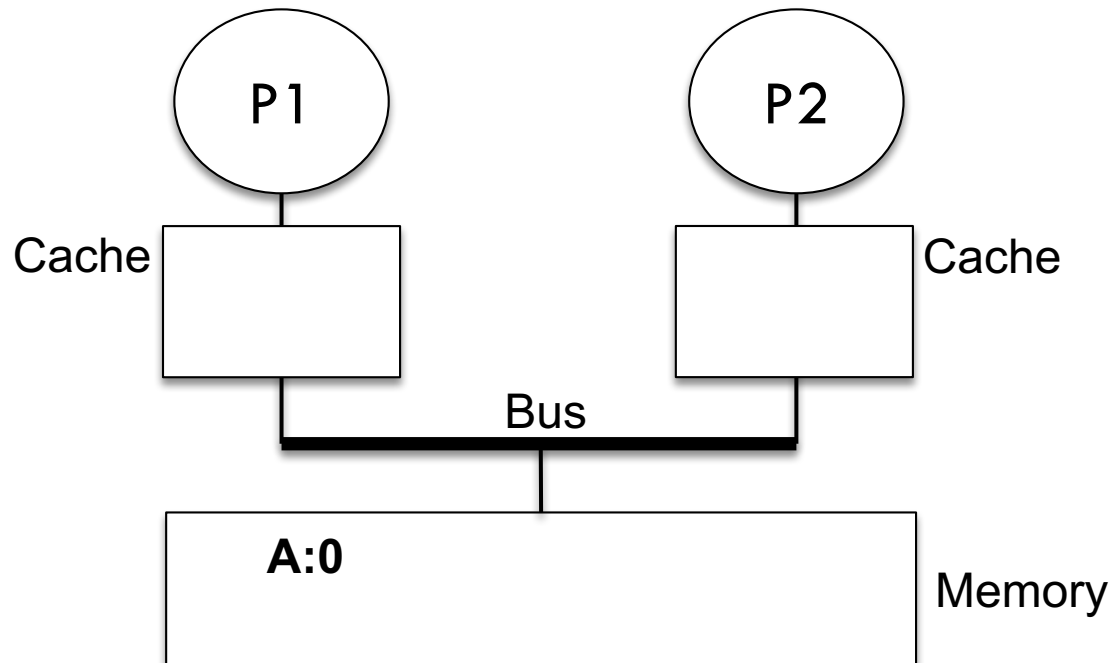
# Simple Snooping State Machine

- Every node updates its one-bit valid flag using a simple finite state machine (FSM)

- Processor actions
  - Load, Store, Evict

- Bus traffic
  - BusRd, BusWr

Load/--    Store/BusWr

( Valid )

Load/BusRd    Evict/--    BusWr/--

( Invalid )

Store/BusWr
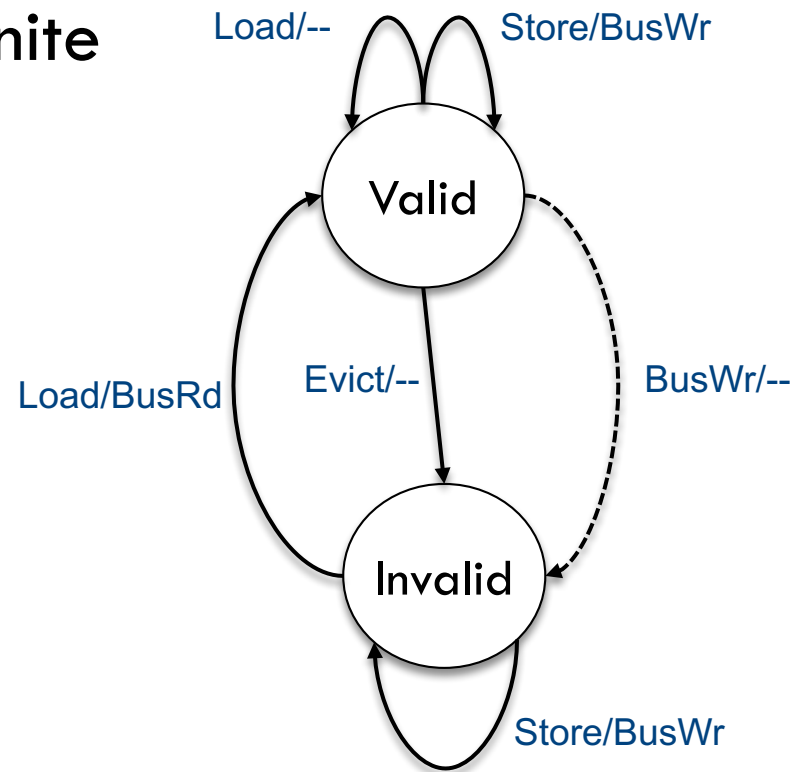
→ Transaction by local actions

- - - → Transaction by bus traffic

# Snooping with Writeback Policy

- **Problem**: writes are not propagated to memory until eviction
  - Cache data maybe different from main memory

