Multicores and cache coherence

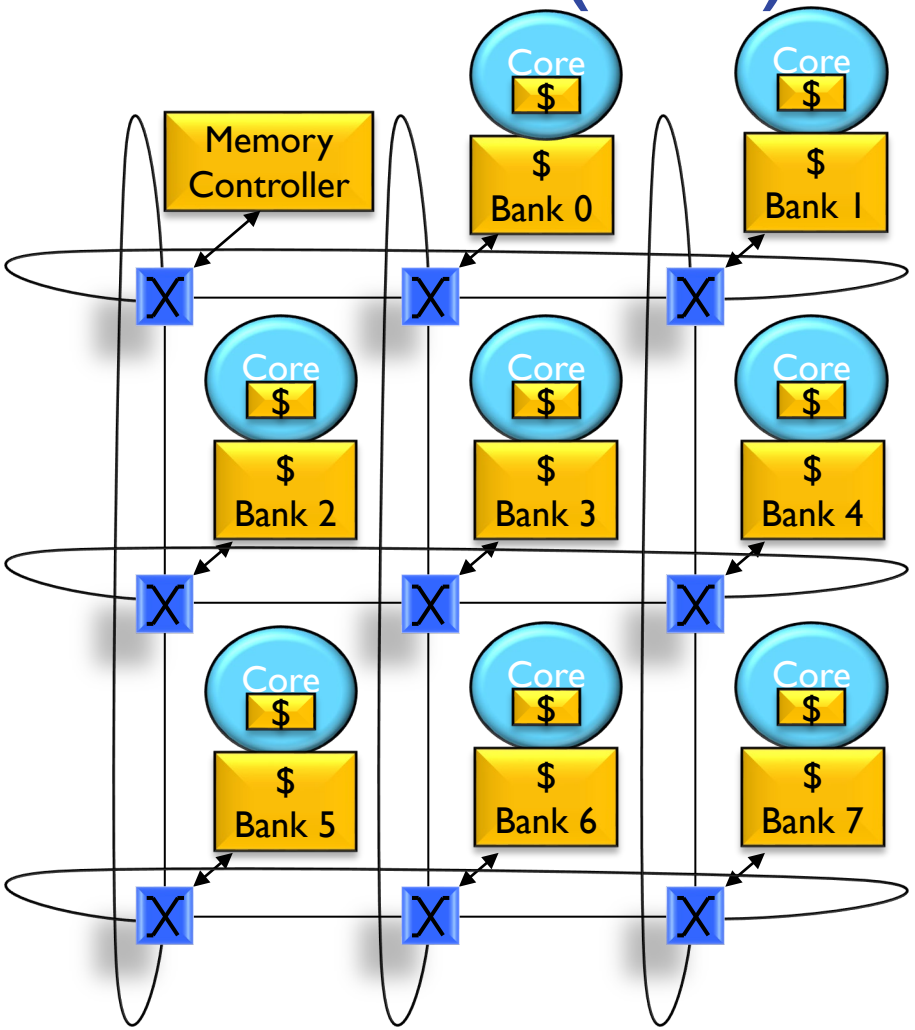
Acknowledgements

- Some of the slides in the deck were provided by Profs. Luis Ceze (Washington), Nima Horanmand (Stony Brook), Mark Hill, David Wood, Karu Sankaralingam (Wisconsin), Abhishek Bhattacharjee(Yale).

- Development of this course is partially supported by funding from Western Digital corporations.

On-chip Interconnects (5/5)

- Possible topologies
  - Bus
  - Crossbar
  - Ring
  - Mesh
  - *Torus*
- 5 ports per switch
- Can be “folded” to avoid long links



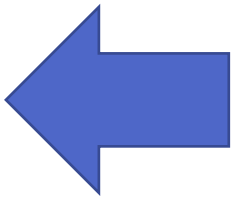
How to connect multiple cores?

- Physical connection:
  - How multiple cores and caches are connected on a CMP?
  - Different on-chip interconnection topologies
    - Also called Network-on-chip or NOC
- Logical connection/Logical view:
  - What is the software view?
  - Fundamentally how the memory is shared?

How to connect multiple cores?

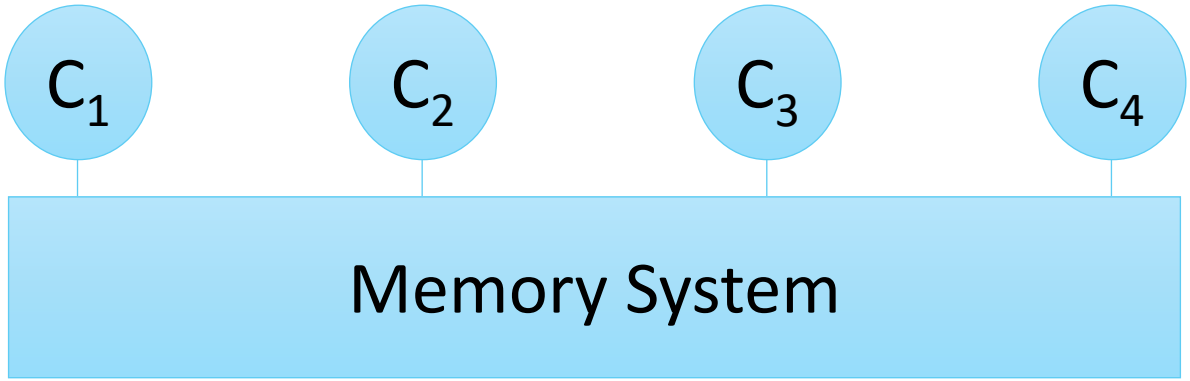
- Physical connection:
  - How multiple cores and caches are connected on a CMP?
  - Different on-chip interconnection topologies
    - Also called Network-on-chip or NOC

- Logical connection/Logical view:
  - What is the software view?
  - Fundamentally how the memory is shared?



Logical View

- Multiple threads use **shared memory** (physical address space) to communicate
  - "Threads" in software
- Communication **implicit via loads and stores**
  - Opposite of explicit **message-passing multiprocessors**



Classes of Multiprocessors

- **How do the multiple cores/processors communicate?**

Classes of Multiprocessors

- **How do the multiple cores/processors communicate?**
- **Shared Memory Multiprocessors (e.g., multicores)**
  - Single OS manages all the processors/cores
  - OS sees multiple CPUs
    - Runs one process (or thread) on each CPU
  - Code running on different cores communicate by reading/writing a shared physical address space
  - Most common form of multiprocessors today (e.g., multicores)

Classes of Multiprocessors

- **How do the multiple cores/processors communicate?**
- **Shared Memory Multiprocessors (e.g., multicores)**
  - Single OS manages all the processors/cores
  - OS sees multiple CPUs
    - Runs one process (or thread) on each CPU
  - Code running on different cores communicate by reading/writing a shared physical address space
  - Most common form of multiprocessors today (e.g., multicores)
- **Message-Passing Multiprocessors (Multi-node clusters)**
  - Composed of multiple nodes (computers)
  - Nodes may not have access to each other's memory (no shared memory)
  - Nodes communicate by passing explicit messages
  - Supercomputers are the most common examples

Shared -Memory Multiprocessors (e.g., Multi -core chips)

Why Shared Memory?

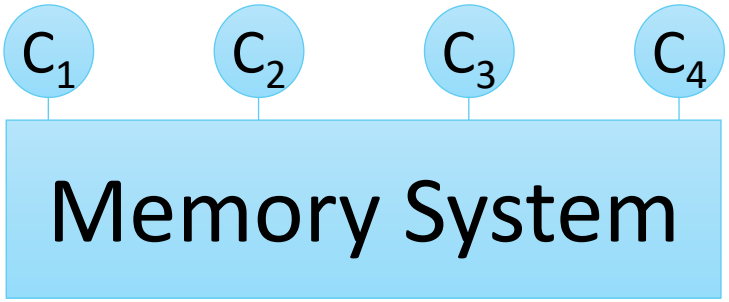
- Pros
  - + Programmers **don't** need to learn about **explicit** communications
    - Because communication is **implicit** (through memory)
  - + Applications similar to the case of multitasking uniprocessor
    - Programmers already know about synchronization
  - + OS needs only evolutionary extensions

Why Shared Memory?

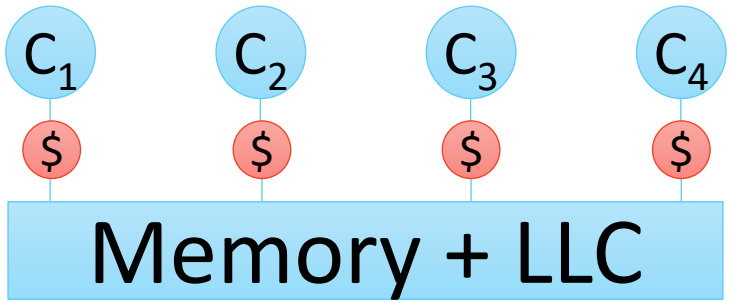
- Pros
  - + Programmers **don't** need to learn about **explicit** communications
    - Because communication is **implicit** (through memory)
  - + Applications similar to the case of multitasking uniprocessor
    - Programmers already know about synchronization
  - + OS needs only evolutionary extensions
- Cons
  - Communication is hard to optimize
    - Because it is implicit
    - Not easy to get good performance out of shared-memory programs
  - Synchronization is complex
    - Over-synchronization → bad performance
    - Under-synchronization → incorrect programs
    - **Very** difficult to debug
  - Hard to scale up to very large number of cores/processors

Cache Coherence problem

- Multiple copies of each cache block
  - One in main memory
  - Up to one in each cache
- Multiple copies can get inconsistent when writes happen
  - Should make sure all processors have a consistent view of memory

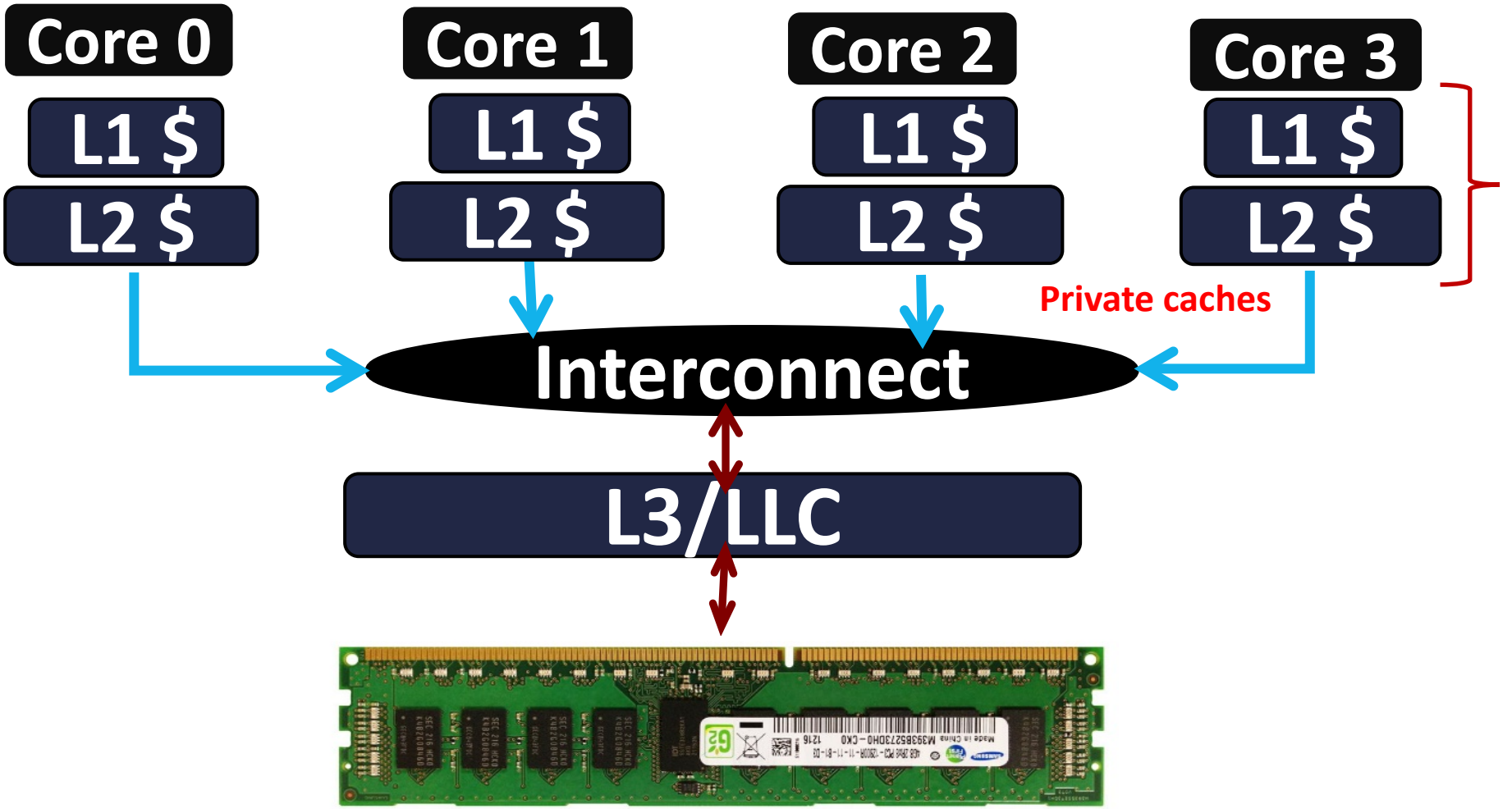


Logical View



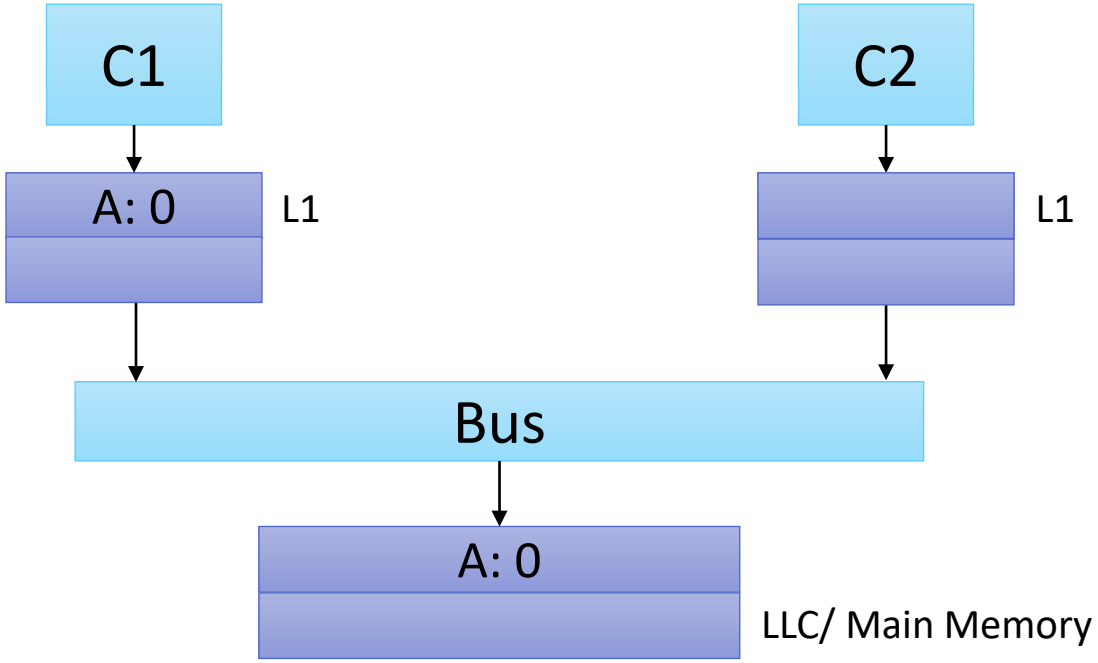
More Realistic View

Recap: Typical cache hierarchy



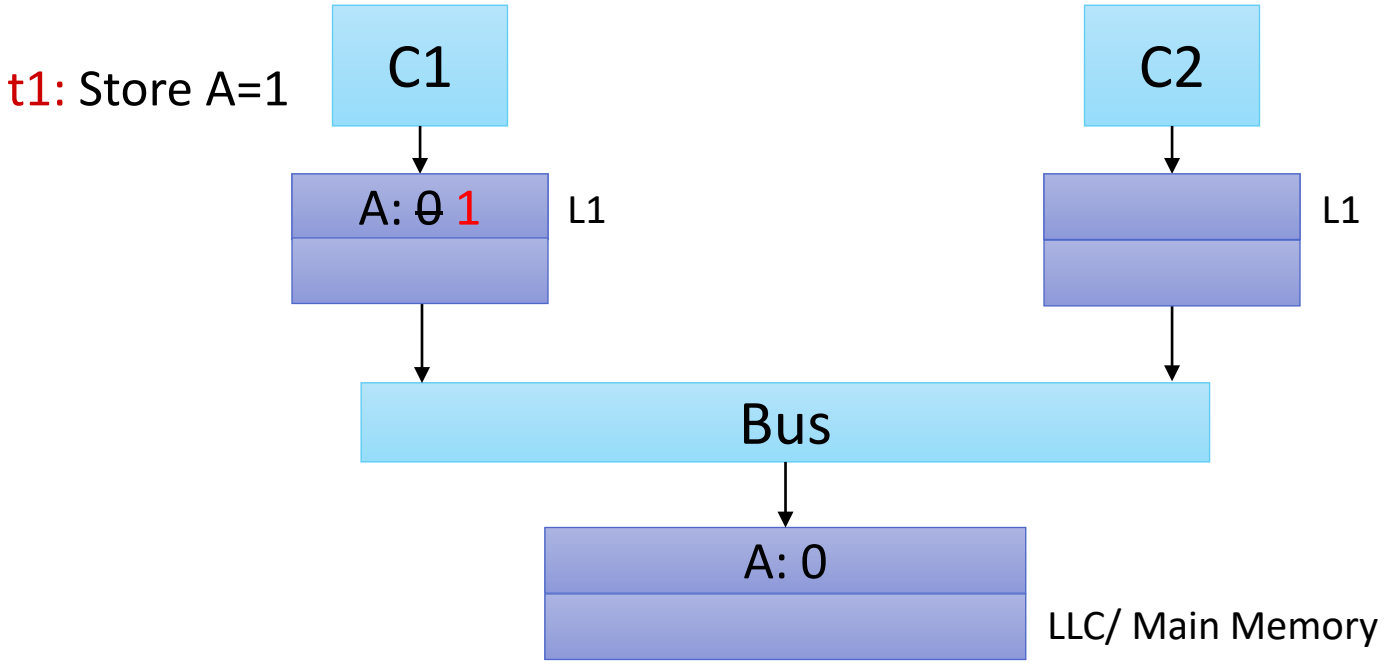
Cache Coherence problem

- Variable A initially has value 0
- C1 stores value 1 into A
- C2 loads A from memory and sees old value 0



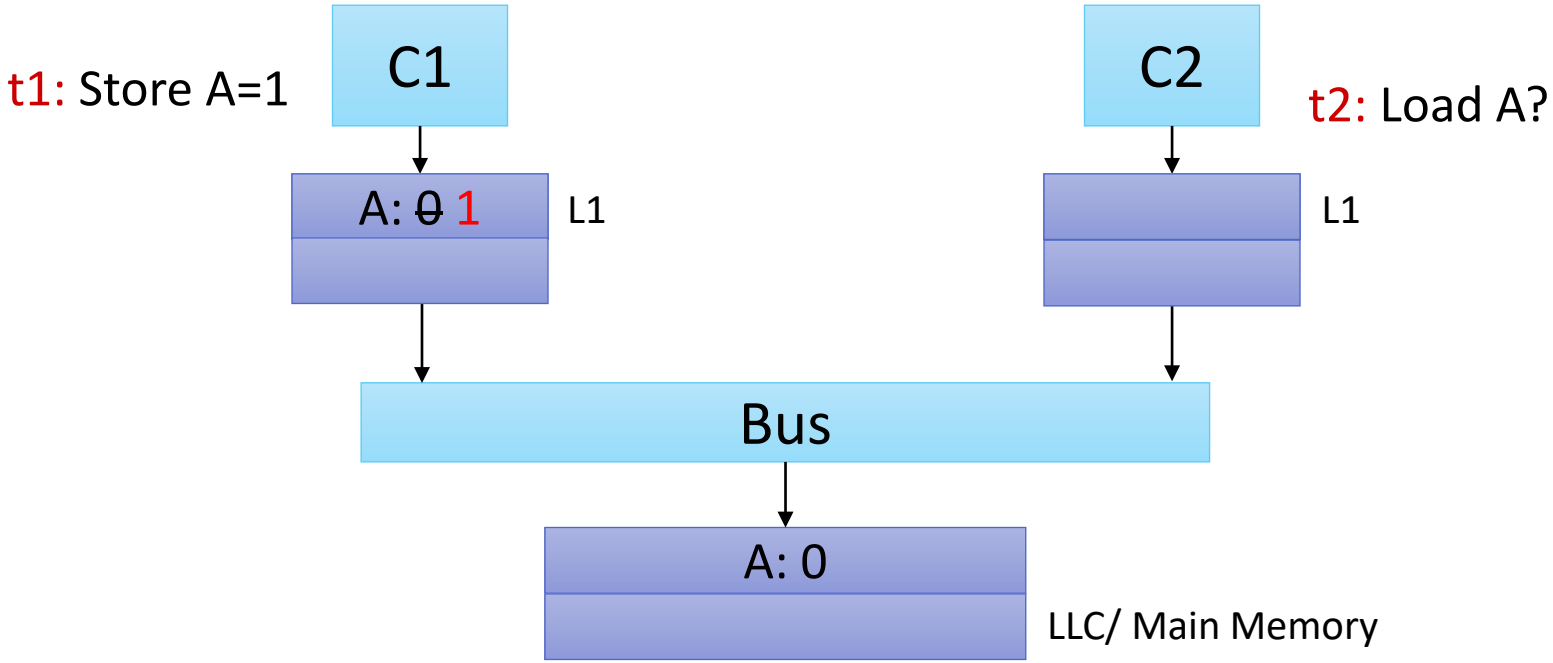
Cache Coherence problem

- Variable A initially has value 0
- C1 stores value 1 into A
- C2 loads A from memory and sees old value 0



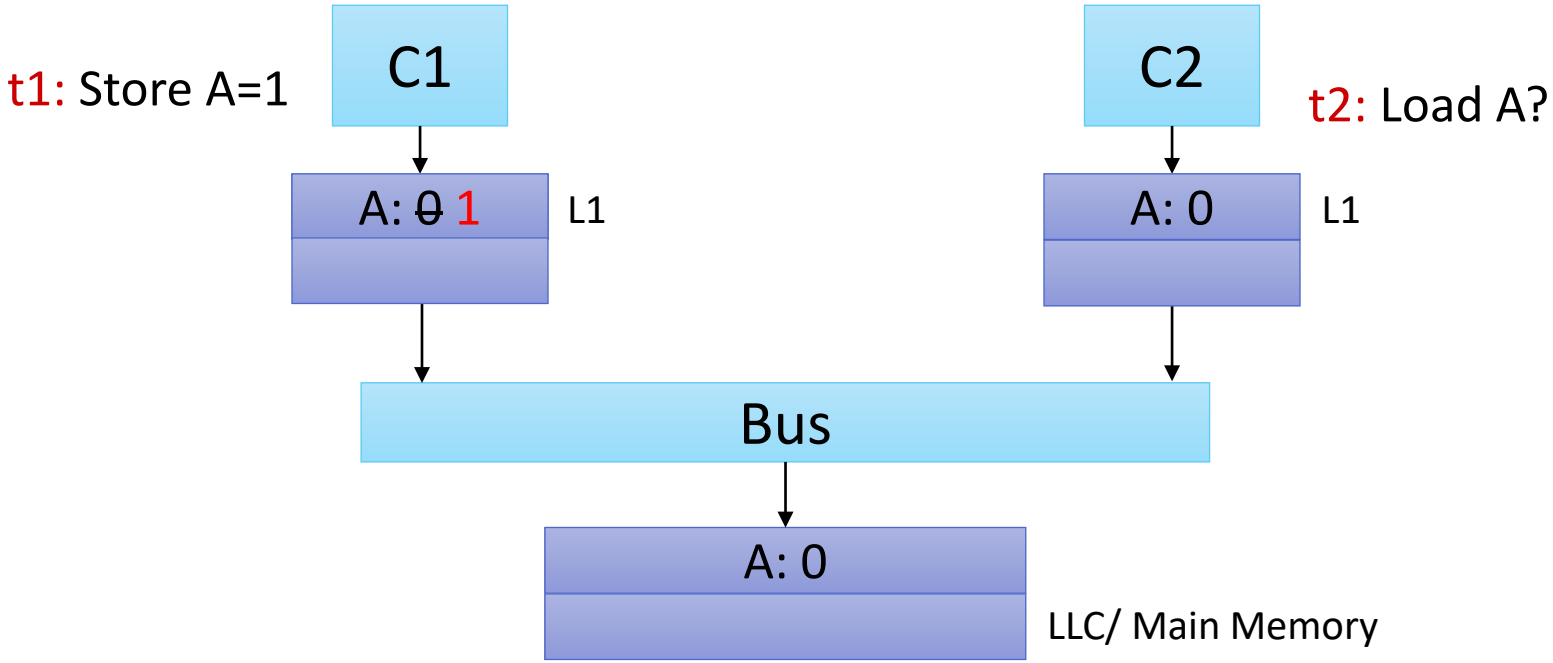
Cache Coherence problem

- Variable A initially has value 0
- C1 stores value 1 into A
- C2 loads A from memory and sees old value 0



