Multiprocessors and Thread-Level Parallelism

UNIT IV

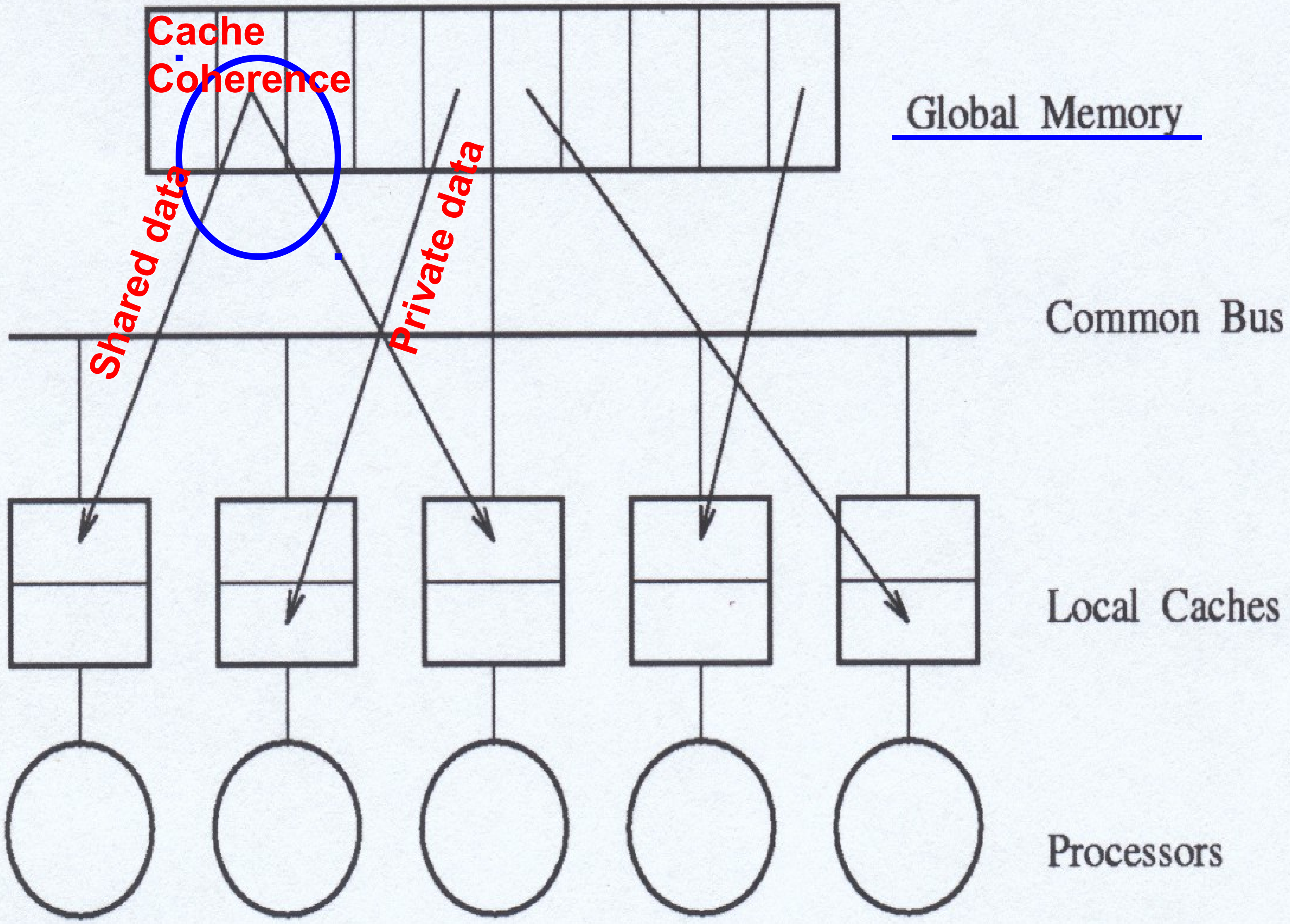
Symmetric Shared-Memory Architectures

SMP[*symmetric (shared-memory) multiprocessors]*

- There is a Uniform Memory Access from every processor
- Also there is a single main memory that has a symmetric relationship to all processors
- Therefore it is "Symmetric "➔ all PE's have same access to I/O, memory, executive (OS) capability etc.
- This style of architecture is sometimes called *UMA for uniform memory access.*

Overview

- The use of large, multi-level caches can substantially reduce memory bandwidth demands of a processor
  - → Multi-processors, each having a local cache, share the same memory system
- Cache both shared and private data
  - Private: used by a single processor → migrate to the cache
  - Shared: use by multiple processors → replicate to the cache
- Cache coherence Problem



Basic organizational units for data sharing: block

(a) Multiprocessor cache architecture

Symmetric shared-memory multiprocessor

- Definition: Several processors shared a single physical memory connected by a shared bus.
  - Small scale, shared memory machines usually support the caching of both shared & private data
  - Private data are used by single processor while shared data are used by multiple processors
  - When a private item is cached, its location is migrated to the cache
  - When shared data are cached, the shared value may be replicated in multiple caches.
  - Caching of shared data introduces a new problem: **cache coherence**

Cache Coherence Problem

Write-through cache

Initially, the two caches do not contain X

Time	Event	Cache Contents for CPU A	Cache Contents for CPU B	Memory contents location X
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

In the above table,

1.CPU A reads the value of X = 1 from memory into cache

2.CPU B reads the value of X = 1 from memory into cache

3.CPU A writes the value of X = 0 from cache into memory but CPU B hold the value of X = 1

2 different aspects of Memory System

1. Coherence
2. Consistency

1. Coherence

- It defines what values can be returned by a read.

