Parallel processors

And a coherent view of shared memory

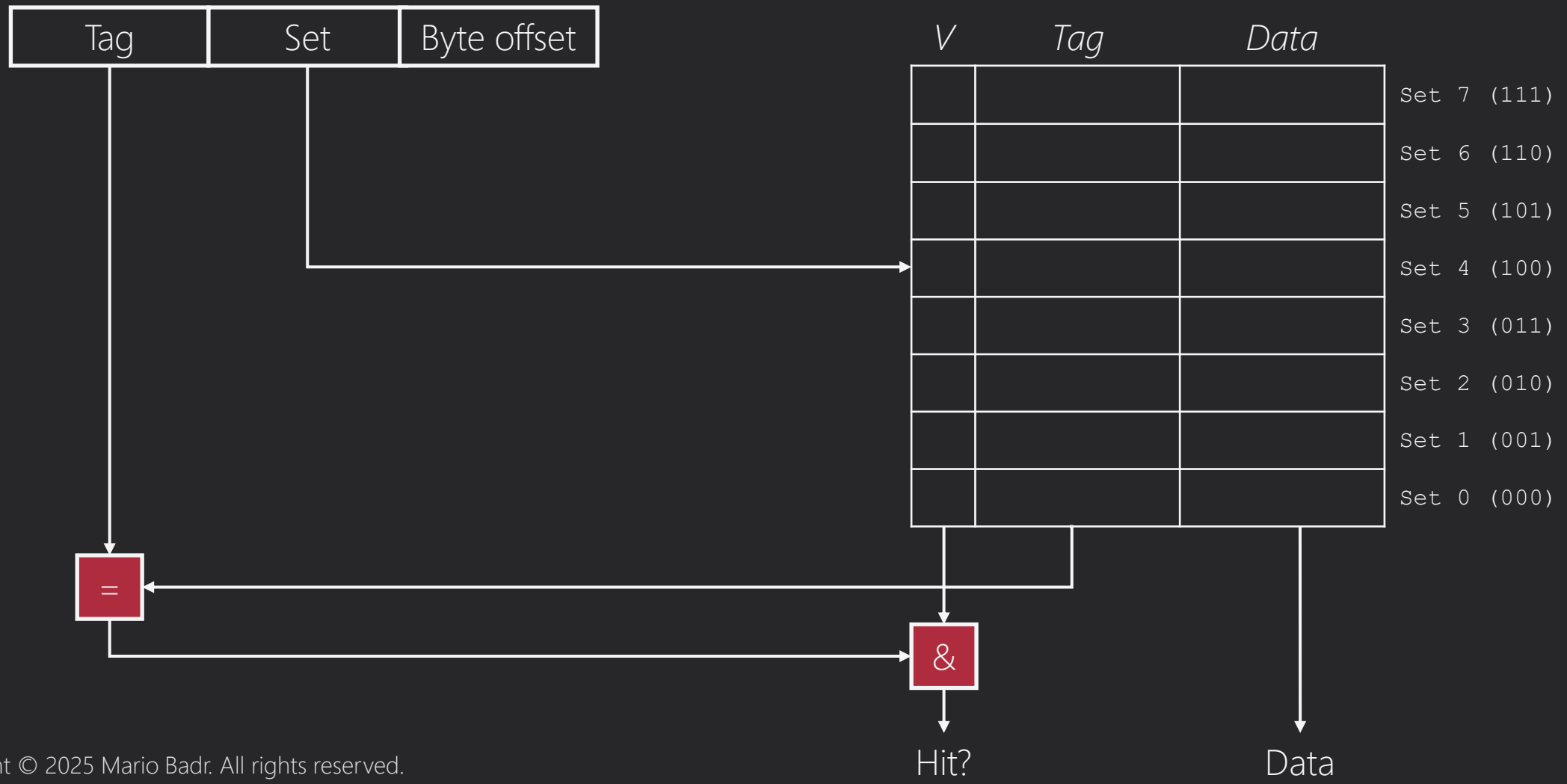
Introduction

Shared bus

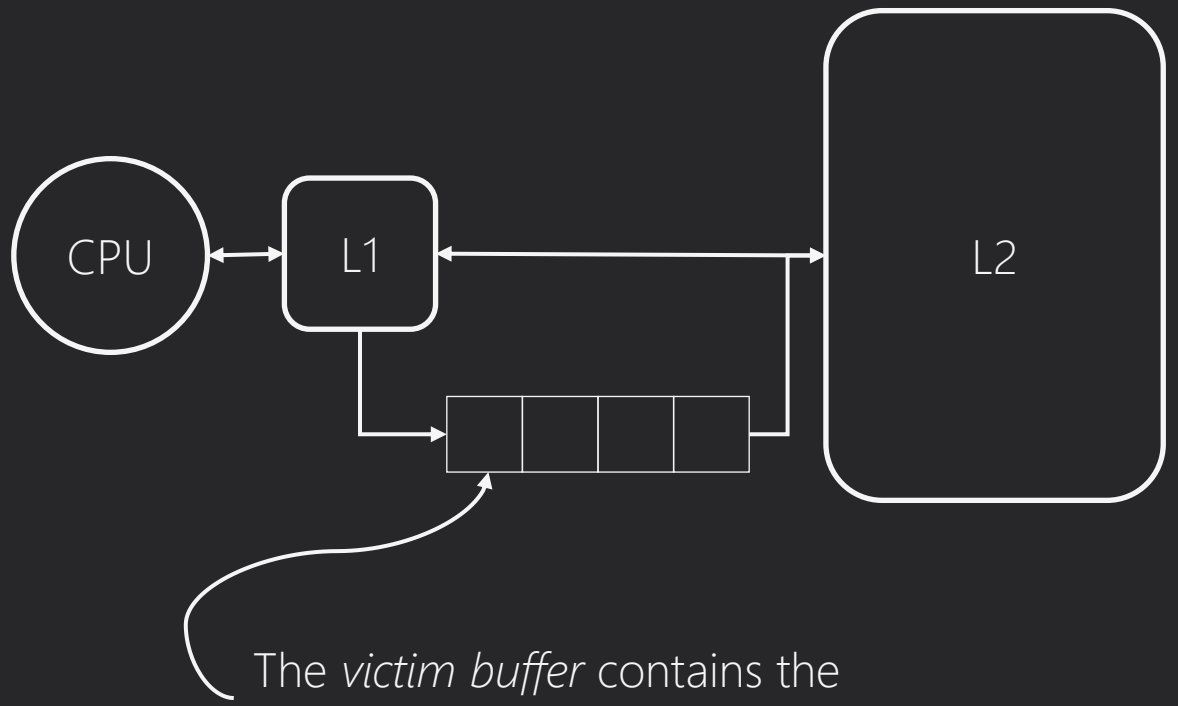
Cache coherence

Conclusion

8-word direct mapped cache (b=1)