Cache Coherence problem

- Variable A initially has value 0
- C1 stores value 1 into A
- C2 loads A from memory and sees old value 0



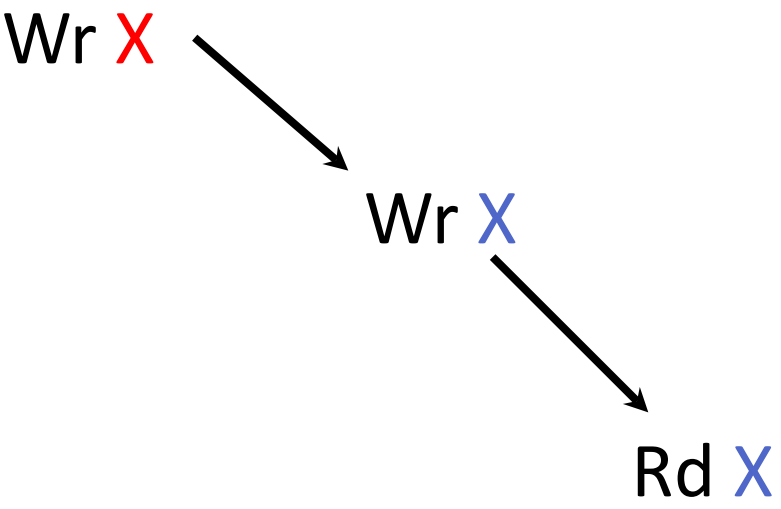
Formal Definition of Coherence

- A memory system is *coherent* if the results of any execution of a program are such that for each location, it **is possible to construct a hypothetical serial order** of all operations ***to the location*** that is consistent with the results of the execution and in which:
  - operations issued by any particular process occur in the order issued by that thread, and
  - the value returned by a read is the value written by the **last write** to that location in the serial order

Formal Definition of Coherence

- A memory system is *coherent* if the results of any execution of a program are such that for each location, it **is possible to construct a hypothetical serial order** of all operations ***to the location*** that is consistent with the results of the execution and in which:
  - operations issued by any particular process occur in the order issued by that thread, and
  - the value returned by a read is the value written by the **last write** to that location in the serial order
- Two necessary conditions:
  - *Write propagation*: value written must become visible to others **eventually**
  - *Write serialization*: writes to location seen in same order by all
    - if I see w1 after w2, you should not see w2 before w1

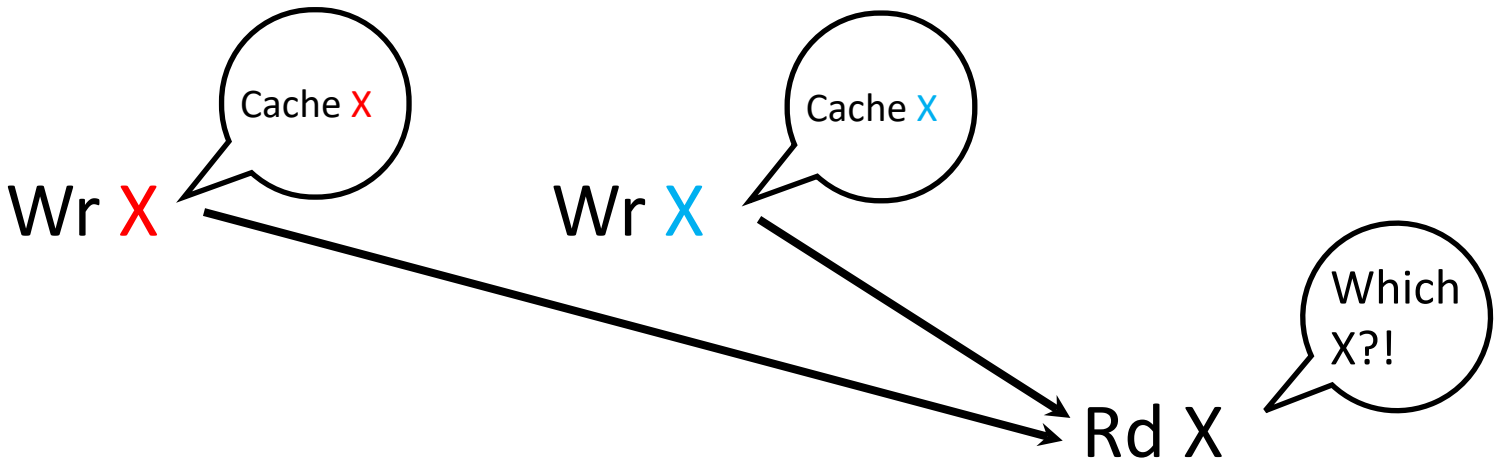
Coherence



"Reading X gets the value of the last write to X"

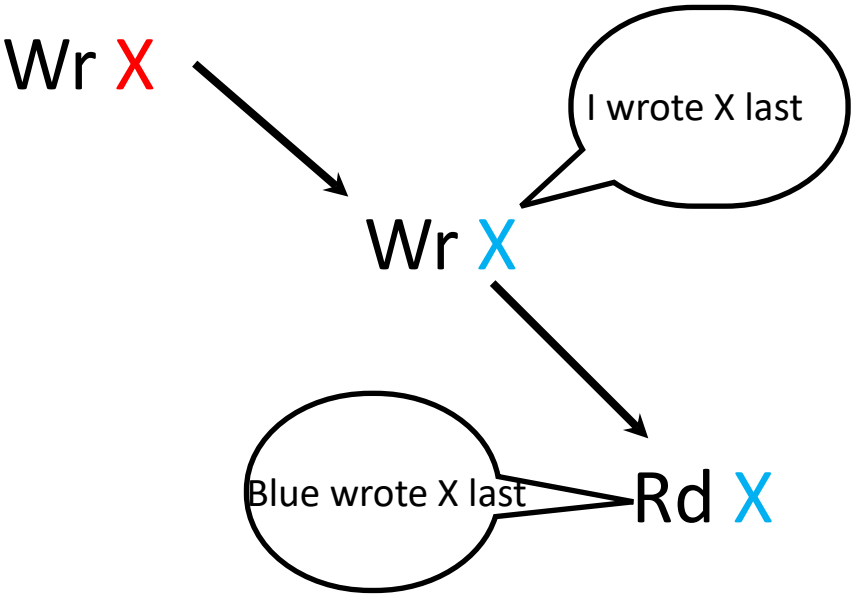
Without Coherence

(The coherence invariants prevent this from happening)



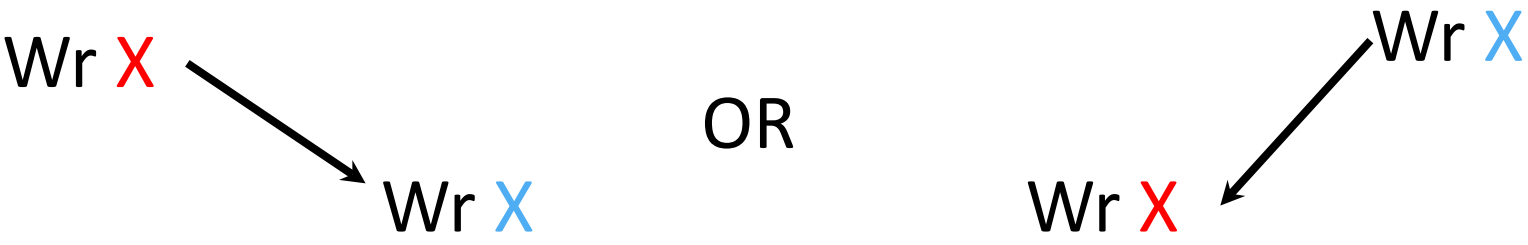
How to decide who wrote last?

Coherence

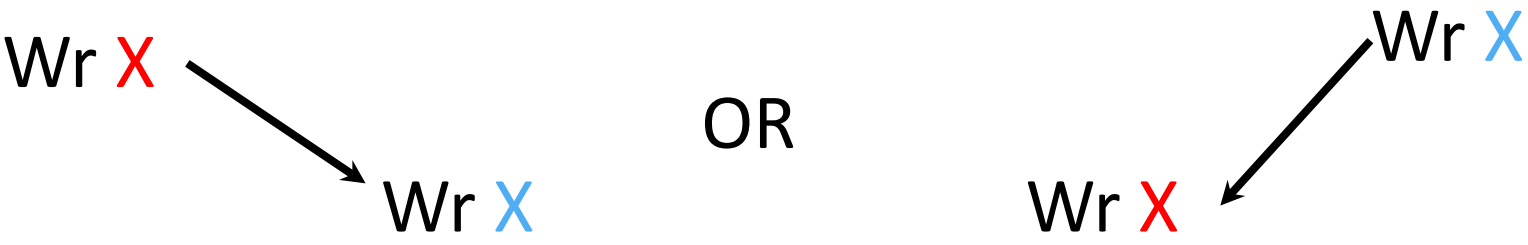


2) “Reading X gets the value of the last write to X”

Coherence enforces Order of Writes



Coherence enforces Order of Writes



Coherence defines a total order of ***writes*** to a **single memory** location.

How do we provide this guarantee?
Need to serialize somewhere…

Hardware Cache Coherence

- Hardware to ensure that everyone always sees the latest value for each physical memory location

Hardware Cache Coherence

- Hardware to ensure that everyone always sees the latest value for each physical memory location

Two important aspects

- ***Update*** vs. ***Invalidate***: on a write
  - update other copies, or
  - invalidate other copies
  - Invalidation protocols are far more common (our focus)

Hardware Cache Coherence

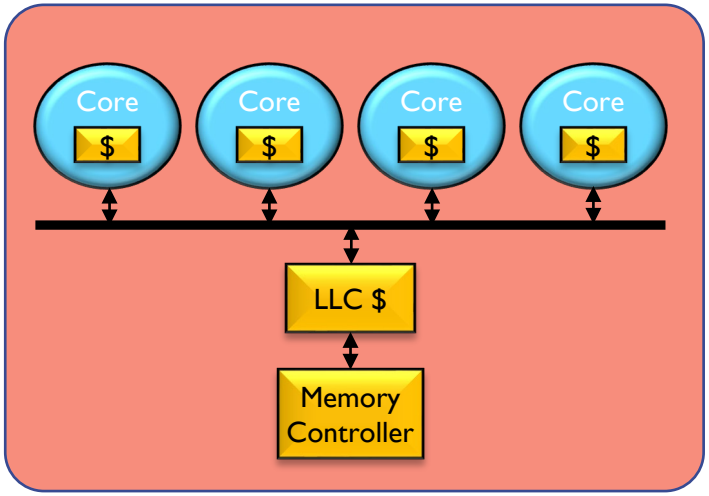
- Hardware to ensure that everyone always sees the latest value for each physical memory location

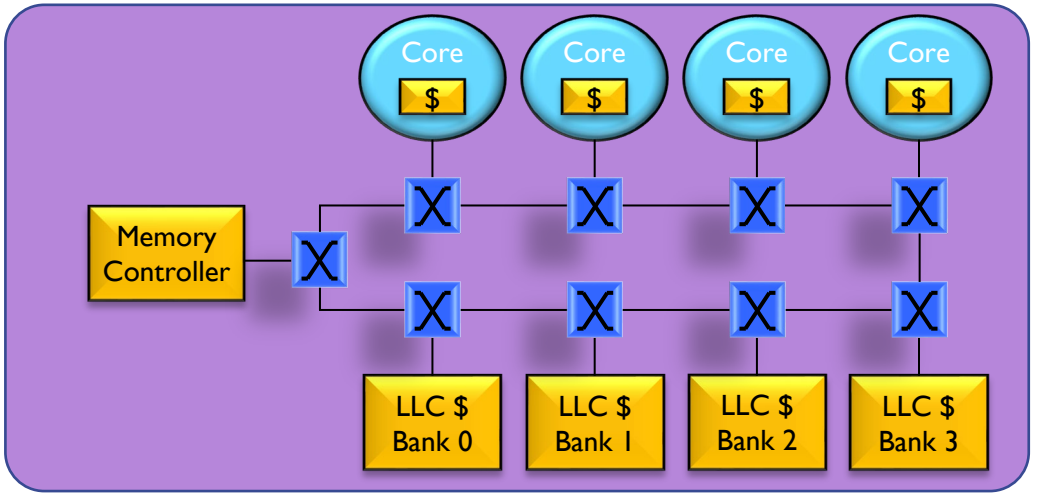
Two important aspects

- ***Update*** vs. ***Invalidate***: on a write
  - update other copies, or
  - invalidate other copies
  - Invalidation protocols are far more common (our focus)
- ***Broadcast*** vs. ***multicast***: send the update/invalidate…
  - to all other processors (aka ***snoopy coherence***) , or
  - only those that have a cached copy of the line (aka ***directory coherence*** or ***scalable coherence***)

Snoopy Protocols

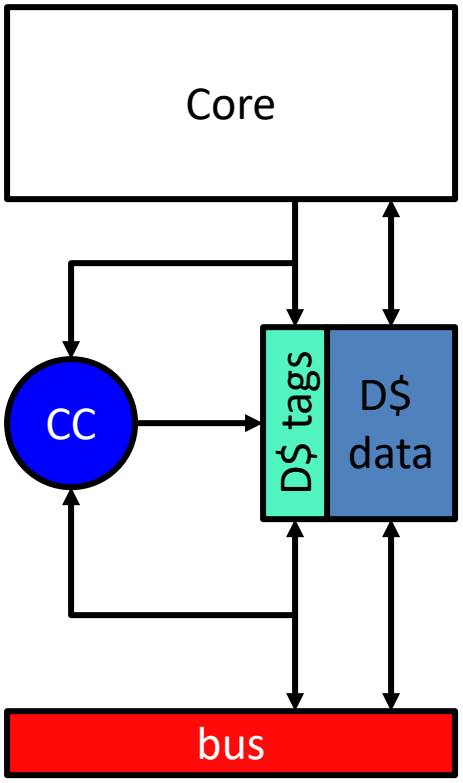
- Rely on broadcast-based interconnection network between caches
  - Typically Bus or Ring





- All caches must monitor (aka "*snoop*") all traffic
  - And keep track of cache line *states* based on the observed traffic

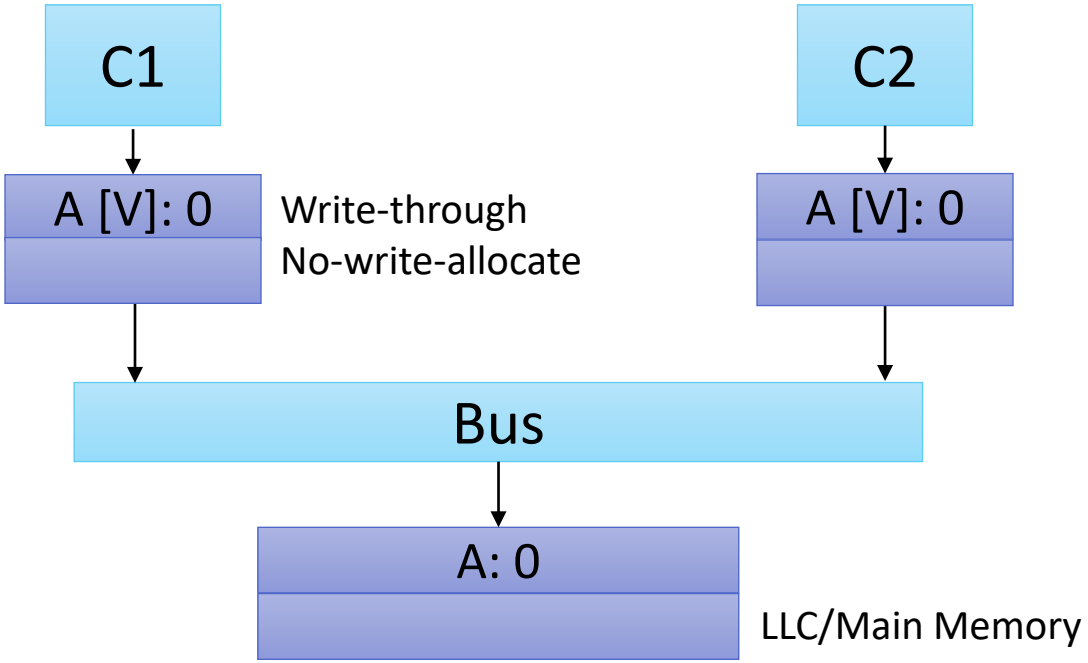
Snoopy Cache Coherence



- **Coherence controller with cache**:
  - Examines core's/processor's cache access type
  - Examines bus traffic (addresses and data)
  - Executes **coherence protocol**
    - What to do with local copy when you see different things happening on bus
    - Finite state automation
- Protocol is a distributed algorithm: cooperating finite state machines in coherence controllers
  - Set of states, state transition diagram, actions
- Granularity of coherence is a cache block
  - Coherence state maintained as extension to tag
- Assume bus messages totally ordered and atomic
- *What should the protocol (s) look like?*

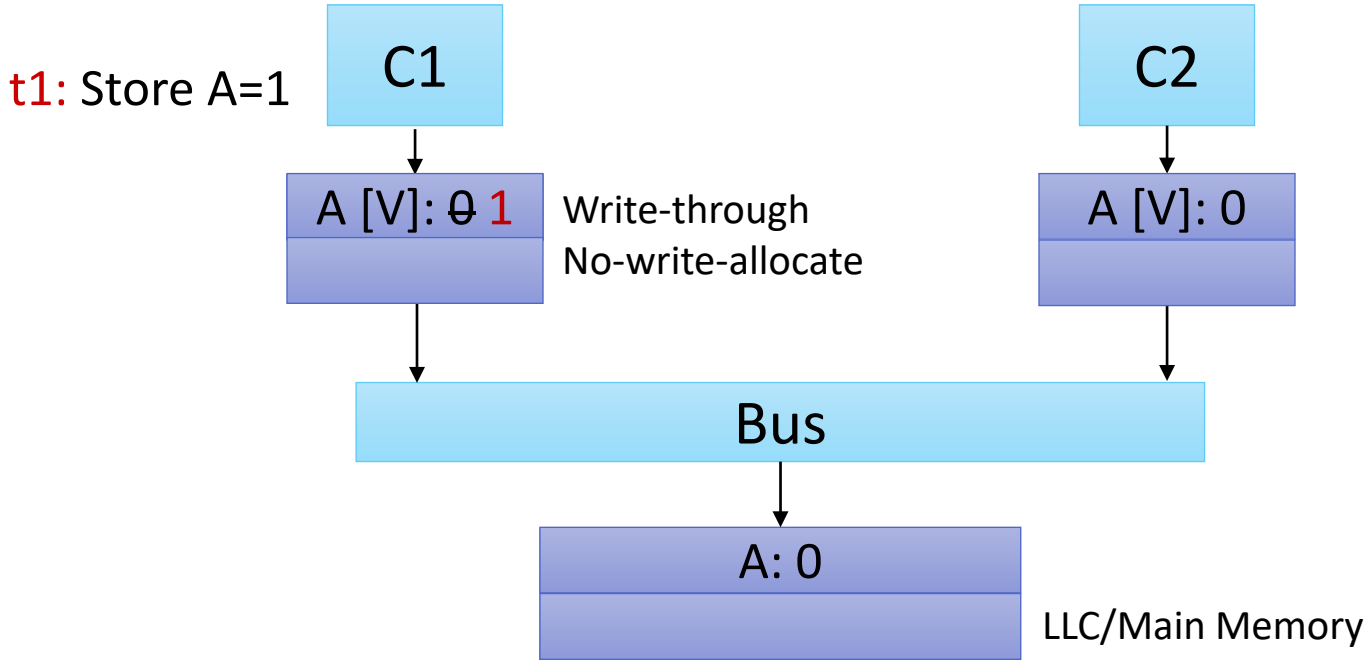
Example 1: Snoopy w/ Write -through

- Assume write-through, no-write-allocate cache
- Allows multiple readers, but writes through to bus
- Simple state machine for each cache frame



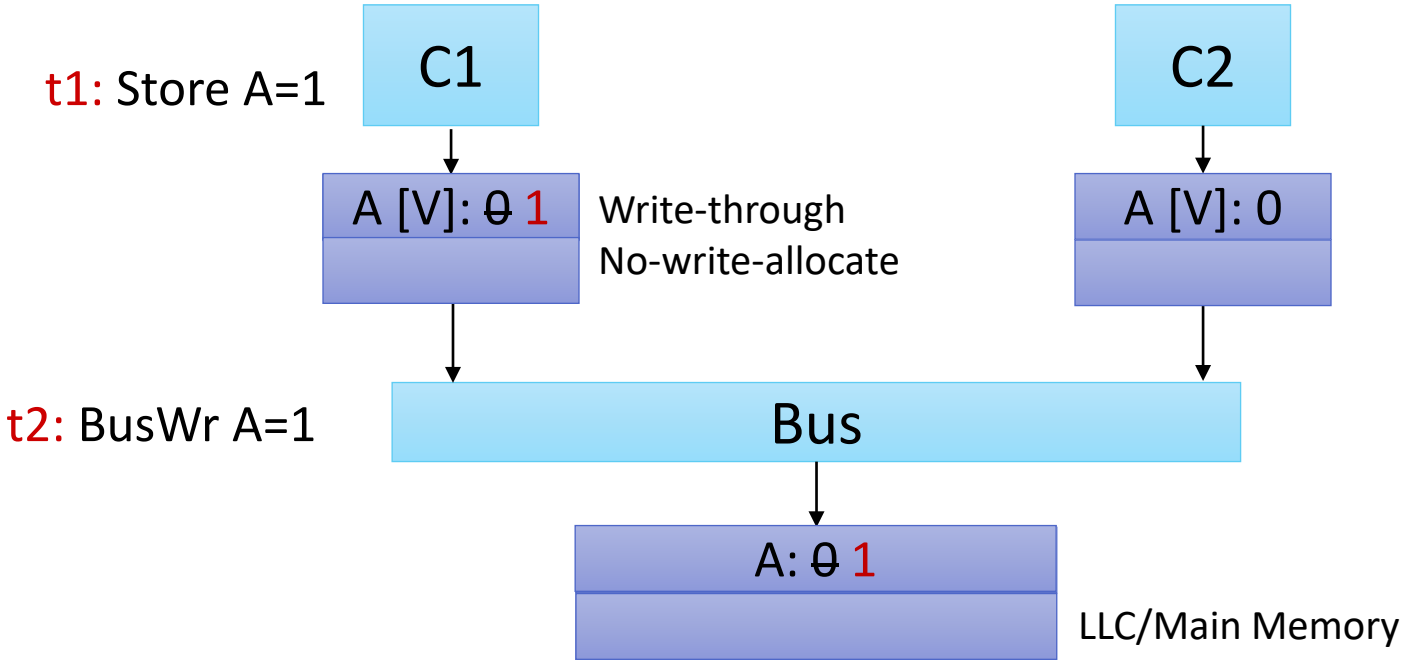
Example 1: Snoopy w/ Write -through

- Assume write-through, no-write-allocate cache
- Allows multiple readers, but writes through to bus
- Simple state machine for each cache frame



