4 • Chip Multiprocessors (I)

Chip Multiprocessors (ACS MPhil)

Robert Mullins

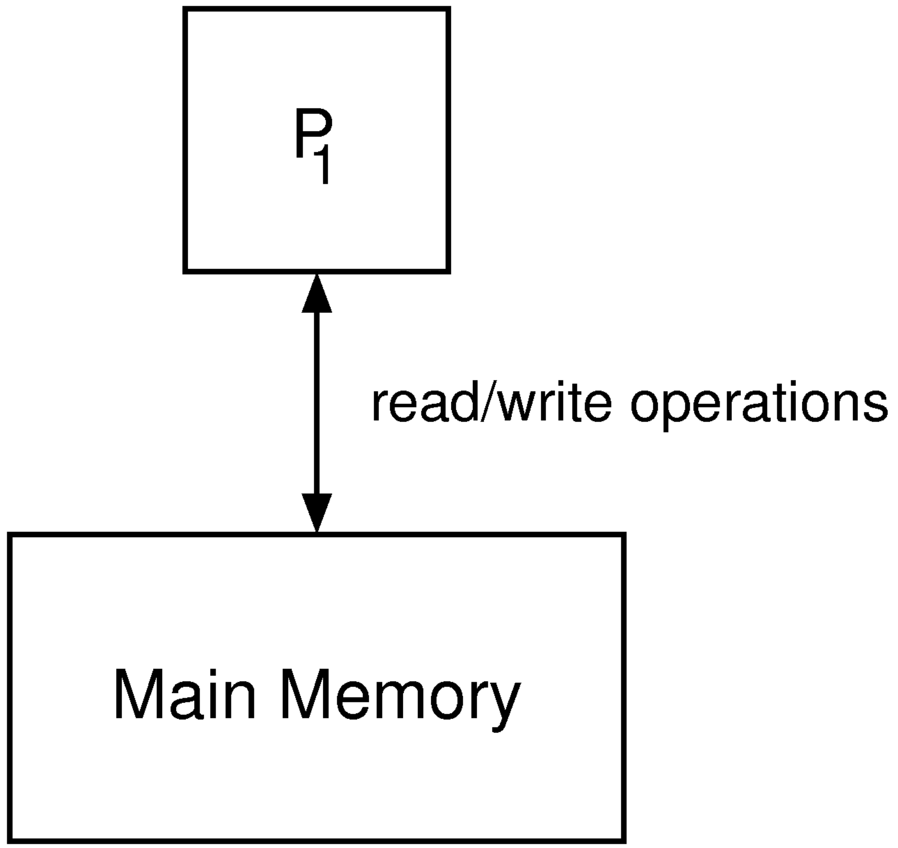
2010-11

Overview

- **Coherent memory systems**
- **Introduction to cache coherency protocols**
  - Advanced cache coherency protocols, memory systems and synchronization covered in the next seminar
- **Memory consistency models**
  - Discuss tutorial paper in reading group

Memory

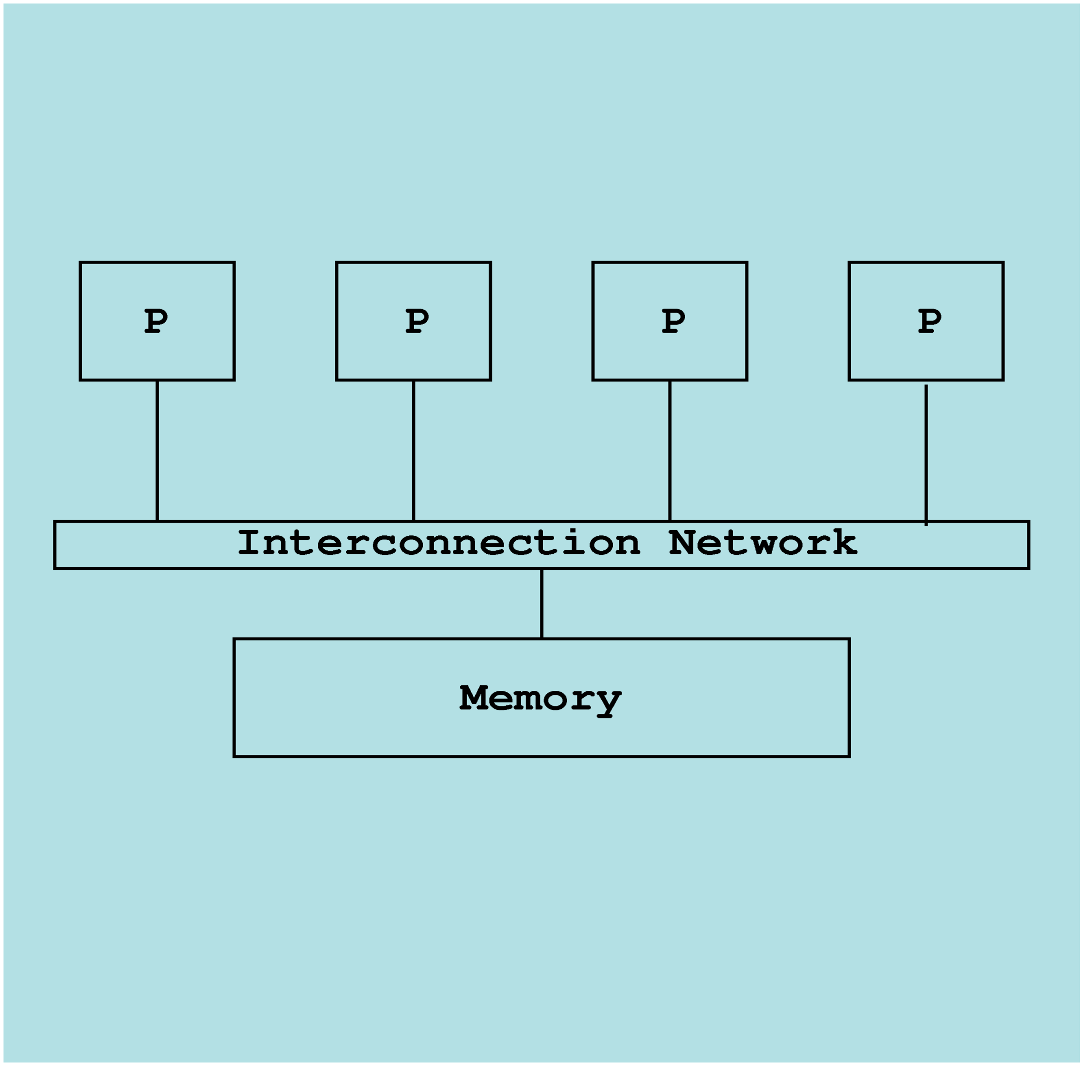
- We expect memory to provide a set of locations that hold the values we write to them
  - In a uniprocessor system we boost performance by buffering and reordering memory operations and introducing caches
  - These optimisations rarely affect our intuitive view of how memory should behave



Multiprocessor memory systems

- How do we expect memory to behave in a multiprocessor system?
- How can we provide a high-performance memory system?
  - What are the implications of supporting caches and other memory system optimisations?
    - What are the different ways we may organise our memory hierarchy?
  - What impact does the choice of interconnection network have on the memory system?
  - How do we build memory systems that can support hundreds of processors?
    - Seminar 5

Shared-memory



A “coherent” memory

- *How might we expect a* ***single memory location*** *to behave when accessed by multiple processors?*
- Informally, we would expect (it would at least appear that) the read/write operations from each processor are interleaved and that the memory location will respond to this combined stream of operations as if it came from a single processor
  - We have no reason to believe that the memory should interleave the accesses from different processors in a particular way (only that individual program orders should be preserved)
  - For any interleaving the memory does permit, it should maintain our expected view of how a memory location should behave

A “coherent” memory

- A memory system is coherent if, for **each location**, it can serialise all operations such that:
  1) Operations issued by each process occur in the order they we issued
  2) The value returned by a read operation is the value written by the last write (“last” is the most recent operation in the apparent serial order that a coherent memory imposes)

- Implicit properties:
  - *Write propagation* – writes become visible to other processes
  - *Write serialisation* – all writes to a location are seen in the same order by all processes

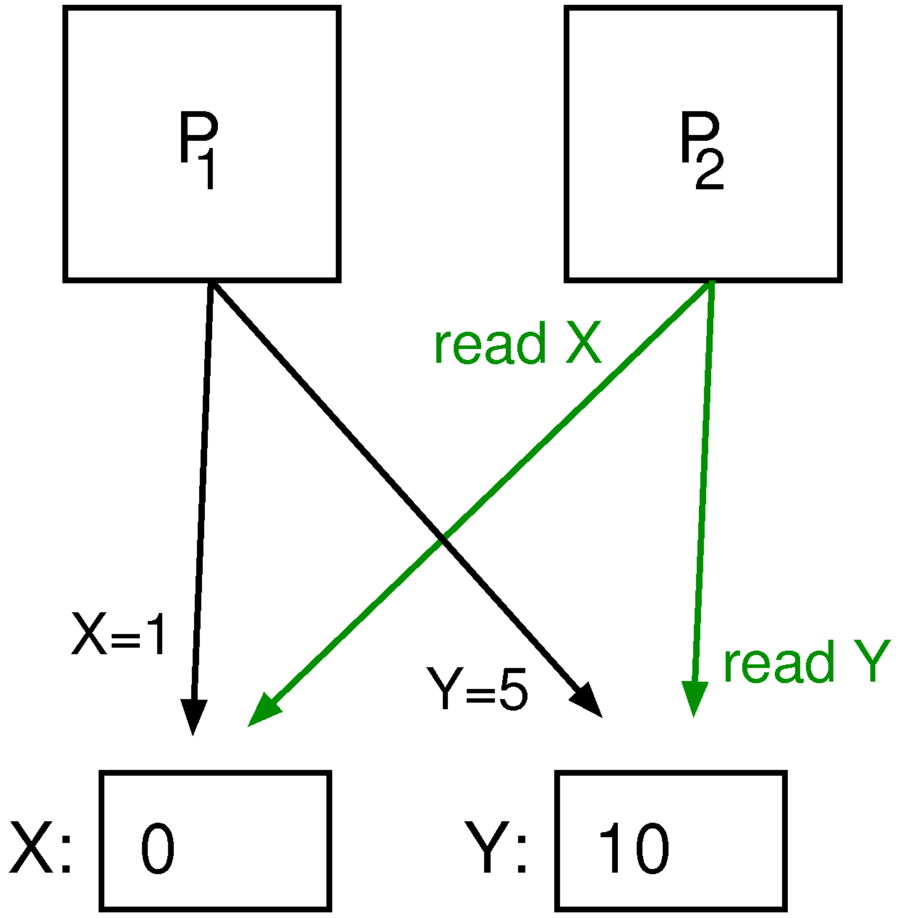
See Culler book p.273-277

Is coherence all that we expect?

- Coherence is concerned with the behaviour of **individual** memory locations
- The memory system illustrated (containing locations X and Y) is coherent but does not guarantee anything about when writes become visible to other processors
- Consider this program:

```
P1        P2

Y=5       while (X=0)
X=1       read Y
```

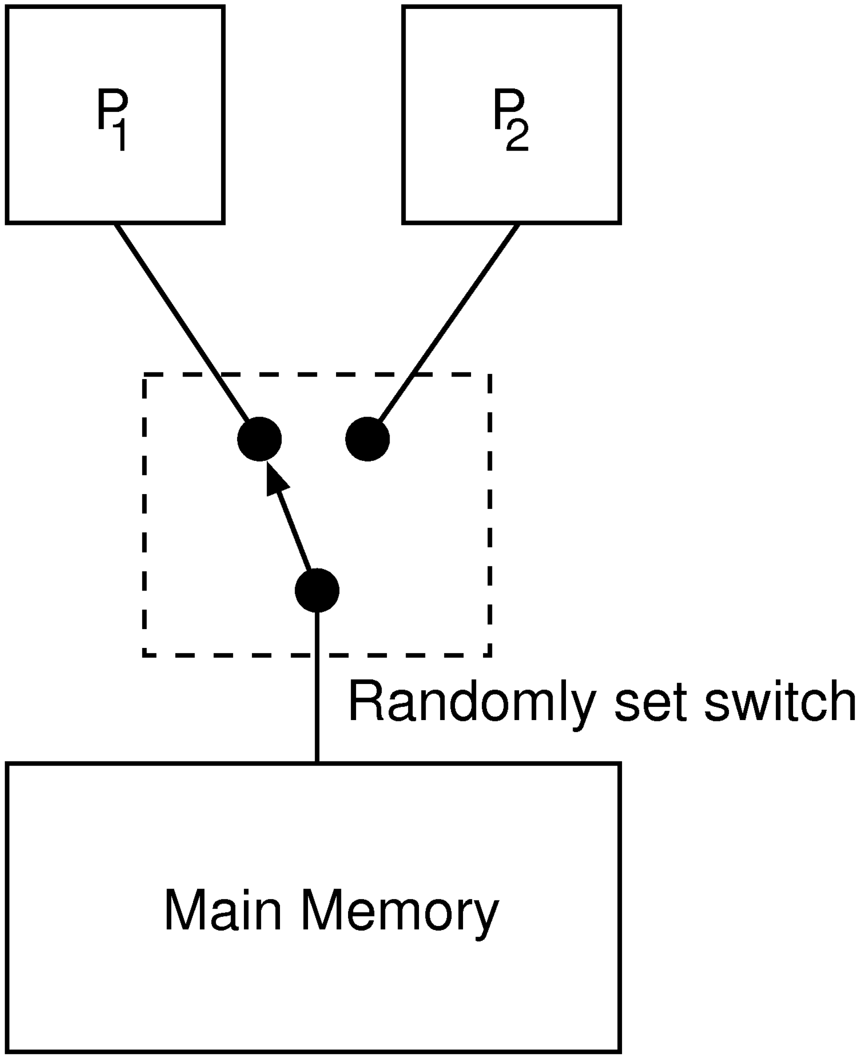


Is coherence all that we expect?