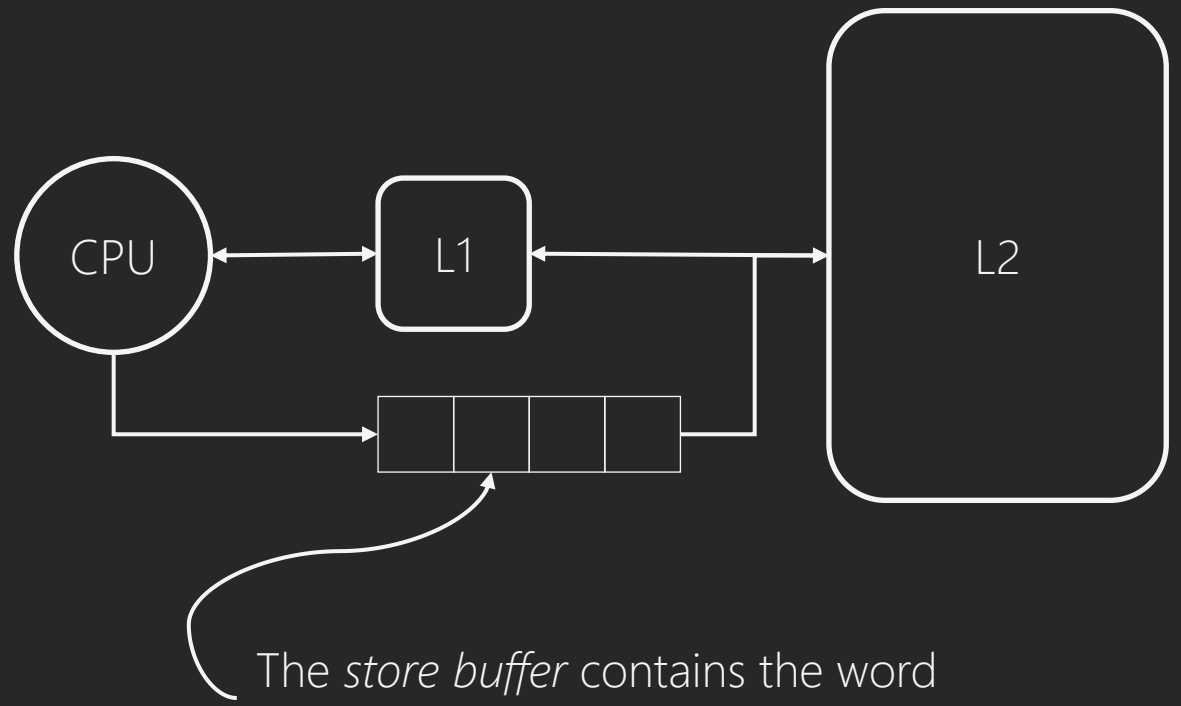


Write-back and the victim buffer



- Each cache block now needs a *dirty bit (D)*
  - D = 1: cache block has been written
  - D = 0: otherwise (“clean”)
- Written to main memory only on eviction if the block is dirty
- Load misses to a dirty cache block?
  - Move the *victim* to a buffer while handling the load (on critical path)
  - Write back the victim later to the next level (off critical path)

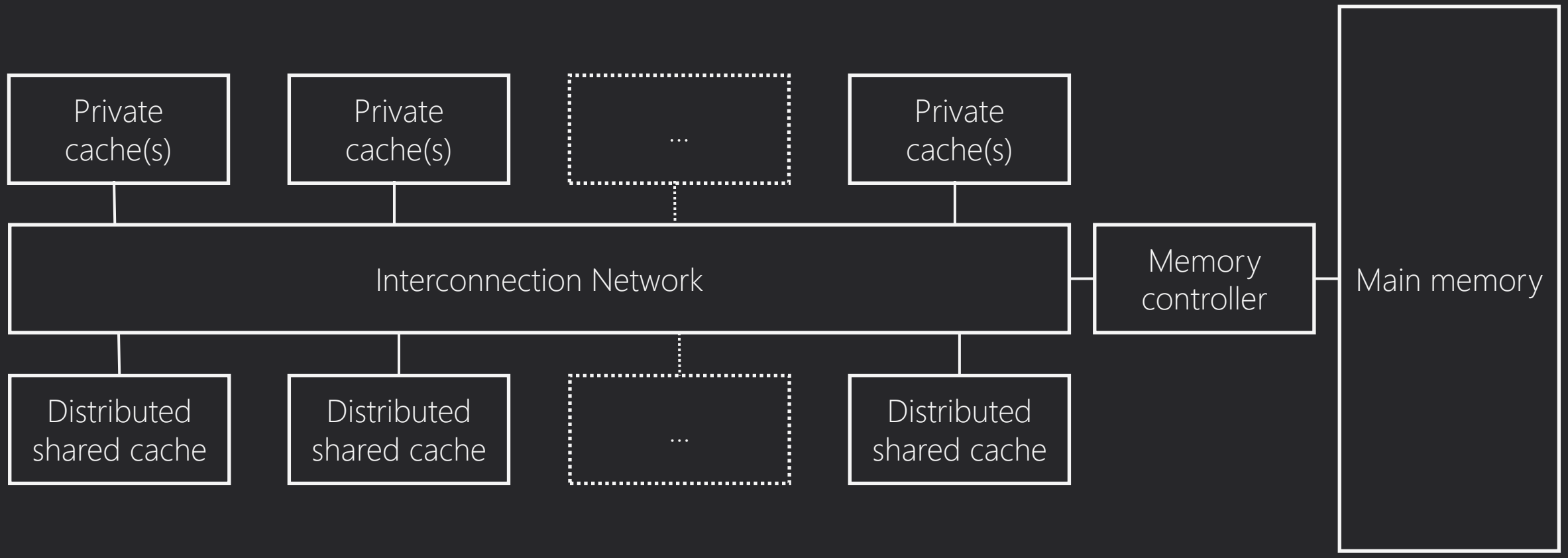
Write-through and the store buffer



- Writing immediately to L2 would increase the latency of stores
  - A store buffer removes the processor from the critical path
  - But stalls if store buffer is full
- Typically, a store miss does not update the L1 cache
  - Only inserted into store buffer
  - Load misses need to check both L1 and store buffer now

What is an interconnection network?

- An infrastructure to support sending messages across the chip
- Lots of wires and logic dedicated to communication



What is cache coherence?

Defines the behaviour of reads and writes to the same memory location (“What values can be returned by a read.”)

Time	Event	A’s private cache	B’s private cache	Main memory
0				42
1	A reads foo	42		
2	B reads foo	42	42	42
3	A writes to foo	16	42	16

Interconnection networks

A closer look at a shared bus

What is a shared bus?

Definition

- An interconnection network shared by all processors
- Using the bus, processors can:
  - Broadcast messages
  - Listen for messages (“snoop”)
- Only one message can be broadcast at a time

Pros and Cons

- Easy to use and easy to implement (simple) coherence
- Limited number of connections
  - Due to electrical constraints
- Contention (structural hazard)
  - Need to arbitrate between processors that want to broadcast at the same time

How do we measure bus performance?

Bus utilization for a single core

- Modeling percentage of time a core uses the bus

- U: utilization of the bus by a core
- t_b: bus transaction time (latency)
- t_c: average time between transactions

