MANCHESTER 1824
The University of Manchester

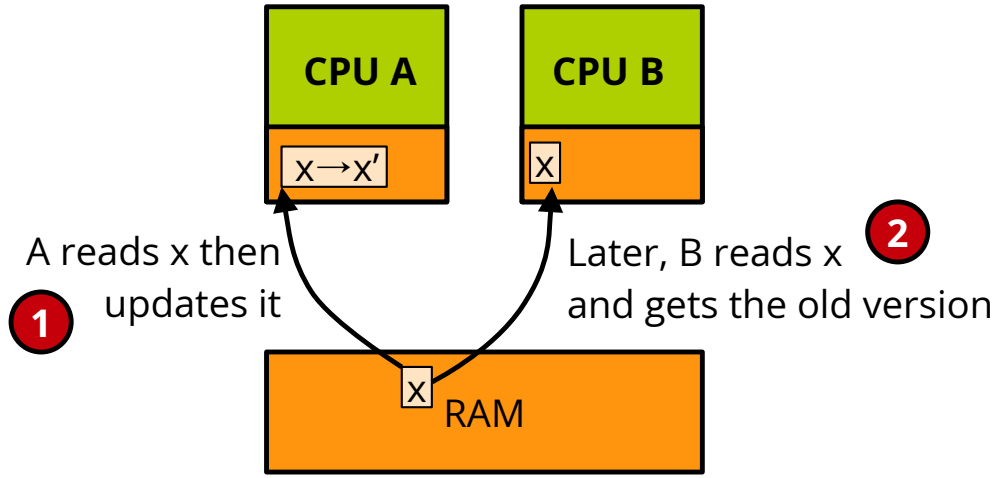
# COMP35112 Chip Multiprocessors

## Cache Coherence in Multiprocessors

Pierre Olivier

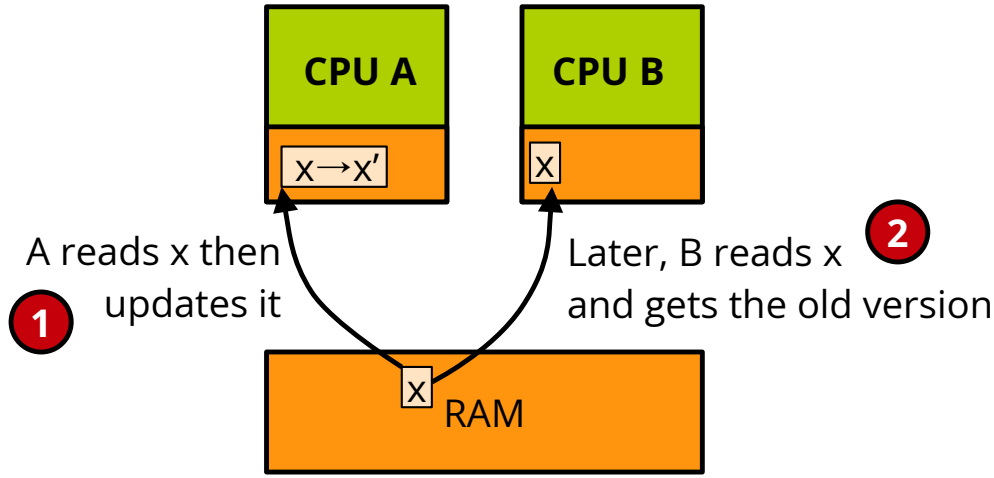
# Cache Coherence in Multiprocessors

- **Cache coherence**: avoid having multiple different copies of the same data in different caches of a shared memory multiprocessor



# Cache Coherence in Multiprocessors

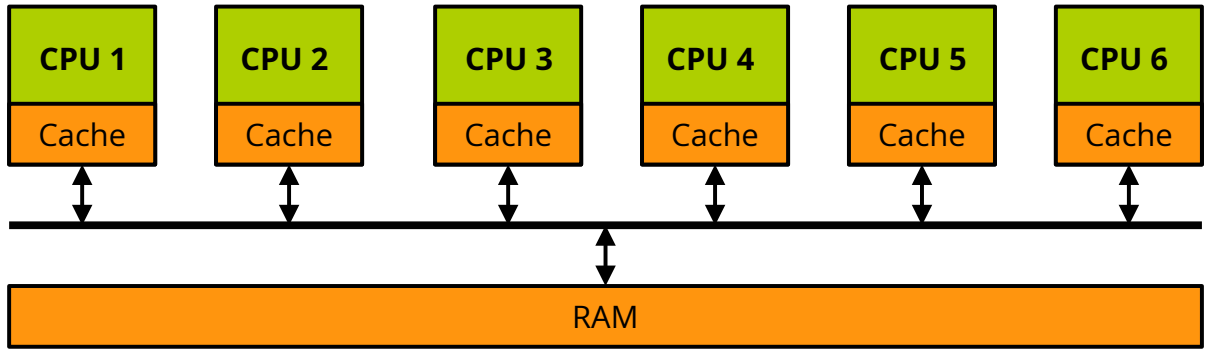
- **Cache coherence**: avoid having multiple different copies of the same data in different caches of a shared memory multiprocessor




- Need **cache to cache communication** for performance to avoid involving the slow memory

# Coping with Multiple Cores

- A bus is attached to every cache
- **Bus snooping**: hardware attached to each core's cache
  - Observes all transactions on the bus
  - Able to modify the cache independently of the core
- This hardware can take action on seeing pertinent transactions on the bus
- Another way to look at it:
  **a cache can send/receive messages to/from other caches**