- The operation Y=5 does not need to have completed (as we might expect) before X=1 is performed and X is read by P2
- Perhaps surprisingly, this allows P2 to exit from the while loop and read the value 10 from memory location Y, clearly not the intent of the programmer
  - In reality, there are many reasons why the Y=5 write operation may be delayed in this way (*e.g.* due to congestion in the interconnection network)

Sequential consistency

- An intuitive model of an ordering (or consistency model) for a shared address space is Lamport's **sequential consistency** (SC)
- *"A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program"*



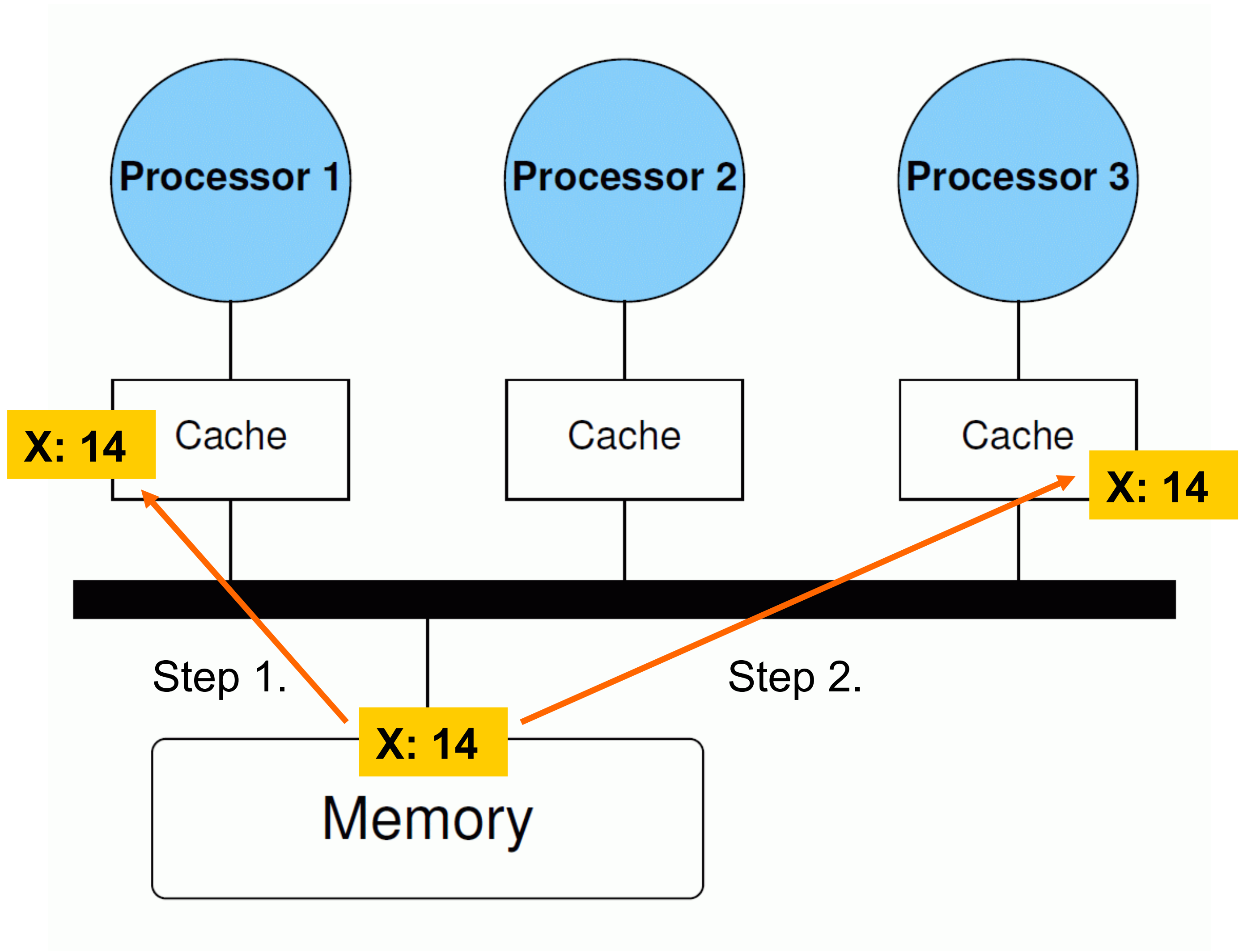
Sequential consistency

- Unfortunately, sequential consistency restricts the use of many common memory system optimisations
  - *e.g.* write buffers, overlapping write operations, non-blocking read operations, use of caches and even compiler optimisations
- The majority (if not all) modern multiprocessors instead adopt a relaxed memory consistency model
  - We will discuss this in detail in the reading group

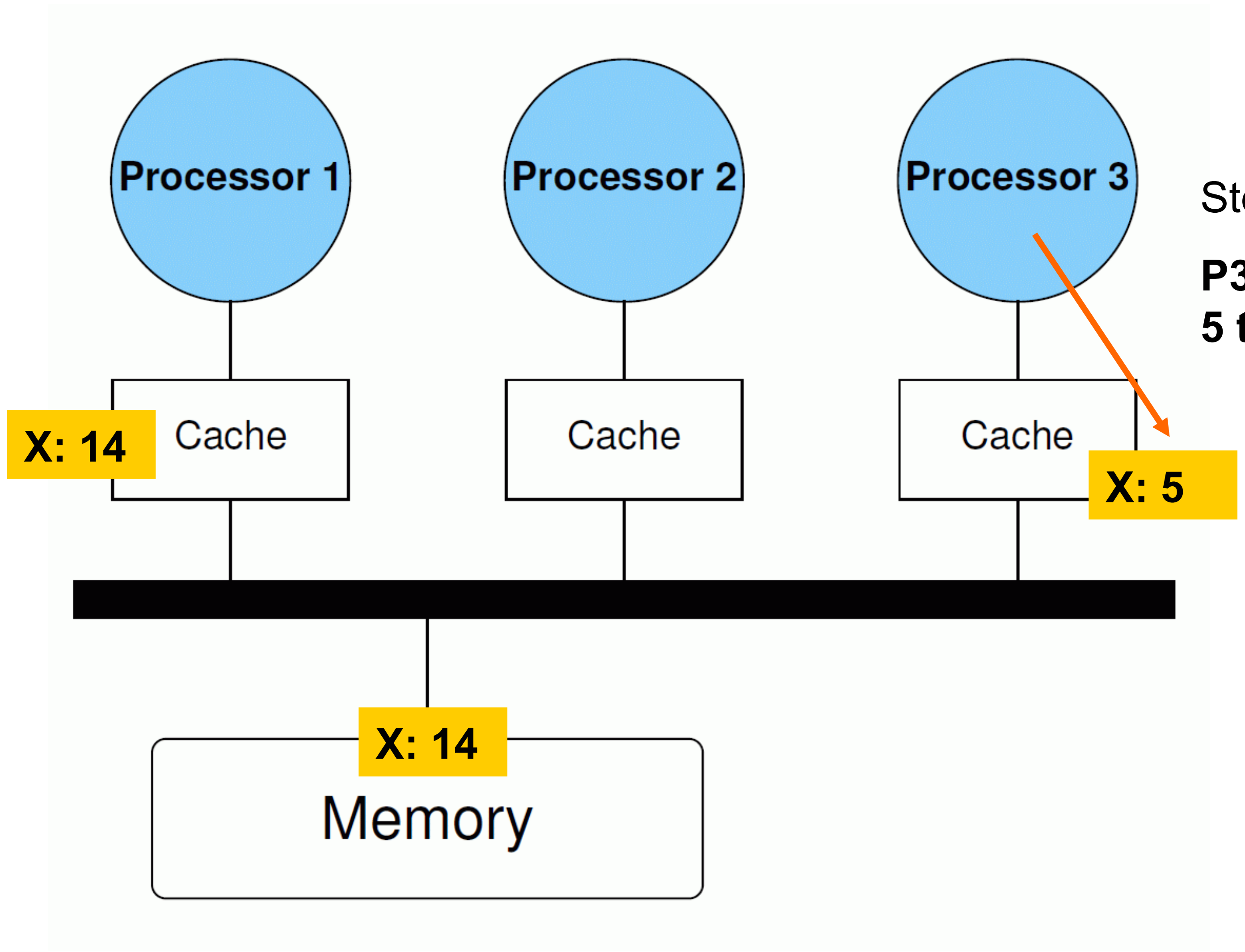
Cache coherency

- Let's examine the problem of providing a coherent memory system in a multiprocessor where each processor has a private cache
- In general, we consider coherency issues at the boundary between private caches and shared memory (be it main memory or a shared cache)

Cache coherency



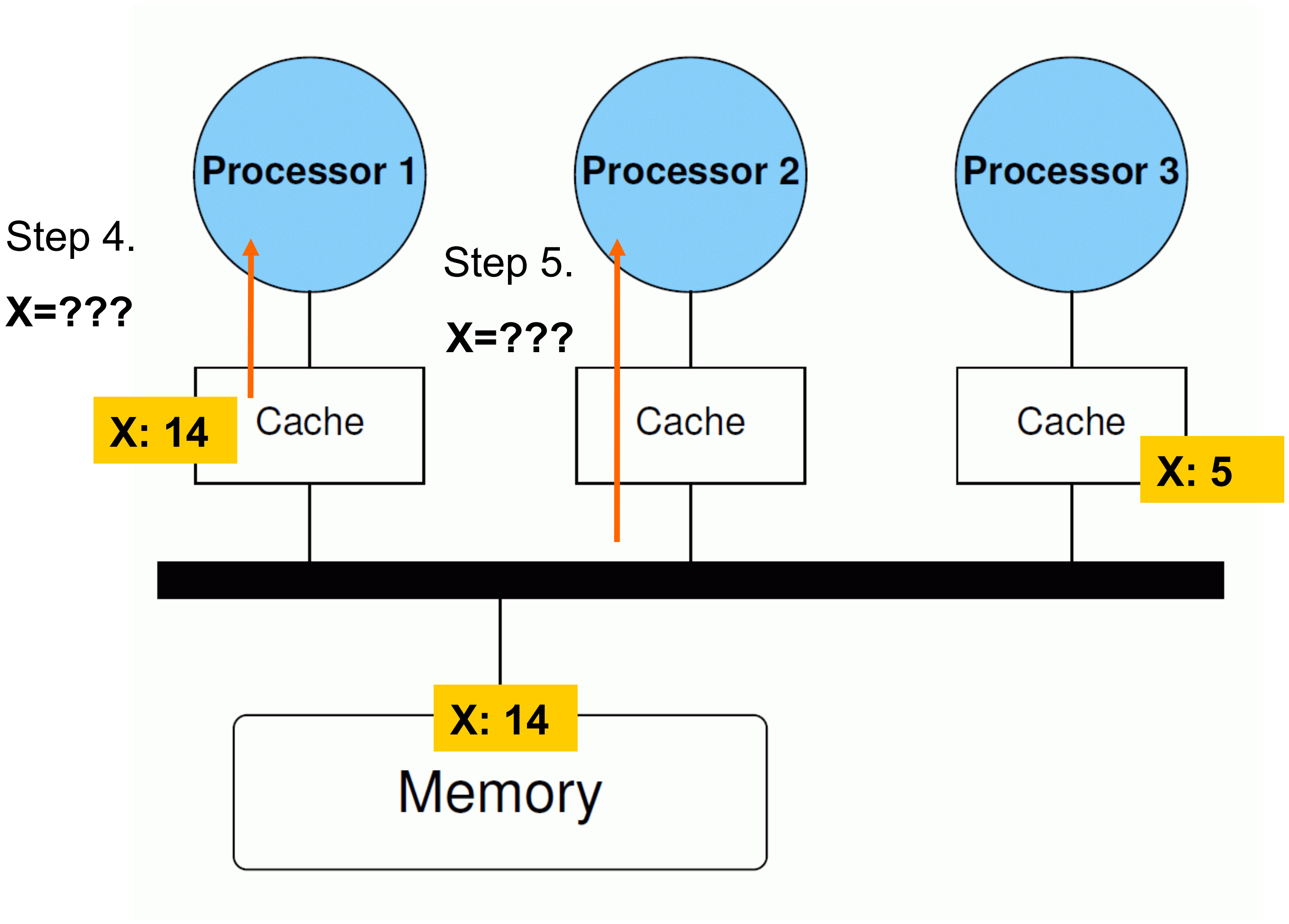
Cache coherency



Step 3.

