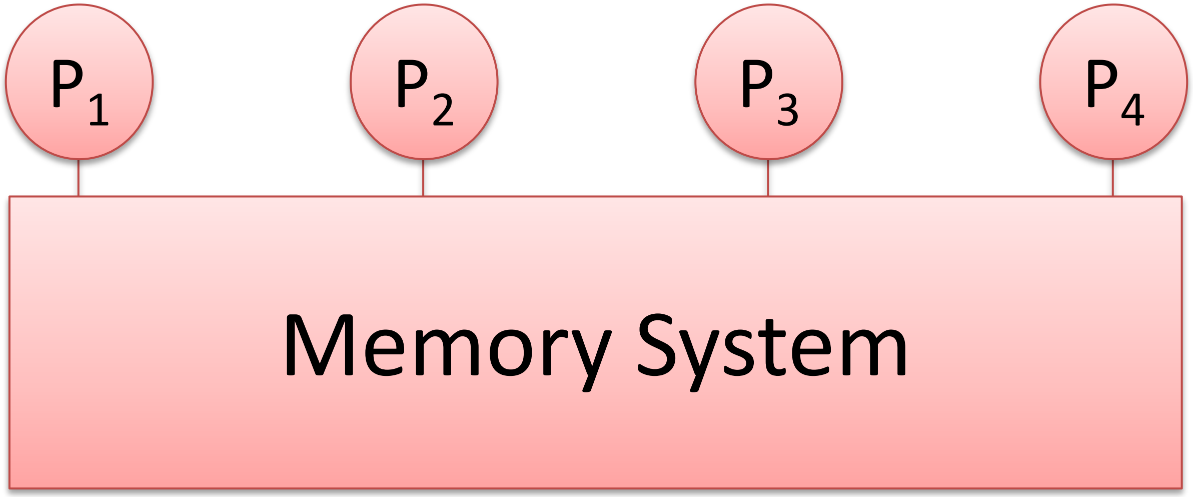
CSE 502: Computer Architecture

Shared-Memory Multi-Processors

Shared-Memory Multiprocessors

- Multiple threads use *shared memory* (address space)
  - "SysV Shared Memory" or "Threads" in software
- Communication implicit via loads and stores
  - Opposite of explicit *message-passing multiprocessors*
- Theoretical foundation: *PRAM model*

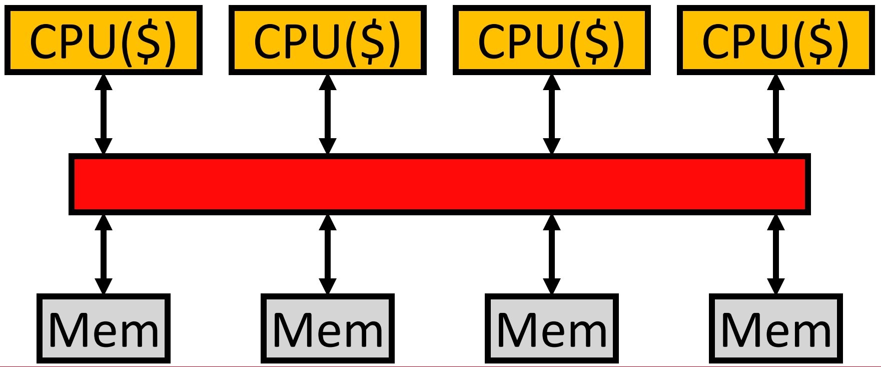


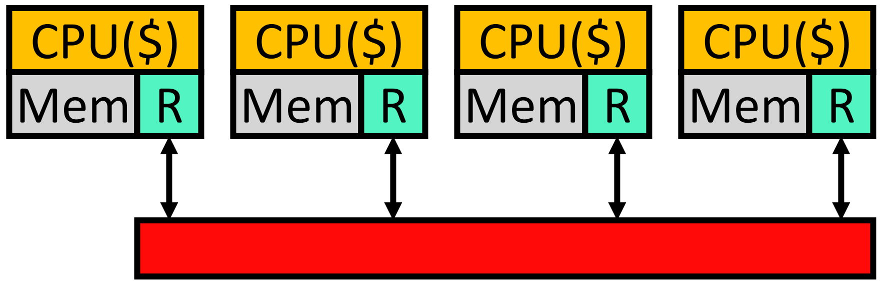
Why Shared Memory?

- Pluses
  - App sees multitasking uniprocessor
  - OS needs only evolutionary extensions
  - Communication happens without OS
- Minuses
  - Synchronization is complex
  - Communication is implicit (hard to optimize)
  - Hard to implement (in hardware)
- Result
  - SMPs and CMPs are most successful machines to date
  - First with multi-billion-dollar markets

Paired vs. Separate Processor/Memory?

- Separate CPU/memory
  - *Uniform memory access* (*UMA*)
    - Equal latency to memory
  - Low peak performance
- Paired CPU/memory
  - *Non-uniform memory access* (*NUMA*)
    - Faster local memory
    - Data placement matters
  - High peak performance

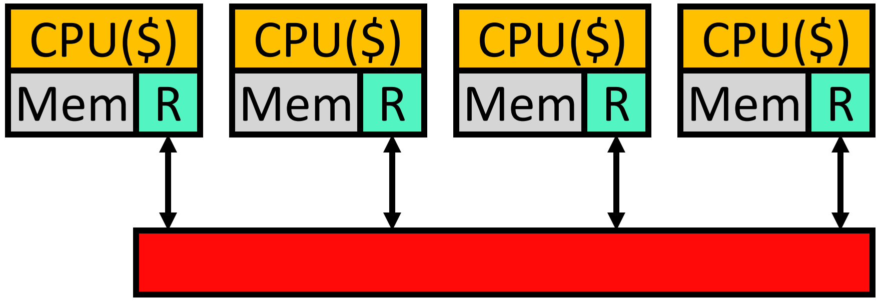


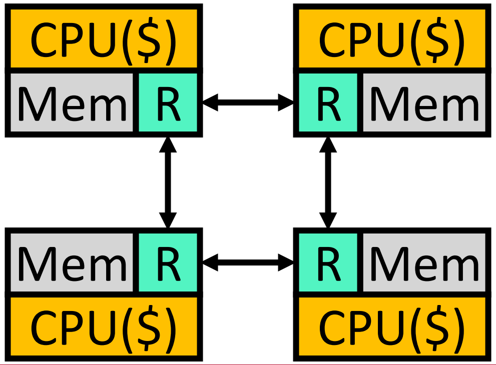


Shared vs. Point-to-Point Networks

- Shared network
  - Example: bus
  - Low latency
  - Low bandwidth
    - Doesn't scale >~16 cores
  - Simple cache coherence

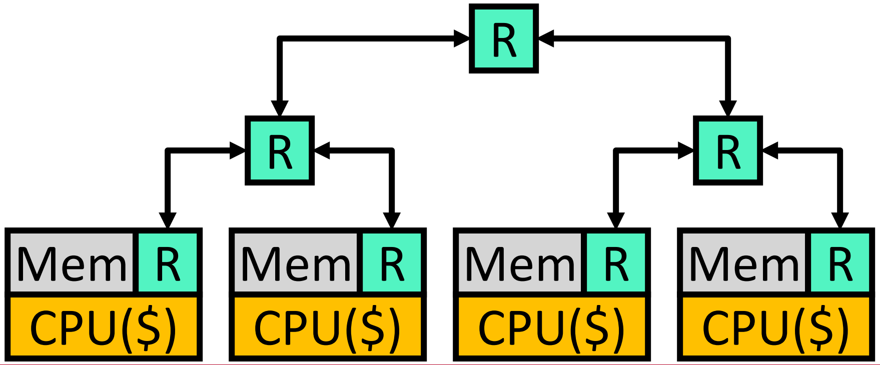
- Point-to-point network:
  - Example: mesh, ring
  - High latency (many "*hops*")
  - Higher bandwidth
    - Scales to 1000s of cores
  - Complex cache coherence

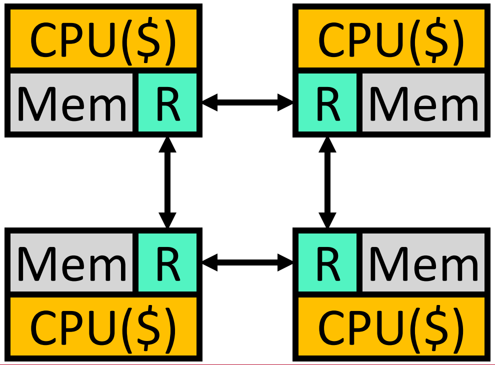




Organizing Point-To-Point Networks

- *Network topology*: organization of network
  - Tradeoff perf. (connectivity, latency, bandwidth) ↔ cost
- Router chips
  - Networks w/separate router chips are *indirect*
  - Networks w/ processor/memory/router in chip are *direct*
    - Fewer components, "*Glueless MP*"



