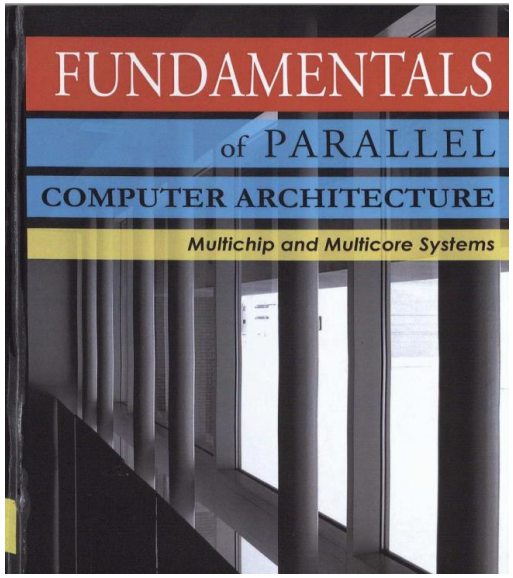
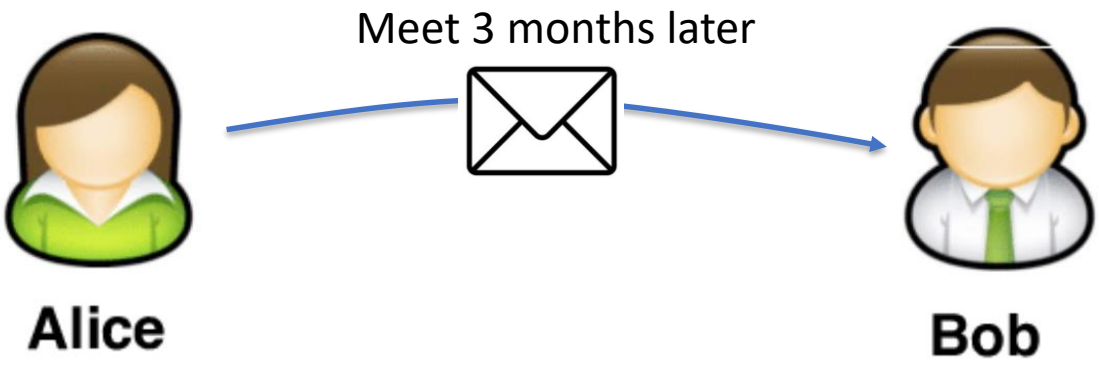
Cache Coherence

课程内容

- **参考材料：**
  - Fundamentals of Parallel Computer Architecture. Chapter 7 and 8.



Cache Coherence Problem



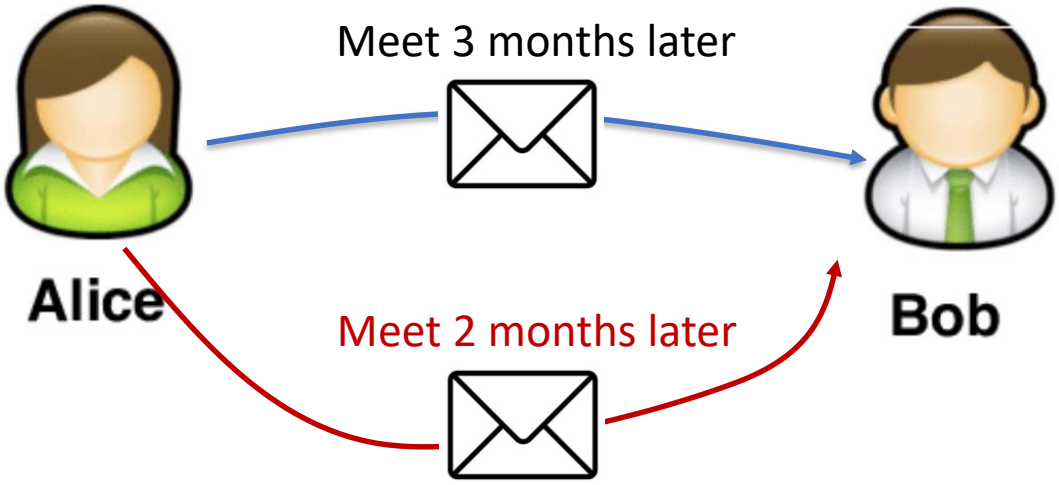
- **Illustration 1:**
  - "You (A) **met a friend B yesterday** and made an appointment to meet her at a place exactly **one month from now**.
  - Later you found out that you have an emergency and cannot possibly meet on the agreed date.
  - You want to change the appointment to three months from now."
- **Constraints**
  - The only mode of communication is **mail**
  - In your mail, you can only write the new appointment date and nothing else

Cache Coherence Problem

- **Question 1: How can you ensure that A will meet B?**

- **Principle 1:** When a processor performs a write operation on a memory address, the write must be **propagated to all other processors**
  - The **write propagation** principle

Cache Coherence Problem



- **Illustration 2:**
  - You have **placed a mail** on the mailbox telling B to change the meeting month to 3 months from now.
  - Unfortunately, you find out again that you cannot meet 3 months from now
  - You need to change it to 2 months from now.

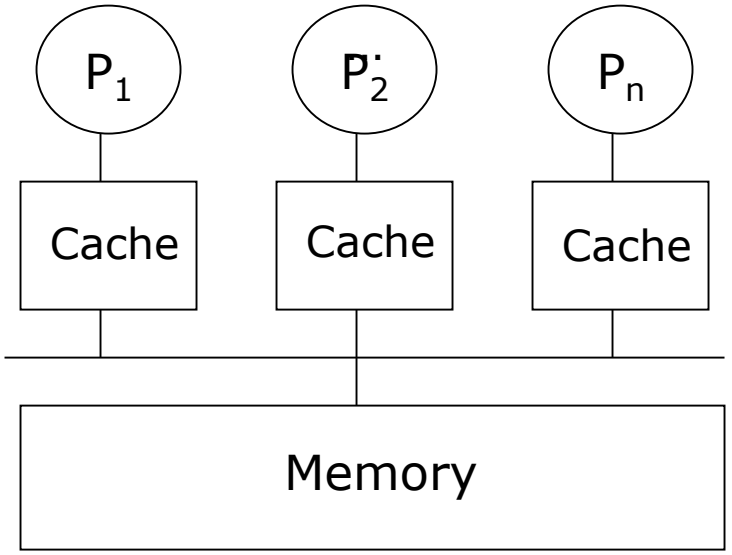
Cache Coherence Problem

- **Question 2: How can you ensure that A will meet B two months from now?**

- **Principle 2:** When there are two writes to the same memory address, the order in which the writes are seen by all processors must be the same.
  - The **write serialization** principle

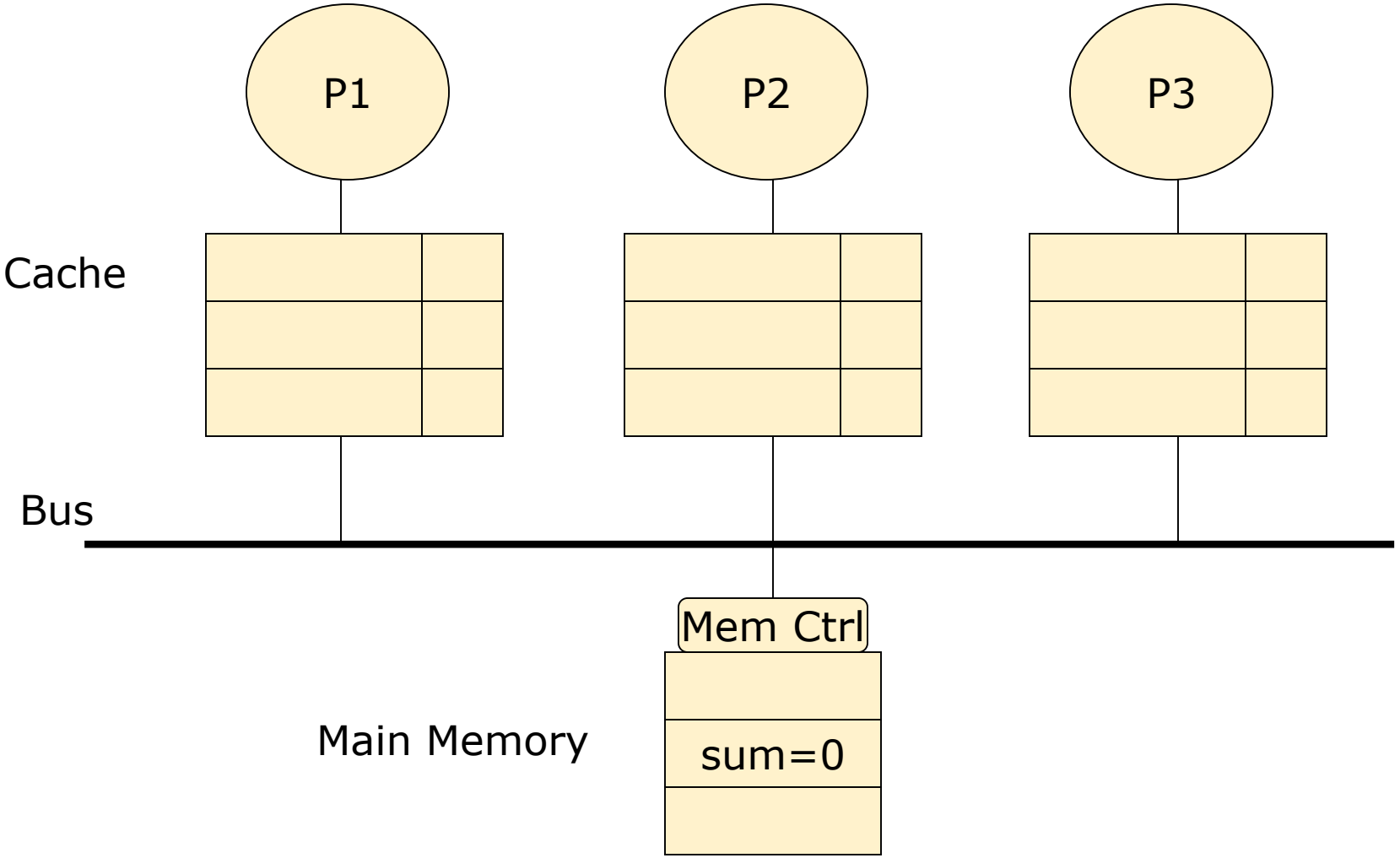
Will This Parallel Code Work Correctly?

```
    sum = 0;
    begin parallel
    for (i=0; i<2; i++) {
      lock(id, myLock);
      sum = sum + a[i];
      unlock(id, myLock);
    end parallel
    Print sum;

Suppose a[0] = 3 and a[1] = 7
```

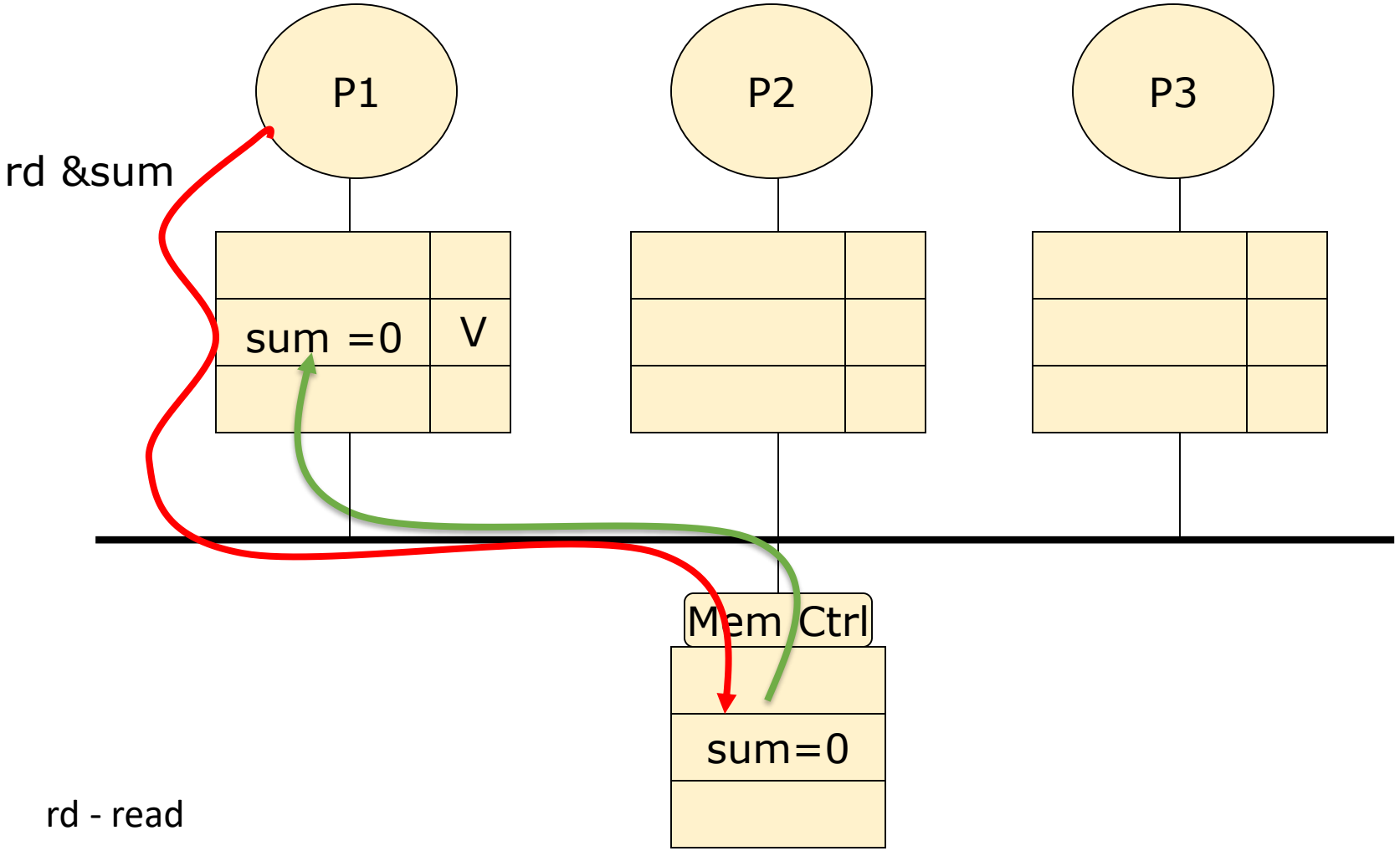


Will it print sum = 10?

