

---

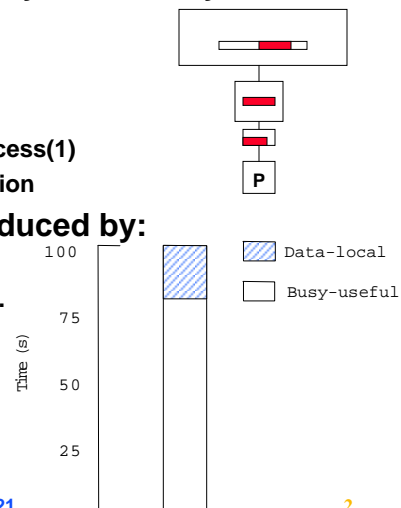
## Shared Memory Multiprocessors

CS 252, Spring 2002  
David E. Culler  
Computer Science Division  
U.C. Berkeley

### Uniprocessor View

---

- Performance depends heavily on memory hierarchy
- Managed by hardware
- Time spent by a program
  - $\text{Timeprog}(1) = \text{Busy}(1) + \text{Data Access}(1)$
  - Divide by cycles to get CPI equation
- Data access time can be reduced by:
  - Optimizing machine
    - » bigger caches, lower latency...
  - Optimizing program
    - » temporal and spatial locality

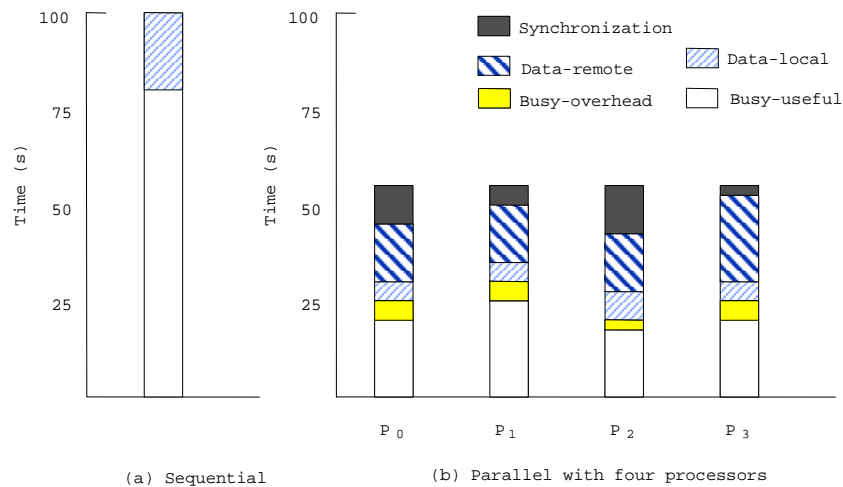


4/11/02

CS252 S02.21

2

## Same Processor-Centric Perspective



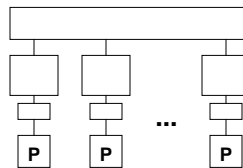
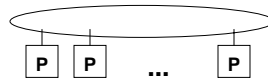
4/11/02

CS252 S02.21

3

## What is a Multiprocessor?

- **A collection of communicating processors**
  - Goals: balance load, reduce inherent communication and extra work
- **A multi-cache, multi-memory system**
  - Role of these components essential regardless of programming model
  - Prog. model and comm. abstr. affect specific performance tradeoffs

