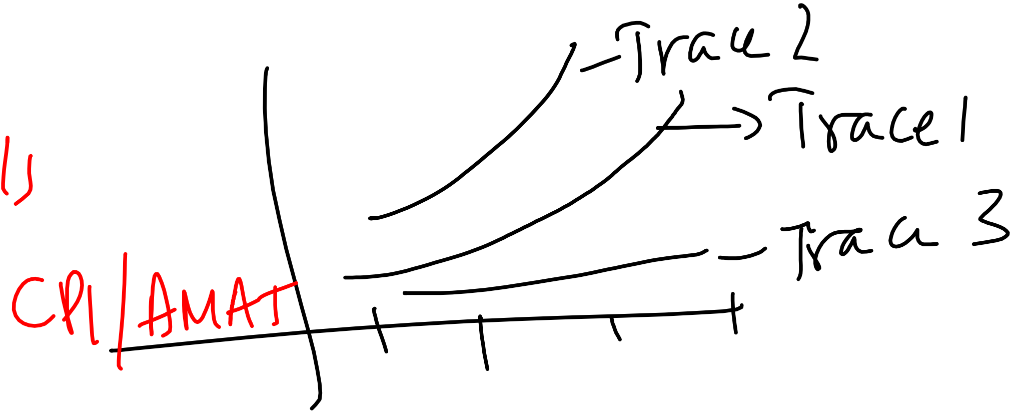
✓Caches →

→ Multicore protocols

Q+k



→ BPrediction

→ Exceptions

→ Unrolling

→ ☆ Virtual memory

Multicore architecture
Cache coherency

CA- QA – H&P – Ch 5

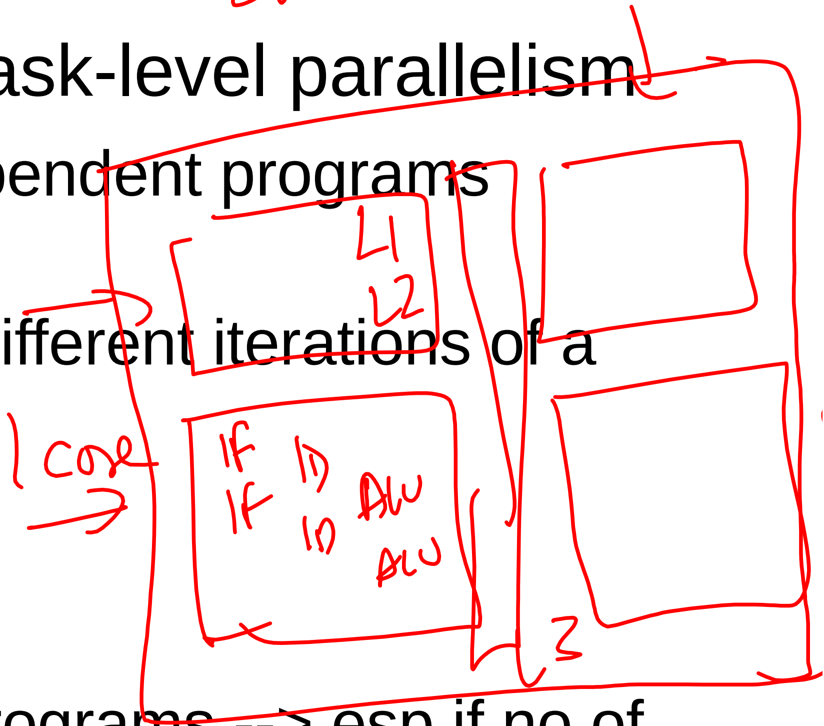
Motivation

LW
LW
SW
SW

4-way

4-core.

- Better performance
- ILP --> Thread level parallelism/ Task-level parallelism
  - Use multiple processors to run independent programs simultaneously
  - Or tasks within the same program- different iterations of a loop
- Multicore processor
  - Multiple cores on the same die
  - Difficult to create parallel software programs --> esp if no of cores increase

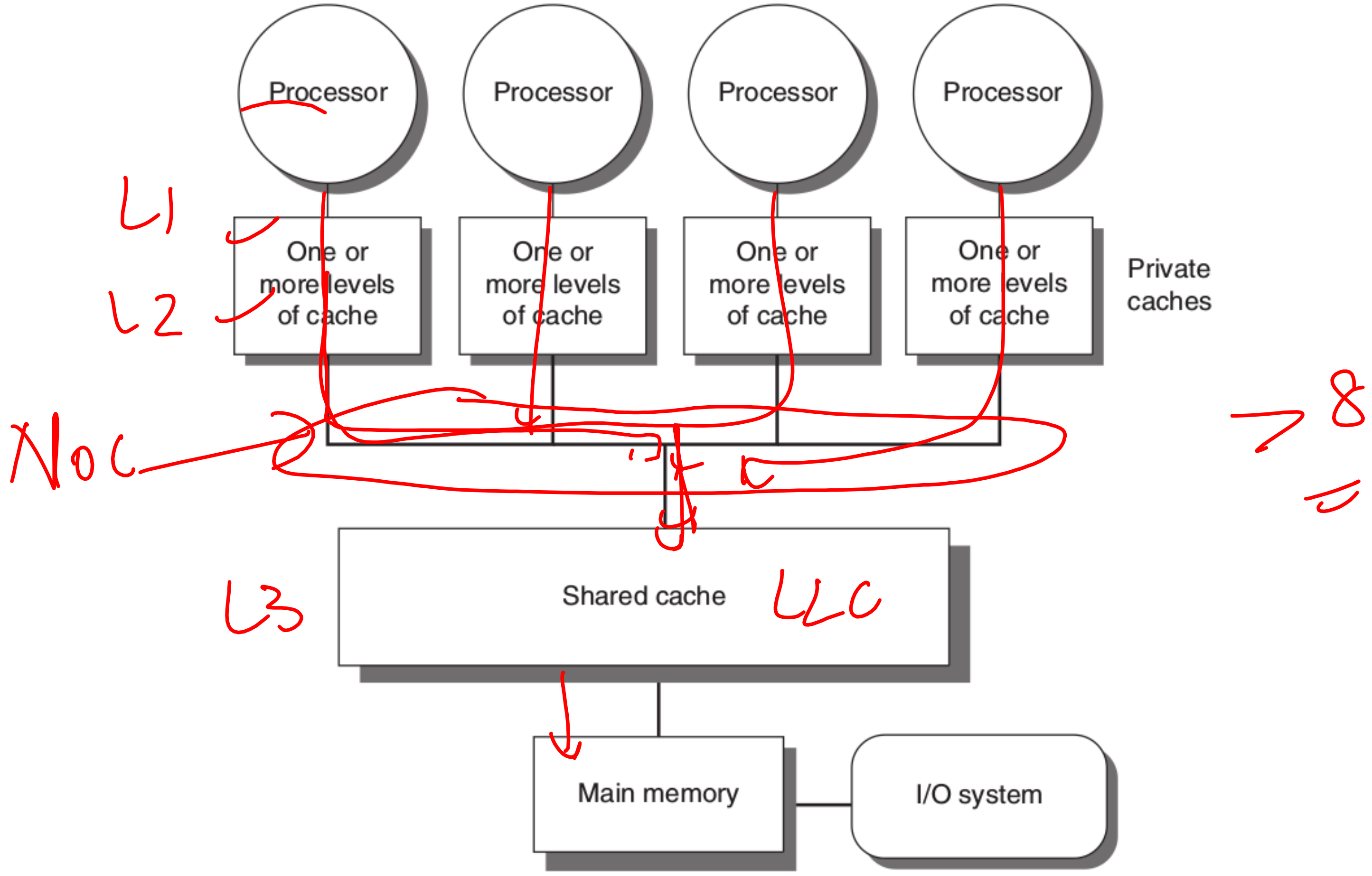


Challenges

- Superscalar and OOO on a single processor use ILP --> No user intervention
- Multicore-->
  - Programs should be written to exploit parallelism
  - Tasks should be broken down into parallel small tasks
  - Balance the load
  - Communicate between tasks, synchronization
  - Compiler exploits data level parallelism and generates parallel threads

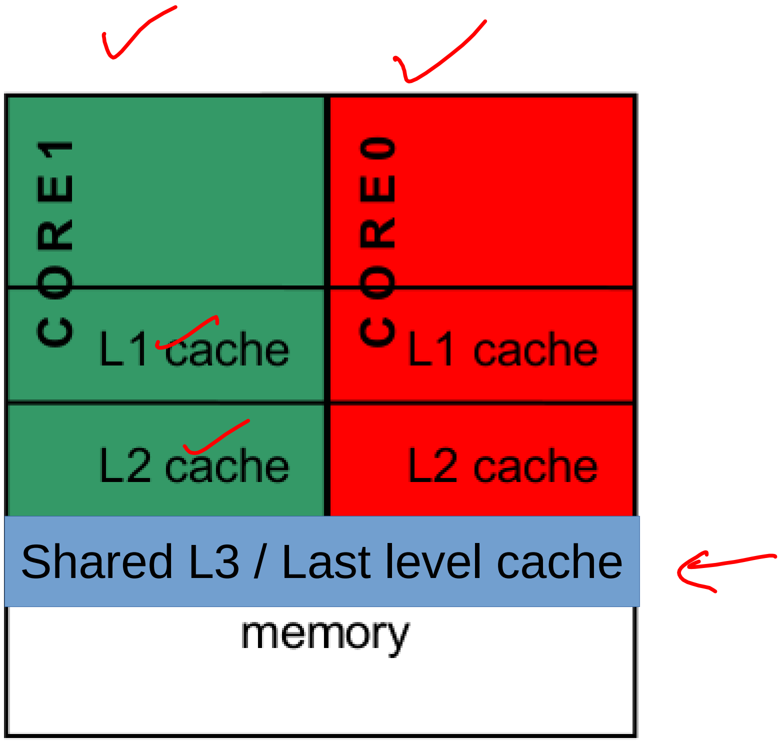
Shared memory processors (SMP)

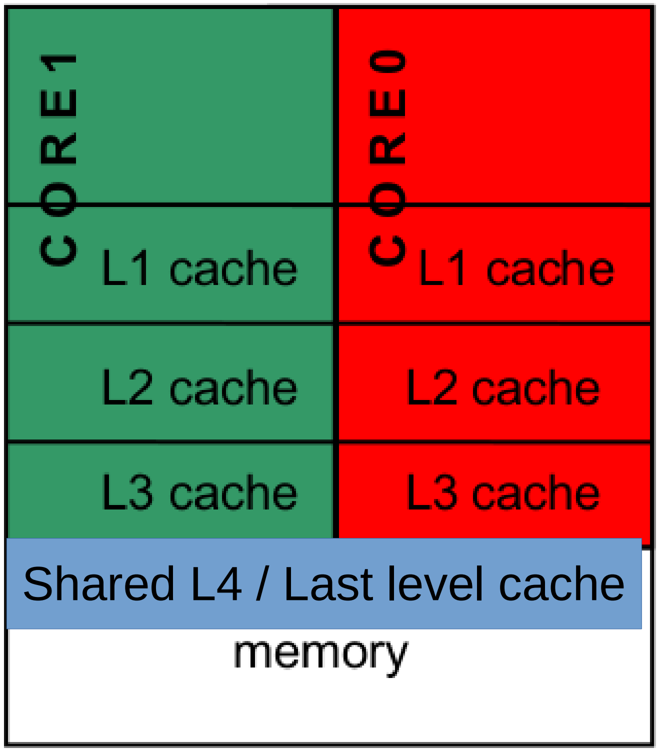
Core


Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Private caches
Shared cache
Main memory
I/O system
L1
L2
NoC
L3
LLC
> 8

- Upto 8 cores
- Share same memory address space (centralized memory architectures)
  - Uniform memory architectures (UMA)
  - All processors have the same latency from memory

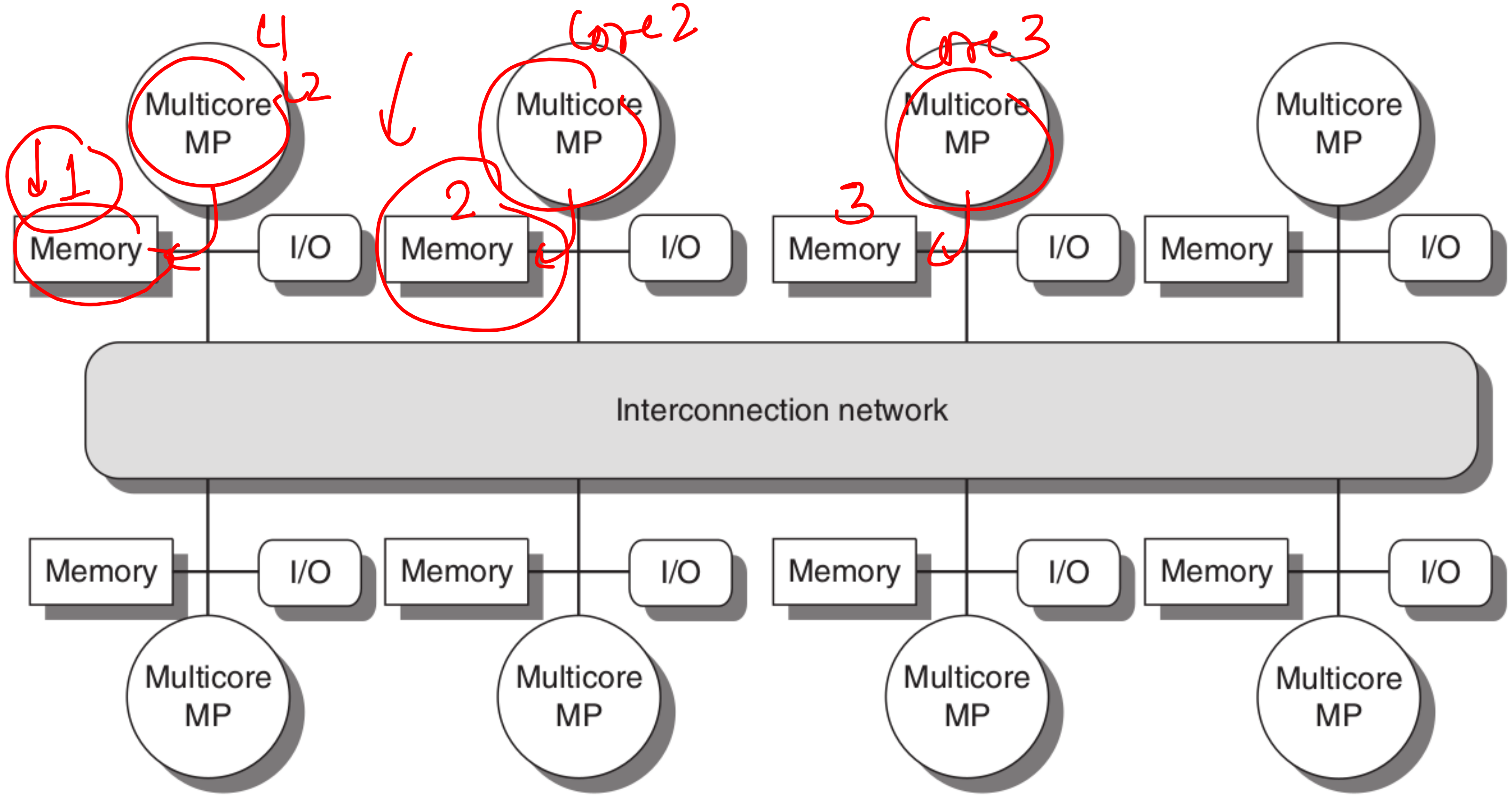
Architectures





Local data: Local data used only by that core- can be in the local cache
Shared data: May be replicated in multiple caches

Distributed memory processors



- More cores > 8. Share same memory address space.
- Individual memory with processors, to reduce long memory access latency with several cores
- Non-uniform memory access (NUMA)
- Focus on interconnect network design
- Communication between processors is a challenge

Challenges in multi-cores

- Insufficient parallelism in programs
  - Write better algorithms or programs

- Long latency communications between processors
  - Architectural changes: Caches can store shared data: How do we maintain coherence between multiple copies
    - Local data (Not shared)
    - Shared data
  - Consistent view of memory
  - Multi-threading
  - Prefetching

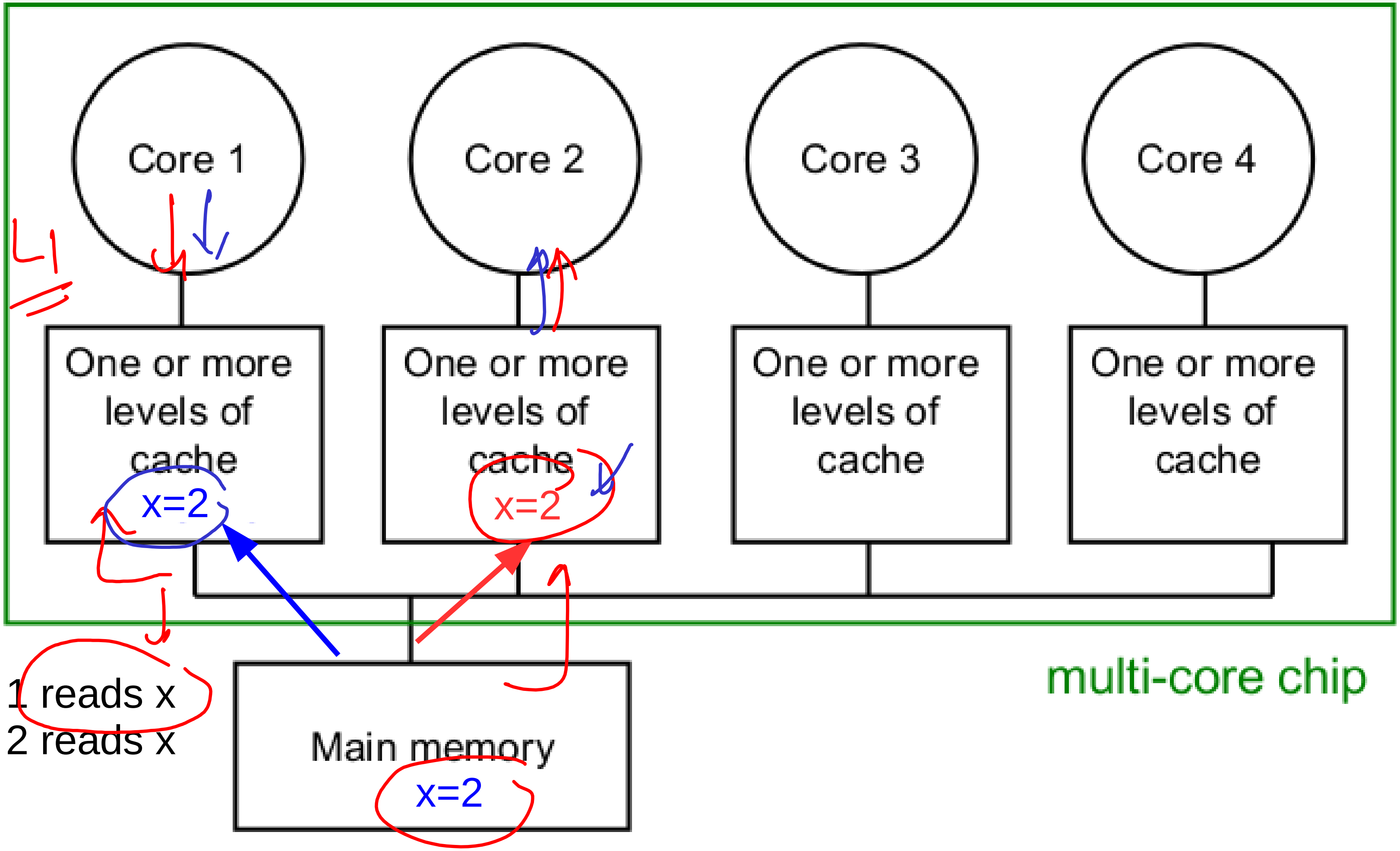
Shared data

- Memory system is coherent if any read of a data item returns the most recently written value of that data item.
- Coherence: What is the value returned by a read
- Consistent: When is the value returned by the read

Cache coherence in SMP

Caching of shared data/address space

Reads


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
x=2
x=2
Main memory
x=2
L1
multi-core chip

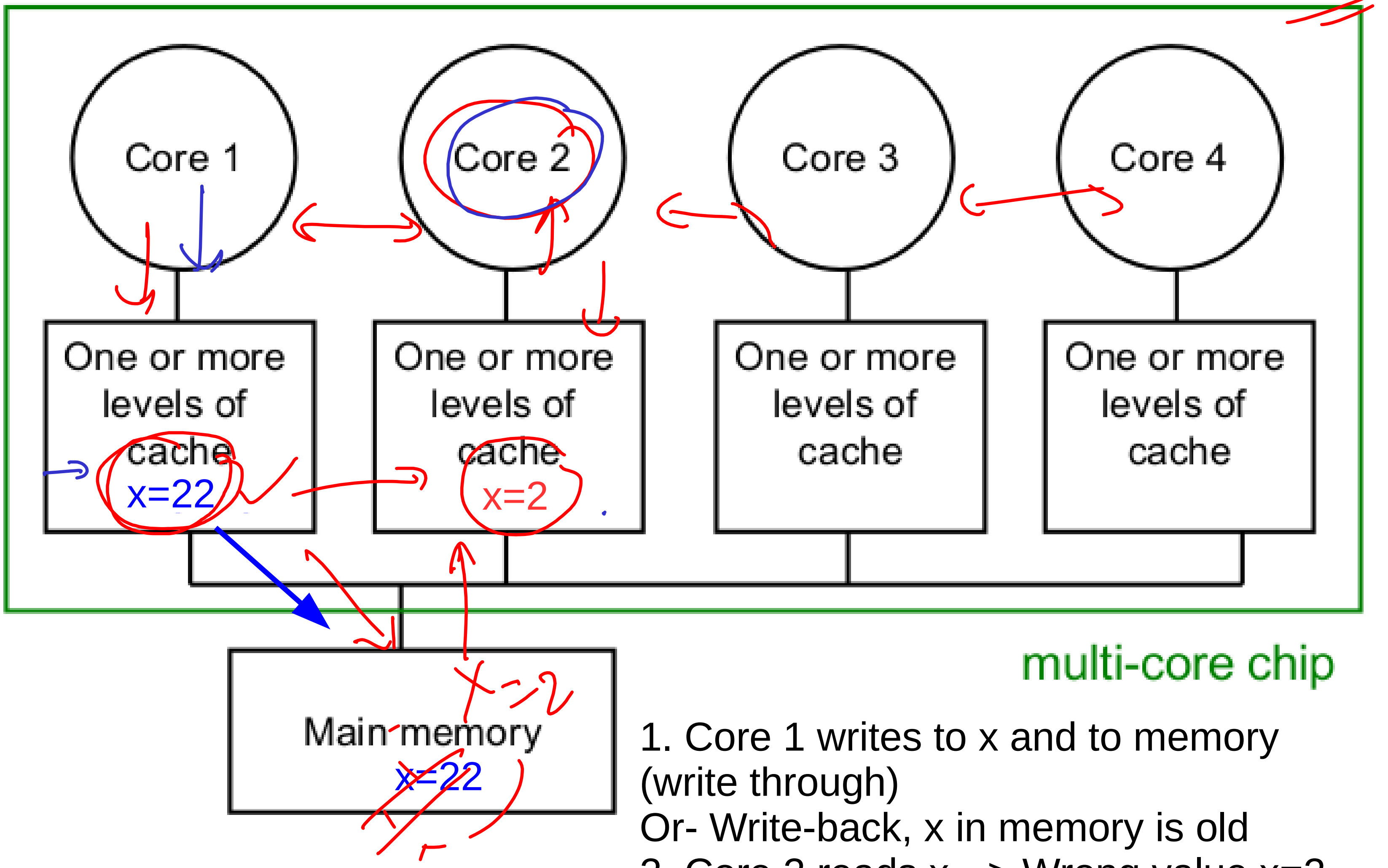
1. Core 1 reads x
2. Core 2 reads x

Cache coherence

Cache writes

Coherence: Processors read the latest value

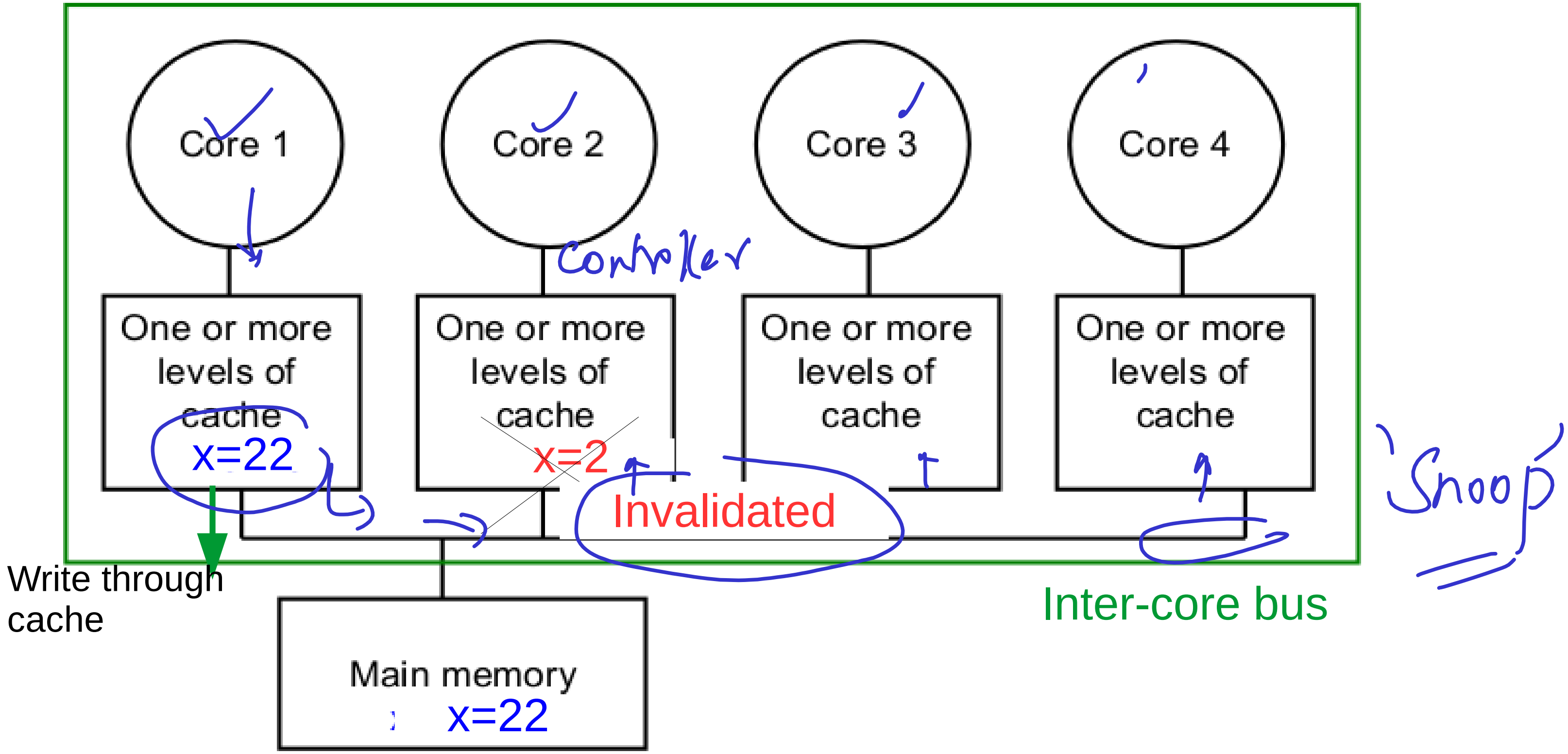
FSM



Core 1
Core 2
Core 3
Core 4
One or more levels of cache
x=22
One or more levels of cache
x=2
One or more levels of cache
One or more levels of cache
multi-core chip
Main memory
x=22
x=2