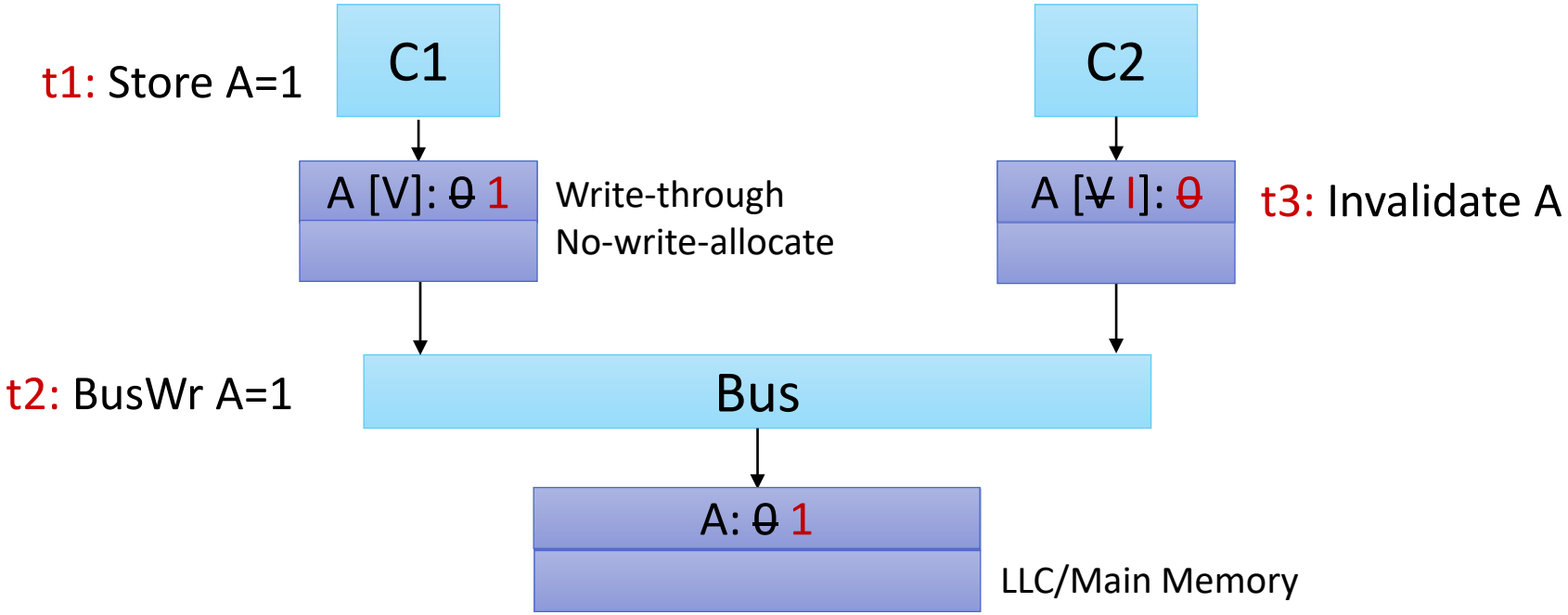
Example 1: Snoopy w/ Write -through

- Assume write-through, no-write-allocate cache
- Allows multiple readers, but writes through to bus
- Simple state machine for each cache frame



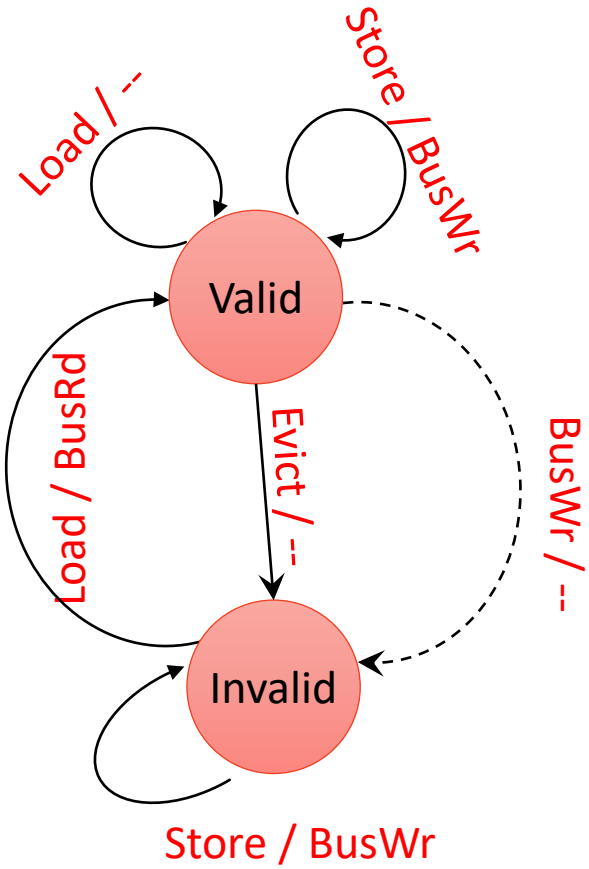
Example 1: Snoopy w/ Write -through

- Assume write-through, no-write-allocate cache
- Allows multiple readers, but writes through to bus
- Simple state machine for each cache frame



Valid/Invalid Snooping Protocol

- 1 bit to tack coherence state per cache block
  - Valid/Invalid
- Default state of any address not present in cache is “Invalid”
- Processor Actions
  - Ld, St, Evict
- Bus Messages
  - BusRd, BusWr



Transition caused by local action

Transition caused by bus message

Example 2: Write -Back caches

- Write-back caches are good
  - Drastically reduce bus write bandwidth

Example 2: Write -Back caches

- Write-back caches are good
  - Drastically reduce bus write bandwidth
- Add notion of "*ownership*" to Valid/Invalid
  - The "*owner*" has the **only UpToDate replica** of a cache block
    - Can update it freely

Example 2: Write -Back caches

- Write-back caches are good
  - Drastically reduce bus write bandwidth
- Add notion of "*ownership*" to Valid/Invalid
  - The "*owner*" has the **only UpToDate replica** of a cache block
    - Can update it freely
  - **Multiple sharers** are ok
    - None is allowed to write without gaining ownership
  - But, there could be **only one owner**
  - On a read, system must check if there is an owner
    - If yes, take away ownership and owner becomes a sharer
    - The reader becomes another sharer

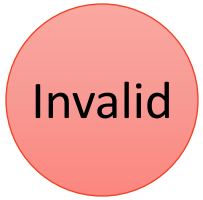
Modified/Shared/Invalid (MSI) States

- Each cache, tracks 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 lower-level caches) out of date

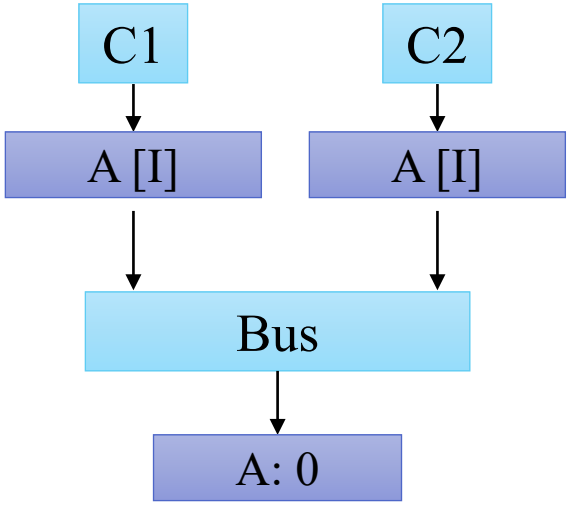
Modified/Shared/Invalid (MSI) States

- Each cache, tracks 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 lower-level caches) out of date
- Processor/Core Actions
  - Load, Store, ($ block) Evict
- Bus Messages
  - BusRd, BusRdX, BusInv, BusWB, BusReply
    (Here for simplicity, some messages can be combined)

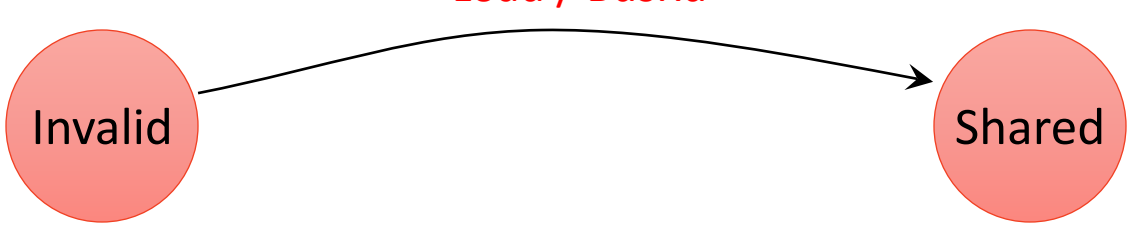
Simple MSI Protocol (1)



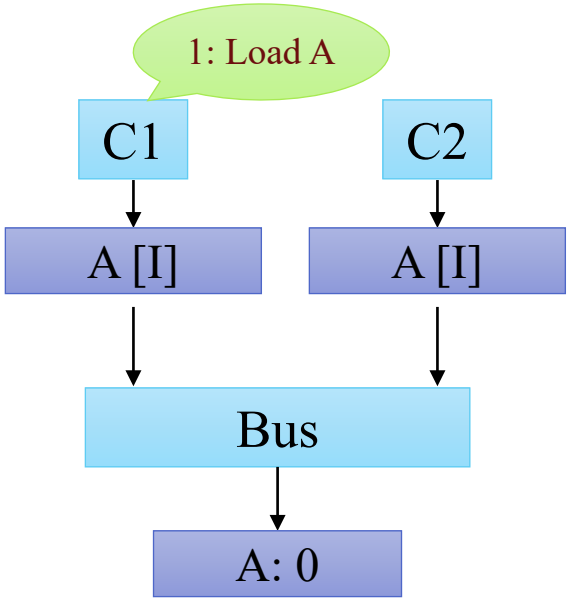
→ **Transition caused by local action**



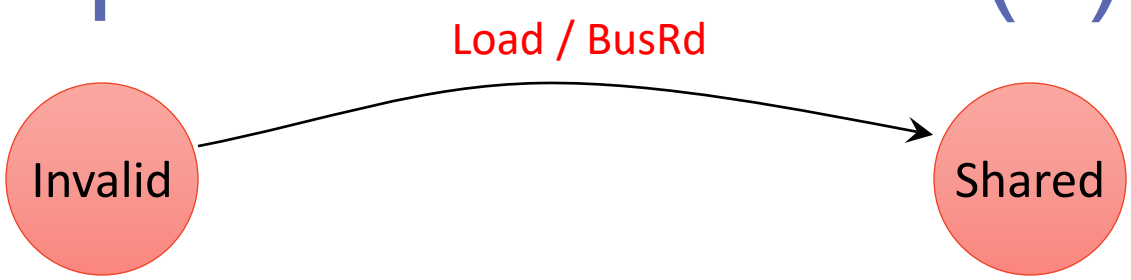
Simple MSI Protocol (1)



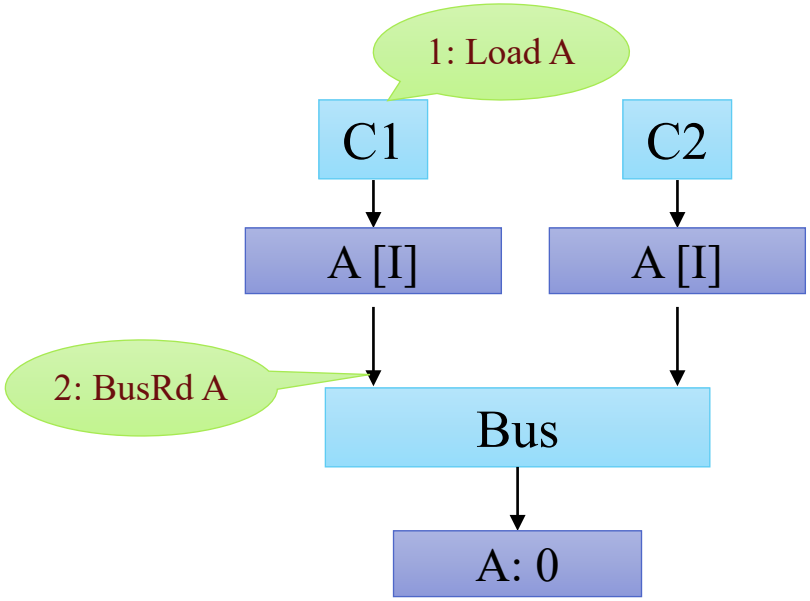
→ **Transition caused by local action**



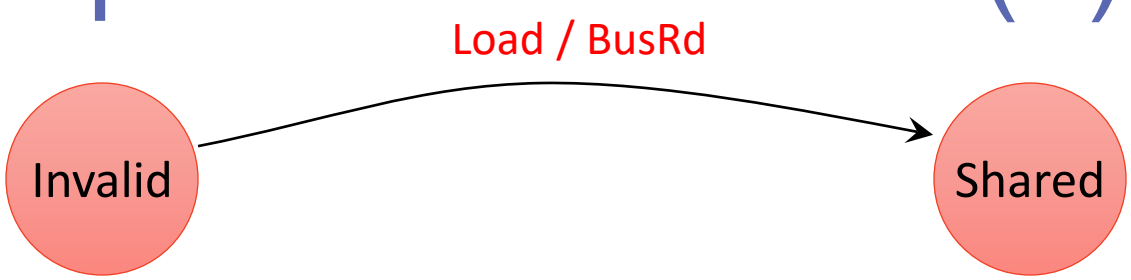
Simple MSI Protocol (1)



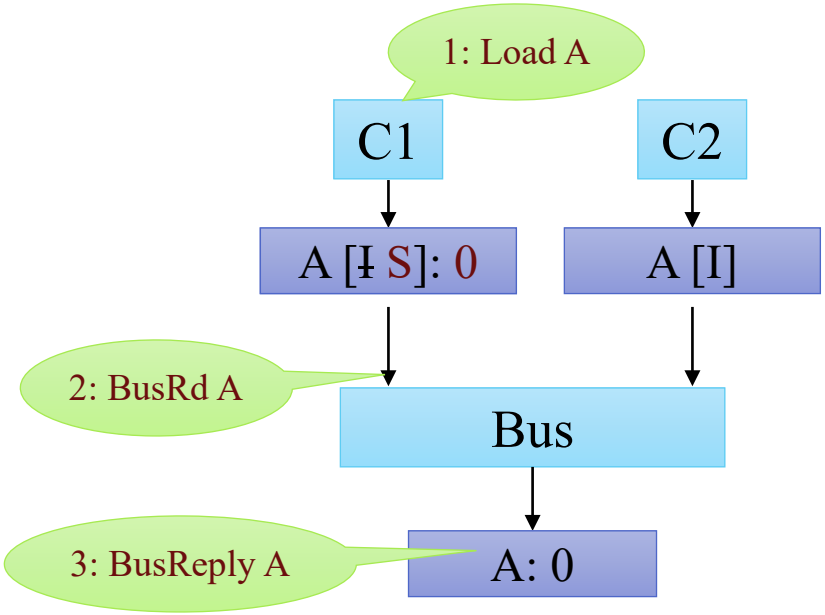
→ **Transition caused by local action**



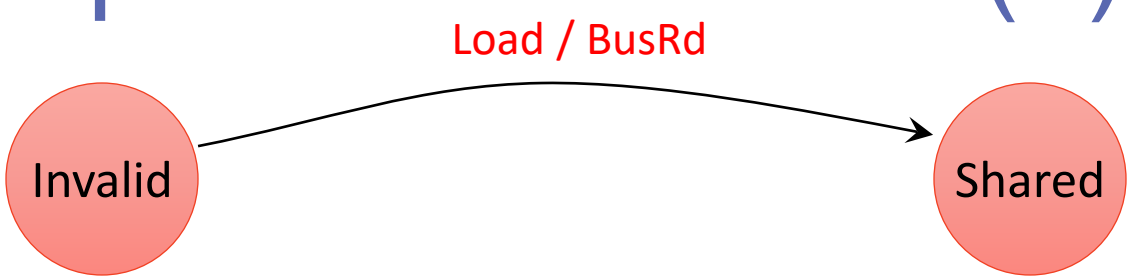
Simple MSI Protocol (1)



→ **Transition caused by local action**

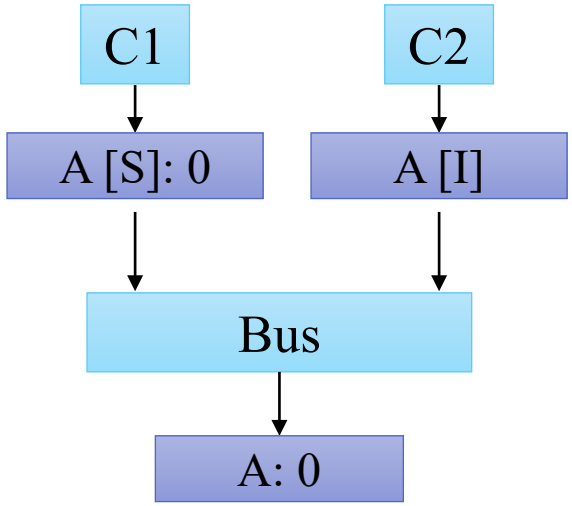


Simple MSI Protocol (2)

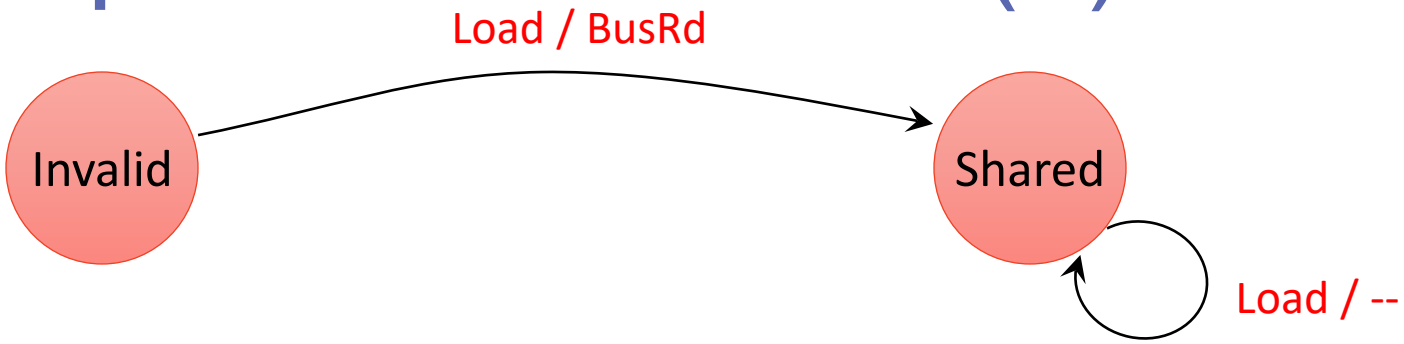


Transition caused by local action

Transition caused by bus message

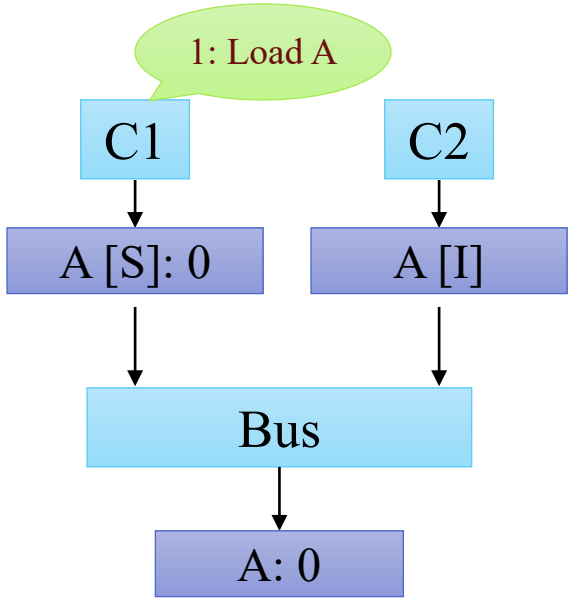


Simple MSI Protocol (2)

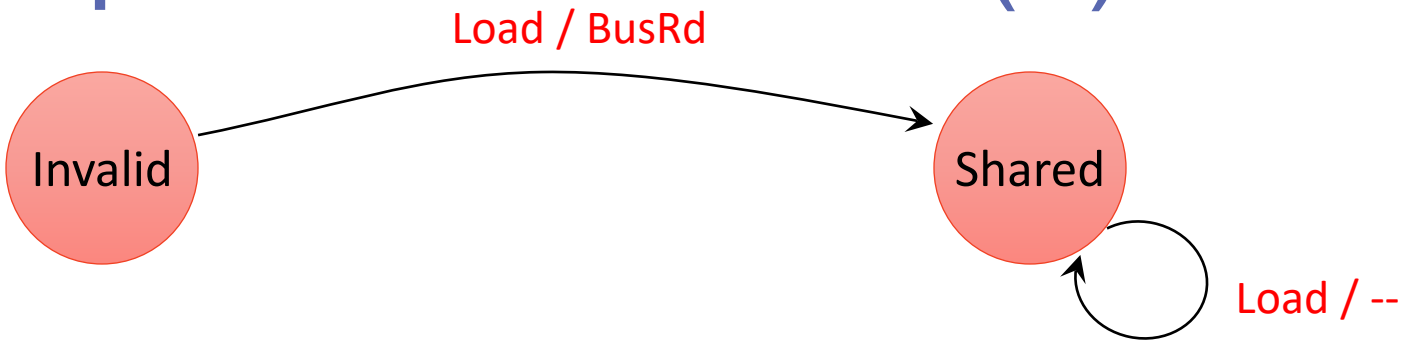


→ **Transition caused by local action**

⇢ **Transition caused by bus message**

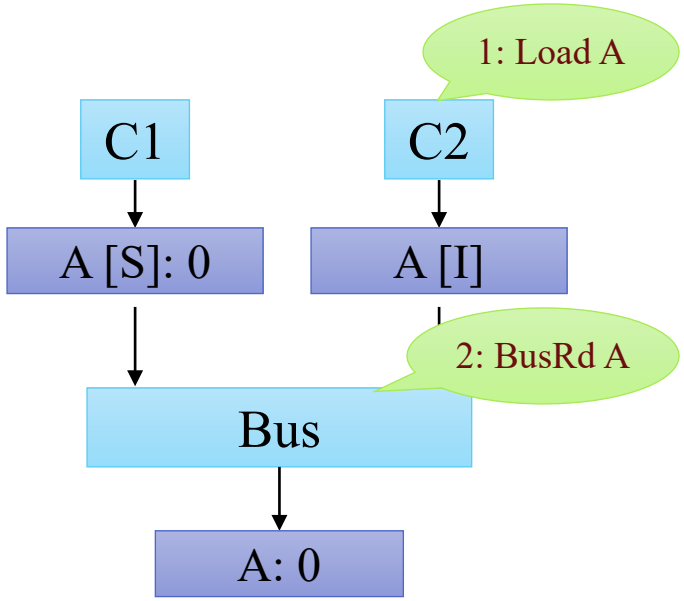


Simple MSI Protocol (2)