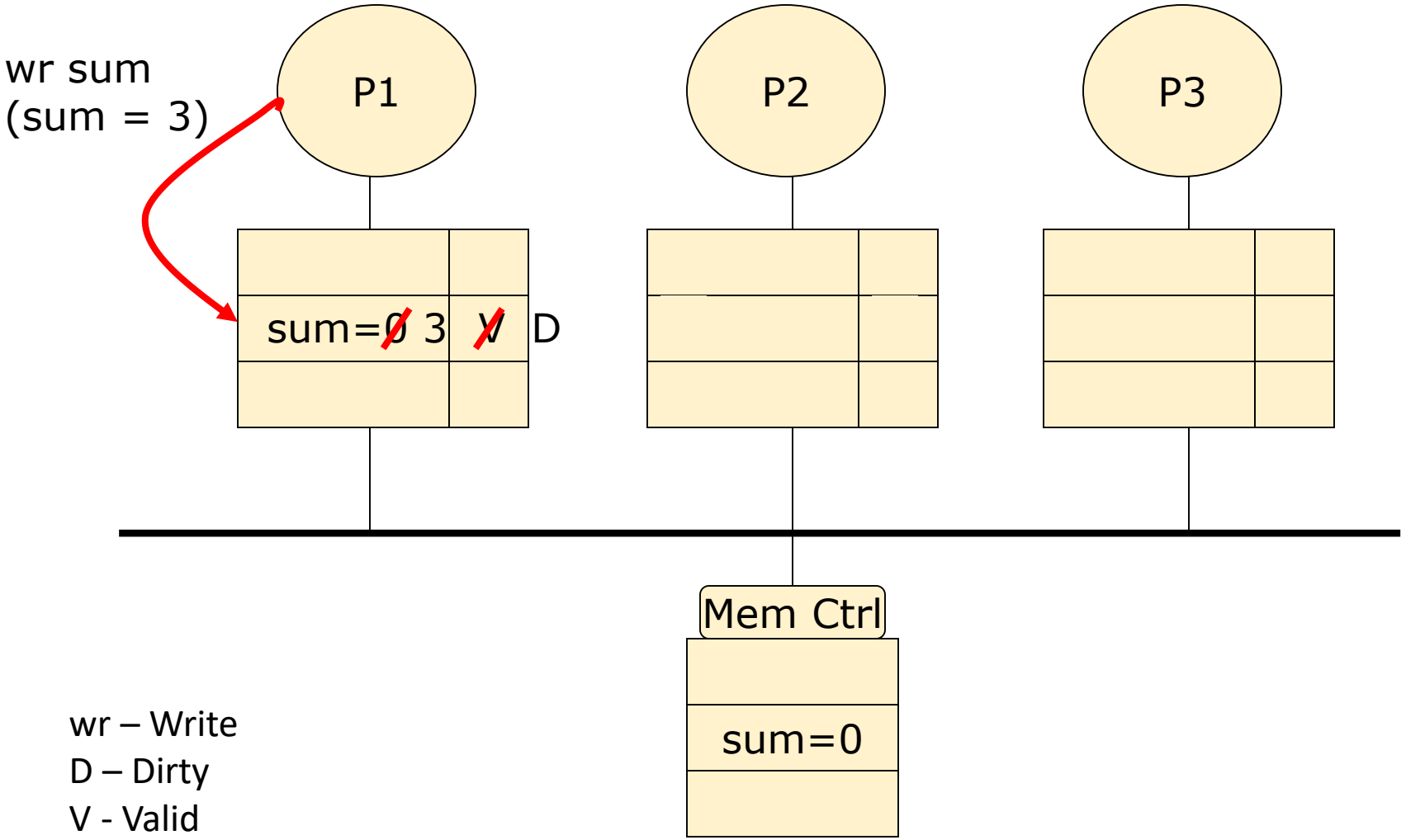
Cache Coherence Problem Illustration



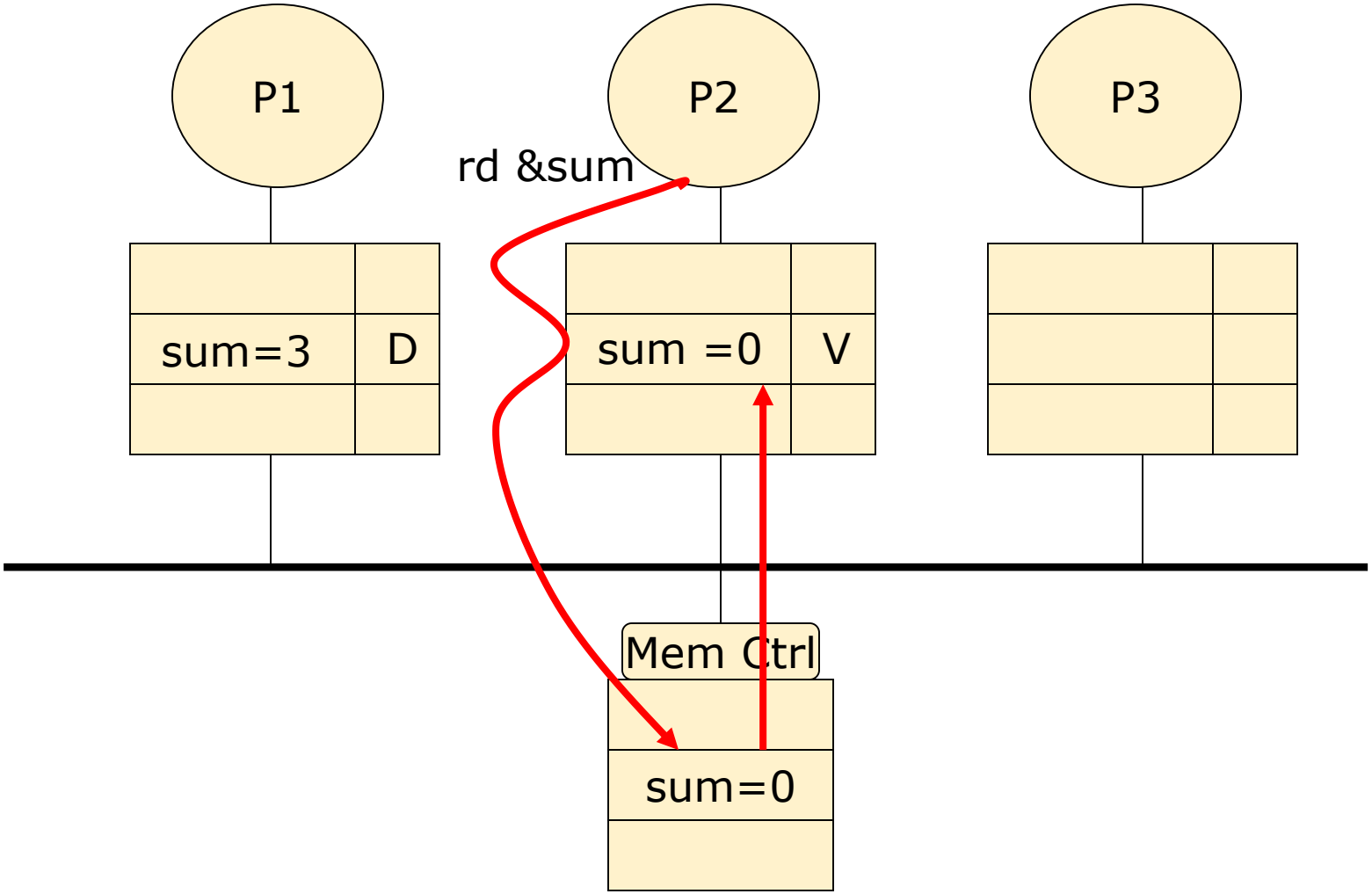
Cache Coherence Problem Illustration



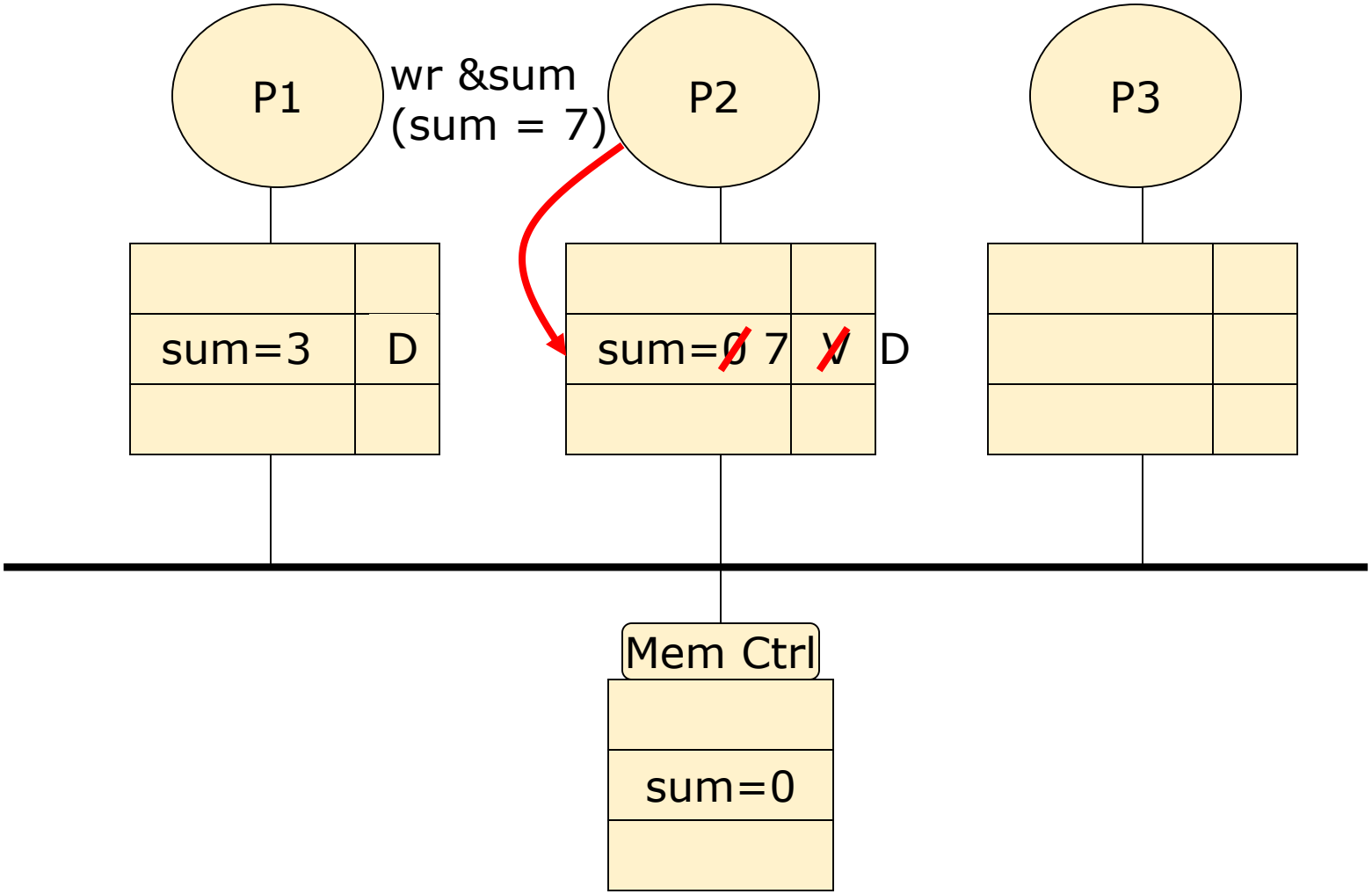
Cache Coherence Problem Illustration



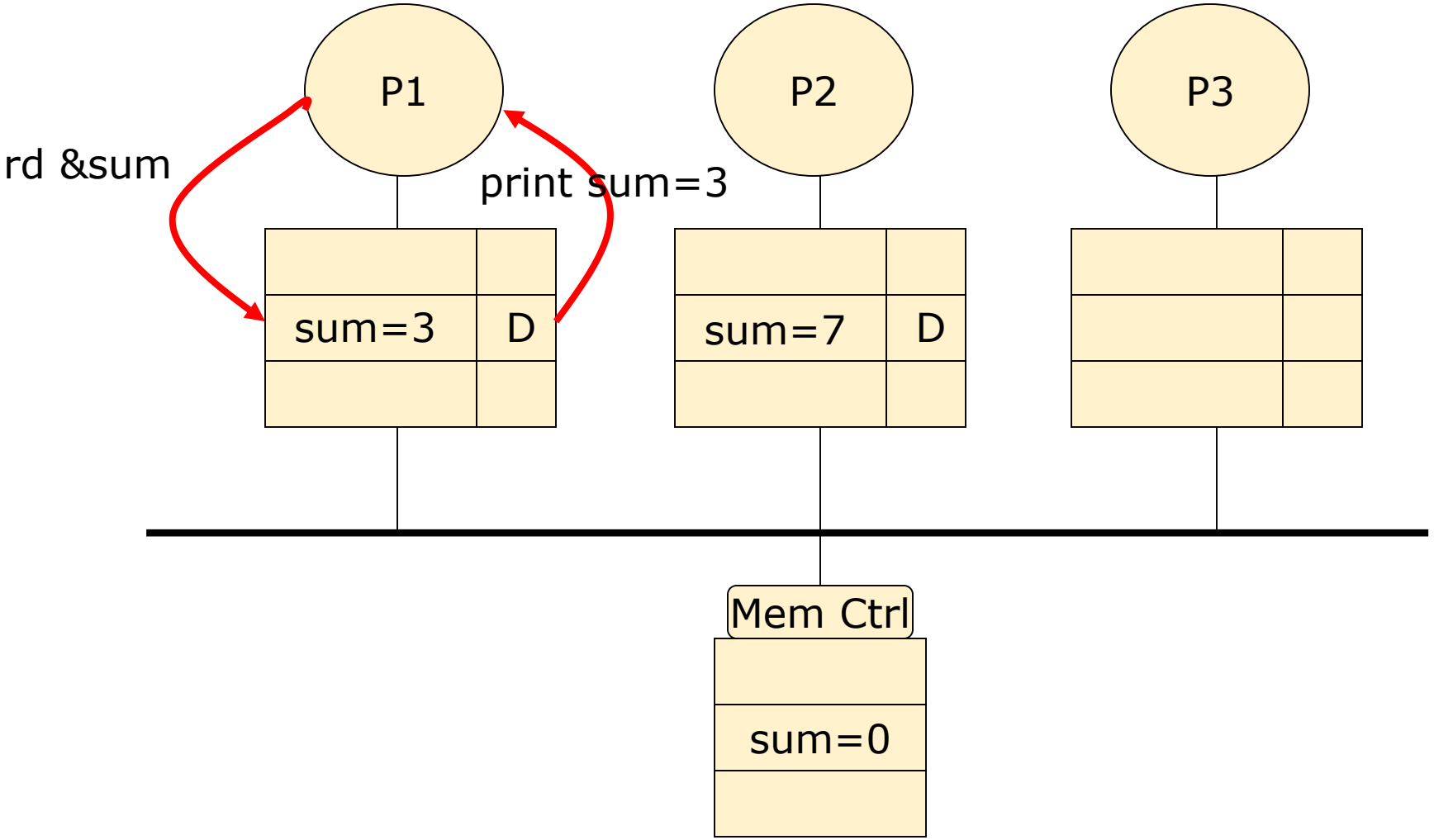
Cache Coherence Problem Illustration



Cache Coherence Problem Illustration



Cache Coherence Problem Illustration



Questions

- **Do P1 and P2 see the same value of sum at the end?**
- **How is the problem affected by cache organization?**
  - For example: What happens if we use write-through caches instead?
- **What if we do not have caches, or sum is uncachable. Will it work?**

Main Observations

- **P1 - Write propagation:** values written in one cache must be propagated to other caches
- **P2 - Write serialization:** values written successively on the same data must be seen in the same order by all caches

- **Need a protocol to ensure both properties**
  - Called ***cache coherence protocol***
- **Cache coherence protocol can be implemented in hardware or software**
  - Overheads of software implementations are often very high

Cache Coherence Problem

- **What do we do about it?**
  - Organize the mem hierarchy to make it go away
  - Detect and take actions to eliminate the problem

- **Cache coherence protocols**
  - **Snooping**
    - Each core tracks sharing status of each block
  - **Directory based**
    - Sharing status of each block kept in one location

Snooping-Based Coherence Protocol

Basic Idea: Snooping-Based Coherence

- ***Processor-Side:***
  - Add a snooper to each node to snoop all bus transactions
  - Caches (via cache controllers) react by changing cache line states

- ***Memory-Side:***
  - Memory controller (MC) also snoops bus transactions
  - MC reacts by deciding whether to read line from memory and return it

Coherence Correctness Requirements

- **Write propagation: on a snooped write,**
  - ***Update***: snooped value updates local copy
  - ***Invalidate***: local copy invalidated to force future read to re-fetch the line

- **Write serialization**
  - Bus provides global ordering of writes
  - Each snooper reacts in bus order

Recall: CPU读Cache

- **CPU读Cache**
  - **Read through**
    - 即直接从内存中读取数据；
  - **Read allocate**
    - 先把数据读取到Cache中，再从Cache中读数据

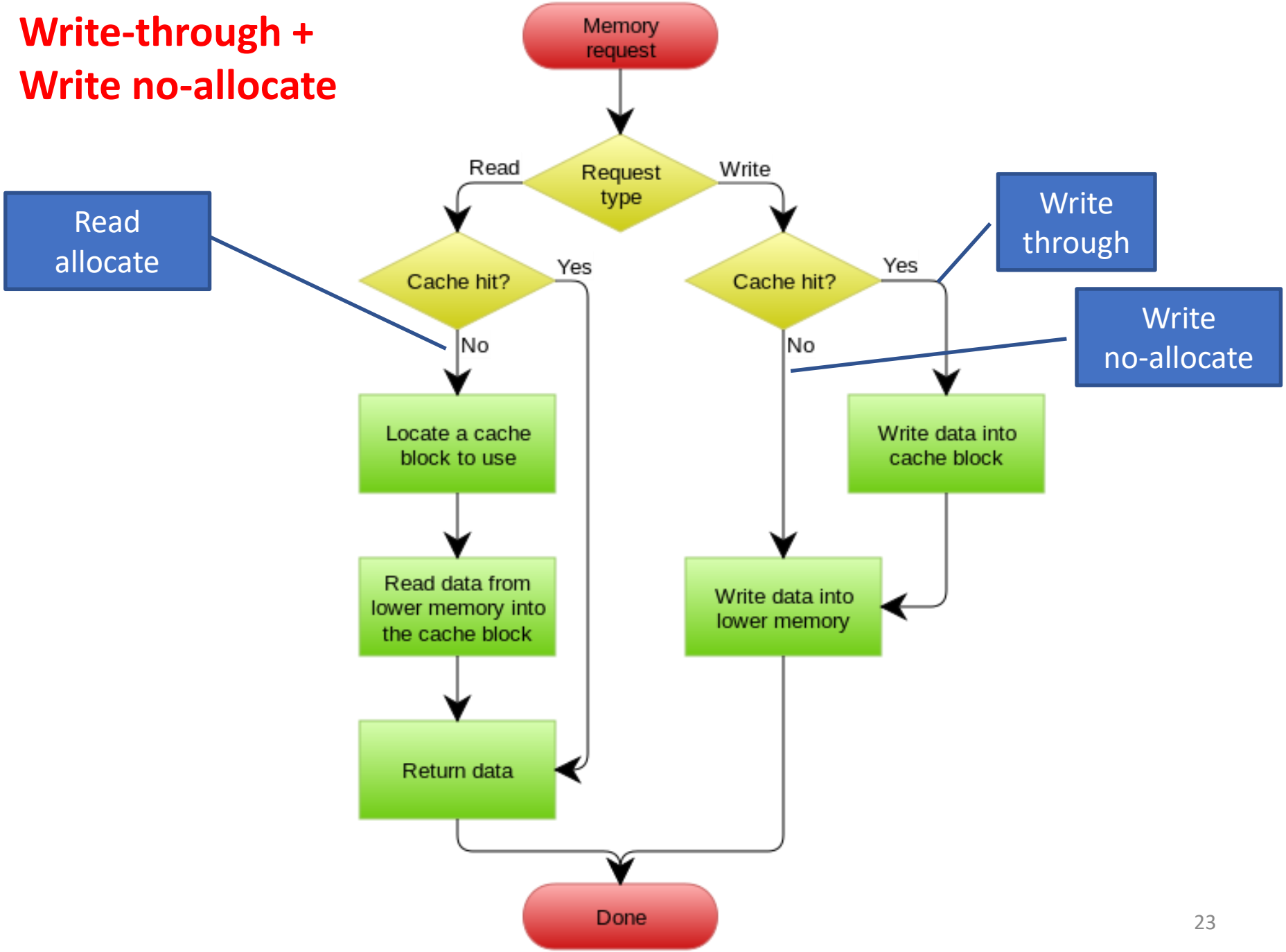
Recall: CPU写Cache

- **若cache hit命中，有两种处理方式：**
  - **Write-through（直写模式）**
    - 在数据更新时，把数据同时写入Cache和内存
    - 优点是操作简单；
    - 缺点是因为数据修改需要同时写入存储，数据写入速度较慢
  - **Write-back（回写模式）**
    - 在数据更新时只写入缓存Cache，只在数据被替换出缓存时，被修改的缓存数据才会被写到内存
    - 优点是数据写入速度快，因为不需要写内存
    - 缺点是一旦更新后的数据未被写入后端存储时出现系统掉电的情况，数据将无法找回

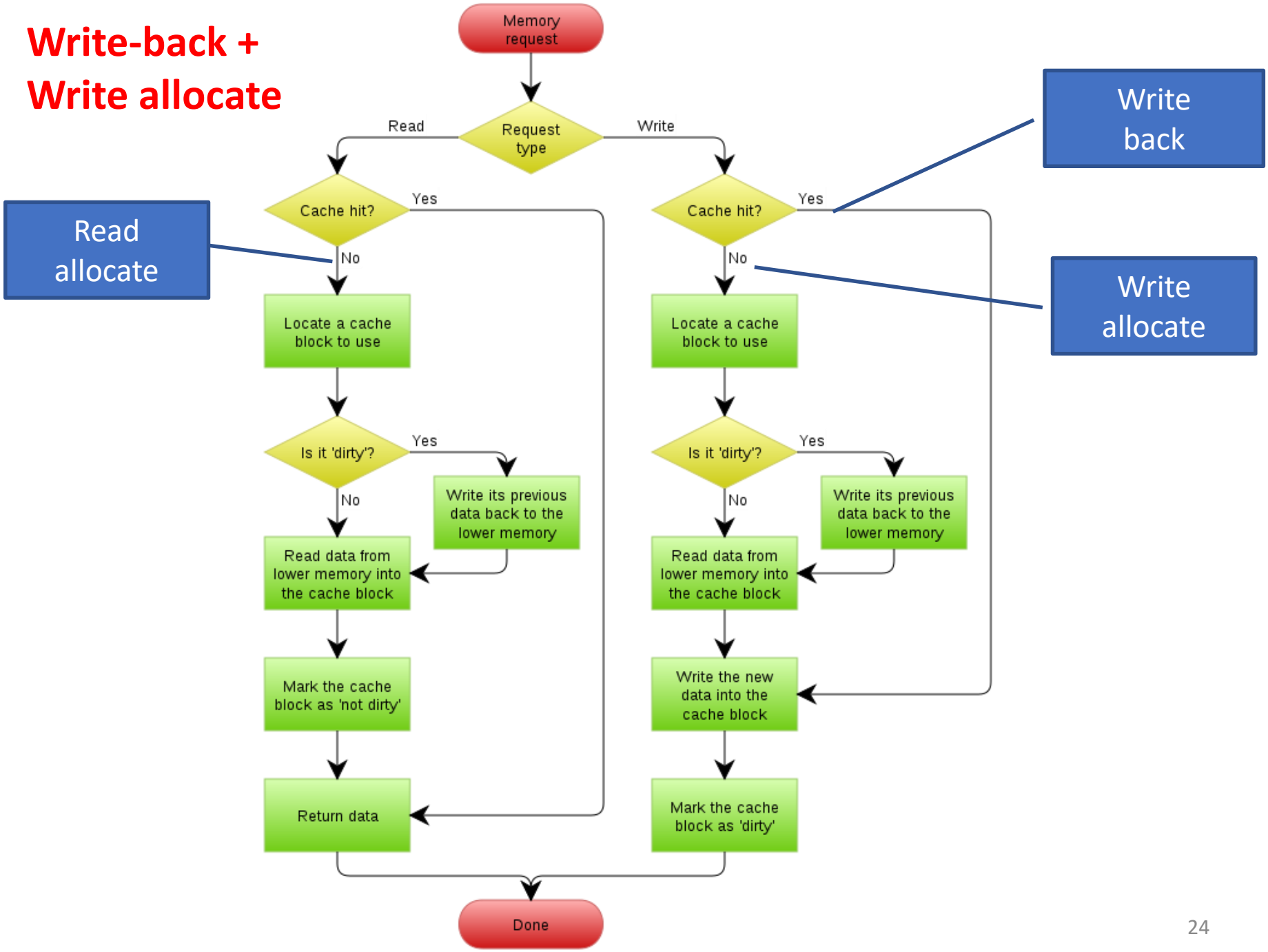
Recall: CPU写Cache

- **若cache miss，有两种处理方式：**
  - **Write allocate（写分配模式）**
    - 把要写的地址所在的块先从内存调入cache中，然后写cache
  - **Write no-allocate (也称为no-write allocate，非写分配模式)**
    - 并不将写入位置的数据读入缓存，直接把要写的数据写入到内存中