# Cache States, MSI Protocol

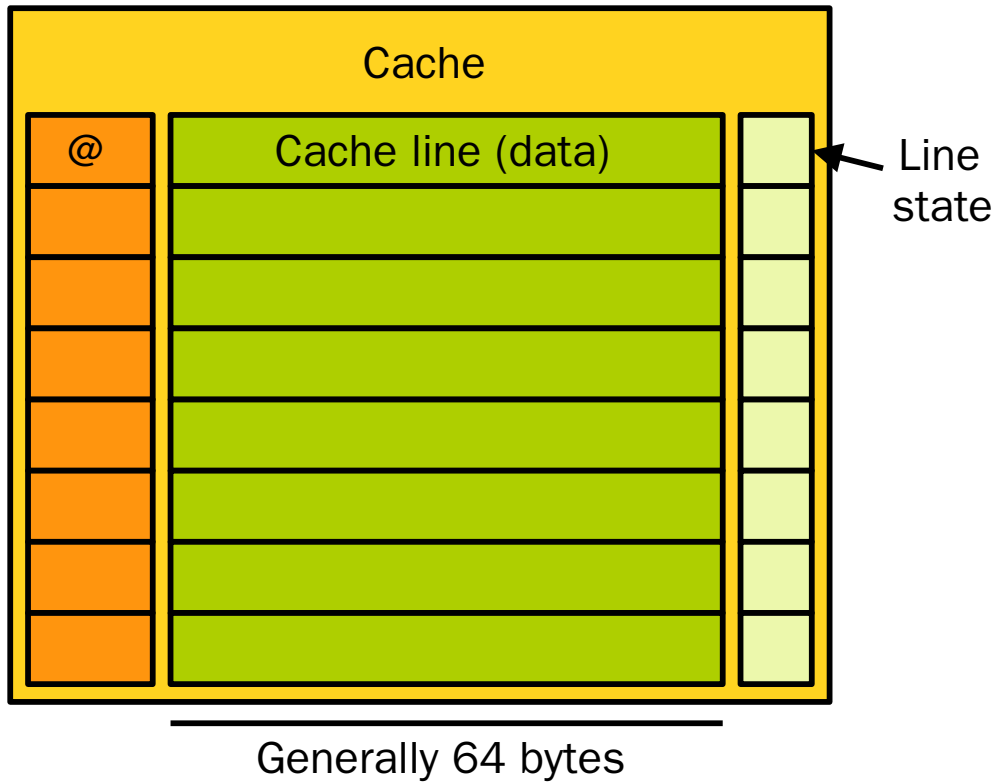
Cache has 2 control bits for each line it contains, indicating its state

# Cache States, MSI Protocol

Cache has 2 control bits for each line it contains, indicating its state

# Cache States, MSI Protocol

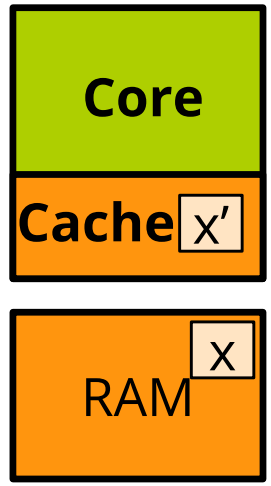
Cache has 2 control bits for each line it contains, indicating its state



# Cache States, MSI Protocol

Cache has 2 control bits for each line it contains, indicating its state

- **Modified** state: the cache line is valid and has been written to but the latest values have not been updated in memory yet
  - **A line can be in the modified state in at most 1 core**

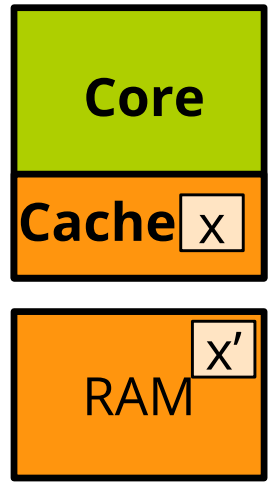



**Modified**

# Cache States, MSI Protocol

Cache has 2 control bits for each line it contains, indicating its state

- **Invalid**: there may be an address match on this line but the data is not valid
  - Load/stores should not be served from this cache
  - Must fetch line from memory or get it from another cache

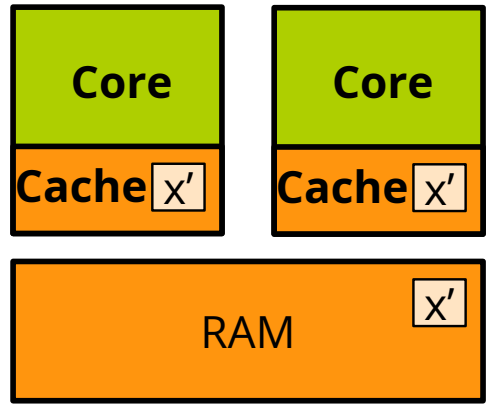



**Invalid**

# Cache States, MSI Protocol

Cache has 2 control bits for each line it contains, indicating its state

- **Shared**: not invalid and not modified
  - A valid cache entry exists and the line has the same values as main memory
  - **Several caches can have the same line in the shared state**