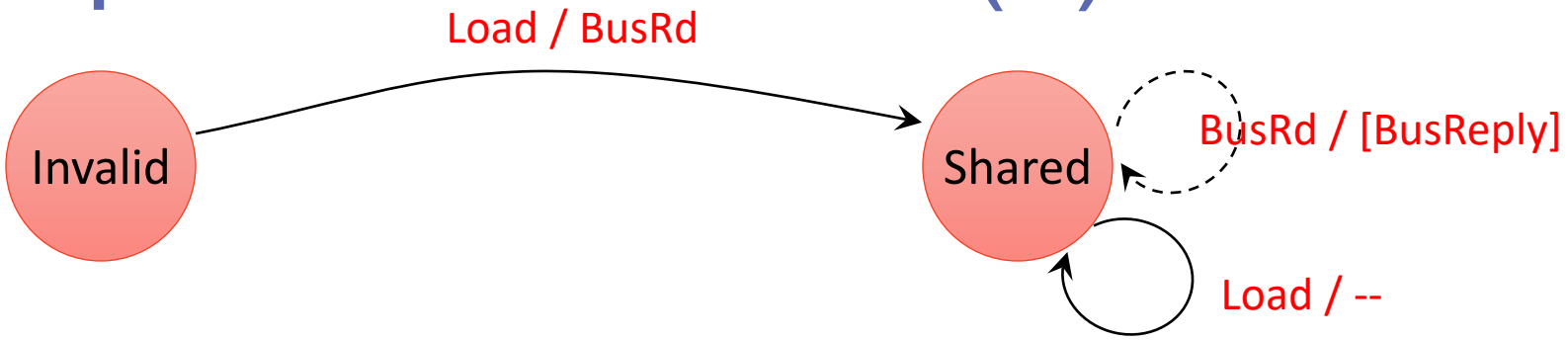


Transition caused by local action

Transition caused by bus message

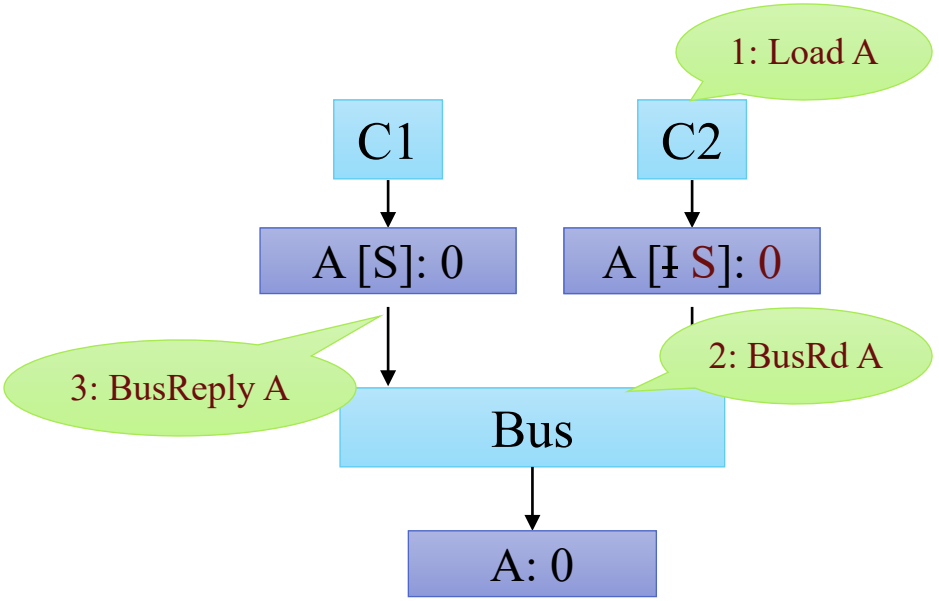


Simple MSI Protocol (2)

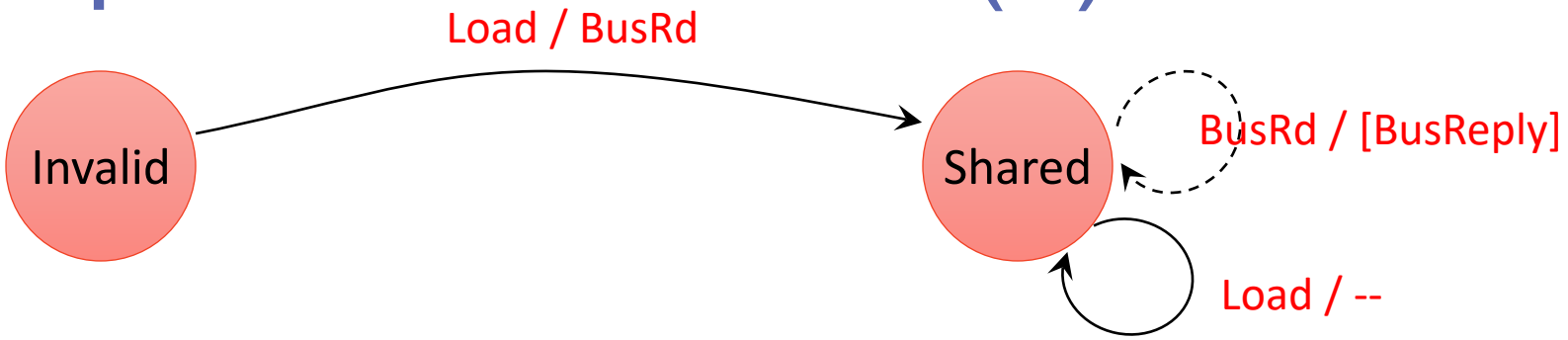


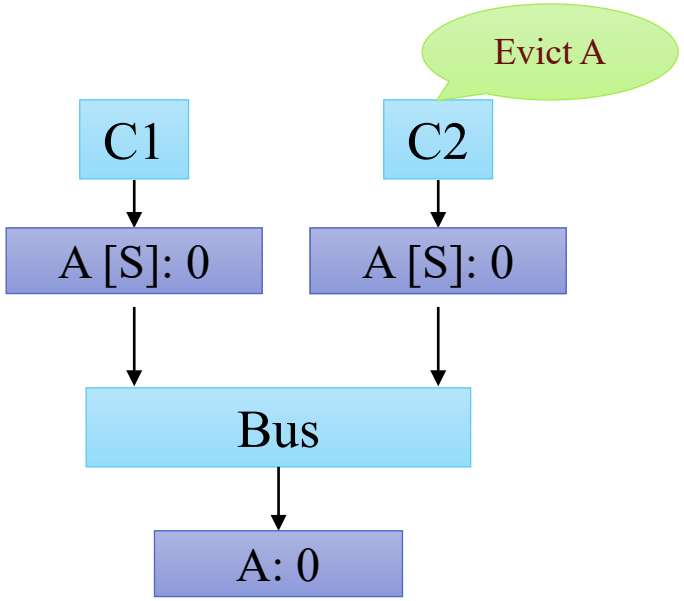
Transition caused by local action

Transition caused by bus message

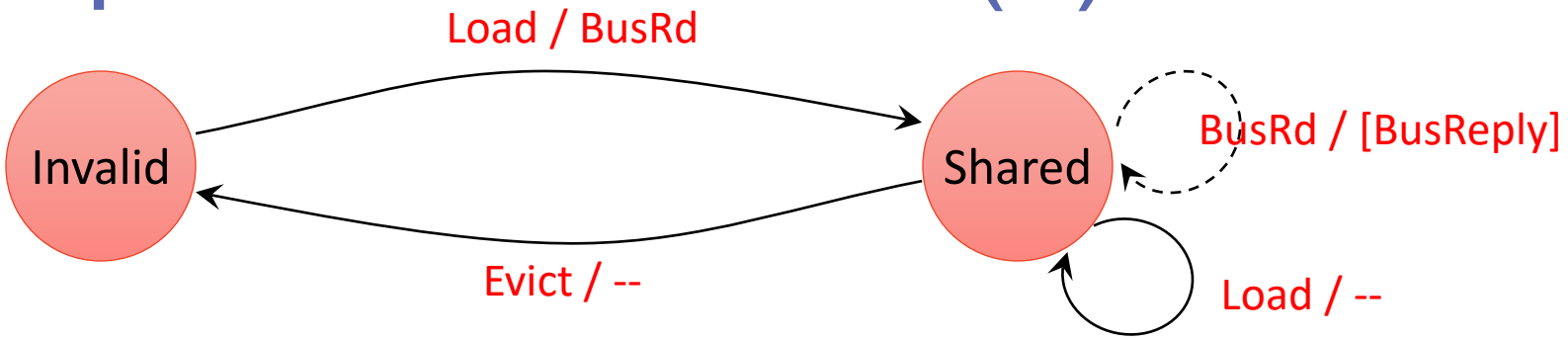


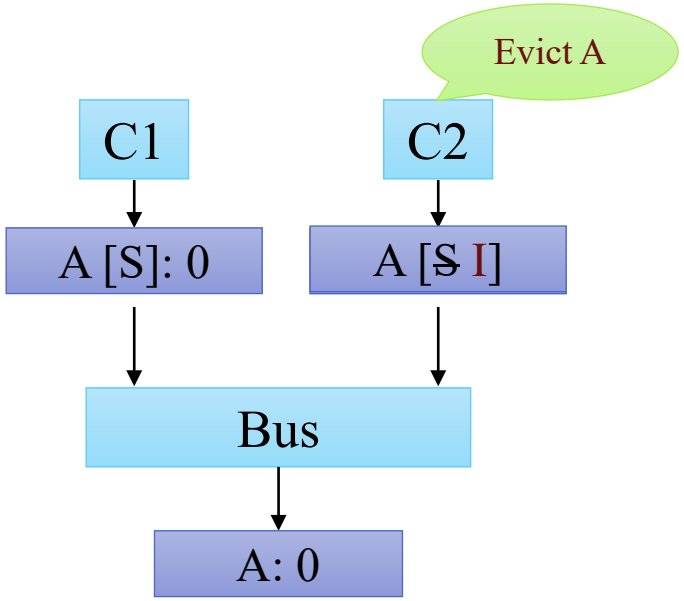
Simple MSI Protocol (3)



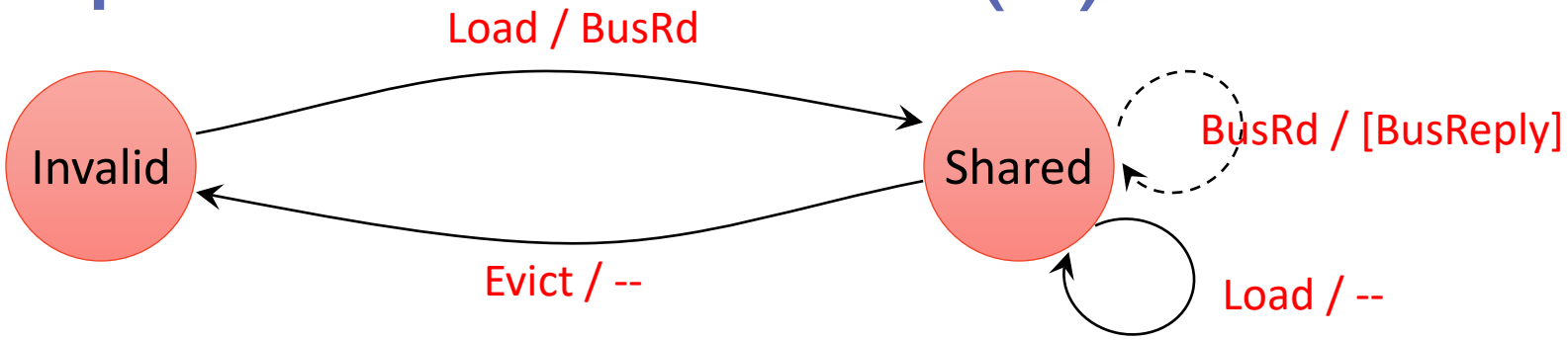


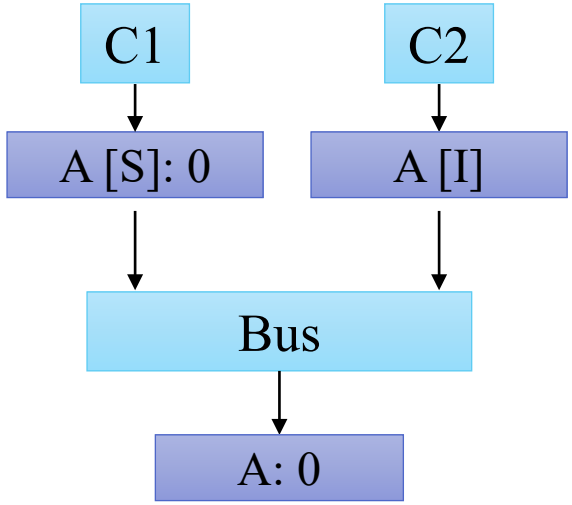
Simple MSI Protocol (3)



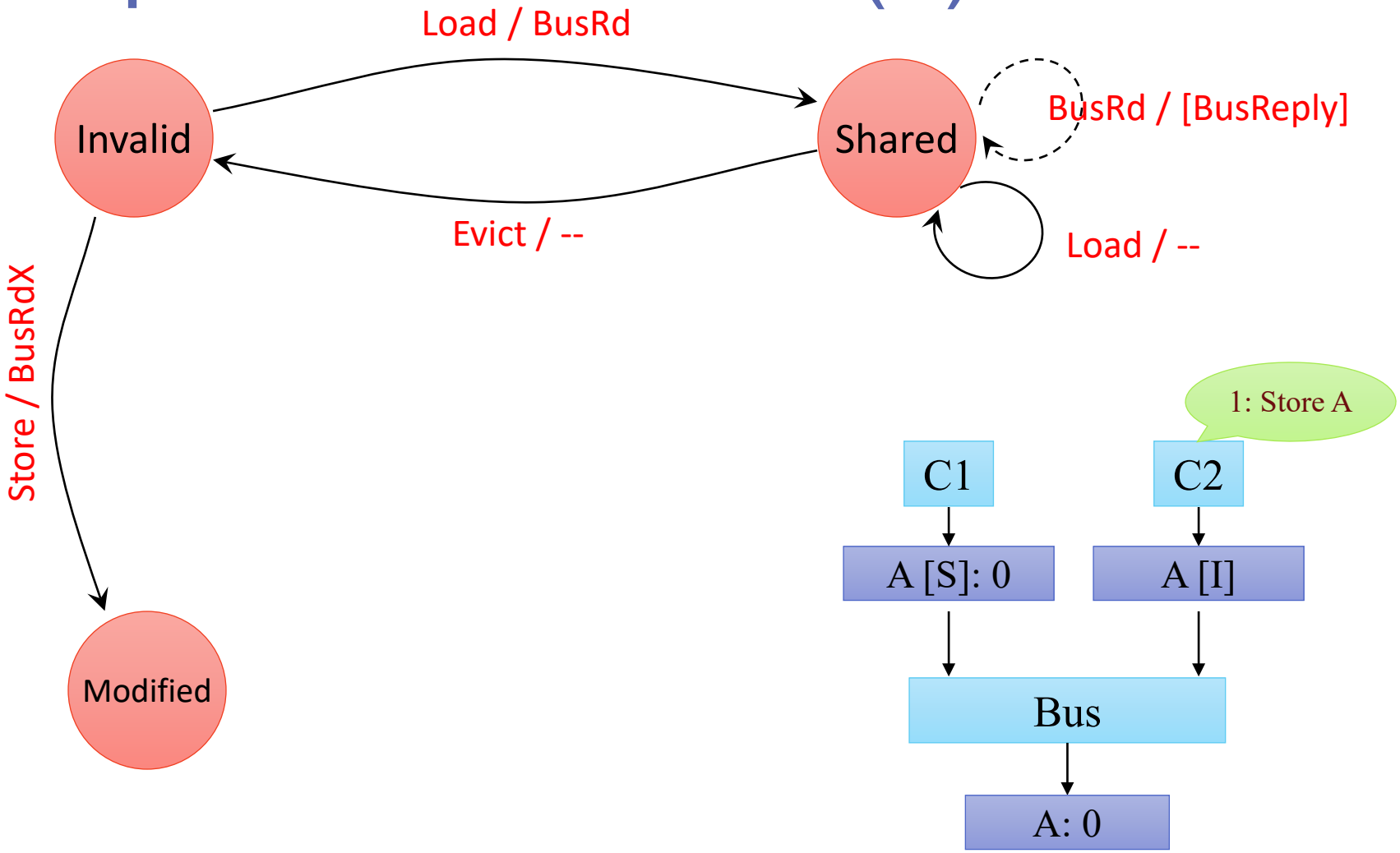


Simple MSI Protocol (4)

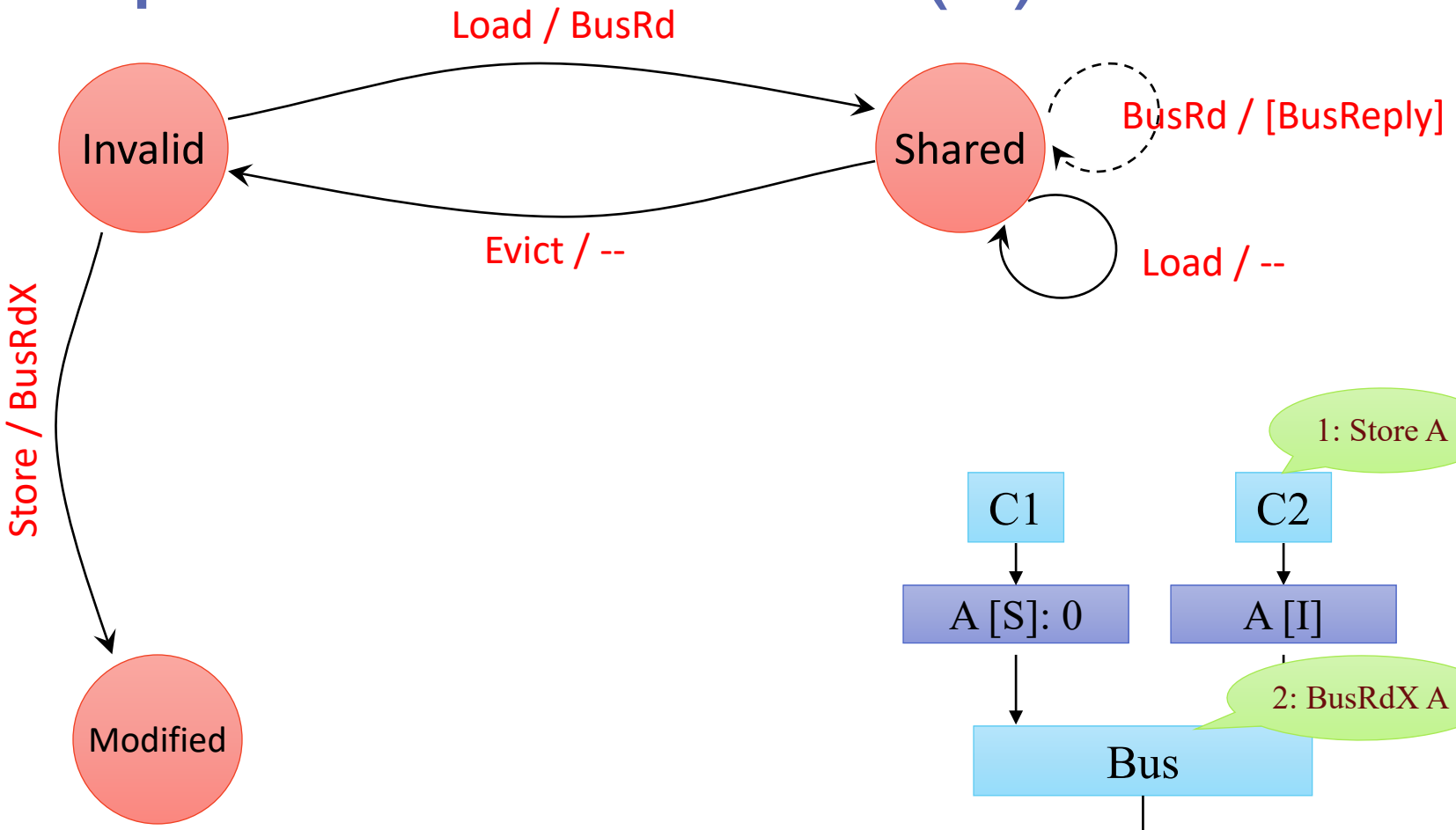




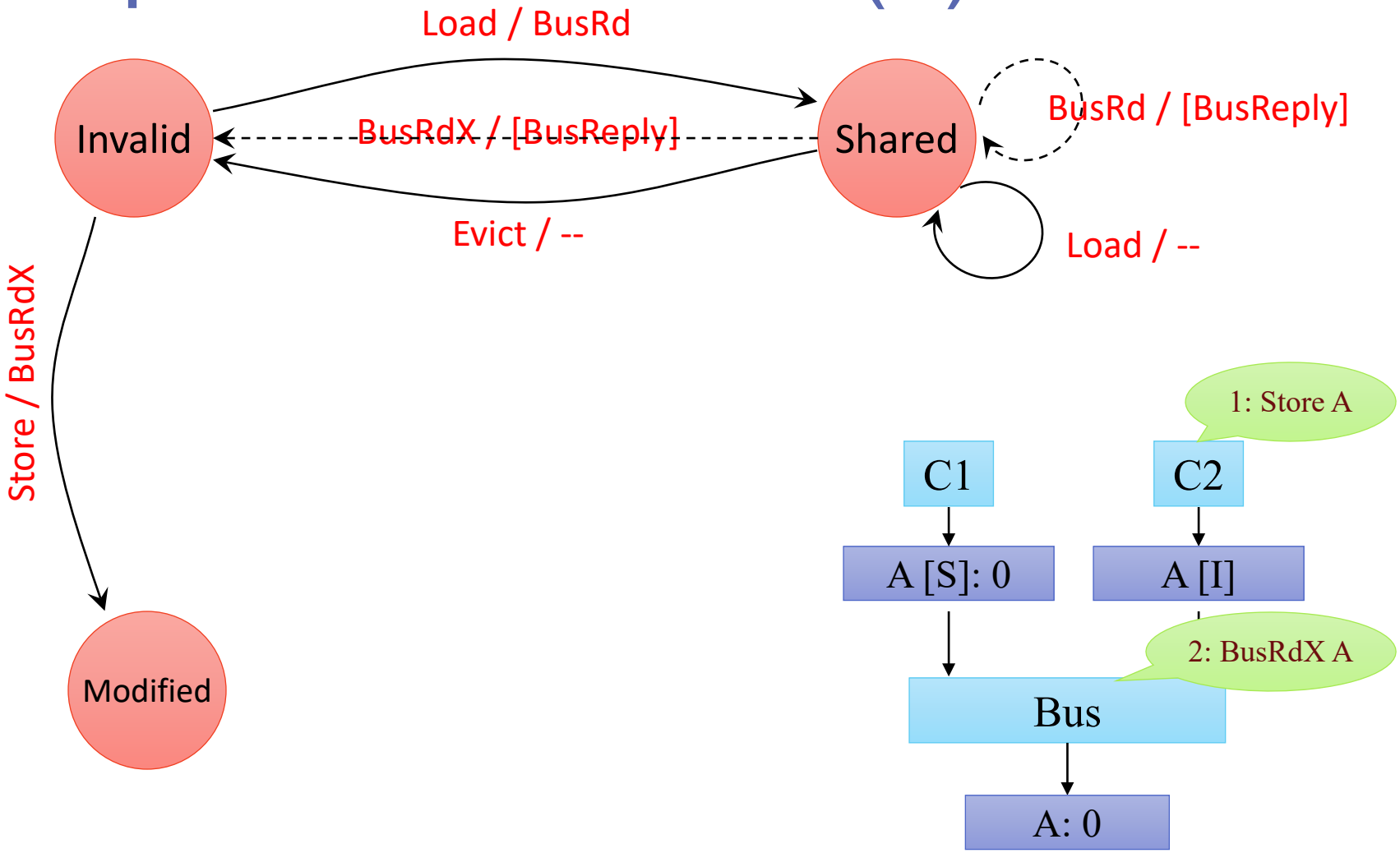
Simple MSI Protocol (4)