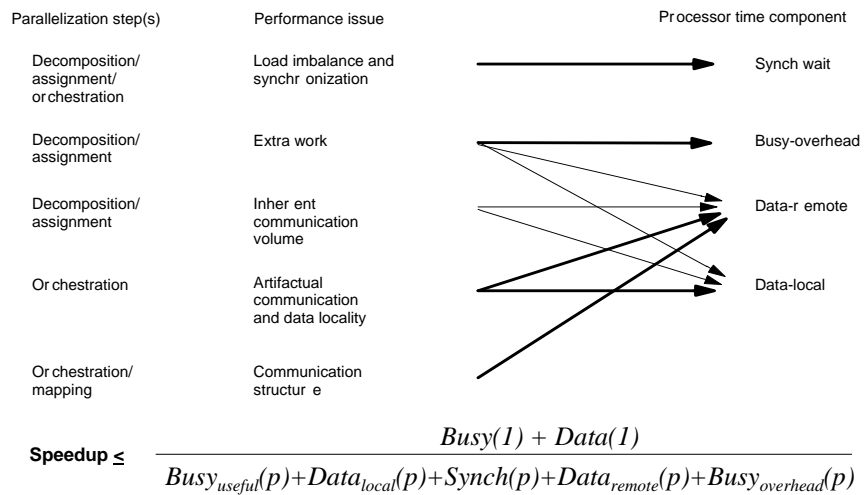


4/11/02

CS252 S02.21

4

## Relationship between Perspectives



4/11/02

CS252 S02.21

5

## Back to Basics

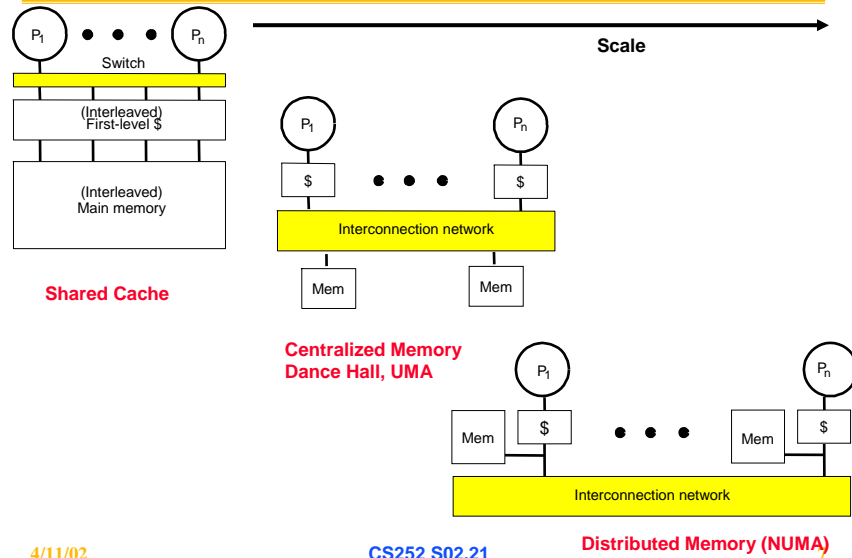
- **Parallel Architecture = Computer Architecture + Communication Architecture**
- **Small-scale shared memory**
  - extend the memory system to support multiple processors
  - good for multiprogramming throughput and parallel computing
  - allows *fine-grain sharing* of resources
- **Naming & synchronization**
  - communication is implicit in store/load of shared address
  - synchronization is performed by operations on shared addresses
- **Latency & Bandwidth**
  - utilize the normal migration within the storage to avoid long latency operations and to reduce bandwidth
  - economical medium with fundamental BW limit
  - => focus on eliminating unnecessary traffic

4/11/02

CS252 S02.21

6

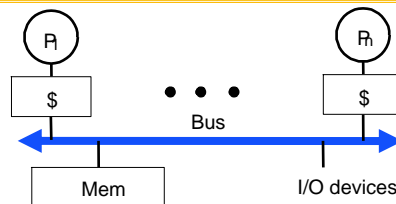
## Natural Extensions of Memory System



4/11/02

CS252 S02.21

## Bus-Based Symmetric Shared Memory



- **Dominate the server market**
  - Building blocks for larger systems; arriving to desktop
- **Attractive as throughput servers and for parallel programs**
  - Fine-grain resource sharing
  - Uniform access via loads/stores
  - Automatic data movement and coherent replication in caches
  - Cheap and powerful extension
- **Normal uniprocessor mechanisms to access data**
  - Key is extension of memory hierarchy to support multiple processors

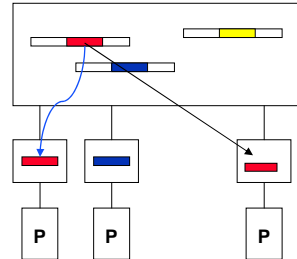
4/11/02

CS252 S02.21

8

## Caches are Critical for Performance

- **Reduce average latency**
  - automatic replication closer to processor
- **Reduce average bandwidth**
- **Data is logically transferred from producer to consumer to memory**
  - store reg --> mem
  - load reg <-- mem
- **Many processors can shared data efficiently**
- **What happens when store & load are executed on different processors?**

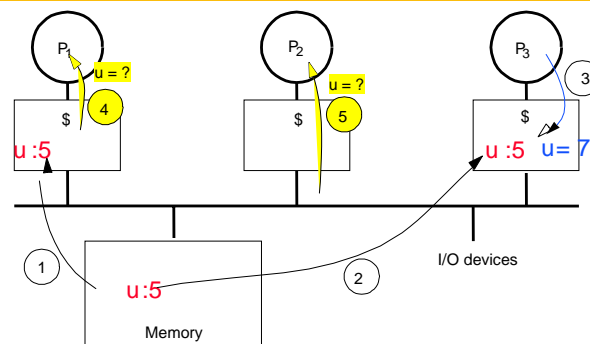


4/11/02

CS252 S02.21

9

## Example Cache Coherence Problem



- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  - » Processes accessing main memory may see very stale value
- Unacceptable to programs, and frequent!

