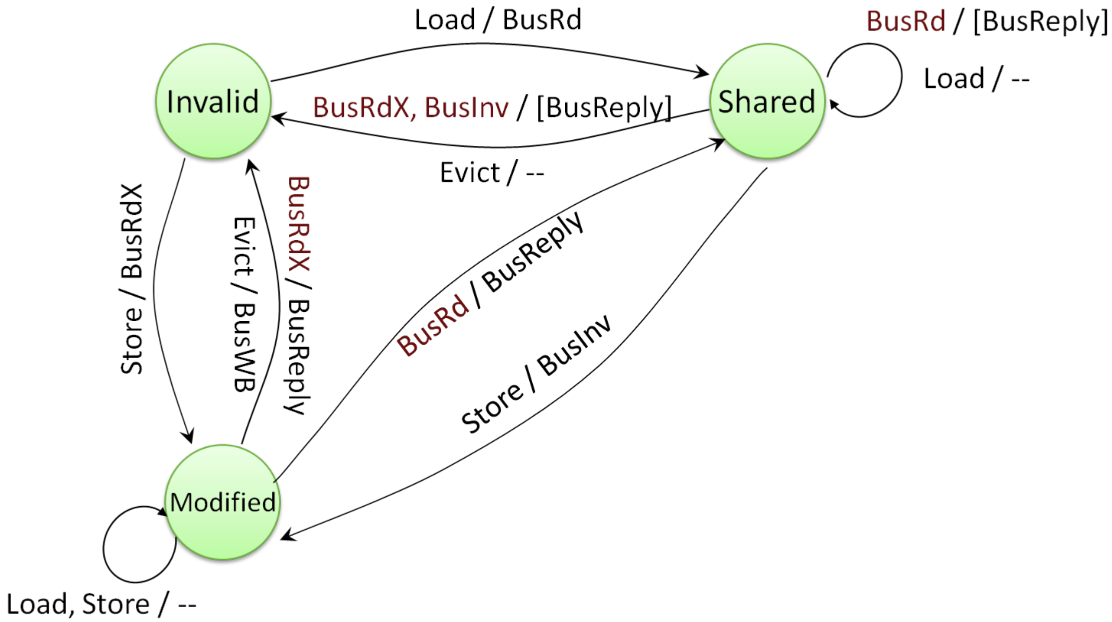
EECS 570

Lecture 8

Snooping Coherence



Winter 2025

Prof. Satish Narayanasamy

http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Falsafi, Hardavellas, Nowatzyk, and Wenisch of EPFL, Northwestern, CMU, U-M.

Readings

Daniel J. Sorin, Mark D. Hill, and David A. Wood, A Primer on Memory Consistency and Cache Coherence (Ch. 6 & 7)

Unit 2 – Cache Coherence & Memory Consistency

Cache Coherence

Processor 0

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

Processor 1

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

CPU0	CPU1	Mem

- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track `accts[241].bal` (address is in `r3`)

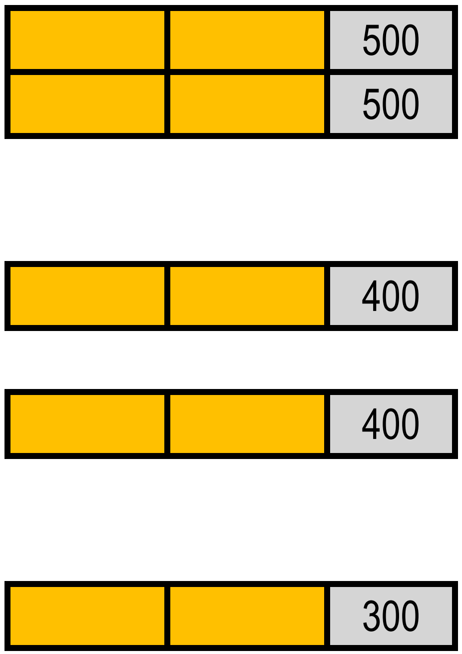
No-Cache, No-Problem

Processor 0

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

Processor 1

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```



- Scenario I: processors have no caches
  - No problem
  - Only one location where the value can reside!

Cache Incoherence

Processor 0

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

Processor 1

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

		500
V:500		500

D:400		500

D:400	V:500	500

D:400	D:400	500

- Scenario II: processors have write-back caches
  - Potentially 3 copies of **accts[241].bal**: memory, p0$, p1$
  - Can get incoherent (out of sync)

Coherence, more formally defined

- Coherence can be thought of as two invariants:
- **SWMR** = Single-Writer Multiple Readers
  - There is either one writer or zero or more readers of a cache line at any (logical) time
- **DVI** = Data Value Invariant
  - All cores see the values of the address/line update in the same order
  - e.g. if core 0 observes x go from 0 -> 1 -> 3, then all other cores must observe this sequence of updates as well

Snooping Cache-Coherence Protocols

Bus provides serialization point

Each cache controller “snoops” all bus transactions

- take action to ensure coherence
  - invalidate
  - update
  - supply value
- depends on state of the block and the protocol

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)

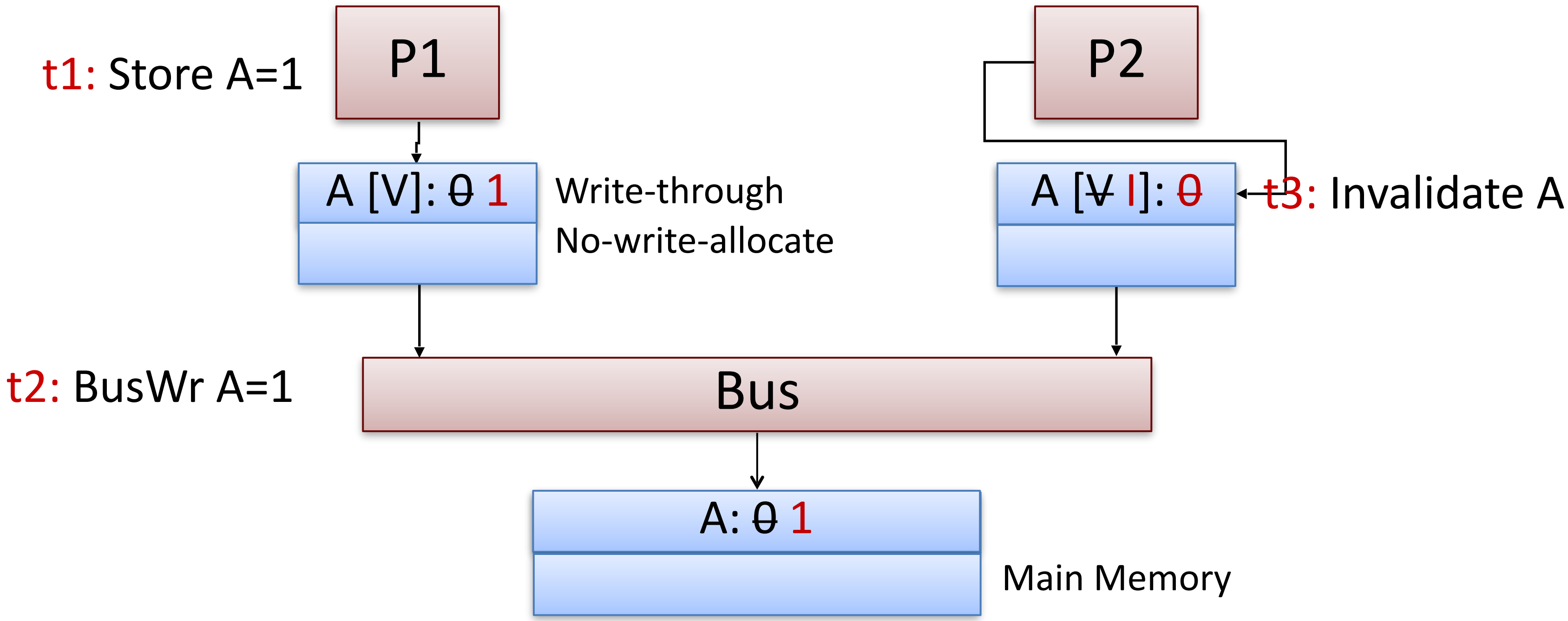
- Part II: **processor snooping bandwidth**
  - Interesting: most snoops result in no action
  - Replace non-scalable broadcast protocol (spam everyone)…
  - …with scalable **directory protocol** (only spam processors that care)