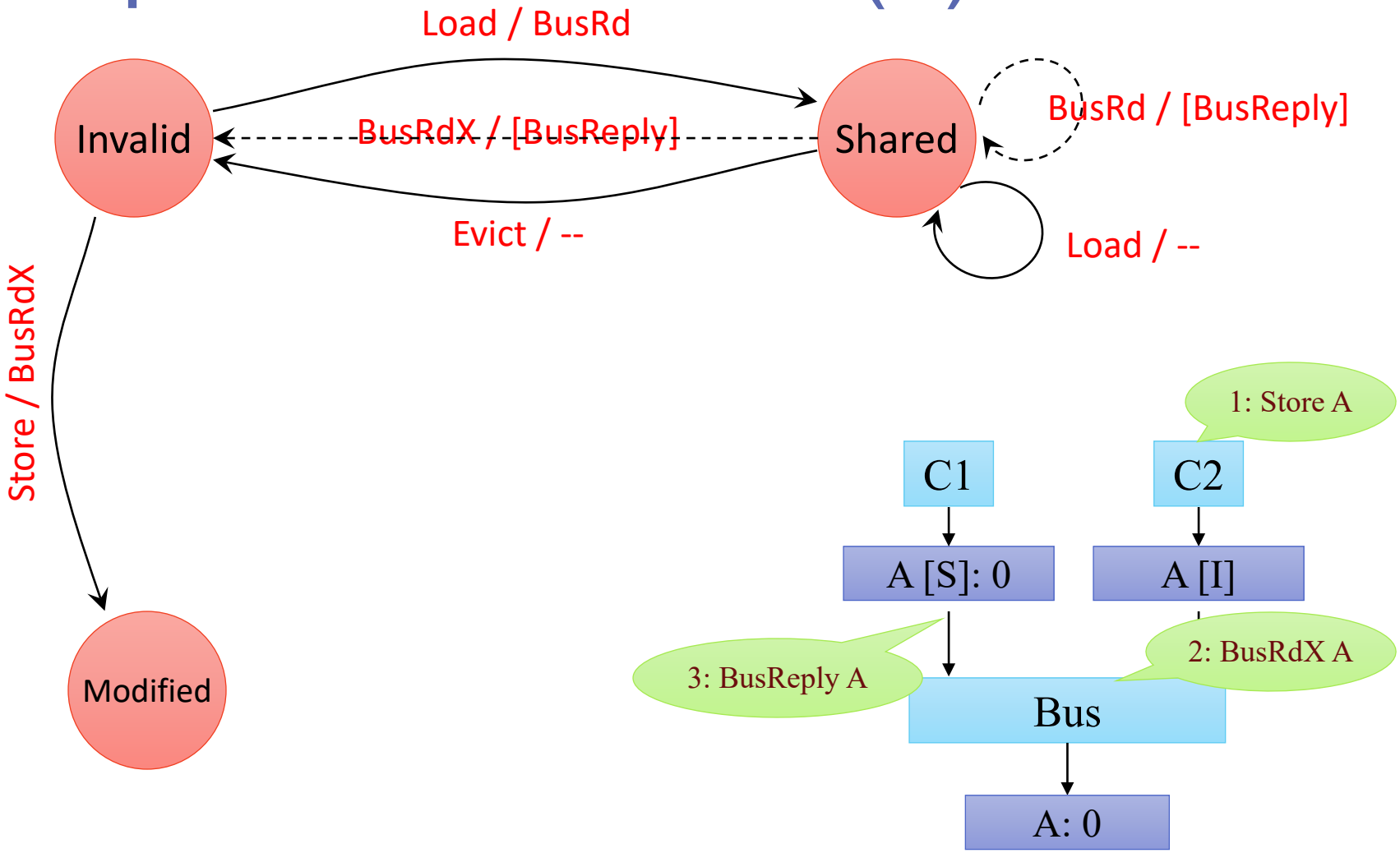
Simple MSI Protocol (4)



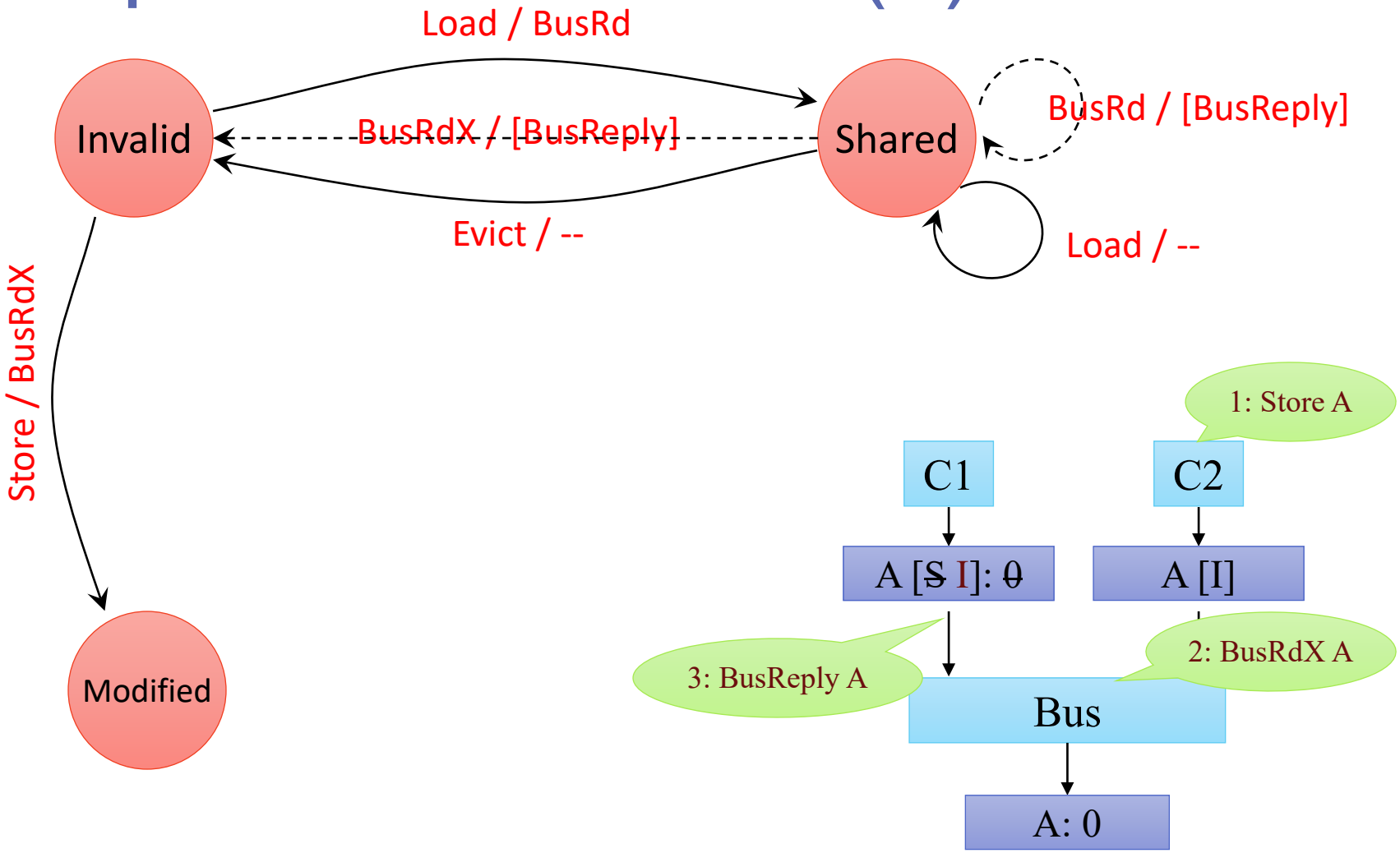
Simple MSI Protocol (4)



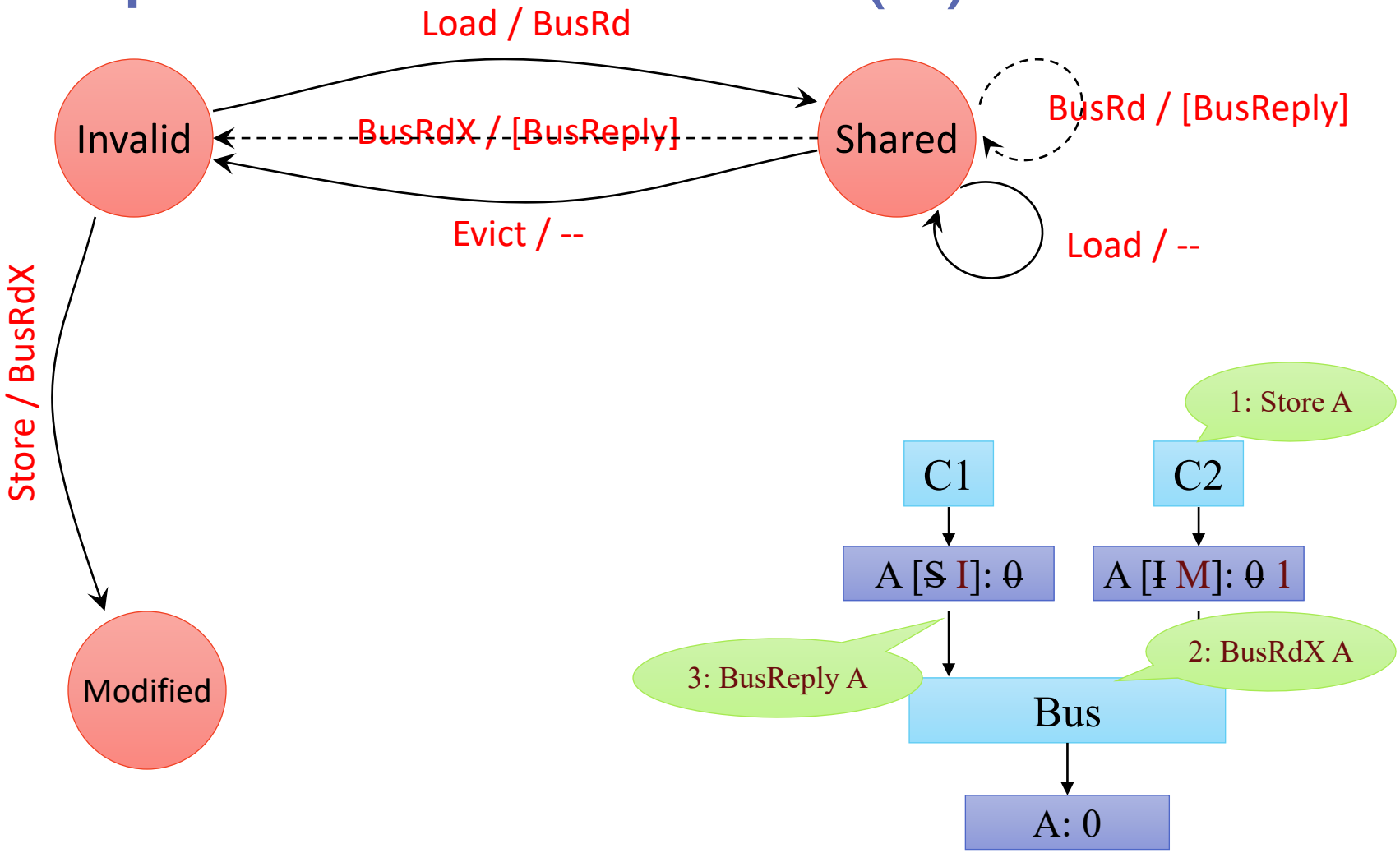
Simple MSI Protocol (4)



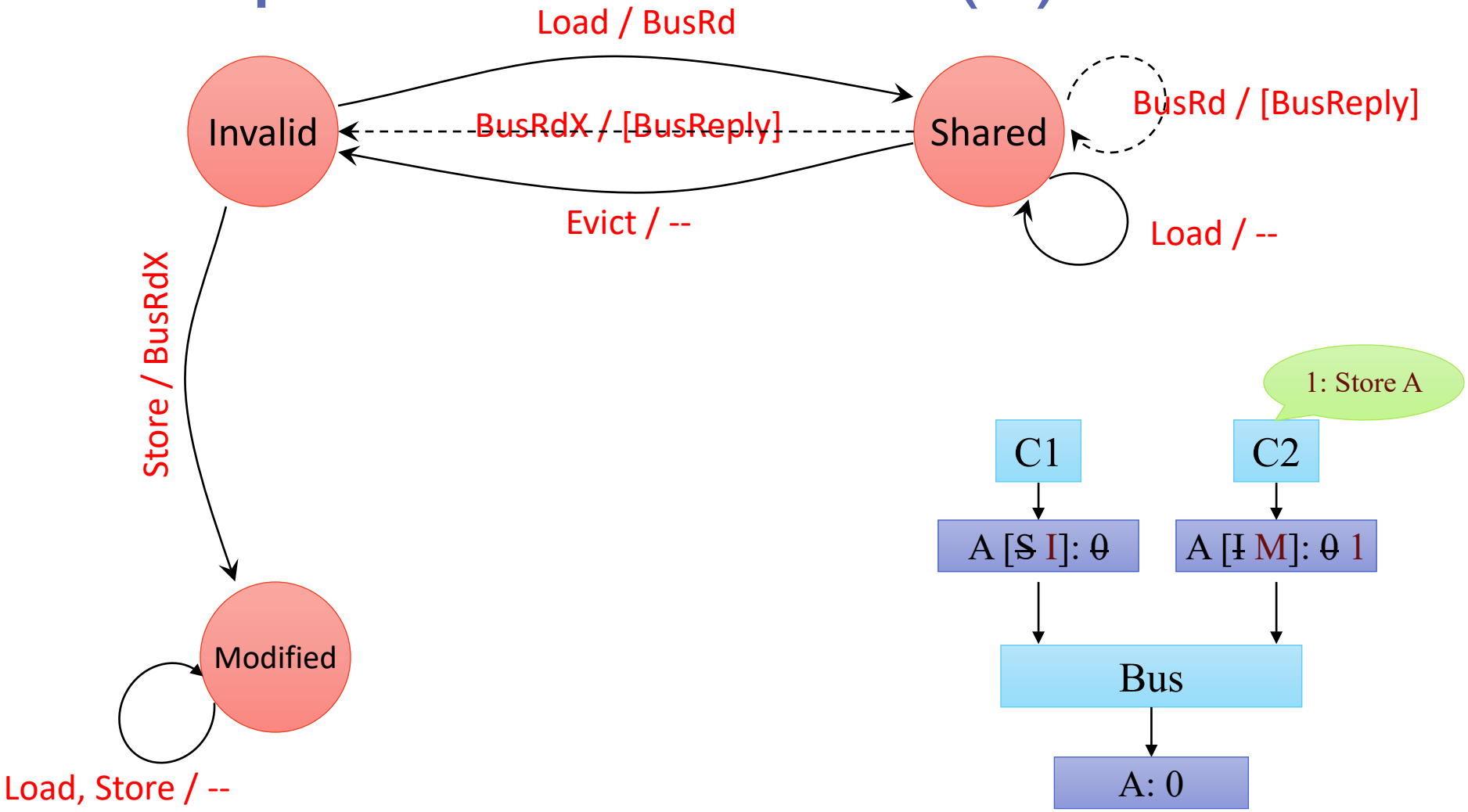
Simple MSI Protocol (4)



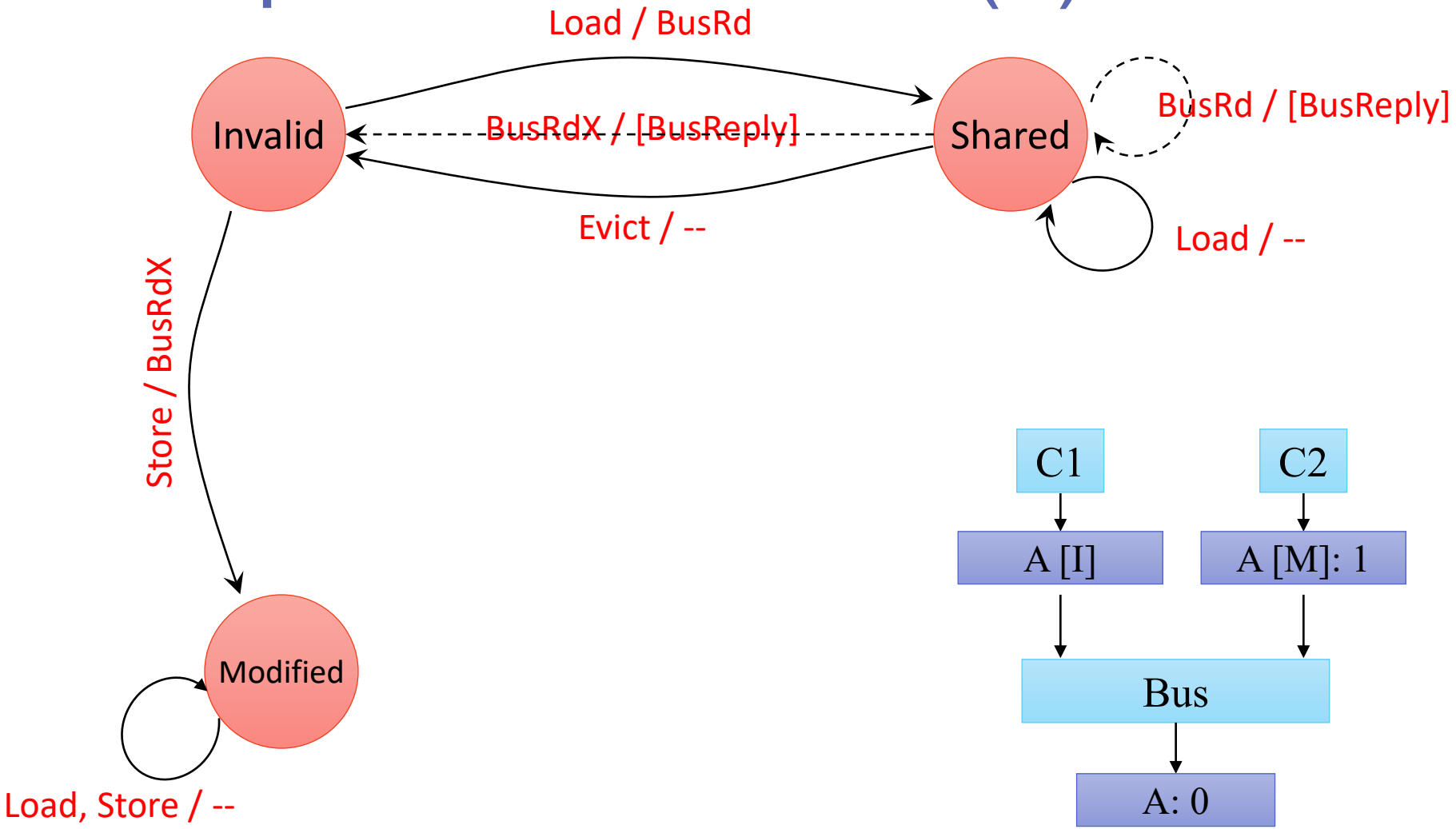
Simple MSI Protocol (4)



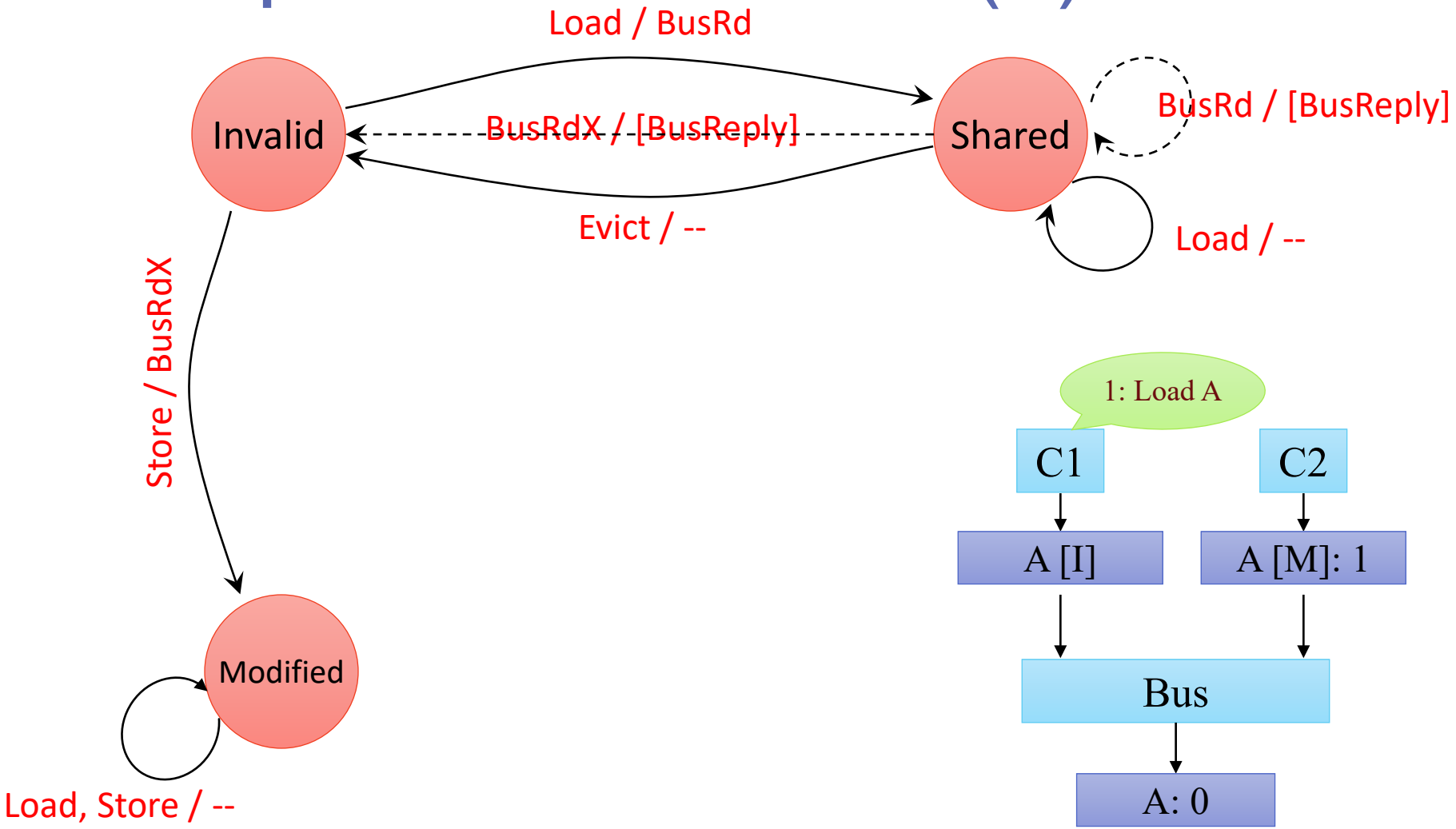
Simple MSI Protocol (4)



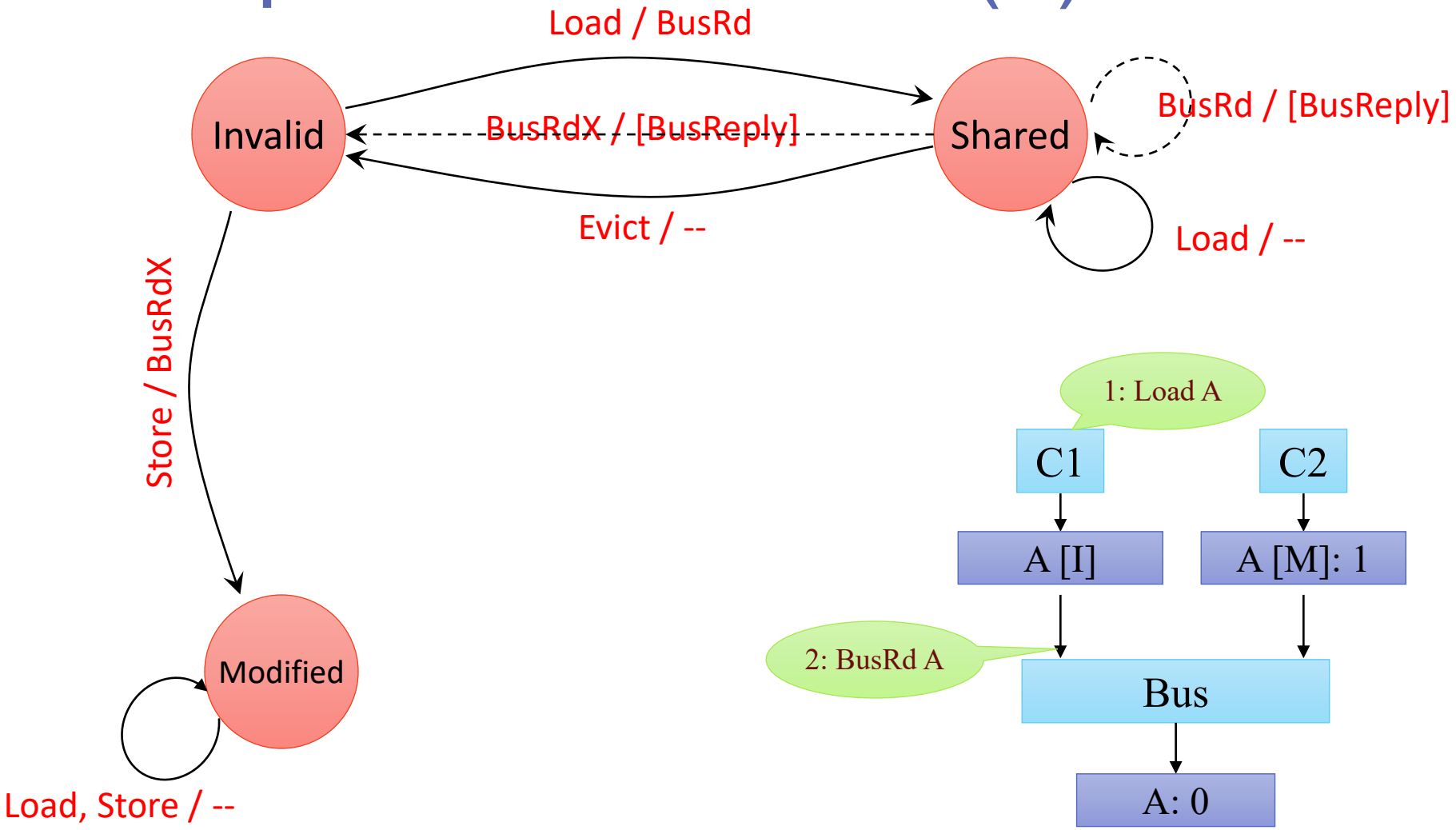
Simple MSI Protocol (5)