$$U = \frac{t_b}{t_c + t_b}$$

Bus occupancy

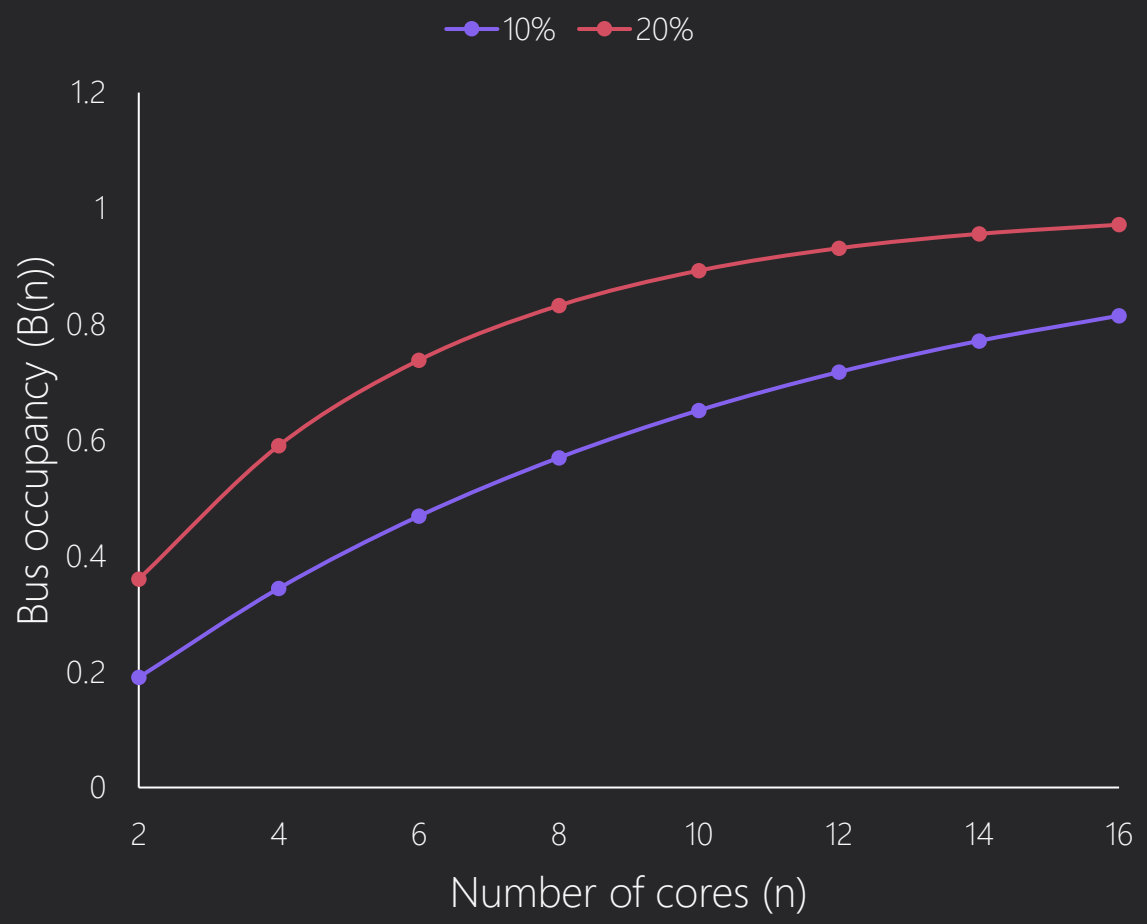
- Modeling when the bus is busy

- P: probability bus is busy for a core (U)
- n: number of cores
- B: probability of bus occupancy

$$B(n) = 1 - (1 - P)^n$$

How quickly does a shared bus saturate?

- Assume each core has an equal probability for utilizing the bus
- Cores will use the bus when they miss in their L1 cache
  - Many transactions needed for cache coherence
- Even with a high hit ratio, the bus saturates quickly
  - i.e., lots of contention for the bus



Summary: the shared bus

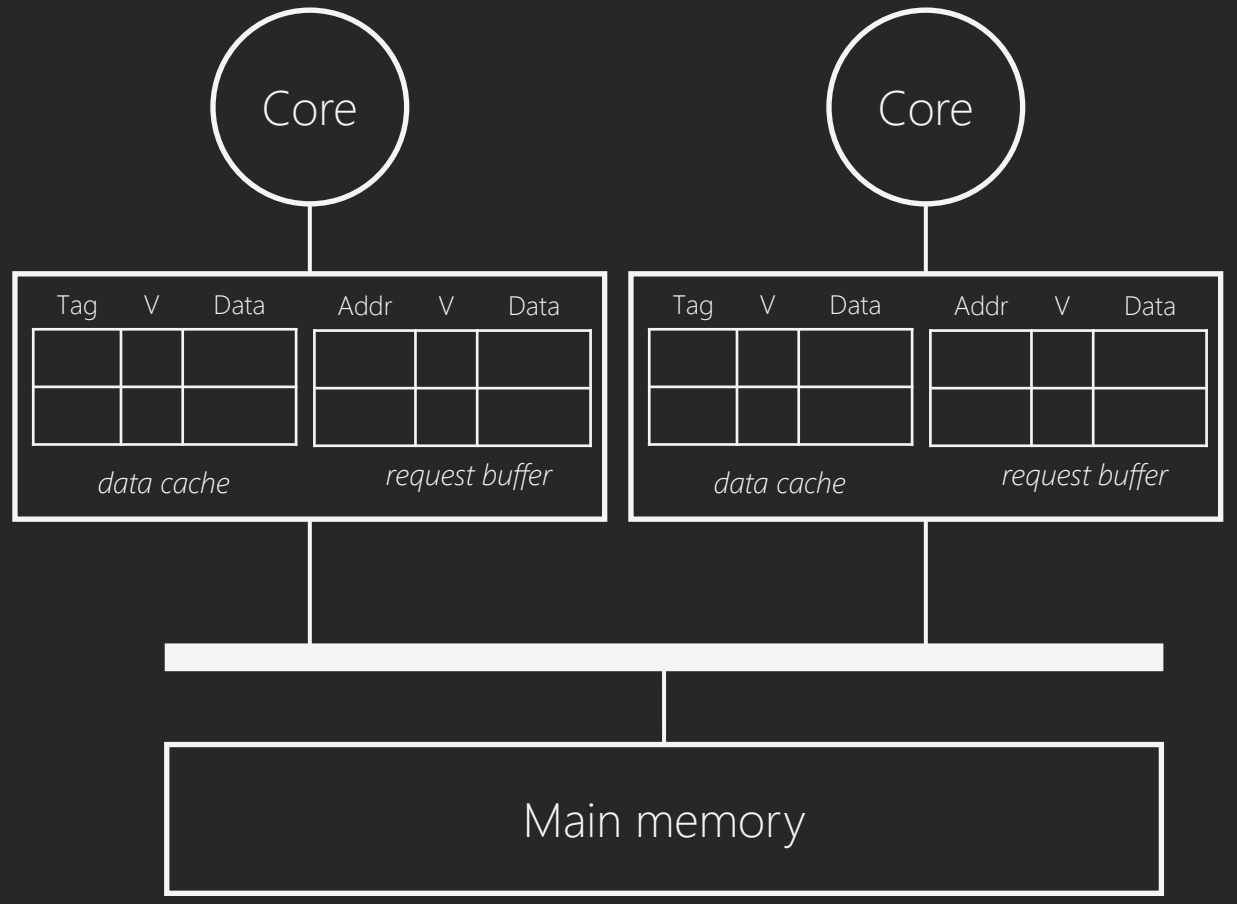
- Useful for communicating to many cores at once
- Limited in how many cores can connect to it
- Saturates quickly
  - Structural hazard
  - Contention exacerbated by cache coherence

Cache coherence

- A simple “snoopy” protocol

The system configuration

- One level of cache
  - Each core has a private L1 blocking, write-through, write-allocate cache
  - Blocking: core must stall until cache request is complete
  - Write-through: stores must update the next level of the hierarchy
  - Write-allocate: store misses most have their data loaded into the cache first, before being updated
- Shared bus (one transaction at a time)