- **Solution**: identify the owner of the most recently updated replica
  - Every data may have only one owner at any time
  - Only the owner can update the replica
  - Multiple readers can share the data
    - No one can write without gaining ownership first

# Modified-Shared-Invalid Protocol

- Every cache block transitions among three states
  - Invalid: no replica in the cache
  - Shared: a read-only copy in the cache
    - Multiple units may have the same copy
  - Modified: a writable copy of the data in the cache
    - The replica has been updated
    - The cache has the only valid copy of the data block

- Processor actions
  - Load, store, evict
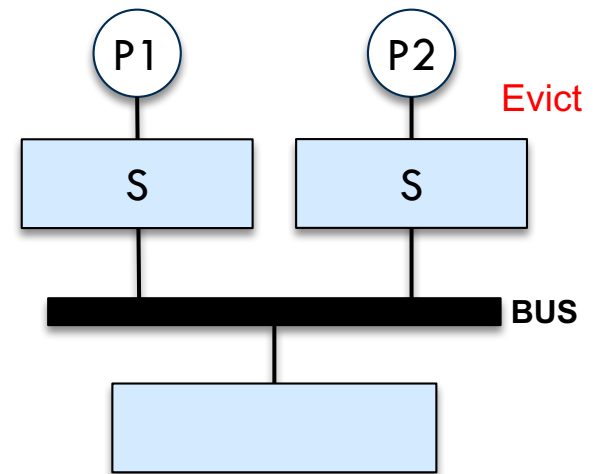
- Bus messages
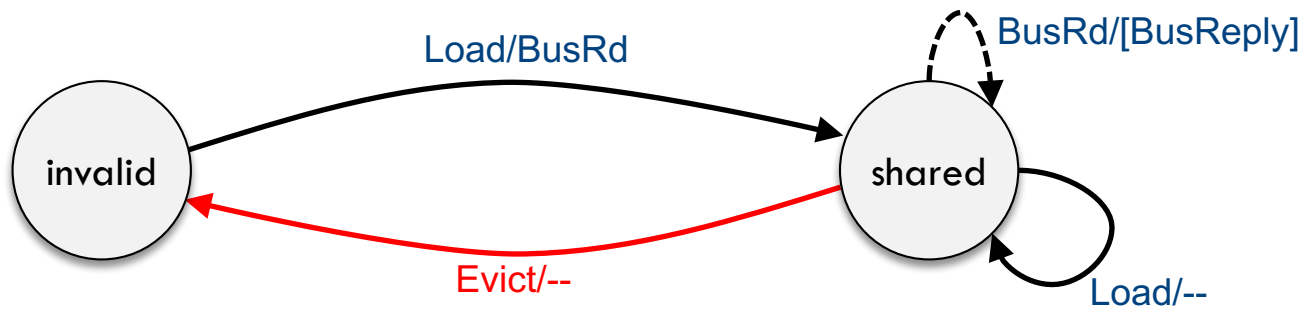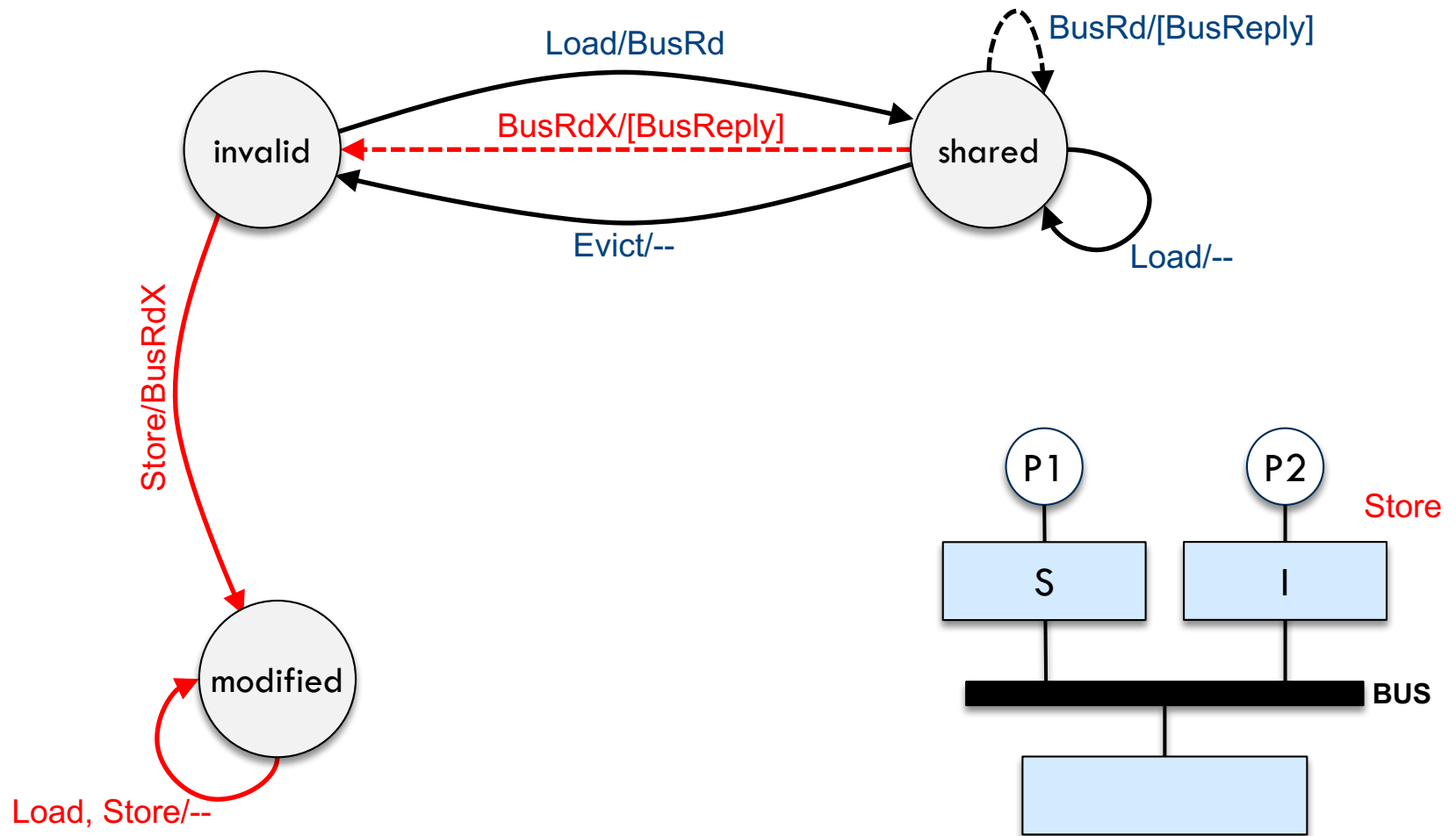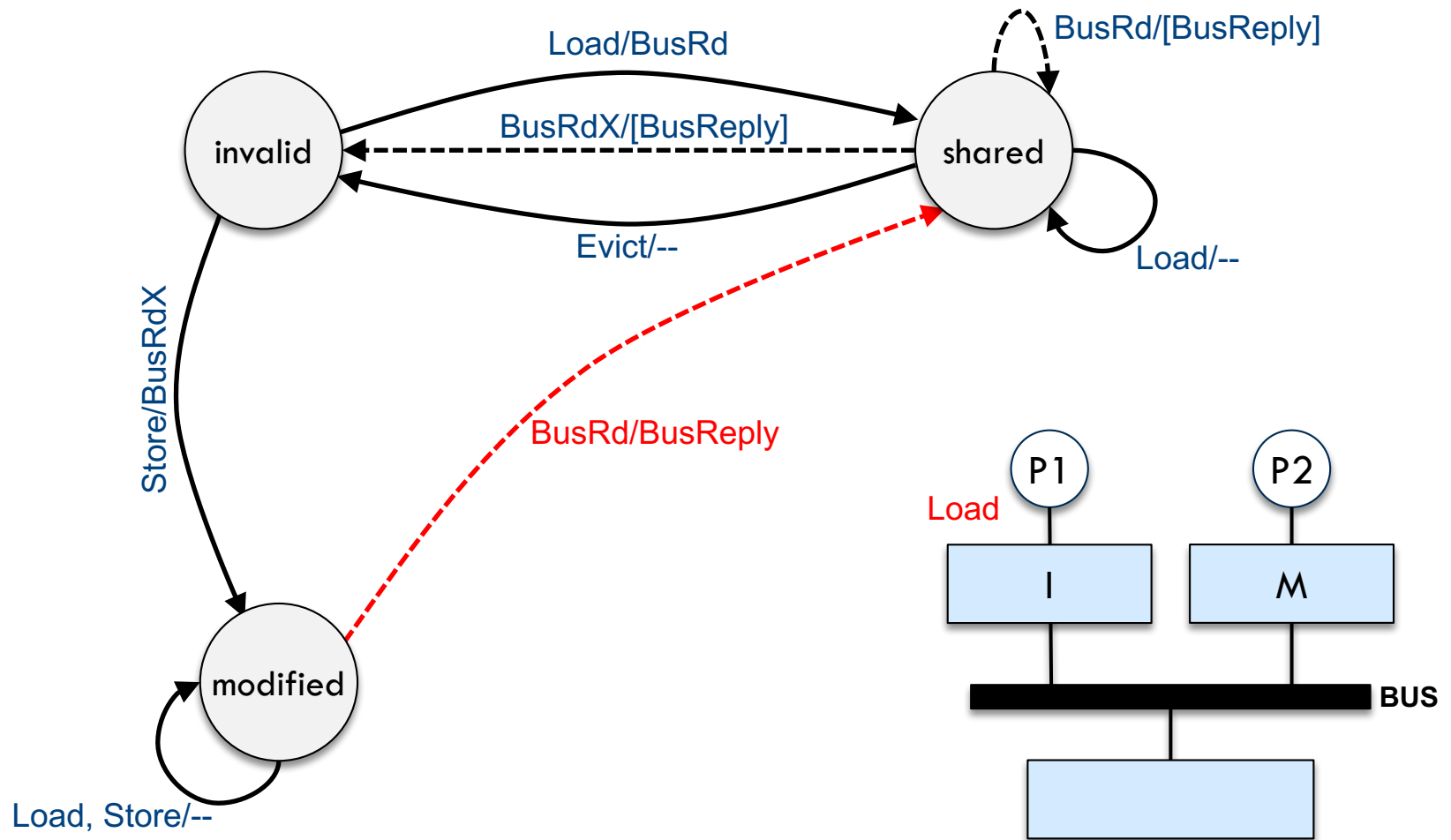  - BusRd, BusRdX, BusInv, BusWB, BusReply

