# CS / EE 320
# Computer Organization and Assembly Language

Spring 2025

Lecture 28

Shahid Masud

Topics: Virtual Memory, Page Table Addresses, TLB,

Introduction to Multi-Cores and Multi-Processors

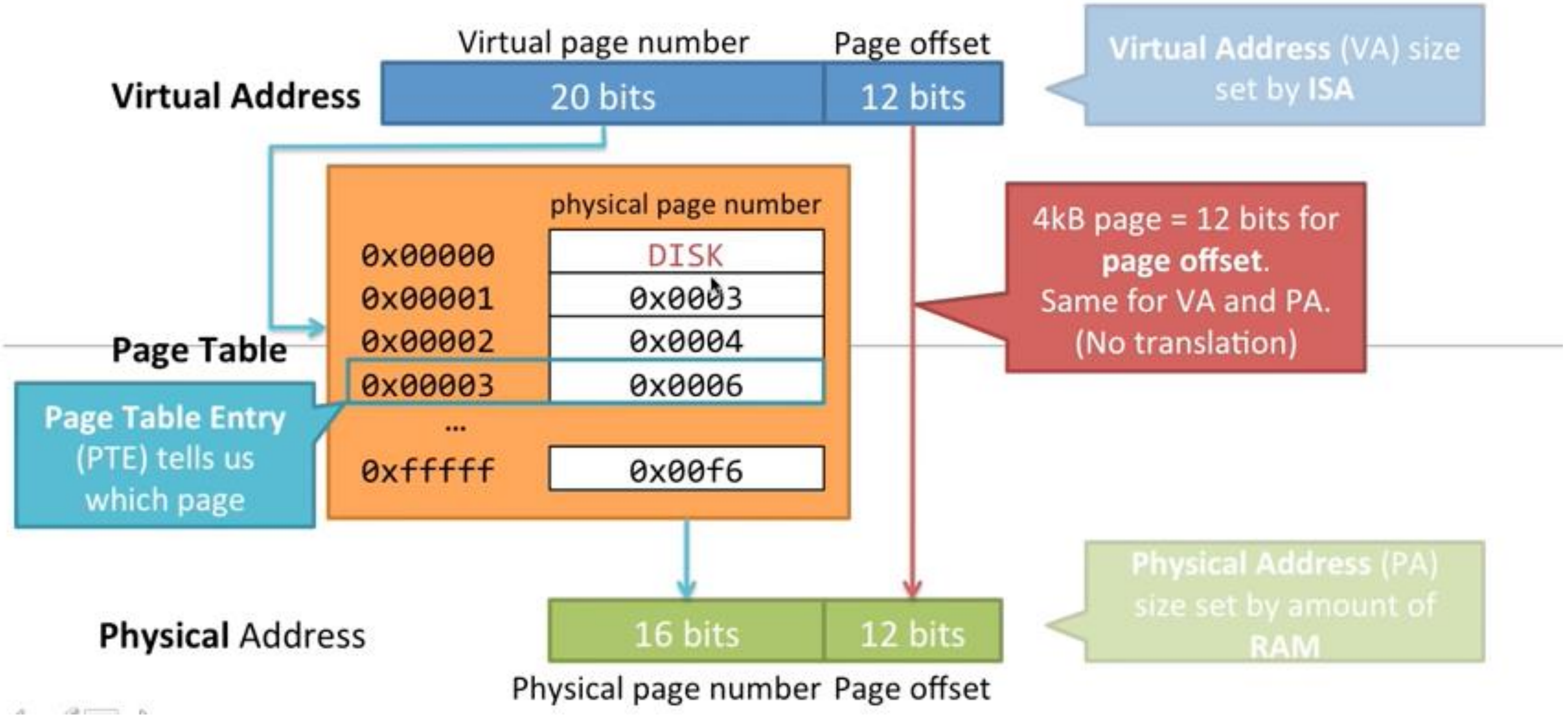
## Topics

- Virtual Memory
  - Page Table Address Calculation - Example
  - Translation Lookaside Buffer
- Superscalar, Parallel Processing and Multiprocessor Architecture
- Using Cache Coherence to achieve synchronization in Multiprocessors
- Study Example of MSI Cache Coherence Protocol in a Multi-Core system