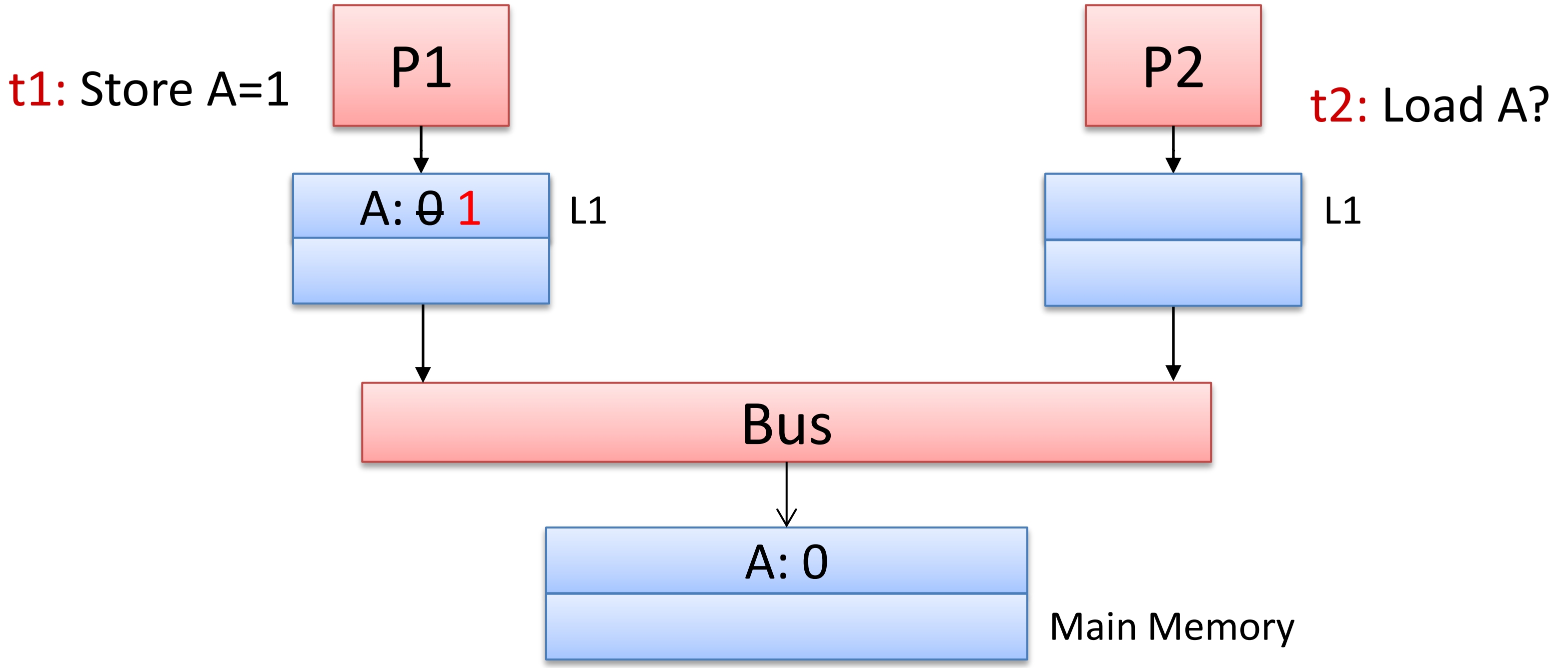


Issues for Shared Memory Systems

- Two big ones
  - Cache coherence
  - Memory consistency model
- Closely related
- Often confused

Cache Coherence: The Problem (1/2)

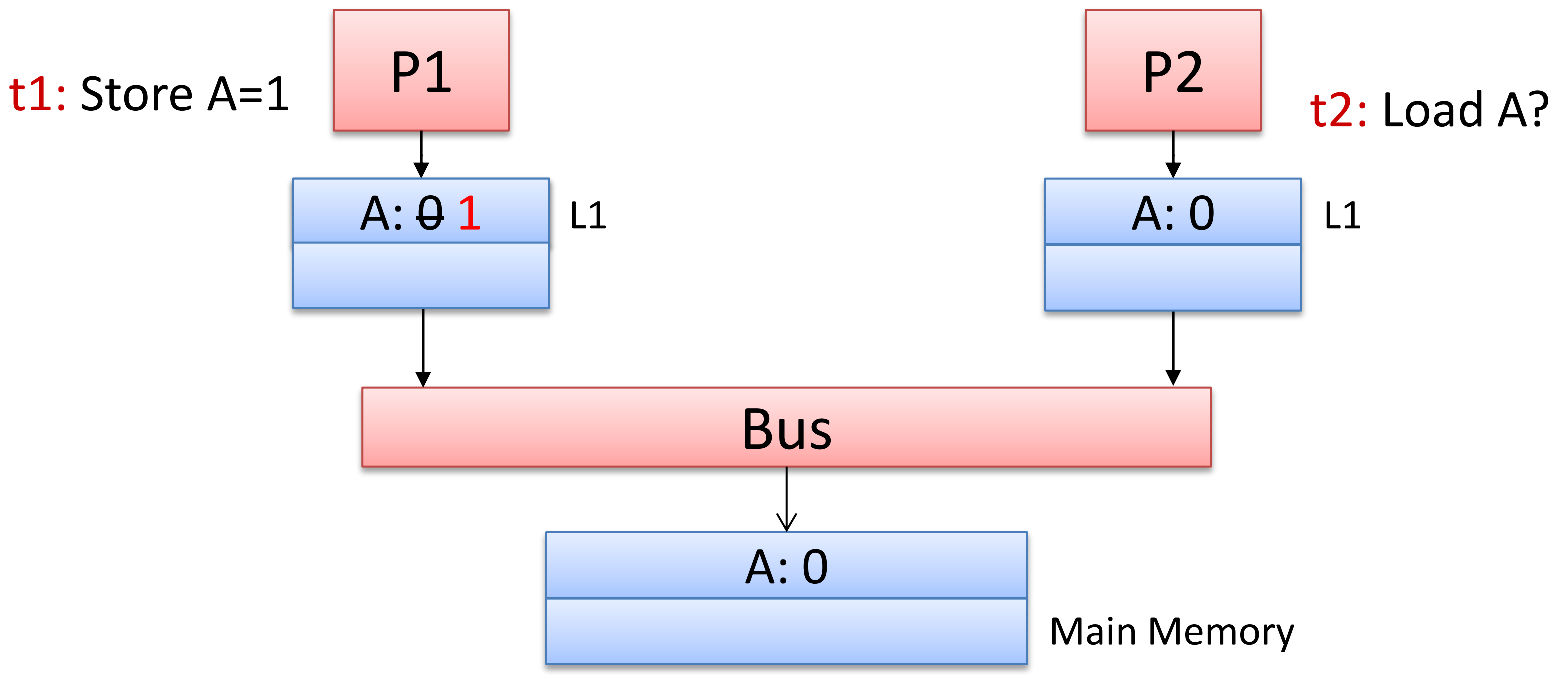
- Variable A initially has value 0
- P1 stores value 1 into A
- P2 loads A from memory and sees old value 0



Need to do something to keep P2's cache *coherent*

Cache Coherence: The Problem (2/2)

- P1 and P2 have variable A (value 0) in their caches
- P1 stores value 1 into A
- P2 loads A from its cache and sees old value 0



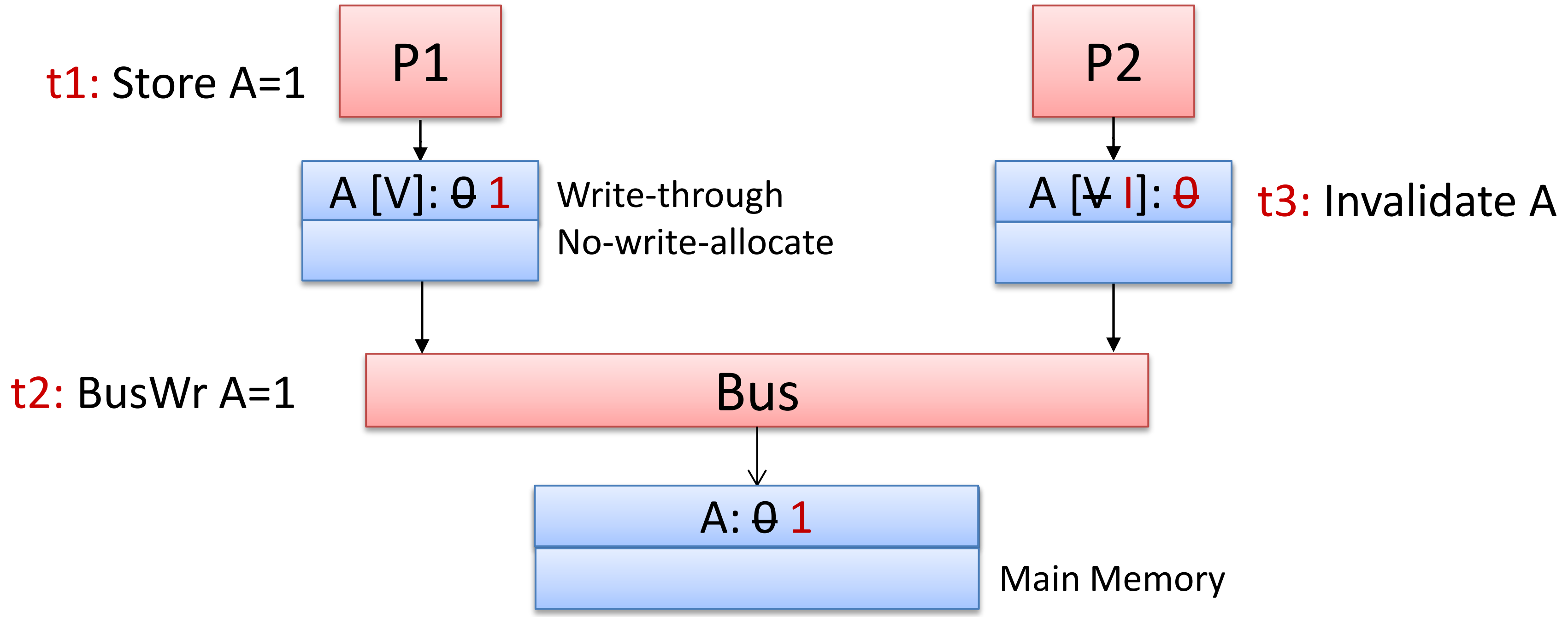
Need to do something to keep P2's cache *coherent*

Approaches to Cache Coherence

- Software-based solutions
  - Mechanisms:
    - Mark cache blocks/memory pages as cacheable/non-cacheable
    - Add “Flush” and “Invalidate” instructions
  - Could be done by compiler or run-time system
  - Difficult to get perfect (e.g., what about memory aliasing?)
- Hardware solutions are far more common
  - System ensures everyone always sees the latest value