2. Consistency

- Determines when a written value will be returned by a read

  A memory system is coherent if the following conditions are met.

  1. CPU A writes N to location X, all future reads of location X will return N if no other processor writes location X after the write of CPU A.

NOTE: Due to communication latency, the writes cannot be seen instantaneously

2. Consistency (Cont…)

- Memory system is coherent if
  - A read by a processor P1 to a location X that follows a write by P2 to X always returns the value written by P2, if no other writes to X made by any processor occurring between the 2 accesses
  - Writes to the same location are *serialized*; i.e. two writes to the same location by any two processors are seen in the same order by all processors

Suppose we did not serialize writes, & processors P1 writes location X followed by P2 writing location X.
In this case some processor could see the write of P2, first & then see the write of P1.
To avoid such difficulties is, to serialize writes. So all writes to the same location are seen in the same order, this property is called **write serialization**

Basic Schemes for Enforcing Coherence

- In a coherent multiprocessor, the caches provide both migration and replication of shared data items.
  - **Migration**: Since a data item can be moved to a local cache and used there in a transparent fashion, this reduces the latency to a shared data item that is allocated remotely.
  - **Replication**: Since the caches make a copy of the data item in the local cache, it reduces both latency of access and contention for a read shared data item.
- Cache coherence protocol implemented in hardware to maintain coherent caches.
- Key to implementing a cache-coherence protocol is tracking the state of any sharing of a data block.

Need for protocols ?

- Migration reduces both latency & bandwidth on shared memory
- Replication reduces both latency of access & contention for a read shared data item.
- Both the migration & replication is critical to performance in accessing shared data, so to avoid this, **protocols are introduced**

NOTE: Key to implementing a cache-coherence protocol is tracking the state of any sharing of a data block.

Cache Coherence Protocol

- Key: tracking the state of any sharing of a data block

1. Directory based – the sharing status of a block of physical memory is kept in just one location, the *directory*

Cache Coherence Protocol

2. Snooping
   - Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, and no centralized state is kept
   - Caches are usually on a shared-memory bus, and all cache controllers monitor or snoop on the bus to determine if they have a copy of a block that is requested on the bus

Cache Coherence Protocols

- Coherence enforcement strategy – how caches are kept consistent with the copies stored at servers
  - Write-invalidate: Writer sends invalidation to all caches whenever data is modified
    - Winner
  - Write-update: Writer propagates the update
    - Also called write broadcast

1. Write-Invalidate (Snooping Bus)

- **Definition:** Processor has exclusive access to a data item before it writes that item.

Write-back Cache

Processor activity	Bus activity	Cache Contents for CPU A	Cache Contents for CPU B	Memory contents location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 into X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

In above table, consider a write followed by a read by another processor

Explanation

- The Write requires exclusive access, any copy held by the reading processor must be invalidated.
  - Thus when read occurs, it misses in the cache & is forced to fetch a new copy of the data
- If 2 processors do attempt to write the same data simultaneously, one of them wins the race causing the other processor's copy to be invalidated.
- For the other processor to complete its write, it must obtain a new copy of the data, which must now contain the updated value.
  - Therefore this protocol enforces **Write Serialization**

2. Write-Update (Snooping Bus)

- **Definition:** Updates all the cached copies of a data item when that item is written.

Processor activity	Bus activity	Cache Contents for CPU A	Cache Contents for CPU B	Memory contents location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 into X	Write broadcast of X	1	**1**	**1**
CPU B reads X	Cache miss for X	1	1	1

Write-Update VS. Write-Invalidate

- Multiple writes to the same word with no intervening reads require multiple write broadcast in an update protocol, but only one initial invalidation in a write invalidate protocol
- With multiword cache block, each word written in a cache block requires
  - a write broadcast in an update protocol,
  - although only the first write to any word in the block needs to generate an invalidate in an invalidation protocol.
  - Invalidation protocol works on cache blocks
  - Update protocol works on individual words

Write-Update VS. Write-Invalidate (Cont.)

- Delay between writing a word in one processor and reading the value in another processor is usually less in a write update scheme, since the written data are immediately updated in the reader's cache
  - In an invalidation protocol, the reader is invalidated first, then later reads the data and is stalled until a copy can be read and returned to the processor
- Invalidate protocols generate less bus and memory traffic
- Update protocols cause problems for memory consistency model

Basic Snooping Implementation Techniques

- Use the bus to perform invalidates
  - Acquire bus access and broadcast address to be invalidated on bus
  - All processors continuously snoop on the bus, watching the address
  - Check if the address on the bus is in their cache $\rightarrow$ invalidate

Basic Snooping Implementation Techniques

- Serialization of access enforced by bus also forces serialization of writes
  - First processor to obtain bus access will cause others' copies to be invalidated
- A write to a shared data item cannot complete until it obtains bus access
- Assume **atomic operations**

Basic Snooping Implementation Techniques (Cont.)