# MSI Example

invalid —— Load/BusRd ——> shared

P1    P2

Load

I        I

BusRd

BUS

BusReply
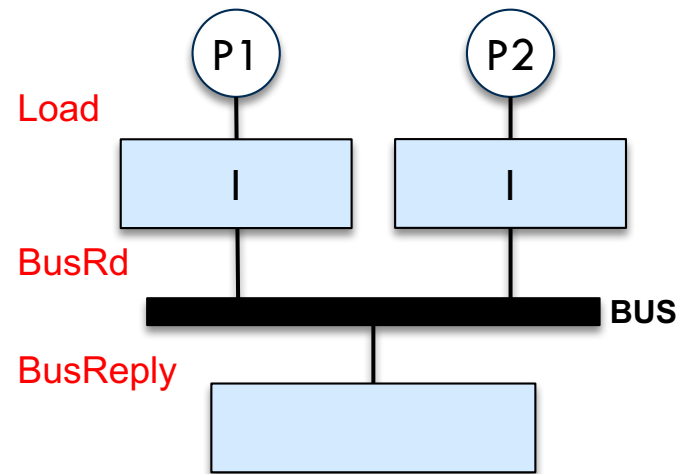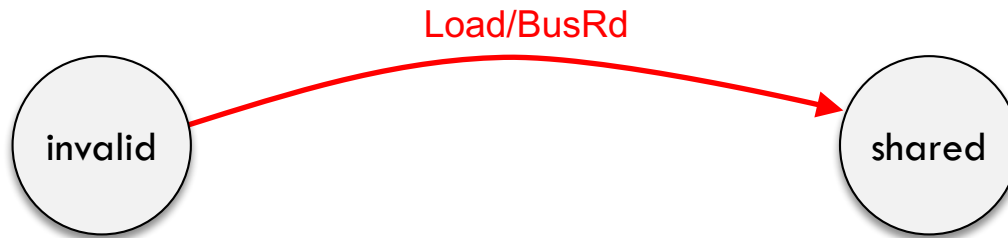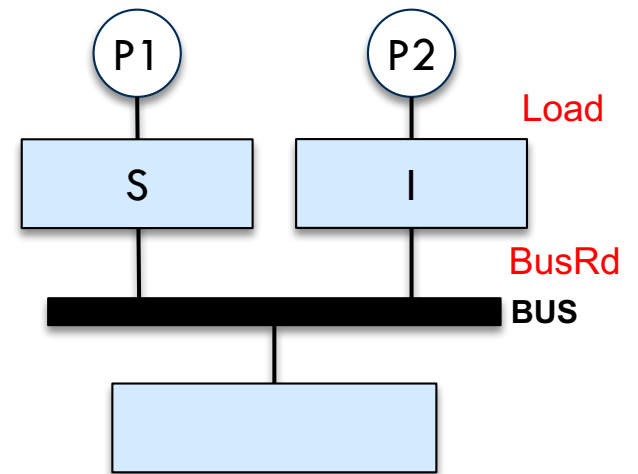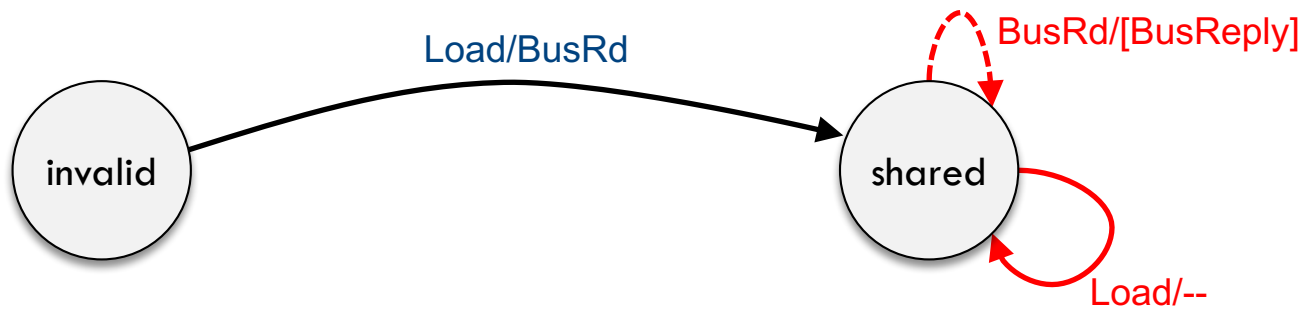
# MSI Example