P3 writes 5 to X

Cache coherency



Cache coherency

- Clearly this memory system can violate our definition of a coherent memory
  - It doesn't even guarantee that writes are propagated
  - This is a result of the ability of the caches to duplicate data

Cache coherency

The most common solution is to add support for cache coherency in hardware

- reading and writing shared variables is a frequent event, we don't want to restrict caching (to private data) or handle these common events in software
  - We'll look at some alternatives to full coherence
- Caches automatically replicate and migrate data closer to the processor, they help reduce communication (energy/power/congestion) and memory latency
- Cache coherent shared memory provides a flexible general-purpose platform
  - Although efficient hardware implementations can quickly become complex

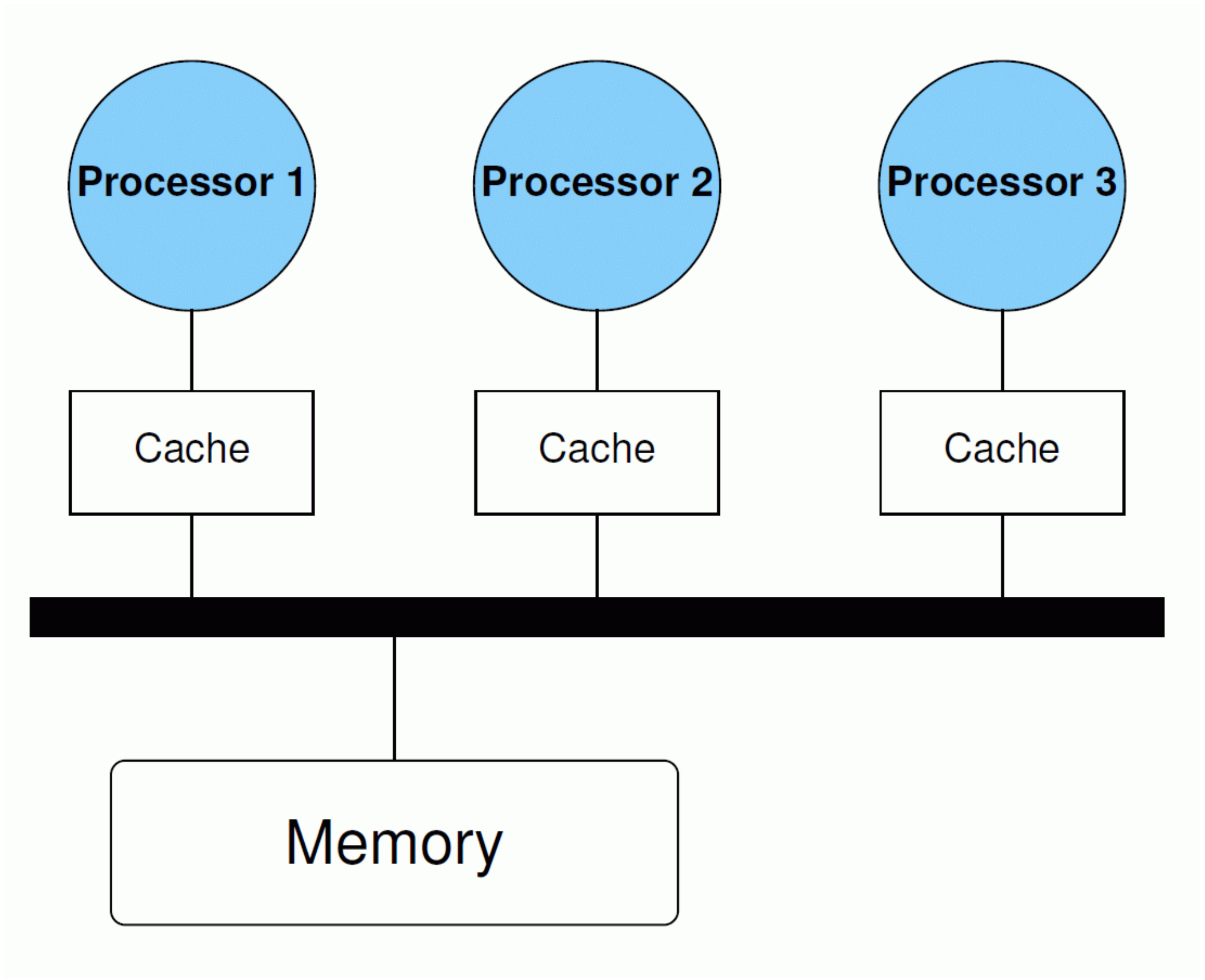
Cache coherency protocols

- Let's examine some examples:
  - Simple 2-state write-through invalidate protocol
  - 3-state (MSI) write-back invalidate protocol
  - 4-state MESI (or Illinois) invalidate protocol
  - Dragon (update) protocol

Cache coherency protocols

- The simple protocols we will examine today all assume that the processors are connected to main memory via a single **shared bus**
  - Access to the bus is arbitrated – at most one transaction takes place at a time
  - All bus transactions are broadcast and can be observed by all processors (in the same order)
  - Coherence is maintained by having all cache controllers “snoop” (**snoopy protocol**) on the bus and monitor the transactions
    - The controller takes action if the bus transaction involves a memory block of which it has a copy

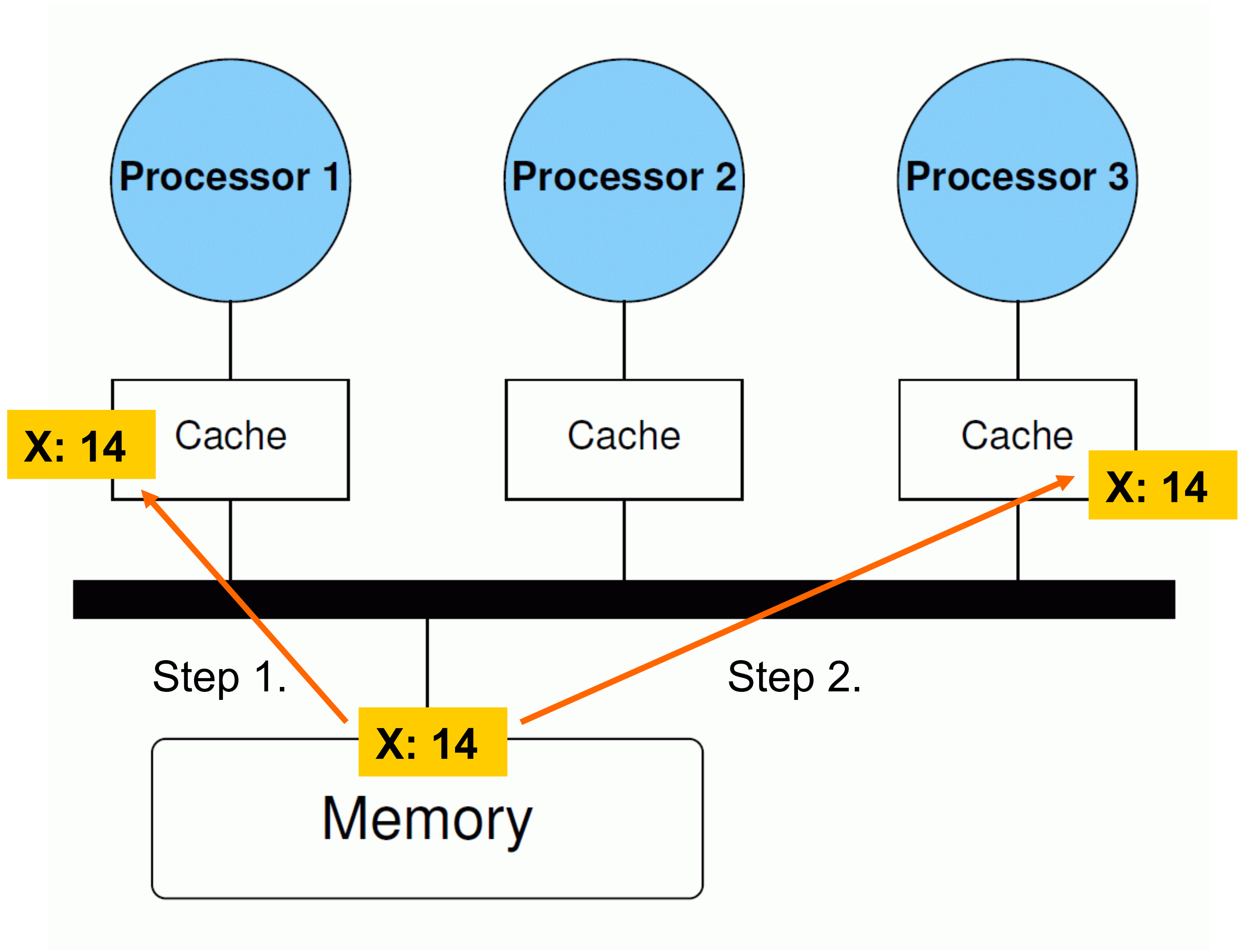
A bus-based system



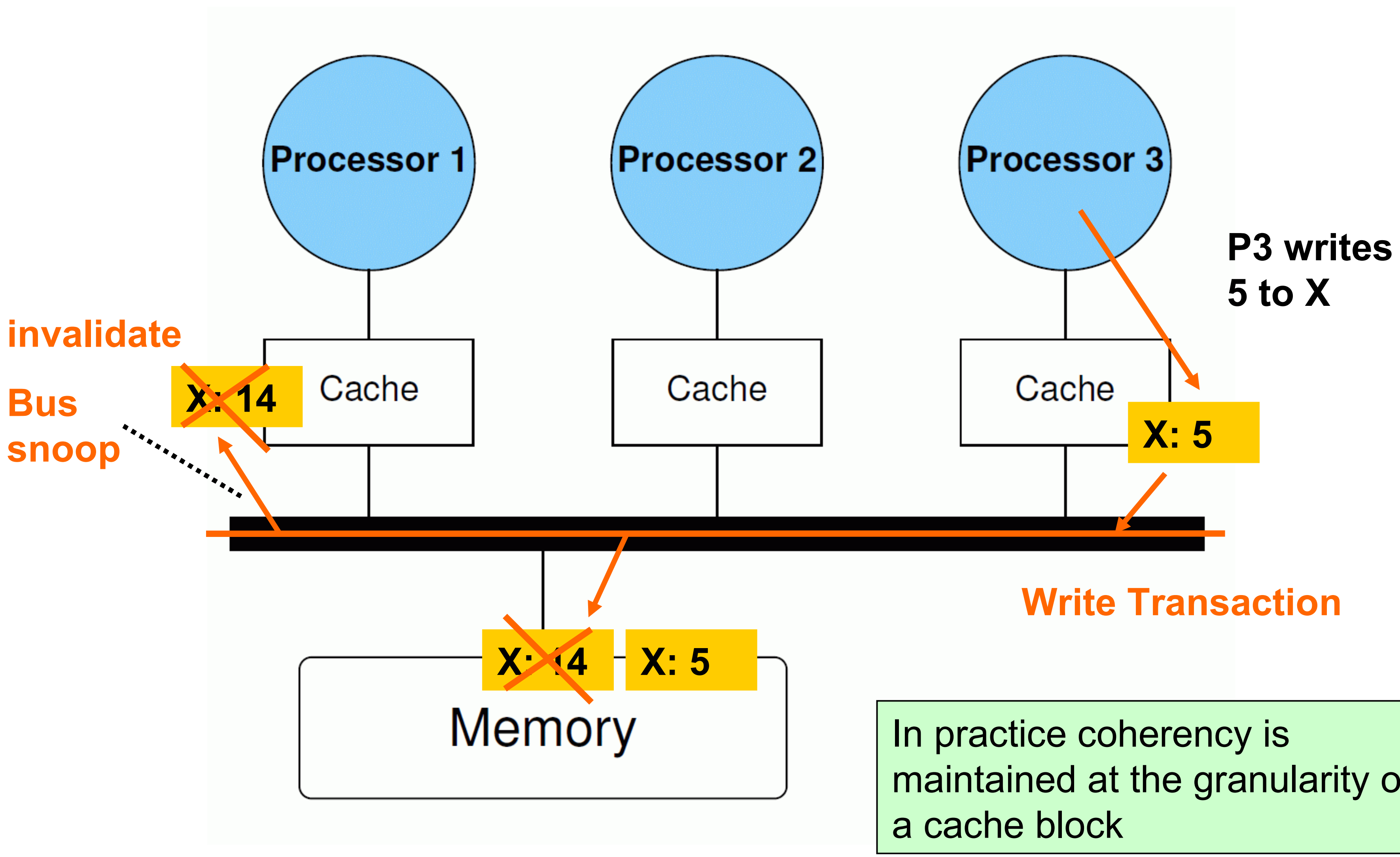
2-state invalidate protocol

- Let's examine a simple write-through invalidation protocol
  - Write-through caches
    - Every write operation (even if the block is in the cache) causes a write transaction on the bus and main memory to be updated
  - Invalidate or invalidation-based protocols
    - The snooping cache monitors the bus for writes. If it detects that another processor has written to a block it is caching, it invalidates its copy.
    - This requires each cache controller to perform a tag match operation
    - Cache tags can be made dual-ported