- How to locate a data when a cache miss occurs?
  - Write-through cache: memory always has the most updated data
  - Write-back cache
    - Every processor snoops on the bus
    - Has a dirty copy of data → provide it and abort memory access
- HW cache structure for implementing snooping
  - Cache tags, valid bit
  - Shared bit: shared mode or exclusive mode

Conceptual Write-Invalid Protocol with Snooping

- Read hit (valid): reads the data block and continues
- Read miss: does not hold the block or invalid block
  - Transfer the block from the shared memory (write-through, or write-back and clean), or from the copy-holder (write-back dirty)
  - Set the corresponding valid-bit and shared-mode bit
    - The sole holder? → Yes, set the shared-mode bit exclusive
    - If the block before the read is exclusive? Yes →
      - Set the shared-mode bit to shared

Conceptual Write-Invalid Protocol with Snooping (Cont.)

- Write hit (block owner)
  - Write hit to an exclusive cache block → proceeds and continues
  - Write hit to a shared-read-only block → need to obtain permission
    - Invalidate all cache copies
    - Completion of invalidation → write data and set the exclusive bit
      - The processor becomes the sole owner of the cache block until other read accesses arrive from other processors
        - » Can be detected by snooping
        - » Then the block changes to the shared state

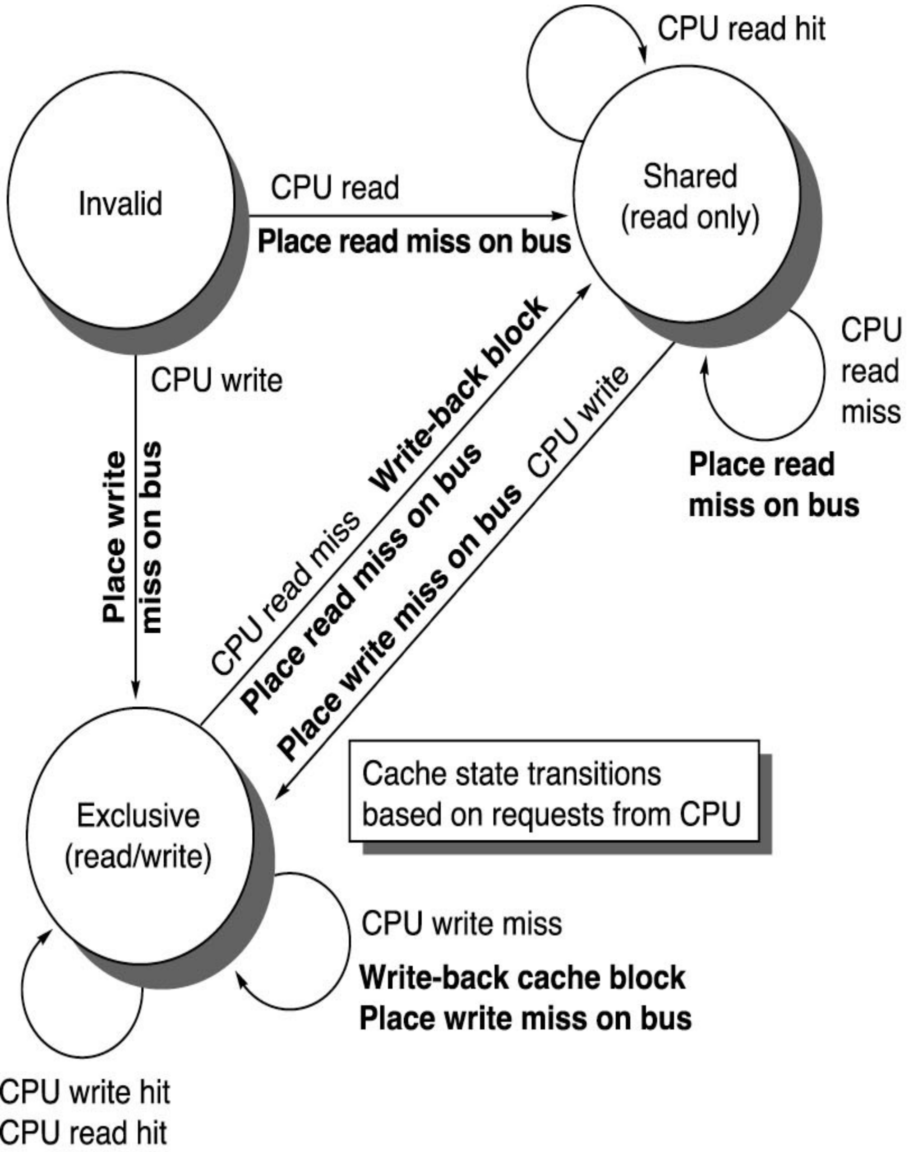
Conceptual Write-Invalid Protocol with Snooping (Cont.)