4/11/02

CS252 S02.21

10

## Caches and Cache Coherence

- **Caches play key role in all cases**
  - Reduce average data access time
  - Reduce bandwidth demands placed on shared interconnect
- **private processor caches create a problem**
  - Copies of a variable can be present in multiple caches
  - A write by one processor may not become visible to others
    - » They'll keep accessing stale value in their caches

**=> 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

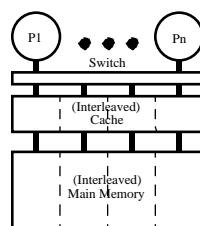
4/11/02

CS252 S02.21

11

## Shared Cache: Examples

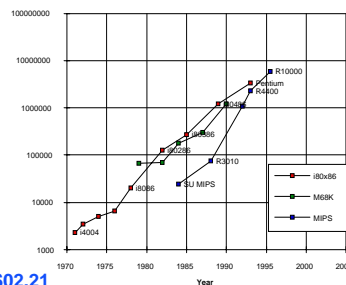
- **Alliant FX-8**
  - early 80's
  - eight 68020s with x-bar to 512 KB interleaved cache
- **Encore & Sequent**
  - first 32-bit micros (N32032)
  - two to a board with a shared cache
- **coming soon to microprocessors near you...**



4/11/02

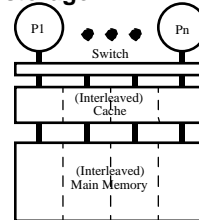
CS252 S02.21

12



## Advantages

- **Cache placement identical to single cache**
  - only one copy of any cached block
- **fine-grain sharing**
  - communication latency determined level in the storage hierarchy where the access paths meet
    - » 2-10 cycles
    - » Cray Xmp has shared registers!
- **Potential for positive interference**
  - one proc prefetches data for another
- **Smaller total storage**
  - only one copy of code/data used by both proc.
- **Can share data within a line without “ping-pong”**
  - long lines without false sharing



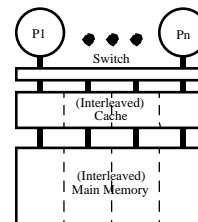
4/11/02

CS252 S02.21

13

## Disadvantages

- **Fundamental BW limitation**
- **Increases latency of all accesses**
  - X-bar
  - Larger cache
  - L1 hit time determines proc. cycle time !!!
- **Potential for negative interference**
  - one proc flushes data needed by another
- **Many L2 caches are shared today**

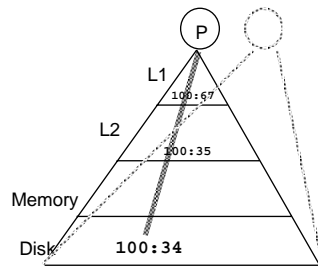


4/11/02

CS252 S02.21

14

## Intuitive Memory Model



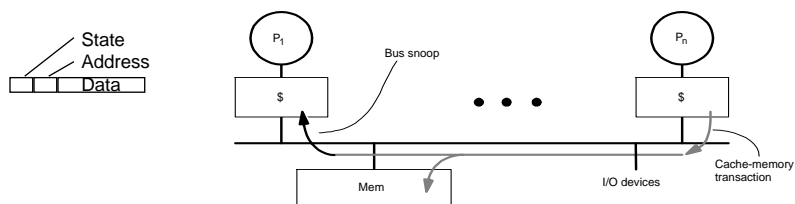
- Reading an address should **return the last value written** to that address
- Easy in uniprocessors
  - except for I/O
- Cache coherence problem in MPs is more pervasive and more performance critical

4/11/02

CS252 S02.21

15

## Snoopy Cache-Coherence Protocols



- Bus is a broadcast medium & Caches know what they have
- Cache Controller “snoops” all transactions on the shared bus
  - relevant transaction if for a block it contains
  - take action to ensure coherence
    - » invalidate, update, or supply value
  - depends on state of the block and the protocol