2-state invalidate protocol



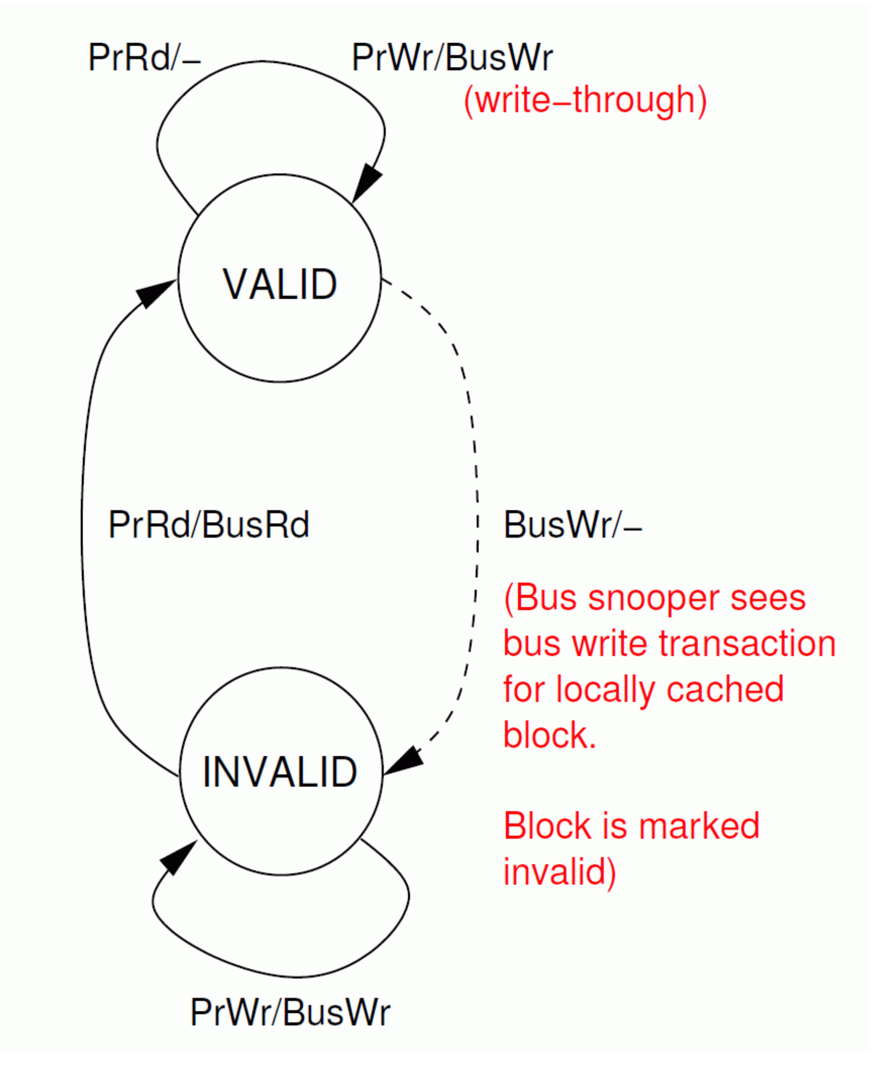
2-state invalidate protocol



In practice coherency is maintained at the granularity of a cache block

2-state invalidate protocol

- The protocol is defined by a collection of cooperating finite state machines
- Each cache controller receives
  - Processor memory requests
  - Information from snooping the bus
- In response the controller may
  - Update the state of a particular cache block
  - Initiate a new bus transaction
- The state transitions shown using dotted lines are in response to bus transactions

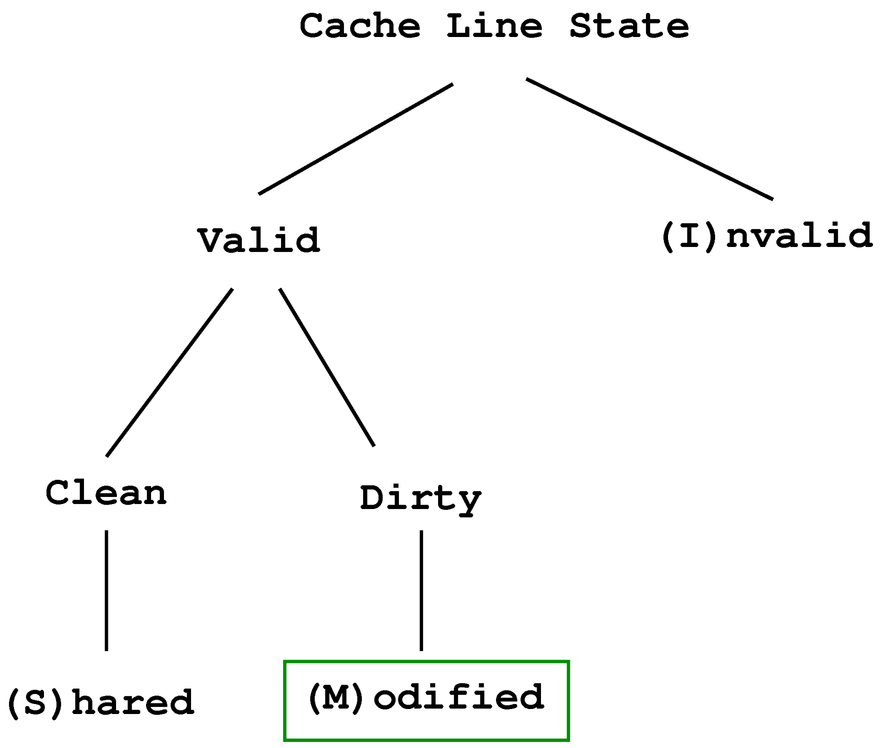


MSI write-back invalidate protocol

- Write-through caches simplify cache coherence as all writes are broadcast over the bus and we can always read the most recent value of a data item from main memory
- Unfortunately they require additional memory bandwidth. For this reason write-back caches are used in most multiprocessors, as they can support more/faster processors.

MSI write-back invalidate protocol

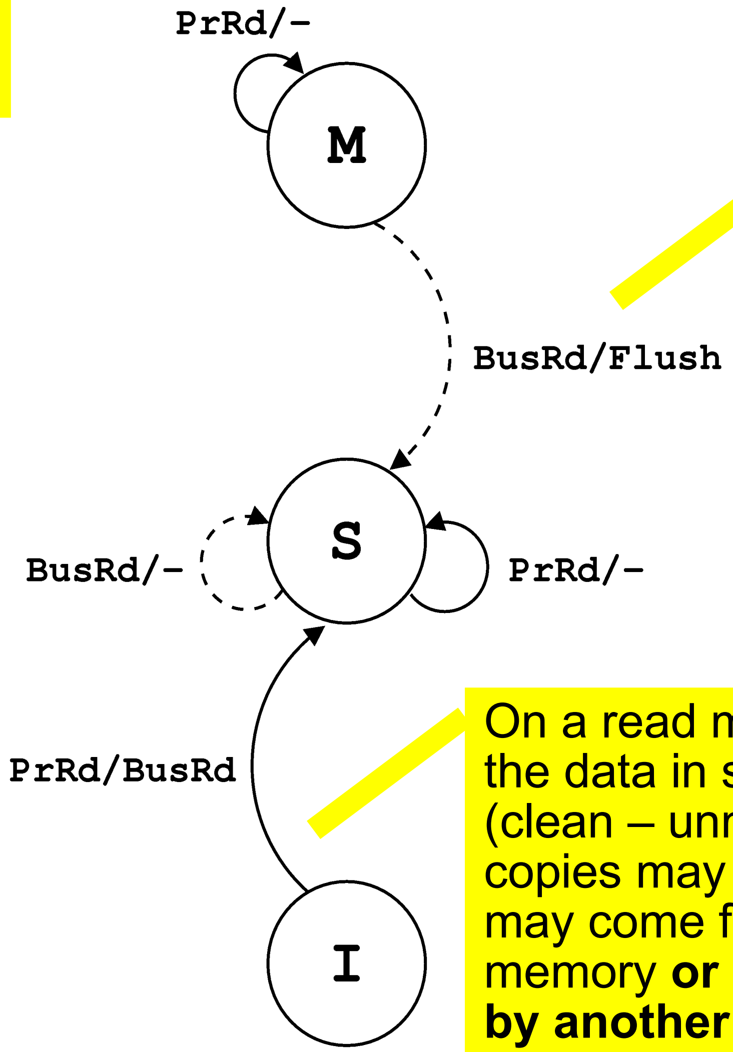
- Cache line states:
  - (S)hared
    - The block is present in an unmodified state in this cache
    - Main memory is up-to-date
    - Zero or more up-to-date copies exist in other caches
  - (M)odified
    - Only this cache has a valid copy, the copy in memory is stale



Only copy of block

MSI write-back invalidate protocol

Let's build the protocol up in stages...

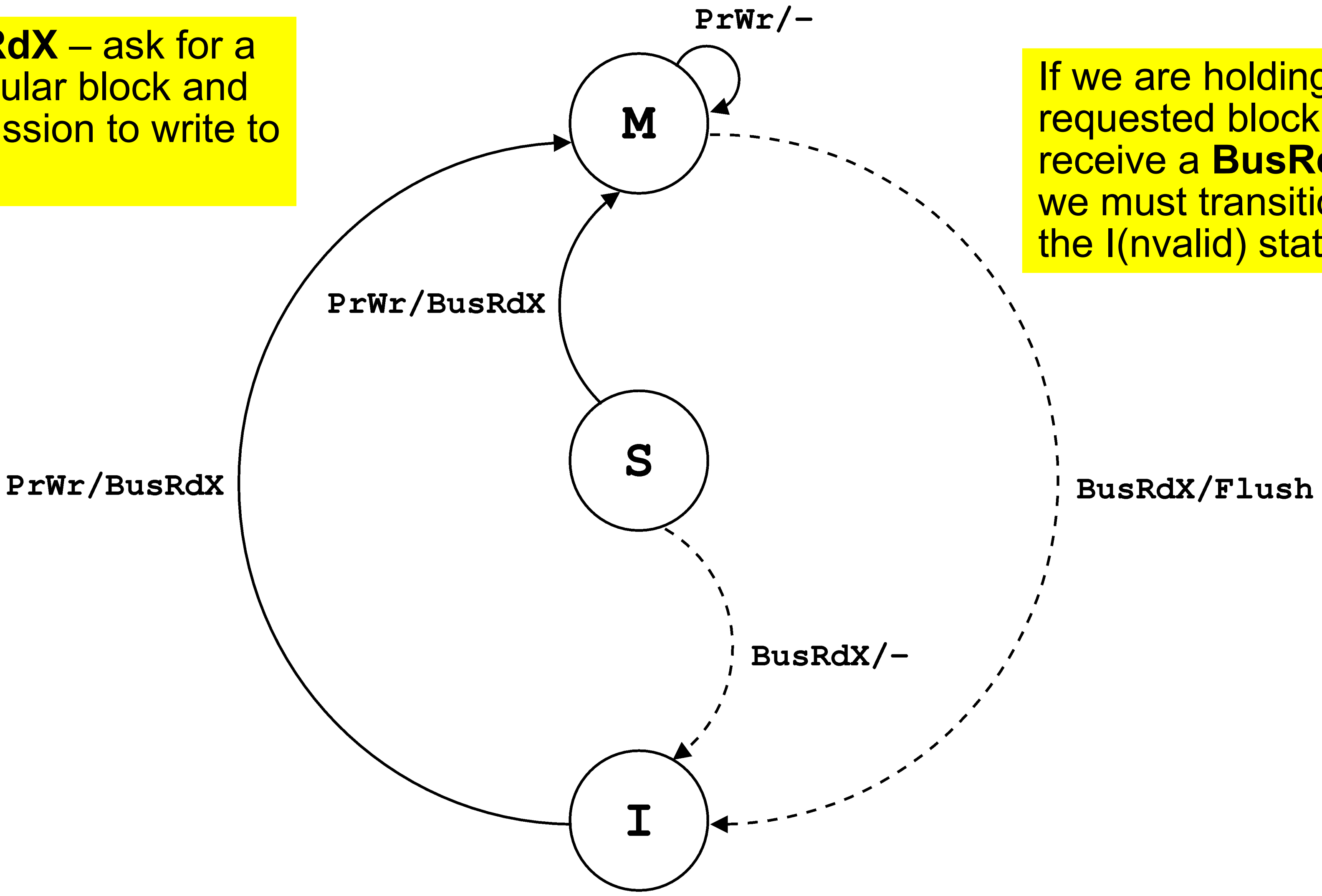


If another cache needs the block that we have in state M (the block is dirty/modified), we must flush the block to memory and provide the data to the requesting cache (over the bus)