Write-through + Write no-allocate



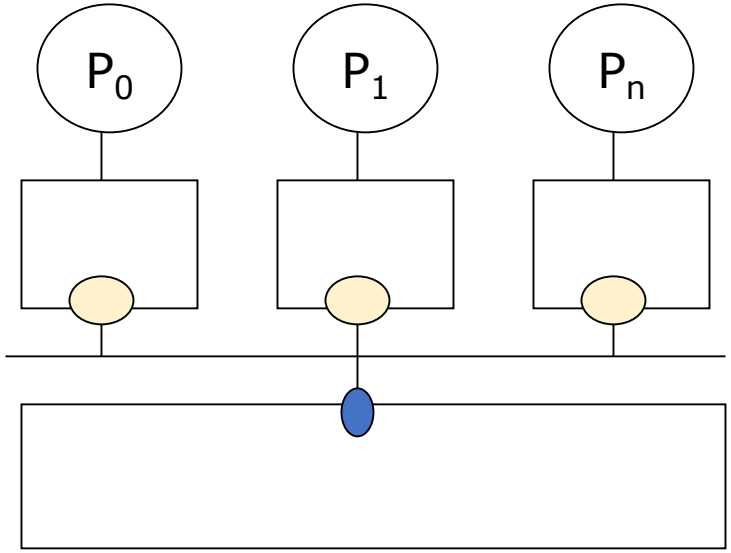
Write-back + Write allocate



Coherence with Write-through Caches

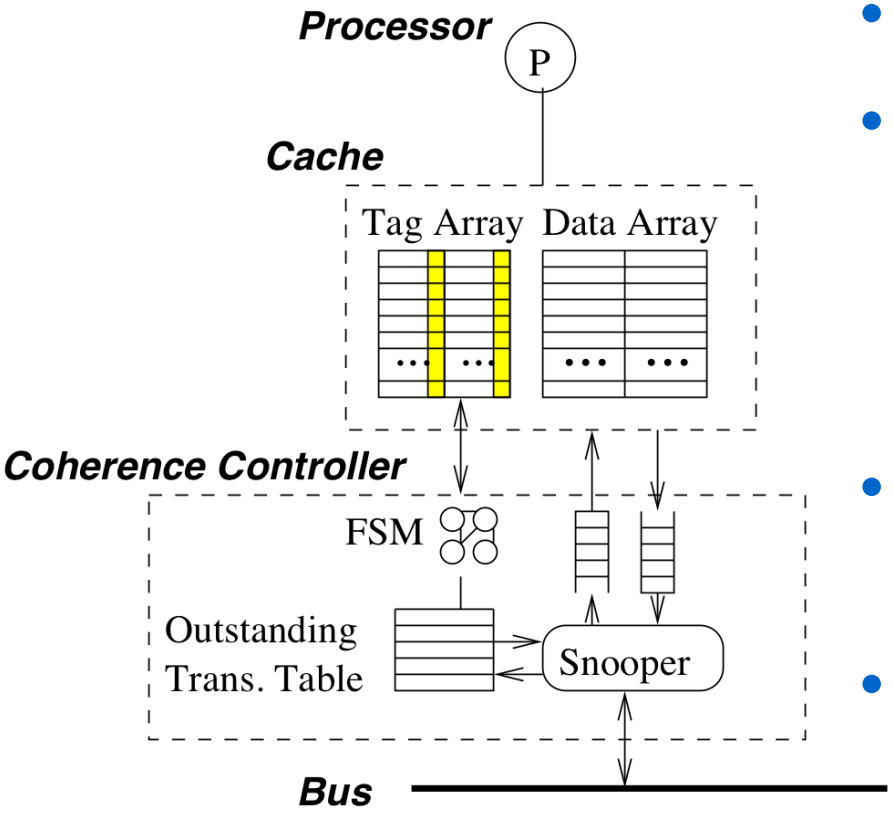
```
sum = 0;
begin parallel
for (i=0; i<2; i++) {
  lock(id, myLock);
  sum = sum + a[i];
  unlock(id, myLock);
end parallel
Print sum;

Suppose a[0] = 3 and a[1] = 7
```

= snooper



– Bus-based SMP implementation choice in the mid 80s

Snooper Architecture

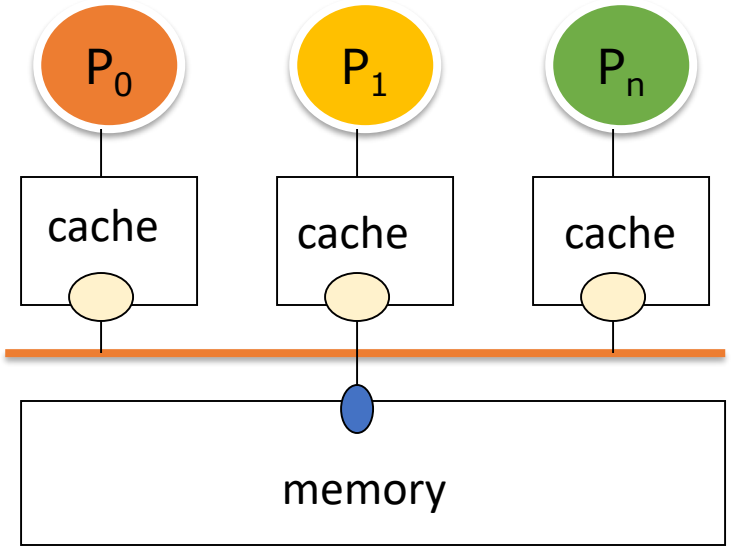


- Coherence controller is added
- Snooper snoops all bus transactions
  - If a bus transaction is relevant, an action is taken
- Finite State Machine (FSM) determines the new state
- Outstanding transaction table keeps transactions that are pending (not completed yet)

Coherence Protocol for Write-Through Caches

Coherence Protocol for Write-Through Caches

- **The simplest cache coherence protocol**
- **Assume a single level cache**
  - which may get requests from the processor side, as well as from the bus side as snooped by the snooper



Coherence Protocol for Write-Through Caches

- **Processor requests to the cache include:**
  - **`PrRd:`** processor-side request to read a cache block
  - **`PrWr:`** processor-side request to write a cache block

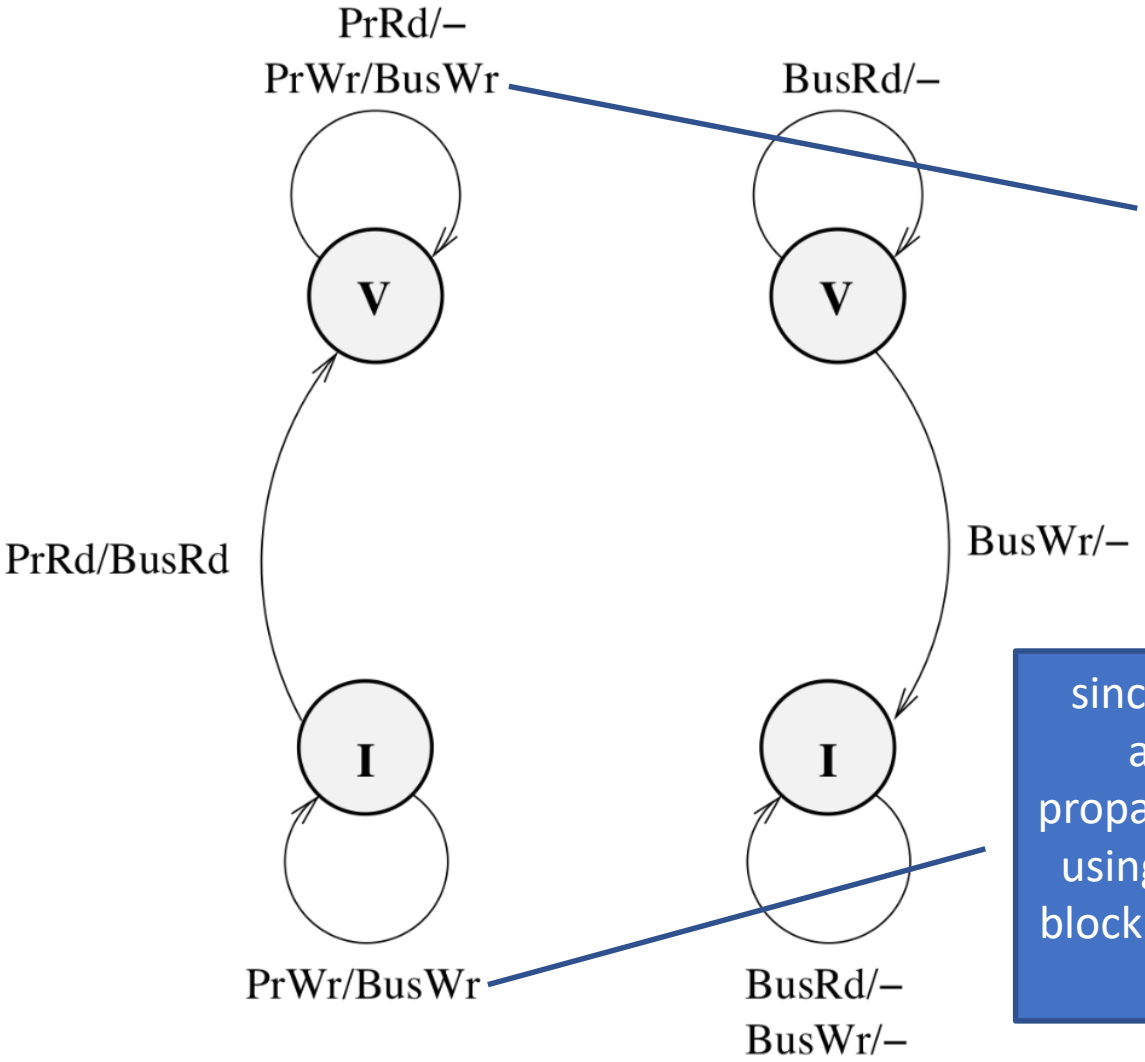
- **Snooped requests to the cache include:**
  - **`BusRd:`** snooped request that indicates there is a read request to a block made by another processor.
  - **`BusWr:`** snooped request that indicates there is a write request to a block made by another processor.

Coherence Protocol for Write-Through Caches

- **Cache States**
  - **`Valid (V):`** the cache block is **valid and clean**
    - the cached value is the same with that in the lower level memory component
  - **`Invalid (I):`** the cache block is **invalid**
    - Accesses to this cache block will generate cache misses
  - There is **no dirty state** in the write-through cache
    - since all writes are written through to the lower level.

Assume that the caches use:

- Write no-allocate policy
- Write invalidate cache coherence policy



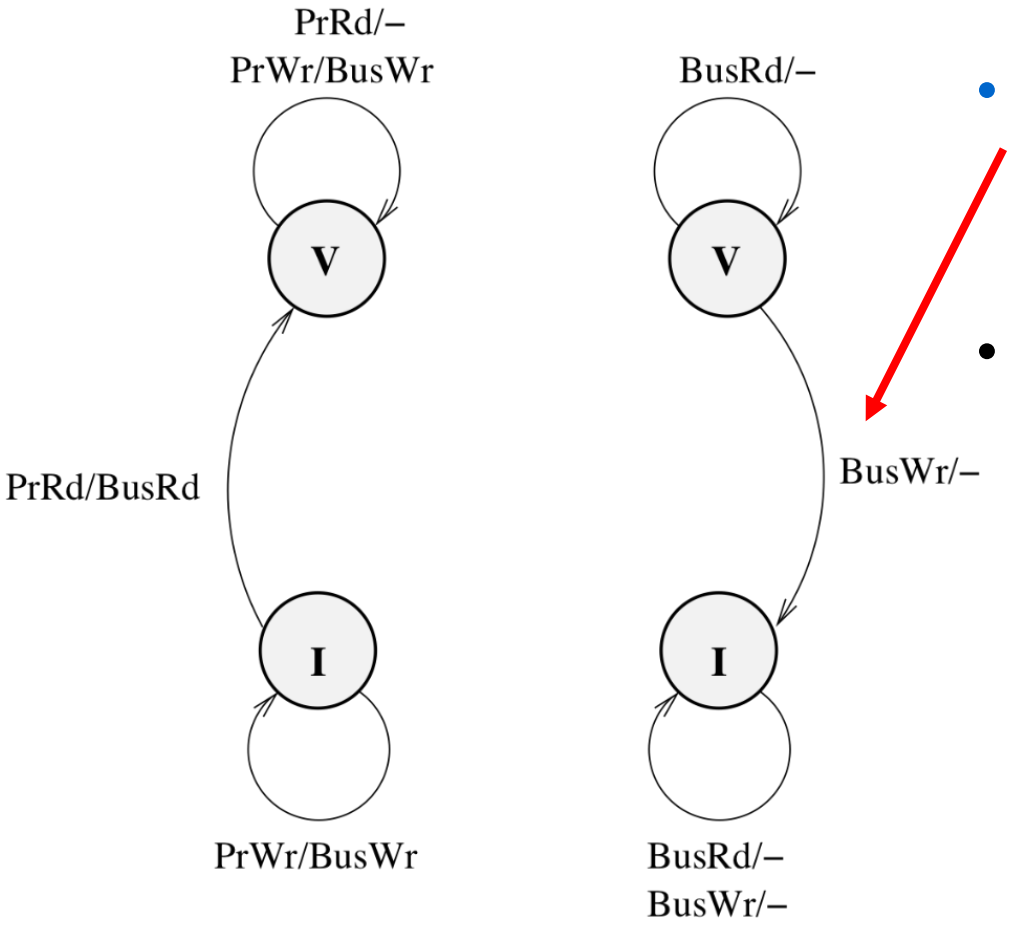
Notation:
Triggering event / action (or response taken as a result of the event).

since the cache uses a write no-allocate policy, the write is propagated down to the lower level using BusWr without fetching the block into the cache. Therefore, the state remains invalid.

Response to processor-side requests

response to snooper-side requests

State Transition Diagram



- **Key points**: write invalidates all other caches
- Therefore, we have:
  - **Modified line**: exists as Valid in only 1 cache
  - **Clean line**: exists as Valid in at least 1 cache
  - **Invalid state** represents invalidated line or not present in the cache

Problem with Write-Through Caches

- **High bandwidth requirement**
  - due to each write causing a write propagation

- **What if we use write-back caches?**
  - Write hits stop at the cache → eliminate most bus write transactions

MSI Protocol with Write-back Caches

MSI Writeback Invalidate Protocol

- **States**
  - **`Invalid (I)`**
  - **`Shared (S)`**: one or more copies, and memory copy is up-to-date
    - the cache block is valid, potentially shared by multiple processors
  - **`Modified (M)`**: only one copy
    - the cache block is valid in only one cache
    - the value is (likely) different from the one in the main memory

Modified vs. Dirty

- **Modified state extends dirty state in a write-back cache**
  - Modified state implies exclusive ownership

- **Dirty:**
  - (1) cached value is different from the value in main memory
- **Modified:**
  - (1) cached value is different from the value in main memory
  - (2) cached only in one location

MSI Writeback Invalidate Protocol

- **Processor-side Requests:**
  - **`PrRd`** (read)
    - processor-side request to read a cache block
  - **`PrWr`** (write)
    - processor-side request to write to a cache block

MSI Writeback Invalidate Protocol

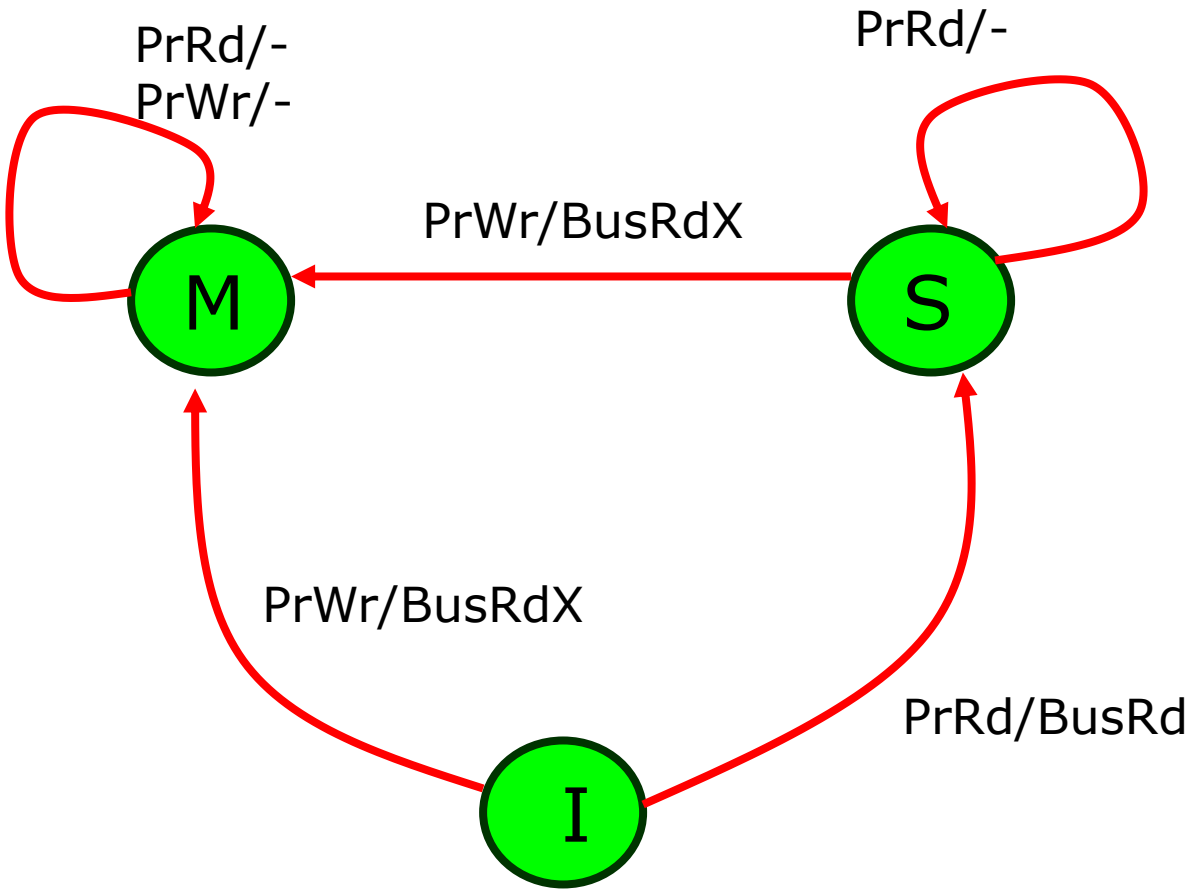
- **Bus-side Requests:**
  - **BusRd**: snooped request that indicates there is a *read request* to a cache block made by another processor
  - **BusRdX**: snooped request that indicates there is a *read exclusive (**write**) request* to a cache block made by another processor (instead of BusWr)
  - **Flush**: snooped request that indicates that an entire cache block is written back to the main memory by another processor