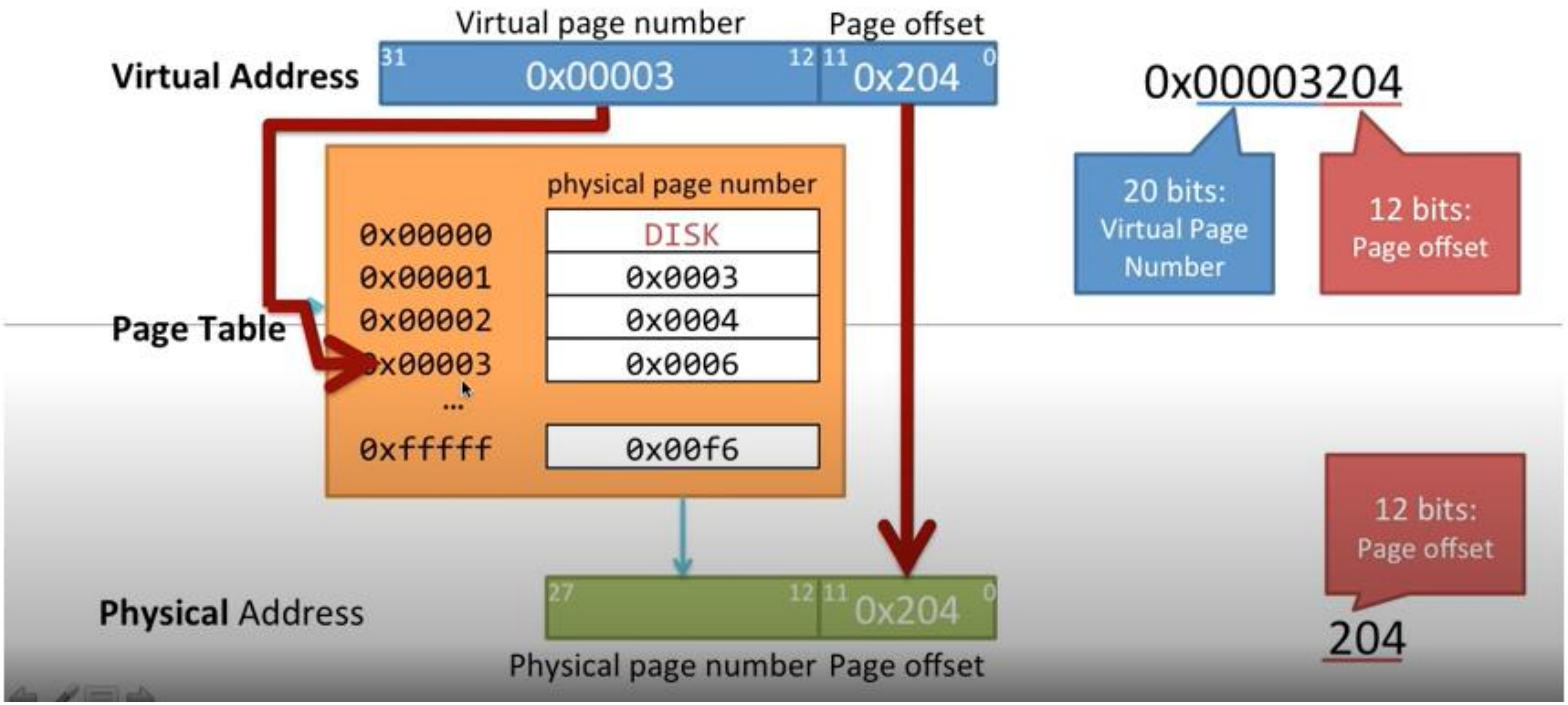
**QUIZ 6 Today**

## Virtual Memory Examples

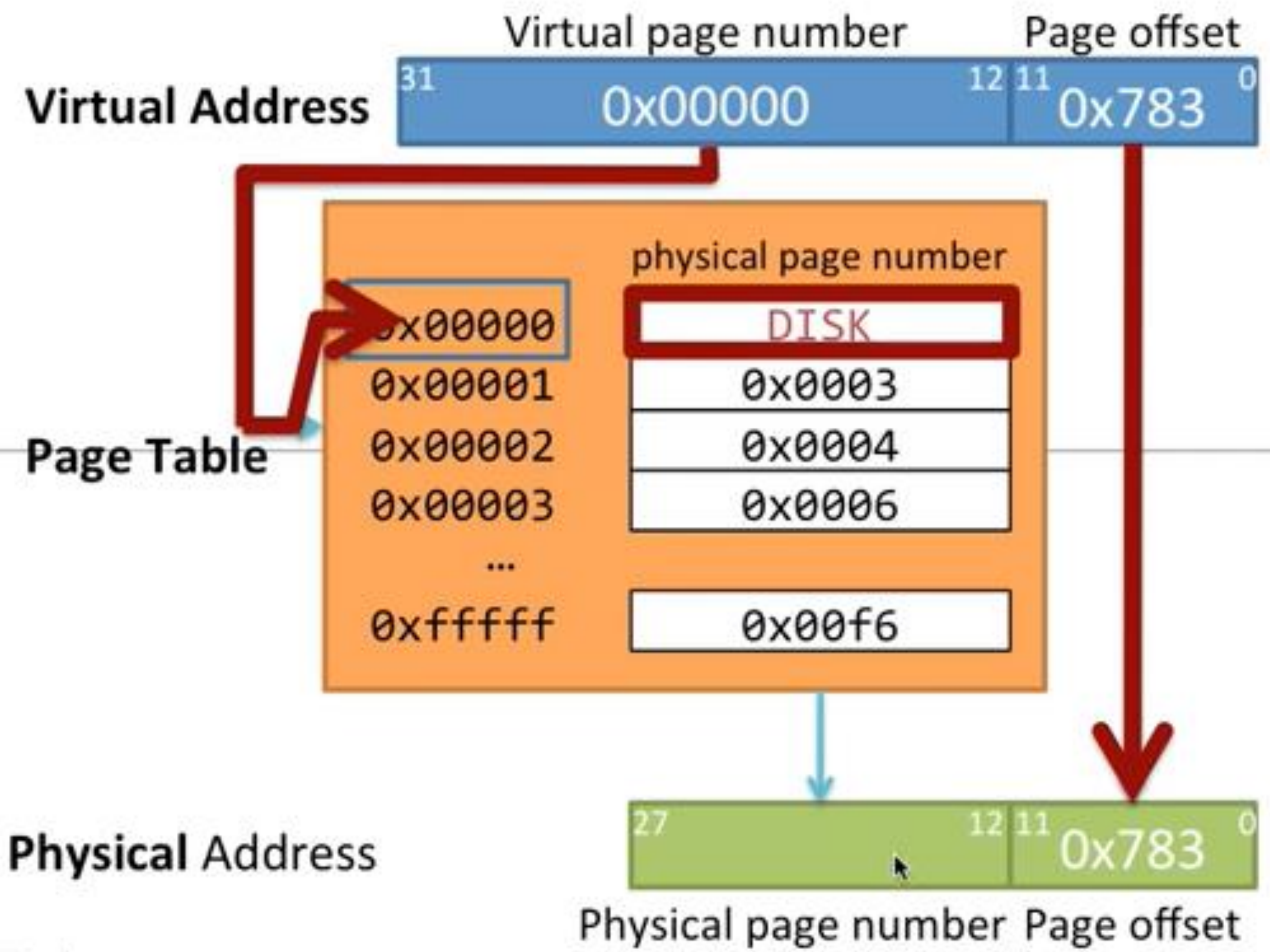
# How to do a Page Table Lookup

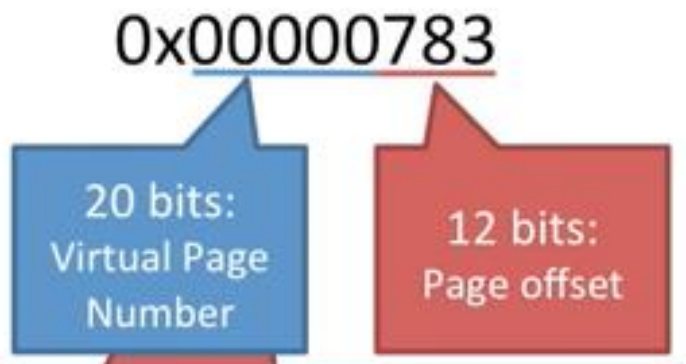


# Example Translation (1)



# Example Translation (2)



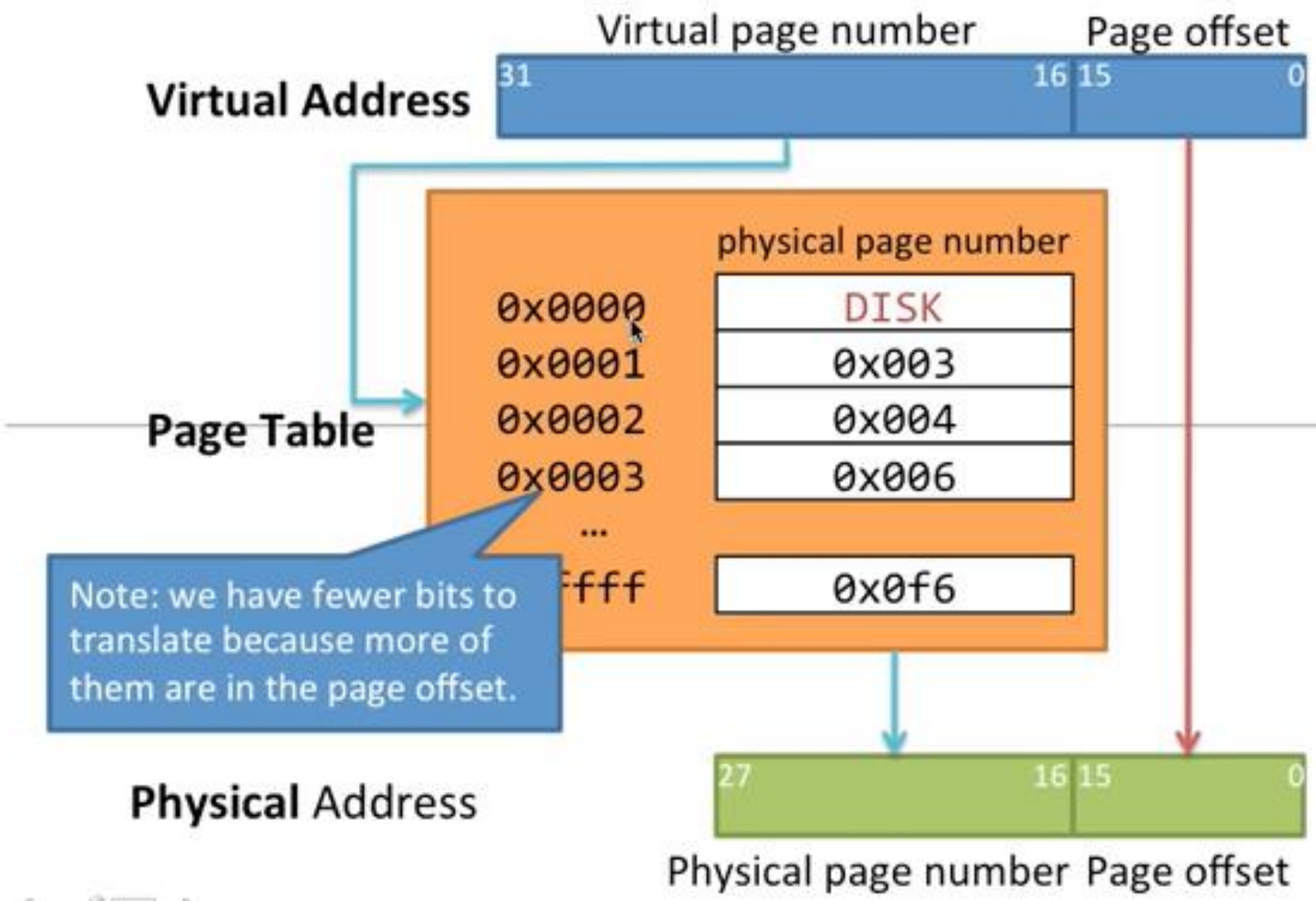


**Q: What do we get when we translate the VA 0x00000783?**

- O 0x00000783
- O 0x00006783
- O Have to get it from the disk
- O Unknown

**A: Have to get it from the disk**
The Page Table says that VA 0x00000 maps to a page that is on the disk, and not in memory.

# Example Translation for 64KB Pages



Note: we have fewer bits to translate because more of them are in the page offset.

**Q: If we have 64kB pages, how many bits do we use for the page offset?**

- O 12
- O 13
- O 16
- O Unknown

**A: 16**

$2^{16}$ is 64k, so we use the lowest 16 bits as the page offset.

# What happens if a Page is not in RAM - Fault

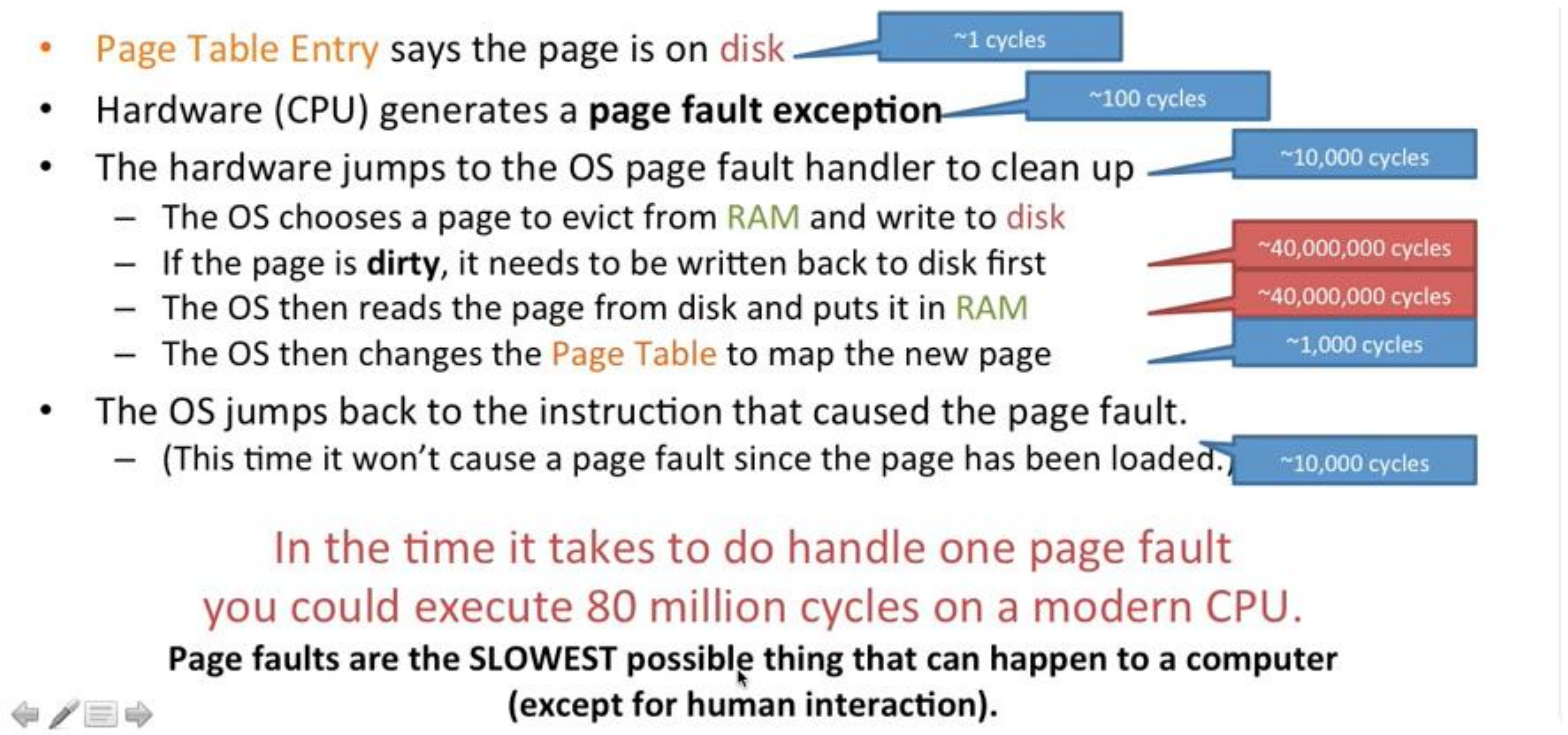
- Page Table Entry says the page is on disk
- Hardware (CPU) generates a **page fault exception**
- The hardware jumps to the OS page fault handler to clean up
  - The OS chooses a page to evict from RAM and write to disk
  - If the page is **dirty**, it needs to be written back to disk first
  - The OS then reads the page from disk and puts it in RAM
  - The OS then changes the Page Table to map the new page
- The OS jumps back to the instruction that caused the page fault.
  - (This time it won't cause a page fault since the page has been loaded.)