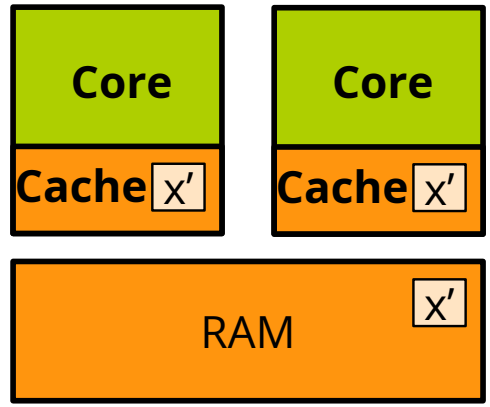



**Shared**

# Cache States, MSI Protocol

Cache has 2 control bits for each line it contains, indicating its state

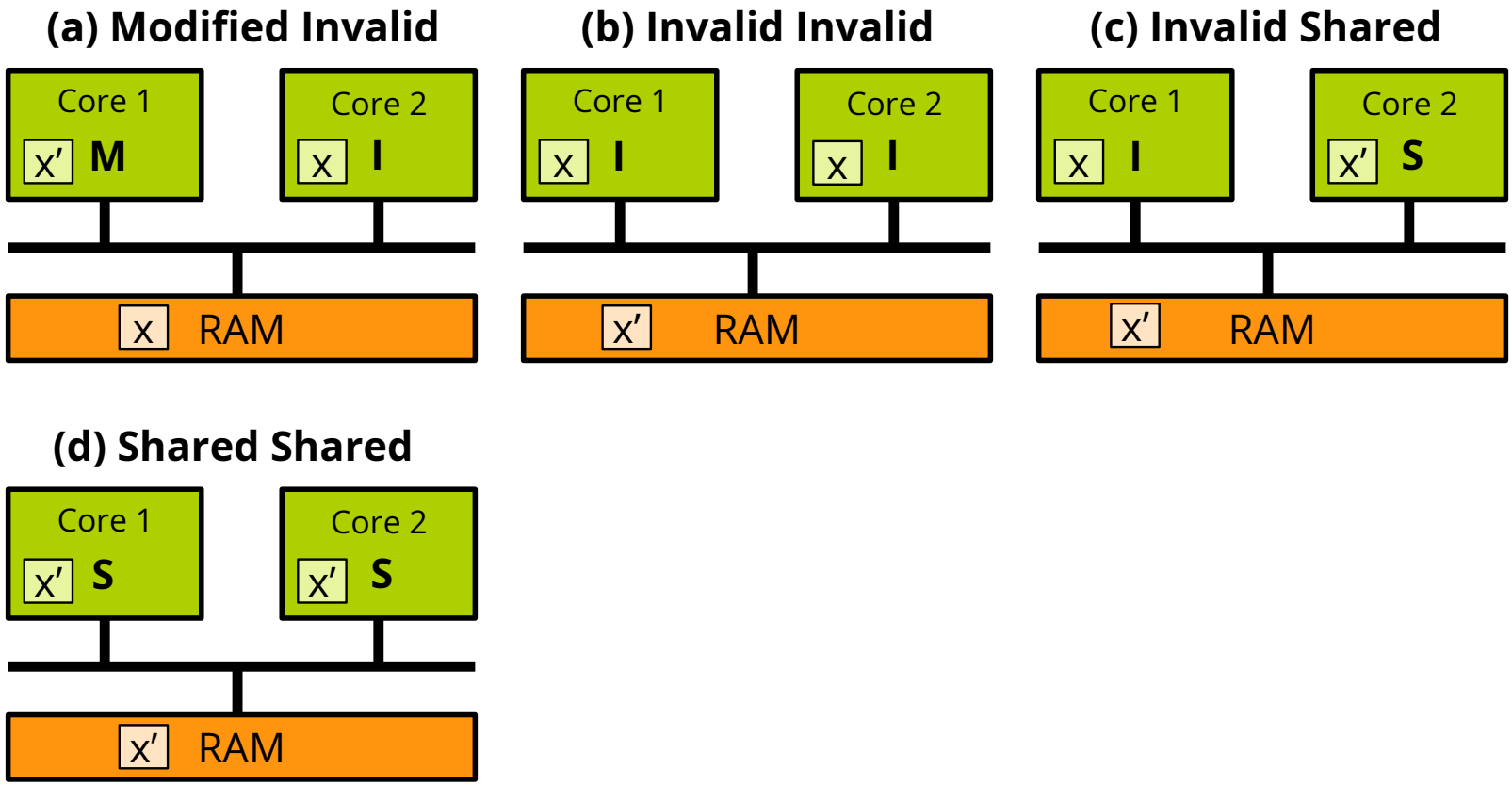
- **Shared**: not invalid and not modified
  - A valid cache entry exists and the line has the same values as main memory
  - **Several caches can have the same line in the shared state**



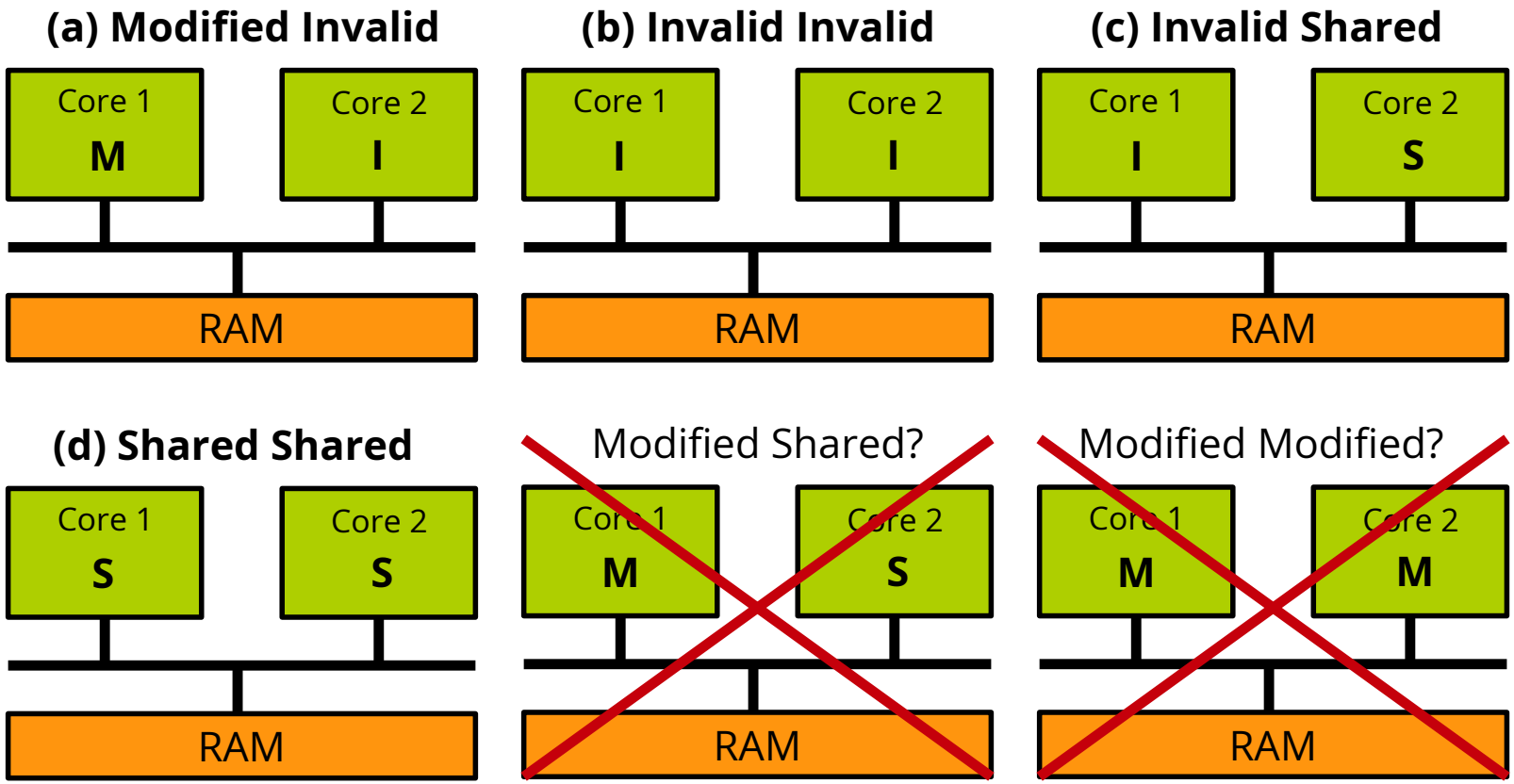

**Shared**

Modified/Shared/Invalid states and the transitions taken upon cache accesses by the core define the **MSI protocol**