Basic MSI Writeback Invalidate Protocol

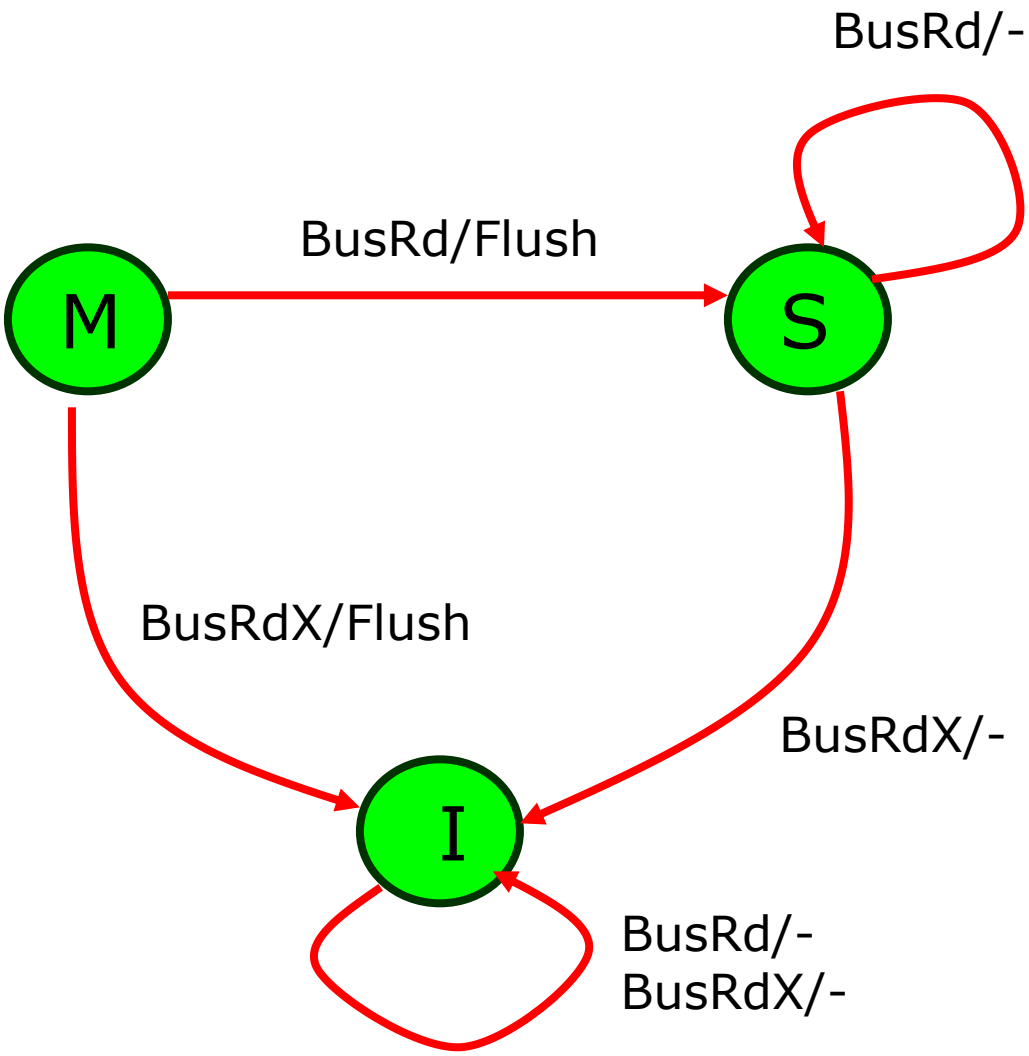
- **Actions**
  - Update state
  - perform bus transaction
  - flush value onto bus

- **Invalidation: Any → Invalid**
- **Intervention: Modified → Shared**

Processor Initiated Transactions

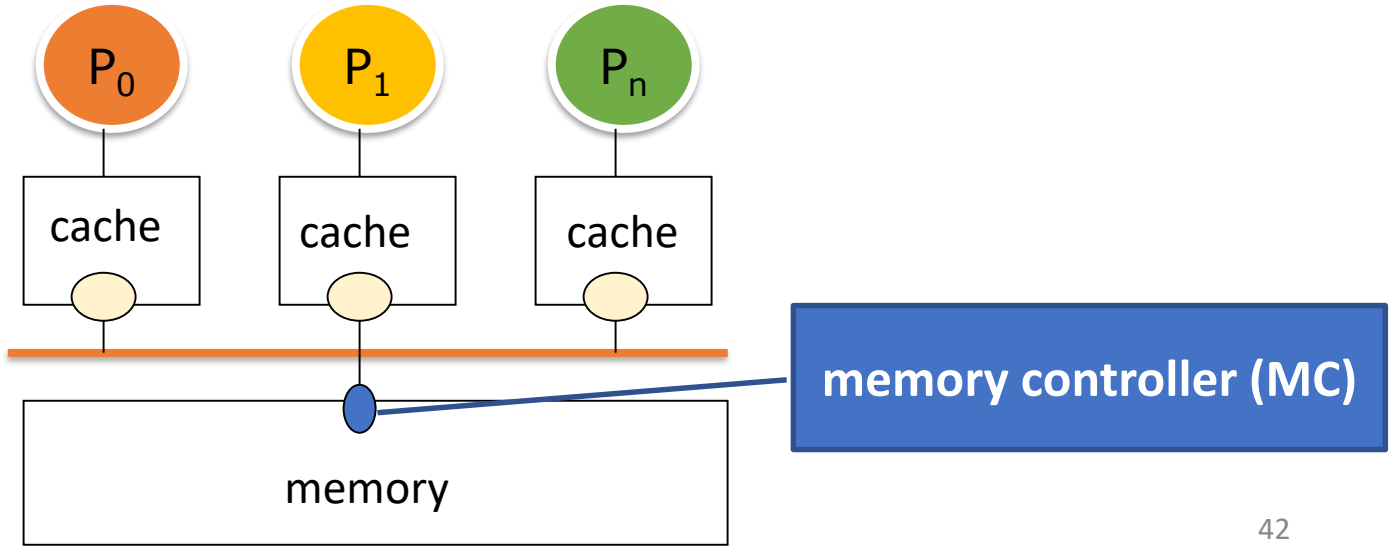


Bus Initiated Transactions



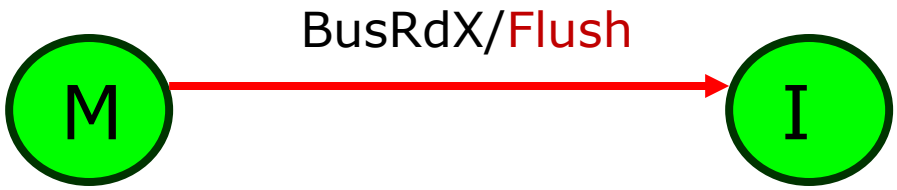
Question 1

- **What are the actions at memory controller (MC)?**
  - Snooped a BusRd: read and supply data
  - Snooped a BusRdX: read and supply data
  - Snooped a Flush: update main memory



Question 2

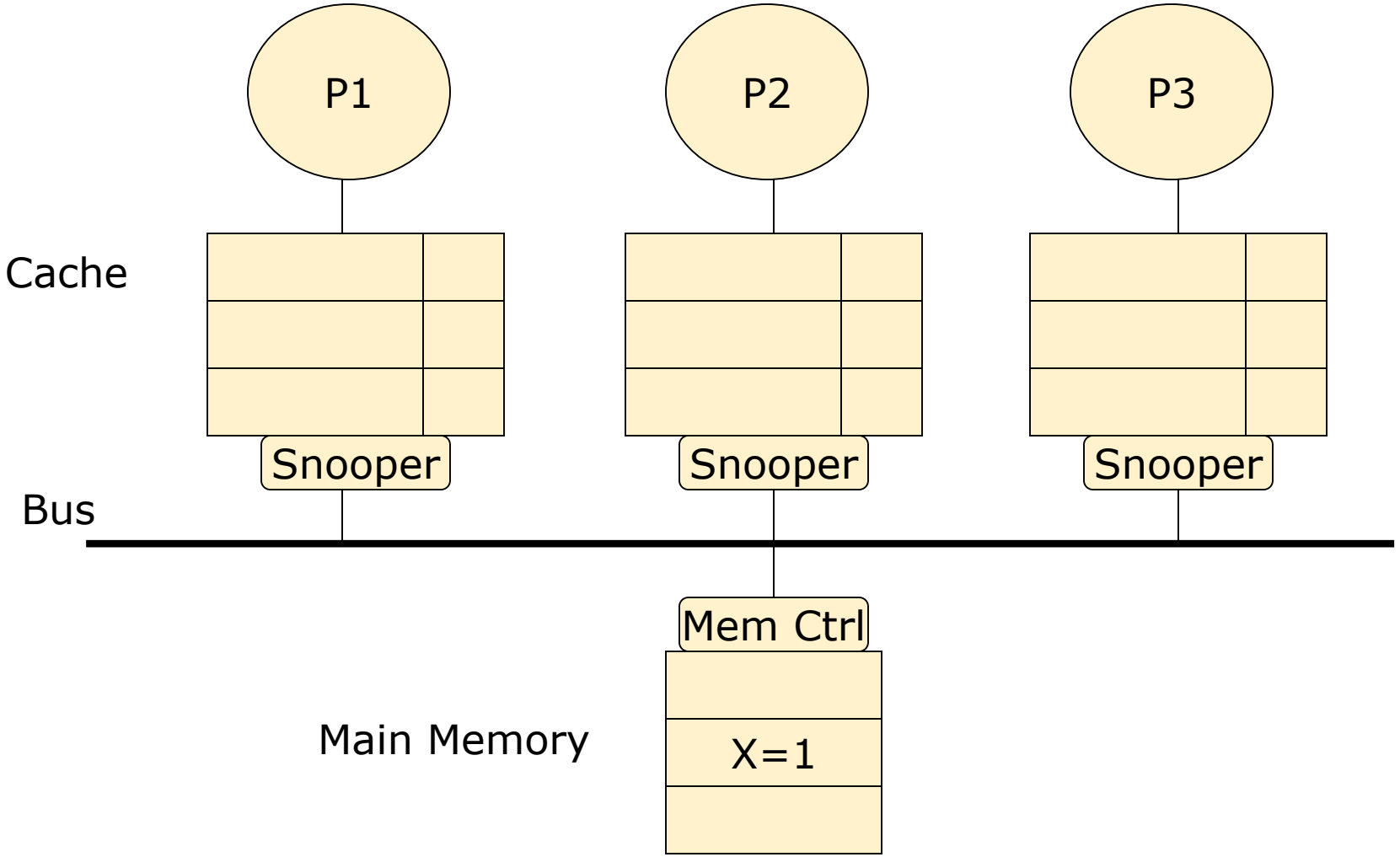
- **Q: Upon snooping BusRdX, if a cache line's state is M, why do we need to Flush the cache line?**
  - This flush may sound redundant
  - since the processor that issues a BusRdX is going to overwrite the cache block



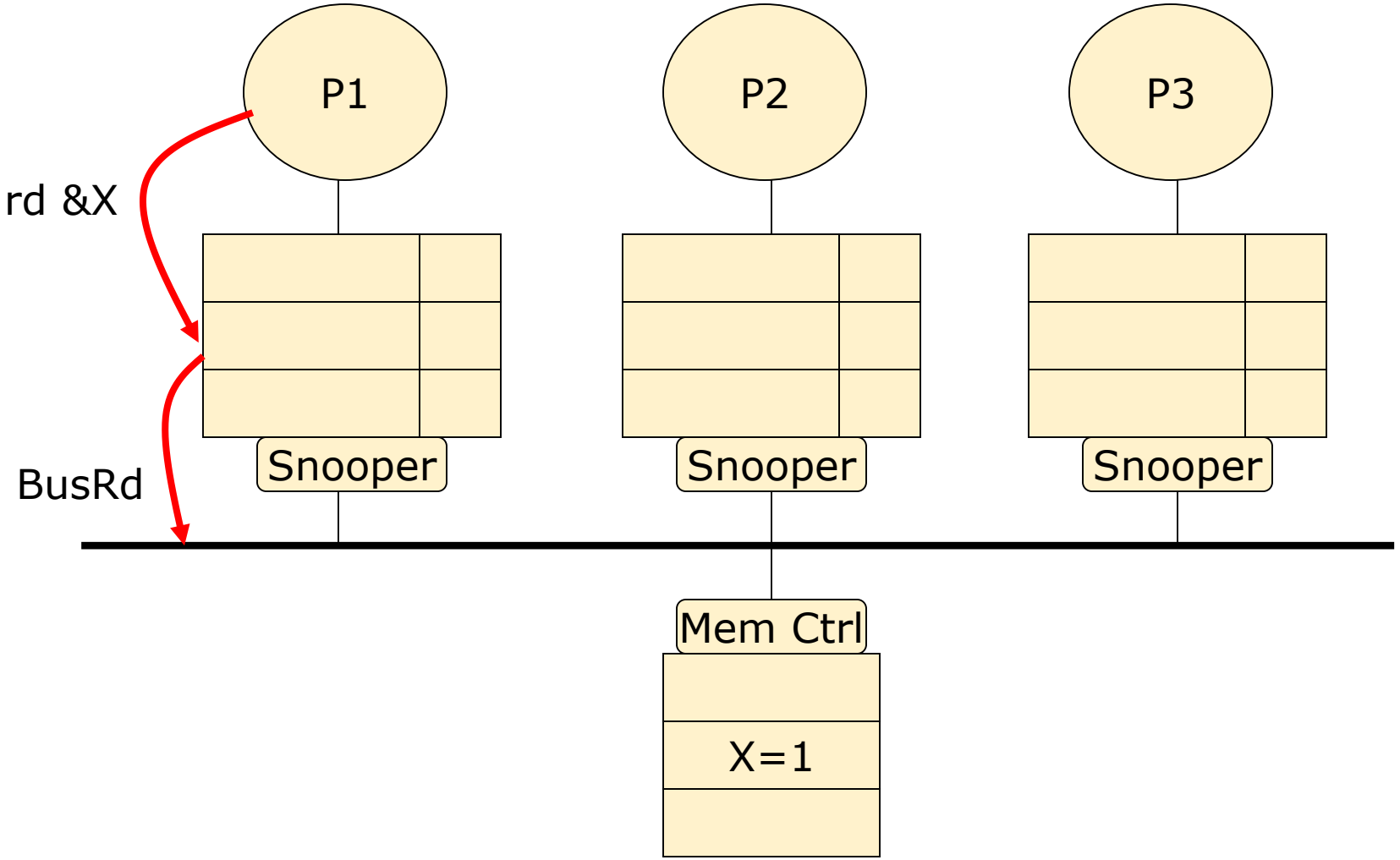
Question 2

- **Answer:**
  - a typical cache block is large, containing multiple bytes
  - The bytes that have been modified are not necessarily the same bytes that another processor wants to overwrite
  - Another processor may want to write to other bytes, but read from the bytes that have been modified.
- **Flushing the entire block is a correctness requirement**
  - The processor that issues the BusRdX must pick the block up and place it in the cache prior to writing to it.