"Dirty" means the data has been changed (written). If the page has not been written since it was loaded from disk, then it doesn't have to be written back.

**Q: How long does this take?**
- No time
- A short time
- A long time
- An amazingly, incredibly, painfully long time

**A: An amazingly, incredibly, painfully long time**
Disks are *much* slower than RAM, so every time you have a page fault it takes an amazingly, incredibly, painfully long time.

# How long does a Page Fault Take?



- Page Table Entry says the page is on disk ~1 cycles
- Hardware (CPU) generates a **page fault exception** ~100 cycles
- The hardware jumps to the OS page fault handler to clean up ~10,000 cycles
  - The OS chooses a page to evict from RAM and write to disk
  - If the page is **dirty**, it needs to be written back to disk first ~40,000,000 cycles
  - The OS then reads the page from disk and puts it in RAM ~40,000,000 cycles
  - The OS then changes the Page Table to map the new page ~1,000 cycles
- The OS jumps back to the instruction that caused the page fault.
  - (This time it won't cause a page fault since the page has been loaded.) ~10,000 cycles

In the time it takes to do handle one page fault
you could execute 80 million cycles on a modern CPU.

**Page faults are the SLOWEST possible thing that can happen to a computer (except for human interaction).**

## Virtual Memory in Practice

- VM is great:
  - Unlimited programs/memory, protection, flexibility, etc.

- But it comes at a high cost:
  - *Every* memory operation has to look up in the page table
  - Need to access **1) the page table** and **2) the memory address** (2x memory accesses)

  (Remember, 1.33 memory accesses per instruction. This is going to hurt.)

- How can we make a page table look up ***really really*** fast?
  - Software would be far too slow
    (e.g., an extra 5 instructions for every memory access would kill performance)

- Perhaps a hardware page table **cache**?

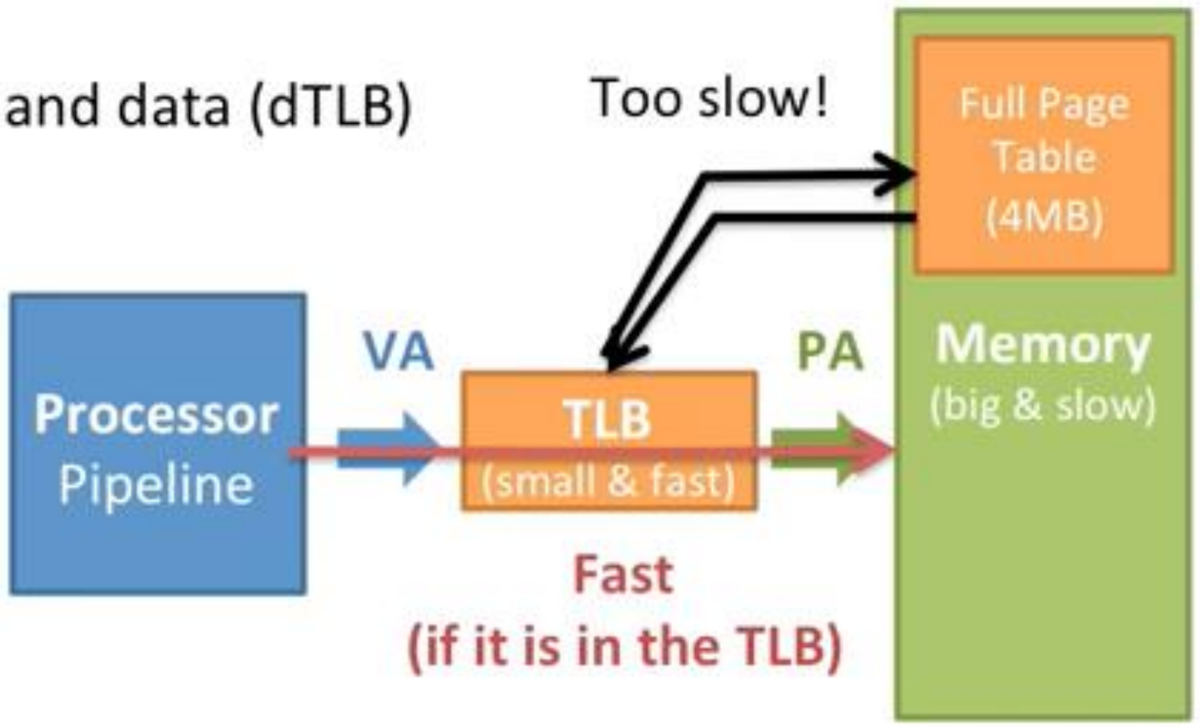
# Making VM Fast through the TLB

- To make VM fast we add a special **Page Table cache:** the **Translation Lookaside Buffer (TLB)**
  - Fast: less than 1 cycle (have to do it for every memory access)
  - Very similar to a cache

- To be fast, TLBs must be small:
  - Separate TLBs for instructions (iTLB) and data (dTLB)
  - 64 entries, 4-way (4kB pages)
  - 32 entries, 4-way (2MB pages)

  (Page Table is 1M entries)

Lots of locality!
Miss rates are typically only a few percent.



# What can happen when we access memory?

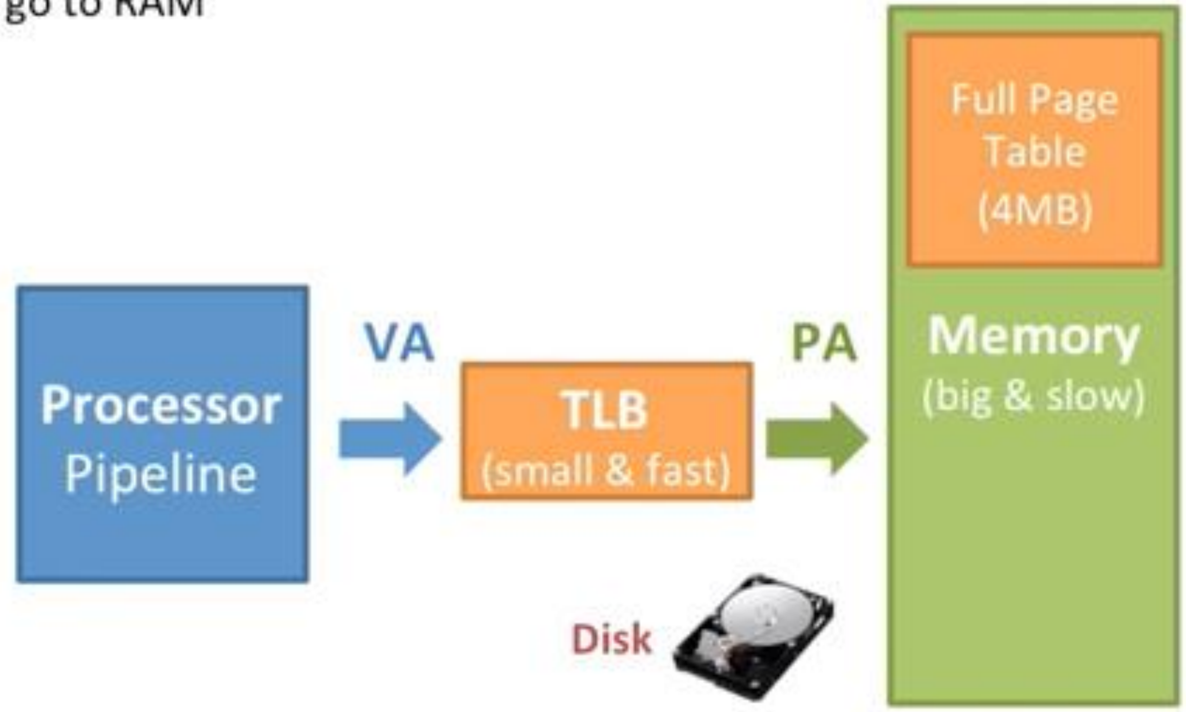
**Good: Page in RAM**

- **PTE in the TLB**
  - **Excellent**
  - **<1** cycle to translate, then go to RAM (or cache)
- **PTE not in the TLB**
  - **Poor**
  - **20-1000** cycles to load PTE from RAM, then go to RAM

With 1.33 memory accesses per instruction we can't afford 20-1000 cycles very often.

**Bad: Page not in RAM**

- **PTE in the TLB** (unlikely)
  - **Horrible**
  - 1 cycle to know it's on disk
  - **~80M** cycles to get it from disk
- **PTE not in the TLB**
  - **(ever so slightly more) horrible**
  - **~80M** cycles to get it from disk



## MULTI-PROCESSORS AND MULTI-CORES

# Superscalar and Instruction Level Parallelism

- In Superscalar architectures, multiple Instructions are Fetched in Pipeline
- and Multiple Integer and Floating-Point units process multiple pipelines simultaneously

# NUMA Architectures

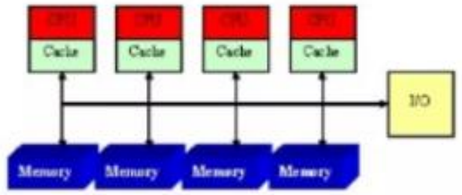
- Non-Uniform Memory Architectures – use Message Passing or Shared Disk for Synchronization

Groups of processors (NUMA node) have their own local memory

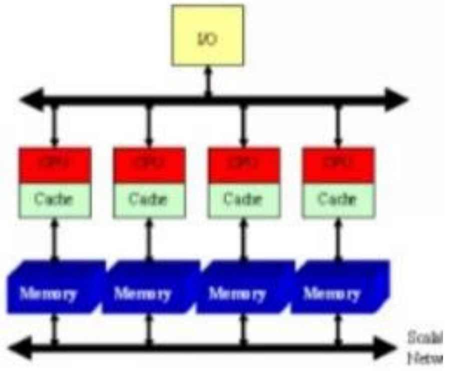
- Any processor can access any memory, including the one not "owned" by its group (remote memory)
- Non-uniform: accessing local memory is faster than accessing remote memory

## UMA, NUMA & NUMA SMP architect

**Uniform memory access**(UMA): all processors have same latency to access memory. This architecture is scalable only for limited nmber of processors.