On a read miss, we get the data in state S (clean – unmodified, other copies may exist). Data may come from main memory **or be provided by another cache**

MSI write-back invalidate protocol

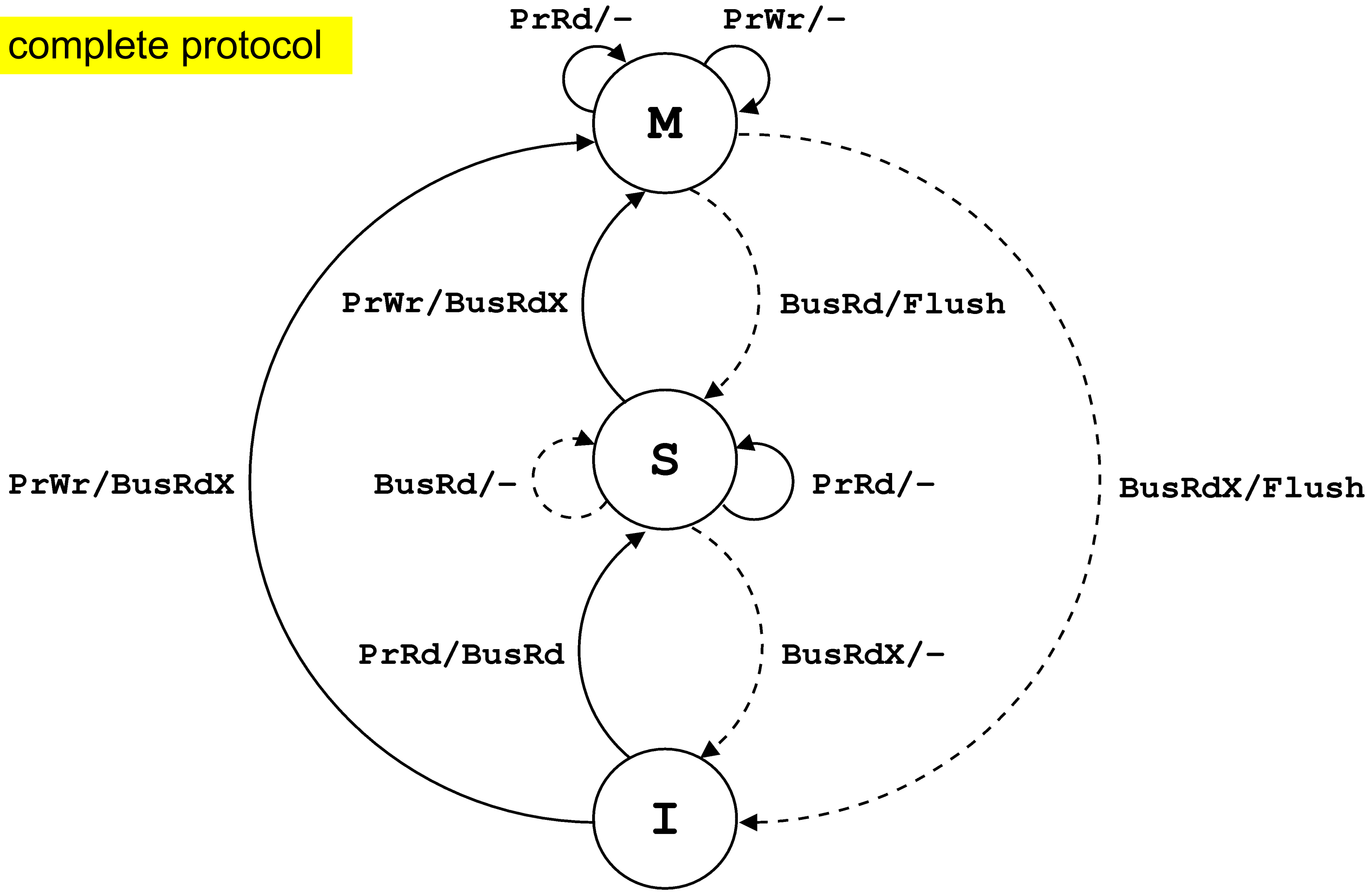
BusRdX – ask for a particular block and permission to write to it.



If we are holding the requested block and receive a **BusRdX**, we must transition to the I(nvalid) state

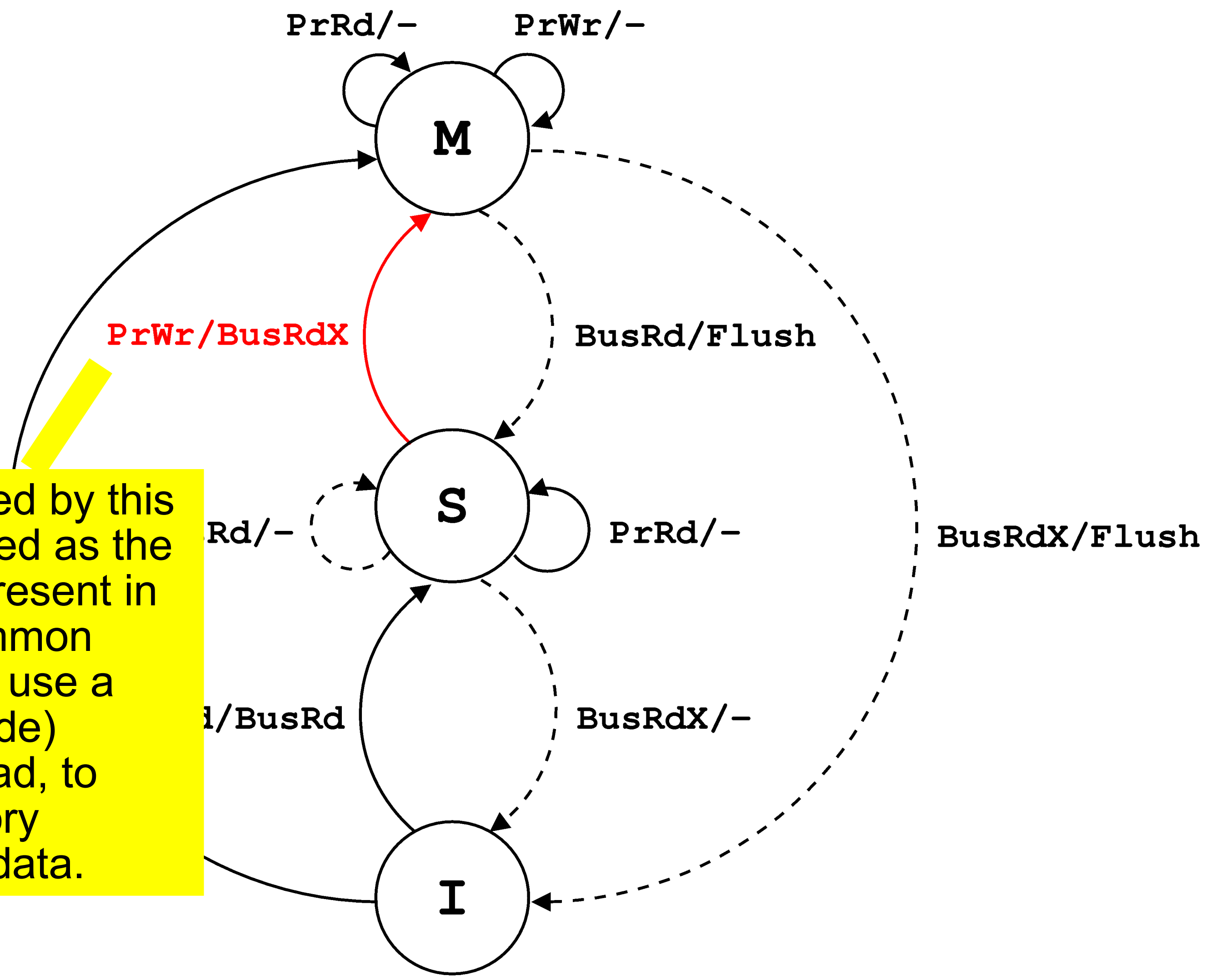
MSI write-back invalidate protocol

The complete protocol



Demo!

MSI write-back invalidate protocol

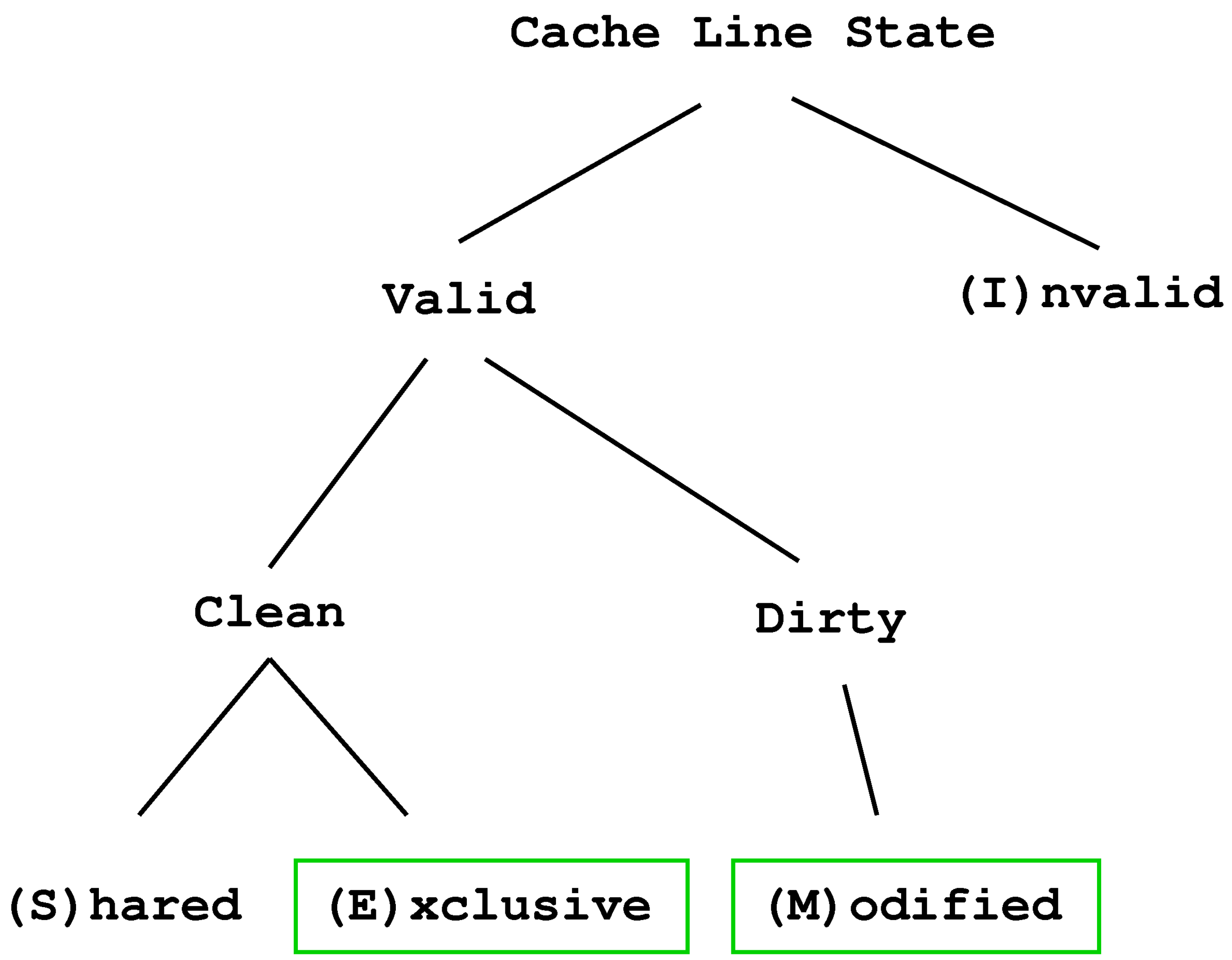


The data produced by this **BusRdX** is ignored as the data is already present in the cache. A common optimisation is to use a **BusUpgr** (upgrade) transaction instead, to save main memory responding with data.