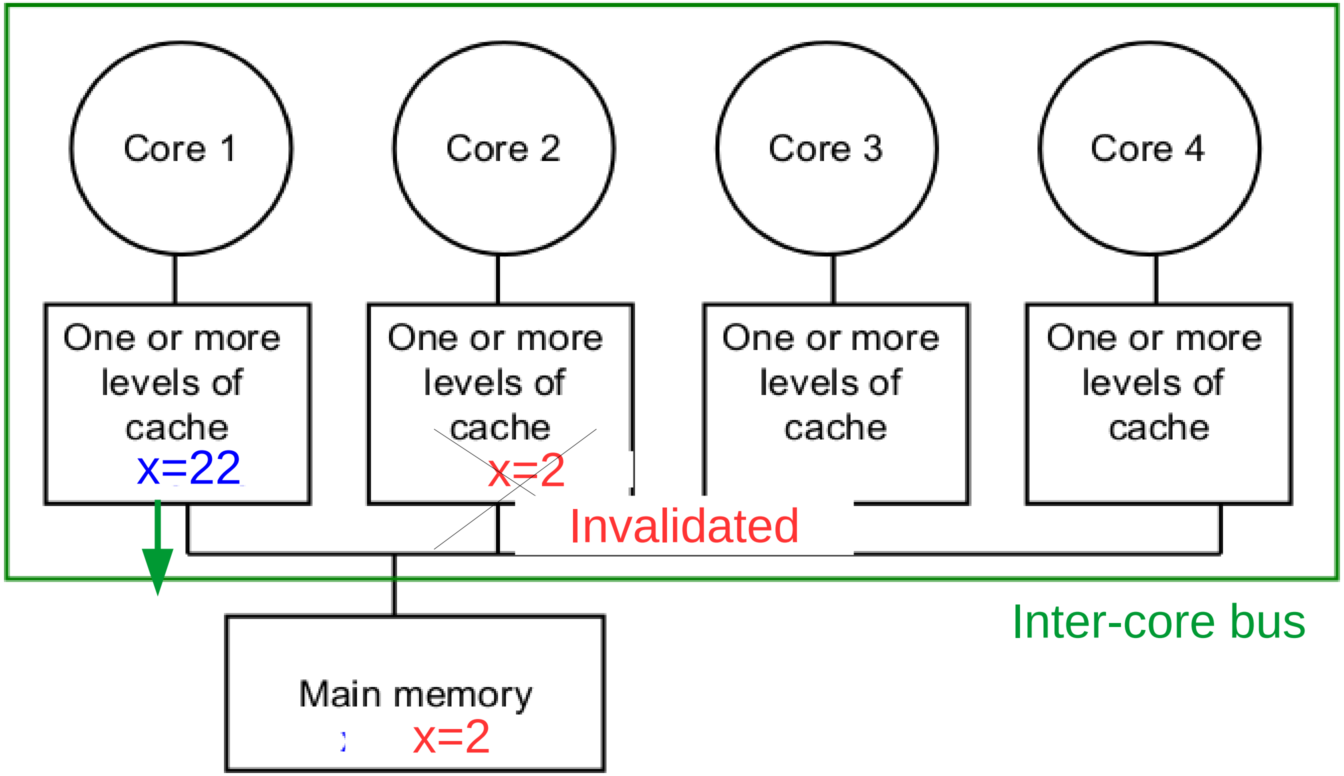
1. Core 1 writes to x and to memory (write through)
Or- Write-back, x in memory is old
2. Core 2 reads x --> Wrong value x=2

Option 1: Invalidation protocol


Core 1
Core 2
Core 3
Core 4
controller
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
x=22
x=2
Invalidated
Snoop
Write through cache
Inter-core bus
Main memory
x=22

1. Core 1 gains control of the bus
2. Sends invalidation request on the bus --> copies of this block in other caches are invalidated (this should be done before writing. Else other cores might write to x)
3. All cores continuously “snoop” (monitor) the bus. Each core’s cache controller invalidates its local copy
4. Core 1 has exclusive write permission to x and to memory (write through). Write x.
5. Core 2 reads x --> Cache miss--> loads new value from memory

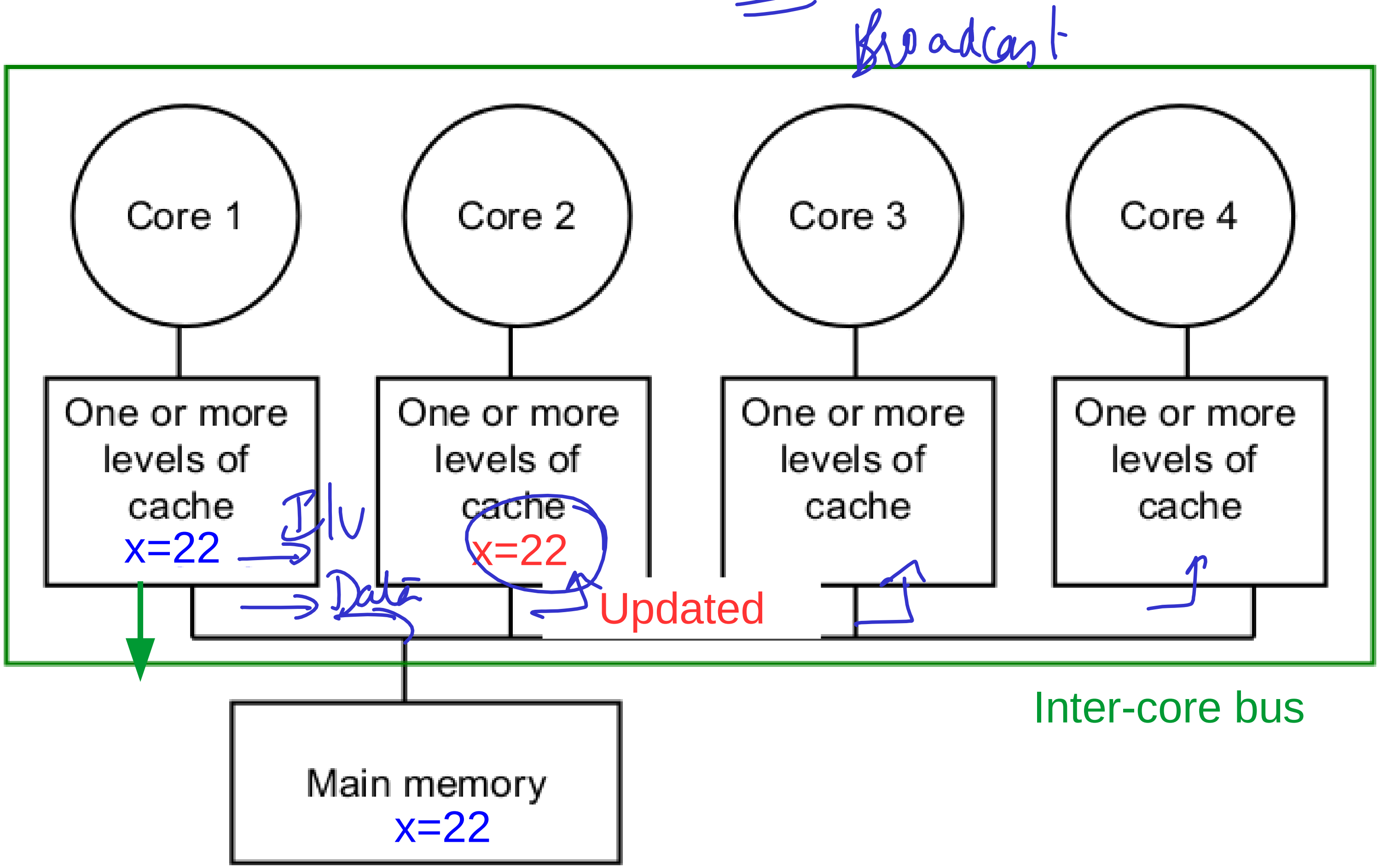
Write back: Invalidation protocol


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
x=22
One or more levels of cache
x=2
Invalidated
One or more levels of cache
One or more levels of cache
Main memory
x=2
Inter-core bus

1)Write back cache
2)Core 2- read request --> All other cores check if they have a local copy whose dirty bit=1
3)Read from memory is terminated.
4)Core 1 places the data on the bus, Core2 reads it.
5)Core 1 writes to memory when the line is evicted

Option 2: Update protocol

Broadcast

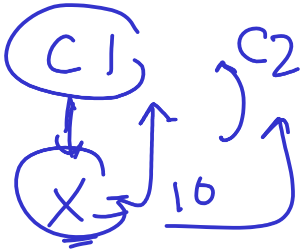

Core 1
Core 2
Core 3
Core 4
One or more levels of cache
x=22
I/U
Data
One or more levels of cache
x=22
Updated
One or more levels of cache
One or more levels of cache
Inter-core bus
Main memory
x=22

1. Core 1 writes to x and to memory (write through)
2. Core 1 sends the updated value on the bus (Broadcast)
3. All cores containing the data block update the value
4. Core 2 reads x --> new value

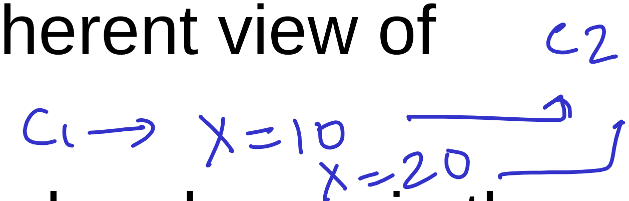
Invalidation vs Update

	Update	Invalidate
Writes from a core to the same address	Broadcast each write and update value in all cores More bus traffic.	Invalidate first time, remaining writes do not send invalidation on the bus
Write to different words in the same block	Updates each word	First write invalidates all words in that block
C1 writes--> followed by C2 reads	Updates the value --> This works better	Invalidate causes a miss in C2 everytime. Not a good protocol here

Cache coherence Definition



- Core C1: Write to a location X, followed by a read from location X, returns the most recently written value (just like a uni-processor)
  - If there is no other write to X by another processor occurring between the write and the read
- C1 writes to X, C2 reads from X after “sufficient” time, C2 reads the most recent value from X: Coherent view of memory



- Writes to the same location are serialized and seen in the same order
  - C1 writes to Y and then C2 writes to Y, all cores must see the writes in the same order

Cache coherence protocols

Key idea: Track the state of any shared data block

- Directory based protocol
  - Information of shared data is maintained in a directory. Every cache can track the sharing status.
- Snooping based protocol
  - Every cache maintains the status of each data block
  - Data is accessed through a shared bus between processors
  - Cache controllers monitor (snoop) the caches and the bus for the presence of the shared data

Cache coherence protocols
Invalidation based protocol
MSI protocol

Key idea: Track the state of any shared data block

- Acknowledgements:
  - CA, Quantitative approach, Henessey and Patterson
  - Multicore Memory Caching Issues - Cache Coherency: PRACE Summer School 21-23 June 2012
  - Biswa Panda, IITK, Cache Coherence Protocols, CS422-Spring 2018
  - MSI Coherence - Georgia Tech - HPCA: Part 5

MSI protocol

Cache coherence

00 ← M

01 ← S

10 ← I

C1 C2 S

Write

3 states for a cache block:

One cache

- M. **Modified** : The block has been modified in the cache. (Usually for writes). The data in the cache is then inconsistent with memory. Similar to Valid=1, dirty bit=1. Need to writeback before evicting

core.

- S Shared: This block is in one/more cache. Memory-copy is up-to-date (Valid=1, dirty bit=0). The cache can evict the data without writing it to memory
- I Invalid: This block is either not present in cache or has been invalidated by a bus request. Valid=0

MSI protocol

Processor requests

- PrRd: Processor request to read a cache block.
- PrWr: Processor request to write a cache block.

Architectural state of a processor

Instruct

Reg file

P C.

R1

R2

⋮

R32

AC

Flags.

Data / Instr.

Instr memory

Data memory

Pipeline reg

Snapshot

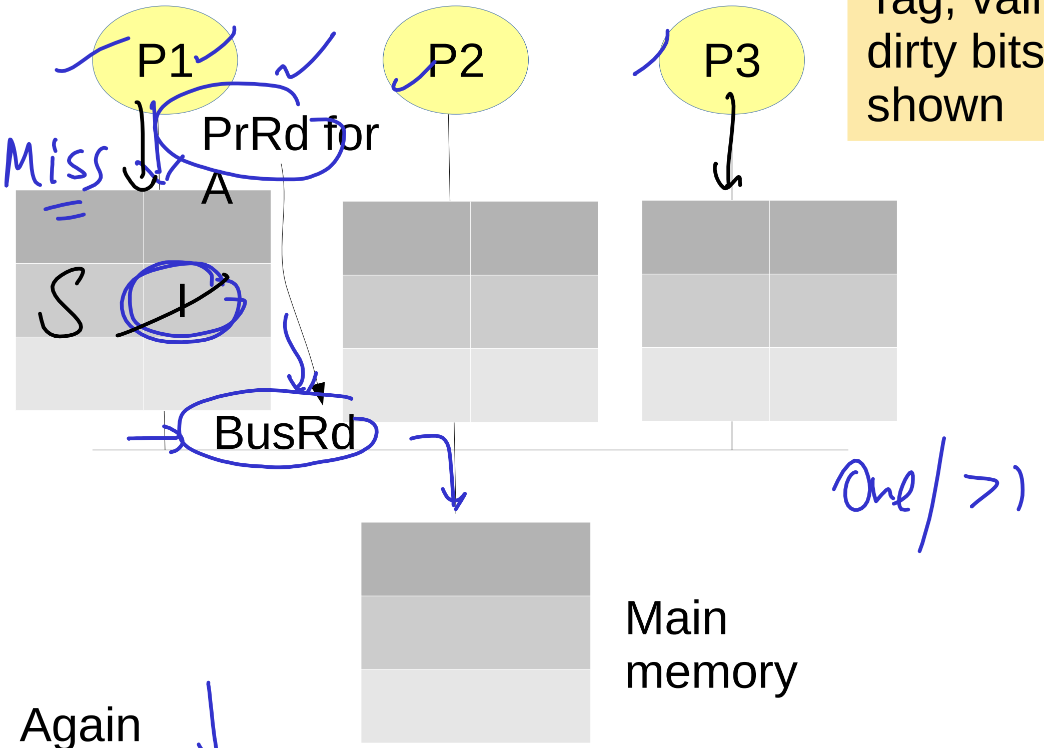
'Checkpointing'

Memory

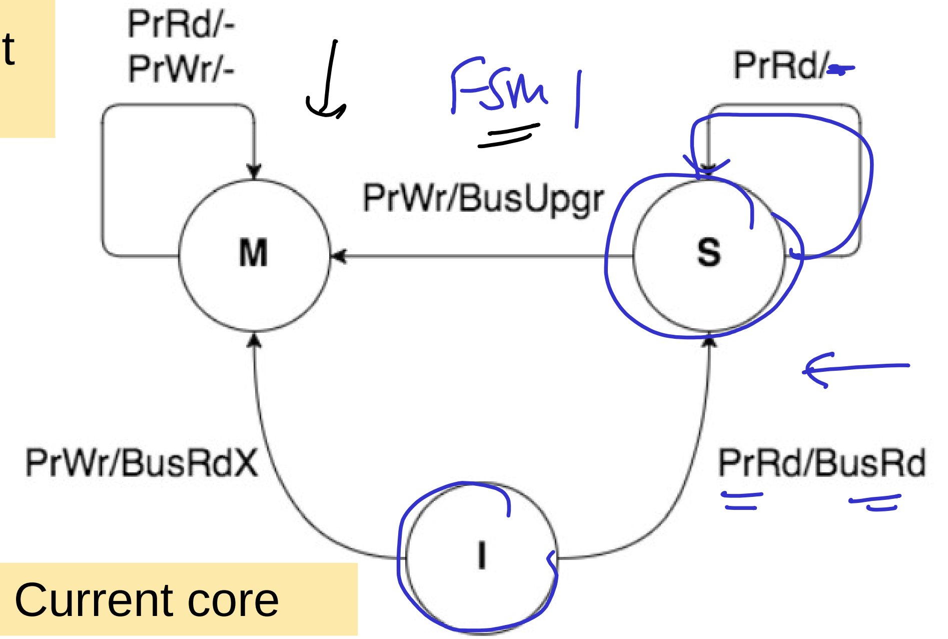
MSI protocol

Cache requests

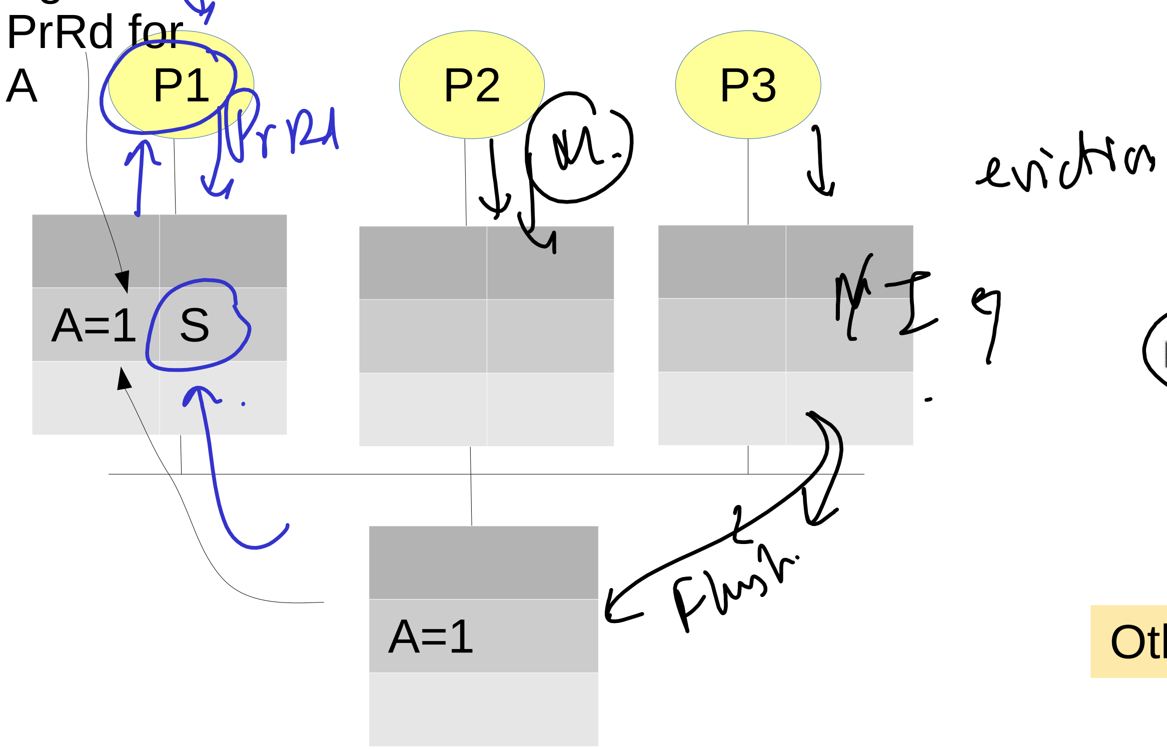
- BusRd: When a read miss occurs in a processor’s cache, it sends a BusRd request on the bus and expects data from memory
- BusRdX: When a write miss occurs in a processor’s cache, it sends a BusRdX request on the bus which returns the cache block and gives exclusive write access to this cache. Needs to invalidate the block in the caches of other processors.
- BusUpgr: When there's a write hit in a processor's cache, it sends a BusUpgr request on the bus to invalidate the block in the caches of other processors. BusUpgr and BusRdX can be combined
- Flush/Writeback: Request indicates that a whole cache block is being written back to the memory