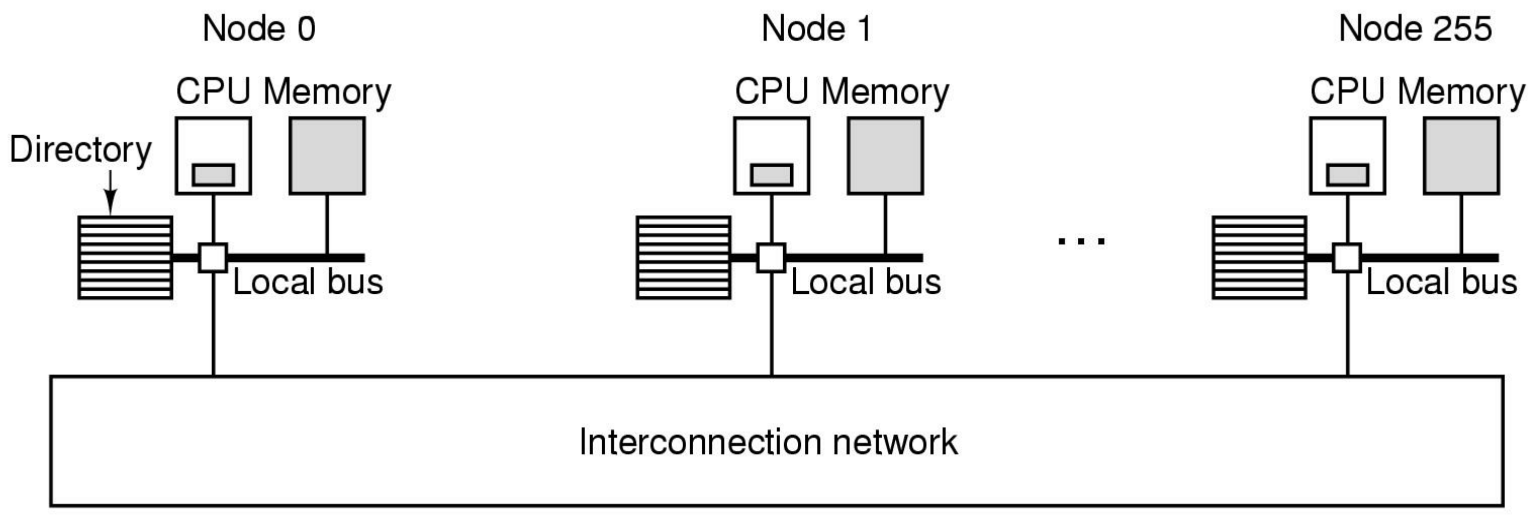
**Nom Uniform Memory Access**(NUMA): each processor has its own local memory, the memory of other processor is accessible but the lantency to access them is not the same which this event called " remote memory access"



# Modern Computer Architectures

- **DSA** – Domain Specific Architectures, Example is Google Tensor Processor, Tensor Flow Software customized language for ML, AI
- **Reconfigurable Architectures** – Based on run-time hardware configuration using FPGA (Field Programmable Logic Arrays)
- **Hybrid Architectures** – Combination of multiple type of Processor Cores, DSA and Reconf. All combined into one Chip.

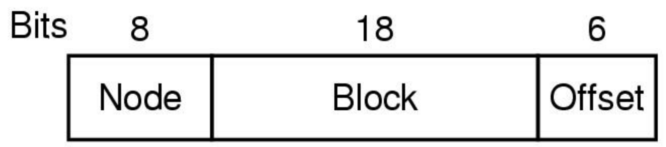
## Multiprocessor Hardware



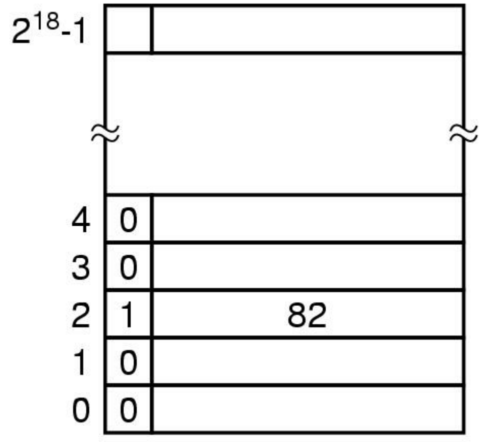
(a)

**Directory-based coherence** is a mechanism used to handle cache coherence problems in distributed shared memory (DSM) systems. In this approach: [5]

- Directories are used to manage caches instead of bus snooping. [1]
- The data being shared is placed in a common directory that maintains coherence between caches. [1]
- The directory keeps track of where each copy of a block is cached and its state in each cache. [1]



(b)



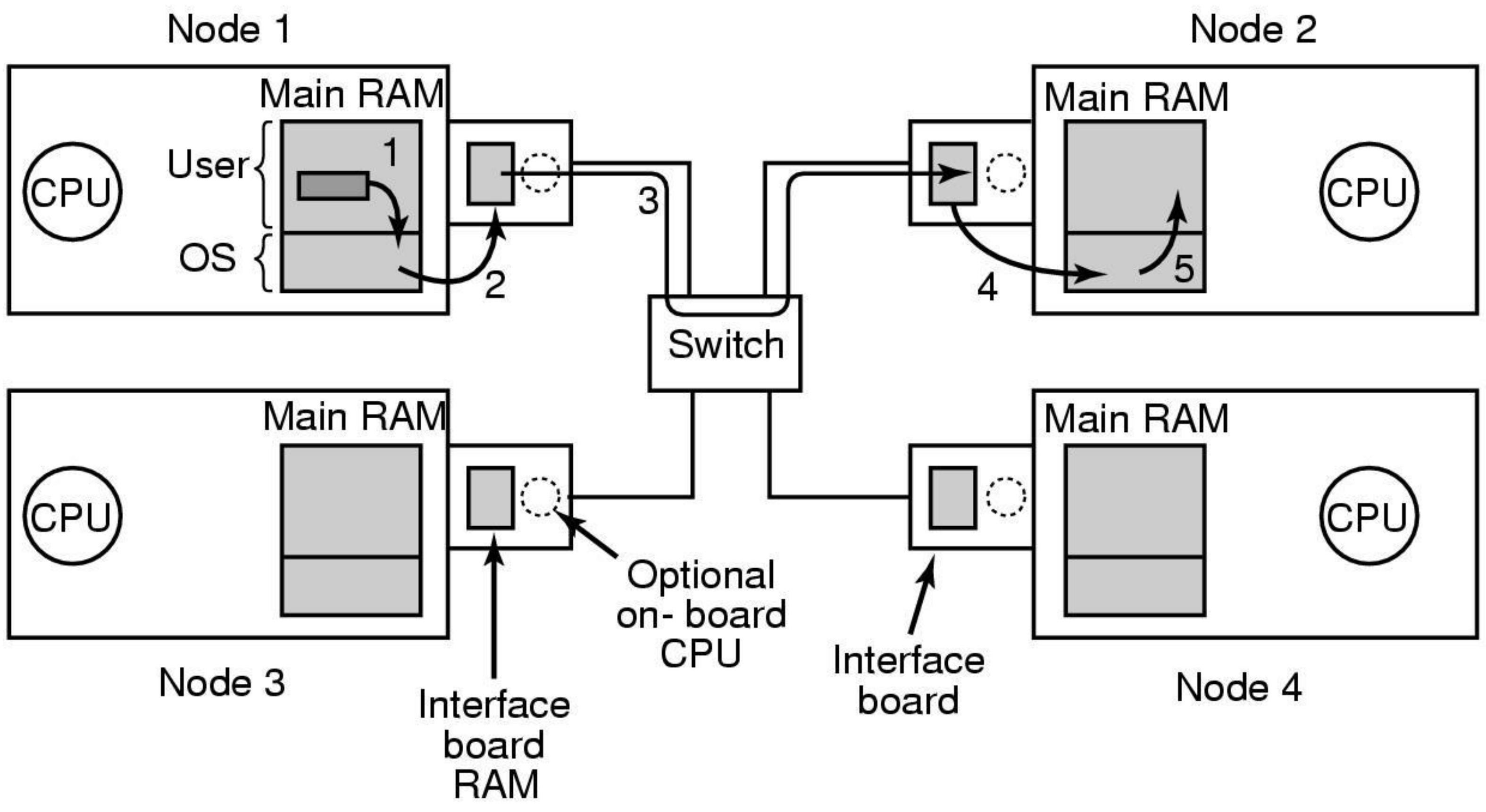
(c)

(a) 256-node directory-based multiprocessor

(b) Fields of 32-bit memory address

(c) Directory at node 36

## Multicomputer Hardware (Cluster Computing)



Network interface boards in a multicomputer

# Multi-Cores Synchronization Through Cache Coherence Protocols

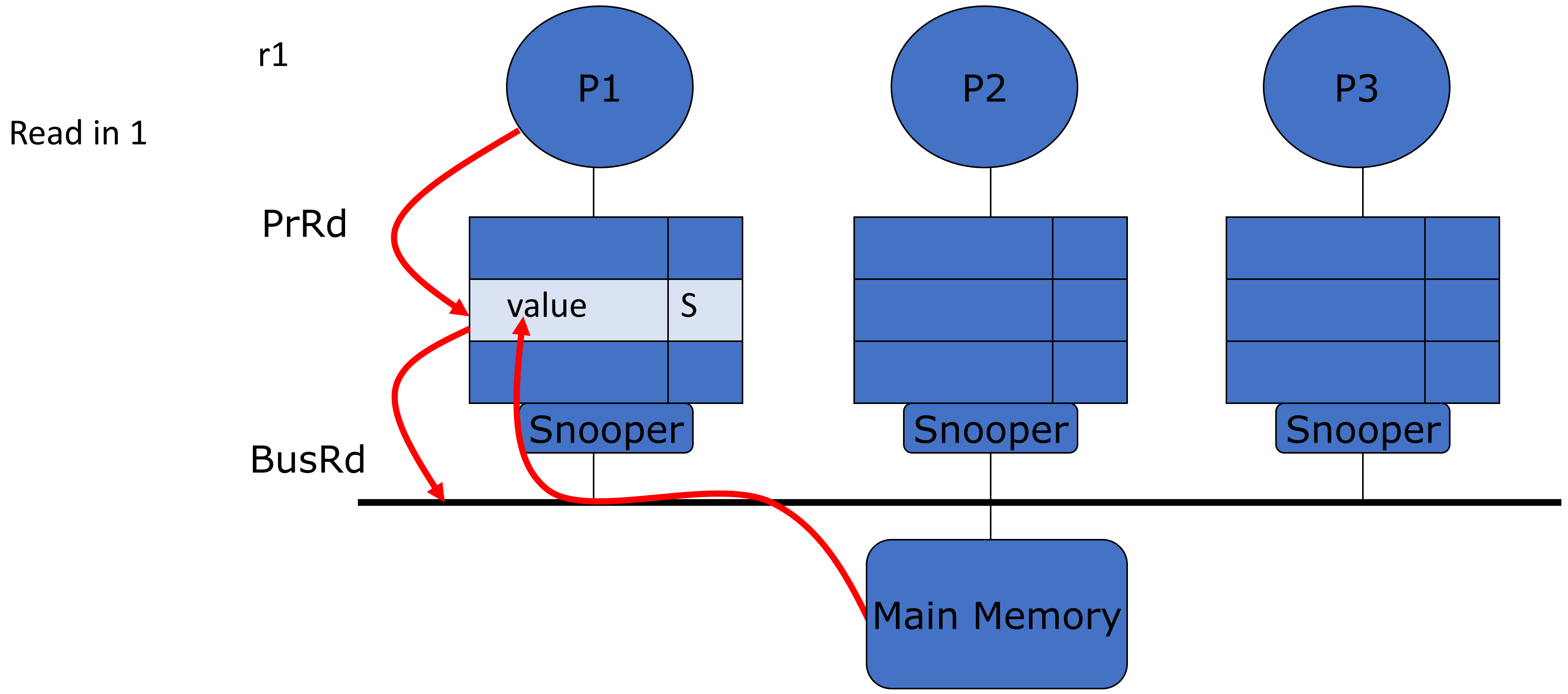
## MSI Protocol (Modified, Shared, Invalid)