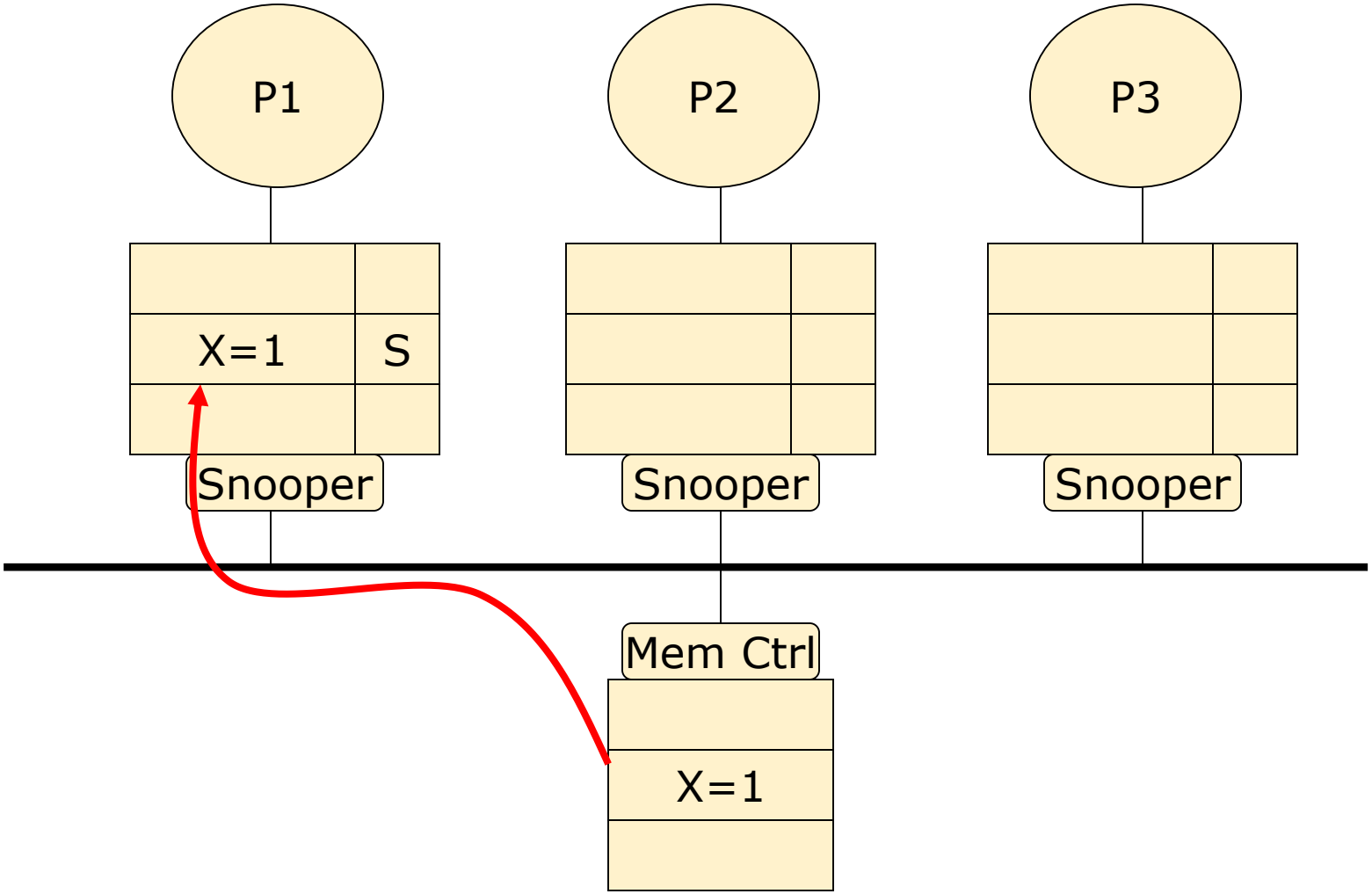
MSI Visualization



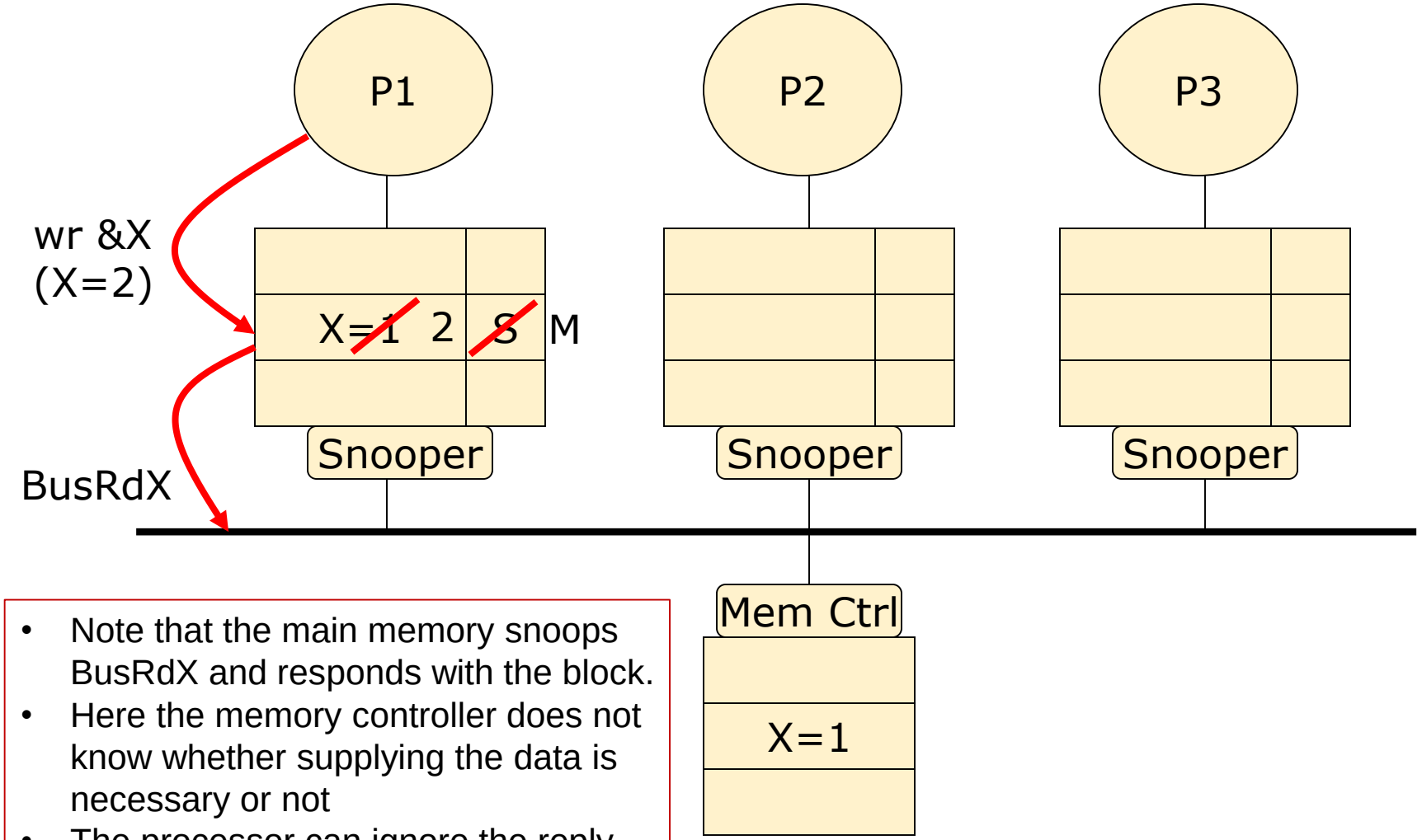
MSI Visualization



Proc Request	State P1	State P2	State P3	Bus Request	Data Supplier
R1	S	-	-	BusRd	Mem

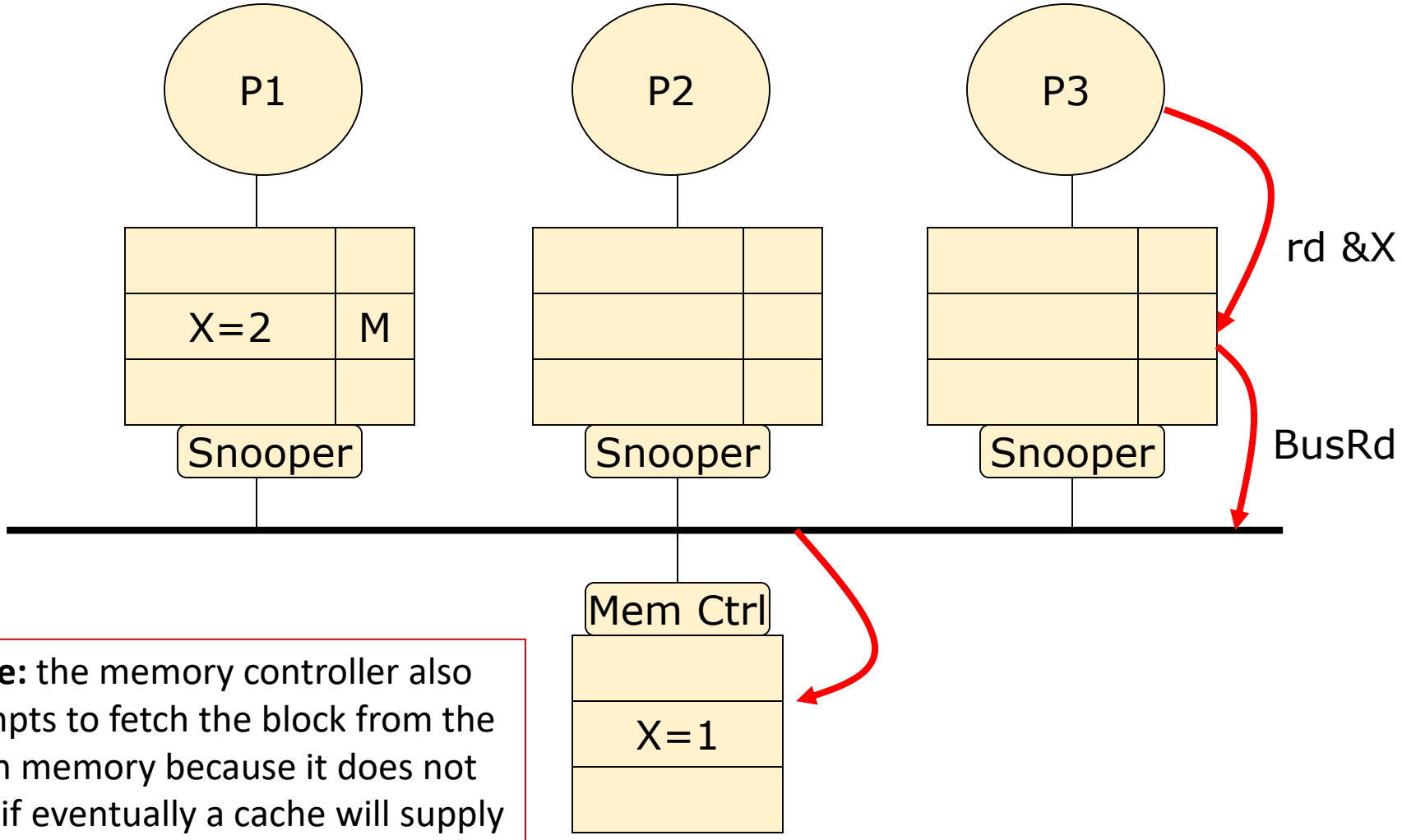


Proc Request	State P1	State P2	State P3	Bus Request	Data Supplier
W1	M	-	-	BusRdX	**Mem**



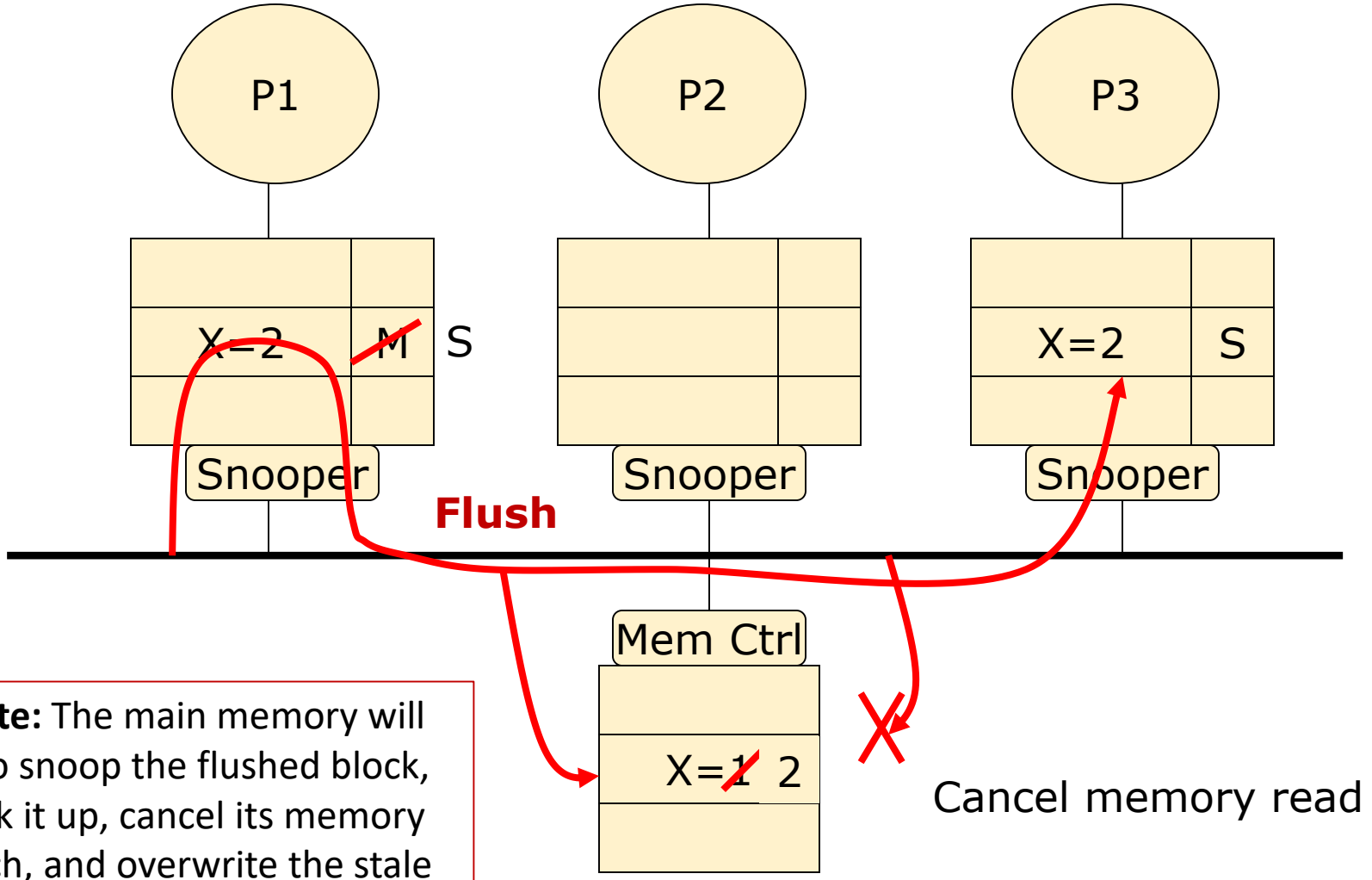
- Note that the main memory snoops BusRdX and responds with the block.
- Here the memory controller does not know whether supplying the data is necessary or not
- The processor can ignore the reply from the memory controller.

MSI Visualization



Note: the memory controller also attempts to fetch the block from the main memory because it does not know if eventually a cache will supply the data or not.

Proc Request	State P1	State P2	State P3	Bus Request	Data Supplier
R3	S	-	S	BusRd	P1 cache

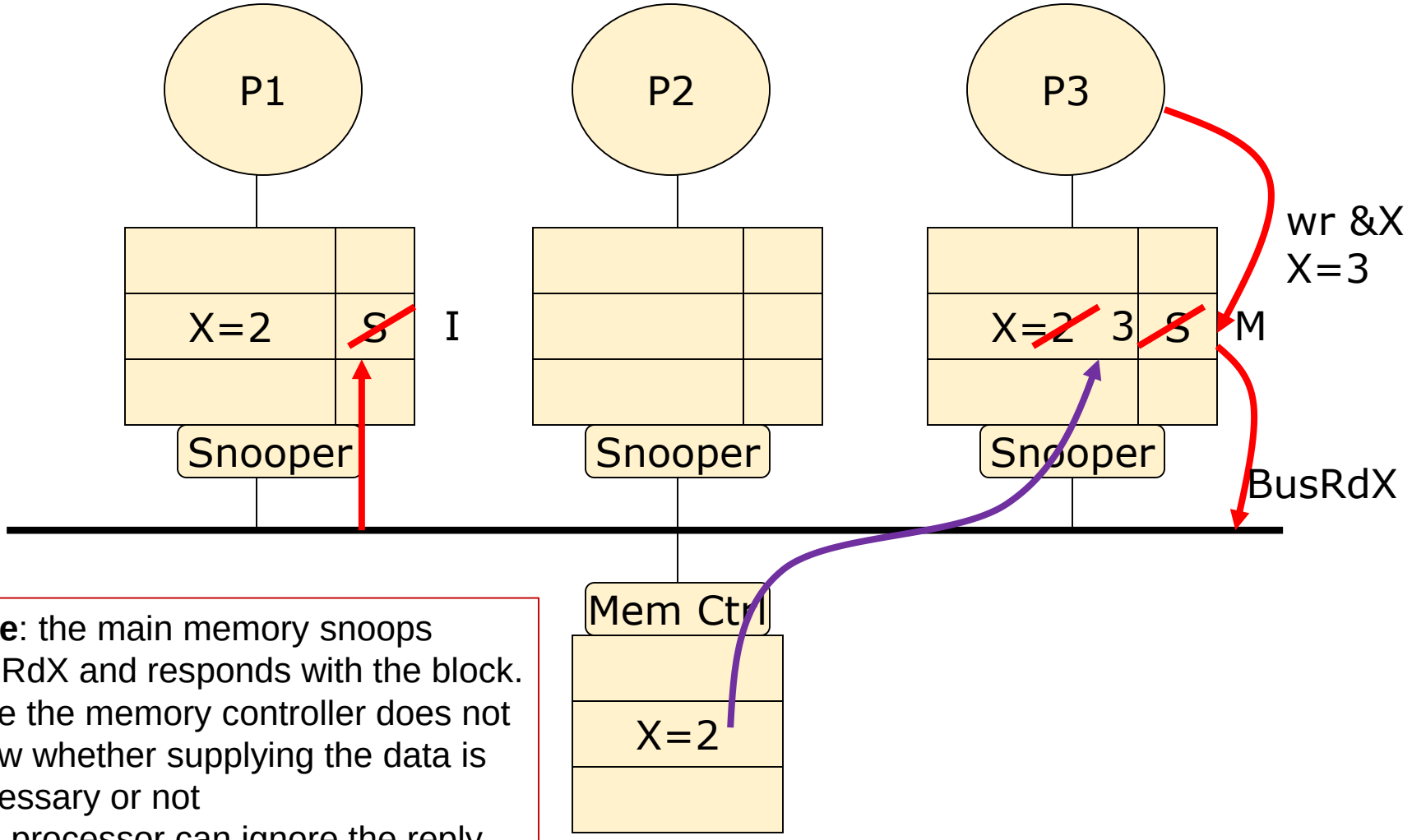


Note: The main memory will also snoop the flushed block, pick it up, cancel its memory fetch, and overwrite the stale copy of the block in memory.

Question 3

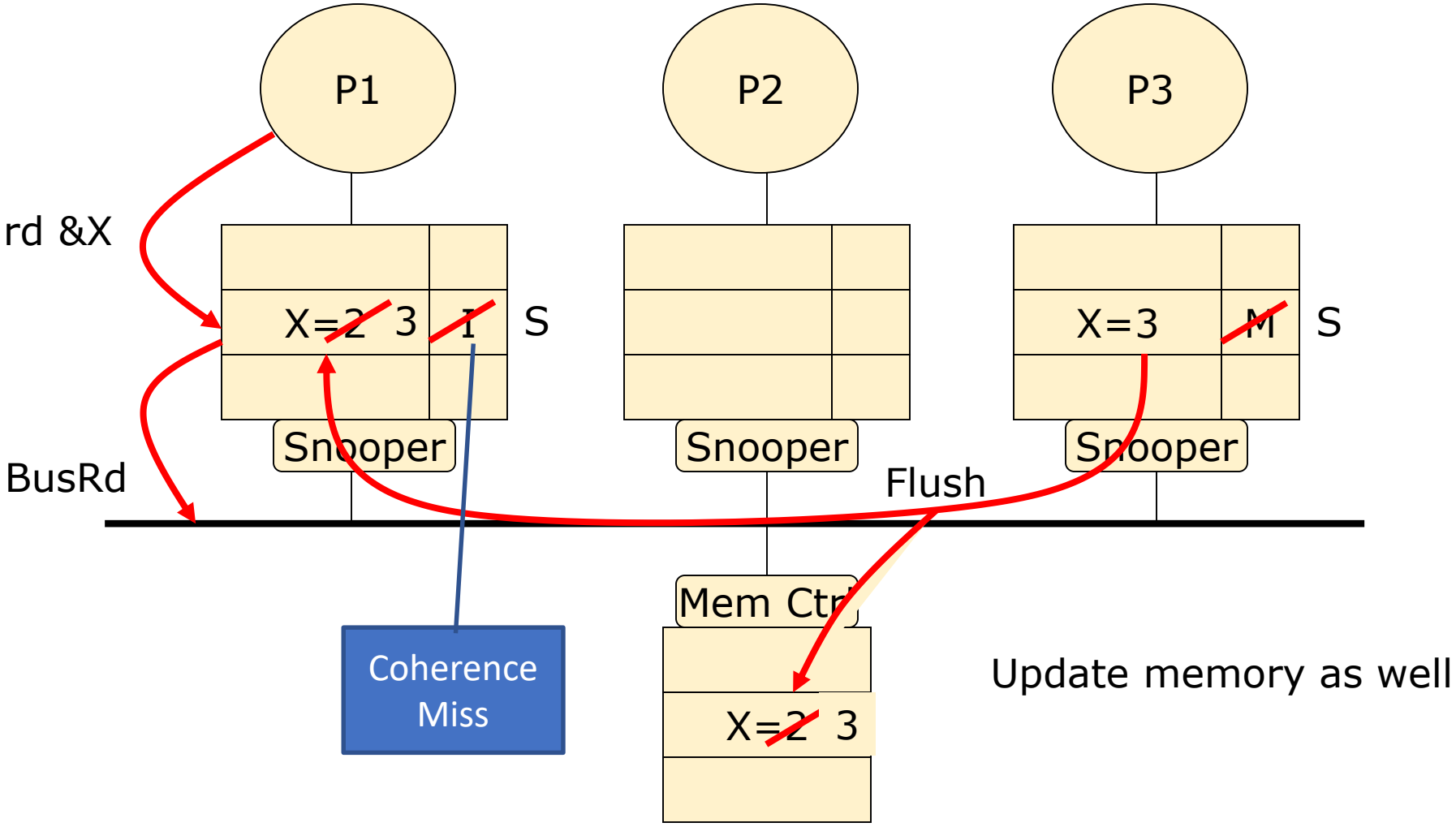
- **Is there a correctness problem if the main memory has supplied a stale copy before the owner has a chance to flush its block?**
  - **Answer**: this cannot be allowed to occur.
  - **Solution**: to give enough time for all processors to finish snooping and responding to a bus request, before the memory controller replies with a block.