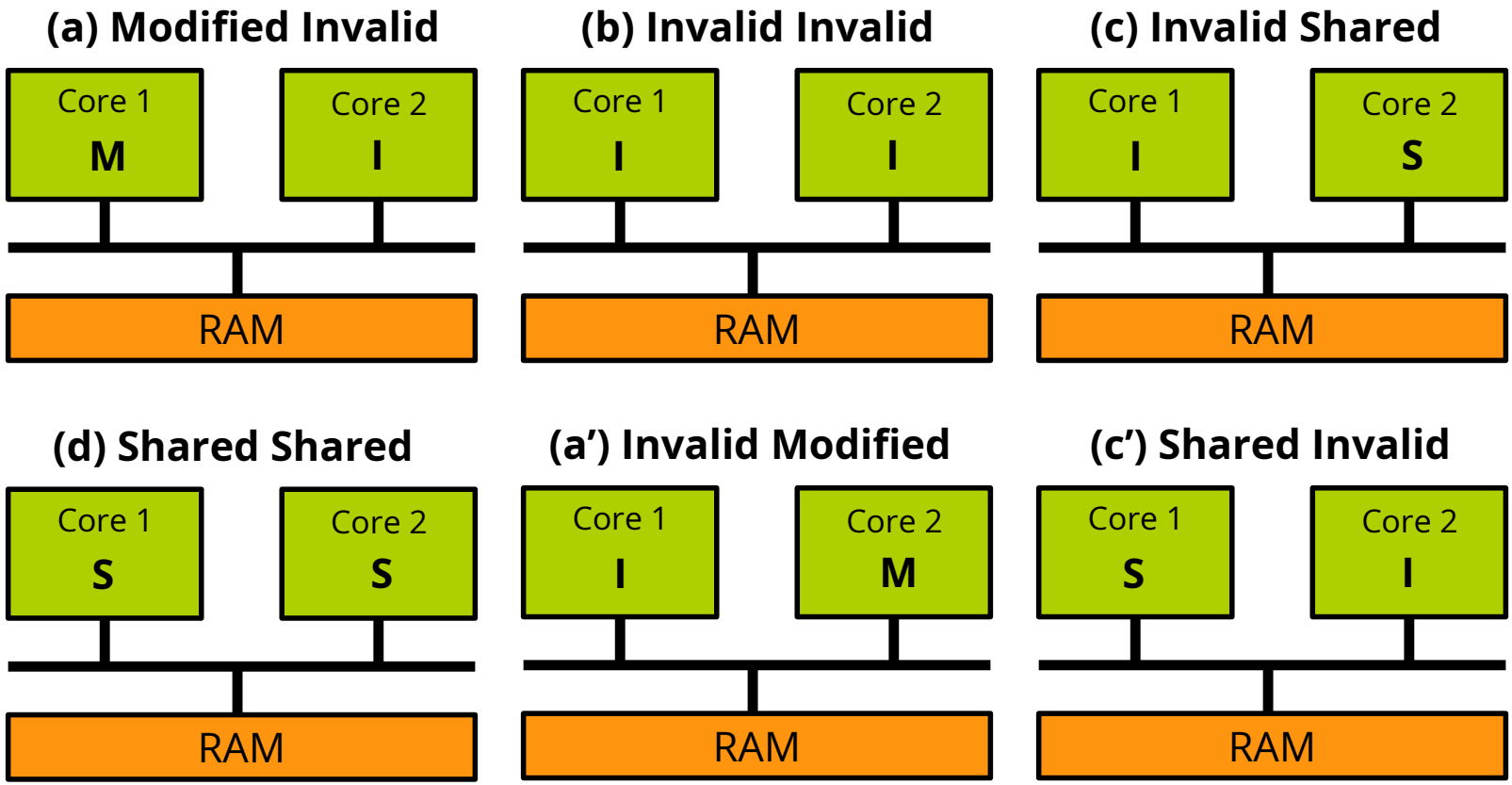
# Possible States for a Given Cache Line in a Dual-Core CPU



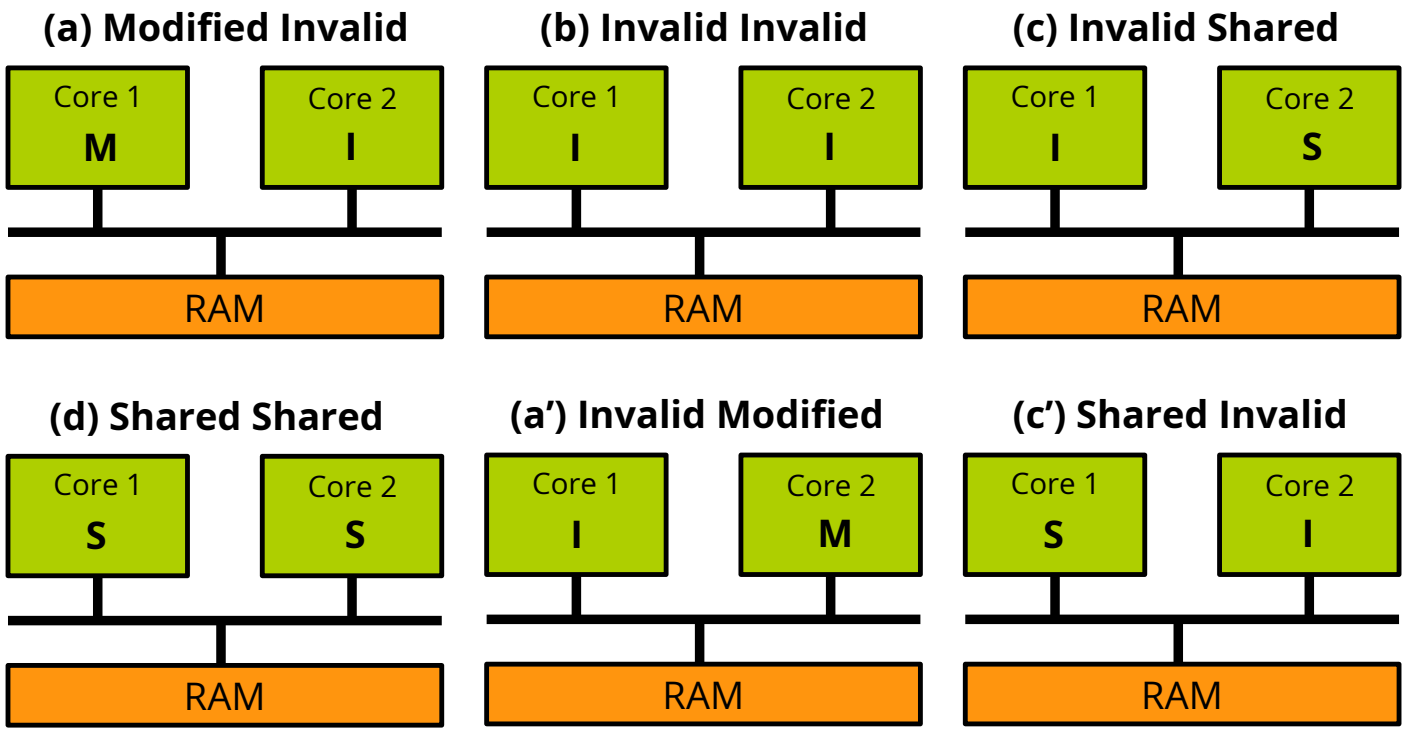
# Possible States for a Given Cache Line in a Dual-Core CPU