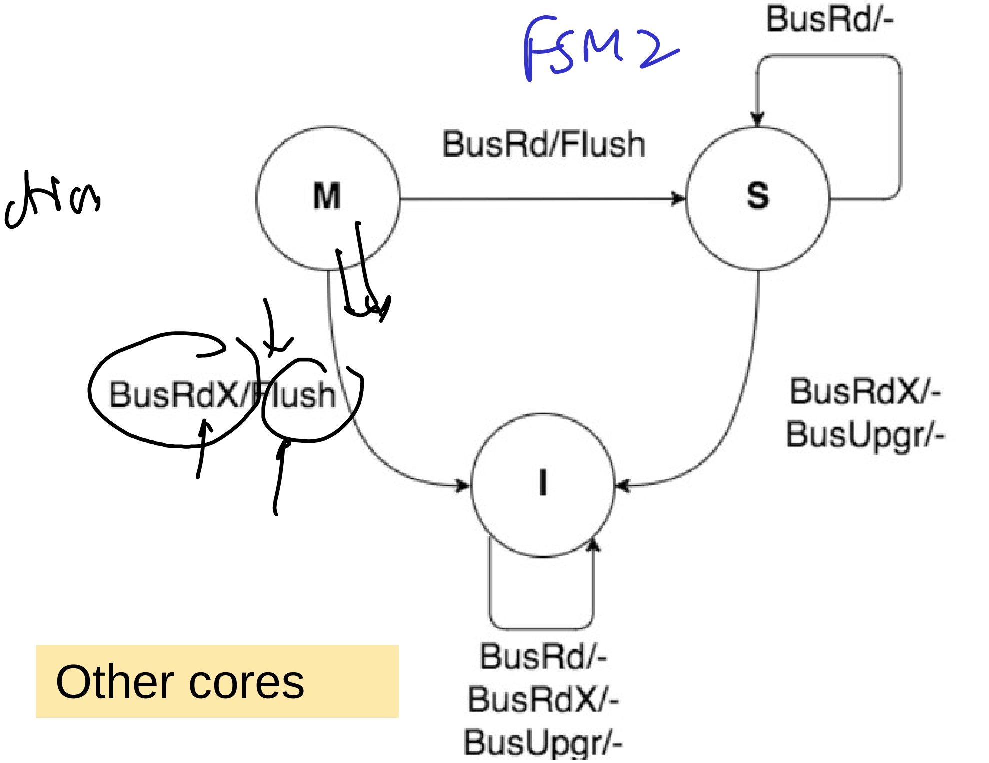


Tag, valid, dirty bits not shown



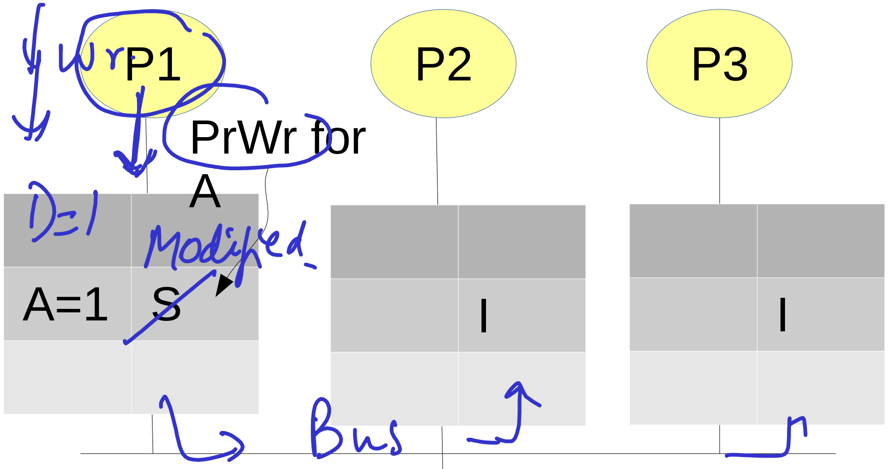
Current core



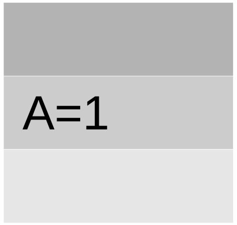


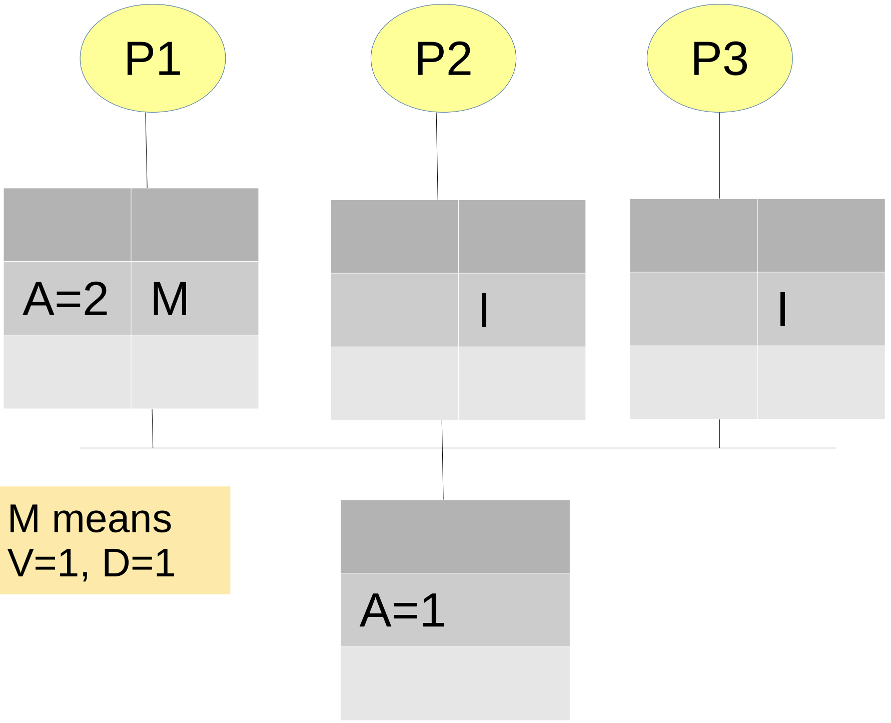
Other cores

Wiki



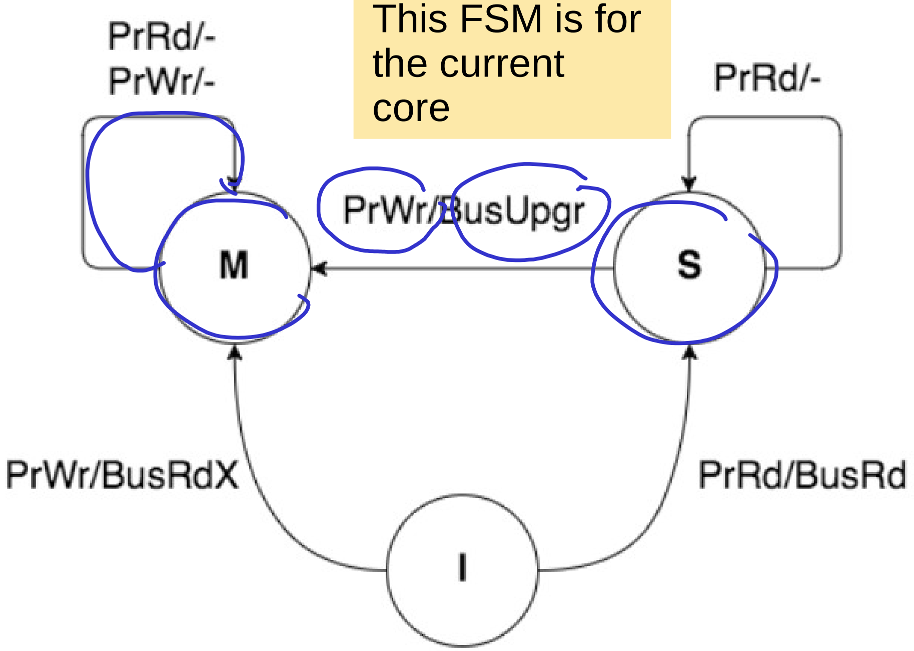
Write back cache assumed Cache hit



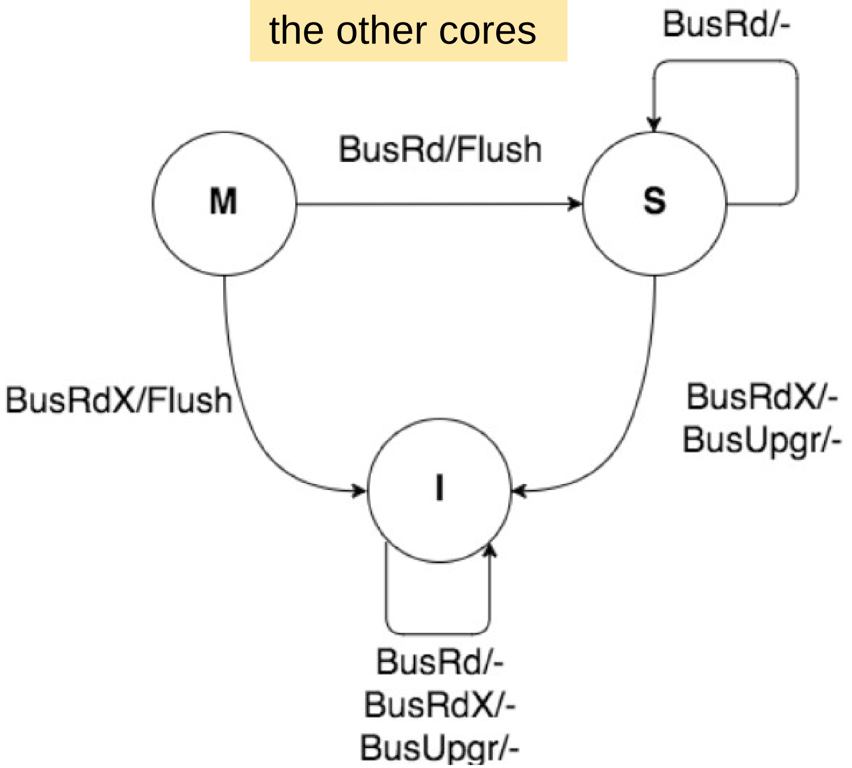


M means V=1, D=1

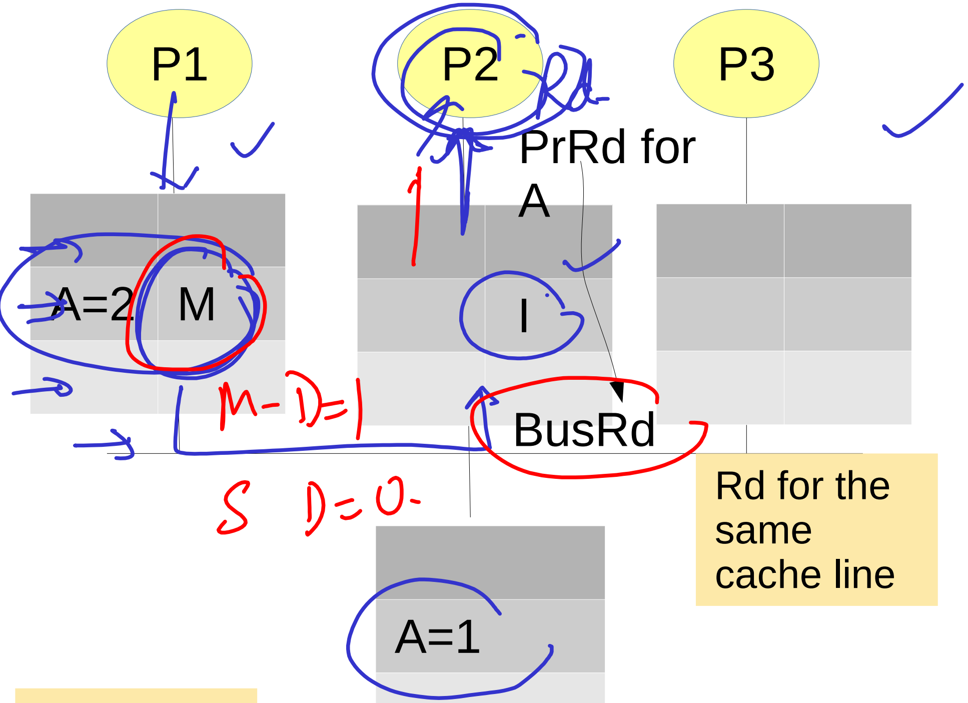
This FSM is for the current core



This FSM is for the other cores

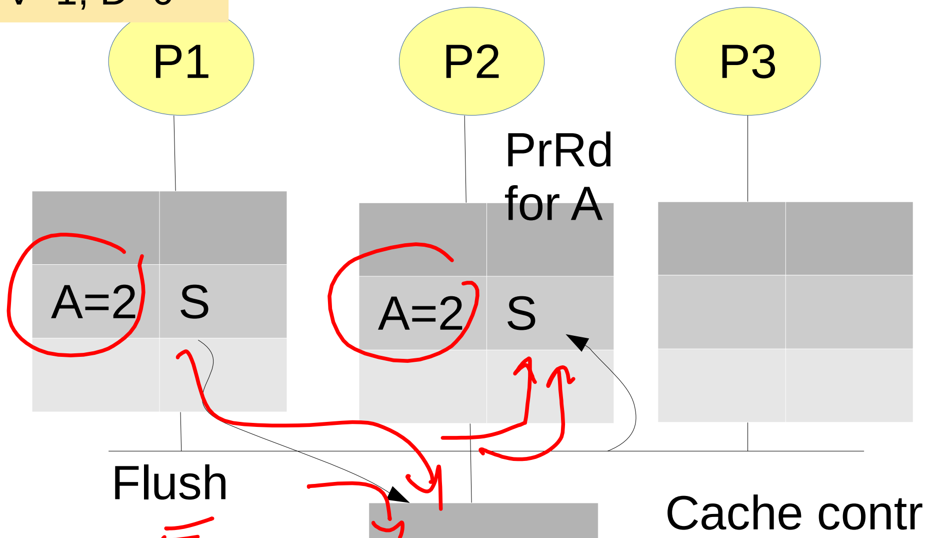


Wiki



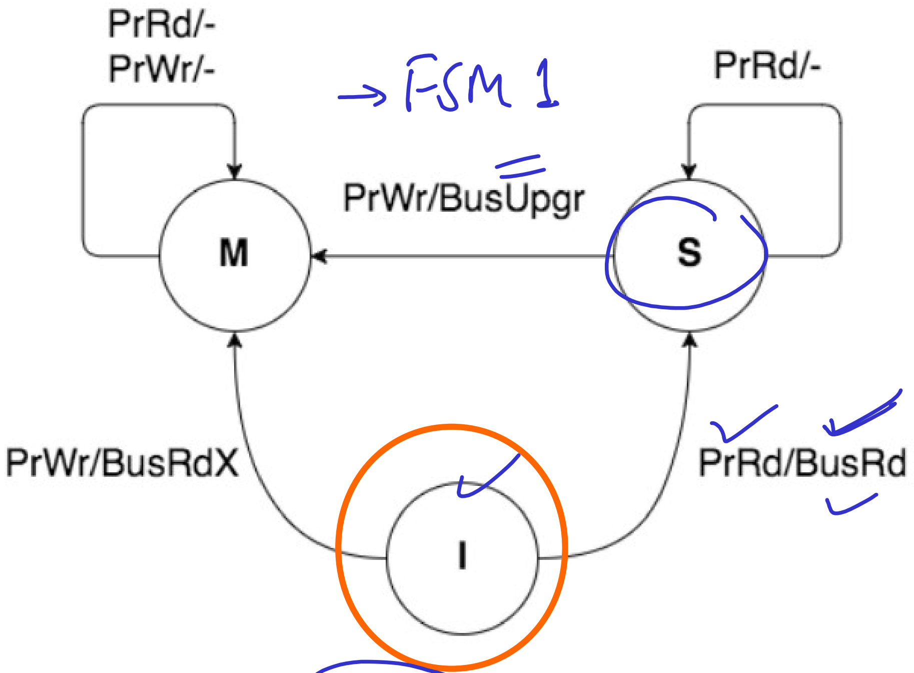
Rd for the same cache line

S means V=1, D=0

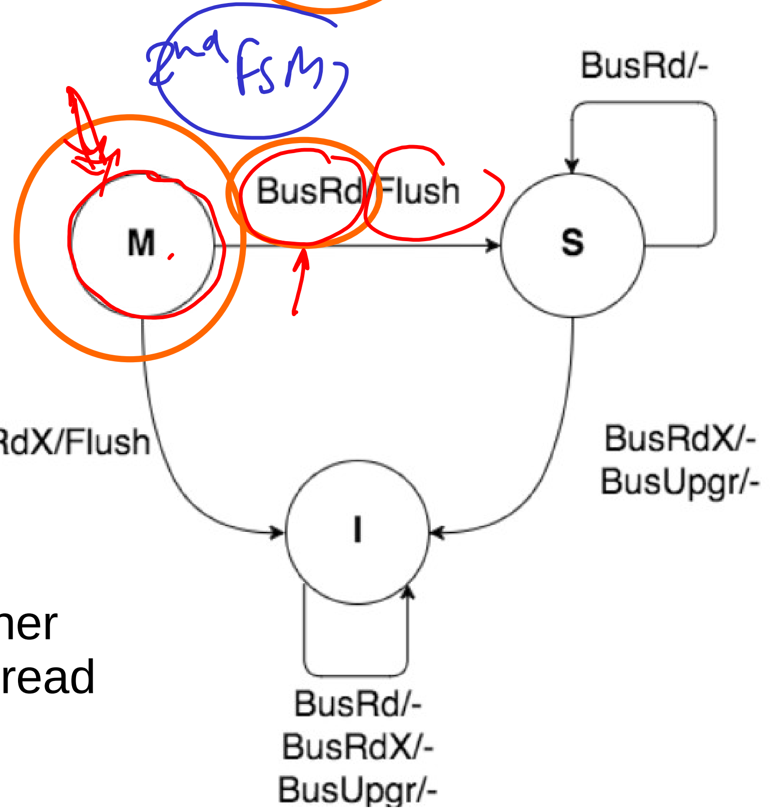


Flush

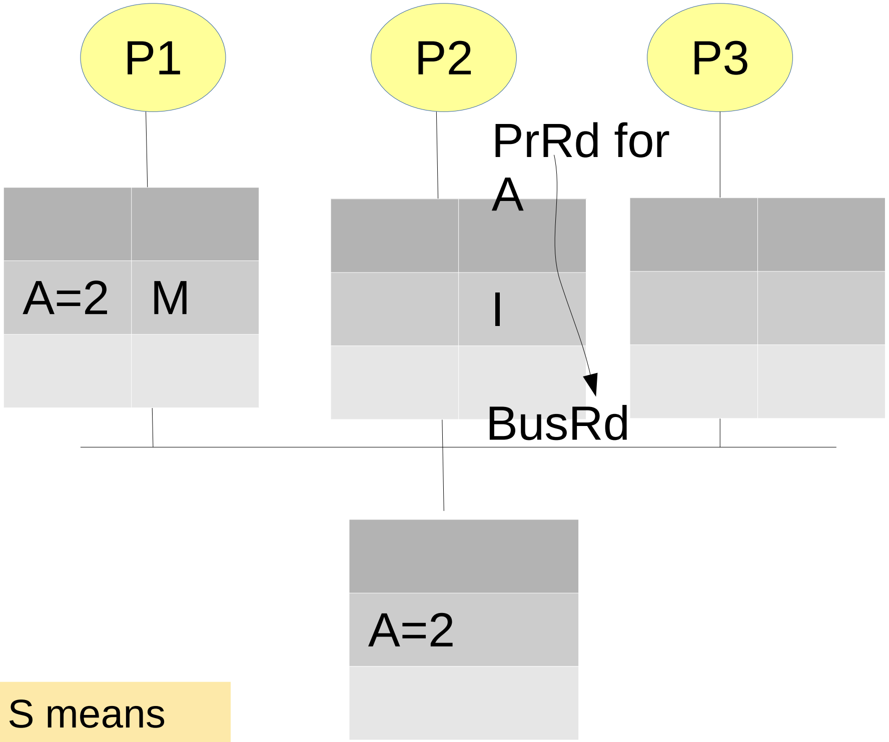
Writeback while changing from M state to S state (sinice S means dirty=0)



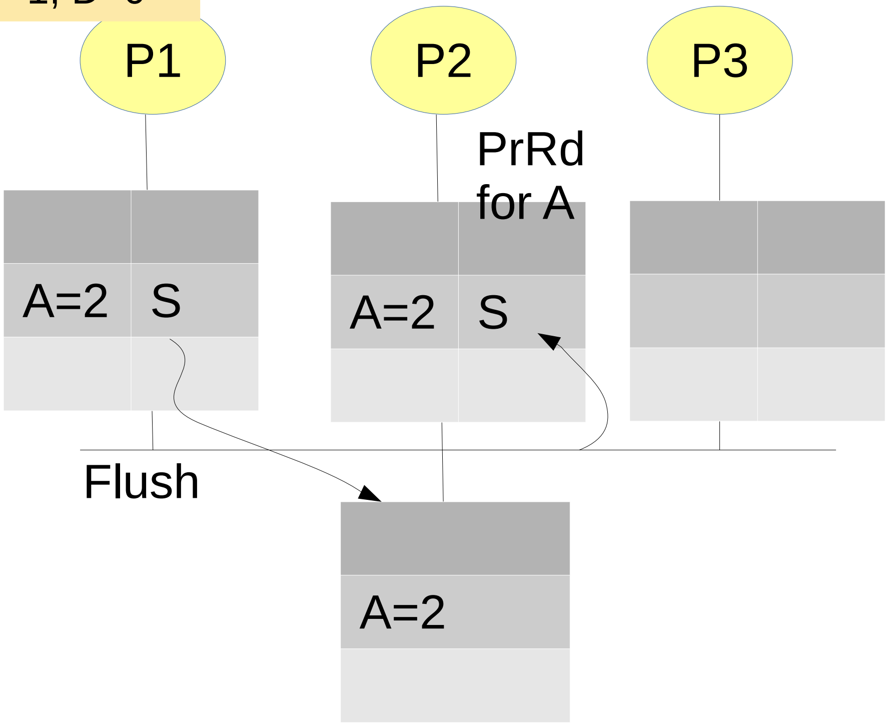
P1 sees BusRd on bus



Cache controller checks other caches: no direct memory read



S means V=1, D=0



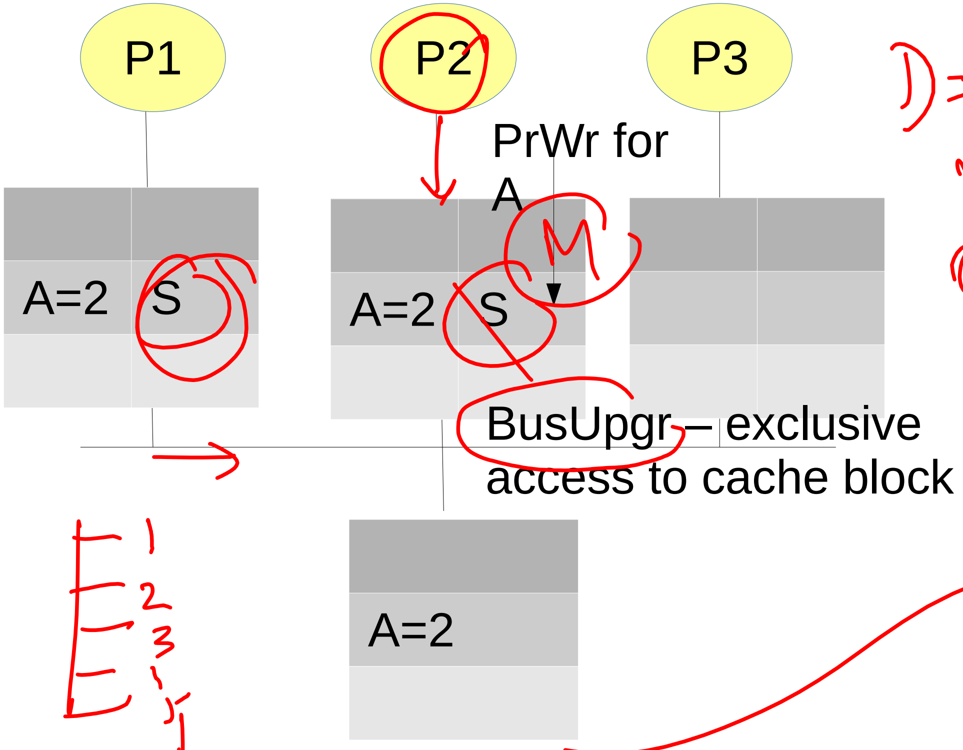
Cache to cache transfer:

Option 1:

- Ask Core 2 to abort/retry.
- Core 1 writes back to memory and then Core 2 reads from memory
- Large memory latency

Option 2: Core 1 intervention

- Core 1 informs memory that Core1 will respond to read request
- Core 1 will send the data to Core 2
- Since both cores are in Shared state (dirty=0), memory needs to pick this data and get updated



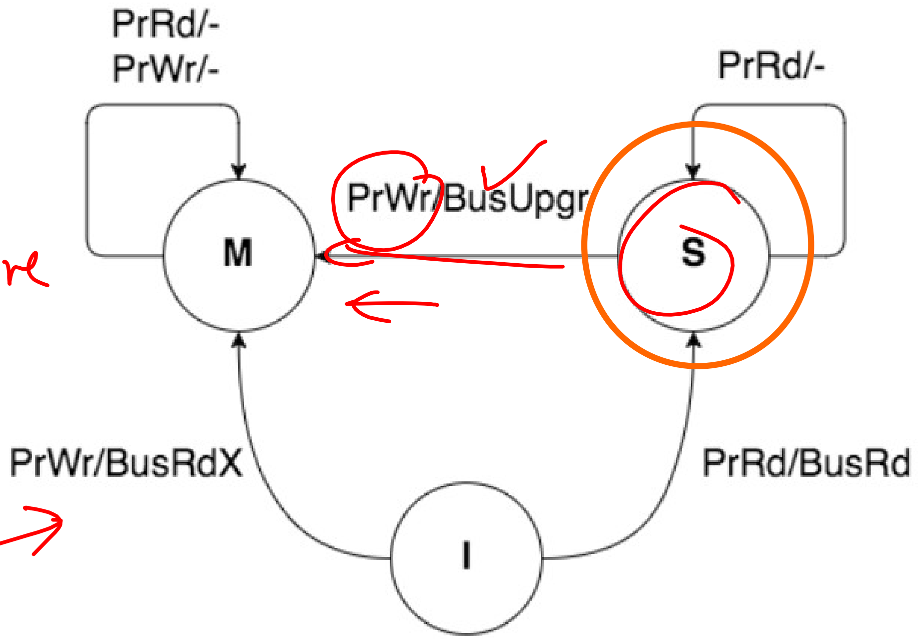
PrWr for A

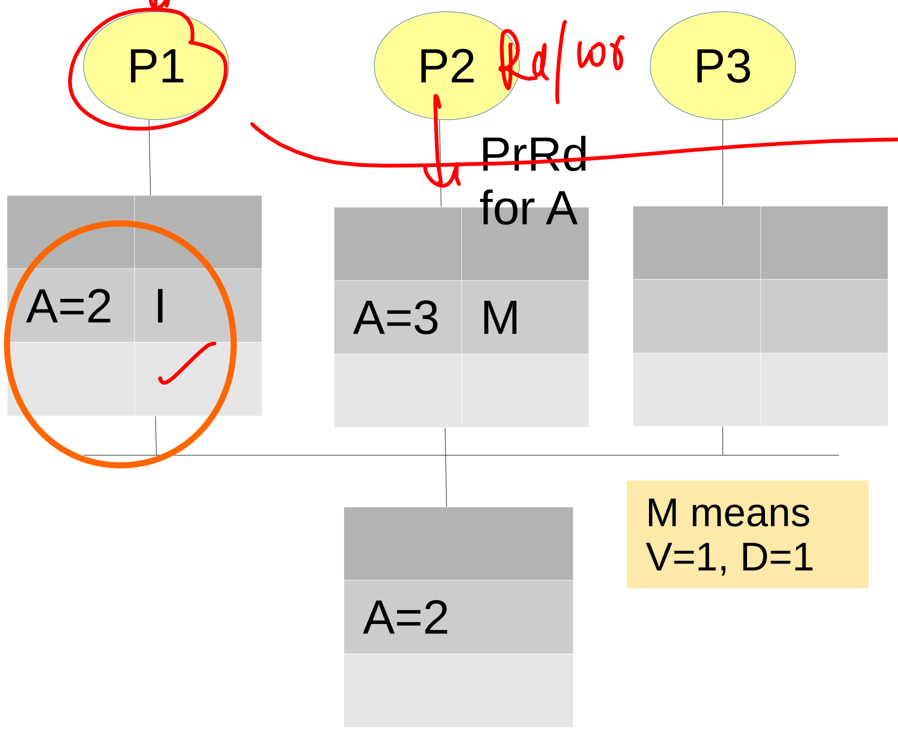
BusUpgr – exclusive access to cache block