# MSI Example

# MSI Example

# MSI Example
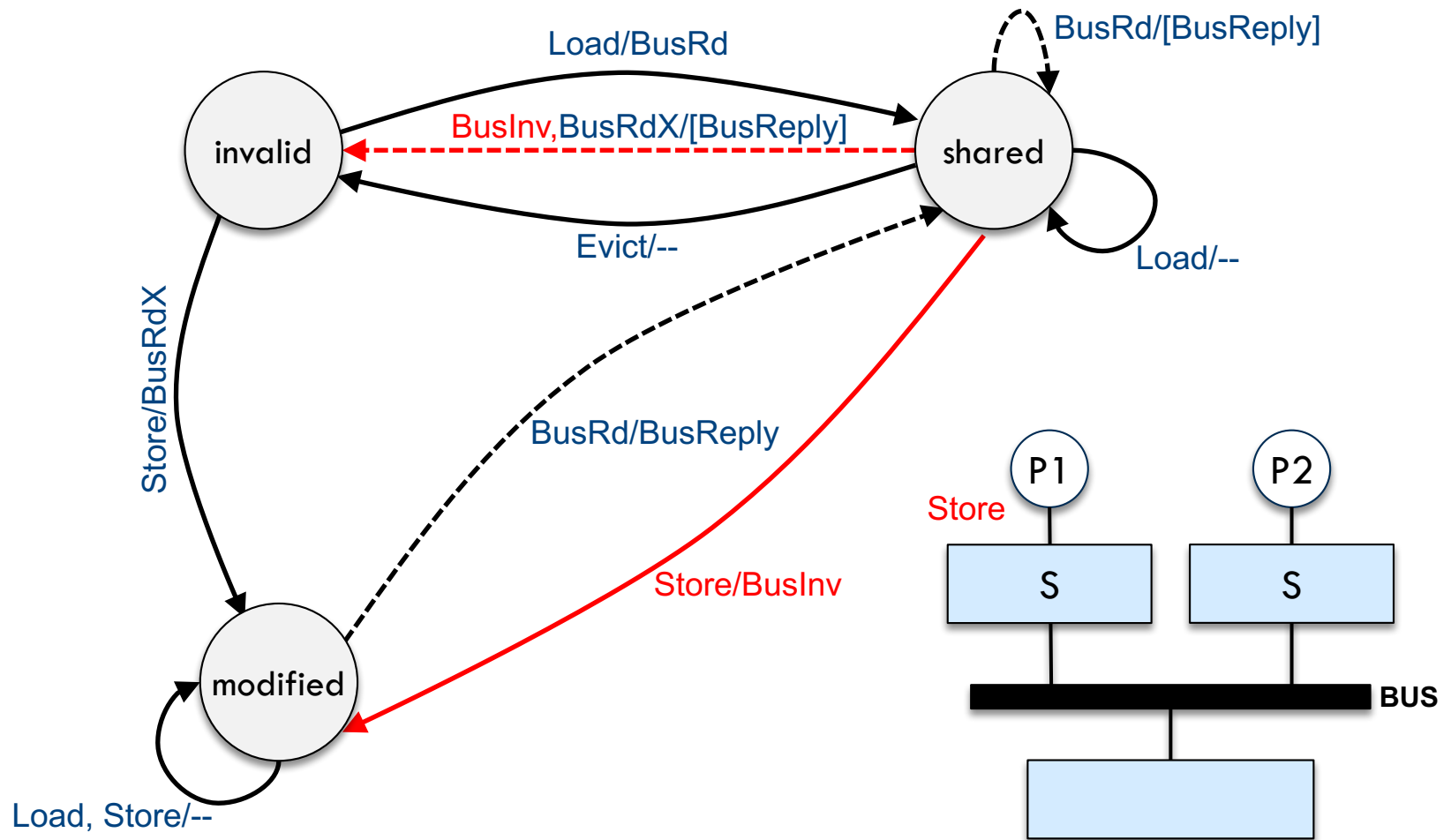