# Possible States for a Given Cache Line in a Dual-Core CPU



# State Transitions




- All of these are legal states. **let's study how read and writes on each core should affect these states**

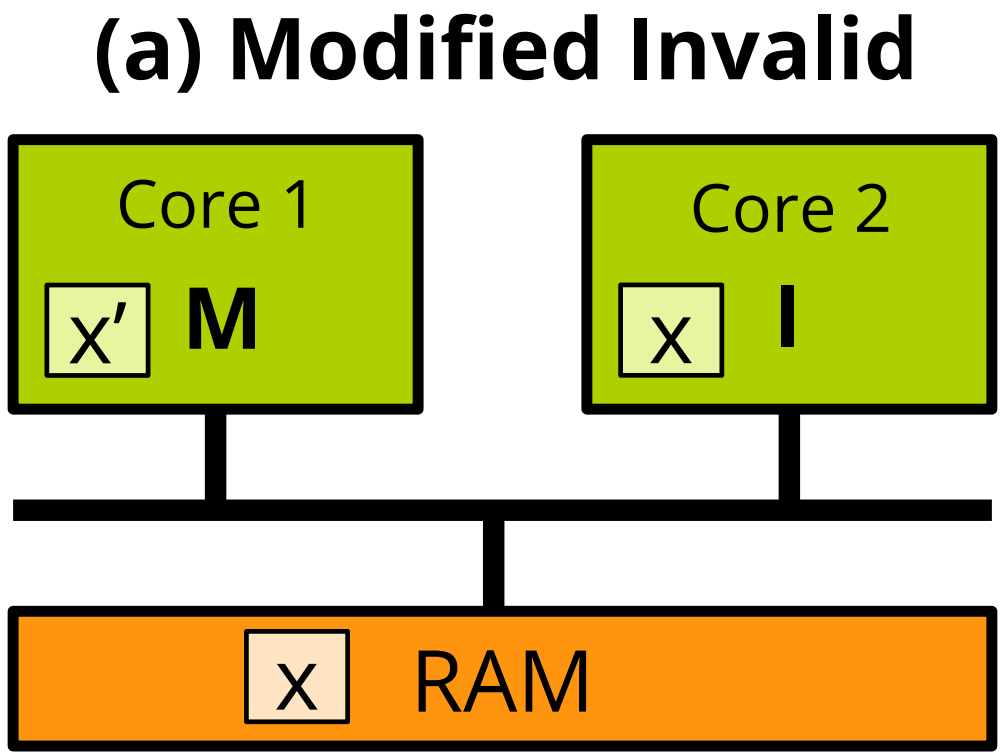
# State Transitions:

- State of a cache line may be changed when the core tries to read/write data in that line
- State transitions have 3 aspects:
  - What are the **messages** sent between caches:
    - *Read* messages: a cache requests a cache line from another
    - *Invalidate* messages: a cache asks another cache to invalidate one of its cache lines
  - Is there any **access made to main memory**
  - What are the **state changes**

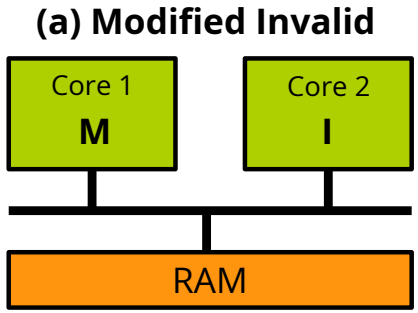
# State Transitions:

- State of a cache line may be changed when the core tries to read/write data in that line
- State transitions have 3 aspects:
  - What are the **messages** sent between caches:
    - *Read* messages: a cache requests a cache line from another
    - *Invalidate* messages: a cache asks another cache to invalidate one of its cache lines
  - Is there any **access made to main memory**
  - What are the **state changes**
- For a line in each valid state we identified, let's study what happens if the cores tries to read/write data in that line

# State Transitions from (a)

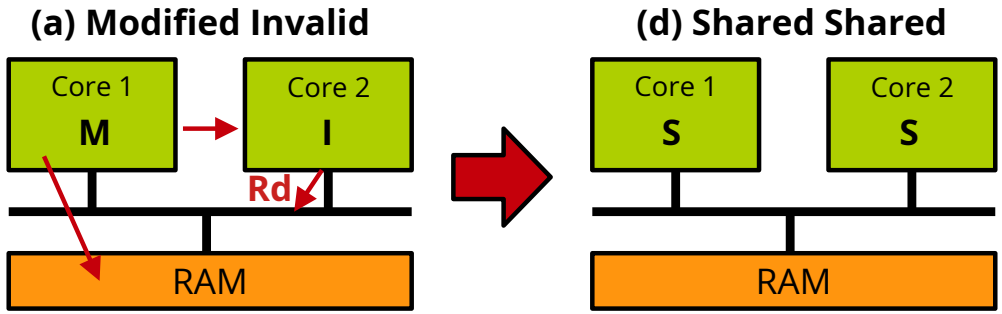


# State Transitions from (a)



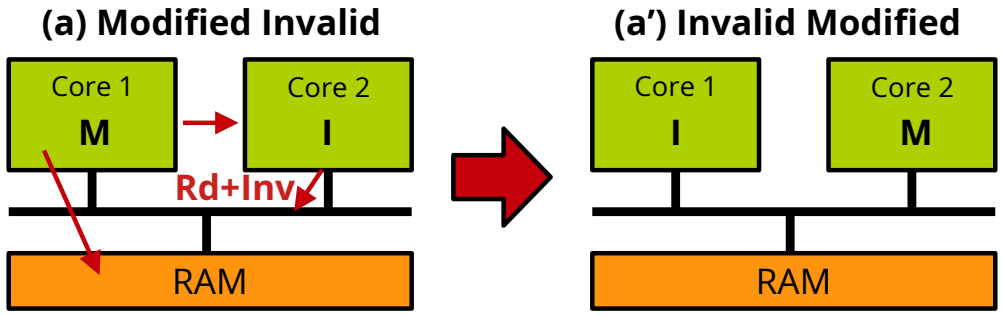

- **Read on core 1**: cache hit, served from cache
- **Write on core 1**: cache hit, served from cache