MESI protocol

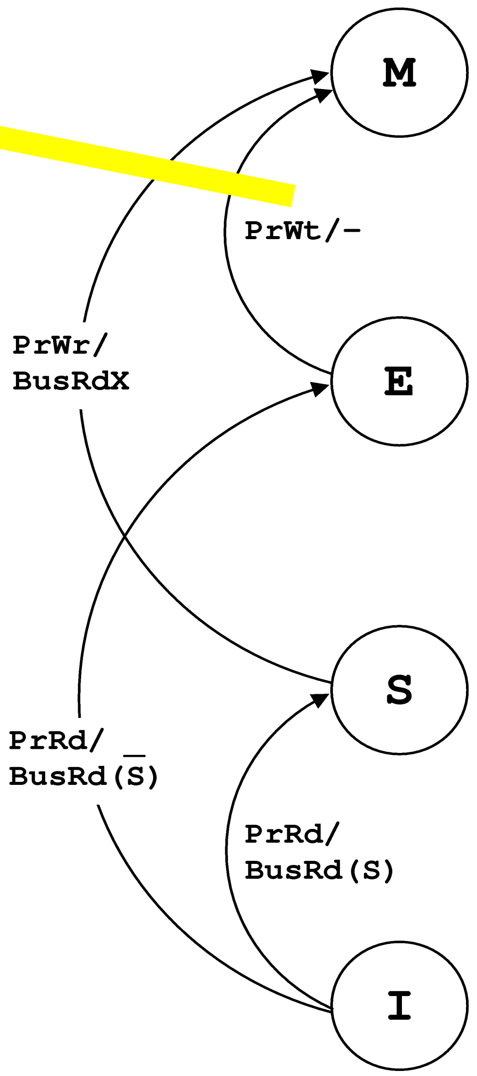
- Let's consider what happens if we read a block and then subsequently wish to modify it
  - This will require two bus transactions using the 3-state MSI protocol
  - But if we know that we have the only copy of the block, the transaction (**BusUpgr**) required to transition from state S to M is really unnecessary
    - We could safely, and silently, transition from S to M
  - This will be a common event in parallel programs and especially in sequential applications
    - Very common to see sequential apps. running on CMP

MESI protocol



MESI protocol

If we are in state E and need to write, we require no bus transaction to take place

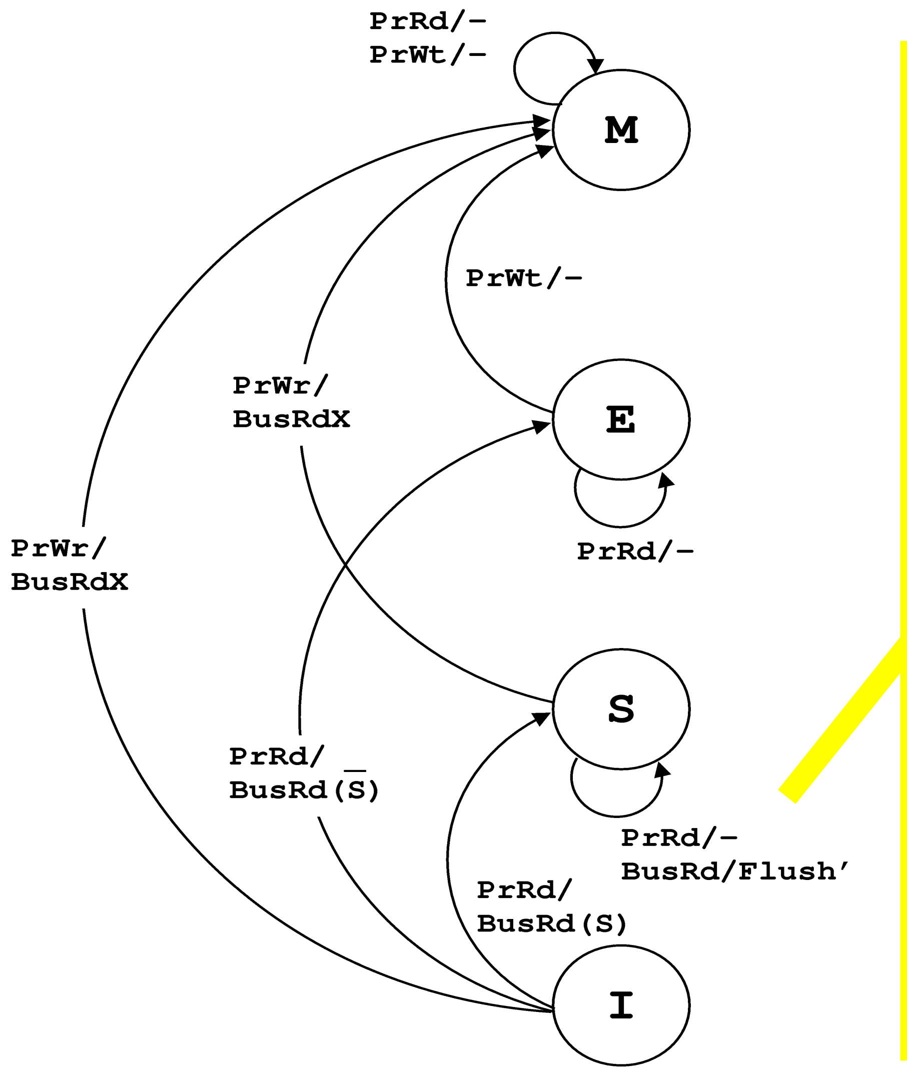


The shared signal (S) is used to determine if any caches currently hold the requested data on a PrRd.

BusRd(S) means the bus read transaction caused the shared signal to be asserted (another cache has a copy of the data). **BusRd(Not S)** means no cache has the data.

This signal is used to decide if we transition from state I to S or E

MESI protocol

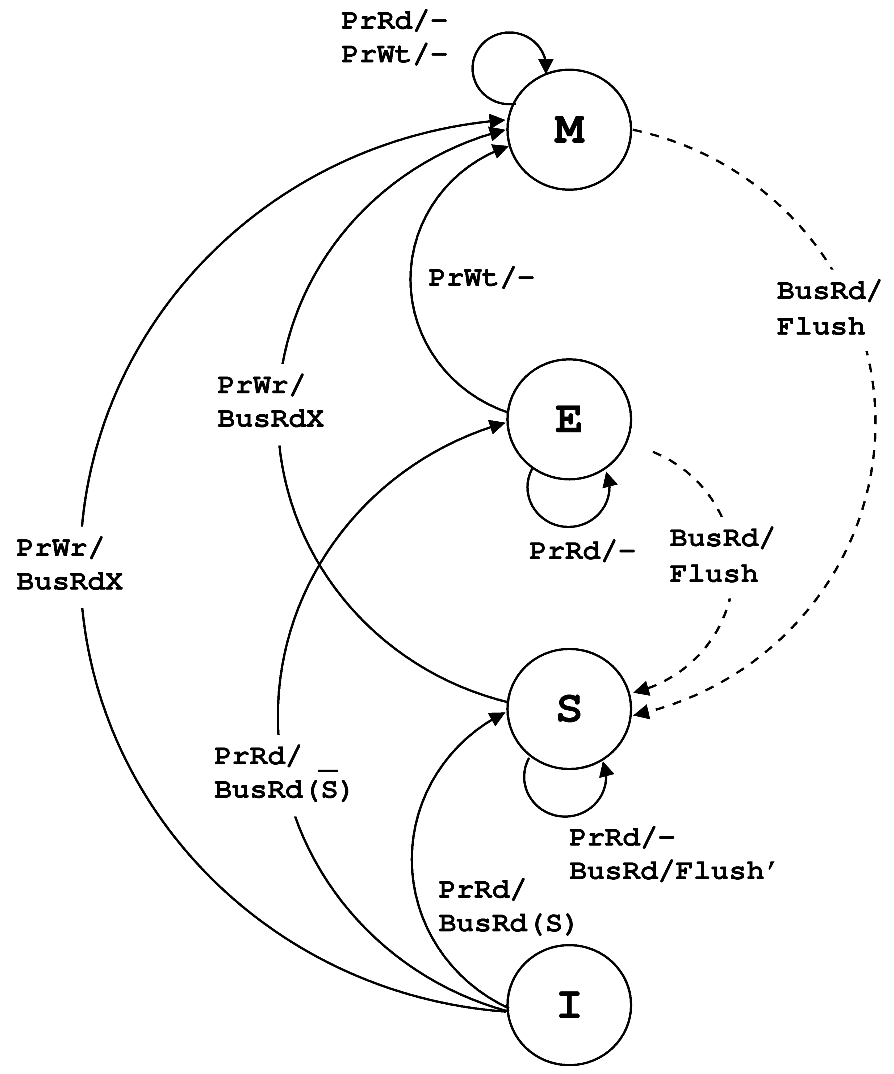


BusRd/Flush'

To enable cache-to-cache sharing we must be sure to select only one cache to provide the required block. Flush' is used to indicate that only **one** cache will flush the data (in order to supply it to the requesting cache). This is easy in a bus-based system of course as only one transaction can take place at a time.

Of course a cache will normally be able to provide the block faster than main memory.

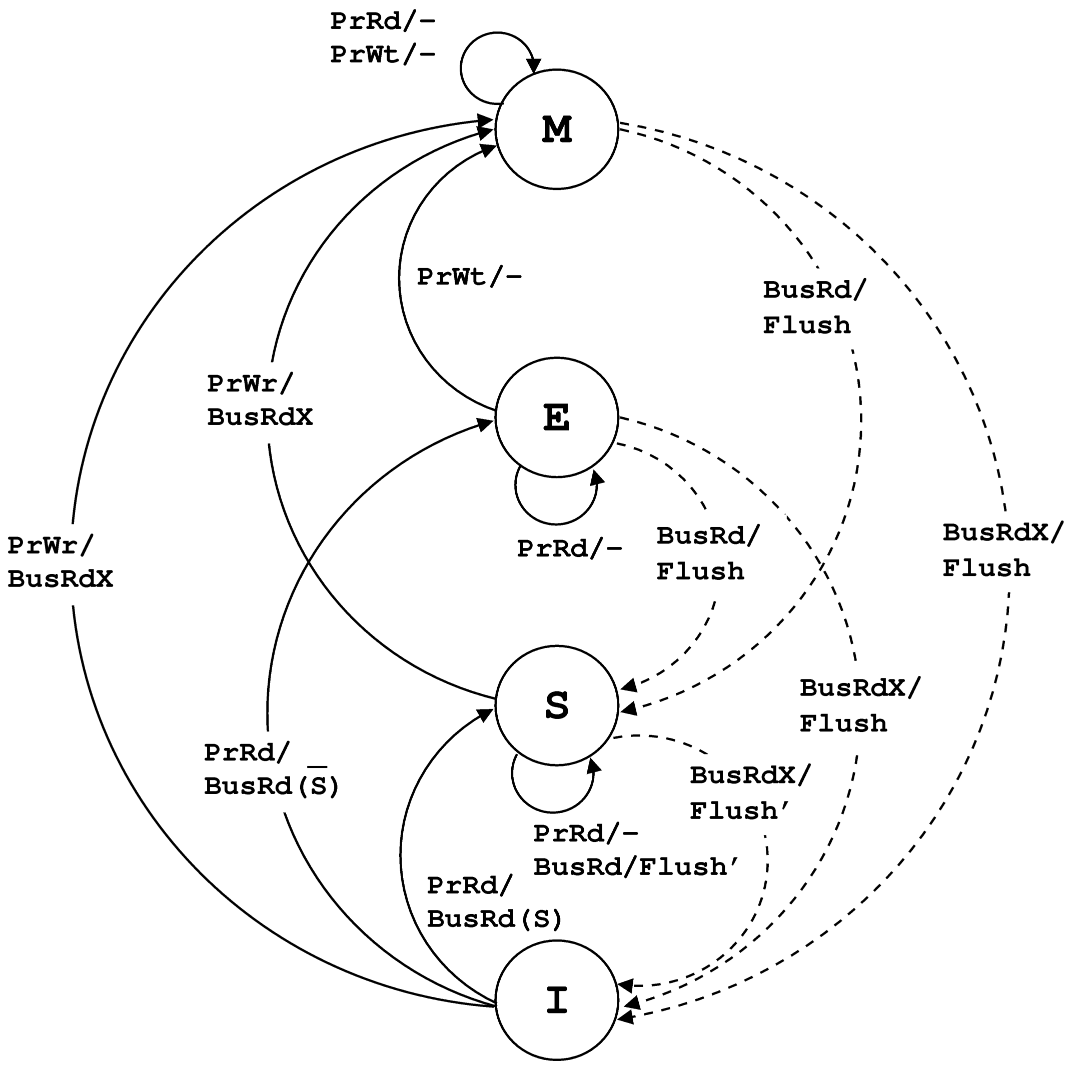
MESI protocol



BusRd/Flush

If we are in state M or E and other cache requests the block, we provide the data (we have the only copy) and move to state S (clean & zero or more copies)