Approaches to Cache Coherence

- Software-based solutions
  - Mechanisms:
    - Mark cache blocks/memory pages as cacheable/non-cacheable
    - Add "Flush" and "Invalidate" instructions
    - *When are each of these needed?*
  - Could be done by compiler or run-time system
  - Difficult to get perfect (e.g., what about memory aliasing?)
  - Will revisit this briefly later

- Hardware solutions are far more common
  - Today we will study schemes that rely on broadcast over a bus

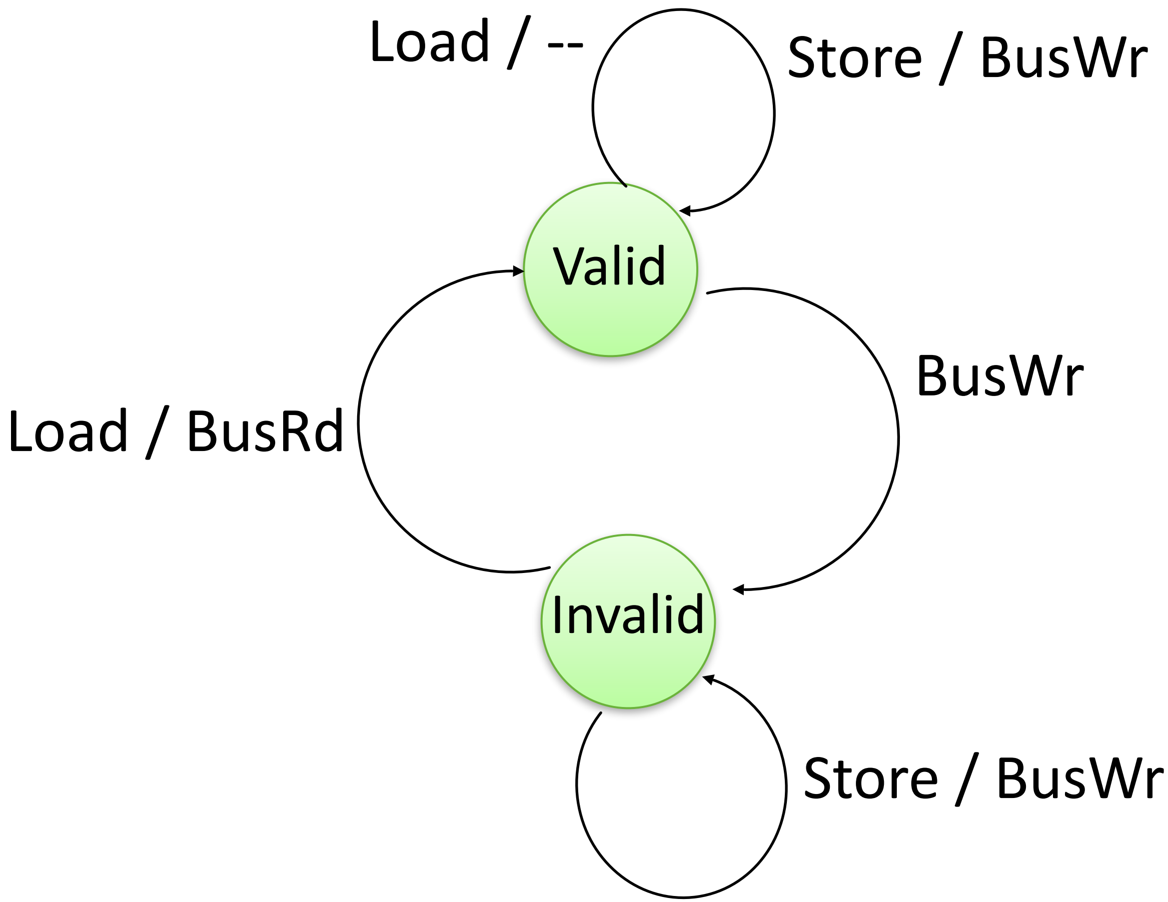
Write-Through Scheme 1:
Valid-Invalid Coherence



Valid-Invalid Coherence

- Allows multiple readers, but must write through to bus
  → Write-through, no-write-allocate cache
- All caches must monitor (aka "snoop") all bus traffic
  - simple state machine for each cache frame

Valid-Invalid Snooping Protocol

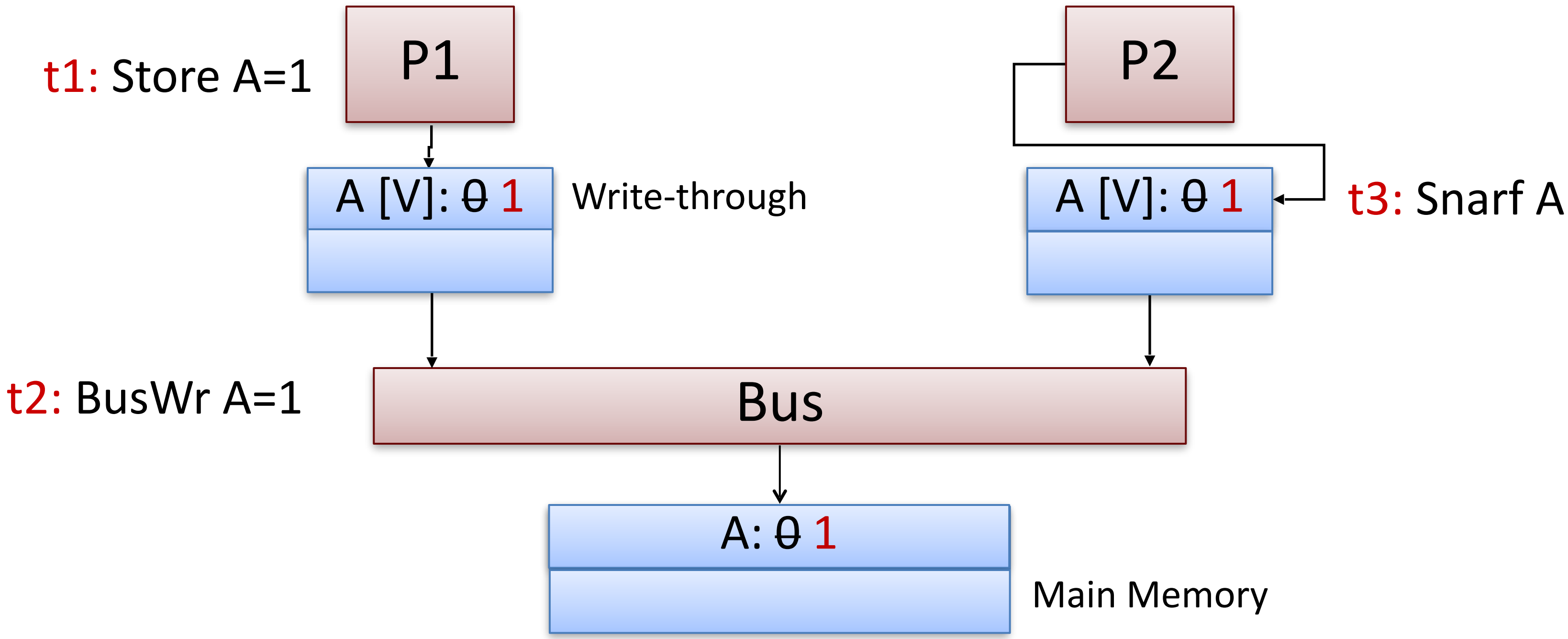


Actions:
  Ld, St, BusRd, BusWr

Write-through,
  no-write-allocate
  cache

1 bit of storage
  overhead per
  cache frame

Write Through Scheme 2: Write-Update Coherence



Write-Update Coherence

- Instead of invalidation, “Snarf” new value of A off the Bus
- But, 15% of cache accesses are stores
  - Tremendous bus and cache tag BW requirement