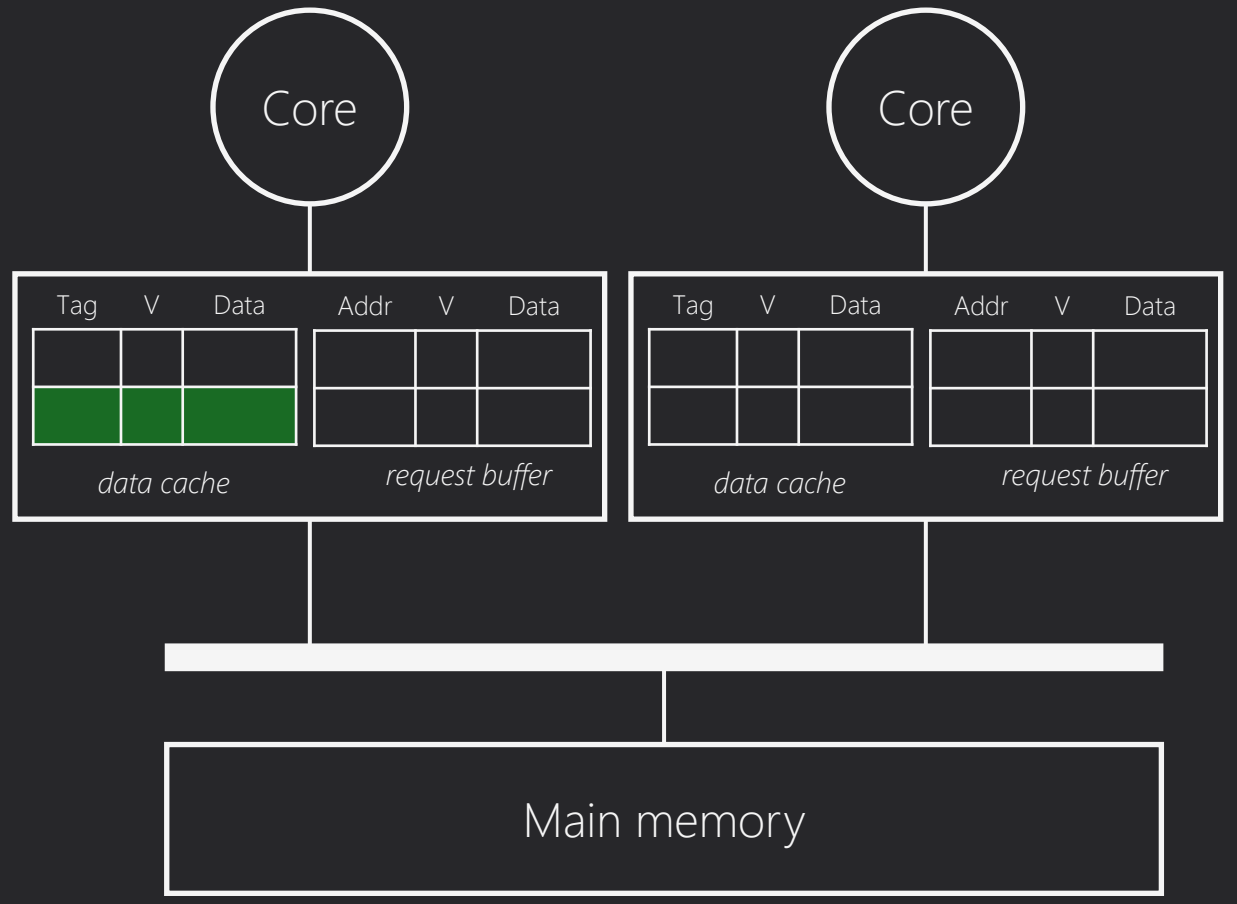
How should the cache behave?

- Need to respond to read and write requests from the attached core
  - Read-miss vs. read-hit
  - Write-miss vs. write-hit
- Need to snoop on the shared bus
  - Why?
- Need to broadcast on the shared bus
  - Why?

Source	Event	Action(s)
Local	Read-miss	
	Read-hit	
	Write-miss	
	Write-hit	
Remote	Read	
	Write	

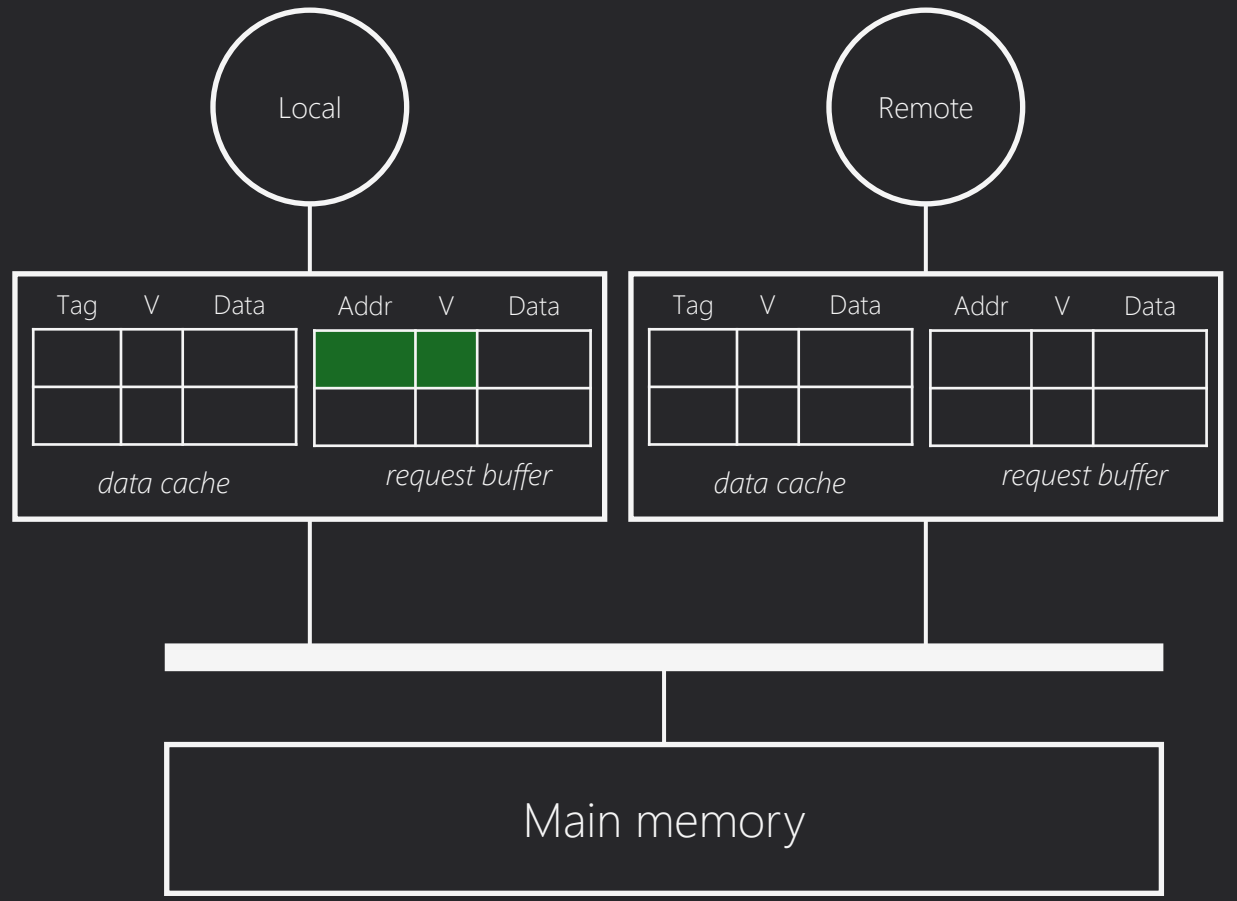
How does the cache handle a read hit?

1. Send data back to the core



How does the cache handle a read miss?