# MSI Example
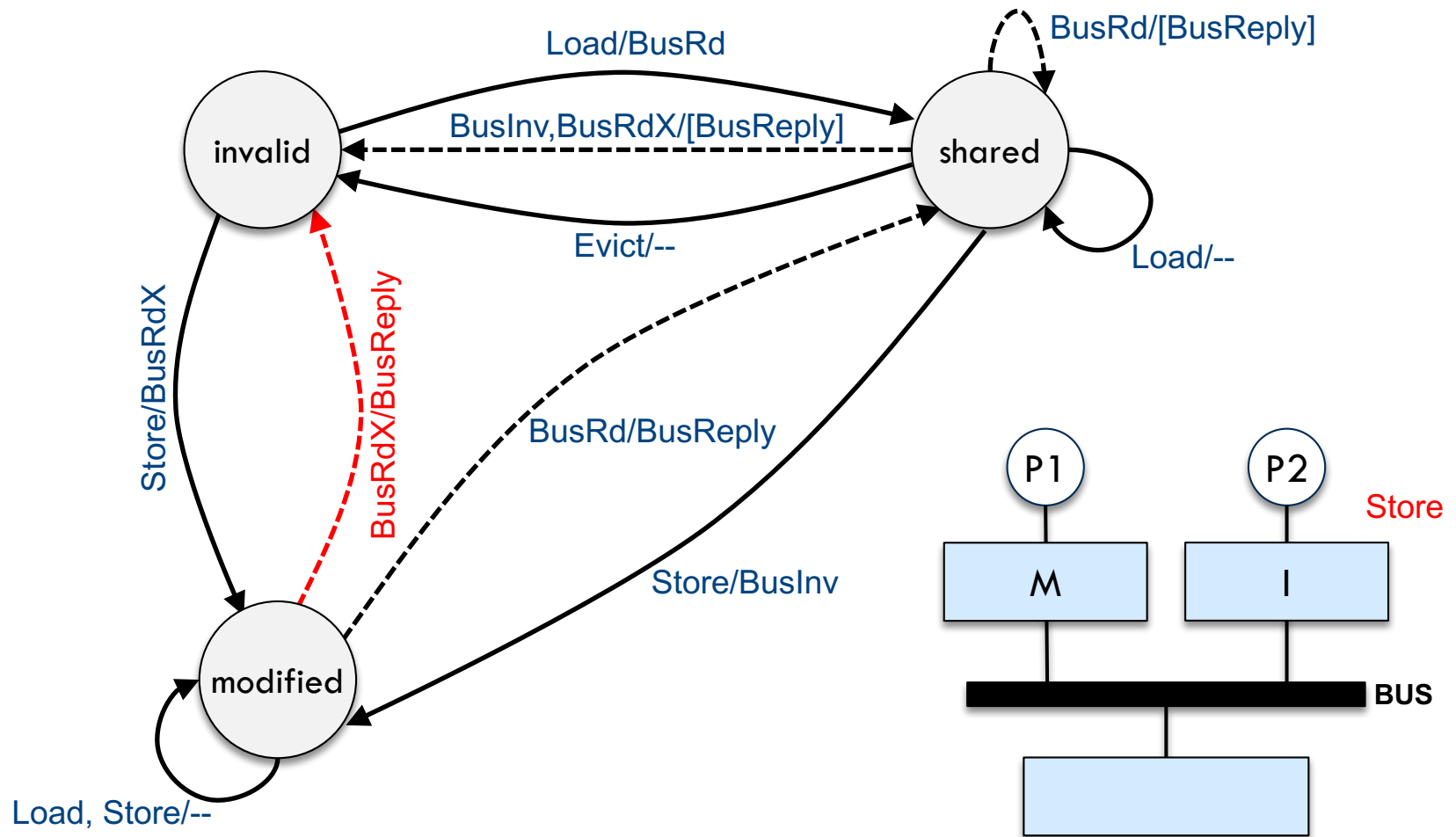
# MSI Example