# State Transitions from (a)



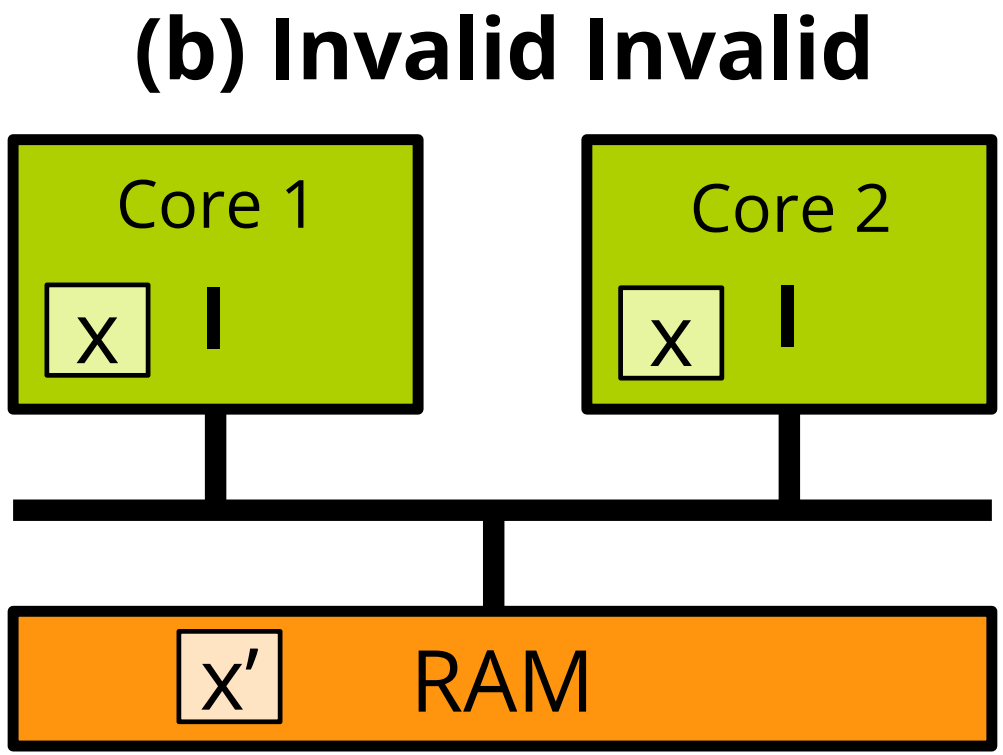

- **Read on core 1**: cache hit, served from cache
- **Write on core 1**: cache hit, served from cache
- **Read on core 2**:
  - 2 places read request on the bus, snooped by 1
  - 1 writes back to memory, goes to S state, and sends the data to 2 which goes to S state
  - Overall change to state (d): shared/shared

# State Transitions from (a)

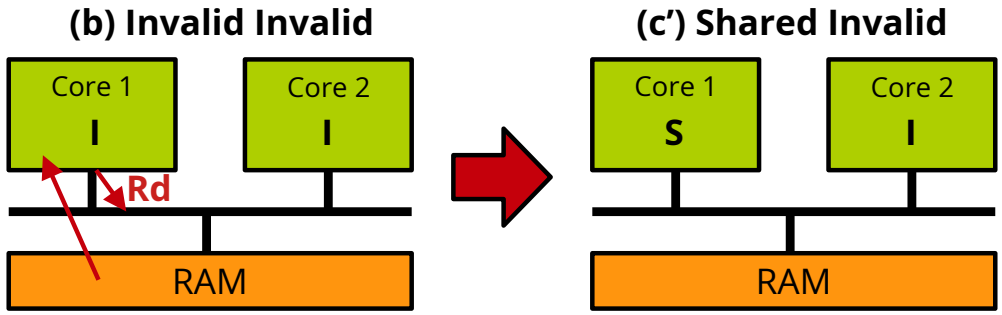



- **Write on core 2**:
  - 2 first needs to get the line **(write size < line size)**
  - 2 places read request on the bus
  - 1 snoops the request, sends to 2 and, as it is in M state, writes back to memory
  - 2 places invalidate request, core 1 switches to I
  - 2 writes in cache and switches to M
  - Overall state changes to (a'): invalid/modified