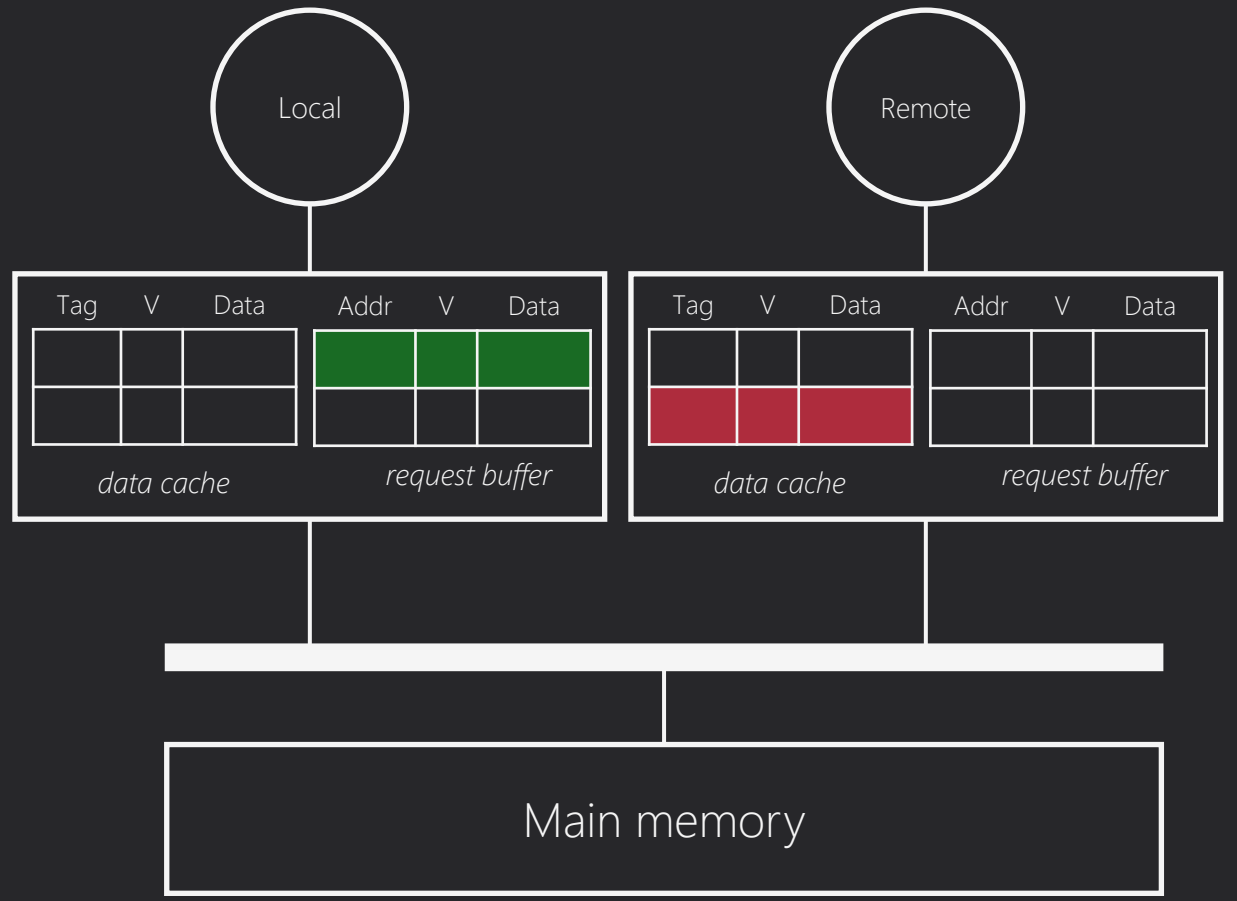
1. Insert request into request buffer
   - Set valid bit to *valid*
2. When request is at head
   I. Acquire the bus
   II. Issue a *bus-read* request
   III. Wait for response
   IV. Put data in cache / send data back to the core
   V. Clear request from request buffer



How does the cache handle a write hit?

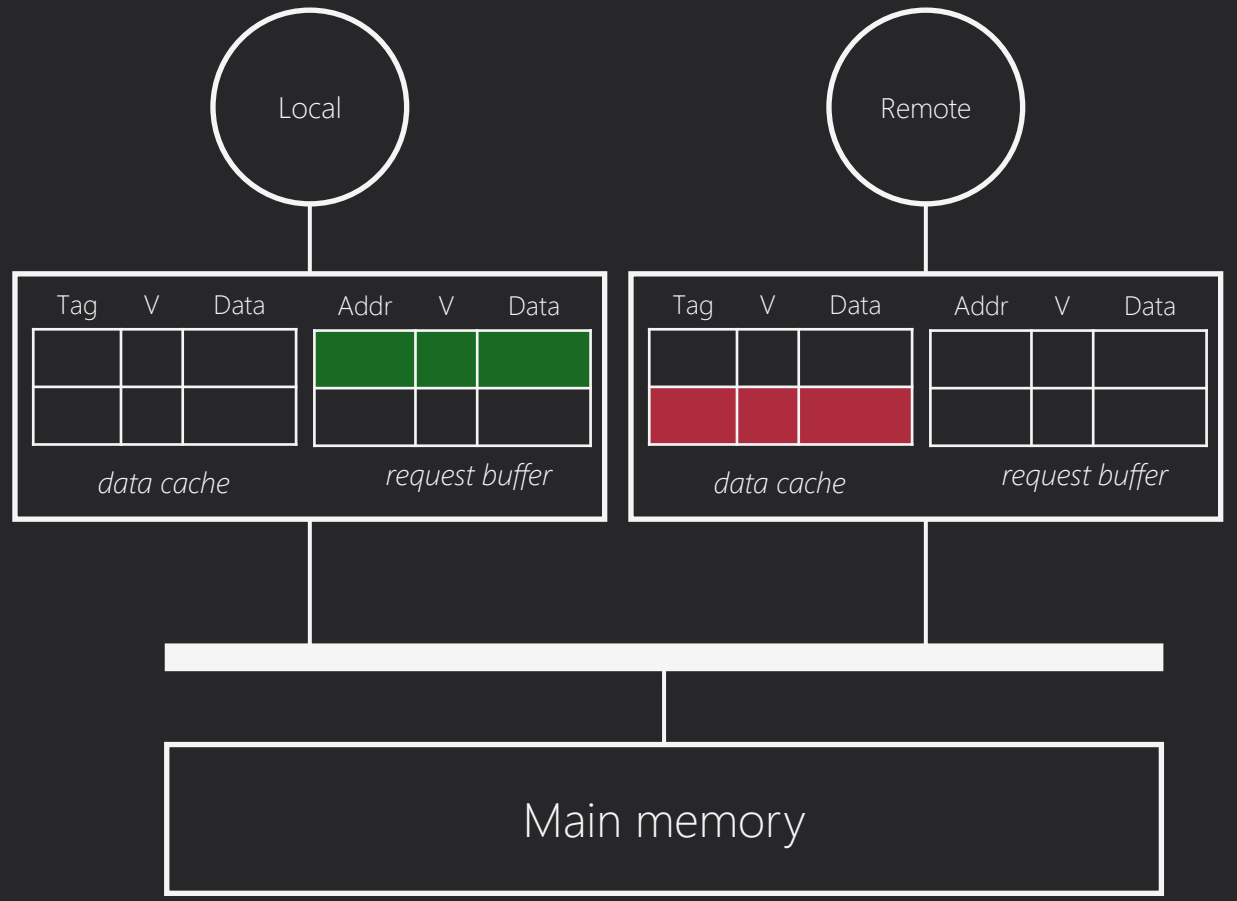
1. Insert request into request buffer
   - Set valid bit to *valid*
   - Remember: stores have data
2. When request is at head
   I. Acquire the bus
   II. Issue a *bus-write* request (hold)
   III. Put data in cache / unblock the core
   IV. Release the bus

- If the tag in a *remote cache* matches with the bus-write, set the data cache's corresponding valid bit to *invalid*



How does the cache handle a write miss?