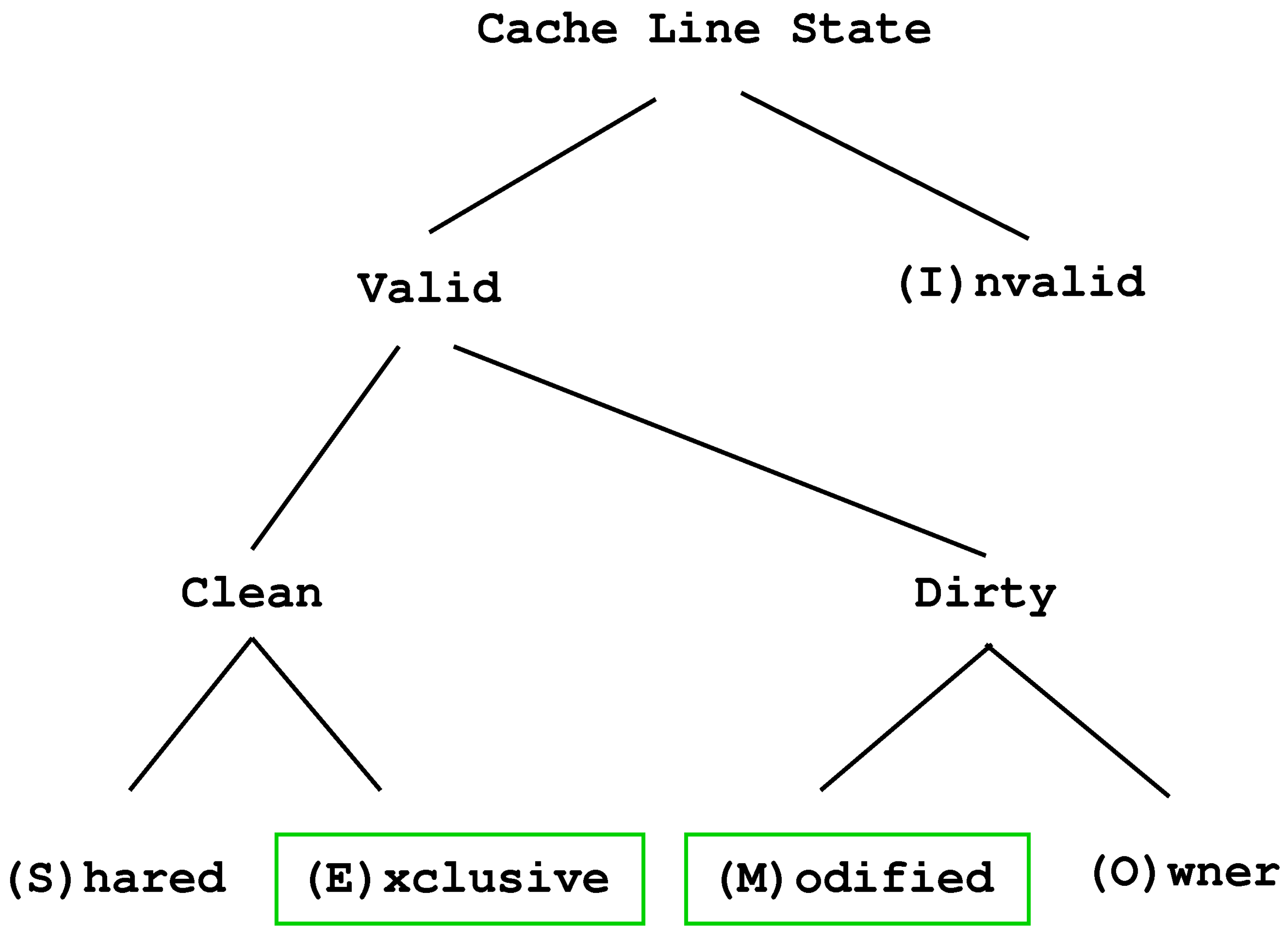
MESI protocol



BusRdX/Flush

If we are in states M, E or S and receive a BusRdX we simply flush our data and move to state I

MOESI protocol



MOESI protocol

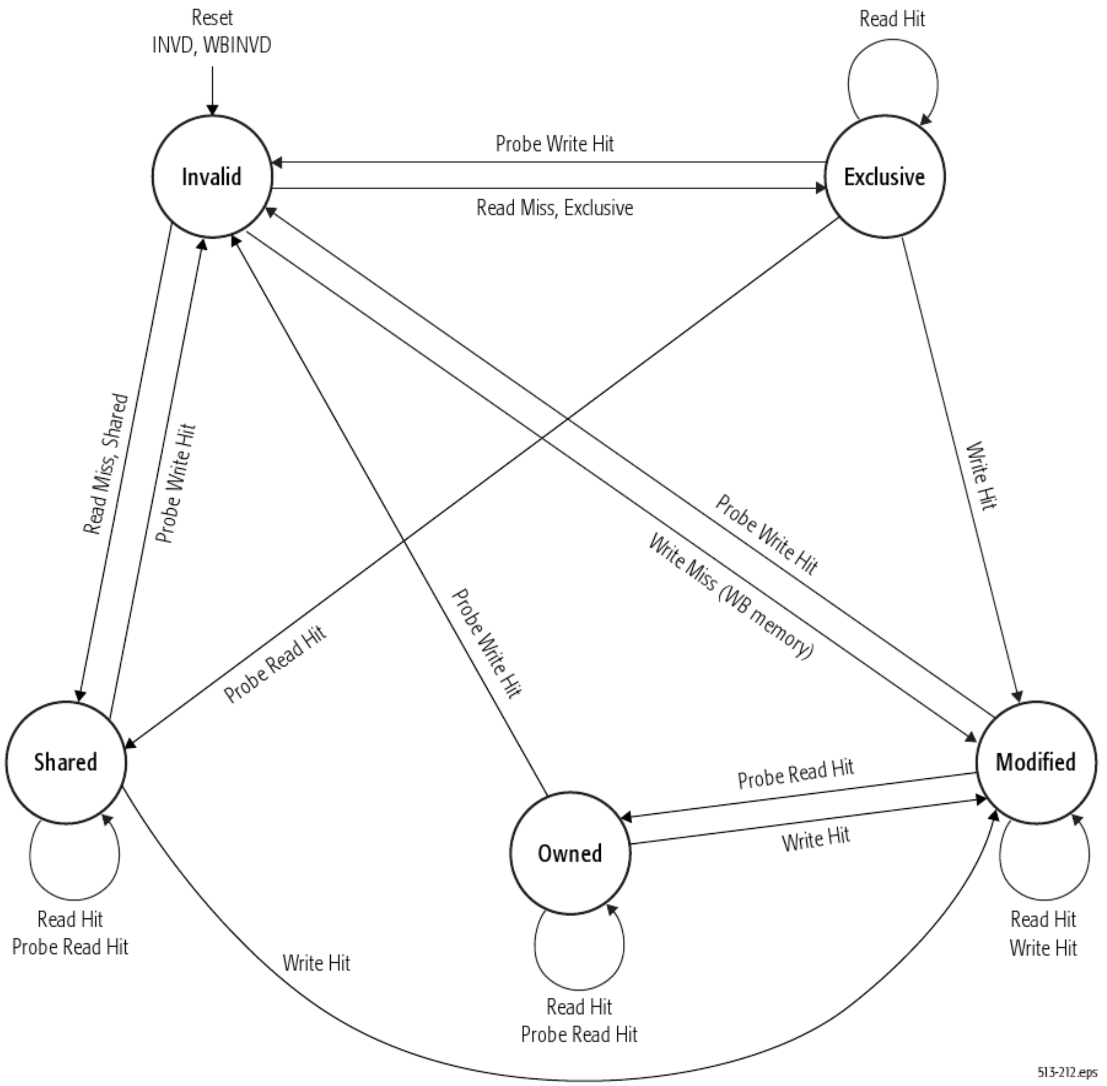
- O(wner) state
  - Other shared copies of this block exist, but memory is stale. This cache (the owner) is responsible for supplying the data when it observes the relevant bus transaction.
  - This avoids the need to write modified data back to memory when another processor wants to read it
    - Look at the M to S transition in the MSI protocol
- A summary of the cache states and an annotated transition diagram for the MSI protocol (hopefully useful “cheat” sheets) are available from the course wiki

MOESI protocol

AMD MOESI protocol state transition diagram

Reproduced from "AMD64 Architecture Programmer's Manual, Volume 2: System Programming" p.168

You can think of
probe write hit as BusRdX
probe read hit as BusRd
(the actions taken are omitted from the transitions in this diagram)



Exercise

Processor Action	P1 state	P2 state	P3 state	Bus Action	Data supplied from
P1 read x	S	--	--	BusRd	Memory
P2 read x					
P1 write x					
P1 read x					
...					

- Add some more processor actions of your own and complete this table for the MSI and MESI protocols we have discussed.

Update protocols

- On a write we have two options
  - (1) Invalidate cache copies
    - We have looked at invalidate-based protocols so far
  - (2) Update the cache copies with the new data
    - These are unsurprisingly called “update protocols”

Update protocols

- Update protocols keep the cached copies of the block up to date.
  - Low-latency communication
  - Conserve bandwidth, we only need one update transaction to keep multiple sharers up to date
  - Unnecessary traffic when a single processor writes repeatedly to a block (and no other processor accesses the block)

Dragon update protocol

- Dragon protocol (4-state update protocol)
  - Dragon multiprocessor, Xerox, Thacker *et al*, '84
  - Culler book p.301
  - Note: there is no (**I**) state, blocks are kept up-to-date

  - **(E)** Exclusive-Clean
    - clean, only-copy of data

  - **(Sc)** Shared-clean
    - two or more caches potentially have a copy of this block
    - main-memory main or may not be up-to-date
      - The block may be in state Sm in one other cache