Proc Request	State P1	State P2	State P3	Bus Request	Data Supplier
W3	I	-	M	BusRdX	**Mem**

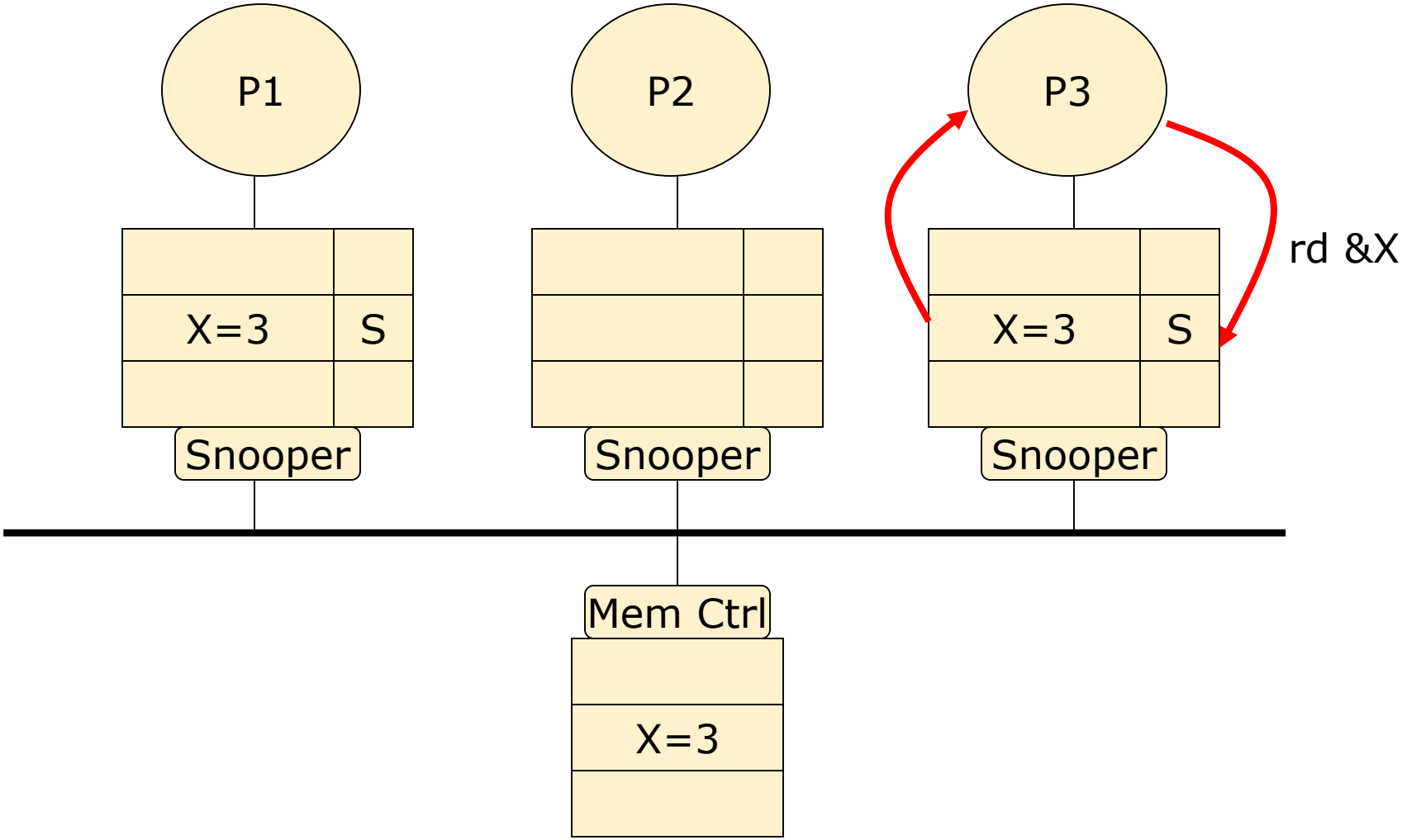


- **Note**: the main memory snoops BusRdX and responds with the block.
- Here the memory controller does not know whether supplying the data is necessary or not
- The processor can ignore the reply from the memory controller.

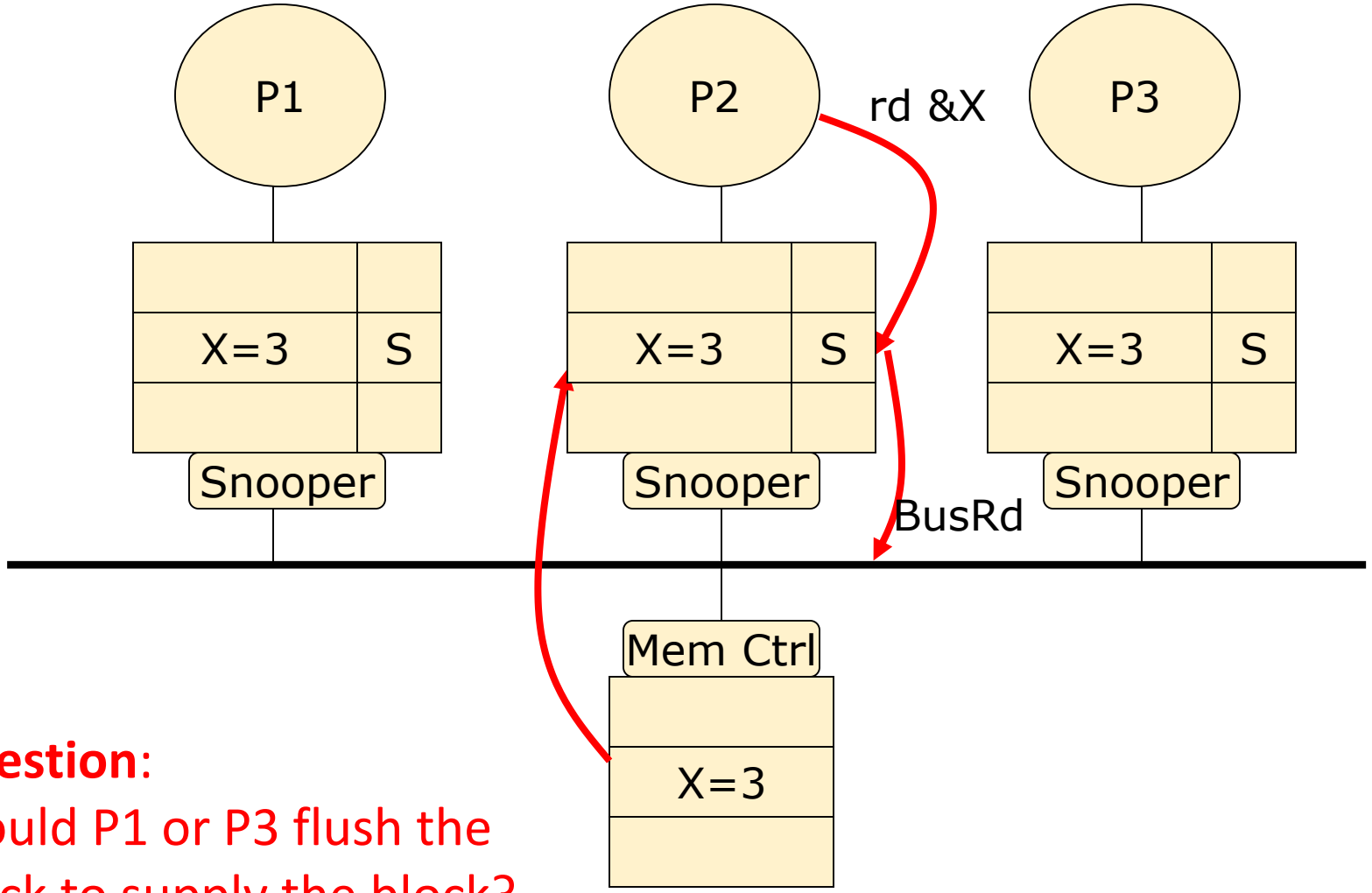
Proc Request	State P1	State P2	State P3	Bus Request	Data Supplier
R1	S	-	S	BusRd	P3 cache



Proc Request	State P1	State P2	State P3	Bus Request	Data Supplier
R3	S	-	S	-	-



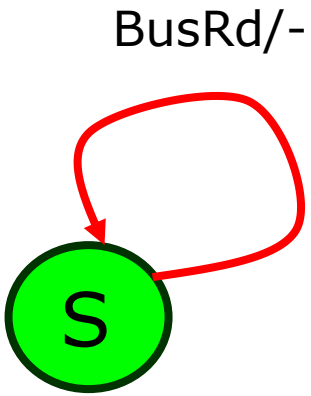
Proc Request	State P1	State P2	State P3	Bus Request	Data Supplier
R2	S	S	S	BusRd	Mem



Question:
should P1 or P3 flush the block to supply the block?

Discussion

- **Although not required for correctness, P1 or P3 can definitely do that**
  - In fact, in MESI protocol, cache-to-cache transfer is employed



Summary of State Changes

Proc Request	State P1	State P2	State P3	Bus Request	Data Supplier
R1	S	-	-	BusRd	Mem
W1	M	-	-	BusRdX	Mem
R3	S	-	S	BusRd	P1 cache
W3	I	-	M	BusRdX	Mem
R1	S	-	S	BusRd	P3 cache
R3	S	-	S	-	-
R2	S	S	S	BusRd	Mem

MESI Protocol with Write-back Caches

Problem with MSI protocol

- **Each read-write sequence incurs two bus transactions**
  - BusRd + BusRdX
  - regardless of whether the block is stored in **only one cache or not.**

- **Penalizing programs that have little data sharing is unacceptable!**
  - Sequential program (no data sharing)
  - Highly optimized parallel program (little data sharing)

MESI (4-state) Invalidation Protocol

- **MESI adds an *exclusive* state**
  - to distinguish between shared by multiple caches vs. shared by one cache (exclusively cached)
- **Four states:**
  - Invalid
  - modified (dirty)
  - shared (two or more caches may have copies)
  - **Exclusive**: (only this cache has *clean* copy, same value as in memory)

Question

- **How can we check whether there are existing copies in other caches?**
  - To achieve that, a **new bus line** can be added.
  - Called "COPIES-EXIST" or "C" bus line
    - High value: when there is at least one cache that asserts (声明) it.
    - Low value: when there are no cached copies

MESI (4-state) Invalidation Protocol

- **Bus-side Requests:**
  - **BusRd**:
    - snooped request that indicates there is a *read request* to a cache block made by another processor
  - **BusRdX**:
    - snooped request that indicates there is a *read exclusive (write) request* to a cache block made by another processor which doesn't have the block
  - **BusUpgr**:
    - snooped request that indicates that there is a write request to a cache block that another processor which already has in its cache

BusRdx vs. BusUpgr

- **BusUpgr:**
  - If a cache already has a valid copy, and only needs to upgrade its value, it posts a BusUpgr.
  - The memory controller ignores the BusUpgr.
- **BusRdx:**
  - If a cache does not have the block in the cache and needs the memory (or another cache) to supply it, it posts a BusRdx.
  - The memory controller fetches the block when it snoops a BusRdx.

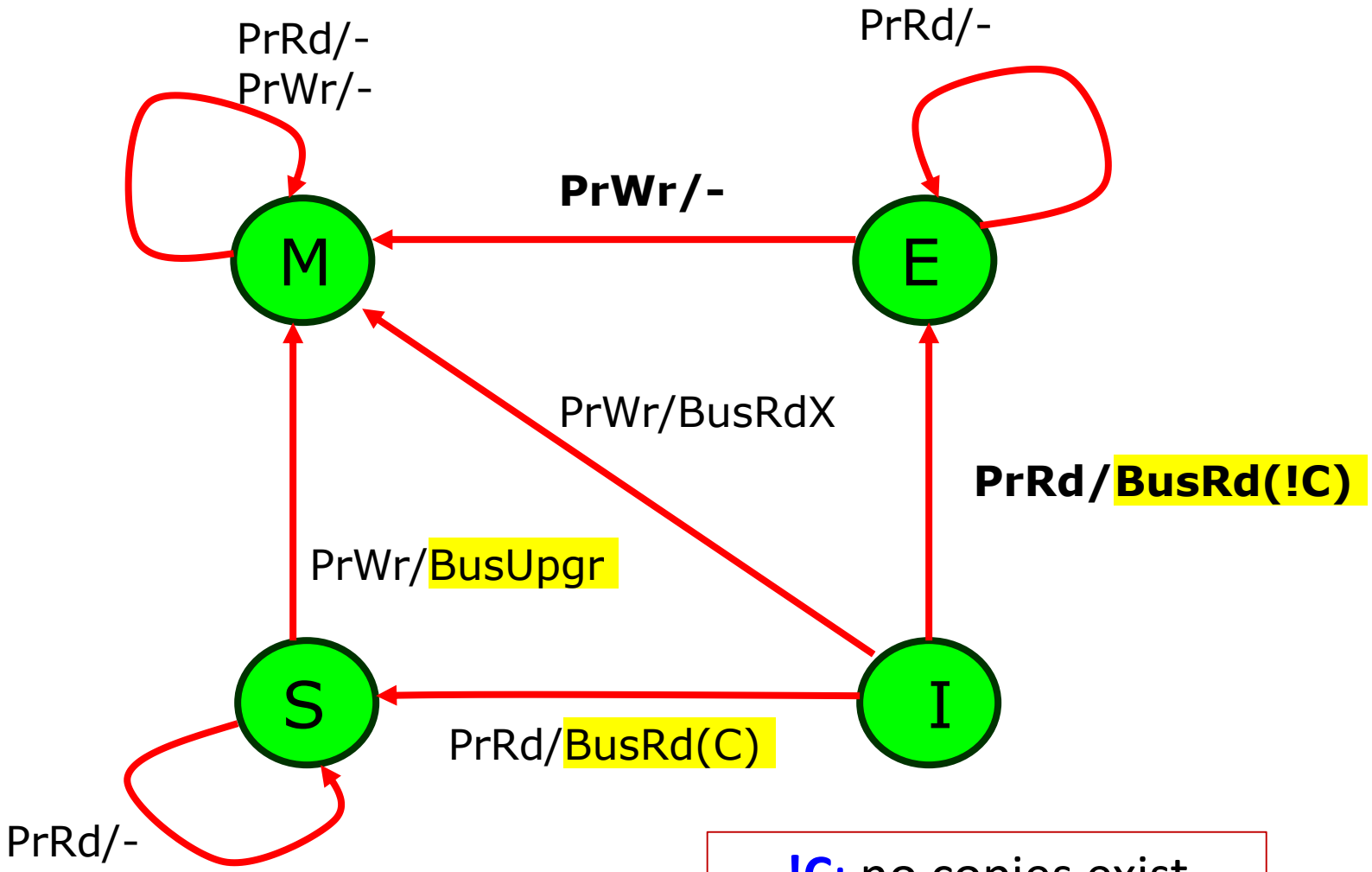
MESI (4-state) Invalidation Protocol

- **Bus-side Requests:**
  - **Flush**:
    - snooped request that indicates that an entire cache block is written back to the main memory by another processor
  - **FlushOpt**
    - Snooped request that indicates that an entire cache block is posted on the bus in order to supply it to another processor.

Flush vs. FlushOpt

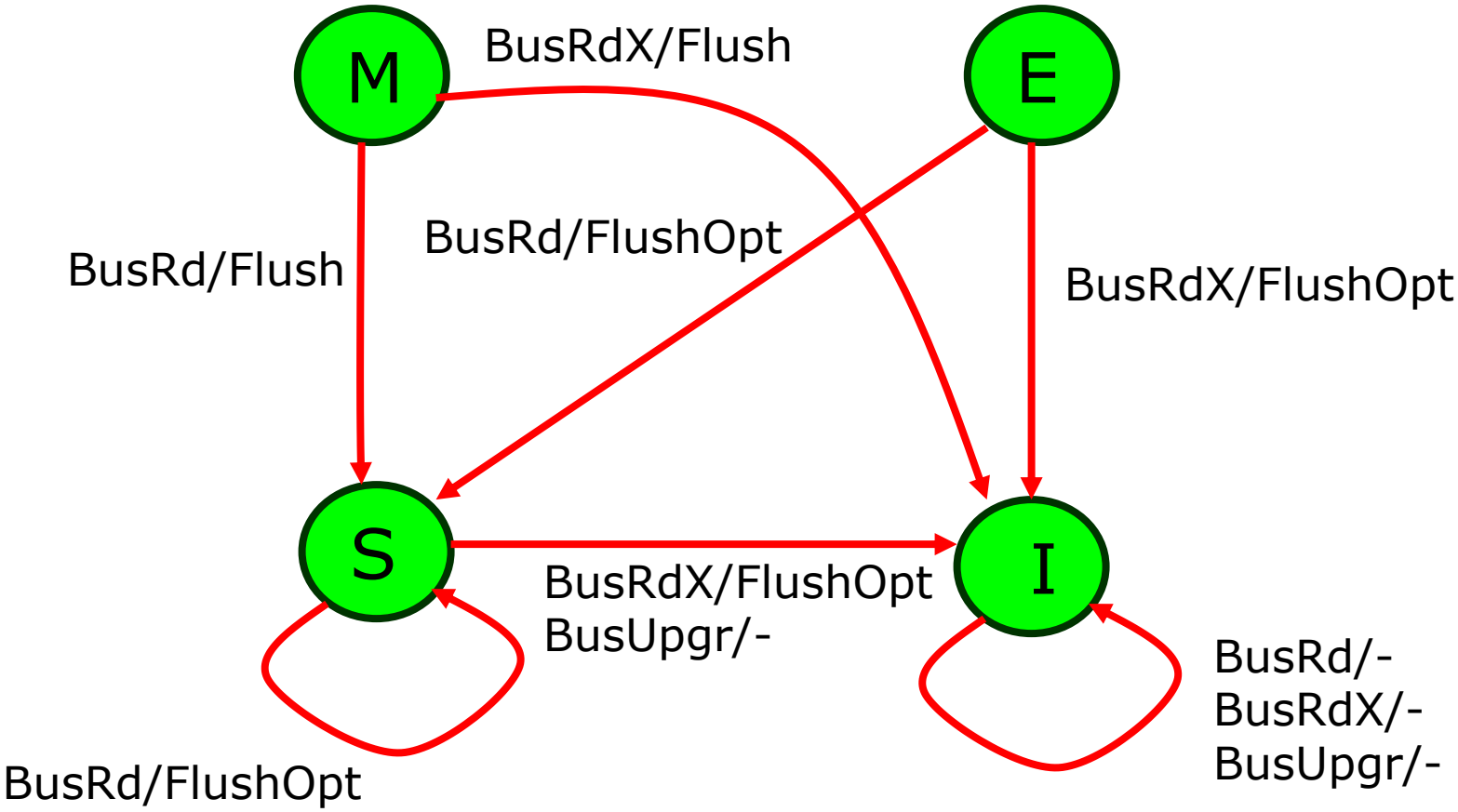
- **Flush: mandatory, needed for write propagation**
- **FlushOpt: optional, not required for correctness.**
  - Implemented as a performance enhancing feature that can be removed without impacting correctness
    - Based on a premise that obtaining data from another cache is faster than from the memory
  - Also referred to as "**cache-to-cache transfer**"