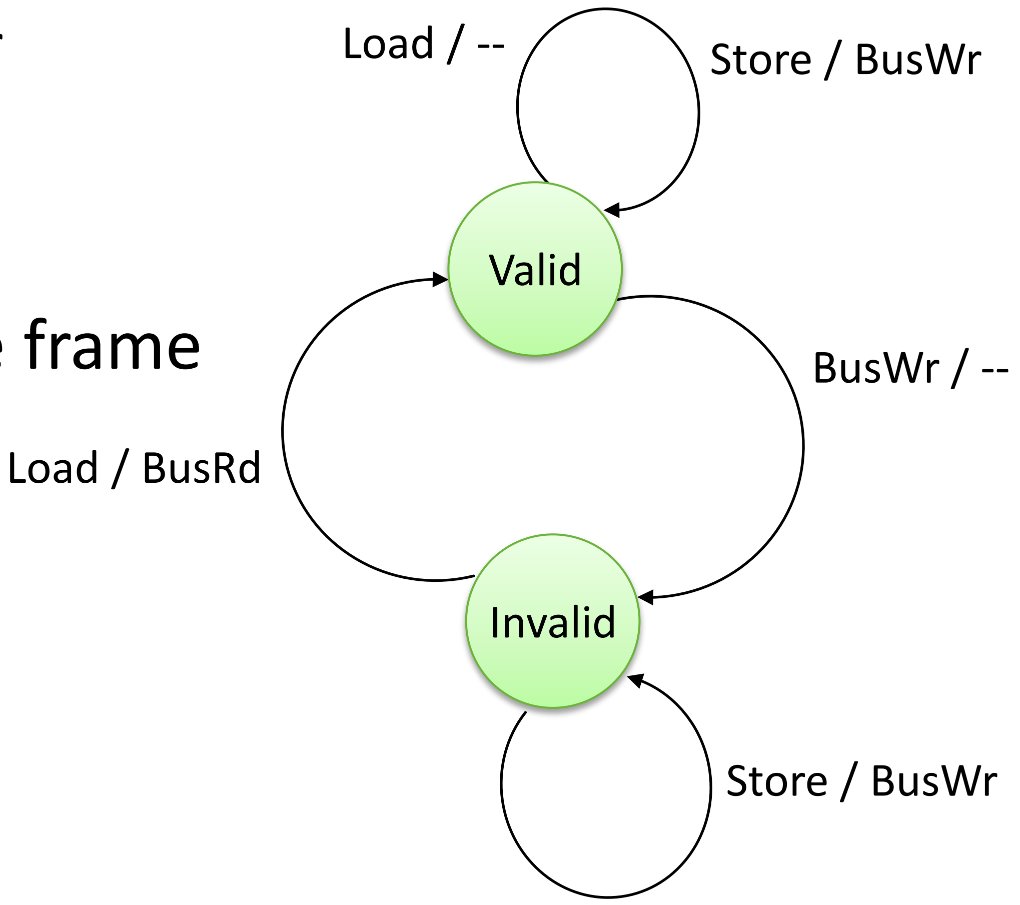
Coherence with Write-through Caches

- Allows multiple readers, but writes through to bus
  - Requires 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

- Processor Actions
  - Ld, St, BusRd, BusWr
- Bus Messages
  - BusRd, BusWr
- Track 1 bit per cache frame
  - Valid/Invalid



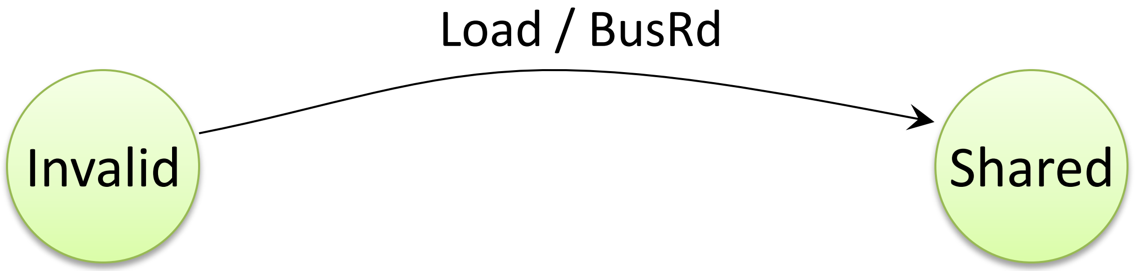
Supporting Write-Back Caches

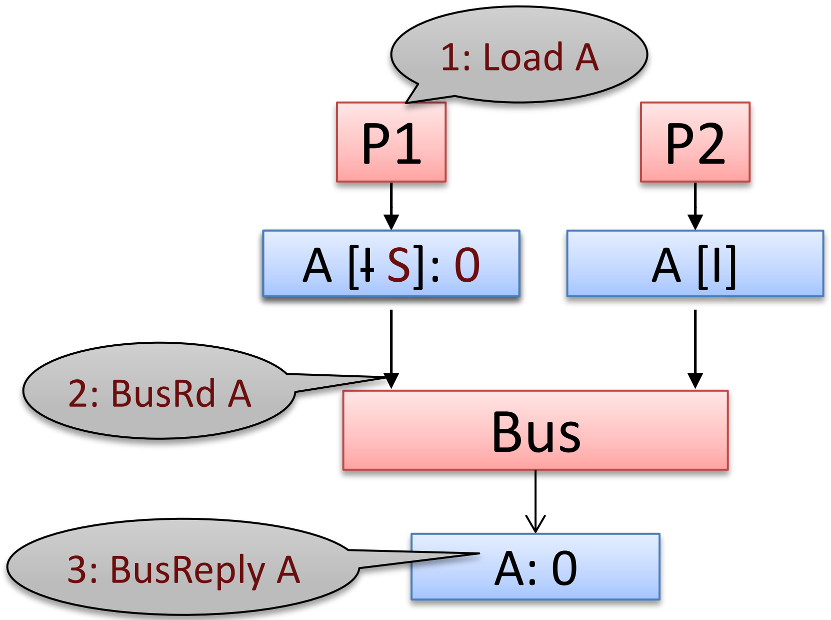
- Write-back caches are good
  - Drastically reduce bus write bandwidth
- Add notion of "*ownership*" to Valid-Invalid
  - When "*owner*" has only replica of a cache block
    - Update it freely
  - Multiple readers are ok
    - Not allowed to write without gaining ownership
  - On a read, system must check if there is an owner
    - If yes, take away ownership

Modified-Shared-Invalid (MSI) States

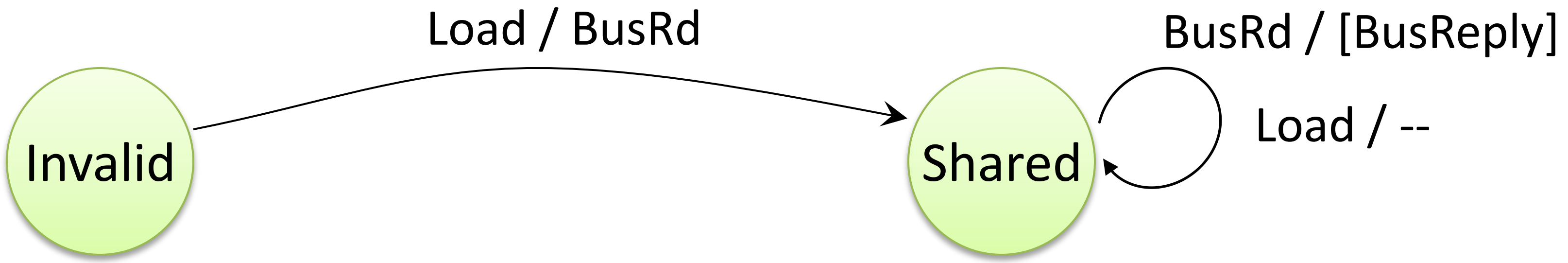
- Processor Actions
  - Load, Store, Evict
- Bus Messages
  - BusRd, BusRdX, BusInv, BusWB, BusReply
    (Here for simplicity, some messages can be combined)
- Track 3 states per cache frame
  - *Invalid*: cache does not have a copy
  - *Shared*: cache has a read-only copy; clean
    - Clean: memory (or later caches) is up to date
  - *Modified*: cache has the only valid copy; writable; dirty
    - Dirty: memory (or later caches) is out of date

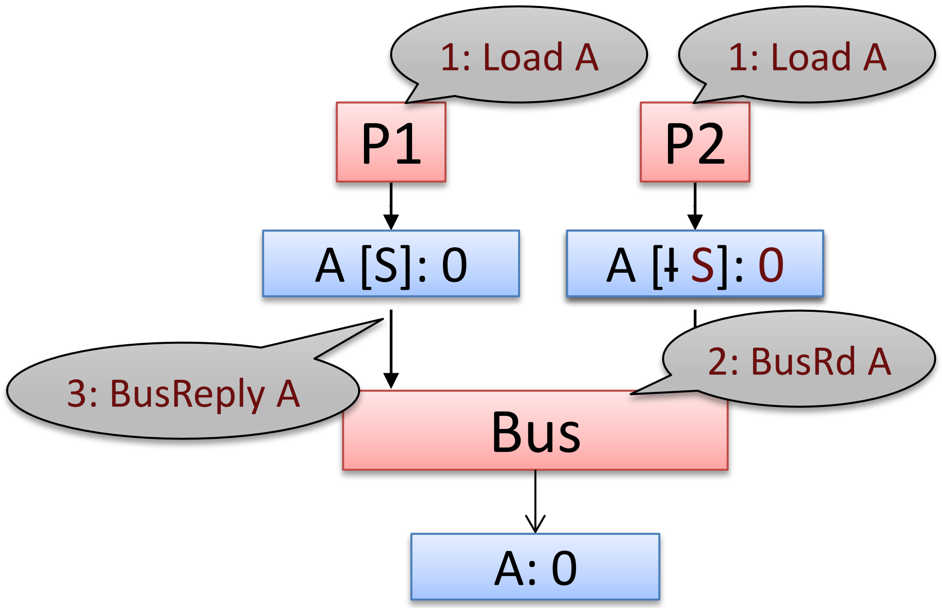
Simple MSI Protocol (1/9)