# MSI Example
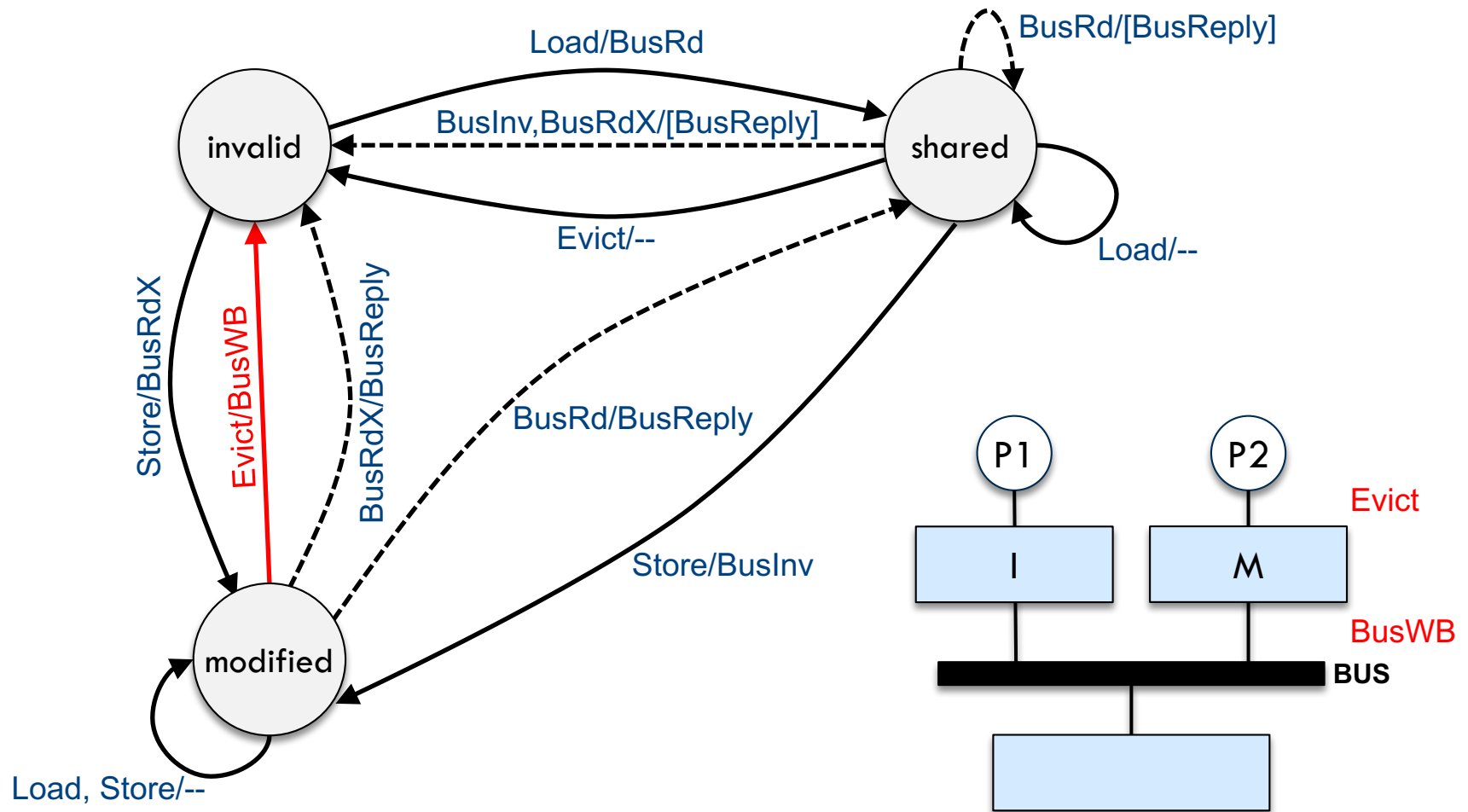
# Modified, Exclusive, Shared, Invalid

- Also known as Illinois protocol
  - Employed by real processors
  - A cache may have an exclusive copy of the data
  - The exclusive copy may be copied between caches
- Pros
  - No invalidation traffic on write-hits in the E state
  - Lower overheads in sequential applications
- Cons
  - More complex protocol
  - Longer memory latency due to the protocol

# Alternatives to Snoopy Protocols

- Problem: snooping based protocols are not scalable
  - Shared bus bandwidth is limited
  - Every node broadcasts messages and monitors the bus

- Solution: limit the traffic using directory structures
  - Home directory keeps track of sharers of each block
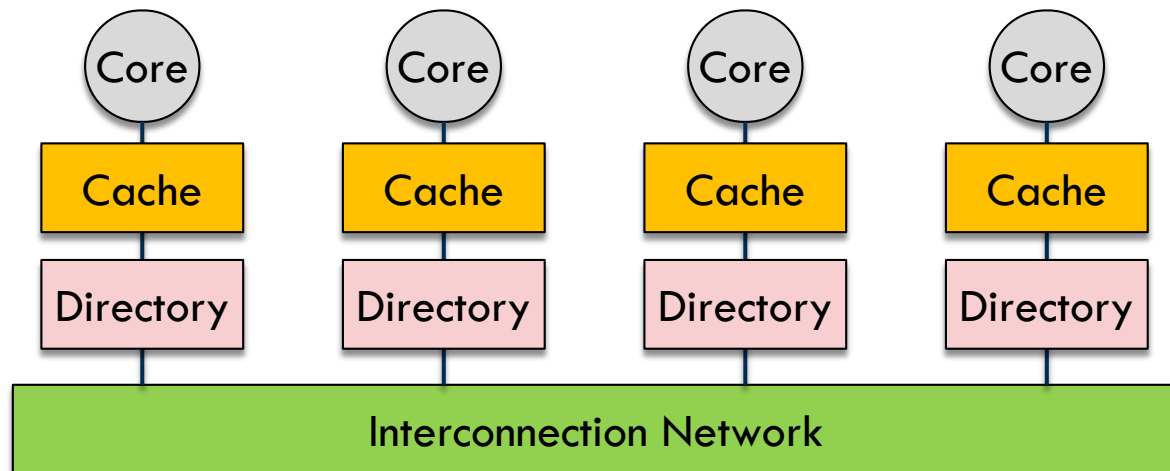
# Memory Consistency Model

☐ Memory operations are reordered to improve performance

☐ A memory consistency model for a shared address space specifies constraints on the order in which memory operations must appear to be performed with respect to one another.