Processor Initiated Transactions



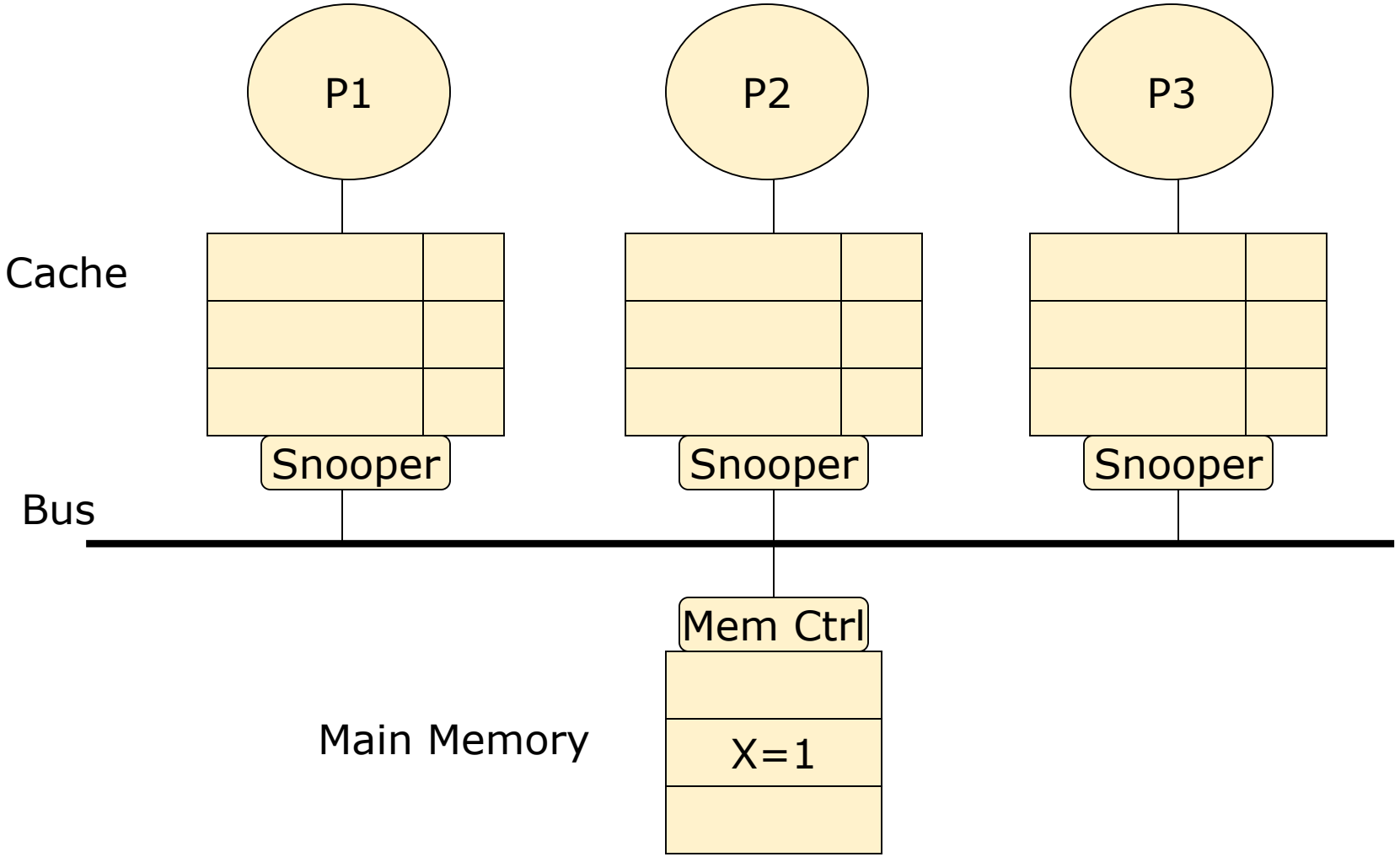
!C: no copies exist
C: other copies exist

Bus Initiated Transactions

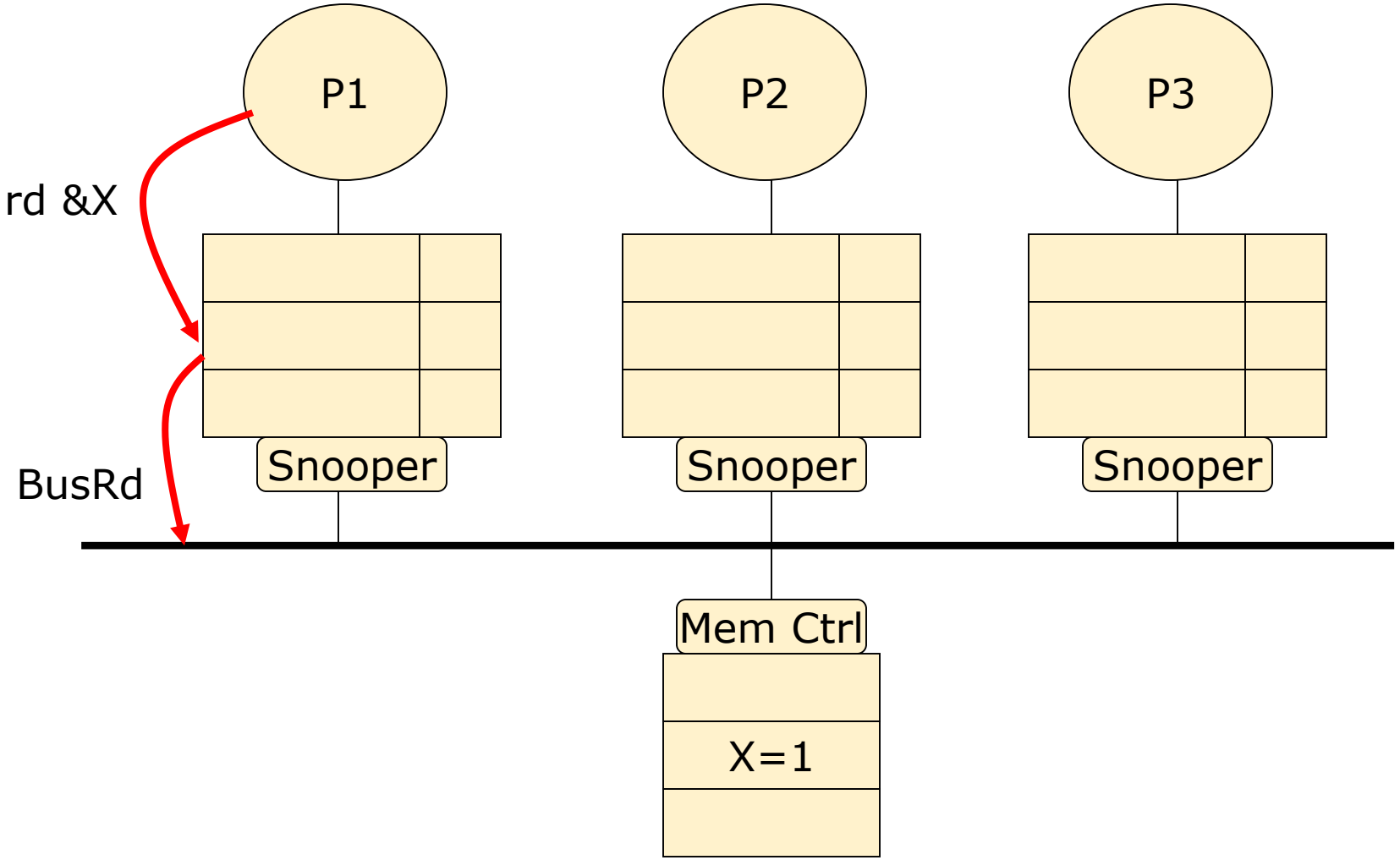


Note: FlushOpt = cache-to-cache transfer

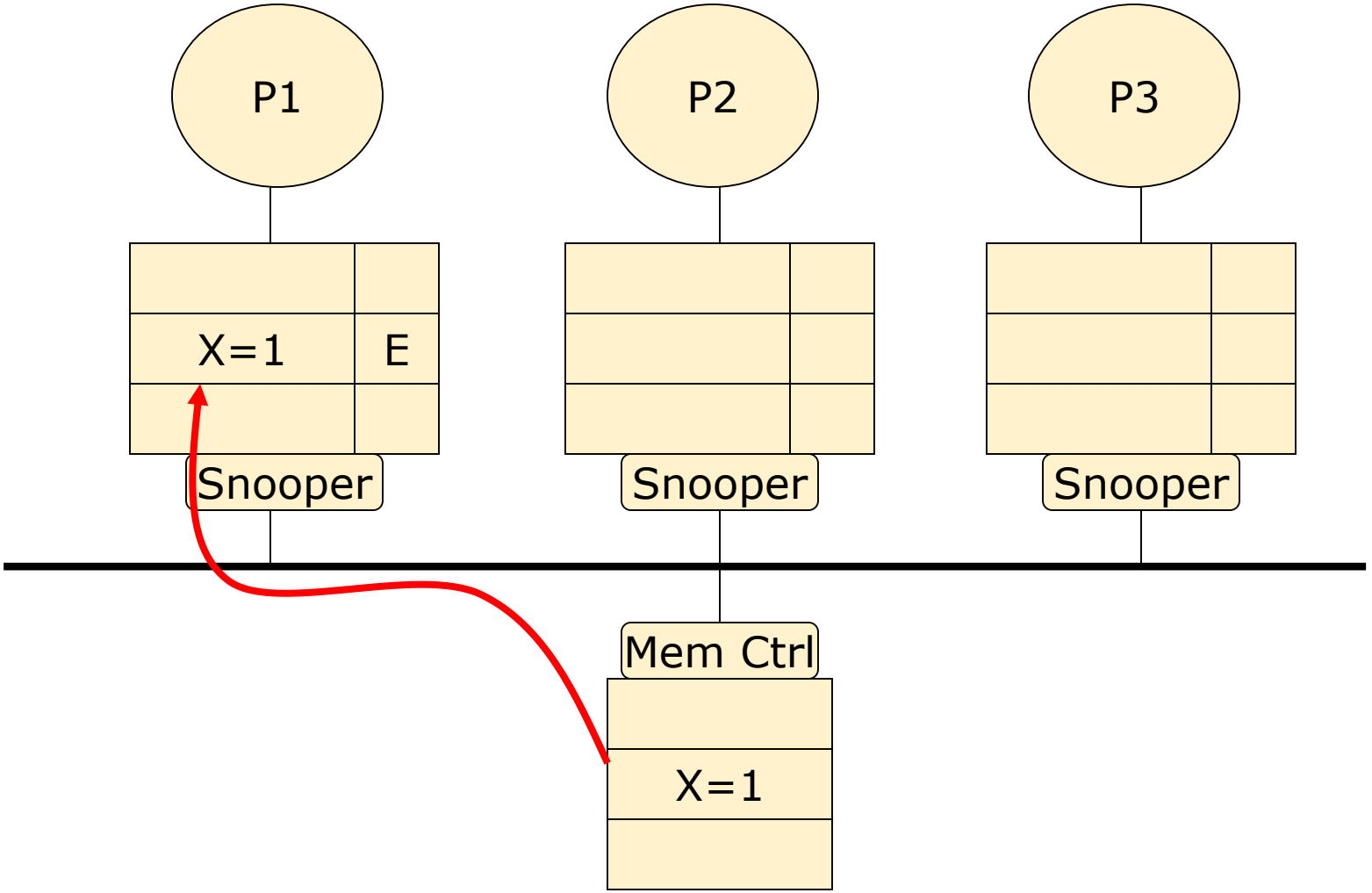
MESI Visualization



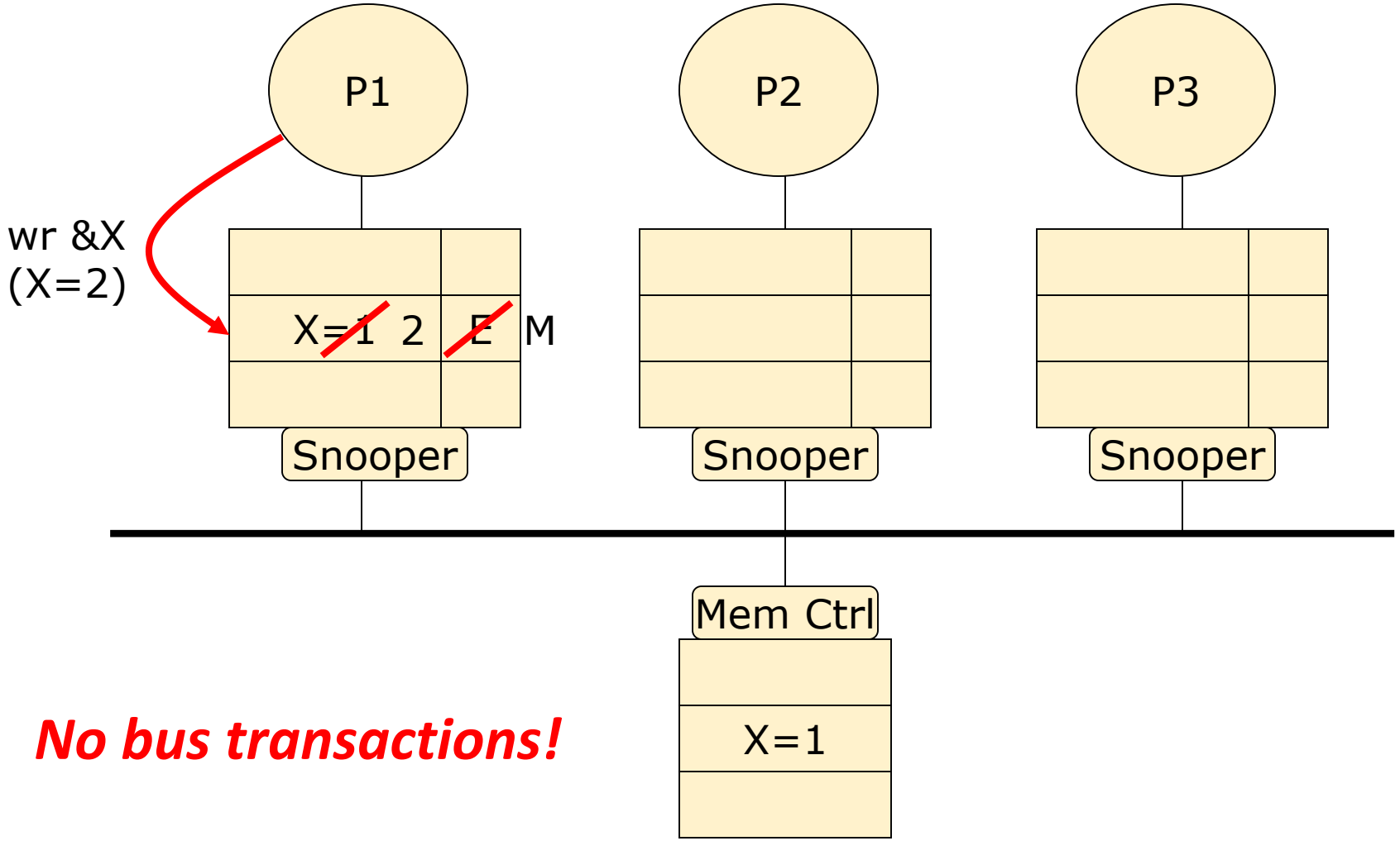
MESI Visualization



Proc Action	State P1	State P2	State P3	Bus Action	Data From
R1	**E**	-	-	BusRd	Mem