D=1

V=1

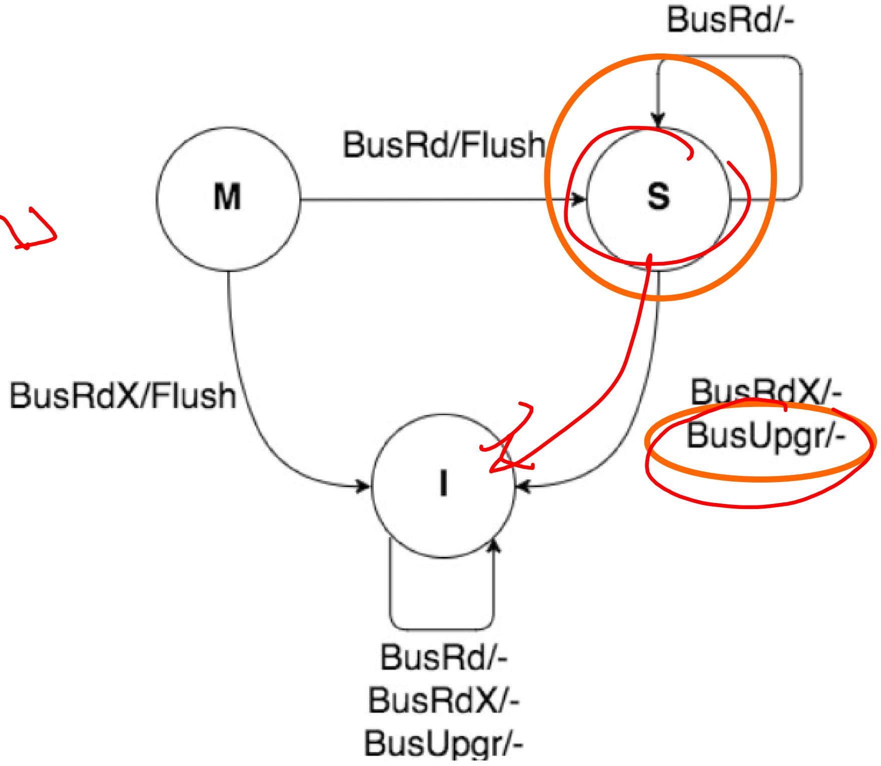
Only core



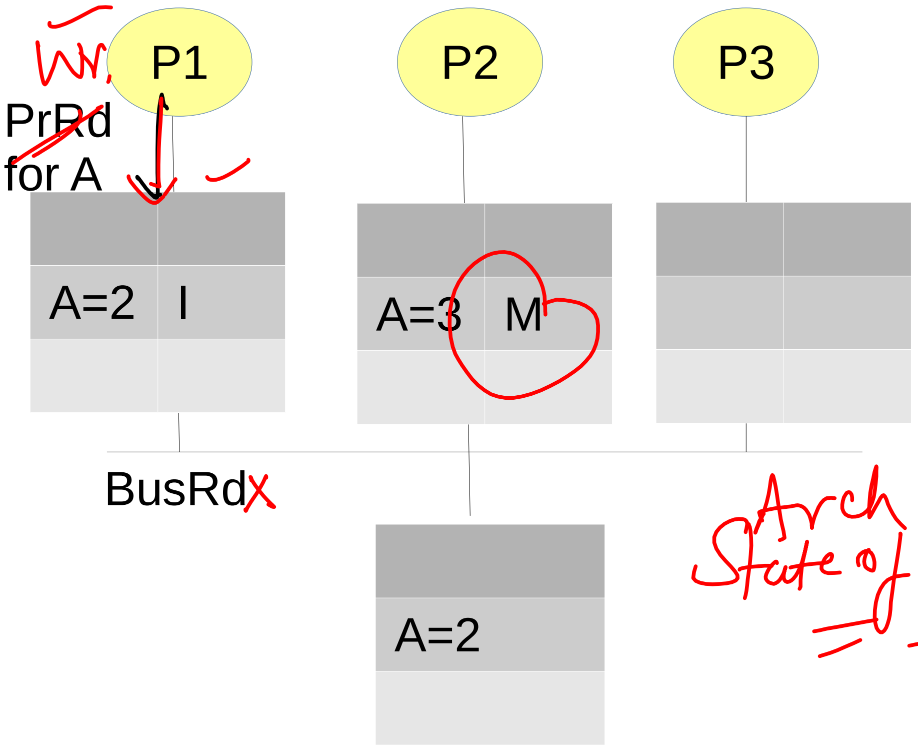


PrRd for A

M means V=1, D=1



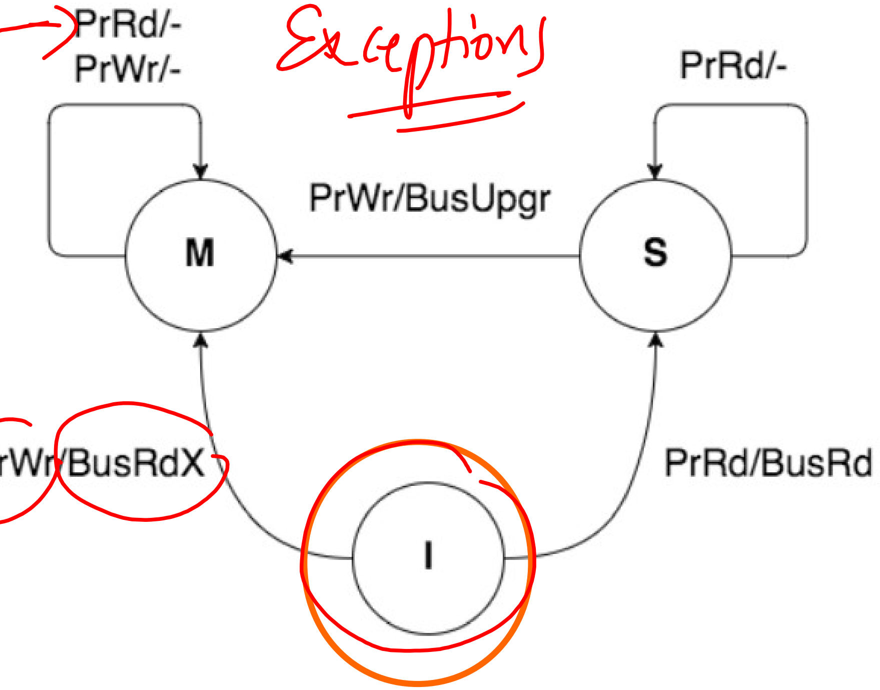
Wiki

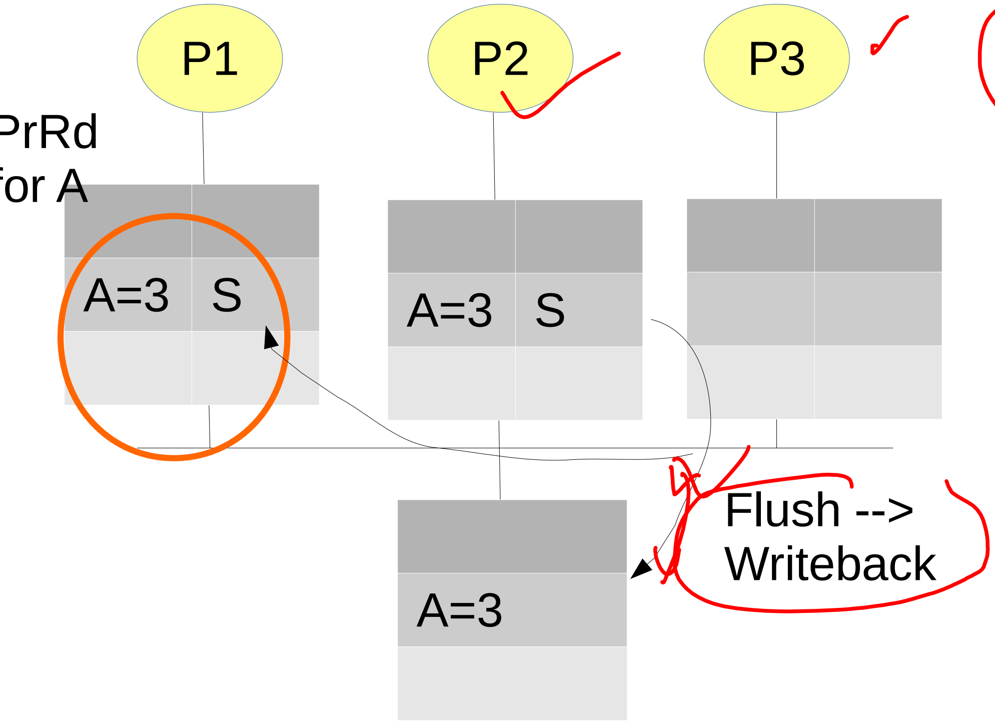


Arch
State of processor

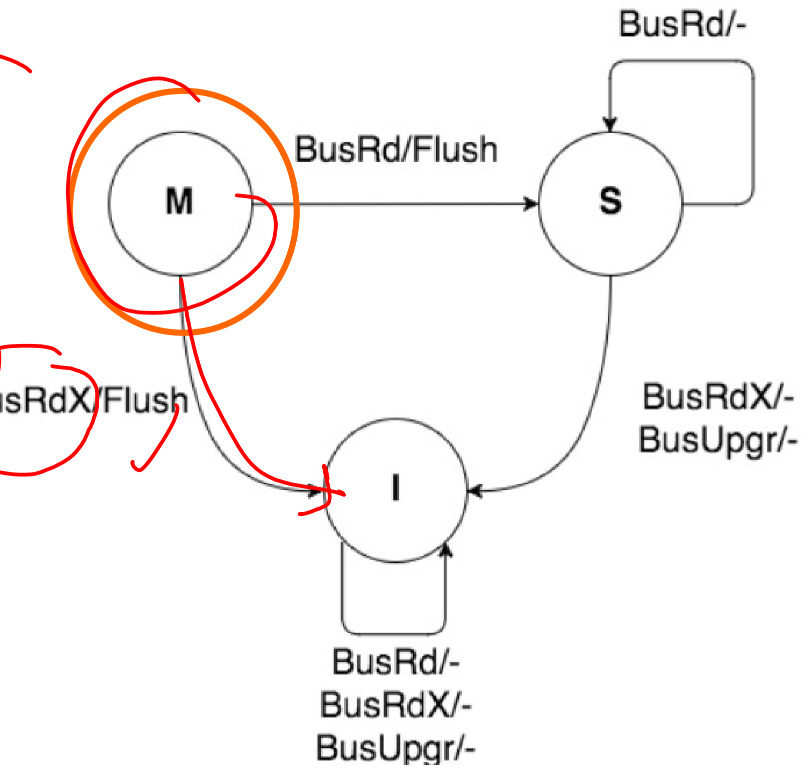
P1

Exceptions

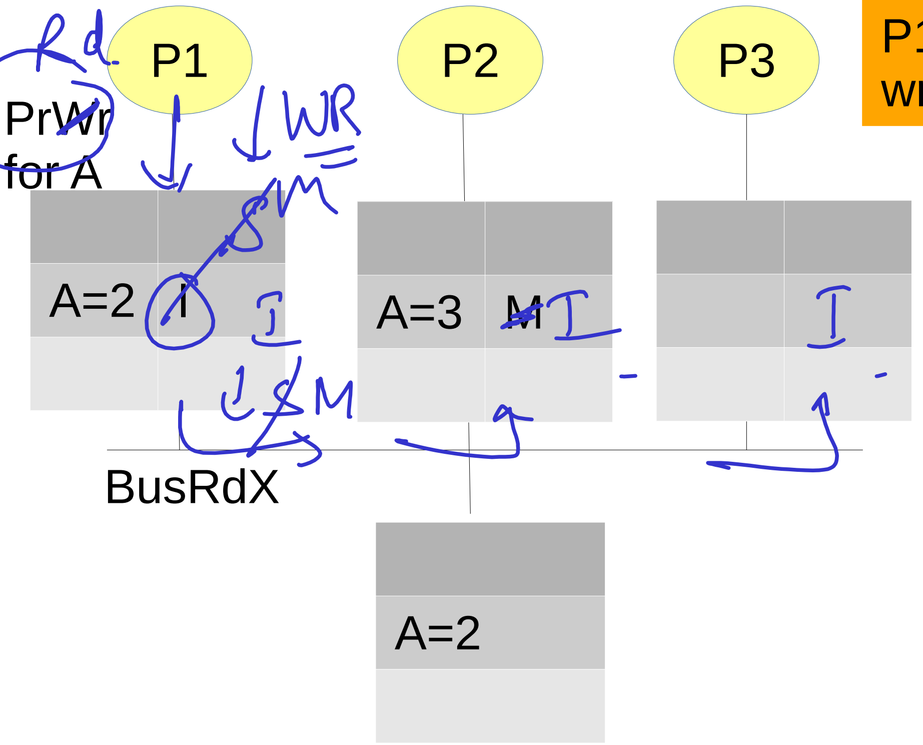




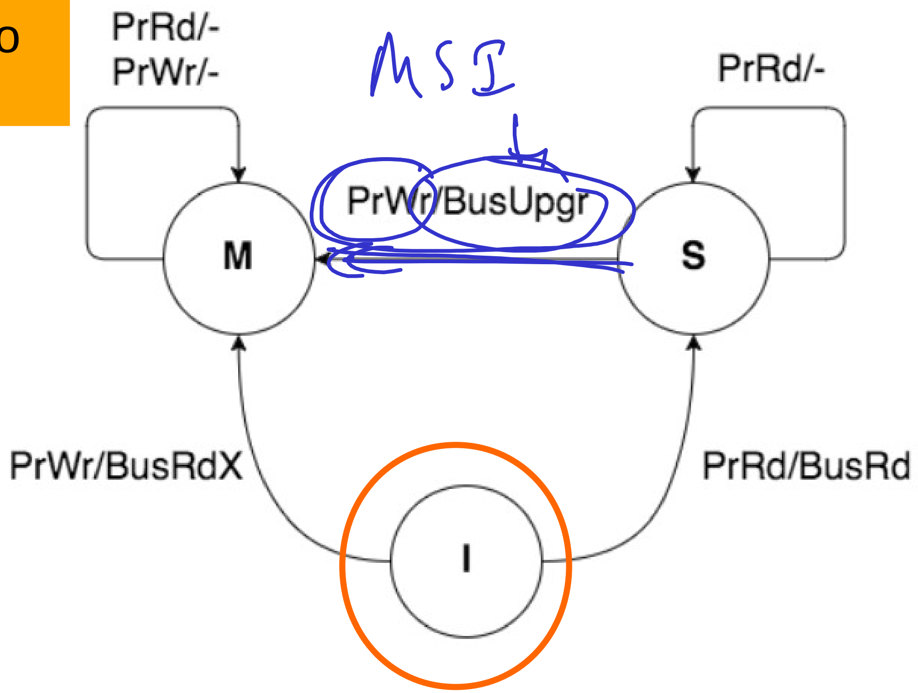
P2 sees BusRd on bus

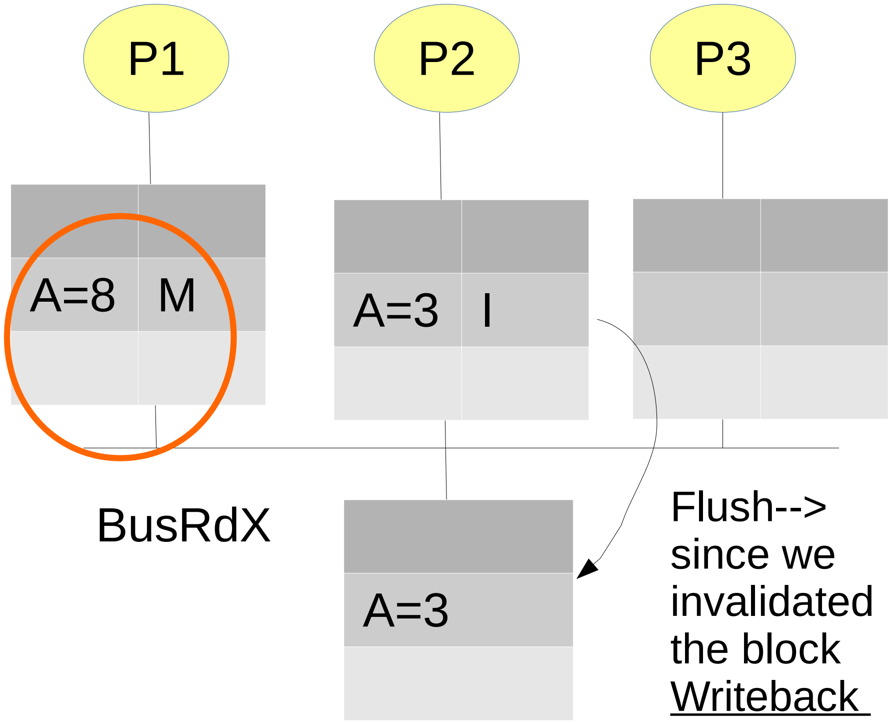


Wiki



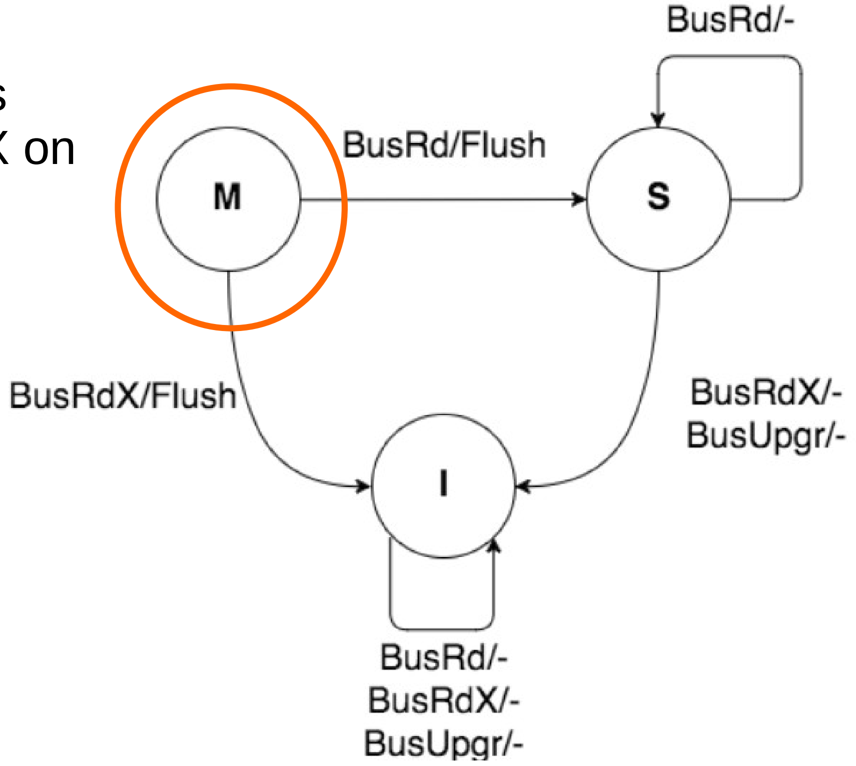
Alternately if P1 wanted to write





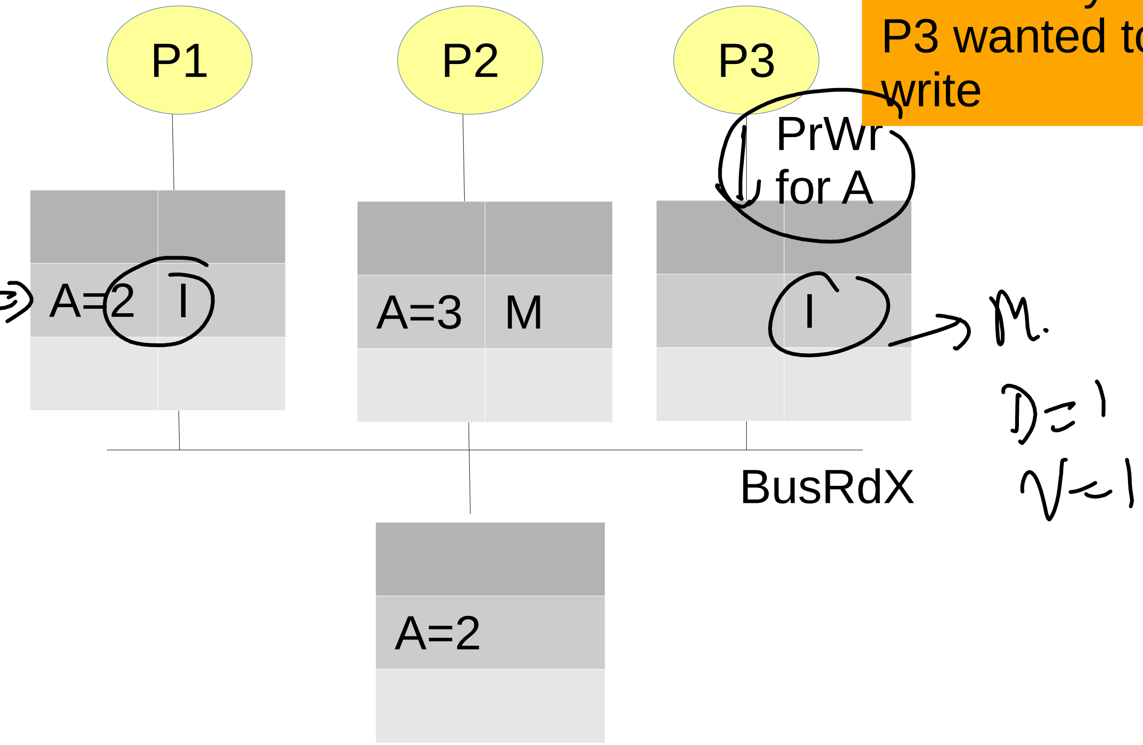
Flush-->
since we
invalidated
the block
Writeback

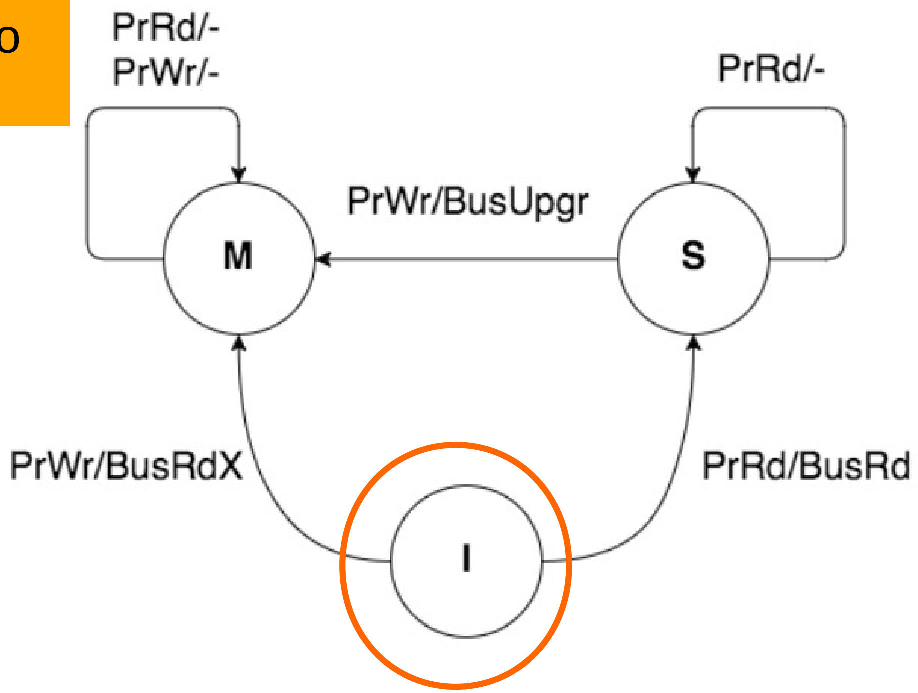
P2 sees BusRdX on bus

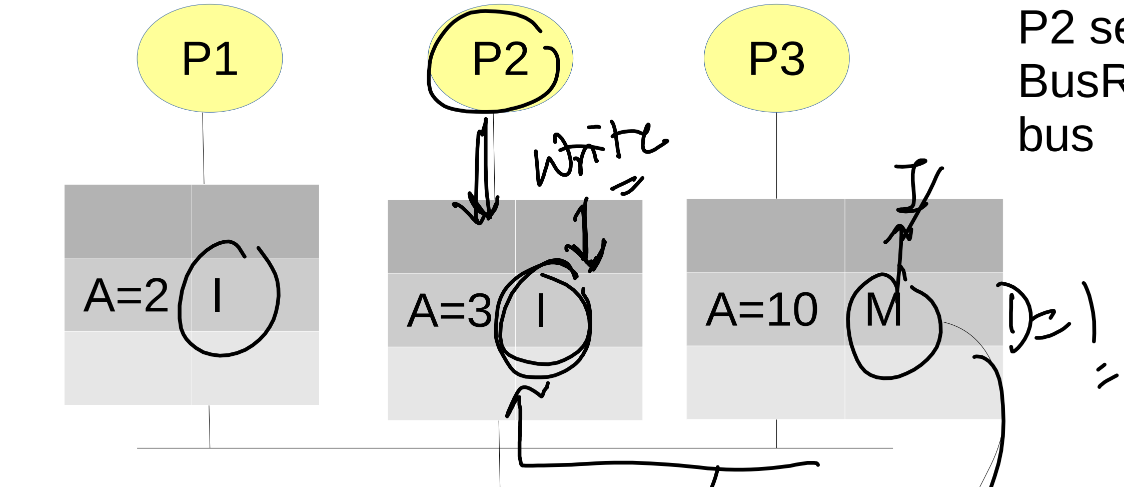


Wiki

Alternately if P3 wanted to write

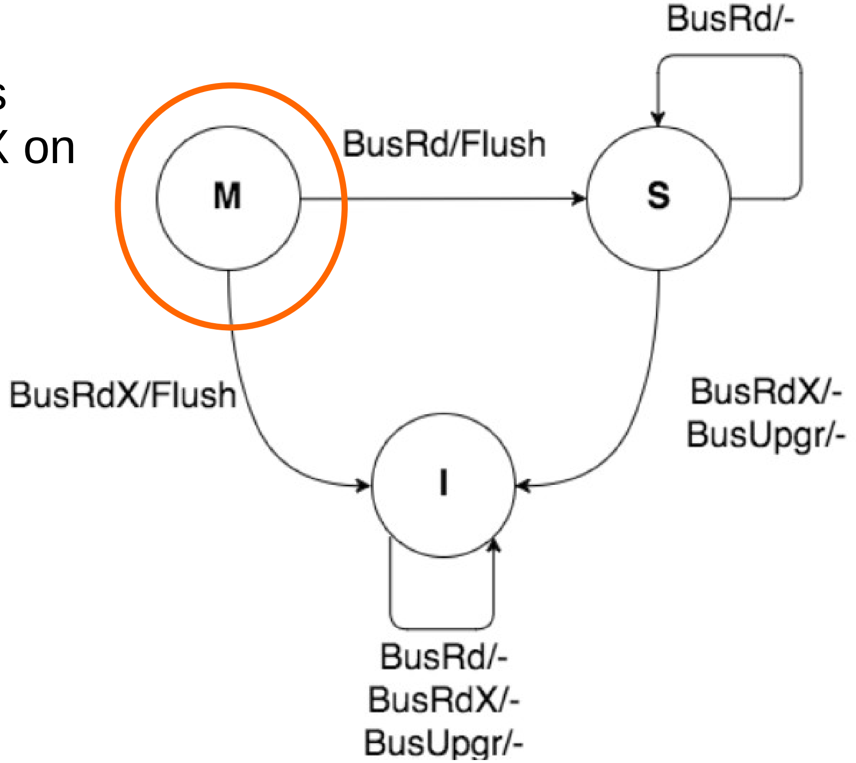






Flush--> since we invalidated the block Writeback

P2 sees BusRdX on bus



Wiki

I M I ⇒ I ~~M~~E I

y=20

I ~~E~~ I

M → I

M → D=1

Eg 2

M→S

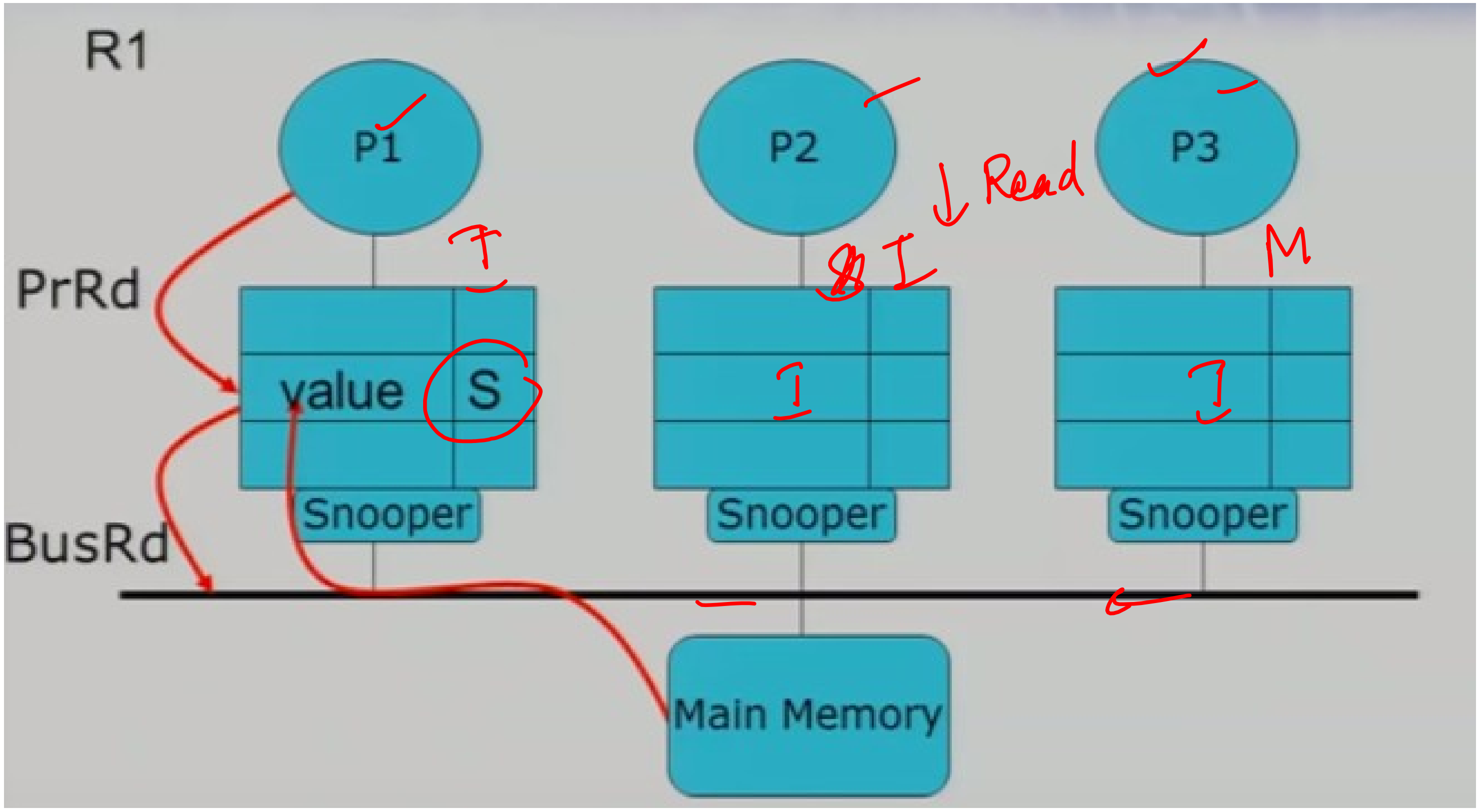
- R1, R2, W3, R2, W1, W2, R3, R2

MSI

I M M
S S S
I I I
M M M
None of

→ I ~~M~~ M
→ S S S
→ I I I
→ ~~M M M~~
None
All.

S I I
I S S
I S M
M S I
M M I



Run through this sequence

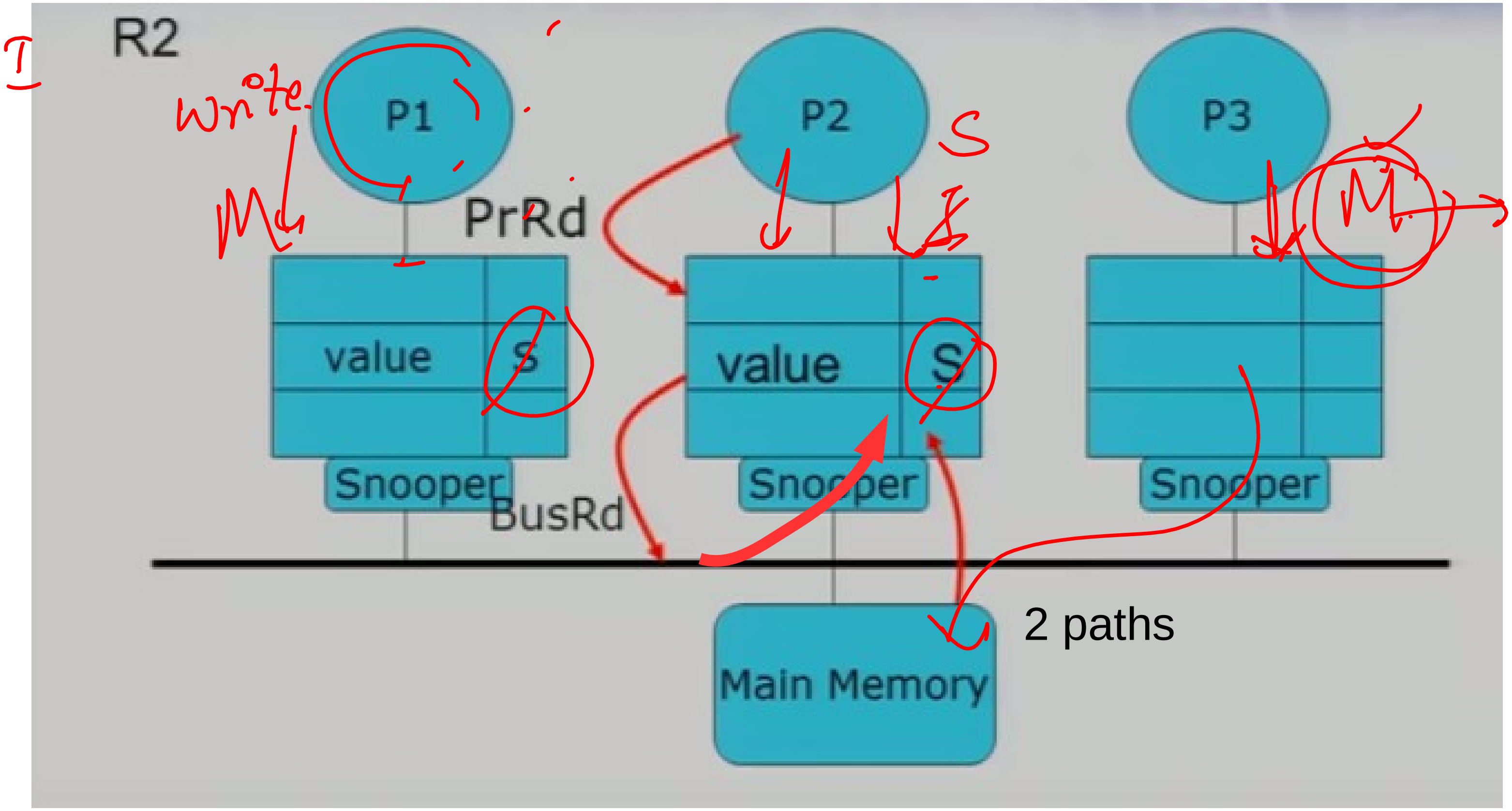
I I I

I I I   S S I   I I M   I S S   M I I   I M I   I S S   I S S

- R1, R2, W3, R2, W1, W2, R3, R2

S I I

S S   I I M   D=0   20



2 paths

00 M
01 E
10 S
11 I

VM - ~~4~~ 3.