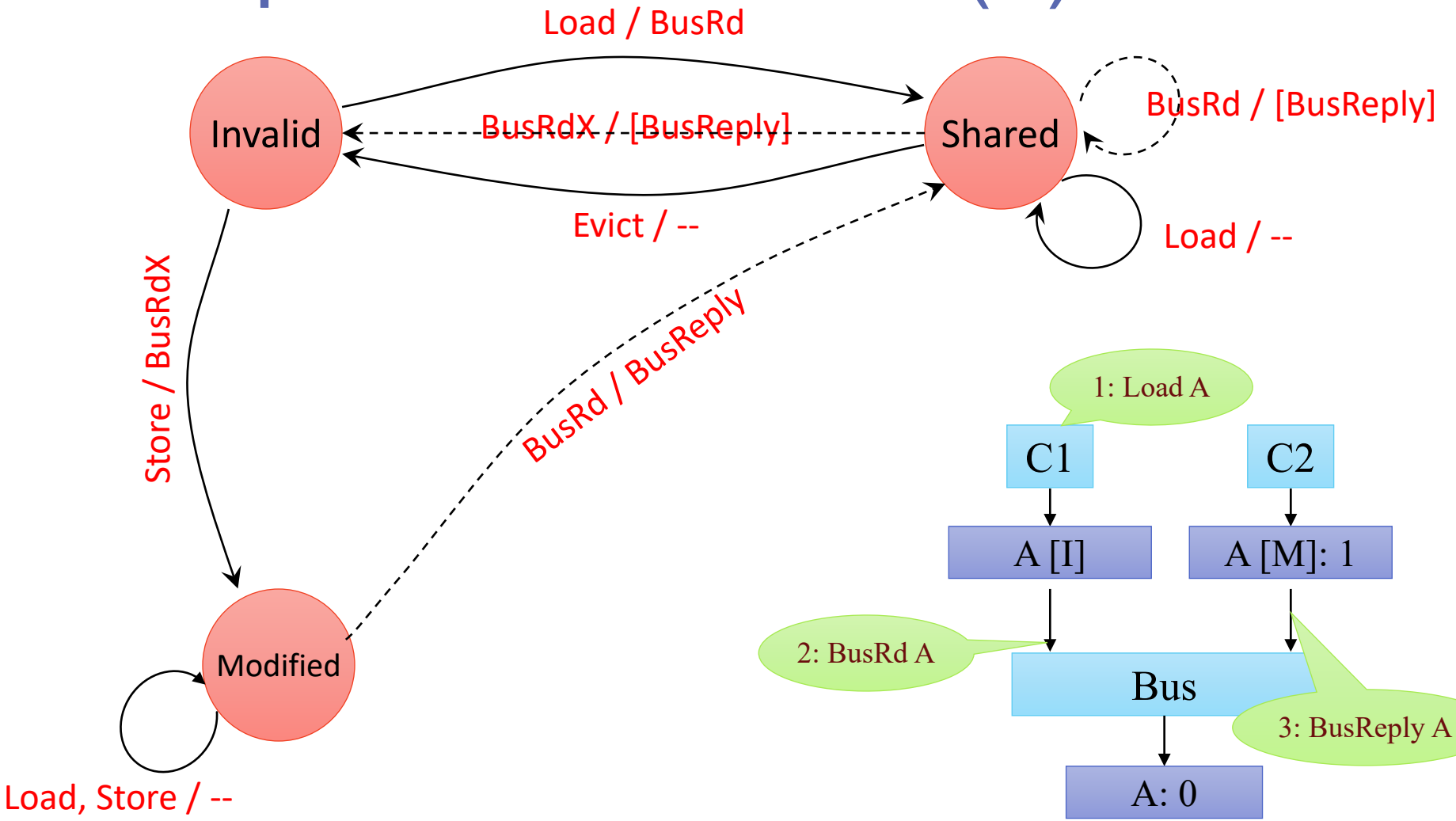
Simple MSI Protocol (5)



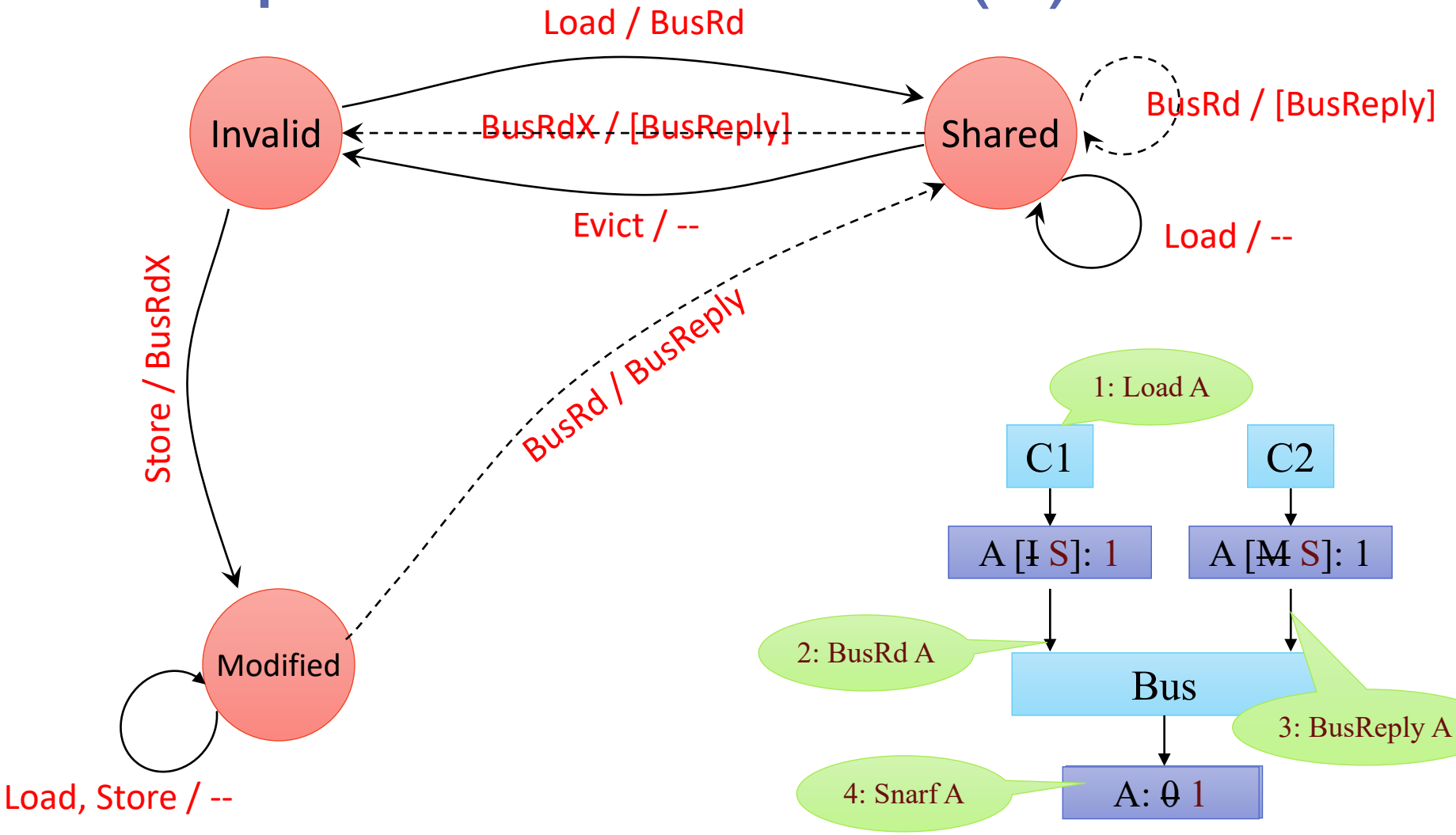
Simple MSI Protocol (5)



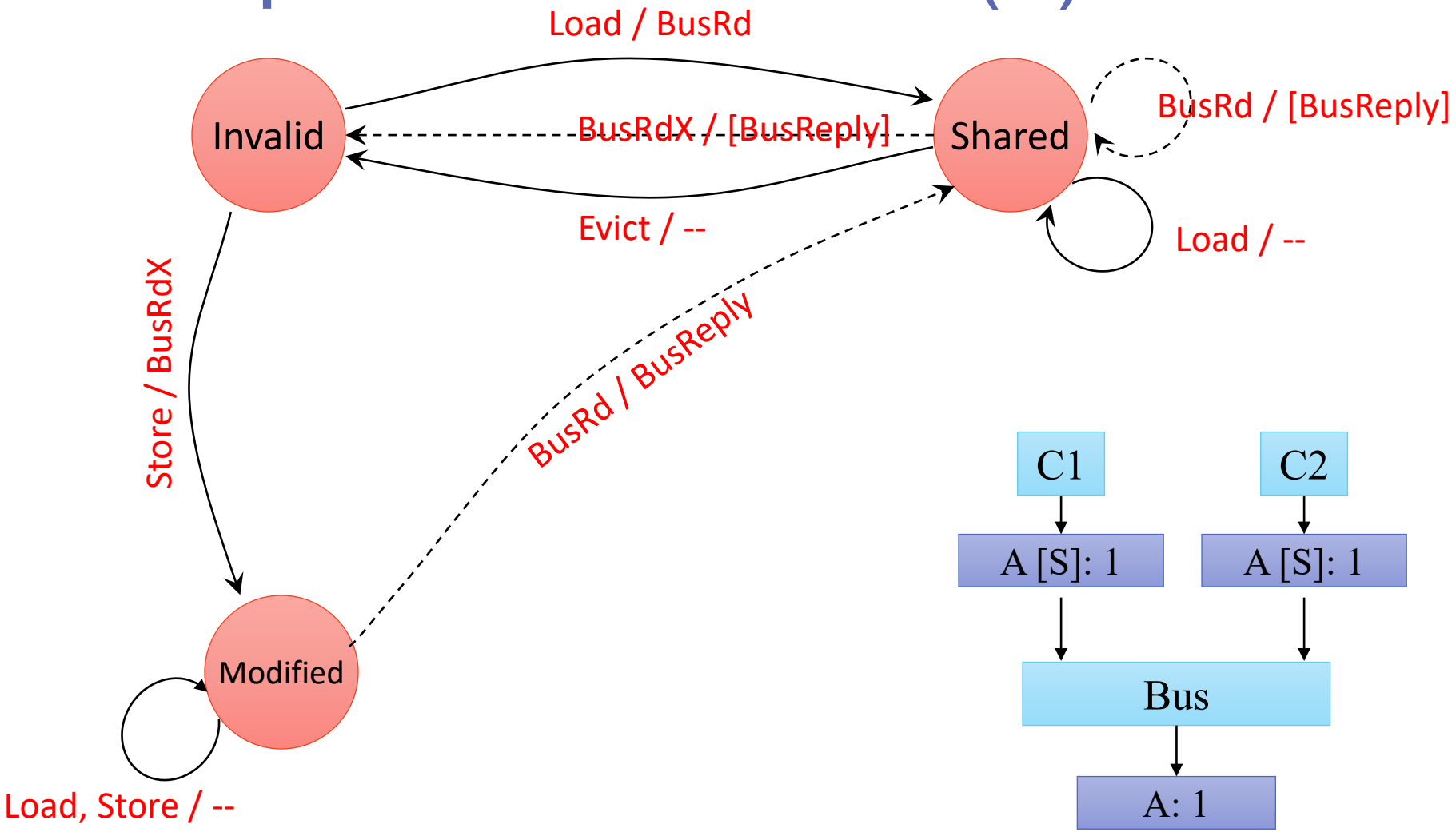
Simple MSI Protocol (5)



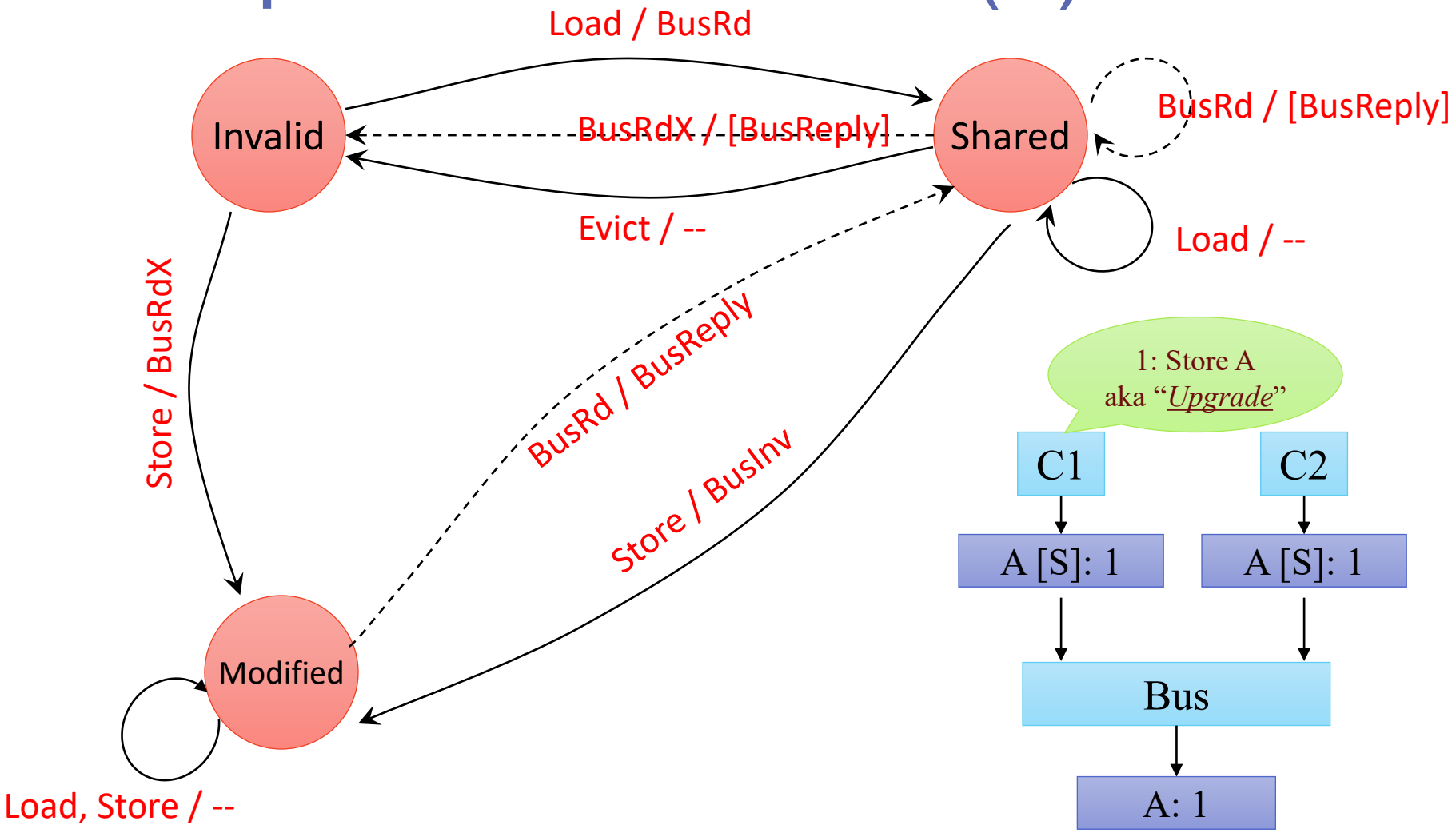
Simple MSI Protocol (5)



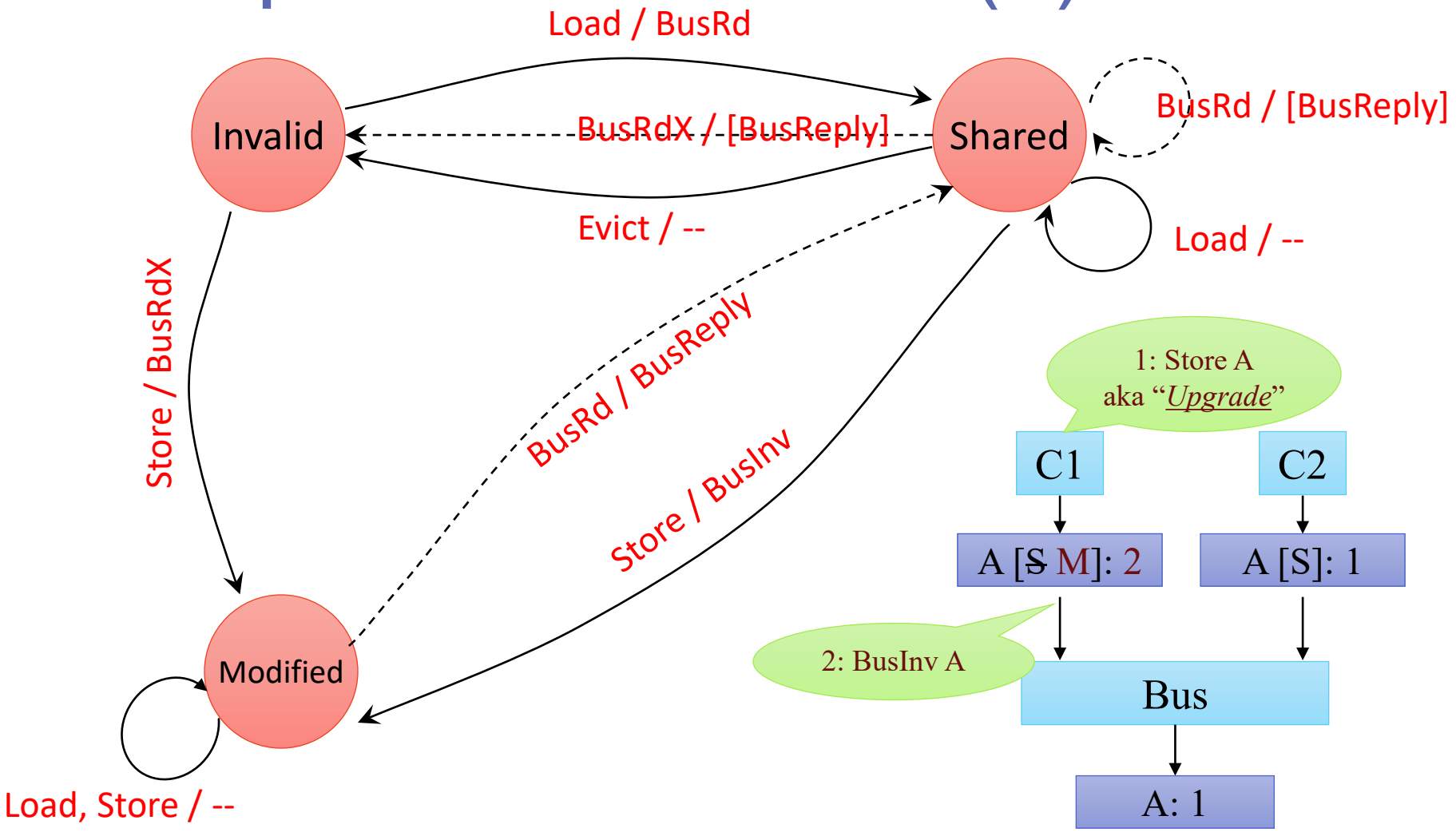
Simple MSI Protocol (6)



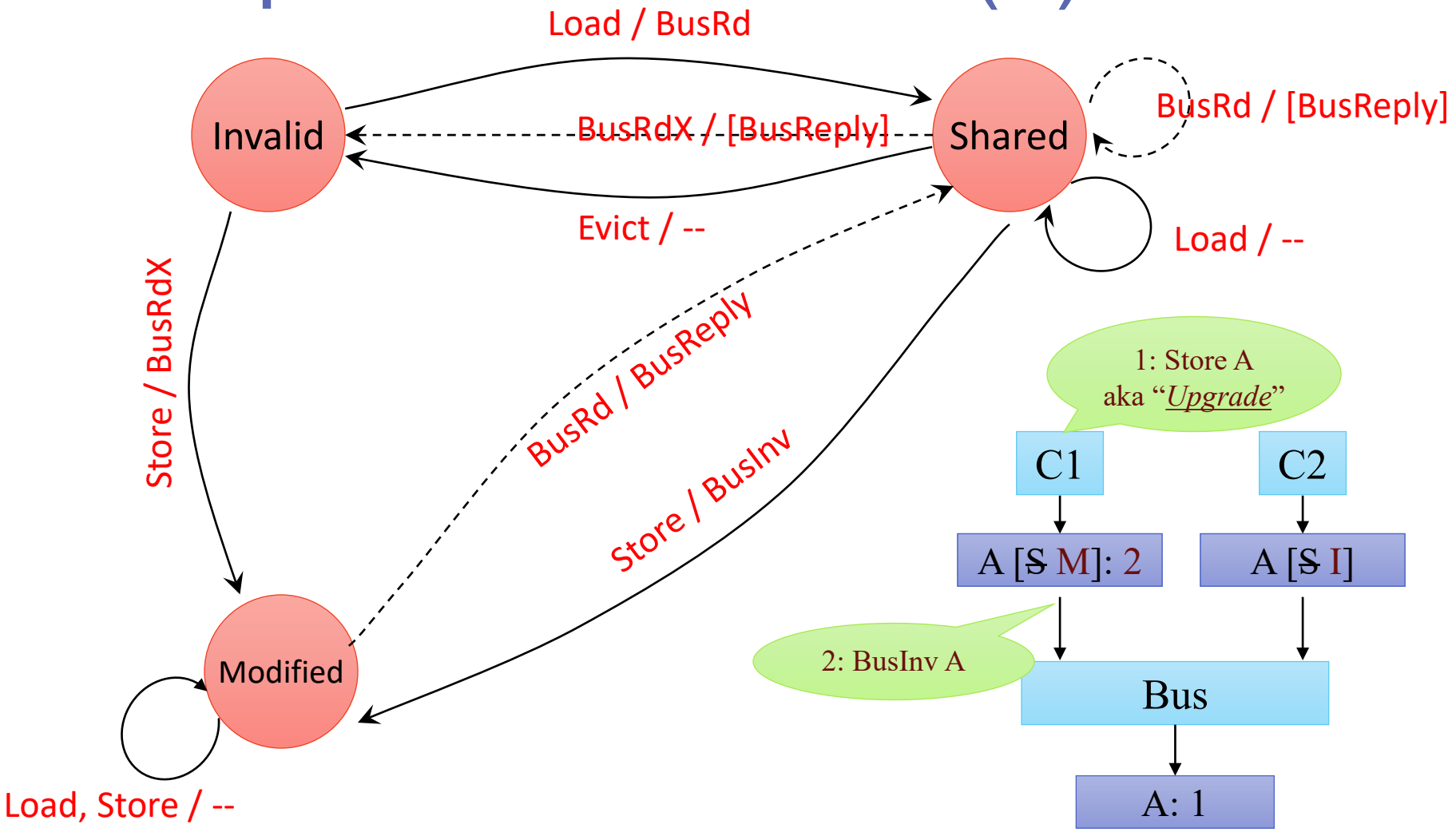
Simple MSI Protocol (6)