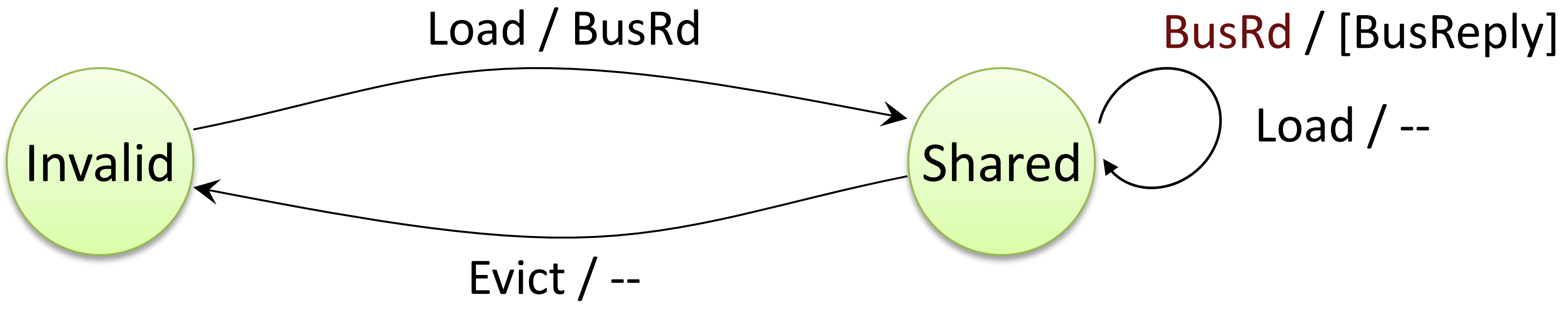


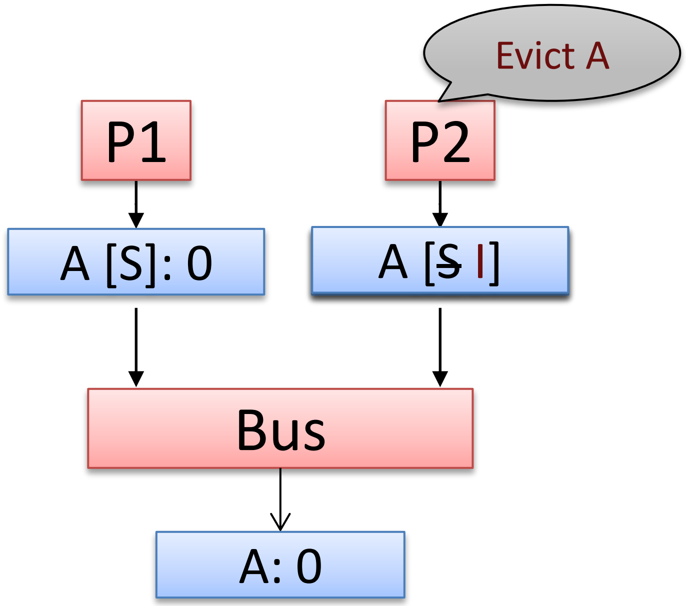
Simple MSI Protocol (2/9)



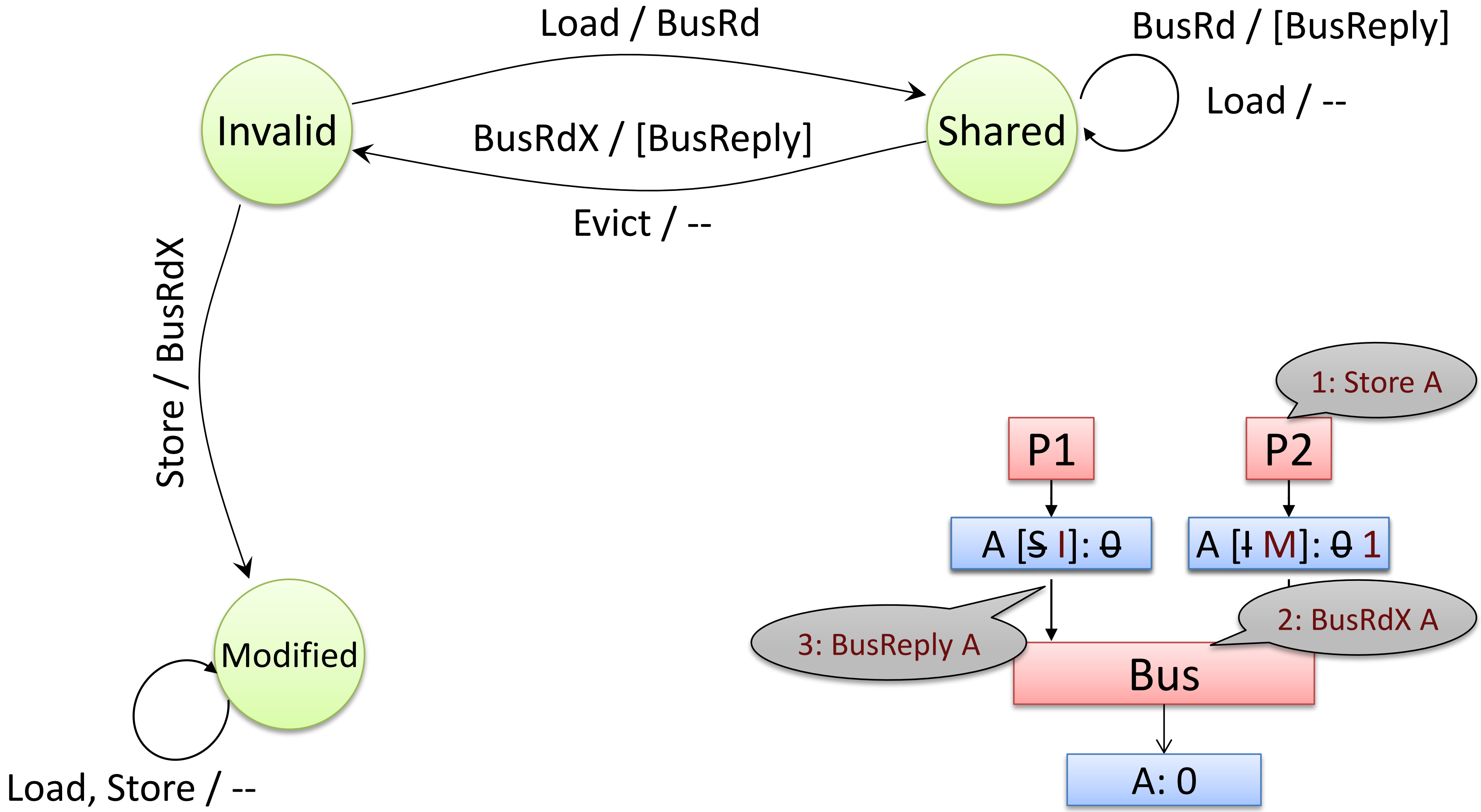


Simple MSI Protocol (3/9)