Ex - 1 -
uhro

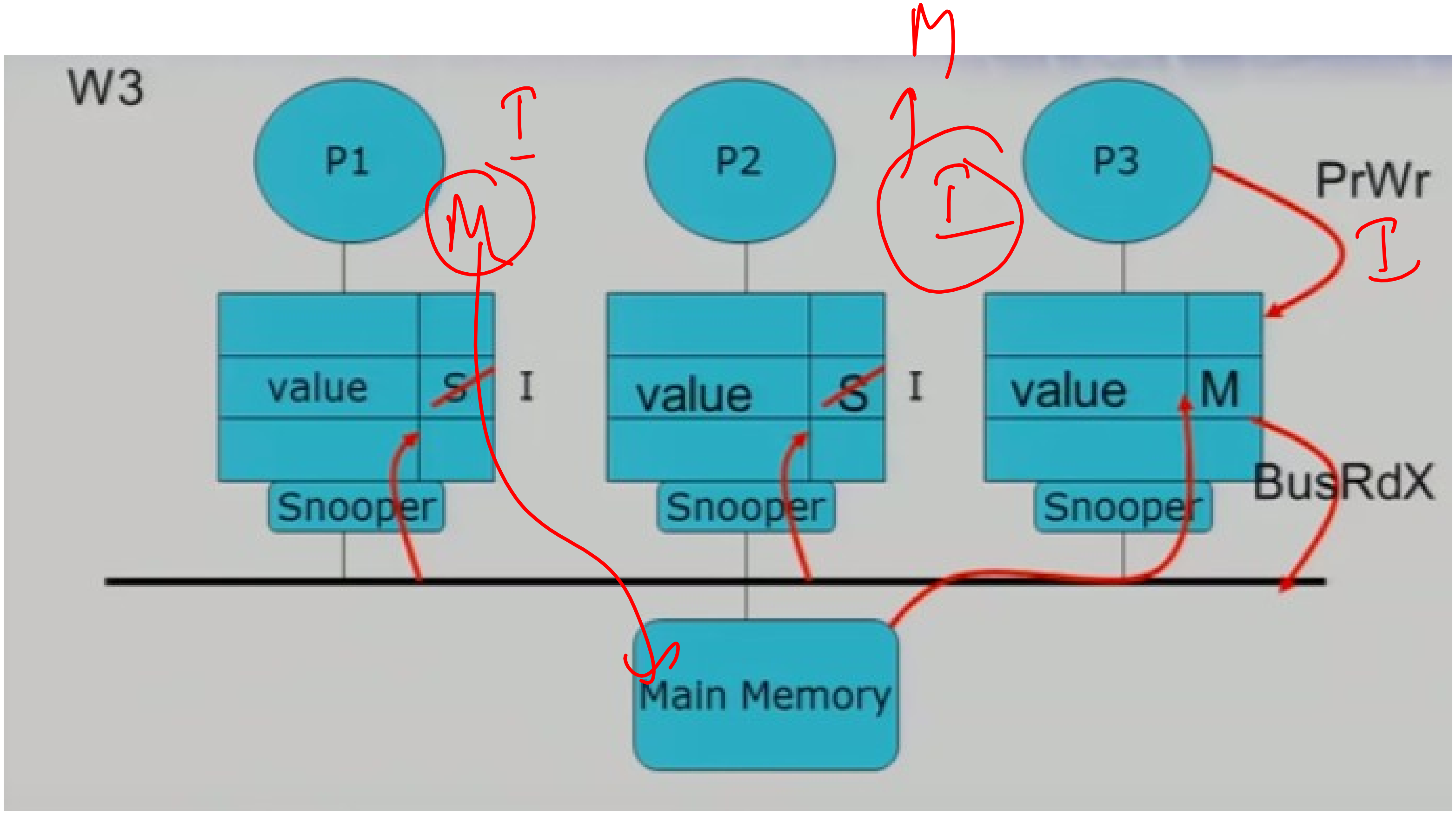
BP - 9/2
6 classes.

Run through this sequence

~~existing~~

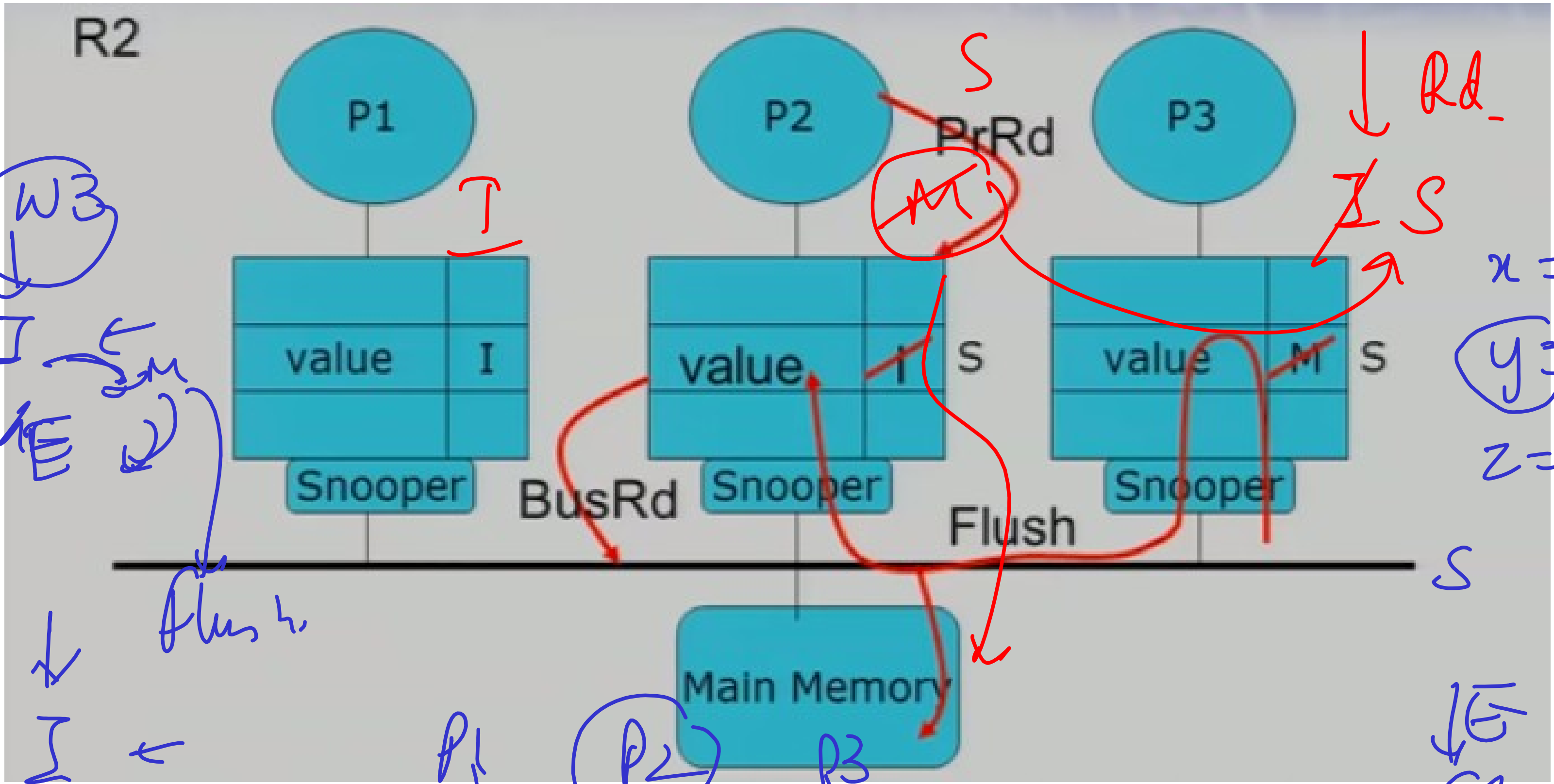
- R1, R2, W3, R2, W1, W2, R3, R2

M → I
M → S →
data
Memo



Run through this sequence

- R1, R2, W3, R2, W1, W2, R3, R2



W3

S S I ←
I I M E

flush.

y=10 y=10
S S I ←
I I
S S M
y=10 y=10 y=20 →

x = a+b
y = x + c.
z = y + h.

S S I ←

E I
I E I ←

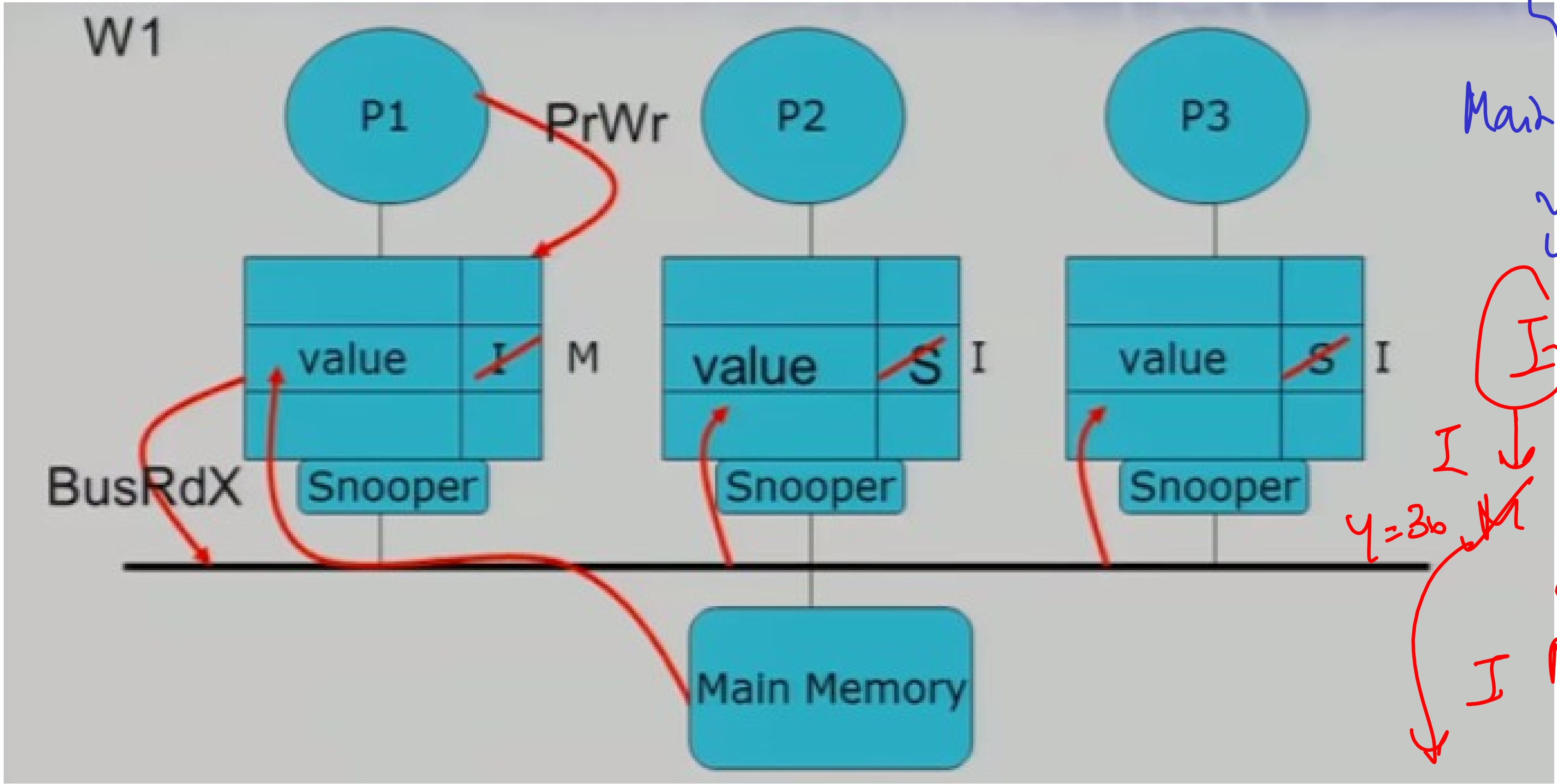
P1 P2 P3
I I I
E → I I

I
S S I

Run through this sequence

P2-Read.

- R1, R2, W3, R2, W1, W2, R3, R2



S S I

I I M

y=W y=20

D=1

I S S

Main mem

y= ~~20~~

I S S

I M

y=30

I I

I M I

WB

y=30

Same cache line

Run through this sequence

Address 20

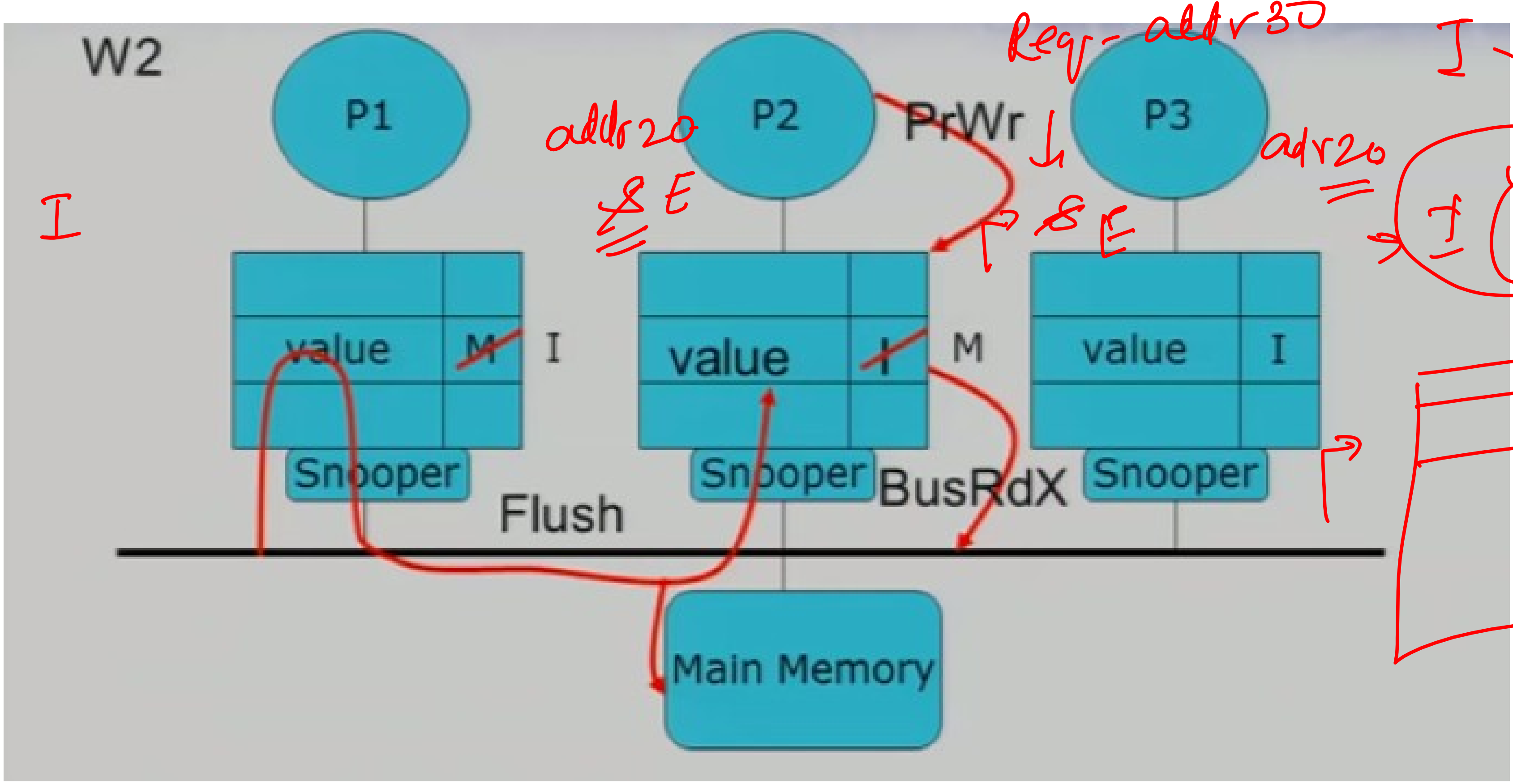
I M I

- R1, R2, W3, R2, W1, W2, R3, R2

Rd

Req - addr30

I S S



addr20 I

addr20 ~~S~~ E

I ~~S~~ E

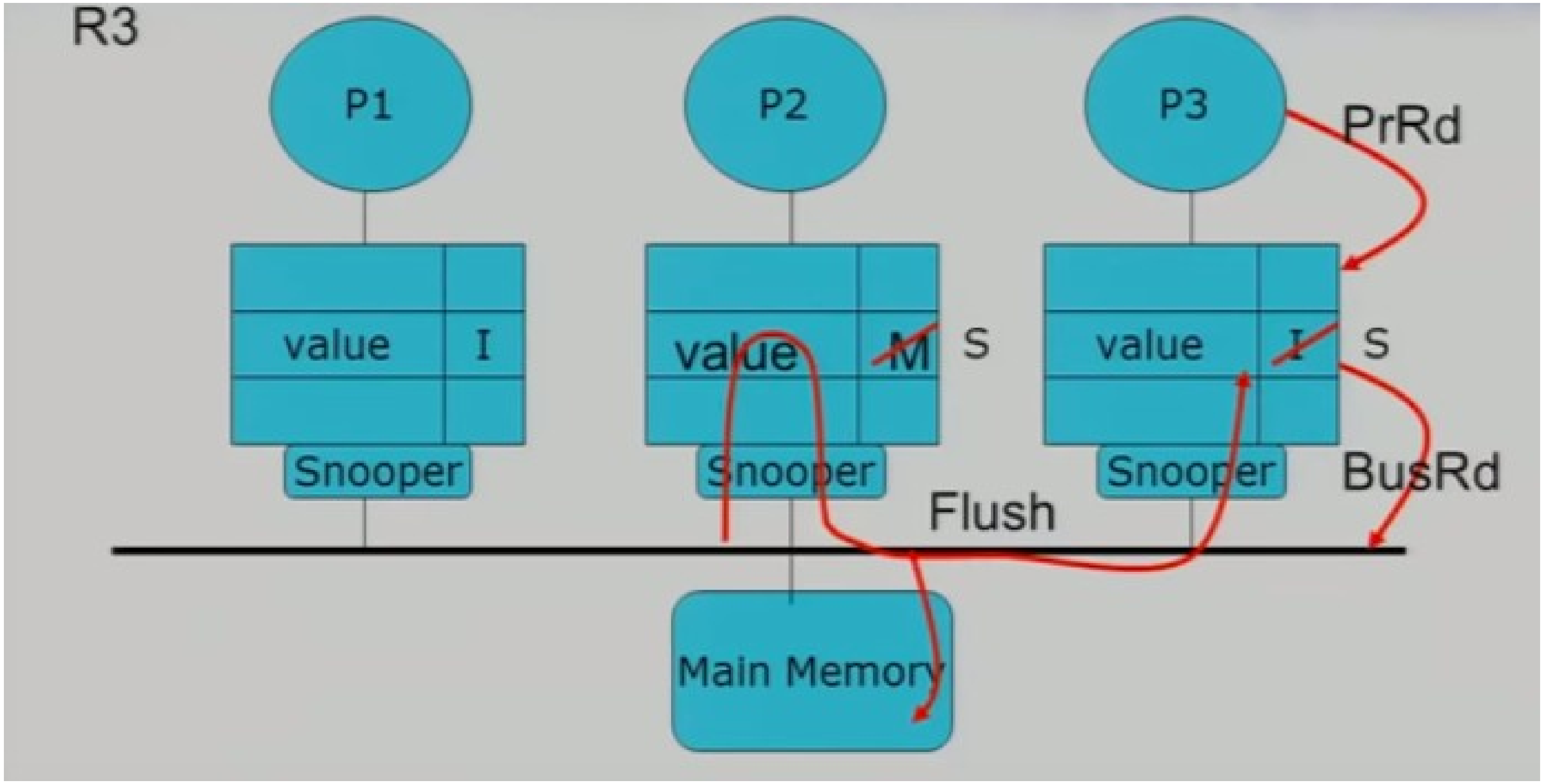
adr20

Read

I S ~~S~~

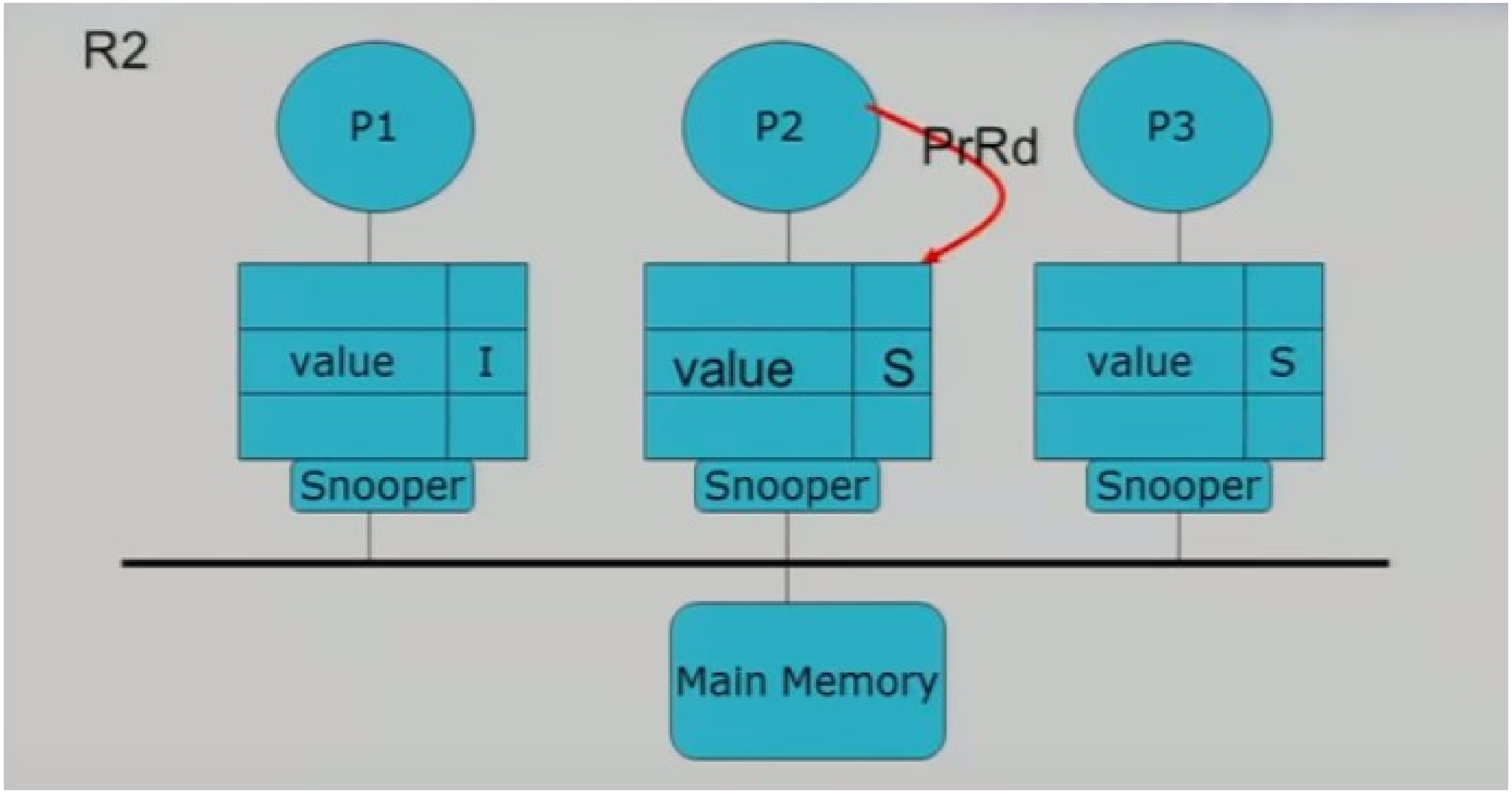
Run through this sequence

- R1, R2, W3, R2, W1, W2, R3, R2



Run through this sequence

- R1, R2, W3, R2, W1, W2, R3, R2



Summary: Invalidation based protocol

- Ensure that a processor has exclusive access to a data item before it writes that item.
- It invalidates other copies on a write.
- Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: All other cached copies of the item are invalidated.
- To perform an invalidate, the processor acquires bus access and broadcasts the address to be invalidated on the bus.
- All processors continuously snoop on the bus, watching the addresses. The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache are invalidated
- Two processors trying to gain bus access: First one gains priority

Challenges and solutions

- Beyond 8 cores snooping bandwidth at the caches is a bottleneck
- Every cache must examine every miss and request placed on the bus
- Directory based protocol is a solution:
  - Intel i7 in Xeon series places a directory with cache status at the last level cache
  - AMD Opteron: Mix of snoop and directory. Broadcasts to find shared copies, uses acknowledgement for ordering operations
- To increase the communication bandwidth between processors and memory, we can use multiple buses and interconnection networks, such as crossbars or small point-to-point networks.

MSI disadvantage

- A processor reads a cache line and then writes to it, when no other core has that cache line
- I--> S --> M, Bus Rd --> BusUpgr
- BusUpgr- to tell all other processors that it is about to write to the block. Not useful if no other processors have that block
- To remove the need to issue this BusUpgr/ bus transactions, MESI adds E.

→ MESI protocol

- Modified: Valid=1,Dirty=1 (Writes). Only copy, only updated copy.

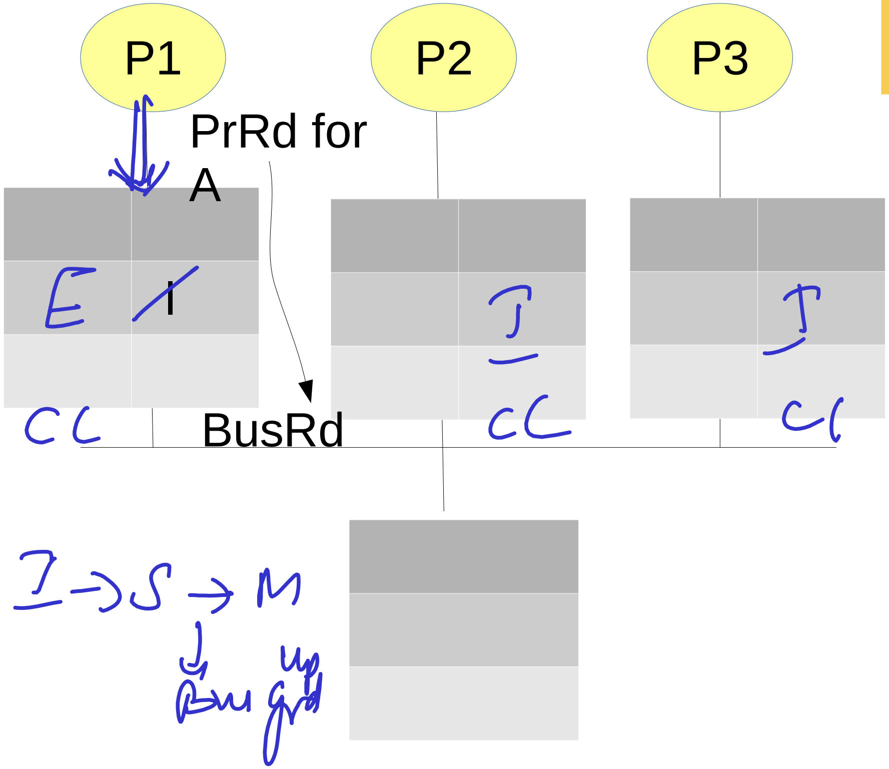
Read ← E

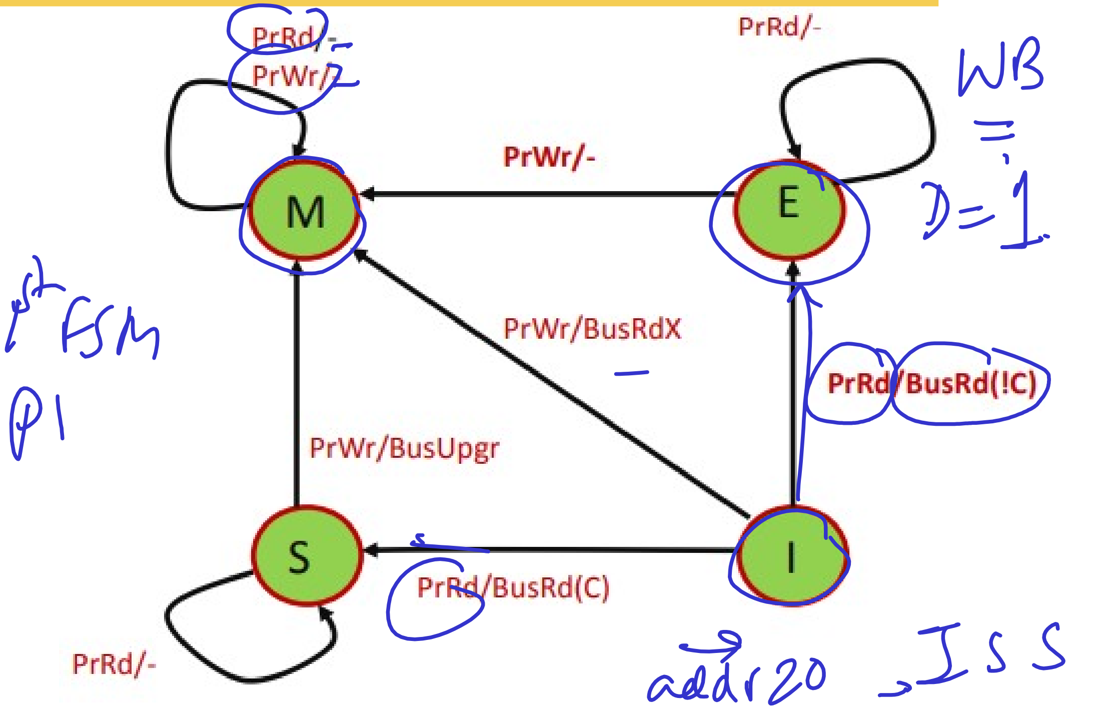
- Exclusive: Valid=1, Dirty=0, Only copy, clean copy, consistent with memory (Reads)