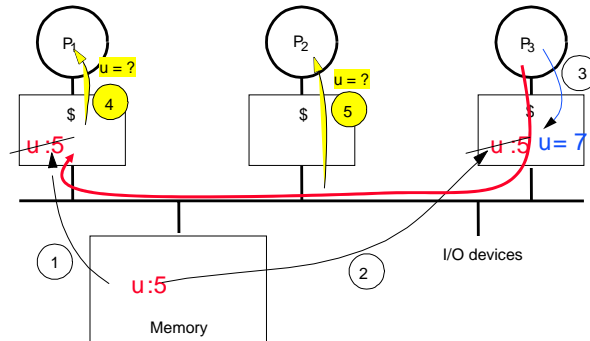
4/11/02

CS252 S02.21

16



## Example: Write-thru Invalidate



4/11/02

CS252 S02.21

17

## Architectural Building Blocks

- **Bus Transactions**
  - fundamental system design abstraction
  - single set of wires connect several devices
  - bus protocol: arbitration, command/addr, data
  - => Every device observes every transaction
- **Cache block state transition diagram**
  - FSM specifying how disposition of block changes
    - » invalid, valid, dirty

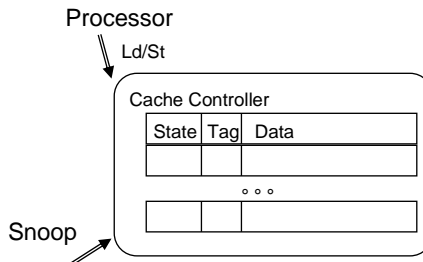
4/11/02

CS252 S02.21

18

## Design Choices

- Controller updates state of blocks in response to processor and snoop events and generates bus transactions
- **Snoopy protocol**
  - set of states
  - state-transition diagram
  - actions
- **Basic Choices**
  - Write-through vs Write-back
  - Invalidate vs. Update



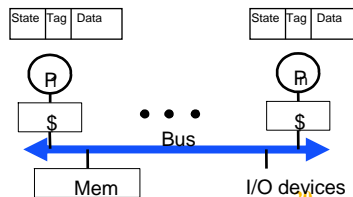
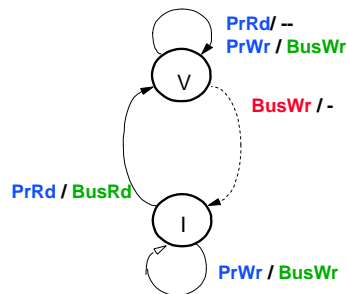
4/11/02

CS252 S02.21

19

## Write-through Invalidate Protocol

- Two states per block in each cache
  - as in uniprocessor
  - state of a block is a *p*-vector of states
  - Hardware state bits associated with blocks that are in the cache
  - other blocks can be seen as being in invalid (not-present) state in that cache
- **Writes invalidate all other caches**
  - can have multiple simultaneous readers of block, but write invalidates them



4/11/02

CS252 S02.21

20

## Write-through vs. Write-back

---

- **Write-through protocol is simple**
  - every write is observable
- **Every write goes on the bus**
  - => Only one write can take place at a time in any processor
- **Uses a lot of bandwidth!**

Example: 200 MHz dual issue, CPI = 1, 15% stores of 8 bytes

=> 30 M stores per second per processor

=> 240 MB/s per processor

1GB/s bus can support only about 4  
processors without saturating

4/11/02

CS252 S02.21

21

## Invalidate vs. Update

---

- **Basic question of program behavior:**
    - Is a block written by one processor later read by others before it is overwritten?
  - **Invalidate.**
    - yes: readers will take a miss
    - no: multiple writes without addition traffic
      - » also clears out copies that will never be used again
  - **Update.**
    - yes: avoids misses on later references
    - no: multiple useless updates
      - » even to pack rats
- => Need to look at program reference patterns and hardware complexity

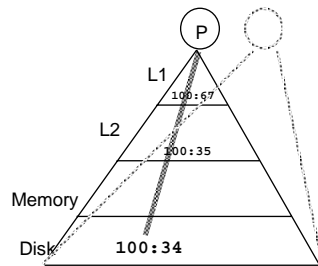
but first - correctness

4/11/02

CS252 S02.21

22

## Intuitive Memory Model???



- Reading an address should **return the last value written** to that address
- What does that mean in a multiprocessor?

4/11/02

CS252 S02.21

23

## Coherence?

- Caches are supposed to be transparent
- What would happen if there were no caches
- Every memory operation would go “to the memory location”
  - may have multiple memory banks
  - all operations on a particular location would be serialized
    - » all would see THE order
- Interleaving among accesses from different processors
  - within individual processor => program order
  - across processors => only constrained by explicit synchronization
- **Processor only observes state of memory system by issuing memory operations!**

4/11/02

CS252 S02.21

24

## Definitions

---

- **Memory operation**
  - load, store, read-modify-write
- **Issues**
  - leaves processor's internal environment and is presented to the memory subsystem (caches, buffers, busses, dram, etc)
- **Performed with respect to a processor**
  - write: subsequent reads return the value
  - read: subsequent writes cannot affect the value
- **Coherent Memory System**
  - there exists a serial order of mem operations on each location s. t.
    - » operations issued by a process appear in order issued
    - » value returned by each read is that written by previous write in the serial order