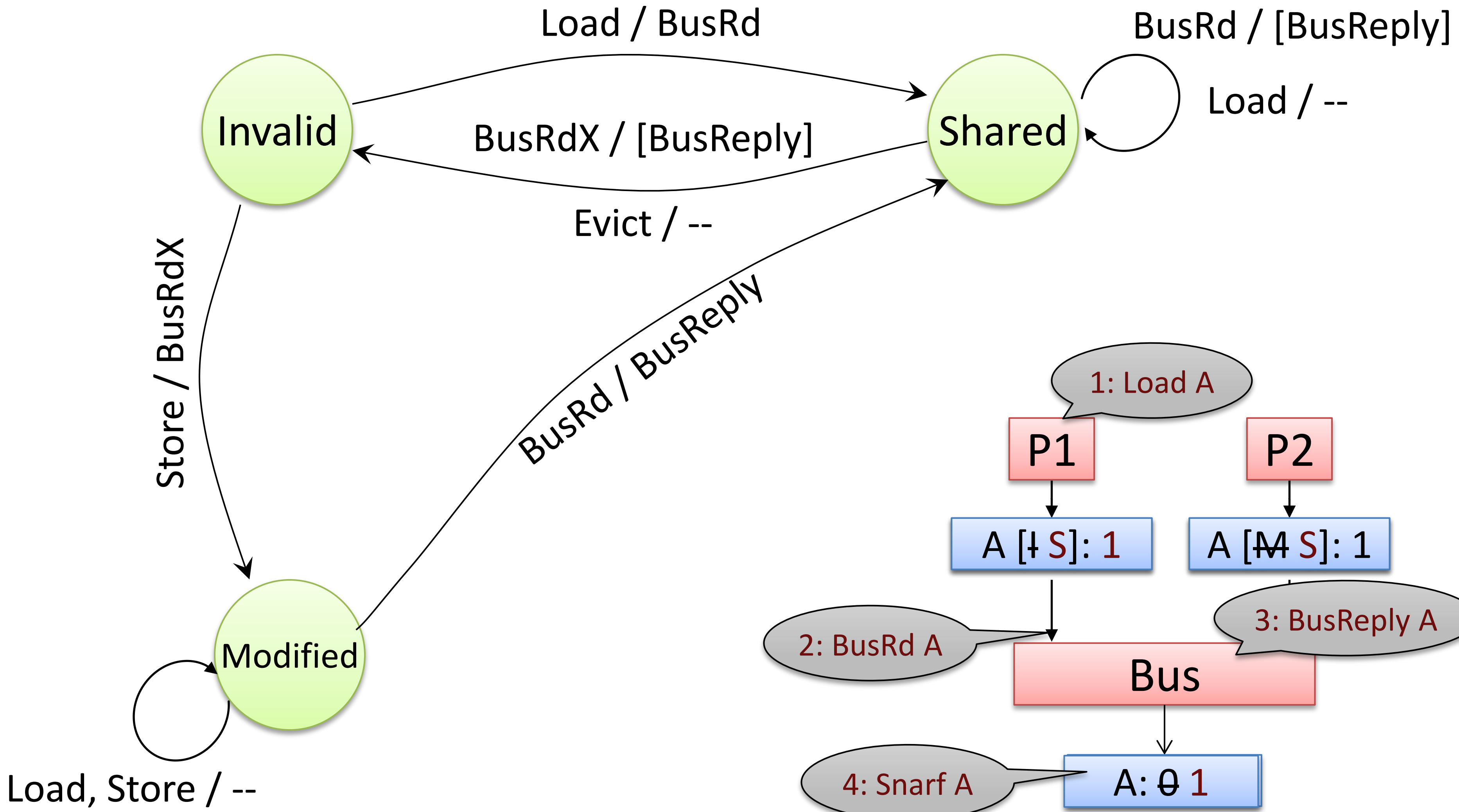




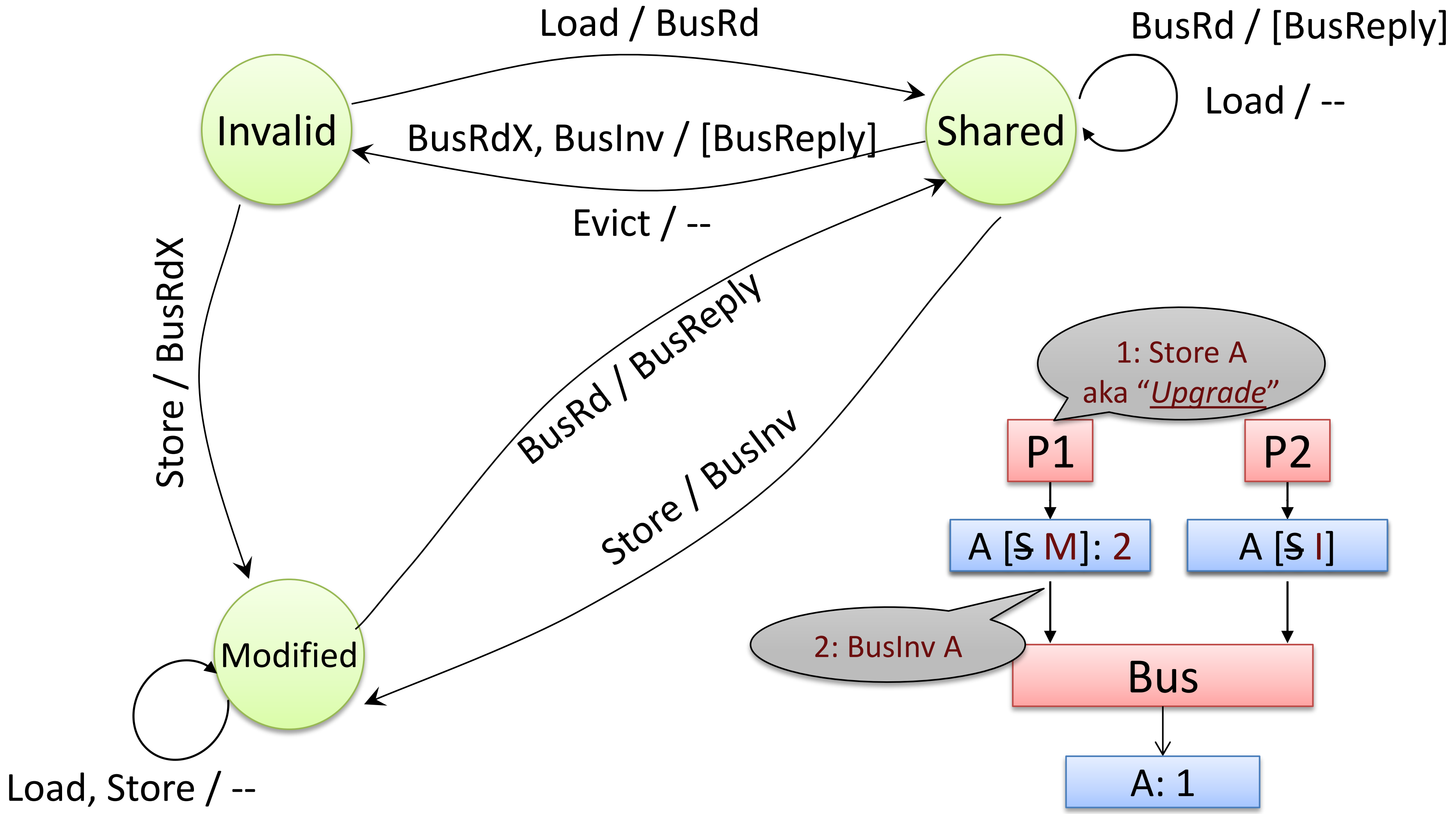
Simple MSI Protocol (4/9)



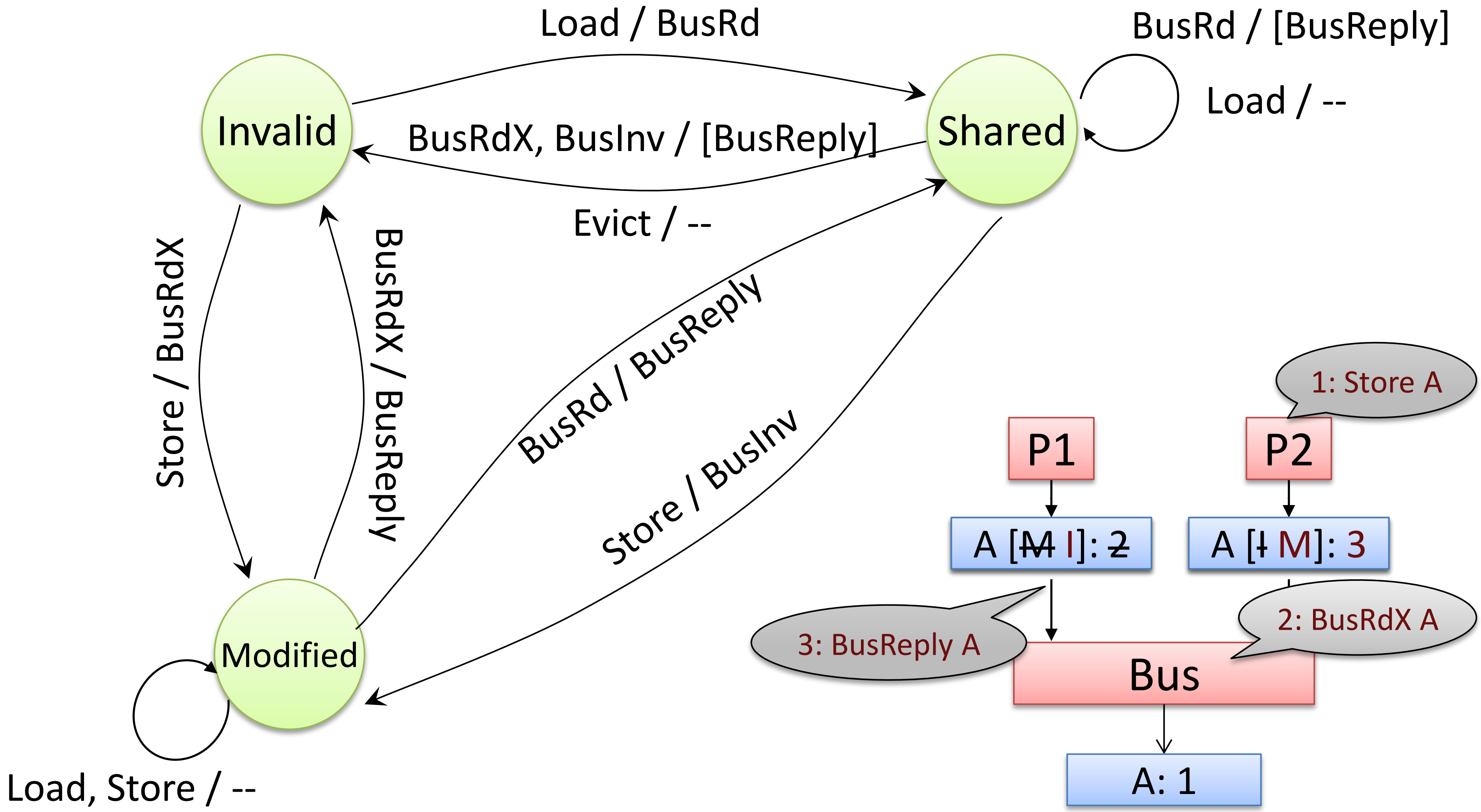
Simple MSI Protocol (5/9)