Supporting Write-Back Caches

- Write-back caches drastically reduce bus write bandwidth

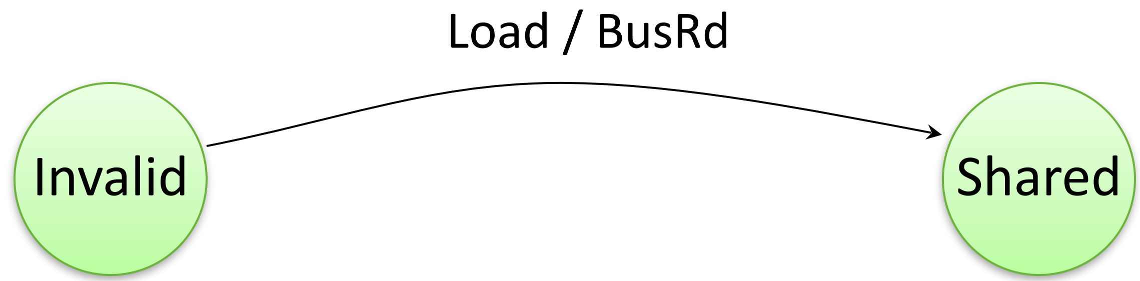
- Key idea: add notion of "ownership" to Valid-Invalid
  - Mutual exclusion – when "owner" has only replica of a cache block, it may update it freely
  - Sharing – multiple readers are ok, but they may not write without gaining ownership

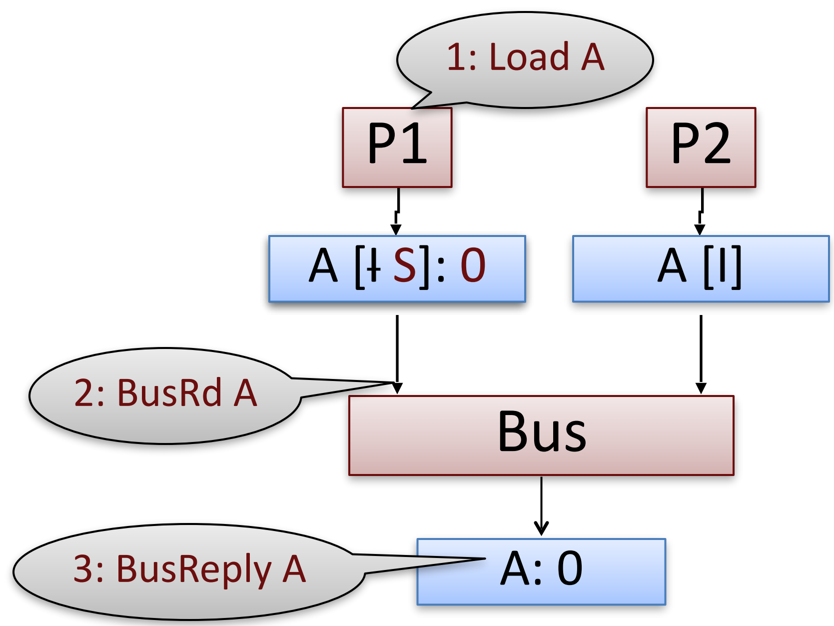
  - Need to find which cache (if any) is an owner on read misses
  - Need to eventually update memory so writes are not lost

Modified-Shared-Invalid (MSI) Protocol

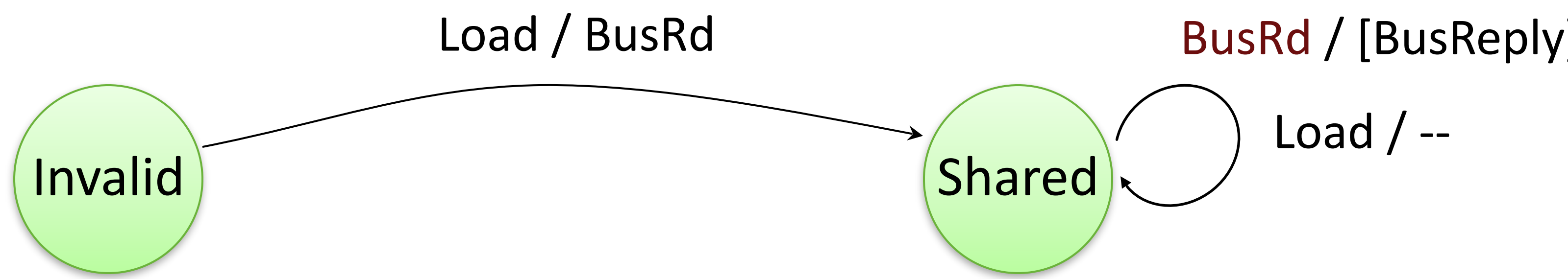
- Three states tracked per-block at each cache
  - Invalid – cache does not have a copy
  - Shared – cache has a read-only copy; clean
    - Clean == memory is up to date
  - Modified – cache has the only copy; writable; dirty
    - Dirty == memory is out of date
- Three processor actions
  - Load, Store, Evict
- Five bus messages
  - BusRd, BusRdX, BusInv, BusWB, BusReply
  - Could combine some of these