# State Transitions from (b)

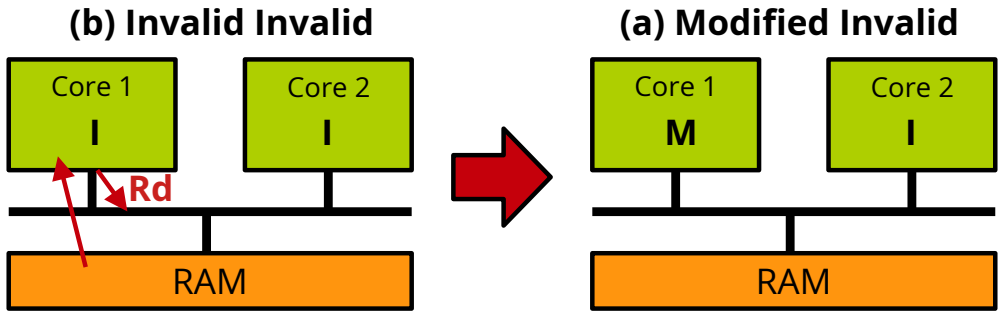


# State Transitions from (b)



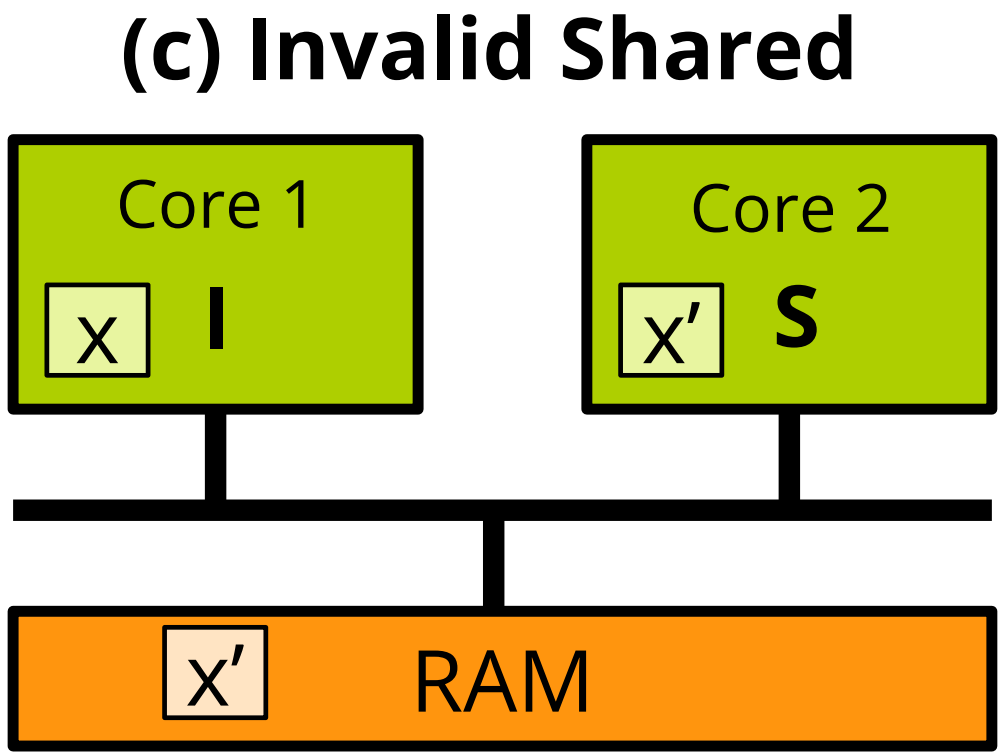

- **Read on core 1**
  - Cache miss, core 1 does not know the state of the line in other core, place read request on bus, nobody answers
  - Fetches from memory, switches to S
  - Go to (c')
- **Read on core 2** is similar by symmetry - goes to state (c): invalid/shared

# State Transitions from (b)

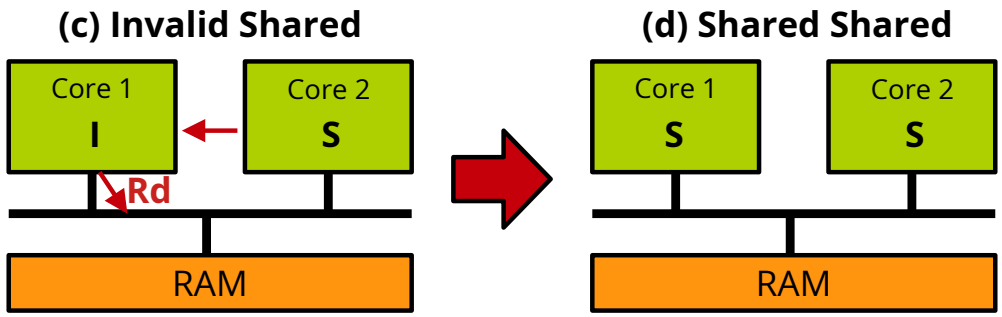



- **Write on core 1**:
  - Core 1 does not know the state of the line in other cores, places read request on the bus, nobody answers
  - Fetches from memory and performs write in cache, switches to M
  - State goes to (a)
- **Write on core 2** is similar by symmetry - goes to state (a'): invalid/modified

# State Transitions from (c)

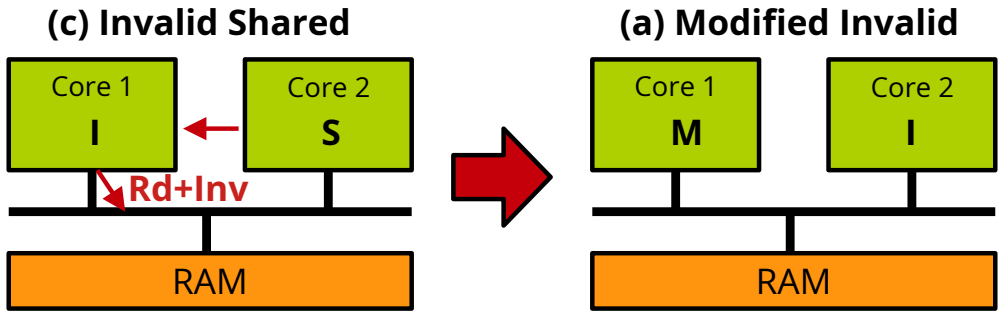


# State Transitions from (c)



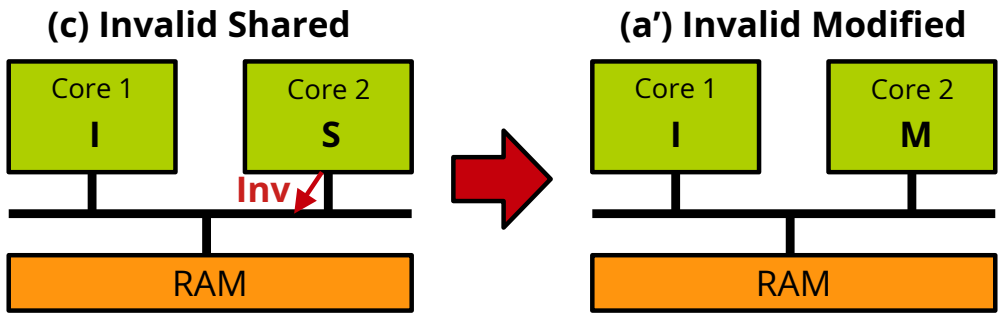

- **Read on core 1:**
  - 1 places request on the bus, gets snooped by 2
  - 2 sends value to 1, which goes to S
  - Overall state goes to (d): shared/shared
- **Read on core 2**:
  - Cache hit, served from the cache, stays in (c): invalid/shared