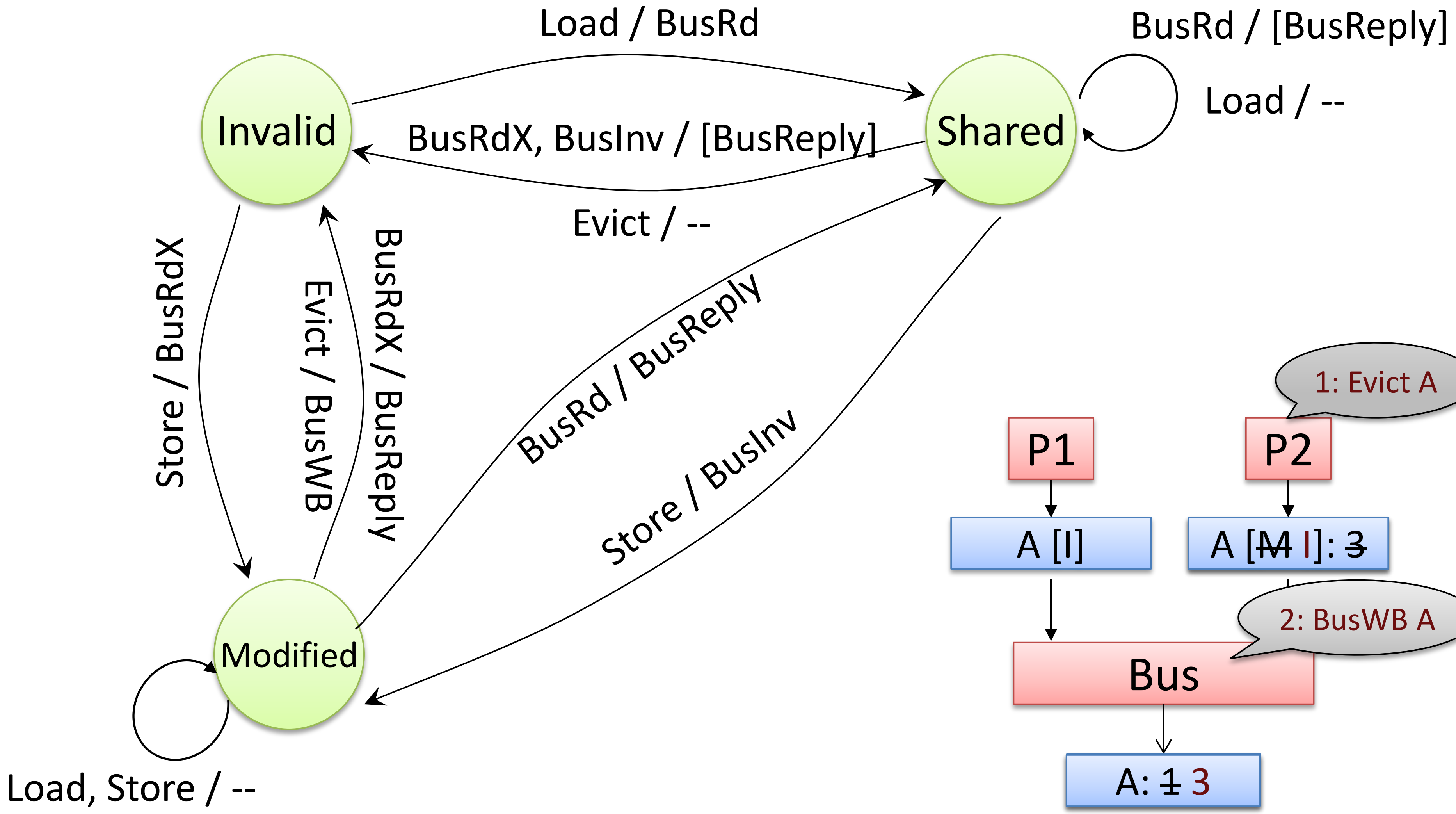
Simple MSI Protocol (6/9)



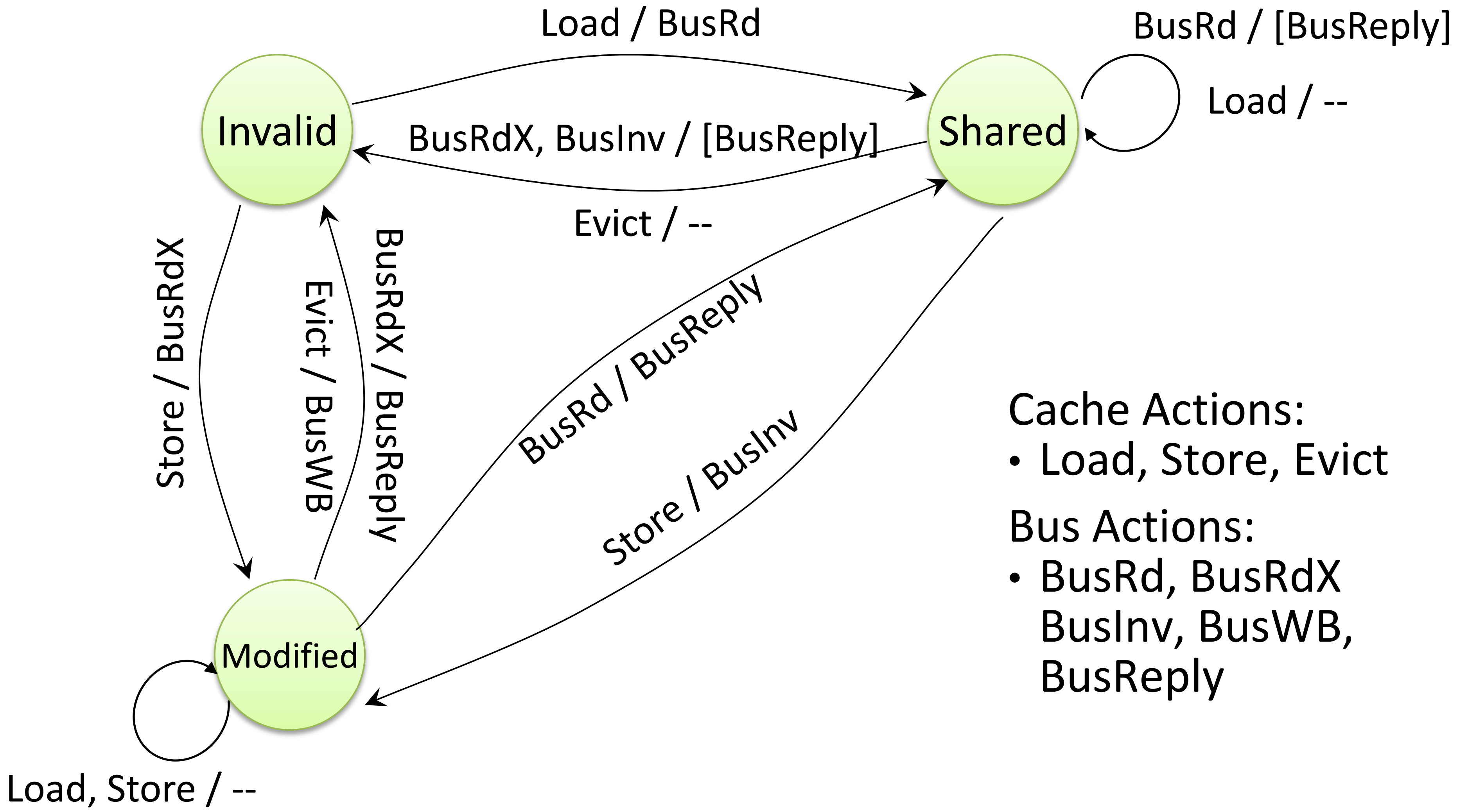
Simple MSI Protocol (7/9)



Simple MSI Protocol (8/9)



Simple MSI Protocol (9/9)



Cache Actions:
- Load, Store, Evict

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

Usable coherence protocol

Scalable Cache Coherence

- Part I: bus bandwidth
  – Replace non-scalable bandwidth substrate (bus) …with scalable-bandwidth one (e.g., mesh)
- Part II: processor snooping bandwidth
  – Most snoops result in no action
  – Replace non-scalable broadcast protocol (spam everyone) …with scalable directory protocol (spam cores that care)

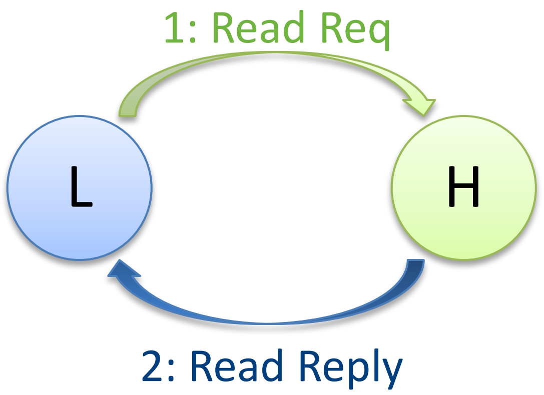
Requires a "*directory*" to keep track of "*sharers*"