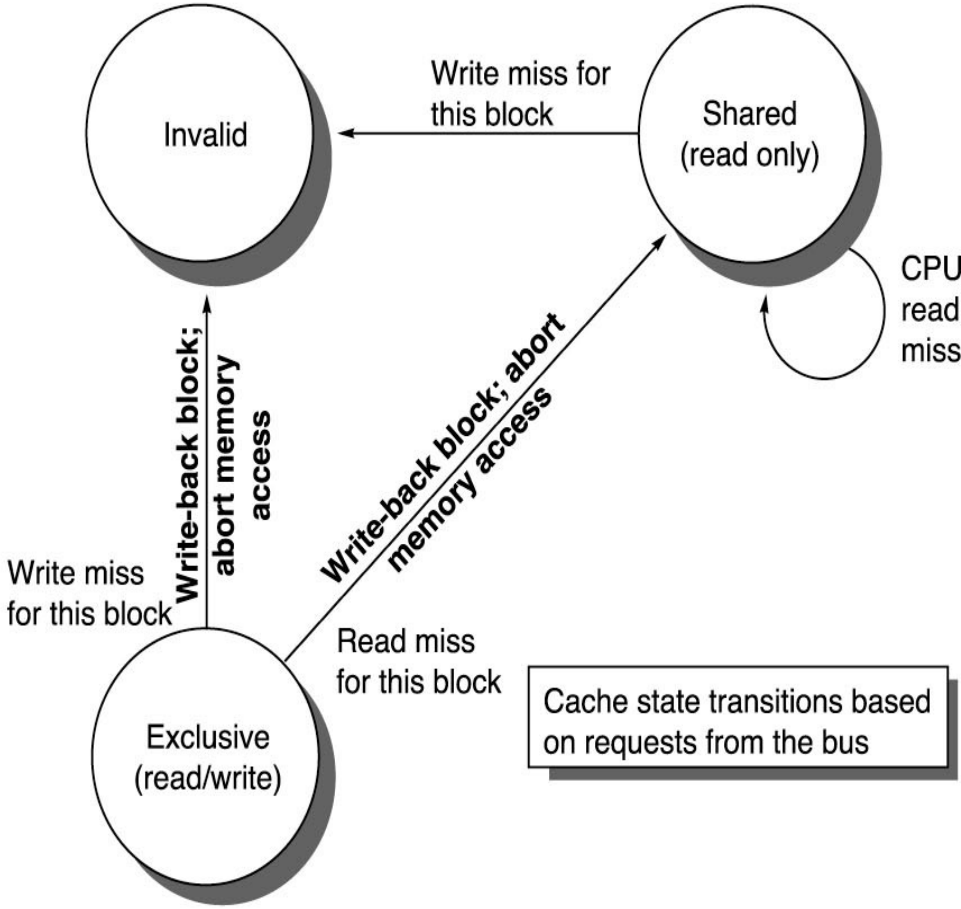
- Write miss: action similar to that of a write hit, except
  - A block copy is transfer to the processor after the invalidation

An Example Snooping Protocol (Cont.)

- Treat write-hit to a shared cache block as write-miss
  - Place write-miss on bus. Any processors with the block → invalidate
  - Reduce no. of different bus transactions and simplifies controller