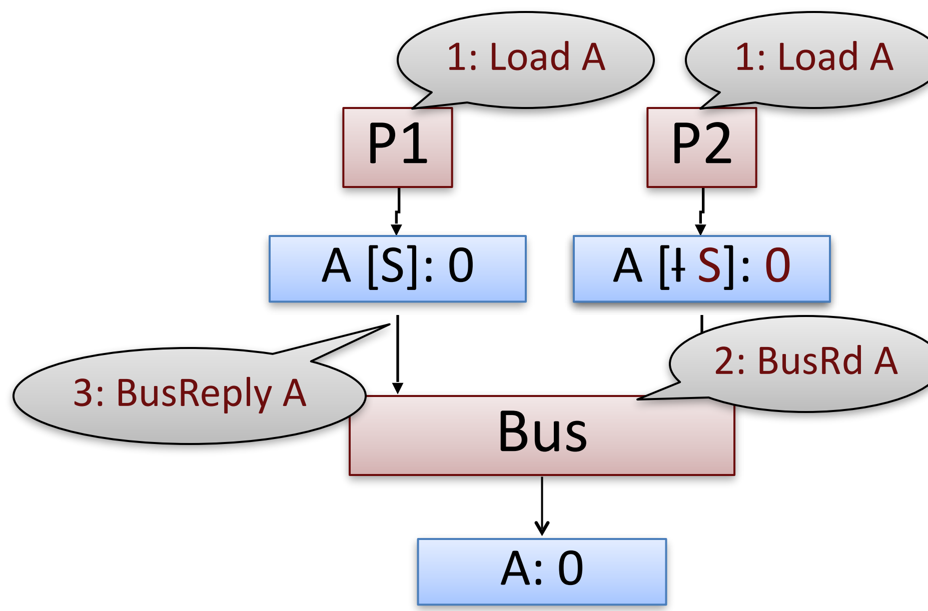
Modified-Shared -Invalid (MSI) Protocol



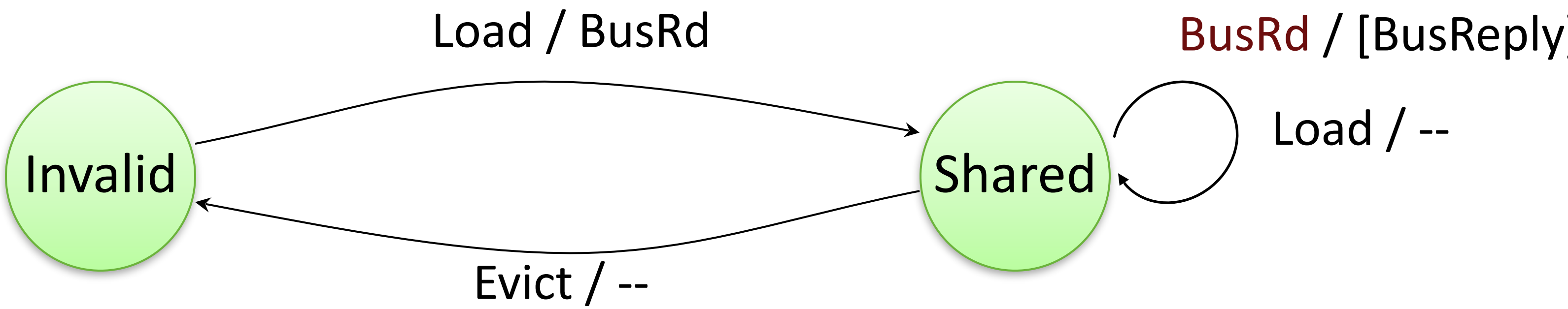


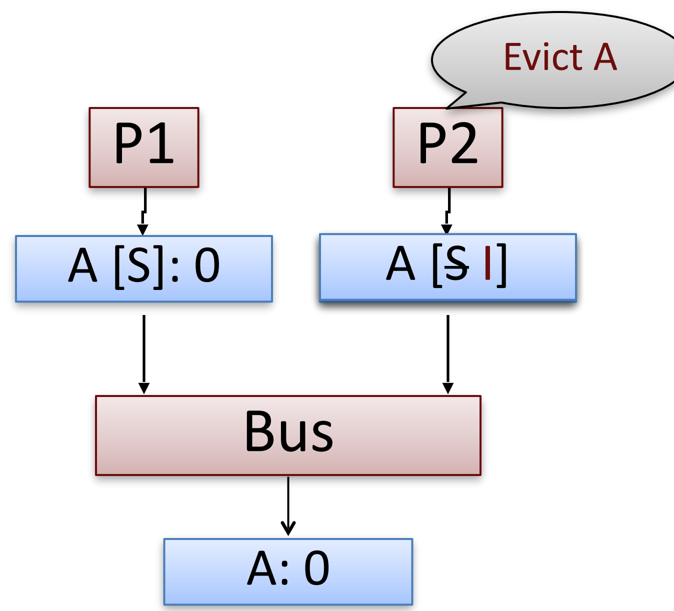
Modified-Shared -Invalid (MSI) Protocol



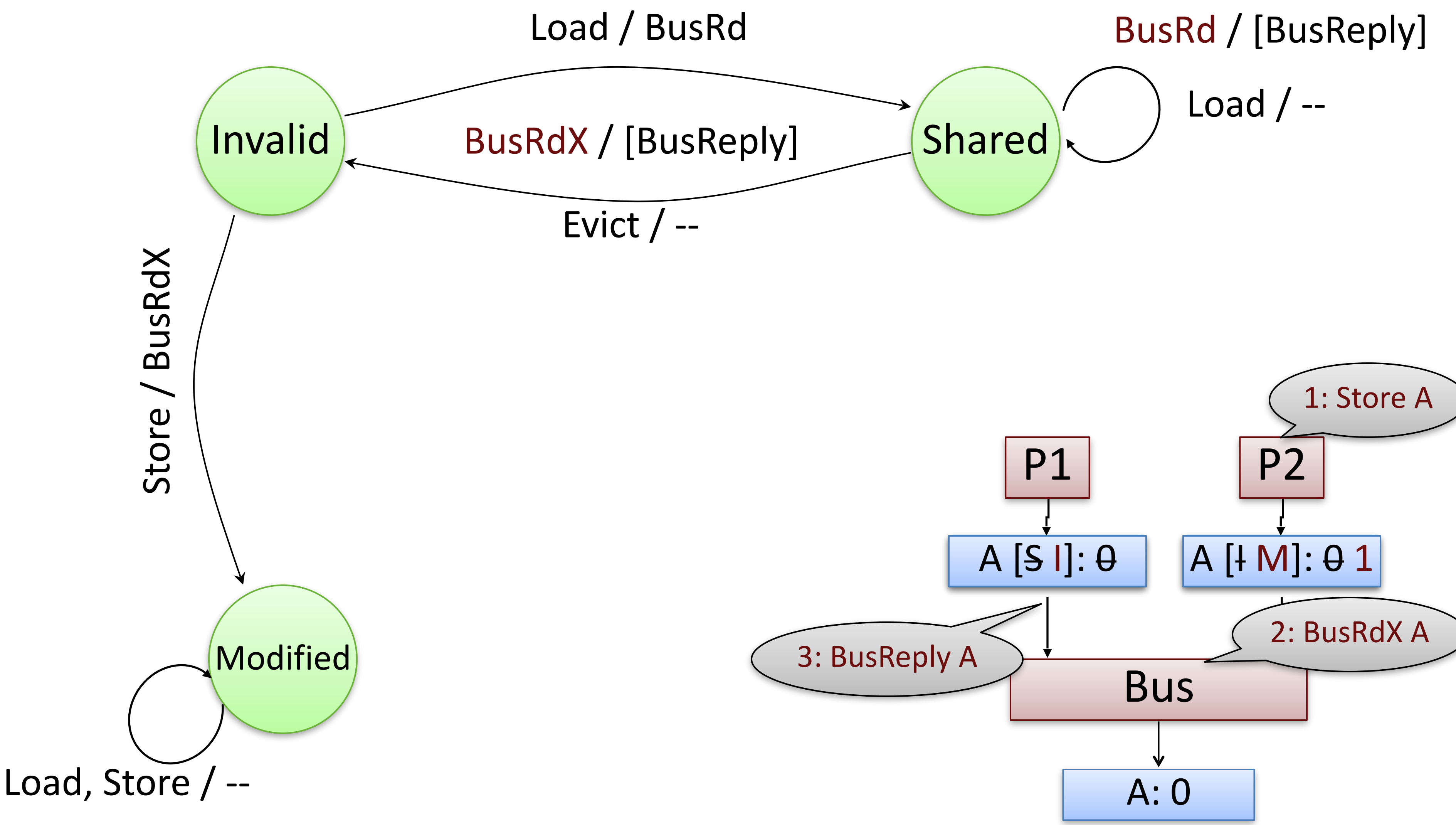


Modified-Shared -Invalid (MSI) Protocol

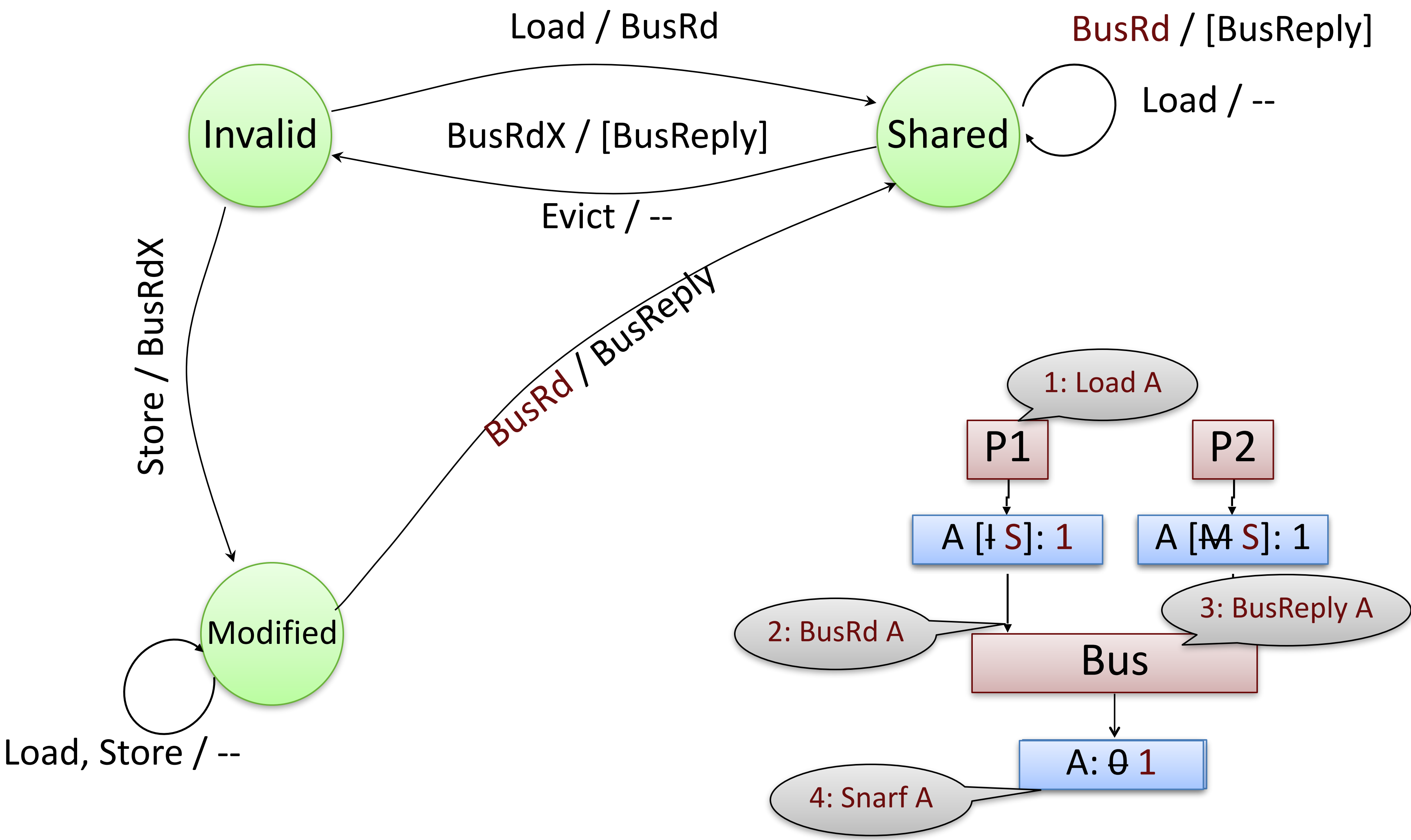




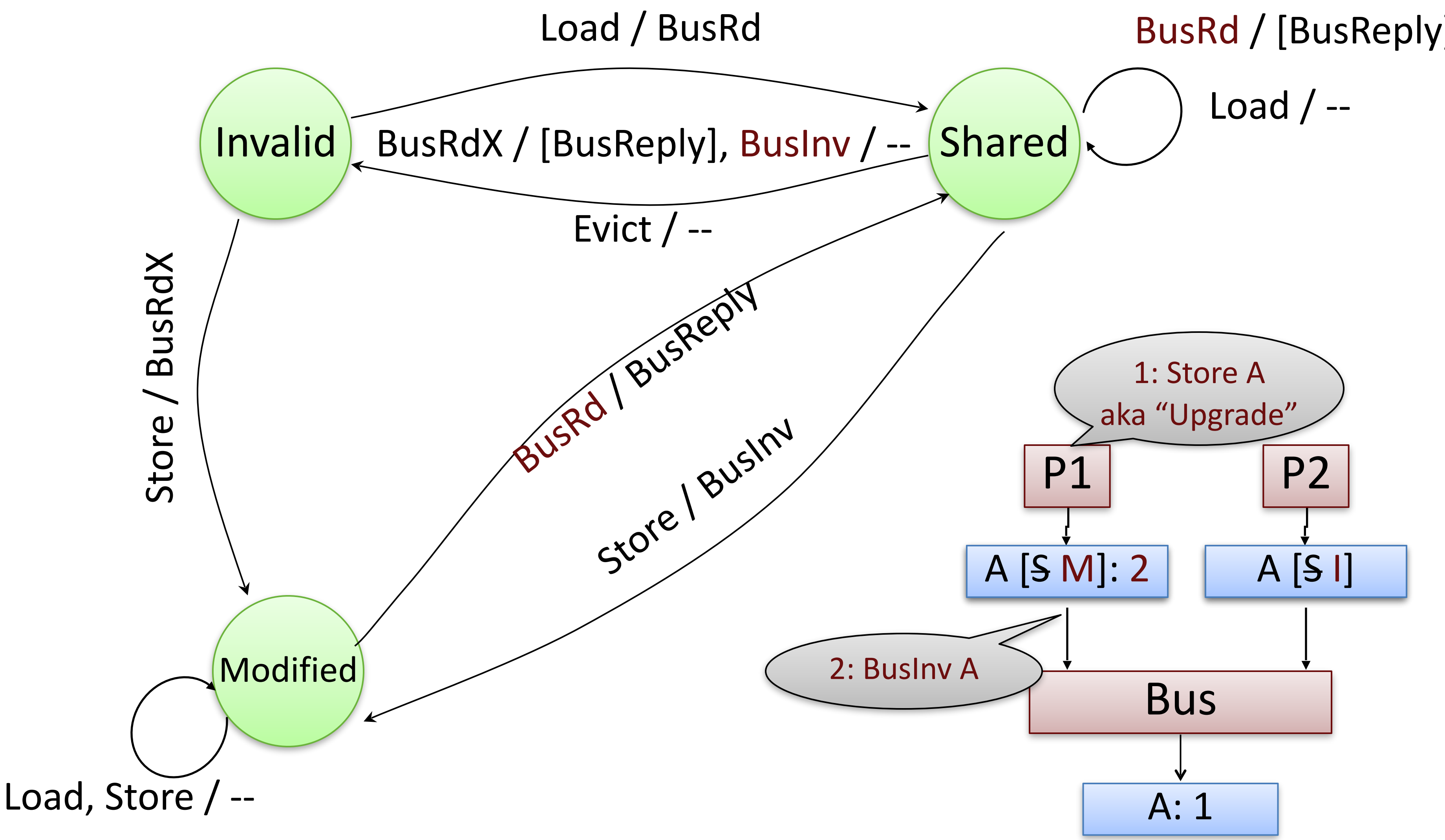
Modified-Shared -Invalid (MSI) Protocol