4/11/02 => write propagation + write serialization CS252 S02.21

25

## Is 2-state Protocol Coherent?

---

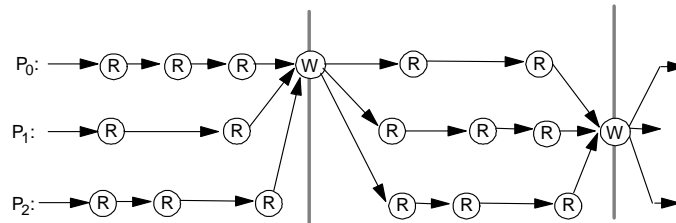
- **Assume bus transactions and memory operations are atomic, one-level cache**
  - all phases of one bus transaction complete before next one starts
  - processor waits for memory operation to complete before issuing next
  - with one-level cache, assume invalidations applied during bus xaction
- **All writes go to bus + atomicity**
  - **Writes serialized** by order in which they appear on bus (bus order)
  - => invalidations applied to caches in bus order
- **How to insert reads in this order?**
  - Important since processors see writes through reads, so determines whether write serialization is satisfied
  - But read hits may happen independently and do not appear on bus or enter directly in bus order

4/11/02

CS252 S02.21

26

## Ordering



- **Writes establish a partial order**
- **Doesn't constrain ordering of reads, though bus will order read misses too**
  - any order among reads between writes is fine, as long as in program order

4/11/02

CS252 S02.21

27

## Write-Through vs Write-Back

- **Write-thru requires high bandwidth**
  - **Write-back caches absorb most writes as cache hits**
- => Write hits don't go on bus**
- But now how do we ensure write propagation and serialization?
  - Need more sophisticated protocols: large design space
- **But first, let's understand other ordering issues**

4/11/02

CS252 S02.21

28

## Setup for Mem. Consistency

- Coherence => Writes to a location become visible to all in the same order
- But when does a write become visible?
- How do we establish orders between a write and a read by different procs?
  - use event synchronization
- typically use more than one location!

4/11/02

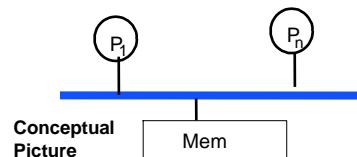
CS252 S02.21

29

## Example

$P_1$	$P_2$
<hr/>	
/*Assume initial value of A and ag is 0*/	
A = 1;	while (flag == 0); /*spin idly*/
flag = 1;	print A;

- Intuition not guaranteed by coherence
- expect memory to respect order between accesses to *different* locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Coherence is not enough!
  - pertains only to single location



4/11/02

CS252 S02.21

30

## Memory Consistency Model

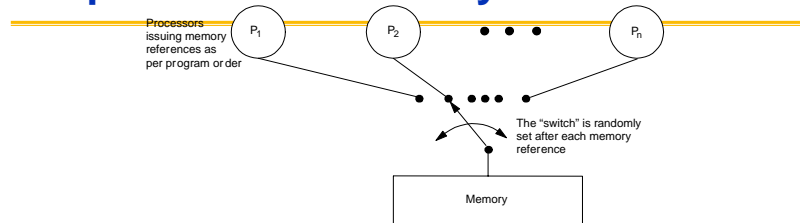
- Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to one another
  - What orders are preserved?
  - Given a load, constrains the possible values returned by it
- Without it, can't tell much about an SAS program's execution
- Implications for both programmer and system designer
  - Programmer uses to reason about correctness and possible results
  - System designer can use to constrain how much accesses can be reordered by compiler or hardware
- Contract between programmer and system

4/11/02

CS252 S02.21

31

## Sequential Consistency



- Total order achieved by *interleaving* accesses from different processes
  - Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others
  - as if there were no caches, and a single memory
- "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

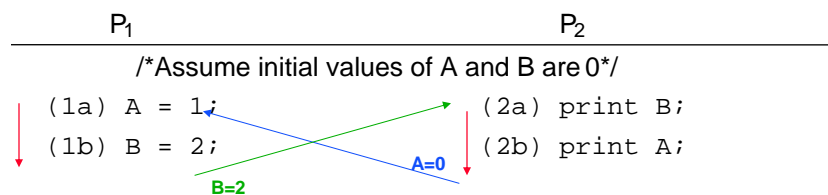
4/11/02

CS252 S02.21

32



## SC Example



- What matters is order in which operations *appear to execute*, not the chronological order of events
- Possible outcomes for (A,B): (0,0), (1,0), (1,2)
- What about (0,2) ?
  - program order  $\Rightarrow$  1a  $\rightarrow$  1b and 2a  $\rightarrow$  2b
  - A = 0 implies 2b  $\rightarrow$  1a, which implies 2a  $\rightarrow$  1b
  - B = 2 implies 1b  $\rightarrow$  2a, which leads to a contradiction