Write-Invalidate Coherence Protocol for Write-Back Cache

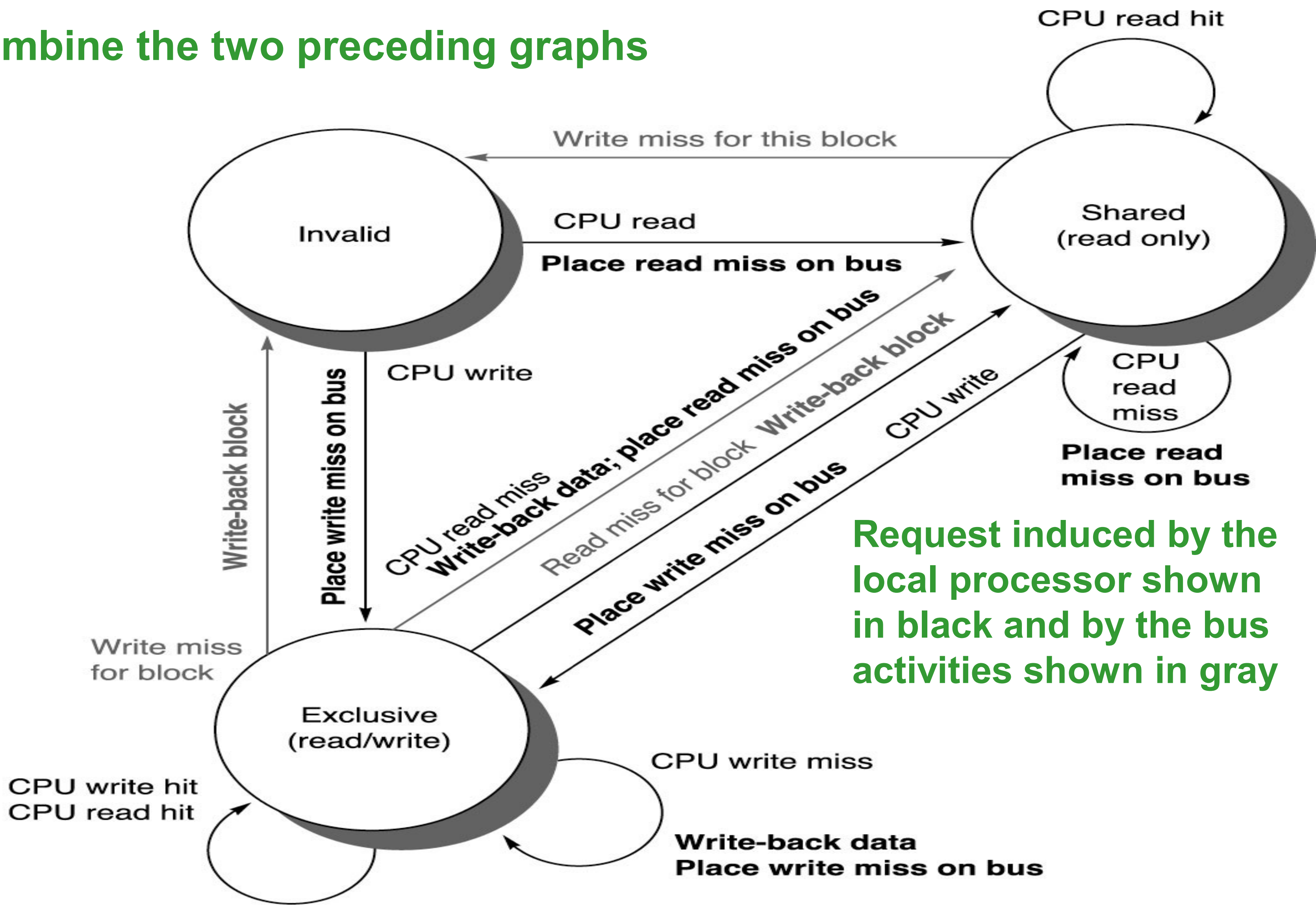




Invalid
Write miss for this block
Shared (read only)
CPU read miss
Write miss for this block
Write-back block; abort memory access
Write-back block; abort memory access
Read miss for this block
Exclusive (read/write)
Cache state transitions based on requests from the bus

Cache Coherence State Diagram

Combine the two preceding graphs



Request induced by the local processor shown in black and by the bus activities shown in gray

Actions Done by versus Requests from the Processor and Bus

Request	Source	State of addressed cache block	Function and Explanation
Read hit	Processor	Shared / Exclusive	Read data in cache
Read miss	Processor	*Invalid*	Place read miss on the bus
Read miss	Processor	Shared	Place read miss on the bus
Read miss	Processor	Exclusive	Write back block
Write hit	Processor	Exclusive	Write data in cache
Write hit	Processor	Shared	Place read miss on the bus
Write miss	Processor	Invalid	Place write miss on the bus

Actions Done by versus Requests from the Processor and Bus

Request	Source	State of addressed cache block	Function and Explanation
Write miss	Processor	Shared	Place write miss on the bus
Write miss	Processor	Exclusive	Write back block then write miss on the bus
Read miss	Bus	Shared	Allow memory to service read miss
Read miss	Bus	Exclusive	Place cache block on bus & change state to shared
Write miss	Bus	Shared	Invalidate the block
Write miss	Bus	Exclusive	Write back the cache block & make its state invalid

State transition diagrams

Finite-State transition diagram for a single cache block using a Write Invalidation protocol and a Write-back cache is discussed further.

- ✓ 3 states of protocol represents transitions based on CPU requests & Bus requests

1. Invalid
2. Shared
3. Exclusive

- ✓ All state changes indicated by arcs

State transition diagrams

Finite-State transition diagram for a single cache block using a **Write Invalidation protocol** and a **Write-back cache**.

How ?

Cache Organization to Support WIP for a Write-Back Cache

- Each cache block consists of
  - ***Valid bit***: Set by invalidation message on the bus.
  - ***Dirty bit***: Set when a block is modified.
  - ***State bits***: Private, shared, or unshared; set to shared when the cache block become shared.

Cache Organization to Support WIP for a Write-Back Cache (Cont…)

- How do these bits work?
  - When a write to a block in the shared state occurs, the cache generates an invalidation on the bus and marks the block as ***private.*** The processor becomes the owner of the cache block. No further invalidations will be sent by that processor for that block.
  - When an invalidation is sent, the state of the owner's cache block is changed from ***shared* to *unshared*.** If another processor later requests this cache block, the state must be made shared again.

An Example Protocol

- In a bus-based coherence protocol, a finite state controller in each node is implemented to
  - Responds to requests from the processor and from the bus,
  - Change the state of the selected cache block, and
  - use the bus to access data or to invalidate it.

An Example Snooping Protocol

- **This controller responds to the request from the processor and from the bus based on:**
  - **the type of the request**
  - **Its hit or miss status in the cache; and**
  - **State of the cache block specified in the request**
- **Furthermore, each block of memory is in one of the three states: <u>Shared</u>, <u>Exclusive</u> or <u>Invalid</u> (Not in any caches) and each cache block tracks these three states**

Example: Working of Finite State Machine Controller

- Today we will continue our discussion on the finite state machine controller for the implementation of snooping protocol;

  and will try to understand its working with the help of example

- Here, we assume that two processors P1 and P2 each having its own cache, share the main memory connected on bus

Example: Working of Finite State Machine Controller

- The status of the processors, bus transaction and the memory is depicted in a table for each step of the state machine
- Here, the state of the machine for each processor and cache address and value cached, the bus action and shared-memory status is shown for each step of operation
- Initially the cache state is invalid (i.e., the block of memory is not in the cache); and …

Example: Working of Finite State Machine Controller

- memory blocks A1 and A2 map to the same cache block where the address A1 is not equal to A2
- At Step 1 – P1 writes 10 to A1

  write miss on bus occurs and the state transition from invalid to exclusive takes place