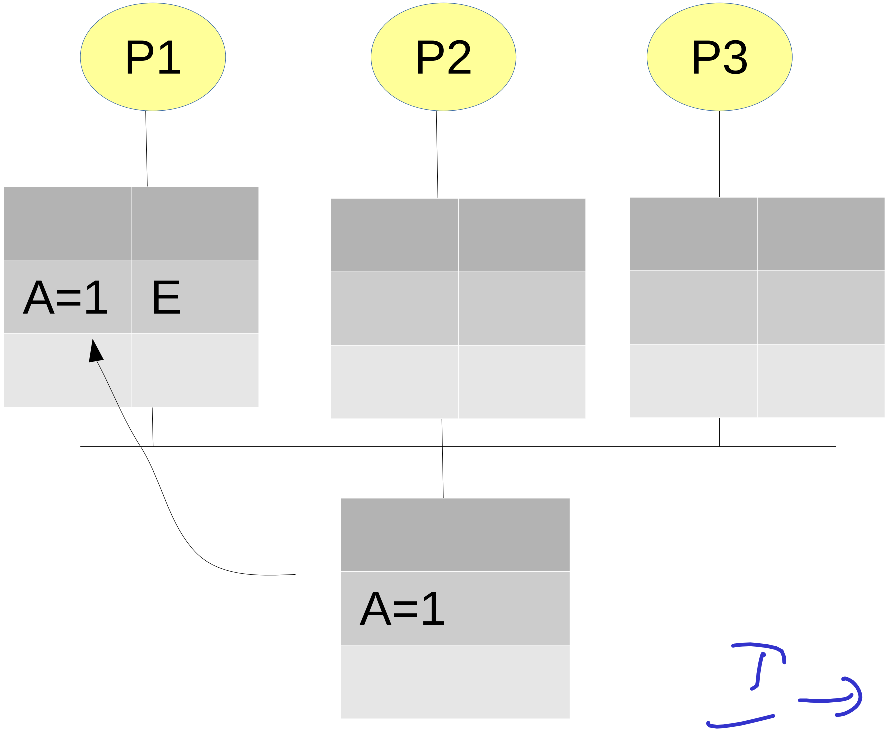
Rd > 1

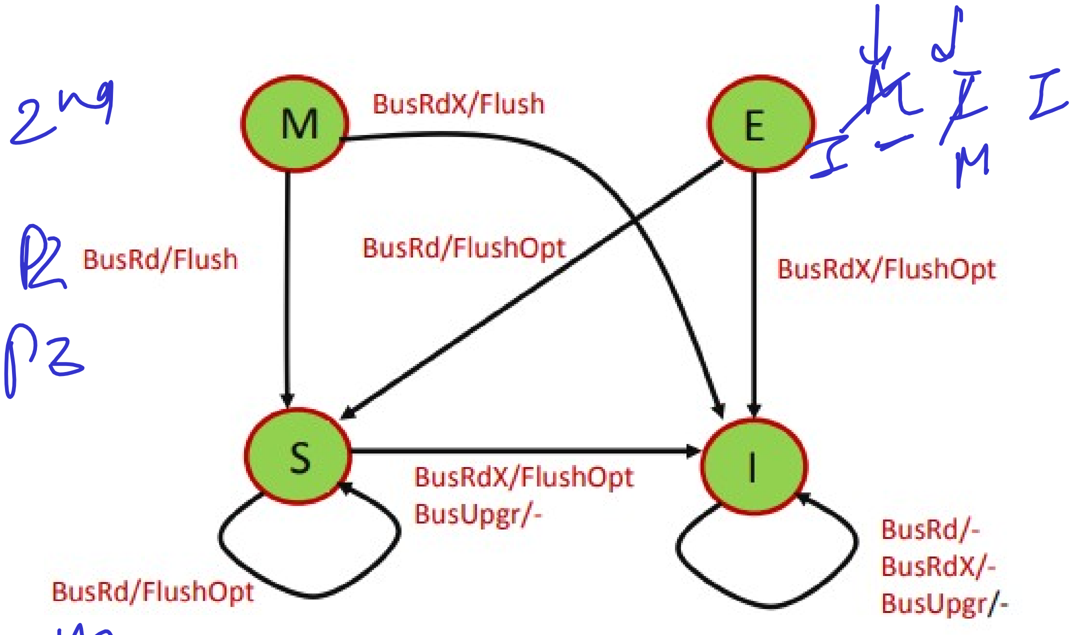
- Shared: Valid=1, Dirty=0, This block is in one more cache. The cache can evict the data without writing it to memory
- Invalid: Valid=0, This block is either not present in cache or has been invalidated by a bus request.