4/11/02

CS252 S02.21

33

## Implementing SC

- Two kinds of requirements
  - Program order
    - » memory operations issued by a process must appear to execute (become visible to others and itself) in program order
  - Atomicity
    - » in the overall hypothetical total order, one memory operation should appear to complete with respect to all processes before the next one is issued
    - » guarantees that total order is consistent across processes
  - tricky part is making writes atomic

4/11/02

CS252 S02.21

34

## An Example Snoopy Protocol

- Invalidation protocol, write-back cache
- Each block of memory is in one state:
  - Clean in all caches and up-to-date in memory (**Shared**)
  - OR Dirty in exactly one cache (**Exclusive**)
  - OR Not in any caches
- Each cache block is in one state (track these):
  - **Shared** : block can be read
  - OR **Exclusive** : cache has only copy, its writeable, and dirty
  - OR **Invalid** : block contains no data
- Read misses: cause all caches to snoop bus
- Writes to clean line are treated as misses

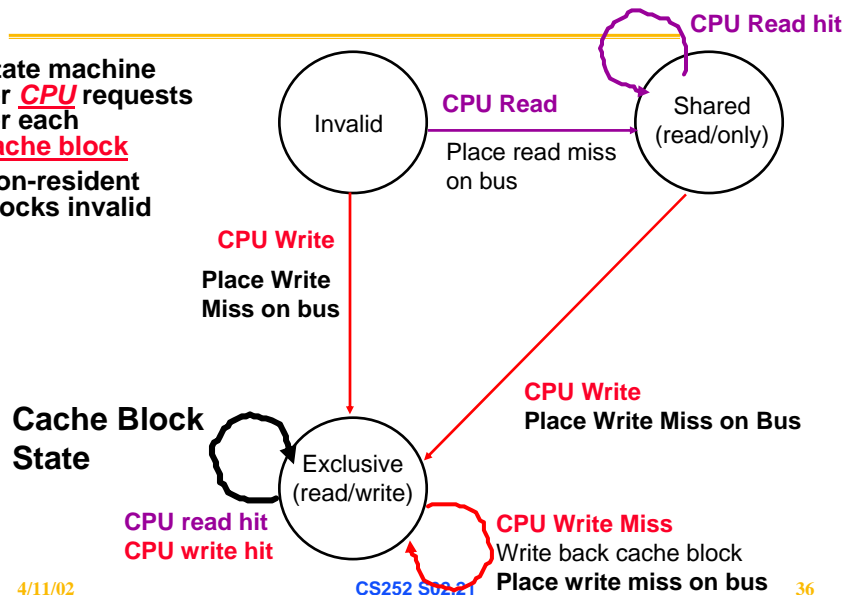
4/11/02

CS252 S02.21

35

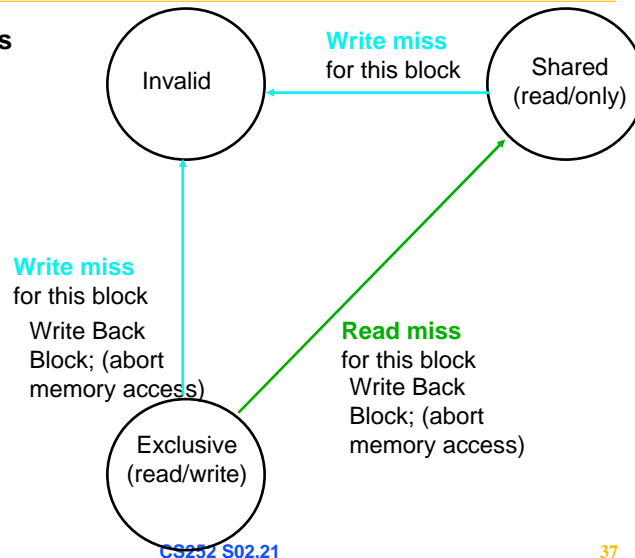
## Write-Back State Machine - CPU

- State machine for **CPU** requests for each **cache block**
- Non-resident blocks invalid