Directory 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

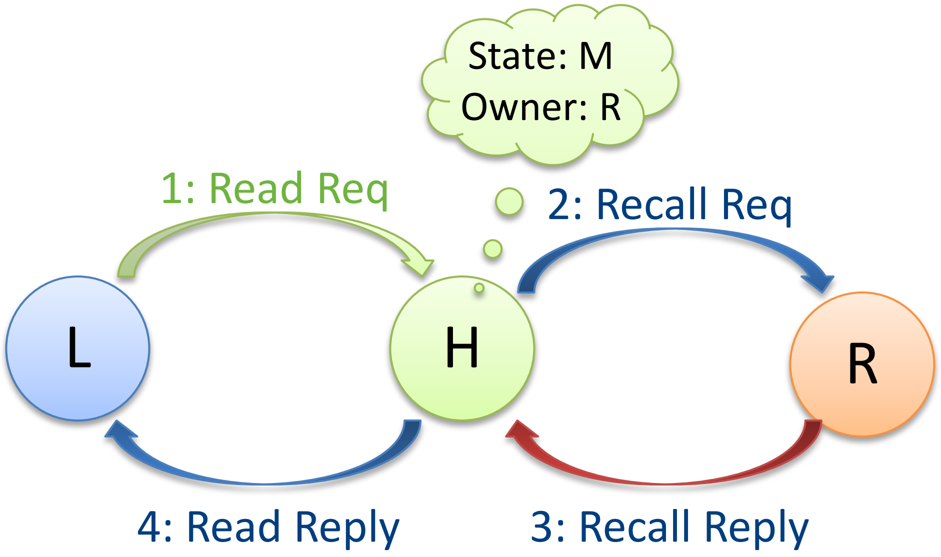
Read Transaction

- L has a cache miss on a load instruction



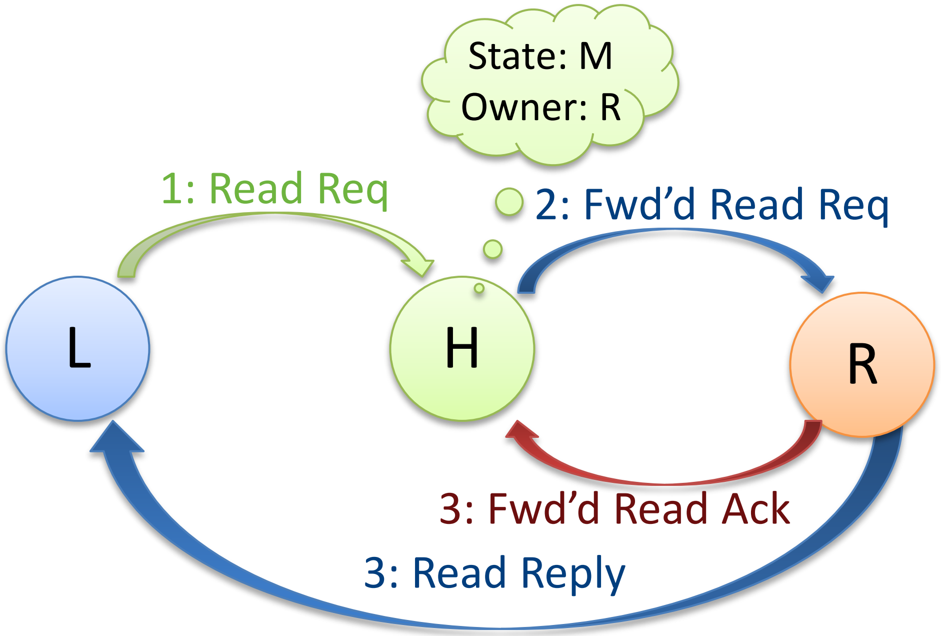
4-hop Read Transaction

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



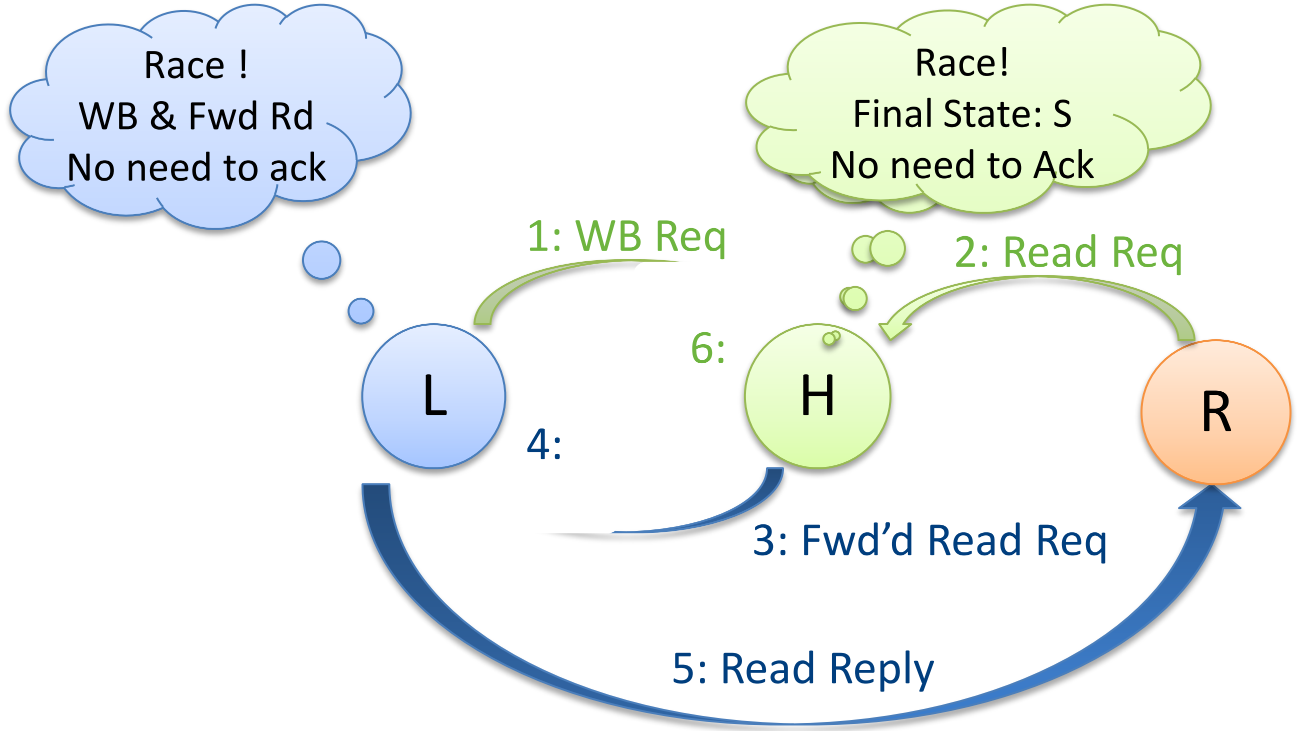
3-hop Read Transaction

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



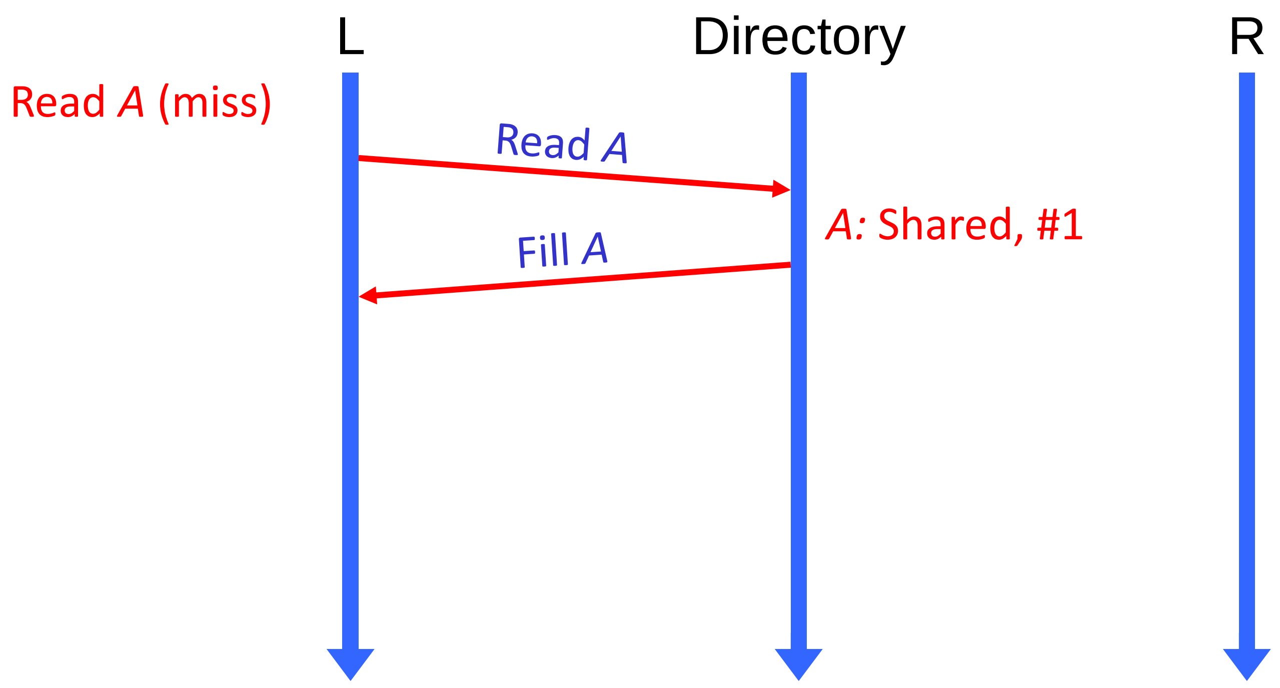
An Example Race: Writeback & Read

- L has dirty copy, wants to write back to H
- R concurrently sends a read to H



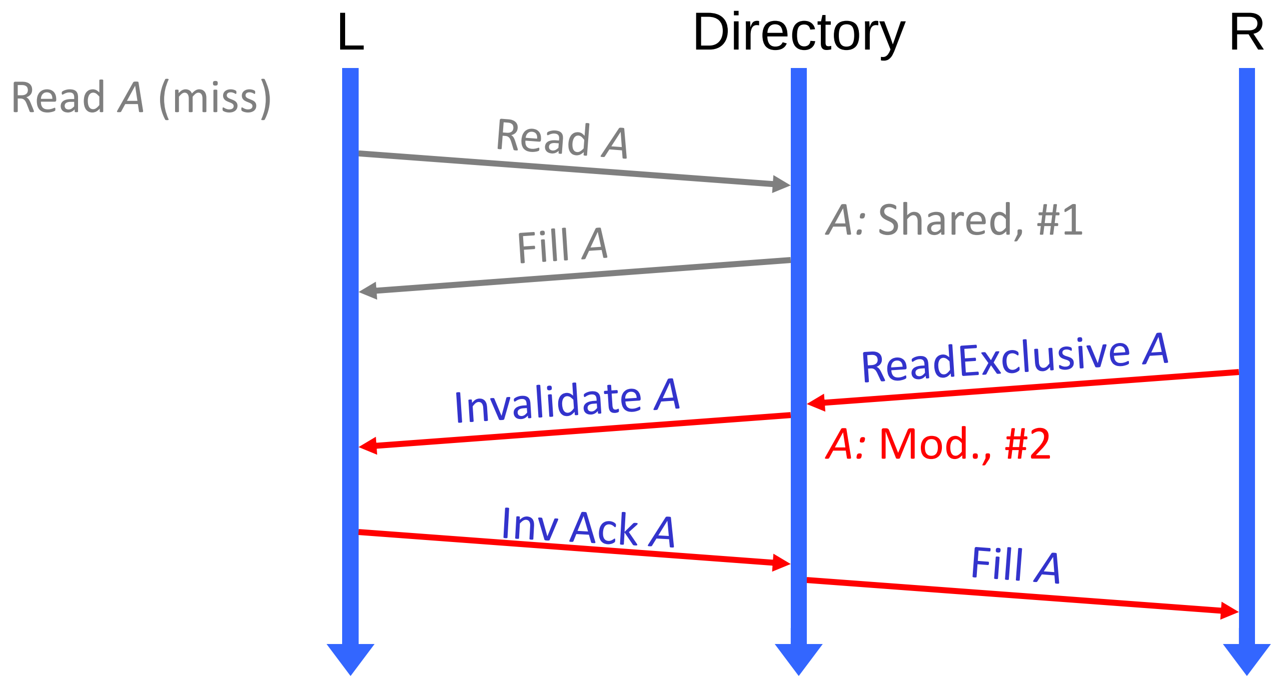
Races require complex *intermediate* states

Basic Operation: Read



Typical way to reason about directories

Basic Operation: Write



Coherence vs. Consistency

- Coherence concerns only one memory location
- Consistency concerns ordering for all locations
- A Memory System is Coherent if
  - Can serialize all operations to that location
    - Operations performed by any core appear in program order
  - Read returns value written by last store to that location
- A Memory System is Consistent if
  - It follows the rules of its *Memory Model*
    - Operations on memory locations appear in ***some*** defined order

Why Coherence != Consistency

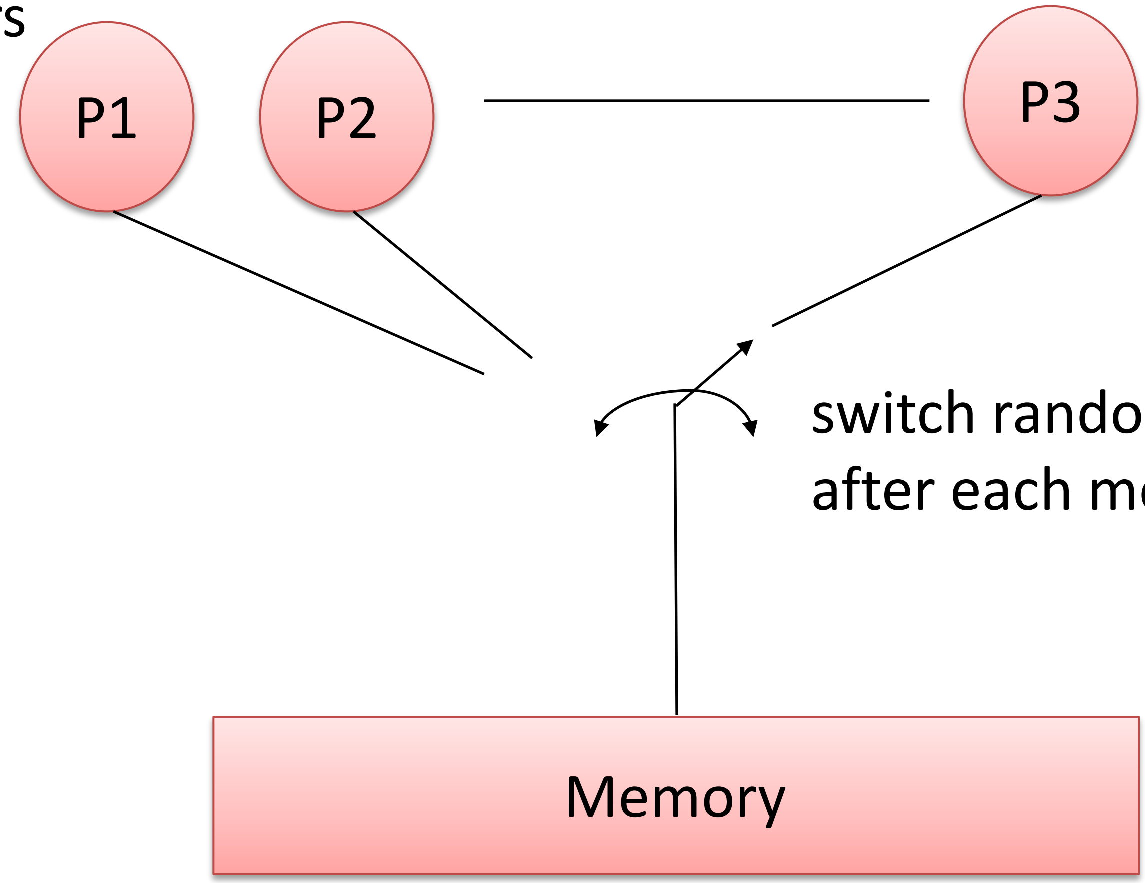
```
/* initial A = B = flag = 0 */
 P1                  P2
A = 1;              while (flag == 0); /* spin */
B = 1;              print A;
flag = 1;           print B;
```

- Intuition says we see “1” printed twice (A,B)
- Coherence doesn’t say anything
  - Difference memory locations
- Uniprocessor ordering (LSQ) won’t help

Consistency defines what is “correct” behavior

Sequential Consistency (SC)

processors issue memory ops in program order



Defines Single Sequential Order Among All Ops.

Sufficient Conditions for SC

“A multiprocessor 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 this sequence in the order specified by its program.”

-Lamport, 1979

- Every proc. issues memory ops in program order
- Memory ops happen (start and end) atomically