# State Transitions from (c)



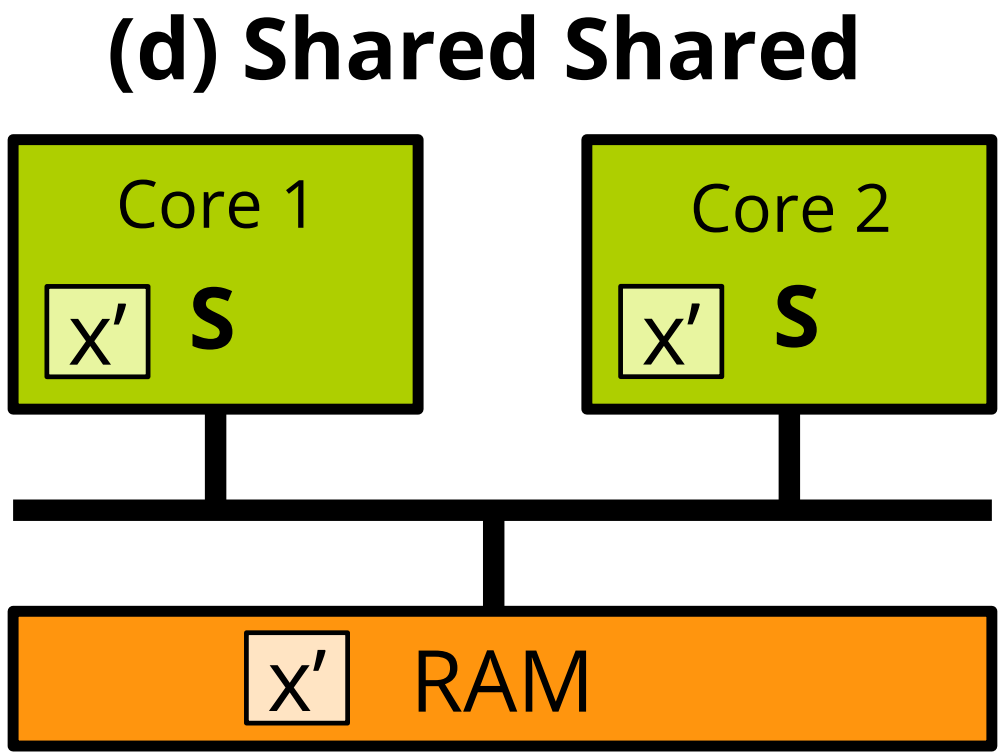

- **Write on core 1**:
  - 1 places a read request on the bus, snooped by 2
  - 2 sends line to 1, it was in S so no need for writeback
  - 1 places invalidate request on the bus, 2 goes to I
  - 1 performs write in cache and goes to M
  - Overall state goes to (a): modified/invalid

# State Transitions from (c)

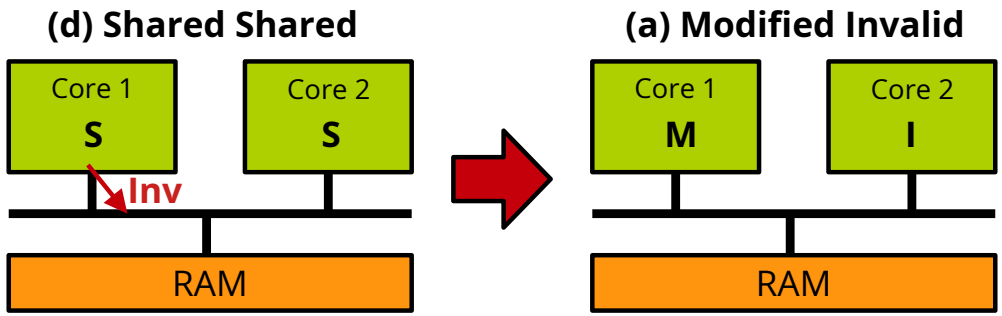



- **Write on core 2**:
  - 2 does not know the line state in other caches, places invalidate request on the bus
  - 2 performs write in cache and goes to M
  - Overall state goes to (a'): invalid/modified

# State Transitions from (d)



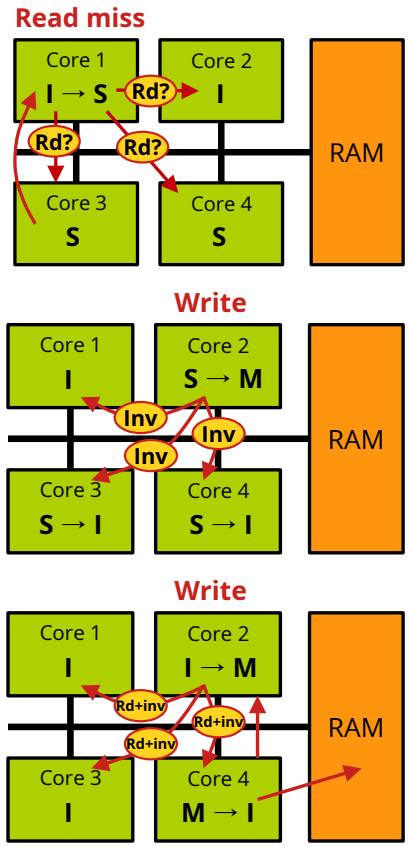
# State Transitions from (d)




- **Read on core 1 or 2**: cache hit, served from the cache
- **Write on core 1**:
  - 1 places invalidate request on the bus, get snooped by 2
  - 2 goes to I, it was in S so no need for writeback
  - 1 performs write in cache, goes to M
  - Overall state goes to (a): modified/invalid
- **Write on core 2**: symmetry, state goes to (a'): invalid/modified

# Beyond Two Cores

- Extension beyond 2 cores:
  - Snoopy bus messages are **broadcasted to all cores**
  - Any core with a valid value can respond to a read request
  - Upon receiving an invalidate request:
    - Any core in S invalidates without writeback
    - A core in M writes back then invalidates



# Write-Invalidate vs. Write-Update

2 types of snooping protocols:

- **Write-invalidate**:
  - When a core updates a cache line, other copies of that line in other caches are **invalidated**
  - Future accesses on the other copies will require fetching the updated line from memory/other caches
  - Most widespread protocol, used in MSI (this lecture), MESI, MOESI (next video)

# Write-Invalidate vs. Write-Update

2 types of snooping protocols:

- **Write-invalidate**:
  - When a core updates a cache line, other copies of that line in other caches are **invalidated**
  - Future accesses on the other copies will require fetching the updated line from memory/other caches
  - Most widespread protocol, used in MSI (this lecture), MESI, MOESI (next video)
- **Write-update**:
  - When a core updates a cache line, the modification is broadcast to copies of that line in other caches: they are **updated**
  - Leads to higher bus traffic vs. write-invalidate
  - Example protocols: Dragon, Firefly

# Major Implication

- With cache coherence all cores must always see exactly the same state of a location in memory
- If one core writes and broadcasts invalidate:
  - No other core must be able to perform a read/write to that location as though they haven't seen the invalidate
- **All cores must see the invalidate at the same time, i.e. within the same bus cycle**

# Major Implication

- With cache coherence all cores must always see exactly the same state of a location in memory
- If one core writes and broadcasts invalidate:
  - No other core must be able to perform a read/write to that location as though they haven't seen the invalidate
- **All cores must see the invalidate at the same time, i.e. within the same bus cycle**
  - As we connect more cores this becomes more and more difficult
  - **The coherence protocol is a major limitation to the number of cores that can be supported**

# Summary

- **Cache coherence** is necessary in shared memory multiprocessors, for cores to have a consistent view on memory
- Simple MSI protocol
  - **Modified**/**Shared**/**Invalid** states
  - Associated transitions upon read/writes from cores
    - Invalidate and line read requests on the interconnect
    - Read/write from/to main memory
    - State changes
- Bus-based CC protocol is limiting the number of cores supported
- Next:
  - MSI's optimisations: MESI and MOESI
  - Directory-based coherence protocol