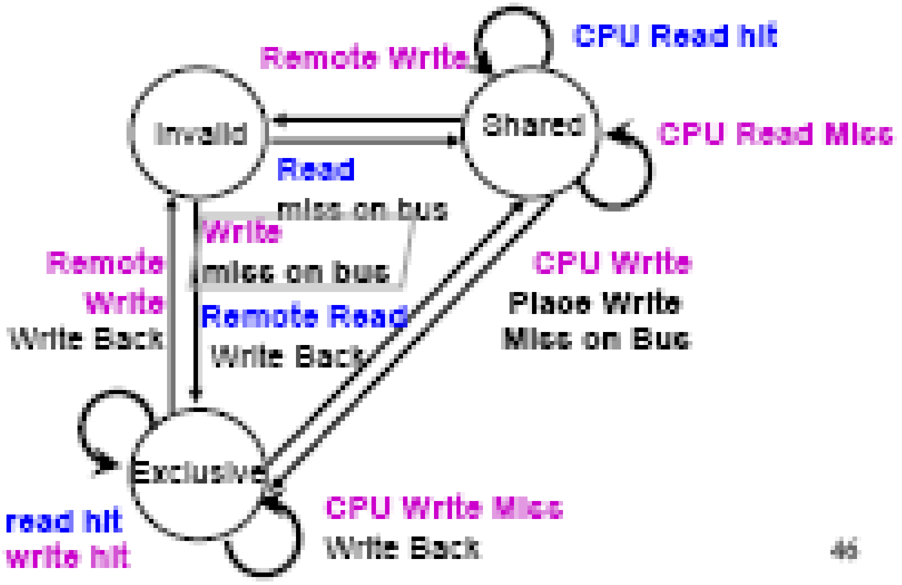
Example: Working of Finite State Machine Controller

Example: Step 1

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.