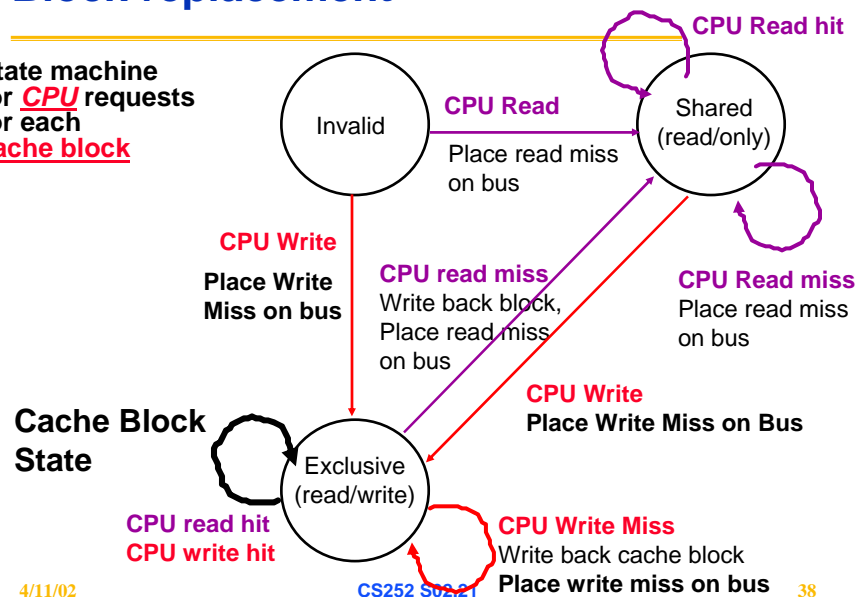
## Write-Back State Machine- Bus req

- State machine for bus requests for each cache block



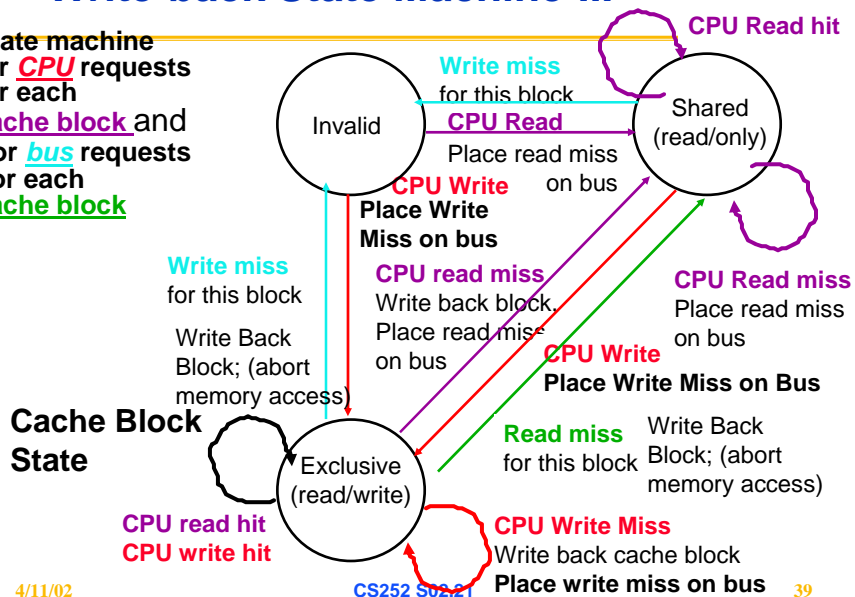
## Block-replacement

- State machine for CPU requests for each cache block



## Write-back State Machine-III

- State machine for **CPU** requests for each **cache block** and for **bus** requests for each **cache block**



## Example

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

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

4/11/02

CS252 S02.21

40

## Example

step	P1			P2			Bus				Memory	
	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 A1 and A2 map to same cache block

4/11/02

CS252 S02.21

41

## Example

step	P1			P2			Bus				Memory	
	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 A1 and A2 map to same cache block

4/11/02

CS252 S02.21

42

## Example

step	P1 State	Addr	Value	P2 State	Addr	Value	Bus Action	Proc.	Addr	Value	Memory 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 A1 and A2 map to same cache block

4/11/02

CS252 S02.21

43

## Example

step	P1 State	Addr	Value	P2 State	Addr	Value	Bus Action	Proc.	Addr	Value	Memory 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			
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

4/11/02

CS252 S02.21

44

## Example

step	P1			P2			Bus				Memory	
	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	10	Shar.	A1		RdMs	P2	A1			
							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 A1 and A2 map to same cache block,  
but A1 != A2