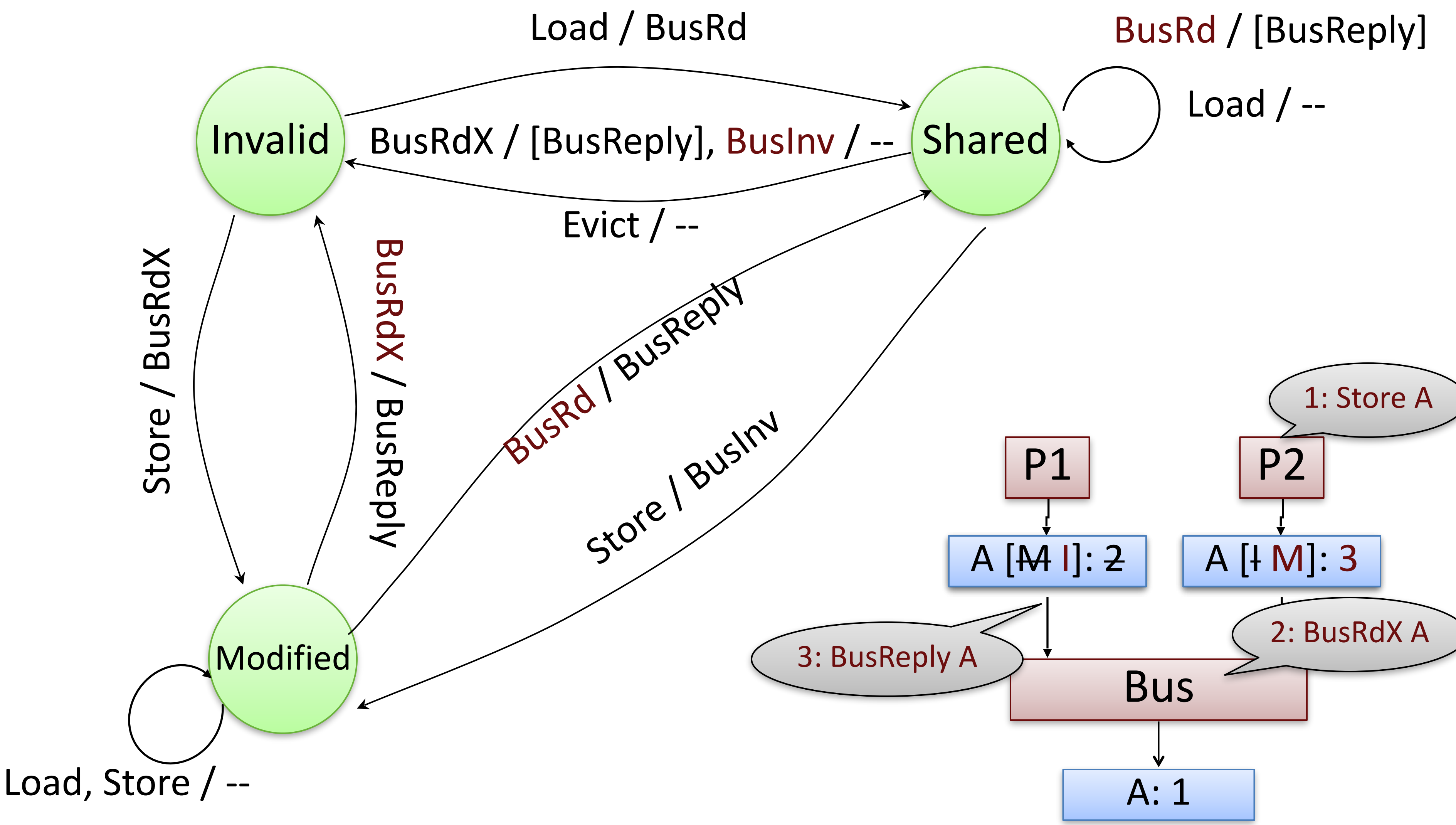
Modified-Shared -Invalid (MSI) Protocol



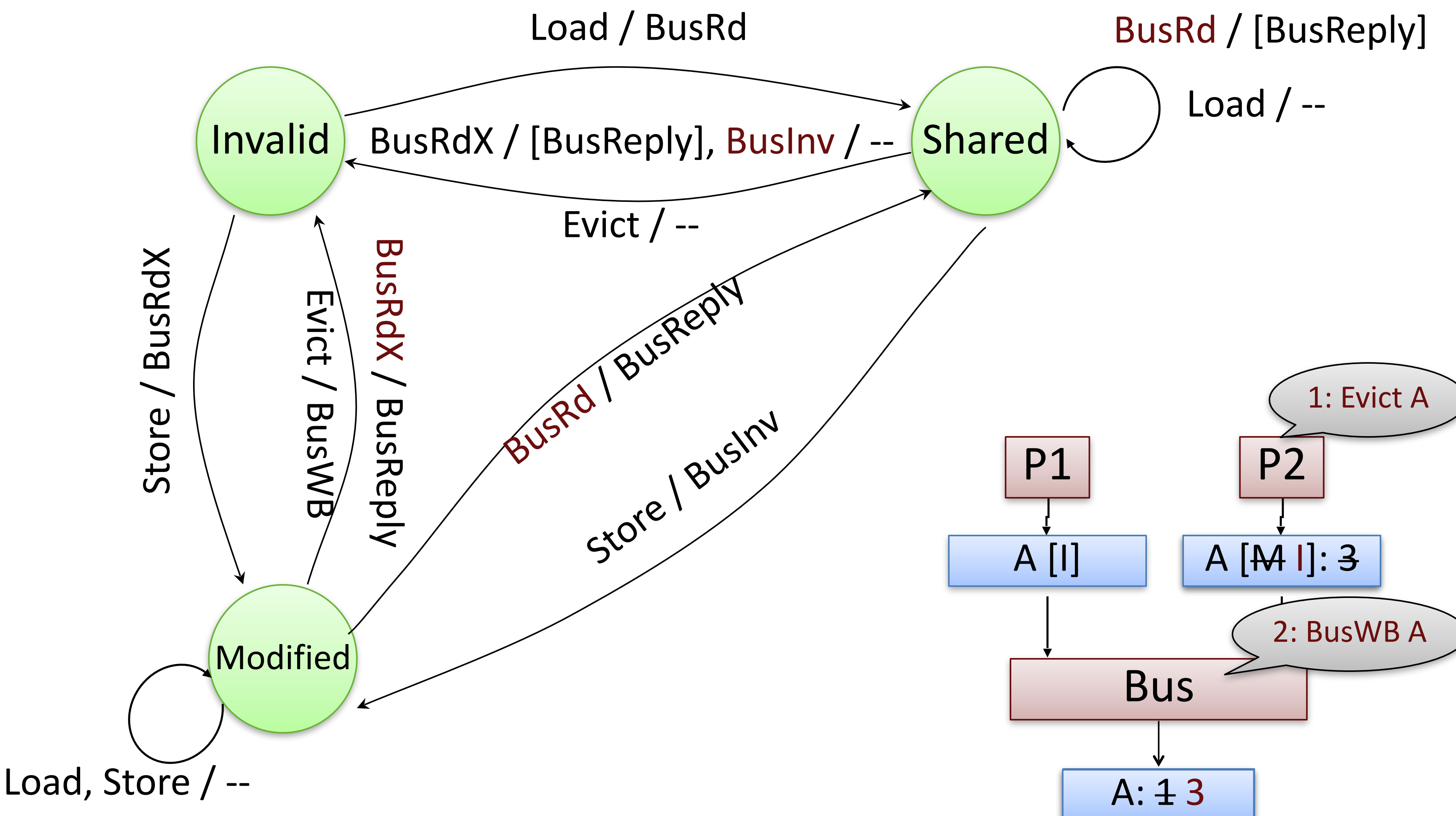
Modified-Shared -Invalid (MSI) Protocol



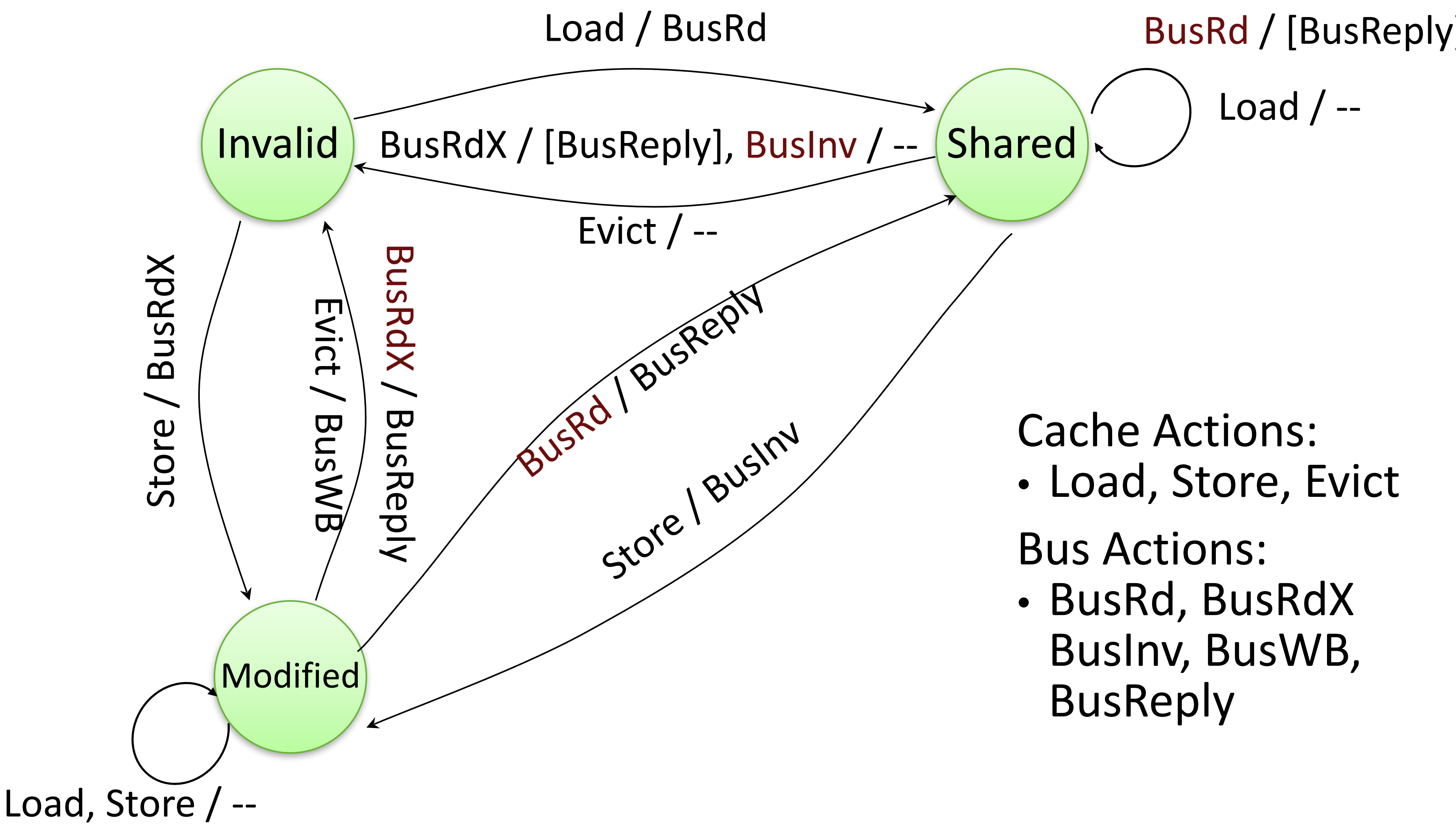
Modified-Shared -Invalid (MSI) Protocol



Modified-Shared -Invalid (MSI) Protocol



MSI Protocol Summary



Cache Actions:
- Load, Store, Evict

Bus Actions:
- BusRd, BusRdX BusInv, BusWB, BusReply