- Remember this is a write-allocate policy
  1. Read data into cache (same as read miss?)
  2. Write data (same as write hit?)
- How is this different from a read miss?
  - Use *bus-read-exclusive* request, which invalidates remote caches with a matching tag
  - Then data is updated in the cache
- What about a no-write-allocate policy?
  - Replace *bus-read-exclusive* with *bus-write*

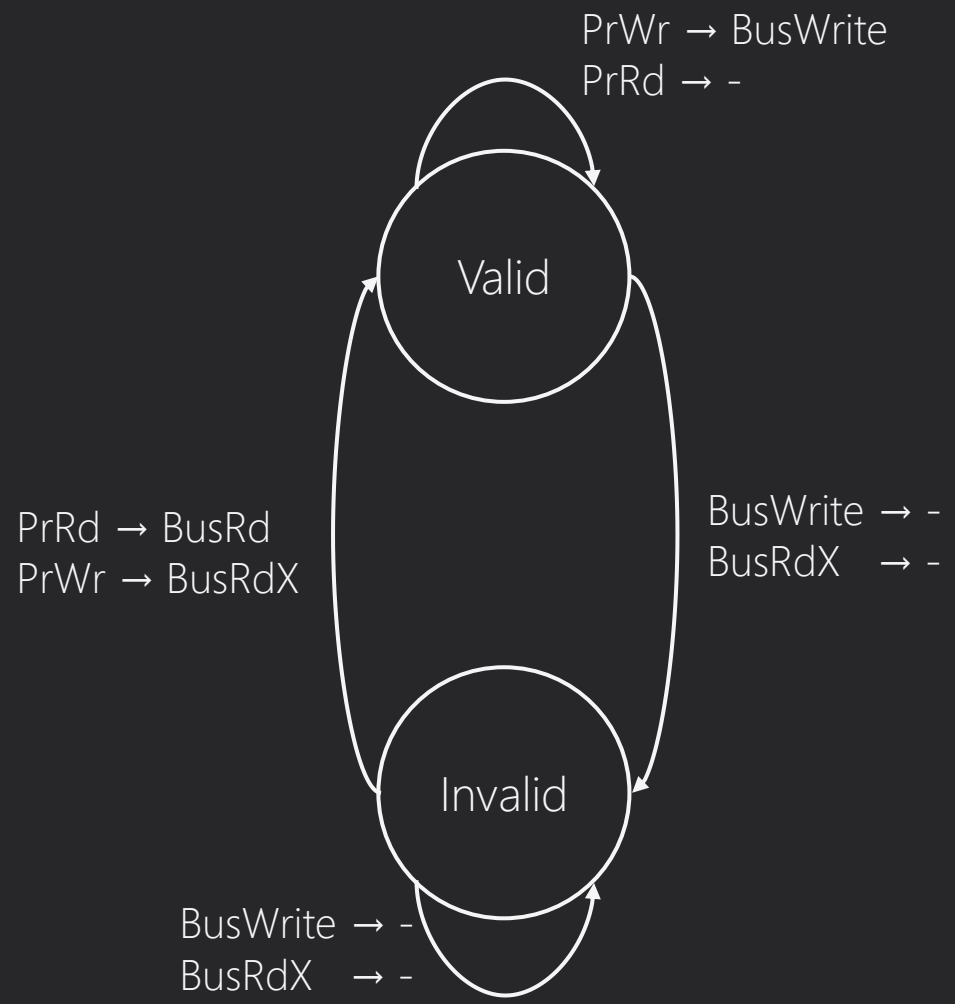


Specifying behaviour: state-transition diagram

- A hit or a miss is dictated by the *valid bit* in the data cache
  - Two states: valid and invalid

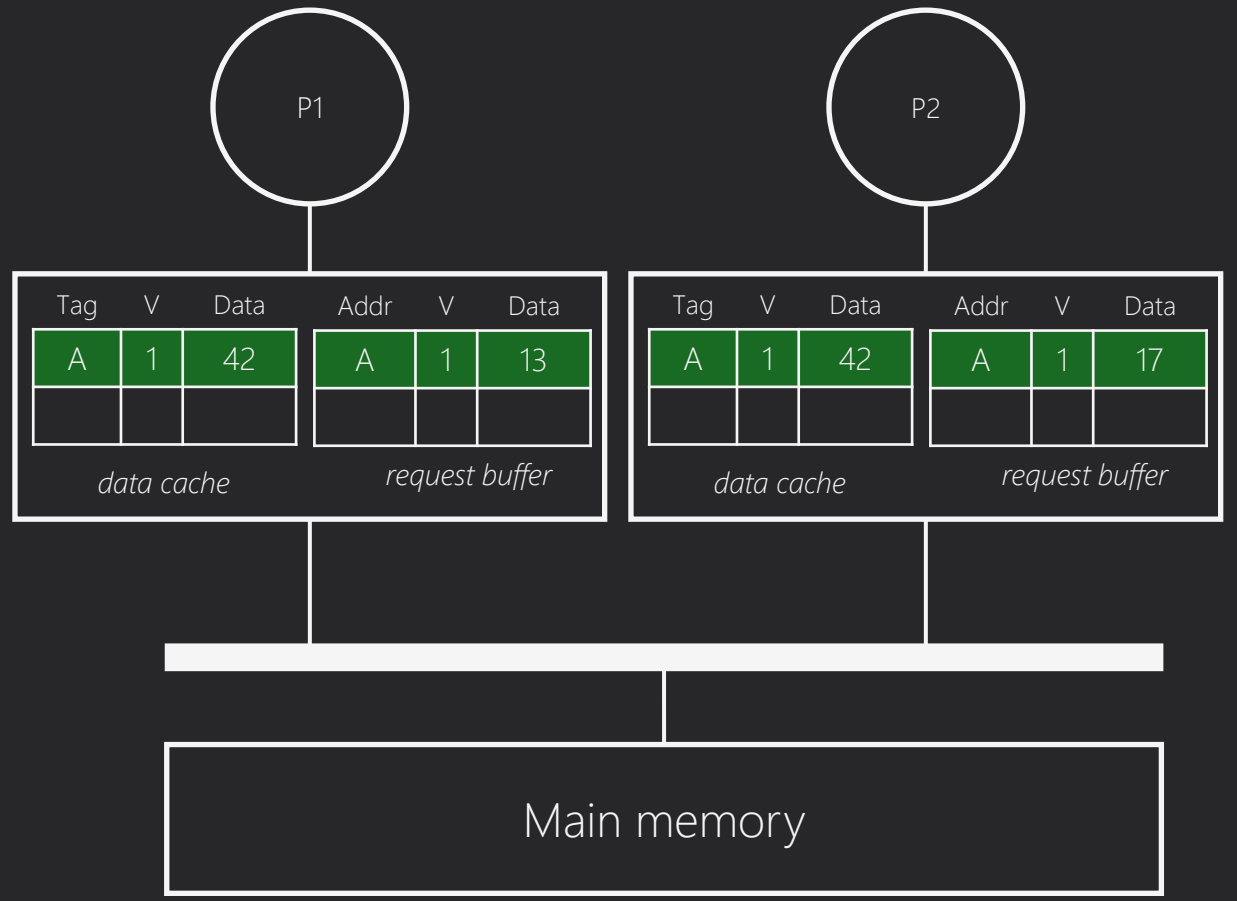
- Transitions to/from states require actions

PrRd	Processor read
PrWr	Processor write
BusRd	Read from memory
BusRdX	Read form memory and invalidate copies in remote caches
BusWrite	Write to memory and invalidate copies in remote caches