Example:
R1, W1, R2, W2, R1



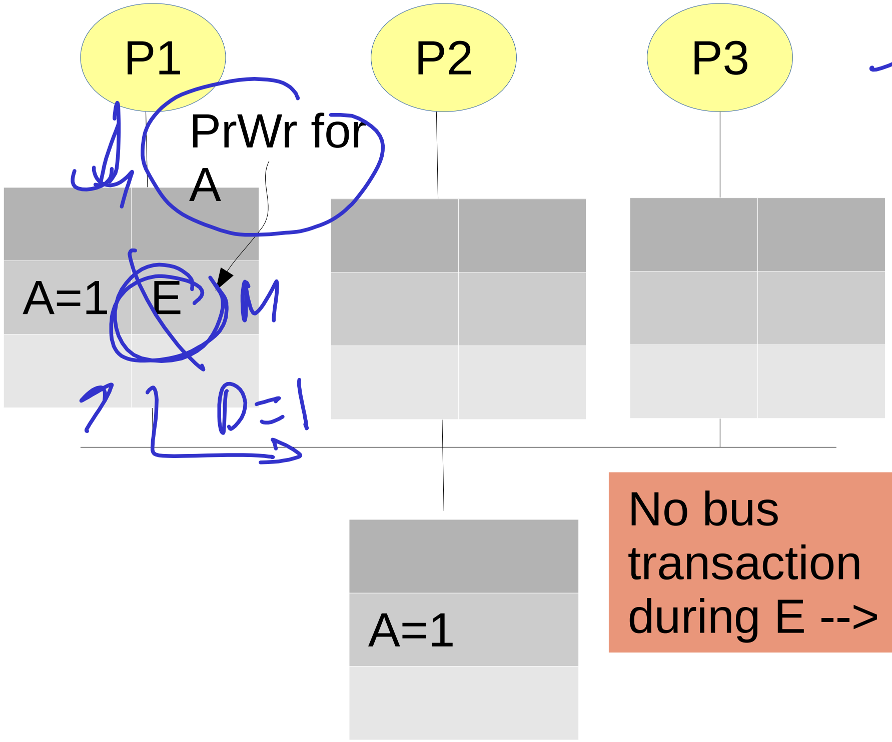






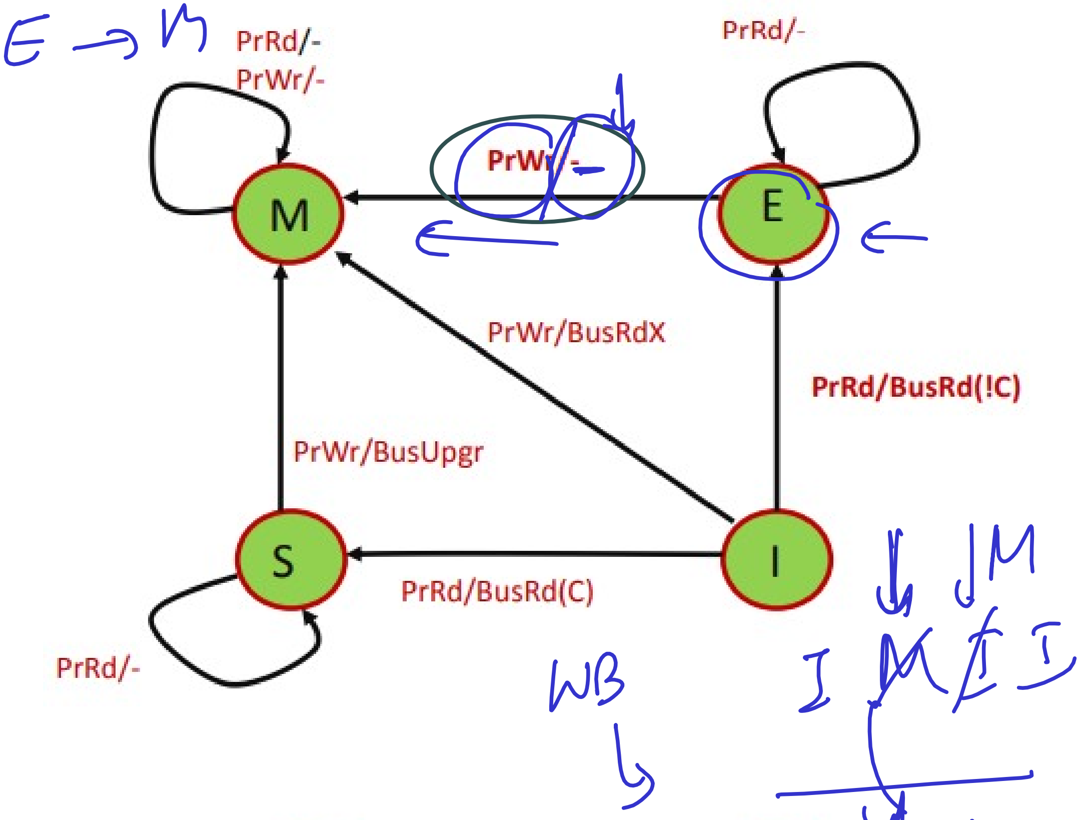
I → S → M

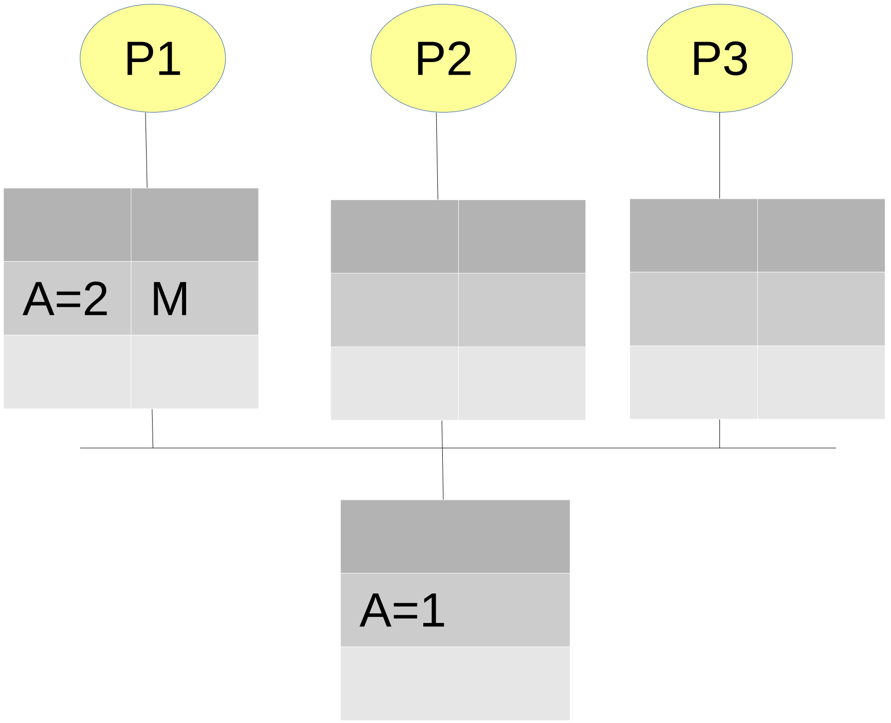
Wiki

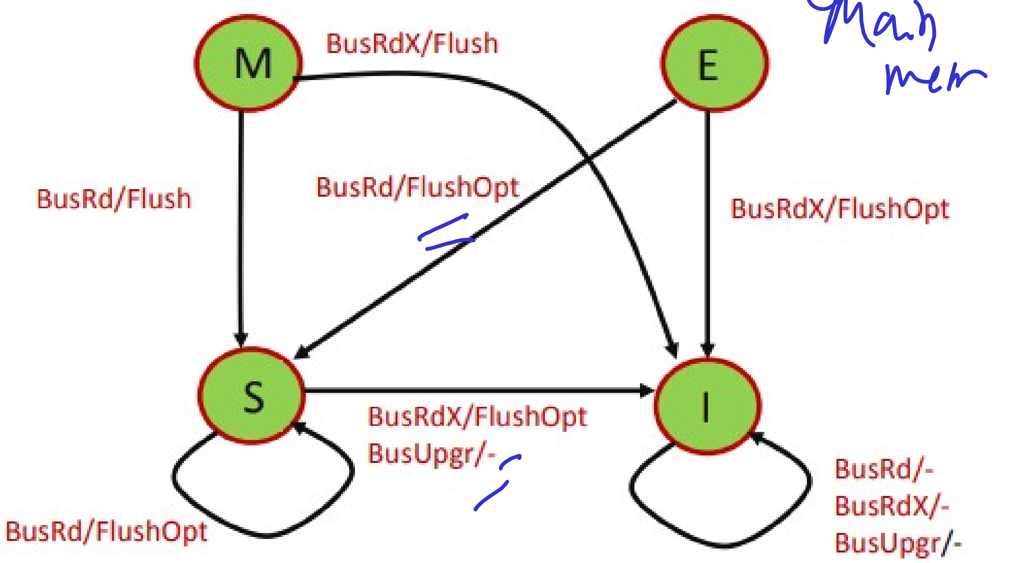


No bus transaction during E --> M

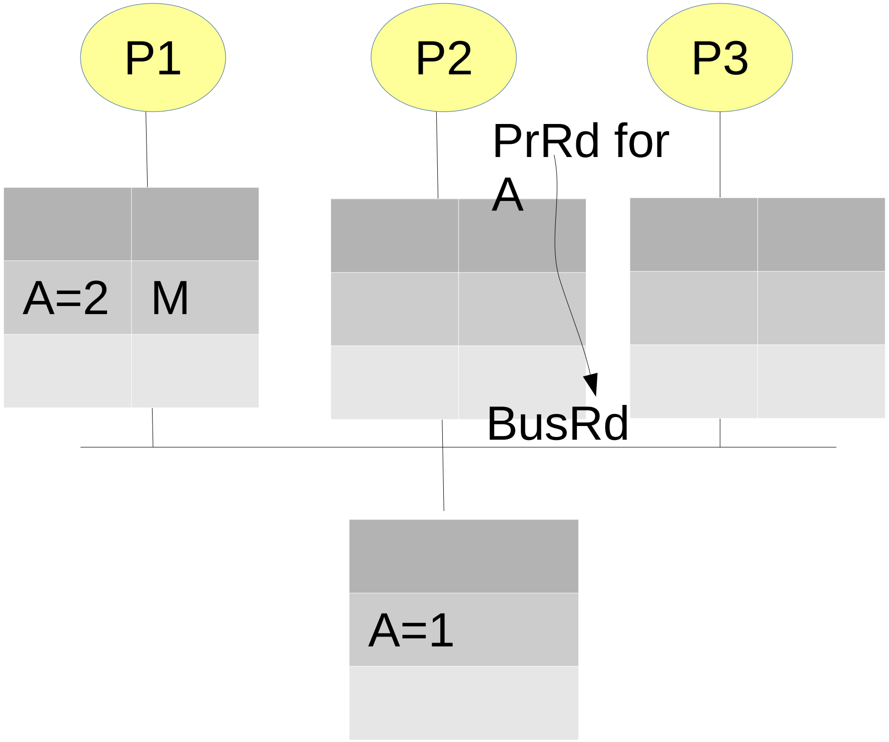
I → E → M





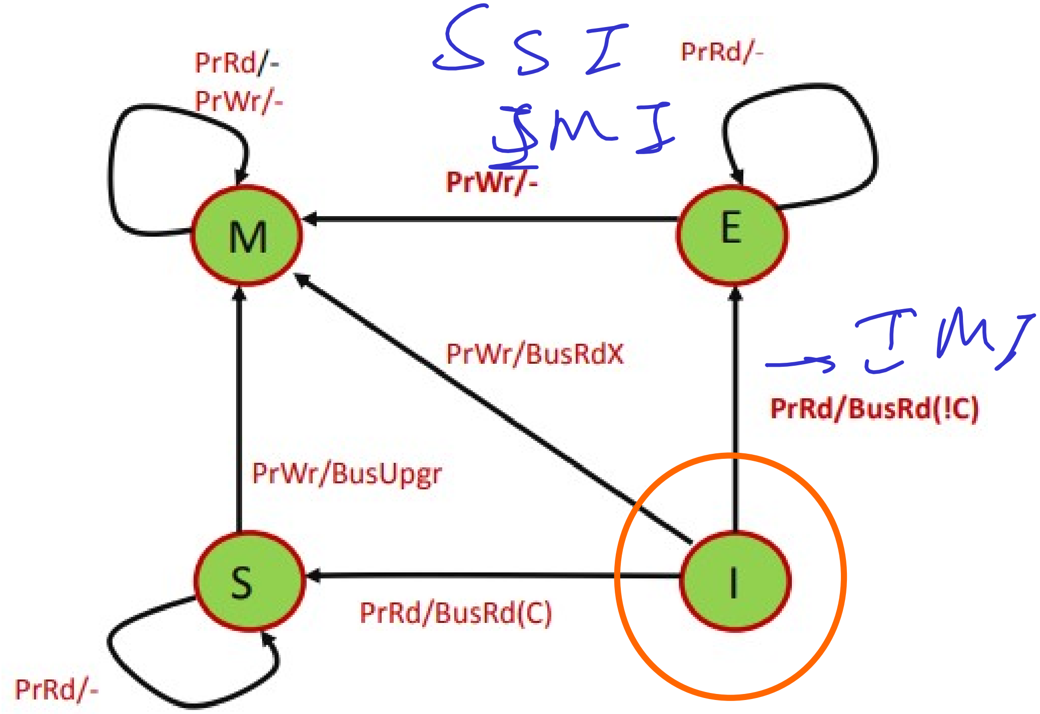


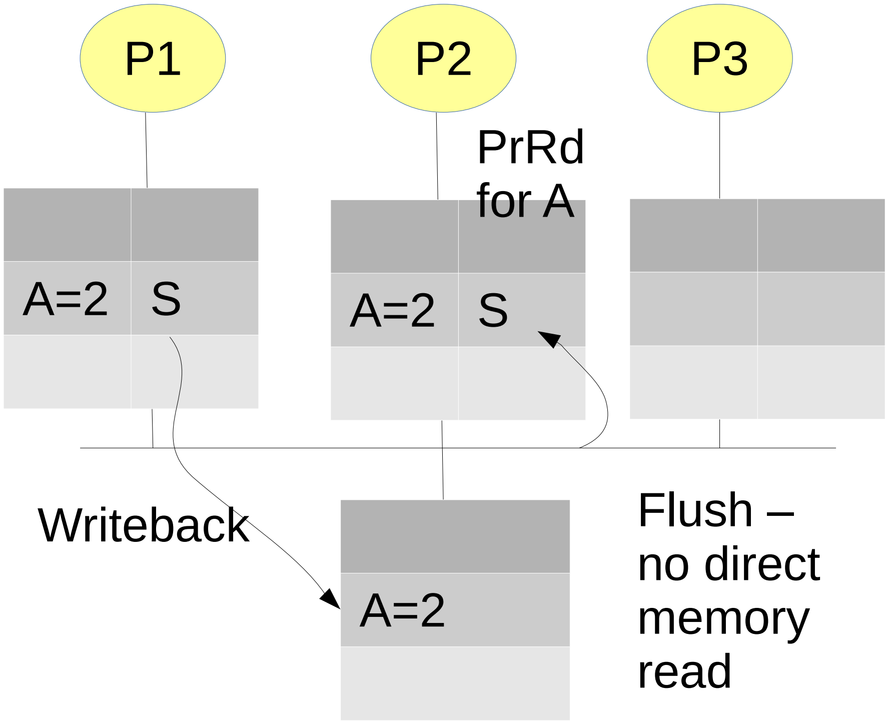
Wiki



PrRd for A

BusRd



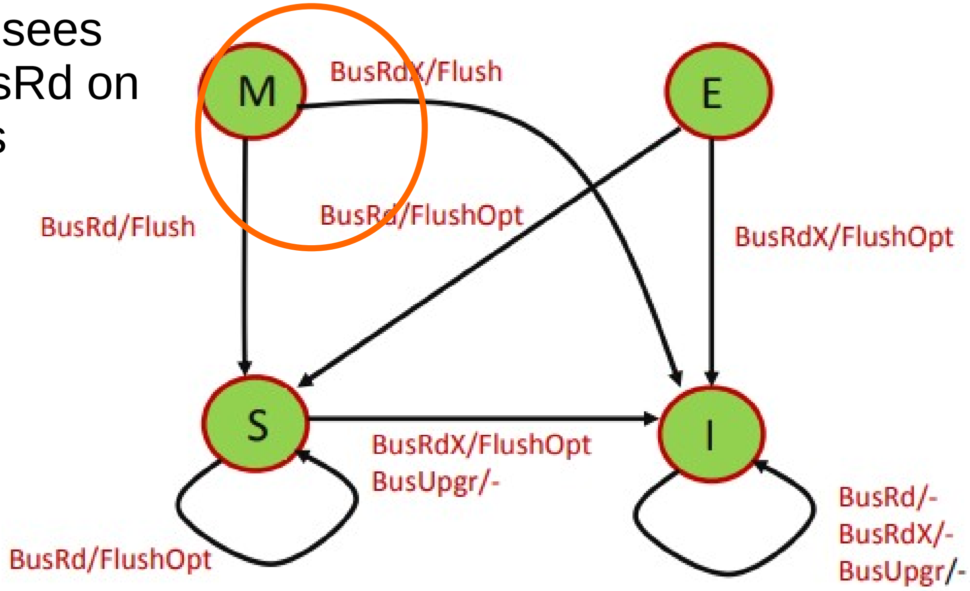


PrRd for A

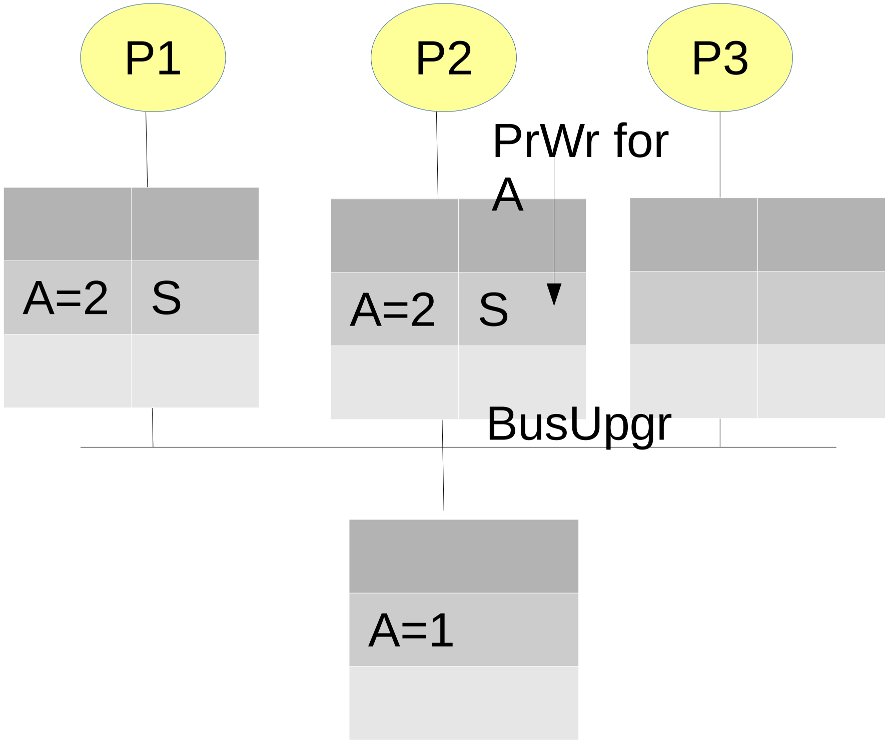
Writeback

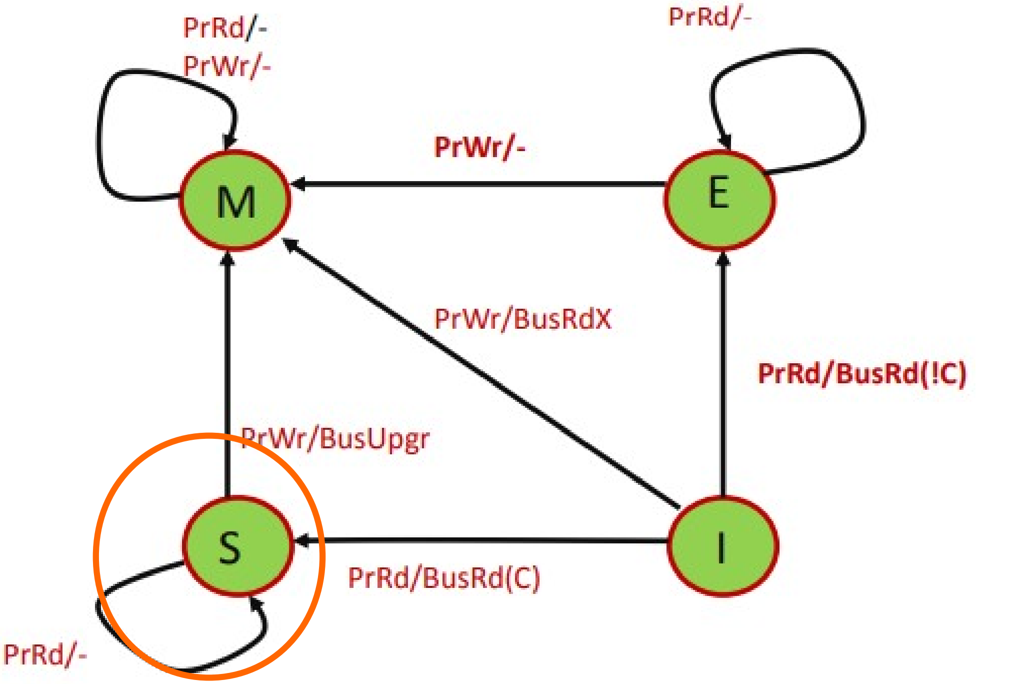
Flush – no direct memory read

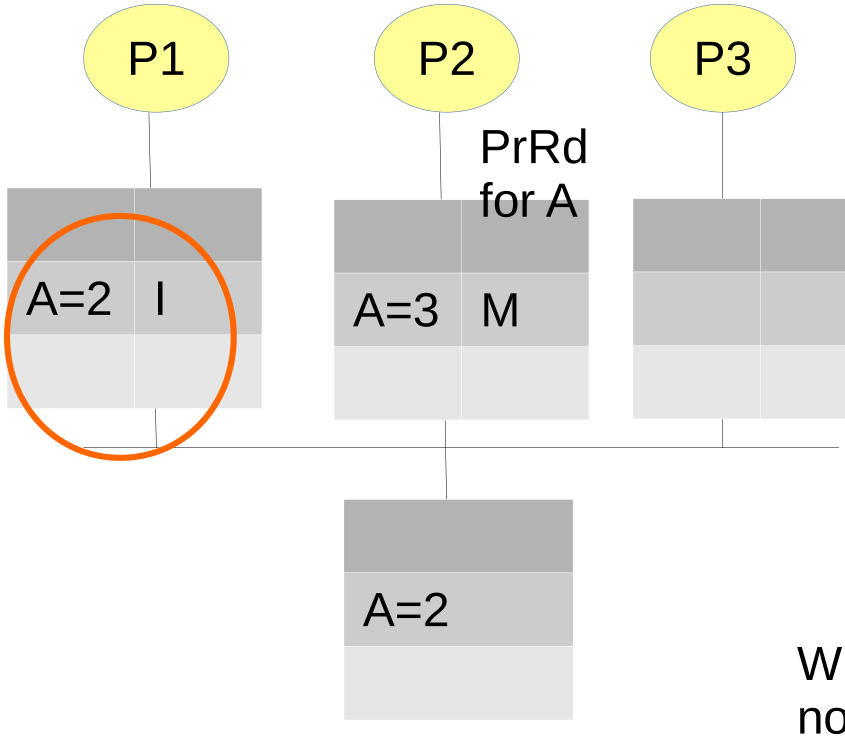
P1 sees BusRd on bus

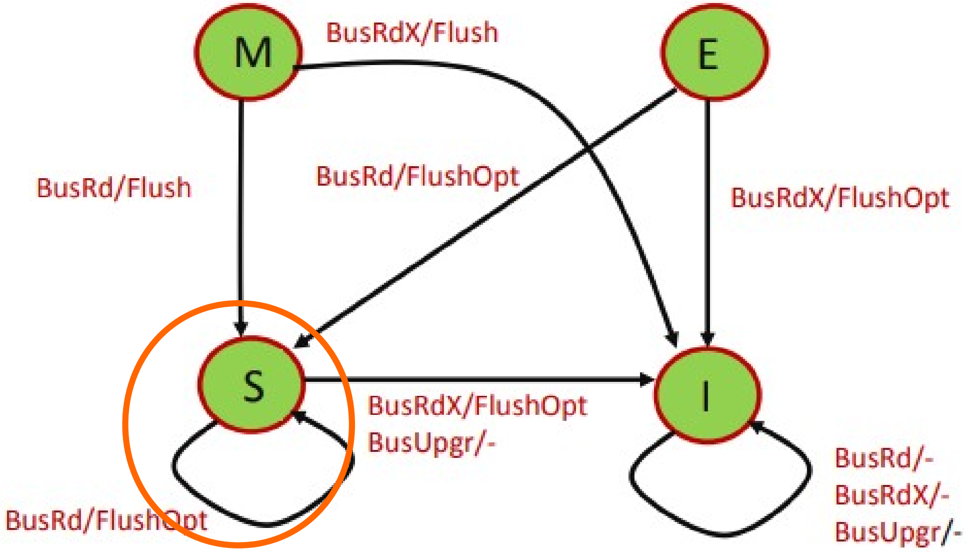


Wiki





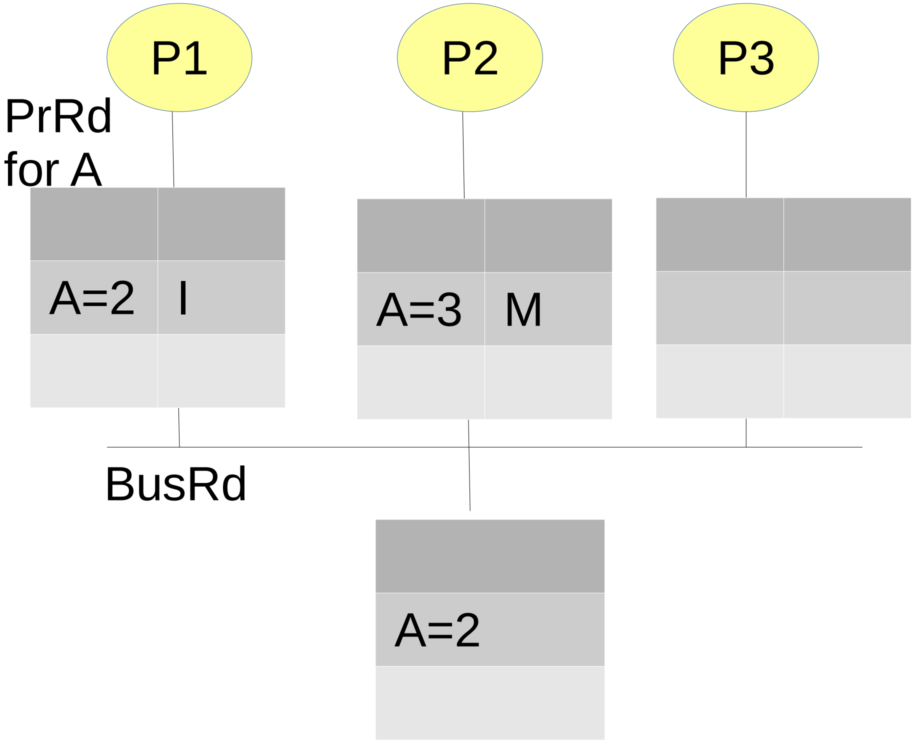




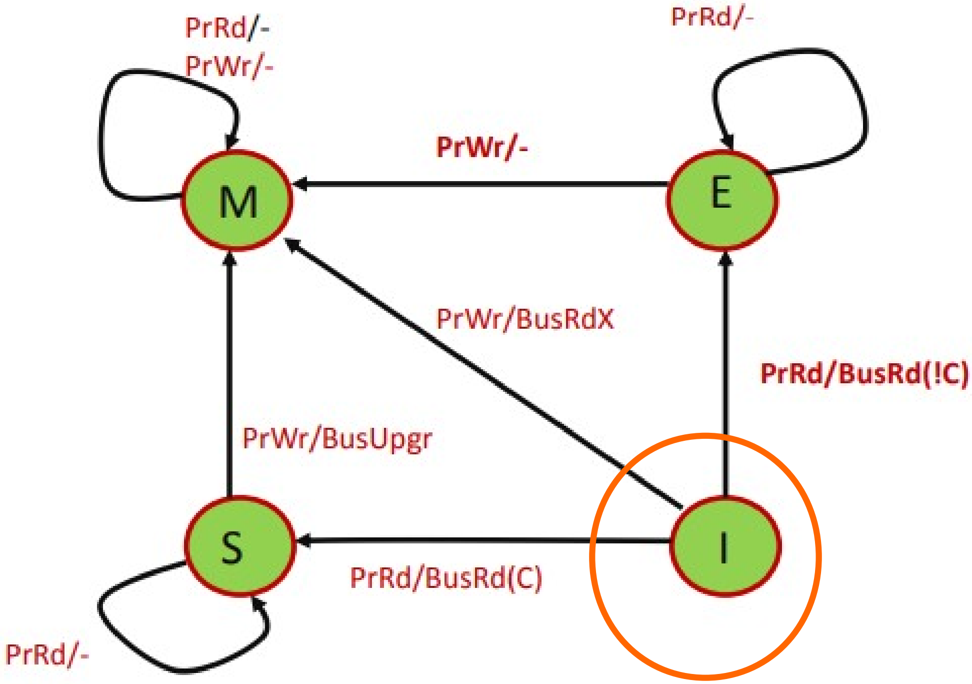
What happens if P1 reads now?

Wiki

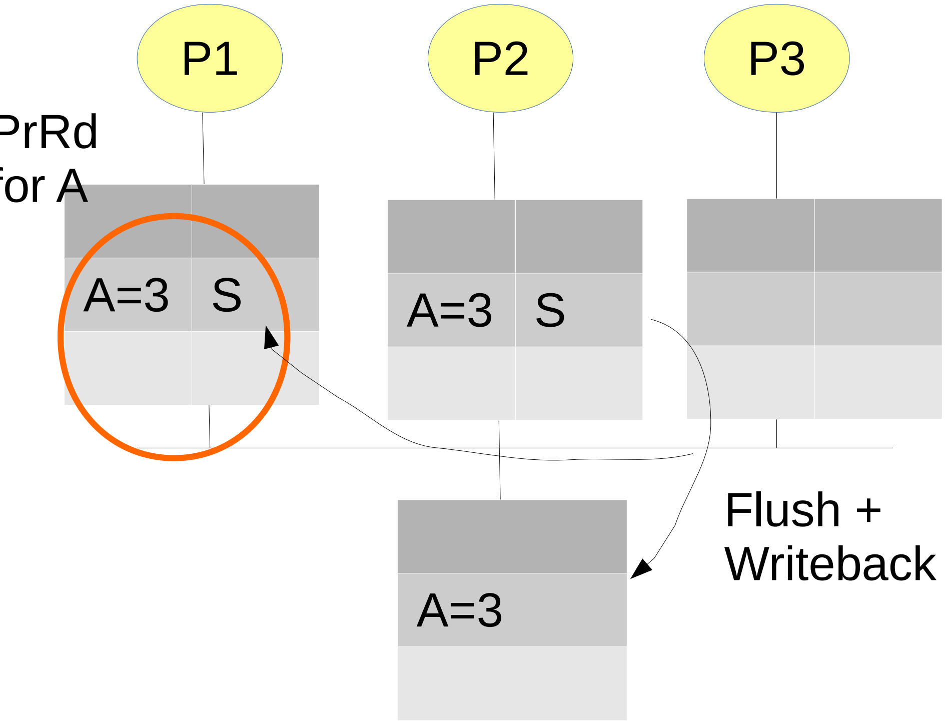
PrRd
for A



BusRd

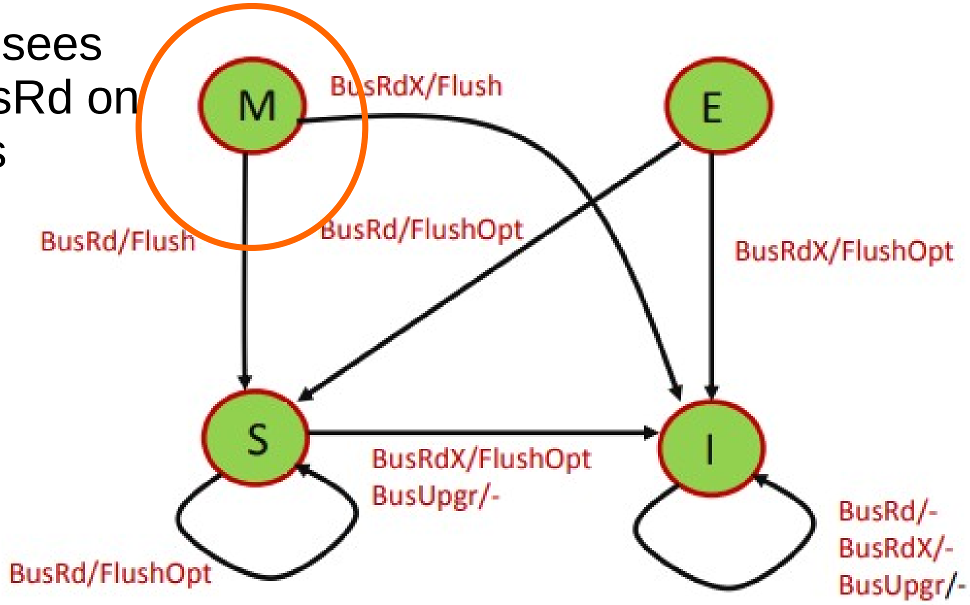


rRd
or A



Flush +
Writeback

P2 sees
BusRd on
bus



Wiki

MSI vs MESI

- MSI: A processor reads a cache line and then writes to it --> bus transactions?
  - I--> S --> M, Bus Rd --> BusUpgr
  - BusUpgr- to tell all other processors that it is about to write to X. Useless as no other processors have that line
  - To remove the need to issue this BusRdX, MESI adds E.
- MESI: If a cache line is in E state, then it is the only cache to contain that line. If the processor wants to write to that line, no bus operations need to occur.
  - Saves lot of bus transactions if a single processor is doing serial read after writes

Valid/invalid transitions

✓ addr 20 ←

- M -- > E X
- E --> M Rd/lw → sw/wr ✓
- E-->I ✓
- S-->E ✓ → X
- I-->S ✓ Read + M/S
- E-->E → Rd-Rd

I S S addr20

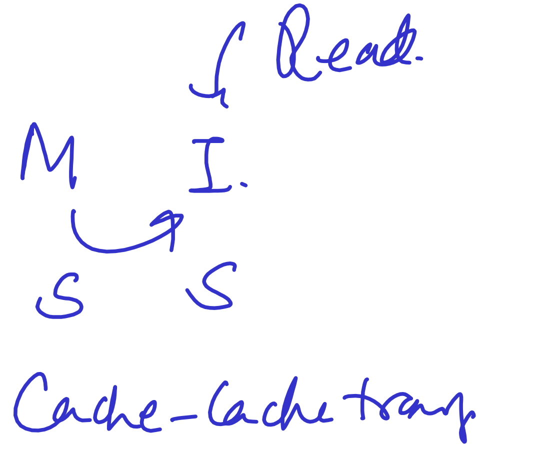
Valid/invalid transitions

- M -- > I--> E
- E --> M: On a local write
- E-->I: If another core writes to same line
- S-->I--> E
- I-->S: Read, while another core has the same block
- E-->E: Local reads

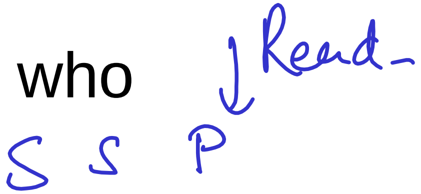
?

- Assume processor P1 and P2 trying to access cache line X. Say, P1 has X in E state, P2 has it in M state. If P2 requests for a cache write and it is a hit. What will be the new states of the 2 cache lines in a MESI protocol?
- Assume processor P1 and P2 trying to access cache line X. Say, P1 has X in M state, P2 has it in I state. If P2 requests for a cache write. What will be the new states of the 2 cache lines in a MESI protocol?

Problem with MSI or MESI



- Core 1 in M state, Core 2 is in I state
  - Core 2 Read
  - Core 1 does **Memory write**
  - Core 1--> S, Core 2--> S
  - **Memory writes on cache-cache transfers**
- Core 1 and Core 2 in S state. If Core 3 wants to read, who provides the data?
  - If one of the cores have to respond, which one? And how do we decide?
  - Memory responds --> **Memory read**
- **Memory reads/writes are expensive**
- Solution: A single “owner” of the data to respond to cache reads (avoid memory reads) and writes to memory later



MSI
MESI

MOESI protocol

Shared

owner

M → O
Modified
S → O

- Owned: Combines M and S states. Valid=1, Dirty=1. Not the only copy. Shared read access. Owner responds for reads and also writes back to memory

Read O S S

- Modified: Valid=1,Dirty=1. Only copy, only updated copy. (Writes)

M S S

- Shared: Valid=1, Dirty=0, This block is in more than one cache. Clean copy. Shared read access

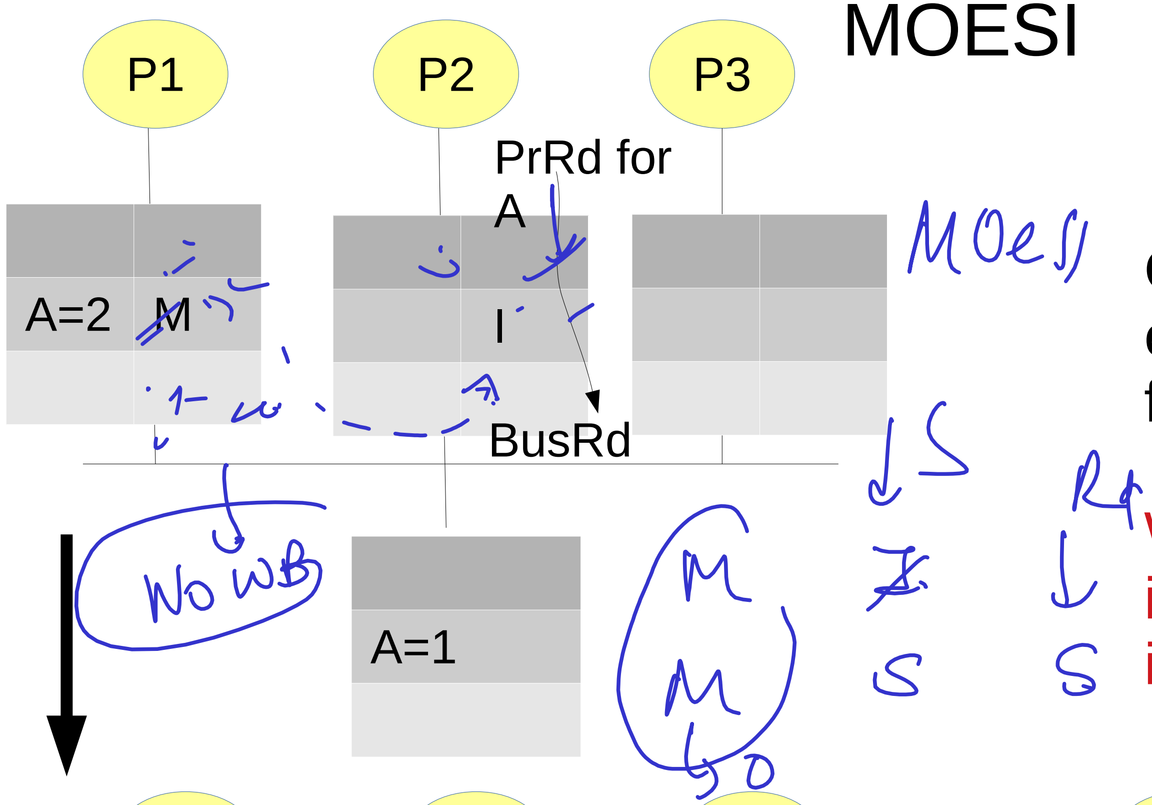
M S S

- Exclusive: Valid=1, Dirty=0, Only copy, clean copy (Reads)
- Invalid: Valid=0, This block is either not present in cache or has been invalidated by a bus request.

I I Id
I ~~E~~ → ~~I~~ S
S

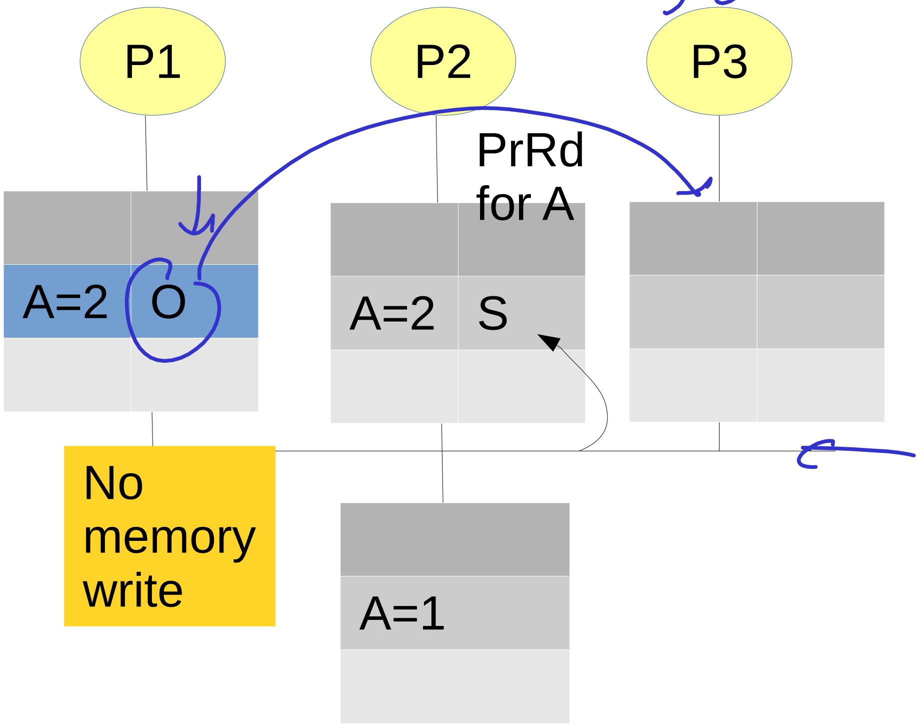
M
S → O

MOESI

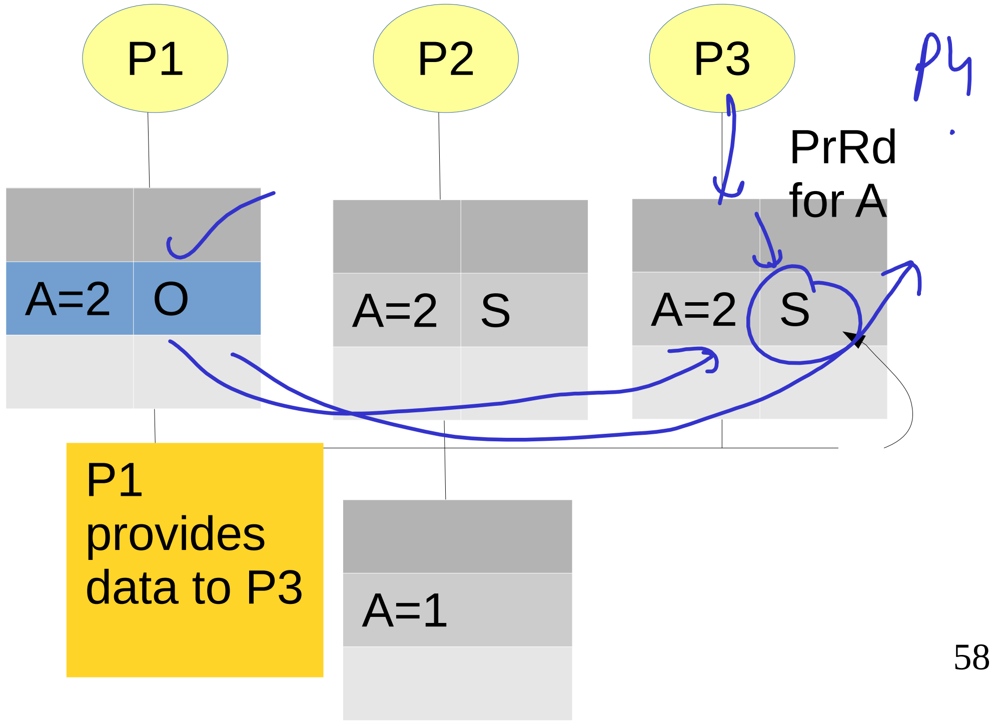


O behaves like S state, except that it is responsible for providing data to Core 3.

Writes back to memory when it gets replaced or gets invalidated.

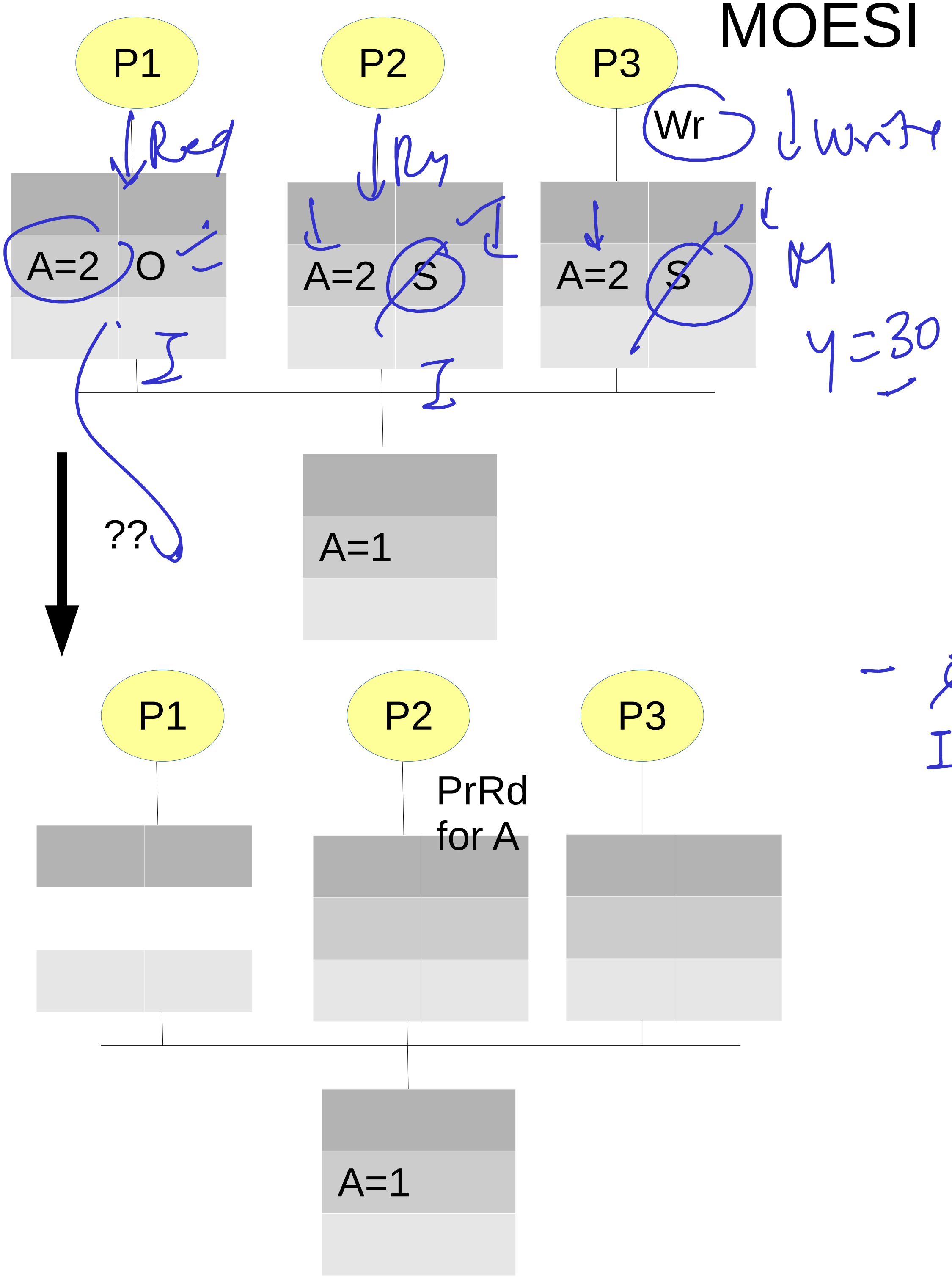


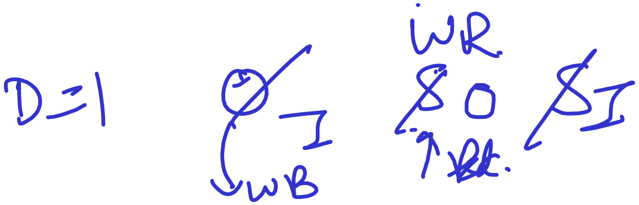
No memory write



P1 provides data to P3

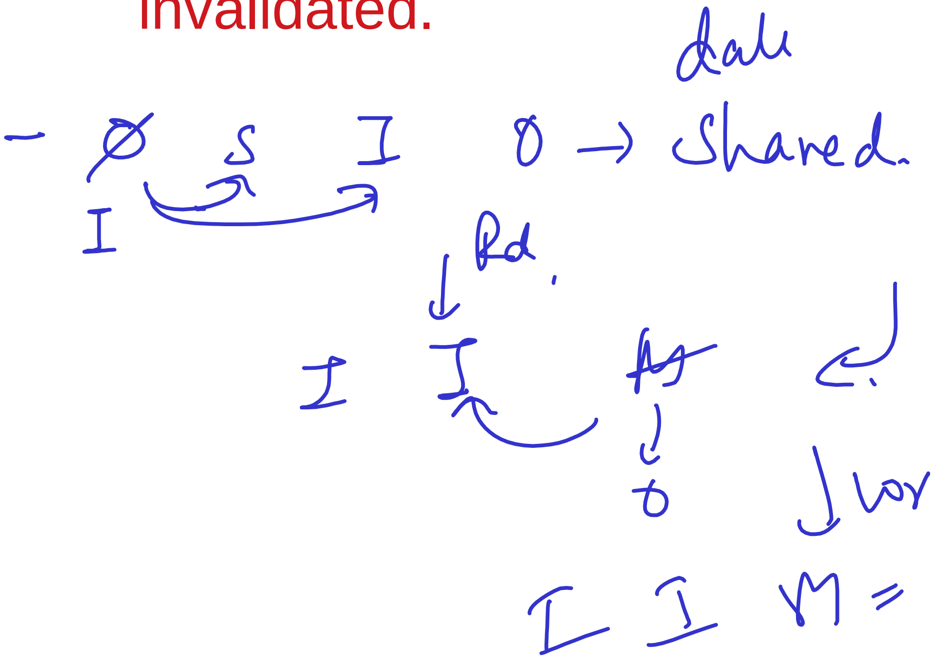
MOESI



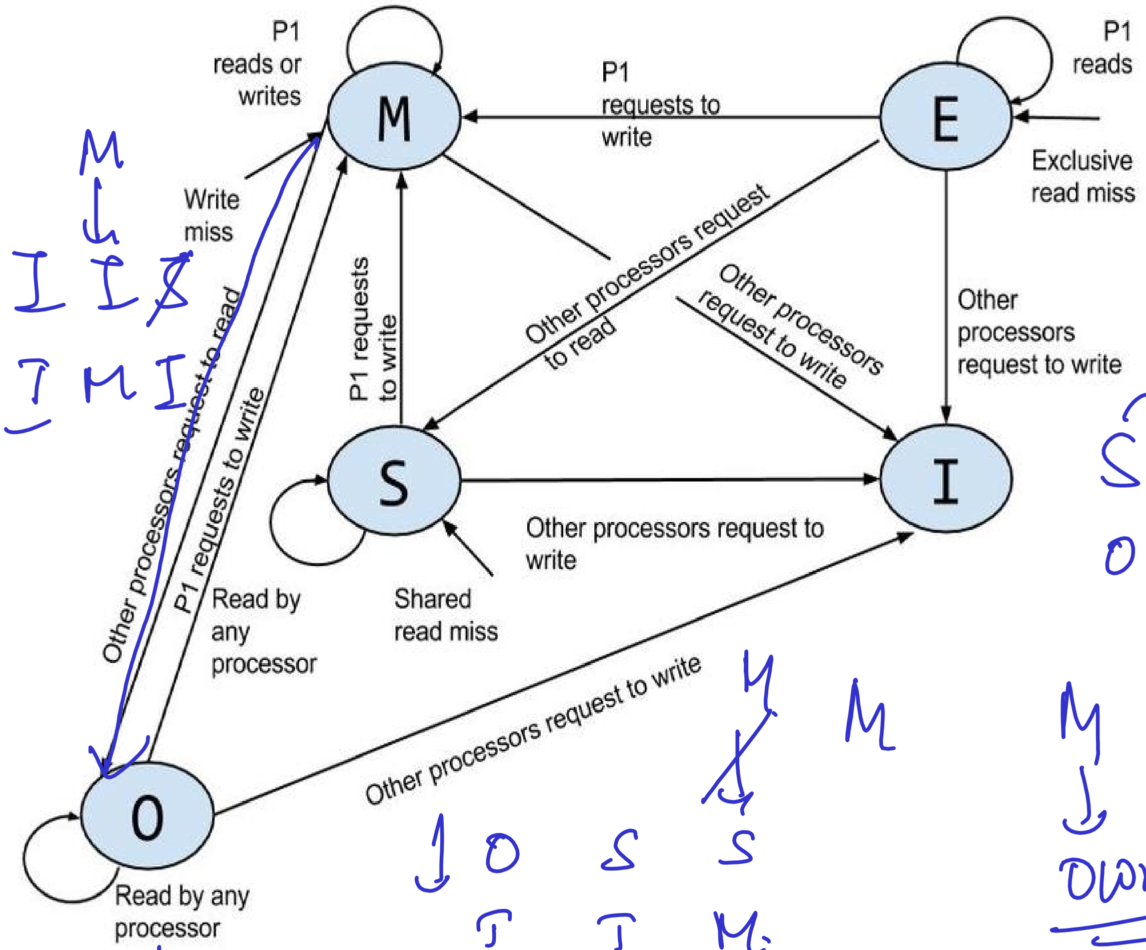


O behaves like S state, except that it is responsible for providing data to Core 3.