Initially A = flag = 0

| P1 | P2 |
|---|---|
| A=1; | while (flag==0); |
| flag = 1; | printf ("%d", A); |

**What is the expected output of this application?**

# Memory Consistency

- Recall: load-store queue architecture
  - Check availability of operands
  - Compute the effective address
  - Send the request to memory if no memory hazards

Initially A = flag = 0

| P1 | P2 | |
|---|---|---|
| (2) A=1; | while (flag==0); | 0 |
| (1) flag = 1; | printf ("%d", A); | 1 |

# Dekker's Algorithm Example

□ Critical region with mutually exclusive access

   �« Any time, one process is allowed to be in the region

□ Reordering in load-store queue may result in failure

Initially A = B = 0

| P1 | P2 |
|---|---|
| (2)  LOCK_A: **A** = 1; | (2)  LOCK_B: **B** = 1; |
| (1)  if (B != 0) { | (1)  if (A != 0) { |
|          **A** = 0; |          **B** = 0; |
|          goto LOCK_A; |          goto LOCK_B; |
|      } |      } |
|      // … |      // … |
|      **A** = 0; |      **B** = 0; |

# Sequential Consistency

- 1. within a program, program order is preserved

- 2. each instruction executes atomically

- 3. instructions from different threads can be interleaved arbitrarily

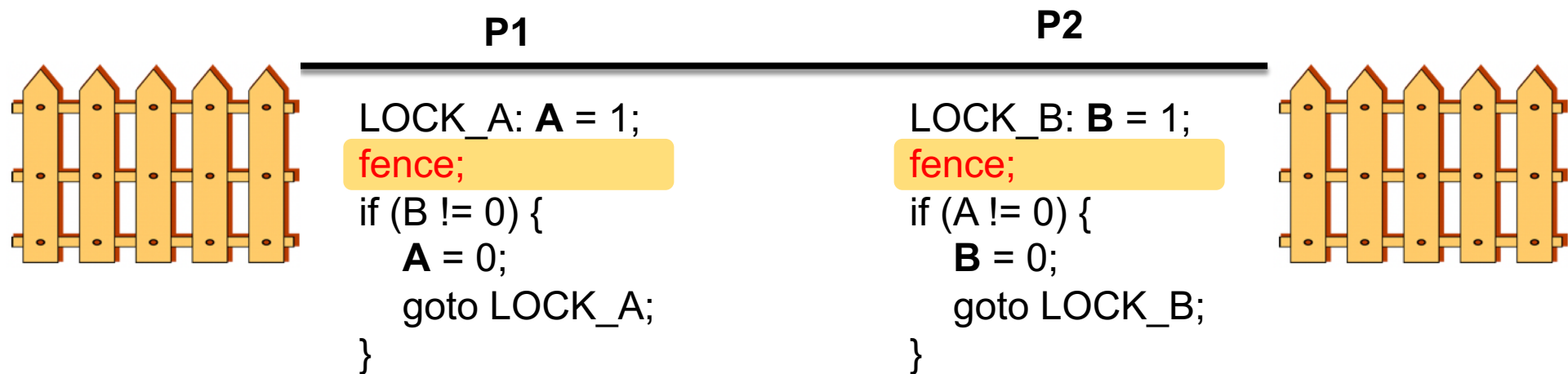| P1 | P2 |
|----|----|
| a | A |
| b | B |
| c | C |
| d | D |

1. abAcBCDdeE
2. aAbBcCdDeE
3. ABCDEabcde
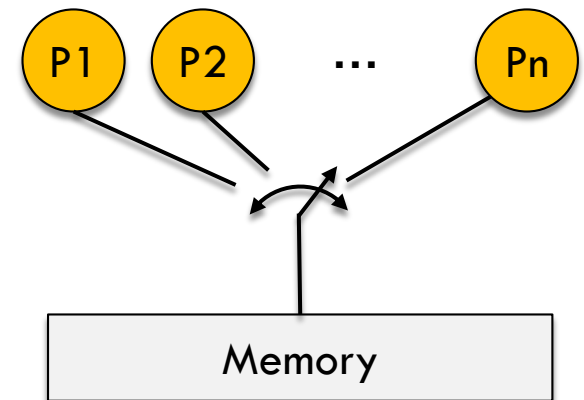


**Bad Performance!**

# Relaxed Consistency Model

- ☐ Real processors do not implement sequential consistency
  - ☐ Not all instructions need to be executed in program order
  - ☐ e.g., a read can bypass earlier writes
- ☐ A fence instruction can be used to enforce ordering among memory instructions
  - ☐ e.g., Dekker's algorithm with fence

| P1 | P2 |
|---|---|
| LOCK_A: **A** = 1;<br>fence;<br>if (B != 0) {<br>  **A** = 0;<br>  goto LOCK_A;<br>} | LOCK_B: **B** = 1;<br>fence;<br>if (A != 0) {<br>  **B** = 0;<br>  goto LOCK_B;<br>} |

# Fence Example

**P1**

{
   Region of code
   with no races
}

Fence
Acquire_lock
Fence

{
   Racy code
}

Fence
Release_lock
Fence

**P2**

{
   Region of code
   with no races
}

Fence
Acquire_lock
Fence

{
   Racy code
}

Fence
Release_lock
Fence