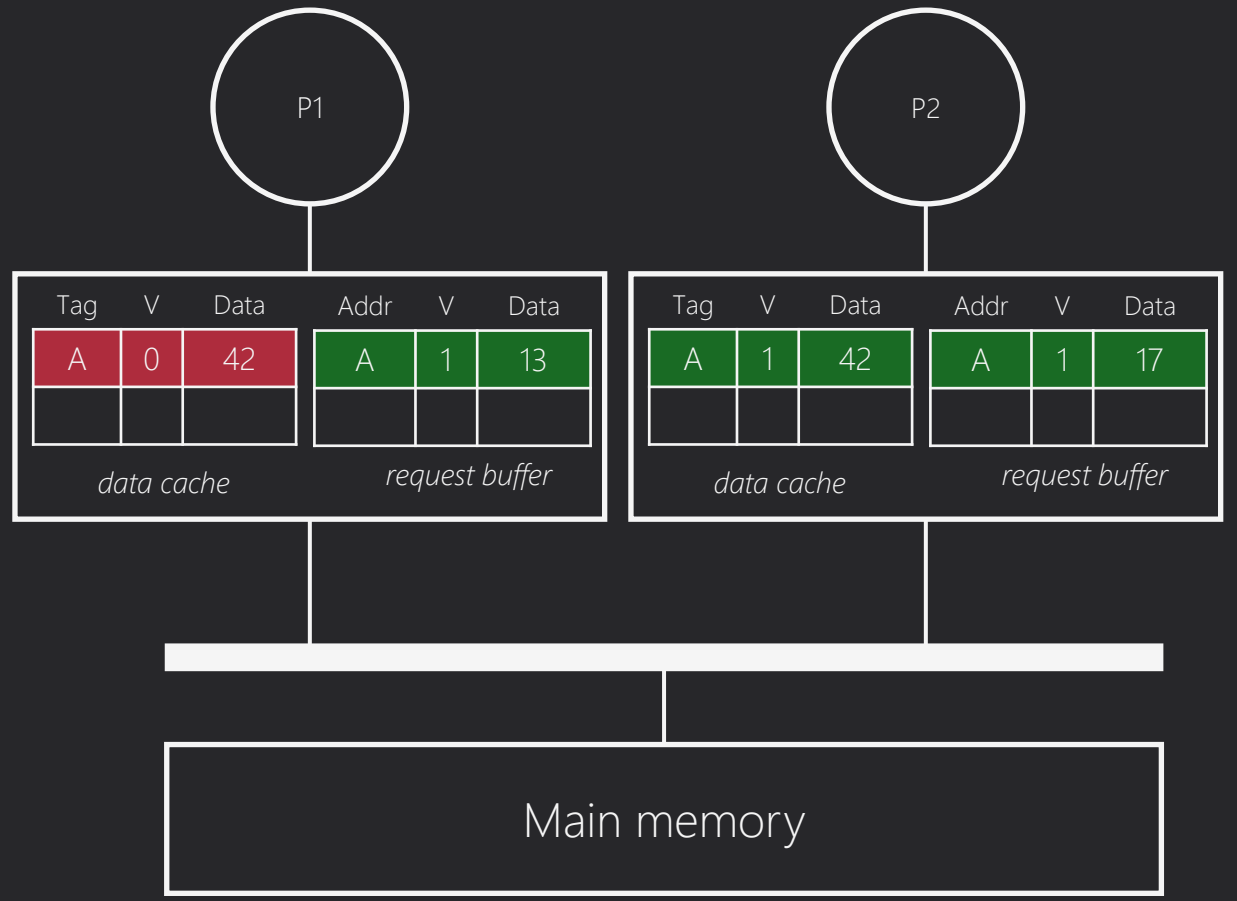
Example: parallel writes to block A

- Two processors P1 and P2 have a valid copy of A in their private cache
- What are the protocol actions when both issue a write request to block A “at the same time”?



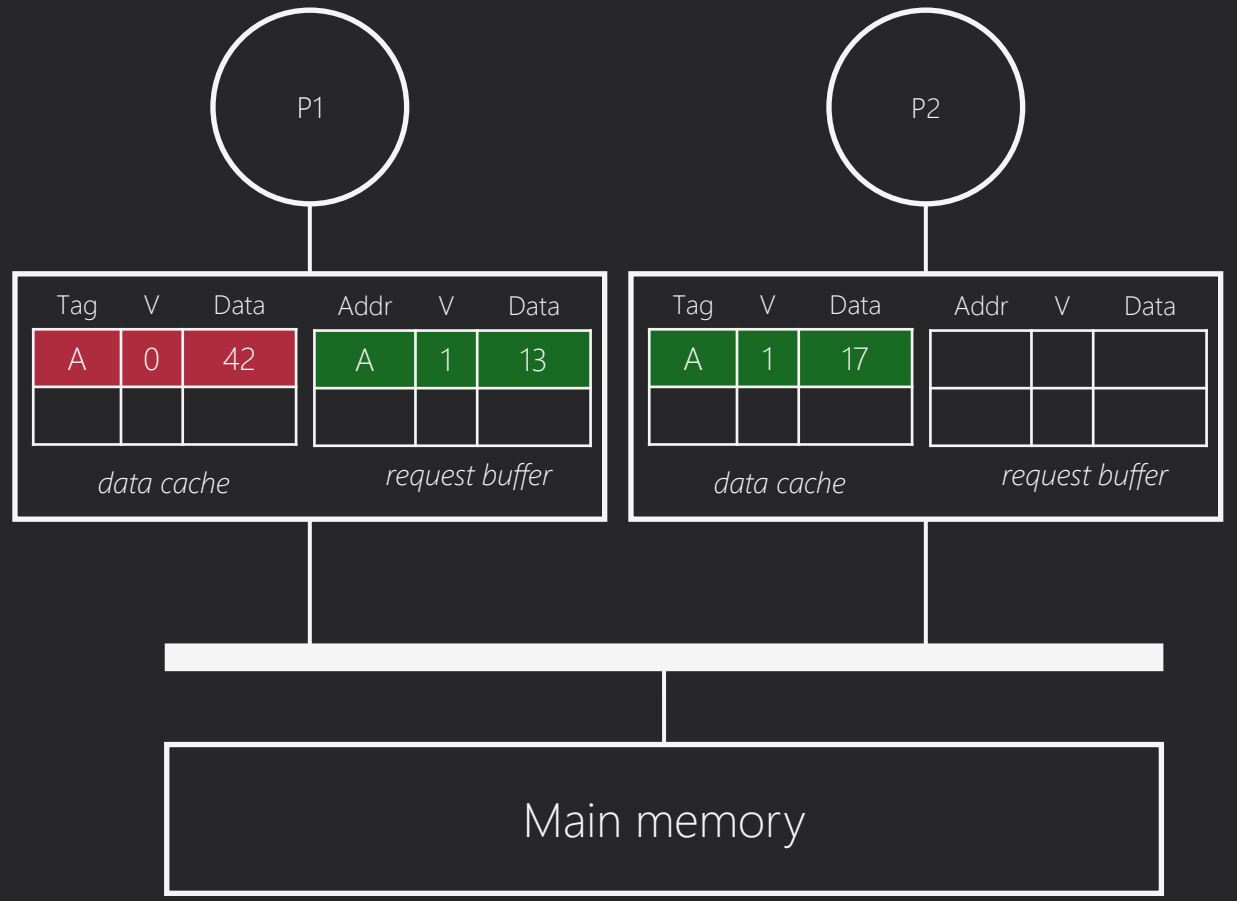
Step 1: suppose P2 acquires the bus first