4/11/02

CS252 S02.21

45

## Snooping Cache Variations

Basic Protocol	Berkeley Protocol	Illinois Protocol	MESI Protocol
Exclusive	Owned Exclusive	Private Dirty	Modified (private, != Memory)
Shared	Owned Shared	Private Clean	exclusive (private, = Memory)
Invalid	Shared	Shared	Shared (shared, = Memory)
	Invalid	Invalid	Invalid

Owner can update via bus invalidate operation  
Owner must write back when replaced in cache

If read sourced from memory, then Private Clean  
if read sourced from other cache, then Shared  
Can write in cache if held private clean or dirty

4/11/02

CS252 S02.21

46

## Implementation Complications

- **Write Races:**
  - Cannot update cache until bus is obtained
    - » Otherwise, another processor may get bus first, and then write the same cache block!
  - Two step process:
    - » Arbitrate for bus
    - » Place miss on bus and complete operation
  - If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.
  - Split transaction bus:
    - » Bus transaction is not atomic: 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
- **Must support interventions and invalidations**

4/11/02

CS252 S02.21

47

## Implementing Snooping Caches

- **Multiple processors must be on bus, access to both addresses and data**
- **Add a few new commands to perform coherency, in addition to read and write**
- **Processors continuously snoop on address bus**
  - If address matches tag, either invalidate or update
- **Since every bus transaction checks cache tags, could interfere with CPU just to check:**
  - solution 1: **duplicate set of tags for L1 caches** just to allow checks in parallel with CPU
  - solution 2: L2 cache already duplicate, **provided L2 obeys inclusion** with L1 cache
    - » block size, associativity of L2 affects L1

4/11/02

CS252 S02.21

48



## Implementing Snooping Caches

---

- Bus serializes writes, getting bus ensures no one else can perform memory operation
- On a miss in a write back cache, may have the desired copy and its dirty, so must reply
- Add extra state bit to cache to determine shared or not
- Add 4th state (MESI)

4/11/02

CS252 S02.21

49

## Artifactual Communication

---

- **Accesses not satisfied in local portion of memory hierarchy cause “communication”**
  - Inherent communication, implicit or explicit, causes transfers
    - » determined by program
  - Artifactual communication
    - » determined by program implementation and arch. interactions
    - » poor allocation of data across distributed memories
    - » unnecessary data in a transfer
    - » unnecessary transfers due to system granularities
    - » redundant communication of data
    - » finite replication capacity (in cache or main memory)
  - Inherent communication is what occurs with unlimited capacity, small transfers, and perfect knowledge of what is needed.

4/11/02

CS252 S02.21

50

## Fundamental Issues

---

- 3 Issues to characterize parallel machines
  - 1) **Naming**
  - 2) **Synchronization**
  - 3) **Performance: Latency and Bandwidth (covered earlier)**

4/11/02

CS252 S02.21

51

## Fundamental Issue #1: Naming

---

- **Naming:**
  - what data is shared
  - how it is addressed
  - what operations can access data
  - how processes refer to each other
- Choice of naming affects code produced by a compiler; via load where just remember address or keep track of processor number and local virtual address for msg. passing
- Choice of naming affects replication of data; via load in cache memory hierarchy or via SW replication and consistency

4/11/02

CS252 S02.21

52

## Fundamental Issue #1: Naming

---

- **Global physical address space:**  
any processor can generate, address and access it in a single operation
  - memory can be anywhere:  
virtual addr. translation handles it
- **Global virtual address space:** if the address space of each process can be configured to contain all shared data of the parallel program
- **Segmented shared address space:**  
locations are named  
<process number, address>  
uniformly for all processes of the parallel program

4/11/02

CS252 S02.21

53

## Fundamental Issue #2: Synchronization

---

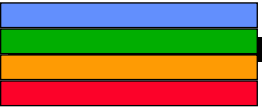
- **To cooperate, processes must coordinate**
- Message passing is implicit coordination with transmission or arrival of data
- Shared address  
=> additional operations to explicitly coordinate:  
e.g., write a flag, awaken a thread, interrupt a processor

4/11/02

CS252 S02.21

54

## Summary: Parallel Framework

- 
- **Layers:**
    - Programming Model:
      - » **Multiprogramming** : lots of jobs, no communication
      - » **Shared address space**: communicate via memory
      - » **Message passing**: send and receive messages
      - » **Data Parallel**: several agents operate on several data sets simultaneously and then exchange information globally and simultaneously (shared or message passing)
    - Communication Abstraction:
      - » **Shared address space**: e.g., load, store, atomic swap
      - » **Message passing**: e.g., send, receive library calls
      - » Debate over this topic (ease of programming, scaling)  
=> many hardware designs 1:1 programming model

4/11/02

CS252 S02.21

55