Active arrow =

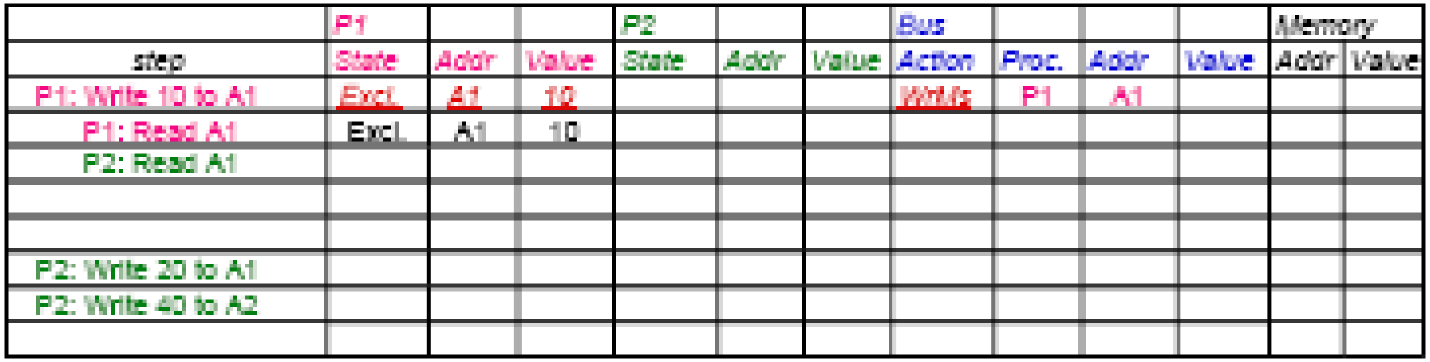


4/3/2006 46

Example: Working of Finite State Machine Controller

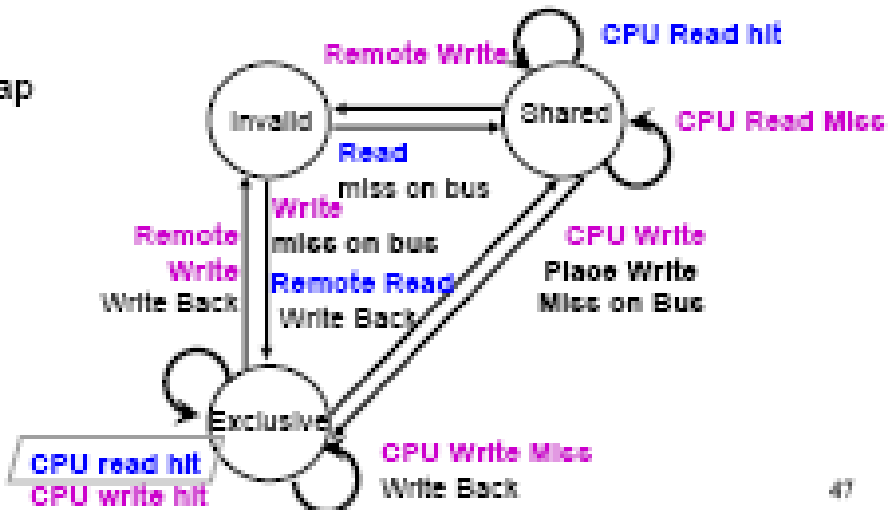
- **At Step 2 – P1 reads A1**

CPU read HITs occurs, hence the FSM Stays in exclusive state



	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



4/3/2006 47

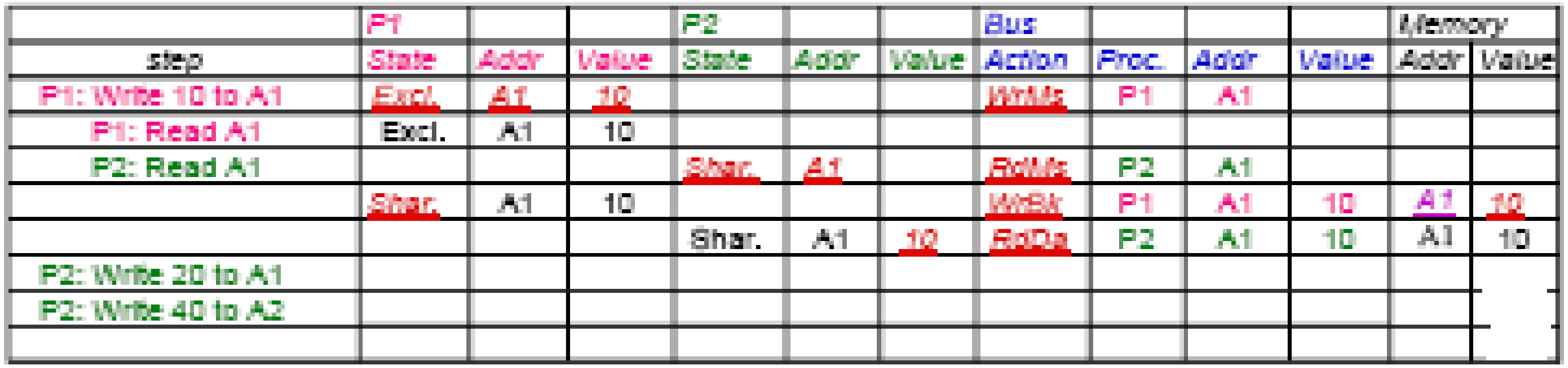
Example: Working of Finite State Machine Controller

- **At Step 3: P2 reads A1**

i) As P2 is initially in invalid state, therefore, read miss on the bus occurs; the controller state changes from invalid to Shared

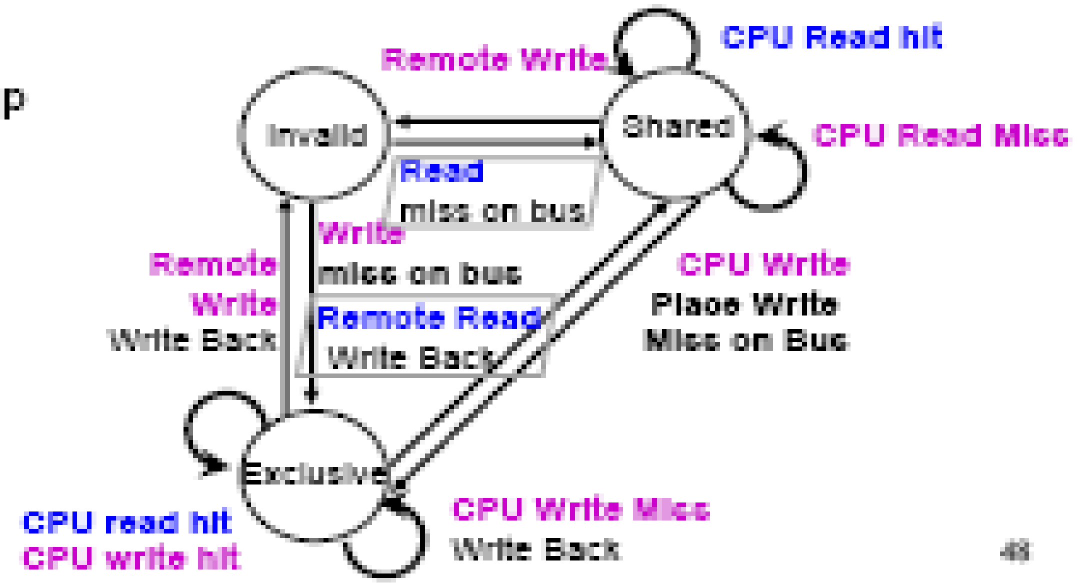
Example: Working of Finite State Machine Controller

Example: Step 3



	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.



4/3/2006

48

Example: Working of Finite State Machine Controller

ii) P1 being in Exclusive state, remote read write-back is asserted and the state changes from exclusive to Shared; and

iii) the value (10) is read 1 from the shared-memory at address A1, into P1 and P2 caches at A1; and both P1 and P2 controllers are in shared state

Example: Working of Finite State Machine Controller

At Step 4: P2 write 20 to A1

i) **P1 find a remote write, so the state of the controller changes from shared to Invalid**

ii) **P2 find a CPU write, so places write miss on the bus and changes the state from shared to exclusive and writes value 20 to A1**