Update vs. Invalidate

- Invalidation is bad when:
  - Single producer and many consumers of data
- Update is bad when:
  - Multiple writes by one CPU before read by another
  - Junk data accumulates in large caches (e.g., process migration)

Coherence Decoupling

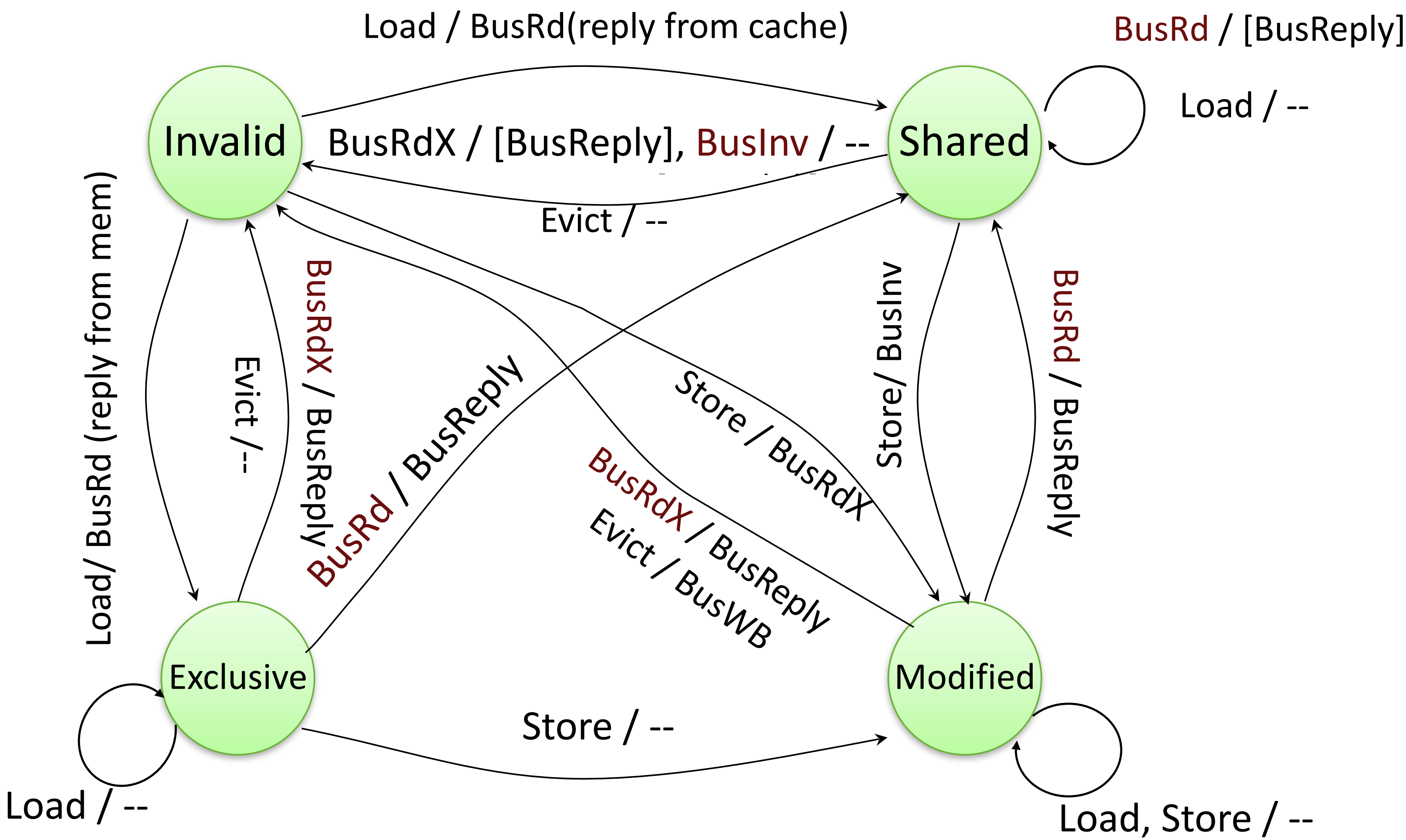
[Huh, Chang, Burger, Sohi ASPLOS04]

- After invalidate, keep stale data around
  - On subsequent read, speculatively supply stale value
  - Confirm speculation with a normal read operations
  - Need a branch-prediction-like rewind mechanism
  - Completely solves false sharing problem
  - Also addresses "silent", "temporally-silent" stores
- Can use update-like mechanisms to improve prediction
  - Paper explores a variety of update heuristics
  - E.g., piggy-back value of 1st write on invalidation message

MESI Protocol (aka Illinois)