Writes back to memory when it gets replaced or gets invalidated.



MOESI



M
E
S
I
O
P1 reads or writes
P1 reads
P1 requests to write
Exclusive read miss
Write miss
P1 requests to write
Other processors request to read
Other processors request to write
Other processors request to write
Other processors request to write
Other processors request to read
P1 requests to write
Read by any processor
Shared read miss
Other processors request to write
Read by any processor
M
↓
I I ~~S~~
I M I

Researchgate

E I I
E S I

↓ > 1

S S I
O S I

M S I

M E S I
=
Only one
↳ > 1

M
~~↓~~
↓ O S S
I I M

M

M
↓
Owner
↳ 3rd cor

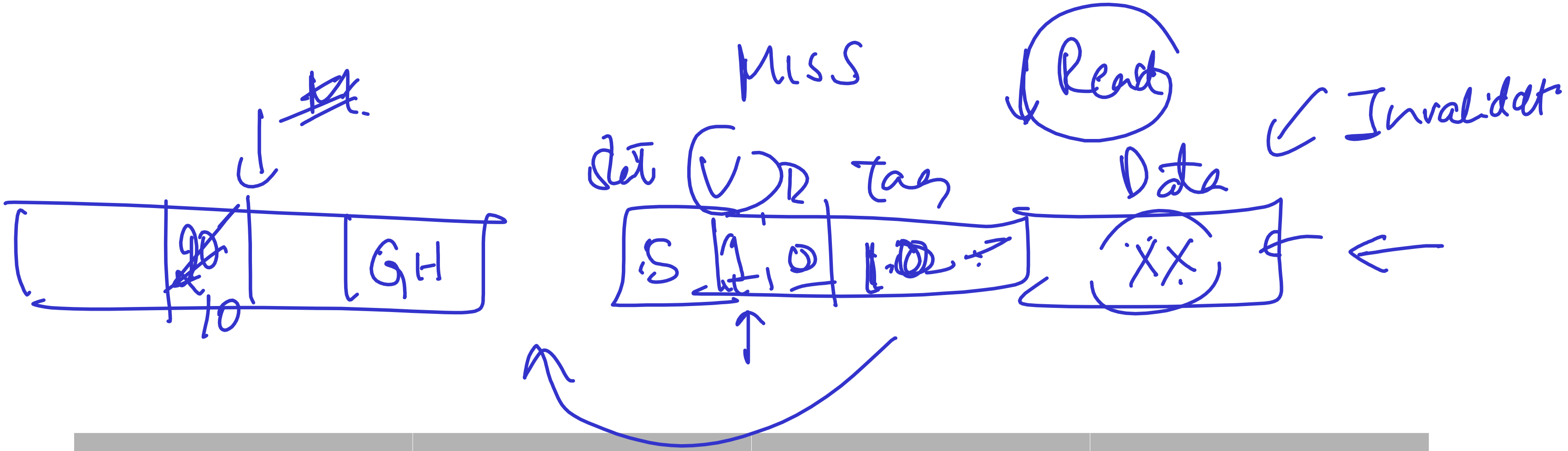
M O E S I

↓
E I I
↓
S S
✓ ✓
E ~~I~~ I
↓
S S I
S S S

- P1. M-->O when other processors read
- P1. O--> M when P1 writes
- P1. O --> I when other processor writes
- P1. O --> O, when other processors read

Summary

M	Exclusive R/W access	Dirty=1	Respond with data and update memory
E	Exclusive R/W access	Dirty=0	Reach E state upon a read from I
O	Shared access (only read)	Dirty=1	Respond with data and update memory when replaced
S	Shared access (only read)	Dirty = 0	Not responsible for providing data or updating memory
I	Invalid block		



	MSI	MESI	MOESI
RD Mem A	I -->	I -->	I -->
Wr Mem A			

64

256

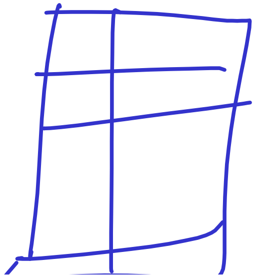
Change in states and bus transactions??

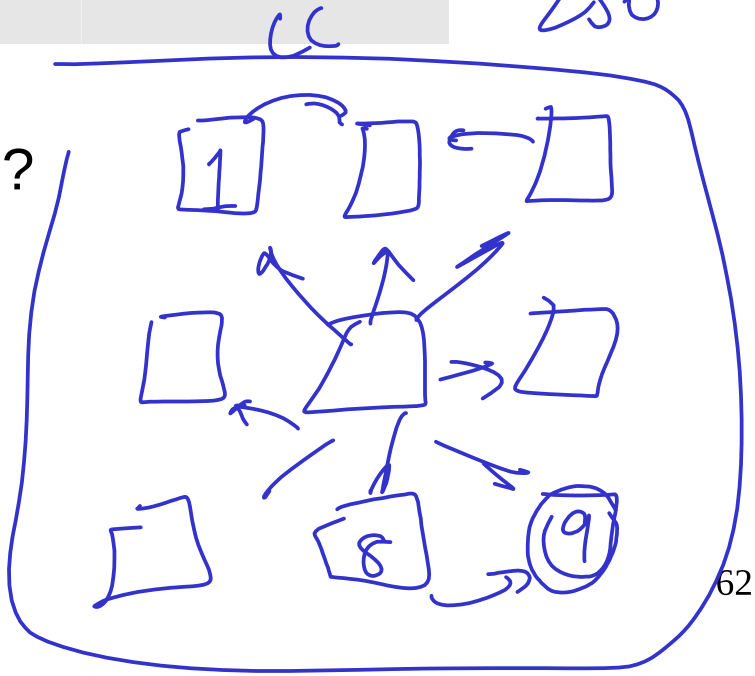
Directory

Snoop

MOESI

16 cores





	MSI	MESI	MOESI
RD Mem A	I-->S	I-->E	I-->E
Wr Mem A	S-->M (send invalidation on bus)	E-->M (No need to send invalidation on bus)	E--> M (No need to send invalidation on bus)

Distributed Shared-Memory and Directory-Based Coherence

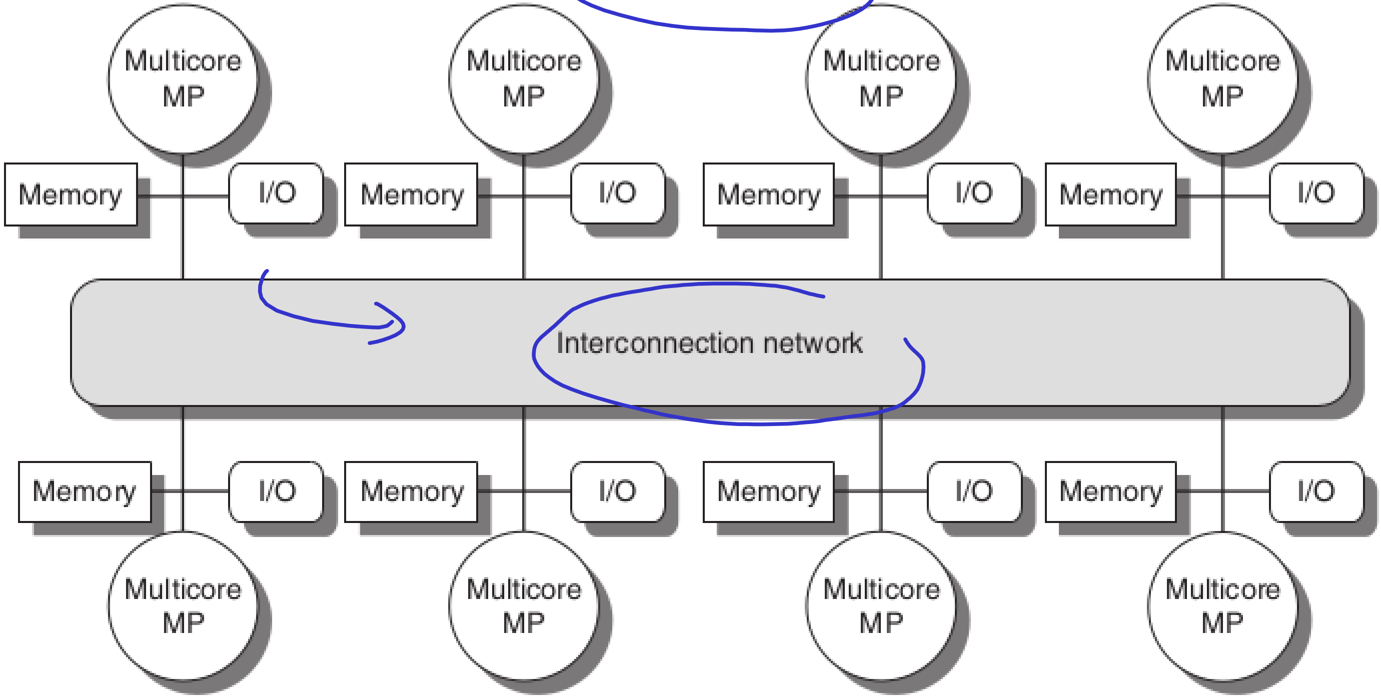
CA- QA - H&P Ch – 5

HPCA Part 5:
https://www.youtube.com/watch?v=xjRDejGF26M&list=PLAwxTw4SYaPkr-vo9gKBTid_BWpWEfuXe&index=110

“A Primer on Memory Consistency and Cache Coherence.pdf”
Synthesis lectures on CA

Disadvantages of snoop based protocols

- Snooping protocol: Communication with all caches on every cache miss and on writes
- Huge bus bandwidth for synchronization > 8 cores
- Solution: Non-broadcast network. Distributed memory, and eliminate the need for broadcast on every miss

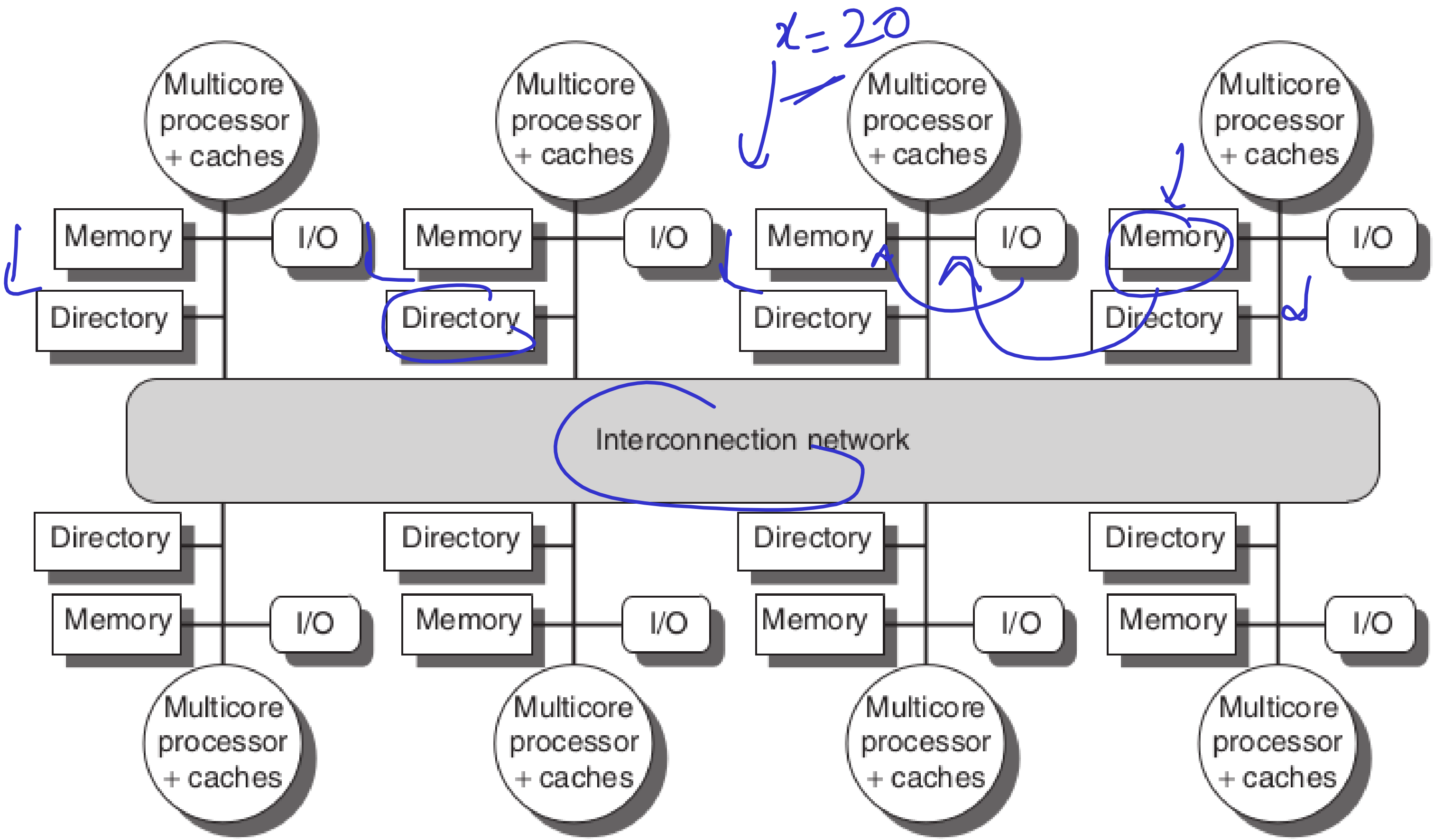


Single directory

- A directory keeps the state of every block that may be cached, which caches have copies of the block
- Directory is maintained in the last level shared cache. Eg., Intel i7 maintains this in the inclusive L3 and optionally does snoop when the bus is free (Combines directory + snoop)
- Cache controllers send all requests to the directory. For eg., if the data is present in Core2’s cache, the data is forwarded to Core2 by the directory

Distributed directory

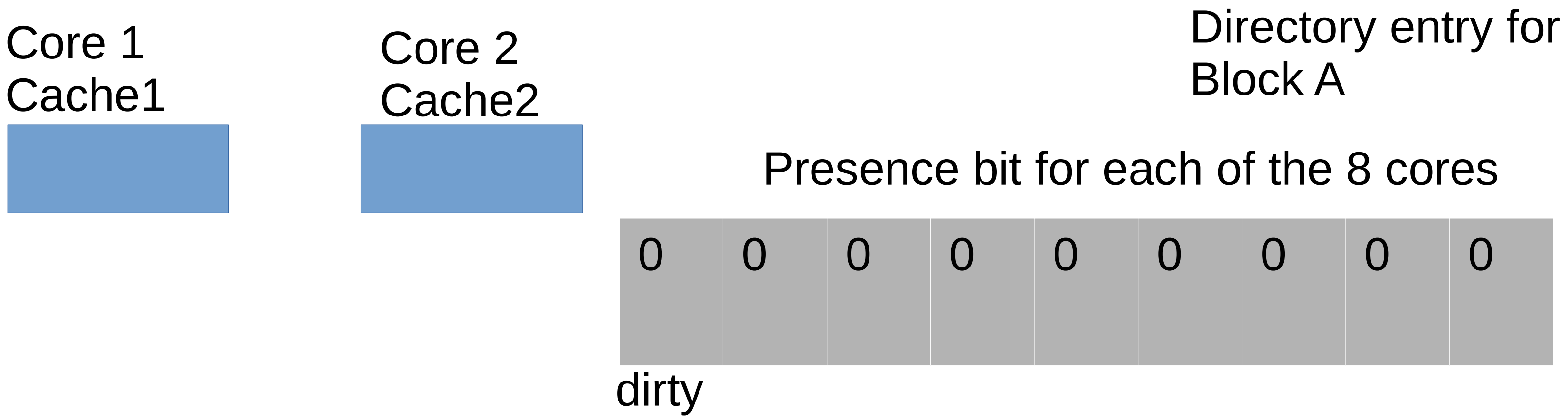
- Directory is distributed such that, the coherence protocol knows where to find the directory information
- Each directory tracks the caches that share the memory addresses of the memory in that node.



Same cache states

- Shared—One or more nodes have the block cached, and the value in memory is up to date (and in all the caches).
- Uncached—No node has a copy of the cache block.
- Modified—Exactly one node has a copy of the cache block, and it has written the block, so the memory copy is out of date. The processor is the owner of the block.
- State transitions are similar to snooping protocol
- Requests do not go on the bus. Goes to the directory

State transitions

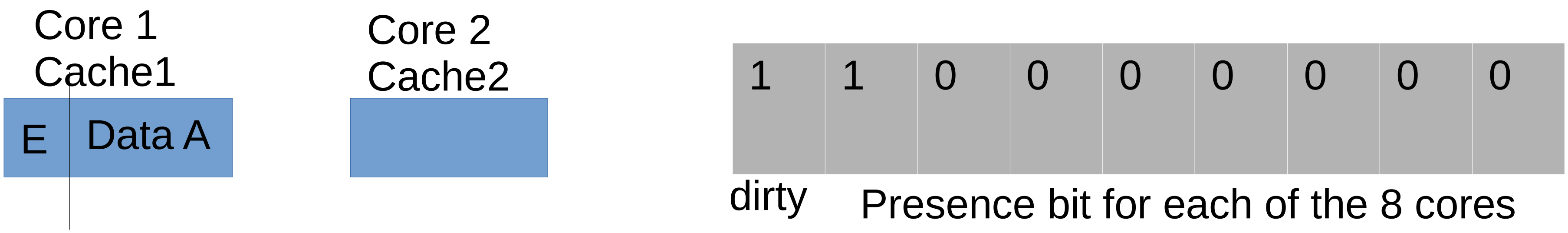


0	0	0	0	0	0	0	0	0

Core 1: Read block A --> Read miss --> Read request goes to the directory slice which has the particular entry --> Indicates not present in any core

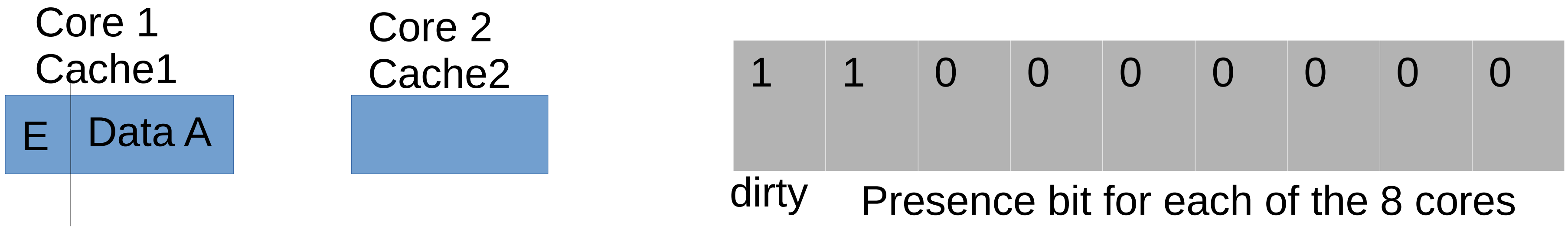
Data fetched from memory sent to Cache 1 --> Change Cache 1 from Invalid to Exclusive state (if doing MESI)--> Change directory state. (Dirty=1, since E can become M without broadcasting)

Advantage: Parallely Cache 2 could be interacting with another directory without bus bottleneck



1	1	0	0	0	0	0	0	0

State transitions


Core 1
Cache1

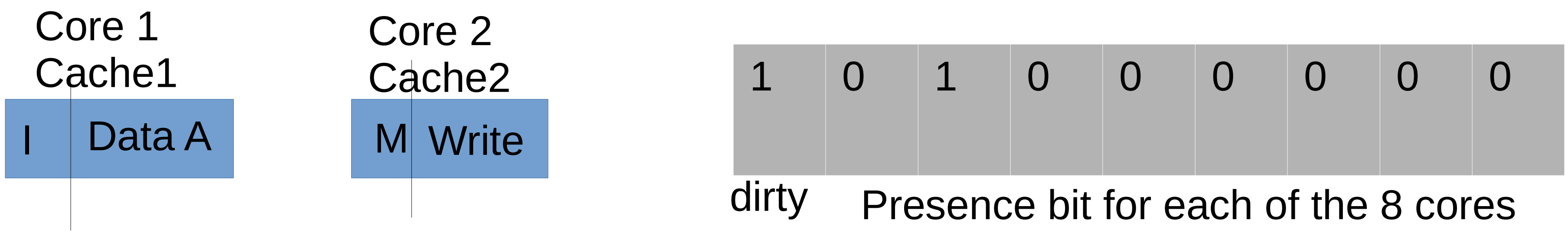
E Data A

Core 2
Cache2

1	1	0	0	0	0	0	0	0

dirty Presence bit for each of the 8 cores

Core 2: Write to block A --> Sent to directory --> Block is in Cache 1. Directory forwards write request to Cache 1. Cache 1 **can invalidate and send acknowledgement back to the directory**


Core 1
Cache1

I Data A

Core 2
Cache2

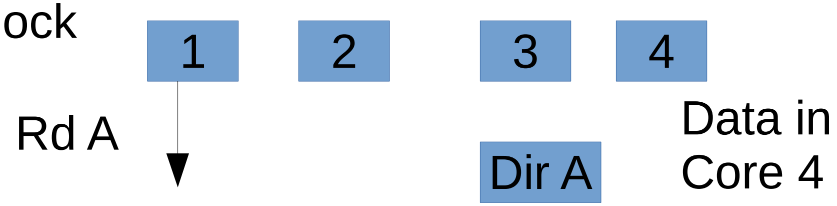
M Write

1	0	1	0	0	0	0	0	0

dirty Presence bit for each of the 8 cores

Local node is the node where a request originates
Home node: memory location/directory entry of address reside
[Physical address is distributed so the node that contains the memory and directory for a given physical address is known]
A remote node is the node that has a copy of a cache block

1: Local, 3: Home, 4: Remote
P = requesting node, A = requested address



1

2

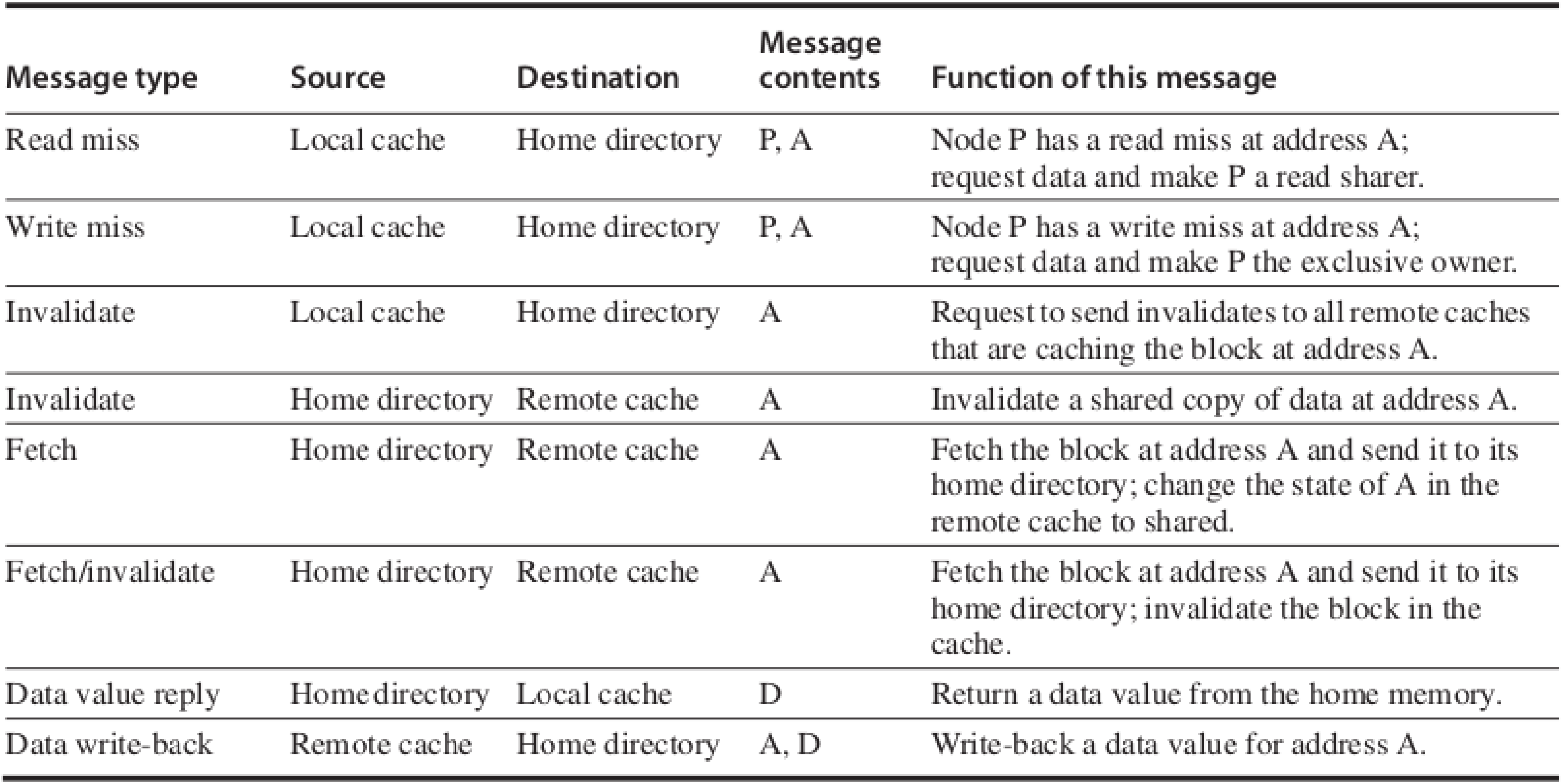
3

4

Rd A

Dir A

Data in Core 4



Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write-back a data value for address A.

A 4-core example with directory

- HPCA Part 5 Directory example: https://www.youtube.com/watch?v=lZZYILcQ68Y&list=PLAwxTw4SYaPkr-vo9gKBTid_BWpWEfuXe&index=113