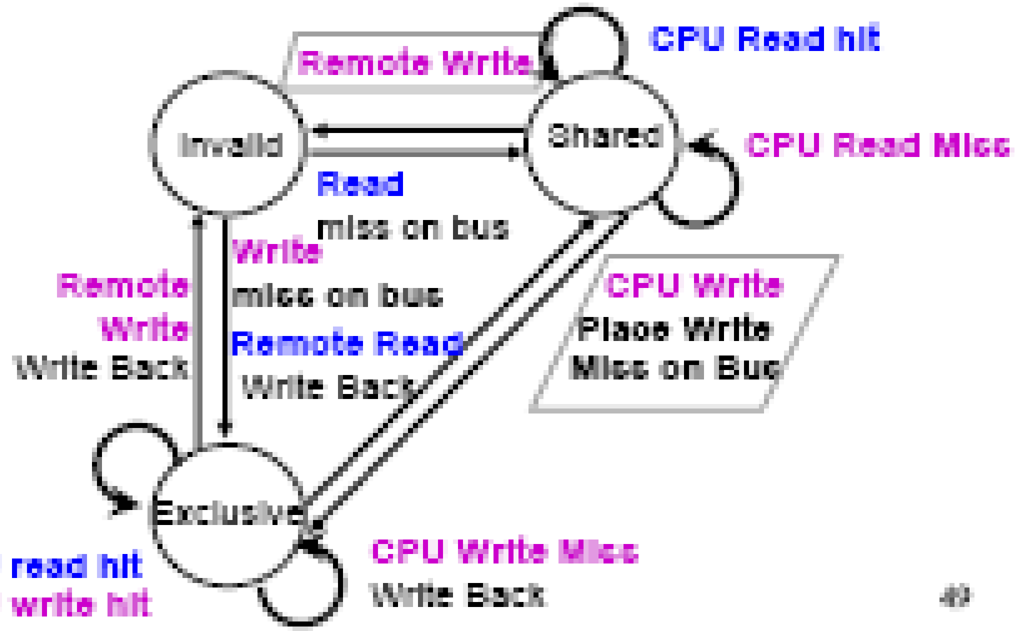
iii) **The memory address to A1 with value A1**

Example: Working of Finite State Machine Controller

Example: Step 4

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2												

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



Example: Working of Finite State Machine Controller

At Step 5: P2 write 40 to A2

i) P2 being in Exclusive state, CPU write Miss occurs, and initiates write-back to P2 at A2

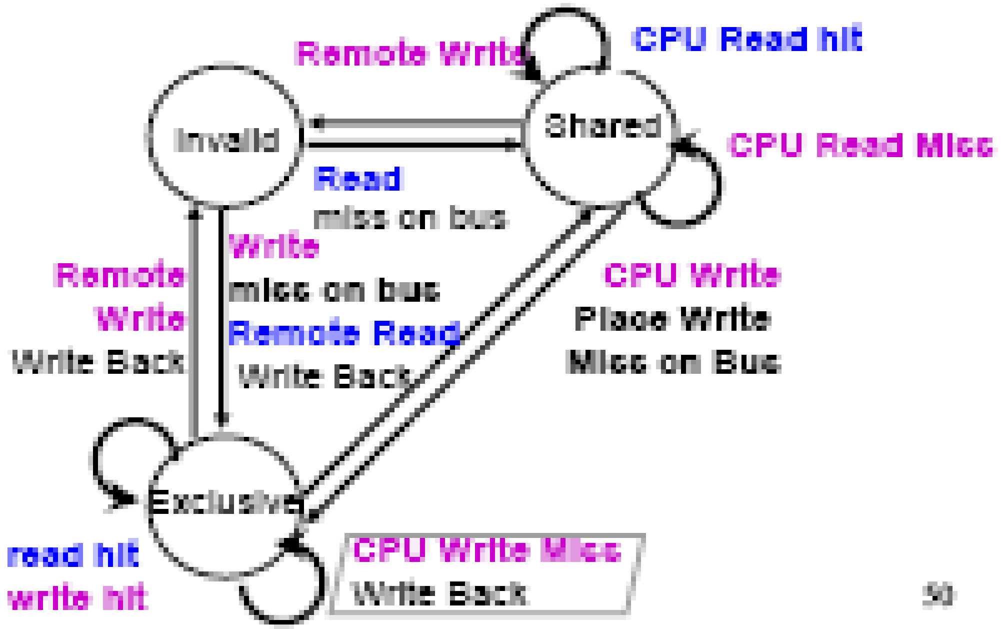
ii) P2 remains in Exclusive state, with address A2 and value 40

Example: Working of Finite State Machine Controller

Example: Step 5

step	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2							WrMs	P2	A2		A1	10
				Excl.	A2	40	WrBk	P2	A1	20	A1	20

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



Implementation Complications

- **With this example, we have observed that the finite state machine implementation of the snooping protocols works well**
- **However, the following implementation complications have been observed**
- **Write Races**
- **Interventions and invalidations**

Implementation Complications

- Write Races occur when one processor wants to update the cache but another processor may get bus first and then write the same cache block!
- We know that bus transaction is a two step process:
  - Arbitrate for bus
  - Place miss on bus and complete operation
  - If miss occurs to block while waiting for bus, handle miss, i.e., invalidate, and then restart.

Implementation Complications

- Furthermore, to overcome the write races, split transaction bus, so that
  - it can have multiple outstanding transactions for a block
  - Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
  - Must track and prevent multiple misses for one block

Processor 1 Processor 2 Interconnect Directory Memory

	P1			*P2*			*Bus*				*Directory*			*Memor*
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*State*	*{Procs}*	*Value*
P1: Write 10 to A1														
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			*P2*			*Bus*				*Directory*			*Memor*
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*State*	*{Procs}*	*Value*
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1			10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1														10
														10
P2: Write 40 to A2														10

Write Back

A1 and A2 map to the same cache block

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1		10	
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2														10

A1 and A2 map to the same cache block

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1		10	
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2							WrMs	P2	A2		A2	Excl.	{P2}	0
							WrBk	P2	A1	20	A1	Unca.	{}	20
				Excl.	A2	40	DaRp	P2	A2	0	A2	Excl.	{P2}	0

A1 and A2 map to the same cache block