  - On Store, ***wait to commit*** before issuing next memory op
  - On Load, ***wait to write back*** before issuing next op

Easy to reason about, very slow (without ugly tricks)

Mutual Exclusion Example

- Mutually exclusive access to a critical region
  - Works as advertised under Sequential Consistency
  - Fails if P1 and P2 see different Load/Store order
    - OoO allows P1 to read B before writing (committing) A

P1

```
lockA: A = 1;
if (B != 0)
   { A = 0; goto lockA; }
/* critical section*/
A = 0;
```

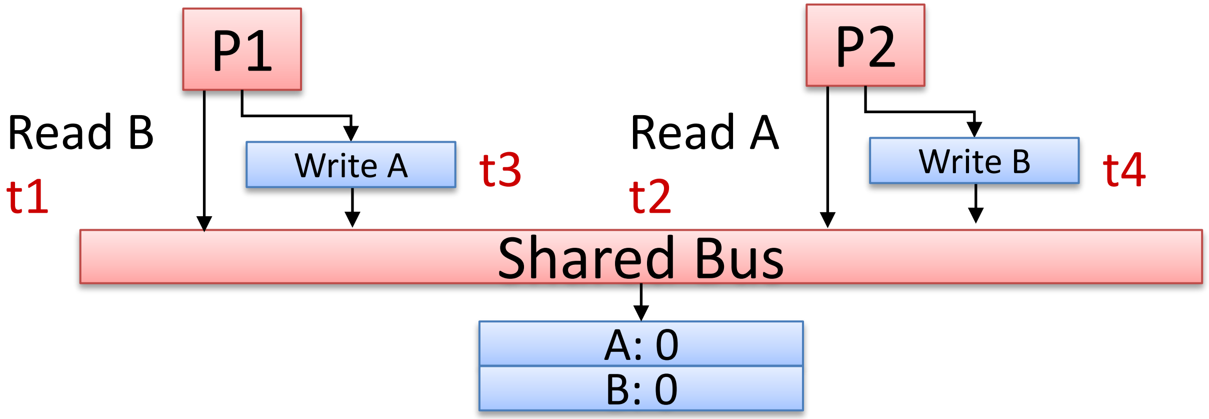
P2

```
lockB: B=1;
if (A != 0)
   { B = 0; goto lockB; }
/* critical section*/
B = 0;
```

Problems with SC Memory Model

- Difficult to implement efficiently in hardware
  - Straight-forward implementations:
    - No concurrency among memory access
    - Strict ordering of memory accesses at each node
    - Essentially precludes out-of-order CPUs
- Unnecessarily restrictive
  - Most parallel programs won’t notice out-of-order accesses
- Conflicts with latency hiding techniques

Mutex Example w/ Store Buffer



```
P1
lockA: A = 1;
if (B != 0)
   { A = 0; goto lockA; }
/* critical section*/
A = 0;
```

```
P2
lockB: B=1;
if (A != 0)
   { B = 0; goto lockB; }
/* critical section*/
B = 0;
```

Does not work

Relaxed Consistency Models

$X \rightarrow Y$
X must complete before Y

- Sequential Consistency (SC):
  - R → W, R → R, W → R, W → W
- *Total Store Ordering* (TSO) relaxes W → R
  - R → W, R → R, W → W
- *Partial Store Ordering* relaxes W → W (coalescing WB)
  - 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

Atomic Operations & Synchronization

- Atomic operations perform multiple actions together
  - Each of these can implement the others

- *Compare-and-Swap* (CAS)
  - If memory value matches, overwrite with new value
- *Test-and-Set*
  - Write new value and return old value
- *Fetch-and-Increment*
  - Increment value in memory and return the old value
- *Load-Linked/Store-Conditional* (LL/SC)
  - ***Two*** operations, but Store succeeds iff value unchanged