- There are three processors.
- Each is reading/writing the same value from memory where r1 means a read by processor 1 and w3 means a write by processor 3.
- For simplicity sake, the memory location will be referred to as “value.”
- The memory access stream is:

  r1, r2, w3, r2, w1, w2, r3, r2, r1

**SNOOPY BUS BASED CACHE COHERENCE PROTOCOLS**

# Demonstrating a Cache Coherence Protocol - 1

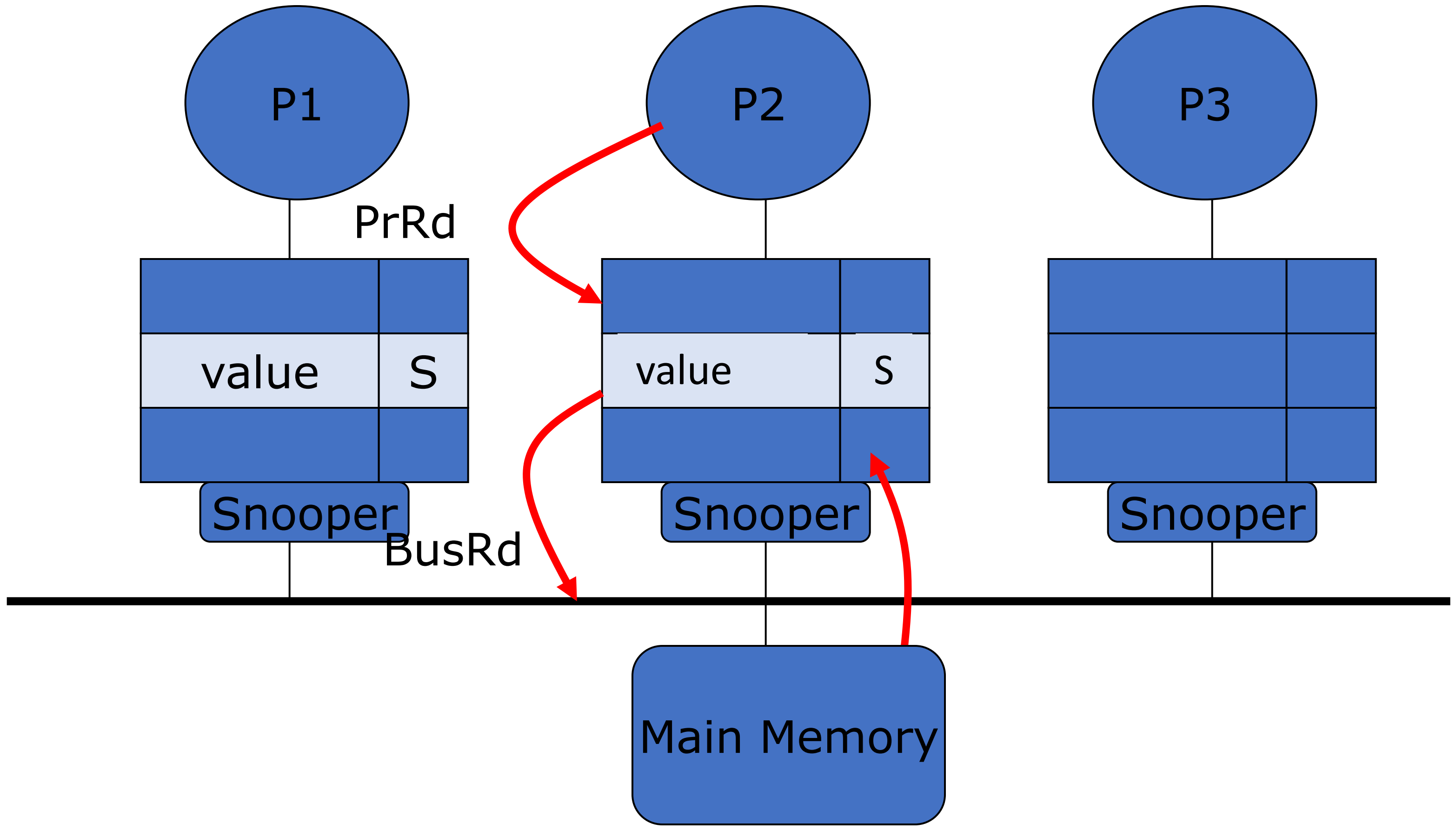


P1 wants to read the value. The cache does not have it and generates a BusRd for the data. Main memory controller provides the data. The data goes into the cache in the shared state.

# Demonstrating a Cache Coherence Protocol - 2

r2

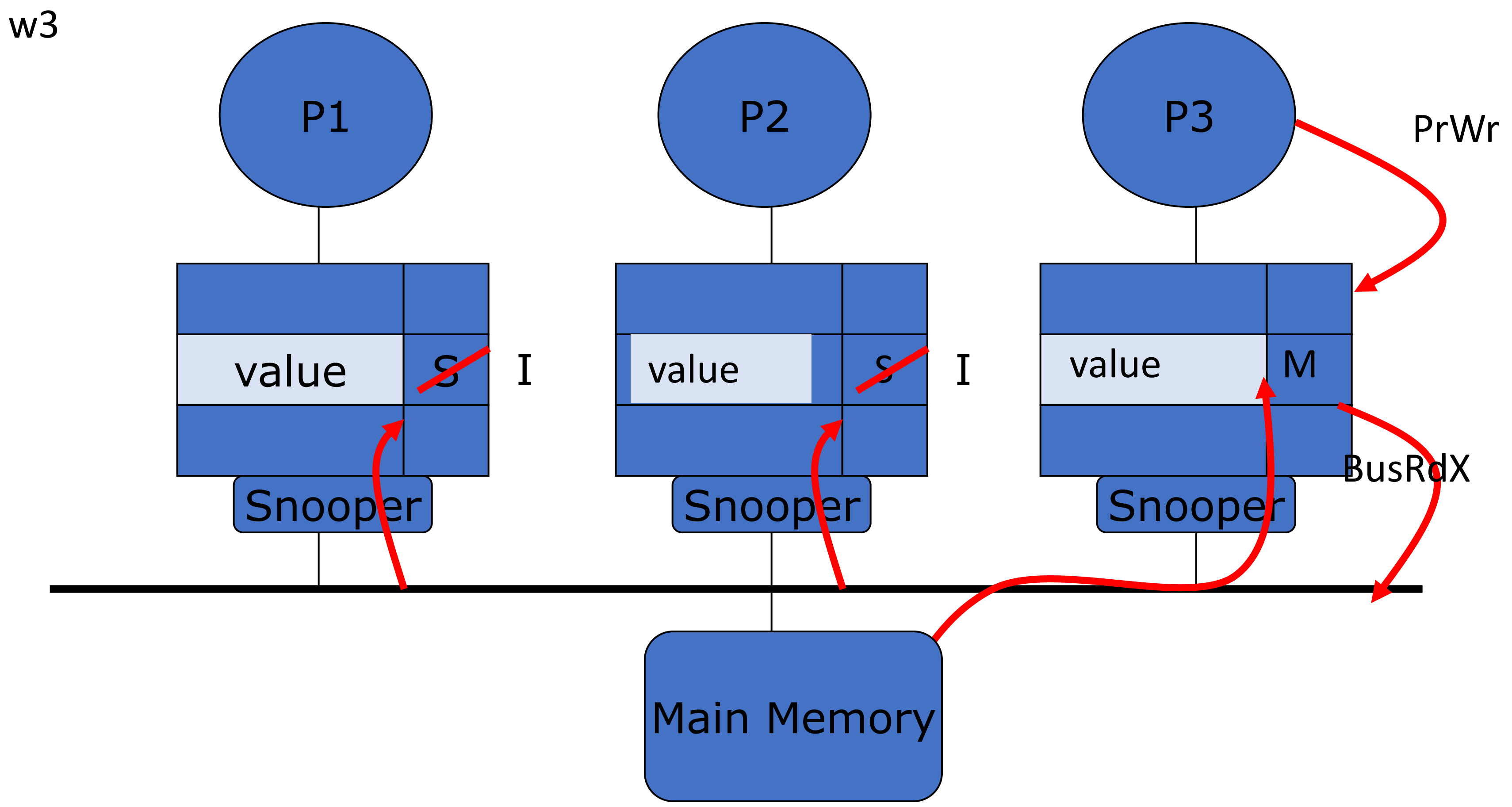
Read in 2



P2 wants to read the value. Its cache does not have the data, so it places a BusRd to notify other processors and ask for the data. The main memory controller provides the data.