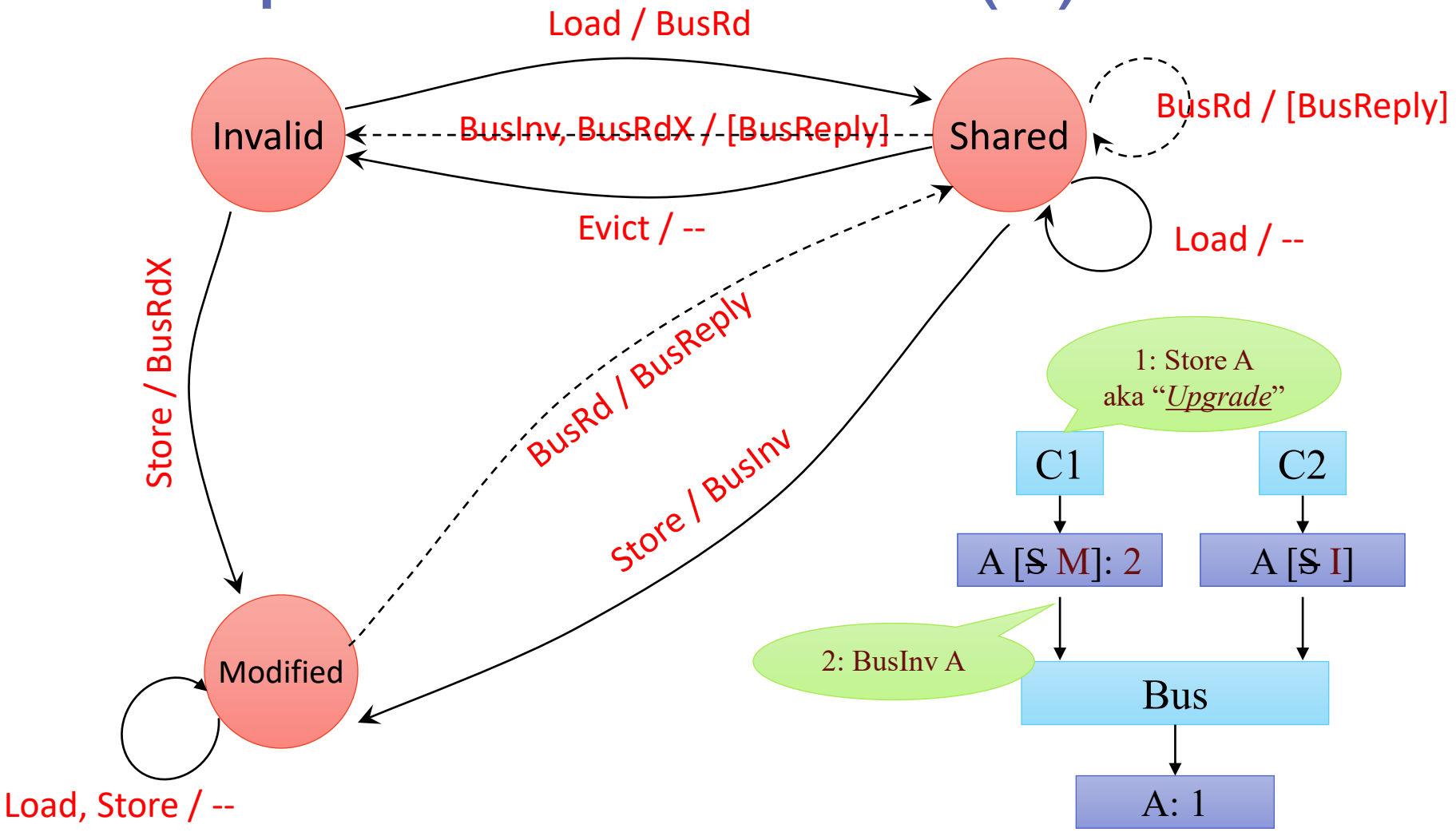
Simple MSI Protocol (6)



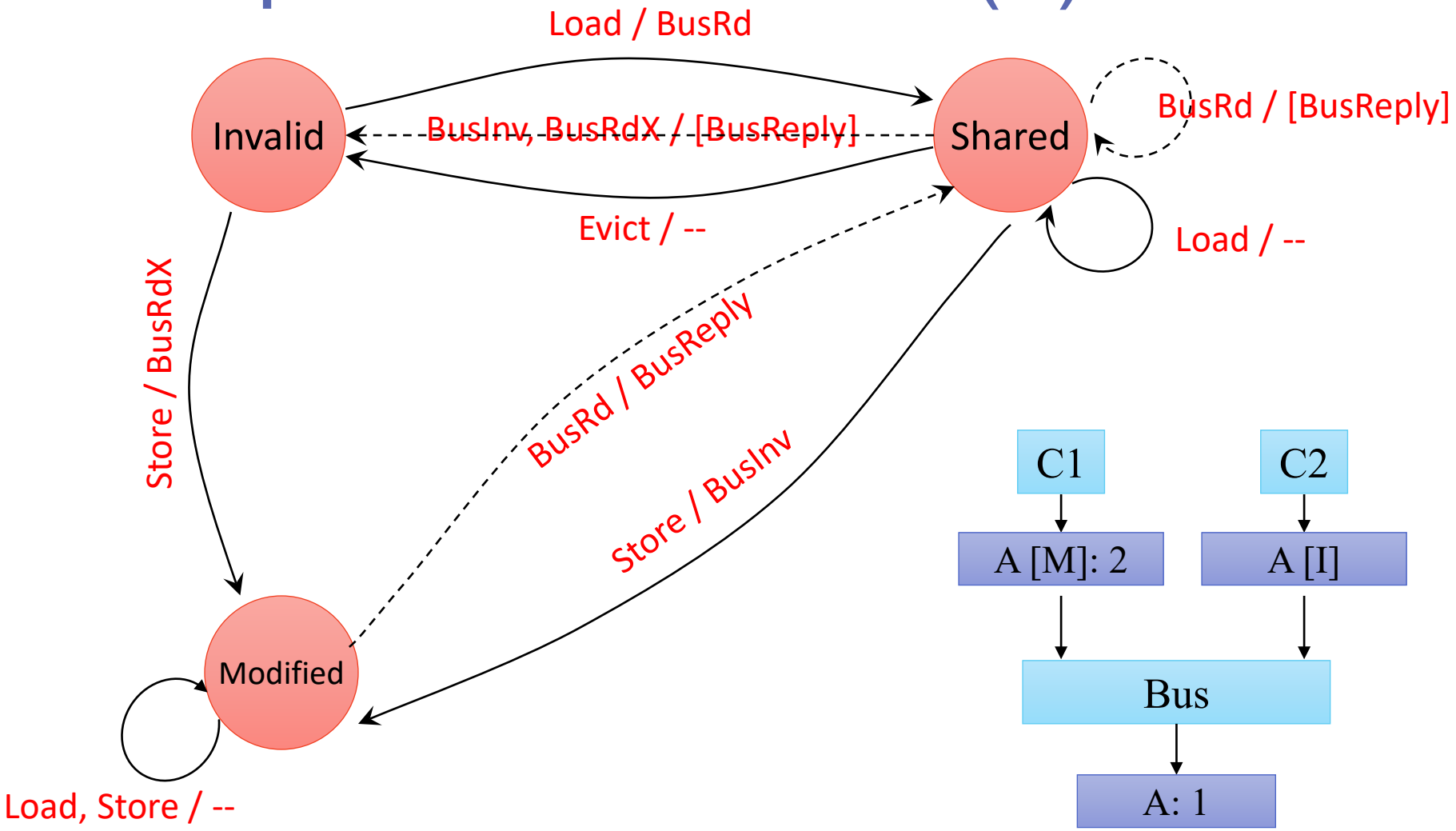
Simple MSI Protocol (6)



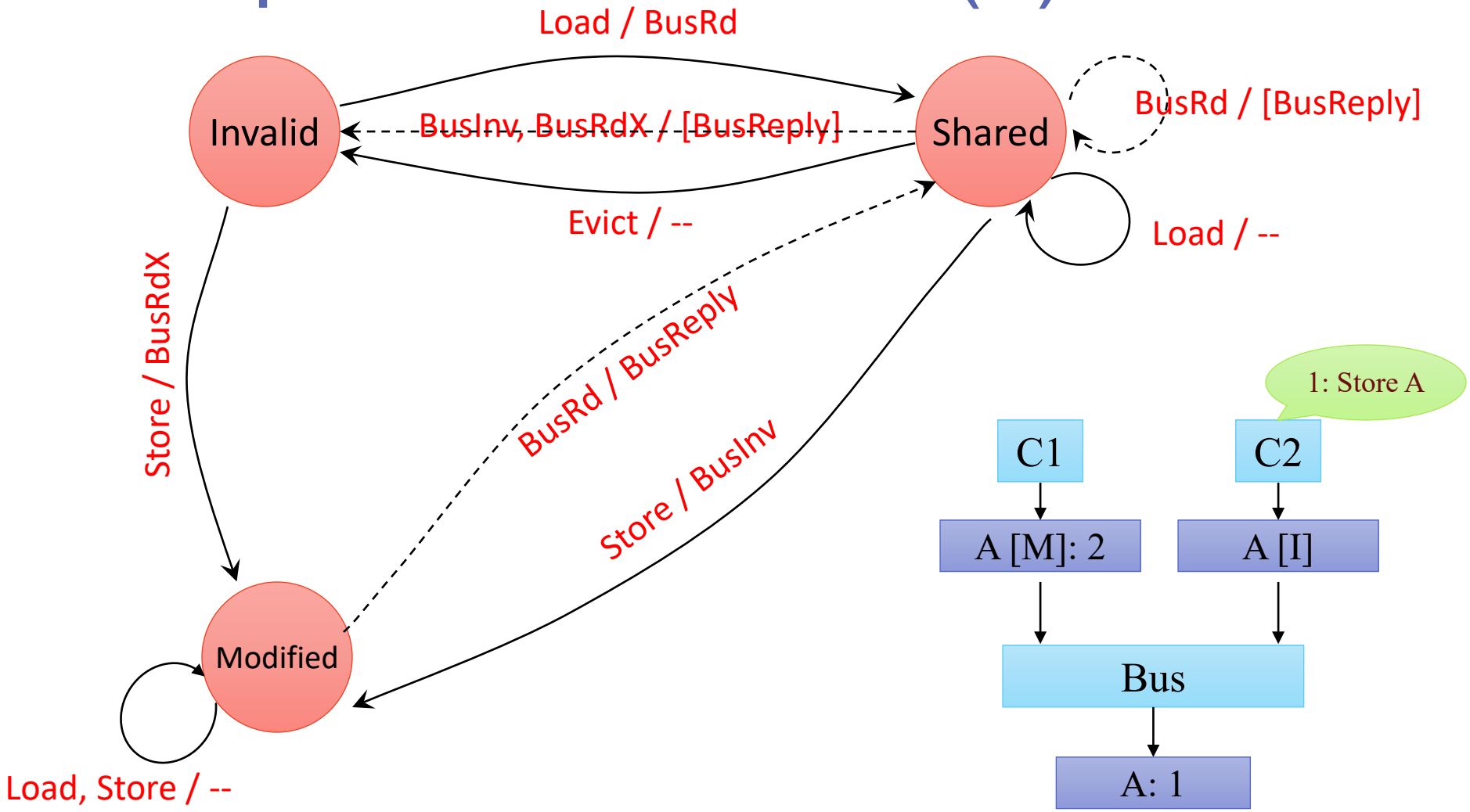
Simple MSI Protocol (6)



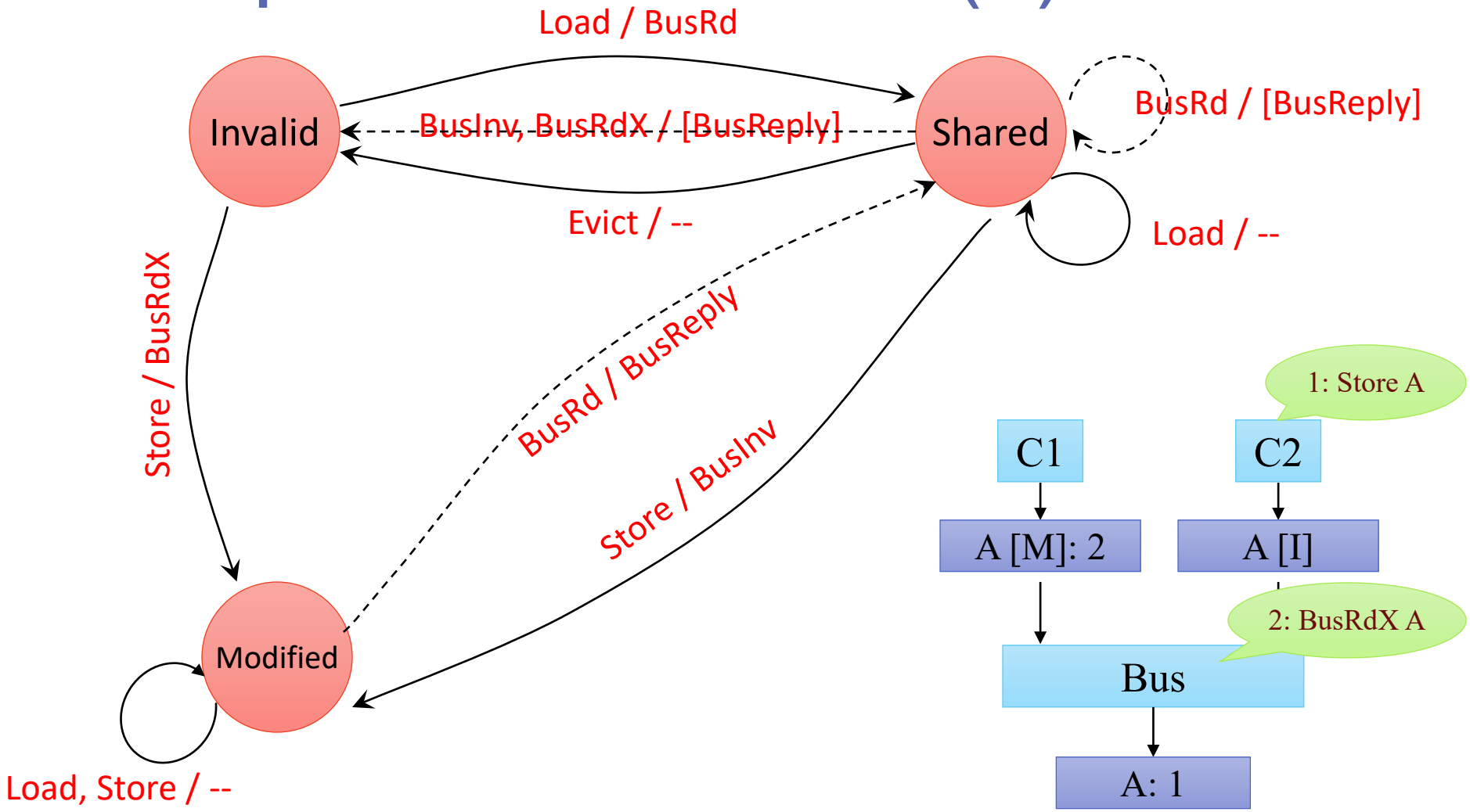
Simple MSI Protocol (7)



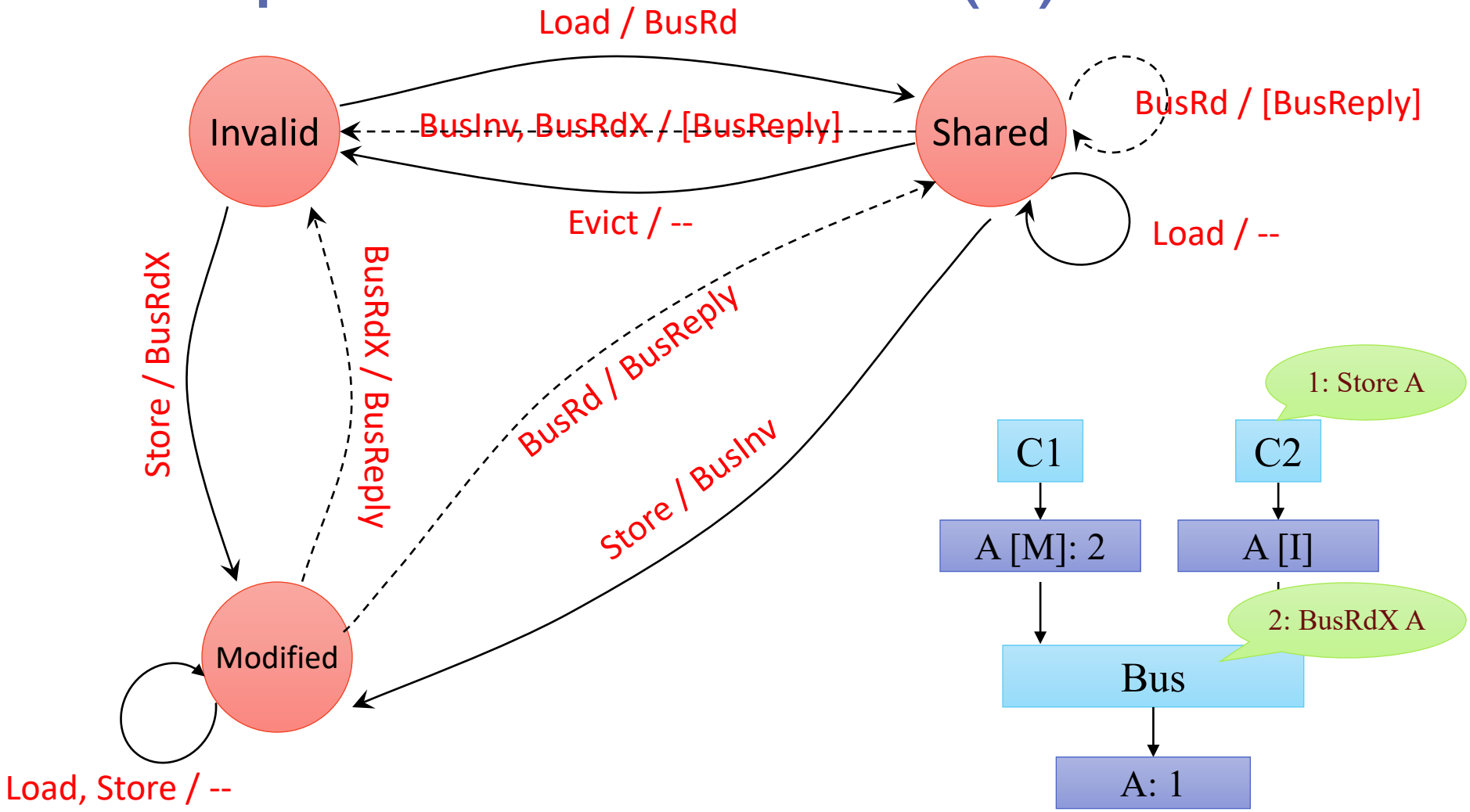
Simple MSI Protocol (7)



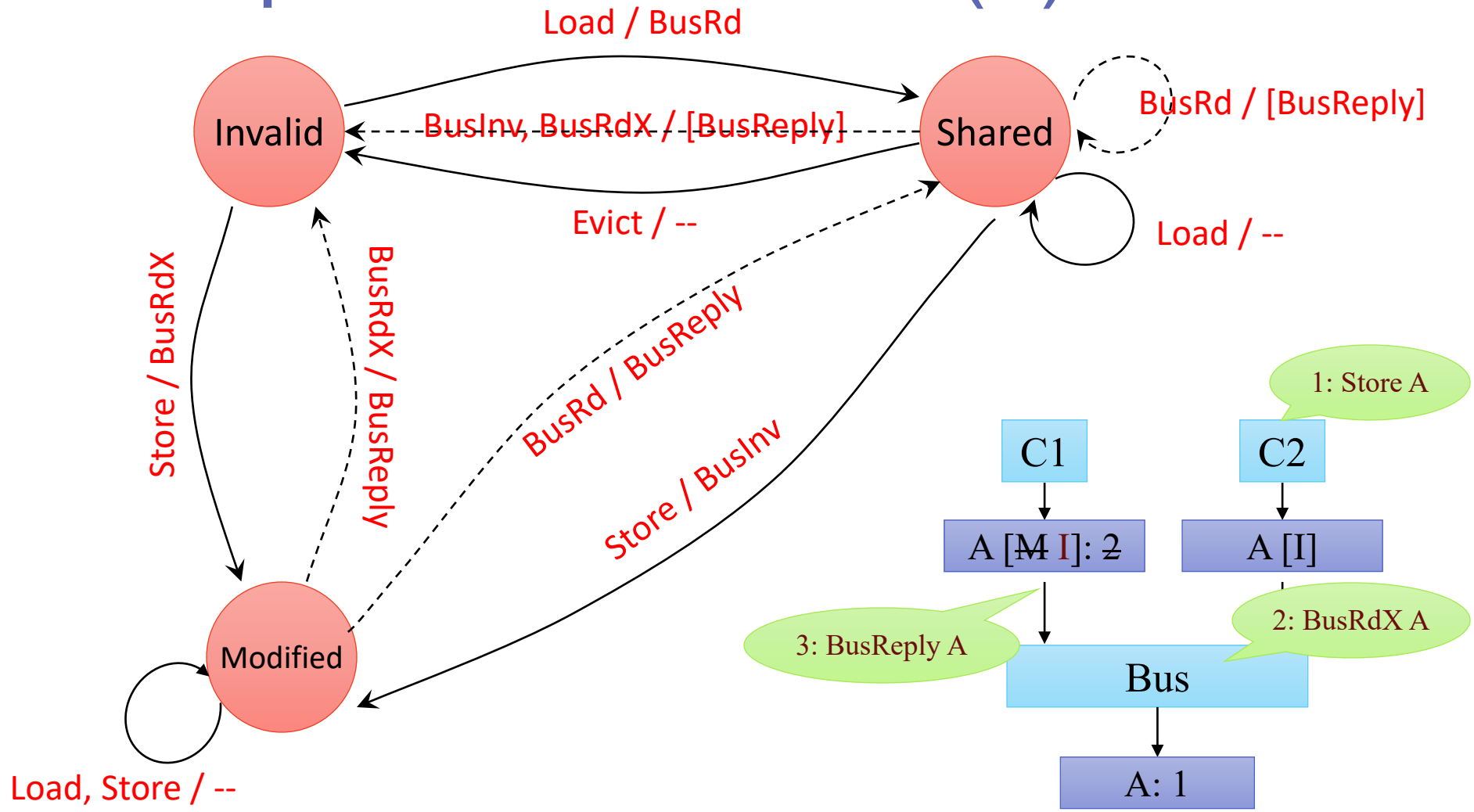
Simple MSI Protocol (7)



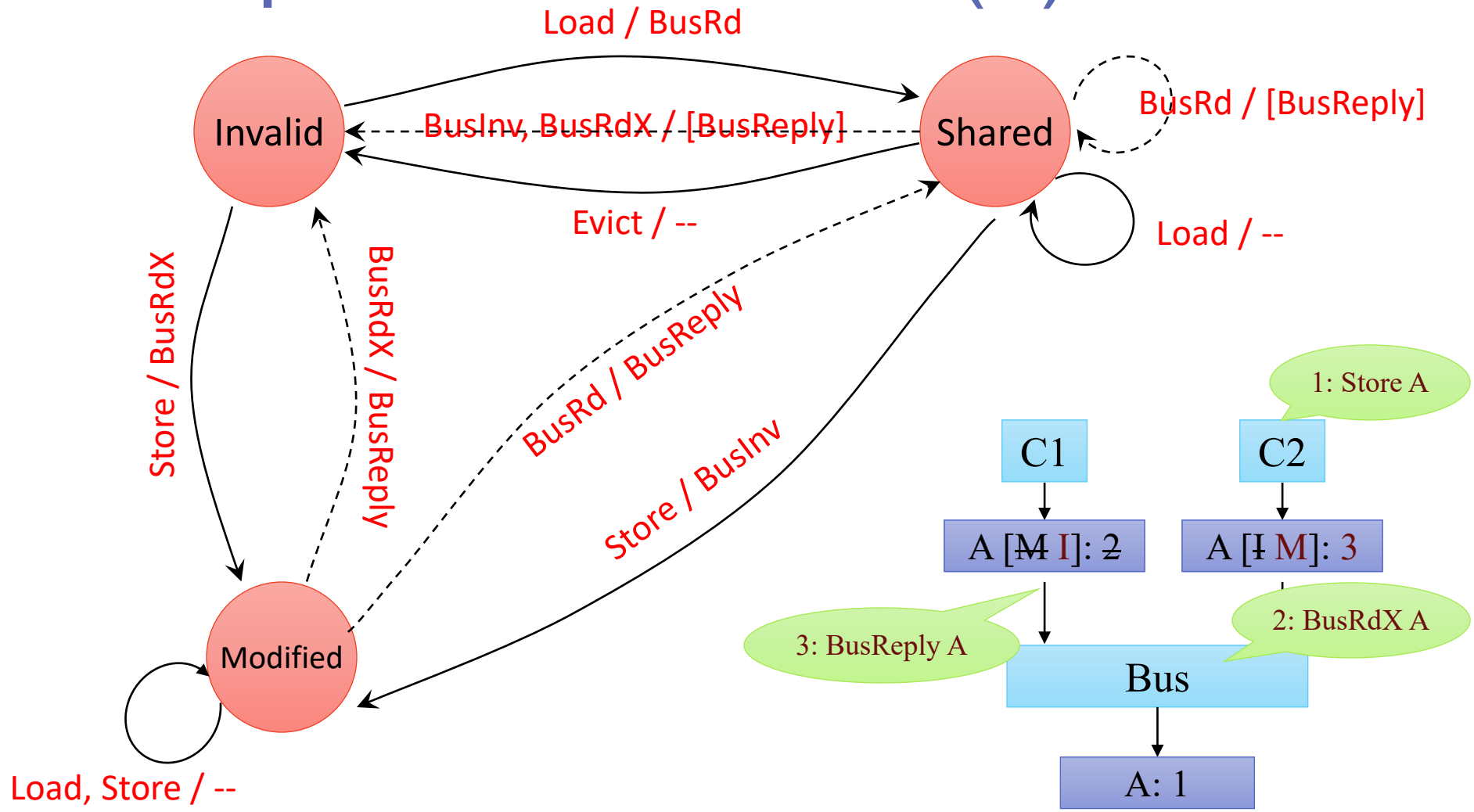
Simple MSI Protocol (7)