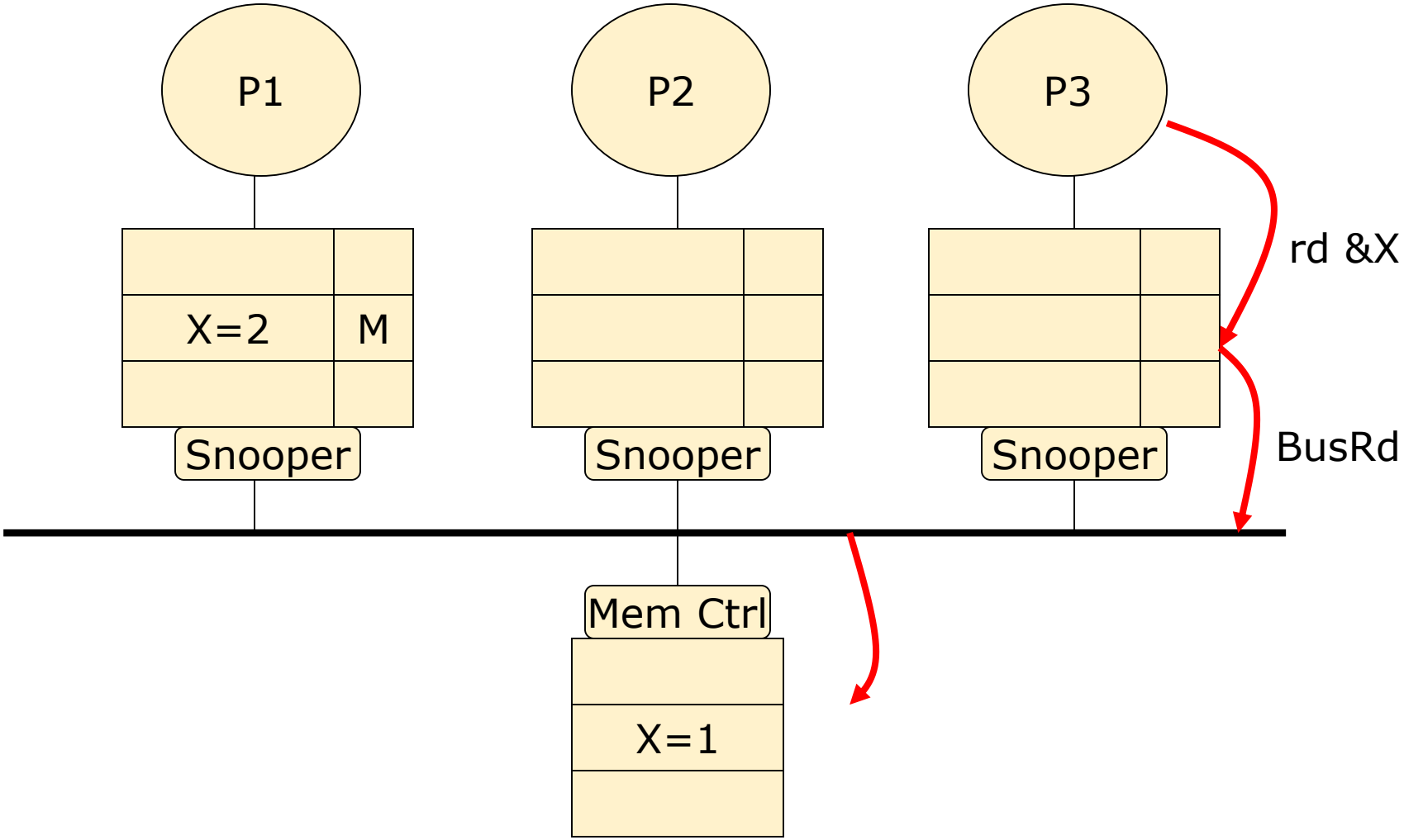


Proc Action	State P1	State P2	State P3	Bus Action	Data From
W1	M	-	-	-	-

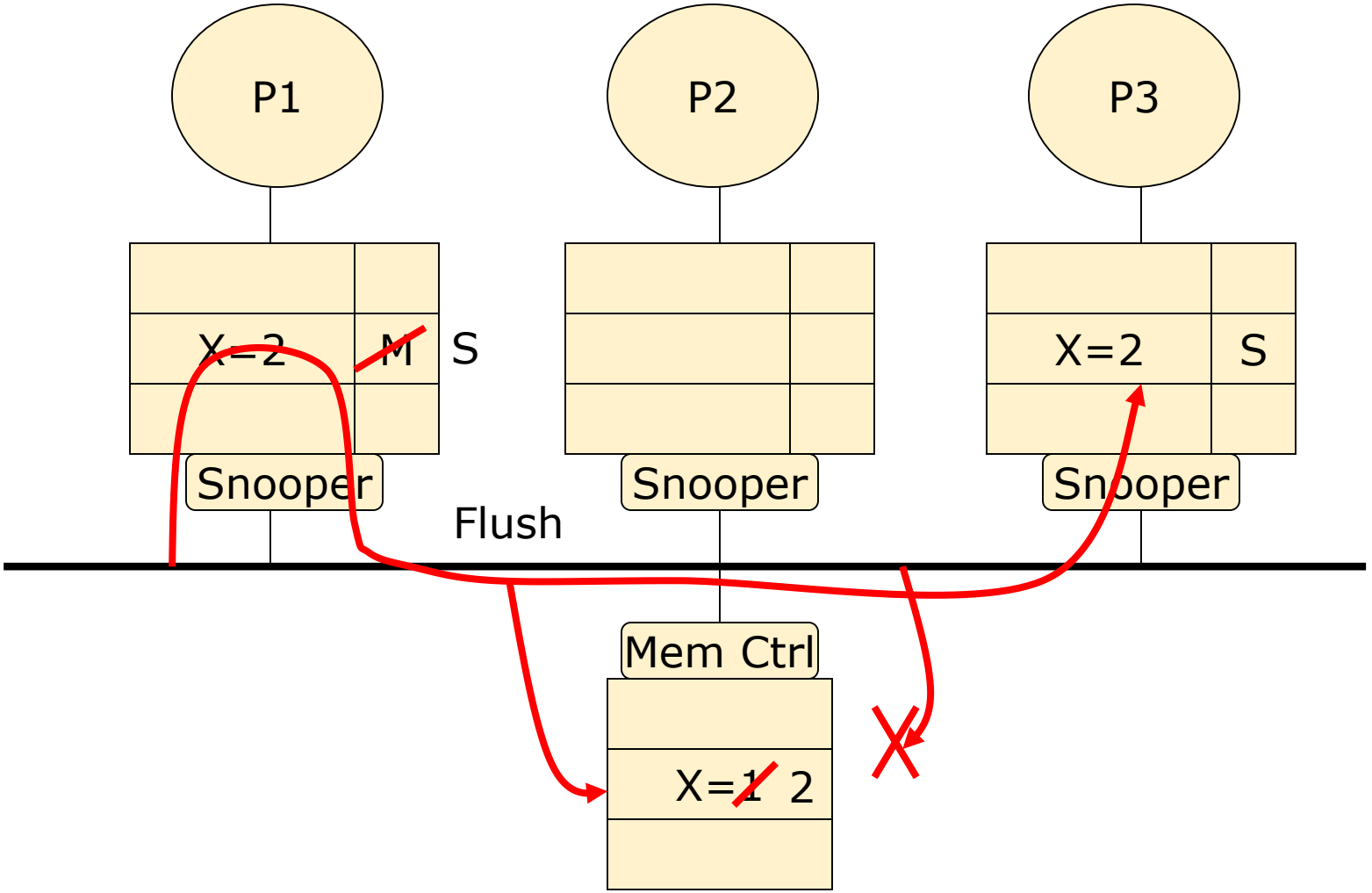


No bus transactions!

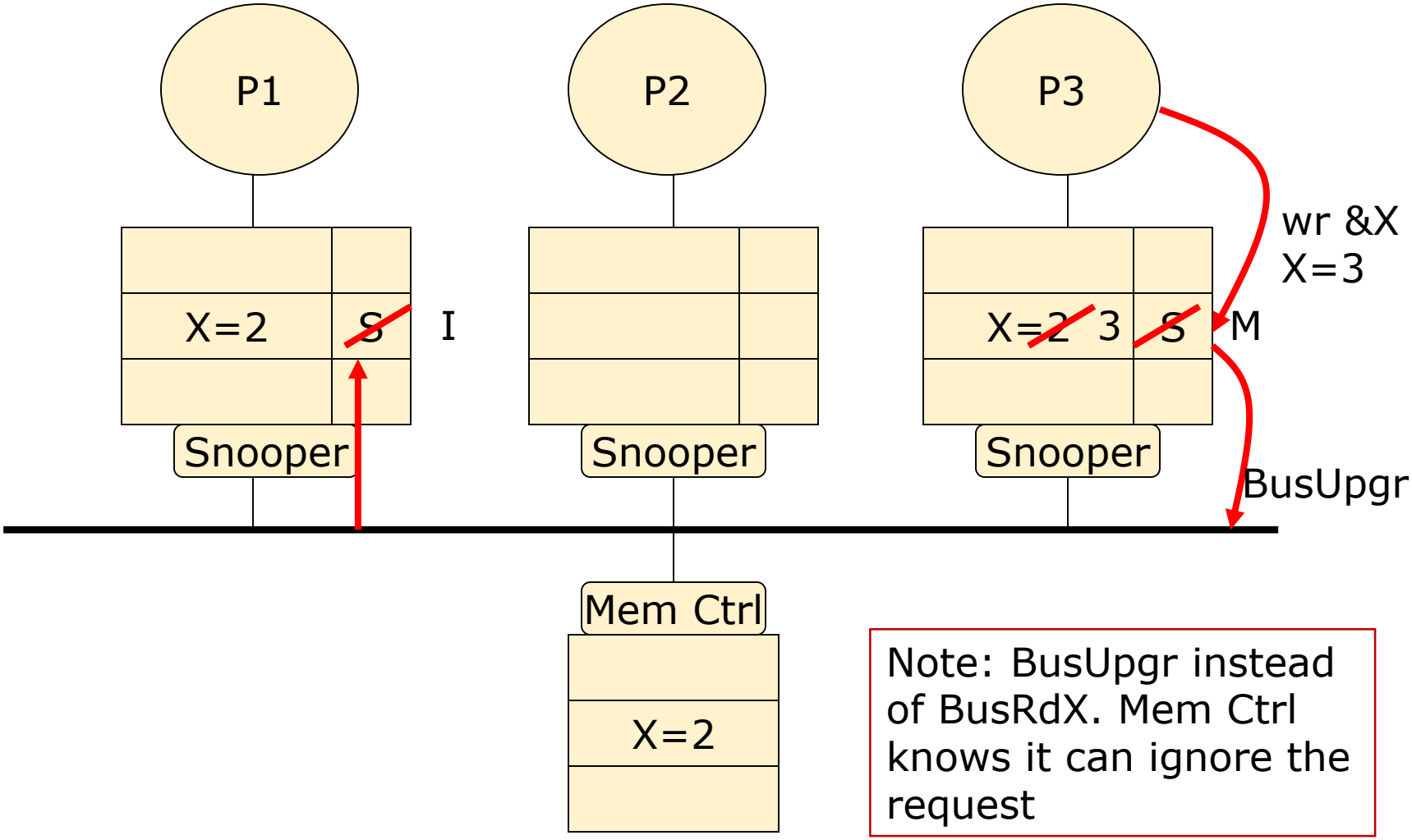
MESI Visualization



Proc Action	State P1	State P2	State P3	Bus Action	Data From
R3	S	-	S	BusRd/Flush	P1's cache

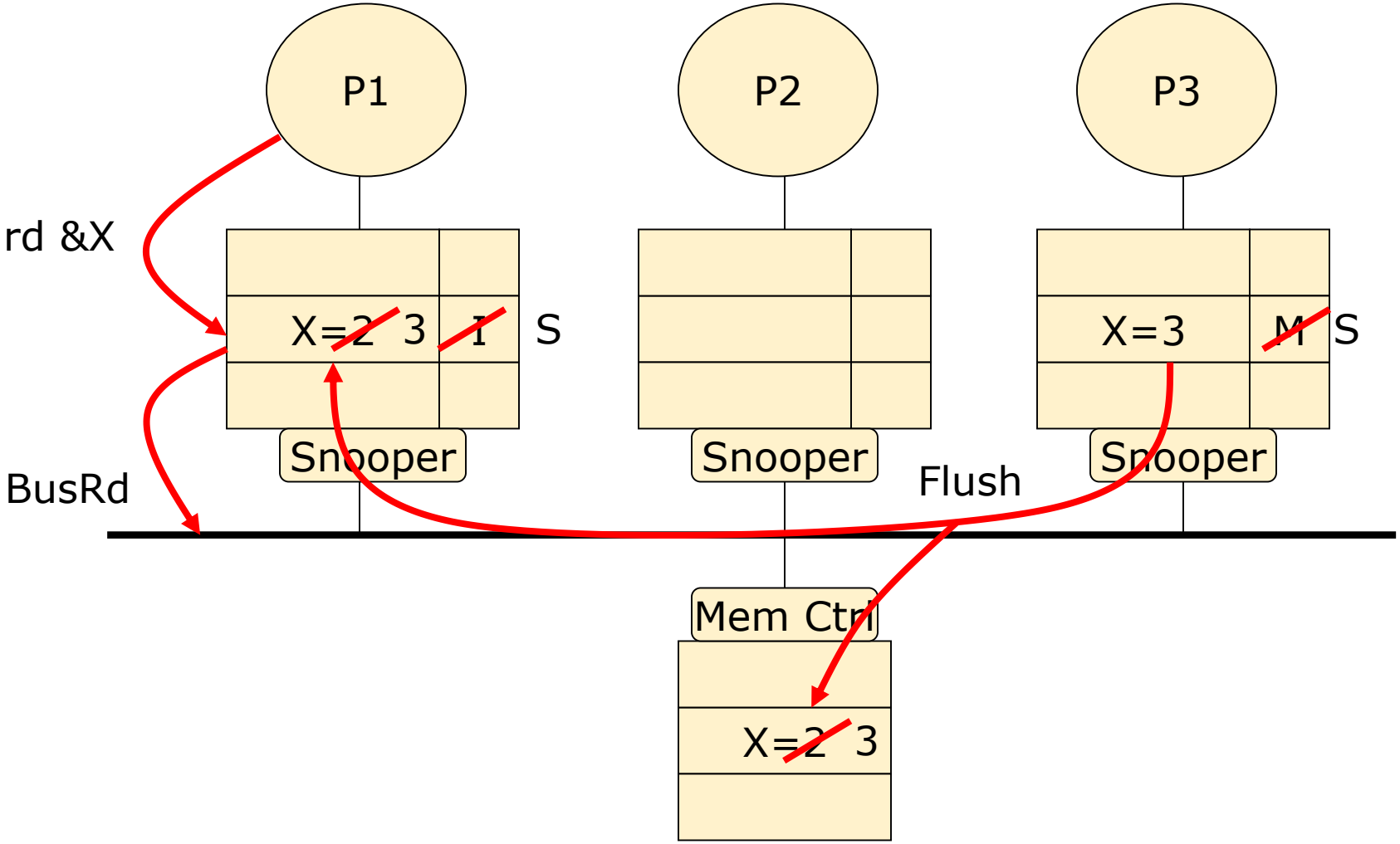


Proc Action	State P1	State P2	State P3	Bus Action	Data From
W3	I	-	M	BusUpgr	-

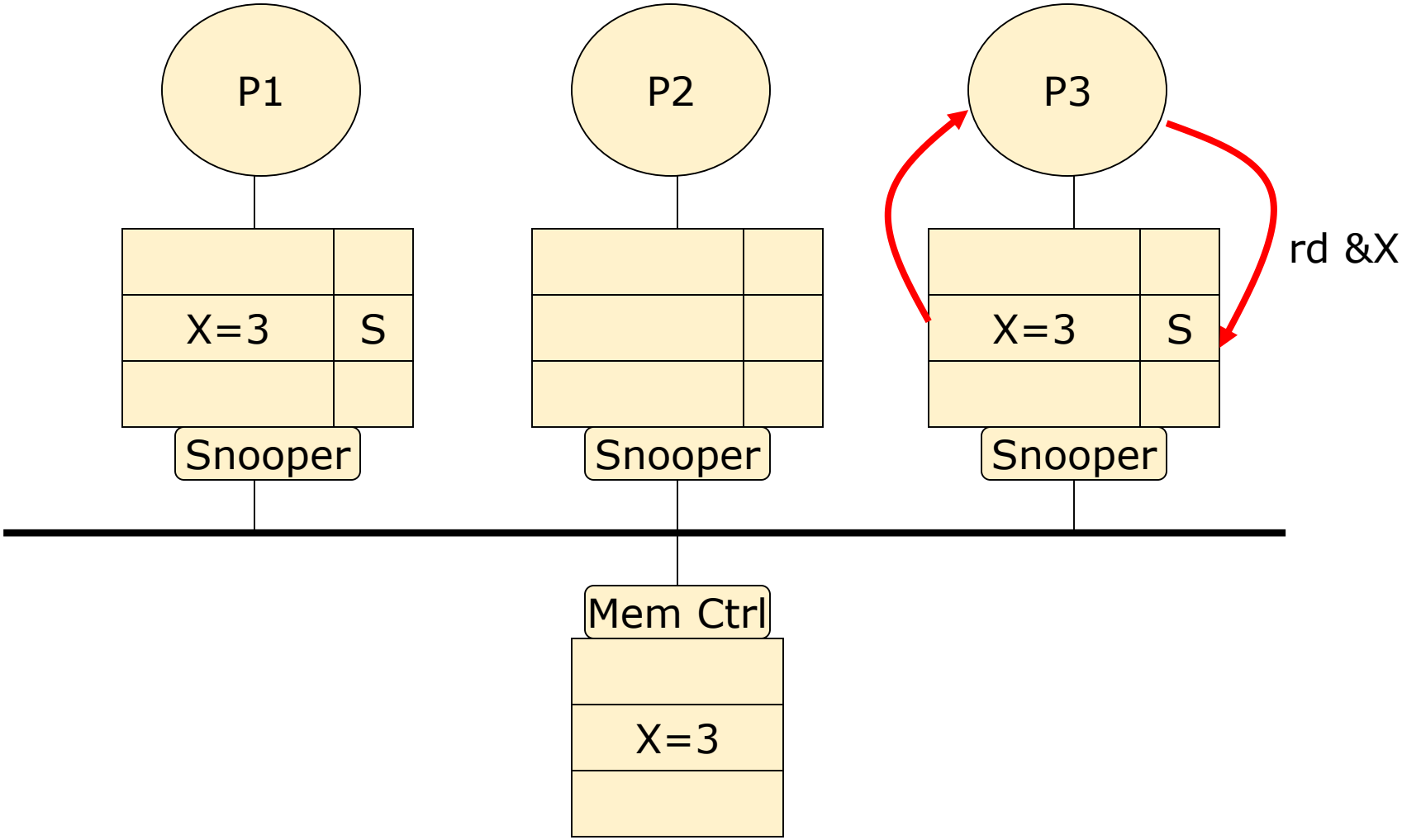


Note: BusUpgr instead of BusRdX. Mem Ctrl knows it can ignore the request

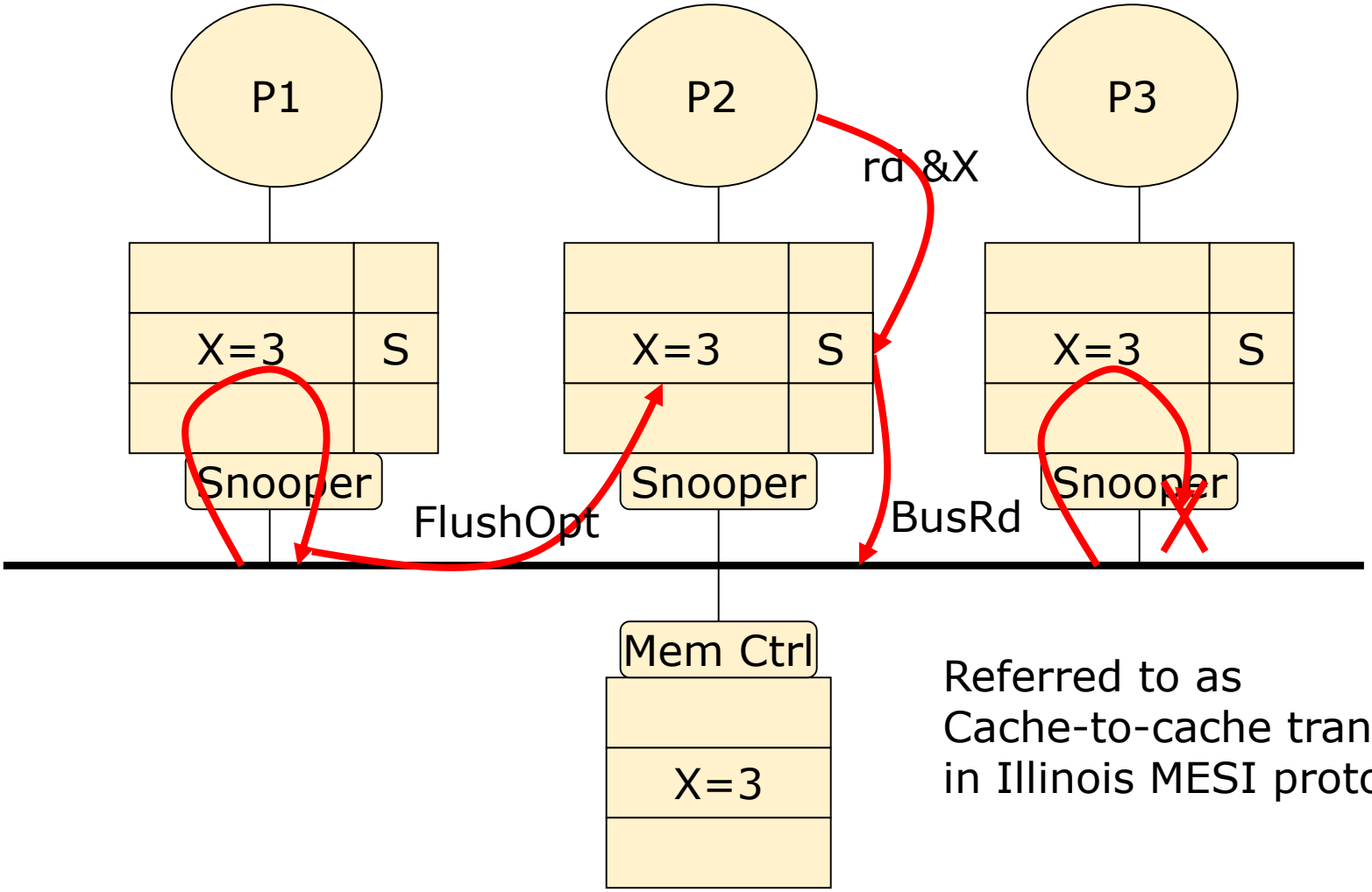
Proc Action	State P1	State P2	State P3	Bus Action	Data From
R1	S	-	S	BusRd/Flush	P3's cache



Proc Action	State P1	State P2	State P3	Bus Action	Data From
R3	S	-	S	-	-



Proc Action	State P1	State P2	State P3	Bus Action	Data From
R2	S	S	S	BusRd/Flush Opt	**P1/P3's cache**



Referred to as Cache-to-cache transfer in Illinois MESI protocol

Summary of State Changes

Proc Action	State P1	State P2	State P3	Bus Action	Data From
R1	**E**	-	-	BusRd	Mem
W1	M	-	-	-	-
R3	S	-	S	BusRd/Flush	P1's cache
W3	I	-	M	BusUpgr	-
R1	S	-	S	BusRd/Flush	P3's cache
R3	S	-	S	-	-
R2	S	S	S	BusRd/Flush Opt	**P1/P3's cache**

Summary

- **Cache Coherence Problem**

- **Snooping-Based Coherence Protocol**
  - Coherence Protocol for Write-Through Caches
  - MSI Protocol with Write-back Caches
  - MESI Protocol with Write-back Caches