# Demonstrating a Cache Coherence Protocol - 3

Write in 3

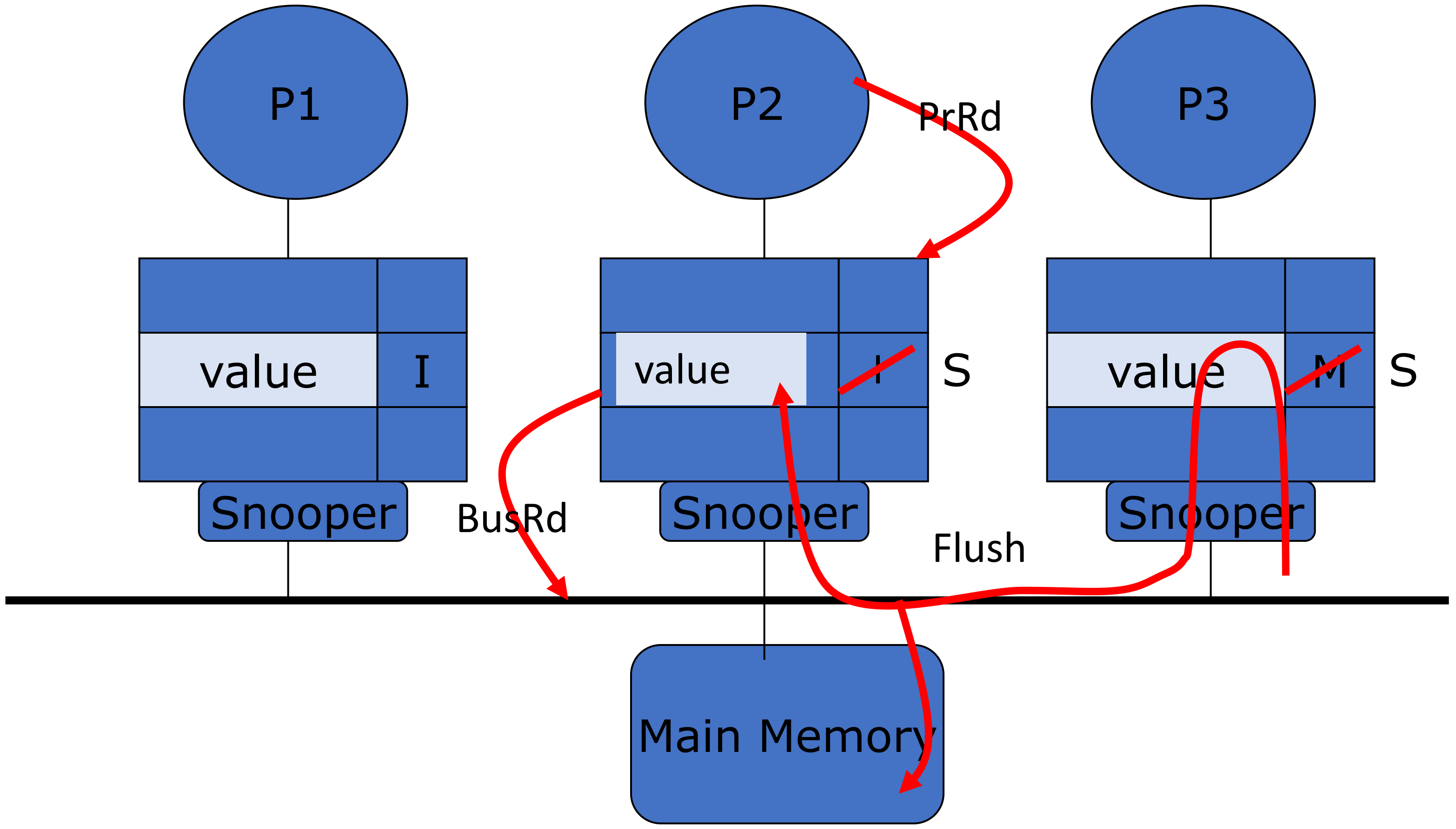


P3 wants to write the value. It places a BusRdX to get exclusive access and the most recent copy of the data. The caches of P1 and P2 see the BusRdX and invalidate their copies. Because the value is still up-to-date in main memory, memory provides the data.

# Demonstrating a Cache Coherence Protocol - 4

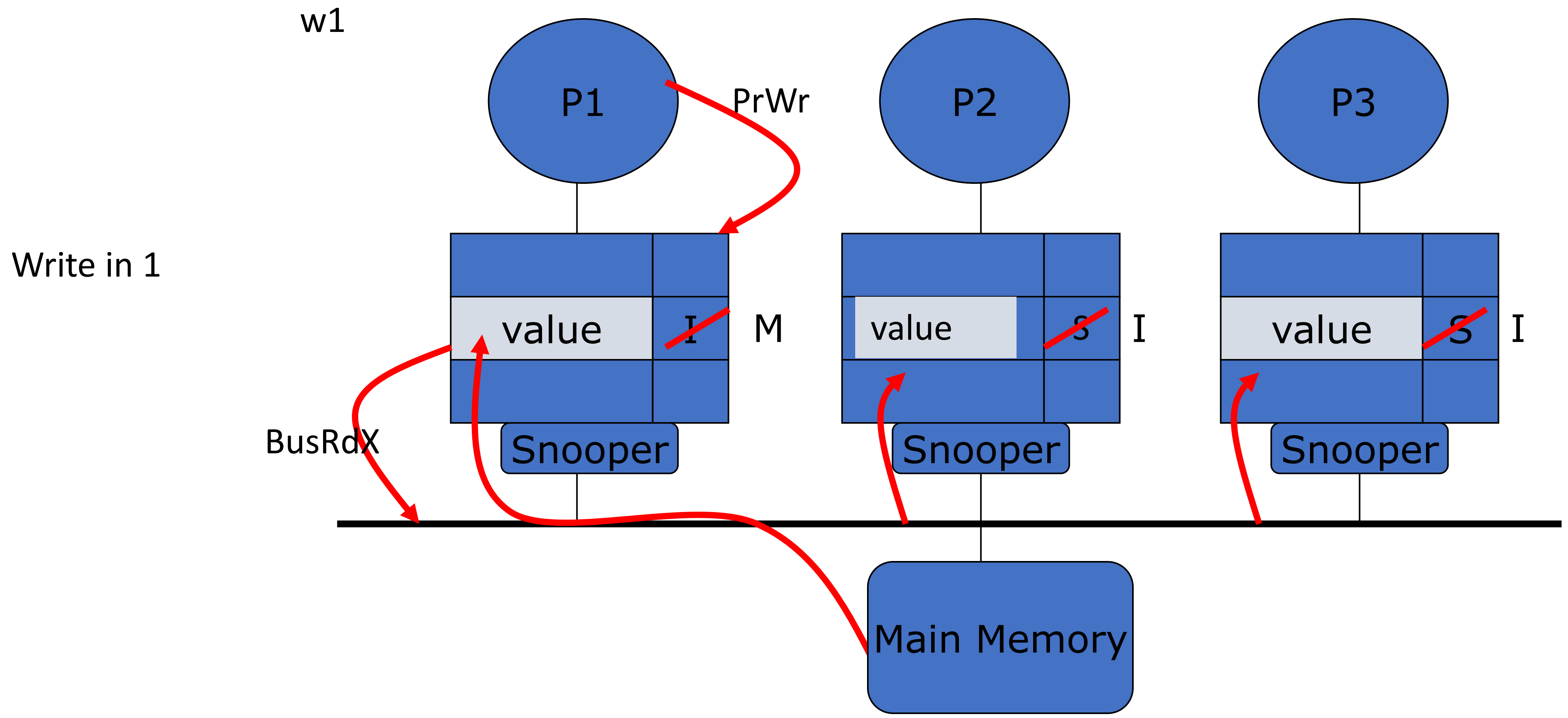
r2

Read in 2



P2 wants to read the value. P3's cache has the most up-to-date copy and will provide it. P2's cache puts a BusRd on the bus. P3's cache snoops this and cancels the main memory access because it will provide the data. P3's cache flushes the data to the bus.

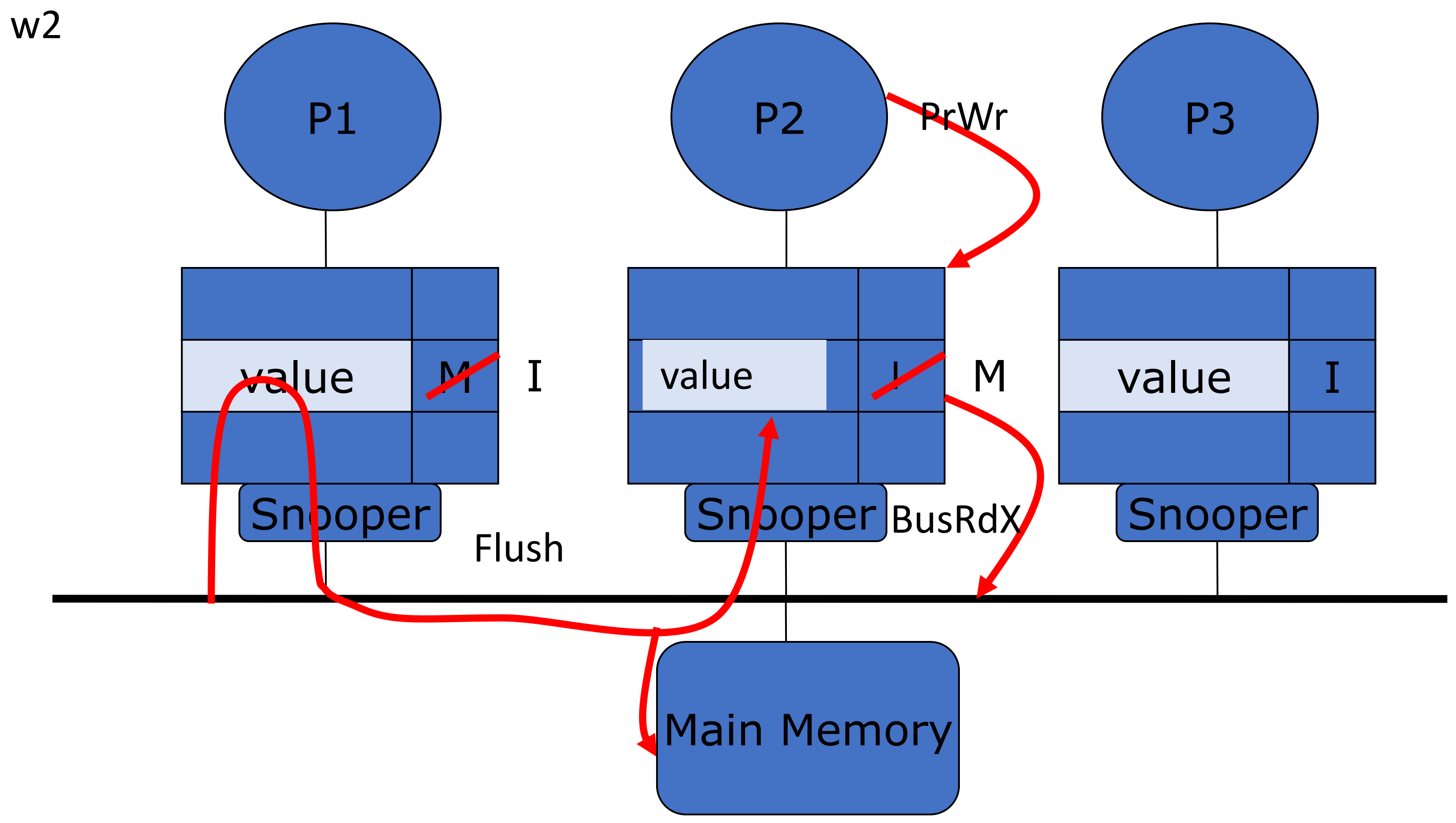
# Demonstrating a Cache Coherence Protocol - 5



P1 wants to write to its cache. The cache places a BusRdX on the bus to gain exclusive access and the most up-to-date value. Main memory is not stale so it provides the data. The snoopers for P2 and P3 see the BusRdX and invalidate their copies in cache.