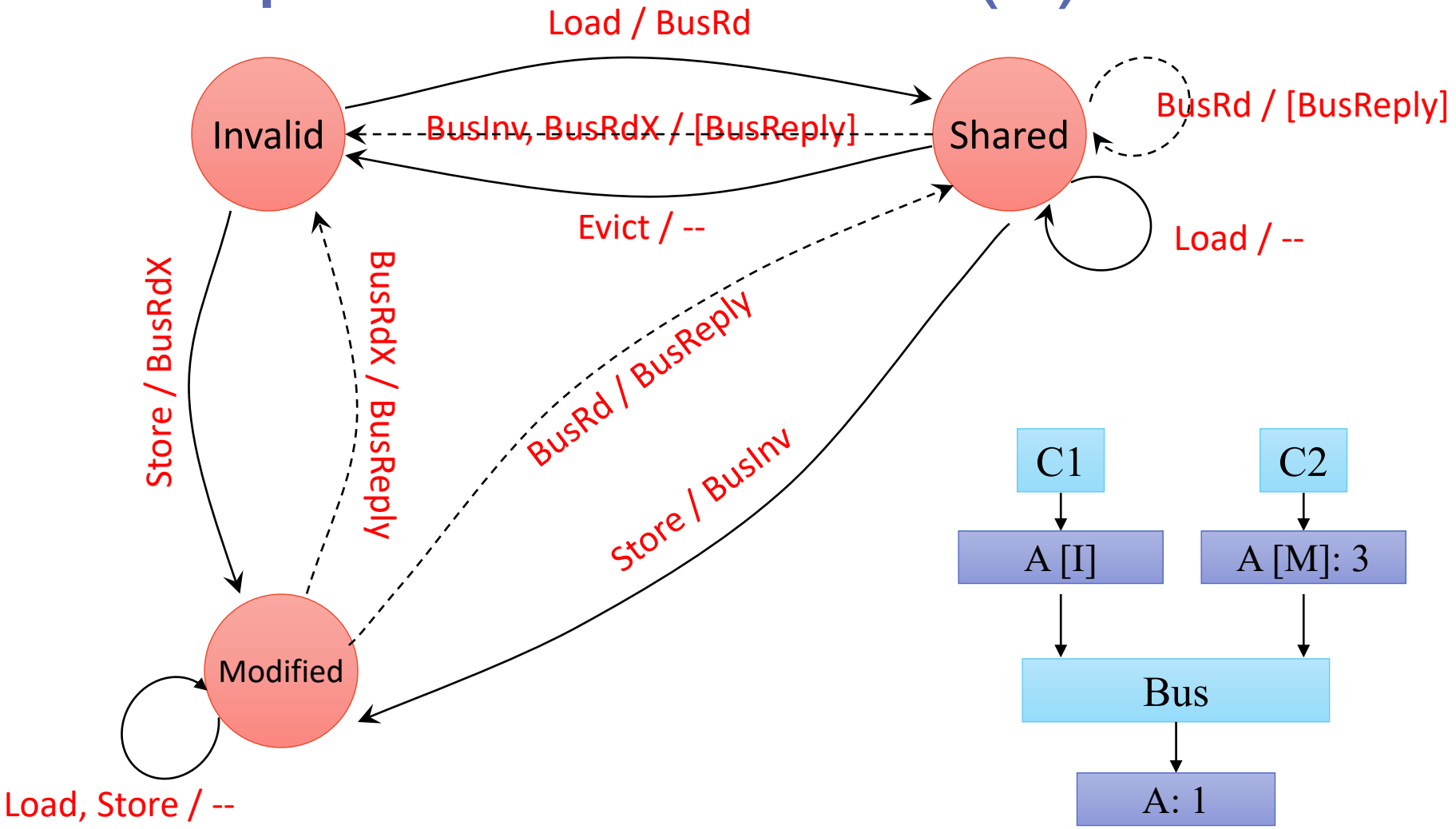
Simple MSI Protocol (7)



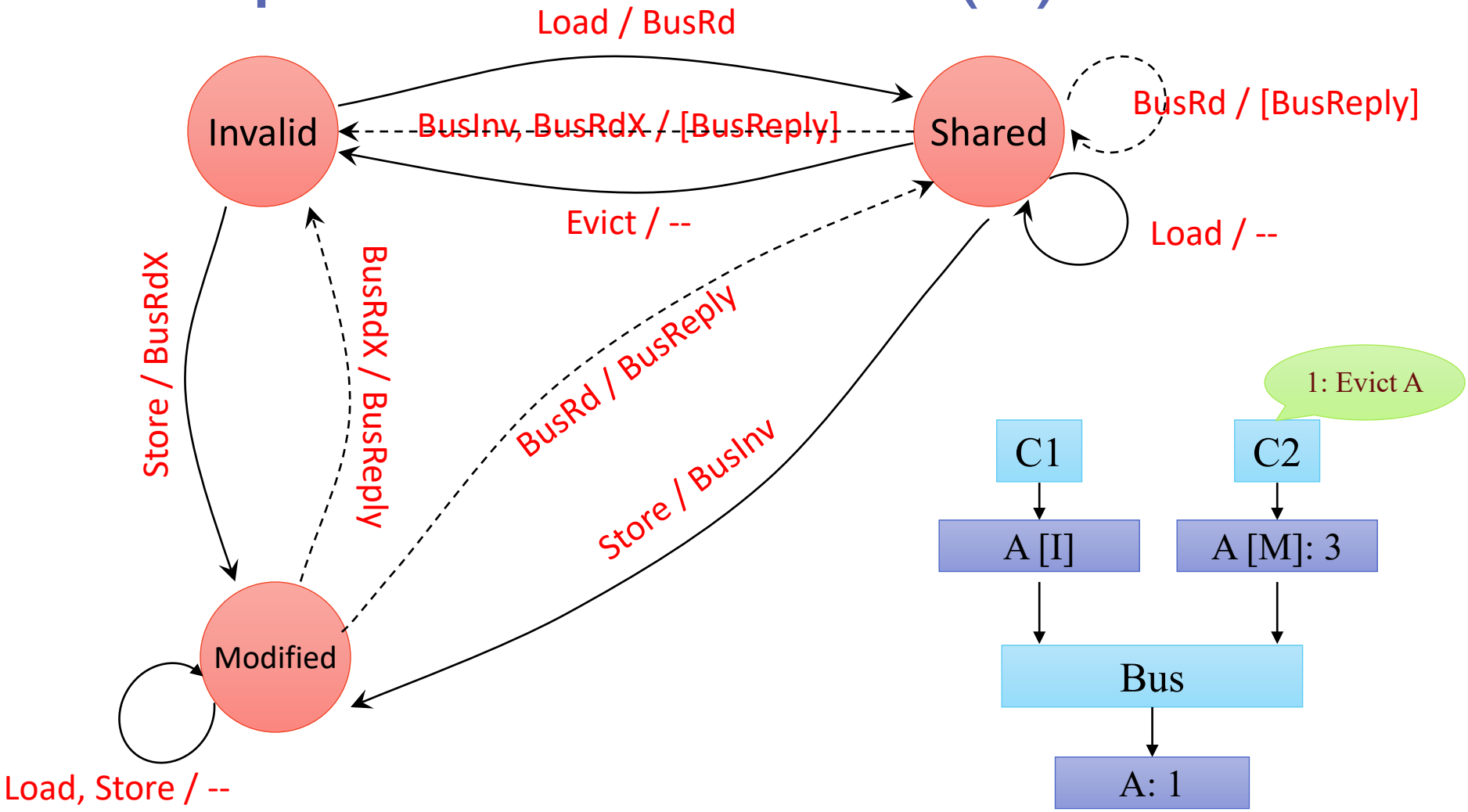
Simple MSI Protocol (7)



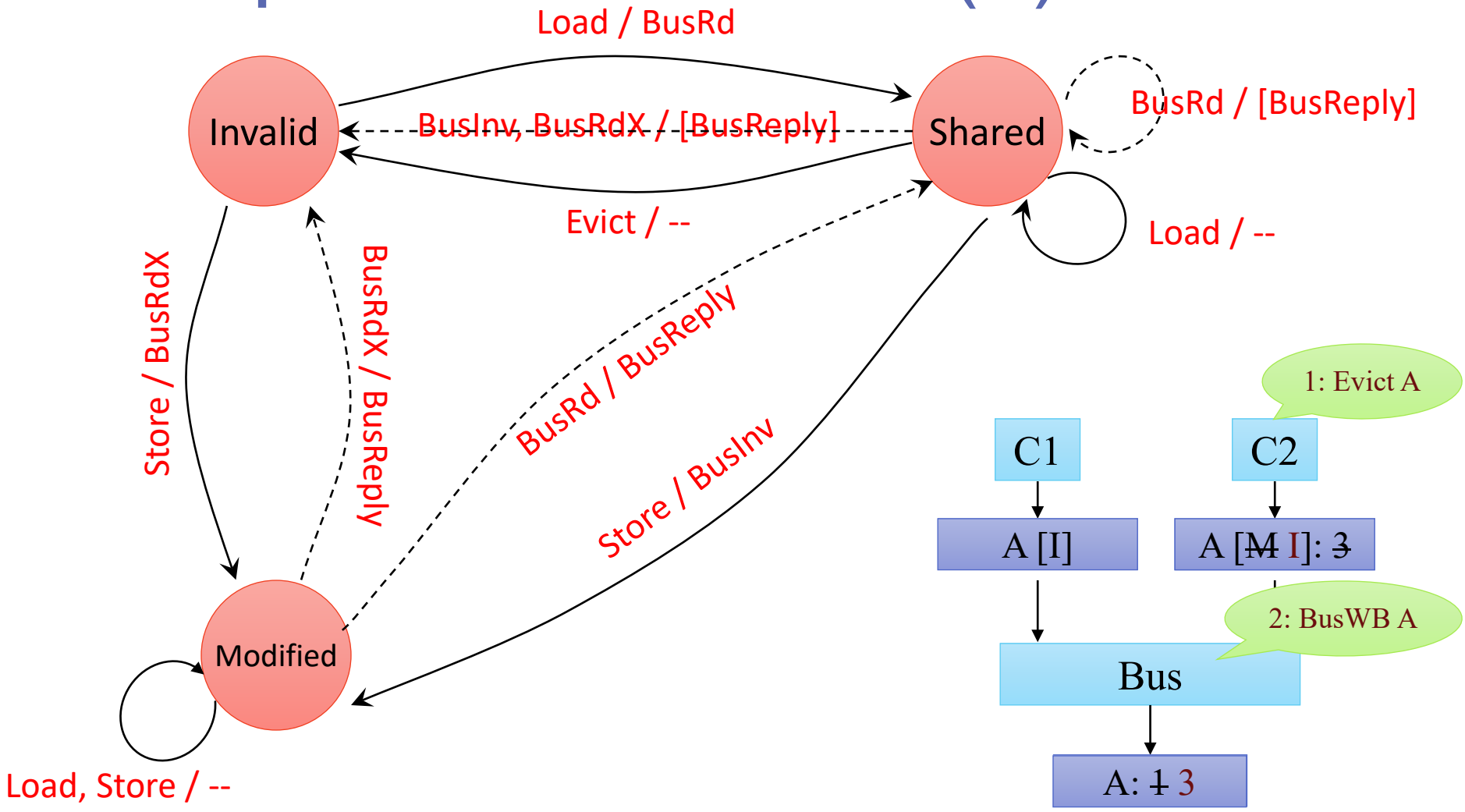
Simple MSI Protocol (8)



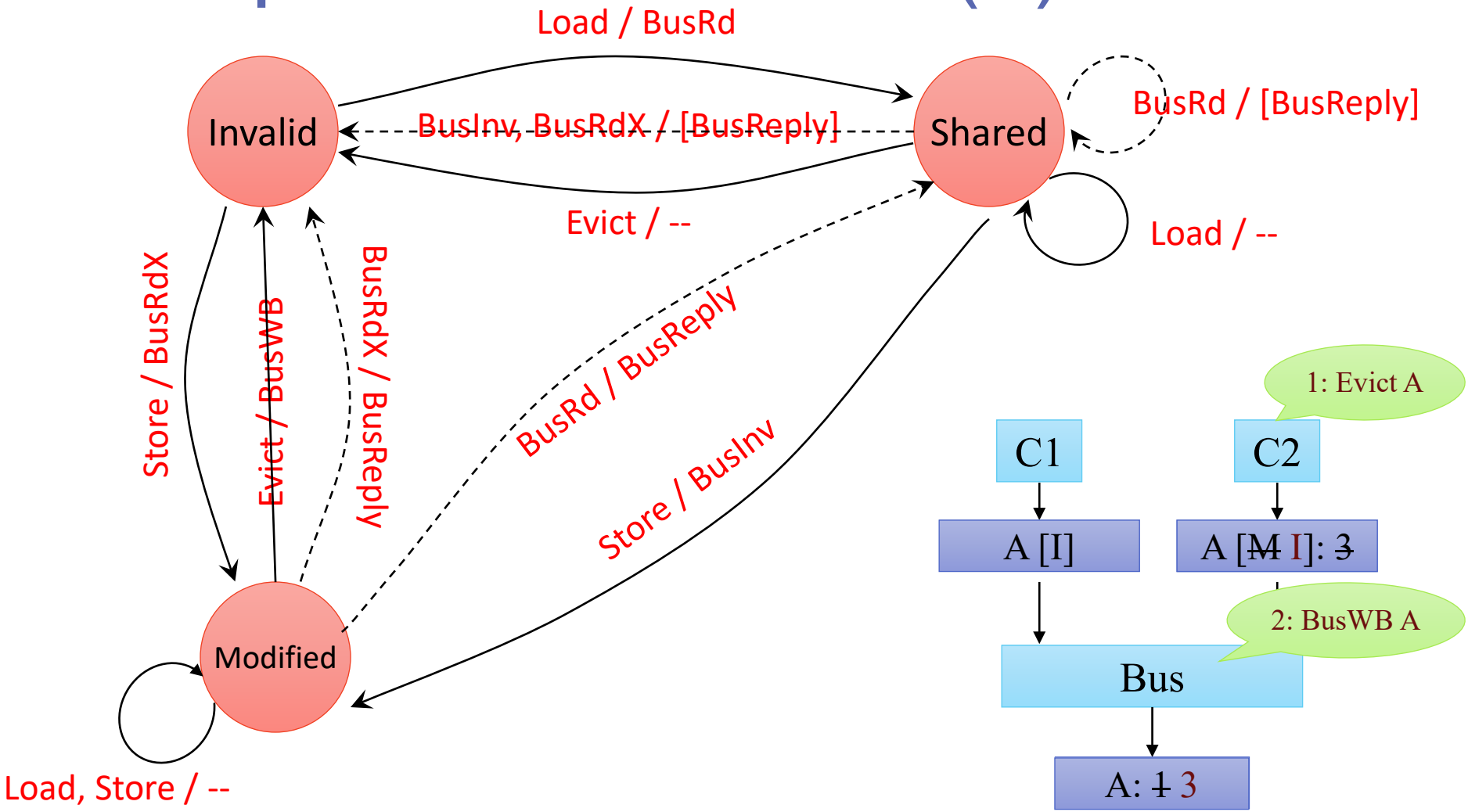
Simple MSI Protocol (8)



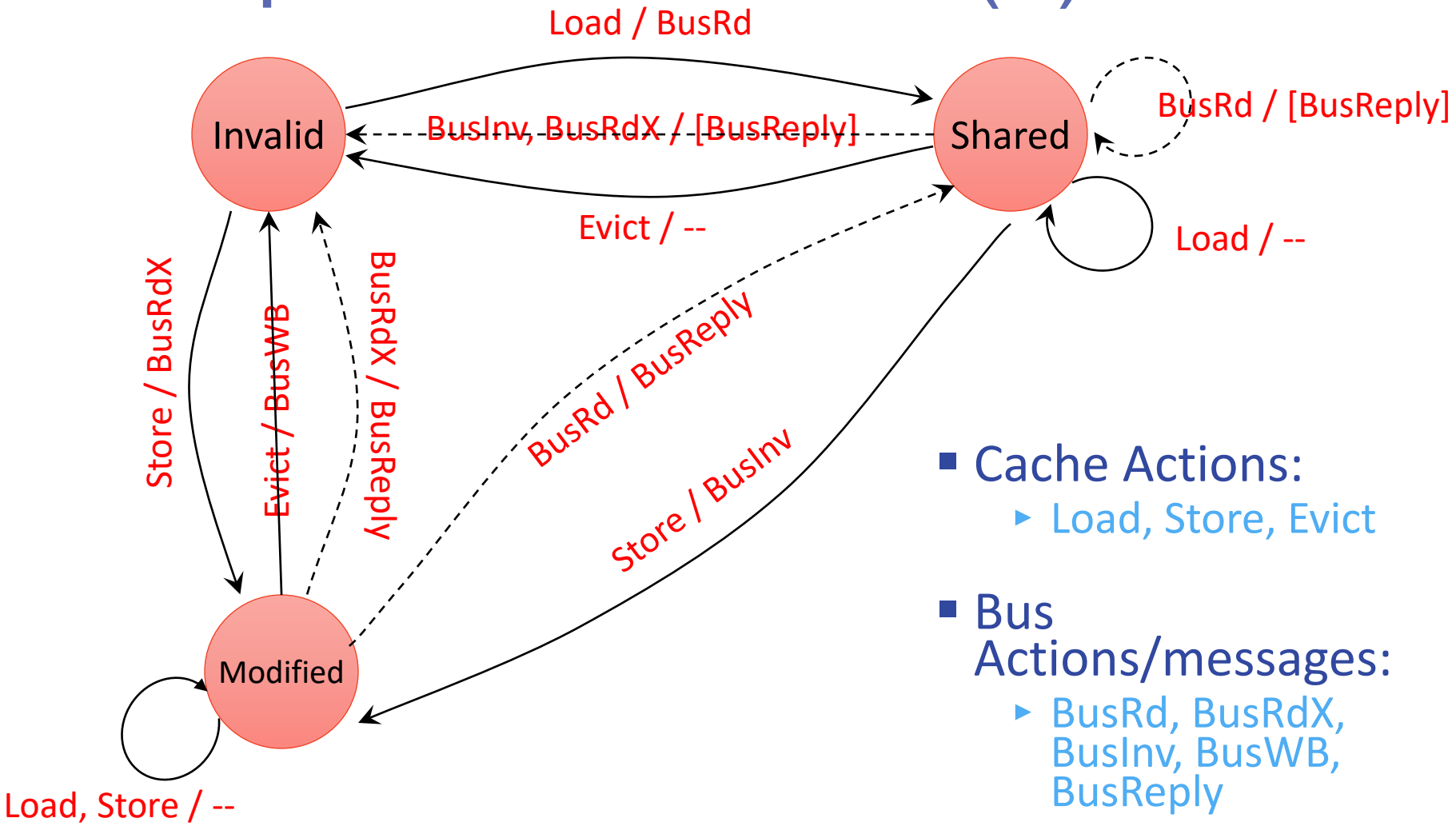
Simple MSI Protocol (8)



Simple MSI Protocol (8)

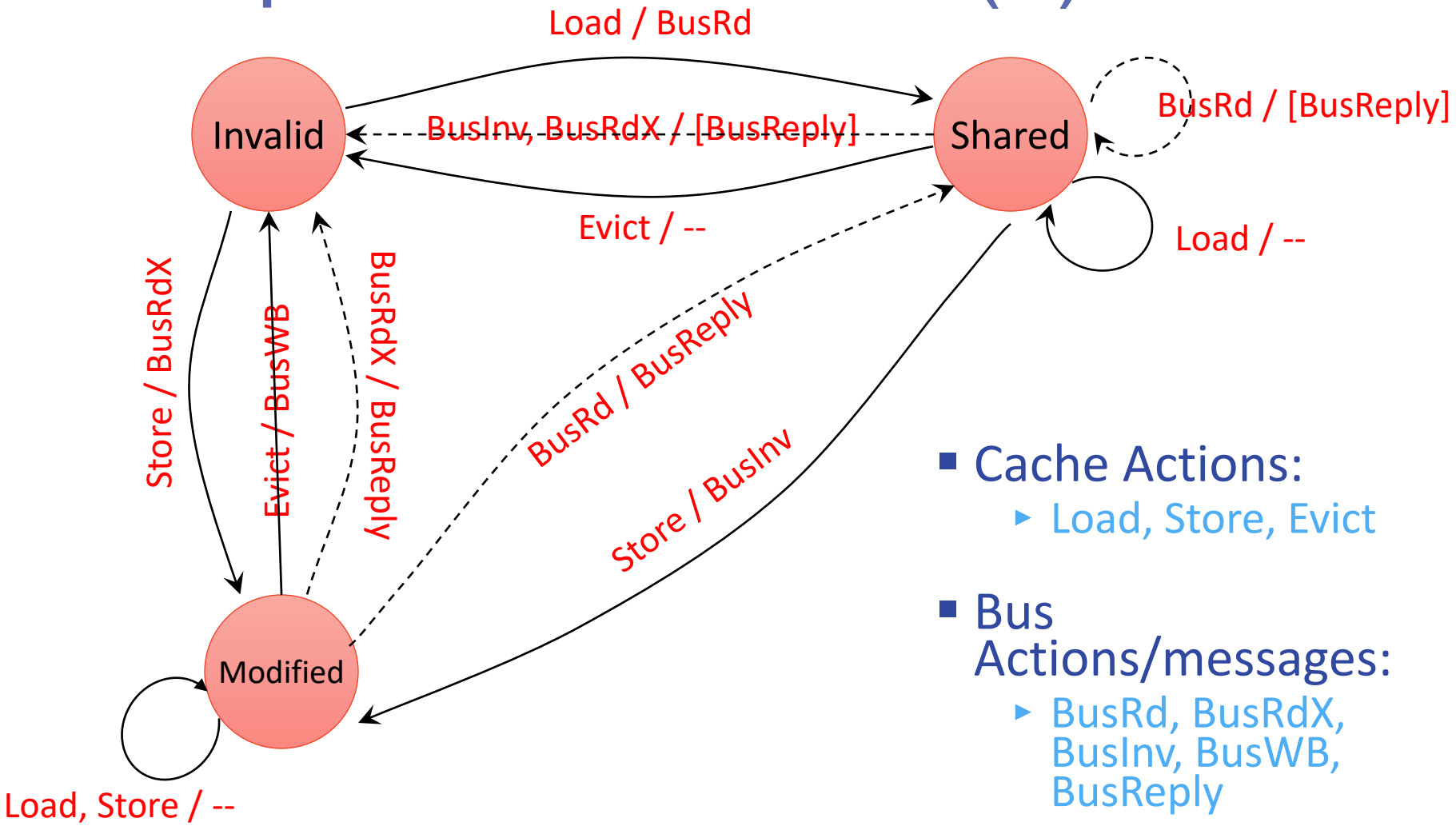


Simple MSI Protocol (9)



- Cache Actions:
  - Load, Store, Evict
- Bus Actions/messages:
  - BusRd, BusRdX, BusInv, BusWB, BusReply

Simple MSI Protocol (9)



- Cache Actions:
  - Load, Store, Evict
- Bus Actions/messages:
  - BusRd, BusRdX, BusInv, BusWB, BusReply

Each core's cache controller (i.e, at private caches) implements the same FSM