Dragon update protocol

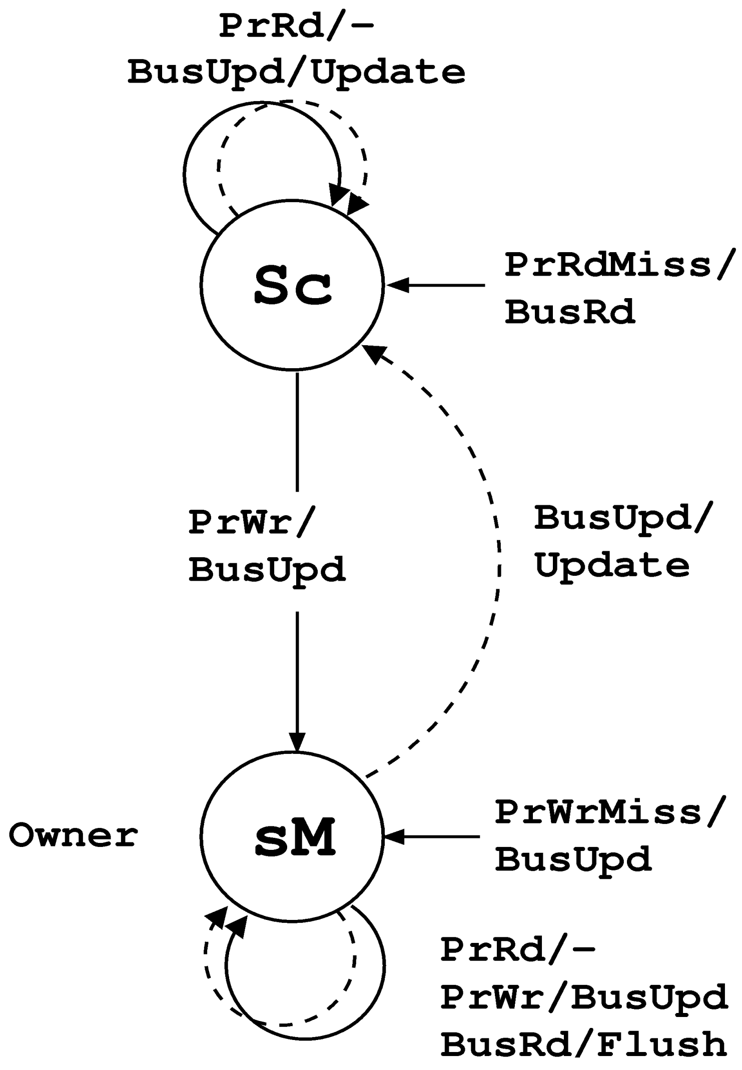
- **(Sm)** Shared-modified
    - two or more caches potentially have a copy of this block
    - main-memory is not up-to-date
    - it is this cache's responsibility to update memory (when block is replaced)
        - A block can only be in state Sm in one cache at a time
- **(M)** modified (dirty)
    - Only this cache has a copy, the block is dirty (main-memory's copy is stale)

Dragon update protocol

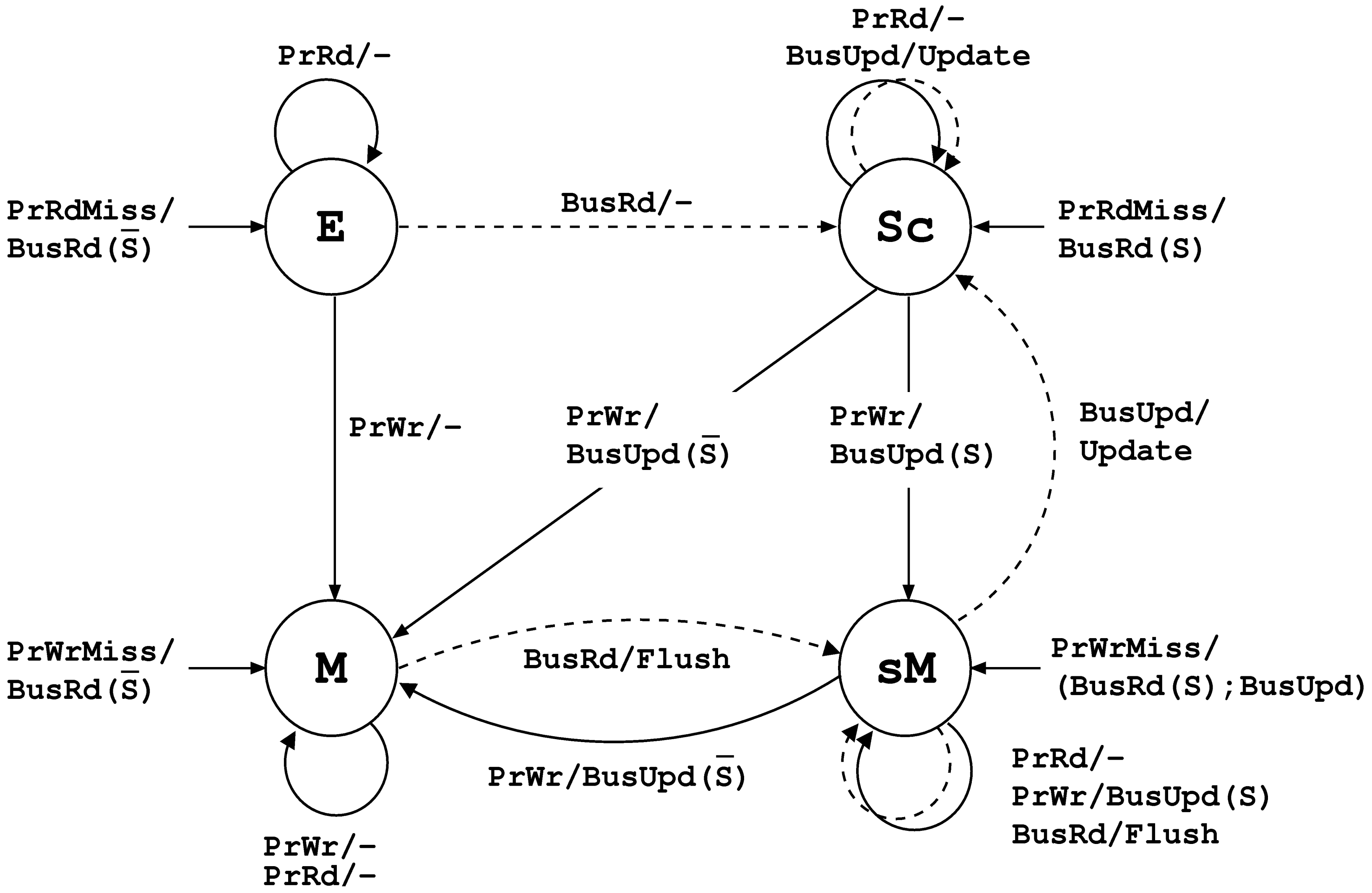
- Since we no longer have an (I)nvalid state we distinguish between a cache miss and cache hit:
  - **PrRdMiss** and **PrRd**
  - **PrWrMiss** and **PrWr**
- We also need to support **BusUpd** (Bus Update transactions) and **Update** (update block with new data) actions

Simple update protocol

- Simple 2-state update protocol
- We just need to remember who “owns” the data (the writer)
- Superfluous bus transactions can be removed by adding states for when we hold the only copy of the data
  - E – Exclusive (clean)
  - M – Modified (dirty)



Dragon update protocol



Update protocols

- Update vs. invalidate
  - Depends on sharing patterns
    - *e.g.* producer-consumer pattern favours update
    - Update produces more traffic (scalability and energy cost worries)
    - Could dynamically choose best protocol at run-time or be assisted by compiler hints
  - Recent work
    - In-network cache coherency work from Princeton
    - Analysis of sharing patterns and exploitation of direct cache-to-cache accesses (Cambridge)

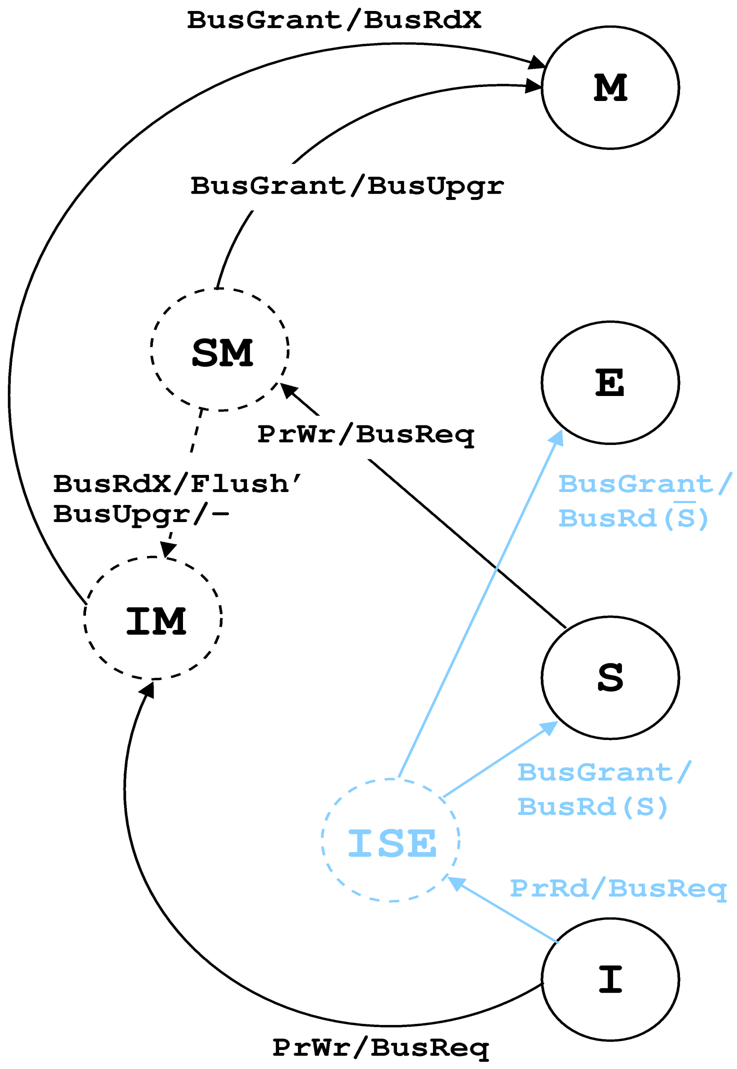
Transient (intermediate) states

- Nonatomic state transitions
  - Bus transactions are atomic, but a processor's actions are often more complex and must be granted access to bus before they can complete
  - What happens if we snoop a transaction that we must respond to while we are waiting for an outstanding request to be acted upon?
    - *e.g.* we are waiting for access to the bus

Transient (intermediate) states

Processor Action	P1 state	P2 state	Bus Action
....	S	S	
P1/P2 write x (both need to perform a BusUpgr – slide 27)			P2 wins bus arb. **P2 issues a BusUpgr**
P1 needs to downgrade state of X to I(nvalid)	I	M	
P1 must now issue a BusRdX (not a BusUpgr as orginally planned) Doesn't require a new BusReq	.....		**BusRdX**
...			

Transient (intermediate) states



Note: This figure omits the transitions between the normal states (see Culler book, p.388 for complete state diagram)

Split-transaction buses

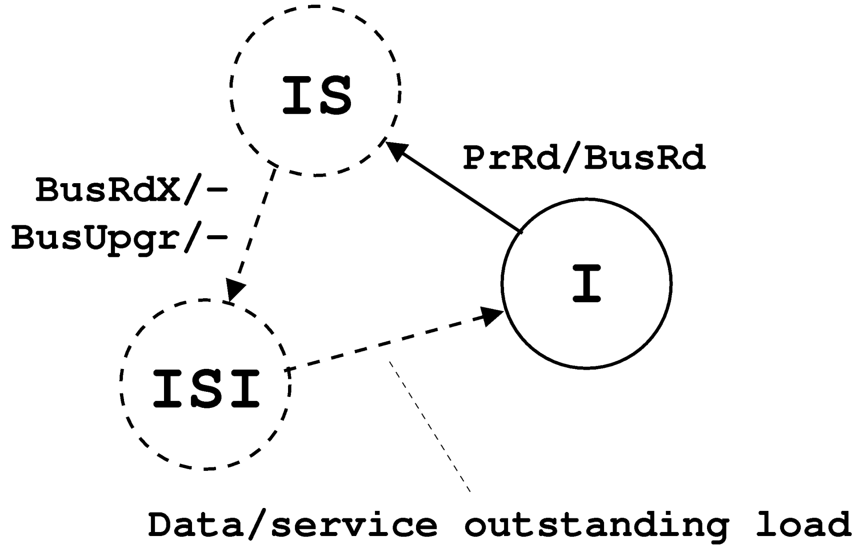
- An atomic bus remains idle between each request and response (*e.g.* while data is read from main memory)
- Split-transaction buses allow multiple outstanding requests in order to make better use of the bus
  - This introduces the possibility of receiving a response to a request after you have snooped a transaction that has resulted in a transition to another state
- Culler Section 6.4

Split-transaction buses

- Example:
  - P1: I->S
    - Issued **BusRd** and awaiting response (in state IS)
      - The IS state records the fact that the request has been sent
  - P2: S->M
    - Issued an invalidate, P1 sees this before the response to its **BusRd**
  - P1:
    - P1 can't simply move to state I, it needs to service the outstanding load.
      - Don't want to reissue BusRd either as this may prevent forward progress being made
    - Answer move to another intermediate state (ISI) wait for original response, service load, then move to state I

Split-transaction buses

- ISI state:
  - Wait for data from original BusRd
  - Service the outstanding load instruction
  - Move to the I(nvalid) state



Bus-based cache coherency protocols

- In a real system it is common to see 4 stable states and perhaps 10 transient states
  - Design and verification is complex
    - Basic concept is relatively simple
    - Optimisations introduce complexity
  - Bugs do slip through the net
    - Problematic for correctness and security
    - Intel Core 2 Duo
      - A139: "*Cache data access request from one core hitting a modified line in the L1 data cache of the other core may cause unpredictable system behaviour*"