# Demonstrating a Cache Coherence Protocol - 6

Write in 2

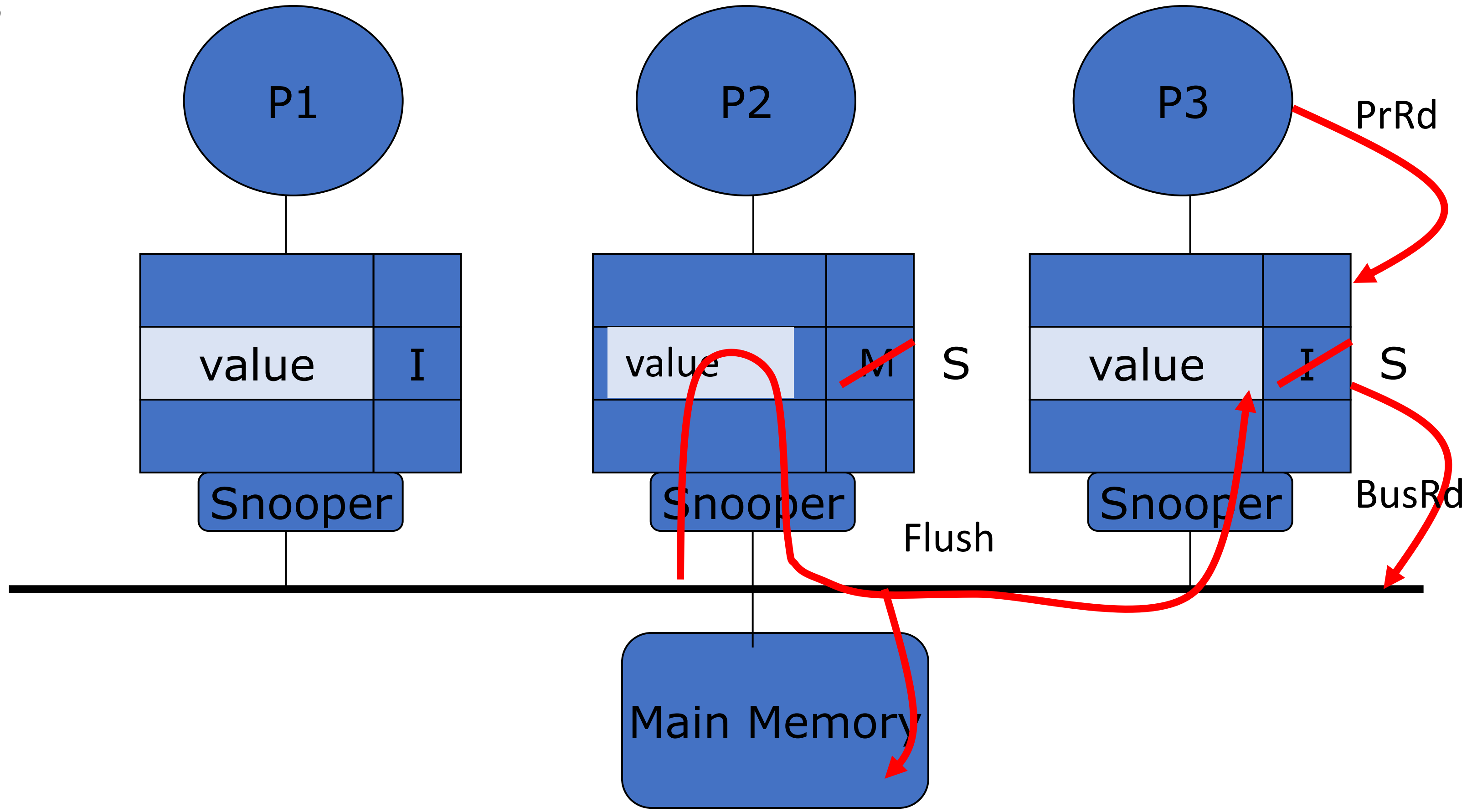


P2 wants to write the value. Its cache places a BusRdX to get exclusive access and the most recent copy of the data. P1's snooper sees the BusRdX and flushes the data to the bus. Also, it invalidates the data in its cache and cancels the main memory access.

# Demonstrating a Cache Coherence Protocol - 7

r3

Read in 3

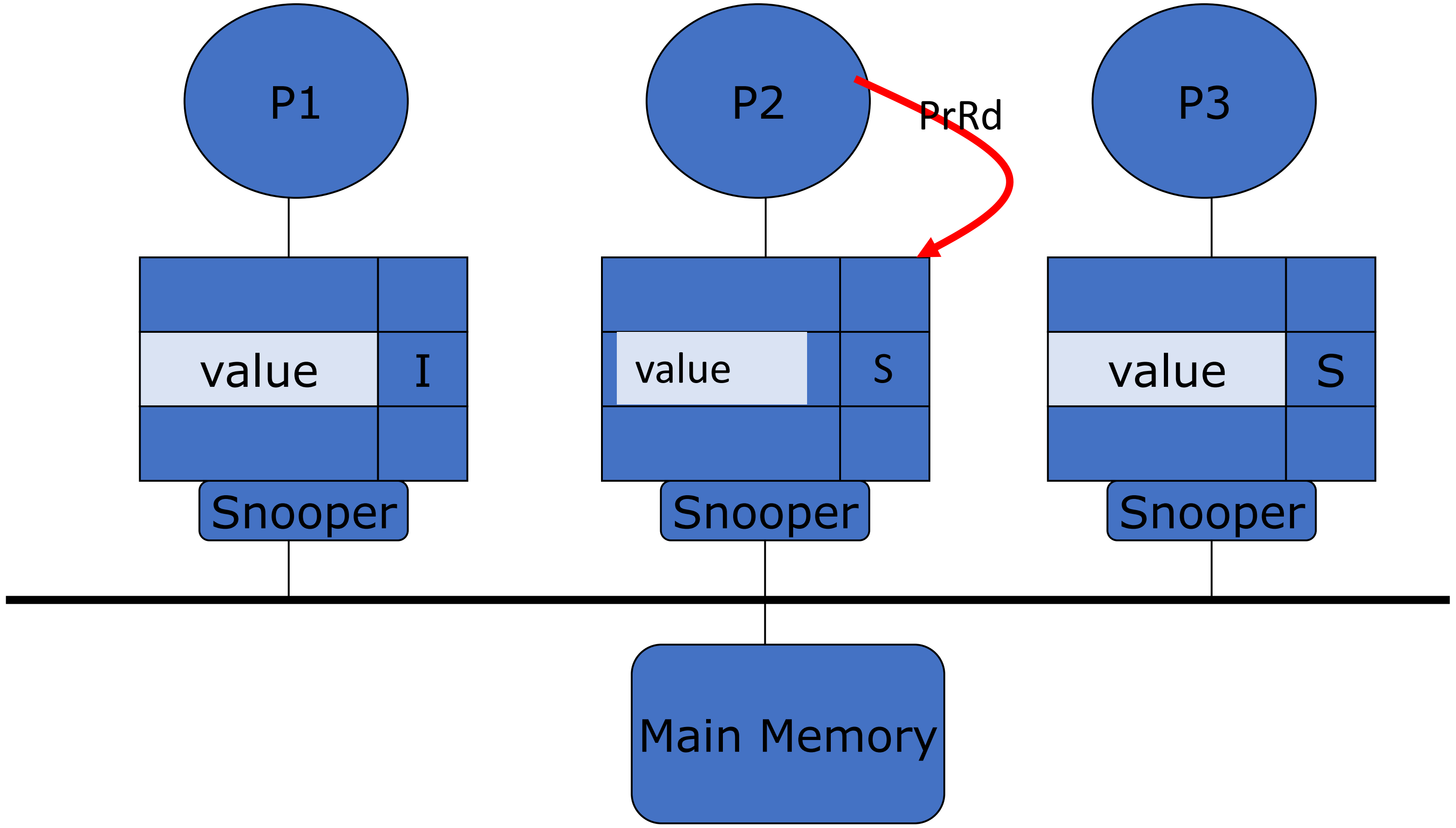


P3 wants to read the value. Its cache does not have a valid copy, so it places a BusRd on the bus. P2 has a modified copy, so it flushes the data on the bus and changes the status of the cache data to shared. The flush cancels the main memory access and updates the data in memory as well.