- MSI suffers from frequent read-upgrade sequences
  - Leads to two bus transactions, even for private blocks
  - 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
  - Stores transition to Modified to indicate data is dirty
    - Can design things so that there is no need for a BusWB from Exclusive

MESI Protocol Summary



MOESI Protocol

- MESI must write-back to memory on M→S transitions
  - Because protocol allows "silent" evicts from shared state, a dirty block might otherwise be lost
  - But, the writebacks might be a waste of bandwidth
    - E.g., if there is a subsequent store
    - Common case in producer-consumer scenarios
- Solution: add an "Owned" state
  - Owned – shared, but dirty; only one owner (others enter S)
    - Entered on M→S transition, aka "downgrade"
  - Owner is responsible for writeback upon eviction

MOESI Framework

[Sweazey & Smith ISCA86]

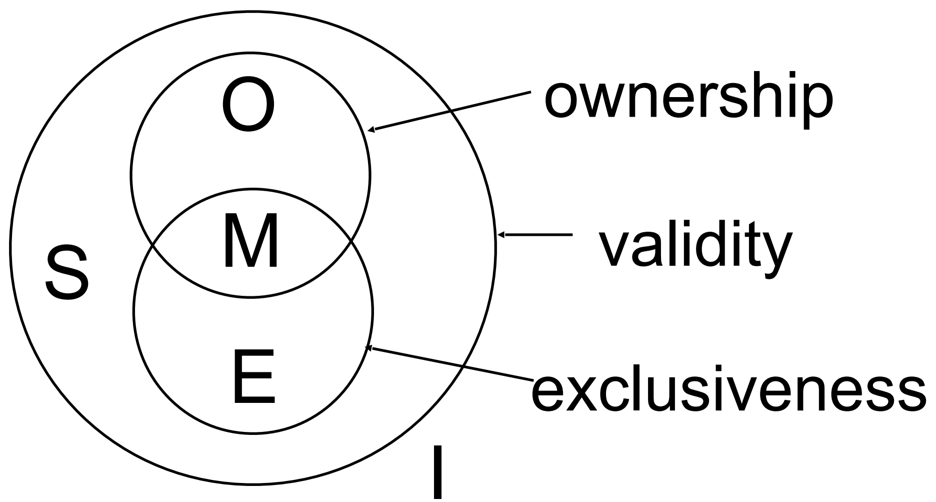
M - Modified (dirty)

O - Owned (dirty but shared)   WHY?

E - Exclusive (clean unshared) only copy, not dirty

S - Shared

I - Invalid



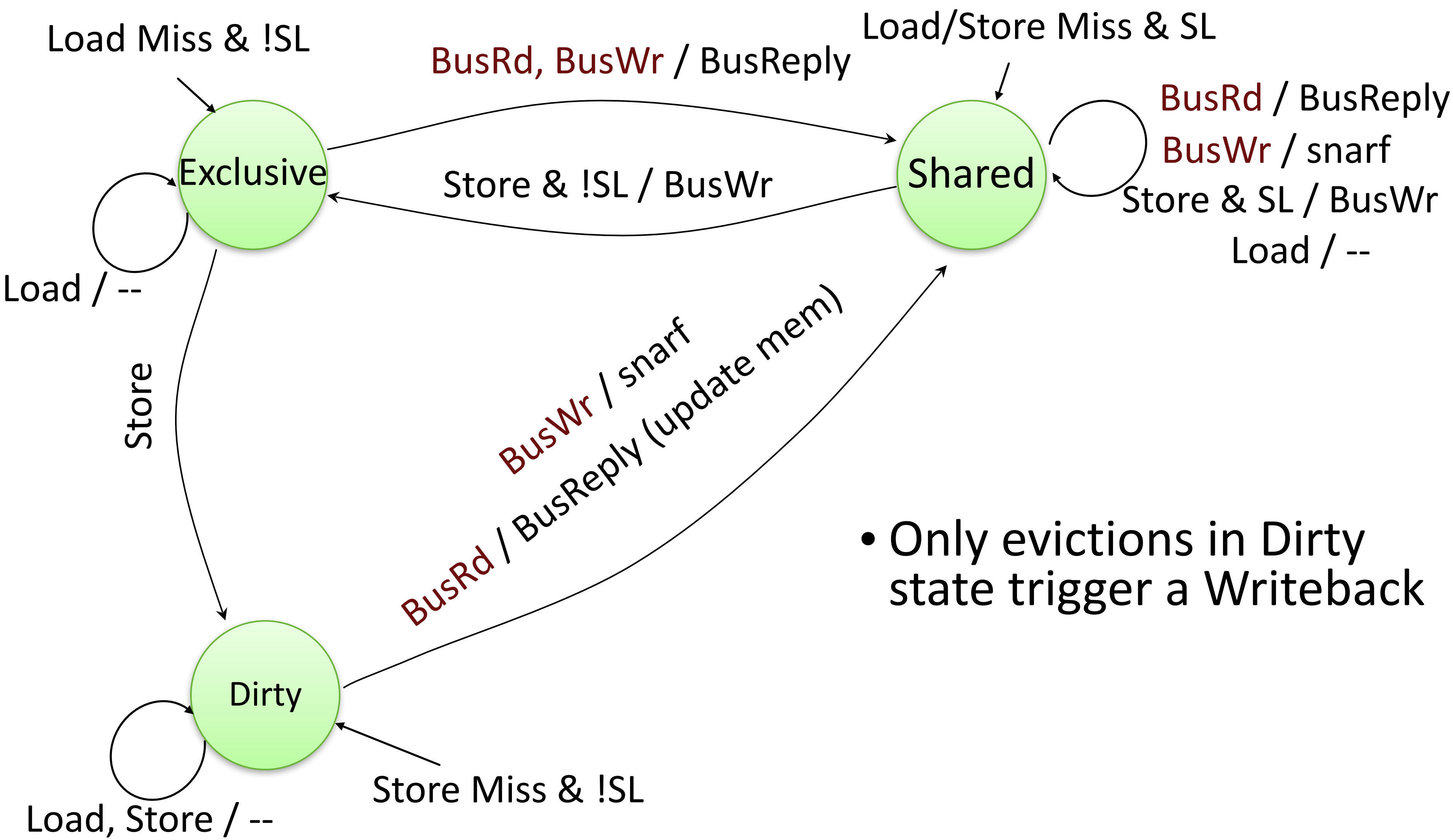
Variants

- MSI
- MESI
- MOSI
- MOESI

DEC Firefly

- An update protocol for write-back caches
- States
  - Exclusive – only one copy; writable; clean
  - Shared – multiple copies; write hits write-through to all sharers and memory
  - Dirty – only one copy; writeable; dirty
- Exclusive/dirty provide write-back semantics for private data
- Shared state provides update semantics for shared data
  - Uses “shared line” bus wire to detect sharing status
- Well suited to producer-consumer; process migration hurts

DEC Firefly Protocol Summary



- Only evictions in Dirty state trigger a Writeback

Non-Atomic State Transitions

Operations involve multiple actions

- Look up cache tags
- Bus arbitration
- Check for writeback
- Even if bus is atomic, overall set of actions is not
- Race conditions among multiple operations

Suppose P1 and P2 attempt to write cached block A

- Each decides to issue BusUpgr to allow S –> M

Issues

- Handle requests for other blocks while waiting to acquire bus
- Must handle requests for this block A

You’ll see a lot of this in PA2! ☺

Scalability problems of Snoopy Coherence

- Prohibitive **bus bandwidth**
  - Required bandwidth grows with # CPUS…
  - … but available BW per bus is fixed
  - Adding busses makes serialization/ordering hard
- Prohibitive **processor snooping bandwidth**
  - All caches do tag lookup when ANY processor accesses memory
  - Inclusion limits this to L2, but still lots of lookups

- **Upshot**: bus-based coherence doesn’t scale beyond 8–16 CPUs