1. P2's block in the data cache is valid
2. P2 issues a BusWrite request
   - Invalidates P1's cache (block A)



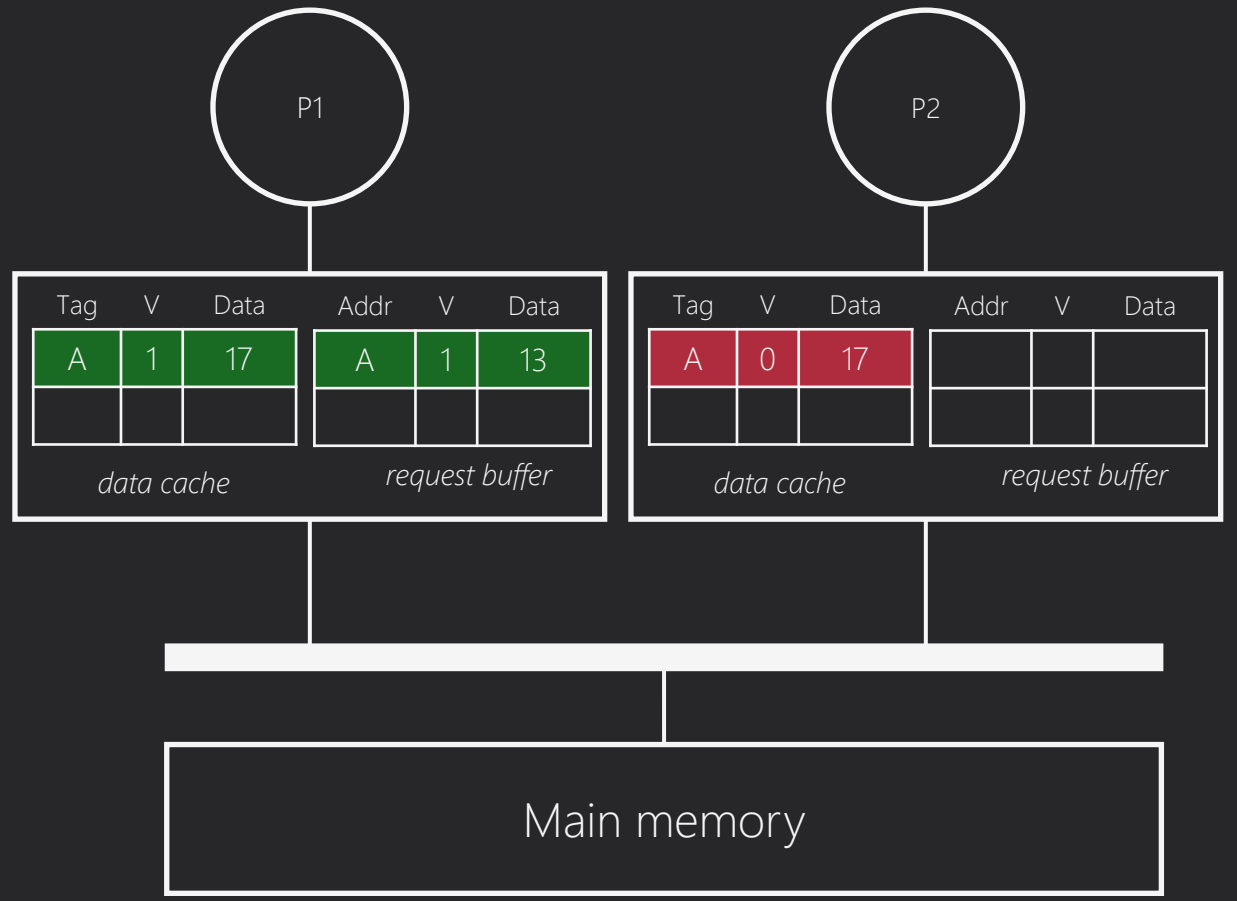
Step 2: P2 completes the write

1. P2's block in the data cache is valid
2. P2 issues a BusWrite request
   - Invalidates P1's cache (block A)
3. P2 updates its data cache
4. P2 releases the bus



Step 3: P1 acquires the bus next

1. P1's block in the data cache is invalid
2. P1 issues a BusRdX (due to write-allocate policy)
   - Invalidates P2's cache (block A)
   - Block is fetched from main memory



Are there other design issues?

New problem

- Caches must respond to requests from two sources: core and bus
- All bus requests result in tag lookups in all caches
  - A bus request could stall a processor

Solution

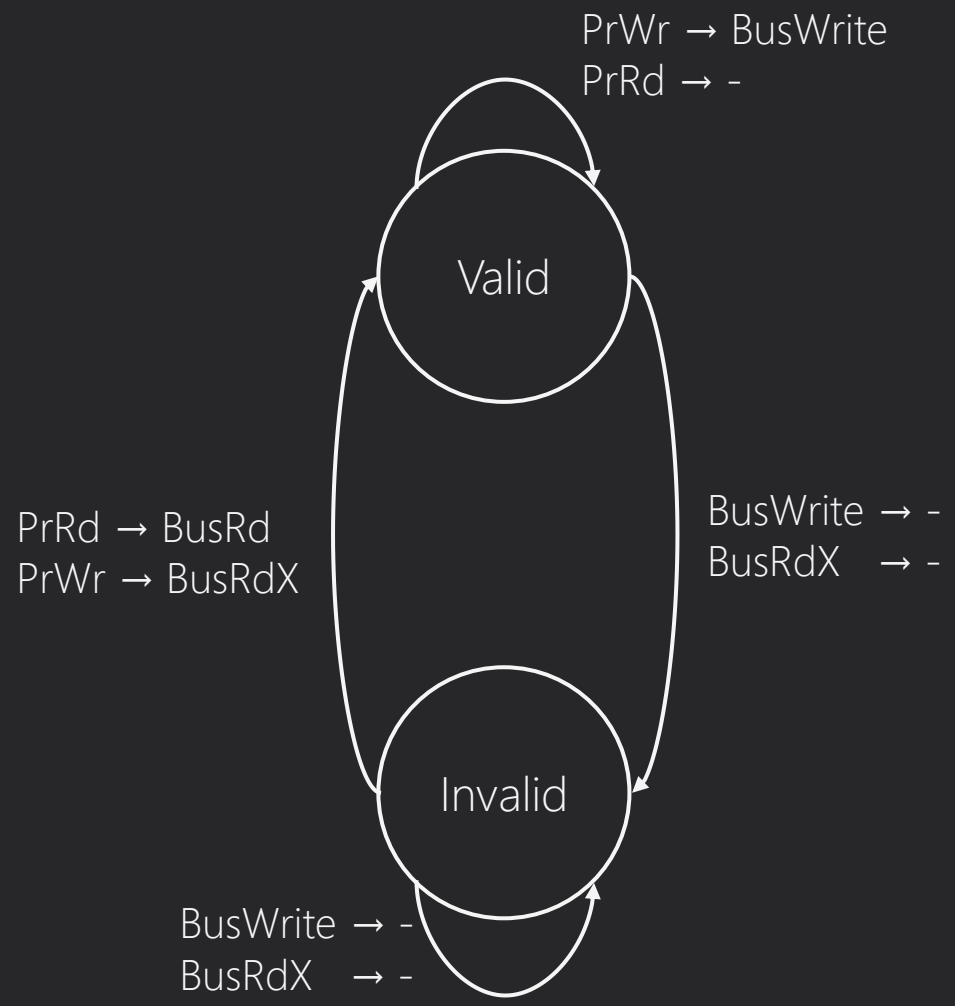
- Duplicate tag storage (a *tag store*)
  - One for the core
  - One for the bus
- Keep the two storages synchronized
  - Only some bus requests will update the bus tag store

Conclusion

A summary and reflection on our simple coherence protocol

A simple snoopy cache coherence protocol

- Two states help us know whether a core's local data is valid
- When a core mutates data, other caches are notified on the shared bus
- Caches now respond to requests from the core and the bus to maintain cache coherence



What about performance?

- Every write request triggers a bus transaction
  - But not all cache blocks that are written are shared…

- How can we track whether a cache block is shared or not?
  - We need more states (Invalid, Shared, Modified)
  - We need more types of actions (to transition between states)