# Demonstrating a Cache Coherence Protocol - 8

r2

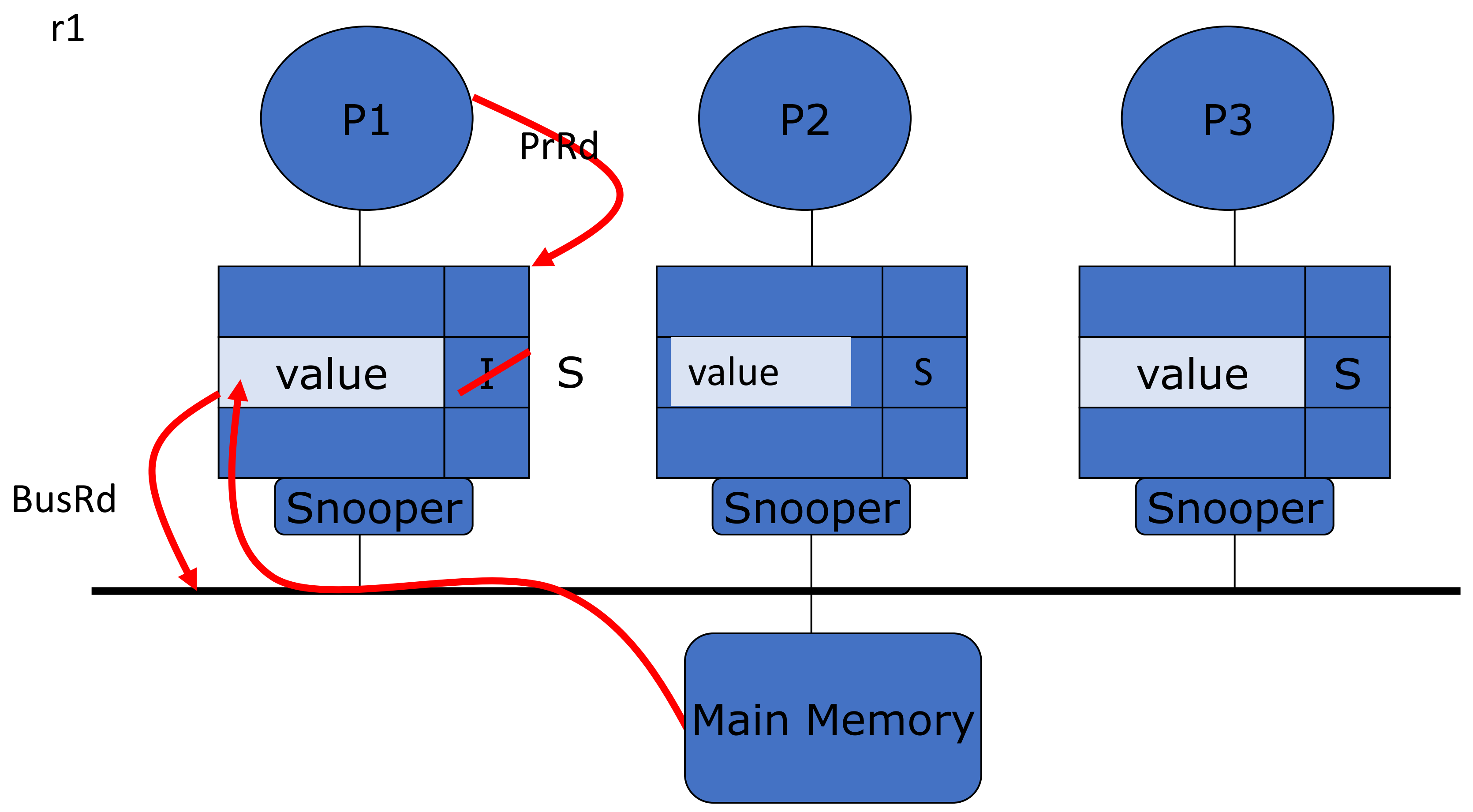
Read in 2



P2 wants to read the value. Its cache has an up-to-date copy. No bus transactions need to take place as there is no cache miss.

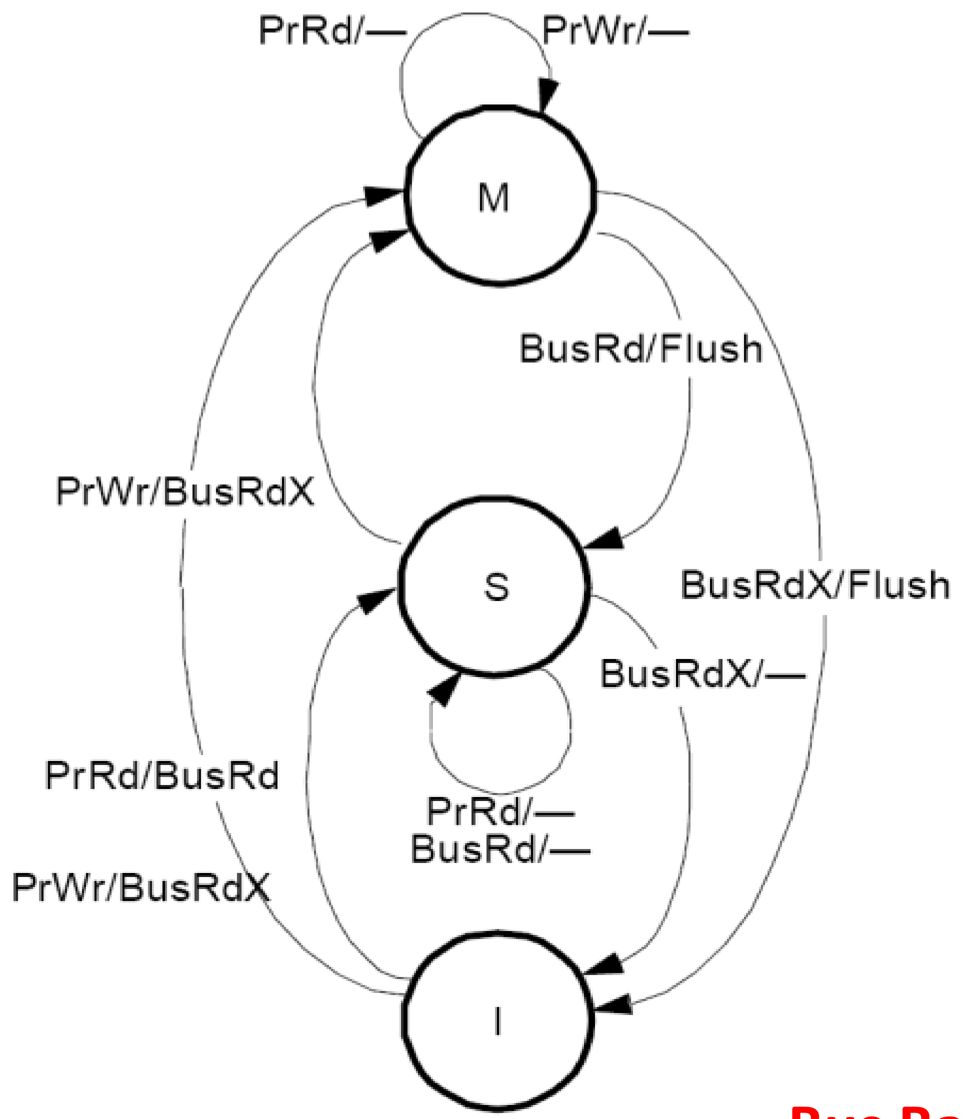
# Demonstrating a Cache Coherence Protocol - 9

Read in 1

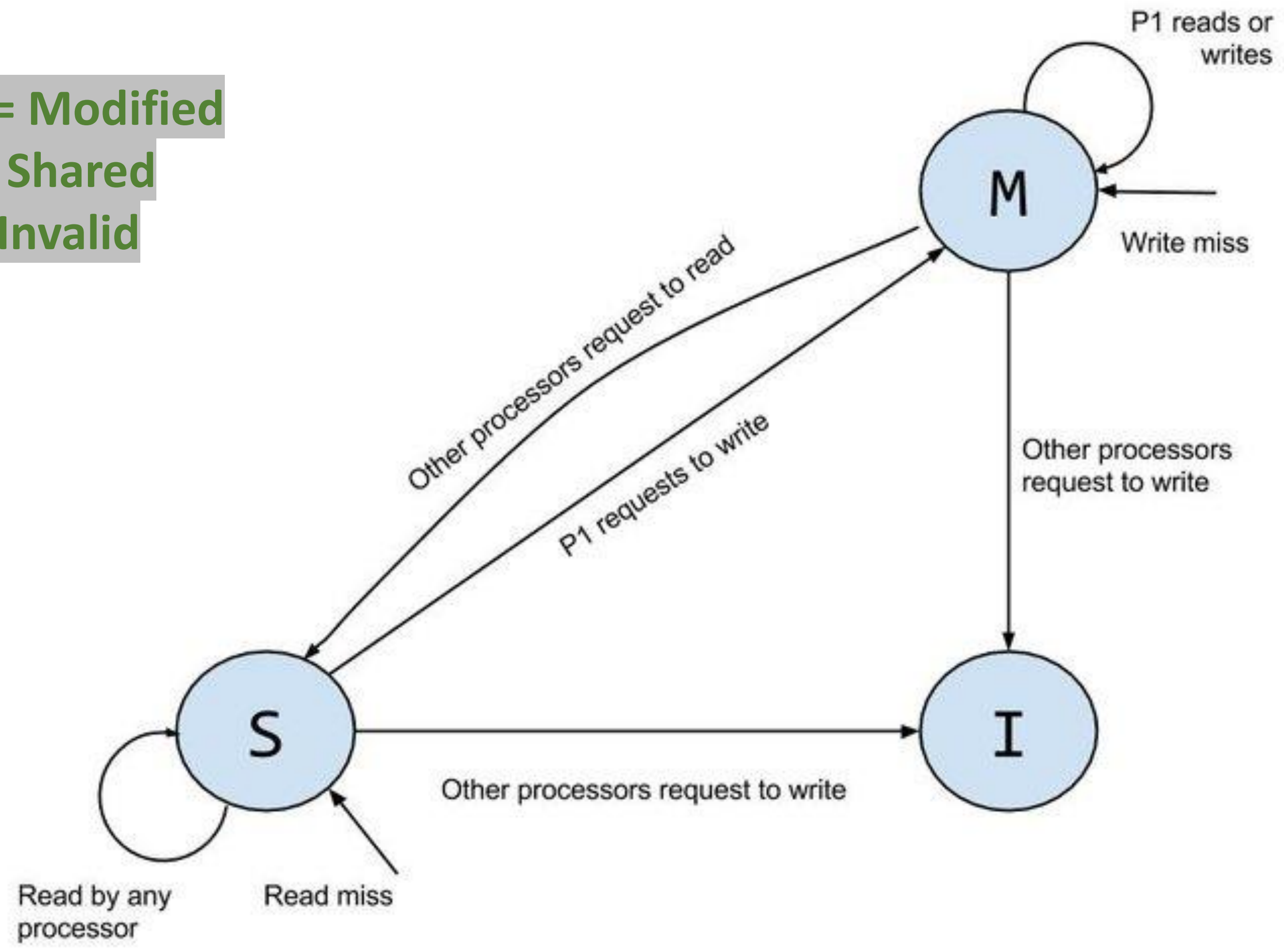


P1 wants to read the value. The cache does not have it, so it places a BusRd onto the bus for the data. The main memory controller provides the data as it has an up-to-date copy. The data goes into the cache in the shared state.

# FSM of MSI for Proc Rd and Proc Wr



M = Modified
S = Shared
I = Invalid


P1 reads or writes
M
Write miss
Other processors request to read
P1 requests to write
Other processors request to write
S
I
Other processors request to write
Read by any processor
Read miss

**Bus Based Snoopy Protocol for Cache Coherence**

## Readings

- Chap 5, 6 of P&H Textbook

- Image Credits: Youtube channel of David Schaffer